CDESF Tutorial

August 19, 2020

Abstract

Concept Drift in Event Stream Framework (CDESF) is a completely available tool for the detection of drifts in event streams. Moreover, CDESF supports several online Process Mining tasks, such as process model evolution and anomaly detection. In this tutorial, we present a practical guide to the main aspects of CDESF and how to explore the results.

1 Introduction

CDESF is built for Python 3.6 or higher versions. The library dependencies required to run CDESF are the following: networkx $\sim=$ 2.4, numpy $\sim=$ 1.18.4, setuptools $\sim=$ 41.2.0, pandas $\sim=$ 1.0.4, scipy $\sim=$ 1.5.1, matplotlib $\sim=$ 3.2.2, seaborn $\sim=$ 0.10.1, pygraphviz $\sim=$ 1.6, graphviz $\sim=$ 2.38.0. The first step is to clone the repository[1] into your local machine. Following, am example on how to trigger the processing of event streams in CDESF.

```
from cdesf2.utils import read_csv
from cdesf2.core import CDESF

path = "demo"
filename = "hospital_billing.csv"

event_stream_test = read_csv(f"{path}/{filename}")
cdesf = CDESF(name='hospital_billing',
              time_horizon=259200,
              lambda_=0.05,
              beta=0.2,
              epsilon=0.2,
              mu=4,
              stream_speed=100,
              n_features=2,
              gen_metrics=True,
              gen_plot=True)
cdesf.run(event_stream_test)
```

The hyperparameters that control event processing are are crucial to execution of CDESF. We list them in detail next:

[1] https://github.com/gbrltv/cdesf2

- *path*: path of the folder containing the event log files.
- *filename*: event log file to be processed.
- *name*: the name of the process you want to analyze (also used to save the output).
- *time_horizon*: window that controls both model updating and forgetting mechanism. Smaller time horizons trigger more frequent updates, hence, a higher rate of concept drifts. Larger time horizons obtain fewer updates, hence, the process model is more stable and less drift warnings will be triggered. A good value is the mean time of a case execution. Time horizon uses seconds as a time unit.
- *lambda*: the decay factor impacts on how much historical data is relevant. Higher values implies in less importance to historical data. The optimal range lies between 0.05 and 0.4.
- *beta*: controls the promotion of micro-clusters, consequently impacting on drift sensibility. Higher values are more robust to drifts. Instead, lower values trigger more drift warnings. Usually the range of values is between 0.1 and 0.6.
- *epsilon*: represents the maximum actuation range of a micro-cluster. Higher values enable bigger micro-clusters, and consequently more cover area. Optimal values range from 0.05 to 0.5.
- *mu*: used in conjunction with *beta*, controls the minimum density of a micro-cluster to be considered core behavior.
- *stream_speed*: controls the application of the decay factor, i.e., influences how fast the micro-clusters weight decay.
- *n_features*: the number of dimensions DenStream must consider, here it is set to 2 since we have two attributes (graph distance and time distance). However, CDESF can consider multiple dimensions too.
- *gen_metrics*: controls metrics generation and saving. Useful to investigate case labels and micro-cluster behavior.
- *gen_plot*: control the generation of feature space plots after each event. Useful for detailed analysis of the micro-clusters evolution.

2 Reading files

CDESF framework is capable to read and analyze event streams to perform real-time tasks, such as process discovery, anomaly detection, clustering and concept drift detection. For that, it has to read the log file and extract the data. This operation is made by the following reading function:

```
def read_csv(path: str) -> np.ndarray:
    event_stream = pd.read_csv(path,
                               usecols=['case', 'activity', '
    timestamp'],
                               parse_dates=['timestamp'],
                               infer_datetime_format=True,
                               memory_map=True)
    event_stream['activity'].replace(' ', '_', regex=True, inplace=
    True)
    return event_stream.values
```

This function uses a pandas component to read the file at the given path. Moreover, it also converts timestamps to the datetime object and transform names of activities to make them suitable for CDESF execution. After that, the function returns the stream of events as an array. In detail for pandas function *read_csv* we use the parameters

- *usecols*: it is required that the log file contains the columns *case*, *activity* and *timestamp*. Otherwise the file will not be read.
- *parse_dates*: selects the column containing the date and time when events were executed.
- *infer_datetime_format*: infers the datetime string format and transforms to a python object.
- *memory_map*: improves performance by mapping the file directly to memory.

3 Case and Activity Structures

This section covers the basic structures of CDESF. As presented previously, the reading function prepares the log file and identifies cases, activities and timestamps. CDESF has two data structures to represent basic Process Mining concepts: Case and Activity. The activities and timestamps taken are respectively activity name and timestamp attributes of Activity objects. Moreover, the case column provides the identifiers of Case objects. In details, the Case object has the following attributes:

- *id*: the case identifier, taken and assigned from the event log.
- *activities*: a list of activities of type Activity executed within the case.
- *graph_distance*: value produced by the comparison between the case flow and the process model.
- *time_distance*: value produced by the comparison of time between case activities and the process model.

Case has also two properties. The first one calculate its point, which is a two-element array containing *graph_distance* and *time_distance*. The point is used later in the clusterization step. The property's structure is the following:

```
@property
def point(self) -> np.array:
    return np.array([self.graph_distance, self.time_distance])
```

The second property retrieves the last event timestamp:

```
@property
def last_time(self) -> datetime:
    return self.activities[-1].timestamp
```

Furthermore, Case has also other useful methods:

- `set_activity(activity_name, activity_timestamp)`

  Allows the attachment of a new Activity object to the Case.

- `get_trace()`

  Retrieves the list of activities within that case.

- `get_timestamp()`

  Retrieves the list of activities timestamps.

4 Running CDESF

With the basic structures presented, we can proceed to analyze how CDESF runs. As explained previously, the first step is reading the log file and storing its return in a variable. Then, a *run* function can be triggered with the stream and the hyperparameters. The function simulates the event stream by iterating through the stream returned from the reading. Each new event is processed (*process_event*) using the case, the activity name and the timestamp as input from the stream. If it is the first event in the stream, the function sets the activity timestamp as the first check point. Once the stream is over, an output with the total number of drifts and the drift points is shown on the console. Moreover, a *visualization* function creates a to plot with all drifts to provide further insights. The detailed function is:

```
for index, event in enumerate(stream):
            self.event_index = index
            case_id, activity_name, activity_timestamp
                 = (event[0], event[1], event[2])
            if index == 0:
                self.check_point = activity_timestamp
            self.process_event(case_id, activity_name,
   activity_timestamp)
```

4.1 Processing events

The *process_event* function is the core of CDESF. For new events that belong to never seen cases, it instantiates a new case, otherwise, it adds the new activity and timestamp to an existing case through the *set_case* function. In details *set_case* as follows:

```
    index = self.get_case(case_id)
    if index is None: # new case
        self.cases.append(Case(case_id))
        self.check_point_cases += 1
        index = self.get_case(case_id)
    self.cases[index].set_activity(act_name, act_timestamp) #
    appends new activity
    self.cases.insert(0, self.cases.pop(index))
    return index
```

It checks through *get_case* if a case exists and it creates one if it does not exist. After creating the case, CDESF adds the activity with the timestamp to the case list in the first position, maintaining a first in first out logic. Then *process_event* checks if the difference between the last timestamp of the modified case and the check point is greater than the *time_horizon* hyperparameter. If the condition is met for the first time, the initialization function (*initialize_cdesf*) is triggered. The goal is to create the first process model, extract the graph distances of the existing cases and initialize the online clustering feature space. First, *initialize_cdesf* sets system variables and creates the first Process Model Graph (PMG) and, with the extracted case distances to the graph, CDESF initializes the micro-clusters used by the DenStream algorithm in the online clustering step. If the hyperparameters *gen_metrics* and *gen_plot* are set to *True*, it also initializes metrics and graphic functions, respectively. The part of the code regarding the initialization of the system variables and DenStream is the following:

```
self.nyquist = self.check_point_cases * 2
self.process_model_graph = initialize_graph(nx.DiGraph(), self.
    cases)
self.initialize_case_metrics()
self.denstream.dbscan(self.cases)
groups = self.denstream.generate_clusters()
    for group in groups[0]:
        for cluster in group:
            self.active_core_clusters.add(cluster.id)
```

4.2 Graph operations

As you can see form the code above, the PMG graph structure relies on the networkx DiGraph object. CDESF uses it to build PMG as directed graphs since it successfully represents directly-follows relations. There are three main graph operations:

- *normalize_graph*: makes a normalization of time and weight for each edge in the graph. From a trace perspective, the normalization is the occurrence of a specific transition divided by the value of the most occurred transition. The result of this operation is referred to as *weight_normalized*. For the time perspective, edge normalization computes the mean time of a transition between activities. The result of this operation is referred to as *time_normalized*.

```
max_weight = max([attributes['weight']
                  for n1, n2, attributes
                  in graph.edges(data=True)])
for node1, node2, data in graph.edges(data=True):
    data['weight_normalized'] = data['weight'] / max_weight
    data['time_normalized'] = data['time'] / data['weight']
return graph

```

- *initialize_graph*: this function initializes a graph based on the weights and time differences from a list of cases. The resulting graph has activities as nodes and *time* and *weight* as edges' attributes. It creates the edge if it does not already exist, otherwise it increments its weight by 1 and time by time difference for each directly-follows relation. The normalized graph is returned.

```
    trace_list, time_list = extract_cases_time_and_trace(
    case_list)
    time_list = time_difference(time_list)

    for trace, time in zip(trace_list, time_list):
        for i in range(len(trace)-1):
            edges = (trace[i], trace[i+1])
            if edges not in graph.edges:
                graph.add_edge(*edges, weight=1, time=time[i])
            else:
                graph[edges[0]][edges[1]]['weight'] += 1
                graph[edges[0]][edges[1]]['time'] += time[i]

    return normalize_graph(graph)

```

- *merge_graph*: this function receives two graphs and merge them. Typically the two graphs are the PMG and the graph built using only new cases, i.e., cases that appeared in the last time horizon, named as Check Point Graph (CPG). First of all this function decays the 5% of the PMG's edges weight. For each edge in CPG, it checks if there is a corresponding edge in the PMG. If it already exists, it updates the weight and the time of the PMG edge with the corresponding weight and time of the CPG edge. If it does not exists in the PMG, it creates the new edge and assigns the weight and the time value taken from CPG. Finally the function returns the normalized (PMG).

```
for node1, node2, data in process_model_graph.edges(data=True)
    :
    data['weight'] *= 0.95

for node1, node2, data in check_point_graph.edges(data=True):
    path = (node1, node2)
    if path in process_model_graph.edges:
        process_model_graph[node1][node2]['weight'] += data['
    weight']
        process_model_graph[node1][node2]['time'] += data['
    time']
```

```
    else:
        process_model_graph.add_edge(*path, weight=data['
    weight'], time=data['time'])

return normalize_graph(process_model_graph)

```

4.3 Case metrics

After process model creation, case metrics are initialized in the function *initialize_case_metrics*, which extracts case distances, GD_{trace} (*graph_distance*) and GD_{time} (*time_distance*), for cases in the first time horizon cycle. Here the detailed code:

```
def initialize_case_metrics(self) -> None:
    for case in self.cases:
        graph_distance, time_distance = extract_case_distances(self
    .process_model_graph, case)
        case.graph_distance = graph_distance
        case.time_distance = time_distance
```

The *extract_case_distances* function receives a graph and a case and returns graph and time distances based on Equations 1 and 2:

$$GD_{trace}(tr) = \frac{\sum_{i=1}^{T_{tr}} 1 - PMG_{weight}(tr[i])}{T_{tr}} \tag{1}$$

$$GD_{time}(tr) = \log_{10}\left(\frac{\sum_{i=1}^{T_{tr}} |PMG_{time}(tr[i]) - tr[i]_{time}|}{\sum_{j=1}^{T_{tr}} PMG_{time}(tr[j])}\right) \tag{2}$$

Given a trace tr, T_{tr} is the total amount of edges in tr, $tr[i]$ corresponds to the ith edge in tr, $tr[i]_{time}$ is the time delta in $tr[i]$ and $tr[i]_{weight}$ is the weight of $tr[i]$.

4.4 Online clustering

After the case metrics are initialized, *intialize_cdesf* triggers DenStream, which starts its micro-clusters using the DBSCAN algorithm. First of all a case is taken and all other cases in the buffer passed in input, The input for this function is the list of cases currently active. One by one, cases are analyzed and their euclidean distances are measured. A micro-cluster is a group of cases with a distance lower than the *epsilon* hyperparameter. This process is repeated recursively until all cases are used. If the formed group of cases has enough weight, i.e., the weight is higher than *beta*mu* (hyperparameters), then a micro-cluster is created with them. If weight is lower than *beta*mu*, the cases are anomalous, since they belong to a low density region. The complete code of DBSCAN is the following:

```
def dbscan(self, buffer: List) -> None:
    used_cases = set()
    for case in (case for case in buffer if case.id not in
    used_cases):
```

```
        used_cases.add(case.id)
        group = [case]
        for other_case in (case for case in buffer if case.id not
    in used_cases):
            dist = self.euclidean_distance(case.point, other_case.
    point)
            if dist <= self.epsilon:
                group.append(other_case)

        weight = len(group)
        if weight >= self.beta * self.mu:
            new_p_mc = MicroCluster(id_=self.mc_id,
                                    n_features=self.n_features,
                                    creation_time=self.time,
                                    lambda_=self.lambda_)
            for case_ in group:
                used_cases.add(case_.id)
                new_p_mc.update(case_)
                self.all_cases[case_.id] = new_p_mc.id
            self.mc_id += 1
            self.p_micro_clusters.append(new_p_mc)
        else:
            used_cases.remove(case.id)
```

Once CDESF is initialized, for each new event, process_event() calculates the distances of the case and assigns them to its attributes. Then it triggers DenStream using the function *train* to accomodate the case in the online clustering space.

```
def train(self, case: Case):
    micro_cluster_updated = self.add_point(case)
    self.all_cases[case.id] = micro_cluster_updated

    self.no_processed_points += 1
    if self.no_processed_points % self.stream_speed == 0:
        self.time += 1
        for mc in self.p_micro_clusters:
            mc.decay()

    if self.time % self.tp == 0:
        for i, mc in enumerate(self.p_micro_clusters):
            if mc.weight < self.beta * self.mu:
                self.p_micro_clusters.pop(i)

        for i, mc in enumerate(self.o_micro_clusters):
            to = mc.creation_time
            e = ((math.pow(2, - self.lambda_ * (self.time - to +
    self.tp)) - 1) /
                 (math.pow(2, - self.lambda_ * self.tp) - 1))
            if mc.weight < e:
                self.o_micro_clusters.pop(i)
```

This way, *train* adds a new point to a micro-cluster trough the *add_point* function based using distances in a sequence of steps:

- first, Denstream tries merge the case into the closest potential micro-cluster:

```
i, closest_p_mc, _ = self.find_closest_mc(case.point, self.
    p_micro_clusters)
if closest_p_mc.radius_with_new_point(case.point) <= self.
    epsilon:
    closest_p_mc.update(case)
    return closest_p_mc.id

```

- if it fails, it tries to merge the case outlier micro-cluster and eventually try to promote it to a potential micro-cluster. To have a promotion the outlier micro-cluster must have its weight higher than *beta*mu*:

```
i, closest_o_mc, _ = self.find_closest_mc(case.point, self.
    o_micro_clusters)
if closest_o_mc.radius_with_new_point(case.point) <= self.
    epsilon:
    closest_o_mc.update(case)
    if closest_o_mc.weight > self.beta * self.mu:
        self.o_micro_clusters.pop(i)
        self.p_micro_clusters.append(closest_o_mc)
        return closest_o_mc.id

```

- if the merge fails, a new outlier micro-cluster is created with this case:

```
new_o_mc = MicroCluster(id_=self.mc_id,
                        n_features=self.n_features,
                        creation_time=self.time,
                        lambda_=self.lambda_)
self.mc_id += 1
new_o_mc.update(case)
self.o_micro_clusters.append(new_o_mc)
return new_o_mc.id

```

A dedicated function (*find_closest_mc*) to find the closest micro-clusters is needed for these operations. It takes the point and the micro-cluster list and it uses the euclidean distance between the point and the cluster's centroid to find the closest cluster.

```
if len(micro_cluster_list) == 0:
    raise NoMicroClusterException

distances = [(i, self.euclidean_distance(point, np.array(cluster.
    centroid)))
             for i, cluster in enumerate(micro_cluster_list)]
i, dist = min(distances, key=lambda i_dist: i_dist[1])
return i, micro_cluster_list[i], dist
```

After adding the point, the *train* function decays the potential micro-clusters weights. This decay frequency is controlled by the *lambda* hyperparameter. Outlier micro-clusters with weights lower than *epsilon* are deleted.

4.5 Check point

In *process_event*, if the difference between the event timestamp and the check point is greater than the *time_horizon*, a check point is triggered. The main goal of a check point is to update the PMG and control memory consumption by releasing older cases. Moreover, the PMG is saved at each check point. The following code controls check point execution:

```
if (current_time - self.check_point).total_seconds() > self.time_horizon:
    self.check_point = current_time
    self.check_point_update()

    self.metrics.save_pmg_on_check_point(self.process_model_graph, self.cp_count)
    if self.gen_metrics:
        self.metrics.save_case_metrics_on_check_point()
        self.metrics.save_cluster_metrics_on_check_point()
```

The *check_point_update* releases older cases according to the Nyquist frequency theorem and updates the PMG using the *merge_graphs* function:

```
def check_point_update(self):
    self.cp_count += 1
    if len(self.cases) > self.nyquist:
        self.release_cases_from_memory()
        if self.check_point_cases > 5:
            self.nyquist = self.check_point_cases * 2
        check_point_graph = initialize_graph(nx.DiGraph(), self.cases)
        self.process_model_graph = merge_graphs(self.process_model_graph, check_point_graph)
    self.check_point_cases = 0
```

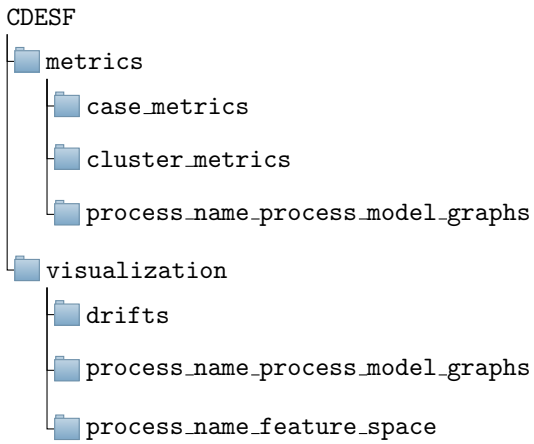
If the length of the list of cases is greater than the Nyquist valye, older cases are released. This operation is guaranteed by *release_case_from_memory* function:

```
def release_cases_from_memory(self) -> None:
    self.cases = self.cases[:self.nyquist]
```

Also *check_point_update* updates the Nyquist value according to the theorem using the frequency of new cases arrives in the last time horizon.

5 Metrics and visualization

Metrics and plotting are triggered in CDESF by two flags: *gen_metrics* and *gen_plot*. Two root folders contain CDESF produced outputs:



Note that "process_name" will be filled with the *name* hyperparameter. The case metrics (stored inside "metrics/case_metrics") contains the stream index position, the timestamp, the check point number, the case identifier, the graph and time distances and the label of a case. The label is either normal or anomalous. If a case belongs to a low density region, DenStream classifies it as anomalous, otherwise, it is a normal execution. The cluster metrics (stored inside "metrics/cluster_metrics") contains the stream index position, the timestamp, the check point, the cluster identification, position, radius, weight and type (core, potential or outlier). The metrics files are useful to understand the evolution of objects in the stream processing. For instance, a cluster can be traced since its discovery in the feature space to analyze how it behaves in the events processing. Metrics functions are started in *initialize_cdesf*. Further, it is controlled by *save_case_metrics_on_check_point* and *save_cluster_metrics_on_check_point*. Moreover, the PMG is saved at each check point (*save_pmg_on_check_point*), making possible to observe its evolution.

- *compute_case_metrics*: this function takes in input from CDESF the event index, the timestamp, the current check point value, the case to be saved, and a boolean flag label to control if case is normal or anomalous. *Compute_case_metrics* appends the case metrics to the *case_metrics* attribute.

- *save_case_metrics_on_check_point*: works in conjunction with *compute_case_metrics*. It saves the case metrics into the output file on every check point. Moreover, the function frees the *case_metrics* attribute to save memory.

- *compute_cluster_metrics*: generates the cluster metrics and save them into the *cluster_metrics* attribute. Cluster metrics come from the online processing, provided by DenStream. The saved values are the event index, the check point timestamp, the current check point number and cluster attributes.

- *save_cluster_metrics_on_check_point*: works in conjunction with *compute_cluster_metrics*.

It takes the data written in the *cluster_metrics* attribute and saves into a file, then, relases the cluster_metrics attribute.

- *save_pmg_on_check_point*: this function takes the PMG as input, then, it saves the PMG in a *JSON* file and also plots it using the function for visualization. This function plots the graph using the pygraphviz library.

Another crucial visualization is provided by the *feature_space* function, which plots the clusters and cases in the feature space after every new event in the stream. Core micro-clusters (black ellipses) represent common process behavior, potential micro-clusters (blue ellipses) represent potential core behavior and outlier micro-clusters (red ellipses) represent outlier behavoir. Normal and anomalous cases are also shown in black dots and yellow markers, respectively. The x-axis is marked as GD_{trace} and the y-axis is marked as GD_{time}. The plots are saved under the "process_name_feature_space" folder.

6 Experiments

To test the effectiveness of the CDESF toolkit, we have analyzed a synthetic event log, which contains one drift and 20% of anomalous traces[2].

The hyperparameters for the framework were set to: *time_horizon*=259200, *lambda_*=0.05, *beta*=0.2, *epsilon*=0.6, *mu*=4, *stream_speed*=100, *n_features*=2, *gen_metrics*=True, *gen_plot*=True.

The first output is the console drift warnings. This fast identification of drifts is interesting in scenarios where actions must be taken in a short timespan. Figure 1 shows a drift warning using the synthetic event log. Furthermore, Figure 1 points to a drift in event 2532 and that the new detected core micro-cluster is the number 4. To complement this information, the cluster metrics file can be analyzed to further investigate cluster 4 evolution. Figure 2 presents the metrics ouput for this event log. The first appearance of cluster number 4 is on event 2525 (row 1546), as an outlier micro-cluster. Note that on previous events there were only cluster number 0, 1, 2 and 3. From event 2525 on, all subsequent events maintain the cluster 4, as Figure 2 shows.

Once the drift is found, the feature space at that event can be analyzed for further insights. Figure 3 presents the feature space before and after the drift has occurred. Initially, the feature space is depicted before cluster 4 appears (Fig. 3a). Then, cluster 4 is detected as an outlier micro-cluster since its weight is still low at this point (Fig. 3b). After another event arrival (Fig. 3c), cluster 4 increases its weight, making the region more dense, and steps to a potential micro-cluster. Finally, on event 2532 (Fig. 3d), cluster 4 is promoted to a core micro-cluster, i.e., a new common behavior. This way, a drift has been detected. With that, the drift detection can be seen from various complementing perspectives (console output, metrics evaluation and feature space evolution).

[2] `https://ieee-dataport.org/open-access/synthetic-event-streams`

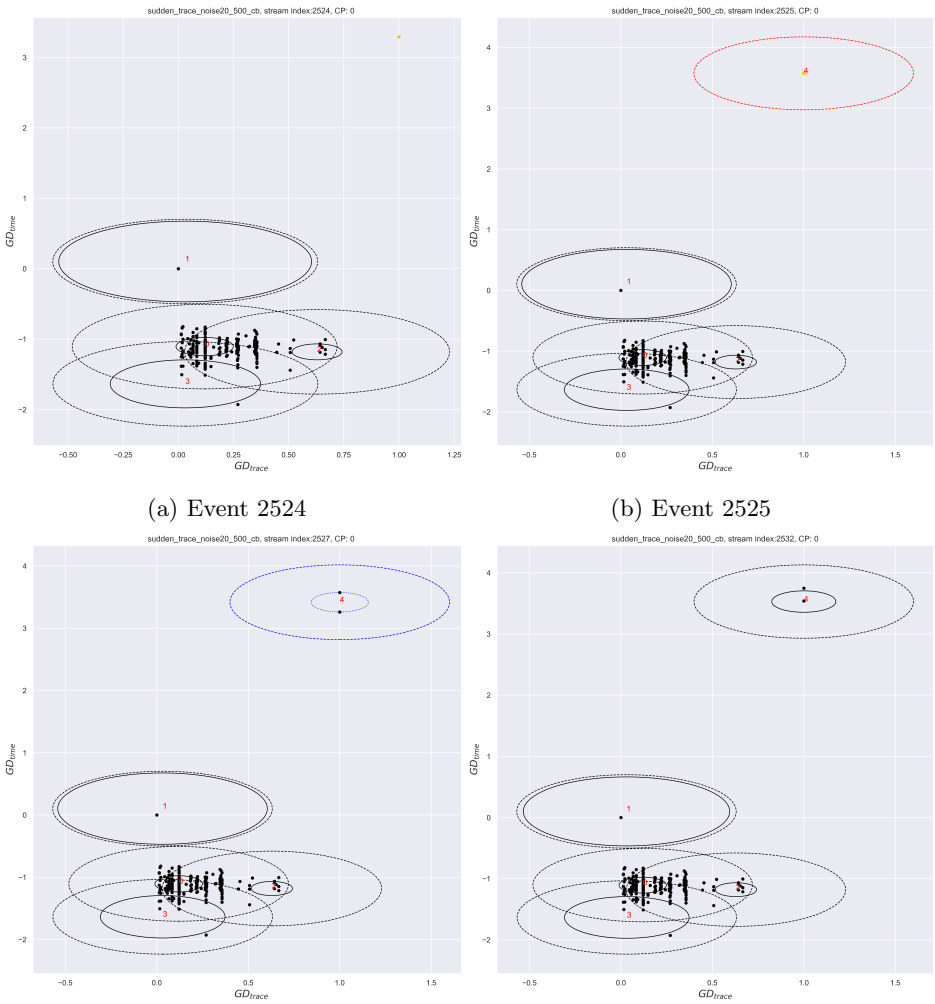
```
tests/test_pipeline.py::test_pipeline DRIFT ALERT
New core behavior detected: cluster 4
Cluster weight: 4
Cluster radius: 0.17517490031151506
Cases in cluster: [250, 252]

Total number of drifts: 1
Drift points: [2532]
PASSED
```

Figure 1: The console output of the synthetic event log. A drift warning and its related information is available. The total number of drifts and their positions is listed at the end of processing.

```
1534 2523,2019-01-14 04:40:00,0,0,0.11954613584681513,-1.105444261412353,0.12988614093400516,494.6819204265275,core micro-cluster
1535 2523,2019-01-14 04:40:00,0,2,0.6288660399168414,-1.1799062052800964,0.11309860569876394,9.373439104589828,core micro-cluster
1536 2523,2019-01-14 04:40:00,0,3,0.03212308372963736,-1.634372632158441,0.3407106946450931,24.43702179456719,core micro-cluster
1537 2523,2019-01-14 04:40:00,0,1,0.032255777653574826,0.10617886197807086,0.5815868810109573,31.002197830724832,core micro-cluster
1538 2524,2019-01-14 05:02:30.844000,0,0,0.11954613584681513,-1.105444261412353,0.12988614093400516,494.6819204265275,core micro-cluster
1539 2524,2019-01-14 05:02:30.844000,0,2,0.6288660399168414,-1.1799062052800964,0.11309860569876394,9.373439104589828,core micro-cluster
1540 2524,2019-01-14 05:02:30.844000,0,3,0.03212308372963736,-1.634372632158441,0.3407106946450931,24.43702179456719,core micro-cluster
1541 2524,2019-01-14 05:02:30.844000,0,1,0.031247853828336596,0.10286100041932097,0.5727261138810581,32.00219783072483,core micro-cluster
1542 2525,2019-01-14 05:10:24.573000,0,0,0.11954613584681513,-1.105444261412353,0.12988614093400516,494.6819204265275,core micro-cluster
1543 2525,2019-01-14 05:10:24.573000,0,2,0.6288660399168414,-1.1799062052800964,0.11309860569876394,9.373439104589828,core micro-cluster
1544 2525,2019-01-14 05:10:24.573000,0,3,0.03212308372963736,-1.634372632158441,0.3407106946450931,24.43702179456719,core micro-cluster
1545 2525,2019-01-14 05:10:24.573000,0,1,0.031247853828336596,0.10286100041932097,0.5727261138810581,32.00219783072483,core micro-cluster
1546 2525,2019-01-14 05:10:24.573000,0,4,1.0,3.5741290019411682,0.0,1.0,outlier micro-cluster
1547 2526,2019-01-14 05:20:00,0,0,0.11933136680597618,-1.1054879051848783,0.12975868802389515,495.6819204265275,core micro-cluster
1548 2526,2019-01-14 05:20:00,0,2,0.6288660399168414,-1.1799062052800964,0.11309860569876394,9.373439104589828,core micro-cluster
1549 2526,2019-01-14 05:20:00,0,3,0.03212308372963736,-1.634372632158441,0.3407106946450931,24.43702179456719,core micro-cluster
1550 2526,2019-01-14 05:20:00,0,1,0.031247853828336596,0.10286100041932097,0.5727261138810581,32.00219783072483,core micro-cluster
1551 2526,2019-01-14 05:20:00,0,4,1.0,3.5741290019411682,0.0,1.0,outlier micro-cluster
```

Figure 2: Metrics showing the appearance of a new core micro-cluster, i.e., the detection of a concept drift.

Another complementing output is the PMG, which is presented in Figure 4. And the cumulative number of drifts and their position is presented after the stream processing (Fig. 5).



(a) Event 2524

(b) Event 2525

(c) Event 2527

(d) Event 2532

Figure 3: Evolution of the feature space detecting a concept drift.

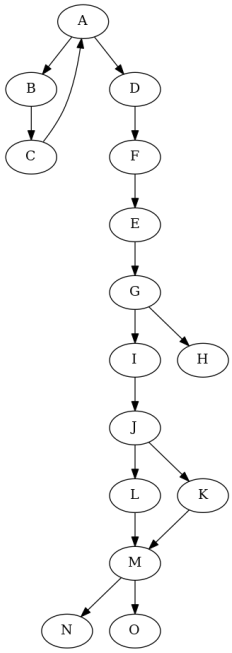


Figure 4: PMG of the synthetic event log at the first check point.

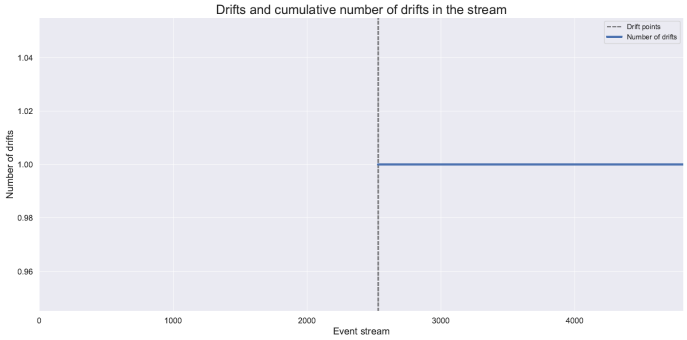


Figure 5: Relation of the event stream processing, number of detected drifts and their positions. Vertical lines represent the drift positions and the curve shows the cumulative number of drifts, which is 1 in this case.