Checking Unsupervised Learning for Nondeterminism and Inconsistency via SMT Solving

Muyeed Ahmed
New Jersey Institute of Technology
Newark, NJ, USA
ma234@njit.edu

Iulian Neamtiu
New Jersey Institute of Technology
Newark, NJ, USA
ineamtiu@njit.edu

Abstract

Unsupervised Learning (UL) implementations are widely used, but prone to issues such as nondeterminism and inconsistency, which makes them unreliable and exploitable. Moreover, the lack of a specification (or ground truth) makes UL implementations hard to verify or test: when they produce diverging results on a dataset, they do so subtly and silently, which gives users a false sense of security, and complicates developers' job of tracing the error to its root cause. We introduce an approach, Ocelot, that reduces nondeterminism/inconsistency checking to satisfiability checking in Z3. Given the Python code of an UL implementation, Ocelot automatically isolates a nondeterministic "kernel" (variables and code), which forms the basis for a Z3 specification. A Z3-found solution acts as a "witness" dataset – a typically small dataset that can be used to reliably reproduce the error. Ocelot has exposed sources of nondeterminism and inconsistency in popular implementations (Scikit-learn, MATLAB, and R) of well-known Clustering algorithms (Affinity Propagation, DBSCAN, HAC, and K-means) and Anomaly Detection algorithms (Isolation Forest, Local Outlier Factor). Moreover, Ocelot has exposed regressions (changes in semantics) in the evolution of these implementations.

CCS Concepts

• **Software and its engineering** → **Formal software verification**; **Software verification and validation**; • **Computing methodologies** → **Cluster analysis**; **Anomaly detection**.

Keywords

Unsupervised Learning, SMT Solving, Formal checking

ACM Reference Format:
Muyeed Ahmed and Iulian Neamtiu. 2026. Checking Unsupervised Learning for Nondeterminism and Inconsistency via SMT Solving. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787844

1 Introduction

Unsupervised Learning is a branch of Machine Learning that constructs approaches and algorithms for performing learning tasks, e.g., inferring patterns, without training or labels (as these are scarce or unavailable). However, UL implementations can be unreliable, and their failures can be silent (Section 2). We focus on two popular UL areas. Clustering (cluster analysis) groups together entities that are related or share similar characteristics. Anomaly Detection (Outlier Detection) identifies points in a dataset that were generated by means other than "normal" processes. We introduce an SMT-based approach for exposing nondeterminism and inconsistency. *Nondeterminism* refers to an UL implementation producing different outputs when run repeatedly on the same input dataset and with the same hyperparameters. Note that a nondeterministic implementation can still behave deterministically for some run sequences and on some datasets, which complicates finding a "witness" dataset, especially a small one, and reproducing the behavior. Figure 1 illustrates nondeterminism and our solution. Scikit-learn's AP clustering implementation, when run twice on the same 3-point dataset, produces three clusters in one run, and two clusters in a subsequent run. Our approach has automatically constructed this witness dataset. *Inconsistency* refers to two toolkits, implementing the same UL algorithm, producing different outputs on the same input dataset. Figure 2 shows MATLAB and Scikit-learn producing inconsistent results (different clusterings) on the same 5-point input. Our approach has constructed this witness dataset automatically.

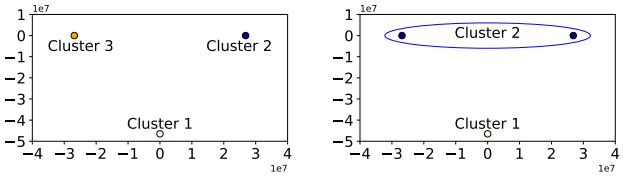
Nondeterminism and inconsistency undermine two key assumptions: (1) toolkit output is the same across runs, and (2) toolkits implementing the same algorithm are interchangeable. This affects users and developers. Users might not even be aware of nondeterminism. For example, in an exploratory study, we found that a toolkit can exhibit nondeterministic output after a sequence of 4–79 runs with deterministic (unchanged) output. Developers might notice output differences (with the right dataset, hyperparameters, and abundant testing time), but they cannot easily isolate, or trace the error's root cause. Generally, debugging and testing UL is challenging when outputs vary unpredictably between runs or toolkits.

Hence we need a *constructive, algorithmic way* of exposing such issues. We introduce a novel symbolic encoding that reduces nondeterminism or inconsistency checking to satisfiability checking in Z3 [19]. Our approach (Section 5) is implemented in a tool named Ocelot[1] (NOndeterminism and InConsistency VErifier for CLustering and AnOmaly DeTection). Ocelot operates on Python code. First, Ocelot extracts implementation parts responsible for nondeterminism or inconsistency, which we call the "nondeterministic kernel". Though our examined UL implementations are 5,800–8,600 LoC, Ocelot reduced them to 109–382 LoC of kernel code. The kernel is used to create a "Z3 encoding", i.e., a simplified representation (containing just the critical variables/conditions, and bindings to Z3 functions corresponding to Python ones), requiring

ICSE '26, Rio de Janeiro, Brazil

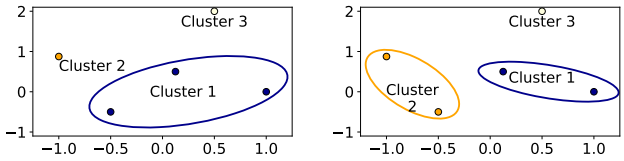
ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787844

[1] https://github.com/MuyeedAhmed/Ocelot



(a) Run 1 (b) Run 2

Figure 1: Affinity Propagation (AP): Our Approach Found a 3-Point Witness Dataset Exposing SKlearn Nondeterminism.



(a) MATLAB (b) Scikit-learn

Figure 2: Hierarchical Agglomerative Clustering (HAC): Our Approach Found a 5-Point Witness Dataset Exposing the MATLAB v. SKlearn Inconsistency.

a manual effort of less than 100 LoC per algorithm, amenable to satisfiability checking via Z3. When Z3 proves satisfiability, OCELOT produces a witness set, which establishes nondeterminism (or inconsistency), and is used to reproduce the issue. In our evaluation, all witness sets had at most 5 points, showing OCELOT's effectiveness.

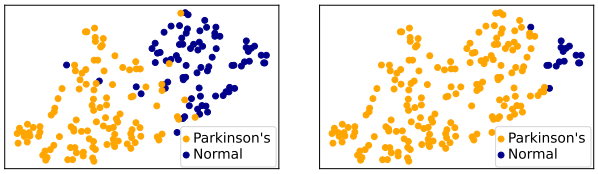
OCELOT proved nondeterminism in the popular Scikit-learn toolkit, as well as inconsistency between Scikit-learn and other toolkits, e.g., MATLAB and R. Moreover, OCELOT revealed the root causes of nondeterminism and inconsistency, helping developers devise mitigation strategies (Section 6.1). Checking was efficient, taking less than 6 minutes (Section 6.2). Section 7 describes our specification and Z3 encoding strategies. Our approach exposed inconsistencies arising in the evolution of our examined toolkits (Section 8) which demonstrates its effectiveness for establishing update correctness, performing regression testing, or exposing performance optimizations' trade-offs. Section 9 discusses soundness assumptions.

In summary, this paper makes the following contributions:

- A formal framework for checking nondeterminism and inconsistency in UL algorithms using an SMT solver.
- A technique based on trace differencing and taint analysis to expose variables and code inducing nondeterminism.
- A tool, OCELOT, that helps UL developers/users/researchers expose nondeterminism and inconsistency arising during UL toolkit development, evolution, or use.
- A demonstration of OCELOT's effectiveness on popular UL algorithms and popular ML toolkits.

2 Motivation

Algorithms, as well specific implementations that we check in this work, are used in the industry as part of enterprise AI [20], for bioinformatics research such as biomarker discovery [36], marketing [15], analytics [18], and a variety of enterprise applications [47]. More broadly, Clustering and AD are used in critical domains, where



(a) Run 1 (b) Run 2

Figure 3: Nondeterminism: AP on the parkinsons Dataset Produces Substantially Different Results in Two Different Runs.

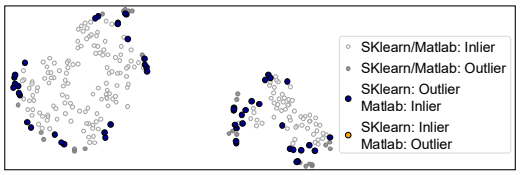


Figure 4: Inconsistency: Differences in Isolation Forest Output on the analcatdata_lawsuit Dataset (SKlearn v. MATLAB).

results must be reliable and reproducible.[2] In such settings, nondeterminism leads to unreliable results and flaky tests [56]; inconsistency can "break the ML pipeline" due to diverging results when modifying or switching toolkits. In a security context, attackers can exploit AD nondeterminism or inconsistency [5]. Ultimately, this leads to untrustworthy ML. OCELOT alleviates these issues because it exposes deliberate, as well as accidentally-introduced, nondeterminism/inconsistency, occurring when, e.g., developing a new implementation, or an implementation/its libraries change, or a code snippet is rewritten for performance. Prior research on nondeterminism and inconsistency [3, 58, 59] used testing on existing datasets, which is not rigorous/generalizable, and the proposed solutions were ad-hoc. We exemplify how nondeterminism and inconsistency affect outcomes in reliability-critical scenarios.

Example 1: Clustering nondeterminism on a Parkinson's disease dataset. Figure 3 illustrates clustering nondeterminism: different clustering outcomes in two different runs of Scikit-learn's AP algorithm implementation. The 195-point parkinsons dataset [32] contains voice measurements used to detect Parkinson's disease, with two classes: individuals with Parkinson's ("Cluster 1" in orange) and without Parkinson's ("Cluster 0" in blue). Though users expect the same clustering across the two runs, 64 points initially classified as cluster 0 flipped to cluster 1 in the second run. In fact, the Adjusted Rand Index (ARI) score[3] between the two runs was 0.096, indicating significant disagreement between the two clusterings. For easier visualization we used t-SNE [53] to show the data in 2D, but the dataset has 22 dimensions. For datasets with high dimensionality (or #points), we argue that even experts would struggle to spot or interpret cluster differences between runs or toolkits.

Example 2: Anomaly detection inconsistency on a lawsuit dataset. Figure 4 demonstrates AD inconsistency when running the IF algorithm using Scikit-learn and MATLAB on the analcatdata_lawsuit dataset [46] that contains lawsuit outcomes. Here, outliers (anomalies) correspond to losing the case, while inliers represent winning

[2] E.g., medical image processing [40] or diagnosis [54]; predicting disease-related genes [52], resource allocation for diseases [31], criminal justice [13], or finance [16].
[3] ARI [29] is a scalar measure of similarity between two clusterings of the same set, ranging between -1 and +1, where -1 means strong disagreement, 0 indicates no relation/random assignment, and +1 shows perfect agreement.

the case. The results varied significantly between the toolkits. Grey points represent data classified as inliers by both toolkits, while blue and orange points represent disagreements: blue points were classified as outliers by Scikit-learn only, while orange points were classified as outliers by MATLAB only. Inconsistency is also evident from the low ARI, 0.29, between classifications, indicating substantial disagreement between the toolkits.

3 Background

Given a dataset of n points $X = x_1, x_2, ..., x_n$ where each x_i is a vector in R^d, a UL algorithm aims to find patterns or groupings in X without knowing their associated labels.

3.1 Clustering

Clustering groups similar data points based on shared characteristics. Given a dataset X, a clustering algorithm partitions X into k clusters $c_1, c_2, \ldots, c_k$, such that points in the same cluster are closer to each other than to points in other clusters. Some algorithms require that users specify k, while others automatically determine k. Algorithms, discussed next, can be categorized into deterministic (but potentially inefficient), aiming to produce the same clustering when run repeatedly on the same dataset, and nondeterministic (but efficient). We now present a brief overview of each algorithm.

Deterministic Algorithms. These algorithms tend to have high asymptotic complexity (time&space) but aim to produce high-quality deterministic results. *Affinity Propagation (AP)* [23] groups together points having high "affinity" (low distance) toward an "exemplar", i.e., cluster representative. Users do not have to indicate the number of clusters k, as the AP infers it. AP is a deterministic algorithm [22], but can become nondeterministic due to implementation choices, as Ocelot has proven. When developers make such design decisions, unaware users (who assume determinism) will be caught off-guard. *Hierarchical Agglomerative Clustering (HAC)* [6] builds a cluster tree, starting with individual points as separate clusters, and merging them until k clusters are formed. *Density-Based Spatial Clustering (DBSCAN)* defines clusters by analyzing the density of points within a specified radius, and grouping densely connected points. Similar to AP, DBSCAN automatically infers k based on the radius and minimum number of samples for a cluster [21].

Nondeterministic Algorithms. These algorithms are efficient, but their results can differ across runs, due to randomization. For example *K-means* [34] partitions the dataset into k clusters by minimizing the distance between data points and their respective cluster centroids; centroids are initialized randomly and iteratively updated until convergence. Attempts to make K-means deterministic (or consistent) involve fixing the centroids across runs (or toolkits, respectively) [58]; however, as Ocelot has proven, those attempts can still fail, due to subtle semantic causes, e.g., tie-breaking strategy.

3.2 Anomaly Detection

Anomaly Detection identifies points that deviate significantly from the normal distribution of a set. Given a set X (without labels), AD separates normal data points from outliers. We examined two AD algorithms. *Local Outlier Factor (LOF)* [14] is a deterministic, density-based method that detects outliers by analyzing the local density of a point w.r.t. the density of its neighbors. A point is flagged as outlier if its "anomaly score", i.e., the ratio of its local reachability distance (*lrd*, the distance to its neighbors) to the reachability of those neighbors, exceeds a predefined threshold. *Isolation Forest (IF)* [33] is a nondeterministic algorithm that constructs random binary trees ("isolation trees" or "estimators") through random feature splits, recursively splitting the set. The depth of the tree for a data point indicates how anomalous it is.

4 Example: Symbolic Encoding

We show how our approach checks a clustering implementation, by juxtaposing the Python code with its Z3 symbolic encoding. The Python code implements a simplified version of AP (Section 3.1). Given a set of n points, their pairwise distances are computed first. For each point, its candidate exemplar (cluster center) is computed as the median of that point's distances. Figure 5 (a) shows the Python implementation, a function named AP_light. The function takes an n×n matrix X as argument (the input points). The function first calculates the pairwise Euclidean distances dist between points, and then the median (or preference) of these distances. Next, it replaces the diagonal elements of dist with the preference value, and adds a small random noise (ϵ). Finally, for each point, the function assigns its closest point as its label and returns the labels.

Figure 5 (b) shows the Python code after Ocelot extracted the ND kernel, with Z3 bindings (Z3 API calls invokable from Python) manually added. This program, AP_light_Z3, operates similarly to AP_light but instead of the Python implementations, it uses Z3 symbolic functions to compute Euclidean distance (z3_euclidean_dist), median (z3_median), and the nearest point (z3_arg_min). Note that these Z3 functions are not built-in; rather we implemented them, along with other mathematical and statistical functions (discussed in Section 10). We model the random noise ϵ as an abstract choice between $-\epsilon$ and ϵ using Z3's symbolic, logical OR (z3_Or).

To detect nondeterminism formally, we use a generic function nondeterminism_check (Figure 6) that imposes a timeout and a bound on set size. The function creates a matrix X of Z3 Reals of size n (line 3). Next, it invokes the Z3-encoded function AP_light_Z3 twice on the same input and generates two sets of labels , i.e., outputs, labels_r1 and labels_r2 (lines 4 and 5) . Label differences prove nondeterminism: the same input produces different outputs. Z3 checks if this is satisfiable (line 6). If yes, X is a nondeterminism witness. When Z3 returns Unsatisfiable, we increase the set size and re-run the solver. If Z3 times out or reaches the bound, we conclude that the implementation is deterministic (or consistent) up to the bound. Note that nondeterminism_check is generic: it is applicable to any algorithm by simply replacing AP_light_Z3 with that algorithm's Python-with-Z3-bindings code. Once Z3 has proven satisfiability, i.e., found a witness set of size n, no further checking is needed: the algorithm is certainly nondeterministic for set sizes > n. Ocelot can also explore different dimensionalities for witness sets (Figure 6 omits this for brevity); we used 2-dimensions as default, though for some algorithms, 1-dimension sufficed.

Figure 5 (c) shows the constraints that Z3 generates automatically from AP_light_Z3 and nondeterminism_check. The preference (median) constraints on lines 2 and 3 ensure that the selected preference value is the median of the distribution of pairwise squared distances: notice the counting of distances that are ≤ preference or

(a) Original Code

```
1 def AP_light(X):
2   n = size(X)
3
4   dist = euclidean_dist(X)
5
6   preference = median(dist)
7   for i in range(n):
8     dist[i][i] = preference
9
10  dist += random.uniform(−ϵ, ϵ)
11
12  labels = arg_min(dist)
13  return labels
```

(b) Code After Abstraction and Z3 Bindings

```
1 def AP_light_Z3(X):
2   n = size(X)
3   # Z3 symbolic function
4   dist = z3_euclidean_dist(X)
5   # Z3 symbolic function
6   preference = z3_median(dist)
7   for i in range(n):
8     dist[i][i] = preference
9   # Data Abstraction
10  dist += −ϵ z3_Or ϵ
11  # Z3 symbolic function
12  labels = z3_arg_min(dist)
13  return labels
```

(c) Z3 Constraints

```
1 # Preference (Median) conditions:
2 ∑_{i=0}^{n} ∑_{j=0}^{n} If((X_i − X_j)^2 ≤ preference, 1, 0) ≥ n^2
3 ∑_{i=0}^{n} ∑_{j=0}^{n} If((X_i − X_j)^2 ≥ preference, 1, 0) ≥ n^2
4 # Distance matrix conditions:
5 dist^{r1}_{i,j} = ((X_i − X_j)^2 − ϵ) ∨ ((X_i − X_j)^2 + ϵ)   for i ≠ j
6 dist^{r1}_{i,i} = (preference − ϵ) ∨ (preference + ϵ)
7 # Check minimum distance & assign labels
8 Labels^{r1}_i = { 1   if min(dist^{r1}_{i,1}, dist^{r1}_{i,2}, ..., dist^{r1}_{i,n}) == dist^{r1}_{i,1}
                  { 2   if min(dist^{r1}_{i,1}, dist^{r1}_{i,2}, ..., dist^{r1}_{i,n}) == dist^{r1}_{i,2}
                  { ... up to n
9 # Similar conds. for dist^{r2} and Labels^{r2}.
10 # Check nondeterminism
11 ∃i ∈ {1, ..., n}  such that  Labels^{r1}_i ≠ Labels^{r2}_i
```

Figure 5: Example: Code Abstraction and Z3 Encoding.

```
1 def nondeterminism_check(bound,timeout):
2  for n in range(3, bound):
3    X = [Real(X_i) for i in {1, 2, ..., n}]
4    labels_r1 = AP_light_Z3(X)
5    labels_r2 = AP_light_Z3(X)
6    if checkSat( labels_r1 != labels_r2 ):
7      return Satisfiable , X
8    # when checkSat times out, return Unsat. (timeout)
```

Figure 6: Checking Procedure.

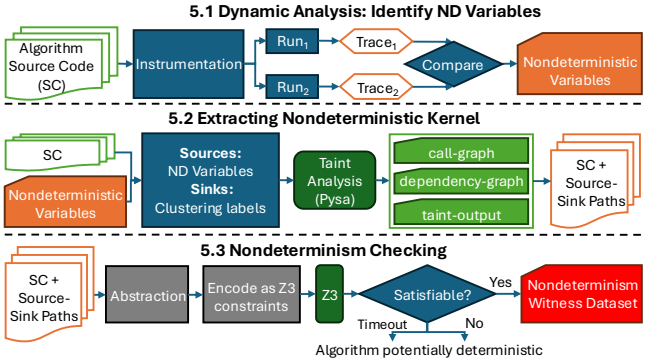


Figure 7: Ocelot Architecture.

≥ preference. For the distance matrix *dist*, the constraints on lines 5 and 6 use a symbolic choice ($-\epsilon$ or ϵ) to model randomness without assigning a specific value. This abstraction is key, allowing Z3 to explore different paths that lead to different outcomes. Label assignment on line 8 uses the minimum distance in the modified distance matrix to determine the index (label) of the closest point. The final constraint (line 10) checks for discrepancies in label assignments ($Labels^{r1}$ vs $Labels^{r2}$) across two runs of the function, enabling Z3 to determine if nondeterminism (different clusterings) is possible on identical inputs: if this condition is satisfiable, set X is a nondeterminism witness.

5 Implementation

Figure 7 shows Ocelot's architecture. Ocelot supports as-is, real-world Python code. Our approach has three stages: 1) dynamic analysis to identify the nondeterministic variables, 2) extracting the nondeterministic kernel via taint analysis, and 3) nondeterminism checking via SMT satisfiability. We describe each stage; the artifact's `README.md` file discusses the implementation at length.

5.1 Dynamic Analysis: Identifying ND Variables

Ocelot first runs a dynamic analysis to identify *ND variables*, i.e., variables that exhibit nondeterminism. Ocelot modifies the Python code's AST to instrument and serialize all variables (via the Pickle serialization library [44]). We call this 'Run 1'; the serialized execution is a trace akin to those obtained via execution indexing [57]. In 'Run 2', Ocelot reruns the algorithm. Next, it compares Run 2 values with Run 1 values at each point (instruction instance), to identify the variables whose values differ between runs. This comparison is straightforward for base types, but becomes more intricate for complex, nested, or recursive data structures, such as dictionaries or NumPy arrays. This first step over-approximates the variables of interest as it includes (a) variables that introduce nondeterminism into the execution and their dependents, and (b) other variables that change across runs but do not contribute to the clustering output (these are filtered out via taint analysis, as explained next). Section 9.1 discusses the soundness of this step.

5.2 Extracting the Nondeterministic Kernel

Ocelot uses the Pysa static taint analyzer [35] to isolate the sources of nondeterminism: ND variables are taint *sources*, while the algorithm's labels are taint *sinks*. This allows Ocelot to track how the nondeterministic variables influence the algorithm's execution and ultimately the clustering output. Pysa generates the call graph and data dependency graph that expose the paths leading to nondeterministic behavior. Ocelot automatically annotates the source code with comments to visualize taint propagation and highlight the *nondeterministic (ND) kernel*. These annotations guide developers, providing visibility into the sources of nondeterminism, their propagation, and how they reach the clustering output.

Table 1: Issues Exposed and the Sizes of Witness Sets.

		Algorithm	Toolkit	Witness size
ND	Clustering	AP	SKlearn	(3,2)
		K-means	SKlearn	(4,2)
	AD	IF	SKlearn	(5,1)
Inconsistency	Clustering	AP	SKlearn v. R	(5,2)
		K-means	SKlearn v. R	(4,2)
		HAC	SKlearn v. MATLAB	(5,2)
		DBSCAN	SKlearn/R v. MLpack	(3,2)
	AD	IF	SKlearn v. MATLAB	(5,1)
		LOF	SKlearn v. MATLAB	(5,1)

5.3 Checking as SMT Satisfiability

We now describe how we encode the kernel into the Z3 solver, yielding the "Python-with-Z3-bindings" or simply "Z3 encoding". Bridging the Python-Z3 gap involves two steps.

Step 1: Abstraction. The first step, data and control abstraction, maps Python types and operations into Z3 symbolic equivalents. While not strictly necessary, abstraction improves the efficiency of the symbolic analysis. *Data abstraction* involves simplifying or generalizing some data structures, e.g., instead of tracking every element in a large data structure, we may represent it symbolically or track aggregate properties. *Control abstraction* reduces the complexity of control flow e.g., by abstracting loop conditions (Section 10).

Step 2: Integrating the Z3 API. The second step consists of replacing the kernel's Python operations and library functions with equivalent symbolic functions (e.g., median with `z3_median`, as discussed in Section 4). For example, most operations in the ND kernel are on matrices, but Z3 does not have direct support for data types, associated operations, and symbolic reasoning procedures specific to UL. Therefore, many of these equivalent functions come from our library (Section 10). Abstraction and Z3 integration form the "Manual effort" discussed in our evaluation (Section 6.1/Table 2); this burden could be reduced with more engineering.

Checking Satisfiability. Z3 takes the encoding and runs the checking procedure shown in Figure 6 to find a solution, i.e., a set of values for the symbolic variables that satisfy the given conditions. If Z3 can find a solution where multiple output labels are possible for the same input set X, nondeterminism is confirmed, with X as a witness. Ocelot starts with a small set size, such as (3,2), i.e., 3 points and 2 dimensions; if no solution is found, it gradually increases the set size. For some algorithms, a size of (3,2) was sufficient, while others required larger sizes, such as (5,2), to observe nondeterminism. As the witness dataset reproduce the nondeterministic behavior, developers can see the specific cases where the implementation deviates from expected deterministic behavior. If Z3 times out or reaches the bound, the implementation is deemed deterministic (or consistent) up to the bound (Section 4).

6 Evaluation

We have evaluated Ocelot on the six algorithms discussed in Sections 3.1 and 3.2. Ocelot operates on Python code, the language used for implementation in Scikit-learn, so all Scikit-learn implementations could be checked for nondeterminism. For the other toolkits, while the implementation's source code is available, it is not written in Python: MATLAB and R have their eponymous programming languages, while MLpack uses C++. Therefore, for these toolkits, we only checked for inconsistency and regressions, by manually encoding their non-Python source code into Python-with-Z3-bindings. While other languages could be supported with more engineering, we found that once a Python implementation of an algorithm is encoded, specifying another, non-Python implementation, requires only a modest, incremental cost.

Table 2: Ocelot Effectiveness: Nondeterminism Results.

Algo.	#Variables		Code Size (LoC)		
	Total	ND(%)	Original	ND kernel(%)	Manual effort
AP	113	21 (18.6%)	5,890	250 (4.24%)	78
K-means	144	44 (30.6%)	8,672	382 (4.40%)	63
IF	170	25 (14.7%)	7,178	109 (1.52%)	96

High-level results. Table 1 shows the configurations where Ocelot proved nondeterminism (three algorithms as implemented in Scikit-learn) and inconsistency (all six algorithms). The small sizes of the witness sets – at most 5 points and 2 dimensions – evidence Ocelot's effectiveness and are essential for reproducibility as well as program understanding.

6.1 Effectiveness

We now evaluate the effectiveness of Ocelot.

Nondeterminism. *What percentage of variables are nondeterministic?* The '#Variables' columns in Table 2 show the total number of variables and the number/percentage that are nondeterministic. Ocelot has revealed that only 15–31% of variables are nondeterministic, which is a promising first step in helping developers isolate nondeterminism issues.

What fraction of the code constitutes the nondeterministic kernel? The 'Original' column in Table 2 shows the size of the Python implementation, while the 'ND kernel' column shows the size of the kernel after Ocelot's taint analysis phase. The reduction is substantial: the kernel makes up less than 5% of the code, which helps focus the analysis on the nondeterminism-critical code parts.

What is the manual burden? The manual effort (LoC) required to complete the encoding (Section 5.3) is shown in the last column of Table 2. We believe that this amount, less than 100 LoC per algorithm, is acceptable.

Is Ocelot effective at exposing the root cause of nondeterminism? For Scikit-learn's implementations of AP, K-means, and IF, Ocelot found that a single variable named random_state is primarily responsible for nondeterminism. However, this variable's semantics are completely different across algorithms. In AP, random_state affects the addition of noise to the similarity matrix S; in K-means, it is used to select random centroids; in IF, it determines the selection of estimator points and thresholds. Nevertheless, Ocelot exposed all these variables, indicating that its abstractions are effective.

Inconsistency. *What is the manual burden?* The effort for encoding non-Python code into our representation is shown in Table 5: in all cases less than 100 LoC, which we believe is acceptable, compared to the engineering effort required to add support for another language (as UL implementations tend to be written in languages that do not have a good program analysis infrastructure).

Is Ocelot effective at exposing the root cause of inconsistency? Ocelot was effective for all six algorithms. Table 3 shows the causes:

Table 3: Inconsistency Root Causes.

Algorithm	Cause
DBSCAN	default value: hyperparameter eps
HAC	default value: hyperparameter linkage
AP	default value: hyperparameter max_iter
K-means	semantic: tie-breaking
IF	semantic: incompatible hyperparam. contaminFactor/threshold
LOF	semantic: incompatible hyperparam. contaminFactor/threshold

Table 4: Ocelot Analysis Time for Nondeterminism.

Algorithm	Time (seconds)		
	ND variable identification	Kernel extraction	Verification (Z3)
AP	11.4	187	153
K-means	7.9	237	0.8
IF	7.1	147	0.57

in three cases, hyperparameter default values differed across toolkits, which is easily addressed. However, the other cases require source code changes. For HAC, Scikit-learn and MATLAB use completely different distance metric implementations. For K-means, the tie-breaking implementation is different. For IF and LOF, hyperparameters have different semantics in different toolkits.

How can developers eliminate inconsistency? Per Table 3, for three algorithms, developers can eliminate inconsistency by "harmonizing" hyperparameters–using the same value, rather than the (inconsistent) defaults. However, for K-means, IF, and LOF, Ocelot shows that even after eliminating nondeterminism and harmonizing hyperparameters, implementations are still inconsistent, for semantic reasons. The solution in such cases is to change the implementation.

6.2 Efficiency

We now discuss Ocelot's efficiency; Ocelot was run on an M1 MacBook Air with 8 GB RAM. For nondeterminism, Table 4 shows the time, in seconds, taken by each stage. ND variable identification took at most 11.4 seconds, kernel extraction at most 4 minutes, and the checking stage took less than 3 minutes. For inconsistency, Table 5 shows the time Z3 takes to prove inconsistency and produce a witness set. While we set a nominal two-hour timeout for satisfiability checking, this was unnecessary, as Ocelot took substantially less. We believe these times indicate that Ocelot is efficient, especially given the nontrivial properties it attempts to prove.

7 Checking Strategy

We now outline our strategies for symbolic encoding and specification. Our encodings allow other implementations, and clustering/AD implementations in general, to be specified/checked at a small cost, because much of the Z3 encoding can be reused.

7.1 Nondeterminism Checking

Affinity Propagation. While the research paper introducing the algorithm [24] does not specify that noise must be added, Scikit-klearn and R implementations add a small amount of random noise

Table 5: Inconsistency: Specification Burden and Ocelot Analysis Time.

	DBSCAN	HAC	K-means	IF	LOF
Manual effort (LoC)	42	61	58	95	75
Time (seconds)	0.17	1.39	0.46	0.31	0.79

to the similarity matrix to prevent "degeneracies" (including ties and division-by-zero errors) [10, 42]. This noise makes implementations nondeterministic, and its effect on the output varies, depending on the input dataset. Unfortunately, users cannot tell *a priori* if the noise will affect their specific dataset. This is an example of accidentally-introduced nondeterminism (e.g., during implementation, bug fixing, or optimization) that Ocelot exposes. We now discuss the AP Z3 encoding and the generated constraints.

Figure 8 juxtaposes the original Python implementation with the Z3 constraints. The first constraints involve calculating the similarity matrix $Sdist_{i,j}$ (line 3 on the left, line 2 on the right). Then, the AP code (left) introduces randomness in the similarity matrix S at line 4, which is reflected in the Z3 model as an OR (lines 4–7 on the right). The noise is defined as a function of $Sdist_{i,j}$, consistent with the code. However, unlike the original implementation, which allows a range of noise values, our abstraction simply adds the maximum (or minimum) possible noise value. In Section 9.2 we discuss the soundness of this abstraction. This abstraction is easier to model, and reason about in, Z3. Subsequently, the similarity matrix S is updated by replacing the diagonal with preference and adding random noise to all the elements.

We now discuss the other two matrices (lines 8–17 on the left, 9–12 on the right). The responsibility matrix R indicates how suitable point k would be as exemplar for point i, while the availability matrix A shows how appropriate it is for i to choose k as its exemplar. The updates follow the AP algorithm's rules and incorporate a damping factor λ to ensure convergence. Z3 constraints ensure these updates are consistent, allowing the solver to model the algorithm's iterative behavior. Once R and A have been updated, the exemplar for each point i is determined as follows: if $A_{i,i} + R_{i,i} > 0$, point i is considered an exemplar (cluster center). The same process (noise addition, matrix updates, and exemplar calculation) is used for Runs 1 and 2, producing two sets of exemplars E^{r1} and E^{r2}. The final constraint (line 16 on the right) checks whether there exists a point i for which the exemplar from Run 1 is different from the exemplar from Run 2. If yes ($E^{r1} \neq E^{r2}$), the process stops, as this indicates a divergence point, hence nondeterminism. Note that we check for differences in exemplars rather than calculating the labels directly, as different exemplars will inherently lead to different labels. This approach also simplifies the calculations significantly, reducing the computational complexity, as calculating the full set of labels for each iteration would add more constraints.

Isolation Forest. IF creates n estimators (typically between 100 and 500); each operating on a randomly selected sub-sample of the data (typically 256 points). Within each estimator, a tree is built by recursively partitioning the data. The partition boundaries are determined by threshold values chosen randomly at each step. This process continues until each leaf node contains one point or a predefined max_depth is reached.

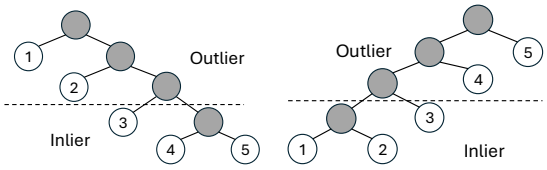
(a) Python

```
Input:X, preferences, max_iter, damping
Affinity_Propagation:
  S = -euclidean_distance(X)
  S += (double.eps * S + double.tiny * 100)*
       random_state.randn
  for i in range(n_samples):
    S[i, i] = preferences
  for iter in range(max_iter):
    # Update responsibility matrix R, using S and A
    for i in range(n_samples):
      for k in range(n_samples):
        R[i,k] = S[i,k] - max(S[i,j] + A[i,j] for j!=k)
        R = damping * R + (1 - damping) * R
    # Update affinity matrix A, using R
    for i in range(n_samples):
      for k in range(n_samples):
        A[i,k] = min(0,R[k,k]+sum(max(0, R[j,k]) for j!=i))
        A = damping * A + (1 - damping) * A
    if converged: break
    exemplars=[A[i,i]+R[i,i]>0 for i in range(n_samples)]
  labels =[argmax(S[i,exemplars]) for i in range(n_samples)]
  return labels
```

(b) Z3

```
# Calculate similarity matrix
Sdist_{i,j} = sum_{k=1}^{N} (X_{i,k} - X_{j,k})^2
# Constraints for Run 1; conditions for noise
noise_{i,j} = (ε · Sdist_{i,j} + tiny) · 6 ∨ -(ε · Sdist_{i,j} + tiny) · 6
# Replace diagonal with preference, add noise to Sdist_{i,j}
S_{i,j} = Sdist_{i,j} + noise^{r1}_{i,j}   if i ≠ j
S_{i,j} = preference + noise^{r1}_{i,j}    if i == j
# Conditions: R( responsibility ), A( availability ) matrix
∀t, i, k  R^{(t+1)}_{i,k} = λ · R^{(t)}_{i,k} + (1 − λ)
        · (S1_{i,k} − max_{kk≠k} (A^{(t)}_{i,kk} + S1_{i,kk}))
∀t, i   A^{(t+1)}_{i,i} = λ · A^{(t)}_{i,i} + (1 − λ) · Σ_{j≠i} max(R^{(t+1)}_{j,i}, 0)
∀t, i, k  sum_Rp_{i,k} =
        { 0, if R^{(t+1)}_{k,k} + Σ_{j≠i} max(R^{(t+1)}_{j,k}, 0) > 0
        { else R^{(t+1)}_{k,k} + Σ_{j≠i} max(R^{(t+1)}_{j,k}, 0)
∀t, i, k  A^{(t+1)}_{i,k} = λ · A^{(t)}_{i,k} + (1 − λ) · sum_Rp_{i,k}
# Condition for exemplar E
∀i ∈ {1, ..., n},  E_i = A_{i,i} + R_{i,i} > 0
# Similar constraints for Run 2, and get E^{r2}
∃i ∈ {1, ..., n} such that  E^{r1}_i ≠ E^{r2}_i  # Check nondeterminism
```

Figure 8: Affinity Propagation: Python Code (Left) and Z3 Constraints (Right).



(a) Run 1 (b) Run 2

Figure 9: IF: Inconsistency When Applying the Same Thresholds in Different Order.

```
1  # Declare n−1 split values to be used as threshold
2  splitVals = {α_1, α_2 ..., α_{n−1}}
3  # Get the path length for each point in X
4  PathL^{r1} = z3_estimator_fit(X, splitVals={α_1, α_2, ..., α_{n−1}})
5  PathL^{r2} = z3_estimator_fit(X, splitVals={α_{n−1}, α_{n−2}, ..., α_0})
6  # Get the anomaly score per point with the score func.
7  ∀i ∈ {1, ..., n}, anomaly^{r1}_i = { −1  if score(PathL^{r1}_i) > 0.5
                                      {  1  otherwise
8  ∀i ∈ {1, ..., n}, anomaly^{r2}_i = { −1  if score(PathL^{r2}_i) > 0.5
                                      {  1  otherwise
9  # Check nondeterminism
10 ∃i ∈ {1, ..., n}  such that  anomaly^{r1}_i ≠ anomaly^{r2}_i
```

Figure 10: Z3 Constraints Proving IF Nondeterminism.

Z3 can prove that *even with a fixed set of samples*, an estimator can still produce different results due to the random selection of partitions. Figure 9 illustrates how the output (inliers vs. outliers) differs for witness dataset X= [1, 2, 3, 4, 5] when using the same thresholds, [1.5, 2.5, 3.5, 4.5] and simply changing the threshold application order. In the first run, the input X is initially split using the threshold 1.5: value 1 is placed on the left and values 2. . .5 on the right, which assigns a path length of one to value 1. Next, X is split again using the threshold 2.5. This places value 2 to the left and values 3. . .5 to the right, giving value 2 a path length of two, while the remaining points (3. . .5) have path lengths greater than two. As a result, the output for this run identifies [1, 2] as outliers and [3, 4, 5] as inliers (Figure 9(a)). In contrast, in the second run, the initial split uses the threshold 4.5. This places value 5 in the right subtree and values 1. . .4 in the left subtree, assigning value 5 a path length of one. In the subsequent split, with threshold 3.5, value 4 is separated with a path length of two, while values 1. . .3 remain on the left. Therefore, this run identifies [4, 5] as outliers and [1, 2, 3] as inliers (Figure 9(b)).

Figure 10 shows the Z3 constraints generated from our encoding. The partition boundaries are set via the splitVals on line 2. Estimator trees for sample X are created by pre-selecting the splitVals and applying them in different order (lines 4–5). For the first run $r1$, splitVals use ascending order $\{\alpha_1, \alpha_2, \ldots, \alpha_{n-1}\}$, where $\{\alpha_i < \alpha_{i+1}\}$, and generate $PathL^{r1}$. With the $PathL^{r1}$ we can isolate the anomalies $anomaly^{r1}_i$ using the score function which indicates length (shorter length yields higher score, and vice versa). Similarly, we compute anomaly$^{r2}_i$ and compare it with anomaly$^{r1}_i$. Hence Ocelot proved the nondeterminism and generated a witness dataset, X= [1, 2, 3, 4, 5] and splitVals = [1.5, 2.5, 3.5, 4.5].

K-means. The K-means algorithm is nondeterministic due to randomness in the initial selection of centroids. Z3 can prove the

```
# Iteratively update cluster
For iter in range(max_iter):
  # Compute each point's distance to the centroids
  distance^{r1}_{i,c} = ∑_{j=0}^{d-1}(X_{i,j} − centroids^{r1}_{c,j})^2   ∀c ∈ {0,1}
  distance^{r1}_{i,c} = ∑_{j=0}^{d-1}(X_{i,j} − centroids^{r2}_{c,j})^2   ∀c ∈ {0,1}
  # Label assignment
  labels_new^{r1}_i = { 0  if distance^{r1}_{i,c0} > distance^{r1}_{i,c1}
                      { 1  if distance^{r1}_{i,c0} ≤ distance^{r1}_{i,c1}
  labels_new^{r2}_i = { 0  if distance^{r2}_{i,c0} > distance^{r2}_{i,c1}
                      { 1  if distance^{r2}_{i,c0} ≤ distance^{r2}_{i,c1}
  # Run1: force X_0 and X_1 to have labels=0
  distance^{r1}_{0,c0} ≤ distance^{r1}_{0,c1}
  distance^{r1}_{1,c0} ≤ distance^{r1}_{1,c1}
  # Run2: force X_0 to have label=0 and X_1 to have label=1
  distance^{r1}_{0,c0} ≤ distance^{r1}_{0,c1}
  distance^{r1}_{1,c0} > distance^{r1}_{1,c1}
  # Update centroids ; add conditions for convergence
```

Figure 11: Z3 Constraints Proving K-means Nondeterminism.

```
# Initialize equal centroids for both Sklearn and R
centroids^{sk}_{i,j} = centroids^{r}_{i,j}   ∀ i, j
# Iteratively update cluster
For iter in range(max_iter):
  # Compute each point's distance to the centroids
  distance^{sk}_{i,c} = ∑_{j=0}^{d-1}(X_{i,j} − centroids^{sk}_{c,j})^2   ∀c ∈ {0,1}
  distance^{r}_{i,c} = ∑_{j=0}^{d-1}(X_{i,j} − centroids^{r}_{c,j})^2   ∀c ∈ {0,1}
  # Label assignment for Scikit-learn:
  labels_new^{sk}_i = 0  if distance^{sk}_{i,c0} > distance^{sk}_{i,c1},  else 1
  # Label assignment for R:
  labels_new^{r}_i = 0  if distance^{r}_{i,c0} ≥ distance^{r}_{i,c1},  else 1
  # Update centroids for both Scikit-learn and R
```

Figure 12: Z3 Constraints Proving K-means Inconsistency.

nondeterminism and generate a witness, as shown in Figure 11. For simplicity, the number of clusters k is set to 2. In each iteration, the algorithm calculates the distance between each data point and the centroids, assigns clustering labels, and updates the centroids accordingly. To check for nondeterminism, four specific conditions are introduced (lines 9–14). In Run 1, both X_0 and X_1 are forced to have labels 0. In Run 2, X_0 is forced to have label 0, while X_1 is forced to have label 1. Z3 found these constraints satisfiable, confirming the nondeterminism and producing the witness dataset X and initial centroids where this behavior can be reproduced.

7.2 Inconsistency Checking

We now discuss encoding strategies and generated constraints for inconsistency.

```
# Get the distance matrix
distance_{i,j} = ∑_{k=0}^{d-1}(X_{i,k} − X_{j,k})^2
# Condition: distance (X_0,X_1) is the smallest
distance_{0,1} < distance_{i,j}   ∀ i < j and (i,j) ≠ (0,1)
# Single linkage
# Condition: distance (X_1,X_2) is the 2^nd smallest
# X_0, X_1 and X_2 might form a cluster after 2^nd iteration
distance_{1,2} < distance_{i,j}   ∀ i < j, (i,j) ≠ (0,1), (i,j) ≠ (1,2)
# Ward linkage
# New X^ward1 matrix after 1st ward iteration
X^{ward1}_{0,j} = (X_{0,j} + X_{1,j}) / 2   ∀ j ∈ {0,1,...,d−1}
X^{ward1}_{i,j} = X_{i+1,j},   if i > 0
# Calculate the new Ward distances
distance^{ward1}_{i,j} = (size_i × size_j)/(size_i + size_j) × ∑_{k=0}^{d-1}(X^{ward1}_{i,k} − X^{ward1}_{j,k})^2
# Condition: distance (X_1^ward1, X_2^ward1) is the smallest
# This might make distance (X_2,X_3) the 2^nd smallest
distance^{ward1}_{1,2} < distance^{ward1}_{i,j}   ∀ i < j, (i,j) ≠ (1,2)
```

Figure 13: Z3 Constraints Proving HAC Inconsistency.

K-means. While two K-means implementations can be inconsistent simply due to random initial centroids, prior research [58] attempted to make them consistent by using the same initial centroids. However, Ocelot has exposed that even with the same centroids, two toolkits can still produce inconsistent results, due to a subtle difference in tie-breaking strategy. Figure 12 shows the Z3 constraints for proving Scikit-learn v. R inconsistency. The key difference in the implementations is how they break ties while labeling (lines 9 and 11): notice the different handling of cases where data points are equidistant to two centroids, i.e., '>' for Scikit-learn and '≥' for R. This difference impacts how the clusters are formed and can lead to discrepancies in the final clustering outcomes, even when the initial cluster centers are identical. By incorporating these constraints, Z3 was able to demonstrate the existence of inconsistencies between the two methods and provided a witness dataset.

HAC. Figure 13 shows Z3 constraints proving inconsistency between Scikit-learn and MATLAB's HAC implementations. Interestingly, here the inconsistency is due not to hyperparameters, but rather the distance metric implementation: by default, Scikit-learn uses Ward linkage [37], while MATLAB uses single linkage [27]. The encoding uses a condition for the smallest distance to be between X_0 and X_1, forming the first cluster. For single linkage, the second smallest distance is enforced between X_1 and X_2, ensuring $[X_0, X_1, X_2]$ forms the next cluster. Conversely, for Ward linkage, the figure shows the formation of a new dataset where X_0 and X_1 are replaced by their centroid. The smallest distance in this transformed dataset is then enforced between X_1^{ward1} and X_2^{ward1}, leading to a different clustering outcome, where X_2 and X_3 cluster next. These decisions show the impact of linkage choice (Ward's uses centroid-based merging, while single uses nearest points) which ultimately results in different clusterings on the same input.

1 # Get negighbors of X_i
2 region_query(X, i, eps) =

$$\forall j \in \{1,\ldots,n\},\ \text{neighbors}_{i,j} = \begin{cases} 1 & \text{if } \|X_i - X_j\| < eps \\ 0 & \text{otherwise} \end{cases}$$

3 dbscan(X, eps):
4 neighbors_0 = region_query(X, 0, eps)
5 # If the number of neighbors is >= to min_pts that implies the label of X[0] is True
6 $\sum_{j=1}^{n} \text{neighbors}_0 \geq \text{min\_pts} \implies \text{label}_0 = \text{True}$
7 # For each neighbor of X[0] check if their neighbor count is greater than or equal to min_pts
8 For each $i \in \{1, 2, \ldots, n\}$:
9 neighbors_i = region_query(X, i, eps)
10 $\left(\sum_{j=1}^{n} \text{neighbors}_i \geq \text{min\_pts}\right) \wedge (\text{neighbors}_{i,0} = 1) \implies \text{label}_0 = \text{True}$
11 return label_0

Figure 14: Z3 Constraints Proving DBSCAN Inconsistency.

1 # Get the depth and anomaly score for each elements
2 $\text{score}_i = \text{anomaly\_score\_z3}(\text{depth}_i)$
3 # Sklearn: values over threshold 0.5 considered anomaly
4 $\text{labels}_i^{sk} = \begin{cases} 1 & \text{if score}_i > 0.5 \\ 0 & \text{otherwise} \end{cases} \quad \forall\, i \in \{1, ..., n\}$
5 #MATLAB: infer threshold T using contamination_factor
6 $\text{labels}_i^{mat} = \begin{cases} 1 & \text{if score}_i > T \\ 0 & \text{otherwise} \end{cases} \quad \forall\, i \in \{1, ..., n\}$
7 $\sum_{i=0}^{n-1} \text{labels}_i^{mat} < n * \text{contamination\_factor}$
8 # Check inconsistency
9 $\exists i \in \{1, ..., n\}$ such that $\text{labels}_i^{sk} \neq \text{labels}_i^{mat}$

Figure 15: Z3 Constraints Proving IF Inconsistency.

DBSCAN. While DBSCAN is deterministic, due to different default hyperparameter values in different toolkits, the outputs produced by two toolkits can be inconsistent.

For example, for hyperparameter *eps* (neighborhood radius), Scikit-learn and R use 0.5 as default value, while MLpack uses 1. Our Z3 encoding, shown in Figure 14, proved that the first point in the dataset (X_0) can be different in two different toolkits. Our encoding first calculates X_0's distance to other points in the dataset; if the distance to a point is less than *eps*, it counts as a neighbor. Next, it checks if X_0 has enough neighbors; if so it is classified as $True$, i.e., a cluster member. Otherwise, its neighbors are examined to see if at least one of them belongs to a cluster; if not, X_0 is classified as *False*, i.e., noise. We ran the Z3 procedure with two different *eps* values marked as eps_{sk} and eps_{ml}. Z3 was able to generate two different classifications for X_0, $classification0_{sk}$ and $classification0_{ml}$, which serve as a witness for the inconsistency.

Isolation Forest. Toolkits implementing IF can be inconsistent due to the random selection of samples and splits when building the isolation trees. To eliminate the inconsistency, we can fix the selection of samples and the splits. Even after fixing these selections,

Table 6: Regression Root Causes.

Algo	Tool(Version)	Cause
AP	Sklearn (1.4.1)	Tie-breaking behavior
AP	Sklearn (1.1.0)	Non-convergence handling
K-means	Sklearn (0.15.1)	Convergence with sparse data
K-means	Sklearn (1.0.0)	Inertia comparison tolerance
K-means	R (2.1.0)	Duplicate initial centroid handling
LOF	R(dbscan:0.9-7)	Duplicate points handling

Ocelot showed that Scikit-learn and MATLAB are still inconsistent due to the different methods they use for threshold handling. We show the Z3 constraints in Figure 15. Up to the point of anomaly score calculation, both Scikit-learn and MATLAB follow the same procedure (line 2). The divergence happens during the assignment of outliers, where Scikit-learn uses a fixed threshold of 0.5 to identify anomalies, while MATLAB computes a threshold T based on the contamination_factor hyperparameter (lines 4 and 6, respectively), which can lead to inconsistency.

Local Outlier Factor. While LOF is deterministic, Scikit-learn and MATLAB are inconsistent due to threshold handling. Both toolkits compute the Negative Outlier Factor (*NOF*) of each point i using Local Reachability Density (*LRD*):

$$NOF_i = -\frac{1}{k} \sum_{j \in N_k(i)} LRD_j / LRD_i$$

However, i's outlier status depends on how NOF_i compares against hardcoded (Scikit-learn) or inferred (MATLAB) thresholds. For brevity we omit constraints, but they were similar to Figure 15.

8 Regression Testing

Our approach can also check for inconsistency between old and new code – *regressions* or conversely, *update correctness*. We analyzed the commit histories of our examined toolkits and found a variety of scenarios (Table 6) where code changes (patches) led to changes in output (inconsistency). We now discuss the findings.

Case 1: AP Tie-breaking Behavior. Earlier versions of AP exhibited nondeterministic label assignments when input samples had equal pairwise distances [39, 51]. This behavior was caused by a tie-breaking condition that used '>=' instead of a strict '>'. Using symbolic inputs with equal distances, our tool successfully identified cases where label assignments varied across runs.

Case 2: AP Non-convergence Handling. In older versions, if AP failed to converge within the maximum number of iterations, it returned a label array consisting of −1. In contrast, newer versions return the last computed label assignment [50]. Our tool was able to generate symbolic inputs that revealed this inconsistency.

Case 3: K-means Convergence Criterion with Sparse Data. Originally, K-means checked convergence by verifying whether the total shift in cluster centers was less than *tol* multiplied by the mean variance of the dataset. When support for sparse matrices was introduced, the mean variance term was removed, altering the convergence condition [48]. Our tool identified symbolic inputs that led to inconsistent outputs due to this change. Additionally, a later fix replaced '<' with '<=' to handle edge cases such as $tol = 0$, which again caused inconsistencies. In both cases, our tool successfully produced witness inputs exposing the output change.

Case 4: K-means Inertia Comparison Tolerance. A patch was introduced to tolerate numerical errors in the algorithm when

selecting the best clustering result across multiple initializations. When choosing the inertia, the condition was changed [49] from $inertia < best\_inertia$ to $inertia < best\_inertia * (1 - 10^{-6})$. Our tool proved that the final selected label assignment can still differ across the two versions due to the relaxed comparison condition.

Case 5: K-means Duplicate Initial Centroid Handling. R's implementation of K-means had a patch that addressed an issue with duplicate initial centroids. Versions 2.0.1 and earlier allowed two or more initial centroids to be the same, which was addressed in version 2.1.0 by checking and replacing duplicate centroids [45].

Case 6: LOF Duplicate Points Handling. We analyzed R's LOF implementation (`dbscan` package) and found a patch that fixes a duplicate points issue. In earlier versions, when more than $MinPts$ identical points were present, the LRD (local reachability density) computation could result in a division by zero [28], causing the resulting lof_score to be $Null$. In the patched version, all $Null$ lof_score values are explicitly set to 1, hence marking such points as non-outliers. This behavioral change can lead to inconsistencies across versions, and we verified this discrepancy using our tool.

9 Soundness

We now discuss Ocelot's soundness assumptions, limitations, and mitigations. In short, our analysis guarantees that nondeterminism or inconsistency will be exposed if the following assumptions hold. For nondeterminism only, we assume that two different runs will expose low-level nondeterminism, i.e., variables; this would not hold in the (theoretical, rather than practical) scenario where nondeterministic functions behave deterministically in the two runs examined by Ocelot. We also assume that our Python-to-Z3 abstraction is correct; proving this correct would require formal modeling of Python, hence outside the scope of this work. We assume that specification (encoding) is correct; proving this correctness formally would be akin to proving the correctness of specification. We also assume that Pysa and Z3 are sound.

9.1 ND Variables: Soundness and Rigor

Soundness. Ocelot uses two runs to identify ND variables (Section 5.1), under the assumption that nondeterminism-induced differences are visible at the level of variables (i.e., Python execution trace); we'll illustrate shortly how these differences might not be visible at the output level. This assumption is sound whenever the sources of nondeterminism come from numerical or mathematical randomness, e.g., random initialization, random number APIs, floating-point rounding – as is the norm and expectation for AI/ML implementations. Nondeterminism can also occur due to other, uncommon factors, e.g., a program using getpid() or gettimeofday() in the computation of UL results. Ocelot does not consider those, as we believe such uses are unjustifiable and uncommon – we have not observed the use of such APIs in our examined implementations. Nevertheless, Ocelot can be extended with a simple configuration file indicating such sources of randomness, which the ND kernel extraction (Section 5.2) can use to verify if those API results (sources) propagate to the output (labels). If they do propagate, they can be included in the Z3 encoding (e.g., a PID variable ranging over integers) to soundly decide whether they contribute to nondeterministic output.

Table 7: # of Re-runs Required to Observe Nondeterminism.

	Algorithm	# Re-runs (Range)				
		1	2–10	11–50	51–100	>100
# Datasets	AP	94	50	22	6	215
	K-means	49	220	34	4	80
	IF	271	81	9	2	24

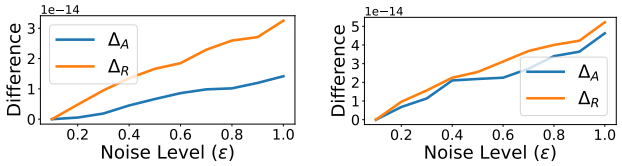
Rigor: the perils of resting. To demonstrate the importance of rigorous nondeterminism identification and testing's inadequacy, we ran Scikit-learn's nondeterministic implementations (AP, K-means, IF) on 387 datasets from OpenML and UCI [2, 7] repositories for at least 102 runs. Essentially, we measured how many re-runs are required to observe nondeterminism when using testing (i.e., only looking at the output). Labeling the initial run r_0, if the output is the same as r_0 for a sequence $r_1, r_2, \ldots, r_{n-1}$ and differs at r_n, the required number of re-runs is n. Unfortunately, when using testing, n cannot be predicted and can be arbitrarily large. In contrast, Ocelot requires a single re-run ($n = 1$). Table 7 shows the results. For example, for AP, 28 datasets required 11–100 re-runs before observing nondeterminism, whereas the last column shows that a substantial number of datasets require *more than 100 re-runs* to observe output differences. These results evince testing's inadequacy: a developer might run an implementation a few times, observe identical outputs, and mistakenly declare it deterministic.

9.2 Data Abstraction

SKlearn's AP implementation adds noise to the similarity matrix S. In our Z3 encoding (Section 7.1), we use only the two extreme values ($-\epsilon$ and ϵ) instead of allowing the noise variable to take any real value in the interval $[-\epsilon, \epsilon]$. This abstraction is AP-specific and was not used in other algorithms. We justify why this abstraction maintains soundness, and quantify how it speeds up checking.

Claim: abstraction is sound due to the monotonic relation between noise amplification and the A v. R divergence between runs. Since AP message updates are composed of additive and monotone operations (summation, maximization) over the entries of S, the magnitude of divergence grows monotonically with the amplitude of the perturbation ϵ. Because exemplar (E) selection depends on the summation of the availability (A) and responsibility (R) matrices (as illustrated in Figure 8 (b) line 14), widening differences in A and R directly translate to stronger separation in E, reinforcing the distinct exemplar decisions between runs. Consequently, pushing the noise values to the boundaries $\{-\epsilon, \epsilon\}$ preserves or amplifies any strict inequality that drives exemplar divergence. This guarantees that the endpoint noise abstraction remains sound for proving nondeterminism in AP [11]. We illustrate this monotonicity in Figure 16: the figure shows the aggregate divergence between A and R matrices from two independent runs as the uniform noise range expands from $[-t \times \epsilon, t \times \epsilon]$ for $t \in [0.1, 0.2, \ldots, 1]$. For each run pair, divergence is calculated as $\Delta_A = \sum_{i=1}^{N} |A_{ii}^{(1)} - A_{ii}^{(2)}|$ and $\Delta_R = \sum_{i=1}^{N} |R_{ii}^{(1)} - R_{ii}^{(2)}|$. Both Δ_A and Δ_R increase steadily with larger noise amplification, showing that stronger perturbations in the similarity matrix consistently lead to greater divergence in AP.

Abstraction's effect on runtime. Defining the search space for noise as a continuous interval in Z3 drastically increases the search



(a) 2nd iteration (b) 10th iteration

Figure 16: Monotonicity: Increase in A and R Divergence Between Two Runs w.r.t. Range of Noise Added.

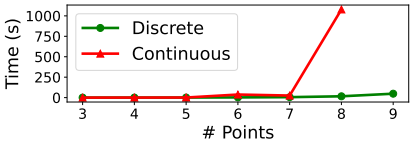


Figure 17: Data Abstraction (Discrete) Facilitates Scalability.

space, and therefore the solver's runtime grows exponentially. To quantify this, we ran a simplified version of the AP encoding and compared runtimes with and without abstraction (discrete vs. continuous). Figure 17 shows the runtime of the two different noise constraints: (1) discrete $\{-\epsilon, \epsilon\}$, and (2) continuous $[-\epsilon, \epsilon]$, across different sample sizes. With abstraction, we observe only a modest runtime increase as the number of points grows. However, without abstraction the runtime spikes to 1,078 seconds for just 8 points, compared to 16 seconds for the abstracted encoding. With 9 points, the continuous encoding failed to finish within a 2-hour timeout, while the abstracted version completed in 48 seconds.

9.3 Other Assumptions

Our approach assumes that the specification, i.e., integrating the Z3 API, is correct. Figure 5 (a)&(b) illustrate why the error risk is low: integration simply entails replacing a Python method call with its Z3 version (from Z3 or our library). The low specification burden, 42–96 LoC (Tables 2 and 5), reduces the trusted computing base by two orders of magnitude from the original implementation. Ocelot uses the Pysa static analyzer to extract the ND kernel. Pysa was built "to avoid false negatives and catch as many issues as possible" [9] so Ocelot, conservatively, *overapproximates* the kernel.

10 Extending Z3 with UL Symbolic Reasoning

Z3 has built-in "theories" offering data types, abstractions, and decision procedures for common symbolic reasoning, e.g., booleans, bitvectors, arithmetic over integers and reals. However Z3 does not have direct support for operations and symbolic reasoning used in UL implementations. Therefore, we wrote new versions of standard library functions (min, max, abs, exp, logarithm, Euclidean distance, etc.) involving arithmetic operations, so they work with symbolic variables. This library is available in our artifact. For example, our symbolic implementation of $\sqrt{x}$ using Newton-Raphson was:

```
1  def sqrt(x, initial_guess =1.0, iterations =10):
2      guess = RealVal( initial_guess )
3      for _ in range( iterations ):
4          guess = (guess + x / guess) / 2
5      return guess
```

Other abstraction tasks included symbolic variables used as array/matrix indices, or loops without fixed bounds (not supported directly in Z3). For symbolic indexing, we replaced direct accesses with explicitly-encoded constraints over array elements (e.g., when sorting a symbolic array, we introduced a new array representing the sorted version and constraints ensuring proper ordering). For loops without fixed bounds, we replaced some iterations with symbolic constraints, e.g., when termination conditions depend on symbolic values, we used recursive encoding or bounded unrolling.

11 Related Work

SMLP [12] applies SMT-based verification to *supervised* ML models, requiring users to specify formal assertions, e.g., variable constraints or stability conditions. SMLP assumes single execution hence cannot check nondeterminism and inconsistency (which requires reasoning across two executions). Katz et al. [30] used a custom SMT theory built on top of the GLPK LP solver [1] to reason about ReLU NN properties. Their properties encode safety for collision avoidance in unmanned aircraft. Similarly, we developed UL-specific abstractions to encode UL-domain properties in Z3. AI^2 [26] uses abstract interpretation to check NN properties, e.g., robustness, or resilience to perturbations. As UL is qualitatively different from NN, these approaches do not translate to our domain.

Invivo Fuzzing [25], introduced by Galland et al., is a method for library fuzzing that leverages existing application execution contexts to steer coverage-driven testing. This eliminates the necessity for hand-crafted fuzz drivers, allowing for more comprehensive exploration of authentic code paths. Similarly, CSFuzz [17] introduced by Chen et al. improves directed fuzzing by tracking critical variables and their runtime states to expose complex, state-dependent bugs. While effective at uncovering concrete runtime issues, fuzzing is inherently input-driven and probabilistic, a fundamental limitation that highlights the need for rigorous formal reasoning.

Nondeterminism and inconsistency studies on clustering [38, 58] and AD [3] proposed solutions such as controlling noise and hyperparameters [4, 59]. However, their focus was empirical (testing on existing datasets) rather than checking, or witness construction. Related empirical work in ML has studied stability (clustering resilience to changes in how points are chosen [55]), e.g., in AD [43], HAC [8], NN [41]. Stability is different from our objectives.

12 Conclusions

Clustering and Anomaly Detection are popular, but their implementations have not been formally checked. Our constructive, algorithmic approach implemented as the Ocelot Python code analyzer, has checked popular implementations and produced small witness datasets that expose implementation and regression issues. Our approach, usable by UL researchers or practitioners, is extensible, to verify other classes of programs similar to our examined algorithms.

Acknowledgments

We thank the reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-2007730 and CCF-2106710.

References

[1] [n. d.]. GNU Linear Programming Kit. https://www.gnu.org/software/glpk/. Accessed: 2024-10-17.

[2] 2025. OpenML — openml.org. https://www.openml.org. [Accessed 10-30-2025].

[3] Muyeed Ahmed and Iulian Neamtiu. 2022. Anomalous Anomaly Detection. In *AITest'22*. 1–6. doi:10.1109/AITest55621.2022.00009

[4] Muyeed Ahmed and Iulian Neamtiu. 2023. DeAnomalyzer: Improving Determinism and Consistency in Anomaly Detection Implementations. In *AITest'23*. doi:10.1109/AITest58265.2023.00012

[5] Muyeed Ahmed and Iulian Neamtiu. 2024. Quantifying the Vulnerability of Anomaly Detection Implementations to Nondeterminism-based Attacks. In *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*. IEEE, 37–46.

[6] Michael R Anderberg. 2014. *Cluster analysis for applications: probability and mathematical statistics: a series of monographs and textbooks*. Vol. 19. Academic press.

[7] Arthur Asuncion, David Newman, et al. 2007. UCI machine learning repository.

[8] Asa Ben-Hur, André Elisseeff, and Isabelle Guyon. 2002. A Stability Based Method for Discovering Structure in Clustered Data. In *Proceedings of the 7th Pacific Symposium on Biocomputing, PSB 2002, Lihue, Hawaii, USA, January 3-7, 2002*. 6–17. http://psb.stanford.edu/psb-online/proceedings/psb02/benhur.pdf

[9] Graham Bleaney and Sinan Cepel. 2020. Pysa: An open source static analysis tool to detect and prevent security issues in Python code. Engineering at Meta blog. (7 Aug. 2020). https://engineering.fb.com/2020/08/07/security/pysa/

[10] Ulrich Bodenhofer, Andreas Kothmeier, and Sepp Hochreiter. 2025. apcluster: apcluster-methods.R. https://github.com/UBod/apcluster/blob/master/R/apcluster-methods.R. Accessed: 2025-06-30, (lines 64–73).

[11] Stephen P Boyd and Lieven Vandenberghe. 2004. *Convex optimization*. Cambridge university press.

[12] Franz Brauße, Zurab Khasidashvili, and Konstantin Korovin. 2024. SMLP: Symbolic Machine Learning Prover. In *International Conference on Computer Aided Verification*. Springer, 219–233.

[13] Tim Brennan and William L. Oliver. 2013. The Emergence of Machine Learning Techniques in Criminology. *Criminology & Public Policy* 12 (08 2013). doi:10.1111/1745-9133.12055

[14] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. 2000. LOF: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 93–104.

[15] Adobe Business. 2025. Adobe's use of K-means and HAC for Marketing. Available at: https://business.adobe.com/blog/basics/cluster-analysis (Accessed: 2025-07-01).

[16] Fan Cai, Nhien-An Le-Khac, and Tahar Kechadi. 2016. Clustering Approaches for Financial Data Analysis: a Survey. (09 2016).

[17] Xu Chen, Ningning Cui, Zhe Pan, Liwei Chen, Gang Shi, and Dan Meng. 2025. Critical Variable State-Aware Directed Greybox Fuzzing. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 755–755.

[18] SAS Communities. 2025. SAS Enterprise supports HAC, K-means, DBSCAN. Available at: https://communities.sas.com/t5/SAS-Communities-Library/Tip-Guidelines-for-Choosing-a-Clustering-Method-in-the-Cluster/ta-p/223483 (Accessed: 2025-07-01).

[19] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[20] IBM Developer. 2025. Implement hierarchical clustering in Python. https://developer.ibm.com/tutorials/awb-implement-hierarchical-clustering-python/ Accessed: 2025-07-01.

[21] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD'96*.

[22] Brendan J Frey and Delbert Dueck. 2005. Mixture modeling by affinity propagation. *Advances in neural information processing systems* 18 (2005).

[23] Brendan J Frey and Delbert Dueck. 2007. Clustering by passing messages between data points. *science* 315, 5814 (2007), 972–976.

[24] Brendan J. Frey and Delbert Dueck. 2007. Clustering by Passing Messages Between Data Points. *Science* 315, 5814 (2007), 972–976. arXiv:https://science.sciencemag.org/content/315/5814/972.full.pdf doi:10.1126/science.1136800

[25] Octavio Galland and Marcel Böhme. 2025. Invivo Fuzzing by Amplifying Actual Executions. In *Proceedings of the 47th International Conference on Software Engineering (ICSE'25)*.

[26] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 3–18. doi:10.1109/SP.2018.00058

[27] John C Gower and Gavin JS Ross. 1969. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 18, 1 (1969), 54–64.

[28] Michael Hahsler. 2022. Fix index bug for single-point clusters in DBSCAN. https://github.com/mhahsler/dbscan/commit/e865cd3766a9a2c0c7220485aeaaf17a59c3c91c Commit, Accessed: 2025-04-23.

[29] Lawrence Hubert and Phipps Arabie. 1985. Comparing Partitions. 2 (02 1985), 193–218.

[30] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10426)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 97–117. doi:10.1007/978-3-319-63387-9_5

[31] Minlei Liao, Yunfeng Li, Farid Kianifard, Engels Obi, and Stephen Arcona. 2016. Cluster analysis and its application to healthcare claims data: a study of end-stage renal disease patients who initiated hemodialysis. *BMC nephrology* 17 (03 2016), 25; 25–25.

[32] Max Little. 2007. Parkinsons. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C59C74.

[33] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth ieee international conference on data mining*. IEEE, 413–422.

[34] J Macqueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability/University of California Press*.

[35] Meta. 2020. Pyre: A performant type checker for Python, with Pysa for security analysis. https://github.com/facebook/pyre-check. Accessed: 2024-10-04.

[36] Metabolon. 2025. Metabolon's use of K-means, HAC, and DBSCAN for Bioinformatics. Available at: https://www.metabolon.com/bioinformatics/clustering-analysis/ (Accessed: 2025-07-01).

[37] Fionn Murtagh and Pierre Legendre. 2014. Ward's hierarchical agglomerative clustering method: which algorithms implement Ward's criterion? *Journal of classification* 31 (2014), 274–295.

[38] Vincenzo Musco, Xin Yin, and Iulian Neamtiu. 2019. SmokeOut: An Approach for Testing Clustering Implementations. In *12th IEEE International Conference on Software Testing, Verification and Validation, ICST 2019*.

[39] User: n1k31t4 and Response by: Andreas Mueller. 2018. Affinity Propagation sklearn strange behavior. https://stats.stackexchange.com/questions/156924/affinity-propagation-sklearn-strange-behavior/323665 Accessed: 2025-04-23.

[40] H. P. Ng, S. H. Ong, K. W. C. Foong, P. S. Goh, and W. L. Nowinski. 2006. Medical Image Segmentation Using K-Means Clustering and Improved Watershed Algorithm. In *2006 IEEE Southwest Symposium on Image Analysis and Interpretation*. 61–65. doi:10.1109/SSIAI.2006.1633722

[41] YeongHyeon Park. 2023. Concise Logarithmic Loss Function for Robust Training of Anomaly Detection Model. arXiv:2201.05748 [cs.LG] https://arxiv.org/abs/2201.05748

[42] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2025. scikit-learn: Affinity Propagation clustering implementation. https://github.com/scikit-learn/scikit-learn/blob/main/sklearn/cluster/_affinity_propagation.py. Accessed: 2025-06-30; (lines 75–78).

[43] Lorenzo Perini, Connor Galvin, and Vincent Vercruyssen. 2020. A Ranking Stability Measure for Quantifying the Robustness of Anomaly Detection Methods. In *ECML-PKDD'20*. Springer, 397–408.

[44] Python Software Foundation. 2023. *pickle — Python object serialization*. https://docs.python.org/3/library/pickle.html Python version 3.11.

[45] R Core Team. 2004. R: A Language and Environment for Statistical Computing, Version 2.0.1 - 2.1.0. https://cran.r-project.org/src/base/R-2/ Accessed: 2025-04-23.

[46] Joseph D Romano, Trang T Le, William La Cava, John T Gregg, Daniel J Goldberg, Praneel Chakraborty, Natasha L Ray, Daniel Himmelstein, Weixuan Fu, and Jason H Moore. 2021. PMLB v1.0: an open source dataset collection for benchmarking machine learning methods. *arXiv preprint arXiv:2012.00058v2* (2021).

[47] Scikit-learn. 2025. SKlearn use in the enterprise. Available at: https://scikit-learn.org/stable/testimonials/testimonials.html (Accessed: 2025-07-01).

[48] scikit-learn contributors. 2021. MISC: <= rather than < in tol check (KMeans). https://github.com/scikit-learn/scikit-learn/commit/94d3bd42f863e90fbd7ad3659cee17d8b79d931b Commit, Accessed: 2025-04-23.

[49] scikit-learn contributors. 2021. More stable n_init runs for KMeans. https://github.com/scikit-learn/scikit-learn/pull/20200/files Pull request, Accessed: 2025-04-23.

[50] scikit-learn contributors. 2022. ENH Set Affinity propagation labels if any clusters were found. https://github.com/scikit-learn/scikit-learn/pull/22217/files Pull request, Accessed: 2025-04-23.

[51] scikit-learn contributors. 2023. Fix AffinityPropagation assigning multiple clusters for equal points. https://github.com/scikit-learn/scikit-learn/pull/28121 Pull request, Accessed: 2025-04-23.

[52] Peng Gang Sun, Lin Gao, and Shan Han. 2011. Prediction of human disease-related gene clusters by clustering analysis. *International journal of biological sciences* 7, 1 (01 2011), 61–73.

[53] Laurens Van der Maaten and Geoffrey Hinton. 2012. Visualizing non-metric similarities in multiple maps. *Machine learning* 87, 1 (2012), 33–55.

[54] Wolfgang Vogt and Dorothea Nagel. 1992. Cluster analysis in diagnosis. *Clinical Chemistry* 38, 2 (1992), 182–198.

[55] Ulrike Von Luxburg et al. 2010. Clustering stability: an overview. *Foundations and Trends® in Machine Learning* 2, 3 (2010), 235–274.

[56] Chunqiu Steven Xia, Saikat Dutta, Sasa Misailovic, Darko Marinov, and Lingming Zhang. 2023. Balancing effectiveness and flakiness of non-deterministic machine learning tests. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1801–1813.

[57] Bin Xin, William N. Sumner, and Xiangyu Zhang. 2008. Efficient program execution indexing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 238–248. doi:10.1145/1375581.1375611

[58] Xin Yin, Vincenzo Musco, Iulian Neamtiu, and Usman Roshan. 2019. Statistically Rigorous Testing of Clustering Implementations. In *AITEST 2019*.

[59] Xin Yin, Iulian Neamtiu, Saketan Patil, and Sean T. Andrews. 2020. Implementation-induced Inconsistency and Nondeterminism in Deterministic Clustering Algorithms. In *ICST 2020*.