University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Anomaly Detection in IoT networks based on Federated Learning

Mustafa O. Cay

March,17 2025

Abstract

Federated Learning (FL) is a promising alternative to traditional centralised learning, particularly in scenarios where data privacy is paramount. By allowing data to remain on client devices rather than on a central server, FL increases user control and enhances privacy. In this study, we investigate the viability of FL within IoT networks for anomaly detection by comparing its performance to that of traditional centralised learning methods. Using Random Forest (RF) models on client devices, our experiments demonstrate that FL can achieve prediction performance comparable to that of centralised models. Although explicit privacy enhancements and communication overhead reductions were not quantitatively evaluated, these results lay the groundwork for future work aimed at developing targeted privacy improvements and communication optimization strategies.

Acknowledgements

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Mustafa Onur Cay Date: 17 March 2025

Contents

1 Introduction

1.1 Extensive use of IoT devices

With Internet of Things (IoT) devices now being practically in everywhere in the forms of smart speakers, smart light bulbs, smart switches, smart appliances and many other devices including industrial applications, it draws more attention to the security and privacy of these devices. We interact with them almost daily, whether it be turning on the lights, heating up your home, or doing laundry. Some estimates put the number of IoT devices at 30 billion devices in 2022, with expectations that it will double in the next 4 years Dhull et al. (2022). Because of these reasons, we need to be prepared for an all-connected eventuality. We need to be able to protect our IoT networks, in turn protecting our own privacy. Many of these devices gather a lot of data, from temperature, which is relatively public information, to more concerning data like movement and noise Ari et al. (2024).

1.2 Anomaly detection

Anomaly detection refers to the process of identifying patterns or observations that do not conform to the expected behaviour. According to Chandola et al. (2009), anomalies are data points that deviate significantly from the norm, making them useful indicators for critical incidents in various applications such as fraud detection, system monitoring, and security.

1.3 What is Federated Learning?

Federated Learning (FL) which was first coined by Google is a machine learning technique. The main idea of FL is to maintain separate data sets on separate devices while preventing data leakage. This prevents one single party from having unlimited access to the data, which can violate people's privacy Yang et al. (2019). In its core, FL works by utilizing different client data sets and training a central model that is shared among clients. There are different versions and methods of FL. There is vertical FL, which is where users' space is the same, but they have different features Liu et al. (2024). An example would be a bank that has information like current balance, monthly money in, and monthly money out. On the other hand, a trading platform would have data on their trading patterns, such as what they trade and how much they trade. Using FL these organisations, while keeping their users' data private, can develop a joint model that predicts how likely a user is to make profit trading. The other method is horizontal FL, which is the focus of this paper. Horizontal is when the users are different, but they have the same data Yang et al. (2019). You can imagine 2 banks in this scenario. Again, using FL and keeping their data private they can essentially train with more data that will allow them to develop models that will generalise better. There is also the concept of centralised and decentralised. Centralised can be classified as the traditional approach where there is a central server that handles tasks other than training. Such as model aggregation and client selection. This server is also known as the orchestration server. Decentralised approach is, as the name suggests, there is no central server. The clients communicate amongst themselves to handle these tasks. This helps remove the single point of failure and a potential bottleneck from the system Kairouz et al. (2019).

1.4 Motivation

In September 2016 a large Distributed Denial of Service (DDoS) attack, consisting of 620Gbps of traffic, took down the website of Brian Krebs, a security consultant company. The magnitude of the attack was much larger than what was required to bring down a website, as pointed out by Kolias et al. (2017). Carried out by an attacker that utilised something called botnet, which can be classified a cyberweapon. Inside the network there are thousands if not millions of devices that have internet connections. They can be made to work together by a threat actor and overwhelm targets' sites. Before the prevalence of IoT, these networks consisted mainly personal computers or servers. However, with the increasing number of IoT devices, attackers are now utilizing IoT devices as part of their bot net. In addition, a particularly motivated attacker can also infect targeted IoT devices, for use other than DDoS related. Since these devices have cameras and sensors, they can be exploited as spyware for attackers as well.

The recent improvements in chip design allow IoT devices to be more powerful than ever. Valente et al. (2022). This allows for more processing power on the device; with this power, we can run small but powerful RF models on the device. Also considering, tree-based models are a lot more hardware friendly and require fewer resources while maintaining high accuracy Daghero et al. (2021). Looking at the amount of information these devices store on us, it is preferable for privacy that this data stays on this device. FL is aimed at making this a reality with privacy protecting measures like Differential Privacy, where noise is added to obscure contributions from clients Dwork et al. (2006). FL also has applications in easing the computational cost required by the server.

So in summary, while our simulation focuses on demonstrating the effectiveness of FL for intrusion detection, it is important to note that FL inherently supports privacy by keeping data local, which contrasts with traditional centralized approaches where security concerns often arise due to data aggregation.

1.5 Aim

The aim of this paper is to demonstrate the feasibility of FL in the context of IoT anomaly detection using Random Forest, which is a tree-based model. We hope to achieve similar if not better accuracy results compared to centralised training. In order to achieve this, we will be following this structure;

- Choose a dataset that is state of the art and is representative of the problem we are trying to recreate and simulate.
- Carefully process and partition the data to suit our needs
- Design, implement and train an FL approach for anomaly detection
- Evaluate our model using standard metrics and experiment with different strategies and parameters.

Table 1.1: *Acronyms used in this paper*

Acronym	Description
FL	Federated Learning
IoT	Internet of Things
RF	Random Forest
DDoS	Distributed Denial of Service
Gbps	Gigabits per second
SOC	Security Operations Centre
ML	Machine Learning
DT	Decision Tree

1.6 Summary

We have looked at potential aims and motivations behind the Federated learning technology. In the next chapter we will be looking at how specific parts of this technology work and its real-world uses and the remainder of the paper will cover how we have designed and evaluated this technologies' viability for our use case. We will be structuring this paper as such:

- **Chapter 1** - Introduction, aims and motivations
- **Chapter 2** - Real-world use of this technology and how specific components of it works helping us gain an insight in to the requirements of our experiments
- **Chapter 3** - Requirements, we will discuss the core components of our experiments and what we expect from it.
- **Chapter 4** - Design, we will have a look at how we designed our system to achieve what we need, we will cover how we realised our requirements
- **Chapter 5** - Implementation, specific technologies, logic and algorithms we used to bring our design to life.
- **Chapter 6** - Evaluation, we will evaluate and compare our results to a centralised system, as well as compare different merge methods we have implemented.
- **Chapter 7** - Conclusion, we will draw our conclusion on this technology weather if its viable or not and propose future improvements of this technology.

2 Background

In this section, we will be looking at different aspects of this project. Namely, I will examine the following

- Anomaly detection in IoT networks
- Federated Learning
- Machine Learning and Random Forests

2.1 Anomaly Detection

IoT devices have changed the way we interact with our surroundings. The way we collect data, perform autonomous tasks and the way devices talk to each other. As the IoT space grows, its attack surface grows. Giving threat actors new ways to exploit devices and networks. This is why anomaly detection is a crucial part of any network, not just IoT devices. Anomaly detection can help spot malicious activity on a network. Specifically, by monitoring packets over the network.

2.1.1 Traditional anomaly detection

Traditionally, anomaly detection has been used in many settings. From cybersecurity to industrial applications. Essentially anywhere that needs monitoring for unexpected events like security breaches or drop in equipment performance in factories. According to Chandola et al. (2009) there are couple of different ways anomaly detection is carried out;

- **Classification based**, where we learn a model (classifier) from a set of labelled data (training) then testing
- **Nearest Neighbour based**, where we define a distance measured between 2 data instances. We then spot anomalies using the assumption that normal data is dense in terms of our distance definition and abnormal instances are more isolated from neighbours.
- **Clustering based**, where we cluster the data points in to groups and point that don't belong to a cluster are classified as anomalies. The difference from Nearest neighbour is Cluster based models don't require labels, unlike neighbour models needing labels for the data to train.
- **Statistical**, where we follow the assumption that normal data follows a known probabilistic model or distribution. For example, in a large text we can assume that the English language has a statistical model, i.e. its vowels are predictable. Using this assumption, we could spot sentences in German because German vowels won't look like English vowels statistically.
- **Information Theoretic**, information theory states the more unpredictable data is more information it contains. For example, there is 90% chance that an event occurs. This means that the outcome of the event contains less information because it was predictable. So this method analyses the information content of the data using different measures like *Kolomogorov Complexity* and others. The assumption is that data irregularities lead to irregularities in information content of the dataset.

- **Spectral**, this is a very complex method, but to simplify it. If we reduce the dimensionality of the data the anomalies will be easier to spot. The assumption is that normal data will be easier to distinguish from abnormal data in reduced dimensionality.

While ensemble models, like RF, aren't stated explicitly, decision trees are mentioned under rule based classification techniques, where the training step essentially determines a set of rules for determining classes. Ensemble type models like Random Forest have been used in Security Operations Centre to monitor real time data coming in from their clients and would make decisions on whether the traffic is normal or if there is an intruder active in the network.

2.1.2 Challenges of anomaly detection in IoT networks

I will be exploring the challenges in the context of cybersecurity, but many do apply to other fields. There are many challenges with anomaly detection in the context of IoT networks, whether it be centralised versions or distributed versions. So usually data is streamed in to SOC's and this poses the challenge of real time processing from those incoming endpoints. Because SOC's can and usually do operate in time-critical environments like banking or infrastructure. Anomaly detection also suffers from imbalanced distributions, i.e. there are not that many credit card frauds happening compared to normal transactions. This leads to size and quality of the data that is used for training anomaly detection algorithms. There needs to be high quality, preferable purpose made data for training these algorithms.

2.2 Federated Learning

When Google coined the term Federated Learning in 2016, people have been splitting up their data and computation across multiple devices for a while. However, FL came at a very opportune time just 2 years before the Cambridge Analytica scandal. In its core, federated learning is using distributed data sources to train a global model. Yang et al. (2019) defines N data owners $F_1 ... F_n$ which hope to train an ML model using their respective data sets. Conventionally, all of this data would be converged in to one large data set to then train a model, M. But in a Federated Learning context, data owners collaboratively train a model, M_{FED}. This way they protect their data privacy.

In Figure 2.1 IoT clients are the data owners, and they train a model which they send to the server at intervals which sends back a global model for them to keep training on.

2.2.1 Communication Costs

Google outlines how federated learning can be useful in privacy and proposes techniques to reduce communication overhead Konecný et al. (2016). They acknowledge that communication overhead is a problem, especially with the internet speeds in the US at the time. Stating that it is important to investigate methods aimed at reducing uplink communication cost. They provide 2 different approaches to this issue; "Structured updates, where we learn from a restricted lower-dimensional space" and "Sketched updates, where we learn a full model update, but then compress it before sending it to the server". Both of these methods are aimed at making the update coming back from the client as small and compact as possible.

2.2.2 Non-IID Data and system heterogeneity

By nature Federated learning happens in a distributed manner. This means that data is stored in different client devices, so unlike centralized learning where data is assumed to be independent and identically distributed(IID) it is not possible to make this assumption in FL. Data in FL settings is often non-IID this means that data can vary significantly in it's distribution, quality and

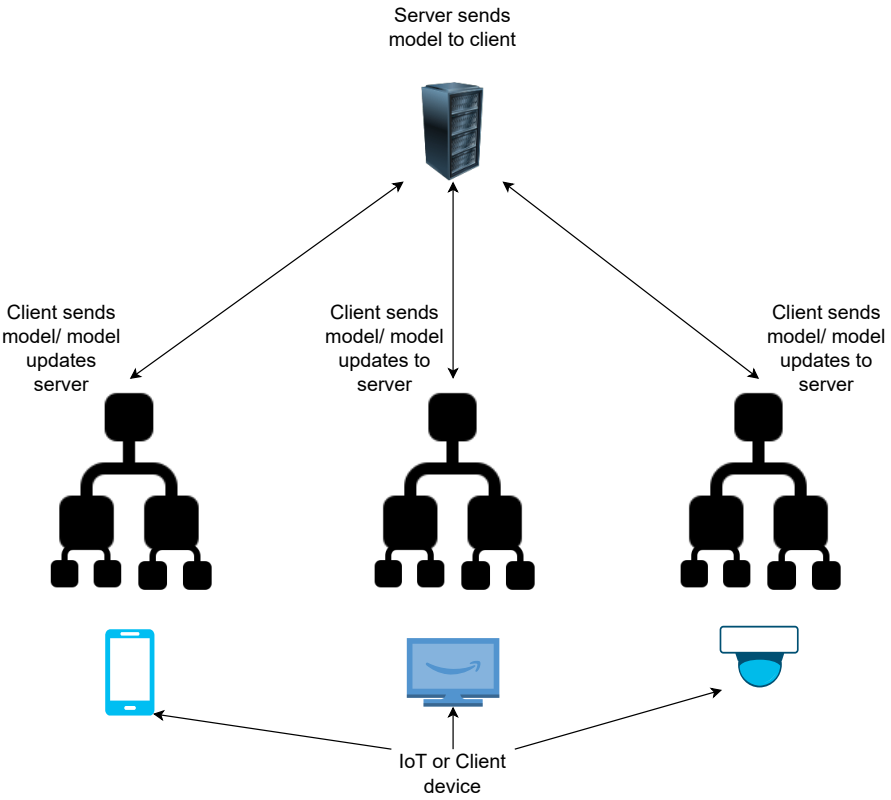

Figure 2.1: *Simple diagram explaining basic FL flow*

quantity. There are couple different ways of classifying Non-IID'ness as explored mathematically by Kairouz et al. (2019) which can be summarised as.

- **Feature distribution skew**, different clients can have data that is of the same nature but slightly different. A good example mentioned in the paper is that handwriting is different from people to people in terms of stroke width and slant, so 2 people can write the word "hello" in different ways.
- **Label distribution skew**, can be summarised as the amount of a specific label existing, can change from client to client. An example is that people from Scotland might use the Scottish flag emoji more often compared to a person from Japan. The example given in the paper is that you might see more pictures of wild kangaroos on the device of a client that live in Australia.
- **Same label, different features**, as the name suggest same label i.e. what it is doesn't change. but how it looks different due to differences in clients. Like time of day, weather events, cultural differences etc. For example, Greek houses are different, often painted white on the outside. While houses in England tend to be built with red bricks.
- **Same features, different labels**, this time we have similar or same data, but our labels are different. A simple example could be that 2 doctors can look at the same symptoms and give different diagnosis.
- **Quantity skew**, somewhat obviously clients can have different amounts of data.

The author adds that in the real-world, data is likely a mixture of these effects. And this becomes a problem when trying to create a model that is robust and generalizable. If these factors are not considered, the model will struggle to generalize and will be affected by clients' data more or less than its intended amount.

Systems heterogeneity is a problem that rises in FL and that is explored by Li et al. (2020). The author correctly states the differences in client devices such as available storage, compute power, etc. They also mention that in a typical FL network there can be millions of clients and only a limited number of clients participate at any given time. They consider the possibility of clients dropping out of the session during training due to third party issues like network or battery. So developed FL methods must be resistant client drops, differences in hardware and be robust enough to handle low client participation as well as the other issues mentioned with data heterogeneity.

2.2.3 Merging of Models

As explained previously, models that arrive from the graph have to be merged somehow. There are different ways to achieve this and it depends on the type of model that is being used. Arguably the most common model used in FL are Neural Networks. In Neural Networks, the model is updated iteratively through epochs; an epoch represents a pass over the data. After each epoch the model is refined using optimization techniques, like Stochastic Gradient Descent (SGD). In FL at this point we send our model's parameters to the server, parameters are essentially the values that make the model *our model* as opposed to any model. These parameters are aggregated in the server. McMahan et al. (2016) in this paper, FedAvg is proposed by the author. The purpose of FedAvg is to aggregate local updates by weighting them according to the size of the client data. The formula can be given as follows.

$$w_{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k \quad \text{where} \quad n = \sum_{k=1}^{K} n_k$$

Where K is the total number of clients we have in our training regiment, n_k is the number of data points at client k. n can be defined as the total number of data points across all clients. Client's local model is defined as w_{t+1}^k. Each client k starts with global model w_t they perform their respective training locally with local data, which can be multiple epochs. Which then becomes w_{t+1}^k and is then sent to the server to generate the new global model. This ensures balance in the global model and allows clients with more data points to influence the model more. However, this only works with model parameters. In models with no parameters but instead splits, nodes and leaves that represent the decision-making of the tree it's not possible to use FedAvg. While there are some alternatives for tree based models the research on ensemble type models like RF is relatively sparse.

2.2.4 Privacy Concerns

Privacy is an essential part of the FL process. The reason FL exists is to increase the privacy of the data owners. In the canonical paper from McMahan et al. (2016) the privacy issue in FL is discussed, stating that model inversion attacks exist in FL processes just like traditional networks. Briefly, this type of attack is when the threat actor is using inference to learn the training data from the model. They also consider a possible attack to local updates while they are in transit to the server. Their suggestion is encryption to prevent this, along with differential privacy. According to Yang et al. (2019) the methods of differential privacy involve adding noise to the data. This helps obscure sensitive attributes until a third party is unable to distinguish them. Some other considerations have been made in regard to privacy as well;

- **Secure aggregation**, this uses advanced cryptographic techniques to aggregate models. Shortly summarised, it is used to encrypt and mask local updates on the client side in such a way that any client and decrypt and unmask the data for use. Importantly, as outlined in Bonawitz et al. (2016) the server will not be able to distinguish any client update and will receive a masked or aggregated sum.
- **Homomorphic encryption**, similar to Secure aggregation, it's focused on the server never seeing the updates. But in this case it's more flexible in that the server gets the client model directly and can perform computations on the ciphers.

There are other types of attacks that can exist in FL. Model poisoning or model backdoor is where there is a client that is an attacker. They can look to feed the model bad data to poison it. Bagdasaryan et al. (2020) argues states, beyond just poisoning the training data FL allows bad actors to replace the model with their chosen model where they can keep the models accuracy, but they manipulate their chosen class to respond in a certain way. For example, in image classification their model still predicts dogs and cats correctly, but they alter it so that zebras are classified as horses. In the paper, they demonstrate this attack and do not give solutions; rather, they leave it as future work. In summary, while FL's aim is to prevent violations of privacy, there should be special consideration to the circumstances, and these issues must be underlined when designing a proper FL workflow.

2.2.5 GBoard Real-world use

Federated learning is currently being used in GBoard, Google' keyboard application. Yang et al. (2018) gives an insight to how Google is applying federated learning in a commercial sense. GBoard has a built-in Google search query function, so they train an FL model that will predict whether query suggestions are useful. They outline that they schedule jobs in a way that doesn't affect user experience, preferably at night while the user device is connected to power and is on an unmetered connection. Their structure has a *baseline* and a *trigger* model that will be used for inference, and a separate training cache and training process on the device. As the user is clicking suggestions this data is recorded, then at a suitable time the phone reaches out to the server. Once the server has the predefined number of clients for a given population, it sends out a training task. This contains the model, metadata and selection criteria for what's to be trained. The phone then trains the model using task defined parameters. Once training is complete, the client sends the model updates to the server, where it is anonymously updated using FedAvg McMahan et al. (2016). They also impose a minimum delay to the client before it participates in another round of training to avoid over-representing devices in training. This data is used specifically in the training of the trigger model. As we outlined previously, there is a baseline and a trigger model. The baseline model is trained offline beforehand, this model is the model that makes the predictions. The trigger model then chooses if these suggestions are relevant and proposes them to the user.

2.3 Machine Learning and Random Forests

2.3.1 What is machine learning

Machine learning can be defined as a set of algorithms and techniques to allow computers to identify patterns in data and make predictions without explicitly being programmed. Ayodele (2010) further defines machine learning as computer systems that automatically improve with experience through a process of inference, model fitting or learning from examples and implement a learning process. There are different machine learning techniques. Most common ones are supervised learning, reinforcement learning and unsupervised learning. Unsupervised learning is when the system does not know what its trying to predict but rather trying to categorize the

data without predefined labels. This method can be useful in clustering, imagine a company has data on their customers and want to understand their customer base better. They hope that their algorithm will group their customers in to relevant clusters. Reinforcement learning is where an agent, often by trial and error, optimizes their actions. The most common application of this technique is in game bots, bots learn how to play the game by trial and error over many rounds. Lastly, supervised learning, which is what we are focusing on, is when the data and the label is available. We give our model our data and what is should predict. The model then learns the patterns between the labels and the data to make accurate predictions on data it has not seen.

2.3.2 Supervised Learning

Supervised learning, can be further broken down in to 2 different parts. Regression and Classification algorithms. Regression style algorithms, like Linear Regression, are used to predict continues values. By leveraging historical data like house prices in a location with inflation, we can predict the house prices for a given location. Classification can come in 2 forms; Binary classification where the model chooses from 2 options, or multi label classification. The algorithms existing for tackling these classification problems can be used for both types of classification problems. It is also important to note that algorithms generally used for one type of problem can be adapted or used for the other type of problem.

Lazy Vs. Eager Learners Lazy learners, instead of fitting a model, memorize the training data and when they make predictions they approximate the label based on closest neighbour. Some examples are K-Nearest Neighbour and Case-Based reasoning. On the other hand, eager learners learn the data during training by building and fitting a model. They then make their predictions based on this training. Some examples are Logistic regression, Support Vector Machines, Neural Networks and Decision trees. Which is what are discussing next

2.3.3 What are decision trees

Decision trees are ML algorithms that can be used both for regression or classification problems but are often used for classification problems. As the name suggests, they have a tree structure and they work by recursively splitting the data in to subsets based on features.
Typically a decision tree consist of:

- **Root Node**: The top node, this node contains all points and represents the entire dataset
- **Internal Node**: Represent decision, splitting the data based on feature thresholds
- **Branches**: Edges that connect the nodes, representing decision outcomes
- **Leaf node**: End points that represent the labels as final predictions, resulting from the decisions made in its respective path.

You can see a nice visual representation of a Decision Tree in figure 2.2

Decision trees are built top to down. We start at the root node, then we work our way down. Since the tree is built up on decision, we need to make a decision at every node. This is called **feature selection,** we select the most important feature in the data set at that point to make a decision on. There are built in methods for feature selection as well as external methods. As part of this paper, we will be focusing on the built-in versions rather than the external methods, as they are not as relevant in the case of FL.

Gini Impurity measures the probability of incorrectly classifying(labelling) a randomly chosen sample from the dataset, if your labelling was random.
Gini impurity is given as:

$$Gini = 1 - \sum_{i=1}^{C} p_i^2$$

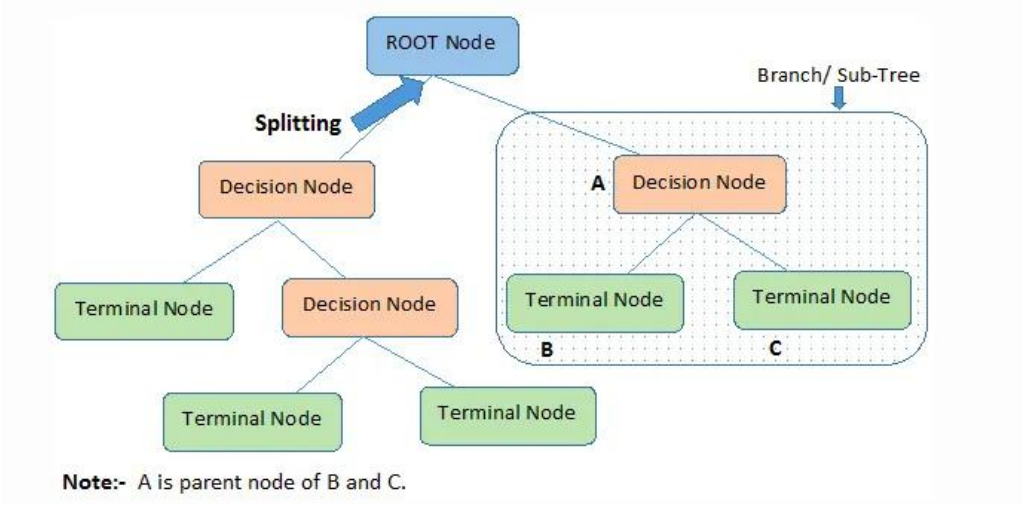

Figure 2.2: *Typical DT structure Bandara (2021)*

where C is the total number of classes we are trying to predict and p_i is the probability of class i within the node. For example, in a bag of 4 apples and 6 oranges the chances you randomly pick an apple is 40% and you pick an orange is 60% so your p_i would be 0.4 for an apple and for an orange it would be 0.6.

$$Gini = 1 - (0.4^2 + 0.6^2) = 1 - (0.16 + 0.36) = 0.48$$

This value represents how mixed (impure) the initial dataset is.

Next, the algorithm evaluates potential splits based on available features. Suppose we have two numeric features—weight (in grams) and colour (represented numerically). The algorithm considers each feature and evaluates multiple possible splits at midpoints between data points. For each candidate split, it computes the Gini impurity separately for each resulting node and then calculates a weighted average impurity across nodes. The algorithm selects the feature and corresponding threshold that results in the lowest weighted average Gini impurity, thus creating the most informative split.

Entropy measures the level of uncertainty or disorder within a dataset. Intuitively, entropy represents how mixed or uncertain a node is in terms of its class composition. Formally, entropy is defined as:

$$Entropy = -\sum_{i=1}^{C} p_i \log_2 (p_i)$$

where C is the total number of classes and p_i is the probability of class i in the node. Using the previous example of apples and oranges with respective probabilities as 0.4 and 0.6 we can calculate the entropy as:

$$Entropy = -\left(0.4\log_2(0.4) + 0.6\log_2(0.6)\right) \approx 0.97$$

Similar to Gini impurity, after calculating the initial entropy, the decision tree algorithm tests various splits based on features (e.g., weight, colour). For each potential split, the algorithm calculates the entropy for the resulting subsets and computes a weighted average entropy. The feature and split with the highest information gain (greatest reduction in entropy) are selected for splitting, effectively creating subsets that are more homogeneous and less uncertain.

There are several other parameters that can be changed to optimize a decision tree:

- **Maximum Depth (max_depth):** Defines the maximum number of splits allowed from the root node to the furthest leaf. Restricting this depth can prevent overfitting.
- **Minimum Samples per Split (min_samples_split):** The minimum number of samples required to split an internal node. A higher value can reduce tree complexity.
- **Minimum Samples per Leaf (min_samples_leaf):** Specifies the minimum number of samples required at a leaf node, helping control tree complexity and improve generalization.
- **Criterion (criterion):** Determines the function used to measure the quality of a split. Common criteria include Gini impurity and entropy, as discussed above.

It is important to mention normalisation as well. Normalisation typically involves rescaling features to a similar numeric range to prevent features with large numerical ranges from dominating the learning process. While normalisation is essential in distance-based algorithms, decision trees inherently perform splits based on threshold comparisons rather than numeric distances, which means normalisation is not required for Decision Trees.

2.3.4 Random Forests

Random forest is an ensemble learning method. Ensemble means that, it is a group of smaller learning algorithm and they can be used for classification or regression tasks that is grouped together to create an ensemble. These smaller "base models" prediction is aggregated to come up with a final prediction for the forest. In the case of RF these are decision trees Breiman (2001). RF is particularly useful because it helps to make models more generalizable. DT's are prone to overfitting the data resulting in bad predictions for unseen data. In addition DT's tend to be unstable. There are 3 main processes: bootstrapping(bagging), random feature selection and prediction aggregation. Using these processes RF aims to overcome the challenges of a DT.

Bootstrapping (Bagging): Bootstrapping refers to creating multiple independent training datasets by randomly sampling from the original dataset with replacement. Each decision tree within the Random Forest is trained on one of these bootstrapped datasets. This ensures diversity among the individual trees, reducing correlation and variance in the overall model.

Random Feature Selection: At each node within a decision tree, only a random subset of the available features is considered when determining the optimal split. This randomization further enhances the diversity among the trees, ensuring they do not converge towards similar structures, thereby reducing the overall correlation between their predictions.

Aggregation of Predictions: After training all the decision trees independently, Random Forests aggregate predictions from these trees to determine the final outcome. For classification problems, the final prediction is, usually, determined through majority voting, where the class predicted most frequently by individual trees becomes the final prediction. For regression problems, predictions are typically aggregated by averaging the outputs of all the trees, thus providing the final numeric prediction.

Hyperparameters There are 2 main parameters that can be controlled for a RF model. Bagging is considered a built-in feature rather than a parameter.

Number of Trees (n_estimators): The number of trees constructed in the forest. Increasing this value typically improves accuracy but may also increase computation time.

Max Features (max_features): The number of features randomly selected for splitting at each node. Adjusting this parameter impacts the diversity among trees.

Random Forests in Federated Learning Recently, Random Forests have also been adapted for use in federated learning contexts. In federated learning, individual decision trees can be trained locally on decentralized data, with only aggregated results shared centrally. The inherent diversity and independence of Random Forests complement federated learning's decentralized architecture, helping maintain accuracy and privacy across distributed datasets. Markovic et al. (2022) discusses the viability of RF's for IOT intrusion detection. They use a mixture of different datasets and train their algorithm for multi-label classification.

3 Analysis/Requirements

Even though in this paper we discuss the viability of Random Forests models in Federated Learning scenarios, especially for IoT networks. We will be designing and implementing an experimentation strategy. Hence we need to outline some requirements of what we aim to achieve and learn from our experiments. For this we will use MoSCoW prioritization framework. This framework help us look at the priorities of requirements. This technique helps make the requirements more digestible by splitting them in to 4 categories

- **Must have**: These items are the bare minimum must have these items to even be considered "Federated". Regarded as the minimum viable product(MVP).
- **Should have**:The items in this section are pretty much an extension of the must-haves. These are not must have but we can class these items as without these items, our experiment results won't be satisfactory.
- **Could have**:These are desirable, nice to have features but are not requirements
- **Won't have**:These features will not be present in this iteration of the program/experiment

I will encode the requirements in the format of **F** for Functional, **NF** for non-functional, Then another letter will follow denoting its importance **M** for must have, **S** for should have and **C** for could have. Then I will be numbering them amongst themselves. We will do this for easy referencing when talking about the design later on. For example, the first non-functional must have will be NFM1 and the second functional could have will be FC2. The "won't have" requirements won't be included as they will not need referencing.

3.1 Functional Requirements

3.1.1 Must have

- Must be able to train Random Forest models both centrally and in a federated manner. **FM1**
- Must be able to partition data amongst clients. **FM2**
- Must be able to produce experiment results. **FM3**
- Must be able to merge models from clients. **FM4**

3.1.2 Should have

- Should have proper non-iid partitioning to represent real world scenarios. **FS1**
- Should have competent merger(s) that will produce decent results. **FS2**
- Should be able to produce meaningful results like graphs, rather than just lots of data **FS3**

3.1.3 Could have

- Could have a somewhat representative way of measuring model sizes. **FC1**
- Could have methods aimed at reducing communication overhead. **FC2**

3.1.4 Won't Have

- Wont have privacy considerations as they can be considered a research area on their own and we wont have enough time to implement this.

3.2 Non-Functional Requirements

3.2.1 Must have

- Must have a dataset that is suitable, in terms of quality and quantity, for the task of Federated learning. **NFM1**

3.2.2 Should have

- Should have results that are easily interpretable. **NFS1**

3.2.3 Could have

- Could have a straight forward layout similar to a configs file for easy experimentation. **NFC1**

3.2.4 Won't have

- Won't have multi threading to run clients simultaneously
- Won't have resilience against clients dropping out

3.3 Summary

In this chapter, we outlined the functional and non-functional requirements of our experiment to be deemed successful. In the next chapter, we will discuss the design of this experiment with the requirements in mind

4 Design

The overall design can be separated in to 4 sections.

- **The Data** - We load the data, preprocess it and partition to clients
- **The Server** - Setting up the server, merging models etc.
- **The Clients** - Training parameters, evaluation etc.
- **Some Helpers** - Some helpers for tracking metrics and evaluating models.

Since these parts are somewhat independent of each other, we can cover them on their own. While they work together, they are not tightly coupled. For this reason, we did not include the actual simulation running in the 4 main parts of the design. As this was just initializing all the classes and running them. It is also important to note that we have used a config system for controlling the experiments. You can find an example here C.1. This helps us easily change parameters about our experiments and easily see what experiment we are running. Also, note that this was not an external config file, but a dictionary that is accessible by all parts of the system. In addition to this, we used a random state derived from a seed. This helped us reduce the unpredictability of the experiment, allowing us to compare results. This was especially crucial in the data section, as the training would be effected greatly between tests if data partitions changed.
NFC1

Since this experiment focuses on merge methods we do not make predictions, rather we evaluate the model on data it has not seen. Figure 4.1 shows how we handle the model updates. We begin by sending the train command from the server to the selected client. The client then uses their own data and the model defined by the server to train their own new model. Once the client completes training, it sends the final model to the server to be merged with other client models and the global model. This helps to make sure that the new global model is able to inherit information from the old global model. This general overview will help make the detailed design easier to understand, which is what we are going to discuss next.

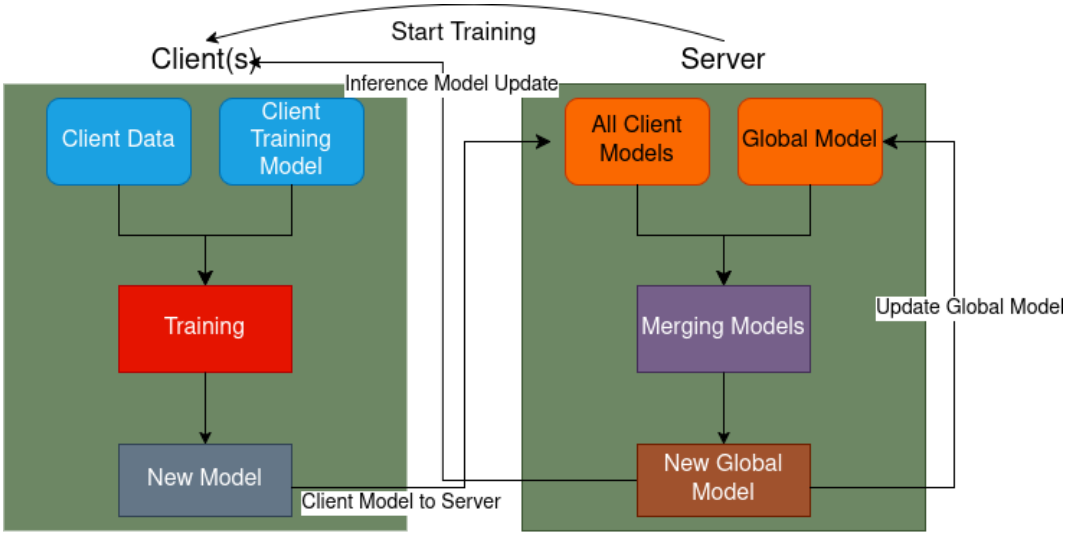

Figure 4.1: Model Update and training

4.1 The Data

The data is perhaps the most important part of a machine learning process, for these reasons we have to pay attention to the dataset we choose as well as how we partition the data amongst our clients. In figure 4.2 we show how the system works overall.

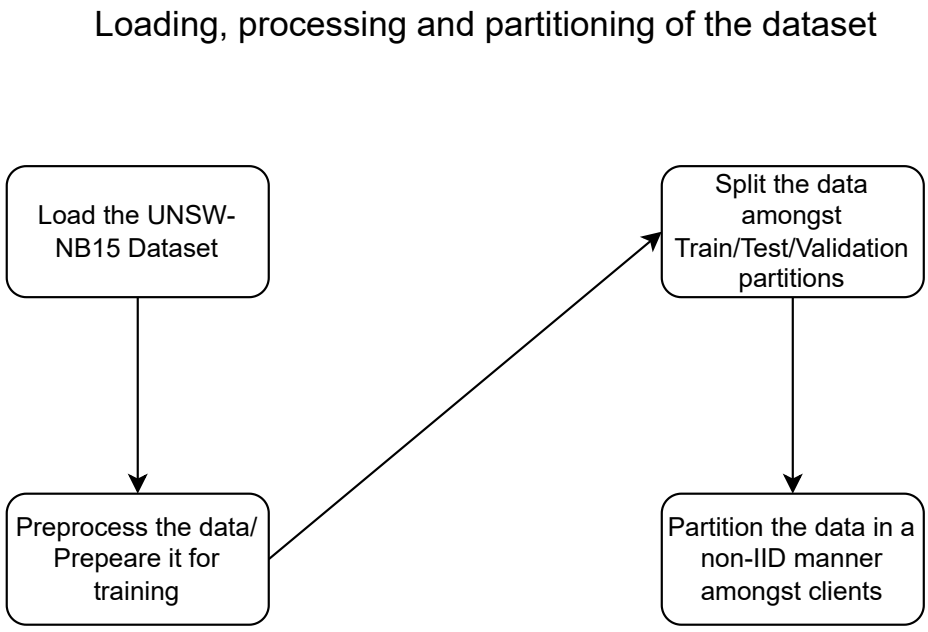

Figure 4.2: System design of data partition

4.1.1 Dataset

We decided to use the UNSW-NB15 dataset. Developed at the Australian Centre for Cyber Security using IXIA PerfectStorm to simulate network traffic. This is a comprehensive dataset that captures modern network behaviour and attacks. In its entirety, it consists of 2 million records and 49 features. However, for our machine learning purposes, we have decided to use the official training and testing dataset given in CSV format. We merge these files so that we have more control over how we distribute the data. This version has **257672** records and **45** features. It has 2 labels, one depicting whether if a record is an attack or not, the other gives insight to the type of attack. Each record essentially contains information about a packet. Its latency, protocol and service are all included as features. **NFM1**

4.1.2 Preprocessing of data for training

We have to get our data in a state so that its ready for training by clients. Since we want to predict the type of attack. We begin by dropping the *label* feature. This is to ensure that we do not leak any data to the models about the type of attack, since the *attack_cat* feature also has *normal* as a class, depicting that the record in question is not an attack. After dropping the data we encode some features. The dataset is mainly integer based, however some features are strings, like *protocol, attack_cat, service, state*. Even though we are trying to predict *attack_cat* we still encoded it because while training is fine, evaluation in our specific implementation doesn't function well.

4.1.3 Data Splitting

For our training, we have to split the data. Rather simply, we decided to go with a 70/20/10 split. We use 70 percent of the data for training, the 20 percent of it for evaluating or testing the data. The remaining 10 percent is used for validation. By convention, this data set is called validation data set. We use this data set to choose our trees, it is an essential part of our tree selection process. Again for the splitting of the data we used a random seed. This made sure that the train test and validation splits were the same over experiments

4.1.4 Partition the data

In this step, we partition the training data only. This is done because we don't need to evaluate on the client. All the evaluation is done on a central data set to ensure consistency. For the actual partitioning, we use Dirichlet distribution. This allows us to distribute the data amongst the clients in a non-IID manner. We have to define a parameter for our partitioner called alpha. This controls how skewed and non iid the data is. The lower the alpha value is the more skewed our data becomes. At the partitioning stage, we again use a random state derived from a seed. This helps us make sure that our clients have the same data for each experiment. We then have a simple dictionary that load the data to clients. Since we know the number of clients (needed for the Dirichlet partitioner) we just assign the data to the clients. Since this part is deterministic, we don't need a random seed here.**FM2 and FS1**

4.2 The Server

The server is responsible for selecting the clients that participate on a given training round, coordinating the training and merging of the models from clients. At the start of the federated training regiment, we initialize the server. The server then picks the clients for us, since this is a simulation we don't have any constraints like battery or whether the device is being used. For this reason we choose our clients randomly, however we again use a seed to ensure that our experiments are representative. Once the clients are selected, we send them training parameters

like number of trees for their forest, max depth of their trees, how many features should be sampled for bagging in RF and the number of compute cores to use for training. Once training is over for the clients we get their models and merge them. We also save the client's models to disk, this is to tell us how much bandwidth would be used, if this model was transported over the network as it is. While we acknowledge this is not an exact method we mainly use it to compare them against each other. At this stage we move on to merging the trees.**FC1**

It is also important to mention that we can train centralized by changing the configs. We have conditions through-out the code that check to see how many clients are participating, if it is one client then we do centralized training. This is true for data partition as well.**FM1**

We have 7 methods that we use for merging. Many of these methods are similar to one and other with tweaks and we refer to these methods as "mergers" from time to time in this paper. However, before the mergers we have a tree pruning process which can be turned on or off depending on the experiment. These mergers can get quite expensive computationally, so we have an optional method that can be used to prune the trees that the mergers get.

- **Tree Pruning On Server**
- **Random**
- **Best Accuracy**
- **Accuracy Weighted**
- **Data and Accuracy Weighted**
- **Impurity**
- **Diversity**
- **Top_K**

All of these mergers get the client models and the global model. This is done to ensure that we are improving the global model rather than just wiping it each round. However, in the case of *top_k* and *data weighted* the global model is handled differently rather than being evaluated like a client model. Also note that a parameter we deal with a lot during merging is the maximum number of trees allowed in the global model. We cannot allow the global model to balloon with all the trees that is received from the client, so we cap it. We refer to this as *global_num_trees*.

4.2.1 Tree Pruning On Server

This method removes structurally similar trees before merging by comparing incoming client trees against each other. If a tree is deemed too similar to an existing one — based on shared split features, thresholds, and structural characteristics — it is discarded. This prevents duplicate or redundant trees from being added, encouraging structural diversity in the global model. As a result, the pruned client models returned from this method only include unique trees that meaningfully contribute to the ensemble.

4.2.2 Random

As the name suggest, this method is randomly merging. This method adds all the trees in to a pot and chooses *global_num_trees* trees randomly. There is no seed this time. This method was designed as a sanity check and a baseline.**FM4**

4.2.3 Best Accuracy

This merger uses the validation data set to sort the trees based on accuracy. Once the trees sorted, we just choose the top *global_num_trees* from the list and make that the new global model.**FS2**

4.2.4 Accuracy Weighted

This merger can also be regarded as weighted voting approach. We begin by rating the models. This gives us a list that we can reference when we give out the spaces to our global model. We have a minimum accuracy threshold combined with the low accuracy contribution value, which is a percentage, we can dictate how many trees low accuracy models contribute to the global model. For example, if model accuracy is lower than 40 percent, that client can only contribute max 5 percent of the forest. This is useful when we have a large forest to fill and not many clients. We also dictate that the maximum a client can contribute to the global tree is 30 percent, no matter their accuracy. This helps to keep the global model balanced and also gives models other than the global model a chance. All of these parameters can be tuned. Using these principles, we assign spots in the new global model. Then for each client, we pick the best trees from their models. If we have spaces that are unallocated we use the top 30 percent of client based on accuracy to give out the rest of the trees in a round-robin fashion. Keep in mind this is done during spot allocation rather than after trees have been selected.

4.2.5 Data Weighted

This method is similar to accuracy weighted, however it does not have those special parameters. Instead, we can control the weights of accuracy and data from clients. This merger gets the total data size distributed among all clients. We then calculate the model accuracies and list them as well as the client data sizes fro clients who participated in that round. To make sure that global model is treated fairly and doesnt take over the model we assign it a "neutral" data weight. Once we know the data weight of other clients in that round we assign the global model the median of all the weights of the clients. This essentially puts the previous global model at the middle of the list. Then using the parameters *alpha* and *beta* we control how much accuracy and data proportion, respectively, affect tree allocation by assigning each client weights. Once we have the weights we assign their spots on the global model and distribute the leftover spots using the top 30 percent weight. Then for each client using their allocated tree spots we pick the best trees from them. Once tree selection is over we copy the top model and set the metadata and the selected trees for that model. Which then becomes the new global model.

4.2.6 Impurity

Each tree has a notion of impurity based on how uncertain its splits are. In this merger, we compute the impurity reduction of each tree by taking the difference between the root node's impurity and the average impurity of all its leaf nodes. This serves as a static and label-free measure of how effectively a tree partitions the data. We select the trees with the highest impurity reduction, build a fresh global model, and copy over the metadata from the first client. The number of selected trees is determined by the *global_num_trees* parameter.

4.2.7 Diversity

The diversity merger is designed to select a group of trees that not only perform well but also make different predictions from one another. The core idea is to combine accuracy and diversity into a unified selection strategy. First, each tree is evaluated using the global validation set to determine its standalone accuracy. Alongside this, we track what each tree predicts so that we can later measure how different its predictions are compared to previously selected trees.

To balance accuracy and prediction uniqueness, a greedy approach is used. At every step, the tree that scores highest on a combined metric—weighted accuracy and diversity—is selected. Diversity is measured by comparing the candidate tree's predictions to the ones already selected using the Hamming distance. Additionally, the model that the tree came from is also considered:

trees from clients with better overall performance are given more weight in the final score. This gives the merger a bias toward quality models, while still maintaining prediction diversity. As with the other mergers, the final model is constructed by selecting a base model to copy the metadata and then injecting the selected trees into that new global model.

4.2.8 Top_K

This merger is similar to the *best_acc* merger, but is designed for gradual tree growth. Instead of selecting a fixed number of trees from the entire pool each round, it adds a few of the high-performing trees (based on accuracy) to the global model incrementally. The number of trees added per round is defined by the parameter *top_k*, which can be an absolute count or a percentage of all available client trees. If the number of trees in the global model is still below the limit defined by *global_num_trees*, the selected top-k trees are added. Once the limit is reached, the merger begins replacing the lowest-performing trees in the global model with better ones from the current round. While this method does not explicitly reduce communication overhead it can train smaller forests which mean the model upload wont be as large. With some additional work, this method can reduce communication overhead significantly, however this is not covered in this paper. **FC2**

Once the merger happens we are left with a global model that is supplied by the mergers. After that we get the size of the global model by saving to the disk, the same way we got the size of the client models. We then evaluate and track the global model performance. The clients are then individually evaluated again to help with debugging and monitoring. And lastly we distribute the global model to all registered clients.**FM3 and NFS1**

4.3 The Clients

The client is straight forward. It gets some data assigned to it at the start of the federated process. When they are chosen by the server, they run their training method as defined by the server. The clients train their models independtly of the global model. Rather than buidling on the global model they build their own fresh trees. This means that there is less redundancy between the models.

4.4 Metrics Tracker and Evaluator

4.4.1 Metrics Tracker

We had a separate class for tracking metrics and evaluating models. We integrated the tracker to the main federated loop. At this level it could track the global models evolution over rounds. We tracked Accuracy, Recall, F1 Score, Precision and ROC_AUC score of the model. Using these data points from each round we were able to constructs graphs. **FS3**

4.4.2 Evaluator

The evaluator class provides a standardized way to evaluate ensemble models, especially in federated settings where class imbalance or partial class visibility can occur. It includes a patched probability method to align class predictions across trees with differing class sets. During evaluation, each tree's probability outputs are adjusted to match a global class list, and the average probability is used for final predictions. The class supports common metrics such as accuracy, precision, recall, F1-score, and ROC AUC. It handles both binary and multi-class problems, and

gracefully degrades if metrics like ROC AUC cannot be computed. This approach ensures fair evaluation of models built from heterogeneous client contributions.

4.5 Summary

We managed to meet the requirements of this project and we covered the design decisions we made in this section. Next section we will look at how we tackle the implementation of this design.

5 Implementation

We will be following a similar structure to the design section when looking at the implementation. We developed this experiment using python notebooks, specifically python 3.13.2. Initial development started on the cloud, but later was moved locally. We have 8 cells in the notebook, this includes the imports' cell and main simulation cell. The code is split up in to classes, except the aforementioned cells. The class names are as follows `MetricsTracker`, `ModelMerger`, `ModelEvaluator`, `Server` and finally `Client`.

5.1 Libraries

We use 2 main libraries, `flwr` the flower library, which is generally used for federated experiments to facilitate server and client connection, merge methods etc. We only use this library to help us partition the data. `scikit-learn`, also known as sklearn. This library provides our model, our evaluation metrics and the train test split functionality. Other libraries include `matplotlib`, `datasets`, `numpy`, `pandas` and other general things that we will cover as they are needed. We use virtual environments to manage our packets.

5.2 The Data Implementation

This function loads the CSV file containing the data, drops the 'label' column if present, and converts specified non-numeric columns ("proto", "service", "state" and "attack_cat") to numeric using label encoding. The dataset is then split globally into training, validation, and test sets. The global training set is partitioned non-iid among clients using Flower's DirichletPartitioner. You can see how we utilize this partitioner in 5.1. We then simply load the partition's to client's.

```
partitioner = DirichletPartitioner(
   num_partitions=num_clients,
   partition_by="attack_cat",
   alpha=alpha,
   min_partition_size=2,
   self_balancing=True,
   shuffle=True,
   seed=random_state,s
)
```

Listing 5.1: *Flower's DirichletPartitioner*

5.3 The Server Implementation

The server class is initialized with federated training parameters, necessary datasets and the model merger class. We also keep a local list that contains all the clients. The most important function on the server is `train_federated`. This function coordinates the training between clients, the pseudocode can be given as Pseduo-code1.

Data: C, a list of available clients;
R, number of training rounds;
`configs`, containing tree count, depth, feature count, and other parameters
Result: A trained global model and metrics tracked over rounds
begin
    $S \longleftarrow []$ `// Global model sizes`
    $L \longleftarrow []$ `// Avg. local model sizes`
    for $r \leftarrow 1$ **to** R **do**
        select a subset of clients from C
        foreach *client c in selected clients* **do**
            train local model using `configs`
            collect trained trees from c
        end
        compute and store average local model size in L
        if *more than one model* **then**
            merge client models into new global model
        else
            use single model as global model
        end
        measure and store global model size in S
        evaluate global model on validation set
        log round metrics
        foreach *client c in selected clients* **do**
            evaluate local model of c
        end
        distribute global model to all clients
    end
    compute and display average sizes from S and L
    save metrics and plot graphs
end

Algorithm 1: Federated training loop: clients train local models, merge them to form a global model, and track evaluation metrics over rounds.

5.4 Model Merger Class

We define what type of merger we want in the configs. The server implementation consults *configs* global dictionary and decides which merger to run from our `ModelMerger` class.

The most important part on the server is the merger, the focus of this paper. Our mergers are similar to each other, with slight changes. They all produce a merged single model. This is except the pruning logic, which resides in the same class as the mergers however it returns respective models instead of a merged model.

Our server side pruning logic is comparing trees. We compare on the basis of node count. If we determine the number of their node is grater than 5 we label those 2 trees as sufficiently different. If not we then check the first few nodes to determine if they are similar. The metric is that if both nodes are leaf nodes those nodes are similar, if one is a leaf node and the other is not a leaf node then we determine that they are not similar. If both nodes are not leaf we check if they are using the same feature and the threshold for that feature. If we determine that they are using the same feature and that the feature threshold difference is small those nodes are similar nodes. We can see this in code in C.2. Once we determine if the tree is not similar, we add it to a list of unique trees and also add it to their respective clients forests. This way we keep structurally unique trees. While structurally similar trees does not have to be the same predictively, as in predicting the same things, it helps with overfitting and thinning the forests. Also, nothing that structurally different trees tend to predict differently.

For merging, regardless of the method we create a new forest, and we copy over some information from the previous generation. While this is not necessary for prediction, especially if the trees are compatible, it is a nice to have to prevent any issues. We simply choose the first client, the choice of client does not matter. We copy over the number of features used in training, we copy the prediction classes, number of classes, number of outputs and feature names. Again these are not strictly required but are a nice to have especially if we need to debug the models. Because we choose clients randomly, or without consideration, they might not have all the classes we need. A tree might not be able to predict all classes because they have not seen those classes. So for this we have the `ModelEvaluator` class, which we will look at shortly.

Most of these mergers predict the trees, some of them even predict the models themselves. This causes the mergers to be computationally heavy.

5.4.1 best_acc

- puts all trees in a list and sorts them by their accuracy after predicting them. 5.2

```
all_trees = [(tree, accuracy_score(tree.predict(X_val), y_val))
          for model in client_models for tree in model.estimators_]
best_trees = sorted(all_trees, key=lambda x: x[1], reverse=True)[:num_input]
merged_model = RandomForestClassifier(n_estimators=len(best_trees),
    warm_start=True)
merged_model.estimators_ = [tree for tree, _ in best_trees]
#Copy metadata ...
```

Listing 5.2: *Selecting the best trees*

5.4.2 accuracy_weighted

- weight the model accuracies, then allocate trees in the final forest.

5.4.3 data_weighted

- weight by data and accuracy, then allocate trees in the final forest. Remember we have to inject the data proportion for the global model for this merger. 5.3

```
if any(c == "_global_" for c, _, _ in client_info):
   avg_data_prop = np.median(real_clients_info) if real_clients_info else 0.0
   for p in real_clients_info:
      print(p)
   avg_data_prop = avg_data_prop # Can add a scaler here to make the weight of
       global model data larger or smaller
   client_info = [
      (c, a, (avg_data_prop if c == "_global_" else d))
      for (c, a, d) in client_info
   ]
```

Listing 5.3: *Injecting data proportion for global model*

Both of these mergers assign a weight, then distribute slots in the global forest for trees. An example used in the `accuracy_weighted` can be seen here C.3

5.4.4 top_k

- take all trees and rank them by accuracy, determine how many trees we are going to select, then either add to the global trees or replace global model trees. We can see how we implement this in 5.4 both global_trees and selected_new_trees have been populated with the tree and the accuracy score of the tree

```
if len(global_trees) + len(selected_new_trees) <= num_input:
   final_trees = global_trees + selected_new_trees
else:
   combined = global_trees + selected_new_trees
   final_trees = sorted(combined, key=lambda x: x[1], reverse=True)[:num_input]
```

Listing 5.4: *Append or replace based on current size*

5.4.5 diversity

- this merger uses both prediction accuracy and diversity to select the best trees from across all clients. Compared to other mergers, this implementation is computationally expensive due to the nested evaluation of both models and trees. We start by computing model-level accuracy across all clients using the global validation set. Then, for each tree, we compute its individual accuracy and collect its prediction vector on the validation set. Each tree is stored along with its accuracy, its prediction vector, and the overall accuracy of the client model it came from. During the selection phase, we use a greedy loop to pick the best tree at each step. The score for each candidate tree is calculated using a weighted sum of its own accuracy and its diversity relative to the already selected trees. Diversity is computed using the Hamming distance on the prediction vectors—`preds` and `p`—which represent the predictions each tree makes over the validation dataset. Hamming returns a float representing the fraction of predictions that differ. Additionally, each tree's score is weighted based on how well its parent model performed compared to the rest of the clients. This ensures that trees from high-performing clients are more likely to be chosen, even if they are not that unique. This selection process is shown in 5.5.

5.4.6 impurity

- unlike other merging methods that rely on predictive accuracy, this merger is static and does not require any evaluation on a validation set. It instead uses a structural heuristic: *impurity reduction*. This is calculated as the difference between the impurity at the root node and the average impurity across all leaf nodes. The idea is that trees which more effectively reduce impurity (uncertainty) from the root to the leaves are likely to be more informative, even without evaluating their actual predictions.

This method is lightweight and fast because it avoids the overhead of predicting outputs or computing accuracy metrics. The tree list is first populated by iterating over all client trees and calculating their impurity reduction. The trees are then sorted in descending order of reduction, and the top *n* trees (based on `global_num_trees`) are selected. These trees are then placed into a fresh model, with the necessary metadata copied from the first client. This implementation can be seen in 5.6.

```
for _ in range(num_input):
   best_tree = None
   best_score = -1
   for tree, preds, tree_acc, model_acc in tree_pool:
      if not selected:
         diversity = 1.0
      else:
         diversity = np.mean([hamming(preds, p) for p in selected_preds])
      # Weight tree accuracy by its model's relative accuracy
      weight = model_acc / total_acc
      score = weight * (0.6 * tree_acc + 0.4 * diversity)
      if score > best_score:
         best_tree = (tree, preds)
         best_score = score
   if best_tree is None:
      break
   tree, preds = best_tree
   selected.append(tree)
   selected_preds.append(preds)
   tree_pool = [t for t in tree_pool if not np.array_equal(t[1], preds)]
```

Listing 5.5: *Greedy diversity tree selection*

5.5 Client

The implementation of the client is straightforward. The client is responsible for training the model. The client trains an independent model from the global model with the parameters dictated by the server. The implementation can be seen here 5.7. The random seed here adds further diversity to the client's models.

5.6 ModelEvaluator

The `ModelEvaluator` class is a utility designed to ensure reliable and fair evaluation of random forest models in federated learning settings, especially when class imbalance or non-identical class distributions exist across clients. A key challenge in such environments is that individual decision trees—or entire client models—may only be trained on a subset of the total class labels seen globally. This leads to inconsistencies during evaluation, particularly when computing class-dependent metrics like precision, recall, or ROC AUC. To address this, `ModelEvaluator` provides patched prediction mechanisms that align all predictions with the global class space.

The `patch_single_tree_proba` function takes in a probability matrix output from a single tree's `predict_proba` method and aligns it to the global class set. It creates a new probability matrix with the same number of rows but with columns matching the global number of classes. If a class is missing from the original tree, its column is filled with zeros. This allows predictions from different trees to be aggregated without dimension mismatch, even when some trees have never seen certain classes. C.5

The `patched_predict_proba` method builds on this by iterating over all trees in the random forest model, applying the patching logic, and averaging their aligned outputs. The result is a clean, unified prediction array that is compatible with downstream evaluation steps and consistent across client models.C.4

```
for tree in model.estimators_:
   impurity = tree.tree_.impurity
   is_leaf = tree.tree_.children_left == -1
   root_impurity = impurity[0]
   avg_leaf_impurity = np.mean(impurity[is_leaf])
   reduction = root_impurity - avg_leaf_impurity
   all_trees.append((tree, reduction))

sorted_trees = sorted(all_trees, key=lambda x: x[1], reverse=True)
selected_trees = [tree for tree, _ in sorted_trees[:num_input]]
```

Listing 5.6: *Selecting trees based on impurity reduction*

```
def train(self, num_trees, max_depth, num_max_features, n_jobs):
   print(f"Trees received from server at Client: {self.client_id} No Estimators:
       {self.model.n_estimators}")

   X_train = self.train_data.drop(columns=["attack_cat"]).values
   y_train = self.train_data["attack_cat"].values

   self.model = RandomForestClassifier(
      n_estimators=num_trees,
      max_depth=max_depth,
      max_features=num_max_features,
      n_jobs=n_jobs,
      random_state=self.client_id # To ensure diversity across clients
   )
   self.model.fit(X_train, y_train)
```

Listing 5.7: *Training logic on client*

The `evaluate_model` method then takes a model, a test dataset, and optionally the global class labels, and computes a range of evaluation metrics. These include accuracy, macro-averaged recall, precision, F1-score, and ROC AUC. Special handling is implemented for both binary and multi-class ROC AUC scoring, and errors are caught and reported gracefully if prediction issues occur. Additionally, a quiet mode allows for silent metric collection when batch logging is required without console output.

This modular structure makes `ModelEvaluator` reusable across different stages of training and evaluation, and ensures consistent model comparison regardless of class distribution at the client level. It is especially critical in scenarios where trees are merged across heterogeneous clients and predictions need to be interpreted in a shared global context.

6 Evaluation

Our data is highly imbalanced, which can be seen in table 6.1. Attack category feature of our data set contains 10 different labels, ranging from normal to Worms and shell code.

Table 6.1: *Distribution of attack_cat column*

attack_cat	Count	Percentage
Normal	92999	36.09
Generic	58871	22.85
Exploits	44525	17.28
Fuzzers	24246	9.41
DoS	16353	6.35
Reconnaissance	13987	5.43
Analysis	2677	1.04
Backdoor	2329	0.90
Shellcode	1511	0.59
Worms	174	0.07

For this reason, the same metric in different styles report different scores. Specifically, Micro vs Macro averaging. Micro averaging tells us how good the model is overall for that metric. Be it recall or precision. This is weighted by frequent classes like *Normal* and *Generic*. This can be representative of global performance, especially when class imbalance or edge cases are not that important. However, this hides poor performance in minority classes. On the other hand, macro averaging tells us how, on average, good our model is for each class. Gives all classes equal contribution and is good for evaluating performance on imbalanced datasets where classes may be under-represented. Although, macro performance can be misleading and make performance look worse than it actually is.

6.1 Metrics

So for evaluating our results we have 5 metrics.

- **Accuracy** - Our main metric. This simply tells us how accurate our models predictions are. How many of the cases we presented to it did our model managed to guess correctly. This is given as a percentage.
- **Precision –** The equation for macro precision in multi-class classification is given as:

$$\text{Precision}_{\text{macro}} = \frac{1}{C} \sum_{c=1}^{C} \frac{\text{TP}_c}{\text{TP}_c + \text{FP}_c}$$

  A false positive (FP) occurs when a sample does *not* belong to a class c, but is incorrectly predicted as class c. The use of macro precision allows all classes to be represented equally, by computing precision per class and taking the unweighted average across all classes.

- **Recall –** The equation for macro recall in multi-class classification is:

$$\text{Recall}_{\text{macro}} = \frac{1}{C}\sum_{c=1}^{C}\frac{\text{TP}_c}{\text{TP}_c + \text{FN}_c}$$

  A false negative (FN) occurs when a sample that truly belongs to class c is incorrectly predicted as a different class. Macro recall helps measure how well the model captures each class individually.
- **F1 Score (Macro) –** The macro F1 score is the unweighted average of the F1 scores computed for each class. It treats all classes equally regardless of support, making it suitable for imbalanced datasets.

$$\text{F1}_{\text{macro}} = \frac{1}{C}\sum_{c=1}^{C}\frac{2 \cdot \text{Precision}_c \cdot \text{Recall}_c}{\text{Precision}_c + \text{Recall}_c}$$

- **ROC-AUC Score –** The ROC-AUC (Receiver Operating Characteristic – Area Under the Curve) score evaluates the model's ability to distinguish between classes. The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold settings. The AUC measures the entire two-dimensional area under this curve, with a value of 1 indicating perfect separation and 0.5 representing random guessing.
  In multi-class classification, the ROC-AUC is calculated using a one-vs-rest (OvR) strategy. For each class, the model is evaluated as if it were a binary classification problem: that class vs. all others. The final multi-class ROC-AUC is the average AUC across all these OvR evaluations:

$$\text{ROC-AUC}_{\text{ovr}} = \frac{1}{C}\sum_{c=1}^{C}\text{AUC}(\text{class}_c \text{ vs rest})$$

  A higher ROC-AUC score indicates that the model is better at ranking positive instances higher than negative ones across all classes.

6.2 Baseline and Methodology

We need to draw a baseline to compare our models and merges performance against. For this we will be using a centrally trained model. We tweak our configs dictionary to switch to centralized mode, by setting number of clients to 1 and adjusting the rounds we can get an accurate centrally trained model. Basically, one client has all the data and trains all that data in one round, representing central training. Firstly, we decided that we want to use a 70/20/10 split for our experiments. This way we would have enough data to train test and validate our models. The validation dataset, while not commonly used during centralized learning, helped us tune the centrally trained model without overfitting the data, by comparing validation and test data set accuracies and making sure they don't vary much with changes. Then it was decided that we would not limit the depths of our trees, normally this could cause overfitting on single DT's however bagging helps prevent this and unlimited depth trees work better with limited data which is a bonus for federated learning. Our testing yielded that the change in criterion did not affect our results, we decided to use "gini" which is also the default method for `Sklearn`. Normally, the number of maximum features selected for bagging is either the square root of the total number of features, which is 6.6 or 7 for our dataset, or log_2 , which is again around 6. We have tested with 7, 15 and 17 max features. We found that 17 performs the best without overfitting, and any more than 17 is redundant and may cause overfitting. The reason why our model requires a larger number of features is likely down to the number of features that does not hold enough data for a proper conclusion to be drawn from. So our baseline looks as such: Table 6.2.

Table 6.2: *Baseline for centrally trained model*

Depth	Criterion	Max_Features	Data Split
None	Gini	17	70/20/10

We had to decide the number of trees we wanted to have for our forest. Though experimentation, we realised that after 30 trees we are getting diminishing returns 6.3.

Table 6.3: *Tree Count vs Accuracy*

Number of Trees	Accuracy
10	86.22%
30	86.74%
50	86.84%
100	86.86%

As per this experiment, we will be training 30 trees at the client. This will be more than enough for us.6.4

Table 6.4: *Full performance picture of our 30 tree forest*

Accuracy	Recall	ROC_AUC	Precision	F1
86.7%	63.1%	89.4%	64.2%	63.6%

6.3 Federated Training

We decided that for a representative federated learning scenario only 1%, while 10% client participation yields better results, of the clients will participate every round. Since having 100 clients and only 1 client training every round is not representative, due to there being no merging, we decided to have 1000 clients and only 10 clients participate every round. This means that every client has around 200 data instances which is the sweet spot for our dataset. We also had to make the choice of how many trees the global model will have, after merging all client models. Due to our design, we can control how many trees clients train and how many of those total trees we want to retain. This meant that we could somewhat reduce the number of trees clients have to send to the server, reducing communication costs. Yet, still maintain a compressive enough global model. So our mergers will be working to keep the prediction power of our models strong without having incredibly large models. So we decided on 50 global trees. While this number could be adjusted given the performance of the central model, we believe this number would strike a balance between having enough prediction power while maintaining low prediction times.

6.3.1 Expected performance

While, there is no research directly comparing the accuracy between centralized and federated random forest set-ups. Given the non-iid nature and the fact that we are training for multi class prediction, we can expect a drop of about 5-15% accuracy compared to the global model.

6.4 Random Merger

The results of the random merger does not denote anything but for the sake of completeness the random mergers figure can be seen in figure ??

6.5 Best Accuracy

So our `best_acc` merger is given in Figure B.2. From the graph we can observe that our model start with less than 60% accuracy. This is expected of systems with only 1% client participation. However, our model converges rather quickly at round 20~ and stabilizing at around 76.7%. Our other metrics like precision, F1 and recall seem to be low. This is due to what was discussed earlier about macro averaging. We can see the effects of uneven distribution of our `attack_cat` class. With categories like shellcode and worms brining the average down. While, ROC_AUC score is great this is again because of our data distribution and how ROC_AUC is calculated. Because our attack_cat class is dominated by the top 3 classes if the model does well in those classes it can achieve a high score in ROC_AUC without learning the minority classes

6.6 Accuracy weighted

Figure B.3 shows the graph of our Accuracy weighted merger. We can see that our model is quite oscillated. Our accuracy jitters between 68 and 77%. This is due to the weighted nature of this merger. When we choose our model and give it a weight based on that we have no way of guaranteeing that there will sufficient amount of "good" trees in that model. This can be potentially countered with smaller global model however this would mean that we are extracting less diversity from the clients. While our other metrics jitter as well, their behaviour is normal. While we have no implemented way to track individual trees over the rounds due to the way the merger works we can make the deduction that, trees tend to be replaced by new ones, unlike our previous method, which further explains the jittery nature. Our final accuracy for the global model settles at around 75.2%.

6.7 Data Weighted

Similar to our other weighted method this merger also shows oscillation over the rounds. However, the jitter seems to be less severe. In figure B.4 we can see that the jitter has reduced, especially in the earlier rounds. This is due to the stability of data *and* accuracy weighting, for this test we have used 60/40 favouring accuracy. The fact that our model is more stable over the rounds seems to show some benefit, putting our overall global model accuracy after 100 rounds at 76.5%. Our other metrics show similar behaviour to accuracy weighted, but the addition of data weights has stabilized them as well making the drops less severe. In figure B.5 we can see the jitter increase again and interestingly at round 48 we see the model take a sharp dive. When we compare this to the more balanced accuracy favoured version we can see a drop in accuracy but nowhere near as much. When investigating this further, we found that all clients at round 48 has very low accuracies. Given we are favouring data proportion and our global model is neutrally weighted it contributed less trees, leading to a very low accuracy. This likely caused a loss of trees for the model as most of the trees up to round 48 were wiped. However, our final accuracy was still respectable, at 76.1%, as well as other metrics also performing as expected.

6.8 Impurity

This merger is static and does not rely on accuracy which makes it less computationally expensive compared to the other models, making it ideal for multiparty decentralized computation where there is no server to coordinate. However, this is beyond the scope of this paper. Figure B.6 shows that our starting accuracy is very low. Similar to other methods the data is very non iid this causes a slower start. However, impurity reduction is likely more affected by this than other mergers. Even clients with lower accuracies can have high impurity reduction and this causes the global model to have a slower start compared to other mergers. Once the model stabilizes around 65% accuracy other metrics improve with it as well. The model shows improvement until the last rounds, around round 80 the model starts declining. This is again due to impurity reduction ≠ accuracy. While theoretically impurity reduction should mean better performance, in the context of FL and highly non-iid data impurity reduction alone does not perform well as a merging metric. Our model finishes around 64.3%.

6.9 Diversity

Figure 6.1 shows that diversity merger similar to other mergers start at around 58% accuracy. Over the first 6 or so rounds it recovers quickly and gets to about 75% accuracy. This is expected as the global model has now seen a lot more data to make tree choices. After shortly stabilizing, the accuracy jumps to 78.1% at around round 20 which is where it stays for the rest of the training. Many other mergers increase in accuracy at the same round. The reason why the merger does not improve further from new data is because it believes that the 50 trees it has is sufficiently diverse and there are no other models or trees that can provide more accurate results. This is due to how we weight our trees. This experiment was tuned once again giving an emphasis on accuracy with diversity taking a back seat. However, our final performance is better than purely accuracy based merger, or weighted accuracy merger, at 78.1%. Also noting that our other metrics outperformed `best accuracy` merger as well.

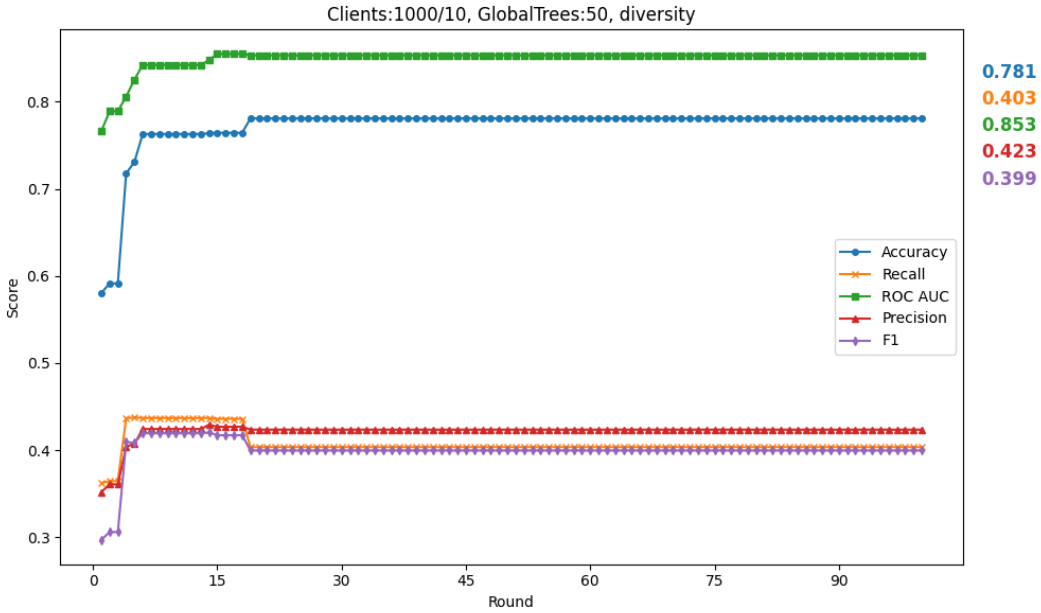

Figure 6.1: *Diversity merger performance over 100 rounds with 1000 clients and 1% participation.*

6.10 Top_k

Top_k merger behaves very similarly to `best accuracy` merger. Top_k is designed to slowly fill the global model. For our experiment, we test with 10% of the global model is filled each round. So our global model size is 50 trees and each round we choose 5 trees for the global model from the client models. Once the global model is filled we replace trees using the same logic as `best accuracy` merger. Figure B.7 shows that our model is faster to converge and our final accuracy is given as 77.0%. However, this is not the intended use of this merger. To make the client model uploads smaller we will decrease the number of trees clients train. In turn we can increase the global model size to compliment this, helping us capture more data. However for the sake of fairness we will keep the global model size at 50 trees for this experiment. Training with 10 trees on each client, 50 trees on the global model and filling only the top 10 percent of the global model each round we can see that in Figure B.8 we have comparable results. While our global model decreases in accuracy initially it bounces back right after. The effects of this decrease is less so visible with more trees on clients. However our accuracy is still 77.0% with a slightly increased ROC_AUC score. And a minor decrease to F1 score. This indicates that since the model is smaller it is having a harder time capturing as much information. This proves that even with around 30% of the original client ensemble size, we can get comparable results while reducing the communication overhead greatly.

6.11 Best Performing Merger

Most of our mergers performed good. Table 6.5 shows our mergers accuracies. We can see that the best performing merger is diversity at 78.1%, performing better than the closes merger by 1.1%. Our worse performing merger was impurity with an accuracy score of 64.3%.

Table 6.5: *Table of merger accuracies*

Merger Method	**Accuracy**
Best Accuracy	76.7%
Accuracy Weighted	75.2%
Data Weighted - 60/40	76.5%
Data Weighted - 40/60	76.1%
Impurity	64.3%
Diversity	78.1%
Top_K Classic	77.0%
Top_K Small	77.0%

Since diversity is our best performing model we will be conducting our remaining experiments on it. We will be testing the change in performance in terms of accuracy when using serverside pruning. The expectation is that this pruner will lighten the computational load without reducing accuracy. However we have no direct way to calculate compute cost for this reason we will focus on accuracy to see if it increases or decreases

6.12 Pruning

When using pruning with the diversity method we can see in figure B.9 that there is an improvement of precision and a slight increase in ROC_AUC score, 50.3% and 86.4% respectively. However, the accuracy, F1-Score and recall are all slightly worse, accuracy dropping to 77.5% from the baseline of 78.1%. While we have no direct way of measuring the speed of the model, other than timing which would be imprecise, we can look at what the pruner does. It is implemented to print the trees its pruning and by following the printout of our code round by round

we can see that around 0-25 trees every round is pruned. This still needs further work to be conclusive but it proves to be promising.

7 Conclusion

In this paper we investigated the experimental application of federated learning with Random Forest for multi-class IoT Anomaly detection. The goal was to explore whether intelligent model merging strategies could perform comparable to a centrally trained model. Amongst our merge strategies, **diversity-based merge** stood out. Delivering accuracies up to **78.1%** while balancing accuracy and uniqueness of predictions across trees.

The experiments demonstrated that combining diverse predictions with a client's overall performance produced a more stable and generalizable global model. Interestingly, the **Top_K** method, even with drastically smaller local models, did not lead to performance degradation. This finding suggests that significant reductions in communication and memory cost are possible without compromising accuracy, provided the global model is structured appropriately. Furthermore, server-side pruning showed potential for reducing computational load, making it a promising direction for efficiency improvements.

7.1 Reflection

The computational cost of this project was rather high. I should have switched to a HPC (High-Performance Compute) node from the start. Also the specialized nature of the topic meant that a lot of research was required but there was not enough research on my topic.

7.2 Future work

There are several other directions this paper can be extended. First, privacy considerations being made, concepts like differential privacy, blockchain and secure multipart computation/aggregation could be explored to bring federated learning closer to real life implementation. Secondly, communication and computation costs can be optimized. Lastly, more emphasis on client side can be given, using methods like personal client updates and using Multi-Part Computation (MPC) we can aim to reduce redundant trees by pruning them.

A Appendices

B Evaluation Graphs

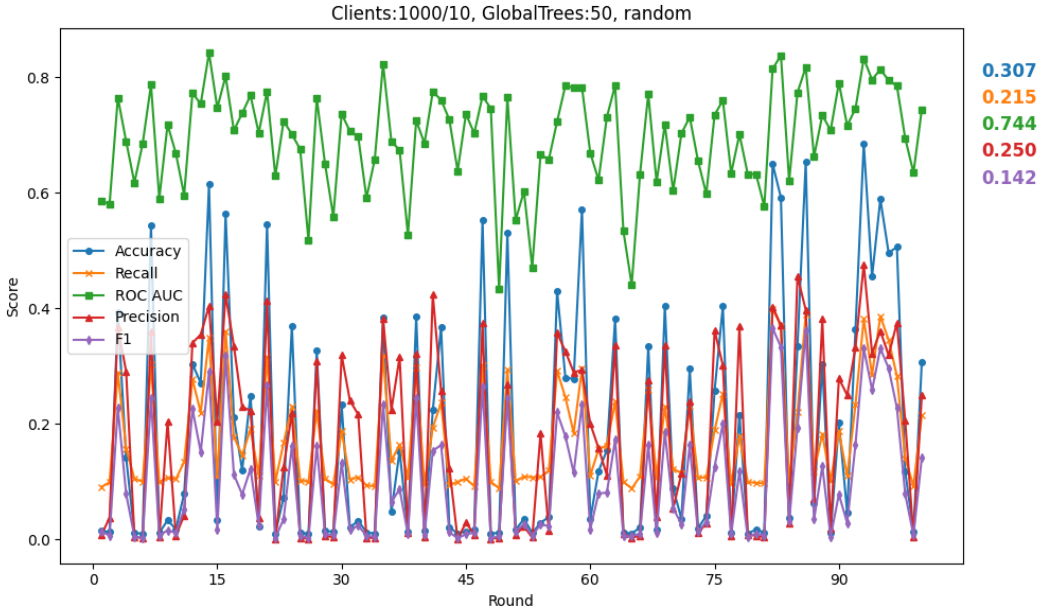

Figure B.1: *Random merger performance over 100 rounds with 1000 clients and 1% participation.*

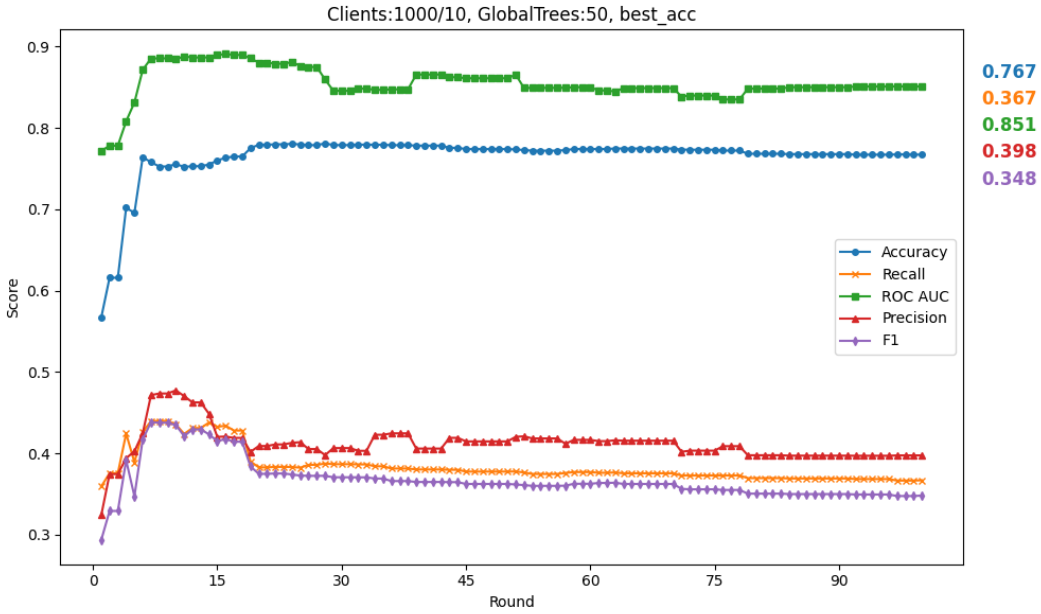

Figure B.2: *Best accuracy merger performance over 100 rounds with 1000 clients and 1% participation.*

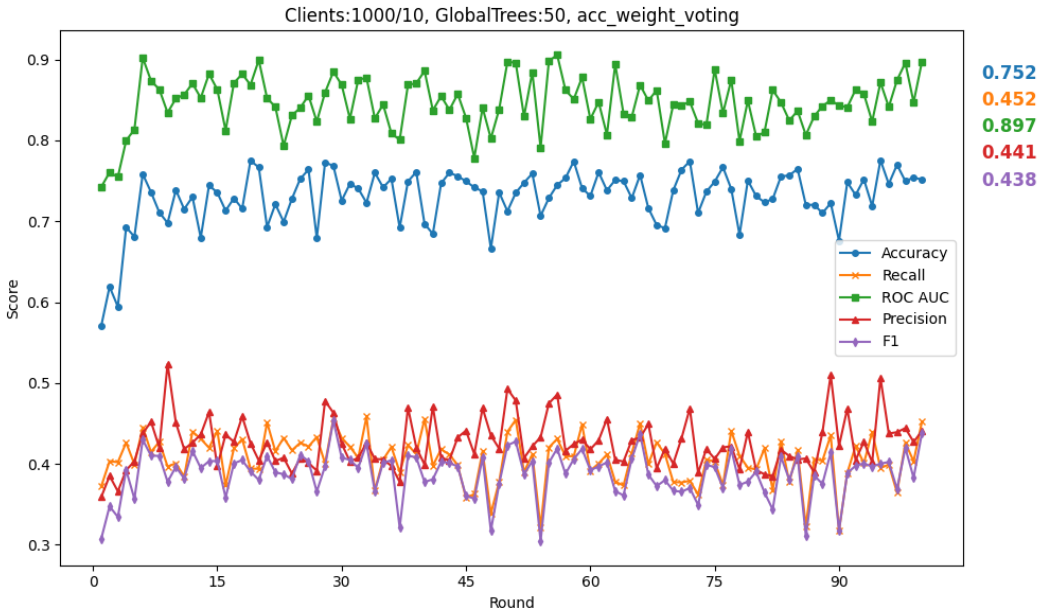

Figure B.3: *Accuracy Weighted merger performance over 100 rounds with 1000 clients and 1% participation.*

Figure B.4: *Data Weighted, 60/40 favouring accuracy, merger performance over 100 rounds with 1000 clients and 1% participation.*

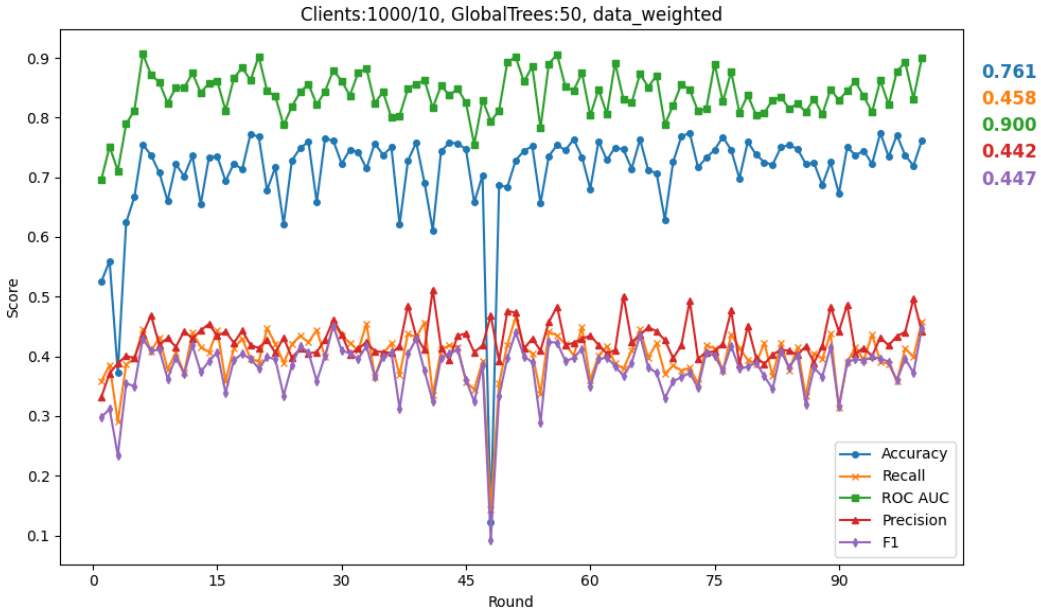

Figure B.5: *Data Weighted, 40/60 favouring data, merger performance over 100 rounds with 1000 clients and 1% participation.*

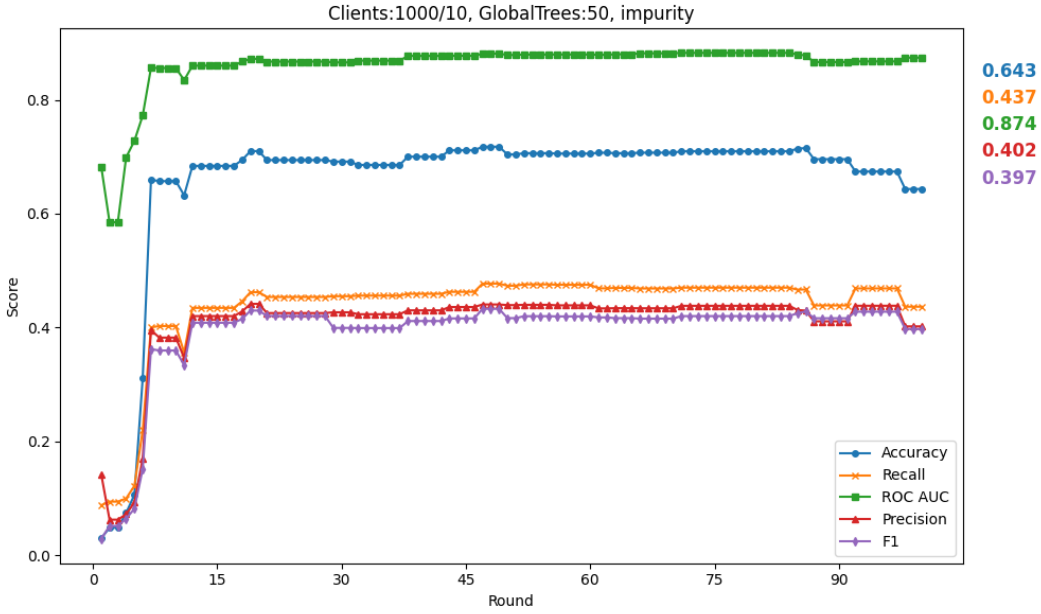

Figure B.6: *Impurity merger performance over 100 rounds with 1000 clients and 1% participation.*

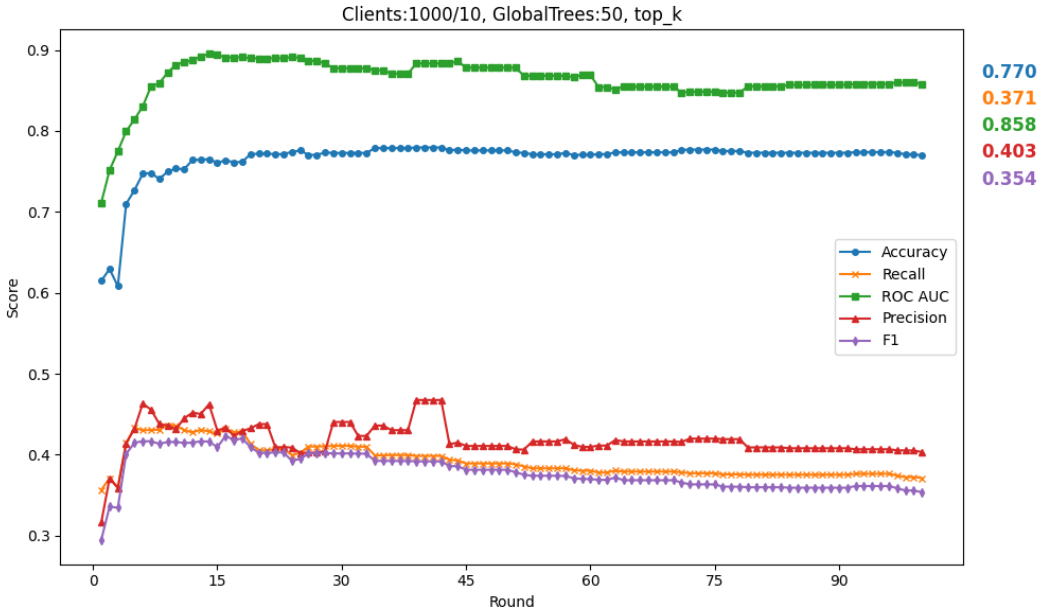

Figure B.7: *Top_k merger performance over 100 rounds with 1000 clients and 1% participation.*

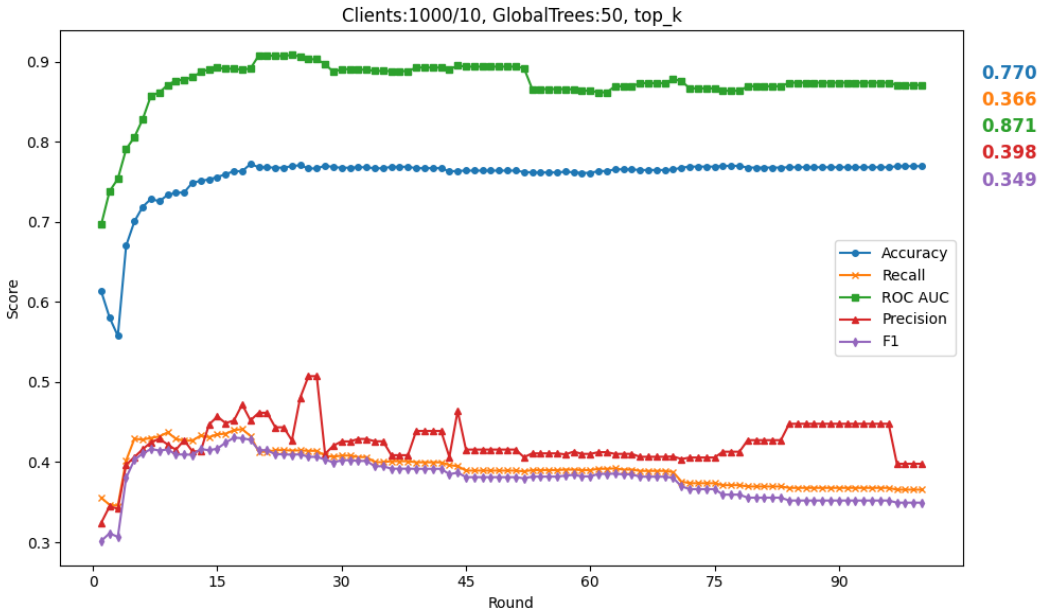

Figure B.8: *Top_k merger performance over 100 rounds with 1000 clients and 1% participation this time using 5 tree ensemble for clients.*

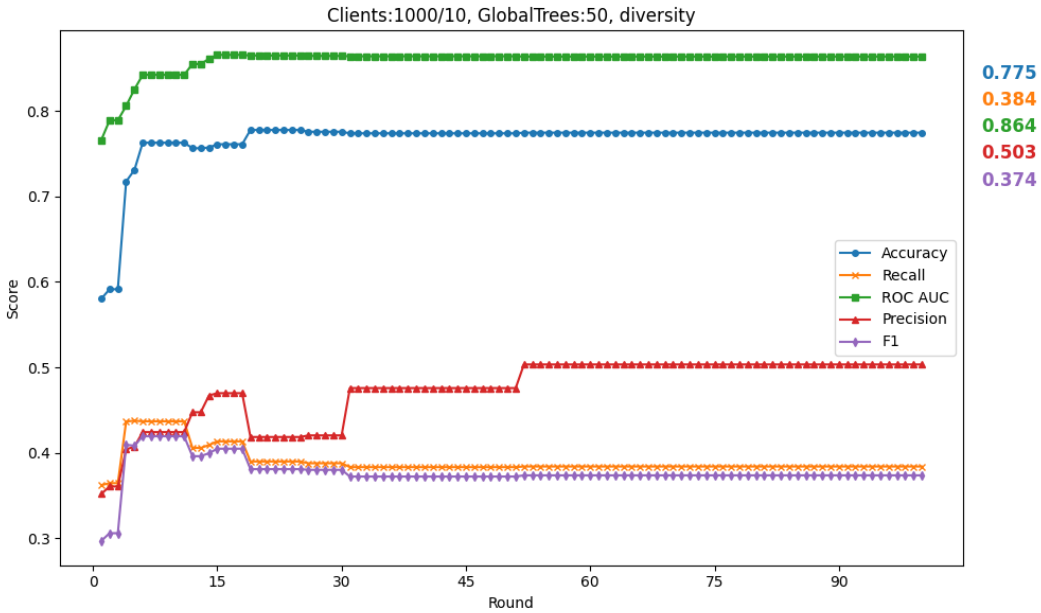

Figure B.9: *Diversity merger performance with server pruning over 100 rounds with 1000 clients and 1% participation.*

C Code

- Copies of ethics approvals (you must include these if you needed to get them)
- Copies of questionnaires etc. used to gather data from subjects. Don't include voluminous data logs; instead submit these electronically alongside your source code.
- Extensive tables or figures that are too bulky to fit in the main body of the report, particularly ones that are repetitive and summarised in the body.
- Outline of the source code (e.g. directory structure), or other architecture documentation like class diagrams.
- User manuals, and any guides to starting/running the software. Your equivalent of `readme.md` should be included.

Don't include your source code in the appendices. It will be submitted separately.

```
configs = {
   #Federated Features
   "NUM_CLIENTS": 1000,# If you set the client as 1 it will be centralized
       training, you should also set the size of val to 0 aswell then
   "clients_per_round": 10,
   "global_num_trees": 150,
   "num_rounds": 100,
   "merge_method": "top_k",# random, best_acc, acc_weight_voting, data_weighted,
       impurity, diversity, top_k
   "prune_similar": False,
   "client_pruning": False,
   "global_rev_data_wgt": True, #Only used for data_weighted merge method to
       include global model in reevaluation
   "top_k": 10.0, #Percentage of trees to select from all clients or absolute
       number of trees eg 5.0 or 5 respectively
   #Training parameters
   "num_trees": 5, #On Client
   "max_depth": None,#maybe none
   "num_max_features": 17,
   "n_jobs": -1,
   #Data features partition
   "test_size": 0.2,
   "val_size": 0.1, #There should be a val dataset even for centralized for
       quality checks.
   "dirichlet_alpha": 0.8,
   #Model parameters
   "criterion": "gini", # "gini" or "entropy"
}
```

Listing C.1: *Configs Dictionary used for easy access of experiment variables with sample values*

```
def are_trees_similar(self, tree1, tree2, similarity_threshold=0.8):
   """Compare two trees based on their structure and parameters."""
   tree1_struct = tree1.tree_
   tree2_struct = tree2.tree_

   # If trees have very different node counts, they're not similar
   if abs(tree1_struct.node_count - tree2_struct.node_count) > 5:
      return False

   # Check only the first few nodes for efficiency.
   max_check_node = min(7, min(tree1_struct.node_count, tree2_struct.node_count))
   similar_nodes = 0
   for i in range(max_check_node):
      # If both nodes are leaf nodes, count as similar
      if tree1_struct.children_left[i] == -1 and tree2_struct.children_left[i] ==
          -1:
         similar_nodes += 1
         continue

      # If one is leaf and the other is not, skip comparison.
      if (tree1_struct.children_left[i] == -1) != (tree2_struct.children_left[i]
          == -1):
         continue

      # Check if split features are the same and thresholds are close.
      if tree1_struct.feature[i] == tree2_struct.feature[i]:
         if abs(tree1_struct.threshold[i] - tree2_struct.threshold[i]) < 0.1:
            similar_nodes += 1

   similarity = similar_nodes / max_check_node
   return similarity > similarity_threshold
```

Listing C.2: *Tree similarity metric*

```
# ---------------------------------------------------------------------
# 2) Allocate fraction for each client with constraints
# - If client acc < min_acc => fraction <= low_acc_contrib
# - Otherwise fraction = (acc / total_acc), but <= max_contrib
# ---------------------------------------------------------------------
# We'll store fraction_of_trees in a dict, then do integer pass
fractions = {}
n_clients = len(client_models)
# If total_acc == 0, fallback: distribute evenly or skip them all
if total_acc == 0:
   # fallback approach
   for i, _ in sorted_by_acc:
      fractions[i] = 1.0 / n_clients # or 0
else:
   for i, acc in sorted_by_acc:
      if acc < min_acc:
         # Force them to at most low_acc_contrib
         frac = min(low_acc_contrib, 1.0) # just in case
      else:
         # Proposed fraction is (acc / total_acc)
         proposed = (acc / total_acc)
         # clamp to max_contrib
         frac = min(proposed, max_contrib)
      fractions[i] = frac
# Normalize fractions so they sum to <= 1
# But some clients might be clamped, so sum_of_fractions could be < 1
sum_frac = sum(fractions.values())
# If sum_frac > 1, we reduce them proportionally
if sum_frac > 1:
   for i in fractions:
      fractions[i] /= sum_frac
   sum_frac = 1.0
# ---------------------------------------------------------------------
# 3) Integer allocation + leftover distribution
# ---------------------------------------------------------------------
trees_per_client = {}
leftover = num_input
# First pass: integer allocation
for i, acc in sorted_by_acc:
   frac = fractions[i]
   count = int(math.floor(frac * num_input))
   trees_per_client[i] = count
   leftover -= count
# Now distribute leftover among top 30% of *scores* in round-robin
top_k = max(1, int(0.3 * n_clients))
top_slice = sorted_by_acc[:top_k] # list of (client_idx, accuracy), best first
idx_roundrobin = 0
while leftover > 0 and top_slice:
   c_idx, _ = top_slice[idx_roundrobin]
   trees_per_client[c_idx] += 1
   leftover -= 1
   idx_roundrobin += 1
   if idx_roundrobin >= len(top_slice):
      idx_roundrobin = 0
print("Trees allocated (before picking best trees per client):")
for i in range(len(client_models)):
   print(f"Client {i} => {trees_per_client.get(i, 0)}")
```

Listing C.3: *Weiging and tree allocation with comments for undrestandbility. Taken from the merge_models_acc_weighted function in ModelMerger class*

```
def patched_predict_proba(model, X, global_classes):
   tree_probas = []
   for tree in model.estimators_:
      p = tree.predict_proba(X)
      tree_model_classes = tree.classes_
      patched_p = ModelEvaluator.patch_single_tree_proba(p, tree_model_classes,
          global_classes)
      tree_probas.append(patched_p)
   return np.mean(tree_probas, axis=0)
```

Listing C.4: *Patches the whole model*

```
def patch_single_tree_proba(p, model_classes, global_classes):
   full_p = np.zeros((p.shape[0], len(global_classes)))
   for i, cls in enumerate(global_classes):
      if cls in model_classes:
         full_p[:, i] = p[:, model_classes == cls].flatten()
   return full_p
```

Listing C.5: *Patched tree proba logic*

Bibliography

Ari, A. A. A., Aziz, H. A., Njoya, A. N., Aboubakar, M., Djedouboum, A. C., Thiaré, O. and Mohamadou, A. (2024), 'Data collection in iot networks: Architecture, solutions, protocols and challenges', *IET Wireless Sensor Systems* .
URL: *https://api.semanticscholar.org/CorpusID:270297779*

Ayodele, T. O. (2010), 'Machine learning overview', *New Advances in Machine Learning* **2**(9-18), 16.

Bagdasaryan, E., Veit, A., Hua, Y., Estrin, D. and Shmatikov, V. (2020), How to backdoor federated learning, *in* S. Chiappa and R. Calandra, eds, 'Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics', Vol. 108 of *Proceedings of Machine Learning Research*, PMLR, pp. 2938–2948.
URL: *https://proceedings.mlr.press/v108/bagdasaryan20a.html*

Bandara, V. L. (2021), 'What is a decision tree in ml?'. Accessed: 2025-03-24.
URL: *https://vitiya99.medium.com/what-is-a-decision-tree-in-ml-5bd76efc2232*

Bonawitz, K., Ivanov, V., Kreuter, B., Marcedone, A., McMahan, H. B., Patel, S., Ramage, D., Segal, A. and Seth, K. (2016), 'Practical secure aggregation for federated learning on user-held data', *ArXiv* **abs/1611.04482**.
URL: *https://api.semanticscholar.org/CorpusID:10933707*

Breiman, L. (2001), 'Random forests', *Mach. Learn.* **45**(1), 5–32.
URL: *https://doi.org/10.1023/A:1010933404324*

Chandola, V., Banerjee, A. and Kumar, V. (2009), 'Anomaly detection: A survey', *ACM Comput. Surv.* **41**(3).
URL: *https://doi.org/10.1145/1541880.1541882*

Daghero, F., Burrello, A., Xie, C., Benini, L., Calimera, A., Macii, E., Poncino, M. and Pagliari, D. J. (2021), 'Adaptive random forests for energy-efficient inference on microcontrollers', *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)* pp. 1–6.
URL: *https://api.semanticscholar.org/CorpusID:244272575*

Dhull, P., Guevara, A. P., Ansari, M., Pollin, S., Shariati, N. and Schreurs, D. (2022), 'Internet of things networks: Enabling simultaneous wireless information and power transfer', *IEEE Microwave Magazine* **23**(3), 39–54.

Dwork, C., McSherry, F., Nissim, K. and Smith, A. (2006), Calibrating noise to sensitivity in private data analysis, *in* 'Proceedings of the Third Conference on Theory of Cryptography', TCC'06, Springer-Verlag, Berlin, Heidelberg, p. 265–284.
URL: *https://doi.org/10.1007/11681878*$_1$4

Kairouz, P., McMahan, H. B., Avent, B., Bellet, A., Bennis, M., Bhagoji, A. N., Bonawitz, K., Charles, Z. B., Cormode, G., Cummings, R., D'Oliveira, R. G. L., Rouayheb, S. Y. E., Evans, D., Gardner, J., Garrett, Z., Gascón, A., Ghazi, B., Gibbons, P. B., Gruteser, M., Harchaoui, Z., He, C., He, L., Huo, Z., Hutchinson, B., Hsu, J., Jaggi, M., Javidi, T., Joshi, G., Khodak,

M., Konecný, J., Korolova, A., Koushanfar, F., Koyejo, O., Lepoint, T., Liu, Y., Mittal, P., Mohri, M., Nock, R., Özgür, A., Pagh, R., Raykova, M., Qi, H., Ramage, D., Raskar, R., Song, D. X., Song, W., Stich, S. U., Sun, Z., Suresh, A. T., Tramèr, F., Vepakomma, P., Wang, J., Xiong, L., Xu, Z., Yang, Q., Yu, F. X., Yu, H. and Zhao, S. (2019), 'Advances and open problems in federated learning', *Found. Trends Mach. Learn.* **14**, 1–210.
URL: *https://api.semanticscholar.org/CorpusID:209202606*

Kolias, C., Kambourakis, G., Stavrou, A. and Voas, J. (2017), 'Ddos in the iot: Mirai and other botnets', *Computer* **50**(7), 80–84.

Konecný, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T. and Bacon, D. (2016), 'Federated learning: Strategies for improving communication efficiency', *ArXiv* **abs/1610.05492**.
URL: *https://api.semanticscholar.org/CorpusID:14999259*

Li, T., Sahu, A. K., Talwalkar, A. and Smith, V. (2020), 'Federated learning: Challenges, methods, and future directions', *IEEE Signal Processing Magazine* **37**(3), 50–60.

Liu, Y., Kang, Y., Zou, T., Pu, Y., He, Y., Ye, X., Ouyang, Y., Zhang, Y.-Q. and Yang, Q. (2024), 'Vertical federated learning: Concepts, advances, and challenges', *IEEE Transactions on Knowledge and Data Engineering* **36**(7), 3615–3634.

Markovic, T., Leon, M., Buffoni, D. and Punnekkat, S. (2022), Random forest based on federated learning for intrusion detection, *in* I. Maglogiannis, L. Iliadis, J. Macintyre and P. Cortez, eds, 'Artificial Intelligence Applications and Innovations', Springer International Publishing, Cham, pp. 132–144.

McMahan, H., Moore, E. and Ramage, D. (2016), 'Federated learning of deep networks using model averaging', *Google Research* .

Valente, L., Tortorella, Y., Sinigaglia, M., Tagliavini, G., Capotondi, A., Benini, L. and Rossi, D. (2022), 'Hulk-v: a heterogeneous ultra-low-power linux capable risc-v soc', *2023 Design, Automation & Test in Europe Conference & Exhibition* pp. 1–6.
URL: *https://api.semanticscholar.org/CorpusID:254044474*

Yang, Q., Liu, Y., Chen, T. and Tong, Y. (2019), 'Federated machine learning: Concept and applications', *ACM Trans. Intell. Syst. Technol.* **10**(2).
URL: *https://doi.org/10.1145/3298981*

Yang, T., Andrew, G., Eichner, H., Sun, H., Li, W., Kong, N., Ramage, D. and Beaufays, F. (2018), 'Applied federated learning: Improving google keyboard query suggestions'.
URL: *https://arxiv.org/abs/1812.02903*