

Enabling collaborative data science development with the Ballet framework

MICAH J. SMITH, Massachusetts Institute of Technology, USA

JÜRGEN CITO, TU Wien, Austria and Massachusetts Institute of Technology, USA

KELVIN LU, Massachusetts Institute of Technology, USA

KALYAN VEERAMACHANENI, Massachusetts Institute of Technology, USA

While the open-source software development model has led to successful large-scale collaborations in building software systems, data science projects are frequently developed by individuals or small teams. We describe challenges to scaling data science collaborations and present a conceptual framework and ML programming model to address them. We instantiate these ideas in Ballet, the first lightweight framework for collaborative, open-source data science through a focus on feature engineering, and an accompanying cloud-based development environment. Using our framework, collaborators incrementally propose feature definitions to a repository which are each subjected to software and ML performance validation and can be automatically merged into an executable feature engineering pipeline. We leverage Ballet to conduct a case study analysis of an income prediction problem with 27 collaborators, and discuss implications for future designers of collaborative projects.

CCS Concepts: • **Human-centered computing** → **Collaborative and social computing systems and tools**; *Empirical studies in collaborative and social computing*; • **Computing methodologies** → *Machine learning*; • **Software and its engineering** → *Collaboration in software development*.

Additional Key Words and Phrases: collaborative framework; machine learning; data science; feature engineering; feature definition; feature validation; streaming feature selection; mutual information

ACM Reference Format:

Micah J. Smith, Jürgen Cito, Kelvin Lu, and Kalyan Veeramachaneni. 2021. Enabling collaborative data science development with the Ballet framework. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW2, Article 431 (October 2021), 39 pages. <https://doi.org/10.1145/3479575>

1 INTRODUCTION

The open-source software development model has led to successful, large-scale collaborations in building software libraries, software systems, chess engines, scientific analyses, and more [12, 33, 57, 68, 82]. However, data science, and in particular, predictive machine learning (ML) modeling, has not similarly benefited from this development paradigm. Predictive modeling projects – where the output of the project is not general-purpose software but rather a specific trained model and library capable of serving predictions for new data instances – are rarely developed in open-source collaborations, and when they are, they generally have orders of magnitude fewer contributors (Table 1 and [20]).

There is great potential for large-scale, collaborative data science to address societal problems through community-driven analyses of public datasets [20, 41]. For example, the Fragile Families Challenge tasked researchers and data scientists with predicting outcomes, including GPA and

Authors' addresses: Micah J. Smith, micahs@mit.edu, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA; Jürgen Cito, juergen.cito@tuwien.ac.at, TU Wien, Vienna, Austria, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA; Kelvin Lu, kellu1997@gmail.com, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA; Kalyan Veeramachaneni, kalyanv@mit.edu, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA.

© 2021 Copyright held by the owner/author(s).

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Human-Computer Interaction*, <https://doi.org/10.1145/3479575>.

eviction, for a set of disadvantaged children and their families [74], and *crash-model* is an application for predicting car crashes and thereby directing safety interventions [44]. Such projects, which involve complex and unwieldy datasets, attract scores of interested citizen scientists and developers whose knowledge, insight, and intuition can be significant if they are able to contribute and collaborate.

To make progress toward this outcome, we must first better understand the capabilities and challenges of collaborative data science as projects scale beyond small teams. We make two main contributions toward this goal.

First, we ask, can we support larger collaborations in predictive modeling projects by applying successful open-source development models? In this work, we show that we can successfully adapt and extend the pull request development model to support collaboration during important steps within the data science process by introducing a new development workflow and ML programming model. Our approach is based on decomposing steps in the data science process into modular data science “patches” that can then be intelligently combined, representing objects like “feature definition,” “labeling function,” or “slice function.” Prospective collaborators work in parallel to write patches and submit them to a shared repository. Our framework provides the underlying functionality to support interactive development, test and merge high-quality contributions, and compose the accepted contributions into a single product. Projects built with Ballet are structured by these modular patches, yielding additional benefits including reusability, maintainability, reproducibility, and automated analysis. Our lightweight framework does not require any computing infrastructure beyond that which is freely available in open-source software development.

While data science and predictive modeling have many steps, we identify feature engineering as an important one that could benefit from a more collaborative approach. We thus instantiate these ideas in *Ballet*,¹ the first lightweight software framework for collaborative data science that supports collaborative feature engineering on tabular data. Developers can use Ballet to grow a shared feature engineering pipeline by contributing feature definitions, which are each subjected to software and ML performance validation. Together with *Assemblé* — a data science development environment customized for Ballet — even novice developers can contribute to large collaborations.

Second, we seek to better understand the opportunities and challenges in large open-source data science collaborations. Although much research has focused on elucidating diverse challenges involved in data science development [17, 20, 62, 77, 84, 99, 103], little attention has been given to large collaborations, partly due to the lack of real-world examples available for study. Leveraging Ballet as a probe, we create and conduct an analysis of *predict-census-income*, a collaboration to engineer features from raw individual survey responses to the U.S. Census American Community Survey (ACS) and predict personal income. We use a mixed-method software engineering case study approach to understand the experience of 27 developers collaborating on this task, focusing on understanding the experience and performance of participants from varying backgrounds, the characteristics of collaboratively developed feature engineering code, and the performance of the resulting model compared to alternative approaches. The resulting project is one of the largest ML modeling collaborations on GitHub, and outperforms both state-of-the-art tabular AutoML systems and independent data science experts. We find that both beginners and experts (in terms of their background in software development, data science, and the problem domain) can successfully contribute to such projects and that domain expertise in collaborators is critical. We also identify themes of goal clarity, learning by example, distribution of work, and developer-friendly workflows as important touchpoints for future design and research in this area.

¹<https://ballet.github.io>, <https://github.com/ballet/ballet>

Table 1. The number of unique contributors to large open-source collaborations in either software engineering or predictive machine learning modeling. ML modeling projects that are developed in open-source have orders of magnitude fewer contributors. (The methodology is described in Appendix A.)

Software engineering		ML modeling	
torvalds/linux	20,000+	tesseract-ocr/tesseract	130
DefinitelyTyped/DefinitelyTyped	12,600+	CMU-PCL/openpose	79
Homebrew/homebrew-cask	6,500+	deepfakes/faceswap	71
ansible/ansible	5,100+	JaidedAI/EasyOCR	62
rails/rails	4,300+	ageitgey/face_recognition	43
gatsbyjs/gatsby	3,600+	<i>predict-census-income</i> (this work)	27
helm/charts	3,400+	microsoft/CameraTraps	21
rust-lang/rust	3,000+	Data4Democracy/drug-spending	21

The remainder of this paper proceeds as follows. We first provide background on data science, collaborative data work, and open-source software development. We then describe a conceptual framework for collaboration in data science projects in Section 3, which we then apply to create the Ballet framework in Section 4. We next describe key components of Ballet’s support for collaborative feature engineering in Section 5, such as the feature definition abstraction, feature engineering pipeline abstraction, feature validation algorithms, and Assemblé development environment. Next, we introduce the method and procedures for the *predict-census-income* case study in Section 6 and analyze the results in Section 7. We discuss our work and results in Section 8, including future directions for designers and researchers in collaborative data science frameworks.

2 BACKGROUND AND RELATED WORK

2.1 Data science and feature engineering

The increasing availability of data and computational resources has led many organizations to turn to data science, or a data-driven approach to decision-making under uncertainty. Consequently, researchers have studied data science work practices on several levels, and the data science workflow is now understood as a complex, iterative process that includes many stages and steps. The stages can be summarized as Preparation, Modeling, and Deployment [62, 92] and encompass other smaller steps such as data cleaning and labeling, feature engineering, model development, monitoring, and analyzing bias. Within the larger set of data science workers involved in this process, we use *data science developers* to refer to those who write code in data science projects.

Within this broad setting, the step of feature engineering holds special importance in some applications. Feature engineering — also called feature creation, development, or extraction — is the process through which data scientists write code to transform raw variables into feature values that can be used as input to a machine learning model. (This process is sometimes grouped with data cleaning and preparation steps [62].) Features form the cornerstone of many data science tasks, including not only predictive ML modeling, in which a learning algorithm finds predictive relationships between feature values and an outcome of interest, but also causal modeling through propensity score analysis, clustering, business intelligence, and exploratory data analysis. Practitioners and researchers have widely acknowledged the importance of engineering good features for modeling success, particularly in predictive modeling [3, 27, 88].

Before we continue discussing feature engineering, we introduce some terminology that we will use throughout this paper. A *feature function* is a transformation applied to raw variables

that extracts *feature values*, or measurable characteristics and properties of each observation. A *feature definition* is source code written by a data scientist to create a feature function.² If many feature functions are created, they can be collected into a single *feature engineering pipeline* that executes the corresponding computational graph and concatenates the result into a *feature matrix*. For example, given a dataset about houses, a feature definition might define, in source code, a feature function that computes the distance from each house to the nearest grocery store, and imputes missing values with the mean distance of houses from the same zip code.

In an additional step in ML systems, feature engineering is increasingly augmented by applications like feature stores and feature management platforms to help with critical functionality like feature serving, curation, and discovery [40, 95].

Though there have been attempts to automate the feature engineering process in certain domains, including relational databases and time series analysis [21, 47, 49, 51], it is widely accepted that in many areas that involve large and complex datasets, like health and business analytics, human insight and intuition are necessary for success in feature engineering [4, 27, 81, 88, 89]. Human expertise is invaluable for understanding the complexity of a dataset, theorizing about different relationships, patterns, and representations in the data, and implementing these ideas in code in the context of the machine learning problem. Muller et al. [62] observe that “feature extraction requires an interaction of domain knowledge with practices of design-of-data.” As more people become involved in this process, there is a greater chance that impactful “handcrafted” feature ideas will be expressed; automation can be a valuable supplement to manual development. Indeed, Smith et al. [81] introduce a feature engineering platform in which contributors log in to a cloud platform and submit source code directly to a machine learning backend server.

In this paper, we build on understanding of the importance of human interaction within the feature engineering process by creating a workflow that supports collaboration in feature engineering as a component of a larger data science project. Ballet takes a lightweight and decentralized approach suitable for the open-source setting, an integrated development environment, and a focus on modularity and supporting collaborative workflows.

2.2 Collaborative and open data work

Just as we explore how multiple human perspectives enhance feature engineering, there has been much interest within the HCI and CSCW communities in achieving a broader understanding of collaboration in data work. For example, within a wider typology of *collaboratories* (collaborative organizational entities), Bos et al. [12] study both community data systems and open community contribution systems, such as the Protein Databank and Open Mind Initiative. Zhang et al. [103] show that data science workers in a large company are highly collaborative in small teams, using a plethora of tools for communication, code management, and more. Teams include workers in many roles such as researchers, engineers, domain experts, managers, and communicators [62], and include both experts and non-experts in technical practices [61]. In an experiment with a prototype machine learning platform, Smith et al. [81] show that 32 data scientists made contributions to a shared feature engineering project and that a model using all of their contributions outperformed a model from the best individual performer. Functionalities including a feature discovery method and a discussion forum helped data scientists learn how to use the platform and avoid duplicating work. Often, data science workers in teams coordinate around a shared work product, such as a data science pipeline, like in the MIDST project [24]. We expand on this body of work by extending

²Any of these terms may be referred to as “features” in other settings, but we make a distinction between the source code, the transformation applied, and the resulting values. In cases where this distinction is not important, we may also use “feature.”

the study of collaborative data work to feature engineering and by using the feature engineering pipeline, as a shared work product, to coordinate collaborators at a larger scale than previously observed.

One finding in common in previous studies is that data science teams are usually small, with six or fewer members [103]. There are a variety of explanations for this phenomenon in the literature. Technical and non-technical team members may speak “different languages” [41]. Different team members may lack common ground while observing project progress and may use different performance metrics [58]. Individuals may be highly specialized, and the lack of a true “hub” role on teams [103] along with the use of synchronous communication forms like telephone calls and in-person discussion [20] make communication challenges likely as teams grow larger. One possible implication of this finding is that, in the absence of other tools and processes, human factors of communication, coordination, and observability make it challenging for teams to work well at scale. Difficulties with validation and curation of feature contributions presented challenges for Smith et al. [81], which points to the limitations of existing feature evaluation algorithms. Thus, algorithmic challenges may complement human factors as obstacles to scaling data science teams. However, additional research is needed into the question of why data science collaborations are not larger. We provide a starting point through a case study analysis in this work.

Moving from understanding to implementation, other approaches to collaboration in data science work include crowdsourcing, synchronous editing, and competition. Unskilled crowd workers can be harnessed for feature engineering tasks within the Flock platform, such as by labeling data to provide the basis for further manual feature engineering [19]. Synchronous editing interfaces, like those of Google Colab and others for computational notebooks [31, 52, 90], could facilitate multiple users to edit a machine learning model specification, typically targeting pair programming or other very small groups. In our work, we explore *different-time, different-place* collaboration [78] in an attempt to move beyond the limitations of small group work. A form of collaboration is also achieved in data science competitions like the KDD Cup, Kaggle, and the Netflix Challenge [8] and using networked science hubs like OpenML [85]. While these have led to state-of-the-art modeling performance, there is no natural way for competitors to systematically integrate source code components into a single shared product. In addition, individual teams formed in competitions hosted on Kaggle are small, with the mean team having 2.6 members and 90% of teams having four or fewer members, similar to other types of data science teams as discussed above.³

Closely related is *open data analysis* or *open data science*, in which publicly available datasets are used by “civic hackers” and other technologists to address civic problems, such as visualizations of lobbyist activity and estimates of child labor usage in product manufacturing [20]. Existing open data analysis projects involve a small number of collaborators (median of three) and make use of synchronous communication [20]. A common setting for open data work is hackathons, during which volunteers collaborate with non-profit organizations to analyze their internal and open data. Hou and Wang [41] find that civic data hackathons create actionable outputs and improve organizations’ data literacy, relying on “client teams” to prepare data for analysis during the events and to broker relationships between participants. Looking more broadly at collaborative data work in open science, interdisciplinary collaborations in data science and biomedical science are studied in Mao et al. [58], who find that readiness of a team to collaborate is influenced by its organizational structures, such as dependence on different forms of expertise and the introduction of an intermediate broker role. In our work, we are motivated by the potential of open data analysis, but focus more narrowly on data science and feature engineering.

³Author’s calculation from Meta Kaggle [60] of all teams with more than one member.

2.3 Open-source development

The *open-source model* for developing software has been adopted and advanced by many individuals and through many projects [68]. In the open-source model, projects are developed publicly and source code and other materials are freely available on the internet; the more widely available the source code, the more likely it is that a contributor will find a defect or implement new functionality (“with enough eyes, all bugs are shallow”). With freely available source code, open-source projects may attract thousands of contributors: developers who fix bugs, contribute new functionality, write documentation and test cases, and more. With more contributors comes the prospect of conflicting patches, leading to the problem of *integration*. In order to support open-source developers, companies and organizations have made a variety of lightweight infrastructure and developer tooling freely available for this community, such as build server minutes and code analysis tools.

Closely associated with the open-source model is the *open-source software development process*, exemplified by the *pull-based development model* (or *pull request model*), a form of distributed development in which changes are pulled from other repositories and merged locally. As implemented on the social code platform GitHub, developers fork a repository to obtain their own copy and make changes independently; proposed changes (pull requests) are subject to discussions and code reviews in context and are analyzed by a variety of automated tooling. The pull request model has been successful in easing the challenges of integration at scale and facilitating massive software collaborations.

As of 2013, 14% of active repositories on GitHub used pull requests. An equal proportion used shared repositories without pull requests, while the remainder were single-developer projects [34]. Pull request authors use contextual discussions to cover low-level issues but supplement this with other channels for higher-level discussions [35]. Pull request integrators play a critical role in this process but can have difficulty prioritizing contributions at high volume [36]. Additional tooling has continued to grow in popularity partly based on these observations. Recent research has visited the use of modern development tools like continuous integration [86, 87, 104], continuous delivery [76], and crowdsourcing [55]. In this work, we specifically situate data science development within the open-source development process and explore what changes and enhancements are required for this development model to meet the needs of data scientists during a collaboration.

2.4 Testing and end-to-end ML

As part of our framework, we discuss the use of testing in continuous integration to validate contributions to data science pipelines. Other research has also explored the use of continuous integration in machine learning. Renggli et al. [69] investigate practical and statistical considerations arising from testing conditions on overall model accuracy in a continuous integration setting. Specific models and algorithms can be tested [37] and input data can be validated directly [13, 43]. Testing can also be tied to reproducibility in ML research [71]. We build on this work by designing and implementing the first system and algorithms that conduct ML testing at the level of individual feature definitions.

Feature engineering is just one of many steps involved in data science. Other research has looked at the entire endeavor from a distance, considering the end-to-end process of delivering a predictive model from some initial specification. DAWNbench [22] provides a standardized benchmarking environment for end-to-end deep learning models, while automated machine learning (AutoML) systems like AutoBazaar, AutoGluon, and commercial offerings from cloud vendors [30, 80] can automatically create predictive models for a variety of ML tasks. A survey of techniques used in AutoML, such as hyperparameter tuning, model selection, and neural architecture search, can be

found in Yao et al. [100]. On the other hand, researchers and practitioners are increasingly realizing that AutoML does not solve all problems and that human factors such as design, monitoring, and configuration are still required [16, 91, 94, 97]. In our experiments, we use an AutoML system to evaluate the performance of different feature sets without otherwise incorporating these powerful techniques into our framework.

3 CONCEPTUAL FRAMEWORK

Having noticed the success of open-source software development along with the challenges in collaborative data science, we set out to understand whether these two paradigms could complement each other, as well as to better grasp the current state of large-scale collaboration in data science. In this section, we describe the formation of key design concepts that underlie the creation of Ballet. Our methods reflect an iterative and informal design process that played out over the time we have worked on this problem as well as through two preliminary user studies (Section 6.1).

The pull request model (Section 2.3) has been particularly successful in enabling integration of proposed changes in shared repositories, and is already used for well over one million shared repositories on GitHub [35, 36]. We informally summarize this development model using the concepts of *product*, *patch*, and *acceptance procedure*. A software artifact is created in a shared repository (product). An improvement to the product is provided in a standalone source code contribution proposed as a pull request (patch). Not every contribution is worthy of inclusion, so high-quality and low-quality contributions must be distinguished (acceptance procedure). If accepted, the pull request can be merged.

Given the success of open-source development processes like the pull request model, we ask: *Can we apply the pull request model to data science projects in order to collaborate at a larger scale?*

3.1 Challenges

When we set out to apply the pull request model to data science projects, we found the model was not a natural fit, and discovered key challenges to address, which we describe here. As people embedded in data science work, we built on our own experience developing and researching feature engineering pipelines and other data science steps from a machine learning perspective. We also uncovered and investigated these challenges in preliminary user studies with prototypes of our framework (Section 6).

We synthesize these challenges in the context of the literature on collaborative data work and machine learning workflows. Previous work outside the context of open-source development has identified challenges in communication, coordination, observability, and algorithmic aspects (Section 2.2). In addition, a classic observation is that the number of possible direct communication channels in a collaborative software project scales quadratically with the number of developers [14]. As a result, at small scales, data science teams may use phone calls or video chats, with 74% communicating synchronously and in person [20]. At larger scales, like those made possible by the open-source development process, communication can take place more effectively through coordination around a shared work product, or through discussion threads and chat rooms.

Ultimately, we list four challenges for a collaborative framework to address. While not exhaustive, this list comprises the challenges we mainly focus on in this work, though we review and discuss additional ones in Sections 2 and 8.

C1 Task management. Working alone, data science developers often write end-to-end scripts that prepare the data, extract features, build and train model models, and tune hyperparameters [62, 72, 84]. How can this large task be broken down so that all collaborators can coordinate with each other and contribute without duplicating work?

Table 2. Addressing challenges in collaborative data science development by applying our design concepts in the Ballet framework.

<i>challenge</i>	<i>design concept</i>	<i>components of Ballet</i>
task management (C1)	data science patches (D1)	feature definitions (5.1), patch development in Assemblé (4.2)
tool mismatch (C2)	data science products in open-source workflows (D2)	feature engineering pipeline abstraction (5.2), patch contribution in Assemblé (4.2), CLI for project administration (4.1)
evaluating contributions (C3)	software and statistical acceptance procedures (D3)	feature API validation (5.3), streaming feature definition selection (5.3), continuous integration (4.1), Ballet Bot (4.1)
maintaining infrastructure (C4)	decentralized development (D4)	F/OSS package (4.1), free infrastructure and services (4.1), bring your own compute (4.2)

C2 *Tool mismatch*. Data science developers are accustomed to working in computational notebooks and have varying expertise with version control tools like git [17, 50, 72, 84]. How can these workflows be adapted to use a shared codebase and build a single product?

C3 *Evaluating contributions*. Prospective collaborators may submit code to a shared codebase. Some code may introduce errors or decrease the performance of the ML model [46, 48, 69, 81]. How can code contributions be evaluated?

C4 *Maintaining infrastructure*. Data science requires careful management of data and computation [77, 81]. Will it be necessary to establish shared data stores and computing infrastructure? Would this be expensive and require significant technical and DevOps expertise? Is this appropriate for the open-source setting?

3.2 Design concepts

To address these challenges, we think creatively about how certain data science steps might fit into a modified open-source development process. Our starting point is to look for processes in which some important functionality can be decomposed into smaller, similarly-structured patches that can be evaluated using standardized measures. We found that we could extend and adapt the pull request model to facilitate collaborative development in data science by following a series of four corresponding design concepts (Table 2), which form the basis for our framework.

D1 *Data science patches*. We identify steps of the data science process that can be broken down into many *patches* — modular source code units — which can be developed and contributed separately in an incremental process. For example, given a feature engineering pipeline, a patch is a new feature definition to be added to the pipeline.

D2 *Data science products in open-source workflows*. A usable data science artifact forms a *product* that is stored in an open-source repository. For example, when solving a feature engineering task, the product is an executable feature engineering pipeline. The composition of many patches from different collaborators forms a product that is stored in a repository on a source code host in which patches are proposed as individual pull requests. We design this process to accommodate collaborators of all backgrounds by providing multiple development interfaces.

Notebook-based workflows are popular among data scientists, so our framework supports creation and submission of patches entirely within the notebook.

- D3 *Software and statistical acceptance procedures.* ML products have the usual software quality measures along with statistical/ML performance metrics. Collaborators receive feedback on the quality of their work from both of these points of view.
- D4 *Decentralized development.* A lightweight approach is needed for managing code, data, and computation. In our decentralized model, each collaborator uses their own storage and compute, and we leverage existing community infrastructure for source code management and patch acceptance.

Through our experience researching and developing feature engineering pipelines and systems, as well as our review of the requirements and key characteristics of the feature engineering process, we found that we could extend and adapt the pull request model to facilitate collaboration in feature engineering.

Besides feature engineering, how and when can this framework be used? Several conditions must be met. First, the data science product must be able to be decomposed into small, similarly-structured patches. Otherwise, the framework has a limited ability to integrate contributions. Second, human knowledge and expertise must be relevant to the generation of the data science patches. Otherwise, automation or learning alone may suffice. Third, measures of statistical and ML performance, or good proxies thereof, must be definable at the level of individual patches. Otherwise, it is difficult for maintainers to reason about how and whether to integrate patches. Finally, dataset size and evaluation time requirements must not be excessive. Otherwise, we could not use existing services that are free for open-source development.⁴

So while we focus on feature engineering, this framework can apply to other steps in data science pipelines — for example, data programming with labeling functions and slicing functions [18, 67], which we leave for future work.

In the next section, we apply these design principles to describe a framework for collaboration on predictive modeling projects, referring back to these challenges and design concepts as they appear. Then in Section 5, we implement this general approach more specifically for collaborative feature engineering on tabular data.

4 AN OVERVIEW OF BALLET


Ballet extends the open-source development process to support collaborative data science by applying the concepts of data science patches, data science products and open-source workflows, software and statistical acceptance procedures, and decentralized development. As this process is complex, we illustrate how Ballet works by showing the experience of using it from three perspectives — maintainer, collaborator, and consumer — building on existing work that investigates different users' roles in open source development and ecosystems [6, 9, 39, 70, 102]. This development cycle is illustrated in Figure 1. In Section 5, we present a more concrete example of feature engineering on tabular datasets.

4.1 Maintainer

A maintainer wants to build a predictive model. They first define the prediction goal and upload their dataset. They install the Ballet package, which includes the core framework libraries and CLI. Next, they use the CLI to automatically render a new repository from the provided project

⁴As a rough guideline, running the evaluation procedure on the validation data should take no more than five minutes in order to facilitate interactivity.

a Create project



"We need to build a predictive model for X"

```
$ ballet quickstart
Generating new ballet project...
full_name [Your Name]: Alice
```

README.md

built with [ballet](#) [chat on gitter](#) [launch Assemblé](#)

Predict X

Join our data science collaboration! Your task is to develop and submit feature definitions to the project.

c Contribute structured patches

Propose new feature #37

Open bob wants to merge 1 commit into alice/ballet-predict-x from bob/ballet-predict-x-submit-feature

src/predict_x/features/contrib/user_bob/_init_.py

Empty file.

src/predict_x/features/contrib/user_bob/feature.py

```
+ from ballet import Feature
+ from ballet.eng.external import SimpleImputer
+ input = 'FHINS3C'
+ transformer = SimpleImputer(strategy='median')
+ feature = Feature(input, transformer)
```

e Continuous delivery

ballet-bot (bot) commented

After validation, your feature was accepted. It will be automatically merged into the project.

Beep beep - I'm a bot that helps manage Ballet projects. [Learn more about me](#) or [report a problem](#).

ballet-bot (bot) merged commit c03452e into alice:master

2 checks passed

b Develop feature definitions

```
[1] from ballet import b
    entities, targets = b.api.load_data()

[8] from ballet import Feature
    input = 'FHINS3C'
    transformer = None
    feature = Feature(input, transformer)

[24] b.validate_feature_api(feature)

INFO - Building features and target...
INFO - Building features and target...DONE
INFO - Feature is NOT valid; here is some advice for resolving the feature API issues:
INFO - NotMissingValuesCheck: When transforming sample data, the feature produces NaN values. If you reasonably expect these missing values, make sure you clean missing values as an additional step in your transformer list. For example: NullFiller(replacement=replacement)

[24] False

[25] b.validate_feature_acceptance(feature)


INFO - Building features and target...
INFO - Building features and target...DONE
INFO - Judging feature using RFRegressor lambda_1=0.01, lambda_2=0.01
INFO - I(feature ; target | existing_features) = .173

[25] True
```

d Automatic validation

Job	Python	State
Project structure validation	3.8	✓
Feature API validation	3.8	✓
ML performance validation	3.8	✓

f Pipeline usage



```
$ pip install github.com/alice/ballet-predict-x
$ python -m predict_x_engineer-features \
  --train-dir path/to/train/data \
  --path/to/test/data \
  path/to/features/output
```

Fig. 1. Collaborative data science development with the Ballet framework for a feature engineering project. (a) A maintainer with a dataset wants to mobilize the power of the open data science community to solve a predictive modeling task. They use the Ballet CLI to render a new project from a provided template and push to GitHub. (b) A developer interested in the project is tasked with writing feature definitions (defining Feature instances). They can launch the project in Assemblé, a custom development environment built on Binder and JupyterLab. Ballet's high-level client supports them in automatically detecting the project configuration, exploring the data, developing candidate feature definitions, and surfacing any API and ML performance issues. Once issues are fixed, the developer can submit the feature definition alone from within their messy notebook by selecting the code cell and using Assemblé's submit button. (c) The selected code is automatically extracted and processed as a pull request following the project structure imposed by Ballet. (d) In continuous integration, Ballet runs feature API and ML performance validation on this one feature definition. (e) Feature definitions that pass can be automatically and safely merged by the Ballet Bot. (f) Ballet will collect and compose this new feature definition into the existing feature engineering pipeline, which can be used by the community to modeling their own raw data (usage stylized).

template, which contains the minimal files and structure required for their project, such as directory structures, configuration files, and problem metadata. They define a task for collaborators: create and submit a data science patch that performs well (C1/D1) — for example, a feature definition that has high predictive power. The resulting repository contains a usable (if, at first, empty) data science

pipeline (C2/D2). After pushing to GitHub and enabling our CI tools and bots, the maintainer begins recruiting collaborators.

Collaborators working in parallel submit patches as pull requests with a careful structure provided by Ballet. Not every patch is worthy of inclusion in the product. As patches arrive from collaborators in the form of pull requests, the CI service is repurposed to run Ballet’s acceptance procedure such that only high-quality patches are accepted (C3/D3). This “working pipeline invariant” aligns data science pipelines with the aim of continuous delivery in software development [42]. In feature engineering, the acceptance procedure is a feature validation suite (Section 5.3) which marks individual feature definitions as accepted/rejected, and the resulting feature engineering pipeline on the default branch can always be executed to engineer feature values from new data instances.

One challenge for maintainers is to integrate data science patches as they begin to stream in. Unlike software projects where contributions can take any form, these types of patches are all structured similarly, and if they validate successfully, they may be safely merged without further review. To support maintainers, the *Ballet Bot*⁵ can automatically manage contributions, performing tasks such as merging pull requests of accepted patches and closing rejected ones. The process continues until either the performance of the ML product exceeds some threshold, or improvements are exhausted.

Ballet projects are lightweight, as our framework is distributed as a free and open-source Python package, and use only lightweight infrastructure that is freely available to open-source projects, like GitHub, Travis, and Binder (C4/D4). This avoids spinning up data stores or servers – or relying on large commercial sponsors to do the same.

4.2 Collaborators

A data science developer is interested in the project and wants to contribute. They find the task description and begin learning about Ballet and the project. They can review and learn from existing patches contributed by others and discuss ideas in an integrated chatroom. They begin developing a new patch in their preferred development environment (*patch development task*). When they are satisfied with its performance, they propose to add it to the upstream project at a specific location in the directory structure using the pull request model (*patch contribution task*). In designing Ballet, we aimed to make it as easy as possible for data scientists with varying backgrounds to accomplish the patch development and patch contribution tasks within open-source workflows.

The Ballet interactive client, included in the core library, supports developers in loading data, exploring and analyzing existing patches, validating the performance of their work, and accessing functionality provided by the shared project. It also decreases the time to beginning development by automatically detecting and loading a Ballet project’s configuration. Use of the client is shown in Figure 1, Panel B.

We enable several interfaces for collaborators to develop and submit patches like feature definitions, where different interfaces are appropriate for collaborators with different types of development expertise, relaxing requirements on the development style of collaborators. Collaborators who are experienced in open-source development processes can use their preferred tools and workflow to submit their patch as a pull request, supported by the Ballet CLI – the project is still just a layer built on top of familiar technologies like git. However, in preliminary user studies (Section 6.1), we found that adapting from usual data science workflows was a huge obstacle. Many data science developers we worked with had never successfully used open-source development processes to contribute to any shared project.

⁵<https://github.com/ballet/ballet-bot>

We addressed this by centering all development in the notebook with *Assemblé*,⁶ a cloud-based workflow and development environment for contribution to Ballet projects (C1/D1). We created a custom experience on top of community tooling that enables data science developers to develop and submit features entirely within the notebook.

Assemblé consists of several pieces. First, we support the patch development task by easing the process for developers to set up a computational environment and begin exploratory analysis. We tightly integrate *Assemblé* to be deployed on Binder,⁷ a free community service for cloud-hosted notebooks, such that *Assemblé* can be launched from every Ballet project from a hyperlinked “badge.” Following a step-by-step guide, developers create a patch within a messy, exploratory notebook. Second, we support the patch contribution task through a new, entirely in-notebook interface. We design and implement a JupyterLab extension for submitting code to a Ballet project using a simple, one-click “Submit” button. Developers isolate a patch, such as a single feature definition, in a code cell and click to submit it. This causes the patch to be first preprocessed and subjected to initial server-side validation before being formulated as a pull request on the collaborator’s behalf. Low-level version control details are abstracted away from users, and collaborators can view their new pull request with its validation results in a matter of seconds.

The submitted feature definitions are marked as accepted or rejected, and collaborators can proceed accordingly, either moving on to their next idea or reviewing diagnostic information and trying to fix a rejected submission.

4.3 Consumer

Ballet project consumers — members of the wider data science community — are now free to use the data science product however they desire, such as by executing the feature engineering pipeline to extract features from new raw data instances, making predictions for their own datasets. The project can easily be installed using a package manager like `pip` as a versioned dependency, and ML engineers can extract the machine-readable feature definitions into a feature store or other environment for further analysis and deployment. The “pipeline as code”/“pipeline as package” approach makes it easy to use and re-use the pipeline and helps address data lineage/versioning issues.

For example, the Ballet client for feature engineering projects exposes a method that fits the feature engineering pipeline on training data and can then engineer features from new data instances, and an instance of a pipeline that is already fitted on data from of the upstream project. These can be easily used in other library code.

5 COLLABORATIVE FEATURE ENGINEERING

We now briefly describe the design and implementation of a feature engineering plugin for Ballet for collaborative feature engineering projects. While we spoke in general terms about data science patches and products and the statistical acceptance procedure, here we define these concepts for feature engineering. Full details are presented in Appendix B.

We start from the insight that feature engineering can be represented as a dataflow graph over individual features. We structure code that extracts a group of feature values as a patch, calling these *feature definitions* and representing them with a `Feature` interface. Feature definitions are composed into a feature engineering pipeline product. Newly contributed feature definitions are accepted if they pass a two-stage acceptance procedure that tests both the feature API and its

⁶<https://github.com/ballet/ballet-assemble>

⁷<https://mybinder.org/>

```

from ballet import Feature
from ballet.eng import ConditionalTransformer
from ballet.eng.external import SimpleImputer
import numpy as np

input = 'Lot Area'
transformer = [
    ConditionalTransformer(
        lambda ser: ser.skew() > 0.75,
        lambda ser: np.log1p(ser)),
    SimpleImputer(strategy='mean'),
]
name = 'Lot area unskewed'
feature = Feature(input, transformer, name=name)

```

(a) A feature definition that conditionally unskews lot area (for a house price prediction problem) by applying a log transformation only if skew is present in the training data and then mean-imputing missing values.

```

from ballet import Feature
from ballet.eng.external import SimpleImputer
import numpy as np

input = 'JWAP' # Time of arrival at work
transformer = [
    SimpleImputer(missing_values=np.nan,
        strategy='constant', fill_value=0.0),
    lambda df: np.where((df >= 70) & (df <= 124), 1, 0),
]
name = 'If job has a morning start time'
description = 'Return 1 if the Work arrival time'
    <=> >=6:30AM and <=10:30AM'
feature = Feature(input, transformer, name=name,
    <=> description=description)

```

(b) A feature definition that defines a transformation of work arrival time (for a personal income prediction problem) by filling missing values and then applying a custom function.

Fig. 2. Examples of user-submitted feature definitions in two different Ballet projects. Feature functions are collected and imported from standalone modules like these and composed into a single feature engineering pipeline that separately applies each feature function and concatenates resulting feature values.

contribution to ML performance. Finally, the plugin specifies a module organization that allows features to be collected programmatically.

In Ballet, we create a flexible and powerful language for feature engineering that is embedded within the larger framework. It supports functionality such as learned feature transformations, supervised feature transformations, nested transformer steps, syntactic sugar for functional transformations, data frame-style transformations, and recovery from errors due to type incompatibility.

5.1 Feature definitions

A *feature definition* is the code that is used to extract semantically related feature values from raw data. Let us observe data $\mathcal{D} = (\mathbf{v}_i, \mathbf{y}_i)_{i=1}^n$, where $\mathbf{v}_i \in \mathcal{V}$ are the raw variables and $\mathbf{y}_i \in \mathcal{Y}$ is the target. In this formulation, the raw variable domain \mathcal{V} includes strings, missing values, categories, and other non-numeric types that cannot typically be inputted to learning algorithms. Thus our goal in feature engineering is to develop a learned map from \mathcal{V} to \mathcal{X} where $\mathcal{X} \subseteq \mathbb{R}^n$ is a real-valued feature space.

DEFINITION 1. A feature function is a learned map from raw variables in one data instance to feature values, $f : (\mathcal{V}, \mathcal{Y}) \rightarrow \mathcal{V} \rightarrow \mathcal{X}$.

Each feature function learns a specific map from \mathcal{D} , such that any information it uses, such as variable means and variances, is learned from the development (training) dataset. This formalizes the separation between development and testing data to avoid any *leakage* of information during the feature engineering process.

The `Feature` abstraction in Ballet is a way to express a feature function in code. It is a tuple (input, transformer). The input declares the variable(s) from \mathcal{V} that are needed by the feature, which will be passed to `transformer`, one or more transformer steps. Each transformer step implements the learned map via `fit` and `transform` methods, a standard interface in machine learning pipelines [15]. A data scientist then simply provides values for the input and transformer of a `Feature` object in their code. Additional metadata, like `name`, `description`, and `source`, can also be provided. Two example feature definitions are shown in Figure 2.

5.2 Feature engineering pipelines

Features are then composed together in a feature engineering pipeline. A *feature engineering pipeline* applies each feature function to the new data instances and concatenates the result, yielding a feature matrix X . This is implemented in Ballet by the `FeatureEngineeringPipeline` class that operates directly on data frames. Each feature function within the pipeline is passed the input variables it requires that it then transforms appropriately, internally using a sequence of one or more transformer steps (Figure 3).

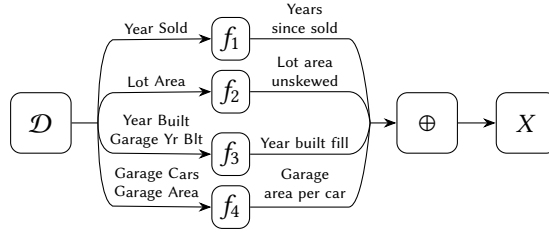


Fig. 3. A feature engineering pipeline for a house price prediction problem with four feature functions operating on six raw variables.

5.3 Acceptance procedures for feature definitions

Contributions of feature engineering code, just like other code contributions, must be evaluated for quality before being accepted, at risk of introducing errors, malicious behavior, or design flaws. For example, a feature function that produces non-numeric values can result in an unusable feature engineering pipeline. Large feature engineering collaborations can also be susceptible to “feature spam,” a high volume of low-quality feature definitions (submitted either intentionally or accidentally) that harm the collaboration [81]. Modeling performance can suffer and require an additional feature selection step — violating the working pipeline invariant — and the experience of other collaborators can be harmed who are not able to assume that existing feature definitions are high-quality.

To address these possibilities, we extensively validate feature definition contributions for software quality and ML performance, implemented as a test suite that is both exposed by the Ballet client and executed in CI for every pull request. Thus the same method that is used in CI for validating feature contributions is available to data science developers for debugging and performance evaluation in their development environment. Ballet Bot can automatically merge pull requests corresponding to accepted feature definitions and close pull requests corresponding to rejected feature definitions.

Feature API validation. User-contributed feature definitions should satisfy the `Feature` interface and successfully deal with common error situations, such as intermediate computations producing missing values. We fit the feature function to a separate subsampled training dataset in an isolated environment and extract feature values from subsampled training and validation datasets, failing immediately on any implementation errors. We then conduct a battery of 15 tests to increase confidence that the feature function would also extract acceptable feature values on unseen inputs (Table 6). Each test is paired with “advice” that can be surfaced back to the user to fix any issues (Figure 1).

Another part of feature API validation is an analysis of the changes introduced in a proposed PR to ensure that the required project structure is preserved and that the collaborator has not accidentally included irrelevant code that would need to be evaluated separately.

ML performance validation. A complementary aspect of the acceptance procedure is validating a feature contribution in terms of its impact on machine learning performance, which we cast as a streaming feature definition selection (SFDS) problem. This is a variant of streaming feature selection where we select from among feature definitions rather than feature values. Features that improve ML performance will pass this step; otherwise, the contribution will be rejected. Not only does this discourage low-quality contributions, but it provides a way for collaborators to evaluate their performance, incentivizing more deliberate and creative feature engineering.

We first compile requirements for an SFDS algorithm to be deployed in our setting, including that the algorithm should be stateless, support real-world data types (mixed discrete and continuous), and be robust to over-submission. While there is a wealth of research into streaming feature selection [93, 96, 101, 105], no existing algorithm satisfies all requirements. Instead, we extend prior work for our situation. Our SFDS algorithm proceeds in two stages (Appendix B.8). In the *acceptance* stage, we compute the conditional mutual information of the new feature values with the target given the existing feature matrix and accept the feature if it is above a dynamic threshold. In the *pruning* stage, existing features that have been made newly redundant by accepted features can be pruned. Maintainers of individual projects are also free to configure alternative ML performance validation algorithms given their needs, and Ballet provides several such implementations.

6 USER STUDIES

The success of a collaborative framework must be evaluated in the context of its usage. To that extent, we report on three user studies, including a large-scale case study.

6.1 Preliminary studies

We conducted several preliminary studies and evaluation steps during the iterative design process for Ballet.

Disease incidence. We first evaluated an initial prototype of Ballet in a user study with eight data science developers. All participants had at least basic knowledge of collaborative software engineering and open-source development practices (i.e., using git and pull requests). We explained the framework and gave a brief tutorial on how to write feature definitions. Participants were then tasked with writing feature definitions to help predict the incidence of dengue fever given historical data from Iquitos, Peru and San Juan, Puerto Rico [29], for which they were allotted 30 minutes. Three participants were successfully able to merge their first feature definition within this period, while the remainder produced features with errors or were unable to write one. In interviews, participants suggested that they found the Ballet framework helpful for structuring contributions and validating features, but were unfamiliar with writing feature engineering code in terms of feature definitions with separate fit/transform behavior (Section 5.1), and struggled to translate exploratory work in notebooks all the way to pull requests to a shared project. Based on this feedback, we created the `ballet.eng` library of feature engineering primitives (Appendix B.4) and created tutorial materials for new collaborators. We also began the design process that became the Assemblé environment that supports notebook-based developers (Section 4.2).

House price prediction. We evaluated a subsequent version of Ballet in a user study with 13 researchers and data science developers. This version included changes made since the first preliminary study and introduced an alpha version of Assemblé that did not yet include server-side or in-notebook validation functionality. Five of the participants had little to no prior experience contributing to open-source projects, six reported contributing occasionally, and two contributed frequently. All self-reported as intermediate or expert data scientists and Python developers. Participants were given a starter notebook that guided the development and contribution of feature

definitions, and documentation on the framework. They were tasked with writing feature definitions to help predict the selling price of a house given administrative data collected in Ames, Iowa [25]. After contributing, participants completed a short survey with a usability evaluation and provided semi-structured free-text feedback. Participants reported that they were moderately successful at learning to write and submit feature definitions but wanted more code examples. They also reported that they wanted to validate their features within their notebook using the same methods that were used in the automated testing in CI. Based on this feedback, among other improvements, we expanded and improved our feature engineering guide and the starter notebook. We also made methods for feature API validation and ML performance validation available in the interactive client (Section 5.3) and expanded the Assemblé server-side validation to catch common issues.

6.2 Case study

Finally, we conducted a full user study with the versions of Ballet and Assemblé described in this paper. To better understand the characteristics of live collaborative data science projects, we use a mixed-method software engineering case study approach [73]. The case study approach allows us to study the phenomenon of collaborative data science in its “real-life context.” This choice of evaluation methodology allows us to move beyond a laboratory setting and gain deeper insights into how large-scale collaborations function and perform. Through this study, we aim to answer the following four research questions:

RQ1 What are the most important aspects of our collaborative framework to support participant experience and project outcomes?

RQ2 What is the relationship between participant background and participant experience/performance?

RQ3 What are the characteristics of feature engineering code in a collaborative project?

RQ4 How does a collaborative model perform in comparison to other approaches?

These research questions build on our conceptual framework, allowing us to better understand the effects of our design choices as well as move forward our understanding of collaborative data science projects in general.

General procedures. We created an open-source project using Ballet, *predict-census-income*,⁸ to produce a feature engineering pipeline for personal income prediction. After invited participants consented to the research study terms, we asked them to fill out a pre-participation survey with background information about themselves, which served as the independent variables of our study. Next, they were directed to the public repository containing the collaborative project and asked to complete the task described in the project README. They were instructed to use either their preferred development environment or Assemblé. After they completed this task, we surveyed them about their experience.

Participants. In recruiting participants, we wanted to ensure that our study included beginners and experts in data science, software development, and survey data analysis (the problem domain). To achieve this, we compiled personal contacts with various backgrounds. After reaching these contacts, we then used snowball sampling to recruit more participants with similar backgrounds. We expanded our outreach by posting to relevant forums and mailing lists in data science development, Python programming, and survey data analysis. Participants were entered into a drawing for several nominal prizes but were not otherwise compensated.

⁸<https://github.com/ballet/predict-census-income>

Dataset. The input data is the raw survey responses to the 2018 U.S. Census American Community Survey (ACS) for Massachusetts (Table 3). This “Public Use Microdata Sample” (PUMS) has anonymized individual-level responses. Unlike the classic ML “adult census” dataset [53] which is highly preprocessed, raw ACS responses are a realistic form for a dataset used in an open data science project. Following Kohavi [53], we define the prediction target as whether an individual respondent will earn more than \$84,770 in 2018 (adjusting the original prediction target of \$50,000 for inflation), and filter a set of “reasonable” rows by keeping people older than 16 with personal income greater than \$100 with hours worked in a typical week greater than zero. We merged the “household” and “person” parts of the survey to get compound records and split the survey responses into a development set and a held-out test set.

Table 3. ACS dataset used in *predict-census-income* project.

	Development	Test
Number of rows	30085	10029
Entity variables	494	494
High income	7532	2521
Low income	22553	7508

Research instruments. Our mixed-method study synthesizes and triangulates data from five sources:

- *Pre-participation survey.* Participants provide background information about themselves, such as their education; occupation; self-reported background with data science, feature engineering, Python programming, open-source development, analysis of survey data, and familiarity with the U.S. Census/ACS specifically; and preferred development environment. Participants were also asked to opt in to telemetry data collection.
- *Assemblée telemetry.* To better understand the experience of participants who use Assemblée on Binder, we instrumented the extensions and installed an instrumented version of Ballet to collect usage statistics and some intermediate outputs. Once participants authenticated with GitHub, we checked with our telemetry server to see whether they had opted in to telemetry data collection. If they did so, we sent and recorded the buffered telemetry events.
- *Post-participation survey.* Participants who attempted and/or completed the task were asked to fill out a survey about their experience, including the development environment they used, how much time they spent on each sub-task, and which activities they did and functionality they used as part of the task and which of these were most important. They were also asked to provide open-ended feedback on different aspects, and to report how demanding the task was using the NASA-TLX Task Load Index [38], a workload assessment that is widely used in usability evaluations in software engineering and other domains [23, 75]. Participants indicate on a scale the temporal demand, mental demand, and effort required by the task, their perceived performance, and their frustration. The TLX score is a weighted average of responses (0=very low task demand, 100=very high task demand).
- *Code contributions.* For participants who progressed in the task to the point of submitting a feature definition to the upstream *predict-census-income* project, we analyze both the submitted source code as well as the performance characteristics of the submission.

- *Expert and AutoML baselines.* To obtain comparisons to solutions born from Ballet collaborations, we also obtain baseline solutions to the personal income prediction problem from outside data science experts and from a cloud provider’s AutoML service. First, we asked two outside data science experts working independently to solve the combined feature engineering and modeling task (without knowledge of the collaborative project).⁹ These experts were asked to work until they were satisfied with the performance of their predictive model, but not to exceed four hours, and were not compensated. Second, we used Google Cloud AutoML Tables,¹⁰ which supports structured data “as found in the wild,” to automatically solve the task, and ran it with its default settings until convergence.

We include the study description, pre-participation survey, and post-participation survey in our supplementary materials.

Analysis. After linking our data sources together, we performed a quantitative analysis to summarize results (e.g., participant backgrounds, average time spent) and relate measures to each other (e.g., participant expertise to cognitive load). Where appropriate, we also conducted statistical tests to report on significant differences for phenomena of interest. For qualitative analysis, we employed open and axial coding methodology to categorize the free-text responses and relate codes to each other to form emergent themes [10]. Two researchers first coded each response independently, and responses could receive multiple codes, which were then collaboratively discussed. We resolved disagreements by revisiting the responses, potentially introducing new codes in relation to themes discovered in other responses. We later revisited all responses and codes to investigate how they relate to each other, which led us to the emergent themes we present in our results. Finally, to understand the kind of source code that is produced in a collaborative data science setting, we performed lightweight program analysis to extract and quantify the feature engineering primitives used by our participants.

7 RESULTS

We present our results by interleaving the outcomes of quantitative and qualitative analysis (including verbatim quotes from free-text responses) to form a coherent narrative around our research questions.

In total, 50 people signed up to participate in the case study and 27 people from four global regions completed the task in its entirety. To the best of our knowledge, this makes our project the sixth largest ML modeling collaboration hosted on GitHub in terms of code contributors (Table 1). During the case study, 28 features were merged that together extract 32 feature values from the raw data. Of case study participants, 26 submitted at least one feature and 22 had at least one feature merged. As we went through participants’ qualitative feedback about their experience, several key themes emerged, which we discuss inline.

7.1 RQ1: Collaborative framework design

We identified several themes that relate to the design of frameworks for collaborative data science. We start by connecting these themes to the design decisions we made about Ballet.

Goal Clarity. The project-level goal is clear — to produce a predictive model. In the case of survey data that requires feature engineering, Ballet takes the approach of decomposing this data into individual goals via the feature definition abstraction, and asking collaborators to create and submit a patch that introduces a well-performing feature. Success in this task is validated using statistical

⁹<https://github.com/micahjsmith/ballet-cscw-2021>

¹⁰<https://cloud.google.com/automl-tables/>

tests (Section 5.3). However, the relationship between the individual and project goals may not always appear aligned to all participants. This negatively impacted some participants' experiences by introducing confusion about the direction and goal of their task. Some of the concerns expressed had to do with specific documentation elements, but others indicated a deeper confusion: "*Do the resulting features have to be 'meaningful' for a human or can they be built as combinations that maximize some statistical measure?*" (P2). Using the concept of software and statistical acceptance procedures, many high-quality features were merged into the project. However, the procedure was not fully transparent to the case study participants and may have prevented them from optimizing their features. While a feature that maximizes some statistical measure is best in the short term, it may constrain group productivity overall, as other participants benefit from being able to learn from existing features. And while having specific individual goals incentivizes high-quality feature engineering, participants are then less focused on the project-level goal and maintainers must either implement new project functionality themselves or define additional individual goals. This is a classic tension in designing collaborative mechanisms when it comes to appropriately structuring goals and incentives [63].

Learning by Example. We asked participants to rank the functionalities that were most important for completing the task, focusing both on creating and submitting feature definitions (Figure 4). For the patch development task, participants ranked most highly the ability to refer to example code written by fellow participants or project maintainers. This form of implicit collaboration was useful for participants to accelerate the onboarding process, learn new feature engineering techniques, and coordinate their efforts.

Distribution of Work. However, this led to feedback about difficulties in identifying how to effectively participate in the collaboration. Participants wanted the framework to provide more functionality to determine how to partition the input space: "*for better collaboration, different users can get different subsets of variables*" (P1). Some participants specifically asked for methods to review the input variables that had and had not been used and to limit the number of variables that one person would need to consider. This is a promising direction for future work, and similar ideas appear in automatic code reviewer recommendation [66]. Other participants, however, were satisfied with a more passive approach in which they used the Ballet client to programmatically explore existing feature definitions.

Cloud-Based Workflow. In terms of the patch contribution task, the most popular element by far was Assemblé. Importantly, all of the nine participants who reported that they "never" contribute to open-source software were able to successfully submit a PR to the *predict-census-income* project with Assemblé— seven in the cloud and the others locally.¹¹ Attracting participants like these who are not experienced data scientists is critical for sustaining large collaborations, and prioritizing interfaces that provide first-class support for collaboration can support these developers. The adaptation of the open-source development process reflected in Assemblé shows that concepts of open-source workflows and decentralized development did effectively address the aforementioned challenges for some developers.

In summary, we found that the feature definition abstraction, the cloud-based workflow in Assemblé, and the coordination and learning from referring to shared feature definitions were the aspects that contributed most to the participants' experiences. While the concept of data science patches makes significant progress toward addressing task management challenges, frictions remain around goal clarity and division of work, which should be addressed in future designs.

¹¹Local use involves installing JupyterLab and Assemblé on a local machine, rather than using the version running on Binder.

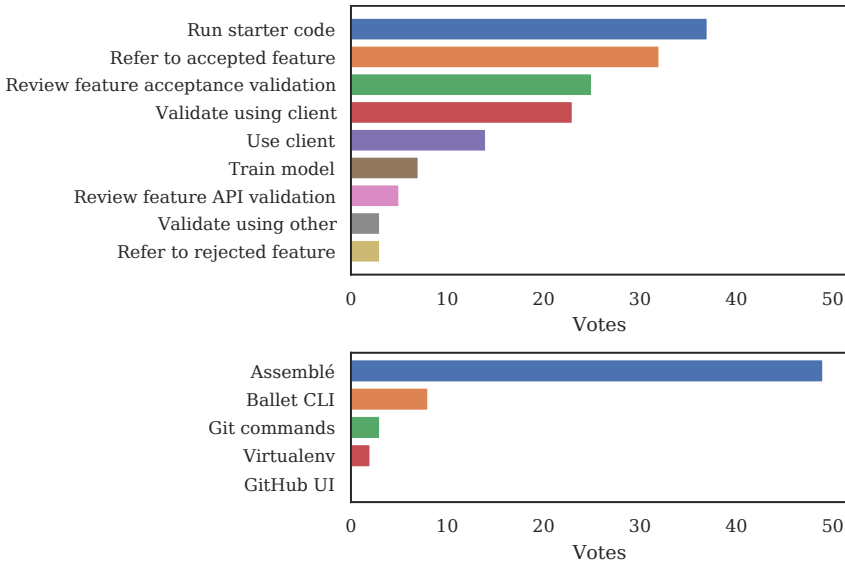


Fig. 4. Most important functionality within a collaborative feature engineering project, for the patch development task (top) and the patch contribution task (bottom), according to participant votes. Participants were asked to rank their top three items for creating feature definitions (awarded three, two, and one points in aggregating votes) and their top two items for submitting feature definitions (awarded two and one points in aggregating votes).

7.2 RQ2: Participant background, experience, and performance

In answering this question, we look at six dimensions of participants' backgrounds. Because many are complementary, for purposes of analysis, we collapse them into the broader categories of data science background, software development background, and domain expertise. Our main dependent variables for illustrating participant experience are the overall cognitive load (TLX - Overall) and total minutes spent on the task (Minutes Spent). Our main dependent variables for illustrating participant performance are two measures of the ML performance of each feature: its mutual information with the target and its feature importance as assessed by AutoML. We summarize the relationship between background, experience, and performance measures in Figure 5.

Beginners find the task accessible. Beginners found the task to be accessible, as across different backgrounds, beginners had a median task demand of 45.2 (lower is less demanding, $p_{25}=28.5$, $p_{75}=60.4$). The groups that found the task most demanding were those with little experience analyzing survey data or developing open-source projects.

Experts find the task less demanding but perform similarly. We found that broadly, participants with increased expertise in any of the background areas perceived the task as less demanding. However, data science and feature engineering experts spent more time working on the task than beginners did. They were not necessarily using this time to fix errors in their feature definitions, as they invoked the Ballet client's validation functions fewer times, according to telemetry data (16 times for experts, 33.5 times for non-experts). They may have been spending more time learning about the project and data without writing code. Then, they may have used their preferred methods to help evaluate their features during development. However, our hypothesis that experts would onboard faster than non-experts when measured by minutes spent learning

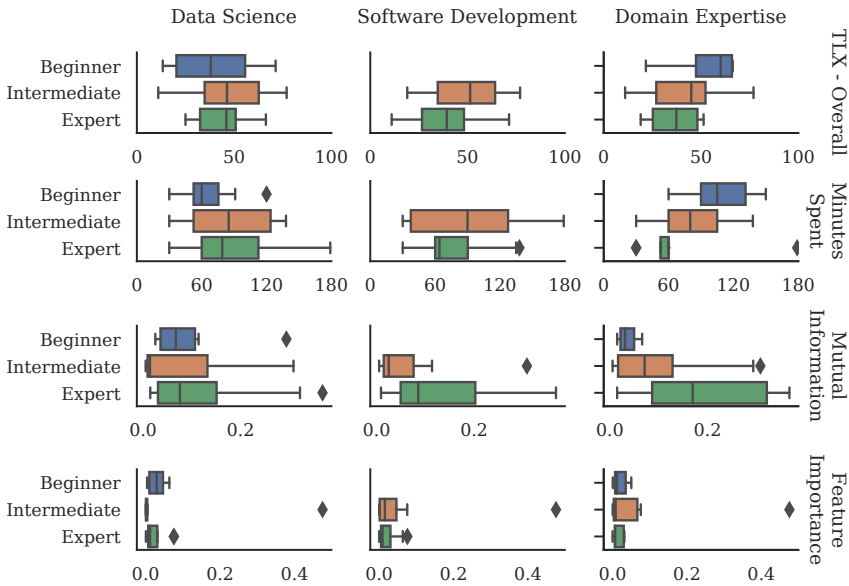


Fig. 5. Task demand, total minutes spent on task, mutual information of best feature with target on the test set, and total global feature importance assigned by AutoML service on development set, for participants of varying experience sorted by type of background.

about Ballet (a component of the total minutes spent) is rejected for data science background (Mann-Whitney $U=85.0$, Δ medians -6.0 minutes) and for software development background ($U=103.0$, Δ medians -1.5 minutes).

Domain expertise is critical. Of the different types of participant background, domain expertise had the strongest relationship with better participant outcomes. This is encouraging because it suggests that if collaborative data science projects attract experts in the project domain, these experts can be successful as long as they have data science and software development skills above a certain threshold and are supported by user-friendly tooling like Assemblé. One explanation for the relative importance of domain expertise is that participants can become overwhelmed or confused by *dataset challenges* with the wide and dirty survey dataset: “There are a lot of values in the data, and I couldn’t figure out the meaning of the values, because I didn’t know much about the topic” (P20). We speculate that given the time constraints of the task, participants who were more familiar with survey data analysis were able to allocate time they would have spent here to learning about Ballet or developing features. We find that beginners spent substantially more time learning about the prediction problem and data — a median of 36 minutes vs. 13.6 minutes for intermediate and expert participants (Mann-Whitney $U=36.5$, $p=0.064$, $n_1=6$, $n_2=21$).

7.3 RQ3: Collaborative feature engineering code

We were interested in understanding the kind of feature engineering code that participants write in this collaborative setting. Participants in the case study contributed 28 feature definitions to the project, which together extract 32 feature values. The features had 47 transformers, with most feature functions applying a single transformer to the input but some applying up to four transformers sequentially.

Feature engineering primitives. Participants collectively used 10 different feature engineering primitives (Appendix B.4). Our source code analysis shows that 17/47 transformers were `FunctionTransformer` primitives that can wrap standard statistical functions or are used by Ballet to automatically wrap anonymous functions. Use of these was broadly split between simple functions to process variables that needed minimal cleaning/transformation vs. complex functions that extracted custom mappings from ordinal or categorical variables based on a careful reading of the survey codebook.

Feature characteristics. These feature functions consumed 137 distinct variables from the raw ACS responses, out of a total of 494 present in the entities table. Most of these variables were consumed by just one feature, but several were transformed in different ways, such as SCHL (educational attainment), which was an input to five different features. Thus 357 variables, or 72%, were ignored by the collaborators. Some were ignored because they are not predictive of personal income. For example, the end-to-end AutoML model that operates directly on the raw ACS responses assigns a feature importance of less than 0.001 to 418 variables (where the feature importance values sum to one). However, there may still be missed opportunities by the collaborators, as the AutoML model assigns feature importance of greater than 0.01 to seven variables that were not used by any of the participants' features — such as RELP, which indicates the person's relationship to the "reference person" in the household and is an intuitive predictor of income because it allows the modeler to differentiate between adults who are dependents of their parents. This suggests an opportunity for developers of collaborative frameworks like Ballet to provide more formal direction about where to invest feature engineering effort — for example, by providing methods to summarize the inputs that have or have not been included in patches, in line with the *distribution of work* theme that emerged from participant responses and the challenge of task management. Of the features, 11/28 had a learned transformer while the remainder did not learn any feature engineering-specific parameters from the training data, and 14/47 transformers were learned transformers.

Feature definition abstraction. The Feature abstraction of Ballet yields a one-to-one correspondence between the task and feature definitions. This new programming paradigm required participants to adjust their usual feature engineering toolbox. For many respondents, this was a positive change, with benefits for reusability, shareability, and tracking prior features: *"It allows for a better level of abstraction as it raises Features up to their own entity instead of just being a standalone column"* (P16). For others, it was difficult to adjust, and participants noted challenges in learning how to express their ideas using transformers and feature engineering primitives and how to debug failures.

7.4 RQ4: Comparative performance

While we focus on better understanding how data science developers work together in a collaborative setting, ultimately one important measure of the success of a collaborative model is its ability to demonstrate good ML performance. To evaluate this, we compare the performance of the feature engineering pipeline built by the case study participants against several alternatives built from our baseline solutions we obtained from outside data science experts and a commercial AutoML service, Google Cloud AutoML Tables (Section 6.2).

We found that among these alternatives, the best ML performance came from using the Ballet feature engineering pipeline and passing the extracted feature values to AutoML Tables (Table 4). This hybrid human-AI approach outperformed end-to-end AutoML Tables and both of the outside experts. This finding also confirms previous results suggesting that feature engineering is sometimes difficult to automate, and that advances in AutoML have led to expert- or super-expert performance on clean, well-defined inputs.

Table 4. ML Performance of Ballet and alternatives. The AutoML feature engineering is not robust to changes from the development set and fails with errors on almost half of the test rows. But when using the feature definitions produced by the Ballet collaboration, the AutoML method outperforms human experts.

Feature Engineering	Modeling	Accuracy	Precision	Recall	F1	Failure rate
Ballet	AutoML	0.876	0.838	0.830	0.834	0.000
AutoML	AutoML	0.462	0.440	0.423	0.431	0.475
Ballet	Expert 1	0.828	0.799	0.707	0.734	0.000
Ballet	Expert 2	0.840	0.793	0.858	0.811	0.000
Expert 1	Expert 1	0.814	0.775	0.686	0.710	0.000
Expert 2	Expert 2	0.857	0.809	0.867	0.828	0.000

Qualitative differences. The three approaches to the task varied widely. Due to Ballet’s structure, participants spent all of their development effort on creating a small set of high-quality features. AutoML Tables performs basic feature engineering according to the inferred variable type (normalize and bucketize numeric variables, create one-hot encoding and embeddings for categorical variables) but spends most of its runtime budget searching and tuning models, resulting in a gradient-boosted decision tree for solving the census problem. The experts similarly performed minimal feature engineering (encoding and imputing); the resulting models were a minority class oversampling step followed by a tuned AdaBoost classifier (Expert 1) and a custom greedy forward feature selection step followed by a linear probability model (Expert 2).

8 DISCUSSION

In this section, we reflect on Ballet and on the case study presented in this paper with a particular eye toward scale, human factors, security, privacy, and the limitations of our research. We also discuss future areas of focus for HCI researchers and designers of collaborative frameworks.

8.1 Effects of scale

Although Ballet makes strides in scaling collaborative data science projects beyond small teams, we expect additional challenges to arise as collaborations are scaled even further.

The number of possible direct communication channels in a project scales quadratically with the number of developers [14]. At the scale of our case study, communication among collaborators can take place effectively through discussion threads, chat rooms, and shared work products. But projects with hundreds or thousands of developers require other strategies, such as search, filtering, and recommendation of relevant content. The metadata exposed by feature definitions (Section 5.1) makes it possible to explore these functionalities in future work.

A task management challenge that goes hand-in-hand with communication is distribution of work, a theme from our qualitative analysis in Section 7. Even at the scale of our case study, some collaborators wanted Ballet itself to support them in the distribution of feature engineering work. At larger scales, this need becomes more pressing if redundant work is to be avoided. In addition to strategies like partitioning the variable space and surfacing unused variables for developers, other solutions may include ticketing systems, clustering of variables and partitioning of clusters, and algorithms to rank variables by their impact after transformations have been applied.

8.2 Effects of culture

Data science teams are often made up of like-minded people from similar backgrounds. For example, Choi and Tausczik [20] report that most of the open data analysis projects they reviewed were comprised of people who already knew each other — partly because teammates wanted to be confident that everyone there had the required expertise.

Our more formalized and structured notion of collaboration may allow data science developers with few or no personal connections to form more diverse, cross-cultural teams. For example, the *predict-census-income* project included collaborators from four different global regions (North America, Europe, Asia, and the Middle East). The support for validating contributions like feature definitions with statistical and software quality measures may allow teammates to have confidence in each other even without knowing each other's backgrounds or specific expertise.

8.3 Security

Any software project that receives untrusted code must be mindful of security considerations. The primary threat model for Ballet projects is a well-meaning collaborator that submits poorly-performing feature definitions or inadvertently “breaks” the feature engineering pipeline, a consideration that partly informed the acceptance procedure in Section 5.3. Ballet's support for automatic merging of accepted features presents a risk that harmful code may be embedded within otherwise relevant features. While requiring a maintainer to give final approval for accepted features is a practical defense, defending against malicious contributions is an ongoing struggle for open-source projects [5, 26, 65].

8.4 Privacy

All data science projects require data. This can pose challenges for open data projects if the data they want to use is sensitive or confidential — for example, if it contains personally identifiable information. The main way to address this issue is to secure the dataset but open the codebase. In this formulation, access to the data is limited to those who have undergone training or signed a restricted use agreement. But at the same time, the entirety of the code, including the feature definitions, can be developed publicly without revealing any non-public information about the data. With this strategy, developers and maintainers must monitor submissions to ensure that data is not accidentally copied into source code files — a process that can be automated, similar to scanning for secure tokens and credentials [32, 59].

One alternative is to make the entire repository private, ensuring that only people who have been approved have access to the code and data. However, this curtails most of the benefits of open data science and makes it more difficult to attract collaborators.

Another alternative is to anonymize data or use synthetic data for development while keeping the actual data private and secure. Recent advances in synthetic data generation [64, 98] allow a synthetic dataset to be generated with the same schema and joint distribution as the real dataset, even for complex tables. This may be sufficient to allow data science developers to discover patterns and create feature definitions that can then be executed on the real, unseen dataset. This follows work releasing sensitive datasets for analysis in a privacy-preserving way using techniques like differential privacy [28]. Indeed, the U.S. Census is now using differential privacy techniques in the release of data products such as the ACS [1]. Analyses developed in an open setting could then be re-run privately on the original data according to the privacy budget of each researcher.

8.5 Interpretability and documentation

Zhang et al. [103] observe that data science workers rarely create documentation about their work during feature engineering, suggesting that human decision-making may be reflected in the data pipeline while “simultaneously becoming invisible.” This poses a risk for replicability, maintenance, and interpretability. In Ballet, the structure provided by our feature abstractions means that the resulting feature values have clear provenance and are interpretable, in the sense that the raw variables used and the exact transformations applied are easily surfaced and understood.

8.6 Feature maintenance

Just as software libraries require maintenance to fix bugs and make updates in response to changing APIs or dependencies, so too do feature definitions and feature engineering pipelines. Feature maintenance may be required in several situations. First, components from libraries used in a feature definition, such as the name or behavior of an imputation primitive, could change. Second, the schema of the target dataset could change, such as if a survey is conducted in a new year, with certain questions from prior years replaced with new ones.¹² Third, feature maintenance may be required due to distribution shift, in which new observations following the same schema have a different data distribution, causing the assumptions reflected in a feature definition to be invalidated.

Though we have focused mainly on the scale of a collaboration in terms of the number of code contributors, another important measure of scale is the length of time the project remains in a developed, maintained state, and as such is useful to consumers. As projects age, these secondary issues of feature maintenance, as well as dataset and model versioning and changing usage scenarios, become more salient.

A similar development workflow to the one presented in this paper could also be used for feature maintenance, and researchers have pointed out that the open-source model is particularly well suited for ensuring software is maintained [45]. Currently, Ballet focuses on supporting the addition of new features; to support the modification of existing features would require additional design considerations, such as how developers using Assemblé could indicate which feature should be updated/removed by their pull request. Automatically detecting the need for maintenance due to distribution shift or otherwise is an important research direction, and can be supported in the meantime by *ad hoc* statistical tests created by project maintainers.

8.7 Ethical considerations

As the field of machine learning rapidly advances, more and more ethical considerations are being raised, including of recent models [7, 11, 83]. The same set of concerns could also be raised about any model developed using Ballet. Addressing the underlying issues is beyond the scope of this paper. However, we emphasize that Ballet provides several benefits from an ethical perspective. The open-source setting means that models are open and transparent from the outset. Similarly, we focus on the development of models that aim to address societal problems, such as vehicle fatality prediction and government survey optimization.

8.8 Limitations

There are several limitations to our approach. Feature engineering is a complex process, and we have not yet provided support for several common practices (or potential new practices). For

¹²For example, the U.S. Census has modified the language used to ask about respondents’ race several times in response to an evolving understanding of this construct. A changelog [2] of a recently conducted survey compared to the prior year contained 42 entries.

example, many features are trivial to specify and can be enumerated by automated approaches [47], and some data cleaning and preparation can be performed automatically. We have referred the responsibility for adding these techniques to the feature engineering pipeline to individual project maintainers. Similarly, feature engineering with higher-level features, or meta-features, that operate on variable types rather than specific variables or that operate on existing feature values rather than raw variables, could enhance developer productivity.

Feature engineering is only one part of the larger data science process, albeit an important one. Indeed, many domains, including computer vision and natural language processing, have largely replaced manually engineered features with learned ones extracted by deep neural networks. Applying our conceptual framework to other aspects of data science, like data programming or ensembling in developing predictive models, can increase the impact of collaborations. Similarly, improving collaboration in other aspects of data work — like data journalism, exploratory data analysis, causal modeling, and neural network architecture design — remains an important challenge.

9 CONCLUSION

We provided a conceptual framework for collaboration in open-source data science projects and implemented it in Ballet. Our work develops the conceptual, algorithmic, engineering, and interaction approaches to move forward the vision of large-scale, collaborative data science. The success of the *predict-census-income* project shows the potential of this approach and gives further direction for framework developers.

ACKNOWLEDGMENTS

We'd like to thank the following people: members of the Data To AI Lab for feedback and pilot testing, participants in the *predict-house-prices* demonstration at MLSys 2020, participants in the *predict-census-income* case study, Ignacio Arnaldo, Zhuofan Xie, and Fahad Alhasoun. This work is supported in part by NSF Award 1761812.

REFERENCES

- [1] John M Abowd, Gary L Benedetto, Simson L Garfinkel, Scot A Dahl, Aref N Dajani, Matthew Graham, Michael B Hawes, Vishesh Karwa, Daniel Kifer, Hang Kim, Philip Leclerc, Ashwin Machanavajjhala, Jerome P Reiter, Rolando Rodriguez, Ian M Schmutte, William N Sexton, Phyllis E Singer, and Lars Vilhuber. 2020. *The Modernization of Statistical Disclosure Limitation at the U.S. Census Bureau*. Working Paper. U.S. Census Bureau.
- [2] American Community Survey Office. 2019. American Community Survey 2018 ACS 1-Year PUMS Files ReadMe. https://www2.census.gov/programs-surveys/acs/tech_docs/pums/ACS2018_PUMS_README.pdf. Accessed 2021-08-21.
- [3] Michael Anderson, Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher Ré, and Ce Zhang. 2013. Brainwash: A Data System for Feature Engineering. In *6th Biennial Conference on Innovative Data Systems Research*. 4.
- [4] Peter Bailis. 2020. Humans, Not Machines, Are the Main Bottleneck in Modern Analytics. <https://sisudata.com/blog/humans-not-machines-are-the-bottleneck-in-modern-analytics>.
- [5] Adam Baldwin. 2018. Details about the event-stream incident — The npm Blog. <https://web.archive.org/web/20190913183109/https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident>. Accessed 2018-11-30.
- [6] Flore Barcellini, Françoise Détéienne, and Jean-Marie Burkhardt. 2014. A situated approach of roles and participation in Open Source Software Communities. *Human-Computer Interaction* 29, 3 (2014), 205–255.
- [7] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency* (Virtual Event, Canada) (FAccT '21). Association for Computing Machinery, New York, NY, USA, 610–623. <https://doi.org/10.1145/3442188.3445922>
- [8] James Bennett and Stan Lanning. 2007. The Netflix Prize. In *Proceedings of KDD Cup and Workshop 2007*. 1–4.
- [9] Evangelia Berdou. 2010. *Organization in open source communities: At the crossroads of the gift and market economies*. Routledge.
- [10] Andreas Böhm. 2004. Theoretical Coding: Text Analysis in. *A companion to qualitative research* 1 (2004).
- [11] Tolga Bolukbasi, Kai-Wei Chang, James Y. Zou, Venkatesh Saligrama, and A. Kalai. 2016. Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings. In *NIPS*.
- [12] Nathan Bos, Ann Zimmerman, Judith Olson, Jude Yew, Jason Yerkie, Erik Dahl, and Gary Olson. 2007. From shared databases to communities of practice: A taxonomy of collaboratories. *Journal of Computer-Mediated Communication* 12, 2 (2007), 318–338.
- [13] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *Proceedings of the 2nd SysML Conference*. 1–14.
- [14] Frederick P Brooks Jr. 1995. *The mythical man-month: essays on software engineering*. Pearson Education.
- [15] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.
- [16] José P. Cambronero, Jürgen Cito, and Martin C. Rinard. 2020. AMS: Generating AutoML Search Spaces from Weak Specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event USA, 763–774. <https://doi.org/10.1145/3368089.3409700>
- [17] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, 1–12. <https://doi.org/10.1145/3313831.3376729>
- [18] Vincent Chen, Sen Wu, Alexander J Ratner, Jen Weng, and Christopher Ré. 2019. Slice-based Learning: A Programming Model for Residual Learning in Critical Data Slices. In *33rd Conference on Neural Information Processing Systems*. 11.
- [19] Justin Cheng and Michael S. Bernstein. 2015. Flock: Hybrid Crowd-Machine Learning Classifiers. *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15* (2015), 600–611.
- [20] Joohee Choi and Yla Tausczik. 2017. Characteristics of Collaboration in the Emerging Practice of Open Data Analysis. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing - CSCW '17*. ACM Press, Portland, Oregon, USA, 835–846. <https://doi.org/10.1145/2998181.2998265>
- [21] Maximilian Christ, Nils Braun, Julius Neuffer, and Andreas W Kempa-Liehr. 2018. Time series feature extraction on basis of scalable hypothesis tests (tsfresh—a python package). *Neurocomputing* 307 (2018), 72–77.
- [22] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. DAWNbench: An End-to-End Deep Learning Benchmark and Competition. In *ML Systems Workshops at NIPS*. 10.

- [23] Carl Cook, Warwick Irwin, and Neville Churcher. 2005. A User Evaluation of Synchronous Collaborative Software Engineering Tools. In *12th Asia-Pacific Software Engineering Conference (APSEC'05)*. 6 pp.–. <https://doi.org/10.1109/APSEC.2005.22>
- [24] Kevin Crowston, Jeff S. Saltz, Amira Rezgui, Yatish Hegde, and Sangseok You. 2019. Socio-Technical Affordances for Stigmergic Coordination Implemented in MIDST, a Tool for Data-Science Teams. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (Nov. 2019), 1–25. <https://doi.org/10.1145/3359219>
- [25] Dean De Cock. 2011. Ames, Iowa: Alternative to the Boston housing data as an end of semester regression project. *Journal of Statistics Education* 19, 3 (2011).
- [26] Alexandre Decan, Tom Mens, and Maelick Claes. 2016. On the Topology of Package Dependency Networks: A Comparison of Three Programming Language Ecosystems. In *Proceedings of the 10th European Conference on Software Architecture Workshops (Copenhagen, Denmark) (ECSAW '16)*. ACM, New York, NY, USA, Article 21, 4 pages.
- [27] Pedro Domingos. 2012. A Few Useful Things to Know about Machine Learning. *Commun. ACM* 55, 10 (Oct. 2012), 78–87. <https://doi.org/10.1145/2347736.2347755>
- [28] Cynthia Dwork. 2008. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*. Springer, 1–19.
- [29] Epidemic Prediction Initiative [n.d.]. Dengue Forecasting Project. <https://web.archive.org/web/20190916180225/https://predict.phiresearchlab.org/post/5a4fcc3e2c1b1669c22aa261>. Accessed 2018-04-30.
- [30] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data. *arXiv:2003.06505 [cs, stat]* (March 2020). arXiv:2003.06505 [cs, stat]
- [31] Utsav Garg, Viraj Prabhu, Deshraj Yadav, Ram Ramrakhya, Harsh Agrawal, and Dhruv Batra. 2018. Fabrik: An Online Collaborative Neural Network Editor. *arXiv e-prints*, Article arXiv:1810.11649 (2018), arXiv:1810.11649 pages. arXiv:1810.11649
- [32] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. 2020. Hidden in plain sight: Obfuscated strings threatening your privacy. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 694–707.
- [33] GNU [n.d.]. The GNU Operating System. <https://www.gnu.org>.
- [34] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An Exploratory Study of the Pull-Based Software Development Model. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, Hyderabad, India, 345–355. <https://doi.org/10.1145/2568225.2568260>
- [35] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 285–296. <https://doi.org/10.1145/2884781.2884826>
- [36] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. 2015. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 358–368. <https://doi.org/10.1109/ICSE.2015.55>
- [37] Roger B Grosse and David K Duvenaud. 2014. Testing MCMC code. In *2014 NIPS Workshop on Software Engineering for Machine Learning*. 1–8.
- [38] Sandra G. Hart and Lowell E. Staveland. 1988. *Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research*. Vol. 52. Elsevier, 139–183. [https://doi.org/10.1016/s0166-4115\(08\)62386-9](https://doi.org/10.1016/s0166-4115(08)62386-9)
- [39] Øyvind Hauge, Claudia Ayala, and Reidar Conradi. 2010. Adoption of open source software in software-intensive organizations—A systematic literature review. *Information and Software Technology* 52, 11 (2010), 1133–1154.
- [40] Jeremy Hermann and Mike Del Balso. 2017. Meet Michelangelo: Uber's Machine Learning Platform. <https://web.archive.org/web/20190713135310/https://eng.uber.com/michelangelo/>. Accessed 2019-07-01.
- [41] Youyang Hou and Dakuo Wang. 2017. Hacking with NPOs: Collaborative Analytics and Broker Roles in Civic Data Hackathons. *Proceedings of the ACM on Human-Computer Interaction* 1, CSCW (Dec. 2017), 1–16. <https://doi.org/10.1145/3134688>
- [42] Jez Humble and David Farley. 2010. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.
- [43] Nick Hynes, D Sculley, and Michael Terry. 2017. The Data Linter: Lightweight, Automated Sanity Checking for ML Data Sets. *Workshop on ML Systems at NIPS 2017* (2017).
- [44] Insight Lane 2019. Crash Model. <https://github.com/insight-lane/crash-model>.
- [45] Justin P. Johnson. 2006. Collaboration, Peer Review and Open Source Software. *Information Economics and Policy* 18, 4 (Nov. 2006), 477–497. <https://doi.org/10.1016/j.infoecopol.2006.07.001>
- [46] Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. 2020. Model Assertions for Monitoring and Improving ML Models. *arXiv:2003.01668 [cs]* (March 2020). arXiv:2003.01668 [cs]

- [47] James Max Kanter and Kalyan Veeramachaneni. 2015. Deep Feature Synthesis: Towards Automating Data Science Endeavors. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 1–10. <https://doi.org/10.1109/DSAA.2015.7344858>
- [48] 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*. ACM, Virtual Event CA USA, 2407–2415. <https://doi.org/10.1145/3394486.3403290>
- [49] Gilad Katz, Eui Chul Richard Shin, and Dawn Song. 2016. ExploreKit: Automatic Feature Generation and Selection. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, Barcelona, Spain, 979–984. <https://doi.org/10.1109/ICDM.2016.0123>
- [50] 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 (CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. <https://doi.org/10.1145/3173574.3173748>
- [51] Udayan Khurana, Deepak Turaga, Horst Samulowitz, and Srinivasan Parthasarathy. 2016. Cognito: Automated feature engineering for supervised learning. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*. IEEE, 1304–1307.
- [52] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87–90.
- [53] Ron Kohavi. 1996. Scaling Up the Accuracy of Naive-Bayes Classifiers: a Decision-Tree Hybrid. In *KDD*. 6.
- [54] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. 2004. Estimating mutual information. *Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics* 69, 6 (2004), 16.
- [55] Thomas D Latoza and André Van Der Hoek. 2016. Crowdsourcing in Software Engineering: Models, Opportunities, and Challenges. *IEEE Software* (2016), 1–13.
- [56] Haiguang Li, Xindong Wu, Zhao Li, and Wei Ding. 2013. Group feature selection with streaming features. *Proceedings - IEEE International Conference on Data Mining, ICDM* (2013), 1109–1114.
- [57] Linux [n.d.]. The Linux Kernel Organization. <https://www.kernel.org>.
- [58] Yaoli Mao, Dakuo Wang, Michael Muller, Kush R. Varshney, Ioana Baldini, Casey Dugan, and Aleksandra Mojsilović. 2019. How Data Scientists Work Together With Domain Experts in Scientific Collaborations: To Find The Right Answer Or To Ask The Right Question? *Proceedings of the ACM on Human-Computer Interaction* 3, GROUP (Dec. 2019), 1–23. <https://doi.org/10.1145/3361118>
- [59] Michael Meli, Matthew R McNiece, and Bradley Reaves. 2019. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. In *Network and Distributed Systems Security (NDSS) Symposium*. San Diego, CA, USA, 15. <https://doi.org/10.1145/3290605.3300356>
- [60] Meta Kaggle. 2021. Meta Kaggle: Kaggle’s public data on competitions, users, submission scores, and kernels. <https://www.kaggle.com/kaggle/meta-kaggle>. Version 539.
- [61] Justin Middleton, Emerson Murphy-Hill, and Kathryn T. Stolee. 2020. Data Analysts and Their Software Practices: A Profile of the Sabermetrics Community and Beyond. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW1 (May 2020), 1–27. <https://doi.org/10.1145/3392859>
- [62] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Q. Vera Liao, Casey Dugan, and Thomas Erickson. 2019. How Data Science Workers Work with Data: Discovery, Capture, Curation, Design, Creation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. <https://doi.org/10.1145/3290605.3300356>
- [63] William G Ouchi. 1979. A conceptual framework for the design of organizational control mechanisms. *Management science* 25, 9 (1979), 833–848.
- [64] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. 2016. The Synthetic Data Vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 399–410. <https://doi.org/10.1109/DSAA.2016.49>
- [65] Christian Payne. 2002. On the security of open source software. *Information systems journal* 12, 1 (2002), 61–78.
- [66] Zhenhui Peng, Jeehoon Yoo, Meng Xia, Sunghun Kim, and Xiaojuan Ma. 2018. Exploring How Software Developers Work with Mention Bot in GitHub. In *Proceedings of the Sixth International Symposium of Chinese CHI on – ChineseCHI '18*. ACM Press, 152–155. <https://doi.org/10.1145/3202667.3202694>
- [67] Alexander Ratner, Christopher De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. 2016. Data Programming: Creating Large Training Sets, Quickly. *Advances in neural information processing systems* 29 (2016), 3567–3575.
- [68] Eric Raymond. 1999. The cathedral and the bazaar. *Knowledge, Technology & Policy* 12, 3 (1999), 23–49.

- [69] Cedric Renggli, Bojan Karlaš, Bolin Ding, Feng Liu, Kevin Schawinski, Wentao Wu, and Ce Zhang. 2019. Continuous Integration of Machine Learning Models With ease.ml/ci: Towards a Rigorous Yet Practical Treatment. In *Proceedings of the 2nd SysML Conference*. 1–12.
- [70] Jeffrey A Roberts, Il-Horn Hann, and Sandra A Slaughter. 2006. Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the Apache projects. *Management science* 52, 7 (2006), 984–999.
- [71] Andrew Slavin Ross and Jessica Zosa Forde. 2018. Refactoring Machine Learning. In *Workshop on Critiquing and Correcting Trends in Machine Learning at NeurIPS 2018*. 1–6.
- [72] 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*. ACM, Montreal QC Canada, 1–12. <https://doi.org/10.1145/3173574.3173606>
- [73] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (Apr 2009), 131–164. <https://doi.org/10.1007/s10664-008-9102-8>
- [74] Matthew J. Salganik, Ian Lundberg, Alexander T. Kindel, et al. 2020. Measuring the Predictability of Life Outcomes with a Scientific Mass Collaboration. *Proceedings of the National Academy of Sciences* 117, 15 (April 2020), 8398–8403. <https://doi.org/10.1073/pnas.1915006117>
- [75] Iflaah Salman and Burak Turhan. 2018. Effect of time-pressure on perceived and actual performance in functional software testing. In *Proceedings of the 2018 International Conference on Software and System Process - ICSSP '18*. ACM Press, 130–139. <https://doi.org/10.1145/3202710.3203148>
- [76] Gerald Schermann, Jürgen Cito, Philipp Leitner, and Harald Gall. 2016. Towards Quality Gates in Continuous Delivery and Deployment. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–4. <https://doi.org/10.1109/ICPC.2016.7503737>
- [77] D Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. *Advances in Neural Information Processing Systems* (2015), 2494–2502.
- [78] Ben Shneiderman, Catherine Plaisant, Maxine Cohen, Steven Jacobs, Niklas Elmquist, and Nicholas Diakopoulos. 2016. *Designing the user interface: strategies for effective human-computer interaction*. Pearson.
- [79] Micah J. Smith. 2021. *Collaborative, Open, and Automated Data Science*. Ph.D. Thesis. Massachusetts Institute of Technology.
- [80] Micah J. Smith, Carles Sala, James Max Kanter, and Kalyan Veeramachaneni. 2020. The Machine Learning Bazaar: Harnessing the ML Ecosystem for Effective System Development. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, Portland, OR, USA, 785–800. <https://doi.org/10.1145/3318464.3386146>
- [81] Micah J. Smith, Roy Wedge, and Kalyan Veeramachaneni. 2017. FeatureHub: Towards Collaborative Data Science. In *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 590–600.
- [82] Stockfish [n.d.]. Stockfish: A strong open source chess engine. <https://stockfishchess.org>. Accessed 2019-09-05.
- [83] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 3645–3650. <https://doi.org/10.18653/v1/P19-1355>
- [84] Krishna Subramanian, Nur Hamdan, and Jan Borchers. 2020. Casual Notebooks and Rigid Scripts: Understanding Data Science Programming. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–5. <https://doi.org/10.1109/VL/HCC50065.2020.9127207>
- [85] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luís Torgo. 2013. OpenML: networked science in machine learning. *SIGKDD Explorations* 15 (2013), 49–60.
- [86] Bogdan Vasilescu, Stef van Schuylenburg, Jules Wulms, Aerebrenik Serebrenik, and Mark. G. J. van den Brand. 2014. Continuous Integration in a Social-Coding World: Empirical Evidence from GitHub. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 401–405. <https://doi.org/10.1109/ICSME.2014.62>
- [87] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015* (2015), 805–816.
- [88] Kalyan Veeramachaneni, Una-May O'Reilly, and Colin Taylor. 2014. Towards Feature Engineering at Scale for Data from Massive Open Online Courses. (2014). arXiv:1407.5238
- [89] Kiri L. Wagstaff. 2012. Machine Learning That Matters. In *Proceedings of the 29th International Conference on Machine Learning*. Edinburgh, Scotland, UK, 6.
- [90] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How Data Scientists Use Computational Notebooks for Real-Time Collaboration. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (Nov. 2019), 1–30. <https://doi.org/10.1145/3359141>

- [91] Dakuo Wang, Q. Vera Liao, Yunfeng Zhang, Udayan Khurana, Horst Samulowitz, Soya Park, Michael Muller, and Lisa Amini. 2021. How Much Automation Does a Data Scientist Want? *arXiv:2101.03970 [cs]* (Jan. 2021). [arXiv:2101.03970 \[cs\]](https://arxiv.org/abs/2101.03970)
- [92] Dakuo Wang, Justin D. Weisz, Michael Muller, Parikshit Ram, Werner Geyer, Casey Dugan, Yla Tausczik, Horst Samulowitz, and Alexander Gray. 2019. Human-AI Collaboration in Data Science: Exploring Data Scientists' Perceptions of Automated AI. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (Nov. 2019), 1–24. <https://doi.org/10.1145/3359313>
- [93] Jing Wang, Meng Wang, Peipei Li, Luoqi Liu, Zhongqiu Zhao, Xuegang Hu, and Xindong Wu. 2015. Online Feature Selection with Group Structure Analysis. *IEEE Transactions on Knowledge and Data Engineering* 27, 11 (2015), 3029–3041.
- [94] Qianwen Wang, Yao Ming, Zhihua Jin, Qiaomu Shen, Dongyu Liu, Micah J. Smith, Kalyan Veeramachaneni, and Huamin Qu. 2019. ATMSeer: Increasing Transparency and Controllability in Automated Machine Learning. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, Glasgow Scotland Uk, 1–12. <https://doi.org/10.1145/3290605.3300911>
- [95] Sarah Wooders, Peter Schafhalter, and Joseph E. Gonzalez. 2021. Feature Stores: The Data Side of ML Pipelines. <https://medium.com/riselab/feature-stores-the-data-side-of-ml-pipelines-7083d69bff1c>.
- [96] Xindong Wu, Kui Yu, Wei Ding, Hao Wang, and Xingquan Zhu. 2013. Online feature selection with streaming features. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 5 (2013), 1178–1192.
- [97] Doris Xin, Eva Yiwei Wu, Doris Jung-Lin Lee, Niloufar Salehi, and Aditya Parameswaran. 2021. Whither AutoML? Understanding the Role of Automation in Machine Learning Workflows. *arXiv:2101.04834 [cs]* (Jan. 2021). [arXiv:2101.04834 \[cs\]](https://arxiv.org/abs/2101.04834)
- [98] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. 2019. Modeling Tabular data using Conditional GAN. In *NeurIPS*.
- [99] Qian Yang, Jina Suh, Nan-Chen Chen, and Gonzalo Ramos. 2018. Grounding Interactive Machine Learning Tool Design in How Non-Experts Actually Build Models. *Proceedings of the 2018 on Designing Interactive Systems Conference 2018 - DIS '18* (2018), 573–584. <https://doi.org/10.1145/3196709.3196729>
- [100] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Yu-Feng Li, Wei-Wei Tu, Qiang Yang, and Yang Yu. 2019. Taking Human out of Learning Applications: A Survey on Automated Machine Learning. *arXiv:1810.13306 [cs, stat]* (Dec. 2019). [arXiv:1810.13306 \[cs, stat\]](https://arxiv.org/abs/1810.13306)
- [101] Kui Yu, Xindong Wu, Wei Ding, and Jian Pei. 2016. Scalable and Accurate Online Feature Selection for Big Data. *TKDD* 11 (2016), 16:1–16:39.
- [102] Liguoyu and Srini Ramaswamy. 2007. Mining cvs repositories to understand open-source project developer roles. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. IEEE, 8–8.
- [103] Amy X. Zhang, Michael Muller, and Dakuo Wang. 2020. How Do Data Science Workers Collaborate? Roles, Workflows, and Tools. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW1 (May 2020), 1–23. <https://doi.org/10.1145/3392826>
- [104] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The Impact of Continuous Integration on Other Software Development Practices: A Large-Scale Empirical Study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 60–71. <https://doi.org/10.1109/ASE.2017.8115619>
- [105] Jing Zhou, Dean Foster, Robert Stine, and Lyle Ungar. 2005. Streaming feature selection using alpha-investing. *Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining - KDD '05* (2005), 384.

Received January 2021; revised April 2021; revised July 2021; accepted July 2021

A LARGEST COLLABORATIONS

In Table 1, we show the number of unique contributors to selected large open-source collaborations in either software engineering or ML modeling, as of October 2020. In this section, we briefly describe the systematic review of GitHub projects that underlies this table.

We obtain a list of large software engineering collaborations as follows. We define a software engineering collaboration as a source code repository for a software library, framework, or application. We use GitHub's API to query for repositories with at least 50 forks and 250 stars. We exclude repositories that have the topics `awesome` or `interview` as these represent large crowd-sourced wikis/resources that are not actually software projects. For each repository, we count the number of contributors by scraping the count from the "Contributors" section of the repository home page. As computed by GitHub, this represents the number of unique users (including developers whose emails appear in the git log but do not have an associated GitHub account) who have at least one commit, excluding merge commits and empty commits. In the case that the scrape fails, we fall back to using the GitHub API "List repository contributors" endpoint and count the number of results.

We obtain a list of large ML modeling collaborations as follows. First, we define an ML modeling collaboration as a source code repository for a predictive ML model. We do not impose any requirements on dataset availability. We first search for projects with topics including `kaggle-competition` and `tensorflow-model`, for projects that mention the term `cookiecutterdatascience` in the README, and filter to results with greater than five forks and zero stars. We also searched for projects created by well-known organizations like Google Research, Intel, IBM, and NVIDIA. Then, we augment this list with an additional list of data science projects of interest that we manually curate as part of our research (32 in total). This list includes projects from organizations such as dssg (Data Science for Social Good), Data For Democracy, and Microsoft's AI for Earth initiative. We use the same methodology to count contributions as in the case of software projects.

There are some limitations to our methodology for identifying large data science collaborations. There may be additional data science search terms or projects that we did not find. And we purposefully excluded projects that defined many different models in subdirectories, so-called "model zoos" or "model gardens." In these projects, each subdirectory is a different model contributed by a different set of contributors, and counting contributions per model is a non-trivial task. For example, `tensorflow/models` has 709 contributors but at least 50 model implementations. Similarly, we excluded projects that are frameworks for applying one of many possible models, such as the `OpenNMT` framework for machine translation or Facebook Research's `detectron2` platform for object detection. Furthermore, this represents a narrow view of contribution that excludes creating and triaging bug reports, reviewing code, and creating documentation outside of the source tree.

Replication files are available here: <https://github.com/micahjsmith/ballet-cscw-2021>.

B A LANGUAGE FOR FEATURE ENGINEERING

In this section, we provide more detail on feature engineering within Ballet. Additional details are also presented in Smith [79, Chapter 3].

We start from the insight that feature engineering can be represented as a dataflow graph over individual features. We structure code that extracts a group of feature values as a patch, calling these *feature definitions* and representing them with a `Feature` interface. Feature definitions are composed into a feature engineering pipeline product. Newly contributed feature definitions are accepted if they pass a two-stage acceptance procedure that tests both the feature API and its contribution to ML performance. Finally, the plugin specifies the organization of modules within a repository to allow features to be collected programmatically.

In Ballet, we create a flexible and powerful language for feature engineering that is embedded within the larger framework. It supports functionality such as learned feature transformations, supervised feature transformations, nested transformer steps, syntactic sugar for functional transformations, data frame-style transformations, and recovery from errors due to type incompatibility.

B.1 Feature definitions

A *feature definition* is the code that is used to extract semantically related feature values from raw data. Let us observe data $\mathcal{D} = (\mathbf{v}_i, y_i)_{i=1}^n$, where $\mathbf{v}_i \in \mathcal{V}$ are the raw variables and $y_i \in \mathcal{Y}$ is the target. In this formulation,

the raw variable domain \mathcal{V} includes strings, missing values, categories, and other non-numeric types that cannot typically be inputted to learning algorithms. Thus our goal in feature engineering is to develop a learned map from \mathcal{V} to \mathcal{X} where $\mathcal{X} \subseteq \mathbb{R}^n$ is a real-valued feature space.

DEFINITION 2. A feature function is a learned map from raw variables in one data instance to feature values, $f : (\mathcal{V}, \mathcal{Y}) \rightarrow \mathcal{V} \rightarrow \mathcal{X}$.

We indicate the map learned from a specific dataset \mathcal{D} by $f^{\mathcal{D}}$, i.e., $f^{\mathcal{D}}(v) = f(\mathcal{D})(v)$.

A feature function can produce output of different dimensionality. Let $q(f)$ be the dimensionality of the feature space \mathcal{X} for a feature f . We call f a *scalar-valued feature* if $q(f) = 1$ or a *vector-valued feature* if $q(f) > 1$. For example, the embedding of a categorical variable, such as a one-hot encoding, would result in a vector-valued feature.

We can decompose a feature function into two parts, its *input projection* and its *transformer steps*. The input projection is the subspace of the variable space that it operates on, and the transformer steps, when composed together, equal the learned map on this subspace.

DEFINITION 3. A feature input projection is a projection from the full variable space to the feature input space, the set of variables that are used in a feature function, $\pi : \mathcal{V} \rightarrow \mathcal{V}$.

DEFINITION 4. A feature transformer step is a learned map from the variable space to the variable space, $f_i : (\mathcal{V}, \mathcal{Y}) \rightarrow \mathcal{V} \rightarrow \mathcal{V}$.

We require that individual feature transformer steps compose together to yield the feature function, where the first step applies the input projection and the last step maps to \mathcal{X} rather than \mathcal{V} . That is, each transformer step applies some arbitrary transformation as long as the final step maps to allowed feature values.

$$\begin{aligned} f_0 &: (V, Y) \mapsto \pi \\ f_i \circ f_{i-1} &: (V, Y) \mapsto f_i(f_{i-1}(\dots, Y)(V), Y) \\ f_n &: (\mathcal{V}, \mathcal{Y}) \rightarrow \mathcal{V} \rightarrow \mathcal{X} \\ f &: (V, Y) \mapsto f_n(\dots (f_1(f_0(V, Y)(V), Y) \dots), Y) \end{aligned}$$

The **Feature** abstraction in Ballet is a way to express a feature function in code. It is a tuple (input, transformer). The input declares the variable(s) from \mathcal{V} that are needed by the feature, which will be passed to **transformer**, one or more transformer steps. Each transformer step implements the learned map via **fit** and **transform** methods, a standard interface in machine learning pipelines [15]. A data scientist then simply provides values for the input and transformer of a **Feature** object in their code. Additional metadata, including **name**, **description**, **output**, and **source**, is also exposed by the **Feature** abstraction. For example, the feature **output** is a column name (or set of column names) for the feature value (or feature values) in the resulting feature matrix; if it is not provided by the data scientist, it can be inferred by Ballet using various heuristics.

Two example feature definitions are shown in Figure 2.

B.2 Learned feature transformations

In machine learning, we estimate the generalization performance of a model by evaluating it on a set of test observations that are unseen by the model during training. *Leakage* is a problem in which information about the test set is accidentally exposed to the model during training, artificially inflating its performance on the test set and thus underestimating generalization error.

Each feature function learns a specific map $\mathcal{V} \rightarrow \mathcal{X}$ from \mathcal{D} , such that any parameters it uses, such as variable means and variances, are learned from the development (training) dataset. This formalizes the separation between development and testing data to avoid any leakage of information during the feature engineering process.

As in the common pattern, a feature engineering pipeline by itself or within a model has two stages within training: a fit stage and a transform stage. During the fit stage, parameters are learned from the training data and stored within the individual transformer steps. During the transform stage, the learned parameters are

used to transform the raw variables into a feature matrix. The same parameters are also used at prediction time.

B.3 Nested feature definitions

Feature definitions or feature functions can also be nested within the `transformer` field of another feature. If an existing feature is used as one transformer step in another feature, when the new feature function is executed, the nested feature is executed in a sub-procedure and the resulting feature values are available to the new feature for further transformation. Data scientists can also introspect an existing feature to access its own `input` and `transformer` attributes and use them directly within a new feature.

Through its support for nested feature definitions, Ballet allows collaborating data scientists to define an arbitrary directed acyclic dataflow graph from the raw variables to the feature matrix.

B.4 Feature engineering primitives

Many features exhibit common patterns, such as scaling or imputing variables using simple procedures. And while some features are relatively simple and have no learned parameters, others are more complicated to express in a fit/transform style. Data scientists commonly extract these more advanced features by manipulating development and test tables directly using popular data frame libraries, often leading to leakage. In preliminary studies (Section 6.1), we found that data scientists sometimes struggled to create features one at a time, given their familiarity with writing long processing scripts. Responding to this feedback, we provided a library of *feature engineering primitives* that implements many common utilities and learned transformations.

DEFINITION 5. *A feature engineering primitive is a class that can be instantiated within a sequence of transformer steps to express a common feature engineering pattern.*

This library, `ballet.eng`, includes `ConditionalTransformer`, which applies a secondary transformation depending on whether a condition is satisfied on the development data, and `GroupwiseTransformer`, which learns a transformer separately for each group of a group-by aggregation on the development set. We also organize and re-export 77 primitives¹³ from six popular Python libraries for feature engineering, such as scikit-learn's `SimpleImputer` (Table 5).

Table 5. Feature engineering primitives implemented or re-exported in `ballet.eng`, by library.

Library	Number of primitives
<code>ballet</code>	16
<code>category_encoders</code>	17
<code>feature_engine</code>	29
<code>featuretools</code>	1
<code>skits</code>	10
<code>scikit-learn</code>	19
<code>tsfresh</code>	1

B.5 Feature engineering pipelines

Features are then composed together in a feature engineering pipeline.

DEFINITION 6. *Let f_1, \dots, f_m be a collection of feature functions, \mathcal{D} be a development dataset, and $\mathcal{D}' = (V', Y')$ be a collection of new data instances. A feature engineering pipeline $\mathcal{F} = \{f_i\}_{i=1}^m$ applies each feature function to the new data instances and concatenates the result, yielding the feature matrix*

$$X = \mathcal{F}^{\mathcal{D}}(V') = f_1^{\mathcal{D}}(V') \oplus \dots \oplus f_m^{\mathcal{D}}(V').$$

¹³As of ballet v0.19.

A feature engineering pipeline can be thought of as similar to a collection of feature functions. It is implemented in Ballet in the `FeatureEngineeringPipeline` class.

In Ballet’s standard configuration, only feature functions that have been explicitly defined by data scientists (and accepted by the feature validation) are included in the feature engineering pipeline. Thus, raw variables are outputted by the pipeline unless they are explicitly requested (by a data scientists developing a feature definition that applies the identity transformation to a raw variable). In alternative configurations, project maintainers can define a set of fixed feature definitions to include in the pipeline, which can include a set of important raw variables with the identity transformation or other transformations applied.

B.6 Feature execution engine

Ballet’s feature execution engine is responsible for applying the full feature engineering pipeline or an individual feature to extract feature values from a given set of data instances. Each feature function within the pipeline is passed the input columns it requires, which it then transforms appropriately, internally using one or more transformer steps (Figure 3). It operates as follows, starting from a set of `Feature` objects.

- (1) The transformer steps of each feature are postprocessed in a single step. First, any syntactic sugar is replaced with the appropriate objects. For example, an anonymous function is replaced by a `FunctionTransformer` object that applies the function, or a tuple of an input and another transformer is replaced by an `SubsetTransformer` object that applies the transformer on the given subset of the input and passes through the remaining columns unchanged. Second, the transformer steps are all wrapped in functionality that allows them to recover from any errors that are due to type incompatibility. For example, if the underlying transformation expects a 1-d array (column vector), but receives as input a 2-d array with a single column, the wrapper will catch the error, convert the 2-d array to a 1-d array, and retry the transformation. The wrapper pre-defines a set of these “conversion approaches” (one of which is the identity transformation), which will be tried in sequence until one is successful. The successful approach is stored so that it can be re-used during subsequent applications of the feature.
- (2) The features are composed together into a feature engineering pipeline object.
- (3) The fit stage of the feature engineering pipeline is executed. For each feature, the execution engine indexes out the declared input columns from the raw data and passes them to the wrapped `fit` method of the feature’s transformer. (This stage only occurs during training.)
- (4) The transform stage of the feature engineering pipeline is executed. For each feature, the execution engine indexes out the declared input columns from the raw data and passes them to the wrapped `transform` method of the feature’s transformer.

This process can also be parallelized across features. Since support for nested feature definitions (Appendix B.3) means that if features were executed independently there may be redundant computation if there were dependencies between features, this would necessitate a more careful approach in which the features are first sorted topologically and then resulting feature values are cached after first computation. For very large feature sets or datasets, full-featured dataflow engines should be considered.

B.7 Acceptance procedures for feature definitions

Contributions of feature engineering code, just like other code contributions, must be evaluated for quality before being accepted in order to mitigate the risk of introducing errors, malicious behavior, or design flaws. For example, a feature function that produces non-numeric values can result in an unusable feature engineering pipeline. Large feature engineering collaborations can also be susceptible to “feature spam,” a high volume of low-quality feature definitions (submitted either intentionally or unintentionally) that harm the collaboration [81]. Modeling performance can suffer and require an additional feature selection step — violating the working pipeline invariant — and the experience of other collaborators can be harmed if they are not able to assume that existing feature definitions are high-quality.

To address these possibilities, we extensively validate feature definition contributions for software quality and ML performance. Validation is implemented as a test suite that is both exposed by the Ballet client and executed in CI for every pull request. Thus, the same method that is used in CI for validating feature

IsFeatureCheck	HasCorrectInputTypeCheck	HasCorrectOutputDimensionsCheck
HasTransformerInterfaceCheck	CanFitCheck	CanFitOneRowCheck
CanTransformCheck	CanTransformNewRowsCheck	CanTransformOneRowCheck
CanFitTransformCheck	CanMakeMapperCheck	NoMissingValuesCheck
NoInfiniteValuesCheck	CanDeepcopyCheck	CanPickleCheck

Table 6. Feature API validation suite in (`ballet.validation.feature_api.checks`) that ensures the proper functioning of the shared feature engineering pipeline.

contributions is available to data scientists for debugging and performance evaluation in their development environment. Ballet Bot can automatically merge pull requests corresponding to accepted feature definitions and close pull requests corresponding to rejected feature definitions.

This automatic acceptance procedure is defined for the addition of new feature definitions only.

B.7.1 Feature API validation. User-contributed feature definitions should satisfy the `Feature` interface and successfully deal with common error situations, such as intermediate computations producing missing values. We fit the feature function to a separate subsampled training dataset in an isolated environment and extract feature values from subsampled training and validation datasets, failing immediately on any implementation errors. We then conduct a battery of 15 tests to increase confidence that the feature function would also extract acceptable feature values on unseen inputs (Table 6). Each test is paired with “advice” that can be surfaced back to the user to fix any issues (Figure 1).

Another part of feature API validation is an analysis of the changes introduced in a proposed PR to ensure that the required project structure is preserved and that the collaborator has not accidentally included irrelevant code that would need to be evaluated separately.¹⁴ A feature contribution is valid if it consists of the addition of a valid source file within the project’s `src/features/contrib` subdirectory that also follows a specified naming convention using the user’s login name and the given feature name. The introduced module must define exactly one object — an instance of `Feature` — which will then be imported by the framework.

B.7.2 ML performance validation. A complementary aspect of the acceptance procedure is validating a feature contribution in terms of its impact on machine learning performance, which we cast as a streaming feature definition selection (SFDS) problem. This is a variant of streaming feature selection where we select from among feature definitions rather than feature values. Features that improve ML performance will pass this step; otherwise, the contribution will be rejected. Not only does this discourage low-quality contributions, but it provides a way for collaborators to evaluate their performance, incentivizing more deliberate and creative feature engineering.

We first compile requirements for an SFDS algorithm to be deployed in our setting, including that the algorithm should be stateless, support real-world data types (mixed discrete and continuous), and be robust to over-submission. While there has been a wealth of research into streaming feature selection [93, 96, 101, 105], no existing algorithm satisfies all requirements. Instead, we extend prior work to apply to our situation [54, 56]. Our SFDS algorithm proceeds in two stages.¹⁵ In the *acceptance* stage, we compute the conditional mutual information of the new feature values with the target given the existing feature matrix and accept the feature if it is above a dynamic threshold. In the *pruning* stage, existing features that have been made newly redundant by accepted features can be pruned. Details are presented in the following sections.

B.8 Streaming feature definition selection

Feature selection is a classic problem in machine learning and statistics. The problem of feature selection is to select a subset of the available feature values such that a learning algorithm that is run on the subset generates a predictive model with the best performance according to some measure.

¹⁴This “project structure validation” is only relevant in CI and is not exposed by the Ballet client.

¹⁵Yes, we abbreviate the general problem of streaming feature definition selection as SFDS, and also call our algorithm to solve this problem SFDS. We trust that readers can disambiguate based on context.

DEFINITION 7. *The feature selection problem is to select a subset of feature values that maximizes some utility,*

$$X^* = \arg \max_{X' \in \mathcal{P}(X)} U(X'), \quad (1)$$

where $\mathcal{P}(A)$ denotes the power set of A . For example, U could simply measure the empirical risk of a model trained on X' .

If there exists a group structure in X , then this formulation ignores the group structure and allows feature values to be subselected from within groups. In some cases, this may not be desirable, such as if it is necessary to preserve the coherence and interpretability of each feature group. In the case of feature engineering using feature functions, it further conflicts with the understanding of each feature function as extracting a semantically related set of feature values.

Thus we instead consider the related problem of feature definition selection.

DEFINITION 8. *The feature definition selection problem is to select a subset of feature definitions that maximizes some utility,*

$$\mathcal{F}^* = \arg \max_{\mathcal{F}' \in \mathcal{P}(\mathcal{F})} U(\mathcal{F}'), \quad (2)$$

This constrains the feature selection problem to select either all of or none of the feature values extracted by a given feature.

In Ballet, as collaborators develop new features, each feature arrives at the project in a streaming fashion, at which point it must be accepted or rejected immediately. Streaming feature definition selection is a streaming extension of feature definition selection.

DEFINITION 9. *Let Γ be a feature stream of unknown size, let \mathcal{F} be the set of features accepted as of some time, and let $f \in \Gamma$ arrive next. The streaming feature definition selection problem is to select a subset of feature definitions that maximizes some utility,*

$$\mathcal{F}^* = \arg \max_{\mathcal{F}' \in \mathcal{P}(\mathcal{F} \cup f)} U(\mathcal{F}'). \quad (3)$$

Streaming feature definition selection consists of two decision problems, considered as sub-procedures. The *streaming feature definition acceptance* decision problem is to *accept* f , setting $\mathcal{F} \leftarrow \mathcal{F} \cup f$, or *reject*, leaving \mathcal{F} unchanged. The *streaming feature pruning* decision problem is to remove a subset $\mathcal{F}_0 \subset \mathcal{F}$ of low-quality features, setting $\mathcal{F} = \mathcal{F} \setminus \mathcal{F}_0$.

Design criteria. Streaming feature definition selection algorithms must be carefully designed to best support collaborations in Ballet. We consider the following design criteria, motivated by engineering challenges, security risks, and experience from system prototypes:

- (1) *Definitions, not values.* The algorithm should have first-class support for feature definitions (or feature groups) rather than selecting individual feature values.
- (2) *Stateless.* The algorithm should require as inputs only the current state of the Ballet project (i.e., the problem data and accepted features) and the pull request details (i.e., the proposed feature). Otherwise, each Ballet project (i.e., its GitHub repository) would require additional infrastructure to securely store the algorithm state.
- (3) *Robust to over-submission.* The algorithm should be robust to processing many more feature submissions than raw variables present in the data (i.e., $|\Gamma| \gg |\mathcal{V}|$). Otherwise malicious (or careless) contributors can automatically submit many features, unacceptably increasing the dimensionality of the resulting feature matrix.
- (4) *Support real-world data.* The algorithm should support mixed continuous- and discrete-valued features, common in real-world data.

Surprisingly, there is no existing algorithm that satisfies these design criteria. Algorithms for feature value selection might only support discrete data, algorithms for feature group selection might require persistent storage of decision parameters, etc. And the robustness criterion remains important given the results of

Smith et al. [81], in which users of a collaborative feature engineering system programmatically submitted thousands of irrelevant features, constraining modeling performance. These factors motivate us to create our own algorithm.

SFDS. Instead, we present a new algorithm, SFDS, for streaming feature definition selection based on mutual information criteria. It extends the GFSSF algorithm [56] both to support feature definitions rather than feature values and to support real-world tabular datasets with a mix of continuous and discrete variables.

The algorithm works as follows. In the acceptance stage (Figure 7), we first determine if a new feature f is *strongly relevant*; that is, whether the information $f(\mathcal{D})$ provides about Y above and beyond the information that is already provided by $\mathcal{F}(\mathcal{D})$ is above some threshold governed by hyperparameters λ_1 and λ_2 , which penalize the number of features and the number of feature values, respectively. If so, we accept it immediately. Otherwise, the feature may still be *weakly relevant*, in which case we consider whether f and some other feature $f' \in \mathcal{F}$ provide similar information about Y . If f is determined to be superior to such an f' , then f can be accepted. Later, in the pruning stage (Figure 8), f' and any other redundant features are pruned.

Algorithm 1: SFDS

```

input   : feature stream  $\Gamma$ , evaluation dataset  $\mathcal{D}$ 
output  : accepted feature set  $\mathcal{F}$ 

1  $\mathcal{F} \leftarrow \emptyset$ 
2 while  $\Gamma$  has new features do
3    $f \leftarrow$  get next feature from  $\Gamma$ 
4   if  $\text{accept}(\mathcal{F}, f, \mathcal{D})$  then
5      $\mathcal{F} \leftarrow \text{prune}(\mathcal{F}, f, \mathcal{D})$ 
6      $\mathcal{F} \leftarrow \mathcal{F} \cup f$ 
7   end
8 end
9 return  $\mathcal{F}$ 

```

Fig. 6. SFDS algorithm for streaming feature definition selection. It relies on two lower-level procedures, `accept` and `prune` to accept new feature definitions and to possibly prune newly redundant feature definitions.

Alternative validators. Maintainers of Ballet projects are free to configure alternative ML performance validation algorithms given the needs of their own projects. While we use SFDS for the *predict-census-income* project, Ballet provides implementations of the following alternative validators: `AlwaysAcceptor` (accept every feature definition), `MutualInformationAcceptor` (accept feature definitions where the mutual information of the extracted feature values with the prediction target is above a threshold), `VarianceThresholdAcceptor` (accept feature definitions where the variance of each feature value is above a threshold), and `CompoundAcceptor` (accept feature definitions based on the conjunction or disjunction of the results of multiple underlying validators). Additional validators can be easily created by defining a subclass of `ballet.validation.base.FeatureAcceptor` and/or `ballet.validation.base.FeaturePruner`.

Procedure $\text{accept}(\mathcal{F}, f, \mathcal{D})$

input : accepted feature set \mathcal{F} , proposed feature f , evaluation dataset \mathcal{D}
params : penalty on number of feature definitions λ_1 , penalty on number of feature values λ_2
output : accept/reject

```

1 if  $I(f(\mathcal{D}); Y|\mathcal{F}(\mathcal{D})) > \lambda_1 + \lambda_2 \times q(f)$  then
2   | return true
3 end
4 for  $f' \in \mathcal{F}$  do
5   |  $\mathcal{F}' \leftarrow \mathcal{F} \setminus f'$ 
6   | if  $I(f(\mathcal{D}); Y|\mathcal{F}'(\mathcal{D})) - I(f'(\mathcal{D}); Y|\mathcal{F}'(\mathcal{D})) > \lambda_1 + \lambda_2 \times (q(f) - q(f'))$  then
7   | | return true
8   | end
9 end
10 return false

```

Fig. 7. SFDS acceptance procedure for feature f .**Procedure** $\text{prune}(\mathcal{F}, f, \mathcal{D})$

input : previously accepted feature set \mathcal{F} , newly accepted feature f , evaluation dataset \mathcal{D}
params : penalty on number of feature definitions λ_1 , penalty on number of feature values λ_2
output : pruned feature set \mathcal{F}

```

1 for  $f' \in \mathcal{F}$  do
2   |  $\mathcal{F}' \leftarrow \mathcal{F} \setminus f' \cup f$ 
3   | if  $I(f'(\mathcal{D}); Y|\mathcal{F}'(\mathcal{D})) < \lambda_1 + \lambda_2 \times q(f')$  then
4   | |  $\mathcal{F} \leftarrow \mathcal{F} \setminus f'$ 
5   | end
6 end
7 return  $\mathcal{F}$ 

```

Fig. 8. SFDS pruning procedure for newly accepted feature f .