Inferring Mobile User Activities via. Device-Side Cellular Network Traces via. Machine Learning Approaches

Christopher Nosowsky and Hamzeh Alzweri

Abstract

With recent developments in internet technology and an increasing use of mobile smartphones, there exists a need for optimization of cellular networks through the means of traffic classification. By utilizing a tool, MobileInsight [1], we can monitor cellular network traces across different applications with the goal of successfully classifying user activities. Our classification will be determined by a machine learning model, which will learn to classify applications into pre-determined categories such as tooling, shopping, entertainment, and social media.

Introduction

When it comes to advances in technology, it is no surprise that there have been major advancements in mobility and cellular network technology. Speed and coverage have never been more important in a generation, as many are on the go or utilizing cell phones to access documents, join conferences, and perform critical tasks over the internet. This is projected to grow, reaching 77.5 exabytes per month worldwide, an annual increase of 46 percent [2]. With this increase, we are looking for ways to not just expand the infrastructure for user support, but also looking into ways of optimizing the network operations. The problem, however, is that access is limited to the runtime operations on cellular protocols. The lack of access has created many blockers for researchers and developers looking to refine the process of network operations and understand how cellular protocols operate. We will not be able to advance our understanding in this closed cellular network system. As an example of our lack of understanding, let us take a phone trying to upload a photo. If the photo takes a long time to upload, we cannot fully understand if this is caused by network issues or just a poor cellular reception. By having this access to the protocols design, we can perform various functions that can help improve our infrastructure's performance and optimization. This would be especially beneficial to businesses, personal, internet service provider (ISP), and government networks who are looking at ways to support large amounts of users and utilize the most resources at particular times of the day. We could further classify applications with such data to help improve our network traffic quality. It would be nice to have a way of exploring such problems in the network infrastructure by providing us with a tool that can be built and shared across devices. One such tool can allow us to access cellular network traces and be able to see the trace data being sent and received between the different protocol layers. If we are able to extract this information, then we could further refine the features or data that is meaningful, to determine what user activities are being utilized the most. This can then lead to solving network problems such as capacity planning, faulty diagnosis, application performance, anomaly detection, and trend analysis [3]. Furthermore, classification of applications shows its benefits. The first way it is beneficial is by helping with the Quality of Service on the network. Resources are limited, so being able to prioritize different applications properly can help avoid common issues like the network congestion problem. It could also help with analysis of traffic in performing trend analysis. This would be especially

beneficial for internet service providers checking user activities throughout the day to determine peak and low utilization times so that they can build their infrastructure accordingly.

Problem Description

The problem entails that internet service providers are not able to effectively build and plan for capacity load on their networks without properly knowing what happens beyond just the cellular interface. Commodity phones with an operating system and applications have very limited access to such lower-level cellular information at runtime. The operating system tends to abstract such fine grain information to reduce complexity. To help internet service providers and businesses solve network problems, we will be introducing a machine learning approach that can help by providing a classification of user activities in the network based on user cellular trace data. Using MobileInsight, we can achieve proper user activity classification. MobileInsight gives us access to below-IP protocol messages to obtain low-level network information. Using this tool, we can record and monitor cellular traces that come in through each application. For our applications, we will be choosing twenty applications that are common on the Google Play store [4]. In Table 1 below, we list the applications that we ended up using. For our considerations, we have decided to limit our user activity choices into four distinct categories: Shopping, Tooling, Entertainment, and Social. Each category chosen has distinctive properties that will be justified in our section describing our machine learning approach. We ensured each category was distributed with at least three applications to collect data evenly.

Shopping	Tooling	Entertainment	Social
Airbnb	Gmail	IMDB	LinkedIn
Doordash	Google Chrome	Twitch	New York Post
Macys	Google Maps	Youtube	Pinterest
McDonalds	Google Play		Reddit
Starbucks			Snapchat
Walmart			Twitter
Zillow			

Table 1. Chosen applications

Challenges

When approaching our solution, we came across various challenges. One of the first challenges we dealt with was device support for the MobileInsight tool. It is great to have such support to begin with, but the devices this application was working on were limited to the devices listed in Table 2 below. While there were not many devices, we ended up getting access to a Google Pixel 3 with a Qualcomm Snapdragon chipset. This device is in the list of compatible phones pulled from the MobileInsight website [1].

Compatible Phones		
XiaoMi Mi 9	ZTE Obsidian	LG G3
Lenovo Z5 Pro GT	Huawei Nexus 6P	Samsung Galaxy Note 3
Google Pixel 3/3 XL	ZTE Nubia Z9	LG Tribute
XiaoMi Mi 8 SE	Samsung Galaxy S5	Sony Xperia Z3
Google Pixel 2/2 XL	Samsung Galaxy S4	Motorola Nexus 6
Google Pixel XL	Sony Xperia Z3	
Huawei Pixel	Xiaomi Mi 4	

Table 2. Phones compatibility

Another challenge we came across was the lack of documentation for setting up MobileInsight. The website and GitHub repository did not provide many details on setting up the mobileinsight-core repository [5] that we had to utilize to access their modules in order for this project to work. We were hoping to be pointed in the right direction, for example, when it came to the location of the build folder after running the shell script for setting up the mobile insight graphical user interface. We were running into compilation errors, unable to locate library .so files when pointing to the mobileinsight-core source code folder. We also faced additional challenges when running through the tutorials in the documentation. Some source code was missing import statements. We were able to resolve all of these issues with no problem.

Another challenge we faced was that there was no one-to-one relationship between the cellular network traces we gathered for each application. For example, recording on the Twitter application resulted in more MAC layer packets versus RLC layer packets. This was a risk we accepted and we understood that we could not combine features across different packets when building our machine learning model. Instead, we went with choosing one packet for our model that produced the highest accuracy results across the applications.

Ensuring consistent recording time was another challenge we faced. In order to avoid human selection bias, we could not record on one particular application longer than another application. We set a five-minute timer to record each application so that we could gather enough data in a consistent fashion.

We also found that another challenge dealt with running concurrent applications when recording with MobileInsight. Having applications running the background caused cross-interference between the background and foreground applications, mixing packets from the two together. To avoid this, we made sure we had no background applications running prior to starting the recording for each application.

The last challenge we came across was that the MobileInsight API was only supported on MacOS and Linux systems. This left us with either setting up a Vagrant machine that they provided or setting up a custom build virtual machine that we can create. We went with the custom build option since we had experience in setting up virtual machines and wanted to test its compatibility on Ubuntu 20.04.

System Design

Figure 1 below shows a high-level view of our intended system design.

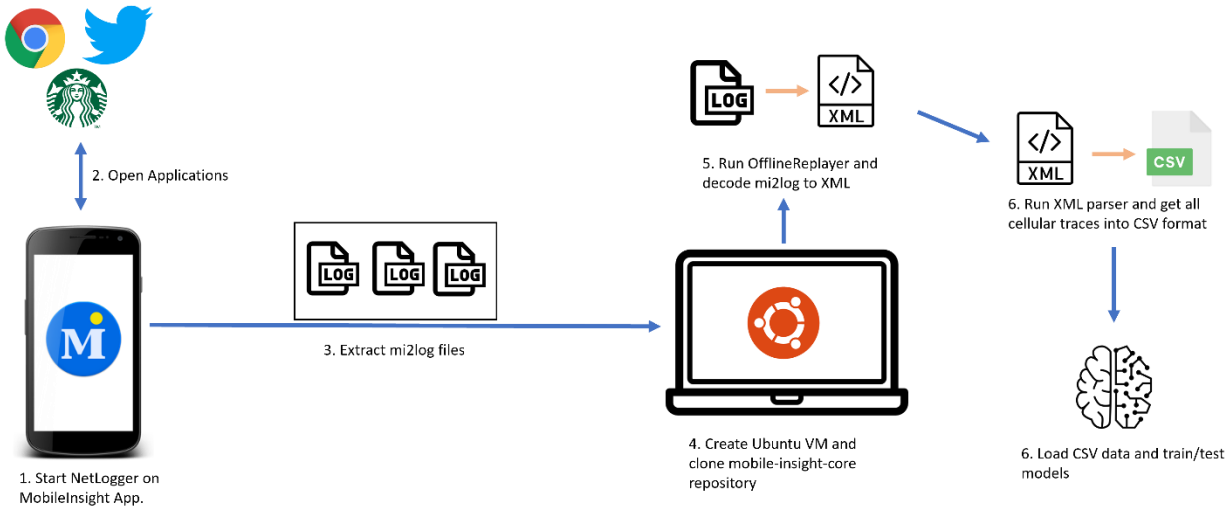


Figure 1. System design high-level view

The first step is to be able to download the MobileInsight application onto our mobile device. We will then start the NetLogger monitor on the application and ensure we went to the settings and checked to log all protocol packets. The second step is to individually open our applications per recording and perform normal tasks on each application to ensure we cover typical user behavior. We will perform this function for five minutes for each application. After we gather all the recordings, we extract the log files located in our device's storage. These log files have a mi2log file format. We made sure that we extracted each log file after each application recorded since all the cellular trace logs are stored in the same location and it would be hard to distinguish between the different applications logs. These log files got extracted and stored into our Ubuntu VM, where it hosts our mobileinsight-core repository. This is the repository that has the modules in Python for our project to utilize for running analyzers to look at the cellular trace data. For step 5, we utilize this repository and run the MobileInsight OfflineReplayer, which will read the mi2log files and parse it into any format we choose. We decided to decode our log files into XML format. Once we saved our log files into XML, we were able to create a Python script for parsing the XML into CSV format. This is the last format we convert it to since it can be read in by the Python Pandas library we use for our preprocessing. That is the last step, to load in our CSV data, preprocess the data, and send it into our machine learning model to learn the data. The result would be a highly versatile model capable of classifying user activities.

Project Setup

For the initial project setup, we downloaded the twenty apps listed in Table 1 above on the Google Pixel 3 device we intended to gather our network trace data on. These apps can all be found on the Google Play store. We also went to the MobileInsight website and downloaded their application. The next step for our setup was to open the MobileInsight app and create log files for 100KB of packet traces as well as log all type of packet traces. This would complete the setup on the mobile side. For the actual

development work to be done, we had to prepare our machines by setting up an Oracle VirtualBox virtual machine with Ubuntu 20.04. We got this image from the internet. Once all the operating system packages and updates were installed on the virtual machine, we installed the Git version control software so that we could clone the Mobileinsight-core repository to utilize the MobileInsight modules. We also had to setup our environment for the Python language to use in our project. We used Python 3.8.0 and the Pip package manager to install all dependencies. Finally, our project was created within the PyCharm IDE.

Data Collection

We used the MobileInsight mobile application to collect log files that contain the network traces we used the NetLogger plugin in the application to collect 4G LTE packet traces for 20 different applications. Many types of network traces can be logged e.g 5G_NR_RRC_OTA_Packet, LTE_RRC_OTA_Packet, etc. We allowed the NetLogger to log all type of network traces because we didn't know in advance which type of packet will help the machine learning model to get the best results.

The code below extracts the mi2log files that we collected from the NetLogger as XML files, we run it for each application log files separately:

```
if __name__ == "__main__":
    home = str(Path.home())

    for app in applications:
        print("Generating XML for application: " + app)
        # get all log files
        log_files = glob.glob(home + "/MobileInsightData/" + app + "/*")
        print(log_files)
        for i, file in enumerate(log_files):

            # Initialize a monitor
            src = OfflineReplayer()
            src.set_input_path(file)
            src.enable_log_all()

            dumper = MsgLogger()  # Declare an analyzer
            dumper.set_source(src)  # Bind the analyzer to the monitor
            dumper.set_decoding(MsgLogger.XML)  # decode the message as xml
            dumper.save_decoded_msg_as(home + "/MobileInsightData/" + app + "-xml/log-" + str(i) + ".txt")
            # Start the monitoring
            src.run()
```

Figure 2. mi2log to XML converter

The result is a folder filled with XML files that contain all types of network packets as an XML tag called <dm_log_packet>, each <dm_log_packet> represents a network packet that was logged by the mobile application, see figure 5. However, these files contain all types of packets in each log file and we need to rearrange these packets so that all packets of the same type are placed in the same file.

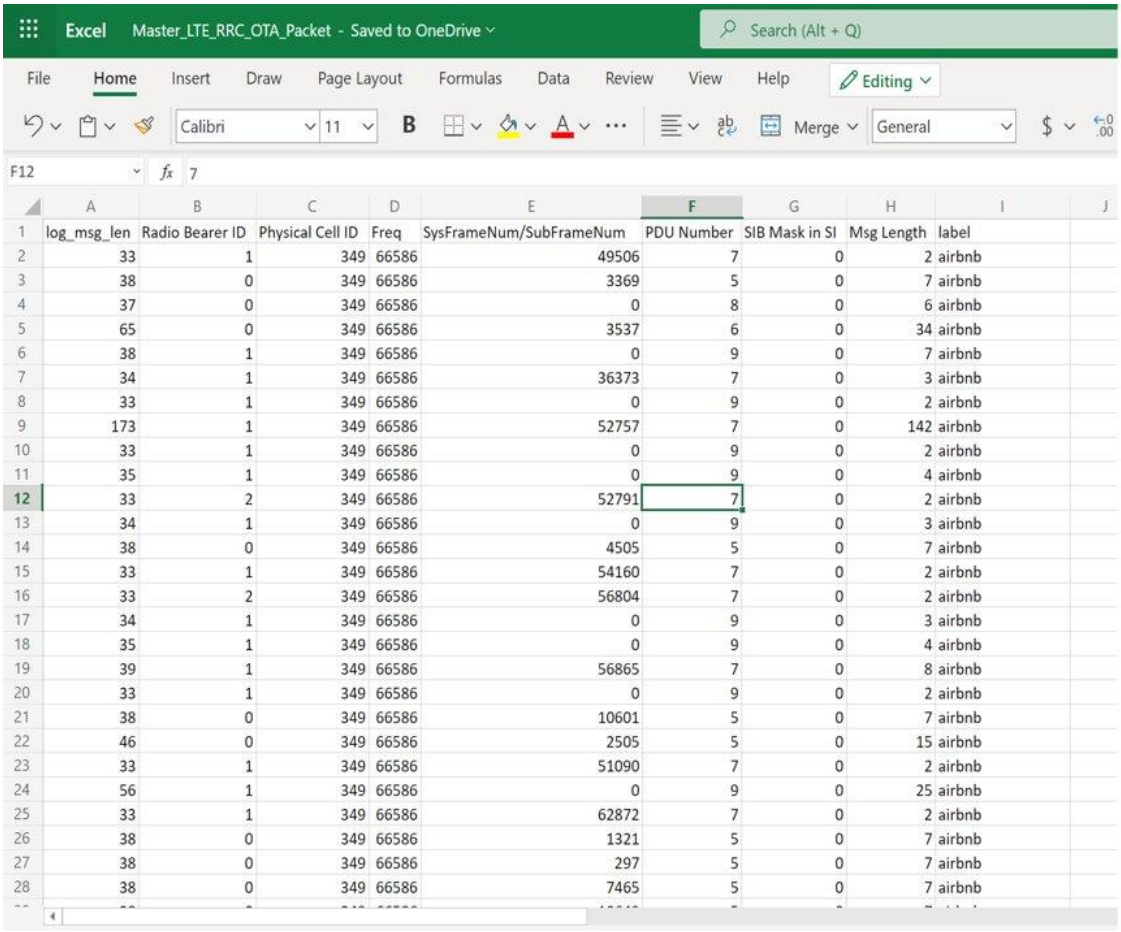
Figure 3. A network packet in XML format

The below function collects all packets of the specified type from all log files in a directory and saves them into a list.

```
def get_packets_with_type(packet_type, directory_path):
    logfiles = glob.glob(directory_path + "/*")
    target_packets = []
    for logfile in logfiles:
        with open(logfile) as file:
            # adding dummy root
            fileString = '<ROOT>{}</ROOT>'.format(file.read())
        root = ET.fromstring(fileString)
        packets = root.findall("dm_log_packet")
        for packet in packets:
            for children in packet:
                if children.get("key") == "type_id":
                    if children.text == packet_type:
                        target_packets.append(packet)
    return target_packets
```

Figure 4. Collecting packets of the same type

Next, we convert the list of packets into a csv file where each row represents a packet, we have to do this in order to preprocess the data and feed it into the model as a data frame for training.



	log_msg_len	Radio Bearer ID	Physical Cell ID	Freq	SysFrameNum/SubFrameNum	PDU Number	SIB Mask in SI	Msg Length	label
2	33	1	349	66586	49506	7	0	2	airbnb
3	38	0	349	66586	3369	5	0	7	airbnb
4	37	0	349	66586	0	8	0	6	airbnb
5	65	0	349	66586	3537	6	0	34	airbnb
6	38	1	349	66586	0	9	0	7	airbnb
7	34	1	349	66586	36373	7	0	3	airbnb
8	33	1	349	66586	0	9	0	2	airbnb
9	173	1	349	66586	52757	7	0	142	airbnb
10	33	1	349	66586	0	9	0	2	airbnb
11	35	1	349	66586	0	9	0	4	airbnb
12	33	2	349	66586	52791	7	0	2	airbnb
13	34	1	349	66586	0	9	0	3	airbnb
14	38	0	349	66586	4505	5	0	7	airbnb
15	33	1	349	66586	54160	7	0	2	airbnb
16	33	2	349	66586	56804	7	0	2	airbnb
17	34	1	349	66586	0	9	0	3	airbnb
18	35	1	349	66586	0	9	0	4	airbnb
19	39	1	349	66586	56865	7	0	8	airbnb
20	33	1	349	66586	0	9	0	2	airbnb
21	38	0	349	66586	10601	5	0	7	airbnb
22	46	0	349	66586	2505	5	0	15	airbnb
23	33	1	349	66586	51090	7	0	2	airbnb
24	56	1	349	66586	0	9	0	25	airbnb
25	33	1	349	66586	62872	7	0	2	airbnb
26	38	0	349	66586	1321	5	0	7	airbnb
27	38	0	349	66586	297	5	0	7	airbnb
28	38	0	349	66586	7465	5	0	7	airbnb

Figure 5. Master CSV file for single packet type

Data Preprocessing

The following are the preprocessing operations we applied to the data in the csv file:

- Merge all applications' files of a particular packet in a single master file.
- Drop all empty columns.
- Drop columns that have the same value for all rows (these won't help the model differentiate between the applications).
- Drop rows with empty values, some packets had missing data, so we decided to drop them instead of filling the missing data with dummy data using mean for example. The reason we did this is that we have a lot of data, so we didn't want the dummy data to affect the model's performance.
- Drop duplicated columns.

- Add a label column to each master file, where each label is the category that the packet belongs to.
- Apply one-hot encoding to the non-numerical data.
- Normalize the data to have mean 0 and standard deviation of 1.

Attempts and Results Analysis

For the results analysis, we were not entirely sure about what packets were going to produce the most sufficient results. For this, we performed feature engineering and data analysis to select the packets to focus our attention to. Below in Table 3 lists the packets we selected to train and test our model predictions on. We also included the number rows as an indication of how much data was available. Keep in mind that the row count was the overall combination of all packet data across the twenty applications once our preprocessing was completed. The data was easy to distinguish since we made sure we were labeling each record with its respective application.

Packet	Data Count
LTE_MAC_UL_Tx_Statistics	3788
LTE_RRC_OTA_Packet	603
LTE_PDCP_DL_Stats	1186
LTE_PDCP_UL_Stats	5573
LTE_RLC_UL_Stats	343

Table 3. Data description

Table 4 below lists the machine learning models we used for testing each packet. We considered a variance of models that tended to perform well with multi-label classification and ones that could perform better with lower amounts of data.

Model
Logistic Regression
Support Vector Machine
XGBoost classifier
Fully connected neural network

Table 4. Machine Learning models used

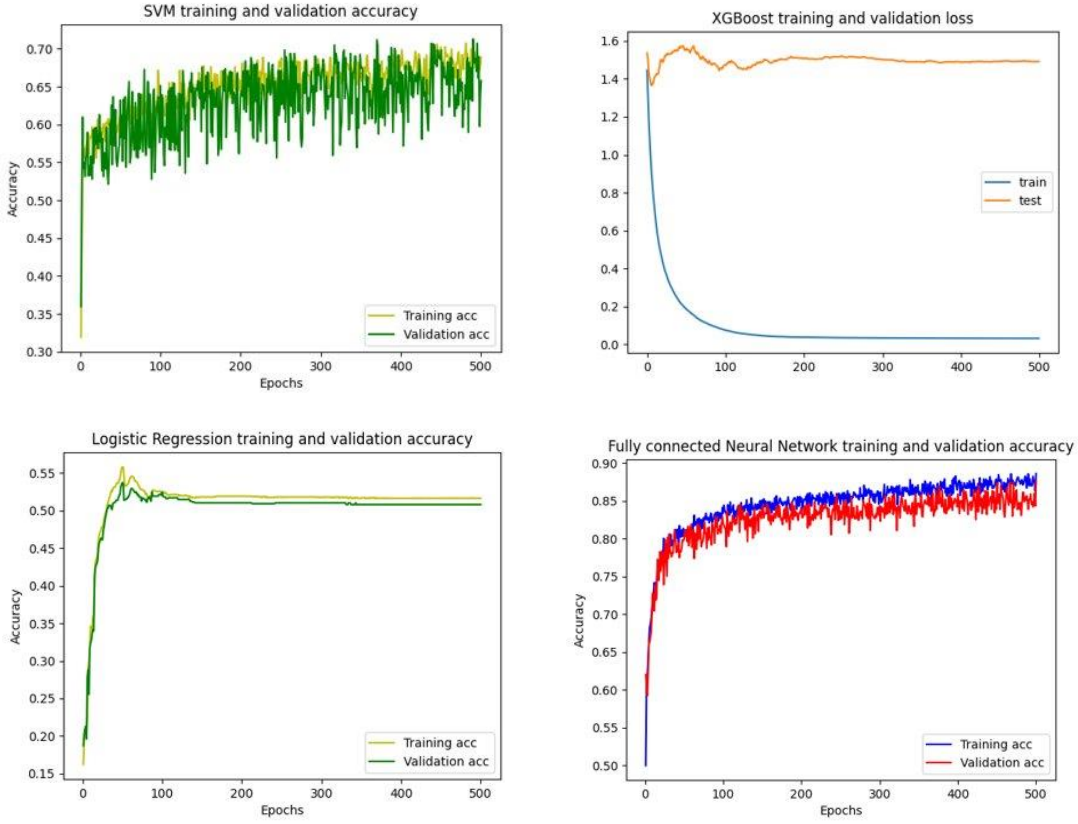
Selected Packet and Model

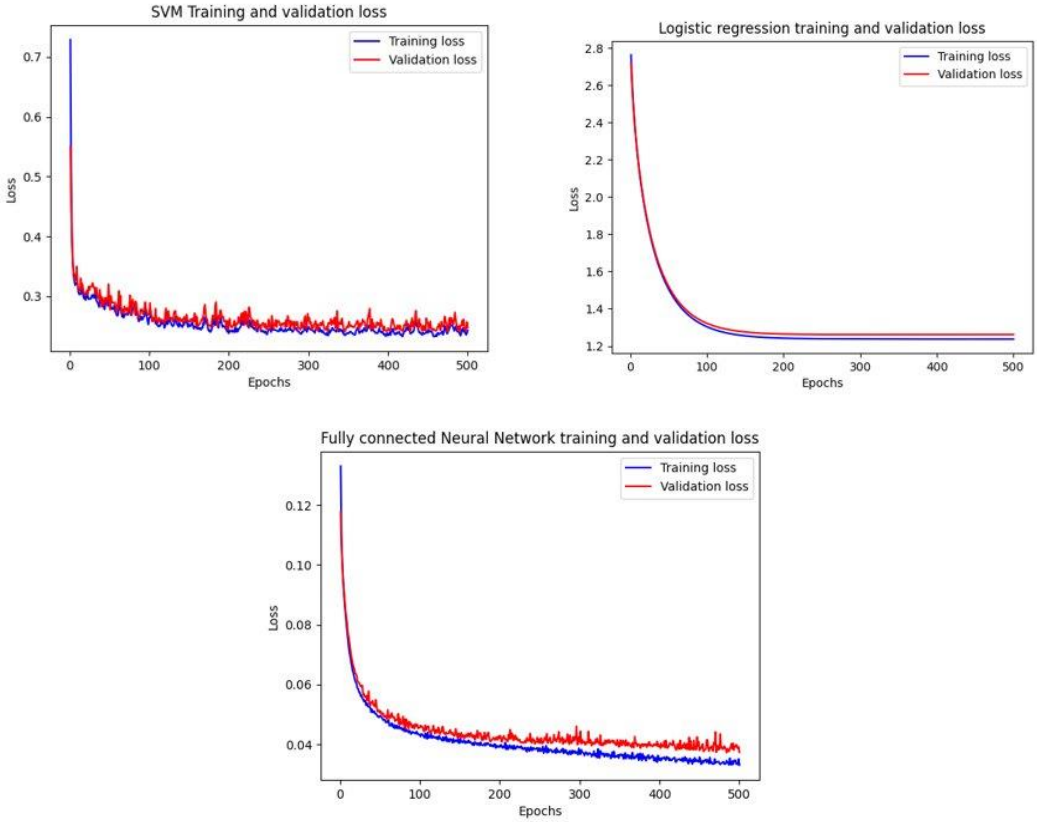
After training using the packets mentioned above, we chose to stick with the LTE_PDCP_UL_Stats packet as it outperformed all other packets on all models. This packet carried data about the number of bytes for Data and Control plane PDCP PDU, number of SDU missing from upper layer, and included PDCP Header length as well. We believe that these features are what helped the model distinguish between the applications, the below table shows the results obtained on each model using this packet:

Packet	Logistic regression	SVM	Full connected network	
LTE_RRC_OTA_Packet	0.336	0.336	0.277	0.235
LTE_MAC_UL_Tx_Statistic	0.165	0.145	0.165	0.176
LTE_PDCP_DL_Stats	0.435	0.300	0.502	0.317
LTE_PDCP_UL_Stats	0.627	0.630	0.882	0.598
LTE_RLC_UL_Stats	0.406	0.264	0.348	0.232

Table 5. Results obtained on selected Machine Learning models

The below shows the losses and accuracies for the training and validation data over epochs:





Figures 6-12. Loss and accuracy results for training and testing data

From the above, we can see that the Fully connected neural network model outperforms all other models in accuracies and losses with around an 88.2% test accuracy. We also observed that the models were not overfitting the data since the training and testing curves were close to each other (results were almost similar even on unseen data) So the model learned useful features and is not memorizing input data. However, the XGBoost model was overfitting the data as we can see in the plot.

Lessons Learned

We learned that the amount of data was not as much as we were expecting when running the recording for just 5 minutes. We believe that based on the accuracy results, we saw higher variance between the different seeds we tested on. The accuracy we recorded was the best we saw across all the seeds (shuffled data) but varied depending on how the data was shuffled. With a higher amount of data, we would have ended up getting less variance and possible better results since the model has more data to make representations on instead of generalizing, which could lead to poorer model performance. In hindsight, we learned that data is important for a classification problem like this one. We further expand upon this by thinking that we could have applied a K-fold cross validation to break our data into K partitions rather than dedicating a full 20% of our dataset towards validation.

Another lesson that we learned is that PDCP plays an important role in distinguishing between different user activities. We noticed that applications between social media and shopping varied greatly when it came to the PDU bytes. It also varied when it came to number of radio bearers and PDCP header length. The PDU bytes was explainable as it varied depending on higher layer signaling and user traffic for the user activities. Since it deals with transferring user plane and control plane data, it was receiving SDU byte information from upper layers such as the application layer. Ultimately, the features were enough to distinguish between user activities and achieve high accuracy.

Conclusions and Future Work

In conclusion, we were able to utilize the MobileInsight data to train a machine learning model to classify the 20 selected applications with high accuracy. So, we can confidently say that MobileInsight has the capability of classifying common applications with its device-side tool for collecting cellular network traces. We were able to successfully collect and preprocess the necessary network packet traces for training the model and getting a decent accuracy.

For our future work, we are considering trying to train a model that classifies more than 20 applications, can we receive around the same performance by expanding the application set? Also, what if we were recording on other carriers (Sprint, AT&T, etc.) beyond the one we recorded on (Verizon)?
Also, one thing that this model might be useful for is utilizing this model to prompt the user based on his preferences, for example notify me whenever I'm spending too much time on Facebook.

References

[1] MobileInsight, getting started with mobileinsight. Retrieved 8th of December, 2021. From http://www.mobileinsight.net/get_started.html

[2] Statista, global mobile data traffic growth rate. Retrieved 9th of December, 2021 from https://www.statista.com/statistics/271405/global-mobile-data-traffic-forecast/#:~:text=By%202022%2C%20mobile%20data%20traffic,growth%20rate%20of%2046%20percent.

[3] B. Yamansavascilar, M. A. Guvensan, A. G. Yavuz and M. E. Karsligil, "Application identification via network traffic classification," 2017 International Conference on Computing, Networking and Communications (ICNC), 2017, pp. 843-848.

[4] MobileInsight, MobileInsight-core framework. Retrieved 8th of December, 2021 form https://github.com/mobile-insight/mobileinsight-core