Scaling Collaborative Open Data Science

by

Micah J. Smith

B.A., Columbia University (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2018

Certified by .....
Kalyan Veeramachaneni
Principal Research Scientist
Laboratory for Information and Decision Systems
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Scaling Collaborative Open Data Science

by

Micah J. Smith

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Large-scale, collaborative, open data science projects have the potential to address important societal problems using the tools of predictive machine learning. However, no suitable framework exists to develop such projects collaboratively and openly, at scale. In this thesis, I discuss the deficiencies of current approaches and then develop new approaches for this problem through systems, algorithms, and interfaces. A central theme is the restructuring of data science projects into scalable, fundamental units of contribution. I focus on feature engineering, structuring contributions as the creation of independent units of feature function source code. This then facilitates the integration of many submissions by diverse collaborators into a single, unified, machine learning model, where contributions can be rigorously validated and verified to ensure reproducibility and trustworthiness. I validate this concept by designing and implementing a cloud-based collaborative feature engineering platform, FeatureHub, as well as an associated discussion platform for real-time collaboration. The platform is validated through an extensive user study and modeling performance is benchmarked against data science competition results. In the process, I also collect and analyze a novel data set on the feature engineering source code submitted by crowd data scientist workers of varying backgrounds around the world. Within this context, I discuss paths forward for collaborative data science.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist, Laboratory for Information and Decision Systems

Acknowledgments

I'd like to thank my advisor, Kalyan Veeramachaneni, for invaluable mentorship, feedback, resources, and support. I'd like to thank all of the members of the Data To AI Lab, both past and present, who have contributed feedback and valuable discussions. Roy Wedge and Max Kanter have been great collaborators. I'd also like to thank the members of our FeatureHub platform user study for their participation and feedback. Finally, I'd like to thank all of my friends and family who have supported me in my studies.

Contents

1 Introduction **19**

1.1 Motivating open data science 21

1.2 The modern data science process 23

1.3 The state of collaboration 27

1.3.1 Shared notebooks 29

1.3.2 Vanilla software engineering 30

1.3.3 Hosted platforms 30

1.3.4 Data science competitions 32

1.3.5 Challenges of open data science 34

1.4 Data science abstractions 35

2 Background and Related Work **39**

2.1 Crowdsourcing methods in machine learning 39

2.1.1 Crowdsourced data labeling 39

2.1.2 Crowdsourced data cleaning 40

2.1.3 Crowdsourced feature engineering 40

2.2 Automated methods in machine learning 40

2.2.1 Automated machine learning 41

2.2.2 Automated feature engineering 42

2.3 Programming and data science competitions 42

2.4 Collaborative software development 44

3 A collaborative data science platform **45**

3.1 Overview . . . 47

3.2 Launching a project . . . 48

3.3 Creating new features . . . 49

3.3.1 Load and explore . . . 49

3.3.2 Write features . . . 51

3.3.3 Evaluate and submit . . . 53

3.3.4 Collaborate . . . 55

3.4 Combining contributions . . . 56

3.5 Design . . . 58

4 Assessment **61**

4.1 Experimental conditions . . . 61

4.1.1 Study participants . . . 62

4.1.2 Prediction problems . . . 63

4.1.3 Experiment groups . . . 63

4.1.4 Performance evaluation . . . 64

4.2 Modeling performance . . . 65

4.3 Evaluating collaboration . . . 69

4.4 Analysis of contributors . . . 71

4.4.1 Worker attributes and output . . . 72

4.4.2 Contributions in source code . . . 74

4.4.3 Educational opportunities . . . 74

5 Discussion **85**

5.1 Opportunities . . . 85

5.1.1 Flexible data science abstractions . . . 85

5.1.2 Automated methods to augment collaboration . . . 87

5.1.3 Creating interpretable models . . . 89

5.2 Challenges we faced . . . 90

5.2.1 Maintaining platforms . . . 90

5.2.2 Administering user studies . . . 92
5.2.3 Motivating contributions . . . 93
5.2.4 Adversarial behavior . . . 95

6 The future of collaborative data science 97
6.1 The problem with platforms . . . 97
6.2 A turn towards platformless collaboration . . . 99
6.3 Lightweight collaborative systems . . . 100

A Implementation of FeatureHub 109
A.1 Configuration . . . 109
A.2 User interface . . . 112
A.3 Feature evaluation . . . 113
A.4 Discussion forum . . . 114
A.5 Automated machine learning . . . 114
A.6 Data management . . . 115
A.7 Deployment . . . 115

B Prediction problem details 119
B.1 Problem statement . . . 119
B.2 Data schemata . . . 121
B.3 Pre-extracted features . . . 123

C User study survey 127
C.1 Survey questions . . . 127
C.1.1 Getting oriented . . . 128
C.1.2 FeatureHub Platform . . . 128
C.1.3 Forum . . . 130
C.1.4 Collaboration . . . 130
C.2 Survey responses . . . 132

D Listings 135

List of Figures

1-1 High-level view of the modern data science process[1]. Different phases of this highly iterative process can cause prior work or assumptions to be revisited. First, an organization must determine that data science solutions are needed for a problem rather than traditional software or logic systems. The organization's high-level goal is then formulated as a precise prediction problem. Data engineers and data scientists survey and prepare data resources, like databases and APIs. Next is the all-important feature engineering step, in which data scientists try to transform these raw data sources into a *feature matrix* that can be used as the input to a predictive machine learning model. With the feature matrix in hand, data scientists try to formulate and train a model and then evaluate it for predictive accuracy and generalization potential. Finally, the model can be deployed to make predictions. The process doesn't stop there, as monitoring and maintenance of the model reveal more details about the relationship between the predictions and the organization's goals. 24

3-1 Overview of FeatureHub workflow. A coordinator receives the problem to be solved, and associated data. She prepares the data and does the necessary preprocessing using different available functions. Then, the platform is launched with the given problem and dataset. Data scientists log in, interact with the data, and write features. The feature scripts are stored in a database along with meta information. Finally, the coordinator automatically combines multiple features and generates a machine learning model solution. 46

3-2 It is straightforward for users to use the FeatureHub client to import a sample dataset for an example problem called `demo`. 51

3-3 Example interactions with the `evaluate` and `submit` API methods. In evaluating a feature, the feature is executed in the user's notebook environment to extract feature values, which are validated against a list of criteria. If successful, feature performance metrics are returned. In submitting a feature, the user is prompted for a description, after which the feature source code is extracted and sent to the evaluation server. Upon success, the evaluation metrics are returned, as well as a link to an automatically-generated forum post (if applicable). 53

3-4 Natural language description of a feature function written in Python. These descriptions can be helpful for interpreting model performance and establishing trust in the implementation. 54

3-5 Output of a call to `discover_features` in the Jupyter Notebook. In this example, the user has filtered features to those containing the string `'action'` and then scrolled to inspect Feature 48. Scrolling further, the user will be able to see the performance metrics (accuracy, precision, recall, and ROC AUC) for this feature before continuing on to the other matching features in the database. 56

3-6 A forum post for the *sberbank* prediction problem (Section 4.1.2) showing an actual feature submitted by a crowd worker. The feature description, source code, and computed metrics are shown, along with prompts for further discussion. 57

3-7 Overview of FeatureHub platform architecture, comprising the FeatureHub computing platform and the Discourse-based discussion platform. Users log into the platform to find a customized Jupyter Notebook environment with dedicated compute and storage resources (1). They can query the feature database to see the work that others have already done (2). Once they have an idea for a feature, the user implements it and evaluates it locally on training data, then submits their code to the feature evaluation server (3). The code is validated as satisfying some constraints to ensure that it can be integrated into the predictive model without errors. The corresponding feature is extracted and an automated machine learning module selects and trains a predictive model using the candidate features and other previously-registered features (4, 5). The results are registered to the database (6), posted to the discussion forum (7), and returned to the user (8). 59

4-1 Hourly wages of data scientists from Upwork. 62

4-2 Performance of FeatureHub integrated model as compared to independent data scientists. For *airbnb*, performance is measured as normalized discounted cumulative gain at $k = 5$. The negative of the scoring metric is reported on the y-axis to facilitate a "smaller-is-better" comparison. The automatically generated FeatureHub model ranked 1089 out of 1461 valid submissions, but was within 0.03 points of the best submission. For *sberbank*[2], performance is measured as root mean squared logarithmic error. The automatically generated FeatureHub model ranked 2067 out of 2536 valid submissions, but was within 0.05 points of the best submission. 67

4-3 The amount of days, by Kaggle competitor, from their first submission to the problem to their first submission that achieved a better score than FeatureHub on *airbnb*. Since we only see the first submission, rather than when the competitor first starting working, this omits the effect of participants' preprocessing, feature engineering, and modeling work over as much as 2.5 months before initially submitting a solution. 69

4-4 Features for *airbnb* projected on first two PCA components by FeatureHub (grey circles) and Deep Feature Synthesis (red squares). . . 70

4-5 Features for *sberbank* projected on first two PCA components by FeatureHub (grey circles) and Deep Feature Synthesis (red squares). . . 70

4-6 Relationship between hourly wage and feature performance, including linear trend and 95% confidence interval. For the *sberbank* problem, performance is measured by the maximum R^2 of any feature submitted ($\beta = 0.0008,\ p = 0.025$). For the *airbnb* problem, performance is measured by the maximum ROC AUC of any feature submitted ($\beta = 1.8e-5,\ p = 0.018$). Y-axis scales are not comparable. 72

4-7 Relationship between data scientist experience and feature performance. For the *sberbank* problem, performance is measured by the maximum R^2 of any feature submitted. For the *airbnb* problem, performance is measured by the maximum ROC AUC of any feature submitted. Y-axis scales are not comparable. 73

4-8 Anti-pattern of iteration on `DataFrame`s. The original submission, adapted for clarity, uses a Python `range` to iterate over rows directly. An improved version instead leverages highly-optimized vectorized code under the hood to achieve performance and readability benefits. . . . 76

4-9 Anti-pattern of reinventing the wheel in terms of *pandas* library functions. The original submission, adapted for clarity, implements a procedure to fill missing values with zeros, even though the library method `fillna` is available and widely used for the same task. (The original source code also demonstrates more general programming anti-patterns, such as the use of `.apply(lambda a: cleankitch_sq(a))` instead of `.apply(cleankitch_sq)`.) 77

4-10 A frequent task for data scientists is to process datetime variables, which is often done by converting the variable to a native datetime data type and then using robust library methods. In each listing, a snippet of a submitted feature instead shows the processing of a timestamp using string methods or arithmetic directly. While these approaches "work," they should be discouraged. (Example raw timestamps in the data for the `timestamp` and `timestamp_first_active` columns take the form `"2013-05-27"` and `20120519182207`, respectively.) 79

5-1 Feature engineering represented as a directed, acyclic data-flow graph. Edges represent the flow of data between nodes, which transform their inputs and send the output to other consumers. 88

A-1 Entity-relationship diagram for FeatureHub database. The crow's foot arrow represents a one-to-many relationship; that is, each feature has many metrics associated with it, and so on. 116

B-1 Entity-relationship diagram for Kaggle *airbnb* prediction problem. The entities table is `users` and the target table is `target`. The list of attributes is truncated for `users` (15 total). 121

B-2 Entity-relationship diagram for Kaggle *sberbank* prediction problem. The entities table is `transactions` and the target table is `target`. The list of attributes is truncated for `transactions` (291 total) and `macro` (100 total). 122

List of Tables

1.1 Comparison of selected hosted data science platforms. Each platform shown, both from commercial and academic developers, provides the ability to write, edit, and host source code; store data; and compute using cloud resources. The aims of these platforms vary — from enabling data scientists in companies to work more effectively (the commercial products), to providing a managed compute environment to support analysis of large atmospheric and oceanographic datasets (Pangeo), to enabling reproducibility of computational academic research (CodeOcean). 31

3.1 Set of methods for data scientists to interact with FeatureHub. . . . 50

4.1 Prediction problems used in FeatureHub user testing. 63

4.2 Bonus payment weights (in %) for control and experimental groups. For groups 1 and 2, the "Collaboration" category was listed as "Documentation" to avoid bias, and the "Forum interactions" subcategory was not shown. 65

B.1 Metrics for decision tree model training on pre-extracted feature matrix for different splits. 124

B.2 Metrics for decision tree model training on pre-extracted feature matrix for different splits. 125

C.1 Summary of survey responses for intro questions (Appendix C.1.1), . 132

C.2 Summary of survey responses for platform questions (Appendix C.1.2). 133

C.3 Summary of survey responses for forum questions (Appendix C.1.3). . 133
C.4 Summary of survey responses for collaboration questions (Appendix C.1.4).134

Chapter 1

Introduction

In the modern world, the vast quantities of data that have been collected by organizations are affecting our lives in many ways. Every day, we rely on services that can predict the best train routes, suggest the most interesting movies and hottest musical tracks, and protect us from financial fraud. However, for every successful data science application, there are many more potential projects that have not yet been developed. Every two days, we generate as much data as we did from the dawn of civilization until 2003 [43], and the size of the digital universe is doubling every two years, up to a projected 44 trillion gigabytes by 2020 [49].

The amount of data collected has significantly outpaced the supply of data scientists and analytics professionals who are able to derive value from this data. Job openings alone for data scientists are projected to grow to over 60,000 by 2020 [6]. Even so, qualified candidates are often expected to have completed doctoral degrees. The data science skill set includes knowledge of statistics, optimization, and machine learning; data cleaning, munging, and wrangling; big data computing and distributed systems engineering; communication and presentation ability; and problem domain knowledge. It may require many years of education and experience for one person to accumulate all these distinct skills.

To relieve human-related bottlenecks associated with a shortage of data science resources, organizations can resort to various innovations. These include data science automation technologies, which can extract insights from datasets without human in-

tervention; crowdsourcing-augmented workflows; and cloud computing and improved hardware.

Another promising approach is to facilitate increased collaboration in data science projects, scaling the size of data science *teams* from ones or tens of collaborators to hundreds or thousands. These larger collaborations can include many diverse types of collaborators: lay contributors like business analysts and domain experts as well as other technical stakeholders. However, also at this scale, new barriers to collaboration arise that do not apply to the same extent in more mature areas such as software engineering. These include co-management of data and source code, including provenance, versioning, storage, and branching for raw data, data pipelines, and other data science artifacts; communicating and coordinating work to avoid redundancy; and combining contributions into a single data science product. Existing workflows for dealing with these sorts of issues are not easily parallelizable and do not easily scale to very large teams. More innovations are needed to realize the potential of large-scale data science collaborations.

For the largest enterprises, to which data science, predictive modeling, and artificial intelligence are important assets, intense engineering effort has made these problems more manageable. But the state of affairs is poor in organizations with fewer resources, in particular for *open data science*. To this point, open data science collaborations have had difficulty scaling beyond tens of contributors, in stark contrast to the most successful open-source software projects. Tailored solutions are needed in order to scale these valuable projects to thousands of contributors.

Given the challenges of scaling data science efforts, and the unique obstacles facing teams of data scientists within even the most resourceful organizations, new approaches to collaboration are needed. In this thesis, I will present my research on scaling collaborative data science endeavors with a focus on open data science.

1.1 Motivating open data science

What is *open data science*? In an open data science project, diverse collaborators try to use data science and predictive modeling in the open-source paradigm for societal benefit[1].

In a commercial setting, many of the most successful applications of data science come in advertising technology ("Ad Tech"). Companies in e-commerce, social media, and digital news have developed highly sophisticated data operations in which clickstreams from customer interactions with digital platforms are combined with harvested personal and demographic information to price and sell digital ads. In a welcome contrast, the goals of open data science efforts are analyses or predictive models that have outsized societal impacts. These projects are crucial in areas with large potential impact but less formal resources available, such as education, health care, environmental studies, conservation, civic administration, and government accountability.

Successful open data science projects are relatively scarce at this point, but they hint at what is possible for this movement.

- The *Fragile Families Challenge* tasks researchers with predicting key socioeconomic outcomes for disadvantaged children and their families, with the goal of targeting clinical and sociological interventions [40].

- The *Boston crash modeling* project, developed by the organization Data For Democracy, is an application that can predict where and when car crashes are most likely to occur on Boston city streets in order to increase pedestrian safety [9]. Through this application, city officials could target timely interventions to predict crashes before they happen. The development of this application was spurred by city officials and concerned citizens under a "Vision Zero" initiative, but with insufficient resources available to city governments, the open data

[1] Contrast this definition with another sense of "open data science" to mean open-source software libraries that have been developed for use within data science projects. Examples include *pandas*, *ggplot2*, *scikit-learn*, and *Tensorflow*. Each of these projects in itself is a software library, not a data science project.

science community was put to the task.

- The city of Chicago, along with private and open-source collaborators, has developed a *Food inspections evaluation* application to make predictions of critical violations at food establishments in Chicago [33]. Using this application, city public health staff can prioritize inspections to find critical violations faster.

While it is a challenge to build machine learning models in centralized organizations, it is even more difficult to do so in the open data science setting. Contributors to open data science projects may range from experienced data scientists and application developers to novice programmers and lay domain experts. Collaborators may be located around the world, in different time zones, speaking different languages. They may not have the fastest internet connections, let alone the resources to train deep neural networks on cloud GPU clusters or subscribe to expensive commercial data science productivity software products.

What can we actually build with approaches like these? Consider what has been accomplished with large-scale collaborations in open-source software development. The Linux kernel is considered one of the most successful open-source projects ever, with over $13,000$ developers having contributed over $700,000$ commits to the codebase. By taking advantage of decentralized version control systems, highly active mailing lists and forums, and a larger ecosystem of open-source tooling and systems, developers from around the world have contributed actively and effectively, to this project and many others. The development of massively collaborative software systems is a major achievement of software engineering and computer systems practitioners and researchers.

If we could scale the size of open data science collaborations to match that of the most successful open-source software projects, what could we achieve? Let us return to the Fragile Families Challenge. Researchers were given a large and complex longitudinal dataset on children and their families, and tasked with predicting various key socioeconomic outcomes, such as GPA and eviction. Under the large-scale open data science paradigm, this problem, with the power to impact so many lives, could

be worked on by hundreds or thousands of social scientists, data scientists, clinicians, economists, educators, and citizen scientists. A psychologist could write a feature that encodes their expertise on the relationship between maternal engagement and academic performance. An educator could write a feature that combines several measures of teacher performance together. A statistician could devise a customized imputation procedure for highly correlated variables. And a machine learning researcher could implement a learning algorithm to deal with the high dimensionality of the dataset. Sophisticated automated machine learning technologies and tightly engineered systems would support the efforts of these researchers, enabling them to extract the most insight with the least effort. The progress of the unified model could be transparently reviewed in real time, and deployed for clinical and sociological interventions.

In this vein, I consider the promise of large-scale collaborative open data science. This will be characterized by the ability of thousands of people around the world to openly collaborate on one solution to a single data science task, incrementally building upon and improving the same predictive machine learning model. This "integrated model" can be deployed at any point with minimal effort to address important societal challenges.

1.2 The modern data science process

In discussing the topic of this thesis, it will help to review the modern data science process. Certainly, the realities of data science as it is currently practiced vary widely, and there is considerable disagreement about how to define the scope of the term "data science" at all. Without dwelling on those discussions, I will focus at first on predictive modeling through machine learning as deployed in a commercial setting. (In Section 1.3.5, I will describe similarities to and differences from this process in the open data science paradigm.) The data science process is highly interconnected and iterative, and the results of one phase frequently cause some other phase to be revisited. It may involve many people of different roles, from organization leadership to data scientists to DevOps engineers. A depiction of this process can be seen in

Figure 1-1. As one example, the reader can keep an mind an internet company that sells customers a monthly subscription to receive a new novel in the mail.

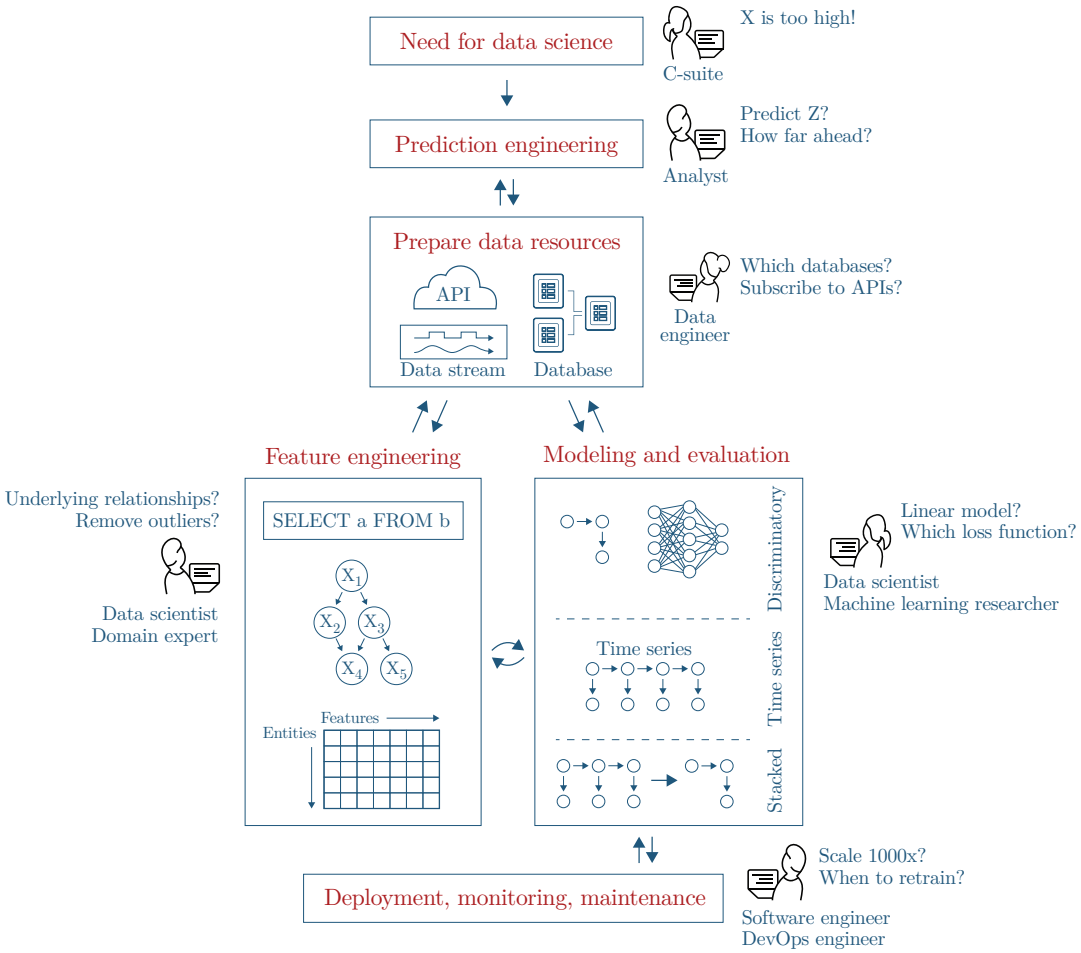


Figure 1-1: High-level view of the modern data science process[2]. Different phases of this highly iterative process can cause prior work or assumptions to be revisited. First, an organization must determine that data science solutions are needed for a problem rather than traditional software or logic systems. The organization's high-level goal is then formulated as a precise prediction problem. Data engineers and data scientists survey and prepare data resources, like databases and APIs. Next is the all-important feature engineering step, in which data scientists try to transform these raw data sources into a *feature matrix* that can be used as the input to a predictive machine learning model. With the feature matrix in hand, data scientists try to formulate and train a model and then evaluate it for predictive accuracy and generalization potential. Finally, the model can be deployed to make predictions. The process doesn't stop there, as monitoring and maintenance of the model reveal more details about the relationship between the predictions and the organization's goals.

[2] Adapted from [17].

Recognize the need for data science. At first, an organization may not see a need for data science at all. Existing processes or non-data driven software systems may produce suitable support for decision making. It may be difficult to commit to a potentially costly investment in data science resources. Ultimately, the organization may decide that the benefit from predictive modeling (or some other data science product) outweighs its costs. In our case, perhaps the book subscription company is struggling with identifying which of its customers are likely to churn (discontinue subscriptions) and has decided to try to predict this outcome in order to take preemptive action.

Formulate a problem. Translating a business need into a well-defined prediction problem is not an easy task. Data scientists must take open-ended directives (e.g. "predict customer churn") and determine specific prediction problems that are most useful. This involves specifying exact prediction horizons, training datasets, and labeling functions and writing code to apply these ideas. For our example company, a data scientist might decide to formulate a problem as follows: *for each customer and each week, predict whether the customer will deactivate their subscription within the next one week using data from the last 60 days*. The data scientists must also take care not to forget the business objective of *reducing customer churn* — that is, a formulation of a prediction problem that results in lower predictive accuracy might actually yield *more* benefit to the company if action taken in response to such predictions is more impactful. Previous authors have termed this phase "prediction engineering" [42].

Prepare data resources. It is not of much use to engage in data science if there is no data to work with. Data scientists take stock of the data resources in their organization: databases backing transactional systems, website clickstream logs, unstructured and textual data, and external linkable datasets and APIs. How long has data for each resource been collected such that it is usable for training machine learning models? How frequently is each resource updated and how quickly are updates visible to production systems that are serving predictions? What are the financial costs of using different data sources, and are there licensing or privacy issues? In some

cases, organizations might choose this opportunity to begin collecting new datasets for future use. The data resources and associated infrastructure available to different organizations varies enormously. For our example company, data scientists might identify a Postgres database backing the transaction system, clickstream logs uploaded nightly to Amazon Web Services (AWS) S3, email marketing records dumped by a third party accessible at noon each day, and demographic information from third party data brokers.

Clean and preprocess. All data is dirty. Once data resources has been identified, significant effort may be necessary for cleaning and preprocessing the data. For example, our example company might find that after the front-end team redesigned the website, the logging event `sign_out` changed to `log_out`, and a script must be written to standardize this name over time. Some cleaning and preprocessing work may be integrated directly into the subsequent feature engineering phase.

Feature engineering. Even after cleaning and preprocessing, the data is not in a form that is suitable for inclusion in machine learning models, in that it may be in a multi-table relational format or represented as categories, text or images. In the feature engineering step, data scientists create "features" from the raw data that associate numeric values with each entity. For example, data scientists might conjecture that the number of times customers of our example company log in to the website over a preceding period is predictive of churn. To extract feature values following this definition, the data scientists would need to write code to extract and aggregate specific login events from the website log data. This is a phase of the modern data science process in which the creativity and ingenuity of data scientists can, after sufficient effort, lead to significant performance gains. The output of the feature engineering process is a collection of vectors of numeric values associated with each entity, known as a *feature matrix*.

Formulate and evaluate machine learning models. With the feature matrix in hand, data scientists can begin to formulate machine learning models to predict the desired outcome. While composing modeling strategies, the data scientists will pay special care to overfitting and other problems that can negatively impact general-

ization performance on the live predictions. Data scientists will manually tune the hyperparameters of these models or resort to various automated methods. An iterative process of model development, model training, and model evaluation can be undertaken until the data scientists are satisfied with the model's performance. One common misconception about the data science process is that it entirely consists of finding an appropriate machine learning model given a feature matrix and then computing cross-validated metrics. We see here that such a phase, of formulating and evaluating machine learning models, is just one small part of the data science process.

Deploy, monitor, maintain. Finally, the data scientists are happy with their model and wish to deploy it to make live predictions. Depending on the organization and the development process, this could be as simple as changing a few lines in a configuration file, or as involved as providing detailed specifications to a separate engineering team to build a productionized version of the model and integrate it with the rest of the product. The deployed model must be monitored and maintained to ensure that its performance is as expected. Thus, this is not the final phase of the modern data science process at all, but a continuous undertaking for the lifetime of the product or business need.

1.3 The state of collaboration

Given this view of the modern data science process in commercial and open data science settings, we can consider the role that collaboration plays in data science and in existing attempts to collaboratively develop data science projects, with an attention to the unique challenges of the open data science paradigm.

It will be instructive to keep in mind the state of collaboration in the closely related field of software engineering. Since early large-scale projects like the development of the Apollo flight software by Margaret Hamilton and NASA and the development of IBM's OS/360 operating system, the field of software engineering has grown by leaps and bounds. Over this time, there has been ample work by both researchers and practitioners in creating tools, workflows, and methods to enable large-scale and

productive collaborations. An expansive ecosystem of tooling is freely available for software engineers at all stages of the software creation process. Innovations in the software engineering and programming languages communities include object-oriented programming, version-control systems, software testing, integrated development environments, and formal verification. These have enabled projects like Linux ($13,000+$ contributors), GNU ($4,500+$ contributors), nodejs ($2,000+$ contributors), and django ($1,500+$ contributors) to provide irreplaceable services to their users. Unfortunately, many of these tools, techniques, and successful large-scale collaborations have no analogue in data science.

Collaborative data science workflows include a wide spectrum of approaches, none of which has proven to be completely suitable for the demands of practitioners. In particular, it is challenging to collaboratively develop a *single data science solution*, rather than many parallel but co-located analyses, and to appropriately treat the data artifacts of such a project. The following data science collaboration approaches together form some of the options available for data science teams working today to organize their communication, software, analyses, and deployments.

I will evaluate commonly-used existing approaches to collaboration in data science projects along several dimensions: *integrability*, *usability*, and *scalability*. For the purposes of studying open data science, I will also address a dimension of *openness*.

- **Integrability.** The ability for the contributions of each collaborator to be readily integrated into a single data science product. For example, if two contributors each write source code to create a different feature, can their source code be easily combined to result in one procedure that generates two features?

- **Usability.** The usability of the collaborative approach in the general sense of software systems, with a special focus on the learnability for collaborators of various backgrounds, especially those who are not software engineers or computer scientists.

- **Scalability.** The degree to which the collaborative system can support contributions from a large number of collaborators. The system may enable realtime

(or close-to-realtime) feedback on contributions, an organizational structure to manage the volume of contributions, and software or automated systems to scale the review and integration of contributions into the main project.

- **Openness.** The fidelity of the software underlying the collaborative approach to the open-source software model as well as other issues of openness such as financial cost and data availability.

1.3.1 Shared notebooks

In this approach, data scientists develop analyses independently using interactive notebooks in the literate programming paradigm [21]. These interactive notebooks, such as Jupyter [19], R Markdown[3], Beaker[4], Zeppelin[5], and Mathematica[6], are a popular format for development and interactive editing and dissemination of results. Analyses are generally structured as long scripts, with intermediate outputs interspersed. When results are ready to be distributed, the notebooks are emailed to collaborators or uploaded to shared file hosting solutions like Dropbox. The shared notebooks approach is rarely used beyond teams sized in the tens of collaborators, though it is common to find notebooks included in much larger software projects for tutorial or explicatory purposes.

This approach earns high marks from a usability point of view, as these interactive documents are self-contained and widely used among engineers and researchers of many backgrounds. However, it is quite difficult to reuse any encapsulated routines from one notebook in another analysis, as most are structured as long scripts with expressions executed at the global scope, reducing the possibility of integrating code from multiple notebooks in a single data science product. Scaling collaboration becomes prohibitively difficult, as it is challenging enough to keep track of the notebooks of dozens or hundreds of collaborators, let alone version code snippets and marshal these to productive ends.

[3] https://rmarkdown.rstudio.com/
[4] http://beakernotebook.com/
[5] https://zeppelin.apache.org/
[6] https://www.wolfram.com/technologies/nb/

1.3.2 Vanilla software engineering

Another commonly used approach in medium or large teams is to treat a data science project just like a software engineering project. The data science process then plays out on top of a version control system (VCS) like `git` — modifications to source files, updates to exploratory notebooks, and data artifacts like model parameters are checked in to the repository. Developers can write unit tests (for non-data bound routines), provide code reviews for collaborators, and hopefully easily reuse code within the repository.

This structure allows the project to make use of the robust open-source tooling that has been developed in that field. These tools include version control systems like `git`, repository hosting providers like GitHub, code review tools, continuous integration testing, and so on. Also, under this workflow, contributors can submit patches to specific modules within a repository, allowing new functionality to be easily integrated into the larger project, rather than creating separate, redundant analyses.

However, tools such as `git` are famously difficult to use for many data scientists and statisticians, let alone for software engineers themselves [36]. The concept of a "commit" of source code is also not an ideal abstraction to use for data science, where data scientists have not typically composed their project out of small, self-contained blocks, and where data artifacts, like trained model parameters or images, are included alongside source code, which can often lead to merge conflicts. Furthermore, it is difficult to scale this approach beyond tens or hundreds of collaborators working on the same phase of the data science process as without additional structure on the project, redundant feature definitions or model specifications can be contributed side-by-side to the repository without being detected. Overall, the typical usability issues of `git` are compounded under this paradigm when scaling the number of collaborators.

1.3.3 Hosted platforms

Hosted data science platforms are applications that manage all aspects of data science product infrastructure, development, and deployment. In typical use, members

of a data science team log onto the application hosted on servers of a commercial entity, prepare and run analyses in notebooks, and share and deploy selected outputs. Organizations pay a monthly subscription fee for each user as well as a usage rate for compute and storage. As of this writing, companies such as Domino Data Lab[7], Dataiku[8], and Algorithmia [9] have developed hosted platforms in this increasingly competitive space. In a similar vein, academic entities have created hosted platforms, such as CodeOcean[10] for targeting reproducibility in scientific research or Pangeo [38] for enabling analysis of large geoscience datasets. These academic platforms are less likely to provide features for the end-to-end data science process. A comparison of selected platforms can be seen in Table 1.1.

Platform	Type	Capabilities			
		Code	Data	Compute	Integration
Dataiku	Commercial	✓	✓	✓	✗
Domino Data Lab	Commercial	✓	✓	✓	✗
Algorithmia Enterprise	Commercial	✓	✓	✓	✗
Pangeo	Academic	✓	✓	✓	✗
CodeOcean	Academic	✓	✓	✓	✗

Table 1.1: Comparison of selected hosted data science platforms. Each platform shown, both from commercial and academic developers, provides the ability to write, edit, and host source code; store data; and compute using cloud resources. The aims of these platforms vary — from enabling data scientists in companies to work more effectively (the commercial products), to providing a managed compute environment to support analysis of large atmospheric and oceanographic datasets (Pangeo), to enabling reproducibility of computational academic research (CodeOcean).

These platforms are easy to use in terms of providing helpful user interfaces and managing "pain points" such as data versioning, provenance, compute, deployment, and access control for collaborators. Various platforms have also experimented with novel collaborative mechanisms, such as bringing lay users into the platform in specific roles and enabling native discussion alongside notebooks and reports. In addition, due

[7] https://dominodatalab.com
[8] https://www.dataiku.com
[9] https://algorithmia.com/enterprise
[10] https://www.codeocean.com

to robust underlying infrastructure, these platforms can scale to support concurrent usage and discussion by large numbers of collaborators.

These platforms should not be thought of as solutions to the problem of integrability in large collaborations. They do not provide additional structure to the data science process to allow units of contribution from many collaborators to be integrated. Rather, they often encourage users to develop new analyses within notebooks or scripts, which can have the same problems described in Section 1.3.1.

Most importantly, as hosted, for-profit systems, they are not suitable in their current iterations for large-scale, open-source collaborations. For an open data science project with hundreds of collaborators, project organizers could expect subscription fees in the tens of thousands of dollars per year, in addition to the usage rates for cloud services. This is an untenable situation, especially when there is a risk that a new collaborator to the project would not just be using their subscription for personal use and free compute. Such open data science projects would also be handicapped by the non-free nature of the underlying platform. These problems will be discussed further in Section 6.1.

1.3.4 Data science competitions

Data science competitions such as the Netflix Prize[11], KDD Cup[12], and Kaggle[13] attract thousands of data scientists from around the world to work on interesting data science problems. In the competition model, a project sponsor provides a significant pool of money to reward the creators of winning solutions. Then, any data scientist can download a training dataset and project description and independently develop a predictive model. The competitors then download a test dataset that has had the target variable withheld and upload predictions in turn for each entity. The competition software automatically scores these submissions and records the results on a public leaderboard.

[11] https://www.netflixprize.com/
[12] http://www.kdd.org/kdd-cup
[13] https://www.kaggle.com

The competitive format yields results that push the state-of-the-art in terms of predictive modeling methodology. These competitions can also scale without limit as more and more data scientists enter the competition, develop models, and submit entries. Individual data scientists may find this format more or less usable depending on their strengths and comfort with their preferred toolset and compute environment.

At the end of a predefined period, the competitor who has achieved the best score on an unseen test set is crowned the winner and earns a monetary payout. What next? The project organizer is handed a dump of source code, notebooks, and model parameters, which may be in an arbitrary state of organization and usefulness. To integrate this model into production systems to the point of being able to serve predictions may require months more effort from data scientists and software engineers within the organization. Similarly, what happens to the $n - 1$ submissions that did not yield the best testing performance? They are more or less jettisoned, even though they may contain innovative ideas or approaches. Since these competitions impose no additional structure on the data science process besides providing the inputs, innovative approaches to data preprocessing, feature engineering, or modeling cannot be integrated with each other into one, larger, unified solution.

There are several other drawbacks of the data science competition approach to collaboration. First, the format may lead users to primarily tackle the machine learning algorithms themselves, instead of the features that go into them. Rather than focusing their collective brainpower on the difficult "bottleneck" step of feature engineering, users may engineer only a few features before attempting to tinker with different learning algorithms and their parameters, a potentially less efficient road to progress. Second, the pool of organizations that are willing to sponsor challenges is relatively small, because such sponsorship requires releasing proprietary data to the public. Third, because datasets are released in varying formats, users spend a significant amount of initial time wrangling and preprocessing the data, and these same tasks are redundantly performed by thousands of competitors over the course of the exercise.

Data science competitions have been highly successful at marshaling many data

scientists to each work to the same end goal, but provide little new or additional structure for them to work collaboratively.

1.3.5 Challenges of open data science

At the scale of open data science that we are considering, new challenges emerge to collaboration that do not previously appear in internal data science projects within smaller organizations. Some of these concerns will then be familiar to open-source software collaborations, and similarities and differences will be noted.

Redundancy. It becomes difficult, if not impossible, for individual contributors to keep track of the work being done by other collaborators. Redundancy in data science projects can be separated into at least three types. Take for example, redundancy that can arise in feature engineering (Section 1.2). The first is redundant code from one person duplicating work that had already been contributed to the project earlier. A second is redundant code from two people working on the same idea simultaneously without coordinating. And a third is feature functions that extract highly correlated feature values.

Now, these first two are redundancy related to the software aspect of data science, and as such, appear in software development projects as well. They are usually addressed as follows. A developer wants to fix a bug or implement a new software feature. They first consult a list of outstanding issues — if they find their issue on the list, they can assign themselves to work on it, or if someone else is already assigned, they can communicate with the over developer directly about joining forces. This does not work so well in the context of data science. Partly, this is because it has not been really tried, but largely because the units of contribution in a data science project — a data labeling function, a feature engineering function, or otherwise — may be too numerous and intricate to be enumerated as just described, and are parts of a creative process with no pre-defined end.

The third type of redundancy, a data-related redundancy, is unique to data science projects. That is, it is not known before the fact that two features will turn out to be highly correlated. If the two features are transformations of the same variable in the

raw data, then certainly they may be correlated. One can conceive of "assigning" raw variables or sets of variables to different collaborators to avoid this issue. But two different correlated raw variables, say `height` and `weight`, could lead to two correlated features as well. To address data-related redundancy, machine-driven automation must be leveraged.

Quality control. It becomes essential to be able to differentiate between high-quality contributions and low-quality contributions or noise, and to filter out the latter category of contributions without too much of a human burden to project maintainers. In OSS projects, automated tools can run tests and assess code quality using simple heuristics, giving project maintainers a first filter for quality. For data science, these filters are helpful, but the project maintainers must still evaluate the ideas underlying contributions like feature engineering functions on their own merits while discouraging noise.

Distributed computation. Systems must be engineered such that a substantial amount of computation can be distributed among the heterogeneous resources brought to the table by collaborators, while taking special care to avoid re-computation and to delicately manage data dependencies and data science artifacts. In OSS development, computation is not usually as scarce a resource given the demands.

Manipulation.: Finally, the project itself must not be vulnerable to manipulation or attack by adversaries including malicious contributors and irresponsible maintainers — or even accidental damage by well-meaning but naïve contributors. These concerns are shared in OSS development as well.

1.4 Data science abstractions

Inherent in this discussion is the idea that collaborative data science workflows need to be restructured such that units of source code contribution better align with fundamental pieces of a data science project. Traditionally, one commit in a software development project might include a new (software) feature along with documentation and test cases, all spread out over several files. In a data science project, on the

other hand, one unit of contribution could be the source code for one (machine learning) feature as added to a database comprising all features included in the model. By rethinking data science in terms of features, processing functions, labeling functions, prediction functions, model specifications, hyperparameter sets, and more, instead of lines of code or files, data science projects can be better organized, and automation strategies can more easily process this information into a coherent whole.

Of these abstractions, I will focus on tailoring innovations in collaborative data science to feature engineering. Feature engineering is a rather lengthy part of the data science process, in which practitioners ideate and write scripts to extract explanatory features from the data. These features are measurable quantities that describe each entity. For example, in an online learning platform such as a massive open online course[14] (MOOC), features that measure each student's progress include the amount of time she spends on the platform in a particular week, the number of visits she makes to the website, and the number of problems she attempts. The set of all features extracted by the data scientists can then be used as input to a predictive machine learning model. When attempting to solve a particular prediction problem, a small, easily-interpretable subset of features may prove to be highly predictive of the outcome in question. For example, the average pre-deadline submission time is highly predictive of a MOOC student's *stopout*, or withdrawal from the course [47]. To identify and extract these particular features requires human intuition, domain expertise, sound statistical understanding, and specialized programming ability. Successful practitioners contend that feature engineering is one of the most challenging aspects of a data science task, and that the features themselves often determine whether a learning algorithm performs well or poorly [11, 28].

Of the different units of contribution to a predictive modeling effort, I argue that feature engineering is the most amenable to a collaborative approach. Ideation by multiple people with different perspectives, intuitions, and experiences is yields a more diverse and predictive set of features. Then, implementing these features in code can be done in parallel, as long as the abstractions in place are well-structured

[14] http://mooc.org/

such that the features can be combined in a single framework.

Regardless of my focus on feature engineering, the methods I develop and describe in this thesis will apply more generally, to other parts of the data science process such as prediction engineering [42] and data programming [37]. Feature engineering will be cast as a particularly useful special case of a general procedure to combine units of source code that satisfy some fixed structure. These and more will be fit within a class of abstractions that apply to data science more generally.

The rest of this thesis is structured as follows. In Chapter 2, I present additional background and review related work. In Chapter 3, I present FeatureHub, a new approach to scaling data science through collaborative feature engineering, and describe my implementation of this platform as a scalable cloud application. Next, in Chapter 4, I provide a detailed assessment of this approach through user studies, quantitative evaluation, and more. In Chapter 5, I discuss the strengths and weaknesses of our proposed collaborative framework from a high level. Finally, in Chapter 6, within the context of my research, I discuss potential paths forward for collaborative data science and conclude.

Chapter 2

Background and Related Work

This thesis builds off of and relates to work in several areas, including crowdsourced data analysis, automated machine learning, and data science competitions. In this chapter, I review in detail background material in these areas.

2.1 Crowdsourcing methods in machine learning

Given the sheer volume of data that confronts data science and machine learning practitioners, crowdsourcing techniques that leverage human intelligence and labor are widely used in domains like image labeling and topic modeling.

2.1.1 Crowdsourced data labeling

More and more researchers have used the crowd to label datasets at scale, using crowdsourcing platforms like Amazon Mechanical Turk [10, 26]. These platforms are best suited for relatively unskilled workers performing microtasks, such as labeling a single image. Crowd workers can also be leveraged for more complex tasks via hybrid systems that use machine learning models to prepare or combine data for human workers. For example, [45] use crowd workers to continuously evaluate machine-generated classifications in a large-scale multiclass classification setting. In another example, [41] use crowd workers to count the number of objects in an image by

first segmenting the image into smaller frames using machine vision algorithms. [31] use crowd workers to estimate the nutritional content of images of meals using a framework for managing the crowdsourcing of complex tasks.

2.1.2 Crowdsourced data cleaning

The problem of dirty data has been an intense focus of researchers from various disciplines. Recently, some research has involved delegating this aspect of data management to systems using crowdsourcing. [8] propose a hybrid system to leverage the power of knowledge bases and crowdsourced labor to identify and repair incorrect data. [13] build a crowd-powered database query engine that leverages crowd workers for tasks including entity resolution.

2.1.3 Crowdsourced feature engineering

Crowd workers have been involved in feature engineering in the past, but to a limited extent. Most systems have used crowd workers either to assess the value of a feature on an example-by-example basis (which can be thought of as data labeling, as above) or to compose features using natural language or other interface elements. For example, [7] incorporate crowd workers in the data labeling and feature engineering steps when constructing hybrid human-machine classifiers. In their system, users label data and give their opinions on different feature definitions via a structured voting system. Feature values must be manually extracted from these feature definitions; thus, scalability and automation remain issues.

2.2 Automated methods in machine learning

Machine learning applications can be very complex and developing these applications can require highly specialized skills. Any machine-driven automation can make a substantial impact on the success of these projects. Automated methods have been introduced for all aspects of the modern data science process (Section 1.2). Of these,

I review some methods for machine learning and feature engineering.

2.2.1 Automated machine learning

Recent advances in hyperparameter optimization and open-source software packages have led to increased use of, and attention paid to, automated machine learning (*AutoML*) methods. [48] formalize *AutoML* problems as *combined algorithm selection and hyperparameter optimization*. [12] use Bayesian optimization techniques to automatically choose approaches to feature preprocessing and machine learning algorithms, and to select hyperparameters for these approaches. In a similar approach, [34] use genetic programming to optimize machine learning pipelines. These algorithms enable automated machine learning models to be generated with relatively little difficulty, performing well in various machine learning tasks. Importantly, they remove the time-consuming burden of tuning and evaluating models from the data scientists themselves. Unfortunately, these systems require a well-formed feature matrix as input, requiring feature engineering to transform unstructured or relational data, limiting their application as end-to-end solutions.

Other systems address the automated machine learning space from different angles. In [27], the authors develop a general purpose automated system for statistical inference called the "Automated Statistician. [3] use a technique called "bag of little bootstraps" to efficiently subsample data and enable hyperparameter search over more complex, end-to-end, data science pipelines with very large search spaces. In [51], the authors address the problem of neural network architecture search through a reinforcement learning-based solution, and are able to design novel architectures that achieve state-of-the-art results on some problems.

The success of these software libraries has motivated algorithm and systems development that targets more practical aspects of the modern data science process. [46] develop a distributed, multi-user automated machine learning system that can be used to tune models at scale. This is one of the first examples of approaches to *AutoML* that scals beyond the use case of a single data scientist working within their notebook, and is suitable as a component within a production machine learning

system.

2.2.2 Automated feature engineering

Feature engineering is a very broad field that applies to many different modalities. Chief of interest in this thesis is multi-table relational data which is commonly used in practical data science projects within organizations. In [18], the authors develop an algorithm for automatically extracting features from relational datasets. This algorithm, called *Deep Feature Synthesis*, enumerates a subset of the feature space through understanding of the dataset schema and the primary key-foreign key relationships. Other attempts at automated feature engineering include [25] which extends the *Deep Feature Synthesis* methodology and [24] which uses recurrent neural networks to learn a composition of functions. Alternately, in the databases community, much focus has been paid to relational data mining, often via *propositionalizing* relational databases into "attribute-value" form [20, 22].

There is a vast body of work on extracting features from other unstructured data such as images, audio, and text. Convolutional neural networks and auto-encoders are widely used to extract feature representations of image data, while techniques like `tf-idf` vectors, `word2vec` [30], and Latent Dirichlet Allocation [4] are commonly used to automatically extract features from text data, whether at the level of individual words or entire documents.

2.3 Programming and data science competitions

There are several existing platforms that present data science or machine learning problems as *competitions*. The most prominent of these, Kaggle, takes data science problems submitted by companies or organizations and presents them to the public as contests, with the winners receiving prizes provided by the problem sponsor. Competing data scientists can view problem statements and training datasets, and can develop and train models on their own machines or in cloud environments. They can also discuss the competition, comment on each others' code, and even modify

existing notebooks. The idea of gamifying machine learning competitions has led to increased interest in data science, and the Kaggle platform has lowered the barrier to entry for new practitioners.

Building off the success of Kaggle, a variety of other data science competition platforms have emerged, focusing on both commercial and social good projects. Driven Data[1] emulates Kaggle competitions but for projects like predicting whether water pumps in Tanzania are faulty, predicting outcomes from the United Nation's Millenium Development Goals, and predicting whether a blood donor will return for future blood donations. EvalAI[2] is an extensible, open-source platform that aims to help AI researchers, practitioners, and students to host, collaborate, and participate in AI challenges organized around the globe. Other similar platforms for data science and machine learning competitions include CodaLab[3], ChaLearn[4], and InnoCentive[5]. The Epidemic Prevention Initiative of the United States Centers for Disease Control administers a competitive epidemic prediction site[6] with competitions such as FluSight, in which participants forecast influenza incidence at various geographies. The *OpenML* platform [50] aims to build an open ecosystem for machine learning. Contributors can easily explore different datasets (presented as a single-table feature matrix), upload algorithms, and compare their results to others.

A variety of one-off data science competitions have also emerged. The US National Oceanic and Atmospheric Administration, in coordination with other US federal agencies, created a competition to try to predict incidence of dengue fever in San Juan, Puerto Rico and Iquitos, Peru [32]. Within the biomedical image analysis community, a series of "Grand Challenges"[7] produces independent competitions co-located with academic conferences, such as "CADDementia", a competition to diagnose dementia based on structural MRI data [5].

In another vein, researchers have focused on factors that drive innovation in pro-

[1] https://www.drivendata.org/
[2] https://evalai.cloudcv.org/
[3] https://competitions.codalab.org/
[4] https://www.chalearn.org
[5] https://www.innocentive.com
[6] https://predict.phiresearchlab.org/about
[7] https://grand-challenge.org/

gramming contests [14]. These factors include a competitive structure highlighted by leaderboards, real-time feedback, discussion, and open-source code.

2.4 Collaborative software development

Software has been developed in teams of various sizes since the dawn of computers. The open-source software (OSS) community, which thrives on highly collaborative development for many motivated developers, has seen its growth accelerate in recent years with the proliferation of online tools and communities. Some researchers have addressed the question of why developers contribute to open-source projects, when they are usually not compensated and the effort devoted can be significant. In [15], the authors survey Linux developers and find that identification with the project and available time are important factors for volume of contributions. [23] survey open-source developers and find that enjoyment-based intrinsic motivation, such as expression of creativity and intellectual stimulation, is the primary factor for contribution. Other important factors include the user's need for the software itself and the desire to improve his or her programming skills. [39] survey available evidence on motivations of open-source contributors, under the organizational psychology and economics lenses, and find that a large number of motivations underpins the contributions of a very heterogenous developer pool, including user needs and a reputation of generosity.

Though there is a large body of (somewhat inconclusive) evidence on OSS developers, there is little that is known about motivations for contributing to open data science projects. Analogy to software development suggests that the projects that may be most successful in attracting contributors are those that allow data scientists to express themselves creatively and to work on solutions that could affect them and their communities. As the open data science community continues to grow, more research will be needed on this subject.

Chapter 3

A collaborative data science platform

In this chapter, I present FeatureHub, an approach to collaborative data science based on collaborate feature engineering [44]. With the FeatureHub platform, multiple users log into a cloud platform to explore a predictive modeling problem, write source code for feature extraction, request an evaluation of their proposed features, and submit their work to a central location. The platform then aggregates features from multiple users, and automatically builds a machine learning model for the problem at hand. Through this process, collaborators can incrementally develop an integrated predictive model.

In Section 1.2, I presented a high-level overview of the modern data science process. There, I highlighted feature engineering as one of the most important steps in predictive modeling that is also particularly amenable to a collaborative approach. Despite the benefits of a collaborative model, no system previously existed to enable it as I have described. This desired system is distinguished by the ability to develop features in parallel across contributors in real-time and to integrate contributions into a single modeling solution, as introduced in Section 1.3. To build such a system, we must break down the data science process into well-defined steps, provide scaffolds and structure such that data scientists can focus on the feature engineering process, provide automated methods for everything else in the data science process, and provide the functionality to view, merge, evaluate, and deploy combined efforts.

FeatureHub is the first collaborative engineering platform in this spirit. In this

chapter, I first propose a new approach to collaborative data science efforts, in which a skilled crowd of data scientists focuses creative effort on writing code to perform feature engineering. Next, I architect and develop a cloud platform to instantiate this approach, during which I develop scaffolding that allows us to safely and robustly integrate heterogeneous source code into a single predictive machine learning model. In the next chapter, I will present experimental results that show that crowd-generated features and automatically trained predictive models can compete against expert data scientists.

In the following section, I motivate the FeatureHub workflow, before then describing in detail how both coordinators and feature creators interact with the platform. The workflow is divided into three phases: LAUNCH (Section 3.2), CREATE (Section 3.3) and COMBINE (Section 3.4). Coordinators execute LAUNCH and COMBINE, and feature creators interact with the platform in the CREATE phase. A stylized view of this workflow is shown in Figure 3-1.

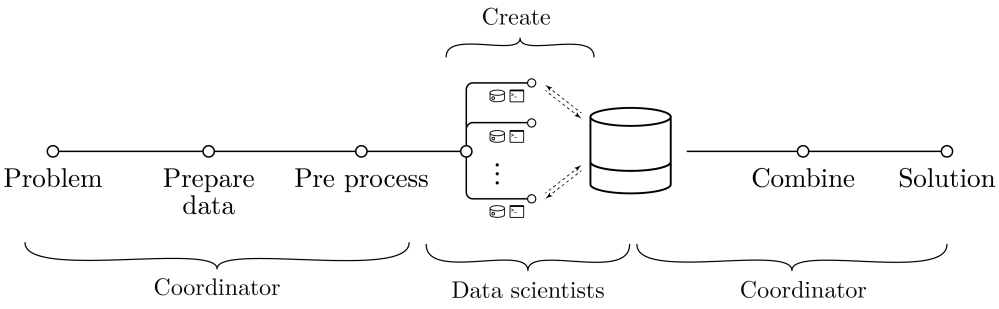


Figure 3-1: Overview of FeatureHub workflow. A coordinator receives the problem to be solved, and associated data. She prepares the data and does the necessary preprocessing using different available functions. Then, the platform is launched with the given problem and dataset. Data scientists log in, interact with the data, and write features. The feature scripts are stored in a database along with meta information. Finally, the coordinator automatically combines multiple features and generates a machine learning model solution.

3.1 Overview

To motivate the use of FeatureHub for collaborative data science, consider an example of predicting to which country users of the home rental site Airbnb will travel [1]. The problem includes background information on the prediction problem and a dataset, in a relational format, containing information on users, their interactions with the Airbnb website, and their potential destinations. Data scientists are asked to make predictions such that the five most highly ranked destinations of their model match the actual destination of each user as closely as possible. (This dataset is described in more detail in Section 4.1.) Under the approach facilitated by FeatureHub, the coordinator first prepares the prediction problem for feature engineering by taking several steps. They deploy a FeatureHub instance or uses an already-running system. They upload the dataset and problem metadata to the server, and then perform some minimal cleaning and preprocessing to prepare the dataset for use in feature engineering. This includes identifying the `users` table as containing the entities instances, moving the target `country_destination` column into a separate table, splitting the data into a training and validation set, and more.

The coordinator steps back and the data scientists log in. On the platform, they can read background information about the problem, load the dataset, and conduct exploratory data analysis and visualization. When they are familiar with the problem and the dataset, they begin writing features. One data scientist may use her intuition about what aspects of countries are most appealing to travelers, and write a set of features that encodes whether the user speaks the language spoken in different destinations. Another data scientist may look for patterns hidden deep within users' clickstream interactions with the Airbnb site, and write a feature that encodes the number of actions taken by a user in their most recent session. Once these users have written their features, FeatureHub automatically builds a simple machine learning model on training data using each feature, and reports important metrics to the data scientists in real-time. If the predictive performance of a feature meets expectations, the feature can be submitted and "registered" to a feature database, at which point

performance on the unseen test set is also assessed. Though these data scientists are looking for signal in different places, their work can be combined together easily within FeatureHub's scaffolding. They may be following their ideas in isolation, or using integrated collaboration and discussion tools to split up work and exchange ideas.

At this point in a typical data science workflow, data scientists working independently might have accumulated several features, and, having spent much time on preparing a working environment and cleaning data, would be anxious to test the performance of a model trained on their features. They might specify several machine learning models in order to get a sense of baseline performance and compute cross-validated metrics. As they continue ideating and doing feature engineering, they might take successively longer pauses to focus on model training and selection. However, using FeatureHub, individual workers can focus their creative efforts entirely on writing features while the system takes care of evaluating performance.

Meanwhile, the coordinator is monitoring the progress of workers. Each time they register a new feature, a model is selected and trained completely automatically and the performance is reported to the coordinator. After three days of feature engineering, the coordinator finds that the model has crossed a suitable threshold of performance for her business purpose and notifies the data scientists that the feature engineering has concluded. The data scientists can move on to a new task.

3.2 Launching a project

Let us return to the beginning of a FeatureHub project workflow. In the LAUNCH phase, a problem coordinator initializes a FeatureHub problem. The coordinator starts with a prediction problem that they want to solve, along with a dataset in relational format. Next, the coordinator performs preprocessing on the dataset to extract important metadata, including the problem type, the target error metric used to evaluate solutions, and the locations and names of data tables. The coordinator also has the option to pre-extract a set of basic features that can be used to initialize

the machine learning models. Often, these are features that require the most minimal and obvious transformations, such as one-hot encodings of categorical variables or conversions of string-encoded dates to timestamps. In fact, this entire step could be automated using existing frameworks such as Featuretools/Deep Feature Synthesis [18]. Finally, in order to orient workers, the coordinator prepares a description of the prediction problem, the data tables along with primary key-foreign key relationships, and details of the pre-extracted features.

This requires that all relevant tables are available from the outset. One might argue that a key contribution of a talented feature engineer may be to ingest and merge previously-unknown, external data sources that are relevant to the problem at hand. For our purposes, though, we include this in the data preprocessing step.

3.3 Creating new features

In the CREATE phase, data scientists log into a server using credentials provided by the coordinator. Their working environment is the Jupyter Notebook, a widely used interactive document that contains explanatory text, live code, and inline visualizations.

3.3.1 Load and explore

Notebooks for each problem contain detailed problem information provided by the coordinator, as in Section 3.2. In this self-contained environment, users have all of the information they need to understand the problem and engage with the available data. The environment also comes pre-loaded with all of the packages users require for data science and feature engineering, including the FeatureHub client library. After reading through the problem setup, users import a FeatureHub client, which provides a minimal but powerful set of methods for interacting with the feature engineering process. The full functionality exposed by the client is shown in Table 3.1.

Data scientists then load the data into their workspace by calling `get_sample_dataset`. This method returns the variables `dataset`, a mapping from

Phase	Method	Functionality
Launch (Coordinator)	`prepare_dataset`	Prepare the dataset and load it into the FeatureHub interface.
	`preextract_features`	Extract simple features.
	`setup`	Launch multiple machines in the cloud with JupyterHub installed and the dataset set up.
Create (Data Scientists)	`get_sample_dataset`	Load the dataset into the workspace.
	`evaluate`	Validate and evaluate a candidate feature locally, on training data.
	`discover_features`	Discover and filter features in the database that have been added by the user and other workers.
	`submit`	Validate and evaluate* a candidate feature remotely, on test data, and, if successful, submit it to the feature database.
Combine (Coordinator)	`extract_features`	Execute the feature extraction scripts submitted by the data scientists.
	`learn_model`	Learn a machine learning model using *AutoML*.

* Though both of these last two methods "evaluate" the performance of the candidate feature, they are named from the perspective of the data scientist's workflow.

Table 3.1: Set of methods for data scientists to interact with FeatureHub.

table names to tabular data objects, and `target`, a table of target values for each entity in the sample dataset. This data representation is a simple way to provide access to a relational dataset in Python and *pandas*, a tabular data processing library. Users can then explore the dataset using all of the familiar and powerful features of the Jupyter Notebook. The sequence of commands for an example problem is shown in Figure 3-2.

```
from featurehub.problems.demo import commands
dataset, target = commands.get_sample_dataset()
```

Figure 3-2: It is straightforward for users to use the FeatureHub client to import a sample dataset for an example problem called `demo`.

3.3.2 Write features

We ask workers to observe a basic scaffolding of their source code when they write new features, to allow us to standardize the process and vet candidate features. This setup is crucial in allowing us to safely and robustly combine source code of varying quality. In this scaffold, a feature maps the problem dataset to a single column of numbers, where one value is associated with each entity instance.

More formally, we are given a dataset composed of k tables

$$\mathcal{D} = \{T_i \mid i = 1, \ldots, k\}. \tag{3.1}$$

The first table T_1 is identified as the *entities table*, in that there is a one-to-one mapping between rows of the table and entities for which we will make predictions. We define the number of entities n in correspondence with the number of rows of the entities table. We also identify table $T_k \in \mathcal{R}^n$ as the *target table*, where the ith entry of T_k is the target for entity i.

We then define a *feature function*

$$f : \mathcal{P}(\mathcal{D}) \to \mathcal{R}^n \tag{3.2}$$

which maps any subset of the tables in the dataset to a vector of length n, the *feature values*.

Given a collection $\mathcal{F} = \{f_i \mid i = 1, \ldots, m\}$ of feature functions and a dataset $\mathcal{D}$, we extract a feature matrix $X_{\mathcal{F},\mathcal{D}}$ as

$$X_{\mathcal{F},\mathcal{D}} = \big(f_1(\mathcal{D}), \ldots, f_m(\mathcal{D})\big) \tag{3.3}$$

Our focus is on the feature functions $\mathcal{F}$ itself, rather than the feature matrix $X_{\mathcal{F},\mathcal{D}}$, which is what is otherwise usually taken as the starting point for a machine learning analysis.

At the outset, this definition seems to disallow categorical features or other "logical features" that consist of multiple columns. However, these can be represented simply as numerical encodings or encoded one column at a time, in the case of one-hot encodings of categorical features or lags of time series variables.

In this spirit, we require that a candidate feature be a Python function[1] that

- accepts a single input parameter, `dataset`, and
- returns a single column of non-null numerical values that contains as many rows as there are entity instances, that is ordered in the same way as the entity table, and that can be represented as a tabular data structure such as a `DataFrame`.

In order for the feature values to be reproducible, we also require that features not use variables, functions, or modules that are defined outside of the function scope, nor external resources located on the file system or elsewhere. This ensures that the source code defining the feature is sufficient to reproduce feature values by itself, such that it can be re-executed on unseen test data. We verify that a feature meets these requirements during the feature submission process. A trivial feature that fits this scaffold is shown in Listing 3.1.

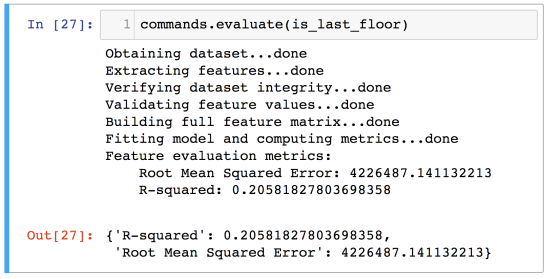
```
1 def hi_lo_age(dataset):
2     """Whether users are older than 30 years"""
3     from sklearn.preprocessing import binarize
4     threshold = 30
5     return binarize(dataset["users"]["age"]
6                     .values.reshape(-1,1), threshold)
```

Listing 3.1: A simple feature. The input parameter is a mapping (`dict`) of table names to `DataFrame`s. This function imports external packages within its body and returns a single column of values.

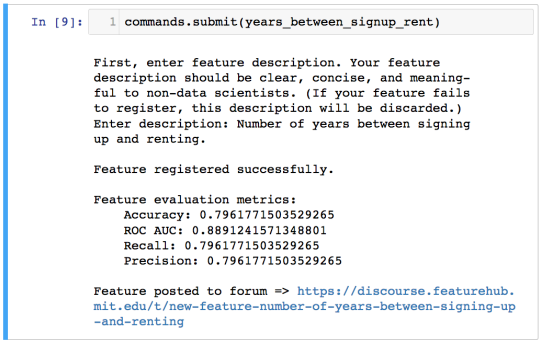
[1] We could easily extend this framework to allow the use of other languages commonly used in data science, such as R, Julia, or Scala.

3.3.3 Evaluate and submit

After the user has written a candidate feature, they can evaluate its validity and performance locally[2] on training data using the `evaluate` command. This procedure executes the candidate feature on the training dataset to extract feature values, and builds a feature matrix by concatenating these values with any pre-extracted features provided by the problem coordinator. A reference machine learning model is then fit on this feature matrix, and metrics of interest are computed via cross-validation and returned to the user. This procedure serves two purposes. First, it confirms to the user that the feature has no syntax errors and minimally satisfies the scaffolding. (As we will see, this by itself is not sufficient to ensure that the feature values are reproducible.) Second, it allows them to see how their feature has contributed to building a machine learning model. If the resulting metrics are not suitable, the data scientist can continue to develop and ideate. Example output of a feature evaluation is shown in Figure 3-3.



(a) `evaluate`



(b) `submit`

Figure 3-3: Example interactions with the `evaluate` and `submit` API methods. In evaluating a feature, the feature is executed in the user's notebook environment to extract feature values, which are validated against a list of criteria. If successful, feature performance metrics are returned. In submitting a feature, the user is prompted for a description, after which the feature source code is extracted and sent to the evaluation server. Upon success, the evaluation metrics are returned, as well as a link to an automatically-generated forum post (if applicable).

[2]Here, "local" is used to mean that evaluation occurs in the same executation context as the rest of the user's development, i.e. in their Jupyter Notebook kernel. This kernel, however, runs on our own servers, not the user's own device.

Once the user has confirmed the validity of their feature and is satisfied by its performance, they submit it to the feature evaluation server using the `submit` command. In this step, they are also asked to provide a *description* of the feature in natural language. This description serves several valuable purposes. It allows the feature to be easily labeled and categorized for in-notebook or forum viewing. It also facilitates a redundant layer of validation, allowing other data scientists or problem coordinators to verify that the code as written matches the idea that the data scientist had in mind. It may even be used to provide a measure of interpretability of the final model, because descriptions of the features included in the final model can be shown to domain experts or business analysts. An example feature source code with the accompanying natural language description is shown in Figure 3-4.

```
Average rent price for one room apartment (economy class)
at time of transaction.
```

Listing 3.2: Description

```
def one_room_rent(dataset):
    merged = pd.merge(
        dataset['transactions'],
        dataset['macro'][['timestamp', 'rent_price_1room_eco']],
        how='left',
        left_on='timestamp',
        right_on='timestamp'
    )
    rent = merged['rent_price_1room_eco']
    return rent
```

Listing 3.3: Source code

Figure 3-4: Natural language description of a feature function written in Python. These descriptions can be helpful for interpreting model performance and establishing trust in the implementation.

At the server, the same steps are repeated, with slight exceptions. For example, the machine learning model is fit using the entire training dataset, and metrics are computed on the test set. The fact that the feature is extracted in a separate environment with an isolated namespace and a different filesystem ensures that the resulting feature values can be reproduced for future use.

If the feature is confirmed to be valid, the feature is then both registered to the feature database and posted to a forum. The URL of the forum post is returned

to the user, so that with just one click, they can begin, view, or participate in a discussion around their feature. In this way, we try to make discussion as frictionless as possible. It would also be possible to integrate discussion into the FeatureHub site directly through Jupyter extensions.

3.3.4 Collaborate

We make a distinction between implicit and explicit collaboration in data science projects. *Implicit collaboration* occurs when contributions from different people are combined into a single, integrated model. *Explicit collaboration* is the ability to communicate directly with collaborators; view, discuss, and improve their work; and coordinate further development. Although the data scientists are already collaborating implicitly, in the sense that their features are combined into a single feature matrix, FeatureHub also aims to make this collaboration more explicit. We facilitate this through several approaches.

First, we provide an in-notebook API method, `discover_features`, that allows users to query the feature database for features that have been submitted by others, optionally filtering on search terms. This, for example, allows a user who is considering developing a feature for a particular attribute to see all features that mention this attribute. If there are close matches, the user could avoid repeating the work, and develop another feature instead. The user could also use the existing feature as a jumping-off point for a related or new idea. The matching feature source code is displayed to the user as well as its description and performance metrics, as shown in Figure 3-5. Though coarse, this output can be copied and adapted in other Notebook cells.

Second, we tightly integrate a Discourse-based forum. Discourse[3] is an open-source discussion platform that is often used as a forum for discussing software projects. Users are provided with a forum account by the coordinator. They can then post to several categories, including *help*, where they can ask and answer questions about technical usage of the platform, and *features*, where they can see the formatted fea-

[3] https://www.discourse.org

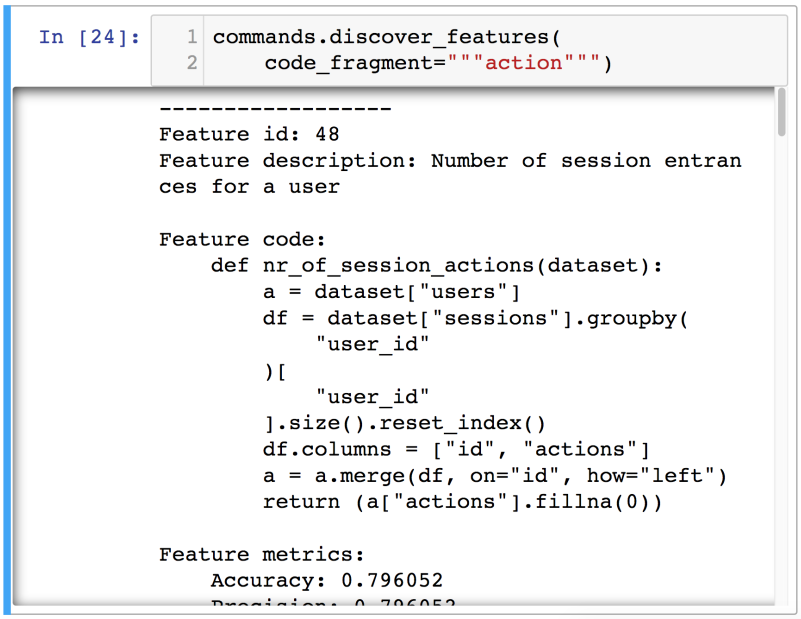


Figure 3-5: Output of a call to `discover_features` in the Jupyter Notebook. In this example, the user has filtered features to those containing the string `'action'` and then scrolled to inspect Feature 48. Scrolling further, the user will be able to see the performance metrics (accuracy, precision, recall, and ROC AUC) for this feature before continuing on to the other matching features in the database.

tures, automatically posted, that have been successfully submitted to the feature database. This provides an opportunity for users to discuss details of the feature engineering process, post ideas, get help from other users about how to set up a feature correctly, and use existing features as a jumping-off point for a related or new idea. An example forum post is shown in Figure 3-6.

While the specifics of this forum integration are relevant only for our current instantiation of FeatureHub, the overarching idea is to define the unit of discussion at the *feature* level, facilitating pointed feedback, exploration, and ideation — one feature at a time. This contrasts with the software engineering approach (Section 1.3.2), in which feedback is often presented at the commit or pull request level.

3.4 Combining contributions

During or after the feature engineering process, the coordinator can use FeatureHub to build a single machine learning model, using the feature values from every feature

Figure 3-6: A forum post for the *sberbank* prediction problem (Section 4.1.2) showing an actual feature submitted by a crowd worker. The feature description, source code, and computed metrics are shown, along with prompts for further discussion.

submitted thus far. These tasks can be executed by the coordinator multiple times, at each point assessing the combined work of the data scientists to date. To do so, the coordinator uses the following methods:

- `extract_features`: Feature source code is queried from the database and compiled into functions, which are executed on the training and test datasets to extract corresponding feature matrices.

- `learn_model`: A sophisticated automated machine learning framework, `auto-sklearn` [12], is used to build a single predictive model. The coordinator can modify *AutoML* hyperparameters, but the goal is to build the model with little intervention. Alternately, the coordinator can integrate a standalone automated machine learning system such as *ATM* [46], or even override this behavior and build models directly for finer control.

3.5 Design

The platform is designed around several priorities. First, it must support concurrent usage by dozens of data scientists, each of whom needs an isolated environment suitable for data science, along with a copy of the training data. Next, it must integrate heterogeneous source code contributions by different workers into a single machine learning model. The platform must be able to safely execute and validate untrusted source code as best as possible without leaking information about the test sample. Finally, it must enable detailed logging of users' interactions with the platform, to be used in further analysis of data scientists' workflows.

A schematic of the FeatureHub platform is shown in Figure 3-7. We leverage JupyterHub, a server that manages authentication and provisioning of Jupyter Notebook container environments [19]. Each individual user environment is pre-loaded with the most common data science packages, as well as FeatureHub-specific abstractions for data acquisition and feature evaluation and submission, as discussed in Section 3.3.

Careful consideration is given to ensuring that all code contributions are thoroughly validated, and that extracted feature values can be fully reproduced. To achieve this, a user first evaluates their feature on training data in the local environment. This ensures that the feature code has no syntax errors and minimally satisfies the scaffolding. However, it is not sufficient to ensure that the feature values are reproducible.

When the user attempts to submit a feature, FeatureHub both extracts the feature source code and serializes the Python function. Then, the function is deserialized by a remote evaluation service, which attempts to extract the feature values again, this time using the unseen test dataset as input. This reveals reproducibility errors such as the use of variables or libraries defined at the global scope, or auxiliary files or modules stored on the local file system. This service also has flexible post-evaluation hooks, which we use to integrate with a separate Discourse forum, but can also enable other custom behaviors.

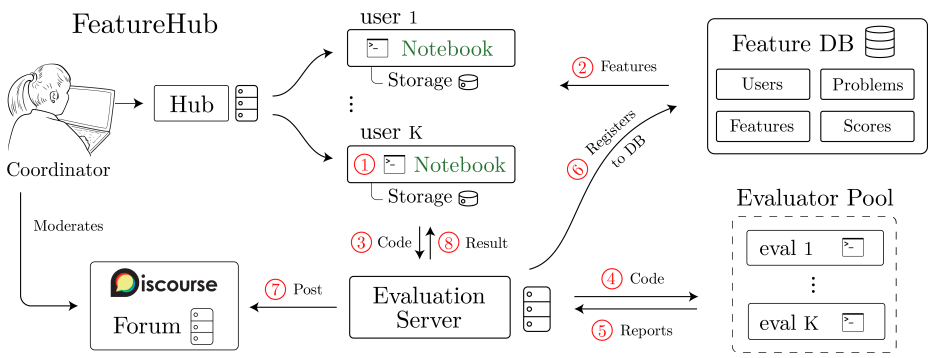


Figure 3-7: Overview of FeatureHub platform architecture, comprising the FeatureHub computing platform and the Discourse-based discussion platform. Users log into the platform to find a customized Jupyter Notebook environment with dedicated compute and storage resources (1). They can query the feature database to see the work that others have already done (2). Once they have an idea for a feature, the user implements it and evaluates it locally on training data, then submits their code to the feature evaluation server (3). The code is validated as satisfying some constraints to ensure that it can be integrated into the predictive model without errors. The corresponding feature is extracted and an automated machine learning module selects and trains a predictive model using the candidate features and other previously-registered features (4, 5). The results are registered to the database (6), posted to the discussion forum (7), and returned to the user (8).

Logging the users' interactions with FeatureHub allows us to more carefully analyze the efficacy of the platform and user performance. Indeed, this is a key advantage of developing and deploying our own system. We log user session events and detailed information each time the user attempts to evaluate or submit a feature in the database backend.

Finally, we design the platform with an aim towards easy deployment, so that it can more easily be used in classroom or intra-organizational environments.

Full details on the implementation of the platform are available in Appendix A.

Chapter 4

Assessment

I assess the FeatureHub platform along two main dimensions: the ability to develop powerful predictive machine learning models and the usability by collaborating data scientists.

To preview the results, we find that models built using FeatureHub and collaborating data scientists achieve performance that is competitive with the best models submitted by expert data scientists working for weeks on end on the data science competition platform Kaggle. We also find that functionality in our platform to enable explicit collaboration between data scientists substantially reduces the time to successfully submitting a first feature. One factor contributing to this effect was that some data scientists frequently used a functionality we developed to search for and discover features written by others, allowing them to see and build off existing successful submissions.

4.1 Experimental conditions

We conducted a user test to validate the FeatureHub concept and to compare the performance of a collaborative feature engineering model to that of independent data scientists who work through the entire data science pipeline. We also assessed to what extent the collaborative functionality built into the platform affects collaboration and performance.

4.1.1 Study participants

We recruited data scientists on Upwork[1], a popular freelancing platform. We advertised a position in feature engineering, filtering users that had at least basic experience in feature engineering using Python and *pandas*, a tabular data processing library. Unlike other crowdsourcing tasks, in which relatively unskilled workers execute "microtasks," the challenges posed by feature engineering require data scientists to have a minimum level of expertise. To allow data scientists with different experience levels to attempt the task, we hired each freelancer at their proposed rate in response to our job posting, up to a $45 per hour limit. Figure 4-1 shows the distribution of hourly rates for 41 hired data scientists. This exposes FeatureHub to an extreme test in enabling collaboration; as opposed to a small, in-house, data science team or a group of academic collaborators, our experimental participants live in many countries around the world, work in different time zones, and possess greatly varying communication ability, skill levels, and experience.

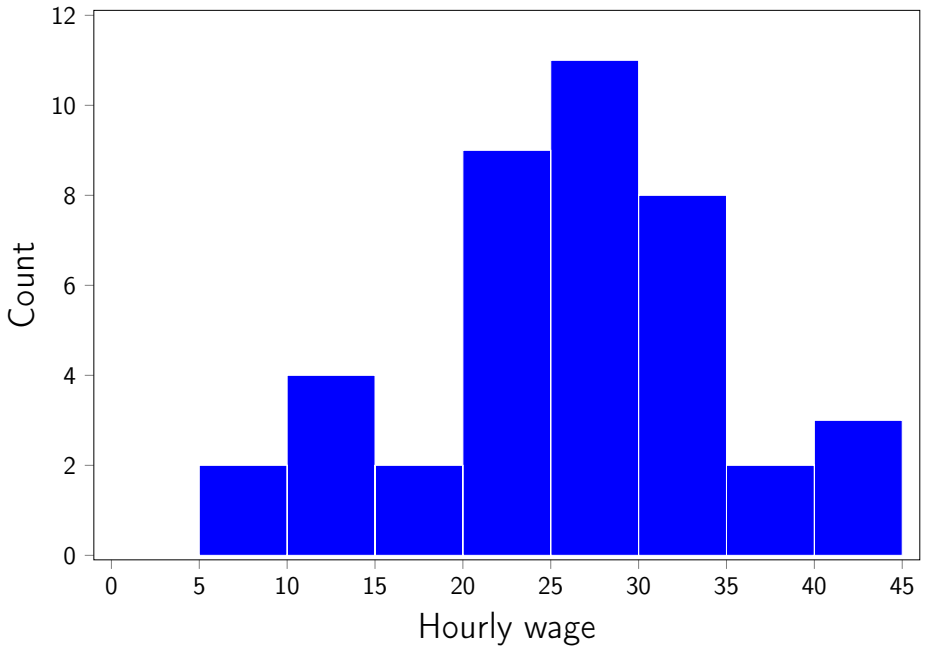


Figure 4-1: Hourly wages of data scientists from Upwork.

We asked data scientists to work a maximum of 5 hours on the platform. First,

[1] https://www.upwork.com

they were provided with a tutorial on platform usage and other documentation. Then, they were presented with background on various prediction problems. Next, they were tasked with writing feature engineering source code that would be helpful in predicting values of the target variable. Finally, we administered a post-experiment survey (Appendix C.1) and provided feedback on their performance[2].

4.1.2 Prediction problems

We presented data scientists with two prediction problems. In the first problem, data scientists were given data about users of the home rental site Airbnb and each user's activity on the site [1]. Data scientists were then tasked with predicting, for a given user, the country in which they would book their first rental. In the second problem, data scientists were given data provided by Sberbank, a state-owned Russian bank and financial services company, on apartment sale transactions and economic conditions in Russia [2]. Data scientists were then tasked with predicting, for a given transaction, the apartment's final selling price. Summaries of these prediction problems are shown in Table 4.1, while further details are available in Appendix B.

Problem Name	Type	Tables	Instances	
			Train	Test
airbnb	Classification	4	213451	62096
sberbank	Regression	2	30472	7662

Table 4.1: Prediction problems used in FeatureHub user testing.

4.1.3 Experiment groups

In order to assess the effects of different types of collaborative functionality on feature performance, we randomly split data scientists into three groups. The first group was provided access to the FeatureHub platform, but was not able to use the in-notebook

[2]The Upwork platform requires employers to provide feedback on freelancers after the conclusion of the work. As our study was an experiment and not an evaluation, we gave 5 star ratings to everyone who completed the experiment.

feature discovery method nor the forum for viewing and discussing submitted features. Thus, this group consisted of isolated data scientists who were *implicitly* collaborating in the sense that their features were combined into a single model, but otherwise had no coordination. A second group was provided access to and documentation for the `discover_features` client method as well as the integrated feature forum. A third group was provided access to the same functionality as the second group, and was also provided monetary incentives to avail themselves of this functionality.

4.1.4 Performance evaluation

Although we hired data scientists at an hourly rate, we also wanted to create mechanisms that would motivate both high-quality features and collaborative behavior. We sought to align incentives that would motivate an entire group towards a particular goal: achieving the best-performing model using the features written by all data scientists in that group. To that extent, we offered participants bonus payments based on their performance in two broad areas.

- *Collaborative behavior*: Contributing to forum discussions, building off code written by other participants, writing well-documented and reusable code, and writing clear, accurate, informative, and concise natural language descriptions of features.

- *Evaluation of features*: Feature ranking of final model and incremental models, and qualitative evaluation of complexity and usefulness of features.

Bonus payments were offered in terms of percentage points over the worker's base hourly rate, allowing us to ignore differences in base salary. We then created a bonus pool consisting of 25 percentage points per user. Thus, a user that did no work might receive no bonus, an average user might receive a bonus of 25% of their base salary, and an exceptional worker could receive a bonus of as much as $25n\%$ of their base salary, where n is the number of data scientists in their group. We then clearly defined, to each worker, what objective we were working toward, how we were

measuring their performance with respect to this objective, and what weight we gave to different performance categories and subcategories. The weights shown to each group can be seen in Table 4.2.

Group	1,2	3
Collaboration	28	50
... Feature descriptions	28	20
... Forum interactions	0	30
Evaluation of features	72	50
... Quantitative	57	40
... Qualitative	15	10

Table 4.2: Bonus payment weights (in %) for control and experimental groups. For groups 1 and 2, the "Collaboration" category was listed as "Documentation" to avoid bias, and the "Forum interactions" subcategory was not shown.

This was a first attempt to align financial incentives in collaborative feature engineering with the outcome of the overall predictive model. In general, this is a challenging problem for algorithm and market design reasons, as discussed further in Section 5.2.3.

4.2 Modeling performance

Collectively, the data scientists spent 171 hours on the platform. Of 41 people who logged into the platform, 32 successfully submitted at least one feature, comprising 150 of the hours worked. In total, we collected 1952 features. We also received 28 responses to our post-experiment survey to participants (Appendix C.2).

We limited data scientists to 5 hours on the platform, and those that submitted at least one feature used, on average, slightly less than that time. This constraint meant that a worker was allotted at most 2.5 hours per problem to read the problem description, familiarize themselves with the data, and write, test, and submit features. (Though we did not restrict data scientists from dedicating more effort to one of the problems). Before a data scientist could begin to ideate and write features, they needed to invest some time in learning the basics of the platform and under-

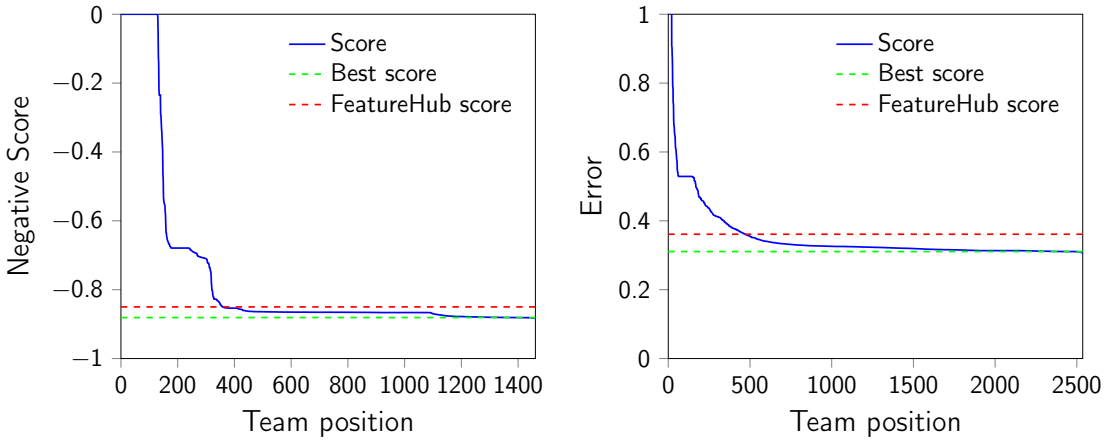
standing the specific prediction problem. According to our survey[3], though 21% of users reported beginning to work on a specific prediction problem within 20 minutes of logging in, another 39% reported working through tutorials and documentation for 40 minutes or more, restricting the time they spent on feature engineering directly. In a real-world setting, data scientists may spend days or weeks becoming familiar with the data they are modeling. Even so, we find that useful features can be created within minutes or hours.

During or after the feature engineering process, the project coordinator can combine source code contributions into a single predictive model, as described in Section 3.4. This should be accomplished with minimal intervention or manual modeling on the part of the coordinator, a key consideration reflected in the design. FeatureHub provides abstractions that allow the coordinator to automatically execute functions that extract feature values and use these as inputs to a machine learning model. Modeling can then be done either via an integrated wrapper around an automated machine learning library, or via manual model training, selection, and tuning.

Upon the conclusion of the experiment, we use this functionality to extract final feature matrices and model the feature matrix using our integrated *auto-sklearn*-based modeling. Using these completely automatically generated models, we make predictions for the unseen test sets on Kaggle. The results of our models, as well as those of other competitors, are shown in Figure 4-2.

Overall, our models achieve performance that is competitive with the best models submitted by expert data scientists working for weeks or months at a time. The scores we achieve, though not close to winning such competitions, place our automatically generated models within several hundredths of a point of the best scores: 0.03 and 0.05 for *airbnb* and *sberbank*, respectively. In both cases, the scores achieved by our models put them at an inflection point, in which an important amount of predictive capability has been unlocked, but the last few hundredths or thousandths or a point of performance on the target metric have not been met. To be sure, to a business, the value proposition of this difference, though small in magnitude, can be significant.

[3]A full summary of survey responses is shown in Appendix C.2.



(a) Performance on *airbnb*.

(b) Performance on *sberbank*.

Figure 4-2: Performance of FeatureHub integrated model as compared to independent data scientists. For *airbnb*, performance is measured as normalized discounted cumulative gain at $k = 5$. The negative of the scoring metric is reported on the y-axis to facilitate a "smaller-is-better" comparison. The automatically generated FeatureHub model ranked 1089 out of 1461 valid submissions, but was within 0.03 points of the best submission. For *sberbank*[4], performance is measured as root mean squared logarithmic error. The automatically generated FeatureHub model ranked 2067 out of 2536 valid submissions, but was within 0.05 points of the best submission.

Regardless, this exercise demonstrates that the combination of collaborating data scientists writing creative features and automated modeling can produce useful, "good-enough" results. If so desired, the coordinator can devote more resources to increasing the length of the CREATE phase, the automated machine learning modeling, or direct modeling, up to the point where they are satisfied with the model's performance.

Time to solution

If we are to compare data scientists working for weeks or months and FeatureHub collaborators working for 5 hours on two problems, it may not be surprising that the former group outperforms the latter. However, for many organizations or employers of data science solutions, an operative concern is the end-to-end turnaround time for producing a solution. Under the FeatureHub approach, data scientists can be requisitioned immediately and work in parallel, meaning that the parameters controlling

[4] *Sberbank* competition leaderboard was accessed on 2017-06-04.

time to model solution are the amount of time for each data scientist to work on feature engineering and the amount of time to run the automated modeling. In our experiment, given the 2.5 hour per problem limit and automated modeling over six hours, the potential turnaround time is less than one day.

On the other hand, in the independent or competitive approach (Section 1.3.4), each data scientist has little incentive to begin work immediately, and rather considers the total amount of time they anticipate working on the problem and the final submission deadline, which may be two to three months ahead. In Figure 4-3, we show the time from first submission to first recording a submission that beats the score of the FeatureHub automated model output. Of the participants who succeed in surpassing FeatureHub at all, 29% take two days or longer to produce these submissions. This, too, is an extremely conservative analysis, in that many competitors are working actively before recording a first submission. Finally, once a winning solution has been identified at the end of the entire competition period, the problem sponsor must wait for the winner's code and documentation to be submitted, up to two weeks later, and integrate the black box model into their own systems [1].

Diverse features

Given that freelance data scientists have a variety of backgrounds, skill levels, and intuitions, it is unsurprising that we observed significant variation in the number and quality of features submitted. We visualize the collected features in Figures 4-4 and 4-5. Data scientists were able to draft features that covered a wide subset of the feature space. By comparison, we also show features automatically generated by Deep Feature Synthesis [18]. In the two-dimensional projection, features generated by FeatureHub data scientists span the feature space.

One challenge was that some participants thought they could maximize the reward for their features by registering very similar features in a loop. Though individual features scored low, based on criteria in Table 4.2, we observed a single data scientist who still submitted an overwhelming $1,444$ features in our experiment. Automated modeling strategies that perform feature selection or use tree-based estimators are

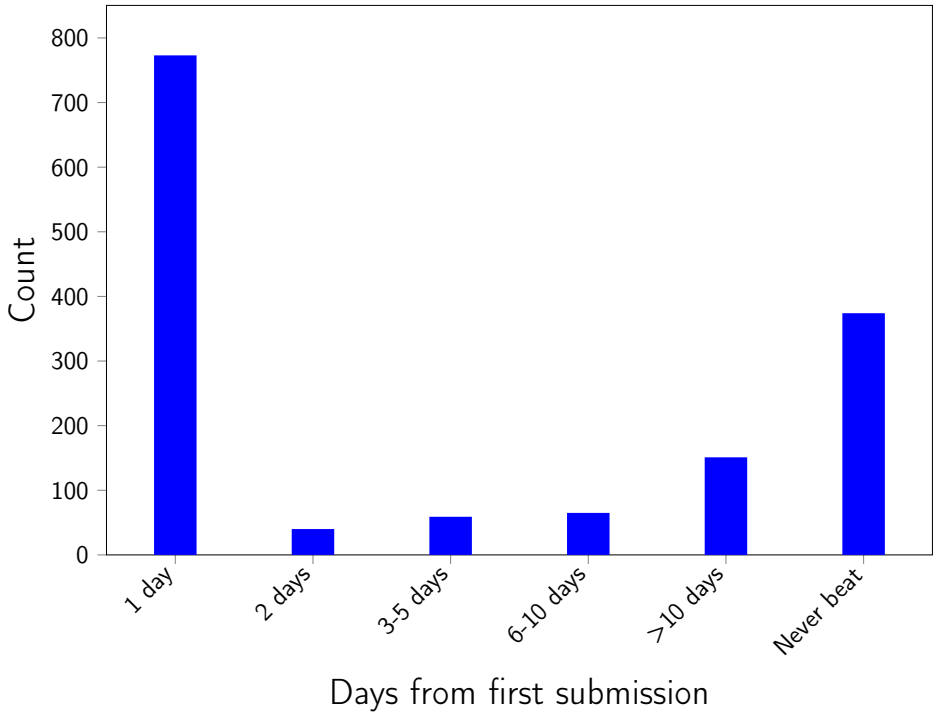


Figure 4-3: The amount of days, by Kaggle competitor, from their first submission to the problem to their first submission that achieved a better score than FeatureHub on *airbnb*. Since we only see the first submission, rather than when the competitor first starting working, this omits the effect of participants' preprocessing, feature engineering, and modeling work over as much as 2.5 months before initially submitting a solution.

able to discard less predictive features. Nevertheless, such behavior should be restricted as best as possible.

4.3 Evaluating collaboration

Facilitating collaboration among data scientists is a challenging prospect. Data scientists, who may not have much software engineering experience, often do not take advantage of existing approaches such as version control systems.

FeatureHub facilitates implicit collaboration among data scientists through the integration of submissions from all contributors. In this conception, the model of the group outperforms the model of any individual. We probe this suggestion by building models for each user separately based on the set of features that they submitted, and

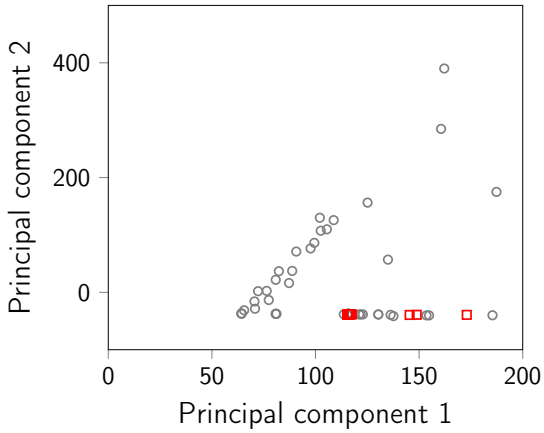


Figure 4-4: Features for *airbnb* projected on first two PCA components by FeatureHub (grey circles) and Deep Feature Synthesis (red squares).

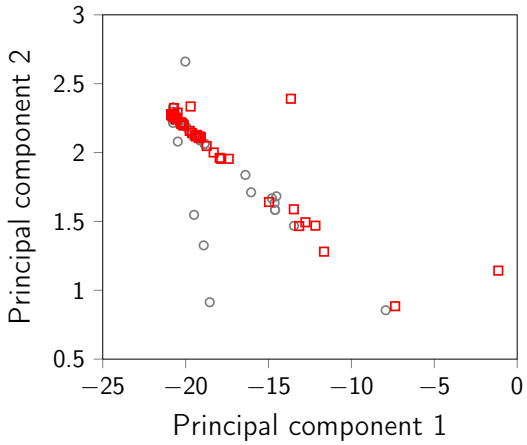


Figure 4-5: Features for *sberbank* projected on first two PCA components by FeatureHub (grey circles) and Deep Feature Synthesis (red squares).

comparing this to a single model built on the entire group's feature matrix. We find that in the case of the *sberbank* prediction problem, the group's model improved upon the best user's model by 0.06 points. To be sure, it is not surprising that adding more features to a model can improve its performance. However, this still shows that the amount of work that can be produced by individuals working in isolation is limited, in practice, compared to the scalability of the crowd.

As discussed in Sections 4.1.3 and 4.1.4, one approach to facilitating more explicit collaboration is through the implementation of in-notebook feature discovery methods, as well as through the integration of a discussion forum. Through our user study, we attempted to evaluate the impacts of providing the `discover_features` API

method and the integrated Discourse-base forum on user performance and user experience. Due to the substantial variation in individual worker quality and productivity, we were not able to robustly disentangle the quantitative effects of these mechanisms on feature performance. However, qualitative observations can be made. According to our survey, we found that including functionality for explicit collaboration reduced the mean time from first logging in to the platform to successfully submitting a first feature by about 45 minutes, as reported by participants. One factor contributing to this effect was that data scientists with access to the `discover_features` command reported using this functionality approximately 5 times each, on average, allowing them to see and build off existing successful submissions. Furthermore, this subset of data scientists reported that they found the integrated discussion most helpful for learning how to successfully submit features, through imitation or by others.

4.4 Analysis of contributors

One of the innovations of FeatureHub is that we not only collect the feature values submitted by collaborators but also the feature engineering source code that generates those values. Through the experiments detailed in this chapter, I collected a unique, detailed dataset consisting of the interactions of data scientists with the feature engineering process. Analysis of this dataset can reveal insights into users' behavior and the source code they wrote.

The study of workers on traditional crowdsourcing platforms like Amazon Mechanical Turk has received much attention in the literature [16] due to the importance of crowdsourcing in many human intelligence tasks. Unfortunately, the study of high-skilled workers — such as data scientists contributing to FeatureHub — has received less focus. In this section, I identify salient relationships in the data science code written by our users. I then propose future research directions for understanding type of work.

4.4.1 Worker attributes and output

When freelancers respond to project proposals on sites like Upwork, they present their education, experience, and qualifications. This gives us the opportunity to investigate the effect of some of these worker attributes on feature engineering output within our platform. In this section, I investigate the relationship between data scientists' proposed wages and stated experience on their feature engineering performance.

In our user experiment, potential data scientists were asked to submit a bid with an proposed wage, and we then accepted all minimally qualified data scientists that bid under a threshold. The relationship between stated wage and feature performance is shown in Figure 4-6. Here, performance is measured by the score of the best feature submitted to each problem (using R^2 for *sberbank* and ROC AUC for *airbnb*). There is a small, but statistically significant, positive weight on `wage` ($\beta = 0.0008,\ p = 0.025,\ n = 17$ and $\beta = 1.8e-5,\ p = 0.018,\ n = 27$). One should note that there are many potentially confounding variables in this analysis, such as that data scientsits who live in developed countries with higher purchasing power are likely to both charge more (in USD terms) and speak better English.

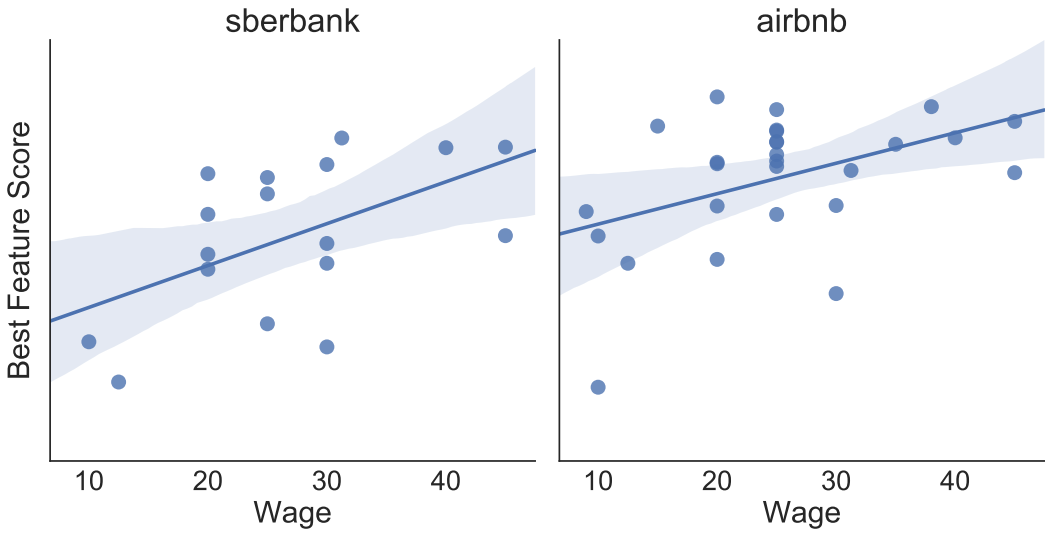


Figure 4-6: Relationship between hourly wage and feature performance, including linear trend and 95% confidence interval. For the *sberbank* problem, performance is measured by the maximum R^2 of any feature submitted ($\beta = 0.0008,\ p = 0.025$). For the *airbnb* problem, performance is measured by the maximum ROC AUC of any feature submitted ($\beta = 1.8e-5,\ p = 0.018$). Y-axis scales are not comparable.

Alternately, we can try to tease out the effect of data science experience on feature engineering performance. I categorize each participant into low, medium, and high levels of experience by parsing their self-reported descriptions of Python/*pandas* ability and data science experience, as well as their cover letter and freelancer overview blurb. My heuristic categorization focuses on self-reported markers such as years of data science experience and scope of past projects. The relationship between experience and feature performance for both problems is shown in Figure 4-7. Based on this categorization, there is no statistically significant difference of the groups across experience levels for either *sberbank* or *airbnb* (one-way Kruskal-Wallis $H = 3.08$, $p = 0.21$, $n = 17$ and $H = 0.38$, $p = 0.83$, $n = 27$).

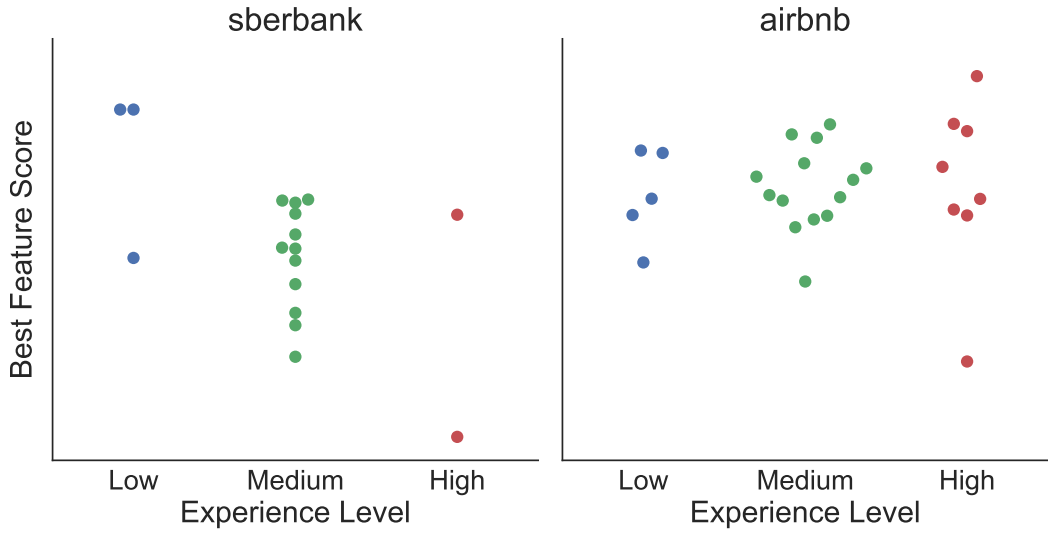


Figure 4-7: Relationship between data scientist experience and feature performance. For the *sberbank* problem, performance is measured by the maximum R^2 of any feature submitted. For the *airbnb* problem, performance is measured by the maximum ROC AUC of any feature submitted. Y-axis scales are not comparable.

The small or nonexistent effect of wage and experience can be viewed as encouraging. This suggests feature engineering can be effectively scaled to many collaborators, few of whom need to be highly paid experts, but rather sufficiently experienced contributors with nuggets of insight. Alternately, we could interpret this to mean that the structure imposed on the feature engineering process by FeatureHub is too limiting for experts to show the extent of their skills — or that the limited scope of our user experiment did not allow experts enough time to shine. Further experiments

could disentangle these explanations.

4.4.2 Contributions in source code

In Section 4.2, I investigate the quality of feature engineering source code written by freelance data scientists from the perspective of the overall modeling performance. We can also evaluate the source code on its own merits. What can we learn about the contributions of our data scientists?

A first major contribution is the domain knowledge incorporated into features by our data scientists. For example, one data scientist had the insight that the number of distinct actions that is taken by a user of *airbnb* suggests higher engagement with the website and an interest in a "broader range" of destinations. Another data scientist working on *sberbank* had the insight that, in "Soviet" apartments, a larger kitchen relative to overall living area is desirable, and wrote a feature to that extent. Still another encoded the distance from desirable shopping areas in Moscow. Many more insights like these reveal the power of a diverse set of collaborators, each with different experiences and perspectives.

A second major contribution is the development of very complex features that attempt to capture non-trivial patterns in the data. For example, one data scientist wrote a feature for *airbnb* to perform affinity propagation clustering of similar users using principal components extracted from a subset of important raw variables. Another data scientist tried encoding *airbnb* user session device types using a *tf-idf* vectorizer, rather than just treating them like normal categorical variables.

Another contribution was sophisticated processing of dirty data. Our contributors used contextual knowledge to impute missing values by within-group means or intelligently incorporate lag information.

4.4.3 Educational opportunities

To implement data science ideas correctly and efficiently is a challenge that requires strong programming skills and statistical know-how. Ultimately, feature engineering

is as much a problem solving endeavor as much as it is programming one. In our feature corpus, a variety of *anti-patterns*, or techniques that are frequently-used but considered improper practice, are evident that merit further discussion. I view these anti-patterns as both educational opportunities and directions for improvement of the FeatureHub platform. By studying these anti-patterns, we can identify opportunities for targeted guidance to alleviate the challenge of bringing innovative feature ideas to the screen. Some of these motivate extensions to the feature engineering abstractions presented in FeatureHub, as will be discussed in Section 5.1.1.

Features written exhibited a wide range of anti-patterns, from inefficient code to improper data science techniques. None of the features demonstrate *errors*, in the sense of bad syntax or API nonconformance, but many may do something *wrong* nonetheless. The following paragraphs outline some of the main anti-patterns that emerge.

Inefficient code. Often times, data scientists wrote *pandas* code for manipulating `DataFrame`s that was very inefficient for the task. This does not pose a problem for small datasets, but as the size of the analysis scales, small inefficiencies begin to add up and can make it difficult for the underlying *AutoML* to provide real-time feedback.

- *Premature denormalization.* Some data scientists did not seem accustomed to working with relational, multi-table datasets and began every feature by denormalizing the entire dataset into one large table. In some cases, only a subset of the tables (such as exactly one table) was needed to compute the proposed feature, so the expensive joins were included needlessly.

- *Iteration.* Another common anti-pattern is to operate on tabular data through iteration. This consists of iterating over entities in `DataFrame`s using Python `range`s, *pandas* `iterrows`, or some other method. In Python code, these iteration methods can be substantially less efficient than native C implementations that underlie "vectorized" methods. An instance of this problem, as well as in improved implementation of the same idea, is shown in Listings 4.1 and 4.2.

- *Reinventing the wheel.* Some data scientists were not as familiar with *pandas*

library functions for common tasks, and reimplemented them from scratch instead. These reimplementations come at a sizable cost in terms of programmer time, computational efficiency, and recognizability by others. One example of this problem, as well as an improvement using the appropriate library function, is shown in Listings 4.3 and 4.4.

```
def far_from_metro_station(dataset):
    distance = []
    mean = dataset['transactions']['metro_km_walk'].mean()
    for i in range(0, len(dataset['transactions'])):
        if dataset['transactions']['metro_km_walk'][i] > mean:
            distance.append(1)
        else:
            distance.append(0)
    return distance
```

Listing 4.1: Original

```
def far_from_metro_station(dataset):
    mean = dataset['transactions']['metro_km_walk'].mean()
    result = dataset['transactions']['metro_km_walk'] > mean
    return result.astype(int)
```

Listing 4.2: Improved

Figure 4-8: Anti-pattern of iteration on `DataFrame`s. The original submission, adapted for clarity, uses a Python `range` to iterate over rows directly. An improved version instead leverages highly-optimized vectorized code under the hood to achieve performance and readability benefits.

Mutated data. Feature engineering best practices suggest to avoid mutating `DataFrames` passed into a function, as such a function might then corrupt a larger data processing pipeline. Many features written exhibited this pattern, which justified precautions taken in the FeatureHub evaluation and modeling modules to ensure data integrity and reload data that had been modified by submitted features during execution. Unfortunately, knowing when operations on `DataFrame`s will create a copy or mutate the original data requires in-depth knowledge of the *pandas* data model and implementation. In general, programmers avoid this issue by using the `DataFrame.copy` instance method to ensure that columns they have initially accessed will be copied before invoking further operations that might mutate the underlying data.

```
def percent_kitch_sq(dataset):
    pct_kitch_area = (
        dataset['transactions']['kitch_sq']
        / dataset['transactions']['full_sq']
        * 100
    )

    def cleankitch_sq(a):
        import math
        if math.isnan(a):
            return 0
        else:
            return a

    pct_kitch_area = pct_kitch_area.apply(
        lambda a: cleankitch_sq(a)
    )
    return pct_kitch_area
```

Listing 4.3: Original

```
def percent_kitch_sq(dataset):
    pct_kitch_area = (
        dataset['transactions']['kitch_sq']
        / dataset['transactions']['full_sq']
        * 100
    ).fillna(0)
    return pct_kitch_area
```

Listing 4.4: Improved

Figure 4-9: Anti-pattern of reinventing the wheel in terms of *pandas* library functions. The original submission, adapted for clarity, implements a procedure to fill missing values with zeros, even though the library method `fillna` is available and widely used for the same task. (The original source code also demonstrates more general programming anti-patterns, such as the use of `.apply(lambda a: cleankitch_sq(a))` instead of `.apply(cleankitch_sq)`.)

Inappropriate treatment of categorical variables. Handling categorical variables is one of the most common tasks for a data scientist during feature engineering. These discrete variables can contain important information but are usually not suitable for direct inclusion in a machine learning model. For a categorical variable with few distinct levels, one of the most common approaches is to represent it using a one-hot encoding. Unless there is a natural "ordering" of categories (an *ordinal variable* like "quality" with levels like "poor", "fair", "good"), it rarely makes sense to represent this type of variable using a single numerical column[5].

[5]That is, one should not encode a categorical variable "color" with levels like "blue", "green," and "yellow," as a single column with the numbers 1, 2, and 3, because there is no natural ordering between the colors.

- *Inappropriate encoding of categorical variables.* A fairly common anti-pattern in the corpus is representing categorical variables in a single numeric column. This is generally a poor modeling choice. One example of of this problem is shown in Listing 4.5.

- *Inappropriate interaction of categorizations.* An astute data scientist may find that one categorical variable correlates differently with the target when "cut by" (interacted with) another categorical variable to create a compound category. This can become an anti-pattern if there is no realistic expectation that the categorization has any semantic meaning. For example, there are 19 features in our corpus that represent the one-hot encoding of the interaction of the "Season" and "Time of Day" of the Airbnb user's first booking (e.g. `AutumnMorning`, `WinterNight`). While either of these categories (derived from the timestamp of the first booking) can be useful on their own, there is no reason to suspect that there is any *interaction* between them.

```
# ...
users = dataset['users']
users = users.set_value(
    users[
        (users['age'] > 40)
        & (users['affiliate_channel'] == 'sem-brand')
    ].index,
    'age_channel',
    1,
)
users = users.set_value(
    users[
        (users['age'] > 40)
        & (users['affiliate_channel'] == 'sem-non-brand')
    ].index,
    'age_channel',
    2,
)
# ...
```

Listing 4.5: Anti-pattern of encoding categorical variables as single numeric feature. This selection of the original submission shows the data scientist is representing the type of affiliate channel that the Airbnb user signed up with as the values $1, 2, 3, \ldots$. In this case, they are also interacting this categorical variable with the user's age.

Processing dates and times. One of the most common non-numeric data types is temporal data. Dates and times are commonly represented in raw data (or parsed by import routines) as strings, and it is the job of the data scientist to first find an

appropriate representation and then do further feature engineering. In our corpus, one anti-pattern was improper processing of datetime data. Some examples are shown in Figure 4-10. It was common to see features that convert temporal data to strings and then operate on the strings directly, when library functions are much more efficient, robust, and readily available. As one possible education opportunity, a system could attempt to infer temporal variables that were not previously annotated as such, and then propose native datetime manipulation methods in place of string- or integer-based manipulation.

```
yearmonth = dataset['transactions']['timestamp'].apply(
    lambda x: int(x[:4] + x[5:7])
)
```

```
hour = dataset['users']['timestamp_first_active'].apply(
    lambda a: str(a)[8:-4]
)
```

```
hour = dataset['users']['timestamp_first_active'].apply(
    lambda v: (v % 1000000) // 10000
)
```

Figure 4-10: A frequent task for data scientists is to process datetime variables, which is often done by converting the variable to a native datetime data type and then using robust library methods. In each listing, a snippet of a submitted feature instead shows the processing of a timestamp using string methods or arithmetic directly. While these approaches "work," they should be discouraged. (Example raw timestamps in the data for the `timestamp` and `timestamp_first_active` columns take the form `"2013-05-27"` and `20120519182207`, respectively.)

Missing values. Any real-world data science analysis must confront dirty data, especially the presence missing values. Though some of this cleaning may occur in a standalone data preprocessing phase, it is often deeply interconnected with the feature engineering phase. Our data scientists paid careful attention to missing values, as any features with missing values would be rejected by the evaluation system. However, we observe a variety of anti-patterns related to treatment of these missing values. To alleviate these issues, we could allow data scientists to expose features that still have missing values, deferring the cleaning of missing values as a separate phase, or incorporate automated methods that fill missing values with sensible methods for the specific column.

- *Sentinel values.* One approach to handle missing values is to replace them with some constant sentinel value. This is a perfectly valid approach to cleaning categorical variables, as the sentinel values then represent a new "category," as long as the categorical variables are dealt with further. Otherwise, sentinel values can cause issues for linear estimators and many others. One example of this anti-pattern is shown in Listing 4.6. Missing values for "year" are replaced with 0, even though the year of the first transaction in the data is 2011. This may be a "doubly-bad" anti-pattern because it more likely conceals data quality or parsing issues than true missingness. In our corpus, many feature definitions simply ended with a call to the *pandas* function `fillna`, filling missing values with a parameter 0, -1, -999, or otherwise. It is probably gratuitous to think of these cases as the purposeful use of sentinel values; instead, this may be more of a desire to get rid of the missing values as quickly as possible, even if not done in a rigorous manner.

- *Naïve strategies.* Based on the context, there may be very reasonable, simple strategies for imputing missing values, such as imputing the mean for population measurements, the mode for categorical values, and the first lag or a rolling mean for time series features. The lack of sophistication in treating missing values in many cases in our corpus can be thought of as an anti-pattern. For example, one feature filled missing values in a feature representing the average rental price of a one-bedroom apartment at 0. Better options include the global mean or the mean of the subset of transactions from the same month or approximate location.

```
def year_from_timestamp(dataset):
    import pandas as pd

    year_from_timestamp = pd.to_datetime(
        dataset['transactions']['timestamp']
    ).dt.year.fillna(
        0
    )
    return year_from_timestamp
```

Listing 4.6: Simple feature demonstrating the anti-pattern of filling nulls with sentinel values (*sberbank* problem).

Rescaling. Rescaling feature values, via centering, standardizing, or taking some non-linear transformation, can be an important step in several cases, such as when using a nearest neighbor estimator (e.g. *kNN*) or when the conjectured relationship between the variable and the target is non-linear. One important contribution of a data scientist is to choose such transformations carefully. In our corpus, one anti-pattern was the inappropriate use of these rescaling transformations, such as taking the logarithm of an age feature, as this results in a highly-skewed distribution. To be fair, one weakness of the FeatureHub approach is that there is no ability for data scientists to target specific transformations to different estimators, as the automated machine learning is performed without their control.

Unexplained behaviors. Finally, there were some practices in the corpus of features we collected that may be *wrong*, but would never raise errors in any software testing suite. One feature tried to weight certain variables by the maximum of another variable in the same group, but the maximum over the entire dataset, rather than the maximum of the subgroup, was incorrectly applied. Another computed a meaningless interaction between the logarithm of users' ages and the period of the day that the users took some action. Still another wrote 35 lines of code to count the number of *airbnb* website compound events of type `-unknown-_I_calendar_tab_inner2_I_-unknown-`, an event type that would be better-off ignored. When scaling collaboration, more and more curious or unexplained behaviors like this emerge. Algorithms and systems need to be awar of this kind of behavior and rigorously use feature selection methods to filter out such features.

Many features that displayed the anti-patterns presented in this section could be improved by their authors in a very educational revision process. Further research could focus on the automated detection of these data science anti-patterns and provide real-time interventions.

Going through hoops

In analyzing the corpus of features submitted by FeatureHub collaborators, in some cases, what first appear to be anti-patterns actually represent concerted efforts to

evade restrictions imposed by the platform. In these cases, where collaborators "go through hoops" to make their features comply with the structure imposed by FeatureHub, it reveals weaknesses with the platform itself that should be considered in future development.

Low-hanging fruit. Many FeatureHub collaborators submitted excessively simple features. These include passing through a single column from the raw data without little or no modification and taking the sum, difference, or ratio of two related columns. We had several strategies to discourage these relatively unproductive contributions: pre-extracting basic features (Section 3.2) and providing real-time feedback on feature performance. The evidence from the feature corpus suggests that further interventions, such as additional tutorial materials and more aggressively utilizing automation techniques like *Deep Feature Synthesis* [18], are needed to reduce this behavior.

Duplicated joins. More involved joins of different data tables were re-implemented by different users. For example, two users separately implemented a fuzzy join of columns in two tables that encoded the name of a spoken language using different values. If the two FeatureHub contributors could have collaborated at this intermediate level, they could have eliminated redundant work and combined the best of their two approaches.

Inconsistent cleaning. Multiple FeatureHub collaborators cleaned extreme values by detecting values above some threshold, but were inconsistent in the thresholds they used, even for the same column. For consistency within a project, FeatureHub could support exposing these preprocessing steps which could be reused by others.

Multiple columns of output. In FeatureHub, we restrict the output of features to be a single column of values. In Section 3.3.2, I discuss why this is not an unreasonable restriction. In some cases, however, our contributors wanted to produce a multi-column encoding of a categorical variable. To do this, they duplicated tens of lines of code and iteratively selected out a different column of the logical, multi-column feature they had created. Further effort can be paid to finding the balance between allowing submissions of multiple columns of logically related feature values and dissuading

monolithic contributions and spam.

Feature tagging. Patterns of similar features emerged from our corpus, such as "user engagement" features for the *airbnb* problem (e.g. unique action types) or "apartment characteristics" features for the *sberbank* problem (e.g. relative size of kitchen). Clustering or collecting similar features during the development process would provide more avenues for collaboration. Unfortunately, FeatureHub does not allow users to provide tags, keywords, or any other annotations besides a high-level description.

Chapter 5

Discussion

Having designed and realized a new collaborative data science platform, I have had a chance to reflect on opportunities for improvement and challenges of the approach. FeatureHub, as a research prototype, can be extended and improved along many dimensions in order to unlock the full potential of collaborative data science.

5.1 Opportunities

5.1.1 Flexible data science abstractions

In this thesis, I introduced a simple structure for feature engineering that allows heterogeneous source code contributions to be integrated successfully. Recall from Section 3.3 that the feature engineering structure imposed is a single Python function that accepts as input a mapping of table names to `DataFrame`s and produces as output a column of feature values. However, this abstraction may not be sufficiently flexible or powerful for all situations.

- *Separate fit and transform phases.* In the existing feature abstraction, there is no concept of being able to "fit" a feature on the training set and then apply it to make new predictions in evaluation or after deployment. For example, we would like to center a variable by computing the mean on the training set values only, to avoid leakage. The fit-transform paradigm popularized in *scikit-learn*

[35] is one potential approach.

- *Ease database-style joins.* The input data to the feature function is generally relational, in the problems we focus on, but the input data structure is unaware of relationships between tables. Collaborators must possess an intermediate level of *pandas* expertise in order to cleanly join tables together. Higher-level programming facilities that enable more declarative processing could help this issue.

- *Fetch external data sources.* The integration of external data sources into a feature engineering pipeline is one valuable contribution that data scientists can make. For example, a data scientist using a retail dataset from a German department store may want to identify whether each day was a national holiday. To do so, she would need to find an external data source identifying German holidays and join it with the current project, either by identifying some external API or library or storing a calendar dataset within the project. We could expose the ability to integrate external data sources directly, and provide assistance with storing and querying the new data.

- *Become cleaning-aware.* We present data preprocessing and cleaning as a separate step in the data science process that is completed before the feature engineering step begins. However, feature engineering and data cleaning go hand-in-hand in practice (Figure 1-1). For example, in the process of encoding some categorical variables during feature engineering, the data scientist may discover that some categories are duplicates of others and desires to merge the two categories. Under our framework, they would need to re-do this cleaning within every feature that references the dirty column. Ideally, upon discovering new data cleaning operations, the data scientist could add them to the cleaning step and make the cleaned data available to all collaborators.

- *Transform intermediate results.* The previous point underscores a larger problem in which we want to be able to reuse intermediate results for further com-

putations. Imagine that a first data scientist writes a feature to compute an entity's age at each observation by subtracting the timestamp of the observation with the entity's date of birth. A second data scientist may want to create new features that further transform this age feature, say raising it to some power or binarizing it at some threshold. This can be most easily accomplished by taking the age feature produced by the first data scientist as input, rather than the entire dataset. FeatureHub does not allow this sort of composition; the second data scientist would have to recreate the age feature within every derived feature that they write. Indeed, one can conceive of feature engineering more generally as the creation of a dynamic, acyclic graph (DAG) of data-flow. In this conception, edges represent the flow of data and nodes represent the application of transformations, as shown in Figure 5-1. We would then like to allow collaborators to identify the specific inputs to and outputs from their features, incrementally assembling a data-flow graph in the process. Each node produces feature output that can be sent directly into the feature matrix, consumed as input to another node, or both. Feature selection would be performed over all features proposed for inclusion in the meature matrix. Feature selection would then be performed over all nodes except the root nodes. A graphical user interface could allow data scientists to visualize the feature engineering graph and position a new feature at a specific location.

These suggestions touch on feature engineering alone, but similar concerns apply to other aspects of the modern data science process, such as prediction engineering, data preprocessing, and model development.

5.1.2 Automated methods to augment collaboration

In FeatureHub, we enable and support collaboration through a variety of approaches, including the automated combination of feature values into a single feature matrix, the in-notebook feature discovery API and the integrated Discourse-based discussion forum (Section 3.3.4). However, there are many possible avenues for further support-

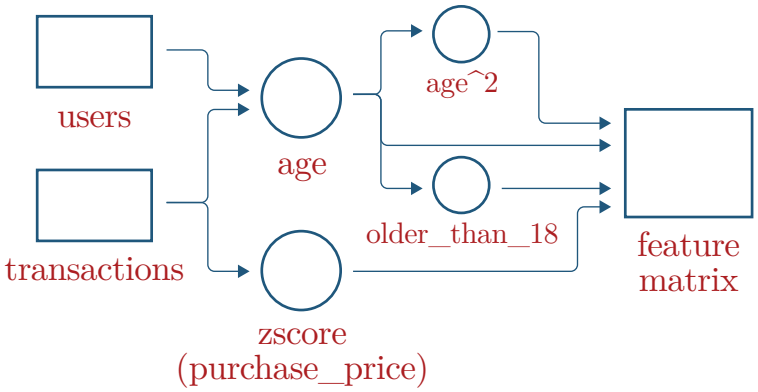


Figure 5-1: Feature engineering represented as a directed, acyclic data-flow graph. Edges represent the flow of data between nodes, which transform their inputs and send the output to other consumers.

ing collaboration.

Organizing efforts at scale

As the number of collaborating data scientists and the number of feature submissions increases, it becomes more difficult for any one data scientist to process the sheer volume of work being produced by collaborators. That is, when the scale of the collaboration is small, two data scientists writing features to transform the same column or table can easily notice that they are working on the same problem and can join forces. If there are one thousand other data scientists, then this might not be the case. To be most successful in this endeavor, we will attempt to make use of several strategies including both interface design and automation.

- *Proactive organization.* In this approach, we begin by assigning data scientists into small groups focused on different feature engineering subtasks. These tasks could be organized by different tables of raw data, different columns, different types of features (time series, categorical, standardization, etc.), or something else entirely.

- *Real-time organization.* We attempt to detect in real-time that a data scientist is working on a feature that has already been created. To detect these duplicate, redundant, or highly correlated features, we could operate on source code,

feature values, or both. By operating on the source code, we could detect using either static or dynamic analysis whether two functions are accessing the same column of the underlying data table. We could also wait for the feature to be evaluated, and then determine pairwise correlations between the proposed feature values and existing feature values from already-submitted features. A hybrid approach may combine the best of each strategy. Detection of redundancies could be achieved before writing the feature is too far underway, saving time and effort.

- *Novel user interfaces.* We ask FeatureHub collaborators to work in the Jupyter Notebook environment, which raises the possibility of creating novel Notebook-based (or Jupyter Lab-based) interface elements. For example, a sidebar showing the features submitted by other data scientists that updated in real-time would take advantage of Jupyter Notebook's powerful extensibility. Users could select a feature description in the sidebar and have a new cell be created in the Notebook that they could then adapt for their own purposes and ideas.

5.1.3 Creating interpretable models

One opportunity that is afforded by our feature abstractions is to associate with each feature an interpretable, natural-language description that can be shown to domain experts. Such descriptions aid greatly in the interpretability of some models. Indeed, we require that our FeatureHub collaborators include self-reported descriptions of their feature in natural language alongside their feature submissions.

This also raises the possibility of automated feature description generation. General-purpose tools for mapping source code to natural language could be applied to the collected feature functions, though they often produce dense and unnatural results, especially for complex features. Since we have collected a unique dataset of mappings between features and descriptions through our FeatureHub experiments, we could consider supervised generative modeling to automate the creation of natural language descriptions of features. By targeting our application to the feature engineering do-

main, and by collecting source code, self-reported descriptions, and extracted feature values jointly, we may be able to improve on existing approaches.

To be sure, there is substantial variation in the way data scientists described the 1952 features that we collected. Likely, it would first be necessary to normalize descriptions using some controlled natural language.

5.2 Challenges we faced

The development and evaluation of FeatureHub revealed a variety of challenges. Some of these challenges are engineering in nature, others are algorithmic, and still others are design-oriented. Regardless of the specific challenge, these all present opportunities for future research and development.

5.2.1 Maintaining platforms

A first challenge was the development and implementation of FeatureHub itself. It is no small task to develop the means to isolate, evaluate, submit, and combine feature engineering source code. But as a hosted platform (see Section 1.3.3), there is also a need to support the functioning of many other aspects of this full-fledged cloud application. The engineering effort to do this is nontrivial, and requires knowledge of many different layers of the application stack. Here are a few of the specific engineering problems that I addressed.

- *Cloud hosting, computing, and storage.* Our application is built on various cloud technologies, including Amazon Web Services S3, EC2, EBS, Route53, and SES. It was necessary to become familiar with these technologies in order to configure them for our use, including configuring access permissions and networking between different components. For many organizations, a dedicated professional would lead this effort.

- *User management and authentication.* JupyterHub provides a pluggable system to manage and authenticate users. This system must be configured for the

specific application. In our case, we wanted to create usernames and passwords for our experiment participants to avoid having them connect other OAuth-capable accounts for privacy purposes. This turned out to be a very manual process (without dedicating further engineering effort).

- *Application security.* In exposing any application to the internet, security risks are present. Specific configuration was required to enable TLS/HTTPS, without which JupyterHub would not operate. We also provided users with a full-fledged execution environment on our servers, which had to be suitably virtualized using Docker containers to avoid compromise.

- *Application deployment.* Using a containerized architecture makes it easier than otherwise to deploy the entire FeatureHub application. Regardless, the application consists of a JupyterHub instance container, a MySQL database instance container, an evaluation server container, and separate containers provisioned on demand for each authenticated user. To deploy this application, we used *Docker Compose*. A separate server was provisioned and configured manually to host the Discourse-based forum. (Additional details are available in Appendix A.7.)

- *Efficient feature evaluation.* Users expect evaluation of their features in real-time. To support that, we implement lightweight feature evaluation routines which can be executed in seconds (Section 3.3.3). However, more engineering would allow some intermediate results to be cached and more involved feature evaluation techniques to be supported.

- *Efficient automated machine learning.* In the ideal FeatureHub application, the problem coordinator refits an automated machine learning model after every feature is submitted (Section 3.4). However, many existing *AutoML* libraries require on the order of days to search the model and parameter space, which is impractical for our purposes. Strategies focusing on models that can be trained partially in the feature dimension would be appropriate, as would be distributed,

stand-alone *AutoML* systems,

- *Application scaling.* Given the cloud computing resources available during the time of our experiment, it was possible to use one large AWS EC2 instance to host most of the application, including the Hub and the Notebook containers, by imposing per-user resource limits. To scale much beyond on the order of 50 concurrent users, much more careful engineering would be required. To achieve this, *Docker Swarm* could be used to provision new compute servers for user notebooks. Alternately, other battle-tested orchestration technologies like *Ansible* or *Kubernetes* could be integrated. This level of DevOps work may require experienced collaborators. This is nothing to say of the challenge of scaling the automated machine learning services present in the application.

- *Integration testing.* FeatureHub is a complex application, as has hopefully been made clear in this section. To ensure such an application functions correctly requires complex testing infrastructure. For integration tests, I resorted to ad-hoc scripts that deploy a fresh FeatureHub instance, create users, log in as a user, and submit features to the backend. This is a fragile and arduous approach that could be better addressed using mocking and other testing strategies.

All of these problems would be severely compounded if not for the plethora of high-quality, open-source software upon which FeatureHub is built. Most notably, Jupyter Notebook and JupyterHub are fundamental building blocks, for which I am very appreciative.

5.2.2 Administering user studies

Even after addressing the challenges in Section 5.2.1, it was another step to administer user studies on top of our platform. One aspect of our user study was to separate users into two experimental groups and one control group to investigate the efficacy of our different attempts for facilitating collaboration. This required us to create three separate FeatureHub instances to maintain experiment integrity. However, this more

than tripled the amount of work to deploy and administer the FeatureHob platform instances.

Another challenge was that we wanted to provide comprehensive documentation for users to learn the system as well as possible, hopefully allowing us to isolate the effect of the platform on facilitating collaboration. Unfortunately, even with significant effort in creating documentation and tutorial materials, user errors were found. Specifically, some users did not understand that they needed to first evaluate and then also *submit* their feature to the system. Postmortem investigation revealed that many high-quality features had been created within users' notebooks but not submitted to the system. It is likely that our modeling performance in comparison to competitors on Kaggle would have been improved if some of these high-quality features had been included.

5.2.3 Motivating contributions

The FeatureHub model for collaborative data science can be used in several different environments. In one environment, an organization with a tight-knit data science team deploys a FeatureHub instance internally as a way to facilitate collaboration and easily develop integrated predictive modeling solutions. In another environment, a FeatureHub project can be made available more widely, either to paid freelancers or open-source contributors. In this latter situation, the problem of motivating feature contributions arises (Section 2.4). One overarching strategy is to be able to provide feedback on the value of feature contributions, whether for determining financial compensation or public acknowledgment.

Evaluating feature contributions

Why is evaluating feature contributions so challenging? There are several countervailing factors. The first is that features cannot be evaluated in isolation; rather, they should be evaluated in the context of their contribution to the overall model. This means that it is not enough to compute mutual information between the feature and

the target, or some other similar feature selection/evaluation strategy — the entire model must be considered. The second is that we want to simultaneously incentivize collaboration (sharing existing work, suggesting improvements to the work of others, etc.) and discourage manipulation. There are many forms that this manipulation can take. Suppose we were to use a naïve scheme of evaluating features based on mutual information with the target and awarding compensation based on the most highly ranked features according to this criterion. Then, a freerider could observe existing work in the spirit of collaboration, submit it as her own, and then absorb a portion of the payout from the original author. Another form is a spammer who generates random vectors centered at the mean of the training dataset target and submits thousands of these vectors. Some of them might be highly ranked by pure chance, earning the spammer a non-zero payout. The burden to the system of processing all these submissions would be significant.

This gives rise to the following considerations:

- reward features for their marginal contribution to a final model,
- reward features based on their positive influence on other features, and
- penalize spam features, malicious features, and other features submitted "in bad faith".

Even these motivating principles for evaluating features may not be enough. One possible approach is to make a submodularity assumption and then put forth an efficient approximation to the otherwise combinatorial problem of evaluating all feature subsets.

A compensation scheme for features

Without an immediate solution to the problem posed in Section 5.2.3, we resorted to intermediate approaches, as described in Section 4.1.4. This solution, based on bonus payments over several categories, tries to avoid rewarding manipulative behavior. Even so, the assessment of performance becomes highly manual and burdensome for the experimenters.

5.2.4 Adversarial behavior

The considerations of the previous sections should motivate future work on evaluating feature performance in the collaborative setting with the possibility of manipulative behavior. However, that discussion does not even cover the full range of adversarial behavior. This behavior is all the more possible when financial compensation is at stake. Adversarial behavior in the FeatureHub paradigm includes the following possibilities and more:

- Users submit manipulative features for financial gain, as previously discussed.
- Users submit low quality features, whether maliciously or accidentally, in terms of noise, runtime, code quality, or similar.
- Users take advantage of the ability to get the system to execute untrusted code to sabotage the platform and extract withheld test set target values or other sensitive data.

Fully addressing all of these is beyond the scope of a prototype collaborative data science system, but is necessary in the future for production systems depending on the threat model of the specific collaborative project.

Chapter 6

The future of collaborative data science

The ideas and systems presented so far in this thesis represent one foray into the new world of collaborative data science. This world shares similarities with collaborative software development, to be sure, but it would be a mistake to approach these worlds the same way. Instead, I have proposed a new approach based on restructuring the modern data science process using well-defined, task-specific abstractions. I focus on feature engineering and develop a full-fledged collaborative data science platform and bring together freelance data scientists from around the world to create an integrated predictive machine learning model using this platform.

But is this approach, represented in the FeatureHub platform, the way forward? In this chapter, I speculate on the future of collaborative data science. I will recall some of the opportunities and challenges presented in the previous chapter (Sections 5.1 and 5.2), and consider whether they can be seized or overcome under current approaches.

6.1 The problem with platforms

Collaborative data science through hosted platforms (Section 1.3.3) is a fraught path. As previously discussed, there are huge engineering challenges with creating, scaling,

and maintaining these systems. However, these engineering challenges are not even the main reason why, in my opinion, hosted platforms are not the way forward for collaborative data science, and especially not the way forward for open data science efforts.

While FeatureHub and other similar platforms have enabled significant progress towards collaborative data science, as "hosted platforms" they are constrained in important ways. In using these hosted platforms, users log into a remote server, where compute, storage, and source code editing resources are made available. The reliance on external hosting becomes a fundamental limitations for how successful collaborative endeavors can be, particularly for open data science projects. There are a variety of problems with hosted platforms.

Financial costs. To operate a hosted platform, a company or organization incurs various financial costs. These include the costs of servers, storage, engineering, and maintenance. Even if the platform is developed and maintained completely by volunteers, the cloud "hard costs" are not insignificant. In the case of a commercial offering, the costs of using a hosted platform can run into the tens or hundreds of dollars per-user, per-month, or on the order of tens or hundreds of thousands of dollars per year for larger collaborations. This is an untenable situation for open data science efforts. Even a model in which nonprofit organizations provide grants to open data science efforts to cover these costs is flawed, partly because the number of open data science collaborators cannot easily scale up or down.

There is also a significant risk of *lock-in*, especially for ambitious projects with many-year horizons. The lock-in risk is that a project hosted on a third party's servers and running a third party's application cannot be easily ported elsewhere. This allows the third party to charge a higher price for the use of their platform and the resold cloud resources.

Environment flexibility. To use a hosted platform, data scientists connect via a web interface and write code using an editing environment providing by the platform. This situation constrains data scientists into a certain development environment, whereas they otherwise have varied preferences on editors, programming languages,

and visualization packages. For the same reasons that organizations do not constrain their software developers to all use the same editor or IDE (let alone a web-based solution), neither should large-scale collaborative data science projects.

Trust and security. In using a hosted platform as the one-stop-shop for an entire data science project, the organization becomes vulnerable to any number of failures on the part of the platform. This includes data leaks with sensitive data, security vulnerabilities, server downtime, delays in supporting requested features, and more. For many applications, putting this much faith in a third party may not be acceptable.

Transparency. For open data science projects, transparency is a requirement during development. The same open-source philosophy of open-source software (that the source code itself be freely accessible by all who are interested) also applies to open data science projects. Imagine that collaborators were developing an application to predict the incidence of some diseases yielding an associated recommendation to deploy public health resources to specific areas to combat that disease. Why should policymakers and the public trust the recommendations of the application if they cannot even inspect the modeling assumptions, and request third party audits? Transparent development is crucial for high-leverage open data science projects.

Freedom. Finally, hosted platforms, as commercial products, are usually not free (as in *libre*) software. Often times the software is closed source as well. Though there are exceptions, such as FeatureHub and other academic projects, this lack of freedom is inappropriate for the open data science paradigm.

6.2 A turn towards platformless collaboration

These considerations motivate a turn towards platformless collaboration. In this approach, we facilitate collaboration on an integrated machine learning model without relying on functionality provided by a monolithic central server — and without yielding control to a hosted platform.

The challenge of platformless collaboration is that we need to find replacements for all the functionality that a hosted platform provides. This includes the ability

for collaborators to easily acquire training data, write and evaluate features, and submit them to a centralized repository; and the ability to validate submissions before accepting them to the repository, collect and import submitted features, and train and deploy an integrated model. Without the ability to operate and control our own server, it will require a fair amount of imagination to achieve all of these objectives.

I propose the development of a new paradigm for platformless collaborative data science, with a focus on feature engineering. Under this approach collaborators will develop feature engineering source code on their own machines, in their own preferred environments. They will submit their source code to a authoritative repository that will use still other services to verify that the proposed source code is syntactically and semantically valid and to evaluate performance on an unseen test set. If tests pass and performance is sufficient, the proposed code can be merged into the repository of features comprising the machine learning model.

In one sense, this is fairly similar to the FeatureHub paradigm. However, the key innovation is to remove the dependence on a managed and development environment, which was previously integral to the functioning of the platform. Indeed, I envision that these dependencies would otherwise be harmful in the long run. Rather, this concentration of functionality and separation of concerns will allow us to focus on developing more natural abstractions and prepare us for a new phase of research.

6.3 Lightweight collaborative systems

Lightweight collaborative systems enable collaborative data science projects without relying on a hosted platform to manage collaboration. These systems will be built on top of open, widely-used existing platforms and tools.

Motivated by the enormous ecosystem of free and open tooling for open-source software development, I propose to repurpose some of these tools for collaborative data science. First, one can scaffold a predictive modeling project in a specific repository structure, leveraging widely-used templating tools such as *cookiecutter*. GitHub, as the primary home for open-source software, can host the repository and be the

focal point for external software integrations. Code submissions can be formatted as highly structured pull requests. Continuous integration providers freely test and evaluate open-source codebases, and these providers can be used to dynamically create and execute test suites for submitted code in pull requests. Overall, by imposing a specific structure on data science projects, and by creating and leveraging a constellation of externals libraries, tools and integrations, we can develop integrated predictive models without the drawbacks of hosted platforms (Section 1.3.3) or vanilla software engineering approaches (Section 1.3.2).

With this vision for one future for collaborative data science, I conclude. Over the course of this thesis, I have introduced the reader to the modern data science process and how existing approaches to collaborative data science fit in. I then propose a new paradigm for collaborative data science, focused in integrating feature engineering contributions from diverse collaborators, and instantiate this approach in a cloud platform. Ultimately, while this platform enables successful collaborations, I propose to turn attention in a slightly different direction for future development of large-scale, collaborative open data science.

Bibliography

[1] Kaggle airbnb new user bookings. https://www.kaggle.com/c/airbnb-recruiting-new-user-bookings. Accessed: 2017-06-04.

[2] Kaggle sberbank russian housing market. https://www.kaggle.com/c/sberbank-russian-housing-market. Accessed: 2017-06-04.

[3] Alec Anderson, Sébastien Dubois, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. Sample, estimate, tune: Scaling bayesian auto-tuning of data science pipelines. *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 361–372, 2017.

[4] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[5] Esther E Bron, Marion Smits, Wiesje M Van Der Flier, Hugo Vrenken, Frederik Barkhof, Philip Scheltens, Janne M Papma, Rebecca ME Steketee, Carolina Méndez Orellana, Rozanna Meijboom, et al. Standardized evaluation of algorithms for computer-aided diagnosis of dementia based on structural mri: the caddementia challenge. *NeuroImage*, 111:562–579, 2015.

[6] The quant crunch: How the demand for data science skills is disrupting the job market. https://public.dhe.ibm.com/common/ssi/ecm/im/en/iml14576usen/analytics-analytics-platform-im-analyst-paper-or-report-iml14576usen-20171229.pdf, 2017.

[7] Justin Cheng and Michael S Bernstein. Flock: Hybrid crowd-machine learning classifiers. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 600–611. ACM, 2015.

[8] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD Conference*, 2015.

[9] Data For Democracy. Boston crash modeling. https://github.com/Data4Democracy/boston-crash-modeling.

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern*

Recognition, 2009. CVPR 2009. IEEE Conference on, pages 248–255. IEEE, 2009.

[11] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[12] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, pages 2962–2970, 2015.

[13] Michael J. Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD Conference*, 2011.

[14] Ned Gulley. Patterns of innovation: a web-based matlab programming contest. In *CHI'01 extended abstracts on Human factors in computing systems*, pages 337–338. ACM, 2001.

[15] Guido Hertel, Sven Niedner, and Stefanie Herrmann. Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel. *Research policy*, 32(7):1159–1177, 2003.

[16] Ayush Jain, Akash Das Sarma, Aditya Parameswaran, and Jennifer Widom. Understanding workers, developing effective tasks, and enhancing marketplace dynamics: a study of a large crowdsourcing marketplace. *Proceedings of the VLDB Endowment*, 10(7):829–840, 2017.

[17] James Max Kanter. *The data science machine: emulating human intelligence in data science endeavors.* PhD thesis, Massachusetts Institute of Technology, 2015.

[18] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *Data Science and Advanced Analytics (DSAA), 2015. 36678 2015. IEEE International Conference on*, pages 1–10. IEEE, 2015.

[19] 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. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[20] Arno J. Knobbe, Marc de Haas, and Arno Siebes. *Propositionalisation and Aggregates*, volume 2168, pages 277–288. Springer Science & Business Media, 2001.

[21] Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[22] Stefan Kramer, Nada Lavrač, and Peter Flach. *Propositionalization Approaches to Relational Data Mining*, pages 262–291. Springer Berlin Heidelberg, 2001.

[23] Karim R Lakhani and Robert G Wolf. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. *MIT Sloan Working Paper No. 4425-03*, 2003.

[24] Hoang Thanh Lam, Tran Ngoc Minh, Mathieu Sinn, Beat Buesser, and Martin Wistuba. Learning features for relational data. *CoRR*, abs/1801.05372, 2018.

[25] Hoang Thanh Lam, Johann-Michael Thiebaut, Mathieu Sinn, Bei Chen, Tiep Mai, and Oznur Alkan. One button machine for automating feature engineering in relational databases. *CoRR*, abs/1706.00327, 2017.

[26] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[27] James Robert Lloyd, David K Duvenaud, Roger B Grosse, Joshua B Tenenbaum, and Zoubin Ghahramani. Automatic construction and natural-language description of nonparametric regression models. In *AAAI*, pages 1242–1250, 2014.

[28] Ben Lorica. Crowdsourcing feature discovery. http://radar.oreilly.com/2014/03/crowdsourcing-feature-discovery.html, 2014.

[29] Michael McKerns and Michael Aivazis. Pathos: a framework for heterogeneous computing. http://trac.mystic.cacr.caltech.edu/project/pathos, 2010.

[30] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[31] Jon Noronha, Eric Hysen, Haoqi Zhang, and Krzysztof Z. Gajos. Platemate: crowdsourcing nutritional analysis from food photographs. In *UIST*, 2011.

[32] National Oceanic and Atmospheric Administration. Dengue forecasting project. http://dengueforecasting.noaa.gov/docs/project_description.pdf.

[33] City of Chicago. Food inspections evaluation. https://github.com/Chicago/food-inspections-evaluation.

[34] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 485–492, New York, NY, USA, 2016. ACM.

[35] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, November 2011.

[36] Santiago Perez De Rosso and Daniel Jackson. What's wrong with git?: a conceptual design analysis. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 37–52. ACM, 2013.

[37] Alexander J Ratner, Christopher M De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. Data programming: Creating large training sets, quickly. In *Advances in Neural Information Processing Systems*, pages 3567–3575, 2016.

[38] Matthew Rocklin. Pangeo: Jupyterhub, dask, and xarray on the cloud. http://matthewrocklin.com/blog/work/2018/01/22/pangeo-2, 2018.

[39] Maria Rossi. Decoding the "free/open source (f/oss) software puzzle" a survey of theoretical and empirical contributions. *Università degli Studi di Siena Working Paper No. 424*, 2004.

[40] Matthew Salganik, Ian Lundberg, Alex Kindel, and Sara McLanahan. Fragile families challenge. https://fragilefamilieschallenge.org.

[41] Akash Das Sarma, Ayush Jain, Arnab Nandi, Aditya Parameswaran, and Jennifer Widom. Surpassing humans and computers with jellybean: Crowd-vision-hybrid counting algorithms. In *Third AAAI Conference on Human Computation and Crowdsourcing*, 2015.

[42] Benjamin Schreck and Kalyan Veeramachaneni. What would a data scientist ask? automatically formulating and solving predictive problems. In *Data Science and Advanced Analytics (DSAA), 2016 IEEE International Conference on*, pages 440–451. IEEE, 2016.

[43] MG Siegler. Eric schmidt: Every 2 days we create as much information as we did up to 2003. *TechCrunch. August*, 4, 2010.

[44] Micah J. Smith, Roy Wedge, and Kalyan Veeramachaneni. Featurehub: Towards collaborative data science. *2017 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 590–600, 2017.

[45] Chong Sun, Narasimhan Rampalli, Frank Yang, and AnHai Doan. Chimera: Large-scale classification using machine learning, rules, and crowdsourcing. *Proceedings of the VLDB Endowment*, 7(13):1529–1540, 2014.

[46] Thomas Swearingen, Will Drevo, Bennett Cyphers, Alfredo Cuesta-Infante, Arun Ross, and Kalyan Veeramachaneni. Atm: A distributed, collaborative, scalable system for automated machine learning. *2017 IEEE International Conference on Big Data (Big Data)*, pages 151–162, 2017.

[47] Colin Taylor, Kalyan Veeramachaneni, and Una-May O'Reilly. Likely to stop? predicting stopout in massive open online courses. *CoRR*, abs/1408.3382, 2014.

[48] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proc. of KDD-2013*, pages 847–855, 2013.

[49] Vernon Turner, John F Gantz, David Reinsel, and Stephen Minton. The digital universe of opportunities: Rich data and the increasing value of the internet of things. *IDC Analyze the Future*, page 5, 2014.

[50] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[51] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Appendix A

Implementation of FeatureHub

In this appendix, I present implementation details of the FeatureHub platform. The full source code is available on GitHub under the MIT License at https://github.com/HDI-Project/FeatureHub.

A.1 Configuration

There are a variety of configuration settings that can be set by the app administrator. This first set of configuration variables must be set to ensure proper functioning of the app.

- `FF_DOMAIN_NAME`: Fully-qualified domain name of this server, used for Let's Encrypt SSL certificate.
- `FF_DOMAIN_EMAIL`: Email associated with domain name registration, used for SSL certificate.
- `MYSQL_ROOT_PASSWORD`: Password for DB root user.
- `EVAL_API_TOKEN`: API token for eval server. The administrator should generate a valid API token by executing `openssl rand -hex 32`.
- `HUB_CLIENT_API_TOKEN`: API token for Hub API client script.

- `DISCOURSE_DOMAIN_NAME`: Domain name of the Discourse forum.
- `DISCOURSE_CLIENT_API_USERNAME`: Username for Discourse admin, to use with Discourse API client. The administrator must use the same username when setting up the Discourse instance.
- `DISCOURSE_CLIENT_API_TOKEN`: API token for Discourse API client. The administrator must use the same token when setting up the Discourse instance.

The app administrator can reset the following configuration variables if they desire, but reasonable defaults are otherwise provided.

- `DOCKER_NETWORK_NAME`: Name for docker network used for networking containers.
- `HUB_IMAGE_NAME`: Name for Docker JupyterHub image.
- `HUB_CONTAINER_NAME`: Name for Docker JupyterHub container.
- `HUB_API_PORT`: Port for JupyterHub REST API to listen on.
- `FF_IMAGE_NAME`: Name for Docker user Jupyter Notebook server image.
- `FF_CONTAINER_NAME`: Prefix for Docker user containers. Actual containers will have the user's username appended.
- `FF_DATA_DIR`: Path to data directory on host machine (deployment machine).
- `FF_PROJECT_NAME`: Name for docker-compose app.
- `FF_IDLE_SERVER_TIMEOUT`: Timeout interval for idle servers (i.e., user containers). If a server is idle for this amount of time, then it will be automatically stopped. Due to some JupyterHub defaults, this value should be set to at least 5m 30s (330).
- `FF_CONTAINER_MEMLIMIT`:Memory limit for user containers. The value can either be an integer (bytes) or a string with a `K`, `M`, `G` or `T` prefix.

- `MYSQL_CONTAINER_NAME`: Name for Docker DB container.
- `MYSQL_ROOT_USERNAME`: Username for DB root user.
- `MYSQL_DATABASE`: Competition database name in DB.
- `MYSQL_DATA_VOLUME_NAME`: Name for DB data volume.
- `SECRETS_VOLUME_NAME`: Name for JupyterHub secrets volume.
- `USE_LETSENCRYPT_CERT`: Flag (`yes`/`no`) to use Let's Encrypt certificates. For testing purposes, one can use openssl to issue a self-signed certificate.
- `EVAL_IMAGE_NAME`: Name for Docker eval server image.
- `EVAL_CONTAINER_NAME`: Name for Docker eval server container. Since there is one eval container, this will be the exact name.
- `EVAL_CONTAINER_PORT`: Port for eval server to listen on.
- `EVAL_FLASK_DEBUG`: Whether eval server Flask app should be started with `DEBUG` flag.
- `DISCOURSE_FEATURE_GROUP_NAME`: Group for new users to be added to on Discourse. This group will control the visibility of the feature category.
- `DISCOURSE_FEATURE_CATEGORY_NAME`: Category for new features to be posted to on Discourse. Posts to this category will only be visible by members of the feature group (above) and administrators.
- `USE_DISCOURSE`: Flag (`yes`/`no`) to use Discourse integration. If Discourse integration is enabled, then forum posts are created for each feature that is successfully added.

An example `.env` configuration file is shown in Listing D.3.

A.2 User interface

The user interacts with a FeatureHub application instance through a client object, `Session`, created in their Jupyter Notebook session. The user imports a client that is dynamically created for the data science problem. The client will be used to acquire data, evaluated proposed features, and register them with the feature database. As shown in Table 3.1, the user can interact via the following methods provided by the client.

- `get_sample_dataset`: This method is used to load the training data into our workspace. The result is a tuple where the first element is `dataset`, a dictionary mapping table names to `DataFrame`s, and the second element is `target`, a `DataFrame` with one column. This one column is what we are trying to predict. The dataset can then be explored inline within the notebook.

- `evaluate`: This method evaluates the proposed feature on the training dataset and returns key performance metrics. The user first writes a candidate feature for submission to the FeatureHub project. Then, they can evaluate it on training data. The evaluation routine proceeds as follows.

  The evaluate function takes a single argument: the candidate feature. An evaluation client then runs the feature in an isolated environment to extract the feature values. The isolated environment is used to ensure that the global variables in the user's working environment are not necessary for the feature's execution. Next, the feature values themselves are validated to make sure that some constraints are satisfied, such as that the shape of the feature values is correct and there are no null values. Then, a simple machine learning model is built using the one feature and important cross-validated metrics are computed. If the feature is valid, the results are printed a mapping is returned where the keys are the metric name and the values are the metric values. If the feature is invalid, the reason is printed (a list of validation failures) and an empty dictionary is returned.

In the user's workflow, this method may be run several times. At first, it may reveal bugs or syntax errors to be fixed. Next, it may reveal that the feature did not meet some of the FeatureHub requirements, such as returning a single column of values or using function-scope imports. Finally, the user may find that your feature's performance, in terms of metrics like classification accuracy or mean squared error, are not as good as hoped, and the feature may then be modified or discarded.

- `discover_features`: This method allows the user to discover and filter features in the database that have been added by the user and other workers. Alternately, the user can call the wrapper method `print_my_features` to filter the submitted features down to the ones submitted by the user themself.

- `submit`: This method submits feature to the evaluation server for evaluation on test data. If successful, the feature is registered in the feature database and key performance metrics are returned. The exact same steps as in `evaluate` are repeated on the remote evaluation server, except using unseen test data. This ensures that the user does not "self-report" local evaluation results that are invalid.

A.3 Feature evaluation

Feature evaluation is tightly tied to automated modeling, in that our primary strategy to evaluate proposed features is to build models and assess performance with and without the proposed feature.

Feature evaluation is performed by an "eval server" running as a separate Hub-managed service. This service is a Flask server that exposes `log_evaluation_attempt` and `submit` endpoints. A lightly-edited implementation is shown in Listing D.1. The submit endpoint re-purposes the same evaluation code that is used by the user locally (`EvaluatorServer`, a subclass of `EvaluatorClient`). The main difference is that with the test set available, we can compute metrics using a train/test split (rather

than via cross-validation). Upon success, the eval server will also insert results into the feature database and post the source code to the integrated Discourse forum.

A.4 Discussion forum

We tightly integrate a Discourse-based discussion forum to enable communication and collaboration between users. The application administrator must follow several simple steps to set up the forum. Afterwards, users can log into the forum to view and discuss features, and the FeatureHub backend will automatically post new features to the server for all to see.

1. Setup a mail server for the forum software to use. We use AWS Simple Email Service (SES).

2. Provision a server and install Discourse, for example using the official cloud installation instructions[1].

3. Create accounts for each of your users and provide them with credentials. In future engineering work, a SSO (single sign-on) service could be configured to unify authentication for the FeatureHub app and the Discourse forum, possibly through GitHub integration.

4. Create a category for new features to be posted to.

A.5 Automated machine learning

The modeling interface follows from Table 3.1 and includes the `extract_features` and `learn_model` methods.

- `extract_features`: The project coordinator can compile all feature source code and extract feature values on an arbitrary dataset using the function `featurehub.admin.build_feature_matrix`.

[1] https://github.com/discourse/discourse/blob/master/docs/INSTALL-cloud.md

- `learn_model`: FeatureHub provides a flexible `Model` object which configures an underlying estimator for the different problem types that can be handled by the system, covering regression and classification, with different number of classes and loss functions. By default, a simply decision tree estimator is fit using the extracted feature values. The coordinator can also choose to use the extended `AutoModel` class, which uses `auto-sklearn` to automatically select and tune a model from the *scikit-learn* library.

A.6 Data management

The FeatureHub application uses a MySQL database to store both data and metadata related to the collaboration. In one FeatureHub instance, many users can each be working on many prediction problems. This motivates storing metadata about the separate prediction problems as well, where the metadata consists of the type of prediction problem (e.g. classification or regression), the locations of data files, and more. The full entity-relationship diagram is shown in Figure A-1.

The database is also the central location for storing the features that have been submitted by collaborators. We store the feature source code, the feature description, and the MD5 hash of the feature source code. In addition, we serialize the Python function using the `dill` library [29] and insert the serialized function directly into the database (shown as `function`) for the benefit of faster feature extraction during the modeling phase.

A.7 Deployment

A variety of resources facilitate easy deployment of FeatureHub. These include:

- *Makefile*, to manage all resources.
- *Docker*, to configure, create, and manage service containers.
- *Docker Compose*, for actual deployment of Hub, User, and DB containers.

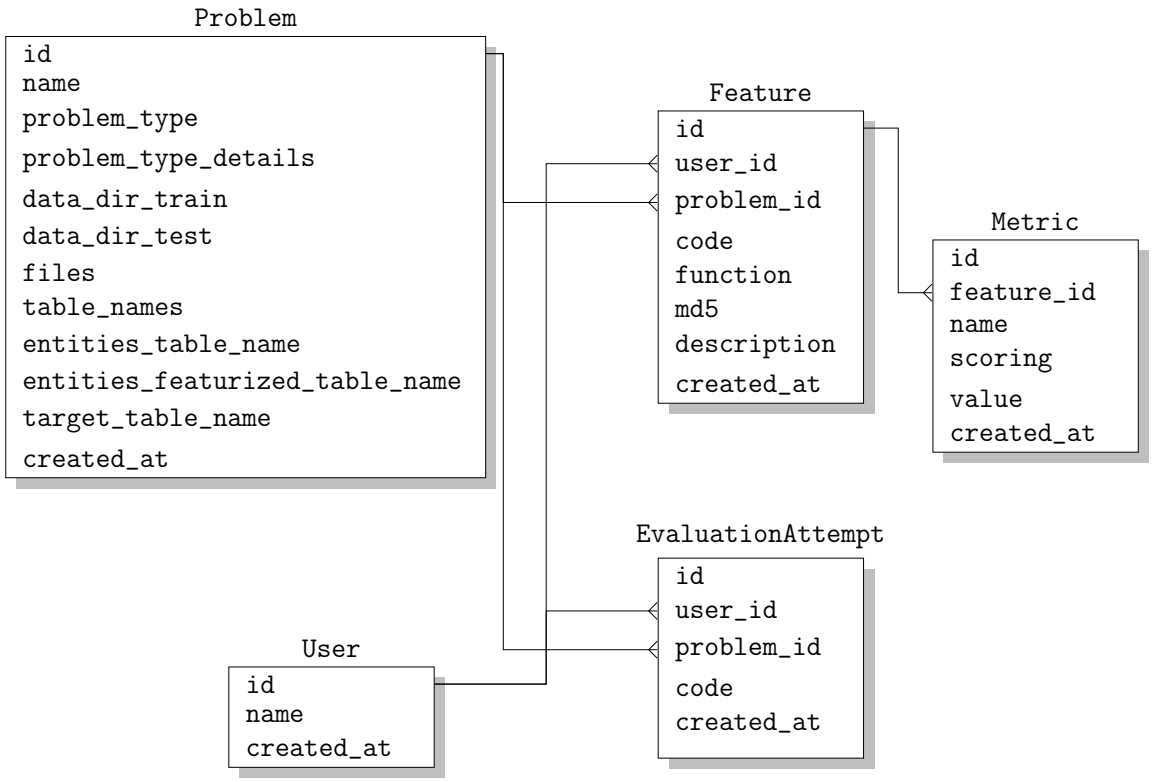


Figure A-1: Entity-relationship diagram for FeatureHub database. The crow's foot arrow represents a one-to-many relationship; that is, each feature has many metrics associated with it, and so on.

- Custom command-line utilities to create users, delete users, generate SSL certificates using LetsEncrypt, and more.
- Configuration files for JupyterHub, Jupyter Notebook, and the application itself (`.env`).

This makes deployment as easy as the following:

```
$ make build
$ make up
```

For full control, the following Make targets are included:

- *Application lifecycle management*: `up`, `down`, `start`, `stop`
- *Storage management*: `setup_files`, `teardown_files`, `kill_network`, `volumes`, `kill_db_volumes`, `kill_secrets_volume`
- *Container management*: `build`, `build_static`, `build_services`, `kill_containers`, `kill_images`
- *Network management*: `network`, `ssl`
- *Application resource monitoring*: `monitor_start`, `monitor_stop`, `monitor_delete`

These leverage Docker Compose configuration files for application management and container build management. The main Docker Compose configuration file is reproduced in Listing D.2.

Appendix B

Prediction problem details

In this chapter, I provide further details on the prediction problems used in FeatureHub experiments.

B.1 Problem statement

We introduce users to the prediction problems by providing them with a problem statement inspired by that accompanying the Kaggle competition.

For *airbnb*, we provided the following problem statement:

> Instead of waking to overlooked "Do not disturb" signs, Airbnb travelers find themselves rising with the birds in a whimsical treehouse, having their morning coffee on the deck of a houseboat, or cooking a shared regional breakfast with their hosts.
>
> New users on Airbnb can book a place to stay in 34,000+ cities across 190+ countries. By accurately predicting where a new user will book their first travel experience, Airbnb can share more personalized content with their community, decrease the average time to first booking, and better forecast demand.

> In this task, we aim to predict in which country a new user will make his or her first booking.
>
> You are challenged to identify and derive or generate the features which would help the most in predicting in which country a new user will make their first booking.

For *sberbank*, we provided the following problem statement:

> Housing costs demand a significant investment from both consumers and developers. And when it comes to planning a budgetâĂŤwhether personal or corporateâĂŤthe last thing anyone needs is uncertainty about one of their biggest expenses. Sberbank, RussiaâĂŹs oldest and largest bank, helps their customers by making predictions about realty prices so renters, developers, and lenders are more confident when they sign a lease or purchase a building.
>
> Although the housing market is relatively stable in Russia, the country's volatile economy makes forecasting prices as a function of apartment characteristics a unique challenge. Complex interactions between housing features such as number of bedrooms and location are enough to make pricing predictions complicated. Adding an unstable economy to the mix means Sberbank and their customers need more than simple regression models in their arsenal.
>
> In this task, we aim to predict realty prices by developing algorithms which use a broad spectrum of features.
>
> You are challenged to identify and derive or generate the features which would help the most in predicting realty prices.

B.2 Data schemata

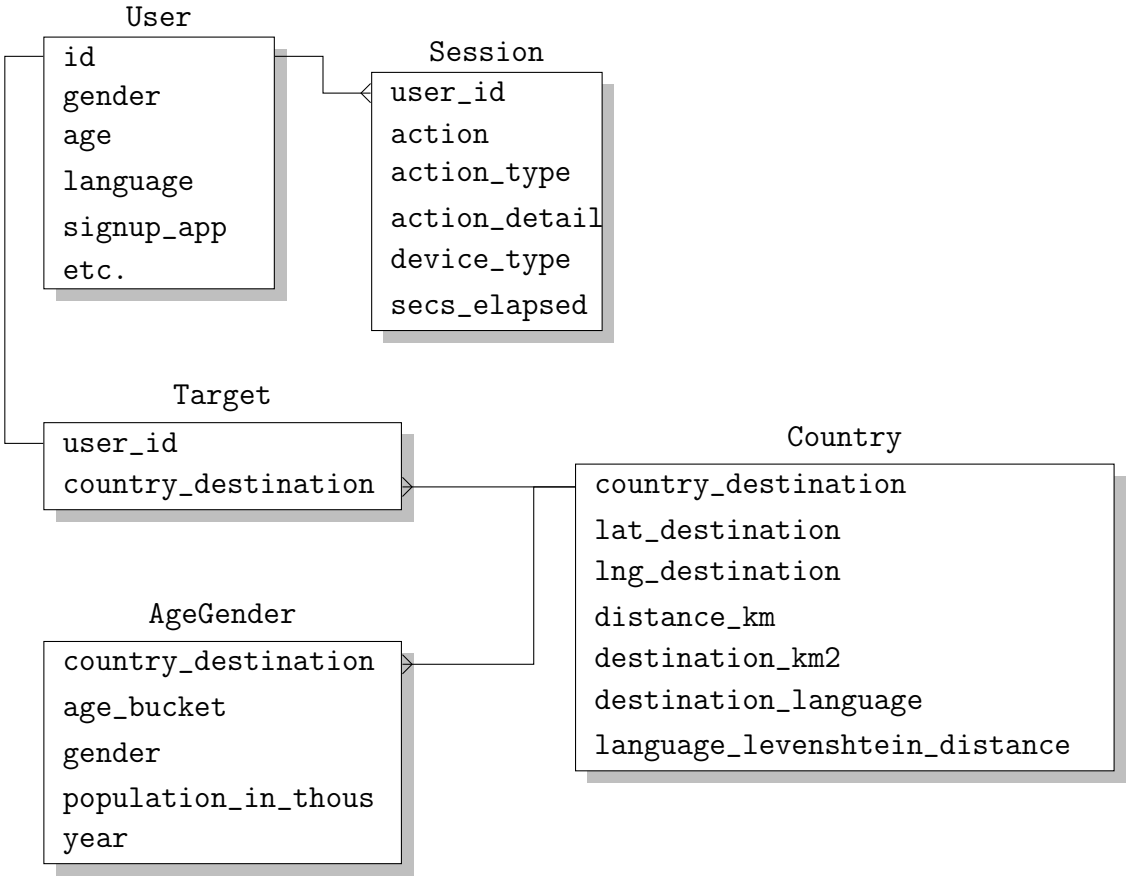


Figure B-1: Entity-relationship diagram for Kaggle *airbnb* prediction problem. The entities table is `users` and the target table is `target`. The list of attributes is truncated for `users` (15 total).

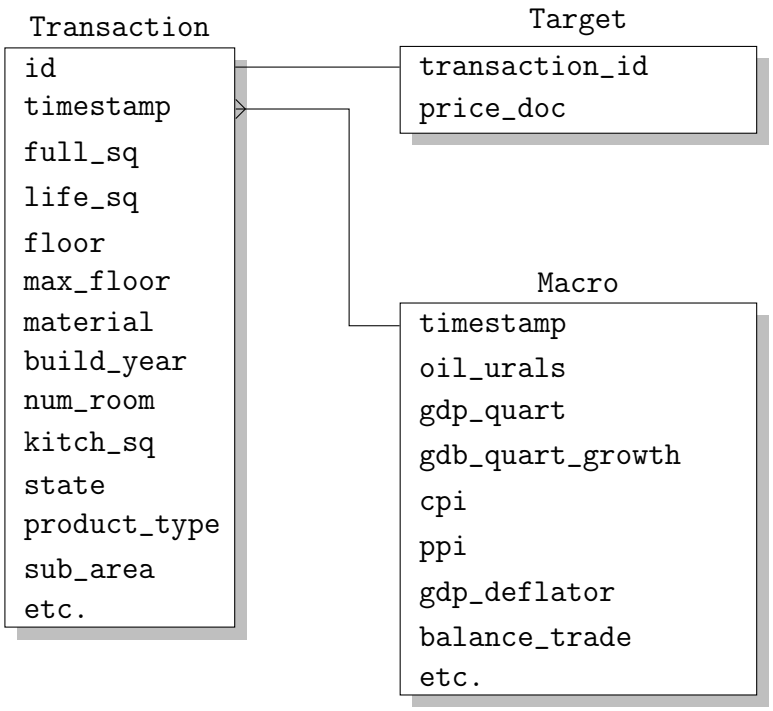


Figure B-2: Entity-relationship diagram for Kaggle *sberbank* prediction problem. The entities table is `transactions` and the target table is `target`. The list of attributes is truncated for `transactions` (291 total) and `macro` (100 total).

B.3 Pre-extracted features

For both problems, we pre-extracted simple features to initialize the feature engineering, as described in Section 3.2. Users were told that these pre-extracted features would be joined with the features they write during evaluation.

For the *airbnb* problem, the following features were pre-extracted:

- `date_account_created`: Convert string date to days since August 11, 2008, the launch of Airbnb (according to Wikipedia).
- `timestamp_first_active`: Convert integer timestamp to days since August 11, 2008.
- `date_first_booking`: Convert string date to days since August 11, 2008. Replace NaNs with 0 and add another column for missingness.
- `gender`: Convert to one-hot encoding.
- `age`: Replace NaNs with 0 and add another column for missingness.
- `signup_method`: Convert to one-hot encoding.
- `signup_flow`: Convert to one-hot encoding.
- `language`: Convert to one-hot encoding.
- `affiliate_channel`: Convert to one-hot encoding.
- `affiliate_provider`: Convert to one-hot encoding.
- `first_affiliate_tracked`: Convert to one-hot encoding.
- `signup_app`: Convert to one-hot encoding.
- `first_device_type`: Convert to one-hot encoding.
- `first_browser`: Convert to one-hot encoding.

Split	Metric	Value
Train	Accuracy	0.79
Train	Precision	0.79
Train	Recall	0.79
Train	ROC AUC	0.89
Test	Accuracy	0.79
Test	Precision	0.79
Test	Recall	0.79
Test	ROC AUC	0.89

Table B.1: Metrics for decision tree model training on pre-extracted feature matrix for different splits.

The resulting feature matrix (before any feature engineering on the part of the users) has 153 columns. We also informed the users that a model trained on just the pre-extracted feature matrix yields the metrics in Table B.1.

For the *sberbank* problem, the following features were pre-extracted:

- `full_sq`: This column was extracted directly.
- `floor`: Replace NaNs with 0 and add another column to indicate missingness.
- `build_year`: Replace NaNs with 0 and add another column to indicate missingness.
- `life_sq`: Replace NaNs with 0 and add another column to indicate missingness.
- `num_room`: Replace NaNs with 0 and add another column to indicate missingness.
- `kitch_sq`: Replace NaNs with 0 and add another column to indicate missingness.
- `material`: Convert to one-hot encoding.
- `state`: Convert to one-hot encoding.
- `product_type`: Convert to one-hot encoding.

Split	Metric	Value
Train	RMSE	4278266.78
Train	R-squared	0.19
Test	RMSE	3716072.47
Train	R-squared	0.40

Table B.2: Metrics for decision tree model training on pre-extracted feature matrix for different splits.

- `sub_area`: Convert to one-hot encoding.

The resulting feature matrix (before any feature engineering on the part of the users) has 170 columns. We also informed the users that a model trained on just the pre-extracted feature matrix yields the metrics in Table B.2.

Appendix C

User study survey

In the next section, the "Data Scientists for Feature Extraction" survey that was given to participants of our user study is reproduced. Users were informed that their responses would be completely anonymous, and no identifying information was collected.

Per Section 4.1.3, we separated participants into three experiment groups for the purpose of evaluating the effect of the explicit collaboration functionality. For Group 1, we did not provide access to the `discover_features` command nor the *features* category on the forum, so we did not ask about these functionalities in the survey. Thus, Appendix C.1.4 was only shown to Groups 2 and 3.

Then, in Appendix C.2, I provide summary statistics about all survey responses.

C.1 Survey questions

Thanks for participating in Data Scientists for Feature Extraction. We really appreciate your answers to the following questions. This helps us evaluate and improve our research. This survey should take less than 15 minutes. Your response is completely anonymous.

C.1.1 Getting oriented

1. Before this task, were you familiar with the term "feature engineering"?

   ❍ No

   ❍ Yes

2. What did you think about the documentation and introduction materials (tutorial notebook, website, instructions in email)?

   ❍ These materials answered all of my questions.

   ❍ I was a bit confused at first, but eventually figured things out myself.

   ❍ I was a bit confused at first, but I received help from someone else that answered my questions.

   ❍ I never completely figured out what to do.

3. How long did you spend, at the beginning, learning about the platform through the documentation and introduction materials?

   ❍ 0-10 minutes

   ❍ 11-20 minutes

   ❍ 21-40 minutes

   ❍ 41-60 minutes

   ❍ More than 60 minutes

4. Do you have any feedback about the documentation and introduction materials?

   _____

C.1.2 FeatureHub Platform

1. How easy was the FeatureHub platform to use? (This includes logging in, using the notebook, and evaluating and submitting features.)

	1	2	3	4	5	
Difficult to use	❍	❍	❍	❍	❍	Easy to use

2. How long did it take to write and *evaluate* your first feature, starting when you first logged in? (Not including the tutorial.)

   ❍ 0-15 minutes

   ❍ 16-30 minutes

   ❍ 31-60 minutes

   ❍ 1-2 hours

   ❍ 2-3 hours

   ❍ 3-5 hours

   ❍ I did not evaluate any features.

3. How long did it take to write and *submit* your first feature, starting when you first logged in? (Not including the tutorial.)

   ❍ 0-15 minutes

   ❍ 16-30 minutes

   ❍ 31-60 minutes

   ❍ 1-2 hours

   ❍ 2-3 hours

   ❍ 3-5 hours

   ❍ I did not evaluate any features.

4. How easy was it to use FeatureHub commands including `get_sample_dataset`, `print_my_features`, `evaluate`, and `submit`?

	1	2	3	4	5	
Difficult to use	❍	❍	❍	❍	❍	Easy to use

5. Do you have any feedback about the FeatureHub commands?

   _____

6. How long did you spend, in total, writing features?

❍ 0-1 hours

❍ 1-2 hours

❍ 2-3 hours

❍ 3-4 hours

❍ 4-5 hours

❍ More than 5 hours

7. Do you have any feedback about the FeatureHub platform as a whole?

   _____

C.1.3 Forum

1. How helpful was the forum for the feature engineering task overall?

	1	2	3	4	5	
Not helpful at all	❍	❍	❍	❍	❍	Very helpful

2. How easy was the forum to use?

	1	2	3	4	5	
Difficult to use	❍	❍	❍	❍	❍	Easy to use

3. Do you have any feedback about the forum?

   _____

C.1.4 Collaboration

[*This section was shown to Groups 2 and 3, i.e. those that* were *given explicit collaboration functionality.*]

1. How many times did you use the `discover_features` command to discover new features?

   ❍ Never

   ❍ 1-2 times

❍ 3-5 times

❍ 6-10 times

❍ 11-20 times

❍ More than 20 times

2. How helpful was the `discover_features` command for the feature engineering task overall?

	1	2	3	4	5	
Not helpful at all	❍	❍	❍	❍	❍	Very helpful

3. How many times did you browse features that were automatically posted to the forum?

❍ Never

❍ 1-2 times

❍ 3-5 times

❍ 6-10 times

❍ 11-20 times

❍ More than 20 times

4. How much did using the `discover_features` command and browsing features on the forum help to...

If you did not use these features at all, please leave these questions blank.

(a) ...learn how to properly submit features?

	1	2	3	4	5	
Did not help at all	❍	❍	❍	❍	❍	Helped quite a bit

(b) ...learn new feature engineering techniques (such as joining/merging Data Frames or using scikit-learn Encoders)?

	1	2	3	4	5	
Did not help at all	❍	❍	❍	❍	❍	Helped quite a bit

(c) ...see features written by others in order to avoid duplicating work?

	1	2	3	4	5	
Did not help at all	❍	❍	❍	❍	❍	Helped quite a bit

(d) ...see features written by others in order to modify and improve those ideas?

	1	2	3	4	5	
Did not help at all	❍	❍	❍	❍	❍	Helped quite a bit

(e) ...see which columns in the data were most frequently used?

	1	2	3	4	5	
Did not help at all	❍	❍	❍	❍	❍	Helped quite a bit

C.2 Survey responses

We asked about our introductory materials to help data scientists learn the FeatureHub platform, the usability of the platform, the helpfulness of the forum, and the helpfulness and usability of the collaboration mechanisms. The results are shown in Tables C.1 to C.4. Answers on a likert scale were converted to numerical 1 to 5 values. Answers to bucketed questions were set at the median point of the bucket or the minimum for open-ended questions.

	count	Familiar with feature eng.	Helpfulness of intro		Time spent on intro		
		mean	median	mean	median	mean	std
group							
1	6	0.67	3.00	3.17	50	39.17	18.55
2	12	0.83	3.50	3.25	30	32.08	14.53
3	10	1.00	3.00	3.30	30	36.00	15.60
Total	28	0.86	3.00	3.25	30	35.00	15.46

Table C.1: Summary of survey responses for intro questions (Appendix C.1.1),

group	count	Usability of platform		Time until first evaluation			Time until first submission		
		mean	median	mean	median	std	mean	median	std
1	6	4.50	5	96.25	33.75	111.99	125.00	120.00	102.58
2	12	4.17	4	42.50	22.50	44.47	80.45	90.00	52.19
3	10	4.10	4	70.00	45.00	71.94	81.67	45.00	73.24
Total	28	4.21	4	63.61	22.50	72.85	91.15	90.00	72.58

(a)

group	count	Usability of APIs		Time spent writing features		
		mean	median	mean	median	std
1	6	4.50	5	285.00	285	16.43
2	12	4.25	5	257.50	270	50.29
3	10	4.60	5	264.00	270	62.93
Total	28	4.43	5	265.71	270	50.14

(b)

Table C.2: Summary of survey responses for platform questions (Appendix C.1.2).

group	count	Helpfulness of forum			Usability of forum		
		median	mean	std	median	mean	std
1	6	4.00	4.17	0.75	5.00	4.83	0.41
2	12	4.00	3.42	1.24	4.50	4.17	1.03
3	10	3.50	3.40	0.97	4.00	4.20	0.79
Total	28	4.00	3.57	1.07	5.00	4.32	0.86

Table C.3: Summary of survey responses for forum questions (Appendix C.1.3).

group	count	Times used `discover_features`			Helpfulness of `discover_features`		
		mean	median	std	mean	median	std
2	12	4.88	1.50	5.65	2.92	3.00	1.44
3	10	4.65	1.50	5.85	3.40	4.00	1.17
Total	28	4.77	1.50	5.60	3.14	3.00	1.32

(a)

group	count	Times browsed forum			Helpfulness overall for submit		
		mean	median	std	mean	median	std
2	12	5.71	4.00	6.82	3.45	3.00	1.51
3	10	5.60	2.75	6.91	3.43	4.00	1.51
Total	28	5.66	4.00	6.69	3.44	3.50	1.46

(b)

group	count	Helpfulness overall learn feature eng.			Helpfulness overall to avoid dup.		
		mean	median	std	mean	median	std
2	12	2.90	3.00	1.29	3.30	3.50	1.42
3	10	3.67	4.00	1.51	4.00	4.00	1.15
Total	28	3.19	3.00	1.38	3.59	4.00	1.33

(c)

group	count	Helpfulness overall to improve ideas			Helpfulness overall to see used cols		
		mean	median	std	mean	median	std
2	12	3.00	3.00	1.15	2.67	3.00	1.50
3	10	3.17	3.00	1.17	3.43	3.00	1.51
Total	28	3.06	3.00	1.12	3.00	3.00	1.51

(d)

Table C.4: Summary of survey responses for collaboration questions (Appendix C.1.4).

Appendix D

Listings

```
"""Evaluation server for FeatureHub user notebooks."""

import os
from functools import wraps
from urllib.parse import unquote_to_bytes

import dill
from flask import Flask, Response, redirect, request
from sqlalchemy.orm.exc import (
    MultipleResultsFound, NoResultFound)

from featurehub.admin.sqlalchemy_declarative import (
    EvaluationAttempt, Feature, Metric, Problem, User)
from featurehub.admin.sqlalchemy_main import ORMManager
from featurehub.evaluation import (
    EvaluationResponse, EvaluatorServer)
from featurehub.evaluation.discourse import DiscourseFeatureTopic
from featurehub.evaluation.future import HubAuth
from featurehub.evaluation.util import authenticated
from featurehub.util import get_function, is_positive_env, myhash

# setup
prefix = "/services/eval-server"
hub_api_token = os.environ.get("EVAL_API_TOKEN")
hub_api_url = "http://{}:{}/hub/api".format(
    os.environ.get("HUB_CONTAINER_NAME"),
    os.environ.get("HUB_API_PORT")
)
auth = HubAuth(
    api_token=hub_api_token,
    api_url=hub_api_url,
    cookie_cache_max_age=60,
)

DEMO_PROBLEM_NAME = "demo"

# app
app = Flask("eval-server")


@app.route(prefix + "/log-evaluation-attempt", methods=["POST"])
@authenticated
def log_evaluation_attempt(user):
    """Log user evaluation of feature.
```

```
    Extracts 'database', 'problem_id', and 'code' from POST body.
    """

    try:
        # read parameters from form
        database = request.form["database"]
        problem_id = request.form["problem_id"]
        code = request.form["code"]

        # insert evaluation attempt into database
        user_name = user["name"]
        orm = ORMManager(database, admin=True)
        with orm.session_scope() as session:
            user_obj = (session
                        .query(User)
                        .filter(User.name == user_name)
                        .one())
            problem_obj = (session
                           .query(Problem)
                           .filter(Problem.id == problem_id)
                           .one())
            evaluation_attempt_obj = EvaluationAttempt(
                user=user_obj, problem=problem_obj, code=code)
            session.add(evaluation_attempt_obj)
    finally:
        return Response()


@app.route(prefix + "/submit", methods=["POST"])
@authenticated
def submit(user):
    """Process user request to submit feature.

    Extracts 'database', 'problem_id', 'code', and 'description'
    from POST body.
    """

    # read parameters from form
    try:
        database = request.form["database"]
        problem_id = request.form["problem_id"]
        feature_dill = request.form["feature_dill"]
        code = request.form["code"]
        description = request.form["description"]
    except Exception:
        return EvaluationResponse(
            status_code=EvaluationResponse
                        .STATUS_CODE_BAD_REQUEST)

    # preprocessing
    orm = ORMManager(database, admin=True)
    with orm.session_scope() as session:
        # look up the problem in the databasse
        try:
            problem_obj = (session
                           .query(Problem)
                           .filter(Problem.id == problem_id)
                           .one())
        except (NoResultFound, MultipleResultsFound) as e:
            return EvaluationResponse(
                status_code=EvaluationResponse
                            .STATUS_CODE_BAD_REQUEST)
        except Exception:
            return EvaluationResponse(
                status_code=EvaluationResponse
```

```
                        .STATUS_CODE_SERVER_ERROR)

        # look up the user in the database
        user_name = user["name"]
        try:
            user_obj = (session
                        .query(User)
                        .filter(User.name == user_name)
                        .one()
        except (NoResultFound, MultipleResultsFound) as e:
            return EvaluationResponse(
                status_code=EvaluationResponse
                        .STATUS_CODE_BAD_REQUEST)

        # compute the md5 hash of the feature code
        md5 = myhash(code)

        # check for duplicate feature code
        evaluator = EvaluatorServer(problem_id, user_name, orm)
        try:
            is_registered = evaluator.check_if_registered(code)
            if is_registered:
                return EvaluationResponse(
                    status_code=EvaluationResponse
                                .STATUS_CODE_DUPLICATE_FEATURE)
        except Exception:
            return EvaluationResponse(
                status_code=EvaluationResponse
                        .STATUS_CODE_SERVER_ERROR)

        # convert the feature code into a function
        try:
            feature = dill.loads(unquote_to_bytes(feature_dill))
        except Exception:
            return EvaluationResponse(
                status_code=EvaluationResponse
                        .STATUS_CODE_BAD_FEATURE)

        # processing
        # evaluate feature
        try:
            metrics = evaluator.evaluate(feature)
        except ValueError:
            # feature is invalid
            return EvaluationResponse(
                status_code=EvaluationResponse
                        .STATUS_CODE_BAD_FEATURE)
        except Exception:
            return EvaluationResponse(
                status_code=EvaluationResponse
                        .STATUS_CODE_SERVER_ERROR)

        try:
            # write to db
            feature_obj = Feature(
                description=description,
                feature_dill_quoted=feature_dill,
                code=code,
                md5=md5,
                user=user_obj,
                problem=problem_obj
            )
            session.add(feature_obj)
            for metric in metrics:
                metric_db = metric.convert(kind="db")
                metric_obj = Metric(
```

```
                    feature=feature_obj,
                    name=metric_db["name"],
                    scoring=metric_db["scoring"],
                    value=metric_db["value"]
                )
                session.add(metric_obj)
        except Exception:
            return EvaluationResponse(
                status_code=EvaluationResponse
                            .STATUS_CODE_DB_ERROR)

        # post to forum
        problem_name = problem_obj.name
        if (is_positive_env(os.environ.get("USE_DISCOURSE")) and
                problem_name != DEMO_PROBLEM_NAME):
            try:
                topic_obj = DiscourseFeatureTopic(
                    feature_obj, metrics)
                topic_url = topic_obj.post_feature()
            except Exception:
                topic_url = ""
        else:
            topic_url = ""

    # return status code and metrics dict
    return EvaluationResponse(
        status_code=EvaluationResponse.STATUS_CODE_OKAY,
        metrics=metrics,
        topic_url=topic_url,
    )


if __name__ == "__main__":
    # run app
    host = "0.0.0.0"
    port = int(os.environ.get("EVAL_CONTAINER_PORT", 5000))
    debug = is_positive_env(
        os.environ.get("EVAL_FLASK_DEBUG", False))
    app.run(host=host, port=port, debug=debug)
```

Listing D.1: FeatureHub eval server. The exposed endpoints include `log_evaluation_attempt` and `submit`. Some logging and authentication-related code has been removed for brevity.

```
version: '2'

networks:
    default:
        external:
            name: ${DOCKER_NETWORK_NAME}

volumes:
    db-data:
        external:
            name: ${MYSQL_DATA_VOLUME_NAME}
    secrets:
        external:
            name: ${SECRETS_VOLUME_NAME}

services:
    hub:
        build:
            context: .
            dockerfile: ./Dockerfile-hub
        image: ${HUB_IMAGE_NAME}
        container_name: ${HUB_CONTAINER_NAME}
        volumes:
            - /var/run/docker.sock:/var/run/docker.sock:rw
            - ${FF_DATA_DIR}:${FF_DATA_DIR}:rw
            - secrets:/etc/letsencrypt
        ports:
            - 443:443
            - ${HUB_API_PORT}:${HUB_API_PORT}
        environment:
            DOCKER_NETWORK_NAME: ${DOCKER_NETWORK_NAME}
            FF_DATA_DIR: ${FF_DATA_DIR}
            FF_IMAGE_NAME: ${FF_IMAGE_NAME}
            FF_CONTAINER_NAME: ${FF_CONTAINER_NAME}
            FF_IDLE_SERVER_TIMEOUT: ${FF_IDLE_SERVER_TIMEOUT}
            FF_CONTAINER_MEMLIMIT: ${FF_CONTAINER_MEMLIMIT}
            HUB_CONTAINER_NAME: ${HUB_CONTAINER_NAME}
            HUB_API_PORT: ${HUB_API_PORT}
            MYSQL_CONTAINER_NAME: ${MYSQL_CONTAINER_NAME}
            EVAL_CONTAINER_NAME: ${EVAL_CONTAINER_NAME}
            EVAL_CONTAINER_PORT: ${EVAL_CONTAINER_PORT}
            EVAL_API_TOKEN: ${EVAL_API_TOKEN}
            SSL_KEY: "/etc/letsencrypt/privkey.pem"
            SSL_CERT: "/etc/letsencrypt/cert.pem"
            HUB_CLIENT_API_TOKEN: ${HUB_CLIENT_API_TOKEN}
        command: >
            jupyterhub -f ${FF_DATA_DIR}/config/jupyterhub/ ↩
   jupyterhub_config.py
    db:
        image: mysql:5.7
        container_name: ${MYSQL_CONTAINER_NAME}
        ports:
            - 3306:3306
        environment:
            MYSQL_DATABASE: ${MYSQL_DATABASE}
            MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
        volumes:
            - db-data:/var/lib/mysql
    eval-server:
        build:
            context: .
            dockerfile: ./Dockerfile-eval
        image: ${EVAL_IMAGE_NAME}
        container_name: ${EVAL_CONTAINER_NAME}
        volumes:
```

```
            - ${FF_DATA_DIR}:${FF_DATA_DIR}:rw
            - ${FF_DATA_DIR}/data:/data:ro
        ports:
            - ${EVAL_CONTAINER_PORT}:${EVAL_CONTAINER_PORT}
        environment:
            FF_DATA_DIR: ${FF_DATA_DIR}
            HUB_CONTAINER_NAME: ${HUB_CONTAINER_NAME}
            HUB_API_PORT: ${HUB_API_PORT}
            MYSQL_CONTAINER_NAME: ${MYSQL_CONTAINER_NAME}
            EVAL_CONTAINER_PORT: ${EVAL_CONTAINER_PORT}
            EVAL_API_TOKEN: ${EVAL_API_TOKEN}
            EVAL_FLASK_DEBUG: ${EVAL_FLASK_DEBUG}
            MYSQL_ROOT_USERNAME: ${MYSQL_ROOT_USERNAME}
            MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
            DISCOURSE_DOMAIN_NAME: ${DISCOURSE_DOMAIN_NAME}
            DISCOURSE_CLIENT_API_USERNAME: ${ ↩
    DISCOURSE_CLIENT_API_USERNAME}
            DISCOURSE_CLIENT_API_TOKEN: ${ ↩
    DISCOURSE_CLIENT_API_TOKEN}
            DISCOURSE_FEATURE_GROUP_NAME: ${ ↩
    DISCOURSE_FEATURE_GROUP_NAME}
            DISCOURSE_FEATURE_CATEGORY_NAME: ${ ↩
    DISCOURSE_FEATURE_CATEGORY_NAME}
            USE_DISCOURSE: ${USE_DISCOURSE}
```

Listing D.2: Docker Compose configuration file for FeatureHub application.

```
DOCKER_NETWORK_NAME=featurehub-network
HUB_IMAGE_NAME=featurehubhub
HUB_CONTAINER_NAME=featurehubhub
HUB_API_PORT=8081
FF_IMAGE_NAME=featurehubuser
FF_CONTAINER_NAME=featurehubuser
FF_DATA_DIR=/var/lib/featurehub
FF_PROJECT_NAME=featurehub
FF_DOMAIN_NAME=
FF_DOMAIN_EMAIL=
FF_IDLE_SERVER_TIMEOUT=3600
FF_CONTAINER_MEMLIMIT=4G
MYSQL_CONTAINER_NAME=featurehubmysql
MYSQL_ROOT_USERNAME=root
MYSQL_ROOT_PASSWORD=
MYSQL_DATABASE=featurehub
MYSQL_DATA_VOLUME_NAME=db-data
SECRETS_VOLUME_NAME=secrets
USE_LETSENCRYPT_CERT=yes
EVAL_IMAGE_NAME=featurehubeval
EVAL_CONTAINER_NAME=featurehubeval
EVAL_CONTAINER_PORT=5000
EVAL_API_TOKEN=
EVAL_FLASK_DEBUG=
HUB_CLIENT_API_TOKEN=
DISCOURSE_DOMAIN_NAME=
DISCOURSE_CLIENT_API_USERNAME=featurehub
DISCOURSE_CLIENT_API_TOKEN=
DISCOURSE_FEATURE_GROUP_NAME=writers
DISCOURSE_FEATURE_CATEGORY_NAME=features
USE_DISCOURSE=yes
```

Listing D.3: Template .env configuration file for FeatureHub application.