CDESF User Guide

Gabriel Tavares

August 2018

Version 1.0

Date 2018-08-27

Title Concept Drift in Event Stream Framework

Description A framework that handles concept drift in event log streams and identify anomalous cases. The goal of this guide is to document the framework, its classes, methods and attributes.

Depends Python (>= 3.5.3), matplotlib, pandas, numpy, graphviz

NeedsCompilation yes

Main

`Main`	*Sets up the environment for processing*

Description

The main class sets the initial parameters of the framework which will be used in subsequent classes. If the user intends to generate plots and metrics, the main class sets up the directories and paths.

Attributes

`path` : *str*

path to directory containing the event log

`process` : *str*

name of the file to be read

`gen_plot` : *bool*

if True triggers the plot generation

`gen_metrics` : *bool*

if True triggers the generation and recording of metrics

`plot_path` : *str*

path to save plots

`metrics_path` : *str*

path to save metrics

`th` : *int*

the time horizon window that controls Check Point ocurrence (unit: seconds)

`epsilon` : *float*

defines the maximum distance between an instance and a cluster for that instance to be considered inside the cluster (DenStream parameter)

`lambda_` : *float*

sets the importance of historical data for the current clusters (DenStream parameter)

`beta` : *float*

defines the threshold for a micro-clusters weight for it to be considered either an o-micro-cluster or a p-micro-cluster (DenStream parameter)

`mu` : *int*

is the minimum weight an overall neighbourhood needs to be considered a core object (DenStream parameter)

`n_features` : *int*

number of features for DenStream to use as attributes, should be set to 2 since we have only two attributes, GD_{trace} and GD_{time} (DenStream parameter)

Notes

The event log may be in CSV format. We use pandas library to read the event log in a dataframe fashion. The columns must be `case ID`, `activity name` (event), `timestamp`, `process name`. Event logs that do not conform to this will throw an error and won't be consumed by the framework.

CDESF

Description

CDESF simulates the stream and handles the event to event management. If the event log contains more than one process, it can be handled properly. Moreover, this class calls the methods for metrics retrieval.

Methods

`__init__(self, stream, th, gen_plot, plot_path, gen_metrics, metrics_path, denstream_kwargs)`
`setProcess(name, case_id, act_name, act_timestamp, event_index)`
`eventProcessing()`

`__init__` *Initializes the attributes*

Arguments

`stream` : *numpy.ndarray*

contains the loaded event log

`th` : *int*

the time horizon window that controls Check Point ocurrence (unit: seconds)

`gen_plot` : *bool*

if True triggers the plot generation

`plot_path` : *str*

path to save plots

`gen_metrics` : *bool*

if True triggers the generation and recording of metrics

`metrics_path` : *str*

path to save metrics

`denstream_kwargs` : *dict*

packed dictionary containing all DenStream parameters

`processes` : *dict*

dictionary containing processes where keys are names and values are process objects

Notes

Except from `process` all other arguments are attributes of `__init__`

`setProcess` *Handles multiple processes and arranges case setup*

Description

This function receives the basic case attributes and sets them up accordingly. If a process is new in the stream, the function instantiates a new process object and adds it to the `process` dictionary.

Arguments

`name` : *str*

name of the process used as a key in the `process` dictionary

`case_id` : *str*

case identifier

`act_name` : *str*

activity name

`act_timestamp` : *str*

activity timestamp (current format of timestamp is set to YYYY/mm/dd HH:MM:SS.fff, e.g: 2011/03/23 14:00:23.000)

`event_index` : *int*

index of the event in the stream

`eventProcessing` *Simulates the event stream*

Description

Simulates the event stream by iterating through the `stream` variable, calls `setProcess` at each event and also controls the metrics recording.

Process

Description

The `Process` class deals with the core concepts of CDESF controlling several operations such as the setting of a case, graphs management (Nyquist), DenStream triggering, plotting results and metrics recording.

Methods

`__init__(name, timestamp, th, gen_plot, plot_path, gen_metrics, metrics_path, denstream_kwargs)`
`convertAct(act_name)`
`getListForGP(case_id)`
`getList()`
`getCase(case_id)`
`delCases()`
`GPova()`
`getCasePoints()`
`genClusterMetrics()`
`genCaseMetrics(event_index, index)`
`genPmgMetrics()`
`genPlots()`
`initialiseCDESF(current_time)`
`setCase(case_id, act_name, act_timestamp, event_index)`
`clusterMetrics()`
`caseMetrics()`
`pmgStateByCp()`

Attributes

`gen_plot` : *bool*

if True, triggers the plot generation

`plot_path` : *str*

path to save plots

`gen_metrics` : *bool*

if True, triggers the metrics recording

`metrics_path` : *str*

path to save metrics

`event_count` : *int*

counts the total number of events that went through the stream

`total_cases` : *set*

a set containing the unique cases that went through the stream `cases` : *list*

list of current cases maintained by the framework

`name` : *str*

name of the process

`th` : *int*

the time horizon window that controls Check Point ocurrence (unit: seconds)

`act_dic` : *dict*

stores the relation between activity names and aliases

`possible_act_names` : *collections.deque*

stores the remaining aliases to be used

`gpCreation` : *bool*

controls the first TH cycle

`check_point` : *datetime.datetime*

a datetime controlling the check point timespan

`cp_count` : *int*

counts the number of check points

`nyquist` : *int*

stores the Nyquist value

`cp_cases` : *int*

stores the number of new cases in the current check point

`process_graph` : *dict*

stores the Process Model Graph

`denstream` : *DenStream*

stores the DenStream object

`cluster_metrics` : *list*

auxiliary list for cluster metrics

`case_metrics` : *list*

auxiliary list for case metrics

`pmg_by_cp` : *list*

auxiliary list for PMG recording

`__init__` *Initializes the process*

Description

This function sets up a new process, defining its name, and preparing initial attributes.

Arguments

`name` : *str*

name of the process

`timestamp` : *str*

timestamp of the first event which will be used as a mark for the first check point

`th` : *int*

the time horizon window that controls Check Point ocurrence (unit: seconds)

`gen_plot` : *bool*

if True, triggers the plot generation

`plot_path` : *str*

path to save plots

`gen_metrics` : *bool*

if True, triggers the metrics recording

`metrics_path` : *str*

path to save metrics

`denstream_kwargs` : *dict*

packed dictionary containing all DenStream parameters

`convertAct` *Converts activity name*

Description

Receives an activity name and converts it using the `possible_act_names` deque.

Arguments

`act_name` : *str*

activity's name

Returns

`act_dic[act_name]` : *str*

the converted activity's name

`getListForGP` *Retrieves a list of cases*

Description

Retrieves a list of cases from the first check point excluding the case passed as an argument.

Arguments

`case_id` : *str*

the case identifier

Returns

`trace_list` : *list*

a list of lists containing all retrieved traces

`timestamp_list` : *list*

a list of lists containing all retrieved timestamps

`getList` *Retrieves a list of cases*

Description

Retrieves a list of cases traces and timestamps.

Returns

`trace_list` : *list*

a list of lists containing all retrieved traces

`timestamp_list` : *list*

a list of lists containing all retrieved timestamps

`getCase` *Retrieves a case index*

Description

Returns the case index from the `cases` list.

Arguments

`case_id` : *str*

the case identifier

Returns

`index` : *int*

the index of `case_id` in the list if found or None if not found

`delCases` *Releases older cases*

Description

Releases older cases based on the Nyquist value. The result is stored in `cases`.

`GPova` *Calculates metrics for cases in the first check point*

Description

Calculates GD_{trace} and GD_{time} for cases in the first time horizon cycle. For every case, a graph is constructed with all other cases and the metrics computed. Note that none of this graphs are the Process Model Graph, instead they are only auxiliary graphs used once for metrics extraction for the initial cases.

`getCasePoints` *Retrieves a case point*

Description

Returns the `point` attribute of all cases in `cases`.

Returns

`cases_points` : *list*

a list of case points used for plotting

`genClusterMetrics`	*Generates cluster metrics*

Description

Generates cluster metrics and saves them in the `cluster_metrics` attribute.

`genCaseMetrics`	*Generates case metrics*

Description

Generates case metrics and saves them in the `case_metrics` attribute.

Arguments

`event_index` : *int*

index of the current event

`index` : *int*

index of the current case

`genPmgMetrics`	*Generates graph metrics*

Description

Saves the Process Model Graph at all check points in the `pmg_by_cp` attribute.

`genPlots`	*Generates plots*

Description

Controls the plotting and handles all necessary arguments.

`initialiseCDESF`	*Initializes the framework after the first time horizon*

Description

Initializes system variables, creates the first Process Model Graph and initializes DenStream.

Arguments

`current_time` : *datetime.datetime*

the last event time, used for check point marking

`setCase` *Handles case management*

Description

The core function in the `Process` class. Sets a new case and controls the check point. If `gpCreation` is True, calculates the case metrics and uses them to train DenStream, recalculates the Nyquist and releases old cases if necessary.

Arguments

`current_time` : *datetime.datetime*

the last event time, used for check point marking

`case_id` : *str*

the case identifier

`act_name` : *str*

activity's name

`act_timestamp` : *str*

activity timestamp (current format of timestamp is YYYY/mm/dd HH:MM:SS.fff)

`event_index` : *int*

index of the current event

`clusterMetrics` *Saves cluster metrics*

Description

Converts `cluster_metrics` into a dataframe and saves it.

`caseMetrics` *Saves case metrics*

Description

Converts `case_metrics` into a dataframe and saves it.

`pmgStateByCp` *Saves PMG metrics*

Description

Converts `pmg_by_cp` into a dataframe and saves it.

Graph

Description

Generates and maintains the graphs and calculates GD_{trace} and GD_{time}.

Methods

```
timeProcessing(time_list)
normGraph(graph)
createGraph(trace_list, time_list)
computeFeatures(graph, trace, raw_time)
mergeGraphs(proc_graph, cp_graph)
```

`timeProcessing` *Retrieves time differences from case's activities*

Description

This function receives a list of lists containing timestamps from the selected cases. It processes the timestamps and calculates the time difference within activities for all separate cases.

Arguments

`time_list` : *list*

list of case's timestamps

Returns

`outer_time` : *list*

list of time differences within activities

Example

time_list = [['2010/09/21 09:00:21.000', '2010/09/21 10:00:00.000'], ['2010/09/21 10:00:59.000', '2010/09/21 15:00:51.000', '2010/09/22 10:00:43.000']]

outer_time = timeProcessing(time_list)

print(outer_time) [[3579], [17992, 68392]]

`normGraph` *Normalizes graph weights*

Description

Time and weight normalization for each `Transition` in the graph. Time normalization is the mean time.

Arguments

`graph` : *dict*

graph is a dictionary of transitions

Returns

`graph` : *dict*

the normalized graph

`createGraph` *Creates a graph from a set of traces and timestamps*

Description

Creates a graph based on a list of traces and timestamps.

Arguments

`trace_list` : *list*

a list of lists containing traces from selected cases

`time_list` : *list*

a list of lists containing events timestamps from selected cases

Returns

`graph` : *dict*

the created graph with its set of transitions

`computeFeatures` *Calculates GD_{trace} and GD_{time}*

Description

Receives a graph, trace and timestamps from a selected case and computes the metrics for that case. Contains several rules for GD_{trace} and GD_{time}.

Arguments

`graph` : *dict*

graph is a dictionary of transitions

`trace` : *list*

a list containing the activities from a case

`raw_time` : *list*

a list containing activities timestamps

Returns

`gwd` : *float*

the computed graph weight distance (GD_{trace})

`gwd` : *float*

the computed graph time distance (GD_{time})

`mergeGraphs` *Merges two graphs*

Description

Receives two graphs and merge them. Important to notice that the first graph has more weight and incorporates the second graph. 5% of the original graph weight is disconted (decay).

`proc_graph` : *dict*

represents the process graph

`cp_graph` : *dict*

the Check Point graph constructed using the newest cases

Transition

Description

Class to represent transitions from one activity to another, retains the transition as a tuple and saves the time span.

Methods

`__init__(name)`
`add(weight, time)`

`__init__` *Initializes a new transition*

Description

Receives a name and initializes the attributes of a new transition.

Arguments

`name` : *tuple*

the name of a transition in a tuple format, e.g. ('Act1', 'Act2')

Attributes

`name` : *tuple*

the name of a transition in a tuple format, e.g. ('Act1', 'Act2')

`weight` : *int*

sum of weights in a transition

`time` : *float*

sum of timestamps in a transition

`count` : *int*

counter for the occurrence of a transition

`time_norm` : *float*

stores the normalized time, which is given by the `time` divided by `count`

`weight_norm` : *float*

stores the normalized weight, scales all transitions from 0 (least occurrence) to 1 (highest occurrence)

`add` *Computes a new occurrence of a transition*

Description

Receives a weight and a timestamp and adds them cumulatively to `weight` and `time`, respectively. Also adds the counter.

Arguments

`weight` : *int*

weight to be added

`time` : *float*

inter-activity time to be added

DenStream

Description

Manages the DenStream algorithm and implements classes `MicroCluster` and `Cluster`.

Methods

```
__init__(n_features, lambda_, beta, epsilon, mu)
euclidean_distance(point1, point2)
find_closest_p_mc(point)
find_closest_o_mc(point)
decay_p_mc(last_mc_updated_index=None)
decay_o_mc(last_mc_updated_index=None)
merge(case, t)
train(case)
is_normal(point)
DBSCAN(buffer)
generate_clusters()
generate_outliers_clusters()
class MicroCluster
class Cluster
```

Attributes

`n_features` : *int*

the number of features DenStream must consider, in our case is always set to 2, since we have two attributes (GD_{trace} and GD_{time})

`lambda` : *float*

sets the importance of historical data for the current clusters

`beta` : *float*

defines the threshold for a micro-clusters weight for it to be considered either an o-micro-cluster or a p-micro-cluster

`epsilon` : *float*

defines the maximum distance between an instance and a cluster for that instance to be considered inside the cluster

`mu` : *int*

is the minimum weight an overall neighbourhood needs to be considered a core object

`p_micro_clusters` : *dict*

dictionary containing all p-micro-clusters

`o_micro_clusters` : *dict*

dictionary containing all o-micro-clusters

`label` : *int*

cluster identification label for plotting

`time` : *int*

time in single units

`initiated` : *bool*

controls if DenStream has been initialized

`all_cases` : *set*

contains cases that went through DenStream

`__init__` *Initializes the DenStream class*

Description

Initializes the DenStream class.

Arguments

`n_features` : *int*

the number of features DenStream must consider, in our case is always set to 2, since we have two attributes (GD_{trace} and GD_{time})

`lambda` : *float*

sets the importance of historical data for the current clusters

`beta` : *float*

defines the threshold for a micro-clusters weight for it to be considered either an o-micro-cluster or a p-micro-cluster

`epsilon` : *float*

defines the maximum distance between an instance and a cluster for that instance to be considered inside the cluster

`mu` : *int*

is the minimum weight an overall neighbourhood needs to be considered a core object

---

`euclidean_distance` *calculates the Euclidian Distance*

Description

Computes the Euclidean Distance between two points.

Arguments

`point1` : *numpy.ndarray*

array position of a point

`point2` : *numpy.ndarray*

array position of a point

`find_closest_p_mc` *Finds the closest p-micro-cluster*

Description

Finds the closest `p_micro_cluster` to the point according to the Euclidean Distance between the point and the cluster's centroid

Arguments

`point` : *numpy.ndarray*

array position of a point

Returns

`i` : *int*

index of the `p_micro_cluster`

`p_micro_cluster` : *MicroCluster*

the correspondent p-micro-cluster which the point was added

`dist` : *numpy.float64*

distance between the p-micro-cluster and the point

`find_closest_o_mc` *Finds the closest o-micro-cluster*

Description

Finds the closest `o_micro_cluster` to the point according to the Euclidean Distance between the point and the cluster's centroid

Arguments

`point` : *numpy.ndarray*

array position of a point

Returns

`i` : *int*

index of the `o_micro_cluster`

`o_micro_cluster` : *MicroCluster*

the correspondent o-micro-cluster which the point was added

`dist` : *numpy.float64*

distance between the o-micro-cluster and the point

`decay_p_mc` *Decays p-micro-cluster's weight*

Description

Decays the weight of all `p_micro_clusters` for the exception of an optional parameter `last_mc_updated_index`

Arguments

`last_mc_updated_index` : *int*

index of the last updated micro-cluster

`decay_o_mc` *Decays o-micro-cluster's weight*

Description

Decays the weight of all `o_micro_clusters` for the exception of an optional parameter `last_mc_updated_index`

Arguments

`last_mc_updated_index` : *int*

index of the last updated micro-cluster

`merge` *Merges point to micro-cluster*

Description

Tries to add a point to the existing `p_micro_clusters` at time `t`. Otherwise, tries to add that point to the existing `o_micro_clusters`. If fails, creates a new `o_micro_clusters` with that new point.

Arguments

`case` : *denstream.Case (namedtuple)*

case containing identifier and graph distance metrics

`t` : *int*

time in single units

`train` *Trains DenStream*

Description

Trains Denstream by updating its p-micro-clusters and o-micro-clusters with a new point

Arguments

`case` : *denstream.Case (namedtuple)*

case containing identifier and graph distance metrics

`is_normal` *Finds if point is inside a p-micro-cluster*

Description

Finds if a point is inside any p-micro-cluster and returns a bool.

Arguments

`point` : *numpy.ndarray*

array position of a point

`DBSCAN` *Performs DBSCAN to initialize DenStream*

Description

Performs DBSCAN to create initial p-micro-clusters. Works by grouping points with distance <= `epsilon` and filtering groups that are not dense enough (`weight` >= `beta` * `mu`).

Arguments

`buffer` : *list*

a buffer containing all points which will be used in DBSCAN

`generate_clusters` *Generates c-micro-clusters*

Description

Performs DBSCAN to create the final c-micro-clusters. Works by grouping dense enough p-micro-clusters (weight >= `mu`) with distance <= 2 * `epsilon`

Returns

`dense_groups` : *list*

a `Cluster` object list of dense enough groups of p-micro-clusters `not_dense_groups` : *list*

a `Cluster` object list of not dense enough groups of p-micro-clusters

---

`generate_outlier_clusters` *Generates a list of o-micro-clusters*

Description

Generates a list of o-micro-clusters.

Returns

a Cluster list with o-micro-clusters

`MicroCluster` *A class to represent micro-clusters*

Description

Represents micro-clusters

`Cluster` *A class to represent c-micro-clusters*

Description

Represents c-micro-clusters

MicroCluster

Description

The class represents a micro-cluster and its attributes.

Methods

```
__init__(n_features, creation_time, lambda_)
centroid()
radius()
radius_with_new_point(point)
update(case)
```

Attributes

`CF` : *numpy.ndarray*

the weighted linear sum of the points inside the micro-cluster

`CF2` : *numpy.ndarray*

the weighted squared sum of the points inside the micro-cluster

`weight` : *float*

micro-cluster weight

`creation_time` : *int*

creation time in single units

`case_ids` : *set*

set of case identifiers inside the micro-cluster

`lambda` : *float*

sets the importance of historical data for the current clusters

---

`__init__` *Initializes the MicroCluster*

Description

Initializes the MicroCluster attributes.

Arguments

`n_features` : *int*

the number of features DenStream must consider, in our case is always set to 2, since we have two attributes (GD_{trace} and GD_{time})

`creation_time` : *int*

creation time in single units

`lambda_` : *float*

sets the importance of historical data for the current clusters

---

`centroid` *Computes the centroid*

Description

Computes and returns the micro-cluster's centroid value, which is given by

`CF` divided by `weight`.

`radius` *Computes the radius*

Description

Computes and returns the micro-cluster's radius.

`radius_with_new_point` *Computes the radius considering an additional point*

Description

Computes the micro-cluster's radius considering a new point. The returned value is then compared to `epsilon` to check whether the point must be absorbed or not.

Arguments

`point` : *numpy.ndarray*

contains GD_{trace} and GD_{time}

`update` *Updates MicroCluster attributes*

Description

Updates the micro-cluster weights either considering a new case or not.

Arguments

`case` : *denstream.Case (namedtuple)*

case containing identifier and graph distance metrics

Cluster

Description

Class that represents a cluster.

Methods

`__init__(id, centroid, radius, weight, case_ids)`

Description

Initializes a cluster.

Arguments

`id` : *int*

cluster identifier

`centroid` : *numpy.ndarray*

cluster centroid position

`radius` : *float*

cluster radius (measure of how far is the cluster influence)

`weight` : *float*

cluster weight

`case_ids` : *set*

set of case identifiers inside that cluster

Notes

All arguments from `__init__` are `Cluster` attributes.

denstream

Description

A set of methods to control and plot DenStream data.

Methods

```
gen_data_plot(denstream, window_cases, alpha_range=(0, 1.0))
plot_clusters(process_name, total_cases, event_index, cp, points, outliers,
c_clusters, p_clusters, outlier_clusters, n, epsilon, th, plot_path,
cases_dict=None)
cluster_metrics(total_cases, event_index, cp, c_clusters, p_clusters,
outlier_clusters)
gen_graphviz(p)
```

`gen_data_plot` *Generates plot data*

This function organizes the necessary parameters for plotting.

Arguments

`denstream` : *DenStream*

stores the DenStream object

`window_cases` : *list*

a list of cases which will be plotted

`alpha_range=(0, 1.0))` : *tuple*

a tuple to control the color range

Returns

`points` : *list*

a list containing points with their ids and positions attached

`outliers` : *list*

a list of anomalous cases which will be plotted as outliers

`c_clusters` : *list*

a `Cluster` list with c-micro-clusters

`p_clusters` : *list*

a `Cluster` list with p-micro-clusters

`o_clusters` : *list*

a `Cluster` list with o-micro-clusters

`plot_clusters` *Plot clusters and graphs*

Plot all types of clusters, their respective graphs, and anomalous cases. Saves the plot according to the `plot_path` attribute.

Arguments

`process_name` : *str*

the process name

`total_cases` : *int*

the total number of cases that went through the stream

`event_index` : *int*

the current event index in the stream

`cp` : *int*

the current check point

`points` : *list*

list of points to be plotted (comes from the function `gen_data_plot`)

`outliers` : *list*

list of anomalous cases to be plotted (comes from the function `gen_data_plot`)

`c_clusters` : *list*

list of c-micro-clusters to be plotted (comes from the function `gen_data_plot`)

`p_clusters` : *list*

list of p-micro-clusters to be plotted (comes from the function `gen_data_plot`)

`o_clusters` : *list*

list of o-micro-clusters to be plotted (comes from the function `gen_data_plot`)

`n` : *str*

the concatenation of `Process.cp_count` and `Process.event_count` used as the plot name when the file is saved

`epsilon` : *float*

defines the maximum distance between an instance and a cluster for that instance to be considered inside the cluster

`th` : *int*

the time horizon window that controls Check Point ocurrence (unit: seconds)

`plot_path` : *str*

path to save plots

`cases_dict` : *dict*

a dictionary of cases used for identification

`cluster_metrics` *Retrieves cluster metrics*

Generates cluster metrics to record them. Works in conjunction with `Process.clusterMetrics()`.

Arguments

`total_cases` : *int*

the total number of cases that went through the stream

`event_index` : *int*

the current event index in the stream

`cp` : *int*

the current check point

`c_clusters` : *list*

list of c-micro-clusters

`p_clusters` : *list*

list of p-micro-clusters

`o_clusters` : *list*

list of o-micro-clusters

Returns

`out` : *list*

carries metrics to be saved

`gen_graphviz` *Generate graphs for plotting*

Generates the graphs of each cluster using graphviz technology and saves them in an auxiliary folder, which is later retrieved by `plot_clusters` for the complete plotting.

Arguments

`p` : *tuple*

a tuple containing information related to graph generation and saving

Case

Description

Represents a case and stores its attributes, such as activities, timestamps, GD_{trace}, GD_{time}, among others.

Methods

```
__init__(case_id)
setActivity(act_name, act_timestamp)
getLastTime()
setGwd(gwd)
setTwd(twd)
```

`__init__` *Initializes a new case*

Description

Receives a case identifier and initializes the attributes of a new case.

Arguments

`case_id` : *str*

the case identifier

Attributes

`id` : *str*

the case identifier

`activities` : *list*

list of `Activity` object

`trace` : *list*

list of converted activity names

`timestamp` : *list*

list of event timestamps

`gwd` : *float*

stores the GD_{trace} value

`twd` : *float*

stores the GD_{time} value

`point` : *denstream.Case (namedtuple)*

point is a namedtuple inherited from denstream class and is used for plotting

---

`setActivity` *Sets a new activity in a case*

Description

Creates a new `Activity` and appends it to `activities`.

Arguments

`act_name` : *str*

activity's name

`act_timestapm` : *str*

the timestamp of the recorded activity

`getLastTime` *Returns the time of the last event*

Description

Retrieves the last event timestamp and is used to sort cases before being deleted.

`setGwd` *Stores GD_{trace}*

Description

Receives a value corresponding to GD_{trace} and stores it in both `gwd` and `point` attributes.

Arguments

`gwd` : *float*

the calculated GD_{trace}

`setTwd` *Stores GD_{time}*

Description

Receives a value corresponding to GD_{time} and stores it in both `twd` and `point` attributes.

Arguments

`twd` : *float*

the calculated GD_{time}

Activity

Description

The lowest level representation of an activity.

Methods

`__init__(name, timestamp)`

`__init__` *Initializes the activity*

Description

Initializes an `Activity` setting its name and timestamp.

Arguments

`name` : *str*

activity name