MIT Open Access Articles

Data Civilizer 2.0: a holistic framework for data preparation and analytics

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation	Rezig, El Kindi, Cao, Lei, Stonebraker, Michael, Simonini, Giovanni, Tao, Wenbo et al. 2019. "Data Civilizer 2.0: a holistic framework for data preparation and analytics." Proceedings of the VLDB Endowment, 12 (12).
As Published	10.14778/3352063.3352108
Publisher	VLDB Endowment
Version	Final published version
Citable link	https://hdl.handle.net/1721.1/137529
Terms of Use	Creative Commons Attribution-NonCommercial-NoDerivs License
Detailed Terms	http://creativecommons.org/licenses/by-nc-nd/4.0/

Massachusetts Institute of Technology

DSpace@MIT

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. Elmagarmid◇

*MIT CSAIL ◇Qatar Computing Research Institute, HBKU
{elkindi, lcao, stonebraker, giovanni, wenbo, madden}@csail.mit.edu
{mouzzani, ntang, aelmagarmid}@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.

PVLDB Reference Format:
El Kindi Rezig, Lei Cao, Michael Stonebraker, Giovanni Simonini, Wenbo Tao, Samuel Madden, Mourad Ouzzani, Nan Tang, and Ahmed K. Elmagarmid. Data Civilizer 2.0: A Holistic Framework for Data Preparation and Analytics. *PVLDB*, 12(12): 1954 - 1957, 2019.
DOI: https://doi.org/10.14778/3352063.3352108

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 hyperparamters).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, 13], 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 [7, 9],

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352108

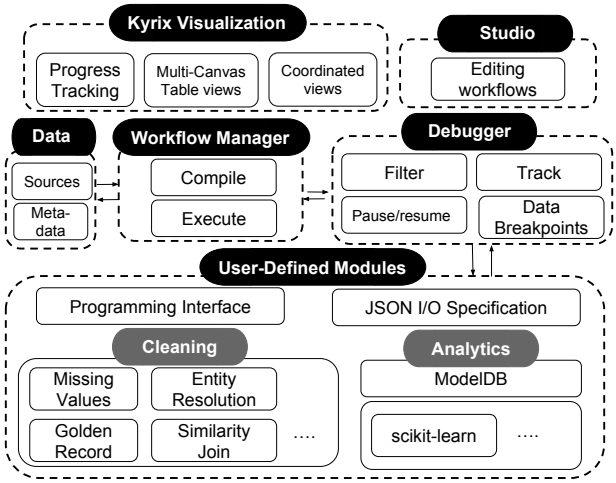

Figure 1: DC2 Architecture

iterative machine learning workflow development [10, 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 can also integrate their data cleaning and machine learning code into *DC2* workflows through a simple API implementation. Users have to simply implement a function that triggers the execution of the module they are adding.
- Debugging: *DC2* features a full-fledged debugger that assists users in debugging their pipelines at the data level and not at the code level. For instance, users can run workflows on a subset of the data, track particular records through the workflow, pause the pipeline execution to inspect output produced so far, and so on.
- Visualization: At the core of *DC2* is a component that allows users to easily implement their own visualizations to better inspect the output of the pipeline's components. We have pre-packaged a few visualizations such as progress bars for arbitrary services, coordinated table views, etc.

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.

executeService. 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 duduplicated 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 [12] 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).

```
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* [11], 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 pipelines on the entirety of the data. The **filter** operation allows 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 (see Sections 2.5 and 2.6). This operation is useful when users want a certain module to run for a limited period of time (e.g. pause after 5 seconds). When users inspect and validate the output, 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 programmatically 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).

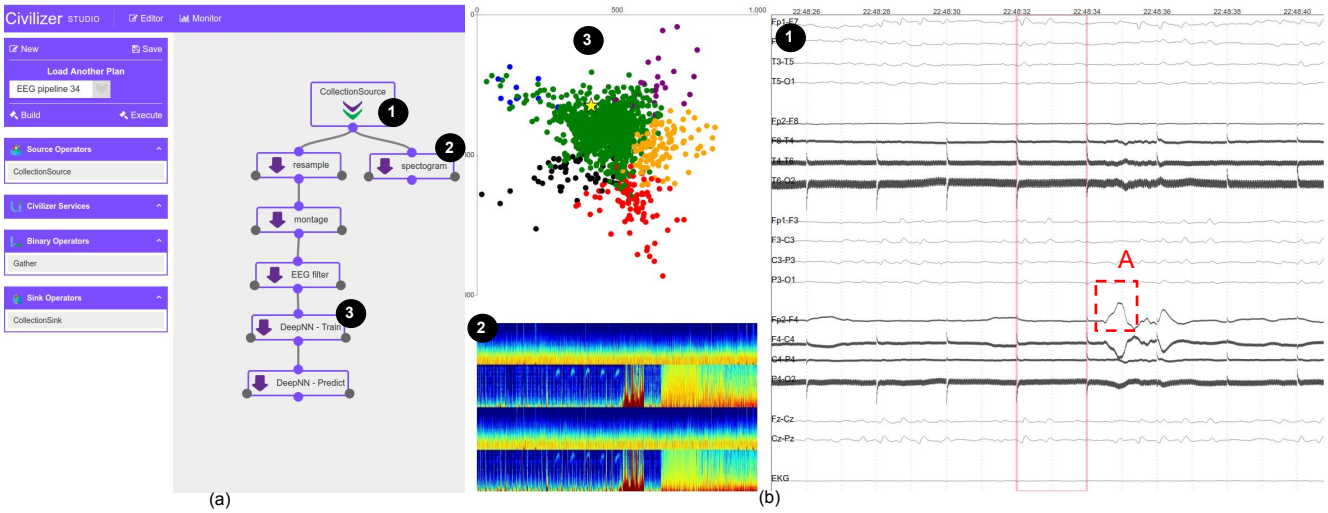

Figure 2: (a) EEG pipeline example. (b) Visualization of numbered components.

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 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 accuracy.

4. REFERENCES

[1] Apache Airflow. `https://airflow.apache.org`. Accessed: March 2019.

[2] mlflow. `https://mlflow.org`. Accessed: March 2019.

[3] R. Castro Fernandez, D. Deng, E. Mansour, A. A. Qahtan, W. Tao, Z. Abedjan, A. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. A demo of the data civilizer system. In *SIGMOD*, pages 1639–1642, 2017.

[4] C. De Sa, A. Ratner, C. Ré, J. Shin, F. Wang, S. Wu, and C. Zhang. Deepdive: Declarative knowledge base construction. *SIGMOD Rec.*, 45(1):60–67, June 2016.

[5] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The data civilizer system. In *CIDR*, 2017.

[6] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. D. Millstein, and M. Kim. Bigdebug: debugging primitives for interactive big data processing in spark. In *ICSE*, pages 784–795. ACM, 2016.

[7] P. Konda, S. Das, P. S. G. C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. F. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.

[8] E. Mansour, D. Deng, R. C. Fernandez, A. A. Qahtan, W. Tao, Z. Abedjan, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, M. Stonebraker, and N. Tang. Building data civilizer pipelines with an advanced workflow engine. In *ICDE*, pages 1593–1596, 2018.

[9] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

[10] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *ICDE*, pages 535–546, 2017.

[11] W. Tao, X. Liu, Ç. Demiralp, R. Chang, and M. Stonebraker. Kyrix: Interactive visual data exploration at scale. In *CIDR*, 2019.

[12] M. Vartak, H. Subramanyam, W. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. Modeldb: a system for machine learning model management. In *HILDA*, page 14, 2016.

[13] D. Xin, L. Ma, J. Liu, S. Macke, S. Song, and A. G. Parameswaran. Helix: Accelerating human-in-the-loop machine learning. *PVLDB*, 11(12):1958–1961, 2018.