The goal of the project was to determine, given raw packet traces captured while some user is performing activities on Twitter, 1) what activities the user has performed, and 2) the start and end times of those activities, where the predicted end times are within a minute of the actual end times of activities. The activities were broadly classified into three categories: *Tweet + Image*, *Tweet Text Only* and *Other*. We developed custom algorithms to convert the raw packet captures into meaningful data, used a deep-learning-based autoencoder to de-noise the structured data and finally used the Naïve Bayes algorithm, with class conditional densities for continuous variables being modeled by Kernel Density Estimates, to perform the classification.

We tested the model on two independent test datasets (totally different from the training dataset, which was SET3/TC1). On the first test dataset (SET3/TC2), we could detect *Tweet + Image* with 95.53% recall and 99.99% precision, *Tweet Text Only* with 91.69% recall and 35% precision and *Other* with 51.35% recall and 94.27% precision. On the second test dataset (SET2/DevQA), we got 89.68% recall and 100% precision for *Tweet + Image*, and 78.93% recall and 90.27% precision for *Tweet Text Only*.

There are three types of activities involved in productionizing the current model: 1) data preprocessing/massaging, 2) applying the model on production data to generate labels and 3) model building. Step 2 may be run not everyday but perhaps once a week or so, if change in distribution of data is anticipated. All the scripts referred to in this document are available in the GitHub folder: https://github.com/bibudhlahiri/cleartrail_analytics

1. Data preprocessing/massaging:

We introduced the concept of *sessions* and *flows* to add semantics on the raw packet data, which became helpful in the later phase of modeling. The sessions and flows are created from the raw packet data by the function *create_session_data()* in the script *create_flows.R*. The current version shows how it has been done for the dataset SET3/TC1/23_Feb_2016_Packets_DataSet_TC1.csv. A *session* is a set of packets (in a given dataset of raw packets) that has flown between a combination of server IP, server port, client IP and client port. In essence, it is equivalent to a TCP socket. The raw packet data gives the *SourcePort*, *DestPort*, *SourceIP* and *DestIP*; we have a set of functions to infer from these which one, between the source and the destination, is the server and which one is the client. This inference functions are *get_server_IP()*, *get_server_port()*, *get_client_IP()* and *get_client_port()*. The logic used is that one party between the source and the destination always uses port 443, and it is the server – the port of the client does vary. We create additional columns *ServerIP*, *ClientIP*, *ServerPort* and *ClientPort* to store the result of this inference. We also create the column named *Direction* which takes the value upstream (from client to server) or downstream (from server to client), which is inferred by the function *get_direction()*. These methods have been called between lines 14 and 18 in *create_session_data()*.

We defined a *flow* as a set of contiguous packets that move in a session, all in the same direction. So, for example, in a session (number 1), the first three packets (numbered 1, 2, 3) all went upstream, the next two packets (numbered 4, 5) went downstream, and then next three packets (numbered 6, 7, 8) again upstream, then, packets 1, 2 and 3 will belong to flow number 1 (in session 1), packets 4 and 5 will belong to flow number 2 (in session 1) and packets 6, 7 and 8 will belong to flow number 3. This has been implemented in the method *create_flow_data()* in *create_flows.R*, and this method is invoked from *create_session_data().*

Once the sessions and flows are constructed, they are stored in files, with the file name indicating that the packet dataset is revised – in the sense that it contains the additional derived columns like *ServerIP*, *ClientIP*, *ServerPort*, *ClientPort, Direction*, *session_id* and *flow_id* for each packet.

Once the sessions and flows are in place, the next step in preprocessing is to create a set of derived variables, which are aggregate characteristics of individual timestamps, and which are captured in a data table called *ts_specific_features*. The related methods are in the script *generate_timestamp_features.R,* starting from method *create_timestamp_specific_features()*. The current code shows how it has been done for SET3/TC2/RevisedPacketData_23_Feb_2016_TC2.csv. The aggregate characteristics are the following:

a) *n_packets* : number of packets flowing at a given timestamp
b) *n_sessions*: number of active sessions at the timestamp
c) *n_flows* : number of active flows at that timestamp. Note that this number is the aggregate across all the sessions active at that given time.
d) *n_downstream_packets* and *n_upstream_packets*: number of downstream and upstream packets flowing at the timestamp. Obtained by calling the functions *get_n_downstream_packets()* and *get_n_upstream_packets()*, respectively.
e) *majority_domain*: Of all the domains of the packets flowing at the timestamp, what was the majority? Obtained by calling the function *get_majority_domain()*.
f) *upstream_bytes*: Total number of upstream bytes flowing at the timestamp. Obtained by calling the function *get_upstream_bytes()*.
g) *downstream_bytes*: Total number of downstream bytes flowing at the timestamp. Obtained by calling the function *get_downstream_bytes()*.
h) *frac_upstream_packets*: What fraction of the packets flowing at that time were upstream? Obtained by dividing *n_upstream_packets* by *n_packets*.
i) *frac_downstream_packets*: What fraction of the packets flowing at that time were downstream? Obtained by dividing *n_downstream_packets* by *n_packets*.

j) *avg_packets_per_session*: What was the average number of packets per session, considering the sessions active at that time? Obtained by dividing *n_packets* by *n_sessions*.
k) *avg_packets_per_flow*: What was the average number of packets per flow, considering the flows active at that time? Obtained by dividing *n_packets* by *n_flows*.
l) *total_bytes*: What was the total number of bytes flowing at that time? Obtained by adding *upstream_bytes* and *downstream_bytes.*
m) *frac_upstream_bytes*: What fraction of bytes were upstream at that time? Obtained by dividing *upstream_bytes* by *total_bytes*.
n) *frac_downstream_bytes*: What fraction of bytes were upstream at that time? Obtained by dividing *downstream_bytes* by *total_bytes*.
o) *avg_bytes_per_packet*: Average number of bytes per packet, obtained by dividing *total_bytes* by *n_packets*.
p) *avg_upstream_bytes_per_packet*: Average number of upstream bytes per packet, obtained by dividing *upstream_bytes* by *n_packets*.
q) *avg_downstream_bytes_per_packet*: Average number of downstream bytes per packet, obtained by dividing *downstream_bytes* by *n_packets*.

In method *prepare_data_for_detecting_event_types()* in *detect_event_types.R*, we also accomplish the task of assigning an event to each timestamp, by calling the method *lookup_event()*, which looks up the (event, start time, end time) dataset to infer what event was going on at a given timestamp, i.e., which one among the given events surround the timestamp within its start time and end time. However, this will only be needed if the dataset is prepared for training purposes, as the (event, start time, end time) data will not be available in production as that is what we are inferring.

2. Generating Labels and Model Building:

We provide with the script *wrapper.R* as a user interface. As per the agreement between Impetus and ClearTrail, ClearTrail users will run this script in R with changes in configuration parameters determined by their environment. Please make sure to update the configuration parameters in *config.R*. The definition of each parameter is provided in the *config.R* file itself within the comments above each parameter. All the configurable parameters are file paths, and the version checked in in GitHub assumes the scripts are run on a Mac. For Windows, change the file path from the format "~/cleartrail_osn/SET3/TC2/RevisedPacketData_23_Feb_2016_TC2.csv" to the format "C:\\cleartrail_osn\\SET3\\TC2\\RevisedPacketData_23_Feb_2016_TC2.csv" – making sure to use double backslash (\\).

When *wrapper.R* is compiled and the function *run_everything()* is run, it provides the user with three options:

Option 1: *label_generation* is the activity the user will do on most days: massage the raw packet captures using the procedure explained in Section 1, and send it through the saved model, and get the predicted start and end times of events. It calls the *generate_labels_only()* method in *detect_event_types.R*, followed by the call to *temporal_aggregation()* in *create_final_viz.R*.

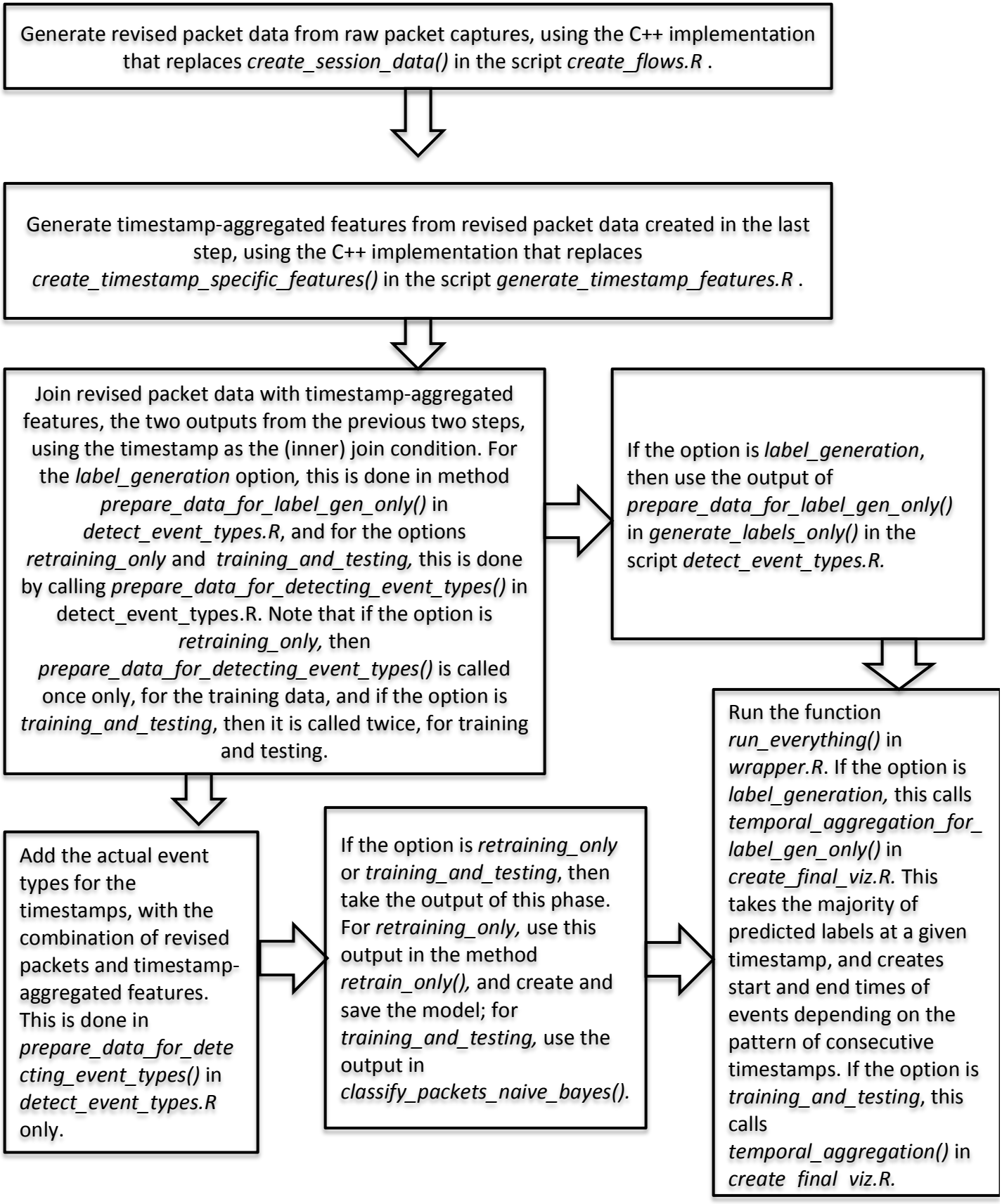
Option 2: *retraining_only* is applicable when the user wants to re-train the model but she does not have a test data to test it on. It just updates the saved model file. Does not return precision/recall. It calls the *retrain_only()* method in *detect_event_types.R,* which re-creates the model. Please make sure that H2O (http://www.h2o.ai/download/) is installed on the machine where this is run, as well as the R library *h2o*, as the deep-learning-based autoencoder, which is used for de-noising the training data, is implemented in h2o.

Option 3: *training_and_testing* is the full package where the user re-trains the model. It assumes the user has a training dataset with start and end times of events, as well as a test dataset with start and end times of events. It gives the user the precision/recall of the individual classes on the test data. This is an expensive operation in the sense that the events and their start and end times need to be manually entered for both training and test data. It also creates a timeline visualization that shows the actual vs the predicted events in a horizontal scale, and the name of the file is provided by the *image_file* parameter in *config.R.*

If the program is run in *retraining_only* or *training_and_testing* modes, it will need labeled events and their start and end times in a CSV file, whose locations are given by events_trg and events_test in config.R. The following **checks** are **to be performed**:

a) All times are in HH24:MI:SS format.
b) There are three columns exactly, namely “Event”, “StartTime” and “EndTime”. There should be no space in the names of these columns.
c) The events are of the following types: “User Login”, “Tweet+Image”, “Tweet Only”, “ReTweet”, “Reply Tweet Text Only”, “Reply Tweet Text and Image”, “User mouse wheel down”, “User mouse drag end” and “Like”. Note that, however, the program will perform the following mappings with the event types:
    i. “Reply Tweet Text and Image” will be mapped to “Tweet+Image”.
    ii. “User Login”, “User mouse drag end” and “User mouse wheel down” will be mapped to “Other”.
    iii. “Reply Tweet Text Only”, “ReTweet”, “Tweet_Only”, “Tweet”, “Reply_Tweet” will be mapped to “Tweet Text Only”.

Hence, the final set of predicted events will have three event types: “Tweet Text Only”, “Tweet+Image” and “Other”.
For a comprehensive understanding of which activities are related to which parts of the code, please refer to the block diagram in the next page.


Generate revised packet data from raw packet captures, using the C++ implementation that replaces *create_session_data()* in the script *create_flows.R* .
Generate timestamp-aggregated features from revised packet data created in the last step, using the C++ implementation that replaces *create_timestamp_specific_features()* in the script *generate_timestamp_features.R* .
Join revised packet data with timestamp-aggregated features, the two outputs from the previous two steps, using the timestamp as the (inner) join condition. For the *label_generation* option, this is done in method *prepare_data_for_label_gen_only()* in *detect_event_types.R*, and for the options *retraining_only* and *training_and_testing,* this is done by calling *prepare_data_for_detecting_event_types()* in detect_event_types.R. Note that if the option is *retraining_only,* then *prepare_data_for_detecting_event_types()* is called once only, for the training data, and if the option is *training_and_testing*, then it is called twice, for training and testing.
If the option is *label_generation*, then use the output of *prepare_data_for_label_gen_only()* in *generate_labels_only()* in the script *detect_event_types.R.*
Add the actual event types for the timestamps, with the combination of revised packets and timestamp-aggregated features. This is done in *prepare_data_for_detecting_event_types()* in *detect_event_types.R* only.
If the option is *retraining_only* or *training_and_testing*, then take the output of this phase. For *retraining_only,* use this output in the method *retrain_only(),* and create and save the model; for *training_and_testing,* use the output in *classify_packets_naive_bayes().*
Run the function *run_everything()* in *wrapper.R*. If the option is *label_generation,* this calls *temporal_aggregation_for_label_gen_only()* in *create_final_viz.R.* This takes the majority of predicted labels at a given timestamp, and creates start and end times of events depending on the pattern of consecutive timestamps. If the option is *training_and_testing*, this calls *temporal_aggregation()* in *create final viz.R.*

For questions about this process, please contact Bibudh Lahiri at blahiri@impetus.com