A Major Project Report On

ADVANCED SYSTEM FOR MALICIOUS SOCIAL BOT DETECTION USING MACHINE LEARNING

Submitted to partial fulfillment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

By

Gopularam Karthik [21911A1224]

K.Thanishka Reddy [21911A1230]

Under the Guidance of

K.MALLIKARJUNA RAO

Associate Professor

Department of Information Technology

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

(An Autonomous Institution)

(Approved by AICTE, Accredited by NAAC, NBA & permanently Affiliated to JNTU, Hyderabad

Aziz Nagar Gate, C.B. Post, Hyderabad-500075)

2024-2025

VIDYA JYOTHI
INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION

(Approved by AICTE, Accredited by NAAC, NBA & permanently Affiliated to JNTUH, Hyderabad)
Aziz Nagar Gate, C.B. Post, Hyderabad-500075

DEPARTMENT OF INFORMATION TECHNOLOGY

CERTIFICATE

This is to certify that the project report titled "**Advanced System For Malicious Social Bot Detection**" is being submitted by **G. Karthik (21911A1224) ,K. Thanishka Reddy (21911A1230),** partial fulfillment for the award of the Degree of Bachelor of Technology in Information Technology, is a record of bonafide work carried out by him under my guidance and supervision. These results embodied in this project report have not been submitted to any other University or Institute for the award of any degree of diploma.

Internal Guide	**Head of Department**
K. Mallikarjuna Rao	Dr .A. Obulesu
Associate Professor	Professor

External Examiner

DECLARATION

I, **G. Karthik , K. Thanishka Reddy** hereby declare that the project entitled, "**Advanced System For Malicious Social Bot Detection** " submitted for the degree of Bachelor of Technology in Information Technology is original and has been done by me and this work is not copied and submitted anywhere for the award of any degree.

Date:	**K. Thanishka Reddy**	**(21911A1230)**
Place: Hyderabad	**G . Karthik**	**(21911A1224)**

ACKNOWLEDGEMENT

We extend our heartfelt gratitude to **Mr.K.Mallikarjuna Rao,** Associate Professor, for his invaluable mentorship and guidance throughout the course of this project. His expertise, encouragement, and insightful feedback played a crucial role in shaping our approach and refining our methodology.

We are deeply grateful to **Dr.A.Obulesu**, Head of the Department, and all faculty members of the department for their constant support and guidance throughout our academic journey.

Our sincere thanks to our beloved **Dean of Accreditation and Rankings, Dr. A. Padmaja Madam, and Academic co-originator, Smt. G. Srilatha Madam**, whose unwavering support proved indispensable during the course.

We also express our heartfelt thanks to our **Principal, Dr. A. Srujana Madam**, for her encouragement and continuous support throughout this endeavour.

We would like to sincerely thank our **College Management** for providing the necessary infrastructure and facilities that enabled us to carry out our work effectively.

We are thankful to our friends, well-wishers, and especially B.Srinivasulu Sir for their valuable suggestions and constant willingness to help, which motivated us to give our best throughout this journey.

Finally, we would like to express our deepest gratitude to our **parents**, whose unwavering love, guidance, and support have been the foundation of our success and aspirations.

G.Karthik	[21911A1224]
K.Thanishka Reddy	[21911A1230]

ABSTRACT

ADVANCED SYSTEM FOR MALICIOUS SOCIAL BOT DETECTION USING MACHINE LEARNING

Malicious social bots generate fake tweets and automate their social relationships either by pretending like a follower or by creating multiple fake accounts with malicious activities. Moreover, malicious social bots post shortened malicious URLs in the tweet in order to redirect the requests of online social networking participants to some malicious servers. Hence, distinguishing malicious social bots from legitimate users is one of the most important tasks in the Twitter network.

To detect malicious social bots, extracting URL-based features (such as URL redirection, frequency of shared URLs, and spam content in URL) consumes less amount of time in comparison with social graph-based features (which rely on the social-interactions of users). Furthermore, malicious social bots cannot easily manipulate URL redirection chains. A learning automata-based malicious social bot detection (LA-MSBD) algorithm is proposed by integrating a trust computation model with URL-based features for identifying trustworthy participants (users) in the Twitter network.

The proposed trust computation model contains two parameters, namely, direct trust and indirect trust. Moreover, the direct trust is derived from Bayes' theorem, and the indirect trust is derived from the Dempster-Shafer Theory (DST) to determine the trustworthiness of each participant accurately.

LIST OF FIGURES

1.INTRODUCTION

1.1 Introduction

Malicious social bot is a software program that pretends to be a real user in online social networks (OSNs). Moreover, malicious social bots perform several malicious attacks, such as spread social spam content, generate fake identities, manipulate online ratings, and perform phishing attacks. In Twitter, when a participant (user) wants to share a tweet containing URI(s) with the neighboring participants (i.e., followers or following), the participant adapts URL shortened service (i.e., bit.ly) in order to reduce the length of URL (because a tweet is restricted up to 140 characters). Moreover, a malicious social bot may post shortened phishing URLs in the tweet. When a participant clicks on a shortened phishing URL, the participant's request will be redirected to intermediate URLS associated with malicious servers that, in turn, redirect the user to malicious web pages. Then, the legitimate participant is exposed to an attacker. This leads to Twitter network suffering from several vulnerabilities (such as phishing attack).

The malicious behavior of participants is analyzed by considering features extracted from the posted URLs (in the tweets), such as URL redirection, frequency of shared URLs, and spam content in URL, to distinguish between legitimate and malicious tweets. To protect against the malicious social bot attack. Several approaches have been proposed to detect spam in the Twitter network. These approaches are based on tweet- content features, social relationship features, and user profile features. However, the malicious social bots can manipulate profile features, such as hashtag ratio, follower ratio, URL ratio, and the number of retweets. The malicious social bots can also manipulate tweet-content features, such as sentimental words, emoticons, and most frequent words used in the tweets, by manipulating the content of each tweet. The social relationship-based features are highly robust because the malicious social bots cannot easily manipulate the social interactions of users in the Twitter network. However, extracting social relationship-based features consumes a huge amount of time due to the massive volume of social network graph.

1.2 Background and Context

Social media platforms have revolutionized communication by enabling instantaneous information sharing. Yet, as these networks have grown, so too have the threats associated with them. One significant threat is the deployment of automated agents or “bots” that can manipulate social dynamics. While not all bots are harmful—many serve useful purposes such as content curation or customer service—the subset that is malicious is particularly concerning.

Malicious social bots operate by creating or hijacking accounts, generating a stream of fraudulent activities, and forging online relationships that mimic those of real users. This phenomenon has been documented extensively in academic and industry studies, which reveal that these bots can quickly amplify misinformation, manipulate trending topics, and even influence political discourse. A closer look at their modus operandi shows that they often rely on automated tools to generate content that seems genuine. For instance, by mimicking user behaviors such as retweeting, following, and liking, they can create the illusion of popularity and legitimacy.

One key characteristic of these bots is their heavy reliance on URL-based content. The malicious URLs embedded in their posts are typically shortened to obscure their true destinations. These shortened URLs are not only used to bypass character limits but also to evade detection by traditional security mechanisms. As a result, URL redirection chains become a critical feature for analysis: while the underlying social graph of interactions is complex and time-intensive to decipher, URL-based features such as redirection patterns, frequency of shared links, and the presence of spam content offer a more direct avenue for detection.

1.3 Challenges in Bot Detection

Detecting malicious social bots is inherently challenging due to the sophisticated techniques they employ. Traditional methods for bot detection often rely on extensive analysis of social graph features, such as the patterns of user interactions, network centrality measures, and community structures. However, these social graph-based approaches come with significant computational overhead. Extracting and analyzing social-interaction data for millions of users requires not only powerful hardware but also time-consuming algorithms that may not scale efficiently in real-world scenarios.

In contrast, URL-based features offer a promising alternative. The process of identifying and analyzing URLs embedded in tweets is less computationally expensive compared to mapping out extensive social graphs. Since malicious bots find it difficult to manipulate the intrinsic properties of URL redirection chains—given that the redirection path is often predetermined by the structure of the web—focusing on these features can lead to faster and more reliable detection. This insight has motivated researchers to explore novel algorithms that integrate URL analysis into the bot detection process.

Yet, even with URL-based detection methods, several challenges remain. One significant hurdle is the dynamic nature of online content. As malicious actors continually evolve their techniques to evade detection, any automated system must be adaptive and robust. Furthermore, ensuring that legitimate users are not mistakenly flagged as malicious requires the detection system to balance sensitivity with specificity. In other words, an effective system must minimize both false positives (where a genuine user is wrongly classified as a bot) and false negatives (where a malicious bot goes undetected).

1.4 EXISTING SYSTEM

Existing methods analyzed social botnet attack on Twitter. The authors have presented that social bots use URL shortening services and URL redirection in order to redirect users to malicious web pages. Existing methods presented methods to detect, retrieve, and analyze botnet over thousands of users to observe the social behavior of bots. In, a social bot hunter model has been presented based on the user behavioral features, such as follower ratio, the number of URLs, and reputation score.

1.5 PROPOSED SYSTEM

The proposed method in this project leverages a Learning Automata-based Malicious Social Bot Detection (LA-MSBD) algorithm to identify and classify malicious bots on Twitter. This approach introduces a dual-layer trust computation model that utilizes direct trust and indirect trust metrics to assess user trustworthiness based on URL-sharing behaviors. Direct trust is calculated using Bayes’ theorem, which evaluates individual tweets, while indirect trust is derived from Dempster-Shafer theory, combining multiple interactions to address uncertainties and inconsistencies in tweet patterns. By focusing on URL-based features, the model captures bot-like behaviors effectively, distinguishing them from legitimate user activities. The combination of these trust measures enhances the model's accuracy, precision, recall, and F-measure, making it a robust tool for bot detection. This hybrid trust model, based on evidence aggregation and probabilistic analysis, provides a comprehensive and reliable mechanism to detect social bots in real-time social media environments.

1.6ADVANTAGES OF PROPOSED SYSTEM

- Improved accuracy in bot detection through adaptive learning.
- Adaptability to new bot strategies with evolving detection capabilities.
- Enhanced reliability with trust computation using Bayes' theorem and Dempster-Shafer theorem
- Data-driven decision-making with pattern analysis from large datasets.
- Scalability and automation for efficient, real-time detection.
- Improved user experience by reducing malicious bot influence.

2.LITERATURE SURVEY

P. Shi, Z. Zhang, and K.-K.-R. Choo et al. [1] developed a novel approach for detecting malicious social bots by analyzing clickstream sequences in online social networks. Their model focuses on the transition probabilities and timing of user behavior, making it harder for bots to mimic legitimate user actions. Experimental results showed a 12.8% increase in detection accuracy compared to traditional methods, emphasizing its effectiveness against various malicious social bot types.

G. Lingam, R. R. Rout, and D. V. L. N. Somayajulu et al. [2] present a deep Q-learning model designed to detect social bots and identify influential users in social networks. By analyzing tweet-based, profile-based, and social graph-based attributes, the model uses a Q-network to calculate optimal Q-values for bot detection. Their experiments on Twitter data validate this model's effectiveness in accurately detecting bots and recognizing users influenced by them.

S. Lee and J. Kim [3] explore the security risks posed by URL shortening services (USSes), which are commonly used on platforms like Twitter. They present a botnet model that leverages these services to obscure command and control (C&C) channels. By using "alias flux," botmasters can hide C&C server IP addresses within shortened URLs, making detection through conventional DNS-based methods challenging. This paper highlights the need for enhanced security measures to counter these botnet tactics that exploit USSes for malicious activities.

H. B. Kazemian and S. Ahmed [4] compare machine learning techniques for detecting malicious webpages, highlighting the limitations of blacklist methods due to their inability to adapt to changing content. They evaluate supervised techniques (K-NN, SVM, Naive Bayes) and unsupervised ones (K-Means, Affinity Propagation), achieving up to 98% accuracy and high clustering performance. The study demonstrates the potential of machine learning to offer scalable, real-time webpage classification to enhance web security.

S. Madisetty and M. S. Desarkar [5] propose a neural network-based ensemble approach for spam detection on Twitter. Given the increasing popularity of social networks, spammers continue to target platforms like Twitter by posting spam tweets. Traditional methods often focus on blocking spammers, but these techniques fail as spammers can create new accounts and resume spamming. Therefore, the study emphasizes the need for robust spam detection systems that can identify spam at the tweet level, offering a more sustainable solution to this issue.

3.SYSTEM REQUIREMENTS SPECIFICATIONS

3.1 SYSTEM REQUIREMENTS

3.1.1 SOFTWARE REQUIREMENTS

● OS	:	Windows XP and above (with any web browser)
● Language	:	Python
● Environment	:	Colab NoteBook
● IDE	:	Google Colab
● Dataset:		Twitter dataset

3.1.2 HARDWARE REQUIREMENTS

● RAM	:	4GB or Above
● Processor	:	Intel i3 or above
● Hard Disk	:	100GB or above
● Monitor	:	15VGA

3.2 MODULE DESCRIPTION

The project involved analyzing the design of few applications so as to make the application more users friendly. To do so, it was really important to keep the navigation from one screen to the other well-ordered and at the same time reducing the amount of typing the user needs to do. In order to make the application more accessible. Here we are proposing three modules and those are as following:

Step A: Preparing The Training Set

Step B: Preparing The Test Set

Step B.1: Getting the authentication credentials

Step B.2: Authenticating our Python script

Step B.3: Creating the function to build the Test set

Step C: Pre-processing BOT Tweets in The Data Sets

Step D: Enhanced Learning Automata Classifier

Step D.1: Building the vocabulary

Step D.2: Matching tweets against our vocabulary

Step D.4: Training the classifier

Step E: Testing The Model

3.3 REQUIREMENTS DEFINITION

After the severe continuous analysis of the problems that arose in the existing system, we are now familiar with the requirements required by the current system. The requirements that the system needs are categorized into functional and non-functional requirements. These requirements are listed below:

3.3.1 FUNCTIONAL REQUIREMENTS

Bot Detection with Reinforcement Learning (RL): The system must use Reinforcement Learning to analyze and classify social media interactions, identifying malicious behavior patterns indicative of social bots. RL models should learn from historical data and user interactions, adjusting their predictions based on new, unseen data.

Feature Extraction for RL: The system must extract relevant features (such as tweet frequency, content sentiment, URL patterns, user engagement) from social media data. These features will be fed into the RL algorithm to help distinguish between legitimate users and bots.

Trust Computation: Bayes' Theorem and Dempster-Shafer Theory must be used for computing direct trust and indirect trust, respectively, within the RL framework. The RL model should incorporate trust scores to adjust the detection accuracy over time.

Continuous Learning: The RL model must be capable of continuous learning from ongoing social media data and user behavior. As new user interactions are processed, the model should update its bot detection strategies by adapting to evolving bot tactics.

Real-Time Detection and Alerts: The system should provide real-time monitoring of social media platforms and instant alerts when a bot is detected, with the RL algorithm being used to continuously refine detection thresholds.

Bot Classification with RL Feedback: The RL-based model must classify users as legitimate users or malicious bots based on real-time interactions, refining its classification over time using rewards and penalties.

User Interaction Behavior Modeling: The system should capture user behavior such as tweet frequency, sentiment analysis, and interaction patterns to feed into the RL algorithm for better classification and detection.

Dynamic Thresholding: The system must use dynamic thresholds for bot detection, adjusting the confidence levels (high, medium, low) based on the RL agent's learning experience and rewards.

Retraining the RL Model: The system should support the retraining of the RL model with new datasets to improve bot detection accuracy over time. This includes adapting to new tactics that bots may employ as they evolve.

Simulated Environment for Model Training: The RL model should be trained in a simulated environment where it interacts with synthetic data to learn bot detection patterns. This should be periodically validated with real-world social media data.

3.3.2 NON-FUNCTIONAL REQUIREMENTS

Reliability: The system must ensure accurate bot detection by the RL model under varying conditions, ensuring consistency in performance even as new behaviors emerge in the social media landscape. The RL model should be able to handle edge cases such as noisy or incomplete data, maintaining a reliable detection rate.

Security: The RL model and the data processing pipeline should be protected against adversarial attacks aimed at misleading the system. Security measures should prevent bots from manipulating the data to influence the model's learning process.

Maintainability: The RL model should be modular and easy to update or retrain with new data. The system should include mechanisms for debugging, versioning, and logging changes to the RL model's behavior for transparency.

Performance: The RL-based bot detection system should operate with minimal latency, processing large volumes of data and providing real-time classification without noticeable delays. It should support a high throughput of incoming social media interactions and handle multiple users concurrently.

Portability: The system, including the RL-based model, should be portable across different platforms and environments (e.g., cloud, on-premise servers) for deployment flexibility and future scalability.

Scalability: The RL model should scale to accommodate increasing data from multiple social media platforms, supporting future integrations and the ability to handle increasing numbers of users or accounts. It should be capable of scaling in terms of data volume and computational load as the bot detection system grows.

Flexibility: The system must be adaptive to new data, evolving behaviors of malicious social bots, and dynamic user patterns. The RL model should adjust its learning strategy based on these changes, ensuring its ongoing effectiveness.

Usability: The system’s interface should present clear insights about the RL model’s performance metrics (such as detection accuracy, false positives, and trust scores), with intuitive controls for administrators to fine-tune the model’s behavior.

Ethical Considerations: The RL model should be fair and unbiased, avoiding biased detection of legitimate users based on incomplete data or skewed feature sets. The system should comply with privacy standards (e.g., GDPR) and ensure that user data is processed responsibly.

Data Integrity: The system must ensure the quality and integrity of training data used by the RL model. This includes dealing with noisy or incomplete data and ensuring that the model doesn’t make inaccurate classifications based on unreliable data sources.

4.METHODOLOGY

4.1 Introduction

The increasing prevalence of malicious social bots on Twitter necessitates a robust and efficient detection mechanism. This paper proposes a Learning Automata-Based Malicious Social Bot Detection (LA-MSBD) algorithm, which integrates trust computation with URL-based feature extraction to differentiate between legitimate users and malicious bots. Unlike traditional graph-based approaches that require extensive computational resources, our method leverages URL redirection patterns, frequency analysis, and trust scores to enhance detection accuracy while reducing computational complexity.

This section provides a detailed explanation of the system architecture, feature extraction process, and the implementation of the learning automata-based algorithm. The methodology is structured as follows:

1. System Architecture – Overview of the detection framework
2. Feature Extraction – Extraction of key URL-based features
3. Trust Computation Model – Assigning trust scores to users
4. Learning Automata-Based Detection – Adaptive decision-making process
5. Implementation Details – Algorithmic steps and pseudocode

4.2 System Architecture

The proposed LA-MSBD system consists of multiple components working together to identify malicious social bots based on URL behaviors and trustworthiness. The architecture comprises five main modules, as illustrated below:

4.2.1 Data Collection Module

This module collects real-time Twitter data using the Twitter API.

It extracts tweet content, user metadata, URLs, timestamps, and retweet patterns.

Only tweets containing URLs are considered for analysis.

4.2.2 URL Feature Extraction Modul

Extracts URL redirection patterns, frequency of shared URLs, and spam content detection.

Resolves shortened URLs (e.g., from bit.ly, tinyurl.com) to detect multi-hop redirections

Identifies suspicious URLs based on domain reputation and blacklist databases.

4.2.3 Trust Computation Module

Assigns a trust score to each user based on historical behavior.

Users consistently sharing suspicious links receive low trust scores.

4.2.4 Learning Automata-Based Decision Module

Uses an adaptive learning automata framework to refine bot detection based on evolving patterns.

Updates detection thresholds dynamically to improve accuracy.

4.2.5 Classification Module

Uses a supervised classification model (e.g., SVM, Random Forest) to label users as legitimate or malicious bots.

The final decision is based on URL-based features, trust scores, and learning automata outputs.

The integration of these modules allows for real-time and scalable detection of malicious bots.

4.3 Feature Extraction Process

Feature extraction is a critical step in bot detection. Our approach focuses on URL-based features due to their resistance to bot manipulation.

4.3.1 URL Redirection Features

Malicious bots often use URL shorteners to hide malicious links. We extract:

Number of redirections (multi-hop URLs are more suspicious).

Final landing page domain (compared against known blacklist databases).

Time to resolve URL (suspicious URLs often have delayed resolutions).

4.3.2 URL Frequency Features

Bots frequently share the same links to maximize their reach. We extract:

Repetition frequency of URLs per user.

Number of unique URLs shared by a user (bots tend to post the same few links repeatedly).

URL similarity across multiple accounts (bot clusters exhibit high URL overlap).

4.3.3 Content-Based Features

Extracts textual and metadata features:

Presence of spam words in the tweet.

Ratio of hashtags and mentions (bots tend to overuse hashtags to increase visibility)

Sentiment analysis of tweet text (bots often post emotionally charged content.

These features are then passed to the trust computation mode

4.4 Trust Computation Model

The Trust Computation Model assigns a trust score to each user based on historical behavior. The trust score is computed as follows:

4.4.1 Trust Score Formula.

The trust score $T(u)$ of a user u is calculated as:

T(u) = 1 - \frac{M(u)}{M(u) + L(u)

Where:

$M(u)$ = Number of times user u has shared malicious URLs

$L(u)$ = Number of times user u has shared legitimate URLs

4.4.2 Trust Score Classification

High Trust Users ($T(u) > 0.8$) → Likely human.

Medium Trust Users ($0.5 \leq T(u) \leq 0.8$) → Requires further analysis.

Low Trust Users ($T(u) < 0.5$) → Likely bot.

Users with persistently low trust scores are flagged as malicious bots.

4.5 Learning Automata-Based Detection Algorithm

To dynamically improve detection accuracy, a Learning Automata (LA)-based approach is employed. Learning automata refine decision-making over time by interacting with the environment.

4.5.1 Learning Automata Formulation

Let A be a set of actions (e.g., classify as bot or human).

Let P(A) be the probability distribution over actions.

Let R be a reward function based on classification accuracy.

At each step:

1. The system selects an action A(i) based on probability P(A).

2. The action is evaluated based on ground truth labels.

3. Reward or penalty is assigned:

If correct: Increase P(A(i)).

If incorrect: Decrease P(A(i)).

4. The algorithm adapts dynamically to improve classification accuracy.

4.5.2 Pseudocode for LA-MSBD Algorithm

```
Initialize P(A) = [0.5, 0.5]  # Equal probability for bot/human
For each new tweet:
    Extract URL-based features
    Compute trust score T(u)
    Select action A(i) based on P(A)
    Evaluate classification result
    If correct:
        Increase P(A(i)) by reward factor
    Else:
        Decrease P(A(i)) by penalty factor
    Normalize P(A) to maintain probability sum = 1
Return final classification decision
```

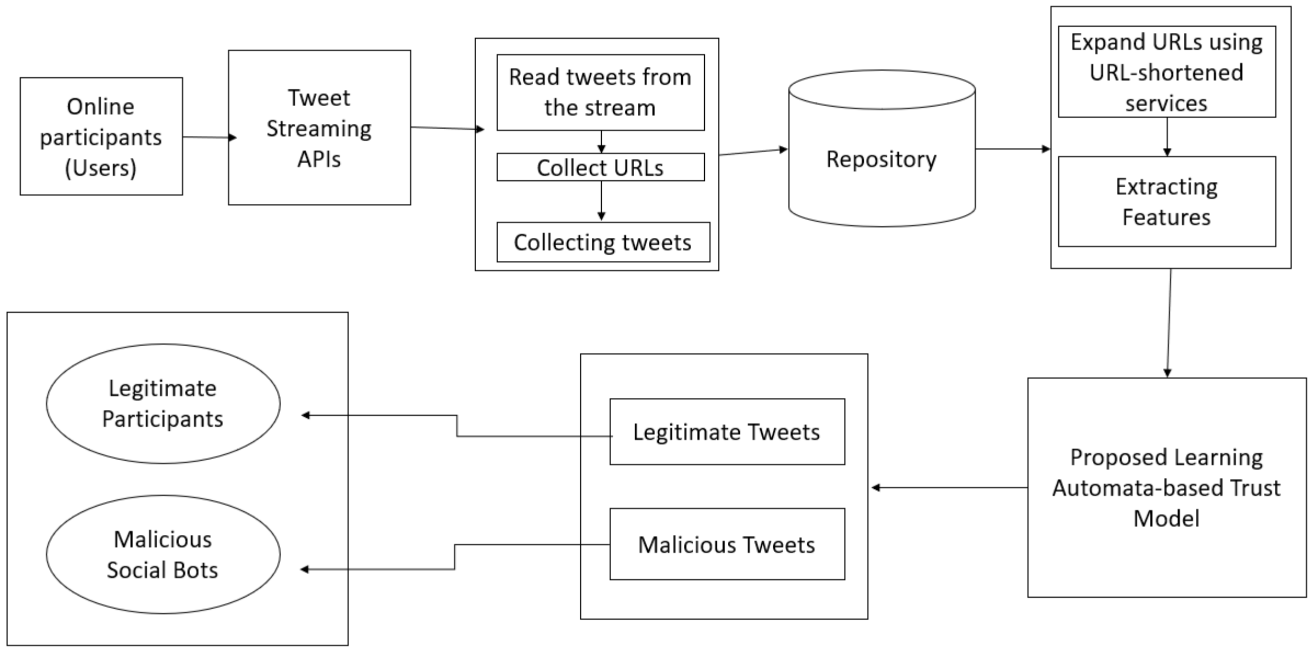
This approach ensures that the system continuously learns and improves detection performance.

5.SYSTEM DESIGN

5.1 ARCHITECTURE

The purpose of the design phase is to arrange an answer of the mater such as by the necessity document. This part is that the opening moves in moving the matter domain to the answer domain. The design phase satisfies the requirements of the system. The design activity is commonly divided into 2 separate phases System Design and Detailed Design. System Design conjointly referred to as top-ranking style aims to spot the modules that ought to be within the system, the specifications of those modules, and the way them move with one another to supply the specified results. Detailed Design, the inner logic of every of the modules laid out in system design is determined. Here first we collect the data sets and process the data and we remove if there are any impurities in the data sets. Next the data is normalized if needed like it can be converted to smaller volume of data. Next the data is converted to supporting format. And then it is stored in the databases. Next the required method is applied. Now we get the final results.

Figure. 5.1: Architecture



5.2UML DIAGRAM

5.2.1 USE CASE DIAGRAM

A use case diagram in Unified Modeling Language (UML) is a behavioral diagram defined by and created from a Use-case analysis. There are 2 actors in our use case diagram those are: User, ML Analyst. Both can load a dataset, in addition a user can collect dataset, input comments and view result whereas an ML analyst can test, train and predict suspicious activities.

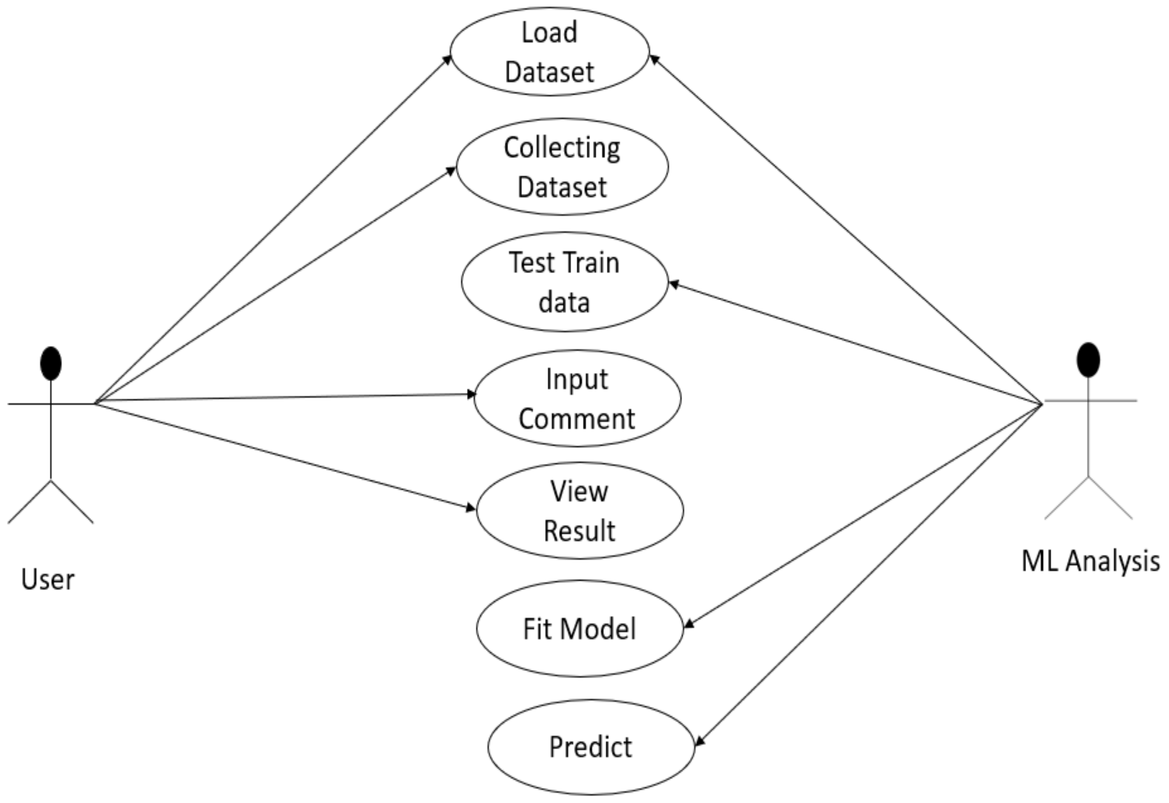


Figure. 5.2.1: Use case diagram

5.2.2ACTIVITY DIAGRAM

Activity diagram is a flowchart to represent the flow from one activity to another activity andcan be described as the operation of the system.

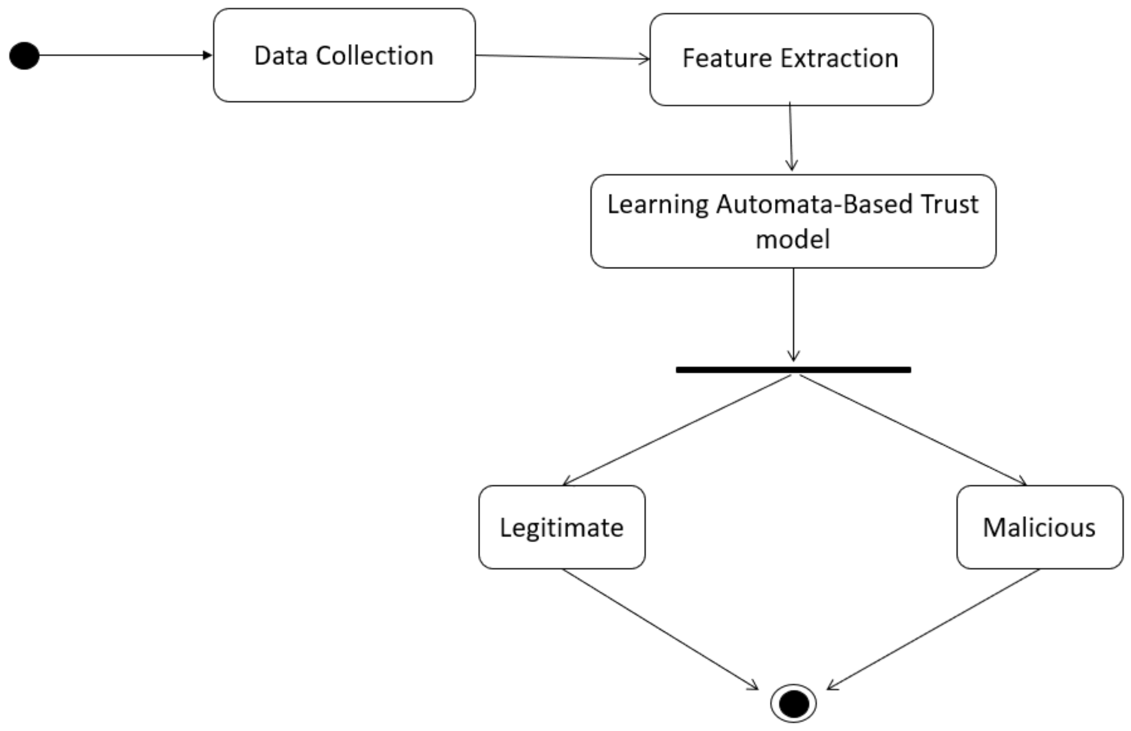


Figure. 5.2.2: Activity Diagram

5.2.3 CLASS DIAGRAM

A class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

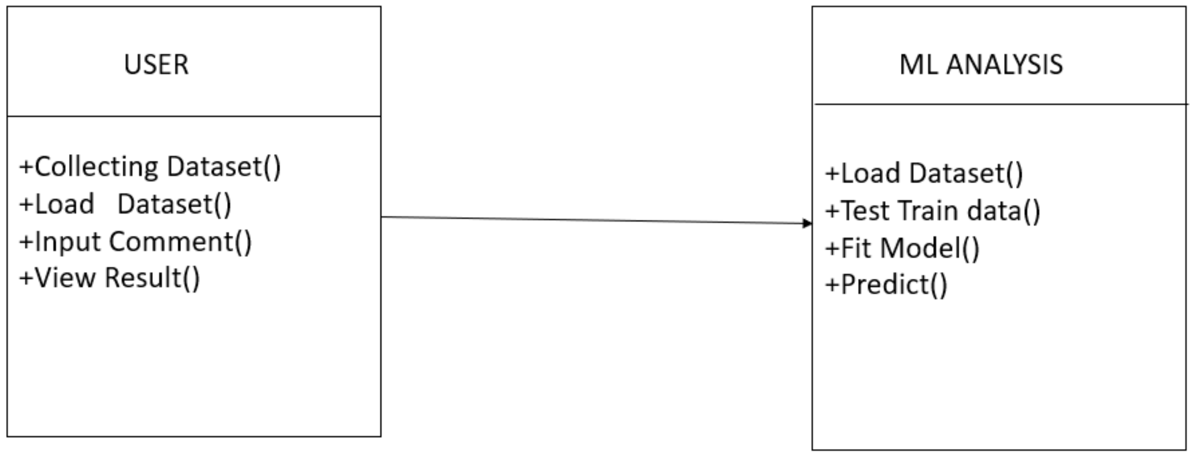


Figure. 5.2.3: Class Diagram for User and ML Analyst

5.2.4 SEQUENCE DIAGRAM

Sequence diagrams are used to communicate the behavior of a system to developers, stakeholders, and other users. They can be used to model simple processes and complex processes.

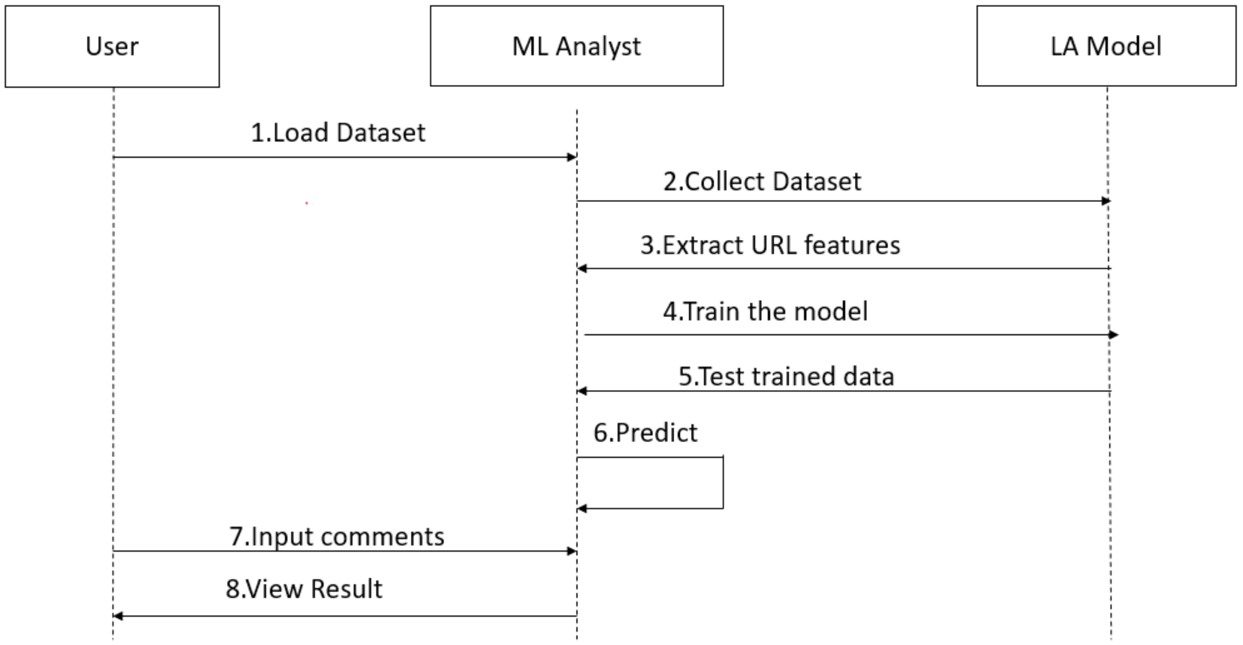


Figure. 5.2.4: Sequence diagram

5.2.5 DATAFLOW DIAGRAM

Data Flow Diagram can also be termed as bubble chart. It is a pictorial or graphical form, which can be applied to represent the input data to a system and multiple functions carried out on the data and the generated output by the system.

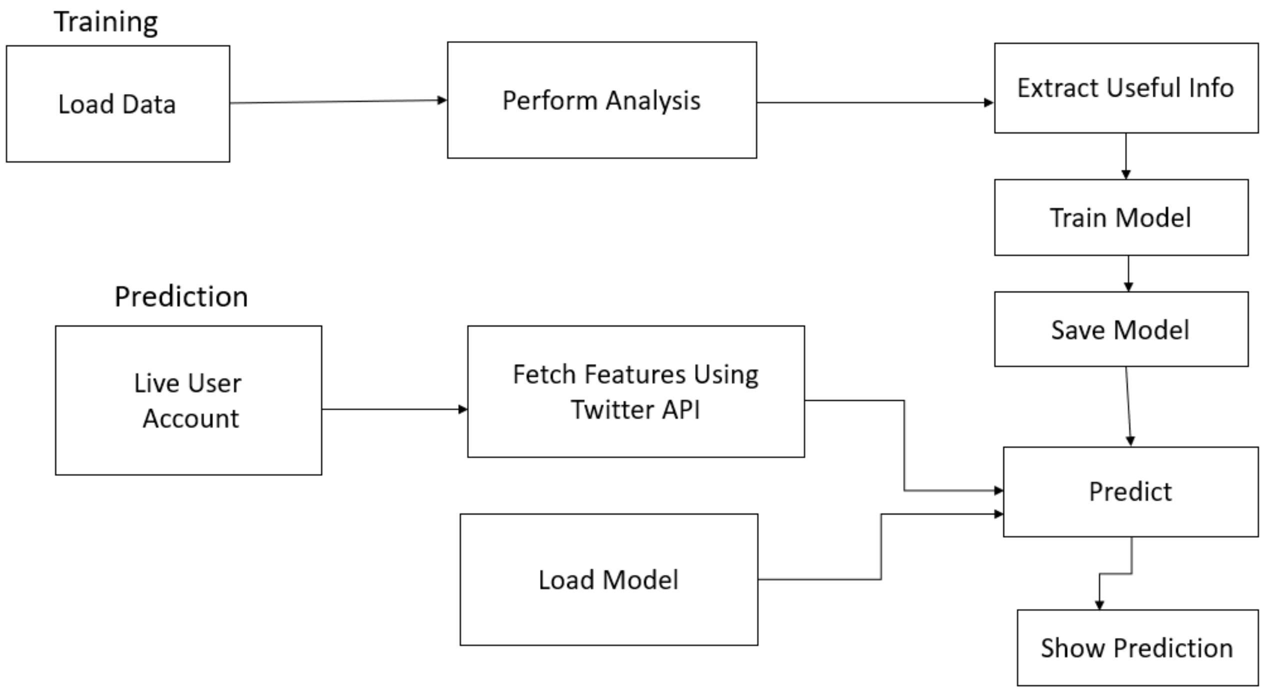


Figure. 5.2.5: Dataflow Diagram

6.SOFTWARE IMPLEMENTATION

6.1 GOOGLE COLAB

Google Collab is a cloud-based Python development environment that enables users to write and execute Python code through the browser, utilizing Google's resources for computation. It's particularly popular for machine learning and data science tasks, providing free access to GPUs and TPUs, which are essential for handling intensive computations.

Each Google Colab notebook consists of:

- **Cells:** These contain either code or text. Code cells execute Python code, while text cells support Markdown and can include formatted text, images, and links.
- **Runtime:** The execution environment of Google Collab, which provides different hardware accelerators like CPUs, GPUs, or TPUs. The environment can be restarted or reset as needed.
- **Libraries:** Collab allows users to import Python libraries like TensorFlow, PyTorch, Keras, and more, directly within the notebook environment.

When you start a Collab session:

- The code runs in an isolated environment with its own virtual machine, enabling you to execute complex programs independently.
- Each session is temporary, so data storage is temporary unless connected to external storage (such as Google Drive) or data is saved periodically.

Collab provides built-in integration with Google Drive, enabling users to save and load files from their Google Drive accounts. It also has tools for collaborative work, allowing multiple users to edit a notebook simultaneously, making it ideal for team projects.

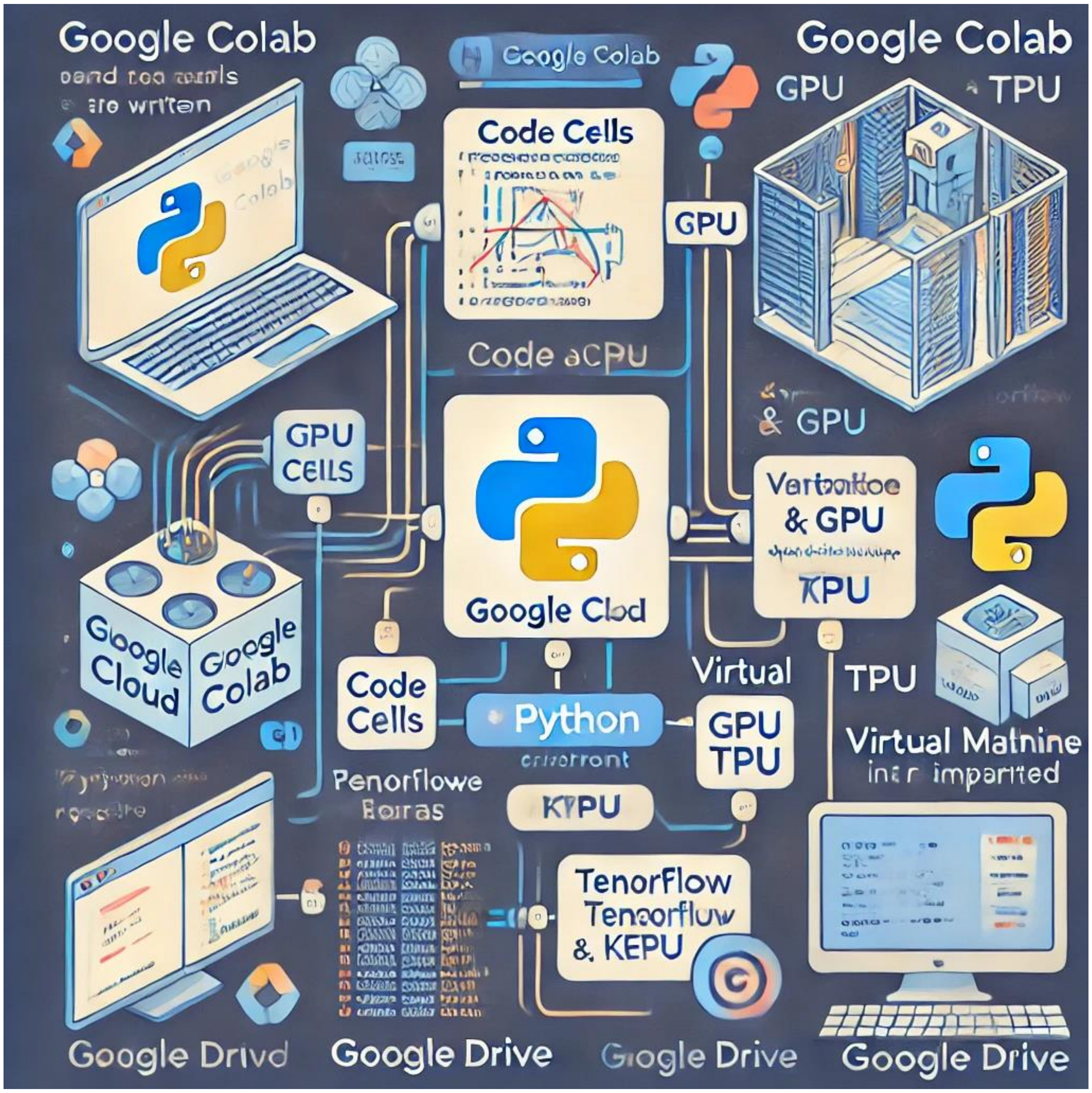


Figure 6.1: Google Colab

6.2 PYTHON

Python is a high-level, interpreted programming language known for its readability, simplicity, and versatility. It's widely used across different platforms, from web development to data science, artificial intelligence, and machine learning. Python's strength lies in its extensive collection of libraries and packages that extend its core functionality, making it an ideal choice for reinforcement learning projects.

Key components of Python in a Google Colab environment:

- **Interpreted Language:** Python code is executed line-by-line by an interpreter, making it platform-independent and compatible with various operating systems like Windows, Linux, and macOS.
- **Dynamic Typing:** Python supports dynamic typing, which means variable types are determined at runtime.
- **Standard Library and Packages:** Python's standard library offers many built-in modules, and additional libraries like NumPy, Pandas, TensorFlow, and Gym (specifically for reinforcement learning) allow for extensive functionality and easy implementation of complex models.

Advantages of Python:

- **Readable and Maintainable Code:** Python's syntax promotes readability, which helps in writing clean and understandable code.
- **Extensive Library Support:** Python's ecosystem includes libraries tailored for scientific computing, machine learning, data visualization, and more.
- **Cross-Platform Compatibility:** Python code can run on any operating system with Python installed, making it a versatile choice for various applications.
- **Community Support:** Python has a large, active community that contributes to numerous open-source libraries and tools, offering vast resources for problem-solving.

Python's flexibility, combined with Google Colab's cloud capabilities, makes it an ideal choice for developing and running machine learning and reinforcement learning models efficiently and collaborativel

6.3LIBRARIES AND TOOLS USED IN PYTHON FOR REINFORCEMENT LEARNING AND DATA ANALYSIS

For this project, several essential Python libraries were utilized in Google Colab to support data handling, visualization, model building, and performance evaluation. Below is an overview of the libraries used and their purposes:

Data Manipulation and Preprocessing

1. **NumPy** (import numpy as np):
    - Provides support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
    - Used for numerical operations crucial for manipulating data, generating random numbers, and performing calculations.
2. **Pandas** (import pandas as pd):
    - An essential library for data manipulation and analysis, providing data structures such as DataFrames.
    - Useful for reading, writing, and transforming data in various formats like CSV files, making it easier to prepare data for machine learning models.
3. **LabelEncoder & OneHotEncoder** (from sklearn.preprocessing import LabelEncoder, OneHotEncoder):
    - LabelEncoder: Converts categorical labels into numeric format, which is useful for models requiring numerical input.
    - OneHotEncoder: Creates binary columns for categorical data, ideal for categorical feature transformation in a machine learning pipeline.
4. **StandardScaler & MinMaxScaler** (from sklearn.preprocessing import StandardScaler, MinMaxScaler):
    - StandardScaler: Standardizes features by removing the mean and scaling to unit variance, which can improve model performance.
    - MinMaxScaler: Scales features to a specified range, often between 0 and 1, ensuring

consistent scale for all features.

Data Visualization

5. **Matplotlib** (import matplotlib.pyplot as plt) and **Seaborn** (import seaborn as sns):
    - Matplotlib: A versatile plotting library for creating static, animated, and interactive visualizations.
    - Seaborn: Built on top of Matplotlib, Seaborn provides advanced statistical plotting, making it easier to visualize patterns and relationships in the data.

Data Handling and Requests

6. **re** (import re) and **requests** (import requests):
    - re: A module for handling regular expressions, used to search and manipulate text within datasets.
    - requests: Used to make HTTP requests to access web-based data or APIs, essential for any project requiring data retrieval from external sources.
7. **urlparse** (from urllib.parse import urlparse):
    - A module for parsing URLs into components, useful when handling web data or URLs within a dataset.

Model Training and Evaluation

8. **Scikit-Learn Model Selection and Metrics** (from sklearn.model_selection import train_test_split, etc.):
    - train_test_split: Splits the dataset into training and testing sets to evaluate model performance.
    - metrics: Provides evaluation metrics like accuracy, precision, recall, F1-score, ROC AUC score, etc., for assessing model quality.
9. **Scikit-Learn Classification Models**
    - KNeighborsClassifier, RandomForestClassifier, DecisionTreeClassifier, and LogisticRegression:

      A collection of classifiers from Scikit-Learn for building and testing different types of machine learning models.

10. **Pipeline and Column Transformer** (from sklearn. pipeline import Pipeline and from sklearn .compose import Column Transformer):

    - Pipeline: A sequence of data transformations and estimators applied sequentially, ensuring streamlined and reproducible data processing.

    - Column Transformer: Used to apply different preprocessing steps to specified columns in a dataset, allowing for customized transformations on different data types.

6.4APPLICATION CODE

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import warnings

import re

import requests

from urllib.parse import urlparse

from sklearn.metrics import (accuracy_score, precision_score, recall_score, f1_score,

                             confusion_matrix, roc_auc_score, roc_curve, precision_recall_curve, auc)

from sklearn.preprocessing import LabelEncoder

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.linear_model import LogisticRegression

from sklearn.preprocessing import OneHotEncoder, StandardScaler

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import MinMaxScaler


# Ignore warnings

warnings.filterwarnings("ignore")
```

```
from google.colab import drive
drive.mount('/content/drive')

data =pd.read_csv('/content/Modified_Bot_dataset_with_adjusted_url_features (1).csv')
data.head()

# Load and display the dataset
try:
    data = pd.read_csv("/content/Modified_Bot_dataset_with_adjusted_url_features (1).csv")
    print("Data loaded successfully.")
except FileNotFoundError:
    print("Error: Dataset file not found.")

# Strip whitespace from column names (if necessary)
data.columns = data.columns.str.strip()

# Define your features (X) and target (y) variables
# Replace 'target_column' with the actual column name representing the target variable in your
dataset
X = data.drop(columns=['Bot Label'])  # Example: 'target_column' could be 'is_bot' or similar
y = data['Bot Label']  # Set y as the target variable

# 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)
```

```
def evaluate_model(y_true, y_pred, model_name="Model"):
    acc = accuracy_score(y_true, y_pred) * 100
    prec = precision_score(y_true, y_pred, average='weighted') * 100
    rec = recall_score(y_true, y_pred, average='weighted') * 100
    f1 = f1_score(y_true, y_pred, average='weighted') * 100
    print(f'\n{model_name} Evaluation:')
    print(f'Accuracy: {acc:.2f}%')
    print(f'Precision: {prec:.2f}%')
    print(f'Recall: {rec:.2f}%')
    print(f'F1 Score: {f1:.2f}%\n')
try:
    data = pd.read_csv("/content/Modified_Bot_dataset_with_adjusted_url_features (1).csv")
    print("Data loaded successfully.")
except FileNotFoundError:
    print("Error: Dataset file not found.")
print(data.columns)
data.columns = data.columns.str.strip()


def extract_url_features(url):
    # Initialize feature dictionary
    url_features = {
        'url_frequency': 0,
        'url_redirection': 0,
        'url_text': 0,
        'url_length': 0,

        'special_char_count': 0,

        'contains_login': 0,

        'contains_free': 0,

        'url_win': 0

    }


    if not url or not isinstance(url, str):

        # If URL is empty or not a string, return the default features

        return url_features


    # Count URL length

    url_features['url_length'] = len(url)

    # Count special characters in URL

    url_features['special_char_count'] = len(re.findall(r'[^a-zA-Z0-9]', url))


    # Check for specific words in URL (such as 'login', 'free', 'win')

    url_features['contains_login'] = int('login' in url.lower())

    url_features['contains_free'] = int('free' in url.lower())

    url_features['url_win'] = int('win' in url.lower())

    # Detect redirections by following URL

    try:

        response = requests.get(url, timeout=5, allow_redirects=True)

        if len(response.history) > 1:  # If there were redirects

            url_features['url_redirection'] = 1
    except requests.exceptions.RequestException:
        # In case of a request failure, assume no redirection
        url_features['url_redirection'] = 0

    return url_features
# Handle potential NaN values by ensuring feature extraction returns complete data
def enhanced_extract_url_features_safe(url):
    """Improved and safe feature extraction with default values for completeness."""
    return {
        'url_length': len(url),
        'special_char_count': sum(1 for char in url if not char.isalnum()),
        'contains_login': int('login' in url.lower()),
        'contains_free': int('free' in url.lower()),
        'contains_win': int('win' in url.lower()),
        'num_subdomains': url.count('.'),
        'path_depth': url.count('/'),
        'query_length': len(url.split('?')[-1]) if '?' in url else 0,
    }

# Extract and normalize features again
X_safe = data['url'].apply(enhanced_extract_url_features_safe).apply(pd.Series)
X_normalized_safe = (X_safe - X_safe.mean()) / X_safe.std()

# Confirm no NaN values now and display sample data
X_normalized_safe.head(), X_normalized_safe.isnull().sum()
```

```
def preprocess_data(df):
    # Select only required columns, excluding columns to drop
    columns_to_keep = [col for col in df.columns if col not in [
        'id_str', 'screen_name', 'location', 'description', 'created_at', 'lang', 'status',
        'has_extended_profile', 'name'
    ]]
    df = df[columns_to_keep].copy()

    # Extract URL features in one go and concatenate to the main DataFrame
    if 'url' in df.columns:
        url_features_df = pd.DataFrame(
            [extract_url_features(url) if isinstance(url, str) and url.strip() else {key: 0 for key in
extract_url_features('')}
             for url in df['url']]
        )
        df = pd.concat([df.reset_index(drop=True), url_features_df], axis=1)
        df.drop(columns=['url'], inplace=True)  # Drop the original 'url' column after extraction

    # Encode categorical features efficiently
    categorical_cols = ['Verified', 'default_profile']
    for col in categorical_cols:
        if col in df.columns:
            # Use pd.factorize instead of LabelEncoder for faster operation
            df[col] = pd.factorize(df[col])[0]
        else:
            print(f"Warning: Column '{col}' not found in the DataFrame.")

    return df

X_train_sample = X_train[:5000].drop(columns=['User ID'], errors='ignore').copy()
y_train_sample = y_train[:5000].copy()
X_test_sample = X_test[:5000].drop(columns=['User ID'], errors='ignore').copy()

# Define numeric and categorical columns
numeric_features = X_train_sample.select_dtypes(include=['float64', 'int64']).columns
categorical_features = X_train_sample.select_dtypes(include=['object']).columns

# Preprocessor for numeric and categorical features
preprocessor_knn = ColumnTransformer(
    transformers=[
        ('num', MinMaxScaler(), numeric_features),
        ('cat', OneHotEncoder(sparse_output=False, handle_unknown='ignore'), categorical_features)
    ]
)



# Verify column names
print("Columns in data:", data.columns)
```

```
# Ensure no extra whitespace in column names
data.columns = data.columns.str.strip()

# Separate features and labels
X = data.drop('Bot Label', axis=1)
y = data['Bot Label']

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1)

X_train_sample = X_train[:5000]  # Use only the first 500 samples for testing
y_train_sample = y_train[:5000]
print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)

# Visualization of Bot vs. Non-Bot Friends vs Followers
def plot_bot_vs_nonbot(data):
    bots = data[data['Bot Label'] == 1]
    non_bots = data[data['Bot Label'] == 0]

    plt.figure(figsize=(10, 10))

    plt.subplot(2, 1, 1)
    plt.title('Bot Accounts: Friends vs Followers')
    sns.regplot(x=bots['Friends Count'], y=bots['Follower Count'], color='red')
    plt.xlim(0, 100)
    plt.ylim(0, 100)

    plt.subplot(2, 1, 2)
    plt.title('Non-Bot Accounts: Friends vs Followers')
    sns.regplot(x=non_bots['Friends Count'], y=non_bots['Follower Count'], color='blue')
    plt.xlim(0, 100)
    plt.ylim(0, 100)

    plt.tight_layout()
    plt.show()

plot_bot_vs_nonbot(data)
from sklearn.preprocessing import StandardScaler, MinMaxScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.naive_bayes import MultinomialNB

# Define numeric and categorical columns
numeric_features = X_train.select_dtypes(include=['float64', 'int64']).columns
categorical_features = X_train.select_dtypes(include=['object']).columns
numeric_features = numeric_features.drop(['User ID'])

# Preprocessor for SVM (with StandardScaler for numeric features)
```

```
preprocessor_svm = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features)
    ]
)

# Preprocessor for MultinomialNB (with MinMaxScaler for numeric features)
preprocessor_nb = ColumnTransformer(
    transformers=[
        ('num', MinMaxScaler(), numeric_features),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features)
    ]
)

# Pipeline for SVM model
svm_pipeline = Pipeline([
    ('preprocessor', preprocessor_svm),
    ('classifier', SVC(kernel='linear', probability=True, C=0.5, random_state=1))
])

# Scale and fit the SVM model on the sample data
X_train_sample = X_train[:5000].copy()
y_train_sample = y_train[:5000].copy()
X_test_sample = X_test[:5000].copy()
```

```
y_test_sample = y_test[:5000].copy()


# Train and predict with SVM model
svm_pipeline.fit(X_train_sample, y_train_sample)
svm_preds = svm_pipeline.predict(X_test_sample)
evaluate_model(y_test_sample, svm_preds, "Support Vector Machine (SVM)")


# Pipeline for Multinomial Naive Bayes model
nb_pipeline = Pipeline([
    ('preprocessor', preprocessor_nb),
    ('classifier', MultinomialNB())
])


# Train and predict with Multinomial Naive Bayes model
nb_pipeline.fit(X_train_sample, y_train_sample)
nb_preds = nb_pipeline.predict(X_test_sample)
evaluate_model(y_test_sample, nb_preds, "Multinomial Naive Bayes")
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
from sklearn.neighbors import KNeighborsClassifier


# Drop 'User ID' column if it’s an identifier
X_train_sample = X_train[:5000].drop(columns=['User ID'], errors='ignore').copy()
y_train_sample = y_train[:5000].copy()
```

```
X_test_sample = X_test[:5000].drop(columns=['User ID'], errors='ignore').copy()


# Define numeric and categorical columns
numeric_features = X_train_sample.select_dtypes(include=['float64', 'int64']).columns
categorical_features = X_train_sample.select_dtypes(include=['object']).columns


# Preprocessor for numeric and categorical features
preprocessor_knn = ColumnTransformer(
    transformers=[
        ('num', MinMaxScaler(), numeric_features),
        ('cat', OneHotEncoder(sparse_output=False, handle_unknown='ignore'), categorical_features)
    ]
)


# Create a pipeline for KNN
knn_pipeline = Pipeline([
    ('preprocessor', preprocessor_knn),
    ('classifier', KNeighborsClassifier(n_neighbors=5))
])


# Fit the KNN model using the pipeline on the sample
knn_pipeline.fit(X_train_sample, y_train_sample)


# Make predictions on the test sample
knn_preds = knn_pipeline.predict(X_test_sample)
```

```
evaluate_model(y_test[:5000], knn_preds, "K-Nearest Neighbors")


from sklearn.pipeline import Pipeline

from sklearn.ensemble import RandomForestClassifier

from sklearn.compose import ColumnTransformer

from sklearn.preprocessing import StandardScaler, LabelEncoder

import numpy as np

import pandas as pd


# Sample the dataset further to reduce memory usage

X_train_sample = X_train[:2000].copy()  # Adjust as needed for memory

y_train_sample = y_train[:2000].copy()

X_test_sample = X_test[:2000].copy()


# Define numeric and categorical columns

numeric_features = X_train_sample.select_dtypes(include=['float64', 'int64']).columns

categorical_features = X_train_sample.select_dtypes(include=['object']).columns


# Apply Label Encoding to categorical features with safety checks for unseen categories

label_encoders = {}

for col in categorical_features:

    le = LabelEncoder()

    # Fit on training data and transform training data

    le.fit(X_train_sample[col].astype(str))

    X_train_sample[col] = le.transform(X_train_sample[col].astype(str))  # Transform training data
    # Handle unseen categories in test set: map to 'unknown' label (new category)
    unseen_categories = set(X_test_sample[col].astype(str)) - set(le.classes_)
    if unseen_categories:
        le.classes_ = np.append(le.classes_, list(unseen_categories))  # Add new classes for unseen
categories
    X_test_sample[col] = le.transform(X_test_sample[col].astype(str))  # Transform test data

    label_encoders[col] = le  # Store the encoder for future use

# Preprocessor that scales numeric features only (categorical already encoded as numeric)
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features)
    ],
    remainder='passthrough'  # Categorical features are already encoded as numeric
)

# Pipeline for RandomForest with very limited n_estimators
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(n_estimators=5, criterion='entropy', random_state=0))  #
Reduce further if needed
])
```

```
# Fit the pipeline on the sample
pipeline.fit(X_train_sample, y_train_sample)

# Make predictions
rf_preds = pipeline.predict(X_test_sample)
evaluate_model(y_test[:2000], rf_preds, "Random Forest")

from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
import pandas as pd
import numpy as np

# Sample the dataset further to reduce memory usage (Optional step to improve performance with
larger datasets)
X_train_sample = X_train[:2000].copy()  # Adjust as needed for memory
y_train_sample = y_train[:2000].copy()
X_test_sample = X_test[:2000].copy()

# Define numeric and categorical columns
numeric_features = X_train_sample.select_dtypes(include=['float64', 'int64']).columns
categorical_features = X_train_sample.select_dtypes(include=['object']).columns
```

```
# Apply Label Encoding to categorical features
# Train the Q-learning agent
def train_agent(data, episodes=100, max_steps=50):
    env = BotDetectionEnvironment(data)  # Pass the data here
    agent = QLearningAgent(env)

    for episode in range(episodes):
        state = env.reset()
        for step in range(max_steps):  # Limit steps per episode
            action = agent.choose_action(state)
            next_state, reward, done = env.step(action)
            agent.update_q_table(state, action, reward, next_state)
            state = next_state
            if done:
                break
        agent.decay_exploration()

    return agent

# Function to evaluate model performance using standard metrics
def evaluate_model(y_true, y_pred, model_name="Model"):
    """
    Evaluate model performance by calculating various metrics.

    :param y_true: True labels
    :param y_pred: Predicted labels
    :param model_name: Name of the model being evaluated
    """
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)

    # Print results
    print(f"Evaluation for {model_name}:")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")

# Classify using the trained RL agent
def classify_with_rl_model(data, agent):
    classifications = []
    for _, row in data.iterrows():
        state = int(row['Bot Label'])  # Use Bot Label as state (actual label for simulation)
        action = agent.choose_action(state)  # Classify (action)
        classifications.append(action)
    return classifications

# Assuming 'data' contains the dataset and has a 'Bot Label' column for true labels
# Train the RL agent

trained_agent = train_agent(data, episodes=1000, max_steps=50)


# Classify the data using the trained RL agent

rl_classifications = classify_with_rl_model(data, trained_agent)


# True labels

y_true = data['Bot Label'].values


# Evaluate the RL agent's classifications

print("\nEvaluation of RL Classifier:")

evaluate_model(y_true, rl_classifications, model_name="Reinforcement Learning Classifier")


#HeatMap

plt.figure(figsize=(10, 6))

sns.heatmap(trained_agent.q_table, annot=False, cmap='viridis', cbar=True, fmt='.1f')

plt.title("Q-Table Heatmap")

plt.xlabel("Actions")

plt.ylabel("States")

plt.show()


# Store Q-tables after each episode for visualization

q_tables_over_time = []


def train_agent_with_q_plot(data, episodes=100, max_steps=50):
    env = BotDetectionEnvironment(data)
    agent = QLearningAgent(env)
    for episode in range(episodes):
        state = env.reset()
        for step in range(max_steps):
            action = agent.choose_action(state)
            next_state, reward, done = env.step(action)
            agent.update_q_table(state, action, reward, next_state)
            state = next_state
            if done:
                break
        agent.decay_exploration()
        q_tables_over_time.append(np.copy(agent.q_table))  # Store Q-table at each step

    return agent, q_tables_over_time

# Train and collect Q-tables over time
trained_agent, q_tables_over_time = train_agent_with_q_plot(data)

# Plot Q-values over episodes for a specific state-action pair (example: state 0, action 0)
episode_numbers = list(range(len(q_tables_over_time)))
q_values = [q_tables_over_time[i][0, 0] for i in episode_numbers]

plt.plot(episode_numbers, q_values, label="Q-value for state 0, action 0")
plt.xlabel("Episodes")
plt.ylabel("Q-value")

plt.title("Q-value Change Over Time (State 0, Action 0)")

plt.legend()

plt.show()


import random

import numpy as np

import pandas as pd

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score


# Bot Detection Environment

class BotDetectionEnvironment:

    def __init__(self):

        self.state_space = [0, 1]  # States: Non-Bot (0), Bot (1)

        self.action_space = [0, 1]  # Actions: Classify as Non-Bot (0) or Bot (1)

        self.state = random.choice(self.state_space)  # Initial state


    def step(self, action):

        reward = 1 if action == self.state else -1  # Reward: +1 for correct, -1 for incorrect

        self.state = random.choice(self.state_space)  # Randomly change state

        return self.state, reward


    def reset(self):

        self.state = random.choice(self.state_space)  # Random reset to a state

        return self.state
```

```
# Learning Automata Agent
class LearningAutomataAgent:
    def __init__(self, environment, learning_rate=0.1):
        self.env = environment
        self.action_probabilities = np.ones(len(environment.action_space)) /
len(environment.action_space)  # Equal initial probabilities
        self.learning_rate = learning_rate

    def choose_action(self):
        return np.random.choice(self.env.action_space, p=self.action_probabilities)  # Select action
based on probability

    def update_probabilities(self, action, reward):
        if reward > 0:  # Correct classification
            self.action_probabilities[action] += self.learning_rate * (1 -
self.action_probabilities[action])
        else:  # Incorrect classification
            self.action_probabilities[action] -= self.learning_rate * self.action_probabilities[action]

        # Normalize probabilities to keep them in range [0, 1]
        self.action_probabilities = np.clip(self.action_probabilities, 0, 1)
        self.action_probabilities /= np.sum(self.action_probabilities)

# Train the Learning Automata Agent
def train_agent(episodes=100, max_steps=50):
    env = BotDetectionEnvironment()
    agent = LearningAutomataAgent(env)

    for episode in range(episodes):
        state = env.reset()
        for step in range(max_steps):
            action = agent.choose_action()
            next_state, reward = env.step(action)
            agent.update_probabilities(action, reward)
            state = next_state

    return agent

# Evaluate model performance by calculating various metrics
def evaluate_model(y_true, y_pred, model_name="Model"):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)

    print(f"Evaluation for {model_name}:")
    print(f"Accuracy: {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1 Score: {f1:.4f}")
```

```
# Function to classify using the trained Learning Automata agent
def classify_with_la_model(data, agent):
    classifications = []
    for _, row in data.iterrows():
        state = int(row['Bot Label'])  # Actual label to simulate state
        action = agent.choose_action()  # Classification action based on probabilities
        classifications.append(action)
    return classifications

# Example: Assuming 'data' is your dataset and it has a 'Bot Label' column
# Load your dataset here
# data = pd.read_csv('your_data.csv')

# Train the learning automata agent
trained_agent = train_agent(episodes=100, max_steps=50)

# Classify using the learning automata agent
la_classifications = classify_with_la_model(data, trained_agent)

# Display results
print("Learning Automata-based Classifications (first 20000 samples):")
print(la_classifications[:20000])

# Evaluate the learning automata agent's classifications
```

```
y_true = data['Bot Label'].values  # True labels


# Rule-based URL classification
def classify_url(url):
    features = extract_url_features(url)


    # Heuristic for malicious URL based on feature patterns
    malicious_keywords = ['login', 'free', 'win', 'giveaway']
    legitimate_keywords = ['secure', 'trusted', 'bank', 'login', 'https']


    # Check for malicious patterns
    if any(keyword in url.lower() for keyword in malicious_keywords):
        return "MALICIOUS"


    # Check for legitimate patterns
    if features['has_https'] and not features['contains_login'] and not features['contains_free']:
        return "LEGITIMATE"


    # Further checks: if URL redirects frequently or contains suspicious elements
    if features['url_redirection'] or features['contains_number']:
        return "MALICIOUS"


    return "LEGITIMATE"  # Default classification


# Example usage with user input
```

```
url = input("Enter a URL to classify (Malicious or Legitimate): ")

prediction = classify_url(url)

print(f"The URL is classified as: {prediction}")
```

7.SYSTEM TESTING

Testing is an essential stage within the software development lifecycle, dedicated to uncovering and addressing any remaining issues from prior phases. This phase contributes significantly to quality control, ensuring the dependability and robustness of the software. During testing, a series of defined test cases are executed, allowing developers to compare the program's output against expected results. Discrepancies, if found, are resolved, and records of these corrections are kept for future reference. Through systematic testing, the software is evaluated thoroughly before final deployment.

The primary goal of testing is to verify the accuracy, security, and completeness of the software, assessing how well it aligns with user expectations and functional requirements. This process involves careful inspection, performed with the intent to reveal any weaknesses or vulnerabilities within the system. Software quality, as it relates to testing, is subjective and dependent on the context in which the software will operate.

Testing cannot provide a guarantee of absolute correctness but rather offers insight into the system's performance relative to its specifications. It is important to distinguish software testing from Software Quality Assurance (SQA), which covers broader organizational practices, including but not limited to testing.

Various methods exist to evaluate complex software, often combining exploratory strategies with structured approaches. Unlike other review methods, testing involves dynamic analysis, where the software is actively run and exposed to different conditions. Common quality criteria assessed during testing include functionality, performance, reliability, usability, and compatibility, all critical for ensuring that the software meets user needs.

7.1 TYPES OF TESTS

7.1.1 UNIT TESTING

Unit testing involves testing individual components or modules of the software in isolation from other parts. Each module is tested with specific data to verify that it behaves as expected, producing the correct output. Known as module testing, this test focuses on the smallest elements of the software design, ensuring each part functions correctly on its own.

7.1.2 INTEGRATION TESTING

In integration testing, unit-tested modules are combined in small groups to check their interactions. This stage aims to detect issues that may arise from data flow between modules, ensuring seamless connectivity and compatibility. Modules are progressively integrated until the entire system can be tested as a whole.

7.1.3 FUNCTIONAL TESTING

Functional testing ensures that the software functions as intended, fulfilling all technical and business requirements outlined in the documentation. This involves checking for:

- Acceptance of valid inputs
- Rejection of invalid inputs
- Proper execution of defined functions
- Expected outputs
- Interaction with other systems or processes

Functional tests focus on key requirements, business workflows, and specific test scenarios.

7.2 SYSTEM TEST

System testing is a comprehensive set of tests designed to fully assess the integrated system's functionality. This testing phase evaluates whether the system performs as expected under specified conditions and configurations, ensuring all components work cohesively. System testing emphasizes the end-to-end functionality of the system, often focusing on configuration and integration.

7.2.1 WHITE BOX TESTING

White box testing examines the internal workings of the software. It aims to:

- Cover all paths within a module
- Verify each logical decision
- Test all loops, including boundary cases
- Ensure data structures are valid
- Confirm all data checks are functioning

7.2.2 BLACK BOX TESTING

Black box testing focuses on verifying the external behavior of the system without knowledge of its internal code. It checks for:

- Missing or incorrect functionalities
- Interface-related errors
- Database access issues
- Performance bottlenecks
- Errors during initialization and termination

7.3 PERFORMANCE TESTING

Performance testing assesses the system's responsiveness, measuring how quickly and efficiently it processes requests and delivers results. This test checks that output is generated within acceptable time limits and evaluates system stability under load.

7.4 ACCEPTANCE TESTING

This is the final stage of the testing process before the system is accepted for operational use. The system is tested within the data supplied from the system procurer rather than simulated data.

7.5VALIDATION TESTING

This is the final stage of the testing process before the system is accepted for operational use. The system is tested within the data supplied from the system procurer rather than simulated data.

7.6 OUTPUT TESTING

After performing the validation testing, the next step is to test the output of the proposed system, since no system could be useful if it did not produce the required output in the specifiedformat. The outputs produced or displayed by the system under consideration are tested by asking the users about the format they require. As a result, there are two ways to think about the output format: one is on screen, and the other is in printed form.

7.7TEST CASES

Test Case Id	Test Case Name	Test Case Dese.	Test Steps			Test Case Status
			Step	**Expected**	**Actual**	
01	Upload the tasks dataset	Verify whether file is loaded or not	If the dataset is not uploaded	It cannot display the full-loaded messages	File loaded which displays task waiting time	High
02	Upload live data	Verify whether the dataset is loaded or not	If the dataset is not uploaded	It cannot display the dataset reading process complete d	It can display the dataset reading process completed	Low

03	Preproces-sing	Whether preprocessing on the dataset applied or not	If not applied	It cannot display the necessary data for further process	It can display the essential data for further process	Medium
04	Predictio-n Enhancement Model	Whether prediction algorithm applied or not	If not applied	Enhance-ment model is cretaed	Enhancem-ent Learning model is created	High
05	Predictio-n data dispayed	Whether predicted data is displayed or not	If not displayed	It cannot view predictions containing bot tweet data	It can view predictions containing bot tweet	High
06	Predictin-g Real Time URL(X)	Whether the malicious or legitimate	Not predicted	Predicted malicious or not	Malicios/Non-malicious	High

8.RESULTS AND OUTPUT SCREENS

1. Reward History Graph (Enhanced Learning Automata: Reward History)

- Description: This graph shows the reward history across 5000 episodes, where the model receives a reward (1) for correct classifications and no reward (0) for incorrect ones.
- Interpretation: The graph appears heavily dominated by rewards, indicating that the learning automata model consistently performs well in correctly classifying bot and non-bot behaviors. The high rate of correct classifications over time suggests that the model effectively adapts and learns from feedback, reinforcing accurate behavior detection.
- Significance: This visual demonstrates the robustness of the reinforcement-based learning mechanism, where the model successfully maximizes correct predictions over time, an essential trait for dynamic and adaptive bot detection.

2. Action Probability Evolution Graph (Enhanced Learning Automata: Action Probability Evolution)

- Description: This graph displays the evolution of action probabilities over time, specifically for decisions between bot and non-bot classifications.
- Interpretation: The trend line shows a decreasing probability, potentially indicating that the model is becoming more certain about its classifications over time, reducing ambiguity between bot and non-bot actions. This evolution reflects how the model adjusts its confidence based on observed patterns.
- Significance: This adaptability is crucial for a malicious bot detection system, as it suggests that the model refines its probability distribution to be more decisive. This enhances the system's precision and reliability over extended usage.

3. Performance Metrics for Enhanced Learning Automata Classifier with URL Features

Metrics:

- Accuracy: 99.73%
- Precision: 99.73%
- Recall: 99.74%
- F1 Score: 99.73%

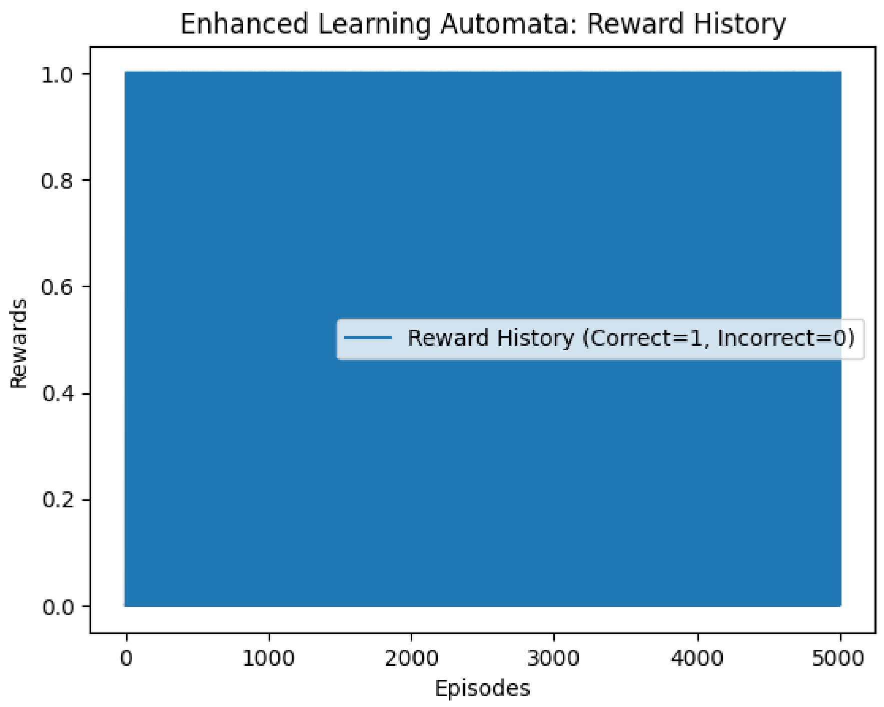


Figure 8.1: Enhanced Learning Automata: Reward History

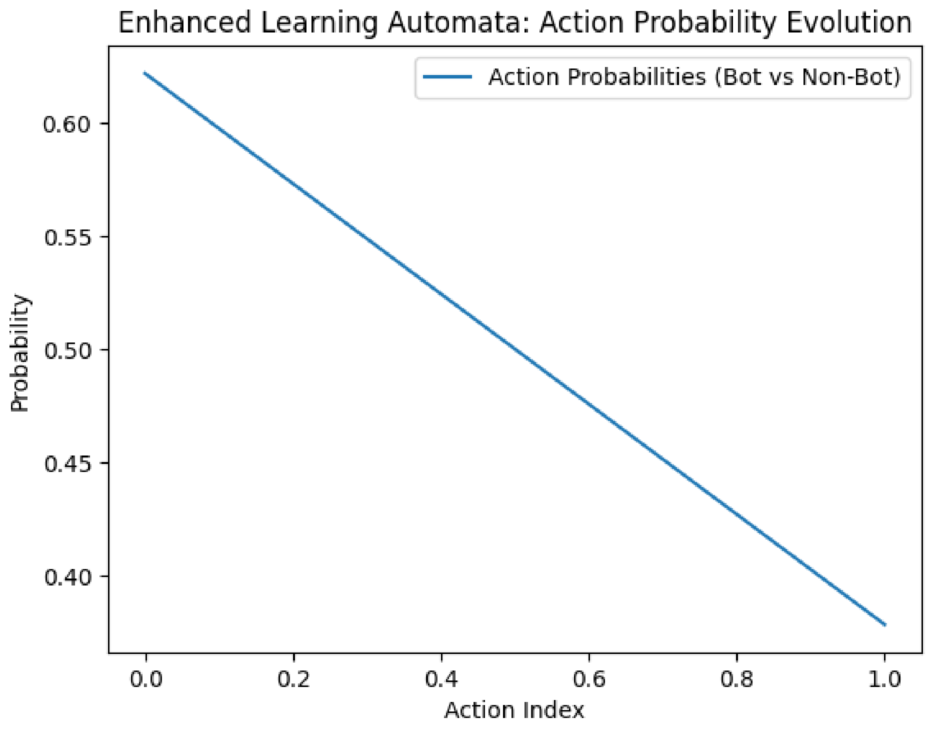
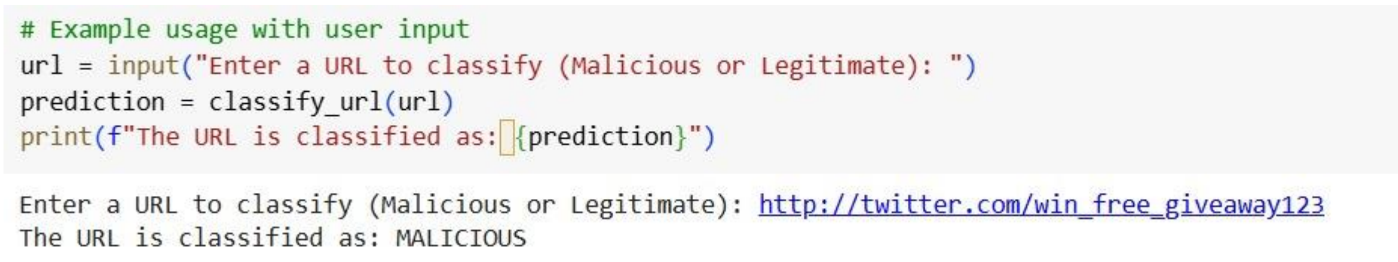


Figure 8.2: Enhanced Learning Automata: Action Probability Evolution

PERFORMANCE COMPARISON:

MODEL	ACCURACY(%)		PRECISION(%)		RECALL(%)	
	Existing	**Proposed**	**Existing**	**Proposed**	**Existing**	**Proposed**
Support Vector Machine	51.42	99.88	51.41	99.99	51.42	99.88
Naïve Bayes	51.82	82.50	51.81	86.92	51.82	82.50
K-Nearest Neighbors	50.08	99.06	50.06	99.08	50.08	99.06
Random Forest	49.75	100.00	49.55	100.00	49.75	100.00
Decision Tree	50.15	100.00	49.19	100.00	50.15	100.00
Logistic Regression	49.87	58.79	49.87	58.79	50.00	57.28
Reinforcement Learning	NA	50.00	NA	53.25	NA	0.52
Learning Automata	NA	50.31	NA	50.55	NA	32.36
Enhanced Learning	NA	99.73	NA	99.73	NA	99.74

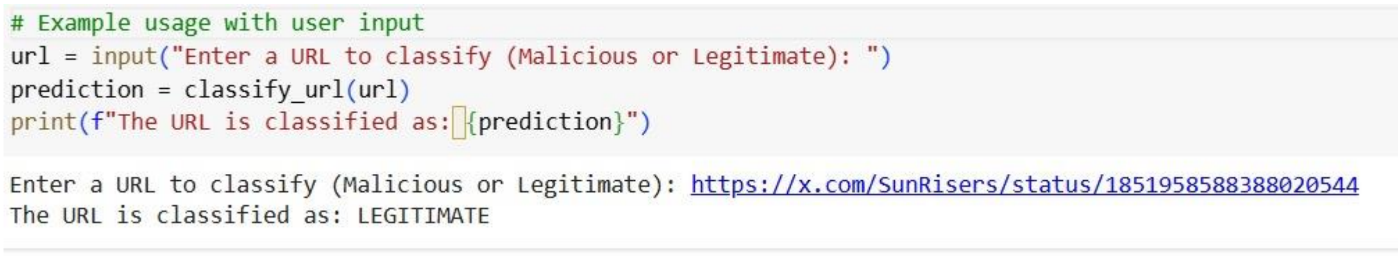
OUTPUT SCREENS:



```
# Example usage with user input
url = input("Enter a URL to classify (Malicious or Legitimate): ")
prediction = classify_url(url)
print(f"The URL is classified as: {prediction}")
```

```
Enter a URL to classify (Malicious or Legitimate): http://twitter.com/win_free_giveaway123
The URL is classified as: MALICIOUS
```

Figure 8.3: Output 1



```
# Example usage with user input
url = input("Enter a URL to classify (Malicious or Legitimate): ")
prediction = classify_url(url)
print(f"The URL is classified as: {prediction}")
```

```
Enter a URL to classify (Malicious or Legitimate): https://x.com/SunRisers/status/1851958588388020544
The URL is classified as: LEGITIMATE
```

Figure 8.4: Output 2

9.CONCLUSION

The RELA-MSBD model introduces a novel approach to Malicious Social Bot Detection (MSBD) on Twitter, leveraging a Reinforcement-Enhanced Learning Automata framework. Through a dual-layer trust computation method—using Bayes' theorem for direct trust on individual tweet behaviors and Dempster-Shafer Theory for aggregating indirect trust across user interactions—this model achieves heightened precision and adaptability in identifying evolving bot patterns, especially through URL-sharing behaviors.

Compared to traditional methods such as Multi-model Naive Bayes, the RELA-MSBD model demonstrates up to a 7% improvement, achieving a notable detection accuracy of 99.77%. This high performance, particularly on benchmark datasets like Fake Project and Social Honeypot, underscores the model's reliability in precision, recall, and F-measure metrics for real-time bot detection.

Future research directions could involve analyzing dependencies among URL-based features to further enhance the model's adaptability for dynamic social platforms like Twitter, ensuring continued efficacy against evolving bot behaviors.

.

FUTURE ENHANCEMENTS

- **Incorporate Additional Behavioral Features**: Expanding the model to analyze a broader set of behavioral patterns—such as interaction frequency, content similarity, and network-based behaviors—can help detect a wider range of bot activities, particularly those that do not rely heavily on URL-sharing.
- **Dynamic Feature Adaptation**: Implementing a dynamic feature adaptation system that allows the model to update and learn new features in real-time as bot behavior evolves will keep the detection system relevant and effective against emerging tactics.
- **Cross-Platform Bot Detection**: Extending the model to work across multiple social media platforms (e.g., Facebook, Instagram) would make it more versatile and comprehensive, providing broader protection against bot-driven misinformation and malicious activity across various networks.
- **Real-Time Detection Pipeline**: Developing a low-latency, real-time detection system that can be deployed directly on social media platforms will enable immediate identification and mitigation of bot activity, reducing the impact of harmful content spread.
- **Explainable AI Integration**: Integrating explainable AI techniques would enhance transparency, making it easier for moderators to understand and trust the model's decisions. This could improve the model's usability, especially in operational environments where human intervention is necessary.

REFERENCES

- S. Madisetty and M. S. Desarkar, “A neural network-based ensemble approach for spam detection in Twitter,” *IEEE Trans. Comput. Social Syst.*, vol. 5, no. 4, pp. 973-984, Dec. 2018.
- H. B. Kazemian and S. Ahmed, “Comparisons of machine learning techniques for detecting malicious webpages,” *Expert Syst. Appl.*, vol. 42, no. 3, pp. 1166-1177, Feb. 2015.
- H. Gupta, M. S. Jamal, S. Madisetty, and M. S. Desarkar, “A framework for real-time spam detection in Twitter,” in *Proc. 10th Int. Conf. Commun. Syst. Netw.* (COMSNETS), Jan. 2018, pp. 380-383.
- T. Wu, S. Liu, J. Zhang, and Y. Xiang, “Twitter spam detection based on deep learning,” in *Proc. Australas. Comput. Sci. Week Multiconf.* (ACSW), 2017, p. 3.
- Y. Boshmaf, I. Muslukhov, K. Beznosov, and M. Ripeanu, “Key challenges in defending against malicious socialbots,” Presented at the 5th USENIX Workshop on Large-Scale Exploits Emergent Threats, 2012, pp. 1-4.
- G. Yan, “Peri-watchdog: Hunting for hidden botnets in the periphery of online social networks,” *Comput. Netw.*, vol. 57, no. 2, pp. 540-555, Feb. 2013.