What is the influence on the energy consumption when using different supervised machine learning models trained using federated learning for the binary classification of network traffic in intrusion detection systems?

A comparison of the energy consumption of federated Support Vector Machine and Random Forest supervised models for binary classification in network intrusion detection systems.

Student ID: 2136685

Supervisor: Dr. Sandy Taramonli

Warwick Manufacturing Group

University of Warwick

2021-2024

Abstract

This project investigates the energy consumption of the Random Forest and Support Vector Machine supervised machine learning models when implemented in a centralised federated learning cluster. The project aims to collect the energy consumption of both models and present them in a way that discerns which model is more accurate and energy efficient.

Most literature found in the field of federated learning for supervised models has diminished, due to deep learning models becoming the emerging research topic. Yet, less complex models are required for devices that do not have the energy or computational overhead. Moreover, their energy consumption must be measured to determine whether they are even applicable for such devices.

This project uses the InSDN, CSE-CIC-IDS2018 and the CIC-IDS2017 datasets as the local datasets for three client machines which each train an RF and an SVM model on their allocated local datasets. These local models are aggregated by a server machine using the FedAvg method.

The project concluded that the RF model is more suited for binary classification in an energy-limited IDS. The RF federated cluster outperforms the SVM federated cluster in terms of accuracy. Moreover, the energy consumption of the RF federated cluster (0.24 kWh) was 25 times less than the SVM federated cluster (6.08 kWh), making the federated RF cluster more energy-efficient.

This project aligns with the following CyBoK skills:

- 8. Security Operations and Incident Management (SOIM)

Acknowledgements

First and foremost, I would like to express my sincerest gratitude to my project supervisor, Dr. Sandy Taramonli, for her continuous, precise and timely support. She has been an irreplaceable and foundational part of this project throughout the whole academic process, always being there to help, even during days off.

I would also like to express my gratitude to my friends and family who have supported me over the course of the three years on the BSc Cyber Security course, who have always pushed me to achieve higher feats.

Abbreviations

Intrusion Detection System	IDS
Federated Learning	FL
Deep Learning	DL
Machine Learning	ML
Denial of Service	DoS
Internet of Things	IoT
Users to Root	U2R
Remote to Local	R2L
Principal Component Analysis	PCA
Personally Identifiable Information	PII
Independent Identically Distributed	IID
Convolutional Neural Network	CNN
Borderline-Synthetic Minority Over-Sampling Technique	B-SMOTE
Confidentiality, Integrity and Availability	CIA
Stochastic Gradient Descent	SGD
Man-in-the-Middle	MITM
Device-to-Device	D2D
Multi-access Edge Computing	MEC
Generative Adversarial Networks	GAN
Long short-term memory	LSTM
Decision Tree	DT
Naive Bayes	NB
Random Forest	RF
Support Vector Machine	SVM

Contents

List of Figures

List of Tables

1 Introduction

Machine learning algorithms have been at the forefront of cyber security organisations developing their own intrusion detection systems for the better part of the decade. However, companies training a machine learning model must process large volumes of highly sensitive network traffic data, which raises concerns for privacy and security. Additionally, the centralised methods of current machine learning techniques further increase the concern for data security and privacy, due to the single-point-of-failure nature of the training methodologies, risking data breaches, leakages and unauthorised access to the sensitive data.

Training a machine learning model for the purpose of implementing an intrusion detection system can prove difficult due to the heterogeneous nature of network data. Datasets with insufficient samples or diversity can render a model attempting to detect anomalies in a network ineffective at best. Moreover, the availability of large and diverse datasets has proven to be difficult to find and commonly, machine learning algorithms are trained on old datasets, such as the KDD-Cup 99 or the DARPA 1999, which do not accurately represent modern network traffic (Khraisat et al. 2019).

In recent years, research into the field of federated learning for machine learning algorithms has become a more discussed subject. Data privacy, security and unauthorised access have become critical points that all organisations must address to protect the information of their users, with added pressure from the data protection laws of the United Kingdom. Federated learning iteratively trains a global model by using an aggregation of the models generated from the result of several machines which each train a local model using their respective local data, resulting in no data being transferred. As this machine learning approach does not result in data being transferred, it addresses privacy concerns that traditional machine learning methods are affected by (Agrawal et al. 2022).

Moreover, the field of federated learning in IoT devices has become an emerging field of research. Researchers are theorising and implementing frameworks and methods of training machine learning models on IoT devices with large limits on their power and computational overhead, while still keeping the local data private and secure using federated learning.

This project aims to investigate the current centralised and decentralised federated learning methodologies and investigate the energy consumption of the Random Forest and Support Vector Machine models for Intrusion Detection Systems using the centralised federated learning method. The datasets used for training and testing the models are the, the InSDN, the CSE-CIC-IDS2018 and the CIC-IDS2017, providing network traffic from a diverse range of network environments (Elsayed et al. 2020).

The project will evaluate the differences in accuracy and energy consumption for each of the models stated. These findings will aid in drawing a conclusion on which model is more accurate while being as energy efficient as possible using the methods.

1.1 Research Question and Objectives

The following outlined question aims to address the aims of the introduction and to draw a conclusion to the research conducted in this paper:

What is the influence on the energy consumption when using different supervised machine learning models trained using federated learning for the binary classification of network traffic in intrusion detection systems?

To successfully investigate the title above, the following objectives have been defined.

Primary Objectives:

- To evaluate the current frameworks for centralised and decentralised federated learning for privacy-preserving network intrusion detection systems.
- To assess the current aggregation methods for federated learning.
- To analyse the suitable datasets chosen for the ML models stated and process them for the training of the RF and SVM ML models.
- To train and validate the RF and SVM ML models for binary classification in an IDS.
- To validate and evaluate the RF and SVM models using centralised federated learning for an IDS.
- To identify the most accurate and energy-efficient federated supervised ML model that is suited for an energy-limited IDS.

2 Literature Review

This section aims to provide an understanding of the established research field that is network intrusion detection systems and machine learning, with a focus on federated learning. The section also aims to highlight the open problems found in the currently available research to determine gaps in it.

The literature is divided into into five distinct sections. The first section explores the available research on Random Forest and SVM ML models. The second section explores the field of centralised federated learning in an IDS. The third section explores the field of decentralised federated learning in an IDS. The fourth section provides an understanding of the available aggregation algorithms for federated learning. The fifth section investigates research that is similar to the research question.

2.1 Machine Learning Models

In comparing machine learning models for network intrusion detection systems, A et al. (2023) found that, when training and testing a Random Forest model with PCA analysis for feature selection, with an 80:20 split against the NSL-KDD dataset, a more modern version of the KDD-99 dataset, the model achieved 99.89% accuracy. They found that this model achieved a better accuracy than the SVM model's accuracy of 97.18%. A et al. (2023) used accuracy, precision, recall and f1-score to compare the models, as well as confusion matrices and receiver operating characteristic (ROC) curves. Kumar & Malathi (2022) support these findings as well, even without the usage of PCA analysis.

Bhoria & Garg (2013) found the C4.5 Decision Tree to be the most accurate classification algorithm, followed by Random Forest and SVM. Aung & Min (2017) expanded on the research and used the KDD-99 dataset to conduct a supervised approach to classification. The KDD-99 consists of simulated DoS, U2R, R2L and Probe attacks and is known for being unbalanced. The researchers use the K-means algorithm to turn the dataset into a more homogeneous one, before using the new formed dataset to train and test the Random Forest model. They found that using K-means clustering in combination with Random Forest reduces CPU usage and memory usage.

Kumar & Dhanalakshmi (2023) compared an SVM and Random Forest model for a host-based IDS, both using a sample size of n = 10 and a g-power value of 80%. They used SMOTE on five datasets to balance the traffic types. They found that SVM (95.89%) outperforms Random Forset (94.12%) in terms of accuracy. The researchers discussed that SVM is better at dealing with extreme cases than Random Forest. The researchers assert that SVM is effective when there are more dimensions than samples. The researchers concluded that the classification accuracy of SVM is greater than that of Random Forest, though further tweaks are possible which may change the outcome. In this paper, the researchers did not use hyperparameters to further improve the accuracy of the models, therefore not reaching the full accuracy potential. They do not show how the accuracy is calculated, and completely omit other metrics such as recall, precision or f1-score.

Waskle et al. (2020) devised a Random Forest model, using PCA to reduce the dimension of the given dataset. The researchers assert that PCA is one of the most efficient and accurate methods of reducing the dimension of data. The method devised gave the researchers an accuracy of 97.78% for the PCA with Random Forest (PCA-RF) model, whereas SVM had an accuracy of 84.34%. Moreover, the performance time of PCA-RF model was 3.42 minutes, whereas SVM had one of 4.57. Additionally, PCA-RF had an error rate of 0.21%, whereas SVM had one of 2.67%. The researchers concluded that PCA-RF had improved detection and false error rates, compared to SVM. Although the research was conducted in 2020, the researchers chose to use the 1999 KDD dataset. This is a dataset that no longer represents the network traffic of the 2020s, especially the attack traffic.

Liao et al. (2020) have implemented an IDS system using the GAN model. They assert that GAN distinguishes itself from other models such as RF by being a supervised learning multi-classification model as opposed to a binary classification model. They implement a three-layer LSTM network as the generator of the model. They used an artificial NN at the classification model. Their results showed that GAN was slightly better than DT and NB, with an accuracy of 76.82%. However, RF still shows better performance in all metrics, including accuracy (83.98%). Zhang et al. (2020) also implemented an LSTM network, with better results than Liao et al. (2020). The difference was that Zhang et al. (2020) had a more complex data processing methodology. In this research, they used a QPSO algorithm for feature selection on the KDD99 dataset, allowing them to reach an accuracy of 97.79%. Liao et al. (2020) did not mention data processing and used the NSL-KDD dataset, an improved version of KDD99.

2.2 Federated Learning for Intrusion Detection Systems

Novikova & Golubev (2023) researched the application of FL in an IDS. They propose a methodology for evaluating the performance of the ML model trained, considering attack detection rate, in the context of the training computational performance. The researchers asserted that this evaluation methodology must include FL specific features, such as data partition and distribution, aggregation functions and computation and memory resources of the collaborating entities. The researchers address the IDS construction and evaluation. They propose an FL IDS architecture and guidelines for assessing its performance. The researchers implemented the FL IDS in Python using the Flower framework, achieving 99% accuracy. They asserted that future research will include a thorough evaluation of different aggregation methods for the scenarios outlined.

Li et al. (2023) propose a dynamic weighted aggregation FL (DAFL) IDS system. Compared to other FL methods, DAFL implements dynamic filtering and weighing strategies for local models. The researchers assert that this allows DAFL to perform better as an IDS in situations with less communication overhead. They state that DAFL can reduce the impact of poorly performing local models on the global model. They provide the pseudocode for the DAFL method. In terms of accuracy, precision, recall and f1-score, the research found that this method produces similar results to FL using FedAvg, as researched in Brendan et al. (2016). However, due to the 50% reduced communication rounds when compared to FedAvg, this research could be applied to areas such as the industrial IoT, where the network communication is limited.

2.3 Decentralised Federated Learning

In this paper, Assis & Hessel (2022) discusses decentralised FL techniques, with a focus on IoT systems. They pose the problem of guaranteeing the CIA triad, authenticity and non-repudiation in distributed IoT systems. They raised the challenges that: (1) IoT devices are not powerful enough for security. (2) Centralised cloud systems are a significant privacy risk, therefore the data must be processed locally. (3) Supervised models are becoming obsolete due to manually selected features and the everchanging features generated by new IoT devices. Deep learning algorithms are becoming more suitable. (4) There must be mechanisms to guarantee the CIA triad, security and privacy in the IoT. Assis & Hessel (2022) describe aggregation methods such as SGD, FedAvg, and FedCS

(FL with Client Selection). Assis & Hessel (2022) identified open problems and future work such as (1) constrained resources being a bottleneck for security in IoT devices. (2) Minimising communication costs. (3) Privacy risks due to model sharing. They propose differential privacy, but it may be incompatability with IoT devices. (4) Deep learning causes high demand for computational and storage resources, which IoT devices do not have. There must be a technique and algorithm that improves power consumption efficiency, including how to estimate available energy at each node. (5) Blockchain-based Access Control. Proof of work and proof of stake in the blockchain consume high amounts of energy, which is incompatible with IoT devices.

Gupta & Alam (2022) conducted a survey for distributed FL approaches, aiming to illustrate a comprehensive review and conduct a comparative analysis of them. The researchers discuss FL in different distributed environments, centralised and decentralised. They state common methods for implementing decentralised FL, mainly distributed ledger techniques such as blockchain flameworks, i.e. BLADE-FL. Another technique discussed was *BrainTorrent*, a peer-to-peer decentralised FL method, researched by Roy et al. (2019). Another technique stated was decentralised SGD with a star-like network topology, as researched by Xing et al. (2020). Another technique was from a study on *fog learning*, by Hosseinalipour et al. (2020). The researchers conclude that FL leans significantly towards IoT networks and that major emphasis has been placed on the privacy of data and data isolation. Future work was described as breaking the blocks between enterprises and exploring a new community where data can be securely shared together. This research analyses mostly theoretical frameworks, none of which have a usable implementation currently available.

Roy et al. (2019) propose a peer-to-peer decentralised FL environment called *BrainTorrent*. BrainTorrent is set up in a mesh topology, where a client, C_1, sends a ping request to all other clients to do a version check of the ML model. Only clients that have a new version of the model send their weights to C_1. An aggregated model is formed at C_1 and the new model is tuned with the local data of C_1. The researchers provide the pseudocode for the training algorithm of BrainTorrent. This paper is for decentralised FL in medical applications, so the researchers demonstrated the effectiveness of BrainTorrent by comparing it with FL while doing whole-brain segmentation of MRI T1 scans. They concluded that under multiple experiments, BrainTorrent achieves up to 7% better performance than traditional FL. They asserted that BrainTorrent resolves the issue of relying on a central server, as well as reaching performance similar to a model trained on pooled data. They state that although the experiments were medically-specific, BrainTorrent is generic and it can be used for any ML training. However, BrainTorrent is only shown to be effective using the mesh topology.

Wilt et al. (2021) developed a decentralised FL library for Python called Scatterbrained, available online at `https://github.com/JHUAPL/scatterbrained`. They state that the decentralised methodology should be decoupled from the ML model, having no bearing on the model. The researchers state that frameworks like Roy et al. (2019) and Lalitha et al. (2019) are too purpose-built and cannot give the user sufficient flexibility for topologies or data-sharing preferences. Wilt et al. (2021) developed an API, abstracting node communications across the topology for customisation of weight sharing. In this paper, they provide example code of the implementation, i.e. a new edge device can connect to an existing network in "leeching" mode, and download an ML model from a peer, allowing the new node to begin participating in the FL network. The researchers discussed the future areas of research as being the field of low-power, low-bandwidth edge-computing resources, such as IoT devices. This library is highly useful for its topology flexibility and model-framework decoupled nature. A common theme across the Python libraries for FL is the need for ongoing development to maintain relevance in the rapidly evolving field of ML. This includes regular updates and comprehensive documentation to ensure that the resources remain useful for practitioners. As a result, this library can not be used in a reasonable amount of time due to the lack of documentation.

2.4 Aggregation Algorithms

Brendan et al. (2016) developed the FedAvg aggregation method. FedAvg is a method based on FedSGD, but distinct as instead of updating the gradients, FedAvg updates the weights. The researchers tested these methods using a dataset built from the complete works of William Shakespeare, which is unbalanced. Another version was created, which is balanced and IID. The researchers' experiments showed that, when compared to FedSGD, FedAvg is more accurate with less communication rounds and a lesser learning rate. Brendan et al. (2016) concluded that FedAvg is able to train high quality ML models in a few rounds of communication on a multi-layer perceptron, two convolutional neural networks, a two-layer character long short-term memory (LSTM) network and a large word-level LSTM.

A decentralised SGD method was applied in Xing et al. (2020), however FedAvg seems to outperform SGD.

Lee et al. (2023) propose a revised FL aggregating method, *ImprovedFedAvg*. They state that this method improves the model performance while reducing training time and the frequency of weight transmission from clients to the server, when compared to FedAvg, which was first proposed in Brendan et al. (2016). They provide pseudocode for both aggregation methods and comprehensively explain the improved algorithm. The results of the trained models show that the proposed method of aggregation outperforms FedAvg in terms of accuracy and f1-score. ImprovedFedAvg takes 6% less training time, as well as reducing 42% of the number of weights transferred to the aggregation server. The researchers concluded that the proposed method improves the model performance, while reducing training time and weight transmission frequency. They stated that future work would be to apply model tuning or expansion to the methods described in this paper, as well as using non-IID data. This research states that there has been a 42% reduction in transmission frequency, however it does not state that ImprovedFedAvg produces more accurate models with less transimssion frequency, as Brendan et al. (2016) states for FedAvg when compared to SGD. Moreover, it is shown that the accuracies and f1-scores between the methods are virtually the same.

2.5 Existing Research Similar to the Research Question

Liu et al. (2024) propose a delay and energy-efficient asynchronous FL framework for IDS (DEAFL-ID) in heterogeneous industrial IoT. Two identified limitations for this framework are that (1) there will be a large number of idle devices with limited resources. Using all the devices will be a waste of resources. (2) The FL IDS will suffer from long training times due to small IoT devices with limited computation capabilities and poor communication conditions. In this paper, the researchers use optimal device selection, data balance preprocessing and CNN model training. They aim to improve detection by using a hybrid sampling-assisted CNN model for IDS, which balances the dataset in addition to no noise and clear classification boundaries. They aim to reduce training cost by designing a resource utilisation efficiency function to explore the accuracy, delay reduction and energy saving in the DEAFL-ID. They used all the methods above to develop a deep Q-network based learning algorithm for the IDS. The researchers provided the energy cost in Joules and time cost of training. Liu et al. (2024) concluded that after implementing all methods, the DEAFL-ID scheme can significantly outperform benchmark schemes. This research uses the NSL-KDD dataset released in 2009, which is an improved version of the 1999 KDD dataset. This traffic is not representative of the current network threat landscape.

2.6 Research Gap

There is a research gap identified in the field of FL on supervised ML models such as SVM and Random Forest for IDS systems. Little to no work has been done on researching the power consumption of these frameworks in combination with these ML models. This information is crucial for devices where there may be a power constraint, such as IoT devices. On the other hand, ensuring an ML model and its training framework are as power efficient as can be can benefit the client devices, be them supercomputers, IoT devices, or mobile devices with batteries.

3 Methodology

3.1 Model Selection

This project consists of training two supervised machine learning models for binary classification.

- **Random Forest:** RF is one of the most powerful yet simple supervised machine learning algorithms for classification (Waskle et al. 2020). RF consists of an ensemble of a large number of decision trees, hence its name (Sruthi 2021). RF classifiers are quick, easy to use and versatile regardless of the data, being able to generate satisfactory results even before hyper-parameter tuning (Kumar & Dhanalakshmi 2023). This is particularly useful for this project, as the RF model will be trained on data from three different datasets, then tested for binary classification. The incomplex nature of the model will allow the RF to be implemented into FL quickly and produce results effectively.

- **Support Vector Machine:** SVM is a supervised machine learning algorithm that classifies data by finding the best hyperplane between all the data points of the classes (MathWorks 2024). SVM has a much better capacity to handle outliers, unlike RF which cannot deal with patterns that cannot be classified by normal learning (Bhoria & Garg 2013). For this project SVM was chosen as it performs best when the data has two classes (MathWorks 2024). This project conducts binary classification, therefore SVM is great fit for this implementation.

3.2 Datasets

In this project, three datasets were used: InSDN, CSE-CIC-IDS2018 and CIC-IDS2017. These datasets were acquired from their respective academic papers on the Internet, which is in accordance with the ethical approval for this project, found in Appendix A. Datasets from different sources were selected to more accurately portray the heterogeneous nature of computer networks.

- **InSDN (Elsayed et al. 2020):** InSDN is a dataset released in 2020 which covers a comprehensive range of attack traffic. The attack types are DoS, DDoS, Web Attacks (XSS and SQL Inject), R2L, Malware, Probe and U2R. The dataset consists of 86 features. This dataset emulates these attacks in a software defined network and it attempts to be as realistic as possible, with attacks from within the network, as well as external attacks. Elsayed et al. (2020) show in their research performance metrics for RF and SVM models, showing a 0.99 accuracy for both models. As this dataset was compiled in 2020, the attack traffic is much more representative of real attacks, compared to older datasets such as KDD99, making it a suitable dataset for this project. A limitation of this dataset is that it is largely unbalanced between the attack types.

- **CIC-IDS2017 (Sharafaldin et al. 2018):** CIC-IDS2017 is a dataset developed at the University of New Brunswick for Intrusion Detection. This dataset covers DoS, DDoS, Brute Force, XSS, SQL Injection, Infiltration, Port scan and Botnet attack traffic. This dataset consists of 83 features (Elsayed et al. 2020). The traffic was captured over the course of five days. The researchers tested the dataset on an RF model and they achieved 98% precision in 74.39 seconds, making RF the fastest model with highest accuracy in their research. This dataset represents a real heterogeneous network with diverse and modern attack types, making it a suitable dataset for this project.
- **CSE-CIC-IDS2018 (University of New Brunswick 2018), available *here*:** CSE-CIC-IDS2018 is a dataset created in collaboration with the Communications Security Establishment & the Canadian Institute for Cybersecurity. The dataset includes Brute-force, Heartbleed, Botnet, DoS, DDoS, Web attacks, and infiltration of the network from inside attack traffic. Heartbleed specifically is a newer type of attack. The dataset's data was collected over the course of ten days and consists of 83 features (Elsayed et al. 2020). As the dataset is conveniently split into multiple days, it is easier to create a training and testing dataset for training the supervised ML models in this project.

3.3 Data Processing

All datasets must be processed before being fed to the ML models to create usable and optimised training and testing datasets for all three FL client machines. The datasets discussed in Section 3.2 must each be pre-processed. This entails in the datasets being split, sliced and concatenated to create balanced training datasets. The datasets must be optimised in a way that suits RF and SVM models, ensuring maximum performance from both. The datasets have large ranges of continuous data, therefore standardisation with z-score normalisation must be done to ensure no model bias towards those features containing large values. The z-score normalisation value, x', of a value x, is calculated using the mean μ and the standard deviation σ, as seen in the equation below.

$$x' = \frac{x - \mu}{\sigma}$$

Oversampling using BSMOTE was done for the datasets that did not have enough samples to produce a balanced training dataset. Moreover, datasets that were too large to produce reasonably-sized (less than 350 MB) datasets were loaded in chunks and data samples were randomly picked. Should resulting training datasets contain an unbalanced ratio of benign to attack data, the larger class was reduced to match the smaller class, as the data samples allowed it. Testing datasets were compiled in the same way, without the emphasis on them being balanced, within reason (i.e. a ratio of benign to attack being 95:5 would be unreasonable).

3.4 Feature Selection

The CSE-CIC-IDS2018 and CIC-IDS2017 datasets have 83 features each, whereas InSDN has 86 features. This large number of features can ruin the RF and SVM models due to the curse of dimensionality. The curse of dimensionality states that the more numbers of features, the amount of data required to accurately perform classification grows exponentially. This can affect the accuracy and cause overfitting due to the noise caused by the unimportant features in the dataset. Moreover, the models can become unnecessarily complex, wasting computational resources and time (Awan 2023). As a result, feature importance must be done on the training and testing datasets resulting from the chosen datasets. Feature importance was done using the information gain of each feature calculated by the mutual information classifier from the Scikit-learn Python library. Moreover, this method of feature selection was chosen as an alternative to using an RF model for feature selection as to not introduce bias towards RF models.

3.5 Python & Libraries

Python was the most suitable language for this project due to the extensive ML, FL, data analysis and visualisation libraries. These libraries have comprehensive documentation, which allows the research to be conducted in a streamlined fashion. The libraries used in this project are as follows:

- **Scikit-learn:** Scikit-learn is a Python library that provides a plethora of ML models, such as classification, regression and clustering models. Moreover, it provides tools for data analysis such as dimensionality reduction and pre-processing. The library provides the tools in a simple and well-documented manner (Scikit-learn 2024). In this project, Scikit-learn was used for the RF and SVM classifiers, as well as z-score normalisation and feature selection. Moreover, an advantage of Scikit-learn is that it integrates well with all of the other Python libraries. Scikit-learn can produce classification reports after the classifiers have been trained and tested. These reports present the accuracy, precision, recall and f1-score metrics of the completed classifiers. The reports were used to compare the classifiers. The metrics in the classification report are calculated using the following formulae, where *TP = True positive; FP = False positive; TN = True negative; FN = False negative* (Miyasato 2020).
  - **Accuracy:**
  $$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$
  - **Precision:**
  $$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$
  - **Recall:**
  $$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$
  - **F1-Score:**
  $$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$
- **Pandas:** Pandas is a Python library that is used for data analysis and manipulation. It was used for analysing and processing the chosen datasets into testing and training datasets (Pandas 2018).
- **Matplotlib:** Matplotlib is a Python library that produces visualisations for data (Matplotlib 2012). It was used for producing pie charts of the datasets' labels to display the balanced or unbalanced nature of them, as well as confusion matrices for the ML models' testing performances.

- **NumPy:** NumPy is a Python library that adds aditional mathematical functionality to Python, for scientific computing (Numpy 2009). NumPy was used for removing infinite values in the chosen datasets.
- **Imbalanced-learn:** Imbalanced-learn is a library that is built upon the Scikit-learn library. It provides tools for handling imbalanced datasets for ML models such as re-sampling techniques (Imbalanced-learn 2024). It was used for re-sampling on unbalanced datasets produced in the data processing task of the research.
- **Flower:** Flower is a Python library that enables FL for the ML models. It allows for evaluation and analysis of the models by producing logs of the epochs and their accuracy, precision, recall and f1-score. Flower provides the FedAvg aggregation method for FL but they encourage users to implement their own aggregation models, should they require (Flower Labs 2024). Moreover, templates are available provided by Flower, or unaffiliated contributors, through GitHub, allowing for a quick start with the FL process. This project uses a template that can be found on GitHub *here*.

3.6 Federated Learning

Federated learning is a method of training ML models by having them be trained only on clients' local data, without the data ever leaving the local machine. The resultant model is shared with an aggregator server, which aggregates all the model weights received from all clients to produce a global model. The global model is shared with the clients and the process is repeated for the number of epochs set by the server. Federated learning keeps sensitive data private as it never has to be sent to a server (Rieke 2019). This type of FL is called centralised federated learning. Centralised federated learning was implemented in this project to portray the current market landscape and to mirror what tech companies such as Google use to train the ML models they use in their products (Buckley et al. 2023).

3.7 Energy Consumption

As Nakip et al. (2023) inquired, the energy consumption of federated ML models in an IDS is a subfield of particular interest in the research field. Collecting the energy consumption of the federated implementation can help researchers choose algorithms that would be suitable for devices where the energy overhead is limited, such as IoT devices. This project collects the energy consumption of the FL implementation on a per-client basis, as well as the emissions produced. The Python library used for collecting these metrics is CodeCarbon.

- **CodeCarbon (CodeCarbon 2021), available *here*:** CodeCarbon is an open-source Python library that tracks emissions based on the power consumption of running code and location-dependent carbon intensity. At the end of the tracking, CodeCarbon produces a spreadsheet of the results, which is useful for storing the results and analysing them. There were multiple energy consumption libraries available, such as pyJoules, pyRAPL and energy-usage, the latter of which was merged with CodeCarbon. A common theme across the libraries mentioned was the lack of recent updates. If support would have been needed with any of those libraries, it was certainly not guaranteed, resulting in a delay in the research. As a result, CodeCarbon was chosen, as the GitHub repository is still being worked on to this day.

4 Design

The solution was implemented on four virtual machines provided by the University of Warwick. The solution implements centralised federated learning where one virtual machine serves as the aggregation server and the rest are clients. The star network topology of the solution is presented in Figure 4.3. The datasets were assigned to the virtual machines as follows:

- InSDN to Client 1
- CSE-CIC-IDS2018 to Client 2
- CIC-IDS2017 to Client 3

The steps of the solution, as seen in Figure 4.2, were as follows:

1. The datasets were processed using Pandas, NumPy, Imbalanced-learn and visualised with Matplotlib, following the steps seen in Figure 4.1.
2. The RF and SVM models were trained using Scikit-learn. Each model was trained and tested using their virtual machine's locally assigned dataset.
3. If the accuracy produced was satisfactory, the model was not trained any further and the parameters were fitted to the federated model.
4. If the accuracy was not satisfactory, hyper-parameter tuning was done. If the accuracy was satisfactory after the tuning, the model was not trained any further and the parameters were fitted to the federated model.
5. If the accuracy was not satisfactory after tuning, the classification threshold of the models was adjusted. The threshold was fitted to the federated model.
6. Federated learning was done using Flower. The models were ran for five epochs each. CodeCarbon was used to track the energy consumption through the whole FL process.

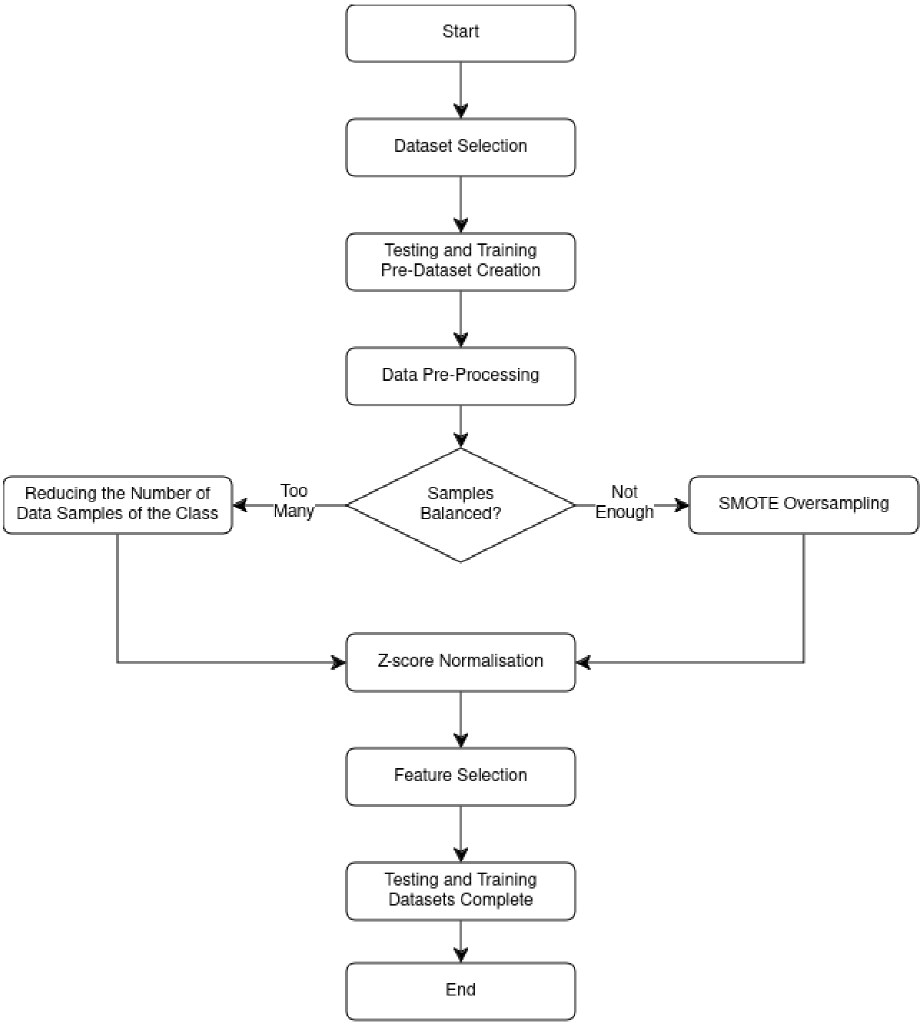

Figure 4.1: The Data Processing Flowchart

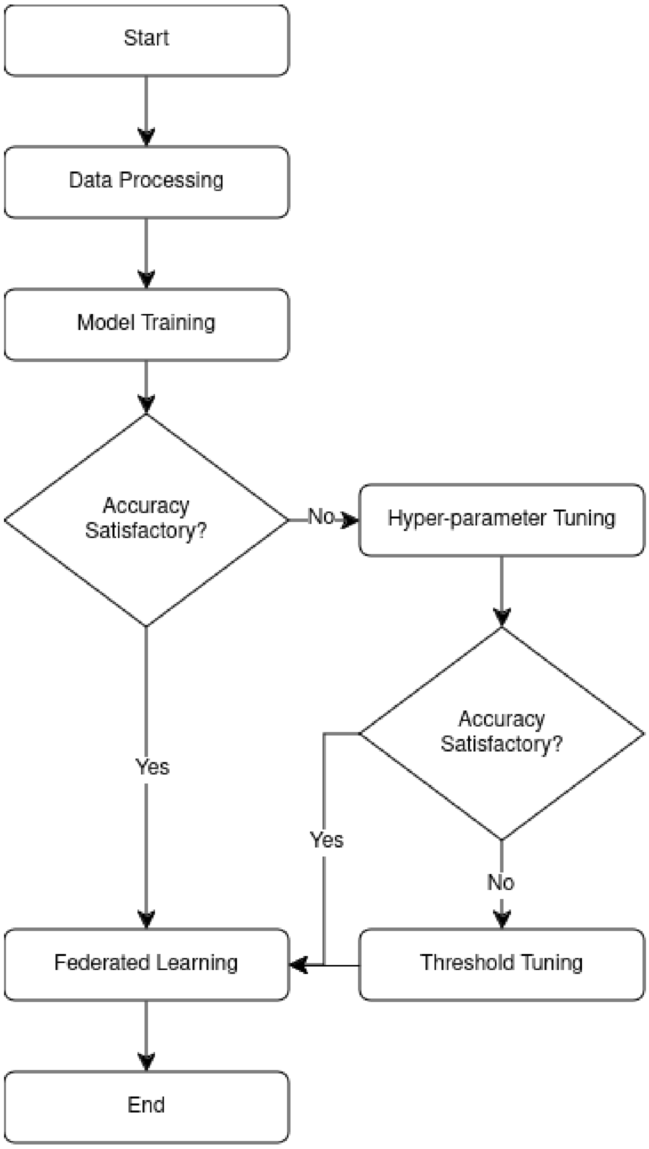

Figure 4.2: The Flowchart of the Solution

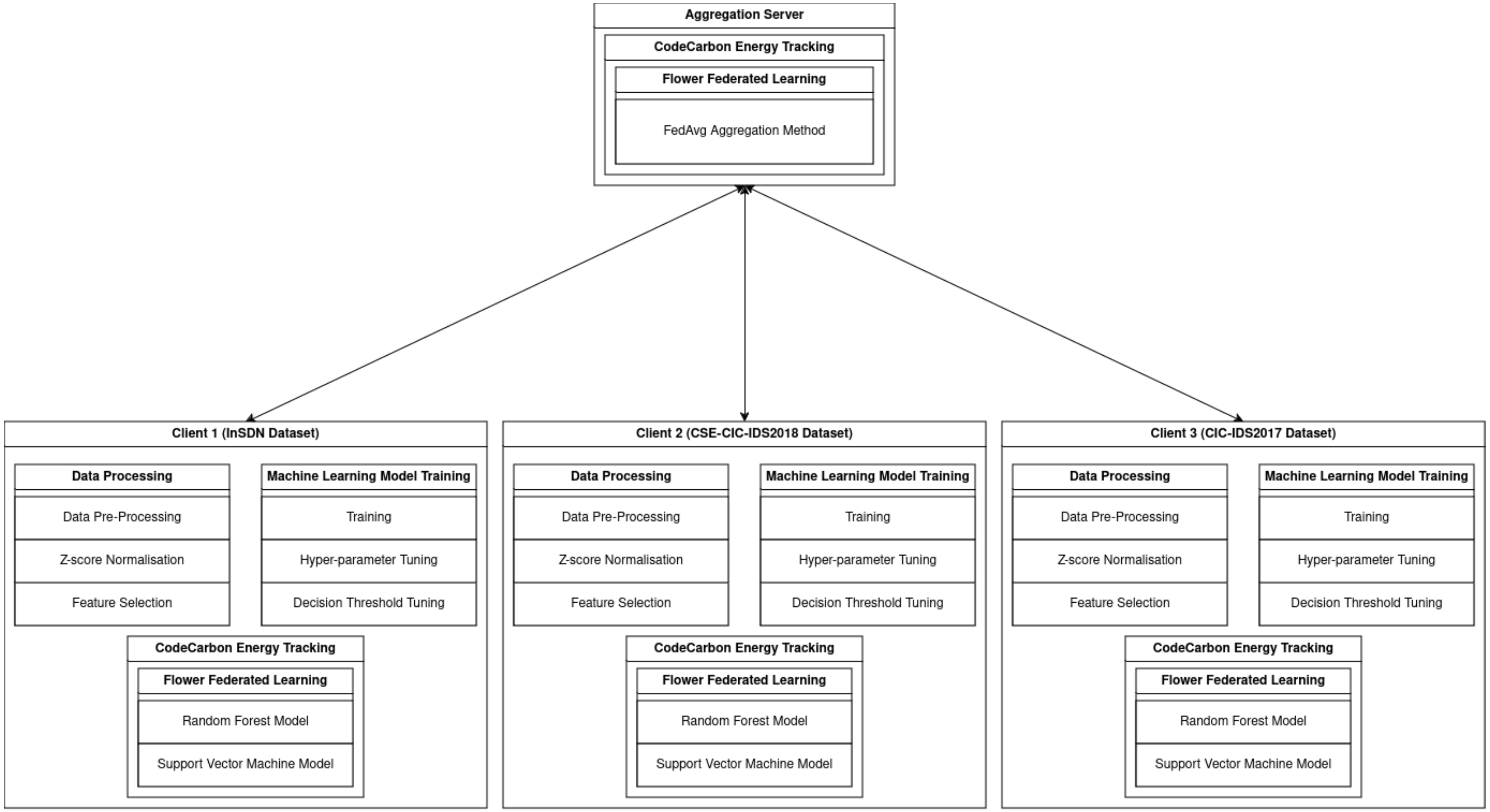

Figure 4.3: The Topology of the Solution

5 Implementation

The code for the whole project can be found in Appendix B and on GitHub *here*. The GitHub repository contains the .zip archive of the code, along with a README file explaining how to run it and what Python libraries are required to run it.

5.1 Data Processing

The data processing task was independently done for each client machine's assigned dataset. Although the task follows the general steps outlined in Fig. 4.1, some finer, per-dataset implementation details differ. This section outlines the general implementation of the data processing task with the fine implementation details shown on a per-dataset basis. The first subprocess in the data processing task, as seen in Fig. 4.1, is creating the training and testing pre-datasets. The pre-datasets are a result of the redistribution of traffic data, which is captured in multiple files in the original datasets. This concatenation of traffic data results into the training and testing pre-datasets, with the help of the Pandas library. The visualisations of the distribution of classes is done using Matplotlib.

- **InSDN:** The InSDN dataset is composed of three files. *Normal_data.csv*, which contains 68424 rows of benign, normal traffic. This equates to 20% of the total data records. Secondly, *OVS.csv* containing 136743 rows of attack traffic data, or 39.76% of the total data records. The attack traffic includes DoS (52471 rows), DDoS (48413 rows), Probe (36372 rows), brute force attack (1110 rows), web attack (192 rows) and botnet (164 rows). Thirdly, *metasploitable-2.csv* consists of 138772 rows of attack traffic data, or 40.34% of the total data records. It contains DoS (1145 rows), DDoS (73529 rows), probe (61757 rows), brute force attack (295 rows) and exploitation (R2L) (17 rows) attack data. The InSDN dataset was split into an 80:20 ratio for the training and testing datasets. Each of the three files were split by applying the 80:20 ratio to each of the classes of attack data present, rounded to the nearest whole. For example, in the OVS.csv file, the probe attack data was split into 20% of 36372 $\approx$ 7274 probe data samples for the testing dataset; and 80% of 36372 $\approx$ 29098 probe data samples for the training dataset. This resulted in an unbalanced distribution of samples, when seen in a binary classification of normal to attack data, as seen in Table 5.1.

- **CSE-CIC-IDS2018:** The CSE-CIC-IDS2018 dataset is split into ten separate files depicting the ten days where network traffic was captured. These ten files are very large, with one being 3.8 GB in size. One file, *02-21-2018.csv*, was saved for creating the testing dataset. For the rest nine files, a chunk of 20000 data samples from each file was loaded into a combined Pandas dataframe. This resulted in a rather balanced training dataset, as seen in Table 5.1. The testing dataset, *02-21-2018.csv* was a 313.7 MB file, almost being too large for Pandas to process into a dataframe in a reasonable amount of time. And yet, the whole file was transformed into the testing dataset with an unbalanced class distribution, as seen in Table 5.1.

- **CIC-IDS2017:** The CIC-IDS2017 machine learning dataset is split into eight separate files depicting the five days where network traffic was captured.
  The file *Wednesday-workingHours.pcap_ISCX.csv* was saved for the testing dataset. The days were processed in the same way as the CSE-CIC-IDS2018 dataset. This resulted in a massively unbalanced dataset with 1,833,066 benign samples, as seen in Table 5.1. The testing dataset, derived from *WednesdayworkingHours.pcap_ISCX.csv* requires no processing to become a pre-dataset, as it is adequate as-is for a testing dataset. The class distribution is shown in Table 5.1.

The second subprocess is data pre-processing. In this section, columns with categorical values or that are inconsequential to the ML models are dropped, null values are filled with the mean value of the feature they belong to, rows with infinite values are removed and the attack labels are turned into the binary classes of benign or attack data.

- **InSDN:** The *'Flow ID', 'Src IP', 'Src Port', 'Dst IP', 'Dst Port', 'Protocol', 'Timestamp'* features were dropped from the training and testing pre-datasets. These features do not include important information that the ML models can use for classification. They simply provide additional details about the traffic captured, therefore they were dropped. There were no null and infinite values found in the InSDN dataset. The next step was to turn the multi-class pre-datasets into binary class datasets. All the attack traffic, except *Normal* traffic, found under the feature *Label* was turned to 0. Normal traffic was turned to 1. This stays consistent across all datasets. However, this turned

Table 5.1: The Class Distribution for the Training and Testing Splits Pre-Binarisation

D.S.	Class	Training Split		Testing Split	
		Amount	%	Amount	%
InSDN	DDoS	97553	35.459	24389	35.461
	Probe	78504	28.535	19625	28.534
	Normal	54739	19.897	13685	19.898
	DoS	42893	15.591	10723	15.591
	BFA	1124	0.409	281	0.409
	Web-Attack	154	0.056	38	0.055
	BOTNET	131	0.048	33	0.048
	U2R	14	0.005	3	0.004
CSE-CIC-IDS2018	Benign	81327	45.182	360833	34.412
	DDoS attacks-HOIC	None	None	686012	65.423
	DDoS attacks-LOIC-UDP	None	None	1730	0.165
	DDoS attacks-LOIC-HTTP	19932	11.074	19625	None
	DoS attacks-GoldenEye	19930	11.072	13685	None
	FTP-BruteForce	19902	11.057	10723	None
	DoS attacks-SlowHTTPTest	19890	11.050	281	None
	Bot	18088	10.049	38	None
	Brute Force -Web	611	0.339	33	None
	Brute Force -XSS	230	0.128	3	None
	SQL Injection	87	0.048	3	None
CIC-IDS2017	Benign	1833066	85.736	440031	63.524
	DoS Hulk	None	None	231073	33.358
	DoS GoldenEye	None	None	10293	1.486
	DoS slowloris	None	None	5796	0.837
	DoS Slowhttptest	None	None	5499	0.794
	Heartbleed	None	None	11	0.002
	DDoS	128027	5.988	13685	None
	FTP-Patator	7938	0.371	10723	None
	SSH-Patator	5897	0.276	281	None
	Bot	1966	0.092	38	None
	Web Attack-Brute Force	1507	0.070	33	None
	Web Attack-XSS	652	0.030	3	None
	Infiltration	36	0.002	3	None
	Web Attack-Sql Injection	21	0.001	3	None

**D.S.: Dataset*

the pre-datasets into largely unbalanced datasets, as seen in Table 5.2.

- **CSE-CIC-IDS2018:** The same process was carried out for the CSE-CIC-IDS2018 dataset as for the InSDN dataset, dropping the *'Flow ID', 'Src IP', 'Src Port', 'Dst IP', 'Dst Port', 'Protocol'* and *'Timestamp'* features. The null values in the dataset were filled with the mean of the feature the value be-

Table 5.2: The Class Distribution for the Training and Testing Splits Post-Binarisation

Dataset	Class	Training Split		Testing Split	
		Amount	%	Amount	%
InSDN	Attack	220373	80.1	55092	80.1
	Normal	54739	19.9	13685	19.9
CSE-CIC-IDS2018	Attack	98670	55.1	687742	34.4
	Normal	80554	44.9	360833	65.6
CIC-IDS2017	Attack	304974	14.3	252672	36.5
	Normal	1833066	85.7	440031	63.5

Table 5.3: The Class Distribution after Re-sampling Training and Testing Splits

Dataset	Class	Training Split		Testing Split	
		Amount	%	Amount	%
InSDN	Attack	220373	50	55092	50
	Normal	220373	50	55092	50
CSE-CIC-IDS2018	Attack	98670	55.1	205804	65.5
	Normal	80554	44.9	108250	34.5
CIC-IDS2017	Attack	304833	48.0	251723	36.4
	Normal	329700	52.0	439683	63.6

longed to. Moreover, rows that had infinite values present were deleted. The binary classification modification was carried out with the same method as the InSDN dataset, resulting in the training and testing pre-dataset class distribution as seen in Table 5.2. As expected, the training dataset is balanced.

- **CIC-IDS2017:** The same process as the other two datasets was done for the CIC-IDS2017 pre-datasets. It is shown that the training pre-dataset is largely unbalanced, with a large majority of it consisting of benign traffic. The training and testing pre-dataset binary class distribution can be seen in Table 5.2.

The third subprocess was balancing the training datasets. It was crucial for the ML models to be trained on balanced data. This was to ensure that the classification of the attack types was as accurate as possible. Therefore the methods for reaching a balanced dataset used were B-SMOTE for oversampling, should the pre-datasets not have enough data samples of a type of class. For datasets that had too much data, meaning its file size was larger than 350 MB, the class containing the largest amount of samples was downsized to a fraction of its original size.

- **InSDN:** The InSDN datasets were largely unbalanced, having many more benign data samples than attack ones. InSDN did not have enough samples for attack data, therefore B-SMOTE was used to generate synthetic attack data samples using the Imbalanced-learn Python library. B-SMOTE uses a nearest neighbour technique to define the number of samples (k-neighbours) used for generating synthetic samples. The technique was used to generate samples for the minority class, using two k-neighbours and two m-neighbours. M-neighbours are the nearest neighbours that define if a minority sample is in "danger" (Imbalanced-learn 2024). This method was used for the training and testing pre-datasets to generate synthetic attack data. As a result, the datasets are now balanced, as seen in Table 5.3.

- **CSE-CIC-IDS2018:** The CSE-CIC-IDS2018 training pre-datasets had a suitable binary class distribution, therefore no re-sampling was necessary. The testing pre-dataset was unnecessarily large, therefore the dataset was reduced to a third of its samples for each binary class, as seen in Table 5.3.
- **CIC-IDS2017:** The CIC-IDS2017 training pre-datasets was largely unbalanced, with more than 85% of the data being benign traffic. Fortunately, there were enough total data samples, allowing for a fraction of the benign traffic to be dropped. This was done using the Pandas library to select a fraction of 0.82 of the benign data traffic to be dropped. This left a balanced testing pre-dataset, as seen in Table 5.3. The testing pre-dataset's binary class distribution was satisfactory, therefore no re-sampling was required.

The fourth subprocess was z-score normalisation. This type of normalisation places the data according to the mean of the feature. If a value is less than the mean, it will be negative. If more than the mean, it will be positive. If it is the mean, it will be zero. The values after the z-score normalisation effectively represent the standard deviation away from the mean.(Codecademy 2024, Google 2022). This normalisation was used on all pre-datasets.

The fifth subprocess was feature selection. The feature selection method used was gathering the information gain using the mutual information classifier from Scikit-learn. Information gain is calculated from the mutual information of features, which is the statistical dependence between two variables. Should the mutual information entropy be zero, the variables are independent. The larger the value, the higher the dependency. The lower the entropy, the higher the information gain (Brownlee 2019). Using this process, the top 25

Table 5.4: Top 25 Features Selected Based on Information Gain

InSDN	**CSE-CIC-IDS2018**	**CIC-IDS2017**
Pkt Len Max	Flow IAT Max	Average Packet Size
Pkt Len Std	Fwd Pkts/s	Packet Length Mean
Pkt Len Mean	Flow Duration	Packet Length Std
Pkt Size Avg	Flow Pkts/s	Packet Length Variance
Pkt Len Var	Flow IAT Mean	Total Length of Fwd Packets
Fwd Header Len	Init Fwd Win Byts	Subflow Fwd Bytes
Bwd Header Len	Pkt Len Max	Max Packet Length
Bwd Pkts/s	Init Bwd Win Byts	Subflow Bwd Bytes
Flow IAT Max	Pkt Len Mean	Total Length of Bwd Packets
Flow IAT Mean	Bwd Pkts/s	Bwd Packet Length Mean
Flow Pkts/s	Fwd Header Len	Avg Bwd Segment Size
Flow IAT Min	Pkt Len Std	Avg Fwd Segment Size
Bwd IAT Max	Pkt Len Var	Fwd Packet Length Mean
Bwd IAT Mean	Fwd Seg Size Avg	Init_Win_bytes_forward
Fwd Seg Size Avg	Fwd Pkt Len Mean	Bwd Packet Length Max
Bwd IAT Tot	Pkt Size Avg	Fwd Packet Length Max
Fwd Pkt Len Mean	Bwd Seg Size Avg	Init_Win_bytes_backward
Bwd Pkt Len Mean	Bwd Pkt Len Max	Flow Bytes/s
Bwd Seg Size Avg	Bwd Header Len	Fwd Header Length
TotLen Bwd Pkts	Subflow Fwd Byts	Fwd Header Length.1
TotLen Fwd Pkts	Bwd Pkt Len Mean	Bwd Header Length
Bwd Pkt Len Max	TotLen Fwd Pkts	Flow IAT Max
Subflow Bwd Byts	Subflow Bwd Byts	Flow Duration
Flow Duration	TotLen Bwd Pkts	Fwd IAT Total
Fwd Pkt Len Max	Fwd Pkt Len Max	Fwd IAT Max

features from each pre-dataset pair were selected, as seen in Table 5.4.

This concludes the data processing. All pre-datasets have become the training and testing datasets that were used for the ML models in and outside of FL, such as for hyper-parameter tuning.

5.2 Machine Learning

Initial training and testing was done for the RF and SVM models. Using the Scikit-learn library, the *RandomForestClassifier* and *SupportVectorClassifier* were imported. The training datasets were split into *X_training*, which contains all features but the *Label* feature. The *y_training* split contains only the *Label* feature, which contains the binary classification of the attack traffic. The models were fitted using the *self.fit()* method of each classifier, using *X_training* and *y_training* as parameters. The testing dataset is split in the same way, however the classifier uses the *X_test* set to make predictions on the data, using the *self.predict()* method, and *y_test* to verify

Table 5.5: Initial Classification Report Metrics for RF and SVM Models Per Class Per Dataset

D.S.	Model	Class	Accuracy	Precision	Recall	F1 Score
InSDN	**RF**	**Attack** **Normal**	0.99	0.99 0.99	0.99 0.99	0.99 0.99
	SVM	**Attack** **Normal**	0.99	0.99 1.00	0.99 1.00	0.99 0.99
CSE-CIC-IDS2018	**RF**	**Attack** **Normal**	0.34	0.00 0.34	0.00 1.00	0.00 0.51
	SVM	**Attack** **Normal**	0.67	1.00 0.51	0.50 1.00	0.67 0.68
CIC-IDS2017	**RF**	**Attack** **Normal**	0.64	0.95 0.64	0.01 1.00	0.02 0.78
	SVM	**Attack** **Normal**	0.71	0.99 0.64	0.41 0.99	0.58 0.78

**D.S.: Dataset*

its predictions, using the *accuracy_score()* function imported from Scikit-learn. The results of the testing are shown using the classification report provided by Scikit-learn, depicting the accuracy, precision, recall and f1-score of the classifier. Moreover, a confusion matrix was produced at the end of the training, showing the true positives and negatives, as well as the false positives and negatives. This shows the effectiveness of the classifier's training and how many attack and benign classes it correctly identifies.

The initial testing results of the classifiers' training are as follows, per processed dataset.

- **InSDN:** For the random forest model, the initial results shown are impressive, with a 0.99 accuracy, as seen in the classification report in Table 5.2. The training and testing took 22 seconds. The results are rounded to two decimal points. Moreover, the confusion matrix shows that the number of false positives or negatives were low, supporting that the model is very accurate even without any additional tuning, as seen in Fig. 5.1. For the SVM model, the results are similar to the RF model with an impressive accuracy of 0.99, as seen in Table 5.2. The training and testing took 6989 seconds. The same confusion matrix is just as impressive, with very little false predictions, as seen in Fig. 5.2.

- **CSE-CIC-IDS2018:** Using this dataset, the Random Forest model had an unexpected result, showing an accuracy of only 0.34, as seen in the classification report in Table 5.2. The

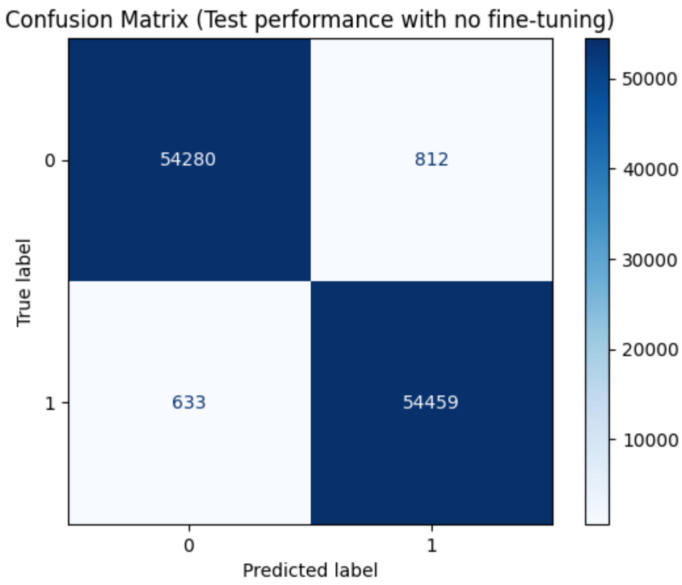

Figure 5.1: The Confusion Matrix of the Initial Random Forest Model Trained on the Processed InSDN Dataset

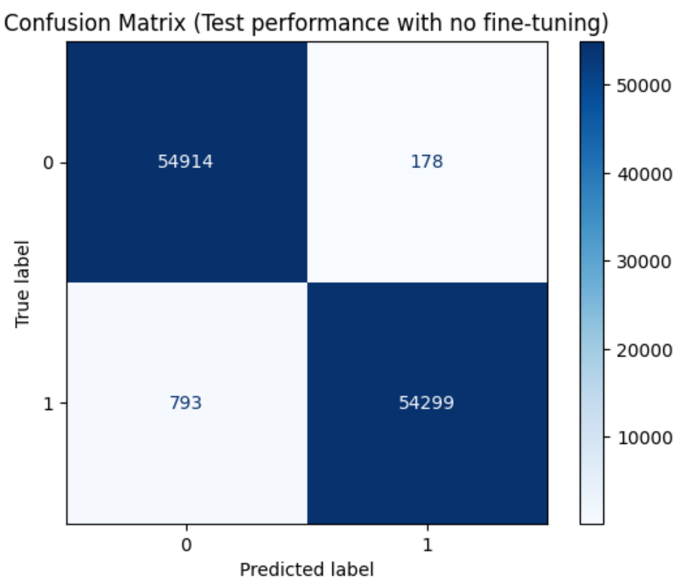

Figure 5.2: The Confusion Matrix of the Initial Support Vector Machine Model Trained on the Processed InSDN Dataset

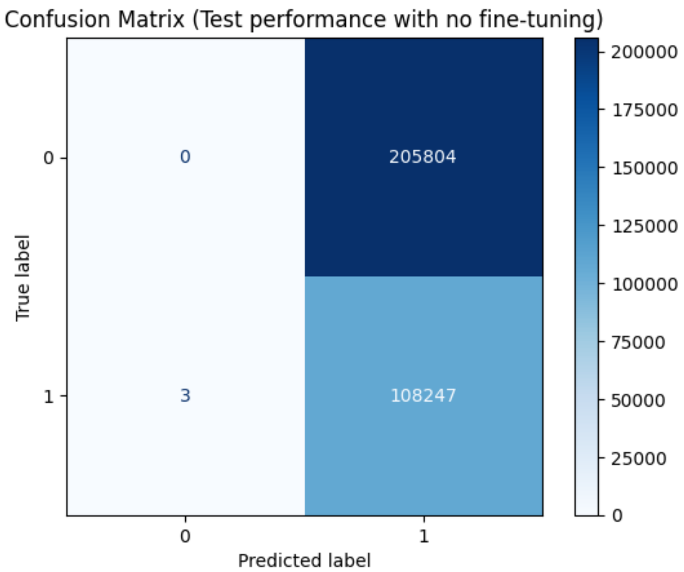

Figure 5.3: The Confusion Matrix of the Initial Random Forest Model Trained on the Processed CSE-CIC-IDS2018 Dataset

training and testing took 10 seconds. The confusion matrix shows that the RF model managed to get zero true negatives correct, although it got most of the true positives correct. This is seen in Fig. 5.3. The SVM model got an accuracy of 0.67, seen in Table 5.2, as it was able to correctly classify more true negatives. It correctly classified half of all true negatives, as seen in Fig. 5.4. The training and testing took 1134 seconds.

- **CIC-IDS2017:** The random forest model showed a low accuracy of 0.64, as seen in Table 5.2. This is due to the low amount of true negatives predicted, as the model incorrectly classified them as false negatives, as seen in Fig. 5.5. The training and testing took 41 seconds. The SVM model had a slightly better accuracy than the RF model, at 0.71, as seen in Table 5.2. This is due to correctly identifying more true negatives, as seen in the confusion matrix in Fig. 5.6. The training and testing took 10368 seconds.

Due to these results, it is clear that the models that did not achieve at least 0.80 accuracy must undergo further tuning before they can be introduced to federated learning.

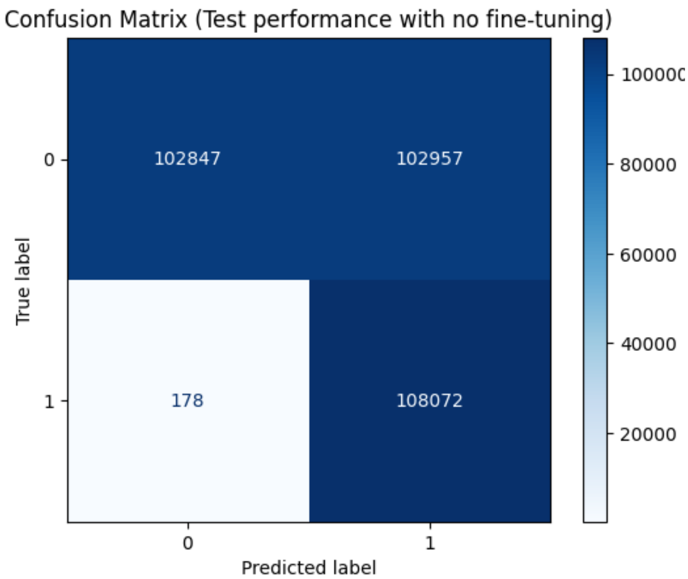

Figure 5.4: The Confusion Matrix of the Initial Support Vector Machine Model Trained on the Processed CSE-CIC-IDS2018 Dataset

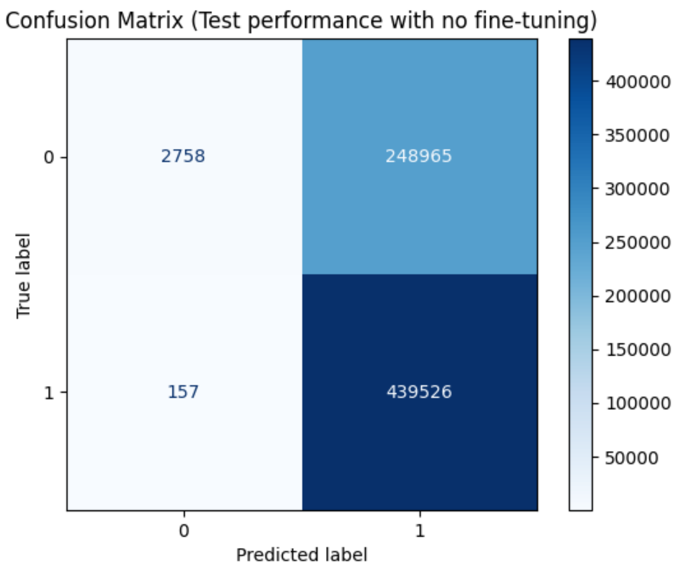

Figure 5.5: The Confusion Matrix of the Initial Random Forest Model Trained on the Processed CIC-IDS2017 Dataset

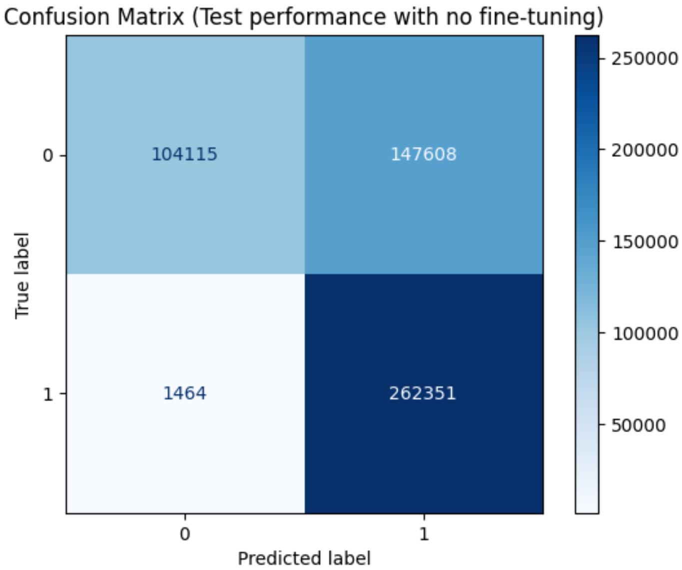

Figure 5.6: The Confusion Matrix of the Initial Support Vector Machine Model Trained on the Processed CIC-IDS2017 Dataset

5.3 Hyper-parameter Tuning

After the initial model results, the models needed further tuning. The Scikit-learn classifiers have modifiable parameters, hyper-parameters, that can be tweaked to further improve the accuracy. The hyper-parameters were tested using the Scikit-learn *GridSearchCV* cross-validation, using a 5-fold cross-validation strategy. Using the parameter grid, the *GridSearchCV* sets the current parameters. The cross-validation process begins by splitting the training dataset into five sets. One of these sets is used as a validation set against the rest of the sets to test for accuracy. This is repeated until all sets have been validation sets once. The average accuracy is obtained by calculating the mean from the cross-validation accuracy for the current parameters set. The next iteration would have different parameters and a different average accuracy. This is continued until all the parameter combinations in the grid are exhausted (Brownlee 2023). The resulting best parameters are the ones with the highest accuracy. For random forest, the parameters tuned were *'max_depth'*, *'max_features'*, *'min_samples_leaf'*, *'min_samples_split'* and *'n_estimators'*.

1. **max_depth:** Max_depth defines the maximum number of splits each decision tree in the random forest can make. A max_depth too low will cause the RF model to be trained less and cause it

to underfit. A max_depth too high can cause the model to be trained too much and cause it to overfit. A balance must be found to ensure that the RF model can accurately classify the data (GeeksforGeeks 2022, Saxena 2020).

2. **max_features:** Max_features defines the amount of features given to each decision tree in the random forest model. Too many features can cause the model to overfit, therefore it is a crucial hyper-parameter to tune (GeeksforGeeks 2022, Saxena 2020).

3. **min_samples_leaf:** Min_samples_leaf defines the minimum amount of samples that must be present in the subnode, or leaf, after splitting a node. A value too low can cause the model to overfit, where a value too high can cause the model to underfit. It is important to find a balance, therefore it must be tuned (GeeksforGeeks 2022, Saxena 2020).

4. **min_samples_split:** Min_samples_split tells the decision trees in the random forest the minimum required samples in a node to split it. If a node has more than two observations of a sample, it can be further split into subnodes. This parameter can prevent the model from overfitting by ensuring that the splits do not occur too quickly (GeeksforGeeks 2022, Saxena 2020).

5. **n_estimators:** N_estimators determines the amount of decision trees inside the random forest model. Increasing the number of estimators can increase complexity but it may not increase the accuracy of the model. A balance must be found to prevent a high computational complexity and cause the CPU to consume energy unnecessarily (GeeksforGeeks 2022, Saxena 2020).

As the SVM model took an unexpectedly large amount of time to train, even initially, the SVM models were not tuned further.

The random forest classifiers' results per dataset after hyper-parameter tuning, in terms of best parameters, classification reports and confusion matrices, were as follows.

- **InSDN:** The accuracy for the InSDN random forest classifier was satisfactory, therefore it did not need any tuning. The default parameters were passed to the federated learning library.

- **CSE-CIC-IDS2018:** The best parameters for the RF classifier were *'max_depth': 50, 'max_features': 'sqrt',*

Table 5.6: Classification Report Metrics for RF Model Per Class Per Dataset After Hyper-Parameter Tuning

D.S.	Model	Class	Accuracy	Precision	Recall	F1 Score
CCI2018	**RF**	**Attack** **Normal**	0.35	1.00 0.35	0.01 1.00	0.03 0.52
CI2017	**RF**	**Attack** **Normal**	0.64	0.85 0.64	0.01 1.00	0.02 0.78

**D.S.: Dataset; CCI2018: CSE-CIC-IDS2018; CI2017: CIC-IDS2017*

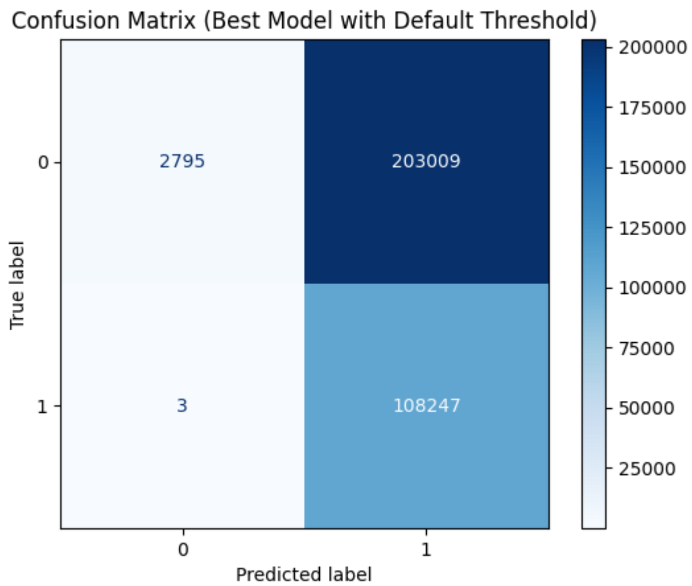

Figure 5.7: The Confusion Matrix of the Random Forest Model Trained on the Processed CSE-CIC-IDS2018 Dataset after Hyper-Parameter Tuning

'min_samples_leaf': 2, 'min_samples_split': 2, 'n_estimators': 500. The increase in accuracy was slight, to 0.35 from 0.34 as seen in Table 5.3. Compared to the 0 true negatives from the initial training, as seen in Fig. 5.3, the new RF classifier got a slight increase of predicted true negatives as well, as seen in Fig. 5.7.

- **CIC-IDS2017:** The best parameters for the RF classifier were *'max_depth': 50, 'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 2, 'n_estimators': 500.* This RF model did not see an increase in accuracy, rather it saw a decrease in benign data precision, as seen in Table 5.3. A visible change can be seen in the confusion matrix with the increase of true negatives but an increase of false positives as well, as seen in

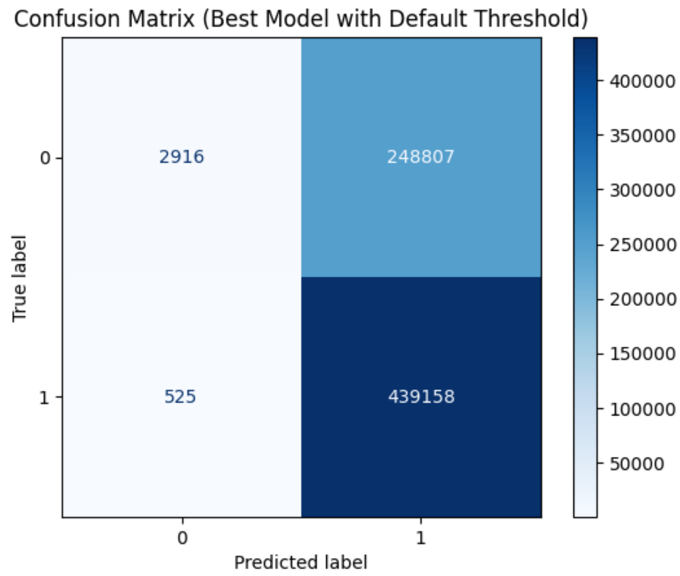

Figure 5.8: The Confusion Matrix of the Random Forest Model Trained on the Processed CIC-IDS2017 Dataset after Hyper-Parameter Tuning

Table 5.7: Classification Report Metrics for RF Model Per Class Per Dataset After Classification Threshold Tuning

D.S.	Model	Class	Accuracy	Precision	Recall	F1 Score
CCI2018	**RF**	**Attack**	0.69	0.99	0.54	0.70
		Normal		0.53	0.99	0.69
CI2017	**RF**	**Attack**	0.87	0.93	0.70	0.80
		Normal		0.85	0.97	0.91

**D.S.: Dataset; CCI2018: CSE-CIC-IDS2018; CI2017: CIC-IDS2017*

Fig. 5.8.

After hyper-parameter tuning, the CSE-CIC-IDS2018 and CIC-IDS2017 datasets RF models' accuracies did not significantly change, therefore they required further tuning.

5.4 Classification Threshold Tuning

The random forest classifiers were next tuned by changing the classification threshold using the Scikit-learn's random forest classifier *self.predict_proba()* method. The classification threshold is a boundary that is used to predict an object to a specific class. For example, the RF model determines a 0.75 probability that an object is benign, however the threshold for an object to be benign is 0.8, therefore the

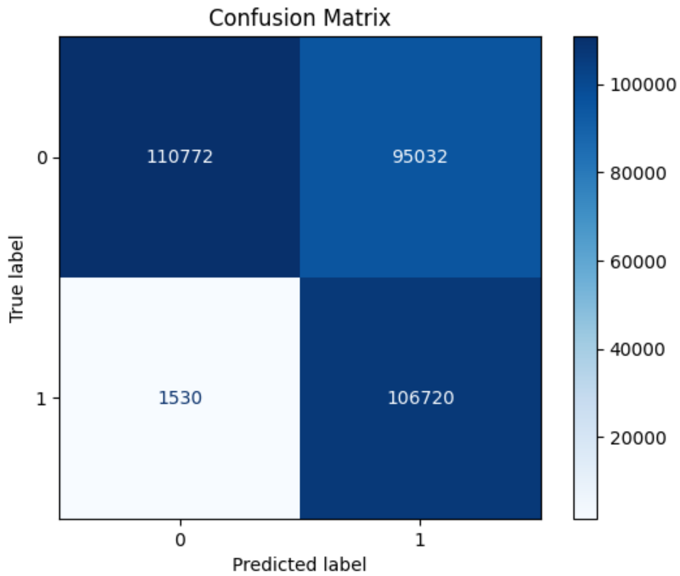

Figure 5.9: The Confusion Matrix of the Random Forest Model Trained on the Processed CSE-CIC-IDS2018 Dataset after Classification Threshold Tuning

probability must be higher than, or equal to 0.8 for it to be classified as benign (Evidently AI 2024). The threshold was tuned by taking all values between 0 and 1, to two decimal points, and predicting the classification of the data in the datasets. The best threshold is set and the current threshold being tested is compared to the best, in terms of f1-score. If the current threshold has a better f1-score than the best threshold, it becomes the best. This process is repeated until all values are exhausted. The random forest models' classification thresholds were tuned and significant increases in accuracies were recorded.

- **CSE-CIC-IDS2018:** The best threshold recorded for this RF classifier was 0.84. This threshold increased the accuracy of the classifier from 0.35 to 0.69, as seen in Table 5.3 and Table 5.4, respectively. As a result, many more true negatives were predicted, although there is still a large number of false negatives, as seen in Fig. 5.9.

- **CIC-IDS2017:** The best threshold recorded for this RF classifier was 0.96. This threshold significantly increased the accuracy of the classifier from 0.64 to 0.87, as seen in Table 5.3 and Table 5.4, respectively. This model benefited largely from the classification threshold tuning, significantly increasing the amount of true negatives and decreasing the number of false

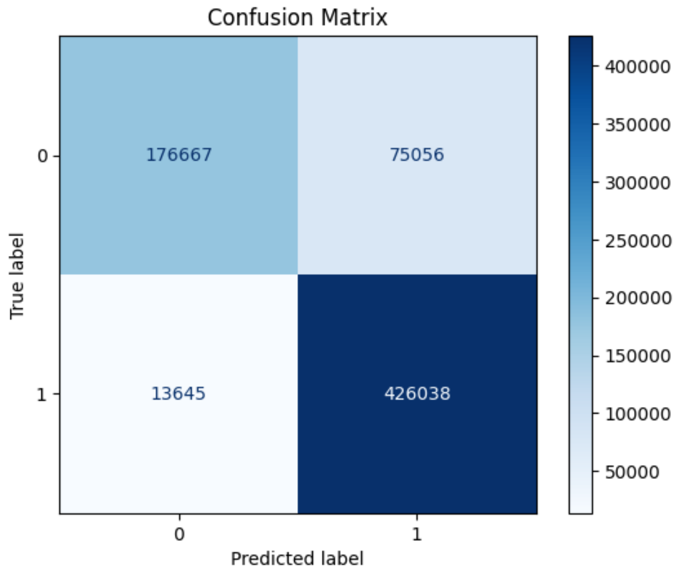

Figure 5.10: The Confusion Matrix of the Random Forest Model Trained on the Processed CIC-IDS2017 Dataset after Classification Threshold Tuning

negatives, although there was an increase in false positives as well, as seen in Fig. 5.10.

Both models that required further tuning received an increase in accuracy from the classification threshold change, which makes this a successful tuning. Now that the models are fully tuned, the thresholds and the hyper-parameters found were used in the federated learning.

5.5 Federated Learning

The Flower Python library was used to implement centralised federated learning, as the diagram in Fig. 4.3 shows. The four virtual machines were split into one aggregation server and three clients. The three clients were each assigned a processed dataset. The server collects the local models weights and uses the FedAvg method to aggregate them before setting them to the global model and distributing them to the clients again for a new epoch. The FedAvg pseudocode can be seen in Fig. 5.11, where $p_k = \frac{n_k}{n}$ is the weight of the k-th client (Zhou et al. 2020).

The Flower library integrates with Scikit-learn well, allowing the usage of the classifiers to be done without much modification. The datasets are processed into X and y sets the same way as the *X_training* and *y_training* splits in Section 5.2. The tuned hyper-

Algorithm 1 `FederatedAveraging`. In the cluster there are N clients in total, each with a learning rate of η. The set containing all clients is denoted as S, the communication interval is denoted as E, and the fraction of clients is denoted as C

Central server do:

Initialization: global model w_0.
for each global iteration $t \in 1, ..., iteration$ **do**
    # Determine the number of participated clients.
    $m \leftarrow max(C \cdot N, 1)$
    # Randomly choose participated clients.
    $S_p = random.choice(S, m)$
    for all each client $k \in S_p$ **do in parallel**
        # Get clients improved model.
        $w_{t+1}^k \leftarrow TrainLocally(k, w_t)$
    end for
    # Update the global model.
    $w_{t+1} \leftarrow \sum_{k=0}^{N} p_k w_{t+1}^k$
end for

TrainLocally(k, w_0):

for each client iteration $e \in 1, ..., E$ **do**
    # Do local model training.
    $w_e \leftarrow w_{e-1} - \eta \nabla F(w_{e-1})$
end for
return w_E

Figure 5.11: FedAvg Pseudocode (Zhou et al. 2020)

parameters of the RF classifiers and the default parameters SVM classifiers can be set as they are set with Scikit-learn, by setting them when calling the classifier function.

Before the output of the loss and classification report, the federated model tunes the classification threshold again to accommodate the new model. The classification report results can be seen in Section 6 for the individual clients and the aggregated global model.

5.6 Energy Consumption

During the runtime of the federated learning, CodeCarbon was tracking the energy consumption of the individual machines involved with the FL. A large upside of this library is its ease of use. CodeCarbon can track the whole code by initialising the *EmissionsTracker()* at the top of the code, starting it and stopping it at the end of the code. CodeCarbon implements timestamps and allows for a customised project name, making it convenient in terms of compiling results. Unfortunately, as these computers are virtual machines, the model of the processing unit is not known. Therefore CodeCarbon uses an estimated power draw of 42.5 watts. The results of the energy consumed can be seen in Section 6.

Table 6.1: Total Time Taken for the Models' Federated Learning

Model	Total Epochs	Time Taken (Seconds)
Random Forest	5	4756
Support Vector Machine	5	120145

Table 6.2: Results of the Aggregated Models

Epoch	Model	Loss	Accuracy	Precision	Recall	F1-Score
1	Random Forest	5.620339510666166	0.844069	0.887465	0.844069	0.841237
	Support Vector Machine	6.741053851047911	0.812975	0.86093	0.812975	0.813208
2	Random Forest	5.798580176490219	0.839123	0.884114	0.839123	0.835891
	Support Vector Machine	6.741053851047911	0.812975	0.86093	0.812975	0.813208
3	Random Forest	5.563381535785546	0.845649	0.888971	0.845649	0.842979
	Support Vector Machine	6.741053851047911	0.812975	0.86093	0.812975	0.813208
4	Random Forest	5.648640986641376	0.843283	0.881992	0.843283	0.84101
	Support Vector Machine	6.741053851047911	0.812975	0.86093	0.812975	0.813208
5	Random Forest	5.731541871859769	0.840983	0.884683	0.840983	0.838019
	Support Vector Machine	6.741053851047911	0.812975	0.86093	0.812975	0.813208

6 Results

The results of the federated learning and the energy consumption are shown here. The results are compiled based on the machine, as the client machines consist of a federated RF classifier instance and a federated SVM classifier instance, each with its respective energy consumption data.

6.1 Federated Learning

The federated learning results are as follows. All results mentioned in the paragraphs below are rounded to two decimal points.

- **Server:** The server consists of the aggregated RF and SVM ML models for five epochs. The total time taken for the federated learning of the Random Forest model is 4756 seconds, whereas the time for SVM is 120145 seconds, as seen in Table 6.1. The classification report for each of the ML models was generated and RF scored higher than SVM in all categories, them being accuracy, precision, recall and f1-score. RF achieved an accuracy of 0.84 whereas SVM achieved an accuracy of 0.81. Moreover, RF achieved a score for precision, recall and f1-score of 0.89, 0.84 and 0.81 respectively. On the other hand, SVM achieved 0.81, 0.86, 0.81 and 0.81, respectively. Additionally, the difference in performance can also be seen through the loss score, which is the difference between the

Table 6.3: Results of the Local ML Models Trained on Client 1

Epoch	Model	Accuracy	Precision	Recall	F1-Score
1	Random Forest	0.98765701	0.98769946	0.98765701	0.98765674
	Support Vector Machine	0.90809918	0.91124181	0.90809918	0.90792327
2	Random Forest	0.98771146	0.98775492	0.98771146	0.98771119
	Support Vector Machine	0.90809918	0.91124181	0.90809918	0.90792327
3	Random Forest	0.98768424	0.98773232	0.98768424	0.98768393
	Support Vector Machine	0.90809918	0.91124181	0.90809918	0.90792327
4	Random Forest	0.98771146	0.98775902	0.98771146	0.98771116
	Support Vector Machine	0.90792327	0.91124181	0.90809918	0.90792327
5	Random Forest	0.98782037	0.98787114	0.98782037	0.98782006
	Support Vector Machine	0.90809918	0.91124181	0.90809918	0.90792327

Table 6.4: Results of the Local ML Models Trained on Client 2

Epoch	Model	Accuracy	Precision	Recall	F1-Score
1	Random Forest	0.71876174	0.84493792	0.71876174	0.72110416
	Support Vector Machine	0.67160106	0.83070213	0.67160106	0.66981250
2	Random Forest	0.71171200	0.84269785	0.71171200	0.71359745
	Support Vector Machine	0.67160106	0.83070213	0.67160106	0.66981250
3	Random Forest	0.71798799	0.84470307	0.71798799	0.72028167
	Support Vector Machine	0.67160106	0.83070213	0.67160106	0.66981250
4	Random Forest	0.72267190	0.84005388	0.72267190	0.72573956
	Support Vector Machine	0.67160106	0.83070213	0.67160106	0.66981250
5	Random Forest	0.71470830	0.84318998	0.71470830	0.71683359
	Support Vector Machine	0.67160106	0.83070213	0.67160106	0.66981250

predicted values against the actual values. RF has a lower loss with 5.62 whereas SVM has a loss of 6.74. These results can all be seen in Table 6.2. The trend of RF outperforming SVM in all metrics continues on all of the clients, as seen in Table 6.3, 6.4 and 6.5 and as discussed below.

- **Client 1:** Client 1 consists of the local RF and SVM models trained on the processed InSDN dataset. The results can be seen in Table 6.3. RF outperforms SVM in terms of accuracy, precision, recall and f1-score. The maximum accuracy of the RF ML model for the InSDN dataset was 0.99, whereas the maximum accuracy for SVM was 0.91. The RF model took 4721 seconds to train the five epochs, whereas the SVM model took 120129 seconds.

- **Client 2:** Client 2 consists of the local RF and SVM models trained on the processed CSE-CIC-IDS2018 datasets. The results can be seen in Table 6.4, in terms of accuracy, precision, recall and f1-score. However, SVM is almost on par with

Table 6.5: Results of the Local ML Models Trained on Client 3

Epoch	Model	Accuracy	Precision	Recall	F1-Score
1	Random Forest	0.87810346	0.89080893	0.87810346	0.87247103
	Support Vector Machine	0.86203186	0.86664183	0.86203186	0.86324736
2	Random Forest	0.87331756	0.88640934	0.87331756	0.86724500
	Support Vector Machine	0.86203186	0.86664183	0.86203186	0.86324736
3	Random Forest	0.88100045	0.89333938	0.88100045	0.87565024
	Support Vector Machine	0.86203186	0.86664183	0.86203186	0.86324736
4	Random Forest	0.87505171	0.88418634	0.87505171	0.86998951
	Support Vector Machine	0.86203186	0.86664183	0.86203186	0.86324736
5	Random Forest	0.87494034	0.88708592	0.87494034	0.86919213
	Support Vector Machine	0.86203186	0.86664183	0.86203186	0.86324736

Table 6.6: The Energy Consumption of Federated Learning

Model	Environment	Time (Seconds)	Emissions (CO2eq in kg)	CPU Energy (kWh)	RAM Energy (kWh)	Total Energy (kWh)
RF	Server	4756	0.0157	0.0561	0.00384	0.06
	Client 1	4790	0.0158	0.0566	0.00387	0.06
	Client 2	4789	0.0158	0.0565	0.00387	0.06
	Client 3	4787	0.0158	0.0565	0.00386	0.06
SVM	Server	120145	0.396	1.418	0.097	1.52
	Client 1	120129	0.396	1.418	0.097	1.52
	Client 2	120125	0.396	1.418	0.097	1.52
	Client 3	120125	0.396	1.418	0.097	1.52

its precision at 0.83, whereas RF has a precision of 0.84. The maximum accuracy of the RF ML model for the CSE-CIC-IDS2018 dataset was 0.72, whereas the maximum accuracy for SVM was 0.67. The RF model took 4720 seconds, whereas the SVM model took 120125 seconds.

- **Client 3:** Client 3 consists of the local RF and SVM models trained on the processed CIC-IDS2017 datasets. The results can be seen in Table 6.5, in terms of accuracy, precision, recall and f1-score. These results were the closest out of all other clients' ML models, having a maximum difference of 0.02 between them. For example, for the first epoch the RF achieved 0.88, 0.89, 0.88 and 0.87, respectively. SVM achieved 0.86, 0.87, 0.86 and 0.86, respectively. The maximum accuracy of the RF ML model for the CIC-IDS2017 dataset was 0.88, whereas the maximum accuracy for SVM was 0.86. The RF model took 4718 seconds, whereas the SVM model took 120125 seconds.

6.2 Energy Consumption

The energy consumption is shown in Table 6.6, for all machines, models and environments. The values discussed in the paragraphs commencing were rounded to three significant figures. The CPU power remained at a constant 42.5 watts, as well as the RAM power, at 2.91 watts. Due to this, the CPU energy consumption and RAM energy consumption remained at a constant as well. For the federated RF cluster this was a mean of 0.0564 *kilowatt-hours* per machine and a total CPU energy consumption of 0.226 kWh. Moreover, the RAM energy consumption per machine was a mean of 0.00386 kWh and a total consumption of 0.0154 kWh.

On the other hand, the CPU energy consumption for the federated SVM cluster had a mean of 1.418 kWh per machine and a total CPU energy consumption of 5.67 kWh. Moreover, the RAM energy consumption per machine was a mean of 0.097 kWh and a total consumption of 0.388.

In Table 6.6, the total energy consumption is the sum of the CPU energy consumption and the RAM energy consumption.

It can be seen that the federated RF ML model cluster consumed less energy, produced less emissions and took less time to complete FL than the federated SVM cluster. According to the server which controls the FL, the RF model completed the full FL in 4756 seconds. The computation on the server side of the federated RF generated 0.0157 *carbon dioxide equivalent in kilograms* and consumed 0.06 kWh of energy. However, that is only the energy consumed for one machine, the server. The total energy consumed by the federated RF cluster is $0.06 \times 4 = 0.24$ kWh and the total emissions generated are $0.0157 \times 4 = 0.0628$ CO2eq in kg.

On the other hand, the federated SVM ML model cluster completed in 120145 seconds and generated 0.396 CO2eq in kg per machine and consumed 1.52 kWh of energy per machine. This accumulates to $0.396 \times 4 = 1.584$ CO2eq in kg of emissions generated and $1.52 \times 4 = 6.08$ kWh of energy consumed for the whole federated SVM ML model cluster.

7 Discussion

Table 6.2 shows the results of the models for each epoch. It can be seen that the RF cluster consistently has a higher accuracy and less loss than the SVM models, although it can be seen that the accuracy does not always increase for the RF model. Moreover, the SVM cluster's accuracy remains the same for each epoch. This is discussed in the Sections 7.1.3 and 7.1.1, respectively. This pattern can also be recognised in the results of the local ML models for each client, in Tables 6.3, 6.4 and 6.5. Moreover, RF had a higher mean federated accuracy than SVM, it being 84%, compared to SVM's 81%.

As seen in Table 6.1, the times taken for the models to complete federated learning was significantly different. The SVM cluster took approximately 2526%, or approximately 25 times, longer than the RF cluster. This has a cumulative effect on the energy consumption of the SVM cluster when compared to the RF cluster, as seen in Table 6.6.

Table 6.6 shows the total energy consumed, in kilowatt-hours, for the RF and SVM clusters. It is shown that all machines consumed the same amount of energy as the other machines in their respective clusters. The energy consumed was 0.06 kWh for RF and 1.52 for SVM, per machine. SVM had a 25 times higher consumption of energy than RF. This result is proportionate with the time taken, as shown in the previous paragraph. In total, the RF cluster consumed 0.24 kWh in approximately 1.32 hours whereas the SVM cluster consumed 6.08 kWh in approximately 33.37 hours. For comparison, on average, a fridge-freezer would consume 1 kWh in 26 hours, a tumble dryer would consume 4.5 kWh for a single cycle, and an electric oven would consume 2 kWh for 30 minutes of use (Ofgem 2024). At the average price of the pence per kWh in the UK, which is 28.62, SVM would cost 1 pound and 74 pence to run five epochs of the federated learning with the parameters in this project. RF would cost 37.77 pence (Ukpanah 2024).

This research aims to investigate the influence on the energy consumption when using different federated ML models for binary classification in an IDS. The results in Section 6 show that when using these two different ML models, RF and SVM, the model chosen for binary classification in an IDS using federated learning has a large influence on the energy consumption. As seen in Table 6.6, the federated RF ML model cluster is shown to consume 0.06 kWh per machine, which is a total of 0.24 kWh for the whole cluster. The federated SVM cluster is shown in Table 6.6 to consume 1.52 kWh per machine, which is a total of 6.08 kWh for the cluster. Therefore, the federated RF ML model cluster consumes 25 times less energy than the federated SVM cluster. Although the federated SVM cluster consumed more energy than the federated RF cluster, the SVM cluster only achieved an average accuracy of 0.81, as discussed in Section 6.1 and seen in Table 6.2. On the other hand, the federated RF cluster consumed less energy than the SVM cluster and it achieved an average accuracy of 0.84, higher than the federated SVM cluster. This reveals that when trained using federated learning, the supervised RF ML model can more accurately carry out binary classification on network traffic for an IDS while consuming less energy to train, making it **the most accurate and energy-efficient supervised ML model** between it and the federated SVM model. This makes the federated RF ML model more suited for an energy-limited IDS than the federated SVM ML model. An example is for an energy-limited IDS implementation where a network of IoT devices must participate in federated learning for the IDS, but they have a limited energy overhead due to the nature of the IoT devices on the network, or the energy limit may be imposed by the owner. Therefore such a device may not be able to have access to 6.08 kWh of energy to train a federated SVM model cluster, but it may have access to 0.24 kWh of energy to train the federated RF cluster.

7.1 Limitations

7.1.1 Hyper-Parameters

The results of the SVM models are the same in every epoch, for every client. This is due to not using hyper-parameters for the SVM models, therefore no hyper-parameters were passed to the FL clusters. As a result, the FedAvg method had no values to aggregate, which resulted in the parameters being the same in every epoch, as well as the results.

A single training for the SVM model took more than 5000 seconds, as discussed in Section 5, therefore a *GridSearchCV* would require that time taken for each combination of the parameters. For example, should a grid have 6 values, the time taken to tune the hyper-parameters would be approximately $6! \times 5000 = 3,600,000$ seconds, or 1000 hours.

7.1.2 Energy Consumption Metrics

All machines took the approximately the same amount of time to finish the federated learning as the server. This shows that there were moments where the client machines sat idle, waiting for the other machines to finish training their local model. This contributed largely to the energy consumption metrics, as CodeCarbon assumes that the clients remain in constant 100% usage.

7.1.3 Amount of Epochs

The results of each epoch in this project shows that the FL global model may not necessarily give increasingly positive results on the clients' local ML models trained on the local datasets. A larger amount of epochs may show an upward trend beginning to form, which is not possible to accurately predict with only five epochs.

7.1.4 Machines

Ideally, machine learning models would be trained using equipment that can parallelise the workload for as many threads as possible. For example, a 64 core processing unit with two threads per core, making 128 threads. This would allow for 128 instances of the model training, reducing the time for training significantly. However, this project only had 8 threads available. Moreover, some of the original dataset files were too large for the 8 gigabytes of memory assigned to each virtual machine, frequently crashing the machine in the early steps of data processing.

8 Conclusion

This project conducted the investigation on what the influence on the energy consumption is when using different supervised machine learning models trained using federated learning for the binary classification of network traffic in intrusion detection systems.

This project achieved the aim of evaluating the currently available research in the field of federated learning frameworks for privacy-preserving network intrusion detection systems. Moreover, it successfully assessed the currently available aggregation methods for federated learning.

The project completed the analysis of the InSDN, CSE-CIC-IDS2018 and CIC-IDS2017 datasets, processing them for the training of the RF and SVM ML models. Moreover, the project successfully trained and validated the RF and SVM ML models using these datasets for binary classification in an IDS.

The project successfully completed the validation and evaluation of the RF and SVM ML models using centralised federated learning. Moreover, the project identified the most accurate and energy-efficient federated supervised ML model that is suited for an energy-limited IDS. It was shown that, when using the InSDN, CSE-CIC-IDS2018 and CIC-IDS2017 datasets, the Random Forest federated learning cluster of models is more accurate (84%) and more energy efficient (0.24 kWh) than the Support Vector Machine (81%) federated learning cluster of models (6.08 kWh), making the RF cluster more suited for an energy-limited IDS, such as an IDS for the industrial IoT.

This project showed the comparison of the energy consumption of two different supervised ML models, RF and SVM, for binary classification in an IDS, providing metrics in kWh, allowing future researchers to make an informed choice on what ML model to implement for an energy-limited privacy-preserving network IDS.

In summary, this project showed the large disparity in energy efficiency between the SVM and RF ML models, effectively placing the RF model in a different class of energy efficiency when compared to an SVM ML model trained using federated learning for binary classification in an IDS.

9 Further Work

In the future, the energy consumption would be measured for unsupervised models, such as CNNs and LSTMs. These types of deep learning models are becoming more widely used for IDS applications and its research. Specifically, measuring the energy consumption for these models would be particularly useful for those developing deep learning models for IDSs in IoT devices, such as the industrial IoT, due to the limited power and computational overhead.

Moreover, the energy consumption of a federated framework that takes advantage of efficient methods should be measured. For example energy, computational and communication efficient methods, such as implementing the ImprovedFedAvg aggregation method instead of the FedAvg aggregation method.

In the future, a decentralised federated learning version of this project should be implemented to further improve the privacy preserving nature of federated learning and remove the server-dependent architecture that implements a single point of failure, putting availability at risk.

References

A, P. A., Maryposonia, A. & S, P. (2023), 'An efficient network intrusion detection system for distributed networks using machine learning technique'.

Agrawal, S., Sarkar, S., Aouedi, O., Yenduri, G., Piamrat, K., Alazab, M., Bhattacharya, S., Maddikunta, P. K. R. & Gadekallu, T. R. (2022), 'Federated learning for intrusion detection system: Concepts, challenges and future directions', *Computer Communications* .

Assis, F. & Hessel, F. (2022), 'Decentralized federated learning for intrusion detection in iot-based systems: A review'.

Aung, Y. Y. & Min, M. M. (2017), 'An analysis of random forest algorithm based network intrusion detection system', *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing* .

Awan, A. A. (2023), 'The curse of dimensionality in machine learning: Challenges, impacts, and solutions'.
URL: *https://www.datacamp.com/blog/curse-of-dimensionality-machine-learning*

Bhoria, P. & Garg, K. (2013), 'An imperial learning of data mining classification algorithms in intrusion detection dataset'.
URL: *https://www.ijser.org/researchpaper/An-Imperial-learning-of-Data-Mining-Classification-Algorithms-in-Intrusion-Detection-Dataset.pdf*

Brendan, M. H., Moore, E., Ramage, D., Hampson, S. & y Arcas, B. A. (2016), 'Communication-efficient learning of deep networks from decentralized data', *arXiv (Cornell University)* .

Brownlee, J. (2019), 'Information gain and mutual information for machine learning'.
URL: *https://machinelearningmastery.com/information-gain-and-mutual-information/*

Brownlee, J. (2023), 'A gentle introduction to k-fold cross-validation'.
URL: *https://machinelearningmastery.com/k-fold-cross-validation/*

Buckley, D., Darais, D., Near, J. & Lefkovitz, N. (2023), 'The uk-us blog series on privacy-preserving federated learning: Introduction – responsible technology adoption unit blog'.
URL: *https://rtau.blog.gov.uk/2023/12/07/the-uk-us-blog-series-on-privacy-preserving-federated-learning-introduction/*

Codecademy (2024), 'Normalization'.
URL: *https://www.codecademy.com/article/normalization*

CodeCarbon (2021), 'Codecarbon.io'.
URL: *https://codecarbon.io/*

Elsayed, M. S., Le-Khac, N.-A. & Jurcut, A. D. (2020), 'Insdn: A novel sdn intrusion dataset', *IEEE Access* **8**, 165263–165284.

Evidently AI (2024), 'How to use classification threshold to balance precision and recall'.
URL: *https://www.evidentlyai.com/classification-metrics/classification-threshold*

Flower Labs (2024), 'Flower: A friendly federated learning framework'.
URL: *https://flower.ai/*

GeeksforGeeks (2022), 'Random forest hyperparameter tuning in python'.
URL: *https://www.geeksforgeeks.org/random-forest-hyperparameter-tuning-in-python/*

Google (2022), 'Normalization'.
URL: *https://developers.google.com/machine-learning/data-prep/transform/normalization*

Gupta, R. & Alam, T. (2022), 'Survey on federated-learning approaches in distributed environment', *Wireless Personal Communications* .

Hosseinalipour, S., Brinton, C. G., Aggarwal, V., Dai, H. & Chiang, M. (2020), 'From federated to fog learning: Distributed machine learning over heterogeneous wireless networks', *IEEE Communications Magazine* **58**, 41–47.

Imbalanced-learn (2024), 'imbalanced-learn documentation — version 0.8.0'.
URL: *https://imbalanced-learn.org/*

Khraisat, A., Gondal, I., Vamplew, P. & Kamruzzaman, J. (2019), 'Survey of intrusion detection systems: techniques, datasets and challenges', *Cybersecurity* **2**, 1–22.
URL: *https://cybersecurity.springeropen.com/articles/10.1186/s42400-019-0038-7*

Kumar, M. R. & Malathi, K. (2022), 'An innovative method in improving the accuracy in intrusion detection by comparing random forest over support vector machine', *2022 International Conference on Business Analytics for Technology and Security (ICBATS)* .

Kumar, N. H. & Dhanalakshmi, R. (2023), 'A novel host based intrusion detection system using supervised learning by comparing svm over random forest'.

Lalitha, A., Kilinc, O. C., Javidi, T. & Koushanfar, F. (2019), 'Peer-to-peer federated learning on graphs', *arXiv (Cornell University)* .

Lee, B.-S., Kim, J.-W. & Choi, M.-J. (2023), 'Federated learning based network intrusion detection model', pp. 330–333.
URL: *https://ieeexplore.ieee.org/document/10258140*

Li, J., Tong, X., Liu, J. & Cheng, L. (2023), 'An efficient federated learning system for network intrusion detection', *IEEE Systems Journal* p. 1–10.
URL: *https://ieeexplore.ieee.org/document/10032055*

Liao, D., Huang, S., Tan, Y. & Bai, G. (2020), 'Network intrusion detection method based on gan model'.

Liu, S., Yu, Y., Zong, Y., Yeoh, P. L., Guo, L., Vucetic, B., Duong, T. Q. & Li, Y. (2024), 'Delay and energy-efficient asynchronous federated learning for intrusion detection in heterogeneous industrial internet of things', *IEEE internet of things journal* pp. 1–1.

MathWorks (2024), 'Support vector machines for binary classification - matlab & simulink'.
URL: *https://www.mathworks.com/help/stats/support-vector-machines-for-binary-classification.html*

Matplotlib (2012), 'Matplotlib: Python plotting — matplotlib 3.1.1 documentation'.
URL: *https://matplotlib.org/*

McMahan, B. & Thakurta, A. (2022), 'Federated learning with formal differential privacy guarantees'.
URL: *https://research.google/blog/federated-learning-with-formal-differential-privacy-guarantees/*

Miyasato, K. (2020), 'Classification report: Precision, recall, f1-score, accuracy'.
URL: *https://medium.com/@kennymiyasato/classification-report-precision-recall-f1-score-accuracy-16a245a437a5*

Nakip, M., Gül, B. C. & Gelenbe, E. (2023), 'Decentralized online federated g-network learning for lightweight intrusion detection'.

Novikova, E. S. & Golubev, S. A. (2023), 'Federated learning based approach to intrusion detection'.

Numpy (2009), 'Numpy'.
URL: *https://numpy.org/*

Ofgem (2024), 'Average gas and electricity usage'.
URL: *https://www.ofgem.gov.uk/average-gas-and-electricity-usage*

Pandas (2018), 'Python data analysis library'.
URL: *https://pandas.pydata.org/*

Rieke, N. (2019), 'What is federated learning?'.
URL: *https://blogs.nvidia.com/blog/what-is-federated-learning/*

Roy, A. G., Siddiqui, S., Pölsterl, S., Navab, N. & Wachinger, C. (2019), 'Braintorrent: A peer-to-peer environment for decentralized federated learning', *arXiv (Cornell University)* .

Saxena, S. (2020), 'A beginner's guide to random forest hyperparameter tuning'.
URL: *https://www.analyticsvidhya.com/blog/2020/03/beginners-guide-random-forest-hyperparameter-tuning*

Scikit-learn (2024), 'scikit-learn: machine learning in python — scikit-learn 0.22.2 documentation'.
URL: *https://scikit-learn.org*

Sharafaldin, I., Habibi Lashkari, A. & Ghorbani, A. A. (2018), 'Toward generating a new intrusion detection dataset and intrusion traffic characterization', *Proceedings of the 4th International Conference on Information Systems Security and Privacy* .
URL: *http://www.scitepress.org/Papers/2018/66398/66398.pdf*

Sruthi, E. R. (2021), 'Random forest — introduction to random forest algorithm'.
URL: *https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest*

Ukpanah, I. (2024), 'What is the average cost of electricity per kwh in the uk?'.
URL: *https://www.greenmatch.co.uk/average-electricity-cost-uk*

University of New Brunswick (2018), 'Ids 2018 — datasets — research — canadian institute for cybersecurity — unb'.
URL: *https://www.unb.ca/cic/datasets/ids-2018.html*

Waskle, S., Parashar, L. & Singh, U. (2020), 'Intrusion detection system using pca with random forest approach', *2020 International Conference on Electronics and Sustainable Communication Systems (ICESC)* .

Wilt, M., Matelsky, J. K. & Gearhart, A. S. (2021), 'Scatterbrained: A flexible and expandable pattern for decentralized machine learning', *arXiv (Cornell University)* .

Xing, H., Simeone, O. & Bi, S. (2020), 'Decentralized federated learning via sgd over wireless d2d networks', *arXiv (Cornell University)* .

Zhang, L., Yan, H. & Zhu, Q. (2020), 'An improved lstm network intrusion detection method'.

Zhou, Y., Qing, Y. & Lv, J. (2020), 'Communication-efficient federated learning with compensated overlap-fedavg', *arXiv (Cornell University)* .

A Ethical Approval Email

This email determines that ethical approval has been waived for this reserach.

WR wmg-FTUndergraduate Resource

Reply Reply all Forward

To:

Wed 07/02/2024 15:04

Cc: Taramonli, Sandy

Date of Approval: **07/02/2024**

Dear

Warwick ID number: 2136685

Project: **How can federated learning be adapted to the domain of privacy-preserving network traffic analysis for intrusion detection?**

Your ethics application number is **WMG-2024- R_6Kyke6ozHAkL7cU**
Your supervisor **Sandy Taramonli** has recommended to the Cyber Ethics Panel the following outcome: **Ethical approval be waived for this research.**

The Cyber Ethics Panel has confirmed this outcome.

You are reminded that you must now adhere to the answers and detail given in the completed ethics form. If anything changes in your research such that any of your answers change to the form for which you received an **ethical waiver** for, then you must contact your supervisor to check if you need to reapply for or update your **ethical waiver** before you proceed with data collection.

When you submit your dissertation, please write **N/A against the ethical approval field** on the cover page of the submission and include a copy of this email into the Appendices of your dissertation.

Kind regards,

WMG Projects Team

B Project Code

B.1 Server Code

B.1.1 Federated Random Forest

```python
import flwr as fl
from typing import Dict
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='Server-FL-RF')
tracker.start()

# Define the global value for the number of clients and the training round
NUM_CLIENTS = 3
ROUNDS = 5


# Return the current round
def fit_config(server_round: int) -> Dict:
    config = {
        "server_round": server_round,
    }
    return config


# Aggregate metrics and calculate weighted averages
def metrics_aggregate(results) -> Dict:
    if not results:
        return {}
```

```

    else:
        total_samples = 0  # Number of samples in the dataset

        # Collecting metrics
        aggregated_metrics = {
            "Accuracy": 0,
            "Precision": 0,
            "Recall": 0,
            "F1_Score": 0,
        }

        # Extracting values from the results
        for samples, metrics in results:
            for key, value in metrics.items():
                if key not in aggregated_metrics:
                    aggregated_metrics[key] = 0
                else:
                    aggregated_metrics[key] += (value * samples)
            total_samples += samples

        # Compute the weighted average for each metric
        for key in aggregated_metrics.keys():
            aggregated_metrics[key] = round(aggregated_metrics[key] / total_samples, 6)

        return aggregated_metrics


if __name__ == "__main__":

    print(f"Server:\n")

    # Build a strategy
    strategy = fl.server.strategy.FedAvg(
        fraction_fit=1.0,
        fraction_evaluate=1.0,
        min_fit_clients=NUM_CLIENTS,
        min_evaluate_clients=NUM_CLIENTS,
        min_available_clients=NUM_CLIENTS,
        on_fit_config_fn=fit_config,
        evaluate_metrics_aggregation_fn=metrics_aggregate,
        fit_metrics_aggregation_fn=metrics_aggregate,
    )

    # Generate a text file for saving the server log
    fl.common.logger.configure(identifier="FL_Test", filename="server-FL-log.txt")

    # Start the server
    fl.server.start_server(
        config=fl.server.ServerConfig(num_rounds=ROUNDS),
        strategy=strategy,
        server_address="0.0.0.0:5556",
```

```
    )
    tracker.stop()
```

B.1.2 Federated Support Vector Machine

```
import flwr as fl
from typing import Dict
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='Server-FL-SVM')
tracker.start()

# Define the global value for the number of clients and the training round
NUM_CLIENTS = 3
ROUNDS = 5


# Return the current round
def fit_config(server_round: int) -> Dict:
    config = {
        "server_round": server_round,
    }
    return config


# Aggregate metrics and calculate weighted averages
def metrics_aggregate(results) -> Dict:
    if not results:
        return {}

    else:
        total_samples = 0  # Number of samples in the dataset

        # Collecting metrics
        aggregated_metrics = {
            "Accuracy": 0,
            "Precision": 0,
            "Recall": 0,
            "F1_Score": 0,
        }

        # Extracting values from the results
        for samples, metrics in results:
            for key, value in metrics.items():
                if key not in aggregated_metrics:
                    aggregated_metrics[key] = 0
                else:
                    aggregated_metrics[key] += (value * samples)
            total_samples += samples

        # Compute the weighted average for each metric
```

```python
        for key in aggregated_metrics.keys():
            aggregated_metrics[key] = round(aggregated_metrics[key] / total_samples, 6)

        return aggregated_metrics


if __name__ == "__main__":

    print(f"Server:\n")

    # Build a strategy
    strategy = fl.server.strategy.FedAvg(
        fraction_fit=1.0,
        fraction_evaluate=1.0,
        min_fit_clients=NUM_CLIENTS,
        min_evaluate_clients=NUM_CLIENTS,
        min_available_clients=NUM_CLIENTS,
        on_fit_config_fn=fit_config,
        evaluate_metrics_aggregation_fn=metrics_aggregate,
        fit_metrics_aggregation_fn=metrics_aggregate,
    )

    # Generate a text file for saving the server log
    fl.common.logger.configure(identifier="FL_Test", filename="server-FL-log.txt")

    # Start the server
    fl.server.start_server(
        config=fl.server.ServerConfig(num_rounds=ROUNDS),
        strategy=strategy,
        server_address="0.0.0.0:5555",
    )
    tracker.stop()
```

B.2 Client 1 Code - InSDN Dataset

B.2.1 Making Pre-Datasets

```python
import pandas as pd

# Sample data for File A, File B, and File C
file_a_data = 'dataset/InSDN_DatasetCSV/Normal_data.csv'
file_b_data = 'dataset/InSDN_DatasetCSV/OVS.csv'
file_c_data = 'dataset/InSDN_DatasetCSV/metasploitable-2.csv'

def fix_ddos(df):
    df['Label'] = df['Label'].replace('DDoS ', 'DDoS')
    return df

# Define the list of labels to split
labels_to_split_a = ['Normal']
```

```
labels_to_split_b = ['DoS', 'DDoS', 'Probe', 'BFA', 'Web-Attack', 'BOTNET']
labels_to_split_c = ['DoS', 'DDoS', 'Probe', 'BFA', 'U2R']

def split_data(df, labels_to_split, ratio=0.8):
    split_data_80 = []
    split_data_20 = []

    for label in labels_to_split:
        label_data = df[df['Label'] == label]
        split_index = round(len(label_data) * ratio)
        # Split the dataframe into 80% and 20% based on the sorted index
        df_80 = label_data.iloc[:split_index]
        df_20 = label_data.iloc[split_index:]

        # Append to respective lists
        split_data_80.append(df_80)
        split_data_20.append(df_20)

    return pd.concat(split_data_80), pd.concat(split_data_20)

def save_to_csv(data_80, data_20, file_prefix):
    data_80.to_csv(f'{file_prefix}_training2.csv', index=False)
    data_20.to_csv(f'{file_prefix}_testing2.csv', index=False)

#Fix file B 'DDos ' to 'DDoS'
df_file_b = pd.read_csv(file_b_data)
df_file_b_fixed = fix_ddos(df_file_b)

# Convert data to DataFrame
df_file_a = pd.read_csv(file_a_data)
df_file_c = pd.read_csv(file_c_data)

# Split data into 80% and 20%
df_file_a_80, df_file_a_20 = split_data(df_file_a, labels_to_split_a)
df_file_b_80, df_file_b_20 = split_data(df_file_b_fixed, labels_to_split_b)
df_file_c_80, df_file_c_20 = split_data(df_file_c, labels_to_split_c)

# Save split data to CSV files
df_combined_80 = pd.concat([df_file_a_80, df_file_b_80, df_file_c_80])
df_combined_20 = pd.concat([df_file_a_20, df_file_b_20, df_file_c_20])
save_to_csv(df_combined_80, df_combined_20, 'full')
```

B.2.2 Data Processing

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
import matplotlib.pyplot as plt
# Load your dataset
```

```
df = pd.read_csv('full_testing.csv')
#df.columns.tolist()

df['Fwd Seg Size Avg'].describe()
columns_to_drop=['Flow ID', 'Src IP', 'Src Port', 'Dst IP', 'Dst Port', 'Protocol', 'Timestamp']
df = df.drop(columns_to_drop, axis=1)

df.head()
label_counts = df['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
# 'Normal' data is 1 and everything else is 0
df['Label'] = df['Label'].apply(lambda x: 1 if x == 'Normal' else 0)

df['Label']
#df.to_csv('full_testing_debloat.csv', index=False)
label_counts = df['Label'].value_counts()
label_counts
label_percentages = df['Label'].value_counts(normalize=True) * 100
label_percentages_formatted = label_percentages.map("{:.3f}%".format)

print(label_percentages_formatted)
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
# Load your dataset
df_test = pd.read_csv('full_testing.csv')
columns_to_drop=['Flow ID', 'Src IP', 'Src Port', 'Dst IP', 'Dst Port', 'Protocol', 'Timestamp']
df_test = df_test.drop(columns_to_drop, axis=1)

df_test.head()
label_counts = df_test['Label'].value_counts()
label_counts
label_percentages = df_test['Label'].value_counts(normalize=True) * 100
label_percentages_formatted = label_percentages.map("{:.3f}%".format)

print(label_percentages_formatted)
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
```

B.2.3 Oversampling

```
# Import necessary libraries
import pandas as pd
from imblearn.over_sampling import BorderlineSMOTE
# Load training dataset
df_train = pd.read_csv('full_training_debloat.csv')
X_train = df_train.drop('Label', axis=1)
y_train = df_train['Label']

bsmote=BorderlineSMOTE(random_state=555, k_neighbors=2, m_neighbors=2, kind='borderline-1')

X_train_resampled, y_train_resampled = bsmote.fit_resample(X_train, y_train)

df_resampled = pd.DataFrame(X_train_resampled, columns=X_train.columns)
df_resampled['Label'] = y_train_resampled
df_resampled
df_resampled.to_csv('full_training_debloat_oversampled.csv', index=False)
# Load testing dataset
df_test = pd.read_csv('full_testing_debloat.csv')
X_test = df_test.drop('Label', axis=1)
y_test = df_test['Label']

bsmote=BorderlineSMOTE(random_state=555, k_neighbors=2, m_neighbors=2, kind='borderline-1')

X_test_resampled, y_test_resampled = bsmote.fit_resample(X_test, y_test)

df_resampled = pd.DataFrame(X_test_resampled, columns=X_test.columns)
df_resampled['Label'] = y_test_resampled
df_resampled
df_resampled.to_csv('full_testing_debloat_oversampled.csv', index=False)
```

B.2.4 Scaling and Feature Importance

```
# Import necessary libraries
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score, confusion_matrix, ConfusionMatrixDisplay, accuracy_score,
classification_report
import matplotlib.pyplot as plt
import numpy as np
# Load training dataset
df_train = pd.read_csv('full_training_debloat_oversampled.csv')
df_train
label_counts = df_train['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
```

```
plt.show()
X_train = df_train.drop('Label', axis=1)
y_train = df_train['Label']
# Load testing dataset
df_test = pd.read_csv('full_testing_debloat_oversampled.csv')
label_counts = df_test['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_test = df_test.drop('Label', axis=1)
y_test = df_test['Label']
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train[X_train.columns] = scaler.fit_transform(X_train[X_train.columns])
X_test[X_test.columns] = scaler.transform(X_test[X_test.columns])
X_train
from sklearn.feature_selection import mutual_info_classif
from sklearn.metrics import accuracy_score

information_gain = mutual_info_classif(X_train, y_train)
information_gain_df = pd.DataFrame({'Feature': X_train.columns, 'Information_Gain': information_gain})
information_gain_df = information_gain_df.sort_values(by='Information_Gain', ascending=False)

#In order to determine which features are most important to train the model with
#I use sklearn's information gain tool to determine the top 25 top features
print("Information Gain for Each Feature:")
print(information_gain_df)

k = 25
selected_features = information_gain_df['Feature'][:k].tolist()
print(f"\nTop {k} Features based on Information Gain:")
print(selected_features)
X_train = X_train[selected_features]
X_test = X_test[selected_features]

#I run this to ensure that the encoding of each set is correct.
columns_match = X_train.columns.equals(X_test.columns)
if columns_match:
    print("The columns in X_train and X_test match.")
else:
    print("The columns in X_train and X_test do not match.")
y_test_df = pd.DataFrame(y_test, columns=['Label'])
df_test_whole = pd.concat([X_test, y_test_df], axis=1)
df_test_whole
df_test_whole.to_csv('testing-full-processed.csv', index=False)
y_train_df = pd.DataFrame(y_train, columns=['Label'])
df_train_whole = pd.concat([X_train, y_train_df], axis=1)
df_train_whole
```

```
df_train_whole.to_csv('training-full-processed.csv', index=False)
```

B.2.5 Initial Random Forest Training

```
# Import necessary libraries
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, accuracy_score, classification_report
import matplotlib.pyplot as plt
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='RF-balanced-oversampled-testing')
tracker.start()
# Load training dataset
df_train = pd.read_csv('full_training_debloat_oversampled.csv')
label_counts = df_train['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X = df_train.drop('Label', axis=1)
y = df_train['Label']
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X, y,
                                                    train_size = 0.8,
                                                    stratify = y,
                                                    random_state = 555)
# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=555, n_jobs=-1)

# Train the classifier on the training data
rf_classifier.fit(X_train, y_train)
rf_classifier.score(X_val, y_val)
# Load testing dataset
df_test = pd.read_csv('full_testing_debloat_oversampled.csv')
label_counts = df_test['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_test = df_test.drop('Label', axis=1)
y_test = df_test['Label']
# Make predictions on the testing data
y_pred = rf_classifier.predict(X_test)
# Evaluate the model performance on testing data
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy on testing data: {accuracy:.2f}')
```

```
# Print classification report for more detailed evaluation
print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix (Test performance with no fine-tuning)')
plt.show()
tracker.stop()
```

B.2.6 Initial Support Vector Machine Training

```
# Import necessary libraries
import pandas as pd
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='SVM-balanced-oversampled-testing')
tracker.start()
# Load training dataset
df_train = pd.read_csv('full_training_debloat_oversampled.csv')
label_counts = df_train['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_train = df_train.drop('Label', axis=1)
y_train = df_train['Label']
# Load testing dataset
df_test = pd.read_csv('full_testing_debloat_oversampled.csv')
label_counts = df_test['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_test = df_test.drop('Label', axis=1)
y_test = df_test['Label']
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Initialize the SVM classifier
svm_classifier = SVC(C=1, kernel='rbf', random_state=555, max_iter=-1)
# Train the classifier on the scaled training data
svm_classifier.fit(X_train_scaled, y_train)
# Make predictions on the scaled testing data
```

```
y_pred = svm_classifier.predict(X_test_scaled)
# Evaluate the model performance on testing data
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy on testing data: {accuracy:.2f}')
# Print classification report for more detailed evaluation
print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=svm_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix (Test performance with no fine-tuning)')
plt.show()
tracker.stop()
```

B.2.7 Random Forest Hyper-Parameter & Classification Threshold Tuning

```
# Import necessary libraries
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score, confusion_matrix, ConfusionMatrixDisplay, accuracy_score,
classification_report
import matplotlib.pyplot as plt
import numpy as np
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='RF-FPdata-hyperparameters')
tracker.start()
# Load training dataset
df_train = pd.read_csv('training-full-processed.csv')
label_counts = df_train['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_train = df_train.drop('Label', axis=1)
y_train = df_train['Label']
# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(random_state=555, n_jobs=-1)

# Train the classifier on the training data
rf_classifier.fit(X_train, y_train)
# Load testing dataset
df_test = pd.read_csv('testing-full-processed.csv')
label_counts = df_test['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
```

```
plt.show()
X_test = df_test.drop('Label', axis=1)
y_test = df_test['Label']
# Make predictions on the testing data
y_pred = rf_classifier.predict(X_test)
# Evaluate the model performance on testing data
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy on testing data: {accuracy:.2f}')
# Print classification report for more detailed evaluation
print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix (Test performance with no fine-tuning)')
plt.show()
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [1, 100, 1000],
    'max_depth': [10, 50],
    'min_samples_split': [2, 4],
    'min_samples_leaf': [2, 4],
    'max_features': ["sqrt", "log2"]
}

grid_search = GridSearchCV(rf_classifier, param_grid, cv=5, scoring='accuracy', n_jobs=-1, verbose=0)

grid_search.fit(X_train, y_train)
print("Best Parameters:", grid_search.best_params_)
best_params = grid_search.best_params_
best_rf_classifier = RandomForestClassifier(random_state=555, **best_params)
best_rf_classifier.fit(X_train, y_train)
#Testing the model after the hyperparameters have been tuned
y_test_pred_default = best_rf_classifier.predict(X_test)
print("\nTest Performance with Default Threshold (Best Model):")
print("Accuracy:", accuracy_score(y_test, y_test_pred_default))
print("Classification Report:")
print(classification_report(y_test, y_test_pred_default))
#Displaying the confusion matrix
cm_default = confusion_matrix(y_test, y_test_pred_default)
disp_default = ConfusionMatrixDisplay(confusion_matrix=cm_default, display_labels=best_rf_classifier.classes_)
disp_default.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix (Best Model with Default Threshold)')
plt.show()
#Initialising the variables needed to test the threshhold.
y_test_probs = best_rf_classifier.predict_proba(X_test)[:, 1]
thresholds_to_try = np.arange(0, 1, 0.01)
best_threshold = 0
best_metric = 0

#If a new threshold gives a higher f1_score then those parameters are saved for the end.
for threshold in thresholds_to_try:
```

```python
    y_test_pred_custom = (y_test_probs >= threshold).astype(int)
    current_metric = f1_score(y_test, y_test_pred_custom)
    if current_metric > best_metric:
        best_metric = current_metric
        best_threshold = threshold

y_test_pred_final = (y_test_probs >= best_threshold).astype(int)
print("\nTest Performance with Best Threshold:")
print("Best Threshold:", round(best_threshold, 3))
print("Accuracy:", accuracy_score(y_test, y_test_pred_final))
print("Classification Report:")
print(classification_report(y_test, y_test_pred_final))
cm = confusion_matrix(y_test, y_test_pred_final)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=best_rf_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix')
plt.show()
tracker.stop()
```

B.2.8 Federated Random Forest Helper

```python
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from typing import List


def load_dataset(client_id: int):
    df_train = pd.read_csv('training.csv')

    X_train = df_train.drop('Label', axis=1)
    y_train = df_train['Label']

    df_test = pd.read_csv('testing.csv')
    X_test = df_test.drop('Label', axis=1)
    y_test = df_test['Label']

    # Each of the following is divided equally into thirds
    return X_train, y_train, X_test, y_test


# Look at the RandomForestClassifier documentation of sklearn and select the parameters
# Get the parameters from the RandomForestClassifier
def get_params(model: RandomForestClassifier) -> List[np.ndarray]:
    params = [
        model.n_estimators,
        model.max_depth,
        model.min_samples_split,
        model.min_samples_leaf,
```

```
    ]
    return params


# Set the parameters in the RandomForestClassifier
def set_params(model: RandomForestClassifier, params: List[np.ndarray]) -> RandomForestClassifier:
    model.n_estimators = int(params[0])
    model.max_depth = int(params[1])
    model.min_samples_split = int(params[2])
    model.min_samples_leaf = int(params[3])
    return model
```

B.2.9 Federated Random Forest Client Script

```
import helper
import numpy as np
import flwr as fl
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import log_loss, accuracy_score, precision_score, recall_score, f1_score
import warnings
warnings.simplefilter('ignore')
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='Client1-FL-RF')
tracker.start()

# Create the flower client
class FlowerClient(fl.client.NumPyClient):

    # Get the current local model parameters
    def get_parameters(self, config):
        print(f"Client {client_id} received the parameters.")
        return helper.get_params(model)

    # Train the local model, return the model parameters to the server
    def fit(self, parameters, config):
        print("Parameters before setting: ", parameters)
        helper.set_params(model, parameters)
        print("Parameters after setting: ", model.get_params())

        model.fit(X_train, y_train)
        print(f"Training finished for round {config['server_round']}.")

        trained_params = helper.get_params(model)
        print("Trained Parameters: ", trained_params)

        return trained_params, len(X_train), {}

    # Evaluate the local model, return the evaluation result to the server
    def evaluate(self, parameters, config):
        #start
```

```python
        #Initialising the variables needed to test the threshhold.
        y_test_probs = model.predict_proba(X_test)[:, 1]
        thresholds_to_try = np.arange(0, 1, 0.01)
        best_threshold = 0
        best_metric = 0

        #If a new threshold gives a higher f1_score then those parameters are saved for the end.
        for threshold in thresholds_to_try:
            y_test_pred_custom = (y_test_probs >= threshold).astype(int)
            current_metric = f1_score(y_test, y_test_pred_custom)
            if current_metric > best_metric:
                best_metric = current_metric
                best_threshold = threshold

        y_test_pred_final = (y_test_probs >= best_threshold).astype(int)
        helper.set_params(model, parameters)
        #end

        loss = log_loss(y_test, y_test_pred_final, labels=[0, 1])

        accuracy = accuracy_score(y_test, y_test_pred_final)
        precision = precision_score(y_test, y_test_pred_final, average='weighted')
        recall = recall_score(y_test, y_test_pred_final, average='weighted')
        f1 = f1_score(y_test, y_test_pred_final, average='weighted')

        line = "-" * 21
        print(line)
        print(f"Accuracy : {accuracy:.8f}")
        print(f"Precision: {precision:.8f}")
        print(f"Recall   : {recall:.8f}")
        print(f"F1 Score : {f1:.8f}")
        print(line)

        return loss, len(X_test), {"Accuracy": accuracy, "Precision": precision, "Recall": recall,
                                   "F1_Score": f1}


if __name__ == "__main__":
    client_id = 1
    print(f"Client {client_id}:\n")

    # Get the dataset for local model
    X_train, y_train, X_test, y_test = helper.load_dataset(client_id - 1)

    # Print the label distribution
    unique, counts = np.unique(y_train, return_counts=True)
    train_counts = dict(zip(unique, counts))
    print("Label distribution in the training set:", train_counts)
    unique, counts = np.unique(y_test, return_counts=True)
    test_counts = dict(zip(unique, counts))
    print("Label distribution in the testing set:", test_counts, '\n')

```

```
    # Create and fit the local model
    model = RandomForestClassifier(
        max_depth= 50,
        max_features= 'sqrt',
        min_samples_leaf= 4,
        min_samples_split= 2,
        n_estimators= 1000
    )

    fl.common.logger.configure(identifier="FL_Test", filename="client1-FL-log.txt")

    model.fit(X_train, y_train)

    # Start the client
    fl.client.start_numpy_client(server_address="10.10.2.230:5556", client=FlowerClient())
    tracker.stop()
```

B.2.10 Federated Random Forest Bash Script

```
#!/bin/sh

python3 helper.py
echo "helper done"
python3 client1-RF.py
```

B.2.11 Federated Support Vector Machine Helper

```
import pandas as pd
import numpy as np
from sklearn.svm import SVC
from typing import List


def load_dataset(client_id: int):
    df_train = pd.read_csv('training.csv')

    X_train = df_train.drop('Label', axis=1)
    y_train = df_train['Label']

    df_test = pd.read_csv('testing.csv')
    X_test = df_test.drop('Label', axis=1)
    y_test = df_test['Label']

    # Each of the following is divided equally into thirds
    return X_train, y_train, X_test, y_test


# Get the parameters from the SVC
```

```python
def get_params(model: SVC) -> List[np.ndarray]:
    params = [
        model.C
    ]
    return params


# Set the parameters in the SVC
def set_params(model: SVC, params: List[np.ndarray]) -> SVC:
    model.C = int(params[0])
    model.kernel = 'rbf'
    model.random_state = 555
    model.max_iter = -1
    return model
```

B.2.12 Federated Support Vector Machine Client Script

```python
import helper
import numpy as np
import flwr as fl
from sklearn.svm import SVC
from sklearn.metrics import log_loss, accuracy_score, precision_score, recall_score, f1_score
import warnings
warnings.simplefilter('ignore')
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='Client1-FL-SVM')
tracker.start()

# Create the flower client
class FlowerClient(fl.client.NumPyClient):

    # Get the current local model parameters
    def get_parameters(self, config):
        print(f"Client {client_id} received the parameters.")
        return helper.get_params(model)

    # Train the local model, return the model parameters to the server
    def fit(self, parameters, config):
        print("Parameters before setting: ", parameters)
        helper.set_params(model, parameters)
        print("Parameters after setting: ", model.get_params())

        model.fit(X_train, y_train)
        print(f"Training finished for round {config['server_round']}.")

        trained_params = helper.get_params(model)
        print("Trained Parameters: ", trained_params)

        return trained_params, len(X_train), {}

```

```
    # Evaluate the local model, return the evaluation result to the server
    def evaluate(self, parameters, config):
        helper.set_params(model, parameters)

        y_pred = model.predict(X_test)
        loss = log_loss(y_test, y_pred, labels=[0, 1])

        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred, average='weighted')
        recall = recall_score(y_test, y_pred, average='weighted')
        f1 = f1_score(y_test, y_pred, average='weighted')

        line = "-" * 21
        print(line)
        print(f"Accuracy : {accuracy:.8f}")
        print(f"Precision: {precision:.8f}")
        print(f"Recall   : {recall:.8f}")
        print(f"F1 Score : {f1:.8f}")
        print(line)

        return loss, len(X_test), {"Accuracy": accuracy, "Precision": precision, "Recall": recall,
                                    "F1_Score": f1}

if __name__ == "__main__":
    client_id = 1
    print(f"Client {client_id}:\n")

    # Get the dataset for local model
    X_train, y_train, X_test, y_test = helper.load_dataset(client_id - 1)

    # Print the label distribution
    unique, counts = np.unique(y_train, return_counts=True)
    train_counts = dict(zip(unique, counts))
    print("Label distribution in the training set:", train_counts)
    unique, counts = np.unique(y_test, return_counts=True)
    test_counts = dict(zip(unique, counts))
    print("Label distribution in the testing set:", test_counts, '\n')

    # Create and fit the local model
    model = SVC(C=1, kernel='rbf', random_state=555, max_iter=-1)

    fl.common.logger.configure(identifier="FL_Test", filename="client1-FL-log.txt")

    model.fit(X_train, y_train)

    # Start the client
    fl.client.start_numpy_client(server_address="10.10.2.230:5555", client=FlowerClient())
    tracker.stop()
```

B.2.13 Federated Support Vector Machine Bash Script

```
#!/bin/sh

python3 helper.py
echo "helper done"
python3 client1.py
```

B.3 Client 2 Code - CSE-CIC-IDS2018 Dataset

B.3.1 Making Pre-Datasets

```
import pandas as pd
import pandas as pd
import os

# Folder containing the datasets
folder_path = 'dataset/CSE-CIC-IDS2018/'

exclude_file = '02-21-2018.csv' # for testing

# List to store DataFrames
dfs = []

# Iterate over files in the folder
for filename in os.listdir(folder_path):
    if filename.endswith('.csv') and filename != exclude_file:  # Assuming CSV files
        file_path = os.path.join(folder_path, filename)
        print(file_path)
        # Read the CSV file in chunks
        chunk_iter = pd.read_csv(file_path, chunksize=20000)  # Adjust chunksize as needed
        for chunk in chunk_iter:
            dfs.append(chunk)
            break

# Concatenate all chunks into a single DataFrame
combined_df = pd.concat(dfs, ignore_index=True)

# Now you have the combined DataFrame 'combined_df' with data from all files
combined_df
unique_labels = combined_df['Label'].unique()
unique_labels
combined_df[combined_df['Label']=='Label']
combined_df = combined_df[combined_df['Label']!='Label'] # drop the weird label ones
label_counts = combined_df['Label'].value_counts()
label_counts
import matplotlib.pyplot as plt
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
```

```
plt.title('Distribution of Labels')
plt.show()
combined_df.to_csv('training.csv', index=False)
```

B.3.2 Data Processing

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
# Load your dataset
df = pd.read_csv('training.csv')
df
#df.columns.tolist()

df['Src Port'].describe()
columns_to_drop=['Flow ID', 'Src IP', 'Src Port', 'Dst IP', 'Dst Port', 'Protocol', 'Timestamp']
df = df.drop(columns_to_drop, axis=1)

df.head()
label_counts = df['Label'].value_counts()
label_counts
label_percentages = df['Label'].value_counts(normalize=True) * 100
label_percentages_formatted = label_percentages.map("{:.3f}%".format)

print(label_percentages_formatted)
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
# 'Normal' data is 1 and everything else is 0
df['Label'] = df['Label'].apply(lambda x: 1 if x == 'Benign' else 0)

df['Label']
label_counts = df['Label'].value_counts()
label_counts
label_percentages = df['Label'].value_counts(normalize=True) * 100
label_percentages_formatted = label_percentages.map("{:.3f}%".format)

print(label_percentages_formatted)
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
a = df.columns
for column in a:
    df.fillna({column:df[column].mean()})
```

```
df
df = df[~df.isin([np.inf, -np.inf, np.nan]).any(axis=1)] # removing infinite values
df
label_counts = df['Label'].value_counts()
label_counts
label_percentages = df['Label'].value_counts(normalize=True) * 100
label_percentages_formatted = label_percentages.map("{:.3f}%".format)

print(label_percentages_formatted)
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
df.to_csv('training_debloat.csv', index=False)
# Load your dataset THIS IS SPECIFICALLY FOR THE TESTING UNBALANCED SET.
# FOR TRAINING, GO TO THE OTHER NOTEBOOK
df = pd.read_csv('dataset/CSE-CIC-IDS2018/02-21-2018.csv')
df
#df.columns.tolist()

df['Dst Port'].describe()
columns_to_drop=['Dst Port', 'Protocol', 'Timestamp']
df = df.drop(columns_to_drop, axis=1)

df.head()
label_counts = df['Label'].value_counts()
label_counts
label_percentages = df['Label'].value_counts(normalize=True) * 100
label_percentages_formatted = label_percentages.map("{:.3f}%".format)

print(label_percentages_formatted)
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
# 'Normal' data is 1 and everything else is 0
df['Label'] = df['Label'].apply(lambda x: 1 if x == 'Benign' else 0)

df['Label']
label_counts = df['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
a = df.columns
for column in a:
    df.fillna({column:df[column].mean()})
df
```

```
df = df[~df.isin([np.inf, -np.inf, np.nan]).any(axis=1)] # removing infinite values
df
label_counts = df['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
df.to_csv('testing_unbalanced.csv', index=False)
# Define the list of labels to split
labels_to_split = ['Benign', 'DDOS attack-HOIC']
file_data = 'dataset/CSE-CIC-IDS2018/02-21-2018.csv'

def split_data(df, labels_to_split, ratio=0.3):
    split_data_benign = []
    split_data_ddos = []

    # Count the number of 'Benign' label

    for label in labels_to_split:
        benign_count = (df['Label'] == label).sum()
        split_count = round(benign_count * ratio)
        label_data = df[df['Label'] == label]

        # Split the dataframe based on the calculated count
        df_split = label_data.iloc[:split_count]

        # Append to respective lists
        if label == 'Benign':
            split_data_benign.append(df_split)
        elif label == 'DDOS attack-HOIC':
            split_data_ddos.append(df_split)

    return pd.concat(split_data_benign), pd.concat(split_data_ddos)

# Convert data to DataFrame
df_file = pd.read_csv(file_data)

# Split data into benign and ddos
df_file_benign, df_file_ddos = split_data(df_file, labels_to_split)

combined_df = pd.concat([df_file_benign, df_file_ddos])

#combined_df.to_csv('testing.csv', index=False)
combined_df
label_counts = combined_df['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
```

```
plt.show()
# 'Normal' data is 1 and everything else is 0
combined_df['Label'] = combined_df['Label'].apply(lambda x: 1 if x == 'Benign' else 0)

combined_df['Label'].value_counts()
combined_df
columns_to_drop=['Dst Port', 'Protocol', 'Timestamp']
combined_df = combined_df.drop(columns_to_drop, axis=1)

combined_df
a = combined_df.columns
for column in a:
    combined_df.fillna({column:combined_df[column].mean()})
combined_df = combined_df[~combined_df.isin([np.inf, -np.inf, np.nan]).any(axis=1)] # removing infinite values
combined_df
combined_df['Label'].value_counts()
combined_df.to_csv('testing_unbalanced.csv', index=False)
```

B.3.3 Scaling and Feature Importance

```
# Import necessary libraries
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score, confusion_matrix, ConfusionMatrixDisplay, accuracy_score,
classification_report
import matplotlib.pyplot as plt
import numpy as np
# Load training dataset
df_train = pd.read_csv('training_debloat.csv')
df_train
label_counts = df_train['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_train = df_train.drop('Label', axis=1)
y_train = df_train['Label']
X_train
# Load testing dataset
df_test = pd.read_csv('testing_unbalanced.csv')
df_test
label_counts = df_test['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
```

```python
X_test = df_test.drop('Label', axis=1)
y_test = df_test['Label']
X_test
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train[X_train.columns] = scaler.fit_transform(X_train[X_train.columns])
X_test[X_test.columns] = scaler.transform(X_test[X_test.columns])
X_train
X_test
from sklearn.feature_selection import mutual_info_classif
from sklearn.metrics import accuracy_score

information_gain = mutual_info_classif(X_train, y_train)
information_gain_df = pd.DataFrame({'Feature': X_train.columns, 'Information_Gain': information_gain})
information_gain_df = information_gain_df.sort_values(by='Information_Gain', ascending=False)

#In order to determine which features are most important to train the model with
# I use sklearn's information gain tool to determine the top 25 top features
print("Information Gain for Each Feature:")
print(information_gain_df)

k = 25
selected_features = information_gain_df['Feature'][:k].tolist()
print(f"\nTop {k} Features based on Information Gain:")
print(selected_features)
X_train = X_train[selected_features]
X_test = X_test[selected_features]

#I run this to ensure that the encoding of each set is correct.
columns_match = X_train.columns.equals(X_test.columns)
if columns_match:
    print("The columns in X_train and X_test match.")
else:
    print("The columns in X_train and X_test do not match.")
y_test_df = pd.DataFrame(y_test, columns=['Label'])
df_test_whole = pd.concat([X_test, y_test_df], axis=1)
df_test_whole
label_counts = df_test_whole['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
df_test_whole.to_csv('testing-full-processed.csv', index=False)
y_train_df = pd.DataFrame(y_train, columns=['Label'])
df_train_whole = pd.concat([X_train, y_train_df], axis=1)
df_train_whole
label_counts = df_train_whole['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
```

```
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
df_train_whole.to_csv('training-full-processed.csv', index=False)
```

B.3.4 Initial Random Forest Training

```
# Import necessary libraries
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, accuracy_score, classification_report
import matplotlib.pyplot as plt
import numpy as np
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='RF-unbalanced-testing')
tracker.start()
# Load training dataset
df_train = pd.read_csv('training_debloat.csv')
df_train
label_counts = df_train['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X = df_train.drop('Label', axis=1)
y = df_train['Label']
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X, y,
                                                  train_size = 0.8,
                                                  stratify = y,
                                                  random_state = 555)
# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=555, n_jobs=-1)

# Train the classifier on the training data
rf_classifier.fit(X_train, y_train)
rf_classifier.score(X_val, y_val)
# Load testing dataset
df_test = pd.read_csv('testing_unbalanced.csv')
label_counts = df_test['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_test = df_test.drop('Label', axis=1)
y_test = df_test['Label']
```

```
# Make predictions on the testing data
y_pred = rf_classifier.predict(X_test)
# Evaluate the model performance on testing data
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy on testing data: {accuracy:.2f}')
# Print classification report for more detailed evaluation
print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix (Test performance with no fine-tuning)')
plt.show()
tracker.stop()
```

B.3.5 Initial Support Vector Machine Training

```
# Import necessary libraries
import pandas as pd
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='SVM-unbalanced-testing')
tracker.start()
# Load training dataset
df_train = pd.read_csv('training_debloat.csv')
label_counts = df_train['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_train = df_train.drop('Label', axis=1)
y_train = df_train['Label']
# Load testing dataset
df_test = pd.read_csv('testing_unbalanced.csv')
label_counts = df_test['Label'].value_counts()
label_counts
label_counts = df_test['Label'].value_counts()
label_counts
X_test = df_test.drop('Label', axis=1)
y_test = df_test['Label']
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Initialize the SVM classifier
svm_classifier = SVC(C=1, kernel='rbf', random_state=555, max_iter=-1)

```

```python
# Train the classifier on the scaled training data
svm_classifier.fit(X_train_scaled, y_train)
# Make predictions on the scaled testing data
y_pred = svm_classifier.predict(X_test_scaled)
# Evaluate the model performance on testing data
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy on testing data: {accuracy:.2f}')
# Print classification report for more detailed evaluation
print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=svm_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix (Test performance with no fine-tuning)')
plt.show()
tracker.stop()
```

B.3.6 Random Forest Hyper-Parameter & Classification Threshold Tuning

```python
# Import necessary libraries
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score, confusion_matrix, ConfusionMatrixDisplay, accuracy_score,
classification_report
import matplotlib.pyplot as plt
import numpy as np
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='RF-FPdata-hyperparameters')
tracker.start()
# Load training dataset
df_train = pd.read_csv('training-full-processed.csv')
df_train
label_counts = df_train['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_train = df_train.drop('Label', axis=1)
y_train = df_train['Label']
# Load testing dataset
df_test = pd.read_csv('testing-full-processed.csv')
label_counts = df_test['Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
```

```
X_test = df_test.drop('Label', axis=1)
y_test = df_test['Label']
# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=555, n_jobs=-1)

# Train the classifier on the training data
rf_classifier.fit(X_train, y_train)
# Make predictions on the testing data
y_pred = rf_classifier.predict(X_test)
# Evaluate the model performance on testing data
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy on testing data: {accuracy:.2f}')
# Print classification report for more detailed evaluation
print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix (Test performance with no fine-tuning)')
plt.show()
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [250, 500],
    'max_depth': [25, 50, 75],
    'min_samples_split': [2],
    'min_samples_leaf': [2],
    'max_features': ["sqrt"]
}

grid_search = GridSearchCV(rf_classifier, param_grid, cv=5, scoring='accuracy', n_jobs=-1, verbose=0)

grid_search.fit(X_train, y_train)
print("Best Parameters:", grid_search.best_params_)
best_params = grid_search.best_params_
best_rf_classifier = RandomForestClassifier(random_state=555, **best_params)
best_rf_classifier.fit(X_train, y_train)
#Testing the model after the hyperparameters have been tuned
y_test_pred_default = best_rf_classifier.predict(X_test)
print("\nTest Performance with Default Threshold (Best Model):")
print("Accuracy:", accuracy_score(y_test, y_test_pred_default))
print("Classification Report:")
print(classification_report(y_test, y_test_pred_default))
#Displaying the confusion matrix
cm_default = confusion_matrix(y_test, y_test_pred_default)
disp_default = ConfusionMatrixDisplay(confusion_matrix=cm_default, display_labels=best_rf_classifier.classes_)
disp_default.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix (Best Model with Default Threshold)')
plt.show()
#Initialising the variables needed to test the threshhold.
y_test_probs = best_rf_classifier.predict_proba(X_test)[:, 1]
thresholds_to_try = np.arange(0, 1, 0.01)
best_threshold = 0
```

```
best_metric = 0

#If a new threshold gives a higher f1_score then those parameters are saved for the end.
for threshold in thresholds_to_try:
    y_test_pred_custom = (y_test_probs >= threshold).astype(int)
    current_metric = f1_score(y_test, y_test_pred_custom)
    if current_metric > best_metric:
        best_metric = current_metric
        best_threshold = threshold

y_test_pred_final = (y_test_probs >= best_threshold).astype(int)
print("\nTest Performance with Best Threshold:")
print("Best Threshold:", round(best_threshold, 3))
print("Accuracy:", accuracy_score(y_test, y_test_pred_final))
print("Classification Report:")
print(classification_report(y_test, y_test_pred_final))
cm = confusion_matrix(y_test, y_test_pred_final)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=best_rf_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix')
plt.show()
tracker.stop()
```

B.3.7 Federated Random Forest Helper

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from typing import List


def load_dataset(client_id: int):
    df_train = pd.read_csv('training.csv')

    X_train = df_train.drop('Label', axis=1)
    y_train = df_train['Label']

    df_test = pd.read_csv('testing.csv')
    X_test = df_test.drop('Label', axis=1)
    y_test = df_test['Label']

    # Each of the following is divided equally into thirds
    return X_train, y_train, X_test, y_test


# Look at the RandomForestClassifier documentation of sklearn and select the parameters
# Get the parameters from the RandomForestClassifier
def get_params(model: RandomForestClassifier) -> List[np.ndarray]:
    params = [
```

```
        model.n_estimators,
        model.max_depth,
        model.min_samples_split,
        model.min_samples_leaf,
    ]
    return params


# Set the parameters in the RandomForestClassifier
def set_params(model: RandomForestClassifier, params: List[np.ndarray]) -> RandomForestClassifier:
    model.n_estimators = int(params[0])
    model.max_depth = int(params[1])
    model.min_samples_split = int(params[2])
    model.min_samples_leaf = int(params[3])
    return model
```

B.3.8 Federated Random Forest Client Script

```
import helper
import numpy as np
import flwr as fl
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import log_loss, accuracy_score, precision_score, recall_score, f1_score
import warnings
warnings.simplefilter('ignore')
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='Client2-FL-RF')
tracker.start()

# Create the flower client
class FlowerClient(fl.client.NumPyClient):

    # Get the current local model parameters
    def get_parameters(self, config):
        print(f"Client {client_id} received the parameters.")
        return helper.get_params(model)

    # Train the local model, return the model parameters to the server
    def fit(self, parameters, config):
        print("Parameters before setting: ", parameters)
        helper.set_params(model, parameters)
        print("Parameters after setting: ", model.get_params())

        model.fit(X_train, y_train)
        print(f"Training finished for round {config['server_round']}.")

        trained_params = helper.get_params(model)
        print("Trained Parameters: ", trained_params)

        return trained_params, len(X_train), {}
```

```python

    # Evaluate the local model, return the evaluation result to the server
    def evaluate(self, parameters, config):
        #start
        #Initialising the variables needed to test the threshhold.
        y_test_probs = model.predict_proba(X_test)[:, 1]
        thresholds_to_try = np.arange(0, 1, 0.01)
        best_threshold = 0
        best_metric = 0

        #If a new threshold gives a higher f1_score then those parameters are saved for the end.
        for threshold in thresholds_to_try:
            y_test_pred_custom = (y_test_probs >= threshold).astype(int)
            current_metric = f1_score(y_test, y_test_pred_custom)
            if current_metric > best_metric:
                best_metric = current_metric
                best_threshold = threshold

        y_test_pred_final = (y_test_probs >= best_threshold).astype(int)
        helper.set_params(model, parameters)
        #end

        loss = log_loss(y_test, y_test_pred_final, labels=[0, 1])

        accuracy = accuracy_score(y_test, y_test_pred_final)
        precision = precision_score(y_test, y_test_pred_final, average='weighted')
        recall = recall_score(y_test, y_test_pred_final, average='weighted')
        f1 = f1_score(y_test, y_test_pred_final, average='weighted')

        line = "-" * 21
        print(line)
        print(f"Accuracy : {accuracy:.8f}")
        print(f"Precision: {precision:.8f}")
        print(f"Recall   : {recall:.8f}")
        print(f"F1 Score : {f1:.8f}")
        print(line)

        return loss, len(X_test), {"Accuracy": accuracy, "Precision": precision, "Recall": recall,
                                   "F1_Score": f1}


if __name__ == "__main__":
    client_id = 2
    print(f"Client {client_id}:\n")

    # Get the dataset for local model
    X_train, y_train, X_test, y_test = helper.load_dataset(client_id - 1)

    # Print the label distribution
    unique, counts = np.unique(y_train, return_counts=True)
    train_counts = dict(zip(unique, counts))
    print("Label distribution in the training set:", train_counts)
```

```
    unique, counts = np.unique(y_test, return_counts=True)
    test_counts = dict(zip(unique, counts))
    print("Label distribution in the testing set:", test_counts, '\n')

    # Create and fit the local model
    model = RandomForestClassifier(
        max_depth= 50,
        max_features= 'sqrt',
        min_samples_leaf= 2,
        min_samples_split= 2,
        n_estimators= 1000
    )

    fl.common.logger.configure(identifier="FL_Test", filename="client2-FL-log.txt")

    model.fit(X_train, y_train)

    # Start the client
    fl.client.start_numpy_client(server_address="10.10.2.230:5556", client=FlowerClient())
    tracker.stop()
```

B.3.9 Federated Random Forest Bash Script

```
#!/bin/sh

python3 helper.py
echo "helper done"
python3 client2-RF.py
```

B.3.10 Federated Support Vector Machine Helper

```
import pandas as pd
import numpy as np
from sklearn.svm import SVC
from typing import List


def load_dataset(client_id: int):
    df_train = pd.read_csv('training.csv')

    X_train = df_train.drop('Label', axis=1)
    y_train = df_train['Label']

    df_test = pd.read_csv('testing.csv')
    X_test = df_test.drop('Label', axis=1)
    y_test = df_test['Label']

    # Each of the following is divided equally into thirds
```

```python
    return X_train, y_train, X_test, y_test


# Get the parameters from the SVC
def get_params(model: SVC) -> List[np.ndarray]:
    params = [
        model.C
    ]
    return params


# Set the parameters in the SVC
def set_params(model: SVC, params: List[np.ndarray]) -> SVC:
    model.C = int(params[0])
    model.kernel = 'rbf'
    model.random_state = 555
    model.max_iter = -1
    return model
```

B.3.11 Federated Support Vector Machine Client Script

```python
import helper
import numpy as np
import flwr as fl
from sklearn.svm import SVC
from sklearn.metrics import log_loss, accuracy_score, precision_score, recall_score, f1_score
import warnings
warnings.simplefilter('ignore')
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='Client2-FL-SVM')
tracker.start()

# Create the flower client
class FlowerClient(fl.client.NumPyClient):

    # Get the current local model parameters
    def get_parameters(self, config):
        print(f"Client {client_id} received the parameters.")
        return helper.get_params(model)

    # Train the local model, return the model parameters to the server
    def fit(self, parameters, config):
        print("Parameters before setting: ", parameters)
        helper.set_params(model, parameters)
        print("Parameters after setting: ", model.get_params())

        model.fit(X_train, y_train)
        print(f"Training finished for round {config['server_round']}.")

        trained_params = helper.get_params(model)
```

```
        print("Trained Parameters: ", trained_params)

        return trained_params, len(X_train), {}

    # Evaluate the local model, return the evaluation result to the server
    def evaluate(self, parameters, config):
        helper.set_params(model, parameters)

        y_pred = model.predict(X_test)
        loss = log_loss(y_test, y_pred, labels=[0, 1])

        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred, average='weighted')
        recall = recall_score(y_test, y_pred, average='weighted')
        f1 = f1_score(y_test, y_pred, average='weighted')

        line = "-" * 21
        print(line)
        print(f"Accuracy : {accuracy:.8f}")
        print(f"Precision: {precision:.8f}")
        print(f"Recall    : {recall:.8f}")
        print(f"F1 Score : {f1:.8f}")
        print(line)

        return loss, len(X_test), {"Accuracy": accuracy, "Precision": precision, "Recall": recall,
                                   "F1_Score": f1}

if __name__ == "__main__":
    client_id = 2
    print(f"Client {client_id}:\n")

    # Get the dataset for local model
    X_train, y_train, X_test, y_test = helper.load_dataset(client_id - 1)

    # Print the label distribution
    unique, counts = np.unique(y_train, return_counts=True)
    train_counts = dict(zip(unique, counts))
    print("Label distribution in the training set:", train_counts)
    unique, counts = np.unique(y_test, return_counts=True)
    test_counts = dict(zip(unique, counts))
    print("Label distribution in the testing set:", test_counts, '\n')

    # Create and fit the local model
    model = SVC(C=1, kernel='rbf', random_state=555, max_iter=-1)

    fl.common.logger.configure(identifier="FL_Test", filename="client2-FL-log.txt")

    model.fit(X_train, y_train)

    # Start the client
    fl.client.start_numpy_client(server_address="10.10.2.230:5555", client=FlowerClient())
```

```
    tracker.stop()
```

B.3.12 Federated Support Vector Machine Bash Script

```
#!/bin/sh

python3 helper.py
echo "helper done"
python3 client2.py
```

B.4 Client 3 Code - CIC-IDS2017 Dataset

B.4.1 Making Pre-Datasets

```
import pandas as pd
import os
# Folder containing the datasets
folder_path = 'dataset/MachineLearningCVE/'

exclude_file = 'Wednesday-workingHours.pcap_ISCX.csv' # for testing

# List to store DataFrames
dfs = []

# Iterate over files in the folder
for filename in os.listdir(folder_path):
    if filename.endswith('.csv') and filename != exclude_file:  # Assuming CSV files
        file_path = os.path.join(folder_path, filename)
        print(file_path)
        # Read the CSV file in chunks
        dataset = pd.read_csv(file_path)  # Adjust chunksize as needed
        dfs.append(dataset)

# Concatenate all chunks into a single DataFrame
combined_df = pd.concat(dfs, ignore_index=True)
print('done')
# Now you have the combined DataFrame 'combined_df' with data from all files
combined_df
combined_df.columns
unique_labels = combined_df[' Label'].unique()
unique_labels
label_counts = combined_df[' Label'].value_counts()
label_counts
import matplotlib.pyplot as plt
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
```

```
plt.show()
combined_df.to_csv('all-training.csv', index=False)
```

B.4.2 Data Processing

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
# Load your dataset
#df = pd.read_csv('dataset/MachineLearningCVE/Wednesday-workingHours.pcap_ISCX.csv')
df = pd.read_csv('all-training.csv')
df
df.columns
columns_to_drop=[' Destination Port']
df = df.drop(columns_to_drop, axis=1)

df
label_counts = df[' Label'].value_counts()
label_counts
label_percentages = df[' Label'].value_counts(normalize=True) * 100
label_percentages_formatted = label_percentages.map("{:.3f}%".format)

print(label_percentages_formatted)
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
# ONLY USE IF YOU NEED TO DELETE SOME BENIGN SAMPLES
                                                                    #none for testing unbalanced
df = df.drop(df[df[' Label'] == 'BENIGN'].sample(frac=0.82).index)  #0.4 for testing balanced
                                                                    #0.8 for training unbalanced
                                                                    #0.82 for training balanced
df
label_counts = df[' Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
# 'Normal' data is 1 and everything else is 0
df[' Label'] = df[' Label'].apply(lambda x: 1 if x == 'BENIGN' else 0)

df[' Label']
label_counts = df[' Label'].value_counts()
label_counts
# Plotting the pie chart
```

```
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
a = df.columns
for column in a:
    df.fillna({column:df[column].mean()})
df
df = df[~df.isin([np.inf, -np.inf, np.nan]).any(axis=1)] # removing infinite values
df
label_counts = df[' Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
df.to_csv('training_balanced.csv', index=False)
```

B.4.3 Scaling and Feature Importance

```
# Import necessary libraries
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score, confusion_matrix, ConfusionMatrixDisplay, accuracy_score,
classification_report
import matplotlib.pyplot as plt
import numpy as np
# Load training dataset
df_train = pd.read_csv('training_balanced.csv')
df_train
label_counts = df_train[' Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_train = df_train.drop(' Label', axis=1)
y_train = df_train[' Label']
# Load testing dataset
df_test = pd.read_csv('testing_unbalanced.csv')
label_counts = df_test[' Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_test = df_test.drop(' Label', axis=1)
```

```
y_test = df_test[' Label']
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train[X_train.columns] = scaler.fit_transform(X_train[X_train.columns])
X_test[X_test.columns] = scaler.transform(X_test[X_test.columns])
X_train
from sklearn.feature_selection import mutual_info_classif
from sklearn.metrics import accuracy_score

information_gain = mutual_info_classif(X_train, y_train)
information_gain_df = pd.DataFrame({'Feature': X_train.columns, 'Information_Gain': information_gain})
information_gain_df = information_gain_df.sort_values(by='Information_Gain', ascending=False)

#In order to determine which features are most important to train the model with
#I use sklearn's information gain tool to determine the top 25 top features
print("Information Gain for Each Feature:")
print(information_gain_df)

k = 25
selected_features = information_gain_df['Feature'][:k].tolist()
print(f"\nTop {k} Features based on Information Gain:")
print(selected_features)
X_train = X_train[selected_features]
X_test = X_test[selected_features]

#I run this to ensure that the encoding of each set is correct.
columns_match = X_train.columns.equals(X_test.columns)
if columns_match:
    print("The columns in X_train and X_test match.")
else:
    print("The columns in X_train and X_test do not match.")
y_test_df = pd.DataFrame(y_test, columns=[' Label'])
df_test_whole = pd.concat([X_test, y_test_df], axis=1)
df_test_whole
df_test_whole.to_csv('testing-full-processed.csv', index=False)
y_train_df = pd.DataFrame(y_train, columns=[' Label'])
df_train_whole = pd.concat([X_train, y_train_df], axis=1)
df_train_whole
df_train_whole.to_csv('training-full-processed.csv', index=False)
```

B.4.4 Initial Random Forest Training

```
# Import necessary libraries
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, accuracy_score, classification_report
import matplotlib.pyplot as plt
import numpy as np
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='RF-unbalanced-testing')
```

```
tracker.start()
# Load training dataset
df_train = pd.read_csv('training_balanced.csv')
df_train
label_counts = df_train[' Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X = df_train.drop(' Label', axis=1)
y = df_train[' Label']
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X, y,
                                                    train_size = 0.8,
                                                    stratify = y,
                                                    random_state = 555)
# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=555, n_jobs=-1)

# Train the classifier on the training data
rf_classifier.fit(X_train, y_train)
rf_classifier.score(X_val, y_val)
# Load testing dataset
df_test = pd.read_csv('testing_unbalanced.csv')
label_counts = df_test[' Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_test = df_test.drop(' Label', axis=1)
y_test = df_test[' Label']
# Make predictions on the testing data
y_pred = rf_classifier.predict(X_test)
# Evaluate the model performance on testing data
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy on testing data: {accuracy:.2f}')
# Print classification report for more detailed evaluation
print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix (Test performance with no fine-tuning)')
plt.show()
tracker.stop()
```

B.4.5 Initial Support Vector Machine Training

```python
# Import necessary libraries
import pandas as pd
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, accuracy_score, classification_report
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='SVM-unbalanced-testing')
tracker.start()
# Load training dataset
df_train = pd.read_csv('training_balanced.csv')
label_counts = df_train[' Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_train = df_train.drop(' Label', axis=1)
y_train = df_train[' Label']
# Load testing dataset
df_test = pd.read_csv('testing_unbalanced.csv')
label_counts = df_test[' Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_test = df_test.drop(' Label', axis=1)
y_test = df_test[' Label']
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Initialize the SVM classifier
svm_classifier = SVC(C=1, kernel='rbf', random_state=555, max_iter=-1)

# Train the classifier on the scaled training data
svm_classifier.fit(X_train_scaled, y_train)
# Make predictions on the scaled testing data
y_pred = svm_classifier.predict(X_test_scaled)
# Evaluate the model performance on testing data
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy on testing data: {accuracy:.2f}')
# Print classification report for more detailed evaluation
print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=svm_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
```

```
plt.title('Confusion Matrix (Test performance with no fine-tuning)')
plt.show()
tracker.stop()
```

B.4.6 Random Forest Hyper-Parameter & Classification Threshold Tuning

```
# Import necessary libraries
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score, confusion_matrix, ConfusionMatrixDisplay, accuracy_score,
classification_report
import matplotlib.pyplot as plt
import numpy as np
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='RF-FPdata-hyperparameters')
tracker.start()
# Load training dataset
df_train = pd.read_csv('training-full-processed.csv')
df_train
label_counts = df_train[' Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_train = df_train.drop(' Label', axis=1)
y_train = df_train[' Label']
# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=555, n_jobs=-1)

# Train the classifier on the training data
rf_classifier.fit(X_train, y_train)
# Load testing dataset
df_test = pd.read_csv('testing-full-processed.csv')
label_counts = df_test[' Label'].value_counts()
label_counts
# Plotting the pie chart
plt.figure(figsize=(8, 6))  # Optional: Adjust the figure size as needed
plt.pie(label_counts, labels=label_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Distribution of Labels')
plt.show()
X_test = df_test.drop(' Label', axis=1)
y_test = df_test[' Label']
# Make predictions on the testing data
y_pred = rf_classifier.predict(X_test)
# Evaluate the model performance on testing data
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy on testing data: {accuracy:.2f}')
```

```
# Print classification report for more detailed evaluation
print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=rf_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix (Test performance with no fine-tuning)')
plt.show()
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [500, 1000, 1500],
    'max_depth': [25, 50],
    'min_samples_split': [2, 3],
    'min_samples_leaf': [3, 4],
    'max_features': ["sqrt"]
}

grid_search = GridSearchCV(rf_classifier, param_grid, cv=5, scoring='accuracy', n_jobs=1, verbose=0)

grid_search.fit(X_train, y_train)
print("Best Parameters:", grid_search.best_params_)
#best_params = grid_search.best_params_
best_params = {'max_depth': 50, 'max_features': 'sqrt', 'min_samples_leaf': 4, 'min_samples_split': 2,
                'n_estimators': 500}
best_rf_classifier = RandomForestClassifier(random_state=555, **best_params)
best_rf_classifier.fit(X_train, y_train)
#Testing the model after the hyperparameters have been tuned
y_test_pred_default = best_rf_classifier.predict(X_test)
print("\nTest Performance with Default Threshold (Best Model):")
print("Accuracy:", accuracy_score(y_test, y_test_pred_default))
print("Classification Report:")
print(classification_report(y_test, y_test_pred_default))
#Displaying the confusion matrix
cm_default = confusion_matrix(y_test, y_test_pred_default)
disp_default = ConfusionMatrixDisplay(confusion_matrix=cm_default, display_labels=best_rf_classifier.classes_)
disp_default.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix (Best Model with Default Threshold)')
plt.show()
#Initialising the variables needed to test the threshhold.
y_test_probs = best_rf_classifier.predict_proba(X_test)[:, 1]
thresholds_to_try = np.arange(0, 1, 0.01)
best_threshold = 0
best_metric = 0

#If a new threshold gives a higher f1_score then those parameters are saved for the end.
for threshold in thresholds_to_try:
    y_test_pred_custom = (y_test_probs >= threshold).astype(int)
    current_metric = f1_score(y_test, y_test_pred_custom)
    if current_metric > best_metric:
        best_metric = current_metric
        best_threshold = threshold

```

```
y_test_pred_final = (y_test_probs >= best_threshold).astype(int)
print("\nTest Performance with Best Threshold:")
print("Best Threshold:", round(best_threshold, 3))
print("Accuracy:", accuracy_score(y_test, y_test_pred_final))
print("Classification Report:")
print(classification_report(y_test, y_test_pred_final))
cm = confusion_matrix(y_test, y_test_pred_final)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=best_rf_classifier.classes_)
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title('Confusion Matrix')
plt.show()
tracker.stop()
```

B.4.7 Federated Random Forest Helper

```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from typing import List


def load_dataset(client_id: int):
    df_train = pd.read_csv('training.csv')

    X_train = df_train.drop(' Label', axis=1)
    y_train = df_train[' Label']

    df_test = pd.read_csv('testing.csv')
    X_test = df_test.drop(' Label', axis=1)
    y_test = df_test[' Label']

    # Each of the following is divided equally into thirds
    return X_train, y_train, X_test, y_test


# Look at the RandomForestClassifier documentation of sklearn and select the parameters
# Get the parameters from the RandomForestClassifier
def get_params(model: RandomForestClassifier) -> List[np.ndarray]:
    params = [
        model.n_estimators,
        model.max_depth,
        model.min_samples_split,
        model.min_samples_leaf,
    ]
    return params


# Set the parameters in the RandomForestClassifier
def set_params(model: RandomForestClassifier, params: List[np.ndarray]) -> RandomForestClassifier:
```

```
    model.n_estimators = int(params[0])
    model.max_depth = int(params[1])
    model.min_samples_split = int(params[2])
    model.min_samples_leaf = int(params[3])
    return model
```

B.4.8 Federated Random Forest Client Script

```
import helper
import numpy as np
import flwr as fl
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import log_loss, accuracy_score, precision_score, recall_score, f1_score
import warnings
warnings.simplefilter('ignore')
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='Client3-FL-RF')
tracker.start()

# Create the flower client
class FlowerClient(fl.client.NumPyClient):

    # Get the current local model parameters
    def get_parameters(self, config):
        print(f"Client {client_id} received the parameters.")
        return helper.get_params(model)

    # Train the local model, return the model parameters to the server
    def fit(self, parameters, config):
        print("Parameters before setting: ", parameters)
        helper.set_params(model, parameters)
        print("Parameters after setting: ", model.get_params())

        model.fit(X_train, y_train)
        print(f"Training finished for round {config['server_round']}.")

        trained_params = helper.get_params(model)
        print("Trained Parameters: ", trained_params)

        return trained_params, len(X_train), {}

    # Evaluate the local model, return the evaluation result to the server
    def evaluate(self, parameters, config):
        #start
        #Initialising the variables needed to test the threshhold.
        y_test_probs = model.predict_proba(X_test)[:, 1]
        thresholds_to_try = np.arange(0, 1, 0.01)
        best_threshold = 0
        best_metric = 0

```

```python
        #If a new threshold gives a higher f1_score then those parameters are saved for the end.
        for threshold in thresholds_to_try:
            y_test_pred_custom = (y_test_probs >= threshold).astype(int)
            current_metric = f1_score(y_test, y_test_pred_custom)
            if current_metric > best_metric:
                best_metric = current_metric
                best_threshold = threshold

        y_test_pred_final = (y_test_probs >= best_threshold).astype(int)
        helper.set_params(model, parameters)
        #end

        loss = log_loss(y_test, y_test_pred_final, labels=[0, 1])

        accuracy = accuracy_score(y_test, y_test_pred_final)
        precision = precision_score(y_test, y_test_pred_final, average='weighted')
        recall = recall_score(y_test, y_test_pred_final, average='weighted')
        f1 = f1_score(y_test, y_test_pred_final, average='weighted')

        line = "-" * 21
        print(line)
        print(f"Accuracy : {accuracy:.8f}")
        print(f"Precision: {precision:.8f}")
        print(f"Recall   : {recall:.8f}")
        print(f"F1 Score : {f1:.8f}")
        print(line)

        return loss, len(X_test), {"Accuracy": accuracy, "Precision": precision, "Recall": recall,
                                   "F1_Score": f1}


if __name__ == "__main__":
    client_id = 3
    print(f"Client {client_id}:\n")

    # Get the dataset for local model
    X_train, y_train, X_test, y_test = helper.load_dataset(client_id - 1)

    # Print the label distribution
    unique, counts = np.unique(y_train, return_counts=True)
    train_counts = dict(zip(unique, counts))
    print("Label distribution in the training set:", train_counts)
    unique, counts = np.unique(y_test, return_counts=True)
    test_counts = dict(zip(unique, counts))
    print("Label distribution in the testing set:", test_counts, '\n')

    # Create and fit the local model
    model = RandomForestClassifier(
        max_depth= 50,
        max_features= 'sqrt',
        min_samples_leaf= 4,
        min_samples_split= 2,
```

```python
        n_estimators= 1000
    )

    fl.common.logger.configure(identifier="FL_Test", filename="client3-FL-log.txt")

    model.fit(X_train, y_train)

    # Start the client
    fl.client.start_numpy_client(server_address="10.10.2.230:5556", client=FlowerClient())
    tracker.stop()
```

B.4.9 Federated Random Forest Bash Script

```sh
#!/bin/sh

python3 helper.py
echo "helper done"
python3 client3-RF.py
```

B.4.10 Federated Support Vector Machine Helper

```python
import pandas as pd
import numpy as np
from sklearn.svm import SVC
from typing import List


def load_dataset(client_id: int):
    df_train = pd.read_csv('training.csv')

    X_train = df_train.drop(' Label', axis=1)
    y_train = df_train[' Label']

    df_test = pd.read_csv('testing.csv')
    X_test = df_test.drop(' Label', axis=1)
    y_test = df_test[' Label']

    # Each of the following is divided equally into thirds
    return X_train, y_train, X_test, y_test


# Get the parameters from the SVC
def get_params(model: SVC) -> List[np.ndarray]:
    params = [
        model.C
    ]
    return params

```

```python

# Set the parameters in the SVC
def set_params(model: SVC, params: List[np.ndarray]) -> SVC:
    model.C = int(params[0])
    model.kernel = 'rbf'
    model.random_state = 555
    model.max_iter = -1
    return model
```

B.4.11 Federated Support Vector Machine Client Script

```python
import helper
import numpy as np
import flwr as fl
from sklearn.svm import SVC
from sklearn.metrics import log_loss, accuracy_score, precision_score, recall_score, f1_score
import warnings
warnings.simplefilter('ignore')
from codecarbon import EmissionsTracker
tracker = EmissionsTracker(project_name='Client3-FL-SVM')
tracker.start()

# Create the flower client
class FlowerClient(fl.client.NumPyClient):

    # Get the current local model parameters
    def get_parameters(self, config):
        print(f"Client {client_id} received the parameters.")
        return helper.get_params(model)

    # Train the local model, return the model parameters to the server
    def fit(self, parameters, config):
        print("Parameters before setting: ", parameters)
        helper.set_params(model, parameters)
        print("Parameters after setting: ", model.get_params())

        model.fit(X_train, y_train)
        print(f"Training finished for round {config['server_round']}.")

        trained_params = helper.get_params(model)
        print("Trained Parameters: ", trained_params)

        return trained_params, len(X_train), {}

    # Evaluate the local model, return the evaluation result to the server
    def evaluate(self, parameters, config):
        helper.set_params(model, parameters)

        y_pred = model.predict(X_test)
        loss = log_loss(y_test, y_pred, labels=[0, 1])
```

```python

        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred, average='weighted')
        recall = recall_score(y_test, y_pred, average='weighted')
        f1 = f1_score(y_test, y_pred, average='weighted')

        line = "-" * 21
        print(line)
        print(f"Accuracy : {accuracy:.8f}")
        print(f"Precision: {precision:.8f}")
        print(f"Recall    : {recall:.8f}")
        print(f"F1 Score : {f1:.8f}")
        print(line)

        return loss, len(X_test), {"Accuracy": accuracy, "Precision": precision, "Recall": recall,
                                    "F1_Score": f1}

if __name__ == "__main__":
    client_id = 3
    print(f"Client {client_id}:\n")

    # Get the dataset for local model
    X_train, y_train, X_test, y_test = helper.load_dataset(client_id - 1)

    # Print the label distribution
    unique, counts = np.unique(y_train, return_counts=True)
    train_counts = dict(zip(unique, counts))
    print("Label distribution in the training set:", train_counts)
    unique, counts = np.unique(y_test, return_counts=True)
    test_counts = dict(zip(unique, counts))
    print("Label distribution in the testing set:", test_counts, '\n')

    # Create and fit the local model
    model = SVC(C=1, kernel='rbf', random_state=555, max_iter=-1)

    fl.common.logger.configure(identifier="FL_Test", filename="client3-FL-log.txt")

    model.fit(X_train, y_train)

    # Start the client
    fl.client.start_numpy_client(server_address="10.10.2.230:5555", client=FlowerClient())
    tracker.stop()
```

B.4.12 Federated Support Vector Machine Bash Script

```sh
#!/bin/sh

python3 helper.py
echo "helper done"
python3 client3.py
```