Restoring the Executability of Jupyter Notebooks by Automatic Upgrade of Deprecated APIs

Chenguang Zhu[1], Ripon K. Saha[2], Mukul R. Prasad[2], and Sarfraz Khurshid[1]

[1]The University of Texas at Austin, [2]Fujitsu Research of America, Inc.

Email: cgzhu@utexas.edu, rsaha@fujitsu.com, mukul@fujitsu.com, khurshid@utexas.edu

Abstract—Data scientists typically practice exploratory programming using computational notebooks, to comprehend new data and extract insights. To do this they iteratively refine their code, actively trying to re-use and re-purpose solutions created by other data scientists, in real time. However, recent studies have shown that a vast majority of publicly available notebooks cannot be executed out of the box. One of the prominent reasons is the deprecation of data science APIs used in such notebooks, due to the rapid evolution of data science libraries. In this work we propose RELANCER, an automatic technique that restores the executability of broken Jupyter Notebooks, in near real time, by upgrading deprecated APIs. RELANCER employs an iterative runtime-error-driven approach to identify and fix one API issue at a time. This is supported by a machine-learned model which uses the runtime error message to predict the kind of API repair needed - an update in the API or package name, a parameter, or a parameter value. Then RELANCER creates a search space of candidate repairs by combining knowledge from API migration examples on GitHub as well as the API documentation and employs a second machine-learned model to rank this space of candidate mappings. An evaluation of RELANCER on a curated dataset of 255 un-executable Jupyter Notebooks from Kaggle shows that RELANCER can successfully restore the executability of 56% of the subjects, while baselines relying on just GitHub examples and just API documentation can only fix 38% and 36% of the subjects respectively. Further, pursuant to its real-time use case, RELANCER can restore execution to 49% of subjects, within a 5 minute time limit, while a baseline lacking its machine learning models can only fix 24%.

Index Terms—data science, API migration, software evolution

I. INTRODUCTION

The ready availability of sensors and inexpensive compute resources, coupled with a number of significant advances in machine learning and data analytics has fueled an explosive growth in the field of *data science* [1], [2]. Kaggle, the relatively nascent online community of data scientists, already boasts over 6 million users and hosts 50,000 public datasets [3].

Data scientists typically follow the paradigm of *exploratory programming*, iteratively refining code to comprehend new data and extract meaningful insights from it [4], [5]. To do this they actively try to re-use and re-purpose solutions created by other data scientists, in *real time*. Indeed, Kaggle proudly claims that: *"Inside Kaggle you'll find all the code & data you need to do your data science work"* [3]. Jupyter Notebooks [6] - interactive computational structures which interleave code snippets, natural language text, computation results, and visualizations - have become the medium of choice for data scientists to program, record, and *share* data analyses [7], [8]. Kaggle hosts 400,000

such public notebooks [3]. Similarly, there are currently nearly 10 million Jupyter Notebooks hosted on GitHub, with the number having grown 8-fold over just the last two years [9].

A data scientist pursuing exploratory programming by re-using existing Jupyter Notebooks would require such notebooks to be easily *reproducible* or at the least *executable*. In fact, producing *"reproducible computational workflows"* was intended to be one of the defining features of Jupyter Notebooks (and computational notebooks in general) [10], [11]. However, this promise has not been realized in practice. A large-scale empirical study by Pimental et al. on Jupyter Notebooks projects on GitHub found that only 24.11% of their selected notebooks could execute without errors, and only 4.03% reproduced the original results [12]. The concerns about executability are echoed in a recent survey of data scientists [13]. Similarly, our study of a sample of machine learning notebooks on Kaggle (reported in Section II-A) revealed that 47% of them could not be executed.

Recent studies have identified a number of root causes for why archived Jupyter Notebooks cannot be easily re-executed. These include, ambiguity in the execution order of the notebook cells [14], lack of knowledge of the notebook's execution environment [15], and references to external resources (such as data on external servers) [12], [14]. An emerging body of work also aims at restoring the executability of such notebooks by automatically inferring correct cell execution order [14] or inferring an appropriate execution environment [15]. Other studies have highlighted the landscape of rapidly evolving Python libraries, particularly data science libraries, which leads to unhandled deprecation issues in programs using these libraries [16], [17], [18]. In fact, our study of Kaggle notebooks (Section II-A) found that deprecated APIs was at least one of the reasons (if not the only reason) for the unexecutability of at least 31% of the unexecutable notebooks. Therefore, in this paper, we propose a technique to restore the executability of Jupyter Notebooks by automatically diagnosing, inferring, and upgrading deprecated APIs used by the notebook.

There is a rich body of existing work on computing API mappings across pairs of libraries, languages, or platforms, to be then used to migrate a project from a source library (platform) to a target library (platform) [19], [20], [21], [22], [23], [24], [25], [26]. There is also recent work that can perform an end-to-end migration [27], [28], [18]. However, our target use-case - automatically making broken Jupyter Notebooks executable in near real-time, by fixing deprecated

APIs - presents some unique challenges and opportunities, which precludes the direct application of existing solutions. First, many of the API mapping techniques [20], [23], [29], [30], [31], [32] only create the mappings rather than perform an end-to-end migration, which our use-case mandates. Second, all the techniques that perform complete migration [27], [28], [18] rely on a reference implementation in the source library/platform as a strong oracle for migration. Our use-case lacks such a reference implementation - we only have an unexecutable notebook. Third, each of the existing techniques rely on a single information source to compute the mappings or migration, typically either existing examples of updates (e.g., on GitHub) [27], [28] or API documentation [18]. However, given the rapid evolution of data science libraries, public examples of recent API changes may not exist [18]. Further, even the API documentation may not comprehensively capture all deprecation information [16]. This motivates a migration technique that combines knowledge from both sources. Finally, our use-case of supporting exploratory programming requires a technique that can fix broken notebooks (chosen by the user for exploration) in *near real time*. Traditional API migration techniques are not similarly constrained.

Insights. We design our approach based on three key insights. First, we observe that individual API upgrades (even different instances of the same change) can be successfully diagnosed and performed independent of each other, i.e., one instance at a time. Second, we build on the observation made in other recent work [17], [18] that runtime error messages in Python, in particular those of Python data science programs, are often accurate enough to unambiguously diagnose the broad cause of error. Specifically, we observe that for deprecated APIs the runtime error can be used to accurately flag the *kind* of program element requiring a change - the API name or package name, a parameter, or the value of a parameter. The third insight is that previous instances of API upgrades (e.g., on GitHub) and API library documentation are complementary sources of knowledge that can collectively inform an automated API migration technique.

Proposed approach. Based on these insights, in this work we propose RELANCER [1], a technique for automatically restoring the executability of broken Jupyter Notebooks, in near real-time, by upgrading deprecated APIs. RELANCER employs a divide-and-conquer approach to upgrade a broken notebook, using runtime errors to iteratively identify and fix one API issue at a time. This iterative strategy is supported by two key components. The first component is a machine learned model that uses the runtime error message to predict the *kind* of repair action to perform, specifically a change in the API name (or package), a parameter name, or a parameter value. This decision accelerates the search by directing focus to the appropriate search space. RELANCER creates this space by aggregating candidates from both GitHub examples as well as API documentation. In the case of name changes to the API or

---

[1]from the French word meaning *to revive*, since RELANCER revives broken notebooks by restoring their executability.

its package, this space could be fairly large. Thus, RELANCER uses a second machine learned model to produce a ranked list of candidate API mappings organically combining knowledge from the two sources.

We evaluate RELANCER on a curated dataset of 255 unexecutable Jupyter Notebooks from Kaggle. RELANCER can successfully restore the executability of 56% of the subjects, while baselines relying on just GitHub examples and just API documentation can only fix 38% and 36% of the subjects respectively. We note that a response time of up to a few minutes is recognized as real-time or near real-time performance for Big Data and data analytics applications [33], [34]. Therefore, we use 5 minutes as a threshold of near real-time performance for the purposes of this paper. Pursuant to its real-time use case, within a 5 minute time-span RELANCER can restore execution to 49% of subjects while a baseline lacking its machine learning models can only fix 24%.

This paper makes the following contributions:

- **Problem:** We highlight the problem of restoring execution of Jupyter Notebooks by upgrading deprecated APIs, in real time, in order to support the use-case of exploratory programming typically employed by data scientists.
- **Technique:** An automated machine learning based technique RELANCER, to orchestrate this upgrade for unexecutable notebooks.
- **Evaluation:** An evaluation of RELANCER on a curated dataset of 255 unexecutable Jupyter Notebooks mined from Kaggle, and ablation studies comparing its performance against 5 different baselines.
- **Implementation & Dataset:** A public release of the source code and dataset used in this paper, to promote replication and open science, is available at https://sites.google.com/view/relancer.

II. MOTIVATION

In this work we restrict our scope to Jupyter Notebooks on Kaggle, the most popular online forum for data scientists. Further, we focus on notebooks written in Python, the language of choice in data science [9] and specifically ones performing predictive (i.e., machine learning (ML)) tasks, since ML is one of most important end-points for data science.

A. A Study on Kaggle Notebooks

We hypothesize that the problem of unexecutable notebooks, reported by recent studies [12], [14], [15] for GitHub projects, is also mirrored on Kaggle. We further hypothesize that the issue of deprecated DS APIs, due to the rapid evolution of DS libraries [16], [18] should be a significant factor in the unexecutability. To validate these hypotheses, we conduct a lightweight study of Jupyter Notebooks on Kaggle.

Data Collection. We use Meta Kaggle [35], the official dump of Kaggle meta data, as the data source for this study. Meta Kaggle links to Jupyter Notebooks on Kaggle, including competitions, datasets, kernels, and discussions on a daily basis. We downloaded the snapshot of Meta Kaggle on July 20,

2020 for this study. It consists of 49,061 datasets and 381,556 notebooks (kernels).

Sampling. To keep the study simple and efficient, we impose several filtering criteria. Starting with Python notebooks for predictive tasks, we exclude datasets containing media data (e.g., images, audio, and video), or spanning over multiple or large files (i.e., >500 MB). We calculated that we require a sample size of 4,000 notebooks to achieve a 99% confidence level within a margin of error of $\pm 2\%$ [36]. To this end, using upvotes on Kaggle as a proxy of quality, we sorted all datasets that passed our filtering criteria by the number of upvotes and similarly notebooks for each dataset by the same metric. We selected top 100 notebooks (or as many available if fewer) per dataset, to respect diversity. This yielded 4043 notebooks from the top 350 datasets.

Execution Environment. To execute the Jupyter Notebooks, we convert them into Python programs using `nbconvert` [37]. We set up a unified virtual execution environment on top of the standard Anaconda [38] scientific computing environment of Python 3.6, also used in other studies [12], [14], [15]. This standard environment includes all the popular libraries (over 200 library packages). Further, we performed a static import analysis on our notebook corpus and manually installed the 20 most commonly used Python libraries.

Execution & Analysis. Among the 4043 notebooks in our study corpus, 2,155 notebooks executed successfully while 1,888 notebooks, i.e., 47% of the total corpus did not execute due to at least one error. Consistent with previous studies on GitHub [12], [14], this shows that unexecutable notebooks is also a significant problem on Kaggle, although the exact numbers differ due to platform differences.

Deprecation Errors. To quantify the impact of API deprecation on notebook executability, we created a ground-truth list of deprecated APIs by manually inspecting the release notes of 12 popular DS libraries, namely: scikit-learn [39], pandas [40], seaborn [41], NumPy [42], SciPy [43], XGBoost [44], Plotly [45], TensorFlow [46], Keras [47], statsmodels [48], imbalanced-learn [49], and CatBoost [50]. These libraries are a subset of the top-20 libraries found in our Kaggle study, which systematically report API deprecations in their documentation. In total, we manually identified 317 upgrades (i.e., deprecations) through the release notes, including 198 function name deprecations and 119 parameter deprecations. Checking these against our corpus of 1,888 unexecutable notebooks showed that 582 of the notebooks, i.e., 31% contained at least one of these deprecated APIs.

B. A Motivating Example

Figure 1 presents a fragment of a real-world Jupyter Notebook on Kaggle [51] with a patch generated by RELANCER, that fixes all API deprecation errors. The original notebook, submitted in 2017, is a high-quality notebook that won a bronze medal on Kaggle. Further, it has gathered 37,767 views and 161 forks (as of April 12th, 2021). This indicates that the notebook has been very useful to other data scientists for learning or expediting their work. However, the original notebook no longer executes with the latest default Anaconda environment due to several deprecated APIs. In particular, Figure 1 shows five locations containing three deprecated APIs:

1) `sklearn.cross_validation.train_test_split`
2) `sklearn.cross_validation.`
   `grid_search.GridSearchCV`
3) `sklearn.cross_validation.ShuffleSplit`

The first two APIs are deprecated due to a change in the package structure, i.e., moving from `sklearn.cross_validation` to `sklearn.model_selection` [52]. This kind of deprecation is called *function deprecation* [16]. The deprecation in third API is more complicated. In addition to the aforementioned package change, the parameter `n_iter` was replaced by `n_splits` starting from version 0.18 [52]. This is called *parameter deprecation* [16]. Therefore, if a data scientist wants to run the notebook on her own machine, often the first step of trying to re-use it, she needs to laboriously find, diagnose, and fix each of the deprecation errors, one by one. This is a big barrier to re-use an otherwise high-quality notebook.

Although API migration is a well-established research area and there are even a few recent techniques that perform end-to-end API migration [27], [28], [18], the Jupyter Notebook in Figure 1 would be outside the scope of these techniques. First, these techniques [27], [28] rely on a single information source, typically either GitHub or the API documentation. However, for the aforementioned example, the patch for `ShuffleSplit` with the argument change is not directly available on GitHub. On the other hand, a tool that only uses documentation may not find the correct mapping for other deprecated APIs in the example easily since `cross_validation` and `model_selection` are not textually similar. A brute-force trial of each API in the documentation would be impractical, especially with multiple deprecated APIs. Specifically, in our experiment, this notebook could not be fixed using only documentation within a 30-minute time limit (Section V, RQ2).

To overcome these challenges, RELANCER effectively combines two primary sources of information - GitHub examples and API documentation - and fixes the problem iteratively guided by the error message. Specifically, RELANCER first executes the original notebook, and find an `ImportError` at line 235, as shown in Figure 2.

At this point, the repair action predictor component of RELANCER automatically identifies that the fully qualified name of a module needs to be changed. RELANCER identifies the affected module name from the error message and also identifies and localizes the affected API by analyzing the abstract syntax tree (AST) of line 240. Then RELANCER searches for potential mappings: $API_{old} \rightarrow API_{new}$ on GitHub and documentation and passes them to its learning-to-rank model. For this specific case, RELANCER ranks the correct API_{new}: `model_selection.train_test_split` on top. The reason is that although the textual similarity between old and new package names is low, this change is frequent on Github. While applying a patch, RELANCER not only fixes the actual reference

```
  1  # kaggle.com/sagarnildass/predicting-boston-house-prices
...  ...
 22  import numpy as np # linear algebra
...  ...
235  -from sklearn import cross_validation
235  +from sklearn import model_selection
...  ...
240  -X_train, X_test, y_train, y_test = cross_validation. train_test_split(features, prices, test_size=0.2, random_state=42)
240  +X_train, X_test, y_train, y_test = model_selection. train_test_split(features, prices, test_size=0.2, random_state=42)
241   print("Training and testing split was successful.")
...  ...
284   from sklearn.tree import DecisionTreeRegressor
285  -from sklearn.cross_validation import ShuffleSplit
285  +from sklearn.model_selection import ShuffleSplit
...  ...
290   # Create 10 cross-validation sets for training and testing
291  -cv = ShuffleSplit(X.shape[0], n_iter=10, test_size=0.2, random_state=0)
291  +cv = ShuffleSplit(X.shape[0], n_splits=10, test_size=0.2, random_state=0)
292   train_sizes = np.rint(np.linspace(1, X.shape[0]*0.8-1,9)).astype(int)
...  ...
514   from sklearn.metrics import make_scorer
515  -from sklearn.grid_search import GridSearchCV
515  +from sklearn.model_selection import GridSearchCV
516  ...
```

Fig. 1: A motivating and illustrative example of RELANCER.

Traceback (most recent call last):
File "predicting-boston-house-prices.py", line 235, in <module>
from sklearn import cross_validation
ImportError: cannot import name 'cross_validation'
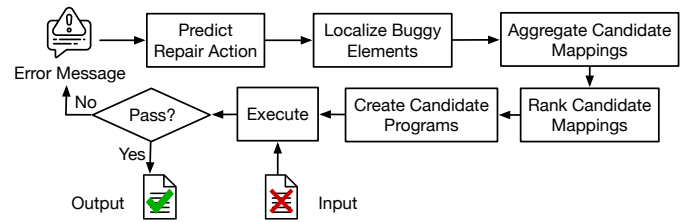
Fig. 2: The error message thrown by line 235.

Fig. 3: Overview of RELANCER.

but also fixes the necessary import statements. For example, as a result of changing `cross_validation.train_test_split` to `cross_validation.train_test_split`. RELANCER also fixes lines 235 and 285. Now after validation, RELANCER finds a new error message pointing to another error at line 291.

Although the deprecated API involved two issues: deprecated module name and parameter name, the first issue at line 285 is already fixed by RELANCER while fixing the previous API. Therefore, the new error shows a `TypeError` at line 291. RELANCER predicts that the appropriate repair action for this error is a parameter name change and identifies the problematic parameter name from the error message. Then RELANCER again leverages Github and documentation to get all the candidate parameter names and validates one by one. In this way, RELANCER fixes all the deprecated errors until the notebook fully executes.

In summary, this is a large notebook that contains 701 lines of code containing five unique API deprecation errors at seven locations. To give an idea about the number of candidate patches, RELANCER had on average 355 choices per location. However, RELANCER systematically narrowed down the search space and fixed all the errors in ten minutes.

III. RELANCER

Given a Jupyter Notebook that does not currently execute, due to one or more deprecated APIs, RELANCER automatically applies necessary program transformations to upgrade all the deprecated APIs so that the resulting notebook executes without any error. Figure 3 presents an overview of RELANCER. At a high level, RELANCER iteratively performs the following operations until the notebook is error-free:

1) **Analysis of Error.** This step analyzes the current error message: first to predict the atomic repair action that is required to solve that particular error and then to identify the deprecated API.
2) **Aggregation of Candidate Mappings.** This step mines the Github repositories and API documentation to aggregate the candidate mapping between program elements: $PE_{dep} \rightarrow PE_{new}$ where a PE is an API or a parameter name, or an argument value.
3) **Ranking of Candidate Mappings.** This step employs a learning-to-rank model to rank the candidate mapping in such a way that the more promising a mapping is, the higher it is in the ranking.

4) **Creation and Validation of Modified Programs.** This step creates a candidate program for each mapping one by one based on the ranking, and executes it to check if it fixes the current error.

The subsequent sections describe each of the above steps of RELANCER in more detail.

A. Step-1: Analysis of Error Message

The analysis of error messages consists of two parts, namely (1) predicting the repair action to take and (2) localizing the buggy element to apply it on.

1) Prediction of Repair Action: There are mainly three kinds of API deprecation issues in Python programs: *function deprecation*, *parameter deprecation*, and *parameter's value deprecation* [16]. Broadly, these map to the kind of repair action required to fix the bug. We make the observation that typically the error message is indicative of the kind of deprecation. For example, fixing a function deprecation requires renaming the API. Some example errors for such a deprecation are:

> *ImportError: cannot import name 'jaccard_similarity_score'.*

> *ModuleNotFoundError: No module named 'sklearn.grid_search'.*

Intuitively, there are some common patterns in the messages, such as a particular name cannot be imported or found. However, they are not exactly the same and may vary across libraries and APIs. To leverage this information systematically and robustly, we build a machine learning classifier to predict the repair action directly from the error message. Specifically, the classifier predicts one of three different repair actions: i) changing the fully qualified name of a function, ii) changing parameter name, and iii) changing an argument value.

Building the Classifier. We first apply standard text pre-processing techniques to clean the error message, such as removing non-alphabetic characters, tokenizing and normalizing text. Then we convert the text into a matrix of token counts where each word becomes a feature and value represents the count of that token. So after this step, each error message in the training dataset becomes a tuple of numeric values representing text in the error message and the target is the corresponding repair action. To predict the repair action from the error message, we use Linear Support Vector Classification [53] since it is generally accepted that it is well suited for text classification [54].

2) Creation of Training Data: To create a training dataset for the repair action predictor, we need instances of pairs of buggy and fixed versions of notebooks corresponding to deprecated APIs, with the buggy version providing the error message and an executable fixed version as the ground truth for the required repair action. However, Kaggle does not provide for the tracked versioning and collaborative project editing (like GitHub) to facilitate gathering such instances. Therefore, we devise a novel mutation-based technique to systematically create our training data.
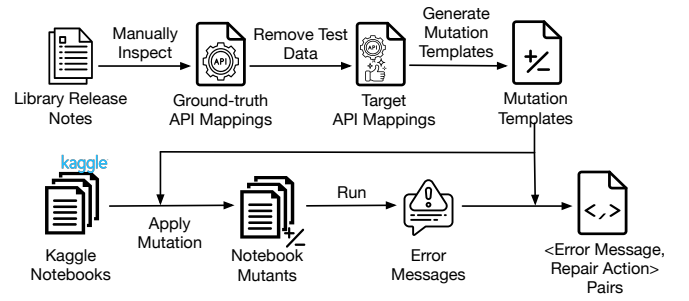

Fig. 4: Mutation-based training dataset creation.

Approach. Our approach is inspired by techniques for data augmentation [55] and simulation-based training data generation [56], [57], [58] used to generate synthetic, but realistic, training data for machine learning. Specifically, we use the insight that given a known instance of an API deprecation $PE^i_{dep} \rightarrow PE^i_{new}$ (which by definition states the repair action) and a working notebook that uses the API PE^i_{new}, we can re-create the buggy version of the notebook by the deprecation mutation $PE^i_{new} \rightarrow PE^i_{dep}$, and execute the buggy notebook to generate the error message. This provides a pair of an error message and its corresponding repair action, i.e., a single instance of training data. Similarly, we use the set of manually collected deprecation mapping $\{PE^1_{dep} \rightarrow PE^1_{new}, PE^2_{dep} \rightarrow PE^2_{new}, ...\}$ and the set of executable notebooks identified in our Kaggle study (Section II-A), and systematically inject deprecation mutations into executable notebooks that are using the new version of the respective API. Figure 4 summarizes our overall workflow.

3) Localization of Buggy Program Elements: As Figure 2 shows, typically an error message describes a single problem and consists of a stack trace, followed by an error description in natural language text. RELANCER exploits this broad structure of the error message to extract (1) the buggy line number in the main file and (2) the specific program element on it, implicated in the deprecation. To do this, RELANCER analyzes the error message along with the corresponding source file as follows.

RELANCER first extracts the line number of the most recent (last) call site for the main source file from the stack trace in the error message–e.g., Line 235 in Figure 2. Second, it extracts all the deprecation-related program elements, specifically module/function/parameter names and values at that line from the AST of that source file. Third, RELANCER searches all the candidate program elements that it identified from the source code, in the text portion of the error message to pinpoint the exact program element that the error is complaining about. For our motivating example, it is the `cross_validation` module.

B. Step-2: Aggregation of Candidate Mappings

Given a deprecated program element PE_{dep} identified in the previous step, where PE is a module name, a function name, a parameter name, or a parameter value, RELANCER utilizes two sources of information: i) the API documentation

and ii) Github to identify all possible candidate mappings: $PE_{dep} \rightarrow PE_{new}$.

1) Mining API documentation: The objective of this step is to create a knowledge base offline that consists of each API from the latest version of each library under consideration. Most of the popular libraries use a very similar format document structure for their API documentation. For each API method, RELANCER parses (using *Beautiful Soap* [59]) the HTML page to collect its parameter names and the set of possible discrete values (if available). It then stores the information of these libraries in an internal JSON database.

2) Mining of Github: Since RELANCER focuses on fixing one PE_{dep} at a time, its on-demand GitHub search identifies a list of potential PE_{new} that can replace PE_{dep}.

Identifying API Upgrade Mappings. To identify plausible API upgrades for a given previously unseen API, we leverage the idea of mining code changes[60]. RELANCER leverages Github's own search engine through the provided REST API [61] to search relevant example changes efficiently.

First, given a deprecated API, F_{dep} that has been identified in the previous step and needs to be fixed, RELANCER tokenizes the fully qualified name of F_{dep} and then constructs a GitHub search query by taking each token *plus* the keywords: "update", "upgrade", "replace", and "deprecate". Then RELANCER executes this query on GitHub using GitHub's REST API, which returns a list of commits sorted by GitHub's "Best Match" metric [62] in descending order.

Second, RELANCER excludes irrelevant commits, i.e., commits that do not contain an upgrade of F_{dep}. Such irrelevant commits can be introduced due to GitHub's imprecision of searching. For each commit in searching result, RELANCER extracts the abstract syntax tree (AST) of the program version before (V_0) and after (V_1) the commit. It discards a commit if the AST of V_0 does not include F_{dep}.

Third, RELANCER analyzes the AST difference of V_0 and V_1. For a deleted deprecated API call node F_{dep} in V_0, we consider an added node in V_1 as an update of F_{dep} if the deletion and addition occur at the same level of the AST, and the deleted and added nodes are under the same parent. If there are multiple addition candidates for a deletion, we pair the deleted node with the addition candidate whose name has the highest textual similarity with it. After AST node changes are identified, RELANCER discards any commit that does not involve a change in F_{dep}. Finally, for each update of F_{dep}, RELANCER extracts the corresponding new API: F_{new}, adding them to a list. This resulting list consists of the candidate APIs mined from GitHub and is cached (stored locally) for future use, to avoid repeating the search if RELANCER encounters the same deprecated API call multiple times.

Identifying Parameter Mappings. The APIs in data science libraries generally have a large set of parameters but few discrete options per parameter [63]. Furthermore, popular data science programming languages like Python can accept parameters in the form of key-value pairs. Therefore, mining the exact change (patch) that RELANCER needs for the subject error is challenging. However, since RELANCER narrows down the scope of search in the previous steps quite a lot: i) by first fixing function deprecation by F_{new} and ii) then identifying the deprecated parameter name P_{dep} from the error message, it launches a focused search to collect the set of all parameter names and the values developers used on GitHub for that specific API. Finally, RELANCER returns a map where the keys are comprised of all the parameter names and values represent the set of values that parameter can take. Then a parameter mapping: $P_{dep} \rightarrow P_{new}$ is established where P_{new} is not currently used in the current call site.

3) Compiling the Final List of Candidate Mappings: As we discussed in Section III-A, RELANCER allows three kinds of upgrades: i) API name changes, ii) parameter name changes, and iii) argument value changes. Furthermore, RELANCER performs one single upgrade at a time. Therefore, for a given deprecated program element, i.e., an API, a parameter, or a value, RELANCER takes a union of both sets of elements mined from: i) the API documentation and ii) GitHub to compile a final set of candidates: $PE_{dep} \rightarrow \{PE_{new}^1, PE_{new}^2, .., PE_{new}^n\}$.

C. Step-3: Ranking of Candidate Mappings

For a given deprecated API: F_{dep}, there can be hundreds of candidate APIs: $\{PE_{new}^1, PE_{new}^2, .., PE_{new}^n\}$. Further, there can be multiple errors in a program, which can increase the search space exponentially. Therefore, an effective ranking strategy is important for the success of RELANCER. Learning to rank is an effective ranking strategy that has been found useful in many applications, including various software engineering tasks such as defect prediction [64] and bug localization [65]. Inspired by these applications, we develop a machine learning based learning-to-rank strategy to rank all the candidate fixes for a given deprecated API. However, for any machine learning technique, designing the features that are related to the target is critical. We design the following four features for our task: two features from the API documentation and two features from GitHub.

- **Occurrences on GitHub** (OG). The number of times RELANCER found a mapping: $F_{dep} \rightarrow F_{new}$ on GitHub commits.
- **Percentage of OG** (POG). It is calculated by the ratio of OG to the number of times RELANCER found a mapping where F_{dep} exists.
- **Distance between Fully Qualified Names** (D_{FQN}). RELANCER computes the distance between the fully qualified name of F_{dep} and F_{new} by the Damerau–Levenshtein Distance algorithm [66]. Then the value is normalized between 0 and 1, with zero corresponding to identical names.
- **Distanceof Simple Name** (D_{Simple}). RELANCER follows the same approach as D_{FQN}, but it computes the scores between the simple names (without considering package or module name) of F_{dep} and F_{new}.

Training Learning-to-Rank Model. We create training data from our ground-truth dataset that are not part of test data. For each ground-truth mapping: $F_{dep} \rightarrow F_{new}$, F_{new} is used as a *positive instance* in the training dataset. However, to learn the

characteristics of meta-features for the correct mappings, we also need some *negative candidate mappings* that syntactically represent a correct replacement but functionally not. So we use the same technique described in Section III-B to find a complete list of candidate program elements and randomly choose four negative examples for F_{dep}. We follow the same procedure to collect all the positive and negative instances for each target mapping in the ground truth dataset. Then we compute all the four aforementioned features for each of the positive and negative mappings. Therefore, our final training dataset has a collection of data samples where each row presents an old to new API mapping in terms of the four feature values along with the information whether it is a valid mapping.

Now we formulate our learning-to-rank algorithm as a binary classification problem, where our objective is to determine whether a particular mapping is correct. To this end, we use LightGBM [67] as our classification model, which is a widely used machine learning technique in practice.

Ranking Candidates in Operation. In operation, for a given deprecated API: F_{dep}, RELANCER gets all the candidates: $\{F_{new}^1, F_{new}^2, .., F_{new}^n\}$ from Step-2 and passes to the trained LightGBM model. For each candidate (F_{new}^i), the model computes a probability score for being it to be correct. Then RELANCER uses that probability score to rank all the candidate mappings. It should be noted that once function mapping is found and parameter name or value is identified from the error message, the number of candidates for a particular parameter is not large. Therefore, RELANCER follows the order in the documentation to rank parameter mappings.

D. Step-4: Creation and Validation of Modified Programs.

Given a ranked list of candidate mappings, RELANCER starts iterating through each mapping from the top of the ranked list. For a given mapping: $PE_{dep} \rightarrow PE_{new}$, RELANCER replaces PE_{dep} by PE_{new} to create a new version of the candidate program and validates the change through execution. If the type of error message is unchanged, RELANCER moves to the next mapping. However, if RELANCER gets a new error message, it assumes that it fixed the current error and goes to Step-1 to analyze the error message. RELANCER continues these steps until there is no error in the notebook or the entire search space of candidate mappings is exhausted, or a given time limit is exceeded.

IV. EXPERIMENTAL SETUP

A. Implementation

RELANCER is implemented in the Python programming language. It has mainly three components: i) mining API documentation, ii) on-demand searching of edit-examples on Github, and iii) AST manipulation and validation. We use *Beautiful Soup* [59] for parsing the API documentation HTML pages to extract relevant information. We use *nbconvert* [37] to convert Jupyter Notebooks into Python programs to ease the process of automatic running and generating error messages. We use GitHub*REST API* [61] for searching potential examples on GitHub and then use *LibCST* [68], a popular static analysis

framework for Python to analyze AST diff and so on. We also use LibCST for making necessary changes to generate potential fixes for the deprecation errors. We use *scikit-learn* [39] and *LightGBM* [67]–two popular open source machine learning libraries to implement our ML models.

B. Creation of Dataset

Since RELANCER is a machine learning based technique, to evaluate it properly, we need a training dataset and a test dataset that are *mutually exclusive*. Our preliminary study in Section II-A, yielded 582 unexecutable notebooks that contain at least one deprecated API. However, since the objective of RELANCER is to fix only deprecated errors, we wanted to filter out those notebooks that have irrelevant errors. However, it is very challenging to detect all such irrelevant errors.

Error analysis provides an automated way to detect candidate deprecated errors since a previous study [17] suggests that certain kinds of error messages are related to API usage problems, which can lead us to deprecated errors. However, an error message only gives a description of the first error during the execution of the program. Therefore, we targeted capturing at least those notebooks where the first error is related to deprecated APIs. We performed an automated analysis on the error messages to identify the initial list of candidate deprecated errors that contains one of the five errors: i) import error, ii) module not found error, iii) type error, iv) value error, or v) attribute error. Then we searched those APIs in the set of our ground-truth API dataset. Finally, we identified 255 notebooks where the first failure is due to a deprecation error. Therefore, we use all the 255 notebooks as our test dataset to evaluate RELANCER. Table I provides more details about the dataset.

C. Training RELANCER

Training Repair Action Model. To keep the training dataset and test dataset mutually exclusive, our mutation based framework (described in Section III-A2) did not use any deprecated APIs, $F_{test} = \{F_{test}^1, F_{test}^2, ..\}$ from the test dataset. To this end, we eliminated F_{test} from our ground-truth dataset, F_G and only used the remaining deprecated APIs ($F_{tr} = F_G - F_{test}$) to design mutation operators. We use all the 2,155 executable notebooks obtained in our preliminary study (Section II-A) as seed programs to create the training data as described in Section III-A2. Finally, we constructed a training dataset of 500 <error message, repair action> pairs.

We also followed the approach described in Section III-C to construct the training data for the learning-to-rank model from F_{tr}. This gave us a training dataset of 485 instances.

D. Research Questions

We evaluate RELANCER with respect to the following research questions.

RQ-1: How effective is RELANCER at fixing API deprecation issues of Jupyter Notebooks?

RQ-2: Does RELANCER need different sources of information to perform well?

RQ-3: Do the machine learning models in RELANCER help with upgrading more APIs in near real-time?

TABLE I: Subjects: Jupyter Notebooks on Kaggle

Description	Total	Max	Min	Median	Average
#Datasets	110	-	-	-	-
#Notebooks	255	-	-	-	-
#Notebooks per dataset	-	13	1	1	2.32
#Libraries	8	-	-	-	-
LOC	-	739	15	120	152.02
LOC (including comments)	-	1193	21	208	263.81
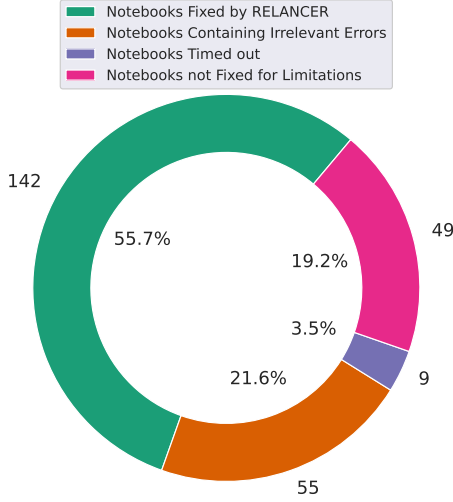
Fig. 5: Effectiveness of RELANCER

E. Experimental Configurations

All the experiments are performed on a 4-core Intel(R) Core(TM) i7-8650 CPU @ 1.90 GHz machine with 16GB of RAM, running Ubuntu 16.04, with Python 3.6.10 and Conda 4.7.10. We set a timeout of 30 minutes for each notebook.

V. EVALUATION

A. RQ-1: Effectiveness of RELANCER

To evaluate the effectiveness of RELANCER, we ran it on all the 255 notebooks in our test dataset with a timeout
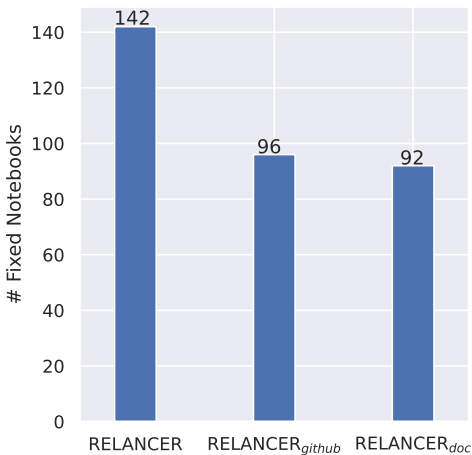

Fig. 6: Effect of removing one source of information

```
1 -from sklearn.metrics import jaccard_similarity_score
2 +from sklearn.metrics import jaccard_score
3  ...
4  knn_yhat = neigh.predict(X_test)
5 -print("KNN Jaccard index: %.2f" %
   jaccard_similarity_score(y_test, knn_yhat))
6 +print("KNN Jaccard index: %.2f" %
   jaccard_score(y_test, knn_yhat, average="micro"))
```

Fig. 7: Another illustrative example of RELANCER.

of 30 minutes for each notebook. Then we measure the effectiveness in terms of the number of fixed notebooks and time to fix the notebooks completely. Figure 5 shows that RELANCER was able to fix 142 out of 255 notebooks completely. Among the 142 notebooks it fixed, 107 notebooks required upgrading only the fully qualified name of APIs. Other 35 notebooks required changing parameter names or argument values with or without the fully qualified name of APIs. As we have shown in the motivating example, RELANCER can fix multiple errors in notebooks that require sophisticated fixes in both API's fully qualified name and parameters. The highest number of deprecated APIs RELANCER has upgraded in a single notebook is eight. Further, it should be noted that Python is a rich language that allows for optional parameters which use a default value if that is not explicitly given by developers. However, sometimes even that default value can be deprecated, leaving the notebook broken. In this case, RELANCER can explicitly pass the problematic parameter with the correct value. For example, as Figure 7 presents after the deprecated API `jaccard_similarity_score` got changed to `jaccard_score`, the default value of `average` is no longer valid. So RELANCER added that parameter in the call site with the correct value.

The size of the patches generated by RELANCER varied from 2 to 25 lines of code (LOC) with a median size of 3 LOC. When RELANCER was able to fix a notebook, the execution time varies from 2 seconds to 28 minutes, with on average two minutes (median 17 seconds) per notebook. This result demonstrates that RELANCER can be used in real time to fix deprecated errors.

Validation. We manually validated all the successful patches to make sure that RELANCER used the latest API and/or parameter for each deprecated API. Further, we also conducted an objective evaluation to make sure that the fixed notebooks serves the actual developer's intention. To this end, we leveraged the output cell feature of Jupyter Notebooks. More specifically, Jupyter Notebooks often have the output cell that stores the accuracy from the original run. We were able to identify the original accuracy from the output cell for 82 notebooks and found that the new accuracy we got after fixing the deprecation errors is within the 1% (median) of original accuracy. It should be noted that most machine learning models have inherent randomness. Therefore, we believe that such a small accuracy difference is not surprising.

```
1  -import plotly.plotly as py
2  +import chart_studio.plotly as py
3  py.iplot(data, filename=...)
```

Fig. 8: Example of moving an API to another library

> **Answer to RQ1:** RELANCER restored execution to 56% of the notebooks, fixing as many as eight errors in a single notebook. The median time to fix a notebook is only 17 secs, making it suitable for real-time use.

B. RQ-2: Contribution of Different Sources of Information

To understand whether both sources of information: the API documentation and Github help RELANCER achieve its full capability, we created two baseline tools from RELANCER, which use only a single source of information along with error message. RELANCER$_{github}$ does not use any information from the API documentation and RELANCER$_{doc}$ does not use any information from GitHub. All other functionalities in two baselines are similar to RELANCER. Here we would like to stress the fact that since there is no tool available for upgrading APIs for Jupyer Notebooks, to the best of our knowledge, we cannot directly compare any existing technique. However, conceptually the baselines: RELANCER$_{github}$ and RELANCER$_{doc}$ roughly simulate the approaches that use only Github such as Meditor [27] and only the API documentation along with error messages such as SOAR [18].

The experimental results in Figure 6 show that RELANCER outperforms both RELANCER$_{github}$ and RELANCER$_{doc}$ significantly in terms of the number of fixed notebooks. More specifically, RELANCER fixed 142 notebooks while RELANCER$_{github}$ and RELANCER$_{doc}$ fixed 96 and 92 notebooks, respectively.

There may be two reasons why RELANCER$_{doc}$ cannot fix a particular bug. First, an API can be completely removed from a library. Figure 8 presents such an example where function `plotly.plotly` is completely removed from *plotly version 4* distribution package. According to the official documentation, it has moved to *chart-studio* [69]. Therefore, the replacement is not simply available in the latest version of the documentation of *plotly*. Second, many APIs move to different modules or the name changes so significantly that it is hard to create a mapping between the old API and the new API by a particular heuristic. For example, in our motivating example in Figure 1, the module `sklearn.grid_search` is deprecated and replaced by `sklearn.model_selection` starting from scikit-learn 0.18 [52]. For all these cases, Github can help rank the correct program element toward the top.

Similarly, not all possible example changes for deprecated APIs with their parameters are available on Github. For example, `tensorflow.train.RMSPropOptimizer` is deprecated and used by a notebook in our dataset, but our on-demand search did not provide any useful candidate. This may often happen for less popular APIs.
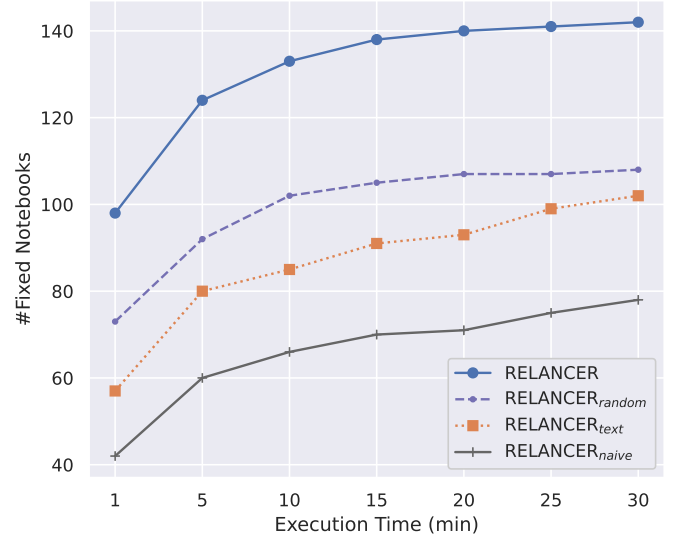

Fig. 9: Number of fixed notebooks over time.

In summary, all these results demonstrate that RELANCER cannot achieve its full potential without harnessing the information from multiple sources since each source plays an important role for a specific task.

> **Answer to RQ2:** Both sources of information contribute to the overall effectiveness of RELANCER. RELANCER can successfully restore the executability of 56% of the subjects, while baselines relying on just GitHub examples and just API documentation can only fix 38% and 36% of the subjects respectively.

C. RQ-3: Contribution of Machine Learning Models

RELANCER uses two machine learned models to explore the search space of all the candidate program elements efficiently. However, it may be plausible to think that two simple static models can be used instead. Therefore, to better understand the contribution of these two models on the end-to-end performance of RELANCER: we have created three baselines:

RELANCER$_{random}$. In this baseline, RELANCER ranks the repair actions randomly instead of using its repair action model.

RELANCER$_{text}$. In this baseline, RELANCER's learning-to-rank model is replaced by an edit distance based ranking. More specifically, we combine the candidate elements from both GitHub and API documentation, and sort them by the Damerau–Levenshtein Distance score [66] between the deprecated element and the new element in descending order.

RELANCER$_{naive}$. In this baseline, we replace both machine learned models by RELANCER$_{random}$ and RELANCER$_{text}$ to understand the effect of two models in aggregation.

Figure 9 presents the experimental results in terms of the number of notebooks fixed by RELANCER and the baselines over their execution time. From the results, it is evident that even we replace one machine learned model with a simple

baseline model, the performance of RELANCER deteriorates. More specifically, RELANCER$_{random}$, RELANCER$_{text}$, and RELANCER$_{naive}$ fix 108, 102, and 78 notebooks, respectively, which is less than the 142 fixed by RELANCER. This indicates that both machine learned models contribute to the overall performance of RELANCER. We performed an unpaired t-test between the result of RELANCER with that of each baseline separately. The p-values of RELANCER vs RELANCER$_{random}$, RELANCER vs RELANCER$_{text}$, and RELANCER vs RELANCER$_{naive}$ are 0.0014, 0.0002, and <0.0001, respectively. The test result indicates that the RELANCER's performance improvement over any baseline is not just numerically but also statistically significant.

If we reduce the timeout of RELANCER and other baselines, as expected the number of notebooks fixed by each technique reduces. However, due to its machine learning models, the performance of RELANCER deteriorates proportionally less than the other baselines. If we set a timeout of 5 minutes for all the tools, RELANCER can fix 124 (49%) notebooks of notebooks while the naïve baseline can fix only 60 (24%) notebooks.

> **Answer to RQ3:** The two machine learned models help RELANCER realize its real-time use case. RELANCER can fix 49% of subjects within 5 minutes, while a baseline lacking its machine learning models can only fix 24%.

D. Limitations and Threats to Validity

1) Limitations: RELANCER was unsuccessful in fixing 113 notebooks. To understand the reasons why RELANCER was unable to fix those notebooks, we manually investigated each of them and found one of the following reasons:

RELANCER is limited to fix only deprecation errors. We found that there are 55 notebooks that contain some other irrelevant errors such as missing dependent files, accessing incorrect columns in the dataset, and so on. Therefore, although RELANCER fixed the deprecation errors in these notebooks, finally the notebook still failed due to the other errors.

Lack of Type Inference. RELANCER determines the fully qualified name of an API based on a static analysis to upgrade it properly. However, since Python is a dynamically typed language, RELANCER cannot determine the type of some complex APIs, especially when it is dependent on the return type of another API. For example, the API: `as_matrix()` in Figure 10 has been deprecated [70]. To fix this API, RELANCER needs to know the fully qualified name: `pandas.DataFrame.as_matrix()` which should be determined by the returned type of `df.drop()`. Currently, RELANCER does not infer the return types of third-party APIs as the dynamic typing of Python makes it complex to retrieve accurate return-type information statically.

Lack of Complex Repair Actions. Currently, RELANCER's repair actions include replacing one API or variable name by another API or variable name respectively. However, some fixes require complex repair actions. For example, Figure 11 represents a deprecated API where the fix requires changing its

```
1  X = df.drop(['Radiation', ...], axis=1). as_matrix()
```

Fig. 10: A deprecation issue requires type inference to fix.

```
1  -scipy.misc.toimage(array)
2  +PIL.Image.fromarray(array.astype('uint8'))
```

Fig. 11: Example of a complex repair action.

parameter by another method call. Although RELANCER successfully changed the fully qualified name of the API, replacing the parameter `array` with the API `array.astype('uint8')` is currently out of its scope.

Change in Functionality. Figure 12 presents an example where due to a change in the functionality of the deprecated API: `tensorflow.placeholder`, the new API requires calling another API: `tf.compat.v1.disable_eager_execution()` with it. Currently RELANCER cannot infer such a change in functionality and thus is unable to fix this kind of errors.

2) External Validity: Our framework has only been instantiated for fixing API deprecation issues of Python-based Jupyter Notebooks, and has been only evaluated on 255 notebooks. Therefore, our results may not hold outside this scope. We tried to mitigate this risk by targeting real-world highly-voted datasets on Kaggle.

3) Quality of Data: The performance of RELANCER is limited by the quality of the training data. To mitigate this threat, we constructed the ground truth set from 12 most popular libraries and manually verified each deprecated API.

VI. RELATED WORK

Restoring the executability of Jupyter Notebooks. Recently, Wang et al. proposed Osiris [14], the first technique targeting unexecutable Jupyter Notebooks, specifically ones impacted by ambiguous or undefined order of cell execution. Osiris reconstructs possible execution orders by automatically satisfying dependencies between code cells. In follow up work they propose SnifferDog [15] that restores executability to a notebook by inferring and providing a compatible environment, comprised of correctly-versioned packages, for it. Our work complements [14] and [15] by remedying a third cause of notebook unexecutability, namely the deprecation of APIs. Moreover, in contrast to SnifferDog, which attempts to re-create the *original* execution environment of a notebook, RELANCER

```
1  -x=tf.placeholder(tf.float32,shape=[None,20],...)
2  +x=tf.compat.v1.disable_eager_execution()
3  +x=tf.compat.v1.placeholder(tf.float32,shape=[None,20],...)
```

Fig. 12: Example of functionality change.

ports it to the *user's* (and/or most recent) environment, in line with its target use-case of exploratory programming.

API Migration. Research on the broad topic of API migration goes back more than two decades [19], [20], [21], [22], [23], [24], [25], [26]. In early work by Chow et al. [19], a library maintainer is tasked with annotating API functions undergoing interface changes with migration rules. Catchup! [21] records API refactoring actions as a library maintainer evolves an API. The recorded migration rules [19] or API refactorings [21] are then used to migrate client applications. Such semi-automatic approaches entail additional effort by library maintainers, while RELANCER is completely automatic.

A second class of techniques automatically infer API mappings and/or migration refactorings. SemDiff [20] and LibSync [23] both mine API migration patterns across different versions of a library, such as renamed methods or changed parameters, from migrated client code. Zhong et al. [29] broaden the scope of API migration by inferring API mappings between related libraries in Java and C#. They do so by using textual similarity to align client code of the two libraries and then inferring corresponding APIs based on their common usage pattern in the aligned client code. Nguyen et al. [30], [31], [32] also infer API mappings between Java and C# libraries but use statistical machine translation instead. All the above techniques focus only on identifying API mappings. By contrast, RELANCER performs an end-to-end migration, including modifying the target code.

Recent work has proposed fully automated API migration techniques [27], [28], [18]. Meditor [27] targets automatic migration of a system from one library version to another. It mines migration-related code changes from open source repositories and instantiates them in the target system, in a context-sensitive manner to produce the migrated system, which is provided to the developer for review. Similarly, APPEVOLVE [28] automatically migrates Android apps by mining API changes from existing projects, applying them to the target app, and validating the patched app through differential testing. In emerging research, SOAR [18] proposes a technique to migrate data science programs from one library to another (e.g., TensorFlow [46] to PyTorch [71]). SOAR infers a likely API mapping from the documentation of the source and target libraries. Then it synthesizes candidates instances of the migrated program, using differential testing against the original (reference) program as an oracle, pruning the search space using logical constraints generated from the Python interpreter's error messages. Unlike the above, RELANCER organically integrates knowledge from two sources - migration examples and API documentation - to derive its search space. This feature is one of the keys to its performance (Section V-B). Further, APPEVOLVE and SOAR use differential testing against the reference implementation as a *strong* oracle to derive the correct migration. Meditor relies on human inspection for validation. In RELANCER's use case, there is no executable reference implementation. Thus, RELANCER uses a carefully orchestrated pair of machine learned models to effectively predict the correct migration.

Program Repair. Some elements of RELANCER's design are also inspired by the body of work on Automatic Program Repair (APR) [72], [73], [74], [75], [76], [77], [78], [79], [80]. APR techniques try to fix functional bugs in client code by implicitly or explicitly searching a space of program mutations for a patch, typically using a test suite as an oracle. At a high level RELANCER also searches a space of program modifications, but unlike APR techniques it needs to perform multiple API migrations (vs. fixing a single functional bug) and do so in real time, without the aid of a strong oracle like a test suite. Because of these key differences, RELANCER relies on machine learned models powered by multiple knowledge sources - runtime error messages, API migration examples on GitHub, and API documentation - as its primary vehicle for efficiently navigating the search space, rather than a test suite.

VII. CONCLUSION

Data scientists typically practice exploratory programming using computational notebooks, iteratively refining their code and actively trying to re-use solutions created by other data scientists. However, most publicly available notebooks cannot be executed. One of the prominent reasons is the deprecation of data science APIs. In this work, we proposed RELANCER, an automatic technique that restores the executability of broken Jupyter Notebooks, by upgrading deprecated APIs. RELANCER integrates an iterative runtime error driven approach with a combined search space sourced from API migration examples and API documentation. Both features are orchestrated through machine learned models. An evaluation of RELANCER on Kaggle notebooks showed that it is effective in restoring executability to 56% of the subjects, and that its error-driven approach, use of a combined search space, and its machine learned models all contribute to its efficacy.

VIII. ACKNOWLEDGMENTS

REFERENCES

[1] Wharton School of Business, University of Pennsylvania. (Accessed in 2021) What's driving the demand for data scientists? https://knowledge.wharton.upenn.edu/article/whats-driving-demand-data-scientist/.

[2] Forbes. (Accessed in 2021) Why data science is such a hot career right now. https://www.forbes.com/sites/quora/2017/10/25/why-data-science-is-such-a-hot-career-right-now/.

[3] Kaggle. (Accessed in 2021) Kaggle. https://www.kaggle.com.

[4] M. Beth Kery and B. A. Myers, "Exploring exploratory programming," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2017, pp. 25–29.

[5] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, "Managing messes in computational notebooks," in *Conference on Human Factors in Computing Systems*, 2019, pp. 1–12.

[6] Jupyter. (Accessed in 2021) Jupyter notebook. https://jupyter.org.

[7] J. M. Perkel, "Why jupyter is data scientists' computational notebook of choice," *Nature*, vol. 563, no. 7732, pp. 145–147, 2018.

[8] D. Toomey, *Jupyter for data science: Exploratory analysis, statistical modeling, machine learning, and data visualization with Jupyter*. Packt Publishing Ltd, 2017.

[9] J. Inc. (Accessed in 2021) We downloaded 10,000,000 jupyter notebooks from github – this is what we learned. https://blog.jetbrains.com/datalore/2020/12/17/we-downloaded-10-000-000-jupyter-notebooks-from-github-this-is-what-we-learned/.

[10] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, and J. development team, "Jupyter notebooks – a publishing format for reproducible computational workflows," in International Conference on Electronic Publishing, 2016, pp. 87–90.

[11] M. Ragan-Kelley, F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, and M. Bussonnier, "The jupyter/ipython architecture: a unified view of computational research, from interactive exploration to communication and publication," in AGU Fall Meeting Abstracts, 2014, pp. H44D–07.

[12] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," in International Working Conference on Mining Software Repositories, 2019, pp. 507–517.

[13] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's wrong with computational notebooks? pain points, needs, and design opportunities," in Conference on Human Factors in Computing Systems, 2020, pp. 1–12.

[14] J. Wang, K. Tzu-Yang, L. Li, and A. Zeller, "Assessing and restoring reproducibility of jupyter notebooks," in International Conference on Automated Software Engineering, 2020, pp. 138–149.

[15] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of jupyter notebooks," in International Conference on Software Engineering, 2021, pp. 138–149.

[16] J. Wang, L. Li, K. Liu, and H. Cai, "Exploring how deprecated python library apis are (not) handled," in International Symposium on Foundations of Software Engineering, 2020, pp. 233–244.

[17] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, and Y. Xiong, "How do python framework apis evolve? an exploratory study," in International Conference on Software Analysis, Evolution and Reengineering, 2020, pp. 81–92.

[18] A. Ni, D. Ramos, A. Yang, I. Lynce, V. Manquinho, R. Martins, and C. Le Goues, "Soar: a synthesis approach for data science api refactoring," in International Conference on Software Engineering, 2021, pp. 112–124.

[19] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in International Conference on Software Maintenance, 1996, p. 359.

[20] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," ACM Transactions on Software Engineering Methodology, vol. 20, no. 4, pp. 1–35, 2011.

[21] J. Henkel and A. Diwan, "Catchup! capturing and replaying refactorings to support api evolution," in International Conference on Software Engineering, 2005, pp. 274–283.

[22] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in International Conference on Software Engineering, 2012, pp. 353–363.

[23] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," ACM Sigplan Notices, vol. 45, no. 10, pp. 302–321, 2010.

[24] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in International Conference on Software Engineering, 2008, pp. 471–480.

[25] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in International Conference on Software Engineering, 2010, pp. 325–334.

[26] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," IEEE Transactions on Software Engineering, vol. 33, no. 12, pp. 818–836, 2007.

[27] S. Xu, Z. Dong, and N. Meng, "Meditor: inference and application of api migration edits," in International Conference on Program Comprehension, 2019, pp. 335–346.

[28] M. Fazzini, Q. Xin, and A. Orso, "Automated api-usage update for android apps," in International Symposium on Software Testing and Analysis, 2019, pp. 204–215.

[29] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in International Conference on Software Engineering, 2010, pp. 195–204.

[30] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining api usage mappings for code migration," in International Conference on Automated Software Engineering, 2014, pp. 457–468.

[31] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in International Symposium on Foundations of Software Engineering, 2013, pp. 651–654.

[32] ——, "Divide-and-conquer approach for multi-phase statistical migration for source code (t)," in International Conference on Automated Software Engineering, 2015, pp. 585–596.

[33] X. Liu, N. Iftikhar, and X. Xie, "Survey of real-time processing systems for big data," in International Database Engineering & Applications Symposium, 2014, pp. 356–361.

[34] J. Cohen. (Accessed in 2021) Near real-time vs. real-time analytics. https://harperdb.io/blog/near-real-time-vs-real-time-analytics/.

[35] M. Kaggle. (Accessed in 2021) Meta kaggle. https://www.kaggle.com/kaggle/meta-kaggle.

[36] Creative Research Systems. (Accessed in 2021) The survey system. https://www.surveysystem.com/sscalc.htm.

[37] Jupyter. (Accessed in 2021) nbconvert. https://github.com/jupyter/nbconvert.

[38] Anaconda. (Accessed in 2021) Anaconda. https://anaconda.org.

[39] scikit learn. (Accessed in 2021) scikit-learn: machine learning in python. https://scikit-learn.org.

[40] pandas. (Accessed in 2021) pandas – python data analysis library. https://pandas.pydata.org.

[41] seaborn. (Accessed in 2021) seaborn: statistical data visualization. https://seaborn.pydata.org.

[42] NumPy. (Accessed in 2021) Numpy. https://numpy.org.

[43] SciPy. (Accessed in 2021) Scipy. https://www.scipy.org.

[44] XGBoost. (Accessed in 2021) Xgboost: extreme gradient boosting. https://github.com/dmlc/xgboost.

[45] Plotly. (Accessed in 2021) Plotly: The front end for ml and data science models. https://plotly.com.

[46] TensorFlow. (Accessed in 2021) Tensorflow. https://www.tensorflow.org.

[47] Keras. (Accessed in 2021) Keras: the python deep learning api. https://keras.io.

[48] statsmodels. (Accessed in 2021) statsmodels. https://www.statsmodels.org.

[49] imbalanced learn. (Accessed in 2021) imbalanced-learn. https://github.com/scikit-learn-contrib/imbalanced-learn.

[50] CatBoost. (Accessed in 2021) Catboost - open-source gradient boosting library. https://catboost.ai.

[51] Kaggle. (Access in 2021) Predicting boston house prices. https://www.kaggle.com/sagarnildass/predicting-boston-house-prices.

[52] scikit learn. (Accessed in 2021) Scikit documentation. https://scikit-learn.org/stable/whats_new/v0.18.html.

[53] ——. (Accessed in 2021) sklearn.svm.linearsvc. https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html.

[54] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," in European Conference on Machine Learning, 1998, pp. 137–142.

[55] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," Journal of Big Data, vol. 6, no. 1, pp. 1–48, 2019.

[56] G. Ros, L. Sellart, J. Materzynska, D. Vazquez, and A. M. Lopez, "The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes," in Conference on Computer Vision and Pattern Recognition, 2016, pp. 3234–3243.

[57] A. Gaidon, Q. Wang, Y. Cabon, and E. Vig, "Virtual worlds as proxy for multi-object tracking analysis," in Conference on Computer Vision and Pattern Recognition, 2016, pp. 4340–4349.

[58] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, "Playing for data: Ground truth from computer games," in European Conference on Computer Vision, 2016, pp. 102–118.

[59] BeautifulSoup. (Accessed in 2021) Beautifulsoup. https://www.crummy.com/software/BeautifulSoup.

[60] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," Empirical Software Engineering, vol. 25, no. 3, pp. 1980–2024, 2020.

[61] GitHub. (Accessed in 2021) Github rest api. https://docs.github.com/en/rest.

[62] ——. (Accessed in 2021) Search. https://docs.github.com/en/rest/reference/search.

[63] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica, "Autopandas: neural-backed generators for program synthesis," in Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2019, pp. 1–27.

[64] X. Yang, K. Tang, and X. Yao, "A learning-to-rank approach to software defect prediction," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 234–246, 2014.

[65] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *International Symposium on Software Testing and Analysis*, 2016, pp. 177–188.

[66] Wikipedia. (Accessed in 2021) Damerau–levenshtein distance. https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance.

[67] LightGBM. (Accessed in 2021) Lightgbm. https://lightgbm.readthedocs.io/en/latest/.

[68] LibCST. (Accessed in 2021) Libcst. https://github.com/Instagram/LibCST.

[69] plotly. (Accessed in 2021) Version 4 migration guide in python. https://plotly.com/python/v4-migration.

[70] pandas. (Accessed in 2021) What's new in 1.0.0. https://pandas.pydata.org/docs/whatsnew/v1.0.0.html#deprecations.

[71] PyTorch. (Accessed in 2021) Pytorch. https://pytorch.org.

[72] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *International Conference on Software Engineering*, 2009, pp. 364–374.

[73] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *International Conference on Software Engineering*, 2013, pp. 802–811.

[74] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multi-line program patch synthesis via symbolic analysis," in *International Conference on Software Engineering*, 2016, pp. 691–701.

[75] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *International Conference on Automated Software Engineering*, 2017, pp. 648–659.

[76] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *International Conference on Software Engineering*, 2019, pp. 13–24.

[77] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *International Symposium on Foundations of Software Engineering*, 2019, pp. 613–624.

[78] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *International Symposium on Software Testing and Analysis*, 2020, pp. 101–114.

[79] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[80] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.