ATUAV Experimenter Platform

Technical Documentation 2019

Cristina Conati, Grigorii Guz, Sébastien Lallé, Willis Peng, Dereck Toker

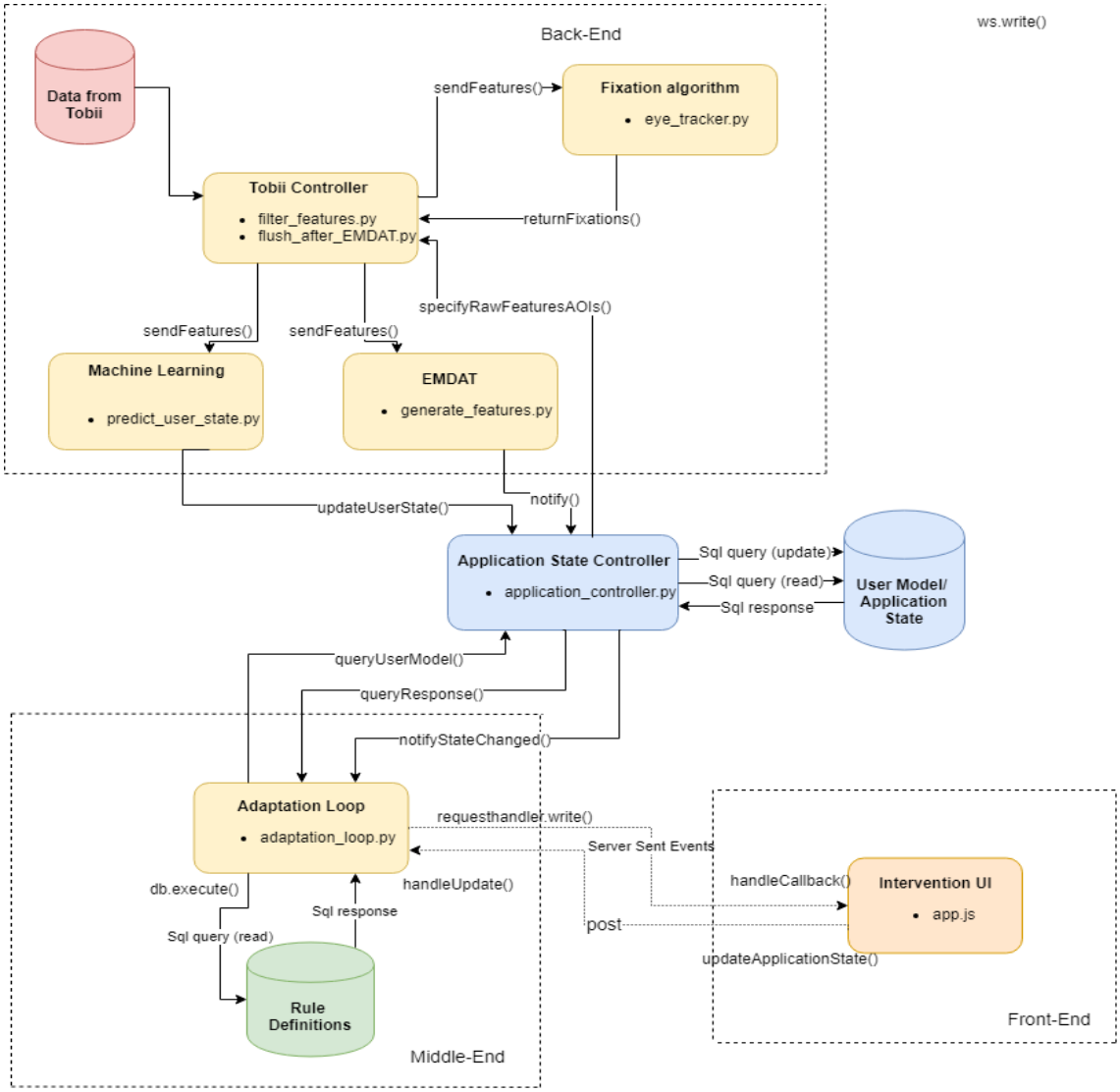
Department of Computer Science, University of British Columbia, Vancouver, BC, Canada

https://github.com/ATUAV/ATUAV_Experimenter_Platform/

CONTENTS

1 SHORT OVERVIEW OF THE SYSTEM'S COMPONENTS AND OVERALL WORKFLOW

The general structure of the platform is depicted in the following diagram:



- The *back-end* is responsible for fetching the data from Tobii eye tracker and analyzing it
  - Tobii controller stores the data and sends the notifications to the databases
  - Fixation algorithm detects fixations
  - EMDAT computes statistical eye tracking features from the raw Tobii data
  - Machine Learning uses features computed by EMDAT in order to make inferences on a user model
- When the backend notifies and sends a new event to the database (User model and Application State), the *middle-end* fetches this event, as well as all previous ones, to analyze the user-defined rules.

    - If any of these rules are satisfied, the middle-end sends a notification to *front-end* to display an intervention and to the *database* for it to keep track of the current interventions.
- The *database* consists of several tables
    - *Rule definitions* table stores the rules specified by the experimenter
    - *User model* keeps the events relevant across the whole execution of the platform (across multiple tasks)
    - *Application state* keeps track of the events for the current task

2 APPLICATION STATE CONTROLLER

2.1 Responsibilities

Inputs:

- user_model_state.db
    - Database specifying events to track, aoi mappings and feature mappings

Responsibilities:

- Maintain the state of the application in the form of a database
- Keep a log of every time user event is triggered

2.2 User Model State Database

/user_model_state.db

An experimenter must create and fill the following tables in user_model_state.db and supply them to the application before running the platform. These tables define the user events that the platform will track and its associated tasks as well as whether each event is active for each task. These tables will be used by the platform to generate dynamic tables used to track the user event logs. It is required to add at least one *task* in the *user_state_task* table.

user_state(event_name, type, aoi, feature)

- **event_name** :
  String
  name of the event to be tracked
- **type**:
  {“fix”, “emdat”, “ml”}
  specifies the type of data the event is tracking
- **aoi**:
  {NULL,String (must reference a name defined in the aoi table)}
  specifies the aoi to assign to this event, or NULL if tracking the entire screen
- **feature:**
  {NULL, String (must be a valid Emdat feature)}
  must reference a valid emdat feature if type is “emdat” otherwise must be NULL

user_state_task(event_name, task)

- **event_name** :
  - String (must reference an event_name defined in user_state table)
  - specifies the associated event
- **task**:
  - Integer,
  - specifies a task in which this event is active (to be tracked)

aoi(name, task, dynamic, polygon)

- **name** :
  - String
  - name of the aoi
- **task**:
  - Integer
  - specifies the task associated
- **dynamic[not implemented]** :
  - Boolean
  - 1 if the aoi is dynamic, 0 otherwise
- **polygon**:
  - List of duples
  - polygon indicating the outline of the aoi in pixel coordinates for this aoi for the current task

Example database:

user_state:

event_name	type	aoi	feature
text_fix	fix	text_aoi	
pupil_event	emdat	vis_aoi	mean_pupil_size
reading_proficiency	ml		

user_state_task:

event_name	task
text_fix	1
text_fix	2
text_fix	3
reading_proffiiciency	1

reading_profficiency	2
pupil_event	3

aoi:

name	task	dynamic	polygon
text_aoi	1	NULL	[(0,0),(0,200),(200,200),(200,0)]
text_aoi	2	NULL	[(0,0),(0,200),(200,200),(200,0)]
text_aoi	3	1	[(100,100),(100,300),(300,300),(300,1000)]
vis_aoi	3	1	[(0,500),(250,500),(250,750),(0,750)]

Example for adding a new User Event using an existing AOI:

```
INSERT INTO user_event VALUES("vis_fix", "fix", "vis_aoi", NULL);
  //define a new event named "vis_fix"  which tracks fixations in the "vis_aoi" AOI

INSERT INTO user_event_task VALUES("vis_fix", 1);
INSERT INTO user_event_task VALUES("vis_fix", 2);
  //set event "vis_fix" to be active for tasks 1 and 2
```

Example for adding a new aoi:

```
INSERT INTO aoi VALUES("top_left_corner_aoi", 1, NULL, "[(0,0),(0,200),(200,200),(200,0)]");
INSERT INTO aoi VALUES("top_left_corner_aoi", 2, NULL, "[(0,0),(0,200),(200,200),(200,0)]");
        INSERT INTO aoi VALUES("top_left_corner_aoi", 3, NULL,
  "[(0,0),(0,200),(200,200),(200,0)]");
  //define an aoi named "top_left_corner_aoi" for tasks 1, 2 and 3
  //note: we also need to define an event in user_state to track this aoi
```

2.3 In Memory Database

During runtime, this is the main database that will track and maintain the user model as well as the application state. In addition to the 3 tables (user_state, user_state_task, aoi) present in user_model_state.db, several tables are created dynamically.

intervention_state(intervention, active, time_stamp, occurences)

- **intervention**:
  - String
  - name of the associated intervention

- **active**:
  - Boolean
  - 1 if the intervention is currently active, 0 otherwise
- **time_stamp**:
  - long
  - time in millisecond of the last delivery of the intervention
- **occurences**:
  - Integer
  - number of times this intervention has been delivered

rule_state(rule, time_stamp, occurences)

- **rule**:
  - String
  - name of the associated rule
- **time_stamp**:
  - long
  - time in milliseconds of the last delivery of the intervention
- **occurences**:
  - Integer
  - number of times this intervention has been delivered

2.3.1 Dynamically generated tables

Tables are generated dynamically at runtime on the in memory database. One table is generated per active user event which has a table name corresponding to its event_name entry in its definition in the **user_state** table. The schema of the tables generated depends on the type of user event, where “fix” maps to a ***fixation_table***, “emdat” maps to an ***emdat_table***, and “ml” maps to an ***ml_table***.

fixation_table(id, time_start, time_end, length)

- **id:**
  - Integer
  - id associated with an entry
- **time_start**:
  - long
  - time in milliseconds of the start of the fixation
- **time_end**:
  - Integer
  - Time in milliseconds of the end of the fixation
- **duration**:
  - long
  - Time in milliseconds of the duration of the fixation

emdat_table(id, interval_value, task_value, runtime_value)

- **id**:
  - Integer
  - Id associated with an entry

- **interval_value**:
  - Float
  - Value associated with the last interval of t-seconds
- **task_value**:
  - Float
  - Value associated with the current task
- **runtime_value**:
  - Float
  - Value associated with the overall runtime of the application

ml_table(id, time_stamp, raw_prediction, value)

- **id**:
  - Integer
  - Id associated with an entry
- **time_stamp**:
  - long
  - Time in milliseconds of the data entry
- **raw_prediction**:
  - Float
  - value between 0 and 1 of the raw feature
- **value**:
  - String
  - The extrapolated value for the prediction

2.4 Application State Controller

/application/application_state_controller.py

A singleton class in charge of facilitating the interfacing between the In Memory Database and the rest of the application. This class is responsible for creating the instance of the In Memory Database upon initialization and deleting it upon termination. In addition, this class creates, maintains and deletes the dynamically generated tables during runtime and also provides querying functionality.

2.4.1 Methods

ApplicationStateController(task)

Creates an instance of the Application State Controller Class, and initializes it to the given *task*, creating the dynamic tables needed to be tracked for that task. Default *task* = 1.

log_task()

Dumps a log of the current state of the In Memory Database in the form of an sql executable script to /log/log_for_task_*.sql where * is the current task.

change_task(*task* **)**

Changes the current active task to the value of given *task*: logs the current state of the In Memory Database, deletes all the dynamically generated tables from the previous task and generates new dynamic tables for the current task.

reset_application()

Prepares the In Memory Database for termination: logs the current state of the database and then deletes all the current dynamically generated tables

get_fix_aoi_mapping()

Returns a dictionary of all active event_names of type “fix” as keys and their corresponding aoi polygon boundaries as values.

get_emdat_aoi_mapping()

Returns a dictionary of all active event_names of type “edmat” as keys and their corresponding aoi polygon boundaries as values

get_emdat_features()

Returns a dictionary of all active event_names of type “emdat” as keys and the emdat feature they are tracking

evaluate_conditional(*query* **)**

Returns the boolean value from evaluating the given *query*. The query must be formed to produce a single column named “result” and a single row with a value of 1 if the conditional was satisfied otherwise 0.

update_fixation_table(*table, id, time_start, time_end, duration* **)**

Inserts a new row into the given *table*, with given values for *id, time_start, time_end* and *duration*.

update_emdat_table(*id, emdat_features* **)**

Inserts rows into the dynamically created tables of type “emdat” given *emdat_features*, a dictionary mapping event_name as keys to a triple with the form of (interval_value, task_value, runtime_value).

update_ml_table(*table, id, time_stamp, raw_prediction, value* **)**

Inserts a new row into the given *table*, with given values for *id, time_stamp, raw_prediction* and *value*.

get_intervention_occurences(*intervention_name* **)**

Returns the number of times the intervention with the given *intervention_name* has been delivered.

get_rule_occurences(*rule_name* **)**

Returns the number of times the rule with the given *rule_name* has delivered an intervention.

set_intervention_active(*intervention_name, rule_name, time_stamp* **)**

Sets the state of the application to reflect that the intervention with the given *intervention_name* is currently active. Also increments the number of occurences for the intervention with the given intervention_name and the rule with the given rule_name at the given time_stamp.

set_intervention_inactive(*intervention_name* **)**

Sets the state of the application to reflect that the intervention with the given *intervention_name* is no longer active

is_intervention_active(*intervention_name* **)**

Returns true if the intervention with the given *intervention_name* is currently active, otherwise returns false.

2.5 Logs

/log/log_for_user_{i}_task_{j}.sql

Experimenter platform creates snapshots of the In Memory Database for debugging or researching purposes between task switches and upon application termination. The state of the database is saved as an executable sql script under *./log/log_for_user_{i}_task_{j}.sql* where *{i}* is the current user id (default 9999) and *{j}* is the relevant task number. If the application crashes or prematurely terminates, the latest snapshot of the database state may not be logged.

3 BACK-END

3.1 Responsibilities

The backend is responsible for

- Receiving raw gaze data from Tobii eye tracker and sending (Tobii Controller)
- Detecting the user's fixations and notifying the database (Fixation Algorithm)
- Analyzing the raw gaze data and calculating the statistical information (EMDAT)
- Predicting the user's cognitive abilities, or solving any other classification problem defined by the user (Machine Learning)

3.2 Tobii Eyetracker

/application/backend/eye_tracker.py

The singleton class used to communicate with Tobii eye tracker API: it initializes the eye tracker, stores the raw gaze data, so the detection components can compute the features they are responsible for, and it stores the EMDAT features computed over the whole execution of the platform.

wait_for_eye_tracker()

Blocks the current thread until Tobii API detects an eye tracker in the system.

on_eye_tracker_browser_event(event_type, event_name, eyetracker_info **)**

Adds a new or updates an existing tracker to self.eyetrackers, if one is available.

- event_type
  - a tobii.eye_tracking_io.browsing.EyetrackerBrowser event
- event_name
  - Passed by some underlying Tobii function specifying a device name, not used here
- eyetracker_info

- a struct containing information on the eye tracker (e.g. it's product_id)

destroy()

Removes the reference to eye tracker and stops all operations (data generating thread, etc.).

activate(eyetracker **)**

Connects to specified eye tracker, initializes the data generating thread.

- eyetracker
  - key for the self.eyetracker dict under which the eye tracker to which you want to connect is found

on_eyetracker_created(error, eyetracker, eyetracker_info **)**

Function is called by TobiiController.activate, to handle all operations after connection to a tracker has been successful. Notifies if errors during connection occurred.

- error
  - some Tobii error message
- eyetracker
  - key for the self.eyetracker dict under which the eye tracker that is currently connected
- eyetracker_info
  - name of the eye tracker to which a connection has been established

startTracking( )

Starts the collection of gaze data, raw data arrays start being filled.

stopTracking( )

Stops the collection of gaze data, empties the raw data arrays.

on_gazedata(error, gaze **)**

Adds new data point to the raw data arrays. If x, y coordinate data is not available, stores the coordinates for this datapoint as (-1280, -1024). Any other feature, if not available, is stored as -1.

- error
  - Tobii error message
- gaze
  - Tobii gaze data struct

add_fixation(x, y, duration **)**

Called by FixationDetector when a new fixation is detected. Adds a new fixation to data array to be used for EMDAT features calculation.

- x
  - coordinate of fixation on the screen
- y
  - coordinate of fixation on the screen
- duration
  - duration of fixation in microseconds

get_pupil_size(pupilleft, pupilright **)**

Used for extracting pupilsize in on_gazedata(). If recordings for both eyes are available, return their average, else return value for a recorded eye (if any)

- pupilleft
  - recording of pupil size on left eye
- pupilright
  - recording of pupil size on right eye

get_pupil_velocity(last_pupilleft, last_pupilright, pupilleft, pupilright, time **)**

Used for extracting pupilvelocity in on_gazedata(). If pupilsizes for both eyes are available, return the average of their difference, else return value for a recorded eye (if any)

- last_pupilleft
  - Pupilsize for left eye from previous gaze object
- Last_pupilright
  - Pupilsize for right eye from previous gaze object
- Pupilleft
  - Pupilsize for left eye from current gaze object
- Pupilright
  - Pupilsize for left eye from current gaze object
- Time
  - Timestamp difference between current and last gaze object

get_distance(distanceleft, distanceright **)**

Used for extracting head distance in on_gazedata(). If recordings for both eyes are available, return their average, else return value for a recorded eye (if any)

- distanceleft
  - recording of distance on left eye
- distanceright
  - recording of distance on right eye

update_aoi_storage(AOIS **)**

Add new aois to global EMDAT feature storage dictionary. Called during a task switch by EMDATComponent.

- AOIS
  - A dictionary of AOIs obtained from Application State

init_emdat_global_features()

Initialize global EMDAT feature storage dictionary. Called by EMDATComponent.

3.3 Detection Component

/application/backend/detection_component.py

Abstract class for defining detection components - classes responsible for calculating features from raw data and sending them to Application State. Abstracts away all the interactions with Tornado framework, providing an intuitive and flexible base for implementing feature extractors.

__init__(tobii_controller, adaptation_loop, is_periodic = False, callback_time = 600000, webSocket = None**)**

- Tobii_controller
  - An instance of TobiiController already connected to an eyetracker
- Adaptation_loop
  - Instance of AdaptationLoop to send the computed features to Application State
- Is_periodic
  - Boolean - True if run() method should be called periodically.
- Callback_time
  - Long - if is_periodic = True, specifies how often should the run() method be called, in microseconds.
- webSocket
  - WebSocket - if needed, the features can be sent over websocket connection.

@abstract
run()

Abstract method for calculating features from raw data. Get called periodically or once, depending on how the DetectionComponent was initialized.

@abstract
notify_app_state_controller()

Abstract method for sending features to Application State. Should be called at the end of execution of run().

start()

Method for scheduling a run() call in Tornado's IOLoop. Depending on how DetectionComponent was initialized, run() gets scheduled to run either periodically or once.

3.4 Fixation Detector

/application/backend/fixation_detector.py

Implementation of DetectionComponent used to detect fixations from raw gaze data stored in TobiiController. Once called, runs indefinitely. To store fixations, 4 fields are used:

- X and Y coordinates, which are the average of the coordinates for all datapoints in the fixation
-

__init__(tobii_controller, adaptation_loop, liveWebSocket**)**

See __init__ in DetectionComponent.

@gen.coroutine
run()

Concurrently detects fixations, defined as consecutive samples with an inter-sample distance of less than a set amount of pixels (disregarding missing data). Uses params.MAXDIST and params.MINDUR for respectively the distance and the smallest possible time length of a fixation. The method is a coroutine, which means that it can pause its execution and give control to other components of the platform.

@gen.coroutine
wait_for_new_data(array_index, array_iterator**)**

Coroutine which yields the control when there are no new datapoints available from Tobii. Called from run()

- Array_index
  - The position of first unused datapoint in raw data arrays so far
- Array_iterator
  - The number of new datapoints needed to run fixation_detection() method

@gen.coroutine
get_data_batch(array_index, array_iterator**)**

Returns array_iterator number of points from data arrays starting at the index array_index. Used by run() method.

fixation_detection(x, y, time, validity**)**

Detects fixations, defined as consecutive samples with an inter-sample distance of less than a set amount of pixels (disregarding missing data). Gets called by run() method on each new batch of array_iterator number of datapoints.

- X
  - Array of x coordinates for array_iterator consecutive datapoints
- Y
  - Array of y coordinates for array_iterator consecutive datapoints
- Validity
  - Array with validity (true/false values) for array_iterator consecutive datapoints

find_new_start(x, y, maxdist, i, si**)**

Helper method for fixation_detection(): when it was detected that fixation is too short, it finds another starting point for the next fixation.

fixation_inside_aoi(x, y, poly**)**

- of

Determines if a point is inside a given polygon or not. The algorithm is called "Ray Casting Method".

- X
  - X - coordinate of a fixation
- Y
  - Y - coordinate of a fixation
- Poly
  - is a list (x,y) pairs defining the polygon

3.5 Emdat Component

/application/backend/emdat_component.py and /application/backend/emdat_utils.py

Implementation of DetectionComponent used to compute statistical eye tracking features. Based on EMDAT: https://github.com/ATUAV/EMDAT. Called periodically, with periodicity specified by the user. The features include:

- Pupil features (AOI and whole screen)
  - Numpupilsizes
  - Numpupilvelocity
  - Meanpupilsize
  - Stddevpupilsize
  - Maxpupilsize
  - Minpupilsize
  - Meanpupilvelocity
  - Stddevpupilvelocity
  - Maxpupilvelocity
  - Minpupilvelocity

- Head distance from the screen features (AOI and whole screen)
    - Numdistancedata
    - Meandistance
    - Stddevdistance
    - Maxdistance
    - Mindistance
- Path and path angle features (whole screen only)
    - Numfixdistances
    - Numabsangles
    - Numrelangles
    - Meanpathdistance
    - Sumpathdistance
    - Stddevpathdistance
    - Eyemovementvelocity
    - Sunmabspathangles
    - Abspathanglesrate
    - Meanabspathangles
    - Stddevabspathangles
    - Sumrelpathangles
    - Relpathanglesrate
    - Meanrelpathangles
    - Stddevrelpathangles
- Fixation features (AOI and whole screen)
    - Numfixations
    - Fixationrate
    - Meanfixationduration
    - Stddevfixationduration
    - Sumfixationduration
    - Fixationrate
- AOI transition features (AOI only)
    - Total_trans_from
    - numtransfrom_AOINAME (for each AOI)
    - proptransfrom_AOINAME (for each AOI)

Description of the features are provided here: https://www.cs.ubc.ca/~skardan/EMDAT/

At each point of time, three sets of features are kept track of. At each call to run() method, we compute the features for the new Tobii data generated since the previous call to run(), and store them in *emdat_interval_features* dictionary. Next, we merge these features with *emdat_task_features* dictionary, which contains the features for all datapoints in this task so far, and with *emdat_global_features* dictionary, which contains the features for all datapoints in all tasks (so data from previous tasks is taken into account).

For most features, value of -1 (or 0 for counting features such as numpupilsizes and numdistances) indicates that this feature was not computed (no valid raw Tobii data available yet).

__init__(tobii_controller, adaptation_loop, liveWebSocket**)**

See __init__() in DetectionComponent

notify_app_state_controller()

Selects features for specified events in Application State and sends them to the database.

run()

Calculates the features for new raw Tobii data collected since the last call to EMDAT Component, merges it with previously computed features and sends the results to the Application State.

init_emdat_features(feature_dictionary**)**

Initializes the given feature dictionary with empty values.

merge_features(part_features, accumulator_features**)**

Merges features from two dictionaries into accumulator_features. Usually called with *emdat_interval_features* as part_features and with *emdat_task_features* or *emdat_global_features* as accumulator_features

- Part_features
  - A dictionary with features used to update values in accumulator_features
- Accumulator_features
  - A dictionary with features used to be updated with part_features

calc_pupil_features()

Called from run(). Calculates pupil features for the whole screen with new raw Tobii datapoints generated since the last call to run(). Features are stored in *emdat_interval_features.*

calc_distance_features()

Called from run(). Calculates distance features for the whole screen with new raw Tobii datapoints generated since the last call to run(). Features are stored in *emdat_interval_features.*

calc_fix_ang_path_features()

Called from run(). Calculates fixation, angle and path features for the whole screen with new raw Tobii datapoints generated since the last call to run(). Features are stored in *emdat_interval_features.*

calc_validity_gaps()

Calculates the validity gaps in new raw Tobii data, i.e. segments with contiguous invalid datapoints, and stores the time segments for which, during platform's execution, the data was invalid.

calc_aoi_features()

Calculates pupil, distance, fixation and transition features for AOIs specified for this task using the helper functions listed below.

generate_aoi_pupil_features(aoi, valid_pupil_data, valid_pupil_velocity**):**

Generates pupil features for given AOI

- Aoi
  - The name of the AOI to compute features for
- Valid_pupil_data
  - Pupil size data for this AOI
- Valid_pupil_velocity
  - Pupil velocity data for this AOI

generate_aoi_distance_features(aoi, valid_distance_data**):**

Generates distance features for given AOI

- Aoi
  - The name of the AOI to compute features for
- Valid_distance_data
  - Distance data for this AOI

generate_aoi_fixation_features(aoi, fixation_data, sum_discarded, num_all_fixations**):**

Generates fixation features for given AOI

- Aoi
  - The name of the AOI to compute features for
- fixation_data
  - Fixation data for this AOI (x, y, duration)
- Sum_discarded
  - Time length of invalid data segments among the new Tobii datapoints produced, result of get_length_invalid()
- Num_all_fixations
  - Number of fixations overall on the new set of datapoints (not only in this AOI)

generate_aoi_transition_features(cur_aoi, fixation_data, fixation_indices**):**

Generates distance features for given AOI

- Aoi
  - The name of the AOI to compute features for
- Valid_distance_data
  - Distance data for this AOI

get_length_invalid():

Takes the result of calc_validity_gaps() to calculate the sum of lengths of invalid segments.

merge_pupil_features(part_features, accumulator_features**)**

Merges pupil features (for whole screen and AOIs) from part_features into accumulator_features

merge_distance_features(part_features, accumulator_features**)**

Merges distance features (for whole screen and AOIs) from part_features into accumulator_features

merge_fixation_features(part_features, accumulator_features, length**):**

Merges fixation features (for whole screen) from part_features into accumulator_features

- Length
  - Sum of time lengths for data used to calculate features in accumulator_features and features in part_features

merge_path_angle_features(part_features, accumulator_features**)**

Merges path and angle features (for whole screen) from part_features into accumulator_features

merge_aoi_fixations(part_features, accumulator_features)

Merges fixation AOI features from part_features into accumulator_features

merge_aoi_transitions(part_features, accumulator_features**)**

Merges transition AOI features from part_features into accumulator_features

calc_distances(fixdata**)**

Returns the Euclidean distances between a sequence of recorded fixations

- fixdata
  - A list of recorded fixations

calc_abs_angles(fixdata**)**

Returns the absolute angles between a sequence of recorded fixations that build a scan path. Absolute angle for each saccade is the angle between that saccade and the horizontal axis.

- fixdata
  - A list of recorded fixations

calc_rel_angles(fixdata**)**

Returns the relative angles between a sequence of "Fixation"s that build a scan path in radians.
Relative angle for each saccade is the angle between that saccade and the previous saccade.
Defined as: angle = acos(v1 dot v2) such that v1 and v2 are normalized vector2coord

- fixdata
  - A list of recorded fixations

minfeat(part_features, accumulator_features, feat, nonevalue**):**

A helper method that calculates the min of a target feature over two feature dictionaries

- Feat
  - a string containing the name of the target feature
- Nonevalue
  - value to be ignored when computing the min (typically -1)

maxfeat(part_features, accumulator_features, feat, nonevalue**):**

A helper method that calculates the max of a target feature over two feature dictionaries

- Feat

    - a string containing the name of the target feature
- Nonevalue
    - value to be ignored when computing the min (typically -1)

weightedmeanfeat(part_features, accumulator_features, totalfeat, ratefeat**):**

A helper method that calculates the weighted average of a target feature over two feature dictionaries

- totalfeat
    - a string containing the name of the feature that has the total value of the target feature
- ratefeat
    - a string containing the name of the feature that has the rate value of the target feature

aggregatestddevfeat(part_features, accumulator_features, totalfeat, sdfeat, meanfeat, aggregate_mean**):**

A helper method that calculates the aggregated standard deviation of a target feature over two feature dictionaries

- totalfeat
    - a string containing the name of the feature that has the total value of the target feature
- sdfeat
    - a string containing the name of the feature that has the rate value of the target feature (name of this particular standard deviation feature)
- Meanfeat
    - a string containing the name of the feature corresponding to the mean value used to calculate a standard deviation feature
- Aggregate_mean
    - Aggregated mean feature for part_features and accumulator_features used to calculate a standard deviation

sumfeat(part_features, accumulator_features, feat**):**

A helper method that calculates the sum of a target feature over a list of objects

3.6 ML Component

/application/backend/ml_component.py

Placeholder provided for users to implement their own classifiers.

In order to use your classifier, you might need to import the library where the classifier is implemented, say,
import sklearn import DecisionTreeClassifier
Or, in case you have a custom classifier, you should

1. Move your code (say, your_classifier.py) to /application/backend
2. Run import your_classifer at the top of ML_Component file.

Then, initialize your classifier as a field of ML_Component and load the model parameters (if needed) in __init__().
In run(), add the code for running the classifier, and store the predicted value in the dictionary self.predicted_features with the name for this feature you used in the database as a key. For example, if the table that should store your feature is called reading_proficiency, run
self.predicted_features['reading_proficiency'] = your_classifier.predict()

Don't forget to set the value of self.threshold to fit your needs. If the predicted value is higher than self.threshold, the column "value" in the database will have the value "high", and "low" otherwise. Of course, the actual raw output of your classifier is also stored in column "raw_prediction".

__init__(tobii_controller, adaptation_loop, liveWebSocket**)**

See __init__() in DetectionComponent. The model, as well as self.threshold, should be specified by the user

run()

See __init__() in DetectionComponent

__init__(tobii_controller, adaptation_loop, liveWebSocket**)**

See __init__() in DetectionComponent

4 MIDDLE-END

4.1 Responsibilities

- Given an user state event, communicate with the Application State Controller to check the currently active interventions to see if the event will trigger any removals
- Given an user state event, check against the active rules for the current task based on the rules defined in Gaze Event Rules database. Communicate with the Application State controller to check if any of the conditionals evaluated to true
- Update the Application State Controller when a rule or intervention has been dispatched
- Dispatch removal and delivery calls to the front-end

4.2 Gaze Event Rules Database

./database/gaze_event_rules.db

The Gaze Event Rules Database is an sqlite database and must be provided by the user prior to running the application. The database contains specifications for the rules and interventions to evaluated and contains the following tables:

intervention (name, max_repeat, function, arguments, delivery_delay, transition_in, transition_out)

- **name**
  - String
  - Name of the intervention
- **max_repeat**
  - {NULL, Integer}
  - Number of maximum repeats allowed, a value of -1 or NULL implies unlimited repeats
- **function**
  - String
  - Name of Javascript function to be called from the front-end, on delivery of this intervention
- **arguments**
  - Dictionary
  - Arguments and values to be provided to called with the function from the front-end
- **delivery_delay[not implemented]**
  - Integer
  - The time delay in milliseconds before delivering the intervention
- **transition_in**
  - Integer
  - The time in milliseconds of the animation duration for delivering the intervention
- **transition_out**
  - Integer
  - The time in milliseconds of the animation duration for removing the intervention

rule (name, delivery_trigger, delivery_sql_condition, intervention, removal_trigger, removal_sql_condition, max_repeat, active_retrigger)

- **name**
  - String
  - Name of the rule
- **delivery_sql_condition**
  - String (must be a valid sqlite query)
  - Sqlite condition to be checked for intervention delivery. Must be formed to return a table with a single column “result” and single row with value 1 if the condition is satisfied otherwise 0.
- **removal_sql_condition**
  - String (must be a valid sqlite query)

    - Sqlite condition to be checked for intervention removal. Must be formed to return a table with a single column “result” and single row with value 1 if the condition is satisfied otherwise 0.
- **max_repeat**
    - Integer
    - Maximal number of times the rule may be delivered.
- **active_retrigger**
    - Boolean
    - If true, rule will redeliver an intervention even if it is already currently active.

rule_delivery_trigger (rule_name, delivery_trigger_event)

- **rule_name**
    - String (must reference the name field of an entry of in the **rule** table)
    - Name of the corresponding rule
- **delivery_trigger_event**
    - String (must reference an event_name from **user_state** table in user_model_application.db)
    - Event_name of event that triggers a check of the delivery condition for this rule

rule_removal_trigger (rule_name, removal_trigger_event)

- **rule_name**
    - String (must reference the name field of an entry in the **rule** table)
    - Name of the corresponding rule
- **removal_trigger_event**
    - String (must reference an event_name from **user_state** table in user_model_application.db)
    - Event_name of event that triggers a check for the removal condition for this rule

rule_intervention_payload (rule_name, intervention_name)

- **rule_name**
    - String (must reference the name field of an entry in the **rule** table)
    - Name of corresponding rule
- **intervention_name**
    - String (must reference an intervention from **intervention** table)
    - Name of an intervention to dispatch if the rule is successfully evaluated

rule_task (rule_name, task)

- **rule_name**
    - String (must reference the name of a rule from the **rule** table)
    - Name of rule this entry references
- **task**
    - Integer
    - Task number in which this rule is active

Example Database

extending from example User Model State database:

intervention:

name	max_repeats	function	arguments	delivery_delay	transition_in	transition_out
intervention_1	1	highlightVisOnly	"type": "reference", "id": 1, "bold": true, "bold_thickness": 3, "desat": true, "color": "green", "arrow": false}’	0	1000	1000
intervention_2	5	highlightVisOnly	"type": "reference", "id": 1, "bold": false, "desat": true, "color": "black", "arrow": true, "arrow_direction": "bottom"}’	0	0	0
intervention_3	NULL	highlightVisOnly	"type": "reference", "id": 1, "bold": true, "bold_thickness": 5, "desat": true, "color": "orange", "arrow": false}’	0	500	500

rule:

name	delivery_sql_condition	removal_sql_condition	max_repeat	active_retrigger
text_highlight	Select 1 as result;	Select 1 as result;	1	0
text_highlight_2	Select 1 as result;	Select 1 as result;	-1	1
rule_3	Select 1 as result;	Select 1 as result;	5	0

rule_delivery_trigger:

rule_name	delivery_trigger_event
text_highlight	text_fix
text_highlight_2	text_fix

rule_3	text_fix
rule_3	pupil_event

rule_removal_trigger:

rule_name	removal_trigger_event
text_highlight	pupil_event
text_highlight	reading_proffiiciency
text_highlight_2	pupil_event
rule_3	text_fix

rule_intervention_payload:

rule_name	intervention_name
text_highlight	intervention_1
text_highlight_2	intervention_2
rule_3	intervention_1
rule_3	intervention_2

rule_task:

rule_name	task
text_highlight	1
text_highlight	2
text_highlight_2	2
rule_3	1
rule_3	3

4.2.1 Adding a new intervention:

INSERT INTO TABLE “intervention” VALUES(“intervention_4”, NULL, “highlightVisOnly”, “”, “500”, “1000”, “1000”)

//create a new intervention called “intervention_4” with unlimited repeats which calls the highlightVisOnly function with a delay of 500ms and a transition in and out of 500ms.

4.2.2 Adding a new rule:

INSERT INTO TABLE “rule” VALUES(“new_rule”, “text_fix”, “Select 1 as result;”, “intervention_4”, “pupil_event”, “Select 1 as a result”, -1, 0);

//create a new rule that always triggers on new “text_fixation” events which dispatches “intervention_4”, is removed by new “pupil_event” events and has unlimited repeats, and does not trigger when already active.

INSERT INTO TABLE “rule_task” VALUES(“new_rule”, 1);
INSERT INTO TABLE “rule_task” VALUES(“new_rule”, 2);

//set the “new_rule” to be active for tasks 1 and 2.

4.3 Adaptation Loop

/application/middleend/adataption_loop.py

The Adaptation Loop Class is a singleton class in charge of evaluating the rules, delivering and removing interventions to and from the front-end.

The evaluate_rules() method is called every time a new user_event is triggered. Every rule which has the delivery_trigger field which matches the triggered user_event performs a check of its delivery_sql_condition. If the delivery condition is evaluated to be true, the associated intervention is added to an array to be passed to the front-end for delivery.

Every rule which has an active intervention and has the removal_trigger which matches the triggered user_event performs a check of its removal_sql_condition. If the removal condition is evaluated to be true, the associated intervention is added to an array to be passed to the front-end for removal.

4.3.1 Methods

AdaptationLoop(*app_state_controller, live_web_socket* **)**

Initializes the class with a given ApplicationStateController *app_state_controller* and an array of websockets as *live_web_sockets* used to communicate with the front-end.

evaluateRules(*event_name, time_stamp* **)**

Evaluates all the rules triggered by an event with *event_name*. The method performs the following steps:

1. All rules which which have an active corresponding intervention with removal_conditions matching event_name performs a conditional check on its removal_sql_condition.
2. Dispatch all rules that have a condition which evaluates to true to the front-end using a websocket. Removal dispatches are passed in the form of a JSON string, having the tag of “remove” with the intervention and its arguments as an array.
3. All rules that have delivery_triggers matching the *event_name* performs a conditional check on its delivery_sql_condition.
4. Dispatch all rules that have a condition which evaluates to true to the front-end using a websocket. Delivery dispatches are passed in the form of a JSON string, having the tag of “delivery” with the interventions and its arguments as an array.
5. If any rules are delivered, the rule and its corresponding intervention updated with the given *time_stamp*.

4.4 Example SQL conditionals:

Listed are some sample SQL conditionals that could be formulated as the conditional checks for a rule definition. These conditionals are extending the database definitions of the same User State Model and the Gaze Event Rules.

At least two fixations in the text:

```
Select
        case when count(*) > 1
                then 1 else 0
        end result
from text_fix;
```

At least two fixations in the text and ‘reading_profficiency’ is ‘low’:

```
Select
        case when count(*) > 1
                then 1 else 0
        End result
from text_fix, reading_profficiency RP
where RP.time_stamp = (select max(time_stamp) from reading_profficiency)
and RP.value = 'low';
```

If a fixation has occurred on the visualization after 1000 milliseconds and two text fixations have occured:

```
Select
   Case when count(distinct text_fix.id) > 2
      Then 1
      Else 0
   End result
From vis_fix, text_fix, intervention_state
where intervetion_state.intervention = 'intervention_2' and vis_fix.time_start > 1000;
```

If 2 consecutive fixations have occured on the text:

```
Select
   Case when count(*) > 0
      Then 1
      Else 0
   End result
From text_fix TF1,tex_fix TF2, intervention_state
Where TF1.id + 1 = TF2.id and intervention_state.intervention = “intervention_1” and
(intervention_state.time_stamp < TF.time_start or intervention_state.occurrences = 0) ;
```

If at least 5 fixations have occured on the text and 1 fixation on vis (vis_intervention) at some later time:

```
Select
   case when count(*) > 0
      then 1
      else 0
   end result
From
        (select * from text_fix, vis_fix, intervention_state
        where text_fix.time_start > vis_fix.time_start
        and intervention_state.intervention = "vis_intervention"
        and (intervention_state.time_stamp < vis_fix.time_start or
        intervention_state.occurrences = 0)
        group by vis_fix.id
        having count(text_fix.id) > 4);
```

If there has a been a fixation that occured 3 seconds after the intervention has been triggered:

```
Select
   case when count(*) > 0
      then 1
      else 0
```

```
    end result
From
text_fix, rule_state
where rule_state.rule = "rule_3"
and text_fix.time_end > rule_state.time_stamp + 3000000;
```

5 FRONT-END

5.1 Responsibilities

The front-end is responsible for receiving update data from the middle-end and delivering, managing and removing the graphical changes required by the interventions. In addition, the front-end is also responsible for communicating with the back-end to indicate when a task change has occurred.

5.2 Components

We provide a sample front-end platform with support for certain graphical functionalities working with magazine-style visualization studies, however you should modify, add or develop your own front-end functionalities or workflow if needed.

The core data-flow between the front-end and middle-end occurs through a websocket at the */websocket* endpoint ("ws://localhost:8888/websocket" if running on the local machine).
The front-end client receives data from the middle-end in the form of a JSON string. Delivery of interventions will have a "deliver" field with a list of interventions, each having properties matching the columns of the **intervention** table. For example:

```
{ "deliver": [{ "name" : "bar_highlight", "function": "highlightVisOnly", "transition_in": 1000, "transition_out": 1000,
"delay": 0, "arguments": {"type": "reference", "id": 1, "bold": true, "bold_thickness": 3, "desat": true, "color": "green",
"arrow": false}, …] }
```

Similarly, removal of interventions will have a "removal" field, followed by a list of the intervention names to be removed. For example:

```
{ "remove": [ "bar_highlight", "legend_highlight", … ] }
```

Once data has been received, it is passed through a handler, which either applies the appropriate javascript function to apply all the required graphical changes. For example, we might define a simple handler:

```
var ws = new WebSocket("ws://localhost:8888/websocket");
   ws.onmessage = function (evt){
      var obj = JSON.parse(evt.data);
      if (obj.remove != null) {
          handleRemoval(obj);          //removes graphics
```

```
    } else if (obj.deliver != null) {
        handleDelivery(obj);      //highlights graphics
    }
}
```

Upon a task change the front-end can notify the back-end via the websocket connection, so long as there is a websockethandler class to receive messages from the front-end and change application accordingly. Using the default ApplicationWebSocket Class we have the following messages as defined actions:

- “Close”, closes the websocket and terminates the adaptation loops
- “Switch task:*”, switches to task number *

For a more complete example of the front-end and the source code see *./application/frontend/static/js/angular/app.js*

6 APPLICATION MAIN LOOP

6.1 Responsabilities

The application main loop is the entry point of the platform. It is responsible for initiating the main components of the plaform, rendering the adaptive system, and controlling the workflow of a user study if needed.

For reference, a fully implemented ‘Application main loop’ file is provided in the Github repository:

experimenter_platform_study_bars_final.py

6.1 Initiate platform

To initiate and run the platform, the experimenter must implement an entry file suitable for her system. By default this entry file must initiation and run the platform as follow:

```
import tornado
from tornado.options import define, options
import os.path

import sqlite3
import datetime
import json
import random
import sys

# Imports required for EYE TRACKING Code:
import time
from application.backend.eye_tracker import TobiiController
from application.middleend.adaptation_loop import AdaptationLoop
```

```
from application.application_state_controller import ApplicationStateController
from application.application_web_socket import ApplicationWebSocket

from application.backend.fixation_detector import FixationDetector
from application.backend.emdat_component import EMDATComponent
from application.backend.ml_component import MLComponent

import params


##########################################

define("port", default=8888, help="run on the given port", type=int)
TOBII_CONTROLLER = "tobii_controller"
APPLICATION_STATE_CONTROLLER = "application_state_controller"
ADAPTATION_LOOP = "adaptation_loop"
FIXATION_ALGORITHM = "fixation_algorithm"
EMDAT_COMPONENT = "emdat_component"
ML_COMPONENT = "ml_component"

class Application(tornado.web.Application):
    def __init__(self):
        #init platform and connects url with code
        self.tobii_controller = TobiiController()
        self.tobii_controller.waitForFindEyeTracker()
        self.tobii_controller.activate(self.tobii_controller.eyetrackers.keys()[0])
        self.app_state_control = ApplicationStateController(0)
        self.adaptation_loop = AdaptationLoop(self.app_state_control)

        self.fixation_component = FixationDetector(self.tobii_controller, self.adaptation_loop)
        self.emdat_component = EMDATComponent(self.tobii_controller, self.adaptation_loop, callback_time =
params.EMDAT_CALL_PERIOD)
        self.ml_component = MLComponent(self.tobii_controller, self.adaptation_loop, callback_time =
params.EMDAT_CALL_PERIOD, emdat_component = self.emdat_component)

        websocket_dict = {TOBII_CONTROLLER: self.tobii_controller,
                    APPLICATION_STATE_CONTROLLER: self.app_state_control,
                    ADAPTATION_LOOP: self.adaptation_loop,
                    FIXATION_ALGORITHM: self.fixation_component,
                    EMDAT_COMPONENT: self.emdat_component,
                    ML_COMPONENT: self.ml_component}

        #Define route handlers (see next section to define handlers)
        handlers = [ (r"/", MainHandler),
                      (r"/websocket", WebSocketHandler, dict(websocket_dict = websocket_dict)) ]

        #initializes web app
        tornado.web.Application.__init__(self, handlers, **settings)


#main function is first thing to run when application starts
def main():
    tornado.options.parse_command_line()
    #Application() refers to 'class Application'
    http_server = tornado.httpserver.HTTPServer(Application())
    http_server.listen(options.port)
    tornado.ioloop.IOLoop.current().start()

if __name__ == "__main__":
```

```
main()
```

6.2 Route Handlers and Control Flow

The route handling mechanism of Tornado is used to control for the workflow of user studies, which are meant to evaluate the adaptive systems. As such, the platform can support the display of a set of questionnaires, tests and tasks… More information about Tornado's handlers can be found here:

- https://www.tornadoweb.org/en/stable/web.html
- https://www.tornadoweb.org/en/stable/guide/structure.html

At least two handlers should be defined, one for displaying the system and one for initiating the websocket. In the code provided in Section 6.1, these handlers are named *MainHandler* and *WebSocketHandler*. Sample code for these handlers is:

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        #First page to render, for instance here index.html
        self.application.start_time = str(datetime.datetime.now().time())
        self.render('index.html')

    def post(self):
        self.redirect('/userID') #redirect to the next handlers, if any


class MMDWebSocket(ApplicationWebSocket):
    #Initiate
    def open(self):
        self.websocket_ping_interval = 0
        self.websocket_ping_timeout = float("inf")
        self.adaptation_loop.liveWebSocket = self
        print self.tobii_controller.eyetrackers

        self.start_detection_components()
        self.tobii_controller.startTracking()

    #Define what to do when receiving messages from the front-end via the websocket
    def on_message(self, message):
        #print("Received message: " + message)
        if (message == "next_task"): #sample message
            #do something
            return
        elif (message == "stop"): #sample message
            #do something
            return
        else:
            #unknown message
            return

    def on_close(self):
        self.app_state_control.logTask(user_id = self.application.cur_user)
```

To add more handlers, experimenters must:

1. Define them in the *handlers* array of the __init__() function of the *Application* class defined in Section 6.2.

2. Define the class for the new handlers. Sample code can be as follow:

```
class MyHandler1(tornado.web.RequestHandler):

    def get(self):
        self.render("webpage1.html") #render a page
    def post(self):
        #do something?

        self.redirect('/handler2)  #redirect to next handler if needed
```

Note that experimenters can solely use the platform to deliver adaptation without wanting to run a user study. In that case there is no need to define additional route handlers beyond the two basic ones (*MainHandler* and *WebSocketHandler*).