The Art and Practice of Data Science Pipelines

A Comprehensive Study of Data Science Pipelines In Theory, In-The-Small, and In-The-Large

Sumon Biswas
Iowa State University
Ames, IA, USA
sumon@iastate.edu

Mohammad Wardat
Iowa State University
Ames, IA, USA
wardat@iastate.edu

Hridesh Rajan
Iowa State University
Ames, IA, USA
hridesh@iastate.edu

ABSTRACT

Increasingly larger number of software systems today are including data science components for descriptive, predictive, and prescriptive analytics. The collection of data science stages from acquisition, to cleaning/curation, to modeling, and so on are referred to as *data science pipelines*. To facilitate research and practice on data science pipelines, it is essential to understand their nature. What are the typical stages of a data science pipeline? How are they connected? Do the pipelines differ in the theoretical representations and that in the practice? Today we do not fully understand these architectural characteristics of data science pipelines. In this work, we present a three-pronged comprehensive study to answer this for the state-of-the-art, data science in-the-small, and data science in-the-large. Our study analyzes three datasets: a collection of 71 proposals for data science pipelines and related concepts in *theory*, a collection of over 105 implementations of curated data science pipelines from Kaggle competitions to understand data science *in-the-small*, and a collection of 21 mature data science projects from GitHub to understand data science *in-the-large*. Our study has led to three representations of data science pipelines that capture the essence of our subjects in theory, in-the-small, and in-the-large.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; • **Computing methodologies → Machine learning**.

KEYWORDS

data science pipelines, data science processes, descriptive, predictive

ACM Reference Format:
Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 2022. The Art and Practice of Data Science Pipelines: A Comprehensive Study of Data Science Pipelines In Theory, In-The-Small, and In-The-Large. In *44th ICSE, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages.

1 INTRODUCTION

Data science processes, also called *data science stages* as in stages of a pipeline, for descriptive, predictive, and prescriptive analytics are becoming integral components of many software systems today. The data science stages are organized into a *data science pipeline*, where data might flow from one stage in the pipeline to the next. These data science stages generally perform different tasks such as data acquisition, data preparation, storage, feature engineering, modeling, training, evaluation of the machine learning model, etc. In order to design and build software systems with data science stages effectively, we must understand the structure of the data science pipelines. Previous work has shown that understanding the structure and patterns used in existing systems and literature can help build better systems [20, 71]. In this work, we have taken the first step to understand the structure and patterns of DS pipelines.

Fortunately, we have a number of instances in both the state-of-the-art and practice to draw observations. In the literature, there have been a number of proposals to organize data science pipelines. We call such proposals *DS Pipelines in theory*. Another source of information is Kaggle, a widely known platform for data scientists to host and participate in DS competitions, share datasets, machine learning models, and code. Kaggle contains a large number of data science pipelines, but these pipelines are typically developed by a single data scientist as small standalone programs. We call such instances *DS Pipelines in-the-small*. The third source of DS pipelines are mature data science projects on GitHub developed by teams, suitable for reuse. We call such instances *DS Pipelines in-the-large*.

This work presents a study of DS pipelines in theory, in-the-small, and in-the-large. We studied 71 different proposals for DS pipelines and related concepts from the literature. We also studied 105 instances of DS pipelines from Kaggle. Finally, we studied 21 matured open-source data science projects from GitHub. For both Kaggle and GitHub, we selected projects that make use of Python to ease comparative analysis. In each setting, we answer the following overarching questions.

(1) **Representative pipeline:** What are the stages in DS pipeline and how frequently they appear?
(2) **Organization:** How are the pipeline stages organized?
(3) **Characteristics:** What are the characteristics of the pipelines in a setting and how does that compare with the others?

This work attempts to inform the terminology and practice for designing DS pipeline. We found that DS pipelines differ significantly in terms of detailed structures and patterns among theory, in-the-small, and in-the-large. Specifically, a number of stages are absent in-the-small, and the pipelines have a more linear structure with an emphasis on data exploration. Out of the eleven stages seen in theory, only six stages are present in pipeline in-the-small, namely *data collection*, *data preparation*, *modeling*, *training*, *evaluation*, and *prediction*. In addition, pipelines in-the-small do not have clear separation between stages which makes the maintenance harder. On the other hand, the DS pipelines in-the-large have a more complex structure with feedback loops and sub-pipelines.

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-XXXX-X/18/06.

We identified different pipeline patterns followed in specific phase (development/post-development) of the large DS projects. The abstraction of stages are stricter in-the-large having both loosely- and tightly-coupled structure.

Our investigation also suggest that DS pipeline is a well used software architecture but often built in ad hoc manner. We demonstrated the importance of standardization and analysis framework for DS pipeline following the traditional software engineering research on software architecture and design patterns [44, 54, 71]. We contributed three representations of DS pipelines that capture the essence of our subjects in theory, in-the-small, and in-the-large that would facilitate building new DS systems. We anticipate our results to inform design decisions made by the pipeline architects, practitioners, and software engineering teams. Our results will also help the DS researchers and developers to identify whether the pipeline is missing any important stage or feedback loops (e.g., *storage* and *evaluation* are missed in many pipelines).

The rest of this paper is organized as follows: in section §2, we present our study of DS pipelines in theory. Section §3 describes our study of DS pipelines in-the-small. In section §4, we describe our study of DS pipelines in-the-large. Section §5 discusses the implications, section §6 describes the threats to the validity, section §7 describes related work, and section §8 concludes.

2 DS PIPELINE IN THEORY

Data Science. Data Science (DS) is a broad area that brings together computational understanding, inferential thinking, and the knowledge of the application area. Wing [82] argues that DS studies how to extract value out of data. However, the value of data and extraction process depends on the application and context. DS includes a broad set of traditional disciplines such as data management, data infrastructure building, data-intensive algorithm development, AI (machine learning and deep learning), etc., that covers both the fundamental and practical perspectives from computer science, mathematics, statistics, and domain-specific knowledge [9, 74]. DS also incorporates the business, organization, policy and privacy issues of data and data-related processes. Any DS project involves three main stages: data collection and preparation, analysis and modeling, and finally deployment [81]. DS is also more than statistics or data mining since it incorporates understanding of data and its pattern, developing important questions and answering them, and communicating results [74].

Data Science Pipeline. The term *pipeline* was introduced by Garlan with *box-and-line* diagrams and explanatory prose that assist software developers to design and describe complex systems so that the software becomes intelligible [23]. Shaw and Garlan have provided the *pipes-and-filter* design pattern that involves stages with processing units (filters) and ordered connections (pipes) [71]. They also argued that pipeline gives proper semantics and vocabulary which helps to describe the concerns, constraints, relationship between the sub-systems, and overall computational paradigm [23, 71]. By *data science pipeline* (DS pipeline), we are referring to a series of processing *stage*s that interact with data, usually acquisition, management, analysis, and reasoning [50, 51]. The sequential DS stages from acquisition, to cleaning/curation, to modeling, and so on are referred to as *data science pipeline*. A DS pipeline may consist of several stages and connections between them. The stages are defined to perform particular tasks and connected to other stage(s) with input-output relations [4]. However, the definitions of the stages are not consistent across studies in the literature. The terminology vary depending on the application context and focus.

Different study in the literature presented DS pipeline based on their context and desiderata. No study has been conducted to unify the notions DS pipeline and collect the concepts [69]. While designing a new DS pipeline [83], dividing roles in DS teams [42], defining software process in data-intensive setting [79], identifying best practices in AI and modularizing DS components [4], it is important to understand the current state of the DS pipeline, its variations and different stages. To understand the DS pipelines and compare them, we collected the available pipelines from the literature and conducted an empirical study to unify the stages with their subtasks. Then we created a representative DS pipeline with the definitions of the stages. Next, we present the methodology and results of our analysis of DS pipelines in theory.

2.1 Methodology

2.1.1 Collecting Data Science Pipelines. We searched for the studies published in the literature and popular press that describes DS pipelines. We considered the studies that described both end-to-end DS pipeline or a partial DS pipeline specific to a context. First, we searched for peer-reviewed papers published in the last decade i.e., from 2010 to 2020. We searched the terms "*data science pipeline*", "*machine learning pipeline*", "*big data lifecycle*", "*deep learning workflow*", and the permutation of these keywords in IEEE Xplore, ACM Digital Library and Google Scholar. From a large pool, we selected 1,566 papers that fall broadly in the area of computer science, software engineering and data science. Then we analyzed each article in this pool to select the ones that propose or describe a DS pipeline. We found many papers in this collection use the terms (e.g., ML lifecycle), but do not contain a DS pipeline. We selected the ones that contain DS pipeline and extracted the pipelines (screenshot/description) as evidence from the article. The extracted raw pipelines are available in the artifact accompanied by this paper [5]. Thus, we found 46 DS pipelines that were published in the last decade.

Besides peer-reviewed papers, by searching the keywords on web, we collected the DS pipelines from US patent, industry blogs (e.g., Microsoft, GoogleCloud, IBM blogs), and popular press published between 2010 and 2020. After manual inspection, we found 25 DS pipelines from this grey literature. Thus, we collected 71 subjects (46 from peer reviewed articles and 25 from grey literature) that contain DS pipeline. We used an open-coding method to analyze these DS pipelines in theory [5] .

2.1.2 Labeling Data Science Pipelines. In the collected references, DS pipeline is defined with a set of stages (*data acquisition*, *data preparation*, *modeling*, etc.) and connections among them. Each stage in the pipeline is defined for performing a specific task and connected to other stages. However, not all the studies depict DS pipelines with the same set of stages and connections. The studies use different terminologies for defining the stages depending on the context. To be able to compare the pipelines, we had to understand the definitions and transform them into a canonical form. For a

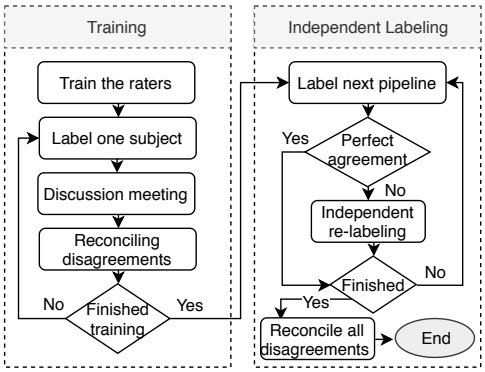

Figure 1: Labeling method for DS pipelines in theory

given DS pipeline, identifying their stages and mapping them to a canonical form is often challenging. The sub-tasks, overall goal of the project, utilities affect the understanding of the pipeline stages. To counter these challenges, we used an open-coding method to label the stages of the pipelines.

Two authors labeled the collected DS pipelines into different criteria. Each author read the article, understood the pipeline, identified the stages, and labeled them. In each iteration, the raters labeled 10% of the subjects (7-8 pipelines). The first 8 subjects were used for training and forming the initial labels. After each iteration, we calculated the Cohen's Kappa coefficient [77], identified the mismatches, and resolved them in the presence of a moderator, who is another author. Thus, we found the representative DS pipeline after rigorous discussions among the raters and the moderator. The methodology of this open-coding procedure is shown in Figure 1. The entire labeling process was divided into two phases: 1) training, and 2) independent labeling.

Training: The two raters were trained on the goal of this project and their roles. We randomly selected eight subjects for training. First, the raters and the moderator had discussions on three subjects and identified the stages in their DS pipeline. Thus, we formed the commonly occurred stages and their definitions, which were updated through the entire labeling and reconciliation process later. After the initial discussion and training, the raters were given the already created definitions of the stages and one pipeline from the remaining five for training. The raters labeled this pipeline independently. After labeling the pipeline, we calculated the agreement and conducted a discussion session among the raters and the moderator. In this session, we reconciled the disagreements and updated the labels with the definitions. We continued the training session until we got perfect agreement independently. The inter-rater agreement was calculated using Cohen's Kappa coefficient [77]. A higher κ ([0, 1]) indicates a better agreement. The interpretation of of κ is shown in Figure 2a. In the discussion meetings, the raters discussed each label (both agreed and disagreed ones) with the other rater and moderator, argued for the disagreed ones and reconciled them. In this way, we came up with most of the stages and a representative terminology for each stage including the sub-tasks.

Independent labeling: After completing the training session, the rest of the subjects were labeled independently by the raters. The raters labeled the remaining 63 labels: 7 subjects (10%) in each of the 9 iterations. The distribution of κ after each independent

Range (κ)	Agreement level
0.00 - 0.20	Slight agreement
0.21 - 0.40	Fair agreement
0.41 - 0.60	Moderate agreement
0.61 - 0.80	Substantial agreement
0.81 - 1.00	Perfect agreement

(a) Interpretation of Kappa (κ)

Iteration #	κ	Iteration #	κ
1	0.67	6	0.91
2	0.74	7	0.87
3	0.82	8	0.90
4	0.84	9	0.94
5	0.84	10	0.91

(b) Agreement in different stages

Figure 2: Labeling agreement calculation

labeling iteration is shown in Figure 2b. In each iteration, first, the raters had the labeling session, and then the raters and moderator had the reconciliation session.

Labeling. The raters labeled separately so that their labels were private, and they did not discuss while labeling. The raters identified the stages and connections between them, and finally labeled whether the DS pipeline involves processes related to cyber, physical or human component in it. In independent labeling, we found almost perfect agreement (κ = 0.83) on average. Even after high agreement, there were very few disagreements in the labels, which were reconciled after each iteration.

Reconciling. Reconciliation happened for each label for the subject studies in the training session, and the disagreed labels for the studies in independent labeling session. In training session, the reconciliation was done in discussion meetings among the raters and the moderator, whereas for the independent labels, reconciliation was done by the moderator after separate meetings with the two raters. For reconciliation, the raters described their arguments for the mislabeled stages. For a few cases, we had straightforward solution to go for one label. For others, both the raters had good arguments for their labels, and we had to decide on that label by updating the stages in the definition of the pipeline. All the labeled pipelines from the subjects are shared in our paper artifact [5].

Furthermore, after finishing labeling the pipelines stages, we also classified the subject references into four classes based on the overall purpose of the article. First, after a few discussions, the raters and moderator came up with the classes. Then, the raters classified each pipeline into one class. We found disagreements in 6 out of 71 references, which the moderator reconciled with separate meetings with the two raters. Based on our labeling, the literature that we collected are divided into four classes: describe or propose DS pipeline, survey or review, DS optimization, and introduce new method or application. Next, we are going to discuss the result of analyzing the DS pipelines in theory.

2.2 Representative Pipeline in Theory

The labeled pipelines[1] with their stages are visually illustrated in the artifact [5]. We found that pipelines in theory can be both software architecture and *team processes* unlike pipelines in-the-small and in-the-large. Through the labeling process, we separated those team processes (25 out of 71), which are discussed in §2.4.

RQ1a: What is a representative definition of the DS pipeline in theory? From the empirical study, we created a representative rendition of DS pipeline with 3 layers, 11 stages and possible connections between stages as shown in Figure 3. Each shaded box represents a DS stage that performs certain sub-tasks (listed under the box). In the preprocessing layer, the stages are *data acquisition*, *preparation*, and *storage*. The preprocessing stage *study design* only

[1] https://github.com/anonymous-authorss/DS-Pipeline/blob/main/pipelines.pdf

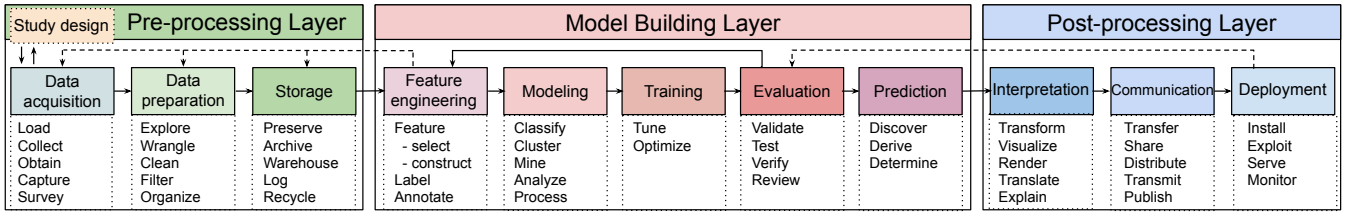

Figure 3: Concepts in a data science pipeline. The sub-tasks are listed below each stage. The stages are connected with feedback loops denoted with arrows. Solid arrows are always present in the lifecycle, while the dashed arrows are optional. Distant feedback loops (e.g., from *deployment* to data *acquisition*) are also possible through intermediate stage(s).

Stages of Data Science Pipeline
Data Acquisition (ACQ): In the beginning of DS pipeline, data are collected from appropriate sources. Data can be acquired manually or automatically. Data acquisition also involves understanding the nature of the data, collecting relevant data, and integrating available datasets.
Data Preparation (PRP): Data are generally acquired in a raw format that needs certain preprocessing steps. This involves exploration and filtering, which helps identify the correct data for further processing. Well prepared data reduces the time required for data analysis and contributes to the success of the DS pipeline.
Storage (STR): It is important to find an appropriate hardware-software combination to preserve data so that it can be processed efficiently. For example, Miao et al. used graph database system Neo4j [48] to build a collaborative analysis pipeline [45], since Neo4J supports querying graph data properties.
Feature Engineering (FTR): The entire dataset might not contribute equally to decision making. In this stage, appropriate features that are useful to build the model are identified or constructed. Features that are not readily available in the dataset, require engineering to create them from raw data.
Modeling (MDL): When data are preprocessed and features are at hand, a model is built to analyze the data. Model building includes model planning, model selection, mining and deriving important properties of data. Appropriate data processing strategies and algorithms are selected to create a good model.
Training (TRN): For a specific model, we need to train the model with available labeled data. By each training iteration, we optimize the model and try to make it better. The quality of the training dataset contributes to the training accuracy of the model.
Evaluation (EVL): After training the model, it is tested with a new dataset which has not been used as training data. Also, the model can be evaluated in real-life scenarios and compared with other competing models. Existing metrics are used or new metrics are created to evaluate the model.
Prediction (PRD): The success of the model depends on how good a model can predict in an unknown setup. After a satisfactory evaluation, we employ the model to solve the problem and see how it works. There are many prediction metrics such as classification accuracy, log-loss, F1-score, to measure the success of the model.
Interpretation (INT): The prediction result might not be enough to make a decision. We often need a transformation of the prediction result and post-processing to translate predictions into knowledge. For example, only numerical results do not help much but a good visualization can help to make a decision.
Communication (CMN): Different components of the DS system might reside in a distributed environment. So, we might need to communicate with the involved parties (e.g., devices, persons, systems) to share and accumulate information. Communication might take place in different geographical locations or the same.
Deployment (DPL): The built DS solution is installed in its problem domain to serve the application. Over time, the performance of the model is monitored so that the model can be improved to handle new situations. Deployment also includes model maintenance and sending feedback to the model building layer.

Table 1: Description of the stages in DS pipeline

appeared in team process pipelines that comprise requirement formulation, specification, and planning, which are often challenging in data science. The algorithmic steps and data processing are done in the model building layer. *Modeling* does not necessarily imply the existence of an ML component, since DS can involve custom data processing or statistical modeling. Post-processing layer includes the tasks that take place after the results have been generated. The DS pipeline stages are described in Figure 1.

RQ1b: What are the frequent and rare stages of the DS pipeline in theory? The frequency of stage can depend on the

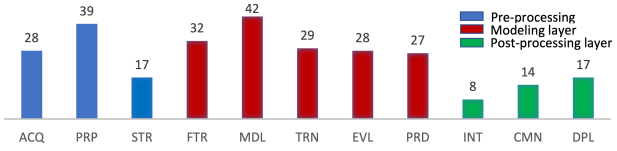

Figure 4: Frequency of pipeline stages in theory

focus of the pipeline or its importance in certain context (ML, big-data management). Among 46 DS pipelines (which are not team processes), Figure 4 shows the number of times each stage appears. A few pipelines present stages with broad terminology that fit multiple stage-definitions. In those cases, the pipelines were labeled with the fitted stages and counted multiple times. *Modeling, data preparation, and feature engineering* appear most frequently in the literature. While *modeling* is present in 93% of the pipelines, other model related stages (*feature engineering, training, evaluation, prediction*) are not used consistently. Often *training* is not considered as a separate stage and included inside the *modeling* stage. Similarly, we found that *evaluation* and *prediction* are often not depicted as separate stages. However, by separating the stages and modularizing the tasks, the DS process can be maintained better [4, 69]. The pipeline created with the most number of stages (11) is provided by Ashmore et al. [7]. On the other hand, about 15% of the pipelines from the literature are created with a minimal number (3) of stages. Among them, 80% are ML processes and falls in the category of DS optimizations. We found that these pipelines are very specific to particular applications, which include context-specific stages like data sampling, querying, visualization, etc., but do not cover most of the representative stages. A pipeline in theory may not require all representative stages, since it can have novelty in certain stages and exclude the others. However, the representative pipeline provides common terminology and facilitate comparative analysis.

Finding 1: *Post-processing layers are included infrequently (52%) compared to pre-processing (96%) and model building (96%) layers of pipelines in theory.*

Clearly, preprocessing and model building layers are considered in almost all of the studies. In most of the cases, the pipelines do not consider the post-processing activities (*interpretation, communication, deployment*). These pipelines often end with the predictive process and thus do not follow up with the later stages which entails how the result is interpreted, communicated and deployed to the external environment. Miao et al. argued that overall lifecycle management tasks (e.g., model versioning, sharing) are largely ignored for deep learning systems [46]. Previous studies

also showed that significant amount of cost and effort is spent in the post-development phases in traditional software lifecycle [44, 59]. In data-intensive software, the maintenance cost can go even higher with the high-interest technical debt in the pipeline [68]. Therefore, post-processing stages should be incorporated for a better understanding of the maintenance of the DS pipeline.

2.3 Organization of Pipeline Stages in Theory

RQ2: How are pipeline stages connected to each other? In Figure 3, for simplicity, we depicted the DS pipeline as a mostly linear chain. However, our subject DS pipelines often have non-linear behavior. In any stage, the system might have to return to the previous stage for refinement and upgrade, e.g., if a system faces a real-world challenge in *modeling*, it has to update the algorithm which might affect the data pre-processing and feature engineering as well. Furthermore, the stages do not have strict boundaries in the DS lifecycle. In Figure 3, two backward arrows, from *feature engineering* and *evaluation*, indicate feedback to any of the previous stages. Although in traditional software engineering processes (e.g., waterfall model, agile development, etc.), feedback loop is not uncommon, in DS lifecycle, there are multiple stakeholders and models in a distributed environment which makes the feedback loops more frequent and complex. Sculley et al. pointed that DS tasks such as sampling, learning, hyperparameter choice, etc. are entangled together so that Changing Anything Changes Everything (CACE principle) [69], which in turn creates implicit feedback loops that are not depicted in the pipelines [13, 21, 60, 76]. The feedback loops inside any specific layer are more frequent than the feedback loops from one layer to another. Also, a feedback loop to a distant previous stage is expensive. For example, if we do *data preparation* after *evaluation* then the intermediate stages also require updates.

2.4 Characteristics of the Pipelines in Theory

RQ3: What are the different types of pipelines available in theory? The context and requirements of the project can influence pipeline design and architecture [22]. Here, we present the types of pipelines with different characteristics that are available in theory. We classified each subject in our study into four classes based on the overall goal of the article. The most of the pipelines in theory (39%) are *describing or proposing* new pipelines to solve a new or existing problem. About 31% of the pipelines are on *reviewing or comparing* the existing pipelines. The third group of DS pipelines (14%) are intended to *optimize* a certain part of the pipeline. For example, Van Der Weide et al. proposes a pipeline for managing multiple versions of pipelines and optimize performance [76]. Most of the pipelines in this category are application specific and include very few stages that are necessary for the optimization. Fourth, some research *introduce new application* or method and present within the pipeline. We observed that there is no standard methodology to develop comparable and inter-operable DS pipelines. Using the labeling methodology shown in Figure 1, we labeled each pipeline and found three types of DS pipelines in the literature: 1) ML process, 2) big data management process, and 3) team process.

ML process: 46% of all the pipelines we found in the literature are describing machine learning processes. The recent advent of artificial intelligence, supervised learning and deep learning has led to more DS systems that involve ML components. The pipelines in this category emphasize the algorithmic process, learning patterns, and building predictive models. However, the post-processing stages are rare in these type of pipelines. The ML pipelines are often thought of as algorithmic process in the laboratory scenario. But as mentioned in [7], incorporating the post-processing stages would be desired to ensure safe real-world deployment of such pipelines.

Big data management: The references in this category present DS pipelines that manage a large amount of data or describes a framework (software-hardware infrastructure) for data processing but do not contain machine learning components in the pipeline. Processing large amount data often requires specific algorithms and engineering methods for efficiency and further processing. We found that 18% of all the subject studies fall in this category.

Team process: We also found some DS pipelines that are not describing DS software architecture. These pipelines describe workflow of human activities that needs to be followed in a DS pipeline. These studies present a high-level view for building DS component in a team environment. The data science teams require specific expertise and management to build successful DS pipelines [4, 42]. In this paper, in §3 and §4, we are only focusing on DS pipeline as software architecture, and therefore, we did not compare the team process pipelines in the rest of this section.

Finding 2: *Most of the pipelines in-theory involve cyber and physical components, only a few with human processes in the loop.*

We identified whether the pipelines involve cyber, physical or human process, using our labeling process described in section §2.1.2. **Cyber** processes refer to activities that involve automated systems and machinery computations. Since modern DS systems involves large amount of data and requires extensive computation, all of the pipelines include cyber component in it. **Physical** processes include the activities which require real-world connections with the system. For example, collecting data using mobile sensors or cameras is a physical process. Although 23% of the big data pipelines include physical processes, only 9% of the ML pipelines include that in the pipeline. In many DS systems, developers or researchers participate in the pipelines actively to make decisions that need **human** interventions [74, 76]. For example, in many DS systems, analytical model validation, troubleshooting, data interpretation is necessary which requires human involvement. However, only 13% of the pipelines acknowledged human involvement in the pipeline.

3 DS PIPELINE IN-THE-SMALL

Similar to the DS pipelines in large systems and frameworks, for a very specific data science task (e.g., object recognition, weather forecasting, etc.), programmers build pipeline. Different stages of the program perform a specific sub-task and connect with the other stages using data-flow or control-flow relations. In this section, we described such DS pipelines *in-the-small*.

3.1 Methodology

We collected 105 DS programs from Kaggle competition notebooks [31]. Kaggle is one of the most popular crowd-sourced platforms for DS competitions, owned by Google. Besides participating in competitions, data scientists, researchers, developers collaborate to

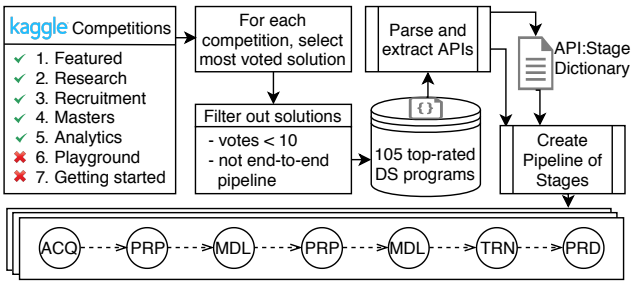

Figure 5: The pipeline creation process for Kaggle programs

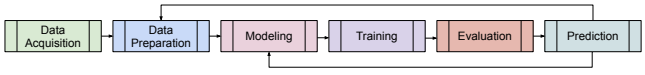

Figure 6: Pipeline *in-the-small* extracted from API usages

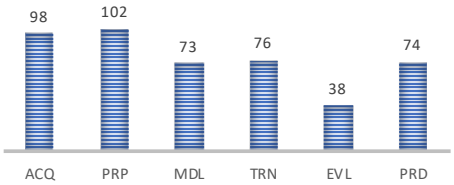

Figure 7: Frequency of pipeline stages *in-the-small*

learn and share DS knowledge in variety of domains. The users and organizations can host a DS competition in Kaggle to solve real-world problems. A competition is accompanied by a dataset and prize money. Many Kaggle solutions have resulted in impactful DS algorithms and research such as neural networks used by Hinton and Dahl [15], improving the search for the Higgs Boson at CERN [30], etc. We chose Kaggle solutions to analyze DS pipeline for three reasons: 1) all programs perform a DS task and provide solution to a well specified problem associated with a dataset, 2) solutions with the highest number of votes are well accepted solutions for a specific problem, and 3) the problems cover a wide range of domains.

There are 331 completed competitions in Kaggle to date. They categorized the competitions into *Featured*, *Research*, *Recruitment*, *Masters*, *Analytics*, *Playground* and *Getting started*. We collected solutions of all the competitions from each category except *Getting started* and *Playground* (these two categories are intended to serve as DS tutorials and toy projects). First, we filtered the competitions for which there are solutions available (many old competitions do not contain any public solution). We found 138 such competitions. For a given competition problem, we selected the most voted solution which has at least 10 votes. Thus, we got 105 top-rated DS solutions for analyzing pipelines in-the-small. This selection and pipeline creation process is shown in Figure 5.

All of the DS programs are written in Python using ML libraries like Keras, Scikit-learn, Tensorflow, etc. These packages provide high-level Application Programming Interfaces (APIs) for performing a specific task on data or model. We parsed the programs into Abstract Syntax Tree (AST) and collected all the API calls from the programs. Then the functionality of an API is used to identify the stage of the pipeline. We extracted the temporal order of API calls to identify the stages. Standard static analysis of the Python programs facilitate the extraction process. Our analysis suggests that the DS programs follow a linear structure with less than 4% AST nodes being conditional or loops. Wang et al. proposed a similar approach for extracting external dependencies in Jupyter Notebooks by creating an API database and analyzing AST [80].

We created a dictionary by mapping each API collected from the programs, to one of the 11 stages of the DS pipeline described in section §2. During the mapping, we excluded the generic APIs from the dictionary. For example, `model.summary()` is used to print the model parameters and does not represent any stage of the pipeline. For creating the dictionary, we taken a two-fold approach. First, we understand the context of the program and API usage. Second, we look at the API documentation to confirm the corresponding pipeline stage. We found that DS APIs are definitive in their operations and well-categorized by the library. For example, the APIs in Keras [38] and Scikit-learn [39] are grouped into preprocessing, models, etc. Our API-dictionary was manually validated by a second-rater and moderator who labeled DS pipelines in section §2. Then, we built a tool which takes the API dictionary and DS program, and automatically creates the DS pipeline. For a sequence of APIs with the same stage, we abstracted them into a single stage. As an example, Figure 5 shows a DS pipeline created from a Kaggle solution [32]. Each stage in the pipeline (e.g., ACQ, PRP) represents one or more API usages. The arrows in the pipeline denote the temporal sequence of stages. Note that, one stage can appear multiple times in a pipeline. The API dictionary, Kaggle programs, and tool to generate the pipelines is shared in the paper artifact [5].

3.2 Representative Pipeline in-the-Small

RQ4: What are the stages of *DS pipeline in-the-small*? Among the 11 pipeline stages described in Figure 3, we found only 6 stages in the DS programs that are depicted in Figure 6. Other stages (e.g., *storage, feature engineering, interpretation, communication, deployment*) are not found in these programs because these stages occur while building a production-scale large DS system and often not present in the DS notebooks. Therefore, the pipeline in DS programs consists of the subset of pipeline stages in theory.

We summarized the frequency of each stage of the DS programs in Figure 7. Among 105 programs, *data acquisition* and *data preparation* are present in almost all of them. Surprisingly, *modeling* is present in only 70% of the programs. We found that, in many programs, no modeling APIs had been used because developers did not use any built-in ML algorithm from libraries, e.g., LogisticRegression, LSTM, etc. In these cases, the developers use data-processing APIs on the training data to build custom model, e.g., this notebook [33] uses *data preparation* APIs to produce results. To enable more abstraction of the stages in these pipelines, further modularization is necessary, which has been investigated in RQ8.

Finding 3: *Evaluation stage is infrequent, appearing only in 36% of the pipelines in-the-small.*

Evaluation is a tricky stage of the DS pipeline. Developers have to choose the appropriate metric and methodology to evaluate their model. Based on the evaluation result, the model is updated over multiple iterations. We found that, besides using metrics, in many cases, evaluation requires human understanding and comparison

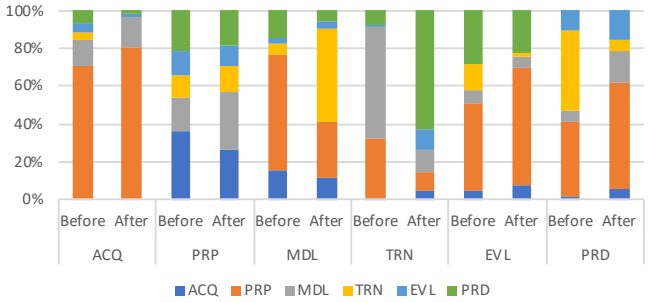

Figure 8: Stages occurring before and after each stage

of the result produced by the model. The reason for having less number of *evaluation* stage in the pipeline is that often the developers evaluate the performance by plotting and visualizing the result. Since the visualization APIs are not considered as *evaluation* stage, we found this stage less frequently in pipelines. Also, many programs directly go to the *prediction* stage without going to *evaluation* stage at all. Furthermore, notebooks are often used for experimentation purposes so that many computations are performed during development but eliminated when the notebooks are shared [40]. For example, one developer might try a number of classifiers and evaluate their accuracy. After finding the best performing classifier, it can be the only one shared in the notebook. Therefore, we experienced many missing stages in the pipeline in-the-small. The complex DS tasks require several computations which might not be used in producing the final prediction, but definitely should be considered as part of the pipeline.

3.3 Pipeline Organization in the Small

RQ5: How are the stages connected with each other in pipeline in-the-small? To answer RQ5, we considered each occurrence of the stages in a DS program and looked at its previous and next stage. In Figure 8, we showed which stages are followed or preceded by each stage. We found that *data preparation* can occur before or after all other stages. Apart from that, *data acquisition* is followed by *data preparation* most of the time, which in turn is followed by *modeling*. *Modeling* is followed mostly by *training*, which in turn is followed by *prediction*. *Evaluation* is mostly surrounded by *prediction* and *data preparation*. From Figure 8, we can also find some most occurring feedback loop: *evaluation* to *preparation*, *evaluation* to *modeling* and *prediction* to *modeling*.

Data preparation tasks (e.g., formatting, reshaping, sorting) are not limited to just before the *modeling* stage, rather it is done on a *whenever-needed* basis. For example, in the following code snippet from a Kaggle competition [34], while creating model-layers, data preprocessing API has been called in line 2.

```
1 x = Conv2D(mid, (4, 1), activation='relu', padding='valid')(x)
2 x = Reshape((branch_model.output_shape[1], mid, 1))(x)
3 x = Conv2D(1, (1, mid), activation='linear', padding='valid')(x)
4 x = Flatten(name='flatten')(x)
5 head_model = Model([xa_inp, xb_inp], x, name='head'
```

The *modeling* stage is always surrounded by other stages of the pipeline. However, there is often a loop around *modeling, training, evaluation, and prediction*. *Modeling* often repeats many times to improve the model over multiple iterations. For example, in the following Kaggle code snippet [35], the model is created and trained multiple times to find the best one.

```
1 random_forest = RandomForestClassifier(n_estimators=100,
      random_state=50, verbose=1, n_jobs=-1) # Modeling
2 random_forest.fit(train, train_labels) # Train
3 ...
4 poly_features = scaler.fit_transform(poly_features)
5 poly_features_test = scaler.transform(poly_features_test)
6 random_forest_poly = RandomForestClassifier(n_estimators=100,
      random_state=50, verbose=1, n_jobs=-1) # Modeling
7 random_forest_poly.fit(poly_features, train_labels) # Training
8 pred = random_forest_poly.predict_proba(poly_features_test)[:,1]
```

Finding 4: *Stages of pipelines in-the-small are often tangled with each other.*

All of the DS programs fail to maintain a good separation of concerns [17] between stages. Strong abstraction boundaries help to make the program modular and easy-to-maintain [54]. In addition, a good DS solution should not only compute a better predictive result, but also facilitate software engineering activities (e.g., debugging, testing, monitoring, etc.) [25]. However, we found that stages are often *tangled* with other stages [12, 41, 57] across the pipelines. The code for one stage is interspersed with the code for other stages. For example, while building the deep learning network (*modeling*), the developers often switch to different *data preparation* tasks, e.g., reshaping, resizing [29], which tangles data preparation concern with the modeling concern. We observed some early attempts to adopt modular design practices. For instance, this notebook [36] separated code into different high-level stages, namely, preparation, feature extraction, exploratory data analysis (EDA), topic model, etc. These high-level pipelines can improve the abstraction, which further enable the maintainability, and reusability [64]. In some scenerios, reuse or maintainance might not be desired for pipelines in-the-small. However, to enahnce readability [40] and repeatability [25] and ease of testing/debugging, more attention on modular design practices is needed for DS pipelines.

Finding 5: *Data preparation stage is occurring significant number of times between any two stages of pipelines in-the-small, which is causing pipeline jungles.*

We found that new data sources are added, new features are identified, and new values are calculated incrementally in the pipeline which evolves organically. This results in a large number of data preprocessing tasks like sampling, joining, resizing along with random file input-output. This is called *pipeline jungles* [69], which causes technical debt for DS systems in the long run. *Pipeline jungles* are hard to test and any small change in the pipeline will take a lot of effort to integrate. The situation gets worse in case of larger DS pipelines, where several data management activities (e.g., clean, serve, validate) are necessary through the pipeline in different stages [55, 56]. The recommended way is to think about the pipeline holistically and scrape the pipeline jungle by redesigning it, which in turn takes further engineering effort [69]. We found that the large DS projects, which are discussed in §4, isolate the data preparation tasks into separate files and modules [11, 66, 75, 85], which alleviates the pipeline jungles problem. So, DS pipeline in-the–small needs further IDE (e.g., Jupyter Notebook, etc.) support and methodologies for code isolation and modularization.

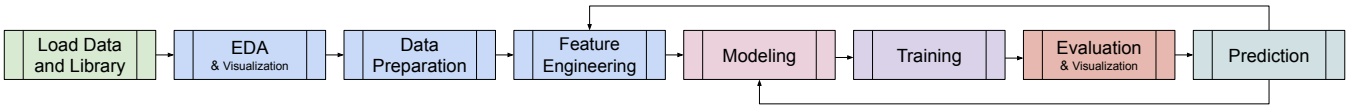

Figure 9: Representative data science pipeline *in-the-small*

3.4 Characteristics of Pipelines in-the-Small

RQ6: What are the patterns in pipeline in-the-small and how it compares to pipeline in theory? We have not found many stages from Figure 3, e.g., feature engineering, interpretation, communication, in pipeline in-the-small. One reason is that the low-level pipeline extracted from the API usages cannot capture some stages. For example, even if a developer is conducting feature engineering, the used APIs might be from the data preparation stage. Fortunately, we found many Kaggle notebooks that are organized by the pipeline stages. We visited all the 105 Kaggle notebooks in our collection and extracted these high-level pipelines manually. Unlike the low-level pipelines (extracted using API usages), a high-level pipeline consists of the stages abstracted by the developers and specified in the notebooks.

The Kaggle notebooks follow literate programming paradigm [64, 78], which allows the developers to describe the code using rich text and separate them into sections. We found that 34 out of 105 notebooks divided the code into stages. We collected those stages from the Kaggle notebooks. Furthermore, we labeled these notebooks into the 11 stages from DS pipeline in theory by two raters, and extracted the stages that are not present in theory. The extracted high-level pipelines and labels are available in the paper artifact [5].

We observed that no notebooks specify these stages: *storage, interpretation, communication,* and *deployment*. These DS programs are not production-scale projects. Therefore, they do not include the post-processing stages in the pipeline. The most common stages are *modeling* (79%), *data preparation* (62%), *data acquisition* (53%), and *feature engineering*(35%), which is aligned with the finding of DS pipeline in theory. In addition, we have these stages which are not present in DS pipeline in theory: *library loading, exploratory data analysis (EDA), visualization*. Among them *EDA* has been used most of the times, i.e., 43% of the pipelines. Furthermore, *EDA* covered the most part of those pipeline. Before going to the modeling and successive stages, a lot of effort is given on understanding the data, compute feature importance, and visualize the patterns, which help to build effective models quickly in the later stages [7].

Furthermore, some notebooks present *library loading* as separate stage. We observed that choosing appropriate library/framework and setting up the environment is an important step while developing pipeline in-the-small. We also found that data visualization is an recurring stage mentioned by the developers. Visualization can be done for EDA or feature engineering (before modeling), or for evaluation (after modeling). Based on these observations we updated the representative pipeline in-the-small in Figure 9. The high-level pipeline provides an overall representation of the system, which can be leveraged to design software process. It would be beneficial for the developers to close the gap between the low-level and the high-level pipeline by identifying the tangled stages.

Table 2: GitHub projects for analyzing pipeline in-the-large

Project Name	Purpose	#Files	#AST	LOC
Autopilot [3]	Pilot a car using computer vision	36	11185	348
CNN-Text-Classification [11]	Sentence classification	69	47797	11.4K
Darkflow [75]	Real-time object detection and classification	1025	655670	8.6K
Deep ANPR [19]	Automatic number plate recognition	64	70464	10.8K
Deep Text Corrector [52]	Correct input errors in short text	47	50770	3.0K
Face Classification [6]	Real-time face and emotion/gender detection	292	117901	35.3K
FaceNet [66]	Face recognition	1352	1889529	18.2K
KittiSeg [73]	Road segmentation	276	187143	4.8K
LSTM-Neural-Network [8]	Predict time series steps and sequences	24	11434	1.2K
Mask R-CNN [2]	Object detection and instance segmentation	256	1567786	15.6K
MobileNet SSD [58]	Object detection network	28	21272	25.6K
MTCNN [14]	Joint face detection and alignment	153	121138	219.7K
Object-Detector-App [16]	Real-time object recognition	215	318534	47.9K
Password-Analysis [65]	Analyze a large corpus of text passwords	148	67870	3.6K
Person Blocker [84]	Block people in images	12	44517	977
QANet [85]	Machine reading comprehension	83	107669	2K
Speech-to-Text-WaveNet	Sentence level english speech recognition	32	18626	5.1K
Tacotron [53]	Text-to-speech synthesis	114	58845	1.4K
Text-Detection-CTPN [63]	Text detection	640	257083	18.4K
TF-Recomm [72]	Recommendation systems	17	7789	535
XLNet [86]	Language understanding	36	143172	11.5K

4 DS PIPELINE IN-THE-LARGE

The DS solutions described in the previous section are specific to a given dataset and a well-defined problem. However, there are many DS projects which are large, not limited to a single source file, and contains multiple modules. These solutions are intended to solve more general problems which might not be specific to a dataset. For example, the objective of the *Face Classification* project in GitHub [6] is to detect face from images or videos and classify them based on gender and emotion. This problem is not specific to a particular dataset and the scope is broader compared to the Kaggle solutions. We have collected such top-rated DS projects from GitHub to analyze DS pipeline in-the-large.

4.1 Methodology

Biswas et al. published a dataset containing top rated DS projects from GitHub [10]. From the list of projects in this dataset, we have filtered mature DS projects having more than 1000 stars. Thus, we have found 269 mature GitHub projects. However, there are many projects in this list which are DS libraries, frameworks or utilities. Since we want to analyze the pipeline of data science software, we have removed those projects. Finally, we have also removed the repositories which serve educational purposes. Thus, we have found a list of 21 mature open-source DS projects. The list of projects, and their purpose are shown in Table 2.

For each project, we have created two pipelines: high-level pipeline and low-level pipeline. For creating the high-level pipeline, we have manually checked the project architecture, module structure and execution process. This gave us a good understanding of the source file organization and linkage between modules. After identifying the high-level pipeline and execution sequences of the source files, we have used the same API based method used to analyze Kaggle programs in the previous section, to create low-level pipeline of these GitHub projects. The methodology of selecting and extracting pipelines from the GitHub projects is shown in Figure 10.

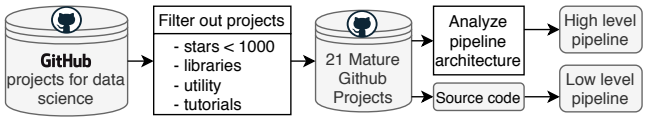

Figure 10: Pipeline creation process for GitHub projects

For example, the project *QANet* [85] is intended to do machine reading comprehension. Here, Python has been used as the primary language, and shell script has been used for data downloading and project setup. The high-level pipeline for *QANet* includes the stages: *data acquisition*, *data preparation*, *modeling*, *training*, *evaluation* and *prediction*. In the beginning, `config.py` file integrates the modules (preparation, modeling, and training) and provides an interface to configure a model by specifying dataset and other parameters. Then, the file `evaluate.py` is executed to perform the evaluation and prediction. For the low-level pipeline, for a specific file, we used the API based analysis to generate the pipeline, which was used to analyze pipeline in-the-small. For instance, in the project *QANet*, although `model.py` serves modeling at a high level, it also does data preparation, training, and evaluation, when APIs are considered. In addition to the pipeline stages, we have also identified a few other properties of each project: 1) number of contributors, 2) AST count, 2) technology/language used, 3) entry points and 4) execution sequence. We have leveraged the Boa infrastructure [18] to analyze the different properties of the projects. These properties helped us to categorize and analyze the pipeline in-the-large. The details of the projects are available in the paper artifact [5].

The projects are from various domains: object detection, face classification, automated driving, speech synthesis, number plate recognition, etc. and using Python as the primary language. The number of developers in each project ranges between 1 and 40 with an average of 8. Among 21 projects, 16 of them are developed by teams and 5 of them are developed by individuals.

4.2 Representative Pipeline in-the-Large

Compared to the Kaggle programs, we have found a significant difference in the pipeline of large DS projects. Because of the larger size of the projects, the pipeline architecture is different. All the projects contain multiple source files for handling different tasks (e.g., modeling, training) and about 50% of the projects organize the source files into modules (e.g., *utils*, *preprocessing*, *model*, etc.).

RQ7: What is the representative DS pipeline in-the-large? Each of the projects contains six stages described in Figure 6: *acquisition, preparation, modeling, training, evaluation*, and *prediction*. However, since the projects are not coupled to a specific dataset and they solve a more general problem, the projects are not limited to one single pipeline. We have found that the pipeline of each project is divided into two phases: 1) development phase and 2) post-development phase, which is depicted in Figure 11.

In **development phase**, the main goal is to build a model that solves the problem in general. A base dataset is used to build the model that would be used for other future datasets. After completing a *modeling, training, evaluation* loop, the final model is created and saved as an artifact. Afterwards, the projects also create model interfaces, which lets the user modify and exploit the model in the post-development phase. Finally, the model artifact is saved

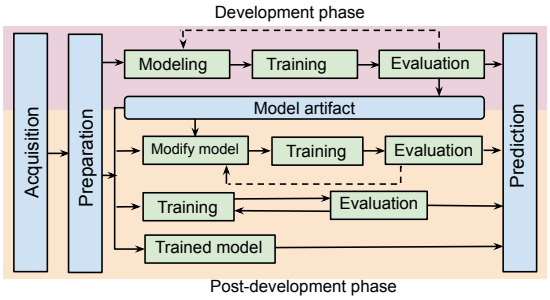

Figure 11: DS pipeline in-the-large. Development phase (top) runs during model building and post-development phase (bottom) runs for making prediction.

as a source file or some model archiving formats. For example, the project *Person-Blocker* [84] and *Speech-to-Text-WaveNet* [43] saved the model in the source file (`model.py`) and lets the users train the model in the next phase. On the other hand, the project *KittiSeg* [73] and *Autopilot* [3] saved the built model artifact in JSON format (`.json`) and checkpoint format (`.ckpt`) respectively. We have observed that the evaluation and prediction is often not the main goal in this phase; rather, building an appropriate model and making it available for further usage is the central activity.

In **post-development phase**, the users access the pre-built model and use that for prediction. After acquiring data, a few pre-processing steps are needed to feed the model. In all of the projects under this study, we have found that the development phase is similar. However, we have identified three different patterns in the post-development phase which are shown in Figure 11. First, the users can modify the model by setting its hyperparameters and use that to make prediction on a new dataset. Second, the users can use the model as-it-is and train the model on the new dataset to make prediction. Third, the users can also download the pre-trained model and directly leverage that for prediction. Finally, at the end of this phase, the prediction result is obtained.

The post-development phase in the pipeline enabled software reusability of the models. All of these projects have instructions in their `readme` or documentation explaining the usage and customization. For example, the project Deep ANPR [19] provides instructions for obtaining large training data, retraining the models, and build it for prediction. However, not all the projects enable reusability in the development pipelines. Only a few of them provides access to the modules by importing in new development scenario. For instance, Darkflow [75] let users access the `darkflow.net.build` module and use it in new application development. To increase the reusability of DS programs, it would be desired to consider similar access to the development pipeline of these large projects.

4.3 Organization of DS Pipeline in-the-Large

RQ8: How are the stages connected in pipeline in-the-large? The abstraction in DS projects is stricter than the DS programs described in §3. The projects are built in a modular fashion, i.e., one source file for a broad task (e.g., `train.py, model.py`). However, inside one specific file, there are many other possible stages, especially data *preprocessing* appears inside all the source files. In addition, the module connectivity is not linear. All of the modules

use external libraries for performing different tasks. As a result, there are a lot of interdependencies (both internal and external) in the DS pipeline. One immediate difference of these pipelines with traditional software is DS pipelines are heavily dependant on the data. For example, the project *Speech-to-Text-WaveNet* [43] requires a certain format of data. When we want to use that in a new situation, the data properties might be different. So, the usage pipelines would have a few additional stages. In some cases, the original pipeline is modified. Here, there are many sub-pipelines work together to build a large pipeline. However, we have not found any framework or common methodology these software are using. The different patterns of DS pipelines seek more advanced methodology or framework to build DS pipeline and release for production.

4.4 Characteristics of Pipelines in-the-Large

RQ9: What are the patterns found in the pipelines? The pipelines found in this setting can be categorized into 1) loosely coupled and 2) tightly coupled, based on their modularity in the pipeline. A high number of contributors in the project resulted in a more modular and loosely coupled pipeline. We found the loosely coupled ones are designed in a modular fashion such that one module (e.g., data cleaning, modeling) is built to be used by other modules as a library functionality. These modules accept the parameters as inputs and executed in *as-needed* basis in the pipeline. Usually, there are multiple entry-points in a loosely coupled pipeline and user has more flexibility. On the other hand, in a tightly coupled pipeline, the modules are stricter and integrated tightly with other modules in the software. There is only one or two entry-points to the pipeline, which automatically calls the modules sequentially. We have found that the projects with 6 or more contributors (~75%) have followed a loosely coupled architecture and projects with 1 to 5 contributors have followed tightly coupled architecture.

Finding 6: *There is need for integration and deployment tools for pipelines in-the-large but no common framework is used in practice.*

Although all the project under this study are written using Python, no project is using any common tool that integrates the DS modules and provides interface to the pipeline. Today, continuous integration and deployment (CI/CD) tools are widely used as a common practice in traditional software lifecycle to automate compilation, building, and testing [26, 37]. Additionally, from our subject studies of pipelines in theory, we found some CI/CD tools designed for ML pipelines are available [27, 47, 49]. Surprisingly, here we found no projects in pipeline in-the-large are using any CI/CD tools. However, the projects demonstrate the need of CI/CD in the repositories. In most of the projects, the environment setup and access to functionalities are configured through command lines scripts [3, 63]. Some projects have used *docker container* [6, 43, 66, 84] to set up the environment and run the pipeline. A few others have used Python notebooks that call different modules to integrate the pipeline stages [2, 16, 52]. 7 out of 21 projects used shell script for integration (e.g., sending HTTP request to download data, model reuse, etc.) [58, 85]. Although CI/CD frameworks e.g., TravisCI, GitHub Actions, Microsoft Azure DevOps are well established for traditional software such as web applications, several challenges remain for DS pipelines. Karlaš et al. outlined a major CI/CD challenge for DS pipeline i.e., unlike traditional software, ML testing is probabilistic [37]. The authors also pointed out to the gap between recent theoretical development of CI/CD in DS and their usage in practice. Hence, further research is needed to investigate the practical challenges of using CI/CD in data science projects.

5 DISCUSSION

Through our survey, empirical study, and analysis, we presented the state of data science pipeline that describes its semantics, design concerns, and the overall computational paradigm. Furthermore, the findings show the importance of studying the pipeline structure reminiscing the traditional software engineering works on design patterns and architecture.

In Theory: We presented all the representative stages and subtasks that inform the terminology of DS pipelines to be used in future works. By comparing with the available pipeline categories e.g., ML process, big data, and team processes, similarities and divergences can be directly identified. The presence of implicit feedback loops and lack of post-processing stages suggest ad hoc pipeline construction at the present time. This paper takes the first step towards comparable and reusable pipeline construction.

In-The-Small: The novel API-based analysis can be utilized for mining, extracting, and statically analyzing pipelines. We also elicited the notion of high-level and low-level pipelines, where the high-level abstraction has more similarity with that in theory. However, low-level pipelines exhibit many differences such as missing some stages, sparse data preparation, lack of modularization. The gap between low-level pipeline and its presentation in high-level can be reduced by making pipeline specific features available in development environment e.g., pipeline template in Jupyter Notebook. Additionally, the low-level pipelines often had an important stage *exploratory data analysis* missing which incurs much time and effort. Pipeline versioning techniques that consider data, model, and source code will facilitate storing such intermediate stages.

In-The-Large: Different pipeline patterns emerged in development and post-development phase of the large projects, which suggest creating separate *developer-centric* and *user-centric* pipeline structure. In tightly-coupled projects, the abstraction of stages are contingent upon the project-specific requirements and internal/external dependencies, whereas, in loosely-coupled projects, opportunities remain to build reusable sub-pipelines that span over project boundaries. Finally, there is a need for building automated CI/CD tools for data science specific testing, deployment, and maintenance.

To researchers and tool builders. (1) Modularization of DS pipeline into stages is challenging over all three representations. Further works are needed for standardization of pipeline architecture e.g., defining the interfaces of stages, enumerating externally visible properties, identifying domain-specific constraints, to develop reusable and interoperable DS pipelines. (2) We showed potentials for automatic pipeline analysis framework based on static analysis and API specifications. A few future directions would be mining pipeline patterns, (sub-)pipeline matching, and analyzing evolution of pipelines. (3) We confirmed several antipatterns of pipelines that call for actions e.g., CACE principle, pipeline jungles, scarce post-processing, implicit feedback loop, CI/CD challenges.

(4) Pipeline specific tool support is needed such as version control for data and models, storing intermediate results between stages.

To data scientists and engineers. (1) Pipelines are often built for a prototype in-the-small, which might not scale to a production level system. A well-designed pipeline in the early stage will help to identify key components, estimate cost, optimize, and manage risks better in the lifecycle. (2) The representative views of pipelines will serve as a checklist of stages and their connections. (3) Data and algorithms being the focus of DS pipeline, preprocessing and modeling activities are well understood and practiced by data scientists. However, they should emphasize more on including rigorous *evaluation* beyond accuracy such as robustness, fairness, explainability. (4) Many people with diverse backgrounds are involved in a DS pipeline. A pipeline with human-in-the-loop approach will benefit identifying collaboration points, decomposing tasks, and manage transdisciplinary teams. For example, a pipeline can encourage data scientists to choose a modeling technique that is maintainable. (5) Future work is necessary to identify the interactions of DS pipeline with the real world i.e., which stages receive inputs, when a checkpoint is saved, how results are disseminated, etc.

6 THREAT TO VALIDITY

For building the pipelines from DS programs, we relied on the APIs. One threat might be, what happens if the developer does not use any API for completing a stage in the program. We examined this possibility and found that DS programs are heavily dependent on libraries and external APIs and ML tasks are always performed using library APIs. Additionally, we validated the API-to-stage dictionary with the API documentation and manual verification.

Another possible threat is that the Kaggle solutions might not be representative. We adopted a two-fold strategy to mitigate that threat. First, we selected the solutions with the most number of votes and at least 10 votes. Second, we manually verified each program whether it is an end-to-end DS solution. Since some most voted solutions are only for introduction and exploratory analysis of the dataset, by manual verification, we excluded those programs. The GitHub projects are also taken from a previously published dataset containing DS repositories. We further filtered them based on the number of stars and whether they perform a DS task.

Moreover, since the chosen DS programs from Kaggle and GitHub are using Python as the primary language, another question might be on the generalization of them as DS programs. According to GitHub and *Stack Overflow*, Python has become the most growing language in recent times [28, 61]. In data science, Python is the most used language because of the availability of numerous ML, DL and data analysis packages such as Pandas, NumPy, TensorFlow, Keras, Caffe, Theano, Scikit-Learn and many more.

7 RELATED WORK

Many studies presented ML pipeline in their own context, which can not be generalized for all DS systems. Garcia et al. focused on building an iterative process with three main phases: development, training and inference. They described the interpretation of data and code while integrating the whole lifecycle [22]. Polyzotis et al. presented the challenges of data management in building production-level ML pipeline in Google around three broad themes: data understanding, data validation and cleaning, and data preparation [55, 56]. They also provided an overview of an end-to-end large-scale ML pipeline with a data point of view. Carlton E. Sapp defined ML concepts, business challenges, stages in the lifecycle, roles of DS teams with comprehensive end-to-end ML architecture [67]. This gives us a holistic understanding of the business processes (e.g., acquire, organize, analyze, deliver) of a DS project.

A few other studies try to capture the DS process by surveying and interviewing developers. Roh et al. surveyed the data collection techniques in the field of big data. They presented the workflow of data collection answering how to improve data or models in an ML system [62]. Another study identified the software engineering practices and challenges in building AI applications inside Microsoft development teams [4]. They found some key differences in AI software process compared to other domains. They considered a 9-stage workflow for DS software development. Hill et al. interviewed experienced AI developers and identified problems they face in each stage [25]. They also tried to compare the traditional software process and the AI process. Zhou presented her own view to build a better ML pipeline [87]. They presented three challenges in building ML pipelines: data quality, reliability and accessibility.

Some articles described ML applications and frameworks which present DS pipelines from industry. For example, *Databricks* provides high-level APIs for programming languages [27]. Team Data Science Process (TDSP) is an agile and iterative process to build intelligent applications inside Microsoft corporation [70]. In a US patent, the authors compared two data analytic lifecycles [74], and presented the difference in the set of parameters with respect to time and cost. CRoss Industry Standard Process for Data Mining (CRISP-DM) is a 6-stage comprehensive process model for data mining projects across any industry [83]. Google Cloud Blog described the workflow of an AI platform [24]. They explained tasks completed in each stage with respect to Google Cloud and TensorFlow[1]. Although there are many papers in the literature presenting DS pipeline, there is no comprehensive study that tries to understand and compare DS pipelines in theory and practice.

8 CONCLUSION

Many software systems today are incorporating a data science pipeline as their integral part. In this work, we argued that to facilitate research and practice on data science pipelines, it is essential to understand their nature. To that end, we presented a three-pronged comprehensive study of data science pipelines in theory, data science pipelines in-the-small, and data science pipelines in-the-large. Our study analyzed three datasets: a collection of 71 proposals for data science pipelines and related concepts in theory, a collection of 105 implementations of data science pipelines from Kaggle competitions to understand data science in-the-small, and a collection of 21 mature data science projects from GitHub to understand data science in-the-large. We have found that DS pipelines differ significantly between these settings. Specifically, a number of stages are absent in-the-small, and the DS pipelines have a more linear structure. The DS pipelines in-the-large have a more complex structure and feedback loops compared to the theoretical representations. We also contribute three representations of DS pipelines that capture the essence of our subjects in theory, in-the-small, and in-the-large.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[2] Waleed Abdulla. 2017. Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow. https://github.com/matterport/Mask_RCNN.

[3] Jesse Hu Alexis Chan, Octavio Arriaga. 2017. Autopilot-TensorFlow. https://github.com/SullyChen/Autopilot-TensorFlow.

[4] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software Engineering for Machine Learning: A Case Study. In *Proceedings of the 41st International Conference on Software Engineering*. ACM.

[5] Anonymous. 2021. Data Science Pipline Artifact. https://github.com/anonymous-authorss/DS-Pipeline.

[6] Octavio Arriaga. 2018. Face classification and detectionn. https://github.com/oarriaga/face_classification.

[7] Rob Ashmore, Radu Calinescu, and Colin Paterson. 2021. Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges. *ACM Comput. Surv.* 54, 5, Article 111 (may 2021). https://doi.org/10.1145/3453444

[8] Jakob Aungiers. 2019. LSTM Neural Network for Time Series Prediction. https://github.com/jaungiers/LSTM-Neural-Network-for-Time-Series-Prediction.

[9] Francine Berman, Rob Rutenbar, Brent Hailpern, Henrik Christensen, Susan Davidson, Deborah Estrin, Michael Franklin, Margaret Martonosi, Padma Raghavan, Victoria Stodden, et al. 2018. Realizing the potential of data science. *Commun. ACM* 61, 4 (2018), 67–72.

[10] Sumon Biswas, Md Johirul Islam, Yijia Huang, and Hridesh Rajan. 2019. Boa meets Python: a Boa dataset of data science software in Python language. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 577–581.

[11] Denny Britz. 2018. Convolutional Neural Network for Text Classification in Tensorflow. https://github.com/dennybritz/cnn-text-classification-tf.

[12] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Comput. Netw.* 41, 1 (Jan. 2003), 115–141. https://doi.org/10.1016/S1389-1286(02)00352-3

[13] CL Philip Chen and Chun-Yang Zhang. 2014. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information sciences* 275 (2014), 314–347.

[14] Z Ming Chen Mengda. 2018. reproduce MTCNN,a Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks. https://github.com/AITTSMD/MTCNN-Tensorflow.

[15] George E Dahl, Navdeep Jaitly, and Ruslan Salakhutdinov. 2014. Multi-task neural networks for QSAR predictions. *arXiv preprint arXiv:1406.1231* (2014).

[16] Sam Crane Dat Tran. 2018. Real-Time Object Recognition App with Tensorflow and OpenCV. https://github.com/datitran/object_detector_app.

[17] Edsger W Dijkstra. 1982. On the role of scientific thought. In *Selected writings on computing: a personal perspective*. Springer, 60–66.

[18] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, CA) *(ICSE'13)*. 422–431.

[19] Matthew Earl. 2016. Using neural networks to build an automatic number plate recognition system. https://github.com/matthewearl/deep-anpr.

[20] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*. Springer-Verlag, Berlin, Heidelberg, 406–431.

[21] Amir Gandomi and Murtaza Haider. 2015. Beyond the hype: Big data concepts, methods, and analytics. *International journal of information management* 35, 2 (2015), 137–144.

[22] Rolando Garcia, Vikram Sreekanti, Neeraja Yadwadkar, Daniel Crankshaw, Joseph E Gonzalez, and Joseph M Hellerstein. 2018. Context: The missing piece in the machine learning lifecycle. In *KDD CMI Workshop*, Vol. 114.

[23] David Garlan. 2000. Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*. 91–101.

[24] Google Cloud Blog. 2019. Machine Learning Workflow. https://cloud.google.com/ml-engine/docs/tensorflow/ml-solutions-overview.

[25] Charles Hill, Rachel Bellamy, Thomas Erickson, and Margaret Burnett. 2016. Trials and tribulations of developers of intelligent systems: A field study. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 162–170.

[26] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 197–207. https://doi.org/10.1145/3106237.3106270

[27] Sue Ann Hong and Tim Hunter. 2017. Build, Scale, and Deploy Deep Learning Pipelines with Ease. https://databricks.com/blog/2017/09/06/build-scale-deploy-deep-learning-pipelines-ease.html.

[28] GitHub Inc. 2019. Octoverse 2018. https://octoverse.github.com/projects.

[29] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *ESEC/FSE'19: The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[30] Kathryn Jepsen. 2014. The machine learning community takes on the Higgs. https://www.symmetrymagazine.org/article/july-2014/the-machine-learning-community-takes-on-the-higgs.

[31] Kaggle. 2021. Kaggle Notebook. www.kaggle.com/competitions.

[32] Kaggle. 2021. Kaggle Notebook. www.kaggle.com/thousandvoices/simple-lstm.

[33] Kaggle. 2021. Kaggle Notebook. https://www.kaggle.com/zfturbo/simple-ru-baseline-lb-0-9627.

[34] Kaggle. 2021. Kaggle Notebook. www.kaggle.com/seesee/siamese-pretrained-0-822.

[35] Kaggle. 2021. Kaggle Notebook. www.kaggle.com/willkoehrsen/start-here-a-gentle-introduction.

[36] Kaggle. 2021. Kaggle Notebook. https://www.kaggle.com/danielbecker/careervillage-org-recommendation-engine.

[37] Bojan Karlaš, Matteo Interlandi, Cedric Renggli, Wentao Wu, Ce Zhang, Deepak Mukunthu Iyappan Babu, Jordan Edwards, Chris Lauren, Andy Xu, and Markus Weimer. 2020. Building continuous integration services for machine learning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2407–2415. https://doi.org/10.1145/3394486.3403290

[38] Keras. 2021. Keras API Reference. https://keras.io/api/.

[39] Keras. 2021. Scikit-Learn API Reference. https://scikit-learn.org/stable/modules/classes.html.

[40] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. 2018. 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*. 1–11.

[41] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 220–242.

[42] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 96–107.

[43] Namju Kim. 2018. Speech-to-Text-WaveNet : End-to-end sentence level English speech recognition based on DeepMind's WaveNet and tensorflow. https://github.com/buriburisuri/speech-to-text-wavenet.

[44] Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. 1978. Characteristics of application software maintenance. *Commun. ACM* 21, 6 (1978), 466–471.

[45] Hui Miao, Amit Chavan, and Amol Deshpande. 2017. Provdb: Lifecycle management of collaborative analysis workflows. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*. ACM, 7.

[46] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. 2017. Towards unified data and lifecycle management for deep learning. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 571–582.

[47] Microsoft Blog. 2019. What are ML pipelines in Azure Machine Learning service? https://docs.microsoft.com/en-us/azure/machine-learning/service/concept-ml-pipelines.

[48] Justin J Miller. 2013. Graph database applications and concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, Vol. 2324.

[49] Valohai MLOps. 2020. What Is a Machine Learning Pipeline? https://valohai.com/machine-learning-pipeline/.

[50] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchỳ. 2019. Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review* (2019), 1–48.

[51] Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. 2016. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. ACM, 485–492.

[52] Alex Paino. 2017. Deep learning models trained to correct input errors in short, message-like text. https://github.com/atpaino/deep-text-corrector.

[53] Kyubyong Park. 2018. A TensorFlow Implementation of Tacotron: A Fully End-to-End Text-To-Speech Synthesis Model. https://github.com/Kyubyong/tacotron.

[54] David Lorge Parnas, Paul C Clements, and David M Weiss. 1985. The modular structure of complex systems. *IEEE Transactions on software Engineering* 3 (1985), 259–266.

[55] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1723–1726.

[56] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data Lifecycle Challenges in Production Machine Learning: A Survey. *ACM SIGMOD Record* 47, 2 (2018), 17–28.

[57] Christian Prehofer. 1997. Feature-oriented programming: A fresh look at objects. In *ECOOP'97 — Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 419–443.

[58] Chuan Qi. 2019. Caffe implementation of Google MobileNet SSD detection network. https://github.com/chuanqi305/MobileNet-SSD.

[59] Václav Rajlich. 2014. Software evolution and maintenance. In *Future of Software Engineering Proceedings*. 133–144.

[60] Muhammad Habib Rehman, Victor Chang, Aisha Batool, and Teh Ying Wah. 2016. Big data reduction framework for value creation in sustainable enterprises. *International Journal of Information Management* 36, 6 (2016), 917–928.

[61] David Robinson. 2017. The Incredible Growth of Python. https://stackoverflow.blog/2017/09/06/incredible-growth-python/.

[62] Yuji Roh, Geon Heo, and Steven Euijong Whang. 2019. A Survey on Data Collection for Machine Learning: a Big Data-AI Integration Perspective. *IEEE Transactions on Knowledge and Data Engineering* (2019).

[63] Eragon Ruan. 2019. Scene text detection based on ctpn (connectionist text proposal network). https://github.com/eragonruan/text-detection-ctpn.

[64] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

[65] Philippe Rémy. 2018. Deep Learning model to analyze a large corpus of clear text passwords. https://github.com/philipperemy/tensorflow-1.4-billion-password-analysis.

[66] David Sandberg. 2018. Face Recognition using Tensorflow. https://github.com/davidsandberg/facenet.

[67] Carlton E Sapp. 2017. Preparing and architecting machine learning. *Gartner Technical Professional Advice* (2017), 1–37.

[68] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine learning: The high interest credit card of technical debt. (2014).

[69] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*. 2503–2511.

[70] Roald Bradley Severtson. 2017. What is the Team Data Science Process? https://docs.microsoft.com/en-us/azure/machine-learning/team-data-science-process/overview.

[71] Mary Shaw and David Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc., USA.

[72] Guocong Song. 2017. Tensorflow-based Recommendation systems. https://github.com/songgc/TF-recomm.

[73] Marvin Teichmann. 2018. A Kitti Road Segmentation Model Implemented in Tensorflow. https://github.com/MarvinTeichmann/KittiSeg.

[74] Stephen Todd and David Dietrich. 2017. Computing resource re-provisioning during data analytic lifecycle. US Patent 9,619,550.

[75] Andrew Bagshaw Trieu. 2018. Real-time object detection and classification. https://github.com/thtrieu/darkflow.

[76] Tom Van Der Weide, Dimitris Papadopoulos, Oleg Smirnov, Michal Zielinski, and Tim Van Kasteren. 2017. Versioning for end-to-end machine learning pipelines. In *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning*. ACM, 2.

[77] Anthony J Viera, Joanne M Garrett, et al. 2005. Understanding interobserver agreement: the kappa statistic. *Fam med* 37, 5 (2005), 360–363.

[78] Ben Wagner. 2020. Accountability by design in technology research. *Computer Law & Security Review* 37 (2020), 105398.

[79] Zhiyuan Wan, Xin Xia, David Lo, and Gail C Murphy. 2019. How does machine learning change software development practices? *IEEE Transactions on Software Engineering* (2019).

[80] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring Execution Environments of Jupyter Notebooks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1622–1633.

[81] Hadley Wickham. 2019. Data science: how is it different to statistics? *IMS Bulletin* 48 (2019).

[82] Jeannette M Wing. 2019. The Data Life Cycle. *Harvard Data Science Review* (2019).

[83] Rüdiger Wirth and Jochen Hipp. 2000. CRISP-DM: Towards a standard process model for data mining. In *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*. Citeseer, 29–39.

[84] Max Woolf. 2018. Automatically "block" people in images (like Black Mirror) using a pretrained neural network. https://github.com/minimaxir/person-blocker.

[85] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V Le. 2018. A Tensorflow implementation of QANet for machine reading comprehension. https://github.com/NLPLearn/QANet.

[86] Charlie Bickerton Zhilin Yang, Zihang Dai. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. https://github.com/zihangdai/xlnet.

[87] Linda Zhou. 2019. How to Build a Better Machine Learning Pipeline. https://www.datanami.com/2018/09/05/how-to-build-a-better-machine-learning-pipeline.