# Unboxing Default Argument Breaking Changes in Scikit Learn

João Eduardo Montandon*, Luciana Lourdes Silva†, Cristiano Politowski‡,
Ghizlane El Boussaidi‡ and Marco Tulio Valente*
*Universidade Federal de Minas Gerais, Brazil
jemaf@ufmg.br, mtov@dcc.ufmg.br
†Instituto Federal de Minas Gerais, Brazil
luciana.lourdes.silva@ifmg.edu.br
‡École de Technologie Supérieure, Canada
cristiano.politowski@etsmtl.ca, ghizlane.elboussaidi@etsmtl.ca

***Abstract*—Machine Learning (ML) has revolutionized the field of computer software development, enabling data-based predictions and decision-making across several domains. Following modern software development practices, developers use third-party libraries—e.g., Scikit Learn, TensorFlow, and PyTorch—to integrate ML-based functionalities into their applications. Due to the complexity inherent in ML techniques, the models available in the APIs of these tools often require an extensive list of arguments to be set up. Library maintainers overcome this issue by defining default values for most of these arguments so developers can use ML models in their client applications effortlessly. By relying on these default arguments, the clients inadvertently depend on the value defined in these parameters to keep running as expected. We interpret this problem as a semantical breaking change variant, which we named Default Argument Breaking Change (DABC). In this work, we leverage 77 DABCs in Scikit Learn—a well-known ML library—and investigate how 194K client applications are vulnerable to them. Our results show that 72 DABCs (93%) are responsible for exposing 67,747 clients (35%). We also detected that most DABCs (61, 79%) involve APIs used in ML model training and model evaluation stages. Finally, we discuss the importance of managing DABCs in third-party ML libraries and provide insights for developers to mitigate the potential impact of these changes in their applications.**

## I. Introduction

Machine Learning (ML) has changed the landscape of creating computer software in the last decade. Thanks to their ability to make data-based predictions, ML algorithms are integrated into software systems assisting decision-making in several domains like image recognition, cybersecurity, and fraud detection applications [1]–[5]. These algorithms now recommend whom to follow in our social network, filter spam in our e-mail inboxes, and approve our credit score.

Following the practices of modern software development, software developers also depend on third-party components to empower their applications with ML-based functionalities [6]. In this context, the Python ecosystem stands out from others due to the machine learning libraries available [7]–[9], like Scikit Learn,[1] TensorFlow,[2] and PyTorch.[3]

[1]https://scikit-learn.org/
[2]https://www.tensorflow.org/
[3]https://pytorch.org/

These tools provide comprehensive APIs for developers interested in reusing their implemented models. Due to the characteristics of machine learning, the models available in these APIs often require an extensive list of arguments to be set up [9]. Considering the Scikit Learn API as an example, the constructor of `SVC`[4]—a widely used classifier based on SVM—provides 14 arguments to be defined. Many of these arguments are rather specific to the model being reused, e.g., the kernel type to be used in the classifier (argument `kernel`). Fortunately, the maintainers defined a set of default values for such arguments so developers can effortlessly use these classifiers. In practice, developers can create a `SVC` model from scratch without passing any specific value.

On the other hand, relying on these default arguments increases the coupling between the library and its client applications; now clients depend not only on the method syntax provided in the API but also on the values assigned to the arguments to keep running as expected. Consequently, clients' behaviour can be affected by just changing the values of these arguments. For instance, Scikit Learn maintainers changed the Kernel coefficient formula used by default in `SVC`—`gamma` argument—from `"auto"` to `"scale"` between versions 0.21 and 0.22, which can drastically change the model's results.

The lack of backward compatibility between library versions is known as *breaking changes* [10]–[13]. Most studies deal with breaking changes from a syntactical perspective [10], [13]. On the other hand, breaking changes can also encompass semantical modifications, i.e., they may change library behaviour, but clients are not syntactically broken [11], [14]. We interpret this problem as a Semantical Breaking Change variant, which we named Default Argument Breaking Change (DABC). To the best of our knowledge, we did not find any study investigating this type of breaking change.

Default Argument Breaking Changes play a particular role in the context of machine learning tools. First, most components provided by these libraries are hard to inspect, i.e., machine learning engineers may spend significant effort debugging why a given model returned a given result [15].

[4]https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

DABCs add another difficulty layer to this process as a subtle change in a model's argument value may drastically modify its outcome. Second, numerous Jupyter Notebooks—the tool of choice for many data scientists experimenting with their ML models[5]—lack configuration files declaring their module dependencies [16], [17]. A recent study by Pimentel et al. [16] indicates that less than 14% of public notebooks declare some dependency file, e.g., *requirements.txt*. In other words, most notebooks rely on the latest default argument value assigned in their models.

In this paper, we study the occurrence of DABCs in Scikit Learn, one of Python's most used machine learning libraries, and their potential impact on client applications. We manually analyze the changes made to the default values of the arguments in Scikit Learn functions, reported in the official documentation, to leverage the main characteristics of DABCs. We then investigate the likely impact that DABCs have on client applications by analyzing the use of Scikit Learn functions in 194,099 Jupyter Notebooks, publicly available on GitHub. We propose four research questions:

*RQ.1: What are the Most Common DABCs?* In total, we identified 77 DABCs declared in SCIKIT-LEARN. From these, 56 DABCs point to class constructors, e.g., `SVC.__init__()`. This finding suggests that developers should pay particular attention when initializing and configuring their models. The `cv` argument is the most redefined one in 20 DABCs. This argument defines strategies to split data during models' training and validation.

*RQ.2: In which Version the DABCs were Introduced?* We identified DABCs in eight major versions. Version 0.22 stands out with 43 occurrences, followed by 0.20 and 1.1 with 11 each; these three versions concentrate 84% of all DABCs. The changes in these versions are related to SCIKIT-LEARN popular features, including cross-validation training, tree-based model setup, and parallel processing.

*RQ.3: In which Modules the DABCs were Introduced?* The majority of DABCs were reported on *Model Training* and *Model Evaluation* APIs; together, these modules hold 61 out of 77 DABC (79%). This suggests that ML models are in the spotlight regarding DABCs.

*RQ.4: How Clients are Vulnerable to DABCs?* Overall, 67,747 out of 194,099 (35%) client applications are vulnerable to one DABC at least. These calls covered 72 out of 77 DABCs (93%) identified in RQ.1. The presence of DABCs does not correlate with other software metrics, such as LOC, function coupling, and cyclomatic complexity. We also observe that two-thirds of all vulnerable calls are affected by DABCs introduced from version 0.22 onwards. The most vulnerable calls occur during the models' training and evaluation stages concerning the machine learning pipeline modules.

We summarize the contributions of this paper as follows.

- We characterize an unexplored behaviour-breaking change focused on changes performed in API arguments, called DEFAULT ARGUMENT BREAKING CHANGE (DABC).
- We leverage the characteristics of DABCs in Scikit Learn, a well-known machine learning library in Python, and measured how real-world client applications are exposed to them.
- Finally, we discuss strategies that client developers should adopt to avoid being affected by DABCs.

This paper is organized as follows. Section II defines in detail what a DEFAULT ARGUMENT BREAKING CHANGE is, and how it can make client applications vulnerable. Sections III and IV describe the procedure we adopted to collect and identify the DABCs in Scikit Learn and how to map their occurrences in clients. The obtained results are described in Section V. Section VI reports the implications of this work. Section VII reports threats to validity, and Section VIII summarizes the related work. Finally, we conclude this paper in Section IX.

## II. BACKGROUND

### A. Default Arguments in a Nutshell

The Python language supports functions that, once implemented, can be called with only some arguments. For instance, the `round(number, digits)` function[6] has two parameters[7] and returns the `number` rounded to `digits` decimal places. It turns out that `round()` can be called with both one and two arguments; if `digits` is omitted, the function rounds `number` to its nearest integer value. This means that calling `round(3.1415, 2)` returns 3.14, while calling `round(3.1415)` gives 3.0. Such behavior is possible due to Default Arguments,[8] which specify values that functions will use if the caller provides no value to its corresponding argument. In the above example, the function automatically assumed `digits=0` when invoking `round(3.1415)`.

One assigns a Default Argument during function definition by attributing an arbitrary value to the arguments the developer wants to become optional, as shown in Figure 1. In this scenario, executing `sum(10, 20)` will return 30 (`a=10`, `b=20`), `sum(10)` will return 10 (`a=10`, `b=0`), and calling `sum()` will return 0 (`a=0`, `b=0`).

```
def sum(a=0, b=0):
    return a + b
```

Fig. 1. Function definition with Default Argument Values.

Default Arguments are a powerful resource as they mimic method overloading—methods with the same name but different parameters—which is not supported in Python by default [18]. This concept promotes flexibility, readability, and reusability to classes interface; relevant to successful APIs [19], [20].

[5]https://netflixtechblog.com/notebook-innovation-591ee3221233

[6]https://docs.python.org/3/library/functions.html#round

[7]We use arguments and parameters interchangeably in this paper.

[8]https://docs.python.org/3/tutorial/controlflow.html#default-argument-values

## B. *The* Scikit-Learn *Library*

Scikit-Learn is a free and open-source machine learning library for Python, released in 2011 [21]. The library implements well-known supervised and unsupervised machine learning algorithms, such as linear and logistic regressions, support vector machines, decision trees, and k-means clustering. The library also provides techniques to manage, evaluate, and deploy the above-mentioned models.

Such features contributed to its adoption in several industry and research projects. Scikit-Learn is one of the most popular machine learning libraries worldwide, with more than 1 million downloads daily.[9] As of Jan 24th, 2023, the Scikit-Learn repository on GitHub has more than 52,7K stars, almost 30K commits, and was forked above 23,9K times.

Despite its wide adoption, Scikit-Learn's API is constantly changing. For instance, version 1.0 was released in September 2021; the maintainers followed with eight new versions since then. This scenario may challenge the developers whose client applications depend on Scikit-Learn features.

*Default Arguments in* Scikit-Learn*:* Scikit-Learn's API extensively relies on Default Arguments so users can set up the models available in the library with low effort. For example, the `SVC` class provides 14 parameters to be defined through its constructor.[10] These arguments are responsible for configuring several aspects of a SVC model, including the Kernel type used by the model (`kernel`), its Kernel coefficient (`gamma`), random seed values (`random_state`), etc.

Since all arguments have default values assigned, the user can quickly get started without defining any parameter to the model. Figure 2 exemplifies this fact when creating a SVC model. Except for `random_state`, all arguments rely on default values; `kernel` was defined to `"rbf"`, and `gamma` was assigned to `"scale"`.[11]

```python
from sklearn.svm import SVC

clf = SVC(random_state=42)
clf.fit(X_train, y_train)
```

Fig. 2. Default Argument Values in action in Scikit-Learn.

## C. *What is a Default Argument Breaking Change (DABC)?*

Despite the advantages of using Default Arguments, they might bring issues to client applications relying on them. Specifically, library maintainers can update the default values of some parameters to meet new conditions. Changes of this nature do not break clients' code since the function signature (name and arguments) remains the same. Nevertheless, they might introduce incompatibilities as the new value change function's behaviour.

[9]According to https://pypistats.org/

[10]https://scikit-learn.org/1.1/modules/generated/sklearn.svm.SVC.html

[11]Default values available from version 1.1.2, the latest stable one at the time of this work.

For instance, Scikit-Learn maintainers updated the value of `gamma` argument of the `SVC` classifier from `"auto"` to `"scale"` in version 0.22. This update clearly affects models relying on this default value as it changes the math formula used to calculate the *gamma* value. Consequently, the code that creates and trains a SVC model in Figure 2 outputs very different results in versions 0.21 and 0.22.

Default Argument Breaking Change *Example:* We illustrate this maintenance problem by implementing the minimum working example in Listing 3. This example classifies the 20 newsgroup dataset, a popular real-world collection containing 18,000 newsgroup posts grouped into 20 distinct topics. The Stanford Natural Language Processing Group collected this dataset over several years, and it has become a popular alternative for experiments in text applications of machine learning techniques. Currently, it has been used as a benchmark in popular research works [22], [23].

```python
1  from sklearn import datasets
2  from sklearn.model_selection import train_test_split
3  from sklearn.metrics import accuracy_score
4  from sklearn.svm import SVC
5
6  # Load dataset
7  ds = datasets.fetch_20newsgroups_vectorized()
8  X = ds.data[:, 2:]
9  y = ds.target
10
11 # Create training/test data split
12 X_train, X_test,
13 y_train, y_test = train_test_split(X, y,
14                                    test_size=0.3,
15                                    random_state=42,
16                                    stratify=y)
17
18 # Create an instance of SVC Classifier
19 clf = SVC(random_state=42)
20
21 # Fit, predict, and measure model's performance
22 clf.fit(X_train, y_train)
23 y_pred = clf.predict(X_test)
24 print('Acc: %.3f' % accuracy_score(y_test, y_pred))
```

Fig. 3. Illustrative example of DABC in Scikit-Learn.

In this script, we first download the 20 newsgroup dataset and obtain their descriptive and predictive variables (lines 7–9). Next, as with typical ML applications, we split the data into training and test groups in lines 11–16. In line 19, we create a new `SVC` instance; we intentionally did not define any argument except for `random_state` to ensure the same randomness will be present in any execution. Finally, lines 21–24 fit the model with the training data, predict it with test data, and measure its accuracy level.

We execute this script using Scikit-Learn in both versions 0.21 (`gamma="auto"`) and 0.22 (`gamma="scale"`). In version 0.21, we scored 0.05 points for accuracy. The same script reached 0.82 points for accuracy in version 0.22, a difference of 77 points.

This issue encompasses a specific type of breaking change, which we named Default Argument Breaking Change

(DABC). This paper focuses on characterizing DABCs in the SCIKIT-LEARN library and measuring its impact on client applications.

## III. DATA COLLECTION

The SCIKIT-LEARN project adopts a strict contribution guide to enforce better software practices, with source code documentation conventions included.[12] The documentation section provides specific instructions to report changes in "*the default value of a parameter*". According to the guideline, every modification involving the value of an argument should have its docstring's documentation annotated with the $versionchanged$ directive. Also, the old and new default values should be reported together with the version the change became effective. Figure 4 presents an example of this description in SCIKIT-LEARN project. In this case, the parameter `gamma`—which belongs to the constructor of `SVC` class—had its value changed from `"auto"` to `"scale"`, valid from version 0.22 onward. We relied on this guideline to collect the DEFAULT ARGUMENT BREAKING CHANGES studied in this work.

```
class SVC(BaseSVC):
    """C-Support Vector Classification.
    ...
    Parameters
    ----------
    ...
    gamma : {'scale', 'auto'} or float,
        default='scale'
        Kernel coefficient for 'rbf', 'poly'
        and 'sigmoid'.
        - if ``gamma='scale'`` (default) ...
        - if 'auto', ...
        - if float, ...
        .. versionchanged:: 0.22
        The default value of ``gamma`` changed
        from 'auto' to 'scale'.
    """
```

Fig. 4. Example of a default argument changed in SCIKIT-LEARN documentation.

*Mining Changes on Functions Arguments:* On October 11th, 2022, we cloned the SCIKIT-LEARN project from GitHub,[13] and manually checked out the commit of version 1.1.2; the latest public release available. Next, we selected all Python files in the $sklearn$ directory, as the library's source files are located in this directory. We then opened each Python file and filtered out the lines containing the $versionchanged$ directive; we did this using the regular expression "$\backslash.\backslash.versionchanged\ ::\ .+$". This procedure initially returned 179 occurrences.

*Selected Attributes:* For each occurrence, we collected a list of five attributes used to answer the research questions proposed in this paper. We listed these attributes below:

- *dabc_msg*: This attribute contains the message used to justify each occurrence. We collected this information manually for each occurrence during the categorization we did to answer RQ.1.
- *version*: This attribute keeps the version assigned to each occurrence as collected by the regex matching procedure. We used this information to answer RQ.2.
- *path*: Collected during the repository analysis, this attribute holds the relative path from the $scikit$ directory to the file containing the DABC. We used this attribute to leverage the modules to answer RQ.3.
- *fqn*: This attribute holds the full qualified name of the function where the occurrence was found, i.e., class name followed by the method signature. Similar to *dabc_msg*, we manually leveraged this information to answer RQ.4.
- *dabc_url*: We generate the GitHub URL referring to the exact point in the source code where the DABC was declared. We refer to this URL whenever we need more context about the DABC to answer the research questions.

## IV. RESEARCH QUESTIONS

In this section, we describe the methodology steps to answer the four research questions proposed in this study.

### A. RQ.1: What are the Most Common DABCs?

This RQ aims to identify the DEFAULT ARGUMENT BREAKING CHANGES from the changes in arguments retrieved in Section III. Due to the number of occurrences retrieved, three authors manually inspected and discussed together all 179 occurrences, selecting the ones they considered valid. For this, they perform a two-step filtering approach. First, the three authors read the description below each $versionchanged$ to select the occurrences dealing only with arguments, i.e., they filtered out unrelated changes. The authors discarded 46 occurrences in this step reporting other changes, such as return values, object attributes, function refactorings, etc. Then, they analyzed each of the remaining 133 occurrences in detail to verify which of the changes can be characterized as DEFAULT ARGUMENT BREAKING CHANGE.

They discarded another 56 occurrences performing other changes in argument values, such as type changes (e.g., the `min_samples_split` parameter started accepting float values in version 0.18),[14] and in other values that can be passed to the parameter besides the default one (e.g., the `metric` argument no longer accepts a specific value in version 0.19).[15] After removing these cases, we remained with 77 DEFAULT ARGUMENT BREAKING CHANGES.

### B. RQ.2: In which Version the DABCs were Introduced?

This RQ aims to discover in which part of the release cycle these changes are introduced. To do so, we analyzed the distribution of DABCs among the SCIKIT-LEARN's release

[12]The contribution guide can be accessed at https://scikit-learn.org/stable/developers/contributing.html.

[13]Repository available at https://github.com/scikit-learn/scikit-learn/.

[14]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[15]https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestCentroid.html

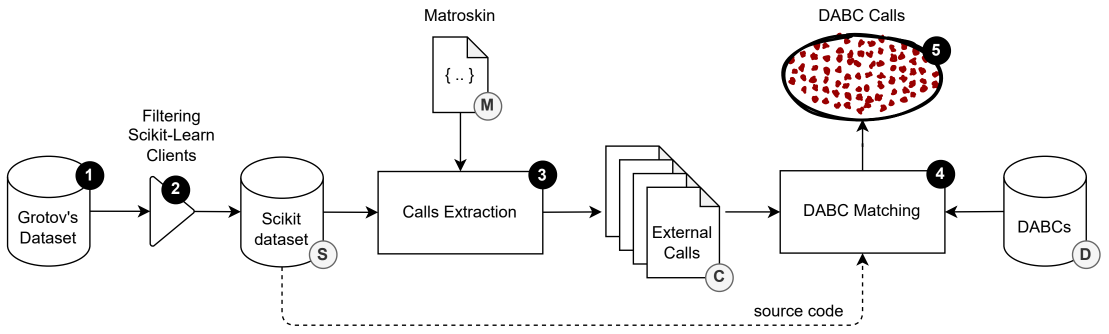

Fig. 5. The data-collection pipeline adopted to answer RQ.4. Steps annotated in black (e.g., ❶) are described in detail, while steps in gray (e.g., Ⓢ) are mentioned during Section IV-D.

versions. As described in SCIKIT-LEARN documentation,[16] the project maintainers nominate its releases based on PEP101. This specification describes the library's versions using the $X.Y.Z$ triplet. Minor versions are tracked by the $.Z$ suffix and should include bug fixes and some relevant documentation changes only, i.e., they should not contain a behaviour change besides a bug fix. On the other hand, major versions indicating new releases are annotated with the $X.Y$ prefix; these versions can contain new features and significant maintenance that modify the library behaviour.

To answer this question, we leveraged all versions containing the $X.Y.Z$ syntax released before version 1.1.2 from the git tags available in the SCIKIT-LEARN git repository. We then collected the date of the commits responsible for creating these tags and annotated them as $major$ or $minor$ according to the SCIKIT-LEARN convention described above. We identified 56 versions released in a 12-year period, where 26 are major and 30 are minor versions.

As the last step, we combined the release information with the $version$ field of each DABC identified in Section III. For example, the DABC present in Figure 4 matches with version 0.22, so we considered the maintainers introduced this DABC in a major release on 2019-12-02.

### C. RQ.3: In which Modules the DABCs were Introduced?

This RQ aims to investigate in which modules of SCIKIT-LEARN DABCs are located. We named nine modules that comprise the phases of Machine Learning [24]. For each commit, we manually inspected the changed source code files using *path* and *dabc_url* (described in Section III). Then, these changes were labelled into one of the following modules:

- *Dataset* contains utilities to handle large datasets (e.g., functions to download and load data) and traditional datasets (e.g., load and get data from a public repository).
- *Data preprocessing* comprises utility functions and transformation techniques to apply on raw features for standardizing datasets.
- *Data Decomposition* consists of functions that implement dimensionality reduction or feature selection techniques to apply to the dataset.

[16]https://scikit-learn.org/stable/developers/maintainer.html#releasing

- *Data Analysis* contains the implementation of statistical techniques to support the understanding of data process.
- *Feature Processing* includes techniques to transform arbitrary data into usable data supported by Machine Learning algorithms.
- *Model Training* consists of algorithms' implementations for unsupervised and supervised learning methods.
- *Model Evaluation* contains several techniques to measure the estimator performance and evaluate the model predictions' quality.
- *Utils* comprises several utilities, such as estimate class weights for unbalanced datasets.
- *Pipeline* consists of utilities to build a composite estimator. We followed SCIKIT-LEARN's documentation to label this module.

Finally, we assigned all changes that are not directly related to ML tasks to *Others*. In this study, we observed that these changes are related to exception handlers.

### D. RQ.4: How Clients are Vulnerable to DABCs?

In this question, we investigate the potential impact that DEFAULT ARGUMENT BREAKING CHANGES have on client applications that use the SCIKIT-LEARN library. For this, we implemented a data-collection pipeline that obtains a list of real-world client applications that use SCIKIT-LEARN library, extracts the method calls performed in these clients, and selects the calls vulnerable to DABCs in SCIKIT-LEARN. Figure 5 depicts this procedure; more details about each step are described in the remainder of this section.

❶ *Clients Dataset:* We studied some datasets containing SCIKIT-LEARN clients and data collection strategies to identify the best fit for our needs [8], [9], [16], [17], [25]. We consider the dataset's size, available documentation, and the effort to replicate and adapt it to our context. We selected Grotov et al.'s dataset [8] due to two reasons: (a) it comes in a structured format that can be queried using SQL; and (b) the authors also provide a tool that analyzes the source code of Jupyter Notebook and Python scripts, called Matroskin. This dataset contains 847,881 preprocessed Jupyter Notebooks written in Python, extracted from GitHub between September and October 2020.

TABLE I
THE TOP 10 MOST REFERRED ARGUMENTS IN COLLECTED DABCS.

| **Changed Argument** | **DABCs** | |
|---|---|---|
| | # | % |
| *cv* | 20 | 26.0 |
| *n_jobs* | 8 | 10.4 |
| *max_features* | 6 | 7.8 |
| *gamma* | 5 | 6.5 |
| *n_estimators* | 5 | 6.5 |
| *n_splits* | 4 | 5.2 |
| *init* | 4 | 5.2 |
| *multi_class* | 3 | 3.9 |
| *return_train_score* | 3 | 3.9 |
| *algorithm* | 2 | 2.6 |
| **Total** | **60** | **77.9** |

TABLE II
DABCS FOR EACH VERSION.

| **Version** | **DABCs** | |
|---|---|---|
| | # | % |
| 0.19 | 3 | 3.9 |
| 0.20 | 11 | 14.3 |
| 0.21 | 3 | 3.9 |
| 0.22 | 43 | 55.8 |
| 0.23 | 3 | 3.9 |
| 0.24 | 2 | 2.6 |
| 1.0 | 1 | 1.3 |
| 1.1 | 11 | 14.3 |
| **Total** | **77** | **100** |

TABLE III
DABCS FOR EACH MODULE.

| Modules | **DABCs** | |
|---|---|---|
| | # | % |
| Dataset | 1 | 1.3 |
| Data preprocessing | 2 | 2.6 |
| Data Decomposition | 5 | 6.5 |
| Data Analysis | 2 | 2.6 |
| Feature Processing | 1 | 1.3 |
| Model Training | **42** | **54.5** |
| Model Evaluation | **19** | **24.7** |
| Utils | 1 | 1.3 |
| Pipeline | 3 | 3.9 |
| Others | 1 | 1.3 |
| **Total** | **77** | **100** |

❷ *Filtering* SCIKIT-LEARN *Clients:* We selected all notebooks relying on the SCIKIT-LEARN library from the initially obtained dataset. Specifically, the preprocessed database contains the list of imported modules for each client notebook. We then queried the database for all notebooks containing the *"sklearn"* string—the name of SCIKIT-LEARN module—in their import list. In total, 194,099 notebooks met this criterion (Ⓢ).

❸ *Calls Extraction:* In this step, we extracted all existing method calls performed in each notebook. However—unlike the list of imports—this information is not available by default in the dataset. Instead, the authors provide only the number of method calls that belong to external sources, i.e., imported and third-party modules. To obtain the actual method calls, we instrumented *Matroskin* (Ⓜ) to extract all external calls during its syntactical analysis and re-executed it on the notebooks of *Scikit dataset* (Ⓢ). We ended up with 17,436,073 *external calls* (Ⓒ) extracted from the 194,099 clients.

❹ *DABC Matching:* In the last step of this pipeline, we selected all method calls vulnerable to DABCs. Traditionally this could be achieved by tracking down the declaration of the called method, retrieving its arguments, and checking if the default argument is assigned in the call. However, it is not straightforward to infer this information since Python is a dynamically-typed language [7], [8], [26]. Due to this reason, we worked on a static matching heuristic to detect calls to methods identified as DABC.

The heuristic works as follows. We first parse[17] the definition of all 77 *DABCs* (Ⓓ) identified in Section IV-A and extract their class name (if any), method name, and list of defined arguments. Next, for each *external call* (Ⓒ), we parse and extract its method name and list of argument values; note that we can not directly obtain class names as Python is dynamically-typed. Then we match DABCs and calls based on two conditions: (a) the DABC class name—if it exists—is in the same file where the call was retrieved; and (b) the method name in both DABC and in the call are the same. For each successful match, we pair the call's argument values to the DABC's defined arguments by assigning all positional arguments in sequence and assigning all keywords arguments based on the key provided. Lastly, we check if the DABC's default argument is assigned. The call is considered vulnerable if **no value is assigned to the DABC argument**. In practice, the call did not provide a value for it, so it relies on the DABC's default argument value.

❺ *DABCs Calls Dataset:* The DABC matching procedure identified 317,648 calls vulnerable to DABCs in 67,747 client applications. We test the effectiveness of this heuristic by manually analyzing a randomly selected sample of 384 calls, equally divided among three authors.[18] They verified whether both method's call and DABC point to the same function and if the call is vulnerable. To ensure the authors followed a similar verification pattern, they analyzed 38 calls together (10% of the sample size). In their evaluations, the authors identified 366 calls (95.3%, ±5%) as valid ones. From the remaining 18, the heuristic mostly fail at detecting arguments outside the function call; e.g., arguments that were passed inside a Python dictionary, instead. Such issue is beyond identification in static analysis, hence out of scope in our heuristic.

## V. RESULTS

### A. RQ.1: What are the Most Common DABCs?

The 77 DABCs are spread over 61 distinct methods. From these, 19 methods are declared outside of any class—e.g., `cross_validate()`, `k_means()`, etc—and account for 21 DABCs. All 42 remaining methods declared in classes are constructors; they are responsible for 56 DABCs. This finding emphasizes the central role that method constructors play when configuring SCIKIT-LEARN models.

Individually, the methods `GridSearchCV.__init__()` and `RandomizedSearchCV.__init__()` lead the rank of DABCs containing three occurrences, each. Both `GridSearchCV` and `RandomizedSearchCV` classes implement strategies for optimizing ML models. Moreover, both methods are vulnerable to changes performed in the same

[17]We used the Python `gast` module at (https://pypi.org/project/gast/).

[18]The sample size was determined considering 95% confidence level and 5% confidence interval.

arguments: (i) `cv` defines the cross-validation strategy; (ii) `n_jobs` determines the number of jobs to run in parallel; and (iii) `return_train_score` determines if the method returns the computed training scores. Twelve other methods have two DABCs each. The remaining 47 methods show up with one DABC only.

We also analyzed the distribution of DABCs among the changed arguments. In total, 24 arguments had their default value changed by at least one DABC. Table I lists the top 10 most modified ones. The `cv` argument stands out with 20 occurrences (26%). This argument defines the cross-validation strategy to split the data during model training and validation. It is widely used in SCIKIT-LEARN, as most supervised models rely on data-splitting techniques when they are trained. Similarly, `n_jobs` (8 occurrences, 10.4%) is also adopted in different scenarios. The remaining arguments belong to specific classes and models. For instance, `max_features` (6 occurrences, 7.8%) and `n_estimators` (5 occurrences, 6.5%) configure tree-based models.

> The 77 DABCs are spread across 61 methods. 56 DABCs occurs in class constructors. `cv`—responsible for defining cross-validation strategies in ML models—is the most modified argument present in 20 DABCs.

## B. RQ.2: In which Version the DABCs were Introduced?

Table II presents the distribution of DABCs among each SCIKIT-LEARN's release. They are distributed in eight versions; the first ones appeared on version 0.19, released in November 2017. Since then, we have identified DABCs in all major releases, i.e., no DABC was reported in minor versions.

Three versions concentrate 65 occurrences, representing 84.4% of all DABCs reported in this study. Specifically, version 0.22 stands out with 43 modifications in default arguments (55.8%); both versions 0.20 and 1.1 appear next with 11 (14.3%) DABCs. The five remaining versions gather 12 DABCs in total.

We inspected in detail versions 0.22, 0.20, and 1.1 to better understand the reason for such disparity. We find that the changes in both versions deal with SCIKIT-LEARN's popular features. For instance, 19 out of 43 occurrences in 0.22 deal with `cv`. Other five occurrences modify `n_estimators` argument. Version 0.20 presents a similar characteristic, as eight occurrences point to the `n_jobs` argument. Differently, DABCs are regularly distributed in version 1.1 with four distinct changes varying between two and three occurrences.

> We identified DABCs in all major versions since 0.19; no DABC was found in minor ones. Version 0.22 alone concentrates 43 out of 77 DABCs (56%), followed by 0.20 and 1.1, with 11 occurrences (14%) each; together, these three versions gather 84% of all DABCs.

## C. RQ.3: In which Modules the DABCs were Introduced?

Table III describes the classified modules. We observe that both *Model Training* and *Model Evaluation* stand out from the other modules, with 42 and 19 DABCs, respectively; together, they represent more than 79% of all identified DABCs. Note that the classes implemented in both modules deal directly with machine learning algorithms, hence they are at the core of most machine learning applications. The remaining modules contain five changes or fewer.

> *Model Training* and *Model Evaluation* are the ones with most DABCs, with 42 and 19 occurrences, respectively.

## D. RQ.4: How Clients are Vulnerable to DABCs?

TABLE IV
MOST FREQUENT DABCS CALLS IN CLIENT APPLICATIONS.

| **Class.Method(Default Argument)** | **Calls** # | **Calls** % |
|---|---|---|
| *LogisticRegression.__init__(multi_class)* | 38,323 | 12.1 |
| *LogisticRegression.__init__(solver)* | 31,290 | 9.9 |
| *RandomForestClassifier.__init__(max_features)* | 30,874 | 9.7 |
| *SVC.__init__(decision_funciton_shape)* | 29,904 | 9.4 |
| *GridSearchCV.__init__(return_train_score)* | 24,930 | 7.8 |
| *SVC.__init__(gamma)* | 22,258 | 7.0 |
| *KMeans.__init__(algorithm)* | 22,063 | 6.9 |
| *r2_score(multioutput)* | 20,805 | 6.5 |
| *GridSearchCV.__init__(n_jobs)* | 16,396 | 5.2 |
| *RandomForestRegressor.__init__(max_features)* | 14,970 | 4.7 |
| **Total** | **251,813** | **79.2** |

*1) What are the most frequent DABCs?:* We detected vulnerable calls for 72 out of the 77 DABCs identified previously (93%); Table IV lists the most frequent DABCs identified in client applications. The top 10 gathered 251,813 of the 317,648 vulnerable calls (79.2%), suggesting a heavy-tail distribution. While the 10th most frequent DABC contains 14,970 vulnerable calls, the median value is 365.

We observe that nine vulnerable calls refer to class constructors. This highlights a common practice in SCIKIT-LEARN where most configuration arguments are passed when creating the model instance. The first two calls in the table refer to the same method (`LogisticRegression.__init__()`), but with different argument values (`multi_class` and `solver`). Same behavior applies for `SVC.__init__()` (`decision_function_shape` and `gamma`) and `GridSearchCV.__init__()` (`return_train_score` and `n_jobs`) methods. We also observe the same default argument used in two distinct classes: `max_features` is used when calling `RandomForestClassifier.__init__()` and `RandomForestRegressor.__init__()` methods.

> We detected vulnerable calls in clients for 93% of DABCs identified previously. The top 10 most frequent DABCs accounted for almost 80% of vulnerable calls, with nine

of them referring to SCIKIT-LEARN models initialization.

*2) How many clients are vulnerable to DABCs?:* In total, 67,747 out of 194,099 clients are vulnerable to at least one DABC (35%). Although each had to deal with 4.69 DABCs on average, we identified nine clients with more than 100 vulnerable calls. The 99%, 95%, and 50% client percentiles respond to 30, 16, and 2 vulnerable calls, respectively.

TABLE V
SPEARMAN CORRELATION BETWEEN THE STRUCTURAL METRICS COLLECTED BY GROTOV ET AL. [8] AND THE NUMBER OF CALLS VULNERABLE TO DABCS IN CLIENT APPLICATIONS. THE BULLET SHAPE QUANTIFIES THE CORRELATION LEVEL: NEGLIGIBLE (●), LOW (●●), AND MODERATE (● ● ●).

| | Metric | Correlation | |
|---|---|---|---|
| | | Coeff. | Level |
| CODE WRITING | **SLOC** | **0.38** | ●● |
| | Blank LOC | 0.28 | ● |
| | Extended comments LOC | 0.24 | ● |
| | Comments LOC | 0.19 | ● |
| FUNCTION USAGE | **API functions (count)** | **0.50** | ● ● ● |
| | **API functions (unique)** | **0.38** | ●● |
| | Other functions (count) | 0.32 | ●● |
| | Built-in functions (count) | 0.29 | ● |
| | Built-in functions (unique) | 0.19 | ● |
| | User-defined functions (count) | 0.18 | ● |
| | User-defined functions (unique) | 0.15 | ● |
| COMPLEXITY | Cell coupling | 0.41 | ●● |
| | Function coupling | 0.14 | ● |
| | NPAVG | 0.13 | ● |
| | Cyclomatic complexity | 0.09 | ● |

We triangulated the number of vulnerable calls with the 15 structural metrics collected by Grotov et al. [8] to verify how traditional software metrics relate to DABCs. This is a first step towards understanding how software quality influences the emergence of DABCs. For this, we executed the Spearman correlation test between the number of calls and each metric separately. We opted for Spearman due to its robustness in interpreting non-normalized distributions [27]. Following the guidelines proposed in other works [6], [26], [28], we interpret its coefficient according to the following: $0.00 \leq negligible < 0.30 \leq low < 0.50 \leq moderate < 0.70 \leq high < 0.90 \leq veryhigh < 1.00$.

Table V presents the correlation results; we marked in bold the top three metrics with higher correlation coefficients. Overall, we did not identify any high correlation between the structural metrics and the number of vulnerable calls in client applications. On the contrary, the correlation levels of most metrics are either $low$ (four) or $negligible$ (ten). Only *API functions (count)* presented a $moderate$ correlation with the number of vulnerable calls (0.50); the correlation with *API functions (unique)* is lower, though (0.38, $low$ level). *Complexity*-based are independent to the number of DABCs calls: *Cyclomatic complexity*, *Function coupling*, and *NPAVG* scored the lowest correlation coefficients with 0.09, 0.13, and 0.14, respectively. These findings suggest that the presence of DABCs is relatively independent of the codebase size, functions usage, and complexity.

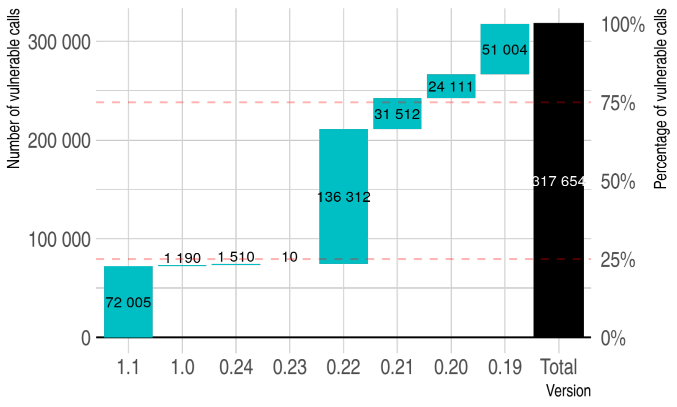

Fig. 6. Number of vulnerable calls in each version.

35% of client applications are vulnerable to DABCs. Besides *API functions (count)*, we did not find any substantial correlation between source code metrics and DABCs calls.

*3) Which versions make clients more vulnerable?:* Figure 6 depicts the vulnerable calls in each SCIKIT-LEARN version. Versions 0.22, 1.1, and 0.19 stand out with 136,312 (42.9%), 72,005 (22.7%), and 51,004 (16.1%) vulnerable calls, respectively; altogether, these versions concentrate 81.6% of the vulnerable calls. By contrast, versions 0.23, 1.0, and 0.24 had the most negligible impact on clients with 10 (<0.01%), 1,190 (0.37%), and 1,510 (0.48%) calls.

We observe that 23% of vulnerable calls happen in more recent versions, i.e., version 1.0 onwards. The proportion goes to 66% when we extend this analysis to version 0.22. In other words, two-thirds of all vulnerable calls are due to DABCs reported from versions 0.22 onwards.

Versions 0.22, 1.1, and 0.19 had the highest impact on clients, accounting for 81% of vulnerable calls; versions 0.23, 1.0, and 0.24 had the least. Two-thirds of all vulnerable calls are due to DABCs reported from version 0.22 onwards.

*4) Which ML modules are more vulnerable in clients?:* Table VI presents the number of vulnerable calls located in each ML module leveraged in Section IV-C. *Model Training* and *Model Evaluation* clearly stand out with 248,014 and 64,927 calls each; both modules condense 98.5% of all vulnerable calls. Such higher concentration indicates that most DABCs show up when dealing with the machine learning models. On the other hand, no other module gathers more than 1% of vulnerable calls; *Pipeline* is the highest remaining one with 0.59%. We also verified the modules that are vulnerable together, i.e., in the same client. In this perspective, 16,233 (24.0%) clients are simultaneously vulnerable in two modules,

TABLE VI
NUMBER OF VULNERABLE CALLS FOR EACH ML MODULE.

| **Module** | **Calls** | |
|---|---|---|
| | **#** | **%** |
| Data Analysis | 32 | 0.01 |
| Data Decomposition | 1,067 | 0.33 |
| Feature Processing | 103 | 0.03 |
| Model Evaluation | 64,927 | 20.43 |
| Model Training | 248,014 | 78.07 |
| Utils | 173 | 0.05 |
| Dataset | 1,070 | 0.33 |
| Pipeline | 1,876 | 0.59 |
| Preprocessing | 392 | 0.12 |

1,204 (1.8%) in three, and 172 (0.2%) in four modules.

> DABCs located in *Model Training* and *Model Evaluation* are responsible for most vulnerable calls (98.5%). From the clients' perspective, 74% are vulnerable in one module, only.

## VI. DISCUSSION

We understand that the findings reported in this paper unfold implications for researchers, library maintainers, and library users. We discuss them in the following subsections.

### A. Implications for Researchers

*DABCs are a reality.* As presented in Section V-D, we found that more than one-third of client applications are exposed to DABCs. These clients have method calls covering 93% of all DABCs leveraged in Section V-A. These results show that DABCs do exist, yet we did not find studies investigating this particular type of breaking change before. In this context, we believe our work is a first step towards a better understanding of DABCs in the ML ecosystem and beyond.

*We lack studies that investigate breaking changes in dynamically typed languages.* Public APIs frequently rely on function overloading to promote flexibility, readability, and reusability [19], [20]. Although dynamically-typed languages do not support this technique natively,[19] we can still make use of it—calling the same function with a variadic number of parameters—by using default argument values. Consequently, we can say that languages like Python, JavaScript, and PHP are especially exposed to DABCs. Therefore, we understand that dynamically-typed languages should receive more attention in breaking changes studies [14], [29], as more issues specific to these languages may arise.

*We need to understand why library maintainers rely on default argument values.* In this study, we describe *what* DEFAULT ARGUMENT BREAKING CHANGES are, *when* and *where* they are introduced, as well as *who* is vulnerable to them. However, understanding *why* maintainers introduce such modifications remains open. We claim that investigating this aspect is paramount as it could reveal typical scenarios where these values are modified. Such findings can contribute to design recommendations and good practices for using default argument values.

[19]For more details see https://softwareengineering.stackexchange.com/questions/425422/do-all-dynamically-typed-languages-not-support-function-overloading

### B. Implications for Library Maintainers

*DABCs present a ripple effect.* As the results in Section V-A show, DABCs are concentrated on a few arguments. For example, the `cv` argument—responsible for defining cross-validation strategies during models' training and validation stages—is the pivot of 26% of the DABCs we detected. This finding suggests that, as with traditional breaking changes, DABCs may lead to a ripple effect among the API interface—updating the default value of one argument might affect the behaviour of other functions in the API—ultimately impacting a much larger number of client applications [11], [30]. We argue that library maintainers should be aware of this issue when updating the default values of their APIs.

*Maintainers should follow semantic versioning strategies.* SCIKIT-LEARN maintainers look aware of the risks involved in changing default argument values. As we can observe in Section V-B, all DABCs were reported on the library's major versions, hence complying with semantic versioning. This finding contradicts other results reported in the literature [12], [13]. Although adopting this approach does not suppress the occurrence of DABCs on client applications—for example, clients might not rely on versioning strategies at all [16]—adopting strict versioning guidelines is still an effective way to keep the compatibility of libraries APIs [14].

### C. Implications for Library Clients

*Clients should work with package managers.* The results reported in Section V-D show that client applications are vulnerable to DABCs; 67,747 out of 194,099 SCIKIT-LEARN clients (35%) are exposed to at least one DABC Moreover, clients turned out to be vulnerable to multiple API versions. We reinforce a recommendation already mentioned in other works: client developers must adopt minimum versioning strategies when maintaining their dependencies, such as using package managers [12], [14], [16].

*Clients should choose carefully when to rely on default values.* As stated previously, using default argument values reduces the effort when using a given API method. Section II shows that, although the `SVC` constructor receives 14 parameters, it is possible to create a new model without any specific as all arguments have default values assigned. Under the hood, using default arguments introduce data dependencies in client applications as the values provided to their functions are defined by third parties, of which they have no control [11], [14]. Considering the context of SCIKIT-LEARN library, our work shows these dependencies expose clients in crucial machine learning stages, such as *Model Training* and *Model Evaluation* (see sections V-C and V-D). Therefore, we argue that client developers should be diligent when relying on default values.

Specifically, we suggest temporarily relying on these values; for example, during machine learning model development and experimentation.

## VII. Threats to Validity

*Selected Library:* We scoped this work on analyzing DABCs for one library, specifically. Despite SCIKIT-LEARN being one of the most adopted machine learning libraries, we acknowledge it is not possible to generalize our findings to other third-party components.

*DABCs Identification Process:* We rely on the documentation provided by SCIKIT-LEARN maintainers to leverage the DABCs investigated in this study. Naturally, this strategy may pose some threats in detecting API changes, as we are restricted to the changes properly documented in the library API. In our favour, SCIKIT-LEARN maintainers follow strict guidelines for reporting API changes, e.g., they provide clear instructions about how to document modifications in the library's API, including default value changes.[20] Moreover, other researchers also relied on API documentation to obtain breaking change candidates [9].

*Clients Dataset:* We rely on the dataset provided by Grotov et al. [8] to investigate how clients are exposed to DABCs. Although we could select other datasets to perform this analysis [9], [16], [17], [25], we take into account the documentation available to download and configure the dataset locally and the publicly available tool that—after proper adaptations—helped us in extracting the clients' function calls. Yet, we understand it is important to expand this analysis to other artifacts besides Jupyter Notebooks, such as Python script files [16], [25].

*Function Call Heuristic:* We implemented a heuristic to identify DABCs calls in client applications. Ideally, we could overcome this threat by executing the source code of each client and performing a dynamic analysis over the functions called. Even though, we opted for an static-based analysis since previous works reported great difficulty in performing this task [16], [17]. To ensure the reliability of our heuristic, we manually analyzed a sample of 384 calls and find out that 95.3% ($\pm$5%) were correctly classified by the heuristic.

## VIII. Related Work

### A. Library Updates and Breaking Changes

Various studies proposed techniques to detect and understand breaking changes in libraries and frameworks [10], [14], [31]–[34]. Mezzetti et al. [14] conducted a study on breaking changes in the *npm* repository and introduced a technique called *type regression testing* to automatically detect whether a library update affects the types provided by its public interface. According to the authors, at least 5% of all packages experienced a breaking change in a patch or minor update, with most of these changes attributed to modifications in the public package API. Mostafa et al. [32] describe a large-scale regression testing performed over 68 adjacent version pairs from 15 popular Java libraries to comprehend APIs' behavioural changes over time. For this, the authors executed each version pair and compared the output produced by them, i.e., whether the updated code changed library behaviour. Their result reveals that behavioural backward-incompatibilities are common in Java libraries and are the root of many backward-incompatibility issues. Our study investigated an unexplored kind of behaviour-breaking change (DABCs) in Machine Learning libraries.

[20]https://scikit-learn.org/dev/developers/contributing.html#change-the-default-value-of-a-parameter

### B. Machine Learning APIs Smells

The literature shows that Machine Learning (ML) systems are also prone to traditional software engineering issues [3], [5], [35], [36]. For example, Tang et al. [37] study refactoring and technical debt issues in ML systems. OBrien et al. [2] also investigate technical debts in ML applications. In the context of code smells, Zhang et al. [38] identify 22 machine-learning-specific code smells, while Gesi et al. [36] investigate code smell in Deep Learning software systems. Regarding API maintenance, Haryono et al. [7] studied a list of 112 deprecated APIs from three popular Python ML libraries to better understand how they can be migrated. The authors identified three dimensions involving deprecated API migrations: update operation, API mapping, and context dependency. Zhang et al. [9] investigated changes performed on the API documentation of multiple TensorFlow versions to analyze their evolution. Then, they classified these changes into ten categories according to the reason behind the modifications; the most common ones are efficiency and compatibility. Differently, we studied a behaviour-breaking change specific for changes performed in APIs default arguments, i.e., DABCs.

## IX. Conclusion

In this work, we investigate an unexplored type of behaviour breaking change, which we named *Default Argument Breaking Change (DABC)*. Specifically, we identified the DABCs in SCIKIT-LEARN—a well-known Machine Learning library—and analyzed how client applications are vulnerable to them. Overall, we analyzed 77 DABCs among eight major versions of SCIKIT-LEARN; 93% of them were detected in client applications We also discuss the implications of our findings for researchers, library maintainers, and clients.

We intend to extend this work in the following directions: (a) reproduce this study with other machine learning libraries, such as TensorFlow, NumPy, etc; (b) investigate *why* library maintainers introduce and modify default values in libraries' APIs; and (c) study the qualitative impact that such modifications have on clients.

*Replication Package:* Data and scripts are publicly available at Zenodo: https://doi.org/10.5281/zenodo.7868228.

## Acknowledgment

This research is supported by the National Council for Scientific and Technological Development (CNPq).

## References

[1] F. Provost and T. Fawcett, *Data Science for Business: What You Need to Know about Data Mining and Data-Analytic Thinking*, 1st ed. Beijing Köln: O'Reilly, 2013.

[2] D. OBrien, S. Biswas, S. M. Imtiaz, R. Abdalkareem, E. Shihab, and H. Rajan, "23 Shades of Self-Admitted Technical Debt: An Empirical Study on Machine Learning Software," in *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022, p. 13.

[3] H. Washizaki, F. Khomh, Y.-G. Guéhéneuc, H. Takeuchi, N. Natori, T. Doi, and S. Okuda, "Software-Engineering Design Patterns for Machine Learning Applications," *Computer*, vol. 55, no. 3, pp. 30–39, 2022.

[4] F. Ferreira, L. L. Silva, and M. T. Valente, "Software engineering meets deep learning: A mapping study," in *36th Annual ACM Symposium on Applied Computing (SAC)*. ACM, 2021, pp. 1542–1549.

[5] 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 *41st ACM/IEEE International Conference on Software Engineering (ICSE)*, 2019.

[6] J. E. Montandon, L. Lourdes Silva, and M. T. Valente, "Identifying Experts in Software Libraries and Frameworks Among GitHub Users," in *16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 276–287.

[7] S. A. Haryono, F. Thung, D. Lo, J. Lawall, and L. Jiang, "Characterization and Automatic Updates of Deprecated Machine-Learning API Usages," in *International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2021, pp. 137–147.

[8] K. Grotov, S. Titov, V. Sotnikov, Y. Golubev, and T. Bryksin, "A Large-Scale Comparison of Python Code in Jupyter Notebooks and Scripts," in *19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 353–364.

[9] Z. Zhang, Y. Yang, X. Xia, D. Lo, X. Ren, and J. Grundy, "Unveiling the Mystery of API Evolution in Deep Learning Frameworks: A Case Study of Tensorflow 2," in *43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 238–247.

[10] A. Brito, M. T. Valente, L. Xavier, and A. Hora, "You broke my code: Understanding the motivations for breaking changes in APIs," *Empirical Software Engineering*, vol. 25, pp. 1458–1492, 2020.

[11] A. Ponomarenko and V. Rubanov, "Backward compatibility of software interfaces: Steps towards automatic verification," *Programming and Computer Software*, vol. 38, pp. 257–267, 2012.

[12] D. Venturini, F. R. Cogo, I. Polato, M. A. Gerosa, and I. S. Wiese, "I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages," *ACM Transactions on Software Engineering and Methodology*, 2023.

[13] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, "Breaking bad? Semantic versioning and impact of breaking changes in Maven Central," *Empirical Software Engineering*, vol. 27, pp. 1–42, 2022.

[14] G. Mezzetti, A. Møller, and M. T. Torp, "Type Regression Testing to Detect Breaking Changes in Node.js Libraries," in *32nd European Conference on Object-Oriented Programming (ECOOP)*, 2018, pp. 1–24.

[15] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why Should I Trust You?": Explaining the Predictions of Any Classifier," in *22nd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2016, pp. 1135–1144.

[16] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks," in *16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 507–517.

[17] J. Wang, T.-Y. KUO, L. Li, and A. Zeller, "Assessing and Restoring Reproducibility of Jupyter Notebooks," in *35th International Conference on Automated Software Engineering (ASE)*, 2020, pp. 138–149.

[18] D. Phillips, *Python 3 Object-Oriented Programming: Build robust and maintainable software with object-oriented design patterns in Python 3.8*, 3rd ed. Packt Publishing, 2018.

[19] J. Bloch, *Effective Java*, 3rd ed. Addison-Wesley Professional, 2017.

[20] 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.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011.

[22] D. Wang, P. Cui, and W. Zhu, "Structural Deep Network Embedding," in *22nd International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 1225–1234.

[23] F. Bianchi, S. Terragni, and D. Hovy, "Pre-training is a Hot Topic: Contextualized Document Embeddings Improve Topic Coherence," in *59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (IJCNLP)*, 2021, pp. 759–766.

[24] P. Domingos, "A few useful things to know about machine learning," *Commun. ACM*, vol. 55, no. 10, p. 78–87, oct 2012.

[25] L. Quaranta, F. Calefato, and F. Lanubile, "KGTorrent: A Dataset of Python Jupyter Notebooks from Kaggle," in *18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 550–554.

[26] J. Tan, D. Feitosa, and P. Avgeriou, "Investigating the Relationship between Co-occurring Technical Debt in Python," in *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 487–494.

[27] D. E. Hinkle, W. Wiersma, and S. G. Jurs, *Applied Statistics for the Behavioral Sciences*, 5th ed. Belmont, CA: Wadsworth Cengage Learning, 2003.

[28] H. Borges and M. Tulio Valente, "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[29] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A Systematic Review of API Evolution Literature," *ACM Computing Surveys*, vol. 54, no. 8, pp. 171:1–171:36, 2021.

[30] R. Robbes and M. Lungu, "A Study of Ripple Effects in Software Ecosystems," in *33rd International Conference on Software Engineering (ICSE, NIER Track)*, 2011, pp. 904–907.

[31] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "An empirical study on the impact of refactoring activities on evolving client-used apis," *Information and Software Technology*, vol. 93, no. C, pp. 186–199, jan 2018.

[32] S. Mostafa, R. Rodriguez, and X. Wang, "Experience paper: A study on behavioral backward incompatibilities of Java software libraries," in *26th International Symposium on Software Testing and Analysis (ISSTA)*, Jul. 2017, pp. 215–225.

[33] K. Jezek, J. Dietrich, and P. Brada, "How java apis break - an empirical study," *Information and Software Technology*, vol. 65, no. C, pp. 129–146, sep 2015.

[34] D. Dig and R. Johnson, "How do apis evolve? a story of refactoring: Research articles," *Journal of Software Maintenance and Evolution*, vol. 18, no. 2, pp. 83–107, mar 2006.

[35] 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 *28th International Conference on Neural Information Processing Systems (NIPS)*, 2015.

[36] J. Gesi, S. Liu, J. Li, I. Ahmed, N. Nagappan, D. Lo, E. S. de Almeida, P. S. Kochhar, and L. Bao, "Code smells in machine learning systems," 2022.

[37] Y. Tang, R. Khatchadourian, M. Bagherzadeh, R. Singh, A. Stewart, and A. Raja, "An empirical study of refactorings and technical debt in machine learning systems," in *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 238–250.

[38] H. Zhang, L. Cruz, and A. van Deursen, "Code smells for machine learning applications," in *1st International Conference on AI Engineering: Software Engineering for AI*, ser. CAIN'22, 2022, pp. 217–228.