Data Science Tasks Implemented with Scripts versus GUI-Based Workflows: The Good, the Bad, and the Ugly

1st Alexander K. Taylor
Computer Science Department
UC Los Angeles
Los Angeles, USA
ataylor2@cs.ucla.edu

1st Yicong Huang
Computer Science Department
UC Irvine
Irvine, USA
yicongh1@ics.uci.edu

3rd Junheng Hao
Computer Science Department
UC Los Angeles
Los Angeles, USA
jhao@cs.ucla.edu

4th Xinyuan Lin
Computer Science Department
UC Irvine
Irvine, USA
xinyual3@ics.uci.edu

5th Xiusi Chen
Computer Science Department
UC Los Angeles
Los Angeles, USA
xchen@cs.ucla.edu

6th Wei Wang
Computer Science Department
UC Los Angeles
Los Angeles, USA
weiwang@cs.ucla.edu

7th Chen Li
Computer Science Department
UC Irvine
Irvine, USA
chenli@ics.uci.edu

***Abstract*—As leveraging large-scale data analytics becomes the norm for many applications, platforms used to develop these capabilities have become increasingly important. In this work, we compare the benefits and drawbacks of implementations of two commonly used data science platform paradigms: code-based scripts and GUI-based workflows. We implement tasks in both paradigms that provide examples of phases in the typical life cycle of a data science project, including data wrangling, machine learning (ML) model training, and inference. We examine the relative performance of the implementations under each paradigm in various experimental settings. We discuss the benefits and drawbacks associated with each platform implementation and provide a foundation for future work in comparing data science platform paradigms.**

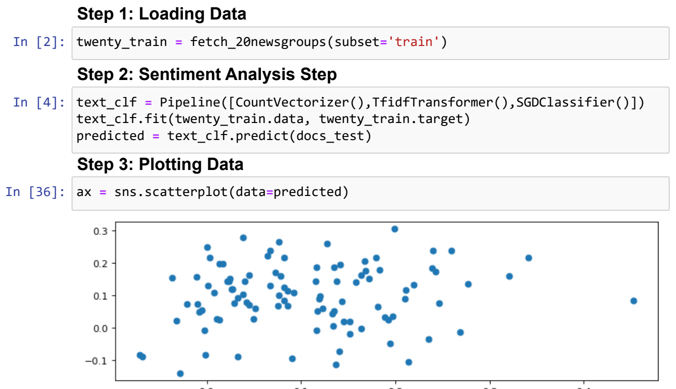
***Index Terms*—Data Science, Workflows, Jupyter Notebook, Comparison, Scalability**

I. Introduction

The demand for the development of analytics capabilities has grown tremendously in recent years due to continued improvements in leveraging data using various techniques such as big data and machine learning (ML). As such, the need for platforms to support data science projects has grown in parallel. We have observed a rising trend in the data science community to use various paradigms for implementing and conducting data science tasks, such as scripts, GUI-based workflows, and spreadsheets [1], [2].

This study focuses on comparing implementations of the script-based paradigm and GUI-based workflow paradigm through typical data science tasks. Although either platform can accommodate these tasks, each offers a distinct experience concerning task development, execution, and scalability.

Script-based paradigm. Writing a code-based script in their choice of language (e.g., Python) is a natural method of conducting a data-processing task for software developers. The script format requires knowledge of the implementation language, and provides users the medium to implement tasks with few restrictions and control over the execution of the finished script. Users are able to execute components of the task in their specified order and are able to automate the extension of the script to as many data files or datasets as the user desires. Figure 1 shows an example script, in



Step 1: Loading Data

```
In [2]: twenty_train = fetch_20newsgroups(subset='train')
```

Step 2: Sentiment Analysis Step

```
In [4]: text_clf = Pipeline([CountVectorizer(),TfidfTransformer(),SGDClassifier()])
        text_clf.fit(twenty_train.data, twenty_train.target)
        predicted = text_clf.predict(docs_test)
```

Step 3: Plotting Data

```
In [36]: ax = sns.scatterplot(data=predicted)
```

Fig. 1: An example Jupyter notebook script that trains, evaluates, and plots the results of a sentiment analysis model.

which a sentiment analysis model is trained, evaluated, and the predicted results are plotted. There are several popular script-based platforms for developing code in languages such as Python and R. In this paper, we focus on Python and implement tasks using Jupyter Notebook [3], due to the platform's huge popularity. For instance, Jupyter Notebook is used in 11 million of the roughly 28 million public GitHub repositories. Jupyter Notebook provides a script-based tool for users to both implement and present the results of data science tasks.

GUI-based workflow paradigm. Graphical user interface (GUI)-based workflow systems, or "workflow systems" for short, such as Alteryx [4], Knime [5], RapidMiner [6], Einblick [7], and Texera [8], allow users to construct and execute data science workflows using a visual, intuitive interface. A workflow example implemented in Texera is shown in

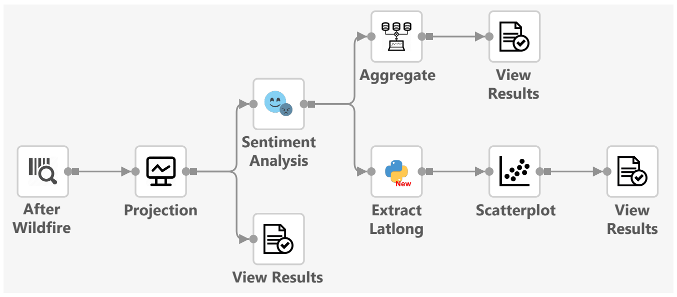


Fig. 2: A GUI-based workflow in the Texera system for sentiment analysis on post-wildfire tweets. The workflow forms a directed-acyclic graph (DAG) comprising operators linked by edges. The operators handle data received from upstream operators through input edges and transmit produced data to downstream operators via output edges.

Figure 2. In this example, similar to Figure 1, a sentiment analysis model is trained, evaluated, and the predicted results are plotted. Workflow systems enable users to implement complex data science tasks without the need for programming, making them particularly advantageous for non-IT users with limited coding skills. We are using Texera [8] as an example of GUI-based workflow systems. Texera adopts a modular approach with operators serving as the basic building blocks of workflows. A broad range of operations are supported by this system, ranging from simple filtering and projection to visualization techniques. Moreover, Texera allows the execution of a workflow to use multiple machines in a cluster.

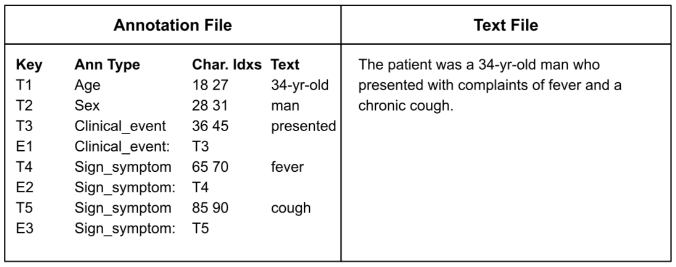
In this work, we conduct a comparison of these two platforms using Jupyter notebook and Texera as representatives. Our comparison revolves around real-world use cases, with a particular emphasis on essential stages in data science, including data wrangling, model training, and model inference. We present the benefits and drawbacks of the paradigms in terms of ease of use, modularity, scalability, and parallelization.

II. Data Science Tasks

In this section, we describe four tasks to compare Texera and Jupyter Notebook to cover common steps in data science.

A. Task 1: DICE (Data Wrangling)

This task represents a fairly complicated data wrangling procedure. The goal of the task is to pre-process biomedical data with a novel ML-based event extraction (EE) technique called "data-efficient clinical event extraction" (DICE) [9]. Figure 3 shows a sample of a data set called "MACCROBAT",



Annotation File				Text File
Key	**Ann Type**	**Char. Idxs**	**Text**	The patient was a 34-yr-old man who presented with complaints of fever and a chronic cough.
T1	Age	18 27	34-yr-old	
T2	Sex	28 31	man	
T3	Clinical_event	36 45	presented	
E1	Clinical_event:	T3		
T4	Sign_symptom	65 70	fever	
E2	Sign_symptom:	T4		
T5	Sign_symptom	85 90	cough	
E3	Sign_symptom:	T5		

Fig. 3: Sample of the annotations and sentences contained in the MACCROBAT dataset. Entity annotations are denoted as T_i and event annotations are denoted as E_i.

which consists of 200 pairs of text files of clinical case reports with accompanying files of annotations such as events and entities. DICE takes in MACCROBAT and constructs an output dataset called MACCROBAT-EE by linking each sentence to its respective annotations.

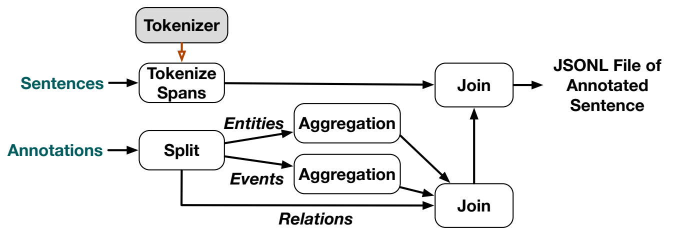


Fig. 4: Detailed steps of the DICE data wrangling task. The annotation files and text files are first processed separately before sentences are linked with their respective annotations.

As shown in Figure 4, the DICE task requires filtering event annotations based on certain conditions, and joining a subset with entity annotations. The results are then rejoined with the held-out subset of events, and each of these collections of annotations is joined with its respective sentence. We choose this task as an example of data wrangling as it involves complex extraction and join operations over text data in different formats, which is a crucial aspect of data preparation in data science.

B. Task 2: WEF (Model Training)

This task, called "Wildfire Experience Framing" (WEF), performs a multi-label classification over a dataset of 800 human-expert-labeled tweets related to climate change during the onset of 20 wildfires in California between 2017 and 2021. Each tweet has one to four climate framings, namely making explicit links between wildfires and climate change, suggesting climate actions, attributing climate change to adversities besides wildfires, or being labeled as not relevant. WEF fine-tunes four pre-trained BERT models [10] to classify whether each tweet belonged to a given framing. As a typical machine learning training task, WEF provides insights into the training of machine learning models under both paradigms, as shown in Figure 5. We choose this task as an example of the typical training step in data science.

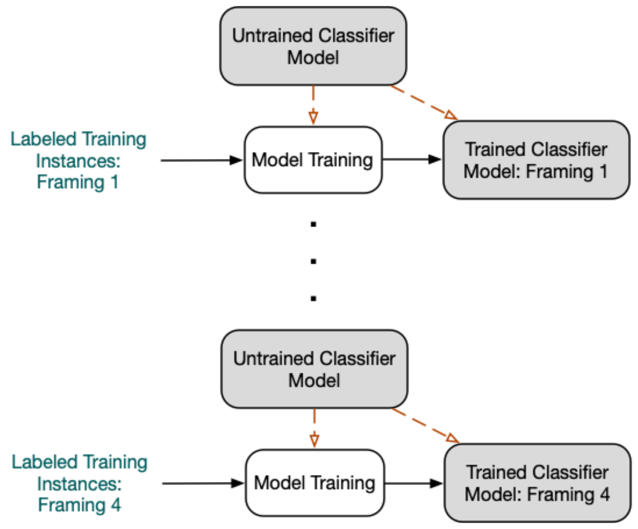

Untrained Classifier Model
Labeled Training Instances: Framing 1
Model Training
Trained Classifier Model: Framing 1
Untrained Classifier Model
Labeled Training Instances: Framing 4
Model Training
Trained Classifier Model: Framing 4

Fig. 5: WEF is an ensemble machine learning pipeline that trains four binary classification models to perform multi-label classification for the four wildfire framings. Each framing model is labeled 1-4 above.

C. Task 3: GOTTA (One-Step Inference)

Generative prompt-based data augmentation (GOTTA) [11] is a novel model of few-shot question-answering (FSQA), which aims to predict answers to a set of questions from passages in a setting with limited resources. As shown in Figure 6, GOTTA augments a question-answering training data set with cloze text [10] to force the model to understand the context of the data beyond the original questions. This task utilizes a BART model [12] that has been fine-tuned to the FSQA task. The model takes a paragraph and several cloze text questions as the input, and returns generated responses as the output. GOTTA is a typical machine learning inference task, which consists of preparing questions and their answers based on input data, applying a forward-pass of a trained model to batched input, and evaluating the correctness of the output. We choose this task as an example of one-step inference because, similar to many machine learning models, it requires only a forward-pass of data through the model to generate predictions.

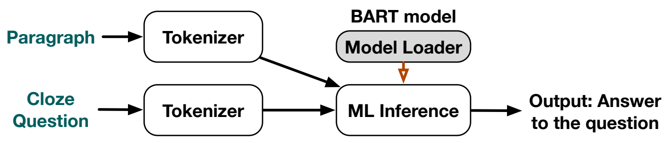


Fig. 6: GOTTA is an example task of black-box machine learning inference, where the trained model is applied to input and returns predictions.

D. Task 4: KGE (Multi-Step Inference)

This task is about triple prediction via knowledge graph embeddings [13] ("KGE" for short) on graph machine learning. The KGE task takes candidate Amazon products as the input, and uses a pre-trained knowledge graph model of a particular Amazon user to predict the products that the user is likely to purchase in the future. The KGE task is a typical data science inference job.

As shown in Figure 7, the task consists of multiple mini-steps. First, each product candidate goes through a filter, which removes candidates that are no longer available, e.g., those that are out-of-stock. A knowledge graph embedding table is then loaded into memory and used to match each product with its corresponding embedding. The product embeddings are then scored, ranked, and fed through a reverse lookup function to return the most likely products that the user would purchase in the future. We choose this task as an example of multistep inference generation since it provides insight into the impact of the platform used to implement models that require operations in addition to the forward pass to generate predictions.

III. Comparative Analysis of Jupyter Notebook and Texera

In this section, we undertake a comparative analysis of the chosen workflow and script paradigms examples, Texera and Jupyter Notebook, respectively, and exemplified by the chosen implementations for the tasks introduced in Section section II. By comparing these two paradigms, we aim to gain insights into their strengths and limitations in the context of data science.

A. Aspect #1: Abstraction

The level of abstraction in a paradigm plays a crucial role in conveying information to the user during the implementation of a data analysis task. In comparing the levels of abstraction between Texera and Jupyter Notebook, we assess how each platform represents the task, reports execution progress, and presents the data, highlighting the variations introduced by different paradigms.

Presentation of a Task. The first level of abstraction is how a task is being presented to the user. In Jupyter Notebook, the user's code is presented in a top-down, sequential manner, as demonstrated in the example in Figure 1. This presentation style offers a detailed view of the task's implementation, displaying executable code or mark-up text within each cell along with the cell's output. On the other hand, the workflow-based paradigm adopted by Texera provides a graphical user interface (GUI) that offers a high-level abstracted representation of the workflow as a graph, showcasing the flow of data through various operators, as shown in Figure 2. Optionally, users can choose to elaborate a particular operator and even view the code if desired. Jupyter Notebook also allows users to collaborate on individual operators simultaneously. Jupyter Notebook does not explicitly show the relationship between functions and cells, users can choose to execute the cells in an

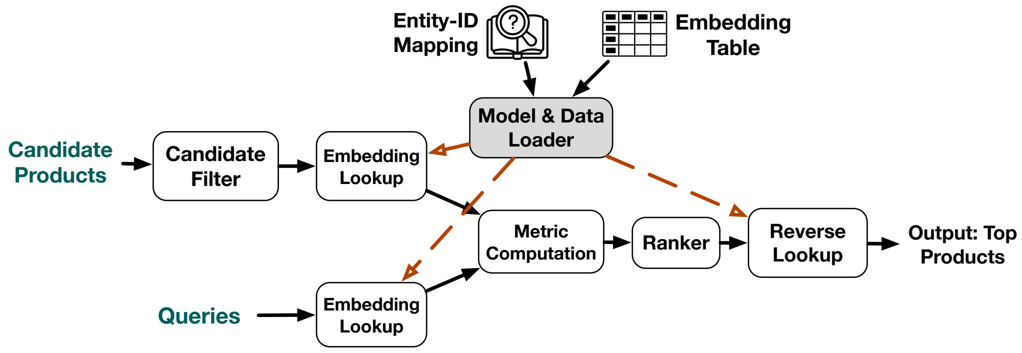
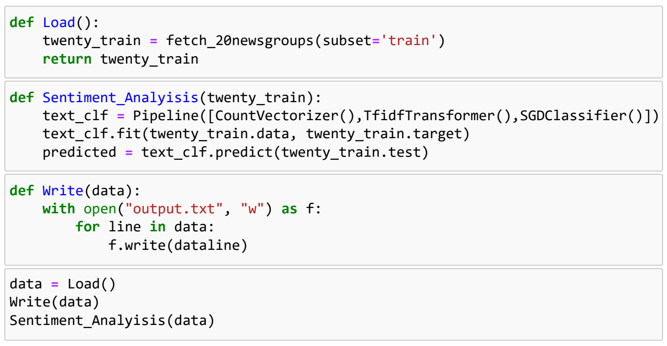


Fig. 7: Detailed steps of the KGE task. Products are filtered by relevance and matched with users deemed most likely to purchase them in the future.

arbitrary order, with the state stored in the kernel being used by different cells implicitly. Texera workflows, on the other hand, consist of operators with explicit connections that indicate data flow. Each set of expectations provides different experiences when multiple users collaborate on the same task. The level of detail in the presentation approach taken by Jupyter Notebook is suitable for presenting and editing implementation specifics, while Texera is more suited to visualizing high-level abstractions of data flow; both applications serve valuable purposes in a data science project and are accessible to collaborators with varying levels of expertise.

Displaying the state of a task. The second level of abstraction is how a task's runtime status is displayed. In Jupyter Notebook, the execution progress of a task is presented by displaying a loading icon, indicating the currently running cell, and utilizing a sequential execution counter to label the order of cell execution. Additionally, users have the option to incorporate external libraries like "tqdm" in Python to enhance the user experience further. These libraries enable the display of a progress bar, providing a visual representation of the task's completion status. This progress bar offers a time-based indication of the progress being made during the task execution. In contrast, Texera focuses on displaying data progress rather than time progress. The Texera interface utilizes different colors to visually represent the status of each operator. It indicates various states, including initializing, running, paused, and resumed. Furthermore, Texera also provides information about the amount of data being processed by each operator, offering a clear depiction of the data progress within the workflow. In addition to displaying the progress of a task, another consideration is how error traces are presented. Jupyter Notebook presents a stack trace at the cell level, allowing users to track the steps of execution to the earliest function call in the cell. The workflow-based paradigm reports error traces at the operator level, meaning that only the operator in which the issue occurred reports the error. Both approaches allow users to identify and isolate the problematic cell or operator to perform debugging.

Visualizing data linkage of a task. The third level of abstraction is how a task's data lineage is presented. Data linkage refers to the connection and flow of data between different steps or components. In Jupyter Notebook, data linkage is typically established through programming functions that define the input and output of each step. The script is then executed cell by cell, without imposing a specific order on the execution sequence. Note that the order of executable code cells may not necessarily align with the actual flow of data. As illustrated in Figure 8, although the function "Write" is defined after "Sentiment_Analysis", the user may choose to execute "Write" *before* "Sentiment_Analysis". This flexibility allows users to freely orchestrate their implementation, but it may result in an inaccurate representation of the applied steps to the input data In contrast, Texera executes the operators of



```
def Load():
    twenty_train = fetch_20newsgroups(subset='train')
    return twenty_train
```

```
def Sentiment_Analyisis(twenty_train):
    text_clf = Pipeline([CountVectorizer(),TfidfTransformer(),SGDClassifier()])
    text_clf.fit(twenty_train.data, twenty_train.target)
    predicted = text_clf.predict(twenty_train.test)
```

```
def Write(data):
    with open("output.txt", "w") as f:
        for line in data:
            f.write(dataline)
```

```
data = Load()
Write(data)
Sentiment_Analyisis(data)
```

Fig. 8: Example showing that script functions in Jupyter Notebook can be executed in any arbitrary user-defined order.

a workflow based on the order specified in the directed acyclic graph (DAG) of operators. This approach necessitates that each operator explicitly defines its input data and output data. Users are required to explicitly connect operators with links that represent the flow of data. An execution of a visualizing task in Jupyter Notebook is dependent on a user-chosen order and is represented linearly, which is useful for communicating the progression of execution. On the other hand, Texera visualizes an execution as a DAG, which provides an interactive, high-level representation of the task.

B. Aspect #2: Execution Customization vs. Ease of Use

The second comparison we consider is the amount of environmental and methodological control that the platform provides to users. Jupyter Notebook allows users to build frameworks according to their own specifications, not encumbering the implementation with constraints but leaving the user responsible for ease of future use. Texera, on the other hand, requires the user to develop their framework in a set of iterative operators, which is less flexible but results in a pipeline with a more accessible user interface.

User control of Implementation. Users that intend to increase the scale of processing data when developing a Jupyter Notebook on a single machine need to manually build the support infrastructure, such as data partitioning, result aggregation, into their code. For instance, consider the DICE task. A straightforward approach that can be implemented using Jupyter Notebook is to load the set of annotations into memory as a hash table and loop through the sentences while probing the annotation hash table to match sentences with their corresponding annotations. Texera currently does not support global variables (i.e., variables available to all operators) and requires explicit data passing between operators. Thus, it prevents the usage of a global annotation table, and instead requires passing copies of both the list of sentences and annotation table through each operator in which they are needed. Jupyter Notebook does not place restrictions on user control of a script beyond those placed on a typical Python script. On the other hand, Texera requires users to adhere to its pre-defined explicit data passing structure, which reduces customization options for the execution of the task in return for a structured user interface and pipelined operator execution as shown in Figure 9.

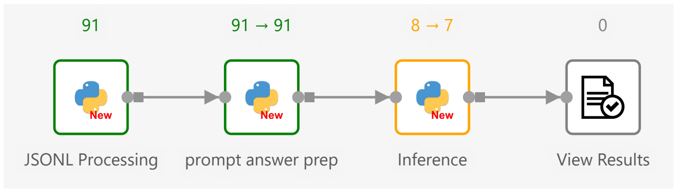


Fig. 9: Visual representation of a simple Texera workflow displaying the data processed by individual operators during execution. Each operator features two numbers, denoting the number of input tuples and output tuples, respectively. The source operator (JSONL Processing) only shows the output-tuple count, while the sink operator (View Results) only shows the input-tuple count.

Tuning resource usage for data batching and module parallelism. Another comparison we consider is how each platform is able to leverage available computational resources. Jupyter Notebook requires the user to specify how their process leverages computational resources and manually search for an optimal configuration for each new environment the process is run in. Texera automates the tuning of computational resource usage configurations, thus removing the burden from the user. For instance, users are able to parallelize operators by selecting the number of workers used to execute each operator and the Texera backend manages the distribution of work amongst computational resources. Consider the batching done during in subsection II-C for the GOTTA inference task, which, in the Jupyter Notebook, is implemented by the user leveraging imported functions (e.g., PyTorch) as shown in Figure 10. This batching method requires the user to manually

```
for context in data:
    for qa in question_answers:
        question = qa["question"]
        answers = qa["answers"]
        answer = f"Question: {question} Answers: {answers}"
        prompt = f"Question: {question} Context: {context}"
        test_dataset = TextDataset(test_dataset,...)
        val_params = {...}
        test_loader = DataLoader(test_set, **val_params)
```

Fig. 10: Explicit construction of a batched dataset.

tune the batch size for the given environment. Batching in Texera can be done by using an operator to construct each input (question, masked answer, paragraph) and passing the individual input to the subsequent operator in a batch size that Texera tunes to the available computational resources. Jupyter Notebook requires users to manually configure their memory usage, level of module parallelism, and configure their work to effectively leverage the hardware. In comparison, Texera enforces iterative data processing so that it is able to automatically tune resource usage, removing the burden of management of parallelism and hardware usage from users.

C. Aspect #3: Supporting Multiple Programming Languages

The third comparison we consider is how each platform leverages programming languages to implement the task. Both Jupyter Notebook and Texera benefit from common functions that are efficiently implemented and highly reusable, e.g., Python libraries. Jupyter Notebook typically leverages packaged software written in Python. Although it is possible to use libraries in C (e.g., numpy), it takes a substantial amount of user effort to write code in another language. On the other hand, Texera operators can be implemented in languages designed for the desired task, e.g., Python for data science tasks, Scala for functional programming, etc.

Incorporation of multiple languages. There is a need for data science platforms to support different languages to support collaborations because users with different domain backgrounds may use different languages (e.g., Python for ML, R for statistics, Julia for Bioinformatics, etc.). Jupyter Notebook requires users to implement code in Python, but allows for processes written in other languages to be launched during the execution of the notebook; this allows scripts written in other languages to be incorporated into a data science task, but requires that users manually define data passing between scripts. Texera allows for operators written in languages other than Python (e.g., Java, Scala, etc.) to be incorporated into user-defined workflows with no additional overhead. Both Jupyter Notebook and Texera support the execution of processes written in multiple languages to cater to cross-domain collaboration, however Texera provides a lower barrier-to-entry because it provides cross-language data handling out-of-the-box.

Impacts on collaboration. We also consider how the flexibility of each platform with respect to languages used in implementations impacts collaboration. Collaborations on data science projects typically involve individuals with diverse backgrounds and skill sets. Jupyter Notebook does not provide a framework for collaboration between two users working in different languages other than saving intermediate data between steps and transferring it manually. As previously mentioned, Texera is able to incorporate modules written in different languages. This capability allows users from different backgrounds to implement tasks in the language they find the most suitable. Both platforms are able to incorporate modules written in other languages, however, Texera provides an infrastructure for data transfer between modules written in different languages without additional user intervention.

D. Aspect #4: Performance on Large Data

Lastly, we compare the performance of each platform as the data size increases. We consider several factors that impact the scalability of both platforms, such as the overhead, pipelining/parallelism, hardware utilization, and language efficiency.

Runtime overhead. Jupyter Notebook does not require data serialization and deserialization between code blocks and can be executed similarly to a standard Python script. Texera is implemented in Scala and supports operators written in multiple languages. When using Texera, data transfer occurs between these operators, requiring serialization and deserialization processes to bridge the gap between different languages, which introduce runtime overhead.

Pipelining/Parallelism and resource usage. The Jupyter Notebook requires the user to incorporate multi-threading or multiprocessing to optimize the usage of computational resources for the data processing task. To fully leverage the available resources, users typically rely on available packages to orchestrate multi-threading or multicore resource usage, such as "ray" [14] in Python. In contrast, Texera workflows are able to pipeline the execution of data tuples across operators without user intervention. In particular, two operators can process different tuples at the same time. Texera also provides operator-level parallelism without any user intervention.

Language Efficiency Another aspect of comparison we consider is the language-based efficiency of modules implemented in Jupyter Notebook and Texera. For instance, consider the join step of the KGE task subsection II-D, which loads an embedding table into memory and probes the table for the embedding associated with each product. The join step is a bottleneck with respect to time in this task, and Jupyter Notebook users are able simply call the Pandas function *dataframe.merge* to leverage a Python implementation of the join step. While this option is provided to Texera users, Texera also provides users a join operator written in Scala, which is a more efficient language for this task, as shown in Figure 11. This allows users to leverage more efficient implementations of subtasks without needing to implement cross-language data transfer. However, this requires that Texera provides an operator or set of operators off-the-shelf that can accomplish a logically similar task. Both Texera and Jupyter Notebooks are able to incorporate packages written in more efficient languages, and Texera is able to pass data between functions written in different languages in a single workflow.

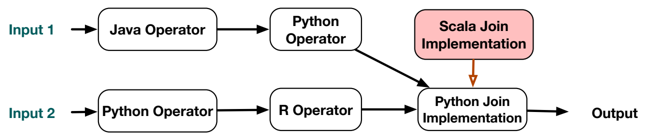


Fig. 11: A Texera workflow can contain operators implemented with multiple languages. Operators can be replaced by operators implemented in other languages.

The platform-specific benefits and drawbacks discussed earlier become more apparent as the input data size increases. Jupyter Notebook offers users implementation flexibility and reduced execution overhead. Texera provides automated pipelining, and resource optimization.

IV. Experiments

In this section, we show the results of the experiments used in our comparison of Jupyter Notebook and Texera.

A. Experimental Setup

We ran experiments on two computing clusters on the Google Cloud Platform [15]. Each cluster had four virtual machines, each of which had 8 vCPUs, 64 GB RAM, and 100 GB HDD. We used Python of version 3.8.10, and PyTorch of version 1.12.1 for all the experiments. The following are the details of the two clusters.

Ray-cluster. This cluster was used to conduct Ray-based experiments. Ray is a Python-based framework for scaling data science applications using computational resource clusters. We used one machine as the head node of the cluster to host the Ray service, and four machines as the worker nodes. The scripts were submitted from a command line interface on the head node. We recorded the duration of each script, from the time it was submitted to the Ray server until the return of results.

Texera-cluster. This cluster was used to conduct Texera-related experiments. One machine was used as the controller node, which hosted the Texera service and its web server for its graphical user interface (GUI). We used four machines as worker nodes connected to the controller. We initiated tasks from the Texera GUI on the controller node, and recorded the duration of each task, from the time it was submitted on the GUI until completion of all the workers and return of results to the controller.

Worker configuration. In the experiments, Ray relied on a resource pool for its scheduler to allocate workers and Texera users explicitly configured the number of workers for an operator. The only way to change the number of workers in Ray was to configure the number of CPUs that Ray could use [16], [17]. To have a fair comparison with the one-worker setting in Texera, we configured Ray's `num_cpus` parameter to 1. Ray configured the underlying frameworks (PyTorch) to use 1 CPU by default to limit contention for resources [16], [17]. The Texera implementation did not limit resource contention or the underlying frameworks (PyTorch).

B. Performance Metrics

We use the following metrics in the experiments:

- **Total execution time**: this metric provides an indication of how efficient each task is;
- **Number of parallel processes**: this metric measures the parallelism of the execution and is used together with the previous metric to provide an insight into the efficiency;
- **Number of lines of code**: this metric provides a measure of how much work is needed to implement a task.
- **Number of operators**: this metric measures the number of subtasks each task can be divided into.

C. Experiment #1: Modularity

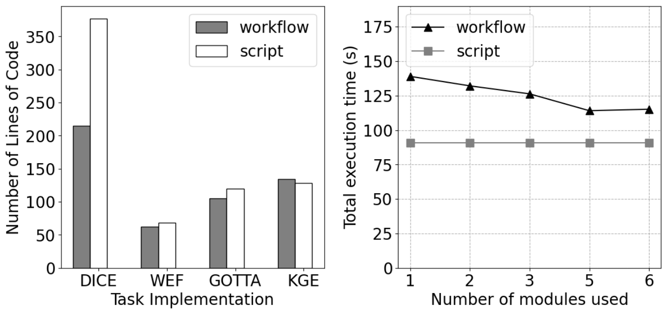
Level of modularity. We measured the relationship between the number of modules that a task is implemented by and the total execution time in both Texera and Jupyter Notebook. The KGE inference task was the only task we performed experiments in which separating and combining subtasks could be performed without changing the task's logic, thus it is the only task shown in this experiment. As shown in Figure 12b, we saw a negatively-correlated linear trend in the execution time associated with the increase in the number of logically separable components used to implement the task, or a greater degree of modularity, with diminishing returns. Specifically, the 1-operator workflow took 138.97 seconds and the 5-operator workflow took 114.05 seconds (19.70% faster) for the input data of 6.8k products. In contrast, the 6-operator workflow took 115.143 seconds (0.95% slower). This result supports the claim that Texera benefits from data pipelining, i.e., starting the execution of the subsequent operator before the current operator has processed all its data points, to offset the overhead it introduces when transferring the data between operators until a task-specific threshold of modularity.

Total Lines of Code. We measured the number of lines of code used to implement each task in both Texera and Jupyter Notebook. We observed that for the DICE, WEF, GOTTA, and KGE tasks, the Jupyter Notebook implementations required 377, 68, 120, and 128 lines, respectively, and the Texera implementations required 215, 62, 105, and 134 lines, respectively.

	6.8K pairs	68K pairs
Time for Scala-based operators (s)	98.67	1,159.82
Time for Python-based operators (s)	126.28	1,170.57

TABLE I: Comparison of KGE execution times of swapping Python-based operators and Scala-based operators.

This result validates the assertion that Texera can implement the tasks in similar or fewer lines as the Jupyter Notebook implementation. Texera benefited from the data pipelining as shown in the previous section.



(a) (b)

Fig. 12: Comparison of the modularity of tasks. (a) Number of lines of code in Jupyter Notebook and Texera implementations, and (b) KGE execution time on different numbers of workflow operators (the time for script is included for reference).

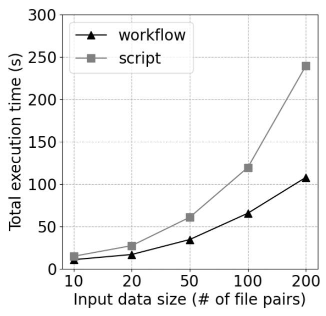
D. Experiment #2: Language Efficiency

We used the KGE inference task to evaluate the impact of languages on the performance. We employed two types of operators in Texera: Scala operators and Python operators. We chose a KGE Texera implementation with three Python operators. To construct the corresponding workflow with Scala operators, we replaced one of the three Python operators performing table joins with nine Scala operators to implement the same logic. We compared the performance of these two Texera workflows.

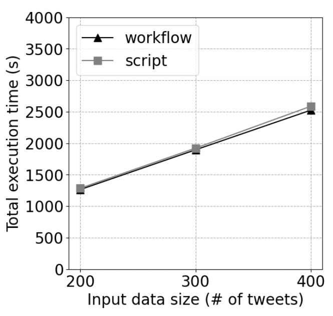
As shown in Table I, the Scala-based KGE workflow was 24.54% and 0.92% faster than the Python-based KGE workflow for the data set of 6.8k products and the data set of 68k products, respectively. This result indicated that the workflow-based implementation benefited modestly from the efficiency of the operators written in Scala when the input data size was smaller, and this efficiency did not scale as the input data size increased.

E. Experiment #3: Scaling Dataset Size

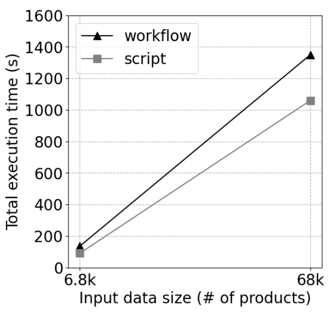
We examined the performance of the four data science tasks over datasets of different scales to compare how the two paradigms perform as the dataset size increased.



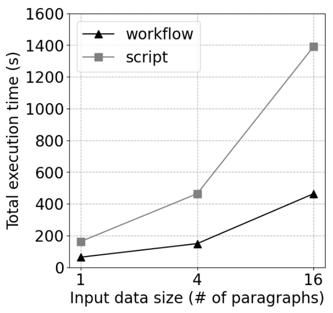
(a) DICE



(b) WEF



(c) KGE



(d) GOTTA

Fig. 13: Comparing execution time as the data size increased.

DICE. The execution time of the Jupyter Notebook followed a roughly linear curve, while the execution time of the Texera workflow followed a roughly logarithmic curve. As shown in Figure 13a, at the smallest and largest dataset sizes of 10 and 200 text and annotation file pairs, the Jupyter Notebook implementation took 14.71 and 239.54 seconds. On the other hand, the Texera workflow took 10.73 and 107.83 seconds (37.12% and 122.15% slower), respectively. This performance gain was due to the pipelining execution in Texera (as discussed in subsection III-D).

WEF. As shown in Figure 13b, the execution time of the Jupyter Notebook implementation and Texera implementation of WEF both followed a roughly linear curve and achieved similar performances. The Jupyter Notebook implementation took 1285.82, 1922.86, and 2587.94 seconds. The Texera workflow took 1264.93, 1896.01, and 2525.96 seconds (2%, 1%, and 3% faster) to train the model on 200, 300, and 400 tweets, respectively. This result was expected as machine learning training tasks are CPU intensive rather than data incentive. Since WEF did not use a distributed training algorithm, each paradigm was executing it with no parallelism.

GOTTA. The execution time of the Jupyter Notebook implementation and Texera implementation of GOTTA followed a roughly logarithmic curve. As shown in Figure 13d, for the dataset size of 1, 4, and 16 paragraphs, the Jupyter Notebook implementation took 163.22, 463.96, and 1389.93 seconds. The Texera workflow took 64.14, 149.45, and 460.13 seconds (151%, 211%, and 201% faster), respectively. This result was due to two reasons. The first was that Ray required uploading large objects such as models into an object store [14], which required a lot of memory and added execution time for each access. On the other hand, the Texera implementation loaded the model and distributed it through the network to each worker, which resulted in a lower overhead compared to Ray's shared object space. The second reason was that Ray limited PyTorch to 1 CPU, while Texera did not enforce such a limitation on PyTorch.

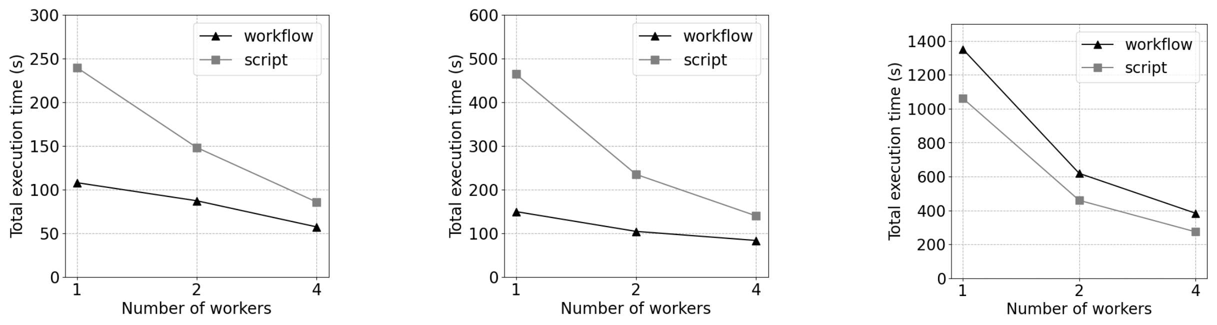
KGE. As shown in Figure 13c, the execution time of both the Jupyter Notebook and Texera workflow implementations of KGE followed a roughly linear curve. The Jupyter Notebook took 90.69 and 975.46 seconds. The Texera workflow took 135.85 and 1350.50 seconds (33% and 28% slower, respectively) at the two data scales. In contrast to GOTTA, the performance of KGE using Jupyter Notebook was less impacted by the overhead of Ray's shared object space due to the fact that the KGE model was 375 MB, which was much smaller than the GOTTA model (1.59 GB). Furthermore, the KGE model did not use parallel subprocesses.

F. Experiment #4: Number of workers

We examined the performance of DICE, GOTTA, and KGE when we assigned different numbers of workers to compare the two paradigms. We excluded WEF from this experiment because under this setting WEF becomes a distributed training task, which is not the focus of this work.

DICE. The execution times of the DICE Jupyter Notebook and Texera implementations followed a roughly linear curve. As shown in Figure 14a, for 1, 2, and 4 workers, the Jupyter Notebook implementation took 239.54, 148.04, and 85.65 seconds respectively, compared to the Texera workflow that took 107.82, 87.13, and 57.21 seconds (122%, 70%, and 50% slower), respectively. The initial performance difference was due to the pipelined execution in the Texera implementation. When multiple workers were used, the Jupyter Notebook implementation was able to reduce the difference in execution times by roughly 50%. The Texera implementation still outperformed the Jupyter Notebook implementation.

GOTTA. The execution times of the GOTTA Jupyter Notebook and Texera implementations followed a roughly linear curve. As shown in Figure 14b, for 1, 2, and 4 workers, respectively, the Jupyter Notebook implementation took 463.96, 234.68, and 139.66 seconds respectively, compared to the Texera workflow that took 149.45, 104.16, and 83.37 seconds (210%, 125%, and 67% slower), respectively. The initial performance difference was due to the fact that the Jupyter Notebook implementation relied on the shared object space, which was less efficient for the one-worker setting. However, as additional workers were introduced, the Jupyter Notebook implementation was able to reduce the relative difference in the execution times by roughly 70%. The Texera implementation still outperformed the Jupyter Notebook implementation.



(a) DICE (b) GOTTA (c) KGE

Fig. 14: Comparison of execution time as the number of workers increased.

KGE. The execution times of the KGE Jupyter Notebook and Texera implementations followed a roughly linear curve. As shown in Figure 14c, for 1, 2, and 4 workers respectively, the Jupyter Notebook implementation took 975.46, 459.46, and 273.89 seconds compared to the Texera workflow that took 1350.50, 618.39, and 383.58 seconds (28%, 26%, and 29% slower), respectively. We observed that the Jupyter Notebook implementation consistently outperformed the Texera workflow and that both implementations showed intuitive reductions in runtime as the number of workers was increased for this task.

V. Related Work

Data science work practices. Many papers [18]–[20] discussed common practices to conduct a data science task. These studies show that a data science task comprises numerous stages such as preparation, modeling, and deployment, where each stage consists of multiple steps that require collaboration among individuals with diverse backgrounds. This insight motivates our evaluation of the two paradigms using examples.

Evaluations of workflow systems. Data-processing systems such as Spark [21] and Flink [22] have been evaluated in many works [23], [24]. These implementations have been evaluated on performance metrics, but have not been compared to GUI-based workflow systems based on these metrics. Similarly, GUI-based workflows such as RapidMiner [6] and KNIME [5] have been most evaluated and compared for their user experiences [25], [26]. Additionally, some studies have specifically evaluated the parallel performance of each individual system [27], [28] rather than comparing it to notebooks. In this work, we contribute a holistic evaluation of both the user experience and the performance of the GUI-based workflow system Texera and Jupyter Notebook.

Evaluations of Jupyter Notebook. Several works [29]–[32] have evaluated the use of the Jupyter Notebook platform with respect to its use in data science projects. Many works evaluate Jupyter Notebook on the basis of its use as a collaborative tool and focus mainly on qualitative issues with reusability and interpretability [29], [31], [32]. Other studies surveying data scientists focus on improving user practices with Jupyter Notebook, i.e. instilling guidelines for users to follow [30]. In this work, we present a novel comparison of the Jupyter Notebook platform with a GUI-based workflow platform, Texera.

Parallel frameworks. Data science tasks incorporate many frameworks that manage parallelism and distributed computing, such as Ray and PyTorch [14], [16], [17]. These frameworks are subsumed by both the script-based paradigm and workflow-based paradigm, as exemplified by our experiments.

VI. Conclusions

In this paper, we performed a comparison between examples of the script-based and workflow-based paradigms for conducting data science tasks using Jupyter Notebook and Texera. We noticed that in settings with low computational resources, Texera performs well and is able to improve performance as resources scale up. We also find that Jupyter Notebook is able to achieve large relative performance improvements as more computational resources are used. Texera provides automated pipelining, optimization for iterative processes, and ease of incorporating operators in efficient languages. These optimizations allow Texera users to achieve similar or improved performance compared to the Jupyter Notebook implementations of the same task (in terms of execution time). Meanwhile, Texera users need to implement their tasks according to the specifications established by the platform. On the other hand, users of Jupyter Notebook can manually incorporate the same multi-processing functionality as Texera provided. Jupyter Notebook users are able to implement tasks with no constraints imposed by the platform. Our results provide empirical support for showing that Texera is an alternative to Jupyter Notebook for data science tasks and can, in some cases, outperform Jupyter Notebook in terms of execution time. Future work in this vein will be able to use this study as a foundation to build comparisons of platforms for data science projects.

Acknowledgment

This work was funded by the National Science Foundation (NSF) under award III-2107150.

References

[1] A. Kumar, S. Alsudais, S. Ni, Z. Wang, Y. Huang, and C. Li, “Reshape: Adaptive result-aware skew handling for exploratory analysis on big data,” *CoRR*, vol. abs/2208.13143, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2208.13143

[2] X. Liu, Z. Wang, S. Ni, S. Alsudais, Y. Huang, A. Kumar, and C. Li, “Demonstration of collaborative and interactive workflow-based data analytics in texera,” *Proc. VLDB Endow.*, vol. 15, no. 12, pp. 3738–3741, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p3738-liu.pdf

[3] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. D. Team, “Jupyter notebooks - a publishing format for reproducible computational workflows,” in *Positioning and Power in Academic Publishing: Players, Agents and Agendas, 20th International Conference on Electronic Publishing, Göttingen, Germany, June 7-9, 2016*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87–90. [Online]. Available: https://doi.org/10.3233/978-1-61499-649-1-87

[4] 2023, alteryx Website, https://www.alteryx.com.

[5] 2023, knime Website, https://www.knime.com.

[6] 2023, rapidMiner Website, https://rapidminer.com/.

[7] 2023, einblick Website, https://www.einblick.ai.

[8] 2023, collaborative Data Analytics Using Workflows, https://github.com/Texera/texera/.

[9] M. D. Ma, A. Taylor, W. Wang, and N. Peng, “DICE: data-efficient clinical event extraction with generative models,” in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, A. Rogers, J. L. Boyd-Graber, and N. Okazaki, Eds. Association for Computational Linguistics, 2023, pp. 15 898–15 917. [Online]. Available: https://doi.org/10.18653/v1/2023.acl-long.886

[10] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[11] X. Chen, Y. Zhang, J. Deng, J. Jiang, and W. Wang, “Gotta: Generative few-shot question answering by prompt-based cloze data augmentation,” in *Proceedings of the 2023 SIAM International Conference on Data Mining, SDM 2023, Minneapolis-St. Paul Twin Cities, MN, USA, April 27-29, 2023*, S. Shekhar, Z. Zhou, Y. Chiang, and G. Stiglic, Eds. SIAM, 2023, pp. 909–917. [Online]. Available: https://doi.org/10.1137/1.9781611977653.ch102

[12] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, “BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, Eds. Association for Computational Linguistics, 2020, pp. 7871–7880. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.703

[13] X. Huang, J. Zhang, D. Li, and P. Li, “Knowledge graph embedding based question answering,” in *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining, WSDM 2019, Melbourne, VIC, Australia, February 11-15, 2019*, J. S. Culpepper, A. Moffat, P. N. Bennett, and K. Lerman, Eds. ACM, 2019, pp. 105–113. [Online]. Available: https://doi.org/10.1145/3289600.3290956

[14] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, “Ray: A distributed framework for emerging AI applications,” in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018, pp. 561–577. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/nishihara

[15] Cloud Computing, Hosting Services, and APIs — Google Cloud https://cloud.google.com.

[16] 2023, configuring Ray - Ray 2.3.0 https://docs.ray.io/en/latest/ray-core/configure.html.

[17] 2023, cPU threading and TorchScript inference - PyTorch 1.13 documentation, https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html.

[18] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer, “Proactive wrangling: mixed-initiative end-user programming of data transformation scripts,” in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, Santa Barbara, CA, USA, October 16-19, 2011*, J. S. Pierce, M. Agrawala, and S. R. Klemmer, Eds. ACM, 2011, pp. 65–74. [Online]. Available: https://doi.org/10.1145/2047196.2047205

[19] Y. Hou and D. Wang, “Hacking with npos: Collaborative analytics and broker roles in civic data hackathons,” *Proc. ACM Hum. Comput. Interact.*, vol. 1, no. CSCW, pp. 53:1–53:16, 2017. [Online]. Available: https://doi.org/10.1145/3134688

[20] D. Wang, J. D. Weisz, M. J. Muller, P. Ram, W. Geyer, C. Dugan, Y. R. Tausczik, H. Samulowitz, and A. G. Gray, “Human-ai collaboration in data science: Exploring data scientists' perceptions of automated AI,” *Proc. ACM Hum. Comput. Interact.*, vol. 3, no. CSCW, pp. 211:1–211:24, 2019. [Online]. Available: https://doi.org/10.1145/3359313

[21] Apache Spark, http://spark.apache.org.

[22] Apache Flink, http://flink.apache.org.

[23] I. Mavridis and H. D. Karatza, “Performance evaluation of cloud-based log file analysis with apache hadoop and apache spark,” *J. Syst. Softw.*, vol. 125, pp. 133–151, 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2016.11.037

[24] X. Zhang, U. Khanal, X. Zhao, and S. P. Ficklin, “Understanding software platforms for in-memory scientific data analysis: A case study of the spark system,” in *22nd IEEE International Conference on Parallel and Distributed Systems, ICPADS 2016, Wuhan, China, December 13-16, 2016*. IEEE Computer Society, 2016, pp. 1135–1144. [Online]. Available: https://doi.org/10.1109/ICPADS.2016.0149

[25] S. Dwivedi, P. Kasliwal, and S. Soni, “Comprehensive study of data analytics tools (rapidminer, weka, r tool, knime),” in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. IEEE, 2016, pp. 1–8.

[26] A. Minanovic, H. Gabelica, and Z. Krstic, “Big data and sentiment analysis using KNIME: online reviews vs. social media,” in *37th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2014, Opatija, Croatia, May 26-30, 2014*. IEEE, 2014, pp. 1464–1468. [Online]. Available: https://doi.org/10.1109/MIPRO.2014.6859797

[27] C. Sieb, T. Meinl, and M. R. Berthold, “Parallel and distributed data pipelining with knime,” *Mediterranean Journal of Computers and Networks*, vol. 3, no. 2, pp. 43–51, 2007.

[28] A. Arimond, C. Kofler, and F. Shafait, “Distributed pattern recognition in rapidminer,” in *Proceedings of rapidminer community meeting and conference*. Citeseer, 2010.

[29] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, “Understanding and improving the quality and reproducibility of jupyter notebooks,” *Empir. Softw. Eng.*, vol. 26, no. 4, p. 65, 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-09961-9

[30] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, “The story in the notebook: Exploratory data science using a literate programming tool,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, R. L. Mandryk, M. Hancock, M. Perry, and A. L. Cox, Eds. ACM, 2018, p. 174. [Online]. Available: https://doi.org/10.1145/3173574.3173748

[31] J. Wang, T. Kuo, L. Li, and A. Zeller, “Assessing and restoring reproducibility of jupyter notebooks,” in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 138–149. [Online]. Available: https://doi.org/10.1145/3324884.3416585

[32] A. Rule, A. Tabard, and J. D. Hollan, “Exploration and explanation in computational notebooks,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI 2018, Montreal, QC, Canada, April 21-26, 2018*, R. L. Mandryk, M. Hancock, M. Perry, and A. L. Cox, Eds. ACM, 2018, p. 32. [Online]. Available: https://doi.org/10.1145/3173574.3173606