Dagger: A Data (not code) Debugger

El Kindi Rezig[⋆] Lei Cao[⋆] Giovanni Simonini[⋆] Maxime Schoemans[⋄]
Samuel Madden[⋆] Mourad Ouzzani[†] Nan Tang[†] Michael Stonebraker[⋆]
[⋆]MIT CSAIL [⋄]Université Libre de Bruxelles [†]Qatar Computing Research Institute
{elkindi, lcao, giovanni, madden, stonebraker}@csail.mit.edu
maxime.schoemans@ulb.ac.be {mouzzani, ntang}@hbku.edu.qa

ABSTRACT

With the democratization of data science libraries and frameworks, most data scientists manage and generate their data analytics pipelines using a collection of scripts (e.g., Python, R). This marks a shift from traditional applications that communicate back and forth with a DBMS that stores and manages the application data. While code debuggers have reached impressive maturity over the past decades, they fall short in assisting users to explore data-driven what-if scenarios (e.g., split the training set into two and build two ML models). Those scenarios, while doable programmatically, are a substantial burden for users to manage themselves. *Dagger* (Data Debugger) is an end-to-end data debugger that abstracts key data-centric primitives to enable users to quickly identify and mitigate data-related problems in a given pipeline. *Dagger* was motivated by a series of interviews we conducted with data scientists across several organizations. A preliminary version of *Dagger* has been incorporated into Data Civilizer 2.0 to help physicians at the Massachusetts General Hospital process complex pipelines.

1. INTRODUCTION

As we are moving towards data-centric systems, building and debugging data pipelines has become more crucial than ever. Choosing and tuning those pipelines is a difficult problem. As they increase in complexity, a number of things can go wrong during pipeline modeling or execution: (1) code bugs: there may be bugs in one or more of the pipeline's modules. This requires tracking down potential bugs (e.g., using a code debugger) and then fixing the code; (2) wrong parameters: it is often the case that one or more components require substantial parameter tuning, e.g., in ML models [17]. In this case, the user (or an algorithm) evaluates the pipeline using a set of parameters to get better results (e.g., the accuracy of the ML model); and (3) data errors: the input data or any of the intermediate data produced by the code has a problem, e.g., training data is not large enough for a machine learning model to make reasonable inferences or the data is in the wrong format (e.g., bad schema alignment or the data is wrong). In *Dagger*, we focus on addressing problems related to data errors. In doing so, users may be able to also identify errors related to code bugs or non-ideal parameters, but they would discover those issues by investigating the data that is handled in their code. In a given data pipeline, data goes through a myriad of transformations across different components. Even if the input data is valid, its intermediate versions may not be. We refer to volatile data that is handled in the code (e.g., scalar variable, arrays) as *code-handled* data.

We conducted a series of interviews with data scientists and engineers at the Massachusetts General Hospital (MGH), Tamr, Paradigm4 and Recorded Future to gauge interest in developing a tool that assists users in debugging code-handled data, and found widespread demand for such a *data debugger*.

To address this need, we introduce *Dagger*, an end-to-end framework that treats code-handled data as a first-class citizen. By putting code-handled data at the forefront, *Dagger* allows users to (1) create data pipelines from input Python scripts; (2) interact with the pipeline-handled data to spot data errors; (3) identify and debug data-related problems at any stage of the pipeline by using a series of primitives we developed based on real-world data debugging use-cases (Section 2).

Motivated by use-cases we encountered while collaborating with MGH, *Dagger* is a work-in-progress project, and we have already incorporated its preliminary version into Data Civilizer 2.0 (*DC2*) [15]. In this paper, we sketch the design of the extended version of *Dagger*.

1.1 Dagger overview

Figure 1(a) illustrates the architecture of *Dagger*. In a nutshell, the user interacts with *Dagger* through a declarative SQL-like language to query, inspect, and debug the data that is input to and produced by their code.

Workflow manager: In order to integrate user pipelines into off-the-shelf workflow management systems [2, 15, 10], users are often required to write additional code. *Dagger* allows users to leave their code in its native environment (e.g., Python) and only requires them to tag different code blocks that will become the pipeline nodes. The code still runs as it normally would, i.e., *Dagger* pipelines do not alter how the underlying code runs, instead, *Dagger* will only monitor the code-handled data at the tagged code sections. The only exception to this rule is with one of the debugging primitives (SPLIT) which creates different run branches (more details in Section 2). We chose Python as the target programming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CIDR '20 January 12–15, 2020, Amsterdam, Netherlands
© 2019 ACM. ISBN . . . $15.00
DOI:

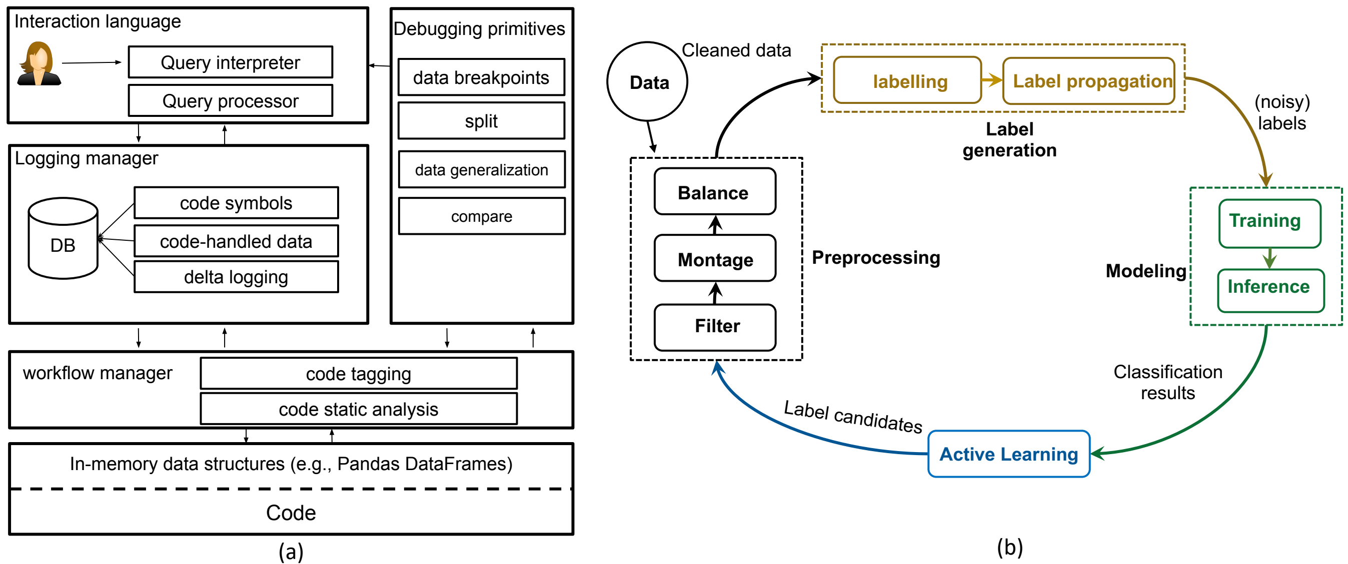

Figure 1: (a) Dagger Architecture; (b) Workflow used in EEG Application

environment because it is the most popular language among data scientists [1]. We refer to pipeline nodes as blocks in the rest of the paper. For instance, users can create the pipeline in Figure 1(b) that includes scripts scattered in different Python files by running the following *Dagger* statements (we will explain the pipeline in Section 2):

```
CREATE BLOCK b1 FOR PIPELINE P1: preprocessing.py:1-600
CREATE BLOCK b2 FOR PIPELINE P1: label_generation.py:10-1440
CREATE BLOCK b3 FOR PIPELINE P1: modeling.py:200-5000
CREATE BLOCK b4 FOR PIPELINE P1: active_learning.py:40-3000
```

Every statement of the above snippet creates a pipeline block that is attached to a code section of interest. After the blocks are tagged, *Dagger* performs a static analysis of the code to extract the dataflow. This requires generating a call graph (i.e., how the different code blocks are interlinked). This step will create the edges in a *Dagger* pipeline.

Logging manager: once the pipeline blocks are defined, the logging manager stores, for a given data value, its initial value and its updated versions across pipeline blocks at runtime. The logger can track the value of any data structure (e.g., Pandas DataFrames, variables, etc.). For well-known data types such as Pandas DataFrames [13] and Python built-in types, the logging manager logs their values automatically. However, for user-defined objects, a logging function has to be provided to extract the value of the objects. The logging manager stores the tracked code-handled data and metadata (e.g., variable names) into a relational database.

Interaction language (DQL): Because *Dagger* exposes code-handled data to users, it features a simple-to-use SQL-like language, DQL (*Dagger* Query Language), to facilitate access and interaction with the pipeline data (e.g., look up all the occurrences of a given data value in the pipeline). Users post their DQL debugging queries through a command-line interface.

Debugging primitives: *Dagger* supports key debugging primitives: (1) data breakpoints to write data assertions (e.g., is the salary of an employee ever set to a negative value?) that are checked across the pipeline blocks; (2) split to partition input tables and run a user-specified function on all of them. For instance, we could partition the training dataset on a value of an attribute and build classifiers using each partition as the training data. In this case, the user-defined function is the classifier. (3) data generalization to find, given example data values in a table, a partition based on a user-provided similarity function (e.g., Euclidean distance) (4) compare to compare different pipelines using their code-handled data, i.e., if two pipelines generate similar intermediate data, then, it is likely their downstream blocks would produce similar results.

1.2 Use Cases

Dagger is being used in our ongoing collaborations with two separate groups at the Massachusetts General Hospital (MGH). We will provide examples based on those two use cases throughout the paper.

Use case 1: Our first use case is an ML pipeline for classifying seizures in electroencephalogram (EEG) data from medical patients at MGH. By employing data cleaning and regularization techniques in *Dagger*, we have been able to achieve better accuracy for the ML models used in the EEG application. The better performing data pipelines were devised by iteratively building pipelines and trying various cleaning routines to preprocess the data.

Use case 2: We have recently started a collaboration with another MGH group that focuses on tracking infections inside the hospital. For instance, a patient might spread an infection to a nurse, who then, spreads it to other patients. In this use case, structured data about patients has been used to visualize the spread of infections across the hospital floor plans using *Kyrix* [16]. This data has several errors due to manual editing (e.g., misspelling, wrong date/time recording) or bad database design, e.g., a newborn inherits the same identifier as the mother which results in two patients having the same identifier. Furthermore, some information has to be inferred using scripts, e.g., the date and time of the patient's check-out from a room (some records only have the check-in date and time). We used *Dagger* to help build pipelines to prepare this data, and to efficiently help detect data errors in the intermediate steps of the pipeline (more details in Section 4).

2. MOTIVATING EXAMPLE

We introduce the key primitives of *Dagger* through motivating scenarios we have experienced in Use Case 1. We first introduce some background on the EEG application, and then proceed with data debugging primitives we found most useful to build a classifier that predicts seizures from EEG data.

2.1 Setup

In our collaboration with MGH, we built an end-to-end large scale ML system that automatically captures and classifies seizures by analyzing the data produced during the EEG monitoring of the patients in the intensive care unit (ICU) [5]. This system is designed to detect the possibility of neural injury by capturing seizures as early as possible. The classifier predicts six classes, which correspond to different EEG patterns that characterize different types of seizures [8] A deep neural network was used to build a classifier from the training EEG data (for details, refer to [5]). To meet the classification accuracy requirement of clinical practice, we built a pipeline that includes data preprocessing, labeling, and classification, as shown in Figure 1(b). This enables the neurologists to iteratively and interactively label and clean the data driven by the output of the classification model, progressively improving the classification accuracy.

Figure 1(b) illustrates the final pipeline that we built for the prediction of seizure likelihood. In a nutshell, there are three phases in the pipeline: (1) Preprocessing: this phase includes the Balance, Montage and Filter modules, which are EEG-specific preprocessing operators. (2) Label propagation: an EEG segment encodes several features that are extracted and then clustered with similar segments using Euclidean distance. This clustering step allows us to propagate labels from human-labeled segments to other segments in the same clusters. This allows us to generate more training data for the neural network that is used to classify EEG segments; and (3) Modeling: finally, we train a neural network using both the manually-labeled as well as the propagated EEG segments and infer various seizure-related classes in the inference step.

How Dagger fits in: Data debugging is particularly important in this iterative learning process, since the accuracy of the classification model relies on the availability of a large amount of high quality labels and clean training data. For example, since the experienced neurologists are a scarce resource and their time is precious, it is not practical to rely on them to manually label the big EEG dataset collected by MGH (30TB, 450 million EEG segments). *Dagger* facilitates all of the steps in Figure 1(b) to help them improve the accuracy of the classifier. In this case, Dagger proved particularly helpful in preprocessing the data, i.e., we identified anomalies such as outliers and all-zero EEG segments in the data flowing across the pipeline blocks.

2.2 Putting Dagger Primitives in Context

We present a walk-through of how the debugging primitives of *Dagger* helped us build a high-accuracy ML model to predict seizures. We will present the examples using a language that we use in *Dagger* (DQL) to specify debugging primitives and interact with the code-handled data. We present the DQL specifications informally and explain them in the remainder of the paper. Due to space constraints, we omit the full syntax of DQL statements and their variations in this paper.

Data Generalization: *Dagger* provides a Data Generalization primitive to automatically and reliably produce labels. More specifically, given a few EEG segments manually labeled by the neurologists, the data generalization primitive quickly finds and propagates labels to the objects similar to these labeled examples.

Example 2.1. Given the following statements, *Dagger* uses the Euclidean distance on features derived from EEG segments to find segments that are most similar to segments E1, E2 and E3 in table *table_eegs*. The output of the following statement is a set of clusters of EEG segments within a 0.8 Euclidean distance to $E1$, $E2$ and $E3$.

```
GENERALIZE E1, E2, E3 FROM table_eegs
USING EUCLIDEAN THRESHOLD 0.8
```

Split: In order to effectively use the neurologists' time, we have to make sure that they only label the critical examples. In particular, we ask them to focus on examples from the classes that are most likely to be misclassified, because fixing errors in these classes likely reduces classification errors. Therefore, *Dagger* features a split primitive that partitions the training dataset based on different predicates (e.g., class labels) so that users can analyze different models produced with different partitions.

Example 2.2. The split primitive is important in the data preprocessing step (Figure 1(b)). For example, *Montage* is a technique widely used in EEG data preprocessing [14], which involves creating additional training data by slicing, reversing, and replicating segments of EEG data. When used appropriately, *Montage* can improve the classification accuracy. However, it does not necessarily improve performance on all EEG segment classes equally. Therefore, once the neurologists figure out how *Montage* performs on each individual class by applying the split primitive on the classification results, they can then use the split primitive again to partition the training data into six disjoint subsets corresponding to the six seizure classes. Then, *Montage* is only applied to the data subsets that clearly benefit from it. We express this use-case through the following *Dagger* specification:

```
1: RUN PIPELINE P1 WITH eeg_all
2: RUN BLOCK MONTAGE IN P1 WITH SPLIT eeg_all
   WHERE class_label <> Other
```

In line 1, the user runs pipeline P1 (Figure 1(b)) on all the training data (*eeg_all*). Then the user realizes that the accuracy of the produced classification model is not good for one class (*Other*). In line 2, *Dagger* splits the training data into different partitions based on the class label. *Dagger* then invokes the *Montage* routine (encapsulated in the block Montage in P1) on the resulting partitions except those with the class label *Other*. The user will then use the output of this pipeline to train the model.

Data breakpoints: Data breakpoints are used to express test conditions on the code-handled data at different stages in the pipeline. They are useful in speeding up our learning pipeline, i.e., we can stop the execution of the pipeline if an assertion on the data fails. For example, the neurologists often want to know if the classification accuracy of a targeted

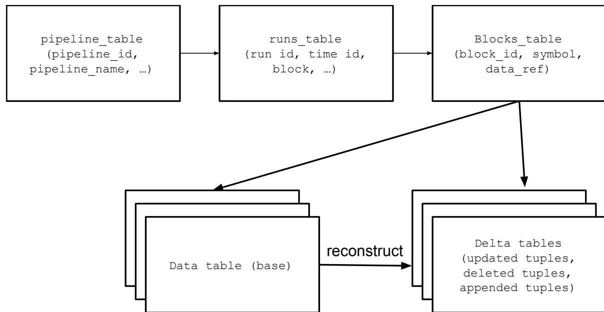

Figure 2: Logging manager tables

class is improved after collecting new labeled EEG examples for this class and update the model. In this case, the neurologists can set a data breakpoint that will terminate the modeling process immediately once the error rate in this targeted class is above a given threshold.

Compare: The compare primitive is also frequently used in the learning pipeline to help neurologists determine if a new preprocessing operation or parameter setting would make a difference on the final output of the pipeline. For example, the neurologists expected that high pass/low pass filters (which discard frequencies that are not of interest from the EEG data) could improve the classification accuracy. This hypothesis was not confirmed after running the model. Consequently, if we can compare the EEG segments after applying the filter operator with those obtained without filtering, and find they are similar, then we can infer that the ML accuracy does not change much, allowing us to skip the expensive modeling and training process on the version of the pipeline with the filter operators.

3. LOGGING MANAGER

Now that we have described a motivating application, we discuss some of the technical challenges in building *Dagger*, beginning with a description of how the logging manager works to efficiently record intermediate pipeline states.

Every run of the pipeline produces data across the pipeline blocks. In order to persist this data for debugging, we need to design a storage model that can accommodate data types that are popular in Python.

The storage model of *Dagger* (Figure 2) consists a set of tables (stored in a Postgres database), each containing information about: (1) variable names in the code along with their values; (2) pipeline block identifiers where each variable value is reported; (3) run identifier that tracks which data items correspond to each pipeline run; and (4) time identifiers that reflect the order in which those data items were produced in a given run (as determined by the pipeline).

By default, we support two data types in the storage model of *Dagger*: (1) **Scalar:** these include numeric types (e.g., integer, float); and (2) **Arrays:** which include any data item that can be represented as an array of any dimensions (e.g., tensor, time series). For instance, Pandas DataFrames are captured using Arrays

While we could have more complex data types, by initially supporting these two types we found we were able to support the needs of most pipelines.

3.1 Delta Logging

An important question to answer is: what to log from a given pipeline? Obviously, logging at the code-statement level is not an option, i.e., it will produce too much redundant data. Recent efforts have addressed the data versioning problem [4, 11, 9], however, those proposals do not deal with code-handled data and are mainly meant for versioning input datasets and managing their evolution over time [18]. We propose to log data values at their initial state (base version) and then only record their changes across the pipeline blocks. This requires implementing a process that keeps track of the part of data that has changed from one statement (or pipeline block) to another. We propose a 3-phase approach to (1) build a list of all the variables that are accessed at least once (read or written) in each block; (2) detect which data structures have changed from one block to another; and (3) version the changed data structures across blocks.

Scope determination: in this phase, *Dagger* performs a static analysis of the code to determine which variables (or data structures) are within the scope of each user-defined pipeline block. We are just interested in the scope of variables, i.e., we just want to know which variables can be accessed within each block. This operation can be performed using classical dataflow analysis methods [3] (e.g., worklist algorithm). After this step, *Dagger* will have a list of all the variables that are in the scope of each block.

Change detection: in this step, *Dagger* determines if a variable has changed its value from one block to another (the same variable can appear in multiple blocks). Specifically, for a given variable v in the scope of a block b_1, *Dagger* determines if v has changed its value in b_2, where b_1 ! b_2. While this is trivial for scalar types, more efficient methods are needed for tables (e.g., Pandas DataFrames) of large sizes.

Versioning: There are many ways we could approach data versioning, but in general, given consecutive blocks b_1 and b_2 where b_1 ! b_2 and a variable D with two versions D^1 and D^2 in b_1 and b_2 respectively, we would like to determine the changes that happened from D^1 to D^2 (i.e., which data values have been changed). We first determine the types of changes from D^1 to D^2, i.e., update, append and deletion and then store sufficient state to reconstruct both D^1 and D^2 (e.g. D^1 and a delta from it to D^2). It is important to point out that the intermediate data across pipeline blocks can be quite large, especially if the pipeline includes many blocks. We are planning to address this problem by adapting existing data structures (e.g., Merkel trees [6]) to work efficiently in the context of code-handled data.

4. DAGGER QUERY LANGUAGE (DQL)

Because *Dagger* manages code-handled data, it is crucial to have an easy-to-use language for users to interact with *Dagger* pipelines without having to write SQL queries directly on its storage manager tables. *Dagger* supports querying the data across the pipeline blocks. Since *Dagger* stores the data in a relational database, all the SQL operations on the stored data come for free. *Dagger* exposes a SQL-like language to abstract the key primitives we use in data debugging. We now present the overall syntax of *DQL*. The *DQL* examples reported in the following are based on Use Case 2 introduced in Section 1.1, where a pipeline (composed of several Python scripts) has been devised using *Dagger* to preprocess spatio-temporal data.

block: *DQL* supports querying data structures in specific

pipeline blocks or between specific blocks. For instance, we have the following *DQL* query Q_1:

```
LOOKUP A FROM P2 WHERE SCOPE BETWEEN B=1 and B=2
```

The above query will perform a projection over all the attributes of data structure A (stored as a table) from pipeline $P2$ between the blocks 1 and 2. Internally, Q_1 is rewritten in SQL and posted to *Dagger*'s database.

Value lookup: we formulate LOOKUP queries to search for a specific data value in a given block or pipeline. The supported values for lookup are strings and numerical types.

Example 4.1. A researcher at MGH, Julie, is interested in analyzing and predicting a particular infection disease (C. diff [7]), in a particular floor of the hospital, but at the end of the data preparation pipeline she finds data errors (e.g., dates inconsistencies and patients assigned to rooms that do not exist). To narrow down the search space to locate and fix those errors, she writes the following *DQL* query:

```
LOOKUP `C. DIFF', `F1%'
FROM PIPELINE P2
WHERE SCOPE BETWEEN B=1 AND B=5
```

The query returns all intermediate data of pipeline P2 (between blocks 1 and 5), that contain the string "C. DIFF" and a string starting with "F1" (i.e., building F, floor 1).

Ordered lookup: a typical query one would want to write is: did a change X (to a data structure) happen before or after another change Y. Or did a particular data item appear before or after another? Those queries are particularly important for causality analysis [12]. To express such queries, we need to be able to reason about the "time steps" within a given run. As we discussed in Section 3, the logging manager keeps track of the time steps for any given run.

Example 4.2. Julie notices that two patients tested positive to the same rare bacterial strain, but never shared the same hospital room. This could be an error introduced in the pipeline, i.e., maybe at some point in the pipeline, those two patients shared the same room, but then the room number was modified. To test this hypothesis, Julie writes the following *DQL* query:

```
LOOKUP A FROM PIPELINE P2
WHERE (A[ROOM] = `F12'
BEFORE A[ROOM] <> `F12')
AND A[INFECTION] = `C. DIFF'
```

The query returns the records for which the value of the attribute ROOM has changed from 'F12' to something else, for patients with the C. diff infection.

Backward and forward propagation: *DQL* supports editing a data value or a set of values and propagate the change forward or backward in the pipeline. For instance, if we want to propagate the value of D from v_1 to v_2 forward from blocks b_1 to b_2 where b_1 ! b_2, *Dagger* proceeds as follows: (1) *Dagger* first posts LOOKUP queries for $D = v_1$ on blocks b_1 and b_2; (2) the corresponding records in the database are updated to v_2; (3) from the database, *Dagger* keeps track of each code line number where D is accessed; (3) when b_1 or b_2 are re-run, *Dagger* pulls D from the database at runtime to reflect their updated state in the code.

Example 4.3. Julie notices that some patients are in multiple locations (in the hospital floor plan) at the same time (e.g., with a LOOKUP query) because newborns inherit their unique identifiers from their mothers. To fix this data issue, and check the effects of this fix on the pipeline output, Julie writes the following *DQL* query:

```
EDIT A[ID] = new_id()
FROM PIPELINE P2
WHERE A[DESCRIPTION] LIKE `%NEWBORN%'
```

where *new_id()* is a function that returns an id that is not present in the id column of *A*.

Inter-run queries: users can also query data items across different runs. For instance:

```
LOOKUP A FROM P2 WHERE RUNS ALL
LOOKUP A FROM P2 WHERE RUNS BETWEEN 1 and 10
```

The first query reports the values of A in all the runs performed so far of pipeline $P2$ while the second one reports A's value in $P2$ only in the first 10 runs.

5. CURRENT PROTOTYPE

We have a preliminary version of *Dagger* already incorporated into Data Civilizer 2.0 (DC2) [15]. In *DC2*, pipeline modules are created through a Python API. The version of *Dagger* in *DC2* provides the following features for pipeline debugging: (1) breakpoints: users implement an API to create a data breakpoint object or the breakpoints are created automatically by splitting the input data into multiple chunks of varying size (e.g., [10, 20, 100]); (2) tracking: track data items that meet certain conditions; (3) pause/resume: suspend pipeline execution upon the user's request; (4) filter: based on user-defined predicates, we filter the input/output data from different modules in the pipeline.

DC2 and *Dagger* have been used as part of our collaboration with MGH (Use Case 1, presented in Section 2). The collaboration with the Infectious Disease group (Use Case 2) is still in its early stages, and we are at the stage of data preparation.

Figure 3 illustrates a running example of building the MGH pipelines (Use Case 1) in *DC2*. Figure 3(a) illustrates the seizure prediction pipeline (ML 4). This pipeline includes all the cleaning operators and outperforms other earlier iterations of the pipeline (ML 1, ML 2 and ML3) which either did not include any cleaning, or included only a subset of the cleaning operators. The results are reported in Figure 3(b) where we note that ML 4 has better validation accuracy than the other pipelines. It also shows a gradual improvement in accuracy over time as it processes more data. Figure 3(c) shows an overview of the preliminary version of *Dagger*, the red part indicates that a data breakpoint evaluated to true during the run of the pipeline and hence the pipeline execution was suspended. During the presentation of this paper, we plan to conduct a significant demo of *Dagger* which will feature real-world pipeline debugging examples using real datasets and scenarios.

6. CONCLUSION

We presented *Dagger*, an end-to-end data debugger that assists users in debugging code-handled data. We presented the primitives of *Dagger* in the context of real-world scenarios we encountered over the course of our ongoing collaborations with two groups from Mass. General Hospital.