

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 refinement 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 is 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 an xAPI data point

$[AGENT, GROUP]$

- Basic types for Agents and collections of Agents

1.2 Id Schema

Id
$id : \mathbb{F}_1 \#1$

- 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

<i>Agent</i>	
<i>agent</i> : <i>AGENT</i>	
<i>objectType</i> : <i>OBJECTTYPE</i>	
<i>name</i> : $\mathbb{F}_1 \#1$	
<i>ifi</i> : <i>IFI</i>	
<i>objectType</i> = <i>Agent</i>	
<i>agent</i> = $\{ifi\} \cup \mathbb{P}\{name, objectType\}$	

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

<i>Member</i>	
<i>Agent</i>	
<i>member</i> : \mathbb{F}_1	
<i>member</i> = $\{a : AGENT \mid \forall a : a_0..a_n \bullet a = agent\}$	

- 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*

<i>Group</i>	
<i>Member</i>	
<i>group</i> : <i>GROUP</i>	
<i>objectType</i> : <i>OBJECTTYPE</i>	
<i>ifi</i> : <i>IFI</i>	
<i>name</i> : $\mathbb{F}_1 \#1$	
<i>objectType</i> = <i>Group</i>	
<i>group</i> = $\{objectType, name, member\} \vee \{objectType, member\} \vee \{objectType, ifi\} \cup \mathbb{P}\{name, member\}$	

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

<i>Actor</i>	
<i>Agent</i>	
<i>Group</i>	
<i>actor</i> : <i>AGENT</i> \vee <i>GROUP</i>	
<i>actor</i> = <i>agent</i> \vee <i>group</i>	

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

1.4 Verb Schema

<i>Verb</i>	_____
<i>Id</i>	
<i>display, verb</i> : \mathbb{F}_1	
<i>verb</i> = $\{id, display\} \vee \{id\}$	

- 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

<i>Extensions</i>	_____
<i>extensions, extensionVal</i> : \mathbb{F}_1	
<i>extensionId</i> : $\mathbb{F}_1 \#1$	
<i>extensions</i> = $\{e : (extensionId, extensionVal) \mid \forall e : e_i..e_j \bullet$ $(extensionId_i, extensionVal_i) \vee (extensionId_i, extensionVal_j) \wedge$ $(extensionId_j, extensionVal_i) \vee (extensionId_j, extensionVal_j) \wedge$ $extensionId_i \neq extensionId_j\}$	

- The schema *Extensions* introduces the component *extensions* which is a non-empty, finite set that consists of ordered pairs of *extensionId* and *extensionVal*. Different *extensionIds* 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

<i>InteractionActivity</i>	_____
<i>interactionType</i> : <i>INTERACTIONTYPE</i>	
<i>correctResponsePattern</i> : seq_1	
<i>interactionComponent</i> : <i>INTERACTIONCOMPONENT</i>	
<i>interactionActivity</i> = $\{interactionType, correctReponsePattern, interactionComponent\} \vee$ $\{interactionType, correctResponsePattern\}$	

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

<i>Definition</i>
<i>InteractionActivity</i>
<i>Extensions</i>
<i>definition, name, description</i> : \mathbb{F}_1
<i>type, moreInfo</i> : $\mathbb{F}_1 \#1$
<i>definition</i> = $\mathbb{P}_1\{name, description, type, moreInfo, extensions, interactionActivity\}$

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

<i>Object</i>
<i>Id</i>
<i>Definition</i>
<i>Agent</i>
<i>Group</i>
<i>Statement</i>
<i>objectTypeA, objectTypeS, objectTypeSub, objectType</i> : <i>OBJECTTYPE</i>
<i>substatement</i> : <i>STATEMENT</i>
<i>object</i> : \mathbb{F}_1
<i>substatement</i> = <i>statement</i>
<i>objectTypeA</i> = <i>Activity</i>
<i>objectTypeS</i> = <i>StatementRef</i>
<i>objectTypeSub</i> = <i>SubStatement</i>
<i>objectType</i> = <i>objectTypeA</i> \vee <i>objectTypeS</i>
<i>object</i> = $\{id\} \vee \{id, objectType\} \vee \{id, objectTypeA, definition\}$ $\vee \{id, definition\} \vee \{agent\} \vee \{group\} \vee \{objectTypeSub, substatement\}$ $\vee \{id, objectTypeA\}$

- The schema *Object* introduces the component *object* which is a non-empty, finite set of either *id*, *id* and *objectType*, *id* and *objectTypeA*, *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

<i>Score</i>
<i>score</i> : \mathbb{F}_1 <i>scaled, min, max, raw</i> : \mathbb{Z}
<i>scaled</i> = $\{n : \mathbb{Z} \mid -1.0 \leq n \leq 1.0\}$ <i>min</i> = $n < \text{max}$ <i>max</i> = $n > \text{min}$ <i>raw</i> = $\{n : \mathbb{Z} \mid \text{min} \leq n \leq \text{max}\}$ <i>score</i> = $\mathbb{P}_1\{\text{scaled}, \text{raw}, \text{min}, \text{max}\}$

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

<i>Result</i>
<i>Score</i> <i>Extensions</i> <i>success, completion, response, duration</i> : $\mathbb{F}_1 \# 1$ <i>result</i> : \mathbb{F}_1
<i>success</i> = $\{\text{true}\} \vee \{\text{false}\}$ <i>completion</i> = $\{\text{true}\} \vee \{\text{false}\}$ <i>result</i> = $\mathbb{P}_1\{\text{score}, \text{success}, \text{completion}, \text{response}, \text{duration}, \text{extensions}\}$

- 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

<i>Instructor</i>
<i>Agent</i> <i>Group</i> <i>instructor</i> : $AGENT \vee GROUP$
<i>instructor</i> = $agent \vee group$

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

<i>Team</i>
<i>Group</i> <i>team</i> : $GROUP$
<i>team</i> = $group$

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

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

- The schema *Context* introduces the component *context* which is the non-empty powerset of *registration*, *instructor*, *team*, *contextActivities*, *revision*, *platform*, *language*, *statement* and *extensions*
- The notation $object \setminus agent$ represents the component *object* except for its subcomponent *agent*

1.8 Timestamp and Stored Schema

<i>Timestamp</i> $timestamp : \mathbb{F}_1 \#1$
<i>Stored</i> $stored : \mathbb{F}_1 \#1$

- 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

<i>Attachments</i>	
<i>display, description, attachment, attachments</i> : \mathbb{F}_1	
<i>usageType, sha2, fileUrl, contextType</i> : $\mathbb{F}_1 \#1$	
<i>length</i> : \mathbb{N}	
<i>attachment</i> = { <i>usageType, display, contentType, length, sha2</i> } $\cup \mathbb{P}\{description, fileUrl\}$	
<i>attachments</i> = { <i>a</i> : <i>attachment</i> }	

- The schema *Attachments* introduces the component *attachments* which is a non-empty, finite set of the component *attachment*
- The component *attachment* is a non-empty, finite set of the component *usageType, display, contentType, length, sha2* with optionally *description* and/or *fileUrl*

1.10 Statement and Statements Schema

<i>Statement</i>	
<i>Id</i>	
<i>Actor</i>	
<i>Verb</i>	
<i>Object</i>	
<i>Result</i>	
<i>Context</i>	
<i>Timestamp</i>	
<i>Stored</i>	
<i>Attachments</i>	
<i>statement</i> : <i>STATEMENT</i>	
<i>statement</i> = { <i>actor, verb, object, stored</i> } \cup $\mathbb{P}\{id, result, context, timestamp, attachments\}$	

- 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 be referenced via the . (selection) operator

<i>Statements</i>
<i>Statement</i>
<i>statements</i> : \mathbb{F}_1
<i>statements</i> = { <i>s</i> : <i>statement</i> }

- The schema *Statements* introduces the component *statements* which is a non-empty, finite set of the component *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 base 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 via curl. The following section contains the appropriate parameters with example values

as well as the curl command necessary for making the request.^{1 2 3}

```
Agent = "agent={\"account\":  
              {\"homePage\": \"https://example.homepage\",  
                \"name\": 123456}}\"  
  
Since = \"since=2018-07-20T12:08:47Z\"  
  
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 compatible with the xAPI Z notation due to the defined hierarchy of components. Within the Z specifications, a variable name will be used instead of the \$

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

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 utilized

³ If you want to query across the entire history of a LRS, omit Since and Until from the endpoint and remove the associated & symbols.

2.4 2018 Pilot TLA Statement Problems

The initial pilot test data supports this algorithm. Given that the official 2018 pilot test is scheduled to take place on July 27th, 2018, this section may require updates pending future 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 $\$.timestamp$
 - extract the score values from $\$.result.score.raw$, $\$.result.score.min$ and $\$.result.score.max$ and convert them to the scale 0..100
 - create a pair of $[\$.timestamp, \#]$

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

$TimelineLearnerSuccess$	_____
$Statements$	
$S_{all} : \mathbb{F}_1$	
$S_{completion}, S_{success}, S_{processed} : \mathbb{F}$	
$S_{all} = statements$	
$S_{completion} \subseteq S_{all}$	
$S_{success} \subseteq S_{completion}$	
$S_{processed} = \{pair : (statement.timestamp, \mathbb{N})\}$	

- 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} which may contain every value within S_{all}
- the set $S_{success}$ is a subset of $S_{completion}$ which may contain every value within $S_{completion}$
- 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

<i>InitTimelineLearnerSuccess</i>	_____
<i>TimelineLearnerSuccess</i>	
$S_{all} \neq \emptyset$	
$S_{completion} = \emptyset$	
$S_{success} = \emptyset$	
$S_{processed} = \emptyset$	

- 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

<i>Completion</i>	_____
<i>Statement</i>	
$completion : STATEMENT \rightarrow \mathbb{F}$	
$s? : STATEMENT$	
$s! : \mathbb{F}$	
$s? = statement$	
$s! = completion(s?)$	
$completion(s?) = \text{if } s?.verb.id : COMPLETION$	
$\quad \text{then } s! = s?$	
$\quad \text{else } s! = \emptyset$	

- The schema *Completion* introduces 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 $s?.verb.id$ is of the type *COMPLETION* otherwise $s!$ is an empty set

FilterForCompletion

$\Delta TimelineLearnerSuccess$

Completion

$completions : \mathbb{F}$

$completions \subseteq S_{all}$

$completions' = \{s : STATEMENT \mid completion(s) \neq \emptyset\}$

$S'_{completion} = S_{completion} \cup completions'$

- the set *completions* is a subset of S_{all} which may contain every value within 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

Success

Statement

$success : STATEMENT \rightarrow \mathbb{F}$

$s? : STATEMENT$

$s! : \mathbb{F}$

$s? = statement$

$s! = success(s?)$

$success(s?) = \text{if } s?.result.success : SUCCESS$

then $s! = s?$

else $s! = \emptyset$

- 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 $s?.result.success$ is of the type *SUCCESS* otherwise $s!$ is an empty set

FilterForSuccess

$\Delta TimelineLearnerSuccess$

Success

$successes : \mathbb{F}$

$successes \subseteq S_{completion}$

$successes' = \{s : STATEMENT \mid success(s) \neq \emptyset\}$

$S'_{success} = S_{success} \cup successes'$

- the set $successes$ is a subset of $S_{completion}$ which may contain every value within $S_{completion}$
- 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

$Scale$ $scaled! : \mathbb{N}$ $raw?, min?, max? : \mathbb{Z}$ $scale : \mathbb{Z} \rightarrow \mathbb{N}$
$scaled! = scale(raw?, min?, max?)$ $scale(raw?, min?, max?) =$ $(raw? * ((0.0 - 100.0) div (min? - max?))) +$ $(0.0 - (min? * ((0.0 - 100.0) div (min? - max?))))$

- 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 $0.0..100.0$

$ProcessStatements$ $\Delta TimelineLearnerSuccess$ $Scale$ $FilterStatements$ $processed : \mathbb{F}$
$processed \subseteq S_{success}$ $processed' = \{p : (\mathbb{F}_1 \# 1, \mathbb{N}) \mid$ $\quad \text{let } \{processed_i..processed_j\} == \{s_i..s_j\} \bullet$ $\quad \forall s : s_i..s_j \bullet \exists p_i..p_j \bullet$ $\quad first\ p_i = s_i.timestamp \wedge$ $\quad second\ p_i = scale(s_i.result.score.raw,$ $\quad \quad \quad s_i.result.score.min,$ $\quad \quad \quad s_i.result.score.max) \wedge$ $\quad first\ p_j = s_j.timestamp \wedge$ $\quad second\ p_j = scale(s_j.result.score.raw,$ $\quad \quad \quad s_j.result.score.min,$ $\quad \quad \quad s_j.result.score.max)\}$ $S'_{processed} = S_{processed} \cup processed'$

- The operation $ProcessStatements$ introduces the variable $processed$ which is a subset of $S_{success}$ which may contain every value within $S_{success}$

- $S_{success}$ is the result of the operation $FilterStatements$
- The operation defines the variable $processed'$ which is a set of objects p which are ordered pairs 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 arguments 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 update the set $S_{processed}$ with the values contained within $processed'$

2.6.7 Sequence of Operations

$FilterStatements \hat{=} FilterForCompletion \circ FilterForSuccess$

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

$ProcessedStatements \hat{=} FilterStatements \circ ProcessStatements$

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

2.6.8 Return

$Return$ $\exists TimelineLearnerSuccess$ $ProcessedStatements$ $S_{processed}! : \mathbb{F}$
$S_{processed}! = S_{processed}$

- 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
end
```

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

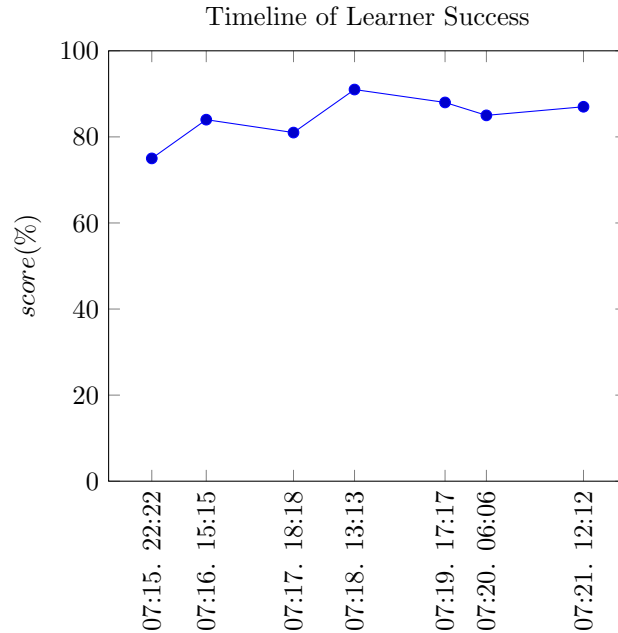
2.8 JSON Schema

```
{ "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 subarray 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

Additional features may be implemented on top of this base specification but they would require adding additional values to each subarray returned by the algorithm. These additional values can be retrieved via (1) performing metadata lookup within or independently of the algorithm (2) by utilizing additional xAPI statement parameters and/or (3) by performing additional computations. The following examples assume the metadata is contained within each statement available to the algorithm.

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

3 Which Assessment Questions are the Most Difficult

As learners engage in a blended eLearning ecosystem, they will experience learning content as well as assessment content. Assessments are designed to measure the effectiveness of learning content and help assess knowledge gained. It is possible that certain assessment questions do not accurately represent the concepts contained within learning content 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 very difficult. The following algorithm will identify these types of 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 difficult, 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 should⁴ 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 via curl. The following section contains the appropriate parameters with example values as well as the curl command necessary for making the request.^{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 compatible with the xAPI Z notation due to the defined hierarchy of components. Within the Z specifications, a variable name will be used instead of the \$

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

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.

3.4 2018 Pilot TLA Statement Problems

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

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 $\$.object.id$
 - determine the count of each group
 - create a collection of pairs = $[\$.object.id, \#]$

3.6 Formal Specification

3.6.1 Basic Types

$INCORRECT ::= \{false\}$

3.6.2 System State

$MostDifficultAssessmentQuestions$
$Statements$
$S_{all} : \mathbb{F}_1$
$S_{incorrect}, S_{grouped}, S_{processed} : \mathbb{F}$
$S_{all} = statements$
$S_{incorrect} \subseteq S_{all}$
$S_{grouped} = \{groups : seq_1 statement\}$
$S_{processed} = \{pair : (id, \mathbb{N})\}$

- 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} which may contain every value within 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

<i>InitMostDifficultAssessmentQuestions</i>	_____
<i>MostDifficultAssessmentQuestions</i>	
$S_{all} \neq \emptyset$	
$S_{incorrect} = \emptyset$	
$S_{grouped} = \emptyset$	
$S_{processed} = \emptyset$	

- 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

<i>Incorrect</i>	_____
<i>Statement</i>	
$incorrect : STATEMENT \rightarrow \mathbb{F}$	
$s? : STATEMENT$	
$s! : \mathbb{F}$	
$s? = statement$	
$s! = incorrect(s?)$	
$incorrect(s?) = \text{if } s?.result.success : INCORRECT$	
$\quad \text{then } s! = s?$	
$\quad \text{else } s! = \emptyset$	

- 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 $s?.result.success$ is of the type *INCORRECT* otherwise $s!$ is an empty set

<i>FilterForIncorrect</i>	_____
$\Delta MostDifficultAssessmentQuestions$	
<i>Incorrect</i>	
$incorrects : \mathbb{F}$	
$incorrects \subseteq S_{all}$	
$incorrects' = \{s : STATEMENT \mid incorrect(s) \neq \emptyset\}$	
$S'_{incorrect} = S_{incorrect} \cup incorrects'$	

- the set *incorrects* is a subset of S_{all} which may contain every value within 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

$GroupByActivityId$	_____
$Statements$ $g? : \mathbb{F}$ $g! : \mathbb{F}$ $group : \mathbb{F} \rightarrow \mathbb{F}$	
$g? = statements \Rightarrow \{g : statement\}$ $g! = group(g?)$ $g! = \{groups : seq_1 statement \mid$ $\quad \text{let } seq_1 statement_i..statement_j == seq_1 s_i..s_j \bullet$ $\quad \forall s : s_i..s_j \bullet s_i.object.id = s_j.object.id\}$	

- 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 the same $object.id$

$CountPerGroup$	_____
$Statement$ $c? : seq_1 statement$ $c! : \mathbb{N}$ $count : seq_1 statement \rightarrow \mathbb{N}$	
$c! = count(c?)$ $c! \geq 1$ $count(c?) = \forall c? : c?_i..c?_j \bullet i, j : \mathbb{N} \wedge i = 0 \wedge i = j \vee i \neq j \bullet$ $\quad \exists_1 c! : \mathbb{N} \bullet c! = j + 1$	

- 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 exists a number $c!$ which is equal to $j + 1$

<i>AggregateQuestionStatements</i> Δ <i>MostDifficultAssessmentQuestions</i> <i>FilterForIncorrect</i> <i>GroupByActivityId</i> <i>CountPerGroup</i> <i>grouped, processed</i> : \mathbb{F}
$grouped = \emptyset$ $grouped' = group(S_{incorrect})$ $S'_{grouped} = S_{grouped} \cup grouped'$ $processed \subseteq S'_{grouped}$ $processed' = \{p : (id, \mathbb{N}) \mid$ $\text{let } \{processed_i..processed_j\} == \{g_i..g_j\} \bullet$ $\forall g : g_i..g_j \bullet \exists p_i..p_j \bullet$ $first\ p_i = head\ g_i.object.id \wedge second\ p_i = count(g_i)$ $first\ p_j = head\ g_j.object.id \wedge second\ p_j = count(g_j)\}$ $S'_{processed} = S_{processed} \cup processed'$

- 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 operation *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 a subset of $S'_{grouped}$ which can contain every value within $S'_{grouped}$
- 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

$ProcessedQuestions \hat{=} FilterForIncorrect \circ AggregateQuestionStatements$

- The schema *ProcessedQuestions* is the sequential composition of operation schemas *FilterForIncorrect* and *AggregateQuestionStatements*
- *FilterForIncorrect* happens before *AggregateQuestionStatements*

3.6.7 Return

$ReturnAggregate$ $\exists MostDifficultAssessmentQuestions$ $ProcessedQuestions$ $S_{processed}! : \mathbb{F}$	
$S_{processed}! = S_{processed}$	

- 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  
end  
Sort id-to-count by second value of each subarray (#)  
take the 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 length display-n, where each element of display is an array of [$statement.object.id$, #] where # representing the number of times $statement.object.id$ appeared within $S_{incorrect}$

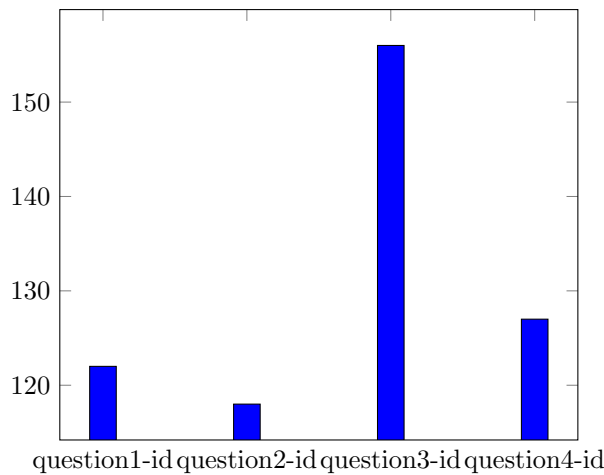
3.8 JSON Schema

```
{ "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 consists of *statement.object.id* and the range is a number greater than or equal to 1. Every subarray within the array display will be a grouping within the bar chart. The pseudocode specifies an input parameter *display-n* which controls the length of the array display and therefore the number of groups contained within the visualization.

3.10 Visualization prototype



3.11 Prototype Improvement Suggestions

Additional features may be implemented on top of this base specification but they would require adding additional values to each subarray returned by the algorithm. These additional values can be retrieved via (1) performing metadata lookup within or independently of the algorithm (2) by utilizing additional xAPI statement parameters and/or (3) by performing additional computations. The following examples assume the metadata is contained within each statement available to the algorithm.

- 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 contains interaction properties
 - Answer choices
 - * *\$.object.definition* which contains interaction properties
 - Correct answer
 - * *\$.object.definition* which contains interaction properties
 - Most popular incorrect answer
 - * This would require an extra step of processing and all statements would need to 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 strictly defined by the xAPI specification but is instead up to the authors of the LRP. This results in the inability to reliably compare scaled scores across LRPs.
 - * if *\$.result.score.raw* , *\$.result.score.min* and *\$.result.score.max* are reported for all questions, it becomes possible to reliably compare scores across questions and LRPs.
 - 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 and/or course
 - * *\$.context.contextActivities*
- metadata about the learning content associated with the question such as average time spent engaging with associated content before attempting the question.
- this would require additional steps of processing to retrieve metadata about the content and its usage.
 - * *\$.context.contextActivities*