CLUSTERING WEB USERS BY MOUSE MOVEMENT TO DETECT BOTS AND BOTNET ATTACKS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Justin Morgan

March 2021

COMMITTEE MEMBERSHIP

TITLE:	Clustering Web Users By Mouse Movement to Detect Bots and Botnet Attacks
AUTHOR:	Justin Morgan
DATE SUBMITTED:	March 2021
COMMITTEE CHAIR:	Franz Kurfess, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Phoenix (Dongfeng) Fang, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Maria Pantoja, Ph.D. Professor of Computer Science

ABSTRACT

Clustering Web Users By Mouse Movement to Detect Bots and Botnet Attacks

Justin Morgan

The need for website administrators to efficiently and accurately detect the presence of web bots has shown to be a challenging problem. As the sophistication of modern web bots increases, specifically their ability to more closely mimic the behavior of humans, web bot detection schemes are more quickly becoming obsolete by failing to maintain effectiveness [18] [23] [24]. Though machine learning-based detection schemes have been a successful approach to recent implementations, web bots are able to apply similar machine learning tactics to mimic human users, thus bypassing such detection schemes. This work seeks to address the issue of machine learning-based bots bypassing machine learning-based detection schemes, by introducing a novel unsupervised learning approach to cluster users based on behavioral biometrics. The idea is that, by differentiating users based on their behavior [9], for example how they use the mouse or type on the keyboard, information can be provided for website administrators to make more informed decisions on declaring if a user is a human or a bot. This approach is similar to how modern websites require users to login before browsing their website; which in doing so, website administrators can make informed decisions on declaring if a user is a human or a bot. An added benefit of this approach is that it is a human observational proof (HOP); meaning that it will not inconvenience the user (user friction) with human interactive proofs (HIP) such as CAPTCHA, or with login requirements.

ACKNOWLEDGMENTS

Thanks to:

- Dr. Franz Kurfess, for your never-ending support and understanding
- Dr. Phoenix Fang and Dr. Maria Pantoja, for your advice and suggestions as committee members
- My family, for supporting and motivating me
- Dr. John Walker, for your statistical advice
- Dr. Alex Dekhtyar, for your advice on machine learning strategies
- Dr. Theresa Migler, Kurt Voelker, and Kurt Mammen, for recommending my admission into this grad program
- Dr. Chris Lupo, for never being too busy to meet with a student and, for giving me a lecturing opportunity
- Leanne Fiorentino, for always making it happen
- Skyler Ceronio, for being a solid dude
- Nate Jones, for giving me a career-starting internship opportunity
- Andrew Guenther, for uploading this template

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

This thesis paper outlines research of web bot and botnet detection schemes, addresses their strengths and weaknesses, as well as introduces a novel approach to detect bots and botnet attacks. The work in this thesis includes an unsupervised machine learning algorithm to solve the dilemma of detecting sophisticated web bots masqueraded as humans. This section will provide a context of the dilemma and explain the motivation behind the novel approach.

1.1 Background

1.1.1 Bots and Botnets

Web bots, otherwise known as bots, are human-imitating programs created with the intention of reducing the amount of simple and repetitive tasks a human would have to do, while performing these tasks at a speed much faster than a human can perform. Some examples of these tasks may include searching for specific items, with a given criteria, through many items on an ecommerce site such as Amazon. Another example can be the action of downloading lists textual data on a web page that would require the user to do so manually. Search engines use bots, also known as search engine bots, crawlers or spiders, to index, or traverse and access, web pages of a website, storing information seen during these traversals, to later be referenced in a user's search. These search engine bots, as well as link checkers, monitoring bots, and feed fetchers, are examples of "good" bots permitted by website administrators, via a robots.txt file, to traverse a website [23].

Similarly, a botnet is a collection of bots working in unison. These bot collections are hosted by either multiple computers belonging to the botnet administrator, or more commonly, malware-infected computers belonging to victims of a botnet attack. In the latter case, a botnet is controlled by a "botmaster" and used as a means to conduct network and browser attacks such as distributed denial-of-sevice (DDoS) attacks, as well as fraudulent activities such as spam, phishing, identity theft, and information exfiltration [12]. The "net" portion in the "botnet" name derives from method of communication among the botmaster and bots in the botnet.

The [3] report states that a "bad" bot, or "bot" in the context of this paper, presents problems when they scrape, or index and extract, data from websites without permission with the intention of reusing it, for example pricing or inventory levels, to gain a competitive advantage. This, however, does not imply that "good" bots, as described in [3], are always good. A primary reason why these seemingly benign bots may be problematic is their high intensity nature. The presence of "good" bots on a website can skew analytics reports, thus falsely representing the popularity of certain pages of a website. Therefore, being able to separate webisite traffic generated by human users and either type of bots, is essential for making business decisions [3]. Regardless, bots can be used maliciously and irresponsibly, thus introducing a number of problems for the users and administrators of a website [27] [3]. Efforts to detect these bots have proven to be successful [5] [13] [32]. However, due to the increasing sophistication of bots, said detection schemes are often bypassed and deemed obsolete [23] [24] [18].

1.1.2 Supervised and Unsupervised Learning Methods

Machine learning is a widely used term that implies the autonomous improvement of algorithms as they are invoked, hence the learning aspect. Two common methods of these algorithms implement what is known as supervised and unsupervised learning. Although both methods autonomously improve in accuracy and efficiency as they are used, supervised learning requires a labeled dataset as a source of truth whereas unsupervised learning finds patterns and meaning in a non-labeled dataset.

Labels identify a datapoint for a supervised learning implementation to compare against an inputted, non-labeled datapoint. If the labeled datapoints is not representative of a typical inputted, non-labeled datapoint, the supervised learning implementation will lack accuracy. Therefore, a supervised learning implementation is as effective as the robustness of its labeled dataset. An example of a datapoint with a label would be a series of behavioral features that are represented, or labeled, with a username. If an inputted, non-labeled datapoint closely resembles that of a datapoint(s) with a specific user's username label, then there is an high probability that the inputted, non-labeled datapoint originated from that specific user.

An unsupervised learning implementation does not use or require labels in a dataset. Instead, information and meaning of a dataset is interpreted through patterns in the data. Therefore, an unsupervised learning implementation is as effective as the methods used to identify and interpret patterns in a dataset. In the context of this work, features are extracted from a user's mouse movement behavior, and unique patterns of that user are identified and leveraged.

Classification is the identification of a datapoint type, or class, that is represented by a label in the labeled dataset. Classification is an outcome of a supervised learning implementation. **Clustering** is the identification of patterns in dataset and the grouping of the datapoints, by their patterns, in that dataset. Clustering is an outcome of an unsupervised learning implementation.

1.2 Behavioral User Metrics

When using a destop or laptop device, humans use a mouse or touchpad peripheral device to interact and navigate the graphical user interface, as well as a keyboard to input text into the graphical user interface. During this navigation, a human user applies their own unique strategy and style, thus leaving traces or signature traits pertaining to and identifying that user. Styles are unique to for a user's mouse and keyboard use, otherwise known as "keystroke signatures". Feature profiles generated, by human computer interaction-based researchers, are used and quantified in attempt to successfully differentiate and verify users [29]. This thesis work applies a similar strategy of differentiating users. Since users are differentiable based on their behavioral metrics, and since sophisticated web bots are closely mimicking the behavior of human users, differentiating users by clustering the feature profiles of their behavioral metrics should be a means to separate users, bot or not, without any need for labeling or precursory knowledge of the behavior of potential bot users. Users, or the feature metrics pertaining to a user's session, are differentiated into separate clusters that provide website administrators a user-specific network request log that is not IP address-specific. Having this ability enables website administrators to make more informed decisions about the botness of users, or the clusters of similar behavioral metrics.

Human computer interaction-based biometrics researchers verify users by classifying their mouse and keyboard behavior [29]. Additionally, the user provides these biometrics inadvertently, without interfering with the UX experience or causing user friction.

1.3 Motivation for Clustering

As outlined in the background section, a supervised learning implementation is as useful as the robustness of its labeled dataset. Supervised learning-based bot detection implementations that classify bot and human users, otherwise known as a "binary classifier", require labeled datapoints for both bot and human users. Use of an unsupervised learning implementation, specifically clustering, is motivated by the increasing sophistication of bots and the decreasing availability of labeled datasets to represent such bots.

Distil Networks [3] classifies bot sophistication levels as follows:

- **Simple** Connecting from a single, ISP-assigned IP address, this type connects to websites using automated scripts, not browers, and doesn't self-report, or masquerade, as being a real browser.
- **Moderate** Being more complex, this type uses "headless browser" software that simulates browser technology, including the ability to execute JavaScript code.
- **Sophisticated** Producing mouse movements and clicks that fool even sophisticated detection methods, these bad bots mimic human behavior and are the most evasive. They use browser automation software, or malware installed within real browsers, to connect to websites.
- **Advanced Persistent Bots (APBs)** APBs are a combination of moderate and sophisticated bad bots. They tend to cycle through random IP addresses,

enter through anonymous proxies and peer-to-peer networks, and are able to change their user agents. They use a mix of technologies and methods to evade detection while maintaining persistency on target websites.

This thesis presents a novel approach to detect bots classified as either sophisticated or advanced persistent. The objective of this approach is to develop a detection algorithm that is effective irrespective to the level of sophistication a bot masquerades a human user. By clustering the behavioral metrics of users, or visitors of a website, detecting of the "botness" of said users will be similar to the nature of how users are tracked on websites that require login credentials upon entry. If users can be differentiated based solely on their behavior, and not by relying on the client and IP info available in the server logs, than said users can be monitored as if they were logged-in to the website. Having the ability to monitor users at this micro-level, without bias of any IP or proxy anonymity, enables website administrators to make more informed decisions about the web botness of users.

Chapter 2

RELATED WORK

The unsupervised machine learning approach of this thesis research was inspired by a few preexisting implementations of the like. By observing the strengths and weaknesses of these implementations, the outcome of this work is expected to be an improvement and beneficial contribution to the field of web bot and botnet detection research. This section will provide a synopsis and evaluation of the related work that provided inspiration to this work.

2.1 Supervised Learning

There are several supervised learning 1.1.2 bot detection schemes that utilize labeled datasets to train models in their implementation.

2.1.1 Deep Learning with Mouse Behavior Metrics

The detection scheme described in [25] analyzes the mouse movement and operations of a user to decide whether these gestures are consistent with that of a human user or bot user. Images containing the spatial and kinetic information that represents the mouse behavioral metrics are passed into a deep neural network for making the bot or not decision. Representing the models in such a way has enabled the use of a Convolutional Neural Network, or "CNN", to automate feature learning from a user's raw mouse position data.

Rather than following the patterns of previous implementations that fed hand-crafted features into shallow machine learning models, this work transforms every mouse movement sequence into a separate image that contains the spatial and kinetic information about the mouse positions and movements. By doing so, CNN models can be used in this deep learning approach. Related detection schemes that utilize mouse behavioral metrics most likely use statistics of mouse movement sequences as a means to create features. Some of these features, common to this thesis work, include the mean, median, or variance of velocities, accelerations, etc. As an improvement from this trend, the said work contributes to the field by proposing a method of more accurately representing shape information of the user's mouse movement trajectories. This is an underlying motivation to use CNNs, since the complexities of these mouse trajectories can not be represented by using common statistical metrics.

The architecture of this deep learning approach allows for decisions of a user's botness, meaning if they are a bot or not, in a realtime setting. A user's botness is determined

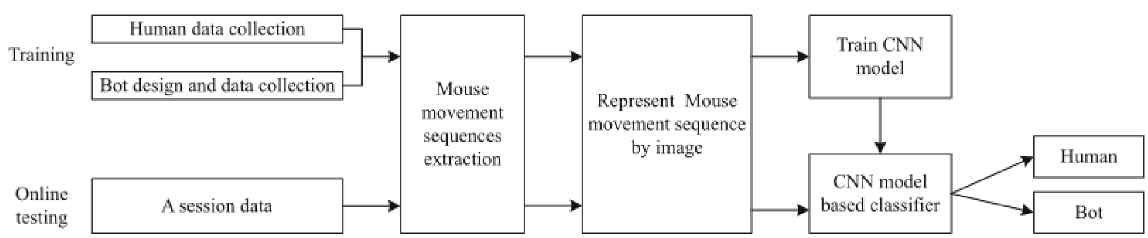

Figure 2.1: Architecture of the deep learning model using CNNs
System design from the research in [25]

by considering the classification results of most mouse movement sequences extracted from the operation data created at the left-most, first stage of the architecture diagram. Therefore, if the classifier depicts inputted mouose movement sequences to be that of a bot, than that particular user, or source of raw mouse position data, will be classified as a bot. A few metrics were created to represent a user's mouse movement behavior:

- **mouse movement sequence**: the records of a series of consecutive mouse moves

- **mouse trajectory**: a time-space curve corresponding to a mouse movement sequence

- **step size**: the distance between two consecutive points

- **event interval**: the time difference between two points

Similarly to this thesis research, the **mouse movement sequence**s are in the format (x_i, y_i, t_i), where $i = 1 \ldots n$ with n being the number of mouse positions in the sequence, x and y are the 2D coordinates of a mouse cursor's position, and t is the time of when that cursor was at the respective position.

Representing a user's mouse movements is done by separating the continuous sequences of mouse events into segments of sequences that correspond to individual mouse movements. Their assumption is that the intervals between mouse events does not exceed 100ms. This closely aligns with the average mouse position polling rate of 125hz [15] [16]. Any mouse movement sequence segments containing less that 15 events were discarded, as these events were said to not contain enough information for bot detection. Following this sequence segmentation stage, a two-step process was implemented to transform the mouse movement sequence segments into images that can be used in the CNN model.

Mapping spatial information into the image was achieved by using shape and distance metrics between consecutive events of a segment. The images were centered by finding the halfway point of the max distance between any two points in the segment. Zoom was considered since the max distances vary among different segments. With this in mind, all mouse positions in the segements were set to a constant 8px diameter. **Mapping kinetic information into the image** was achieved by reflecting the event interval series and step size series. The paper's denotations for these were $(\Delta t_1, \ldots, \Delta t_{n-1})$ and $(\Delta d_1, \ldots, \Delta d_{n-1})$ respectively. In short, they used velocity as the feature to be represented in the images. Every point in the segments were color-coded based on the magnitude of velocity calculated at each point. The colors were on the red, blue, green (RGB) scale, and decided by mapping the delta values to an integer ranging from 0 to 255. Bot users, specifically, were fabricated by 4 different

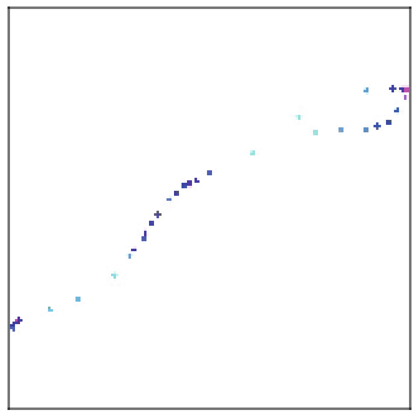

Figure 2.2: Conversion of a mouse sequence segment to an image
Spatial and kinetic information of a users mouse movement behavior are represented in images similar to this one, as shown in [25]. The mouse positions of mouse movement are shown in each sequence segment, a single image, and colors denote the velocity of mouse movement.

types of bot scripts, each generating a unique pattern to their mouse movement trajectories. Figure 2.3 displays images that represent these 4 types. The semi-straight

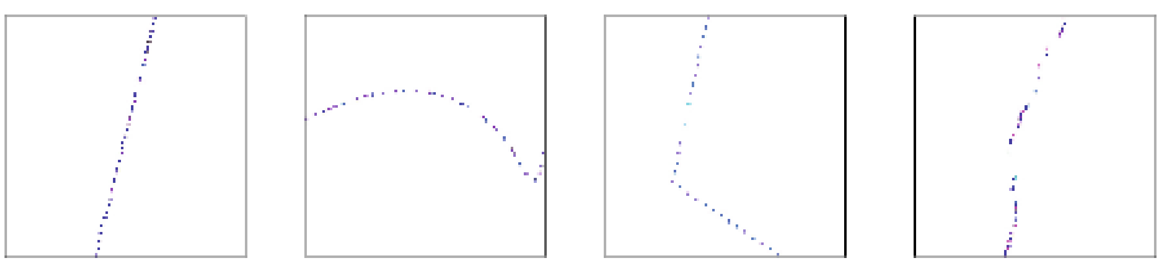

Figure 2.3: Conversion of bot mouse sequence segments to an image
Shown are examples of the 4 different types of bot mouse trajectories: *straight-line*, *curve*, *polyline*, and *semi-straight line*, respectively, as shown in [25]

line was created by adding noise to the points on the straight-line mouse trajectory. Bezier curves were used to make the curve and polyline trajectories. All mouse sequence segments were created with a few different intervals between mouse events. These intervals were added by either a set constant value, uniformly distributed values, or values with a Gaussian distribution. Human datasets were gathered over a 2 month period of recording the activity of several human users. With these datasets, images were generated in the manner previously described, thus allowing for a supervised learning, CNN model approach. Although this paper reflects 90-99% detection accuracy for all 4 types of bots, their results may reflect a testing bias. Since the authors of this paper implemented the bot datasets, their methods of detecting said bots may cause testing biased to their own implementations, thus skewing results.

2.1.2 Graphs and CNNs

BotGraph[20] is one which represented the sitemap, or order of page indexing, of a user with a nodes-and-edges graph. Since these generated graphs are images that represent a number of metrics based on user behavior, not user identification, the research utilized CNNs to predict users types of either bot or human. The BotGraph research concludes that this approach yields about a 95% success rate in detecting bots.

This article begins by introducing key concepts mentioned by a company that specializes in web bot detection, ShieldSquare, the difference of identity-based and behavior-based bot detection. The identify-based method utilizes client-side JavaScript to collect parameters like browser fingerprints; a collection of information about your browser type and version, as well as your operating system, active plugins, timezone, language, screen resolution and various other active settings [14]. Whereas behavior-based methods utilize the number pages visited per session, the duration of time per page visit, and the number of page revisits, etc. This research more closely resembles the behavior-based method but instead represents such metrics via a graph image which, according to the research, contains more unique features per user. The paper continues by introducing a violator blacklist, biometric data validation like scroll and mouse movement method by Distil Networks, as well as the UserAgent variable present in HTTP protocol; an unstable bot-detection method as more advanced bots can falsify their identity by simply hiding or modifying the UserAgent variable. DeepDefense [31] introduced a recurrent neural network (RNN)-based model that takes as input same-shape segment splits from the web server access logs, then encoded the request information in each line of segment to a numerical matrix. The paper states that, while this method is most similar to the outlined research implementation, the inference efficiency of Deep Defense relies too heavily on the length of the same-shape segments; in which BotGraph supposedly proves to be more stable.

The research implementation included BotGraph, a program that generates graph-based representations of users' sitemap traversals. Since these graphs were in image format, the implementation employed convolutional neural network (CNN) inferences to distinguish bots from human user types. The details of BotGraph are as follows:

- request - timestamp, HTTP method, request URI, status, host IP, user_agent, client IP variables
- session - a method if identifying a series of client requests (by bot or human)
- identity - user_agent and client IP variables for the client, and host IP variable for the server
- behavior - request URI and status variables for the access frequency metric per graph node.

The graph can be described as $G = (V, E)$:

- G: a directed graph
- V: set of nodes representing all same-pattern URLs visited, i.e **/page?id=3** is pattern **/page?id=***
- E: set of directed edges, each representing access points, i.e. ***a*** tag elements with same href

Below is a figure of the BotGraph architecture. As you can see, BotGraph runs in a three-step process:

1. Build a sitemap through one of three methods: *active crawling, passive sniffing, self providing*
   (a) *active crawling*: crawling typically starts from website homepage and recursively enters each hyperlink from the current page
   (b) *passive sniffing*: a website's traffic is monitored, learned then used to build the sitemap. This is a less intrusive alternative to active crawling
   (c) *self providing*: the site provides its own sitemap for bot detection. This is the most accurate
2. Map requests listed in server access logs to denote sessions as subgraphs in a sitemap
3. Generate 2-dimensional trace images, translating a bot detection task into an image recognition

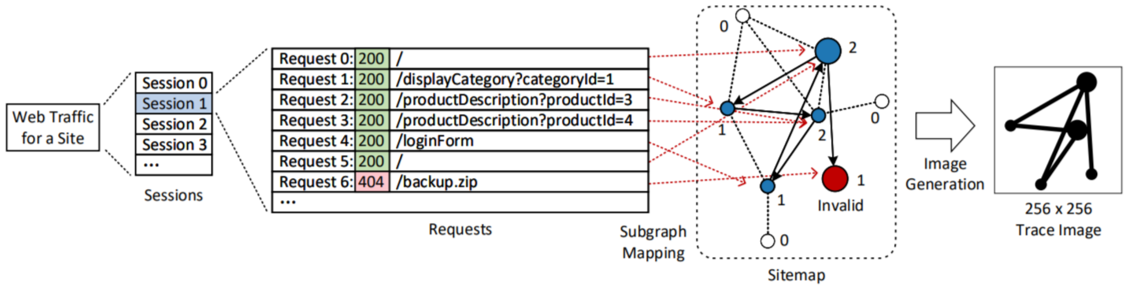

Figure 2.4: Architecture of BotGraph
Diagram from BotGraph [20]

This implementation used a model trained on a data set generated by 30+ professionals that manually tagged web traffic via JavaScript support checking, mouse movement and click tracking, IP reputation, UserAgent blacklisting.

A weakness in the BotGraph method of bot detection is when a user visits a lower number pages per session, i.e. less than 3 visits. This is because bot and human users have too similar browsing behavior, namely their session sitemap traversal, creating near-identical sitemap graphs. However, this research claims that BotGraph is a very effective and efficient method of detecting bots as it achieves about 95% in precision and recall while relying only on the client's behavior and not the client's identity variables. Some needed improvements include an implementation that generates more detailed graph-related features to better describe the characteristics of user sessions, specifically to identify the behavior of web bots.

2.1.3 Training Data Generation and Semi-Supervised Learning

The research described in [19] addresses the common problem with supervised learning-based bot detection schemes. Due to the high traffic of modern search engines, it is infeasible to rely on human judges to generate labeled datasets, used in supervised learning approaches, by manually inspecting the search logs to label bot and human users. On a controlled webserver environment, labeled datasets were created by analyzing the response and activity of CAPTCHA challenges sent to the users. In an effort to enhance the user experience, challenges were sent selectively, either when the webserver is experiencing a high volume of network traffic, or when a user makes a high number of requests in a short amount of time. When presented with a challenge, the user can either disregard the challenge by exiting the session, answer correctly, or answer correctly, thus answering ”no response”, ”correct response”, or ”wrong response”, respectively. About 80% of the received responses were correct. This ac-

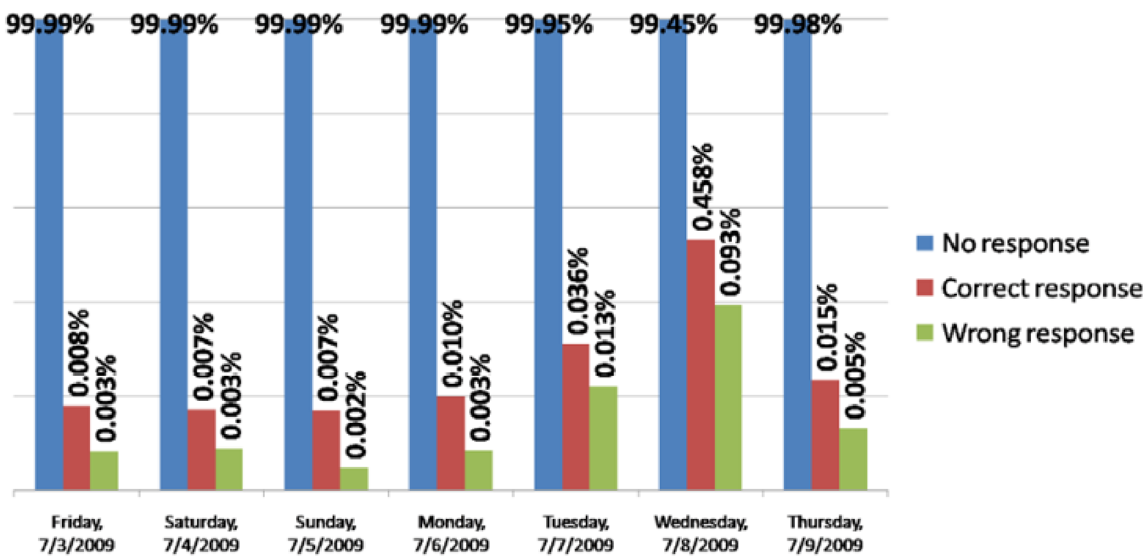

Figure 2.5: User answers to CAPTCHA challenges
The research notes that since the users were selected non-uniformly, most answers to the challenge were ”no response”, as shown in [19]

counted for the majority of the training data with ”human” labels. The remaining ”human” labels were pulled from the large set of ”no response” answers by analyzing heuristics of the user’s number of clicks in a time period, the number of search result pages browsed, as well as information of the user’s IP address. Users were labeled ”bot” if the user’s answer was ”no response”, and the user did not satisfy thresholds of the heuristics previously described. This CAPTCHA challenge-method accounted for the ”0-cost” training data generation method, as described in the research.

From the creation of the labeled training dataset, the following features were extracted from the users:

- **PageTrackedCount**: measures the number of pages that the user browses
- **UserClickCount**: measures the number of mouse clicks on the search result pages
- **AllHitCount**: measures the overall "impressions" that the user receives in addition to the search results
- **UserUniqueIPs**: measures the unique number of IPs a user is using
- **UserUniqueQueries**: measures the unique number of queries issued by a single user in a search session
- **Blacklisting Rules**:
    1. **Form**: triggered when a user includes in the query certain obscure codes that are designed mostly for internal search engine instrumentation purposes that should be unfamiliar to most genuine human users
    2. **IP**: a list of IPs that are publicly identified as Internet spammers and labeled all the traffic from these IPs as "bot"
    3. **Query**: this rule is triggered when the query composition is too complicated to be manually typed in by a human user

The research stated, regarding the **PageTrackedCount**, that bots tend to behave in two extremes. Some bots will only submit queries and not browse any of the result pages (except the first one), ostensibly with the intention to increase the query frequency for certain keywords. The other extreme sees the bots fetch all the result pages for each query, probably trying to reverse engineer the index of the search engine, while genuine human users would probably just browse the first few pages of the query results selectively. For **UserClickCount**, clicks on the search engine results, as well as clicks on advertisements within the results, were included in the click counts. The research noted that the advertisement clicks, though are not distinguished in this work, may include bots specifically targeting ads to click.

With these features, a supervised learning method of bot detection was used. In said research, the C4.5 [22] algorithm was the decision tree used in lieu of a custom implementation, stating "details of the decision tree algorithm are omitted here". By leveraging the autonomously created labeled training dataset, sourced from CAPTCHA response and filtered by following the previously mentioned heuristics, the decision tree algorithm was projected to work well. However, despite the positive projections, the results have shown to be inconsistent from the true data distribution, begging the question of the classification's integrity. This may be a result of the training dataset generation stage, since the "bot" labels are subject to the heuristics determined by domain experts. A useful approach to this dilemma was their statistical method of using numerous unlabeled data, due to uncertainties in the "0-cost" training data generation. Despite the evaluated performance improvement from the tested supervised learning approach, some inconsistencies would need to be addressed in future work. Due to the limited time of one week to present CAPTCHA challenges to users, some search engine bots may have been undetected, thus diminishing the integrity of the training data generation. Therefore, it is unclear if this bot detection scheme can be as useful during more real, long-term scenarios.

2.2 Decision Tree

An implementation introduced in [8] uses the C4.5 decision tree algorithm to determine if a user is a bot or not by characterizing human behaviors from bot behaviors in online services. This approach consists of a two-step system that (1) extracts a user's behavioral metrics via a client-side logger, and (2) determines the botness of a user, based on the retrieved behavioral metrics, with a server-side classifier. A client-side logger, in this context, records a user's behavioral metrics such as mouse movements and keystrokes. Upon gathering this client-side data, the records are sent to the server-side classifier that utilizes the C4.5 decision tree algorithm.

There are two different bot detection approaches that this article seeks to address. Firstly, **content-based filtering**, used by third party clients, needs to be improved upon since they suffer from high false negative rates. This is due to mostly in part of the bot makers finding ways to evade filtering rules set by enforcers. Secondly, **human interactive proofs** (HIP)s, such as CAPTCHA, are problematic for more than one reason. Not only are CAPTCHAs breakable by bot implementors, but the they are intrusive to the user experience. Although there are ways to increase the reliability of HIPs to detect and block bots, such ways would require more interaction from the authentic human users, thus increasing user friction and diminishing the user experience. Human observational proofs (HOB)s are a solution to this dilemma. Unlike the HIPs that are commonly used, HOPs passively observe the actions and behavior of users performing tasks that are meant to be challenging for bots.

A blog site was the testing location for this research. With over 65,000 users, averaging about 800 users simultaneously online, human user sessions were recorded to describe the various behavior characterizations. Additionally, bot profiles were generated under 3 distinct categories (ranked by level of sophistication):

1. **replay**: the most sophisticated type of bot, thus making it the most difficult bot to detect among the other 2 types. This bot records the actions of a human user doing something on the website, such as filling out a form or clicking on a button. Then the bot will impersonate that human user by replaying the recorded actions, exactly following the keystroke and mouse movement styles of that user.

2. **mimic**: since most standard bot detection schemes use keystroke and mouse movement info, or the lackthereof, of the user to determine their botness, bot implementors need to include such details in their mimicking bot schemes. Using OS API calls to generate keystroke and mouse events, mimicking bots are able to bypass older or standard detection schemes. This is possible when a detection scheme only relies on UI events, such as mousemove, keystrokes, etc., which can be triggered by either hardware, such as a mouse and keyboard, or software, such as the OS API calls a mimicking bot would utilize.

3. **inject**: while being the most common type of bot, this type of bot does not interact with the UI. Instead, it triggers the same code that UI events would tirgger, such as an HTTP request that would regularly be triggered by clicking on a UI button. Despite being detectable by most bot detection schemes, injection bots are still able to evade a server's check on HTTP protocols by forging certain fields in the headers, such as Referer, User Agent, and Cookie.

While human user behavioral characterization data was collected from 1,078 signed-in site members during several two-hour monitoring sessions, bot user behavioral characterization data was collected by using existing libraries and frameworks. For the injection bots, any UI features that triggered a POST request were instead triggered by cURL and a predifined string for the request body. An open source Windows program, AutoHotkey, that is designed for automating the Windows GUI and for general scripting was used to configure the mimicking bot. The AutoHotkey script they used was customized to the blog site, mimicking all sorts of human actions, such as moving, clicking, and scrolling the mouse cursor, as well as typing keys. All mimicked actions were masked by entropies, such as random speeds of mouse movements or random delays when typing, to create some sort of human-like characteristics. With the ability to record and replay mouse and keyboard API calls, the Global Mouse and Keyboard Library for Windows was used to configure the replay bot.

Their detection system design consists of a webpage-embeded logger, that records a user's behavior on the client-side, and a server-side classifier that considers the user's data from the client-side and determines if that user is a bot or not. Similarly to

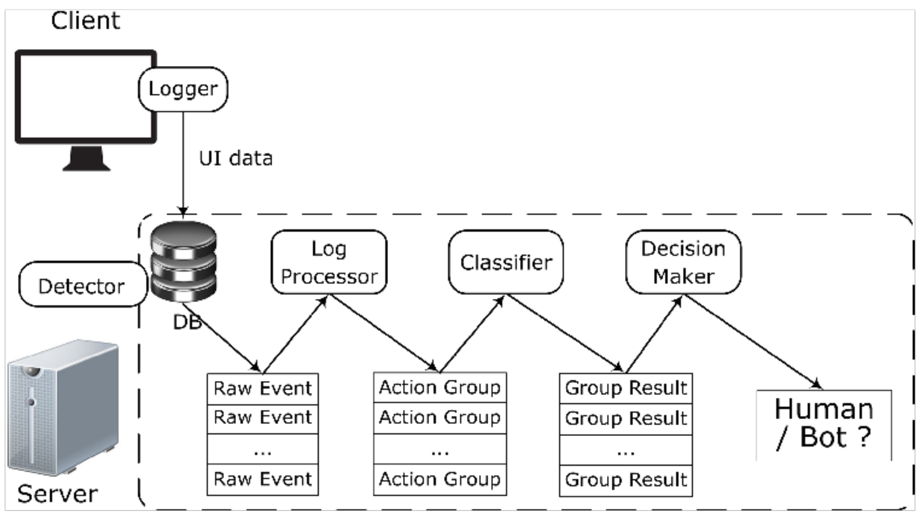

Figure 2.6: Architecture of the client-side Logger and server-side Detector
The client-side Logger and server-side Detector setup above, as shown in [8], closely resemble that of this thesis work's system design.

this thesis research implementation, the client-side Logger is JavaScript code that stores UI event recordings into a buffer and periodically POSTs data to the server. Mouse metrics are recorded at an average of 125hz polling rate. Unbeknownst to the user, the Logger collects, via JavaScript events, five UI events: *key press*, *key release*, *mouse move*, *mouse button press*, and *mouse button release*. When the recordings are POSTed to the server-side Detector, a **log processor** begins to calculate the timing entropy of intervals of the whole raw event data sequence in the POSTed user log. This detects periodic or regular timing of the entire user behavior. The entropy rate can be used to distinguish a human user from a bot user. Human behavior is often more complicated than bot behavior. Such complexity can be measured by the entropy rate. Once the log processor finishes, the C4.5 algorithm is used as a base for the Detector's **classifier**. Reasons for using the C4.5 decision tree algorithm include its efficiency to process large amounts of training data in a short time, the logic is easy to understand and not a black-box, the tree is able to process continuous and discrete values, and lastly, the tree has autonomous tree height constraints to avoid over fitting. Finally, the **decision maker** in the Detector considers all classifications previously made on a user, and holistically makes a decision of its botness.

2.3 Cluster Analysis

Clustering-based bot detection schemes are similar to this thesis work since they do not require knowledge of inputted data to detect bots. However, despite their similarities, there are clustering-based implementations that differ from this research.

2.3.1 Deductions by Similarities

The bot detection implementation [26] uses traffic analysis, unsupervised machine learning, removal of duplicate flows, and similarity between malicious and benign traffic flows to provide insight on the botness of a web user. The research refers to bot web traffic as malicious web traffic and non-malicious web traffic as benign web traffic, which may contain bots that are considered not harmful and are necessary to a system. Search engine bots, for example, would be categorized as benign web traffic. A series of clustering algorithms were tested and used to determine which clusters contained the most flows, leading to insight on the characteristics of a bot. By conducting similarity analysis among these clusters, the work in this research provides a similarity coefficient to describe how malicious traffic data can be distinguished from benign traffic data.

Majority clusters were identified by the number of flows in a cluster. Since the dataset contained mostly malicious flows, the cluster containing the most flows would also be the malicious flows cluster. If this was not the case, than the clustering accuracy was therefore inaccurate. Duplicate flows are flows that share the same values for the selected features, a set of networking-related metrics pertaining to packets traveling to and from the webserver. Similarity between clusters was evaluated using the Jaccard Similarity Coefficient, which was a number ranging from 0 to 1 and the number was the cardinality of the intersection between two clusters, divided by the cardinality of the union between two clusters.

K-means was used to cluster benign and malicious flows, where $k = 2$. Although the number of clusters was known to be two, and the features in these clusters was not biometric data like mouse movement, the method of feature engineering was used as inspiration in this thesis work. Removal of duplicate flows were shown to make the Jaccard Coefficient less computationally expensive. However, a large reduction of duplicate flows within a cluster indicated that the cluster contained bots of a botnet. Detecting anomalies such as this is important to consider when engineering features, a crucial step in the clustering process, in this thesis work.

2.3.2 Outlier Detection

There exists a few methods of outlier detection in bot and human user profiles. Traditional **statistical outlier detection methods** are univariate. Such techniques measure a particular atribute in a data distribution, while examining the degree of that value's outlierness. The parameters, either known or unknown, the number of expected outliers, and the types of expected outliers are the focus of a statistical method. Commons statistical measures, for example mean and standard deviation, can help find outliers in datasets. For **density-based outlier detection methods**, the data points, and their relations to neighbors, are an integral metric to identifying outliers. By definition, a datapoint is considered an outlier if there aren't many datapoints, or neighbors, near it. One common algorithm, local outlier factor, measures the density of a datapoint withing a given k-number of datapoint pertaining to the nearest neighbors of a datapoint Through this approach, outliers are identified as datapoints that have a substantially lower density that its neighbors. A drawback to this approach, as well as other similar approaches, is that it's only capable of measuring the outlierness of a single datapoint, while it's incapable of identifying clusters of outliers. Similarly, **distance-based outlier detection methods** is a method that may apply the local outlier factor. A key benefit to the distance-based method is its ability to detect single datapoint outliers, as well as clusters of outliers.

The implementation [6] uses outlier detection with a particle swarm optimization algorithm, hierarchical particle swarm based clustering, to detect web bots among human users. Web bots are said to be examples of outliers since they are able to index a large number of pages in a short amount of time, contrary to human users. There were two modules included in this work: a clustering module and an outlier detection module. Both modules work simultaneously to label suspected outliers, while the clustering module performs clustering in a hierarchical agglomerative manner.

Meanwhile, the outlier detection removes user profiles, that are labeled as suspecting outliers, from succeeding clusters.

This implementation was tested by using a dataset of user profiles that mimic a bot's behavior, as well as dataset without any ground truth, meaning the dataset contained user profiles without labels of their botness. Three different metrics were used to predict the botness of user profiles: average intra-cluster distance, maximum intra-cluster distance, and the intersection of the average and maximum intra-cluster distances. The results have shown that, by using the average and maximum intra-cluster distance metrics, bots are detectable when they are "significantly different from [a] legitimate web user" [6].

Chapter 3

IMPLEMENTATION

This thesis work utilizes an approach that begins by identifying users, which include human and bot users, based solely on their mouse movement behavior [7]. Specifically, this project presents a user differentiation method based on mouse behavior metrics, such as movement angle, movement velocity, scroll velocity, etc., instead of relying on client IP addresses present in the server logs. Further study could include metrics that are also used in previously implemented web bot detection schemes. But this project will start with just the mouse behavior metrics. The presented approach in this project implies that, upon identifying users based on their mouse use behavior, decisions to declare userX, which is a single cluster, as a human or a bot are reinforced by empirical evidence of web traffic patterns corresponding to userX. This approach could be an improvement to the inaccuracies present in previous supervised learning-based web bot detection schemes. Additionally, this "identify users first, then classify as human or bot" method is similar to the current industry standard of websites requiring all visitors to log into an account for further use of their website; which reinforces decisions to declare account-holderX as a human or a bot, regardless of the IP address of account-holderX. However, this "log-in, then use website" method causes user friction [10], a concept introduced and considered by the latest "covert" versions of reCAPTCHA, that implies the inconveniences a user must experience to prove they are human and not a bot. An example of this could be requiring a user to click/check "I'm not a robot" on older versions of reCAPTCHA. In conclusion, this project presents an unsupervised, clustering method to autonomously identify users, as if they were to log in to an account, providing a means to make more informed decisions of the "web bot-ness" of visitors on a website.

3.1 Objective

The objective of this project is to present a novel approach for website administrators to detect web bots. Supervised learning is a common ML approach to detect web bots. In fact, most ML approaches to robot detection apply supervised learning [17]. This sort of approach consists of training a classifier, i.e. a function mapping an input, which are usually feature vectors describing sessions, to an output, a session's class labels, based on a training dataset, which includes labelled training samples. The ability of the inferred function to determine correct class labels for new, unseen samples is assessed on a test dataset. Many supervised learning techniques demonstrated their efficiency in classification of bots and humans, e.g., decision trees support vector machine, neural networks , and k-Nearest Neighbours. All supervised learning approaches, however, share a common disadvantage, related to a difficulty with preparation of a reliable training dataset, in particular with assigning accurate class labels to sessions of camouflaged robots [23]. In conclusion, since web bots are increasing in sophistication, meaning they are behaving more like humans in terms of mouse and HTTP request behavior [18] [30], obtaining accurate training data that represents such complex web bots has been an issue. This, combined with the anonymity of proxies that scramble the IP addresses of web bots, has motivated this project.

3.2 General Architecture

The general architecture of this implementation is as follows:

1. **data extraction**: The position metrics, timestamp and coordinates, of the user's mouse cursor are collected from the user's computer and sent to the server for analysis. The *Dataset* section 3.3 outlines this stage.

2. **features generation**: From the inputted raw data recorded and sent from the user's machine, features that characterize the raw data, and the user's session pertaining to it, are generated. The *Features Engineering* section 3.4 outlines this stage.

3. **clustering**: Sessions are clustered based on the generated features. The *Clustering* section 3.5 outlines this stage.

4. **classification**: Clusters of sessions are used to determine bottness of users at client IP addresses. The *Classification* section 3.6 outlines this stage, a stage anticipated for future work.

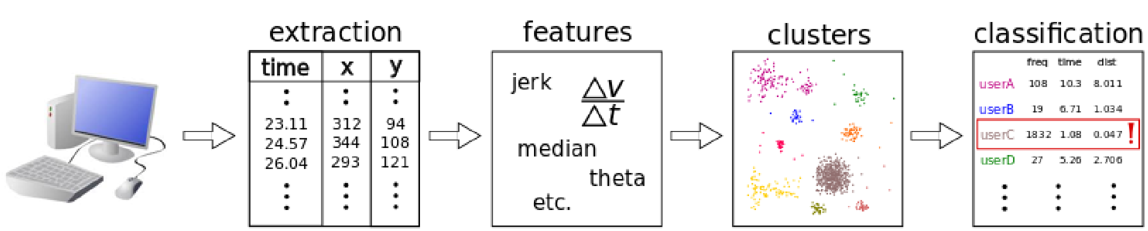

Figure 3.1: Architecture of this bot detection scheme

The *Implementation* chapter 3 described how this architecture is built.

3.3 Dataset

In order to cluster users based on their mouse movement behavior, data of their mouse movement must be obtained. An ideal scenario for this research would be to record the mouse movement of a high number of users, while navigating a specified user interface, thus creating a mouse movement recordings dataset. This would be ideal because there would be more user profiles, as well as knowledge of the graphical user interface used to generate these user profiles. Unfortunately, doing so would require time, people, and resources that are not available during time-frame of this thesis research. However, there are alternatives that maintain the validity of this research, such as using a predefined dataset with mouse behavior-based user profiles.

3.3.1 Balabit Dataset

For testing and analysis purposes, a predefined dataset of users were used in this research. The Balabit Mouse Challenge Dataset [1] was the primary dataset used in this research. This dataset includes timing and positioning information of a web user's mouse pointer. The authors of the dataset advertise that it can be used for authentication and identification purposes. Researchers with focus on creating and evaluating the performance of behavioral biometric algorithms, which in this case draws from the mouse movement metrics of a user, are an intended audience for this publicly accessible dataset. Originating from a data science competition on datapallet.io, the dataset is helpful to researchers and experts in the fields of IT security and datascience.

The competition for which the Balabit dataset originates from included a challenge of protecting users from unauthorized accesses into their accounts. When users would login to their account, located on a remote server, recording their mouse movement behavior was a necessary step in an effort to increase account security. Supposing that the method for which a user moves their mouse was unique to that user, a sort of biometric identifier can be obtained for account user authenticity. If the mouse movement characteristics of a user, in a particular session, does not match the recorded and expected characteristics of the account holder, than that user in the particular session is said to be an unauthorized accessor. In order to apply such a intrusion detection schema, a supervised learning-based model would need to be built and utilized. However, this research does not intend on detecting unauthorized accessors, nor does it intend on using supervised learning in its implementation.

Although the Balabit dataset was intended to be used for creating and evaluating supervised learning-based models, the dataset still contains valuable user profiles and session recordings. These user profiles are defined by a series of session recordings that are labeled with a single user of an account. There are about 100 to 200 sessions recordings, spread over 10 users, with an average of 15 minutes of recording time per session. Session recordings are split into two sets: training and testing. This is for supervised learning uses. For this research, all session files were combined into one set that is to be clustered and analyzed. A session record is a csv file containing these fields [1]:

- **record timestamp**: elapsed time (in seconds) since the start of the session as recorded by the network monitoring device used in the creation of the Balabit dataset
- **client timestamp**: elapsed time (in seconds) since the start of the session as recorded by the RDP client used by each of the 10 users
- **button**: the current condition of the mouse buttons

Table 3.1: Balabit dataset session file format
An example of a session record

record timestamp	client timestamp	button	state	x	y
0.0	0.0	NoButton	Move	399	962
0.157999992371	0.155999999959	NoButton	Move	402	962
0.365999937057	0.248999999953	NoButton	Move	407	962
0.365999937057	0.358000000007	Left	Pressed	430	962
0.476999998093	0.467999999993	Left	Released	474	963
...	...	...	...	...	...

- **state**: additional information about the current state of the mouse
- **x**: the x coordinate (in pixels) of the mouse cursor on the screen
- **y**: the y coordinate (in pixels) of the mouse cursor on the screen

3.3.2 Realtime Dataset

In a realtime environment, where the user-differentiating algorithm is deployed, mouse movement data would come from the browser on the user's computer. By using a JavaScript mousemove event listener, the coordinates of a user's mouse can be determined and recorded while a user is in session. On average, a computers mouse position is polled 125 times per second [15] [16]. This means that if a 10 second user session is recorded, there should be an average of 1,250 mouse position records of any single user browsing a website. Records can be in a csv format:

- **time**: elapsed time (in seconds) since the start of the session as recorded by the user's browser
- **x**: the x coordinate (in pixels) of the mouse cursor on the screen
- **y**: the y coordinate (in pixels) of the mouse cursor on the screen

Table 3.2: Realtime session file format

time	x	y
0.0	241	93
0.008121	278	77
0.015828	291	54
0.026942	302	48
0.037201	317	50
...	...	...

A typical recorded session of a user's mouse movement metrics would look like: These records can be stored on the user's computer, most likely in the browser via a JavaScript variable, then periodically sent to the server for which the website is hosted. From this input, on the server, the web bot and botnet detection scheme will begin. Pseudo code that generalizes the process of obtaining and sending a user's mouse movement metrics looks like:

```
// the list of "times" "x" and "y" values POSTed to the server
var records;

function flushRecords(bufSize)
    // reallocate "bufSize" number of indices for "times" "x" and "y" lists

function postAndFlushRecords(bufSize)
    // POST all recorded "times" "x" and "y" values to the server

function log(elapsedTime, x, y, bufSize)
    // insert "elapsedTime" "x" and "y" values into "records"
    // postAndFlushRecords() if there are "bufSize" number of records

function initBufferTimeout(timeLimit, bufSize)
    // postAndFlushRecords() every "timeLimit" duration
```

```
function init()
    // use the "window" object's "onmousemove" func to log() mouse positions
    // initBufferTimeout() to periodically POST logged "records" to the server

// called once upon every page load
init();
```

The actual code, client_mouse_tracker.js, can be found in the src/ dir on the remote repo [4] of this research.

3.4 Features Engineering

The intrustion detection scheme [7], while also using the Balabit dataset as it was intended to be used, extracted a set of features from the raw mouse position data. Though their work implemented a supervised learning-based binary classifier, the features they extracted were proven to be effective metrics in differentiating and identifying users. Instead of using all 6 elements of a datapoint vector, as outlined in the *Balabit Dataset* section 3.3.1, we elected to only use the client timestamp (t), x position (x), and y position (y) values. These three values construct a triplet, (t_i, x_i, y_i), $i = 1 \ldots n$, where n is the number of recorded mouse positions, or datapoint vectors, in a session file. The three values, time and 2d coordinates, of a mouse position datapoint are all that is needed to generate the mouse movement features in this thesis research. From these three values, or triplets, of a single datapoint vector, in the list of vectors of a session file, the following features were extracted:

- **velocity**: $v_i = \frac{\Delta p_i}{\Delta t_i}$, where $\Delta p_i = |p_{i+1} - p_i|$ and $\Delta t_i = t_{i+1} - t_i$
- **horizontal velocity**: $v_{xi} = \frac{\Delta x_i}{\Delta t_i}$, where $\Delta x_i = |x_{i+1} - x_i|$ and $\Delta t_i = t_{i+1} - t_i$

- **vertical velocity**: $v_{y_i} = \frac{\Delta y_i}{\Delta t_i}$, where $\Delta y_i = |y_{i+1} - y_i|$ and $\Delta t_i = t_{i+1} - t_i$
- **acceleration**: $a_i = \frac{\Delta v_i}{\Delta t_i}$, where $\Delta v_i = |v_{i+1} - v_i|$ and $\Delta t_i = t_{i+1} - t_i$
- **jerk**: $j_i = \frac{\Delta a_i}{\Delta t_i}$, where $\Delta a_i = |a_{i+1} - a_i|$ and $\Delta t_i = t_{i+1} - t_i$
- **theta**: $\Theta_i = \arctan 2(\frac{\Delta y_i}{\Delta x_i})$, where $\Delta y_i = |y_{i+1} - y_i|$ and $\Delta x_i = |x_{i+1} - x_i|$

3.4.1 Realtime Generation

As described in the objective, this implementation is meant to detect web bots and botnet attacks in realtime. At this stage of the detection scheme, a program would need to be run in realtime to compute the 6 features outline above. Initially, a Python program was used to generate these 6 features as described. The program calculated all 1676 session files, from all 10 users, in an average of 7 minutes. By pre-allocating lists of numeric values, and incrementing a counter variable that keeps track of where to insert the next calculated feature value into the list of numeric values, the runtime was reduced from 7 minutes to slightly more than 3 minutes. Further, the entire features generator program was converted from Python to Golang.

By creating Go-routines on each of the 6 features, the average runtime of the Golang program calculating all 1676 sessions files was less than 30 seconds. Figure 3.2 shows the parallelization of calculating the 6 features. All runtimes for realtime feature generation do not include data cleaning and prep. Since the Balabit dataset has many duplicate timestamp values, with different x and y values, a pre-generation step would need to take place to remove erooneous duplicates, as a means to "clean" the raw data input.

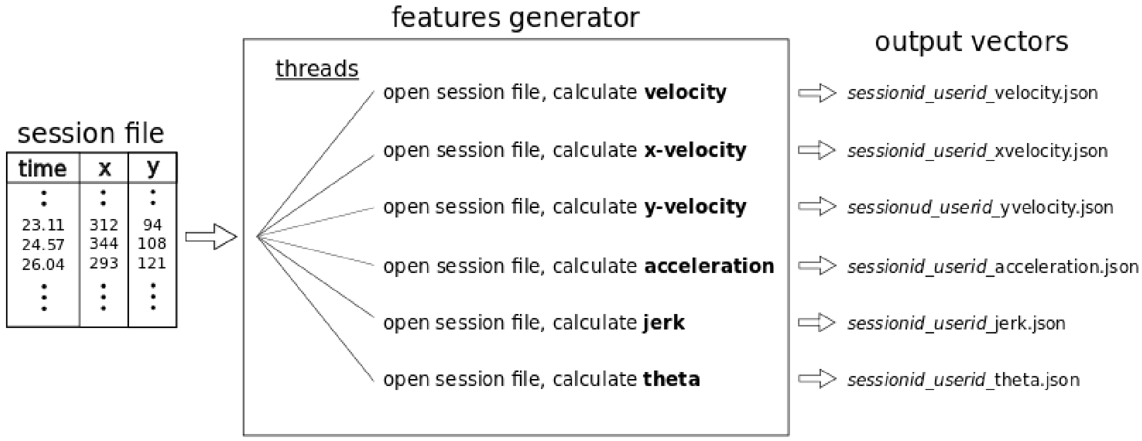

Figure 3.2: Parallelization of the features generator
There are 6 Go routines, or threads as shown in this diagram, that parallelize the features generation stage of the system design. Figure 4.1 displays how this parallelized features generator can be distributed.

3.4.2 Statistics

After the n feature values have been generated, where n is the number of datapoint vectors or triplets in a session, for each of the 6 features, the values would need to be represented with statistical values. The statistical values used for each of the 6 features are **mean**, **median**, **mode**, **interquartile range**, **minimum**, **maximum**, **range**, and **standard deviation**.

It is worth noting that the *mode* and *minimum* values did not appear to be as useful as the other statistical metrics. The minimum values of each of the feature values lists were mostly zero. This is a result of the feature calculations. For example, horizontal velocity could be zero if the x position does not change in two successive datapoint vectors. Formally, $v_{xi} = \frac{\Delta x_i}{\Delta t_i} = 0$ if $\Delta x_i = |x_i - x_{i+1}| = 0$, meaning $x_i = x_{i+1}$. This is one example of why the *maximum* and *range* values were often the same. If $range = |maximum - minimum|$, where $mimimum = 0$, than $range = maxmimum$.

3.5 Clustering

At this point in the web bot and botnet attack detection scheme, a number of metrics are obtained that can be used to differentiate users. For each of the 6 features, 8 statistical metrics are calculated to represent their respective features, totaling to 48 values that can be used for clustering. The purpose of clustering sessions, based on the outlined 48 values, is to differentiate users based solely on their mouse movement behavior. Session files, or lists of mouse position vectors, would be inputted into this clustering algorithm to output clusters of differentiated users. In a realtime environment, the session files would be the mouse position metrics that are periodically inputted from the user's machine, as described in the *Realtime Dataset* section 3.3.2.

During the testing and analysis stages of this research, 2 clustering algorithms were used on the 1676 x 48 inputted session files. With **k-means** clustering, the value for k was set to 10, the number of users present in the 1676 session files. All 48 values pertaining to each session file, were used as the passed-in clustering features. With **hierarchical** clustering, all 48 values were also used.

3.6 Classification

Knowing that a certain user, differentiated by clustering, has a specific number of network requests, hard-coded request thresholds can be enforced. Similar to threshold enforcement on user-based websites, the classification staged of this bot detection scheme will entail blacklisting IP addresses of users of a profile, or cluster, users exceed network request thresholds. However, users would not need to login, as they would on a user-based website, since all users are autonomously differentiated in the clustering stage of this thesis implementation.

Differentiating users by their mouse movement behavior will enable additional classification schemes. The evaluation chapter 4 outlines how accurate the clustering implementation in 3.5 differentiates users so this classification stage will occur. You will see that, at this current state in the thesis research, users profiles are not distinctly clustered for a classification step to occur.

Chapter 4

EVALUATION

4.1 Features Generation

As session files, containing mouse positions of users, are inputted to the back-end server, the features generator is deployed to extract features sets in realtime. Doing so is costly on a server. Since the average number of polled mouse positions is about 125 times per second, and sessions last for numerous minutes, session files contain an average of tens of thousands of lines. Every line of the session files are used and fed through the features generator. Generation of the 6 outputted feature vectors, velocity, horizontal velocity, vertical velocity, acceleration, jerk, and theta, require input of each line of a session file. The computational cost of the features generation step is very high.

Evaluation of the features generator consisted of a few different architectures. For the sake of comparison and additional test data, a Windows and Linux machine were used to run programs written in Python and Golang. With the intention of viewing results and keeping the research process simple, the features generator was originally written in Python. As the program's robustness increased, results were viewed, and runtimes increased, the decision to create the features generator in Golang was influenced by runtime optimization. Since this detection scheme is intended to be run in realtime on a server, and the computation cost of the features generation step is intensive, converting to and parallelizing with Golang seemed like a reasonable investment.

Below are metrics reflecting the total runtimes of each architecture. The Linux machine is run on an Open_SUSE Tumbleweed Linux instance. A Bash terminal was

Table 4.1: Environments used to generate session features

Both systems use a solid-state hard drive. Comparison of the different environments used to generate features from the mouse position data inputted from the user's computer. *Each result shown are averages of 3 separate runtimes*

system	cores	clock_speed	memory	cpu_desc	exec_env
Open_SUSE	4	1.6GHz	8GB-DDR4	8th-Gen-i5	native-Bash
Windows_10	4	3.4GHz	12GB-DDR4	6th-Gen-i7	winpty-Bash

used to execute the Python and Golang programs. However, the Python runtime results varied drastically on the Windows machine. There is one explanation relating to a "wrapper" program, winpty, that had to be used to invoke the Python interpreter. This wrapper-like program acts as an interface between the underlying Windows operating system and the Bash terminal. Enabling users to run Unix programs, winpty must have an overhead cost when using its interface. As you can see, the runtimes are nearly 3 times the amount of the Linux equivalents. Regardless, all runtimes were most often shorter when run on the Linux machine.

There is a bottleneck in the system design. A Bash script was used to execute the features generator program, which takes one session file as input and outputs 6 feature vectors by writing them to files named with the convention *sessionid* _ *userid* _ *featurename*.json in a separate directory.

While this Bash script is sufficient in this research on how to differentiate users by clustering, there is one design change that will significantly reduce the runtime of the system. Instead of using a program that sequentially takes input, like this Bash script, a parallelized "master" program is a necessary design decision. Figure 4.1 shows how distribution of the features generation may occur. Although the Golang metrics reflect a parallelized program, which generates the 6 feature vectors in parallel, a master node can be used to take inputted session files and trigger multiple executions of the features generator. Additionally, session files can be split into several "shards" that would

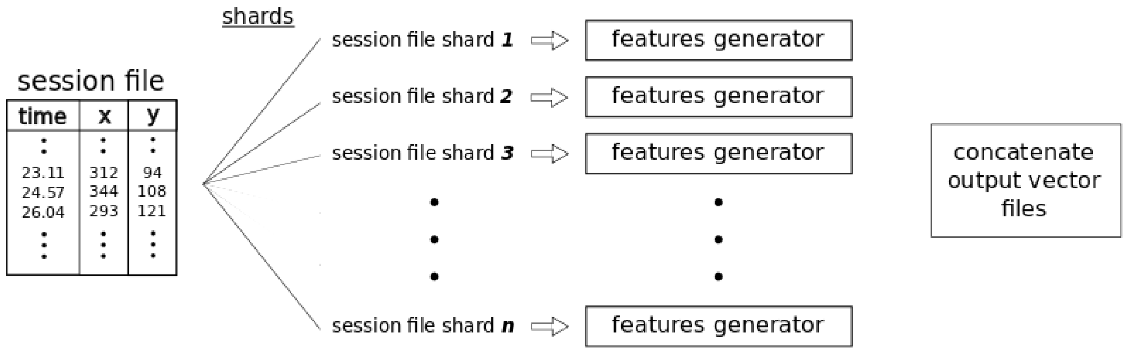

Figure 4.1: Distribution of the parallelized features generator
The figure at 3.2 describes the parallelization of the features generator. This figure displays the sharding and distribution of calculating features from an inputted sessions file.

reduce the required execution time of each features generator executable. Preferably, this parallelization would be distributed across many Linux boxes equipped with the features generator executable.

Table 4.2: Runtimes of different features generation methods
A "list" buffer was used for Python execs, while a "slice" was the equivalent buffer used for Golang execs. Python on the Windows system was run through an API, winpty, that enables Bash terminals to run Unix-like programs. This may account for the significantly slower Python runtimes on Windows

method	**lin_go**	**win_go**	**lin_py**	**win_py**
list_buf	1m5s	1m42s	3m16s	11m16s
prealloc_list_buf	0m36s	2m25s	3m31s	10m52s
file_append	1m13s	1m42s	3m18s	10m55s
go_seq	0m46s	3m4s	–	–

Three different methods, *list or slice buffer*, *preallocated list or slice buffer*, *file append*, and an additional sequential Golang method, were used to compare the different features generator architectures. The list or slice buffer method showed slower runtimes since memory had to be reallocated upon data insertions. By preallocating the memory space, with a given size number, there is no longer a need to reallocate memory for this buffer type. This explains why the preallocated list or slice buffers had a

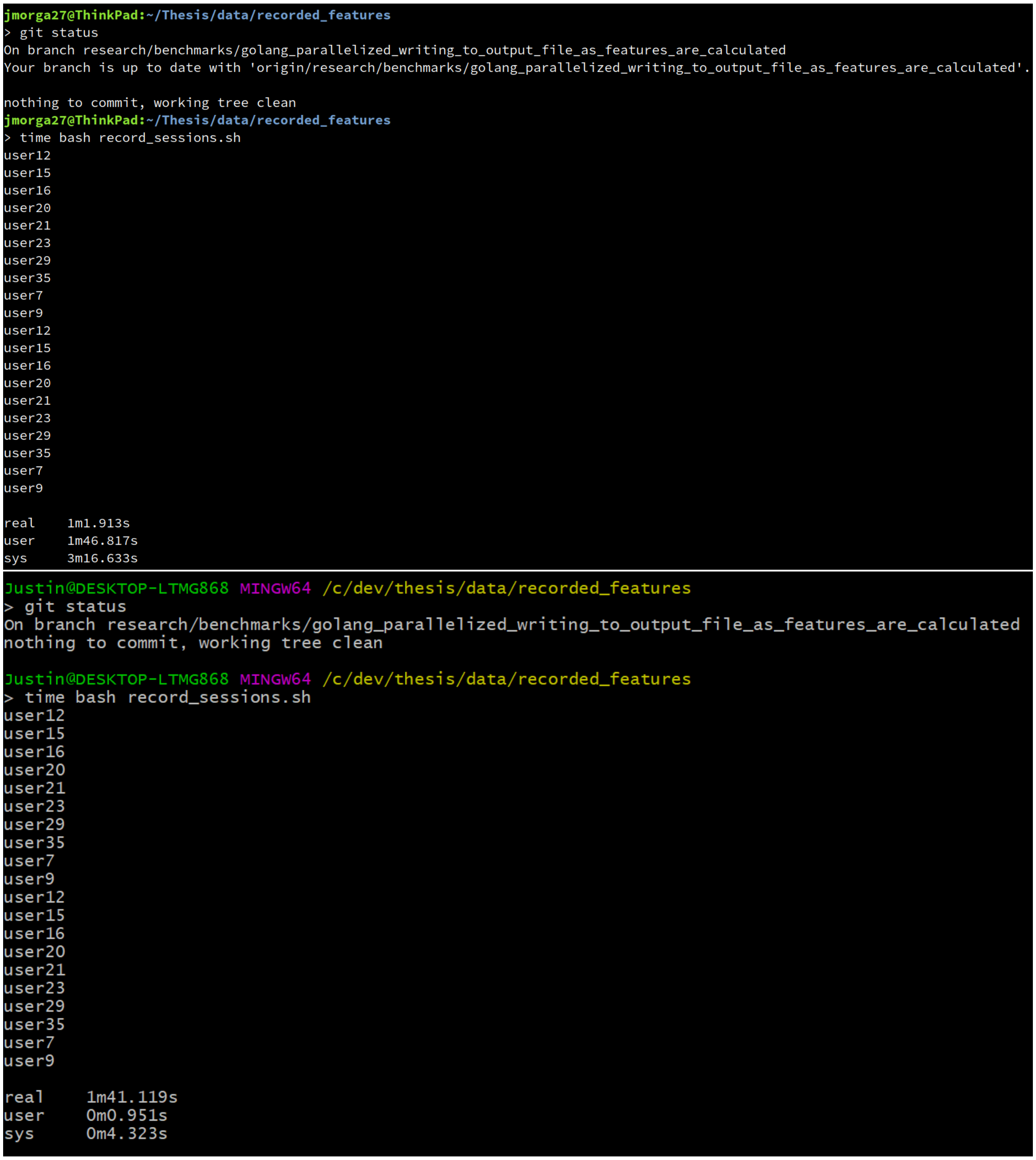

Figure 4.2: Screen shots of the Bash script running the features generator
Both screenshots, Linux (top) and Windows (bottom), include two printouts of the users. This is because the inputted session files consist of a training and testing dataset, per the Balabit dataset [1]

faster average runtime. However, an interesting observation is with runtimes of the Windows preallocated slice buffer and file append methods. Results show that, while using a non-preallocated slice in Golang had the same runtime as the file append method in Golang, the preallocated slice buffer method resulted in slower runtimes. One explaination would be the quality of the solid state hard drives. Thought both systems use a solid state hard drive as their main storage, the Windows system is equipped with an aftermarket SSD. Perhaps the file write speed is quicker with this hard drive, thus decreasing the runtime of the file append method. However, there is uncertainty with this theory.

Of the different methods used, the preallocated slice buffer with Golang method used on the Linux machine showed the best results. This is what prompted the deployment of a non-parallelized, sequential version of the preallocated slice buffer method with Golang was added for comparison. As the results reflect, preallocated slice buffers boast the lowest runtimes in Golang. However, the sequential version of the Golang generator showed poor results on the Windows system. This may be attributed to the slow runtimes of the parallelized preallocated slice buffer method results. An additional explanation could be the method used to obtain the size needed for preallocation. Since the number of lines of the input file is the size used to preallocate the slice buffer, the process of reading the input file may be the culprit. Perhaps there were file system blocks associated with the slower runtimes. Regardless, the optimal architecture for the features generator was the parallelized, preallocated slice buffer method with Golang on the Linux machine.

4.2 Clustering

Though the clustering stage of this detection scheme is to be used in realtime, this thesis research mostly consists of offline testing and evaluation. In the context of this thesis, an optimal outcome for the clustering stage would include clear distinction of clusters that reflect a differentiation between users. By using solely mouse data from the user, and generating features from this data, differentiating users through unlabeled data is the intended outcome in this evaluation.

A statistical software, JMP, was used to perform principal component analysis and generate clusters. Although there were multiple clustering methods available, all of which showed promising results, kmeans was the clustering method used in this research for the multitude of analysis tools and metrics available in the JMP software. Outputted distance and membership metrics reflect the level of effectiveness between different features used with the kmeans method. Since there were 10 users in the inputted dataset [1], and the intention is to differentiate all users, the k value was set to 10 in all clustering samples.

By observing the plotted Eigen vectors, the number of components chosen from PCA was the number of components needed to cumulatively attribute to at least 80% of the variance in the inputted features. In the PCA that observes all 48 features, 7 components were needed to satisfy this 80% threshold. Eigen vectors of each feature were summed and compared. The max sum of all features up to the nth component indicates that feature as having the most "weight" or influence on the variation in the data. A challenging aspect of PCA was deciding how many components to sum to. 7 components were chosen here. But that is only because 7 components were needed to satisfy a self-suggested threshold of 80% that seemed reasonable. There is a need to analyze further to reinforce this approach.

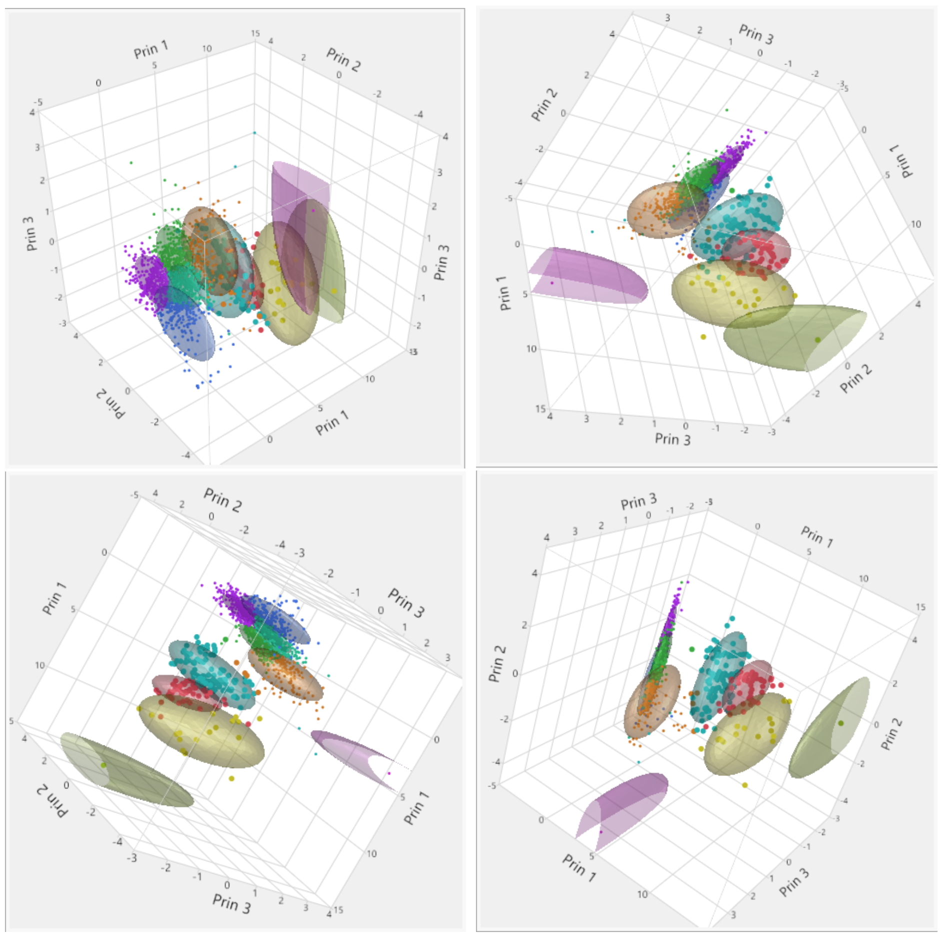

Figure 4.3: 3D illustrations of clustering

Shown here are 4 images of the same (median) clustering results, using the top 3 principal components.

Table 4.3: Results from PCA (all)

Shown are the 8 features used as a result of the principal component analysis on all 48 features. Per the right-most sums column, the left-most column contains features that account for most of the variation of the data. Note how the theta features have such high values, despite them having the lowest original values. This may be a result of data standardization in JMP

feature	PC1	PC2	PC3	PC4	PC5	PC6	PC7	sum
vel-median	0.07	0.29	-0.09	-0.02	0.04	0.07	0.06	**43.72**
vel-iqr	0.08	0.19	-0.28	-0.09	0.05	0.21	0.18	**35.59**
yvel-median	0.07	0.28	0.04	0.07	0.06	-0.01	0.19	**70.60**
yvel-mode	0.01	0.07	-0.03	0.05	-0.03	-0.01	0.28	**33.75**
yvel-iqr	0.08	0.22	-0.12	0.01	0.12	0.18	0.30	**78.26**
accl-median	0.06	0.31	0.07	-0.01	0.07	-0.02	-0.13	**35.89**
theta-mean	0.01	0.02	0.30	0.19	0.23	0.16	0.38	**128.40**
theta-median	0.01	0.03	0.28	0.20	0.20	0.14	0.40	**126.28**
theta-iqr	-0.03	-0.07	0.28	0.00	0.30	0.29	-0.07	**69.52**
theta-stdev	-0.04	-0.07	0.30	0.00	0.27	0.24	-0.18	**52.62**

Table 4.4: Eigen values in PCA (all)

Principal component analysis on all 48 features

PrinComp	01	02	03	04	05	06	07	08
Eigen	14.47	8.94	3.76	3.09	2.71	2.00	1.91	1.22
Per	30.8	19.0	8.0	6.6	5.8	4.3	4.1	2.6
Cum	30.1	49.8	57.8	64.4	70.2	74.4	78.5	81.1

Once the principal components were chosen, the clustering stage begins. Note that the PCA step is not going to be a part of the detection scheme. The only reason why it was applied here was to see which mouse movement features are most effective in user differentiation by clustering. Despite the PCA-generated features, there were a number of clustering versions used for comparison. The **mean**, **median**, **mode**, **standard deviation**, and the **interquartile range** of each of the 6 movement features, *velocity*, *horizontal velocity*, *vertical velocity*, *acceleration*, *jerk*, and *theta*, were used to build clusters. Of all 48 feature vectors that are to be used in clustering, 6 features, one movement feature per statistical value, were chosen as parameters to cluster.

Table 4.5: Clustering distribution results

features_set	pop_stdev	pop_range	bias	dist_mean	dist_stdev
iqr	152.42	14-445	3.628	1.794	2.012
mean	294.60	1-732	2.647	2.037	1.846
median	174.16	1-478	3.475	1.369	1.374
mode	477.47	1-1523	2.360	2.480	3.073
stdev	329.97	1-837	1.054	2.098	1.892
pca_8	215.36	2-590	1.272	5.859	4.397
pca_10	139.14	2-396	1.400	3.862	3.523

For example, the "mean" features set used as parameters for clustering were **velocity**-*mean*, **horizontal velocity**-*mean*, **vertical velocity**-*mean*, **acceleration**-*mean*, **jerk**-*mean*, and **theta**-*mean*. This pattern continued through all 6 of the movement features, over all 8 of the statistical features. Hence the 48 features to choose from.

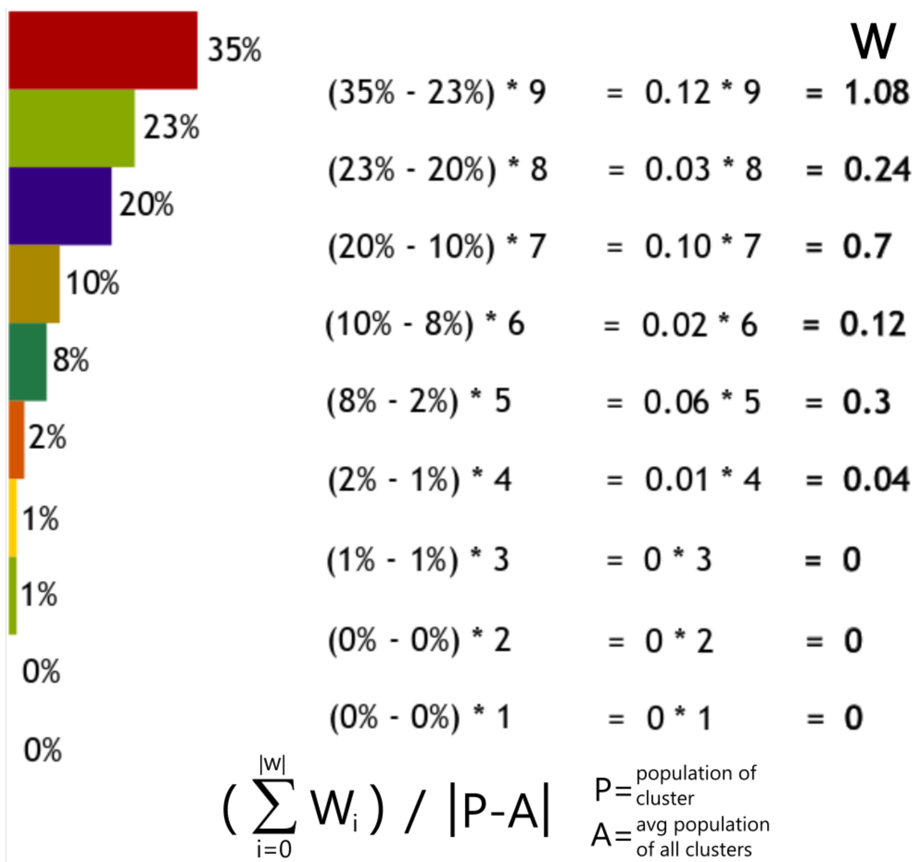

Figure 4.4: User frequency bias metric

Table 4.6: User differentiation accuracy

Accuracy of the different features_sets in clustering to differentiate users

features_set	**iqr**	**mean**	**median**	**mode**	**pca_8**	**pca_10**	**stdev**	total
user7	34	0	40	3	52	80	96	158
user9	43	0	48	7	12	43	0	130
user12	93	2	37	6	103	88	3	240
user15	32	3	59	2	28	65	150	253
user16	67	3	1	1	4	53	3	201
user20	18	51	4	1	0	2	6	114
user21	0	4	52	0	3	0	1	121
user23	40	76	11	136	2	10	1	142
user29	61	1	0	0	10	58	1	124
user35	7	3	84	2	85	15	2	193
accuracy	**2.361**	**1.074**	**1.992**	**1.088**	**1.541**	**2.358**	**1.314**	–

Table 4.6 shows how clusters were assigned to users and clustering accuracy was calculated. Users were differentiated by iterating the frequency rankings of all 10 clusters and assigning clusters to users. The frequencies of the users in their respective clusters are listed in the columns, with the far-right *total* column containing the user's total number of session files clustered. Each feature_set cluster's accuracy is calculated by adding the percentages of *total* users in each cluster. Feature_set *iqr* clusters had the highest accuracy, whereas the *mean* clusters had the lowest accuracy.

The user frequency bias metric describes the distribution of users in a cluster. A high value indicates that the cluster is biased to a certain user(s). A low value indicates that the cluster is not biased to any user(s) and is evenly spread. This metric accounts for variance and weights applied to the hierarchy of frequencies.

The user frequency bias metric can be expressed as

$$\frac{\sum_{i=1}^{n-1} |p_i - p_{i+1}| * (n - i)}{|P - A|}$$

where n is the number of users in a cluster, p_i is the percentage of the ith user in comparison to the total cluster population, P is the total cluster population, and A is the average population for all clusters.

With the user frequency bias metric and the distance values previously outlined, a conclusion of this clustering approach will outline the validity and meaning of this thesis work.

Chapter 5

CONCLUSION

5.1 Contributions

This thesis research includes the following contributions:

1) A parallelized and scalable Golang implementation that extracts movement features from inputted position data in realtime. The format of the position data is in a standard time-position format. Despite the 2 dimensions of the intended input data, adding an additional dimension should not pose any issues. Since the program outputs vectors of the 1st, 2nd, and 3rd derivatives of a postion dataset, there may be other applications that require parallelized features generation in realtime.

2) A membership bias metric that measures how well a clustering method differentiates users. Measuring variance is not sufficient when valuating clusters in this context. Though it's optimal to have high variance between members of a cluster and the member for which is represented by that cluster, variance would not be a simple method of measuring the disparity between members of a cluster in the intended way. The membership bias metric accounts for the variance among irrelevant, non-represented members of a cluster.

3) Several mouse movement features sets that have shown to be useful to differentiate users. The majority of related works that utilize a users mouse movement behavior require labeling of data. This is to say that, because unsupervised learning methods are not common with mouse movement metrics, providing preliminary results about which features to use for clustering will be beneficial.

4) An unsupervised learning approach to web bot detection. Prior research indicates that supervised learning methods, though still the popular methods, are not able to detect the growing number of highly sophisticated web bots. A simple explanation of this is that, since supervised learning methods require labeled datasets, and that trained models are only as effective as the robustness of the training data, supervised learning methods can not maintaing the cutting edge in bot detection. This thesis attempts to resolve this ongoing issue by supplying researchers with a method detecting bots by solely looking at the mouse movement behavior of a user, regardless of their bot or humanness.

5) A web bot detection scheme that does not require users to login or prove their humanness. Most websites require users to have an account, or pass a test to prove their humanness, to access a website. This traditional design not only causes user friction, but it also poses several threats and liabilities to the company hosting the website. Being able to differentiate users without requiring them to login or prove their humanness, i.e. CAPTCHA, is a meaningful contribution.

5.2 Future Work

Due to time constraints, implementing the classification stage of the system design **??** is a plan for future work. Classification is meant to occur after users have been clustered and differentiated. By knowing that users A and B exist in the inputted sessions datas, limiting the frequency of a user's network requests on a website can be achieved without them needing to login. However, this thesis research did not reach that point in the system design. Additionally, more features can be analyzed to improve the clustering accuracy. This includes the validity of the principal component analysis done in the evaluation's clustering section 2.3.

Realtime features generation is an integral part of this detection scheme. Webservers having the ability to generate features in realtime, without sacrificing performance, is a important aspect to consider. Specifically, a distributed system can be applied to decrease runtimes of the features generation step. By introducing a set of nodes, one of which includes a master that directs workers and their inputs and outputs, the high computational costs of generating features from numerous sessions can be manageable.

BIBLIOGRAPHY

[1] Balabit Mouse Challenge Dataset. https://github.com/balabit/Mouse-Dynamics-Challenge. Accessed: 2021-02-04.

[2] Cal Poly Thesis Template. https://github.com/CalPolyCSC/thesis-template.

[3] Distil Networks. 2019 BAD BOT REPORT: The Bot Arms Race Continues. Technical report.

[4] Justin Morgan's thesis git repo. https://github.com/2justinmorgan/thesis.

[5] Akamai. Akamai's Bot Manager - Advanced strategies to flexibly manage the long-term business and IT impact of bots. Technical report, 2018.

[6] S. Alam, G. Dobbie, Y. S. Koh, and P. Riddle. Web bots detection using particle swarm optimization based clustering. *Proceedings of the 2014 IEEE Congress on Evolutionary Computation, CEC 2014*, pages 2955–2962, 09 2014.

[7] M. Antal and E. Egyed-Zsigmond. Intrusion Detection Using Mouse Dynamics. *IET Biom.*, 8:285–294, 2019.

[8] Z. Chu, S. Gianvecchio, and H. Wang. *Bot or Human? A Behavior-Based Online Bot Detection System*, pages 432–449. Springer International Publishing, Cham, 2018.

[9] C. Feher, Y. Elovici, R. Moskovitch, L. Rokach, and A. Schclar. User identity verification via mouse dynamics. *Information Sciences*, 201:19 – 36, 2012.

[10] R. from Entyce. How recaptcha in improving user experience. https://entyce-creative.com/how-is-recaptcha-improving-user-experience/, 2019.

[11] Google. What is clustering? — clustering in machine learning. https://developers.google.com/machine-learning/clustering/overview, 2021.

[12] G. Gu, R. Perdisci, J. Zhang, and W. Lee. Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. CCS'08. pages 139–154, 01 2008.

[13] J. Hamidzadeh, M. Zabihimayvan, and R. Sadeghi. Detection of web site visitors based on fuzzy rough sets. *Soft Computing*, 22(7):2175–2188, Apr 2018.

[14] C. Hauk. Browser fingerprinting: What is it and what should you do about it? https://pixelprivacy.com/resources/browser-fingerprinting/, 2021.

[15] C. Hoffman. Mouse DPI and Polling Rates Explained: Do They Matter for Gaming? http://tipsgeneral.com/how-to/mouse-dpi-and-polling-rates-explained-do-they-matter-for-gaming.html, 2017.

[16] C. Horror. Mouse DPI and USB Polling Rate. https://www.howtogeek.com/182702/mouse-dpi-and-polling-rates-explained-do-they-matter-for-gaming/, 2007.

[17] C. Iliou, T. Kostoulas, T. Tsikrika, V. Katos, S. Vrochidis, and Y. Kompatsiaris. Towards a framework for detecting advanced web bots. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, New York, NY, USA, 2019. Association for Computing Machinery.

[18] J. Jin, J. Offutt, N. Zheng, F. Mao, A. Koehl, and H. Wang. Evasive bots masquerading as human beings on the web. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '13, page 1–12, USA, 2013. IEEE Computer Society.

[19] H. Kang, K. Wang, D. Soukal, F. Behr, and Z. Zheng. Large-scale bot detection for search engines. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 501–510, New York, NY, USA, 2010. Association for Computing Machinery.

[20] Y. Luo, G. She, P. Cheng, and Y. Xiong. Botgraph: Web bot detection based on sitemap. *ArXiv*, abs/1903.08074, 2019.

[21] R. Peter and D. Divya. Optimized outlier based web bot detection. *Journal of Network and Information Security*, 5(1), 2017.

[22] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[23] S. Rovetta, G. Suchacka, and F. Masulli. Bot recognition in a web store: An approach based on unsupervised learning. *Journal of Network and Computer Applications*, 157:102577, 2020.

[24] D. Stevanovic, N. Vlajic, and A. An. Detection of malicious and non-malicious website visitors using unsupervised neural network learning. *Applied Soft Computing*, 13(1):698 – 708, 2013.

[25] A. Wei, Y. Zhao, and Z. Cai. A deep learning approach to web bot detection using mouse behavioral biometrics. In Z. Sun, R. He, J. Feng, S. Shan, and Z. Guo, editors, *Biometric Recognition*, pages 388–395, Cham, 2019. Springer International Publishing.

[26] W. Wu, J. Alvarez, C. Liu, and H.-M. Sun. Bot detection using unsupervised machine learning. *Microsystem Technologies*, 24, 01 2018.

[27] H. Xu, Z. Li, C. Chu, Y. Chen, Y. Yang, H. Lu, H. Wang, and A. Stavrou. Detecting and characterizing web bot traffic in a large e-commerce marketplace. In J. Zhou, M. Soriano, and J. Lopez, editors, *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 143–163. Springer Verlag, Jan. 2018. 23rd European Symposium on Research in Computer Security, ESORICS 2018 ; Conference date: 03-09-2018 Through 07-09-2018.

[28] M. Yahyazadeh and M. Abadi. Botonus: An online unsupervised method for botnet detection. *ISeCure: The ISC International Journal of Information Security*, 4:51–62, 01 2012.

[29] R. V. Yampolskiy. Human computer interaction based intrusion detection. In *Fourth International Conference on Information Technology (ITNG'07)*, pages 837–842, 2007.

[30] Y. Yang, N. Vlajic, and U. T. Nguyen. Next generation of impersonator bots: Mimicking human browsing on previously unvisited sites. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, pages 356–361, 2015.

[31] X. Yuan, C. Li, and X. Li. Deepdefense: Identifying ddos attack via deep learning. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–8, 2017.

[32] M. Zabihimayvan, R. Sadeghi, H. N. Rude, and D. Doran. A soft computing approach for benign and malicious web robot detection. *Expert Systems with Applications*, 87:129 – 140, 2017.

APPENDICES

Appendix A

CLUSTERS USER FREQUENCY TABLES

Table A.1: User frequencies and bias metrics for (9 of 10) iqr clusters

The bias metric is in the bottom-right cell of each cluster table. The median bias metric for these clusters is 3.63

user	**freq**	**%**	**wgt**
u15	70	20	–
u16	67	19	0.1
u12	47	14	0.4
u35	39	11	0.1
u23	31	9	0.1
u7	24	7	0.1
u21	23	7	0.0
u9	21	6	0.0
u20	12	3	0.0
u29	10	3	0.0
	344		**0.5**

user	**freq**	**%**	**wgt**
u15	32	23	–
u20	19	14	0.8
u35	17	12	0.1
u12	17	12	0.0
u16	16	12	0.0
u23	13	9	0.1
u21	8	6	0.1
u7	8	6	0.0
u9	6	4	0.0
u29	2	1	0.0
	138		**3.7**

user	**freq**	**%**	**wgt**
u29	61	18	–
u35	52	15	0.2
u12	46	14	0.1
u15	45	13	0.0
u23	43	13	0.0
u16	27	8	0.2
u21	23	7	0.0
u9	14	4	0.1
u7	14	4	0.0
u20	13	4	0.0
	338		**0.4**

user	**freq**	**%**	**wgt**
u7	34	63	–
u20	10	19	3.6
u12	3	6	0.9
u15	3	6	0.0
u23	2	4	0.1
u29	1	2	0.1
u16	1	2	0.0
u21	0	0	0.0
u9	0	0	0.0
u35	0	0	0.0
	54		**4.1**

user	**freq**	**%**	**wgt**
u9	43	68	–
u20	12	19	3.9
u7	3	5	1.0
u23	2	3	0.1
u12	2	3	0.0
u15	1	2	0.1
u21	0	0	0.0
u35	0	0	0.0
u29	0	0	0.0
u16	0	0	0.0
	63		**4.9**

user	**freq**	**%**	**wgt**
u15	9	19	–
u35	7	15	0.3
u9	5	10	0.3
u23	5	10	0.0
u12	5	10	0.0
u21	4	8	0.1
u7	4	8	0.0
u16	4	8	0.0
u29	3	6	0.0
u20	2	4	0.0
	48		**0.6**

user	**freq**	**%**	**wgt**
u7	39	55	–
u20	18	25	2.4
u9	8	11	1.0
u29	2	3	0.5
u21	1	1	0.1
u35	1	1	0.0
u12	1	1	0.0
u15	1	1	0.0
u16	0	0	0.0
u23	0	0	0.0
	71		**4.1**

user	**freq**	**%**	**wgt**
u16	31	19	–
u15	31	19	0.0
u12	26	16	0.2
u9	16	10	0.4
u21	15	9	0.0
u35	14	9	0.0
u29	11	7	0.1
u20	8	5	0.0
u23	6	4	0.0
u7	3	2	0.0
	161		**10.6**

user	**freq**	**%**	**wgt**
u12	93	21	–
u35	62	14	0.6
u15	60	13	0.0
u16	55	12	0.1
u21	47	11	0.1
u23	40	9	0.1
u29	34	8	0.0
u7	19	4	0.1
u20	18	4	0.0
u9	17	4	0.0
	445		**0.4**

Table A.2: User frequencies and bias metrics for (9 of 10) mean clusters

The bias metric is in the bottom-right cell of each cluster table. The median bias metric for these clusters is 2.65

user	freq	%	wgt
u29	4	24	–
u16	3	18	0.5
u12	3	18	0.0
u23	2	12	0.4
u21	1	6	0.3
u9	1	6	0.0
u35	1	6	0.0
u7	1	6	0.0
u15	1	6	0.0
u20	0	0	0.0
	17		**0.8**

user	freq	%	wgt
u12	136	19	–
u15	114	16	0.3
u16	106	15	0.1
u21	64	9	0.4
u23	59	8	0.0
u35	57	8	0.0
u20	51	7	0.0
u29	47	7	0.0
u9	39	6	0.0
u7	29	4	0.0
	702		**0.1**

user	freq	%	wgt
u35	8	26	–
u29	5	16	0.8
u15	5	16	0.0
u21	4	13	0.2
u7	3	10	0.2
u20	2	6	0.1
u12	2	6	0.0
u16	1	3	0.1
u23	1	3	0.0
u9	0	0	0.0
	31		**1.0**

user	freq	%	wgt
u12	2	100	–
u21	0	0	8.0
u9	0	0	0.0
u20	0	0	0.0
u35	0	0	0.0
u7	0	0	0.0
u29	0	0	0.0
u16	0	0	0.0
u23	0	0	0.0
u15	0	0	0.0
	2		**4.8**

user	freq	%	wgt
u7	86	48	–
u9	41	23	2.0
u20	30	17	0.4
u15	6	3	0.8
u12	5	3	0.0
u23	4	2	0.0
u29	3	2	0.0
u35	2	1	0.0
u21	1	1	0.0
u16	1	1	0.0
	179		**28.1**

user	freq	%	wgt
u35	3	75	–
u7	1	25	4.0
u21	0	0	1.8
u9	0	0	0.0
u20	0	0	0.0
u29	0	0	0.0
u16	0	0	0.0
u23	0	0	0.0
u12	0	0	0.0
u15	0	0	0.0
	4		**3.5**

user	freq	%	wgt
u29	1	100	–
u21	0	0	8.0
u9	0	0	0.0
u20	0	0	0.0
u35	0	0	0.0
u7	0	0	0.0
u16	0	0	0.0
u23	0	0	0.0
u12	0	0	0.0
u15	0	0	0.0
	1		**4.8**

user	freq	%	wgt
u15	3	43	–
u9	1	14	2.3
u20	1	14	0.0
u16	1	14	0.0
u12	1	14	0.0
u21	0	0	0.6
u35	0	0	0.0
u7	0	0	0.0
u29	0	0	0.0
u23	0	0	0.0
	7		**1.8**

user	freq	%	wgt
u15	124	17	–
u35	122	17	0.0
u12	91	12	0.3
u16	89	12	0.0
u23	76	10	0.1
u29	64	9	0.1
u21	50	7	0.1
u9	48	7	0.0
u7	38	5	0.0
u20	30	4	0.0
	732		**0.1**

Table A.3: User frequencies and bias metrics for (9 of 10) median clusters

The bias metric is in the bottom-right cell of each cluster table. The median bias metric for these clusters is 3.48

user	freq	%	wgt
u16	69	23	–
u15	59	20	0.3
u35	38	13	0.5
u12	27	9	0.2
u23	23	8	0.1
u21	19	6	0.1
u9	18	6	0.0
u20	15	5	0.0
u29	15	5	0.0
u7	13	4	0.0
	296		**0.9**

user	freq	%	wgt
u12	37	22	–
u15	26	16	0.5
u20	19	11	0.3
u21	18	11	0.0
u16	18	11	0.0
u23	16	10	0.0
u35	11	7	0.1
u9	8	5	0.0
u29	8	5	0.0
u7	6	4	0.0
	167		**150.0**

user	freq	%	wgt
u35	84	20	–
u15	67	16	0.3
u12	57	14	0.2
u29	51	12	0.1
u23	43	10	0.1
u16	35	8	0.1
u7	26	6	0.1
u21	24	6	0.0
u9	19	5	0.0
u20	15	4	0.0
	421		**0.4**

user	freq	%	wgt
u16	24	20	–
u15	17	14	0.5
u12	14	11	0.2
u20	13	11	0.0
u9	12	10	0.0
u35	11	9	0.0
u23	11	9	0.0
u29	9	7	0.0
u21	7	6	0.0
u7	5	4	0.0
	123		**1.6**

user	freq	%	wgt
u7	8	47	–
u20	4	24	1.9
u9	1	6	1.2
u35	1	6	0.0
u29	1	6	0.0
u23	1	6	0.0
u15	1	6	0.0
u21	0	0	0.1
u16	0	0	0.0
u12	0	0	0.0
	17		**2.1**

user	freq	%	wgt
u9	48	40	–
u7	37	31	0.7
u20	22	18	0.9
u12	4	3	0.9
u15	4	3	0.0
u35	2	2	0.1
u23	2	2	0.0
u21	1	1	0.0
u29	1	1	0.0
u16	0	0	0.0
	121		**5.6**

user	freq	%	wgt
u12	99	21	–
u15	77	16	0.4
u16	53	11	0.4
u21	52	11	0.0
u35	45	9	0.1
u23	45	9	0.0
u29	39	8	0.0
u20	24	5	0.1
u9	22	5	0.0
u7	22	5	0.0
	478		**0.3**

user	freq	%	wgt
u7	40	78	–
u9	2	4	6.0
u20	2	4	0.0
u12	2	4	0.0
u15	2	4	0.0
u35	1	2	0.1
u16	1	2	0.0
u23	1	2	0.0
u21	0	0	0.0
u29	0	0	0.0
	51		**5.2**

user	freq	%	wgt
u16	1	100	–
u21	0	0	8.0
u9	0	0	0.0
u20	0	0	0.0
u35	0	0	0.0
u7	0	0	0.0
u29	0	0	0.0
u23	0	0	0.0
u12	0	0	0.0
u15	0	0	0.0
	1		**4.8**

Table A.4: User frequencies and bias metrics for (9 of 10) mode clusters

The bias metric is in the bottom-right cell of each cluster table. The median bias metric for these clusters is 2.36

user	freq	%	wgt
u7	4	57	–
u35	2	29	2.3
u12	1	14	1.0
u21	0	0	0.9
u9	0	0	0.0
u20	0	0	0.0
u29	0	0	0.0
u16	0	0	0.0
u23	0	0	0.0
u15	0	0	0.0
	7		**2.6**

user	freq	%	wgt
u16	1	100	–
u21	0	0	8.0
u9	0	0	0.0
u20	0	0	0.0
u35	0	0	0.0
u7	0	0	0.0
u29	0	0	0.0
u23	0	0	0.0
u12	0	0	0.0
u15	0	0	0.0
	1		**4.8**

user	freq	%	wgt
u15	27	24	–
u12	22	19	0.4
u16	17	15	0.3
u35	15	13	0.1
u7	8	7	0.3
u21	7	6	0.0
u9	7	6	0.0
u29	4	4	0.1
u23	4	4	0.0
u20	2	2	0.0
	113		**2.2**

user	freq	%	wgt
u20	1	100	–
u21	0	0	8.0
u9	0	0	0.0
u35	0	0	0.0
u7	0	0	0.0
u29	0	0	0.0
u16	0	0	0.0
u23	0	0	0.0
u12	0	0	0.0
u15	0	0	0.0
	1		**4.8**

user	freq	%	wgt
u7	3	100	–
u21	0	0	8.0
u9	0	0	0.0
u20	0	0	0.0
u35	0	0	0.0
u29	0	0	0.0
u16	0	0	0.0
u23	0	0	0.0
u12	0	0	0.0
u15	0	0	0.0
	3		**4.9**

user	freq	%	wgt
u35	2	33	–
u9	1	17	1.3
u20	1	17	0.0
u29	1	17	0.0
u16	1	17	0.0
u21	0	0	0.7
u7	0	0	0.0
u23	0	0	0.0
u12	0	0	0.0
u15	0	0	0.0
	6		**1.2**

user	freq	%	wgt
u15	2	33	–
u21	1	17	1.3
u29	1	17	0.0
u16	1	17	0.0
u23	1	17	0.0
u9	0	0	0.7
u20	0	0	0.0
u35	0	0	0.0
u7	0	0	0.0
u12	0	0	0.0
	6		**1.2**

user	freq	%	wgt
u15	221	15	–
u12	211	14	0.1
u16	180	12	0.1
u35	174	11	0.0
u7	141	9	0.1
u23	136	9	0.0
u9	122	8	0.0
u29	118	8	0.0
u21	111	7	0.0
u20	109	7	0.0
	1523		**0.0**

user	freq	%	wgt
u12	6	40	–
u21	2	13	2.1
u7	2	13	0.0
u15	2	13	0.0
u20	1	7	0.3
u16	1	7	0.0
u23	1	7	0.0
u9	0	0	0.1
u35	0	0	0.0
u29	0	0	0.0
	15		**1.6**

Table A.5: User frequencies and bias metrics for (9 of 10) pca top 8 without theta clusters

The bias metric is in the bottom-right cell of each cluster table. The median bias metric for these clusters is 1.27

user	freq	%	wgt
u16	4	36	–
u35	2	18	1.5
u15	2	18	0.0
u21	1	9	0.5
u7	1	9	0.0
u23	1	9	0.0
u9	0	0	0.3
u20	0	0	0.0
u29	0	0	0.0
u12	0	0	0.0
	11		**1.5**

user	freq	%	wgt
u15	17	16	–
u16	16	15	0.1
u21	13	12	0.2
u35	13	12	0.0
u29	10	9	0.1
u20	9	8	0.0
u12	9	8	0.0
u7	8	7	0.0
u9	7	6	0.0
u23	7	6	0.0
	109		**0.7**

user	freq	%	wgt
u12	103	17	–
u16	90	15	0.2
u15	84	14	0.1
u9	66	11	0.2
u35	57	10	0.1
u23	50	8	0.0
u21	36	6	0.1
u29	36	6	0.0
u20	35	6	0.0
u7	33	6	0.0
	590		**0.2**

user	freq	%	wgt
u7	5	26	–
u15	4	21	0.4
u21	3	16	0.4
u20	3	16	0.0
u9	1	5	0.5
u35	1	5	0.0
u16	1	5	0.0
u12	1	5	0.0
u29	0	0	0.1
u23	0	0	0.0
	19		**0.9**

user	freq	%	wgt
u7	52	32	–
u20	25	15	1.3
u15	19	12	0.3
u16	13	8	0.2
u9	12	7	0.0
u35	11	7	0.0
u23	10	6	0.0
u29	8	5	0.0
u12	8	5	0.0
u21	4	2	0.0
	162		**32.1**

user	freq	%	wgt
u35	85	16	–
u12	85	16	0.0
u15	79	15	0.1
u23	62	12	0.2
u29	58	11	0.0
u21	46	9	0.1
u16	39	7	0.0
u20	31	6	0.0
u7	25	5	0.0
u9	23	4	0.0
	533		**0.1**

user	freq	%	wgt
u7	25	23	–
u16	17	15	0.6
u15	15	14	0.1
u9	12	11	0.2
u35	8	7	0.2
u12	8	7	0.0
u21	7	6	0.0
u20	7	6	0.0
u29	6	5	0.0
u23	5	5	0.0
	110		**1.9**

user	freq	%	wgt
u15	5	29	–
u16	4	24	0.5
u35	3	18	0.4
u20	2	12	0.4
u23	2	12	0.0
u29	1	6	0.2
u21	0	0	0.2
u9	0	0	0.0
u7	0	0	0.0
u12	0	0	0.0
	17		**1.1**

user	freq	%	wgt
u15	28	23	–
u12	26	21	0.1
u16	17	14	0.5
u35	13	11	0.2
u21	10	8	0.1
u7	9	7	0.0
u9	8	7	0.0
u29	5	4	0.0
u23	5	4	0.0
u20	2	2	0.0
	123		**2.0**

Table A.6: User frequencies and bias metrics for (9 of 10) pca top 10 clusters

The bias metric is in the bottom-right cell of each cluster table. The median bias metric for these clusters is 1.4

user	freq	%	wgt
u15	65	23	–
u16	51	18	0.4
u12	36	13	0.4
u35	27	10	0.2
u9	23	8	0.1
u23	19	7	0.1
u21	17	6	0.0
u20	15	5	0.0
u7	15	5	0.0
u29	9	3	0.0
	277		**1.1**

user	freq	%	wgt
u20	18	23	–
u12	15	19	0.3
u16	12	15	0.3
u23	10	13	0.2
u15	10	13	0.0
u21	4	5	0.3
u35	4	5	0.0
u7	3	4	0.0
u9	2	3	0.0
u29	0	0	0.0
	78		**1.2**

user	freq	%	wgt
u9	43	65	–
u7	6	9	4.5
u16	6	9	0.0
u15	4	6	0.2
u23	3	5	0.1
u20	1	2	0.1
u35	1	2	0.0
u29	1	2	0.0
u12	1	2	0.0
u21	0	0	0.0
	66		**4.8**

user	freq	%	wgt
u12	88	22	–
u21	52	13	0.7
u15	51	13	0.0
u35	41	10	0.2
u16	40	10	0.0
u23	39	10	0.0
u29	31	8	0.1
u20	21	5	0.1
u7	18	5	0.0
u9	15	4	0.0
	396		**0.5**

user	freq	%	wgt
u20	2	40	–
u9	1	20	1.6
u35	1	20	0.0
u16	1	20	0.0
u21	0	0	1.0
u7	0	0	0.0
u29	0	0	0.0
u23	0	0	0.0
u12	0	0	0.0
u15	0	0	0.0
	5		**1.6**

user	freq	%	wgt
u16	53	18	–
u15	51	17	0.1
u35	49	16	0.0
u12	33	11	0.3
u23	29	10	0.1
u9	20	7	0.1
u20	18	6	0.0
u21	17	6	0.0
u7	17	6	0.0
u29	15	5	0.0
	302		**0.4**

user	freq	%	wgt
u7	80	66	–
u20	25	20	3.6
u12	4	3	1.2
u15	4	3	0.0
u9	2	2	0.1
u35	2	2	0.0
u23	2	2	0.0
u21	1	1	0.0
u29	1	1	0.0
u16	1	1	0.0
	122		**10.7**

user	freq	%	wgt
u29	58	20	–
u35	52	18	0.2
u12	45	15	0.2
u15	42	14	0.1
u23	28	9	0.2
u16	21	7	0.1
u21	16	5	0.1
u9	13	4	0.0
u7	12	4	0.0
u20	9	3	0.0
	296		**0.7**

user	freq	%	wgt
u15	26	20	–
u12	18	14	0.5
u16	16	12	0.1
u35	15	11	0.0
u21	14	11	0.0
u23	12	9	0.1
u9	11	8	0.0
u29	8	6	0.0
u7	7	5	0.0
u20	5	4	0.0
	132		**2.0**

Table A.7: User frequencies and bias metrics for (9 of 10) stdev clusters

The bias metric is in the bottom-right cell of each cluster table. The median bias metric for these clusters is 1.05

user	freq	%	wgt
u20	6	16	–
u35	6	16	0.0
u9	5	13	0.2
u21	4	11	0.2
u7	4	11	0.0
u29	4	11	0.0
u12	4	11	0.0
u15	3	8	0.1
u16	1	3	0.1
u23	1	3	0.0
	38		**0.5**

user	freq	%	wgt
u15	150	20	–
u16	106	14	0.5
u35	87	12	0.2
u9	85	11	0.0
u12	85	11	0.0
u23	62	8	0.1
u7	54	7	0.0
u21	53	7	0.0
u20	50	7	0.0
u29	15	2	0.0
	747		**0.1**

user	freq	%	wgt
u35	2	40	–
u7	1	20	1.6
u29	1	20	0.0
u23	1	20	0.0
u21	0	0	1.0
u9	0	0	0.0
u20	0	0	0.0
u16	0	0	0.0
u12	0	0	0.0
u15	0	0	0.0
	5		**1.6**

user	freq	%	wgt
u12	143	17	–
u29	97	12	0.4
u7	96	11	0.0
u15	93	11	0.0
u35	91	11	0.0
u16	89	11	0.0
u23	76	9	0.0
u21	58	7	0.0
u20	56	7	0.0
u9	38	5	0.0
	837		**0.1**

user	freq	%	wgt
u35	5	25	–
u15	4	20	0.4
u21	3	15	0.4
u29	3	15	0.0
u20	1	5	0.5
u7	1	5	0.0
u16	1	5	0.0
u23	1	5	0.0
u12	1	5	0.0
u9	0	0	0.0
	20		**0.9**

user	freq	%	wgt
u21	1	100	–
u9	0	0	8.0
u20	0	0	0.0
u35	0	0	0.0
u7	0	0	0.0
u29	0	0	0.0
u16	0	0	0.0
u23	0	0	0.0
u12	0	0	0.0
u15	0	0	0.0
	1		**4.8**

user	freq	%	wgt
u29	1	100	–
u21	0	0	8.0
u9	0	0	0.0
u20	0	0	0.0
u35	0	0	0.0
u7	0	0	0.0
u16	0	0	0.0
u23	0	0	0.0
u12	0	0	0.0
u15	0	0	0.0
	1		**4.8**

user	freq	%	wgt
u12	3	33	–
u15	2	22	0.9
u21	1	11	0.8
u9	1	11	0.0
u20	1	11	0.0
u16	1	11	0.0
u35	0	0	0.3
u7	0	0	0.0
u29	0	0	0.0
u23	0	0	0.0
	9		**1.3**

user	freq	%	wgt
u12	4	24	–
u29	3	18	0.5
u16	3	18	0.0
u35	2	12	0.4
u7	2	12	0.0
u21	1	6	0.2
u23	1	6	0.0
u15	1	6	0.0
u9	0	0	0.1
u20	0	0	0.0
	17		**0.8**