Data Civilizer 2.0: A Holistic Framework for Data Preparation and Analytics

El Kindi Rezig⋆ Lei Cao⋆ Michael Stonebraker⋆ Giovanni Simonini⋆ Wenbo Tao⋆ Samuel Madden⋆ Mourad Ouzzani⋄ Nan Tang ○ Ahmed K. Elmargamid○

⋆MIT CSAIL ○Qatar Computing Research Institute, HBKU
{elkindi, lcao, stonebraker, giovanni, wenbo, madden}@csail.mit.edu
{mouzzani, ntang, aelmargamid}@hbku.edu.qa

ABSTRACT
Data scientists spend over 80% of their time (1) parameter-tuning machine learning models and (2) iterating between data cleaning and machine learning model execution. While there are existing efforts to support the first requirement, there is currently no integrated workflow system that couples data cleaning and machine learning development. The previous version of Data Civilizer was geared towards data cleaning and discovery using a set of pre-defined tools. In this paper, we introduce Data Civilizer 2.0, an end-to-end workflow system satisfying both requirements. In addition, this system also supports a sophisticated data debugger and a workflow visualization system. In this demo, we will show how we used Data Civilizer 2.0 to help scientists at the Massachusetts General Hospital build their cleaning and machine learning pipeline on their 30TB brain activity dataset.

1. INTRODUCTION
Data scientists spend the bulk of their time cleaning and refining data workflows to answer various analytical questions. Even the most simple tasks require using a collection of tools to clean, transform and then analyze the data. When a machine learning model does not produce accurate results, it is due to (1) raw data not prepared correctly (e.g., missing values); or (2) the model needs to be tuned (e.g. fine-tuning of the model’s hyperparameters). While there are many efforts to address those two problems independently, there is currently no system that addresses both of them holistically. Users need to be able to iterate between data preparation and fine-tuning their machine learning models in one workflow system. We worked with scientists at the Massachusetts General Hospital (MGH), one of the largest hospitals in the US, to accelerate their workflow development process. Scientists at MGH spend most of their time building and refining data pipelines that involve extensive data preparation and model tuning. Through our interaction, we pinpointed the following hurdles that stand in the way of fast development of data science pipelines (in the sequel, we use the words “pipeline” and “workflow” interchangeably).

Decoupling Data Cleaning and Machine Learning: When it comes to building complex end-to-end data science pipelines, data cleaning is often the elephant in the room. It is estimated that data scientists spend most of their time cleaning and pre-processing raw data before being able to analyze it. While there are a few emerging machine learning frameworks [2, 1, 14], they fall short when it comes to data cleaning support. There is currently no interactive end-to-end framework that walks users from the data preparation step to training and running machine learning models.

Coding Overhead: In larger organizations, it is typically the case that several scientists/engineers write scripts that deal with different parts of the data science pipeline. While many data science toolkits and libraries (e.g., scikit-learn) have gained a wide adoption amongst data scientists, they are only meant to build standalone components and hence are not well-suited to building and maintaining pipelines involving a wide variety of tools and datasets. As a result, scientists have to write code to build and maintain data pipelines and update the code whenever they need to refine them. Because building data pipelines is a trial-and-error process, maintaining scripts hardwired for specific pipelines is time-consuming. Moreover, the effort required to try out different pipelines typically limits the exploration space.

Debugging Pipelines: When building a pipeline involving different modules and datasets, it is typical that the final output data does not look right. This is typically due to (1) a problem in the modules (e.g., bug, bad parameters); or (2) the input data to the modules was not good enough to produce reasonable results (e.g., missing values). The latter case is hard to debug using current debuggers that focus mainly on code, i.e., users have to dump and inspect intermediate data to find where it went wrong. Since it takes hundreds of iterations to converge to a pipeline that works well for the task at hand, a data-driven debugger can significantly decrease the time spent in this process.

Visualization: Different datasets require different types of visualizations (e.g., time series, tables). Typically, scientists visualize the data in its raw format (e.g., tables) or manually visualize the data using commodity software like Microsoft Excel. However, when building pipelines iteratively, it is daunting to seamlessly integrate visualization applications (panning, zooming) into the pipeline-building process. Moreover, users need to spend a lot of time if they elect to write custom visualizations of their datasets.

There are several efforts to support data cleaning tasks [9, 7, 10], iterative machine learning workflow development [11, 1, 2, 4], and data workflow debugging [6]. Each of those efforts focuses on one aspect of the pipeline development at a time, but not all.

The previous version of Data Civilizer [5, 3, 8] focused on data discovery and cleaning using pre-defined tools. In most scenarios, users clean their data to feed it to machine learning models. We introduce Data Civilizer 2.0 (DC2, for short) to fill the gap between data cleaning and machine learning workflows and to accelerate iterative pipeline building through robust visualization and debugging capabilities. In particular, DC2 allows integrating general-purpose data cleaning and machine learning models into workflows with minimal coding effort. The key features of DC2 are:

- User-defined modules: In addition to a state-of-the-art cleaning and discovery toolkit that we already provide [8], users
2. SYSTEM ARCHITECTURE

We provide a high-level description of the DC2 architecture (Figure 1) and details are discussed in the subsequent subsections. DC2 includes three core components: (1) User-Defined Modules cover required functionalities to support plugging-in existing user-defined modules into the workflow system (Section 2.1); (2) Debugger which includes a set of operations to do data-driven debugging of pipelines (Section 2.4) and; (3) Visualization abstractions to facilitate building scalable visualization applications to inspect the data produced at different stages of the pipeline (Section 2.3). Users interact with DC2 using the DC2 Studio, which is a front-end Web GUI interface to author and monitor pipelines.

2.1 User-defined Modules

Users can plug-in any of their existing code into a DC2 workflow. Because cleaning and machine learning tools can vary widely, DC2 features a programming interface that is abstract enough to cover any data cleaning or machine learning module.

2.1.1 Module Specification

In order to specify a new module in DC2, users must (1) implement a module execution function (executeService) using the DC2 Python API; (2) load the module into DC2 by specifying its entry point file, i.e., the file that contains the implementation of the module execution function; and (3) write a JSON file to list the parameters the module requires for execution.

2.1.2 Pipeline Execution

Service execution happens in two phases: (1) Studio generates a JSON object containing the authored workflow, which includes: module names, parameters and the connections between modules. This JSON object is then passed to the backend (workflow manager in Figure 1) to run the workflow and; (2) every module produces a JSON object containing the path of output CSV files which are then passed to the next module in the workflow. All the DC2 modules use a “table-in, table-out” principle, i.e., input and output of all modules is a table. In case the module fails to run, an error code is sent back to DC2 and the pipeline execution is stopped.

eexecuteService. The module execution function (executeService) takes as argument the JSON file generated from the DC2 studio. This JSON file contains the parameter values as specified from the studio for the individual modules as well as the authored workflow. Every module (1) reads a set of CSV input files; (2) writes a set of CSV output files; and (3) might use metadata files if specified as an argument.

Every module can produce various output streams. We separate them into: output and metadata. Files produced under the output stream are passed on to any successor modules in the pipeline while files in the metadata stream are just meant to serve as “logs” that users can inspect to debug the module. For instance, a similarity-based deduplication module can produce an output stream containing the deduplicated tuples and a metadata stream that includes the similarity scores between pairs of tuples that were marked as duplicates. Each module has to produce a JSON file (output JSON) that specifies which files are produced as output or metadata.

2.1.3 I/O Specification

Every DC2 module is associated with a JSON file (input JSON) containing the list of parameters the module expects and their type. Additionally, the input JSON contains the module metadata (e.g., module name, module file path). DC2 Studio needs this specification to load the module into the GUI (e.g., if a module expects two parameters, two input fields are created in the GUI for that module).

2.2 Managing Machine Learning Models

DC2 supports adding machine learning models in the workflow. We integrated ModelDB [13] into DC2 to offer first-class support for machine learning model development. ModelDB supports the widely used scikit-learn library in Python. Users who include machine learning modules in the pipeline can (1) track the models on defined metrics (e.g., RMSE, F1 score); (2) implement the ModelDB API to manage models built using any machine learning environment; (3) query models’ metadata and metrics through the frontend; and (4) track every run of the model and its associated hyperparameters and metrics.

Moreover, we have implemented a generalization of ModelDB to track metrics in any user-defined module through a light API. The DCMetric class contains the following methods:

- DCMetric(metric_name): constructor which takes the name of the metric as a string (e.g., f1 score).
- setValue(value): sets the metric value. The metric can be set multiple times per run but only the final set value is exposed in DC2 Studio.
- DC.register(metric): the defined metric object is registered through this function. Registration is required so the metric is surfaced in the studio.

The following is an example code snippet to track a metric “f1”. First, the metric is defined (line 1). Then, the metric value is set (line 2). The metric value is finally reported to DC2 (line 3).

Figure 1: DC2 Architecture
1 DCMetric metric_f1 = new DCMetric("f1")
2 metric_f1.setValue(f1score)
3 DC.register_metric(metric_f1)

2.3 Visualization

MGH datasets are massive. For instance, the one we use in our demo is 30TB. Because we wanted to enable interactive visualizations at scale, we integrated Kyrix [12], a state-of-the-art visualization system for massive datasets into DC2. With Kyrix, users can write simple code to build intuitive visualization applications that support panning and zooming. The MGH scientists we worked with confirmed that visualization is a key component to make it easier for them to inspect their datasets. While users can write their own visualization applications using the Kyrix API, DC2 comes with a few generic visualization applications: (1) Progress reporting: services report their progress periodically to the Studio through a progress bar; (2) Multi-Canvas Table Views (MCTV): users can click on arcs interconnecting modules on the pipeline to visually inspect the intermediate records passing between the modules and run queries on them (e.g., filter based on predicate); and (3) Coordinated Views: in the MCTV, when users select a record in one canvas, other records are selected on other canvases based on a user-defined function (e.g., provenance, records sharing same key). DC2 comes with an API for easy integration of Kyrix visualization applications in the DC2 Studio (e.g., show a visualization application after clicking on a particular module).

2.4 Debugging Suite

We have seen pipelines that run for hours, so the goal of the DC2 debugger is to catch data-related anomalies (e.g. input data is malformed in one of the modules) early in the workflow execution, so that “bad” data is not passed to downstream processing. DC2 features a set of human-friendly debugging operations to assist users in debugging their pipelines. We implement a GDB-like debugger that is data-driven. Users can add breakpoints by specifying a record or a set of records that satisfy predicates. Pipeline execution is paused upon reaching a breakpoint so that users can inspect visually what is going on so far in the pipeline. The following are the key debugging operations that DC2 provides.

- **filter**: while building a data pipeline, users typically experiment with smaller datasets before testing their pipelines on the entirety of the data. The filter operation allows users to specify a set of predicates to extract smaller subsets from the input datasets. For instance, if the filter is City = “Chicago”, then, only records with City value of “Chicago” will be passed as input to the respective module.

- **track**: an important operation when refining pipelines is to be able to track a set of records to make sure the pipeline is working as expected. Users can specify filters to track records in the pipeline (e.g., track records whose City attribute value is “Chicago”). Whenever a record satisfies the defined filter, it is added to a tracking file which contains (1) the attribute values of the record before and after going through the module; and (2) information related to the module that produced the record (e.g., name of the module, list of parameter values).

- **breakpoints**: users can specify breakpoints in the pipeline using filters. Whenever a record satisfies the filter, the execution is paused to allow the user to inspect the record at the breakpoint. Users can then manually resume the execution.

- **pause/resume**: this is a way for users to pause/resume the execution from the Studio. This functionality is implemented using breakpoints (more details in Sections 2.5 and 2.6). This operation is useful when users only want a certain module to run for a limited period of time (e.g. pause after 5 seconds). When users inspect the output and validate it, then they can resume the execution.

2.5 Manual Breakpoints

Data breakpoints serve as “inspection” points in the pipeline, i.e., they are used to inspect records of interest. For instance, in a deduplication module, if users notice that records whose “City” value is “Chicago” are always incorrectly deduplicated, they can add a breakpoint on records that meet the filter City = “Chicago”, then the pipeline execution is paused whenever a record that meets the filter is encountered. We provide an API to allow users to programatically define functions to set data-driven breakpoints. Those functions are used by the DC2 Studio to allow users to interactively set breakpoints on records that satisfy a given user-provided filter. Three key functions need to be implemented in the entry point file (file containing the DC2 API implementation) to enable manual breakpoints: (1) setBreakpoint which takes as argument a filter (e.g., City = “Chicago”); (2) pause to pause the execution when a record satisfying the filter is encountered; and (3) resume to resume the execution after the user has inspected the records on the breakpoint.

2.6 Automatic Breakpoints

In some cases, implementing the API to enable manual breakpoints can be time-consuming. To address this hurdle, DC2 can create breakpoints in modules automatically (i.e., without requiring users to implement an API). This is done by partitioning the input data (of the module) into different subsets and running the module with each partition. The goal is to be able to detect errors in the output of the module run with fewer partitions than with the entirety of the data. For instance, when running a classification module (with an already trained model), users might want to inspect the output for every 10% of the input data which results in nine breakpoints, i.e., output is shown after 10%, then after 20%, and so on. Additionally, the classification label of a given record does not change whether we run the model with the entire data or only a partition. If users detect misclassified records with a run using 20% of the input data, then, there is no reason to run the module for the remaining 80% records. Moreover, users can specify predicates to create partitions (blocking). For instance, “City = *” would create partitions (or blocks) where records in the same partition share the same value of the “City” attribute. Users can create automatic breakpoints from the DC2 studio.

3. DEMONSTRATION SCENARIO

We demonstrate DC2 through a medical use case with a group of scientists at MGH studying brain activity data captured using electroencephalography (EEG). Figure 2(a) illustrates an example pipeline to clean the EEG data before running it through a machine learning model. In Figure 2(a), each numbered module in the pipeline has its corresponding visualization in Figure 2(b) (e.g., module numbered 1 corresponds to raw data input).

**Study**: Scientists at MGH start with a study goal (e.g., early detection of seizures using EEG data), and then prepare the relevant datasets using cleaning modules. They then apply machine learning models to perform a prediction task. In the case of this demo, they want to predict seizure likelihood given EEG labeled segments. This process is iterative in nature and it takes several iterations to converge to a “good” data pipeline. We helped the MGH scientists clean and then analyze the EEG data using machine learning models. We will walk the audience through how DC2 was used to help
quickly design and execute data pipelines to carry out the study at hand.

**Dataset.** The EEG dataset pertains to over 2,500 patients and contains 350 million EEG segments. The total dataset size is around 30TB. Active learning is employed to iteratively acquire more and more labeled EEG segments as described in the scenario below.

**Scenario.** The demonstration scenario goes as follows: (1) Raw EEG data is cleaned. In addition to the cleaning toolkit that comes with DC2, we plugged the cleaning tools MGH scientists use to clean the data into DC2 as user-defined modules. An example cleaning task is to remove high-frequency signals (e.g., area A in Figure 2(b)); (2) Using the visualization component of DC2, the specialists interactively explore the 30T EEG data and then label the EEG segments based on their domain knowledge; (3) After acquiring a set of manually labeled segments, a label propagation algorithm, as a user-defined component of DC2, automatically propagates labels to the nearby segments of the existing labeled segments; (4) A deep learning model is then learned using part of the labeled segments as training set. During this process, the DC2 debugger is fully explored to tune the hyper-parameters and the network structures; (5) Active learning is then conducted to improve the quality of the automatically acquired labels. First, the labeled segments out of the training set are classified by the learned model. Then using the ModelDB component of DC2 the 2000 segments are efficiently extracted where the neural net had highest confidence but disagreed with the labels; (6) These segments are then fed back into the visualization component for the domain experts to decide whether they need to update their labels (go back to step 3) or review the cleaning step (go back to step 1). This iterative process proceeds until the neural net reaches a satisfactory classification accuracy.

4. REFERENCES


