

H2020 – ICT-13-2018-2019



**Machine Learning to Augment Shared Knowledge in
Federated Privacy-Preserving Scenarios (MUSKETEER)**

Grant No 824988

**D4.6 Machine Learning Algorithms over
Semi Honest Operation Modes algorithms**

– Initial Version

January 20

Imprint

Contractual Date of Delivery to the EC: 31 Jan 2020

Author(s): Ángel Navia-Vázquez (UC3M), Francisco González-Serrano (UC3M)

Participant(s): Jesús Cid Sueiro (UC3M), Manuel Vázquez López (UC3M)

Reviewer(s): Mathieu Sinn
Chiara Napione

Project: Machine learning to augment shared knowledge in federated privacy-preserving scenarios (MUSKETEER)

Work package: WP4

Dissemination level: Internal

Version: 1.0

Contact: angel.navia@uc3m.es

Website: www.MUSKETEER.eu

Legal disclaimer

The project Machine Learning to Augment Shared Knowledge in Federated Privacy-Preserving Scenarios (MUSKETEER) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 824988. The sole responsibility for the content of this publication lies with the authors.

Copyright

© MUSKETEER Consortium. Copies of this publication – also of extracts thereof – may only be made with reference to the publisher.

Executive Summary

This deliverable (D4.6 Machine Learning Algorithms over Semi Honest Operation Modes – Initial Version) comprises the Machine Learning Library needed to execute the distributed learning under POMs 4, 5 and 6, as well as some demonstration scripts to check the correct execution of the code. The list of available algorithms is as compromised for this deliverable (Linear models, Kernel methods and Clustering). The full collection of training methods will be available in the final version of the library (D4.7, M30). The design of the new models will be analogous to the already available ones, so the integration and use will not be a problem. We also provide the software documentation and description of the software components. In future versions, more learning algorithms will be available and some redesign may be necessary to facilitate the integration with the rest of MUSKETEER components, as well as some code and algorithm optimization.

Document History

Version	Date	Status	Author	Comment
1	08 Jan 2020	For internal review	Angel Navia-Vázquez	First draft
2	14 Jan 2020	Internal review	Chiara Napione	
3	16 Jan 2020	Internal review	Angel Navia-Vázquez	
4	16 Jan 2020	Final review	Mathieu Sinn	
5	17 Jan 2020	Final review	Gal Weiss	Final

Table of Contents

LIST OF FIGURES.....	5
LIST OF ACRONYMS AND ABBREVIATIONS.....	7
1 INTRODUCTION.....	8
1.1 Purpose	8
1.2 Related Documents.....	8
1.3 Document Structure.....	9
2 CONTEXT OF THE MACHINE LEARNING LIBRARY.....	10
3 POMS 4, 5 AND 6 REVISITED	12
3.1 POM 4.....	12
3.2 POM 5.....	13
3.3 POM 6.....	15
4 METHODOLOGY	16
4.1 General development process	16
4.2 Current status of the library and future steps	16
5 LIBRARY DEMONSTRATION PRELIMINARY ASSUMPTIONS.....	19
6 MUSKETEER MACHINE LEARNING LIBRARY USAGE.....	23
6.1 Communications setup.....	23
6.2 Setting up the Worker Node (end user side)	23
6.3 Setting up the Master Node.....	25
7 MUSKETEER MACHINE LEARNING LIBRARY RESULTS	26
7.1 Cross-Correlation (XC) estimation	27
7.2 Ridge Regression (RR) estimation	28
7.3 Kernel Regression (KR) estimation	29

7.4	Logistic Classifier (LC)	30
7.5	Multiclass Logistic Classifier (MLC)	32
7.6	Budget Support Vector Machine (BSVM)	35
7.7	Clustering (K-means)	37
8	INSTALLING THE LIBRARY	38
9	EXECUTION OF THE DEMOS	39
9.1	Simple execution	40
9.2	Full detail execution	41
10	SOFTWARE DOCUMENTATION (SAMPLE)	44
11	CONCLUSIONS	48
12	REFERENCES	48

List of Figures

Figure 1: MUSKETEER’s PERT diagram	8
Figure 2: Centralized (a) vs. distributed scenario (b). Every user provides a portion of the training dataset. Data confidentiality must be preserved.	10
Figure 3: Detailed process interactions in a MUSKETEER learning process.....	11
Figure 4: POM 4 general setup.....	12
Figure 5: POM 5 general setup.....	14
Figure 6: POM 6 general setup.....	15
Figure 7: Normalized cross-correlation values among inputs for the redwine dataset.	27
Figure 8: Normalized cross-correlation values between inputs and outputs for the redwine dataset.....	28
Figure 9: Illustration of the target and predicted values using the Ridge Regression estimation for the redwine dataset.	29
Figure 10: Results of the Kernel Regression model on the 1D synthetic dataset.....	30
Figure 11: ROC curves for the Logistic Classifier model on the pima dataset.	31
Figure 12: ROC curves for the Logistic Classifier model on the binarized MNIST handwritten digits dataset.	31
Figure 13: ROC curves for the Multiclass Logistic Classifier model on the MNIST handwritten digits dataset. One ROC curve is shown here for every class value, under a one-vs-all approach.....	32
Figure 14: Confusion matrix for the Multiclass Logistic Classifier model on the MNIST handwritten digits dataset.	33
Figure 15: ROC curves for the Multiclass Logistic Classifier model on the MNIST fashion dataset. One ROC curve is shown here for every class value, under a one-vs-all approach...	34
Figure 16: Confusion matrix for the Multiclass Logistic Classifier model on the MNIST fashion dataset.....	34
Figure 17: ROC curves for the Budget Support Vector Machine model on the 2D synthetic dataset.....	35
Figure 18: Contour plots (decision boundary) for the Support Vector Machine model on the 2D synthetic dataset.....	36
Figure 19: ROC curves for the Support Vector Machine model on the 2D synthetic dataset.	36

Figure 20: Obtained centroids for the MNIST handwritten digit dataset using K-means.	37
Figure 21: Obtained centroids for the MNIST fashion dataset using K-means.....	37
Figure 22: The local communications terminal (Flask Server)	40
Figure 23: The demo execution of a Kernel Regression under POM6 in a single terminal	41
Figure 24: The local communications terminal (Flask Server)	42
Figure 25: The demo execution of a Kernel Regression under POM6 in full detail (WorkerNode)	43
Figure 26: The demo execution of a Kernel Regression under POM6 in full detail (MasterNode).....	43

List of Acronyms and Abbreviations

Abbreviation	Definition
AUC	Area Under (ROC) Curve
BSVM	Budget SVM
CA	Consortium Agreement
CN	Cryptonode
DC	Data Connector
FML	Federated Machine Learning
FR	Functional Requirements
GA	Grant Agreement
HBC	Honest But Curious (a.k.a SH)
IDR	Intermediate Data Representation
KM	Kernel Method
KR	Kernel Regression
LC	Logistic Classifier
LM	Linear Model
LR	Logistic Regression
ML	Machine Learning
MMLL	Musketeer Machine Learning Library
MLC	Multiclass Logistic Classifier
MN	Master Node
OS	Operating System
PERT	Program evaluation and review technique
POM	Privacy Operation Mode
PP	Privacy Preserving
PPML	Privacy Preserving Machine Learning (a.k.a. Privacy Preserving Data Mining)
ROC	Receiver Operating Characteristics
RR	Ridge Regression
SH	Semi Honest (a.k.a HBC)
SQL	Structured Query Language
UI	User Interface
WN	Worker Node

1 Introduction

1.1 Purpose

This deliverable comprises the first preliminary version of the Machine Learning Library to be integrated in the MUSKETEER platform, under POMs 4, 5 and 6. From now on we will name this library as “MUSKETEER Machine Learning Library” (MMLL). Some demos to illustrate the behaviour of the library are also provided. These demos are not intended to be a benchmark of the library, they are provided for illustration purposes (the complete benchmark will be carried out in WP6). This deliverable will help other partners in the understanding of the POM 4, 5 and 6 algorithms design and usage, to facilitate their integration and use in the MUSKETEER platform.

1.2 Related Documents

D4.6 is the first deliverable associated to the task T4.4 (Algorithms over semi-honest privacy preserving operation modes), as indicated in the PERT diagram below. It uses as input previous outcomes of WP4 (D4.1 and D4.2), where a preliminary version of the library design has been described, as well as a possible usage in the form of a MUSKETEER demonstrator. It also takes as inputs the requirements and specifications detailed in WP2, and, although not indicated in the PERT diagram, it is also respectful with the functional requirements FR017-FR024 described in D3.1 in relationship with the communications library.

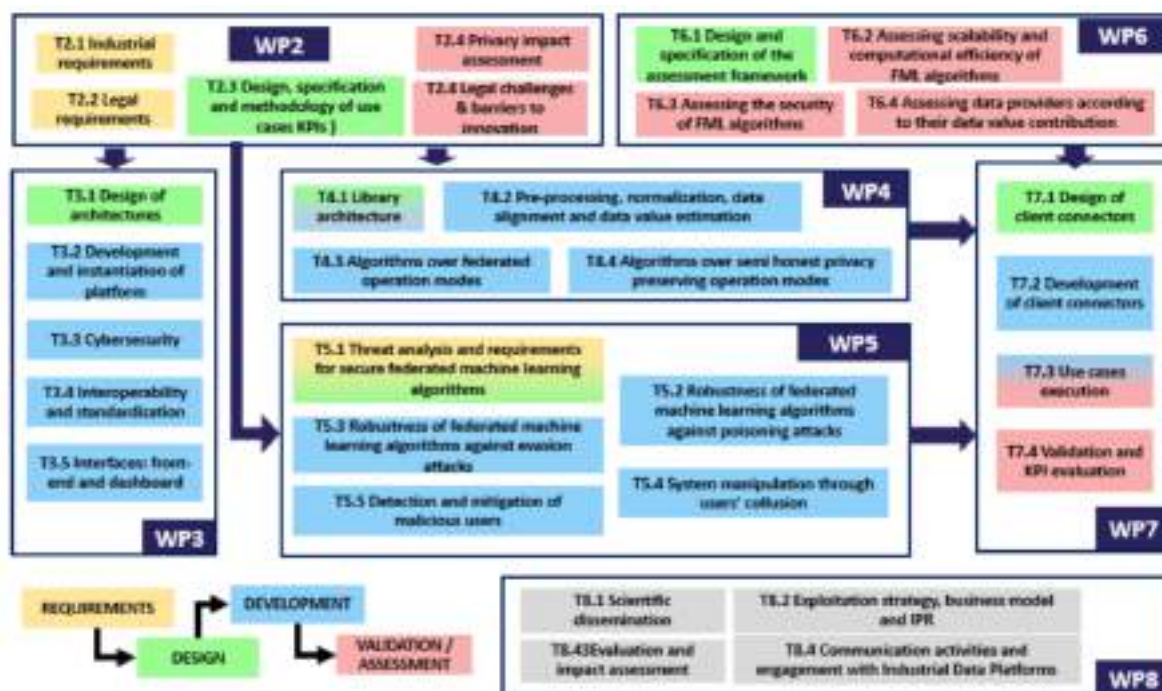


Figure 1: MUSKETEER's PERT diagram

1.3 Document Structure

This document is structured as follows:

- The current section (Introduction), presents the general aspects about this document and its relationship with other developments in the project.
- The section "Context of the Machine Learning Library" briefly revisits the main objectives of MUSKETEER from a Machine Learning point of view. We revisit some of the basic concepts about the platform execution, the participant processes and the corresponding objects. We also summarize the behaviour of every Privacy Operation Mode (POM) and how it will operate once integrated in the platform.
- The section 5: "Methodology" describes the development process of the software, its current state and the future goals.
- In Section 6 "Library Usage" we briefly describe the main steps needed to use the MMLL outside of the demos, to ease the integration step into the MUSKETEER platform. By now, some aspects still need to be re-engineered, but this could serve to get a general understanding of the library design and behaviour.
- In Section 7: "Results", we include some of the results of the algorithms applied on some selected datasets. This will serve as a first reference of the library behaviour, before proceeding with the execution of the demos in the following sections.
- The Section 8: "Installation" describes on a step-by-step basis the procedure to correctly install and execute the library in different Operating Systems (Windows, Linux and Mac OS).
- In Section 8: "Execution of the demos" we provide further detailed explanation about the demo execution process.
- In Section 9: "Sample software documentation" we provide some examples of the produced software documentation. The full version of the documentation is provided along with the code, and can be read with any web browser (html format).
- In Section 10: Conclusions, we provide a summary on the contents of the deliverable and the obtained results.
- Finally, some references are included in the last section.

2 Context of the Machine Learning Library

The library developed in this deliverable and described in this document is a preliminary yet fully operable version of the Machine Learning Library to be used in MUSKETEER under POMs 4, 5 and 6. Essentially, it aims at deploying a distributed ML setup (Figure 2b) such that a model equivalent to the one obtained in the centralized setup (Figure 2a) is obtained.

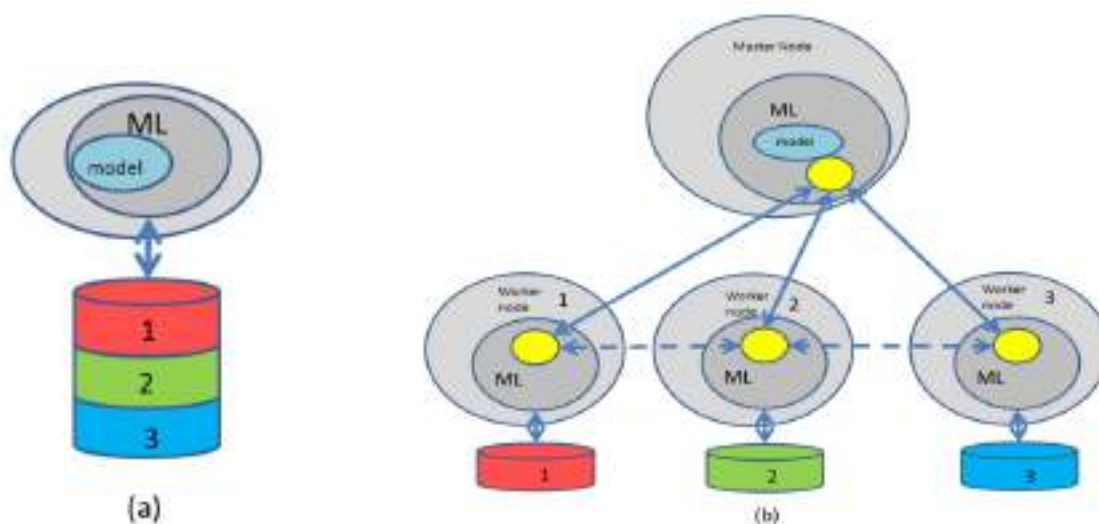


Figure 2: Centralized (a) vs. distributed scenario (b). Every user provides a portion of the training dataset. Data confidentiality must be preserved.

The centralized solution requires that the data from different users is gathered in a common location, something that is not always possible due to privacy/confidentiality restrictions. On the other hand, the distributed privacy preserving approach requires to exchange some information (intermediate data representation¹, IDR) among the participating users such that a Master Node (MN) obtains the final ML model without ever receiving/seeing the raw data of the users.

In a second level of detail, we can describe the interaction among nodes as shown in the next Figure:

¹ Any intermediate data representation should carry some information about the data it is derived from (to allow learning), while hiding the actual raw data values to the participants in the protocol. Averaged gradients, auto-correlation matrices or cross-correlation vectors could be examples of IDR, each one revealing different partial information about the datasets. Any form of (homomorphic) encryption can also be considered as an IDR, since it allows to compute some operations on the data, while protecting the raw data.

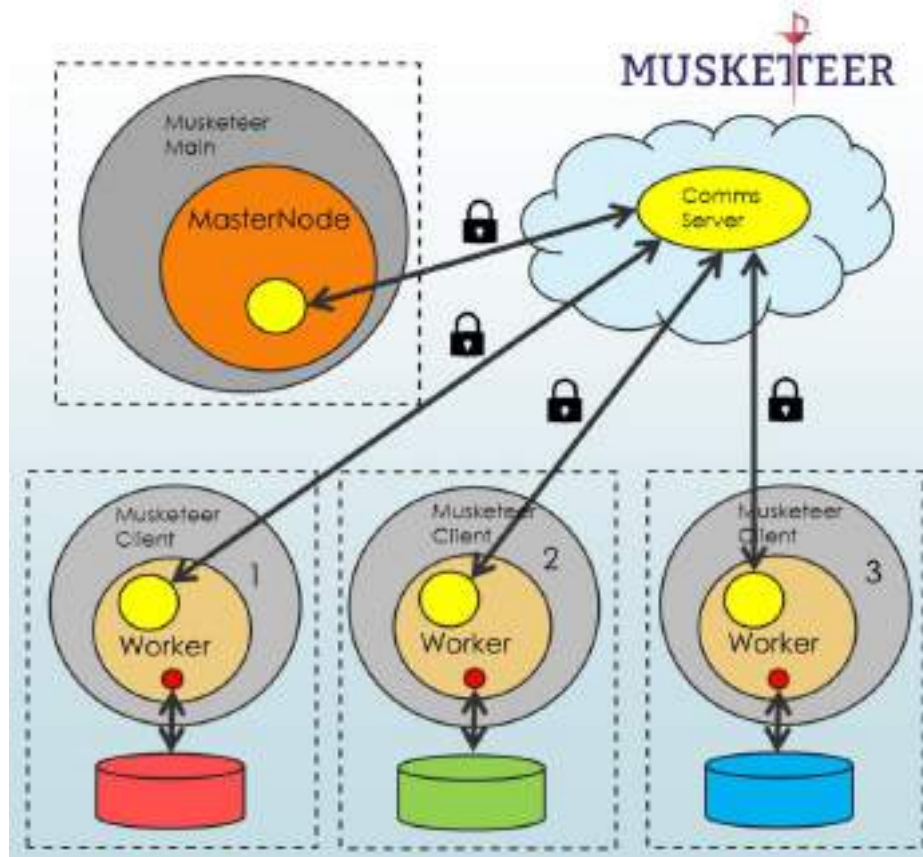


Figure 3: Detailed process interactions in a MUSKETEER learning process.

We observe the participation of several actors in a learning process, everyone marked as a dashed box and supposedly running on a different (remote) machine:

- The **MUSKETEER main process**: it is the process that orchestrates the training procedure, identifies the potential contributors and obtains the final model. It runs the “MasterNode” object (dark orange circle) from the MMLL. It communicates by means of the communication object (yellow circle) with the other participants through the Communications Service at the Cloud.
- The **MUSKETEER client**: it is the process that every participant must locally execute. It runs the “WorkerNode” object (light orange circle) from the MMLL. The Worker has access to the local data through the specific data connector (red circle) provided by the end user and communicates with the MasterNode by means of the communication object (yellow circle) through the Communications Service at the Cloud.

In the next Section we describe in a deeper detail the structure of the objects participating in POMs 4, 5 and 6, as well as the expected interactions among the different types of nodes.

3 POMs 4, 5 and 6 revisited

General aspects:

The following nodes (objects) are to be executed:

Common to POMs 4, 5 and 6:

- **Master Node (MN)**: a central object (process) that controls the execution of the training procedure
- **Worker Node (WN)**: an object to be executed in the end user side, possibly as a part of the MUSKETEER client. It is the only node that has a direct access to the raw data provided by every user, through an 'ad-hoc' Data Connector (DC).

Specific to POM 4:

- **Crypto Node (CN)**: an object providing some cryptographic operations. It can be run anywhere but it cannot collude with the Master Node. It is only needed in POM 4, because POM6 does not use encryption and in POM5 the Master Node plays the role of Crypto Node.

In what follows we describe the normal operation of a training algorithm under every POM.

3.1 POM 4

This POM uses an additively homomorphic cryptosystem to protect the confidentiality of the data. The CN will help in some of the unsupported operations. The scheme is cryptographically secure if we guarantee that there is no collusion between the MN and the CN. In the next Figure we represent the interaction among the participants.

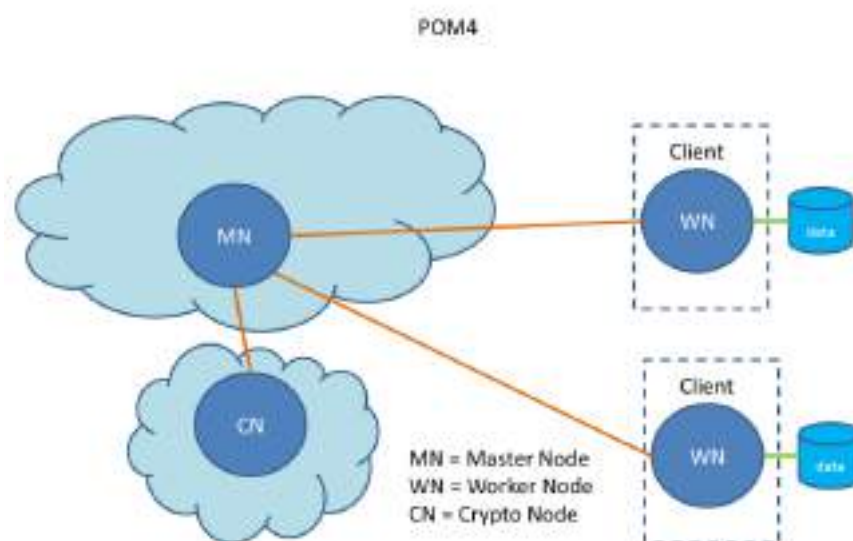


Figure 4: POM 4 general setup.

The steps to train a given model are:

1. The MN asks to the CN some general public parameters, and distributes them to the WNs.
2. Every Node will use those parameters to generate public and private keys. The public keys are distributed. The CN generates a Master key, able to decrypt anything.
3. Every WN encrypts the data with their secret keys and sends the encrypted data to the MN.
4. The MN sends the data with blinding to the CN, to re-encrypt it to the Master key. The re-encrypted data is returned to the MN.
5. The MN starts the training procedure by operating on the (encrypted) model parameters and (encrypted) users data. The initial model parameters are generated at random by the MN.
6. The MN is able to perform some operations on the encrypted data (the homomorphically supported ones).
7. For the unsupported ones, it needs to establish a secure protocol with the CN consisting in:
 - a. The MN sends some data with blinding to the CN
 - b. The CN decrypts the data and computes the unsupported operation in clear text. Then it encrypts the result.
 - c. The MN receives the encrypted result and removes the blinding.

As a result of this protocol, the MN never sees the data or the result in clear text and the CN only sees the clear text of a blinded message, different from the raw data.

8. The procedure goes back to 5 until a stopping criterion is met.

POM 4 is a cryptographically secure procedure, providing that MN and CN do not collude.

3.2 POM 5

This POM has been re-engineered to better comply with some of the platform requisites: improved performance and no need to run non-colluding nodes. It uses an additively homomorphic cryptosystem to protect the confidentiality of the data and model. The MN will help in some of the unsupported operations, this is, the MN will play the role of CN. The scheme is cryptographically secure. In the next Figure we represent the interaction among the participants.

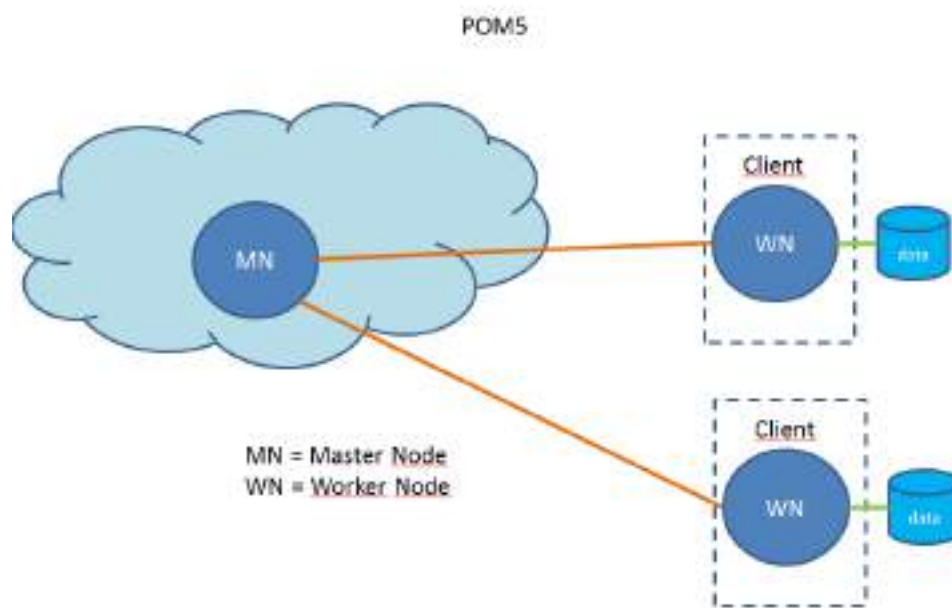


Figure 5: POM 5 general setup.

The steps to train a given model are:

1. The MN generates public and private keys. The public keys are distributed to all participants.
2. The initial model parameters are generated at random by the MN. The MN encrypts the model parameters with his secret keys and sends the encrypted model to the WNs.
3. The WN starts the training procedure by operating on the (encrypted) model and (un-encrypted) users data.
4. The WN is able to perform some operations on the encrypted data (the homomorphically supported ones).
5. For the unsupported ones, the WN needs to establish a secure protocol with the MN consisting in:
 - a. The WN sends some encrypted data with blinding to the MN
 - b. The MN decrypts the data and computes the unsupported operation in clear text. Then it encrypts the result.
 - c. The WN receives the encrypted result and removes the blinding.

As a result of this protocol, the MN never sees the data or the result in clear text, and the WN only sees the encrypted model.

6. The procedure goes back to 5 until a stopping criterion is met.

POM 5 is a cryptographically secure procedure.

3.3 POM 6

This POM does not use encryption; it relies on Secure Multiparty Computation and possibly other (two-party) Secret Sharing protocols to solve some operations on distributed data. In the next Figure we represent the interaction among the participants.

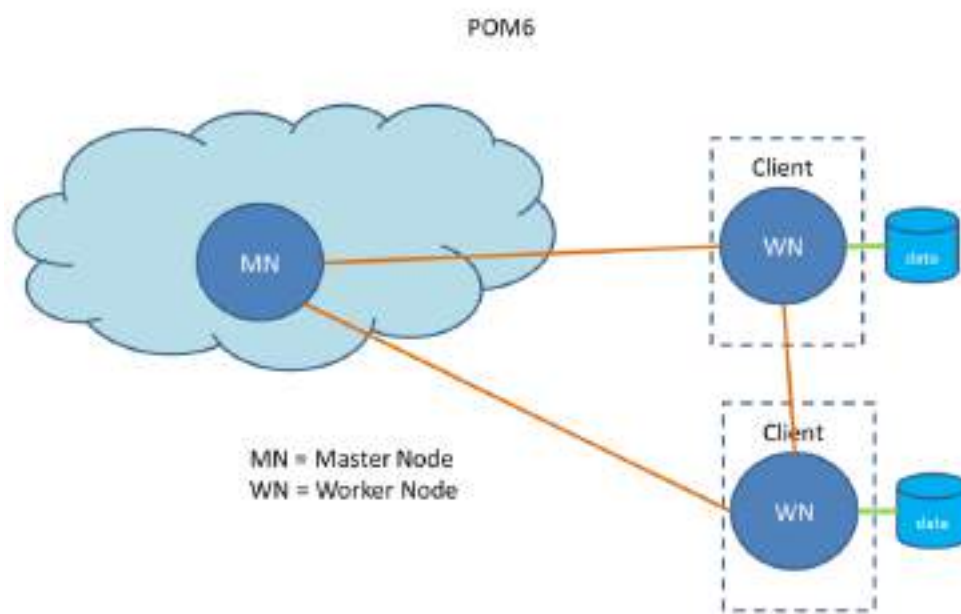


Figure 6: POM 6 general setup.

Under this POM, raw data is not encrypted, but it is never sent outside the WN. The model trained in the MN can also be kept secret to the WN. Some transformations of the data can be exchanged with the MN, such as aggregated values, correlation matrices, etc. Every implemented algorithm will describe which information is revealed to the MN, for instance: covariance matrices, number of training patterns, average of the training patterns, etc. In any case, the raw data (individual training patterns) will not be revealed and cannot be obtained by inverse engineering on the exchanged data.

Some of the operations can be directly implemented using SMC protocols such as secure dot product, secure matrix multiplication, etc. The security of these operations will be as described in the reference sources describing every protocol. POM6 is not a general procedure, it requires that every algorithm is implemented from scratch, and it is not guaranteed that any algorithm can be implemented under POM6. For some operations, a

“round robin” protocol is required; therefore direct connections among some of the WNs are needed (ring network).

As an illustrative example, let’s imagine a training procedure that requires at every step to receive the average covariance matrix among all the WNs and to compute one dot product. The procedure could be as follows:

1. The MN asks the WNs to compute their covariance matrices.
2. The MN starts a round robin protocol with blinding to obtain the accumulated covariance matrix
3. The MN starts a SMC protocol to obtain the dot product with the data from every WN.
4. Using the received information the MN updates the model (the specific correlation matrices of every worker are not revealed).
5. The procedure goes back to 1 until a stopping criterion is met.

4 Methodology

4.1 General development process

The library development follows these steps:

1. Develop an algorithm prototype without communications library
2. Adaptation to the provisional local communications library provided by IBM
3. Preliminary version with the code structure agreed between UC3M and TREE
4. MMLL 1.0: preliminary version of the library (provided with this Deliverable D4.6, as long as some demos)
5. Algorithm & code optimization (mainly to be carried out during the next months)
6. Usage of the final communications service (IBM Cloud)
7. MMLL 2.0: final version of the library (to be provided in D4.7 (M30))

4.2 Current status of the library and future steps

We briefly describe here the current status of the algorithms/POMs, mainly concentrating on some of the mentioned algorithms for the current Deliverable (D4.6): Linear models, kernel methods and clustering (Kmeans).

Cross-correlation estimation

1. Prototype without communications library	Done
2. Adaptation to IBM’s local communications library	Done
3. Preliminary version with common code structure	Done
4. Cross-correlation DEMO v1.0.0	Released (Provided in D4.6)
5. Algorithm & code optimization	To be provided in D4.7 (M30)
6. Usage of the final comms. service (IBM Cloud)	To be provided in D4.7 (M30)
7.- MMLL 2.0: final version of the library	To be released in D4.7 (M30)

Ridge Regression	
1. Prototype without communications library	Done
2. Adaptation to IBM’s local communications library	Done
3. Preliminary version with common code structure	Done
4. Ridge Regression DEMO v1.0.0	Released (Provided in D4.6)
5. Algorithm & code optimization	To be provided in D4.7 (M30)
6. Usage of the final comms. service (IBM Cloud)	To be provided in D4.7 (M30)
7.- MMLL 2.0: final version of the library	To be released in D4.7 (M30)

Linear Regression	
1. Prototype without communications library	Done
2. Adaptation to IBM’s local communications library	Done
3. Preliminary version with common code structure	Done
4. Linear Regression DEMO v1.0.0	Released (Provided in D4.6)
5. Algorithm & code optimization	To be provided in D4.7 (M30)
6. Usage of the final comms. service (IBM Cloud)	To be provided in D4.7 (M30)
7.- MMLL 2.0: final version of the library	To be released in D4.7 (M30)

Logistic Classifier	
1. Prototype without communications library	Done

2. Adaptation to IBM’s local communications library	Done
3. Preliminary version with common code structure	Done
4. Logistic Classifier DEMO v1.0.0	Released (Provided in D4.6)
5. Algorithm & code optimization	To be provided in D4.7 (M30)
6. Usage of the final comms. service (IBM Cloud)	To be provided in D4.7 (M30)
7.- MMLL 2.0: final version of the library	To be released in D4.7 (M30)

Multiclass Logistic Classifier	
1. Prototype without communications library	Done
2. Adaptation to IBM’s local communications library	Done
3. Preliminary version with common code structure	Done
4. Multiclass Logistic Classifier DEMO v1.0.0	Released (Provided in D4.6)
5. Algorithm & code optimization	To be provided in D4.7 (M30)
6. Usage of the final comms. service (IBM Cloud)	To be provided in D4.7 (M30)
7.- MMLL 2.0: final version of the library	To be released in D4.7 (M30)

Clustering (Kmeans)	
1. Prototype without communications library	Done
2. Adaptation to IBM’s local communications library	Done
3. Preliminary version with common code structure	Done
4. Clustering (Kmeans) DEMO v1.0.0	Released (Provided in D4.6)
5. Algorithm & code optimization	To be provided in D4.7 (M30)
6. Usage of the final comms. service (IBM Cloud)	To be provided in D4.7 (M30)
7.- MMLL 2.0: final version of the library	To be released in D4.7 (M30)

Kernel Regression	
1. Prototype without communications library	Done
2. Adaptation to IBM’s local communications library	Done

3. Preliminary version with common code structure	Done
4. Kernel Regression DEMO v1.0.0	Released (Provided in D4.6)
5. Algorithm & code optimization	To be provided in D4.7 (M30)
6. Usage of the final comms. service (IBM Cloud)	To be provided in D4.7 (M30)
7.- MMLL 2.0: final version of the library	To be released in D4.7 (M30)

Budget Support Vector Machine	
1. Prototype without communications library	Done
2. Adaptation to IBM’s local communications library	Done
3. Preliminary version with common code structure	Done
4. Budget Support Vector Machine DEMO v1.0.0	Released (Provided in D4.6)
5. Algorithm & code optimization	To be provided in D4.7 (M30)
6. Usage of the final comms. service (IBM Cloud)	To be provided in D4.7 (M30)
7.- MMLL 2.0: final version of the library	To be released in D4.7 (M30)

5 Library Demonstration preliminary assumptions

In what follows, we assume that a Machine Learning task has already been defined, and that the MUSKETEER platform has already identified all the potential users participating in the training process. In the complete, end-to-end version of the MUSKETEER platform, the services which allow users to register to the platform, define tasks and join tasks will be developed under WP3.

Therefore, for the purpose of this demonstrator, we will assume the following:

- **General description of the task:** All participants have access to this description and agree to participate and contribute some data to the learning process. A preliminary check procedure has already been executed to guarantee that the contributed data follows the needed format (number and type of input features, number and type of target values, etc.).
- **User addresses and execution:** the list of addresses of the participating nodes (Worker Nodes (WN)) is available to the MasterNode, according to FR017 in D3.1. In the final version every participant (Master/Cryptonode/Workers) will be a separate process in a potentially different machine/location. The current version of the

Communications Library (CL) is primarily designed to communicate between processes in the same machine, and we have executed these simulations using this approach, but in the future the experiments will also cover different remote machines communicated through the IBM Cloud.

- **Data:** the data for training, validating and testing will be provided to MUSKETEER by means of a Data Connector (DC). For illustration purposes we provide here a DC to be used in the demonstrator that simply loads data from a file. The final DC for the user cases will have to be developed in other parts of the project, possibly at WP7. For future uses, any other compatible data connector can be used if provided by the end user (SQL access, for instance). For the purpose of this demonstration we provide some public datasets along with the specific needed Data Connector. Some other larger datasets can also be downloaded if extra experiments are to be done.
- **Confidentiality requirements:** We will assume that the raw data is never sent (in clear text, or unencrypted) outside of the owner's context and that the trained model is kept secret (only known to the Master Node). We will allow to exchange among the participants some IDR, transformations of the data (such as aggregations, cross-correlation matrices, encrypted values, etc.), but in any case that information cannot be used to reconstruct the raw input data or targets. The final end users will be aware in advance of the type of information exchanged under every Privacy Operation Mode (POM), and it is their ultimate responsibility to choose among one POM or another.
- **Communications library:** The MMLL needs a Comms object to operate and it is agnostic with the particular implementation of the communication service whenever the Functional Requirements FR017-FR024 in D3.1 have been respected. Namely, the Comms object must provide basic "send/receive" functionalities, and its interface needs to contain, as a minimum, the following methods²:

- **At Master Node (MN):**

- Send a message**

- Functional description: send a message from the MN to the worker identified with "*worker_id*":

- Input:

- message*: the message to be sent. It can be of any type, the Comms object must serialize/deserialize it if needed.

² We will use the term MN (Master Node) here, but it can also be interpreted as the Aggregator

worker_id: the recipient id, type=string

Output:

None

Example of use:

```
comms.send(message, worker_id)
```

Broadcast:

Functional description: send a message from the MN to all workers:

Input:

message: the message to be sent. It can be of any type, the Comms object must serialize/deserialize it if needed.

Output:

None

Example of use:

```
comms.broadcast(message)
```

Send over ring:

Functional description: Send a message through all workers (the order is irrelevant), following the ring topology, starting and ending in the MN (MN -> worker1 -> worker2 -> ... -> workerN -> MN):

Input:

message: the message to be sent. It can be of any type, the Comms object must serialize/deserialize it if needed.

Output:

None

Example of use:

```
comms.send_ring(message)
```

Receive:

Functional description: enter in a “receive” state until a message is received or a timeout is passed):

Input:

None

Output:

message: the received message, in the same format as sent by the sender.

Example of use:

```
message = comms.receive()
```

○ **At any Worker Node (WN):**

Send:

Functional description: send a message from the Worker Node to the MN:

Input:

message: the message to be sent. It can be of any type, the Comms object must serialize/deserialize it if needed.

Output:

None

Example of use:

```
comms.send(message)
```

Receive:

Functional description: enter in a “receive” state until a message is received or a timeout is passed):

Input:

None

Output:

message: the received message, in the same format as sent by the sender.

Example of use:

```
message = comms.receive()
```

6 MUSKETEER Machine Learning Library Usage

In this section we will briefly describe the potential usage of the library outside of the demos, to ease its integration in the final prototypes.

Important note: the pseudocode shown in this section is only for illustrative purposes and library comprehension, it is not intended to work as it is. The interested reader will need to look into one of the demo scripts to fully understand all the needed parameters.

6.1 Communications setup

As mentioned before, we will restrict by now to using different processes in the same machine and interconnect them with the local communications library provided by IBM (Flask Server). Therefore, we need to start that server, by running:

```
python3 local_flask_server.py
```

The communications system is now ready to exchange messages among the participating nodes. In the future, when the Cloud Communications service and the corresponding API will be completed according to FR017-FR024 in D3.1, it will be possible to communicate processes among different machines.

6.2 Setting up the Worker Node (end user side)

The Worker Node is the object that controls the behaviour of the MMLL on the end-user side. First of all we need to import it from the library:

```
from MMLL.nodes.WorkerNode import WorkerNode
```

Before instantiating it, we need some extra objects³: the data connector (DC), the Communications object (Comms) and, in some POMs, the Crypto object. We start importing them from the library:

```
from MMLL.data_connectors.Load_from_file import Load_From_File as DC  
from MMLL.comms.comms_local_Flask import Comms
```

³ We will restrict here to the description of the main variables, the interested reader may read the code of one of the demos for a full understanding.


```
from MMLL.crypto.CryptoBCP_beta import CryptoBCP
```

We instantiate the DC object. In the “load from file” case, we need to provide as input parameter the filename where the data is stored, in other cases, the DC will need parameters to access the data. The DC must have a “get_data_Worker” that returns one 2D array with the input features (Number of patterns x Number of features), and a 1D array with the targets (if the task is a supervised one). This method will be used by the WorkerNode to get the training data.

```
data_file = './mydata.txt'  
dc = DC(data_file)
```

We then instantiate the Comms object, which needs as input parameter the Worker ID (any unique string will serve in this case):

```
worker_id = 'worker_1'  
comms = Comms(worker_id)
```

Algorithms in POMs 4 and 5 need a Cryptographic object to operate, the key_size parameters determines the strength of the encryption:

```
cr = CryptoBCP(key_size=512)
```

The next step is to create the WorkerNode itself, and we pass as parameters the selected pom, the worker ID, the address of the Master Node, the Comms object, the DC object and the Cryptographic object:

```
pom = 5  
wn = WorkerNode(pom, worker_id, comms, dc, master_address='ma',  
cr=cr)
```

We load the data:

```
wn.load_data()
```

We create the model of the selected type:

```
model_type = 'Kmeans'  
wn.create_model_worker(model_type)
```

And we execute the training loop at the worker:

```
wn.run()
```

The worker will enter into a listening state, waiting for instructions from the Master Node. It will stop when the training is completed.

6.3 Setting up the Master Node

The Master Node is the object that orchestrates the training procedure among all other participating nodes. First of all we need to import it from the library:

```
from MMLL.nodes.MasterNode import MasterNode
```

Before instantiating it, we need some extra objects⁴: the data connector (DC) is only needed if some validation or test data is to be used by the MasterNode, the Communications object (Comms) and, in some POMs, the Crypto object. We start importing them from the library:

```
from MMLL.data_connectors.Load_from_file import Load_From_File as DC  
from MMLL.comms.comms_local_Flask import Comms  
from MMLL.crypto.CryptoBCP_beta import CryptoBCP
```

We instantiate the DC object. In the “load from file” case, we need to provide as input parameter the filename where the data is stored, in other cases, the DC will need parameters to access the data. The DC must have a “get_data_Master” that returns one 2D array with the input features (Number of patterns x Number of features), and a 1D array with the targets (if the task is a supervised one), for both validation and test cases. This method will be used by the MasterNode to get the training data.

```
data_file = './mydata.txt'  
dc = DC(data_file)
```

We then instantiate the Comms object, which needs as input parameter the MasterNode ID (any unique string will serve in this case):

⁴ We will restrict here to the description of the main variables, the interested reader may read the code of one of the demos for a full understanding.

```
master_address = 'ma'  
comms = Comms(master_address)
```

Algorithms in POMs 4 and 5 need a Cryptographic object to operate, the `key_size` parameters determines the strength of the encryption:

```
cr = CryptoBCP(key_size=512)
```

The next step is to create the `MasterNode` itself, and we pass as parameters the selected POM, the worker ID, the address of the Master Node, the `Comms` object, the `DC` object and the `Cryptographic` object:

```
pom = 5  
master_address = 'ma'  
  
mn = MasterNode(pom, master_address, comms, dc, cr=cr)
```

(Note: some extra parameters may be needed, depending on the model to be trained...)

We load the data:

```
mn.load_data()
```

We create the model of the selected type:

```
model_type = 'Kmeans'  
mn.create_model_worker(model_type)
```

And we start the training procedure:

```
mn.fit()
```

7 MUSKETEER Machine Learning Library results

In the upcoming sections we will describe the steps to install the library and run the experiments on a variety of simulations to evaluate the correct operation of the library. In this section we anticipate some of the results of Machine Learning experiments using the

developed Machine Learning Library, so the reader can see some results before running some experiments by him/herself.

The results shown here are for illustration purpose, they do not represent any kind of benchmark of the library, such a task will be completed during WP 6. All the datasets used here have already been described in D6.1. Anyhow, the observed results are as expected and they represent solutions comparable to those obtainable in the centralized case.

All the results shown here have been obtained using the MUSKETEER Machine Learning Library under privacy constraints, this is, the data provided by the users is always protected and kept as confidential, not revealed to the training algorithm (at least in clear text form). The experiments have been run using 5 data providers (5 worker nodes, hence every training dataset has been split into 5 separate participants).

7.1 Cross-Correlation (XC) estimation

We provide means to securely estimate the normalized cross-correlation (XC in short) among inputs and between input and output following the Pearson correlation definition [Pearson_Corr]. As an example, we show here the results obtained for the first 10 highest (in absolute value) correlation values among variables in the redwine dataset:

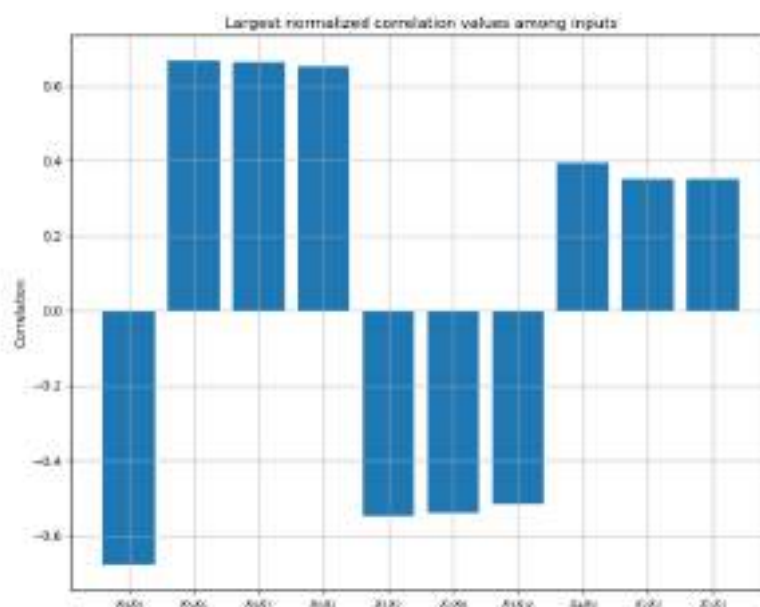


Figure 7: Normalized cross-correlation values among inputs for the redwine dataset.

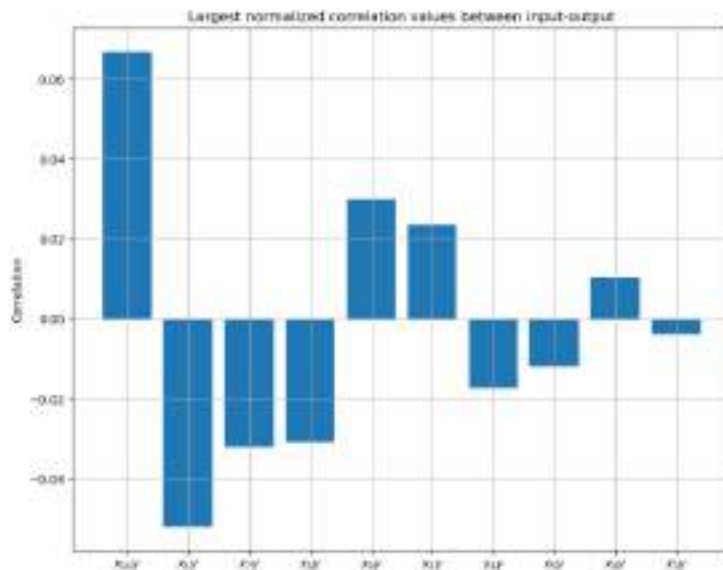


Figure 8: Normalized cross-correlation values between inputs and outputs for the redwine dataset.

7.2 Ridge Regression (RR) estimation

We have implemented a Ridge Regression model (RR in short, also known as Tikhonov regularization) operating under privacy constraints, which is essentially a linear model that includes a regularization term, providing robustness against overfitting [Ridge_Regression]. We have applied that model to the redwine dataset to estimate the quality of the wine, obtaining the following performance results:

NMSE on validation set = 0.0167

NMSE on test set = 0.0144

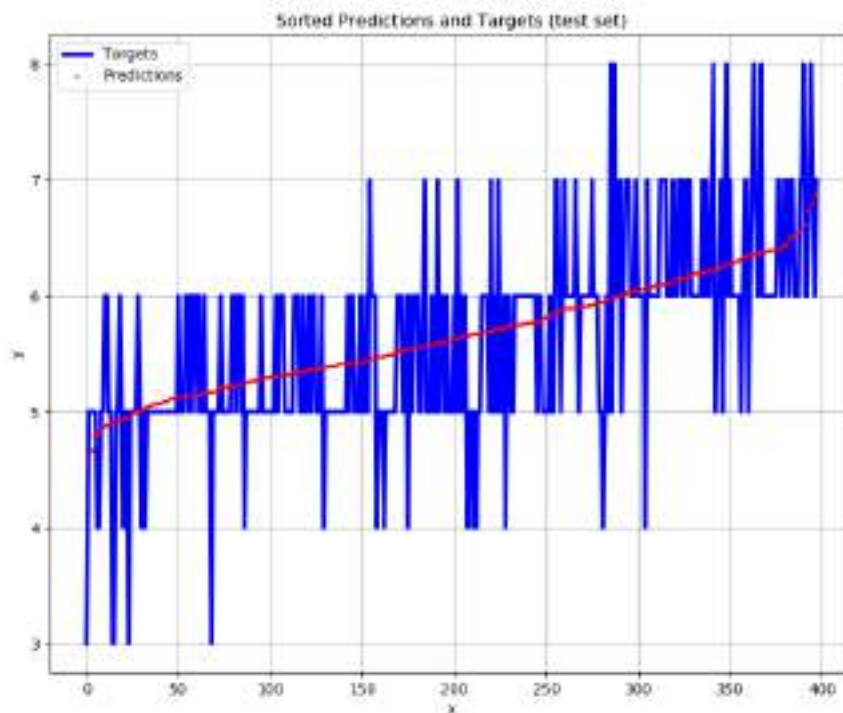


Figure 9: Illustration of the target and predicted values using the Ridge Regression estimation for the redwine dataset.

7.3 Kernel Regression (KR) estimation

This is the result of a Kernel Regression model (KR in short) on a synthetic 1-D signal. Kernel regression uses a nonlinear transformation of the data (here using a Gaussian Kernel), to improve the prediction capabilities of the model. The cost function used in the output layer is the quadratic loss.

NMSE on validation set = 0.0053

NMSE on test set = 0.0055

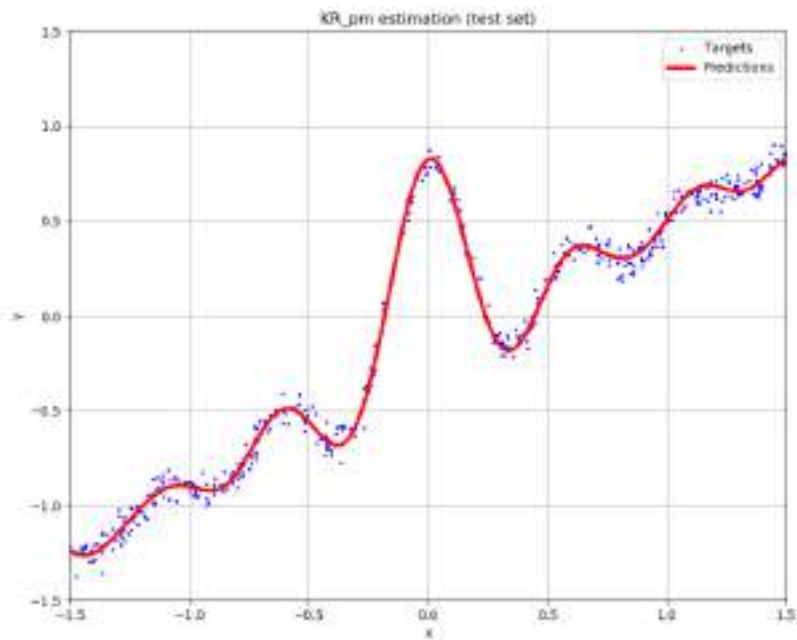


Figure 10: Results of the Kernel Regression model on the 1D synthetic dataset.

7.4 Logistic Classifier (LC)

A Logistic Regression model can easily be converted into a Logistic Classifier (LC in short) by simply adding a threshold on the outputs after training [Logistic Classifier]. It is a very popular model in the Machine Learning Community because of its simplicity and good performance in many tasks. We show in the next Figure the results for the pima dataset, where the ROC curves (on validation and test sets) are shown:

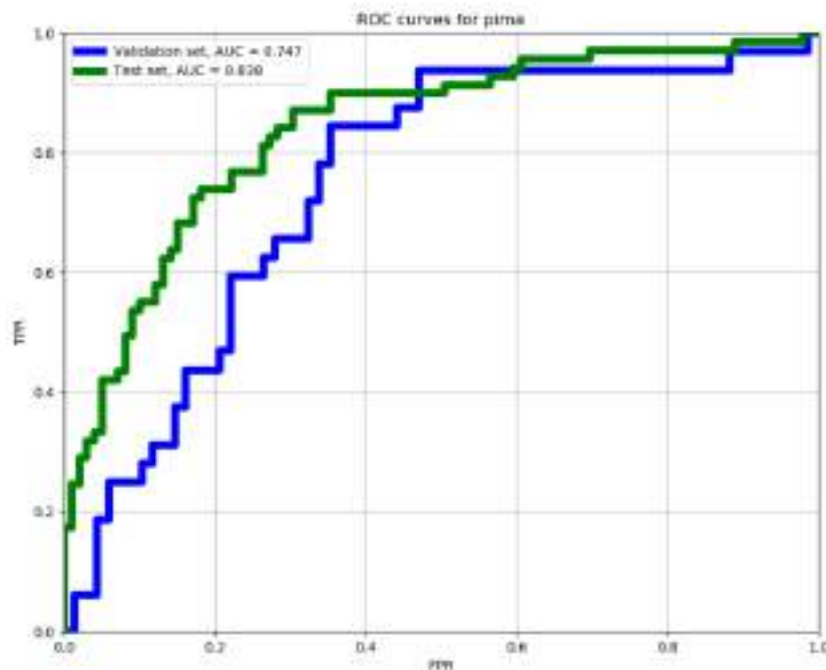


Figure 11: ROC curves for the Logistic Classifier model on the pima dataset.

We also show results for the LC model on the MNIST handwritten digits dataset (binary transformation of the dataset, such that the new task is to separate between even and odd numbers):

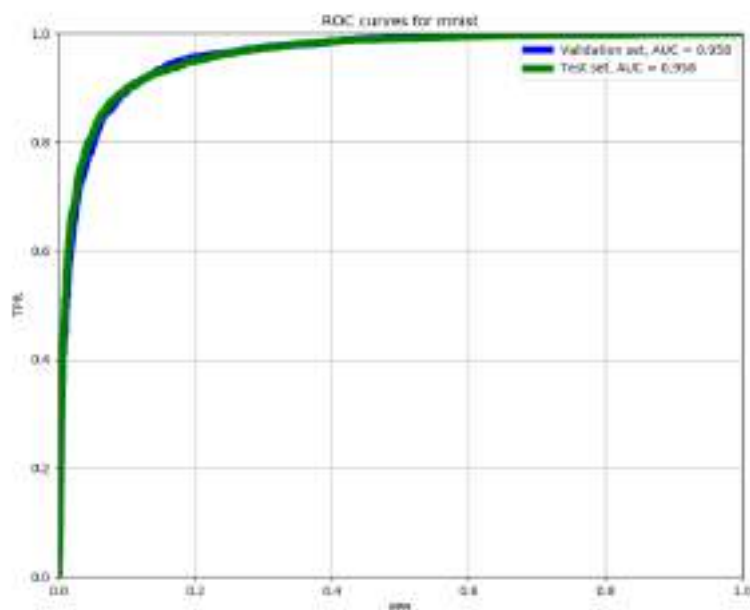


Figure 12: ROC curves for the Logistic Classifier model on the binarized MNIST handwritten digits dataset.

7.5 Multiclass Logistic Classifier (MLC)

We have implemented the multiclass extension of the Logistic Classifier (MLC in short), to deal with datasets with multiple classification targets. We show here the results for the MNIST dataset:

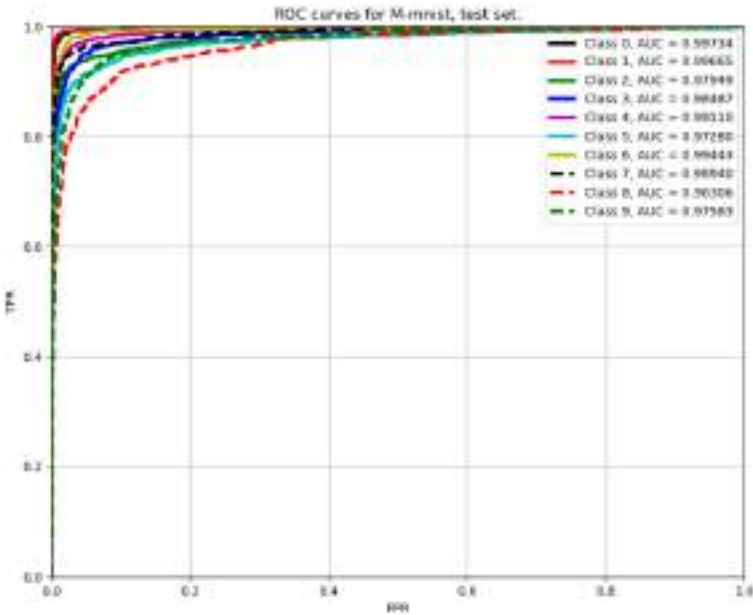


Figure 13: ROC curves for the Multiclass Logistic Classifier model on the MNIST handwritten digits dataset. One ROC curve is shown here for every class value, under a one-vs-all approach.

We have also computed the confusion matrix for this task and we show the results on the test set in the next Figure. That matrix indicates the number of confusions among target and predicted classes. The larger the diagonal values, the better, the error are always shown off the diagonal.

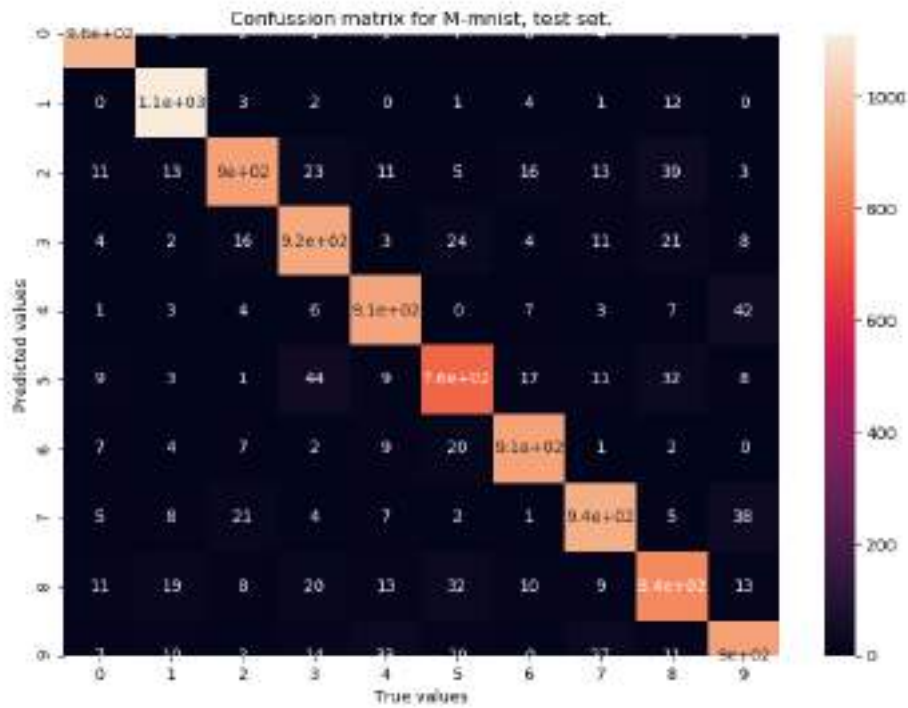


Figure 14: Confusion matrix for the Multiclass Logistic Classifier model on the MNIST handwritten digits dataset.

We also provide results for the MNIST-fashion dataset:

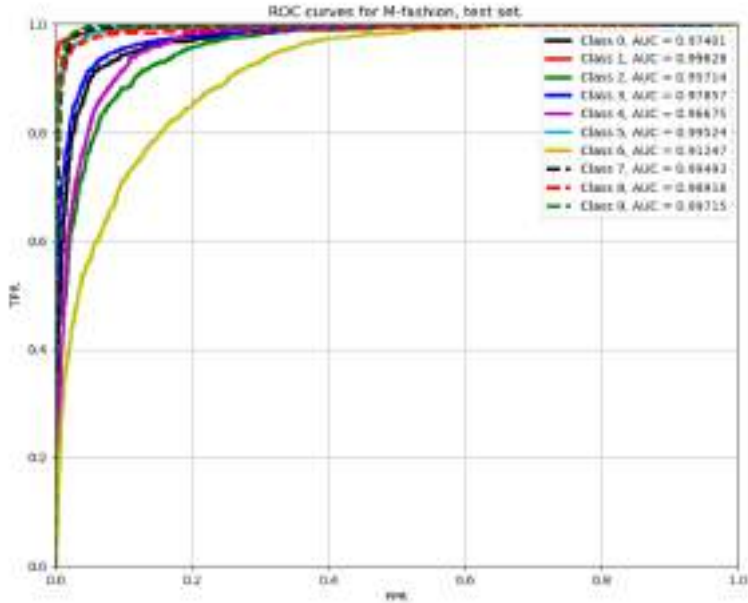


Figure 15: ROC curves for the Multiclass Logistic Classifier model on the MNIST fashion dataset. One ROC curve is shown here for every class value, under a one-vs-all approach.

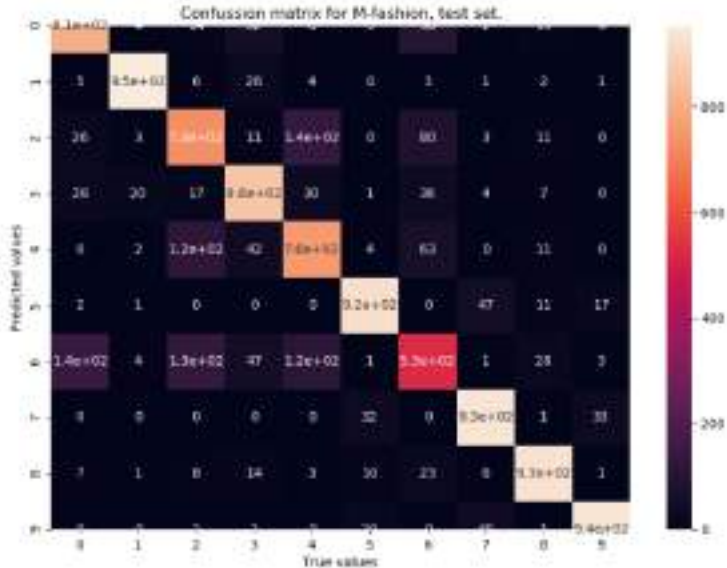


Figure 16: Confusion matrix for the Multiclass Logistic Classifier model on the MNIST fashion dataset.

7.6 Budget Support Vector Machine (BSVM)

Support Vector Machines [Support Vector Machine] are a very popular ML method, known by their robustness in real world problems. They are one subtype of the ML approaches broadly known as Kernel Methods. In this library we provide an implementation of the Budget SVM version (BSVM in short), which relies on a previous transformation of the input data (in our case, by using a clustering approach to define Gaussian Kernels). The main advantage of this approach is that the complexity of the resulting model is bounded, since the size of the machine is defined a priori. Furthermore, it represents a much secure approach, since in the original formulation of the SVM, the model is constructed using Support Vectors, which are representatives of the input training data, and hence it does not preserve data confidentiality.

In the next Figures we show the results for a synthetic 2-D dataset: ROC curves and contour plots.

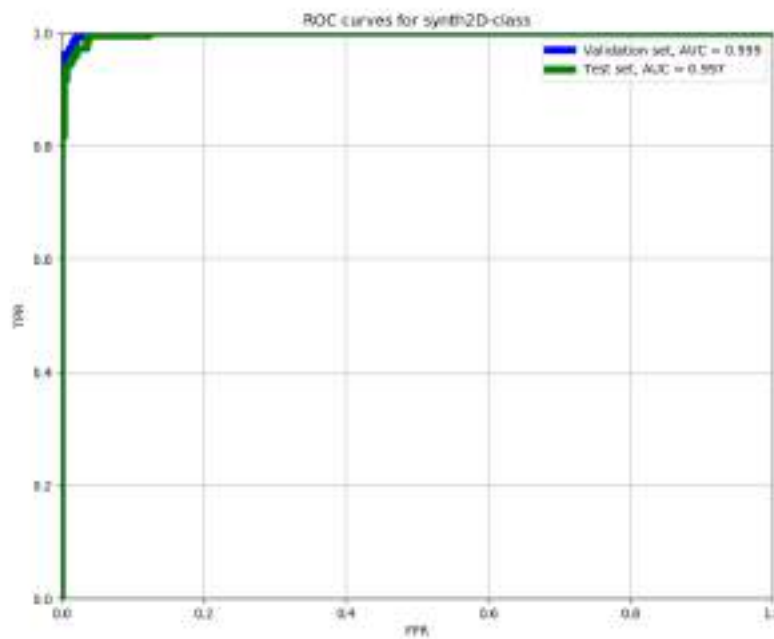


Figure 17: ROC curves for the Budget Support Vector Machine model on the 2D synthetic dataset.

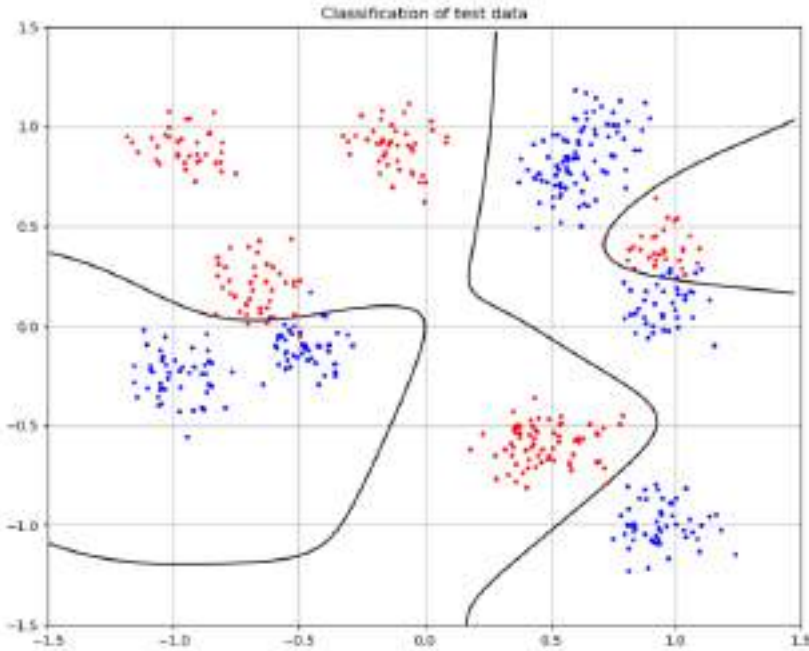


Figure 18: Contour plots (decision boundary) for the Support Vector Machine model on the 2D synthetic dataset.

We also show the results of BSVM on the pima dataset.

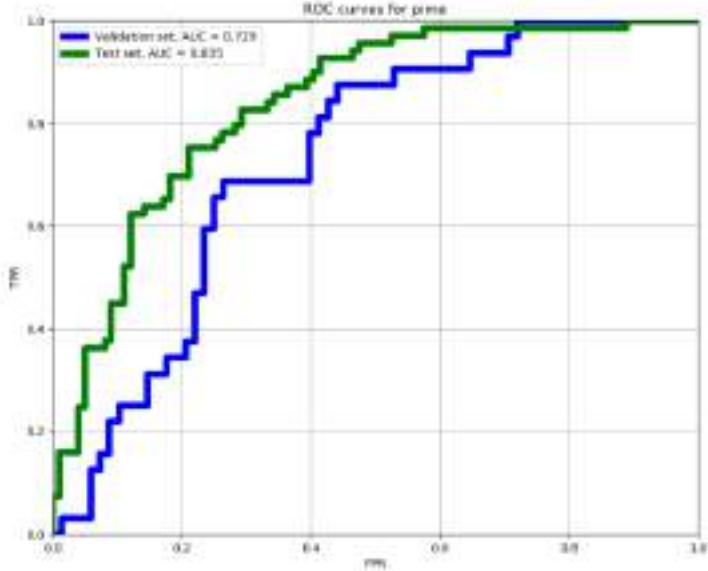


Figure 19: ROC curves for the Support Vector Machine model on the 2D synthetic dataset.

7.7 Clustering (K-means)

Results for MNIST (handwritten digits 0-9), showing 42 centroids obtained with a variety of digit writing.

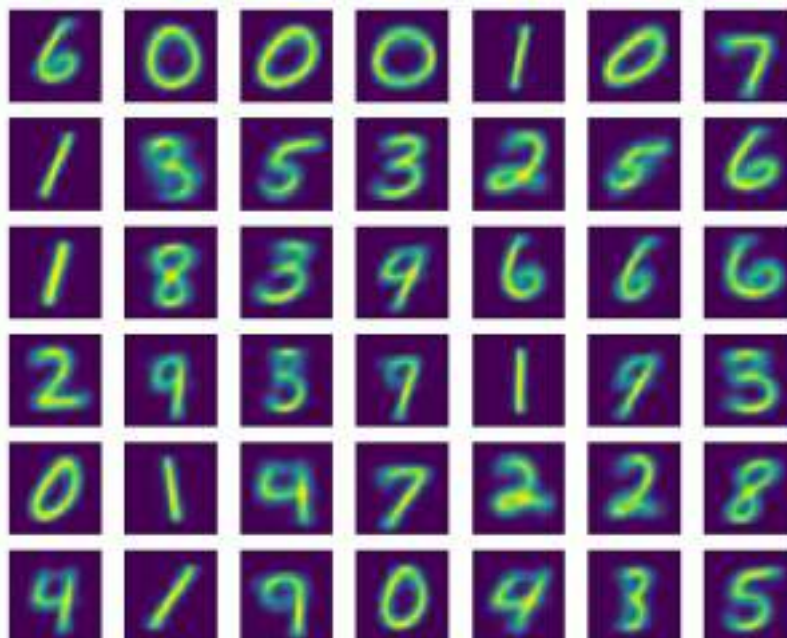


Figure 20: Obtained centroids for the MNIST handwritten digit dataset using K-means.

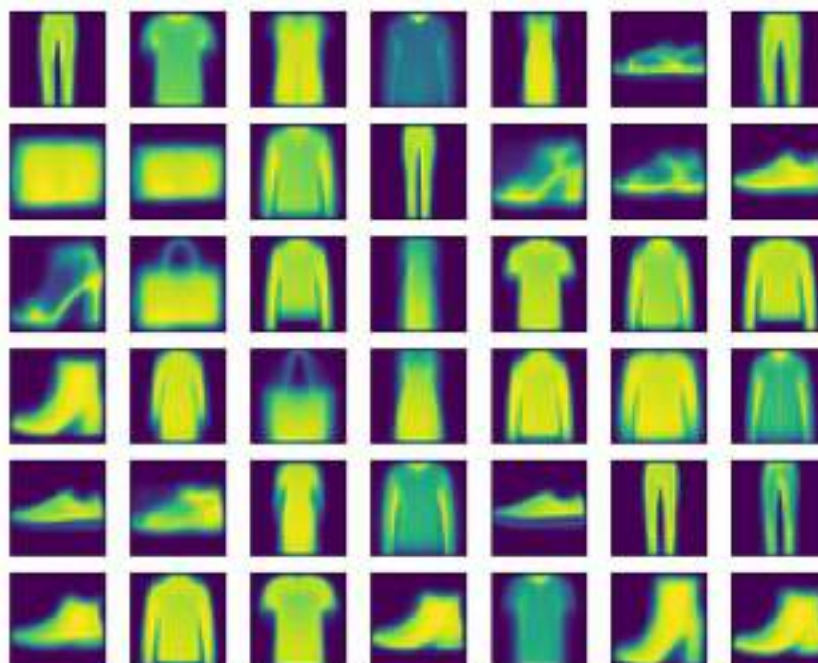


Figure 21: Obtained centroids for the MNIST fashion dataset using K-means.

8 Installing the library

Before executing the Demos, it is necessary to correctly configure a Python 3 environment with all the required libraries. In the final version of the platform, such configuration will be simplified, since the code will be embedded in a “docker” container.

Fully detailed installation instructions are included in the library, so please refer to the files:

Install_linux.txt

Install_Windows.txt

Install_macOS.txt

We also explain the process here (for the linux case):

It is advisable to install a python distribution like Anaconda (Python 3.7). Please proceed to the Anaconda download page (<https://www.anaconda.com/distribution/>) and follow the instructions according to your Operative System.

- Once Anaconda is correctly installed, open a bash/dash terminal and execute the following commands:

```
conda update conda
```

```
conda update anaconda
```

- Next, we create a conda environment with all the required libraries (Note that the next command is a single line)

```
conda create -n demo python=3.7.4 flask requests numpy ipython  
scikit-learn matplotlib tqdm pytorch-cpu torchvision seaborn  
transitions==0.6.9 pygraphviz==1.5 -c pytorch -c defaults -c  
conda-forge --yes
```

You may need some assistance from a System Manager if you fail to install the Python required libraries.

Uncompress the file D4_6.zip. In the D4_6 folder, you should find the following subfolder structure:

```
demo/
```

```
documentation_html/
```

```
input_data/
```

```
MMLL/
```


results/

- * **demo:** the folder where the execution scripts are.
- * **documentation_html:** the folder where the software documentation is. To browse it, just open the index file in it by double-clicking on it. The documentation will be shown in a web browser.
- * **input_data:** some small datasets are provided for running the demos. If you want to execute any demo with a larger dataset, you must download them from this link (https://drive.google.com/open?id=1NOMvmppt5qfGmGjA14hsdsTgB9KD7_Oz), but the provided datasets are enough to explore the Machine Learning Library usage.
- * **MMLL:** The MUSKETEER Machine Learning Library (POMs 4, 5 and 6).
- * **results:** some output figures are saved here. Also a subfolder with execution logs is available.

The installation process is complete, you may proceed with the demos execution, as described in the next Section.

9 Execution of the demos

In this section we describe the steps needed to test the developed library in some selected Machine Learning Tasks. All the tests and demos described here will use a local communication mechanism among processes in the same machine, to ease the executions. The communications library using the IBM cloud has already been partially tested and the library is ready to easily replace the local communications by the IBM cloud based communication.

To facilitate those tests, we provide two execution alternatives⁵:

- **Simple execution: single terminal:** all needed processes to complete the machine learning tasks are executed from a single terminal. In this case, the detailed messages are hidden, to produce a clearer result in the terminal.
- **Full detail execution: different terminals.** In this case, every participating process will be run on a different terminal, such that a detailed list of messages are shown in every screen. This option is useful for easy monitoring of the operations and protocols behaviour.

⁵ In both cases, the communications server process must be run in a separate terminal, as it will be described later.

Both execution options rely on the execution of the same processes and libraries, so their result is completely equivalent. The unique difference between them is the ease of execution and the amount of messages shown on the screen.

The scripts needed to execute the demos are included in the library, named as (choose accordingly your Operative System):

demo_linux.txt

demo_Windows.txt

demo_macOS.txt

You can easily copy-paste the scripts from those files, to ease the launch of experiments. There are some extra advices in those files, worth reading before launching the demos.

9.1 Simple execution

For illustration purposes, we use here the Windows OS. We open two terminals, activate the conda environment and move to the demo folder.

We execute in the first terminal:

```
python local_flask_server.py
```

and in the second:

```
demo_POM6_KR_sinc1D.bat
```

We observe the results in the terminals:

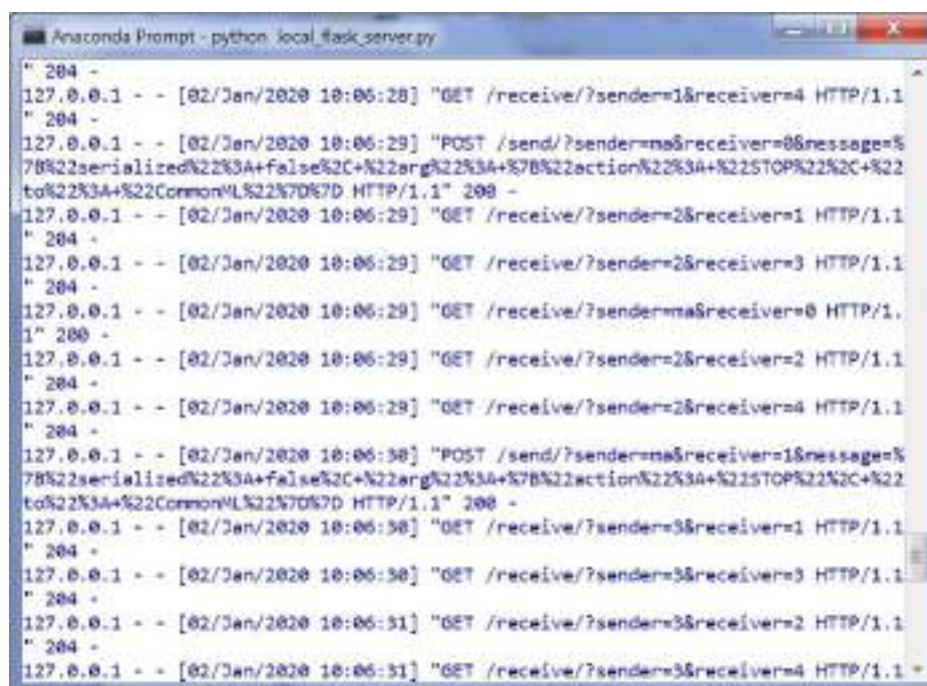
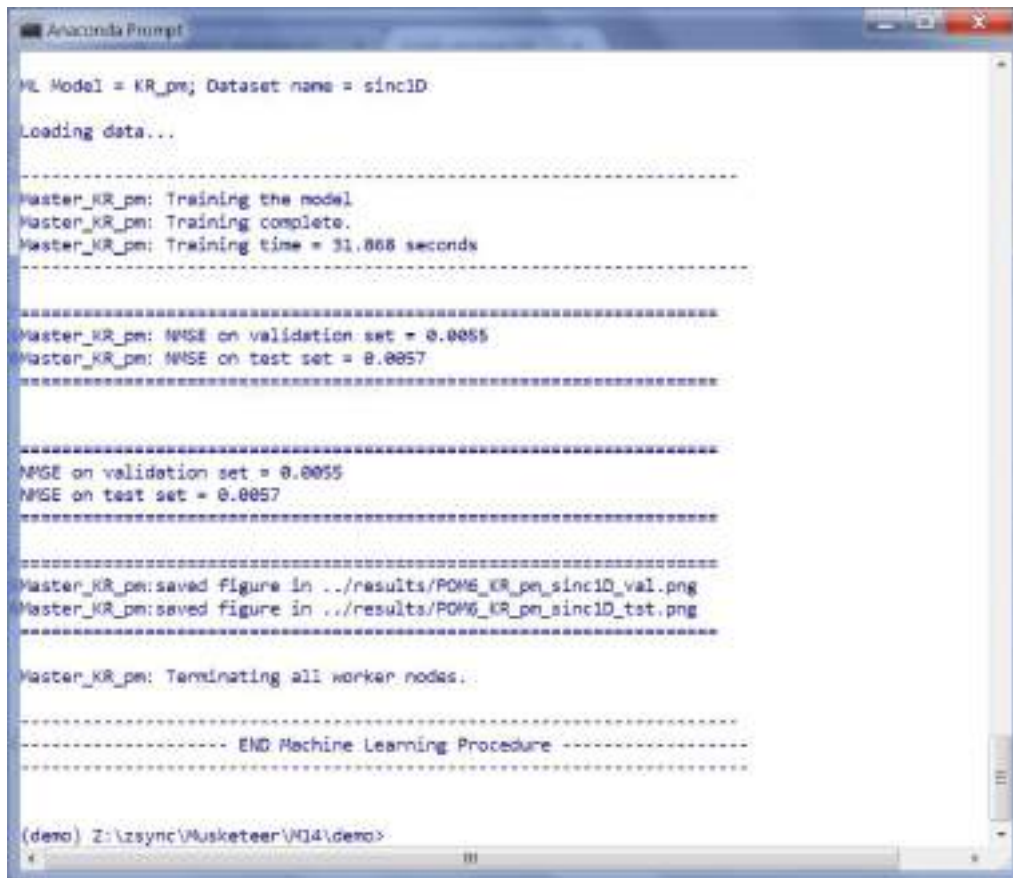


Figure 22: The local communications terminal (Flask Server)



```
ML Model = KR_pm; Dataset name = sinc1D
Loading data...

-----
Master_KR_pm: Training the model
Master_KR_pm: Training complete.
Master_KR_pm: Training time = 31.868 seconds
-----

-----
Master_KR_pm: NMSE on validation set = 0.0055
Master_KR_pm: NMSE on test set = 0.0057
-----

-----
NMSE on validation set = 0.0055
NMSE on test set = 0.0057
-----

-----
Master_KR_pm: saved figure in ../results/POM6_KR_pm_sinc1D_val.png
Master_KR_pm: saved figure in ../results/POM6_KR_pm_sinc1D_tst.png
-----

Master_KR_pm: Terminating all worker nodes..

-----
----- END Machine Learning Procedure -----
-----

(demo) Z:\zsync\Musketeer\WQ4\demo>
```

Figure 23: The demo execution of a Kernel Regression under POM6 in a single terminal

9.2 Full detail execution

For illustration purposes, we use here the Linux OS. We open seven terminals, activate the conda environment and move to the demo folder.

We execute in the first terminal:

```
Python3 local_flask_server.py
```

And observe:

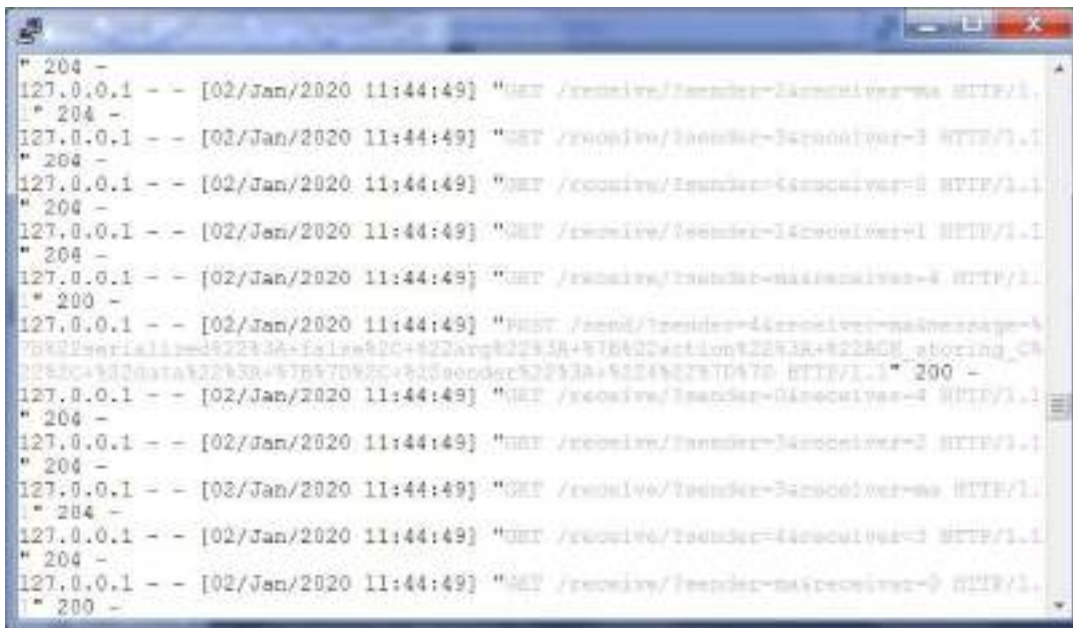


Figure 24: The local communications terminal (Flask Server)

And in the rest of terminals:

Terminal 2: `python3 pom6_KR_worker.py --id 0 --dataset sinc1D --verbose True`

Terminal 3: `python3 pom6_KR_worker.py --id 1 --dataset sinc1D --verbose True`

Terminal 4: `python3 pom6_KR_worker.py --id 2 --dataset sinc1D --verbose True`

Terminal 5: `python3 pom6_KR_worker.py --id 3 --dataset sinc1D --verbose True`

Terminal 6: `python3 pom6_KR_worker.py --id 4 --dataset sinc1D --verbose True`

Terminal 7: `python3 pom6_KR_master.py --dataset sinc1D --verbose True`

Every worker produces:

```
Worker_KR_pm 0: communicating through localflask
Worker_KR_pm 0: Creating worker object
WorkerNode 0: Loading cooms
WorkerNode 0: Loading Data Connector
WorkerNode 0: Initiated
WorkerNode_KR_pm 0: loading data from ../input_data/siclid_demonstrator_data.pkl
...
WorkerNode 0: Data loaded, 100 patterns with 1 features
Worker_KR_pm 0: Creating ML modal of type KR_pm
POM6_CommonML Worker 0: creating FSM
WorkerNode 0: Created CommonML model
KR_pm Worker 0: creating FSM
WorkerNode_KR_pm 0: Created KR_pm model
Worker_KR_pm 0 is running...
WorkerNode_KR_pm 0: running KR_pm ...
POM6_CommonML Worker 0 received sending_C from na
KR_pm Worker 0: sent ACK storing_C
KR_pm Worker 0: WAITING for instructions...
POM6_CommonML Worker 0 received compute_KTK from na
KR_pm Worker 0: sent ACK sending_KTK
KR_pm Worker 0: WAITING for instructions...
POM6_CommonML Worker 0 received STOP from na
POM6_CommonML Worker 0: terminated by Master
Worker_KR_pm 0: EXIT
```

Figure 25: The demo execution of a Kernel Regression under POM6 in full detail (WorkerNode)

And the master produces:

```
-----
Master_KR_pm: Training the model
KR_pm_Master: Starting training
KR_pm_Master: broadcasted C to all Workers
KR_pm_Master na received ACK storing_C from 4
KR_pm_Master: received ACK from 4: ACK storing_C
KR_pm_Master na received ACK storing_C from 0
KR_pm_Master: received ACK from 0: ACK storing_C
KR_pm_Master na received ACK storing_C from 1
KR_pm_Master: received ACK from 1: ACK storing_C
KR_pm_Master na received ACK storing_C from 3
KR_pm_Master: received ACK from 3: ACK storing_C
KR_pm_Master na received ACK storing_C from 2
KR_pm_Master: received ACK from 2: ACK storing_C
KR_pm_Master: WAITING for instructions...
KR_pm_Master: broadcasted compute_KTK to all Workers
KR_pm_Master na received ACK sending_KTK from 4
KR_pm_Master: received ACK from 4: ACK sending_KTK
KR_pm_Master na received ACK sending_KTK from 1
KR_pm_Master: received ACK from 1: ACK sending_KTK
KR_pm_Master na received ACK sending_KTK from 3
KR_pm_Master: received ACK from 3: ACK sending_KTK
KR_pm_Master na received ACK sending_KTK from 0
KR_pm_Master: received ACK from 0: ACK sending_KTK
KR_pm_Master na received ACK sending_KTK from 2
KR_pm_Master: received ACK from 2: ACK sending_KTK
KR_pm_Master: WAITING for instructions...
KR_pm_Master: Training is done
Master_KR_pm: Training complete.
Master_KR_pm: Training time = 2.1299 seconds
-----

Master_KR_pm: NMSE on validation set = 0.0055
Master_KR_pm: NMSE on test set = 0.0057
-----
```

Figure 26: The demo execution of a Kernel Regression under POM6 in full detail (MasterNode)

10 Software documentation (sample)

The documentation of the software is provided in html format along with the code. The documentation has been generated with Sphinx⁶, and it will be maintained and expanded as the software project grows. We include in what follows some sample pages from that documentation, but the interested reader should load into any web browser the **index.html** file provided in the **documentation_html/** folder.



The screenshot shows a web page for the 'Musketeer ML Library'. On the left is a navigation sidebar with the title 'Musketeer Machine Learning Library' and a 'Navigation' section containing links for 'Nodes', 'Machine Learning Models', 'Communications', 'Cryptography libraries', and 'Data Connectors'. Below this is a 'Quick search' box with a 'Go' button. The main content area is titled 'Musketeer ML Library' and contains a 'Contents:' section with a tree structure of links: 'Nodes' (with sub-links for 'Master Node', 'Worker Node', and 'Crypto Node'), 'Machine Learning Models' (with sub-links for 'Common', 'Privacy Operation Mode 4', and 'Privacy Operation Mode 5'), and under 'Privacy Operation Mode 4' and '5', there are sub-links for 'Common', 'Cross-Correlation', 'Linear Regression', 'Logistic Classifier', 'Clustering (Kmeans)', and 'Kernel Regression'.

⁶ sphinx-doc.org

- [Privacy Operation Mode 6](#)
 - [Common](#)
 - [Cross-Correlation](#)
 - [Ridge Regression](#)
 - [Logistic Classifier](#)
 - [Multiclass Logistic Classifier](#)
 - [Clustering \(Kmeans\)](#)
 - [Kernel Regression](#)
 - [Budget Support Vector Machine](#)
- [Communications](#)
 - [Local Communications](#)
- [Cryptography libraries](#)
 - [Crypto BCP](#)
- [Data Connectors](#)
 - [Load data from local file](#)

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

Musketeer Machine Learning Library

Navigation

Contents:

Nodes

- [Master Node](#)
- [Worker Node](#)
- [Crypto Node](#)

[Machine Learning Models](#)

[Communications](#)

[Cryptography libraries](#)

[Data Connectors](#)

Quick search

Master Node

Master node object

```
class nodes.MasterNode.MasterNode(pom, master_address, workers_addresses,  
comms, dc, logger, verbose=False, **kwargs)
```

Bases: **object**

This class represents the main process associated to the Master Node, and serves to coordinate the training procedure under POMs 4, 5 and 6

Creates a **MasterNode** instance.

- Parameters:**
- **pom** (*integer*) – the selected POM
 - **master_address** (*string*) – address of the master node
 - **workers_addresses** (*list of strings*) – list of the addresses of the workers
 - **comms** (*comms object instance*) – object providing communications
 - **dc** (*DataConnector object instance*) – data connector used by master and workers to read the data
 - **logger** (*class:logging.Logger*) – logging object instance
 - **verbose** (*boolean*) – indicates if messages are print or not on screen
 - ****kwargs** (*Arbitrary keyword arguments.*) –

Optional or POM dependant arguments

- Parameters:**
- **er** (*encryption object instance*) – the encryption library to be used in POMs 4 and 5
 - **cryptonode_address** (*string*) – address of the crypto node
 - **Nmaxiter** (*integer*) – Maximum number of iterations during learning
 - **NC** (*integer*) – Number of centroids
 - **regularization** (*float*) – Regularization parameter
 - **classes** (*list of strings*) – Possible class values in a multiclass problem
 - **balance_classes** (*Boolean*) – If True, the algorithm takes into account unbalanced datasets
 - **C** (*array of floats*) – Centroids matrix
 - **nf** (*integer*) – Number of bits for the floating part
 - **N** (*integer*) – Number of
 - **fsigma** (*float*) – factor to multiply standard sigma value = $\sqrt{\text{Number of inputs}}$
 - **normalize_data** (*Boolean*) – If True, data normalization is applied, irrespectively if it has been previously normalized

`create_model_Master(model_type, C=None, fsigma=None)`

Create the model object to be used for training at the Master side.

- Parameters:**
- **model_type** (*str*) – Type of model to be used
 - **C** (*numpy array*) – centroids matrix (Optional)
 - **fsigma** (*float*) – Sigma multiplying factor (Optional)

display(*message*)
 Save message to log file and display on screen if *verbose=True*.

Parameters: *message* (*str*) – string message to be shown/logged

fit()
 Train the Machine Learning Model

gen_crypto_keys()
 Create Cryptographic keys, under POM 5

get_crypto_keys()
 Obtain Cryptographic keys, under POM 4

get_encrypted_data()
 Obtain Encrypted data from workers, under POM 4

load_data(*add_bias=True*)
 Load data to be used for validation/testing. The access to the data is provided by the Data Connector.

Parameters: *add_bias* (*boolean*) – If true, it adds a column of ones to the input data matrix

predict(*X*)
 Use the trained Machine Learning Model to predict new output

Parameters: *X* (*numpy array*) – Input data matrix

Musketeer Machine Learning Library

Navigation

Contents:

Nodes

Machine Learning Models

- Common
- Privacy Operation Mode 4
- Privacy Operation Mode 5
- Privacy Operation Mode 6
 - Common
 - Cross-Correlation
 - Ridge Regression
 - Logistic Classifier
 - Multiclass Logistic Classifier
 - Clustering (Kmeans)
 - Kernel Regression
 - Budget Support Vector

Budget Support Vector Machine

Budget Support Vector Machine (public model) under POM6

```
class models.POM6.BSVM_pn.BSVM_pn.BSVM_pn_Master(master_address,
workers_addresses, comms, logger, verbose=False, NC=None, Nixdeter=None,
regularization=a, C=None, fsigma=None, Xval_b=None, yval=None)
```

Bases: `models.Common_to_all_POMs.Common_to_all_POMs`

This class implements the Budget Support Vector Machine (public model), run at Master node. It inherits from `Common_to_all_POMs`,

Create a `BSVM_pn_Master` instance.

- Parameters:**
- **master_address** (*string*) – address of the master node
 - **workers_addresses** (*list of strings*) – list of the addresses of the workers
 - **comms** (*comms object instance*) – object providing communications
 - **logger** (*class logging.Logger*) – logging object instance
 - **verbose** (*boolean*) – indicates if messages are print or not on screen
 - ****kwargs** (*Arbitrary keyword arguments.*) –

Optional or POM dependant arguments

`train_Master()`

This is the main training loop, it runs the following actions until the stop condition is met:

- Update the execution state
- Process the received packets
- Perform actions according to the state

Parameters: None –

```
class models.POMs.BSVM_pm.BSVM_pm.BSVM_pm_Worker(master_address,  
worker_address, workers_addresses, model_type, comms, logger, verbose=False,  
Xtr_b=None, ytr=None)
```

Bases: `MMLL.models.Common_to_all_POMs.Common_to_all_POMs`

Class implementing Budget Support Vector Machine (public model), run at Worker

Create a `BSVM_pm_Worker` instance.

- Parameters:**
- `master_address` (*string*) – address of the master node
 - `worker_address` (*string*) – id of this worker
 - `workers_addresses` (*list of strings*) – list of the addresses of the workers
 - `model_type` (*string*) – type of ML model
 - `comms` (*comms object instance*) – object providing communications
 - `logger` (*class:logging.Logger*) – logging object instance
 - `verbose` (*boolean*) – indicates if messages are print or not on

11 Conclusions

In this deliverable (D4.6) we have presented a preliminary version of the MUSKETEER Machine Learning Library under POMs 4, 5 and 6 (MMLL V1.0). We have implemented Linear models, Clustering (Kmeans) and Kernel methods. This version 1.0 of the library uses the local communications library and has been implemented using the final code structure. The algorithms and code still need to be optimized, and some extra algorithms need to be developed; these tasks will be carried out during the next months and delivered in M30 (D4.7) in the form of version 2.0 of the library (MMLL V2.0).

12 References

- [Kernel Regression] https://en.wikipedia.org/wiki/Kernel_regression
[Logistic Classifier] https://en.wikipedia.org/wiki/Logistic_regression#Logistic_model
[Pearson_Corr] https://en.wikipedia.org/wiki/Correlation_coefficient
[Ridge_Regression] https://en.wikipedia.org/wiki/Tikhonov_regularization
[Sphinx] <https://sphinx-doc.org>
[Support Vector Machine] https://en.wikipedia.org/wiki/Support-vector_machine