Discover User-App Interactions and Solutions to Reducing the Initial User-CPU Latency

Thy Nguyen
University of California, San Diego
tcn002@ucsd.edu

Milon Chakkalakal
University of California, San Diego
mlonappa@ucsd.edu

Pranav Thaenraj
University of California, San Diego
ethaenra@ucsd.edu

1. Abstract

Regardless of what operating systems are being used, most users are familiar with the ***data loading*** icon, which signals an unpleasant user-wait experience. The amount of waiting time can be varied across the devices and the applications that are launched; for instance, some apps such as MSPaint, Zoom, and Fornite can take approximately 9, 12, and 16 seconds on average to launch respectively [1]. As the problem adversely affects user experience, and limited research was previously conducted in this field, we perform a study to collect user-app interaction data, analyze past behaviors to understand the user-wait events, and propose solutions to reduce such waiting times. In particular, we collect user data over multiple weeks using Intel's XLSDK, which stands for *Input, Actuator, or Logger Libraries Software Development Kit* [2]. Using these data with Statistical and Machine Learning methods – namely Hidden Markov Model and Long Short-Term Memory (LSTM) model – we are able to predict the usual wait time when a user opens an app so that we can launch the app beforehand, and the user can quickly perform their crucial tasks on the app without the long and unnecessary waiting time.

2. Introduction

The user-wait is an undesirable experience when opening an app. This is indicated by a rolling rainbow beach ball (on a Mac machine) or a spinning blue circle next to the mouse cursor (on a Windows/Linux machine). Such symbols are busy cursors, demonstrating that the processes execute other operations and need to set a busy state to the cursors [3]. As a result, the users have to wait for the other processes to complete running and the busy cursors to finally disappear so that they can start interacting with the apps again. The waiting time is unexpected; some might take only a few seconds, but others can last a few minutes. Nevertheless, in today's world, time is of the utmost importance. When interacting with any application, if there is a long running or loading time, people will turn away from the application because of its poor performance on any operating system. Ten to twenty years ago, this was not a problem. Back then, it was understood that applications could take time. However, today, with the progression of technology, applications are competing with each other to retain users, and the best way to do so is by having the user wait time on the application be as small as possible so that the user can have the best experience possible. Hence, our goal is to be able to predict the app launch patterns, specifically by collecting the data using the methods we have learned in the last 10 weeks. Even though this research topic can vastly enhance the user experience, we can hardly find a similar former

research paper or methodology to build upon. Therefore, with the guidance of the Intel experts, we took initiative in exploring how to collect data to be able to use it to predict app launch patterns and set some success criteria using Intel's telemetry framework, which included (1) successfully writing functional collection modules for the Intel® System Usage Report (SUR), (2) performing prediction models (HMM and LSTM/RNN) with an accuracy of more than 50%, and (3) enhance the app launch behavior by 50%+ to better the overall user experience [4]. Finally, the tools we used in this project are comprised of (1) Intel's XLSDK, (2) programming languages, namely C and Python, (3) Version Control System - GitHub, as well as (4) Tableau for visualizations. To analyze the data that we collect, we plan to use the Hidden Markov Model (HMM) to *predict the next application that will be opened by the user* and Long Short-Term Memory (LSTM) model to *predict the duration spent on a particular application by the user*. This will provide us insight into how to reduce the load time for these applications and allow us to give solutions to reduce user wait time.

3. Methodology

3.1 Methodology of Data Collection

During the first 10 weeks, we constantly wrote code to record different kinds of data from our individual desktops/laptops in relation to our system usage data. To accomplish this, we first familiarized ourselves with the Intel® System Usage Report (SUR), a data-collection and analysis framework that enables us to anonymously gather data usage from each device and analyze such data from multiple devices [5]. Accompanying the Intel® SUR collector, the ESRV (energy server) toolchain was also introduced and got set up on the machine on which we collected the data. We then studied the XLSDK User Guide to develop various input libraries (ILs), namely Mouse-Input, User-Wait, Foreground Window, and Desktop Mapper ILs. In general, each one of these input libraries extracts data alongside the timestamps, and we will further explain these input libraries as well as their functions in the below section.

a) Mouse-Input IL

Introduced in week 1 of the Fall quarter, the Mouse-Input IL was used to familiarize ourselves with the Windows machine and help us delve into understanding ESRV plus SUR using XLSDK. As this was our first input library, we were provided with step-by-step instructions on how to write the code from the Intel team. The main purpose of Mouse-Input IL is to capture the mouse (X, Y) positions in pixels, with or without noise [11]. In other words, this input library collects the mouse movements being controlled by the user, and we collect the X and Y positions of the mouse on the screen. We can control the intervals in which we wish to collect the data, allowing us to control the amount of data we acquire over a span of time. The major logic of the Mouse-Input library relies upon the *static_standard_input* sample template, which assists us in understanding the concepts, creating the IL, and collecting the data as desired. Furthermore, we update the Mouse-Input to also track the noise in the X and Y positions. This is accomplished by

applying a 1D Kalman predictor per dimension. In particular, we add some noise to create noisy mouse X and Y positions measured in pixels, then utilize the 1D Kalman predictor to compute the predicted signal of those X and Y coordinates.

The sample outputs are demonstrated in the following figures

A	B	C	D	E	F	G
key	MOUSE(1)	Mouse Y position in pixel(s)				
key	MOUSE(2)	Mouse noisy X position in pixel(s)				
key	MOUSE(3)	Mouse noisy Y position in pixel(s)				
key	MOUSE(4)	Mouse X position Kalman predicted value in pixel(s)				
key	MOUSE(5)	Mouse Y position Kalman predicted value in pixel(s)				

Figure: A part of our *test_key-000000.csv* output dataset
which shows what input from the mouse positions are being collected

---

	A	B	C	D	E	F	G	H	I	J	K
1	Time Stamp	Elapsed Time (ms)	Pause Time (ms)	MOUSE	6	MOUSE(0)	MOUSE(1)	MOUSE(2)	MOUSE(3)	MOUSE(4)	MOUSE(5)
2	Fri Jul 18 12:13:34 2014	0	10	MOUSE	6	187	264	148	269	143	261
3	Fri Jul 18 12:13:34 2014	1	10	MOUSE	6	187	264	163	288	154	276
4	Fri Jul 18 12:13:34 2014	16	10	MOUSE	6	191	289	197	288	175	282
5	Fri Jul 18 12:13:34 2014	31	10	MOUSE	6	195	313	184	344	179	311

Figure: Sample data captured by *mouse_input* IL [11]

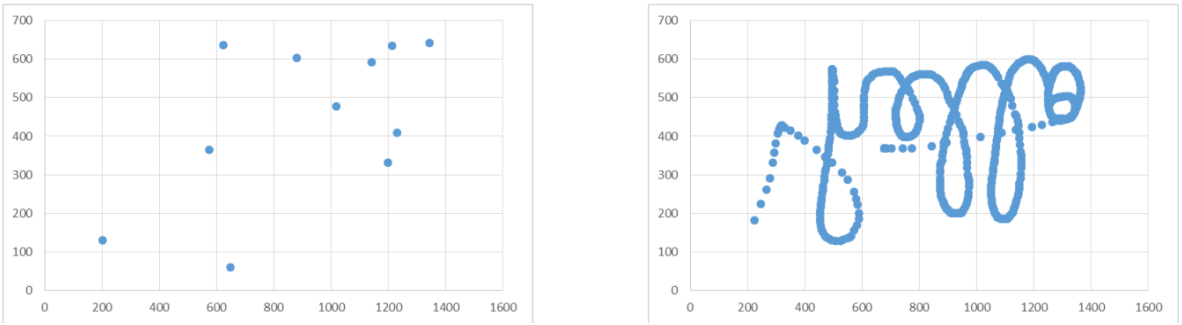


Figure: Sample scatter plots of the mouse movements when captured at a frequency of 1 Hz (left) and 100 Hz (right) [11]

b) User-Wait IL

The User-Wait is the next input library we implemented. In contrast to the above Mouse-Input, we had to build this ourselves. However, we were able to reuse parts of the code from Mouse-Input IL and continued developing the source code. The User-Wait IL is used to retrieve the cursor type and its timestamp. We had to build a collector thread that monitored the state of the cursor icon during intervals. Once again, the span of these intervals can be controlled by us, which allows adjustment in the amount of data collected. The collected data – which describes

the cursor – can vary from a standard arrow to an arrow with a spinning wheel. We also collected data using this input library on whether the mouse is static or dynamic. Therefore, such information is useful in predicting the cursor icon and state.

	A	B	C
5	key	MOUSE-UX(0)	Samples Count (no unit)
6	key	MOUSE-UX(1)	Dynamic IDC_ARROW Samples Count (no unit)
7	key	MOUSE-UX(2)	Dynamic IDC_IBEAM Samples Count (no unit)
8	key	MOUSE-UX(3)	Dynamic IDC_WAIT Samples Count (no unit)
9	key	MOUSE-UX(4)	Dynamic IDC_CROSS Samples Count (no unit)
10	key	MOUSE-UX(5)	Dynamic IDC_UPARROW Samples Count (no unit)
11	key	MOUSE-UX(6)	Dynamic IDC_SIZE Samples Count (no unit)
12	key	MOUSE-UX(7)	Dynamic IDC_ICON Samples Count (no unit)
13	key	MOUSE-UX(8)	Dynamic IDC_SIZENWSE Samples Count (no unit)
14	key	MOUSE-UX(9)	Dynamic IDC_SIZENESW Samples Count (no unit)
15	key	MOUSE-UX(10)	Dynamic IDC_SIZEWE Samples Count (no unit)
16	key	MOUSE-UX(11)	Dynamic IDC_SIZENS Samples Count (no unit)
17	key	MOUSE-UX(12)	Dynamic IDC_SIZEALL Samples Count (no unit)
18	key	MOUSE-UX(13)	Dynamic IDC_NO Samples Count (no unit)
19	key	MOUSE-UX(14)	Dynamic IDC_APPSTARTING Samples Count (no unit)
20	key	MOUSE-UX(15)	Dynamic IDC_HAND Samples Count (no unit)
21	key	MOUSE-UX(16)	Dynamic IDC_HELP Samples Count (no unit)
22	key	MOUSE-UX(17)	Dynamic IDC_UNKNOWN Samples Count (no unit)

23	key	MOUSE-UX(18)	Static IDC_ARROW Samples Count (no unit)
24	key	MOUSE-UX(19)	Static IDC_IBEAM Samples Count (no unit)
25	key	MOUSE-UX(20)	Static IDC_WAIT Samples Count (no unit)
26	key	MOUSE-UX(21)	Static IDC_CROSS Samples Count (no unit)
27	key	MOUSE-UX(22)	Static IDC_UPARROW Samples Count (no unit)
28	key	MOUSE-UX(23)	Static IDC_SIZE Samples Count (no unit)
29	key	MOUSE-UX(24)	Static IDC_ICON Samples Count (no unit)
30	key	MOUSE-UX(25)	Static IDC_SIZENWSE Samples Count (no unit)
31	key	MOUSE-UX(26)	Static IDC_SIZENESW Samples Count (no unit)
32	key	MOUSE-UX(27)	Static IDC_SIZEWE Samples Count (no unit)
33	key	MOUSE-UX(28)	Static IDC_SIZENS Samples Count (no unit)
34	key	MOUSE-UX(29)	Static IDC_SIZEALL Samples Count (no unit)
35	key	MOUSE-UX(30)	Static IDC_NO Samples Count (no unit)
36	key	MOUSE-UX(31)	Static IDC_APPSTARTING Samples Count (no unit)
37	key	MOUSE-UX(32)	Static IDC_HAND Samples Count (no unit)
38	key	MOUSE-UX(33)	Static IDC_HELP Samples Count (no unit)
39	key	MOUSE-UX(34)	Static IDC_UNKNOWN Samples Count (no unit)

Figure: A part of our *test_key-0000001.csv* output dataset
which shows what input from the mouse cursor is being collected

Figure: Example of data loading icons [11]

c) Foreground Window IL

Next, we'll introduce the Foreground Window IL. At this point, we had gotten a basic understanding of how to build Input Libraries and also incorporate Windows APIs to provide us with more useful information.

When implementing this library, we also apply the knowledge of the *hook_input*, whose general purpose is to use a system hook to track the UI objects clicked by the mouse. Simply put, whenever we use the mouse, it generates a message, and the hook will notify the operating system about this information. Some inputs that we can collect from the mouse hook are the X, and Y positions in pixels, the clicked UI object's name, ID, root ID, class name, style, extended style, and the clicked UI object's owning process image as shown in the following database.

INPUT_NAME	INPUT_DESCRIPTION	INPUT_TYPE	INPUT_CATALOG_TIME_UTC
Filter	Filter	Filter	Filter
MOUSE-HOOK(0)	Click X Position (in pixels).	1	2022-10-21 05:33:05.192
MOUSE-HOOK(1)	Click Y Position (in pixels).	1	2022-10-21 05:33:05.192
MOUSE-HOOK(2)	Clicked UI Object ID (no unit).	1	2022-10-21 05:33:05.192
MOUSE-HOOK(3)	Clicked UI Object Root ID (no unit).	1	2022-10-21 05:33:05.192
MOUSE-HOOK(4)	Clicked UI Object Class Name (no unit).	4	2022-10-21 05:33:05.192
MOUSE-HOOK(5)	Clicked UI Object Style (no unit).	1	2022-10-21 05:33:05.192
MOUSE-HOOK(6)	Clicked UI Object Extended Style (no unit).	1	2022-10-21 05:33:05.192
MOUSE-HOOK(7)	Clicked UI Object Name (no unit).	4	2022-10-21 05:33:05.192
MOUSE-HOOK(8)	Clicked UI Object Owning Process Image (no unit).	4	2022-10-21 05:33:05.192

Figure: A part of our *test-0000002.db* output dataset
which shows what inputs from the mouse cursor are being collected

Now, let's go back to our Foreground Window. We use the Foreground Window input library to extract and log the application's name that sits the furthest up front, along with its position, size, and other information. In particular, we utilized two events as triggers for this input library; they are the mouse click and the time tick. In fact, we can detect a change in the foreground window using the mouse click, and we include a time tick in cases where there is no mouse click, but there exists a change to the foreground window, such as when using the task scheduler. With those two triggers, we can retrieve the handle of the foreground window and check if the handle is valid. After that, we further extract and split the window name by the separator character forward flash "/" to obtain only the window's title bar instead of the whole file path. This will prevent us from exposing the user's personal identifiable information (PII). Similarly, we collect the module name, which is the .exe name rather than the complete path to the foreground window, thus handling privacy issues as well. We also obtain the window's class name to further supply useful detail for the analysis and predictions in Quarter 2's project. In addition, the window rectangle's dimensions are recorded by retrieving the X and Y coordinates on the upper-left and lower-right corners using GetWindowRect() and the object type RECT (which is short for "rectangle"). Lastly, we can verify if the window app is immersive and is hung or not by using IsImmersiveProcess() and IsHungAppWindow() functions. In particular, "is_immersive" checks if the application is a Windows Store application, and we capture "is_hung", which checks if the application is responsive or not. A few other functions from Windows APIs that we use are WaitForSingleObject, which waits until the object is in the signaled state (i.e. receiving the STOP_SIGNAL to exit the loop) or until the time-out interval elapses, and we use GetForegroundWindow to retrieve the handle of the foreground window, etc. [10]

	MEASUREMENT_TIME	ID_INPUT	VALUE	PRIVATE_DATA
	Filter	Filter	Filter	Filter
1	2022-12-04 22:49:13.030	3	esrv.exe	0
2	2022-12-04 22:49:13.030	4	VsDebugConsole.exe	0
3	2022-12-04 22:49:13.030	5	ConsoleWindowClass	0
4	2022-12-04 22:49:15.053	3	DSC180A_HW_Week4 - Microsoft Visual Studio	0
5	2022-12-04 22:49:15.053	4	devenv.exe	0
6	2022-12-04 22:49:15.053	5	HwndWrapper	0
7	2022-12-04 22:49:17.072	3	test-000066.db	0
8	2022-12-04 22:49:17.072	4	DB Browser for SQLite.exe	0
9	2022-12-04 22:49:17.072	5	Qt5QWindowIcon	0
10	2022-12-04 22:49:19.074	3	Team QA (CCG DCA UCSD-HDSI Capstone) | Microsoft Teams - Google Chrome	0
11	2022-12-04 22:49:19.074	4	chrome.exe	0
12	2022-12-04 22:49:19.074	5	Chrome_WidgetWin_1	0

Figure: A sample data of our *test-000067.db* database showing the window's title bar (ID_INPUT = 3), image name (ID_INPUT = 4), and class name (ID_INPUT = 5)

d) Desktop Mapper IL

The final input library we worked on last quarter was the Desktop Mapper. This input library was the most challenging input library to develop so far. We initially struggled to understand how to use it. However, with the help of the Intel team and our peers, we were able to create it to the extent that it recorded data from the desktop window. When triggered, the desktop mapper input library maps all the open application windows in z-order and stores information about each one of them, such as their position on the screen and their individual sizes as well. More specifically, the z-order "shows a window's position when given a list of overlapping windows. The z-axis points outward from the screen, where the top window of the z-order overlaps all other windows, and the bottom window is overlapped by all other windows on the z-order axis. The z-order gives us a nice way of visualizing the order of the windows on the screen, and we can connect this with the geometry field to better understand how the windows are positioned on the screen" [12]. Besides, the desktop mapper requires us to understand the use of "private data", which essentially means we can store any data we want and decide how we want to store them. For example, the private data can be the window's z-order or the monitor's enumeration rank [12]. Additionally, just like the Foreground Window IL, we capture if the window "is_immersive" and "is_hung". A few other functions from Windows APIs that we use are GetWindowProcessId, IsWindowVisible, IsZoomed, GetTopWindow, etc. The use of these functions can be understood from their names themselves.

A	B	C
key	DSKTOPMAP(0)	OS:DWM:PROCESS:PID::
key	DSKTOPMAP(1)	OS:DWM:PROCESS:TID::
key	DSKTOPMAP(2)	OS:DWM:WINDOW:TITLE::
key	DSKTOPMAP(3)	OS:DWM:WINDOW:MODULE::
key	DSKTOPMAP(4)	OS:DWM:WINDOW:CLASS::
key	DSKTOPMAP(5)	OS:DWM:WINDOW:DISPLAY::
key	DSKTOPMAP(6)	OS:DWM:WINDOW:LEFT:LOGICAL_UNIT:
key	DSKTOPMAP(7)	OS:DWM:WINDOW:TOP:LOGICAL_UNIT:
key	DSKTOPMAP(8)	OS:DWM:WINDOW:RIGHT:LOGICAL_UNIT:
key	DSKTOPMAP(9)	OS:DWM:WINDOW:BOTTOM:LOGICAL_UNIT:
key	DSKTOPMAP(10)	OS:DWM:WINDOW:WIDTH:LOGICAL_UNIT:
key	DSKTOPMAP(11)	OS:DWM:WINDOW:HEIGHT:LOGICAL_UNIT:
key	DSKTOPMAP(12)	OS:DWM:WINDOW:FOREGROUND:FLAG:
key	DSKTOPMAP(13)	OS:DWM:WINDOW:TOPMOST:FLAG:
key	DSKTOPMAP(14)	OS:DWM:WINDOW:FULLSCREEN:FLAG:
key	DSKTOPMAP(15)	OS:DWM:WINDOW:OCCULTED:FLAG:
key	DSKTOPMAP(16)	OS:DWM:WINDOW:STYLE::
key	DSKTOPMAP(17)	OS:DWM:WINDOW:EXTENDED_STYLE::
key	DSKTOPMAP(18)	OS:DWM:MONITOR:NAME
key	DSKTOPMAP(19)	OS:DWM:MONITOR:PRIMARY:FLAG:
key	DSKTOPMAP(20)	OS:DWM:MONITOR:DISPLAY:LEFT:LOGICAL_UNIT:
key	DSKTOPMAP(21)	OS:DWM:MONITOR:DISPLAY:TOP:LOGICAL_UNIT:
key	DSKTOPMAP(22)	OS:DWM:MONITOR:DISPLAY:RIGHT:LOGICAL_UNIT:
key	DSKTOPMAP(23)	OS:DWM:MONITOR:DISPLAY:BOTTOM:LOGICAL_UNIT:
key	DSKTOPMAP(24)	OS:DWM:MONITOR:DISPLAY:WIDTH:LOGICAL_UNIT:
key	DSKTOPMAP(25)	OS:DWM:MONITOR:DISPLAY:HEIGHT:LOGICAL_UNIT:
key	DSKTOPMAP(26)	OS:DWM:MONITOR:WORK:LEFT:LOGICAL_UNIT:
key	DSKTOPMAP(27)	OS:DWM:MONITOR:WORK:TOP:LOGICAL_UNIT:
key	DSKTOPMAP(28)	OS:DWM:MONITOR:WORK:RIGHT:LOGICAL_UNIT:
key	DSKTOPMAP(29)	OS:DWM:MONITOR:WORK:BOTTOM:LOGICAL_UNIT:
key	DSKTOPMAP(30)	OS:DWM:MONITOR:WORK:WIDTH:LOGICAL_UNIT:
key	DSKTOPMAP(31)	OS:DWM:MONITOR:WORK:HEIGHT:LOGICAL_UNIT:

Figure: A part of our *test_key-000088.csv* output dataset
which shows what inputs are captured by the desktop mapper IL

Methodology of Data Collection - Summary

The code we created to collect data was developed in C and run on Microsoft Visual Studio Community 2019. Each time we run our project solutions (.sln files), the data is generated and stored in three separate files with the following naming convention: test_key-000000.csv, test_timing-000000.csv, and test-000000.db [6]. Specifically, the sequence 000000 is used to denote the order in which the file is generated. It will be automatically enumerated every time we rerun the project solution. The key file (e.g. test_key-000000.csv) explains the column names of the timing and data files, whereas the timing file (e.g. test_timing-000000.csv) stores the execution time for each SUR collector's component and each data sample taken [7]. The test-000000.db – as its extension filename suggests – is a database file, and the (optional) error file might be generated if the project solution has bugs.

As we wanted to study the user-app interactions, we tracked the app launches in the time series format. To collect such information, we used the inputs from the mouse click, mouse cursor, the foreground window, and desktop mapper via the input libraries we built earlier. These ILs allow us to get the positions of the cursor on the screen, capture the mouse movements and activities, and report the change of the foreground windows as well as the desktops and the monitors over time. Some helpful APIs when collecting data from the user side can be found in the above sections, where we discuss each input library in detail.

As a result, the data collected from these ILs are crucial factors in keeping track of user-app interactions, with specific records of the mouse locations, changes in the user interface, as well as user activities. Consequently, we will be able to understand the overall patterns of using the Windows machine of a user; then, we can predict when a user opens an app, make comparisons and understand which apps take the longest time to launch, and make precise predictions in app launching time.

Lastly, when collecting data, the security and privacy of the user are of utmost importance. Even though we had a chance to gather the user's data usage, we respect every user's privacy and concerns. We took careful steps to make sure we weren't accessing the private information of the user on the device without obtaining any permission. Since we will be writing functional modules for SUR, we followed its security guidelines with the assistance of the Intel team.

3.2 Methodology of Predictive Models

In this section, we will quickly explain the conceptual aspects of our prediction models. The two main problems we need to resolve are (1) predicting the probability of using an app, given the former sequence of application usage and (2) predicting the use time of an application in the foreground window.

a). Problem 1: HMM

The first task motivates the use of conditional probabilities due to the word "given" in the problem statement. In general, we have the formula: $P(A|B) = \frac{P(A \cap B)}{P(B)}$. Besides, we notice the Markov Chain strongly assumes that only the current state plays the most crucial role in predicting the future in the sequence, and any other states before the that will not influence the future states. In other words, the Markov Assumption can be briefly described by $P(q_i = a \mid q_1 q_2 \dots q_{i-1}) = P(q_i = a \mid q_{i-1})$, where q_k are the states (for $k \in \{1, 2, \dots, i\}$) [14]. Applying this logic, we can derive a transition probability a_{ij} as the probability of moving from state i to state j. For example, the transition from using a "chrome.exe" to a "cmd.exe" is calculated by the likelihood of using "cmd.exe", given that we have just used "chrome.exe", with the formula: $P(cmd.exe \mid chrome.exe)$

$= \frac{P(chrome.exe, cmd.exe)}{P(chrome.exe)} = \frac{the\ number\ of\ pair\ occurences\ of\ chrome.exe\ and\ cmd.exe}{the\ number\ of\ all\ occurences\ of\ chrome.exe}$,

Thus, a Markov chain helps us quickly figure out the probability for a sequence containing observable events. Nevertheless, confronting the hidden (i.e. unobserved) events requires us to consider implementing a Hidden Markov Model instead. According to Jurafsky and Martin, such a model enables us to interact with both observed and hidden events, allowing the generalization of sequence profiles. Apart from the above Markov Assumption, we have to take into account another assumption called "Output Independence". In short, the probability of observing an event o_i only relies on the state q_i that directly produced o_i. This can be demonstrated by the formula $P(o_i \mid q_1 \dots q_i, \dots, q_T, o_1, \dots, o_i, \dots, o_T) = P(o_i \mid q_i)$, where $o_1 o_2 \dots o_T$ is a sequence of observations of length T, and $q_1, q_2, \dots, q_T$ denote the T states [14]. Together, the two assumptions allow us to understand the idea of the first-order HMM, which can be used to predict the next value based on the previous state sequence.

Now, considering our case specifically, we can denote the *hidden states* as the executables, namely "VsDebugConsole.exe", "explorer.exe", or "ShellExperienceHost.exe". The *observations* can be the apps/tabs such as "Youtube", "Netflix", and "Hulu". Additionally, the *start probabilities* indicate the probability of a state when it appears first in the sequence. For instance, the start probability of "chrome.exe" would be P("chrome.exe" comes first in the sequence). The *emission probabilities* are the likelihood of moving from one executable to an app/tab. Or we can also define the emission probabilities as the likelihood of emitting some types of actions. For example, from "chrome.exe" to "Youtube", we can have the emission probability to be $P(Youtube \mid chrome.exe)$, which can be understood as the chance of watching Youtube when the user has just used Chrome, and the calculation of *emission probabilities* is similar to the ones demonstrated above.

b). Problem 2: RNN (Vanilla, LSTM, and GRU)

The second task requires the use of time series, and Recurrent Neural Networks (RNN) will be a good fit for solving problems related to forecasting sequenced data. There are three main characteristics of RNNs. Firstly, RNNs introduce a looping mechanism, where the current prediction is impacted by the former predictions. An analogy of this is how our brain recognizes the patterns of the alphabetical order "ABCD…" by using the sequential information of a text [15]. Secondly, RNNs share parameters (i.e. the weight matrices W's) across the timestamps and across the positions. Hence, RNNs with more parameters will take more computational costs in training [15]. Thirdly, RNNs can easily face the gradient problems of vanishing or exploding. More specifically, the vanishing problem means the parameter update step will become insignificant once the gradient diminishes, whereas the update step becomes highly large when there exists a fast increase in the gradient [15].

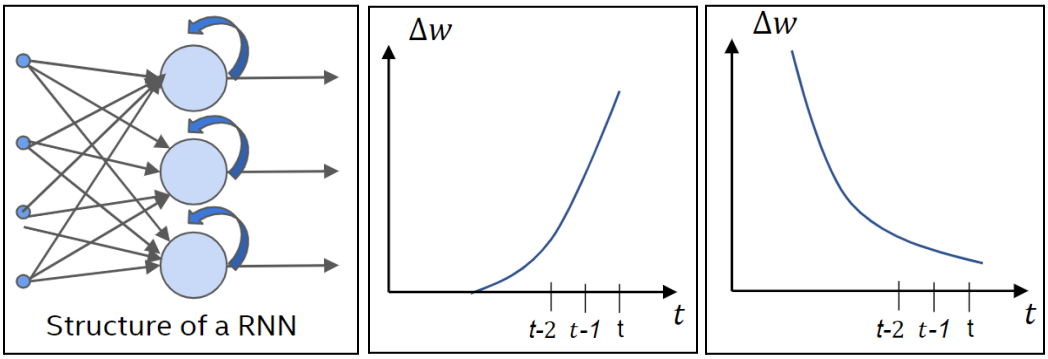


Figure: (from left to right): (1) An example of the structure of RNN, (2)-(3) the vanishing and exploding gradient problems [15]

In our case, each dataset used in a model is obtained from a particular user over some period of time, so we may observe trends and seasonality, with some temporal features in our noisy data, and these properties can be feasibly handled by RNNs. We consider three cases of RNNs, namely

Vanilla, LSTM, and GRU RNNs. In contrast to LSTM, Vanilla RNNs do not have a cell state; they only have hidden states that serve as the memory for RNNs. LSTM, on the other hand, has the cell states along with the hidden states, so it allows the addition or removal of information to and from the cell. Such characteristics of LSTM are regulated by the "gates" [16]. In comparison to LSTM, GRU utilizes fewer training parameters, so it requires less memory and achieves a faster execution time. However, the drawback of GRU would be that it becomes less accurate than LSTM when working with longer sequenced data. Overall, our project will be more inclined toward exploring the performance of LSTM RNNs.

4. Data Analysis

In this section, we will first discuss Exploratory Data Analysis (EDA). Initially, the raw data is collected through the various processes described in this paper. The Mouse-Input IL first collects the coordinates of mouse movement. The User-wait IL collects the cursor type and status along with a timestamp, providing valuable data regarding the static or dynamic nature of the cursor, allowing for cursor state prediction down the line. The Mouse-Hook uses a system hook to record the UI icons that the cursor clicked. Foreground-Window IL collects the various metadata in regard to programs running while the data collection process is happening. This allows for data collection regarding the type of file being run and whether or not the process is hung. Desktop-Mapper IL collects data regarding the positions of all items while the data collection process occurs. All of these sources are combined to provide the raw data we are looking to cleanse.

4.1 Exploratory Data Analysis (EDA)

The first step in EDA is to clean the data that we have collected. This step requires us to drop undesired columns and check for null values in each existing column. Upon looking further into the data that was collected, we discerned that the data from our Foreground-Window IL would be best suited for our prediction model. To understand the data collected, we look at the schema of the data that was collected as shown in the below figure.

	ID_INPUT	INPUT_NAME	INPUT_DESCRIPTION
0	0	FOREGROUND-WIND(0)	Foreground Window Root ID
1	1	FOREGROUND-WIND(1)	Foreground Window Process ID
2	2	FOREGROUND-WIND(2)	Foreground Window Thread ID
3	3	FOREGROUND-WIND(3)	Foreground Window Name
4	4	FOREGROUND-WIND(4)	Foreground Window Image Name
5	5	FOREGROUND-WIND(5)	Foreground Window Class Name
6	6	FOREGROUND-WIND(6)	Window Upper Left X Coordinate
7	7	FOREGROUND-WIND(7)	Window Lower Right X Coordinate
8	8	FOREGROUND-WIND(8)	Window Upper Left Y Coordinate
9	9	FOREGROUND-WIND(9)	Window Lower Right Y Coordinate
10	10	FOREGROUND-WIND(10)	Check if the App is Hung or Not
11	11	FOREGROUND-WIND(11)	Check if the App is Immersive or Not

Figure: Schema of Foreground-Window IL

We then made sure that each value of the data was of the correct data type. Further evaluating the data, we noticed there were instances where our data contained "Missing String." in a few rows in relation to a few applications as shown in a figure named *"Timestamp with "Missing String."*

291	2023-01-19 12:17:51.166	3	Missing String.	0
292	2023-01-19 12:17:51.166	4	explorer.exe	0
293	2023-01-19 12:17:51.166	5	ApplicationManager_DesktopShellWindow	0

Figure: Timestamp with "Missing String."

After our thorough investigation, we came to realize that this was because we were not saving the name of the window in Unicode, and therefore, special characters were not being recorded; instead, it was showing us the "Missing String." However, upon fixing this, there were still instances where the file explorer tab returned an empty string when it was accessed from the Shell Tray as shown in Figure *"Timestamp with an empty string"*.

15023	2023-01-25 13:27:19.556	4	explorer.exe	0
15024	2023-01-25 13:27:19.556	5	ApplicationFrameWindow	0
15025	2023-01-25 13:27:19.556	3		0

Figure: Timestamp with an empty string

To keep everything consistent, we decided to impute both the “Missing String.” and the empty entries (related to the File Explorer) with empty strings. Imputation allowed these rows to be used without affecting the overall distribution of the data. Even though there were imputations implemented in our pre-processing step, they did not affect our data as we mainly worked with application names.

Next, we looked at the spread of the data and calculated the use time for each instance of the recorded data. We looked into the top 10 most used applications in terms of time spent on them and created a scatter plot to see if there were any outliers.

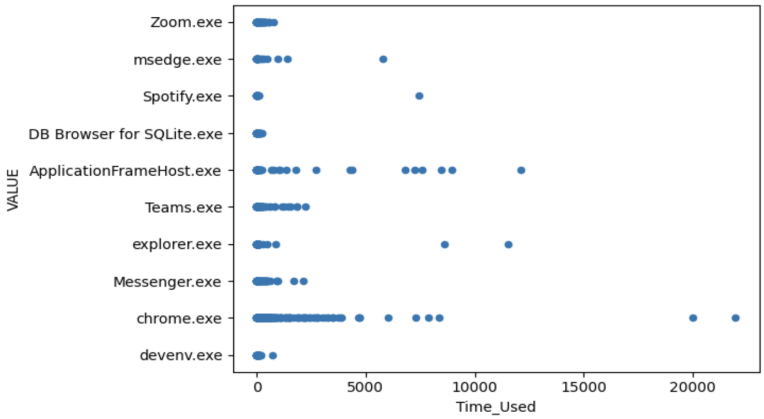


Figure: Use time of applications in seconds

It can be seen that chrome.exe has two instances of time recorded that are about 20000 seconds. We did not consider this to be something wrong with the data, but rather it seemed to be aligned with a student’s usage of Chrome.

We also draw bar charts to explore and summarize the trend of the user’s usage time across the top 5 used apps.

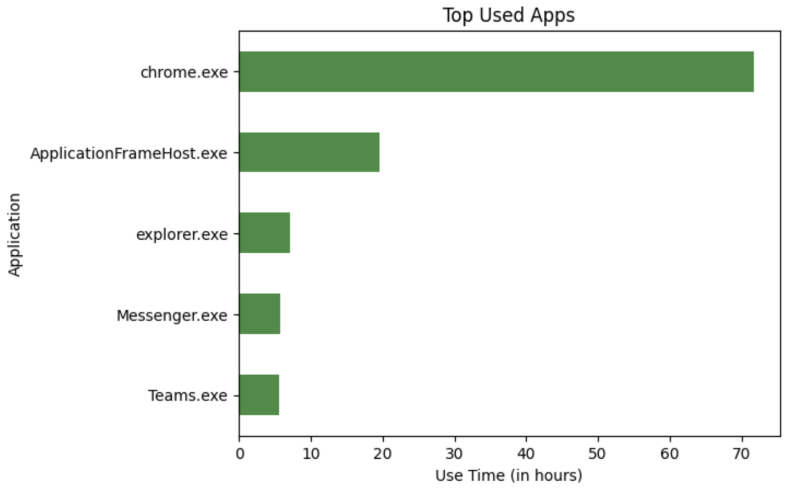


Figure: Application use time in hours

In addition, we also created several data visualizations through the use of Tableau. Providing visual analysis is essential to truly understand the user system used throughout the day. An example of such analysis can be seen below:

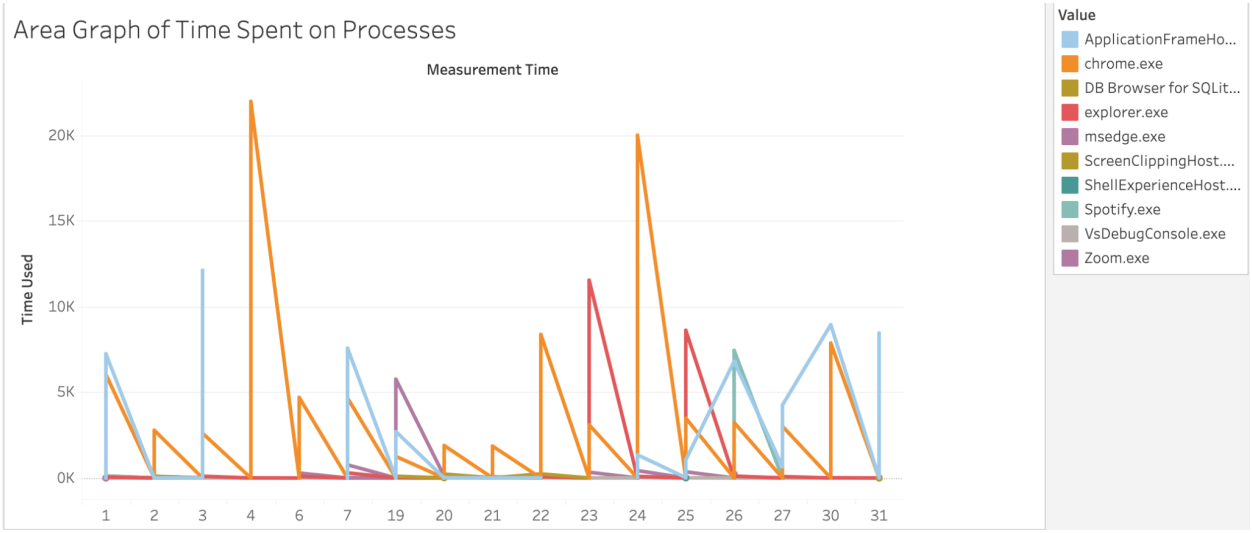


Figure: Area Graph of Daily Time Spent on Processes (in seconds) during 31 days in January 2023

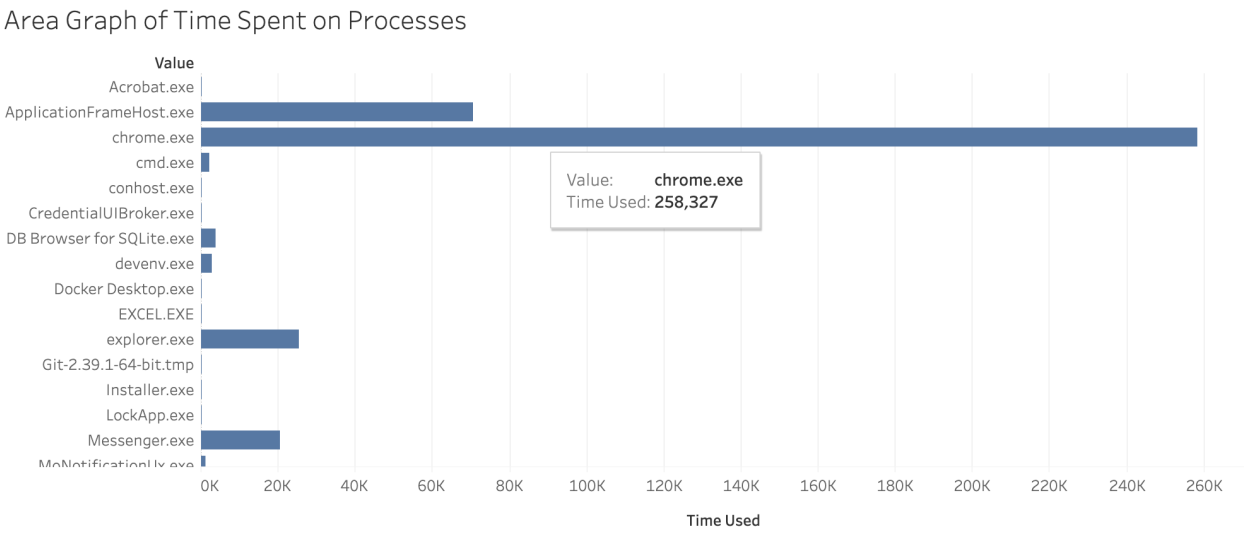
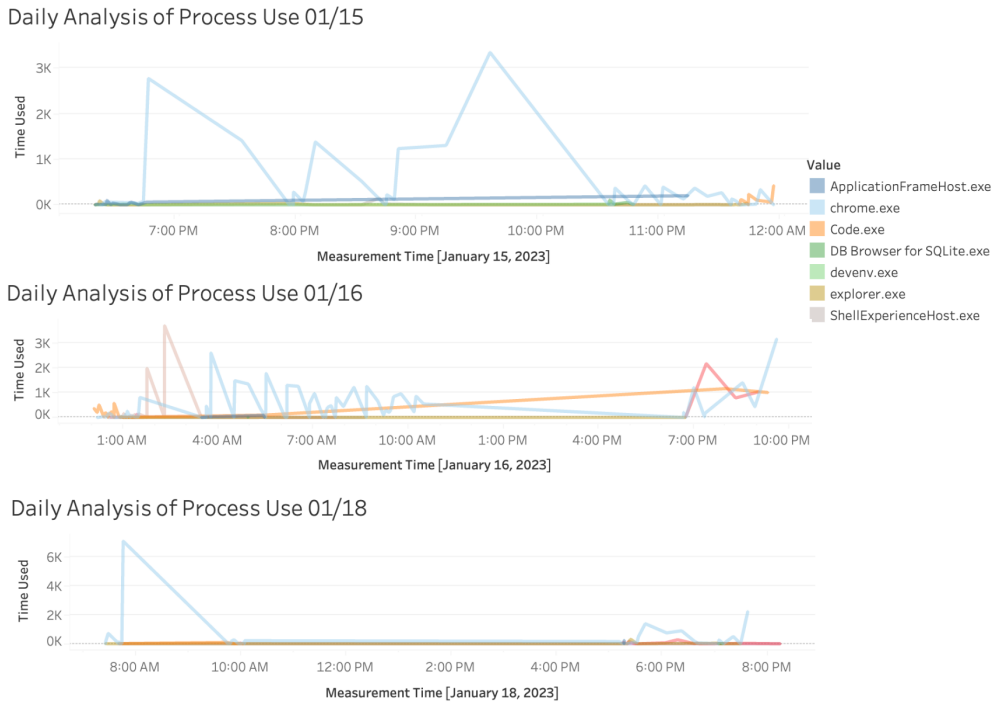


Figure: Process Runtime

The above two charts simply show the usage of processes in terms of elapsed time for a given day. From visualizations like the bar chart, we can see the largest contributor to the data collected on a given day is Chrome, but the area chart adds an additional layer of information regarding the distribution of chrome usage across the entire day. In this way, these visualizations supplement one another well. On top of these visualizations, we also focused on the use of time series analysis to further note patterns in the data:



Here we can see the time-series analysis of process usage for three different days. The chart shows the distribution for the top 5 most frequently used processes throughout the 3-day period. When the data is split by days, we can see patterns that didn't emerge before such as the absence of ApplicationFrameHost on the second day. The presence or absence of processes tells a story regarding how the user spent their time collecting their data and could lead to interesting insights. When combining all of the processes and all of the days together in a single visualization, you get a time series analysis that looks similar to a calendar, allowing for a full understanding of the user's data collection:

Figure: Day / Hour Process Usage

4.2 Data Analysis Using HMM

Next, we use all the cleaned data and preprocess them to be run through the Hidden Markov Model and Long-Short Term Memory Model.

Hidden Markov Model (HMM) is a statistical model to predict the next value based on the sequence of the previous states. It uses the Markov chain which is a sequence of possible events where the probability of each event only depends on the state attained in the previous event. [13] More information about the theoretical aspect of HMM can be found in section 3.2 a). To be able to mimic this, we create a transition and emission matrices. A transition matrix consists of the probabilities of going from one executable to another. An emission matrix consists of the probabilities of going from one executable to another app or tab (or showing an emission of some types of actions). To be able to create these matrices, we have to preprocess the data. Since we are looking only at executables and the applications/tabs, we extract from the Foreground Window only those data that have ID-INPUT values of 3 and 4.

The data preprocessing step for HMM includes the use of Natural Language Processing (NLP) to greatly improve the accuracy of our predictions. Before the text processing, we saw instances of multiple distinct records being associated with the same tabs/apps. This happened due to the description of particular apps/tabs being more detailed than they were intended to be. For instance, "DSC180A – Google Doc – Google Chrome" and "DSC180B – Google Doc – Google Chrome" should both be referred to as the same tab name "Google Doc – Google Chrome". The purpose of text processing is to identify the presence of a tab/app instead of the whole tab's/app's

description and isolate just that portion of the text to replace the original text. The collected data made this process easier by keeping the variations in the tab/app description fairly uniform. Correcting such descriptions allowed for the emission probability calculations to be unaffected by the improper split of identical tab/app names.

In this process, we first split the data into training and testing sets - each representing 80% and 20% of the entire dataset, respectively. We then input these data into our program that learns the Markov chain and creates a transition matrix. Once we train the data on our training set, we then run it on our test set and calculate the accuracy of the model. To calculate the accuracy, we decide that: The prediction is considered accurate as long as the prediction falls in the top *n (number of applications)* probabilities calculated for that specific app. As the value of *n* increases, the prediction accuracy increases as well. The code artifact of this section can be found on our GitHub. The results of HMM will be discussed in more detail within section 5.

4.3 Data Analysis Using LSTM/RNN

Finally, we move to analyze data using LSTM/RNN. We conduct many experiments to finally find out the ones that work for our data. For the reader's convenience, we would like to reiterate the problem statement: *Predict the duration a user spends on an app within an hour, given the past time-series data.*

Initially, we observe the *periodical occurrence* of the days of the week as well as the hours of the day, so we choose to incorporate those elements into our model by applying *sine* function transformation. Later, we also try out the *one-hot encoding* of the weekdays because this technique allows us to turn categorical data like Monday to Sunday into numerical data that can be fit into the model. Unfortunately, these feature engineering trials do not give good predictive results. Hence, we continue thinking of how to utilize the fact that the current timestamp depends on the patterns of the previous timestamps. In fact, if we look closely at the usage pattern, we can observe there are times (namely, from 1 am to 7:30 am) when the users go to sleep and stop using the app, which results in zero values in the amount of usage. After waking up and starting working, the users continuously use the device until the end of the day. We notice the *lookback* technique is able to observe what happens at some timesteps right before the current time point, and since it can detect changes in the user's activity on an app, it gives better predictive results and can capture both the peak time usage as well as the low time usage.

Using such feature engineering, we've tried multiple models such as Vanilla RNN, simple LSTM with just one LSTM layer and one Dense output layer, as well as the Bidirectional LSTM model. The accuracy is calculated based the percentage of predictions that are within some small distance from themselves to the real values. Additionally, we used Root Mean Squared Error,

RMSE = $\sqrt{\frac{\sum_{i=1}^{n} (Predicted_i - Actual_i)^2}{N}}$, as another performance metrics. More information about these model structure/performance can be found in the later section.

While working with the LSTM models, we also come up with another problem statement. Specifically, using the collected data and some data manipulation, we are capable of *providing predictions regarding the total amount of time a particular user spends using a particular process on any given day.*

We take the elapsed time data associated with each process and create a pivot table of sorts per process. Each record (each row in the dataframe) is associated with a day when the data for the process was collected, and the columns of each record are associated with the hour of the day. Thus, there are 24 columns, labeled 0 to 23, referring to the hours of the day and the time spent using the process during that hour. We also have 2 other columns (i.e. the *Total_Usage* and the *Application*) along with 24 mentioned columns. The *Total_Usage* demonstrates how many hours the app is used during a day, and each *Application* is encoded with a unique numeric identifier, which indicates the process being used.

Each record is paired with its respective process in a dictionary. Taking this dataset, we create several deep-learning models using Keras. The first model attempted is the simple RNN model. The Sequential model from the Keras package is utilized to create a model with an input layer, a hidden layer, and a dense layer.

```
# Define the model
model = Sequential()
model.add(SimpleRNN(units=32, return_sequences=True, input_shape=(1, input_dim))
model.add(SimpleRNN(units=16, return_sequences=False))
model.add(Dense(units=1))

# Compile the model
model.compile(loss='mean_squared_error', optimizer='adam')
```

Figure: A Simple RNN Model

Later, we go forward to create a more complex model using the LSTM algorithm through Keras. The LSTM model consisted of Sequential layers, namely two LSTM layers, two Dense layers, and an output Dense layer.

```
# build the RNN model
model = Sequential()
model.add(LSTM(32, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(LSTM(16))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))

# compile the model
model.compile(loss='mean_squared_error', optimizer='adam')

# fit the model to the training data
model.fit(X, y, epochs=100, batch_size=32, verbose=0)

# make predictions using the model
y_pred = model.predict(X)
```

Figure: A LSTM Model

In this particular sample model, the choice is to fit it across 100 epochs as well as the addition of redundant LSTM and dense layers to allow for a much better performance than the simple RNN. The model produces a column of continuous predictions regarding the total time spent using a particular process using the hourly elapsed time. To calculate the accuracy of this model, we need to create numeric bins to place each prediction into. Specifically, we create 4 different bins: [0, 0.01], (0.01, 0.02], (0.02, 0.2], and (0.2, max] based on our data; then we find the True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) to calculate the accuracy ACC = $\frac{TP + TN}{TP + TN + FP + FN}$. The accuracy could be improved dramatically if the binning bounds are shifted such that they are better at accommodating the spread of the data.

5. Results

5.1 HMM

The accuracy of HMM models is calculated based on whether or not the prediction falls in the top n probabilities for that application. As the value of n (the number of applications) increases, the prediction accuracy increases as well. However, when reaching a particular value of n such as $n = 10$, the amount of accuracy increase starts to plateau and does not fluctuate much. Continuing to increment n won't produce new interesting results or help us visualize the significant increase in the transition matrix's accuracy. Thus, we conclude the optimal n should be $n = 5$ in our case.

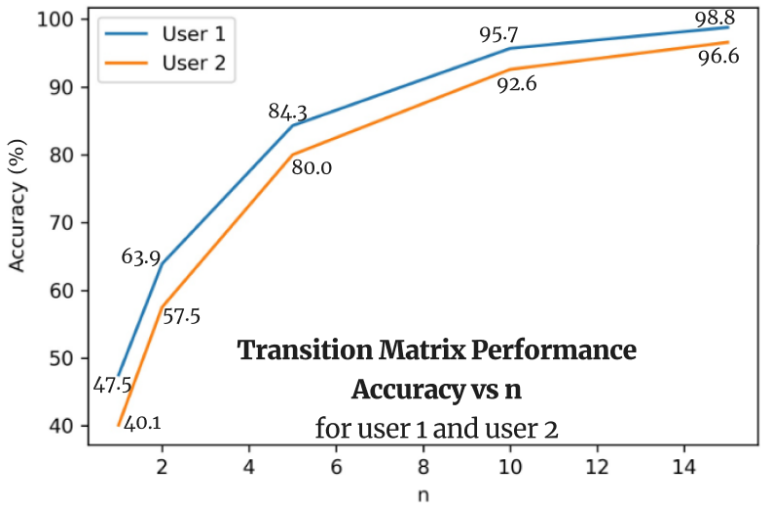


Figure: Comparisons of the Transition Matrix Performance on Chrome between 2 Users

Additionally, when comparing the performance of the transition matrix between two users, we observe that the predictive model works better for user 1 as there are more data collected from user 1 than from user 2 at the moment we run the HMM model. This also supports the statement that more data will give us better results.

For more details about how our transition matrices look like, we attach the two figures showing parts of our results when we set $n = 1$

	ApplicationFrameHost.exe	Code.exe	GitHub.UI.exe	GitHubDesktop.exe	explorer.exe
ApplicationFrameHost.exe	0.000000	0.290323	0.000000	0.000000	0.221198
Code.exe	0.025998	0.000000	0.001393	0.004643	0.050604
GitHub.UI.exe	0.000000	0.333333	0.000000	0.000000	0.333333
GitHubDesktop.exe	0.000000	0.600000	0.000000	0.000000	0.066667
explorer.exe	0.027095	0.095778	0.000000	0.001890	0.000000

Figure: A snippet of the HMM's transition matrix for user 1

As observed from the matrix, user 1 has 2.6%, 0.46%, and 5.06% chance of moving from *Code.exe* to *ApplicationFrameHost.exe*, *GitHubDesktop.exe*, and *explorer.exe* respectively.

	explorer.exe	mintty.exe	msedge.exe	msiexec.exe	msteams.exe
explorer.exe	0.000000	0.000000	0.005747	0.000000	0.000
mintty.exe	0.142857	0.000000	0.000000	0.000000	0.000
msedge.exe	0.210526	0.000000	0.000000	0.000000	0.000
msiexec.exe	0.000000	0.000000	0.000000	0.000000	0.000
msteams.exe	0.000000	0.000000	0.000000	0.000000	0.000

Figure: A snippet of the HMM's transition matrix for user 2

User 2 is likely to enter *explorer.exe* after *mintty.exe* and *msedge.exe* at 14.29% and 21.05% chance, according to the above result.

Along with the transition matrix, we implement the emission matrices for both users 1 and 2. Since the emission matrix relates to the emission of some types of actions, we would interpret the table below as the following. For example, the likelihood of watching Movies&TV after using Chrome of user 1 is around 0.1%, whereas the user would be likely to interact with File Explorer at 21.28% after using Chrome.

	esrv.exe	Downloads	DSC180A_HW_Week4 - Microsoft Visual Studio	Movies & TV	File Explorer	2023 (CCG DCA UCSD-HDSI Capstone) | Microsoft Teams - Google Chrome	Volume Control	Search	Task Scheduler	Create Task	...	patch-1 - Google Chrome	How to Fix Git Error: You need to resolve your current index first - Google Chrome	• HMM.ipynb - System-Usage-Analysis - Visual Studio Code
VsDebugConsole.exe	0.164634	0.012195	0.036585	0.012195	0.182927	0.000000	0.006098	0.000000	0.012195	0.000000	...	0.00	0.000000	0.000000
explorer.exe	0.018135	0.000000	0.023316	0.020725	0.049223	0.002591	0.067358	0.012953	0.002591	0.000000	...	0.00	0.000000	0.000000
devenv.exe	0.050360	0.000000	0.050360	0.007194	0.100719	0.000000	0.000000	0.007194	0.021583	0.007194	...	0.00	0.000000	0.000000
ApplicationFrameHost.exe	0.000000	0.071429	0.023810	0.000000	0.095238	0.023810	0.023810	0.095238	0.071429	0.166667	...	0.00	0.000000	0.000000
chrome.exe	0.088207	0.013423	0.024928	0.000959	0.212848	0.000000	0.062320	0.016299	0.000959	0.000000	...	0.00	0.000000	0.005753
ShellExperienceHost.exe	0.025862	0.000000	0.000000	0.008621	0.025862	0.000000	0.000000	0.008621	0.000000	0.000000	...	0.00	0.000000	0.000000
SearchApp.exe	0.000000	0.000000	0.000000	0.058824	0.411765	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.00	0.000000	0.000000
Unable To Open Process	0.166667	0.000000	0.083333	0.500000	0.083333	0.000000	0.000000	0.041667	0.000000	0.000000	...	0.00	0.000000	0.000000
SnippingTool.exe	0.000000	0.000000	0.013514	0.006757	0.121622	0.006757	0.000000	0.000000	0.000000	0.000000	...	0.00	0.000000	0.000000
DB Browser for SQLite.exe	0.009091	0.000000	0.018182	0.000000	0.054545	0.000000	0.009091	0.000000	0.000000	0.000000	...	0.00	0.000000	0.000000
Code.exe	0.026882	0.005376	0.010753	0.000000	0.037634	0.000000	0.021505	0.005376	0.000000	0.000000	...	0.00	0.002688	0.000000

Figure: Results of the HMM's emission matrix for user 1

The second user has different app interaction from user 1. For example, this user would check their Messenger at 15.50% after using Chrome, and find their documents in File Explorer at

15.25% after Chrome, but they don't interact with Movies&TV (after Chrome) at all.

	esrv.exe	Foreground - Microsoft Visual Studio	Google Docs - Google Chrome	Messenger	sdk	Public -- 2022-2023 (CCG DCA UCSD-HDSI Capstone) | Microsoft Teams	Search	Administrator: Command Prompt	pip documentation v22.3.1 - Google Chrome	File Explorer	...	output - Notepad	output - Excel	Output Dataframe as txt - Google Chrome
VsDebugConsole.exe	0.220339	0.254237	0.025424	0.016949	0.008475	0.016949	0.042373	0.008475	0.000000	0.084746	...	0.000000	0.000000	0.000000
devenv.exe	0.132353	0.000000	0.000000	0.014706	0.000000	0.029412	0.014706	0.000000	0.000000	0.220588	...	0.000000	0.000000	0.000000
chrome.exe	0.057500	0.042500	0.000000	0.155000	0.000000	0.000000	0.037500	0.045000	0.000000	0.152500	...	0.000000	0.000000	0.000000
Messenger.exe	0.017391	0.008696	0.008696	0.000000	0.000000	0.000000	0.000000	0.008696	0.000000	0.156522	...	0.000000	0.000000	0.000000
explorer.exe	0.134831	0.011236	0.000000	0.082397	0.000000	0.003745	0.029963	0.003745	0.000000	0.011236	...	0.007491	0.011236	0.003745
Teams.exe	0.075758	0.166667	0.015152	0.166667	0.000000	0.000000	0.000000	0.000000	0.000000	0.181818	...	0.000000	0.000000	0.000000
SearchHost.exe	0.060606	0.030303	0.000000	0.000000	0.000000	0.000000	0.000000	0.060606	0.000000	0.212121	...	0.000000	0.000000	0.000000

Figure: Results of the HMM's emission matrix used on user 2

5.2 LSTM/RNN

We provide the readers with a table summarizing some of our experiments as shown below. The predictions are made on the Chrome app. The train/test ratio is 80/20 without shuffling when splitting data to ensure the consecutiveness of the time-series data. (The blue rows show our experiments for the second LSTM problem statement)

Models	Design (N=nodes)	Eval Bins / Criteria	Performance	Comments
Vanilla LSTM (Keras) > Split data usage hourly > One-hot encode process names	Input RNN (60N) Hidden Dense (1N) Output Dense (1N) Activation = 'relu' Optimizer = 'adam' Loss = 'binary_crossentropy'	[0, 0.01], (0.01, 0.02], (0.02, 0.2], (0.2, max]	TP = 691, TN = 0, FP = 65, FN = 0 ACC = 91.4%	The model seems to be overfitting as quite a lot of information was provided
Stacked LSTM (Keras) > Split data usage hourly > One-hot encode process names	Input LSTM (16N) Hidden LSTM (16N) Hidden Dense (64N) Output Dense (1N) Activation = 'relu' Optimizer = 'adam' Loss = 'mse'	[0, 0.01], (0.01, 0.02], (0.02, 0.2], (0.2, max]	TP = 467, TN = 52, FP = 13, FN = 224 ACC = 68.65%	The model performs pretty well and reasonably. It's not overfitting as the above

Vanilla RNN (Pytorch)
> One-hot (Mon → Sun)
	# nodes = # layers = 5,
	batch_size = 7,
		RMSE
		Pred == True
			RMSE = 1507.43
			ACC = 7%
				The features are not suitable for

e.g. Mon =[1,0,0,0,0,0,0] > np.sin(hours)	dropout = 0.1, lr = 1e-3, epochs = 150	if within 5 mins of real vals		this predictive problem; nothing meaningful was learned
Vanilla LSTM (Keras) > Lookback 3 timesteps	LSTM (50N) Output Dense (1N) Activation = 'tanh' Optimizer = 'adam' Loss = 'mse'	RMSE, Pred == True if within 3 mins of real values	RMSE = 937.00 ACC = 15.54%	The model is learning something but too simple to capture all the peaks of the time series
Vanilla LSTM (Keras) > Lookback 5 timesteps	LSTM (50N) Output Dense (1N) Activation = 'tanh' Optimizer = 'adam' Loss = 'mse'	RMSE, Pred == True if within 3 mins of real values	RMSE = 945.10 ACC = 23.61%	The model is more complex (with an increase in the number of lookback timesteps), which helps increase the accuracy
Bidirectional LSTM (Keras) > Lookback 5 timesteps	Bi-LSTM (50N) Activation = 'relu' Output Dense (1N) Activation = 'tanh' Optimizer = 'adam' Loss = 'mse'	RMSE, Pred == True if within 3 mins of real values	RMSE = 1103.4 ACC = 52 %	Adding a Bidirectional LSTM layer helps the model learn something actually useful. The accuracy increases noticeably
Bidirectional LSTM (Keras) > Lookback 5 timesteps	Bi-LSTM (128N) Activation = 'relu' Output Dense (1N) Activation = 'relu' Optimizer = 'adam' Loss = 'mae'	RMSE, Pred == True if within some seconds of real values	RMSE = 938.36 ACC (if preds are within 1 sec) = 60.47% ACC (if preds are within 15 secs) = 61.42% ACC (if preds are within 180 secs) = 70.51 %	Changing the activation function from 'tanh' to 'relu' to make sure that no negative values are outputted as the predictive results. Changing the loss function from MSE to MAE (mean absolute error) as well as incrementing the number of nodes (from 50 to 128) help improve the accuracy significantly

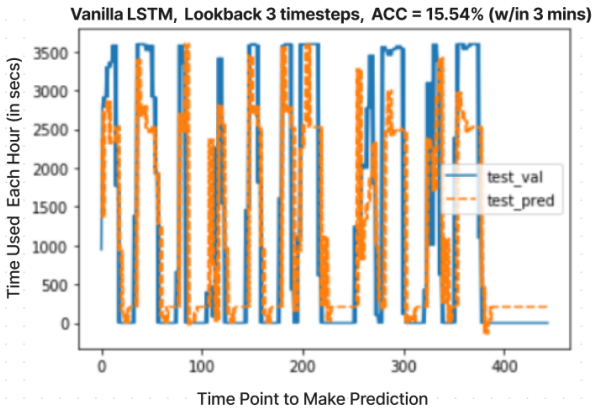
Table: LSTM/RNN experiments

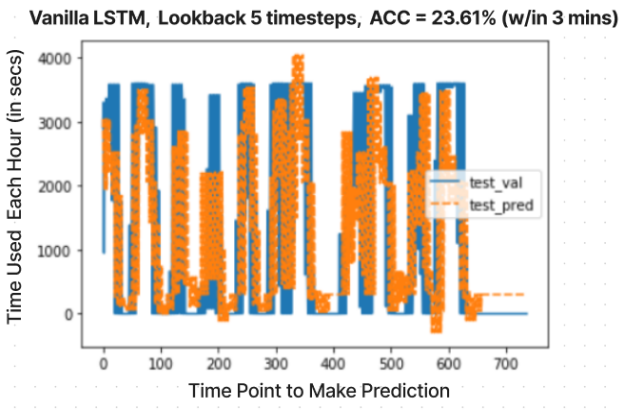
To sum up, the simple RNN model does not perform with the accuracy that was expected. The model seems to have trouble understanding the continuous nature of the prediction column, resulting in low accuracies.

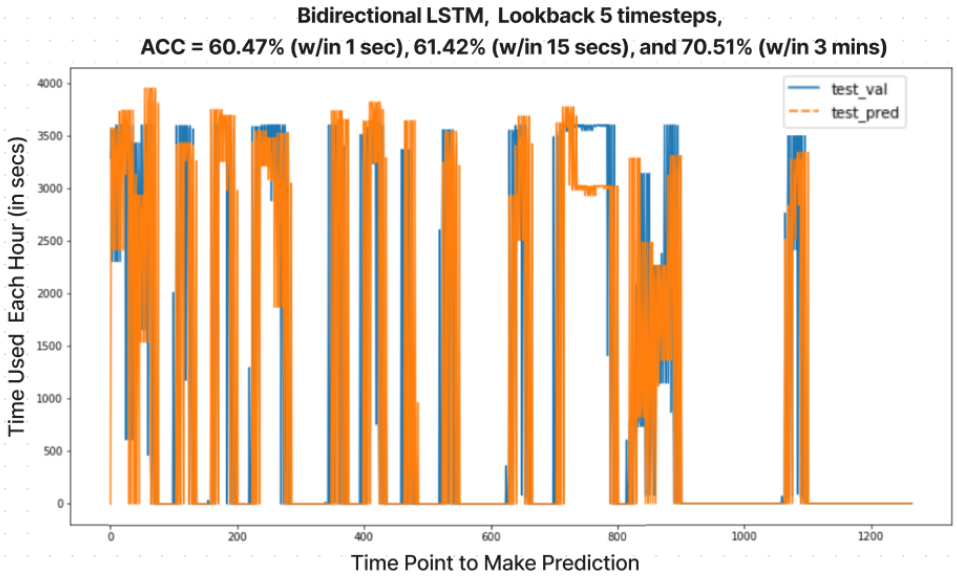
Compared to the simple RNN, the LSTM model is more complex and achieves better results. In particular, the LSTM model can reach an accuracy of more than 60%, sometimes more than 90%, depending on how we divide the bins of our data (for evaluation purposes) and/or how we determine the threshold difference between the predictions and the ground truth values.

a). Problem Statement #1: *Predict the duration a user spends on an app within an hour, given the past time-series data.*

After trying different models and feature engineering techniques, we conclude that the Bidirectional LSTM provides the best possible result for this problem statement. It can capture both the peaks and the low values of the time usage. In fact, the bidirectional characteristics allow the model to learn in both forward and backward directions [17]. Additionally, we can improve the predictive results by tuning the hyperparameters (i.e. with different values of nodes and loss/activation functions). Therefore, to further enhance the accuracy, we can continue doing cross-validation to find the best possible combination of the hyperparameters as well as exploring various feature engineering techniques to see if they could help our predictions.







b). Problem Statement #2: *Predict the total amount of time a particular user spends using a particular app on any given day.*

Regarding the overfitting issue in LSTM problem statement #2, taking into consideration what we found through the model creation process, we observe: The amount of data that we have collected has been reduced by a factor of 24 due to our choice to create the pivot tables in the manner that we did. This makes it impossible to tell if the model is intensely overfitting without adding more data. Furthermore, we find the idea of shifting around the binning bounds to alter accuracy calculations to be bad practice. The ability to do so allows us to provide misleading results if we choose to do so by simply changing the arbitrary partitions we have created of the data.

Adding to this problem, we find the concept of utilizing 24 hours of usage data to predict the total daily usage to be a misuse of deep learning to produce predictions. There is a very clear correlation between the columns of the dataset and the prediction column. When the model is provided with the proper amounts of data, it will see that the sum of the 24 columns will equal the 25th column, and the model will begin to overfit.

Because of this, we believe our next steps could be to incorporate other features into the dataset we are using. Furthermore, adding on a numeric column that associates with the process being used might be a possible solution, along with any other pertinent numeric features. Another solution would be to slightly alter our problem statement. Rather than using hourly usage to predict daily usage, we could potentially use hourly and daily usage distributions as means to predict the process being used. This would take away the continuous aspect of the prediction column, turning the problem into a classification problem. Also, this change could help with the lack of data since the data being used will no longer focus on individual processes at a time and can focus on the entire dataset we collected.

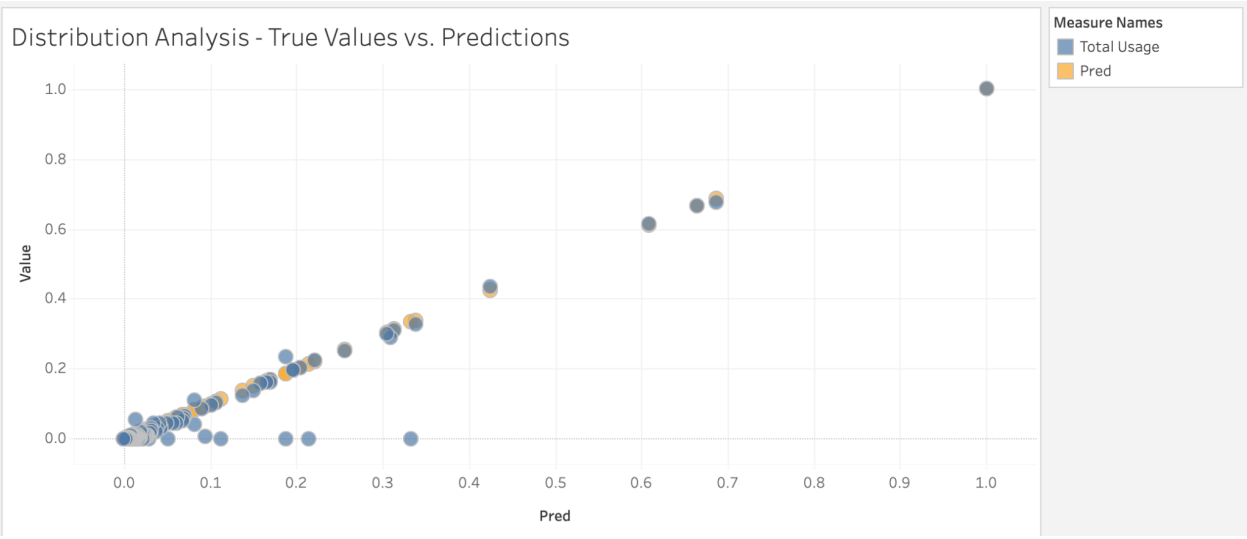


Figure: True vs Predicted Values

6. Conclusion

As the two-quarter project comes to an end, we wish to recapitulate what we've learned over the last six months.

Specifically, we have learned and written codes for the input libraries using *Intel's Software Development Kit*. These input libraries act as data collectors, which allow us to gather information related to the system usage of a user. With data in our hands, we are able to do Data Science. In contrast, *"Without Data, there is no (Data) Science"* [18]. In addition, by knowing how to collect data, we have control over how we want our data to be and what should be collected to improve the analysis.

During the data acquisition process, we need to ensure data quality (i.e. collecting the desired measurements that we claim we would collect), flexibility (i.e. considering how to quickly adapt the collection process to unseen situations), and data privacy (i.e. protecting the personally identifiable information of the users).

Later, we input the data collected to conduct Exploratory Data Analysis. For EDA, we discover the "Missing String." problem and resolve it by imputation. Besides, we observe that Chrome is the top frequently used app among all applications that both users use. The major reason is that Chrome allows users to visit other applications/websites quickly; for example, we can enter the websites of Messenger, Facebook, and Netflix just by using Chrome.

Next, as perceived from the result section, our HMM models are working pretty well with some accuracies that go beyond 90%. The accuracy is defined by the percentage of correct predicted probabilities. A prediction is deemed correct if it falls within the top *n (num_apps)* probabilities predicted. As the value of *n* increases, we receive a higher accuracy.

Additionally, we develop different experiments for our LSTM models. The predictive accuracies are acceptable as the *working* LSTM models are all giving the accuracies above 50% (i.e. above the probability of a coin flip), which shows our models' functionality. Some can reach more than 90%, which seems to be an overfitting problem as we give too much information to the model. We can further improve our models by (1) considering different sets of hyperparameters when tuning our models and (2) re-adjusting our feature engineering process, which allows us to input other types of features that we are able to extract from the data, and avoid the overfitting / underfitting problems.

In short, learning how to collect data and input them into different predictive models helps us vastly in our journey of becoming Data Scientists. In fact, by understanding where the data comes from and how the data are generated, we can recognize the informative and useful data features to feed into our predictive models. This is also what makes us different from other Data

Scientists who have not known about data acquisition yet. On the other hand, by observing the results of predictive models, we can recognize what features should be collected; then we can come back, re-adjust the input libraries, and generate new data with new features.

In the future, we will continue collecting more data to improve the patterns that can be learned in our predictive models. By applying this data analysis pipeline, we can infer the sequence of application usage along with the time used. The analysis will assist us in developing the scripts that process background tasks, and we can - for example - use the Task Scheduler to pre-launch the app 2-3 minutes beforehand. That is how we can reduce the initial latency and better user experience.

7. References

[1] Intel. (2022, October 7). UCSD Data Acquisition Capstone Using DCA Collector-20220930_140834-Meeting Recording.mp4 [Online; accessed 30-October-2022].

[2] Intel. (2016). *XLSDK User Guide (Windows)* (2nd ed.). Page 15

[3] Wikipedia, "Windows wait cursor — Wikipedia, the free encyclopedia," 2021. [Online; accessed 30-October-2022].

[4] Intel. (2022, October 7). UCSD Data Acquisition Capstone Using DCA Collector-20220930_140834-Meeting Recording.mp4 [Online; accessed 30-October-2022].

[5] Intel. (2016). *Xlsdk User Guide (Windows)* (2nd ed.). Page 15

[6] Intel. (2016). *Xlsdk User Guide (Windows)* (2nd ed.). Page 38

[7] Intel. (2016). *Xlsdk User Guide (Windows)* (2nd ed.). Page 40

[8] Intel. (2022, October 7). UCSD Data Acquisition Capstone Using DCA Collector-20220930_140834-Meeting Recording.mp4 [Online; accessed 30-October-2022].

[9] Wibjorn. (n.d.). *Microsoft learn: Build skills that open doors in your career*. Microsoft Learn: Build skills that open doors in your career. Retrieved October 30, 2022. Link

[10] Karl Bridge Microsoft. "WaitForSingleObject Function." *Win32 Apps | Microsoft Learn*, Link

[11] Intel. (2022, December 5). UCSD Data Acquisition Capstone Using DCA. Link

[12] Intel. (2022, December 5). UCSD Data Acquisition Capstone Using DCA. Link

[13] Intel. (2023, February 12). UCSD Data Acquisition Capstone Using DCA. Link

[14] Stanford. (2023, February 12). HMM. Link

[15] Intel. (2023, February 12). UCSD Data Acquisition Capstone Using DCA. Link

[16] Intel. (2023, February 12). UCSD Data Acquisition Capstone Using DCA. Link

[17] Machine Learning Mastery. (2023, March 12). How to Develop LSTM Models for Time Series Forecasting. Link

[18] Intel. (2020). Lecture 1. Link

8. Appendix

- **Section 1**: Abstract
- **Section 2**: Introduction
- **Section 3**: Methodology
    - 3.1: Methodology of Data Collection
        - a) Mouse-Input IL
        - b) User-Wait IL
        - c) Foreground Window IL
        - d) Desktop Mapper IL
        - Methodology of Data Collection - Summary
    - 3.2: Methodology of Predictive Models
        - a). Problem 1: HMM
        - b). Problem 2: RNN (Vanilla, LSTM, and GRU)
- **Section 4**: Data Analysis
    - 4.1: Exploratory Data Analysis (EDA)
    - 4.2 Data Analysis Using HMM
    - 4.3 Data Analysis Using LSTM/RNN
- **Section 5**: Results
    - 5.1 HMM
    - 5.2 LSTM/RNN
- **Section 6**: Conclusion
- **Section 7**: References
- **Section 8**: Appendix