DAVE Framework Learning Analytics Algorithms

Yet Analytics

July 27, 2018

Introduction

This document introduces the initial learning analytics algorithms, **timeline of learner success** and **which assessment questions are the most difficult** of the DAVE framework. This document will be updated to include the remaining learning analytics questions defined within the 2018 TLA Data Requirements document in addition to other learning analytics algorithms which have yet to be defined. Updates may also address refinment of these algorithms and this document should be understood to be an example of algorithm presentation and not the final state of any defined algorithm.

The structure of this documents is as follows:

1. A formal specification for the data standard xAPI written in Z and referenced within the formal specifications of learning analytics algorithms
2. An algorithm definition which will consist of:
   (a) an introduction for the algorithm
   (b) the structure of the ideal input data
   (c) how to retrieve input data from an LRS
   (d) the statement parameters which the algorithm will utilize
   (e) any issues with the data collected during the 2018 pilot test of the TLA
   (f) a summary of the algorithm
   (g) the formal specification of the algorithm
   (h) pseudocode representation of the algorithm
   (i) JSONSchema for the output of the algorithm
   (j) a description of the associated visualization
   (k) a prototype of the visualization
   (l) a collection of suggestions describing how the algorithm could be adapted to improve the quality of the visualization prototype

1 xAPI Formal Specification

The current formal specification only defines xAPI statements abstractly within the context of Z. A concrete definition for xAPI statements it outside the scope of this document.

1.1 Basic Types

$IFI ::= mbox \mid mbox\_sha1sum \mid openid \mid account$

- Type unique to Agents and Groups, The concrete definition of the listed values is outside the scope of this specification

$OBJECTTYPE ::= Agent \mid Group \mid SubStatement \mid StatementRef \mid Activity$

- A type which can be present in all activities as defined by the xAPI specification

$INTERACTIONTYPE ::= true-false \mid choice \mid fill-in \mid long-fill-in \mid matching \mid performance \mid sequencing \mid likert \mid numeric \mid other$

- A type which represents the possible interactionTypes as defined within the xAPI specification

$INTERACTIONCOMPONENT ::= choices \mid scale \mid source \mid target \mid steps$

- A type which represents the possible interaction components as defined within the xAPI specification
- the concrete definition of the listed values is outside the scope of this specification

$CONTEXTTYPES ::= parent \mid grouping \mid category \mid other$

- A type which represents the possible context types as defined within the xAPI specification

$[STATEMENT]$

- Basic type for the results of querying an LRS

$[AGENT, GROUP]$

- Basic types for Agents and collections of Agents

1.2 Id Schema

$$
\begin{array}{|l}
\hline
Id \\
\hline
id : \mathbb{F}_1 \#1 \\
\hline
\end{array}
$$

- the schema Id introduces the component id which is a non-empty finite set of 1 value

1.3 Schemas for Agents, Groups and Actors

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Agent}} \\
agent : AGENT \\
objectType : OBJECTTYPE \\
name : \mathbb{F}_1 \#1 \\
ifi : IFI \\
\hline
objectType = Agent \\
agent = \{ifi\} \cup \mathbb{P}\{name, objectType\} \\
\hline
\end{array}
$$

- The schema *Agent* introduces the component *agent* which is a set consisting of an ifi and optionally an *objectType* and/or *name*

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Member}} \\
Agent \\
member : \mathbb{F}_1 \\
\hline
member = \{a : AGENT \mid \forall a : a_0..a_n \bullet a = agent\} \\
\hline
\end{array}
$$

- The schema $Member$ introduces the component *member* which is a set of objects a, where for every a within $a_0..a_n$, a is an *agent*

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Group}} \\
Member \\
group : GROUP \\
objectType : OBJECTTYPE \\
ifi : IFI \\
name : \mathbb{F}_1 \#1 \\
\hline
objectType = Group \\
group = \{objectType, name, member\} \vee \{objectType, member\} \vee \\
\qquad \{objectType, ifi\} \cup \mathbb{P}\{name, member\} \\
\hline
\end{array}
$$

- The schema *Group* introduces the component *group* which is of type *GROUP* and is a set of either *objectType* and *member* with optionaly *name* or *objectType* and ifi with optionally *name* and/or *member*

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\textit{Actor}} \\
Agent \\
Group \\
actor : AGENT \vee GROUP \\
\hline
actor = agent \vee group \\
\hline
\end{array}
$$

- The schema *Actor* introduces the component *actor* which is either an *agent* or *group*

1.4 Verb Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{Verb} \\ \hline
Id \\
display, verb : \mathbb{F}_1 \\ \hline
verb = \{id, display\} \vee \{id\} \\ \hline
\end{array}
$$

- The schema $Verb$ introduces the component $verb$ which is a set that consists of either id and the non-empty finite set $display$ or just id

1.5 Object Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{Extensions} \\ \hline
extensions, extensionVal : \mathbb{F}_1 \\
extensionId : \mathbb{F}_1 \ \#1 \\ \hline
extensions = \{e : (extensionId, extensionVal) \ | \forall i, j : e_i..e_j \bullet \\
\qquad (extensionId_i, extensionVal_i) \vee (extensionId_i, extensionVal_j) \wedge \\
\qquad (extensionId_j, extensionVal_i) \vee (extensionId_j, extensionVal_j) \wedge \\
\qquad extensionId_i \neq extensionId_j\} \\ \hline
\end{array}
$$

- The schema $Extensions$ introduces the component $extensions$ which is a non-empty finite set that consists of ordered pairs of $extensionId$ and $extensionVal$. Different $extensionId$s can have the same $extensionVal$ but there can not be two identical $extensionId$ values
- $extensionId$ is a non-empty finite set with one value
- $extensionVal$ is a non-empty finite set

$$
\begin{array}{|l}
\multicolumn{1}{l}{InteractionActivity} \\ \hline
interactionType : INTERACTIONTYPE \\
correctResponsePattern : \mathrm{seq}_1 \\
interactionComponent : INTERACTIONCOMPONENT \\ \hline
interactionActivity = \{interactionType, correctReponsePattern, interactionComponent\} \vee \\
\qquad \{interactionType, correctResponsePattern\} \\ \hline
\end{array}
$$

- The schema $InteractionActivity$ introduces the component $interactionActivity$ which is a set of either $interactionType$ and $correctResponsePattern$ or $interactionType$ and $correctResponsePattern$ and $interactionComponent$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Definition}} \\
\hline
InteractionActivity \\
Extensions \\
definition, name, description : \mathbb{F}_1 \\
type, moreInfo : \mathbb{F}_1 \#1 \\
\hline
definition = \mathbb{P}_1\{name, description, type, moreInfo, extensions, interactionActivity\} \\
\hline
\end{array}
$$

- The schema $Definition$ introduces the component $definition$ which is the non-empty, finite power set of $name$, $description$, $type$, $moreInfo$ and $extensions$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Object}} \\
\hline
Id \\
Definition \\
Agent \\
Group \\
Statement \\
objectTypeA, objectTypeS, objectTypeSub, objectType : OBJECTTYPE \\
substatement : STATEMENT \\
object : \mathbb{F}_1 \\
\hline
substatement = statement \\
objectTypeA = Activity \\
objectTypeS = StatementRef \\
objectTypeSub = SubStatement \\
objectType = objectTypeA \vee objectTypeS \\
object = \{id\} \vee \{id, objectType\} \vee \{id, objectTypeA, definition\} \\
\qquad \vee\{id, definition\} \vee \{agent\} \vee \{group\} \vee \{objectTypeSub, substatement\} \\
\qquad \vee\{id, objectTypeA\} \\
\hline
\end{array}
$$

- The schema $Object$ introduces the component $object$ which is a non-empty finite set of either id, id and $objectType$, id and $objectTypeA$ and $definition$, $agent$, $group$, or $substatement$
- The schema $Statement$ and the corresponding component $statement$ will be defined later on in this specification

1.6 Result Schema

$$\begin{array}{|l}
\textit{Score} \\
\hline
score : \mathbb{F}_1 \\
scaled, min, max, raw : \mathbb{Z} \\
\hline
scaled = \{n : \mathbb{Z} \mid -1.0 \leq n \leq 1.0\} \\
min = n < max \\
max = n > min \\
raw = \{n : \mathbb{Z} \mid min \leq n \leq max\} \\
score = \mathbb{P}_1\{scaled, raw, min, max\} \\
\hline
\end{array}$$

- The schema *Score* introduces the component *score* which is the non-empty powerset of *min*, *max*, *raw* and *scaled*

$$\begin{array}{|l}
\textit{Result} \\
\hline
Score \\
Extensions \\
success, completion, response, duration : \mathbb{F}_1 \#1 \\
result : \mathbb{F}_1 \\
\hline
success = \{true\} \vee \{false\} \\
completion = \{true\} \vee \{false\} \\
result = \mathbb{P}_1\{score, success, completion, response, duration, extensions\} \\
\hline
\end{array}$$

- The schema *Result* introduces the component *result* which is the non-empty power set of *score*, *success*, *completion*, *response*, *duration* and *extensions*

1.7 Context Schema

$$\begin{array}{|l}
\textit{Instructor} \\
\hline
Agent \\
Group \\
instructor : AGENT \vee GROUP \\
\hline
instructor = agent \vee group \\
\hline
\end{array}$$

- The schema *Instructor* introduces the component *instructor* which can be ether an *agent* or a *group*

$$\begin{array}{|l}
\textit{Team} \\
\hline
Group \\
team : GROUP \\
\hline
team = group \\
\hline
\end{array}$$

- The schema $Team$ introduces the component $team$ which is a $group$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ Context\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
Instructor\\
Team\\
Object\\
Extensions\\
registration, revision, platform, language : \mathbb{F}_1\ \#1\\
parentT, groupingT, categoryT, otherT : CONTEXTTYPES\\
contextActivities, statement : \mathbb{F}_1\\
\hline
statement = object \setminus (id, objectType, agent, group, definition)\\
parentT = parent\\
groupingT = grouping\\
categoryT = category\\
otherT = other\\
contextActivity = \{ca : object \setminus (agent, group, objectType, objectTypeSub, substatement)\}\\
contextActivityParent = (parentT, contextActivity)\\
contextActivityCategory = (categoryT, contextActivity)\\
contextActivityGrouping = (groupingT, contextActivity)\\
contextActivityOther = (otherT, contextActivity)\\
contextActivities = \mathbb{P}_1\{contextActivityParent, contextActivityCategory,\\
\qquad\qquad\qquad\qquad contextActivityGrouping, contextActivityOther\}\\
context = \mathbb{P}_1\{registration, instructor, team, contextActivities, revision,\\
\qquad\qquad platform, language, statement, extensions\}\\
\hline
\end{array}
$$

- The schema $Context$ introduces the component $context$ which is the non-empty powerset of $registration$, $instructor$, $team$, $contextActivities$, $revision$, $platform$, $language$, $statement$ and $extensions$

1.8 Timestamp and Stored Schema

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ Timestamp\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
timestamp : \mathbb{F}_1\ \#1\\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ Stored\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
stored : \mathbb{F}_1\ \#1\\
\hline
\end{array}
$$

- The schema $Timestamp$ and $stored$ introduce the components $timestamp$ and $stored$ respectively. Each are non-empty finite sets containing one value

1.9 Attachements Schema

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Attachments}} \\ \hline
display, description, attachment, attachments : \mathbb{F}_1 \\
usageType, sha2, fileUrl, contexntType : \mathbb{F}_1 \ \#1 \\
length : \mathbb{N} \\ \hline
attachment = \{usageType, display, contentType, length, sha2\} \cup \mathbb{P}\{description, fileUrl\} \\
attachments = \{a : attachment\} \\ \hline
\end{array}$$

- The schema *Attachements* introduces the componenet *attachements* which is a non-empty finite set of the component *attachement*
- The component *attachment* is a non-empty finite set of the componenets *usageType*, *display*, *contentType*, *length*, *sha*2 with optionally *description* and/or *fileUrl*

1.10 Statement and Statements Schema

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Statement}} \\ \hline
Id \\
Actor \\
Verb \\
Object \\
Result \\
Context \\
Timestamp \\
Stored \\
Attachements \\
statement : STATEMENT \\ \hline
statement = \{actor, verb, object, stored\} \cup \\
\qquad\qquad \mathbb{P}\{\mathrm{id}, result, context, timestamp, attachments\} \\ \hline
\end{array}$$

- The schema *Statement* introduces the component *statement* which consists of the components *actor*, *verb*, *object* and *stored* and the optional components *id*, *result*, *context*, *timestamp*, and/or *attachments*
- The schema *Statement* allows for subcomponent of *statement* to refrenced via the . (selection) operator

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Statements}} \\ \hline
Statement \\
statements : \mathbb{F}_1 \\ \hline
statements = \{s : statement\} \\ \hline
\end{array}$$

- The schema *Statements* introduces the component *statements* which is a non-empty finite set of components *statement*

2 Timeline Of Learner Success

As learners engage in a blended eLearning ecosystem, they will build up a history of learning experiences. When that eLearning ecosystem adheres to a framework dedicated to supporting and understanding the learner, such as the Total Learning Architecture (TLA), it becomes possible to retell their story through data. One important aspect of that story is the learner's history of success.

2.1 Ideal Statements

In order to accurately portray a learner's timeline of success, there are a few requirements of the data produced by a Learning Record Provider (LRP). They are as follows:

- the learner must be uniquely and consistently identified across all statements
- learning activities which evaluate a learner's understanding of material must report if the learner was successful or not
  - the grade earned by the learner must be reported
  - the minimum and maximum possible grade must be reported
- The learning activities must be uniquely and consistently identified across all statements
- The time at which a learner completed a learning activity must be recorded
  - The timestamp should contain an appropriate level of specificity.
  - ie. Year, Month, Day, Hour, Minute, Second, Timezone

2.2 Input Data Retrieval

How to query an LRS via a GET request to the Statements Resource [1] [2] [3]

```
Agent = "agent={"account":
                {"homePage": "https://example.homepage",
                 "name": 123456}}"

Since = "since=2018-07-20T12:08:47Z"
```

[1] S is the set of all statements parsed from the statements array within the HTTP response to the Curl request. It may be possible that multiple Curl requests are needed to retrieve all query results. If multiple requests are necessary, S is the result of concatenating the result of each request into a single set

[2] Querying an LRS will not be defined within the following Z specifications but the results of the query will be

[3] If you want to query across the entire history of a LRS, omit Since and Until from the endpoint

```
Until = "until=2018-07-21T12:08:47Z"

Base = "https://example.endpoint/statements?"

endpoint = Base + Agent + "&" + Since + "&" + Until

Auth = Hash generated from basic auth

S = curl -X GET -H "Authorization: Auth"
         -H "Content-Type: application/json"
         -H "X-Experience-API-Version: 1.0.3"
         Endpoint
```

2.3 Statement Parameters to Utilize

The statement parameter locations here are written in JSONPath. This notation is also compatable with the xAPI Z notation

- $\$.timestamp$
- $\$.result.success$
- $\$.result.score.raw$
- $\$.result.score.min$
- $\$.result.score.max$
- $\$.verb.id$

2.4 2018 Pilot TLA Statement Problems

The initial pilot test data supports this algorithm. Given that the offical 2018 pilot test is scheduled to take place on July 27th, 2018, this section may require updates pending data review.

2.5 Summary

1. Query an LRS via a GET request to the statements endpoint using the parameters agent, since and until
2. Filter the results to the set of statements where:
   - $\$.verb.id$ is one of:
     - http://adlnet.gov/expapi/verbs/passed
     - https://w3id.org/xapi/dod-isd/verbs/answered

    - http://adlnet.gov/expapi/verbs/completed
  - $\$.result.success$ is true

3. process the filtered data
   - extract the timestamp
   - extract the score values from the statement and convert them to a scale of 0..100
   - create a pair of [timestamp, scaled-score]

2.6 Formal Specification

2.6.1 Basic Types

$COMPLETION ::==$
$\{http://adlnet.gov/expapi/verbs/passed\} \mid$
$\{https://w3id.org/xapi/dod-isd/verbs/answered\} \mid$
$\{http://adlnet.gov/expapi/verbs/completed\}$

$SUCCESS ::== \{true\}$

2.6.2 System State

$$
\begin{array}{|l}
\hline
\textit{TimelineLearnerSuccess} \\
Statements \\
S_{all} : \mathbb{F}_1 \\
S_{completion}, S_{success}, S_{processed} : \mathbb{F} \\
\hline
S_{all} = statements \\
S_{completion} \subseteq S_{all} \\
S_{success} \subseteq S_{completion} \\
S_{processed} = \{pair : (statement.timestamp, \mathbb{N}\#1)\} \\
\hline
\end{array}
$$

- The set S_{all} is a non-empty finite set and is the component *statements*
- The sets $S_{completion}$ and $S_{success}$ are both finite sets
- the set $S_{completion}$ is a subset of S_{all}
- the set $S_{success}$ is a subset of $S_{completion}$
- the set $S_{processed}$ is a finite set of pairs where each contains a *statement.timestamp* and a natural number

2.6.3 Initial System State

$$
\begin{array}{|l}
\hline
\textit{InitTimelineLearnerSuccess} \\
\textit{TimelineLearnerSuccess} \\
\hline
S_{all} \neq \emptyset \\
S_{completion} = \emptyset \\
S_{success} = \emptyset \\
S_{processed} = \emptyset \\
\hline
\end{array}
$$

- The set S_{all} is a non-empty set
- The sets $S_{completion}, S_{success}$ and $S_{processed}$ are all initially empty

2.6.4 Filter for Completion

$$
\begin{array}{|l}
\hline
\textit{Completion} \\
\textit{Statement} \\
\textit{completion} : \textit{STATEMENT} \nrightarrow \mathbb{F} \\
s? : \textit{STATEMENT} \\
s! : \mathbb{F} \\
\hline
s? = \textit{statement} \\
s! = \textit{completion}(s?) \\
\textit{completion}(s?) = \textbf{if } s?.\textit{verb.id} : \textit{COMPLETION} \\
\qquad\qquad\qquad\qquad \textbf{then } s! = s? \\
\qquad\qquad\qquad\qquad \textbf{else } s! = \emptyset \\
\hline
\end{array}
$$

- The schema *Completion* inroduces the function *completion* which takes in the variable $s?$ and returns the variable $s!$
- The variable $s?$ is the component *statement*
- $s!$ is equal to $s?$ if $\$.verb.id$ is of the type $COMPLETION$ otherwise $s!$ is an empty set

$$
\begin{array}{|l}
\hline
\textit{FilterForCompletion} \\
\Delta \textit{TimelineLearnerSuccess} \\
\textit{Completion} \\
\textit{completions} : \mathbb{F} \\
\hline
\textit{completions} = S_{all} \\
\textit{completions}' = \{ s : \textit{STATEMENT} \mid \textit{completion}(s) \neq \emptyset \} \\
S'_{completion} = S_{completion} \cup \textit{completions}' \\
\hline
\end{array}
$$

- the set *completions* is the set S_{all}
- The set *completions'* is the set of all statements s where the result of $completion(s)$ is not an empty set

- the updated set $S'_{completion}$ is the union of the previous state of set $S_{completion}$ and the set $completions'$

2.6.5 Filter for Success

$$
\begin{array}{|l}
\hline
\textit{Success} \\
\textit{Statement} \\
success : STATEMENT \rightarrow \mathbb{F} \\
s? : STATEMENT \\
s! : \mathbb{F} \\
\hline
s? = statement \\
s! = success(s?) \\
success(s?) = \textbf{if } s?.result.success : SUCCESS \\
\qquad\qquad\qquad \textbf{then } s! = s? \\
\qquad\qquad\qquad \textbf{else } s! = \emptyset \\
\hline
\end{array}
$$

- the schema *Success* introduces the function *success* which takes in the variable $s?$ and returns the variable $s!$
- the variable $s?$ is the component *statement*
- $s!$ is equal to $s?$ if $\$.result.success$ is of the type $SUCCESS$ otherwise $s!$ is an empty set

$$
\begin{array}{|l}
\hline
\textit{FilterForSuccess} \\
\Delta TimelineLearnerSuccess \\
Success \\
successes : \mathbb{F} \\
\hline
successes = S_{completion} \\
successes' = \{s : STATEMENT \mid success(s) \neq \emptyset\} \\
S'_{success} = S_{success} \cup successes' \\
\hline
\end{array}
$$

- the set *successes* is the set $S_{completion}$
- The set $successes'$ contains elements s of type $STATEMENT$ where $success(s)$ is not an empty set
- The updated set $S'_{success}$ is the union of the previous state of $S_{success}$ and $successes'$

2.6.6 Processes Results

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;Scale\;}} \\
scaled! : \mathbb{N} \\
raw?, min?, max? : \mathbb{Z} \\
scale : \mathbb{Z} \rightarrow \mathbb{N} \\
\hline
scaled! = scale(raw?, min?, max?) \\
scale(raw?, min?, max?) = \\
\qquad\qquad (raw? * ((0.0 - 100.0)\ div\ (min? - max?))) + \\
\qquad\qquad (0.0 - (min? * ((0.0 - 100.0) div (min? - max?)))) \\
\hline
\end{array}
$$

- The schema $Scale$ introduces the function $scale$ which takes 3 arguments, $raw?$, $min?$ and $max?$. The function converts $raw?$ from the range $min? .. max?$ to 0.0..100.0

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;ProcessStatements\;}} \\
\Delta TimelineLearnerSuccess \\
Scale \\
FilterStatements \\
processed : \mathbb{F} \\
\hline
processed = S_{success} \\
processed' = \{p : (\mathbb{F}_1 \#1, \mathbb{N}\#1) \mid \\
\qquad\qquad \mathbf{let}\ \{processed_i..processed_j\} == \{s_i..s_j\} \bullet \\
\qquad\qquad \forall i, j : s_i..s_j \bullet \exists p_i..p_j \bullet \\
\qquad\qquad first\, p_i = s_i.timestamp \wedge \\
\qquad\qquad second\, p_i = scale(s_i.result.score.raw, \\
\qquad\qquad\qquad\qquad s_i.result.score.min, \\
\qquad\qquad\qquad\qquad s_i.result.score.max) \wedge \\
\qquad\qquad first\, p_j = s_j.timestamp \wedge \\
\qquad\qquad second\, p_j = scale(s_j.result.score.raw, \\
\qquad\qquad\qquad\qquad s_j.result.score.min, \\
\qquad\qquad\qquad\qquad s_j.result.score.max)\} \\
S'_{processed} = S_{processed} \cup processed' \\
\hline
\end{array}
$$

- The operation $ProcessStatements$ introduces the variable $processed$ which is equalivant to the set $S_{success}$ which is the result of the operation $FilterStatements$
- The operation defines the variable $processsed'$ which is a set of objects p which are each an ordered pair of (1) a finite set containing one value and (2) a single positive number.
- The first component of every object p, is the timestamp from the associated $statement$ within $processed$ ie. $s.timestamp$
- The second component of every object p is the result of the function $scale$. The score values contained within the associated $statement\ s$ are the arugments passed to $scale$. ie $scale(s.result.score.raw, s.result.score.min, s.result.score.max)$

- The result of the operation $ProcessStatements$ is to updated the set $S_{processed}$ with the values contained within $processed'$

2.6.7 Sequence of Operations

$FilterStatements \mathrel{\widehat=} FilterForCompletion \mathbin{;} FilterForSuccess$

- The schema $FilterStatements$ is the sequential composition of operation schemas $FilterForCompletion$ and $FilterForSuccess$
- $FilterForCompletion$ happens before $FilterForSuccess$

$ProcessedStatements \mathrel{\widehat=} FilterStatements \mathbin{;} ProcessStatements$

- The schema $ProcessedStatements$ is the sequential composition of operation schemas $FilterStatements$ and $ProcessStatements$
- $FilterStatements$ happens before $ProcessStatements$

2.6.8 Return

$$
\begin{array}{|l}
\textit{Return} \\
\hline
\Xi TimelineLearnerSuccess \\
ProcessedStatements \\
S_{processed}! : \mathbb{F} \\
\hline
S_{processed}! = S_{processed} \\
\hline
\end{array}
$$

- The returned variable $S_{processed}!$ is equal to the current state of variable $S_{processed}$ after the operations $FilterForCompletion$, $FilterForSuccess$ and $ProcessStatements$

2.7 Pseudocode

Algorithm 1: Timeline of Learner Success

```
Input: S_all
Result: coll
init coll := []
while S_all is not empty do
    for each statement s in S_all
    if s.verb.id = COMPLETION then
        add s to S_completion
        remove s from S_all
        recur
    else
        remove s from S_all
        recur
    end
end
while S_completion is not empty do
    for each statement sc in S_completion
    if sc.result.success = SUCCESS then
        add sc to S_success
        remove sc from S_completion
        recur
    else
        remove sc from S_completion
        recur
    end
end
while S_success is not empty do
    for each statement ss in S_success
        let ss.result.score.raw = raw?
            ss.result.score.max = max?
            ss.result.score.min = min?
            scaled = scale(raw?, min?, max?)
         concat coll [ss.timestamp, scaled]
         remove ss from S_success
         recur
end
```

- The Z schemas are used within this pseudocode
- The return value coll is an array of arrays, each containing a timestamp and a scaled score.

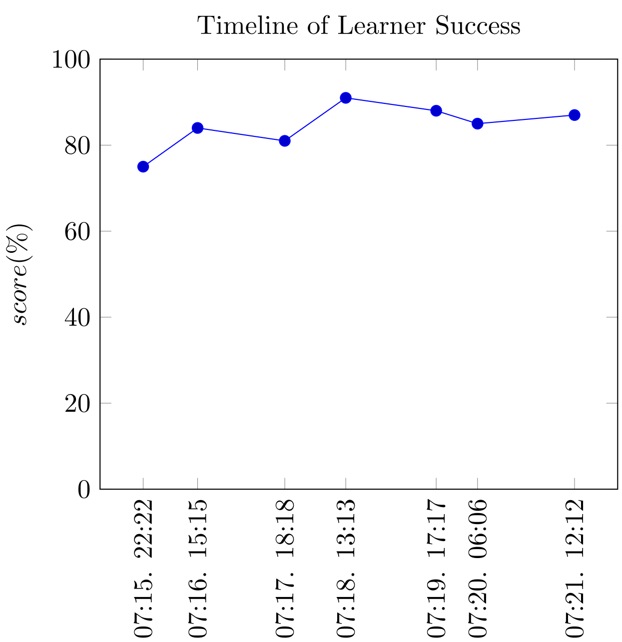
2.8 JSON Schema

```
{"type":"array",
   "items":{"type":"array",
      "items":[{"type":"string"}, {"type":"number"}]}
}
```

2.9 Visualization Description

The Timeline of Learner Success visualization will be a line chart where the domain is time and the range is score on a scale of 0.0 to 100.0. Every array within the array returned by the algorithm will be a point on the chart. The domain of the graph should be in chronological order.

2.10 Visualization prototype



2.11 Prototype Improvement Suggestions

Aditional features may be implemented on top of this base specification but would require adding aditional values to each sub-array returned by the algorithm. These values would only be limtied by the fields contained within the xAPI statements being used to populate the visualization. Examples of fields which could be used to populate the visualization include, but are not limited to:

- A tooltip containing the name of an activity when hovering over a specific point on the chart
  - this would require adding the value of $\$.object.definition.name$ to each subarray
- A tooltip containing the device on which the activity was experienced
  - this would require adding the value from $\$.context.platform$ to each subarray
- A tooltip containing the instructor associated with a particular data point
  - this would require adding the value from $\$.context.instructor$ to each subarray

3 Which Assessment Questions are the Most Difficult

As learners engage in a blended eLearning ecosystem, they will experiecne teaching content as well as assessment content. Assessments are designed to measure the effectiveness of learning content and help measure learning. It is possible that certain assessment questions do not accurately represent the concepts contained within teaching material and this may be indicated by a majority of learners getting the question wrong. It is also possible that the question accurately represents the learning content but is just hard. The following algorithm will identify these questions but will not be able to deduce why learners answer them incorrectly.

3.1 Ideal Statements

In order to accurately determine which assessment questions are the most dificult, there are a few requirements of the data produced by a LRP. They are as follows:

- statements describing a learner answering a question must report if the learner got the question correct or incorrect via $\$.result.success$
- if it is possible to get partial credit on a question, the amount of credit should be reported within the statement
  - the credit earned by the learner should be reported within $\$.result.score.raw$
  - the minimum and maximum possible credit amount should be reported within $\$.result.score.min$ and $\$.result.score.max$ respectively
- If it is possible to get partial credit on a question, it must still be reported if the learner reached the threshold of success via $\$.result.success$

- Statements describing a learner answering a question should contain activities of the type *cmi.interaction*
- activities must be uniquely and consistently identified across all statements
- Statements describing a learner answering a question should[4] use the verb *http://adlnet.gov/expapi/verbs/answered*

3.2 Input Data Retrieval

How to query an LRS via a GET request to the Statements Resource [5] [6] [7]

```
Verb = "verb=http://adlnet.gov/expapi/verbs/answered"

Since = "since=2018-07-20T12:08:47Z"

Until = "until=2018-07-21T12:08:47Z"

Base = "https://example.endpoint/statements?"

endpoint = Base + Verb + "&" + Since + "&" + Until

Auth = Hash generated from basic auth

S = curl -X GET -H "Authorization: Auth"
         -H "Content-Type: application/json"
         -H "X-Experience-API-Version: 1.0.3"
         Endpoint
```

3.3 Statement Parameters to Utilize

The statement parameter locations here are written in JSONPath. This notation is also compatable with the xAPI Z notation

- $\$.result.success$
- $\$.object.id$

3.4 2018 Pilot TLA Statement Problems

The initial pilot test data supports this algorithm. Given that the offical 2018 pilot test is scheduled to take place on July 27th, 2018, this section may require updates pending data review.

[4] it is possible to use another verb but if another is used, that will need to be accounted for in data retrieval

[5] See footnote 1.

[6] See footnote 2.

[7] See footnote 3.

3.5 Summary

1. Query an LRS via a GET request to the statements endpoint using the parameters verb, since and until

2. Filter the results to the set of statements where:

    - $\$.result.success$ is false

3. process the filtered data

    - group by activity ID
    - determine the count of each group
    - create a collection of pairs = [activity ID, count]

3.6 Formal Specification

3.6.1 Basic Types

$INCORRECT ::= \{false\}$

3.6.2 System State

$$
\begin{array}{|l}
\hline
MostDifficultAssessmentQuestions \\
Statements \\
S_{all} : \mathbb{F}_1 \\
S_{incorrect}, S_{grouped}, S_{processed} : \mathbb{F} \\
\hline
S_{all} = statements \\
S_{incorrect} \subseteq S_{all} \\
S_{grouped} = \{groups : \mathrm{seq}_1\ statement\} \\
S_{processed} = \{pair : (id, \mathbb{N})\} \\
\hline
\end{array}
$$

- The set S_{all} is a non-empty finite set and is the component *statements*
- The sets $S_{incorrect}$, $S_{grouped}$ and $S_{processed}$ are all finite sets
- the set $S_{incorrect}$ is a subset of S_{all}
- the set $S_{grouped}$ is a finite set of objects *groups* which are non-empty finite sequences of the component *statement*
- the set $S_{processed}$ is a finite set of pairs where each contains the component *id* and a natural number

3.6.3 Initial System State

$$
\begin{array}{|l}
\hline
\textit{InitMostDifficultAssessmentQuestions} \\
\textit{MostDifficultAssessmentQuestions} \\
\hline
S_{all} \neq \emptyset \\
S_{incorrect} = \emptyset \\
S_{grouped} = \emptyset \\
S_{processed} = \emptyset \\
\hline
\end{array}
$$

- The set S_{all} is a non-empty set
- The sets $S_{incorrect}$, $S_{grouped}$ and $S_{processed}$ are all initially empty

3.6.4 Filter for Incorrect

$$
\begin{array}{|l}
\hline
\textit{Incorrect} \\
\textit{Statement} \\
incorrect : STATEMENT \pfun \mathbb{F} \\
s? : STATEMENT \\
s! : \mathbb{F} \\
\hline
s? = statement \\
s! = incorrect(s?) \\
incorrect(s?) = \textbf{if } s?.result.success : INCORRECT \\
\qquad\qquad\qquad \textbf{then } s! = s? \\
\qquad\qquad\qquad \textbf{else } s! = \emptyset \\
\hline
\end{array}
$$

- the schema $Incorrect$ introduces the function $incorrect$ which takes in the variable $s?$ and returns the variable $s!$
- the variable $s?$ is the component $statement$
- $s!$ is equal to $s?$ if $\$.result.success$ is of the type $INCORRECT$ otherwise $s!$ is an empty set

$$
\begin{array}{|l}
\hline
\textit{FilterForIncorrect} \\
\Delta \textit{MostDifficultAssessmentQuestions} \\
\textit{Incorrect} \\
incorrects : \mathbb{F} \\
\hline
incorrects = S_{all} \\
incorrects' = \{s : STATEMENT \mid incorrect(s) \neq \emptyset\} \\
S'_{incorrect} = S_{incorrect} \cup incorrects' \\
\hline
\end{array}
$$

- the set $incorrects$ is the set S_{all}
- The set $incorrects'$ contains elements s of type $STATEMENT$ where $incorrect(s)$ is not an empty set

- The updated set $S'_{incorrect}$ is the union of the previous state of $S_{incorrect}$ and $incorrects'$

3.6.5 Processes Results

$$
\begin{array}{l}
\textit{GroupByActivityId} \\
\hline
\textit{Statements} \\
g? : \mathbb{F} \\
g! : \mathbb{F} \\
group : \mathbb{F} \rightarrow \mathbb{F} \\
\hline
g? = statements \Rightarrow \{g : statement\} \\
g! = group(g?) \\
g! = \{groups : \mathrm{seq}_1\, statement \mid \\
\quad \textbf{let } \mathrm{seq}_1\, statement_i..statement_j == \mathrm{seq}_1\, s_i..s_j \bullet \\
\quad \forall i, j : s_i..s_j \bullet s_i.object.id = s_j.object.id\} \\
\hline
\end{array}
$$

- The schema $GroupByActivityId$ introduces the function $group$ which has the input of $g?$ and the output of $g!$
- The input variable $g?$ is the component $statements$ which implies its a set of objects g which are each a $statement$
- the output variable $g!$ is a set of objects $groups$ which are each a non-empty finite sequence of $statement$ where each member of the sequence $s_i..s_j$ has identical object ids.

$$
\begin{array}{l}
\textit{CountPerGroup} \\
\hline
\textit{Statement} \\
c? : \mathrm{seq}_1\, statement \\
c! : \mathbb{N} \\
count : \mathrm{seq}_1\, statement \rightarrow \mathbb{N} \\
\hline
c! = count(c?) \\
c! \geq 1 \\
count(c?) = \forall i, j : \mathrm{seq}_1\, c?_i..c?_j \bullet i, j : \mathbb{N} \wedge i = 0 \wedge i = j \vee i \neq j \bullet \\
\qquad \exists_1\, c! : \mathbb{N} \bullet c! = j + 1 \\
\hline
\end{array}
$$

- The schema $CountPerGroup$ introduces the function $count$ which has the input of $c?$ and the output of $c!$
- The input variable $c?$ is a non-empty finite sequence in which each element is a $statement$
- The function $count$ reads: for all elements within the sequence $c?_i..c?_j$, i and j are natural numbers, i is equal to zero and may or may not be equal to j such that there exits a number $c!$ which is equal to $j+1$

$$
\begin{array}{l}
\underline{\quad AggregateQuestionStatements \quad} \\
\Delta MostDifficultAssessmentQuestions \\
FilterForIncorrect \\
GroupByActivityId \\
CountPerGroup \\
grouped, processed : \mathbb{F} \\
\hline
grouped = \emptyset \\
grouped' = group(S_{incorrect}) \\
S'_{grouped} = S_{grouped} \cup grouped' \\
processed = S'_{grouped} \\
processed' = \{p : (id, \mathbb{N}) \mid \\
\qquad \textbf{let } \{processed_i..processed_j\} == \{g_i..g_j\} \bullet \\
\qquad \forall i, j : g_i..g_j \bullet \exists\ p_i..p_j \bullet \\
\qquad first\, p_i = head\, g_i.object.id \wedge second\, p_i = count(g_i) \\
\qquad first\, p_j = headg_j.object.id \wedge second\, p_j = count(g_j)\} \\
S'_{processed} = S_{processed} \cup processed'
\end{array}
$$

- The schema *AggregateQuestionStatements* introduces the variables *grouped* and *processed*
- *grouped* starts as an empty set but then becomes $grouped'$ which is the output of applying the function *group* to the set of statements $S_{incorrect}$ created by the opperation *FilterForIncorrect*
- $grouped'$ is a set of sequences. The elements of those sequences are statements which all have the same *statement.object.id*
- The set $S_{grouped}$ is updated to the set $S'_{grouped}$ which is the union of $S_{grouped}$ and $grouped'$
- the variable *processed* is initialized to be $S'_{grouped}$
- the variable *processed* is updated to be the variable $processed'$ which is a set of objects p which are ordered pairs of the component *id* and a natural number. p is defined as:
  - for all sequences $g_i..g_j$ within the set *processed*, there exists ordered pairs $p_i..p_j$ such that:
    * the first element of p_i is equal to the *object.id* of the first statement within the sequence g_i.
    * The second element of p_i is equal to the value returned when g_i is passed to the function *count*.
    * The first element of p_j is equal to the *object.id* of the first statement within the sequence g_j.
    * the second element of p_j is equal to the value returned when g_j is passed to the function *count*
- The set $S'_{processed}$ is the union of the sets $S_{processed}$ and $processed'$

3.6.6 Sequence of Operations

$ProcessedQuestions \mathrel{\widehat=} FilterForIncorrect \mathbin{⨟} AggregateQuestionStatements$

- The schema $ProcessedQuestions$ is the sequential composition of operation schemas $FilterForIncorrect$ and $AggregateQuestionStatements$

- $FilterForIncorrect$ happens before $AggregateQuestionStatements$

3.6.7 Return

$$
\begin{array}{|l}
\hline
\textit{ReturnAggregate} \\
\Xi MostDifficultAssessmentQuestions \\
ProcessedQuestions \\
S_{processed}! : \mathbb{F} \\
\hline
S_{processed}! = S_{processed} \\
\hline
\end{array}
$$

- The returned variable $S_{processed}!$ is equal to the current state of variable $S_{processed}$ after the operations $FilterForIncorrect$ and $AggregateQuestionStatements$

3.7 Pseudocode

Algorithm 2: Most Difficult Assessment Questions

```
Input: S_all, display-n
Result: display
init id-to-count := []
init display := []
while S_all is not empty do
    for each statement s in S_all
    if s.result.success = INCORRECT then
        add s to S_incorrect
        remove s from S_all
        recur
    else
        remove s from S_all
        recur
    end
end
while S_incorrect is not empty do
    for each statement si in S_incorrect
        let si.object.id = id
    if id-to-count is empty then
        concat id-to-count [id , 1]
        remove si from S_incorrect
        recur;
    else
        if id-to-count contains [id , #] then
            add one to #
            remove si from S_incorrect
            recur
        else
            concat id-to-count [id , 1]
            remove si from S_incorrect
            recur
        end
    end
end
Sort id-to-count by second value of each subarray (#)
take first display-n subarrays from id-to-count and concat them into
  display
```

- The Z schemas are used within this pseudocode
- The return value display is an array of display-n arrays, each containing an object id and a number representing the number of times it showed up.

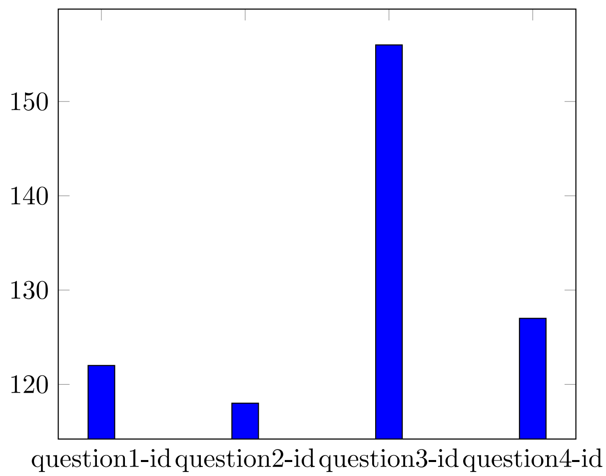
3.8 JSON Schema

```
{"type":"array",
   "items":{"type":"array",
      "items":[{"type":"string"}, {"type":"number"}]}
}
```

3.9 Visualization Description

The Most Difficult Assessment Questions visualization will be a bar chart where the domain is statement id and the range is a number. Every subarray within the array returned by the algorithm will contribute to the bar chart. The pseudocode specifies an input paramter display-n which controls how many activitys will be displayed within the visualization.

3.10 Visualization prototype



3.11 Prototype Improvement Suggestions

Additional features may be implemented on top of this base specification but would require adding aditional values to each sub array returned by the algorithm or performing metadata lookup. The following examples assume the metadata is contained within each statement.

- Use the name of the activity for the x-axis label instead of its id.
  - $\$.object.definition.name$
  - grouping of statements should still happen by $\$.object.id$ to ensure an accurate count
- a tooltip containing contextual information about the question such as:
  - The question text

    - $\$.object.definition.description$
  - Interaction Type
    - $\$.object.definition$ which specifies interaction properties
  - Answer choices
    - $\$.object.definition$ which specifies interaction properties
  - Correct answer
    - $\$.object.definition$ which specifies interaction properties
  - Most popular incorrect answer
    - This would require an extra step of processing to be added to the algorithm and that all statements utilize interaction properties within $\$.object.definition$
  - average partial credit earned (if applicable)
    - $\$.result.score.scaled$
    - The one potential issue with using scaled score is the calculation of scaled is not stricly defined by the xAPI specification and thus the average scaled score may not be comparable across questions.
    - if $\$.result.score.raw$, $\$.result.score.min$ and $\$.result.score.max$ are reported for all questions, it becomes possible to reliably compare across questions.
  - average number of re-attempts
    - this would require additional steps of processing so that $\$.actor$ is considered as well
    - due to the problem of actor unification, ie the same person being identified differently across statements, this metric may not be accurate
  - average time spent on the question
    - $\$.result.duration$
    - this would require additional steps of processing to extract the duration and average it.

- a tooltip containing contextual information about the course and/or assessment the question was within
  - the instructor for the course
    - $\$.context.instructor$
  - competency associated with the question
    - $\$.context.contextActivities$
  - metadata about the learning content associated with the question such as average time spent engaging with associated learning content before attempting the question
  - this would require additional steps of processing to retrieve metadata about the content and its usage
    - $\$.context.contextActivities$