H2020 – ICT-13-2018-2019

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

Grant No 824988

D4.4 Machine Learning Algorithms over Federated Operation Modes algorithms – Initial version

July 20

Imprint

Contractual Date of Delivery to the EC: **31 January 2020**

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

Participant(s): **TREE, COMAU, IBM, UC3M**
Reviewer(s): **Mathieu Sinn (IBM Research Ireland), Lucrecia Morabito (COMAU)**

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

Work package: **WP4**
Dissemination level: Internal
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

This deliverable aims to provide a first version of the machine learning algorithms under each of the Federated Collaborative Privacy Operation Modes

Document History

Version	Date	Status	Author	Comment
1	12 Dec 2019	Internal writing	Jaime Medina	Structure and table of contents
2	17 Dec 2019	First version	Roberto Díaz	First content included
	13 Jan 2020	Second version	Roberto Díaz	First full version
	13 Jan 2020	Internal review version	Mathieu Sinn	Reviewed version
	13 Jan 2020	Internal review version	Lucrezia Morabito	Reviewed version
	17 Jan 2020	Updated reviewed version	Roberto Díaz	First reviewed version
	17 Jan 2020	Updated clean reviewed version	Jaime Medina	Second reviewed version
	20 Jan 2020	Final version	Gal Weiss	Final version

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

Abbreviation	Definition
CL	Communication Library
FML	Federated Machine Learning

MMLL	MUSKETEER Machine Learning Library
MN	Master Node
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.

In this deliverable it is explained how the development process of the machine learning algorithms under each of the Federated Collaborative Privacy Operation Modes has been done. 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 POM 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.
- 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 private.
- 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.

This deliverable will provide a first version of the library with linear models and kernel methods in the Federated Collaborative Privacy Operation Modes.

1.2 Related Documents

This deliverable is labelled as DEMONSTRATOR, so the information included here is just a summary of the works done.

For the main developments and progresses see following code stored in:

ML_ MUSKETEER_POMS1-2-3 _v1.0

https://ibm.ent.box.com/folder/82343438031.

1.3 Document Structure

The remainder of this document is structured as follows:

Section 2 describes the methodology that the working team has carried out in the process of generation this first version of machine learning algorithms.

Section 3, 4 and 5 correspond to each of the Privacy Operation Modes in which this deliverable is focused on. Each of the section has been divided in 3 different subsections, in which each particular algorithm will be addressed.

Finally, Section 5 includes general conclusion of the works reported in this deliverable.

2 Methodology

2.1 Development process

The progress has been made following iterative process:

Table 1. Neural networks internal development process

Neural networks	
1. DEMO without communications library	Done
2. First version of communications library provided by IBM	
3. DEMO with communications library 1st version	Done
4. First version of code structure agreed between UC3M and TREE	
5. DEMO with updated implementation of code structure	Done
Neural Networks DEMO v1.0.0	**Released. Provided in D4.4**
Algorithm optimization	To be provided in D4.5 (M30)
DEMO with final communications library	To be provided in D4.5 (M30)
DEMO with final code structure	To be provided in D4.5 (M30)
Neural Networks DEMO v2.0.0	**To be released in D4.5 (M30)**

Table 2. K-means internal development process

Clustering (K-means)	
1. DEMO without communications library	Done
2. First version of communications library provided by IBM	
3. DEMO with communications library 1st version	Done
4. First version of code structure agreed between UC3M and TREE	
5. DEMO with updated implementation of code structure	Done
Clustering (K-means) DEMO v1.0.0	**Released. Provided in D4.4**
Algorithm optimization	To be provided in D4.5 (M30)
DEMO with final communications library	To be provided in D4.5 (M30)
DEMO with final code structure	To be provided in D4.5 (M30)
Clustering (K-means) DEMO v2.0.0	**To be released in D4.5 (M30)**

Table 3. SVM internal development process

Kernel Methods (SVM)	
1. DEMO without communications library	Done
2. First version of communications library provided by IBM	
3. DEMO with communications library 1st version	Done
4. First version of code structure agreed between UC3M and TREE	
5. DEMO with updated implementation of code structure	Done
Kernel Methods (SVM) DEMO v1.0.0	**Released. Provided in D4.4**
Algorithm optimization	To be provided in D4.5 (M30)
DEMO with final communications library	To be provided in D4.5 (M30)
DEMO with final code structure	To be provided in D4.5 (M30)
Kernel Methods (SVM) v2.0.0	**To be released in D4.5 (M30)**

3 Federated Collaborative Privacy Operation Modes

A traditional centralized solution requires that the data from different users is gathered in a common location. A distributed privacy preserving approach requires to exchange some information among the participating such we can train a ML model without ever receiving/seeing the raw data of the users.

This demonstrator presents an initial version of some distributed privacy preserving operation modes in the MUSKETEER platform, concretely the federated Privacy Operation Modes (POMs).

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 and at the end all users get access to the trained model in unencrypted form.

Currently, there are three Federate POMs that will be briefly described here. For further details, consult de MUSKETEER deliverable 4.1: "Investigative overview of targeted architecture and algorithms".

3.1 Privacy Operation Mode 1 (Aramis)

This POM make is a Federated Machine Learning (FML) schema. FML, that was prosed in [McMahan_2017] [Konečný_2016] [Shokri_2015] is as an alternative to a traditional local or cloud computing for training predictive models using machine learning.

Under this paradigm, a shared global model is trained under the coordination of a central node, from a federation of participating devices.

FML enables different devices to collaboratively learn a shared prediction model while keeping all the training data on device, decoupling the ability to perform machine learning from the need to store the data in the cloud.

Using this approach, data owners can offer their data to train a predictive model without being exposed to data leakage or data attacks. In addition, since the model updates are specific to improving the current model, there is no reason to store them on the server once they have been applied.

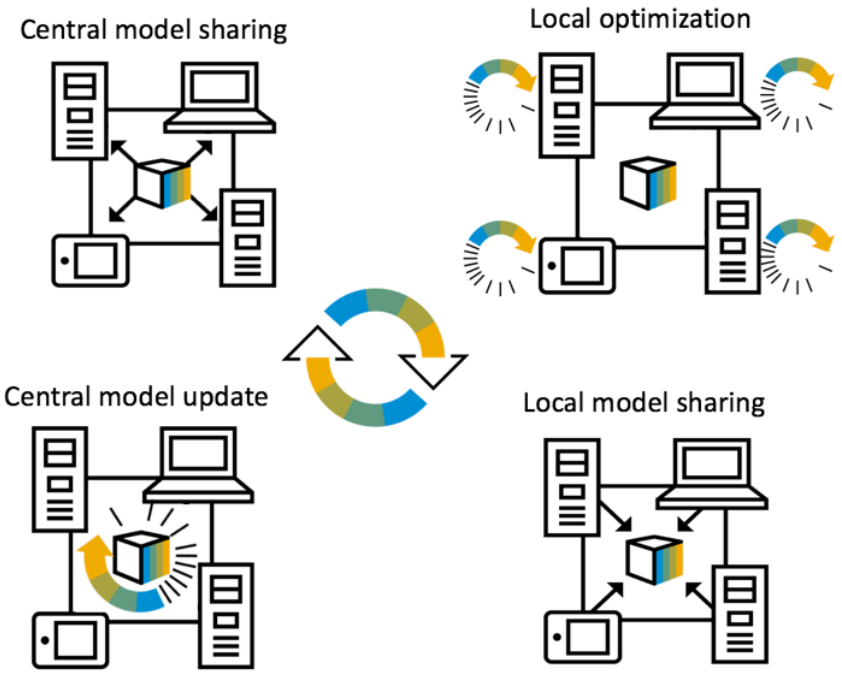

Figure 1. FML training schema (figure obtained from https://medium.com/sap-machine-learning-research/privacy-preserving-collaborative-machine-learning-35236870cd43).

FML turns the update of Machine Learning models upside-down by allowing the devices with data on the edge to participate in the training. Instead of sending the data in the client to a centralised location, Federated Learning (see Figure 1) sends the model to the devices participating in the federation. The model is then improved with the local data. And the data never leaves the local device. After that, the clients send updates to the model to the central node that can aggregate the different partial updates to globally update the model.

3.2 Privacy Operation Mode 2 (Athos)

In POM1 data information may be leaked to an honest-but-curious server since the server has access to the predictive model. In some use cases, data owners belong to the same company (e.g. different factories of the same company) and the server that orchestrates the training is in the cloud. POM2 fixes that problem with two properties:

- No information is leaked to the server: POM2 leaks no information of participants to the honest-but-curious cloud server.
- The accuracy is kept intact compared to POM1: Achieves identical accuracy to a corresponding system trained using stochastic gradient descent.

The work proposed in [Aono_2018] shows that having access to the predictive model and to the gradients it is possible to leak information. Since the orchestrator has access to this information in POM1, if is not completely under our control (e.g. Azure or AWS cloud), POM2 solves the problem by protecting the gradients over the honest-but-curious cloud server.

POM2 uses additively homomorphic encryption. All gradients are encrypted and stored on the cloud server and the additive property enables the computation across the gradients. D4.1 Investigative overview of targeted architecture and algorithms 23 Machine Learning to Augment Shared Knowledge in Federated Privacy-Preserving Scenarios (MUSKETEER)

This protection of gradients against the server comes with the cost of increased computational and communication between the learning participants and the server.

3.3 Privacy Operation Mode 3 (Porthos)

In POM2, every data owner trusts each other and they can share the private key of the homomorphic encryption (e.g. different servers with data that belongs to the same owner). Using the same key, every data owner uses the same encrypted domain. In many situations it is not possible to transfer the private key in a safe way.

POM3 is an extension of POM2 that makes use of a proxy re-encryption [[Fuchsbauer_2019]] protocol to allow that every data owner can handle her/his own private key.

4 The Machine Learning Library

This deliverable is a preliminary but operable version of the Machine Learning Library to be used in MUSKETEER project under POMs 1, 2 and 3.

Essentially, it aims at deploying a distributed ML setup (Figure 1b) such that a model equivalent to the one obtained in the centralized setup (Figure 1a) is obtained.

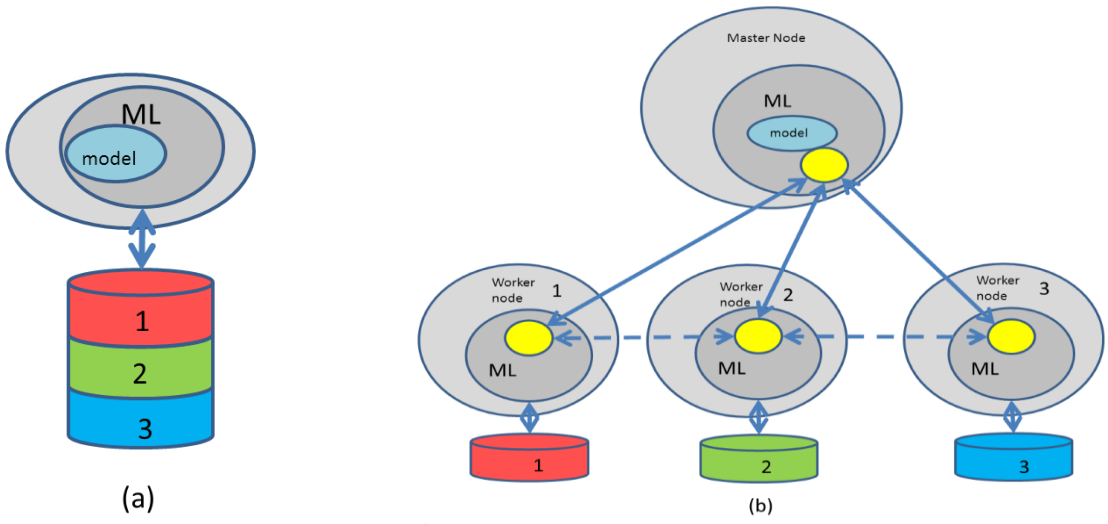

Figure 2. Centralized (a) vs. distributed scenario (b). Every user provides a portion of the training dataset. Data confidentiality must be preserved.

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

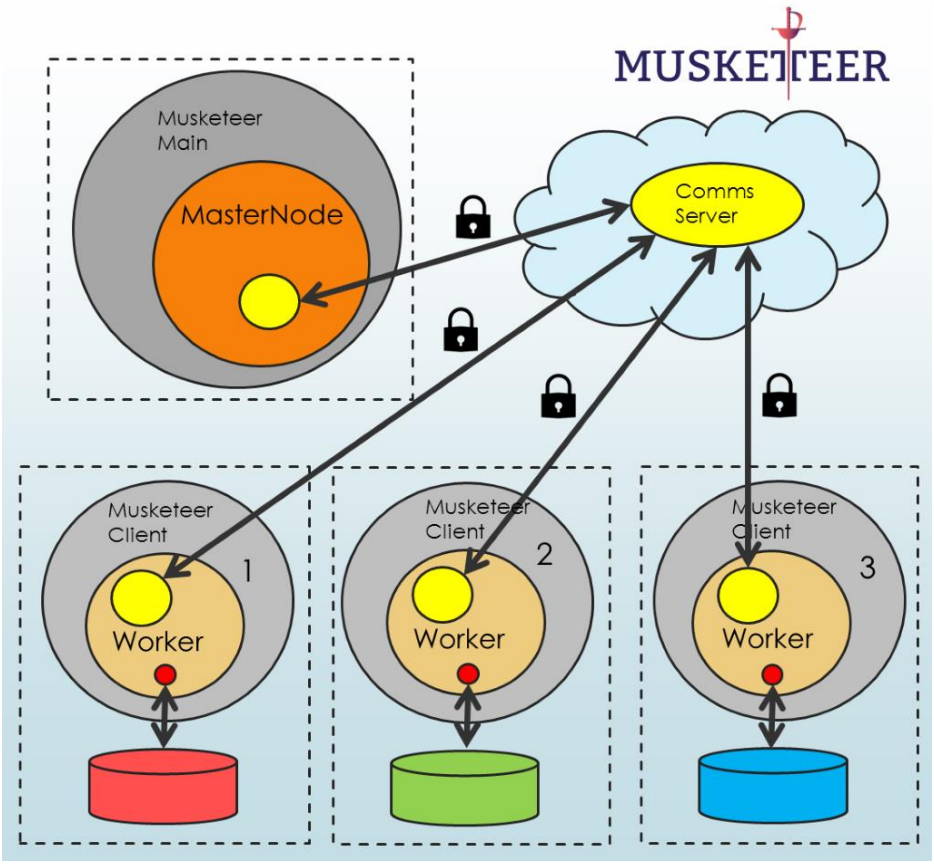

Figure 3. Detailed process interactions in a MUSKETEER learning process.

There are two main 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.

The main process and every client can be running on a different machine.

5 Current set of algorithms

5.1 Deep Neural Networks

Deep learning architectures such as recurrent neural networks or convolutional neural networks 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.

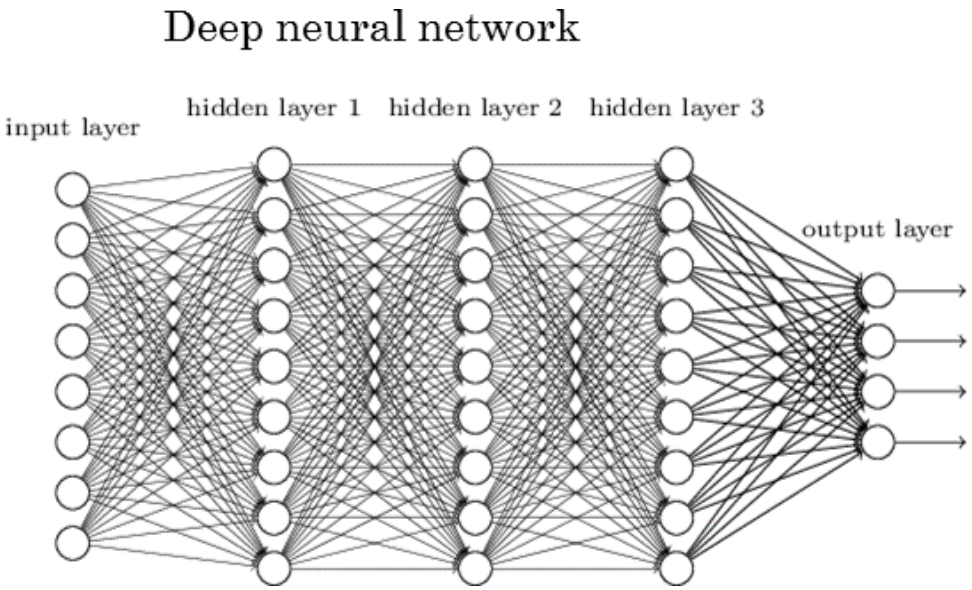

Figure 4. Deep Neural Network architecture (figure extracted from https://datawarrior.wordpress.com/2016/04/16/relevance-and-deep-learning/)

5.1.1 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:

- 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.

Main process:

- Compute the global gradients adding the gradients of every client.
- Update the Neural Network using a gradient descent step.
- If the defined number of iterations is reached, send a signal to every client to finish the training.

5.1.2 Neural Networks over POM 2 explained:

POM2 uses the same schema but using homomorphic encryption.

Every client in the initialization step load the private and public key, 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 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.

5.1.3 Neural Networks over POM 3 explained:

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

5.2 Clustering (K-means)

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 center of the cluster. Every data point is allocated to each of the clusters through reducing the predefined distance matric.

In other words, the K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible.

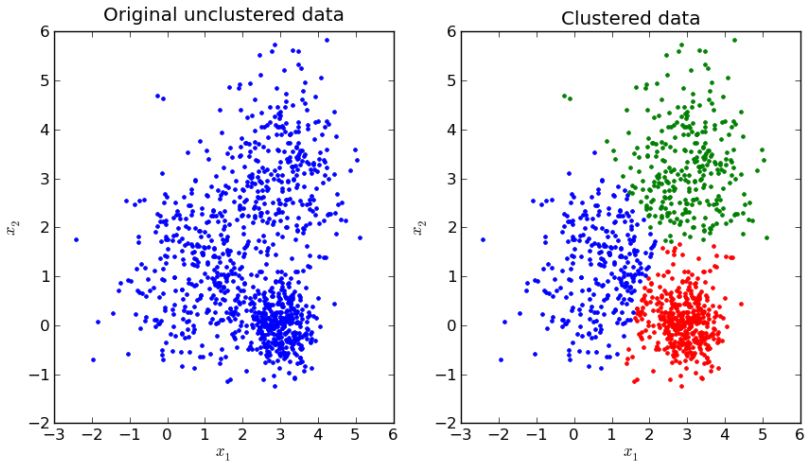

Figure 5. Example of clustering to divide data into three different groups.

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.

5.2.1 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:

- Load its dataset.
- Define a subset of the initial centroids by averaging some training data.
- Send the subset of initial centroids to the MUSKETEER.

Main process:

- Collect the initial centroids from every client.

2-Iterative process:

Main process:

- Send the centroids to every client.

Every client:

- Assign each local data to the closest corresponding centroid, using the Euclidean distance.
- For each centroid, calculate the local mean of the values of all the points belonging to it.
- Send 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.
- Replace every centroid by the global mean.
- Detect 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 has been reached, send a signal to every client to finish the training.

5.2.2 Kmeans over POM 2 explained:

POM2 uses the same schema but using homomorphic encryption.

Every client in the initialization step load 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 update 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.

5.2.3 Kmeans over POM 3 explained:

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

5.3 Kernel Methods

Kernel Methods [Scholkopf_2001] 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 products 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). Patterns that do not respect this margin distance or directly are wrongly classified are called Support Vectors (SVs).

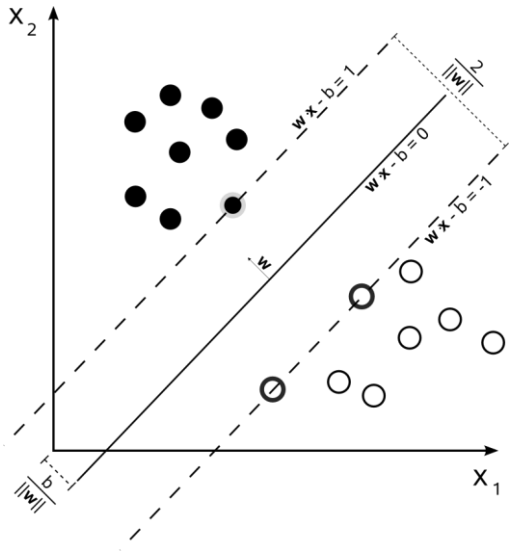

Figure 6. Maximum margin classifier (figure extracted from https://www.quora.com/Why-do-we-call-an-SVM-a-large-margin-classifier).

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.

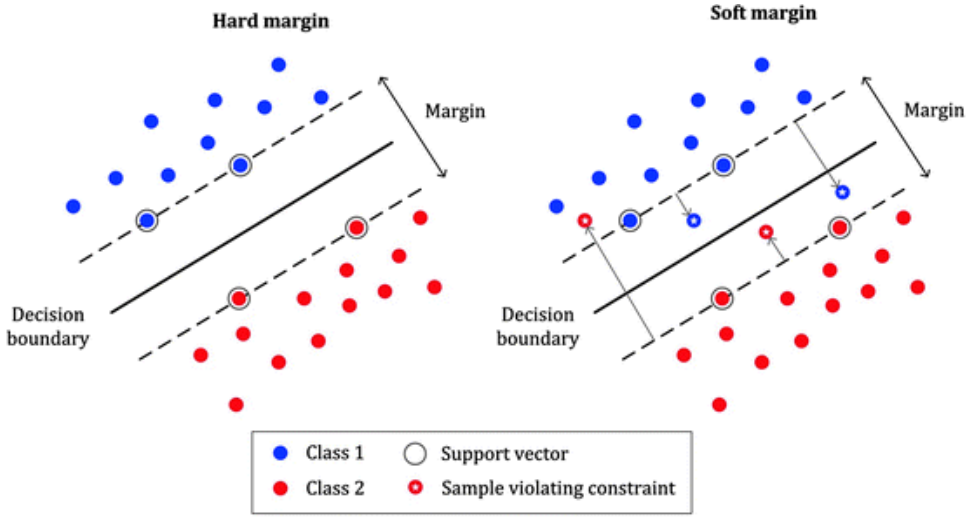

Figure 7. Hard margin vs soft margin (figure extracted from https://mc.ai/math-behind-svmsupport-vector-machine/).

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.

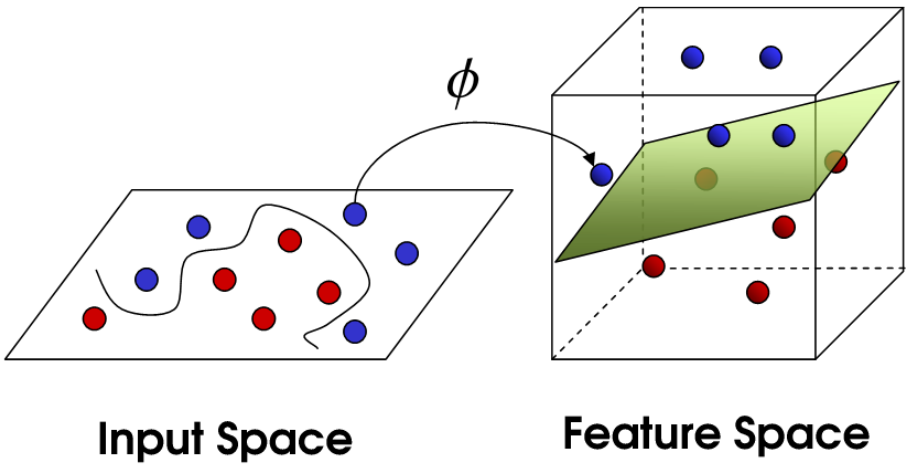

Figure 8. Kernel trick (figure extracted from https://es.switch-case.com/52732403).

Semiparametric (budgeted) models have been proposed to keep the classifier complexity under control [Diaz_2016], [Diaz_2017], [Diaz_2018]. In these models, having a dataset in the form:

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

We have two different steps:

1 - Centroid selection:

A procedure to select m basis centroids C = {c1, ..., cm} among the training set is carried out at a first step.

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:

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

5.3.1 Budgeted SVM over POM 1 explained:

1-Initialization:

Every client:

- Load its dataset.

2-Centroid selection:

To select the centroids, we make use of the K-means algorithm described in section 5.2.1. At the end, every client has a copy of these centroids.

3-Optimization procedure:

We make use of the gradient descent algorithm to solve the optimization problem.

Main process:

- Send the weights to every client.

Every client:

- Compute the gradients of every training data in their respective datasets.
- Add the gradients and send the result to the main process.

Main process:

- Receive the gradients from every client.
- Update the weights a step in the gradient descent algorithm.
- Detect if the stop criteria has 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 has been reached, send a signal to every client to finish the training.

5.3.2 Budgeted SVM over POM 2 explained:

POM2 uses the same schema but using homomorphic encryption.

Every client in the initialization step load the private and public key, 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 it using the private key.

5.3.3 Budged SVM over POM 3 explained:

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

6 Library Demonstration 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 (Master Node (MN) and Worker Nodes (WN)) is available to every node. In the final version every participant (Master/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 master node and the participants gradients or other optimization parameters for the predictive models. 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.

7 External dependencies

This demonstration makes use of some external public Python libraries, all of them freely available:

numpy: is the fundamental package for scientific computing with Python.

Matplotlib: is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

Scikit-learn: is a free software machine learning library for the Python programming language

seaborn: Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

pandas: pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

Flask: Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications.

Tensorflow: To run the Deep Neural Network demo. TensorFlow is an end-to-end open source platform for machine 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.

Keras: To run the Deep Neural Network demo. Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation.

IPython: IPython provides a rich architecture for interactive computing.

8 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.

8.1 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. Therefore, we need to start that server, by running:

```
python3 MUSKETEER.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 is completed, the local server will no longer be needed, we will just need some credentials to access the IBM Cloud service.

8.2 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 objects[2]: 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 import Comms
```

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)
```

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 = 1
wn = WorkerNode(pom, worker_id, comms, dc, master_address='ma')
```

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.

8.3 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 objects[3]: 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 the logger object. We start importing them from the library:

```
from MMLL.data_connectors.Load_from_file import Load_From_File as DCfrom MMLL.comms.comms import Comms
from MuskLib.logging.logger_v1 import Logger
```

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):

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

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. The Cryptographic object is not needed since it is not used in POMs 1, 2 and 3:

```
pom = 1
master_address = 'ma'
Crypto_address = None
mn = MasterNode(pom, master_address, comms, dc, crypto_address)
```

(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()
```

9 Execution of 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.

9.1 Technical requirements

Before executing the Demos, it is necessary to correctly configurate a Python 3 environment with all the required libraries described in section 7. In the final version of the platform, such configuration will be simplified, since the code will be embedded in a "docker" container.

Uncompress the file ML_ MUSKETEER_POMS1-2-3.zip. You should find the following subfolder structure:

```
input_data/
MuskLib/
results/
```

* 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://ibm.ent.box.com/folder/101041827355), but the provided datasets are enough to explore the Machine Learning Library usage.

* MuskLib: The MUSKETEER Machine Learning Library with (POMs 1, 2 and 3)

* results. Some output figures are saved here. Also, a subfolder with execution logs is available.

9.2 Execution

This demonstration is prepared to run the **master node** and from 1 to 5 **clients**.

There are three different model types available: NeuralNetwork, SVM and Kmeans.

Every participating process will be run on a different terminal, such that a detailed list of messages are shown in every screen for easy monitoring of the operations and protocols behaviour.

For illustration purposes, we use here the Linux OS.

9.2.1 The communication service:

First, we need to execute in the first terminal:

```
>>python3 MUSKETEER.py
```

And observe:

```
roberto.diaz@TLPORTATIL125:~/Documentos/ML_Musketeer$ python3 musketeer.py
 * Serving Flask app "musketeer" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://localhost:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 335-507-609
```

Figure 9. The local communications terminal

After that, we can execute the clients and the master node.

9.2.2 The clients:

Every client needs as parameters:

- The POM: In this demo, we have available 1, 2 and 3.
- Model type: NeuralNetwork, SVM and Kmeans.
- The master address: The identifier of the master node (different from the identifier of the clients).
- The list of client addresses.
- The address of this client.

For example, if we can to run a service with three different clients (identifiers 0, 1 and 2) and a master node (identifier 3), each client must be executed in a differet terminal like this:

```
>>python3 worker.py  --pom 1 --master_address 3 --worker_address 0  --workers_addresses  0_1_2  --model_type  NeuralNetwork  --data_file  input_data/mnist_pickled.pkl # Worker 0

>>python3 worker.py  --pom 1 --master_address 3 --worker_address 1  --workers_addresses  0_1_2  --model_type  NeuralNetwork  --data_file  input_data/mnist_pickled.pkl # Worker 1

>>python3 worker.py  --pom 1 --master_address 3 --worker_address 2  --workers_addresses  0_1_2  --model_type  NeuralNetwork  --data_file  input_data/mnist_pickled.pkl # Worker 2
```

Every worker produces:

```
Worker 0: Loading comms
Worker 0: Creating worker object
worker 0: Loading comms
worker 0: Loading Data Connector
worker 0: Initiated
Worker 0: Loading data from input_data/mnist_pickled.pkl ...
Worker 0: Data loaded, 10000 patterns with 784 features
Worker 0: Creating ML model of type NeuralNetwork
```

Figure 10. The demo execution of a Kmeans under POM1 in full detail (WorkerNode)

9.2.3 Master node:

The master node needs as a parameter:

- The POM: Available 1, 2 and 3 in this demo.
- The dataset to be used.
- Model type: NeuralNetwork, SVM and Kmeans.
- The master address: The address of this node.
- The list of client addresses: The list if client addresses.

```
>> python master.py  --pom 1 --master_address 3 --workers_addresses 0_1_2
--model_type NeuralNetwork --data_file input_data/mnist_pickled.pkl
```

And the master produces:

```
Kmeans_Master: Sent COMPUTE_LOCAL_GRADIENTS to all workers
Kmeans_Master: Received UPDATE_GRADIENTS from worker 1
Kmeans_Master: Received UPDATE_GRADIENTS from worker 2
Kmeans_Master: Received UPDATE_GRADIENTS from worker 0
Kmeans_Master: Resetting local data
10000/10000 [==============================] - 0s 7us/step
Kmeans_Master: Iteration 6, loss: 2.2737, accuracy: 0.1686
Kmeans_Master: Sent COMPUTE_LOCAL_GRADIENTS to all workers
Kmeans_Master: Received UPDATE_GRADIENTS from worker 1
Kmeans_Master: Received UPDATE_GRADIENTS from worker 2
Kmeans_Master: Received UPDATE_GRADIENTS from worker 0
Kmeans_Master: Resetting local data
10000/10000 [==============================] - 0s 8us/step
Kmeans_Master: Iteration 7, loss: 2.2702, accuracy: 0.1707
Kmeans_Master: Sent COMPUTE_LOCAL_GRADIENTS to all workers
Kmeans_Master: Received UPDATE_GRADIENTS from worker 1
Kmeans_Master: Received UPDATE_GRADIENTS from worker 2
Kmeans_Master: Received UPDATE_GRADIENTS from worker 0
Kmeans_Master: Resetting local data
10000/10000 [==============================] - 0s 8us/step
Kmeans_Master: Iteration 8, loss: 2.2667, accuracy: 0.1748
Kmeans_Master: Sent COMPUTE_LOCAL_GRADIENTS to all workers
Kmeans_Master: Received UPDATE_GRADIENTS from worker 1
Kmeans_Master: Received UPDATE_GRADIENTS from worker 2
Kmeans_Master: Received UPDATE_GRADIENTS from worker 0
Kmeans_Master: Resetting local data
10000/10000 [==============================] - 0s 9us/step
Kmeans_Master: Iteration 9, loss: 2.2630, accuracy: 0.1763
Kmeans_Master: Sent COMPUTE_LOCAL_GRADIENTS to all workers
Kmeans_Master: Received UPDATE_GRADIENTS from worker 0
Kmeans_Master: Received UPDATE_GRADIENTS from worker 2
Kmeans_Master: Received UPDATE_GRADIENTS from worker 1
Kmeans_Master: Resetting local data
10000/10000 [==============================] - 0s 8us/step
Kmeans_Master: Iteration 10, test accuracy: 0.1772
Kmeans_Master: Sent SEND_FINAL_MODEL to all workers
Kmeans_Master: Training is done
Master: Training complete.
Master: Training time = 8 seconds
Master: Predicting new data
Master: Terminating all user nodes.
CommonML_Master sent STOP to 3 Workers
```

Figure 11. The demo execution of a Kmeans under POM1 in full detail (MasterNode)

9.3 Demo modificiation

Kmeans demo:

You can modify the file master.py. The variable NC is the number of centroids and the variables Nmaxiter and tolerance are the maximum number of iterations and the threshold of the stop criteria.

```
if model_type in ['Kmeans']:
    if dataset_name in ['mnist']:
        NC = 3
        Nmaxiter = 200
        tolerance = 0.0001
    if dataset_name in ['synth2D', 'synth2Db']:
        NC = 7
        Nmaxiter = 5
        tolerance = 0.0001
```

SVM demo:

You can modify the file master.py. The variable NC is the number of centroids and the variables Nmaxiter and tolerance are the maximum number of iterations and the threshold of the stop criteria to learn the centroids. Sigma is the parameter of the kernel function, C the parameter in the optimization problem and eta is the learning rate in the gradient descent algorithm.

```
if model_type in ['SVM']:
    if dataset_name in ['mnist']:
        NC = 3
        Nmaxiter = 200
        tolerance = 0.0001
        sigma = 0.01
        C = 1
        NmaxiterGD = 100
        eta = 0.05
    if dataset_name in ['synth2D', 'synth2Db']:
        NC = 3
```

```
Nmaxiter = 200
tolerance = 0.0001
sigma = 0.01
C = 1
NmaxiterGD = 100
eta = 0.05
```

Neural Network demo:

The number of iterations and learning rate in the gradient descent can be modified in the file master.py:

```
if model_type in ['NeuralNetwork']:
    if dataset_name in ['mnist']:
        Nmaxiter = 1000
        learning_rate = 0.0003
```

The neural network architecture can be modified in the file neural_network.py. It is described in keras format:

```
model = Sequential()
model.add(Dense(100, input_shape=(self.num_features,)))
model.add(Activation('relu'))
model.add(Dense(self.num_classes))
model.add(Activation('sigmoid'))
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
self.model = model
```

10 Conclusion

The present report presents a preliminary version of the MUSKETEER Machine Learning Library (MMLL) under POMs 1, 2 and 3. We have available Neural Networks, Clustering and Kernel Methods.

11 References

[Aono_2018] Aono, Yoshinori, et al. "Privacy-preserving deep learning via additively homomorphic encryption." IEEE Transactions on Information Forensics and Security 13.5 (2018): 1333-1345

[Cortes_1995] Cortes, Corinna, and Vladimir Vapnik. "Support-vector networks." Machine learning 20.3 (1995): 273-297.

[Diaz_2016] Díaz-Morales, R., & Navia-Vázquez, Á. (2016). Improving the efficiency of IRWLS SVMs using parallel Cholesky factorization. Pattern Recognition Letters, 84, 91-98.

[Diaz_2017] Díaz-Morales, R., & Navia-Vázquez, Á. (2017). LIBIRWLS: A parallel IRWLS library for full and budgeted SVMs. Knowledge-Based Systems, 136, 183-186.

[Diaz_2018] Díaz-Morales, R., & Navia-Vázquez, Á. (2018). Distributed Nonlinear Semiparametric Support Vector Machine for Big Data Applications on Spark Frameworks. IEEE Transactions on Systems, Man, and Cybernetics: Systems.

[Fuchsbauer_2019] Fuchsbauer, Georg, et al. "Adaptively secure proxy re-encryption." IACR International Workshop on Public Key Cryptography. Springer, Cham, 2019.

[Jain_2010] Jain, Anil K. "Data clustering: 50 years beyond K-means." Pattern recognition letters 31.8 (2010): 651-666.

[Konečný_2016] Konečný, Jakub, et al. "Federated optimization: Distributed machine learning for on-device intelligence." arXiv preprint arXiv:1610.02527 (2016).

[McMahan_2017] McMahan, Brendan, and Daniel Ramage. "Federated learning: Collaborative machine learning without centralized training data." Google Research Blog 3 (2017)

[Meng_2016] Meng, Xiangrui, et al. "Mllib: Machine learning in apache spark." The Journal of Machine Learning Research 17.1 (2016): 1235-1241.

[Scholkopf_2001] Scholkopf, Bernhard, and Alexander J. Smola. Learning with kernels: support vector machines, regularization, optimization, and beyond. MIT press, 2001.

[Shokri_2015] Shokri, Reza, and Vitaly Shmatikov. "Privacy-preserving deep learning." Proceedings of the 22nd ACM SIGSAC conference on computer and communications security. ACM, 2015.