A Versatile Federated Machine Learning Strategy with Applications to XGBoost, GMM and DBSCAN

Ioanna Vasilopoulou

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Bioinformatics

University of Crete
School of Medicine
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisors:
Dr. *George Potamias*,
Dr. *Alexandros Kanterakis*

This work has been performed at the University of Crete, School of Medicine.
The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
School of Medicine

A Versatile Federated Machine Learning Strategy with Applications to XGBoost, GMM and DBSCAN

Thesis submitted by
Ioanna Vasilopoulou
in partial fulfillment of the requirements for the
Masters' of Science degree in Bioinformatics

Author: Ioanna Vasilopoulou

Three member committee: George Potamias
Thesis Supervisor

Alexandros Kanterakis
Committee Member

Pavlos Pavlidhs
Committee Member

Heraklion, February 2023

A Versatile Federated Machine Learning Strategy with Applications to XGBoost, GMM and DBSCAN

Abstract

Latest algorithmic, hardware and programming advancements have brought Machine Learning (ML) closer to the solution of increasingly more complex problems. However, most of these solutions are only applicable to centralized approaches, where the complete set of training data and the analysis software are located on the same computational entity. For privacy critical applications like biomedical informatics and user data analytics this can be an important issue. Simply, very strict and active legislations on privacy and security, forbid the transfer of any personal information outside of a well defined data sylo (i.e. hospital, mobile device).

Federated Machine Learning (FML) comes to address this issue by offering a different learning approach. Through an iterative process, partial models are trained in each data-sylo and then transferred into a server and aggregated into a single model without exchanging user data. One of the most known frameworks for FML is Flower which deals with all trivial tasks like client-server communication with consistency and security. Flower offers many strategies with which ML methods can coordinate through the process of local training and model aggregation. Existing strategies focus mainly on simple ML methods mainly in the area of statistical learning (i.e. regression) and deep neural networks where model aggregation is straight forward. For more advanced ML methods like tree-based for classification, and distance-based for clustering, not only there is no Flower strategy, but even the concept of "federalization" is an active field of research.

Here we first create a new "Federalized" implementation of the well known tree-based XGBoost algorithm for classification. Then we introduce "FIN", a novel Flower strategy that allows the inclusion of more sophisticated ML methods and we demonstrate how it can be used in our XGBoost implementation. Finally we implement a novel Federated data clustering method based on the synergy of DBSCAN and Gaussian Mixture Models. For all our implementations we perform an extensive set of tests in real and in simulated data and we demonstrate how these federated implementations have the same efficiency as their "classic" counterparts.

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Potamias, for his valuable guidance. I also want to express my deepest gratitude to my advisor, Dr. Alexandros Kanterakis, for his wise advice, support, and encouragement throughout the entire research process. His advice have greatly improved the quality and clarity of my work that contributed to my academic and technical growth. I would also like to thank Alexander Shevtsov for his helpful discussions and suggestions. Finally, I would like to thank my family and friends for their love, support, and encouragement throughout this journey. This work would not have been possible without their unwavering belief in me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Federated Learning

Organizations are typically hesitant to release or share their data due to privacy and legal requirements like the General Data Protection Regulation (GDPR). This is the main reason why the healthcare industry does not share their data. Medical data is extremely sensitive and its usage is strictly controlled. In such cases, trivial privacy preserving techniques, like hiding basic information of a patient, is insufficient [34]. Also, in real world scenarios many data owners do not have sufficient amount of data to create accurate models. For these reasons, data sharing, without privacy leakage, between various data owners has the potential to dramatically increase the performance of machine learning tasks with no additional cost for data maintenance.

Federated Learning (FL) has developed a solution to this issue that does not require data sharing to produce a single model that is trained over multiple data-sets. In other words, FL is a setup for machine learning (ML) in which numerous clients work together to jointly train a model while the training data is kept decentralized [17]. More specifically, decentralized devices or organizations (also known as clients) train their data locally and then send the model parameters back to the central server. On the server side, the model updates are aggregated to create a new, improved global model. This process is repeated until the global model reaches a satisfactory level of accuracy. It was first created for use cases involving mobile and edge devices, among other fields, but it has recently acquired popularity for healthcare applications [19]. FL is often divided into three categories [20]: horizontal FL, vertical FL and federated transfer learning. In the case of horizontal FL, data-sets have the same feature space across all clients. In the vertical FL, feature space is not the same but the data-sets have overlap on samples. In some cases, the data does not share either the sample space or feature space, making it impossible to use the horizontal and vertical categories of federated learning. In such scenarios only the federated transfer learning provides the solution. In this research we are concentrated only on the case of horizontal FL.

Federated Learning represents a promising approach for enabling the training

of high-quality machine learning models on decentralized data sources. It has been applied to a variety of use cases, including natural language processing, computer vision, and speech recognition, and is expected to continue to be an active area of research and development in the field of machine learning.

1.2 Federated Frameworks

As mentioned earlier, federated learning is a very popular and important research topic, since it provides a solution in cases where traditional machine learning models are not possible to apply due to data sharing policy or restrictions. For this reason, several open-source FL frameworks have been proposed by the research community where each of them provides a unique functionality. For example, FedML [14] is an open source library that reduces the complexity of the FL model development. This framework supports diverse computation paradigms (distributed, standalone and on-device) with a wide range of implemented models and strategies which are accessible via API. Unfortunately, FedML does not include API documentation that enables users to quickly set up various FL scenarios [23]. Another example is Federated AI Technology Enabler (FATE) [24] which is an industrial-oriented federated framework that focuses on real-life applications. Unfortunately, it is not user-friendly for beginners and academics due to their difficult environment setup and heavy system architecture. The most known framework is TensorFlow Federated (TFF) [1], an open-source framework that provides a diversity of implemented models and enables the users to apply FL without the need for implementing FL algorithms. As a limitation this solution can be executed only on a single machine, although it facilitates the simulation of the distributed training of FL models.

In our case, we selected the Flower framework that offers a reliable machine learning implementation of the core components of a FL system [2]. Moreover, with heterogeneous computation, memory, and network resources, Flower supports extending FL implementations to mobile and wireless clients. A huge convenience of Flower is that can be used with any machine learning framework, for example, TensorFlow, MXNet, scikit-learn, Pandas, or even raw NumPy for federated analytics. Furthermore, selected framework supports many learning schemes mainly for deep learning and tensor based algorithms but it does not support a huge collection of popular tree based algorithms: Decision Trees, Gradient Descent Trees, XGBoost.

Presented FL frameworks apply different strategies during the federated training. A strategy is the main FL algorithm that is executed on the server side. Strategy describes the configuration procedure and sampling of clients that participate in training. Also, it is responsible for aggregation of the updates and model evaluation. Currently there are several proposed strategies in federated ML implementations that are available on Flower. Federated Averaging (FedAvg) is the most popular strategy, which was initially proposed by [27] and provides high performance improvement according to communication cost. In particular strategy, the server randomly selects a subset of clients and forward the initial parameters. Each client updates those

parameters via training on local data and provides the updated parameters to the server. Finally, server aggregates the new parameters via weighted average computation and spreads the updated weights to all clients [15]. FedAvg, one of the standard federated optimization techniques, is frequently challenging to adjust and has undesirable convergence behavior. For this reason many researchers proposed other federated strategies. One of them is FedProx [35], where authors manage to develop an updated strategy that improves generalization and re-parameterization of FedAvg. Such an approach allows extending FL in heterogeneous network conditions. During the past years, the research community presented a range of different strategies such as FedAdagrad, FedYogi and FedAdam [32]. Fortunately, all the above strategies are available on Flower framework but still the users have the potential to create a new strategy.

Flower [2] offers a very pragmatic abstraction for performing a wide range of actions that are typically involved in a federated learning task. On top of this abstraction lies the `strategy` abstract class which can be implemented in order to define how the server will coordinate with the clients. A detailed description of this abstraction is presented in the official documentation of flower [1]. Briefly, in each round of FL, two procedures are taking place: *Train* and *Evaluation*. During *Train*, clients fit the model on their local data and return the updates to the server and during *Evaluation* clients evaluate the performance of their models. In Flower's terms (Figure: 1.1), the server sends `Fit Instructions` to a predefined set of clients and each client returns a `Fit Response`. After all clients have returned a `Fit Response`, the server aggregates all responses and builds or adjusts the central model. The next part is the *Evaluation* where the server sends a set of `Evaluation Instructions` to each client that took part in the *Fit* procedure and receives a `Evaluation Response`. Finally, the server aggregates all `Evaluation Responses` and assesses the overall evaluation of the model. The server then proceeds to the next round of the federated learning process . It is important to note that the total number of rounds is a predefined parameter that has to be set at the beginning. Overall, Flower offers a rather rigid server-client communication schema that can be used without any changes if we want to use models like Logistic Regression, Support Vector Machines (SVM) or Gaussian Mixture Models (GMM).

1.3 Federated Machine Learning algorithms

There are various machine learning algorithms that can be used for federated learning, depending on the specific requirements and constraints of the federated learning scenarios, as well as the type and quality of the available data. In machine learning there are two main categories of learning, supervised and unsupervised. Each category represents a set of algorithms that can be adjusted to federated learning. Some of the models are challenging to modify for federated setting but algorithms

[1] Flower - Implementing Strategies `https://flower.dev/docs/implementing-strategies.html`.

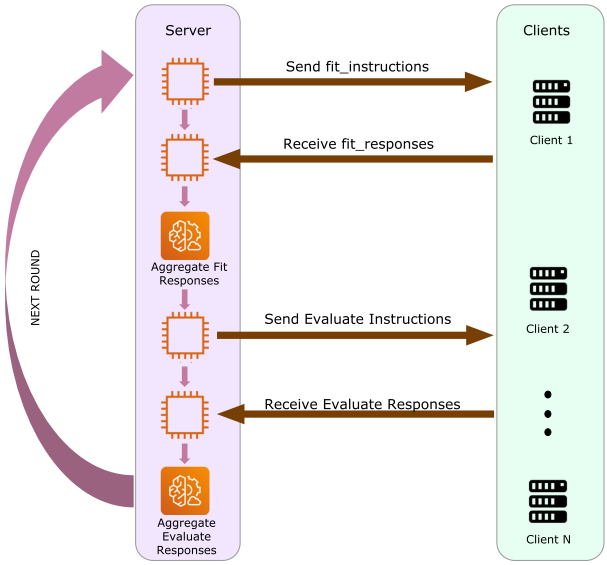


Figure 1.1: Flower Strategy.

like logistic regression and Neural Networks(NN) it's simple to implement because the structure of the model's parameters is predefined. Since the server and clients, for these models, share a set of parameters that don't reveal any characteristics of the raw data distribution. Most of the existing federated learning frameworks have already implemented these models, such as FATE and FedMl that presented in the previous section. These frameworks also include NN models but other frameworks, like TFF, specialized in NN and Deep Learning federated algorithms. The majority of the existing federated frameworks or federated algorithms deal with NN due to the popularity of them and the success in centralized scenarios. As a result, less research has focused on tree based models or even in clustering models.

Compared to the other models, implementing tree-based models within a Federated Learning environment required researchers to overcome a number of unique fundamental problems. These difficulties include developing models, in contrast to neural networks where the model structures are predetermined, where the underlying structures are not initially defined [29]. Also, researchers deal with the kind of secure information that is required to be exchanged between each of the contributing clients and the server in order to aggregate diverse tree based model structures (with no-information leakage). One of the key issues in training a gradient boosted decision tree is to find the best feature and value to split using the computed gain score. FATE framework [24] is the first framework that provides implementations of Gradient Boosted Trees (GBT) mainly on vertically partitioned data (Vertical FL) and even supports clustering algorithms like K-Means.

Flower framework, as we mentioned earlier, provides a variety of federated implementations based on NN and Deep learning models. On the set of supervised algorithms, only the Logistic Regression model is implemented on a particular

framework and there is no implementation of tree based models. Furthermore, on unsupervised approach, Flower lacks clustering algorithms like K-Means, GMM or density based algorithms like DBSCAN. Based on that, our goal is to integrate an unsupervised model (GMM, DBSCAN) and a tree based model, more specifically XGBoost, into the Flower framework. The federated implementation of the GMM model makes the easy part of this study with the exchange of model weights between client and server. On the other hand, the federated XGBoost or DBSCAN was more challenging. At this point, we implemented a new federated server strategy in Flower in order to succeed this challenge.

1.4 Purpose of the study

The purpose of this study is to investigate the potential of "federalized" machine learning algorithms in solving the problem of data privacy and security in distributed and decentralized systems. First of all, we provide a novel, federated, implementation of the XGBoost algorithm. The proposed implementation should have the same architecture as the traditional XGBoost algorithm and also satisfy the typical FL restrictions, such as: multiple clients, a server that orchestrates the learning process, and the server has no access to individual data points and cannot infer the training data. Additionally, the proposed implementation should be as efficient as possible compared to traditional "un-federalized" XGBoost implementations. Also, we need to provide a programmatic framework that will allow programmers to "federalize" any machine learning algorithm. The proposed framework should: be easy to adopt, offer a simple abstraction, and imply minimal requirements on the ML implementation. Finally, we provide a novel federalized clustering algorithm. The algorithm is a combination of two well-known data clustering methods, namely DBSCAN and GMM. DBSCAN is a very efficient distance-based clustering algorithm which does not generate a transferable model and thus cannot be used without revealing information of at least some of the training data samples. On the other hand, GMM is a clustering method of mediocre efficiency which generates an easily transferable model that does not reveal details about the training data samples. The proposed clustering method should combine the advantages of these methods without being hindered by their disadvantages.

Chapter 2

Datasets

We have selected a diverse set of datasets that cover a wide range of domains and characteristics, in order to ensure that the proposed algorithms are thoroughly tested and evaluated. These data sets include both real-world and synthetic data, and have been chosen based on their relevance and suitability for the algorithms under investigation. We used the real-world datasets on federated XGBoost classification and the synthetic datasets on the clustering algorithms (GMM and DBSCAN).

More specifically, in order to evaluate the performance of federated XGBoost, we used four standard binary classification datasets (Table: 2.1). The Wisconsin breast cancer dataset is a collection of clinical measurements from breast cancer tumors, which is publicly available from the UCI Machine Learning Repository [11]. It contains information about the size, shape, and other characteristics of the tumors. The dataset includes information on more than 500 patients, where each patient is represented by a record containing 30 clinical measurements. The measurements include characteristics of the tumor, such as its radius, texture, smoothness, and perimeter. The dataset, also, includes a binary outcome variable indicating whether the patient's tumor was benign or malignant with 212 or 357 number of patients respectively. Overall, the Wisconsin breast cancer data-set is a valuable resource for researchers studying breast cancer and developing machine learning algorithms for predicting tumor characteristics and outcomes.

The heart disease dataset [21] is a collection of clinical data related to heart health. It typically includes information on patients' medical histories and measurements taken from medical tests, such as electrocardiograms (ECGs) and blood tests. The dataset also includes information on the patients' demographics and the category of the chance to have a heart attack. On this dataset we have 303 samples (165 and 138 samples on each class) and 13 features, like the number of major vessels, the presence of chest pain or the absence, the highest heart rate that was recorded and others.

The Pima Indians Diabetes Database [11] contains clinical data of female patients from the Pima Indian tribe. The dataset includes demographic information for each patient and clinical measurements, such as their blood pressure, body mass

	Samples	Features	Positive samples	Negative samples
Breast Cancer	569	30	357	212
Heart disease	303	13	165	138
Smoking	55.692	26	20.455	35.237
Diabetes	768	8	268	500

Table 2.1: Binary classification datasets selected for XGBoost model.

index (BMI), and skin thickness. The dataset also includes a binary outcome variable indicating whether the patient developed diabetes or not. Totally, we have 700 patients, with each patient represented by a record containing eight clinical measurements. Also, this dataset has unbalanced in the target class with 500 non-diabetic patients and 268 that developed diabetes.

The last dataset is a collection of basic health biological signal data of smoking [16]. Includes information for the age, height and gender for each patient and other clinical measurements, such as blood pressure, dental caries, etc. In total, we have 26 measurements for each of the 55.692 patients and the goal is to use bio-signals to assess whether smoking is present on the patients or not. Is an unbalanced dataset with a massive number of samples in both categories (non-smoking and smoking patients with 35.237 and 20.455 samples respectively).

In addition to these datasets, we will also use several synthetic datasets which will be generated to evaluate the performance of the proposed clustering algorithms in different scenarios. These synthetic datasets will be designed to mimic different "interesting" types of real-world datasets and will be useful in evaluating the robustness and scalability of the GMM and DBSCAN algorithm. We created six datasets from [30] with 2 dimensions to check the clustering algorithms.

The first dataset is one that we can create from scikit-learn with the use of `make_circles` function that creates a circular-shaped binary dataset. The samples are generated in such a way that they are arranged in two concentric circles with the samples in the inner circle belong to one class and the samples in the outer circle belong to the other class. The second dataset has the structure of two interleaving half-circles (`make_moons`) where samples are arranged in two half-circles, one on top and one on bottom. Also, we used the function `make_blobs`, a dataset generator in scikit-learn that creates a multi-class dataset. It takes in several parameters such as $n\_samples$, $centers$, and $cluster\_std$, which can be used to control the number of samples, number of centers, and standard deviation, respectively. The samples are generated randomly around the centers provided with a Gaussian distribution and each center corresponds to a different class. From the same library, we derived one more dataset with different standard deviation of each cluster by providing an array of variances as an argument. When you specify varied variances, it means that the clusters will have different standard deviations or spread of data points around the center. The fifth dataset is anisotropically distributed data, which refers

to data that is not evenly distributed across all dimensions. It is characterized by a non-uniform distribution of samples along different axes or features. This type of data distribution can be caused by several factors, such as natural phenomena, measurement errors, or experimental design. For example, in a 2-dimensional space, an isotropic distribution would be a circular distribution around the center, while an anisotropic distribution could take the form of an elongated ellipse, where the data is more concentrated along one axis than the other. In higher-dimensional spaces, the anisotropy can take more complex forms. Finally, the last dataset is a random dataset without structure, derived from the `random_rand` function in python. All of the datasets described above have 1000 samples and 2 features.

The presented datasets is often used in machine learning and data mining research, as they provides a relatively large and well-organized collection of data for testing and developing predictive models. These includes a wide range of information that can be used to study the presence or absence of a disease based on clinical measurements. The results of the algorithms performance on these data sets will be presented and analyzed in later sections, providing a comprehensive evaluation of the proposed algorithms and their capabilities. This will help us to understand the strengths and limitations of the proposed algorithms and identify areas for future work.

Chapter 3

Federated XGBoost

3.1 Preliminaries of XGBoost

Ensemble learning is an interesting meta-learning strategy where a large number of relatively weak simple models are combined in order to obtain a stronger ensemble prediction [31]. The most prominent examples of such machine-learning ensemble techniques are random forests [5] that use the "Bagging" technique (sampling with replacement from the data set and averaging of the sub-models) and have gained a lot of popularity in recent years. Instead of the simple model averaging technique that Random forests use, the so called "boosting" technique and the AdaBoost algorithm in ensemble learning [36] follows an iterative, "stage-wise" additive approach where a new model is added and trained based on the errors of the whole ensemble in the current iteration. A statistical view of the boosting techniques led to the introduction of the gradient boosting machines [12, 13] and gradient tree boosting when decision trees are used as the "weak" learners. Under this statistical framework, the new base-learners are selected according to their correlation with the negative gradient of the loss function, associated with the whole ensemble. Here, we are going to quickly present XGBoost which is one of most famous examples of gradient tree boosting algorithms [7].

In the Gradient Boosting Decision Trees (GBDT) setting a sequence of decision trees are trained. Formally, given a loss function l and a data set with n instances and d features $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}$, where $|\mathcal{D}| = n$, $\mathbf{x}_i \in \mathbb{R}^d$, and $y_i \in \mathbb{R}$, GBDT minimizes the following objective function [7]:

$$\tilde{\mathcal{L}} = \sum_i l\left(\hat{y}_i, y_i\right) + \sum_k \Omega\left(f_k\right)$$

where $\Omega(f) = \gamma T_l + \frac{1}{2}\lambda|w|^2$ is a regularization term to penalize the complexity of the model. Here γ and λ are hyper-parameters, T_l is the number of leaves and w is the leaf weight. Each f_k corresponds to a decision tree. Training the model in an additive manner, GBDT minimizes the following objective function at the t-th iteration.

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} \left[g_i f_t\left(\mathbf{x}_i\right) + \frac{1}{2} h_i f_t^2\left(\mathbf{x}_i\right) \right] + \Omega\left(f_t\right)$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l\left(y_i, \hat{y}^{(t-1)}\right)$ and $h_i = \partial^2_{\hat{y}^{(t-1)}} l\left(y_i, \hat{y}^{(t-1)}\right)$ are first and second order gradient statistics on the loss function. The decision tree is built from the root until reaching the restrictions such as the maximum depth. If I_L and I_R are the instance sets of left and right nodes after the split and letting $I = I_L \cup I_R$, then the "gain" of the split is given by

$$\mathcal{L}_{\text{split}} = \frac{1}{2} \left[\frac{\left(\sum_{i \in I_L} g_i\right)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{\left(\sum_{i \in I_R} g_i\right)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{\left(\sum_{i \in I} g_i\right)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma \tag{3.1}$$

Now if we assume that the loss function is "log-loss", one of the most common functions for binary classification:

$$l(\hat{y}_i, y_i) = -\left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right]$$

then the first and second order gradients of this loss faction are $g_i = \hat{y}_i - y_i$, or else the residuals and $h_i = \hat{y}_i(1 - \hat{y}_i)$. If we substitute these values in $\mathfrak{L}_{split}$ then the optimal split is the one that maximizes the expression:

$$L_{\text{split}} = \text{SimilarityScore}_{\text{left}} + \text{SimilarityScore}_{\text{right}} - \text{SimilarityScore}_{\text{root}}$$

where the $SimilarityScore$ for a collection of samples is:

$$SimilarityScore = \frac{\left(\sum_{i=1}^{n} g_i\right)^2}{\sum_{i=1}^{n} h_i + \lambda}$$

or else:

$$SimilarityScore = \frac{\left(\sum_{i=1}^{n} residual_i\right)^2}{\sum_{i=1}^{n} \hat{y}_i * (1 - \hat{y}_i) + \lambda} \tag{3.2}$$

Then, when the maximum depth of the tree reached, we calculate the output values of the leaves, or the weights w_j^* of the leaf j. According to the XGBoost publication this value is equal to:

$$w_j^* = \frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_j + \lambda}$$

The computation of the output value happens after the optimal tree has been constructed. Given the same loss function l and substituting g_i and h_i in the formula above we have:

$$w_j^* = \frac{\sum_{i=1}^{n} residual_i}{\sum_{i=1}^{n} \hat{y}_i(1 - \hat{y}_i) + \lambda}$$

3.2 XGBoost on a Federated Learning setting

As described previously, during the process of construction of a tree, XGBoost (and other Gradient Boost algorithms) deal with the problem of locating an optimal split for the residuals of the data set. In XGBoost splitting is applied by locating a threshold that maximizes the equation (3.1), which simplifies to the following since γ is constant:

$$\mathfrak{L}_{split} = \frac{1}{2}\left[\frac{\left(\sum_{i\in I_L} g_i\right)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{\left(\sum_{i\in I_R} g_i\right)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{\left(\sum_{i\in I} g_i\right)^2}{\sum_{i\in I} h_i + \lambda}\right]$$

Where I_L, I_R are the instances (i.e. samples) of the data on the left and the right nodes respectively after the split, and I are all the instances before the split. Also, g_i and h_i are the first and second order gradient statistics of the loss function: $l(\hat{y}_i, y_i)$. In this function $\hat{y}_i$ is the predicted probability of sample i and y_i is the target of sample i. Finally, λ is a regularization parameter ($\lambda > 0$), the greater the λ the more pruning is applied from the algorithm.

In order to locate the optimal split of a leaf in the tree, we just need to find: (i) sums of the residuals (for g_i) and (ii) the sum of the likelihoods of the predictions (for h_i) based on the equation 3.2. XGBoost locates this optimal split through an exact Greedy algorithm. Ultimately this means that it just applies all possible splits. A large part of the optimizations that are described in the XGBoost algorithm are dealing with substituting the greedy algorithm with heuristics that limit the number of possible splits. Here we present a split finding algorithm that is suitable for federated learning.

An ideal solution would be for all federated clients to transmit all g_i and h_i values for each split to the master node server and then the server locates the optimal split as suggested in [39]. This method has the disadvantage of revealing too much information regarding the data in each client. For example in the first iteration the server would get the residuals (g_i) for every instance for every client. Assuming an initial prediction probability of 0.5, it could simply check if the sample belongs to the positive or negative class by just checking if the residual is positive or negative. Furthermore, this requires a lot of communication with the central orchestrating node that will negatively affect the training time.

Here we suggest the following approach. Each client computes the S different splits that yield the top S $\mathfrak{L}_{split}$ values, where S is a configuration hyperparameter. For example if $S = 10$ then each client computes the splits that produce the top 10 $\mathfrak{L}_{split}$ values. Then each client returns to the server three types of data: (i) the splits, (ii) their corresponding values of $\mathfrak{L}_{split}$ and (iii) the number of samples that belong to the root node of this split for this client. Assume that SP_s is the s-th split, C is the number of federated clients, $\mathfrak{L}_{s,p}$ is the $\mathfrak{L}_{split}$ value returned from the p-th client when it applies the s-th split. Also assume that N_p is the number of samples that the root node contains in the p-th client (this is before the split, so it

is irrelevant to the split). After receiving this data the server computes the split that maximized the following expression:

$$\underset{SP_s}{argmax} \left\{ \sum_{p=1}^{C} N_p \mathfrak{L}_{s,p} \right\} \tag{3.3}$$

Or else it locates the split that contains a high number of samples over many nodes (N_p) and also high values of $\mathfrak{L}_{split}$. After obtaining the optimal split the server transmits that to the clients so that they can continue building the tree. Again some information about the distribution of the data can be deduced, but important information like class values is not leaked.

In contrast to the optimal split, the computation of weight w_j^* of a leaf j haven't got any "search" that needs to take place. We notice that the output value of a leaf can be computed from the sum of the residuals and, also, from the sum of the likelihoods of the predicted probabilities that this leaf contains. Therefore after a split, each client can just transmit these two sums to the server. The server adds all sums and can compute the w_j^* for all leaves which transmits back to the clients. It is important to note that during this federated process all clients 'work' on constructing the same set of trees.

3.3 Implementation

In this section, we describe in details our federated XGBoost implementation. On purpose, we will omit the description of the technical aspects of server and client communication. We will explore in details the communication in Chapter:4. At the beginning server and clients exchange a set of parameters. Particularly, server sends the parameter S to all clients and the clients sends to the server the number of samples on their dataset, the initial probabilities and the learning rate. The initial probabilities (Pr_{init}) computed by the formula:

$$Pr_{init} = \frac{P}{P+N}$$

where P is the positive and N the negative number of samples. Since each client has a different number of positive and negative samples, the server computes the weighted average of Pr_{init} over all clients. Similarly each client may have different fit parameters. For example the learning rate can be set with different values in each client. These parameters are transmitted back to the server and the server aggregates them into a single value. By default the aggregating function is the weighted average. After the exchange of these parameters, the iterative process of the gradient descent begins. In each iteration a new binary tree is generated that fits the data to the current residuals. The challenge here is to synchronize all clients so that a single tree is generated. If we had a single client (unfederated), then we would have a depth-first recursive process where in each node we would

	Data
Client 1	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
Client 2	1, 2, 3, 4, 9, 10, 11, 12
Client 3	9, 10, 11, 12, 13, 14, 15, 16

Table 3.1: Data in each client (example).

compute the optimal split. Then we would split all data to two groups: these data points for which the target feature is greater than the optimal split and these data points for which the target feature is less than the optimal split. Therefore, we would continue with the process recursively until a stopping criterion is met. In our case, each client has different data points and all clients are building the same tree. Subsequently, given a path $Path$ from the root to a node, a client might have data points in this node, whereas another client might not have data points (or even nodes) on this specific path. For example, assume that we have three clients with the data presented on table 3.1. At the beginning we ask all clients to send the top S best splits. A split is in the form `target < midpoint` where `target` is the value of the target feature and `midpoint` is the average of two adjacent values of the target feature. For example a split might be: `target < 10.5`. The server receives S splits from each client and chooses the best based on the information that each client provides. As we shown, this information includes the midpoints, the gain of the split $\mathfrak{L}_{split}$ (formula 3.1) and the number of residuals for each feature that are necessary for the server in order to apply the formula in 3.3. After this computation, the server finds the feature and the midpoint with highest value and sends this information back to clients. Based on the described procedure, each client will receive the same splitting criterion according to global information. Back to our example, let's assume that this split is: `target < 8.5`. The clients split their data in two groups, lower and higher than the midpoint of the selected split. This generates the following trees in each client, shown in Figure: 3.1.

For each split that the client transmits to the server, the client stores the following information that is kept private:

1. The predicted probabilities. As we mentioned earlier, making a first prediction is the first stage in fitting XGBoost to training data. When the first tree is complete, we are ready to make new predictions, by starting with the initial probabilities/predictions. The new predicted probabilities for each observation is equal to the *initial_probabilities* plus the *output_value* scaled by the *learning_rate*. That is:

   $$new\_predictions = initial\_predictions + Learning\_Rate * Output\_value$$

2. The residuals ($g_i = \hat{y}_i - yi$) or else the difference between the target class and the prediction for each sample i from the previous iteration.

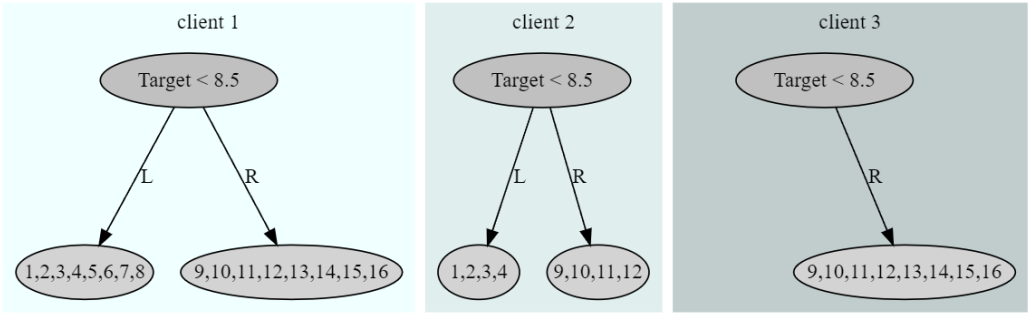


Figure 3.1: Server sends the best split to the clients (`Target<8.5`) and they split their data into two different groups. 'Left' from a node contains all data points that have lower feature values than the optimal split and 'right', contains all data points that have values greater than the optimal split.

Cl.1	L	LL	LLL	LLR	LR	LRL	LRR	R	RL	RLL	RLR	RR	RRL	RRR
Cl.2	L	LL	LLL	LLR	-	-	-	R	RL	RLL	RLR	-	-	-
Cl.3	-	-	-	-	-	-	-	R	RL	RLL	RLR	RR	RRL	RRR

Table 3.2: The order of the paths with which the clients are instructed to send their best splits in our example. Not all clients have nodes in all paths (denoted with -).

In the next step the server asks the clients to send the 'next path' of their trees. The next path is a depth-first enumeration of all the paths in the current trees of the clients. In our example the first client responds with `L`, the second with `L` and the third with `R` (Table 3.2). Notice that the third tree does not have any nodes 'left' from the root node, therefore its 'next path' is `R`. After receiving all 'next paths' from the clients, the server selects the minimum path according to the lexicographic order (i.e all the following orderings are `True`: $L < R$, $L < RR$, $LR < LRL$). In our case the minimum path is `L`. Since only clients 1 and 2 have this path, only these clients take part in the next round. Therefore, the server instructs only clients 1 and 2 to receive their next best S splits, that exist in the `L` path. In our case this would generate the following trees as showed in Figure: 3.2. Once the `L` branch has finished for clients 1 and 2, then the next branch is `R` for which all three clients have nodes. Here we notice that although each client has different data, all clients contribute in constructing the same tree. After the generation of a tree, each client measure the value of the log-loss function and compare it with the value of the previous tree. This process stops until a minimum criterion is met. For example if the log-loss differences between new and previous tree is lower than 0.004. For the sake of simplicity we assume that the tree shown in Figure:3.2 is the final. This could happen for example if we have set a maximum tree depth of 2 as a convergence criterion. After convergence, fitting proceeds to the next part of the algorithm which is the normalization of the trees from all clients.

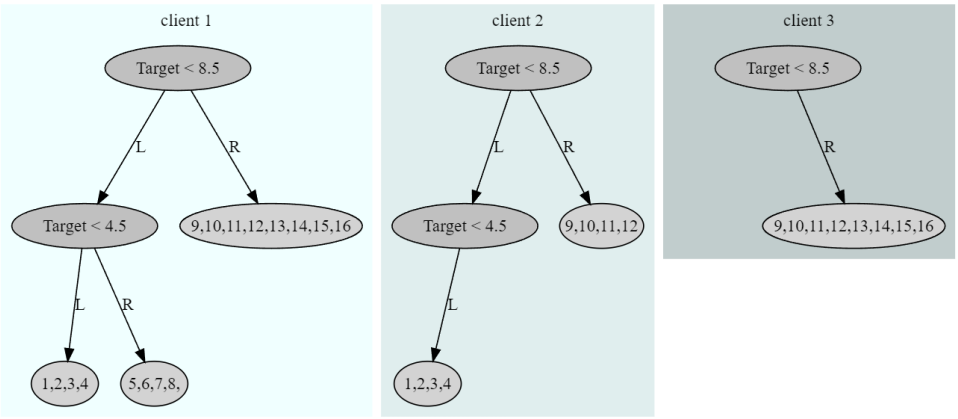


Figure 3.2: Server sends the next best split to the selected clients (`Target < 4.5`) and they split their data, again, into two different groups.

As described before, all clients have constructed trees with the same structure, nevertheless each client has different data on the leaves of each tree, and for some clients some branches might be missing (see Figure: 3.2). Also, each client has multiple trees where each created during an i_{th} iteration with a specific rule. For this reason we can normalize only the trees that were created during the same i_{th} iteration. The function `TREE_NORMALIZER` (Algorithm 1) is responsible for collecting i_{th} trees across all clients. The normalization of these trees is implemented in the function `RECURSIVE_TREE_NORMALIZER` (Algorithm 1). This function recursively traverses all i_{th} trees in parallel starting from the left branches to the right. At the end of this procedure will generate a single i_{th} tree where the output nodes contain the summation of the individual tree residuals. The nodes rule stays the same since the i_{th} trees among all clients created based on the aggregated splits. After the normalization of the trees on the server, we send a message to the flower server in order to stop the federated rounds. When we have a single model on the server, we have the ability to store the model and use it for further usage.

After normalization, the federated server, holds a unique set of decision trees as if we had run a non-federated XGBoost implementation. Therefore, the part in FIN strategy (see Chapter:4) that yields predictions of unseen data is exactly the same as the original implementation. Briefly, to predict the sample target we traverse the decision tree starting from the root node. Based on the node decision rule, we follow the path of the tree until the output leaf is reached. This leaf contains the summary of all residuals and covers based on the training data. At this point, we compute the output value of the tree as shown in Algorithm 2, line 3. This procedure is repeated for all the trees and finally, we compute the target probability based on the summary of output values multiplied by learning rate.

Algorithm 1 Recursive tree normalization algorithm

```
1: function RECURSIVE_TREE_NORMALIZER(nodes)
2:     if nodes is empty then
3:         return
4:     end if
5:     nodes_L ← [ ]                                   ▷ [ ] is an empty list
6:     nodes_R ← [ ]
7:     normalized_node.residuals ← 0
8:     for node in nodes do
9:         if node is an output node then               ▷ or else this is a leaf
10:            normalized_node.residuals ← normalized_node.residuals + node.residuals
11:        else
12:            nodes_L.append(node.L)    ▷ Add to nodes_L the left branch of node
13:            nodes_R.append(node.R)    ▷ Add to nodes_R the right branch of node
14:            normalized_node.rule ← node.rule
15:        end if
16:    end for
17:    normalized_node.L ← RECURSIVE_TREE_NORMALIZER(nodes_L)
18:    normalized_node.R ← RECURSIVE_TREE_NORMALIZER(nodes_R)
19:    return normalized_node
20: end function
21: function TREE_NORMALIZER(clients)
22:    normalized_trees ← [ ]
23:    N ← Maximum number of trees over all clients
24:    for k ← 1 to N do
25:        root_k ← [ ]
26:        for client in clients do
27:            tree_k ← k_th tree of client
28:            root_k.append(tree_k.root)
29:        end for
30:        normalized_k ← RECURSIVE_TREE_NORMALIZER(root_k)
31:        normalized_trees.append(normalized_k)
32:    end for
33:    return normalized_trees
34: end function
```

Algorithm 2 Prediction

function RECURSIVE_TREE_OUTPUT(X, *node*, SUM_R, SUM_C)
  if *node* is an output node **then** ▷ or else this is a leaf
    return $\frac{SUM_R + node.residuals}{SUM_C + node.cover}$
  else
    if $node_{rule}(X) \rightarrow$ go left **then**
      $R \leftarrow$ RECURSIVE_TREE_OUTPUT(X, *node.L*, *node.residuals*, *node.cover*)
    else if $node_{rule}(X) \rightarrow$ go right **then**
      $R \leftarrow$ RECURSIVE_TREE_OUTPUT(X, *node.R*, *node.residuals*, *node.cover*)
    end if
    return R
  end if
end function
function PREDICT(X, LR)
  $log\_odds \leftarrow initial\_log\_odds$
  for *tree* in *normalized_trees* **do**
    $output \leftarrow$ RECURSIVE_TREE_OUTPUT(X, *tree.root*, 0, 0)
    $log\_odds \leftarrow log\_odds + LR * output$
  end for
  $prob \leftarrow \frac{e^{log\_odds}}{1+e^{log\_odds}}$
  return *prob* ▷ Or transform to binary to use for classification: $sign(prob - 0.5)$
end function

3.4 Experiments

In order to measure the effectiveness of the proposed federated XGBoost algorithm we design experiments over selected classification datasets (presented in Table: 2.1). Furthermore, we are interested in comparing the performance of the proposed method in comparison with traditional XGBoost implementation [8].

In order to proceed with the analysis, each dataset is separated in two portions, train and test (also known as hold-out) set. The train portion will be used to train both models (federated and unfederated XGBoost) during the *K-fold Cross Validation* and the test set to assess the performance of each model. *K-fold Cross Validation* is a data re-sampling method to assess the generalization ability of predictive models and to prevent overfitting. During the *K-fold Cross Validation*, data set partitioned into K disjoint subsets of equal size. The *K-1* folds represents the training set for both federated and unfederated XGBoost. In the case of federated XGBoost, we split the *K-1* folds (the training set) into N parts, where N represents the number of clients. In our case the value of K is equal to 3. So, classifiers trained on the 2 out of three folds and the performance measured on the remaining *K-th* fold for federated and unfederated cases. More specifically for federated XGBoost, when the server has the aggregated final model derived from the clients, we measure the performance on validation sets for this model ($FedM\_val$). Then we train both methods on training data (80% of the entire dataset) and the performance evaluated on the initial test set (20% of the data). We describe the entire pipeline in Figure: 3.3. The train-test split and the splits for *K-fold Cross Validation* are made by preserving the percentage of samples for each class (*stratified folds*).

The performance metrics that was used to compare the performance of the models/methods was the Area Under the Curve of Receiver Characteristic Operator (ROC-AUC), sensitivity, specificity and accuracy. The ROC curve is an evaluation metric for binary classification problems. Is a probability curve that plots the True Positive Rate (TPR) against False Positive Rate (FPR) at various threshold values. The Area Under the Curve (AUC) is the measure of the ability of a classifier to distinguish between classes and is used as a summary of the ROC curve. The higher the AUC, the better the performance of the model at distinguishing between the two classes [3]. The Sensitivity (Sens) is the ratio of correctly predicted positive observations to all observations in the actual class. Specificity (Spec) is the ratio of true negative observations to all observations in the actual class. And finally, accuracy (Acc) is the ratio of correctly predicted observation to the total observations and it can be used correctly in the case of balanced data.

3.4.1 Scalability measurements

In this section, we investigate the scalability in terms of the number of clients. At this experiment, we kept a fixed number of samples (n_samples) that are shared between N clients and each client has $n\_samples/N$ samples. According to this

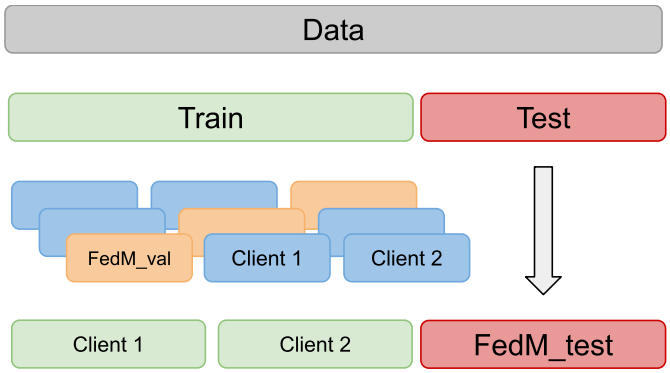


Figure 3.3: Pipeline of XGBoost experiments.

formula the increase of clients will reduce the number of training samples that each client has. For this experiment we selected multiple dataset in order to reduce the random effect of a selected dataset and show the generalization capability of the presented approach. For each dataset, we proceed with the K-Fold Cross Validation pipeline.

Furthermore, we investigate the effect of training samples volume on the final model performance. For this purpose, we manage to implement an experiment where a diverse range of samples are used during the model training. In this case we require a dataset with a high volume of samples. For this purpose, we selected the smoking dataset. As an experiment scenario, we selected samples of size 500, 2.000, 4.000, 8.000, 16.000 and 32.000 where the class ratio is maintained for a proper model training. We measure the performance of both methods over identical samples sized data in order to compare the accuracy of each model over the different sample size. In the case of the federated approach we manage to perform experiments with the use of 2 and 5 clients for comparison purposes.

3.4.2 Impact of class imbalance

In real world scenarios the datasets are mostly unbalanced. An unbalanced dataset refers to a dataset where the number of samples in one class is significantly different from the number of samples in another class. In other words, it is a dataset where the distribution of classes is not equal or balanced. For example, a dataset that is used to predict the presence or absence of a disease, where the majority of the observations are healthy individuals and a small number of observations are individuals with the disease, would be considered an unbalanced dataset. In unbalanced datasets, the classifier tends to predict the majority class more often, resulting in poor performance for the minority class. This can lead to a bias in the model towards the majority class, resulting in a poor prediction of the minority class.

To measure how our model is affected by such class imbalance we developed a set of experiments where we manipulate the class imbalance. For this purpose

	Validation				Test			
	Acc	AUC	Sens	Spec	Acc	AUC	Sens	Spec
2 clients	0.927	0.984	0.954	0.924	0.929	0.990	0.930	0.886
3 clients	0.934	0.981	0.947	0.917	0.973	0.994	0.986	0.975
5 clients	0.903	0.976	0.943	0.916	0.921	0.987	0.930	0.883
7 clients	0.898	0.960	0.905	0.870	0.903	0.985	0.888	0.829
10 clients	0.863	0.972	0.863	0.817	0.912	0.986	0.902	0.847
Original	0.951	0.989	0.964	0.944	0.956	0.994	0.944	0.911

Table 3.3: Measurements on Breast Cancer dataset with respect to the number of clients.

we selected a dataset with large volume of samples that is capable for diverse range of imbalance manipulation. At this point, we choose the largest dataset that we have, the Smoking dataset. As we shown, the smoking dataset has 20.455 patients/samples on the positive class (smokers) and 35.237 patients on the negative samples (non-smokers). To create balanced dataset, we decrease the number of samples on the negative class by choosing randomly 35.237 - 20.455 = 14.782 sample to throw from the entire dataset. So we used the new balanced data to test our implementation. Also, we create 4 unbalanced datasets from the original smoking dataset, where we sample randomly, 60%, 70%, 80% and 90% of samples that belongs to negative class and the remaining portion belongs to the positive class. These analysis follow the K-Fold Cross Validation pipeline describe at the beginning of this chapter.

3.5 Results

As described in previous sections we managed to develop multiple experiments for multiple scenarios. Initially, we measured the ability of our model to scale over a diverse number of clients with the use of breast cancer, diabetes and heart disease datasets. For the breast cancer dataset, the given Table: 3.3 compare the performance of federated XGBoost with respect to the number of clients and we report the performance of the original unfederated XGBoost. The table has two sets of columns: "Validation" and "Test". The validation columns show the average performance of the model on the validation datasets, and the test columns show the performance of the model on the test dataset. To measure the performance we used various metrics that are shown in the table and include accuracy (Acc), AUC (area under the curve), sensitivity, and specificity. Also, for the federated setting, in the left column we provide the number of clients used in each round. The final row in the table, "Original", shows the performance of the unfederated XGBoost. As you can see, when we increase the number of clients, the performance of federated XGBoost decreases. For example, on the validation data, when we have 2 clients,

	Validation				Test			
	Acc	AUC	Sens	Spec	Acc	AUC	Sens	Spec
2 clients	0.744	0.807	0.560	0.782	0.805	0.862	0.629	0.818
3 clients	0.742	0.807	0.542	0.776	0.785	0.862	0.611	0.807
5 clients	0.684	0.764	0.448	0.733	0.824	0.882	0.703	0.847
7 clients	0.682	0.741	0.406	0.722	0.785	0.875	0.648	0.819
10 clients	0.700	0.767	0.420	0.734	0.746	0.808	0.629	0.801
Original	0.701	0.775	0.537	0.762	0.818	0.877	0.666	0.833

Table 3.4: Measurements on Diabetes dataset with respect to the number of clients.

the AUC score is 0.984 higher than the AUC score on 10 clients (0.972). The same trend we observe also on the other metrics and on the test data. In addition, the performance on unfederated XGBoost ("Original") is slightly better than federated XGBoost. For instance, on the test set, the AUC on the federated XGBoost in 2 clients is equal to 0.990 and the AUC of the original XGBoost is equal to 0.994. Overall, it appears that the model performs well on the validation and test datasets, with high accuracy and AUC values. The sensitivity and specificity values are also generally high, indicating that the model is able to accurately classify positive and negative cases. However, the performance of the model does seem to vary somewhat depending on the number of clients used in the data set.

Similarly, on the Diabetes dataset, we present the Table: 3.4 with the same information as before. With this data, federated XGBoost (with two clients) achieved a better performance, for all metrics, in validation measurements in comparison with the original XGBoost. Nevertheless, on test data the original XGBoost has better performance. For example, the specificity of federated method on 2 clients are equal to 0.782 on validation data and 0.818 on test data in comparison with the specificity on "original" XGBoost which is equal to 0.762 and 0.833 respectively. The same trend is observed in the Heart disease dataset (see Table: 3.5). The performance metrics of federated XGBoost, are higher from unfederated XGBoost on validation set, but on test set we have only better specificity and sensitivity (0.909 and 0.808 vs 0.878 and 0.846).

Beside the scalability in terms of clients we measure the scalability according to the volume of the training data that the model (and each client) has. As mentioned in Section: 3.4.1 we selected the smoking dataset for this purpose where we randomly selected a portion of samples. The rows of the Table: 3.6 represent different numbers of samples that we use, ranging from 500 to 32.000. The columns of the table show different metrics of the algorithm's performance, including the AUC, sensitivity and specificity. Additionally, we provide the results for different numbers of clients compared with the performance of the "original" model. According to our results, all of the implemented methods provide very similar results with small performance differences (in some cases our implementation has better performance, in other cases

	Validation				Test			
	Acc	AUC	Sens	Spec	Acc	AUC	Sens	Spec
2 clients	0.789	0.865	0.818	0.780	0.852	0.918	0.909	0.880
3 clients	0.793	0.897	0.856	0.806	0.819	0.884	0.848	0.814
5 clients	0.752	0.847	0.787	0.746	0.885	0.933	0.939	0.920
7 clients	0.822	0.891	0.901	0.867	0.868	0.922	0.939	0.916
10 clients	0.801	0.870	0.840	0.800	0.819	0.901	0.939	0.904
Original	0.764	0.860	0.803	0.759	0.836	0.927	0.878	0.846

Table 3.5: Measurements on Heart Disease dataset with respect to the number of clients.

the ”original” one). Generally, both methods increase their performances while the number of samples are increased, with the highest AUC values being achieved on the maximum number of samples. However, the sensitivity and specificity values for the models trained on different numbers of clients are relatively similar, indicating that the performance of these metrics is relatively consistent across different numbers of clients.

As mentioned in Section: 3.4.2 we are also investigating the impact of class imbalance over final model performance. In the designed experiment we randomly create 5 dataset with variety of class ratio : 50-50, 60-40, 70-30, 80-20 and 90-10 (Table: 3.7). Relatively to the previous results, the columns of the table show different metrics of the algorithm's performance (AUC, sensitivity and specificity) with respect to the number of clients. For example, in the first row under the "50-50" ratio, we see an AUC of 0.847 which corresponds to 2 clients. On the same dataset the original XGBoost succeeded a higher performance not only in AUC (0.856) but also in the other measurements. In addition, original and federated XGBoost has lower performance when we have extremely unbalanced data (”90-10”), in comparison to an unbalanced dataset of 60% on one class and 40% on another class. Based on the provided results we identify that both methods are affected similarly by class imbalance and provided federated implementation manage to compete the original XGBoost with slightly lower performance (around of 1%).

		Samples					
		500	2000	4000	8000	16000	32000
2 clients	AUC	0.796	0.803	0.817	0.826	0.830	0.848
	Sens	0.533	0.654	0.638	0.659	0.649	0.666
	Spec	0.753	0.793	0.790	0.801	0.798	0.809
5 clients	AUC	0.797	0.803	0.811	0.824	0.829	0.842
	Sens	0.514	0.629	0.672	0.656	0.658	0.684
	Spec	0.745	0.785	0.805	0.799	0.802	0.814
Original	AUC	0.782	0.794	0.812	0.822	0.834	0.854
	Sens	0.549	0.635	0.634	0.643	0.682	0.692
	Spec	0.754	0.783	0.788	0.793	0.812	0.820

Table 3.6: Measurements on smoking dataset with respect to the number of clients.

		Portion of classes				
		50-50	60-40	70-30	80-20	90-10
2 clients	AUC	0.847	0.843	0.840	0.837	0.825
	Sens	0.843	0.720	0.553	0.307	0.06
	Spec	0.816	0.737	0.661	0.578	0.513
5 clients	AUC	0.837	0.835	0.834	0.834	0.824
	Sens	0.839	0.731	0.520	0.277	0.02
	Spec	0.807	0.741	0.647	0.569	0.504
Original	AUC	0.856	0.853	0.846	0.841	0.835
	Sens	0.847	0.740	0.579	0.362	0.127
	Spec	0.824	0.752	0.673	0.596	0.528

Table 3.7: Measurements on smoking balanced/unbalanced dataset (test set).

Chapter 4

FIN Strategy

As we showed earlier, Flower provides a basic server-client communication schema that makes easy the addition of machine learning models in this framework. Indicatively, during a single round, the server can communicate only twice with the clients. This schema is suitable for federated learning algorithms and strategies that do not require any server-client communication during the *Train* procedure. Nevertheless, federated learning, as a generic machine learning method, does not preclude the communication between server and client during training. As mentioned in [17], "the separation of the client computation, aggregation, and model update phases is not a strict requirement of federated learning". One example is split learning [37] where certain layers of deep neural networks are communicated to the server from the clients prior to the completion of the training procedure. Therefore the main predicament here is to implement a Flower strategy that allows for any communication between the server and the clients during the *Train* procedure and this is what the `FIN` strategy is. The main purpose of `FIN` is to allow for more complex ML algorithms to be "federalized" through the Flower framework. Or more precisely, as we will show, the `FIN` strategy allows for any federated ML algorithm that handles the server-client communication through two different functions (i.e. `ML_SERVER`, `ML_CLIENT`) to run through Flower. Since Flower is a widely known and supported FL framework, this can help the standardization, benchmarking and easier deployment of more FL algorithms.

4.1 Implementation of FIN

In order for any federated machine learning algorithm to be implemented in FIN, it needs to follow three design principles. The first is that the algorithm should be separated in two parts: the server, referred to as `ML_SERVER` (Algorithm 3), and the client, referred as `ML_CLIENT` (Algorithm 4). There is only one instance of `ML_SERVER` running whereas there are multiple instances of `ML_CLIENT` running, each for every client. The second assumption is that the server orchestrates the complete learning process through a single function called `send_receive_cycle`.

Whenever the server wants to communicate with a client with a given ID, it calls the send_receive_cycle function with two arguments: the ID and the data that it wants to send to that client. The function sends the data to the client and returns its response. The third design principle is that the client is implemented as a coroutine. Namely, whenever the client expects instructions or data from the server, it performs an await until the data is received and yields an appropriate response. The data types and instruction semantics are left to the specifics of the ML algorithm.

Once a ML algorithm follows these design principles it can be included in Flower through the FIN strategy with minimal programming effort. The FIN strategy is separated in two parts according to Flower specifications: The FIN_SERVER part which implements the abstract class server and the FIN_CLIENT which implements the abstract class client. Both server and client are provided by Flower.

More specifically (Figure: 4.1), when FIN_SERVER is initiated it starts a thread which runs the ML_SERVER. The ML_SERVER thread and FIN_SERVER share two thread-safe queues with which they can communicate. These queues are send_queue for sending data to the clients and receive_queue for receiving data from the clients. After that, the FIN_SERVER and the ML_SERVER run in parallel. When the configure_fit function of the strategy is called, FIN_SERVER waits for data to be added in send_queue from ML_SERVER. When ML_SERVER wants to communicate with a client it calls the send_receive_cycle function and passes the ID of the client and the data that wants to receive. The send_receive_cycle adds both ID and the data to send_queue and waits for the receive_queue to get the response from the client. Once the data is added to send_queue, the FIN_SERVER thread awakes and reads the data from this queue. This data contains the client with which the ML_SERVER wants to communicate along with the data that it wants to send to this client. The configure_fit returns a fit_instructions object that contains this information. By having a proper fit_instructions object, the Flower framework can communicate with the corresponding FIN_CLIENT. The FIN_CLIENT that is reached, calls its fit function which contains all data that are sent from the server. This function yields the data to the ML_CLIENT and receives a response that is addressed to the server. Then the fit function, returns a fit_response object that contains this data. The Flower framework, communicates the fit_response to the FIN_SERVER which in turn calls the aggregate_fit function. In the aggregate_fit function, the data from the client are put to the receive_queue. This awakes the ML_SERVER thread that reads the data from the receive_queue and returns with this data from the send_receive_cycle function. This completes the cycle of server-client communication. After the last step, the ML_SERVER thread runs until the send_receive_cycle function is called again which sends data to the send_queue and waits until there are data to read from receive_queue. On the other hand the FIN_SERVER thread calls again the configure_fit function which waits for data to be added in the send_queue.

Algorithm 3 FIN Server

```
1: class FIN_SERVER
2:     send_queue ← Queue                          ▷ Server sends in this queue
3:     receive_queue ← Queue                       ▷ Server receives in this queue
4:     thread ← ML_SERVER.FIT
5:     function CONFIGURE_FIT( )
6:         client_id, data ← send_queue.get()
7:         return fit_instructions(client_id, data)
8:     end function
9:     function AGGREGATE_FIT(fit_response)
10:        receive_queue.put(fit_response.data)     ▷ Return data to the server
11:    end function
12: end class
13: class ML_SERVER
14:    function SEND_RECEIVE_CYCLE(client_id, data)
15:        send_queue.put(client_id, data)
16:        return receive_queue.get()
17:    end function
18:    function FIT( )
19:        while not finished do
20:            computations...
21:            # Send data to client client_id and store the response in r
22:            r ← SEND_RECEIVE_CYCLE(client_id, data)
23:            computations...
24:        end while
25:    end function
26: end class
```

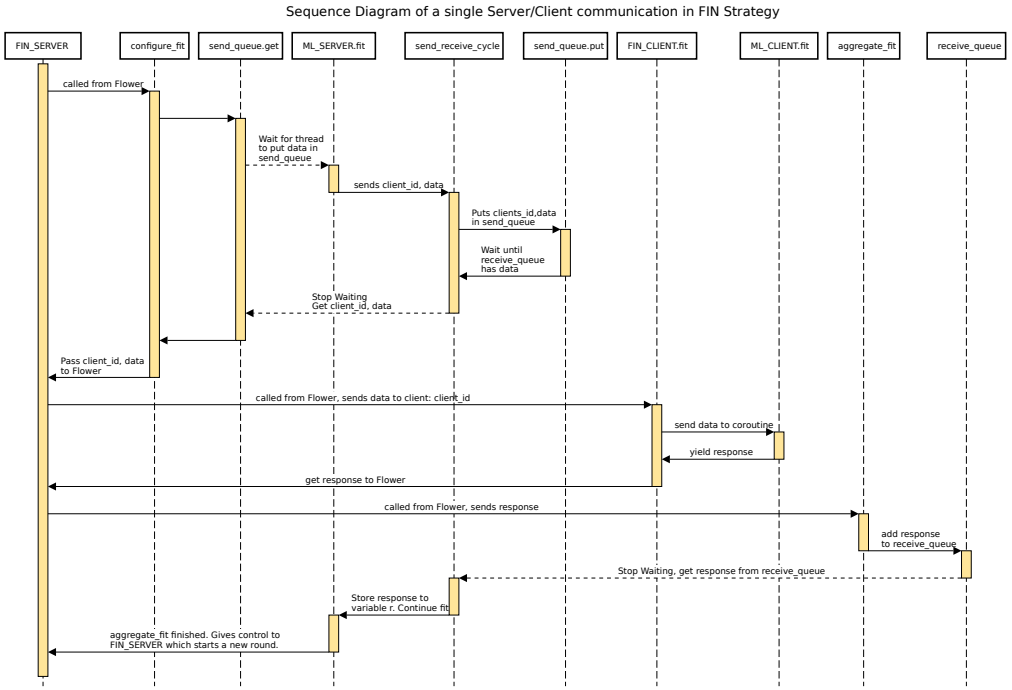


Figure 4.1: Sequence diagram of FIN communication.

Algorithm 4 FIN Client

```
1:  class FIN_CLIENT
2:      client ← ML_CLIENT.FIT( )            ▷ ML_CLIENT.FIT is a co-routine
3:      function FIT(fit_instructions)
4:          response ← client.send(fit_instructions.data) ▷ Send data to co-routine
5:          return fit_response(response)
6:      end function
7:  end class
8:  class ML_CLIENT
9:      coroutine FIT( )
10:         while not finished do
11:             computations...
12:             data_from_server ← yield data_to_server
13:             computations...
14:         end while
15:     end coroutine
16: end class
```

Chapter 5

GMM and DBSCAN on a Federated Learning setting

5.1 Federated GMM

The Gaussian Mixture Models (GMM)[33] is a widely used model for clustering. GMM are useful for understanding and modeling complex data distributions, and they have a wide range of applications. Is a probabilistic model that assumes that the data is generated from a mixture of several different Gaussian distributions. In the GMM algorithm, each cluster is represented by a Gaussian distribution, which is defined by its mean μ_k and covariance σ_k. The model estimates the parameters of these distributions based on the data points in the dataset, and assigns each data point to the cluster with the closest matching distribution [4].

GMM is a parametric probability density function which is represented as a weighted sum of K Gaussian component densities [6], as given from the equation 5.1.

$$p(x) = \sum_{k=1}^{K} \pi_k N(x|\mu_k, \sigma_k) \tag{5.1}$$

where $0 \leq \pi_k \leq 1 \; \forall k = 1, ..., K$, $\sum_{k=1}^{K} \pi_k = 1$ and $N(x|\mu_k, \sigma_k)$ describes the multivariate Gaussian with:

$$N(x_i|\mu_k, \sigma_k) = \frac{1}{(2\pi)^{\frac{n}{2}}|\sigma_k|^{\frac{1}{2}}} \exp(-\frac{1}{2}(x_i - \mu_k)^T \sigma_k^{-1}(x_i - \mu_k))$$

where n is the number of dimensions of the data and $|\sigma_k|$ is the determinant of the covariance matrix of *k-th* Gaussian. As we mentioned earlier, each distribution parameterized by a mean μ_k that is a $n \times 1$ vector and a $n \times n$ covariance σ_k matrix.

Finding the model parameters that maximize the probability of the GMM, given the training data, is the goal of maximum likelihood estimation. Unfortunately,

direct maximization is not feasible because we are dealing with multiple Gaussians rather than just one. The expectation-maximization (EM) approach, however, has a specific instance that can be used to iteratively acquire the maximum likelihood parameter estimates [33]. More specifically, in the GMM algorithm, each cluster is represented by a Gaussian distribution, and the data points are assumed to be generated from a mixture of these distributions. The probability that a data point x_n belongs to a cluster with parameters μ_k and σ_k is given by:

$$p(k|x_n) = \frac{\textit{Probability that } x_n \textit{ belongs to class } k}{\textit{Probability of } x_n \textit{ belongs to all classes.}} = \frac{\pi_k N(x_n|\mu_k, \sigma_k)}{\sum_{j=1}^{K} \pi_j N(x_n|\mu_j, \sigma_j)}$$

where π_k is the mixing coefficient for the *k-th* cluster, which represents the proportion of data points that belong to that cluster, and K is the total number of clusters. In essence, this probability (or the responsibilities) indicate which Gaussian each data point is most likely to originate from. For example, if we have three Gaussians and a data point x_1 and the calculated responsibilities are $\{0.1, 0.2, 0.7\}$, then we have 70% chance the data point x_1 belongs to the third Gaussian, 10% belongs to the first and 20% belongs to the second Gaussian.

To estimate the parameters of the clusters (the mean μ_k, covariance σ_k and responsibilities π_k), the GMM algorithm uses the Expectation-Maximization (EM) algorithm [9], which is an iterative process that alternates between two steps. The expectation step where the algorithm calculates the posterior probability $p(k|x_n)$ for each data point x_n based on the current estimates of the cluster parameters and the maximization step where we update the estimates of the cluster parameters based on the probabilities calculated in the expectation step. The calculation of the new parameters derives from the following formulas:

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} p(k|x_n) x_n$$

$$\sigma_k = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} (x_n - \mu_k)(x_n - \mu_k)^T$$

$$\pi_k = \frac{N_k}{N}, \text{where}$$

$$N_k = \sum_{n=1}^{N} p(k|x_n)$$

is the total responsibility of the k-th component.

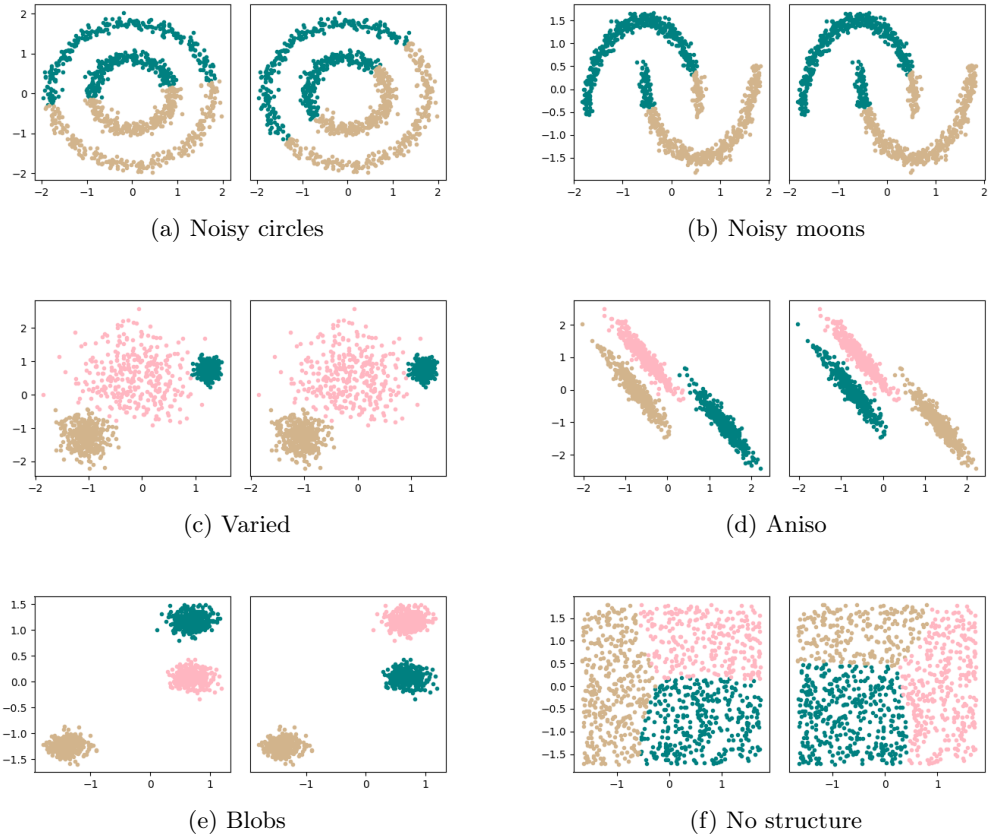
Iteratively repeat the Expectation and Maximization step until the log-likelihood function converges. It is also possible to specify a maximum number of iterations for the GMM cluster to run, and consider the model to have converged when the maximum number of iterations has been reached.

In the context of federated learning, GMM can be used for clustering into the Flower. To perform GMM clustering in a federated learning setting, each client trains a local GMM model using its own data, and then shares the parameters (μ_k, σ_k, π_k) of the GMM in a central server. The server aggregates the parameters from all the clients and updates the global GMM model. The server, then, sends the new global model back to clients and this process is repeated until the maximum number of federated rounds is reached. This is a basic implementation that can be, very easily, integrated into the Flower framework. For this purpose, we used the already implemented GMM model from scikit-learn [30] and we integrate this model into Flower. Flower provides the abstract structure of a server and client in order to integrate a new model into this framework. With this structure we created a server that asks from a random client to receive the first global parameters with a specific number of Gaussians distributions (i.e `n_components` = 2). The random client sends to the server the parameters by fitting the GMM model on the data that is at his disposal and then the server sends these parameters to all clients. When all clients return the new parameters, the server aggregates them with the FedAvg strategy that is already implemented in Flower. Every time that a client receives the updated parameters from the server, evaluates these parameters with their available data.

5.1.1 Experiments

In order to measure the performance of created federated GMM model we use several synthetic datasets from scikit-learn: make_circles, half-circles (make_moons), make_blobs (original and with the variances), anisotropically distributed data and a random dataset without structure (all of them described in Chapter: 2). Explained datasets contain samples in two dimensional space, since we compare the results of unfederated GMM from scikit-learn [30] and federated implementation via visualization of detected clusters and silhouette score.

Initially, data is partitioned into N portions, where N is the number of clients. Each client normalizes its own data and trains the GMM model with the parameter *n_ components* equal to 2. After the training step each client predicts the outcome class (0 or 1) of each sample. In order to properly compare federated and unfederated approaches we train both methods on identical train data portions. Such approaches promise equality in comparison, avoiding any randomness in results. To evaluate the performance of this algorithm, to determine if it is working well we use visualization and evaluation metrics like silhouette score. The simplest way to evaluate the performance of a clustering algorithm is to visualize the data and the clusters produced by the algorithm. This can help to identify any obvious patterns or structure in the data and to assess whether the clusters produced by the algorithm reflect these patterns. Next, there are a number of internal evaluation metrics that can be used to assess the quality of the clusters produced by a clustering algorithm. These metrics measure the compactness and separation of the clusters, and include measures such as the silhouette score and the Davies-Bouldin index. In our case, we



(a) Noisy circles

(b) Noisy moons

(c) Varied

(d) Aniso

(e) Blobs

(f) No structure

Figure 5.1: Non-federated GMM on the left sub-figures and federated GMM clustering on the right sub-figures.

used the silhouette score [30]. Silhouette score, also known as silhouette coefficient, returns a value between -1 and 1. The value of 1, or close to 1, means clusters are obviously distinct from one another and separated significantly apart. The opposite value, -1, means that clusters are assigned incorrectly and the values close to zero means that the separation between the clusters is not significant.

5.1.2 Results

Based on the experiments described in the previous section we present the silhouette coefficient for each of the 6 clustering datasets (Table: 5.1). As we mentioned earlier, the silhouette score ranges from -1 to 1, where a score of 1 indicates that the sample is very well matched to its assigned cluster, and a score of -1 indicates that the sample is poorly matched to its assigned cluster. The table compares the two different approaches, the federated and unfederated clustering with GMM. According to silhouette coefficients both methods provide very similar

Datasets	Fed. GMM	GMM
Noisy circles	0.352	0.350
Noisy moons	0.496	0.497
Varied	0.603	0.603
Aniso	0.462	0.463
Blobs	0.865	0.865
No structure	0.321	0.374

Table 5.1: Silhouette score of federated and non-federated GMM on each dataset.

results across all datasets. The differences between federated and unfederated implementations vary in range of $\pm 1\%$ with the exception of no-structure dataset where methods provide different results since the data do not have any particular clusters. Furthermore, we can identify the similarity and differences between the two approaches from the clustering results that presented in Figure: 5.1.

In general the values of the silhouette coefficient are low on the majority of the datasets. These results refer to the weakness of the GMM algorithm to identify the structure of the data on both models. Overall, the results suggest that the federated approach has almost identical ability of learning generic patterns as an non-federated clustering model.

5.2 Federated DBSCAN based on GMM data description

Due to the weakness of the GMM algorithm to identify the structure of the aforementioned datasets, we decided to use a different algorithm, DBSCAN, that can successfully cluster complex data. DBSCAN is a density-based clustering algorithm that groups together points in a dataset that are closely packed together (points with many nearby neighbors), marking as outliers points that lie alone in low-density regions (whose nearest neighbors are too far away). The key idea of the algorithm is that for each point of a cluster the neighborhood of a given radius (*Eps*) has to contain at least a minimum number of points *MinPts* [18]. One of the most important properties of DBSCAN is that it does not require specifying the number of clusters (they are detected dynamically) and that it can handle and identify noise points.

The algorithm starts by defining a neighborhood around each point, determined by two parameters. The first one is the *Eps* that refers to the distance threshold (euclidean distance in our case) that defines the radius of the neighborhood around each point. The second one is the *MinPts*, that is the minimum number of points required to form a dense region. Points in the dataset are then classified as core points, non-core points, or noise points based on the number of points in their neighborhood. A core point is a point that has at least *MinPts* points within a

distance of *Eps*. A non-core point is a point that has fewer than *MinPts* points within a distance of *Eps*, but is within the *Eps* distance of a core point. Finally, a noise point or an outlier is a point that does not belong to any of the previous categories. At the beginning, the DBSCAN algorithm selects a random unvisited sample A from the data. Initially we search in *Eps* region the number of neighbors around this point. If the number of neighbors is at least *MinPts* within *Eps* circle, we assign this point as a core point and these neighbors are part of a single cluster. In this case we will proceed to use the same approach on all neighbors of A. If the point A is a non-core point, we will ignore the neighbors and expansion of the cluster. In cases where there are no more points to take into account for the current cluster, we restart the process with new points and create a new cluster. It is important to notice that the choice of the *Eps* and *MinPts* parameters can greatly affect the results of the DBSCAN algorithm and that in practice, these parameters are often chosen through trial and error or by using methods such as the elbow method or the silhouette method.

At a more abstract level, DBSCAN algorithm groups together all the core points that are directly or indirectly reachable from each other (i.e., all core points that are in the same connected component) and forms a cluster. Non-core points are assigned to the cluster of the nearest core point. Noise points are not assigned to any cluster (DBSCAN labeling these points as -1).

5.2.1 DBSCAN on federated setting

As we mentioned earlier, DBSCAN calculates distances between data points and based on specific parameters (*Eps*, *MinPts*) assigns these points on the same or different clusters. That means that the DBSCAN algorithm requires the sharing of data points and their distances in order to identify clusters. So, in a federated setting, it is impossible to share the coordinates of the data from each client to the server because we reveal sensitive information about the data such as individual data points or the structure of the clusters themselves.

One of the proposed methods [26], is to partition the features space, on the client's side, with a fixed granularity. With this approach, clients send to the server only the number of points within each cell of the grid, restricting the exchange of raw data and protecting privacy. Then, the server aggregates the information received from the clients about the cells and runs a standard DBSCAN based on dense or non-dense cells.

In our case, we follow a different approach that focuses on how to use DBSCAN and GMM models into FIN to share data without information leakage. The main idea of this technique is the use of both clustering models for different functions. The DBSCAN algorithm is used to cluster data on each client without the arbitrary choice of the number of clusters. Based on these clusters, we use the GMM algorithm to describe the data from each cluster. We used a fixed number of Gaussian's in each cluster to describe the data with 3 parameters: *means*, *covariances* and *weights*. As we know, the number of Gaussian's to use in order to describe a data

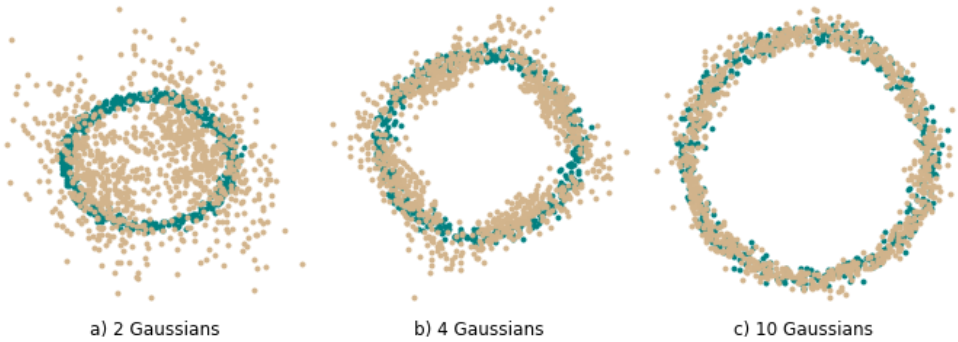


Figure 5.2: Random samples from GMM.

set depends on the complexity of the data itself and the objective of the analysis. When the number of Gaussians increases, the model will be able to capture more complex patterns in the data by dividing it into a larger number of clusters. This can be beneficial if the data has a complex underlying structure (like the datasets that will be used) that is not captured by a simpler model with fewer Gaussians. As shown on Figure: 5.2, we choose a circle cluster from the noisy circles dataset that will be used later and we tried to create random samples from the GMM model. On the first plot we fit only two Gaussians (two components) on the GMM model and the model can't identify the structure of the data and as a consequence can't produce similar data points. With four Gaussians the results are more similar to the original data (Figure: 5.2b) and finally on the third plot we can see how the synthetic data from GMM follows the same structure as the original data. We used 10 Gaussians max because increasing the number of Gaussians may lead to overfitting, where the model becomes too flexible and fits the noise in the data, resulting in revealing the original data structure.

5.2.2 Implementation

In this section, we describe with details our implementation. As mentioned earlier, for any federated machine learning algorithm to be implemented in FIN, it needs to be separated in two parts: the server, referred as `ML_SERVER`, and the client, referred as `ML_CLIENT` that are called from the `FIN_SERVER` and `FIN_CLIENT` respectively.

At the beginning server and clients exchange a set of parameters. Particularly, the server sends the parameters *Eps*, *MinPts* and *n_ components* to all clients. After all clients receive these parameters, they fit the DBSCAN model to identify the clusters on their own, local, data. At the next step each client fits the GMM model for each discovered cluster. By this procedure each detected cluster is described by *n_ components* Gaussians. As we mentioned earlier each Gaussian is described by 3 arrays: *means*, *covariances* and *weights* of each mixture component. Afterwards, all clients share Gaussian attributes in combination with the number of samples per DBSCAN cluster to the server. Since the server receives the GMM attributes from

	Client 1	Client 2
# Samples	140	100
# DBSCAN clusters	2	1
# GMM Clusters	4 * 2	4 * 1
GMM Clusters Size	(50, 20, 5, 7) (32, 18, 4, 4)	(80, 8, 7, 5)
# Server Samples	240	

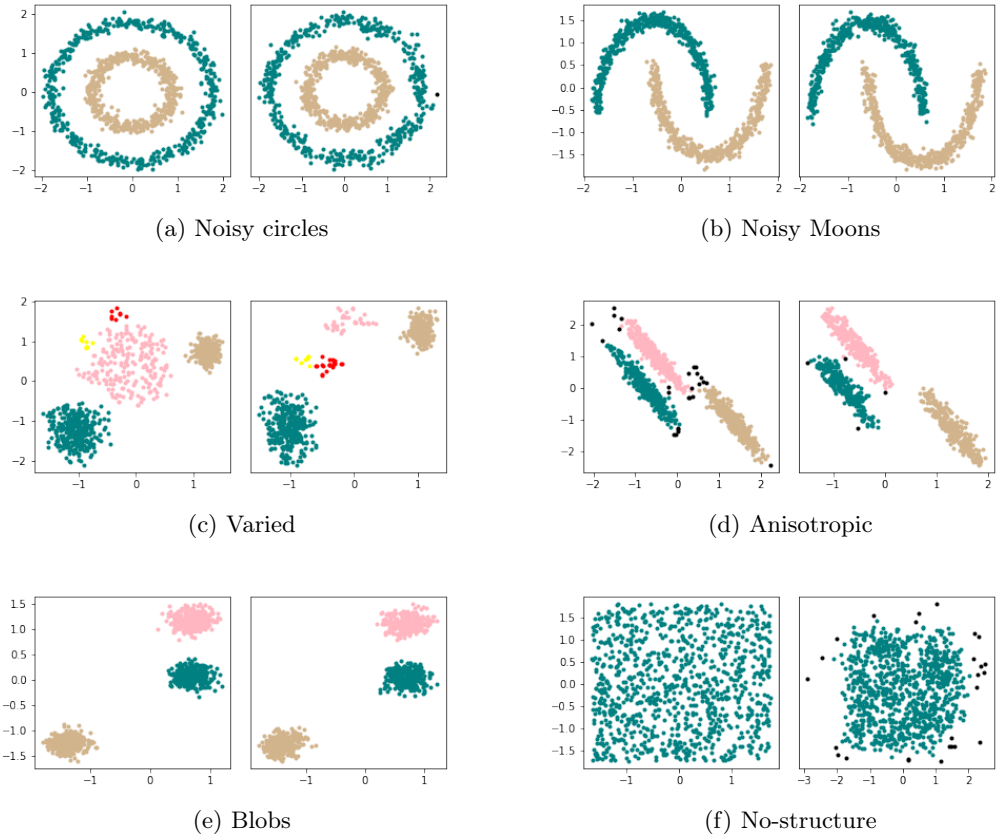
Table 5.2: An example of the new data points created on the server.

the clients, it manages to re-create the data according to GMM models. Those data are utilized in server side in order to execute the DBSCAN clustering model (with the same parameters that was initially used by clients). The result of the server DBSCAN execution forms the final model.

Is important to mention that the number of samples that was created on the server depends on the number of samples that each DBSCAN cluster has. For example (Table: 5.2), suppose that we have two clients with 140 and 100 number of samples respectively. The DBSCAN algorithm identifies two clusters in the first client and one in the second client. When we fit the GMM on a cluster's data with the parameter *n_components* equal to 4, we create 4 Gaussians per each identified cluster. Every spotted cluster has a different number of data points, so the server will re-create a specific number of samples for each cluster. That is, from the first client will create a total amount of 82 new samples that belong to 4 Gaussian's, 50 samples for the first Gaussian, 20 for the second etc. The same approach follows with the second client. FInally, on the server will be created 240 new data points.

5.2.3 Experiments

In order to evaluate the federated DBSCAN algorithm, we used the 6 different clustering datasets, with 'interesting' structures that are described in Chapter: 2. Selected datasets provide a variety of complex structures that allow the benchmarking of clustering methods. As in the previous experiments we utilize the original (non-federated) implementation of clustering algorithm as a ground-truth and compare the results with the proposed federated approach. As a configuration of our model we predefined a number of federated clients equal to 2 and the number of Gaussians that are used for DBSCAN cluster description (*n_components*) equal to 10. In terms of the DBSCAN parameters (in both cases) we used identical hyper-parameters values. We believe that such configuration will allow fair comparison between federated and non-federated implementations. The simplest way to evaluate the performance of these two clustering algorithms, is to visualize the data and the clusters produced by the algorithm. This can help to identify any obvious



(a) Noisy circles

(b) Noisy Moons

(c) Varied

(d) Anisotropic

(e) Blobs

(f) No-structure

Figure 5.3: Non-federated DBSCAN on the left sub-figures and Federated DBSCAN clustering on the right sub-figures.

patterns or structure in the data.

5.2.4 Results

Based on the described experiments in the above section, we accomplish clustering of 6 different datasets (see Figure: 5.3) and compare the performance between unfederated (left sub-figure) and federated (right sub-figure) approaches. Initially, is important to mention that the synthetic data that generated by proposed procedure (GMM cluster description and sampling of a new data), generates slightly different clusters. Such small difference, also known as noise, is important since it allows to preserve the privacy of the original data points. Despite the fact of the noise, the synthetic data points preserve the same structure as the original data.

As shown in Figure: 5.3, both methods can identify the clusters on the majority of the cases. More specifically, on the moons, circles, blobs and anisotropic dataset the results are almost identical. In the case of the varied and no structure dataset,

we observe that the unfederated approach succeeds a better clustering. Also, in the case of the varied dataset (Figure: 5.3c), we observe differences between the number of samples of the pink cluster. These results may be related with the splitting of the data or the weakness of the GMM model to describe these data. Similarly, on the last dataset (Figure: 5.3f) the unfederated model has the best performance, since it succeeds in clustering the data into one cluster. On the other hand, the federated model creates one cluster but also identifies noise points. The noise points are presented as the black data points. Based on our results we can assume that in some cases the unfederated approach has slightly better results but, overall, both methods provide very similar results across all datasets.

Chapter 6

Discussion

In this thesis, we propose FIN, a novel federated learning strategy that is able to support any machine learning in a federated setting. Based on the proposed solution we manage to transfer the state-of-the-art XGBoost tree classification model into a federated setting and compare performance on multiple datasets between federated and unfederated implementations. According to achieved performance we can assume that the federated implementation of XGBoost is able to provide very similar performance with the original XGBoost implementation. Such results show the suitability of federated solutions and their ability to capture generic patterns of the data. Additionally, we manage to transfer the GMM clustering model to a federated scenario with almost identical performance with non-federated implementations. Due to the weakness of the original GMM model over complex data we moved towards more sophisticated clustering methods such as DBSCAN. In this thesis we developed a novel implementation of a federated DBSCAN clustering model with a combination of GMM in order to describe the data. The combination of these two clustering algorithms, succeeds to preserve the privacy of sensitive data and at the same time to use a state-of-the-art density based algorithm that identifies clusters on 'interesting' 2 dimensions datasets.

It is important to mention, that these methods have been tested in medical datasets, but they have not been tested in bioinformatics datasets. As we know, the significant results that XGBoost and DBSCAN models have achieved in bioinformatics datasets are remarkable. These models have the potential to provide valuable insights and improve our understanding of complex biological systems. For example, XGBoost has been used to identify RNA modifications in various species [22], determine the regulatory interactions between genes [25] and to predict protein-protein interactions [10]. In case of DBSCAN model, it has been used for clustering SNPs (Single Nucleotide Polymorphisms) [38] and T-cell receptor sequences (TCR) [28]. Therefore, as a future work, we aim to compare our XGBoost and DBSCAN/GMM implementation with the results that have been obtained when their non-federalized counterparts are used in high profile publications.

Furthermore, in order to improve our implementation we plan to address limitations of the Flower framework, such as the predefined number of rounds, by implementing dynamic iterations or early stopping when a model converges. Moreover, due to the extensive execution time that federated XGBoost requires to produce a final global model, we plan to apply more sophisticated optimizations techniques to reduce this time. Finally, the federated implementation of other machine learning models with high impact in the bioinformatics field will contribute significantly to the research community.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, Pedro PB de Gusmão, and Nicholas D Lane. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390*, 2020.

[3] Aniruddha Bhandari. Auc-roc curve in machine learning clearly explained. `https://www.analyticsvidhya.com/blog/2020/06/auc-roc-curve-machine-learning/`, June 2020.

[4] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[5] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[6] Virendra Chauhan, Shobhana Dwivedi, Pooja Karale, and SM Potdar. Speech to text converter using gaussian mixture model (gmm). *International Research Journal of Engineering and Technology (IRJET)*, 3(5):160–164, 2016.

[7] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, San Francisco California USA, August 2016. ACM.

[8] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[9] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[10] Aijun Deng, Huan Zhang, Wenyan Wang, Jun Zhang, Dingdong Fan, Peng Chen, and Bing Wang. Developing computational model to predict protein-protein interaction sites based on the xgboost algorithm. *International journal of molecular sciences*, 21(7):2274, 2020.

[11] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[12] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[13] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[14] Chaoyang He, Songze Li, Jinhyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, et al. Fedml: A research library and benchmark for federated machine learning. *arXiv preprint arXiv:2007.13518*, 2020.

[15] Wei Huang, Tianrui Li, Dexian Wang, Shengdong Du, and Junbo Zhang. Fairness and accuracy in federated learning. *arXiv preprint arXiv:2012.10069*, 2020.

[16] Kaggle. Body signal of smoking. `https://www.kaggle.com/datasets/kukuroo3/body-signal-of-smoking`, 2022.

[17] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021.

[18] Kamran Khan, Saif Ur Rehman, Kamran Aziz, Simon Fong, and Sababady Sarasvady. Dbscan: Past, present and future. In *The fifth international conference on the applications of digital information and web technologies (ICADIWT 2014)*, pages 232–238. IEEE, 2014.

[19] Junghye Lee, Jimeng Sun, Fei Wang, Shuang Wang, Chi-Hyuck Jun, Xiaoqian Jiang, et al. Privacy-preserving patient similarity learning in a federated environment: development and analysis. *JMIR medical informatics*, 6(2):e7744, 2018.

[20] Li Li, Yuxi Fan, Mike Tse, and Kuo-Yi Lin. A review of applications in federated learning. *Computers & Industrial Engineering*, 149:106854, 2020.

[21] M. Lichman. UCI machine learning repository, 2013.

[22] Kewei Liu and Wei Chen. imrm: a platform for simultaneously identifying multiple kinds of rna modifications. *Bioinformatics*, 36(11):3336–3342, 2020.

[23] Xiaoyuan Liu, Tianneng Shi, Chulin Xie, Qinbin Li, Kangping Hu, Haoyu Kim, Xiaojun Xu, Bo Li, and Dawn Song. Unifed: A benchmark for federated learning frameworks. *arXiv preprint arXiv:2207.10308*, 2022.

[24] Yang Liu, Tao Fan, Tianjian Chen, Qian Xu, and Qiang Yang. Fate: An industrial grade platform for collaborative learning with data protection. *J. Mach. Learn. Res.*, 22(226):1–6, 2021.

[25] Baoshan Ma, Mingkun Fang, and Xiangtian Jiao. Inference of gene regulatory networks based on nonlinear ordinary differential equations. *Bioinformatics*, 36(19):4885–4893, 2020.

[26] Gabrielle Marino. Federated dbscan, 2021. MSc degree.

[27] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[28] Pieter Meysman, Nicolas De Neuter, Sofie Gielis, Danh Bui Thi, Benson Ogunjimi, and Kris Laukens. On the viability of unsupervised t-cell receptor sequence clustering for epitope preference. *Bioinformatics*, 35(9):1461–1468, 2019.

[29] Yuya Jeremy Ong, Nathalie Baracaldo, and Yi Zhou. Tree-based models for federated learning systems. In *Federated Learning*, pages 27–52. Springer, 2022.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[31] R. Polikar. Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*, 6(3):21–45, 2006.

[32] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and H Brendan McMahan. Adaptive federated optimization. *arXiv preprint arXiv:2003.00295*, 2020.

[33] Douglas A Reynolds. Gaussian mixture models. *Encyclopedia of biometrics*, 741(659-663), 2009.

[34] Nicola Rieke, Jonny Hancox, Wenqi Li, Fausto Milletari, Holger R Roth, Shadi Albarqouni, Spyridon Bakas, Mathieu N Galtier, Bennett A Landman, Klaus Maier-Hein, et al. The future of digital health with federated learning. *NPJ digital medicine*, 3(1):1–7, 2020.

[35] Anit Kumar Sahu, Tian Li, Maziar Sanjabi, Manzil Zaheer, Ameet Talwalkar, and Virginia Smith. On the convergence of federated optimization in heterogeneous networks. *arXiv preprint arXiv:1812.06127*, 3:3, 2018.

[36] Robert E Schapire. The boosting approach to machine learning: An overview. *Nonlinear estimation and classification*, pages 149–171, 2003.

[37] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. Split learning for health: Distributed deep learning without sharing raw patient data. *arXiv preprint arXiv:1812.00564*, 2018.

[38] Shichen Wang, Debbie Wong, Kerrie Forrest, Alexandra Allen, Shiaoman Chao, Bevan E Huang, Marco Maccaferri, Silvio Salvi, Sara G Milner, Luigi Cattivelli, et al. Characterization of polyploid wheat genomic diversity using a high-density 90 000 single nucleotide polymorphism array. *Plant biotechnology journal*, 12(6):787–796, 2014.

[39] Mengwei Yang, Linqi Song, Jie Xu, Congduan Li, and Guozhen Tan. The Tradeoff Between Privacy and Accuracy in Anomaly Detection Using Federated XGBoost. *arXiv:1907.07157 [cs, stat]*, October 2019. arXiv: 1907.07157.