Automated Assertion Generation and Regression Testing for Machine Learning Notebooks

Yingao (Elaine) Yao
Cornell University
Ithaca, NY, USA
yy2282@cornell.edu

Vedant Nimje
Veermata Jijabai Technological Institute
Mumbai, India
vedantnimjed@gmail.com

Varun Viswanath
Dwarkadas J Sanghvi College of Engineering
Mumbai, India
varunvis2903@gmail.com

Saikat Dutta
Cornell University
Ithaca, NY, USA
saikatd@cornell.edu

Abstract

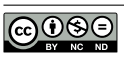
Jupyter Notebooks have become the de-facto choice for data scientists and machine learning (ML) engineers for prototyping and experimenting with ML pipelines. Notebooks provide a rich interactive interface with support for code, data, and visualization in one place. However, notebooks provide limited support for testing. As a result, during continuous development, many silent (non-crashing) regressions often go unnoticed and make notebooks unreliable and results hard to reproduce.

To enable more systematic testing of ML notebooks, we introduce **NBTestGen** – the *first* automated assertion generation approach for ML notebooks. NBTestGen generates regression-based assertions that check properties related to data processing, model building, and model evaluation steps in a typical ML notebook. To support systematic integration of such assertions in notebooks, we introduce the *first* regression testing framework (called **NBTest**) that can be used as a Jupyter plugin and allows developers to write *cell-scoped* assertions in notebooks. Three key features of such assertions are that they are 1) cell-scoped: they are linked to specific notebook cells and execute only after those cells are executed, 2) non-intrusive: they do not block notebook execution (in a Jupyter session), so that development can continue when they fail, and 3) they integrate with `pytest` and CI pipelines, allowing developers to easily do regression testing of their notebooks.

We evaluate NBTestGen on a corpus of 585 notebooks from the popular Kaggle platform. NBTestGen generates a total of 22065 assertions (37.72 on average per notebook). The generated assertions kill 72.21% of ML-specific mutations, while maintaining a high pass-rate of 100%. We also show that NBTestGen can detect 69.68% of historical regressions in 531 older versions of Kaggle notebooks. A popular ML library, SHAP, integrated NBTest into their CI. Further, we perform a user study with 17 ML developers that shows that such users find NBTest highly intuitive and useful.

CCS Concepts

• **Software and its engineering → Software verification and validation**.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
ASE '26, Munich, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834390

Keywords

Jupyter Notebooks, Regression Testing, Automated Assertion Generation, Machine Learning

ACM Reference Format:
Yingao (Elaine) Yao, Vedant Nimje, Varun Viswanath, and Saikat Dutta. 2026. Automated Assertion Generation and Regression Testing for Machine Learning Notebooks. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834390

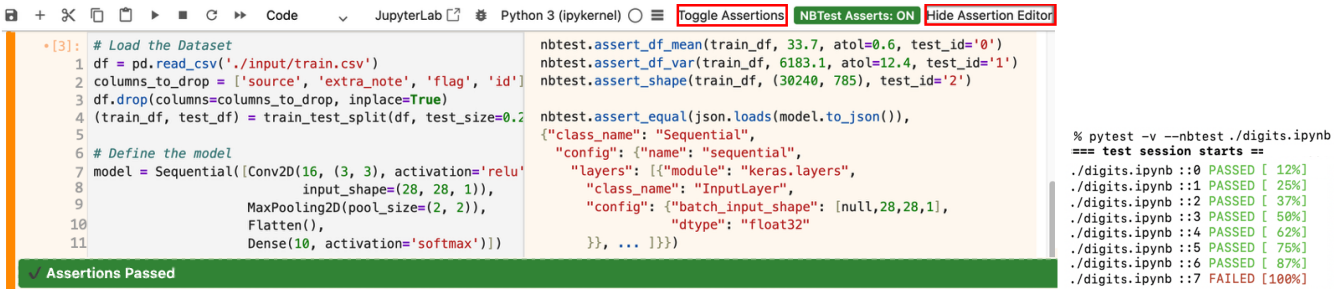
1 Introduction

Computational Notebooks have become the most popular medium for data scientists and machine learning engineers to author machine learning programs [65, 76]. Notebooks are popular because they are an interactive medium for users to write code and visualize results, such as training results, plots, and images – all in the same file. Notebooks are sequences of *cells*. Each cell contains a code snippet or a system command (e.g., to install a dependency) that can be executed independently of other cells and in arbitrary order. Such flexibility makes experimentation and exploration easier, especially for ML code. Some popular platforms that support notebooks are Jupyter [42], Kaggle [40], Pluto [69], and Google Colab [24].

Despite their popularity, notebooks are notorious for their poor reliability and reproducibility [6, 30]. Due to their out-of-order execution model and absence of tests, users often end up with buggy notebooks that become hard to reuse and reproduce results with [29, 68]. A recent study analyzed 1.4 million notebooks and reported that fewer than 25% notebooks could be executed without execution errors, while only 4% reproduced the expected results [68]. Hence, a *working* notebook may become unusable due to regressions caused by internal changes (such as code changes) or external changes (e.g., changes in dependencies or datasets) across versions. Further, regressions might manifest *within a session*, leading to silent (non-crashing) bugs (e.g., due to out-of-order execution or stale states [29]). Hence, like regular programs, notebooks also need to follow regression testing practices. Currently, for testing and debugging, notebook users follow ad hoc practices, such as using print statements, writing test cases within the notebook, using a separate test notebook for validation, or manually inspecting results [6, 30].

1.1 Our Work

Automated Assertion Generation. To address the reliability problem in ML notebooks and to assist developers in writing assertions, we propose an automated technique, **NBTestGen**, to generate assertions for machine learning (ML) notebooks. A recent study showed



(a) NBTest JupyterLab plugin (NBTest-lab-extension)

(b) Executing assertions with pytest

Figure 1: Example usage of NBTest. (a) shows NBTestGen assertions in a JupyterLab side panel, which can be hidden by *Hide Assertion Editor* button, or enabled/disabled with the *Toggle Assertions* button. The green message bar (*Assertions Passed*) indicates assertions passed. (b) shows pytest output for NBTest assertions.

that developers of ML notebooks often implement various integrity checks for data and ML model-related properties, such as data shape, model performance, and network architecture, using assertions and print statements [75]. Motivated by this finding, we design an assertion generation technique that first identifies model/data-related APIs of popular ML/Data libraries (e.g., PyTorch [61], TensorFlow [2], Scikit-Learn [63], and Pandas [60]) used in a notebook, instruments the API calls, executes the notebook to collect expected results, and generates appropriate assertions in the notebook. A key challenge of testing ML notebooks is that their execution can often be non-deterministic due to randomness during training or batching data – causing some assertions to be potentially *flaky* [14]. To mitigate this challenge, we execute the notebooks multiple times to collect execution information and estimate the variance in the results with statistical tests. We use the collected data to generate approximate assertions (e.g., `assert_allclose`) to account for this variance via tolerance bounds where needed [15].

Regression Testing Framework. To systematically integrate such assertions in a notebook environment, we propose a novel regression testing framework for notebooks: **NBTest**. The design of NBTest is guided by a recent study that found that notebook users often follow *cell-based risk management*, where users test changes in a new cell before merging with old cells to isolate any errors [30]. With NBTest, we introduce the idea that developers can write *cell-scoped* assertions to check the expected behavior of intermediate computations in a notebook, specifically within each cell. Cell-scoped assertions are linked to a specific cell and are executed only after the cell completes execution. For instance, a developer may want to check that the dataset was pre-processed correctly (e.g., no null values), if the model training was set up correctly (e.g., model architectures are defined as expected), or if the training accuracy was close to expectations. Hence, cell-scoped assertions can potentially improve the reliability of notebooks by mitigating regressions and nudging developers towards better testing practices in notebooks. The assertions in NBTest are designed to be *non-intrusive* – they neither pollute the notebook state nor block notebook execution when they fail. Non-intrusiveness allows developers to continue development without being interrupted by assertion failures.

NBTest-lib is a Python library in NBTest that provides 11 assertion APIs, covering two categories of assertions: (1) generic assertions (e.g., `assert_true(exp)` to check if `exp` is true and `assert_allclose(X, Y, atol, rtol)` to check if the values X and Y are equal up to the desired tolerance, (2) data-related assertions (e.g., `assert_df_mean(df, col, expected_mean)` to check if the mean of a column in a data frame is close to the expected mean). These APIs allow developers to easily write assertions for regression testing of data, models, and other computations in ML notebooks. NBTest integrates with `pytest` via the NBTest-lib plugin, which allows all developer-written tests to be directly run with `pytest`. This feature enables notebooks to be integrated with continuous integration (CI) pipelines.

Example. Figure 1 presents a simplified example Jupyter Notebook in JupyterLab with NBTest-lab-extension's panel (Figure 1a) and the output of running NBTest's assertions using `pytest` (Figure 1b) via NBTest-lib plugin. The notebook is training a convolutional neural network (CNN) model for a digit recognition task (identify the digit in an image). The left panel of Figure 1a shows one simplified cell of the notebook. The cell performs the following steps: 1) loads the digit recognition dataset (`train.csv`) into a pandas data frame called `df` (Line 1), 2) drops some unnecessary columns (e.g., `source`) from the data frame (Line 2-3), 3) splits the data frame into training and test sets using `train_test_split` in 80%-20% ratio (Line 4), and 4) defines a CNN model for training using tensorflow APIs for neural network layers (`Sequential`, `Conv2D`, `Dense`, etc) (Line 7-11). The rest of the notebook (not shown here) performs training of the model and computes accuracy of the model on the test set using Tensorflow APIs. Figure 1a right panel shows the NBTest-lab-extension's cell containing the assertions automatically generated by NBTestGen written using NBTest's APIs for this cell. NBTestGen automatically generates assertions for checking (1) data integrity properties (here: mean, variance, and shape of training data frame (`train_df`)), (2) model integrity properties (here: names and shapes of each intermediate layer in the CNN), and (3) model performance properties (e.g., training accuracy) – not shown here. Figure 1b shows the `pytest` output of running NBTest assertions, each assertion reported as either `PASSED` or `FAILED`. In this case, the output shows that all but the last assertion failed. Each assertion is labeled by a unique "test_id".

We envision that NBTest's cell-scoped assertions (automated and manual) can enable a lot of use cases for ML notebooks, such as documenting developer assumptions of expected behaviors, catching

regressions during interactive sessions, improving the reproducibility of notebooks, quantifying variance in training performance or intermediate results, and identifying contracts that can be used for monitoring of ML pipelines.

Results. We collect a corpus of 585 executable notebooks from popular competitions in Kaggle. NBTestGen generates a total of 22065 assertions (37.72 assertions per notebook on average), including assertions for dataset properties (18239), model architecture properties (861), and model performance properties (2438). We perform mutation analysis of the notebooks with assertions using ML-specific mutation operations, such as adding outliers in data and modifying layers of neural networks. Overall, we obtain a mutation score of 72.21%, showing that NBTestGen's generated assertions are robust to a variety of ML-specific mutations.

We also collect 531 older versions of Kaggle notebooks with potential regressions and evaluate whether our generated assertions could have caught these real-world regressions. Our generated assertions can successfully detect regressions in 370 (69.68%) buggy versions, showing the effectiveness of NBTestGen's assertions.

User Study and Adoption. We conducted a user study with 17 ML developers. Overall, participants found NBTest very intuitive (Rating 4.3/5), and useful in checking ML properties of interest (Rating 4.24/5), and reported that the show/hide toggle feature in JupyterLab significantly improved readability (Rating 4.7/5). Finally, NBTest has been adopted by SHAP [51, 52], a popular ML library (24.4k stars, 3.4k forks on GitHub) in their CI [45].

Contributions. Our key contributions in this work are as follows:

- ⋆ We propose NBTestGen, the first automated approach for generating assertions for machine learning notebooks; it accounts for non-determinism in ML pipelines using statistical methods.
- ⋆ We develop NBTest, the first regression testing framework for machine learning notebooks, integrated with `pytest` and Jupyter. NBTest allows developers to write non-intrusive cell-scoped assertions to check correctness of computations within each cell.
- ⋆ We evaluate NBTestGen on a corpus of 585 Kaggle notebooks and show that NBTestGen generates 22065 assertions that obtain a mutation score of 72.21% and could have caught 69.68% real-world regressions.
- ⋆ We perform a user study with 17 participants that shows that notebook users find NBTest intuitive and useful in writing assertions and testing notebooks. Our tool and results are available via GitHub [55] and Zenodo [56].

2 Methodology

2.1 Notebook Dataset Curation

We collect a corpus of runnable real-world notebooks from Kaggle. Kaggle is the world's leading platform for ML competitions and contains many high-quality, end-to-end ML pipelines in the form of notebooks. Hence, like prior studies on notebooks [88], we also collect ML related Python notebooks from Kaggle competitions. We select five popular Kaggle competitions [38] that are related to ML tasks: Titanic [41], Housing Prices [35], Spaceship Titanic [39], Jane Street [37], and Digit recognizer [34]. We use the Kaggle API [36] to download all notebooks related to these competitions. To avoid dependency issues, we filter out notebooks older than two years. We filter out notebooks with fewer than 10 votes to remove low-quality samples. After these steps, we obtain 1753 notebooks. We select notebooks that use at least one of the mainstream ML libraries: PyTorch, TensorFlow, Keras, or Scikit-learn, by checking their import statements. We identify 1612 such notebooks.

Environment Setup. We use the pipreqs [43] tool to generate the list of dependencies (a `requirements.txt` file) for each notebook. We then create a conda environment for each notebook, install the identified dependencies, and execute the notebook. Finally, we retain notebooks that execute without any errors. After this process, we obtain 585 Kaggle notebooks, out of which 529 use scikit-learn, 199 use TensorFlow/Keras, and 30 use PyTorch (non-exclusive).

Table 1: ML-related Mutation Operators

Mutation Operator	Description
Data-based Mutations	
Add Outliers [92]	Introduces extreme values in numeric columns by multiplying selected rows with large random values.
Repeat Data [53]	Randomly replaces a portion of rows with other existing rows in the dataset, simulating duplication noise.
Add Nulls [92]	Adds NaN/None entries in a subset of rows and columns to simulate missing values.
Modify Labels [53]	Introduces incorrect labels by randomly changing labels for selected samples.
Source Code-based Mutations	
Remove Zero Grad [44]	Removes `optimizer.zero_grad()`, allowing unwanted gradient accumulation across training steps.
Modify Hyper-parameters [92]	Changes critical hyperparameters like learning rate or dropout rate to harmful but syntactically valid values.
Remove Hyper-parameters [92]	Removes explicit hyperparameter values to fallback on library defaults, potentially reducing model quality.
Remove Layers [53]	Removes non-critical layers like ReLU or Dropout, potentially weakening feature learning or regularization.

2.2 Mutation Testing

To evaluate the fault-detection ability of our generated assertions, we develop eight mutation operators inspired by prior studies on common bugs found in ML and data science projects [44, 53, 86, 92]. We reviewed mutation operators proposed for ML by Ma et al. [53] and identified two main categories: data-related and model-definition-related operators. From these categories, we selected a representative subset that captures key mutation types, including data distribution anomalies, corrupted training data, inappropriate hyperparameters, and flawed model definitions. Our mutators subsume those from [53], while also extending them based on common ML development errors identified in [44, 86, 92].

We exclude certain mutation operators and error patterns found in [44, 53, 86, 92], that typically lead to crashes, such as variable redefinition and missing data. Our goal is to instead explore mutations that lead to silent errors, which enables us to evaluate whether NBTestGen assertions can detect hard-to-detect issues.

Table 1 presents all mutation operators that we support. We implement two categories of mutation operators: (1) *Data-based mutations* that mutate the dataset in the notebook, for example, by adding outliers or introducing spurious data correlations, and (2) *Source code-based mutations* that mutate the Python code in the notebook, for example, by modifying training hyper-parameters or removing neural network layers. Overall, we implement 4 data-based mutations and 4 source code-based mutations.

2.3 Collecting Kaggle Notebook Versions

To evaluate the effectiveness of our generated assertions in detecting real-world regressions, we need a dataset of Jupyter Notebooks that contain regression bugs. Unfortunately, such a dataset is not publicly available. Besides, unlike regular programs, developers do not frequently commit changes in notebooks on Github, and even if they do, the absence of tests often makes the regressions go unnoticed. So, instead we leverage Kaggle, where users often submit multiple public versions of notebooks to improve competition performance. Since the latest version is likely to be the most stable and well-tested, we hypothesize that the older versions may contain real-world silent regression bugs. Based on this hypothesis, we collect older notebook versions, then apply assertions generated from the latest version to each older version, to evaluate if these assertions can detect regressions.

Collection and Filtering. We develop a web scraping tool to extract all available versions of notebooks we consider in § 2.1. Out of the 2172 available versions, we do not consider versions with execution errors or large diffs (more than 500 diff lines) or duplicate versions. This leaves us with 531 notebook versions.

Assertion transfer. We consider an assertion transferable between versions if either (1) the statements it checks remain unchanged, or (2) the statements checked differs syntactically but is semantically equivalent, such as variable renaming or minor keyword change. For the first case, since NBTest inserts the assertion immediately after the statement, we can simply check whether the preceding statement is identical between versions. To decide about assertions in the second case, we require careful reasoning about program semantics to determine whether the assertion is still valid in the older version, despite syntactic differences. Given the strong capabilities of LLMs in code understanding, we leverage an LLM (Gemini 3.0 Flash) for this task.

Specifically, we provide the LLM with (1) the latest version with generated assertions, and (2) the older version with already transferred assertions in the first case, and ask the LLM to decide which remaining assertions are transferable (prompt in [56]). When an assertion is deemed transferable, the LLM inserts a lightweight macro with the assertion ID (later replaced with the full assertion) at the appropriate location, and outputs the modified old version. Then, we perform two validation checks. First, we execute the notebook to verify whether it does not have formatting issues. We found only six notebooks with missing closing brackets, which we corrected manually. Second, to detect hallucinations, we verify that all transferred versions are a subset of assertions to be transferred, then remove any unexpected assertions. This produces the final notebook with transferred assertions. While this process is not perfect, it allows us to easily transfer many assertions for evaluation. In a realistic setting, developers would directly modify the notebook with assertions, so the transfer process would not be needed.

2.4 Assertion-Statement Dependency Mapping

In the presence of mutation or regression in a notebook, a failing assertion may be caused either by the mutation itself, i.e., true positive (TP), or by unrelated factors such as randomness, i.e., false positive (FP). To distinguish FP from TP, we construct a mapping between each failed assertion and the set of preceding statements that can reach it via control or data flow. Then we check if the mutation or regression is within these statements. If so, we can classify the assertion failure as a TP, otherwise as an FP. To build such mapping, we develop a custom Python tool that extracts notebook AST, and performs standard data-flow analysis [3]. Similarly, we can classify a passing assertion as a true negative (TN) or false negative (FN).

2.5 Metrics

Passrate p. Given an assertion, let N be the total number of executions, and let P be the number of executions in which the assertion passes. The *passrate* p of the assertion is defined as: $p = \frac{P}{N}$.

Mutation Score m. Given an assertion, let M be the total number of generated mutants. For each mutant M_i ($i \in [1, M]$), let p_i be the passrate of the assertion on M_i. The *mutation score* m of the assertion is defined as: $m = \frac{1}{M}\sum_{i=1}^{M}(1 - p_i)$.

Number of Versions Killed K_V. Let $\mathcal{V}$ denote the set of versions of a notebook. For each version $v \in \mathcal{V}$, let A^v be the number of assertions, and let p_i be the passrate of assertion $i \in [1, A^v]$ on v. The *number of versions killed* K_V is defined as:
$K_V = \sum_{v \in \mathcal{V}} \mathbb{1}_{(\exists i \in [1, A^v], p_i < 1)}$, where $\mathbb{1}$ is the indicator function.

False Positive and Negative Ratios. Let *#FP* denote the number of failed assertions not caused by mutations or regressions, and *#Failed* the total number of failed assertions. Let *#FN* denote the number of passed assertions in the presence of mutations or regressions, and *#Passed* the total number of passed assertions. Both are determined by the assertion-statement dependency mapping in Section 2.4. The *FPR* and *FNR* are defined as: $FPR = \frac{\#FP}{\#Failed}$, $FNR = \frac{\#FN}{\#Passed}$.

3 NBTestGen: Automated Assertion Generation

There are two key challenges in automating the generation of assertions for ML Notebooks. First, there is a lack of clear criteria for selecting which program variables or computations to target when generating assertions in ML notebooks. In principle, one could add assertions for *every* intermediate result/variable in the notebook, which would provide comprehensive coverage. However, this brute-force strategy may introduce assertions that are not meaningful to the developer, cluttering the notebook interface, and reducing its readability. Conversely, a minimal approach that adds assertions only for the final outputs – such as checking the final evaluation metrics of the trained model – may preserve usability but sacrifice bug-detection capability. So, bugs that manifest in intermediate computations might be hard to detect or localize. Hence, an assertion generation approach must strike a balance between the fault detection capability, testing key properties of the ML pipeline, and preserving the notebook's usability.

Second, it is well-known that ML pipelines often exhibit nondeterminism due to various operations like randomized batching, random model initialization, and parallel execution on GPUs. Assertions generated without accounting for this randomness can lead to *flaky assertions*, where repeated executions of the same notebook yield inconsistent outcomes. Researchers have previously identified *flaky* behavior in unit tests in ML libraries [14, 15, 57] and ML training pipelines [67] showing that assertions must account for underlying nondeterminism.

Algorithm 1 NBTestGen's Assertion Generation

Input: $\mathcal{P}$: Jupyter Notebook, n: num. of runs, *APIs*: set of APIs to check for properties, c: confidence level
Output: $\mathcal{P}'$: Jupyter Notebook with generated assertions
1: $\mathcal{I} \leftarrow$ PropertyFinder($\mathcal{P}$)
2: $\mathcal{P}' \leftarrow$ AssertionGenerator($\mathcal{P}, \mathcal{I}, n, c$)
3: **procedure** PropertyFinder($\mathcal{P}$, *APIs*)
4: $AST_{\mathcal{P}} \leftarrow$ ParseAST($\mathcal{P}$)
5: $\{(v_1, l_1, \tau_1), \ldots, up\ to\ N\} \leftarrow$ FindAPI(*APIs*, $AST_{\mathcal{P}}$)
6: **return** $\{(v_1, l_1, \tau_1), \ldots, up\ to\ N\}$
7: **end procedure**
8: **procedure** AssertionGenerator($\mathcal{P}, \mathcal{I}, n, c$)
9: $\mathcal{P}_I \leftarrow$ Instrument($\mathcal{P}, \mathcal{I}$)
10: $\mathcal{D} \leftarrow$ RunCollect($\mathcal{P}_I, n$) ▷ $\mathcal{D}$ is a set of pairs (v_i, l_i, τ_i, s_i)
11: $\mathcal{P}' \leftarrow \mathcal{P}$
12: **for** $(v_i, l_i, \tau_i, s_i) \in \mathcal{D}$ **do**
13: $\epsilon_i \leftarrow$ ComputeTolerance(s_i, c)
14: $a_i \leftarrow$ GenerateAssertion($v_i, \tau_i, s_i, \epsilon_i$)
15: $\mathcal{P}' \leftarrow$ InsertAssertion($\mathcal{P}', a_i, l_i$)
16: **end for**
17: **return** $\mathcal{P}'$
18: **end procedure**

Our Approach. To solve the above two challenges, we propose NBTestGen's assertion generation algorithm (Algorithm 1). Algorithm 1 takes a notebook ($\mathcal{P}$), the number of runs (n) for tolerance calculation, a set of APIs to check for properties (*APIs*), and a confidence level (c) as input, and outputs a notebook with generated assertions ($\mathcal{P}'$). It consists of two stages: (1) PropertyFinder that identifies ML-specific properties in the notebook to target for assertion generation (Line 1), and (2) Assertion Generator that generates assertions for the identified properties (Line 2) while using statistical techniques to account for randomness.

(1) PropertyFinder. PropertyFinder extracts each notebook cell, analyzes its Abstract Syntax Tree (AST) (Line 4), passes the $AST_{\mathcal{P}}$ and the APIs identifying key ML-related properties to `FINDAPI` (Line 5), and outputs a set of (v_i, l_i, τ_i), where v_i is a variable in the notebook related to the property, l_i is the line number of v_i in the notebook, and τ_i is the type of the property.

A property represents the program computation related to some data or models. [1] Typically, ML notebooks contain various types of computations such as data pre-processing, model initialization and training, and model evaluation. So, for each type, we identified specific properties to check by understanding common data quality issues in machine learning reported by Polyzotis et al. [70], as well as common properties checked by developers via print statements and assertions [75]. We identified the three *types* of properties based on popular APIs used by ML Notebooks:

- **Dataset**: We consider APIs that initialize datasets from a CSV file (using `pandas.read_csv`), or from a dataset library (like `sklearn.datasets`), or by splitting a dataset such as `scikit-learn`'s `train_test_split`.
- **Model Architecture (Model Arch.)**: We consider APIs that initialize models using Scikit-Learn APIs (e.g., `LogisticRegression`) as well as Deep Neural Network models using PyTorch/TensorFlow APIs.
- **Model Performance (Model Perf.)**: We consider APIs that define metrics such as `accuracy_score` and `r2_score`, from `sklearn.metrics`. Sometimes, they also define custom metrics that build on top of common numerical APIs. So, we implement a lightweight static analysis that parses such expressions to identify custom metrics used in the notebook.

PropertyFinder collects statements that either call one of the above APIs, or appear in contexts where developers typically inspect intermediate computations, such as print statements or the last line in a cell [75]. The latter cases produce outputs in the cell. To capture variance in computations, we assign a new random seed to each notebook run. NBTestGen currently supports 100 APIs across four popular ML/Data libraries: Scikit-Learn, PyTorch, TensorFlow, and Pandas. NBTest can be easily extended to similar APIs by adding them to a configurable JSON file.

2) Assertion Generation. For detected properties in PropertyFinder, Assertion Generator first instruments the notebook (Line 9), executes the instrumented notebook for n iterations and collects the set of outputs (v_i, l_i, τ_i, s_i), where s_i is the set of n values collected for variable v_i across the n runs (Line 10). For each set of collected values s_i, Assertion Generator computes a tolerance bound ϵ_i (Line 13) based on the confidence level c. Based on the property type τ_i, the tolerance bound ϵ_i, the checked variable v_i and the corresponding collected values s_i, Assertion Generator generates an assertion (Line 14) to check the integrity of the property. Finally, it inserts the generated assertion in the original notebook (Line 15).

Specifically, Line 14 generates one or more assertions to check the *integrity* of each property as follows:

- **Dataset**: For each dataset variable, it inserts assertions to check properties such as the dataset's size, column names and types, and the mean and variance of numeric columns.
- **Model Arch.**: For model definitions, it inserts assertions to validate the architecture, including the number and types of layers, as well as associated hyperparameters.
- **Model Perf.**: It inserts assertions to check whether evaluation metric values fall within an expected range. Given the inherent randomness in ML pipelines, we use approximate assertions (e.g., `assert_all_close`) to capture expected variance in the results.

Assertion Bound Computation. Chebyshev's inequality [7] is a well-known technique in probability theory that guarantees that no more than a certain proportion of values will exceed a certain distance from the mean. Therefore, for any property with variance in outputs (such as Model Perf.), we use Chebyshev's Concentration Inequality to compute a tolerance bound for the assertion. Given the confidence level C, Equation 1 shows how to compute a value such that the probability of the property exceeding it at most $1-C$, where X is a scalar random variable with a finite mean μ and variance σ^2.

$$P_r(|X - \mu| \geq k\sigma) \leq 1 - C, (k > 0) \quad (1)$$

[1] The notion of property in NBTest is different from what is typically used in property-based testing. Our properties are regression-based (hence strongly tied to previous execution(s) and the notebook), whereas the ones used for property-based testing specify more general behavioral properties that may apply across different executions and notebooks.

Chebyshev's Inequality is a conservative choice when the underlying distribution of the data is unknown. If the distribution is known, more precise bounds can be computed [15].

Test Oracle. Because property-based oracles are scarce in the ML domain, in this work, we rely on *regression-based oracles* to check for correctness. Naturally, this approach assumes that the current version of the notebook is correct, which may not be true. However, regression-based oracles can help document the developer's assumptions, as well as capture the current state of the notebook, and their violation can indicate regressions or reveal incorrect assumptions in the future. Regression oracles have also been proposed in prior test generation techniques [23, 47, 50, 87]. Further, developers can also modify the generated assertions to use manual oracles.

4 Regression Testing Framework

We developed a regression testing framework, NBTest, for Jupyter Notebooks that consists of three components: (1) **NBTest-lib**: a Python library that provides APIs for writing cell-scoped assertions in Jupyter Notebooks. It works as a `pytest` plugin that collects these assertions like standard unit tests; (2) **NBTestGen**: a Python library that automatically generates cell-scoped assertions for a Jupyter Notebook by identifying commonly used ML-related APIs (implementing the technique discussed in § 3); and (3) **NBTest-lab-extension**: a JupyterLab plugin for running NBTest's assertions in Jupyter Notebooks. Users can install these three components independently via *pip*. We describe the design of NBTest-lib and NBTest-lab-extension in § 4.1 and § 4.2 respectively.

4.1 NBTest Python Library (NBTest-lib)

We support two categories of assertions, and 11 assertions in total.

(1) **Generic Assertions:** NBTest-lib provides standard assertions including `assert_equal`, `assert_allclose`, `assert_true`, and `assert_false`. These functions follow the familiar semantics of traditional unit testing frameworks but are tailored for execution within notebooks. Users can also use these assertions to check model architecture and model performance properties.

(2) **Data-related Assertions:** Because data processing is common in machine learning workflows, NBTest-lib includes specialized assertions for validating properties of data represented using `pandas` or `numpy`. The `assert_shape` function checks the expected shape of the dataset. The `assert_df_var` and `assert_df_mean` functions check the statistical properties of dataframes while handling missing values appropriately. The `assert_column_types` and `assert_column_names` assertions verify the structural integrity of data, ensuring that schema changes or data loading errors are detected early in the development process. The `assert_in` assertion verifies the presence of a column name in the dataset. The `assert_no_class_balance` assertion checks if the label distribution in the dataset is overly skewed towards one class.

We designed NBTest-lib's APIs to mirror common assertions such as Python's `assert` and Numpy's `assert_allclose`. Unlike those assertions, NBTest-lib's assertions are inactive during normal execution, to not interrupt development process. NBTest-lib integrates with `pytest` through a custom plugin. When invoked with the `–nbtest` flag, `pytest` parses the Jupyter Notebook, locates cells with NBTest assertions, executes them as tests. Using `pytest`'s collection hooks [71], each assertion is registered as an individual test with a unique identifier. NBTest executes all notebook cells regardless of execution failures, and then reports results.

4.2 JupyterLab Plugin (NBTest-lab-extension)

We developed a JupyterLab plugin (*NBTest-lab-extension*) that provides two key features: (1) toggling the visibility of assertions via the `Hide Assertion Editor` button (Figure 1a), improving readability by hiding assertions in a separate panel, and (2) enabling or disabling assertion execution via the `NBTest Asserts: ON` toggle. When enabled, assertions are executed *with the cell*, allowing developers to quickly validate local changes without exiting the notebook while not being interrupted by failures during normal development.

Comparison with Existing Tools. There are several existing tools for testing Jupyter Notebooks or ML pipelines. For instance, nbval [20] performs regression testing by doing a textual comparison of a notebook's current cell outputs against saved values from prior executions. Deepchecks [8] and Deequ [74] provide libraries of model and data integrity checks for ML pipelines, but they require developers to manually incorporate such checks. Testbook [58] allows developers to write unit tests in a separate file. However, none of these tools can (1) automatically generate ML-specific tests like NBTestGen (except nbval that can perform basic output checking), (2) account for non-determinism like NBTestGen, or (3) allow non-intrusive and cell-scoped testing within a Jupyter session like NBTest-lab-extension. We provide a comparison with nbval in § 5.4. In the future, we plan to extend NBTestGen to generate assertions using Deepchecks' and Deequ's APIs.

5 Evaluation

We answer the following research questions in this work:

- **RQ1**: How many assertions does NBTestGen generate? How does their passing rate change with different confidence thresholds?
- **RQ2**: How effective are the generated assertions in detecting domain-specific mutations?
- **RQ3**: How effective are the generated assertions in detecting real-world regressions in ML notebooks?
- **RQ4**: How does NBTestGen compare with nbval?

Experimental Setup. We run experiments on a large cluster of diverse machines with GPUs and Ubuntu 20.04 OS. To run each notebook, we set up a Conda environment with Python 3.9 and install the dependencies for the notebook. We run NBTestGen with 30 iterations and use the Chebyshev's statistical method with the confidence intervals of 0.5, 0.7, 0.99, and 0.999 during assertion generation for RQs 1-3. For mutation analysis, we need to balance running many notebooks and reliably estimating the mutation score. Hence, similar to prior work [14], we run each notebook 30 times to compute the mutation score.

Implementation Details. NBTestGen uses NBFormat [33] to parse the notebooks and Python's AST library to parse Python code in each notebook cell. Specifically, NBTestGen uses AST's [72] visitor design pattern API to detect the variables related to the ML-specific properties, and the transformer design pattern API to insert the generated assertions back into the original notebook.

Table 2: #Assertions generated per notebook by NBTestGen

Type	Total #Assertions	#Assertions/Notebook
Dataset	18239	31.18
Model Perf.	2438	4.17
Model Arch.	861	1.47
Total	22065	37.72

Table 3: Distribution of Passrates

Tool	0%	$(0-50]$%	$(50-100)$%	100%	Total (Overall)
NBTest	0	0	16	21919	22065 (100%)
nbval	1655	400	113	12164	14332 (86.07%)

NBTest-lab-extension is implemented as a JupyterLab extension in `TypeScript`. It registers two commands and integrates them into the notebook toolbar. The `Toggle Assertions` command controls assertion execution. It sends a `Python` snippet to the notebook's running kernel that flips an environment variable and updates the `NBTest Asserts: ON/OFF` toolbar indicator accordingly. When the variable is enabled, NBTest assertions are executed after the cells containing them are executed. The `Hide Assertion Editor` command toggles assertion visibility by moving NBTest assertions between a cell's source and its metadata. Assertions stored in the metadata are not executed. The total lines of code in NBTest is 2877.

5.1 RQ1: Passrates of NBTestGen's Assertions

Table 2 presents the statistics of the assertions generated by NBTestGen for 585 Kaggle notebooks. Out of 585, NBTestGen generates at least one assertion (22065 in total) for 535 notebooks. The remaining 66 notebooks did not use any of the ML APIs that NBTestGen tracks (as described in 3). As a result, NBTestGen was unable to generate any assertions for them. On average, NBTestGen generates 37.72 assertions per notebook, showing that NBTestGen's approach of leveraging ML APIs to track relevant properties allows it to generate many assertions for a majority of ML Notebooks.

NBTestGen generates the highest number of assertions for datasets (18239). This is because, for each dataframe (a dataset object loaded using `pandas.read_csv`) used in a notebook, NBTestGen will generate five assertions that check the column names, column types, mean and variance of numeric columns, as well as the shape of the dataframe. NBTestGen generates 2438 Model Perf. assertions (4.17 on Avg.). Most notebooks consist of (1) one or more metrics of model performance, such as model accuracy or precision, and (2) a prediction dataframe that contains predictions of the trained model on test sets. In the latter case, similar to dataset dataframes, NBTestGen generates multiple assertions for the prediction dataframe, which are considered as Model Perf.

The average time NBTestGen takes for generating the assertions per notebook is 1.57 hours when using 30 runs per notebook. NBTestGen runtime is proportional to the notebook execution time and the number of runs. However, like other test generation tools, we expect NBTestGen to be used offline. The generated assertions perform simple property checks, hence they do not add any significant overhead to notebook execution time. Table 3 shows the distribution of passrates of assertions generated by NBTestGen and nbval (we compare them in § 5.4).

Table 4: Quality of assertions generated by NBTestGen

Threshold	passrate	mutation score	Est. FPR	Est. FNR
0.5	96.93%	0.82	0.1342	0.18
0.7	98.58%	0.78	0.0712	0.22
0.99	100%	0.72	0.0052	0.28
0.999	100.00%	0.71	0.0042	0.29

We observe that the passrate of all the assertions is 100%, showing that most assertions generated by NBTestGen have high reliability. Out of these, 21919 assertions always pass. 16 assertions have a passrate between 50% and 100%. These 16 assertions, spread across 8 notebooks, are of type `assert_allclose` that check model performance. Due to the small number of runs (30), in some cases, NBTest under-approximates the variance in the results and produces overly strict bounds. We inspect these 16 notebooks and find that in 10 of them, the assertion bounds were overly strict. When rerun with 50 iterations at a 0.999 confidence level and validated with 30 `pytest` runs, all assertions in these notebooks passed. The remaining 6 notebooks failed due to non-deterministic assertions on training history values, where even very loose bounds could not ensure consistency. The checked variables exhibited highly irregular distributions, so even increasing the confidence level to 0.999 with 50 iterations did not help.

Assertion Quality vs Confidence Thresholds. Table 4 shows how the quality of the generated assertions, specifically, the passrate, mutation score, estimated false positive rate (FPR), estimated false negative rate (FNR), changes with different confidence thresholds used during assertion generation. Because the static analysis for Python is imprecise, the reported FPR only estimates the true FPR. Our manual inspection of a sample of the reported FPs revealed that many are in fact true positives, indicating that the reported FPR overestimates the true one. Overall, when the confidence threshold increases from 0.5 to 0.999, the passrate increases from 96.93% to 100.00%, as expected. However, the mutation score drops significantly from 0.82 to 0.71, along with the FPR. Because Chebyshev's method is very conservative, the passrates are already very high even at 0.5.An ideal confidence threshold should result in high passrate, high mutation score, low FPR and low FNR. Both thresholds 0.99 and 0.999 have very high passrates. However, the mutation score at 0.99 is slightly higher than at 0.999. Hence, we recommend using a confidence threshold of 0.99 for generating assertions.

5.2 RQ2: Mutation Analysis of NBTestGen's Assertions

We apply the mutations described in § 2.2 to all the notebooks. This process generates many *mutant notebooks*, each with a single mutation, which we simply refer to as mutants hereon. Some mutations are randomized (e.g., Add Outliers) and/or generate multiple mutants for the same notebook (e.g., Modify Hyper-parameters). Hence, we apply such mutations multiple times and generate a maximum of 4 mutants per mutation per notebook where applicable. We run each mutant 30 times and compute the ratio of runs in which the mutant is killed by an assertion.

Table 5 presents the results. Each row presents the mutation score for one mutation. A mutant is *killed* if at least one assertion fails. The first four rows represent Data-based mutations, the next four represent Source Code-based mutations.

Table 5: Distribution of Mutation Scores

Mut.\Assert.	Dataset	Model Perf.	Model Arch.	Mut. Gen.	Mut. Killed	Mut. Score
Add Outliers	652.47	80.67	0.00	743	655.47	0.88
Repeat Data	761.03	39.67	0.00	934	764.17	0.82
Add Nulls	708.37	38.80	0.00	817	709.37	0.87
Modify Labels	134.80	104.17	0.00	207	181.87	0.88
Data-based Total/Mutation score	2256.67 (0.84)	263.30 (0.10)	0.00 (0.00)	2701	2310.87	0.86
Remove Zero Grad	0.00	8.83	0.00	19	7.90	0.42
Modify Hyper-parameters	0.00	533.57	563.97	1220	742.40	0.61
Remove Hyper-parameters	0.00	263.50	377.23	845	443.17	0.52
Remove Layers	0.00	16.67	32.00	132	46.33	0.35
Code-based Total/Mutation score	0.00 (0.00)	822.57 (0.37)	973.20 (0.44)	2216	1239.80	0.56
Total (All Mutations)	2256.67	1085.87	973.20	4917	3550.67	0.72

General Trends. Overall, the assertions generated by NBTestGen obtain a mutation score of 0.72 (or 72.21%) across all mutations. We generate a total of 2701 dataset-related mutants. For these killed mutants, they are either killed by Dataset or Model Perf. assertions. Because ML pipelines are often resilient to small data errors, we observe that corrupting the dataset does not always cause an observable degradation of the model performance. So, Model Perf. assertions only have a mutation score of 0.10 for data-based mutations. In contrast, the Dataset assertions obtain a mutation score of 0.84. They are generally stricter, e.g., they check if the mean and variance of data columns are close to expected values. Hence, they can detect data-based mutations more accurately and earlier in the ML pipeline. Model Arch. assertions are unable to detect such mutations because they only check for the integrity of the ML model architecture and its parameters.

We generate 2216 code-related mutations that either change the training hyperparameters, or the model architecture (e.g., Remove Layers). The overall mutation score is 0.56. Expectedly, dataset assertions fail to detect most of these because these mutations typically occur later in the pipeline. These mutations are mostly detected by Model Arch. assertions (mutation score 0.44) or Model Perf. assertions (mutation score 0.37). The Model Arch. assertions are stricter in general, so they can detect bugs that affect the model architecture easily.

Trends across Data-based mutants. We observe that almost 10.85% of Add Outliers, 4.75% of Add Nulls mutants, and 4.25% of Repeat Data are killed by Model Perf. assertions. In such cases, the model performance changes significantly. For example, for one notebook [22] the accuracy dropped to 0.197 from 0.63 after applying the Add Outliers mutation. In another example [19], the Add Outliers caused the accuracy to increase to 0.982.

Trends across Code-based mutants. Some mutations like Remove Zero Grad only impact the training process by affecting how model gradients are updated across training iterations. Hence, they often cause non-deterministic assertion failures, where the model performance changes drastically but only in a fraction of the cases. For Modify Hyper-parameters mutants, around 46.23% of them are killed by Model Arch., and around 43.73% of them are killed by Model Perf. Since most of the Modify Hyper-parameters or Remove Hyper-parameters mutants modify the hyperparameters in the model definition (such as choice of activation layer), the regression is detected by the Model Arch. easily. In some other cases, a change in hyperparameters also causes drastic changes in model performance, which is detected by Model Perf. assertions. Among the surviving mutants, the major reason is typically when the hyperparameter change does not alter the training results beyond the expected range in Model Perf. assertions. Because our mutations are applied statically, there is no guarantee that these will alter the pipeline behavior. An interesting future work would be to design mutations that leverage dynamic information to generate behavior-modifying mutants.

Table 6: Regression Testing of Kaggle versions

Assert Type	Dataset	Model Arch.	Model Perf.	Total
Versions killed	277 (52.17%)	116 (21.85%)	119 (22.41%)	370 (69.68%)

Surviving mutants. We manually inspect 70 surviving mutants. We find that 29 mutants survived because the involved APIs are not tracked by NBTestGen– these can potentially be killed by improving API coverage. The remaining 41 are equivalent mutants (6 data-based, 35 code-based) that either do not semantically modify the pipeline or do not create an observable difference during the execution. Among the data-based equivalent mutants, three mutants targeted dataframes with fewer than ten columns, so 10% selection yielded no columns to mutate. Three mutants were applied to empty dataframes and two mutants introduced anomolies which later removed by preprocessing. Among the code-based equivalent mutants, 28 Modify Hyper-parameters/Remove Hyper-parameters mutants only changed the random seed. Three Remove Hyper-parameters removed explicit hyperparameters already set to their API defaults (e.g., `learning_rate=0.1`), and four Modify Hyper-parameters mutants merely changed the verbosity or formatting. The presence of equivalent mutants can underestimate NBTestGen's true mutation score, but computing the true score is generally undecidable [28]. Hence, we leave this for future work.

5.3 RQ3: Detecting Real-World Regressions with NBTestGen's Assertions

Using the methodology described in § 2.3, we collect 531 versions of Kaggle notebooks, with 90 notebooks (out of 585). For these remaining notebook versions, we generate assertions in the latest versions using NBTestGen and transfer the assertions to the older versions using the methodology described in § 2.3. We successfully injected 94.14% of the original assertions into 531 versions, with 39.12 assertions per version on average.

Table 6 presents how many of these versions (which denote regressions) can be *killed*/detected by NBTestGen's assertions. Each cell denotes the number of versions were killed by each type of assertion. Overall, 370 versions (out of 531, 69.68%) are successfully killed by our assertions, showing that our automatically generated assertions are effective at detecting many real-world regressions. Overall, we observe a relatively low false positive rate of 0.0847. However, our assertions also missed many regressions, with a false negative rate of 30.32% (i.e., 161 versions).

Among versions that were killed by our assertions, the changes typically involved dataset pre-processing modifications (e.g., dropping different dataset columns [4]) and model architecture changes

Table 7: NBTestGen vs. nbval: Assertion Statistics and Quality

Tool	# Assertions	#Assertions/Notebook	passrate	mutation score
nbval	14332	24.50	86.07%	0.96
NBTest (0.99)	22065	37.72	100%	0.72

(e.g., changing the number of estimators in scikit-learn's Random Forest Classifier [1]). These are typically detected via our Dataset and Model Arch. assertions, respectively. Often, such changes also significantly affect the performance of the ML pipeline, which is then detected by our Model Perf. assertions. For example, a notebook version [80] was killed by all three kinds of assertions. Most missed regressions fall into one of the two categories: (1) the changes made did not modify the program's semantics, and (2) significant changes to data pre-processing or model architecture hindered the automatic transfer of original assertions.

5.4 RQ4: NBTestGen vs. Nbval

To compare nbval with NBTestGen, we use the same experimental step and collect the same metrics as in RQ1 for nbval. Table 7 presents the results. Overall, we observe that NBTestGen generates 7733 more assertions than nbval (53.96% greater). On average, NBTestGen generates 13.22 more (or 53.96% more) assertions than nbval per notebook. More importantly, the passrate of nbval is 13.93% lower than that of NBTestGen, indicating that NBTestGen produces more reliable assertions. While nbval seems to achieve a higher mutation score, on a closer inspection, we find that most of the failures are unrelated to the mutants, meaning these assertion failures are false positives. Because nbval performs strict output comparisons, it often make brittle checks that break due to differences in plot metadata, low-level logging data, which are not meaningful to developers. This can be further confirmed by the distribution of passrate of nbval's assertions in Table 3. We find 0.00% assertions fail in all runs, even though the code remains unchanged.

While we computed FPR and FNR for NBTestGen, we cannot compute them for nbval because nbval asserts on cell outputs instead of specific variables and in general it is hard to determine which statements are producing outputs. So, we manually inspect 20 notebooks to understand why nbval's assertions fail consistently. We find three main sources of failures: (1) notebook cells often contain model outputs, such as prediction results or training loss, which are typically non-deterministic and exact matches almost always fail, (2) comparing plots almost always fails because of differences in plot metadata, and (3) comparing log data (e.g., timestamps, training logs) that also cannot be compared across runs. Beyond these cases, nbval often checks outputs that are not meaningful and more importantly, not useful for debugging. In contrast, NBTestGen targets important ML-related properties and also accounts for non-determinism, thereby avoiding such problematic scenarios and producing higher quality assertions.

6 User Study and Real-World Adoption

The goal of our user study was to understand: 1) if developers of ML notebooks find NBTest's Python library intuitive and convenient to use in their daily workflows, and 2) if NBTestGen's generated assertions are useful compared to manually written assertions. To fulfill this goal, we recruited 17 developers who have prior experience with writing ML code in Jupyter notebooks.

Recruitment. We recruited 17 participants (15 graduate students, 2 software engineers) through online advertisements on social media and word-of-mouth referrals. Overall, three participants have over five years of experience in machine learning, and four have more than five years of experience using Jupyter notebooks for ML-related tasks. None of the participants is an author of this paper or contributed to this paper in any other way.

Study Design. For each participant, we send them a document containing the setup instructions, a tutorial of NBTest's different components and examples of API usage, the assertion generator, and the JupyterLab plugin. We design four tasks. In the first task, we provide participants with a notebook and ask them to manually write assertions using NBTest APIs for three cases to check for data integrity, model integrity, and model performance. In the second task, we ask participants to evaluate NBTestGen's generated assertions and compare them with their manual assertions. In the third task, we inject a bug in the notebook and ask participants to detect this bug using NBTestGen's generated assertions. The third task demonstrates that a bug can fail silently and remain undetected during typical execution, while NBTestGen's assertions can detect it effectively. In the fourth task, we ask participants to use NBTest-lab-extension's hide/show feature.

After completing the tasks, we ask participants to fill out a survey with 7 questions: (1) time taken to write assertions (in minutes with a timer); (2) ease of writing assertions using NBTest's APIs (1 = very difficult, 5 = very easy); (3) whether NBTest's cell-scoped assertions better capture developer assumptions compared to notebooks without them (Yes/No); (4) usefulness of automatically generated assertions (1 = not useful, 5 = very useful); (5) willingness to integrate NBTest into their Jupyter Notebook workflow (Yes/No); and (6) intuitiveness of using NBTest with Pytest (1 = not intuitive, 5 = very intuitive). (7) Readability of the Jupyter Notebook with hide/show assertions feature (1 = poor, 5 = very good).

Study Results. On average, participants spent 7.9 minutes manually writing the assertions. The average rating for manually writing assertions using NBTest's APIs was 3.94. This suggests that while NBTest's APIs are useful, coming up with assertions manually might be challenging for users. All participants agreed that the notebook with cell-scoped assertions provided a better sense of developer assumptions in the notebook than without them. Participants found NBTestGen's generated assertions very useful (avg. rating 4.24, minimum 3). 16 participants said that they would like to integrate NBTest into their workflow. Finally, the participants found NBTest's interface very intuitive (avg. score 4.3). The average readability score was 4.7, suggesting that the hide/show feature is highly effective in improving the readability of NBTest's assertions. Overall, we conclude that notebook users found NBTest's interface intuitive and its assertions useful.

Real-world Adoption. We reached out to developers of open-source machine learning (ML) projects, and the initial responses were encouraging. We submitted a pull request (PR) [45] to integrate NBTest into the continuous integration (CI) pipeline of SHAP [51, 52], a popular open-source ML project (24.4k stars). The developers have merged our PR and integrated NBTest into SHAP's CI pipeline. We also have an ongoing PR [89] for fg-data-profiling [9], a popular open-source ML project (13k stars). The

developer labeled our issue with the *code quality* tag and commented that they "tag it as part of the roadmap so the community is aware that this is something we are keen to integrate" [21]. Developers of NVIDIA-BioNeMo library [5] (820 stars) have also showed interest in integrating NBTest into their CI [59]. These positive responses demonstrate the promise of NBTest in the real world.

7 Discussion

Limitations. Our assertion generation relies on regression oracles, hence, they are unaware of the expected correct outputs. In the future, we plan to integrate property-based oracles for data and training validations. Also, while the assertion generation only runs once offline, which is realistic for many workflows, the overhead of the bound computation is still high for long-running notebooks. To reduce the overhead of long-running notebooks and updating the assertions after the notebook evolves, we plan to extend NBTest to support: *(1) cost amortization via CI*: collecting multiple samples across CI runs of the notebook and using such samples to estimate bounds, *(2) evolution-awareness*: obtaining the backward and forward slice of changed cells and only re-running those cells to update assertions, *(3) API-specific variance models*: learning expected output variance from historical notebook runs for each API.

Threats to Validity. *(1) Internal:* The internal threat comes from the implementation of NBTest. To address this threat, multiple co-authors carefully performed testing and code reviews to check that it was correctly implemented. *(2) External:* Since our findings are based on ML-focused notebooks, they may not generalize to non-ML workflows, however, this is beyond the scope of this work. Also, the studied notebooks might be limited to the covered ML domains and frameworks. To mitigate this problem, we selected five diverse and representative domains, i.e., tabular classification, time series forecasting, computer vision, and deep learning, which is broader than prior works [12, 88]. *(3) Construct:* Mutants may not be representative of real bugs. To mitigate this, we choose historical versions of Kaggle notebooks, which also showed high mutation scores. To mitigate risk due to non-determinism, we run the notebooks multiple times and use statistical methods to compute metrics.

8 Related Work

Dynamic analysis of Jupyter Notebooks. Many prior techniques improve the executability and reproducibility of Jupyter Notebooks. Sniffer dog [82] automatically infers correct library dependencies for Jupyter Notebooks. Osiris [81] identifies possibly correct execution orders of Jupyter Notebook cells that reproduce the correct results. NBSAFETY [54] combines dynamic and static analyses to identify unsafe cell executions, such as out-of-order executions. NBTest plays a complementary role to these tools by automatically generating assertions that can detect regression bugs.

Empirical analysis of notebooks. Robinson et al. [73] studied how student users debug various issues in notebooks. Chattopadhyay et al. [6] interviewed various notebook users and identified several pain points, which included difficulty in debugging and reproducibility of results. De Santana et al. [10] analyzed numerous GitHub commits/StackOverflow posts to identify common sources of bugs in Jupyter Notebooks. NBTestGen's generated assertions can potentially improve the quality and debuggability of notebooks.

Bug Detection for Notebooks. Prior work has targeted specific types of bugs in Jupyter Notebooks, such as name-value inconsistencies [62], data leakage [12, 77, 88], and syntax or dependency errors [25, 26]. In contrast, NBTestGen uses a regression-based oracle, but has the potential to detect a broader range of bugs, as shown by our mutation analysis and experiments with Kaggle versions.

Assertion/Test Generation. Many ML-based techniques have been proposed for generating assertions for unit tests [11, 31, 78, 83, 85]. However, these techniques focus on simpler assertions for Java. Tools such as Pynguin [50] and DynaPyt [18] are general test generation tools for Python, but they are not specialized for ML programs or notebooks. In contrast, NBTestGen automatically generates assertions to detect regression bugs in notebooks and accounts for ML randomness.

Inline Tests. More recently, Liu et al. [47, 48] introduced a new testing paradigm: *inline testing* that executes tests at a lower level of granularity (statement level) than unit tests. Inline tests only target one statement at a time and test general PL features (e.g., regexes, string manipulation). In contrast, NBTestGen's assertions check for the correctness of logic *within a cell*, they lie in between inline tests and unit tests in granularity and target ML-related properties. Also, inline tests require developer inputs/oracles (NBTestGen's automated assertions do not) and are enforced at test-time (NBTestGen's assertions are enforced at both production-time and test-time).

Testing Non-deterministic Systems. Prior work has developed techniques to test non-deterministic systems such as ML models [64, 90], ML frameworks [46, 66, 84, 91], probabilistic programming systems [13, 16, 49], and randomized algorithms [32, 79]. Most focus on detecting bugs in the learning system or library itself. In contrast, NBTest focuses on detecting regressions in the ML pipeline. Among these, Turcotte et al. [79] is closest to our work as it tests statistical assumptions in ML pipelines, but requires manual annotations. There is also work on detecting flaky tests in non-deterministic systems [14, 15, 17, 27]. In contrast, NBTest focuses on test generation while accounting for non-determinism.

9 Conclusion

We presented NBTestGen, an automated assertion generation approach (and NBTest framework) for generating cell-scoped and non-intrusive assertions for Jupyter Notebooks. We showed that NBTestGen is effective at detecting mutations and real-world regressions. NBTest has also been adopted by SHAP, a popular ML library, and integrated into its CI pipeline, showing its promise for real-world adoption. We envision that NBTest will encourage developers to adopt test-driven development of ML Notebooks and researchers to expand NBTest to detect other classes of bugs.

Data Availability. We share our artifacts and data here [55, 56].

Acknowledgements

We thank Owolabi Legunsen, Kevin Guan, Pengyue Jiang, Shinhae Kim and the anonymous reviewers for their feedback. This work is partially supported by Google Research Award, Meta LLM Evaluation Research Grant, and Google Gemini Cloud Credits.

References

[1] Aaron. 2025. Spaceship Titanic. https://www.kaggle.com/code/aaron95629/spaceship-titanic Accessed: May 28, 2025.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[3] Frances E. Allen and John Cocke. 1976. A program data flow analysis procedure. *Commun. ACM* 19, 3 (1976), 137–147.

[4] Ankit Bhardwaj. 2025. House Prices | HistGradient Boosting. https://www.kaggle.com/code/ankit1743/house-prices-histgradient-boosting Accessed: May 28, 2025.

[5] NVIDIA BioNeMo. 2026. BioNeMo Recipes. https://github.com/NVIDIA-BioNeMo/bionemo-recipes Version 3.0.0.

[6] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. 2020. What's wrong with computational notebooks? Pain points, needs, and design opportunities. In *Proceedings of the 2020 CHI conference on human factors in computing systems*.

[7] Pafnutii Lvovich Chebyshev. 1867. Des valeurs moyennes. *J. Math. Pures Appl* 12, 2 (1867), 177–184.

[8] Shir Chorev, Philip Tannor, Dan Ben Israel, Noam Bressler, Itay Gabbay, Nir Hutnik, Jonatan Liberman, Matan Perlmutter, Yurii Romanyshyn, and Lior Rokach. 2022. Deepchecks: A library for testing and validating machine learning models and data. *Journal of Machine Learning Research* 23, 285 (2022).

[9] Data-Centric AI Community. 2026. fg-data-profiling. https://github.com/Data-Centric-AI-Community/fg-data-profiling Version 4.19.1.

[10] Taijara Loiola De Santana, Paulo Anselmo Da Mota Silveira Neto, Eduardo Santana De Almeida, and Iftekhar Ahmed. 2024. Bug Analysis in Jupyter Notebook Projects: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 33, 4, Article 101 (April 2024), 34 pages. https://doi.org/10.1145/3641539

[11] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*.

[12] Filip Drobnjaković, Pavle Subotić, and Caterina Urban. 2024. An Abstract Interpretation-Based Data Leakage Static Analysis. In *International Symposium on Theoretical Aspects of Software Engineering*. Springer.

[13] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 574–586.

[14] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*.

[15] Saikat Dutta, August Shi, and Sasa Misailovic. 2021. Flex: fixing flaky tests in machine learning projects by updating assertion bounds. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[16] Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: program reduction for testing and debugging probabilistic programming systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[17] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–840.

[18] Aryaz Eghbali and Michael Pradel. 2022. DynaPyt: a dynamic analysis framework for Python. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 760–771.

[19] Youssef Elbadry. 2025. Predict the prices of houses. https://www.kaggle.com/youssefelbadry10/predict-the-prices-of-houses Accessed: May 28, 2025.

[20] Hans Fangohr, Vidar Fauske, Thomas Kluyver, Maximilian Albert, Oliver Laslett, David Cortés-Ortuño, Marijan Beg, and Min Ragan-Kelly. 2020. Testing with jupyter notebooks: Notebook validation (nbval) plug-in for pytest. *arXiv preprint arXiv:2001.04808* (2020).

[21] fg-data profiling. 2026. Try out NBTest: A smarter tool for testing jupyter notebooks. https://github.com/Data-Centric-AI-Community/fg-data-profiling/issues/1761 Accessed: July 9, 2026.

[22] Muhammad Risqi Firdaus. 2025. Problem I: Titanic. https://www.kaggle.com/mrfirdauss20/problem-i-titanic Accessed: May 28, 2025.

[23] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[24] Google. 2025. Google Colaboratory. https://colab.research.google.com/ Accessed: May 28, 2025.

[25] Konstantin Grotov, Artem Borzilov, Maksim Krivobok, Timofey Bryksin, and Yaroslav Zharov. 2024. Debug Smarter, Not Harder: AI Agents for Error Resolution in Computational Notebooks. *arXiv preprint arXiv:2410.14393* (2024).

[26] Konstantin Grotov, Sergey Titov, Yaroslav Zharov, and Timofey Bryksin. 2024. Untangling Knots: Leveraging LLM for Error Resolution in Computational Notebooks. *arXiv preprint arXiv:2405.01559* (2024).

[27] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 148–158. https://doi.org/10.1109/ICST49551.2021.00026

[28] Bernhard J. M. Grün, David Schuler, and Andreas Zeller. 2009. The Impact of Equivalent Mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*. 192–199. https://doi.org/10.1109/ICSTW.2009.37

[29] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*.

[30] Ruanqianqian Huang, Savitha Ravi, Michael He, Boyu Tian, Sorin Lerner, and Michael Coblenz. 2025. How Scientists Use Jupyter Notebooks: Goals, Quality Attributes, and Opportunities. *arXiv preprint arXiv:2503.12309* (2025).

[31] Ali Reza Ibrahimzada, Yigit Varli, Dilara Tekinoglu, and Reyhaneh Jabbarvand. 2022. Perfect is the enemy of test oracle. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[32] Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2019. Statistical algorithmic profiling for randomized approximate programs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 608–618.

[33] Project Jupyter. 2025. nbformat. https://pypi.org/project/nbformat

[34] Kaggle. 2024. Digit Recognizer. https://www.kaggle.com/competitions/digit-recognizer Accessed: Nov 27, 2024.

[35] Kaggle. 2024. House Prices - Advanced Regression Techniques. https://www.kaggle.com/competitions/house-prices-advanced-regression-techniques Accessed: Nov 27, 2024.

[36] Kaggle. 2024. How to Use Kaggle. https://www.kaggle.com/docs/api Accessed: October 25, 2024.

[37] Kaggle. 2024. Jane Street Real-Time Market Data Forecasting. https://www.kaggle.com/competitions/jane-street-real-time-market-data-forecasting Accessed: Nov 27, 2024.

[38] Kaggle. 2024. Kaggle competitions sorted by Hotness. https://www.kaggle.com/competitions?sortOption=default&hostSegmentIdFilter=5 Accessed: Nov 27, 2024.

[39] Kaggle. 2024. Spaceship Titanic. https://www.kaggle.com/competitions/spaceship-titanic Accessed: Nov 27, 2024.

[40] Kaggle. 2025. Kaggle: Your Machine Learning and Data Science Community. https://www.kaggle.com/ Accessed: May 28, 2025.

[41] Kaggle. 2025. Titanic - Machine Learning from Disaster. https://www.kaggle.com/competitions/titanic Accessed: May 28, 2025.

[42] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks–a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: Players, agents and agendas*. IOS press.

[43] Vadim Kravcenko and contributors. 2025. pipreqs: Generate requirements.txt based on imports of any project. https://github.com/bndr/pipreqs.

[44] Ben Liblit, Linghui Luo, Alejandro Molina, Rajdeep Mukherjee, Zachary Patterson, Goran Piskachev, Martin Schäf, Omer Tripp, and Willem Visser. 2023. Shifting left for early detection of machine-learning bugs. In *International Symposium on Formal Methods*. Springer.

[45] Shap Library. 2025. NBTest Pull Request. https://github.com/shap/shap/pull/4143 Accessed: May 28, 2025.

[46] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. Nnsmith: Generating diverse and valid test cases for deep learning compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 530–543.

[47] Yu Liu, Pengyu Nie, Anna Guo, Milos Gligoric, and Owolabi Legunsen. 2023. Extracting Inline Tests from Unit Tests. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[48] Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. 2022. Inline tests. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*.

[49] Yamilet R Serrano Llerena, Marcel Böhme, Marc Brünink, Guoxin Su, and David S Rosenblum. 2018. Verifying the long-run behavior of probabilistic system models in the presence of uncertainty. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[50] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.

[51] Scott Lundberg and SHAP Contributors. 2017. SHAP: A Game Theoretic Approach to Explain the Output of Any Machine Learning Model. https://github.com/shap/shap Accessed: July 9, 2026.

[52] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).

[53] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th international symposium on software reliability engineering (ISSRE)*. IEEE.

[54] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-grained lineage for safer notebook interactions. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1093–1101. https://doi.org/10.14778/3447689.3447712

[55] NBTest Authors. 2025. NBTest GitHub Repository. https://github.com/seal-research/NBTest Accessed: May 28, 2025.

[56] NBTest Authors. 2026. NBTest Zenodo Artifact. https://zenodo.org/records/21149272 Accessed: July 3, 2026.

[57] Mahdi Nejadgholi and Jinqiu Yang. 2019. A study of oracle approximations in testing deep learning libraries. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

[58] nteract. 2025. Testbook. https://github.com/nteract/testbook

[59] NVIDIA-BioNeMo. 2026. Try out NBTest: A smarter tool for testing jupyter notebooks. https://github.com/NVIDIA-BioNeMo/bionemo-recipes/issues/882

[60] The pandas development team. 2020. *pandas-dev/pandas: Pandas*. https://doi.org/10.5281/zenodo.3509134

[61] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32 (2019). https://arxiv.org/abs/1912.01703

[62] Jibesh Patra and Michael Pradel. 2022. Nalin: learning from runtime behavior to find name-value inconsistencies in jupyter notebooks. In *Proceedings of the 44th International Conference on Software Engineering*.

[63] 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 E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011).

[64] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2019. Deepxplore: Automated whitebox testing of deep learning systems. *Commun. ACM* 62, 11 (2019), 137–145.

[65] Jeffrey M Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563, 7732 (2018), 145–147. https://doi.org/10.1038/d41586-018-07196-1

[66] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1027–1038.

[67] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and opportunities in training deep learning software systems: An analysis of variance. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 771–783.

[68] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A large-scale study about quality and reproducibility of jupyter notebooks. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press. https://doi.org/10.1109/MSR.2019.00077

[69] Pluto.jl. 2025. Pluto.jl. https://github.com/fonsp/Pluto.jl

[70] Neoklis Polyzotis, Martin Zinkevich, Sudip Roy, Eric Breck, and Steven Whang. 2019. Data validation for machine learning. *Proceedings of machine learning and systems* 1 (2019).

[71] pytest developers. 2025. Writing hook functions — pytest documentation. https://docs.pytest.org/en/stable/how-to/writing_hook_functions.html.

[72] Python. 2025. AST - Abstract Syntax Trees. https://docs.python.org/3/library/ast.html

[73] Derek Robinson, Neil A Ernst, Enrique Larios Vargas, and Margaret-Anne D Storey. 2022. Error identification strategies for Python Jupyter notebooks. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*.

[74] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating large-scale data quality verification. *Proceedings of the VLDB Endowment* 11, 12 (2018).

[75] Arumoy Shome, Luis Cruz, Diomidis Spinellis, and Arie van Deursen. 2024. Understanding Feedback Mechanisms in Machine Learning Jupyter Notebooks. *arXiv preprint arXiv:2408.00153* (2024). https://arxiv.org/abs/2408.00153

[76] Md Saeed Siddik, Hao Li, and Cor-Paul Bezemer. 2025. A Systematic Literature Review of Software Engineering Research on Jupyter Notebook. *arXiv preprint arXiv:2504.16180* (2025).

[77] Pavle Subotić, Lazar Milikić, and Milan Stojić. 2022. A static analysis framework for data science notebooks. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*.

[78] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).

[79] Alexi Turcotte and Zheyuan Wu. 2025. Expressing and checking statistical assumptions. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2735–2758.

[80] Vishnupriya. 2025. Spaceship Titanic Classification. https://www.kaggle.com/vishnupriyagarige/spaceship-titanic-classification Accessed: May 28, 2025.

[81] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. 2020. Restoring reproducibility of Jupyter notebooks. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering: Companion proceedings*.

[82] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring execution environments of Jupyter notebooks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE.

[83] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*.

[84] Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: Fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*. 995–1007.

[85] Robert White and Jens Krinke. 2020. Reassert: Deep learning for assert generation. *arXiv preprint arXiv:2011.09784* (2020).

[86] Olivia Wiles, Sven Gowal, Florian Stimberg, Sylvestre Alvise-Rebuffi, Ira Ktena, Krishnamurthy Dvijotham, and Taylan Cemgil. 2021. A Fine-Grained Analysis on Distribution Shift. arXiv:2110.11328 [cs.LG] https://arxiv.org/abs/2110.11328

[87] Tao Xie. 2006. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming*. Springer.

[88] Chenyang Yang, Rachel A Brower-Sinning, Grace Lewis, and Christian Kästner. 2022. Data leakage in notebooks: Static detection and better processes. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*.

[89] ydata profiling. 2025. Add support for NBTest. https://github.com/ydataai/ydata-profiling/pull/1772 Accessed: May 28, 2025.

[90] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 1 (2020), 1–36.

[91] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2022. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Trans. Softw. Eng.* 48, 1 (Jan. 2022), 1–36. https://doi.org/10.1109/TSE.2019.2962027

[92] Renato Magela Zimmermann, Sonya Allin, and Lisa Zhang. 2023. Common Errors in Machine Learning Projects: A Second Look. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*.