Feature Engineering and Evaluation in Lightweight Systems

by

Kelvin Lu

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

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author .....
Department of Electrical Engineering and Computer Science
May 28, 2019

Certified by .....
Kalyan Veeramachaneni
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Feature Engineering and Evaluation in Lightweight Systems

by

Kelvin Lu

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 2019, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents `Ballet`, a lightweight, feature engineering framework that allows users to contribute to an open-source data science project. Specifically, it provides a framework for users to easily write flexible and high-quality features from a raw dataset. In addition, it provides a series of tests to ensure that all features in a project follow a consistent API and all provide some level of predictive power towards a target column.

For the latter task, we modified and implemented GFSSF, a feature selection algorithm designed specifically for grouped, streaming features. This included building a performant entropy and mutual information estimator for datasets, as well as integrating this algorithm into Travis CI, a popular continuous integration tool. We then evaluated our framework on a popular test dataset for data scientists, evaluating for performance and ease of development.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

I would like to thank my advisor, Kalyan Veeramachaneni, for mentorship, feedback, resources, and support. The ideas and results shown in this thesis would not have been possible without his collaboration and guidance.

I would also like to thank the members of the Data to AI lab for excellent collaboration on projects. I would like to especially thank Micah Smith for invaluable advice at the beginning of the project and sustained collaboration throughout. This project is a continuation of his previous work in the lab, and as such he has laid most of the foundations. This work would not have been possible without his guidance and support on every step of the way.

Additionally, I would like to thank the world-wide community of open-source contributors, for whom this work is targeted towards. Open source projects have benefited almost anyone who has touched a computer, and continues to be a source of interesting and useful software today.

Specifically, I would like to thank several projects that have allowed `Ballet` to run relatively cost-free: GitHub and Travis CI. I would also like to thank the python open-source data science library Scikit Learn, whose high quality python code has indirectly provided mentorship in my software development studies and whose own information estimators inspired the design of the estimators shown in this thesis.

Finally, I would like to thank friends and family who have supported me throughout my college career. It takes a village to raise a child, and I am deeply grateful for those who have shaped me to where I am today and those who have provided a supportive environment to grow.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, data has been collected in increasingly larger scales on every aspect of our lives, from disease monitoring in patients to user tracking on websites. However, data collection is not the end of the story; processing and data science are necessary to gain insight or make predictions from a corpus of data. In particular, many organizations and companies hire large teams of data scientists to tackle such problems.

However, many data-sets exist outside the focus of for-profit organizations; for example, the Fragile Families challenge [9] is a project that analyzes the results of interviews and questionnaires with different families over the course of many years. The goal is to link traits in a family's behavior and background with the child's later years. These projects rely on outside data scientists to contribute and collaborate in a public setting. This mode of development is close to open source software development, which has seen success in combining the efforts of many authors incrementally into coherent software; the most successful examples include the Linux Kernel, with a still-active contributor base of over thousands of software engineers and used by millions of consumers worldwide.

Open data science projects are a subset of open source software in which community members contribute to the analysis and engineering of public data-sets. However, these open data science projects are not nearly as prevalent as other open source software projects despite how much more prevalent public data-sets have become. One

issue is that there are few frameworks that help users effectively collaborate on different data science tasks.

In this chapter, we introduce and provide context for `Ballet` [16], a lightweight framework that allow users to contribute collaboratively to an open-source data science project on the task of feature engineering. We first present a high-level overview of `Ballet` and define feature engineering and the sub-tasks associated with it. Finally, we elaborate on the feature selection process, which is the main contribution of this thesis.

1.1 Current Collaborative Data Science

There currently exist a handful of platforms that allow for collaboration towards a data science project. We will discuss these platforms and their feasibility of use towards open-source data science projects.

1.1.1 Commercial Data Science Platforms

Kaggle [8] and other sites geared towards data science competitions often provide areas for collaborators to contribute their work publicly; specific to Kaggle, users can create "kernels" that can be viewed by others. These kernels are generally formatted as computational notebooks [10] and usually either provide the implementation of a certain aspect of the data science project, either to showcase a winning strategy or to provide tutorials for beginner data scientists. Using these kernels, users can share their work and also use the work of others to improve their data projects.

One large issue with Kaggle is that these kernels are specific to each user that created them and themselves do not contribute to a centralized project. Because of this, there usually exist many kernels that contain redundant work. Furthermore, because kernels themselves are not necessarily end-to-end data science projects, the formatting and readability can vary from kernel to kernel. For example, one kernel may explicitly share the code used to pre-process data while another simply describes the steps they took in an essay and leaves the implementation as an exercise to the

reader. This lack of common API detracts from the learning experience for beginners and makes it difficult to integrate different kernels together.

Furthermore, Kaggle itself is not meant for collaboration; it is a data science competition site. As such, collaboration and public kernels are limited and generally released post-competition. While Kaggle competitions sometimes provide motivation for users to contribute to public good data science problems, it is different than open-source data science; because it is a competition, each group works separately, which means that much of the work put towards building an end-to-end data pipeline is redundant between groups and that only a small fraction of the competition base is rewarded for their effort.

There also exist commercial data science platforms such as Domino Data Lab [1] and Daitaku [2] that aim to provide platforms for data scientists to support all aspects of data science product infrastructure and deployment. However, similar to Kaggle these platforms do not explicitly provide scaffolding for collaborative, incremental models.

1.1.2 Research in Collaborative Data Science

There has been much interest in facilitating increased collaboration in machine learning, though most existing work does not provide a structured way to ensure an effective division of labor. In some approaches, unskilled crowd workers can be harnessed for feature engineering tasks, such as by labeling data to provide the basis for further manual feature engineering [3]. Other proposed solutions include real-time editing interfaces, like that of [6, 7, 10], facilitating multiple users to edit a machine learning model specification at the same time. While these have led to state-of-the-art modeling performance, there is no natural way for competitors to integrate source code components in a systematic way.

Finally, FeatureHub [17] is a precursor to `Ballet` built to solve the issues surrounding collaborative, incremental data science projects. Many design aspects from

[1] `https://www.dominodatalab.com/`
[2] `https://www.dataiku.com/`

`Ballet` come as a result of research related to FeatureHub and will be introduced in detail with `Ballet`.

1.2 Feature Engineering

Feature engineering is the task of taking raw data and creating features that are suitable for a predictive model to use and relevant to a predictive target. There are several reasons for feature engineering in real-world data-sets:

gender	age	signup_method	signup_flow	language
-unknown- 54%	28.0 3%	basic 73%		en 95%
FEMALE 23%	27.0 3%	facebook 24%		zh 2%
Other (3) 22%	Other (123) 94%	Other (3) 3%	0 25	Other (23) 3%
FEMALE	35.0	facebook	0	en
-unknown-		basic	0	en
-unknown-		basic	0	en
-unknown-		basic	0	en
-unknown-		basic	0	en
FEMALE	28.0	basic	0	en
MALE	48.0	basic	25	en
-unknown-		basic	0	en
-unknown-		basic	0	en
-unknown-		basic	0	en
-unknown-		basic	0	en
FEMALE	30.0	basic	25	en
MALE	24.0	facebook	0	en
-unknown-		basic	0	en
-unknown-		basic	0	en
FEMALE	56.0	basic	0	en
-unknown-		basic	0	en

Figure 1-1: The first few columns out of the Airbnb New User Booking Dataset [1]. The gender and age of most users is unknown.

- **Data Cleaning and Processing**: In real-world data-sets, raw data-sets often have missing values and other issues that make prediction problematic. For example, in the Airbnb New Bookings Dataset, user data can only be collected on users who have logged in; however a considerable amount of the data-set pertains to actions taken by browsing users who have not logged in, as seen in Figure 1-2. It is in the best interest of data scientists to "clean" these columns by creating features that impute the values of missing data. It takes considerable

effort to fill in missing values and extract information from incompatible data columns.

- **Relational Databases**: Data is often broken into multiple tables with relational columns that connect or link them, with the relational values providing little information without the context of other data sets. For example, the Airbnb new bookings data set contains a "sessions" table that contains information about the actions a user takes (e.g. how long they browse, the device they use etc.). The value in this table's user column is a foreign key to the main "user" table. It takes time and effort to create features that link users to their sessions and extract meaningful data [20].

- **Engineering Challenges**: Even after a data set is pre-processed and cleaned, it may be prohibitively slow to run the predictive model on the entire data-set; for example, the Webb Spam Corpus [18] contains over 16,000,000 data rows collected from 350,000 web sites and is unsuitable for use as is. It falls upon Domain Experts to identify and create useful features from the existing data.

Each of these tasks is difficult to automate and often become bottlenecks in data science projects. The goal of `Ballet` is to reduce the time taken on each of these tasks by having users work together to solve these challenges.

1.3 Ballet

`Ballet` is framework for creating predictive models, with an emphasis on feature engineering. Within Ballet, one can instantiate a project, develop features, and validate their work [16] . `Ballet` projects live as git repositories and enforce API correctness and feature quality through Travis CI, a continuous integration tool that ensures every change to the project still allows the project to run smoothly and produce a high-quality feature matrix. The use of open-source tools such as git and Travis keeps in line with the lightweight and availability constraints for open source projects.

user_id	action	action_type	action_detail	device_type
135483 unique values	show 26% index 8% Other (357) 66%	view 34% data 20% Other (8) 46%	view_search_resu... 17% p3 13% Other (153) 70%	Mac Desktop 34% Windows Desktop 25% Other (12) 41%
d1mm9tcy42	search_results	click	view_search_results	Windows Desktop
d1mm9tcy42	lookup			Windows Desktop
d1mm9tcy42	search_results	click	view_search_results	Windows Desktop
d1mm9tcy42	lookup			Windows Desktop
d1mm9tcy42	search_results	click	view_search_results	Windows Desktop
d1mm9tcy42	lookup			Windows Desktop
d1mm9tcy42	personalize	data	wishlist_content_update	Windows Desktop
d1mm9tcy42	index	view	view_search_results	Windows Desktop
d1mm9tcy42	lookup			Windows Desktop
d1mm9tcy42	search_results	click	view_search_results	Windows Desktop
d1mm9tcy42	lookup			Windows Desktop
d1mm9tcy42	personalize	data	wishlist_content_update	Windows Desktop
d1mm9tcy42	index	view	view_search_results	Windows Desktop
d1mm9tcy42	similar_listings	data	similar_listings	Windows Desktop
d1mm9tcy42	ajax_refresh_subtotal	click	change_trip_characteristics	Windows Desktop
d1mm9tcy42	similar_listings	data	similar_listings	Windows Desktop
d1mm9tcy42	ajax_refresh_subtotal	click	change_trip_characteristics	Windows Desktop

Figure 1-2: The first few columns out of the Airbnb sessions data-set [1]. The user column is a foreign key that is meaningless without the context of another data-set.

1.3.1 Background and Previous Works

`Ballet` is based off previous work done in the Data to AI group; specifically, it is the descendant of a previous framework called FeatureHub.

Similar to `Ballet`, the goal of FeatureHub is to allow for multiple contributors to incrementally contribute to a centralized data science project through building features in a consistent API [17]. However, unlike `Ballet` which emphasizes a lightweight framework built off of available open-source tools, FeatureHub is a hosted platform built on the cloud designed for real-time collaboration.

While the FeatureHub platform provided more scaffolding for collaborators, ultimately several factors constrained the ways it could be used and who could use it. Of these factors included the maintenance and other costs of running FeatureHub; organizations using FeatureHub take on the burden of keeping infrastructure for the platform up-to-date and in-shape, which can sometimes be more cumbersome than

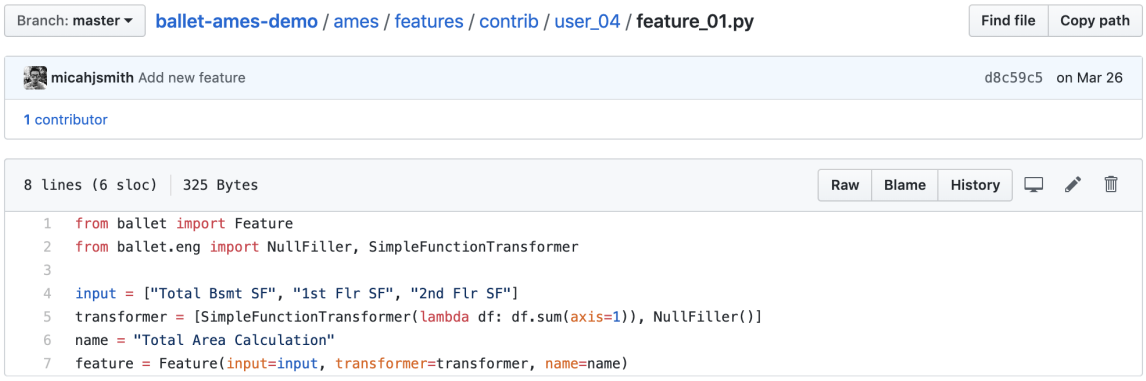

Figure 1-3: An example of a simple `Ballet` feature. This feature adds three columns together.

the feature engineering itself. Additionally, hosted platforms are generally not free software, which makes them inappropriate for open-source projects.

The work done with FeatureHub and previous projects motivate many of the design goals for `Ballet`. For example, to address the issue of financial costs, `Ballet` is built entirely on free, open-source tools such as GitHub and Travis CI. This continues to be a major design goal as we build more infrastructure for `Ballet`.

1.3.2 Overview of Ballet

For `Ballet`, the `Feature` abstraction provides a way for contributors to collaboratively build features in a Ballet project. `Feature` objects are transformers that follow a "fit/transform" paradigm that allows for data to be "fit" on training data before the "transform" stage is applied on other data, ensuring that transformations happen in a leakage-free manner. The general life-cycle of a Feature is as follows:

1. Individual contributors clone a Ballet project from GitHub and write features, submitting their work as a pull request. Each feature is formatted as an individual module located in the project's `contrib` directory.

2. Travis CI performs feature validation on the candidate features. This includes API coherency, project structure correctness, and a feature acceptance subroutine that ensures that features are not redundant and contributes information

towards the target feature. If a feature fails validation, it is rejected and no further action is taken in its life-cycle until it is updated.

3. Once Travis CI accepts the feature as correct and relevant, a maintainer of the project reviews the features and decides whether or not to merge.

4. When new features are accepted, another service determines if any existing feature has become redundant or irrelevant and prunes them.

This process ensures that only error-free, high quality features are accepted. This approach also scales well in the number of features; here, maintainers are the slowest step in validation and the CI tests beforehand ensure that they are not slowed down by well-meaning but low-quality contributions.

1.4 Main Contribution

The main contribution of this thesis is the improvement of **feature selection** in `Ballet`— the selection and rejection of features based on their usefulness to a predictive model. While we expect that most contributors to open source projects do not have malign intent, it is possible that well-meaning contributors can submit features that are not useful or even detrimental to the performance of a predictive model. We will first outline the problem statement and then a brief overview of what this entailed for the `Ballet` feature engineering life-cycle.

1.4.1 Problem Statement: Feature Selection

Definition. A **logical feature** f of a data set $\mathcal{D}$ as a function that maps raw data to a vector of values: $f^{\mathcal{D}} : \mathcal{V}^p \to \mathbb{R}^q$ where $\mathcal{V}$ is the set of feasible raw data values, p is the dimensionality of the raw data, and q is the dimensionality of the feature vector.

The problem statement for feature selection is to create a *feature matrix* of logical feature columns $\mathcal{F}(D) = [f_1(\mathcal{D}), f_2(\mathcal{D}), \ldots, f_n(\mathcal{D})]$ from a given set of candidate features that maximizes predictive power towards a target column and minimizes the

Table 1.1: Summary of Mathematical Notation

Notation	Mathematical Meaning
$\mathcal{D}$	Training Dataset, with dimensions $N \times d$
d	Dimension (number of columns) of the $\mathcal{D}$
$x_i^{\mathcal{D}}$	The i-th element of the data-set $\mathcal{D}$, of dimension $1 \times N$
y	Target column, with dimensions $N \times 1$
y_i	The i-th element of the target column y
$\mathcal{V}$	The set of all possible raw data values. Including the real numbers $\mathbb{R}$, this also includes strings and "null" or missing values.
$f_i, f_i^{\mathcal{D}}$	A logical feature (of the data-set $\mathcal{D}$)
$f_t, f_t^{\mathcal{D}}$	A candidate logical feature, proposed at time t
$f_i(\mathcal{D})$	The feature matrix produced by f_i transformed on $\mathcal{D}$
q_i	Number of columns in $f_i(\mathcal{D})$
$\mathcal{F}_t$	The set of accepted features at time t
Γ_t	The set of arrived features at time t
$\lvert * \rvert$	The size of a set.
λ_1, λ_2	Relevance threshold hyper-parameters.
I_T	Threshold information for feature selection.

dimensionality of the feature matrix. We break this down into two sub-problems: feature acceptance and feature pruning.

Recent research in feature selection has included feature selection for *streaming* features — features that arrive one by one, in a streaming fashion. This is because it has become expensive to store entire data sets in one location. Additionally, previous feature selection methods that consider the entire feature set at once are generally more computationally expensive. This is fortunate for `Ballet`, as features necessarily arrive in a streaming fashion as contributors write them. In such a setting, it is important to consider a feature f_i with respect to the set of accepted features that came before it, the current feature matrix $\mathcal{F}_t$ at a time t.

Feature Acceptance

An important quality for features is **relevance**, whether or not the feature adds new information about the target column. The following definitions are taken from Wu et. al.[19] .

Definition. A feature f_i is considered **irrelevant** to a target column y if $P(y|f_i(\mathcal{D})) = P(y)$

Definition. A feature f_i is considered **strongly relevant** to a target column y relative to an already existing feature set $\mathcal{F}$ if $P(y|f_i(\mathcal{D}), \mathcal{F}(\mathcal{D})) \neq P(y|\mathcal{F}(\mathcal{D}))$.

Definition. A feature f_i is considered **weakly relevant** to a target column y relative to an already existing feature set $\mathcal{F}$ if it is not strongly relevant and there exists some subset S of $\mathcal{F}$ such that $P(y|f_i, S(\mathcal{D})) \neq P(y|S(\mathcal{D}))$.

The problem of **Feature Acceptance** is whether or not to accept a candidate feature f_i into our feature matrix, taking into account relevance, dimensionality, and the already accepted features. We emphasize dimensionality as we are still looking to minimize our feature matrix size, logical features that produce matrices with fewer columns are preferred.

Note here that we may choose to accept a weakly relevant feature if it is considered more valuable to the feature matrix than the accepted features that prevented it from being strongly relevant; for example, a new feature that provides the same information as an older feature but with lower dimensionality may be accepted to take the place of the older feature. Feature Acceptance deals mostly with the acceptance of the new feature, while the removal of the existing feature is part of the next sub-problem: feature pruning.

Feature Pruning

As features arrive in a streaming fashion, we may find that new features are "improved" versions of previously accepted features. There are two main cases; the first is that an accepted feature's information is a subset of the candidate feature, and the candidate feature is more relevant. Another case where we may prune accepted features is if the new feature provides the same information as the accepted, but with fewer columns. In either case, it may be necessary for `Ballet` to remove existing and redundant features when a candidate feature is accepted.

For example, in a data-set for temperature prediction we may find that a feature representing the day of the year is relevant to the target (represented as an one-hot encoding $x \in \{0,1\}^{365}$), and may be accepted. While useful, this singular feature contributes 365 columns to our feature matrix. Later on, if a feature representing the month of the year is accepted (similarly represented as $x \in \{0,1\}^{12}$), it is possible that the information the new feature provides almost as much information as the previous feature (it is possible that temperatures stay relatively constant on a month-to-month basis). Then, it makes sense to remove the older feature from our feature matrix as the new feature provides the same information in much fewer columns.

The problem of **Feature Pruning** is to remove features $\{f_i, \dots\} \in \mathcal{F}$ that are no longer strongly relevant to the target after a candidate feature is accepted.

1.4.2 Listing of Contributions

Over the course of this thesis, we achieved the following:

- Created a library for data-set entropy estimation and data-set mutual information estimation for data-sets containing both continuous and discrete columns.
- Created a validator for feature acceptance that judges candidate features for relevance using the GFSSF [13] algorithm.
- Created a validator for feature pruning that judges accepted features for redundancy using the GFSSF algorithm.
- Implemented infrastructure to allow for `Ballet` and Travis CI to run pruning subroutines.
- Created a GitHub App that uses the GitHub API to automate portions of the feature engineering life-cycle, improving ease of using a `Ballet` project on both the contributor and maintainer side.
- Tested the usefulness of `Ballet` by simulating how a data science competition might be translated to a `Ballet` project.

Overall, these contributions allow for `Ballet` to maintain its feature pipeline invariant with little overhead for the project maintainers. As such, they help the success of `Ballet` as both an open source framework and a data science framework for large-scale projects.

1.5 Thesis Organization

The structure of this thesis flows from design, to implementation, to experimental results. Chapter 2 discusses the design goals we require of `Ballet`. Its existence as both an open-source and data science framework puts unique constraints that have shaped its development and design choices. We then discuss the Ballet feature engineering life-cycle in the context of these constraints and how users and maintainers interact with the `Ballet` framework.

Chapter 3 discusses the Feature Selection portions of the Feature Engineering Life-Cycle, and how `Ballet` defines and tests for high quality features. It discusses Grouped Feature Selection for Streaming Features (GFSSF), a feature selection algorithm we modified to fit the `Ballet` context.

Chapter 4 discusses instances of Ballet projects that we have created to test and simulate collaborative data science projects. It goes over key findings from real-world data science problems and the benefits of using Ballet in such a context.

Chapter 2

Ballet Overview

2.1 Design Considerations

As `Ballet` exists in both the open-source and data science communities, there are unique constraints that have influenced its design. Many of these constraints, especially dealing with redundancy and relevance, are shared but will still be discussed in both contexts.

2.1.1 Open Source Considerations

Many design goals arise because `Ballet` is designed to be used as an open-source framework. As, such, many of these design goals relate to ease of use, on the end of both the project owner (as well as other project moderators) and the project contributors.

It is important to consider what kinds of collaborators would take part in a `Ballet` project. In the open-source community, collaborators come from all different backgrounds and domain expertise; one contributor could be a senior data scientist well versed in feature engineering but with little domain knowledge, another could be a researcher in the exact field that the data-set comes from but has little experience coding. Yet another collaborator may be just well-meaning internet denizen trying to contribute however they can. With such a wide variety of collaborators, it is

important that `Ballet` caters to all backgrounds.

1. **Collaboration**: The ability for multiple authors to contribute to the same project and have their work integrate smoothly together. While each author will create unique content, their work must be consistent with a flexible and simple API. Additionally, it should be intuitive and simple to find and understand the work of other contributors.

2. **Lightweight**: While it is tempting to develop a sophisticated web app backed by large cloud compute instances and managed by dedicated DevOps professionals, this approach is simply not sustainable for open-source development. Isolated projects may secure sponsorship from cloud providers or funding agencies, but this is the exception, not the rule. Instead, Ballet projects must be built only on lightweight infrastructure: common components like open-source software libraries, free source code hosting, and free continuous integration testing

   Fortunately, there already exist many tools that are trusted in the open-source community and provide free solutions to otherwise expensive problems; GitHub is commonly used for version control, and Travis CI is popular for ensuring that contributions meet the standards of the project through unit tests and other "check suites". As such, our design of `Ballet` relies heavily on these technologies to ensure that our framework is both lightweight in cost and also built upon tools that contributors are already comfortable using.

3. **Scalability**: The ability for a framework to support multiple authors efficiently. Open source projects can reach many hundreds or thousands of collaborators. Having a maintainer review each and every contribution would be infeasible and would cause slow development. In addition, it is important that the checks run on the `Ballet` project are fast as well.

2.1.2 Data Science Considerations

In addition to the constraints of open-source development, it is important to ensure that the `Ballet` project is able to create a high-quality feature matrix that can be readily used by predictive models. Many of these considerations relate to feature selection and the nature of how features arrive and are processed in `Ballet`.

1. **Robustness**: Our "adversary" in such a context is a collaborator that is well-meaning but unintentionally submits code that is frail and errors when used on certain inputs — if accepted into a project, it could be potentially time consuming to find and remove poor code. As such, it is part of the design `Ballet` to ensure that all contributions are syntactically correct and follow the rules of the framework. This is analogous to a unit test in traditional software development.

2. **Relevance**: It may be the case that a collaborator submits a feature that they think is useful for prediction but in reality provides no useful information. Also, it is likely that different collaborators share similar mindsets and create features that are essentially the same. It is important to ensure features in a `Ballet` project are relevant and not redundant with respect to each other.

3. **Compactness**: In addition to relevance, one of the goals of feature selection is to minimize the size of the resulting feature matrix. As a single feature could create multiple data columns and not all of them are necessarily relevant, features with a large number of columns should be penalized in favor of features that provide similar information but are smaller. We consider two types of compactness here: compactness in the number of features accepted, and compactness in terms of number of feature columns total.

2.1.3 Security Design Considerations

`Ballet` is not built with security as a top priority; as such, the security of a `Ballet` project relies mainly on the security of the agents that comprise `Ballet`. GitHub and

Travis CI each provide security guarantees for different aspects of projects, though open-source software is often engaged in struggle against all sorts of malicious attacks [14, 2, 15]. Because `Ballet` caters to open source data science projects, accidentally leaking data is not as large of an issue as the data is already public.

As mentioned before, the main threat model of `Ballet` is well meaning collaborators that submit frail or broken code — we will have to check these features do not end up in the feature matrix. There are still possibilities of attackers submitting adversarial features or submitting harmful code in an attempt to break the validation routine — this will be in part the maintainer's duty to make sure such code does not make its way into the project.

2.2 Design Overview

This section is an overview of the different components of the `Ballet` framework. We begin with installation and project overview, and end with how contributors and moderators interact with `Ballet`. In the end, the framework provides a **feature engineering pipeline invariant** — at any time, the project is able to preform efficient, high-quality, end-to-end feature engineering on a new data points. Maintaining this invariant in the face of numerous and varied contributors is the responsibility of the **feature engineering life-cycle**, which ensures that any features currently in the project meet the standard of the invariant.

2.2.1 Project Structure

A `Ballet` project exists as a GitHub Repository with three main components:

1. A `ballet.yml` that contains information about the project.
2. A `load_data.py` file that contains a function that is used to load a data points that the features will fit and transform upon.
3. A `contrib` directory that contains the work of all the contributors. Each contributor creates their own sub-directory titled `user_{$name}` in which they write

their features. Features files are required have the format `feature_{$feature_name}.py`.

See Figure 2-1 for an example structure of a `Ballet` project.

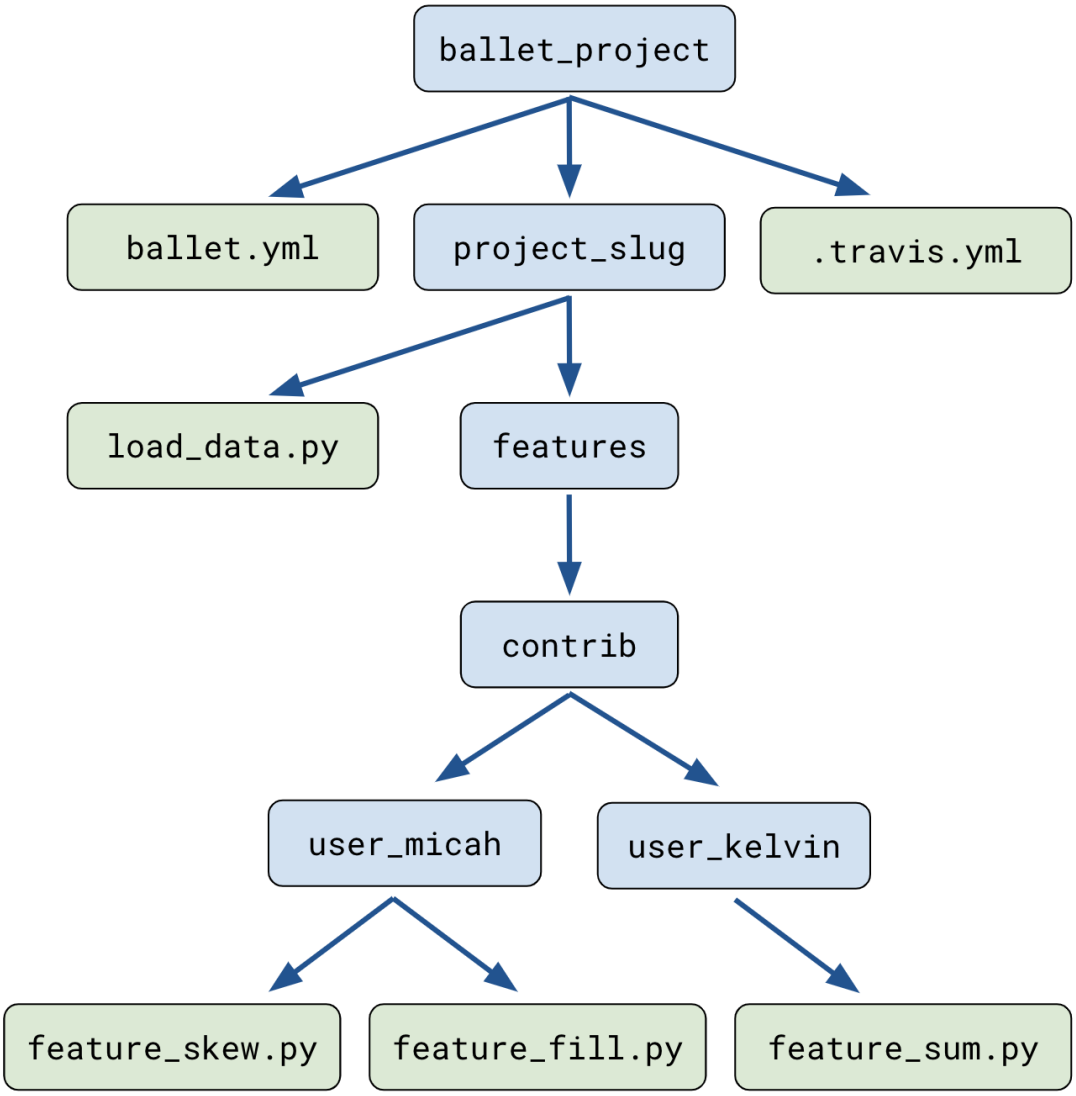

Figure 2-1: The example ballet project structure of a project called "`ballet_project`" and with slug (or package name) "`ballet_slug`". Blue elements are directories with pointers to their contents. Green elements are file objects. This project has 2 contributors (micah, kelvin) and 3 features.

2.2.2 Startup

On startup, `Ballet` renders a new repository from a template, with the three main components. It is then registered with GitHub and a CI provider, usually Travis CI for an open-source project. They will also be directed to install the `Ballet` GitHub app, which provides more automation for the project.

When a contributor joins a project, `Ballet` provides a template to write a new feature. This helps ensure API consistency and ease of development. For both maintainers and contributors, the documentation provides a quick-start command to teach both how to write and to identify quality features and also best practices in feature engineering.

2.2.3 Feature Engineering Life-Cycle

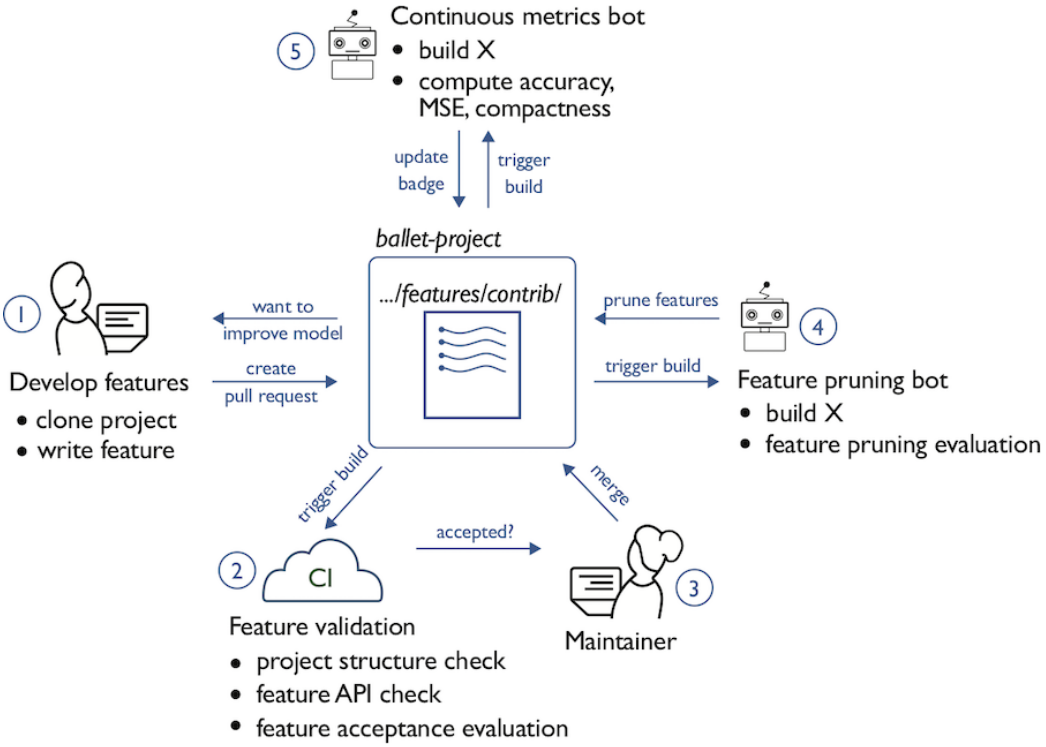

Figure 2-2: The life-cycle of a new feature, from proposal to impact on the model to potential removal [16]. The focus of this thesis is on steps 2 through 4.

The **Feature Engineering Life-Cycle** is the process by which features go from being proposed to contributing to the feature matrix. This also includes the process

by which accepted features are removed to ensure only the highest quality features are kept by the project. This is illustrated in Figure 2-2 and detailed in the following section.

1. Feature Development At any point, data scientists can observe the current performance of the pipeline and be motivated to write new features and contribute them to the repository. To contribute a new feature, collaborators first write source code that instantiates a single Feature object. They then submit their code to the project through one of several mechanisms, depending on their skill level and background. For contributors who desire maximum control, they can manually create new source files in the correct sub-directories, commit the addition to their own fork of the project, and open a new pull request on GitHub. Those who desire an interactive workflow can import a Python client library that automatically generates source code to recreate a live Feature object and creates a new feature proposal under the hood.

2. Feature Validation Once a feature has been proposed by a collaborator, `Ballet` must ensure that only high quality features are merged into the project and that the feature engineering pipeline invariant is kept. `Ballet` enforces quality using automated CI testing that consists of several checks. These will be further detailed in the following section, as it discusses the automated portions of the feature engineering life-cycle.

3. Maintainer Validation After Travis CI finishes the automated feature validation step, it is time for maintainers to review the features before they are merged to the repository. While the Travis CI tests ensure the feature follows the API and is of high quality, it is the responsibility of the maintainer to ensure the code quality is to standard — a feature can follow the API and still follow poor coding practices.

Because this step runs after the Travis CI automated testing, there are fewer features that reach the maintainer review step than actually are proposed. This allows `Ballet` to be scalable even if there are few maintainers to watch over the project. This also ensures the maintainer has less work to do per review.

4. Feature Pruning Bot After a feature is merged, another set of Travis CI tests trigger. This time, the tests are not meant to target the newly merged feature but

rather the features previously accepted to the project. It is possible that the addition of the new feature has made another feature redundant. This step runs redundancy checks against previously accepted features and then suggests features for removal if they are found to be redundant.

Similarly to the Feature Acceptance step, this step uses a variation of the GF-SSF feature selection algorithm to determine redundancy. This is further detailed in Chapter 3. After Travis CI tests suggest features for removal, a GitHub Application handles removing the features from the repository. Depending on the configuration of the project, this application may either directly commit to the master branch, propose the removal in a pull request, or simply do nothing. The level of automation is dependent on the maintainer and how involved they wish to be in this process.

5. Continuous Metrics Finally, a continuous metrics bot rebuilds the pipeline, evaluates a predictive model, and updates metrics like accuracy and feature compactness. This provides insights as to how effective the accepted features are for predictive models. This feature engineering life-cycle is continued for new features and the project features are gradually improved.

Chapter 3

Feature Selection Process

The feature selection process includes the specific parts of the feature engineering life-cycle that ensure that our features are "high quality" and meet the relevance and compactness standards that `Ballet` requires. In this section, we go over the main feature selection algorithm, Grouped Feature Selection for Streaming Features (GFSSF) [13]. We focus on its implementation in `Ballet`, specifically how it was modified to fit the needs of the framework. We begin by discussing efficient entropy and information estimation for data-sets, which are important for use in feature selection algorithms.

3.1 Mutual Information and Entropy

Definition. The **Shannon Entropy** of a discrete random variable X with support $\mathbb{X}$ and probability distribution $P(x)$ is $H(X) = -\sum_{x \in \mathbb{X}} P(x) \log(P(x))$

Definition. The **Differential Entropy** of a continuous random variable X with support $\mathbb{X}$ and probability distribution $P(x)$ is $H(X) = -\int_x P(x) \log(P(x)) dx$

Definition. The **Mutual Information** between two random variables X, Y is defined as $I(X;Y) = H(X) + H(Y) - H(X,Y)$, where $H(X)$ is the entropy of the random variable. Alternatively, mutual information between variables X, Y with supports $\mathbb{X}, \mathbb{Y}$, individual probability distributions $P_X(x), P_Y(y)$, and joint distribution $P_{XY}(x,y)$ can be calculated as:

$$I(X;Y) = \sum_{x \in \mathbb{X}} \sum_{y \in \mathbb{Y}} P_{x,y}(x,y) \log(\frac{P_{XY}(x,y)}{P_X(x)P_Y(y)}) \tag{3.1}$$

Definition. The **Conditional Mutual Information** between two random variables X, Y given a third random variable Z is defined as $I(X;Y|Z) = H(X,Z) + H(Y,Z) - H(X,Y,Z) - H(Z)$, where $H(X)$ is the entropy of the random variable.

From their definitions, Shannon and Differential entropies are both non-negative quantities and are measures of how difficult it is to predict the next drawn value from a particular random variable. By the definition and properties of entropy, mutual information and conditional mutual information are also non-negative quantities that depend on the joint distributions of the random variables.

Intuitively, mutual information is a measure of how much "information" we learn about Y if we know X and vice versa. Similarly, conditional information is a measure of how much information we learn about Y from knowing X, on top of any information we already know about Y from Z. We see from equation 3.1 that if two variables X, Y are mutually independent, then we have $P_{XY}(x,y) = P_X(x)P_Y(y)$ and that the mutual information is zero. Similarly, if X and Y are conditionally independent given Z, then the conditional mutual information $I(X;Y|Z)$ is also zero.

3.1.1 Mutual Information as a Statistic for Relevance

From the previous definitions, we can draw analogies between mutual information and relevance. For a data-set $\mathcal{D}$ with dimensionality $N \times d$, we assume that each of the n samples is drawn i.i.d. from a probability distribution $P(x)$ such that each sample $x \in \mathbb{R}^d$ has d columns. Then, we can make the approximation that each logical feature creates a feature matrix $f_i(\mathcal{D})$ whose rows are drawn i.i.d. from a distribution.

Definition. A feature f_i is **irrelevant** towards a target column y if $I(f_i; y) = 0$

Definition. A feature f_i is **strongly relevant** towards a target column y, given a set of previously accepted features $\mathcal{F}$, if $I(f_i; y|\mathcal{F}) > 0$.

Definition. A feature f_i is **weakly relevant** towards a target column y, given a set of previously accepted features $\mathcal{F}$, if it is neither strongly relevant nor irrelevant.

As such, we can use mutual information as a useful statistic for feature relevance and selection. Our goal is to select the best possible feature set $\mathcal{F}_t$ as features come in a streaming fashion, using mutual information-based heuristics to determine which features to accept and which to reject.

Because we cannot observe the distributions directly and instead are estimating from the feature columns, it is often the case that the information of an irrelevant feature is non-zero only due to estimator and rounding errors. As such, we generally test to see if the information $I(f_i; y|\mathcal{F})$ is above a threshold I_T instead of zero. The definition of this threshold allows us not only to reduce estimator errors, but also to penalize high-dimension features and set a "minimum information contribution" a new feature must provide to be accepted.

3.2 Lightweight Entropy Estimation in Ballet

From their definitions, the most flexible way of calculating mutual information and conditional mutual information is by calculating the entropies of each variable. Therefore, an important subroutine for the many feature selection algorithms is the calculation of **data entropy**. For these data-sets, we assume that they consist entirely of *real-valued numbers* such as a feature matrix. If we assume each matrix is drawn i.i.d. from a probability distribution, the task is to estimate this distribution's entropy from the samples.

However, this is a challenge in `Ballet`, as a feature matrix can contain both columns with discrete values and columns with continuous values. We make this distinction between continuous and discrete random variables because some columns (e.g. one-hot encoding columns, categorical variables, etc.) make more sense as discrete random variables and others (e.g. columns representing a person's height, house price, etc.) as continuous random variables. Furthermore, applying the wrong estimation process for a column can yield wildly different results, so we must consider

both types of entropies for the best accuracy. We will refer to both Shannon entropy and differential entropy as "entropy" when referring to a data-set that contains both logically discrete and logically continuous columns.

3.2.1 Entropy of Fully Discrete Data-Set

Definition. The **Empirical Probability** $P^{\mathcal{D}}(x)$ of a value x in a data-set $\mathcal{D}$ is the frequency of appearance of x in $\mathcal{D}$.

If a data-set $\mathcal{D}$ contains only discrete columns, we can estimate the entropy by estimating the probability distribution $P(x)$. For each unique data entry $x \in \mathcal{D}$, we can estimate the empirical probability $P(x)$ by counting the number of instances that entry appears in the data-set and then dividing by the total number of entries. This provides an efficient and accurate method of calculating the data-set distribution and in turn the data-set entropy.

3.2.2 Entropy of Fully Continuous data-sets

If, instead, our data-set contains only continuous columns, we need to find a way to estimate the integral $H(X) = -\int_x P(x)\log(P(x))dx$. While many other algorithms opt to roughly bucket values and treat them similarly to discrete data-sets, we have chosen to use better entropy estimators.

Another estimator for continuous data-sets is the **Kozachenko-Leonenko Entropy Estimator** [11], or K-L estimator for short, which estimates a data point's "bucket" size using its k-nearest neighbors. If a point's neighbors are far away, then the data-set is more spread out and entropy is higher, whereas a data point with close neighbors contributes much less entropy. This allows for more accurate entropy estimation with a smaller asymptotic bias [5].

Definition. The **digamma function** $\psi(x)$ is defined as the logarithmic derivative of the gamma function:

$$\psi(x) = \frac{d}{dx}\ln(\Gamma(x)) = \frac{\Gamma'(x)}{\Gamma(x)} \tag{3.2}$$

Where the Gamma Function $\Gamma(x)$ is the extension of the factorial function to real numbers and follows the recursion $\Gamma(x) = x\Gamma(x-1)$.

Definition. For a continuous data-set $\mathcal{D}$ with dimensionality $N \times d$ and data entries $x_1, x_2, \ldots, x_N$, the **Kozachenko-Leonenko Estimator** for data-set entropy is calculated as:

$$H_k^{KL}(\mathcal{D}) = -\psi(k) + \psi(N) + \frac{d}{N}\sum_{i=1}^{n}\log(\epsilon_i), \tag{3.3}$$

where ϵ_i is twice the distance from x_i to its k-th nearest neighbor and $\psi(x)$ is the digamma function. For this estimator, k is a hyper-parameter that should be set before-hand.

For the k nearest neighbors, we use the Chebyshev metric (maximum norm) for calculating distance. This allows for easier computation of the estimator and because the Chebyshev metric is used later on for other estimators. The use of the digamma function and the derivation of this estimator can be found in [11] and [5].

3.2.3 Entropy of Mixed data-sets

It is often the case that a data-set will consist of some columns that represent discrete variables and some will represent continuous features. It is important that we are able to accurately estimate the entropy of such cases as well.

Definition. Consider a continuous random variable X and a discrete random variable Y of dimensions d_x, d_y and supports $\mathbb{X} = \mathbb{R}^{d_x}$, $\mathbb{Y} = \mathbb{Z}^{d_y}$ respectively. If they have joint distribution $P_{X,Y}(x, y)$, then the **Joint Entropy** is defined as:

$$H(X, Y) = -\sum_{y \in \mathbb{Y}} \int_x P_{X,Y}(x, y)\log(P_{X,Y}(x, y))dx \tag{3.4}$$

Using the definition of conditional probability that $P_{X,Y}(x, y) = P_{X|Y}(x|y)P_Y(y)$, we can rearrange our equation to be:

$$H(X,Y) = -\sum_{y\in\mathbb{Y}} \int_x P_Y(x) P_{X|Y}(x|y) \log(P_{X|Y}(x|y)) dx \\ - \sum_{y\in\mathbb{Y}} P_Y(y) \log(P_Y(y)) \tag{3.5}$$

We see that we can simplify this equation by using the definitions of Shannon and differential entropy. Let $H(X|Y=y)$ be the entropy of the random variable Y, conditioned on X's value. Note that this is similar to but different from conditional entropy. Using this definition and the previously defined entropies, we have:

$$H(X,Y) = \sum_{y\in\mathbb{Y}} P_Y(y) H(X|Y=y) + H(Y) \tag{3.6}$$

We can take advantage of the last equation to calculate the entropy of a mixed data-set. Let our data-set $\mathcal{D}$ of dimensions $N \times d$ be partitioned into a purely continuous data-set $\mathcal{X}$ with dimensions $N \times d_x$ and a data-set of discrete values $\mathcal{Y}$ with dimensions d_y. For each unique entry $y_i \in \mathbb{Z}^{d_y}$ in the discrete data-set, find the subset of the continuous data-set with entries at the same index, $\mathcal{X}_{y_i}$. This data-set has the underlying probability $P(X|Y=y_i)$. Estimating the continuous entropy of each subset and multiplying it against the empirical probability of y_i in the discrete data-set gives us $\sum_{y\in\mathbb{Y}} P_Y(y) H(X|Y=y)$, and $H(Y)$ can be estimated directly from the discrete portion $\mathcal{Y}$.

3.3 Mutual Information Estimation

To find the mutual information between two data-sets $\mathcal{D}_1, \mathcal{D}_2$, we need to calculate the entropies of three data-sets: $\mathcal{D}_1$, $\mathcal{D}_2$, and the data-set made by concatenating the two data-sets. While the K-L estimator provides an acceptable entropy estimator for a single data-set, this estimator is imperfect when calculating mutual information between two data-sets as the biases from each estimator do not cancel out [12]. Therefore, for continuous entropy estimation in the context of mutual information

estimation, there is a motivation to find a better estimator.

3.3.1 Kraskov Estimator

Let there be two continuous data-sets $\mathcal{D}_1, \mathcal{D}_2$ with dimensions $N \times d_1$ and $N \times d_2$. Let $\mathcal{D}$ be the data-set with dimension $N \times (d_1 + d_2)$ created by concatenating the two data-sets together. Similarly to the K-L estimator, define ϵ_i as twice the distance of a data entry x_i to its k-th nearest neighbor in $\mathcal{D}$. Like the K-L estimator, k is a hyper-parameter.

Finally, let $k_i(\mathcal{D})$ represent the number of data entries within a distance ϵ_i to x_i in the data-set $\mathcal{D}$. Note that this varies from data-set to data-set: for $\mathcal{D}$, $k_i(\mathcal{D}) = k$ for all N data points by definition. However, it is different for $\mathcal{D}_1, \mathcal{D}_2$ as the ϵ_i are calculated from

Definition. The **Kraskov Estimator** [12] for continuous data-set entropy of a dataset with dimensions $N \times d$ is defined as:

$$H_k^K(\mathcal{D}) = -\frac{1}{N}\sum_i \psi(k_i(\mathcal{D})) + \psi(N) + \frac{d}{N}\sum_i \log(\epsilon_i) \tag{3.7}$$

Essentially, each data-set $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}$ uses the same ϵ_i values. In practice, this is an improvement in estimator bias for mutual information. Additionally, a partial estimator may also be calculated to save computation costs:

Definition. The **Partial Kraskov Estimator** [12] for continuous data-set entropy is defined as:

$$H_k^P(\mathcal{D}) = -\frac{1}{N}\sum_i \psi(k_i(\mathcal{D})) + \psi(N) \tag{3.8}$$

We see that this entropy estimator has bias $b(H^P(\mathcal{D})) = \frac{d}{N}\sum_i \log(\epsilon_i)$. While this is problematic for a single data-set entropy estimation, this bias cancels out when calculating mutual information. Let $b(*)$ be the bias of an estimator. We see that the bias of a mutual information estimator based on the partial estimator is:

$$
\begin{aligned}
b(I^P(\mathcal{D}_1;\mathcal{D}_2)) &= b(H^P(\mathcal{D}_1)) + b(H^P(\mathcal{D}_2)) - b(H^P(\mathcal{D})) \\
&= \frac{d_1}{N}\sum_i \log(\epsilon_i) + \frac{d_2}{N}\sum_i \log(\epsilon_i) - \frac{d_1+d_2}{N}\sum_i \log(\epsilon_i) \\
&= 0
\end{aligned}
$$

Relative to the original Kraskov Estimator, the Partial Estimator for mutual information provides no extra bias with the benefit of cutting down computation. As such, we opt to use the Partial Estimator when calculating mutual information.

As alluded to before, the Kraskov Estimator necessarily uses the Chebyshev metric for calculating k nearest neighbors distance. As such, all knn's in `Ballet` employ this metric as well.

3.4 GFSSF

These estimators for entropy and mutual information give us the tools necessary to introduce GFSSF. Grouped Feature Selection with Streaming Features [13], or **GFSSF**, is a streaming feature selection algorithm that uses mutual information as a proxy for feature relevance. This algorithm uses a heuristic based on both the information the feature provides and the size of the feature, to penalize unnecessarily large feature matrices. We use a variant of GFSSF in Ballet to test for feature relevance and redundancy, changing steps to fit the constraints of our system.

3.4.1 GFSSF Algorithm Outline

When a new feature $f_t(\mathcal{D})$ of dimensionality $q_t \times n$ is proposed to GFSSF, it is tested against the target column y as well as the currently accepted features $\mathcal{F}_t$. If the feature is found to be strongly relevant or weakly relevant, it will be accepted based on its size and relationship with the currently accepted features.

To test for strong relevance, GFSSF calculates the conditional mutual information $I(y; f_t(\mathcal{D})|\mathcal{F}_t(\mathcal{D}))$ between the target and the candidate feature given the current accepted. Intuitively, this is the new information f_t provides about y over the current features. We compare this information statistic against a threshold $I_T = \lambda_1 + \lambda_2 \times q_t$ and accept if the information is higher. λ_1 and λ_2 are user-specified hyper-parameters that will be discussed more in the implementation of GFSSF in `Ballet`. This threshold increases as the dimensionality of the candidate feature increases, so this penalizes large features and ensures that the feature matrix remains relatively small in dimension.

However, if a candidate feature is not strongly relevant we may still want the feature in replacement for a currently accepted feature. We may find that the candidate feature is a more relevant version of the old feature, or it expresses the same information in fewer columns. If GFSSF finds that a feature is not strongly relevant, it will iterate through all features $f_i \in \mathcal{F}_t$ to find a "weaker" feature. To do so, it looks for a feature such that $I(f_t(\mathcal{D}); y|\mathcal{F}'_t(\mathcal{D})) - I(f_i(\mathcal{D}); y|\mathcal{F}'_t(\mathcal{D})) > \lambda_2 \times (q_t - q_i)$, where q_i and q_t are the dimensions of f_i and f_t and $\mathcal{F}'_t = \mathcal{F}_t \setminus f_i$ is the set of all accepted features minus f_i, the candidate for removal. If such a feature is found, it is removed and the candidate feature is accepted.

Once a feature is accepted, all accepted features are now pruned for redundancy. Each feature is checked to make sure to be strongly relevant; that is, that for each $f_i \in \mathcal{F}$, that it satisfies $I(f_i(\mathcal{D}); y|(\mathcal{F} \setminus f_i)(\mathcal{D})) > \lambda_1 + \lambda_2 \times q_i$. The full algorithm can be viewed in Algorithm 1.

3.4.2 Modifications and Implementation of GFSSF

As originally described, the GFSSF algorithm cannot be built in the `Ballet` framework. There are two main reasons for this; one reason is that the original algorithm proposed removing columns from each feature, which is near-impossible to do automatically in `Ballet`. The other is because `Ballet` requires a separation between acceptance and pruning, which GFSSF also does not have.

The original GFSSF algorithm had two parts: InGFSSF and AgGFSSF. The In-

input : A stream of feature columns $\Gamma = f_1, f_2, \ldots, f_t$
output: $\mathcal{F}$, a compact and relevant feature matrix

$\mathcal{F} = \{\}$;
for $f_t \in \Gamma$ **do**
  if $I(f_t; Y|\mathcal{F}) > \lambda_1 + \lambda_2 \times q_t$ **then**
    $\mathcal{F} = \mathcal{F} \cup f_t$;
  else if $\exists f_j$ *s.t.* $I(f_t; Y|\mathcal{F} \setminus f_j) - I(f_j; Y|\mathcal{F} \setminus f_j) > \lambda_2 \times (q_t - q_j)$ **then**
    $\mathcal{F} = \mathcal{F} \cup f_t \setminus f_j$;
  else
    continue;
  end
  for $f_j \in \mathcal{F}$ **do**
    if $I(f_j, Y|\mathcal{F}) < \lambda_1 + \lambda_2 \times q_j$ **then**
      $\mathcal{F} = \mathcal{F} \setminus f_j$;
    end
  end
end

Algorithm 1: AgGFSSF routine for GFSSF

GFSSF subroutine compared columns inside a logical feature and deleted columns found to be irrelevant, thus making each feature more compact. However, it is infeasible within `Ballet`; to automatically "delete" feature columns in a logical feature would require directly changing the source code of the `Feature` object. This would either require having the collaborator or maintainer change the code themselves or find a way to automate source code editing. Both are cumbersome, as the former adds an extra step for a human agent and the latter adds machine generated code to a repository — often with poor readability for other collaborators. As such, the GFSSF implemented in `Ballet` and described earlier is the AgGFSSF subroutine and the InGFSSF subroutine is skipped entirely.

The other issue with the original GFSSF algorithm is that there is not a good separation between acceptance and pruning. In the original algorithm, it is possible that an accepted feature gets pruned while evaluating a candidate feature for acceptance. This is not practical for `Ballet`, as it may be the case that the maintainer does not want to remove redundant features.

To solve this issue, we modify the algorithm slightly: when a feature f_j would be pruned during the acceptance stage of a candidate feature f_t, we instead keep it. In

the feature pruning stage, we also enforce that the newly accepted feature is not a candidate for pruning. The second constraint is necessary, as in the original algorithm the newly accepted f_t is only weakly relevant while f_j is still in the feature set. We expect the performance and behavior of this modified GFSSF algorithm to be similar to the original AgGFSSF algorithm, as we expect f_j to be pruned in the pruning stage anyways. In Algorithm 2, we now have a concise splitting of the algorithm between a "acceptance" stage (lines 3 — 9) and a "pruning" stage (lines 10 — 14), which we implement separately in different stages of `Ballet` as a `GFSSFAcceptanceValidator` and `GFSSFPruningValidator`, respectively.

Finally, we also wanted to place a higher value on features that were already accepted over newly proposed features; in line 5 of the original algorithm (algorithm 1), there is a possibility that the candidate feature f_i is almost redundant with an accepted feature f_j, such that $I(f_i; Y|\mathcal{F} \setminus f_j) - I(f_j, Y|\mathcal{F} \setminus f_j)$ is slightly above zero. In the original algorithm, f_i would replace f_j.

In the context of `Ballet`, we want to avoid this situation because contributors are likely to write logical features that are very similar to one another. In this situation, we would want to keep the older feature if the new feature does not provide substantial information for several reasons; firstly, we would want to rightly credit the original contributor's work by keeping their work in the project. additionally, we would want to avoid needless "feature swapping" as it would be confusing for maintainers to see changes to the project that did not improve model performance. To solve this, we change the threshold in line 5 from $(\lambda_2 \times (q_i - q_j))$ to $(\lambda_1 + \lambda_2 \times (q_i - q_j))$, so that f_i is only accepted if it provides either more information or the same information in fewer columns.

3.4.3 Hyper-Parameter Specification

GFSSF relies on two hyper-parameters to be specified: λ_1 and λ_2 which can be understood as a relevance threshold and a dimension penalty, respectively. As we would not expect maintainers to know the intricacies of the GFSSF algorithm prior to starting a `Ballet` project, these parameters are set by `Ballet` itself.

input : A stream of feature columns $\Gamma = f_1, f_2, \ldots f_t$
output: $\mathcal{F}$, a compact and relevant feature matrix

```
1  F = {};
2  for f_t ∈ Γ do
3  |  if I(f_t, Y|F) > λ1 + λ2 × q_i then
4  |  |  F = F ∪ f_t;
5  |  else if ∃f_j, s.t. I(f_t, Y|F \ f_j) − I(f_j, Y|F \ f_j) > λ1 + λ2 × (q_t − q_j)
   |  |  then
6  |  |  F = F ∪ f_t;
7  |  else
8  |  |  continue;
9  |  end
10 |  for f_j ∈ F \ f_t do
11 |  |  if I(f_j, Y|F) < λ1 + λ2 × q_j then
12 |  |  |  F = F \ f_j;
13 |  |  end
14 |  end
15 end
```

Algorithm 2: Ballet-GFSSF, modifications in line 5 and 6

The original authors opted to use $\lambda_1 = \frac{H(y)}{4|\Gamma_t|}$ and $\lambda_2 = \frac{H(y)}{4(\sum_{f_i \in \Gamma_t} q_i)}$, where Γ_t is the set of all *arrived* features, $H(y)$ is the entropy of the target column y, and 4 is a normalization constant found after experimentation. Thus, $|\Gamma_t|$ is the number of arrived features and $\sum_{f_i \in \Gamma_t} q_i$ is the total number of columns amongst arrived features. The notion of a "rejected" feature is fuzzy in `Ballet`— Even if a pull request is closed, the user can open it again with modifications to their feature - in which case it is difficult to determine if this feature should count towards the number of arrived features twice.

As such, we opted to replace Γ_t with $\mathcal{F}_t$: the set of *accepted* features. This is better defined for a `Ballet` project and is easier to query. As we expect most features to be rejected, we also increased the normalization constant from 4 to 32. Similar to the original paper, we experiment with changing this value and will discuss this later.

3.4.4 Other Considered Algorithms

In the process of developing `Ballet`, the GFSSF algorithm was not the first algorithm we considered. Many other feature selection algorithms were considered and

ultimately not used due to some of the constraints that `Ballet` imposes in terms of feature quality and memory usage. We detail them here to explain why we use GFSSF as our baseline algorithm even though there are arguably more performant algorithms in the state of the art.

Alpha Investing: α-investing [22] is an algorithm that uses a likelihood ratio test and compares it against a parameter α to see if it has enough relevance to be accepted. The core idea is that the α parameter changes as features are accepted or rejected to fit the needs of the feature matrix: we are more lenient towards low-quality features if we have recently accepted many high-quality features, and we are more picky about feature selection if we have recently rejected many low-quality features.

This is an attractive algorithm for `Ballet` because the hyper-parameter α and the adaptive penalty allows `Ballet` to upper bound the false positive rate of accidentally accepting a poor quality feature. Furthermore, there is flexibility in the type of statistic test used, and can be adapted for different uses. In our previous work [16], we found that α-investing did comparably to a batch algorithm trying to accomplish the same goal.

In the end, we decided against alpha investing due to **availability** constraints - because `Ballet` projects exist as GitHub repositories and candidate features are proposed as pull requests, it is difficult to define the idea of "rejecting" a feature as in section 3.4.3. Additionally, this requires the `Ballet` project to keep track of the α parameter as it changes in time. This would pose an issue for security storing this value of α; if it is stored within the project repo, it is vulnerable to tampering by malicious contributors. However, more secure solutions would require adding more infrastructure to `Ballet`, increasing the complexity of startup and potentially adding cost to maintaining a `Ballet` project.

Group-SOALA: Group-SOALA [21] is a streaming feature selection algorithm that is similar to GFSSF in that it uses mutual information as a proxy for relevance. However, instead of considering the conditional mutual information of an entire feature, Group-SOALA tests a feature column by column against a set of heuristics to see if each column is relevant. As entropy calculation scales poorly with dimension, this is

significantly more efficient than GFSSF.

However, we opted against Group-SOALA due to constraints of the `Feature` abstraction and the feature engineering pipeline invariant: instead of penalizing high dimension features, Group-SOALA opts to remove irrelevant feature columns. This would have been cumbersome for our users but without would potentially mean accepting many irrelevant feature columns, so we opted against this method as well.

3.5 Feature Selection Architecture

There are three main agents in the feature selection that work together to ensure that the project invariant is kept. We will go over an overview of all three parts and their roles, and discuss their detailed interactions in the following section.

3.5.1 git/GitHub

`Ballet` projects live as git repositories, developed on top of GitHub. As discussed earlier, this is because GitHub is already used by open source developers and provides a platform for version control and incremental development. As `Ballet` projects are meant to be open-source projects and GitHub allows for the free creation of public data-sets, the expected use case of `Ballet` incurs no cost to maintainer or contributor.

In the Feature Selection process, GitHub maintains permissions so that only maintainers make the final decision to merge a proposed feature into the project. GitHub is also responsible for triggering builds and web-hooks to Travis CI and the GitHub App, respectively.

3.5.2 Travis CI

Travis CI is a continuous integration service for software projects such as `Ballet` projects. They provide free CI testing and computation for open source projects, which allows for open source projects to grow at scale without worrying about the cost of CI testing.

For `Ballet` projects, Travis CI is responsible for running the feature validation checks to ensure candidate features follow the API and are sufficiently relevant. For each pull request and merge, Travis CI runs a check suite where each job is a different stage of validation for features, including the GFSSF acceptance and pruning stages. As Travis CI provides free services for open-source project, this is the most sensible part of the architecture to run the computationally intensive GFSSF algorithm.

3.5.3 GitHub App

Part of `Ballet` is a GitHub App custom made to interact with `Ballet` projects. GitHub Apps are services run by organizations that, when installed by a user or organization onto their repositories, listens to specific events triggered by other GitHub services and acts inside that repository as its own agent. For example, Travis CI is implemented as a GitHub app that listens to when pushes are made on pull requests and then can modify the pull request's check suite accordingly.

The `Ballet` GitHub App is in charge of miscellaneous automation tasks related to GitHub. Currently, one of its larger roles is taking action for pruning - after the Travis pruning check finishes running, the GitHub App takes action to remove the redundant features. As this app runs as a server owned by the Data to AI group, we want to ensure that the app preforms as few computations as possible. We discuss how we achieve this later on in how it interacts with Travis and GitHub.

It is important to note that the GitHub App runs on a server owned by the Data to AI group and therefore incurs a cost to the lab and *no cost to the maintainer of the project*. Currently, the App is hosted on an Amazon Web Service computation instance. As we avoid heavy computation on these servers, we use a relatively small machine that incurs a cost of $10/month. These can also be switched to computers owned directly by the Data to AI group if such costs become infeasible to support.

3.6 Feature Acceptance

The feature acceptance process can be broken into three concrete steps: Travis CI testing, maintainer review, and an optional GitHub app step to automate feature acceptance.

First, when a feature is proposed in a pull request, GitHub triggers Travis CI to start a build. For the acceptance build, this build has three main checks:

a. **Project Structure Check** Collaborators that propose features are only allowed to do so in the `contrib` folder in the correct sub-directory, and nowhere else. This prevents users from tampering with the project metadata or how the project loads data. To do so, the first set of checks creates a diff of what files have been changed in the pull request. We expect that each feature proposal contains exactly one module that exports a `Feature` object. If not, the pull request is rejected.

b. **Feature API Check** In this stage, the candidate feature is fitted and transformed against the data-set provided in `load_data.py`. If errors are raised while fitting or transforming, the check fails as the feature is not able to to produce a viable feature matrix. Afterwards, the resulting feature matrix is tested against a set of heuristics to ensure predictive models will be able to use it - these checks include a check for no missing values and no infinite values.

c. **Feature Acceptance Check** Even if a feature follows the API, it may not be relevant to the target column and therefore useless for any predictive model. The Feature Acceptance Check set uses GFSSF to determine whether or not a feature is either strongly or weakly relevant, and then determines whether it has enough relevance to be accepted.

These three checks are built as sub-modules into the `ballet.validation` module - specifically, Ballet provides a script for Travis that calls either a `FeatureAPIValidator`, `ProjectStructureValidator`, or `GFSSFAcceptanceValidator` depending on the check

being run. Once all the checks have finished running, Travis sends the results back to GitHub to display on the pull request.

On GitHub, the maintainer then reviews feature pull requests that pass the checks. While Travis ensure the feature passes the `Ballet` project invariant, there may be other invariants the maintainer would like to uphold - for example, good documentation of code or good coding practices.

Optionally, the maintainer may configure the project to have the GitHub app automate more parts of the acceptance process. If the maintainer does not have enough time to review every passing feature, they may opt to have the GitHub app automatically accept and merge these features into the project. The application can also close proposed features that fail specifically the acceptance check - while it is relatively easy to fix the API and structure checks, it is difficult to increase a feature's relevance.

3.7 Feature Pruning

Once a candidate feature is accepted into the project, the feature pruning process is triggered. Travis and the GitHub app are mainly responsible, while GitHub itself provides web-hooks that trigger and notify each service. The numbers below correspond to the same step on Figure 3-1.

1. The first step of the pruning process is the merging of the new feature - this is the only event that triggers the process.

2. Once a feature is merged into the project, GitHub triggers a build on Travis. For pruning, the feature validation checks are ignored and only a feature pruning check is run, using the `ballet.validation.GFSSFPruningValidator` object. As Travis itself cannot prune features from the project, it instead stores the output of the pruning routine into its logs and returns a "passing" check back to GitHub.

3. Once GitHub receives a "passing" check, it sends a web-hook to the GitHub

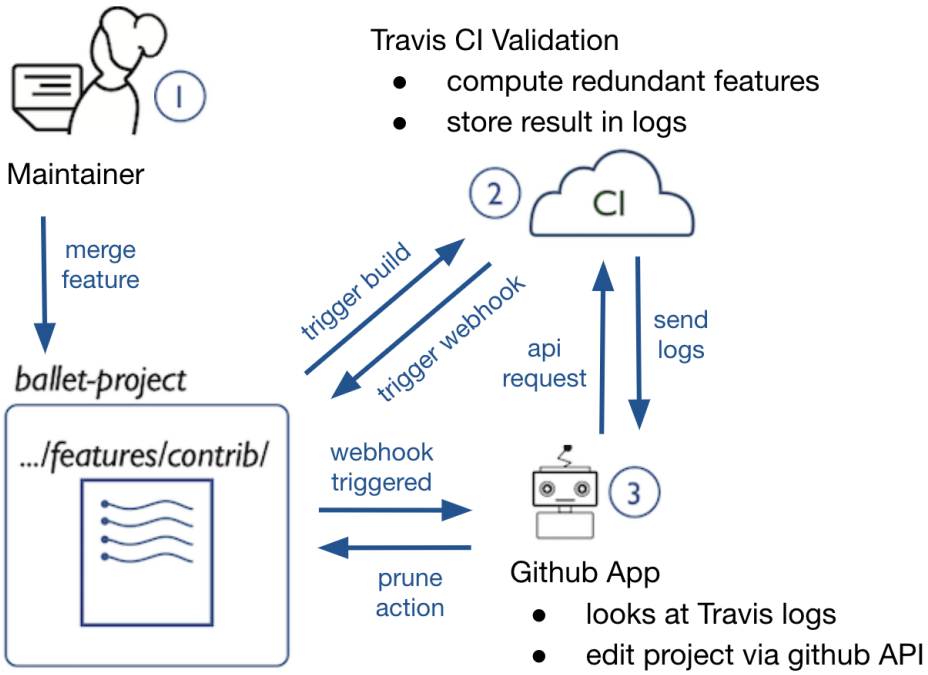

Figure 3-1: The pruning process, starting from the merging of the candidate feature to the pruning of the redundant features.

app. On receiving this, the app uses the Travis API to retrieve the logs of the pruning routine. It then parses the log and figures out which features to remove. The app can then create a commit using the GitHub API on top of the master branch that removes the redundant features if any exist. From here, the next action depends on the project configuration: the app can either update master to this commit, add the commit to a new branch and submit a PR, or do nothing.

Figure 3-2: An example output of a Travis CI log for a pruning subroutine. Highlighted in green are log lines pertaining to features that passed and stayed in the project, while highlighted in red are log lines pertaining to features that failed and were pruned. Features whose information score did not exceed the threshold (a statistic based on the current features and size of the feature q_i) were pruned.

Chapter 4

Simulation and Evaluation

To test the effectiveness of `Ballet` and how well it aligned with our design goals, we simulated what collaboration in a `Ballet` project might be like with multiple collaborators. To do so, we took real-life instances of group-based data science and simulated what the process would have been like in `Ballet` instead of the original context. Then, we analyze the performance of the feature matrix `Ballet` provides and compare the feature engineering process in `Ballet` with the original context. Because we run the `Ballet` project as a simulation of collaboration, we will focus design goals related to data science and scalability over goals related to collaboration, as there are no real human agents collaborating.

4.1 Ames Housing Prediction Overview

The Ames Housing Prediction Competition [4] is a popular starting problem for beginner data scientists on Kaggle. The goal of the problem is to accurately predict the price of a house given different qualities of the house, such as square footage of the first floor and the age of the house.

To do so, users are given a single data table containing a target column for house prices in Ames, Iowa and many columns pertaining to different qualities of the house. This is where feature engineering is important; the data-set contains many missing values that should be filled in. Also, data-set columns are potentially redundant or

irrelevant - for example, a column pertaining to the type of pool a house had ended up being useless for predictive models because a majority of houses did not have a pool. The relevant task for a `Ballet` project designed for the Ames Prediction Problem is to take the original data table and transform it into a compact and more relevant feature matrix.

As discussed earlier, Kaggle provides kernels for users to share and discuss their data science methods with the Kaggle community. As the Ames Prediction Problem is a practice problem that has been online for over two years, many kernels have been built describing different features that users have found to be relevant to house price. Users are allowed to use the works of other kernels to build their own, which provides a simple collaboration model for this problem.

4.2 Experiment Details

As the kernels allow for users to publicly share their work, we wanted to simulate what their work would have been like if they were working collaboratively on a `Ballet` project.

To do so, we identified nine kernels for the Ames Prediction Problem that were popular in the community and had a focus on feature engineering. Each kernel was made by a different Kaggle user, so this would translate to nine different users contributing features in a `Ballet` project. Once we identified the kernels, we set about translating each feature described to fit the `Ballet` API. This involved reading each kernel, identifying each feature, and manually implementing a `Feature` object that performed the same transformations on the data-set that the original kernel feature did. Overall, we identified 311 logical features from the eight kernels.

Once we had our `Feature` objects implemented, We repeatedly simulate a scenario in which each Kaggle member separately submits their features to a Ballet project. In each scenario, we simulate collaboration by having users submit features one at a time at random intervals. We then allow for the `Ballet` app to automatically merge or reject pull requests based on the feedback from the validation step, then allow the

app to automatically remove features found redundant after a new feature is accepted.

4.3 Key Findings

While analyzing the simulation of the Ames demo we found some interesting discoveries that helped us understand better the Ames data-set and the feature selection algorithm.

Of the original 311 logical features proposed to the data-set, 12 features were accepted into the project when they were proposed. 6 of those features were eventually pruned by new features entering the project, resulting in 6 total features left by the end of the simulation.

4.3.1 Mutual Information Limit

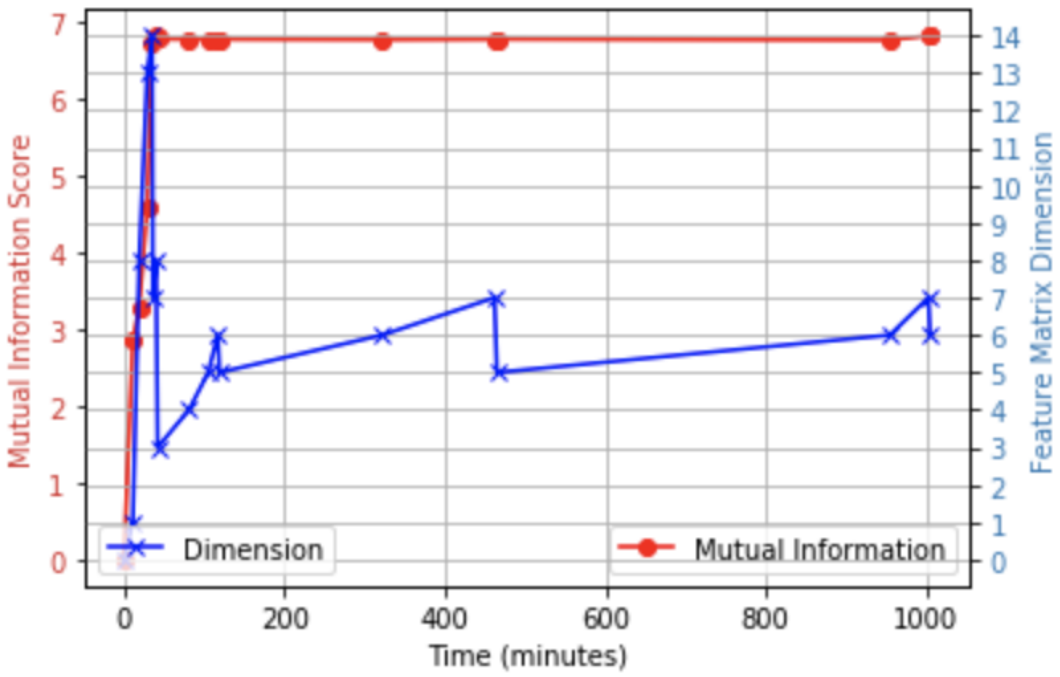

Figure 4-1: A graph of the simulation project's dimensionality and information with the target column, as features are accepted. Time is measured since start of project simulation in minutes. Points only appear after a feature is accepted or a feature is pruned. Pruning generally occurs several (2-3) minutes after a feature is accepted.

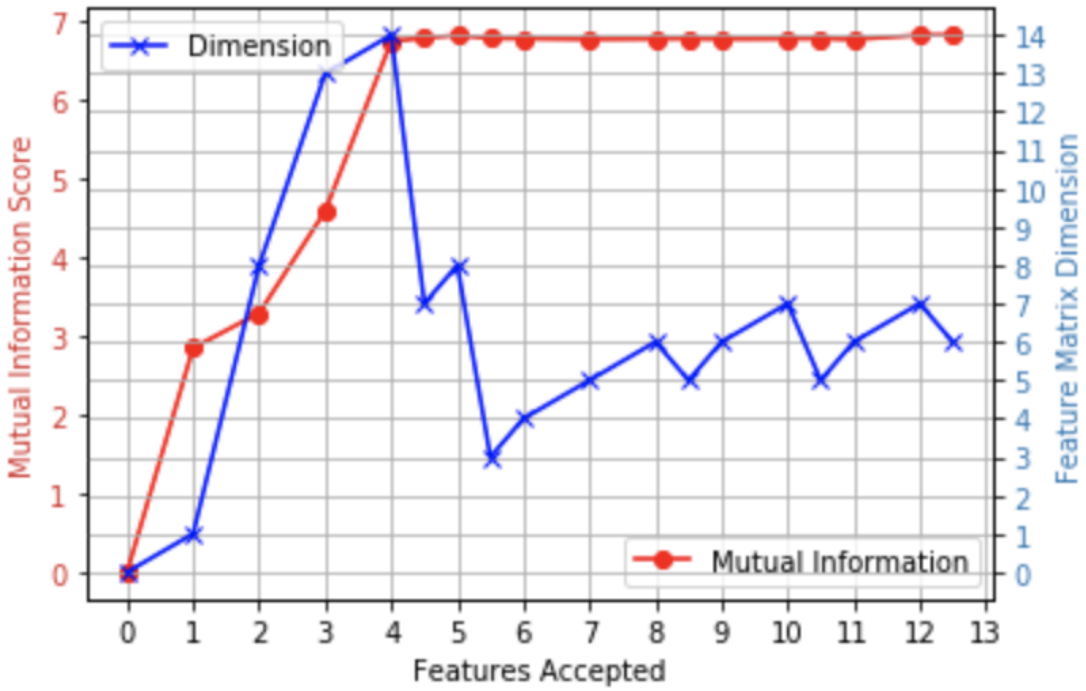

Figure 4-2: The same graph as Figure 4-1, but spaced out to better visualize each feature's contribution Ticks represent the state of the project immediately after a feature is accepted, and points between ticks represent the state of the project if the pruning routine detects and removes redundant features.

Mutual Information between X and Y is calculated as $I(X;Y) = H(X) + H(Y) - H(X;Y)$. As the entropy $H(X,Y)$ is lower bounded by $\max(H(X), H(Y))$, the mutual information between two data-sets is upper bounded by their entropies.

For `Ballet` prediction problems, that means that the mutual information between the feature matrix and its target column y is upper bounded by the entropy of the target y. For the simulation Ames data-set, we find this number to be close to 6.3. We see this reflected in Figure 4-2: After the first four features are accepted, the mutual information between the feature matrix and target column stays relatively the same for the rest of the simulation.

After that point, features that are accepted generally force another to be pruned. We see that these features do not change the mutual information score or dimensionality of the feature matrix appreciably, so this can be thought as a "swapping" of similar features.

We also see that features proposed earlier were more readily accepted than features proposed later in Figure 4-1. This is most likely due to a combination of the mutual information limit and the level of redundancy in the features — by the middle of the project, not many features can contribute new information, and so the project is much less active.

4.3.2 Effects of Varying Hyper-Parameters

To test the effects of varying hyper-parameters, we ran the simulation for different values of λ_1 and λ_2, keeping the features and order of submission constant. We expect more features to be accepted at lower values of these parameters and fewer features to be accepted at higher values.

When decreasing the values of each hyper-parameter by a factor of 2, we found that the total number of accepted features increased, but so did the number of pruned features. By the end of the simulation, the number of accepted features still in the project did not increase appreciably. Similar to the original simulation, the simulation with lower hyper-parameters quickly accepted a handful of features without pruning any features. After that, the simulation exhibited the same "feature-swapping".

We decide to keep the normalization constants for the hyper-parameters to be 32, as it allowed for relevant features to be accepted and had relatively little "feature-swapping" to occur. As mentioned earlier, this is *not* a desirable effect — every change should improve the project's relevance, so these swaps are unnecessary and confusing for maintainers and contributors.

4.3.3 Pruning Graphs

As mentioned earlier, the goal of pruning is to remove redundant features; usually, this is because a new feature shares the same information but is more relevant or compact.

In figure 4-3, we see a graph of which features (represented as nodes) prune which other features. The size of the node corresponds to the information shared with

the target column and therefore also is correlated with the feature's relevance. As expected, usually a feature that gets pruned has less information than the feature that prunes it.

However, there is one large counterexample in the graph where a much less relevant feature prunes a much more relevant feature. Looking into the Travis CI logs to see the results of the pruning check, it seems that the more relevant feature, while having more mutual information overall, had very little information conditioned on the other features. Essentially, the feature was already partially redundant and the acceptance of the less relevant feature made it fully redundant.

We also noticed deep nesting of pruning in the features; often, it may be the case that the feature that pruned a feature f_i may itself be pruned as well. This can be understood as users constructing similar features; because Kaggle members generally do not collaborate during feature engineering, there is a high level of redundancy in the features we collected. Features that perform similar transforms and are built on the same data columns in the raw data-set are often similar and tend to prune each other if the new feature is found to be better.

4.3.4 Understanding Pruning

As the data columns in the Ames data-set logically represent different qualities of each Iowan house, we should be able to find relationships between the inputs of features that prune and were pruned.

In one case, we found this relationship: in our simulation, two features proposed methods related to the `Garage Area` column and the acceptance of one of these features caused the pruning of the other. This is to be expected, as two features that share the same input are likely to be redundant in the information they provide.

In most cases, it is difficult to see why one feature might prune another - even if two features are built off different inputs and do different tasks, they still might provide redundant information for the project. Similarly, many features may be rejected from the project because of their redundancy with an already accepted feature that seems different. This may be frustrating for users who do not understand why their

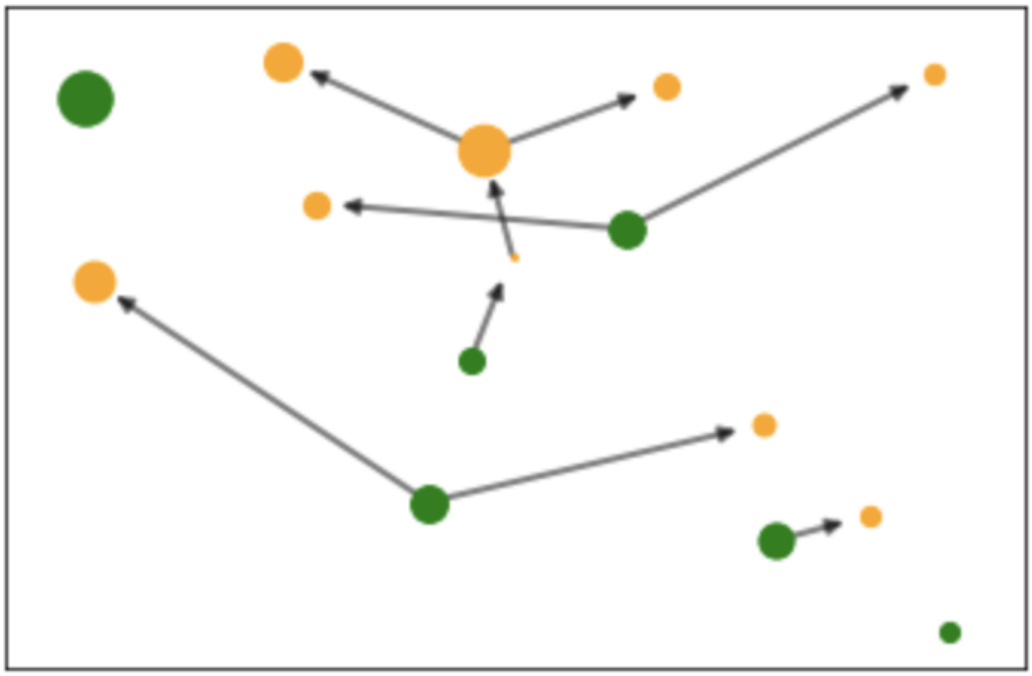

Figure 4-3: A directed graph representing features pruning one another. Green nodes represent features that have been accepted and orange nodes represent features that were accepted but then pruned later on. The size of the node is proportional to the mutual information score of that feature with respect to the target. Edges pointing from $f_i \rightarrow f_j$ represent that the acceptance of f_i caused f_j to be pruned. Feature nodes at the top of the graph were introduced to the project earlier than features lower on the graph, hence why directed edges only point upwards.

features are being removed. For future simulations, better logging and reporting of information statistics should be implemented to make the feature selection process easier and clearer to understand.

4.3.5 Clustering of Features

In Figure 4-4, we take a subset of the first 45 proposed features and graph their relevances with respect to each other. To build this, for this feature subset we calculated each feature's mutual information score with every other feature and created a "relevance edge" between high information sharing features.

Of the first 45 features, we see that features tend to cluster amongst each other and closely share information. We see in Figure 4-4 that a majority of the features are part of one large cluster that consists of a singular larger feature at the center

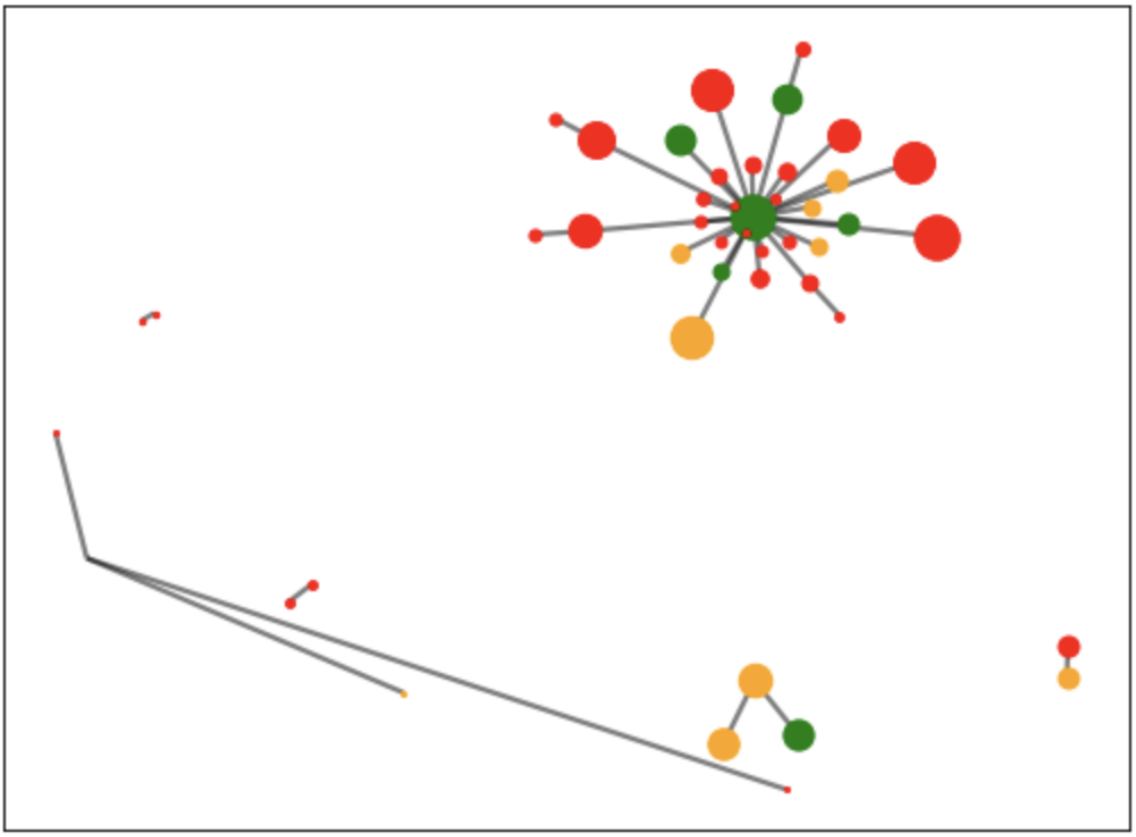

Figure 4-4: An undirected graph representing feature relevance between one another. Green nodes represent features that have been accepted, orange nodes represent features that were accepted but then pruned later on, and red nodes represent features that were rejected in the feature acceptance stage. The size of the node is proportional to the mutual information score of that feature with respect to the target. Edges represent features that share information with each other, with shorter edges representing more information shared. As every feature has some information with each other, only a subset of edges have been shown here.

of a star. This implies that many of the features submitted by Kaggle users shared much of the same information and were redundant with respect to each other.

4.4 Concluding Remarks

As previously discussed, we expected a high level of redundancy and irrelevancy within the features constructed from the Kaggle kernels, as these users are not working collaboratively. Even for this hypothesis, the number of features selected by GFSSF is surprisingly low; a similar simulation done using α-investing accepted an order of magnitude more features [16].

While we can visualize the redundancies between features in the simulation, understanding the interactions in the GFSSF algorithm are tougher because of the lack of interpretability in the algorithm. As such, it is a main motivation for future work to be able to understand the `Ballet` validators at each step of the Feature Engineering Life-Cycle.

Chapter 5

Conclusion

5.1 Future Work

Though the feature selection pipeline for `Ballet` is in good shape, there are still more tasks at hand to ensure that `Ballet` is useful for open source data science projects. The selection helps address many data science oriented goals for `Ballet`, but the framework must also consider the open source community.

5.1.1 Better Acceptance/Pruning Reporting

Considering the current `Ballet` framework, it is difficult to figure out why a feature is accepted, rejected, or pruned. Currently, the only source of information about a feature's acceptance exists in Travis CI, though many lines of logging code. Not only is this cumbersome to find, but figuring out what the logs mean is near impossible without understanding the GFSSF algorithm.

For `Ballet` to be successful, it should be easier for maintainers and collaborators to find, interpret, and understand the different stages of the feature selection process. This could mean using mutual information scores to let users know if their feature is similar to other features, or using the GitHub app to comment directly on pull requests different statistics of their feature.

5.1.2 Configuration / Detection of Continuous and Discrete Columns

One major issue with the current validators is the use of heuristics to determine whether a certain dataset should be treated as discrete or continuous. Currently, `Ballet` uses a set of simple heuristics to determine if a column is discrete or not; if the column is integer-valued or has few unique values, `Ballet` marks the column as discrete and marks it continuous otherwise.

However, the estimations can vary wildly depending on which estimator is used; for example, if $X \sim Bernoulli(0.5), X \in \{0, 1\}$ is the 0-1 result of a coin flip and $Y = 10 \times X, Y \in \{0, 10\}$, a discrete estimator would treat their samples to have the same entropy and a continuous estimator would treat Y to have a higher entropy as the range of the values is higher. More complex issues arise when attempting to calculate information scores with the wrong type of estimators applied for entropy.

This is potentially a problem in the simulation; in the documentation for the Ames Dataset, the target column (Sales Price) is listed as a continuous data column but is treated as a discrete value in `Ballet` as it contained integer values. While user configuration is best for any project, this can be cumbersome as datasets may contain many thousands of columns and having users mark logical feature columns as continuous or discrete may not be effective. However, finding a better solution for determining data column type is necessary for `Ballet` to improve.

5.1.3 Feature Discovery and Contribution

While the project structure of `Ballet` is useful for partitioning work amongst contributors, it is difficult for users to easily discover the features that other contributors have made. In a project of hundreds of features, this increases the risk of redundant proposed features and reduced efficiency for the project. As such, `Ballet` would eventually need a method by which users can quickly and easily browse the features of a project. This solution could take the form of test in a GitHub repository's `readme.md` file, a lightweight website, or hosting through GitHub pages.

Similar to the first goal, it is important for contributors to know the weight of their contributions. The benefit is twofold; firstly, giving users a contribution "score" both increases their satisfaction and reward for contributing. It also allows other contributors to identify "good" logical features, potentially learning behaviors in the dataset and good practices for data science. Like feature discovery, showing a contributor's score can be achieved through several lightweight solutions.

5.1.4 Implementation of other Relevance Algorithms

While GFSSF is a useful algorithm for some predictive models, it is possible that other algorithms for feature selection may be more useful for different use cases. For example, the features chosen by GFSSF for the Ames simulation have a high mutual information score, but a linear regression model fitted on the features preforms poorly ($r^2 \sim 59\%$). In the future, it may be useful to implement different feature selection algorithms based on the prediction problem and the optimization/loss function being used.

Bibliography

[1] Airbnb. New user bookings dataset, 2016. data posted publicly to Kaggle, `https://www.kaggle.com/c/airbnb-recruiting-new-user-bookings/`.

[2] Adam Baldwin. Details about the event-stream incident — the npm blog. `https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident`, 2018. Accessed 2018-11-30.

[3] Justin Cheng and Michael S. Bernstein. Flock: Hybrid Crowd-Machine Learning Classifiers. *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing - CSCW '15*, pages 600–611, 2015.

[4] Dean De Cock. Ames, iowa: Alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, 19(3), 2011.

[5] Nicolas Fournier and Sylvain Delattre. On the kozachenko-leonenko entropy estimator. *arXiv preprint arXiv:1602.07440*, 2016.

[6] Utsav Garg, Viraj Prabhu, Deshraj Yadav, Ram Ramrakhya, Harsh Agrawal, and Dhruv Batra. Fabrik: An online collaborative neural network editor. *CoRR*, abs/1810.11649, 2018.

[7] Google colaboratory.

[8] Kaggle.

[9] Alexander Kindel, Vineet Bansal, Kristin Catena, Thomas Hartshorne, Kate Jaeger, Dawn Koffman, Sara McLanahan, Maya Phillips, Shiva Rouhani, Ryan Vinh, and et al. Improving metadata infrastructure for complex surveys: Insights from the fragile families challenge, Sep 2018.

[10] 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.

[11] LF Kozachenko and Nikolai N Leonenko. Sample estimate of the entropy of a random vector. *Problemy Peredachi Informatsii*, 23(2):9–16, 1987.

[12] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 69 6 Pt 2:066138, 2004.

[13] Haiguang Li, Xindong Wu, Zhao Li, and Wei Ding. Group feature selection with streaming features. In *2013 IEEE 13th International Conference on Data Mining*, pages 1109–1114. IEEE, 2013.

[14] Christian Payne. On the security of open source software. *Information systems journal*, 12(1):61–78, 2002.

[15] Isaac Z. Schlueter. kik, left-pad, and npm — the npm blog. `https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm`, 2018. Accessed 2018-11-30.

[16] Micah J. Smith, Kelvin Lu, and Kalyan Veeramachaneni. Ballet: A lightweight framework for open-source, collaborative feature engineering. In *Workshop on Systems for ML at NeuRIPS 2018*, 2018.

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

[18] Steve Webb, James Caverlee, and Calton Pu. Introducing the webb spam corpus: Using email spam to identify web spam automatically.

[19] Xindong Wu, Kui Yu, Hao Wang, and Wei Ding. Online streaming feature selection. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 1159–1166. Citeseer, 2010.

[20] Katharine Xiao. Towards automatic data element linking. Master's thesis, Massachusetts Institute of Technology, 7 2017.

[21] Kui Yu, Xindong Wu, Wei Ding, and Jian Pei. Towards scalable and accurate online feature selection for big data. In *2014 IEEE International Conference on Data Mining*, pages 660–669. IEEE, 2014.

[22] Jing Zhou, Dean Foster, Robert Stine, and Lyle Ungar. Streaming feature selection using alpha-investing. *Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining - KDD '05*, page 384, 2005.