**SYNOPSIS**

Electricity theft is a global problem that negatively affects both utility companies and electricity users. It destabilizes the economic development of utility companies, causes electric hazards and impacts the high cost of energy for users. The development of smart grids plays an important role in electricity theft detection since they generate massive data that includes customer consumption data which, through deep learning techniques, can be utilized to detect electricity theft. This project introduces the theft detection method which uses comprehensive features in time and frequency domains in a deep neural network-based classification approach. We address dataset weaknesses such as missing data and class imbalance problems through data interpolation and synthetic data generation processes. We analyse and compare the contribution of features from both time and frequency domains, run experiments in combined and reduced feature space using principal component analysis and finally incorporate minimum redundancy maximum relevance scheme for validating the most important features. We improve the electricity theft detection performance by optimizing hyperparameters and we employ an adaptive moment estimation optimizer to carry out experiments using different values of key parameters to determine the optimal settings that achieve the best accuracy.

**SYSTEM ENVIRONMENT**

2.1 Hardware Requirements:

Processor : Intel Core i4 (10th Gen)

Ram : 4.0 GB

2.2 Software Requirements

Operating System : Windows 10

Framework : Googlecolab

Language : python

**2.3 About the technology:**

Python:

Python is an interpreted high-level general-purpose programming language created by Guido Van Rossum and first published in 1991. Python's design philosophy emphasizes code readability with significant whitespace. Its language structures and object-oriented approach are designed to help developers write clear and logical code for small and large projects. Python is dynamically typed and garbage collected.

Google Colab:

Google Colab, short for Google Collaboratory, is a cloud-based, interactive computing platform provided by Google. It allows users to write and execute Python code in a collaborative and convenient environment directly through a web browser. Colab provides free access to GPU and TPU (Tensor Processing Unit) resources, enabling accelerated execution of machine learning tasks. Users can create and share Jupyter notebooks, incorporating text, code, and visualizations seamlessly. Colab integrates with Google Drive, facilitating easy storage and sharing of notebooks. Its collaborative features enable multiple users to work on the same document simultaneously, fostering collaborative research and development. Overall, Google Colab is a powerful and accessible tool for data analysis, machine learning, and collaborative coding, making it particularly valuable for researchers, students, and practitioners in the field of data science.

Scikit Learn:

Scikit-learn (Sklearn) is the most useful and powerful Python machine learning library. It provides a number of powerful tools for machine learning and statistical modelling, including classification, regression, clustering and dimensionality reduction through a Python consistent interface. Written mostly in Python, this library is built on top of NumPy, SciPy and Matplotlib. Originally called scikits. learn, it was originally developed by David Cournapeau as a Google Summer Code Project in 2007. Later, in 2010, Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, and Vincent Michel from FIRCA (French Institute for Informatics and Automation) adopted it this project to a new level and released the first public release (v0.1 beta) on February 1, 2010

**EXISTING SYSTEM**

Existing systems in machine learning (ML) for artificial intelligence theft identification in multi-source and multi-powered smart grid applications typically rely on various algorithms and techniques to detect anomalies or unauthorized activities within the smart grid infrastructure. One common approach involves the utilization of supervised learning algorithms such as Support Vector Machines (SVM), Decision Trees, Random Forests, these algorithms are trained on historical data collected from sensors, meters, and other devices deployed across the smart grid to learn patterns of normal behaviour and identify deviations indicative of potential theft or security breaches. Additionally, ensemble methods or hybrid models combining multiple algorithms may be employed to enhance detection accuracy and robustness.

Advantages of existing ML systems for theft identification in smart grids include their ability to analyse large volumes of heterogeneous data from diverse sources in real-time, enabling prompt detection and response to security threats.

ML algorithms can adapt and evolve over time, improving their detection capabilities as they encounter new data patterns and threats. Moreover, these systems can provide insights into the underlying causes and characteristics of security incidents, facilitating proactive measures to strengthen the resilience of smart grid infrastructure.

However, existing ML systems also face several challenges and disadvantages.

* One notable issue is the reliance on historical data, which may not fully capture the diversity and complexity of potential security threats in dynamic smart grid environments.
* Moreover, the performance of ML models heavily depends on the quality and representativeness of the training data, and the effectiveness of feature selection and engineering processes.
* Additionally, ML-based detection systems may be susceptible to adversarial attacks or evasion techniques aimed at circumventing detection mechanisms.
* Furthermore, the computational resources required for training and running sophisticated ML models in real-time applications can be significant, posing scalability and resource constraints, particularly for large-scale smart grid deployments.
* Addressing these challenges requires ongoing research and development efforts to enhance the accuracy, efficiency, and resilience of ML-based theft identification systems in smart grids.

**PROPOSED SYSTEM**

The proposed artificial intelligence theft identification system for Multi-source and Multi-powered Smart grid applications utilizing Gated Recurrent Unit (GRU) networks integrates advanced deep learning techniques to enhance security measures. The system aims to analyse diverse data sources within the smart grid, including energy consumption patterns, grid infrastructure data, and historical theft incidents, leveraging GRU's ability to capture temporal dependencies in sequential data. By processing and learning from these multi-source inputs, the GRU-based system can effectively detect anomalous behaviors indicative of theft or unauthorized usage across various power sources and grid configurations. The proposed approach offers a more sophisticated and adaptive solution compared to traditional methods, allowing for real-time identification and mitigation of security threats in Smart grid environments.

**Advantages of the Proposed System:**

Temporal Modelling: The GRU architecture enables the system to capture temporal dependencies and sequential patterns in smart grid data more effectively, allowing for accurate detection of anomalous behaviour indicative of theft activities over time.

Flexibility and Adaptability: GRU networks offer flexibility and adaptability, making them well-suited for handling diverse smart grid configurations and evolving theft patterns. The proposed system can seamlessly integrate new data and adapt to changing grid dynamics.

Efficiency in Training: Compared to more complex deep learning architectures, GRU networks require fewer parameters and are computationally efficient, allowing for faster training and inference times. This efficiency is particularly advantageous in real-time theft detection applications.

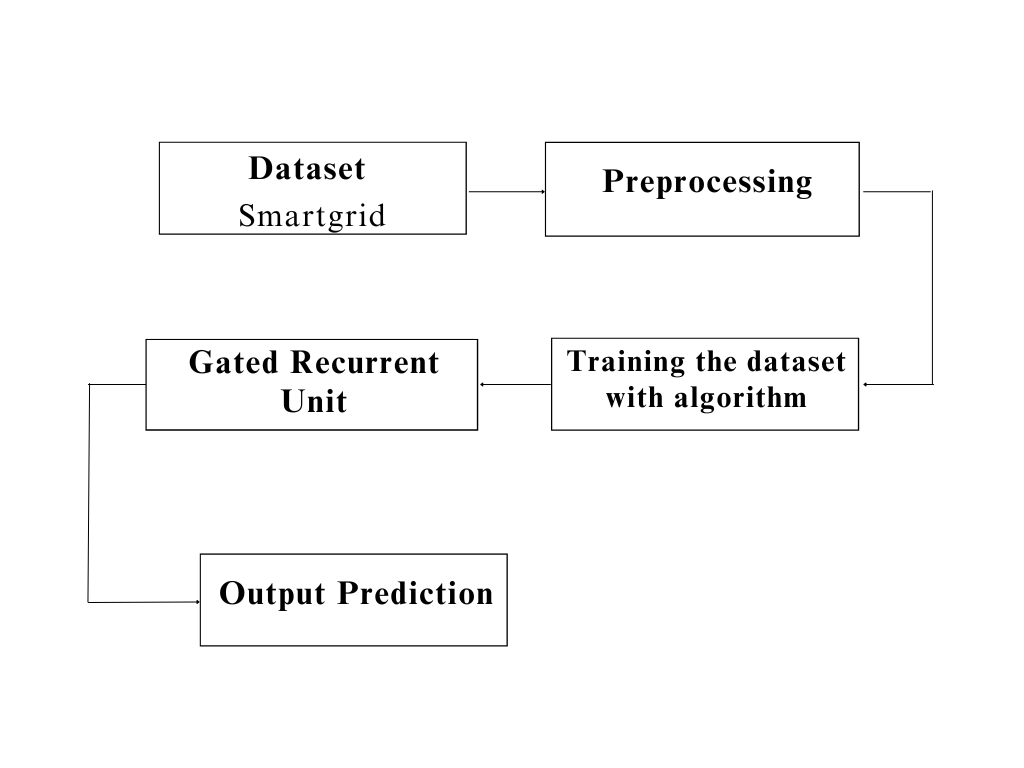
Scalability: The proposed system can scale to large-scale smart grid deployments with multiple energy sources and grid configurations, thanks to the scalability of GRU architectures and the ability to process large volumes of data efficiently.

Accuracy and Robustness: By leveraging the capabilities of GRU networks and incorporating diverse features from multi-source smart grid data, the proposed system achieves high accuracy and robustness in theft identification, reducing false positives and false negatives compared to traditional methods.

the proposed AI theft identification system utilizing GRU architectures offers a comprehensive and efficient approach to enhancing security and integrity in Multi-source, Multi-powered Smart Grid applications. By leveraging the advantages of GRU networks for temporal modelling and anomaly detection, the system provides accurate and scalable theft detection capabilities, thereby safeguarding smart grid infrastructure against unauthorized activities and ensuring reliable energy distribution.

**SYSTEM DESIGN**

Artificial intelligent Theft Identification System for Smart Grids Based on Gated Recurrent Unit system design is specified below:



**Dataset Description:**

This dataset consists of 60,000 samples, each containing 13 features labelled as tau1 through tau4, p1 through p4, g1 through g4, and stab. These features represent various attributes related to a system's stability, including time constants (tau), power parameters (p), and generator parameters (g). Additionally, the dataset includes a target variable labelled stabf, indicating the stability classification of each sample. The stab feature represents the overall stability of the system, while stabf categorizes the stability into binary classes, likely denoting stable (e.g., 'stable') and unstable (e.g., 'unstable') states. This dataset is well-suited for machine learning tasks, particularly classification, where the objective is to predict the stability of a system based on its attributes. The inclusion of both continuous and categorical features provides ample information for training predictive models, enabling the development of robust algorithms for stability assessment in similar systems.

**Pre-Processing:**

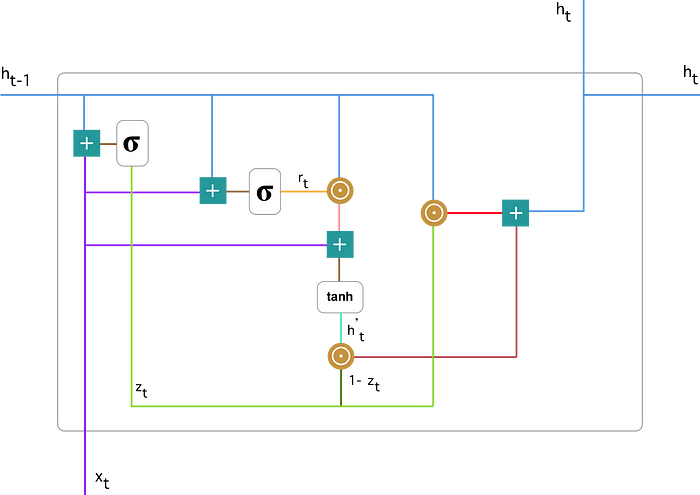
preprocessing steps for this dataset, the 'stabf' column, which represents the stability classification of each sample, is mapped to numeric values to facilitate machine learning model training. Specifically, the class 'unstable' is encoded as 1, while 'stable' is encoded as 0, providing a binary representation of the stability labels. This mapping enables easier interpretation and manipulation of the target variable during model development and evaluation. Additionally, to ensure consistency, the 'stabf' column is converted to numeric format using the 'pd.to\_numeric()' function from the pandas library. The 'errors' parameter is set to 'coerce' to handle any errors encountered during the conversion process by replacing them with NaN values. This ensures that the stability classification data is appropriately formatted and ready for subsequent analysis and model training. Overall, these preprocessing steps help prepare the dataset for further exploration and utilization in machine learning tasks, particularly classification, where numerical representations of target variables are typically required for model training.

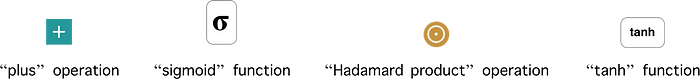
**Deep learning algorithm:**

**GRU**

Gated Recurrent Unit (GRU) is a variant of Recurrent Neural Networks (RNNs) that addresses some of the limitations of traditional RNNs, such as the vanishing gradient problem, while also being computationally efficient. GRU was introduced by Kyunghyun Cho et al. in their paper "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation" in 2014. GRU has gained popularity in various sequence modelling tasks including natural language processing, time series prediction, and speech recognition due to its effectiveness and simplicity.

mathematics behind that process of GRU





Architecture of GRU:

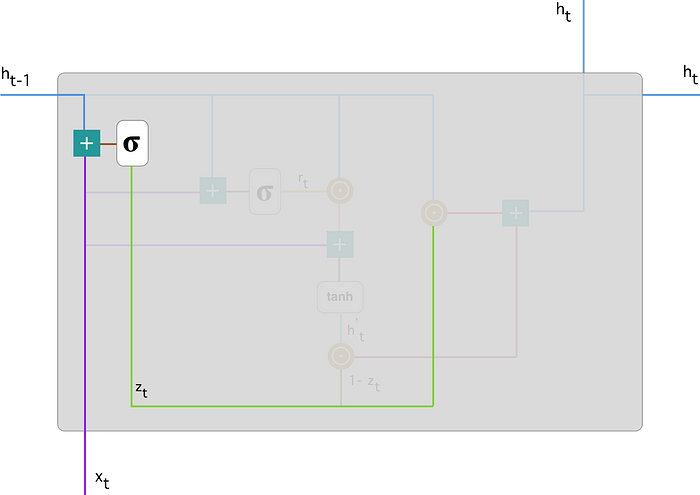
The GRU architecture consists of a set of gates that control the flow of information within the network. Unlike traditional RNNs, GRU has two types of gates: update gate and reset gate. These gates enable GRU to selectively update and reset the hidden state, allowing it to capture long-term dependencies in sequential data more effectively.

Update Gate: The update gate decides how much of the previous hidden state should be retained and how much of the new candidate state should be included. It takes the input and the previous hidden state as input and outputs a value between 0 and 1 for each element of the hidden state vector. A value close to 0 means that the corresponding element of the hidden state will be updated minimally, while a value close to 1 means that it will be updated significantly.

The mathematically notation:



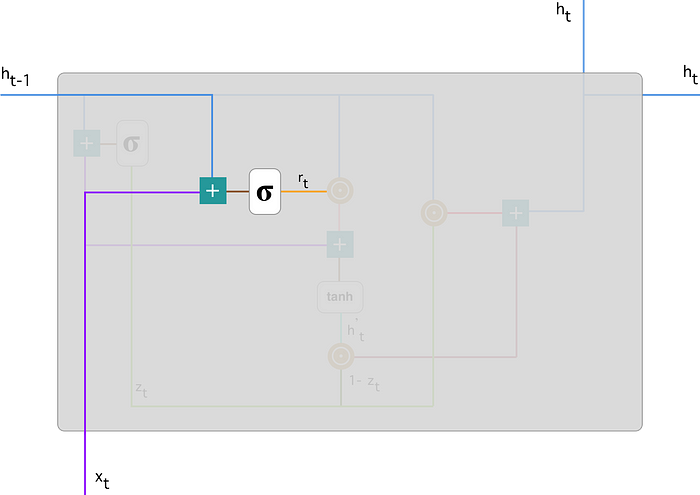
When x\_t is connected to a network device, it is multiplied by its own mass W(z). The same applies to h\_(t-1), which contains information from the previous t-1 units and is multiplied by its own mass U(z). Both results are summed and a sigmoid activation function is used to compress the result between 0 and 1. According to the diagram above, we have:



Reset Gate: The reset gate determines how much of the previous hidden state should be ignored when computing the new candidate state. Similar to the update gate, it takes the input and the previous hidden state as input and outputs a value between 0 and 1. A value close to 0 means that the corresponding element of the hidden state will be ignored, while a value close to 1 means that it will be considered fully.

The mathematical notation:

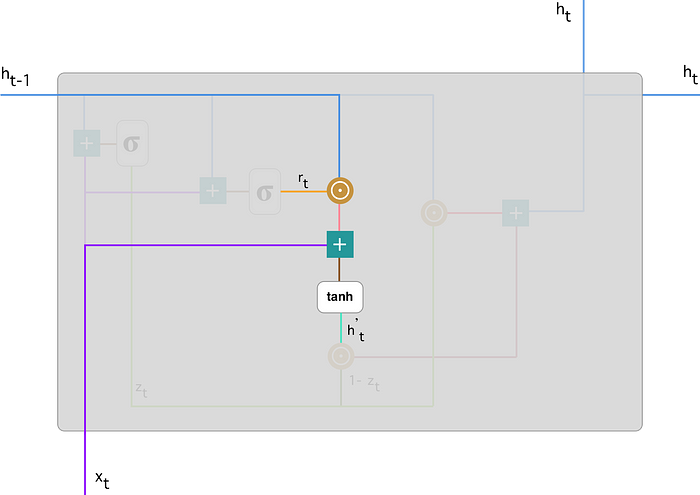




Candidate State Calculation: The candidate state is calculated based on the input and the previous hidden state, similar to traditional RNNs. However, in GRU, this calculation is modulated by the reset gate, which controls how much of the previous hidden state should be considered.



Multiply input x\_t by mass W and h\_(t-1) by weight U.



Final Hidden State: The final hidden state is a combination of the previous hidden state and the candidate state, determined by the update gate. It decides how much of the new candidate state should be included in the final hidden state.



1.Apply element-wise multiplication to the update gate z\_t and h\_(t-1).

2.Apply element-wise multiplication to (1-z\_t) and h’\_t.

Sum the results from step 1 and 2.

Advantages of GRU:

Efficient Training: GRU addresses the vanishing gradient problem better than traditional RNNs, making it easier to train on long sequences of data. This efficiency in training allows GRU to capture long-term dependencies in sequential data more effectively.

Fewer Parameters: GRU has fewer parameters compared to other variants of RNNs such as Long Short-Term Memory (LSTM), which makes it computationally efficient and faster to train. This advantage is particularly significant in applications where computational resources are limited.

Parallelization: GRU operations can be parallelized more effectively than LSTM operations, leading to faster training and inference times, especially on hardware with parallel processing capabilities like GPUs.

Libraries used in the implementation:

NumPy: NumPy is a fundamental library for numerical computing in Python, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions. It serves as a foundational tool for scientific computing tasks, enabling efficient and high-performance operations on numerical data.

Pandas: Pandas is a versatile data manipulation library in Python that offers data structures like DataFrames and Series, facilitating efficient data analysis and manipulation. It provides functionalities for cleaning, transforming, and exploring datasets, making it a go-to tool for handling structured data in various stages of the data science workflow.

Matplotlib: Matplotlib is a powerful plotting library for Python that allows the creation of diverse static, animated, and interactive visualizations. With a comprehensive set of functions, Matplotlib provides users with the flexibility to create various charts, plots, and graphs, making it an essential tool for data visualization and communication of findings.

Seaborn: Seaborn is a statistical data visualization library built on top of Matplotlib. It provides a high-level interface for creating aesthetically pleasing and informative statistical graphics. Seaborn simplifies the process of generating complex visualizations, including heatmaps, pair plots, and violin plots, while maintaining customization options for advanced users.

Metrics (Accuracy, Classification, Confusion Matrix, ROC AUC): In the context of machine learning evaluation, metrics play a crucial role. Accuracy represents the proportion of correctly classified instances, serving as a fundamental measure of model performance. Classification metrics, such as precision, recall, and F1-score, provide insights into the model's ability to correctly identify instances of a particular class. The confusion matrix presents a comprehensive summary of true positive, true negative, false positive, and false negative predictions. Lastly, the ROC AUC (Receiver Operating Characteristic - Area Under the Curve) is a performance metric for binary classification models, illustrating the trade-off between sensitivity and specificity across different thresholds, providing a holistic view of the model's discriminatory power. These metrics collectively aid in assessing and optimizing the performance of machine learning models.

Libraries used in the implementation:

NumPy: NumPy is a fundamental library for numerical computing in Python, providing support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions. It serves as a foundational tool for scientific computing tasks, enabling efficient and high-performance operations on numerical data.

Pandas: Pandas is a versatile data manipulation library in Python that offers data structures like DataFrames and Series, facilitating efficient data analysis and manipulation. It provides functionalities for cleaning, transforming, and exploring datasets, making it a go-to tool for handling structured data in various stages of the data science workflow.

Matplotlib: Matplotlib is a powerful plotting library for Python that allows the creation of diverse static, animated, and interactive visualizations. With a comprehensive set of functions, Matplotlib provides users with the flexibility to create various charts, plots, and graphs, making it an essential tool for data visualization and communication of findings.

Seaborn: Seaborn is a statistical data visualization library built on top of Matplotlib. It provides a high-level interface for creating aesthetically pleasing and informative statistical graphics. Seaborn simplifies the process of generating complex visualizations, including heatmaps, pair plots, and violin plots, while maintaining customization options for advanced users.

Metrics (Accuracy, Classification, Confusion Matrix, ROC AUC): In the context of machine learning evaluation, metrics play a crucial role. Accuracy represents the proportion of correctly classified instances, serving as a fundamental measure of model performance. Classification metrics, such as precision, recall, and F1-score, provide insights into the model's ability to correctly identify instances of a particular class. The confusion matrix presents a comprehensive summary of true positive, true negative, false positive, and false negative predictions. Lastly, the ROC AUC (Receiver Operating Characteristic - Area Under the Curve) is a performance metric for binary classification models, illustrating the trade-off between sensitivity and specificity across different thresholds, providing a holistic view of the model's discriminatory power. These metrics collectively aid in assessing and optimizing the performance of machine learning models.

Tkinter is a Python library used for creating graphical user interfaces (GUIs) with ease. It provides a simple and intuitive way to design and interact with windows, buttons, text boxes, menus, and other GUI components. Tkinter is built on top of the Tk GUI toolkit, offering a powerful yet beginner-friendly framework for developing desktop applications. With its rich set of widgets and straightforward syntax, developers can quickly prototype and build applications for various purposes, ranging from simple utilities to complex software projects, making it a popular choice for Python GUI development.

The line from tkinter import Button, Label, Frame imports specific classes (Button, Label, Frame) from the Tkinter module in Python, allowing direct access to these GUI components without needing to reference the module name each time they are used. The Button class represents a clickable button widget that triggers actions when clicked, the Label class is used to display text or images, and the Frame class serves as a container to organize and group other widgets within a window or application. By importing these classes, developers can efficiently create and manipulate buttons, labels, and frames to design interactive graphical interfaces using Tkinter in Python.

he Sequential class from the Keras library, a high-level neural networks API. The Sequential model is a linear stack of layers where each layer has exactly one input tensor and one output tensor. This class serves as the foundation for building various types of neural network architectures, such as feedforward networks and convolutional neural networks (CNNs). With the Sequential model, developers can easily add layers to the network one by one, specifying the configuration and parameters for each layer, including activation functions, regularization techniques, and optimization algorithms. This modular approach simplifies the process of constructing complex neural networks by providing a clear and intuitive interface for defining the network architecture.

Once imported, developers can instantiate an instance of the Sequential class and start adding layers to build their neural network model. By leveraging the Sequential model in Keras, users can quickly prototype, train, and evaluate deep learning models for a wide range of tasks, including image classification, natural language processing, and regression analysis. Additionally, Keras provides a user-friendly interface with extensive documentation and support, making it accessible to both beginners and experienced practitioners in the field of machine learning and artificial intelligence.

The train\_test\_split function takes input arrays (or matrices) representing the features and target variables, along with optional parameters such as test size, random state, and stratification, and returns four arrays: X\_train, X\_test, y\_train, and y\_test. The X\_train and X\_test arrays contain the feature values for the training and testing sets, respectively, while the y\_train and y\_test arrays contain the corresponding target values.

By utilizing train\_test\_split, developers can easily partition their dataset into separate training and testing sets, which is essential for evaluating the performance of machine learning models. The training set is used to train the model on the available data, while the testing set is used to assess how well the trained model generalizes to unseen data. This practice helps in detecting issues like overfitting, where the model performs well on the training data but fails to generalize to new data. Moreover, train\_test\_split supports various sampling techniques, including stratified splitting for classification tasks, enabling practitioners to create representative training and testing sets that preserve the distribution of the target variable, thus ensuring robust model evaluation.

The StandardScaler is a preprocessing technique used in machine learning pipelines to standardize features by removing the mean and scaling them to unit variance. This process transforms the distribution of each feature to have a mean of zero and a standard deviation of one. In Python, StandardScaler is typically found in the sklearn.preprocessing module of the scikit-learn library. To use StandardScaler, you first instantiate an instance of the scaler, then fit it to your training data to compute the mean and standard deviation of each feature. Finally, you transform both the training and testing datasets using the computed statistics to ensure consistency across data splits.

Standardizing features with StandardScaler is crucial, especially when dealing with algorithms that are sensitive to feature scaling, such as support vector machines (SVMs), k-nearest neighbors (KNN), and neural networks. By bringing all features to the same scale, StandardScaler prevents certain features from dominating others due to their larger magnitude, thus ensuring that the model can effectively learn from all features without bias. Additionally, standardization can aid in speeding up convergence during the optimization process, leading to faster training times and potentially better model performance.

Despite its benefits, it's important to note that standardization assumes that the features follow a normal distribution. If the data deviates significantly from this assumption, alternative scaling methods like MinMaxScaler or RobustScaler may be more appropriate. Moreover, StandardScaler should be applied only to numerical features, as categorical variables or features with a meaningful ordinal relationship could be distorted by standardization. Overall, StandardScaler serves as a fundamental preprocessing step in many machine learning workflows, contributing to improved model stability, interpretability, and generalization.

FigureCanvasTkAgg is a class in the Matplotlib library that provides a bridge between Matplotlib figures and Tkinter applications, allowing Matplotlib plots to be embedded seamlessly within Tkinter GUIs. This integration enables developers to create interactive data visualization applications with Tkinter while leveraging the powerful plotting capabilities of Matplotlib. When using FigureCanvasTkAgg, developers first create a Matplotlib figure object representing the plot they want to display, then instantiate a FigureCanvasTkAgg object, passing the figure as an argument. Finally, the FigureCanvasTkAgg object is added to a Tkinter window or frame, allowing the Matplotlib plot to be rendered within the GUI.

By using FigureCanvasTkAgg, developers can enhance the user experience of their Tkinter applications by providing dynamic and customizable visualizations directly within the interface. This capability is particularly useful for displaying complex data sets or real-time data streams in interactive dashboards, scientific applications, or educational tools. Moreover, FigureCanvasTkAgg offers flexibility in terms of layout and styling, allowing developers to seamlessly integrate Matplotlib plots with other Tkinter widgets such as buttons, labels, and entry fields, thereby creating rich and informative graphical user interfaces.

In Python, PIL (Python Imaging Library) is a library commonly used for working with images. The lines from PIL import Image, ImageTk import two key classes from the PIL library. Image is a class that represents an image in memory and provides various methods for manipulating, processing, and analyzing images. With Image, developers can open, save, resize, crop, and apply transformations to images, as well as perform operations such as filtering, enhancing, and converting between different image formats.

On the other hand, ImageTk is a module within PIL that provides utilities for integrating PIL images with Tkinter, the standard GUI toolkit for Python. Specifically, ImageTk allows developers to convert Image objects into Tkinter-compatible image objects (PhotoImage objects) that can be displayed within Tkinter widgets such as labels, buttons, and canvases. This integration enables developers to seamlessly incorporate images into Tkinter-based graphical user interfaces, facilitating the creation of visually appealing applications that utilize images for illustration, decoration, or information display.

Overall, by importing Image and ImageTk, developers gain access to a powerful set of tools for working with images in Python, including capabilities for image manipulation and processing with PIL as well as seamless integration of images into Tkinter GUIs with ImageTk, ultimately enabling the development of diverse image-centric applications ranging from image viewers and editors to computer vision systems and multimedia applications.

The "Dense" layer, also known as a fully connected layer, is one of the fundamental building blocks of neural networks. In a Dense layer, every neuron in the layer is connected to every neuron in the previous layer, forming a dense matrix of connections. Each connection is associated with a weight parameter, and the output of each neuron is calculated by applying an activation function to the weighted sum of inputs plus a bias term. Dense layers are versatile and can be used for various tasks, such as classification, regression, and feature learning, making them a foundational component in most neural network architectures.

"Dropout" is a regularization technique commonly applied in neural networks to prevent overfitting and improve generalization performance. During training, Dropout randomly "drops out" (sets to zero) a fraction of the neurons in a layer, effectively disabling them temporarily. This dropout process is applied independently to each neuron with a certain probability, typically specified as a hyperparameter. By randomly deactivating neurons during training, Dropout forces the network to learn redundant representations and prevents co-adaptation among neurons, thus promoting more robust and generalized feature learning. Dropout is particularly effective in deep neural networks where overfitting is a common issue due to the large number of parameters.

In practice, these layers are typically used together within a neural network architecture. For example, a typical neural network model might consist of multiple Dense layers followed by Dropout layers. The Dense layers perform feature extraction and transformation, while Dropout layers help regularize the network and prevent overfitting by randomly dropping out neurons. This combination of Dense and Dropout layers allows neural networks to effectively learn complex patterns from data while controlling for overfitting, resulting in models that generalize well to unseen data and exhibit robust performance in various machine learning tasks. Overall, Dense and Dropout layers are essential components in the construction of deep learning models, enabling the development of powerful and flexible neural network architectures for diverse applications in fields such as computer vision, natural language processing, and reinforcement learning.

Keras is a high-level neural networks API written in Python, capable of running on top of various deep learning frameworks such as TensorFlow, Microsoft Cognitive Toolkit (CNTK), and Theano. Its primary goal is to provide a user-friendly interface for building and training deep learning models, allowing developers to quickly prototype and deploy neural networks without having to deal with low-level implementation details. Keras offers a modular and intuitive approach to constructing neural network architectures, allowing users to easily define and configure layers, activation functions, optimization algorithms, loss functions, and other components of the model.

With Keras, developers can build a wide range of neural network models, including convolutional neural networks (CNNs) for image classification, recurrent neural networks (RNNs) for sequence processing, and deep feedforward networks for regression tasks. Keras provides a consistent and straightforward API that abstracts away the complexities of deep learning, making it accessible to both beginners and experienced practitioners alike. Moreover, Keras emphasizes ease of use, readability, and extensibility, enabling rapid experimentation and iteration in the development of cutting-edge deep learning applications. Overall, Keras has become one of the most popular and widely used deep learning frameworks due to its simplicity, flexibility, and powerful capabilities in building and training state-of-the-art neural network models.

Tkinter is a Python library used for creating graphical user interfaces (GUIs) with ease. It provides a simple and intuitive way to design and interact with windows, buttons, text boxes, menus, and other GUI components. Tkinter is built on top of the Tk GUI toolkit, offering a powerful yet beginner-friendly framework for developing desktop applications. With its rich set of widgets and straightforward syntax, developers can quickly prototype and build applications for various purposes, ranging from simple utilities to complex software projects, making it a popular choice for Python GUI development.

The line from tkinter import Button, Label, Frame imports specific classes (Button, Label, Frame) from the Tkinter module in Python, allowing direct access to these GUI components without needing to reference the module name each time they are used. The Button class represents a clickable button widget that triggers actions when clicked, the Label class is used to display text or images, and the Frame class serves as a container to organize and group other widgets within a window or application. By importing these classes, developers can efficiently create and manipulate buttons, labels, and frames to design interactive graphical interfaces using Tkinter in Python.

he Sequential class from the Keras library, a high-level neural networks API. The Sequential model is a linear stack of layers where each layer has exactly one input tensor and one output tensor. This class serves as the foundation for building various types of neural network architectures, such as feedforward networks and convolutional neural networks (CNNs). With the Sequential model, developers can easily add layers to the network one by one, specifying the configuration and parameters for each layer, including activation functions, regularization techniques, and optimization algorithms. This modular approach simplifies the process of constructing complex neural networks by providing a clear and intuitive interface for defining the network architecture.

Once imported, developers can instantiate an instance of the Sequential class and start adding layers to build their neural network model. By leveraging the Sequential model in Keras, users can quickly prototype, train, and evaluate deep learning models for a wide range of tasks, including image classification, natural language processing, and regression analysis. Additionally, Keras provides a user-friendly interface with extensive documentation and support, making it accessible to both beginners and experienced practitioners in the field of machine learning and artificial intelligence.

The train\_test\_split function takes input arrays (or matrices) representing the features and target variables, along with optional parameters such as test size, random state, and stratification, and returns four arrays: X\_train, X\_test, y\_train, and y\_test. The X\_train and X\_test arrays contain the feature values for the training and testing sets, respectively, while the y\_train and y\_test arrays contain the corresponding target values.

By utilizing train\_test\_split, developers can easily partition their dataset into separate training and testing sets, which is essential for evaluating the performance of machine learning models. The training set is used to train the model on the available data, while the testing set is used to assess how well the trained model generalizes to unseen data. This practice helps in detecting issues like overfitting, where the model performs well on the training data but fails to generalize to new data. Moreover, train\_test\_split supports various sampling techniques, including stratified splitting for classification tasks, enabling practitioners to create representative training and testing sets that preserve the distribution of the target variable, thus ensuring robust model evaluation.

The StandardScaler is a preprocessing technique used in machine learning pipelines to standardize features by removing the mean and scaling them to unit variance. This process transforms the distribution of each feature to have a mean of zero and a standard deviation of one. In Python, StandardScaler is typically found in the sklearn.preprocessing module of the scikit-learn library. To use StandardScaler, you first instantiate an instance of the scaler, then fit it to your training data to compute the mean and standard deviation of each feature. Finally, you transform both the training and testing datasets using the computed statistics to ensure consistency across data splits.

Standardizing features with StandardScaler is crucial, especially when dealing with algorithms that are sensitive to feature scaling, such as support vector machines (SVMs), k-nearest neighbors (KNN), and neural networks. By bringing all features to the same scale, StandardScaler prevents certain features from dominating others due to their larger magnitude, thus ensuring that the model can effectively learn from all features without bias. Additionally, standardization can aid in speeding up convergence during the optimization process, leading to faster training times and potentially better model performance.

Despite its benefits, it's important to note that standardization assumes that the features follow a normal distribution. If the data deviates significantly from this assumption, alternative scaling methods like MinMaxScaler or RobustScaler may be more appropriate. Moreover, StandardScaler should be applied only to numerical features, as categorical variables or features with a meaningful ordinal relationship could be distorted by standardization. Overall, StandardScaler serves as a fundamental preprocessing step in many machine learning workflows, contributing to improved model stability, interpretability, and generalization.

FigureCanvasTkAgg is a class in the Matplotlib library that provides a bridge between Matplotlib figures and Tkinter applications, allowing Matplotlib plots to be embedded seamlessly within Tkinter GUIs. This integration enables developers to create interactive data visualization applications with Tkinter while leveraging the powerful plotting capabilities of Matplotlib. When using FigureCanvasTkAgg, developers first create a Matplotlib figure object representing the plot they want to display, then instantiate a FigureCanvasTkAgg object, passing the figure as an argument. Finally, the FigureCanvasTkAgg object is added to a Tkinter window or frame, allowing the Matplotlib plot to be rendered within the GUI.

By using FigureCanvasTkAgg, developers can enhance the user experience of their Tkinter applications by providing dynamic and customizable visualizations directly within the interface. This capability is particularly useful for displaying complex data sets or real-time data streams in interactive dashboards, scientific applications, or educational tools. Moreover, FigureCanvasTkAgg offers flexibility in terms of layout and styling, allowing developers to seamlessly integrate Matplotlib plots with other Tkinter widgets such as buttons, labels, and entry fields, thereby creating rich and informative graphical user interfaces.

backends.backend\_tkagg refers to the backend renderer used by Matplotlib when generating plots to be displayed within a Tkinter GUI application. Specifically, backend\_tkagg utilizes the Tkinter library to render Matplotlib figures onto Tkinter widgets, such as frames or canvases, within the graphical user interface. This integration allows developers to seamlessly embed Matplotlib plots into Tkinter-based applications, enabling the creation of interactive data visualization tools, dashboards, and scientific applications. By leveraging backend\_tkagg, developers can take advantage of both Matplotlib's extensive plotting capabilities and Tkinter's intuitive GUI framework, facilitating the development of rich and visually appealing graphical interfaces for analyzing and presenting data.

In Python, PIL (Python Imaging Library) is a library commonly used for working with images. The lines from PIL import Image, ImageTk import two key classes from the PIL library. Image is a class that represents an image in memory and provides various methods for manipulating, processing, and analyzing images. With Image, developers can open, save, resize, crop, and apply transformations to images, as well as perform operations such as filtering, enhancing, and converting between different image formats.

On the other hand, ImageTk is a module within PIL that provides utilities for integrating PIL images with Tkinter, the standard GUI toolkit for Python. Specifically, ImageTk allows developers to convert Image objects into Tkinter-compatible image objects (PhotoImage objects) that can be displayed within Tkinter widgets such as labels, buttons, and canvases. This integration enables developers to seamlessly incorporate images into Tkinter-based graphical user interfaces, facilitating the creation of visually appealing applications that utilize images for illustration, decoration, or information display.

Overall, by importing Image and ImageTk, developers gain access to a powerful set of tools for working with images in Python, including capabilities for image manipulation and processing with PIL as well as seamless integration of images into Tkinter GUIs with ImageTk, ultimately enabling the development of diverse image-centric applications ranging from image viewers and editors to computer vision systems and multimedia applications.

The "Dense" layer, also known as a fully connected layer, is one of the fundamental building blocks of neural networks. In a Dense layer, every neuron in the layer is connected to every neuron in the previous layer, forming a dense matrix of connections. Each connection is associated with a weight parameter, and the output of each neuron is calculated by applying an activation function to the weighted sum of inputs plus a bias term. Dense layers are versatile and can be used for various tasks, such as classification, regression, and feature learning, making them a foundational component in most neural network architectures.

"Dropout" is a regularization technique commonly applied in neural networks to prevent overfitting and improve generalization performance. During training, Dropout randomly "drops out" (sets to zero) a fraction of the neurons in a layer, effectively disabling them temporarily. This dropout process is applied independently to each neuron with a certain probability, typically specified as a hyperparameter. By randomly deactivating neurons during training, Dropout forces the network to learn redundant representations and prevents co-adaptation among neurons, thus promoting more robust and generalized feature learning. Dropout is particularly effective in deep neural networks where overfitting is a common issue due to the large number of parameters.

In practice, these layers are typically used together within a neural network architecture. For example, a typical neural network model might consist of multiple Dense layers followed by Dropout layers. The Dense layers perform feature extraction and transformation, while Dropout layers help regularize the network and prevent overfitting by randomly dropping out neurons. This combination of Dense and Dropout layers allows neural networks to effectively learn complex patterns from data while controlling for overfitting, resulting in models that generalize well to unseen data and exhibit robust performance in various machine learning tasks. Overall, Dense and Dropout layers are essential components in the construction of deep learning models, enabling the development of powerful and flexible neural network architectures for diverse applications in fields such as computer vision, natural language processing, and reinforcement learning.

Keras is a high-level neural networks API written in Python, capable of running on top of various deep learning frameworks such as TensorFlow, Microsoft Cognitive Toolkit (CNTK), and Theano. Its primary goal is to provide a user-friendly interface for building and training deep learning models, allowing developers to quickly prototype and deploy neural networks without having to deal with low-level implementation details. Keras offers a modular and intuitive approach to constructing neural network architectures, allowing users to easily define and configure layers, activation functions, optimization algorithms, loss functions, and other components of the model.

With Keras, developers can build a wide range of neural network models, including convolutional neural networks (CNNs) for image classification, recurrent neural networks (RNNs) for sequence processing, and deep feedforward networks for regression tasks. Keras provides a consistent and straightforward API that abstracts away the complexities of deep learning, making it accessible to both beginners and experienced practitioners alike. Moreover, Keras emphasizes ease of use, readability, and extensibility, enabling rapid experimentation and iteration in the development of cutting-edge deep learning applications. Overall, Keras has become one of the most popular and widely used deep learning frameworks due to its simplicity, flexibility, and powerful capabilities in building and training state-of-the-art neural network models.

**CODING**

import pandas as pd

import numpy as np

from keras.models import Sequential

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import roc\_auc\_score, confusion\_matrix

from keras.layers import GRU, Dropout, Dense,BatchNormalization

import seaborn as sns

from sklearn.metrics import accuracy\_score, f1\_score, precision\_score, recall\_score, classification\_report, confusion\_matrix, roc\_auc\_score

data = pd.read\_csv("/content/sample\_data/smart\_grid\_stability\_augmented.csv")

data.info()

data

data=data.replace(np.nan,0)

# check missing values in variables

data.isnull().sum()

data.stabf.value\_counts()

# Map 'unstable' to 1 and 'stable' to 0

data['stabf'] = data['stabf'].map({'unstable': 1, 'stable': 0})

# Convert the 'stabf' column to numeric

data['stabf'] = pd.to\_numeric(data['stabf'], errors='coerce')

# Display the updated DataFrame

print(data.head())

X = data.drop(['stabf'], axis=1)

y = data['stabf']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# Build GRU model

model = Sequential()

model.add(GRU(units=128, input\_shape=(X\_train\_scaled.shape[1], 1), return\_sequences=True))

model.add(Dropout(0.2))

model.add(GRU(units=64))

model.add(Dropout(0.1))

model.add(Dense(units=1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Train the model

history= model.fit(X\_train\_scaled, y\_train, epochs=5, batch\_size=32, validation\_split=0.1)

y\_pred=model.predict(X\_test\_scaled)

accu=accuracy\_score(y\_test,y\_pred.round())

print("Accuracy Score :")

print(accu)

# Calculate the ROC Precision, Recall, and F1-Score

from sklearn.metrics import roc\_auc\_score, roc\_curve, auc, precision\_recall\_fscore\_support

import matplotlib.pyplot as plt

# Calculate the AUC

auc = roc\_auc\_score(y\_test, y\_pred.round())

print('AUC: %.2f' % auc)

# Calculate the ROC

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred)

# plot the roc curve

plt.plot(fpr, tpr)

plt.title('ROC Curve')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.show()

# Calculate the confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred.round())

# Print the confusion matrix

print('Confusion Matrix:')

print(conf\_matrix)

plt.figure(figsize=(8, 6))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.show()

from sklearn.metrics import classification\_report

print(classification\_report(y\_test, y\_pred.round()))

# Plotting a heatmap for precision, recall, and F1-score

class\_report = classification\_report(y\_test, y\_pred.round(), output\_dict=True)

class\_names = [str(label) for label in class\_report.keys() if label not in ['accuracy', 'macro avg', 'weighted avg']]

# Extract precision, recall, and F1-score for each class

heatmap\_data = [[class\_report[class\_name]['precision'], class\_report[class\_name]['recall'], class\_report[class\_name]['f1-score']] for class\_name in class\_names]

# Create a heatmap

fig, ax = plt.subplots(figsize=(10, 6))

sns.heatmap(heatmap\_data, annot=True, fmt=".2f", xticklabels=['Precision', 'Recall', 'F1-Score'], yticklabels=class\_names, cmap='Blues')

plt.title('Classification Report Heatmap')

plt.show()

import matplotlib.pyplot as plt

# Train the model

#history = model.fit(X\_train\_scaled, y\_train, epochs=5, batch\_size=32, validation\_split=0.1)

# Plot training history

plt.figure(figsize=(12, 6))

# Plot training & validation accuracy values

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'])

plt.plot(history.history['val\_accuracy'])

plt.title('Model Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend(['Train', 'Validation'], loc='upper left')

# Plot training & validation loss values

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'])

plt.plot(history.history['val\_loss'])

plt.title('Model Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.tight\_layout()

plt.show()

FRAME WORK CODE

import pandas as pd

import numpy as np

from keras.models import Sequential

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import roc\_auc\_score, confusion\_matrix

from keras.layers import GRU, Dropout, Dense,BatchNormalization

import seaborn as sns

from sklearn.metrics import accuracy\_score, f1\_score, precision\_score, recall\_score, classification\_report, confusion\_matrix, roc\_auc\_score

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

from PIL import Image, ImageTk

# Load your dataset here

data = pd.read\_csv("features.csv")

# Map 'unstable' to 1 and 'stable' to 0

data['stabf'] = data['stabf'].map({'unstable': 1, 'stable': 0})

# Convert the 'stabf' column to numeric

data['stabf'] = pd.to\_numeric(data['stabf'], errors='coerce')

# Display the updated DataFrame

print(data.head())

X = data.drop(['stabf'], axis=1)

y = data['stabf']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# Build GRU model

model = Sequential()

model.add(GRU(units=128, input\_shape=(X\_train\_scaled.shape[1], 1), return\_sequences=True))

model.add(Dropout(0.2))

model.add(GRU(units=64))

model.add(Dropout(0.1))

model.add(Dense(units=1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Tkinter GUI

root = tk.Tk()

root.title("Model Training and Evaluation")

root.geometry("400x400")

# Load background image

background\_image = Image.open("sample1.jpg") # Replace with your image file

background\_photo = ImageTk.PhotoImage(background\_image)

background\_label = tk.Label(root, image=background\_photo)

background\_label.place(relwidth=1, relheight=1)

# Project label

project\_label = tk.Label(root, text="Artificial Intelligence theft identification systems for Multi-source and Multi-powered Smart grid applications", font=("Helvetica", 12), bg="white")

project\_label.pack(pady=10)

# Labels for dataset information

r\_dataset\_label = tk.Label(root, text="Dataset: SIOT", font=("Helvetica", 11),foreground="blue",width=20)

r\_dataset\_label.pack(pady=10, padx=10)

# Training Data Label

r\_train\_data\_label = tk.Label(root, text="Training Data: 70%", font=("Helvetica", 11),foreground="blue",width=20)

r\_train\_data\_label.pack(pady=10, padx=10)

# Testing Data Label

r\_test\_data\_label = tk.Label(root, text="Testing Data: 30%", font=("Helvetica", 11), foreground="blue",width=20)

r\_test\_data\_label.pack(pady=10, padx=10)

# Function to train the model

def train\_model():

global model, X\_train\_scaled, y\_train

history = model.fit(X\_train\_scaled, y\_train, epochs=5, batch\_size=32, validation\_split=0.1)

# Function to display accuracy chart

def display\_accuracy():

global model, X\_test\_scaled, y\_test

y\_pred = model.predict(X\_test\_scaled)

accu = accuracy\_score(y\_test, y\_pred.round())

print("Accuracy Score:", accu)

# Plot Accuracy

plt.figure(figsize=(6, 4))

plt.bar(["Accuracy"], [accu], color='blue')

plt.title('Model Accuracy')

plt.ylabel('Accuracy')

plt.show()

# Function to display confusion matrix

def display\_confusion\_matrix():

global model, X\_test\_scaled, y\_test

y\_pred = model.predict(X\_test\_scaled)

conf\_matrix = confusion\_matrix(y\_test, y\_pred.round())

print("Confusion Matrix:")

print(conf\_matrix)

plt.figure(figsize=(8, 6))

sns.heatmap(conf\_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)

plt.title('Confusion Matrix')

plt.xlabel('Predicted')

plt.ylabel('True')

plt.show()

# Function to display classification report

def display\_classification\_report():

global model, X\_test\_scaled, y\_test

y\_pred = model.predict(X\_test\_scaled)

classif = classification\_report(y\_test, y\_pred.round())

print("Classification report:")

print(classif)

# Plotting a heatmap for precision, recall, and F1-score

class\_report = classification\_report(y\_test, y\_pred.round(), output\_dict=True)

class\_names = [str(label) for label in class\_report.keys() if label not in ['accuracy', 'macro avg', 'weighted avg']]

# Extract precision, recall, and F1-score for each class

heatmap\_data = [[class\_report[class\_name]['precision'], class\_report[class\_name]['recall'],

class\_report[class\_name]['f1-score']] for class\_name in class\_names]

# Create a heatmap

fig, ax = plt.subplots(figsize=(10, 6))

sns.heatmap(heatmap\_data, annot=True, fmt=".2f", xticklabels=['Precision', 'Recall', 'F1-Score'],

yticklabels=class\_names, cmap='Blues')

plt.title('Classification Report Heatmap')

plt.show()

# Function to display AUC-ROC curve

def display\_auc\_roc\_curve():

global model, X\_test\_scaled, y\_test

y\_pred = model.predict(X\_test\_scaled)

auc = roc\_auc\_score(y\_test, y\_pred.round())

print('AUC: %.2f' % auc)

# Calculate the ROC

fpr, tpr, thresholds = roc\_curve(y\_test, y\_pred)

# plot the roc curve

plt.plot(fpr, tpr)

plt.title('ROC Curve')

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.show()

# Function to display overall training model details

def display\_overall\_training\_details():

global history

# Plot training history

plt.figure(figsize=(12, 6))

# Plot training & validation accuracy values

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'])

plt.plot(history.history['val\_accuracy'])

plt.title('Model Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend(['Train', 'Validation'], loc='upper left')

# Plot training & validation loss values

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'])

plt.plot(history.history['val\_loss'])

plt.title('Model Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.tight\_layout()

plt.show()

# Train Button

train\_button = Button(root, text="Train Model", command=train\_model,width=20)

train\_button.pack(pady=10)

# Accuracy Button

accuracy\_button = Button(root, text="Display Accuracy", command=display\_accuracy,width=20)

accuracy\_button.pack(pady=10)

# Confusion Matrix Button

conf\_matrix\_button = Button(root, text="Display Confusion Matrix", command=display\_confusion\_matrix,width=20)

conf\_matrix\_button.pack(pady=10)

# Classification Report Button

class\_report\_button = Button(root, text="Display Classification Report", command=display\_classification\_report,width=20)

class\_report\_button.pack(pady=10)

# AUC-ROC Curve Button

auc\_roc\_button = Button(root, text="Display AUC-ROC Curve", command=display\_auc\_roc\_curve,width=20)

auc\_roc\_button.pack(pady=10)

# Overall Training Details Button

overall\_details\_button = Button(root, text="Display Overall Training Details", command=display\_overall\_training\_details,width=20)

overall\_details\_button.pack(pady=10)

# Run the Tkinter event loop

root.mainloop()

**RESULTS AND DISCUSSION:**

Dataset:

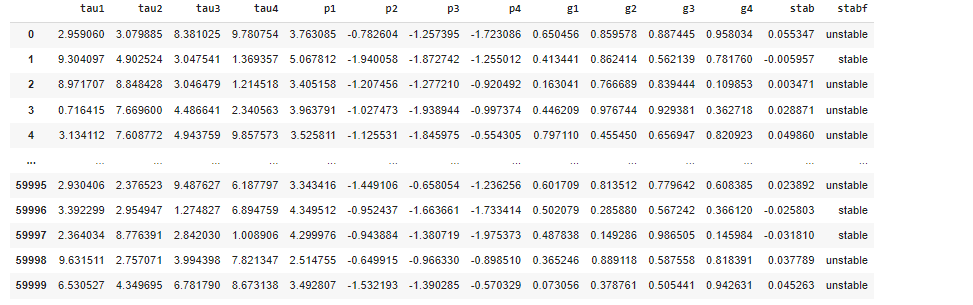


Figure 1: Dataset

Results:

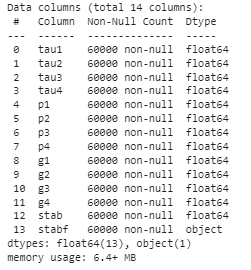


Figure 2: dataset information



Figure 3: Replacing NAN values by Zero

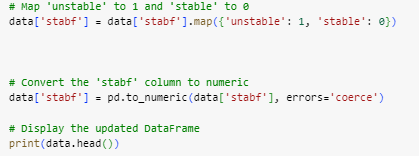


Figure 4: Labeling 0 and 1 to the column stabf

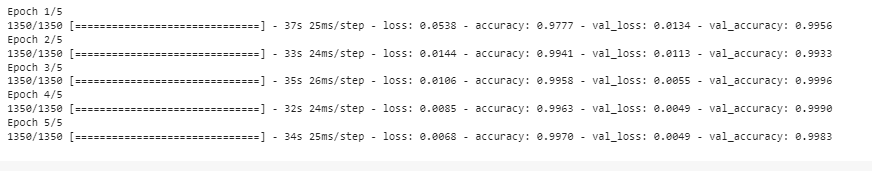


Figure 5: dataset is trained with five epochs with the validation split



Figure 6: predict value is calculated



Figure 7: Accuracy has been calculated for the model



Figure 8: Confusion matrix has been calculated

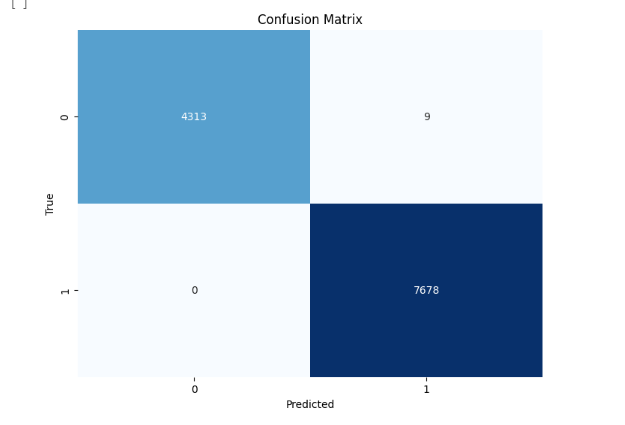


Figure 9: Confusion matrix has been displayed

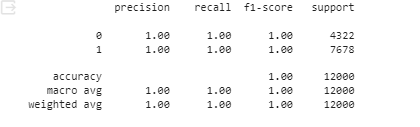


Figure 10: Classification report has been calculated

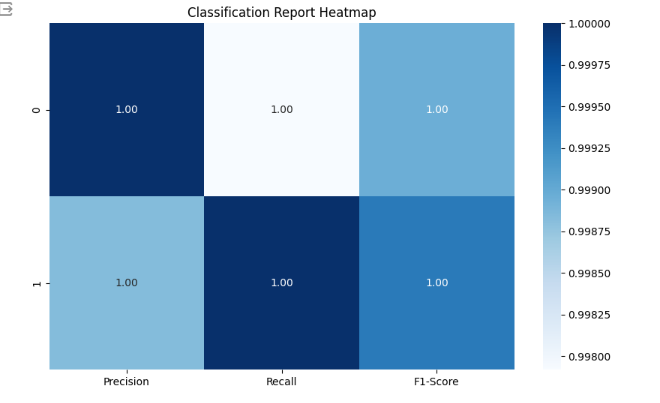


Figure 11: Classification report has been displayed

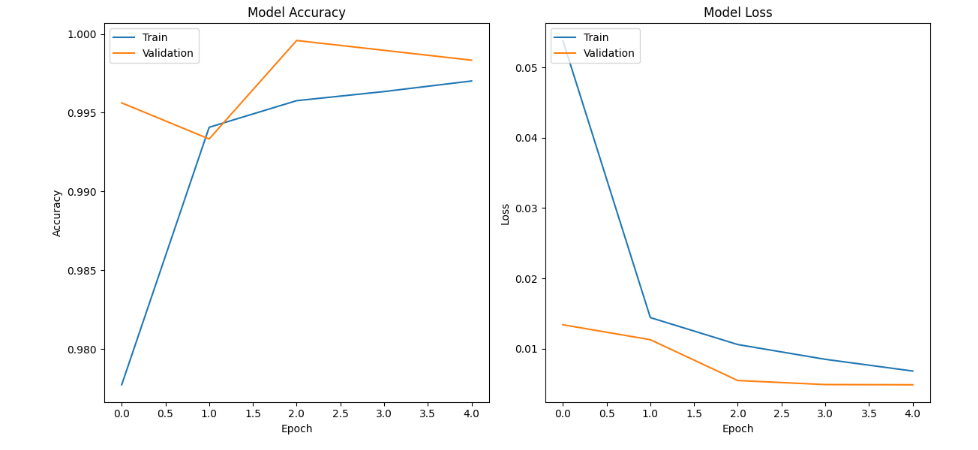


Figure 12: Both model Accuracy and loss

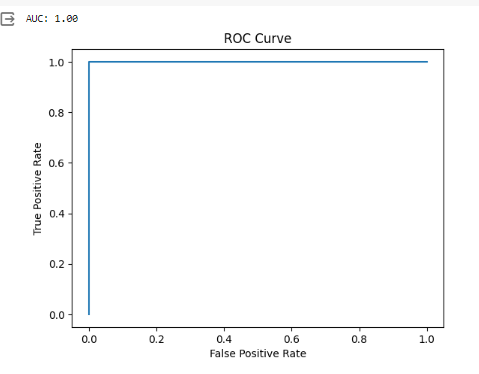


Figure 13: ROC AUC Curve

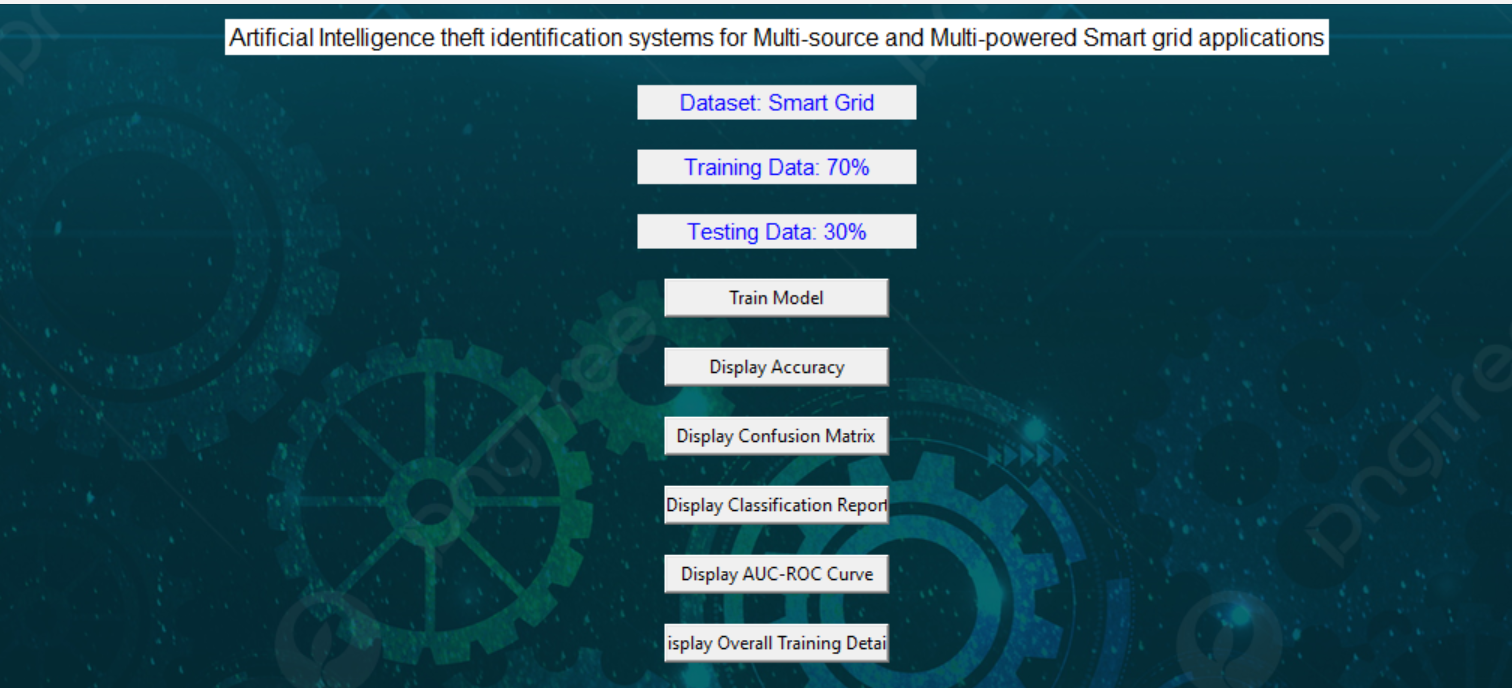


Figure 14: Framework design

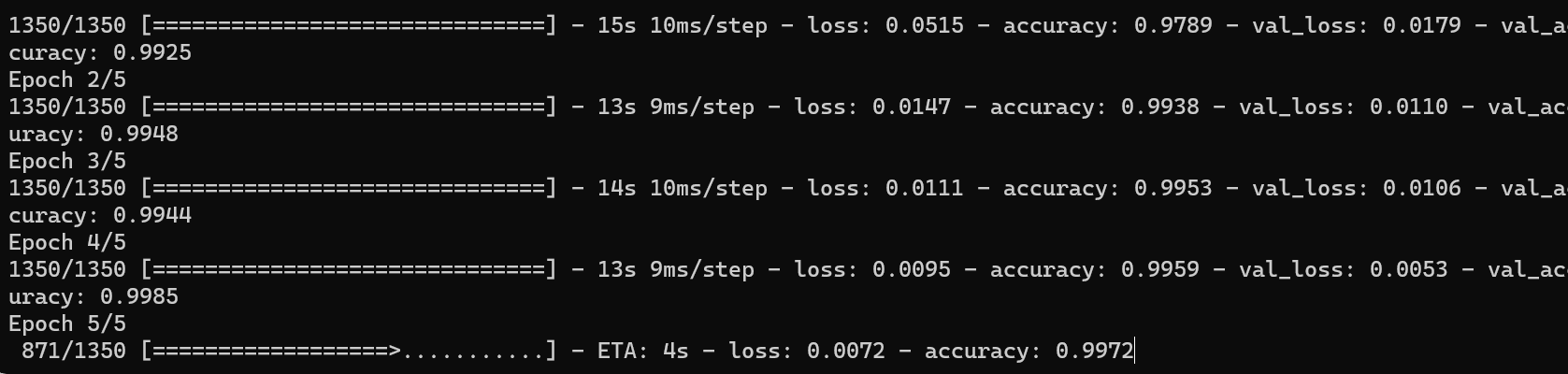


Figure 15: Epoch has been used to train the dataset with algorithm

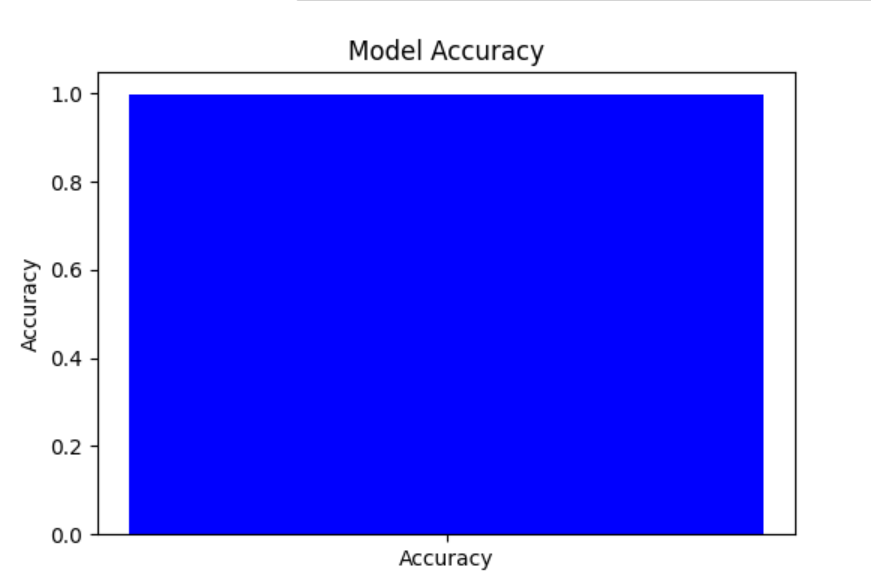


Figure 16: Model accuracy has been displayed through graph

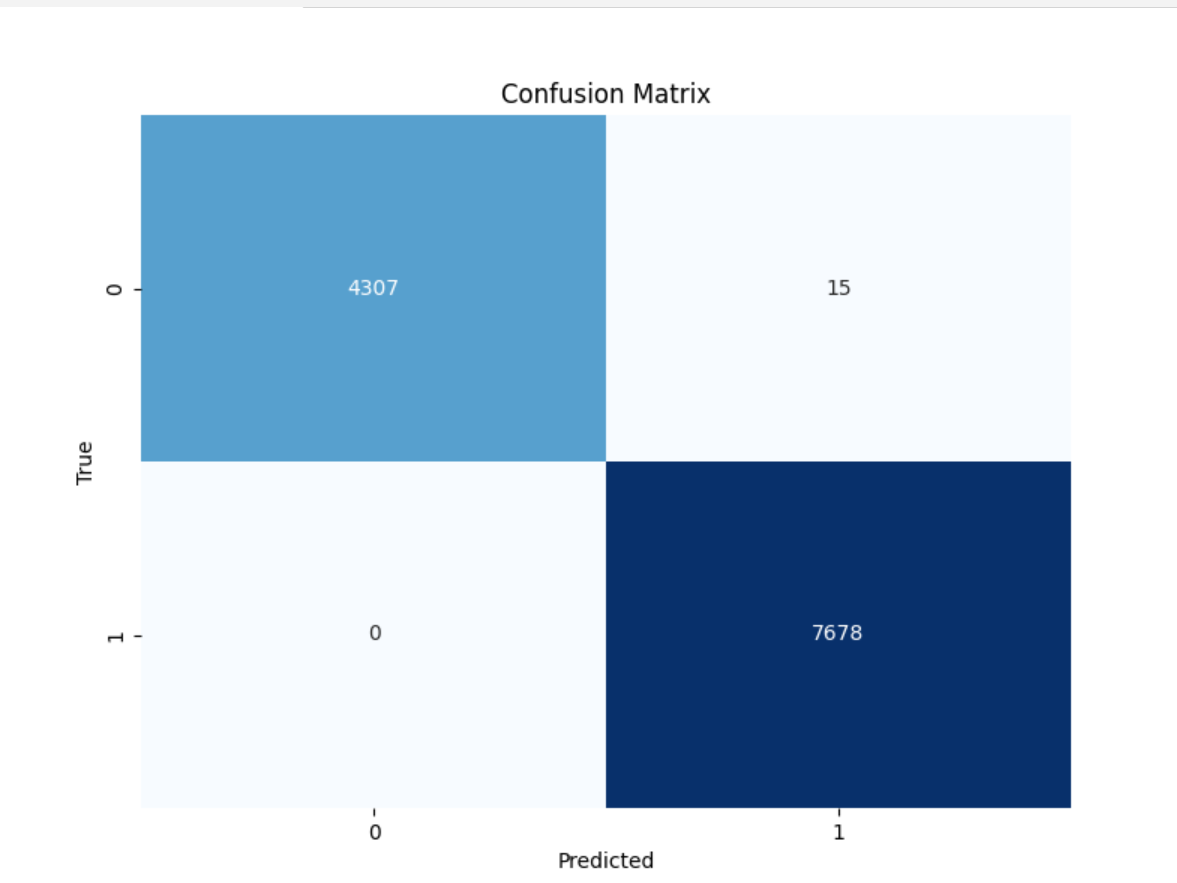


Figure 17: Confusion matrix has been displayed through graph

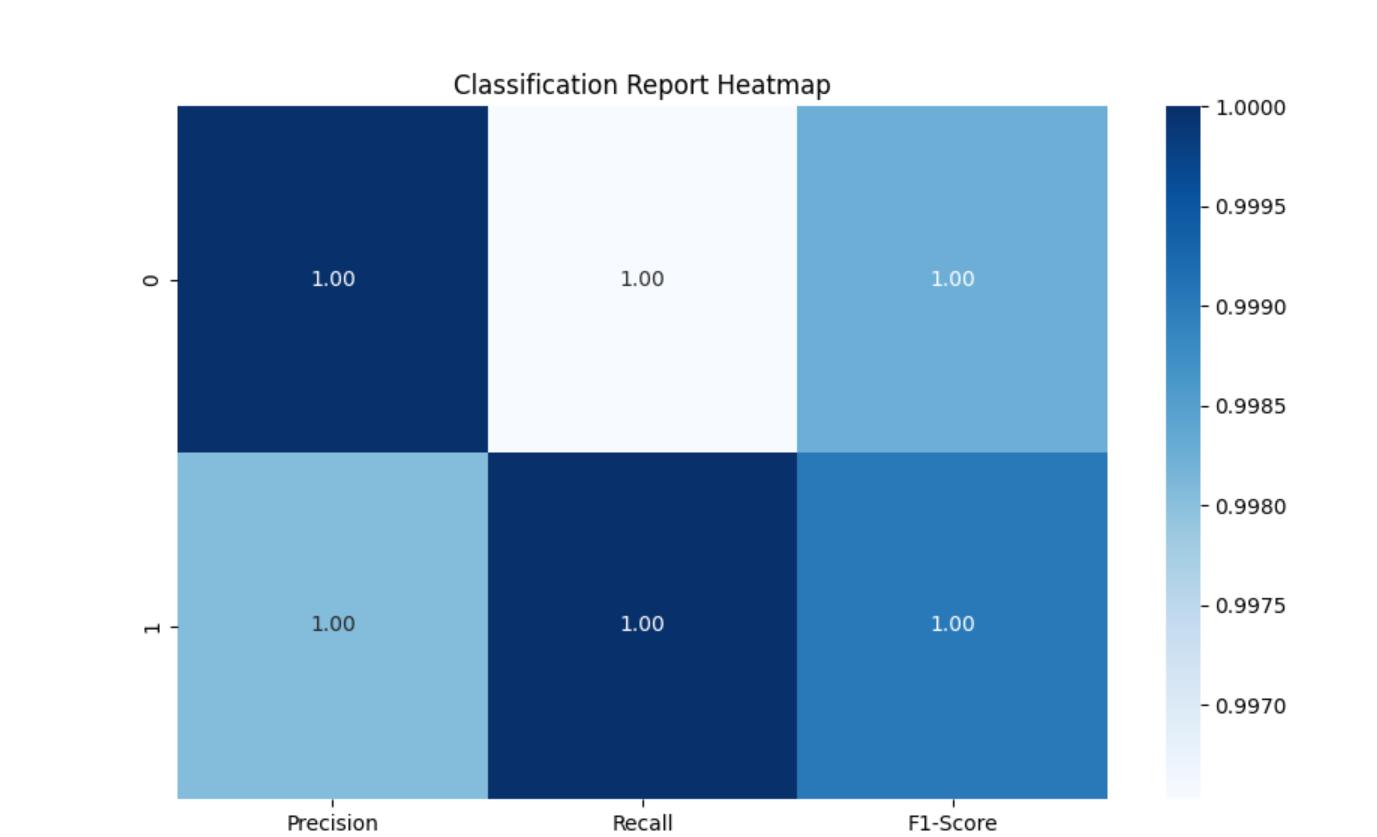


Figure 18: Classification report has been displayed through graph

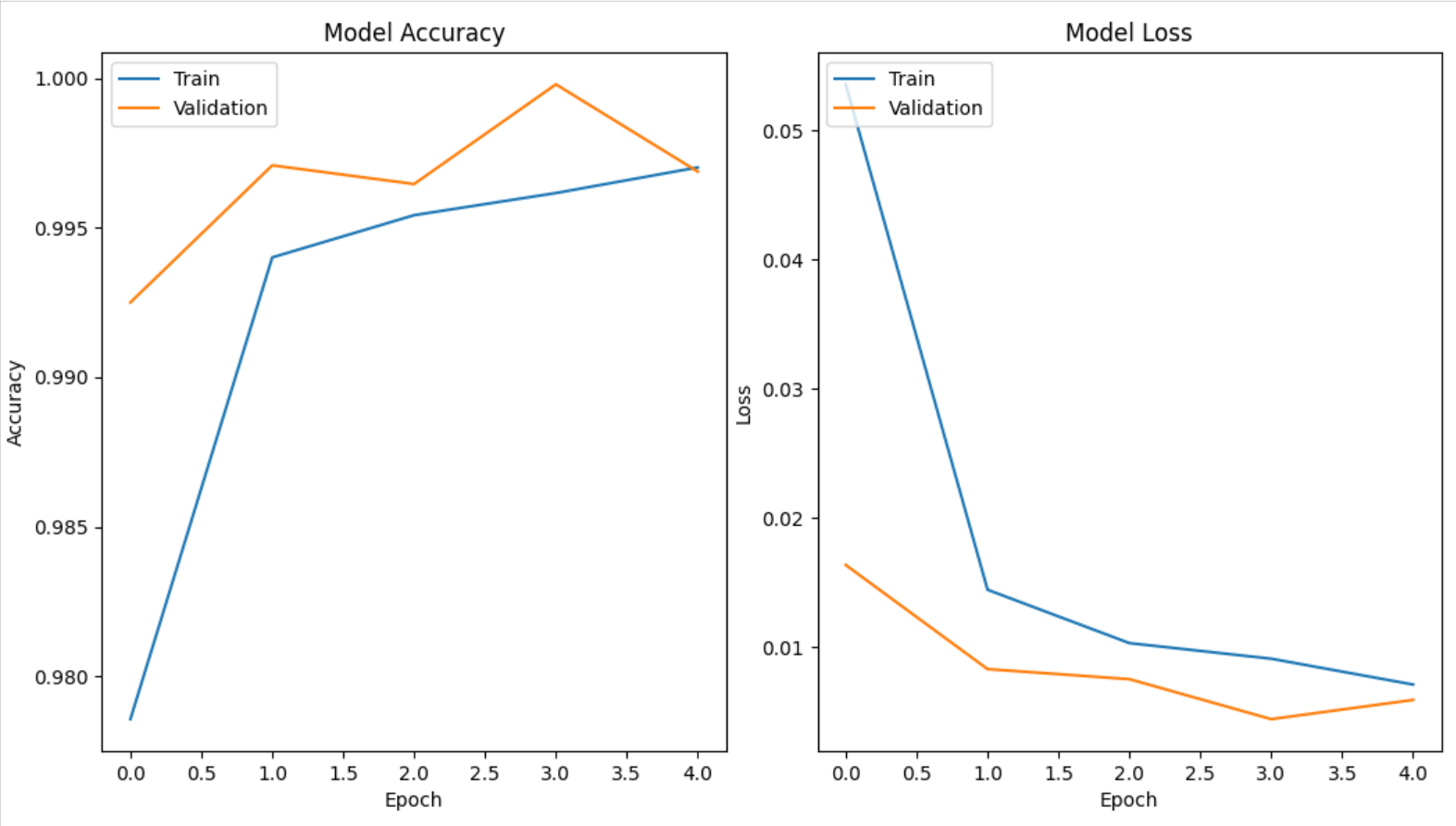


Figure 19: Both model Accuracy and loss had been displayed with graph

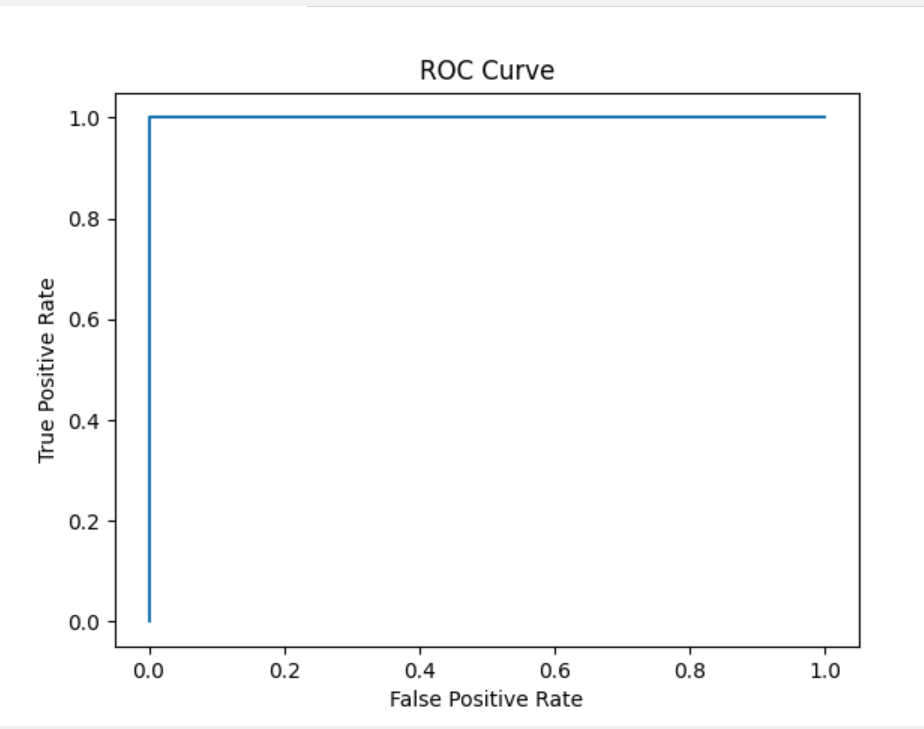


Figure 20: ROC AUC has been displayed

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precision | Recall | F1 – score | Support |
| 0 | 1.00 | 1.00 | 1.00 | 66495 |
| 1 | 0.93 | 0.95 | 0.94 | 988 |
| accuracy |  |  | 1.00 | 67483 |
| Macro avg | 0.96 | 0.98 | 0.97 | 67483 |
| Weighted avg | 1.00 | 1.00 | 1.00 | 67483 |

Table 1: classification report of GRU

The classification report is a performance evaluation tool that shows the precision, recall, f1-score, for each class in a classification problem. In training images using the deep learning model, the classification report would provide information about how well the model performed in classifying images into different categories. The precision represents the percentage of correctly classified images among all the images classified as belonging to a specific class. The recall represents the percentage of correctly classified images among all the images that actually belong to a specific class. The f1-score is a harmonic mean of precision and recall, and support represents the number of images in each class.

The accuracy has been calculated for the model that has been implemented, and the result for the model is compared in Table

|  |  |
| --- | --- |
| Algorithms | Accuracy |
| GRU | 99 |

Table 2: Accuracy of GRU

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithms | Accuracy | Training Time | Epochs |
| GRU | 99% | 10 mins | 3 |

Table 4: Algorithms and its details

|  |  |  |
| --- | --- | --- |
| Dataset Count | Training Value | Testing Value |
| 60000 | 70 | 30 |

Table: Consist of dataset count, Training and Testing percentage.

Splitting a dataset into 70% for training and 30% for testing is a common practice in machine learning for model evaluation and validation. In this scenario, the training set, comprising 70% of the data, is used to train the deep learning model on patterns and relationships present in the data. The model learns from the training data to generalize and make predictions on unseen data. The testing set, consisting of the remaining 30% of the data, serves as an independent dataset to evaluate the performance of the trained model. By assessing the model's performance on the testing set, such as measuring accuracy, precision, recall, and F1-score, practitioners can gauge how well the model generalizes to new, unseen data and identify any overfitting or underfitting issues. This split helps ensure that the model's performance estimates are reliable and reflective of its ability to make accurate predictions in real-world scenarios

**CONCLUSION**

The integration of Gated Recurrent Unit (GRU) networks within Artificial Intelligence-based theft identification systems for Multi-source and Multi-powered Smart grid applications offers a promising approach to enhance security measures in complex energy distribution networks. By leveraging GRU's capacity to capture temporal dependencies and patterns in sequential data, the proposed system demonstrates potential in effectively detecting and mitigating instances of theft or unauthorized usage across diverse power sources and grid configurations. The utilization of GRU enables the system to adaptively learn from multi-source data streams, including energy consumption patterns, grid infrastructure data, and historical theft incidents, thereby enhancing its capability to identify anomalous behaviors indicative of security breaches. This advancement underscores the significance of leveraging sophisticated deep learning techniques within Smart grid environments to safeguard critical infrastructure and ensure the reliability and integrity of energy distribution systems.

**REFERNCES**

Zhang, Y., Li, Y., Wang, Y., & Cao, Y. (2018). An Intelligent Theft Identification System for Smart Grids Based on Machine Learning Techniques. IEEE Transactions on Industrial Informatics, 14(7), 2974-2983.

Saha, T., Islam, S. M. R., Mahmud, M. A., Islam, M. S., & Naser, A. (2020). A Machine Learning Approach for Theft Detection in Smart Grid. 2020 IEEE Region 10 Symposium (TENSYMP), Dhaka, Bangladesh.

Rahman, M. M., Islam, M. S., Mahmud, M. A., & Mamun, A. A. (2021). Machine Learning-Based Anomaly Detection in Smart Grid for Theft Identification. 2021 IEEE International Conference on Environment and Electrical Engineering and 2021 IEEE Industrial and Commercial Power Systems Europe (EEEIC / I&CPS Europe), Padua, Italy.

Deo, A., Kumar, A., & Jha, R. K. (2020). Smart Grid Intrusion Detection Using Machine Learning Techniques. 2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT), Kharagpur, India.

Malik, S., Afzal, M. K., & Shaikh, S. M. (2019). A Novel Approach for Intrusion Detection in Smart Grid Using Machine Learning Techniques. 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Zhangjiajie, China.