Vamsa: Automated Provenance Tracking in Data Science Scripts

Mohammad Hossein Namaki
m.namaki@wsu.edu
Washington State University

Avrilia Floratou
avflor@microsoft.com
Microsoft

Fotis Psallidas
fopsalli@microsoft.com
Microsoft

Subru Krishnan
subru@microsoft.com
Microsoft

Ashvin Agrawal
asagr@microsoft.com
Microsoft

Yinghui Wu
yxw1650@case.edu
Case Western Reserve University

Yiwen Zhu
zhu.yiwen@microsoft.com
Microsoft

Markus Weimer
markus.weimer@microsoft.com
Microsoft

ABSTRACT

There has recently been a lot of ongoing research in the areas of fairness, bias and explainability of machine learning (ML) models due to the self-evident or regulatory requirements of various ML applications. We make the following observation: All of these approaches require a robust understanding of the relationship between ML models and the data used to train them. In this work, we introduce the *ML provenance tracking* problem: the fundamental idea is to automatically track which columns in a dataset have been used to derive the features/labels of an ML model. We discuss the challenges in capturing such information in the context of Python, the most common language used by data scientists.

We then present Vamsa, a modular system that extracts provenance from Python scripts *without requiring any changes to the users' code*. Using 26*K* real data science scripts, we verify the effectiveness of Vamsa in terms of coverage, and performance. We also evaluate Vamsa's accuracy on a smaller subset of manually labeled data. Our analysis shows that Vamsa's precision and recall range from 90.4% to 99.1% and its latency is in the order of milliseconds for average size scripts. Drawing from our experience in deploying ML models in production, we also present an example in which Vamsa helps automatically identify models that are affected by data corruption issues.

CCS CONCEPTS

• **Information systems** → **Data provenance**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

data science, provenance, machine learning

KDD '20, August 23–27, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. 2020. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Increasingly, machine learning is used in domains which demand great care and attention to fairness, correctness and reliability [25] such as the financial, medical and manufacturing industries. Of course, we are not the first to recognize this trend. In fact, KDD itself has been host to several workshops [8], tutorials [7, 9] and research contributions [28, 47] in the areas of fairness, bias and explainability of ML models. This year's call for papers specifically asks for contributions in those areas. Considering these and other works, we make a very simple, and somewhat obvious observation: All these approaches require a robust understanding of the model's provenance: What data was used to train the model? Where did the training data originate? How was it processed? These and other questions are usually assumed to have an answer.

We conducted a survey across 7 Big Data companies to better quantify the need for provenance tracking in the ML space. The participants were responsible for cleaning data, developing, or deploying ML models in production. According to 82% of the participants, tracking provenance between data and ML models can be useful in multiple scenarios such as: model debugging (88% of the participants), model sharing (69%), compliance (56%), and fairness (56%). Most of the participants also pointed out that their teams currently spend multiple hours per week trying to identify information related to the data used to train ML models (files, columns used for features, etc.) either manually or using primitive tools.

Projects such as MLflow [11] and Kubeflow [10] make it easy to run *and record* complex ML pipelines. However, these systems do not currently provide a way to manually or automatically track the relationships between data and ML models.

Building upon and going beyond these insights and works, we here present our approach to *automatic* provenance tracking for ML pipelines that seamlessly tracks which columns in a dataset

have been used to derive the features and labels of a ML model. Consider the Python script presented in Figure 1 that was created in the context of the Kaggle Heart Disease competition [3]. The script trains a ML model using a patient dataset from a hospital. The model takes as input a set of features such as Age and Blood pressure, and predicts whether a patient might have a heart disease in the future. A practical provenance tracking system should not only detect that this script trains a ML model but also that the model is trained using the heart_disease.csv dataset and that the columns Target and SSN are not used to derive the model's features. Having this infomation can help detect violations of compliance regulations such as using PII information (e.g., SSN) when training a ML model.

While this step is obvious in terms of the problem definition, it is far from it in realization: ML pipelines are typically authored in Python, which is beloved for its dynamic typing and extensive meta-programming abilities. The absence of declarative semantics makes automated provenance tracking particularly challenging. Moreover, data science is an evolving field as exemplified by the growth of newly available frameworks like PyTorch [14] and popular libraries like scikit-learn [36] still under active development.

We argue that a fully automated provenance system for data science scripts to be usable, should rely on the following important design principles: (1) provide support for unmodified Python scripts (the most common language used by data scientists [4]) and (2) be extensible to accommodate new ML frameworks/libraries. In other words, the system must strike a balance between two relatively conflicting objectives: on the one hand, input from the user (e.g., in the form of logging operations) is not acceptable as it is a time-consuming and potentially error-prone process, on the other hand support for existing and new ML frameworks/libraries is required.

Towards these goals, we make the following contributions:

(1) We present **Vamsa**, to the best of our knowledge, the first automated provenance tracking system for Python data science scripts. Vamsa relies on the aforementioned design principles: (1) it uses a variety of new and existing static analysis techniques that do not make any assumption about the structure of the script and do not require any user input (2) its core modules are agnostic of the ML libraries invoked in the script. To achieve that, Vamsa queries an external knowledge base containing APIs of various ML libraries. It can thus operate on all kinds of data science scripts as long as the knowledge base has been updated to contain the appropriate APIs. This design allows improving coverage by simply adding more ML APIs in the knowledge base without any further code changes in Vamsa or the user scripts.

(2) Using data science scripts from Kaggle [5] and publicly available Python notebooks [43], we perform experiments using $26K$ scripts and verify the effectiveness of Vamsa in terms of coverage and performance. We also evaluate the accuracy on a smaller subset of manually labeled data. Our analysis shows that Vamsa's precision and recall range from 90.4% to 99.1% and its latency is in the order of milliseconds for average size scripts.

(3) We present a real, end-to-end scenario where Vamsa can help debug an ML model deployed in production to predict job slowdowns in clusters of thousands of nodes.

```
1. import catboost as cb
2. from sklearn.model_selection import train_test_split
3. import pandas as pd

4. train_df = pd.read_csv('heart_disease.csv')

# selecting a set of features and specifying the ground truth
5. train_df2 = train_df.iloc[:, 3:].values
6. train_x = train_df2.drop(['ID', 'SSN'], axis=1)
7. train_y = train_df2['Target']

# splitting data to train and validation sets
8. train_x2, val_x2, train_y2, val_y2 = train_test_split(
     train_x, train_y, test_size=0.20)

# initializing and training a model
9. clf = cb.CatBoostClassifier(eval_metric="AUC",iterations=40)
10.clf.fit(train_x2, train_y2, eval_set=(val_x2, val_y2))
```

Figure 1: A data science script written in Python

2 PROBLEM STATEMENT

We now define the problem of *automated ML provenance tracking* that Vamsa targets. In essence, given a data science script, our goal is to identify which columns in a dataset have been used to train a particular ML model by analyzing the script during static analysis time—hence, automatically capturing the relationships between data sources and models at a coarse-grained level. More formally:

ML Provenance Tracking. Given a data science script, our goal set is to find all triples $\langle M, D, C\rangle$: each $M \in \mathcal{M}$ is a constructed machine learning model trained in the script using data sources D (e.g., database tables or views, CSVs, spreadsheets, or external files).[1] In particular, the model is trained using features (and optionally labels) derived from a subset of columns of data sources D, denoted as C. The goal is to identify each trained model M in the script, its data sources D, and columns C from D that were used to train model M.

Example 1: The input to Vamsa is a data science script such as the one in Figure 1. The script reads from heart_disease.csv as a data source D and trains an ensemble of decision trees using catboost [38]. In this script, only a single model was trained. Note that not all the columns of the data source have been used to derive the model's features and labels. To select features, a range of columns $[3, +\infty)$ from D is explicitly extracted, followed by the drop of the columns {SSN, Target}. Similarly, only the Target column was used for labels. Thus, the desired output is a triple $\langle M, D, C\rangle$ with $M = \{\text{clf}\}$ is the variable that contains the trained model, $D = \{\text{heart\_disease.csv}\}$ is the training dataset, and C is the set $[3, +\infty) - \{\text{SSN}\}$. The goal of Vamsa is to analyze the script and produce this output. □

Note that under the static analysis setting the problem is undecidable. In this work, we will not focus on sources of undecidability (e.g., conditionals and loops) because their presence in data science scripts is minimal [39] (Section 9 discusses future directions to account for such cases). Finally note that, in this work, we focus on scripts written in Python because this is the major language currently used by data scientists [4, 13, 39]. The principals behind our techniques, however, naturally generalize to other languages as well.

[1] Note that there are also data science scripts that do not perform any model training but provide other functionality (e.g., visualization, optimization, etc.) In this work, we focus on data science scripts that include statements that train ML models as our goal is to capture the relationships between data sources and generated ML models.

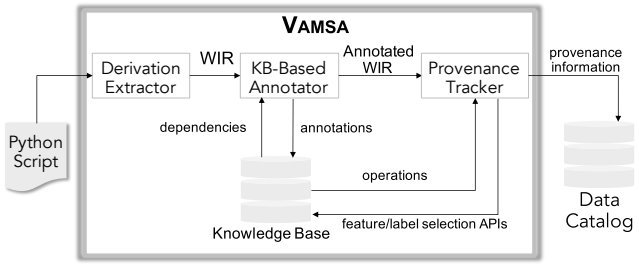


Figure 2: Vamsa Architecture.

3 VAMSA ARCHITECTURE

Vamsa is a system designed to address the ML provenance tracking problem as described in the previous section. To do so, we have built Vamsa based on the following two design principles:

1. Support for unmodified Python scripts: Unlike other works that require the user to add logging information in their script using appropriate APIs [11, 32], our goal was to build a system that does not require any user input or changes to the user code as it is a manual and error-prone process. Moreover, there is already a large corpus of existing production scripts and adjusting them to track provenance information might not be feasible.

2. Ability to incorporate new ML libraries and frameworks: Data science is an evolving field, with new frameworks coming up and existing frameworks evolving their APIs. An ML provenance tracking system should be designed to accommodate all users and their needs and should not be restricted to a specific set of ML libraries. Thus, the algorithms used by the system should be agnostic of the ML frameworks invoked in the script.

Building such a system is challenging. On the one hand, user input in any form is not acceptable. This is because: (1) user input is a manual process–hence, expensive and error-prone—and (2) data scientists that author such scripts are not proficient with neither provenance tracking nor the, often limited, provenance tracking APIs. On the other hand, support of various kinds of ML libraries that data scientists use is needed. To strike a balance between these two relatively conflicting objectives, Vamsa: (1) analyzes Python scripts using existing and novel static analysis techniques that are agnostic of the particular ML libraries invoked (or their different versions) but (2) relies on an external knowledge base of APIs to collect semantic information about the script when needed.

Figure 2 illustrates the architecture of Vamsa. At a high-level, Vamsa performs static analysis on the input Python script to determine the relationships between all the variables in the script, followed by an annotation phase that assigns semantic information to the variables in the script. During this phase, Vamsa queries the external knowledge base to obtain information about the semantics of various ML operations. It then uses a generic provenance tracking algorithm that extracts the feature set for all the ML models trained in the script and stores this information in a central catalog that can be accessed by various provenance applications.

More specifically, Vamsa processes data science scripts with the following three major modules: the *Derivation Extractor*, the *KB-based Annotator*, and the *Provenance Tracker*. Next we give a brief overview of these components, and point forward to individual sections for in-depth discussion on each component.

(1) The *Derivation Extractor* generates a workflow intermediate representation (WIR) of the script by extracting major workflow elements including imported libraries, variables, functions, and their dependencies using standard static analysis techniques (Section 4).
(2) The *KB-based Annotator* annotates variables in WIR based on their roles in the script (e.g., features, labels, and models). To this end, it uses our generic annotation algorithm and a pluggable knowledge base that contains information about the various APIs of different ML libraries. Through the knowledge base, we are able to declaratively introduce semantic information for operations in the Python script. This design allows us to operate on all kinds of Python libraries (ML or otherwise) as long as the appropriate APIs are included in the knowledge base. It also allows users to improve coverage by simply adding the APIs of more libraries in the knowledge base, without having to modify their code or the components of Vamsa. (Section 5)
(3) The *Provenance Tracker* infers a set of columns that were explicitly included in or excluded from the features/labels by using the annotated WIR and consulting the knowledge base. The Provenance Tracker is able to operate in both supervised and unsupervised learning settings. In the former case it tracks both features and labels, while in the latter it tracks only features. (Section 6)

4 DERIVATION EXTRACTOR

As discussed previously, the derivation extractor uses standard static analysis techniques to parse the Python script, build a *workflow model* which captures the dependencies among the elements of the script including imported libraries, input arguments, operations that change the state of the program, and the derived output variables. This model is captured in a *workflow intermediate representation* (WIR). In this section, we will not analyze in depth these techniques, as they rely on well-known concepts [48], but will provide important notation and background required for subsequent components. In [35] we present in detail these techniques.

Operations. We denote the set of all variables in the data science script as V. An operation $p \in P$ operates on an ordered set of input variables I to change the state of the program and/or to derive an ordered set of output variables O. An operation may be called by a variable, denoted as caller c. While an operation may have multiple inputs and outputs, it has at most one caller.

Example 2: In Figure 1, the import statements, read_csv(·) in line 4, attribute values in line 5, CatBoostClassifier(·) in line 9, and fit(·) in line 10 are examples of operations. Consider the fit(·) operation: it is invoked by the clf variable and takes three arguments namely, features and labels, and an evaluation set. While fit(·) does not explicitly produce an output variable, it changes the state of the variable clf. □

Provenance relationship. An invocation of an operation p (by an optional caller c) depicts a *provenance relationship* (PR). A PR is represented as a quadruple (I, c, p, O), where I is an ordered set of input variables, (optional) variable c refers to the caller object, p is the operation, and O is an ordered set of output variables that was derived from this process. A PR can be represented as a labeled directed graph, which includes (1) a set of *input edges* (labeled as *'input_edge'*), where there is an input edge (v, p) for each $v \in I$, (2)

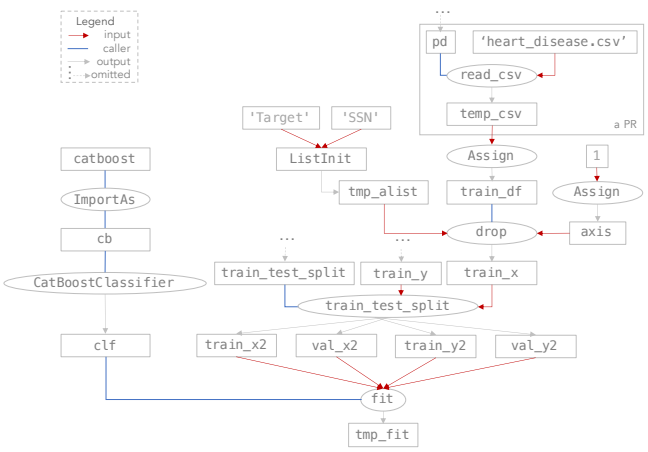


Figure 3: An example WIR.

a *caller edge* (labeled as *'caller_edge'*) (c, p) if p is called by c, and (3) a set of *output edges* (labeled as *'output_edge'*), where there is an output edge (p, v) for each $v \in O$. For consistency, we create a temporary output variable for the operations that do not explicitly generate one.

Example 3: Consider line 4 in Figure 1 where the CSV file 'heart_disease.csv' is read. The corresponding PR is depicted in Figure 3 (rectangle in top right) and corresponds to the quadruple (I, c, p, O) where $I = \{\langle$heart_disease.csv$\rangle\}$, $c =$ pd, and $p =$ read_csv. Finally, we create a temporary variable tmp_csv and set $O =$ {tmp_csv} to used as the input to other PRs. □

Workflow Intermediate Representation. PRs are composed together to form a WIR G, which is a directed graph that represents the sequence and dependencies among the extracted PRs. The WIR is useful to answer queries such as: "Which variables were derived from other variables?", " What type of libraries and modules were used?," and "What operations were applied to each variable?". More formally, a WIR is a directed bipartite graph $G = (V \cup P, E)$ with vertices $V \cup P$ and edges $E \subseteq (V \times P) \cup (P \times V)$. Each edge has a type drawn from the set: {input_edge, output_edge, caller_edge}.

Example 4: Figure 3 illustrates a fraction of the WIR generated for the script of Figure 1. The variables and operations are represented by rectangles and ovals, respectively. The caller; input; and output edges are marked in blue; red; and black color, respectively. Consider the operation fit. One can tell the following from the WIR: 1) it is called by variable clf; 2) it has two ordered input variables train_x2 and train_y2; and 3) a temporary variable, denoted as tmp_fit, was created by Vamsa as its output. □

Vamsa generates workflows using standard data flow analysis techniques [48]. More specifically, it first parses the script to obtain a corresponding *abstract syntax tree (AST)* [6, 12] representation. Then, it identifies the relationships between nodes of the AST to generate PRs. Finally, it composes the generated PRs into the WIR [35].

5 KB-BASED ANNOTATOR

The generated WIRs capture the dependencies among the variables and operations in a script, as we discussed in the previous section.

Library	Module	Caller	API_Name	Inputs	Outputs
catboost	NULL	NULL	CatBoostClassifier	eval_metrics: hyperparameter	model
catboost	NULL	model	fit	features labels eval_set: validation sets	trained model
sklearn	model_selection	NULL	train_test_split	features labels test_size: testing ratio	features validation features …

Table 1: Example of facts in Vamsa knowledge base

Unfortunately, WIRs alone do not provide *semantic information* such as the role of variables in a script (e.g., ML model or features) or the type of objects (e.g., CSV file or dataframe). To address the ML provenance tracking problem, however, this semantic information needs to be included in the output provenance information. To this end, the goal of the KB-Based Annotator, that we focus on in this section, is to annotate variables in WIRs with semantic information.

Finding the role of each variable in a WIR is a challenging task for multiple reasons. First, one cannot accurately deduce the type of inputs and outputs of an operation by only looking at the name of the operation. This is because different ML libraries may use the same operation name for different tasks. Second, even in the same library, an operation may accept different number of inputs or provide different outputs. For example, the fit function of sklearn [36] can accept one or two inputs depending on the task (e.g., one for clustering or two for classification or regression). Finally, some variables (e.g., feature sets) are hard to semantically annotate early on. For instance, we cannot decide whether the returned dataframe from a read_csv call of pandas is a training set, before in itself (or after preprocessing steps) becomes input to a training function.

Besides the technical challenges outlined above, a semantic annotation framework to be usable across various data science scripts must be: (1) compatible with the various ML libraries and their different versions, and (2) extensible to accommodate new libraries.

To this end, we propose a generic annotation algorithm (Section 5.2) that is *agnostic* of the underlying ML libraries used in the script by querying an external knowledge base of ML APIs (KB) when semantic information is needed (Section 5.1) and is able to propagate annotations across different elements of the WIRs.

Note that there is already a substantial effort from the Python community to annotate various libraries and their external APIs with type information for static analysis purposes [15]. As these initiatives mature, the population of the KB that Vamsa queries will become straightforward. In our current prototype, and similar to other efforts for KB population [27], the construction of the KB is manual. As we show in our experiments, however, our manual (yet minimal) KB results in large coverage on big collections of data science scripts. This is primarily because many data science scripts rely on similar coding patterns [39].

5.1 Knowledge Base of ML APIs

The KB contains information about ML libraries and APIs including, but not limited to, library and operation names; versions; modules; as wells as types of inputs and outputs of operations.

Example 5: Table 1 shows three tuples in our KB that are utilized by the annotation algorithm to identify the variables that correspond to models and features in the script of Figure 1. The second tuple shows that when the operation fit is called via a model constructed

Algorithm Annotation

Input: WIR G and knowledge base KB.
Output: Annotated WIR G^+.
1. Find the Import process nodes in G as the seed set $\mathcal{S}$;
2. **for each** $v_s \in \mathcal{S}$ **do**
3. Extract library L and module L';
4. Starting from v_s, follow a DFS forward traversal on PRs:
5. **for each** seen PR = $\langle I, c, p, O\rangle$ **do**
6. Obtain annotation of $v_i \in I$ and $v_o \in O$ by invoking KB(L, L', c, p)
7. **for each** annotated $v_i \in I$ **do**
8. Starting from v_i, follow a DFS backward traversal on PRs:
9. **for each** seen PR = $\langle I, c, p, O\rangle$ **do**
10. Obtain annotation of $v_i \in I$ by invoking KB(O, p)
11. **return** G^+;

Figure 4: Annotation algorithm

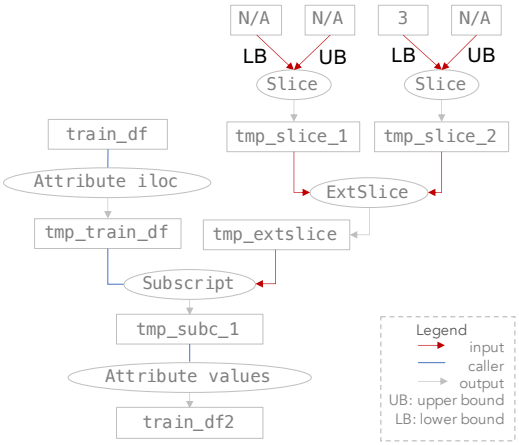


Figure 5: WIR with Subscript operation

by catboost library, its first and second input are features and labels, respectively. It also accepts the validation sets as input. The output of the operation is a trained model. □

To facilitate the annotation of WIR variables, KB supports two types of queries. The first one denoted as KB(L, L', c, p) takes as input the name of a library L, module L', caller type c, and operation p and returns a set of user-defined annotations that describe the role and type for each input/output of operation p. The second one denoted as KB(O, p) obtains the annotations of the input variables of operation p given the annotations of its output O.

5.2 Annotation Algorithm

The annotation algorithm traverses the WIR and annotates its variables by querying the KB when needed. After each annotation, new semantic information about a WIR node is obtained that can be used to enrich the information associated with other WIR variables, as is typical in the analysis of data flow problems [48]. The propagation of semantic information is achieved through a combination of forward and backward traversals of the WIR.

The algorithm (Figure 4) starts by finding a set of PRs with p = Import as a seed set $\mathcal{S}$ for upcoming DFS traversals (line 1). These PRs contain the information about imported libraries and modules in the Python script. For each $v_s \in \mathcal{S}$, the algorithm extracts the library name L and the potentially utilized module L' (line 3). It then initiates a DFS traversal (line 4) that, starting from v_s, traverses the WIR in a forward manner (i.e., by going through the outgoing edges). For each seen PR, it obtains the annotation information for both of its inputs I and outputs O by querying the knowledge base (lines 5-6) as described in the previous section.

If a new annotation was found for an input variable $v_i \in I$, the algorithm initiates a backward DFS traversal. As the input variable v_i can be the output of another PR, for new information discovered for v_i, we can propagate this information to other PRs in which v_i is their output. In particular, starting from v_i, the algorithm traverses the WIR in a backward manner (i.e., by going through the incoming edges) (line 8). During the backward traversal, the KB is used to obtain information about the inputs of an operation given its already annotated output. In each initiated DFS traversal, each edge is visited only once. The algorithm terminates when we cannot obtain more information from initiating more forward or backward traversals [35].

Example 6: Operating on the WIR of Figure 3, the annotation algorithm initializes the seed set $\mathcal{S}$ with one import operation and sets L = catboost and L' =⊥. Once it visits the p = CatBoostClassifier operation, it queries the KB to obtain the annotation of its output. Given, L, L', c = catboost and p, the KB annotates clf as a model. Since there exists no input edge for the CatBoostClassifier node in this WIR, no backward traversal is initiated. The algorithm moves forward and visits the fit function. It queries the KB with the same L and L', but updated c = *model* and p = fit. The algorithm annotates the output of fit as trained model and then stops the forward propagation since there are no more outgoing edges in the node. However, at this time, KB successfully annotated the train_x2 and train_y2 as the features and labels, respectively. Thus, two backward traversals are started to propagate this information as much as possible to the previous nodes in the WIR. Let us follow the DFS that was started from train_x2. By visiting the train_test_split node, the algorithm annotates train_x as features. Similarly, it back-propagates the new annotation to train_df as the caller of drop operation. The algorithm continues until we cannot obtain more annotation information. □

6 PROVENANCE TRACKER

We next introduce the Provenance Tracker component of Vamsa. The provenance tracker takes as input the WIR annotated (i.e., the output of the algorithm in Figure 4) and its goal is to automatically detect the subset of columns in a data source that was used to train a ML model and, as such, forms the overall output of Vamsa.

To identify the columns, we need to investigate the operations in the annotated WIR that are connected to variables that contain features and labels in their annotation set. Note that there are various operations that take features (or labels) as their caller or input, and may apply transformations on them (e.g., drop a set of columns, select a subset of rows upon satisfaction of a condition, or copy into other variables) that the Provenance Tracker needs to account for.

To this end, we enrich our KB with a new table to guide the Provenance Tracker. The new table follows the structure of Table 1. This time, however, each entry is annotated further with two attributes: column_exclusion and traversal_rule. The column_exclusion attribute is set to True, if the operation explicitly excludes columns (e.g., drop and delete in Pandas), and False otherwise. The traversal_rule is a function specifying how to start a backward traversal from the node's input edges in order to identify a set/range of indices/column names. Example 7 provides an example to clarify these notions. Finally, note that we query this table by invoking $\mathsf{KB}_C(p)$ where p is the operation. The query returns $\emptyset$, if there is no matching entry in the KB, or the column_exclusion and traversal_rule, otherwise.

Example 7: Figure 5 is another fraction of the WIR that was generated from line 5 of the script in Figure 1 that includes a Subscript operation. The statement in line 5 keeps all the rows but only includes the columns from index 3 to the last index in the dataset. One can find the set of included columns by traversing backward the nodes following the input edge of the Subscript operation and reaching the constant values connected with the Slice operations. The traversal rule associated with the Subscript operation in our KB indicates that the input edge of this node must be followed in a backward manner to eventually reach the selected columns. Note that this is the case for all WIRs that contain this operation. □

The overall provenance tracking algorithm is illustrated in Figure 6. The algorithm takes as input the annotated WIR G^+ and the KB, and returns two column sets: (1) columns from which features were explicitly derived (denoted as *inclusion set* C^+) and (2) columns that are explicitly excluded from features (denoted as *exclusion set* C^-). The algorithm scans each PR to find the ones with a variable that has been annotated as features and an operation which can be used for feature selection based on the information stored in the KB (line 2-3). Note that the exact same logic is used to derive inclusion and exclusion sets for labels, as opposed to features, and we combine the two in a single pass over the PRs of the WIR G^+.

A core component of the algorithm is the GuideEval operator (shown in Figure 6) that starts a guided traversal of the WIR based on the information in the KB. For each of the selected PR, the GuideEval operator queries the KB and obtains the corresponding column_exclusion flag and traversal_rule (line 1). The operator checks if this PR contains constant values in its input set (line 2). If so, it incorporates the discovered constant values/range of column indices into the inclusion/exclusion sets. In case the PR does not directly contain the columns, the GuideEval operator follows the traversal_rule to obtain a new PR on G^+ (line 7) that needs to be evaluated. It then calls the GuideEval operator again for this PR (line 8).

Example 8: Continuing with example 7, the provenance tracking algorithm finds the drop operation with a caller that was annotated as features (Figure 3) and thus invokes the GuideEval operator.Based on the information in the KB, we know that the operation was used for feature exclusion. Thus, the algorithm follows the traversal rule to perform a backward traversal from its the operation's input edge until it finds the constants 'Target', and 'SSN'. These two columns are then added to the exclusion set. When the feature tracking algorithm finds the Subscript operation (Figure 5) in the annotated WIR, it invokes the GuideEval operator again. The operator only

Algorithm PTracker

Input: Annotated WIR G^+, knowledge base KB.
Output: Column inclusion set C^+, column exclusion set C^-.
1. $C^+ := \emptyset$; $C^- := \emptyset$;
2. **for each** PR **in** PRs **do**
3. **if** it has a variable that was annotated as features or labels **and** $\mathsf{KB}_C(p) \neq \emptyset$
4. GuideEval(PR, G^+, KB, C^+, C^-) ;
5. **return** C^+, C^-;

Operator GuideEval(PR, G^+, KB, C^+, C^-)

Input: Visited PR, annotated WIR G^+, knowledge base KB, column inclusion set C^+, column exclusion set C^-.
Output: Updated C^+, C^-.
1. condition, column_exclusion, traversal_rule = $\mathsf{KB}_C(p)$;
2. **if** PR has constant inputs cnst **then**
3. **if** column_exclusion =True **then**
4. $C^- := C^- \cup$ cnst;
5. **else** $C^+ := C^+ \cup$ cnst;
6. **return** C^+, C^-;
7. Obtain new PR on G^+ based on traversal_rule;
8. GuideEval(PR, G^+, KB, C^+, C^-);

Figure 6: Provenance tracking algorithm

obtains the corresponding traversal rule from the KB and initiates a backward traversal starting from the input edge of the Subscript operation. A similar process is performed when the GuideEval operator visits ExtSlice and Slice nodes. Using the traversal rule for Slice, for instance, the algorithm looks for a range of columns with lower bound (respectively upper bound) that can be found by traversing the appropriate input edges of the Slice node (see Figure 5). □

7 EXPERIMENTAL EVALUATION

We now evaluate Vamsa on large corpora of Python scripts and provide an analysis of our experimental results. We also present an end-to-end scenario that we encountered in production to better show how Vamsa can facilitate model debugging.

7.1 Experimental Settings

Datasets. To evaluate Vamsa on a variety of data science scripts, we downloaded a large set of Python scripts from two data sources: (1) a corpus of Python notebooks published in 2017 that was crawled from Github [43] (NTBK dataset) and (2) a set of Python scripts that we downloaded via the public Kaggle API [5](Kaggle dataset). We filtered these corpora to account only for scripts that include import statements, do not have syntax errors, and are compatible with Python 3 (the Python version that Vamsa's implementation currently targets). After applying these filters, we kept the scripts that included strings (e.g., fit) that popular ML frameworks (e.g., scikit-learn, XGBoost [1], and LightGBM) in our KB use to train ML models. We selected these ML frameworks because they are among the most common for training ML models [39]. Note that it is easy to extend to other libraries by just populating the KB (no code changes are required). The resulting datasets are denoted as NTBK ($24.8K$ scripts) and Kaggle ($1.2K$) scripts.

Experimental methodology. A challenge when evaluating Vamsa with such large-scale corpora is to determine the correctness of the output. Unfortunately, due to the novel nature of ML

Dataset	Feature Exclusion		Feature Inclusion		Label Inclusion		Annotation Precision	
	Precision	Recall	Precision	Recall	Precision	Recall	Model	Train Dataset
Kaggle	99.11%	97.72%	91.37%	92.89%	95.47%	95.67%	100%	99.33%
NTBK	94.83%	96.58%	90.54%	94.08%	90.36%	90.78%	100%	98.66%

Table 2: Accuracy of Vamsa on the labeled datasets

provenance tracking, there is no public benchmark available. The brute force approach would be to manually go over the corpus and determine the relationships between ML models and data sources so that we can evaluate Vamsa's output. Since this is not feasible at the scale we are operating, we decided to perform two classes of experiments. First, we select a small subset of scripts for which we manually extract the provenance information (ground truth) and evaluate the accuracy of Vamsa on those. The second class of experiments is performed on the large corpus. The goal is to evaluate the coverage of the system, defined as how often Vamsa extracts the provenance information. We also evaluate Vamsa's efficiency in terms of latency.

Hardware and software configuration. We conducted our experiments on a Linux machine powered by an Intel 2.30 GHz CPU with 8 GB of memory. For all the experiments we used Python 3.7.2.

7.2 Experiments with Labeled Datasets

These experiments evaluate the accuracy of Vamsa on a set of Python scripts for which we have manually extracted the relationship between data sources and ML models. From each of the Kaggle and NTBK datasets, we randomly selected 150 scripts, ensuring that Vamsa can produce output for all the selected scripts. We evaluate the accuracy of Vamsa for both features and labels under column exclusion and inclusion using two metrics: precision and recall. The precision shows the proportion of discovered included/excluded columns that were truly included/excluded columns. The recall shows the proportion of the true included/excluded columns that were discovered by Vamsa to the actual included/excluded columns.

We further investigate how often Vamsa correctly identifies which variables correspond to ML models and which to training datasets as this is a prerequisite for correctly identifying features and labels. To this end, we also report results that show the precision of the annotation phase (for both models and training datasets).

Table 2 shows the results on the two datasets. For each metric, we report the average values obtained over the 150 scripts of the dataset. As shown in the table, Vamsa achieves high precision and recall values for all the tasks evaluated. Overall, we can make the following observations:

(1) When Vamsa identifies a model, its training dataset, and the corresponding features, the output is highly reliable.

(2) Vamsa reported models 100% accurately and made a few mistakes in detecting their training datasets. We further investigated these scripts and found that the data scientists appended the testing data to the training data in order to perform global value transformations. The merged test data then got separated via a slicing operation immediately before training. Vamsa's annotation algorithm was not able to follow this operation, i.e merge followed by split, and mistakenly identified the testing dataset as the training dataset.

(3) Vamsa detects column exclusion sets slightly better than column inclusion ones. This is because, for column exclusion, data scientists typically use a set of specific APIs such as drop and pop, del which can be tracked more easily. Note that we did not evaluate column exclusion for the labels as none of the scripts used these APIs for label selection.

	Kaggle	NTBK
Derivation Extractor	89.69%	97.08%
KB-based Annotator (Model)	91.88%	96.93%
KB-based Annotator (Train Dataset)	85.15%	88.12%
Provenance Tracker	80.85%	74.48%

Table 3: Vamsa coverage in large-scale evaluation

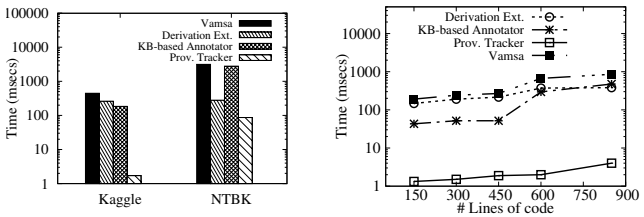
7.3 Large-scale Experiments

In these experiments, we use a large corpus of Python scripts (full NTBK and Kaggle datasets). The goal is to evaluate the coverage of various components of Vamsa as well as the efficiency of the system. We also present a detailed analysis of the cases where Vamsa was not able to produce an answer.

Derivation Extractor. First, we evaluate the coverage of Vamsa on generating the workflow intermediate representation. Table 3 shows the results. The few cases where Vamsa is not able to produce a WIR are mainly due to Vamsa's current implementation. In particular, we have not yet covered certain constructs in the Python grammar such as DictComp, SetComp, and JoinedStr. However, we note that incorporating these constructs is solely a matter of extending the implementation and does not require any change in Vamsa's design.

KB-based Annotator. We investigate how often the annotation algorithm identifies ML models and training datasets. Table 3 shows the percentage of the cases where the Annotator can annotate at least one variable as a model and one other variable as a training dataset. As shown in the table, Vamsa can report model and training datasets annotations for 91.88% and 85.15% of the scripts in the Kaggle dataset. The coverage is a bit higher for the NTBK dataset.

To better understand the cases where Vamsa was not able to perform the annotation, we examined the cases where a model was not found. We identified the following reasons for the failure. (1) Some scripts called APIs commonly used for training models (e.g., fit), to perform other operations (e.g., feature extraction). In these cases, the KB-Based Annotator correctly did not report any model. (2) In a few scripts, the statements used to train a model were commented out. This was not detected by our pre-processing pipeline and thus these scripts were falsely included in the final dataset. (3) Some scripts imported modules using the * notation. In these cases, Vamsa could not relate the import statement to the API calls. (4) In a few other scripts, data scientists imported a module with an alias name and used the alias when invoking the APIs.



(a) Average latency breakdown.

(b) Average latency while varying the script size on the Kaggle dataset.

Figure 7: System Efficiency

Vamsa's implementation does not currently cover such cases. We are continuously addressing these issues in our implementation.

We have also explored the cases where the KB-Based Annotator could not find a training dataset. In our analysis, we found two main reasons why. (1) In some scripts, hard-coded data (e.g., a large numpy array) was used as the training data. (2) Some APIs are not presented in our KB and thus the annotation algorithm is not able to perform back propagation. We note, however, that these cases could be simply covered by extending our KB with more APIs. We further note that providing the ability to increase coverage by enhancing the KB was one of the major requirements from external Vamsa users and, as such, became central to our design.

Provenance Tracker. Table 3 shows the percentage of the cases where the Provenance Tracker can identify at least one set of features. Note that the Provenance Tracker is invoked only if the KB-Based Annotator can identify a model and its corresponding training dataset. We thus expect the coverage of this component to be bounded by the coverage of the KB-Based Annotator.

As shown in Table 3, Vamsa reports a non-empty column set for 80.85% of the scripts in Kaggle dataset and 74.48% of the scripts in the NTBK dataset. We have also analyzed the cases that Vamsa could find both a model and a training dataset but did not discover the column set. We identified three main reasons behind this behavior: (1) In some scripts, the columns have not been selected explicitly but based on a condition on their values (e.g., a column is in the feature set iff it contains at least N non-zero values). (2) Similar to the KB-Based Annotator, some scripts required new rules to be added into the KB for the Provenance Tracker to operate correctly. (3) Some scripts did not include any feature selection operations and thus Vamsa did not produce any output.

7.4 System Efficiency Experiments

In this set of experiments, we evaluate the latency of each component of Vamsa as well as the end-to-end latency. Note that low latency is an important requirement from Vamsa especially due to the many user scenarios where provenance needs to be extracted from many scripts in a limited amount of time (e.g., for compliance, auditing, debugging in production, or security applications).

Breaking down the latency. We present the individual latencies of the Derivation Extraction, KB-based Annotator, and Provenance Tracker as well as the end-to-end latency. Figure 7(a) shows the average latency results. We observe that the time spent by each component is negligible on both datasets and on average is in the order of milliseconds (ms). One aspect that is not reflected in Figure 7(a) is the break down of the latencies of the Derivation Extractor tasks (i.e., AST generation, PR generation, and WIR composition). For these tasks, we observed that most of the time is spent in AST generation and WIR composition. In particular, for the Kaggle dataset, AST generation takes 11.1 ms, PR generation takes 34 ms, and WIR composition takes 215.7 ms. The corresponding numbers for the NTBK dataset are: 11.1 ms, 41.1 ms, and 227 ms.

Latency of Vamsa while varying the lines of code (LOCs). We further evaluate the latency of each component in isolation and end-to-end as the LOCs in the script vary, which is a common metric in benchmarking static analysis tools. Figure 7(b) shows the latency of the components as the script size varies for the Kaggle dataset. We see that increasing the number of LOCs in a Python script naturally increases the latency of all Vamsa components, reaching an average 0.9s end-to-end latency (3.9s max) for scripts with (600, 900] LOCs.

7.5 Real Use Case: Model Debugging

We now present a real scenario where Vamsa can help automatically identify ML models that are affected by data corruption issues. Our internal Big Data platform is comprised of hundreds of thousands of machines, serving over half a million jobs daily. A large fraction of these jobs are recurring and thus have been tuned very carefully. However, some recurring jobs inevitably experience slowdowns. To understand the root cause of these slowdowns, we deploy an ML-based system that looks at various jobs parameters and runtime characteristics and identifies the cause of the slowdown [46].

Developing such a system involves multiple teams: the platform team that collects and manages the job logs, the data scientists that look into this data and develop ML models, and the team that deploys the models in production and monitors their performance.

In one case, an engineer from the platform team identified a data corruption issue in one of the columns in the dataset. Now, she had to identify the upstream teams that use this data and notify them on the corruption issue. This is clearly a time-consuming and error-prone process given that these job logs are accessed by hundreds of teams every day. In this case, this column has been used to train the ML model responsible for root cause analysis of jobs slowdowns.

We ran Vamsa providing the script that the data scientist used to train the model and Vamsa identified the correct set of columns (included the corrupted one) that was used to train this model. Using such a system to collect provenance information on the data scientists' scripts across the organization can help automatically notify the responsible teams on data corruption issues so that they can quickly take the appropriate steps to fix their models.

8 RELATED WORK

We describe relevant related work from three areas:

Model management systems. There has been an emerging interest in systems that manage the lifecycle of ML models [24, 31, 33, 44, 45, 49, 50]. ModelDB [49] stores trained models to enable querying of metadata and artifacts by exposing a logging API for a specific set of libraries. ModelHub [33] is a fine-grained versioning system for ML artifacts with a focus on deep learning. Amazon's ML experiments system [44] tracks provenance of ML experimentation data. This system automated the provenance extraction for SparkML [30] and scikit-learn [36] pipelines whenever a logical abstraction of operations is available. ProvDB [31] focuses on efficiently storing

and querying ML provenance data. In contrast to these systems, Vamsa focuses on the tracking task and, as such, is complementary to systems that focus on other management tasks (e.g., storing or querying). Furthermore, Vamsa introduces design principles (i.e., does not require developers to modify their code, operates on top of any library, and tracks provenance at static analysis time) that are important for end-users, yet not provided by prior systems.

Provenance in databases. Capturing provenance on top of SQL queries is an extensively studied area [19, 26] driven by an immense amount of applications [2, 20–22, 34, 40–42, 51]. In contrast to this line of work, Vamsa is designed to capture provenance in data science scripts as opposed to SQL queries. As such, Vamsa introduces provenance capture techniques tailored to the semantics of imperatively specified data science logic (in Python). Furthermore, Vamsa treats models and datasets as first-class citizens of the captured provenance information, all the while exposing (and semantically annotating) data flows of data science logic. This allow us to better drive applications in the data science space.

Workflow management systems. Workflow management systems collect and manage the provenance information to enable experiment sharing [23]. Closer to our work are the Starflow [17, 18], noWorkflow [37], and YesWorkflow [29] systems. StarFlow statically analyzes a Python program to build provenance traces at the level of functions. noWorkflow captures various forms of provenance information by analyzing scripts during execution. Starflow and YesWorkflow require modifications to the users' script, while noWorkflow handles unmodified programs. Our work differs from this line of work in a vein similar to the other two lines of work discussed above. We note, however, that an interesting direction is to extend our techniques to account for runtime information.

9 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the problem of ML provenance tracking—a fundamental type of provenance information that enables multiple applications including model debugging; compliance; and model maintenance. Our evaluation shows that it is indeed possible to recover this type of provenance with a very high precision and recall across large corpora of Python scripts.

There are many areas to further explore. First, incorporating runtime information can be useful for cases that are undecidable under our static analysis setting. Second, identifying finer-grained provenance information between data sources and ML models (e.g., partitions of a data source were used for training) can better assist upstream applications. Finally, automatically populating the knowledge base is also an important direction for future work.

REFERENCES

[1] Xgboost. https://xgboost.readthedocs.io/en/latest/index.html, 2014.
[2] EU GDPR Regulations. https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules/eu-data-protection-rules_en, 2018.
[3] Kaggle Heart Disease. https://www.kaggle.com/ronitf/heart-disease-uci, 2018.
[4] Kaggle survey. https://www.kaggle.com/kaggle/kaggle-survey-2018, 2018.
[5] Official Kaggle API. https://github.com/Kaggle/kaggle-api, 2018.
[6] Abstract syntax trees. https://docs.python.org/3/library/ast.html, 2019.
[7] Explainable AI in Industry. https://sites.google.com/view/kdd19-explainable-ai-tutorial, 2019.
[8] Explainable AI/ML (XAI) for Accountability, Fairness, and Transparency. https://xai.kdd2019.a.intuit.com/, 2019.
[9] Fairness-Aware Machine Learning: Practical Challenges and Lessons learned. https://sites.google.com/view/kdd19-fairness-tutorial, 2019.
[10] Kubeflow. https://www.kubeflow.org/, 2019.
[11] Mlflow. https://github.com/mlflow/mlflow/, 2019.
[12] Python AST docs. https://greentreesnakes.readthedocs.io/en/latest/, 2019.
[13] Python language. https://towardsdatascience.com/programming-languages-for-data-scientists-afde2eaf5cc5, 2019.
[14] PyTorch. https://pytorch.org/, 2019.
[15] Typeshed. https://github.com/python/typeshed, 2019.
[16] Vamsa. aka.ms/vamsa, 2020.
[17] E. Angelino et al. Provenance integration requires reconciliation. In *TaPP*, 2011.
[18] E. Angelino, D. Yamins, and M. Seltzer. Starflow: A script-centric data analysis environment. In *IPAW*, 2010.
[19] J. Cheney et al. Provenance in databases: Why, how, and where. *TRDB*, pages 379–474, 2009.
[20] L. Chiticariu, W. C. Tan, and G. Vijayvargiya. Dbnotes: A post-it system for relational databases based on provenance. In *SIGMOD*, pages 942–944, 2005.
[21] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *TODS*, 25(2):179–227, 2000.
[22] D. Deutch, N. Frost, and A. Gilad. Provenance for natural language queries. *PVLDB*, 10(5):577–588, 2017.
[23] J. Freire and M. Anand. Provenance in scientific workflow systems. *IEEE Data Engineering Bulletin*, 2007.
[24] R. Garcia et al. Context: The missing piece in the machine learning lifecycle. In *KDD CMI Workshop*, 2018.
[25] T. Gebru et al. Datasheets for datasets, 2018.
[26] R. Ikeda and J. Widom. Data lineage: A survey. Technical report, Stanford InfoLab, 2009.
[27] Z. Ives et al. Dataset relationship management. In *CIDR*, 2019.
[28] M. R. Lee and M. Shen. Winner's Curse: Bias Estimation for Total Effects of Features in Online Controlled Experiments. In *KDD '18*, 2018.
[29] T. McPhillips et al. Yesworkflow: A user-oriented, language-independent tool for recovering workflow information from scripts. *IJDC*, pages 298–313, 2015.
[30] X. Meng et al. Mllib: Machine learning in apache spark. *JMLR*, pages 1235–1241, 2016.
[31] H. Miao and A. Deshpande. Provdb: Provenance-enabled lifecycle management of collaborative data analysis workflows. *IEEE Data Eng. Bull.*, pages 26–38, 2018.
[32] H. Miao et al. Modelhub: Deep learning lifecycle management. In *ICDE*, 2017.
[33] H. Miao et al. Towards unified data and lifecycle management for deep learning. In *ICDE*, 2017.
[34] M. H. Namaki et al. Answering why-questions by exemplars in attributed graphs. In *SIGMOD*, pages 1481–1498, 2019.
[35] M. H. Namaki et al. Vamsa: Tracking provenance in data science scripts (technical report). *arXiv preprint arXiv:2001.01861*, 2020.
[36] F. Pedregosa et al. Scikit-learn: Machine learning in python. *JMLR*, pages 2825–2830, 2011.
[37] J. F. Pimentel et al. noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *VLDB*, 2017.
[38] L. Prokhorenkova et al. Catboost: unbiased boosting with categorical features. In *NIPS*, 2018.
[39] F. Psallidas et al. Data science through the looking glass and what we found there. *arXiv preprint arXiv:1912.09536*, 2019.
[40] F. Psallidas and E. Wu. Provenance for interactive visualizations. 2018.
[41] F. Psallidas and E. Wu. Smoke: Fine-grained lineage at interactive speed. *VLDB*, pages 719–732, 2018.
[42] E. D. Ragan, A. Endert, J. Sanyal, and J. Chen. Characterizing provenance in visualization and data analysis: an organizational framework of provenance types and purposes. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):31–40, 2016.
[43] A. Rule, A. Tabard, and J. D. Hollan. Exploration and explanation in computational notebooks. In *CHI*, page 32, 2018.
[44] S. Schelter et al. Automatically tracking metadata and provenance of machine learning experiments. In *Machine Learning Systems workshop at NIPS*, 2017.
[45] S. Schelter et al. On challenges in machine learning model management. *IEEE Data Eng. Bull.*, pages 5–15, 2018.
[46] L. Shao et al. Griffon: Reasoning about job anomalies with unlabeled data in cloud-based platforms. SoCC '19, 2019.
[47] K. Shu et al. dEFEND: Explainable Fake News Detection. In *In ACM SIGKDD*, pages 395–405. ACM, 2019.
[48] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.
[49] M. Vartak et al. Modeldb: a system for machine learning model management. In *HILDA*, 2016.
[50] M. Vartak et al. Mistique: A system to store and query model intermediates for model diagnosis. In *SIGMOD*, 2018.
[51] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.

REPRODUCIBILITY

In this section we report additional information regarding our experimental evaluation of Section 7. We are taking the following steps to ensure reproducibility of our experiments:

Datasets: As mentioned in Section 7, we used a large corpus of **publicly** available Python scripts (26K). This dataset is available at [16].

Experiments with Labeled Datasets: In Section 7.2, we presented experiments with scripts for which we have manually extracted the provenance information (ground truth). The ground truth for the scripts that were used in these experiments is also available at [16].

Algorithms: The pseudocode for all the algorithms used by Vamsa is either presented in this paper, or is available in our technical report [35].

Knowledge Base: A file with the data stored in the knowledge base that we used for our experiments is available at [16].

Real Use Case: In section 7.5, we presented a real scenario encountered in production that highlights the significance of using tools such as Vamsa to collect provenance information from ML scripts. Figure 8 presents a simplified but representative version of the script that is used in production.

```
import warnings

warnings.filterwarnings("ignore", category=UserWarning)

import pandas as pd
import lightgbm as lgb
from sklearn import metrics
import numpy as np

data_train = pd.read_csv("global_train.csv")
data_test = pd.read_csv("global_test.csv")

# Feature Engineering
data_train["SuccessfulVertices"] = (data_train["TotalNumberOfVertices"] - data_train[
    "RevocationCount"] - data_train["FailedCount"]) / data_train["TotalNumberOfVertices"]

data_test["SuccessfulVertices"] = (data_test["TotalNumberOfVertices"] - data_test[
    "RevocationCount"] - data_test["FailedCount"]) / data_test["TotalNumberOfVertices"]

train_x = data_train.drop(columns=[
    "reason",
    "TotalNumberOfVertices",
])

train_y = data_train["reason"]

test_x = data_test.drop(columns=[
    "reason",
    "TotalNumberOfVertices",
])

test_y = data_test["reason"]

# Train/Test the model
n_leaves = 8
n_trees = 100
clf = lgb.LGBMClassifier(num_leaves=n_leaves, n_estimators=n_trees)
clf.fit(np.array(train_x), train_y)
score = metrics.precision_score(test_y, clf.predict(test_x), average='macro')
print("Precision Score on Test Data: " + str(score))
```

Figure 8: Simplified data science script used in our real example