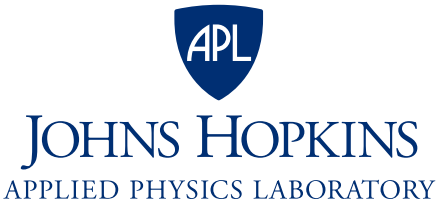

JHU/APL
http://www.jhuapl.edu/

Title:
Lifelong Learning Machines
Metrics Framework &
Proposed Metrics

Project Participants:
JHU/APL Test & Evaluation Team

Authors:
Megan Baker
Gautam Vallabha

Date:
December 5, 2019

Contents

Chapter 1

Introduction

A crucial part of the T&E Framework (TEF) is to log information during the operation of a lifelong learning system and use this to derive *metrics of lifelong learning*, i.e., to characterize how well the system learns along the dimensions (Core Capabilities) set out in the DARPA L2M BAA, such as Continual Learning (CL) and Adaptation to New Tasks (ANT). This is termed the **Metrics Framework**.

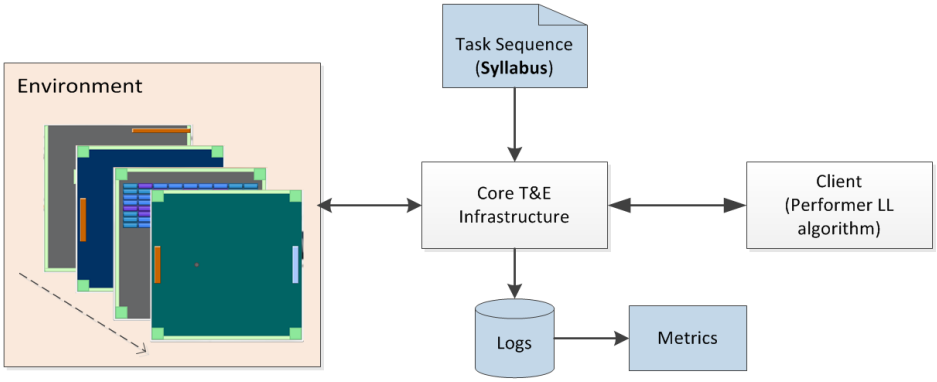

Figure 1.1: Depiction of the Metrics Framework

The Metrics Framework consists of three components:

1. *Performance Logging.* As a learning algorithm goes through a syllabus, Learnkit automatically logs information about the performance of the learning algorithm. For Agent learning, the logged information consists of the total reward for each episode; for classification learning, the logged information consists of the performance error on each batch (the definition of "performance error" is task-specific).

2. *Metrics Calculation.* After the training is completed, the performance logs are read and processed by a separate L2M Python package. This package (the l2metrics repository) consists of a set of predefined metrics, and a Python class hierarchy for extending the predefined set with custom metrics.

3. *Syllabus Annotation.* The metrics calculation assumes that the Learnkit syllabus has specific annotations (e.g., to distinguish training vs. test phases, or different parts of a test phase); relatedly, different syllabi may exercise different aspects of lifelong learning, such as Continual Learning versus Adapting to New Tasks. Consequently, authoring correctly-annotated syllabi is a

crucial component of the Metrics Framework.

The following table summarizes the different ways in which we anticipate performers will engage with the Metrics Framework.

Goal:	Relevant Section:
Generate metrics for a syllabus. Run an L2 algorithm using a predefined syllabus, run the predefined metrics, and view the results	1.2 Sample Workflow to Generate Metrics
Create a custom syllabus. Run an L2 algorithm with a custom syllabus, and run predefined metrics.	3. Creating Custom Syllabi
Create a custom metric. Run an L2 algorithm with a predefined or custom syllabus, and run a custom metric.	3. Creating Custom Syllabi 4. Creating Custom Metrics

1.1 Conceptual Overview

Agent Learning

The T&E Framework uses the information contained in an annotated syllabusto produce a sequence of specific tasks, known as episodes, for an agent learner and automaticallly generates the performance logs from which metrics will be calculated. Figure 1.2 depicts the workflow of the system as described below.

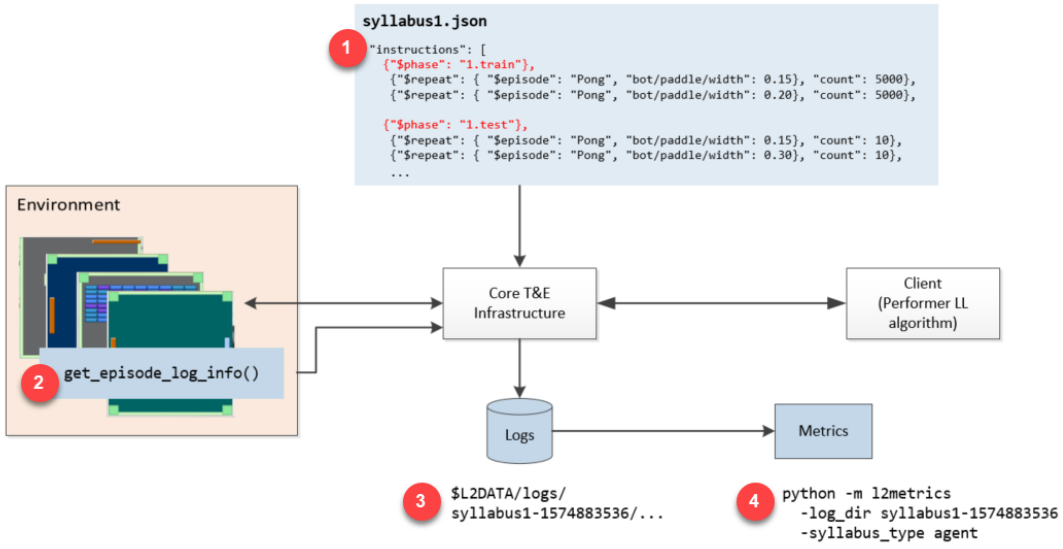

Figure 1.2: Workflow in the Metrics Framework

1. The performance logging starts with an annotated syllabus (see Figure 1.3 for an example). The annotation consists of the "$phase" instructions; the phase values (e.g., "1.train") have a specific structure that is used during the metrics calculation.

2. Tasks in the L2M environment (such as Pong L2Arcade, or L2StarCraft CollectThings) have a special method (`get_episode_log_info`) that reports the total reward and related

```
"instructions": [
  {"$phase": "1.train"},
BLOCK 1 ——{"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.15}, "count": 5000},
BLOCK 2 ——{"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.20}, "count": 5000},

  {"$phase": "1.test"},
    {"$info": {"disable_updates": true} },
BLOCK 3 ——{"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.15}, "count": 10},
BLOCK 4 ——{"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.30}, "count": 10},
    {"$info": {}},

  {"$phase": "2.train"},
BLOCK 5 ——{"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.15}, "count": 5000},
BLOCK 6 ——{"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.20}, "count": 5000},

  {"$phase": "2.test"},
    {"$info": {"disable_updates": true} },
BLOCK 7 ——{"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.15}, "count": 10},
BLOCK 8 ——{"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.30}, "count": 10},
BLOCK 9 ——{"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.15}, "count": 10},
BLOCK 10 ——{"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.30}, "count": 10},
    {"$info": {}}]
```

Figure 1.3: An Annotated Syllabus Example

performance information for each episode. This is automatically called by Learnkit as the syllabus is processed; performer code does not need to do anything (and in particular, the performer algorithm should not call `get_episode_log_info` or use the information reported therein).

3. The performance logs from running the syllabus are automatically saved under the L2M logging directory (the default value of the top-level logging directory is OS-specific, and it can be overridden using the $L2DATA environment variable). The logs for running a specific syllabus are saved under a unique directory name <syllabus_name>-<timestamp>.The `python -m learnkit.info` command shows the location of the directory, along with the most recently created log directory:

```
bakermm1@bakermm1-ll2:~/L2M/l2metrics$ python -m learnkit.info
Learnkit v0.5.0
L2 data folder:
   /home/bakermm1/l2data
L2 logs folder:
   /home/bakermm1/l2data/logs
Most recent log folder:
   /home/bakermm1/l2data/logs/syllabus_CL-1575473425723
   created on 2019-12-04T10:30:25
```

Under $L2DATA/logs/<syllabus_name>-<timestamp>/ the directory tree follows the format: <worker_id>/<phase_name>/<task_name>/block-report, data-log.tsv. A few points are worth noting about this logging structure:

- The worker ID is for distributed training. The ordering of episodes is maintained across workers, so an episode number is unique across the distributed training.
- The syllabus is notionally divided into a sequence of blocks; each block is a sequence of episodes with the same Task name and parameter values. See Figure 1.3 for an example.
- Each episode consists of one or more sub-episodes (each sub-episode is a reset, followed by

one or more steps). This is the granularity for the performance logging – for each sub-episode, the logs record the total reward and whether that sub-episode as complete (i.e., continued until done) or incomplete.

- block-report.tsv contains metadata about each block. data-log.tsv contains the actual logged information about each episode and its sub-episodes. Note that in the logs, "task" and "sub_task" specify the episode and sub-episode number, respectively.

4. Once the learning algorithm is done, the predefined metrics calculations can be invoked by specifying the directory of the log files, e.g.:

```
python -m l2metrics  -log_dir syllabus1-1574883536 -syllabus_type agent
```

Conceptually, the above command loads the log data into a single data table, hands it to each Metric (a Python class), collects the results and prints them out. It is also possible to write a custom script with custom metrics and with full control over which metrics are calculated.

Classification Learning

The logging for classification learning is conceptually similar to that for agent learning. The main difference is in what gets logged and at what granularity (corresponding to Workflow steps 2 and 3 in Figure 1.1).

Recall that a Classification syllabus consists of a sequence of datasets; each dataset consists of one or batches; each batch is a sequence of examples; each example is one input to the learning agent, with an optional output or ground truth value. When performer code interacts a Classification Task (such as CIFAR100), the code looks something like this:

```
for dataset in syllabus.datasets():
       dataset.reset()
       done = False
       while not done:
           batch, done, info = dataset.next_batch()
           y_hat = do_inference(batch['inputs'])  # performer code
           y = dataset.get_labels(batch[id'], y_hat)
           if y:
               calculate_loss_and_update_model(y, y_hat)  # performer code
```

Crucially, the performer code has to submit the estimated outputs `y_hat` via the `get_labels` method; this is done for both training and test data. The Classification Task uses this `y_hat` to calculate the error for the batch, and has a special method `get_batch_log_info` that reports the error information to Learnkit for logging. This method is automatically called by Learnkit as the syllabus is processed; performer code does not need to do anything, and in particular should not invoke `get_batch_log_info`. This information is only intended for Learnkit logging; performers should use their own loss functions as appropriate for their models.

1.2 Sample Workflow to Generate Metrics

The following instuctions illustrate how to operate in the Metrics Framework in a sample environment called Minigrid. Please note that while we have adapted this environment to work with Learnkit, it will not be used for the T&E Framework but rather serves as a quick-to-train example to generate log files in the proper format. After setting up the python packages (steps 1-3), an agent should be trained using the minigridkit script (step 4). Behind the scenes, this script uses Learnkit to load a Continual Learning

syllabus which queues up sequences of episodes which the agent can use to learn. While the agent is being fed these episodes, Learnkit automatically logs the reward achieved at the end of each episode and saves it to TSV files. The location of these files must be passed (step 5) to the l2metrics code, which computes Metrics and prints out the results (step 6).

1. Clone the learnkit repository
   `pip install -e learnkit\`
2. Clone the minigridkit repository
   `pip install -e minigridkit\`
3. Clone the l2metrics repository
   `pip install -e l2metrics\`
4. Train an agent in the Minigrid environment. This will take a few minutes. When it starts you should see an INFO level message showing the top level directory where logs are being written.

   ```
   python minigrid_learnkit/examples/minigrid_train_ppo.py
   2019-12-04 10:30:25,723 <22980>
   [INFO    ] root - Logging to: /home/bakermm1/l2data/logs
   ```

5. Store the last used logging directory in a temporary variable.
   `log_dir =$(ls -t -1 'python -m learnkit.info -get logdir'| head -1)`
6. Pass the variable just set as the `-log_dir` parameter when you run calc_metrics
   ```
   cd l2metrics/examples
   python calc_metrics.py -syllabus_subtype=CL -log_dir=$log_dir
   ```

The output should print to the screen and should look something like this:

```
Metric: Average Within Block Saturation Calculation

Averaged Value: {'global_within_block_saturation': 0.875959925753812,
                 'global_num_eps_to_saturation': 197.45454545454547}

Per Block Values: {
'saturation_value': {0: 0.8488624196510561, 1: 0.822495598845599,
2: 0.8413838383838385, 3: 0.9651983081298474, 4: 0.8791413586413588,
5: 0.8132655122655124, 6: 0.8654549980322708, 7: 0.9549258642961517,
8: 0.9681967251617378, 9: 0.8410522366522368, 10: 0.8355823232323233},

'eps_to_saturation': {0: 185, 1: 30, 2: 81, 3: 107, 4: 610, 5: 83,
6: 84, 7: 100, 8: 739, 9: 103, 10: 50}}

Metric: An Example Custom Metric
Averaged Value: {'global_perf': 0.9217597535934293}
Per Block Values: {'global_perf': 0.9217597535934293}

Process finished with exit code 0
```

Chapter 2

Proposed Metrics

There are a variety of ways a system may demonstrate the BAA's defined Core Capabilities. With these proposed metrics, we hope to capture many dimensions of a system's demonstrated capability. The proposed metrics are framed as an evaluation questions as shown below; the answers to these questions can express the degree to which a system demonstrates the Core Capabilty in question. Please note that some of these metrics are computed as a basis for comparison for future evaluation blocks and are not all used as standalone metrics.

After some combination of these metrics are computed, the agent will receive a score for each syllabus (i.e., an agent's *lifetime*), which will be aggregated to form a score for each Core Capability. For this intial release an average of the scores achieved is reported, but in some cases, more sophisticated methods may replace these in the future.

As was introduced in Figure 1.3, a syllabus is formatted as a JSON file and is divided into a sequence of episodes. Sequences of episodes with the same Task name and parameter values are known as blocks. As shown in the example below, the Task names are "Pong" and "Breakout" and parameter values - encoded here as "bot/paddle/width" - are a range of values: 0.15, 0.20, and 0.30. The syllabus shown below has 10 blocks, and the length of these blocks is encoded as the "count" parameter. Thus, the length of the first two blocks is 5000 episodes, the length of the next two blocks is 10 episodes, the length of the next two blocks is 5000 episodes, and the length of the last four blocks is 10 episodes.

```
"instructions": [
   {"$phase": "1.train"},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.15}, "count": 5000},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.20}, "count": 5000},

   {"$phase": "1.test"},
    {"$info": {"disable_updates": true} },
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.15}, "count": 10},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.30}, "count": 10},
    {"$info": {}},

  {"$phase": "2.train"},
    {"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.15}, "count": 5000},
    {"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.20}, "count": 5000},

   {"$phase": "2.test"},
    {"$info": {"disable_updates": true} },
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.15}, "count": 10},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.30}, "count": 10},
```

```
    {"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.15}, "count": 10},
    {"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.30}, "count": 10},
    {"$info": {}}
]
```

The concept of blocks is critical for establishing computational boundaries within the Metrics Framework. Unless otherwise noted, these Proposed Metrics will be computed within a block, though metrics will often be compared across these boundaries.

2.1 Continual Learning

Proposed Evaluation Metrics

Evaluation Question:	**Relevant Metric:**
What level of performance does the agent achieve during training?	Saturation Value
How quickly does it achieve this saturation value?	Time to Saturation
How quickly does an agent learn during training?	Normalized Integral of Reward over Number Episodes
Can the agent adjust to changes in the environment? How quickly does it recover?	Recovery Time
Can the agent maintain performance on previously learned parameters after being trained with new ones?	Performance Maintenance
How does the lifelong learning agent's performance compare to a traditional agent?	Performance relative to Single Task Expert

Proposed Metric Definitions

1. Saturation Value
Purpose: A saturation value is computed to quantify the maximum maintained reward value by the system, within a block.
Calculated by: Since multiple reward values may be logged per episode (via sub-episodes), the mean reward is taken over all sub-episodes of each episode, giving one aggregate reward value per episode. Then, an unweighted moving average with $\alpha = 11$ is computed over the reward values, returning a smoothed reward signal. Non-overlapping edge values are discarded. The max of the smoothed reward is the system's achieved saturation value.
Compared to: Future or single task expert saturation values of the same task
This Metric is Under Development

Formally:

Given the per episode reward sequence $X_n = x_1, x_2, ..., x_n$, let $S_n = s_1, s_2, ..., s_n$ be the new sequence produced when an unweighted moving average with window size α is applied to X_n. Let α be any odd integer > 0. Thus:

$$s_i = \frac{1}{\alpha} \sum_{j=i}^{i+\alpha-1} x_j$$

$$saturation\_value = max(S_n)$$

2. Time to Saturation
Purpose: Time to saturation is used to quantify how quickly, in number of episodes, the agent took to achieve the saturation value computed above.
Calculated by: The first time the saturation value (or above) is seen in the smoothed reward signal S_n, the number of episodes it took to achieve that performance is recorded as the Time to Saturation
Compared to: Future Times to Saturation of the same task
This Metric is Under Development

3. Area Under the Reward "Curve" Per Number of Episodes
Purpose: Taking the Sum of Accumulated Reward allows for a more robust comparison of the time to learn a particular task, taking into account both the shape and saturation of the learning for future comparison. Two systems given the same number of episodes may achieve the same saturation value, but one may achieve the value faster and this metric can reflect that advantage. However, this metric has limitations and must be normalized by length; additionally, only training phases can be compared to each other
Calculated by: Integrating reward over episodes, then dividing by the number of episodes used to accumulate the reward
Compared to: Future from scratch training instances of this metric
This Metric is Not Yet Implemented

We take the sum of the smoothed series S_n, where n is the integer block length (in number of episodes), normalized by the block length:

$$\frac{1}{n}\sum_{i=0}^{n} S_i$$

4. Recovery Time
Purpose: Recovery time is calculated to determine how quickly (if at all) an agent can "bounce back" after a change is introduced to its environment
Calculated by: After some training phase achieves a saturation value, determine how many episodes, if any, it takes for the agent to regain the same performance (within 2%) that it was previously attaining on the same task before the change
Compared to: An agent's recovery time is comparable across tasks
This Metric is Under Development

5. Performance Maintenance on Test Sets
Purpose: Performance maintenance on test sets is calculated to determine whether an agent catastrophically forgets a previously learned task
Calculated by: Comparing all commonly computed metrics on the train set (saturation values, time to saturation, etc) on the test set and computing the difference in performance
This Metric is Under Development

6. Performance relative to Single Task Expert
Purpose: Single Task Expert (STE) Relative performance assesses whether a lifelong learner outperforms a traditional learner.
Note: Single Task Expert Saturation values for each task **must** be included in a JSON file found in $L2DATA/taskinfo/info.json and without this file, this metric cannot be calculated. Further, the task names contained in the JSON file must match the names in the log files *exactly*. The format for this file must be:

```
{
    "task_name_1" : 0.8746,
```

```
    "task_name_2" : 0.9315,
    ...,
    "task_name_n" : 0.8089
}
```

Calculated by: Normalizing metrics computed on the lifelong learner by the same metrics computed on the traditional learner. No assumptions are made about what is found in the STE file in terms of the type of algorithm used; this metric will simply assume that the numbers found in this file are the saturation values achieved by the STE and will do a direct comparison. This will be used in particular for the computation of transfer matrices.
This Metric is Under Development

2.2 Adapting To New Tasks

Evaluation Questions and Definitions Coming Soon

Chapter 3

Creating Custom Syllabi

As mentioned in previous sections, a semi-rigid structure is required for syllabi in the Metrics Framework.

Phase Annotations

Syllabi used to generate the log files **must** include annotations with phase information in the following format: <phase_number>.<phase_type>, where phase_number is an increasing integer beginning with 1, and phase_type can be either "train" or "test". No other annotations are currently supported. Please note that without these annotations, the Metrics Framework will not properly compute the Proposed Metrics.

Valid Phase annotation format example:

```
Train: {"$phase":  "1.train"}, {"$phase":  "2.train"}, ..., {"$phase":  "n.train"}
Test: {"$phase":  "1.test"}, {"$phase":  "2.test"}, ..., {"$phase":  "n.test"}
```

Train/Test Structure

1. Required
    - The first block in a syllabus must be a Train block
2. Expected
    - Alternating Train phases (consisting of 1 or more blocks) followed by Test phases (consisting of 1 or more blocks)
3. Disallowed
    - Continual Learning Syllabi may not exercise more than one task
    - Adapting to New Tasks Syllabi may not neglect to include Testing phases
    - Adapting to New Tasks Subtypes A and B may not exercise parameter variation

3.1 Continual Learning

Structure

The purpose of this type of syllabus is to assess whether the agent can adjust to changes in the environment and maintain performance on the previous parameters when new ones are introduced. There is only one type of Continual Learning syllabus.

- A Continual Learning syllabus consists of a **single** task with parametric variations
  Can involve interpolation of parameters, noisy parameters, etc. but has only one task
- Each phase in a Continual Learning syllabus consists of a training and optional but recommended testing block
  Training block is training on one or more parametric variations

Example

This syllabus has alternating Train and Test phases (note the phase annotations) which contain blocks of all the same Task (`"Pong"`), but the parameter values (`"bot/paddle/width"`) vary between 0.15, 0.20, 0.25

```
"instructions": [
   {"$phase": "1.train"},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.15}, "count": 5000},

   {"$phase": "1.test"},
    {"$info": {"disable_updates": true} },
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.20}, "count": 100},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.15}, "count": 100},
    {"$info": {}},

  {"$phase": "2.train"},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.20}, "count": 5000},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.25}, "count": 5000},

   {"$phase": "2.test"},
    {"$info": {"disable_updates": true} },
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.15}, "count": 100},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.20}, "count": 100},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.25}, "count": 100},
    {"$info": {}}
]
```

3.2 Adapting to New Tasks

Overall Structure

The purpose of this type of syllabus is to assess whether the agent can learn new tasks and maintain performance on the previous tasks after new ones are introduced. It consists of multiple, distinct tasks, generally without parametric variations (except in Subtype C). The Testing phase is mandatory. There are three subtypes of an Adapting to New Tasks syllabus.

Adapting to New Tasks - Subtype A

- Assesses resistance to so-called "Catastrophic Interference"
- Consists of multiple, distinct Train tasks with subsequent Test blocks to compute whether performance is maintained after learning each new task.
- Does not exercise parametric variations within or across tasks.

Example

This syllabus has alternating Train and Test phases which contain blocks of different Tasks `"Pong, Freeway"` , and `"Breakout"` but no parameter variation. This syllabus can be used to assess whether the system can overcome so-called "Catastrophic Forgetting" by assessing the system's ability to learn multiple tasks while maintaining performance on previously learned tasks.

```
"instructions": [
   {"$phase": "1.train"},
    {"$repeat": { "$episode": "Pong"}, "count": 5000},

   {"$phase": "1.test"},
    {"$info": {"disable_updates": true} },
    {"$repeat": { "$episode": "Freeway"}, "count": 100},
    {"$info": {}},

  {"$phase": "2.train"},
    {"$repeat": { "$episode": "Breakout"}, "count": 5000},

   {"$phase": "2.test"},
    {"$info": {"disable_updates": true} },
    {"$repeat": { "$episode": "Pong"}, "count": 100},
    {"$repeat": { "$episode": "Freeway"}, "count": 100},
    {"$info": {}}
]
```

Adapting to New Tasks - Subtype B

- Assesses basic skill transfer
- Consists of multiple, distinct Train tasks with appropriate Test blocks to compute transfer matrix.
- Forward transfer calculation requires Single-Task-Expert baseline for comparison
- Does not exercise parametric variations within or across tasks.

Example

This syllabus has tasks which are chosen intentionally for the purpose of assessing skill transfer. These tasks should have an expectation of shared knowledge or skill representation so that a sensible transfer matrix can be computed and evaluated. The syllabus below shows an example where there is a shared "skill" - paddle control - and demonstrates the minimum required blocks to compute a transfer matrix with two tasks. In phase "1.test", forward transfer from Pong to Breakout can be assessed. In phase "2.test", reverse transfer from Breakout to Pong can be asseessed.

```
"instructions": [
   {"$phase": "1.train"},
    {"$repeat": { "$episode": "Pong"}, "count": 5000},     <-- Train Task 1

   {"$phase": "1.test"},
    {"$info": {"disable_updates": true} },
    {"$repeat": { "$episode": "Breakout"}, "count": 100},  <-- Test forward transfer
    {"$info": {}},

  {"$phase": "2.train"},
    {"$repeat": { "$episode": "Breakout"}, "count": 5000}, <-- Train Task 2
```

```
   {"$phase": "2.test"},
    {"$info": {"disable_updates": true} },
    {"$repeat": { "$episode": "Pong"}, "count": 100},      <-- Test reverse transfer
    {"$info": {}}
]
```

Adapting to New Tasks - Subtype C

- Assesses parametric skill transfer
- Consists of multiple parametric variations of one Train task, followed by parametric variations of a Test Task in appropriate Test blocks; then the order is switched.
- Run *only* with Tasks which successfully demonstrate transfer of basic skill
- Allows assessment of Transfer of Continual Learning, Improvement of Time-to-Learn, and Transfer Matrix Calculation
- Exercises parametric variations within and across tasks.

Example

This syllabus contains both task and parameter variation and enables more complicated evaluation metrics to be computed, including transfer of Continual Learning, improvement of time-to-learn, etc.

```
"instructions": [
   {"$phase": "1.train"},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.15}, "count": 5000},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.20}, "count": 5000},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.25}, "count": 5000},

   {"$phase": "1.test"},
    {"$info": {"disable_updates": true} },
    {"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.15}, "count": 100},
    {"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.20}, "count": 100},
    {"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.25}, "count": 100},
    {"$info": {}},

  {"$phase": "2.train"},
    {"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.15}, "count": 5000},
    {"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.20}, "count": 5000},
    {"$repeat": { "$episode": "Breakout", "bot/paddle/width": 0.25}, "count": 5000},

   {"$phase": "2.test"},
    {"$info": {"disable_updates": true} },
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.15}, "count": 100},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.20}, "count": 100},
    {"$repeat": { "$episode": "Pong", "bot/paddle/width": 0.25}, "count": 100},
    {"$info": {}}
]
```

Chapter 4

Creating Custom Metrics

4.1 Conceptual Overview

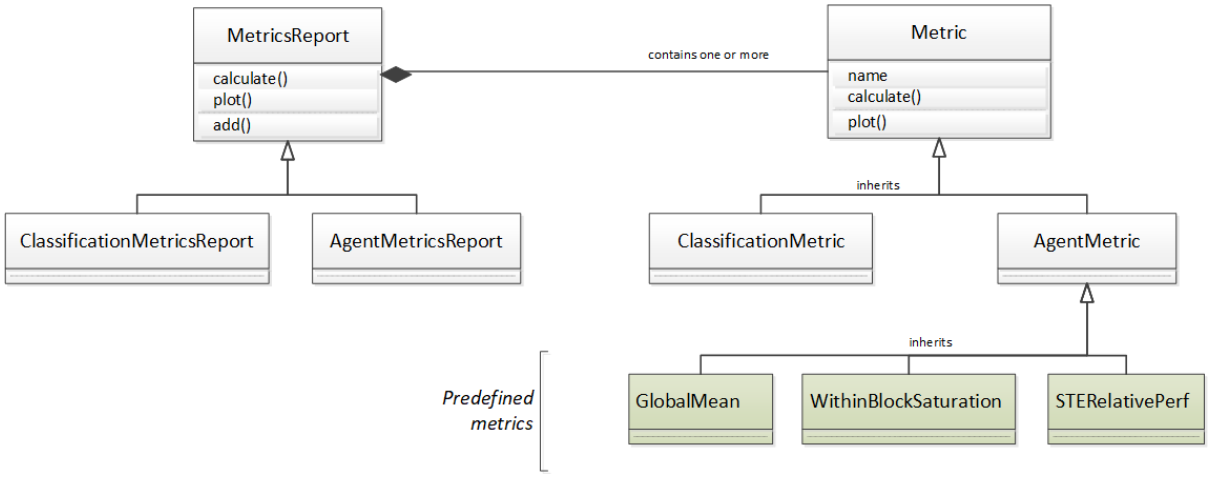

Figure 4.1: UML Diagram for classes in the Metrics Framework. The predefined metrics shown are a subset of the full set of metrics, and shown only for illustration.

Conceptually, a MetricsReport object contains one or more Metrics. By default, this includes a set of predefined Metrics (such as GlobalMean or WithinBlockSaturation). The MetricsReport.calculate() method loads the log data into a Pandas dataframe, does some pre-processing and cleanup, and then hands the dataframe to each Metric's calculate() method. Figure 4.1 shows the diagram for the classes in the Metrics Framework (specifically, the l2metrics Python package).

Consequently, it is easy to author and use a custom Metric, as follows:

1. Create a Python class, subclassing from the ClassificationMetric or AgentMetric as appropriate
2. Implement the interface (in particular, the name property and calculate() method)
3. Write a top-level script that instantiates the ClassificationMetricsReport or AgentMetricsReport (as appropriate) and adds the custom metric to the report (replacing or augmenting the predefined metrics).
4. Run the top-level script.

Chapter 5

Appendix A: Terminology

Core Capabilities and Terminology References

The Metrics Framework and Proposed Metrics rely on the Core Capabilities defined by the BAA to direct their purpose and intent. They are provided here for reference.

1. Continual Learning
Ability to handle, or adapt to, changing input distributions (or noise characteristics) within a single task. For an agent-based system, this means that the state transition matrix and reward function are substantively unchanged, while aspects of the environment may change.

2. Adapting to New Tasks
Ability to learn new tasks, without losing knowledge of already-learned tasks. If possible, system should exploit similarities between old and new tasks to improve its learning performance on new tasks. For an agent-based system, this means substantial changes to the state transition matrix or reward function.

3. Selective Plasticity
Ability to process the same input differently depending on task (or goal). Relatedly, the ability to be sensitive to different features of the input depending on task. Note the focus on processing rather than learning per se.

4. Goal-Driven Perception
Ability to incorporate system and task-level constraints (e.g., overall memory use, relative importance of tasks) into the training process.

5. Safety
Ability to incorporate explicit (failsafe) safety constraints into system performance; Ability to detect differences between training and test environments (e.g., anomalous inputs, distributional shifts), quantify the resulting uncertainty in system output, alert a human operator (with details of difference if possible), and safely handle the anomalous situation where possible.

Terminology

Key Concepts

Task: A single abstract capability (or skill) that a performer system must learn
Episode: A concrete instance of a task
Syllabus: A sequence of episodes
Learning Lifetime: One syllabus or multiple syllabi in sequence

Metric Specific Terms for Syllabus Design

Phase: A subcomponent of a syllabus during which either training or evaluation takes place
Block: A unique combination of task and parameters. It is a subcomponent of phase that is automatically assigned by the logging code and does not need to be annotated by the syllabus designer.