DSC 180B Capstone: Final Report

Section B14, Group 1: Arjun Sawhney, Andrew Chin, Srikar Prayaga

Abstract

Despite advancements in hardware technology, PC users continue to face frustrating app launch times, especially on lower end Windows machines. The desktop experience differs vastly from the instantaneous app launches and optimized experience we have come to expect even from low end smartphones.

We propose a solution to preemptively run Windows apps in the background based on the app usage patterns of the user. Our solution is two-step. First, we built telemetry collector modules to collect real-world app usage data from two of our personal Windows 10 devices. Next, we developed neural network models, trained on the collected data, to predict app usage times and corresponding launch sequences. We achieved impressive results on selected evaluation metrics across different user profiles.

Data Collection Overview

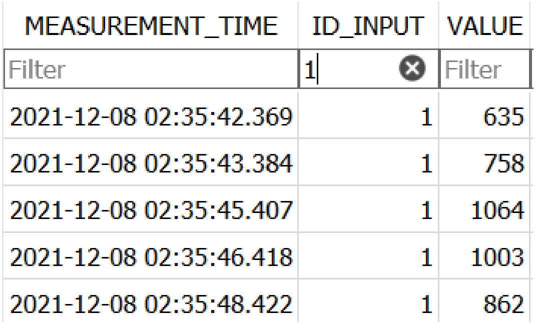
Addressing our problem statement entails understanding the sequence in which apps are launched and how much time is spent on each app. Additional features such as app window placement may indicate the likelihood of usage. To generate this data and required features, we built input libraries in C/C++, utilizing windows APIs from Intel's proprietary XLSDK library. Each input library extracts data alongside collection timestamps. These input libraries are:

- mouse_input.IL
- user_wait.IL
- foreground_window.IL
- desktop_mapper.IL

For the purposes of our project, we focus on visually analyzing and forecasting data obtained from the foreground_window input library.

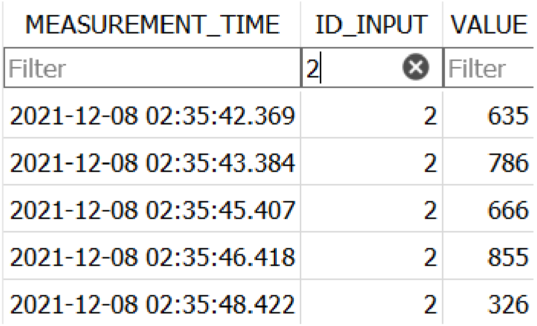
Mouse Input IL

The mouse_input.IL project served as a predicate to the other input libraries. This task was designed to help us familiarize ourselves with the Windows development environment and accompanying configurations. As such, we incorporated Intel provided code into the static_standard_library sample template. The goal of this IL is to store and predict mouse movement data. Every 100 ms, we track the x and y position in pixel(s) of the mouse cursor and apply a 1D Kalman predictor to expose the inputs. We track the mouse noise in both the x and y positions as well as the Kalman predicted value in both dimensions. The following tables are the outputs of the mouse_input.IL:



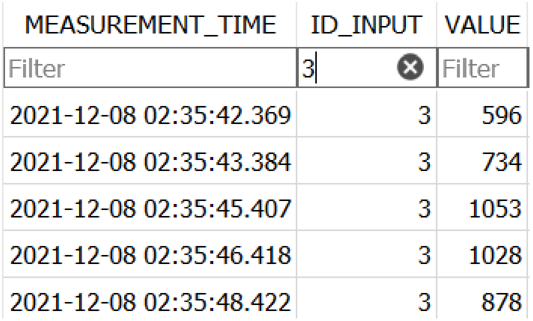
MEASUREMENT_TIME	ID_INPUT	VALUE
Filter	1 ⊗	Filter
2021-12-08 02:35:42.369	1	635
2021-12-08 02:35:43.384	1	758
2021-12-08 02:35:45.407	1	1064
2021-12-08 02:35:46.418	1	1003
2021-12-08 02:35:48.422	1	862

Mouse X position in pixel(s)



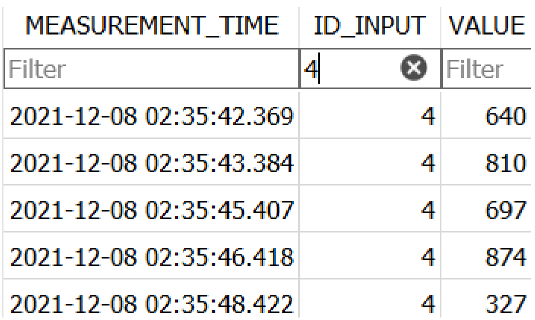
MEASUREMENT_TIME	ID_INPUT	VALUE
Filter	2 ⊗	Filter
2021-12-08 02:35:42.369	2	635
2021-12-08 02:35:43.384	2	786
2021-12-08 02:35:45.407	2	666
2021-12-08 02:35:46.418	2	855
2021-12-08 02:35:48.422	2	326

Mouse Y position in pixel(s)



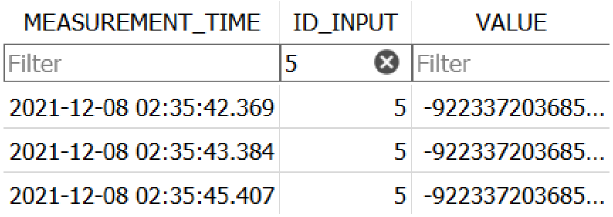
MEASUREMENT_TIME	ID_INPUT	VALUE
Filter	3 ⊗	Filter
2021-12-08 02:35:42.369	3	596
2021-12-08 02:35:43.384	3	734
2021-12-08 02:35:45.407	3	1053
2021-12-08 02:35:46.418	3	1028
2021-12-08 02:35:48.422	3	878

Mouse noisy X pos. in pixel(s)



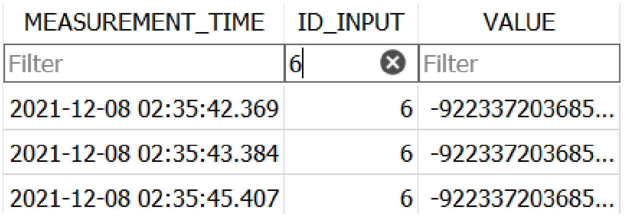
MEASUREMENT_TIME	ID_INPUT	VALUE
Filter	4 ⊗	Filter
2021-12-08 02:35:42.369	4	640
2021-12-08 02:35:43.384	4	810
2021-12-08 02:35:45.407	4	697
2021-12-08 02:35:46.418	4	874
2021-12-08 02:35:48.422	4	327

Mouse noisy Y pos. in pixel(s)



MEASUREMENT_TIME	ID_INPUT	VALUE
Filter	5 ⊗	Filter
2021-12-08 02:35:42.369	5	-922337203685...
2021-12-08 02:35:43.384	5	-922337203685...
2021-12-08 02:35:45.407	5	-922337203685...

Mouse X pos. Kalman pred val in pixel(s)



MEASUREMENT_TIME	ID_INPUT	VALUE
Filter	6 ⊗	Filter
2021-12-08 02:35:42.369	6	-922337203685...
2021-12-08 02:35:43.384	6	-922337203685...
2021-12-08 02:35:45.407	6	-922337203685...

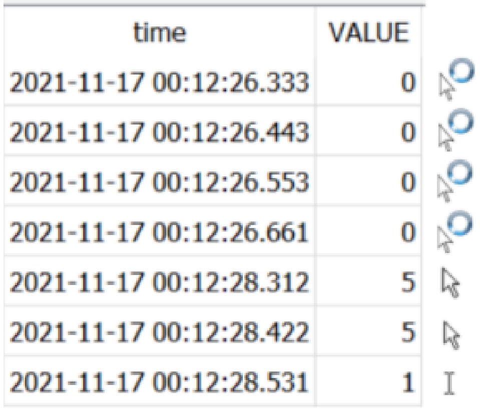
Mouse X pos. Kalman pred val in pixel(s)

We faced a bug with the Kalman predictor wherein it only logged negative float values. However, most importantly, our VSCode configurations were implemented correctly.

User Wait IL

The purpose of the user_wait_IL is to obtain the cursor type and accompanying usage times. We built this IL off the static_standard_pure_event_driven sample template. This project required the creation of a collector thread which monitors the state of the mouse cursor icon at regular intervals (every 100ms). We initialize an array of handles containing references to 16 different mouse cursors (ex. Standard arrow, arrow with spinning wheel, etc.). Our thread process calls the custom_event_listener_thread function, setting off an infinite while loop which runs until it receives the stop_request (when the user presses CTRL+C). Inside the loop, we implement the WaitForSingleObject API function which tracks timestamps in 100 ms pause intervals. During each pause, we store the size of the current mouse cursor and retrieve the current cursor handle using GetCursorInfo. We compare the current handle against every handle in our handle array and store the index of our array where the handles match. This way, we have mapped an integer value to the mouse cursor handles and saved storage space. As soon as the while loop is interrupted, we exit the thread and handle any possible errors.

The outputs are automatically generated in a SQLITE database file. In the sample on the right, we see that the cursor value of 0(app starting) changes to 5 (standard arrow) and then to 0 (ibeam cursor) when we launch photoshop, then open MS Word and start typing.



time	VALUE	
2021-11-17 00:12:26.333	0	↖○
2021-11-17 00:12:26.443	0	↖○
2021-11-17 00:12:26.553	0	↖○
2021-11-17 00:12:26.661	0	↖○
2021-11-17 00:12:28.312	5	↖
2021-11-17 00:12:28.422	5	↖
2021-11-17 00:12:28.531	1	I

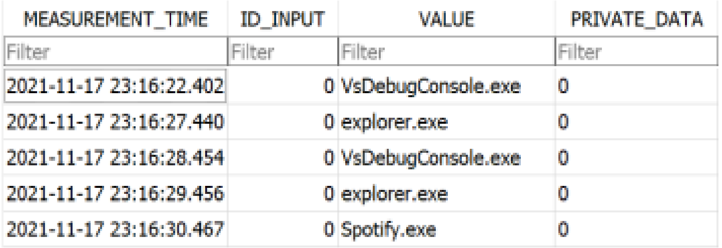
Upon collecting and verifying this data, we performed data wrangling on the output tables. We narrowed our scope to only a few frequently appearing cursor types out of the 15 mapped icons in our data. These include the IDC_APPSTARTING, IDC_ARROW, IDC_HAND, IDC_IBEAM, and IDC_WAIT cursor icons. All other cursor types are classified as “OTHER”. We may wish to only examine app launch times using the IDC_APPSTARTING icon in which case our data will be engineered to contain binary inputs.

As this was our first time building an input library in C++ by ourselves, we struggled to navigate the MSDN and understand the windows API functions. However, with some help from Jamel, we were able to debug our issues, and collect data successfully.

Foreground Window IL

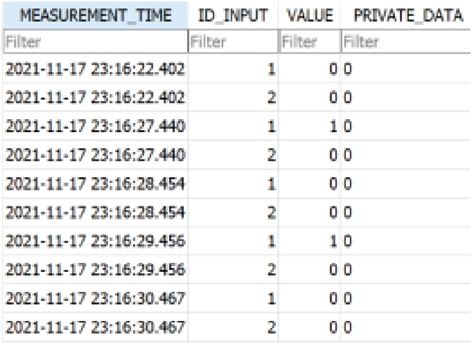
The purpose of the foreground_window IL is to detect and log the name of the executable whose app window sits atop all others. We implemented two waiting events to trigger the foreground window check: mouse clicks, and 1000ms time intervals.

To detect these two events, we created a collector thread which calls on the WaitForMultipleObjects api function. As in user wait, an infinite while loop runs until it receives the stop_request (CTRL+C). To limit the size of our output, we made the decision to log the foreground window only if it is different from the previous foreground window. Therefore, we check if the current window's process ID is different from the previously stored process ID. If so, we retrieve the executable token (the string.exe name) of the new foreground window and log it to our output table. Additionally, we captured the features "isImmersive'' which checks if a process belongs to a Windows store app and "isHung'' which checks if the foreground window is unresponsive. Upon each detected foreground window change, we emit a custom IDCTL signal with the message "FOREGROUND-WINDOW-CHANGED" to be processed by the desktop_mapper IL. The collector pauses if the device enters screen-saver mode or a mouse is clicked over the collector, and stops running when CTRL+C (stop signal) is clicked. The string outputs appeared as expected in the form of app executable names. The accompanying isImmersive and isHung binary features are located in a separate table.



MEASUREMENT_TIME	ID_INPUT	VALUE	PRIVATE_DATA
Filter	Filter	Filter	Filter
2021-11-17 23:16:22.402	0	VsDebugConsole.exe	0
2021-11-17 23:16:27.440	0	explorer.exe	0
2021-11-17 23:16:28.454	0	VsDebugConsole.exe	0
2021-11-17 23:16:29.456	0	explorer.exe	0
2021-11-17 23:16:30.467	0	Spotify.exe	0

Foreground Window Output Table



MEASUREMENT_TIME	ID_INPUT	VALUE	PRIVATE_DATA
Filter	Filter	Filter	Filter
2021-11-17 23:16:22.402	1	0	0
2021-11-17 23:16:22.402	2	0	0
2021-11-17 23:16:27.440	1	1	0
2021-11-17 23:16:27.440	2	0	0
2021-11-17 23:16:28.454	1	0	0
2021-11-17 23:16:28.454	2	0	0
2021-11-17 23:16:29.456	1	1	0
2021-11-17 23:16:29.456	2	0	0
2021-11-17 23:16:30.467	1	0	0
2021-11-17 23:16:30.467	2	0	0

is_immersive Output Table

The primary challenge we faced in the foreground_window.IL was in regard to duplicate exe strings being logged with different timestamps despite our process checking logic. We deduced that this was because the user was using split windows for the app, resulting in two separate processes, and alternating their mouse back and forth between them. Since the split windows were for the same app executable, we decided to drop consecutive duplicates. This does not affect time spent on the app executable as this is calculated from the first measurement timestamp.

Desktop Mapper IL

The Desktop Mapper input library was the most challenging to develop. Although our implementation is not optimal, we hope to eventually enrich our prediction models with features from this data. The aim is to collect detailed information on every desktop window and understand where app windows are positioned with respect to each other. The data collection is triggered whenever the foreground_window emits an iDCTL signal.

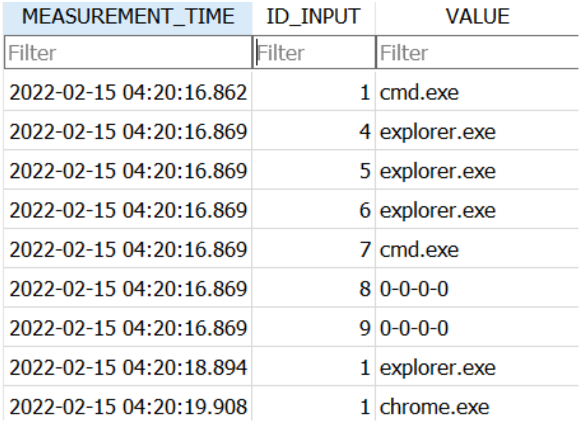
We implement multi-threaded processes consisting of the collector and the logger threads. However, we must first register the custom DCTL signal that we are expecting from the foreground window in modeler_open_inputs. In our case, it is a string that says "FOREGROUND-WINDOW-CHANGED." The collector thread is then signaled to start collecting data through a custom waiting event titled h_foreground_window_changed. Similarly, the logger thread is also signaled to start logging by a wait event titled h_log_window_info after the data on all windows is stored in a desktop array by the collector thread. Within the collector thread we define two essential functions that grab and store window information into an array of custom windows structures. The first function, get_window_info, captures many attributes of a window, including the executable names of the app windows above and below it on the z-axis, window rectangle geometry (in the form of a formatted string), as well as some other attributes such as isHung. To ensure that two processes do not attempt to write over our windows structure at the same time, we enter a CriticalSection before calling the get_window_info function. The second key function, map_desktop(), walks the Z-axis of the desktop from top to bottom and generates a single sample of all window structures by calling get_window_info. This is achieved using a while loop that runs until the topWindow turns null. After receiving the signal through the h_log_window_info custom event, the logger thread then logs data per desktop instance by calling a multiplex logging function. Once we verify that the logger is multiplex capable and is currently running, we unmask and update inputs to start logging window data. The logger outputs each window sample in the desktop array into a db file which is indexed by a timestamp representing the time the data was collected.

We faced a substantial number of challenges building the desktop mapper including: passing references to our custom windows structures as inputs to the get_window_info function and performing multiplex logging with our desktop array. Ensuring that none of our processes overwrote the same bits of memory or the output table also required us to use CriticalSections. However, in doing so, we created a shadow period between the collection time and logging time which we were unable to address. The outputs from the desktop_mapper.IL therefore suffered from duplicate logs as well as mismatching timestamps. The output table contains stacked values of features including:

- Current Window Executable Name
- Next Window Executable Name

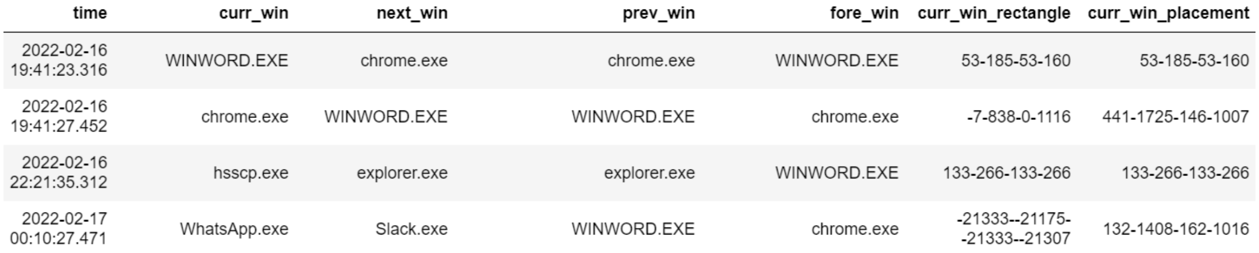
- Previous Window Executable Name
- Foreground Window Executable Name
- Current Window Rectangle
- Current Window Placement

Other window features, is_visible, is_minimized, is_maximized, and is_hung were also collected, however, no valuable information was captured with these features. After some hefty data cleaning, we are able to map a relatively messy table into a relatively clean table.



MEASUREMENT_TIME	ID_INPUT	VALUE
Filter	Filter	Filter
2022-02-15 04:20:16.862	1	cmd.exe
2022-02-15 04:20:16.869	4	explorer.exe
2022-02-15 04:20:16.869	5	explorer.exe
2022-02-15 04:20:16.869	6	explorer.exe
2022-02-15 04:20:16.869	7	cmd.exe
2022-02-15 04:20:16.869	8	0-0-0-0
2022-02-15 04:20:16.869	9	0-0-0-0
2022-02-15 04:20:18.894	1	explorer.exe
2022-02-15 04:20:19.908	1	chrome.exe

Using a custom built python script, we are able to clean this messy table and turn it into a multi-column pandas dataframe.



time	curr_win	next_win	prev_win	fore_win	curr_win_rectangle	curr_win_placement
2022-02-16 19:41:23.316	WINWORD.EXE	chrome.exe	chrome.exe	WINWORD.EXE	53-185-53-160	53-185-53-160
2022-02-16 19:41:27.452	chrome.exe	WINWORD.EXE	WINWORD.EXE	chrome.exe	-7-838-0-1116	441-1725-146-1007
2022-02-16 22:21:35.312	hsscp.exe	explorer.exe	explorer.exe	WINWORD.EXE	133-266-133-266	133-266-133-266
2022-02-17 00:10:27.471	WhatsApp.exe	Slack.exe	WINWORD.EXE	chrome.exe	-21333--21175--21333--21307	132-1408-162-1016

Challenges

As conventional Data Scientists, our skill-sets prior to this project revolved around python and data modeling. Stepping into the shoes of a software engineer and using C for pushing production-ready code was challenging and posed a steep learning curve.

In the foreground window, we noticed that duplicate executables were sporadically being outputted but with different timestamps. We deduced this was because the user had split tabs of that app open or they were changing the size of the windows. We handled this issue by dropping the consecutive duplicates since they account for the same executable.

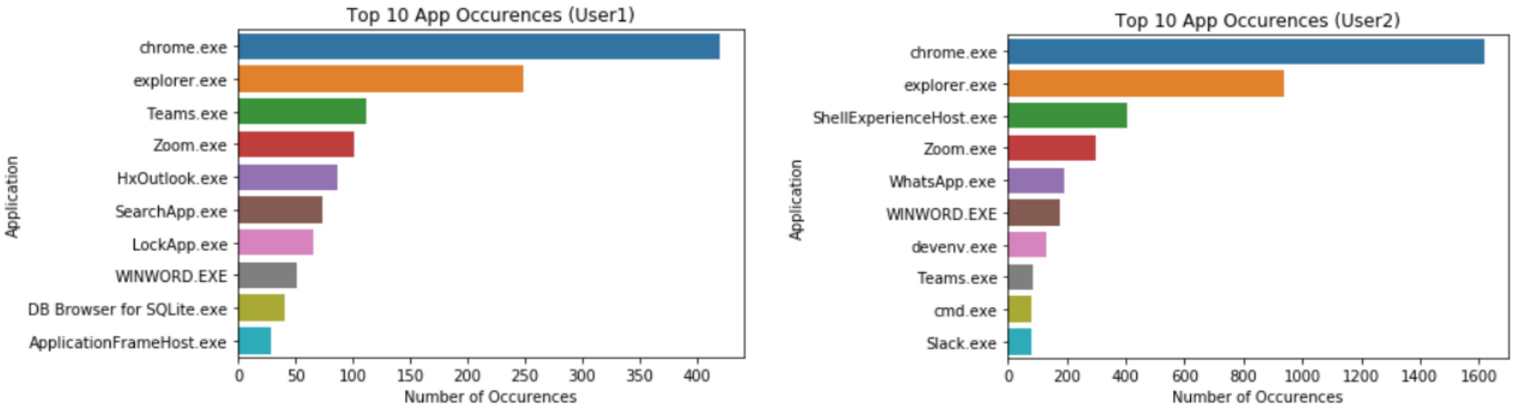
In the desktop mapper, we noticed that our collector had a shadow period between the data collection time and the logging time. Since we're iterating through all desktop windows, it takes time to perform the data collection. And during that time, we might be changing between several foreground windows. This results in backed up logs with wrong timestamps, and a similar duplicate issue as we faced earlier. Fixing this will require us to spend a bit more time optimizing the collector.

Exploratory Data Analysis

We analyzed the collected foreground_window library data from two separate devices (referred to as user1 and user2) as it was used for our predictive models. We did not use the mouse_input.IL and user_wait.IL data since it did not pertain to our problem statements. Instead, we focused on predicting app sequences and usage times. Mouse positions and the amount of time spent on mouse cursors are therefore irrelevant. Data from the desktop mapper was not reliable enough to examine closely.

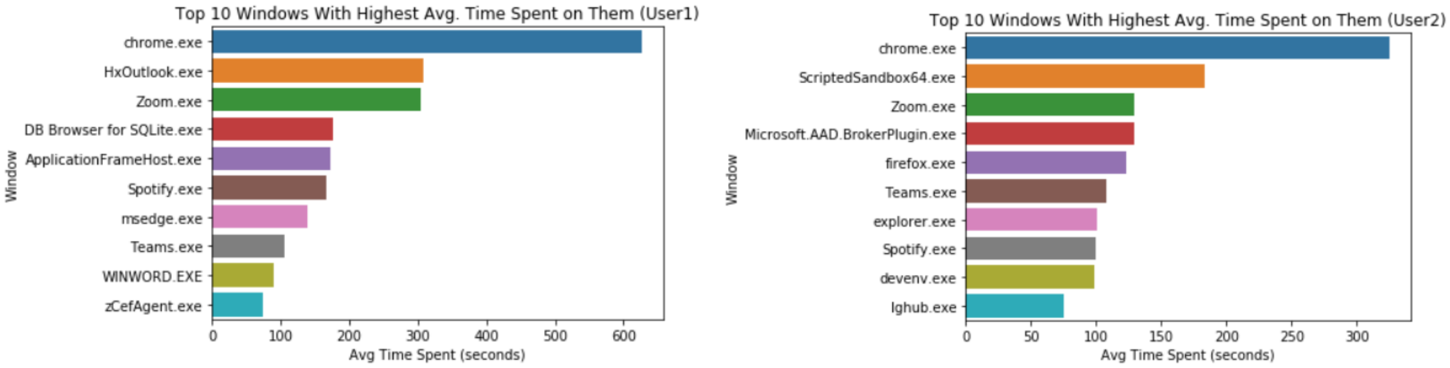
Both users have been collecting data since late november but we decided to only train and test our models on data starting from January 3rd, which is the first day of school. This way, we manage skews in usage behaviors and can fairly contrast the two app usage profiles. For both users, we cleaned the data by first dropping the ID_INPUT and PRIVATE_DATA columns and renaming MEASUREMENT_TIME to "time" and VALUE to "window", "is_immersive", "next_window", etc, depending on the feature in question. We then transformed the "time" column from a string to a date_time object. In the foreground window output table, we calculated a "time_spent" column for each app by subtracting consecutive row "time" values. In the window data, we observed some duplicate windows which occurred sporadically whenever users split screens or reshaped the window size. We simply dropped those rows as they account for the same app executable. For the immersive data, we noticed a few NaN values, which we replaced with 0.

Below are a series of visualizations and corresponding insights:



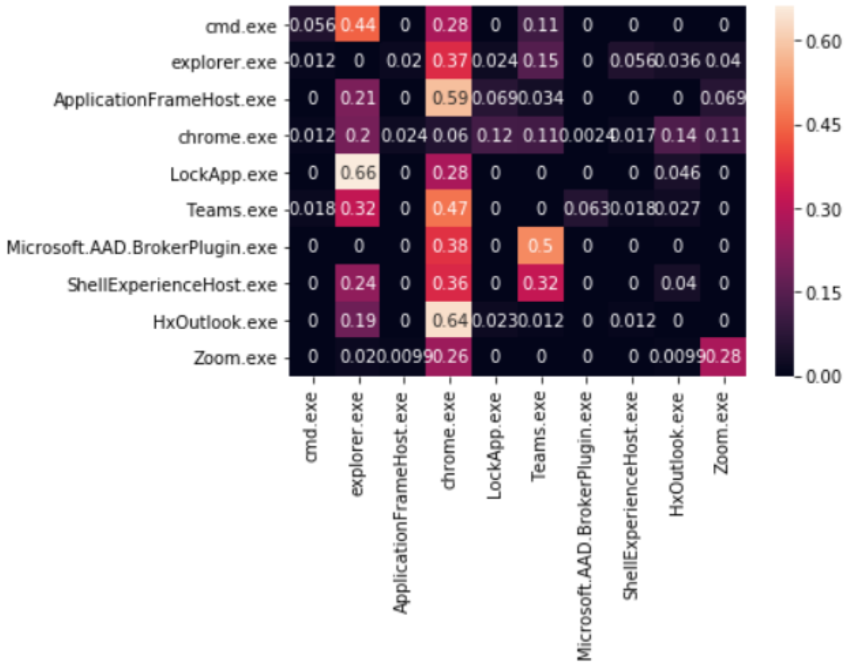
Top 10 Most Occuring Apps by User

Insights: Most of the daily activity for both users revolved around chrome. Chrome made up 30% and 35% of user 1 and user 2's data, respectively. The second most common app, explorer.exe or the file explorer, made up around 18% and 20% of user 1 and user 2's data, respectively. The remaining app windows constituted less than 10% each. User 1 had a total of 32 unique apps while user 2 had 25.



Top 10 ***Average Time*** *Spent on a Window by User*

Insights: These plots represent the top 10 average amount of time (in seconds) spent on a window over the course of the data collection period for each user. Outliers of 9+ hours (>15,000 seconds) were eliminated since they represented idle times.



Conditional Probability Transition Matrix Heatmap (User1)

Insights: This heatmap represents the transition matrix for our HMM. The elements are the conditional probability of the column window appearing after the row window. The dark values represent low switching probabilities while lighter colors represent higher probabilities.

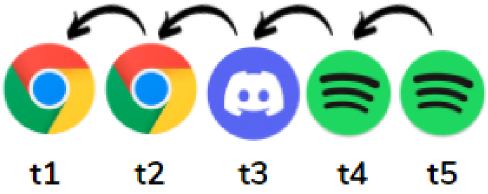
Prediction Overview

We came up with 2 prediction problems to better understand app usage behaviors, using 3 weeks of our collected time-series window data starting from January 3rd 2022.

The first problem is to identify patterns in app launches.

The HMM is best suited for our first problem as it allows us to understand the progression of events based on conditional probabilities. It works on the assumption that the future state relies on the current state which is important considering we are using sequential window data. We implemented three predictors using different heuristics for measuring accuracy to see how they compared with each other.

The second problem is to predict app usage durations and corresponding app sequences.



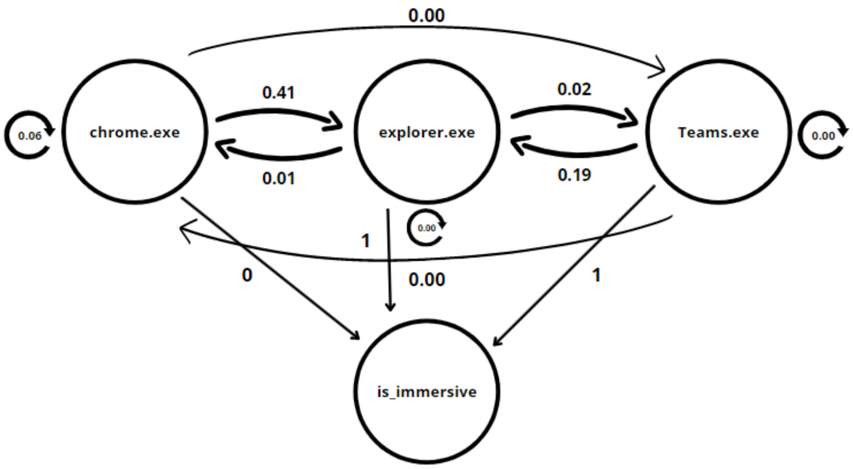
The LSTM is appropriate for our second solution since it's typically used for forecasting problems without being affected by the vanishing gradient problem where convergence happens too quickly. For the second prediction task, we attempted two approaches to the problem: a univariate regression and a multivariate classification.

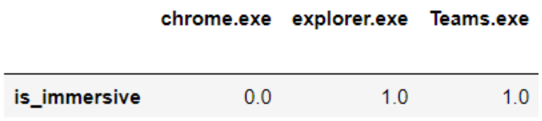
Hidden Markov Model

The Hidden Markov Model is a statistical model that we used to predict an app launch based on the sequence of prior app launches. This model uses a Markov Chain, a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. In our case, we will be basing the conditional probabilities on the sequence of possible *apps*.

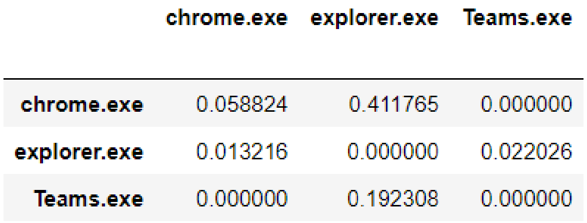
We utilized data from foreground_window.IL which produced window data as well as is_immersive data, all with timestamps. Since there were no apps that were unresponsive, the is_hung column was dropped. We did not delete the duplicates as it was important to see if an app would be opened after closing that same app. Feature engineering for the HMM involves the construction of transition and emission matrices. The transition matrix is an n x n matrix where n is the number of unique windows. The values in this matrix are the conditional probabilities of switching from one app window to the other.

The emission matrix is a 1 x n matrix where n is the number of unique windows and the values contain a feature of each window. In our case, that feature is whether the window is immersive (belongs to the windows app store). This matrix allows us to enhance the accuracy of our model. The HMM can then be visualized as a as seen below:





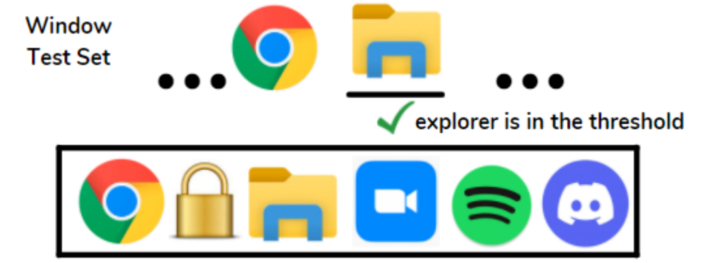
	chrome.exe	explorer.exe	Teams.exe
is_immersive	0.0	1.0	1.0



	chrome.exe	explorer.exe	Teams.exe
chrome.exe	0.058824	0.411765	0.000000
explorer.exe	0.013216	0.000000	0.022026
Teams.exe	0.000000	0.192308	0.000000

We proceeded to build 3 predictors for our HMM.

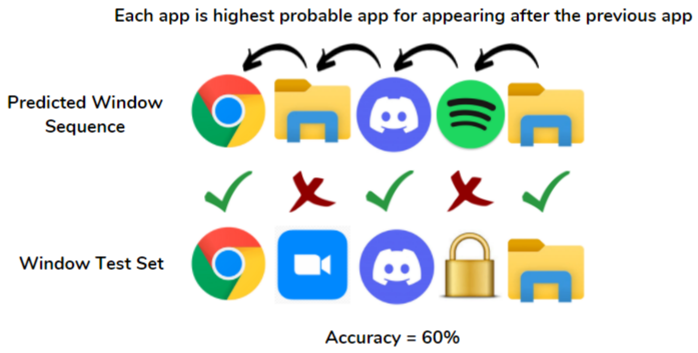
Predictor 1



Threshold of apps with highest probability of appearing after Chrome

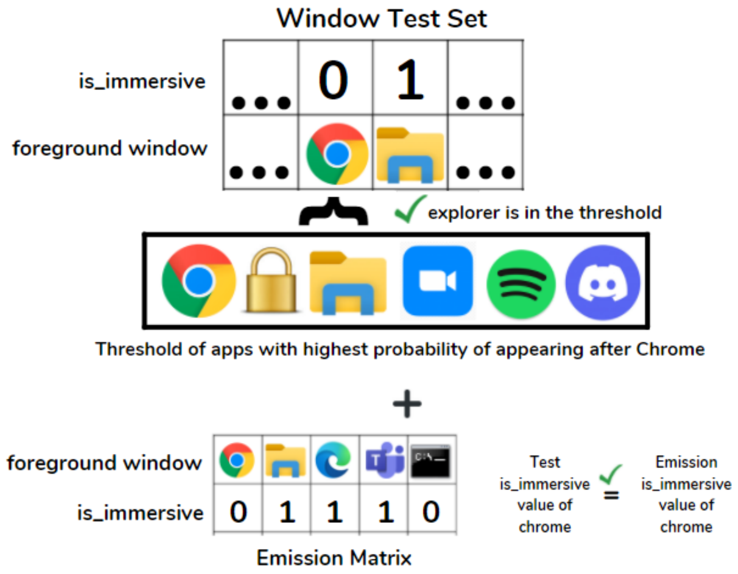
For each test window in the series, the first predictor lists the top 6 (threshold value) most probable windows from the transition matrix. If the next test window is one of these 6 predicted windows, it is assumed to be a correct prediction. A tally of correct and incorrect predictions is maintained at each iteration and an accuracy percentage is outputted based on the final tally.

Predictor 2


Each app is highest probable app for appearing after the previous app
Predicted Window Sequence
Window Test Set
Accuracy = 60%

On the other hand, the second predictor returns a sequence of predicted windows. Starting from the first test window, this predictor predicts the next window to be that with the highest probability in the transition matrix. This process is continued iteratively for each test window, and a sequence of windows is then returned. The test accuracy is measured by comparing the predicted sequence output to the test sequence of windows.

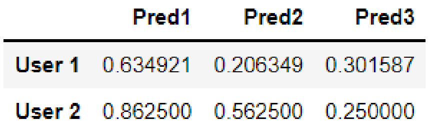
Predictor 3


Window Test Set
is_immersive
... 0 1 ...
foreground window
explorer is in the threshold
Threshold of apps with highest probability of appearing after Chrome
+
foreground window
is_immersive
0 1 1 1 0
Emission Matrix
Test is_immersive value of chrome
=
Emission is_immersive value of chrome

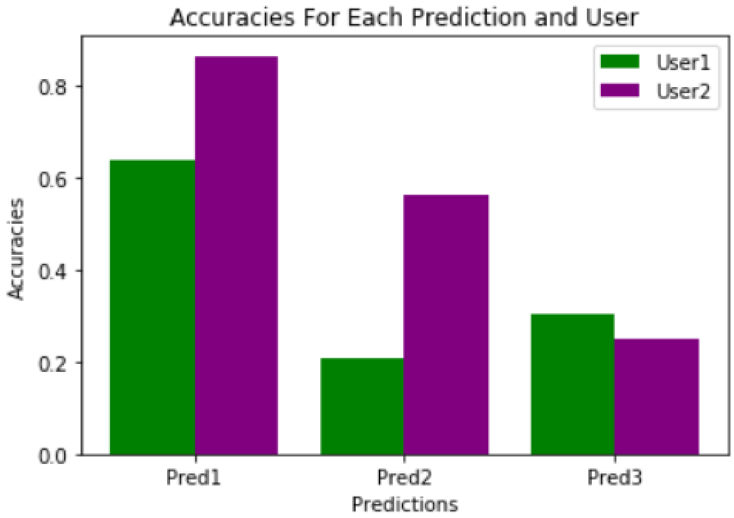
The third predictor is the same as the first with one exception. It uses the emission matrix as well as the transition matrix to assess correct matches. This means that the predicted is_immersive value must also match the test is_immersive value. If either of the conditions are not met, the incorrect tally is incremented in the accuracy calculation. We intend to include more observed variables from desktop mapper outputs in coming weeks which will increase the number of rows of the emission matrix. Although doing this will increase the conditions for a window to “pass”, the test accuracy will be validated further.

Results

Since we had three predictors for the HMM, we ran all of them on the data and yielded the accuracy table presented below:



	Pred1	Pred2	Pred3
User 1	0.634921	0.206349	0.301587
User 2	0.862500	0.562500	0.250000



While the HMM was a good model to use in this scenario, there are drawbacks to using one. For example, the seasonality of the data is missing in the transition matrix representation. Because this is a first order HMM, it depends only on the previous application. The large size of the transition matrix for data collection periods of several weeks makes feature engineering a relatively time and memory expensive model to use.

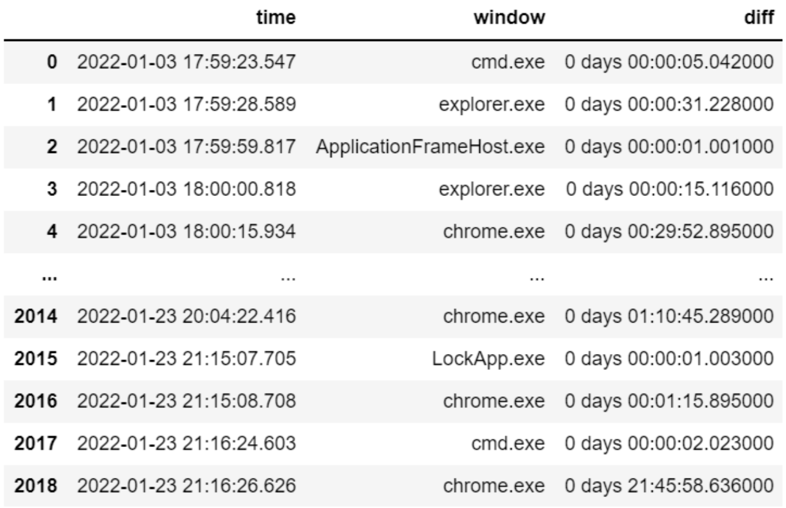
LSTM Model

The Long-short term memory model, otherwise known as the LSTM, is a recurrent neural network that is well suited for time-series forecasting and sequence prediction problems. The LSTM is structured on memory blocks that are connected through layers. It uses backpropagation of layers and time-step lookback values for model training.

Before we trained the model, we performed data cleaning on our datasets for two users. Since we had sporadic occurrences of duplicate windows occurring when apps were in split screen or were resized, we dropped consecutive duplicates since the two windows account for the same app executable. We also removed anomalous executables that appeared only once and never again such as app installers or random popups.

The remaining cleaning steps are enumerated below:

1. Convert ‘time’ from string to datetime
2. Calculate time difference between each row
3. Drop unnecessary columns ('ID_INPUT', 'PRIVATE_DATA')
4. Select 3 week time frame
5. Remove windows appearing only once
6. Drop NULL values



	time	window	diff
0	2022-01-03 17:59:23.547	cmd.exe	0 days 00:00:05.042000
1	2022-01-03 17:59:28.589	explorer.exe	0 days 00:00:31.228000
2	2022-01-03 17:59:59.817	ApplicationFrameHost.exe	0 days 00:00:01.001000
3	2022-01-03 18:00:00.818	explorer.exe	0 days 00:00:15.116000
4	2022-01-03 18:00:15.934	chrome.exe	0 days 00:29:52.895000
...	...	...	...
2014	2022-01-23 20:04:22.416	chrome.exe	0 days 01:10:45.289000
2015	2022-01-23 21:15:07.705	LockApp.exe	0 days 00:00:01.003000
2016	2022-01-23 21:15:08.708	chrome.exe	0 days 00:01:15.895000
2017	2022-01-23 21:16:24.603	cmd.exe	0 days 00:00:02.023000
2018	2022-01-23 21:16:26.626	chrome.exe	0 days 21:45:58.636000

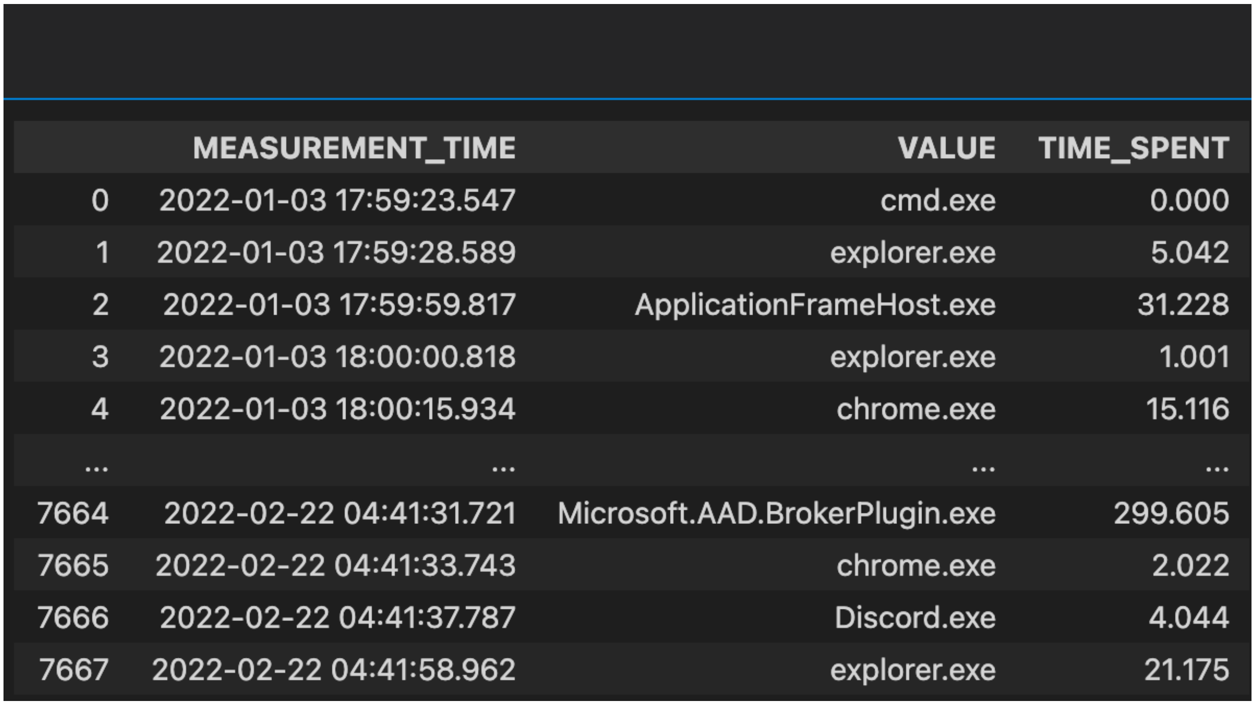
Cleaned foreground window table

Univariate Regression

Our first implementation of the LSTM model forecasts the time spent on a certain app by the hour. The architecture of this solution consists of a stacked LSTM model which helps with adding a level of abstraction to input observations over time. The model is trained on a single feature for each app denoting the time spent across the period of data collection. Preprocessing steps include removing duplicates and anomalies by using a 3 sigma approach verified by an LSTM Reconstruction Loss anomaly detection approach. Hyperparameter tuning was performed using GridSearch where we used all available loss functions, various values for the number of layers/nodes. We have explored results generated by using data from different users as well as time periods.

Data Preprocessing

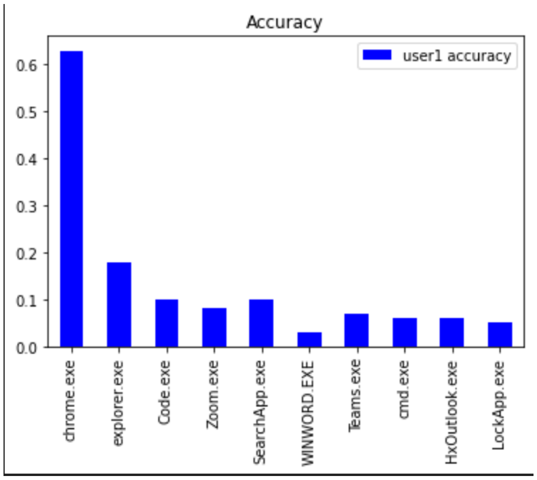
The initial collected data has MEASUREMENT_TIME , ID_INPUT, VALUE, and PRIVATE_DATA columns as specified by the telemetry collectors. ID_INPUT and PRIVATE_DATA are dropped due to there not being any relevant information in the columns. A third feature TIME_SPENT was computed using the datetime differences between MEASUREMENT_TIME of each foreground window represented by column VALUE. Since the LSTM model takes in TIME_SPENT values per application as input, the data is aggregated based on the usage times per app. The numeric values in TIME_SPENT are also scaled between 0 and 1 using a MinMax scaler.



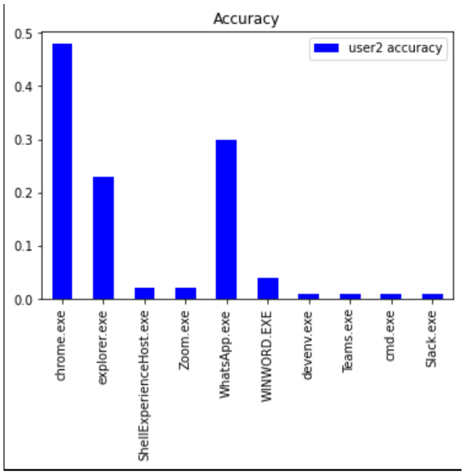
	MEASUREMENT_TIME	VALUE	TIME_SPENT
0	2022-01-03 17:59:23.547	cmd.exe	0.000
1	2022-01-03 17:59:28.589	explorer.exe	5.042
2	2022-01-03 17:59:59.817	ApplicationFrameHost.exe	31.228
3	2022-01-03 18:00:00.818	explorer.exe	1.001
4	2022-01-03 18:00:15.934	chrome.exe	15.116
...	...	...	...
7664	2022-02-22 04:41:31.721	Microsoft.AAD.BrokerPlugin.exe	299.605
7665	2022-02-22 04:41:33.743	chrome.exe	2.022
7666	2022-02-22 04:41:37.787	Discord.exe	4.044
7667	2022-02-22 04:41:58.962	explorer.exe	21.175

Results

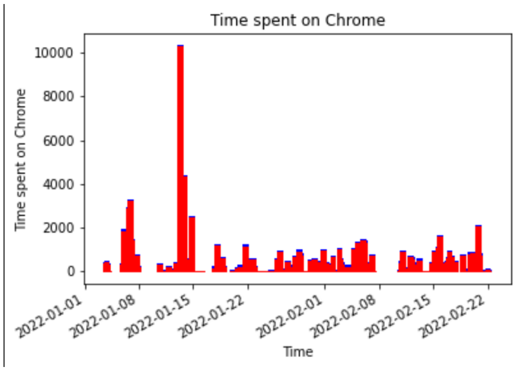
The Stacked LSTM model forecasts the amount of time spent per hour on a particular app. TIME_SPENT values corresponding to the particular app are aggregated to be split into train and test sets. The reported accuracy for TIME_SPENT predictions on training and testing sets are combined for the top 10 apps below. The discrepancy in accuracy values can be attributed to the differences in app usage in each dataset with there being 32 distinct apps for User1 and 25 distinct apps for User2. The accuracy is set to a precision of +/- 10 seconds.



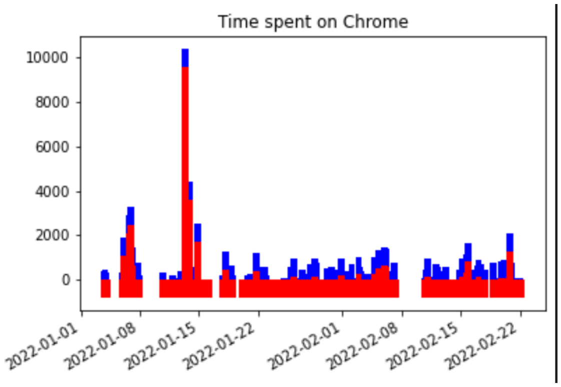
User 1 Accuracy



User 2 Accuracy



User 1 Chrome Usage



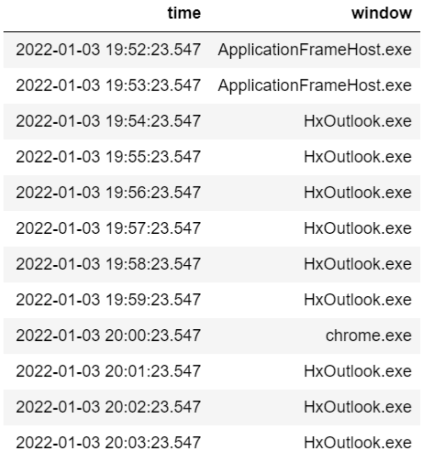
User 2 Chrome Usage

Since the predictions on Chrome were better than that of a coin toss, we will focus on forecasting results for Chrome data. Performance for other apps did not cross this threshold. Due to the significantly low usage of other apps compared to Chrome by both users, the data points were not sufficient to make a reasonable estimate of usage time. The red bars in the figures above represent the prediction values while the blue bars indicate the training and testing data.

Multivariate Classification

We came up with the second classification approach which predicts app sequences. But since we made the decision to log foreground window data at irregular time intervals, determining the timestamps associated with our predictions was tricky. We made the logging decision to suit the regression approach at the time, so overcoming this problem for classification required us to transform our data to regular time-steps.

This was done iteratively such that if there were no windows in the target time-step, we appended NaN. If there was only one window in the time-step, it was appended. If there were multiple windows opened during a particular time-interval, we logged the window that the user spent the most time on during that time interval. A data frame was then tabulated, and the NaNs were forward-filled to account for missing values. In the data shown below, the target time-step is 1 minute.



time	window
2022-01-03 19:52:23.547	ApplicationFrameHost.exe
2022-01-03 19:53:23.547	ApplicationFrameHost.exe
2022-01-03 19:54:23.547	HxOutlook.exe
2022-01-03 19:55:23.547	HxOutlook.exe
2022-01-03 19:56:23.547	HxOutlook.exe
2022-01-03 19:57:23.547	HxOutlook.exe
2022-01-03 19:58:23.547	HxOutlook.exe
2022-01-03 19:59:23.547	HxOutlook.exe
2022-01-03 20:00:23.547	chrome.exe
2022-01-03 20:01:23.547	HxOutlook.exe
2022-01-03 20:02:23.547	HxOutlook.exe
2022-01-03 20:03:23.547	HxOutlook.exe

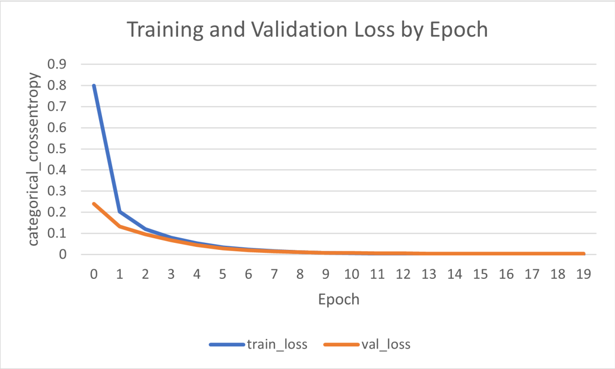
Regularized Data Table: 1min time-step

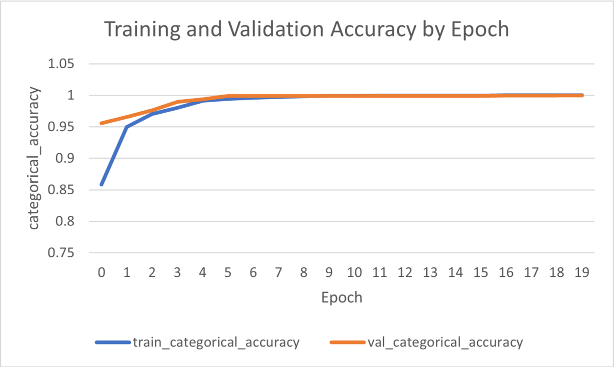
This results in a slight loss of accuracy to the original data, but since our selected time intervals are so small, that loss turned out to be negligible. To have zero loss in regularization, we would actually have to go back and change the foreground window input library to log data at definite time intervals. However, we only came up with this custom approach in the second half of the quarter so we couldn’t collect an additional 3 weeks of data on our PCs.

Next, we had to transform the data so it would be accepted by the LSTM for classification. We label-encoded and one-hot encoded our window strings before the data was split into training (60%), testing (20%), and validation (20%) sets, each of which had to be engineered for a supervised learning problem. This is where we tested different look_back values which dictate how many previous sequence elements to use to predict another sequence. When the look_back is 3, a sequence [1,2,3] may predict [4,5,6]. The next training example [2,3,4] predicts [5,6,7] and so on. For our model, a sequence of one-hot vectors was used to predict another sequence of one-hot vectors in this fashion. The predictions were then decoded to give us back the predicted app executable name labels.

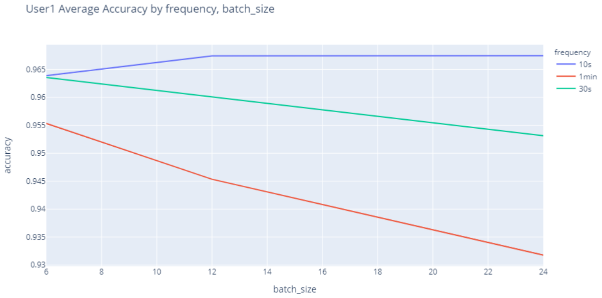
The model itself was a Keras Sequential model. It has one LSTM layer that returns sequences and an activation layer using softmax activation for multiclass prediction. The model was compiled using the categorical_crossentropy loss and categorical_accuracy as the evaluation metric. We chose the time_interval, look_back, number of nodes, and batch_size for hyper-parameter tuning, resulting in 81 different models trained separately for each user's data. We tested the following hyper-parameters:

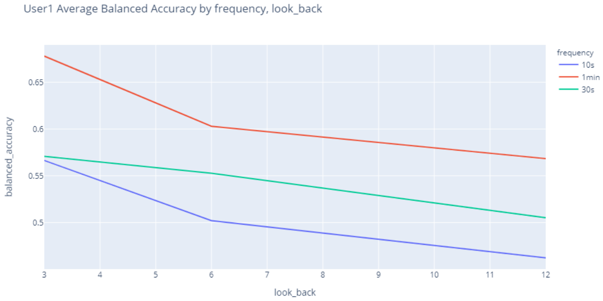
- time intervals = [10s, 30s, 1min]
- look_back values = [3, 6, 12]
- node values = [16, 32, 64]
- batch_size values = [6, 12, 24]

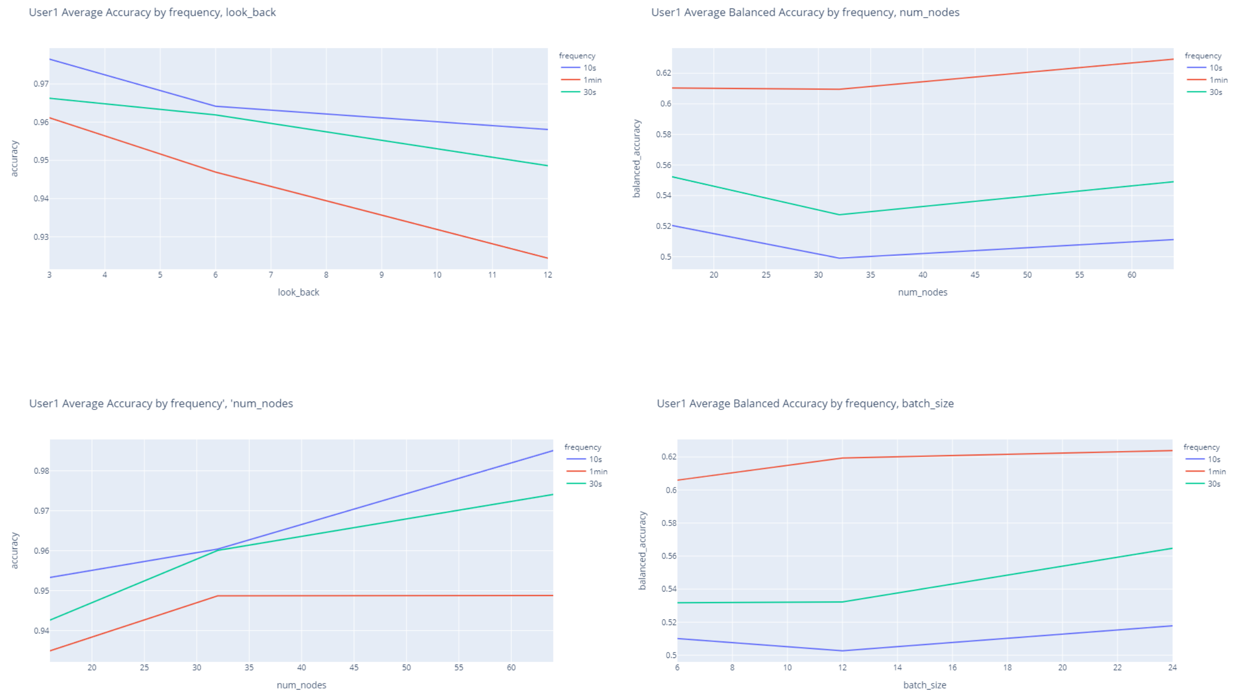




We trained our models for 10 epochs as we found it to be optimal from the training, validation loss and accuracy curves. 10 epochs is the point where the validation loss became smaller than the training loss. To prevent overfitting, we should not run the model for further epochs.







User1 Average Accuracy by frequency, look_back

User1 Average Balanced Accuracy by frequency, num_nodes

User1 Average Accuracy by frequency', 'num_nodes

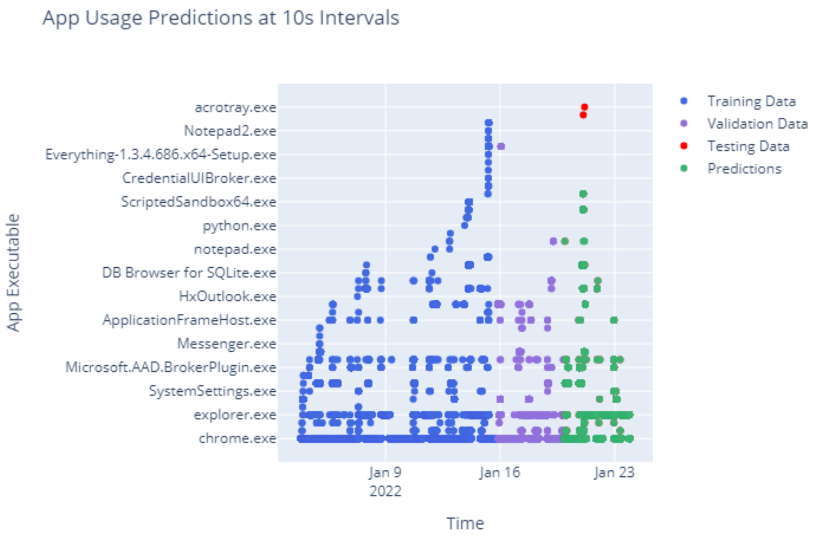
User1 Average Balanced Accuracy by frequency, batch_size

Average and Balanced Accuracy scores by frequency group and hyper-parameters

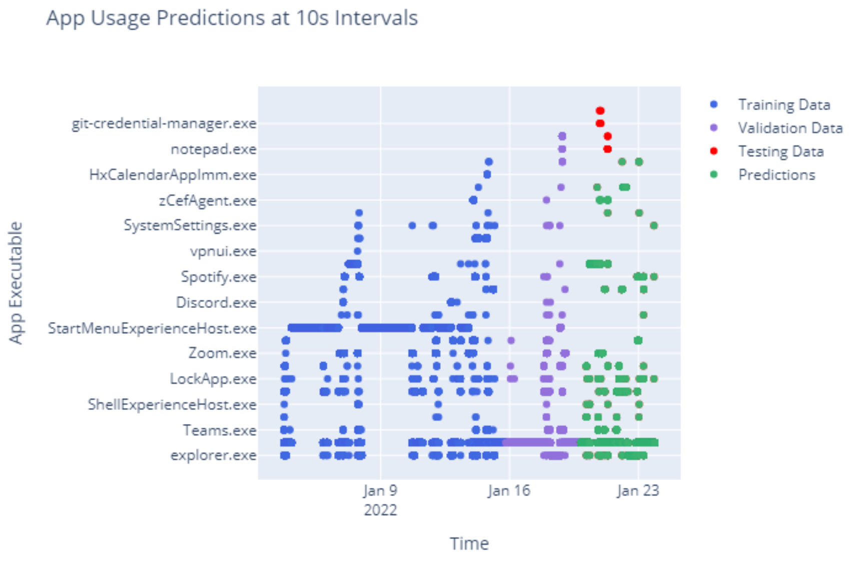
Here we see how each parameter affected model accuracy and balanced accuracy scores for each frequency group. First we note that the 10s frequency group outperforms all others when it comes to accuracy scores while the 1min frequency group does the same for balanced accuracy scores. Next, we observe that a smaller look_back and batch_size give us higher accuracy scores. More nodes also give us higher accuracy scores. The same is true for balanced accuracy scores with the exception of batch_size, which does the opposite. We were accordingly able to deduce the best parameters for our model.

We wrote code to test all 81 hyper-parameter combinations for both user's data. This code programmatically saves each model's artifacts and training logs to a csv file, as well as the output accuracy scores by parameter combination. Additionally, the code programmatically saves plots describing app usage behavior for each user overlaid by model predictions. These results can be seen in the following section.

Results



User 1 Best Model Predictions

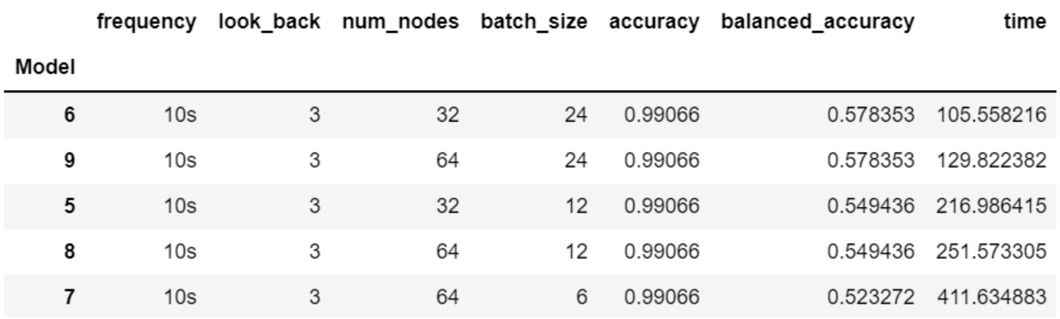


User 2 Best Model Predictions

These graphs summarize app usage behavior for both users. On the x-axis we have the data collection time period of 3 weeks over 10s intervals. On the y-axis, we have the app executable used at each time-step. Each dot therefore represents the app used in a particular 10s interval. Here you can see the entire dataset split by training, validation, and testing sets, which is then overlaid by our best predictions for each user. You can see the effectiveness of our approach by looking at the green prediction dots which almost completely overlay the red testing dots.

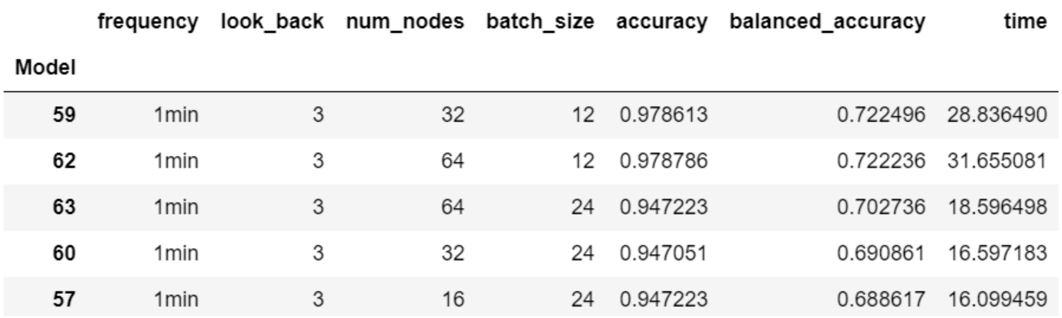
We evaluated our models using several evaluation criteria: the accuracy, balanced_accuracy, weighted f1 score, weighted precision, and weighted recall. We limit our discussion to the accuracy and balanced accuracy scores for this report. The accuracy score is straightforward: of all the predicted windows, what percentage matched the unseen test windows? The balanced accuracy makes a different assumption. It assumes that each class label is equally important. It calculates the prediction accuracy for each individual app, and averages them to give one score. Since our data was heavily imbalanced, we figured both these scores would provide context for our results.

User 1



	frequency	look_back	num_nodes	batch_size	accuracy	balanced_accuracy	time
Model							
6	10s	3	32	24	0.99066	0.578353	105.558216
9	10s	3	64	24	0.99066	0.578353	129.822382
5	10s	3	32	12	0.99066	0.549436	216.986415
8	10s	3	64	12	0.99066	0.549436	251.573305
7	10s	3	64	6	0.99066	0.523272	411.634883

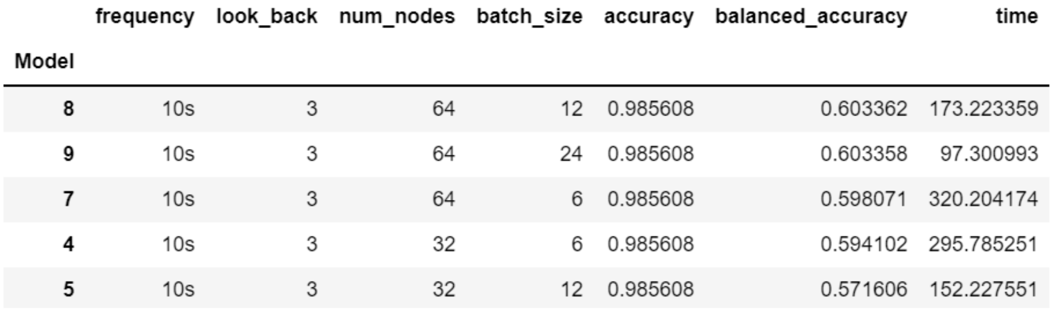
Top 5: Accuracy Score



	frequency	look_back	num_nodes	batch_size	accuracy	balanced_accuracy	time
Model							
59	1min	3	32	12	0.978613	0.722496	28.836490
62	1min	3	64	12	0.978786	0.722236	31.655081
63	1min	3	64	24	0.947223	0.702736	18.596498
60	1min	3	32	24	0.947051	0.690861	16.597183
57	1min	3	16	24	0.947223	0.688617	16.099459

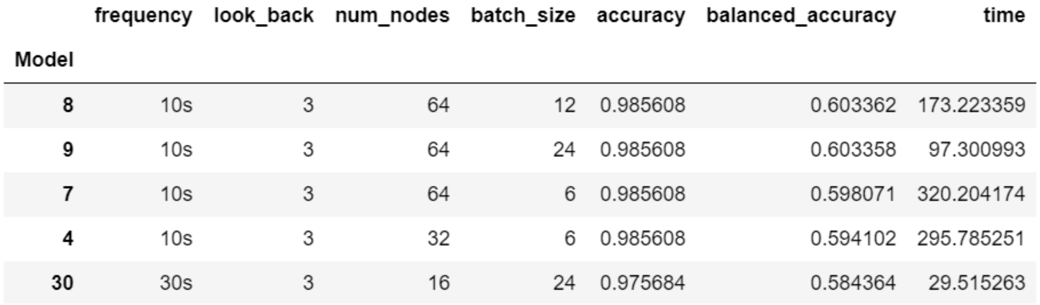
Top 5: Balanced Accuracy Score

User 2



Model	frequency	look_back	num_nodes	batch_size	accuracy	balanced_accuracy	time
8	10s	3	64	12	0.985608	0.603362	173.223359
9	10s	3	64	24	0.985608	0.603358	97.300993
7	10s	3	64	6	0.985608	0.598071	320.204174
4	10s	3	32	6	0.985608	0.594102	295.785251
5	10s	3	32	12	0.985608	0.571606	152.227551

Top 5: Accuracy Score



Model	frequency	look_back	num_nodes	batch_size	accuracy	balanced_accuracy	time
8	10s	3	64	12	0.985608	0.603362	173.223359
9	10s	3	64	24	0.985608	0.603358	97.300993
7	10s	3	64	6	0.985608	0.598071	320.204174
4	10s	3	32	6	0.985608	0.594102	295.785251
30	30s	3	16	24	0.975684	0.584364	29.515263

Top 5: Balanced Accuracy Score

We achieved a stunning **accuracy score** of **99% for User 1** and **98.5% for User 2.** We achieved a more moderate **72% balanced accuracy for User 1** and **60% for User 2.** This may be because User 2 had higher relative chrome usage resulting in more imbalanced data.

Conclusions

For the past 20 weeks we have developed various data collectors modules like mouse input, foreground window, user wait, and desktop mapper to collect data for our specific use case. Using the collected data, we built predictive models using autoencoders like LSTM and generative models such as Hidden Markov Models. The high accuracy of our predictive models imply that they can be used to effectively infer the sequences of apps used on a daily basis and how long each app is used. The design decisions made during the process which include using the memory management capabilities of C for the data collectors and using RNNs for our predictions were mainly for scalability and future expansion goals.

Real-World Implications

The implications of our multivariate LSTM results are massive. Successfully forecasting app usage behavior means that we can further optimize the PC experience. Knowing when an app will be launched by the user means we can launch it preemptively in the background and largely eliminate that dreaded spinning wait cursor. And knowing how long apps will run would allow us to optimize system resources such as battery and memory. This solution can be very beneficial especially on low spec Windows machines which cannot have daily used apps constantly open in the background.

Future Plans

A typical data scientist rarely has control over the data collection process as the task is delegated to software engineers who build tools from which data is extracted. Since we built our own collector modules we were able to define the exact inputs needed for our model, control the polling frequency of the data, and define custom wait signals for data collection. Taking upon the role of software engineers, we plan on modifying our existing foreground window collector to log results at regular time-series instead of mouse click burst events to eliminate lossy regularization. We also want to tune our desktop mapper collector to provide more reliable results. As data scientists we also want to enhance our multivariate LSTM and add new features to the emission matrix for HMM to make our predictions more robust. To scale the existing solution we must also take upon the role of an ml engineer. Due to potential privacy concerns of collecting sensitive information, we plan on generating a unique hash for each user's data and using it as a primary key in our databases. Furthermore due to the large memory requirements of running lstm, we also propose developing a cloud based analytics system which streams data to a data lake like S3 and use other computational resources like EMR and Sagemaker to deploy training and inference jobs.