Ballet: A lightweight framework for open-source, collaborative feature engineering

Micah J. Smith
MIT
micahs@mit.edu

Kelvin Lu
MIT
kelvinlu@mit.edu

Kalyan Veeramachaneni
MIT
kalyanv@mit.edu

Abstract

Collaborative development of open data science projects is unsupported by existing software development paradigms. We introduce Ballet, a software framework for developing new supervised learning projects with many synchronous collaborators. Feature engineering source code is submitted incrementally and validated using extensive software tests and a streaming logical feature selection algorithm. Notably, Ballet is built on the same lightweight community infrastructure as existing open-source software libraries. In a case study of predicting house prices, we show that feature source code extracted from public notebooks can be integrated using our framework to incrementally build a feature matrix without sacrificing modeling performance.

1 Introduction

Open data science is a paradigm in which data science resources are deployed to address societal problems in education, government, health, and more through community-driven, transparent analysis of public datasets. As a recent example, the Fragile Families Challenge [8] tasked researchers and data scientists with predicting GPA and eviction for a set of disadvantaged children. The output of such a project is not a general-purpose software library but rather a *predictive model*.

These projects may attract hundreds of interested data scientists and researchers, each with unique skills and intuition. For prospective contributors to collaborate effectively at scale, there must exist a way to *split* data science tasks, like feature engineering, and then *combine* individual units of data science source code, like feature functions. Unfortunately, no such framework exists.

In this paper, we propose a lightweight, collaborative framework for developing predictive models for open data science applications through a focus on feature engineering. Under this framework, an open-source software repository contains a curated, executable feature engineering pipeline for transforming a raw dataset into a feature matrix. Collaborators incrementally propose new features for inclusion in this pipeline as modular source code contributions. For each proposed feature, an automatic process rigorously validates it for software correctness and performs streaming logical feature selection (SLFS) to judge whether it can be merged. The resulting feature matrix can be used as the input to an automated machine learning model or a custom algorithm. Projects built on our "lightweight" framework do not require any computing infrastructure beyond that which is commonly used in open-source software, like free code hosting and CI services. We implement these ideas in the open-source Ballet framework[1] and demonstrate its use in a house price prediction problem.

[1] https://github.com/HDI-Project/ballet

2 Logical feature selection

The fundamental data science unit in this setting is a logical feature. A *logical feature* is a function that maps raw variables in one data instance to a vector of feature values, $f_j^{\mathcal{D}} : \mathcal{V}^p \to \mathbb{R}^{q_j}$, where $\mathcal{V}$ is the set of feasible raw data values, p is the dimensionality of the raw data, and q_j is the dimensionality of the jth feature vector. Each logical feature is parameterized by $\mathcal{D} \in \mathcal{V}^p \times \mathbb{R}$, a training dataset of n instances, for learned parameters such as variable statistics. Importantly, a logical feature can produce either a scalar feature value for each instance or a vector of feature values, as in the case of an embedding technique like PCA or the one-hot encoding of a categorical variable.

Given a training dataset $\mathcal{D}$, a collection of feature functions $\mathcal{F}^{\mathcal{D}} = \{f_j | j = 1 \dots m\}$, and a collection of new instances $\mathcal{D}\prime$, we extract a feature matrix

$$X^{\mathcal{D}\prime} = \mathcal{F}^{\mathcal{D}}(\mathcal{D}\prime) = (f_1^{\mathcal{D}}(\mathcal{D}\prime), \dots, f_m^{\mathcal{D}}(\mathcal{D}\prime)). \tag{1}$$

The logical feature selection problem is to select a subset of feature functions, $\mathcal{F}^* \subseteq \mathcal{F}$, that minimizes the expected loss of some learner $\mathcal{A}$ trained on the extracted values,

$$\mathcal{F}^* = \underset{\mathcal{F}' \in \mathcal{P}(\mathcal{F})}{\arg\min}\, \mathbb{E}[\mathcal{L}(\mathcal{A}_{\mathcal{F}'})] \tag{2}$$

In contrast, the traditional feature selection problem is to select a subset of the feature values themselves, $X^* \subseteq X$; this may not be directly possible or desirable in order to preserve the coherence and interpretability of logical features.

2.1 Streaming logical feature selection

In Ballet, collaborators incrementally propose logical features, which are accepted or rejected in a streaming fashion, a related problem to streaming feature selection (SFS) and streaming group feature selection (Section 5). The SLFS problem consists of two sub-problems.

Definition 1 *Let $\mathcal{F}_t$ be the set of features accepted as of time t, and let f_{t+1} be proposed at time $t+1$. The* streaming feature acceptance *decision problem is to* accept *f_{t+1}, setting $\mathcal{F}_{t+1} = \mathcal{F}_t \cup f_{t+1}$, or* reject*, setting $\mathcal{F}_{t+1} = \mathcal{F}_t$.*

Definition 2 *The* streaming feature pruning *decision problem is to remove a subset $S \subseteq \mathcal{F}_{t+1}$ of low-quality features, setting $\mathcal{F}_{t+1} = \mathcal{F}_{t+1} \setminus S$.*

Existing algorithms fit into this general formulation. For example, α-investing [25] lies within the streaming feature acceptance stage, while OSFS [23] is implemented as an "online feature relevance" test in the acceptance stage and an "online feature redundancy" test in the pruning stage.

3 The Ballet framework

There are two main design considerations — and associated challenges — for developing open data science projects. First, just as each release of a software library provides patches of additional functionality that can be applied on top of existing versions, successful open-source data science projects must be easily *patchable* such that they can be improved in an incremental manner. To address this, Ballet projects maintain a *feature engineering pipeline invariant* — this invariant provides that the pipeline always be able to perform efficient, high-quality, end-to-end feature engineering on new data instances. Maintaining this invariant in the presence of new proposed features of uncertain quality represents a challenge. We impose a careful structure on new feature engineering source code submissions and rigorously validate them along several dimensions. The outcome is that high-quality features can be confidently merged to the library, while error-prone source code or unhelpful features can be automatically rejected.

Second, we cannot rely upon custom computing infrastructure to facilitate this development experience. 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

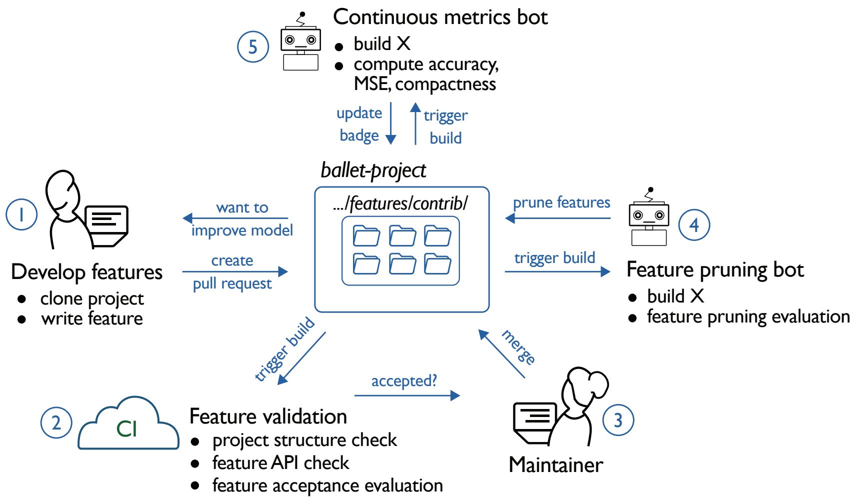

Figure 1: The lifecycle of a new feature, from proposal to impact on the model. A user observes metrics of the predictive model and desires to improve it further. They clone the project, write a new feature, and propose it for inclusion in the pipeline using one of several clients (1). Next, a continuous integration process validates the feature on several levels and labels it as accepted or rejected (2). After an accepted feature is merged (3), it triggers a feature pruning bot which may propose the removal of redundant features (4). Finally, a continuous metrics bot rebuilds the pipeline, evaluates a predictive model, and updates metrics like accuracy and feature compactness (5). The cycle continues until performance improvements are exhausted.

code hosting, and free continuous integration (CI) testing. To achieve this, through careful design decisions we maximize use of "community infrastructure" like GitHub[2] and Travis CI.[3]

Finally, data science platforms often place considerable restrictions on the development style and experience of data scientists. Besides the minimal structure we impose for integrating feature contributions into feature engineering pipelines and validating new submissions, we facilitate completely unopinionated development and modeling. Maintainers or designated collaborators can push arbitrary code to a Ballet project, or it can be used as a versioned dependency of another project.

3.1 Creation and development of a predictive modeling project

In this section, we describe how a predictive modeling project is created, developed, and validated using the Ballet framework.

Project instantiation Ballet renders a new repository from an included template. The repository contains the minimal files and structure required for feature engineering and validation of proposed features: a Ballet configuration file, metadata on the prediction problem, stubs for loading data and building features, and more. All community-developed feature engineering source code lives in a dedicated subdirectory, `contrib`. The repository is registered with GitHub and a CI provider.

The `Feature` abstraction The Ballet `Feature` is a simple but powerful abstraction that represents a single logical feature (Section 2). Mainly, it wraps a series of transformers, each exposing a "fit/transform" interface [3], in a robust transformer pipeline. The resulting pipeline is suitable for implementing complicated feature engineering functions in a composable, leakage-free manner, customized for operating on labeled, tabular datasets. It also stores detailed metadata about the logical feature for recording data provenance and annotating outputs.

Feature engineering pipeline The resulting repository contains a usable (if, at first, empty) feature engineering pipeline that can be executed to transform a raw dataset into a well-formed feature matrix. To execute the pipeline, a Python entry-point is included that walks the project's `contrib` subdirectory, imports `Feature` objects from Python modules, combines them into a pipeline, and

[2]`https://www.github.com`
[3]`https://www.travis-ci.org`

transforms a provided raw dataset into a well-formed feature matrix. Importantly, the pipeline is represented as source code rather than any sort of serialized object.

Collaborative development of new features 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 (Figure 1). To contribute a new feature, data scientists 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 subdirectories, 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.

To ease development, we also provide `ballet.eng`, a library of powerful feature engineering-specific transformers for tasks like cleaning missing data, encoding non-numeric values, and handling time series data. This augments the transformers within `scikit-learn`, which are generally targeted at the machine learning phase.

3.2 Feature validation

Potential contributors who have developed new features now hope to have them included in the project. Once features are proposed, we must ensure that only high-quality submissions are merged into the repository. The risks here are twofold. First, if code is introduced to the repository that contains any errors, whether implementation errors or fragile behavior on unseen inputs, then the feature pipeline will become useless until it is manually repaired by a project maintainer — a blocking, time-consuming process. Second, if features are introduced that are irrelevant, redundant, or have low predictive power, then the performance of the predictive model can be negatively impacted. Submissions must therefore be carefully validated on several counts.

Project structure check We check the project structure given the file diffs introduced in the pull request. The only acceptable changes are file additions of Python source files underneath the project's `contrib` subdirectory that follow a specified naming convention. The introduced module must then define exactly one object that is an instance of `Feature`, which will be imported. Pull requests that fail this check are rejected.

Feature API checks We fit the feature to the training dataset and extract features values from the training and test datasets. We conduct a battery of tests to increase confidence that the feature would produce acceptable feature values on unseen inputs: no missing, infinite, or non-numeric values.

Streaming feature acceptance Regardless of the feature's implementation in software, it must be judged by a streaming logical feature acceptance algorithm to ensure it would not impact the usefulness of the feature matrix produced by the project. In Ballet, we provide a reference implementation of a streaming logical feature acceptance algorithm (Section 2.1), a simple adaptation of the α-investing algorithm [25]. Other algorithms can be configured for specific predictive models.

Let $\mathcal{F}_t$ be the features accepted so far and let $\mathcal{F}_t^\dagger = \mathcal{F}_t \cup f_{t+1}$ include the newly proposed feature. Compute $T = -2(logL(\mathcal{F}_t) - logL(\mathcal{F}_t^\dagger))$, where $L(\cdot)$ is the maximum likelihood of an OLS estimator. Since $T \sim \chi^2(q)$, we compute a p-value and accept if $p < \alpha_t$, setting $\mathcal{F}_{t+1} = \mathcal{F}_t^\dagger$, where α_t is the α-investing step-dependent acceptance threshold [25].

Streaming feature pruning In the previous step, we can only decide to accept or reject a proposed feature. But if a new feature is merged, then it may present an opportunity to prune existing features which may have been newly made redundant. For example, in the OSFS algorithm [23], a feature column $x \in X_t$ is removed if there exists a subset $S \subseteq X_t \setminus x$ such that $y \perp\!\!\!\perp x \mid S$. The reference implementation in Ballet does not prune any features.

3.3 Other Considerations

Other lightweight services can expand on the core Ballet functionality described in this section. For example, today, developers of open-source software projects rely on a variety of external integrations

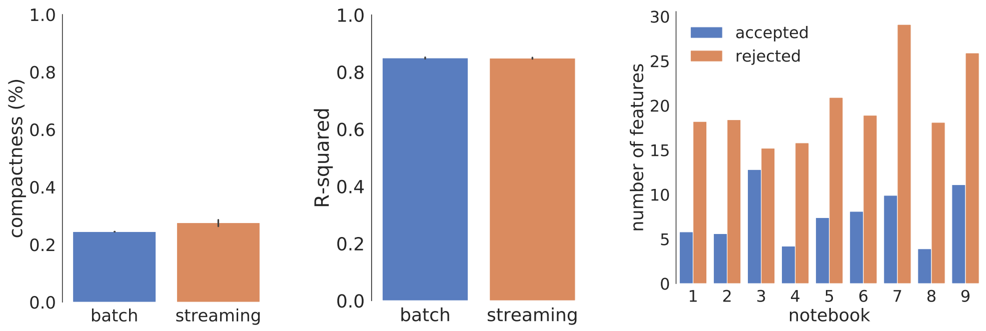

Figure 2: Performance of streaming and batch feature selection on Ames notebook features in terms of compactness (percentage of logical features selected by algorithm, left) and R^2 (center); mean accepted and rejected features per data scientist using SLFS (right).

for functionality like testing, documentation builds, code coverage, and binary distribution. These integrations serve to increase end-user trust that the software they are considering using will work for their particular problem. Similarly, open-source predictive models need to assure users that predictions will be generalize accurately and fairly to new data points. We introduce the notion of *continuous metrics*, services that check out the feature engineering pipeline on each commit, train and tune a predictive model, and report metrics on a withheld validation set. The metrics can then be displayed on the projects documentation files using a live "metrics badge" icon.

Any software project that receives untrusted code from contributes must be mindful of security considerations. The primary threat model for Ballet projects are well-meaning data scientists and other contributors that submit poor features or inadvertently "break" the feature engineering pipeline. We address the former through feature validation (Section 3.2) and the latter through lightweight project status checks to reject patches that improperly modify core project structure, such as the `validate.py` driver script or the `ballet.yml` configuration file. However, Ballet does conceive of facilitating automatic merges of accepted submissions, presenting a risk if harmful code is embedded feature functions or adversarial features are created by attackers, especially given the public availability of validation data. A practical defense is to require a human maintainer's approval in the final acceptance of proposed contributions, but defending against malicious patches is an ongoing struggle for open-source developers [16, 6, 19, 1].

4 Evaluation

There are few existing benchmarks against which to compare Ballet. Here, we focus on assessing the usability of Ballet for synchronous feature engineering collaborations as well as the efficacy of SLFS compared to traditional batch methods.

User study: prototype framework We evaluated a prototype of Ballet in a user study with 8 researchers and data scientists. We explained the framework and gave a brief tutorial on how to write features. Participants were then tasked with writing features to help predict the incidence of dengue fever given historical data from Iquitos, Peru and San Juan, Puerto Rico [7]. Three participants were successfully able to merge their first feature within thirty minutes, while the remainder produced features with errors or were unable to write a new feature. In interviews, participants suggested that they found the Ballet framework helpful for structuring submissions and validating features, but were unfamiliar with the transformer style of writing feature engineering code. Based on this feedback, we created the `ballet.eng` transformer library and provided feature engineering tutorial materias for new contributors.

Case study: Ames housing price prediction As a case study of the Ballet framework, we simulate feature engineering collaboration on the Ames housing price dataset [5]. The challenge is to predict home prices in Ames, Iowa given a large number of dirty, "real-world" variables.

Kaggle [12], a data science competition community, uses the Ames problem as a tutorial. We identify 9 public notebooks by Kaggle members in which they engineer features as part of their end-to-end pipeline. We manually implement each feature as a `Feature` object, resulting in 249 logical features.

We repeatedly simulate a scenario in which each Kaggle member separately submits their features to a Ballet project. In each scenario, we iteratively select the first remaining feature from a randomly-chosen notebook and simulate its submission and validation using SLFS (Section 3.2). For comparison, we start with the entire set of logical features and then use a batch-mode feature selection technique that is comparable in spirit, greedy forward selection with AIC scoring. For each technique, we use the resulting feature matrix to make and score predictions using a baseline model.

We find that the SLFS method and traditional batch methods produce similarly compact feature sets, with 27.6% and 24.5% of logical features accepted, respectively (Figure 2). Even though SLFS considers each feature exactly once, it only accepts slightly more features than a greedy batch algorithm, suggesting that many features may be considered redundant as soon as they are submitted. Indeed, every notebook contained rejected features, suggesting that every data science duplicated some effort of others. One concern about streaming feature selection is that a feature would be incorrectly rejected early on when, combined with another feature that has yet to arrive, it would turn out to be useful [11]. In the logical feature selection setting, this worry may be diminished because the specific feature columns that are correlated arrive as a single logical feature. The modeling performance is also similar for the streaming and the traditional settings, suggesting that modeling performance is not negatively impacted by Ballet-style development.

5 Related Work

Streaming feature selection. Feature selection is an important topic in the machine learning and data mining literature and has been studied extensively [11]. Interest in streaming feature selection has accelerated as big data settings have increased the dimensionality of the variable space. The grafting approach [17] uses stagewise gradient descent to alternately optimize free parameters and select new features for a regularized maximum likelihood. α-investing [25] is an adaptive complexity penalty model that bounds the false discovery rate of selecting poor features. [23] use separate statistical tests to first add relevant features to a candidate set and subsequently remove redundant ones. Similar approaches have been extended to group-wise feature value selection [22, 24].

Collaborative data science and machine learning. 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. One exception is [20], who propose a collaborative feature engineering system in which contributors submit source code directly to a machine learning backend server. While Ballet shares several ideas, it uses a lightweight approach to integrate features — suitable for open data science projects — and improves on the feature selection techniques. In other 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 [4], or real-time editing interfaces, like that of [9, 10, 15], facilitate multiple users to edit a machine learning model specification at the same time. Collaboration can also be achieved *implicitly* in data science and machine learning competitions [2, 12, 14] and using networked science hubs [21]. 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, large commercial data science platforms, such as Domino Data Lab[4] and Dataiku[5], while aiming to support all aspects of data science product infrastructure and deployment, do not explicitly provide for collaborative, incremental model development.

6 Conclusion

In this paper, we introduced Ballet, a new framework for developing predictive models in an open-source, collaborative way through feature engineering. We described the design and realization of this framework and analyzed a case study involving a simulated collaboration scenario. In further research, developer efficiency and ease of use can be evaluated in more detail, and new algorithms can be developed to target the logical feature selection problem, perhaps targeting multiple modalities of information about features. Finally, we hope to expand the framework to other similar problems like data programming [18], prediction engineering [13], or survey dataset preparation and analysis.

[4]`https://www.dominodatalab.com/`

[5]`https://www.dataiku.com/`

References

[1] 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.

[2] James Bennett, Stan Lanning Netflix, and Netflix. The netflix prize. 2007.

[3] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[4] 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.

[5] 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.

[6] Alexandre Decan, Tom Mens, and Maelick Claes. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, ECSAW '16, pages 21:1–21:4, New York, NY, USA, 2016. ACM.

[7] Epidemic Prediction Initiative. Dengue forecasting project. URL `http://predict.phiresearchlab.org/post/5a4fcc3e2c1b1669c22aa261`. Accessed 2018-04-30.

[8] Fragile Families Challenge. Fragile families challenge. URL `http://www.fragilefamilieschallenge.org/`.

[9] 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.

[10] Google Colaboratory. Google colaboratory. URL `https://www.colab.research.google.com`.

[11] Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *Journal of Machine Learning Research (JMLR)*, 3(3):1157–1182, 2003.

[12] Kaggle. Kaggle. URL `https://www.kaggle.com/`.

[13] James Max Kanter, Owen Gillespie, and Kalyan Veeramachaneni. Label, segment, featurize: A cross domain framework for prediction engineering. *Proceedings - 3rd IEEE International Conference on Data Science and Advanced Analytics, DSAA 2016*, pages 430–439, 2016.

[14] KDD Cup. Kdd cup. URL `http://kdd.org/kdd-cup`.

[15] 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.

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

[17] Simon Perkins and James Theiler. Online Feature Selection using Grafting. *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*, 2003.

[18] Alexander Ratner, Christopher De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. Data programming: Creating large training sets, quickly. *Advances in neural information processing systems*, 29:3567–3575, 2016.

[19] 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.

[20] 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.

[21] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luís Torgo. Openml: networked science in machine learning. *SIGKDD Explorations*, 15:49–60, 2013.

[22] Jing Wang, Meng Wang, Peipei Li, Luoqi Liu, Zhongqiu Zhao, Xuegang Hu, and Xindong Wu. Online Feature Selection with Group Structure Analysis. *IEEE Transactions on Knowledge and Data Engineering*, 27(11):3029–3041, 2015.

[23] Xindong Wu, Kui Yu, Wei Ding, Hao Wang, and Xingquan Zhu. Online feature selection with streaming features. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(5): 1178–1192, 2013.

[24] Kui Yu, Xindong Wu, Wei Ding, and Jian Pei. Scalable and accurate online feature selection for big data. *TKDD*, 11:16:1–16:39, 2016.

[25] 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.