# On the Co-evolution of ML Pipelines and Source Code - Empirical Study of DVC Projects

Amine Barrak
*Polytechnique Montreal*
Montreal, Canada
amine.barrak@polymtl.ca

Ellis E. Eghan
*Polytechnique Montreal*
Montreal, Canada
ellis.eghan@polymtl.ca

Bram Adams
*Queen's University*
Kingston, Canada
bram.adams@queensu.ca

***Abstract*—The growing popularity of machine learning (ML) applications has led to the introduction of software engineering tools such as Data Versioning Control (DVC), MLFlow and Pachyderm that enable versioning ML data, models, pipelines and model evaluation metrics. Since these versioned ML artifacts need to be synchronized not only with each other, but also with the source and test code of the software applications into which the models are integrated, prior findings on co-evolution and coupling between software artifacts might need to be revisited. Hence, in order to understand the degree of coupling between ML-related and other software artifacts, as well as the adoption of ML versioning features, this paper empirically studies the usage of DVC in 391 Github projects, 25 of which in detail. Our results show that more than half of the DVC files in a project are changed at least once every one-tenth of the project's lifetime. Furthermore, we observe a tight coupling between DVC files and other artifacts, with 1/4 pull requests changing source code and 1/2 pull requests changing tests requiring a change to DVC files. As additional evidence of the observed complexity associated with adopting ML-related software engineering tools like DVC, an average of 78% of the studied projects showed a non-constant trend in pipeline complexity.**

***Index Terms*—DVC, ML Pipeline, Co-evolution, ML versioning**

## I. Introduction

Modern machine learning (ML) applications require elaborate pipelines for data engineering, model building, and releasing [1], [2]. Data engineers use a pipeline of tools to automate the collection, preprocessing, cleaning and labeling of data. Data scientists, on the other hand, also use a pipeline to extract the right features from the data engineers' data, execute machine learning scripts while experimenting with different sets of values for hyperparameters, validate the resulting models, and then deploy (serve) the selected models. Since these steps have to be repeated over and over whenever the data and/or model scripts or parameters change, in search of ever more accurate models, automation of these pipelines is essential.

These kinds of pipeline, however, are difficult to manage using "traditional" software engineering tools like source code version control systems (e.g., git) or continuous integration (CI) servers. For example, due to the highly experimental nature of data engineering/science, it is important to be able to swiftly revert the model to a prior high-performing model whenever any issues are encountered. Moreover, since ML models and workflows are centered around data (training and testing), it is important to keep track of the data used at each step and iteration of the workflow, i.e., recording the history of the different ML stages, to ensure the reproducibility of the ML pipeline (same inputs, same outputs) and to track data provenance. Furthermore, the huge amount of data used by ML applications makes it infeasible for traditional version control systems to keep tracking them, and more sophisticated tools are required to store and retrieve them efficiently [2].

As such, a new breed of data and model versioning tools have appeared to support data engineers and scientists [3]. Popular tools comprise DVC [4], MLFlow [5], Pachyderm [6], ModelDB [7] and Quilt Data [8]. They typically combine the ability to specify data and/or model pipelines, with advanced versioning support for data/models, and the ability to define and manage model experiments. In the same vein as infrastructure-as-code (IaC) [9], many of these functionalities are based on textual specifications instead of relying on manual intervention.

Given this new generation of tools, this paper aims to empirically study the prevalence of ML pipelines in open source projects, as well as the amount of maintenance effort involved. Previous studies on non-ML projects have shown that frequent changes to source code might require corresponding changes to other software artifacts such as build files [10] and infrastructure-as-code files (IaC) [11] (or vice versa), causing overhead to developers. In the case of ML projects, changes to data and/or model pipelines might induce similar overhead due to the conceptual coupling between data, model and release pipelines.

Hence, through a high-level empirical study of 391 GitHub projects using the popular DVC versioning tool, followed by a more detailed analysis on the 25 most active projects, we address the following research questions:

**(RQ1) How common is the usage of DVC in Github projects?** Our empirical study on 391 Github projects shows that ML versioning is a young practice in open source repositories. However, we observe early adoption and high usage of DVC functionalities in more than a quarter of the studied projects; data versioning is the first

DVC feature adopted in 321/391 of the studied projects. Several projects are still exploring and experimenting with this new practice.

**(RQ2) How much coupling exists between software artifacts and DVC artifacts?** At the pull request-level, results show a high coupling (a median confidence value as high as 91.91%) between DVC files and source code files. The coupling between DVC files and test files, however, generally is low.

**(RQ3) How does the complexity of the DVC ML pipeline evolve over time?** The results show that 78% of the studied projects have a non-constant complexity trend, i.e., complexity may increase over time (average of 26% of projects between both complexity measures), fluctuate (20% of projects), experience sudden drops (8% of projects) or a major increase in complexity (24% of projects).

## II. Background - Data Version Control

Data Version Control (DVC), publicly announced on the 4th of May 2017 [12], is a data/model versioning tool that is integrated with git repositories, such that the history of data, models, and code can evolve together in an efficient manner. Its main features include data versioning, data pipelines (reproduction), and data access (remote storage).

The most obvious functionality of tools like DVC is tracking the evolution of data and model files in a project. Given the large size of such files, git is ineffective for storing them. Thus, tools like DVC utilize a mechanism based on hashes and a local cache to manage the tracking, storing, and sharing of such large files. The *dvc add* command creates a text file containing the md5 hash of a data (or model) file and a URI to its storage location. Instead of having to store the actual data (which is automatically added to the .gitignore file), it is the generated text file that is versioned in git, just like regular source code files. Any change to the data results in a new revision of the text file, containing the new md5 hash and location of the updated data. Any modern cloud storage can then be used to store the physical data/models, while a local cache is used to cache data during model training and testing. Similar to git, the *dvc push* and *dvc pull* commands are used to save and retrieve data files from their remote storage location [13].

Apart from setting up data/model tracking, DVC allows to declaratively define the pipeline that will take the tracked data sets as inputs to create and experiment with models. A DVC pipeline basically is a dependency graph in which each node specifies a particular command that will be run, taking incoming edges (data and/or scripts) as inputs and producing output files that can be used by dependent nodes. Each DVC node is created with the *dvc run* command, yielding a .dvc file as shown in Figure 1 that specifies the command to be run, the two inputs (a directory and a file, both with their current DVC checksum), and the output .json file.

**evaluate.dvc**

```
cmd: python code/evaluate.py
deps:
- md5: d751713988987e9331980363e24189ce.dir
  path: images/test
- md5: 5a6d5b4a35a3ff13b0571f0703694886
  path: model/model.h5
outs:
- cache: false
  md5: 346b27c0aedfe97bc12ca8C584d303F3
  metrics: true
  path: model/metrics.json
wdir: .
```

Figure 1: Example DVC file representing pipeline node.

Similar to "make" build dependency graphs [14], DVC can incrementally execute a pipeline in order to only rerun nodes whose inputs have changed. Combined with DVC's integration with git, pipelines allow reproducing prior models just by checking out the corresponding git commit and *dvc pull*-ing the remote data.

## III. Study Design

This section presents the overall design of our empirical study on the usage of data/model versioning tools and pipelines in open source projects, addressing the research questions introduced in the introduction. The data and scripts used in our analysis are available online [15].

### A. Project Selection

Our empirical study focuses on GitHub projects that adopted the DVC ML versioning tool in their repository. Compared to other similar ML versioning tools (*i.e.,* MLflow, Polyaxon, comet.ml), DVC is considered the most popular tool of data versioning [16]. It is a lightweight, open-source technology designed to be integrated into git repositories and to be programming-language agnostic [17]. Furthermore, DVC provides support for maintaining repositories of non-ML projects, such as traditional software projects that gradually add ML capabilities [18], or pure data repositories.

We used GitHub's advanced search feature (on February 28, 2020) to identify all projects that use DVC. For this, we narrowed our search to projects with ".dvc" files containing the "md5" keyword (cf. Figure 1). Our query returned a result set of 391 projects that have used DVC at some point in their history.

### B. DVC Coupling Analysis

**Filtering Projects**. To answer RQ2 and RQ3, we further filter our initial 391 projects using a manual analysis to discard any toy projects (projects that only try out the DVC tool) or unrelated projects (*i.e.,* projects related to training workshops). We then only considered projects

Table I: Characteristics of the selected DVC projects (alphabetically ordered).

| **Projects name** | **# DVC commits** | **# Months of DVC usage** | **Total # Merged pull requests** | **# DVC files** | **# total files** |
|---|---|---|---|---|---|
| **aerubanov/DS_for_Air** | 98 | 4.71 | 36 | 12 | 1427 |
| AlessandroVol23/kdd-cup-2019* | 56 | 3.11 | 2 | 38 | 178 |
| antonkulaga/rna-seq* | 15 | 1.75 | 1 | 29 | 284 |
| arsalanc-v2/mailsense | 3 | 2.51 | 1 | 17 | 38 |
| **bmeg/bmeg-etl** | 299 | 13.86 | 147 | 217 | 416 |
| BoiseState/bookdata-tools | 87 | 5.83 | 0 | 123 | 149 |
| **CandelaV/UNHCRs_population_of_concern_HDX*** | 8 | 7.56 | 17 | 3 | 72 |
| DAGsHub/client[a] | 3 | 4.62 | 3 | 6 | 28 |
| deep-projects/pcam-with-dvc-cc[a] | 42 | 4.86 | 0 | 52 | 57 |
| deep-projects/pcam-with-dvc-cc-sshfs[a] | 19 | 4.86 | 0 | 33 | 36 |
| gmrukwa/publication-domain-discernibility | 14 | 2.96 | 2 | 10 | 36 |
| **jmhsi/lendingclub** | 14 | 5.18 | 14 | 17 | 188 |
| **kaiml/lease-eda*** | 86 | 0.42 | 51 | 8 | 27 |
| kjappelbaum/learn_mof_ox_state | 11 | 3.84 | 0 | 9 | 39 |
| MikeG27/BCI_Image_EEG_Generator | 17 | 1.55 | 0 | 13 | 59 |
| MIR-MU/regularized-embeddings[a] | 13 | 2.41 | 0 | 3 | 13 |
| **opt-out-tools/study-online-misogyny** | 24 | 5.1 | 11 | 14 | 57 |
| **phramer/phramer** | 24 | 0.43 | 14 | 5 | 60 |
| **piojanu/Planning-in-Imagination** | 3 | 4.17 | 12 | 9 | 146 |
| shuiblue/INTRUDE-refactor | 7 | 5.06 | 1 | 6 | 51 |
| **spacy-pl/utils** | 46 | 6.67 | 20 | 67 | 138 |
| **src-d/formatml** | 24 | 2.06 | 62 | 22 | 125 |
| Valentyn1997/xray* | 39 | 8.85 | 0 | 3 | 70 |
| wildlyclassyprince/data-playground* | 8 | 2.49 | 1 | 31 | 124 |
| yukw777/yureka | 31 | 7.7 | 0 | 37 | 102 |

[a] Projects only using DVC data tracking. * Projects only using DVC pipelines. **Projects with >10 work items (PRs).**

Table II: File type classification examples.

| Source | Test | Data | Gitignore |
|---|---|---|---|
| *.sh, *.py, *.go | */test/*.(sh, .py, .go) | *.h5, *.csv, *.dot | .gitignore |

that used DVC for more than 10 days (62 projects). Finally, we select the 25 most active projects based on the number of commits in their master branch, yielding the final dataset in Table I.

**Project file classification**. We manually classified the files of the 25 filtered projects of Table I as either DVC artifacts or traditional software artifacts. All files with a ".dvc" extension are classified as DVC artifacts, then further classified into three sub-categories: DVC files within the ".dvc" folder (which contains DVC metadata, similar to git's ".git" folder) are classified as `dvc utilities`, files containing the *"deps"* and *"cmd"* keywords (cf. Figure 1) are considered as `dvc pipeline` files, while all other DVC files are classified as `dvc data` files.

Next, we classify the non-DVC files in each project as either source code, tests, data (these are data files that somehow are not put in DVC), gitignore files, and "others". It should be noted that since most of the projects were in Python, they did not have build files. This classification was performed based on a combination of file extensions and the Github Linguist tool [19], followed by a manual verification, which helped to identify the (programming) language used in files with unfamiliar extensions. Table II provides example file classifications.

**Assigning Commits and Pull Request Work Items to Classified Files**. After classifying the DVC and source code files, we identify both the commits in which each of these files were touched (added, deleted, or modified), as well as the work items. Work item aggregation provides a more reliable analysis that considers a pull request as a unit of work [10], enabling the identification of coupling between separate, but related commits. While the commit analysis considers all 25 projects, our work item analysis focuses on the 10 projects in Table I that have at least 10 closed and merged pull requests (using GrimoireLab's Perceval [20]).

**Association Rules**. To measure the coupling between DVC files and other project files, we use association rules, similar to earlier papers in this field [10], [11]. Such an association rule is of the form A⇒B, describing the possible coupling of changes to file type A (e.g., "source code") implying changes to file type B (e.g., "DVC data file"). We use the conventional [21] metrics of "Support" (Supp), "Confidence" (Conf) and "Interest" (Lift) to measure the importance of an association rule. Supp(A) indicates the frequency of appearance of A, while Conf(A⇒B) indicates the percentage of times a change of A will happen together ("is coupled") with a change of B. Moreover, Lift(A⇒B) measures the degree to which the coupling between two file categories is different and independent from each other.

$$Supp(A \Rightarrow B) = Supp(B \Rightarrow A) = P(A \bigcap B) \quad (1)$$

$$Conf(A \Rightarrow B) = \frac{P(A \bigcap B)}{P(A)} = \frac{Supp(A \Rightarrow B)}{P(A)} \quad (2)$$

$$Lift(A \Rightarrow B) = \frac{P(A \bigcap B)}{P(A)P(B)} = \frac{Conf(A \Rightarrow B)}{P(B)} \quad (3)$$

We use a $\chi^2$ chi-squared test [22] to validate the statistical significance of the coupling between changes to A and B. If the $X^2$ statistic is greater than 3.84 ($\alpha = 0.05$), we consider that A and B have a statistically significant coupling. Otherwise, the observed relationship is due to chance. With $n$ the total number of commits or work items in a project:

$$X^2(A \Rightarrow B) = n(Lift - 1)^2 \frac{Supp * Conf}{(Conf - Supp) * (Lift - Conf)} \quad (4)$$

### C. Pipeline Complexity Analysis

In order to estimate the overhead that pipeline descriptions represent for data engineers/scientists and developers, we use two measures of pipeline complexity, i.e., McCabe (graph structure complexity of pipelines) and Halstead (effort to understand the textual form of the .dvc pipeline specification files).

**McCabe complexity**. We adapted the notion of McCabe Cyclomatic Complexity [23] M for pipelines as follows:

$$M = E - N + 2P \quad (5)$$

where $N$ is the number of nodes in the pipeline graph, $E$ is the number of edges linking the nodes, and $P$ is the number of connected pipelines in the project. Similar to McCabe calculations on the control flow graph of a method, instead of directly calculating $P$, we connect all pipeline input nodes to a common dummy input node and all output nodes to a common dummy output node. This results in $P$ equal to 1.

**Halstead complexity**. Halstead complexity has been applied before on source code [24], build dependency graphs [10], etc., and measures the amount of mental effort required to recreate a program:

$$Effort = (\tfrac{n1}{2} * \tfrac{N2}{n2}) * ((N1 + N2) * \log_2(n1 + n2)) \quad (6)$$

where n1 is the number of distinct operators, n2 the number of distinct operands, N1 the total number of operators, and N2 the total number of operands.

Following the example in Figure 1 (Section II), the operators in the case of DVC would be the top-level commands/configuration of a DVC file such as *cmd*, *deps*, *outs*, while the operands will be the parameters passed to the operators (e.g., *path*, *wdir*, *repo*, etc). We consider counting these parameters across all .dvc files constructing a pipeline.

## IV. **How common is the usage of DVC in Github projects?** (RQ1)

### A. Motivation

Similar to other aspects of software engineering for ML applications, little to nothing is known about how current ML versioning tools (like DVC) are used in open-source projects: how much do projects rely on these tools, and what are their most popular features in practice?

### B. Approach

To answer this question, we analyzed the 391 open-source GitHub projects that use DVC selected in Section III-A. For each selected project, we calculate statistics such as the number and evolution of DVC files, days taken to adopt DVC. The adoption/usage of DVC within projects is identified by the commits that touch at least one DVC file. We also identify the specific DVC features (data versioning, data pipelines, and remote data storage) adopted/implemented in the project based on the classified file categories (Section III-B). For example, a project is considered to use DVC's pipeline feature as soon as a commit touches at least one DVC pipeline file.

Figure 2 shows how quickly the studied projects adopted DVC after creating the repository. Figure 3 shows how long a project has used/been using DVC since the first commit introducing a DVC file. Figure 4 shows the prevalence of the different remote storage configurations that were used to provide a storage location for sharing data and models.

Figure 5 shows the median proportion of modified DVC files across the studied projects for different phases in their lifetime, to better understand the amount of maintenance involved with such files. Since the studied projects have different lifetimes, we divided each project's lifetime into 10 phases: the first 10% of the total commits, the next 10%, etc. It should be noted that only 199 projects are included in this particular analysis, basically excluding projects with less than 10 commits in total.

### C. Results

**It took more than 9 months after its initial release for DVC to be adopted in any of the studied open-source projects.** The oldest DVC commit observed in our dataset was in the cats_vs_dogs [25] project. The 6 studied projects dating back before the release of DVC took a median of 23 months to start using DVC after the first DVC official release, with a minimum of 17 months for the seq2seq-chatbot [26] project.

Among the 385 remaining projects (created after the DVC release), **294 projects adopted DVC on the first day of their repository creation**, and 56/385 further projects adopted DVC within the first month of creating the repository. An additional 30/385 projects adopted DVC within their first year, while it took over a year for 5/385 projects to adopt DVC.

**25% of the projects applied DVC for more than a week, i.e., are past the exploration and experimentation stage of DVC.** Conversely, 230/391 (58.8%) of the projects only used DVC for less than a day while 62/391 (15.8%) used DVC between 1 day and 1 week (cf. Figure 3). Among the projects that used DVC for less than a day, 96/391 (24.6%) projects had one DVC commit only (likely for testing). This explains why only a median of 3 commits modifying DVC files was observed in the studied

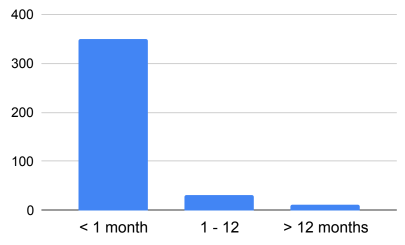


Figure 2: #projects adopting DVC a given period after DVC's launch.

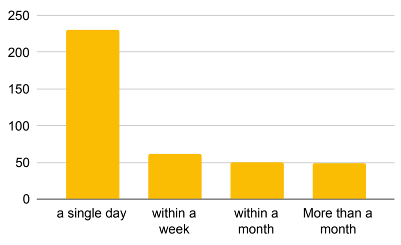


Figure 3: #projects using DVC for a given period.

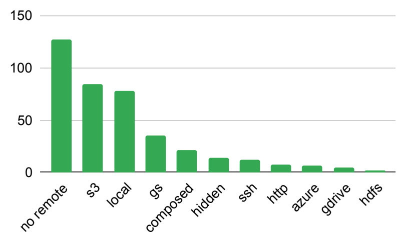


Figure 4: #projects using the different DVC remote storage options.

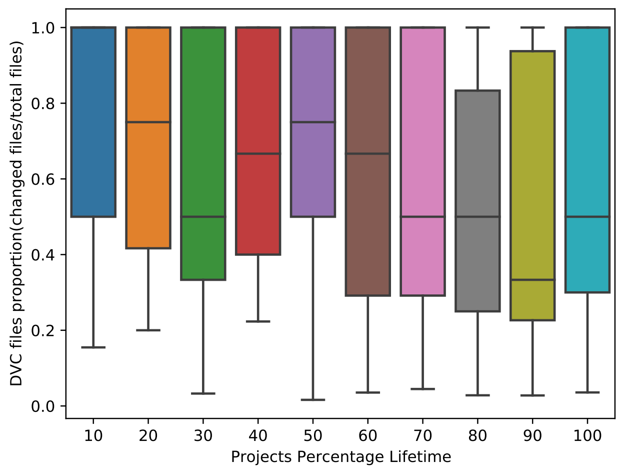


Figure 5: Distribution of the median proportion of DVC files changed by project commits (chronologically grouped into 10% chunks).

projects. The bmeg project [27] had the maximum number of DVC-related commits, with 299 commits.

**50% or more of the DVC files in a project are changed at least once every one-tenth of the project's lifetime.** We observe from Figure 5 that DVC files are changed frequently within the 199 projects (with 10 or more total commits). The only exception is observed at the 90% mark of the timeline, with a median proportion of 33.3% changed DVC files across the studied projects. A total of 5,192 (761) DVC-related file additions (deletions) were performed during the entire lifetime of the projects.

An analysis of the changed files within all DVC-related commits (total of 2,653 commits) in our dataset shows that **data versioning is the first DVC feature** implemented in most projects (added in the first commit of 321/391 projects), while **model pipelines is the second feature adopted.** In total, 1,855 commits modified parts of the projects' DVC files related to data versioning, 1,093 commits modified dvc utility files that contain configurations on remote data storage, and 924 commits modified dvc pipeline-related content. 279/391 (71.4%) projects adopted at least two features throughout their lifetime, while 73 (18.7%) projects ended up using all three of the main DVC features. The remaining 206 (52.7%) projects used only two of the DVC features.

**Amazon S3 (84 projects), the local cache (78), and Google Cloud Storage (35) are the three top used data storage locations**. In addition, 21 projects configured multiple remote storage locations. This phenomenon can be attributed to several factors such as (1) experimentation with different storage platforms and (2) the need to duplicate data on different servers to ensure a higher level of safety. These combinations of storage locations are represented as *composed remote* in Figure 4. The most commonly used combination was *[http, s3]*.

On the other hand, for 127/391 of the projects, we did not find any trace of their DVC remote in their GitHub repository. 84/127 (66%) of these projects only have a single day of DVC usage, while the other projects might have left out the remote information for confidentiality reasons. In fact, we found that 14/391 were not sharing their DVC config file at all in the public repository, likely because their local config file contained sensitive remote storage information [28]. This lack of sharing could be a threat to further empirical studies of ML versioning.

*Despite ML versioning being a young practice in open source repositories, 71.4% of the studied projects use at least two of the main DVC features, i.e., data versioning and pipelines. More than half of the DVC files within projects past the experimentation stage are frequently changed, suggesting non-negligible maintenance effort for practitioners.*

## V. How much coupling exists between software artifacts and DVC artifacts? (RQ2)

### A. *Motivation*

Studies have shown that changes to software artifacts like build and environment files that are coupled with traditional software artifacts such as source code, test files, introduce overhead as developers have to maintain the source code and tests together with these artifacts, and vice versa [10], [11]. Since RQ1 has shown that more than 25% of the studied projects are past the experimentation stage of their DVC adoption, and the software engineering of machine learning applications is a multi-disciplinary activity (data engineers, data scientists, and developers), this RQ aims to study the effort required to add or change data/model files and the ML pipeline specification in a project during regular development.

Table III: Number of projects with significant $\chi^2$ statistical test ($\alpha = 0.05$) for commit-level (pull request-level) coupling between DVC classes and/or source code artifacts.

| ↶ | dvc_pipeline | dvc_data | dvc_utilities | Gitignore | Source | Test | Data | Others |
|---|---|---|---|---|---|---|---|---|
| dvc_pipeline |  | 5 (3) | **10** (2) | **11** (**5**) | 5 (**4**) | 2 (**4**) | 8 (**4**) | 8 (**5**) |
| dvc_data | 5 (3) |  | 5 (**4**) | **16** (**6**) | 5 (**5**) | 2 (**5**) | 4 (3) | 3 (**5**) |
| dvc_utilities | **10** (2) | 5 (**4**) |  | 2 (**6**) | 6 (**5**) | 2 (**5**) | 1 (**4**) | 0 (**4**) |
| Gitignore | **11** (**5**) | **16** (**6**) | 2 (**6**) |  | 9 (**4**) | 4 (**4**) | 9 (**7**) | 5 (2) |
| Source | 5 (**4**) | 5 (**5**) | 6 (**5**) | 9 (**4**) |  | 4 (**4**) | 6 (**6**) | **21** (**4**) |
| Test | 2 (**4**) | 2 (**5**) | 2 (**5**) | 4 (**4**) | 4 (**4**) |  | 3 (3) | 5 (**4**) |
| Data | 8 (**4**) | 4 (3) | 1 (**4**) | 9 (7) | 6 (**6**) | 3 (3) |  | 8 (**6**) |
| Others | 8 (**5**) | 3 (**5**) | 0 (**4**) | 5 (2) | **21** (**4**) | 5 (**4**) | 8 (**6**) |  |

## B. Approach

This RQ performs two levels of coupling analysis between the manually identified categories of files: a fine-grained commit-level analysis and a more coarse-grained pull request-level analysis (see Section III-B). The commit-level coupling analysis is performed on the 25 most active projects in our dataset (based on the number of commits in their master branch), while the pull request-level analysis is performed on the 10 projects having more than 10 merged pull requests (bold projects in Table I).

Figure 6 and 8 show the commit-level and pull request-level coupling (i.e., confidence value of association rules), respectively, between the 3 types of DVC files. It should be noted that only 15/25 (8/10) of the studied projects at the commit-level (PR-level) contain both DVC data and pipeline files. Also, since no prior work has studied the coupling within ML versioning artifacts, we adopt the thresholds used in related work [10], [11] to identify high and low coupling results (coupling below 12% is considered low). Figure 7 shows the commit-level coupling between the DVC and source code artifacts, while Figure 9 shows the corresponding pull request-level coupling.

Finally, Table III shows the number of projects having statistically significant coupling between the different DVC and software artifacts at the commit-level and pull request-level. This helps us determine which of the coupling observations or statistically significant vs. due to noise. The table shows in bold the coupling relations that were significant for at least 40% of the studied projects ($\geq$10/25 for commit-level, and $\geq$4/10 for pull request-level). Note that the $\chi^2$ test is symmetric (same value for A$\Rightarrow$B and B$\Rightarrow$A).

## C. Results

### 1) Commit-level coupling:

**Finding 1. 6.25% of commits changing dvc-utilities co-occur with the changes in dvc-pipeline.** While low, this is the only commit-level coupling relation that is significant for at least 40% of the projects. The highest median coupling value in Figure 7 is 7.14% for dvc-data$\Rightarrow$dvc-utilities. Contrary to expectations, we observe that the median coupling between DVC data files and DVC pipeline files is 0, in either direction (Figure 6). Although DVC pipeline files have to be modified to contain the updated *md5 references* of changed DVC data files, we observe that such changes tend to occur in separate commits (cf. work item-level analysis below).

**Finding 2. The commits changing DVC files mainly tend to change the gitignore file (significant) and source code.** We observe from Figure 7 that Conf(dvc-utilities$\Rightarrow$source), Conf(dvc-pipeline$\Rightarrow$source) and Conf(dvc-data$\Rightarrow$source) have high median values of 50%, 46.6% and 33.3% respectively. This indicates that up to half of the changes to DVC files require a change to the source code, for example, to a Python file that is a part of the workflow of the tracked pipeline. However, none of these coupling relations was significant in at least 40% of the studied projects.

In contrast, 16 (10) projects have a significant DVC data$\Rightarrow$gitignore (DVC pipeline$\Rightarrow$gitignore) coupling, yielding high median coupling values such as 63.6% for DVC$\Rightarrow$gitignore files. This result can be attributed to the fact that whenever a new file or data repository is added to DVC, it will be followed by an automatic update of the gitignore files to specify the files/folders paths that need to be untracked by Git and instead added to the DVC cache or remote (as described in Section II).

**Finding 3. Commits changing source or test files rarely require changes to DVC files.** Conf(source$\Rightarrow$dvc-pipeline) with a median value of 7.4% is the highest observed coupling value between non-DVC and DVC files (the median Conf(test$\Leftrightarrow$DVC) even is 0). This result could be explained, since, first, there are more source files (with proportionally more changes) than DVC files, and, second, only a minority of the code within an ML application is actually related to ML-specific tasks/features, and therefore to the DVC pipeline [29].

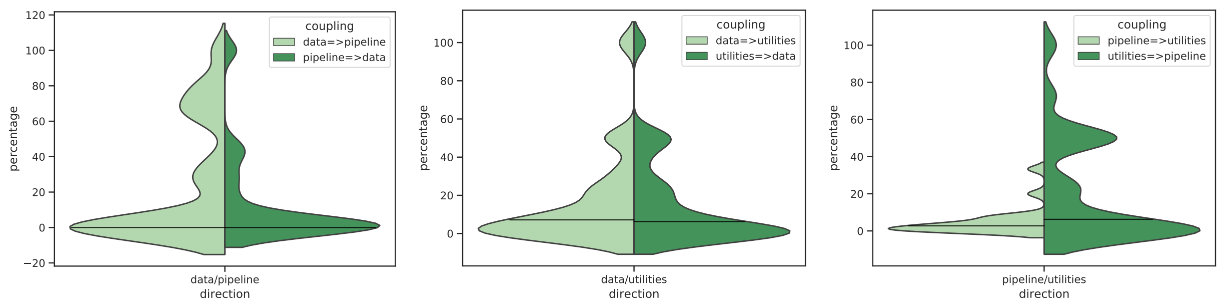
*Although there is low commit-level coupling amongst the DVC files of a project, most coupling observed with dvc-utilities and software artifacts are automated by DVC. On the contrary, DVC files and software artifacts such as tests and data files are rarely changed together at the commit-level.*

### 2) Pull request (work item) level coupling:

**Finding 4. DVC data and utilities files have high (significant) median PR-level coupling of 29.5% and 23.3%.** We observe from Figure 8 that PR-level Conf(dvc-data$\Rightarrow$dvc-pipeline) and Conf(dvc-pipeline$\Rightarrow$dvc-data) have the highest median values of 54.4% and 31.4%, yet only for 3/10 projects. Conf(dvc-data$\Rightarrow$dvc-utilities) and Conf(dvc-utilities$\Rightarrow$dvc-data) have slightly lower median values of 29.5% and 23.3%, but significant for at least 4/10 projects.

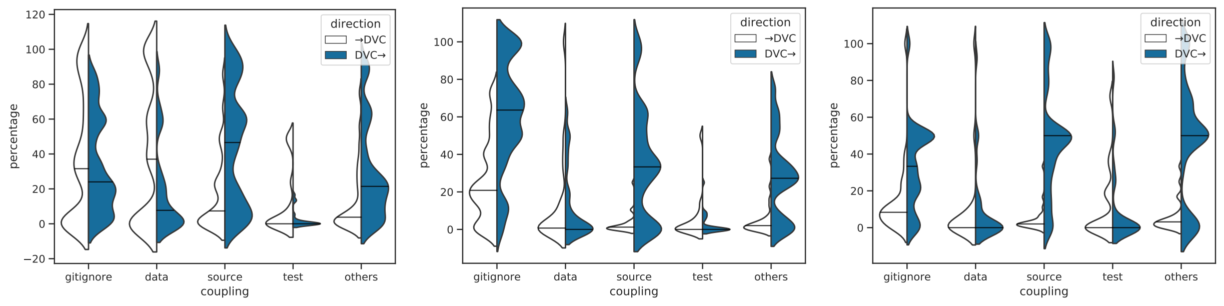
**Finding 5. PR-level analysis confirms finding 2, with source code, test and data files now also showing statistically significant coupling with DVC categories.** In total, 9 of the 10 projects have a significant coupling between at least one DVC and one software artifact category at the pull request level. Almost all coupling relations are significant for at least 4/10 projects.

Figure 9 shows that Conf(dvc-utilities$\Rightarrow$source), Conf(dvc-pipeline$\Rightarrow$source) and Conf(dvc-data$\Rightarrow$source)



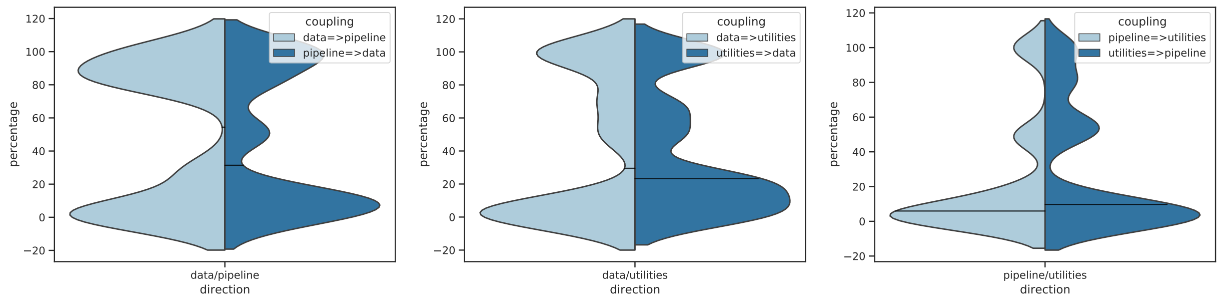
(a) DVC data/pipeline (b) DVC data/utilities (c) DVC pipeline/utilities

Figure 6: Commit-level coupling amongst DVC categories.



(a) DVC pipeline/software artifacts (b) DVC data/software artifacts (c) DVC utilities/software artifacts

Figure 7: Commit-level coupling between DVC and other software artifact categories.



(a) DVC data/pipeline (b) DVC data/utilities (c) DVC pipeline/utilities

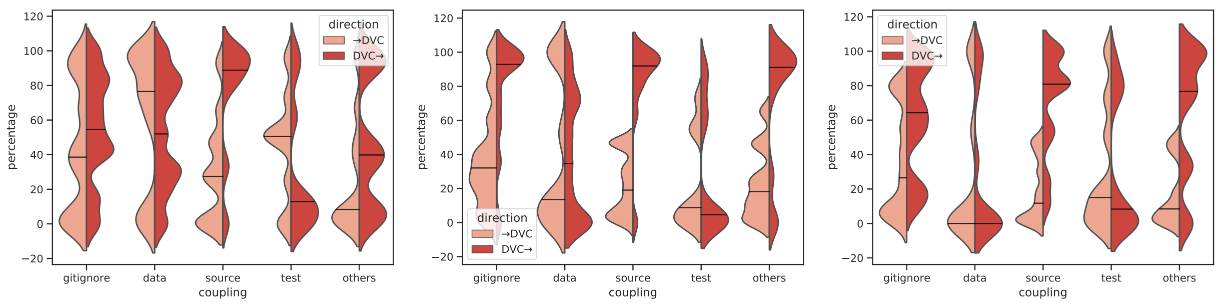
Figure 8: PR-level coupling amongst internal DVC categories.

have a high median values of 80.90%, 88.82% and 91.91% respectively. This indicates that at least 8 out of 10 pull requests changing DVC files require a change to the source code. This makes sense since changes in the structure of an ML pipeline might require the scripts inside the pipeline to accept more or fewer arguments, or to split the output into multiple files. Furthermore, similar to the commit-level analysis, high median coupling values are observed between the DVC and gitignore files (e.g., Conf(dvc-data⇒gitignore) has a median value of 92.79%).

Furthermore, we observe high median coupling values of 51.9% and 34.7% for Conf(dvc-pipeline⇒data) and Conf(dvc-data⇒data). Yet, the coupling between DVC categories and test files, while slightly higher than the observed coupling at commit-level, is still low.

**Finding 6. Changes to software artifacts tend to change DVC files at the PR-level, unlike the commit-level results.** We noticed that Conf(gitignore⇔DVC) again has the highest median coupling values. Similarly, source and test files recorded median coupling values of 27.46% for Conf(Source⇒DVC-pipeline) and 50.54% for Conf(test⇒DVC-pipeline). In other words, one out of four pull requests changing a source code file, and one out of two pull requests changing a test file, require changes to the DVC pipeline. In fact, there are more projects with statistically significant coupling between source code and DVC utilities/data (5) than between source code and tests (4)! This indicates a substantial maintenance effort for ML pipelines both during source code development and testing.

*Coupling between DVC and software artifacts are much stronger than would be expected by chance, with one out of four PRs changing source code, and one out of two PRs changing tests, requiring changes to pipeline files.*



(a) DVC pipeline/software artifacts (b) DVC data/software artifacts (c) DVC utilities/software artifacts

Figure 9: PR-level coupling between DVC and software artifact categories.

## VI. **How does the complexity of the DVC ML pipeline evolve over time?** (RQ3)

### *A. Motivation*

Previous studies have shown that source code [30]–[33] and build code [10] evolve in terms of complexity and size, providing another source of maintenance effort apart from change frequency (RQ1) and file coupling (RQ2). This RQ aims to study whether these previous findings on complexity generalize to ML versioning technologies.

### *B. Approach*

To compute the complexity evolution of the DVC pipeline, we check out all commits where a DVC file was changed. For each checked-out commit, we measure the McCabe and Halstead complexities for DVC pipelines using the formulas introduced in Section III-C. We then plotted the evolutionary trend of complexity over time for each project, such as shown in Figure 10 for McCabe complexity . These trends, both for McCabe and Halstead (not shown), were manually classified by all authors into five categories (identified during the classification process): constant, increasing, fluctuating, sudden drop, and major impact. Constant trends that experience a sudden increase in complexity are classified under major impact while increasing trends that suddenly decrease significantly are classified under sudden drop. It should be noted that our results here exclude 1.3% of the DVC commits that had DVC files that were corrupted by unfixed merge conflicts, leading to syntactic errors [34].

### *C. Results*

**18/25 of the projects have a non-constant McCabe complexity trend.** 6/25 projects have an increasing trend in McCabe complexity. An example of such increasing complexity trends is shown in Figure 10a for the utils project, where the complexity was increasing gradually during the entire project's lifetime.

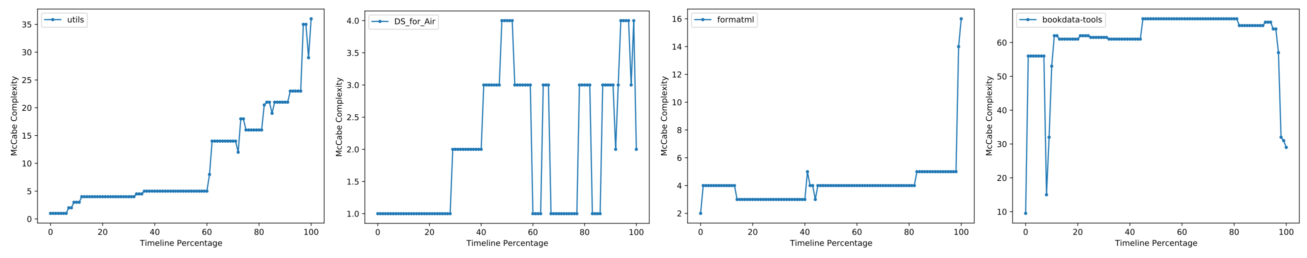
5/25 projects have a high fluctuation in the McCabe complexity. Such fluctuations are caused by frequent removals and additions of DVC files or pipeline stages. For example, in the DS-for-air project shown in Figure 10b, there are several periods within which DVC files are removed or different separate pipelines are merged to reduce the complexity.

We observe that 5/25 projects had a single change at the end of their project lifetime that suddenly increased their complexity. For example, Figure 10c shows a sudden increase in the complexity of the formatml project after additional DVC files and stages were added to track new data files. Similarly, 2/25 of projects show a sudden drop in complexity. For example, in the bookdata-tool project shown in Figure 10d, there is a sudden drop in the complexity caused by a deletion of 20/85 DVC stages in two changesets.

**Similar to the McCabe trends, 21/25 projects have a non-constant Halstead complexity trend.** 7/25 projects experience a sudden increase in complexity due to a single major change to their pipelines. Further investigation shows that this sudden increase is due to large (tangled) commits that introduce multiple stages to the pipeline and increase the total number of operators and operands (N1 and N2).

We also observe an increasing trend in 7/25 projects, a fluctuating trend in 5/25 projects, and a sudden drop in 2/25 projects. The fluctuating Halstead complexity was caused by either refactoring (restructuring) the DVC files, auto-generation of the pipeline via Makefile, or the manual removal of unwanted auto-generated DVC files. The sudden drop in complexity was due to a partial deletion of DVC files in a merged branch (in the phramer project) and a split of the pipeline into multiple components (in the bmeg-etl project).

**The observed median McCabe and Halstead complexities over time do not correlate, with a Pearson correlation value of 0.145 (p-value of 0.48).** As shown in Figure 11, an increase in the McCabe complexity does not correlate with an increase in the Halstead complexity of a project and vice versa. The McCabe complexity metric is primarily concerned with the number of decision points in the generated pipeline graph. The Halstead metric on the other hand focuses on the file's verbosity (operands and operators). Hence, while the pipelines' structural (McCabe) complexity is similar



(a) Increasing trend (b) Fluctuate trend (c) One Major Impact (d) Sudden Drop

Figure 10: McCabe Complexity

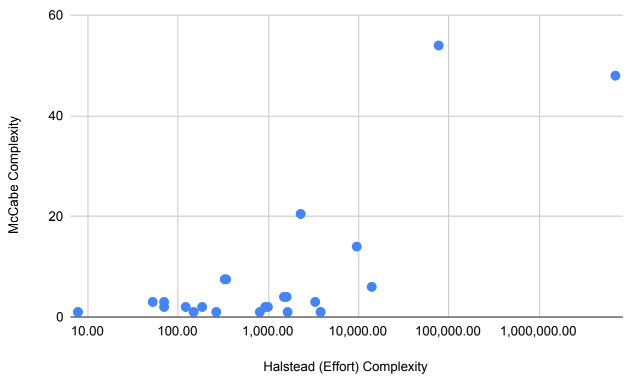


Figure 11: McCabe vs. Halstead Complexity

for most projects, the textual (Halstead) complexity varies substantially between projects.

*While no general trends could be observed about the complexity of DVC files, an average of 78% of the projects (between both complexity measures) had a non-constant complexity trend for DVC files. The structural McCabe and textual Halstead complexity measures do not correlate.*

## VII. Implications of our findings

**Implications to ML application developers**. Our results in RQ1 show that most studied Github projects are still exploring and experimenting with DVC. As a result, we observed a number of cases that do not follow the recommended DVC practices. First, there are cases in which developers add their DVC cache to git (e.g., the DS_for_Air project [35]) instead of using the remote storage functionality provided by DVC. Secondly, we observe the creation of DVC file for each individual data file (e.g., in the bgc-pipeline [36] and dvc-test [37] projects). This practice increases the complexity and effort required to understand or maintain the pipeline. Developers might consider to either group stages together (e.g., regroup similar data files in a folder and only create the DVC stage for the regrouped folder) or break down pipelines into sub-components to facilitate maintenance tasks and reduce the complexity of a pipeline.

**Implications to ML versioning tool developers/-companies**. Interactive environments such as Jupyter notebooks currently are amongst the most popular means used by data scientists for interactive development and presentation of data science projects. The cells within these notebooks are analogous to pipeline stages. However, current ML versioning tools like DVC are only able to track the entire notebook file, not the individual cells. For example, the regularized-embeddings [38] project is using only one DVC file to track a jupyter notebook file containing 35 cells. The fact of being programming-language agnostic makes it impossible for tools like DVC to provide fine-grained pipeline support for notebook-style artifacts.

**Implications to Researchers.** As shown by the key findings of this study, the high coupling and the non-constant trend of complexity of the DVC pipelines introduce overhead to developers during maintenance tasks such as fixing a bug in the pipeline, adding new dependencies, or reverting to an old pipeline stage. Thus, it is easy for a novice developer to introduce a source code change without being aware of a required DVC change, and vice versa. This provides several avenues for researchers to propose techniques and tools that can assist developers to identify code changes that require DVC maintenance.

## VIII. Threats to validity

*Construct validity.* The use of the Halstead (and McCabe) complexity measure for DVC pipeline evolution may be considered unreliable [39] compared to measures such as average degree of a graph. However, we adopt the Halstead complexity definition based on the prior work of McIntosh *et al.* [40], given the similarity between the structure of build files and DVC pipelines, yet other definitions are possible. Also, the classification of project files and complexity trends were manually verified by all authors to mitigate the false positives introduced by the utilized tools (e.g. Linguist). In the coupling analysis of RQ2, we did not filter out tangled changes *i.e.,* commits/PRs that include multiple tasks in the same changeset (new feature, bug fix, code refactoring, etc.) [41].

*Internal validity.* While the observed coupling between DVC and software artifacts in commits and PRs can be due to noise, the reported $X^2$ provides statistical evidence that in many cases these coupling relationships are much stronger than would be expected by chance.

*External validity.* Our study analyzed the 391 GitHub projects that had adopted DVC (and were still active) by February 2020. To some extent, these projects can still be considered as early adopters of ML versioning technology, hence future research should replicate our study on more mature adopters of DVC, as well as on adopters of other ML versioning technology like MLFlow and Pachyderm, both in open- and closed-source systems.

## IX. Related Work

### A. ML versioning tools

The versioning of ML data and models is a young and growing practice, with several tools created to help developers with tracking the various aspects of their workflow. Airflow [42] is used to create, schedule and monitor ML workflows as a directed acyclic graph (DAG) that may be composed from multiple tasks. Similarly, Luigi [43] is a workflow engine framework that helps to write static and fault-tolerant data pipelines in Python.

Miguel *et al.* [44] presented the Marvin engine that supports the exploration and model development of distributed computing systems of data-intensive applications. It provides a standard interface to allow other applications access to shared model artifacts and to support high throughput and processing of large datasets.

MLflow [5] is a platform to streamline machine learning development. It basically is divided into three components for (1) tracking experiments, (2) packaging code into reproducible runs, and (3) sharing and deploying models trained using diverse ML frameworks. Kedro [45] provides a development workflow framework that implements software engineering best-practice for data-pipeline construction, basically leveraging data abstraction and clear code organization to bring models into production.

Pachyderm [6] is a data science platform aimed for enterprise that combines data lineage [3] with end-to-end pipelines on Kubernetes, with a graphical pipeline builder and data versioning.

The choice of DVC [4] in this work is based on its support for gradual adoption of ML capabilities in traditional software projects. This enables empirical analysis of the coupling between source code/DVC artifacts and pipeline complexity in open source projects.

### B. Software evolution

A large body of research exists on co-evolution [11], [46]–[48] and change coupling [49]–[52] , here we focus on studies that have been conducted on the co-evolution of (non) traditional software artifacts. Zaidman *et al.* [46] studied how production and test code co-evolve using projects' version control systems, code coverage reports, and size-related metrics. They find that test coverage is positively correlated with the percentage of test code in the system. They also observe that changes in production code are reflected in tests; new tests are added and old tests are modified to ensure a running test suite.

McIntosh *et al.* [10] explored the complexity of the GNU Make build systems of nine large open-source projects, by measuring the coupling between build- and source code-related files in individual commits and work items. They found that the build maintenance increases the overhead of the source code development and test activities respectively by 27% and 44%, similar to our PR-level findings in RQ3. Jiang *et al.* [11] investigated the co-evolution of Infrastructure-as-Code (IaC) and other software artifacts in OSS repositories, again observing co-evolution relations between IaC and other artifacts like Makefiles.

Passos *et al.* [53] studied the changes in the variability model of the Linux kernel due to changes to related artifacts (i.e., Makefiles and C source code) to extract the co-evolution patterns for the Linux kernel variability model. They created a catalog of patterns describing how certain classes of changes affect different code artifacts.

McIntosh *et al.* [40] studied the evolution of ANT build system specifications using software metrics adopted from the source code domain. They find that ANT build scripts continuously change in existing software projects and have a growing trend in complexity, in sync with changes to the projects' source code.

In this paper, we analyzed the coupling between DVC changes and source code artifacts at commit and pull request granularity levels. We also studied the evolution of the machine learning pipeline complexity using McCabe and Halstead complexity.

## X. Conclusion And Future Work

Software engineering of machine learning applications has led to the introduction of novel tools in the software engineering process, such as ML versioning tools to track and reproduce data, models and pipelines over time. Our empirical study on the DVC versioning tool shows that ML versioning is a growing practice in open source repositories, and it imposes a non-negligible maintenance overhead for developers and data scientists working on machine learning applications.

In particular, we find that 25% of the studied projects are past the initial exploration phase with DVC, and half of them frequently change DVC-related files (especially DVC data files). Furthermore, one out of four PRs changing source code (and one out of two changing test code) require changes to DVC pipeline files. Finally, 78% of the projects (average between both complexity measures) exhibit a non-constant complexity of DVC files over time.

Armed with this understanding, we plan to investigate recommendation models for DVC-related maintenance activities. Furthermore, we plan to extend this study to cover additional ML versioning tools such as MLFLow and also gather insights (through surveys) from developers familiar with these tools.

***Acknowledgment:*** *We would like to thank DVC's Dmitry Petrov and his team for their patience and help to understand the finer details of DVC.*

## References

[1] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2019, pp. 291–300.

[2] C. W. Danilo Sato, Arif Wider, "Continuous delivery for machine learning," https://martinfowler.com/articles/cd4ml.html, 09 2019.

[3] H. Atwal, "Dataops technology," in *Practical DataOps*. Springer, 2020, pp. 215–247.

[4] "Data version control · dvc," https://dvc.org/, (Accessed on 06/05/2020).

[5] "Mlflow - a platform for the machine learning lifecyclew," https://www.mlflow.org/, (Accessed on 10/07/2020).

[6] "Pachyderm — version-controlled data science," https://www.pachyderm.com/, (Accessed on 06/05/2020).

[7] "Modeldb: Home," https://senselab.med.yale.edu/modeldb/, (Accessed on 06/13/2020).

[8] "Quilt is a versioned data portal for aws," https://quiltdata.com/, (Accessed on 06/13/2020).

[9] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[10] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *2011 33rd International Conference on Software Engineering (ICSE)*, May 2011, pp. 141–150.

[11] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code - an empirical study," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 45–55.

[12] D. Petrov, "Release beta-release · iterative/dvc," https://github.com/iterative/dvc/releases/tag/0.8.1, (Accessed on 07/27/2020).

[13] "remote add — data version control · dvc," https://dvc.org/doc/command-reference/remote/add.

[14] S. I. Feldman, "Make—a program for maintaining computer programs," *Software: Practice and experience*, vol. 9, no. 4, pp. 255–265, 1979.

[15] A. Barrak, "Link to the data/scripts used in the study," https://zenodo.org/record/4434719, January 2021.

[16] "kerneld4769833fe — kaggle," https://www.kaggle.com/rtatman/kerneld4769833fe, Jan 2019.

[17] C. Jansen, J. Annuscheit, B. Schilling, K. Strohmenger, M. Witt, F. Bartusch, C. Herta, P. Hufnagl, and D. Krefting, "Curious containers: A framework for computational reproducibility in life sciences with support for deep learning applications," *Future Generation Computer Systems*, 2020.

[18] P. Janardhanan, "Project repositories for machine learning with tensorflow," *Procedia Computer Science*, vol. 171, pp. 188–196, 2020.

[19] "github/linguist: Language savant. if your repository's language is being reported incorrectly, send us a pull request!" https://github.com/github/linguist, (Accessed on 05/18/2020).

[20] valerio Cosentino, "animeshk08/grimoirelab-perceval: Send sir perceval on a quest to retrieve and gather data from software repositories." https://github.com/animeshk08/grimoirelab-perceval, (Accessed on 10/24/2020).

[21] A. A. Sergio, "Chi-squared computation for association rules: preliminary results," *Technical Report BCCS-03–01*, 2003.

[22] S. A. Alvarez, "Chi-squared computation for association rules: preliminary results," *Boston, MA: Boston College*, vol. 13, 2003.

[23] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[24] T. Hariprasad, G. Vidhyagaran, K. Seenu, and C. Thirumalai, "Software complexity analysis using halstead metrics," in *2017 International Conference on Trends in Electronics and Informatics (ICEI)*. IEEE, 2017, pp. 1109–1113.

[25] D. Petrov, "Init dvc · dmpetrov/cats_vs_dogs@7fe0c2b," https://github.com/dmpetrov/cats_vs_dogs/commit/7fe0c2b, Feb 2018.

[26] G. COTER, "gcoter/seq2seq-chatbot: A chatbot using the seq2seq framework." https://github.com/gcoter/seq2seq-chatbot, (Accessed on 10/24/2020).

[27] K. Ellrott, "bmeg/bmeg-etl: Etl configuration for bmeg," https://github.com/bmeg/bmeg-etl, (Accessed on 10/24/2020).

[28] "config — data version control · dvc," https://dvc.org/doc/command-reference/config, (Accessed on 12/29/2020).

[29] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, p. 2503–2511.

[30] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Systems journal*, vol. 15, no. 3, pp. 225–252, 1976.

[31] M. M. Lehman, "On understanding laws, evolution, and conservation in the large-program life cycle," *Journal of Systems and Software*, vol. 1, pp. 213–221, 1979.

[32] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution-the nineties view," in *Proceedings Fourth International Software Metrics Symposium*. IEEE, 1997, pp. 20–32.

[33] Q. Tu *et al.*, "Evolution in open source software: A case study," in *Proceedings 2000 International Conference on Software Maintenance*. IEEE, 2000, pp. 131–142.

[34] A. aerubanov, "change fillnan method - aerubanov/proj_air_quality@7418752," https://github.com/aerubanov/DS/_for/_Air/commit/7418752, Jan 2020.

[35] ——, "partialy update data on 2019-12-08 - aerubanov/proj_air_quality@5a7c8f9," https://github.com/aerubanov/DS/_for/_Air/commit/5a7c8f9, Dec 2019.

[36] Jasonmvictor, "Initial commit · merck/bgc-pipeline@29300da," https://github.com/Merck/bgc-pipeline/commit/29300da, Jan 2019.

[37] T. Jensen, "adding dvc files individually · thorbenjensen/dvc-test@3186671," https://github.com/ThorbenJensen/dvc-test/commit/3186671, Dec 2019.

[38] V. Novotný, "Mir-mu/regularized-embeddings: Experimental code for the "text classification with word embedding regularization and soft similarity measure" (novotný et al., 2019) paper," https://github.com/MIR-MU/regularized-embeddings/, (Accessed on 10/24/2020).

[39] M. Shepperd and D. C. Ince, "A critique of three metrics," *Journal of systems and software*, vol. 26, no. 3, pp. 197–210, 1994.

[40] S. McIntosh, B. Adams, and A. E. Hassan, "The evolution of ant build systems," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 42–51.

[41] K. Herzig and A. Zeller, "The impact of tangled code changes," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 121–130.

[42] "Apache airflow," http://airflow.apache.org/, (Accessed on 06/04/2020).

[43] "spotify/luigi," https://github.com/spotify/luigi, (Accessed on 06/04/2020).

[44] L. B. Miguel, D. Takabayashi, J. R. Pizani, T. Andrade, and B. West, "Marvin-from exploratory models to production," *Journal of Machine Learning Research*, pp. 33–44, 2017.

[45] "quantumblacklabs/kedro: A python library that implements software engineering best-practice for data and ml pipelines." https://github.com/quantumblacklabs/kedro, (Accessed on 06/05/2020).

[46] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production test code," in *2008 1st International Conference on Software Testing, Verification, and Validation*, April 2008, pp. 220–229.

[47] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[48] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, “Cross-project build co-change prediction,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 311–320.

[49] S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan, “Mining co-change information to understand when build changes are necessary,” in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 241–250.

[50] H. Gall, K. Hajek, and M. Jazayeri, “Detection of logical coupling based on product release history,” in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 190–198.

[51] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[52] M. D’Ambros, M. Lanza, and R. Robbes, “On the relationship between change coupling and software defects,” in *2009 16th Working Conference on Reverse Engineering*, 2009, pp. 135–144.

[53] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wasowski, K. Czarnecki, P. Borba, and J. Guo, “Coevolution of variability models and related software artifacts,” *Empirical Software Engineering*, vol. 21, no. 4, pp. 1744–1793, 2016.