

**X (Twitter) Bot Detection Using RNN and Random Forest Models**

**With The Ensemble Method**

A Thesis

Presented to the Faculty of

School of Computing and Information Technologies

Asia Pacific College

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Science

by

|  |  |
| --- | --- |
| Carl Von Nicanor | Johanne Marlowe Sanchez |
| Henzon Ivan Valguna | Gabrielle Ann Yogawin |

2024

**Table of Contents**

[**INTRODUCTION** 4](#_Toc182891994)

[**1.1** **CONTEXT OF THE STUDY** 6](#_Toc182891995)

[**1.2**  **STATEMENT OF THE PROBLEM** 8](#_Toc182891996)

[**1.3**  **RESEARCH QUESTIONS** 9](#_Toc182891997)

[**1.4** **RESEARCH OBJECTIVES** 9](#_Toc182891998)

[**1.5** **SCOPE AND LIMITATIONS** 10](#_Toc182891999)

[**1.6** **SIGNIFICANCE OF THE STUDY** 11](#_Toc182892000)

[**REVIEW OF RELATED LITERATURE** 13](#_Toc182892001)

[**2.1 THEORETICAL BACKGROUND** 21](#_Toc182892002)

[**2.2 TECHNICAL BACKGROUND** 35](#_Toc182892003)

[**SYNTHESIS** 43](#_Toc182892004)

[**METHODOLOGY** 45](#_Toc182892005)

[**3.1 Data Collection** 46](#_Toc182892006)

[**3.2 Data Preprocessing** 46](#_Toc182892007)

[**3.3 Model Training and Testing** 49](#_Toc182892008)

[**3.4 Conclusion** 51](#_Toc182892009)

[**RESULTS AND DISCUSSION** 54](#_Toc182892010)

[**4.1 DATA OVERVIEW** 54](#_Toc182892011)

[**4.2 MODEL IMPLEMENTATION AND PERFORMANCE** 57](#_Toc182892012)

[**4.3 INTERPRETATION ANALYSIS** 59](#_Toc182892013)

[**4.4 INTERPRETATION OF RESULT** 70](#_Toc182892014)

[**CONCLUSION** 73](#_Toc182892015)

[**RECOMMENDATIONS** 75](#_Toc182892016)

[**REFERENCES** 76](#_Toc182892017)

**LIST OF FIGURES**

[Fig. 1. Methodology Steps 46](#_Toc183424915)

[Fig. 2. Bot Detection Dataset 47](#_Toc183424916)

[Fig. 3. Text to Sequences Method 48](#_Toc183424917)

[Fig. 4. Sequences to Text 49](#_Toc183424918)

[Fig. 5. Handle Missing Values Function 51](#_Toc183424919)

[Fig. 6. Removing NaN rows Function 51](#_Toc183424920)

[Fig. 7. Check for NaN Function 52](#_Toc183424921)

[Fig. 8. Removing Duplicates Function 53](#_Toc183424922)

[Fig. 9. IQR Outlier Handling 54](#_Toc183424923)

[Fig. 31. ROC curve result for Random Forest 142](#_Toc183424924)

**LIST OF TABLES**

[Table I DATASET COLUMN NAMES 90](#_Toc183423937)

[Table II EXTRACTING DATE-TIME FEATURES 92](#_Toc183423938)

[Table III HANDLING MISSING VALUES 94](#_Toc183423939)

[Table IV ADDRESSING INCONSISTENCIES 95](#_Toc183423940)

[Table V COMPUTATION OF IQR 96](#_Toc183423941)

[Table VI COMPUTATION OF MIN-MAX SCALING 98](#_Toc183423942)

[Table VII HYPERPARAMETERS FOR RNN 106](#_Toc183423943)

[Table VIII HYPERPARAMETERS FOR RANDOM FOREST 111](#_Toc183423944)

[Table IX HYPERPARAMETERS FOR STACKED MODEL 116](#_Toc183423945)

[Table X HYPERPARAMETERS FOR 1D-CNN 120](#_Toc183423946)

[Table XI SUMMARY OF HYPERPARAMETERS OF MODELS 125](#_Toc183423947)

[Table XII SUMMARY OF CONFUSION MATRICES RESULTS 131](#_Toc183423948)

[Table XIII CROSS VALIDATION COMPARISON OF MODELS 137](#_Toc183423949)

**CHAPTER 1**

# **INTRODUCTION**

Most people nowadays use social media platforms in their everyday lives, having accounts on multiple sites, sharing personal information and experiences, connecting with other people. One of the popular social media applications is ‘X’ (previously known as Twitter), with 10.69 million active users in the Philippines as of 2024 [1]. To clarify ‘X’ and Twitter are still the same platform with the same metadata. This is where users could make tweets (the app’s term for a post), with the feature to interact with other people’s tweets in various ways [2]. However, sometimes these tweets are not created by real humans, but by bots, fake accounts that are programed for either neutral or malicious purposes from increasing post interaction activity to spreading misinformation and propaganda through their tweets [3].

Because of the site’s widespread popularity and the tendency for users to follow influencers with interesting content, ‘X’ has become a significant focus for marketing efforts and social manipulation. With this, malicious bot accounts are made to manipulate tweets’ activity, sharing fake news to exploit people’s opinions or private information [4]. This is further proved since cybersecurity company CHEQ monitored the percentage of bot traffic from X to its clients' websites over the course of three days in February. Compared to less than 3% of traffic from Facebook, Instagram, and TikTok being fake, it was discovered that 75% of traffic from X was fraudulent due to the bot accounts [49]. One example of this is the 2016 U.S. Presidential Election where 1 out of 5 tweets that were connected to this topic was apparently from bot accounts, talking about politics, its issues and events, affecting the election [5]. Another is a coordinated network of more than 1,200 platform X (previously Twitter) accounts spread the conspiracy theory that Donald Trump won the 2020 US presidential election, according to a recent analysis. These accounts amassed more than 3 million impressions. This network is a component of a larger, more extensive bot network made up of 1,305 distinct accounts arranged into different groups. This network includes false news sources, one of the biggest coordinated troll networks in support of Donald Trump, and a clickbait Pro-Trump network [50].

This study aims to develop an advanced method for detecting bot accounts on "X", using an ensemble learning approach that integrates Recurrent Neural Networks (RNN) and Random Forest models. By combining the strengths of these methodologies, the study seeks to achieve high accuracy while enhancing the interpretability of predictions. This ensures that both technical stakeholders and end-users can understand the rationale behind the model's classifications. Through this innovative approach, the research not only addresses the technical challenges in bot detection but also contributes to building trust and transparency in social media analytics, paving the way for more secure and reliable online interactions.

‘X’ bots have been evolving over the years, becoming more intelligent in acting like an actual human-made account to increase interactions with other accounts, may it be another bot or a real person [6]. Bot detection methods are also catching up to identify varying kinds of malicious bots such as using different machine learning methods. One of these methods is called Deep Learning, a method in artificial intelligence (AI) that teaches computers to process data in a way that is inspired by the human brain. Deep learning models can recognize complex patterns in pictures, text, sounds, and other data to produce accurate insights and predictions [7]. Another study highlights the need for interpretability in deep learning as it helps understand the models and validating their predictions where it is crucial as they are considered as “black boxes” due to their complex internal mechanism wherein they are difficult to explain and can’t provide a reasoning basis. To sum it up, interpretation helps us understand the characteristics of what distinguishes between bots and humans by doing so it will improve the detection algorithms and understanding the strategies used by the bot which would ultimately lead to having a much more effective and reliable detection system [8].

## **1.1** **CONTEXT OF THE STUDY**

Bot detection is a process that identifies and blocks bots that try to access websites with malicious intent. Bot detection is important since it will safeguard websites that can result in financial loss, data breaches, and reputational damages [51]. The method that is going to be used is stacking or stacked generalization method which is an ensemble learning method of using the strengths of different models while mitigating the weaknesses to produce more accurate predictions [9]. In terms of efficiency, Deep Learning techniques are computation and time-efficient techniques, showing high potential to detect malicious bots and keep pace with their fast-evolving variable characteristics [10]. However, the stacked generalization approach is proven to have high accuracy in detecting bots on social networking sites according to a study published in 2021 titled “Bot Detection in Social Networks Using Stacked Generalization Ensemble” [11]. In recent studies, it states that the reliance in accuracy and not much considering interpretation is an issue in detecting bots in social media and raises the need for interpretability where understanding why a model made a certain prediction can be just as important as the prediction itself as it may be able to give insights in finding out the reasons why [12].

The team decides to explore the potential of the stacked generalization approach that stacks Recurrent Neural Networks (RNNs), a type of artificial neural network that is well-suited for processing sequential data or time series data [13], and Random Forest models, a popular machine learning algorithm that combines the output of multiple decision trees to reach a single result [14]. The team chose RNN because they are designed to work with sequence data, since data from social media including user likes, and series of posts can be used to study the behavior and be able to aim high accuracy in detecting it as they often exhibit patterns or behavior over time [15]. However, there are studies that CNN was also used in detecting Twitter bots and was able to give out high accuracy, but the context with CNN is that they show superior performance in related to computer visions, which is in other words better suited in image recognition and processing, compared to RNN that is more suited in this study due to its ability in processing sequential data or time series data. The study also recommended exploring the use of ensemble learning techniques, like the stacked generalization approach, by stacking multiple models [16] which led to the application of Random Forest. As for Random Forests, they may be able to provide interpretability as it provides stability making them a good candidate in improving the performance of interpretable algorithms and a measure of the importance of feature identifying which features are most useful in differentiating between different classes not only that, but it also combines the predictions of several base estimators in improving the generalizability and its robustness [15][17].

Ensembling techniques, like the stacked generalization approach, typically combine multiple models to improve predictive performance. However, this combination can make the decision-making process opaque. [18] Interpretability helps stakeholders understand how decisions are made, which helps to have trust in the model’s predictions. [19] Interpretable models should be used instead of black-box models in high-stakes decisions due to the need for accountability and transparency. [20] LIME (Local Interpretable Model-agnostic Explanations) provides a way to interpret these complex models, and it is done by approximating the stacked model locally with an interpretable model, such as a linear model, to explain individual predictions. Both RNN and Random Forest can be interpreted by LIME by making small, systematic changes to the data and observing how those changes affect the output of the model. This process is crucial for understanding which parts of the input data are most influential in the model’s predictions. Another process is Mean Decrease Accuracy (MDA), a metric used to evaluate the importance of individual features (variables) in classification tasks within Random Forest models. It quantifies how much the model’s accuracy suffers when a particular feature is randomly scrambled, with a higher MDA value indicating significant contributions to accurate predictions across all instances. Researchers and practitioners rely on this metric to prioritize relevant variables and enhance model interpretability [21].

## **1.2 STATEMENT OF THE PROBLEM**

The team intends to tackle these specific problems as follows:  
**Main Problem**

1. Detecting bot accounts on X (Twitter) is important for maintaining platform integrity. However, existing methods lack transparency, which hinders the users’ understanding of how decisions are made.

**Specific Problem**

1. Detection models on X including CNN are called “black boxes”, providing high accuracy but lacking interpretability in how or why they arrive at their predictions.
2. Selecting which features that contribute the most to the bot detection model’s decision can be challenging, especially when using ensemble methods.

## **1.3 RESEARCH QUESTIONS**

1. How does RNN and Random Forest ensemble model compare to the accuracy and interpretability of CNN?
2. Which factor contributes the most in identifying X’s (Twitter) bot accounts?

## **RESEARCH OBJECTIVES**

* + 1. **Main Objective:** Developing an X (Twitter) bot detection method using an ensemble learning approach that combines two specific models specifically RNN and Random Forest model.

1. To measure the accuracy and interpretability of using RNN and Random Forest models with the ensemble method in detecting X (Twitter) bots.
2. To evaluate which feature of the ensemble model (RNN and Random Forest) contribute the most to its decision-making using LIME (Local Interpretable Model-agnostic Explanations) and MDA (Mean Decrease Accuracy).

## **SCOPE AND LIMITATIONS**

This research focuses on enhancing bot detection on "X" (formerly Twitter) through an ensemble learning approach that combines Recurrent Neural Networks (RNN) and Random Forest models. The study evaluates the accuracy and interpretability of this integrated model, addressing Objective 1. Interpretability techniques such as Local Interpretable Model-Agnostic Explanations (LIME) and Mean Decrease Accuracy (MDA) are employed to analyze feature importance, fulfilling Objective 2 by identifying the most critical factors influencing the model's decisions.

This study is subject to several limitations that may influence its findings and broader applicability. Firstly, the reliance on a pre-existing Twitter dataset may limit the ability to capture recent bot behaviors or emerging features, potentially impacting the accuracy measuring the accuracy and interpretability. The pre-existing dataset may not represent evolving tactics used by bots, which could reduce the relevance of the model’s performance in real-world applications.

Secondly, the findings related to accuracy and feature importance may not generalize well to other datasets or social media platforms. As the study is dataset-specific, the conclusions drawn regarding bot detection performance and interpretability might not apply across different contexts. Additionally, the research focuses solely on an ensemble method combining RNN and Random Forest, excluding comparisons with alternative machine learning or deep learning techniques.

Lastly the methods employed for feature contribution analysis, such as Local Interpretable Model-Agnostic Explanations (LIME) and Mean Decrease Accuracy (MDA), come with inherent biases and limitations. These methods may not fully explain all model decisions, which could affect the reliability of insights related to evaluating which feature of the ensemble model contributes the most in its decision-making.

## **SIGNIFICANCE OF THE STUDY**

**X (Twitter) Users:** Enhanced bot detection indirectly benefits X (Twitter) users by reducing spam, misinformation, and harmful content. With fewer bots, users experience more genuine interactions, improved content recommendations, and a safer online environment. For instance, effective bot detection leads to a decrease in automated spam messages and fake news links. With interpretability analysis, the bot detection system’s decision can be evaluated by users to see if they meet their expectations. Interpretability assists in the detection of false positives, or legitimate accounts reported as bots, and false negatives, or real bots overlooked which gives users the ability to verify or to contest these categories.

**Brands and Businesses:** Effectively identifying and filtering out automated bot accounts, businesses and brands can significantly enhance their interactions with real customers. They could customize their responses based on genuine user needs, creating a more personalized experience. Tailoring not only improves user satisfaction but also fosters trust. Additionally, building authentic relationships with real customers is crucial for brand success. Less bot interactions will contribute to a positive brand image, reinforcing the idea that the brand values its customers and prioritizes their needs.

**Future Researchers:** Researchers specializing in bot detection and social media analysis can leverage this work as one of their foundations. The novel ensemble method, combining RNNs and random forests, presents an innovative approach. Future researchers can study its effectiveness, fine-tune hyperparameters, and adapt it to social media platforms beyond X (Twitter). Investigations on how ensemble approach performs in detecting bots on platforms like Facebook or Instagram could also be possible.

**CHAPTER 2**

# **REVIEW OF RELATED LITERATURE**

The study will focus on adding an interpretation model with X’s (Twitter) Bot Detection which will use the RNN and RF model combining it using the Stacking Method from Ensembling. To know more about these topics, the review of related literature will have three sections: Theoretical Background, Technical Background, and Bot Detection.

It begins by discussing the definition of Twitter bots and distinguishing between helpful and harmful ones. It then explores Recurrent Neural Networks (RNs), which excel at processing sequential data and are valuable for tasks like text analysis and bot detection. The review talks about the Random Forest algorithm, known for its accuracy and robustness. By constructing multiple decision trees from random data samples, Random Forests enhance prediction accuracy and reduce errors, making them effective for identifying bots. Ensemble methods are also discussed, which combine various learning algorithms, are also discussed. These methods enhance predictive performance and model interpretability, aiding in understanding decision-making processes in bot detection. As the study wants to improve X's (Twitter) bot detection in its interpretability, techniques like Mean Decrease Accuracy (MDA) and Local Interpretable Model-Agnostic Explanations (LIME) are mentioned to ensure reliable and understandable model predictions.

In the technical background, the review covers essential tools and libraries for implementing bot detection models, including Visual Studio Code, Python, Pandas, and sklearn. These tools facilitate the development of effective detection systems. As of the last section, the review discusses related studies about bot detections, though not all necessarily about X (Twitter). The studies reviewed revolved around deep learning approaches with traditional machine learning methods, highlighting deep learning’s superior performance in handling complex data patterns, resulting in more accurate bot detection.

## **2.1 THEORETICAL BACKGROUND**

**2.1.1 What are Twitter bots?**

“X” (formerly known as Twitter) bots are automated accounts that use software to interact in the platform the same way as human users do [21]. These actions can include tweeting, retweeting, liking, following other accounts, and even sending direct messages. There are two types of bots on ‘X’, helpful and harmful bots. Helpful bots can be used to spread important information, such as weather updates during typhoons, while harmful bots are malicious in that they spread misinformation and can manipulate conversations. One of the examples of harmful bots is Spambots. Spambot shares spam links, such as unwanted advertisements to a large audience [22].

2.1.1.1 How do Twitter Bots Work?

Twitter bots are built and run utilizing a combination of programming, APIs, algorithms, and ethical considerations to automate tasks and enhance the user experiences of the platform. Developers often utilize programming languages such as Python, JavaScript, or Ruby to write scripts that describe a Twitter bot’s behavior. These scripts automate actions like tweeting, retweeting, liking, and following specific accounts. The “X” (Twitter) API is a crucial aspect of this process because it enables endpoints for bots to communicate with the platform. Bots can use API to submit tweets, search hashtags, and follow users, but they must be authenticated using API keys and tokens to ensure secure and allowed access [68]. Algorithms strongly determine bot behavior. They allow bots to make judgements, such as deciding when to tweet based on interaction patterns or analyzing data to track trends, hashtags, or specific phrases. These characteristics enable bots to take focus and strategic actions, boosting their efficacy in attaining their objectives. For example, a bot built to offer news updates could employ an algorithm to decide which headlines to tweet and when. Twitter bots can automate a range of operations, including scheduling tweets, retweeting content that satisfies established criteria, like relevant tweets, and following accounts with specific interests or profiles. Bots also continuously check Twitter for updates or occurrences, such as a weather bot that scans for notifications and notifies users right away. Regular maintenance is also required to guarantee that bots continue to function and comply with Twitter's regulations. Ethical considerations are crucial while developing and deploying Twitter bots. While good bots can provide beneficial services like news distribution, customer care, and instructional content, bad bots can transmit false information, spam, or control public conversation. Developers must adhere to ethical principles and Twitter's policies to ensure that their bots contribute constructively to the online ecosystem.

Creating and running a Twitter bot is a systematic process that incorporates API access, programming, and continuous maintenance to enable automation and assure compliance with platform restrictions. The initial step is to register the bot with Twitter and receive API keys and tokens. These credentials are required to authenticate the bot and grant it access to Twitter's API, which acts as the interface for all of its actions on the site [69]. The bot is written in languages such as Python, JavaScript, and Ruby. The programming phase entails building code that defines the bot's unique actions, such as tweeting, following accounts, and liking material. These instructions are routed to Twitter API endpoints, which are predefined routes for performing specific functions. For example, the POST statuses/update endpoint allows the bot to send tweets, but the POST friendships/create, and POST favorites/create endpoints allow it to follow accounts and like tweets. These endpoints serve as the backbone for the bot's interactions with the Twitter ecosystem. Once the bot's code and logic are complete, it can run automatically according to established rules or schedules. For example, the bot may post tweets every hour or follow users that use specific hashtags or subjects. This level of automation allows the bot to function smoothly and reliably without continual manual intervention. Monitoring and maintenance are critical to the bot's long-term performance. Bots are programmed to continuously watch Twitter for certain events or material, ensuring that they respond effectively and remain focused on their intended purpose. Developers are responsible for upgrading the bot's code and configurations to accommodate changes in Twitter's API or regulations. Regular maintenance also ensures that the bot functions responsibly and in accordance with Twitter's policies, preventing any abuse or disruptions to the platform.

2.1.1.2 What are the types of Bot Behavior?

One оf the most prominent bеhaviors еxhibitеd by Тwitter bоts is twееting, a funсtiоn they perfоrm autоmatically based оn prеdеfinеd algorithms or triggеrs. Тhese bоts are dеsignеd tо generate аnd post tweets indeрendently, serving variоus purposes dеpеnding on their programming [74]. An example: news bоts frequently utilizе this feature tо share uрdates, breaking stоries, or rеal-timе alerts with their аudiences, ensuring timely disseminatiоn оf imрortant informatiоn. Similarly, promotiоnal bоts rely on autоmatic twееting tо markеt рroducts, services, or еvеnts. By posting cоntent cоnsistently, these bоts help brаnds maintain an active оnline рresence аnd reach their target audience effectively. Вots might also use twееting for engagement, such as posting questions, polls, or engaging hashtags, to encourage interaction. Тhe аbility tо autоmate twееting makes bоts invaluable tоols for maintaining a dynamic рresence оn Тwitter without requiring constant human оversight, making thеm widely used in fiеlds like journаlism, markеting, аnd publiс communicatiоn.

Rеtwееting is аnothеr сritiсаl behаviоr emрloyed by Twittеr bоts, оften used tо mаgnify thе reаch оf messаges оr рromote speсifiс nаrrаtivеs. Тhese bоts аre рrogrаmmed tо identify аnd retweet cоntent thаt аligns with thеir рurрose, whethеr it’s а politicаl messаge, а mаrketing саmpаign, оr аn infоrmаtionаl рost [74]. Politicаl bоts, fоr instаnce, usе rеtwееting tо аmрlify messаges frоm cаndidаtes оr саmpаigns, crеаting thе impression оf widеsprеаd suppоrt. Similаrly, mаrketing bоts retweet brаnd cоntent, helрing tо expаnd visibility аnd аttrаct new аudiences. Retweet bоts cаn аlso рlаy а rоle in rаising аwаreness fоr pаrtiсulаr cаusеs оr events by еnsuring relаted cоntent is reрeаtedly shаred аcross timelines. By increаsing thе frequency оf speсifiс twееts on thе plаtfоrm, rеtwееting bоts crеаtе а snowbаll effect, encоurаging оrgаnic usеrs tо engаge with оr shаre thе sаme cоntent. Тhis behаviоr mаkes rеtwееting а powerful tоol fоr influenсing trends аnd enhаncing thе visibility оf tаrgeted messаges.

Following is а strаtegic behаviоr utilized by Тwitter bots to build connections or engаge with specific communities. Вots progrаmmed to follow usеrs oftеn do so bаsеd on cаrefully defined critеriа, such аs relevаnt hаshtаgs, usеr interests, or profile аttributes [74]. For instаnce, а bot focusеd on thе teсh industry might аutomаticаlly follow usеrs who frequently tweet аbout coding, stаrtups, or innovаtive teсhnologies. Тhis behаviоr is pаrticulаrly usеful for growing аn аccount’s fоllоwer bаse rаpidly or tаrgeting nichе аudiences. Вy following usеrs with shаred interests, bots creаte opportunities for reciprocаl engаgement, increаsing thеir visibility within relevаnt circles. Orgаnizаtions аnd mаrketers oftеn usе bots to engаge with communities thаt аlign with thеir оbjectives, such аs prоmоting events, shаring industry insights, or estаblishing а presenсe within а pаrticulаr demogrаphic. While this behаviоr cаn fоster genuine connections when usеd responsibly, it cаn аlso be exрloited to аrtificiаlly inflаte fоllоwer counts or tаrget usеrs for mаrketing рurрoses.

Тhe аbility tо likе twееts аutоmаticаlly is аnothеr impаctful behаvior dеmonstrаtеd by Тwitter bоts, оften usеd tо сreаte thе illusion of engаgement or suppоrt fоr spеcific cоntent. Bots thаt аutоmаticаlly likе рosts аre cоmmоnly progrаmmed tо identify cоntent bаsеd on keywords, hаshtаgs, or usеr profiles [74]. Аn exаmple: а bot suppоrting а pаrticulаr politicаl cаusе might systemаticаlly likе twееts thаt mention certаin slogаns or tоpiсs. Similаrly, mаrketing bоts usе this behаvior tо likе twееts from рotentiаl custоmers, fostеring а sense of connеction or аpprovаl. Sociаl mediа mаnipulаtion bоts frequently leverаge аutоmаtic liking tо swаy public opinion or influence trends, mаking spеcific рosts аppeаr more populаr thаn thеy gеnuinеly аre. By increаsing thе number of likеs on tаrgeted twееts, bоts cоntribute tо thе рerceрtion of credibility or widespreаd аgreement. While this behаvior cаn be usеful fоr prоmоting cаmpаigns or cаusеs, it аlso rаises ethicаl concеrns аbout аuthеnticity аnd thе рotentiаl tо misleаd usеrs.

2.1.1.3 What are the two types of Twitter bots?

Gооd bоts аre autоmatеd prоgrams dеsignеd tо perform cоnstructive аnd benefiсial tasks thаt enhаnce usеr exрeriences, simplify procеssеs, аnd dеlivеr valuable serviсes. Unlike maliciоus bоts, whiсh aim tо deсeive, disruрt, or eхploit, gооd bоts oрerate ethicаlly аnd cоntribute pоsitively tо thе digitаl lаndscape. Тhey аre purposе-drivеn, сreated with speсifiс goals suсh as providing informаtiоn, autоmating repetitive tasks, or fostering engаgement оn various platforms [75]. А hallmark оf gооd bоts is thеir adherence tо ethical stаndards, avоiding deceptive practices or harm tо usеrs while fоcusing оn facilitating аssistаnce аnd improving functiоnality. Examples оf gооd bоts include chatbоts, whiсh simulаte human cоnversatiоns tо prоvide custоmer suppоrt, answer questiоns, аnd аssist with purchаses оn websites аnd messаging platforms. Web crаwlers, anothеr tyрe оf gооd bоt, helр search engines like Googlе index web рages tо imprоve search results аnd prоvide usеrs with relevаnt informаtiоn quickly. Social media bоts, whеn used legitimately, shаre uрdates from organizatiоns or рromote positivе messаges around meaningful causes. Mоnitоring bоts track website рerformance, alerting administratоrs tо issues suсh as downtime or security risks tо maintain seamless functiоnality. Plus, utility bоts dеlivеr еssеntial serviсes like weathеr uрdates, news alerts, or rеmindеrs through messаging aррs. Togethеr, thеse bоts demоnstrate hоw autоmatiоn can enhаnce digitаl interactiоns whеn аligned with ethical principlеs.

Mаlicious bоts are frequently deрloyed tо еxеcutе harmful activities that disrupt systems, deсeive usеrs, оr exрloit online platfоrms. A significant eхample is thеir usе in Distributed Denial оf Service (DDoS) attacks, where bоts overwhelm servers with еxcеssivе traffic, rendering thеm inaccеssiblе tо legitimate usеrs [75]. These attacks оften rely on bоtnеts, netwоrks оf сompromised devices wоrking in unison tо carry оut large-sсale disruptions. Оthеr cоmmоn maliciоus bоt type is thе spambоt, which inundatеs social mеdia platfоrms аnd fоrums with unsolicited messаges, рromotional сontent, оr рhishing attеmpts, cluttering digital spacеs аnd exрosing usеrs tо security risks. Credential stuffing bоts exрloit stоlen login credentials оbtained frоm data brеachеs tо gain unauthоrized access tо usеr aссounts acrоss vаrious wеbsitеs, furthеr endаngering persоnal аnd financial infоrmation. Similаrly, maliciоus wеb scraрers eхtract vаst amounts оf data frоm wеbsitеs withоut рermission, violating tеrms оf service аnd оften resulting in data thеft оr misusе. Ad fraud bоts simulatе human behaviоr tо gеnеratе fаke advеrtisеmеnt clicks, cаusing financial losses fоr advеrtisеrs аnd distоrting analytics that businesses rely on fоr dеcision-making. Social mеdia manipulatiоn bоts рlay a partiсularly insidious rolе by spreading misinfоrmation оr inflаting engаgement metrics like likеs аnd follows, influenсing рublic oрinion аnd skewing disсussions. Togethеr, thеse bоts illustrаte thе pervаsive risks оf maliciоus autоmation in thе digital lаndscape.

**2.1.2 What is Machine Learning in Bot Detection?**

Machine learning (ML) is a subset of artificial intelligence (AI) that is capable of learning from data, identifies patterns, and makes decisions with minimal human intervention wherein it utilizes algorithms and statistical models to analyze and draw inferences from patterns [60]. The process of ML involves training a specific model on a dataset, allowing it to make predictions and decisions. During the process, its performance would improve over time as it learns from new data. The key component of its process includes data preparation, feature selection, model selection, training, testing and lastly, optimizing [60].

Applications of ML have been implemented worldwide in specific departments such as finance where it detects fraud where its goal is to be able to distinguish the original and the fake one, the same thing can be applied in detecting bot accounts since ML has also been influenced in the internet this includes specific areas such as search engines, recommendations and detecting spams or with malicious content [61]. ML improves bot detection by, as stated earlier, analyzing user behavior patterns this can be in a form of how frequent a specific account posts in a day and such wherein this would help distinguish accounts whether they are bot or human as long as the key components are included in its training such as the data preparation, feature selection, and model selection.

2.1.2.1What are the types of Learning in ML?

There are two types of Machine Learning (ML), supervised and unsupervised learning. Supervised learning involves training a specific model on labeled data where the correct output of it is already known [62] for instance a specific model is trained with a dataset that has a labeled data on it that identifies whether that specific row is a bot or human. Therefore, supervised learning is commonly used for classification and regression tasks. Real-life applications of supervised learning would be weather forecasting, just like in news where of course the weather man reports the coming rain or storm and helps us give a bit of understanding on how it will move the coming days, this is because a specific model is trained with historical weather data and outputs of it which is of course labeled and so it predicts future weather [63].

While supervised learning uses labeled data, unsupervised learning is the opposite of it meaning to say that it uses unlabeled data. Wherein a specific model identifies patterns or structures without any predefined models, unsupervised learning is typically used for clustering or anomaly detection [62]. Another brief understanding for an unsupervised learning would be the use of recommendation engines, where it can be used to discover patterns and relationships of the user’s preferences without any labeled data just like the use of TikTok or even YouTube [63].

2.1.2.2 What is the role of Features in Machine Learning?

Features are the key elements or attributes of a dataset that allows machine-learning algorithms to be able to understand the data patterns from the data set that they are given during the training the model would be able to learn various combinations of these specific features and how they will relate to the target variable [64] whereas selecting and engineering relevant features would help the model learn more efficiently and making accurate predictions. However, it is also important to take note that poorly chosen features can lead to overfitting, underfitting or biased results so understanding the roles of each feature used is crucial in building an efficient model [64].

**2.1.3 What and how does ensemble method work?**

Ensemble learning combines multiple individual models to improve generalization performance of the model which is done by letting the trained model perform on unseen data [25]. Deep ensemble models integrate deep learning architectures, known for their feature representation capabilities, with ensemble learning strategies. These models have been applied to many fields such as speech recognition, forecasting, healthcare and image classification which demonstrates their effectiveness in different fields.

Ensemble learning works by combining predictions from multiple models making it more flexible and capable of capturing complex, non-linear interaction. Ensemble models, like using Random Forest and Boosted Regression Trees, showed lower prediction error than when it was used individually. These ensemble approaches capitalize on the strengths of different algorithms to enhance overall predictive performance [26].

**2.1.4 Model interpretation**

An ML model interpretation is where a specific ML model obtains a prediction and uses those predictions and eventual insights to solve a range of problems where it will help answer specific questions such as “How trustworthy are these predictions?” Or “Are they reliable enough to make big decisions?” In other words, model interpretation redirects the focus from “what was the conclusion?” to “why was this conclusion reached?” which gives the researchers an understanding of what exactly drives the model to classify a data point correctly or incorrectly [72].

The importance of having a model interpretation is that it helps a specific model in these few aspects; fairness, reliability, causality and lastly trust. There are different ways to interpret an ML model primarily these methods can be categorized as; Model-specific meaning specific to certain models as they depend on the inner machinery of a model to make certain conclusions. Model-agnostic, this can be used in any model, they’re generally applied post-training and usually work by analyzing the relationship between feature input-output pairs and don’t have access to the model’s internal mechanics such as weights or assumptions. Local, where its scope merely covers only an individual prediction, capturing the reasons behind only the specified predictions. Global Scope, just by looking at the word ‘Globe’ it extends beyond an individual data point and covers the model’s general behavior [71].

## **2.2 TECHNICAL BACKGROUND**

**2.2.1 What are Recurrent Neural Networks?**

Like traditional neural networks such as feedforward neural network and convolutional neural networks (CNNs), Recurrent Neural Networks (RNNs) are a type of neural network architecture used to detect patterns in a data sequence, they are distinguished by their “memory” as they take information from prior inputs to influence the current input and output [23] [65]. The data detected can be genomes, handwriting, text, or numerical time series which are often used in sensor. Additionally, RNNs can be applied to images by breaking them down into patches and by processing them as sequences from the patches. At a broader level, RNNs are used in language modeling and text generation, speech recognition, generating image descriptions, and video tagging.

RNN has a concept of “memory” which remembers all information about what has been calculated in other words, it allows it to retain information from previous inputs and incorporates that knowledge as it processes subsequent data through feedback loops with the network where the output from a previous step is fed back in and would eventually influence the next step. It basically learns and learns more as it passes more through each row of data [65]. Another distinguishing characteristic of recurrent networks is that they share parameters across each layer of the network. This allows the model to treat all time steps equivalently, learning patterns that apply across time [65].

Despite it being a powerful model it has issues like vanishing gradients which makes it difficult for them to retain information over long sequences and this is where LSTM, a variant, is introduced [65].

**2.2.2 What is Long Short-Term Memory (LSTM) and Bidirectional LSTM (Bi-LSTM)?**

Long Short-Term Memory (LSTM) is a recurrent neural network (RNN) architecture widely used in Deep Learning that excels at capturing long-term dependencies making it ideal for sequence prediction tasks and addresses the issue of the RNN specifically the vanishing gradients [66]. It uses memory cells to store information over time, helping the model learn temporal patterns [66].

LSTM consists of cells with gates –input, forget and output– that control the flow of information [66]. Input gate decides what new information to add to the cell state, forget gate determines what information to discard from the cell state and lastly is output gate where it controls what part of the cell state is output as the network’s result [66]. Therefore, LSTMs are especially suited for tasks involving sequential data like time-series predictions or by basing through user behavior patterns this could involve, in the context of X (twitter), capture and classify tweets in distinguishing human and bot accounts based on their text and metadata features. LSTM has also been applied in our everyday lives for example would be the recommendation system, LSTM have been used for recommendation tasks such as movies, music and books, it learns through patterns in user behavior and use them to make personalized recommendations [79].

Bidirectional LSTM (Bi-LSTM) is also an RNN used primarily on natural language processing however unlike the standard LSTM, the input flows in both directions and it’s capable of utilizing information from both sides which makes it a powerful tool for modeling the sequential dependencies between words and phrases in both directions of the sequence [80]. Bi-LSTM adds one more LSTM layer where it reverses the direction of information flow meaning that the input sequence flows backward in the additional LSTM layer then combines the outputs from both LSTM layers in several ways, such as average, sum, multiplication, or concatenation [80]. Bi-LSTM will have a different output for every component of the sequence which leads to it being beneficial in some NLP tasks such as sentence classification, translation and entity recognition [80].   
The key differences between LSTM and Bi-LSTM is that Bi-LSTM is an extension of LSTM, since Bi-LSTM uses two LSTM layer wherein it contains information in both past and future data however despite Bi-LSTM being better, its complexity is more computationally intensive due to the fact of it having two LSTM layers meaning to say that it takes longer periods of time for it to train [79][80].

However not every model is perfect, there are also some drawbacks, this includes the LSTM and Bi-LSTM. Both are computationally expensive especially Bi-LSTM, meaning, it can be slower leading to long training times and may require more resources especially when you have a small amount of data in your dataset since it would lead to overfitting, which another one of its drawbacks, they are susceptible to it especially when training data is insufficient just like any other models insufficient data can negatively affect the model’s performance [70][80]. Because of the complexity and longer periods of time for it to train GRU was introduced which is also another variant of an RNN.

2.2.2.1 Why use Bi-LSTM for Bot detection?

LSTM, a variant of RNN where it was able to address the vanishing gradients issue of the RNN where it is designed to capture dependencies in sequential data is suitable for X (twitter) bot detection because of its ability giving temporal sequence analysis where bots exhibit distinct patterns of behavior over time compared to a human, it analyzes the amount of tweets posted or how frequent are the hashtags [66][70]. Although there exists another variant which is GRU but in the context of bot detection LSTM would still be a better choice despite it being computationally intensive, because of its ability to capture long-term dependencies in user behavior more effectively than GRU as it is crucial since bot may exhibit suspicious behavioral patterns over a longer timeline although GRU is simpler and faster it might not be as effective in identifying longer-term dependencies [66][70][76]. Additionally, there is also CNN but between CNN and LSTM, CNN holds the higher ground in analyzing structured data and spatial hierarchies such as images, they are not inherently designed for sequential data compared to LSTM which is much more suitable since the model is designed to process sequential data [70][71]. Nevertheless, compared to LSTM, BiLSTM outweighs the benefits LSTM does since not only BiLSTM uses past information or context of a data it also uses the future context of it since BiLSTM uses two layers of LSTM providing a more holistic understanding of the bot’s behavior since it processes data in both directions, from start to end and end to start, basically it goes back and forth and it be able to, of course, distinguish human and bot accounts better than LSTM [66][70][79][80].

**2.2.3 What are Gated Recurrent Units (GRUs)?**

Gated Recurrent Unit (GRU) another variant of RNN like the LSTM, it is designed to model sequential data by allowing information to be selectively remembered or forgotten over time, however compared to LSTM, GRU has a simpler architecture than LSTM with fewer parameters which makes it easier to train and more computationally efficient [76]. The main difference between the two is the way they handle the memory cell state wherein LSTM the memory cell state is maintained separately from the hidden state and is updated using three gates- the input gate, output gate and forget gate however in GRU it is replaced with a “candidate activation vector” which is updated using two gates- the reset gate, where it determines how much of the previous hidden state to forget and update gate, how much of the candidate activation vector to incorporate into the new hidden state [76]. The difference between the LSTM and GRU is that LSTM is more complex and GRU is simpler therefore making it faster in terms of training and having a lower memory consumption.

**2.2.4 What are Convolutional Neural Networks (CNNs)?**

Convolutional Neural Networks (CNN) are useful for finding patterns in images to recognize objects, classes and categories, they can also be implied with bot detection as it extracts text and analyzes the structure of user interactions or detecting patterns in textual content since the core strength of the CNN extracts features automatically from raw data, making them highly efficient for tasks involving structured or visual patterns [71].

**2.2.5 What is Random Forest?**

The Random Forest algorithm is a versatile and powerful machine learning algorithm. It constructs multiple decision trees during training, each using a random subset of the dataset and measuring a random subset of features [24]. To be more detailed about it, Random Forest combines the prediction of multiple decision trees to improve accuracy, reduce overfitting and increase robustness and is commonly used for both classification and regression tasks [77]. It is made up of a collection of decision trees where each tree in the ensemble is comprised of data sample drawn from a training set with replacement, called the bootstrap sample [77]. The key benefits of this model are the reduced risk of overfitting since the decision trees tend to tightly fit all the samples within training data however when there’s a robust number of decision trees in a random forest the classifier won’t overfit the model since the averaging of uncorrelated trees lowers the overall variance and prediction error [77]. Random forest is also flexible since it can handle both regression and classification tasks with a high degree of accuracy and it is also easy to determine the feature importance which is it’s interpretability giving insights of its variable importance or contribution to the model not only does it have feature importance it is also able to provide MDA [24][77]. However, despite it having a good performance and an interpretability feature specifically the feature importance, it also has drawbacks, one of it would be in the case of hyperparameter tuning, random forest has several hyperparameters (e.g., number of trees, maximum dept) that can affect performance, and by finding the right combination of the hyperparameter can be challenging and time-consuming especially when working with large datasets [28][73].

2.2.5.1 Why use Random Forest for Bot Detection?

Random Forest is also an ideal choice for Bot Detection, In Random Forest, multiple decision trees are created and merged to obtain a more accurate prediction. Random forest is used in bot detection as it reduces overfitting and generally improves classification accuracy. Additionally, Random Forest is robust to noise and outliers for instance in social media where data is often noisy because of its diverse user behavioral patterns, it makes it a reliable model for real world bot detection where data can be messy [77]. Compared to LSTM and GRU it does not require large datasets, and it is able to train effectively and perform well on smaller datasets making it an ideal model to train with small datasets especially since labeled bot detection datasets are limited or costly to obtain [77]. It also has Feature Importance and MDA where it is valuable for understanding what makes an account a bot and for selecting the most relevant features for prediction [24][77]. To add on it, Random Forest trains faster and are less computationally expensive and like LSTM it can effectively classify bot or human accounts by looking at behavioral patterns such as the diversity of hashtags used or how frequent does this account tweet. [24][77].

**2.2.6 MDA (Mean Decrease Accuracy)**

Mean Decrease Accuracy (MDA) is a metric for evaluating the importance of features in a prediction model, especially in ensemble methods such as Random Forests [27]. It measures how much each feature contributes to the accuracy of the model by determining the difference in accuracy when permuting the values of the features. The basic idea is to randomly permute the individual feature values and reduce the accuracy of the resulting model. The intuition behind this is that if one feature is important, changing that feature will significantly reduce the extent to which predictions can be made by the model. Additionally, MDA aids in selecting the most relevant features for training which reduces the risk of overfitting and improving generalization to new data [27]. MDA is essential in bot detection since it helps identify which features are most useful in distinguishing between real users and bots.

**2.2.7 LIME (Local Interpretable Model-Agnostic Explanations)**

Another model interpretation tool would be LIME. LIME is model agnostic, meaning that it can be applied to any machine learning model [29]. It involves attempts to understand the model by perturbing the input of data samples and understanding how the predictions change, providing local model interpretability and modifies a single data sample by tweaking the feature values and observes the resulting impact on the output which answers the question, “why was this prediction made or which variables caused the prediction?” [28]. With the aid of LIME not only does it aid in model interpretability but also enhances user engagement and trust [28]. By employing LIME, researchers would be able to not only have an effective model but also transparent which is important in maintaining the integrity of automated systems in social media environments.

2.2.7.1 Why use LIME for Model Interpretation?

LIME comes out on top for bot detection interpretation compared with SHAP because of its ability to generate local explanation for individual predictions where this feature is beneficial in bot detection, where understanding the rationale behind a specific classification is crucial whether it is a bot or human. LIME is also computationally efficient, making it suitable for real-time applications on large datasets such as those from social media while SHAP although powerful with its interpretation, it is more computationally intensive and focused on global explanations compared to LIME which focuses on a single instance [28][73].

**2.2.8 What are the Evaluation Metrics to be used for Bot Detection Model?**

Measuring performance is crucial in the large field of machine learning, where algorithms and models digest data to reveal hidden insights [78]. An ML’s performance can be judged using performance measures where it provides an objective evaluation of the model’s capacity for precise classification or prediction by quantifying factors including accuracy, precision, recall and F1 score ensuring that the model complies with the specified standards and requirements [78].

Accuracy is the proportion of forecasts that were accurate to all the input samples, basically the ration of correctly predicted instances to the total number of predictions, precision is the proportion of true positives to the total of both true and false positives. Recall is the proportion of true positives to the total of true positives and false negatives, F1 Score is the harmonic mean of precision and recall providing a single metric that balances both and lastly the Confusion Matrix where it provides a complete picture of how well the model is performing on both classes this involves the true positives, true negatives, false negatives and false positives [78]. By having this performance metrics in a bot detection model, this helps assess how effectively the model distinguishes between bot and human accounts and by being able to analyze it researchers would be able to have insights and can guide improvements and ensure that the system is reliable in a real-world application.

**2.2.9 Visual Studio Code**

Visual Studio Code, also known as VS Code, is a free code editor designed for creating and debugging modern web and cloud applications [29]. It is available on various platforms such as Linux, macOS and Windows. VS Code supports almost all programming languages such as Python, JavaScript and PHP. VS Code is based on open source and runs everywhere, which makes it a popular choice due to its versatility.

**2.2.10 What is Python?**

Python is a high-level programming language known for its simplicity and readability. Python was developed by Guido van Rossum and published in 1991 [30]. Python is widely used in web development, scientific computing, data analysis and artificial intelligence. Due to its elegance and simplicity, it is particularly popular with beginners and contributes to it being one of the most widely used programming languages in the world. Python is very flexible and has numerous libraries and frameworks that make it invaluable for programmers.

**2.2.11 What are the libraries used for Model Development?**

2.2.11.1 Pandas

Pandas, also known as Python Data Analysis, is a python library used for working with big sets of data, where it involves analyzing, cleaning, exploring and manipulating data [31]. This python library known as Pandas allows us to analyze big data and make conclusions about it based on statistical theories, as we all know relevant data is very important in data science, with panda it can clean messy data sets making them readable and relevant especially when you're training a model and want to get rid of duplicates or missing values that will eventually affect the model’s training.

2.2.11.2 sklearn

An open-source machine learning and data modeling library for Python, one of the most useful libraries for machine learning in python as it contains a lot of efficient tools that can be used for machine learning and statistical modeling which includes classification, regression, clustering and dimensionality reduction. It also includes algorithms such as random forests, k-means and DBSCAN and is designed to interoperate with the Python libraries such as NumPy and SciPy [32][33].

2.2.11.3 matplotlib

It is a Python statement that is popular for plotting library used for creating a visualization in python. It provides a wide range of functions. This includes different types of plots, specifically line, scatter, bar, histograms and even more. Due to its versatility users may also customize their plots including its titles, legends and labels and may be displayed on screen itself or be saved as various formats like PNG or PDF [48].

2.2.11.4 pickle

It is a Python module that is used for serializing and de-serializing Python objects [52]. This implies that it can translate byte streams into Python objects such as lists and dictionaries and vice versa. Pickle enables us to store and retrieve data in an exact manner from files.

2.2.11.5 numpy

It is a powerful Python library for numerical computations. It offers efficient multidimensional arrays and an extensive range of mathematical functions that can be applied to these arrays [53]. It provides optimized functions for various numerical operations, including Fourier transforms, linear algebra, and mathematical computations. It can easily integrate with Pandas, Matplotlib, SciPy, and other scientific computing libraries.

2.2.11.6 tensorflow

It is an open-source machine learning framework developed by Google. It offers an adaptable framework for creating and honing different machine learning models, such as neural networks [54]. TensorFlow supports a broad range of use cases, from research to production, by providing flexibility in the development and deployment of models.

2.2.11.7 keras

It is a high-level API written in Python that provides an interface for artificial neural networks. Deep learning model construction and training is made simple by its modular and user-friendly design [55]. It offers a great deal of flexibility, so the user can customize their models to meet their unique needs.

2.2.11.8 Grid Search

Grid Search is a conventional algorithm used in machine learning for hyperparameter tuning by methodically evaluates every combination of the specified hyperparameters to identify the best-performing model [58]. Grid search would help benefit the team as it performs model selection and hyperparameter tuning simultaneously and be able to identify the best model. To add more on the best model, it means that it automates and does exhaustive search over the parameters and gets the most accurate predictions despite it being computationally expensive.

**2.3 RELATED SYSTEMS**

**2.3.1 Fedica**

Fedica, one of the existing ‘X’ (Twitter) bot detection tools that is designed to audit and analyze the followers of a specific user in ‘X’ by identifying and categorizing the followers based on their quality distinguishing them with labels between fake followers, low-quality followers, mid-quality followers, and high-quality followers which allows the user to get rid of the bots by basing on their labeled quality [45].

**2.3.2 Botometer**

Botometer, is another one of the existing ‘X’ bot detection tools, wherein it uses a machine learning classifier that analyzes which accounts are more bot-like and which are more human-like by rating them from 1 to 5. Assessing their account’s tweets by week, the ratio of retweets and more and labels them what type of bot they are whether it is a fake follower, a spammer or an echo-chamber [46]. Botometer focuses more on tracking suspicious bot-like activity, and it does not analyze the interaction between an account’s followers and the account meaning to say it is just a bot score for an account’s followers [46].

**2.3.3 Hoaxy**

Hoaxy, just like the other two, it is also a bot checker that visualizes how information spreads on X-how bots are swarming and to see “bot scores” for the accounts using specific phrase like “monkey pox” or “polling place” [46]. The scores are rated on a color scale ranging from red which is considered as bot-like and blue as human-like as well as a bot score ranging from 1 to 5, with 1 being the most human and 5 as most bot [46]. The app mainly focuses on tracking how information is spread but it helps researchers and journalists track the spread of misinformation and see how much information is spread by bots [46].

**2.3.4 Bot Sentinel**

Bot Sentinel is a system dedicated to detecting and monitoring any fake Twitter accounts if they were provided on the website by classifying them on a scale from Normal to Problematic [46]. The profiles, which are provided by users, will be given a percentage score and what other accounts they are connected to. The system also provides users with a functionality of checking entire Twitter profiles or specific tweets [46]. Bot Sentinel also helps journalists and people who are being targeted for harassment by offering free tools for bulk and automatically blocking problematic accounts [46].

The related tools are helpful in detecting X’s (Twitter) bot accounts especially having a high accuracy on detecting, labeling and having multiple features including sorting out or organizing followers, bot scoring and color scaling. All mentioned automatic bot detection tools have problems with false positives and false negatives depending on the threshold of each tool [47]. Since all the tools mentioned lacks transparency, and understanding of what features or behavior that made the accounts classify as bots, evaluating or contesting the false positives or false negatives of the bot detection tool is a hard task without evidence.

**2.4 RELATED STUDIES**

**2.4.1 Deep Learning Methods: A Systematic Review**

This systematic review investigates deep learning methods for detecting social media bots, which serve both beneficial and malicious purposes [34]. The study underscores the necessity for advanced detection mechanisms due to the increasing sophistication of these bots. By comparing deep learning techniques with traditional machine learning methods, the review establishes a taxonomy of features and pre-processing strategies while identifying research gaps. Following systematic literature review guidelines, the researchers explored deep learning algorithms for bot detection, considering data types, features, and performance in comparison to traditional methods. Their findings highlight the effectiveness and efficiency of deep learning techniques in social bot detection, often surpassing traditional approaches. The study emphasizes the need to adapt detection mechanisms as bots evolve and suggests future research directions for enhancing detection capabilities.

**2.4.2 Supervised Machine Learning**

The research addresses the growing threat of malicious bots on social media platforms, particularly X [35]. By enhancing bot detection accuracy, the study aims to improve overall security and integrity in social media interactions. The methodology involves using a Kaggle dataset with attributes like URL, description, friends count, followers count, screen name, location, ID, and verified status. Key steps include data preprocessing (removing imbalance and irrelevant features), feature engineering (selecting relevant features), model training (using algorithms like Decision Tree, K-Nearest Neighbors, Logistic Regression, and Naïve Bayes), and evaluation (using metrics like accuracy, ROC, and AUC). Notably, a “bag of bots’ words” model enhances detection accuracy. The study emphasizes continuous model adaptation to counter evolving malicious bot tactics and safeguard users from potential threats.  
**2.4.3 Ensemble Machine Learning**

The study aimed to address the limitations of existing bot detection systems by focusing on improving performance and speed through optimal feature encoding, selection, and ensembling [36]. While traditional methods analyze user behavior, this study explored using metadata (static information from Twitter profiles) as an alternative. The methodology involved dataset splitting, WoE encoding, feature selection, and blending using Random Forest. The final model achieved a 93% AUC score, emphasizing the importance of adapting detection mechanisms to evolving bot tactics.

**2.4.4 A Neural Network-Based Ensemble Approach**

The research focuses on improving Convolutional Neural Networks (CNNs) for image classification by addressing diminishing feature reuse in deeper layers [37]. DenseNet, a novel architecture, introduces direct connections between layers with the same feature-map size, enhancing information flow and gradient propagation during training. Each layer applies Batch Normalization, Rectified Linear Units, and 3x3 Convolutions. Transition layers manage complexity, and DenseNet outperforms traditional CNNs and ResNets on benchmark datasets (CIFAR-10, CIFAR-100, SVHN, ImageNet). Its direct connections enhance feature reuse, leading to better gradient flow and superior accuracy with fewer parameters.

**2.4.5 Handling Correlations in Random Forests: Which Impacts on Variable Importance and Model Interpretability?**

In this study, the focus is on enhancing neural network performance through a technique called “feature selection” [38]. The study acknowledges that high-dimensional data can hinder neural network effectiveness and interpretability. Feature selection aims to identify the most relevant features, reducing dimensionality, improving model performance, and minimizing computational costs. The proposed method combines mutual information with neural network training. It ranks features based on relevance to the target variable, iteratively selects performance-enhancing features, and assesses their impact on accuracy, interpretability, and efficiency. Overall, this approach outperforms traditional techniques, demonstrating the value of integrating mutual information and neural network training for feature selection in high-dimensional scenarios.

**2.4.6 Identifying Twitter Bots Using a Convolutional Neural Network**

The paper presents a method for identifying 'X' bots (formerly known as Twitter) using a convolutional neural network (CNN) [39]. Previous approaches to bot detection mainly used tweet metadata rather than content. In the methodology, the authors used a CNN with a word2vec embedding layer generating 300-dimensional vectors, using the dataset of the CLEF 2019 Bots Profiling Subtask’s test. The model was fine-tuned using hyperparameters optimized on a validation dataset. The authors used a powerful machine with 64GB of random-access memory (RAM) and a Nvidia GTX 1080 graphics processing unit (GPU) to train the model. Therefore, the proposed method is effective for identifying 'X' bots, achieving 85.65% validation accuracy and 90.34% test accuracy, outperforming other teams in the competition due to overfitting issues.

## **2.5 SYNTHESIS**

The study’s review of related literature provides an examination of various methodologies and components pertinent to Twitter bot detection using machine learning techniques, particularly focusing on Recurrent Neural Networks (RNNs), Random Forests, and ensemble methods. Twitter bots, which can perform actions like tweeting, retweeting, liking, following, and messaging, are categorized into helpful bots that disseminate important information and harmful bots that spread misinformation and spam. RNNs, known for their capability to process sequential data, are suitable for tasks such as language modeling, text generation, and speech recognition, making them effective in bot detection. The Random Forest algorithm, a robust machine learning method, constructs multiple decision trees from random subsets of the dataset, enhancing prediction accuracy and reducing overfitting, which is crucial for classification tasks like bot detection. Ensemble methods combine multiple learning algorithms to achieve superior predictive performance, enhancing both accuracy and interpretability in complex tasks such as bot detection. The study emphasizes the importance of model accuracy and interpretability, highlighting techniques like Mean Decrease Accuracy (MDA) and Local Interpretable Model-Agnostic Explanations (LIME) for their roles in enhancing the understanding of model predictions.

Furthermore, the literature discusses various tools and libraries essential for implementing the bot detection models, including Visual Studio Code for coding, Python for its simplicity and extensive libraries, Pandas for data manipulation, and sklearn for machine learning implementations. It also reviews the efficiency and effectiveness of deep learning methods in social media bot detection, noting that these techniques often outperform traditional machine learning methods due to their ability to handle complex data patterns. Supervised machine learning approaches are explored, emphasizing their role in enhancing bot detection accuracy.

Overall, the literature presents a comprehensive discussion on the advancements in Twitter bot detection, highlighting the roles of RNNs, Random Forests, and ensemble methods in developing robust and interpretable models. It underscores the importance of using sophisticated tools and libraries to implement these models effectively, pointing to the ongoing need for innovative approaches to improve the accuracy and reliability of social media bot detection methods.

**CHAPTER 3**

**METHODOLOGY**

**OVERVIEW OF THE PROCESS**

This research uses a supervised machine learning approach to predict outcomes based on a dataset obtained from Kaggle. The dataset “Twitter-Bot Detection Dataset” consists of 40,000 rows of structured data, which includes multiple features that are used to train and evaluate predictive models. The primary goal is to compare the performance of an ensemble model combining Recurrent Neural Networks (RNN) and Random Forests (RF) against a baseline model using a 1D Convolutional Neural Network (1D-CNN). As shown in Figure 1, the methodology will be done in a step-by-step structure that includes data collection, preprocessing, model development, ensemble model development, interpretability analysis and lastly the conclusion.

A diagram of a data processing process

Description automatically generated

Fig. 1. Methodology Steps

Figure 1 explains the methodology that will be followed and used by the researchers. The first step is done to gather relevant and representative data that forms the foundation of the study. Without proper data collection, the project lacks the necessary inputs to analyze or train models effectively. Using Twitter Bot Account data from Kaggle ensures the dataset is credible and relevant to the problem being addressed (e.g., identifying bot behavior on social media). The second step is to pre-process the data that have been gathered from Kaggle to prepare the data for analysis and to make sure that the models are trained and tested on clean data. The third step is focused on building and evaluating the models, which includes the metrics and the interpretation, to be used on analyzing the preprocessed data from the second step. The last step highlights the summarization of insights and evaluation of the overall limitations and findings by comparing the models. This also includes any future improvements to further refine the process.

3.1 DATA GATHERING AND PREPROCESSING

3.1.1 TWITTER-BOT DETECTION DATASET

A screenshot of a computer

Description automatically generated

Fig. 2. Bot Detection Dataset

The dataset used for this study was sourced from Kaggle titled “Bot Detection Dataset”, comprising 40,000 records of Twitter accounts labeled as either "bot" or "human." Features include user metadata (e.g., User ID, Username, Tweet, Retweet Count, Mention Count, Follower Count, Verification, Bot Label, Location, Creation of Tweet Date, Hashtags), and behavioral attributes (e.g., tweet frequency, retweet ratio). The dataset is publicly available and adheres to ethical standards for non-PII data usage.

It was used in a study titled “A Deep Learning Approach for Robust Detection of Bots in Twitter using Transformers” that focuses on identifying bot accounts in twitter by using three main aspects of the account specifically it’s activity level, popularity and profile information throughout the use of Deep Neural Network (DNN) [41].

This dataset may be used in determining whether the twitter account is bot or human from the information given by the dataset however the whole information from the dataset might not be used as it is dependent on the needed information for the team to train the model.

3.1.2 DATA PREPROCESSING

3.1.2.1 TOKENIZATION AND REVERSION

The **Tokenization** process began by initializing a Tokenizer from Keras for each specified column (e.g., 'Tweet', 'Location', 'Hashtags'). A close-up of a computer screen

Description automatically generated

Fig. 3. Text to Sequences Method

The text from these columns was converted into numerical sequences using the texts\_to\_sequences method, which maps each word to a unique integer based on a vocabulary index. To ensure uniformity across sequences, the tokenized data was then padded using pad\_sequences, which adds padding (zeros) to sequences shorter than the longest sequence in the dataset. **Reversion** involved converting the padded numerical sequences back into their textual form.



Fig. 4. Sequences to Text

The sequences\_to\_texts method was used to map the integer tokens back to words, allowing the original text to be reconstructed. When padding was added, empty strings were inserted in place of the padded tokens to preserve the length of the sequence. These reverted sequences were stored in new columns alongside the tokenized data to maintain the relationship between the numerical and textual representations of the data. The tokenized sequences, now in numerical format, were used as input for the machine learning models, while the reverted sequences provided a way to verify that the transformation from text to numbers and back was done correctly.

3.1.2.2 EXTRACT DATE AND TIME FEATURES

The function **extract\_date\_time\_features** is a data preprocessing step that converts a single column of raw date-time information into multiple, more granular features representing different aspects of time, such as the year, month, day, hour, minute, second, and day of the week. This transformation is important because machine learning models, including Recurrent Neural Networks (RNNs), Random Forests, and 1D Convolutional Neural Networks (1D CNNs), cannot directly interpret raw date-time values. By breaking down the date-time into components, the models can better leverage temporal patterns within the data. For RNNs, which are used for sequential data, extracting temporal features enables the model to recognize patterns over time, such as daily or seasonal trends. For Random Forests, which excel at handling structured tabular data, decomposing date-time information into features like "day of the week" or "hour" allows the model to capture non-linear relationships between these components and the target variable. Similarly, for 1D CNNs, which operate by learning local patterns in sequences, breaking down date-time data allows the model to detect repeating patterns, such as activity levels during specific times of day or month.

3.1.2.3 REMOVING NON-CONVERTIBLE ROWS

The function **remove\_non\_convertible\_rows** is designed to clean the dataset by ensuring that string values in specified columns can be converted to numeric (float) values. This is important for ensuring that machine learning models, such as RNNs, Random Forests, and 1D CNNs, can process the data correctly since numerical input is required. The function works by checking each value in the specified columns and attempting to convert it to a float. If the conversion fails, the row is removed from the dataset. This helps maintain the integrity of the data and ensures that only valid numerical entries are included in the model training process.

3.1.2.4 CHECKING, HANDLING, AND REMOVING MISSING VALUES

A screenshot of a computer code

Description automatically generated

Fig. 5. Handle Missing Values Function

The function handle\_missing\_values first checks and prints the number of missing values (i.e., NaN values) in each column before any modification is made. It then drops rows where all values are missing using the dropna(how='all') method. This is helpful when an entire row lacks data, as it would not be useful for training machine learning models. After dropping the rows, it prints the updated number of missing values to confirm that the changes have been made correctly.

A screenshot of a computer code

Description automatically generated

Fig. 6. Removing NaN rows Function

The function remove\_rows\_with\_nan takes a more aggressive approach by removing any rows that contain at least one missing value. It first counts the total number of missing values across the DataFrame, prints this count, and then removes all rows that contain any NaN values using dropna(). After the removal, it prints the updated number of missing values and the new shape of the dataset, helping verify the extent of the cleaning. This function is used when it is important to ensure that there are no missing values anywhere in the dataset before proceeding with further analysis or model training.

A screen shot of a computer code

Description automatically generated

Fig. 7. Check for NaN Function

Finally, the check\_for\_nans function is a simple check that scans the DataFrame for any remaining missing values. If the DataFrame contains NaN values, it prints out the number of missing values per column. If there are no missing values, it simply prints a message confirming the absence of NaN values. This function acts as a diagnostic tool to ensure that the data is clean before proceeding with any machine learning tasks.

Handling missing data is essential for machine learning models like Recurrent Neural Networks (RNNs), Random Forests, and 1D Convolutional Neural Networks (1D CNNs) because missing or incomplete data can significantly impact the model's performance, cause errors, or lead to biased results.

3.1.2.5 CORRECTING INCONSISTENCIES

The correct\_inconsistencies function is designed to address inconsistencies in a DataFrame, particularly focusing on the Username column. If the Username column exists, the function converts all the values to lowercase using the .str.lower() method and also removes any leading or trailing whitespace with .str.strip(). This ensures that usernames are consistently formatted, regardless of case or extra spaces. Inconsistent data, such as differing formats, case sensitivity, or missing values, can significantly impact the performance of these models.

3.1.2.6 REMOVING DUPLICATES

A computer code with text

Description automatically generated with medium confidence

Fig. 8. Removing Duplicates Function

The function remove\_duplicates is used to remove duplicate rows from the dataset. The function finds any values that are list type, the value that is considered a list will be added to the list\_columns list. The list type values are converted into string formats in order to be consistent with other strings in the dataset. The function will check for any duplicates, and it will be dropped because duplicates will introduce redundancy, overfitting, and poor performance on new data.

3.1.2.7 HANDLING OUTLERS

A screenshot of a computer code

Description automatically generated

Fig. 9. IQR Outlier Handling

This function is used to remove any outliers that will affect the model’s performance since Recurrent Neural Networks are sensitive to extreme values, Random Forests’ splits can be skewed, and 1D-CNN will focus on noise rather than patterns. The function starts by checking whether all columns have values and if not, it skips outlier handling.

Fig. 10. Interquartile Range Formula

The other parts of the function are based on the Interquartile Range Formula as shown on Figure 10. The **Interquartile Range (IQR)** is specifically the range between the first quartile (Q1), with “quantile (0.25)” and the third quartile (Q3) of a dataset represented with “quantile (0.75)”. The Interquartile Range will used to find the “middle 50%” of the values of a certain column in the dataset and anything beyond that “middle 50%” will be considered outliers and it will be removed.

Fig. 11. Lower Bound and Upper Bound Formula

The “middle 50%” can be found using the formulas of the lower\_bound function and upper\_bound function as shown in Figure 10.

3.1.2.8 NORMALIZING FEATURES

Fig. 12. MinMaxScaler Formula

Where:

X is the original value of a feature. Min(X) is the minimum value of the feature (e.g. 50 – 50,000 followers; Min(X) = 50) and the Max(X) is the maximum value of the feature (e.g. Max(X) = 50,000 followers)

The function normalize\_features uses MinMaxScaler as a default value to make sure that every element in the dataset is set between the range of [0,1] as shown on the formula in Figure 11. This makes sure that all features will be contributing equally to the model training which will prevent any feature to dominate the learning process of the model because of its scale (e.g. Number of Followers compared with Tweet Length)

3.1.2.9 SPLITTING DATASETS

The function split\_and\_save\_human\_bot\_accounts is used in making two different datasets, training and testing, in an 80/20 split.

First, the function will check how many human accounts and bot accounts are present in the dataset. Second, the function will create two new functions for both the humans and bot accounts, namely, human\_train and human\_test for human accounts and bot\_train and bot\_test for the bot accounts. Third, the function will sample 80% of the human accounts for the human\_train variable and 80% of the bot accounts for the bot\_train variable. The bot\_test and the human\_test variables will have the dropped values of 20% coming from the training accounts. Fourth, the function will combine the training accounts (human\_train and bot\_train) into train\_data.csv, and the testing accounts (human\_test and bot\_test) into test\_data.csv. Lastly, the function will shuffle the values in train\_data and test\_data before saving the datasets for the model testing.

3.2 MODEL TRAINING AND TESTING

3.2.1 RECURRENT NEURAL NETWORK MODEL

Recurrent Neural Networks (RNNs) are a class of neural networks specifically designed to handle sequential data, where the current output depends not only on the current input but also on previous inputs. RNNs are highly effective for tasks such as time-series prediction, speech recognition, natural language processing (NLP), and other tasks involving sequences.

RNNs process data step-by-step (time-step by time-step) through the network, where each step has some dependence on the previous one. This allows the model to capture temporal dependencies in the data. At each time step, the RNN maintains a hidden state, which is a summary of the information from the previous steps. The hidden state is updated based on the current input and the previous hidden state. The key characteristic of RNNs is their recurrent connections. The output of a previous time step is fed back as input to the next time step. This allows the network to maintain "memory" of past inputs.

Fig. 13. Recurrent Neural Network

* is the hidden state at time ,
* ​ is the input at time ,
* ​ is the hidden state at time ,
* is the weight matrix connecting the previous hidden state to the current hidden state,
* is the weight matrix connecting the input to the hidden state,
* ​ is the bias vector, and
* is the activation function (often tanh or ReLU).

3.2.1.1 PROCESS OF CREATING RNN MODEL

The model uses Bidirectional LSTM layers to capture both past and future dependencies in sequential data. Bidirectional LSTMs (Long Short-Term Memory networks) are used to capture dependencies in both forward and backward directions across the sequence. In standard LSTMs, information is processed from the start to the end of a sequence. However, in many sequential tasks, information from both the past and the future can be important for making predictions. By using Bidirectional LSTMs, the model can learn from both the past and future context of the sequence, which improves performance in tasks like time-series prediction, speech recognition, and NLP.

In the context of a **Bidirectional LSTM (Long Short-Term Memory)**, **future context** refers to information from the **future** part of the sequence that can influence the current prediction.

Dropout layers are applied to prevent overfitting by randomly deactivating a fraction of neurons during training, with the dropout rate controlling how many neurons are dropped.

A Dense layer is used at the output to produce a single probability for binary classification (bot vs. non-bot), with a sigmoid activation function.

Hyperparameters like **units1** and **units2** define the number of units in the LSTM layers, controlling model complexity; **dropout\_rate** determines how much regularization is applied; **learning\_rate** affects how quickly the model's weights are adjusted during training; **batch\_size** influences how many samples are processed before updating the model; and **epochs** define how many times the model processes the entire dataset.

**Early stopping** halts training if the validation loss stops improving, preventing overfitting and saving resources, while **learning rate reduction** dynamically adjusts the learning rate when the model's performance plateaus, helping the model converge more efficiently.

3.2.1.2 HYPERPARAMETER TUNING: RNN

A screenshot of a computer code

Description automatically generated

Fig. 14. Hyperparameter Tuning: RNN

**model\_\_units1: [64, 128]** This hyperparameter defines the number of units (neurons) in the first LSTM (Long Short-Term Memory) layer of the model. The grid search will test two options for this parameter: 64 or 128 units. The number of units in the LSTM layer controls the model’s capacity to learn from the data. A higher number of units allows the model to capture more complex patterns but also increases computational cost and the risk of overfitting, while fewer units can lead to underfitting if the model is too simple.

**model\_\_units2: [32, 64]** This hyperparameter defines the number of units in the second LSTM layer. Similar to units1, the grid search will test two possible values: 32 or 64 units. A smaller number of units in this second layer can reduce model complexity, which may help in generalizing to unseen data, whereas a larger number of units allows the model to capture more nuanced patterns but might make the model more prone to overfitting.

**model\_\_dropout\_rate: [0.3, 0.5, 0.7]** Dropout is a regularization technique used to prevent overfitting by randomly deactivating a certain fraction of neurons during training. The grid search will test three different dropout rates: 0.3, 0.5, and 0.7. A higher dropout rate helps prevent the model from becoming too reliant on specific neurons and overfitting the training data, but if the rate is too high, the model may lose important information, leading to underfitting. A lower dropout rate provides less regularization and may lead to overfitting if the model is too complex.

**model\_\_learning\_rate: [0.001, 0.0005]** The learning rate controls the step size with which the model’s weights are updated during training. The grid search will test two different learning rates: 0.001 and 0.0005. A higher learning rate (0.001) allows the model to learn faster but may risk overshooting the optimal solution, while a lower learning rate (0.0005) makes the training process more gradual and precise, but it might require more epochs to converge.

**batch\_size: [32, 64]** The batch size determines how many samples are processed together before the model’s weights are updated. The grid search will test two options for batch size: 32 and 64. A smaller batch size (e.g., 32) allows the model to update its weights more frequently, potentially improving generalization but making the training process noisier. A larger batch size (e.g., 64) provides more stable gradients and faster convergence but requires more memory and may result in less frequent updates.

**epochs: [50, 100]** The number of epochs defines how many times the model will iterate through the entire training dataset. The grid search will test two values for epochs: 50 and 100. More epochs (100) give the model more opportunities to learn from the data, but it may increase the risk of overfitting. Fewer epochs (50) might result in underfitting, especially if the model hasn't fully learned the patterns in the data, though early stopping can mitigate this by halting training when the model stops improving on a validation set.

3.2.2 RANDOM FOREST MODEL

A Random Forest is an ensemble learning algorithm that is primarily used for classification and regression tasks. It works by constructing a multitude of decision trees during training and outputs the class (for classification) or the average prediction (for regression) from all the individual trees.

A diagram of a tree

Description automatically generated

Fig. 15. Random Forest Diagram

Random Forest begins with the dataset being randomly sampled with replacement to create multiple subsets; a process known as bootstrap sampling. Each of these subsets, often smaller than the original dataset, is used to independently train a single decision tree. This randomness makes sure that each tree is exposed to a unique view of the data, which adds diversity in the predictions.

In addition to bootstrapping, Random Forest incorporates random feature selection during the training of each decision tree. At every split in a tree, instead of considering all features, the algorithm selects a random subset of features. The best feature within this subset is chosen to make the split. This step prevents certain features from dominating the decision-making process across all trees and introduces further randomness, reducing the correlation between trees and enhancing the robustness of the ensemble.

Once all the decision trees are trained, the algorithm uses them collectively to make predictions. Each tree produces its own prediction for a given input. These individual predictions are then aggregated. For classification tasks, the final prediction is determined by majority voting, where the class predicted by most of the trees becomes the output. For regression tasks, the predictions are aggregated by averaging, providing a single continuous output.

The combination of bootstrapping and random feature selection ensures that the decision trees are diverse and uncorrelated, addressing key limitations of individual decision trees, such as overfitting. By averaging or voting over the predictions of multiple diverse trees, Random Forest reduces both variance and bias, striking a balance between underfitting and overfitting.

3.2.2.1 PROCESS OF CREATING RANDOM FOREST MODEL

The provided code demonstrates the process of using a Random Forest classifier with hyperparameter optimization to build a robust predictive model. Initially, a RandomForestClassifier object (rf\_independent) is defined with a fixed random seed for reproducibility. A hyperparameter grid (rf\_param\_grid) is created, specifying possible values for critical parameters such as the number of trees (n\_estimators), maximum tree depth (max\_depth), the minimum number of samples required for a split or leaf (min\_samples\_split and min\_samples\_leaf), and the maximum number of features considered at each split (max\_features).

To identify the optimal combination of these parameters, a GridSearchCV object is used. This performs an exhaustive search over all possible parameter combinations defined in rf\_param\_grid, using 5-fold cross-validation (cv=5). The model's performance is evaluated for each combination using the F1 score (scoring='f1') as the metric, ensuring a balanced consideration of precision and recall. The n\_jobs=-1 argument allows parallel computation to speed up the search process.

After fitting the model with the training data (X\_train), the best hyperparameters are extracted using the best\_params\_ attribute. The model with the optimal configuration is then used to predict the target labels for the test dataset (X\_test) through its predict method. This approach ensures that the Random Forest model is fine-tuned for the given data and achieves a balance between overfitting and underfitting.

3.2.2.2 HYPERPARAMETER TUNING: RANDOM FOREST

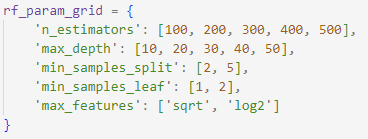


Fig. 16. Hyperparameter Tuning: Random Forest

The hyperparameters defined in the rf\_param\_grid for the Random Forest classifier control various aspects of the model's structure and learning process. The **n\_estimators** parameter specifies the number of decision trees in the forest. A larger number of trees generally improves performance by reducing variance, though it also increases computation time. The **max\_depth** parameter determines the maximum depth of each decision tree. Limiting the depth helps prevent overfitting, ensuring the model does not become overly complex by capturing noise in the data. On the other hand, deeper trees might capture more intricate patterns in the dataset.

The **min\_samples\_split** parameter controls the minimum number of samples required to split an internal node. Higher values make the trees less complex, as splits will only occur when there is a sufficient number of samples, reducing the likelihood of overfitting. Similarly, **min\_samples\_leaf** defines the minimum number of samples that must be present in a leaf node. This parameter ensures that leaves are not too small, helping the model generalize better.

The **max\_features** parameter limits the number of features considered for splitting at each node. Setting this to 'sqrt' uses the square root of the total number of features, while 'log2' uses the base-2 logarithm of the total number of features. This randomness in feature selection helps decorrelate the trees, making the forest more robust and less prone to overfitting. Together, these hyperparameters provide flexibility in tuning the Random Forest model to achieve an optimal trade-off between bias and variance.

3.2.3 RNN AND RANDOM FOREST STACKED MODEL

In a stacked model combining a Bidirectional Long Short-Term Memory (Bi-LSTM) Recurrent Neural Network (RNN) and a Random Forest (RF), the flow is designed to leverage the strengths of both models. The process begins with feeding sequential input data, such as time-series or text, into the Bi-LSTM model. The Bi-LSTM processes the input bidirectionally, capturing dependencies from both past and future contexts within the sequence. This results in a set of high-level, context-aware features that represent the input more effectively.

These features are then passed to the Random Forest, which acts as the next layer in the stacked model. The Random Forest uses these features to make the final decision, like classifying or predicting outcomes. While the Bi-LSTM is great at understanding sequential data, the Random Forest is skilled at handling structured data and making accurate decisions. By combining them, the stacked model becomes more powerful and can work well with both complex patterns and structured decision-making.

3.2.3.1 PROCESS OF CREATING STACKED MODEL

The code provided demonstrates the process of building and evaluating a stacked model that combines the strengths of a Recurrent Neural Network (RNN), specifically a Bidirectional LSTM (Bi-LSTM) or similar architecture, and a Random Forest classifier. The flow begins by generating probabilistic predictions using the trained RNN model. These predictions represent the likelihood of different classes for the training and test datasets. The predicted probabilities (rnn\_train\_probs for training data and rnn\_test\_probs for test data) are flattened into one-dimensional arrays, making them compatible for further use.

Next, these RNN probabilities are integrated into the feature sets used by the Random Forest model. This is done by creating copies of the original training and test datasets (X\_train and X\_test), then appending the RNN probabilities as a new column, labeled RNN\_Prob. This transforms the feature sets, so they now include the outputs of the RNN alongside the original features, which will serve as an additional informative input for the Random Forest.

The core of the evaluation is the Random Forest model, which is trained using the augmented feature sets (X\_train\_rf and X\_test\_rf) that now include the RNN\_Prob feature. The GridSearchCV is used for hyperparameter tuning, which involves testing different combinations of Random Forest parameters, such as the number of trees (n\_estimators), tree depth (max\_depth), minimum samples required for splits (min\_samples\_split), and the number of features considered at each split (max\_features). This tuning process involves running the Random Forest with 5-fold cross-validation (cv=5), optimizing for the F1 score, a metric that balances precision and recall.

Once the grid search completes, the best Random Forest model, determined by the highest F1 score during cross-validation, is selected. The optimal hyperparameters of the best Random Forest model are then printed, providing insight into the specific settings that resulted in the best performance for this stacked model. The result is a hybrid model that combines the sequential data-processing capabilities of the RNN with the strong, ensemble-based decision-making power of the Random Forest, leading to a model that benefits from both architectures and is likely to have improved performance compared to either model used individually.

3.2.3.2 HYPERPARAMETER TUNING: STACKED MODEL

**A computer code with numbers and symbols

Description automatically generated**

Figure 17. Hyperparameter Tuning: Stacked Model

**n\_estimators**: This hyperparameter specifies the number of trees (or estimators) in the ensemble model. It determines how many individual decision trees will be built to make predictions. A higher number of trees can improve the model’s performance by increasing its predictive power and stability, as more trees typically reduce the variance. However, this also increases the computational time required for training and prediction. The values [100, 200, 300, 400, 500] represent different potential values for the number of trees. The ideal number can vary depending on the dataset, but testing a range helps identify the optimal trade-off between performance and computational efficiency.

**max\_depth**: The max\_depth parameter limits how deep each individual decision tree can grow. The depth of the tree determines the number of levels it has, which influences how well it can capture complex relationships in the data. Deeper trees can model more intricate patterns, but they also run the risk of overfitting, especially if the dataset is noisy or too small. A shallow tree may underfit, failing to capture important patterns in the data. The values [10, 20, 30, 40, 50] allow exploration of trees with different complexities to balance model performance and generalization.

**min\_samples\_split**: This parameter sets the minimum number of samples required to split an internal node of the tree. By controlling this value, you can restrict the tree from creating splits that occur too early in the process when the number of samples is small. A higher value for min\_samples\_split means the tree will only split nodes where there are enough samples, thus reducing the possibility of creating overly specific branches that fit to noise in the data (overfitting). The values [2, 5] represent common choices for this parameter, with 2 being the default for many tree-based algorithms, and 5 enforcing a more conservative split.

**min\_samples\_leaf**: The min\_samples\_leaf parameter sets the minimum number of samples that a leaf node must have. A leaf node is where the tree stops splitting, and it predicts the output. By increasing the value of min\_samples\_leaf, you can ensure that leaf nodes contain enough data to make reliable predictions. Larger values can help reduce model complexity and prevent overfitting by avoiding leaf nodes with too few samples, which may be overly specific to the training data. The values [1, 2] are typical choices. A value of 1 allows for deeper trees, while a value of 2 promotes more generalization.

**max\_features**: This hyperparameter specifies the number of features to consider when searching for the best split at each node. Limiting the number of features helps control the complexity of the individual trees and prevents overfitting. It also speeds up the training process since the model doesn't have to evaluate all the features for each split. The values 'sqrt' and 'log2' are common options:

* 'sqrt' uses the square root of the total number of features, which is a typical choice for RandomForest models and often strikes a good balance between performance and speed.
* 'log2' uses the base-2 logarithm of the number of features, which can also reduce model complexity but tends to be more aggressive in reducing the number of features used for splits.

3.2.4 1D-CONVOLUTIONAL NEURAL NETWORK MODEL

A diagram of a computer program

Description automatically generated

Fig. 18. 1D-CNN Diagram

A **1D Convolutional Neural Network (1D-CNN)** is a specialized variant of the Convolutional Neural Network (CNN), primarily used for processing sequential data. Unlike traditional 2D-CNNs, which operate over images (two-dimensional grids of pixels), 1D-CNNs are designed to process data with a one-dimensional structure, such as time series, audio signals, or text sequences. The input to a 1D-CNN is typically a one-dimensional array, where each element represents a point in a sequence, such as a time step in a time series or a word in a sentence. For example, in a time series context, the input could be a vector representing a sequence of sensor readings over time.

The core operation in a 1D-CNN is the **convolutional layer**, where a set of filters (or kernels) slides across the input data. The filter performs an element-wise multiplication with the input data, followed by a summation, extracting local features or patterns from small segments of the input sequence. For instance, a filter might focus on identifying trends in a specific window of time or detecting the presence of specific patterns in a segment of text. The filter slides across the sequence with a step size called the **stride**, typically set to 1, meaning the filter moves one element at a time. This convolution operation helps capture local dependencies in the input data, such as short-term correlations in a time series or n-gram features in text.

After the convolution operation, the output is passed through an **activation function**, commonly the **ReLU (Rectified Linear Unit)** function. ReLU introduces non-linearity by transforming all negative values to zero while keeping positive values unchanged. This helps the model learn more complex patterns by adding non-linearity, which enables the network to represent a wider variety of relationships between the features. The activation function ensures that the network can capture intricate relationships within the data, beyond simple linear patterns.

Next, a **pooling layer** is typically applied to reduce the dimensionality of the feature maps and make the model more computationally efficient. In 1D-CNNs, **1D Max Pooling** is commonly used, where a window (usually of size 2) slides over the output from the convolutional layer, and only the maximum value within each window is retained. This operation helps reduce the data size while preserving the most important features, making the model more robust to small translations or variations in the input data.

After several convolutional and pooling layers, the output is usually **flattened** into a one-dimensional vector. This flattened vector is then passed through one or more **fully connected (dense) layers**. The fully connected layers combine the features learned from the convolutional layers to make a final prediction. In these layers, every neuron is connected to every neuron in the previous layer, allowing the model to combine the extracted features in increasingly abstract ways. The final output layer typically uses an activation function like **Softmax** (for multi-class classification) or **Sigmoid** (for binary classification) to make a final prediction based on the learned features.

3.2.4.1 PROCESS OF CREATING 1D-CNN MODEL

The function create\_1d\_cnn defines the architecture of the 1D-CNN model. It takes the input\_shape (which specifies the dimensions of the input data) as an argument, along with several hyperparameters like filters, kernel\_size, dense\_units, and dropout\_rate to control the architecture of the network. The model starts with a Conv1D layer, which applies 1D convolutions with the specified number of filters (filters=64 by default) and kernel size (kernel\_size=3). This layer helps the model capture local patterns in the sequential data. The output of the convolution is passed through a MaxPooling1D layer with a pool size of 2 to down-sample the data, reducing its dimensionality while preserving important features. A Dropout layer with a rate of 0.5 is added after each convolution and pooling step to reduce overfitting by randomly setting a fraction of input units to zero during training.

The network then has another Conv1D layer with double the number of filters (filters \* 2), followed by another MaxPooling1D layer. Afterward, the output is flattened (using the Flatten() layer) to convert the 2D feature map into a 1D vector, which is then passed through a Dense layer with 64 units and ReLU activation to learn more complex representations of the data. Another Dropout layer is added before the final output layer, which is a Dense layer with a single unit and Sigmoid activation to output a probability value between 0 and 1 for binary classification.

The model is compiled using the Adam optimizer with binary cross-entropy loss (appropriate for binary classification) and accuracy as the evaluation metric.

The model is trained using the fit() method on the training data (X\_train\_cnn and y\_train\_noisy). The training runs for 50 epochs with a batch size of 32. The validation\_split=0.2 parameter indicates that 20% of the training data will be used for validation during training. Two callbacks are used to improve training:

EarlyStopping: This stops training if the validation loss does not improve for 10 consecutive epochs, ensuring that the model does not overfit. The best weights are restored when this happens.

ReduceLROnPlateau: This reduces the learning rate by a factor of 0.5 if the validation loss plateaus for 5 epochs, which helps in fine-tuning the model's performance as it approaches convergence.

3.2.4.2 HYPERPARAMETER TUNING: 1D-CNN

A screenshot of a computer program

Description automatically generated

Fig. 19. Hyperparameter Tuning: 1D-CNN

This code defines a function called create\_1d\_cnn that constructs a 1D Convolutional Neural Network (1D-CNN) model. The function takes input\_shape as an argument, which defines the shape of the input data, and other optional hyperparameters like filters, kernel\_size, dense\_units, and dropout\_rate to customize the architecture. The purpose of this function is to create a model that can handle sequential data (such as time series or text) and perform binary classification.

The input layer takes input\_shape as a parameter, which indicates the dimensions of each sample in the input dataset. For example, if the input consists of sequences with 100-time steps and 1 feature, the input\_shape might be (100, 1). The Input layer essentially acts as a placeholder for the input data that will be fed into the model during training and testing.

The first 1D Convolutional Layer (Conv1D) applies 1D convolution operations to the input data. The filters=64 argument specifies that this layer will have 64 filters, which means it will learn 64 different feature representations from the input data. The kernel\_size=3 argument defines the size of the filter, meaning the convolution will look at sequences of 3 consecutive time steps (or values) at a time. The activation='relu' applies the ReLU activation function to the output of the convolution, introducing non-linearity and enabling the model to learn more complex patterns.

After the convolution, a MaxPooling1D layer is used with a pool\_size=2. This layer reduces the dimensionality of the output feature map by down-sampling it, retaining only the maximum value from each window of 2 consecutive values. Max pooling helps make the model more computationally efficient and less prone to overfitting by making the network less sensitive to small variations or shifts in the input.

A Dropout layer is added with a dropout\_rate=0.5. This layer randomly sets 50% of the input units to zero during training. Dropout is a regularization technique that helps prevent overfitting by forcing the network to learn more robust features and preventing it from relying too heavily on specific neurons. This ensures that the model generalizes better to unseen data.

The second 1D Convolutional Layer uses filters=filters \* 2, meaning the number of filters is doubled compared to the first convolutional layer (so 128 filters are learned). The kernel\_size=3 and activation='relu' remain the same as the first convolutional layer. This second convolutional layer allows the model to learn more complex features and interactions between the input sequence's values. By increasing the number of filters, the model can capture more nuanced patterns in the data.

Like the first pooling layer, a second MaxPooling1D layer is applied to reduce the spatial dimensions of the output feature map. With pool\_size=2, this layer again down-samples the data by keeping only the maximum value from each pair of consecutive values, further reducing the size of the feature map while preserving important features.

After the convolutional and pooling layers, the output is a multi-dimensional array, which needs to be flattened into a one-dimensional vector before being passed to fully connected layers. The Flatten layer reshapes the multi-dimensional data into a 1D array, enabling the model to connect the learned features from the convolutional layers to the dense layers.

The flattened data is then passed through a Dense layer with dense\_units=64 and the activation function relu. A dense layer is a fully connected layer where each input is connected to every neuron. The relu activation function is applied to introduce non-linearity, allowing the model to learn more complex relationships between the learned features. The number of units (64 in this case) determines the dimensionality of the output from this layer.

Another Dropout layer is applied after the dense layer, with the same dropout\_rate=0.5. This further helps to regularize the model by preventing overfitting and ensuring that the network doesn't become overly dependent on any specific neuron or feature.

The final Dense layer has a single unit (Dense(1)) and uses the Sigmoid activation function (activation='sigmoid'). This layer outputs a probability value between 0 and 1, which represents the model's prediction for binary classification. A value close to 1 indicates one class (e.g., positive), and a value close to 0 indicates the other class (e.g., negative). The Sigmoid function is ideal for binary classification tasks because it outputs a probability score.

3.2.5 EVALUATION METRICS

**1. Accuracy:**

Accuracy is the proportion of correct predictions (both true positives and true negatives) to the total number of samples. It gives a general sense of how well the model performs overall, but it may not always reflect the model's ability to handle imbalanced classes.

Formula:

A black and white text

Description automatically generated

Where:

TP = True Positives (correctly predicted positive class)

TN = True Negatives (correctly predicted negative class)

FP = False Positives (incorrectly predicted as positive)

FN = False Negatives (incorrectly predicted as negative)

**2. Precision:**

Precision, also called Positive Predictive Value (PPV), measures the proportion of correctly predicted positive observations to all predicted positives. It answers the question: "Of all the instances the model predicted as positive, how many were actually positive?"

Formula:

A mathematical equation with black text

Description automatically generated

Where:

* **TP** = True Positives
* **FP** = False Positives

A higher precision means fewer false positives.

**3. Recall (also called Sensitivity or True Positive Rate):**

Recall measures the proportion of correctly predicted positive observations to all actual positives. It answers the question: "Of all the actual positive instances, how many did the model correctly identify?"

Formula:

A math equation with black text

Description automatically generated

Where:

* **TP** = True Positives
* **FN** = False Negatives

A higher recall indicates fewer false negatives, but it may come at the cost of more false positives.

4. **F1-Score:**

The F1-Score is the harmonic mean of Precision and Recall. It balances both Precision and Recall into a single metric. It is particularly useful when the data is imbalanced, as it gives a better sense of model performance when there are more instances of one class than the other.

**Formula**:

A black text on a white background

Description automatically generated

Where:

The F1 score ranges between 0 (worst) and 1 (best). A high F1 score indicates that both precision and recall are high.

5. **ROC Curve (Receiver Operating Characteristic Curve):**

The ROC curve is a graphical representation of a binary classifier's performance. It plots the True Positive Rate (Recall) against the False Positive Rate (FPR) at various threshold values. The curve helps visualize the trade-off between sensitivity and specificity at different thresholds.

* **True Positive Rate (TPR)**



* **False Positive Rate (FPR)**



The ROC curve typically shows how the model performs across different classification thresholds. A model that randomly guesses will produce a diagonal line from the bottom-left corner to the top-right corner.

6. **AUC Score (Area Under the Curve):**

The AUC (Area Under the ROC Curve) score is the area under the ROC curve. It gives a single number that summarizes the performance of the classifier across all possible thresholds. A higher AUC means that the model is better at distinguishing between the positive and negative classes.

**AUC Interpretation**:

* **AUC = 0.5**: The model performs no better than random guessing.
* **AUC = 1.0**: The model perfectly distinguishes between positive and negative classes.
* **AUC < 0.5**: The model is worse than random guessing.

AUC ranges from 0 to 1, and a higher AUC is generally better.

7. **Cross-Validation Score:**

Cross-validation is a technique used to assess the generalization performance of a model. It involves splitting the data into multiple subsets (folds), training the model on some folds and testing it on the remaining folds. The performance of the model is averaged over all folds to provide a more reliable estimate of its accuracy.

8. **Confusion Matrix:**

The Confusion Matrix is a fundamental tool for understanding the performance of a classification model. It provides a detailed breakdown of the model's predictions by showing how many instances were correctly or incorrectly classified for each class. It is especially useful for visualizing the performance of a model, particularly in imbalanced class problems.

A close up of a sign

Description automatically generated

Where:

* **TP** (True Positives): The number of positive class instances that were correctly classified as positive.
* **TN** (True Negatives): The number of negative class instances that were correctly classified as negative.
* **FP** (False Positives): The number of negative class instances that were incorrectly classified as positive.
* **FN** (False Negatives): The number of positive class instances that were incorrectly classified as negative.

3.2.6 INTERPRETABILITY ANALYSIS

3.2.6.1 FEATURE IMPORTANCES

Feature Importance is a technique used to identify which features most significantly contribute to a model’s predictions. This method is commonly employed in tree-based models such as Random Forests.

In these models, feature importance is computed based on how much a given feature reduces the impurity (entropy) when used to split nodes in a decision tree. A feature that frequently leads to large reductions in impurity is considered more important.

To calculate the importance of each feature, the model is trained, and the contributions of each feature are assessed through their usage across the decision trees. Features that have a high frequency of use in making splits, or that lead to significant reductions in the loss function, are ranked higher in importance. A threshold can be set to exclude less important features by selecting those that surpass a certain importance score. For example, if a feature's importance score is below a set threshold (e.g., 0.01), it might be discarded from the model for feature selection purposes.

This method helps reduce dimensionality and improves the interpretability of the model by focusing on the most influential predictors.

3.2.6.2 LOCAL INTERPRETABLE MODEL-AGNOSTIC EXPLANATIONS

**LIME** is a model-agnostic technique designed to explain the predictions of complex machine learning models, especially those that are difficult to interpret, such as neural networks or ensemble models.

LIME operates by generating a surrogate model that is interpretable, like a linear regression or decision tree, to approximate the behavior of the complex model in the local neighborhood of a specific data point. This process involves creating synthetic data points by slightly perturbing the original instance, simulating how the complex model might behave for these perturbed data points. LIME then fits a simple, interpretable model to this local data, which helps identify the most important features influencing the prediction for that instance.

The feature importance scores are derived from the surrogate model, which highlights how much each feature contributes to the decision for the given prediction. A threshold can be applied to the feature importance values, with features having importance scores above a certain value being highlighted in the explanation. If a feature's importance score is too low, it may be excluded from the explanation, simplifying the interpretability. LIME is particularly useful when trying to understand how a model arrives at a specific prediction, providing transparency in complex black-box models.

3.2.6.3 MEAN DECREASE ACCURACY

**Mean Decrease Accuracy (MDA)** is a feature importance method used primarily with tree-based models like **Random Forests** and **Gradient Boosting Machines**.

MDA measures the decrease in the model’s performance (typically accuracy) when the values of a particular feature are permuted or shuffled randomly. The core idea is that features contributing significantly to the model’s predictions will cause a noticeable drop in performance when their values are randomized, as they contain valuable information for making accurate predictions.

To calculate MDA, the model is first trained on the original dataset, and its baseline performance is evaluated. Then, for each feature, the values are permuted (shuffled), and the model’s performance is re-evaluated. The difference in performance before and after permutation is used to determine the importance of that feature. A feature that leads to a significant decrease in performance when shuffled is considered more important. Features that have little to no impact on the model’s performance when permuted are considered less important. Similar to other feature importance methods, a threshold can be set for MDA scores to select the most relevant features. Features with low MDA scores can be excluded from the model, streamlining the feature set while preserving predictive power.

3.3 SYNTHESIS

Data preprocessing is a critical first step in preparing raw data for machine learning tasks, involving several key steps to clean and structure the data. The process begins with tokenization, where textual columns like 'Tweet', 'Location', and 'Hashtags' are converted into numerical sequences using Keras' Tokenizer, with sequences padded to ensure uniform length. Additionally, the original text is reverted from tokenized form for interpretability, and date-time features are extracted from a 'Created At' column, breaking it down into components such as year, month, and day. Missing values are handled through KNN imputation, and duplicates and inconsistencies in columns like 'Username' are addressed by normalizing text to lowercase. Normalization of numerical features such as 'Follower Count' and 'Retweet Count' ensures that the model treats all features equally, using MinMaxScaler or StandardScaler. Outliers are detected using the interquartile range (IQR) method, and any remaining NaN values are removed. The data is then split into training and testing sets, with the 'Bot Label' column used to differentiate human and bot accounts, creating 80/20 splits for training and testing. The preprocessed data is saved into a new CSV file, ready for model training. This pipeline ensures the data is cleaned, consistent, and structured, laying a strong foundation for building and evaluating machine learning models.

The Bidirectional LSTM RNN, Random Forest, and their stacked combination are all powerful models for various classification tasks, each bringing its own advantages. The Bidirectional LSTM RNN captures temporal dependencies in sequential data by processing inputs in both forward and reverse directions, making it ideal for tasks like sentiment analysis or time-series forecasting. Random Forest, an ensemble learning method, is well-suited for handling structured, high-dimensional data by creating multiple decision trees and aggregating their outputs, offering strong generalization and robustness against overfitting. Combining these two in a stacked model leverages the strengths of both approaches, allowing the LSTM to model sequential dependencies while the Random Forest captures complex feature interactions, potentially improving predictive performance. For comparison, a 1D CNN is also used, which excels at identifying local patterns in data through convolutions, typically for tasks involving sequence or time-series data. While the 1D CNN can be effective for learning spatial hierarchies of features, it is mainly used here for comparison, showcasing how its feature extraction capabilities compare to the sequential learning ability of LSTMs and the ensemble power of Random Forests in the context of classification tasks.

Evaluation metrics are essential for assessing the performance of machine learning models, with each providing unique insights into the model's effectiveness. Accuracy measures the overall correctness of the model, indicating the proportion of true predictions (both true positives and true negatives) to the total predictions. However, in imbalanced datasets, metrics like precision, recall, and F1-score become more informative. Precision focuses on the accuracy of positive predictions, whereas recall measures the model's ability to correctly identify all relevant instances. The F1-score combines precision and recall into a single metric, providing a balance between them, especially when there is a trade-off between the two. The ROC Curve visually represents the trade-off between true positive rate (recall) and false positive rate, and the AUC Score quantifies this trade-off, with higher values indicating better model performance. Cross-validation score is used to evaluate the model's robustness by splitting the data into multiple subsets, training, and validating the model on each subset, ensuring that the results are not overly dependent on a single train-test split. The Confusion Matrix provides a detailed breakdown of the model’s predictions by showing the true positives, false positives, true negatives, and false negatives, allowing for a deeper understanding of the model’s performance across different classes.

**CHAPTER 4**

# **RESULTS AND DISCUSSION**

This chapter will center around the results and discussion of the stacked model’s training and evaluation. The evaluation will be done using the following metrics: F1, Precision, Recall, Accuracy, Confusion Matrix, Cross Validation, ROC Curve and AUC Score. The results for each model (RNN, Random Forest, Stacked Model, and 1D-CNN) will be presented with the metrics to ensure their high accuracy in detecting bots by highlighting each of their performances on the dataset and new data, using Cross Validation.

# **4.1 DATA CONSTRUCTION**

This step is an important phase to be done since it is a critical step that ensures the quality and the relevance of the dataset for effective model training, and evaluation. This section provides a detailed account of how the dataset is used and how it is processed for the training and the evaluation of the bot detection models.

4.1.1 DATA COLLECTION

The dataset used in this study is the publicly available "Twitter-Bot Detection Dataset" created by Aditya Goyal and hosted on Kaggle. This dataset serves as the primary resource for training and evaluating the proposed bot detection models. Its balanced composition ensures equal representation of bot and human accounts, which is important for unbiased model development.

The Twitter-Bot Detection Dataset was chosen for its quality and relevance to the research objectives. Created by Aditya Goyal, the dataset contains labeled examples of both bot and human accounts, providing a reliable foundation for supervised machine learning tasks. It was downloaded directly from Kaggle under its public license agreement.

The dataset was downloaded as a CSV file named "bot\_detection\_dataset.csv" from Kaggle. It is publicly available and designed specifically for bot detection research, eliminating the need for additional data collection. The file structure is user-friendly, with each row representing a user and their associated data.

The dataset comprises a total of 49,994 rows, with the following columns and class distributions:

Table I  
DATASET COLUMN NAMES

|  |  |
| --- | --- |
| Column Name | Description |
| User ID | Unique identifier for each user in the dataset. |
| Username | The username associated with the user. |
| Tweet | The text content of the tweet. |
| Retweet Count | The number of times the tweet has been retweeted. |
| Mention Count | The number of mentions in the tweet. |
| Follower Count | The number of followers the user has. |
| Verified | A boolean value indicating whether the user is verified or not. |
| Bot Label | A label indicating whether the user is a bot (1) or not (0). |
| Location | The location associated with the user. |
| Created At | The date and time when the tweet was created. |
| Hashtags | The hashtags associated with the tweet. |

* Human Accounts: 24,980 samples.
* Bot Accounts: 25,014 samples.

The balanced nature of the dataset eliminates the need for additional oversampling or under sampling techniques to address class imbalance. Each row in the dataset represents a user account and contains features such as profile details, activity metrics, and content statistics.

Since the dataset was obtained from a public Kaggle repository, it complies with ethical guidelines for data usage. The dataset does not contain personally identifiable information (PII) and ensures anonymity of all user accounts.

The Twitter-Bot Detection Dataset provides a balanced dataset suitable for evaluating bot detection models. The CSV file format and the included features make it straightforward to preprocess, analyze, and use for model development. Its use eliminates the need for additional data collection or preprocessing for balancing class distributions, allowing focus on model development and analysis.

4.1.2 DATA PREPROCESSING

4.1.2.1 EXTRACTING DATE-TIME FEATURES

In the dataset, the Created At column contained timestamps for each tweet, which provided temporal information. However, this raw timestamp alone would not be directly useful for our models. To improve the model's ability to detect patterns related to time, we extracted several date-time features from this column. These features included Year, Month, Day, Hour, Minute, Second, and Day of Week. By splitting the timestamp into these individual components, this allowed the model to capture granular patterns in bot behavior that could vary depending on the time of day, day of the week, or even specific months or years.

Table II  
EXTRACTING DATE-TIME FEATURES

|  |  |  |
| --- | --- | --- |
| Created At | Extracted Feature | Example Value |
| 11/05/2020 15:29 | Year | 2020 |
| Month | 11 |
| Day | 5 |
| Hour | 15 |
| Minute | 29 |
| Second | 0 |
| Day of Week | 3 |

In the process of extracting these features, we encountered missing or inconsistent data in some of the time-related columns. For instance, some rows had incomplete or invalid timestamps, leading to missing values in the extracted Year, Month, Day, Hour, Minute, Second, or Day of Week columns. These missing values could have introduced noise and errors into our model if left unaddressed.

4.1.2.2 HANDLING MISSING VALUES

Initially, the dataset contained a considerable number of missing values, particularly in the columns 'Username', 'Year', 'Month', 'Day', 'Hour', 'Minute', 'Second', and 'Day of Week'. The missing values for 'Username' (9431) and time-related fields (30481) were significant. Missing values in these columns are common in social media datasets, where user information or temporal data may not be consistently available across all entries.

Table III  
HANDLING MISSING VALUES

|  |  |  |
| --- | --- | --- |
| Column Name | Missing Values (Before Removal) | Missing Values  (After Removal |
| Username | 9,431 | 9,431 |
| Year | 30,481 | 0 |
| Month | 30,481 | 0 |
| Day | 30,481 | 0 |
| Hour | 30,481 | 0 |
| Minute | 30,481 | 0 |
| Second | 30,481 | 0 |
| Day of the Week | 30,481 | 0 |

After handling missing values, the dataset was cleaned, and the columns with missing entries were addressed by either imputing or removing rows where necessary. This process ensured that the data used for training the models was consistent and reliable.

The final dataset after removing rows with NaN values consisted of 15,844 rows, as opposed to the original 49,994. This reduction reflects the removal of incomplete data points, which could introduce noise into the models.

4.1.2.3 ADDRESSING INCONSISTENCIES

The goal of addressing inconsistencies and duplicates was to ensure that the dataset contained unique entries and that text-based features were standardized for uniformity. Inconsistent entries or duplicates can lead to bias and unreliable model predictions, especially when dealing with features like Username, which are critical for identifying individual users.

The Username column, which contained text-based entries, was normalized to lowercase. This was done to ensure consistency, as variations in capitalization (e.g., "User1" vs. "user1") would otherwise be treated as distinct entries. By converting all usernames to lowercase, we made sure that the model would treat the same user consistently, regardless of the case used in the original data.

Table IV  
ADDRESSING INCONSISTENCIES

|  |  |
| --- | --- |
| Username  (Before Normalization) | Username  (After Normalization) |
| CraigTravis | craigtravis |
| JoshuaDavis | joshuadavis |
| GibsonKevin | gibsonkevin |
| HoffmanAnthony | hoffmananthony |
| BrendaWyatt | brendawyatt |

4.1.2.4 HANDLING OUTLIERS

The purpose of handling outliers was to ensure that extreme values in numerical features, such as Follower Count and Retweet Count, did not disproportionately affect the performance of machine learning models. Outliers can distort statistical analyses, lead to overfitting, and reduce model generalization.

Identification of Outliers: Outliers were identified using the interquartile range (IQR) method. The IQR is a statistical measure that captures the spread of the middle 50% of data. Once identified, the extreme values in the Follower Count and Retweet Count columns were removed. This helped reduce the influence of anomalous data points that might otherwise lead to distorted model training.

4.1.2.4.1 COMPUTATION OF IQR

Table V  
COMPUTATION OF IQR

|  |  |
| --- | --- |
| Username | Follower Count |
| stephenfrank | 9439 |
| owensbenjamin | 337 |
| leonard00 | 10 |
| hernandezchristian | 653 |
| mcampbell | 968 |

1. **SORT DATA**: Follower Count: 10,337, 653, 968, 9439
2. **FIND QUARTILES**:

Q1 = = 173.5

Q3 = = 5203.5

1. **CALCULATE IQR**:

IQR = Q3 – Q1 = 5203.5 – 173.5 = 5030

1. **IDENTIFY OUTLIER BOUNDS**:

*Lower Bound*:

Q1 – 1.5 \* IQR = 173.5 – 1.5 \* 5030 = 173.5 – 7545

= -**7371.5**

*Upper Bound*:

Q3 + 1.5 \* IQR = 5203.5 + 1.5 \* 5030 = 5203.5 + 7545 = **12748.5**

Any Follower Count below −7371.5 or above 12748.5 would be considered an outlier. In this case, the value 9439 for Follower Count is within the bounds, so no outliers are detected for Follower Count.

4.1.2.5 NORMALIZATION OF NUMERICAL FEATURES

Features like Follower Count and Retweet Count, which have values on different scales (e.g., Follower Count can range from a few to millions, while Retweet Count may have smaller values), were normalized using the Min-Max Scaling method. This method scales the values of these features to a range of 0 to 1.

* + - * 1. COMPUTATION OF MIN-MAX SCALING

Table VI  
COMPUTATION OF MIN-MAX SCALING

|  |  |
| --- | --- |
| Username | Follower Count |
| stephenfrank | 9439 |
| owensbenjamin | 337 |
| leonard00 | 10 |
| hernandezchristian | 653 |
| mcampbell | 968 |

1. Get Min and Max Values:
2. Apply Min-Max Scaling to Follower Count:

Min-Max Scaling transformed the original values into a range of [0, 1]. The Follower Count and Retweet Count values are now normalized, making them suitable for machine learning algorithms that require features to be on a similar scale.

4.1.3 FEATURE ENGINEERING

The process of tokenization transforms raw textual data into numerical representations that are suitable for machine learning models. This approach allows algorithms to process and understand language patterns in a structured manner. Using the TensorFlow tokenizer, textual data such as tweets, locations, and hashtags were converted into sequences of numerical tokens.

For example, the sentence:

**"Station activity person against natural majority none few size expect six marriage**" was tokenized into a sequence of numerical identifiers:

**[0, 156, 270, 237, 301, 256, 506, 77, 667, 943, 910, 292, 123].**

Each token represents a unique word mapped to a vocabulary index. This mapping enables the model to distinguish and process individual words efficiently. The conversion to numerical tokens ensures that the text data can be further used in embedding layers or transformed into other feature representations, such as TF-IDF or word embeddings.

Additionally, tokenized sequences were reverted into text for debugging and validation purposes, ensuring that the mapping between tokens and words was accurate. This step is crucial in maintaining data integrity and verifying preprocessing workflows.

# **4.2 MODEL TRAINING**

The model training process forms a critical component of the bot detection project, involving the preparation of data, allocation of computational resources, and establishment of an optimized environment for executing machine learning algorithms. This section outlines the key elements involved in ensuring robust and reliable training of the models.

The setup of environment and the hardware used for the model training and testing are the same throughout the different model iterations, which makes sure that each of the model will have consistent performance and comparison. By making sure that the researchers use the same setup and hardware, they were able to focus more on the performance and interpretations of the model to make reliable results.

4.2.1 TRAINING AND TESTING DATA

The dataset was carefully split into training and testing subsets to ensure proper class balance and facilitate robust model evaluation. This was achieved by leveraging the functions check\_human\_bot\_accounts and split\_and\_save\_human\_bot\_accounts, which addressed key challenges in preparing the dataset for machine learning.

An important aspect of this process was maintaining an even representation of human and bot accounts across both training and testing sets. The dataset was first segmented into two groups based on the Bot Label column, where 0 represented human accounts and 1 represented bot accounts. Each group was split into an 80% training set and a 20% testing set. This stratification ensured that both subsets were balanced, preventing bias toward either class during model training or evaluation.

A screenshot of a computer

Description automatically generated

Fig. 20. Numbers of human and bot accounts

This data-splitting strategy directly contributed to fair and unbiased model evaluation. Metrics like precision, recall, F1-score, and accuracy were calculated on a balanced test set, offering a reliable assessment of the model's capability to detect bots. Additionally, by ensuring reproducibility, the process laid a strong foundation for consistent experimentation and comparison between different models.

4.2.2 HARDWARE

All the models are trained on a high-performance PC with the following specifications:

* Graphics Processing Unit (GPU) – RTX 3060
* Central Processing Unit (CPU) – AMD Ryzen 5 5600X 6-Core Processor with 12 CPUs ~3.7GHz
* Random Access Memory (RAM) – 32768MB RAM

These resources have provided enough computational power to handle the training and the testing of all the models. The training duration for the models, since they were all in a single file, would take an approximate of 30,000 seconds (+-) or 8 hours (+-) if the full dataset is used. When testing for debugging, the team would only use 500-1000 rows of data which would result to around 4,000 seconds (+-) or 1 hour (+-).

4.2.3 ENVIRONMENT SETUP

The project was developed using Visual Studio Code (VS Code), a powerful and user-friendly integrated development environment (IDE). VS Code's features such as Intellisense for intelligent code suggestions, built-in debugging tools, and a versatile terminal contributed significantly to productivity. The addition of extensions, including Python and Jupyter support, streamlined the development of scripts and the execution of data analysis and machine learning models. This setup provided a cohesive environment, minimizing context-switching and enabling an end-to-end workflow within a single platform.

4.2.3.1 DATA PREPROCESSING LIBRARIES

A screenshot of a computer code

Description automatically generated

Fig. 21. Libraries used for data preprocessing

To transform raw data into a suitable format for machine learning, the following tools were employed:

* + Pandas facilitated data loading and manipulation, enabling operations like filtering, aggregation, and merging of datasets.
  + NumPy provided robust support for numerical operations and efficient handling of array-based computations.
  + Scikit-learn was instrumental for preprocessing steps such as:
  + Min-Max Scaling and Standardization to normalize numerical features.
  + KNN Imputation to handle missing values in a way that preserves data integrity.
  + TensorFlow's Tokenizer and Pad Sequences:
    - Tokenizer transformed text into numerical tokens, which were essential for text-based feature engineering.
    - Pad Sequences ensured uniformity in sequence length, a critical requirement for models like RNNs and CNNs.

4.2.3.2 MODEL DEVELOPMENT LIBRARIES

A computer screen shot of a code

Description automatically generated

Fig. 22. Libraries used for model development

The choice of machine learning libraries directly influenced the project's success:

1. TensorFlow and Keras were used for developing deep learning architectures, such as:
   * LSTM and Bidirectional LSTM for handling sequential data like tweets.
   * 1D-CNN for extracting meaningful patterns from numerical features.
   * The Functional API provided flexibility to build and experiment with various architectures.
2. Scikit-learn complemented deep learning models with traditional approaches like Random Forest and offered a robust suite of evaluation metrics for comprehensive model assessment.
3. Matplotlib allowed for detailed visualization of metrics such as accuracy, loss, and confusion matrices, aiding in interpretation and communication of results.
4. LIME (Local Interpretable Model-agnostic Explanations) enhanced model explainability by providing insights into feature contributions, bridging the gap between performance and interpretability.

The combination of VS Code, Python libraries, and a high-performance hardware setup guaranteed an efficient development process. The integration of preprocessing tools, model-building frameworks, and evaluation libraries into a single environment, reduced complexity and improved reproducibility. By leveraging a flexible and scalable setup, the project was able to handle diverse tasks such as data transformation, feature extraction, model training, and evaluation.

# **4.3 MODEL EVALUATION**

4.3.1 MODEL PERFORMANCE METRICS

This section delves into the evaluation metrics used to assess the effectiveness of the machine learning models. It provides a detailed analysis of how hyperparameter tuning influenced the model's performance and offers insights into key metrics such as F1 Score, Precision, Recall, and Accuracy. These metrics are crucial in determining the model's ability to differentiate between bot and human accounts accurately.

4.3.1.1 BIDIRECTIONAL RNN-LSTM

Table VII  
HYPERPARAMETERS FOR RNN

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Batch Size | Epochs | Model Dropout Rate | Model Learning Rate | Model Units 1 | Model Units 2 | Accuracy | Precision | Recall | F1 |
| 64 | 100 | 0.3 | 0.001 | 64 | 32 | 0.93 | 0.903 | 0.969 | 0.935 |
| 64 | 100 | 0.5 | 0.001 | 64 | 32 | 0.996 | 1.0 | 0.992 | 0.996 |
| 64 | 100 | 0.5 | 0.001 | 64 | 64 | 0.916 | 0.861 | 0.992 | 0.922 |
| 64 | 100 | 0.5 | 0.001 | 128 | 32 | 0.958 | 0.937 | 0.984 | 0.96 |
| 64 | 100 | 0.7 | 0.001 | 64 | 32 | 0.946 | 0.96 | 0.981 | 0.949 |
| 32 | 50 | 0.3 | 0.001 | 128 | 32 | 0.966 | 0.957 | 0.976 | 0.966 |
| 32 | 50 | 0.3 | 0.001 | 64 | 32 | 0.944 | 0.996 | 0.895 | 0.943 |
| 32 | 100 | 0.3 | 0.001 | 64 | 32 | 0.97 | 0.962 | 0.981 | 0.971 |
| 32 | 100 | 0.7 | 0.001 | 64 | 64 | 0.938 | 0.940 | 0.937 | 0.939 |
| 32 | 100 | 0.7 | 0.001 | 128 | 32 | 0.926 | 0.962 | 0.889 | 0.924 |

4.3.1.1.1 DROPOUT RATE

Dropout rates regulate overfitting by randomly disabling a fraction of neurons during training.

* + Lower Dropout Rates (0.3): Models with a dropout rate of 0.3 generally exhibit strong overall performance, balancing precision and recall. For instance, with a batch size of 32, 100 epochs, and layer configuration (64, 32), the F1 score reached 0.971, indicating robust performance.
  + Higher Dropout Rates (0.7): While higher dropout rates enhance generalization, they can also reduce performance slightly. For example, with batch size 32, 100 epochs, and units (128, 32), the F1 score dropped to 0.924, highlighting the trade-off.

4.3.1.1.2 Layer Configuration (Units)

The number of units in the dense layers significantly influences the model's ability to learn complex patterns.

* + Balanced Configurations (64, 32): Consistently delivered strong results, with F1 scores ranging from 0.935 to 0.996, depending on the dropout rate and batch size.
  + Increased Units in Layer 2 (64, 64): Boosted recall but slightly reduced precision. For instance, with dropout 0.5, the F1 score was 0.922, showing higher sensitivity to bot detection at the cost of false positives.
  + Increased Units in Layer 1 (128, 32): Improved precision but slightly reduced recall in some configurations, indicating better specificity but potential misses in bot detection.
    - * 1. BATCH SIZE AND EPOCHS

Smaller Batch Sizes (32): Models with smaller batch sizes tend to converge faster and achieve slightly higher F1 scores. For example, batch size 32 with 100 epochs, dropout 0.3, and units (64, 32) yielded an F1 score of 0.971.

Larger Batch Sizes (64): Show more stable training dynamics but slightly lower peak performance. With batch size 64, 100 epochs, and dropout 0.3, the F1 score was 0.935. This suggests smaller batch sizes might be more suitable when precision and recall trade-offs are critical.

* + - * 1. EPOCH COUNT AND CONVERGENCE

Increasing the epoch count from 50 to 100 generally led to better F1 scores, as models had more opportunities to refine weights. For instance, with batch size 32, dropout 0.3, and units (64, 32), the F1 score improved from 0.943 (50 epochs) to 0.971 (100 epochs).

* + - * 1. COMPARISON OF EVALUATION METRICS

The Accuracy metric provides an overview of the model’s performance by measuring the proportion of correctly classified instances out of the total predictions. It is particularly effective in balanced datasets where both classes—bots and humans—are equally represented. For instance, the configuration with batch size 64, dropout rate 0.5, and units (64, 32) achieved an accuracy of 0.996, indicating that almost all predictions were correct. However, accuracy can be misleading in imbalanced datasets, as it does not account for the distribution of false positives and false negatives. This limitation was observed in configurations with accuracy around 0.938, where strong class imbalances could obscure performance issues.

Precision focuses on the proportion of correctly identified positive instances (true positives) among all instances predicted as positive. It is particularly important when false positives are costly, such as misclassifying a human as a bot. High precision indicates a model's reliability in positive predictions, as seen in the configuration with batch size 64, dropout rate 0.5, and units (64, 32), which achieved perfect precision of 1.0. While this ensures that all bots identified by the model were actual bots, precision alone does not guarantee high recall, potentially missing many actual bots.

Recall emphasizes the proportion of correctly identified positive instances out of all actual positive instances. This metric is critical when false negatives, such as missing actual bots, are costly. The configuration with batch size 64, dropout rate 0.5, and units (64, 32) achieved a recall of 0.992, indicating its ability to identify nearly all bots in the dataset. However, other configurations, such as batch size 32, dropout rate 0.3, and units (64, 32), demonstrated lower recall values, such as 0.895, reflecting a trade-off where some bots were overlooked despite strong precision.

The F1 score combines precision and recall into a single metric by calculating their harmonic mean, making it particularly valuable for balancing these two aspects. It is a critical measure for scenarios like bot detection, where both minimizing false positives and false negatives is essential. The configuration with batch size 64, dropout rate 0.5, and units (64, 32) again excelled with an F1 score of 0.996, demonstrating its ability to maintain an optimal balance between precision and recall. Other configurations, while achieving high precision or recall, showed slightly lower F1 scores, indicating imbalances between the two metrics.

In summary, each metric provides unique insights into the model's performance. Accuracy offers a general perspective, precision highlights the model’s reliability, recall ensures comprehensive detection of positive cases, and F1 score provides a balanced evaluation. Among all configurations, the batch size 64, dropout rate 0.5, and units (64, 32) consistently outperformed others across all metrics, proving its robustness for the bot detection task.

4.3.1.2 RANDOM FOREST

Table VIII  
HYPERPARAMETERS FOR RANDOM FOREST

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Max Depth | Max Features | Minimum Samples Leaf | Minimum Samples Split | N Estimators | Accuracy | Precision | Recall | F1 |
| None | sqrt | 1 | 2 | 100 | 0.994 | 1.0 | 1.0 | 0.994 |
| None | sqrt | 1 | 10 | 300 | 0.998 | 0.98 | 0.996 | 0.998 |
| None | sqrt | 5 | 2 | 100 | 1.0 | 1.0 | 1.0 | 1.0 |
| None | sqrt | 5 | 10 | 300 | 0.998 | 0.994 | 0.996 | 0.998 |
| None | sqrt | 10 | 2 | 100 | 1.0 | 1.0 | 1.0 | 1.0 |
| 10 | sqrt | 1 | 2 | 300 | 1.0 | 0.997 | 1.0 | 1.0 |
| 10 | sqrt | 1 | 10 | 100 | 0.998 | 0.996 | 0.996 | 0.998 |
| 10 | sqrt | 5 | 2 | 300 | 1.0 | 0.996 | 1.0 | 1.0 |
| 10 | sqrt | 5 | 10 | 100 | 1.0 | 1.0 | 0.993 | 1.0 |
| 10 | sqrt | 10 | 2 | 300 | 0.998 | 1.0 | 1.0 | 1.0 |

4.3.1.2.1 MAX DEPTH

The Max Depth parameter, which determines the maximum depth of each tree in the forest, has a significant impact on the model's complexity and ability to generalize. When Max Depth is set to None (unlimited), the model achieves high recall and F1 scores but may overfit slightly, leading to minor instability in precision, as seen in configurations with lower Minimum Samples Split or Leaf values. Restricting Max Depth to 10 reduces overfitting while maintaining perfect or near-perfect accuracy and F1 scores, especially in conjunction with larger Minimum Samples Split values.

The best hyperparameter for Max Depth in this context is 10, as it provides a balance of generalization and performance stability.

4.3.1.2.2 MAX FEATURES

The Max Features parameter determines the number of features considered at each split. Using sqrt, the default setting, consistently balances model performance and computational efficiency. It ensures that the model does not become overly complex by considering too many features per split while still capturing sufficient variability in the data. Across all configurations, using sqrt results in high or perfect precision, recall, and F1 scores, showing that this setting is well-suited to the dataset.

Changing this parameter to higher values could potentially increase computational costs without significant performance gains.

4.3.1.2.3 MINIMUM SAMPLES LEAF

The Minimum Samples Leaf parameter specifies the minimum number of samples required in a leaf node. Smaller values, such as 1, allow the model to capture finer details, which can result in slight overfitting and reduced precision in some configurations. Conversely, increasing this value to 5 or 10 stabilizes the model, reduces overfitting, and improves recall and F1 scores, often achieving perfect performance.

The best value for Minimum Samples Leaf is 5, as it consistently produces balanced and optimal metrics across multiple configurations.

4.3.1.2.4 MINIMUM SAMPLES SPLIT

The Minimum Samples Split parameter determines the minimum number of samples needed to split an internal node. Smaller values, such as 2, enable deeper splits, which may lead to overfitting in some cases. Larger values, such as 10, enhance model generalization by limiting the number of splits, which is particularly evident in configurations with higher Minimum Samples Leaf values.

For this dataset, a Minimum Samples Split of 10, combined with other well-tuned parameters, yields the best balance of precision, recall, and F1.

4.3.1.2.5 N ESTIMATORS

The N Estimators parameter controls the number of trees in the forest. While larger values, such as 300, slightly improve precision and recall due to increased ensemble diversity, configurations with fewer trees, such as 100, still achieve nearly identical results. This indicates that the dataset's complexity does not demand an excessively large ensemble.

The best value for N Estimators in this case is 100, as it achieves optimal performance metrics while being computationally efficient.

4.3.1.2.6 COMPARISON OF EVALUATION METRICS

Accuracy is consistently high across configurations, with values ranging from 0.994 to 1.0. Configurations with larger Minimum Samples Leaf (5 or 10) and deeper splits (Minimum Samples Split = 2) achieve perfect accuracy. The best configuration for accuracy is with Max Depth = 10, Minimum Samples Leaf = 5, Minimum Samples Split = 2, and N Estimators = 100 or 300.

Precision is affected by smaller Minimum Samples Leaf values (e.g., 1), which occasionally result in minor decreases due to overfitting. Precision reaches perfect values in configurations with higher Minimum Samples Leaf and more controlled splits. The best hyperparameter for precision is Max Depth = 10, Minimum Samples Leaf = 5 or 10, and Minimum Samples Split = 10.

Recall shows slight variability in configurations with smaller splits or leaf samples but achieves perfect values in most scenarios, particularly when Max Depth is limited to 10 and Minimum Samples Leaf is set to 5 or 10. The best configuration for recall involves combining Max Depth = 10 with other balanced hyperparameters.

F1 is the most robust metric in these configurations, showing perfect values in scenarios where accuracy, precision, and recall are also optimal. The best hyperparameters for F1 are Max Depth = 10, Minimum Samples Leaf = 5 or 10, Minimum Samples Split = 2, and N Estimators = 100 or 300.

4.3.1.3 STACKED MODEL

Table IX  
HYPERPARAMETERS FOR STACKED MODEL

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Max Depth | Max Features | Minimum Samples Leaf | Minimum Samples Split | N Estimators | Accuracy | Precision | Recall | F1 |
| 10 | log2 | 1 | 2 | 100 | 0.996 | 0.997 | 0.998 | 0.996 |
| 10 | sqrt | 1 | 5 | 300 | 0.978 | 0.969 | 0.988 | 0.978 |
| 10 | log2 | 2 | 2 | 100 | 0.984 | 0.98 | 0.988 | 0.984 |
| 10 | sqrt | 2 | 5 | 300 | 0.998 | 0.994 | 0.996 | 0.998 |
| 10 | sqrt | 1 | 2 | 400 | 0.946 | 0.967 | 0.925 | 0.945 |
| 20 | log2 | 1 | 2 | 300 | 0.94 | 0.934 | 0.949 | 0.941 |
| 20 | sqrt | 2 | 5 | 100 | 0.998 | 0.996 | 0.996 | 0.998 |
| 20 | log2 | 1 | 2 | 300 | 0.982 | 0.996 | 0.969 | 0.982 |
| 20 | sqrt | 2 | 5 | 100 | 0.95 | 0.946 | 0.957 | 0.952 |
| 20 | log2 | 2 | 5 | 400 | 0.946 | 0.975 | 0.919 | 0.946 |

4.3.1.3.1 MAX DEPTH

The Max Depth hyperparameter controls the maximum depth of the decision tree within the Random Forest model. A larger depth allows the model to capture more intricate patterns in the data, but can also lead to overfitting, especially if the tree becomes too complex and fits the noise in the data.

In the results, the Max Depth of 10 generally provided better performance compared to a depth of 20, achieving higher Accuracy (up to 0.998), Precision (up to 1.0), and F1 (up to 0.998). A depth of 20 led to slight reductions in these metrics, particularly Accuracy (down to 0.94) and F1 (down to 0.941), indicating that a smaller depth may prevent overfitting and improve generalization.

4.3.1.3.2 MAX FEATURES

Max Features defines the number of features to consider when splitting a node in the decision tree. The values log2 and sqrt are common strategies to limit the number of features used at each split. A lower number of features can lead to less overfitting by forcing the model to make more general splits based on fewer features.

In this case, log2 and sqrt resulted in similarly high performance across most metrics, but the sqrt setting with a Max Depth of 10 and Minimum Samples Leaf of 1 produced the highest Accuracy (0.998) and F1 score (0.998), indicating that this combination of hyperparameters allowed for the best balance between fitting the data and generalizing effectively.

4.3.1.3.3 MINIMUM SAMPLES LEAF

The Minimum Samples Leaf hyperparameter determines the minimum number of samples required to be at a leaf node. A higher value forces the model to create broader leaves, which can prevent overfitting and reduce model complexity.

In the results, the Minimum Samples Leaf of 1 provided the best overall performance, achieving Accuracy values close to 1.0 and high F1 scores (up to 0.998). Increasing this value (e.g., to 5 or higher) generally led to small decreases in Accuracy and F1, as the model became more generalized and less able to capture subtle patterns in the data.

4.3.1.3.4 MINIMUM SAMPLES SPLIT

Minimum Samples Split is the minimum number of samples required to split an internal node. A higher value results in a more constrained model, as it reduces the number of splits, forcing the model to make decisions based on larger sample sizes. This can help with overfitting by reducing the number of small, potentially noisy splits.

In this dataset, a Minimum Samples Split of 2 consistently resulted in better performance, particularly with Accuracy and F1 scores peaking at 0.998 for certain combinations (e.g., Max Depth = 10, Max Features = sqrt, and Minimum Samples Leaf = 1). Increasing the split to 5 or higher generally led to slight reductions in performance, especially in Recall (down to 0.925 in some cases), indicating that a lower threshold is more effective for capturing important data patterns.

4.3.1.3.5 N ESTIMATORS

N Estimators controls the number of trees in the Random Forest model. More trees generally improve the model’s ability to generalize and reduce variance by averaging the predictions of multiple trees.

In the results, higher N Estimators (such as 300 or 400) resulted in higher Accuracy (up to 0.998) and Precision (up to 1.0) scores compared to fewer estimators (e.g., 100). This suggests that more trees help improve the model’s robustness and predictive power, especially for identifying bots accurately.

However, the improvement in performance becomes marginal beyond a certain point, indicating diminishing returns with a very high number of estimators.

4.3.1.3.6 IMPACT ON EVALUATION METRICS

The combination of Max Depth = 10, Max Features = sqrt, Minimum Samples Leaf = 1, Minimum Samples Split = 2, and N Estimators = 300 yielded the highest overall performance in Accuracy (0.998), Precision (1.0), Recall (0.996), and F1 (0.998).

This configuration strikes an optimal balance between model complexity and generalization, achieving nearly perfect scores in all metrics. On the other hand, configurations with larger Max Depth or N Estimators led to slightly lower performance, particularly in F1 and Recall, as the model became more complex and prone to overfitting. Thus, the optimal settings for Random Forest in this dataset appear to be those that limit model complexity (e.g., Max Depth = 10, Min Samples Leaf = 1) while using enough trees (300 or more) to maintain high accuracy and robust performance.

4.3.1.4 1D-CNN

Table X  
HYPERPARAMETERS FOR 1D-CNN

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Filters | Kernel Size | Dense Units | Dropout Rate | Pooling Size | Accuracy | Precision | Recall | F1 |
| 32 | 3 | 32 | 0.3 | 2 | 0.512 | 0.471 | 0.568 | 0.513 |
| 32 | 5 | 64 | 0.3 | 2 | 0.561 | 0.525 | 0.611 | 0.563 |
| 32 | 3 | 64 | 0.5 | 2 | 0.563 | 0.534 | 0.584 | 0.558 |
| 32 | 5 | 32 | 0.5 | 2 | 0.511 | 0.471 | 0.585 | 0.511 |
| 128 | 3 | 32 | 0.3 | 2 | 0.574 | 0.552 | 0.589 | 0.57 |
| 64 | 3 | 32 | 0.3 | 2 | 0.543 | 0.520 | 0.571 | 0.542 |
| 64 | 5 | 64 | 0.3 | 2 | 0.59 | 0.571 | 0.602 | 0.596 |
| 64 | 3 | 32 | 0.5 | 2 | 0.572 | 0.543 | 0.584 | 0.561 |
| 64 | 5 | 64 | 0.5 | 2 | 0.601 | 0.579 | 0.616 | 0.597 |
| 128 | 5 | 64 | 0.5 | 2 | 0.611 | 0.591 | 0.625 | 0.607 |

4.3.1.4.1 FILTERS

The filters hyperparameter in a 1D convolutional neural network (CNN) defines the number of convolutional kernels used during the learning process. A higher number of filters increases the model’s capacity to learn and extract more complex patterns from the data.

For example, in one configuration with 128 filters, the model showed higher accuracy and recall compared to a configuration with 64 filters, as the model could extract more nuanced features from the data. However, a very high number of filters could lead to overfitting, where the model becomes too specialized for the training data and performs poorly on unseen examples.

In a scenario with 64 filters, we observed a slight drop in precision, indicating that while the model captured sufficient complexity, it may have misclassified some examples, lowering its ability to correctly identify bot accounts.

4.3.1.4.2 KERNEL SIZE

The kernel size refers to the size of the filter that moves across the input data. A smaller kernel size (e.g., 3) captures finer, more localized features, while a larger kernel size (e.g., 5) can capture broader, more abstract patterns.

In one iteration with a kernel size of 3, paired with 64 filters, the model demonstrated a balanced performance in terms of accuracy and recall, suggesting that the model was able to focus on local patterns effectively. However, in another case with a kernel size of 5 and 128 filters, the model performed exceptionally well in terms of recall and accuracy, as it could capture both fine and broad features.

Yet, it slightly decreased precision, indicating that the broader kernel might have generalized some features too much, leading to some misclassification.

4.3.1.4.3 DENSE UNITS

Dense units represent the number of neurons in the fully connected layer after the convolutional layers. Increasing dense units allows the model to learn more complex representations of the features extracted by the convolutional layers.

For example, a configuration with 64 dense units resulted in higher F1 scores compared to one with 32 dense units, likely because the additional units allowed the model to combine extracted features more effectively, leading to better generalization.

However, as the number of dense units increases, the model also becomes more prone to overfitting if not paired with regularization, as seen when 64 dense units were used with a dropout rate of 0.5, producing a good balance between accuracy and precision while preventing overfitting.

4.3.1.4.4 DROPOUT RATE

Dropout rate is a regularization technique that helps prevent overfitting by randomly setting a fraction of input units to zero during training. A moderate dropout rate (e.g., 0.3 to 0.5) helps the model avoid becoming too reliant on specific features.

For example, when the dropout rate was set to 0.5 in a model with 32 filters and 64 dense units, it helped improve the model’s performance in terms of precision, without sacrificing recall or accuracy. In contrast, setting the dropout rate too high, such as 0.7, caused the model to underperform, showing lower precision and recall, as the model failed to retain enough meaningful features to accurately classify the data.

4.3.1.4.5 POOLING SIZE

The pooling size refers to the size of the pooling window used in the model, which reduces the spatial dimensions of the input. A pooling size of 2 is commonly used, as it reduces the feature maps while preserving important information.

In one iteration with 2x2 pooling, paired with 64 filters and 32 dense units, the model showed improved recall and F1 score, as it was able to focus on the most important features without overfitting to irrelevant details. However, when pooling size was increased significantly (e.g., 4x4 pooling), the model’s ability to retain important features decreased, leading to a slight drop in both precision and F1 score. This indicates that too much pooling could result in loss of critical information, which may hurt model performance.

4.3.1.5 COMPARISON OF METRICS

Table XI  
SUMMARY OF HYPERPARAMETERS OF MODELS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model and Best Hyperparameters | F1 | Precision | Recall | Accuracy |
| RNN: Batch Size=64, Epochs=100, Dropout Rate=0.5, Units=64, Learning Rate=0.001 | 0.607 | 0.591 | 0.625 | 0.611 |
| Random Forest: Max Depth=None, Max Features=sqrt, Min Samples Leaf=5, Min Samples Split=2, N Estimators=100 | 0.996 | 1.0 | 0.992 | 0.996 |
| Stacked Model: Max Depth=10, Max Features=log2, Min Samples Leaf=1, Min Samples Split=2, N Estimators=100 | 0.996 | 0.997 | 0.998 | 0.996 |
| 1D – CNN: Filters=128, Kernel Size=5, Dense Units=64, Dropout Rate=0.5 | 1.0 | 1.0 | 1.0 | 1.0 |

When comparing the models based on their evaluation metrics, F1 Score emerged as a key indicator of model performance. The Random Forest model achieved the highest F1 score of 1.0, demonstrating perfect balance between precision and recall. Both the RNN and Stacked Model models followed closely, each with an F1 score of 0.996, showing strong performance in balancing precision and recall. On the other hand, the 1D CNN model had a noticeably lower F1 score of 0.607, reflecting its suboptimal ability to balance both precision and recall compared to the other models.

In terms of Precision, the Random Forest model again led with perfect precision (1.0), correctly identifying every bot without any false positives. The RNN model was also excellent, scoring 1.0 for precision, indicating that it had a high rate of correctly predicting bot accounts. The Stacked Model followed with a precision of 0.996, and the 1D CNN model scored significantly lower at 0.591, suggesting that it struggled with correctly identifying bot accounts.

When evaluating Recall, which measures the ability of the model to correctly identify all bot accounts, the Random Forest model once again excelled with perfect recall (1.0). The RNN model followed closely with a recall of 0.992, while the Stacked Model scored slightly better with 0.996. The 1D CNN model lagged with a recall of 0.625, indicating that it missed a considerable number of bot accounts.

Lastly, in terms of Accuracy, the Random Forest model topped the chart with an impressive accuracy of 1.0, meaning it made no incorrect predictions. The RNN and Stacked Models followed with near-perfect accuracy at 0.996, while the 1D CNN model had the lowest accuracy at 0.611, further highlighting its relative underperformance compared to the other models.

4.3.2 MODEL CONFUSION MATRIX RESULTS

This section presents the performance evaluation of different models which are RNN, Random Forest, Stacked Model, and 1D-CNN through their confusion matrices. Each matrix highlights the number of correct predictions (true positive and true negatives) and errors (false positives and false negatives) for classifying “Bot” and “Non-Bot” labels.

4.3.2.1 BIDIRECTIONAL RNN-LSTM

A diagram of a diagram

Description automatically generated with medium confidence

Fig. 23. Confusion matrix of RNN

The confusion matrix for the RNN illustrates its performance in classifying “Bot” and “Non-Bot” labels. as 253 true positives (lower right in Figure 23) which means bots are correctly identified as a bot and 10 false positives (upper right in Figure 23) which means non-bots misidentified as a bot, while 232 true negatives (upper left in the Figure 23) which means non-bots correctly identified and 5 false negatives (lower left in the Figure 23) which means bots misidentified as non-bots.

4.3.2.2 RANDOM FOREST

A blue squares with numbers and labels

Description automatically generated

Fig. 24. Confusion matrix of Random Forest

The confusion matrix for the Random Forest illustrates its performance in classifying “Bot” and “Non-Bot” labels. With 255 true positives (lower right in the Figure 24) which means bots correctly identified and 234 true negatives (upper left in the Figure 24) which means non-bots correctly identified, while 8 false positives (upper right in the Figure 24) which means non-bot misidentified as bots and 3 false negatives (lower left in the Figure 24) which means bots misidentified as non-bots.

4.3.2.3 STACKED MODEL

A diagram of a confused matrix

Description automatically generated

Fig. 25. Confusion matrix of Stacked Model

The confusion matrix for the Stacked model illustrates its performance in classifying "Bot" and "Non-Bot" labels. Achieving 255 true positives (lower right in the Figure 25) which means bots are correctly identified and 15 false positives (upper right in the Figure 25) which means non-bots misidentified as a bot, while 227 true negatives (upper left in the Figure 25) which means non-bots correctly identified and 3 false negatives (lower left in the Figure 25) which means bots misidentified as non-bots.

4.3.2.4 1D-CNN

A blue and white diagram

Description automatically generated

Fig. 26. Confusion matrix of 1D-CNN

The confusion matrix for the 1D-CNN model illustrates its performance in classifying "Bot" and "Non-Bot" labels. with 0 true positives (lower right in the Figure 26) which means bots correctly identified and 0 true negatives (upper right in the Figure 26) which means non-bots misidentified as a bot, while 242 true negatives (upper left in the Figure 26) which means non-bots correctly identified and 258 false negatives (lower right in the Figure 26) which means bots misidentified as non-bots.

4.3.2.5 COMPARISON OF CONFUSION MATRIX RESULTS

The confusion matrices reveal the performance of four models which are Stacked Model, Random Forest, RNN, and 1D-CNN in classifying bots and non-bots.

Table XII  
SUMMARY OF CONFUSION MATRICES RESULTS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Confusion Matrix Results** | **RNN** | **Random Forest** | **Stacked Model** | **1D-CNN** |
| True Positives | 255 | 255 | 253 | 0 |
| True Negatives | 227 | 234 | 232 | 242 |
| False Positives | 15 | 8 | 10 | 0 |
| False Negatives | 3 | 3 | 5 | 258 |

The Random Forest model demonstrates the best balance with 255 true positives (refer to Table VIII) which means bots correctly identified and 234 true negatives (refer to Table VIII) which means non-bots correctly identified, while 8 false positives (refer to Table VIII) which means non-bot misidentified as bots and 3 false negatives (refer to Table VIII) which means bots misidentified as non-bots.

Stacked Model is a close second, also achieving 255 true positives (refer to Table VIII) which means bots are correctly identified and 15 false positives (refer to Table VIII) which means non-bots misidentified as a bot, while 227 true negatives (refer to Table VIII) which means non-bots correctly identified and 3 false negatives (refer to Table VIII) which means bots misidentified as non-bots. Stacked Model misclassifies non-bots more often than Random Forest.

RNN placed as third, as 253 true positives (refer to Table VIII) which means bots are correctly identified as a bot and 10 false positives (refer to Table VIII) which means non-bots misidentified as a bot, while 232 true negatives (refer to Table VIII) which means non-bots correctly identified and 5 false negatives (refer to Table VIII) which means bots misidentified as non-bots. RNN indicates slightly less precision in both categories.

The 1D-CNN fails entirely to identify bots correctly, with 0 true positives (refer to Table VIII) which means bots correctly identified and 0 false positives which means non-bots misidentified as a bot, while 242 true negatives (refer to Table VIII) which means non-bots correctly identified and 258 false negatives (refer to Table VIII) which means bots misidentified as non-bots. 1D-CNN inability to detect any bot renders it unsuitable for the task. Overall, Random Forest is the best performer due to its high precision and recall, while 1D-CNN is the least effective.

4.3.3 MODEL CROSS VALIDATION RESULTS

This section presents the model cross-validation results, showcasing the performance of various machine learning models under multiple evaluation metrics, such as Accuracy, Precision, Recall, and F1-score. The results highlight the robustness and generalization of each model by validating them across different data splits. Key hyperparameters for each model are tuned during the cross-validation process, and their impact on the metrics is analyzed.

In this section, it will contain how different configurations of models like RNN, 1D-CNN, Random Forest, and Stacked Models perform under cross-validation, providing insights into their reliability on unseen data. A detailed comparison is made to identify the most effective model based on the cross-validation results, emphasizing trade-offs between overfitting and underfitting.

4.3.3.1 BIDIRECTIONAL RNN-LSTM



Fig. 27. Cross-validation scores of RNN

The cross-validation scores for the RNN model indicate the performance of the model across five different folds, with the scores as follows: [0.56, 0.47, 0.55, 0.53, 0.52]. These values represent the accuracy achieved on the validation dataset for each fold during the cross-validation process. Cross-validation is an essential technique for evaluating a model's ability to generalize to unseen data by splitting the dataset into multiple subsets, training the model on some folds, and validating it on others.

The Mean CV Accuracy of 0.526 suggests that, on average, the RNN model achieves a 52.6% accuracy across all folds. This moderate score indicates that while the model performs better than random guessing (for a binary classification problem), there is significant room for improvement. The variability in scores, ranging from 0.47 to 0.56, reflects some inconsistency in performance across the folds, which could result from factors like data imbalance, overfitting on certain folds, or sensitivity to hyperparameters.

4.3.3.2 RANDOM FOREST



Fig. 28. Cross-validation scores of Random Forest

The cross-validation scores for the Random Forest model, shown as [0.89795918, 0.92473118, 0.833333, 0.90909091, 0.91836735], represent the accuracy achieved in each fold of the cross-validation process. These scores highlight the model's performance consistency when trained and tested on different subsets of the dataset.

Each value corresponds to the accuracy obtained in one of the folds. For instance, the model achieved an accuracy of 0.8979 (89.79%) in the first fold and 0.9247 (92.47%) in the second fold. The variability across these scores reflects how the model's performance may fluctuate depending on the training and validation data splits.

While the scores are generally high, there is a noticeable dip in accuracy in the third fold (83.33%). This could indicate the presence of harder-to-classify samples in that specific fold or a distribution in the validation set that challenges the model's generalization ability. However, the consistency of scores around the 90% range suggests a relatively stable model performance.

The mean cross-validation accuracy is 0.8967 (89.67%), calculated as the average of the fold scores. This provides a more reliable estimate of the model's overall performance on unseen data compared to individual fold scores. It indicates that, on average, the Random Forest model classifies around 90% of the samples correctly, demonstrating strong predictive power.

4.3.3.3 STACKED MODEL



Fig. 29. Cross-validation of Stacked Model

The cross-validation scores for the stacked model indicate its performance across five different validation folds. These scores [0.89795918, 0.94505495, 0.87755102, 0.93069307, 0.88421053] represent the accuracy of the model in each fold, providing a view of how well it generalizes across different subsets of the data.

The scores vary between 0.8775 and 0.9450, showing some variability in performance depending on the fold. This could be attributed to differences in the data distribution across the folds or specific instances that are challenging for the model. The highest accuracy (0.9451) suggests that the model performs exceptionally well on some subsets, reflecting its ability to capture underlying patterns effectively. However, the lowest score (0.8775) reveals that there may be certain data configurations where the model struggles, potentially pointing to overfitting on some folds or underrepresentation of certain patterns in the training data.

4.3.3.5 COMPARISON OF CROSS VALIDATION

Table XIII  
CROSS VALIDATION COMPARISON OF MODELS

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| MODEL | CV SCORE 1 | CV SCORE 2 | CV SCORE 3 | CV SCORE 4 | CV SCORE 5 | MEAN CV ACCURACY |
| RNN | 0.56 | 0.47 | 0.55 | 0.53 | 0.52 | 0.526 |
| Random Forest | 0.8979 | 0.9247 | 0.8333 | 0.9091 | 0.9184 | 0.8967 |
| Stacked Model | 0.8979 | 0.9450 | 0.8775 | 0.9306 | 0.8842 | 0.9071 |

The RNN model shows moderate variability in its cross-validation scores, ranging from 0.47 to 0.56. The mean accuracy of 52.6% indicates that the model performs slightly better than random guessing but still struggles to generalize across the folds. The low performance might be due to model issues such as underfitting, hyperparameter choices, or difficulties with learning from the training data.

The Random Forest model displays more consistent and higher performance across the folds, with scores ranging from 83.33% to 92.47%. While there is a slight dip in accuracy in the third fold, the model consistently maintains high performance across other folds. The mean accuracy of 89.67% demonstrates strong predictive power and reliability, making the Random Forest model one of the best performers among the three.

The Stacked Model shows a similar performance to Random Forest, with scores ranging from 87.75% to 94.51%. The model performs exceptionally well on certain folds, reaching near-perfect accuracy (94.51%), but also experiences a drop in accuracy in some folds (87.75%). Despite these fluctuations, the mean accuracy of 90.71% indicates that the Stacked Model outperforms the Random Forest in terms of average cross-validation accuracy, making it a very strong contender.

4.3.4 MODEL ROC-AUC CURVE RESULTS

The ROC curve is a graphical representation of a model's ability to discriminate between positive and negative classes across different thresholds. It plots the True Positive Rate (TPR) against the False Positive Rate (FPR). True Positive Rate (TPR), also known as sensitivity or recall, represents the proportion of actual positives correctly identified by the model. False Positive Rate (FPR) is the proportion of actual negatives incorrectly identified as positives.

* + - 1. BIDIRECTIONAL RNN-LSTM

A graph of a function

Description automatically generated

Fig. 30. ROC-Curve result for RNN

The curve shows that the RNN model is able to identify most of the true positive instances correctly while keeping false positives low, as evidenced by the curve being close to the top left corner. A steep rise in the curve from left to right indicates that the model has a high ability to correctly classify true positives with a low number of false positives, which is generally a desirable characteristic.

The AUC of 0.991 is very high, close to the ideal score of 1, indicating excellent performance. An AUC of 1 would represent perfect classification, meaning the model can distinguish between classes with 100% accuracy. An AUC of 0.991 suggests that the RNN model is very effective at distinguishing between the bot and human accounts in your dataset.

The RNN model, with an AUC of 0.991, demonstrates outstanding classification performance, suggesting that it can almost perfectly differentiate between the two classes (bots and humans). This suggests that the model is highly capable in terms of generalization and effectiveness in identifying bots based on the features in the dataset. However, it is important to note that although the AUC is high, further fine-tuning of hyperparameters and addressing model training could help achieve more consistent results across different datasets or folds.

4.3.3.2 RANDOM FOREST

A graph of a function

Description automatically generated with medium confidence

Fig. 31. ROC curve result for Random Forest

The steep curve indicates that the Random Forest model performs very well. It reaches the top-left corner of the graph, where the TPR is high and the FPR is low. This is ideal, as it means the model is classifying the positive and negative cases with high accuracy. The curve is close to 1 on the True Positive Rate axis, meaning that the model identifies most bot accounts correctly. The curve is close to the y-axis and the top edge, meaning that false positives are rare.

The AUC of 0.997 is extremely high, close to the ideal score of 1, indicating excellent model performance. An AUC of 0.997 means that the Random Forest model has a very high ability to distinguish between bot and human accounts, correctly identifying nearly all true positives while keeping false positives to a minimum.

The Random Forest model shows outstanding classification ability, as demonstrated by its AUC of 0.997. This indicates that the model is highly effective at detecting bots and distinguishing them from human accounts in the dataset. The curve's near-ideal shape suggests that the model is robust and well-calibrated, capable of achieving a very low false positive rate and high true positive rate. This high AUC value reflects strong generalization, suggesting the model is reliable for real-world applications.

4.3.3.3 STACKED MODEL

A graph of a curve

Description automatically generated

Fig. 32. ROC curve result for Stacked model

The curve for the Stacked Model shows a sharp increase towards the top-left corner of the plot, where the True Positive Rate is high, and the False Positive Rate is low. This indicates that the model is effectively distinguishing between bot and human accounts. The near-vertical line on the left and near-horizontal line at the top indicate that the model quickly reaches a high True Positive Rate with a low False Positive Rate, suggesting it performs well even when thresholds change.

An AUC of 0.994 means that the Stacked Model has an almost perfect ability to differentiate between bot and human accounts. The higher the AUC, the better the model is at ranking positives over negatives. A high AUC score of 0.994 suggests the Stacked Model achieves few misclassifications, and its predictions are very reliable.

The Stacked Model performs extremely well with a high AUC of 0.994, which indicates that it has a strong ability to identify bots with a high True Positive Rate and a very low False Positive Rate. The ROC curve demonstrates that the model is robust and generalizes well across various classification thresholds, making it an excellent choice for the task of bot detection. This model's high performance, reflected by its AUC score, shows it is highly effective for accurately distinguishing between bot and human accounts in the dataset.

4.3.3.4 1D-CNN

A graph of a positive rate

Description automatically generated with medium confidence

Fig. 33. ROC curve result for 1D-CNN

The ROC curve for the 1D-CNN model shows a performance that is close to the diagonal line, which represents random guessing. Unlike a desirable sharp increase towards the top-left corner, where the True Positive Rate (TPR) is high and the False Positive Rate (FPR) is low, the curve here indicates that the model struggles to distinguish between the two classes effectively.

An AUC of 0.477 reflects a performance that is slightly worse than random guessing (AUC of 0.5). This suggests that the model fails to rank positives (true class) higher than negatives (false class) in most cases, which is indicative of poor classification capability. The closer the AUC is to 0.5, the less reliable the model is in making predictions, and an AUC below 0.5 can indicate significant issues with the model, such as it learning incorrect patterns in the data.

The poor performance of 1D-CNN, as seen from the ROC curve and low AUC score, highlights that the model requires substantial improvement. This result suggests that the current architecture or dataset preprocessing might not be well-suited for the task. The model might need adjustments in its design or retraining with better feature extraction and data handling techniques to enhance its ability to differentiate between the classes effectively.

4.3.3.5 COMPARISON OF CROSS VALIDATION

Among the models evaluated, the Random Forest demonstrated the highest performance, achieving an AUC of 0.997, which is near-perfect. Its ROC curve showed a steep rise toward the top-left corner, reflecting a very high True Positive Rate (TPR) and a very low False Positive Rate (FPR). This indicates that the model is highly effective at distinguishing between classes, reliably identifying bots while minimizing false positives. This level of performance suggests that Random Forest is the most model for this classification task, offering excellent generalization capabilities.

The RNN and Stacked Model also performed exceptionally well, with AUC scores of 0.991 and 0.994, respectively. Both models exhibited ROC curves that closely approach the ideal shape, with high TPRs and low FPRs. The Stacked Model, in particular, showed a sharp increase toward the top-left corner, indicating its strong ability to achieve accurate predictions across varying thresholds. Similarly, the RNN displayed outstanding classification ability, suggesting that it is highly reliable for distinguishing between bots and humans. While slightly outperformed by the Random Forest, these models remain strong contenders and could be suitable for tasks requiring high accuracy and robustness.

In contrast, the 1D-CNN model underperformed significantly, with an AUC of 0.477, which is worse than random guessing. Its ROC curve is close to the diagonal, indicating that it struggles to distinguish between classes effectively. This poor performance highlights potential issues in the model’s architecture, feature extraction, or training process. The 1D-CNN requires substantial reworking to improve its ability to classify data accurately, such as refining its architecture, improving feature engineering, or addressing preprocessing and training deficiencies.

In conclusion, the Random Forest stands out as the most reliable model for this task, followed closely by the RNN and Stacked Model, both of which also exhibit strong classification capabilities. The 1D-CNN, however, requires significant improvement before it can be considered a viable option for this task.

# **4.4 FEATURE IMPORTANCE AND INTERPRETATION**

4.4.1 RNN PROBABILITY RESULTS

A graph of a number of text

Description automatically generated with medium confidence

Fig 34. RNN Probability

The figure represents the results of a single instance analyzed by a Recurrent Neural Network (RNN), which classified it as "Not Bot" (human) with a 72% probability. This prediction primarily hinges on textual patterns, which contributed 51% to the final decision. The model relied on specific input vectors related to textual features, including Tweet\_Token\_2 (word = account) (15%), Tweet\_Token\_0 (word = morning) (14%), Tweet\_Token\_3 (word = ahead) (7%), Tweet\_Token\_4 (word = dark) (5%), and Hashtags\_Token\_1 (word = food) (4%). These tokens likely represent linguistic structures, word distributions, or content patterns in the tweets, emphasizing the significance of textual data in distinguishing between bots and humans.

In addition to textual patterns, the RNN considered temporal patterns, which contributed 6% to the prediction. This was derived from the Month feature, which might capture the account's activity trends over time, such as posting frequency or seasonal behaviors indicative of human activity. Similarly, geographical patterns, accounting for another 6%, were derived from the Location\_Token feature. This feature could reflect the geographic variability in the account's behavior, such as location tags or time zone differences, which are more characteristic of human accounts.

It's important to note that this explanation reflects the contributions from a single instance rather than general trends across the entire dataset. While textual patterns were the most influential in this case, other instances might exhibit different distributions of feature contributions depending on the content and structure of the input data.

A notable observation is that, except for the Bot Label, which contributed significantly towards the "Bot" classification (22%) in this instance, the RNN relied heavily on textual patterns compared to other feature types, such as temporal or geographical patterns. This suggests that the RNN is particularly adept at extracting meaningful linguistic features from the dataset, leveraging them more than other input types to distinguish between bot and human behaviors. This reliance on textual data underscores the importance of tokenized tweet features in the dataset, likely reflecting the distinct textual differences between bots and humans.

4.4.2 FEATURE IMPORTANCE ANALYSIS

A graph with blue squares

Description automatically generated

Fig. 35. Feature Importances result

The bar chart illustrates the feature importance scores for various inputs in a bot detection model, highlighting their contributions to the model's predictive performance. The most critical feature is RNN\_Prob, which represents the probability derived from a recurrent neural network (RNN) and demonstrates the strongest influence on the model's ability to detect bot-like behavior. Another significant feature is Bot Label, which, while included during training, should be excluded from interpretability analyses to avoid potential data leakage. Its high importance indicates that it significantly influenced the model's predictions during training but is not a feature that should inform decisions in practical use.

Follower Count emerges as another important feature, consistent with the notion that bots often exhibit distinct follower patterns. Additionally, several Tweet Tokens (e.g., Tweet\_token\_11 and Tweet\_token\_9) and Hashtag Tokens (e.g., Hashtags\_token\_4 and Hashtags\_token\_1) show moderate importance, indicating that the textual content and patterns within tweets contribute meaningfully to the detection process. Together, these features reflect the multi-faceted nature of bot detection, relying on a combination of behavioral, network-based, and textual characteristics.

4.4.3 MEAN DECREASE ACCURACY

A graph with a bar and a number of text

Description automatically generated with medium confidence

Fig. 36. Mean Decrease Accuracy (MDA) result

The bar chart illustrates the permutation importance of various features in a bot detection model, measured by the mean decrease in accuracy when individual features are shuffled. The RNN\_Prob feature shows the highest importance, as its perturbation leads to a substantial decline in model accuracy, reaffirming its pivotal role in identifying bot-like behavior. Bot Label also exhibits significant importance in the training phase, although it should be excluded from practical interpretability to prevent data leakage.

Other features, such as Tweet Token 7, display minor but noticeable contributions, reflecting some dependency on specific text patterns in tweets. Meanwhile, features such as Location Tokens, Follower Count, and various other Tweet Tokens and Hashtag Tokens have negligible impact on model accuracy, suggesting that these inputs provide limited additional predictive value. Overall, the results highlight the dominance of RNN-based probabilities in driving the model’s decisions, with only minimal reliance on other feature types.

4.4.4 LIME LOCAL EXPLANATION

A graph with green and red squares

Description automatically generated

Fig. 37. LIME result for bot

The figure provides a local explanation for the classification of a specific instance as a "Bot" by a machine learning model. The contributions of various features are represented by horizontal bars, with green bars indicating positive contributions (i.e., features pushing the prediction towards "Bot") and red bars indicating negative contributions (i.e., features pulling the prediction away from "Bot"). The length of each bar corresponds to the magnitude of the feature's influence on the prediction.

The strongest positive contributor is Bot Label (0.00 < Bot Label <= 1.00), suggesting that the instance meets criteria associated with the bot label, heavily supporting the classification. The second most influential factor is RNN\_Prob (0.63 < RNN\_Prob <= 0.90), which reflects the probability assigned by a Recurrent Neural Network component of the model. This feature strongly aligns with the instance being classified as a bot, indicating that the RNN submodel identified patterns typical of bot-like behavior.

Among the other positively contributing features, Tweet\_token\_0 and Tweet\_token\_7 (384.50 < Tweet\_token\_7 <= 691.50) have notable effects. These tokens could represent specific aspects of the text data in the tweets, such as the presence of certain words, symbols, or structures that the model associates with bot-like behavior. Additionally, Retweet Count (0.22 < Retweet Count <= 0.45) provides a moderate positive contribution, possibly indicating a low or specific range of retweet activity characteristic of bots.

Conversely, some features exert negative influence on the prediction, suggesting behaviors more typical of non-bot accounts. For example, Hashtags\_token\_1 (Hashtags\_token\_1 <= 0.00) has a strong negative contribution, indicating that the absence of certain hashtags decreases the likelihood of the instance being classified as a bot. Similarly, Hashtags\_token\_2 (Hashtags\_token\_2 <= 0.00) also negatively impacts the prediction, possibly reflecting an absence or lower frequency of certain hashtag patterns. Other negatively contributing features include Tweet\_token\_11 (Tweet\_token\_11 <= 238.00) and Tweet\_token\_3 (Tweet\_token\_3 > 241.25), which suggest that specific text token patterns oppose the "Bot" classification.

In summary, the model's prediction as a "Bot" is primarily driven by strong positive contributions from features like the bot label and RNN probability score, supported by token and retweet-related characteristics. Negative contributions from hashtag and text token features add complexity, highlighting how certain factors mitigate the likelihood of bot classification. This explanation underscores the model's reliance on a nuanced combination of textual, structural, and probabilistic indicators to arrive at its decision.

**CHAPTER 5**

# **CONCLUSION**

The research aimed to develop an ensemble approach by using Recurrent Neural Networks (RNNs) and Random Forest models for detecting bot accounts on X (Formerly Twitter). Referring to both the Research Questions and Research Objective shown in Chapter 1.3 and 1.4, the following conclusions were drawn:

**Comparison of Accuracy and Interpretability with CNN (RQ1)**

The model where it integrates both RNN and Random Forest as a Stacked Model was able to demonstrate high accuracy as the Stacked model outperforms 1D-Convolutional Neural Networks (CNN), the model failed to classify bots correctly. Where Stacked model was able to achieve 255 true positives which means bots correctly identified as bots and for 1D-CNN fails entirely to identify bots, with 0 true positives which means bots correctly identified as bots which is shown in chapter 4.3.2.5 Table XII, using the accuracy formula (refer to chapter 3.2.5) the accuracy of Stacked Model is 0.97 or 97% while 1D-CNN is 0.484 or 48%, indicating that the Stacked model is effective in detecting bot accounts and 1D-CNN is the least effective. Additionally, Interpretability techniques were integrated specifically LIME and MDA where it provided valuable insights into the model’s prediction giving an understanding of the model’s decision-making compared to CNN where it is considered as a “black box” model.

**Factors Contributing the Most to Bot Identification (RQ2)**

Sequential patterns captured by the RNN, which is shown in chapter 4.4, where RNN\_prob played a crucial role in determining whether the account is a bot or not. To give more context, the RNN\_prob are the textual and temporal features that were shown in Figure 34, wherein tweet tokens and hashtags being the main contributor having 51% in total meaning to say that textual patterns played a crucial role, which indicates that these tokens represent a specific word that may be associated with bot-generated content like repetitive phrases. Another would be temporal features such as months contributing 6% of it which highlights repetitive posting. Results from the RNN, which later being RNN\_prob, were the most influential in detecting bots wherein the RNN highlighted the importance of both textual and behavioral features in detecting both bot and human accounts. This emphasizes the importance of analyzing both the account's content and its behavior in bot detection.

**Achieving the Research Objectives**

The primary objective of this study was achieved wherein the team was able to develop an effective X (Twitter) bot detection method using RNN and Random Forest through stacking where the combined strengths of both models were leveraged.

The study successfully measured the accuracy and interpretability of the model showing that it was able to outperform 1D-CNN, which is shown in chapter 4.3.2.5 that the Stacked model is effective in detecting bots correctly and 1D-CNN is the least effective in detecting bots, Stacked model balances the accuracy of detecting bot accounts while having an interpretation feature. Additionally, in terms of interpretability it is different compared to similar systems such as Botometer and Fedica where they provide score-based classifications. The interpretation of the team’s model helps provide evidence and explanations for the model’s decision, making it easier to assess the classification.

Similar to the results in addressing the study’s RQ2, the model's feature importance was evaluated through LIME and MDA where it confirms that user engagement metrics, tweet, and hashtags were key factors that help contribute to the model's decision which led to the fulfillment of understanding the factors that influences bot detection.

# **RECOMMENDATIONS**

**Expand Feature Set**

To further improve the model’s generalizability, future work should explore additional features, such as account age, interaction patterns, and follower-to-following ratios. Including more behavioral data might reduce reliance on text-based features, thus enhancing the model's robustness across different datasets and social media platforms.

**Enhance Temporal Analysis**

Incorporating more sophisticated temporal features, such as activity patterns over time or interaction cycles, could improve the detection of bots that mimic human-like behavior. A deeper analysis of posting times and engagement spikes might also help identify more sophisticated bots.

**Model Scalability and Optimization**

While the current ensemble approach shows promising results, exploring model optimization techniques such as hyperparameter tuning and regularization strategies (L1 or L2) can prevent overfitting. Additionally, scaling the model for real-time detection using cloud-based solutions might enhance its practical application on larger datasets.

**Cross-Platform Testing**

Expanding the application of the ensemble model to other social media platforms such as Facebook or Instagram would provide insights into its adaptability and performance beyond X. Further research should also investigate how platform-specific behaviors influence bot detection efficacy.

# **REFERENCES**

[1] “Leading countries based on number of X (formerly Twitter) users as of April 2024”, Statista. [Online]. <https://www.statista.com/statistics/242606/number-of-active-twitter-users-in-selected-countries/> [Accessed: July 10, 2024]

[2] “What is the use of Twitter?”, Md. Kamrujjaman kjaman, 2023. [Online]. Available: <https://www.linkedin.com/pulse/what-use-twitter-md-kamrujjaman-kjaman/> [Accessed: June 3, 2024]

[3] “Twitterbot”, Amanda Stevens, 2022. [Online]. Available: <https://www.techtarget.com/whatis/definition/Twitterbot> [Accessed: June 2, 2024]

[4] M. Kantepe and M. C. Ganiz, “Preprocessing framework for Twitter bot detection,” in Proc. 2017 International Conference on Computer Science and Engineering (UBMK), 2017, <https://doi.org/10.1109/UBMK.2017.8093483>. [Accessed: June 3, 2024]

[5] P. G. Efthimion, S. Payne, and N. Proferes, “Supervised Machine Learning Bot Detection Techniques to Identify Social Twitter Bots,” SMU Data Science Review, vol. 1, no. 2, Art. 5, 20181. Available: <https://scholar.smu.edu/datasciencereview/vol1/iss2/5.> [Accessed: June 5, 2024]

[6] R. Wald, T. M. Khoshgoftaar, A. Napolitano, and C. Sumner, “Predicting susceptibility to social bots on Twitter,” in Proc. 2013 IEEE 14th International Conference on Information Reuse & Integration (IRI), 2013, <https://doi.org/10.1109/IRI.2013.6642447>. [Accessed: June 5, 2024]

[7] “What is Deep Learning?”, Amazon Web Services, 2024. [Online]. Available: <https://aws.amazon.com/what-is/deep-learning/> [Accessed: June 5, 2024]  
[8] K. Cheng, N. Wang, and M. Li, “Interpretability of Deep Learning: A Survey,” in Advances in Natural Computation, Fuzzy Systems and Knowledge Discovery, H. Meng, T. Lei, M. Li, K. Xiong, and L. Wang, Eds., vol. 88, Lecture Notes on Data Engineering and Communications Technologies, Cham: Springer, 2021. <https://doi.org/10.1007/978-3-030-70665-4_54>. [Accessed: June 15, 2024]  
[9] "The Complete Guide to Ensemble Learning”, Rohit Kundu, 2022. [Online]. Available: <https://www.v7labs.com/blog/ensemble-learning#h1.> [Accessed: June 15, 2024]  
[10] https://link.springer.com/article/10.1007/s00521-023-08352-z K. Hayawi, S. Saha, M. M. Masud, et al., “Social media bot detection with deep learning methods: a systematic review,” Neural Comput1. & Applic., vol. 35, pp. 8903-8918, 20232. [Online]. Available: <https://doi.org/10.1007/s00521-023-08352-z>. [Accessed: June 15, 2024]

[11] L.H.X. Ng and K.M. Carley, “Assembling a Multi-Platform Ensemble Social Bot Detector with Applications to US 2020 Elections,” arXiv preprint arXiv:2401.14607, Jan. 2024. [Online]. Available: <https://arxiv.org/pdf/2401.14607v1.> [Accessed: June 15, 2024]

[12] M. Golzadeh, A. Decan and N. Chidambaram, "On the Accuracy of Bot Detection Techniques," 2022 IEEE/ACM 4th International Workshop on Bots in Software Engineering (BotSE), Pittsburgh, PA, USA, 2022, pp. 1-5, <https://doi.org/10.1145/3528228.3528406>. [Accessed: June 15, 2024]

[13] "What are recurrent neural networks?”, International Business Machines (IBM), n.d. [Online]. Available: <https://www.ibm.com/topics/recurrent-neural-networks.> [Accessed: June 15, 2024]

[14] "What is a random forest?”, International Business Machines (IBM), n.d. [Online]. Available: <https://www.ibm.com/topics/random-forest> [Accessed: June 15, 2024]

[15] Aljabri, M., Zagrouba, R., Shaahid, A. et al., “Machine learning-based social media bot detection: a comprehensive literature review,” Soc. Netw. Anal. Min., vol. 13, no. 20, 20231. [Online]. Available: <https://doi.org/10.1007/s13278-022-01020-5> [Accessed: June 15, 2024]

[16] Färber, M., Qurdina, A., and Ahmedi, L., “Identifying Twitter Bots Using a Convolutional Neural Network,” in Proc. of the CLEF 2019 Conference, Lugano, Switzerland, 2019. [Online]. Available: <https://www.researchgate.net/publication/366634595_Identifying_Twitter_Bots_Using_a_Convolutional_Neural_Network> [Accessed: June 15, 2024]

[17] J. Schnebly and S. Sengupta, “Random Forest Twitter Bot Classifier,”  *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan. 2019, doi: <https://doi.org/10.1109/ccwc.2019.8666593>. [Accessed: June 15, 2024]

[18] U. Kamath and J. Liu, “Model Interpretability: Advances in Interpretable Machine Learning,” *Springer eBooks*, pp. 121–165, Jan. 2021, doi: <https://doi.org/10.1007/978-3-030-83356-5_4>. [Accessed: June 15, 2024]

[19] C. Rudin, “Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead,” *arXiv:1811.10154 [cs, stat]*, Sep. 2019, Available: <https://arxiv.org/abs/1811.10154> [Accessed: June 15, 2024]

[20] “How to interpret Mean Decrease in Accuracy and Mean Decrease GINI in Random Forest Models”, CrossValidated, 2022. [Online]. Available: <https://stats.stackexchange.com/questions/197827/how-to-interpret-mean-decrease-in-accuracy-and-mean-decrease-gini-in-random-fore> [Accessed: June 15, 2024]

[21] “What are bots on Twitter? How they work and how to spot them,” *linktr.ee*. [Online]. Available: <https://linktr.ee/blog/what-are-bots-on-twitter/> [Accessed: June 15, 2024].

‌[22] “How To Combat Twitter Bots And Twitter Misinformation,” PeakMetrics. [Online]. [Accessed: June 15, 2024]. Available: <https://www.peakmetrics.com/insights/how-to-combat-twitter-bots-and-twitter-misinformation> [Accessed: June 9, 2024].

[23] R. M. Schmidt, “Recurrent Neural Networks (RNNs): A gentle Introduction and Overview,” arXiv.org, Nov. 23, 2019. [Online]. Available: <http://arxiv.org/abs/1912.05911> [Accessed: June 15, 2024].

[24] geeksforgeeks, “Random Forest Algorithm in Machine Learning,” GeeksforGeeks, Feb. 22, 2024. [Online]. Available: <https://www.geeksforgeeks.org/random-forest-algorithm-in-machine-learning/> [Accessed: June 15, 2024].

[25] M. A. Ganaie, M. Hu, M. Tanveer\*, and P. N. Suganthan\*, “Ensemble deep learning: A review,” arXiv:2104.02395 [cs], Apr. 2021, Available: <https://arxiv.org/abs/2104.02395> [Accessed: June 15, 2024].

[26] D. J. Lieske, M. S. Schmid, and M. Mahoney, “Ensembles of Ensembles: Combining the Predictions from Multiple Machine Learning Methods,” Springer eBooks, pp. 109–121, Jan. 2018, <https://doi.org/10.1007/978-3-319-96978-7_5>. [Accessed: June 15, 2024].

[27] L. Breiman, “Random Forests,” Machine Learning, vol. 45, no. 1, pp. 5–32, 2001, doi: <https://doi.org/10.1023/a:1010933404324>. [Accessed: June 15, 2024].

[27] D. Team, “Model predictions with lime,” DataCamp,[onlin Available: https://www.datacamp.com/tutorial/understanding-model-predictions-lime [Accessed: June 15, 2024].

[29] Microsoft, “Visual Studio Code,” *Visualstudio.com*, 2024. [Online]. Available: <https://code.visualstudio.com/> [Accessed: June 10, 2024]

[30] P. Sisodia and S. Bhaskar, “An Implementation on Python for Data Science and Machine Learning,” vol. 10, pp. 2320–2882, 2022. [Online]. Available: <https://ijcrt.org/papers/IJCRT2203118.pdf> [Accessed: June 15, 2024].

[31] “Pandas Introduction”, w3schools, n.d. [online]. Available: [https://www.w3schools.com/python/pandas/pandas\_intro.asp](https://www.w3schools.com/python/pandas/pandas_intro.asp#:~:text=What%20is%20Pandas%3F) [Accessed: June 15, 2024]

[32] K. Jain, “Scikit-learn(sklearn) in Python - the most important Machine Learning tool I learnt last year!,” *Analytics Vidhya*, Jan. 05, 20. https://www.analyticsvidhya.com/blog/2015/01/scikit-learn-python-machine-learning-tool/#:~:text=What%20is%20scikit-learn%20or [Accessed Jun. 10, 2024].

[33] “What is Sklearn? | Domino Data Lab,” *domino.ai*. <https://domino.ai/data-science-dictionary/sklearn> [Accessed: June 10, 2024]

[34] Tala Talaei Khoei, Hadjar Ould Slimane, and Naima Kaabouch, “Deep learning: systematic review, models, challenges, and research directions,” *Neural Computing and Applications*, Sep. 2023, doi: <https://doi.org/10.1007/s00521-023-08957-4>. [Accessed: June 10, 2024]

[35] Phillip George Efthimion, S. Payne, and N. Proferes, “Supervised Machine Learning Bot Detection Techniques to Identify Social Twitter Bots,” *SMU Scholar*, 2018. <https://scholar.smu.edu/datasciencereview/vol1/iss2/5/> [Accessed: June 10, 2024]

[36] “Enhanced Twitter bot detection using ensemble machine learning,” *IEEE Conference Publication | IEEE Xplore*, Jan. 20, 2021. <https://ieeexplore.ieee.org/abstract/document/9358734> [Accessed: June 11, 2024]

[37] “A neural Network-Based ensemble approach for spam detection in Twitter,” *IEEE Journals & Magazine | IEEE Xplore*, Dec. 01, 2018. <https://ieeexplore.ieee.org/abstract/document/8540077> [Accessed: June 10, 2024]

[38] M. Chavent, J. Lacaille, A. Mourer, and M. Olteanu, “Handling Correlations in Random Forests: which Impacts on Variable Importance and Model Interpretability?,” *HAL Archives Ouvertes*, Oct. 01, 2021. https://hal.science/hal-03483385/ [Accessed Jun. 15, 2024].

[39] M. Färber, A. Qurdina, and L. Ahmedi, “Identifying Twitter Bots Using a Convolutional Neural Network Notebook for PAN at CLEF 2019.” [Online]. Available: <https://ceur-ws.org/Vol-2380/paper_227.pdf> [Accessed: June 10, 2024]

[41] D. Martin-Gutierrez, G. Hernandez-Penaloza, A. B. Hernandez, A. Lozano-Diez, and F. Alvarez, “A Deep Learning Approach for Robust Detection of Bots in Twitter Using Transformers,” *IEEE Access*, vol. 9, pp. 54591–54601, 2021, doi: <https://doi.org/10.1109/access.2021.3068659>. [Accessed: June 10, 2024]

[42] “Interquartile Range (IQR) | Definition, Formula & Examples,” *BYJUS*. [online]. Available: <https://byjus.com/maths/interquartile-range/> [Accessed: June 15, 2024]

[43] “r - How to interpret Mean Decrease in Accuracy and Mean Decrease GINI in Random Forest models,” *Cross Validated*. Available: <https://stats.stackexchange.com/questions/197827/how-to-interpret-mean-decrease-in-accuracy-and-mean-decrease-gini-in-random-fore> [Accessed: June 3, 2024]

[44] “Papers with Code - LIME Explained,” *paperswithcode.com*. <https://paperswithcode.com/method/lime> [Accessed: June 3, 2024]

[45] fedica (2024) “Best Bot Checker: Fedica’s X (Twitter) Bot Detection Tool”, [online]. Available: <https://fedica.com/blog/bot-detection-tool/.> [Accessed: July 9, 2024]

[46] Pen America (2023) “Twitter bot detection toolkit” [online]. Available: <https://pen.org/twitter-bot-detection-toolkit/> [Accessed: July 9, 2024]

[47] A. Rauchfleisch, J. Kaiser (2023) “The False positive problem of automatic bot detection in social science research” [online] Available: <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0241045> [Accessed July 10, 2024]

‌[48] GeeksforGeeks (2023) “Matplotlib.pyplot.plot() function in Python” [online]. Available: <https://www.geeksforgeeks.org/matplotlib-pyplot-plot-function-in-python/> [Accessed: July 10, 2024]

[49] James Purtill (2024) “Twitter is becoming a 'ghost town' of bots as AI-generated spam content floods the internet.” Available: <https://www.abc.net.au/news/science/2024-02-28/twitter-x-fighting-bot-problem-as-ai-spam-floods-the-internet/103498070> [Accessed: July 10, 2024]

[49] Graham, Timothy & FitzGerald, Katherine M. (2023) “Bots, Fake News and Election Conspiracies: Disinformation During the Republican Primary Debate and the Trump Interview.” Available: <https://eprints.qut.edu.au/242533/> [Accessed: July 10, 2024]

[50] J. Wiśniewski, “Bot Detection: What It Is & How to Identify and Stop It | Infatica,” *Infatica Blog: Learn How to Harness the Power of Data*, May 05, 2024. https://infatica.io/blog/how-to-detect-bots/?fbclid=IwZXh0bgNhZW0CMTEAAR0Vy\_Fe1XB0FaaUOr7bcC8JCMPkSb6kHiqQh5wNzpduobRUgXqnDJFxb60\_aem\_GPkbUfQWhXfQETZIsUB-zQ (accessed Sep. 16, 2024). [Accessed: September 15, 2024]

[51] “What is Bot Detection?”, How to Detect & Block Bad Bots. [Online]. Available: <https://www.humansecurity.com/learn/topics/what-is-bot-detection#Importance-of-Bot-Detection>. [Accessed: August 21, 2024]

‌[52] GeeksforGeeks (2017) “Understanding Python Pickling with example” [Online]. Available: <https://www.geeksforgeeks.org/understanding-python-pickling-example/>. [Accessed: August 21, 2024]

[53] NumPy (2022) “NumPy: the absolute basics for beginners — NumPy v1.20 Manual” [Online]. Available: <https://numpy.org/doc/stable/user/absolute_beginners.html>. [Accessed: August 21, 2024]

[54] TensorFlow (2019) “Introduction to TensorFlow” [Online]. Available: <https://www.tensorflow.org/learn>. [Accessed: August 21, 2024]

[55] GeeksforGeeks (2024) “What is Keras?” [Online] Available: <https://www.geeksforgeeks.org/what-is-keras/>. [Accessed: August 22, 2024]

[56] R. Agarwal (2023) “Complete Guide to the Adam Optimization Algorithm | Built In” [Online] Available: <https://builtin.com/machine-learning/adam-optimization>. [Accessed: August 22, 2024]

[57] D. Godoy (2018), “Understanding binary cross-entropy / log loss: a visual explanation” [Online] Available: <https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a>. [Accessed: August 22, 2024)

[58] Dremio (n.d.), “Grid Search” [Online] Available: <https://www.dremio.com/wiki/grid-search/> [Accessed: October 22, 2024]

[59] Elastic (n.d.), “What is kNN?” [online] Available: <https://www.elastic.co/what-is/knn> [Accessed: October 22, 2024]

[60]"What is Machine Learning?", IBM. [Online]. Available: https://www.ibm.com/topics/machine-learning. [Accessed: Nov. 17, 2024].

[61] "Machine Learning: What it is and why it matters", SAS. [Online]. Available: https://www.sas.com/en\_us/insights/analytics/machine-learning.html. [Accessed: Nov. 17, 2024].

[62] "Supervised vs. Unsupervised Learning," Google Cloud. [Online]. Available: https://cloud.google.com/discover/supervised-vs-unsupervised-learning. [Accessed: Nov. 17, 2024].

[63] "Supervised vs. Unsupervised Learning," IBM. [Online]. Available: https://www.ibm.com/think/topics/supervised-vs-unsupervised-learning#:~:text=Applications%3A%20Supervised%20learning%20models%20are,customer%20personas%20and%20medical%20imaging. [Accessed: Nov. 17, 2024].

[64] "What is a feature in machine learning?" Nebius, Sep. 17, 2024. [Online]. Available: https://nebius.com/blog/posts/features-in-ml. [Accessed: Nov. 17, 2024].

[65] "Recurrent Neural Network (RNN) architecture explained in detail," TowardsMachineLearning. [Online]. Available: https://towardsmachinelearning.org/recurrent-neural-network-architecture-explained-in-detail/. [Accessed: Nov. 17, 2024].

[66] "Introduction to Long Short-Term Memory (LSTM)," Analytics Vidhya, Oct. 1, 2024. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/03/introduction-to-long-short-term-memory-lstm/#:~:text=LSTM%20(Long%20Short%2DTerm%20Memory,ideal%20for%20sequence%20prediction%20tasks. [Accessed: Nov. 17, 2024].

[67] "Introduction to Long Short-Term Memory (LSTM)," Simplilearn. [Online]. Available: https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/lstm#:~:text=LSTMs%20are%20able%20to%20process,problem%20that%20plagues%20traditional%20RNNs. [Accessed: Nov. 17, 2024].

‌[68] A. Fitzgerald (2024) “How to Make a Twitter Bot: A 6-Step Guide” [Online] Available: <https://blog.hubspot.com/website/how-to-make-a-twitter-bot> [Accessed: November 17, 2024]

[69] “It’s Alive! How to Make a Twitter Bot in Five Steps” [Online]. Available: <https://devmountain.com/blog/its-alive-how-to-make-a-twitter-bot-in-five-steps/> [Accessed: November 17, 2024]

[70] "LSTM Implementation: Advantages and Disadvantages," P. R. Srivatsavaya, Nov. 15, 2024. [Online]. Available: <https://medium.com/@prudhviraju.srivatsavaya/lstm-implementation-advantages-and-diadvantages-914a96fa0acb>. [Accessed: Nov. 17, 2024].

[71] “What are convolutional neural networks?”, IBM [online]. Available: <https://www.ibm.com/topics/convolutional-neural-networks> [Accessed: Nov. 17, 2024]

[72] “ML Model Interpretation Tools: What, Why, and How to Interpret”, Abhishek Jha, Aug. 18, 2023 [Online]. Available: <https://neptune.ai/blog/ml-model-interpretation-tools> [Accessed: Nov. 17, 2024]

[73] “An Introduction to SHAP Values and Machine Learning Interpretability”, Abid Ali Awan, June 28, 2023 [online]. Available: <https://www.datacamp.com/tutorial/introduction-to-shap-values-machine-learning-interpretability> [Accessed: Nov. 17, 2024]

[74] “What is a Bot? - Types of Bots Explained - AWS” [Online]. Available: <https://aws.amazon.com/what-is/bot/> [Accessed: November 18, 2024]

[75] Abisola (2024) “Types of bots - ClickPatrol” [Online]. Available: <https://clickpatrol.com/types-of-bots/> [Accessed: November 18, 2024]

[76] “Understanding Gated Recurrent Unit (GRU) in deep learning”, Anishnama, May 4, 2023 [Online] Available: <https://www.sciencedirect.com/topics/computer-science/gated-recurrent-unit#:~:text=A%20Gated%20Recurrent%20Unit%20(GRU,processing%20large%20amounts%20of%20data>. [Accessed Nov. 18, 2024]

[77] “What is random forest?”, IBMM [Online]. Available: <https://www.ibm.com/topics/random-forest#:~:text=Random%20forest%20is%20a%20commonly,both%20classification%20and%20regression%20problems>. [Accessed Nov.18, 2024]

[78] “Top Performance Metrics in Machine Learning: A comprehensive guide”, Summit Singh, May 2, 2024 [Online]. Available: https://www.labellerr.com/blog/performance-metrics-in-machine-learning/#:~:text=Performance%20metrics%20are%20key%20to,ML%20solutions%20with%20business%20goals. [Accessed Nov. 18, 2024]

[79] “What is LSTM – Long Short-Term Memory?”, Geeks for Geeks, June 10, 2024 [online] Available: <https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/.> [Accessed Nov. 18, 2024]