Evolving Tokenized Transducer Sequential Detector

January 26, 2026

Contents

1 About

This package has been partly supported under Competitiveness and Cohesion Operational Programme from the European Regional and Development Fund, as part of the Integrated Anti-Fraud System project no. KK.01.2.1.01.0041 (IAFS). This package has also been partly supported by the European Regional Development Fund under the grant KK.01.1.1.01.0009 (DATACROSS).

The package comprises Evolving Tokenized Transducer (ETT) (Krleža et al., 2019), developed to learn and detect data sequences. Although ETT was primarily developed for the process discovery purpose, it can be used to learn and detect any data sequence, e.g., data sequences from data streams or process instance flows from event streams. Although we use terms *data streams* and *event streams* throughout this documentation, limited datasets and logs fall into this category as well.

This package is still in development and will be significantly extended over the next years. We are also aware that things are currently not perfect, and that there are most certainly some bugs hidden around. As a book always has at least one more typo, a piece of software always has at least one more bug.

1.1 Evolving Tokenized Transducer

ETT is a transducer, which is essentially a type of Moore machine, defined as

$$ETT = (Q, \Sigma, \Gamma, \mathcal{T}, \Omega, \eta, \pi, \Delta, S, F) \tag{1}$$

where:

- Q is a set of all states, which helps to define the ETT structure,

- Σ is an input alphabet set taken from the input source,

- Γ is an output alphabet set generated by ETT (the reason why this is a transducer),

- $\mathcal{T}$ is a set of tokens, which describe a set of current data sequences handled by ETT,

- Ω is a state and output alphabet mapping relation,

- η is a state and token mapping relation, which assigns tokens to states,

- π is a token selection function, which is used to select relevant tokens based in the input symbol and data item,

- Δ is a state-token transition and mapping function, which defines transitions in the ETT structure,

- $S \subseteq Q$ is a set of starting states,

- $F \subseteq Q$ is a set of final states.

ETT uses a set of auxiliary algorithms called **actions** for its work.

ETT actions :

- **Extending actions** - Capable to extend the ETT structure, which gives ETT capability to learn new sequences, or to add new data items in the existing data sequences.

- **Pushing actions** - Performing token push throughout the ETT structure, which resembles as sequence detection.

ETT modes :

- **Learning mode** - Uses both extending and pushing actions for intertwining or data sequence learning and detection.

- **Detection mode** - Uses only pushing actions. ETT in this mode is not extended in any circumstances, i.e., it can only detect previously learned data sequences.
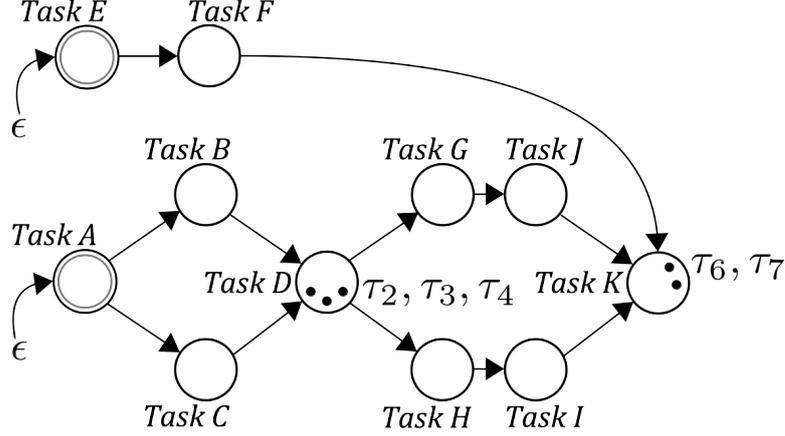
Figure 1: An ETT example

The foundation of ETT is to have a single structure that comprises multiple tokens. Each one of these tokens represents one active data sequence. In the process discovery context, each token represents one process instance flow. ETT is a non-deterministic transducer, whose current context can be described by the set of tokens. By pushing tokens through the ETT structure on input data we advance a data sequence or a process instance flow. Token pushing is performed by the *pushing actions* described previously. If there is no token that can be pushed and ETT is in *learning mode*, *extension actions* are invoked to extend ETT with new data sequence element or process activity by extending the ETT structure. After *extension actions* are performed, tokens are pushed.

1.2 Data and event stream processing

This Sequence Detector is capable of processing data and event streams. For this reason ETT was implemented in C++, to boost its performance memory and processing wise. ETT is a simple mechanism that requires a limited input data structure. On the other hand, a data stream can be composed from various data sources, having different schemata and abundant amount of attributes. For that reason, we need some pre-processing and pre-classification, to compose a consolidated data stream suitable for passing into ETT.
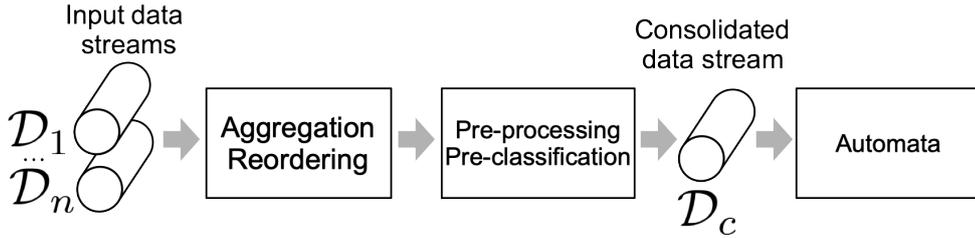
Figure 2: The Sequence Detector hybrid architecture

The Sequence Detector comprises the following two stages:

- **The pre-processing stage** - In which we take data items from multiple input data streams and rearrange them into a single output pre-processed data stream. This involves data item aggregation, reordering and pre-processing (e.g., dimension reduction).

- **The pre-classification stage** - In which we take the pre-processed data stream and classify data items for creating the input ETT alphabet Σ and additional information needed for ETT. So reduced data items are written in the output consolidated data stream $\mathcal{D}_c$.

The consolidated data stream at step k, denoted as $\mathcal{D}_c^{[k]}$, comprises ordered k data items that are the result of the pre-processing and pre-classification stages, which can be defined as

$$\mathcal{D}_c^{[k]} = \{d_c^{[1]}, ..., d_c^{[k]}\}$$
$$|\mathcal{D}_c^{[k]}| = k, k \in \mathbb{N}$$
$$\mathcal{D}_c^{[k]} \subseteq \mathcal{I}^{[k]} \times T^{[k]} \times T^{[k]} \times \mathcal{C}^{[k]} \tag{2}$$
$$\forall 1 \leq j \leq k (d_c^{[j]} = (id^{[j]}, t_s^{[j]}, t_e^{[j]}, class^{[j]}) \in \mathcal{D}_c^{[k]} \wedge j \in \mathbb{N})$$

where at step k, $\mathcal{I}^{[k]}$ is the set of all used context identifiers, $T^{[k]}$ is the set of all timestamps, and $\mathcal{C}^{[k]}$ is the set of all output classes from the pre-processing procedures, which is the same as the input alphabet of the subsequent automata. Each data item is a tuple that comprises a context identifier $id^{[j]} \in \mathcal{I}^{[k]}$, a starting timestamp $t_s^{[j]} \in T^{[k]}$, an ending timestamp $t_e^{[j]} \in T^{[k]}$, and a class of the data item $class^{[j]} \in \mathcal{C}^{[k]}$.

2 Overview

2.1 Pre-processing

The pre-processing stage takes a number of input data streams and performs aggregation, consolidation and reordering on them. The result is a single output stream which is then passed into the pre-classification stage.

We define a set of input streams, each one to be from a specific IT system. To pass a set of input streams into the pre-processing code, we need a named list whose element names relate to the names of IT systems and element values are data frames that represents a slice of the related data stream. First we create four exemplatory slices of input data streams.

ER registration system stream

```
> st1 <- data.frame(patient=c("C156","C156","E9383","C167"),
+                    time=c("05.12.2019. 10:30:20","05.12.2019. 11:59:07","07.12.2019. 08:34:12",
+                           "07.12.2019. 10:45:11"),
+                    age=c(12,12,26,76),
+                    fever=c(TRUE,TRUE,TRUE,FALSE),
+                    action=c("registration","release","registration","registration"),
+                    can_walk=c(TRUE,TRUE,FALSE,TRUE))
```

	patient	time	age	fever	action	can_walk
1	C156	05.12.2019. 10:30:20	12.00	TRUE	registration	TRUE
2	C156	05.12.2019. 11:59:07	12.00	TRUE	release	TRUE
3	E9383	07.12.2019. 08:34:12	26.00	TRUE	registration	FALSE
4	C167	07.12.2019. 10:45:11	76.00	FALSE	registration	TRUE

Table 1: ER registration system data stream slice

Finish it by transforming string dates into POSIXct data types.

```
> st1 <- transform.data.frame(st1,time=as.POSIXct(st1$time,format="%d.%m.%Y. %H:%M:%S"))
```

ER triage system

```
> st2 <- data.frame(patient=c("C156","C156","E9383","E9383","C167","C167"),
+                    time=c("05.12.2019. 10:41:00","05.12.2019. 12:12:00","07.12.2019. 09:56:00",
+                           "07.12.2019. 11:32:00","07.12.2019. 11:01:00","07.12.2019. 13:14:15"),
+                    diagnosis=c(NA,"J04.0",NA,"A41.9",NA,"N41.0"),
+                    action=c("biomarker","release","biomarker","hospital_ic","biomarker","hospital_nc"),
+                    decription=c("suspect. laryng...","course of antibiotics",
+                           "high fever,in shock state!! URGENT!","septic shock? IC..",
+                           "cannot pee,catheter","urology hospitalization"))
```
Finish it by transforming string dates into POSIXct data types.

```
> st2 <- transform.data.frame(st2,time=as.POSIXct(st2$time,format="%d.%m.%Y. %H:%M:%S"))
```

	patient	time	diagnosis	action	decription
1	C156	05.12.2019. 10:41:00		biomarker	suspect. laryng...
2	C156	05.12.2019. 12:12:00	J04.0	release	course of antibiotics
3	E9383	07.12.2019. 09:56:00		biomarker	high fever,in shock state!! URGENT!
4	E9383	07.12.2019. 11:32:00	A41.9	hospital_ic	septic shock? IC..
5	C167	07.12.2019. 11:01:00		biomarker	cannot pee,catheter
6	C167	07.12.2019. 13:14:15	N41.0	hospital_nc	urology hospitalization

Table 2: ER triage system data stream slice

Bio-laboratory system

```
> st3 <- data.frame(request_id=c("2019_645553","2019_654331","2019_654331","2019_654331","2019_655376",
+                              "2019_655376"),
+             request_org=c("ER","ER","ER","ER","ER","ER"),
+             ext_id=c("C156","E9383","E9383","E9383","C167","C167"),
+             date=c("05.12.2019.","07.12.2019.","07.12.2019.","07.12.2019.","07.12.2019.",
+                    "07.12.2019."),
+             biomarker=c("WBC","WBC","CRP","LAC","WBC","CRP"),
+             final=c(14.6,13.11,345.0,4.5,11.43,67.0),stringsAsFactors=FALSE)
```

	request_id	request_org	ext_id	date	biomarker	final
1	2019_645553	ER	C156	05.12.2019.	WBC	14.60
2	2019_654331	ER	E9383	07.12.2019.	WBC	13.11
3	2019_654331	ER	E9383	07.12.2019.	CRP	345.00
4	2019_654331	ER	E9383	07.12.2019.	LAC	4.50
5	2019_655376	ER	C167	07.12.2019.	WBC	11.43
6	2019_655376	ER	C167	07.12.2019.	CRP	67.00

Table 3: Biolab system data stream slice

Finish it by transforming string dates into POSIXct data types.

```
> st3 <- transform.data.frame(st3,date=as.POSIXct(st3$date,format="%d.%m.%Y."))
```

Hospital registration system

```
> st4 <- data.frame(patient_id=c("I93382","N94511"),
+             ext_id=c("E9383","C167"),
+             time_in=c("07.12.2019. 11:35:46","07.12.2019. 12:11:49"),
+             diagnosis=c("A41.9","N41.0"),
+             type=c("IC","NC"),
+             time_release=c("15.12.2019. 08:52:11","11.12.2019. 14:02:11"))
```

	patient_id	ext_id	time_in	diagnosis	type	time_release
1	I93382	E9383	07.12.2019. 11:35:46	A41.9	IC	15.12.2019. 08:52:11
2	N94511	C167	07.12.2019. 12:11:49	N41.0	NC	11.12.2019. 14:02:11

Table 4: Hospital system data stream slice

Finish it by transforming string dates into POSIXct data types.

```
> st4 <- transform.data.frame(st4,time_in=as.POSIXct(st4$time_in,format="%d.%m.%Y. %H:%M:%S"),
+                          time_release=as.POSIXct(st4$time_release,format="%d.%m.%Y. %H:%M:%S"))
```

Previous data streams could be read from Kafka for example. To pre-process these specific data streams, we need to create a pre-processor that inhertis *HSC_PP* class and implements the *process* method, as the following example:

```
> HSC_PP_Hospital <- function(...) {
+   structure(list(),class = c("HSC_PP_Hospital","HSC_PP"))
+ }
> preprocess.HSC_PP_Hospital <- function(x, streams, ...) {
+   # perform some meaningful checking on the input data streams
+   res <- data.frame(stringsAsFactors=FALSE)
+   reg_stream <- streams[["registration_system"]]
+   for(j in 1:nrow(reg_stream)) {
+     el <- reg_stream[j,]
+     cz <- case_when(el[,"action"]=="registration"~"ER registration",
+                 el[,"action"]=="release"~"ER release")
```

```
+        res <- rbind(res,data.frame(id=el[,"patient"],class=cz,time=el[,"time"],out=cz,WBC=NA,CRP=NA,LAC=NA))
+      }
+    triage_stream <- streams[["triage"]]
+    for(j in 1:nrow(triage_stream)) {
+      el <- triage_stream[j,]
+      if(nrow(res[res$id==el[,"patient"] & res$class=="ER triage",])==0)
+        res <- rbind(res,data.frame(id=el[,"patient"],class="ER triage",
+                              time=min(triage_stream[triage_stream[,"patient"]==el[,"patient"],"time"]),
+                              out="ER triage",WBC=NA,CRP=NA,LAC=NA))
+    }
+    biolab_stream <- streams[["biolab"]]
+    for(j in 1:nrow(biolab_stream)) {
+      el <- biolab_stream[j,]
+      if(nrow(res[res$id==el[,"ext_id"] & res[,"class"]=="Biomarker assessment",])==0) {
+        if(el[,"request_org"]=="ER") {
+          t1_time <- triage_stream[triage_stream[,"patient"]==el[,"ext_id"] &
+                                triage_stream[,"action"]=="biomarker","time"]+1
+          res <- rbind(res,data.frame(id=el[,"ext_id"],class="Biomarker assessment",
+                                  time=t1_time,out="Biomarker assessment",
+                                  WBC=NA,CRP=NA,LAC=NA))
+        }
+      }
+      res[res[,"id"]==el[,"ext_id"] & res[,"class"]=="Biomarker assessment",
+          el[,"biomarker"]] <- el[,"final"]
+    }
+    hospital_stream <- streams[["hospital"]]
+    for(j in 1:nrow(hospital_stream)) {
+      el <- hospital_stream[j,]
+      cz1 <- paste0("Admission to ",el[,"type"])
+      res <- rbind(res,data.frame(id=el[,"ext_id"],class=cz1,time=el[,"time_in"],
+                              out=cz1,WBC=NA,CRP=NA,LAC=NA))
+      res <- rbind(res,data.frame(id=el[,"ext_id"],class="Hospital release",
+                              time=el[,"time_release"],out="Hospital release",
+                              WBC=NA,CRP=NA,LAC=NA))
+    }
+    res <- res[order(res[,"time"]),] # order this event stream slice
+    return(list(obj=x,res=res))
+ }
> pp <- HSC_PP_Hospital()
> input_streams <- list(registration_system=st1,triage=st2,biolab=st3,hospital=st4)
> event_stream <- preprocess(pp,input_streams)$res
```

A list comprising *obj* and *res* elements has to be returned from the *process* method of a pre-processor class. The *obj* element returns the same S4 object passed in the *process* method, i.e., the modified variable x. The *res* element must return the composed event stream.

	id	class	time	out	WBC	CRP	LAC
1	C156	ER registration	1575541820.00	ER registration			
5	C156	ER triage	1575542460.00	ER triage			
8	C156	Biomarker assessment	1575542461.00	Biomarker assessment	14.60		
2	C156	ER release	1575547147.00	ER release			
3	E9383	ER registration	1575707652.00	ER registration			
6	E9383	ER triage	1575712560.00	ER triage			
9	E9383	Biomarker assessment	1575712561.00	Biomarker assessment	13.11	345.00	4.50
4	C167	ER registration	1575715511.00	ER registration			
7	C167	ER triage	1575716460.00	ER triage			
10	C167	Biomarker assessment	1575716461.00	Biomarker assessment	11.43	67.00	
11	E9383	Admission to IC	1575718546.00	Admission to IC			
13	C167	Admission to NC	1575720709.00	Admission to NC			
14	C167	Hospital release	1576072931.00	Hospital release			
12	E9383	Hospital release	1576399931.00	Hospital release			

Table 5: The resulting event stream slice

2.1.1 Data and event stream slicing rules

Let us define n input data streams as $I = \{is_1, is_2, ..., is_n\}$. Each data stream is sliced in the same number of slices $is_i = \{s_1(is_i), ..., s_k(is_i)\} \in I$. The event stream made by the pre-processing code can be defined as $E = \bigcup I = \{s_1(E), ..., s_k(E)\}$ and comprises the same number of slices as input data streams. Each of the k slices can be aggregated separately $s_i(E) = \bigcup_{j=1}^{k} s_i(is_j)$.

Timeframe of each event stream slice is bound by a minimal starting timestamp of the composing slices $t_s(s_i(E)) = \min_{j=1}^{k} t_s(s_i(is_j))$ and a maximal ending timestamp of the composing slices $t_e(s_i(E)) = \max_{j=1}^{k} t_e(s_i(is_j))$. Event stream slices cannot overlap time-wise. This means

$$\nexists 1 \leq i, j \leq k(j > i \wedge t_s(s_j(E)) \leq t_e(s_i(E))) \tag{3}$$

We need to take care about the current status of an individual input data stream, i.e., some of the input data streams could be lagging behind, time-wise. Once all input data streams reach a certain time point tp_j, we slice them all between $(tp_{i-1}, tp_i]$. This is the bound of the slice i, meaning

$$tp_{i-1} < t_s(s_i(E)) \leq t_e(s_i(E)) \leq tp_i \tag{4}$$

which can happen when slice $s_i(E)$ events do not occur exactly at time points tp_{i-1} or tp_i.

We can determinte the time point tp_i by examining last data items in all input data streams. If we take $l_i = |is_i|$, and the last data item in the input data stream is_i as $d_i^{[l_i]} \in is_i$, under assumption that there is no back-logging and that each input data stream comprises data item from one unique data source, we can choose the time point as

$$tp_i = \min_{i=1}^{k} t_e(d_i^{[l_i]}) \tag{5}$$

If one of the systems allows back-logging, i.e., retroactive work, the whole slicing time point estimation must be adjusted for the allowed back-logging period. This can create some serious problems, for example when we try to capture ongoing fraudulent processes and react to the ongoing alerts in real-time as much as possible. Overnight batch processing systems might destroy the capability of ETT to track process instances in real-time.

Modernizing IT systems and adopting service-oriented architectural principles can greatly improve the ability to track ongoing processes.

2.2 Pre-classification

Once we create a slice of the event stream, each event in the stream must have sufficient data for the next step: pre-classification. We begin with writing a concrete pre-classifier for the event stream. Pre-classifier is instantiated from a class that inherits *HSC_PC* class, and implements the *classify* method. A simple implementation would be:

```
> HSC_PC_Hospital <- function(...) {
+   structure(list(),class = c("HSC_PC_Hospital","HSC_PC"))
+ }
> classify.HSC_PC_Hospital <- function(x, event_stream, ...) {
+   # perform some meaningful checking on the supplied event stream
+   res <- data.frame(stringsAsFactors=FALSE)
+   for(i in 1:nrow(event_stream)) {
+     event <- event_stream[i,]
+     symbol <- case_when(
+       event$class=="ER registration" ~ "ER_REG",
+       event$class=="ER triage" ~ "ER_TR",
+       event$class=="ER release" ~ "ER_REL",
+       event$class=="Biomarker assessment" ~ "BIO_A",
+       event$class=="Admission to IC" ~ "IC",
+       event$class=="Admission to NC" ~ "NC",
+       event$class=="Hospital release" ~ "H_REL"
+     )
+     if(symbol=="BIO_A") {
+       symbol <- paste0(symbol,case_when(is.na(event$WBC) ~ "#WBC=NONE",
+                                         event$WBC>11 ~ "#WBC=EL",TRUE ~ "#WBC=OK"))
+       symbol <- paste0(symbol,case_when(is.na(event$CRP) ~ "#CRP=NONE",
+                                         event$CRP>50 ~ "#CRP=EL",TRUE ~ "#CRP=OK"))
+       symbol <- paste0(symbol,case_when(is.na(event$LAC) ~ "#LAC=NONE",
+                                         event$LAC>4 ~ "#LAC=EL",TRUE ~ "#LAC=OK"))
+     }
+     res <- rbind(res,data.frame(id=event$id,class=event$class,time=event$time,out=event$out,
+                        .clazz=symbol))
+   }
```

```
+    return(res)
+ }
> pc <- HSC_PC_Hospital()
> consolidated_stream <- classify(pc,event_stream)
```

	id	class	time	out	.clazz
1	C156	ER registration	1575541820.00	ER registration	ER_REG
2	C156	ER triage	1575542460.00	ER triage	ER_TR
3	C156	Biomarker assessment	1575542461.00	Biomarker assessment	BIO_A#WBC=EL#CRP=NONE#LAC=NONE
4	C156	ER release	1575547147.00	ER release	ER_REL
5	E9383	ER registration	1575707652.00	ER registration	ER_REG
6	E9383	ER triage	1575712560.00	ER triage	ER_TR
7	E9383	Biomarker assessment	1575712561.00	Biomarker assessment	BIO_A#WBC=EL#CRP=EL#LAC=EL
8	C167	ER registration	1575715511.00	ER registration	ER_REG
9	C167	ER triage	1575716460.00	ER triage	ER_TR
10	C167	Biomarker assessment	1575716461.00	Biomarker assessment	BIO_A#WBC=EL#CRP=EL#LAC=NONE
11	E9383	Admission to IC	1575718546.00	Admission to IC	IC
12	C167	Admission to NC	1575720709.00	Admission to NC	NC
13	C167	Hospital release	1576072931.00	Hospital release	H_REL
14	E9383	Hospital release	1576399931.00	Hospital release	H_REL

Table 6: The consolidated data stream slice

2.3   Process discovery

After we created pre-processing and pre-classification classes, we can use them in the Sequence Detector.

```
> seq_detector <- HybridSequenceClassifier(c("id","class","time","out"),"time","time",
+                                          "id",preclassifier=pc,preprocessor=pp,
+                                          pattern_field="out")
> seq_detector$process(input_streams)
```

Any subsequent input data streams slice are processed by re-invoking *process* method over again. ETT created from the example is:

```
> seq_detector$printMachines()
```

```
-==* ETT wrapper machines list(1) *==-
Machine:c48e49e05fd1d72e9882
=========
  Common cache:[C156,C167,E9383]
  State:BIO_A#WBC=EL#CRP=EL#LAC=EL Type:Normal Entry:0 Final:0 Population:1
    Cache:[E9383]
    Keys:[]
    Patterns:[Biomarker assessment]
      Transition(413320b25ee13d02968a) to:IC Symbols:[IC] Input state: Output state: Population:1
          Patterns:[Admission to IC]
  State:BIO_A#WBC=EL#CRP=EL#LAC=NONE Type:Normal Entry:0 Final:0 Population:1
    Cache:[C167]
    Keys:[]
    Patterns:[Biomarker assessment]
      Transition(6138be9c0cf72a18d957) to:NC Symbols:[NC] Input state: Output state: Population:1
          Patterns:[Admission to NC]
  State:BIO_A#WBC=EL#CRP=NONE#LAC=NONE Type:Normal Entry:0 Final:0 Population:1
    Cache:[C156]
    Keys:[]
    Patterns:[Biomarker assessment]
      Transition(280022bddc0577306847) to:ER_REL Symbols:[ER_REL] Input state: Output state: Population:1
          Patterns:[ER release]
  State:ER_REG Type:Normal Entry:1 Final:0 Population:3
    Cache:[C156,C167,E9383]
    Keys:[]
    Patterns:[ER registration]
      Transition(c6ccbd049fec98780ce5) to:ER_TR Symbols:[ER_TR] Input state: Output state: Population:3
          Patterns:[ER triage]
      ENTRY Transition(31b1ee6786c709984931) Symbols:[ER_REG] Population:3
          Patterns:[ER registration]
  State:ER_REL Type:Normal Entry:0 Final:0 Population:1
    Cache:[C156]
    Keys:[C156]
    Patterns:[ER release]
  State:ER_TR Type:Normal Entry:0 Final:0 Population:3
    Cache:[C156,C167,E9383]
    Keys:[]
    Patterns:[ER triage]
```

```
        Transition(4dc071b51c286d179872) to:BIO_A#WBC=EL#CRP=NONE#LAC=NONE Symbols:[BIO_A#WBC=EL#CRP=NONE#
            LAC=NONE] Input state: Output state: Population:1
          Patterns:[Biomarker assessment]
        Transition(694d13d7990bd78e344e) to:BIO_A#WBC=EL#CRP=EL#LAC=NONE Symbols:[BIO_A#WBC=EL#CRP=EL#LAC=
            NONE] Input state: Output state: Population:1
          Patterns:[Biomarker assessment]
        Transition(bb2e1413f70402332ae9) to:BIO_A#WBC=EL#CRP=EL#LAC=EL Symbols:[BIO_A#WBC=EL#CRP=EL#LAC=EL]
             Input state: Output state: Population:1
          Patterns:[Biomarker assessment]
  State:H_REL Type:Normal Entry:0 Final:0 Population:2
    Cache:[C167,E9383]
    Keys:[C167,E9383]
    Patterns:[Hospital release]
  State:IC Type:Normal Entry:0 Final:0 Population:1
    Cache:[E9383]
    Keys:[]
    Patterns:[Admission to IC]
      Transition(017d57cb1890bb347e3a) to:H_REL Symbols:[H_REL] Input state: Output state: Population:1
          Patterns:[Hospital release]
  State:NC Type:Normal Entry:0 Final:0 Population:1
    Cache:[C167]
    Keys:[]
    Patterns:[Admission to NC]
      Transition(abcc81e69287d5070bbf) to:H_REL Symbols:[H_REL] Input state: Output state: Population:1
          Patterns:[Hospital release]
=========
-==*****************************************==-
```

And the ETT structure (Γ related) is:

```
> seq_detector$plotMachines()
```
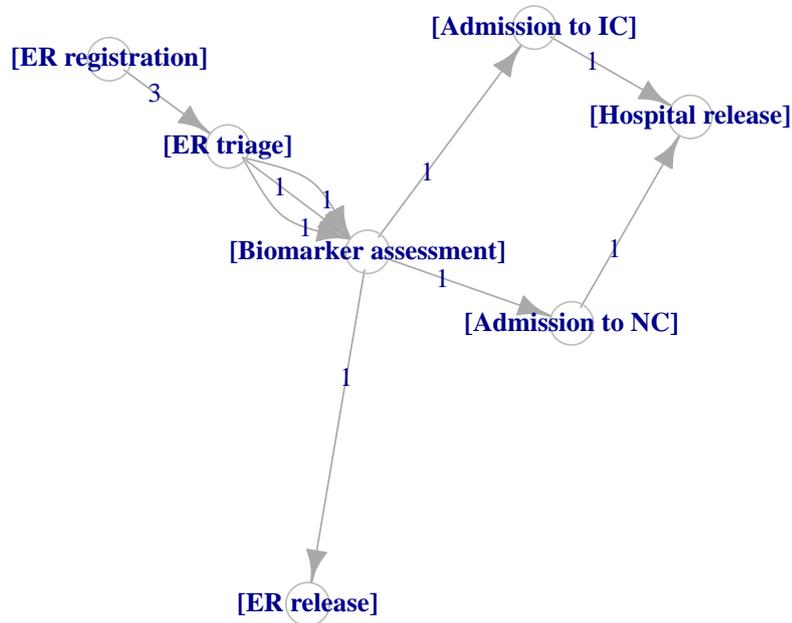
3 Detecting pre-learned sequences

3.1 Pre-learning

There is a specific need to detect specific sequences in process instances or in time-series datasets. Once Sequence Detector detects the sequence, an alerting notification can be generated from the results. Let us define a sequence that we want to learn.

Learning sequence 3.1.

```
> st <- data.frame(sequence=c("A","E","G"),alert=c(NA,NA,"Alert 1"))

> st
```

	sequence	alert
1	A	
2	E	
3	G	Alert 1

Table 7: Learning sequence 3.1

```
> pp <- HSC_PP(c("sequence","alert"),"sequence_id",create_unique_key=TRUE,auto_id=TRUE)
> pc <- HSC_PC_Attribute("sequence")
> seq_detector <- HybridSequenceClassifier(c("sequence_id","sequence","alert"),
+                                           "sequence_id","sequence_id",
+                                           preclassifier=pc,preprocessor=pp,
+                                           pattern_field="alert",reuse_states=FALSE)
> input_streams <- list(stream=st)
> seq_detector$process(input_streams,learn=TRUE)
> seq_detector$cleanKeys()
```

We used predefined pre-processor *HSC_PP*, which filters out supplied fields and orders the output event stream according to one of the filtered streams. A predefined pre-classifier *HSC_PC_Attribute* is used, which uses an attribute value as the input symbol into ETT. ETT created from the example is:

```
> seq_detector$printMachines(print_cache=FALSE,print_keys=FALSE)
```

```
-==* ETT wrapper machines list(1) *==-
Machine:c7760156283cf462f77b
=========
   State:A Type:Normal Entry:1 Final:0 Population:1
      Patterns:[]
         Transition(1c60d8c7c2a30706f9a3) to:E Symbols:[E] Input state: Output state: Population:1
            Patterns:[]
         ENTRY Transition(ea61101b3f0f714c519e) Symbols:[A] Population:1
            Patterns:[]
   State:E Type:Normal Entry:0 Final:0 Population:1
      Patterns:[]
         Transition(85f042515eeb7577ac30) to:G Symbols:[G] Input state: Output state: Population:1
            Patterns:[Alert 1]
   State:G Type:Normal Entry:0 Final:0 Population:1
      Patterns:[Alert 1]
=========
-==****************************************==-
```

3.2 Simple testing streams

Testing sequence 3.1.

```
> seq_test1 <- c("E","I","A","G","E","K","F","E","A","G","G","B","W","L")
> res_test1 <- seq_detector$process(list(stream=data.frame(sequence=seq_test1,
+                                                          alert=NA)),learn=FALSE)
> out_test1 <- data.frame()
```

```
> for(i in 1:nrow(res_test1$stream))
+   out_test1 <- rbind(out_test1,data.frame(sequence=res_test1$stream[i,"sequence"],
+                             alert=c_to_string(res_test1$explanation[[i]]$actual)))

> out_test1
```

	sequence	alert
1	E	[]
2	I	[]
3	A	[]
4	G	[]
5	E	[]
6	K	[]
7	F	[]
8	E	[]
9	A	[]
10	G	[Alert 1]
11	G	[]
12	B	[]
13	W	[]
14	L	[]

Table 8: Results for testing sequence 3.1, $\lambda_n = \infty$

The result in Table 8 is a bit unusual and from the user point of view unexpected. However, if looked carefully, the learned sequence *AEG* actually is there until the *Alert 1* appears. The reason for that is the current level of ETT noise tolerance, set to $\lambda_n = \infty$. If we set the noise tolerance to $\lambda_n = 0$.

```
> pp <- HSC_PP(c("sequence","alert"),"sequence_id",create_unique_key=TRUE,auto_id=TRUE)
> pc <- HSC_PC_Attribute("sequence")
> dd <- list(type="count",count=0,context_related=TRUE)
> seq_detector <- HybridSequenceClassifier(c("sequence_id","sequence","alert"),
+                                          "sequence_id","sequence_id",
+                                          preclassifier=pc,preprocessor=pp,
+                                          pattern_field="alert",reuse_states=FALSE,
+                                          decay_descriptors=list(d1=dd))
> seq_detector$process(input_streams,learn=TRUE)
> seq_detector$cleanKeys()
```

we are getting a bit different result for the same testing sequence, as seen in Table 9

```
> res_test1 <- seq_detector$process(list(stream=data.frame(sequence=seq_test1,alert=NA)),
+                                   learn=FALSE)
> out_test1 <- data.frame()
> for(i in 1:nrow(res_test1$stream))
+   out_test1 <- rbind(out_test1,data.frame(sequence=res_test1$stream[i,"sequence"],
+                             alert=c_to_string(res_test1$explanation[[i]]$actual)))

> out_test1
```

We define a bit diffent testing sequence for noise tolerance $\lambda_n = 0$. Part of the sequence is learned sequence. Later in the following testing sequence we have a sub-sequence *AEBG*, which is not recognized due to the noise tolerance setting, as seen in Table 10.

Testing sequence 3.2.

```
> seq_test2 <- c("E","I","A","E","G","E","K","F","E","A","E","B","G","W","L")
> res_test2 <- seq_detector$process(list(stream=data.frame(sequence=seq_test2,alert=NA)),
+                                   learn=FALSE)
> out_test2 <- data.frame()
> for(i in 1:nrow(res_test2$stream))
+   out_test2 <- rbind(out_test2,data.frame(sequence=res_test2$stream[i,"sequence"],
+                             alert=c_to_string(res_test2$explanation[[i]]$actual)))
```

11

	sequence	alert
1	E	[]
2	I	[]
3	A	[]
4	G	[]
5	E	[]
6	K	[]
7	F	[]
8	E	[]
9	A	[]
10	G	[]
11	G	[]
12	B	[]
13	W	[]
14	L	[]

Table 9: Result for testing sequence 3.1, $\lambda_n = 0$

```
> out_test2
```

	sequence	alert
1	E	[]
2	I	[]
3	A	[]
4	E	[]
5	G	[Alert 1]
6	E	[]
7	K	[]
8	F	[]
9	E	[]
10	A	[]
11	E	[]
12	B	[]
13	G	[]
14	W	[]
15	L	[]

Table 10: Result for testing sequence 3.2, $\lambda_n = 0$

However, if we set the noise tolerance to $\lambda_n = 1$, the sub-sequence $AEBG$ is recognized, as seen in Table 11.

```
> dd <- list(type="count",count=1,context_related=TRUE)
> seq_detector <- HybridSequenceClassifier(c("sequence_id","sequence","alert"),
+                                           "sequence_id","sequence_id",
+                                           preclassifier=pc,preprocessor=pp,
+                                           pattern_field="alert",reuse_states=FALSE,
+                                           decay_descriptors=list(d1=dd))
> seq_detector$process(input_streams,learn=TRUE) # learn
> seq_detector$cleanKeys() # clean all context keys
> res_test2 <- seq_detector$process(list(stream=data.frame(sequence=seq_test2,alert=NA)),
+                                    learn=FALSE)
> out_test2 <- data.frame()
> for(i in 1:nrow(res_test2$stream))
+   out_test2 <- rbind(out_test2,data.frame(sequence=res_test2$stream[i,"sequence"],
+                                    alert=c_to_string(res_test2$explanation[[i]]$actual)))

> out_test2
```

	sequence	alert
1	E	[]
2	I	[]
3	A	[]
4	E	[]
5	G	[Alert 1]
6	E	[]
7	K	[]
8	F	[]
9	E	[]
10	A	[]
11	E	[]
12	B	[]
13	G	[Alert 1]
14	W	[]
15	L	[]

Table 11: Result for testing sequence 3.2, $\lambda_n = 1$

4 Multi-contextual detection

4.1 Multi-contextual learning sequences

When a several sequences are intertwined in the same consolidated event stream $\mathcal{D}_c$, each sequence can be observed as a separated whole identified by a set of data, or sequence context identifiers. A context identifier could be a customer, product, partner, or just an ongoing process identifier. The following learning sequence represents two products and their sales numbers. Let us define a sequence that we want to learn.

Learning sequence 4.1.

```
> st <- data.frame(product=c("P45","P134","P45","P134","P134","P45","P134"),
+                  sales=c(2,12,18,16,18,24,8),
+                  alert=c(NA,NA,NA,NA,NA,"Alert P45","Alert P134"))
> input_streams <- list(stream=st)

> st
```

	product	sales	alert
1	P45	2.00	
2	P134	12.00	
3	P45	18.00	
4	P134	16.00	
5	P134	18.00	
6	P45	24.00	Alert P45
7	P134	8.00	Alert P134

Table 12: A multi-contextual learning sequence 4.1

Now we define a Sequence Detector having noise tolerance $\lambda_n = 0$

```
> pp <- HSC_PP(c("product","sales","alert"),"sequence_id",auto_id=TRUE)
> pc <- HSC_PC_Attribute("sales")
> dd <- list(type="count",count=0,context_related=TRUE)
> seq_detector <- HybridSequenceClassifier(c("sequence_id","product","sales","alert"),"sequence_id",
+                                          "sequence_id",context_field="product",preclassifier=pc,
+                                          preprocessor=pp,pattern_field="alert",reuse_states=FALSE,
+                                          decay_descriptors=list(d1=dd))
> seq_detector$process(input_streams,learn=TRUE)
> seq_detector$cleanKeys()

> seq_detector$printMachines(print_cache=FALSE,print_keys=FALSE)
```

```
-==* ETT wrapper machines list(2) *==-
Machine:4b80d0856b8a9df1acdd
=========
   State:12 Type:Normal Entry:1 Final:0 Population:1
      Patterns:[]
        Transition(690c52643bc66cc87f3e) to:16 Symbols:[16] Input state: Output state: Population:1
            Patterns:[]
        ENTRY Transition(10c1f460ab5e6e38f5b6) Symbols:[12] Population:1
            Patterns:[]
   State:16 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[]
        Transition(6a674f00f27cbf0a13ad) to:18 Symbols:[18] Input state: Output state: Population:1
            Patterns:[]
   State:18 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[]
        Transition(d8978edcbac20777be29) to:8 Symbols:[8] Input state: Output state: Population:1
            Patterns:[Alert P134]
   State:8 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[Alert P134]
=========
Machine:f41e0b6e0301c7a38680
=========
   State:18 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[]
        Transition(03331f9fc0c49e2e5183) to:24 Symbols:[24] Input state: Output state: Population:1
            Patterns:[Alert P45]
   State:2 Type:Normal Entry:1 Final:0 Population:1
      Patterns:[]
        Transition(73288845aed0d34d0d95) to:18 Symbols:[18] Input state: Output state: Population:1
            Patterns:[]
        ENTRY Transition(43b721390c7ead33606d) Symbols:[2] Population:1
            Patterns:[]
   State:24 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[Alert P45]
=========
-==*****************************************==-
```

Since we did not use option of reusing states (reuse_states=FALSE, see (Krleža et al., 2019) for details), two disctinct ETTs were created, each for its own context. If there is any merging point in these two ETTs, such as the state *18*, we can use merging option two create one consolidated ETT. We need to be careful with merging option, as it will only perform merging for ETT_1 and ETT_2 when the following conditions are met:

$$\exists s_1, s_2(s_1 \in Q(ETT_1) \wedge s_2 \in Q(ETT_2) \wedge \exists q((*, q, s_1) \in R_\delta(ETT_1) \wedge (*, q, s_1) \in R_\delta(ETT_2)) \wedge$$
$$\omega_{s_1} = \{(s_1, \gamma_1) | \gamma_1 \in \Gamma(ETT_1)\} \subseteq \omega(ETT_1) \wedge \omega_{s_2} = \{(s_2, \gamma_2) | \gamma_2 \in \Gamma(ETT_2)\} \subseteq \omega(ETT_2) \wedge \quad (6)$$
$$(\omega_{s_1} \cap \omega_{s_2} \neq \emptyset \vee (\omega_{s_1} = \emptyset \wedge \omega_{s_2} = \emptyset)))$$

We need to have a pair of states, each in its own ETT, that share the same input and output symbols, or have no output symbols at all. Since we did not define output symbols for the state *18*, merging should be successful.

```
> seq_detector$mergeMachines()

> seq_detector$printMachines(print_cache=FALSE,print_keys=FALSE)


-==* ETT wrapper machines list(1) *==-
Machine:4b80d0856b8a9df1acdd
=========
   State:12 Type:Normal Entry:1 Final:0 Population:1
      Patterns:[]
        Transition(690c52643bc66cc87f3e) to:16 Symbols:[16] Input state: Output state: Population:1
            Patterns:[]
        ENTRY Transition(10c1f460ab5e6e38f5b6) Symbols:[12] Population:1
            Patterns:[]
   State:16 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[]
        Transition(6a674f00f27cbf0a13ad) to:18 Symbols:[18] Input state: Output state: Population:1
            Patterns:[]
   State:18 Type:Normal Entry:0 Final:0 Population:2
      Patterns:[]
```

```
            Transition(29b7973c593bce7b1e8d) to:24 Symbols:[24] Input state: Output state: Population:1
                Patterns:[Alert P45]
            Transition(d8978edcbac20777be29) to:8 Symbols:[8] Input state: Output state: Population:1
                Patterns:[Alert P134]
      State:2 Type:Normal Entry:1 Final:0 Population:1
          Patterns:[]
            Transition(d4d76e931ffce2f5e53b) to:18 Symbols:[18] Input state: Output state: Population:1
                Patterns:[]
            ENTRY Transition(779cc170080bc0662e2e) Symbols:[2] Population:1
                Patterns:[]
      State:24 Type:Normal Entry:0 Final:0 Population:1
          Patterns:[Alert P45]
      State:8 Type:Normal Entry:0 Final:0 Population:1
          Patterns:[Alert P134]
=========
-==****************************************==-
```

4.2   Multi-contextual testing sequences

We define the following testing sequence.

Testing sequence 4.1.
```
> tt <- data.frame(product=c("P672","P113","P983","P23872","P5","P672","P2982","P983","P672",
+                            "P991","P983","P113","P2982","P344"),
+                   sales=c(2,11,12,98,8,18,298,16,24,25,18,16,43,101),alert=NA)
> test_streams <- list(stream=tt)

> tt
```

	product	sales	alert
1	P672	2.00	
2	P113	11.00	
3	P983	12.00	
4	P23872	98.00	
5	P5	8.00	
6	P672	18.00	
7	P2982	298.00	
8	P983	16.00	
9	P672	24.00	
10	P991	25.00	
11	P983	18.00	
12	P113	16.00	
13	P2982	43.00	
14	P344	101.00	

Table 13: A multi-contextual testing sequence 4.1

```
> res_test1 <- seq_detector$process(test_streams,learn=FALSE)
> out_test1 <- data.frame()
> for(i in 1:nrow(res_test1$stream))
+   out_test1 <- rbind(out_test1,data.frame(product=res_test1$stream[i,"product"],
+                                           sales=res_test1$stream[i,"sales"],
+                                           alert=c_to_string(res_test1$explanation[[i]]$actual)))

> out_test1
```

The result can be seen in Table 14. The final state of ETT is

```
> seq_detector$printMachines()


-==* ETT wrapper machines list(1) *==-
Machine:4b80d0856b8a9df1acdd
=========
    Common cache:[P134,P45,P672,P983]
    State:12 Type:Normal Entry:1 Final:0 Population:2
```

	product	sales	alert
1	P672	2.00	[]
2	P113	11.00	[]
3	P983	12.00	[]
4	P23872	98.00	[]
5	P5	8.00	[]
6	P672	18.00	[]
7	P2982	298.00	[]
8	P983	16.00	[]
9	P672	24.00	[Alert P45]
10	P991	25.00	[]
11	P983	18.00	[]
12	P113	16.00	[]
13	P2982	43.00	[]
14	P344	101.00	[]

Table 14: Results for the testing sequence 4.1

```
    Cache:[P134,P983]
    Keys:[]
    Patterns:[]
      Transition(690c52643bc66cc87f3e) to:16 Symbols:[16] Input state: Output state: Population:2
          Patterns:[]
      ENTRY Transition(10c1f460ab5e6e38f5b6) Symbols:[12] Population:2
          Patterns:[]
  State:16 Type:Normal Entry:0 Final:0 Population:2
    Cache:[P134,P983]
    Keys:[]
    Patterns:[]
      Transition(6a674f00f27cbf0a13ad) to:18 Symbols:[18] Input state: Output state: Population:2
          Patterns:[]
  State:18 Type:Normal Entry:0 Final:0 Population:4
    Cache:[P134,P45,P672,P983]
    Keys:[P983]
    Patterns:[]
      Transition(29b7973c593bce7b1e8d) to:24 Symbols:[24] Input state: Output state: Population:2
          Patterns:[Alert P45]
      Transition(d8978edcbac20777be29) to:8 Symbols:[8] Input state: Output state: Population:1
          Patterns:[Alert P134]
  State:2 Type:Normal Entry:1 Final:0 Population:2
    Cache:[P45,P672]
    Keys:[]
    Patterns:[]
      Transition(d4d76e931ffce2f5e53b) to:18 Symbols:[18] Input state: Output state: Population:2
          Patterns:[]
      ENTRY Transition(779cc170080bc0662e2e) Symbols:[2] Population:2
          Patterns:[]
  State:24 Type:Normal Entry:0 Final:0 Population:2
    Cache:[P45,P672]
    Keys:[P672]
    Patterns:[Alert P45]
  State:8 Type:Normal Entry:0 Final:0 Population:1
    Cache:[P134]
    Keys:[]
    Patterns:[Alert P134]
=========
-==*****************************************==-
```

If we add another slice of the testing stream 4.1

```
> tt <- data.frame(product=c("P115","P45","P22","P983","P9","P19","P73"),
+                  sales=c(91,43,52,8,1,105,35),alert=NA)
> test_streams <- list(stream=tt)

> tt
```

	product	sales	alert
1	P115	91.00	
2	P45	43.00	
3	P22	52.00	
4	P983	8.00	
5	P9	1.00	
6	P19	105.00	
7	P73	35.00	

Table 15: Additional slice of the testing sequence 4.1

```
> res_test1 <- seq_detector$process(test_streams,learn=FALSE)
> out_test1 <- data.frame()
> for(i in 1:nrow(res_test1$stream))
+   out_test1 <- rbind(out_test1,data.frame(product=res_test1$stream[i,"product"],
+                                           sales=res_test1$stream[i,"sales"],
+                                           alert=c_to_string(res_test1$explanation[[i]]$actual)))

> out_test1
```

We get additional sequence recognition alert as in Table 16.

	product	sales	alert
1	P115	91.00	[]
2	P45	43.00	[]
3	P22	52.00	[]
4	P983	8.00	[Alert P134]
5	P9	1.00	[]
6	P19	105.00	[]
7	P73	35.00	[]

Table 16: Results for the additional slice of the testing sequence 4.1

5 Token decay mechanism

5.1 Overview

Token decay mechanism is introduced for the following reasons:

- ETT noise tolerance. Decay mechanism enabled ETT to tolerate extra symbols in the previously learned sequences, both within a context and globally.

- Has table cleaning. As the has table that keeps tokens gets bigger, ETT processing gets slower. Eventually, processing might fall behind the processed data streams velocity. We need to purge irrelevant tokens to keep ETT high performing and taking less memory as possible.

Token decay has three basic options:

1. **Context-related counting decay** - Works within one context. User can define how much extra symbols placed within context is acceptable.

2. **Global counting decay** - Works globally. User can define how much extra symbols is allowable between two token pushes. If a token is not pushed within the defined number of input symbols it decays and gets removed.

3. **Time decay** - Users can define the time needed for a token to decay. If the token does not get pushed within the designated time it decays and gets removed.

Decay mechanism is controlled through passing decay descriptors at the time of Sequence Detector creation, and cannot be changed later on, i.e., a new Sequence Detector needs to be created to use different decay descriptors.

Context-related count decay

```
> dd1 <- list(type="count",count=1,context_related=TRUE)
```

($\lambda_n = 1$) 1 symbol is allowed within the same context. For example, the learning sequence ABC will be successfully recognized within the testing sequence $ABFC$, but not in the testing sequence $ABFTC$.

Global count decay

```
> dd2 <- list(type="count",count=50,context_related=FALSE)
```

If a token is not pushed 50 symbols from its last push, it will decay and get removed.

Time decay

```
> dd3 <- list(type="time",days=0,hours=1,minutes=10,context_related=FALSE)
```

If a token is not pushed within 1 hour and 10 minutes from its last push, it will decay and get removed.

5.2   Example 1: Context-related count decay

Let us define a sequence that we want to learn.

Learning sequence 5.1.

```
> st <- data.frame(product=c("P45","P45"),sales=c(5,10),alert=c(NA,"Alert 1"))
> input_streams <- list(stream=st)

> st
```

	product	sales	alert
1	P45	5.00	
2	P45	10.00	Alert 1

Table 17: Learning sequence 5.1

Now we define a Sequence Detector having context-related count decay $\lambda_n = 1$

```
> pp <- HSC_PP(c("product","sales","alert"),"sequence_id",auto_id=TRUE)
> pc <- HSC_PC_Attribute("sales")
> dd <- list(type="count",count=1,context_related=TRUE)
> seq_detector <- HybridSequenceClassifier(c("sequence_id","product","sales","alert"),"sequence_id",
+                                      "sequence_id",context_field="product",preclassifier=pc,
+                                      preprocessor=pp,pattern_field="alert",reuse_states=FALSE,
+                                      decay_descriptors=list(d1=dd))
> seq_detector$process(input_streams,learn=TRUE)
> seq_detector$cleanKeys()

> seq_detector$printMachines(print_cache=FALSE,print_keys=FALSE)


-==* ETT wrapper machines list(1) *==-
Machine:e6e62fd50384196d0eb3
=========
  State:10 Type:Normal Entry:0 Final:0 Population:1
     Patterns:[Alert 1]
  State:5 Type:Normal Entry:1 Final:0 Population:1
     Patterns:[]
       Transition(223a745a61d5f1f4ca85) to:10 Symbols:[10] Input state: Output state: Population:1
           Patterns:[Alert 1]
       ENTRY Transition(989e0f9933a80e484a37) Symbols:[5] Population:1
           Patterns:[]
=========
-==****************************************==-
```

For the testing we define both correct and incorrect contexts.

Testing sequence 5.1.

```
> tt <- data.frame(product=c("P113","P29","P113","P29","P29","P113","P29"),
+                  sales=c(5,5,7,8,9,10,10),alert=NA)
> test_streams <- list(stream=tt)

> tt
```

	product	sales	alert
1	P113	5.00	
2	P29	5.00	
3	P113	7.00	
4	P29	8.00	
5	P29	9.00	
6	P113	10.00	
7	P29	10.00	

Table 18: Testing sequence 5.1

The testing result is:

```
> res_test1 <- seq_detector$process(test_streams,learn=FALSE)
> out_test1 <- data.frame()
> for(i in 1:nrow(res_test1$stream))
+   out_test1 <- rbind(out_test1,data.frame(product=res_test1$stream[i,"product"],
+                                            sales=res_test1$stream[i,"sales"],
+                                            alert=c_to_string(res_test1$explanation[[i]]$actual)))

> out_test1
```

	product	sales	alert
1	P113	5.00	[]
2	P29	5.00	[]
3	P113	7.00	[]
4	P29	8.00	[]
5	P29	9.00	[]
6	P113	10.00	[Alert 1]
7	P29	10.00	[]

Table 19: Testing sequence 5.1 results

5.3 Example 2: Global count decay

We use the same learning sequence 5.1. However, we define a Sequence Detector having a global count decay.

```
> dd <- list(type="count",count=5,context_related=FALSE)
> seq_detector <- HybridSequenceClassifier(c("sequence_id","product","sales","alert"),"sequence_id",
+                                           "sequence_id",context_field="product",preclassifier=pc,
+                                           preprocessor=pp,pattern_field="alert",reuse_states=FALSE,
+                                           decay_descriptors=list(d1=dd))
> seq_detector$process(input_streams,learn=TRUE)
> seq_detector$cleanKeys()
```

For the testing we define both correct and incorrect contexts.

Testing sequence 5.2.

```
> tt <- data.frame(product=c("P29","P113","P114","P115","P113","P114","P115","P29"),
+                  sales=c(5,5,5,5,10,7,10,10),alert=NA)
> test_streams <- list(stream=tt)

> tt
```

	product	sales	alert
1	P29	5.00	
2	P113	5.00	
3	P114	5.00	
4	P115	5.00	
5	P113	10.00	
6	P114	7.00	
7	P115	10.00	
8	P29	10.00	

Table 20: Testing sequence 5.2

The testing result is:

```
> res_test2 <- seq_detector$process(test_streams,learn=FALSE)
> out_test2 <- data.frame()
> for(i in 1:nrow(res_test2$stream))
+   out_test2 <- rbind(out_test2,data.frame(product=res_test2$stream[i,"product"],
+                                    sales=res_test2$stream[i,"sales"],
+                                    alert=c_to_string(res_test2$explanation[[i]]$actual)))

> out_test2
```

	product	sales	alert
1	P29	5.00	[]
2	P113	5.00	[]
3	P114	5.00	[]
4	P115	5.00	[]
5	P113	10.00	[Alert 1]
6	P114	7.00	[]
7	P115	10.00	[Alert 1]
8	P29	10.00	[]

Table 21: Testing sequence 5.2 results

5.4   Example 3: Time decay

We use the same learning sequence 5.1. However, this time we define a timestamp field for determining an event timing.

```
> st <- data.frame(product=c("P21","P21"),timestamp=c("01.12.2019. 10:00:00","01.12.2019. 10:01:00"),
+               sales=c(5,10),alert=c(NA,"Alert 1"))

> st
```

	product	timestamp	sales	alert
1	P21	01.12.2019. 10:00:00	5.00	
2	P21	01.12.2019. 10:01:00	10.00	Alert 1

Table 22: Learning sequence 5.1 with timestamp

Finish it by transforming timestamp fields into the POSIXct type.

```
> st <- transform.data.frame(st,timestamp=as.POSIXct(st$timestamp,
+                                             format="%d.%m.%Y. %H:%M:%S"))
> input_streams <- list(stream=st)
```

We define a Sequence Detector having a time decay set to 1 hour.

```
> pp <- HSC_PP(c("product","sales","alert","timestamp"),"timestamp")
> pc <- HSC_PC_Attribute("sales")
> dd <- list(type="time",days=0,hours=1,minutes=0,context_related=FALSE)
> seq_detector <- HybridSequenceClassifier(c("timestamp","product","sales","alert"),
```

```
+                                       "timestamp","timestamp",context_field="product",
+                                       preclassifier=pc,preprocessor=pp,pattern_field="alert",
+                                       reuse_states=FALSE,decay_descriptors=list(d1=dd))
> seq_detector$process(input_streams,learn=TRUE)
> seq_detector$cleanKeys()
```

We define a new testing data stream.

Testing sequence 5.3.

```
> tt <- data.frame(product=c("P12","P13","P14","P15","P13","P14","P15","P12"),
+                  sales=c(5,5,5,5,10,10,10,10),
+                  timestamp=c("05.12.2019. 10:30:20","05.12.2019. 10:31:20",
+                              "05.12.2019. 10:32:20","05.12.2019. 10:33:20",
+                              "05.12.2019. 10:34:20","05.12.2019. 10:35:20",
+                              "05.12.2019. 10:40:20","05.12.2019. 12:30:20"),alert=NA)

> tt
```

	product	sales	timestamp	alert
1	P12	5.00	05.12.2019. 10:30:20	
2	P13	5.00	05.12.2019. 10:31:20	
3	P14	5.00	05.12.2019. 10:32:20	
4	P15	5.00	05.12.2019. 10:33:20	
5	P13	10.00	05.12.2019. 10:34:20	
6	P14	10.00	05.12.2019. 10:35:20	
7	P15	10.00	05.12.2019. 10:40:20	
8	P12	10.00	05.12.2019. 12:30:20	

Table 23: Testing sequence 5.3

We finish by transforming timestamp fields into the POSIXct type.

```
> tt <- transform.data.frame(tt,timestamp=as.POSIXct(tt$timestamp,
+                                        format="%d.%m.%Y. %H:%M:%S"))
> test_streams <- list(stream=tt)
```

The testing result is:

```
> res_test3 <- seq_detector$process(test_streams,learn=FALSE)
> out_test3 <- data.frame()
> for(i in 1:nrow(res_test3$stream))
+   out_test3 <- rbind(out_test3,data.frame(product=res_test3$stream[i,"product"],
+                                sales=res_test3$stream[i,"sales"],
+                                timestamp=as.character(res_test3$stream[i,"timestamp"],
+                                               format="%d.%m.%Y. %H:%M:%S"),
+                                alert=c_to_string(res_test3$explanation[[i]]$actual)))

> out_test3
```

	product	sales	timestamp	alert
1	P12	5.00	2019-12-05 10:30:20	[]
2	P13	5.00	2019-12-05 10:31:20	[]
3	P14	5.00	2019-12-05 10:32:20	[]
4	P15	5.00	2019-12-05 10:33:20	[]
5	P13	10.00	2019-12-05 10:34:20	[Alert 1]
6	P14	10.00	2019-12-05 10:35:20	[Alert 1]
7	P15	10.00	2019-12-05 10:40:20	[Alert 1]
8	P12	10.00	2019-12-05 12:30:20	[]

Table 24: Testing sequence 5.3 results

6 Statistical projections

6.1 ETT statistic collection and projection

While working, ETT collects statistic about token pushes. At the moment the collected statistic comprises:

- The number of pushes through an ETT transition

- The number of inserted tokens in an ETT state

These counting statistics $\mathcal{S}_\delta$ can be used to create ETT projections that indicate regularity of a sequence as described in (Krleža et al., 2019). Projection can reveal regular, irregular and anomalous sequences. Using threshold $\Phi_{\mathcal{S}_\delta}$, we can sub-select the ETT structure to

$$
\begin{aligned}
\mathcal{R}_{\delta_{\mathcal{S}+}} &\subseteq \{r_i | r_i \in \mathcal{R}_\delta \wedge \mathcal{S}_\delta(r_i) \geq \Phi_{\mathcal{S}_\delta}\} \\
\mathcal{R}_{\delta_{\mathcal{S}-}} &\subseteq \{r_i | r_i \in \mathcal{R}_\delta \wedge \mathcal{S}_\delta(r_i) < \Phi_{\mathcal{S}_\delta}\}
\end{aligned}
\tag{7}
$$

resulting in

$$
\begin{aligned}
ETT_{\mathcal{S}+} &\subseteq ETT[\mathcal{R}_\delta = \mathcal{R}_{\delta_{\mathcal{S}+}}] \\
ETT_{\mathcal{S}-} &\subseteq ETT[\mathcal{R}_\delta = \mathcal{R}_{\delta_{\mathcal{S}-}}]
\end{aligned}
\tag{8}
$$

where we can isolate the following cases:

- **Regular data sequences**. Data sequences that can be obtained only by traversing through $ETT_{\mathcal{S}+}$.

- **Anomalous data sequences**. Data sequences that can be obtained only by traversing through $ETT_{\mathcal{S}-}$.

- **Irregular data sequences**. Data sequences that can be obtained by traversing through both $ETT_{\mathcal{S}-}$ and $ETT_{\mathcal{S}+}$. These data sequences are made of regular and irregular subsequences.

6.2 Example 1

Let us define sequences that we want to learn. These sequences must have regular and irregular transitions.

Learning sequence 6.1.

```
> st <- data.frame(product=c("P1","P2"),sales=c(5,76),alert=c(NA,NA))
> for(i in 1:400) {
+   st <- rbind(st,data.frame(product=c("P1","P2"),sales=c(10,58),alert=c(NA,NA)))
+   st <- rbind(st,data.frame(product=c("P1","P2"),sales=c(20,31),alert=c(NA,NA)))
+ }
> st <- rbind(st,data.frame(product=c("P1","P2"),sales=c(30,11),
+                           alert=c("Sequence 1","Sequence 2")))
> input_streams <- list(stream=st)
```

Now we define a Sequence Detector for learning the simple sequences we constructed earlier.

```
> pp <- HSC_PP(c("product","sales","alert"),"sequence_id",auto_id=TRUE)
> pc <- HSC_PC_Attribute("sales")
> seq_detector <- HybridSequenceClassifier(c("sequence_id","product","sales","alert"),"sequence_id",
+                                     "sequence_id",context_field="product",preclassifier=pc,
+                                     preprocessor=pp,reuse_states=TRUE,pattern_field="alert")
> seq_detector$process(input_streams,learn=TRUE)
> seq_detector$cleanKeys()

> seq_detector$printMachines(print_cache=FALSE,print_keys=FALSE)


-==* ETT wrapper machines list(2) *==-
Machine:964ab952a1a6db9635cf
=========
  State:11 Type:Normal Entry:0 Final:0 Population:1
    Patterns:[Sequence 2]
```

```
   State:31 Type:Normal Entry:0 Final:0 Population:400
      Patterns:[]
        Transition(c05f9fefb8f43436a4db) to:11 Symbols:[11] Input state: Output state: Population:1
            Patterns:[Sequence 2]
        Transition(ed3850f0d20333c0faad) to:58 Symbols:[58] Input state: Output state: Population:399
            Patterns:[]
   State:58 Type:Normal Entry:0 Final:0 Population:400
      Patterns:[]
        Transition(880f6e3f259ae9640c0c) to:31 Symbols:[31] Input state: Output state: Population:400
            Patterns:[]
   State:76 Type:Normal Entry:1 Final:0 Population:1
      Patterns:[]
        Transition(82e0cc6216320036653b) to:58 Symbols:[58] Input state: Output state: Population:1
            Patterns:[]
        ENTRY Transition(f3e7d129dc008cc5cd33) Symbols:[76] Population:1
            Patterns:[]
=========
Machine:a31ef78d0468ec069227
=========
   State:10 Type:Normal Entry:0 Final:0 Population:400
      Patterns:[]
        Transition(78c19d255dbc506ac0a5) to:20 Symbols:[20] Input state: Output state: Population:400
            Patterns:[]
   State:20 Type:Normal Entry:0 Final:0 Population:400
      Patterns:[]
        Transition(71e25b4eb8f06cbee789) to:10 Symbols:[10] Input state: Output state: Population:399
            Patterns:[]
        Transition(da59ad894efe784627a0) to:30 Symbols:[30] Input state: Output state: Population:1
            Patterns:[Sequence 1]
   State:30 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[Sequence 1]
   State:5 Type:Normal Entry:1 Final:0 Population:1
      Patterns:[]
        Transition(c350a4b51db43024b97f) to:10 Symbols:[10] Input state: Output state: Population:1
            Patterns:[]
        ENTRY Transition(540084cf6c294994ac76) Symbols:[5] Population:1
            Patterns:[]
=========
-==*****************************************==-
```

This results in two distinct ETTs, each ETT comprising its own sequence. By examining the population value in the ETTs, we can see the collected statistic for the learned data stream. Using this Sequence Detector, we can detect only original sequences, for example

Testing sequence 6.1.

```
> tt <- data.frame(product=c("P29","P29","P34","P29","P29","P11","P34","P34",
+                            "P34","P11","P11"),
+               sales=c(5,10,76,20,30,10,58,31,11,20,30),alert=NA)
> test_streams <- list(stream=tt)
```

which results in

```
> res_test <- seq_detector$process(test_streams,learn=FALSE)
> out_test <- data.frame()
> for(i in 1:nrow(res_test$stream))
+   out_test <- rbind(out_test,data.frame(product=res_test$stream[i,"product"],
+                                         sales=res_test$stream[i,"sales"],
+                                         alert=c_to_string(res_test$explanation[[i]]$actual)))

> out_test
```

The sequence 1020 for $P11$ was not recognized as a separated sequence. This is the regular sequence. The Sequence Detector instance can be cloned for later manipulations.

```
> seq_detector1 <- seq_detector$clone()
```

Now we can make an ETT projection by using $\Phi_{\mathcal{S}_\delta} = 200$.

```
> res_is <- seq_detector$induceSubmachine(threshold=200)
```

```
> seq_detector$printMachines(print_cache=FALSE,print_keys=FALSE)
```

	product	sales	alert
1	P29	5.00	[]
2	P29	10.00	[]
3	P34	76.00	[]
4	P29	20.00	[]
5	P29	30.00	[Sequence 1]
6	P11	10.00	[]
7	P34	58.00	[]
8	P34	31.00	[]
9	P34	11.00	[Sequence 2]
10	P11	20.00	[]
11	P11	30.00	[]

Table 25: Testing sequence 6.1 results

```
-==* ETT wrapper machines list(4) *==-
Machine:3433d50f6c80d8e492cd
=========
   State:10 Type:Normal Entry:1 Final:0 Population:401
      Patterns:[]
        Transition(78c19d255dbc506ac0a5) to:20 Symbols:[20] Input state: Output state: Population:401
           Patterns:[]
        ENTRY Transition(6bf62821ee78a0c13150) Symbols:[10] Population:0
           Patterns:[]
   State:20 Type:Normal Entry:0 Final:0 Population:401
      Patterns:[]
        Transition(71e25b4eb8f06cbee789) to:10 Symbols:[10] Input state: Output state: Population:399
           Patterns:[]
=========
Machine:3f4ebb758b399757c8fb
=========
   State:31 Type:Normal Entry:0 Final:0 Population:401
      Patterns:[]
        Transition(ed3850f0d20333c0faad) to:58 Symbols:[58] Input state: Output state: Population:399
           Patterns:[]
   State:58 Type:Normal Entry:1 Final:0 Population:401
      Patterns:[]
        Transition(880f6e3f259ae9640c0c) to:31 Symbols:[31] Input state: Output state: Population:401
           Patterns:[]
        ENTRY Transition(cceb50c3e900c5e58b42) Symbols:[58] Population:0
           Patterns:[]
=========
Machine:964ab952a1a6db9635cf
=========
   State:11 Type:Normal Entry:0 Final:0 Population:2
      Patterns:[Sequence 2]
   State:76 Type:Normal Entry:1 Final:0 Population:2
      Patterns:[]
        Transition(82e0cc6216320036653b) to:df546b7c0618dadf175d Symbols:[58] Input state:58 Output state:
            Population:2
           Patterns:[]
        ENTRY Transition(f3e7d129dc008cc5cd33) Symbols:[76] Population:2
           Patterns:[]
   State:df546b7c0618dadf175d Type:Submachine Entry:0 Final:0 Population:0
     Submachine:3f4ebb758b399757c8fb
        Transition(c05f9fefb8f43436a4db) to:11 Symbols:[11] Input state: Output state:31 Population:2
           Patterns:[Sequence 2]
=========
Machine:a31ef78d0468ec069227
=========
   State:30 Type:Normal Entry:0 Final:0 Population:2
      Patterns:[Sequence 1]
   State:5 Type:Normal Entry:1 Final:0 Population:2
      Patterns:[]
        Transition(c350a4b51db43024b97f) to:875e11671b5e99633a04 Symbols:[10] Input state:10 Output state:
            Population:2
           Patterns:[]
        ENTRY Transition(540084cf6c294994ac76) Symbols:[5] Population:2
           Patterns:[]
```

```
      State:875e11671b5e99633a04 Type:Submachine Entry:0 Final:0 Population:0
        Submachine:3433d50f6c80d8e492cd
          Transition(da59ad894efe784627a0) to:30 Symbols:[30] Input state: Output state:20 Population:2
              Patterns:[Sequence 1]
=========
-==*******************************************==-
```

Four ETTs can be observed. These are two pairs of $ETT_{\mathcal{S}+}$ and $ETT_{\mathcal{S}-}$, where $ETT_{\mathcal{S}-}$ references the $ETT_{\mathcal{S}+}$. Now we can detect regular sequences. Let us create additional alerts for the regular sequences.

```
> seq_detector$setOutputPattern(states=c("20"),transitions=c(),pattern="Reg.sequence 1")
> seq_detector$setOutputPattern(states=c("31"),transitions=c(),pattern="Reg.sequence 2")
> res_test <- seq_detector$process(test_streams,learn=FALSE)
> out_test <- data.frame()
> for(i in 1:nrow(res_test$stream))
+   out_test <- rbind(out_test,data.frame(product=res_test$stream[i,"product"],
+                                          sales=res_test$stream[i,"sales"],
+                                          alert=c_to_string(res_test$explanation[[i]]$actual)))

> out_test
```

	product	sales	alert
1	P29	5.00	[]
2	P29	10.00	[]
3	P34	76.00	[]
4	P29	20.00	[Reg.sequence 1]
5	P29	30.00	[Sequence 1]
6	P11	10.00	[]
7	P34	58.00	[]
8	P34	31.00	[Reg.sequence 2]
9	P34	11.00	[Sequence 2]
10	P11	20.00	[Reg.sequence 1]
11	P11	30.00	[]

Table 26: Testing sequence 6.1 results

At this moment, we are able to detect both, whole and regular sequences.

6.3   Example 2

We can project the original Sequence Detector instance and to discard the irregular part of the learned sequences. The Sequence Detector instance is reverted back to the original, before cloning, and projection on the reverted instance is done.

```
> seq_detector <- seq_detector1$clone()
> res_is <- seq_detector$induceSubmachine(threshold=200,isolate=TRUE)

> seq_detector$printMachines(print_cache=FALSE,print_keys=FALSE)


-==* ETT wrapper machines list(2) *==-
Machine:d4910821b1a08943e540
=========
   State:10 Type:Normal Entry:1 Final:0 Population:401
      Patterns:[]
        Transition(78c19d255dbc506ac0a5) to:20 Symbols:[20] Input state: Output state: Population:401
            Patterns:[]
        ENTRY Transition(51aaeea3bca673f4e8ce) Symbols:[10] Population:0
            Patterns:[]
   State:20 Type:Normal Entry:0 Final:0 Population:401
      Patterns:[]
        Transition(71e25b4eb8f06cbee789) to:10 Symbols:[10] Input state: Output state: Population:399
            Patterns:[]
=========
Machine:7a4a6806dd7a08f83dd3
=========
   State:31 Type:Normal Entry:0 Final:0 Population:401
      Patterns:[]
```

```
            Transition(ed3850f0d20333c0faad) to:58 Symbols:[58] Input state: Output state: Population:399
                Patterns:[]
        State:58 Type:Normal Entry:1 Final:0 Population:401
            Patterns:[]
            Transition(880f6e3f259ae9640c0c) to:31 Symbols:[31] Input state: Output state: Population:401
                Patterns:[]
            ENTRY Transition(a0cd6a4c53db8b67e687) Symbols:[58] Population:0
                Patterns:[]
=========
-==****************************************==-
```

Only ETTs that comprise the most regular sequences, those that satisfy the threshold $\Phi_{\mathcal{S}_\delta} = 200$, can be observed. Isolating only the most regular sequences can have its benefits and shortfalls:

- **Benefit**: We keep only structures that are the most recurring, some would say the most common and regular behaviour, the most common process flows, etc... This helps to manage the size of the Sequence Detector instance in the Big Data environment, which is of uttermost importance.

- **Shortfall**: We lose the ability to detect irregular and anomalous sequences, which are important in detecting fraudulent behaviour and activities.

By processing the testing sequence 6.1 again, we can see that only regular subsequences yield alerts.

```
> seq_detector$setOutputPattern(states=c("20"),transitions=c(),pattern="Reg.sequence 1")
> seq_detector$setOutputPattern(states=c("31"),transitions=c(),pattern="Reg.sequence 2")
> res_test <- seq_detector$process(test_streams,learn=FALSE)
> out_test <- data.frame()
> for(i in 1:nrow(res_test$stream))
+   out_test <- rbind(out_test,data.frame(product=res_test$stream[i,"product"],
+                                 sales=res_test$stream[i,"sales"],
+                                 alert=c_to_string(res_test$explanation[[i]]$actual)))

> out_test
```

	product	sales	alert
1	P29	5.00	[]
2	P29	10.00	[]
3	P34	76.00	[]
4	P29	20.00	[Reg.sequence 1]
5	P29	30.00	[]
6	P11	10.00	[]
7	P34	58.00	[]
8	P34	31.00	[Reg.sequence 2]
9	P34	11.00	[]
10	P11	20.00	[Reg.sequence 1]
11	P11	30.00	[]

Table 27: Testing sequence 6.1 results

7 Compressing Sequence Detector

When processing Big Data, learning actions (Krleža et al., 2019) can generate a lot of states and additional ETTs. This can cause a significant increase in memory consumption and processing power need, and slowing down the processing od data items by a Statistical Detector object. Statistical projection is one of the ways how to deal with Sequence Detector complexity explosion, which helps to select and project only the most common sequences, reducing overall size of the Sequence Detector. Even so, Sequence Detector can create multiple ETTs having a substructure (a sub-graph) that is isomorphic. Since we have the ability to stack and reuse ETTs, isolating isomorphic ETT structures in referenced, child ETTs is a way how to compress Sequence Detector and reduce its size.

7.1 Example 1 - Merging

We can define two input data stream for which we certainly know that will produce two partially isomorphic ETTs.

Learning sequence 7.1.

```
> library(SeqDetect)
> ldf1 <- data.frame(product=c("P1","P1","P1","P1"),sequence_id=c(1,3,5,7),
+                     sales=c(5,76,123,1),alert=c(NA,NA,NA,"Alert P1"))
> ldf2 <- data.frame(product=c("P2","P2","P2","P2"),sequence_id=c(2,4,6,8),
+                     sales=c(21,76,123,42),alert=c(NA,NA,NA,"Alert P2"))
> input_streams <- list(stream1=ldf1,stream2=ldf2)
```

Then we define the Sequence Detector instance.

```
> pp <- HSC_PP(c("product","sales","alert","sequence_id"),"sequence_id")
> pc <- HSC_PC_Attribute("sales")
> seq_detector <- HybridSequenceClassifier(c("sequence_id","product","sales","alert"),
+                                           "sequence_id","sequence_id",context_field="product",
+                                           preclassifier=pc,preprocessor=pp,reuse_states=TRUE,
+                                           pattern_field="alert")
> seq_detector$process(input_streams,learn=TRUE)
> seq_detector$cleanKeys()
> backup_detector <- seq_detector$clone()
```

We get two ETTs after the learning process.

```
> seq_detector$printMachines(print_cache=FALSE,print_keys=FALSE)


-==* ETT wrapper machines list(2) *==-
Machine:16f03314ba4debdb238a
=========
   State:123 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[]
        Transition(819c110770679890e0ec) to:42 Symbols:[42] Input state: Output state: Population:1
           Patterns:[Alert P2]
   State:21 Type:Normal Entry:1 Final:0 Population:1
      Patterns:[]
        Transition(4ab15909114e415f762b) to:76 Symbols:[76] Input state: Output state: Population:1
           Patterns:[]
        ENTRY Transition(ac40f4aed5f405a8f141) Symbols:[21] Population:1
           Patterns:[]
   State:42 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[Alert P2]
   State:76 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[]
        Transition(1065d8306fe7d918aeea) to:123 Symbols:[123] Input state: Output state: Population:1
           Patterns:[]
=========
Machine:5e3c4a0dfd99470d2ff6
=========
   State:1 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[Alert P1]
   State:123 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[]
        Transition(95d8f1d3d91258eeb646) to:1 Symbols:[1] Input state: Output state: Population:1
           Patterns:[Alert P1]
   State:5 Type:Normal Entry:1 Final:0 Population:1
      Patterns:[]
        Transition(13f6f71b2cc897f8c40d) to:76 Symbols:[76] Input state: Output state: Population:1
           Patterns:[]
        ENTRY Transition(f7bb4658e2f245859e52) Symbols:[5] Population:1
           Patterns:[]
   State:76 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[]
        Transition(1015d4348ab883a65216) to:123 Symbols:[123] Input state: Output state: Population:1
           Patterns:[]
=========
-==*******************************==-
```

Sequence detection works as supposed. We define the following testing sequence...

Testing sequence 7.1.

```
> tdf1 <- data.frame(product=c("P3","P3","P3","P3"),sequence_id=c(1,2,3,4),
+                     sales=c(5,76,123,1),alert=NA)
> test_streams <- list(stream1=tdf1)
```

```
> res_test <- seq_detector$process(test_streams,learn=FALSE)
> out_test <- data.frame()
> for(i in 1:nrow(res_test$stream))
+   out_test <- rbind(out_test,data.frame(product=res_test$stream[i,"product"],
+                                          sales=res_test$stream[i,"sales"],
+                                          alert=c_to_string(res_test$explanation[[i]]$actual)))

> out_test
```

	product	sales	alert
1	P3	5.00	[]
2	P3	76.00	[]
3	P3	123.00	[]
4	P3	1.00	[Alert P1]

Table 28: Testing sequence 7.1 results

Merging two ETTs is possible, but it will have a negative impact on the Sequence Detector by fusing learned sequences together.

Testing sequence 7.2.

```
> tdf1 <- data.frame(product=c("P4","P4","P4","P4"),sequence_id=c(1,2,3,4),
+                     sales=c(21,76,123,1),alert=NA)
> test_streams <- list(stream1=tdf1)
> seq_detector$mergeMachines()
> res_test <- seq_detector$process(test_streams,learn=FALSE)
> out_test <- data.frame()
> for(i in 1:nrow(res_test$stream))
+   out_test <- rbind(out_test,data.frame(product=res_test$stream[i,"product"],
+                                          sales=res_test$stream[i,"sales"],
+                                          alert=c_to_string(res_test$explanation[[i]]$actual)))

> seq_detector$printMachines(print_cache=FALSE,print_keys=FALSE)

-==* ETT wrapper machines list(1) *==-
Machine:16f03314ba4debdb238a
=========
  State:1 Type:Normal Entry:0 Final:0 Population:3
    Patterns:[Alert P1]
  State:123 Type:Normal Entry:0 Final:0 Population:4
    Patterns:[]
      Transition(519bb5cca37b799019bd) to:1 Symbols:[1] Input state: Output state: Population:3
        Patterns:[Alert P1]
      Transition(819c110770679890e0ec) to:42 Symbols:[42] Input state: Output state: Population:1
        Patterns:[Alert P2]
  State:21 Type:Normal Entry:1 Final:0 Population:2
    Patterns:[]
      Transition(4ab15909114e415f762b) to:76 Symbols:[76] Input state: Output state: Population:2
        Patterns:[]
      ENTRY Transition(ac40f4aed5f405a8f141) Symbols:[21] Population:2
        Patterns:[]
  State:42 Type:Normal Entry:0 Final:0 Population:1
    Patterns:[Alert P2]
  State:5 Type:Normal Entry:1 Final:0 Population:2
    Patterns:[]
      Transition(00e751eee80395d9d957) to:76 Symbols:[76] Input state: Output state: Population:2
        Patterns:[]
      ENTRY Transition(797a2c67a81d17f5df7c) Symbols:[5] Population:2
        Patterns:[]
  State:76 Type:Normal Entry:0 Final:0 Population:4
    Patterns:[]
      Transition(1065d8306fe7d918aeea) to:123 Symbols:[123] Input state: Output state: Population:4
        Patterns:[]
=========
-==*****************************************==-


> out_test
```

Results in Table 29 are incorrect. Notice that we recognize the sequence *21,76,123,1*, which a combination of sequences for two different contexts in the learning sequence 7.1.

	product	sales	alert
1	P4	21.00	[]
2	P4	76.00	[]
3	P4	123.00	[]
4	P4	1.00	[Alert P1]

Table 29: Testing sequence 7.2 results

7.2 Example 2 - compression

The previous result is not correct, as it wrongly detects the sequence for the product $P1$, while in fact it was a combination of sequences for $P1$ and $P2$. However, if we isolate the isomorphic parts of the ETTs...

Testing sequence 7.3.

```
> tdf1 <- data.frame(product=c("P5","P5","P5","P5"),sequence_id=c(1,2,3,4),
+                     sales=c(21,76,123,1),alert=NA)
> test_streams <- list(stream1=tdf1)

> seq_detector <- backup_detector$clone()
> seq_detector$compressMachines()
> res_test <- seq_detector$process(test_streams,learn=FALSE)
> out_test <- data.frame()
> for(i in 1:nrow(res_test$stream))
+   out_test <- rbind(out_test,data.frame(product=res_test$stream[i,"product"],
+                                          sales=res_test$stream[i,"sales"],
+                                          alert=c_to_string(res_test$explanation[[i]]$actual)))
```

...we get the correct result. No detection occurs when an intertwined sequence is tested.

```
> seq_detector$printMachines(print_cache=FALSE,print_keys=FALSE)


-==* ETT wrapper machines list(3) *==-
Machine:9a381c8c4db80d504d7e
=========
   State:123 Type:Normal Entry:0 Final:0 Population:3
      Patterns:[]
   State:76 Type:Normal Entry:0 Final:0 Population:3
      Patterns:[]
        Transition(d76f4788b5d4193d6753) to:123 Symbols:[123] Input state: Output state: Population:3
           Patterns:[]
=========
Machine:17c6587110923e1d2df7
=========
   State:21 Type:Normal Entry:1 Final:0 Population:2
      Patterns:[]
        Transition(4ab15909114e415f762b) to:e19e60ca3a2c9858b19d Symbols:[76] Input state:76 Output state:
           Population:2
           Patterns:[]
        ENTRY Transition(ac40f4aed5f405a8f141) Symbols:[21] Population:2
           Patterns:[]
   State:42 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[Alert P2]
   State:e19e60ca3a2c9858b19d Type:Submachine Entry:0 Final:0 Population:1
     Submachine:9a381c8c4db80d504d7e
        Transition(819c110770679890e0ec) to:42 Symbols:[42] Input state: Output state:123 Population:1
           Patterns:[Alert P2]
=========
Machine:59e2cba07d25114d75cd
=========
   State:1 Type:Normal Entry:0 Final:0 Population:1
      Patterns:[Alert P1]
   State:5 Type:Normal Entry:1 Final:0 Population:1
      Patterns:[]
        Transition(13f6f71b2cc897f8c40d) to:b5b4fd8ee42793d72738 Symbols:[76] Input state:76 Output state:
           Population:1
           Patterns:[]
        ENTRY Transition(f7bb4658e2f245859e52) Symbols:[5] Population:1
           Patterns:[]
   State:b5b4fd8ee42793d72738 Type:Submachine Entry:0 Final:0 Population:0
```

```
        Submachine:9a381c8c4db80d504d7e
            Transition(95d8f1d3d91258eeb646) to:1 Symbols:[1] Input state: Output state:123 Population:1
                Patterns:[Alert P1]
=========
-==***************************************==-


> out_test
```

	product	sales	alert
1	P5	21.00	[]
2	P5	76.00	[]
3	P5	123.00	[]
4	P5	1.00	[]

Table 30: Testing sequence 7.3 results

However, the correct sequence gets recognized successfully.

Testing sequence 7.4.

```
> tdf1 <- data.frame(product=c("P6","P6","P6","P6"),
+                    sequence_id=c(1,2,3,4),sales=c(5,76,123,1),
+                    alert=NA)
> test_streams <- list(stream1=tdf1)

> res_test <- seq_detector$process(test_streams,learn=FALSE)
> out_test <- data.frame()
> for(i in 1:nrow(res_test$stream))
+   out_test <- rbind(out_test,data.frame(product=res_test$stream[i,"product"],
+                                         sales=res_test$stream[i,"sales"],
+                                         alert=c_to_string(res_test$explanation[[i]]$actual)))

> out_test
```

	product	sales	alert
1	P6	5.00	[]
2	P6	76.00	[]
3	P6	123.00	[]
4	P6	1.00	[Alert P1]

Table 31: Testing sequence 7.4 results

8 Other options

8.1 Saving and loading Sequence Detector

Having C++ objects, a Sequence Detector object cannot be saved properly without serialization. First, we define a simple Sequence Detector object and a learning sequence.

Learning sequence 8.1.

```
> st <- data.frame(product=c("P1","P1"),sales=c(5,76),alert=c(NA,"Alert"))
> input_streams <- list(stream=st)

> pp <- HSC_PP(c("product","sales","alert"),"sequence_id",auto_id=TRUE)
> pc <- HSC_PC_Attribute("sales")
> seq_detector_oo <- HybridSequenceClassifier(c("sequence_id","product","sales","alert"),"sequence_id",
+                                             "sequence_id",context_field="product",preclassifier=pc,
+                                             preprocessor=pp,reuse_states=TRUE,pattern_field="alert")
> seq_detector_oo$process(input_streams,learn=TRUE)
```

Before saving the Sequence Detector object instance, we need to perform the serialization procedure.

```
> seq_detector_oo$serialize()
```

The results of the serialization can be checked in the *cache* field of the Sequence Detector object.

```
> c_to_string(names(seq_detector_oo$cache))
```

```
[1] "[46463c7d278969966be9,options,ETT_Version]"
```

```
> c_to_string(names(seq_detector_oo$cache[[1]]))
```

```
[1] "[states,transitions,tokens,locked,extend_fst_entry]"
```

```
> c_to_string(names(seq_detector_oo$cache[[1]][["states"]]))
```

```
[1] "[76,5]"
```

```
> saveRDS(seq_detector_oo,"test.RDS")
```

After loading, the deserialization method *deserialize* can be invoked explicitly. However, this method is invoked implicitly as soon as some of the Sequence Detector methods is invoked.

```
> new_seq_detector_oo <- readRDS("test.RDS")
```

```
> new_seq_detector_oo$printMachines()
```

```
-==* ETT wrapper machines list(1) *==-
Machine:46463c7d278969966be9
=========
   Common cache:[P1]
   State:5 Type:Normal Entry:1 Final:0 Population:1
      Cache:[P1]
      Keys:[]
      Patterns:[]
        Transition(e53d73cd64ecfa9ae931) to:76 Symbols:[76] Input state: Output state: Population:1
           Patterns:[Alert]
        ENTRY Transition(6d98928c30fc7f1572b8) Symbols:[5] Population:1
           Patterns:[]
   State:76 Type:Normal Entry:0 Final:0 Population:1
      Cache:[P1]
      Keys:[P1]
      Patterns:[Alert]
=========
-==****************************************==-
```

8.1.1   Serializing and deserializing into an external named list

A Sequence Detector object could be serialized and deserialized into an external variable. Serializing is done into a named list.

```
> sd_list <- new_seq_detector_oo$serializeToList()
> c_to_string(names(sd_list))
```

```
[1] "[46463c7d278969966be9,options,ETT_Version,hsc_options]"
```

```
> c_to_string(names(sd_list[[1]]))
```

```
[1] "[states,transitions,tokens,locked,extend_fst_entry]"
```

```
> c_to_string(names(sd_list[[1]][["states"]]))
```

```
[1] "[5,76]"
```

```
> totally_new_sd_oo <- deserializeFromList(sd_list)
```

```
> totally_new_sd_oo$printMachines()
```

```
-==* ETT wrapper machines list(1) *==-
Machine:46463c7d278969966be9
=========
   Common cache:[P1]
   State:5 Type:Normal Entry:1 Final:0 Population:1
      Cache:[P1]
      Keys:[]
      Patterns:[]
        Transition(e53d73cd64ecfa9ae931) to:76 Symbols:[76] Input state: Output state: Population:1
            Patterns:[Alert]
        ENTRY Transition(6d98928c30fc7f1572b8) Symbols:[5] Population:1
            Patterns:[]
   State:76 Type:Normal Entry:0 Final:0 Population:1
      Cache:[P1]
      Keys:[P1]
      Patterns:[Alert]
=========
-==****************************************==-
```

8.2 Merging ETTs

In case when multiple ETTs are created, they can be merged into a single ETT. As mentioned in (Krleža et al., 2019), if there are no common point in ETTs, this will create a single ETT having disconnected structure. Let us amend the previously generated Sequence Detector.

Learning sequence 8.2.

```
> st <- data.frame(product=c("P5","P5","P5"),sales=c(9,76,10),
+                    alert=c(NA,"Alert","Alert P5"))
> input_streams <- list(stream=st)
> totally_new_sd_oo$process(input_streams,learn=TRUE)

> totally_new_sd_oo$printMachines()


-==* ETT wrapper machines list(2) *==-
Machine:ee1b52436427b0f9e353
=========
   Common cache:[P5]
   State:10 Type:Normal Entry:0 Final:0 Population:1
      Cache:[P5]
      Keys:[P5]
      Patterns:[Alert P5]
   State:76 Type:Normal Entry:0 Final:0 Population:1
      Cache:[P5]
      Keys:[]
      Patterns:[Alert]
        Transition(47f260c2c3e02ad7e096) to:10 Symbols:[10] Input state: Output state: Population:1
            Patterns:[Alert P5]
   State:9 Type:Normal Entry:1 Final:0 Population:1
      Cache:[P5]
      Keys:[]
      Patterns:[]
        Transition(d93fe7dd78f96327cf38) to:76 Symbols:[76] Input state: Output state: Population:1
            Patterns:[Alert]
        ENTRY Transition(246051b694cb0cf26ddb) Symbols:[9] Population:1
            Patterns:[]
=========
Machine:46463c7d278969966be9
=========
   Common cache:[P1]
   State:5 Type:Normal Entry:1 Final:0 Population:1
      Cache:[P1]
      Keys:[]
      Patterns:[]
        Transition(e53d73cd64ecfa9ae931) to:76 Symbols:[76] Input state: Output state: Population:1
            Patterns:[Alert]
        ENTRY Transition(6d98928c30fc7f1572b8) Symbols:[5] Population:1
            Patterns:[]
   State:76 Type:Normal Entry:0 Final:0 Population:1
      Cache:[P1]
```

```
        Keys:[P1]
        Patterns:[Alert]
=========
-==*******************************==-
```

The result are two ETTs that have a common state 76 of a pattern *Alert*. Merging the Sequence Detector results in a single connected ETT.

```
> totally_new_sd_oo$mergeMachines()

> totally_new_sd_oo$printMachines()


-==* ETT wrapper machines list(1) *==-
Machine:ee1b52436427b0f9e353
=========
   Common cache:[P1,P5]
   State:10 Type:Normal Entry:0 Final:0 Population:1
      Cache:[P5]
      Keys:[P5]
      Patterns:[Alert P5]
   State:5 Type:Normal Entry:1 Final:0 Population:1
      Cache:[P1]
      Keys:[]
      Patterns:[]
        Transition(191035233fa2240dcc9c) to:76 Symbols:[76] Input state: Output state: Population:1
           Patterns:[Alert]
        ENTRY Transition(c7baac7319b9254abb4a) Symbols:[5] Population:1
           Patterns:[]
   State:76 Type:Normal Entry:0 Final:0 Population:2
      Cache:[P1,P5]
      Keys:[P1]
      Patterns:[Alert]
        Transition(47f260c2c3e02ad7e096) to:10 Symbols:[10] Input state: Output state: Population:1
           Patterns:[Alert P5]
   State:9 Type:Normal Entry:1 Final:0 Population:1
      Cache:[P5]
      Keys:[]
      Patterns:[]
        Transition(d93fe7dd78f96327cf38) to:76 Symbols:[76] Input state: Output state: Population:1
           Patterns:[Alert]
        ENTRY Transition(246051b694cb0cf26ddb) Symbols:[9] Population:1
           Patterns:[]
=========
-==*******************************==-
```

However, such merged ETT intertwines sequences from $P1$ and $P5$. This might be a good thing for the process discovery, but not so good for detecting sequences in time-series datasets. For example, the following testing stream mixes the learned data sequences...

Testing sequence 8.1.

```
> tt <- data.frame(product=c("P10","P10","P10"),sales=c(5,76,10),alert=NA)
> test_streams <- list(stream=tt)

> res_test1 <- totally_new_sd_oo$process(test_streams,learn=FALSE)
> out_test1 <- data.frame()
> for(i in 1:nrow(res_test1$stream))
+   out_test1 <- rbind(out_test1,data.frame(product=res_test1$stream[i,"product"],
+                                        sales=res_test1$stream[i,"sales"],
+                                        alert=c_to_string(res_test1$explanation[[i]]$actual)))
```

	product	sales	alert
1	P10	5.00	[]
2	P10	76.00	[Alert]
3	P10	10.00	[Alert P5]

Table 32: Testing sequence 8.1 results

References

Krleža, D., Vrdoljak, B., and Brčič, M. (2019). Latent process discovery using evolving tokenized transducer. *IEEE Access*, 7.