H2020 – ICT-13-2018-2019

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

Grant No 824988

D4.5 Machine Learning Algorithms over Federated Operation Modes – Final version

May 21

Imprint

Contractual Date of Delivery to the EC: **31 May 2021**

Author(s): **Roberto Díaz (Tree Technology), Marcos Fernández (Tree Technology), Jaime Medina (Tree Technology)**

Participant(s): **TREE, UC3M, COMAU, B3D**
Reviewer(s): **Chiara Napione (COMAU), Joao Correia (B3D)**

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

Work package: **WP4**
Dissemination level: Public
Version: **1.0**

Contact: **Jaime Medina – jaime.medina@treetk.com**
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

Executive Summary

The main objective of this deliverable is to provide the final version of the MUSKETEER Machine Learning Library (MMLL) under Privacy Operation Modes (POMs) 1, 2 and 3.

The final version of the library integrates a wrapper for using the cloud communications developed by IBM, enabling the communication between nodes at different machines. This provides an additional feature to the first release in which the communication among the nodes was restricted to processes running on the same machine or under the same private network.

As far as the implementation of the algorithms is concerned, two additional variants are included within this release: The Federated Budget Support Vector Machine (FBSVM), available for the three POMs, and the Distributed Support Vector Machine (DSVM), available only for POM 1 due to impossibility of meeting the more restrictive requirements of the other two POMs. Regarding the K-means algorithm, an additional naïve sharding initialization has been included here in order to ensure the privacy requirements required at the workers which, for the previous version, could not be met under some particular circumstances. The implementation of the Neural Networks has been extended to include a model averaging approach as well as the gradient averaging already included in the first release. This translates into a more flexible configuration of the networks, enabling the user to solve both regression as well as classification problems. Furthermore, thanks to the inclusion of the model averaging, the performance of the models has been significantly improved and the communication overload has been reduced, resulting in faster training times.

Some pre-processing capabilities have been included in this version of the library as a result of the work covered in D4.3. Normalization strategies, image as well as natural language processing or dimensionality reduction techniques are available. Finally, a complete set of demos and the corresponding instructions as well as a full documentation of the library is available at the public repository.

Document History

Version	Date	Status	Author	Comment
0.1	12 April 2021	Internal writing	Marcos Fernández	Structure and table of contents
0.2	27 April 2021	Internal writing	Marcos Fernández	Sections 4, 5, 6 and 7
0.3	29 April 2021	Internal writing	Marcos Fernández	Executive summary and sections 1 and 2
0.4	3 May 2021	For internal review	Marcos Fernández, Roberto Díaz	Version for internal review
0.5	12 May 2021	Final version after review	Chiara Napione, Joao Correia, Marcos Fernández, Roberto Díaz	Final version after internal review
1.0	13 May 2021	Final review	Mark Purcell and Gal Weiss	

Table of Contents

List of Figures

List of Acronyms and Abbreviations

Abbreviation	Definition
API	Application Programming Interface
DC	Data Connector
DSVM	Distributed Support Vector Machine
FBSVM	Federated Budget Support Vector Machine
FML	Federated Machine Learning
IRWLS	Iteratively ReWeighted Least Squares
JSON	JavaScript Object Notation
KM	Kernel Methods
MMLL	MUSKETEER Machine Learning Library
MN	Master Node
NN	Neural Network
PCA	Principal Component Analysis
POM	Privacy Operation Mode
SVM	Support Vector Machine
WN	Worker Node

1 Introduction

1.1 Purpose

MUSKETEER proposes a collection of POMs, each one describing a potential scenario with different privacy preserving demands, but also with different computational, communication, storage and accountability features.

This deliverable describes the additional features included in the MMLL under each of the Federated Collaborative Privacy Operation Modes. Under these modes, data never leaves the data owners' facilities, since training takes place under the Federated Machine Learning paradigm, where the model is transferred among the users, and everyone contributes by locally updating the model, using their data. The resulting model is unique, common to all the users, but in some POMs not all users get access to the trained model in unencrypted form.

Specifically, the POMs that will be addressed are:

- POM1 (ARAMIS): Here data cannot leave the facilities of each data owner, and the predictive models are transferred without encryption. It is intended for partners who want to collaborate to create a predictive model that will be public and visible among the different MUSKETEER clients and the main process.
- POM2 (ATHOS): The same schema as ARAMIS but using homomorphic encryption with a single private key in every client. The server can operate in the encrypted domain without having access to the unencrypted model. This schema is designed for use cases where the same data owner has data allocated in different locations, data cannot be moved for legal/architectural reasons, and the predictive model will be public for the different clients and remain private for the main process.
- POM3 (PORTHOS): Extension of ATHOS, where different data owners use different private keys for homomorphic encryption, and a re-encryptor on the server side can transform encrypted models among different private keys.

1.2 Related Documents

This deliverable is the continuation of D4.4, which detailed the first version of the MUSKETEER Machine Learning Library. Additionally, it takes inputs coming from D4.3 related to the

different pre-processing techniques ready to be used from within the library, as well as from D3.2 with respect to the cloud communications platform.

The actual code implementation and resources are available at https://github.com/Musketeer-H2020/MMLL.

1.3 Document Structure

The remainder of this document is structured as follows:

Section 2 describes the MUSKETEER Machine learning library, including the exposed API to be used by the clients as well as the wrapper for the cloud communications.

Section 3 corresponds to the algorithms implemented under each of the Privacy Operation Modes.

Installation details and dependencies are detailed in section 4.

Section 5 includes some examples of use for the clients.

The execution of the different available demos is presented in section 6.

Finally, section 7 includes the general conclusions of the work reported in this deliverable.

2 The Machine Learning Library

This deliverable aims at providing the final version of the Machine Learning Library to be used in MUSKETEER project under POMs 1, 2 and 3. It includes an integration with a cloud-based communication library developed by IBM under WP3, allowing developers to build solutions for the training of different algorithms in federated scenarios.

The architecture of the implemented solution is described in Figure 1. There are two core components in a learning process:

- 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.

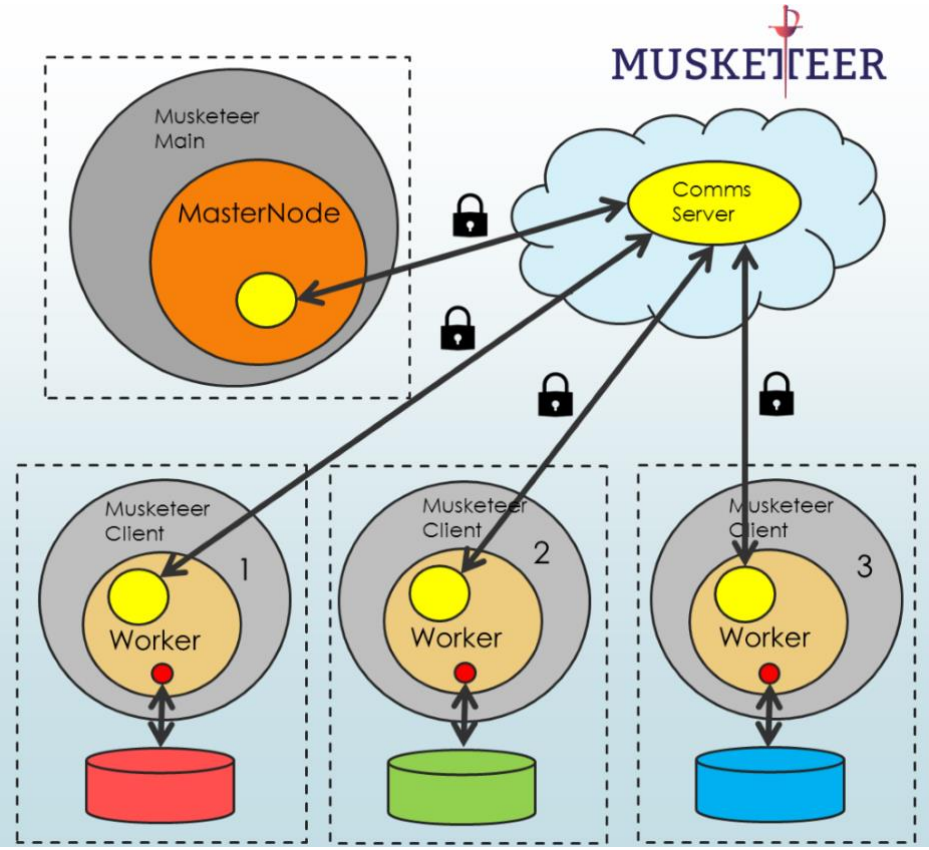

Figure 1. Detailed process interactions in a MUSKETEER learning process.

The main process and the clients can be running on different machines and/or networks since they use the cloud communication service to interchange messages.

2.1 API

This section describes the MMLL API exposed to the clients, both at master node as well as worker node.

2.1.1 MasterNode

class MMLL.nodes.MasterNode.**MasterNode**(*pom*, *comms*, *logger*, *verbose=False*, ***kwargs*)

Bases: **MMLL.Common_to_all_objects.Common_to_all_objects**

This class represents the main process associated to the Master Node, and serves to coordinate the training procedure under the different POMs.

Creates a **MasterNode** instance.

Parameters

- **pom** (*int*) – The selected Privacy Operation Mode.
- **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** (*Variable keyword arguments.*)

- **check_data_at_workers**(*input_data_description*, *target_data_description*)
  - Checking data at workers. Returns None if everything is OK.
  - **Parameters**
    - **input_data_description** (*dict*) – Description of the input features.
    - **target_data_description** (*dict*) – Description of the targets.
  - **Returns**
    - **err** (*string*) – Error message, if any.
    - **bad_workers** (*list*) – List of workers with bad data.
- **compute_statistics**(*X*, *y*, *stats_list*)
  - Compute statistics of given data.

- **Parameters**
  - **X** (*list of lists or numpy array*) – Input data, one pattern per row.
  - **y** (*list of lists or numpy array*) – Target data, one target per row.
- **Returns**
  - **stats_list** – The list of statistics that have to be computed (rxy, meanx, medianx, npatterns, stdx, skewx, kurx, perc25, perc75, staty).
- **Return type**
  - dict

- `create_model_Master`(*model_type*, *model_parameters=None*)
  - Create the model object to be used for training at the Master side.
  - **Parameters**
    - **model_type** (*str*) – Type of model to be used.
    - **model_parameters** (*dict*) –
    - Parameters needed by the different models, for example it may contain:
      - **Nmaxiter** (*int*) – Maximum number of iterations during learning.
      - **NC** (*int*) – Number of centroids.
      - **regularization** (*float*) – Regularization parameter.
      - **C** (*array of floats*) – Centroids matrix.
      - **nf** (*int*) – Number of bits for the floating part.
      - **fsigma** (*float*) – Factor to multiply standard sigma value = sqrt(Number of inputs).
      - **normalize_data** (*Boolean*) – If True, data normalization is applied, irrespectively if it has been previously normalized.
- `data2num_transform_workers`(*input_data_description*)
  - Convert data to numerical vector.
  - **Parameters**
    - **input_data_description** (*dict*) – Description of the input features.
  - **Returns**

- **model** (*transformation model*) – Model to transform data.
- **new_input_data_description** (*dict*) – New dictionary describing the input data.
- **worker_errors** (*dict*) – Dictionary containing the errors (if any) for the different workers.

- `data2num_transform_workers_V`(*input_data_description*)
  - Convert data to numerical vector in vertical partitioning.
  - **Parameters**
    - **input_data_description** (*dict*) – Description of the input features.
  - **Returns**
    - **model** (*transformation model*) – Model to transform data.
    - **new_input_data_description** (*dict*) – New dictionary describing the input data.
    - **worker_errors** (*dict*) – Dictionary containing the errors (if any) for the different workers.
- `deep_learning_transform_workers`(*data_description*)
  - Convert images to numerical vector using Deep Learning.
  - **Parameters**
    - **data_description** (*dict*) – Description of the input features.
  - **Returns**
    - **model** (*DL model*) – The DL model to apply to future data.
    - **new_input_data_description** (*dict*) – Updated description of the input features.
    - **worker_errors** (*list of string*) – List of errors while preprocessing data at workers.
- `fit`(*Xval=None*, *yval=None*, *selected_workers=None*)
  - Train the Machine Learning Model
  - **Parameters**
    - **Xval** (*list of lists or numpy array*) – Validation data, one pattern per row.
    - **yval** (*list of lists or numpy array*) – Validation targets, one target per row.

- **selected_workers** (*list of ids*) – List of selected workers to operate with.

- `get_data_value_aposteriori`(*Xval*, *yval*, *baseline_auc=0*)
  - Obtain "A posterior" Data Value estimation.
  - **Parameters**
    - **Xval** (*list of lists or numpy array*) – Validation data, one pattern per row.
    - **yval** (*list of lists or numpy array*) – Validation targets, one target per row.
    - **baseline_auc** (*float*) – Minimum value of AUC.
  - **Returns**
    - **dv** (*list*) – List of data value estimation values for each worker.
    - **best_workers** (*list*) – List of strings with the worker addresses.
- `get_data_value_apriori`(*Xval*, *yval*, *stats_list*)
  - Obtain "A priori" Data Value estimation.
  - **Parameters**
    - **Xval** (*list of lists or numpy array*) – Validation data, one pattern per row.
    - **yval** (*list of lists or numpy array*) – Validation targets, one target per row.
    - **stats_list** (*list of string*) – The list of statistics that have to be computed (rxy, meanx, medianx, npatterns, stdx, skewx, kurx, perc25, perc75, staty).
- `get_feat_freq_transformer`(*data_description*, *Max_freq*, *NF*)
  - Get features frequency from all workers, generate transformer and transform data at workers.
  - **Parameters**
    - **data_description** (*dict*) – Description of the input features.
    - **Max_freq** (*float*) – Maximal allowed frequency to select a word.
    - **NF** (*int*) – Number of features to retain.
  - **Returns**
    - **feature_extractor** (*object*) – Feature extractor model.

- **new_input_data_description** (*dict*) – Updated description of the input features.

- `get_model`()
  - Returns the ML model as an object, if it is trained, returns None otherwise.
  - **Parameters**
    - **None** –
  - **Returns**
    - **model** – Machine learning model if it has been trained, None otherwise.
  - **Return type**
    - ML model
- `get_statistics_workers`(*stats_list*)
  - Get the statistics from the workers.
  - **Parameters**
    - **stats_list** (*list of string*) – The list of statistics that have to be computed (rxy, meanx, medianx, npatterns, stdx, skewx, kurx, perc25, perc75, staty).
  - **Returns**
    - **stats_dict_workers** – Statistics of every worker.
  - **Return type**
    - dict
- `get_task_alignment`(*Xval*, *yval*)
  - Compute the task alignment of the workers.
  - **Parameters**
    - **Xval** (*list of lists or numpy array*) – Validation data, one pattern per row.
    - **yval** (*list of lists or numpy array*) – Validation targets, one target per row.
  - **Returns**
    - **ta_dict** – Task alignment estimation of every worker.
  - **Return type**
    - dict
- `get_vocabulary_workers`(*data_description*, *init_vocab_dict=None*)
  - Get vocabulary from all workers.

- **Parameters**
  - **data_description** (*dict*) – Description of the input features.
  - **init_vocab_dict** (*dict*) – Initial vocabulary.
- **Returns**
  - **vocab** (*dict*) – Dictionary containing the vocabulary for every worker.
  - **global_df_dict_filtered** (*dict*) – Dictionary containing the vocabulary for every worker with every word appearing at least 10 times.

- `mn_ask_encrypter`()
  - Obtain encrypter from cryptonode, under POM 4.
  - **Parameters**
    - **None** –
- `mn_get_encrypted_data`(*use_bias=False*, *classes=None*)
  - Obtain encrypted data from workers, under POM 4.
  - **Parameters**
    - **use_bias** (*boolean*) – Indicates if a bias must be added.
    - **classes** (*list of string*) – List of possible classes.
- `normalizer_fit_transform_workers`(*input_data_description*, *transform_num='global_mean_std'*, *which_variables='num'*)
  - Adjust the normalizer parameters and transform the training data in the workers.
  - **Parameters**
    - **input_data_description** (*dict*) – Description of the input features.
    - **transform_num** (*string*) – Type of normalization of the numerical inputs. Binary inputs are not transformed, and categorical inputs are transformed using a one-hot encoding.
    - **which_variables** (*string*) – Indicates to which type of features we have to apply the normalization.
  - **Returns**
    - **model** – Object to normalize new data.
  - **Return type**
    - normalizer model

- `pca_fit_transform_workers`(*input_data_description*, *method*, *NF*)
  - Compute a PCA (Principal Component Analysis) transformation based on workers data.
  - **Parameters**
    - **input_data_description** (*dict*) – Description of the input features.
    - **method** (*string*) – Type of aggregation method to be used (direct/roundrobin).
    - **NF** (*int*) – Number of features to retain.
  - **Returns**
    - **pca_model** (*model*) – The trained PCA model.
    - **new_input_data_description** (*dict*) – The new description of the input features.
    - **workers_errors** (*list of string*) – The list of errors at every worker. Under normal operation it is an empty list.
- `ping_workers`()
  - Send ping message to workers to get address info.
  - **Parameters**
    - **None** –
- `preprocess_data_at_workers`(*prep*)
  - Send preprocessing object to workers.
  - **Parameters**
    - **prep** (*object*) – Preprocessing object.
  - **Returns**
    - **worker_errors** – Dictionary containing the errors (if any) for the different workers.
  - **Return type**
    - dict
- `preprocess_data_at_workers_V`(*prep*)
  - Send preprocessing object to workers for vertical partitioning.
  - **Parameters**
    - **prep** (*object*) – Preprocessing object.
  - **Returns**
    - **worker_errors** – Dictionary containing the errors (if any) for the different workers.

- **Return type**
  - dict

- `rank_features_gfs`(*Xval*, *yval*, *input_data_description*, *method*, *NF=None*, *stop_incr=None*)
  - Compute a greedy feature selection based on workers data.
  - **Parameters**
    - **Xval** (*list of lists or numpy array*) – Validation data, one pattern per row.
    - **yval** (*list of lists or numpy array*) – Validation targets, one target per row.
    - **input_data_description** (*dict*) – Description of the input features.
    - **method** (*string*) – Type of aggregation method to be used (direct/roundrobin).
    - **NF** (*int*) – Number of features to retain.
    - **stop_incr** (*float*) – Stop adding features if this tolerance value is reached.
  - **Returns**
    - **ranked_inputs** (*list of ints*) – List of ranked inputs.
    - **performance_evolution** (*list of floats*) – Model performance as a function of the selected inputs.
- `record_linkage_transform_workers`(*linkage_type='full'*)
  - Transform data at workers such that features are aligned.
  - **Parameters**
    - **linkage_type** (*string*) – Choose the type of linkage: full/join.
  - **Returns**
    - **input_data_description_dict** (*dict*) – New dictionary describing the input data.
    - **target_data_description_dict** (*dict*) – New dictionary describing the output data.
- `set_test_data`(*dataset_name*, *Xtst=None*, *ytst=None*)
  - Set data to be used for testing.
  - **Parameters**
    - **dataset_name** (*string*) – Dataset name.
    - **Xtst** (*numpy array*) – Test data, one pattern per row.

- **ytst** (*numpy array*) – Test targets, one target per row.

- `set_test_data_raw`(*dataset_name*, *Xtst=None*, *ytst=None*)
  - Set data to be used for testing.
  - **Parameters**
    - **dataset_name** (*string*) – Dataset name.
    - **Xtst** (*list of lists*) – Test data, one pattern per row.
    - **ytst** (*list of lists*) – Test targets, one target per row.
- `set_validation_data`(*dataset_name*, *Xval=None*, *yval=None*)
  - Set data to be used for validation.
  - **Parameters**
    - **dataset_name** (*string*) – Dataset name.
    - **Xval** (*numpy array*) – Validation data, one pattern per row.
    - **yval** (*numpy array*) – Validation targets, one target per row.
- `set_validation_data_raw`(*dataset_name*, *Xval=None*, *yval=None*)
  - Set data to be used for validation.
  - **Parameters**
    - **dataset_name** (*string*) – Dataset name.
    - **Xval** (*list of lists*) – Validation data, one pattern per row.
    - **yval** (*list of lists*) – Validation targets, one target per row.
- `stop_workers`()
  - Stop workers and start a new training.
  - **Parameters**
    - **None** –
- `terminate_workers`(*workers_addresses_terminate=None*)
  - Terminate selected workers.
  - **Parameters**
    - **workers_addresses_terminate** (*list of strings*) – List of addresses of workers that must be terminated. If the list is empty, all the workers will stop.

2.1.2 WorkerNode

class `MMLL.nodes.WorkerNode.`**`WorkerNode`**(*pom*, *comms*, *logger*, *verbose=False*, ***kwargs*)

Bases: **`MMLL.Common_to_all_objects.Common_to_all_objects`**

This class represents the main process associated to every Worker Node, and it responds to the commands sent by the master to carry out the training procedure under all POMs.

Create a **`WorkerNode`** instance.

Parameters

- **pom** (*int*) – The selected Privacy Operation Mode.
- **comms** (*comms object instance*) – Object providing communications.
- **logger** (class:*logging.Logger*) – Logging object instance.
- **verbose** (*boolean*) – Indicates if messages are printed or not on screen.
- ****kwargs** (*Arbitrary keyword arguments.*) –

- `create_model_worker`(*model_type*)
  - Create the model object to be used for training at the Worker side.
  - **Parameters**
    - **model_type** (*string*) – Type of model to be used.
- `get_model`()
  - Returns the ML model as an object, if it is trained, it returns None otherwise.
  - **Parameters**
    - **None** –
  - **Returns**
    - **model** – Machine learning model if it has been trained, None otherwise.
  - **Return type**
    - ML model
- `get_preprocessors`()
  - Returns the normalizer parameters and transform the training data in the workers.
  - **Parameters**
    - **None**

- **Returns**
  - **preprocessors** – Normalizer object.
- **Return type**
  - object

- `run`()
  - Run the main execution loop at the worker.
  - **Parameters**
    - **None**
- `set_test_data`(*dataset_name*, *Xtst=None*, *ytst=None*)
  - Set data to be used for testing.
  - **Parameters**
    - **dataset_name** (*string*) – The name of the dataset.
    - **Xtst** (*list of lists or numpy array*) – Test data, one pattern per row.
    - **ytst** (*list of lists or numpy array*) – Test targets, one target per row.
- `set_training_data`(*dataset_name*, *Xtr=None*, *ytr=None*, *input_data_description=None*, *target_data_description=None*)
  - Set data to be used for training.
  - **Parameters**
    - **dataset_name** (*string*) – The name of the dataset.
    - **Xtr** (*list of lists or numpy array*) – Training input data, one pattern per row.
    - **ytr** (*list of lists or numpy array*) – Training targets, one target per row.
    - **input_data_description** (*dict*) – Description of the input features.
    - **target_data_description** (*dict*) – Description of the targets.
- `set_validation_data`(*dataset_name*, *Xval=None*, *yval=None*)
  - Set data to be used for validation.
  - **Parameters**
    - **dataset_name** (*string*) – The name of the dataset.

- **Xval** (*list of lists or numpy array*) – Validation data, one pattern per row.
- **yval** (*list of lists or numpy array*) – Validation targets, one target per row.

2.2 Communications

This section contains the API definition for the wrapper object needed by the master and worker nodes in order to be able to use the IBM cloud communications. For more details on the latter please visit https://github.com/IBM/pycloudmessenger.

class `MMLL.comms.comms_pycloudmessenger.`**`Comms_master`**(*commsffl*)

Bases: **`object`**

This class provides an interface with the communication object, run at Master node.

Create a **`Comms_master`** instance.

Parameters

- **commsffl** (ffl.Factory.aggregator) – Object providing communication functionalities at Master for pycloudmessenger.

- `broadcast`(*message*, *receivers_list=None*)
  - Send a packet to a set of workers.
  - **Parameters**
    - **message** (*dict*) – Packet to be sent.
    - **receivers_list** (*list of strings*) – Addresses of the recipients for the message.
- `receive`(*timeout=1*)
  - Wait for a packet to arrive or until timeout expires.
  - **Parameters**
    - **timeout** (*float*) – Time to wait for a packet in seconds.
  - **Returns**
    - **message** – Received packet.
  - **Return type**
    - dict
- `receive_poms_123`(*timeout=10*)
  - Wait for a packet to arrive or until timeout expires. Used in POMs 1, 2 and 3.

- **Parameters**
  - **timeout** (*float*) – Time to wait for a packet in seconds.
- **Returns**
  - **packet** – Received packet.
- **Return type**
  - dict

- **roundrobin**(*message*, *receivers_list=None*)
  - Send a packet to a set of workers using the Round-robin protocol (ring communications).
  - **Parameters**
    - **message** (*dict*) – Packet to be sent.
    - **receivers_list** (*list of strings*) – Addresses of the recipients for the message.
- **send**(*message*, *destiny*)
  - Send a packet to a given destination.
  - **Parameters**
    - **message** (*dict*) – Packet to be sent.
    - **destiny** (*string*) – Address of the recipient for the message.

class MMLL.comms.comms_pycloudmessenger.**Comms_worker**(*commsffl*, *worker_real_name='Anonymous'*)

Bases: **object**

This class provides an interface with the communication object, run at Worker node.

Create a **Comms_worker** instance.

Parameters

- **commsffl** (**ffl.Factory.participant**) – Object providing communication functionalities for pycloudmessenger.
- **worker_real_name** (*string*) – Real name of the worker.

- **receive**(*timeout=0.1*)
  - Wait for a packet to arrive or until timeout expires.
  - **Parameters**
    - **timeout** (*float*) – Time to wait for a packet in seconds.
  - **Returns**

- **message** – Received packet.

- **Return type**
  - dict

- `receive_poms_123`(*timeout=10*)
  - Wait for a packet to arrive or until timeout expires. Used in POMs 1, 2 and 3.
  - **Parameters**
    - **timeout** (*float*) – Time to wait for a packet in seconds.
  - **Returns**
    - **packet** – Received packet.
  - **Return type**
    - dict
- `send`(*message*, *address=None*)
  - Send a packet to the master.
  - **Parameters**
    - **message** (*dict*) – Packet to be sent.
    - **address** (*string*) – Address of the recipient for the message.

`MMLL.comms.comms_pycloudmessenger.get_current_task_name`(*self*)

Function to retrieve the current task name from local disk.

Parameters

None –

Returns

task_name – The current task name currently created from the master.

Return type

string

3 Available algorithms

3.1 Deep Neural Networks

Deep learning architectures [Le Cun, 2015] (see Figure 2) such as recurrent neural networks [Razvan, 2013] or convolutional neural networks [Jiuxiang, 2018] are currently the state of art over a wide variety of fields including computer vision, speech recognition, natural language processing, audio recognition, machine translation, bioinformatics and drug design, where they have produced results comparable to and in some cases superior to human experts.

MMLL includes methods to train deep neural networks using gradient descent [Yang, 2019] or model averaging schemas [Konečný, 2016].

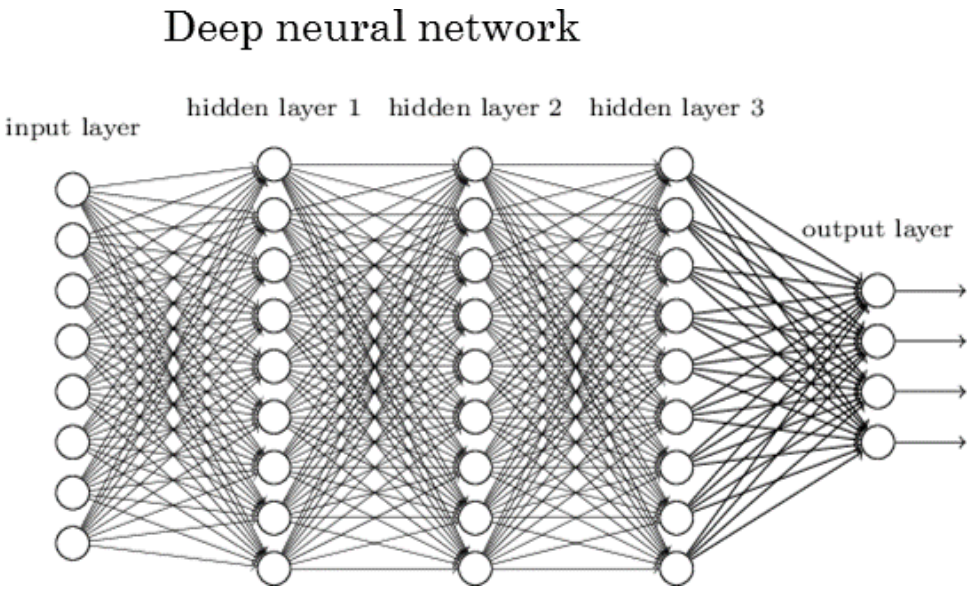

Figure 2. Deep Neural Network architecture (figure extracted from https://datawarrior.wordpress.com/2016/04/16/relevance-and-deep-learning/)

3.1.1 Neural Networks over POM 1 explained:

1-Initialization:

Every client:

- Load its dataset.

Main process:

- Initialize the neural network with random weights.

2-Iterative process:

Main process:

- Send the Neural Network to every client.

Every client:

- In the case of gradient descent:
    - Take a subset of training data and compute the gradients to optimize the weights using the back propagation algorithm.
    - Send to the main process the gradients.
- In the case of model averaging:
    - Train different epochs of neural network using the complete local dataset.
    - Send to the main process the neural network.

Main process:

- In the case of gradient descent:
    - Compute the global gradients adding the gradients of every client.
    - Update the Neural Network using a gradient descent step. There are several options:
        - Stochastic Gradient Descent: Updates every weight (w) using the following formula:
            - w = w - learning_rate * g
            - Where g is the average of the gradients obtained in every client.
        - Stochastic Gradient Descent with Momentum: Updates every weight using the following formula:
            - velocity=momentum*velocity-learning_rate * g
            - w = w + velocity
        - Nesterov Accelerated Gradient: Updates every weight using the following formula:
            - velocity = momentum*velocity -learning_rate*g
            - w = w + momentum*velocity - learning_rate * g
- In the case of model averaging:
    - Averages the received neural networks
    - Replaces the neural network with the current average of models.

- If the defined number of iterations is reached, send a signal to every client to finish the training.

3.1.2 Neural Networks over POM 2 explained:

POM2 uses the same schema as POM1 but using homomorphic encryption.

Every client in the initialization step loads the private and public key, while the main process loads just the public key.

The weights are sent to the main process encrypted with the public key and the main process updates the neural network in the encrypted domain.

Once the clients receive the Neural Network from the main process in the iterative process, they can decrypt it using the private key.

3.1.3 Neural Networks over POM 3 explained:

POM3 uses the same schema as POM2 but using proxy re-encryption.

3.2 Clustering (K-means)

This is the task of dividing the population or data into a number of groups such that data points in the same groups are more similar to other data points in the same group than those in other groups. In simple words, the aim is to segregate groups with similar characteristics and assign them into clusters. The current version of the library has implemented the K-means algorithm.

K-means [Jain_2010] clustering is a popular unsupervised machine learning algorithm. A cluster is a collection of data points aggregated with certain similarities.

The first step is to define the number k, which refers to the number of groups you need. A centroid is the imaginary or real location representing the centre of the cluster. Every data point is allocated to each of the clusters through reducing the predefined distance matrix.

In other words, the K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster (see Figure 3 for a graphical representation of clustering), while keeping the distance with the centroids as small as possible.

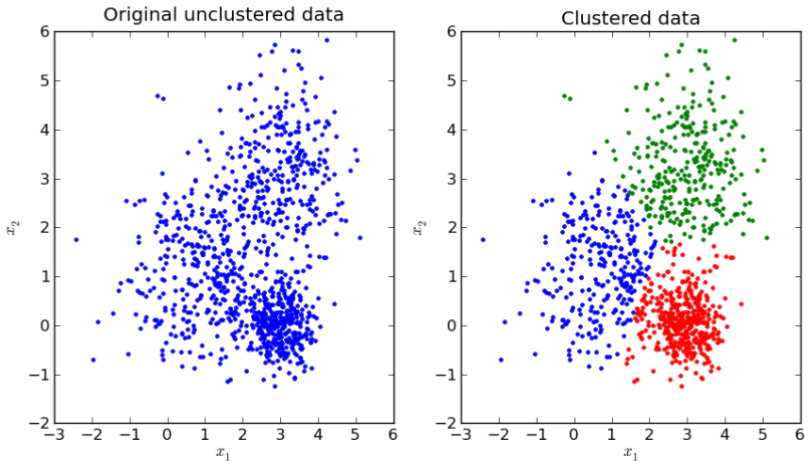

Figure 3. Example of clustering to divide data into three different groups.

To learn the centroids, we first need to initialize them. The K-means algorithm in data mining starts with a first group of randomly selected centroids, which are used as the beginning points for every cluster (although there are several alternatives in the literature to initialize them), and then performs iterative (repetitive) calculations to optimize the positions of the centroids.

The learning process stops when:

- The centroids have stabilized — there is no change in their values because the clustering has been successful.
- The defined number of iterations has been achieved.

3.2.1 K-Means over POM 1 explained:

The optimization process is based on the distributed K-means procedure used in the Spark MLlib library [Meng_2016].

1-Initialization:

Every client:

- Loads its dataset.
- Runs the Naïve Sharding algorithm to initialize centroids:
  - **Step 1**: Sum the attribute (column) values for each instance (row) of a dataset and prepend this new column of instance value sums to the dataset.
  - **Step 2**: Sort the instances of the dataset by the newly created sum column, in ascending order.
  - **Step 3**: Split the dataset horizontally into *k* equal-sized pieces, or shards.

    - **Step 4**: For each shard, sum the attribute columns (excluding the column created in step 1), compute its mean, and place the values into a new row; this new row is effectively one of the centroids used for initialization.
    - **Step 5**: Add each centroid row from each shard to a new set of centroid instances.
    - **Step 6**: Return this set of centroid instances to the calling function for use in the *k*-means clustering algorithm.
- Sends the subset of initial centroids to the MUSKETEER central node.

Main process:

- Collects the initial centroids from every client.

2-Iterative process:

Main process:

- Sends the centroids to every client.

Every client:

- Assigns each local data to the closest corresponding centroid, using the Euclidean distance.
- For each centroid, calculates the local mean of the values of all the points belonging to it.
- Sends to the main process the local mean of every centroids and the number of data belonging to every centroid.

Main process:

- Receives the local means from every client and compute the global mean of every centroid.
- Replaces every centroid by the global mean.
- Detects if the stop criteria has been reached:
    - The centroids have stabilized — the change in their values is lower than a threshold because the clustering has been successful.
    - The defined number of iterations has been achieved.
- If the stop criteria have been reached, sends a signal to every client to finish the training.

3.2.2 K-means over POM 2 explained:

POM2 uses the same schema but using homomorphic encryption.

Every client in the initialization step loads the private and public key, the main process loads just the public key.

The local means are sent to the main process encrypted with the public key and the main process updates the centroids in the encrypted domain.

Once the clients receive the centroids from the main process in the iterative process, they can decrypt it using the private key.

3.2.3 K-means over POM 3 explained:

POM3 uses the same schema but using proxy re-encryption.

3.3 Kernel Methods

Kernel Methods [Hofmann, 2008] comprise a very popular family of Machine Learning algorithms. The main reason of their success is their ability to easily adapt linear models to create non-linear solutions by using the well-known 'kernel trick', i.e., transforming the input data space onto a high dimensional one where inner product between projected vectors can be computed using a kernel function. KM shave proved their practical effectiveness by obtaining highly competitive results in many different tasks. Although some other approaches like those in the Deep Learning family have shown to outperform KMs in several specific tasks, the latter still present a good compromise between complexity and performance in many applications.

The current version of the library has implemented a Budgeted SVM algorithm.

The main idea behind an SVM is to create a hyperplane that separates two different classes of data while maximizing the margin (distance from the separating hyperplane to the closest pattern of every class, see Figure 4). Patterns that do not respect this margin distance or directly are wrongly classified are called Support Vectors (SVs).

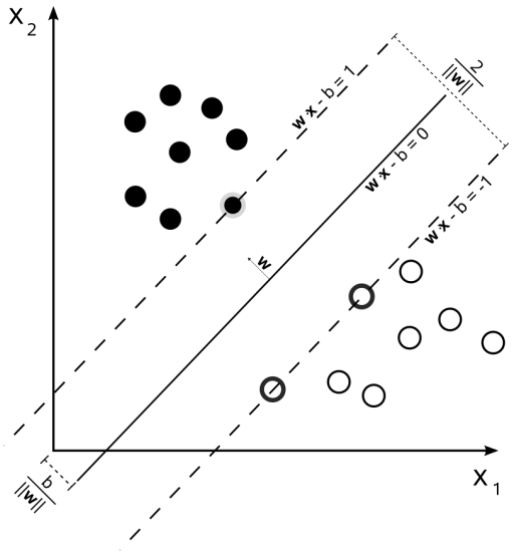

Figure 4. Maximum margin classifier (figure extracted from https://www.quora.com/Why-do-we-call-an-SVM-a-large-margin-classifier).

Most real-world problems are not linearly separable, so we need to somehow relax the restrictions. Soft margin classification [Cortes,1995] uses a hinge loss function that separates the training data while some examples are still inside the margin or in the wrong side of the hyperplane (see Figure 5 for a graphical representation of Hard and Soft margins).

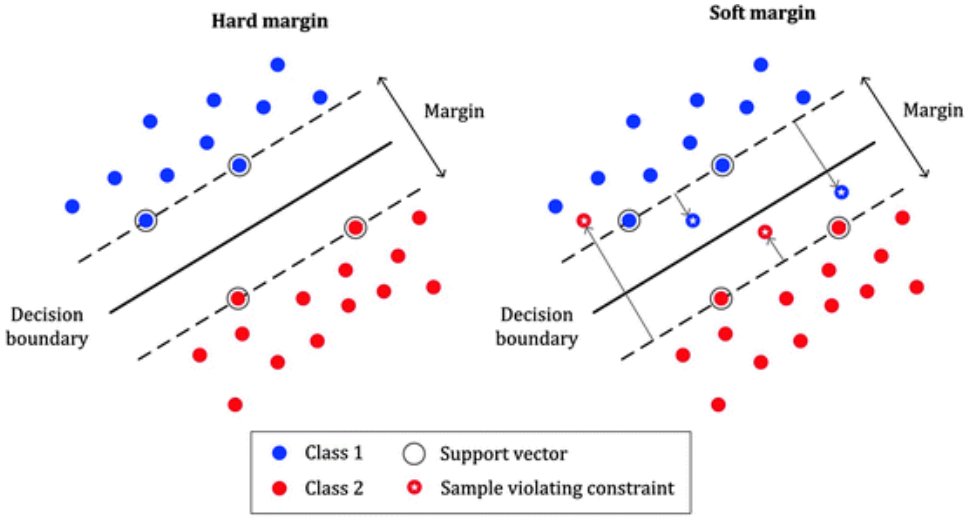

Figure 5. Hard margin vs soft margin (figure extracted from https://mc.ai/math-behind-svmsupport-vector-machine/).

The most common procedure to create a nonlinear classifier is by applying the 'kernel trick', [Scholkopf,2001] which maps the input space to a higher dimensional feature space where inner products are computed using a kernel function (see Figure 6).

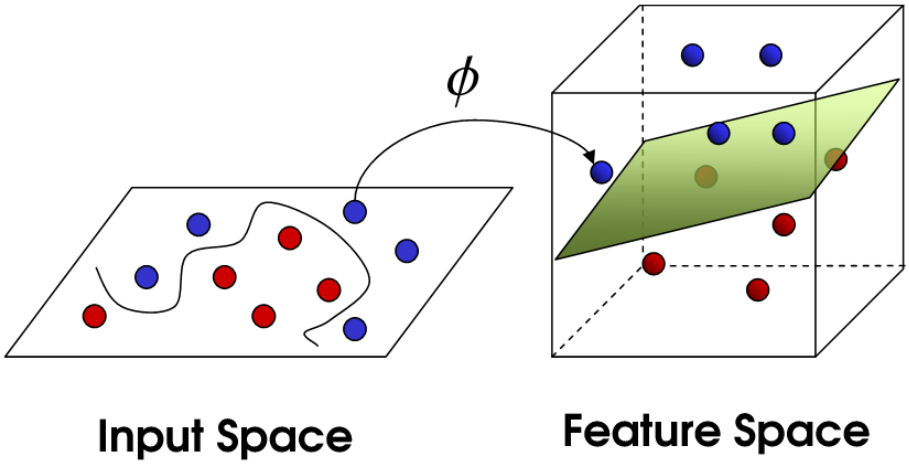

Figure 6. Kernel trick (figure extracted from https://es.switch-case.com/52732403).

Semiparametric (budgeted) models have been proposed to keep the classifier complexity under control [Diaz_2016], [Diaz_2018]. In these models, the dataset has the following form:

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)|\mathbf{x}_i \in \mathbb{R}^P, y_i \in \{-1, 1\}\}_{i=1}^n$$

The process has two different steps:

1 - Centroid selection:

A procedure to select m basis centroids C = {c_1, ..., c_m}.

2 – Optimization problem:

Then the following optimization problem is solved:

$$\min_{\boldsymbol{\beta}} \frac{1}{2}\boldsymbol{\beta}^{\mathbf{T}}\mathbf{K_c}\boldsymbol{\beta} + C\sum_{i=1}^{n}\xi_i$$
$$(\mathbf{K_c})_{i,j} = K(\mathbf{c}_i, \mathbf{c}_j), \forall i, j = 1, ..., m$$

$$\text{s.t.: } y_i(\boldsymbol{\beta}^T\mathbf{k}_i + b) \geq 1 - \xi_i, \forall i = 1, 2, ..., n$$
$$\xi_i \geq 0, \forall i = 1, 2, ..., n$$
$$(\mathbf{k}_i)_j = K(\mathbf{x}_i, \mathbf{c}_j)$$

where K is the kernel function $K(\mathbf{x}_i, \mathbf{x}_j) = exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2)$

which leads to a kernel model, whose size is the number of centroids:

$$f(\mathbf{x}) = \sum_{j=1}^{m} \beta_j K(\mathbf{x}, \mathbf{c}_j) + b$$

Iterative Re-Weighted Least Squares (IRWLS) is way to solve the IRWLS procedure rearranging the problem as follows:

$$\min_{\beta} \frac{1}{2}\boldsymbol{\beta}^{\mathbf{T}}\mathbf{K}_{\mathbf{c}}\boldsymbol{\beta} + \sum_i a_i e_i^2$$

$$e_i = y_i - \left(\sum_{j=0}^{m} \beta_j K(\mathbf{x}_i, \mathbf{c}_j) + b\right)$$

$$a_i = \begin{cases} 0, y_i e_i \leq 0 \\ \frac{C}{e_i y_i}, y_i e_i \geq 0 \end{cases}$$

The procedure works in a similar way to the full SVM, a_i is not a function of β and the algorithm works iteratively following a process of obtaining β and recalculating a_i using these weights until the weights converge.

The library counts with three different training methods of Budgeted SVMs:

- SVM: A training procedure based on gradient descent. Available on POM1, POM2 and POM3.
- FBSVM: A training procedure based on the aggregation of individual models. Available on POM1, POM2 and POM3.
- DSVM: A training procedure based on a distributed implementation of the IRWLS algorithm. Available on POM1.

The details of every implementation are listed below.

3.3.1 SVM

The first approximation is based on a K-means algorithm to obtain the centroids and a gradient descent algorithm to solve the optimization problem.

POM 1 explained:

1-Initialization:

Every client:

- Loads its dataset.

2-Centroid selection:

To select the centroids, the main process and clients make use of the K-means algorithm described in section 3.2.1. At the end, every client has a copy of these centroids.

3-Optimization procedure:

The process makes use of the gradient descent algorithm to solve the optimization problem.

Main process:

- Sends the weights to every client.

Every client:

- Computes the gradients of every training data in their respective datasets.
- Adds the gradients and sends the result to the main process.

Main process:

- Receives the gradients from every client.
- Updates the weights a step in the gradient descent algorithm.
- Detects if the stop criteria have been reached:
    - The weights have stabilized — the change in their values is lower than a threshold because the clustering has been successful.
    - The defined number of iterations has been achieved.
- If the stop criteria have been reached, it sends a signal to every client to finish the training.

POM 2 explained:

POM2 uses the same schema as POM1 but using homomorphic encryption.

Every client in the initialization step loads the private and public key, while the main process loads just the public key.

The weights or centroids are sent to the main process encrypted with the public key and the main process update the centroids or weights in the encrypted domain.

Once the clients receive the weights or centroids from the main process in the iterative process, they can decrypt them using the private key.

POM 3 explained:

POM3 uses the same schema as POM2 but using proxy re-encryption.

3.3.2 FBSVM

Federated Budgeted SVM is based on a random initialization of the centroids. To solve the optimization problem, every worker solves the IRWLS procedure to obtain the weights. The master node aggregates the weights and updates them.

POM 1 explained:

1-Initialization:

Every client:

- Loads its dataset.

2-Centroid selection:

To obtain m centroids, the main process initializes m^2 centroids randomly and computes the kernel matrix of every pair of centroids.

After that, it removes iteratively those which are more similar to each other (looking for in the kernel matrix the positions with higher value since the kernel function is a metric of similitude).

Once we have m centroids, the iterative process finishes.

3-Optimization procedure:

Main process:

- Sends the weights to every client.

Every client:

- Solves the IRWLS algorithm completely using their respective datasets.
- Sends the new weights to the master node.

Master node:

- Receives the weights from every client.
- Averages the weights.
- Updates the weights:
    - new_weights = old_weights + (mean_of_received_weights-old_weights)*μ
    - μ is a parameter to control the step of the update.
- Detects if the stop criteria have been reached:
    - The weights have stabilized — the change in their values is lower than a threshold because the clustering has been successful.
    - The defined number of iterations has been achieved.
- If the stop criteria have been reached, sends a signal to every client to finish the training.

POM 2 explained:

POM2 uses the same schema but using homomorphic encryption.

Every client in the initialization step loads the private and public key, while the main process loads just the public key.

The weights are sent to the main process encrypted with the public key and the main process update the centroids or weights in the encrypted domain.

Once the clients receive the weights from the main process in the iterative process, they can decrypt it using the private key.

POM 3 explained:

POM3 uses the same schema but using proxy re-encryption.

3.3.3 DSVM

Distributed SVM is based on a random initialization of the centroids. To solve the optimization problem, every worker solves distributed implementation of the IRWLS procedure that obtains the same result than in the centralized case. A detailed description of this implementation can be found in [Díaz,2016][Díaz,2018].

POM 1 explained:

1-Initialization:

Every client:

- Loads its dataset.

2-Centroid selection:

To obtain m centroids, the main process initializes m centroids randomly sampling from a uniform distribution.

The master node initializes the weights randomly.

3-Optimization procedure:

Main process:

- Sends the weights to every client.

Every client:

- Updates their datasets the variables a_i and e_i of the IRWLS algorithm.
- Computes the kernel matrices of the centroids with the local dataset.

- Computes the matrices that are necessary for the optimization problem (See [Díaz, 2016] [Díaz, 2018] for a more detailed description).
- Sends the matrices to the master node.

Master node:

- Receives the matrices.
- Solves the optimization problem and obtains new weights.
- Updates the weights.
- Detects if the stop criteria have been reached:
    - The weights have stabilized — the change in their values is lower than a threshold because the clustering has been successful.
    - The defined number of iterations has been achieved.
- If the stop criteria have been reached, sends a signal to every client to finish the training.

4 Installation

The repository containing the MMLL is publicly available at https://github.com/Musketeer-H2020/MMLL. Here, the interested reader can find a general overview of the library, including the available algorithms under the different privacy operation modes as well as the list of dependencies.

4.1 Setup

The MMLL was created as a Python package so that it can be easily installed and used by new developers who intend to train different algorithms under a federated scenario. A setup.py file is included at the root of the repository describing the package and handling the build and installation.

The MMLL requires a minimum version of Python 3.6. There are two possible ways for installing the library:

- Directly using pip from the command line. This is the preferred option:

```
pip install git+https://github.com/Musketeer-H2020/MMLL.git
```

- To install the library manually from the source file just type the following command from the root directory:

```
python setup.py install
```

4.2 Dependencies

MMLL has the following dependencies:

- transitions==0.6.9: a lightweight, object-oriented Python state machine implementation with many extensions.
- pygraphviz==1.5: PyGraphviz is a Python interface to the Graphviz graph layout and visualization package. With PyGraphviz you can create, edit, read, write, and draw graphs using Python to access the Graphviz graph data structure and layout algorithms.
- scipy: SciPy is a Python-based ecosystem of open-source software for mathematics, science and engineering.
- scikit-learn: open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for

model fitting, data pre-processing, model selection and evaluation, and many other utilities.

- matplotlib: comprehensive library for creating static, animated, and interactive visualizations in Python.
- tensorflow===2.4.0: TensorFlow is an end-to-end open-source platform for machine learning and deep learning. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML-powered applications.
- phe==1.4.0: a Python 3 library for Partially Homomorphic Encryption using the Paillier crypto system.
- dill==0.3.2: Dill extends Python's pickle module for serializing and de-serializing python objects to the majority of the built-in python types.
- tqdm==4.50.2: it is an easy-to-use, extensible progress bar Python package that makes adding simple progress bars to Python processes extremely easy.
- pympler==0.8: Pympler is a development tool to measure, monitor and analyse the memory behaviour of Python objects in a running Python application.
- torchvision==0.8.1: this library is part of the PyTorch project, an open-source machine learning framework. The Torchvision package consists of popular datasets, model architectures, and common image transformations for computer vision.
- pillow==7.2.0: this library provides extensive file format support, an efficient internal representation and fairly powerful image processing capabilities.

5 MUSKETEER Machine Learning Library Usage

In this section we will briefly describe the potential usage of the library outside the demos, to ease its integration in the final prototypes with the Client Connector developed by the partner ENG.

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.

5.1 Communications setup

In the previous release of the machine learning library the communications used for interchanging information between the master and the workers were based on a local flask server. This fact implied that an additional terminal was needed to launch the server and the communication was strictly limited to processes within the same machine or at least under the same private network.

However, in order to generalize to workers situated anywhere and without any geographic restriction a new communications library based on the MUSKETEER pycloudmessenger has been integrated in MMLL. Therefore, the only requirement needed for a master and different worker to participate in a common federated training is to adhere to a task under the same name. In order to be able to use the cloud communications the different nodes (both master and workers) need some credentials to access the MUSKETEER pycloudmessenger service.

5.2 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 objects: the data connector (DC) is only needed if some validation or test data is to be used by the MasterNode, the Communications object (Comms), the task manager and the logger object. Only the Comms object has been packaged inside the ML library; the rest of the objects are part of some tools provided for running the different demos:

```
from MMLL.comms.comms_pycloudmessenger import Comms_master as Comms
```

```
from demo_tools.task_manager_pycloudmessenger import Task_Manager
from demo_tools.data_connectors.Load_from_file import Load_From_File as DC
from demo_tools.mylogging.logger_v1 import Logger
from demo_tools.evaluation_tools import display, plot_cm_seaborn, create_folders
```

We then need to provide the credentials for the cloud as well as log into it with the username and password. The MasterNode is also in charge of providing a task name and specifying the configuration parameters for the training of the algorithm.

```
logger = Logger('./results/logs/Master_' + str(user_name) + '.log')
credentials_filename = '../../musketeer.json'
tm = Task_Manager(credentials_filename)
aggregator = tm.create_master_and_taskname(display, logger, task_definition, user_name=user_name, user_password=user_password, task_name=task_name)
```

The communications object should be created afterwards:

```
comms = Comms(aggregator)
```

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_val” and “get_data_tst” that returns one numpy array with the input features and another with the targets (if the task is a supervised one), for both validation and test cases.

```
data_file = ‘./mydata.txt’
dc = DC(data_file)
```

The next step is to create the MasterNode itself, and we pass as parameters the selected POM, the Comms object, the logger and a flag for printing logs on the console:

```
verbose = False
pom = 1
mn = MasterNode(pom, comms, logger, verbose)
```

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

We load the data:

```
[Xval, yval] = dc.get_data_val()
```

We create the model of the selected type, passing as arguments the hyperparameters for the algorithm:

```
model_type = 'SVM'
model_parameters = {}
model_parameters['NC'] = int(task_definition['NC'])
model_parameters['Nmaxiter'] = int(task_definition['Nmaxiter'])
model_parameters['tolerance'] = float(task_definition['tolerance'])
model_parameters['sigma'] = float(task_definition['sigma'])
model_parameters['C'] = float(task_definition['C'])
model_parameters['NmaxiterGD'] = int(task_definition['NmaxiterGD'])
model_parameters['eta'] = float(task_definition['eta'])
mn.create_model_Master(model_type, model_parameters=model_parameters)
```

And we start the training procedure, passing the validation data (if needed):

```
mn.fit(Xval=Xval, yval=yval)
```

5.3 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, the client side needs to import it from the library:

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

Before instantiating it, we need some extra objects. Some are provided directly as part of the MMLL such as the Communications object (Comms) or, in some POMs the Crypto object. Others are part of the tools for running the demos, such is the case of the task manager or the Data Connector (DC) or the Logger to cite some. We start importing them from the library:

```
from MMLL.comms.comms_pycloudmessenger import Comms_worker as Comms
from demo_tools.task_manager_pycloudmessenger import Task_Manager
from demo_tools.data_connectors.Load_from_file import Load_From_File as DC
from demo_tools.mylogging.logger_v1 import Logger
from demo_tools.evaluation_tools import display, plot_cm_seaborn,
create_folders
```

After loading the data, the client has to provide the corresponding credentials, create the user within the cloud (in case it does not already exist) and join an existing task. The parameters

are the path to the JSON file with the credentials, the name of the user, the password and the task name as well as a logger object:

```
logger = Logger('./results/logs/Worker_' + str(user_name) + '.log')
credentials_filename = '../../musketeer.json'
tm = Task_Manager(credentials_filename)
participant = tm.create_worker_and_join_task(user_name, user_password, task_name, display, logger)
```

We then instantiate the Comms object, which needs as input parameter the participant object (to be able to send messages through the cloud) and the username:

```
comms = Comms(participant, user_name)
```

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_all_data_Worker" that returns the numpy array with the input features as well as 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)
```

The next step is to create the WorkerNode itself, and we pass as parameters the selected POM, the Comms object, the logger and a boolean flag indicating whether to print the messages on the console or not:

```
verbose = False
pom = 1
wn = WorkerNode(pom, comms, logger, verbose)
data_partition_id = 0
Xtr, ytr = dc.get_data_train_Worker(int(data_partition_id))
wn.set_training_data(dataset_name, Xtr, ytr)
```

We create the model of the selected type:

```
model_type = 'SVM'
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. After the training is completed, the client will be able to save the model to disk and use it to make predictions on new data:

```
model = wn.get_model()
preds_tst = model.predict(Xtst)
```

6 Execution of demos

In this section we describe the steps needed to test the developed library in some selected Machine Learning Tasks.

6.1 Technical requirements

For the execution of the demos a proper Python environment has to be set up beforehand with all the dependencies correctly installed, including the MMLL library as well as the communications library used here (https://github.com/IBM/pycloudmessenger/). Although this information is present in the Github repository of the project (https://github.com/Musketeer-H2020/MMLL), the details of the configuration are going to be included also in this report.

The environment instructions are described for Anaconda, although any other tool for managing virtual environments could be used. We create a conda environment with the basic configuration and activate it with the following commands:

```
conda create --name MMLL_demo python=3.7.4 git gmpy2==2.0.8 -c defaults
      -c conda-forge --yes
conda activate MMLL_demo
```

6.2 Setup for the demos

In order to be able to run the demos, the user has first to clone the repository for the demos. Working on the same virtual environment created in the above section, the user has to type:

```
git clone https://github.com/Musketeer-H2020/MMLL-demo.git
```

After that, navigate to the root directory of the repository and install the requirements (including pycloudmessenger and MMLL as well as all the dependencies):

```
cd MMLL-demo
pip install -r requirements.txt
```

The repository for the demonstration has two important folders:

- demos: this folder contains all the demos available for the machine learning library, including the pre-processing algorithms described in D4.3 as well as the specific algorithms object of this deliverable. The different demos are organized by POMs and in order to be able to run

them a json credentials file needs to be placed inside demos/demo_pycloudmessenger.

- input_data: folder containing all the open datasets used for the different demos in pkl format. The datasets have to be downloaded from https://drive.google.com/drive/folders/1-piNDL_tL6V4pCI-En02zeCEqoL-dUUu?usp=sharing and placed in the input_data/ folder.

At this point, the virtual environment is ready and the user can execute the demos. Inside the folder for each demo the user can find additional instructions for execution in a README.txt. Additionally, after launching the scripts a new folder, /results, is created. It contains the following subfolders:

- figures: contains pictures used for the evaluation of the models.
- logs: details with the logs of the execution.
- models: stored models after the training are complete.

6.3 Execution

This section provides the commands to execute the different demos available for POMs 1, 2 and 3 from the list of implemented algorithms: Neural Networks, K-means, SVM, DSVM and FBSVM. Inside the folder demos/demo_pycloudmessenger/ the user can find additional instructions and details for all the algorithms available for the different POMs.

All the commands listed in this document have the following common parameters:

- user: string with the name of the user. If the user does not exist in the pycloudmessenger platform a new one will be created.
- password: string with the password.
- task_name: string with the name of the task.
- id: integer representing the partition of data to be used by the worker. Each worker should use a different partition, possible values are 0 to 4.

In order to run each of the demos, open three bash terminals and execute the detailed lines, one at every terminal. Start launching the master and wait until it is ready before launching the workers. Every terminal represents one participant, and it can be in a different machine.

Once the training is completed, these demo scripts produce the output files in the results/ folder (models, figures and logs).

6.3.1 POM1

6.3.1.1 Neural Networks

```
python pom1_NN_master_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name>

python pom1_NN_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 0

python pom1_NN_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 1
```

6.3.1.2 K-means

```
python pom1_Kmeans_master_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name>

python pom1_Kmeans_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 0

python pom1_Kmeans_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 1
```

6.3.1.3 SVM

```
python pom1_SVM_master_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name>

python pom1_SVM_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 0

python pom1_SVM_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 1
```

6.3.1.4 DSVM

```
python pom1_DSVM_master_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name>

python pom1_DSVM_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 0

Python pom1_DSVM_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 1
```

6.3.1.5 FBSVM

```
python pom1_FBSVM_master_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name>

python pom1_FBSVM_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 0

python pom1_FBSVM_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 1
```

6.3.2 POM2

6.3.2.1 Neural Networks

```
python pom2_NN_master_pycloudmessenger.py --user <user> --password <password> --task_name <task_name>

python pom2_NN_worker_pycloudmessenger.py --user <user> --password <password> --task_name <task_name> --id 0

python pom2_NN_worker_pycloudmessenger.py --user <user> --password <password> --task_name <task_name> --id 1
```

6.3.2.2 K-means

```
python pom2_Kmeans_master_pycloudmessenger.py --user <user> --password <password> --task_name <task_name>

python pom2_Kmeans_worker_pycloudmessenger.py --user <user> --password <password> --task_name <task_name> --id 0

python pom2_Kmeans_worker_pycloudmessenger.py --user <user> --password <password> --task_name <task_name> --id 1
```

6.3.2.3 SVM

```
python pom2_SVM_master_pycloudmessenger.py --user <user> --password <password> --task_name <task_name>

python pom2_SVM_worker_pycloudmessenger.py --user <user> --password <password> --task_name <task_name> --id 0

python pom2_SVM_worker_pycloudmessenger.py --user <user> --password <password> --task_name <task_name> --id 1
```

6.3.2.4 FBSVM

```
python pom2_FBSVM_master_pycloudmessenger.py --user <user> --password <password> --task_name <task_name>

python pom2_FBSVM_worker_pycloudmessenger.py --user <user> --password <password> --task_name <task_name> --id 0

python pom2_FBSVM_worker_pycloudmessenger.py --user <user> --password <password> --task_name <task_name> --id 1
```

6.3.3 POM3

6.3.3.1 Neural Networks

```
python pom3_NN_master_pycloudmessenger.py --user <user> --password <password> --task_name <task_name>

python pom3_NN_worker_pycloudmessenger.py --user <user> --password <password> --task_name <task_name> --id 0
```

```
python pom3_NN_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 1
```

6.3.3.2 K-means

```
python pom3_Kmeans_master_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name>

python pom3_Kmeans_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 0

python pom3_Kmeans_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 1
```

6.3.3.3 SVM

```
python pom3_SVM_master_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name>

python pom3_SVM_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 0

python pom3_SVM_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 1
```

6.3.3.4 FBSVM

```
python pom3_FBSVM_master_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name>

python pom3_FBSVM_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 0

python pom3_FBSVM_worker_pycloudmessenger.py --user <user> --password
<password> --task_name <task_name> --id 1
```

6.4 Demo modification

6.4.1 Neural Network demos

This algorithm has a set of hyperparameters that can be modified in the corresponding master script:

- Nmaxiter (*int*): Maximum number of communication rounds.
- learning_rate (*float*): Learning rate for training (only used when model_averaging is False).
- model_architecture (*JSON*): JSON containing the neural network architecture as defined by Keras (in model.to_json()).

- optimizer (*string*): Type of optimizer to use (should be one from https://keras.io/api/optimizers/).
- momentum (*float*): Optimizer momentum (only used when model_averaging is False).
- nesterov (*boolean*): Flag indicating if the momentum optimizer is Nesterov or not (only used when model_averaging is False).
- loss (*string*): Type of loss to use (should be one from https://keras.io/api/losses/).
- metric (*string*): Type of metric to use (should be one from https://keras.io/api/metrics/).
- batch_size (*int*): Number of samples to use for each training step in each worker locally.
- num_epochs (*int*): Number of epochs to train in each worker locally before sending the result to the master.
- model_averaging (*boolean*): Flag indicating whether to use model averaging (True) or gradient averaging (False).

The neural network architecture can be modified in the file model_definition_keras.py inside each of the demos corresponding to neural networks under POMs 1, 2 and 3. It is described in Keras format, for example:

```
model = Sequential()
model.add(Dense(256, input_shape=(784,), activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

6.4.2 K-means demos

The parameters of the model are defined in the master script and are the following:

- NC (*int*): Number of centroids.
- Nmaxiter (*int*): Maximum number of iterations.
- tolerance (*float*): Minimum tolerance for continuing training.

6.4.3 SVM demos

SVM has the following set of hyperparameters defined in the master script:

- NC (*int*): Number of support vectors in the semiparametric model.

- Nmaxiter (*int*): Maximum number of iterations.
- tolerance (*float*): Minimum tolerance for continuing training.
- sigma (*float*): The parameter of the gaussian kernel.
- C (*float*): The cost parameter in the cost function.
- NmaxiterGD (*int*): Maximum number of iterations for the SVM.
- eta (*float*): The step of the gradient descent algorithm.

6.4.4 DSVM demos

The hyperparameters for the Distributed Support Vector Machine are the following ones (they can be modified at master script):

- NC (*int*): Number of support vectors in the semiparametric model.
- Nmaxiter (*int*): Maximum number of iterations.
- tolerance (*float*): Minimum tolerance for continuing training.
- sigma (*float*): The parameter of the gaussian kernel.
- C (*float*): The cost parameter in the cost function.
- eps (*float*): Threshold to update the a variables in the IRWLS algorithm.
- NI (*int*): Number of data features.
- minvalue (*float*): The centroids are initialized randomly from an uniform distribution. This is the minimum value.
- maxvalue (*float*): The centroids are initialized randomly from an uniform distribution. This is the maximum value.

6.4.5 FBSVM demos

The last algorithm included in the POMs object of this deliverable is the Federated Budget Support Vector Machine. These are its hyperparameters, which can be adjusted in the master:

- NC (*int*): Number of support vectors in the semiparametric model.
- Nmaxiter (*int*): Maximum number of iterations.
- tolerance (*float*): Minimum tolerance for continuing training.
- sigma (*float*): The parameter of the gaussian kernel.
- C (*float*): The cost parameter in the cost function.
- num_epochs_worker (*int*): Number of epochs in every worker before sending the weights to the master node in every iteration.

- eps (*float*): Threshold to update the variables in the IRWLS algorithm.
- mu (*float*): Step to update the weights in the master node after every iteration.
- NI (*int*): Number of data features.
- minvalue (*float*): The centroids are initialized randomly from a uniform distribution. This is the minimum value.
- maxvalue (*float*): The centroids are initialized randomly from a uniform distribution. This is the maximum value.

7 Conclusion

This report includes a description of the final release of the MUSKETEER Machine Learning Library (MMLL) under POMs 1, 2 and 3. This final release uses a cloud messaging service in order to be able to communicate decentralized machines at different locations.

The algorithms included enable developers to solve different machine learning problems, from unsupervised to supervised learning. From the one side, unsupervised learning is covered by K-means algorithm with a novel initialization approach, which enhances privacy under POM1. From the other side, supervised learning problems such as classification and regression can also be solved thanks to the implementation of neural networks, SVM, DSVM and FBSVM algorithms.

Finally, an extensive list of pre-processing techniques is also part of the library, including different normalization strategies, natural language and image processing and feature extraction. The detailed list of these techniques can be found in D4.3.

8 References

[Cortes, 1995] Cortes, Corinna, and Vladimir Vapnik. "Support-vector networks." Machine learning 20.3 (1995): 273-297.

[Díaz, 2016] Díaz-Morales, Roberto, and Ángel Navia-Vázquez. "Efficient parallel implementation of kernel methods." Neurocomputing 191 (2016): 175-186.

[Díaz, 2018] Díaz-Morales, Roberto, and Ángel Navia-Vázquez. "Distributed Nonlinear Semiparametric Support Vector Machine for Big Data Applications on Spark Frameworks." IEEE Transactions on Systems, Man, and Cybernetics: Systems 50.11 (2018): 4664-4675.

[Hofmann, 2008] Hofmann, Thomas, Bernhard Schölkopf, and Alexander J. Smola. "Kernel methods in machine learning." The annals of statistics (2008): 1171-1220.

[Jain, 2010] Jain, Anil K. "Data clustering: 50 years beyond K-means." Pattern recognition letters 31.8 (2010): 651-666.

[Jiuxiang, 2018] Gu, Jiuxiang, et al. "Recent advances in convolutional neural networks." Pattern Recognition 77 (2018): 354-377.

[Konečný, 2016] Konečný, Jakub, et al. "Federated learning: Strategies for improving communication efficiency." arXiv preprint arXiv:1610.05492 (2016).

[Le Cun, 2015] Yan, Le Cun, B. Yoshua, and H. Geoffrey. "Deep learning." nature 521.7553 (2015): 436-444.

[Meng, 2016] Meng, Xiangrui, et al. "Mllib: Machine learning in apache spark." The Journal of Machine Learning Research 17.1 (2016): 1235-1241.

[Razvan, 2013] Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. "On the difficulty of training recurrent neural networks." International conference on machine learning. PMLR, 2013.

[Scholkpf, 2001] Scholkopf, Bernhard. "The kernel trick for distances." Advances in neural information processing systems (2001): 301-307.

[Yang, 2019] Yang, Qiang, et al. "Federated learning." Synthesis Lectures on Artificial Intelligence and Machine Learning 13.3 (2019): 1-207.