

ENABLING COLLABORATIVE DATA SCIENCE DEVELOPMENT WITH THE BALLET FRAMEWORK

Micah J. Smith¹ Jürgen Cito^{2,3} Kelvin Lu¹ Kalyan Veeramachaneni¹

ABSTRACT

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

1 INTRODUCTION

The open-source model for software development has led to successful, large-scale collaborations in building software frameworks, software systems, chess engines, scientific analyses, and more (Raymond, 1999; Linux; GNU; Stockfish; Bos et al., 2007). However, data science, and in particular, predictive machine learning (ML) modeling, has not 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, and when they are, they rarely have more than a handful of collaborators, orders of magnitude smaller (Table 1, Choi & Tausczik (2017)).

There is great potential impact of large-scale, collaborative data science to address societal problems through community-driven analysis of public datasets. For example, the Fragile Families Challenge tasked data scientists with predicting GPA and eviction for a set of disadvantaged children (Fragile Families Challenge), and *crash-model* is an application to predict car crashes and thereby direct safety interventions (Insight Lane). Such projects with complex and unwieldy datasets attract scores of interested contributors whose insight and intuition could be

significant if they were able to get involved.

To address these challenges, we introduce a novel ML programming model for collaborative, open-source data science. Drawing inspiration from successful collaboration paradigms in software engineering, our approach is based on decomposing the data science process into modular patches — standalone units of contribution — that can then be intelligently combined, representing objects like “feature definition” or “labeling function.” Prospective contributors work in parallel to write patches and submit them to an open-source repository. Our framework provides the underlying functionality to merge high-quality contributions and compose the accepted contributions into a single product. We instantiate these ideas in *Ballet*¹, a lightweight software framework for collaborative data science, and in a plugin to support feature engineering on tabular data. Projects built with Ballet are structured by these modular patches, yielding benefits beyond collaboration such as maintainability, reproducibility, and automated analysis. Together with *Assemblé*, a novel data science development environment targeting Ballet, even novice open-source developers can contribute to large collaborative projects.

Although there has been much research into understanding diverse challenges in data science development (Chatopadhyay et al., 2020; Yang et al., 2018; Subramanian et al., 2020; Sculley et al., 2015; Choi & Tausczik, 2017), there has been little attention given to understanding opportunities and challenges of large data science collaborations. Leveraging Ballet as a probe, we conduct an anal-

¹Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA ²Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA ³TU Wien, Vienna, Austria. Correspondence to: Micah J. Smith <micahs@mit.edu>.

¹<https://github.com/HDI-Project/ballet>

Table 1. The number of unique contributors to selected large open-source projects in either software engineering or data science. (As of October 2020.)

Software engineering	Data science
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+ JaidevAI/EasyOCR 62
rails/rails	4,300+ ageitgey/face_recognition 43
gatsbys/gatsby	3,600+ microsoft/CameraTraps 21
helm/charts	3,400+ Data4Democracy/drug-spending 21
rust-lang/rust	3,000+ infinitered/nsfwjs 19
kubernetes/kubernetes	2,900+ dssg/police-eis 19
nodejs/node	2,800+ IBM/MAX-Object-Detector 18

ysis of *predict-census-income*, a large real-world collaborative project 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 collaborators working together on this task, focusing on understanding the experience and performance of participants from varying backgrounds, the characteristics of collaborative feature engineering code, and the performance of the resulting model compared to alternative approaches. The resulting project is one of the largest data science collaborations GitHub, and outperforms state-of-the-art tabular AutoML systems and independent data science experts.

2 CONCEPTUAL FRAMEWORK

To develop our framework, we first synthesized a set of challenges of collaboration to address. For each challenge, we make comparisons to software development to build intuition and arrive at a solution. These challenges and corresponding solutions form the basis for the software framework we have implemented.

- C1 *Division of work.* Working alone, data scientists often write an end-to-end script that prepares the data, extracts features, builds and trains a model, and tunes hyperparameters. How can this work be divided so that all collaborators can contribute without redundancy?
- C2 *Data science workflows.* Data scientists are accustomed to working in computational notebooks and have varying expertise with version control tools like git. How can workflows be adapted to use a shared codebase and build a single product?
- C3 *Evaluating contributions.* Suppose a project maintainer has divided up tasks and adopted a workflow to use a shared codebase. As prospective collaborators begin submitting code to the shared codebase, how can their contributions be evaluated? What if a contribution introduces errors to the codebase or decreases the performance of the model?

Table 2. Applying collaboration concepts to software development and feature engineering.

concept	software development	feature engineering
<i>patch</i>	bugfix, software feature	feature definition
<i>product</i>	application, library	feature engineering pipeline
<i>acceptance procedure</i>	unit test, integration test	feature test, streaming feature definition selection

- C4 *Data and computation.* Data science requires carefully managing data and computation. Will it be necessary to establish shared data stores and computing infrastructure? Would this be expensive and require significant technical and DevOps expertise?

Researchers and practitioners have developed many successful methodologies for software engineering. We extract a conceptual framework from these for collaboration. An overview of these concepts is shown in Table 2.

- S1 *Data science patches.* We identify steps of the data science process that can be broken down into many *patches* — modular source code units — that can be developed and contributed separately in an incremental process. In software engineering, this manifests as standalone bugfixes that are contributed or new software features that are implemented.
- S2 *Open-source workflow.* The composition of many patches from different contributors forms a *product* which is stored in an open-source repository. We enforce a git/GitHub-based workflow in which patches are proposed as individual Pull Requests (PRs). This enables a different-time, different-place collaboration style (Shneiderman et al., 2016). We design this process to accommodate contributors of all backgrounds by providing multiple development interfaces, such as supporting and augmenting notebook-based workflows.
- S3 *Acceptance procedure.* Not every contribution is worthy of inclusion in the completed product, so high-quality and low-quality contributions must be separated through some *acceptance procedure*. Contributors receive feedback on the quality of their work from software quality and ML performance points of view.
- S4 *Lightweight framework.* We propose a lightweight framework to 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 accepting patches.

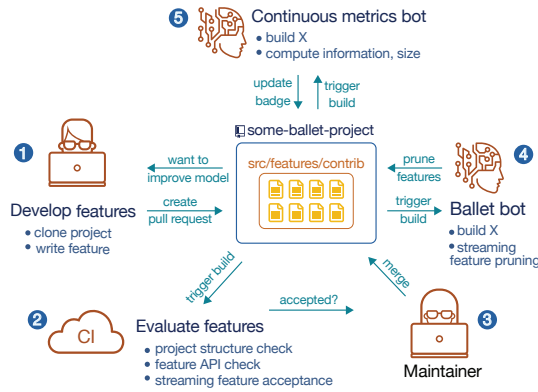


Figure 1. The development lifecycle of a Ballet project, from introduction of a patch to impact on the model, shown here for the case of feature engineering.

Any data science process in which human intuition and expertise is key to generating small patches that can be combined into a single product, and where software and statistical measures can be defined to measure quality in an acceptance procedure, can naturally fit into our collaborative framework. This naturally applies to several aspects of data science development, foremost of which is feature engineering, which is the subject of Sections 4 to 6. However, it can also be applied to data programming with labeling functions and slicing functions (Ratner et al., 2016; Chen et al., 2019), prediction engineering with prediction task definitions (Kanter et al., 2016), and various stacking and ensembling methods.

3 AN OVERVIEW OF BALLET

We have implemented these ideas in Ballet, a general-purpose software framework for collaborative, open-source data science. We illustrate how Ballet works by considering three roles/perspectives: maintainer, contributor, and consumer. This development lifecycle is illustrated in Figure 1. In Section 4, we will make this more concrete for the case of feature engineering on tabular datasets.

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 the ballet CLI, next using the CLI to render a new repository from the provided project template, which contains the minimal files and structure required for their project, such as directory structures, configuration files, problem metadata, etc., while the heavy lifting is delegated to Ballet. The resulting repository contains a usable (if, at first, empty) data science pipeline. They next define a task for contributors: create and submit a data science patch that performs well, for example a feature definition that has high predictive power. After pushing to GitHub and enabling our CI tools and bots, the

maintainer begins recruiting collaborators.

Collaborators working in parallel submit patches as PRs with a careful structure provided by Ballet. Not every patch is worthy of inclusion in the product. As patches arrive from collaborators as PRs, the CI service is repurposed to run Ballet’s acceptance procedure such that only high-quality patches are accepted. This “working pipeline invariant” aligns data science pipelines with the aim of *continuous delivery* in software development. In feature engineering, the acceptance procedure is a feature validation suite (Section 4.4) which marks individual features as accepted/rejected, and the resulting feature engineering pipeline on the trunk can always be executed to engineer feature values from a new raw data.

One challenge for maintainers is to act on data science patches as they begin to stream in. Unlike software projects where contributions can take any form, these type of patches are all structured similarly, and if they validate successfully, they may be merged without further review. To support maintainers, the *Ballet bot* can automatically manage contributions, such as merging PRs of accepted patches and closing rejected ones. The bot,² a GitHub application written in Javascript using the Probot framework, runs freely on open-source platforms. The process continues until performance of the ML product exceeds some threshold or improvements are exhausted.

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

Contributor. A data scientist 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, using the Ballet interactive client which supports them in loading data, exploring and analyzing existing patches, and validating the performance of their own work. When they are satisfied with its performance, they propose to add it to the upstream project.

We enable several interfaces for collaborators to develop and submit features, where different interfaces are appropriate for collaborators with different development expertise, relaxing requirements on the development style of collaborators. Experienced open-source contributors can use their preferred tools and workflow to submit their patch as

²<https://github.com/HDI-Project/ballet-bot>

a PR - the project is still just a layer built on top of familiar technologies like git. However, in pilot testing, we found that adapting from usual data science workflows was a huge obstacle. Many data scientists we worked with had never successfully contributed to any open-source project.

We addressed this by centering all development in the notebook with *Assemblé*, a cloud-based workflow for contribution to Ballet projects. We created a custom experience on top of open-source tooling that enables data scientists to complete the entire task in the notebook only. *Assemblé* consists of three pieces. First, we design and implement a custom Jupyter Lab extension³ for submitting code to a Ballet project. This frontend extension adds a simple one-click “Submit” button to the Notebook panel. Data scientists develop a patch within a messy, exploratory notebook. They isolate the patch, such as a single feature definition, in a code cell and click to submit. Second, we implement a Jupyter Server extension which preprocesses the submission in the context of the running Jupyter application. The code they have selected is subjected to initial server-side validation using static analysis and the patch is then automatically formulated as a PR on the contributors behalf, automating low-level version control details like forking the upstream project and committing the patch at the appropriate location in the project. The contributor can view their new PR in a matter of seconds. Third, we tightly integrate these extensions to be deployed on Binder,⁴ a community service for cloud-hosted notebooks. *Assemblé* can be launched from every Ballet project from a badge in the project’s README and contributors can accordingly submit a PR without using any other tools.

The submitted features are marked as accepted or rejected, and contributors can proceed accordingly, either moving on to their next idea, or reviewing diagnostic information and trying to fix errors or further improve a rejected feature.

Consumer. Ballet project consumers are now free to use the model for whatever end they desire, for example by executing the feature engineering pipeline to extract features from new raw data instances, or 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.

For example, the Ballet client for feature engineering projects exposes an `engineer.features` method that fits the feature engineering pipeline on training data and can then engineer features from new data instances. This can be eas-

³<https://github.com/HDI-Project/ballet-assemble>

⁴<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=input,
    ↪ transformer=transformer, name=name)
```

Listing 1. An example of a user-submitted feature definition in Ballet that conditionally unskews the “lot area” variable (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. Features are collected from standalone modules like this one and composed into a single feature engineering pipeline.

ily used on other library code.

4 COLLABORATIVE FEATURE ENGINEERING

We now describe in detail the design and implementation of a feature engineering plugin for Ballet for collaborative feature engineering projects. A plugin must define the concepts of *patch*, *product*, and *acceptance procedure*.

Feature engineering is the process of writing code to transform raw variables into feature values that can be used as input to an ML model, which we consider here to additionally encompass data understanding, cleaning, and preparation steps. 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 standalone *patch*, calling these *feature definitions* and representing them with a Python `Feature` interface. An example feature is shown in Listing 1. Features are composed into a feature engineering pipeline *product*. Newly contributed features are accepted if they pass a two-stage validation procedure which tests the feature API and its contribution to performance of the ML pipeline. Finally, the plugin specifies a module organization underneath `src/features/contrib` by username and feature-name that allows features to be collected programmatically.

4.1 Feature definitions

We consider a *feature definition*, or simply *feature*, the code that is used to extract semantically related feature values from raw data. In Ballet, each feature is placed in a separate Python module.

In the supervised learning setting, we observe data $\mathcal{D} = (\mathbf{x}_i, \mathbf{y}_i)_{i=1}^n$, where $\mathbf{x}_i \in \mathcal{X}$ are the raw variables and

$y_i \in \mathcal{Y}$ is the target in one data instance.

Definition 1. A feature is a learned map from raw variables in one data instance to feature values, $f : (\mathcal{X}, \mathcal{Y}) \rightarrow \mathcal{X} \rightarrow \mathbb{R}^{q_f}$, where q_f is the dimensionality of the feature values extracted by f .

Each feature learns a specific map from \mathcal{D} , such that any information it uses, such as variable means and variances, is learned from the 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 tuple of input and transformer. The input declares the variable(s) from \mathcal{X} that are needed by the feature, which will be passed to transformer, a sequence of one or more objects that each provide a fit/transform interface to learn parameters and then transform variables into feature values (Buitinck et al., 2013). Additional metadata like name, description, etc. are also exposed. The role of a feature engineering contributor is then simply to provide values for the input and transformer of a Feature object in their code.

A complete feature definition written using Ballet for a house price prediction problem is shown in Listing 1.

4.2 Feature engineering pipelines

Features are then composed together in a feature engineering pipeline.

Definition 2. Let f_1, \dots, f_m be a collection of features, \mathcal{D} be a training dataset, and $\mathcal{D}I$ be a collection of new data instances. A feature engineering pipeline \mathcal{F} applies each feature to the new data instances and concatenates the result, yielding the feature matrix⁵

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

This is implemented in Ballet by the FeatureEngineeringPipeline class which operates directly on dataframes. Each feature within the pipeline is passed the input columns it requires which it then transforms appropriately, internally using a sequence of one or more transformers (Figure 2).

4.3 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 learning component, others are more involved to express in a fit/transform style. Commonly, data scientists ex-

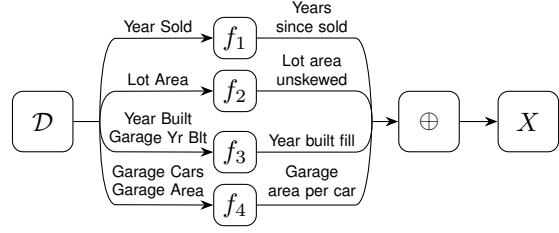


Figure 2. A feature engineering pipeline for a house price prediction problem with four features operating on six raw variables.

tract these more advanced features by manipulating training and test tables directly using popular libraries like *pandas* or *dplyr*, often leading to leakage. In pilot testing, 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*, *ballet.eng*, which implements many common learned transformations and utilities. These include *ConditionalTransformer*, which applies a secondary transformation depending on whether a condition is satisfied on the training data, and *GroupwiseTransformer*, which separately learns a transformer for each group of a group-by aggregation on the development set. In addition, we re-export 71 primitives from six popular libraries, such as *sklearn.preprocessing.SimpleImputer*.

4.4 Validating feature contributions

Code contributions to any software project must be thoroughly evaluated for quality before being accepted, at the risk of introducing errors, malicious behavior, or design flaws. Feature engineering code is no different. For example, a feature that produces non-numeric values can result in an unusable feature engineering pipeline. Large feature engineering collaborations can also be susceptible to “feature spam,” thousands of low-quality features (submitted either intentionally or accidentally) that harm the collaboration (Smith et al., 2017). 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 features are high-quality.

To address these possibilities, we extensively validate feature contributions for software quality and ML performance, implemented as a test suite that is exposed by the Ballet client library and executed in CI for every PR. The Ballet bot can automatically merge PRs corresponding to accepted features and close PRs corresponding to rejected features.

⁵We will usually drop the superscript \mathcal{D} without ambiguity.

Feature API validation. User-contributed feature should satisfy the Feature interface and successfully deal with common error situations, such as intermediate computations producing missing values. We fit the feature 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 would also produce acceptable feature values on unseen inputs, including `NoMissingValuesCheck`, `CanTransformNewRowsCheck`, and `CanPickleCheck`.

Streaming feature selection. Complementary to the software interface aspects of the feature 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 are determined as improving machine learning 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 contributors to evaluate their own 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, and be robust to over-submission. While there is a wealth of research into streaming feature selection (Zhou et al., 2005; Wu et al., 2013; Wang et al., 2015; Yu et al., 2016), no existing algorithm satisfies all requirements. Instead, we extend the GFSSF algorithm (Li et al., 2013) for our situation. Our algorithm proceeds in two stages. 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. Full details are presented in Appendix A.

The Ballet feature engineering plugin exposes this functionality as a `validate_feature_acceptance` method. The same method is used in CI for validating feature contributions as is available to data scientists for debugging and performance evaluation in their development environment.

5 CASE STUDY DESIGN

To better understand the characteristics of live collaborative data science projects, we use a mixed-method software engineering case study approach (Runeson & Höst, 2009). 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 research questions:

- RQ1** What are the most important aspects of a collaborative framework to 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** What is the performance of the collaborative model in comparison to other approaches?

General procedures. We created an open-source project using Ballet, `predict-census-income`,⁶ 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 (independent variables of our study). Next, they were directed to the public GitHub repository containing the collaborative project and asked to complete the task described in the project README. They were instructed to use either their own preferred development environment or Assemblé (Section 3). After task completion, we surveyed them about their experience.

Participants. In recruiting participants, we wanted to ensure that all participant backgrounds were represented: beginners and experts in data science, software development, and survey data analysis (the problem domain). We compiled personal contacts with various backgrounds. After reaching these contacts, we then used snowball sampling to recruit more participants with similar backgrounds.

Dataset. The input data is the raw survey responses to the 2018 US Census American Community Survey (ACS) for Massachusetts (Table 3). This “Public Use Microdata Sample” (PUMS) has anonymized individual-level responses. Unlike the classic ML “census” dataset (Kohavi, 1996) which is highly preprocessed, the raw ACS responses is a realistic form for a dataset used in an open data science project. Following the “census” dataset, 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

⁶<https://github.com/HDI-Project/ballet-predict-census-income>

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

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

typical week greater than 0. 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.

Data collection. 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 experience 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.
- *Binder telemetry.* To better understand the experience of participants who use Assemblé, we installed an instrumented version of the Ballet library on Binder 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, and if so, we recorded the buffered telemetry events.
- *Post-participation survey.* Participants who completed or attempted the task were asked to complete a survey about their experience, including the development environment they used, time spent by sub-task, activities/functionality they did/used as part of the task and which were most important, and open-ended feedback on different aspects. They were also asked about how demanding the task was using the NASA-TLX Task Load Index (Hart & Staveland, 1988), a workload assessment that is widely used in usability evaluations in software engineering and other domains (Cook et al., 2005; Salman & Turhan, 2018). Participants indicate on a scale the temporal demand, mental demand, and effort required by the task, their own 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 to the upstream predict-census-income project, we analyze both the submitted source code as well as performance char-

acteristics of the submitted feature.

- *Expert and AutoML baselines.* As a comparison to the Ballet collaboration, we also obtain baseline solutions to the personal income prediction problem, from two external data science experts working independently, and from a cloud provider’s AutoML service.

Analysis. After linking our data sources together, we perform 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 employ open and axial coding methodology to categorize the free text responses and relate codes to each other to form emergent themes (Böhm, 2004). Two researchers first coded each response independently, in which 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 finally revisited all responses and codes and how they relate to each other that lead 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 perform lightweight program analysis to extract and quantify the feature engineering primitives used by our participants.

6 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 continents completed the task in its entirety. During the case study, 28 features were merged which together extract 32 feature columns from the raw data. Of case study participants, 26 submitted at least one feature and 22 had at least one feature merged. Based on participants’ qualitative feedback on their experience, several key *themes* emerged from our qualitative analysis which we discuss inline.

6.1 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 design decisions of Ballet.

Goal Clarity. The project-level goal is clear – to produce a predictive model. In the case of survey data requiring feature engineering, the approach taken by Ballet is to de-

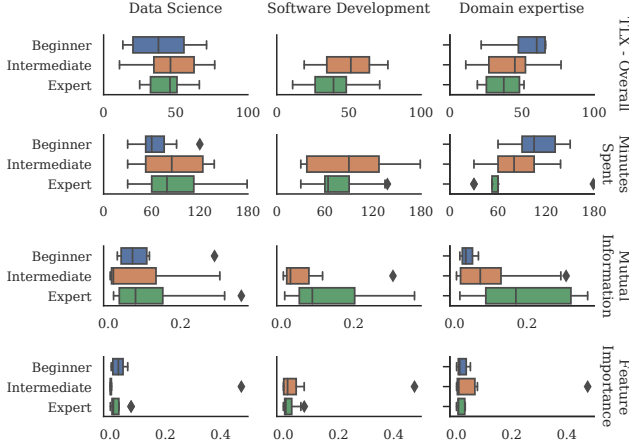


Figure 3. Task demand, total minutes spent on task, mutual information of best feature with target on test set, and total global feature importance assigned by AutoML service on development set, for participants of varying experience by type of background.

compose this into contributor-level goals, asking contributors to create and submit a patch that introduces a feature that performs well. Success in the contributor-level task is validated using statistical tests (Section 4.4). However, the relationship between the contributor-level and project-level goals may not appear aligned for all participants. This negatively impacted some participants’ experience by introducing confusion about the direction and goal of their task. Some concerns were with specific documentation elements, but others indicate 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). While a feature that maximizes some statistical measure would be 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 the specific contributor-level goal incentivizes high-quality feature engineering, participants are less focused on the project-level goal and may not consider implementing new project functionality, like functions to visualize and understand the raw data. This is a classic tension in designing collaborative mechanisms in terms of appropriately structuring goals and incentives (Ouchi, 1979).

Learning by Example. We asked participants to rank the most important functionality for completing the task in terms of both creating and submitting features (Figure 4). For the former, participants ranked most highly the ability to refer to example code written by project maintainers or by fellow participants. This form of implicit collaboration was useful for participants to accelerate the onboarding process, learn new feature engineering techniques, and coordinate effort.

Distribution of Work. However, this led to feedback about

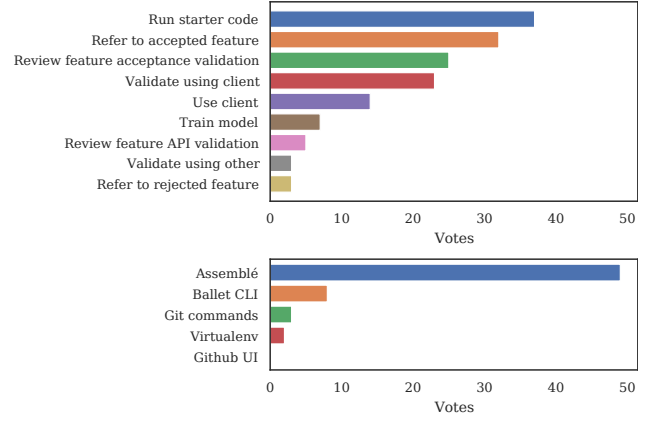


Figure 4. Most important functionality for collaborative feature engineering task, according to participant votes, for creating features (top) and submitting features (bottom).

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 features*” (P1). Some users specifically asked for methods to review the input columns that had and had not been used, and to limit the number of columns that one person would need to consider. Others, however were satisfied with a more passive approach in which they used Ballet’s interactive client to programmatically explored existing features. This is a promising direction of future work, in the same vein as automatic code reviewer recommendation (Peng et al., 2018).

Cloud-Based Workflow. In terms of submitting features, the most popular element by far was Assemblé. Importantly, out of 9 participants who said that they “never” contribute to open-source software, all of them successfully submitted a PR to the predict-census-income project, and all of them used Assemblé, of whom 7 used it the cloud and the others used it locally. Attracting these participants, most of whom are experienced data scientists, is critical to sustaining large collaborations, and prioritizing interfaces that provide first-class support for collaboration can support these contributors.

6.2 Participant experience

We look at six dimensions of participants’ backgrounds, though many are complementary. Our main dependent variables for participant experience are the overall cognitive load (TLX - Overall) and Minutes Spent. Our main dependent variables for participant performance are a measure of the ML performance of each feature, its mutual information with the target. We summarize the relation between background, experience, and performance measures

in Figure 3.

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 experience the most demand from the task were those with little experience analyzing survey data or contributing to open-source projects.

Experts find the task less demanding but perform similarly. We find that broadly, increased expertise in any of the background areas leads to perceiving the task as less demanding. However, data science and feature engineering experts actually spent more time working on the task than beginners did. They were not necessarily using this time to fix errors in their features, as they invoked the Ballet client’s validation functions fewer times. They may have been spending more time on learning about the project and data without writing code; then they may have used their own 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 about Ballet is rejected both for data science (Mann-Whitney $U=85.0$, Δ medians -6.0 min) and for software development (Mann-Whitney $U=103.0$, Δ medians -1.5 min).

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 is that a main issue qualitatively for contributing to the predict-census-income project is about *dataset challenges*, in which participants become overwhelmed or confused due to the wide and dirty raw survey data: “*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; a median of 36 minutes learning about the prediction problem and data vs 13.6 minutes for intermediate and expert participants (Mann-Whitney $U=36.5$, $p=0.064$, $n_1=6$, $n_2=21$).

6.3 Collaborative feature engineering code

We were interested in understanding the kind of feature engineering code participants write in this collaborative set-

ting. Participants in the case study contributed 28 features to the project, which together extracted 32 feature columns. The features had 47 transformers, with most features applying a single transformer to its input but some applying at most 4 transformers sequentially.

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

Feature characteristics. These features consumed 137 distinct columns of the raw survey responses, out of a total of 494 present in the entities table. Most of these columns were consumed by just one feature, but several were transformed in different ways, such as `SCHL` (education attainment) which was an input to 5 different features. Thus 357 columns, or 72%, were ignored by the collaborative project. Some of this is due to columns that are not predictive of personal income. For example, the end-to-end AutoML model that operates directly on the raw survey responses assigns feature importance of less than 0.001 to 418 columns (where the feature importances sum to 1). However, there may still be a missed opportunity by the collaborators, as the AutoML model assigns feature importance of greater than 0.01 to 7 columns that were unused by any of the participants’ features, such as `RELP` which indicates the person’s relationship to the “reference person” in the household and intuitively is predictive of income, as it allows the modeler to differentiate between adults who are dependents of their parents for example. This suggests an opportunity for developers of collaborative frameworks like Ballet to provide more formal direction on where to invest feature engineering effort, for example by providing methods to summarize the inputs that have or have not been included in patches. This is also in line with the theme on *distribution of work* that emerged from participants’ responses. Of the features, 11/28 had a learned transformer whereas 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 to 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.

6.4 Comparative Performance

While we focus on better understanding how data scientists work together in a collaborative setting, ultimately a collaborative model must be able to achieve good performance in order to be a useful paradigm. To evaluate this, we compare the performance of the feature engineering pipeline built by the case study participants against several alternatives. First, we ask two independent data science experts to solve the combined feature engineering and modeling task, without knowledge of the collaborative project. Second, we use Google Cloud AutoML Tables⁷, which supports structured data “as found in the wild”, to automatically solve the task, which we run with its default settings until convergence.

We find that the best ML performance among the alternatives we evaluated is using the Ballet feature engineering pipeline and passing the extracted feature values to AutoML Tables (Table 4). This hybrid approach outperforms both of the experts as well as using AutoML Tables end-to-end. It also confirms previous results which find both that feature engineering is difficult to automate and that advances in AutoML have led to expert- or super-expert performance on clean, well-defined inputs. **Qualitative differences.** The three approaches to the task varied widely. Ballet participants spent all of their engineering 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 features, create one-hot encoding and embeddings for categorical features) but spends most of the runtime budget searching and tuning models, resulting in a gradient-boosted decision tree for the census problem. The experts similarly performed minimal feature engineering (encoding and imputing); the resulting models were (1) a minority class oversampling step followed by a tuned AdaBoost classifier and (2) a custom greedy forward feature selection step followed by a linear probability model.

7 LIMITATIONS

There are several limitations of our approach. Feature engineering is a complex process and we have not yet provided support for several common practices (or potential new practices). For example, many features are trivial to specify and can be enumerated by automated approaches

Feature Engineering	Modeling	Accuracy	Precision	Recall	F1	Failure rate
Ballet	AutoML	0.876	0.838	0.830	0.834	0
AutoML	AutoML	0.462	0.440	0.423	0.431	0.475
Ballet	Expert 1	0.828	0.799	0.707	0.734	0
Ballet	Expert 2	0.840	0.793	0.858	0.811	0
Expert 1	Expert 1	0.814	0.775	0.686	0.710	0
Expert 2	Expert 2	0.857	0.809	0.867	0.828	0

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 test rows. But when using the features produced by the Ballet collaboration, the AutoML method outperforms human experts.

(Kanter & Veeramachaneni, 2015), and some data cleaning and preparation can be performed automatically. We have deferred 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 columns or that operate on existing feature values rather than raw variables, could enhance contributor productivity.

8 RELATED WORK

There has been much interest in facilitating increased collaboration in machine learning. Though most existing work does not provide a structured way to ensure an effective division of work, one exception is Smith et al. (2017), which introduces a collaborative feature engineering platform where contributors log in and submit source code directly to a machine learning backend server. While Ballet shares several ideas, it targets a lightweight approach suitable for open-source and focuses on modularity and supporting contributor workflows.

Unskilled crowd workers can be harnessed for feature engineering tasks, such as by labeling data to provide the basis for further manual feature engineering (Cheng & Bernstein, 2015). Synchronous editing interfaces, like that of (Garg et al., 2018; Google Colaboratory; Kluyver et al., 2016), could facilitate multiple users to edit a machine learning model specification. A form of collaboration can also be achieved in data science competitions (Bennett & Lanning, 2007; KDD Cup) and using networked science hubs (Vanschoren et al., 2013). While these have led to state-of-the-art modeling performance, there is no natural way for competitors to integrate source code components in a systematic way and it is not appropriate for open-source problems.

While we focus on testing individual features, research has focused on other types of ML testing. Renggli et al. (2019) investigate practical and statistical considerations arising from testing model accuracy in a continuous integration setting. Specific models and algorithms can be

⁷<https://cloud.google.com/automl-tables/>

tested (Grosse & Duvenaud, 2014) and input data can be validated directly (Hynes et al., 2017; Breck et al., 2019). Testing can also be tied to reproducibility in ML research (Ross & Forde, 2018).

The open-source model for developing software has been adopted and advanced by many individuals and projects (Raymond, 1999). Recent research has visited the use of modern development tools like continuous integration (Vasilescu et al., 2015) and crowdsourcing (Latoza & Hoek, 2016). Simard et al. (2017) advocate for an approach to developing ML models based on principles of software engineering and programming languages.

9 CONCLUSION

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

ACKNOWLEDGMENT

We'd like to thank the following people: the participants of the ballet-predict-census-income case study; Zhuofan Xie, Fahad Alhasoun.

REFERENCES

- Bennett, J. and Lanning, S. The netflix prize. In *Proceedings of KDD Cup and Workshop 2007*, pp. 1–4, 2007.
- Böhm, A. Theoretical coding: Text analysis in. *A companion to qualitative research*, 1, 2004.
- Bos, N., Zimmerman, A., Olson, J., Yew, J., Yerkie, J., Dahl, E., and Olson, G. From shared databases to communities of practice: A taxonomy of collaboratories. *Journal of Computer-Mediated Communication*, 12(2): 318–338, 2007.
- Breck, E., Polyzotis, N., Roy, S., Whang, S. E., and Zinkevich, M. Data Validation for Machine Learning. In *Proceedings of the 2nd SysML Conference*, pp. 1–14, 2019.
- Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., and Varoquaux, G. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.
- Chattopadhyay, S., Prasad, I., Henley, A. Z., Sarma, A., and Barik, T. 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*, pp. 1–12. ACM, Apr 2020. ISBN 978-1-4503-6708-0. doi: 10.1145/3313831.3376729. URL <https://dl.acm.org/doi/10.1145/3313831.3376729>.
- Chen, V., Wu, S., Ratner, A. J., Weng, J., and Ré, C. Slice-based learning: A programming model for residual learning in critical data slices. In *33rd Conference on Neural Information Processing Systems*, pp. 11, 2019.
- Cheng, J. and Bernstein, M. S. Flock: Hybrid Crowd-Machine Learning Classifiers. *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW ’15*, pp. 600–611, 2015.
- Choi, J. and Tausczik, Y. Characteristics of collaboration in the emerging practice of open data analysis. pp. 835–846. ACM Press, 2017. ISBN 978-1-4503-4335-0. doi: 10.1145/2998181.2998265. URL <http://dl.acm.org/citation.cfm?doid=2998181.2998265>.
- Cook, C., Irwin, W., and Churcher, N. A user evaluation of synchronous collaborative software engineering tools. In *12th Asia-Pacific Software Engineering Conference (APSEC’05)*, pp. 6 pp.–, Dec 2005. doi: 10.1109/APSEC.2005.22.
- De Cock, D. Ames, iowa: Alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, 19(3), 2011.
- Epidemic Prediction Initiative. Dengue forecasting project. URL <https://web.archive.org/web/20190916180225/https://predict.phiresearchlab.org/post/5a4fcc3e2c1b1669c22aa261>. Accessed 2018-04-30.
- Fragile Families Challenge. Fragile families challenge, 2017. URL <http://www.fragilefamilieschallenge.org/>.
- Garg, U., Prabhu, V., Yadav, D., Ramrakhya, R., Agrawal, H., and Batra, D. Fabrik: An online collaborative neural network editor. *arXiv e-prints*, art. arXiv:1810.11649, 2018.
- GNU. The gnu operating system. URL <https://www.gnu.org>.
- Google Colaboratory. Google colaboratory. URL <https://www.colab.research.google.com>.
- Grosse, R. B. and Duvenaud, D. K. Testing MCMC code. In *2014 NIPS Workshop on Software Engineering for Machine Learning*, pp. 1–8, 2014.
- Hart, S. G. and Staveland, L. E. *Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research*, volume 52, pp. 139–183. Elsevier, 1988. ISBN 978-0-444-70388-0. doi: 10.1016/s0166-4115(08)62386-9. URL <https://linkinghub.elsevier.com/retrieve/pii/S0166411508623869>.
- Hynes, N., Sculley, D., and Terry, M. The Data Linter: Lightweight, Automated Sanity Checking for ML Data Sets. *Workshop on ML Systems at NIPS 2017*, 2017.
- Insight Lane. Crash model, 2019. URL <https://github.com/insight-lane/crash-model>.
- Kanter, J. M. and Veeramachaneni, K. Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 1–10, 2015.
- Kanter, J. M., Gillespie, O., and Veeramachaneni, K. Label, segment, featurize: A cross domain framework for prediction engineering. *Proceedings - 3rd IEEE International Conference on Data Science and Advanced Analytics, DSAA 2016*, pp. 430–439, 2016.
- KDD Cup. Kdd cup. URL <http://kdd.org/kdd-cup>.

- Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., and Willing, C. Jupyter notebooks – a publishing format for reproducible computational workflows. In Loizides, F. and Schmidt, B. (eds.), *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pp. 87 – 90. IOS Press, 2016.
- Kohavi, R. Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid. In *KDD*, pp. 6, 1996.
- Kraskov, A., Stögbauer, H., and Grassberger, P. Estimating mutual information. *Physical Review E - Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics*, 69(6):16, 2004.
- Latoza, T. D. and Hoek, A. V. D. Crowdsourcing in Software Engineering: Models, Opportunities, and Challenges. *IEEE Software*, pp. 1–13, 2016.
- Li, H., Wu, X., Li, Z., and Ding, W. Group feature selection with streaming features. *Proceedings - IEEE International Conference on Data Mining, ICDM*, pp. 1109–1114, 2013.
- Linux. The linux kernel organization. URL <https://www.kernel.org>.
- Ouchi, W. G. A conceptual framework for the design of organizational control mechanisms. *Management science*, 25(9):833–848, 1979.
- Peng, Z., Yoo, J., Xia, M., Kim, S., and Ma, X. Exploring how software developers work with mention bot in github. In *Proceedings of the Sixth International Symposium of Chinese CHI on - ChineseCHI '18*, pp. 152–155. ACM Press, 2018. ISBN 978-1-4503-6508-6. doi: 10.1145/3202667.3202694. URL <http://dl.acm.org/citation.cfm?doid=3202667.3202694>.
- Ratner, A., Sa, C. D., Wu, S., Selsam, D., and Ré, C. Data programming: Creating large training sets, quickly. *Advances in neural information processing systems*, 29: 3567–3575, 2016.
- Raymond, E. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.
- Renggli, C., Karlaš, B., Ding, B., Liu, F., Schawinski, K., Wu, W., and Zhang, C. Continuous Integration of Machine Learning Models With ease.ml/ci: Towards a Rigorous Yet Practical Treatment. In *Proceedings of the 2nd SysML Conference*, pp. 1–12, 2019.
- Ross, A. S. and Forde, J. Z. Refactoring Machine Learning. In *Workshop on Critiquing and Correcting Trends in Machine Learning at NeurIPS 2018*, pp. 1–6, 2018.
- Runeson, P. and Höst, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, Apr 2009. ISSN 1382-3256, 1573-7616. doi: 10.1007/s10664-008-9102-8.
- Salman, I. and Turhan, B. 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*, pp. 130–139. ACM Press, 2018. ISBN 978-1-4503-6459-1. doi: 10.1145/3202710.3203148. URL <http://dl.acm.org/citation.cfm?doid=3202710.3203148>.
- Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. Hidden technical debt in machine learning systems. *Advances in Neural Information Processing Systems*, pp. 2494–2502, 2015.
- Shneiderman, B., Plaisant, C., Cohen, M., Jacobs, S., Elmqvist, N., and Diakopoulos, N. *Designing the user interface: strategies for effective human-computer interaction*. Pearson, 2016.
- Simard, P., Amershi, S., Chickering, M., Edelman Pelton, A., Ghorashi, S., Meek, C., Ramos, G., Suh, J., Verwey, J., Wang, M., and Wernsing, J. Machine teaching: A new paradigm for building machine learning systems. Technical Report MSR-TR-2017-26, July 2017.
- Smith, M. J., Wedge, R., and Veeramachaneni, K. Featurehub: Towards collaborative data science. In *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 590–600, Oct 2017.
- Stockfish. Stockfish: A strong open source chess engine. URL <https://stockfishchess.org>. Retrieved 2019-09-05.
- Subramanian, K., Hamdan, N., and Borchers, J. Casual notebooks and rigid scripts: Understanding data science programming. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–5, Aug 2020. doi: 10.1109/VL/HCC50065.2020.9127207.
- Vanschoren, J., van Rijn, J. N., Bischl, B., and Torgo, L. Openml: networked science in machine learning. *SIGKDD Explorations*, 15:49–60, 2013.
- Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., and Filkov, V. 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*, pp. 805–816, 2015.

- Wang, J., Wang, M., Li, P., Liu, L., Zhao, Z., Hu, X., and Wu, X. Online Feature Selection with Group Structure Analysis. *IEEE Transactions on Knowledge and Data Engineering*, 27(11):3029–3041, 2015.
- Wu, X., Yu, K., Ding, W., Wang, H., and Zhu, X. Online feature selection with streaming features. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(5):1178–1192, 2013.
- Yang, Q., Suh, J., Chen, N.-C., and Ramos, G. 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*, pp. 573–584, 2018. doi: 10.1145/3196709.3196729.
- Yu, K., Wu, X., Ding, W., and Pei, J. Scalable and accurate online feature selection for big data. *TKDD*, 11:16:1–16:39, 2016.
- Zhou, J., Foster, D., Stine, R., and Ungar, L. Streaming feature selection using alpha-investing. *Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining - KDD '05*, pp. 384, 2005.

A STREAMING FEATURE DEFINITION SELECTION

We start from feature selection and then consider a streaming extension.

Definition 3. *The feature definition selection problem is to select a subset of feature definitions that maximizes some utility,*

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

where $\mathcal{P}(A)$ denotes the power set of A .

For example, U could simply measure the empirical risk of a learner trained on the extracted feature values, or it could also include a measurement of model interpretability.

In contrast, the traditional feature selection problem is to select a subset of the extracted feature *values* $X^* \subseteq X$. This may not be directly possible (or desirable) in the feature engineering setting as to preserve the coherence and interpretability of each feature definition.

In Ballet, as collaborators develop new features, each feature arrives to the project in a streaming fashion in a feature stream Γ , upon which it must be accepted or rejected immediately. Thus streaming feature selection (Algorithm 1) consists of two decision problems, considered as sub-procedures:

Definition 4. *Let \mathcal{F} be the set of features accepted as of time t , and let f_t be proposed at time t . The streaming feature acceptance decision problem is to accept f_t , setting $\mathcal{F} \leftarrow \mathcal{F} \cup f_t$, or reject, leaving \mathcal{F} unchanged.*

Definition 5. *The streaming feature pruning decision problem is to remove a subset $\mathcal{F}_0 \subset \mathcal{F}$ of low-quality features, setting $\mathcal{F} = \mathcal{F} \setminus \mathcal{F}_0$.*

A.1 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 algorithm state.
3. *Robust to over-submission.* The algorithm should be ro-

bust to processing many more feature submissions than raw variables present in the data (i.e. $|\Gamma| \gg p$, where p is the dimensionality of the data domain \mathcal{X}). 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. \(2017\)](#), 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.

A.2 Logical alpha-investing

As a first (unsuccessful) approach, we consider *logical alpha-investing*. Alpha-investing ([Zhou et al., 2005](#)) is one algorithm for streaming feature selection. It maintains a time-varying parameter, α_t , which controls the algorithm’s false-positive rate and conducts a likelihood ratio test to compare the current features with their potential addition by the new feature.

We can extend it to support feature definitions rather than feature values as follows. Compute the likelihood ratio $T = -2(\log \hat{L}(\mathcal{F}) - \log \hat{L}(\mathcal{F} \cup f_t))$, where $\hat{L}(\cdot)$ is the maximum likelihood of a linear model. Then $T \sim \chi^2(q_{f_t})$ and we compute a p-value accordingly. If $p < \alpha_t$, then f_t is accepted, otherwise it is rejected. α_t is adjusted according to an update rule that is a function of the sequence of accepts/rejects ([Zhou et al., 2005](#)).

Unfortunately, logical alpha-investing algorithm does not satisfy the design criteria of Ballet because it is neither stateless and nor robust to over-submission. The pitfall is that α_t must be securely stored somewhere and is also affected by rejected features — adversaries could repeatedly submit noisy features that are liable to be rejected, artificially lowering the threshold for high-quality features.

A.3 SFDS

Instead, we present a new algorithm, SFDS, for streaming feature definition selection based on mutual information criteria. It extends the GFSSF algorithm ([Li et al., 2013](#)) 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 (Algorithm 2), we first determine if a new feature f_t is *strongly relevant*, that is, whether the information $f_t(\mathcal{D})$ 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_t and some other feature $f \in \mathcal{F}$ provide similar information about Y . If f_t is determined to be superior than such an f , then f_t can be accepted. Later, in the pruning stage (Algorithm 3), f and any other redundant features are pruned.

Algorithm 1 Streaming feature definition selection

input Feature stream Γ , dataset \mathcal{D}
 1: $\mathcal{F} \leftarrow \emptyset$
 2: **loop**
 3: $f_t \leftarrow$ get next feature from Γ
 4: **if** $\text{accept}(\mathcal{F}, f_t, \mathcal{D})$ **then**
 5: $\mathcal{F} \leftarrow \text{prune}(\mathcal{F}, f_t, \mathcal{D})$
 6: $\mathcal{F} \leftarrow \mathcal{F} \cup f_t$
 7: **end if**
 8: **end loop**

Algorithm 2 SFDS Acceptance

input Accepted features \mathcal{F} , proposed feature f_t , dataset \mathcal{D}
output accept/reject
 1: **if** $I(f_t(\mathcal{D}); Y|\mathcal{F}(\mathcal{D})) > \lambda_1 + \lambda_2 \times q_{f_t}$ **then**
 2: **return true**
 3: **end if**
 4: **for** $f \in \mathcal{F}$ **do**
 5: $\mathcal{F}' \leftarrow \mathcal{F} \setminus f$
 6: **if** $I(f_t(\mathcal{D}); Y|\mathcal{F}'(\mathcal{D})) - I(f(\mathcal{D}); Y|\mathcal{F}'(\mathcal{D})) > \lambda_1 + \lambda_2 \times (q_{f_t} - q_f)$ **then**
 7: **return true**
 8: **end if**
 9: **end for**
 10: **return false**

CMI estimation In the SFDS algorithm, we compute several quantities of the form, $I(f_t(\mathcal{D}), Y|\mathcal{F}(\mathcal{D}))$, i.e. the conditional mutual information (CMI) of the proposed feature and the target given the set of accepted features. Since we do not know the true joint distribution of feature values and target, we must derive an estimator for this quantity. Let $Z = f_t(\mathcal{D})$ and $X = \mathcal{F}(\mathcal{D})$, i.e. the feature values extracted by feature f_t and feature set \mathcal{F} , respectively. Then CMI is given by $I(Z; Y|X) = H(Z|X) + H(Y|X) - H(Z, Y|X)$.

Algorithm 3 SFDS Pruning

input Accepted features \mathcal{F} , newly accepted feature f_t , dataset \mathcal{D}
output Pruned features \mathcal{F}
 1: **for** $f \in \mathcal{F} \setminus f_t$ **do**
 2: **if** $I(f(\mathcal{D}); Y|\mathcal{F}(\mathcal{D})) < \lambda_1 + \lambda_2 \times q_f$ **then**
 3: $\mathcal{F} \leftarrow \mathcal{F} \setminus f$
 4: **end if**
 5: **end for**
 6: **return** \mathcal{F}

We represent feature values as joint random variables with separate discrete and continuous components, i.e. $Z = (Z^d, Z^c)$ and $X = (X^d, X^c)$. This poses a challenge in estimation due to the mixed variable types. To address this, we adapt prior work (Kraskov et al., 2004) on mutual information estimation to handle the calculation of CMI in the setting of mixed tabular datasets.

Let \mathcal{F} be the set of accepted features at time t with corresponding feature values $X = \mathcal{F}(\mathcal{D})$. Then a new feature arrives, f_t , with corresponding feature values $Z = f_t(\mathcal{D})$.

The conditional mutual information (CMI) is given by:

$$I(Z; Y|X) = H(Z|X) + H(Y|X) - H(Z, Y|X) \quad (2)$$

Applying the chain rule of entropy, $H(A, B) = H(A) + H(B|A)$, we have:

$$\begin{aligned} I(Z; Y|X) &= H(Z, X) - H(X) + H(Y, X) \\ &\quad - H(X) - H(Z, Y, X) + H(X) \\ &= H(Z, X) + H(Y, X) \\ &\quad - H(Z, Y, X) - H(X) \end{aligned} \quad (3)$$

We represent feature values in separate components of discrete and continuous random variables, i.e. $X = (X^d, X^c)$:

$$\begin{aligned} I(Z; Y|X) &= H(Z^d, Z^c, X^d, X^c) + H(Y, X^d, X^c) \\ &\quad - H(Z^d, Z^c, Y, X^d, X^c) - H(X^d, X^c) \end{aligned} \quad (4)$$

We expand the entropy terms again using the chain rule of entropy to condition on the discrete components of the random variables:

$$\begin{aligned} I(Z; Y|X) &= H(Z^c, X^c|Z^d, X^d) + H(Z^d, X^d) \\ &\quad + H(Y, X^c|X^d) + H(X^d) \\ &\quad - H(Z^c, Y, X^c|Z^d, X^d) - H(Z^d, X^d) \\ &\quad - H(X^c|X^d) - H(X^d) \end{aligned} \quad (5)$$

After cancelling terms:

$$I(Z; Y|X) = H(Z^c, X^c|Z^d, X^d) + H(Y, X^c|X^d) - H(Z^c, Y, X^c|Z^d, X^d) - H(X^c|X^d) \quad (6)$$

We use the definition of conditional entropy and take the weighted sum of the continuous entropies conditional on the unique discrete values. Let Z^d have support U and X^d have support V .

$$I(Z; Y|X) = \sum_{u \in U, v \in V} p_{Z^d, X^d}(u, v) H(Z^c, X^c|Z^d = u, X^d = v) + \sum_{v \in V} p_{X^d}(v) H(Y, X^c|X^d = v) - \sum_{u \in U, v \in V} p_{Z^d, X^d}(u, v) H(Z^c, Y, X^c|Z^d = u, X^d = v) - \sum_{v \in V} p_{X^d}(v) H(X^c|X^d = v) \quad (7)$$

Unfortunately, we cannot perform this calculation directly as we do not know the joint distribution of X , Y , and Z . Thus we will need to estimate the quantities p and H based on samples from their joint distribution observed in \mathcal{D} . For this, we make use of two existing estimators.

A.4 Kraskov entropy estimation

Kraskov et al. (2004) present estimators for mutual information (MI) based on nearest-neighbor statistics. From the assumption that the log density around each point is approximately constant within a ball of small radius, simple formulas for MI and entropy can be derived. The radius $\epsilon(i)/2$ is found as the distance from point i to its k th nearest neighbor. Unfortunately, their MI estimator cannot be used for CMI estimation and also cannot directly handle mixed discrete and continuous datasets. However, we are able to adapt their entropy estimator for our own CMI estimation.

The Kraskov entropy estimator \hat{H}^{KSG} for a variable A is given by:

$$\hat{H}^{\text{KSG}}(A) = \frac{-1}{N} \sum_{i=1}^{N-1} \psi(n_a(i) + 1) + \psi(N) + \log(c_{d_A}) + \frac{d_A}{N} \sum_{i=1}^N \log(\epsilon_k(i)), \quad (9)$$

where ψ is the digamma function, $n_a(i)$ is the number of points within distance $\epsilon_k(i)/2$ from point i , and c_{d_A} is the volume of a unit ball with dimensionality d_A .

Consider the joint random variable $W = (X, Y, Z)$. Then $\epsilon_k^W(i)$ is twice the distance from the i th sample of W to its k th nearest neighbor.

The entropy of W is then given by

$$\hat{H}^{\text{KSG}}(W) = \psi(k) + \psi(N) + \log(c_{d_X} c_{d_Y} c_{d_Z}) + \frac{d_X + d_Y + d_Z}{N} \sum_{i=1}^N \log(\epsilon_k^W(i)), \quad (10)$$

A.5 Empirical probability estimation

Let A be a discrete random variable with an unknown probability mass function p_A . Suppose we observe realizations a_1, \dots, a_n .

Then the empirical probability mass function is given by

$$\hat{p}_A(A = a) = \frac{1}{n} \sum_{i=1}^n \mathbf{1}^{\{a_i=a\}}. \quad (11)$$

A.6 CMI Estimator Formula

Now we can substitute our estimators \hat{p} from Equation (11) and \hat{H}^{KSG} from Equation (10) into Equation (8).

Finally, we can use estimators for p and H to estimate I :

$$\hat{I}(Z; Y|X) = \sum_{u \in U, v \in V} \hat{p}(u, v) \hat{H}^{\text{KSG}}(Z^c, X^c|Z^d = u, X^d = v) + \sum_{v \in V} \hat{p}(v) \hat{H}^{\text{KSG}}(Y, X^c|X^d = v) - \sum_{u \in U, v \in V} \hat{p}(u, v) \hat{H}^{\text{KSG}}(Z^c, Y, X^c|Z^d = u, X^d = v) - \sum_{v \in V} \hat{p}(v) \hat{H}^{\text{KSG}}(X^c|X^d = v) \quad (12)$$

B ADDITIONAL EVALUATION

The case study in the Sections 5 and 6 presents a rigorous and large-scale evaluation to better understand the characteristics of live collaborative data science projects. In this appendix, we report on previously performed smaller evaluation steps during the iterative design process of Ballet to measure efficacy and inform design.

We evaluate Ballet along several dimensions, focusing on the efficacy and usability of this new development paradigm for collaborations of different sizes and the characteristics of the proposed SFDS algorithm (Appendix A.3). In synthetic collaborations, we find that collaborative development leads to feature engineering pipelines with better and more diverse features.

B.1 User study: prototype framework

We evaluated an initial prototype of Ballet in a user study with 8 researchers and data scientists. We explained the framework and gave a brief tutorial on how to write features. Participants were then tasked with writing features to help predict the incidence of dengue fever given historical data from Iquitos, Peru and San Juan, Puerto Rico (Epidemic Prediction Initiative). Three participants were successfully able to merge their first feature within 30 minutes, while the remainder produced features with errors or were unable to write a new feature. In interviews, participants suggested that they found the Ballet framework helpful for structuring submissions and validating features, but were unfamiliar with writing feature engineering code in terms of feature definitions with separate fit and transform behavior. Based on this feedback, we created the `ballet.eng` library of feature engineering primitives and created tutorial materials for new contributors.

B.2 Case study: house price prediction

To test the effectiveness of Ballet and its alignment with our design goals, we simulate a feature engineering collaboration on the Ames housing price prediction problem (De Cock, 2011). The goal of the problem is to accurately predict the selling price of houses in Ames, Iowa given detailed characteristics of each house collected by the local property assessor’s office. For this real-world data, feature engineering plays an important role: there are many missing values, redundant and irrelevant variables, and informative patterns to be found by applying different transformations. For example, various measures of house square footage, such as lot frontage, lot area, first floor area, second floor area, etc. can be intelligently combined in the case that any of these individual measures contains irregularities. Alternately, categorical variables representing administrative data on building material quality, neighborhood, lot type, etc. need to be processed properly to be including in an ML model.

This problem also appears on Kaggle, a data science competition community. We identified 9 public notebooks by Kaggle members in which they engineer features as part of their end-to-end pipeline. We manually re-implement these as Ballet Features, resulting in 311 features. We then simulate a scenario in which each Kaggle user separately submits their features to a Ballet project as if they had been collaborating from the outset, randomly permuting the order the features arrive to the project, and observe the feature validation outcome.

Summary. Of the original 311 features proposed to the dataset, 12 features were accepted into the project when they were proposed. 6 of those features were eventually

pruned by new features entering the project, resulting in 6 total features left by the end of the simulation (Figure 5).

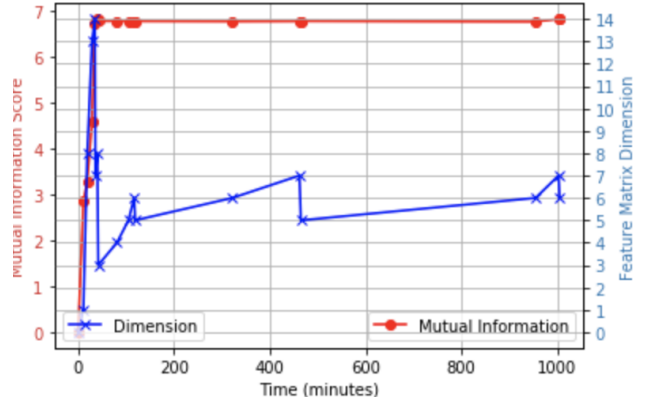


Figure 5. Mutual information score $I(\mathcal{F}(\mathcal{D}); Y)$ and feature matrix dimensionality over time in a simulated collaboration. Time is measured since start of project simulation in minutes. Points only appear after a feature is accepted or a feature is pruned. Pruning generally occurs several minutes (2–3) after a feature is accepted.

Recall that the mutual information between two datasets is bounded above by the smaller of the dataset entropies. Thus using SFDS, the mutual information between the feature matrix $\mathcal{F}(\mathcal{D})$ and its target column Y is upper bounded by the entropy of the target. For the simulation Ames dataset, we find $H(Y) \approx 6.3$. We see this reflected in Figure 5: after the first four features are accepted, the mutual information between the feature matrix and target column stays relatively the same for the rest of the simulation as the algorithm allows fine-grained improvements.

This is an interesting result as it suggests that a significant portion of the feature engineering performed by the nine data scientists, working independently, was redundant with existing features or unhelpful for making predictions. If these data scientists had been actually collaborating (rather than the combination of their work in our simulation), they may have avoided much of this duplication.

Pruning. As features are accepted to the Ames project simulation, they may cause existing features to be pruned and removed from the project. We visualize the effect of pruning features in Figure 6.

As expected, usually a feature that gets pruned has less information than the feature that prunes it. However, there are some other examples that reveal much about the workings of the SFDS algorithm. For example, in the one case in which a feature with low information caused a feature with high information to be pruned, we found that the more relevant feature, while having more information overall, had very little information conditioned on the other features. Essentially, the feature was already partially redun-

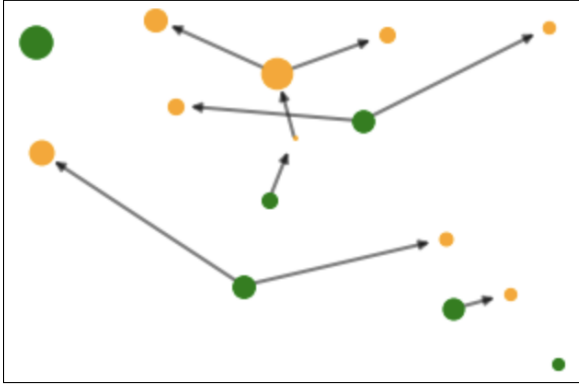


Figure 6. A directed graph representing features pruning one another. Green nodes represent features that have been accepted and orange nodes represent features that were accepted but then pruned later on. The size of the node is proportional to the mutual information score of that feature with respect to the target. An edges $f_i \rightarrow f_j$ represents that the acceptance of f_i caused f_j to be pruned.

dant and the acceptance of the less relevant feature made it fully redundant. We also noticed deep nesting of pruning in the features; often, it may be the case that the feature that pruned a feature f_i may itself be pruned as well. This can be understood as users constructing similar features; because the Kaggle members did not actually collaborate during feature engineering, there was a high level of redundancy in the features we collected. Features that apply similar transformations and are built on the same data columns in the raw dataset are often similar and tend to prune each other if the new feature is found to be better.