**SYNOPSIS**

The Internet of Things (IoT) envisions a vast network of connected applications, spanning diverse domains and facilitating communication among heterogeneous objects to offer a spectrum of services. This ubiquity of connected devices sets the stage for a transformative paradigm within the IoT landscape known as the Social Internet of Things (SIoT). The integration of social and human activities into the IoT platform introduces novel dimensions, presenting SIoT as a burgeoning field.

In the SIoT framework, the working layer is structured into three distinct layers, each fulfilling unique functions and employing diverse technologies, data processing methodologies, and communication protocols. Artificial Intelligence (AI) plays a pivotal role in this landscape, enhancing the capabilities of SIoT systems to process and analyse data intelligently, leading to more informed decision-making.

The rapid expansion of SIoT is evident in real-time IoT applications, where AI is strategically implemented. Instances include user-intervened scenarios with IP-based phones, cameras, health equipment, televisions, and automobiles. In this project, implementation of SIoT and AI propels the evolution of smart and adaptive systems, fostering a dynamic and interconnected environment.

**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 : Google colab

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**

In the existing systems for trust prediction in Social Internet of Things (SIoT) environments, machine learning (ML) techniques are often employed, but they may not fully leverage the centrality of devices within the network. Typically, ML models utilize features extracted from device interactions, communication patterns, and historical trust ratings to predict trust levels. These models may include traditional classifiers such as logistic regression, decision trees, or ensemble methods like random forests or gradient boosting machines. While these approaches can achieve reasonable performance, they may overlook crucial network structure information and temporal dynamics that influence trust relationships. Furthermore, existing ML-based systems may lack the ability to capture long-term dependencies effectively, leading to suboptimal performance in dynamic SIoT environments. Despite their widespread use, the limitations of conventional ML techniques highlight the need for more advanced approaches that can better exploit the inherent characteristics of SIoT networks, such as centrality-based trust prediction systems using recurrent neural network architectures like GRU and LSTM.

**Disadvantages:**

Limited Temporal Modelling:

Traditional machine learning (ML) techniques used in existing systems often struggle to capture temporal dynamics and long-term dependencies in Social Internet of Things (SIoT) environments. They may not effectively model how trust evolves over time, leading to suboptimal predictions, especially in dynamic scenarios.

Ignoring Network Structure:

Many ML models overlook the network structure and centrality of devices within the SIoT network. This can result in the neglect of influential nodes or connections, impacting the accuracy of trust predictions. Ignoring network structure may lead to overlooking critical information necessary for understanding trust dynamics.

Scalability Issues:

Some ML algorithms used in existing systems may face scalability issues when dealing with large-scale SIoT networks. Training and inference times can become prohibitive as the size of the network grows, limiting the practical applicability of the system in real-world scenarios.

Limited Adaptability:

ML models may lack adaptability to changing SIoT environments and evolving trust relationships. They often rely on fixed feature sets and may struggle to incorporate new data or adapt to shifting network dynamics, resulting in decreased performance over time.

**Advantages:**

Interpretability:

Many traditional ML techniques offer interpretability, allowing users to understand the factors influencing trust predictions. This can be valuable for stakeholders in making informed decisions about trust management strategies within SIoT networks.

Wide Availability and Accessibility:

ML algorithms and tools are widely available and accessible, making them easy to implement and deploy in various SIoT applications. This accessibility facilitates the adoption of trust prediction systems in diverse contexts.

Efficiency in Training:

Compared to more complex deep learning approaches, traditional ML techniques often require less computational resources and training time. This efficiency can be advantageous, especially in resource-constrained environments or for applications with limited processing capabilities.

Robust Performance in Certain Scenarios:

In specific SIoT environments with well-defined features and relatively stable trust dynamics, traditional ML models can achieve satisfactory performance. They may provide reliable predictions in scenarios where temporal dynamics and network structure are less critical.

while existing ML-based trust prediction systems offer advantages such as interpretability and accessibility, they suffer from limitations in temporal modelling, scalability, and adaptability. These systems may overlook crucial network structure information and struggle to capture the evolving dynamics of trust relationships in SIoT environments. As a result, there is a growing need for more advanced approaches, such as centrality-based trust prediction systems using recurrent neural networks, to address these shortcomings and enhance trust management in SIoT networks.

**PROPOSED SYSTEM**

In the proposed system, we aim to develop centrality-based trust prediction systems for Social Internet of Things (SIoT) environments utilizing Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) architectures. The objective is to leverage the strengths of these recurrent neural network variants to predict trust levels among interconnected IoT devices in SIoT networks more accurately and efficiently.

**Model Architecture:**

Design separate GRU and LSTM architectures for trust prediction, with each variant processing the extracted features independently.

Input sequences of device interactions and centrality metrics into the GRU and LSTM networks, allowing them to learn temporal patterns and dependencies in the data. Incorporate attention mechanisms within the networks to focus on relevant features and enhance predictive performance.

Training and Validation:

Split the dataset into training, validation, and testing sets to evaluate the performance of the proposed models.

Train the GRU and LSTM networks using backpropagation through time (BPTT) or other suitable optimization techniques, optimizing for trust prediction accuracy.

Validate the trained models using the validation set and fine-tune hyperparameters to optimize performance further.

Evaluation Metrics:

Evaluate the performance of the proposed centrality-based trust prediction systems using metrics such as accuracy, precision, recall, F1-score, and area under the receiver operating characteristic (ROC) curve.

Compare the performance of the GRU and LSTM models with baseline methods and traditional machine learning approaches to assess their effectiveness.

Advantages of the Proposed System:

Temporal Modelling: Both GRU and LSTM architectures are well-suited for capturing temporal dependencies in sequential data, allowing the proposed system to effectively model trust dynamics over time.

Centrality Integration: By incorporating centrality metrics, the proposed system can leverage network structure and device importance to enhance trust prediction accuracy and robustness.

Flexibility and Scalability: GRU and LSTM networks offer flexibility and scalability, allowing the system to accommodate varying SIoT network sizes and complexities.

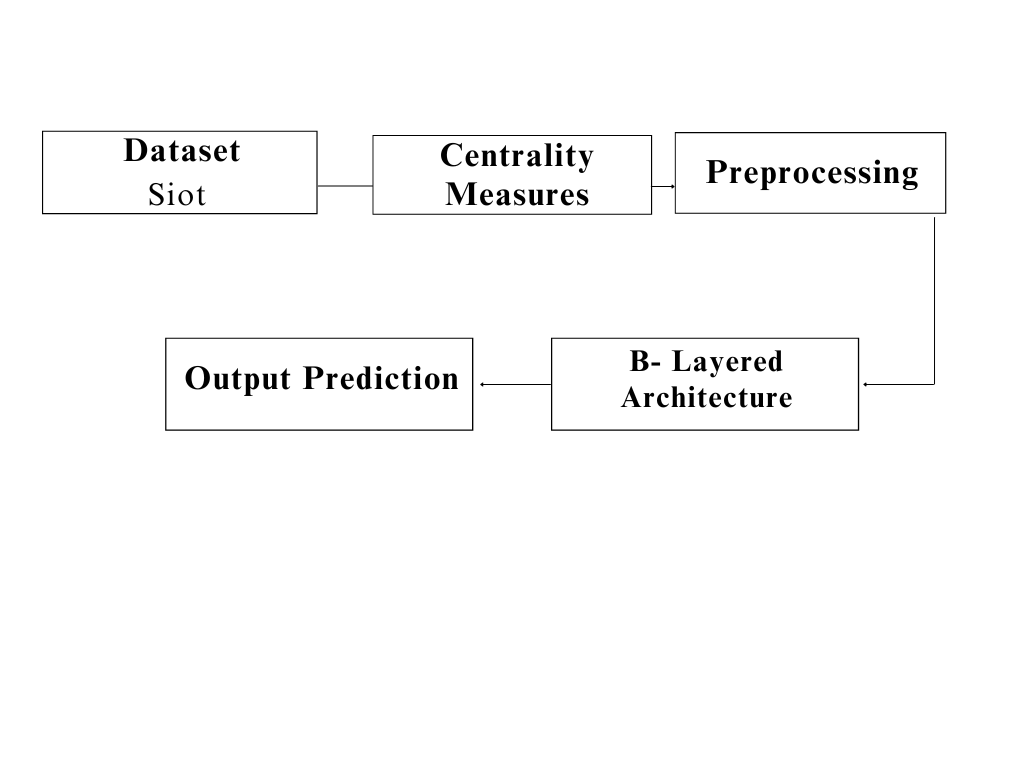
Interpretability: The attention mechanisms incorporated within the networks provide interpretability by highlighting influential features and factors contributing to trust predictions.

Performance: The combination of centrality-based features with recurrent neural network architectures enables the proposed system to achieve superior performance compared to traditional methods, especially in dynamic and evolving SIoT environments.

In summary, the proposed centrality-based SIoT trust prediction systems utilizing GRU and LSTM architectures offer a comprehensive approach to modelling trust dynamics in complex IoT networks. By leveraging temporal information, network structure, and device centrality, the system aims to enhance trust prediction accuracy and reliability, thereby contributing to the security and integrity of SIoT applications.

**SYSTEM DESIGN**

Centrality Based SIoT Trust prediction systems using b-Layered Architectures



**Dataset Description:**

In this dataset, comprised of 32,428 nodes, each node is characterized by several centrality measures including degree centrality, closeness centrality, betweenness centrality, PageRank, and an alternative calculation of betweenness centrality. Degree centrality reflects the number of connections a node has, while closeness centrality indicates how quickly a node can interact with others in the network. Betweenness centrality measures the extent to which a node lies on the shortest paths between other nodes, while PageRank assesses a node's importance based on both its connections and the connections of its neighbours. Additionally, an alternative calculation of betweenness centrality is provided. The dataset also includes a label indicating the classification or category of each node. These centrality measures provide insights into the structural and functional importance of nodes within the network, facilitating analyses such as identifying influential nodes, detecting communities, or predicting node behaviour based on their centrality characteristics.

**Centrality Measures:**

Extract relevant features from the data, including device identities, communication patterns, temporal information, and centrality metrics such as degree centrality, betweenness centrality, and in, out centrality.

Compute centrality metrics to quantify the importance of each device within the network based on its connectivity and influence.

**Pre-Processing:**

Gather data from various sources within the SIoT network, including communication logs, device interactions, and historical trust ratings.

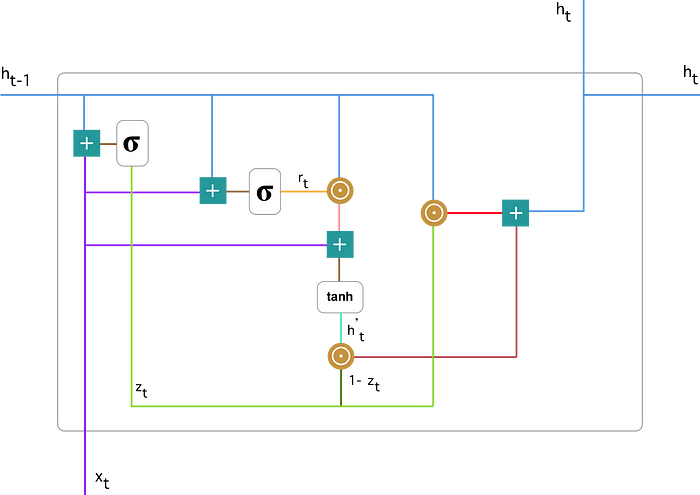
Preprocess the data by cleaning, filtering, and normalizing it to remove noise and ensure consistency.

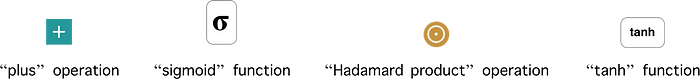
**Deep Learning Algorithm:**

**Gated Recurrent Unit (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.

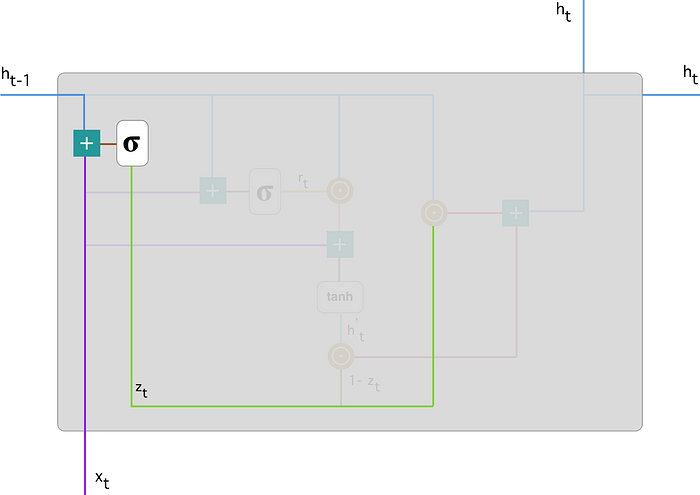
The mathematically notation:



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:



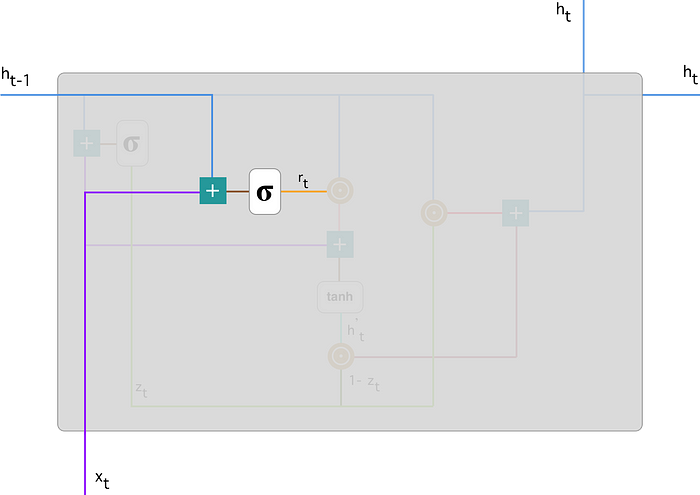
**Reset Gate:**

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:

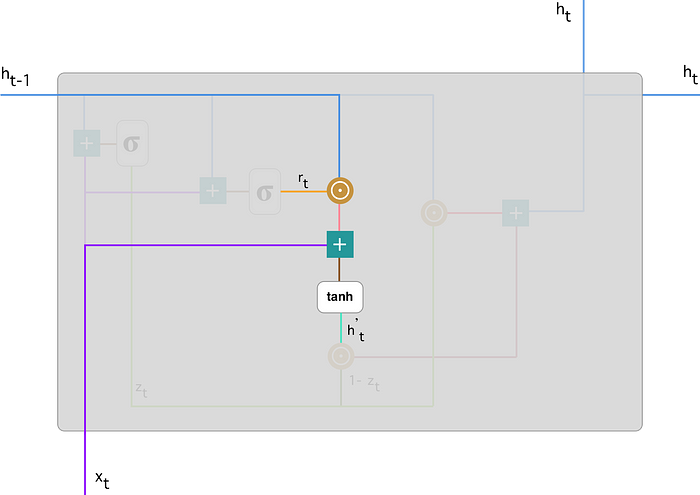




Candidate State Calculation: The candidate state is calculated based on the input and the previous hidden state, similar to traditional RNNs. However, in GRU, this calculation is modulated by the reset gate, which controls how much of the previous hidden state should be considered.



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 and Applications 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.

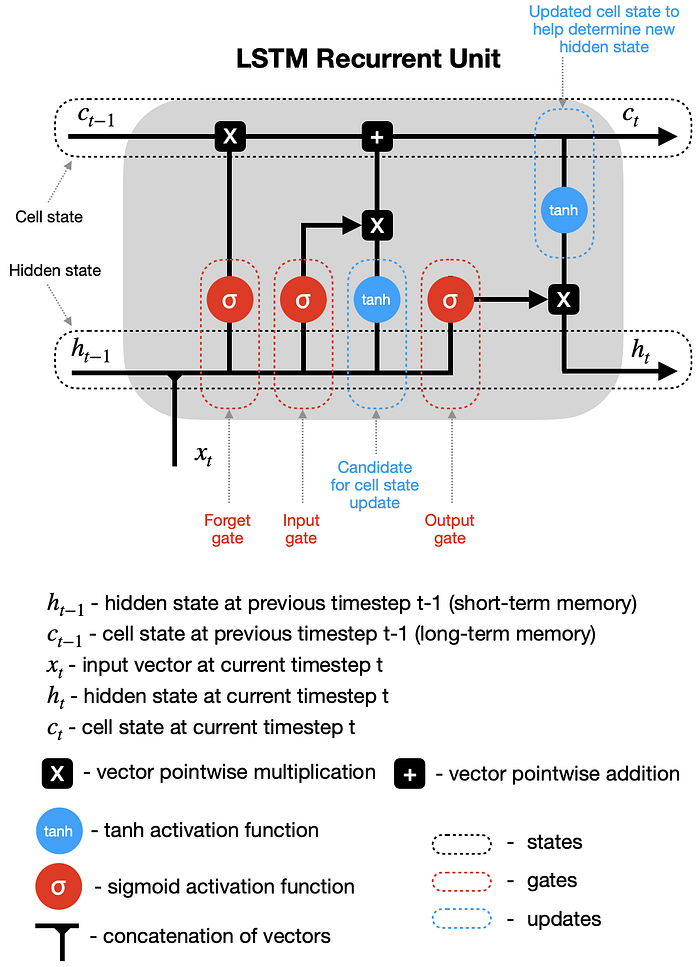
Parallelization: GRU operations can be parallelized more effectively than LSTM operations, leading to faster training and inference times, especially on hardware with parallel processing capabilities like GPUs.

**Applications:**

GRU has been successfully applied in various sequence modelling tasks including natural language processing tasks such as machine translation, sentiment analysis, and text generation. It has also been used in time series prediction tasks such as stock market forecasting, weather prediction, and speech recognition.

**Long Short-Term Memory (LSTM)**

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture that is specifically designed to address the vanishing gradient problem, which is a common issue in traditional RNNs. LSTM was introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997. It has become a widely used and powerful tool in various fields, especially in natural language processing, time series prediction, speech recognition, and many other sequence modelling tasks.



**Architecture of LSTM:**

The LSTM architecture consists of memory cells that are interconnected through a series of gates. These gates regulate the flow of information within the network, allowing LSTMs to learn and remember long-range dependencies in sequential data.

**Memory Cell:**

The memory cell is the core component of the LSTM architecture. It maintains a hidden state vector that captures information from previous time steps and updates it based on the current input. Unlike traditional RNNs, the memory cell can selectively retain or forget information over long periods of time, making it well-suited for capturing long-term dependencies.

**Forget Gate:**

The forget gate determines which information from the previous time step should be discarded and which information should be retained in the memory cell. It takes the input and the previous hidden state as input and outputs a value between 0 and 1 for each element of the memory cell. A value close to 0 means that the corresponding element of the memory cell will be forgotten, while a value close to 1 means that it will be retained.

**Input Gate:**

The input gate decides which new information should be added to the memory cell. It takes the input and the previous hidden state as input and outputs a value between 0 and 1 for each element of the memory cell. A value close to 0 means that the corresponding element of the memory cell will not be updated, while a value close to 1 means that it will be updated fully.

**Output Gate:**

The output gate determines which information from the memory cell should be passed to the next time step. It takes the input and the previous hidden state as input and outputs a value between 0 and 1 for each element of the memory cell. A value close to 0 means that the corresponding element of the memory cell will not be included in the output, while a value close to 1 means that it will be included fully.

**Advantages and Applications of LSTM:**

Long-Term Dependencies: LSTM is specifically designed to capture long-term dependencies in sequential data. Its ability to selectively retain and forget information over long periods of time enables it to remember important information from distant time steps, making it effective in tasks that require modelling complex temporal patterns.

Addressing Vanishing Gradient Problem: LSTM addresses the vanishing gradient problem better than traditional RNNs by incorporating mechanisms such as forget gates and input gates. These gates allow LSTMs to control the flow of gradients during training, enabling them to learn long-range dependencies more effectively.

Versatility: LSTM has been successfully applied in various sequence modelling tasks including natural language processing tasks such as language translation, sentiment analysis, and text generation. It has also been used in time series prediction tasks such as stock market forecasting, weather prediction, and speech recognition.

Robustness to Noise: LSTM architectures are robust to noisy input data and can effectively filter out irrelevant information, making them suitable for tasks involving noisy or incomplete data.

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.

The 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.

**CODING**

import pandas as pd

import numpy as np

from keras.models import Sequential

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

import seaborn as sns

import matplotlib.pyplot as plt

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

data = pd.read\_csv("/content/drive/MyDrive/siot/centrality\_measures\_data.csv")

data

data.info()

data['Label'] = (data['Closeness Centrality'] > 0.3).astype(int)

# to an existing Excel file

data.to\_csv("/content/drive/MyDrive/siot/siot\_file.csv", index=False, mode='a', header=False)

data.info()

data.Label.value\_counts()

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

y = data['Label']

import numpy as np

import pandas as pd

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

from sklearn.preprocessing import StandardScaler

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

from keras.models import Sequential

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

from keras.utils import to\_categorical

# 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)

# Reshape the input data for GRU

X\_train\_reshaped = X\_train.values.reshape((X\_train.shape[0], 1, X\_train.shape[1]))

X\_test\_reshaped = X\_test.values.reshape((X\_test.shape[0], 1, X\_test.shape[1]))

# Standardize features

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train\_reshaped.reshape(-1, X\_train\_reshaped.shape[-1])).reshape(X\_train\_reshaped.shape)

X\_test\_scaled = scaler.transform(X\_test\_reshaped.reshape(-1, X\_test\_reshaped.shape[-1])).reshape(X\_test\_reshaped.shape)

# Build a model with both LSTM and GRU layers

model = Sequential()

model.add(LSTM(units=128,input\_shape=(X\_train\_scaled.shape[1], X\_train\_scaled.shape[2]), return\_sequences=True))

model.add(GRU(units=128,input\_shape=(X\_train\_scaled.shape[1], X\_train\_scaled.shape[2])))

model.add(Dropout(0.2))

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

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

# Train the model

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

# Evaluate the model on the test set

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

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

print("Accuracy Score:" )

print(accu)

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()

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()

# 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()

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\_com.history['accuracy'])

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

plt.plot(history\_com.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()

FRAMEWORK CODING:

import tensorflow as tf

import tkinter as tk

from tkinter import Button, Label, Frame

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report, roc\_auc\_score, roc\_curve

import matplotlib.pyplot as plt

from keras.models import Sequential

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

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

from sklearn.preprocessing import StandardScaler

import seaborn as sns

import pandas as pd

import numpy as np

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

from PIL import Image, ImageTk

# Load your dataset here

data = pd.read\_csv("siot\_file.csv")

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

y = data['Label']

# 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)

# Reshape the input data for GRU

X\_train\_reshaped = X\_train.values.reshape((X\_train.shape[0], 1, X\_train.shape[1]))

X\_test\_reshaped = X\_test.values.reshape((X\_test.shape[0], 1, X\_test.shape[1]))

# Standardize features

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train\_reshaped.reshape(-1, X\_train\_reshaped.shape[-1])).reshape(X\_train\_reshaped.shape)

X\_test\_scaled = scaler.transform(X\_test\_reshaped.reshape(-1, X\_test\_reshaped.shape[-1])).reshape(X\_test\_reshaped.shape)

# Build a model with both LSTM and GRU layers

model = Sequential()

model.add(LSTM(units=128, input\_shape=(X\_train\_scaled.shape[1], X\_train\_scaled.shape[2]), return\_sequences=True))

model.add(GRU(units=128, input\_shape=(X\_train\_scaled.shape[1], X\_train\_scaled.shape[2])))

model.add(Dropout(0.2))

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="Centrality Based SIoT Trust prediction systems using b-Layered Architectures", 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:

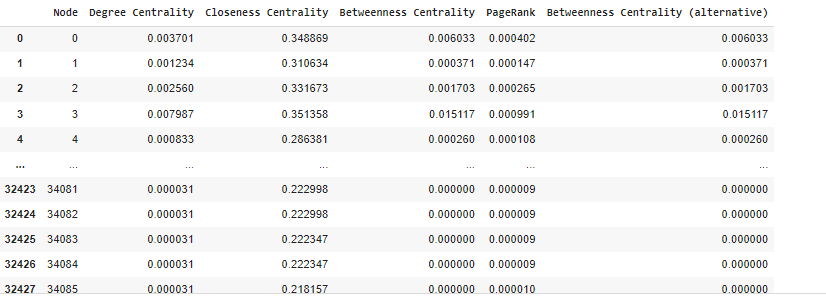


Figure 1: Dataset

Results:

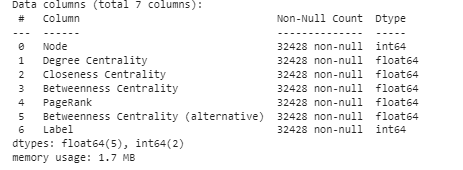


Figure 2: dataset information

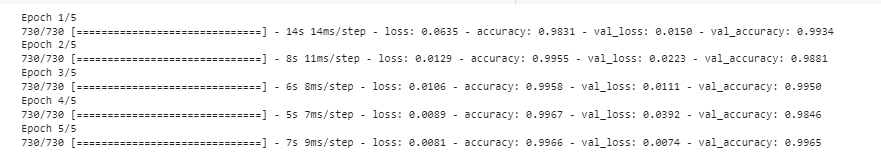


Figure 3: dataset is trained with five epochs with the validation split



Figure 4: predict value is calculated



Figure 5: Accuracy has been calculated for the model

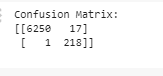


Figure 6: Confusion matrix has been calculated

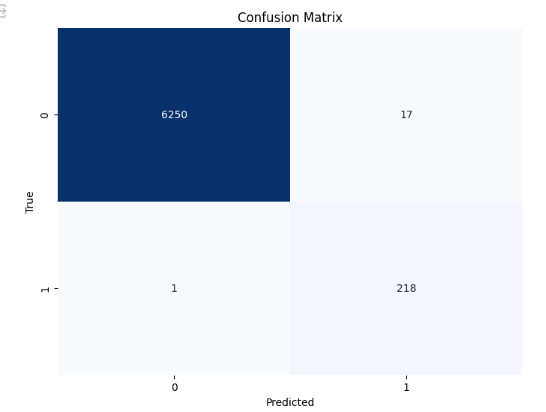


Figure 7: Confusion matrix has been displayed

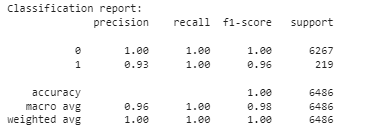


Figure 8: Classification report has been calculated

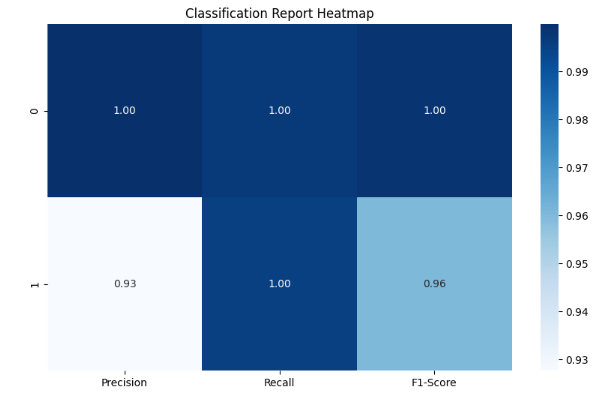


Figure 9: Classification report has been displayed

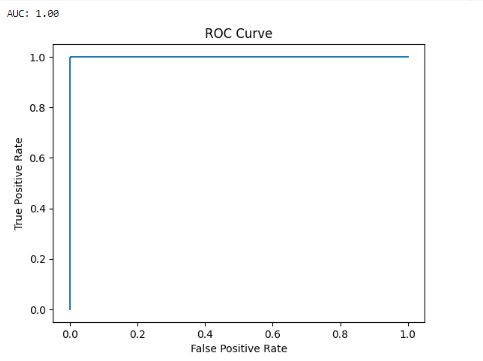


Figure 10: ROC AUC Curve

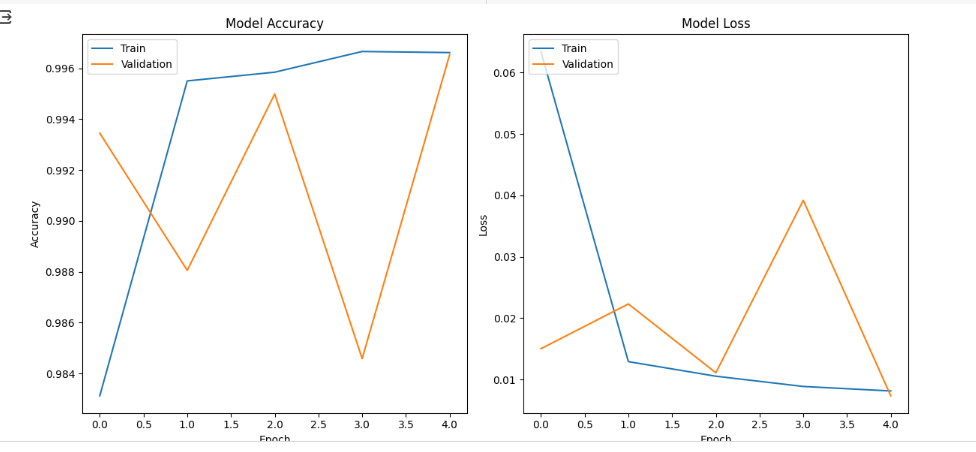


Figure 11: Both model Accuracy and loss

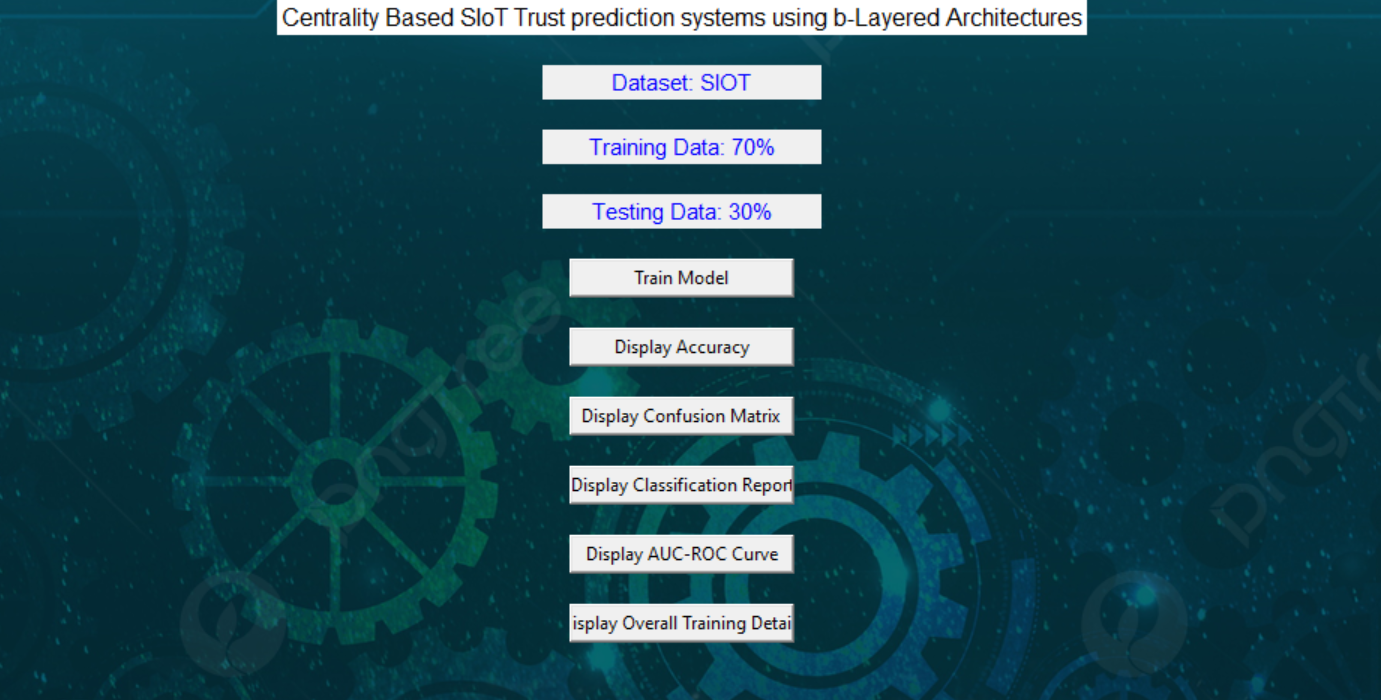


Figure 12: Framework design

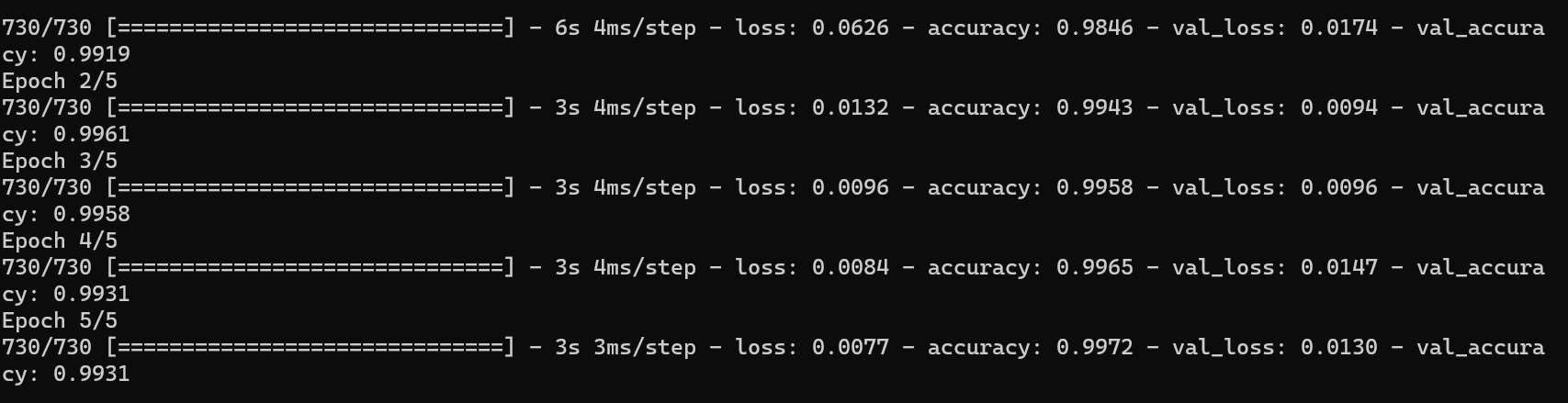
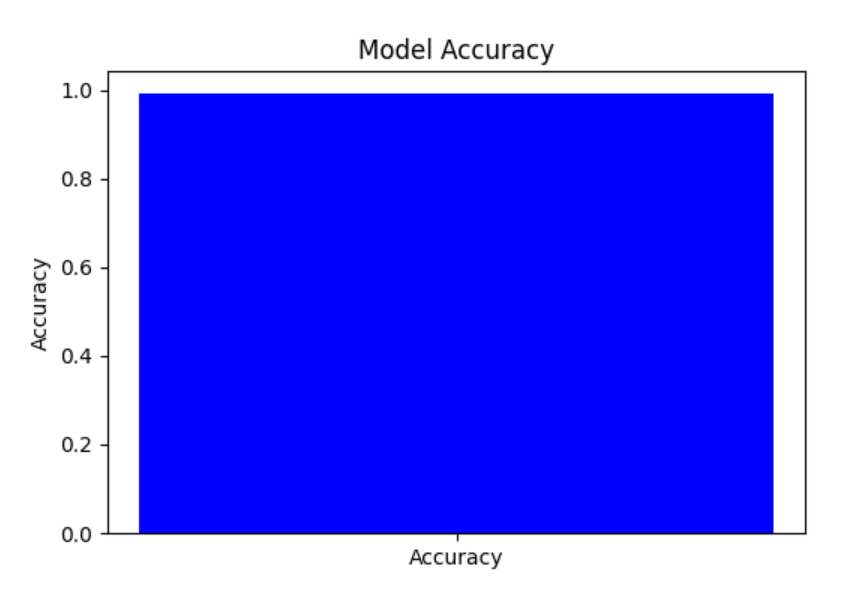


Figure 13: Epoch has been used to train the dataset with algorithm

 Figure 14: Model accuracy has been displayed through graph

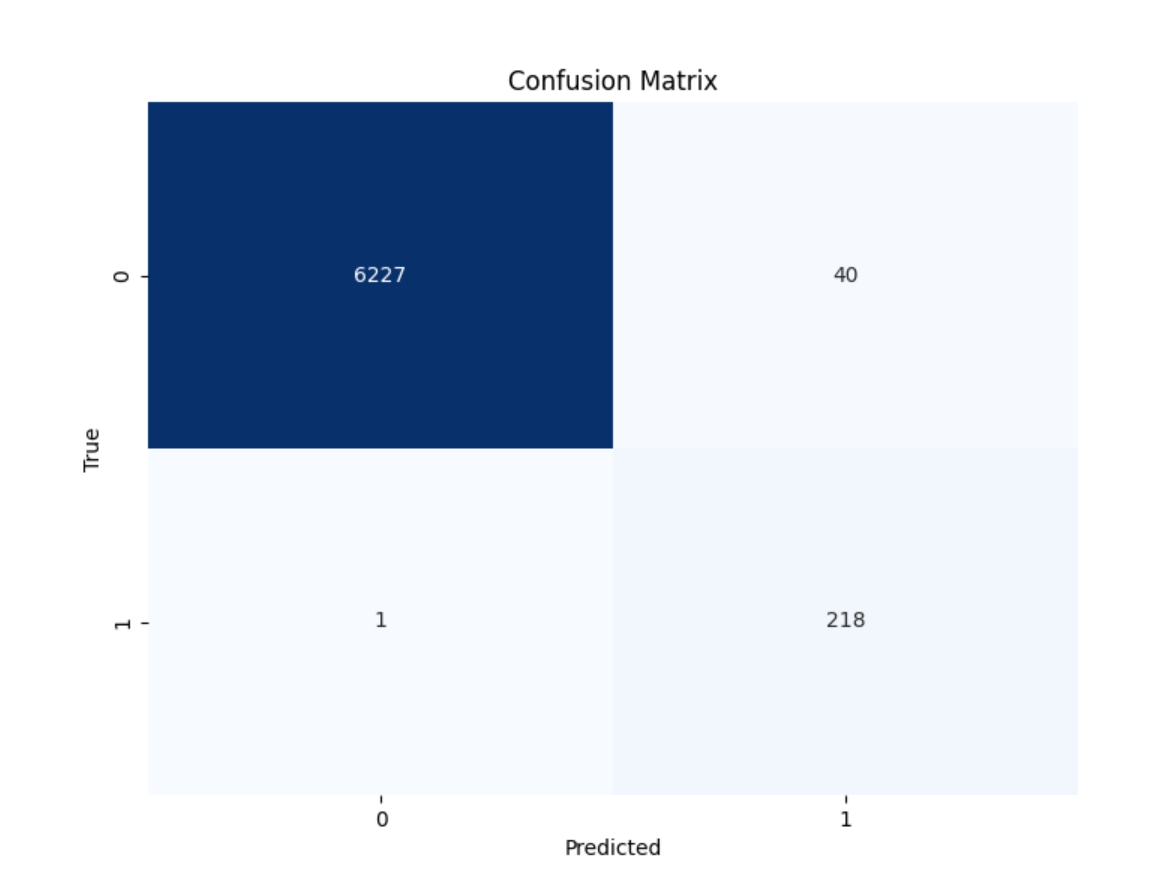


Figure 15: Confusion matrix has been displayed through graph

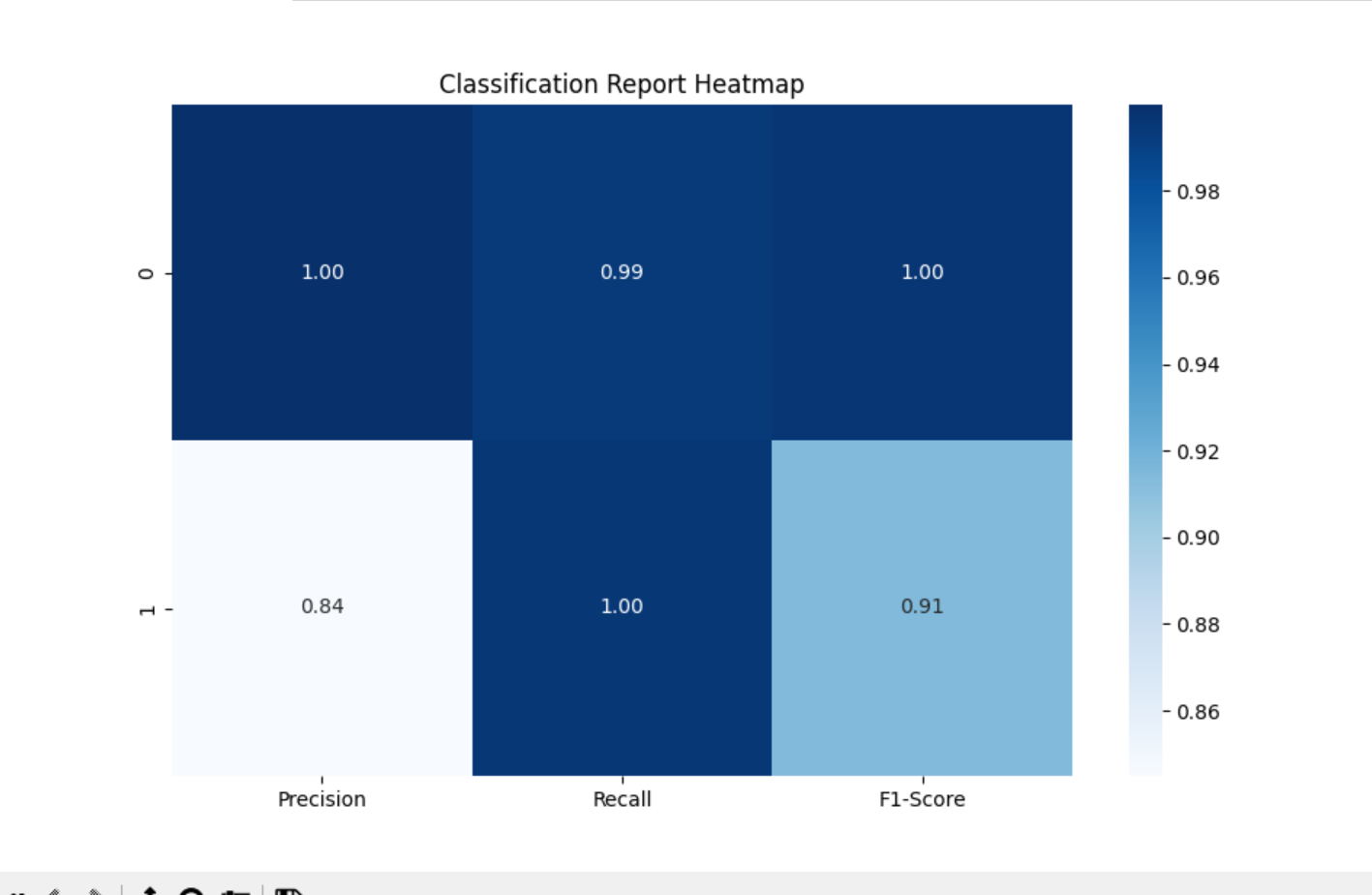


Figure 16: Classification report has been displayed through graph

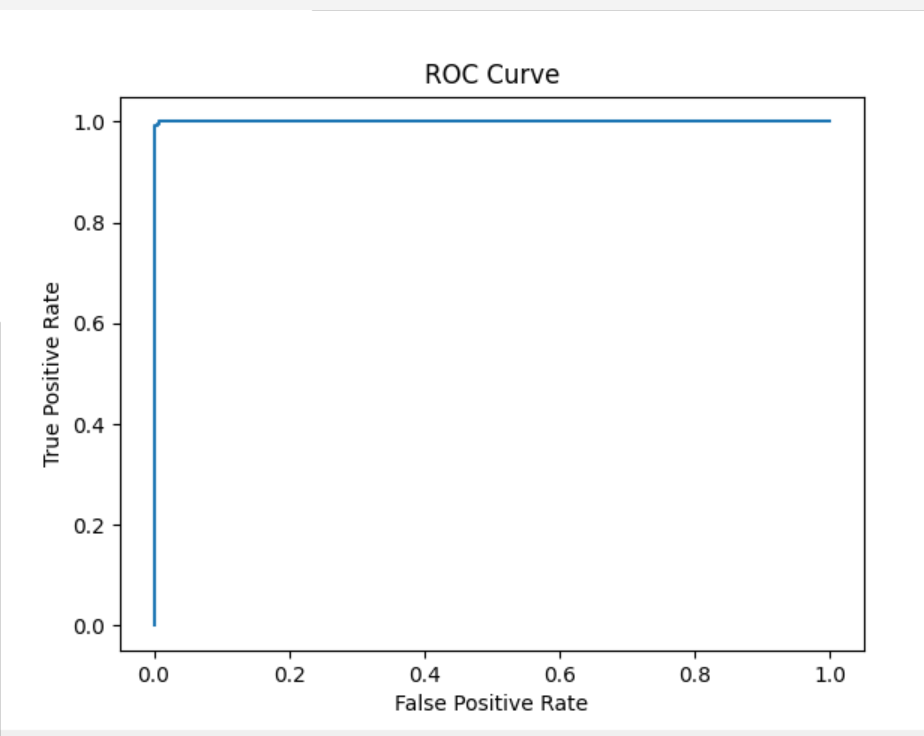


Figure 17: ROC AUC has been displayed

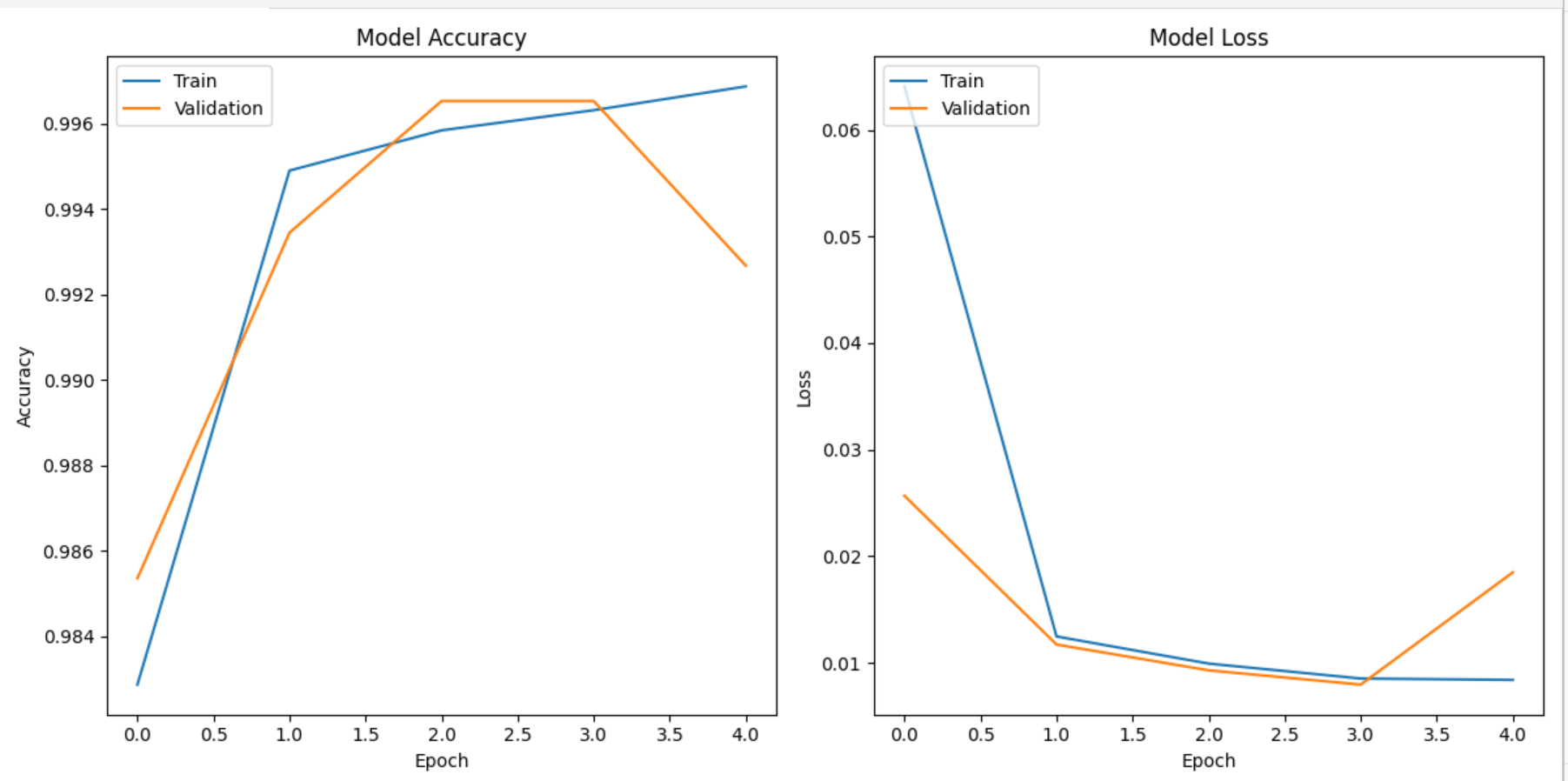


Figure 18: Both model Accuracy and loss had been displayed with graph

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precision | Recall | F1 – score | Support |
| 0 | 1.00 | 1.00 | 1.00 | 6267 |
| 1 | 0.93 | 1.00 | 0.96 | 219 |
| accuracy |  |  | 1.00 | 6486 |
| Macro avg | 0.96 | 0.98 | 0.98 | 6486 |
| Weighted avg | 1.00 | 1.00 | 1.00 | 6486 |

Table 1: classification report of B- Layered

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 |
| LSTM + GRU | 99 |

Table 2: Accuracy of B- Layered

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithms | Accuracy | Training Time | Epochs |
| LSTM + GRU | 99% | 10 mins | 5 |

Table 3: Algorithms and its details

|  |  |  |
| --- | --- | --- |
| Dataset Count | Training Value | Testing Value |
| 204492 | 70 | 30 |

Table 4: 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 proposed centrality-based SIoT trust prediction systems employing GRU and LSTM architectures present a robust framework for modelling and predicting trust dynamics within interconnected IoT networks. By integrating temporal information, device centrality metrics, and recurrent neural network variants, the system aims to accurately capture long-term dependencies and network structure, thereby enhancing trust prediction accuracy and reliability. The flexibility, scalability, and interpretability offered by GRU and LSTM architectures, along with the incorporation of attention mechanisms, contribute to the system's effectiveness in adapting to diverse SIoT environments and highlighting influential factors in trust predictions. Overall, the proposed approach holds promise in bolstering the security and integrity of SIoT applications by providing a comprehensive and data-driven solution for trust management.

**REFERNCES**

Zhang, H., Wang, Y., Fang, Y., & Zhu, Y. (2019). Trust evaluation for social internet of things based on multi-source information and eigenvector centrality. IEEE Access, 7, 131960-131970.

Wang, F., & Liu, J. (2020). Trust Evaluation of Social Internet of Things Based on Node Centrality and Reputation Mechanism. IEEE Access, 8, 102429-102437.

Chen, Z., Cai, Z., Chen, C., & Yang, F. (2018). An Edge Centrality-Based Trust Model in Social Internet of Things. In 2018 IEEE International Conference on Smart Internet of Things (SmartIoT) (pp. 195-200). IEEE.

Yang, F., Zhang, L., Wu, Y., & Jin, H. (2018). Trust evaluation for social Internet of Things systems using multi-dimensional trust feature clustering and self-organizing map. IEEE Access, 6, 54609-54619.

Zhang, L., Yang, F., Wang, H., & Jin, H. (2019). Trust Evaluation for Social Internet of Things Systems Based on Multi-Scale Context Analysis. IEEE Access, 7, 39018-39028.