

# LAMP: Data Provenance for Graph Based Machine Learning Algorithms through Derivative Computation

Shiqing Ma<sup>1</sup>, Yousra Aafer<sup>1</sup>, Zhaogui Xu<sup>2</sup>, Wen-Chuan Lee<sup>1</sup>, Juan Zhai<sup>2</sup>,

Yingqi Liu<sup>1</sup>, Xiangyu Zhang<sup>1</sup>

<sup>1</sup>Purdue University (USA), <sup>2</sup>Nanjing University (China)

## ABSTRACT

Data provenance tracking determines the set of inputs related to a given output. It enables quality control and problem diagnosis in data engineering. Most existing techniques work by tracking program dependencies. They cannot quantitatively assess the importance of related inputs, which is critical to machine learning algorithms, in which an output tends to depend on a huge set of inputs while only some of them are of importance. In this paper, we propose LAMP, a provenance computation system for machine learning algorithms. Inspired by automatic differentiation (AD), LAMP quantifies the importance of an input for an output by computing the partial derivative. LAMP separates the original data processing and the more expensive derivative computation to different processes to achieve cost-effectiveness. In addition, it allows quantifying importance for inputs related to discrete behavior, such as control flow selection. The evaluation on a set of real world programs and data sets illustrates that LAMP produces more precise and succinct provenance than program dependence based techniques, with much less overhead. Our case studies demonstrate the potential of LAMP in problem diagnosis in data engineering.

## CCS CONCEPTS

- Computing methodologies → Learning in probabilistic graphical models;
- Software and its engineering → Software testing and debugging;

## KEYWORDS

Data Provenance, Machine Learning, Debugging

### ACM Reference format:

Shiqing Ma, Yousra Aafer, Zhaogui Xu, Wen-Chuan Lee, Juan Zhai, Yingqi Liu, and Xiangyu Zhang. 2017. LAMP: Data Provenance for Graph Based Machine Learning Algorithms through Derivative Computation. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17)*, 12 pages.

<https://doi.org/10.1145/3106237.3106291>

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany*

© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.

ACM ISBN 978-1-4503-5105-8/17/09...\$15.00

<https://doi.org/10.1145/3106237.3106291>

## 1 INTRODUCTION

We are entering an era of data engineering. Compared to traditional software engineering, the complexity of data engineering largely lies in data and models. For example, many data processing programs, such as well-known machine learning programs, have a small size. But the data processed by these programs and the models generated are often large and complex. It poses new challenges to engineers such as how to validate outputs and how to diagnose problems. Note that faults may likely reside in data and models, while they are more likely present in programs in traditional engineering scenarios. Graph based machine learning (GML) is an important kind of data processing with increasing popularity. Provided with an input graph model and initial weight values, GML algorithms generate an updated model. Most of these algorithms are iterative. In each iteration, a vertex communicates with its neighbors and updates its value until all the values converge. Through multiple iterations, a vertex can affect other vertices that are many edges away. This is called the *rippling effect*. Due to the nature of such computation, it is highly challenging to determine the correctness of the generated models as a fault may get propagated through many steps and faulty states may get accumulated/obfuscated during propagation. Even if the user suspects the incorrectness of final outputs, she can hardly diagnose the procedure to identify the root cause, which could be present in the input graph model, the initial weight values, or even in the GML algorithm itself.

Data provenance is an important approach to addressing the problem. It identifies the input-output dependencies and/or records the operation history. There are a number of existing efforts [42–44, 53, 69] that aim to provide general frameworks for collecting data provenance for GML algorithms. Most of them focus on selectively collecting intermediate results at run time in an effort to provide crash recovery mechanisms, debugging support, and so on. However, these techniques can hardly reason about input-output dependencies. Dynamic information flow tracking, or tainting [28, 61], computes the set of inputs related to a given output, by monitoring program dependencies. However, it cannot quantify the importance of individual inputs. Due to the rippling effect of GML algorithms, an output tends to be dependent on a huge set of inputs even though most of them have negligible impact on the output.

In this paper, we propose LAMP, a technique to quantitatively reason about input importance. Inspired by *automatic differentiation* (AD) techniques [23, 24, 30, 36–39, 49, 60, 72], LAMP works by computing output derivatives regarding inputs. A large derivative indicates the input is of high importance for the output. A zero derivative indicates the input has no impact on the output. However, existing AD techniques cannot be directly used in GML provenance tracking due to the following reasons. (1) Derivative computation is

closely coupled with the original computation, leading to high overhead for production runs; (2) Derivatives cannot be used to quantify the importance of inputs related to discrete behavior, such as the inputs whose changes may lead to control flow changes. They are common in GML algorithms; (3) AD techniques typically compute derivatives for a smaller number of inputs. However in GML, all inputs need to be considered. LAMP addresses these challenges. In particular, it considers the initial weight values (of vertices/edges) as the input, even though they may be initialized to constants and do not come from the program input. The iterative procedure then aggregates, propagates, and updates these values, driven by the graph structure. As such, variations to the initial weight value of a vertex have impact to the final outputs proportional to the structural importance of the vertex. Therefore, derivatives regarding these initial (constant) values reflect the importance of vertices. LAMP features a novel design that separates the original data processing from provenance computation so that very little overhead is introduced during production runs. It quantifies input importance for those that may induce control flow variations by spawning processes to concretely determine the output variations. The workflow of LAMP is as follows. It first collects a lightweight trace during production run that contains branch outcomes and a small set of states. The trace allows the decoupling of the original computation and the provenance computation. It transforms the original program to a new program that solely focuses on provenance computation. The transformed program takes the original inputs and the trace, and produces the provenance (i.e., the importance measurement for each input).

We make the following contributions.

- We formally define the problem of provenance computation for GML algorithms, which has the key challenge of quantifying input importance. We propose a novel solution based on computing the partial derivatives of each output variable with respect to the related input variables.
- We propose a novel design that decouples original data processing from provenance computation.
- We propose an execution based approach to quantify input importance for those related to control flow.
- We develop a prototype LAMP. Our evaluation on a set of real world GML algorithms and large data sets show that LAMP substantially outperforms program dependence tracking based approaches in terms of accuracy and efficiency, producing provenance sets that are orders of magnitude smaller with overhead that is 3-6 times lower. Our case studies demonstrate the potential of LAMP in data engineering, for helping the development process and finding bugs in input data, graph models, and even in GML algorithm implementations.

## 2 MOTIVATION

PageRank [59] is one of the most popular GML algorithms. It computes a numerical weight for each vertex in a graph to measure its relative importance within the graph, called the *rank value*. It is widely used in practice such as in search engines and social network analysis. Based on the assumption that a vertex becomes popular when it has many links with other popular vertices, the algorithm

iteratively computes the rank value for a vertex by aggregating the rank values from its predecessor vertices. During each iteration, the rank value of a vertex is computed by the following equation:

$$PR(u) = \frac{1-d}{|V|} + d * \sum_{j \in B_u} \frac{PR(j) * k_j}{c_j} \quad (1)$$

, where  $B_u$  contains all the vertices which have at least one link/edge to  $u$ ,  $c_j$  is the sum of outgoing edge weights of vertex  $j$ , and  $k_j$  is the weight of the edge from  $j$  to  $u$ .  $|V|$  is the number of vertices in the graph, and  $d$  is a user defined *damping* coefficient to ensure fairness in computation for leaf vertices.

Figure 1 presents a sample PageRank implementation (in Python). It first assigns the initial rank values to all vertices in lines 3-4, and then updates the ranks inside a loop. In each iteration, the algorithm traverses all vertices, collects ranks from the parent nodes and updates the ranks accordingly (lines 16-24). After updating all the vertices, it calculates the difference between the current and the previous ranks to check if it has reached a relatively stable state, by comparing  $\delta$  with an  $\epsilon$  threshold (line 30). If the difference is smaller enough, the algorithm returns the ranks (line 31). Otherwise, it continues computing until the condition is satisfied or a maximum number of iterations is reached (line 7).

**Motivation Case.** We apply the algorithm to rank the accounts in the Weibo social network [9]. In the data set, a node represents an account, and an edge denotes that an account follows another account. Input edge weights are used to represent the number of actions (comment, re-tweet etc.) between followers and followees. In particular, an edge weight is calculated as the number of actions multiplied by a constant  $\tau$ , and subtracted by the number of tweets for which the follower does not do anything. We cross-check the ranking results with those by other methods [27, 46]. We observe that most results are consistent and popular accounts have high ranks. In Figure 2a, we show the provenance graph for a popular account Kai-Fu Lee, who is an IT celebrity. The node size represents the rank value of the node, and the edge width denotes the impact of a node on another node. As we can see, many accounts, including popular ones like NewsPlus and unpopular ones, are following Kai-Fu Lee and actively commenting and re-tweeting his tweets. Effectively, the red node in the center (representing Kai-Fu Lee) has many edges of different weights from many other nodes of various sizes, which represents a typical provenance graph for popular accounts.

However, we noticed that an unknown account, Yangqi, is mysteriously ranked very high (i.e., within top 50), while it has less than 200 followers, far less than popular accounts. Given the complexity of the dataset, it is difficult to diagnose why Yangqi ranks so high, especially that his followers may have their own followers that (transitively) contribute to his rank. Thus, we employ LAMP to investigate this case. Figure 2b presents the provenance graph for Yangqi. Observe that it is quite different from Kai-Fu Lee's graph. It suggests that the rank of Yangqi is highly influenced by another unpopular account Qing Fei. Note that the node for Qing Fei has a very small size but its edge is very heavy, implying that although Qing Fei's rank value is small, it substantially inflates the rank of Yangqi. Further inspection shows that Qing Fei has many negative weights on its outgoing edges, which makes the

```

1: def PageRank(graph, damping, epsilon, debug = True):
2:     # init rank values
3:     for key in graph.nodes():
4:         ranks[key] = __attributes___.__('input').init[key]
5:
6:     iter = 0
7:     for _ in range(MAX_ITERATION):
8:         # first, store old rank values
9:         old_ranks = copy.deepcopy(ranks)
10:
11:        iter = iter + 1
12:        if debug:
13:            print("Iter:", iter, " and ranks:", ranks)
14:
15:        # update ranks for all nodes in the graph
16:        for key in graph.nodes():
17:            rank_sum = 0
18:            for n in graph.out_edges(key):
19:                outlinks += float(weights(graph.out_edges(n)))
20:
21:                if outlinks > 0:
22:                    rank_sum += weight(n)*old_ranks[n]/outlinks
23:
24:                # page rank computation
25:                ranks[key] = (1-damping)/NUM_V + damping*rank_sum
26:
27:                # if converge, return the ranks
28:                delta = 0
29:                for key in graph.nodes():
30:                    delta = delta + abs(old_ranks[key] - ranks[key])
31:
32:                if delta < epsilon:
33:                    return ranks
34:
35:    # hit the MAX_ITERATION, return
36:    return __attributes___.__('output').ranks

```

(a) PageRank Code

```

def PageRankProv(graph, damping, epsilon, debug = True):
    for key in graph.nodes():
        A prov(ranks, key)[key] = 1

    for _ in range(MAX_ITERATION):
        for key in graph.nodes():
            for n in graph.out_edges(key):
                B if load(line20, cnt(line20, plus=True), 0):
                C for k in graph.nodes():
                    prov(rank_sum)[k] = prov(rank_sum)[k] + \
                    prov(old_ranks,n)/load(line21, cnt(line21, plus=True),0)
                D for k in graph.nodes():
                    prov(ranks, key)[k] = damping*prov(rank_sum)[k]

    E if (load(line30, cnt(line30), 0)):
        if (size(line30, cnt(line30)) > 1) and \
        unstable_test(delta-epsilon):
            p = spawn(ranks=load(line30, cnt(line30), 2), line7)
            unstable_logging(p)
            return prov(ranks)
        else:
            # similar to true branch
            cnt_increase(line30)
    return prov(ranks)

```

(b) Transformed PageRank Code

Figure 1: PageRank and the corresponding transformed code

sum of its outgoing edge weights, i.e.,  $c_j$  in Equation (1), far smaller than the edge weight from Qing Fei to Yangqi, i.e.,  $k_j$  in Equation (1). As a result,  $k_j/c_j$  is much larger than 1 such that the rank of Yangqi,  $PR(u)$ , is essentially the rank of Qing Fei,  $PR(j)$ , multiplied by a very large factor according to Equation (1). The root cause is that when the dataset was generated (by other researchers), the parameter  $\tau$  was not well-defined, which has led to negative weights. To fix this problem, we redefine the value of  $\tau$  to preclude negative values. The updated rank of Yangqi correctly represents its unpopularity. Its provenance graph is shown in Figure 2c. This case illustrates a typical fault in data engineering. More cases can be found in §7.

Table 1: Provenance by Tainting

Graph	Node	Edges	Avg	Max	Run	Mem
StanfordWeb	1,000	10,948	723	1,000	825%	33.25%
GoogleWeb	34,546	421,578	6,194	21,349	1103%	42.19%
TencentUser	10,970	170,327	1,293	8,306	1023%	48.24%
TencentMsg	83,306	2,516,122	12,342	82,194	1345%	39.25%
Twitter	96,401	482,834	32,593	72,294	1483%	29.38%

**Dependency Analysis based Provenance Tracking.** A traditional way of collecting provenance for programs is by tracking data and control dependencies [28]. However in GML, an output tends to be (transitively) dependent on a large set of inputs through program dependencies due to the rippling effect. Such huge provenance sets for outputs are hardly useful as the importance of individual input values cannot be distinguished. A lot of inputs in the provenance set have very little impact on the output. Furthermore, taint propagation for individual operations is usually implemented as set operations (e.g., set unions), which are very expensive on provenance sets. We run PageRank for 10 iterations on a few data sets while collecting provenance using dynamic tainting. The results are shown in Table 1, which presents the input data set

(column 1), the number of vertices/edges (column 2/3) in the graph, the average/maximum size of provenance set (column 4/5), and the runtime/memory overhead (column 6/7). Observe that even though we only run it for a small number of iterations, the provenance sets are already of large size, which cause substantial overhead.

In comparison, we show the distribution of derivatives computed by LAMP when running the PageRank algorithm on the Tencent Message data set in Figure 3. The x-axis represents the logarithm of derivative value, and the y-axis represents the number of occurrences. As we can see, most derivatives are small, meaning that the rank of a vertex is mostly determined by a small number of vertices.

### 3 PROBLEM STATEMENT

Our goal is to compute data provenance for GML programs.

**DEFINITION 1.** A Graph based Machine Learning (GML) program  $P$  takes a graph model  $G$  and its nodes' initial weight values  $I = \{x_0, \dots, x_m\}$ , where  $x_i$  is the initial weight of node  $i$ , and produces updated weight values  $F$  without changing the graph structure.

The initial weights of a given graph's nodes may be explicitly provided as part of the input (e.g., as in Bayesian Networks [51], Belief Propagation [68]) or using a pre-defined constant (e.g., as in the original PageRank [59]). Edge weights are handled no differently from node weights as both are represented as program variables. Hence for simplicity, we only assume node weights in our discussion. LAMP can be applied to many ML algorithms that fall in our GML definition.

**PROBLEM STATEMENT 1.** Consider the output weight for a node  $n$  as a function over the initial weights, denoted as  $F_n(x_0, \dots, x_m)$ . For each node  $i$  with an initial weight  $x_i$ , and a given execution  $x_0 = a_0, \dots, x_m = a_m$ , we aim to compute the partial derivative  $\frac{d F_n(a_0, \dots, a_m)}{d x_i}$ , if the partial function  $F_n(a_0, \dots, a_{i-1}, x_i, a_{i+1}, \dots, a_m)$

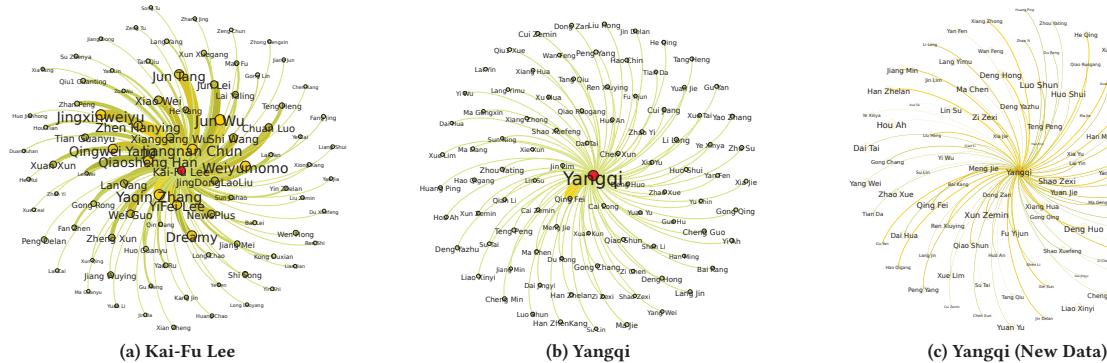


Figure 2: Weighted PageRank Data Debugging Example

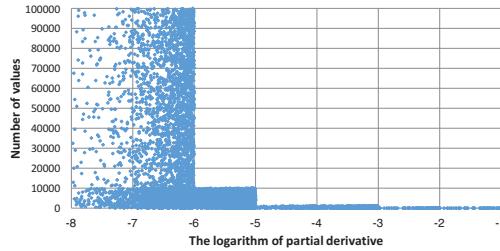


Figure 3: The distribution of impact values (Tencent Message graph)

is continuous at  $x_i = a_i$ . Otherwise, we compute  $|F_n(a_0, \dots, a_i + \epsilon, \dots, a_m) - F_n(a_0, \dots, a_i - \epsilon, \dots, a_m)|$  with  $\epsilon$  being an infinitely small value.

Our goal is to compute the partial derivative that represents the impact of each input on each output. Intuitively, if a vertex is important (i.e., by having a high initial weight or by being connected to many other vertices), a small perturbation of its initial value will change the values of all the connected vertices and eventually lead to substantial output changes. However, while mathematical functions are largely continuous, GML programs have a lot of discrete behaviors. As a result, the  $F_n$  functions are usually discontinuous. Note that in a discontinuous function, an arbitrarily small input variation does not lead to arbitrarily small output variation. As a result, the derivative is infinite. In this case, derivatives do not represent the impact of input variations. Therefore, we report the output variations instead.

Figure 4 depicts an illustrative example. The program, presented in the small box, behaves as follows, based on the input  $x$ :

$$F(x) = \begin{cases} f(x) & x \leq c \\ g(x) & \text{otherwise} \end{cases}$$

$F(x)$  is discontinuous at  $x = c$ . For any  $x = t$  and  $t < c$ , LAMP computes  $\frac{df(t)}{dx}$ , which is the slope of the tangent line to  $f(x)$  at  $x = t$ , denoting how the output varies at the neighborhood of  $x = t$ . Upon  $x = c$ , the derivative is not informative due to the discontinuity. Hence, LAMP computes  $|f(c) - g(c)|$  instead, which gauges the impact of input variation.

In this paper, we do not consider floating point rounding errors. Since we reason about input variations at a much larger scale compared to rounding errors, the effects of rounding errors are largely

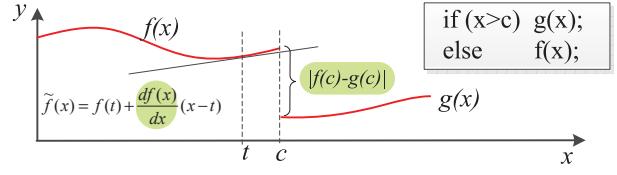


Figure 4: Essence of LAMP

shadowed. We will leave a thorough study of the interference of rounding errors to the future.

## 4 OVERVIEW

**Basic Idea.** LAMP computes the partial derivatives for each variable on the fly. Given a statement  $y = f(t_1, \dots, t_m)$  with  $t_1, \dots, t_m$  the operands, LAMP computes the partial derivatives of  $y$  regarding each initial weight, leveraging the derivative chain rule. It is a rule to compute the derivative of function composition:

$$\frac{d}{dx} [f(u)] = \frac{d}{du} [f(u)] \frac{du}{dx} \quad (2)$$

, where  $f(u)$  is the final output,  $u$  is an intermediate result and  $x$  is the input variable. Intuitively, it says that the derivative of a function regarding its input can be computed from the derivative of the function regarding an intermediate result and the derivative of the intermediate result regarding the input. Leveraging the chain rule, derivative computation can be done locally to a statement, based on the operand values and their partial derivatives that were computed when the operands were defined. Upon a predicate, LAMP checks if a small variation to any initial weight value can cause the predicate to take a different branch outcome. This can be done by linear approximation using the computed partial derivative of the predicate expression. If so, LAMP spawns a new process to take the other branch. At the end, output variations are derived by comparing the outputs from all the processes. According to the problem statement in §3, the partial derivatives and the output variations caused by discontinuity are the resulting data provenance that gauges the impact of inputs on outputs.

Consider the code example on the left side of Figure 5. The program takes two initial weight values:  $x_1$  and  $x_2$ . The predicate at line 4 makes the output weights discontinuous functions. The next two columns in the same figure show the conceptual provenance computation regarding  $x_1$  and  $x_2$ , respectively. At line 1, the partial

Code	Derivatives w.r.t. $x_1$	Derivatives w.r.t. $x_2$
1 $p=2x_1+3x_2;$	① $p'_{x_1}=2;$	1 $p'_{x_2}=3;$
2 $q=x_2*x_2;$	② $q'_{x_1}=0;$	2 $q'_{x_2}=2*x_2;$
3 $r=p*q$	③ $r'_{x_1}=p'_{x_1}*q+p*q'_{x_1};$ //2*x <sub>2</sub> *x <sub>2</sub>	3 $r'_{x_2}=p'_{x_2}*q+p*q'_{x_2};$ //4x <sub>1</sub> x <sub>2</sub> +9x <sub>2</sub> x <sub>2</sub>
4 $if(r>100)$	④ <sub>T</sub> $if(r>100 \&\& r- r'_{x_1}*\Delta <=100)$	4 <sub>T</sub> $if(r>100 \&\& r- r'_{x_2}*\Delta <=100)$
5 $s1;$	/*spawn a process to execute s2*/	/*spawn a process to execute s2*/
6 $else s2;$	④ <sub>F</sub> $if(r<=100 \&\& r+ r'_{x_1}*\Delta >100)$	4 <sub>F</sub> $if(r<=100 \&\& r+ r'_{x_2}*\Delta >100)$
	/*spawn a process to execute s1*/	/*spawn a process to execute s1*/

Figure 5: Example to Illustrate the Basic Idea. Symbol  $p'_{x_1}$  denotes  $\frac{d p}{d x_1}$ .

Program	$P ::= s^*$
Statement	$s ::= x =^\ell e$   $x =^\ell y \text{ op } z$   $\text{while } (x >^\ell 0) : s$   $\text{if } x >^\ell 0 \text{ then } s_1 \text{ else } s_2$
Expr	$e ::= v \mid x \mid x^{input} \mid y^{output}$
Operator	$op ::= + \mid - \mid * \mid / \mid \dots$
Value	$v ::= \{\text{True}', \text{False}', 0, 1, \dots\}$
Label	$\ell ::= \{\ell_1, \ell_2, \dots\}$
Variable	$x, y, z \in \text{Identifier}$

Figure 6: Language Abstraction

derivatives of  $p$  with regards to  $x_1$  and  $x_2$  are 2 and 3, respectively. At line 2, the partial derivatives of  $q$  with regards to  $x_1$  and  $x_2$  are 0 and  $2 * x_2$ , respectively. At line 3, according to the chain rule, the derivatives of  $r$  are computed from the values and the previously computed derivatives of  $p$  and  $q$ . Note that the symbolic value of  $r$  regarding  $x_1$  and  $x_2$  is  $r(x_1, x_2) = 2x_1x_2^2 + 3x_2^3$ ,  $\frac{d r(x_1, x_2)}{d x_2} = 4x_1x_2 + 9x_2^2$ , which is exactly the value computed through the three steps ① – ③. Upon the predicate, ④<sub>T</sub> indicates that if the original execution takes the true branch (i.e.,  $r > 100$ ) but the linear approximation of  $r$ 's variation with a small variation  $\Delta$  of  $x_1$  leads to the opposite branch outcome, a new process is spawned to take the false branch so that the outputs along this different path can be computed and contrasted with the original output.

**Workflow.** The aforementioned procedure is just a conceptual explanation. In practice, it is too expensive to perform provenance computation during production runs. An important design choice of LAMP is hence to decouple the original computation from the provenance computation. Observe that in the provenance computation, the values computed in the original run are usually not needed. For instance, at steps ① and ④<sub>T</sub>, the values of  $p$ ,  $x_1$ , or  $x_2$  are not needed at all. In fact, the operand values are only needed in multiplication and division operations, which are relatively rare compared to additions and subtractions. Therefore, LAMP records the needed variable values during production runs, such as values  $p$  and  $q$  at line 3.

LAMP then transforms the original program to a new one dedicated to provenance computation, which is triggered on demand. During provenance computation, the logged values are used to avoid most of the original computation.

## 5 DESIGN

To facilitate discussion, we introduce a simple language in Figure 6. We use superscripts to indicate input and output variables. Note that here input variables are the initial weight values, which may be

loaded from input files, or initialized to some constant (e.g., line 4 in Figure 1). Each statement is identified by a label  $\ell$ . The language is just for discussion. Our implementation supports Python.

### 5.1 Run Time Information Collection

During production runs, LAMP conducts very lightweight tracing to collect branch outcomes and the results of some operations (e.g., multiplications). The tracing semantics are explained in Figure 7. The expression rules are standard. According to the evaluation context  $E$ , expressions are first evaluated to values before the statement rules are applied. Statement evaluation has the configuration of  $\sigma$ ,  $\omega$ , and  $C$ :  $\sigma$  is the store;  $\omega$  is the tracing log that consists of a sequence of trace entries, each containing a statement label, the execution counter value of the statement, and a set of values;  $C$  records the current counter value for each statement.

The evaluation rules of most statements are standard and hence elided. Rules [MUL-LOG] and [MUL-LOG-Y] indicate that LAMP may log the operand values for multiplications because such values are needed in derivative computation in the later provenance computation phase (§4). If the compiler statically determines that both operand variables  $y$  and  $z$  are related to annotated input variables, their values are logged by attaching an entry to  $\omega$ . The counter is also increased. Similarly, if only one operand is input related, the other operand value is logged. When neither operand is input variable related, LAMP does not need to compute the derivatives and hence the operand values are not recorded (Rule [MUL-NoLOG]).

Upon a conditional statement, LAMP determines if the predicate is related to input variables. If so, it further detects if the branch outcome may be unstable by function *unstable()*. We say that a branch outcome is unstable if a small input perturbation  $\Delta$  flips the branch outcome. Specifically, for a predicate  $v>0$  and a related input  $x_i$ , the predicate is unstable if  $|\frac{d v}{d x_i}| \times \Delta > |v|$ . However during production run, we do not know the derivative. Hence, we test if  $v/\Delta$  is smaller than the pre-defined maximum partial derivative. If so, the predicate may be unstable and we log the branch outcome, the value  $v$ , and the values of critical state variables (Rule [IF-UNTABLE-T]). We determine if the predicate is truly unstable during provenance computation. If so, a process is spawned to take the other branch. Logging the critical state variables is to support the child process. For example in PageRank (Figure 1), the comparison at line 30 is input related and possibly unstable when  $|\delta - \epsilon|/\Delta$  is smaller than  $MAXD$ . Thus,  $ranks[]$  values are recorded, which are sufficient for execution along the other branch. The set of critical variables at a program point  $\ell$ ,  $CS(\ell)$ , is pre-computed by the compiler. In our experience,  $CS$  sets are small

Definitions:		
$\omega \in \text{Log}: \langle \text{Label}, \text{Index}, \text{Value}^* \rangle$	$\sigma \in \text{Store}: \text{Variable} \rightarrow \text{Value}$	$C \in \text{StmtIndex}: \text{Label} \rightarrow \text{Index}$
$\text{input\_rel}(x)$ : if $x$ is (transitively) data dependent on any input		
$CS(\ell)$ : the set of critical state variables at $\ell$		
$\text{unstable}(v) =  v/\Delta  < \text{MAXD}$ , with $\Delta$ the input variation bound and $\text{MAXD}$ the upper bound of partial derivatives. It determines if the predicate may potentially take the opposite branch in the presence of input variation.		
Semantic rules:		
$E ::= E; S \mid [\cdot]_s \mid x = [\cdot]_e \mid x = [\cdot]_e \text{ op } e \mid x = v \text{ op } [\cdot]_e \mid \text{if } [\cdot]_e > y \text{ then } s_1 \text{ else } s_2 \mid \text{if } v > [\cdot]_e \text{ then } s_1 \text{ else } s_2$		
Expression Rule:	$\sigma : e \xrightarrow{e} v$	
$\sigma : v \xrightarrow{e} v$	[E-CONST]	$\sigma : x \xrightarrow{e} \sigma(x)$ [E-VAR]
Statement Rule:	$\sigma, \omega, C : s \xrightarrow{s} \sigma', \omega', C', s'$	
$\sigma, \omega, C : x =^\ell v_y * v_z \xrightarrow{s} \sigma[x \rightarrow v_y * v_z], \omega \cdot \langle \ell, C(\ell), v_y, v_z \rangle, C[\ell \rightarrow C(\ell) + 1], \text{skip}$		, if $\text{input\_rel}(y) \wedge \text{input\_rel}(z)$ [MUL-LOG]
$\sigma, \omega, C : x =^\ell v_y * v_z \xrightarrow{s} \sigma[x \rightarrow v_y * v_z], \omega \cdot \langle \ell, C(\ell), v_y \rangle, C[\ell \rightarrow C(\ell) + 1], \text{skip}$		, if $\neg \text{input\_rel}(y) \wedge \text{input\_rel}(z)$ [MUL-LOG-Y]
$\sigma, \omega, C : x =^\ell v_y * v_z \xrightarrow{s} \sigma[x \rightarrow v_y * v_z], \omega, C, \text{skip}$		, otherwise [MUL-NoLOG]
$\sigma, \omega, C : \text{if } v >^\ell 0 \text{ then } s_1 \text{ else } s_2 \xrightarrow{s} \sigma, \omega \cdot \langle \ell, C(\ell), \text{True}, v, \sigma(CS(\ell)) \rangle, C[\ell \rightarrow C(\ell) + 1], s_1$		, if $v > 0 \wedge \text{input\_rel}(x) \wedge \text{unstable}(v)$ [IF-UNSTABLE-T]
$\sigma, \omega, C : \text{if } v >^\ell 0 \text{ then } s_1 \text{ else } s_2 \xrightarrow{s} \sigma, \omega \cdot \langle \ell, C(\ell), \text{True} \rangle, C[\ell \rightarrow C(\ell) + 1], s_1$		, if $v > 0 \wedge (\neg \text{input\_rel}(x) \vee \neg \text{unstable}(v))$ [IF-STABLE-T]
Global Rules:		
$\frac{\sigma : e \xrightarrow{e} v}{\sigma, \omega, C, E[e]_e \rightarrow \sigma, \omega, C, v}$	[G-EXPR]	$\frac{\sigma, \omega, C : s \xrightarrow{s} \sigma', \omega', C', s'}{\sigma, \omega, C, E[s]_s \rightarrow \sigma', \omega', C', E[s']_s}$ [G-STM]

Figure 7: Semantic rules

in GML programs and the number of unstable predicates at run time is very small, thus the space overhead is low (§6). If a predicate is not input related or is stable, LAMP simply logs the branch outcome (Rule [IF-STABLE-T]). The branch outcomes will be reused during provenance computation to ensure the same control flow.

## 5.2 Code Transformation

LAMP transforms the original program to a new program, which takes the original input graph and the log generated in the tracing phase, and performs provenance computation. Figure 8 describes the set of transformation rules. A number of terms and helper functions are defined in the top of the figure. In particular, two global variables are declared in the transformed program:  $\Gamma$  that maps a variable to its partial derivatives (regarding input variables), and  $cnt$  that maps a statement to its current execution count.

Rule [T-INPUT-ASGN] specifies that for a statement (in the original program) that copies an input variable  $y^{input}$  to  $x$ , statements are added to the transformed program to set the derivative of  $x$  regarding  $y^{input}$  to 1 and the derivatives for other input variables to 0. In all the rules, the transformed statements are boxed. Note that the original assignment is precluded from the transformed program. For an addition statement, the transformed statement adds the corresponding derivatives (Rule [T-ADD]). Rule [T-MUL-Y] specifies that given a multiplication statement  $x = y * z$ , if only  $y$  is input related, the transformed statement computes the partial derivative of  $x$  by multiplying the derivative of  $y$  and the recorded value of  $z$  from the log  $\omega$ . When both  $y$  and  $z$  are input related, the multiplication is transformed to statements that compute the derivative of  $x$  from both the derivatives and the values of  $y$  and  $z$  (Rule [T-MUL-YZ]).

When the variable in predicate is not input related, the statement is transformed to loading the branch outcome from the log (Rule [T-IF-NOINPUT]). According to Rule [T-IF-INPUT], when the variable is input related, the following statements are added

to the transformed program. Line 1 tests if the recorded branch outcome is true. If so, line 2 further tests if the log entry contains additional information (i.e.,  $|\omega[\ell, \text{cnt}(\ell)]| > 1$ ), which indicates the predicate is potentially unstable, and if a small input variation  $\Delta$  induces a value change on  $x$  larger than the (recorded) value, leveraging the partial derivative. If so, the branch outcome can be flipped. Hence, LAMP spawns a process to continue execution along the other branch *in the original program* (lines 3). Before executing the branch, LAMP restores the critical state. The parent process continues the derivative computation in the true branch (line 5). Line 4 is to log the annotated input variables whose variations may flip the branch outcome, and the child process id. At the end of computation, for each input that causes unstable predicates, LAMP collects the values of an output variable  $z$  across all the associated processes with  $z_{max}$  and  $z_{min}$  the maximum and the minimum  $z$ , which denote the impact of the input on  $z$ . Figure 1 shows the transformed PageRank.

## 5.3 Discussion

PROPERTY 1. *If an output is a continuous function of an (annotated) input variable within a range, the partial derivative computation of LAMP is precise in the range.*

For example in Figure 4, the derivative computation in the ranges  $x < c$  and  $x > c$  is precise. This is because LAMP strictly follows the mathematical rules of derivative computation. However, since LAMP does not model derivative variation with regard to input variation. As such, if a derivative varies substantially (e.g., when an output function oscillates rapidly), the derivatives computed by LAMP may not be a good indicator for input impact. In practice, most GML algorithms are iterative algorithms that have very slow derivative variation, as supported by our experiments in §6.

PROPERTY 2. *Assuming the variation of an output function  $f(x)$  within an input variation bound of  $(0, \Delta)$  is bounded by  $\epsilon$ . In the*

<b>Definitions:</b>	
$\Gamma[x]$	: the array storing the derivatives of $x$ regarding inputs
$cnt[\ell]$	: the execution counter for statement $\ell$
$\omega[\ell, n, 0]$	: the first recorded value for the log entry of the $n$ th instance of statement $\ell$
$spawn(s, \ell)$	: spawn a process that first executes $s$ and then original code starting from $\ell$ with the tracing semantics
$log(p, t)$	: record that process $p$ is spawned because of instability caused by input $t$
$IV$	: the set of annotated input variables

<b>Transformation Rule:</b> $s \xrightarrow{ctx} s'$
$x = y^{input} \xrightarrow{ctx} \boxed{for(t \in IV) \Gamma[x, t] = 0; \Gamma[x, y^{input}] = 1}$ [T-INPUT-ASGN]
$x = y + z \xrightarrow{ctx} \boxed{for(t \in IV) \Gamma[x, t] = \Gamma[y, t] + \Gamma[z, t]}$ [T-ADD]
$x =^{\ell} y * z \xrightarrow{ctx} \boxed{\begin{array}{c} \Gamma[x, t] = \Gamma[y, t] * \omega[\ell, cnt[\ell], 0]; \\ cnt[\ell]++; \end{array}}$ [T-MUL-Y]
$x =^{\ell} y * z \xrightarrow{ctx} \boxed{\begin{array}{c} \Gamma[x, t] = \Gamma[z, t] * \omega[\ell, cnt[\ell], 0] + \Gamma[y, t] * \omega[\ell, cnt[\ell], 1]; \\ cnt[\ell]++; \end{array}}$ [T-MUL-YZ]
$if(x >^{\ell} 0) \text{ then } s_1 \text{ else } s_2 \xrightarrow{ctx} \boxed{\begin{array}{c} \Gamma[x, t] = \Gamma[s_1, t] \text{ if } (\omega[\ell, cnt[\ell], 0] \neq 0) \text{ then } s_1' \text{ else } s_2' \\ \Gamma[x, t] = \Gamma[s_2, t] \text{ if } (\omega[\ell, cnt[\ell], 0] \neq 0) \text{ then } s_2' \text{ else } s_1' \end{array}}$ [T-IF-NOINPUT]
$if(x >^{\ell} 0) \text{ then } s_1^{\ell_1} \text{ else } s_2^{\ell_2} \xrightarrow{ctx} \boxed{\begin{array}{c} 1 \text{ if } (\omega[\ell, cnt[\ell], 0] \neq 0) \text{ then } \\ 2 \text{ if }  \omega[\ell, cnt[\ell], 0]  > 1 \& \& \exists d \in \Gamma[x]  d \times \Delta  >  \omega[\ell, cnt[\ell], 1]  \text{ then } \\ 3 \text{ } p = spawn(CS[\ell] = \omega[\ell, cnt[\ell], 2], \ell_2); \\ 4 \text{ } for(t \in IV \text{ s.t. }  \Gamma[x, t] \times \Delta  >  \omega[\ell, cnt[\ell], 1] ) \log(p, t); \\ 5 \text{ } s_1^{\ell_1}; \\ 6 \text{ else } .../*! \omega[\ell, cnt[\ell], 0] */ \\ 7 \text{ } cnt[\ell]++; \end{array}}$ [T-IF-INPUT]

Figure 8: Transformation Rules

presence of discontinuity (caused by control flow), the error of data provenance computed by LAMP is bounded by  $\epsilon$ .

Recall upon discontinuity, LAMP computes output variations, instead of derivatives (§3). Take Figure 4 as an example. At  $x = c$ , ideally LAMP should compute  $f(c) - g(c)$ . However, according to our semantics, when  $x \in (c, c + \Delta)$ , the branch outcome may be flipped, LAMP hence computes  $f(x) - g(x)$ . Note that although  $f(x)$  is undefined in  $(c, c + \Delta)$  (in the original program), LAMP essentially approximates it in this range by spawning a process to take the else branch. According to our assumption, the computed  $f(x) - g(x)$  has a bounded error when compared to  $f(c) - g(c)$ .

LAMP uses the derivatives as provenance and outputs a weighted dependency graph with partial derivatives as weights. It can detect bugs that affects such dependency relationships and corresponding weights. Its capability is also affected concrete numerical values. Bugs with invisible or little effect on the weights have less probability to be detected by LAMP. Additionally, as LAMP targets the machine learning computing process, it will not be able to help if the bugs occurring in the workflow (e.g., choosing the wrong data sets, inappropriate machine learning algorithms).

## 6 EVALUATION

In this section, we report the evaluation results regarding efficiency and effectiveness. Our implementation is based on [66], a Python analysis platform we developed which can perform static analysis and instrumentation. All experiments were conducted on a machine with 4 cores and 64 GB memory, running Ubuntu 14.04 LTS.

Table 2: Code Size (LOC) of Transformed Program

Algorithms	Original	Transformed
PageRank [59]	144	171
Weighted PageRank [65]	168	182
Undirected PageRank [40]	142	170
Visit of Links based PageRank [50]	224	242
Visit of Links based Weighted PageRank [63]	256	270
Personalized PageRank [25]	152	184
Collusionrank [33]	124	135
SimRank [45]	111	158
ASCOS [26]	126	172
TextRank [56]	348	382
Belief Propagation [68]	384	462
Gibbs Sampling [31]	265	296

Table 2 shows the 12 real world algorithms used in our evaluation. Weighted PageRank computes rank values in weighted graphs; Visit of Links based PageRank requires the visit information to rank web pages; Personalized PageRank requires a user input vector (i.e., user preference) to calculate the rank values for individual users. SimRank and ASCOS calculate the similarity of two nodes in a graph with different methods. Belief propagation inferences the marginal distribution for unobserved nodes, conditional on observed nodes. Gibbs sampling is a well-known Markov Chain Monte Carlo (MCMC) sampling method. The 2nd and 3rd columns show the sizes of the original program and the transformed program. Note that although these programs are not large, they are the typical GML programs used in practice. Popular graph analysis tools (e.g., NetworkX [12], graph-tool [8]) implement PageRank in 80– LOCs. Our adaption of GML programs (of such sizes) is also consistent with experiment setup in the literature [54, 55, 67, 70].

## 6.1 Efficiency

**Run Time:** Different programs may require data sets in different formats. For 6 programs, we used the same 14 real world data sets. For the remaining programs, we used 7 other data sets. The data sets are acquired from public sources (e.g. Stanford Large Network Dataset Collection [52] and MovieLens [41]) or crawled by Scrapy [16]. The sources and the data set sizes (number of vertices/number of edges, and text file size for Gibbs Sampling) are reported in Table 3. Some data sets (e.g., Tencent Messages) are huge so that we can only use part of them due to our memory capacity. Here, we only report the overhead of provenance computation. The tracing overhead is less than 1% for all the programs due to the limited instrumentation and the small logs generated. As shown in Table 3, provenance computation takes 2 to 3 times the original computation time. LAMP’s provenance computation is optimized. The optimizations reduce the overhead by an average factor of 3. Details about optimizations can be found in [10]. While we consider the overhead reasonable, it cannot be afforded during production runs. This strongly supports our design that separates the original computation from the provenance computation.

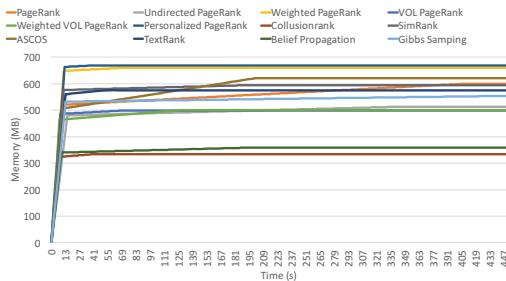


Figure 9: Memory usage of provenance computing

**Memory Consumption:** We measure the memory consumption of provenance computation using Python memory profiler [15] that samples every 0.1 second. Figure 9 presents the results. In the first few minutes, it grows very fast. This is attributed to two reasons. First, the process first needs to load the whole graph. Second, during the first a few iterations, the number of inputs related to a variable rapidly grows. After this, the sets of related inputs are relatively stable, and only their derivative values are being updated in memory. Thus the memory consumption has a slow growth. For most programs, our optimizations can reduce the memory consumption by a factor of 2.

**Log Space:** As LAMP needs to log some variables and branch outcomes, we also measure the log size. For the data sets used (with millions nodes and edges), the log size ranges from 3-7KB. A key reason is that the running encoding optimization is very effective in compressing branch outcomes (details in [10]).

**Comparison with Tainting:** To demonstrate the advantage of LAMP over tainting, we compare the efficiency of the two techniques. Our experiments show that the average runtime and space overheads for tainting are 1133.17% and 41.16%, whereas those for LAMP are 250.16% and 19.71%, respectively. This is because tainting has to manipulate large sets. Details are in [10].

**Unstable Predicates:** The stability of predicates varies according to the input variation threshold  $\Delta$ . We conduct experiments with

$\Delta = 10\%$ ,  $20\%$ , and  $40\%$ . We observe totally 2, 8 and 20 forks, respectively, for our test data sets. For a single execution, the maximum number of forks we observe is 2. Most cases have only one fork. Recall that LAMP forking a child process means that the input variation flips a branch outcome, leading to output variations that cannot be described by derivatives.

## 6.2 Effectiveness

We study the distribution of the derivatives for all the data sets. Our results show that only  $0.03 - 0.08\%$  of derivatives are larger than  $10^{-3}$ ,  $0.52 - 0.86\%$  larger than  $10^{-4}$ , and  $87 - 96\%$  are smaller than  $10^{-6}$ . Details are in [10]. This clearly indicates that most of the related inputs are insignificant. Unfortunately, traditional tainting based approaches would report all these insignificant inputs.

We also perform another experiment to validate the correctness of the computed derivatives. For each data set, we randomly select an input whose derivative is larger than  $10^{-3}$ . We then mutate the input value by 10%, run the program again with the mutated data, and measure the output differences. We then compute the *observed derivative* as the ratio between the observed output difference and the input variation, and compare it with the reported derivative by LAMP. We repeat it 500 times (i.e., randomly selecting 500 different inputs) for each data set and report the average. Our results show that the average difference of observed derivatives and reported derivatives range from  $6.32E - 7$  to  $1.02E - 05$  with the standard deviation from  $5.42E - 08$  to  $3.42E - 06$  (details in [10]). The differences are very small, indicating that the derivatives by LAMP can precisely measure input importance. This supports our assumption that derivatives change slowly with input changes (§5.3).

## 7 CASE STUDIES

### 7.1 Model Evolution

ML models are often generated in an incremental fashion due to the cost of training and the availability of new training data. We use a Bayesian classifier [3] to demonstrate how LAMP can facilitate data engineers in this process. The classifier implements Paul Graham’s algorithm [14] to classify spam comments. The original model was trained on 30,000 manually labeled YouTube comments. Given a new comment, it outputs a score (between 0 and 1) to predict if the comment is a spam. The score is computed from the scores of individual words in the comment. For example, the word `click`, which appears frequently in spam comments and rarely in benign ones, strongly hints a spam comment. We applied the original model [3] to a data set collected from other YouTube videos and observed many mis-classifications. We show two examples in the following.

- C1 (Benign): Billie Jean, this is one of the my favorite videos I have ever seen this year. It is performed by Michael Jackson. The dance is known as the Moonwalk.
- C2 (Spam): Geico is the best auto-insurance company. The price is low. You should apply now.

The computed score of C1 is 0.999721671845, indicating a spam. But it is a false positive. We generate its provenance graph Figure 10a using LAMP. The size of a node indicates the probability of

Table 3: Provenance Computation Overhead

Algorithm	Web Graph-1	Tencent Message	Cit-HepPh	Tencent User	ego-Twitter	p2p-GnuTella08	Wikipedia Link
	1K/1M	382K/13M [20]	34K/421K [52]	438K/6M [20]	81K /2M [52]	6K/21K [52]	4K/823K [22]
PageRank	372.24%	121.06%	274.81%	126.98%	298.34%	309.38%	194.20%
Per PageRank	322.73%	182.23%	276.39%	123.48%	248.39%	327.38%	231.28%
Collusionrank	328.48%	128.28%	263.48%	138.56%	192.39%	382.47%	183.38%
SimRank	273.38%	124.68%	251.37%	183.45%	183.48%	325.27%	182.84%
ASCOS	273.48%	198.28%	174.27%	184.53%	263.59%	385.73%	108.83%
Belief Prop	283.34%	201.28%	273.47%	128.38%	294.38%	294.58%	192.38%
Algorithm	Twitter	p2p-GnuTella09	wiki-Vote	MovieLen 10M	web-Google	soc-Slashdot0922	email-EuAll
	696K/2M [33]	8K/26K [52]	7K/104K [52]	72K/10M [41]	876K/5M [52]	82K/948K [52]	265K/420K [52]
PageRank	121.06%	284.38%	273.27%	185.37%	284.73%	284.92%	283.25%
Per PageRank	204.37%	274.37%	239.37%	183.58%	274.85%	238.26%	294.38%
Collusionrank	197.35%	263.46%	263.84%	183.38%	385.27%	304.28%	301.58%
SimRank	194.35%	263.23%	273.38%	174.28%	302.38%	329.38%	318.85%
ASCOS	237.75%	284.58%	284.37%	173.58%	326.48%	329.48%	333.62%
Belief Prop	206.58%	274.38%	284.58%	184.27%	385.27%	274.59%	321.38%
Algorithm	LDA Sample-1	LDA Sample-2	LDA Sample-3	LDA Sample-4	LDA Sample-5	LDA Sample-6	LDA Sample-7
	121M	143MB	147MB	100MB	137MB	125MB	134MB
GibbsSamp	123.29%	128.34%	124.56%	119.35%	120.35%	118.23%	116.23%
Algorithm	Web Graph-1	Web Graph-2	Web Graph-3	Web Graph-4	Web Graph-5	Web Graph-6	Web Graph-7
	1,000/998,996	5,000/723,283	2,283/374,382	1,927/428,283	6,293/823,283	8,238/823,273	7,238/824,837
VOLPRank	293.56%	192.42%	246.33%	364.21%	352.26%	314.27%	356.25%
WVOLPRank	325.42%	243.56%	236.43%	326.47%	295.63%	324.36%	316.32%
Algorithm	MovieLen 100k	MovieLen 20M	MovieLen 1M	MovieLen 10M	ego-Facebook	ego-Amazon	com-DBLP
	1K/100K [41]	138K/20M [41]	6K/1M [41]	72K/10M [41]	4K/88K [52]	335K/924K [52]	317K/1M [52]
UndirPRank	183.85%	173.47%	173.24%	173.85%	364.28%	285.37%	372.38%
SimRank	124.38%	137.53%	127.38%	136.43%	385.28%	295.37%	336.49%
Collusionrank	173.43%	174.28%	163.48%	172.48%	382.59%	264.69%	317.48%

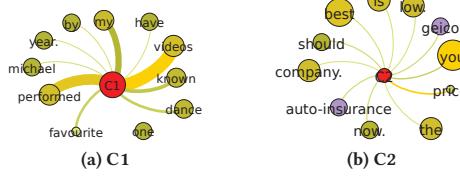


Figure 10: Provenance graphs for C1 and C2

the word/comment being spam, while the weight of an edge represents the partial derivative. As shown in the graph, C1's score is determined by 11 words. It is strange that the word `videos`, which is very common in benign YouTube comments, has a high score and is the most influential node in the classification decision. Moreover, the stop word `my` is the 3rd most influential word while we expected such words should have been filtered out.

The classifier produces a score of 0.129468956748 for C2, which is a false negative. The provenance graph in Figure 10b indicates that the words `Geico` and `auto-insurance` are assigned a pre-defined score value of 0.4, which the model assigns to any word not appearing in the training set. However, we argue that the default value undermines the influence of these two words, which are strong indicators of spam. In fact, when cross-checking the words used in benign and spam comments with the most commonly used 5,000 English words, we find that spam comments are more likely to use uncommon words such as company names.

Based on the above analysis, we improve the model by adding a pre-processing step to filter highly common words and stop words, and initializing the default score of unknown words to 0.8. Table 4 shows the performance before and after our improvement. The first

Table 4: Models for spam comments detection

	Original		New		Total	
	Spam	Benign				
			Spam	Benign		
Spam	523	214	724	13	737	
Benign	2026	2902	23	4905	4928	
Total	2549	3116	747	4918	5665	

row reads as follows: 523 spams are classified as spam and 214 spams are classified as benign by the original model, and the numbers become 724 and 13, respectively by the new model. Observe that the new model has much smaller false positive and false negative rates. We also use LAMP on a few other public spam filtering models to improve performance. The results are shown Table 5. For these models, we can improve their accuracy from 80%– to 90%+.

Table 5: Model Evolution

Model	Accuracy		Model	Accuracy	
	Original	New		Original	New
SpamFilter [18]	78.46%	96.27%	BayesSpam [2]	72.34%	94.32%
NBSF [19]	74.82%	98.28%	BayesianFilter [5]	57.23%	92.35%
AntiSpam [1]	69.23%	96.37%	SMS Filter[17]	60.27%	98.27%

## 7.2 Model Error Debugging

Bayesian networks are widely used in decision making such as diagnosing cancer [32, 47, 51, 62]. In this case, we use a lung cancer diagnosing network [62] with 413 nodes to demonstrate how LAMP can be used in debugging faults in models. We inject a fault to the model by changing a few conditional probabilities related to gender (e.g.,  $P(\text{coughing}|\text{gender} = \text{male}, \dots)$  from 0.01 to 0.9). We then apply the model to a public cancer data set [11]. We encounter a number of misdiagnosis cases with the faulty model. Note that

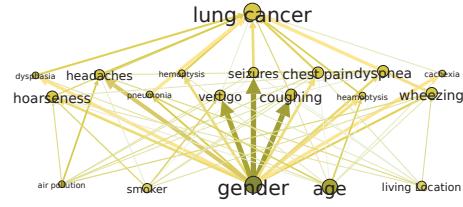


Figure 11: Provenance for model bugs

due to the model complexity, it is really difficult to spot the fault by inspecting the model. For instance, the `lung cancer` node is connected to 64 nodes, which are further connected to 278 nodes and `gender` is one of these nodes. We then use LAMP to generate the provenance for the misdiagnosis in Figure 11. From the graph, the decision is evenly attributed to many nodes. Nothing seems suspicious. But when we further investigate the provenance of the nodes connected to `lung cancer`, we find that a few of them have (unexpectedly) heavy influence from `gender`, indicated by the thick arrows, in comparison with other second layer nodes such as `age`. This suggests that the faulty conditional probabilities of `gender` (transitively) lead to the wrong decision. We inject 12 other bugs into the model by changing the network structure or modifying the probabilities for other nodes/edges. The provenance graphs by LAMP are always able to point to the faulty places.

### 7.3 Debugging GML Implementation

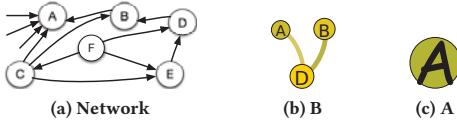


Figure 12: PageRank Debugging Example

Debugging ML programs can be hard because sometimes, we do not have clear constraints to determine if the returned results are correct. In this case, we show how LAMP is used to discover that a popular Python PageRank implementation on Github (with 50+ forks and 60+ stars) is buggy. While the implementation produces very reasonable ranking results for some undirected graph data sets, we observed some seemingly incorrect results on some directed graph data sets. We apply LAMP to further analyze the suspicious results. Figure 12a shows an example input network connection graph. The PageRank implementation generates a very low rank for node A, which does not seem right. The provenance for A's rank (Figure 12c) indicates that other nodes have no impact on A, which is buggy. In contrast, Figure 12b shows that node D has influence from nodes A and B, but not from C, E or F, to which it is also connected. From the two provenance graphs, it becomes clear that the implementation is considering outgoing edges as incoming edges. It hence does not work properly for directed graphs.

Table 6: Bug List

Bug	Provenance
Constructing a wrong graph structure [13]	Wrong dependencies
Feeding a directed graph to an algorithm designed for undirected [21]	Missing dependencies
Missing initial values leads to a nan output [7]	Missing dependencies
Missing computation steps leads to incorrect output [4]	Missing dependencies
Getting the wrong dependencies during computation [6]	Wrong dependencies

We also reproduce 5 other bugs reported on StackOverflow and Github, and use LAMP to help identify the root causes. In Table 6, we show the brief description of the bugs, and the corresponding anomalies in the provenance graphs. Details are in [10].

## 8 RELATED WORK

LAMP is inspired by AD, but differs from AD. First, AD [23, 24, 30, 36–39, 49, 60, 72] computes derivatives alongside with the original computation whereas LAMP decouples the two so that provenance computation can be activated on demand. Second, AD cannot reason about output variations caused by control flow differences while LAMP can. Third, AD typically computes derivatives regarding some inputs but LAMP considers all inputs. LAMP also has a number of optimizations specific to provenance computation.

Gleich et al. [34] studied the PageRank algorithm sensitivity with respect to the damping parameter. A few projects study the behavior of various PageRank algorithms for multiple values of the damping parameter [35, 48]. Zhang et al. [71] infer spam pages by investigating PageRank with different damping parameters. Their argument is that spam pages should be sensitive to a given damping parameter, thus changing it will disclose them. There are also other works [29, 57, 64] trying to develop testing techniques for machine learning algorithms.

Other approaches such as [43, 44, 53, 58] aim to support data provenance in DISC systems. [43] provides a general wrapper for MapReduce jobs providing data provenance capabilities. Matteo et al. [44] proposed Titian, a general provenance collection system for the Spark system, that enables data scientists to interactively trace through the intermediate data of a program execution and identify the input data at the root cause of a potential outlier or bug. While the intermediate results define the provenance for Titian, LAMP calculates the partial derivative as provenance, which can quantitatively measure the output sensitivity with regards to the input and produce precise and succinct dependence relationships with low overhead. These proposed approaches are system specific. LAMP is not bound to any data processing systems.

## 9 CONCLUSION

In this paper, we propose LAMP, a data provenance computation technique for GML algorithms. It features the capability of quantifying input importance. It is inspired by AD techniques and goes beyond them, by decoupling derivative computation from the original computation and supporting control flow path variations. The experimental results show that LAMP is much more efficient and effective than program dependence tracing based techniques. The results by LAMP can be used in optimizing the machine learning models, debugging implementations and debugging data bugs.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668, 1320444, and 1320306, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

## REFERENCES

- [1] *Anti Spam*. <https://github.com/dinever/antispam>.
- [2] *Bayes Spam*. <https://github.com/SunnyMarkLiu/NaiveBayesSpamFilter>.
- [3] *Bayesian classifier*. <https://github.com/browning/comment-troll-classifier>.
- [4] *Bayesian Classifier Bug*. <http://stackoverflow.com/questions/19349567/error-in-naive-bayes-classifier>.
- [5] *Bayesian Spam Filter*. <https://github.com/aashishsatya/Bayesian-Spam-Filter>.
- [6] *Bugs in ByeasianModel*. <https://github.com/pgmpy/pgmpy/issues/607>.
- [7] *Dirichlet Concentration : Error 73*. <https://github.com/bayespy/bayespy/issues/73>.
- [8] *Graph Tool*. <https://graph-tool.skewed.de>.
- [9] *KDD Cup 2012 Track1*. <https://www.kaggle.com/c/kddcup2012-track1>.
- [10] *LAMP*. <https://github.com/PythonLAMP/LAMP>.
- [11] *National Cancer Database - American College of Surgeons*. <https://www.facs.org/quality%20programs/cancer/ncdb>.
- [12] *NetworkX*. <https://networkx.github.io>.
- [13] *PageRank toy example fails to converge*. <http://stackoverflow.com/questions/30017913/pagerank-toy-example-fails-to-converge>.
- [14] *A Plan For Spam*. <http://www.paulgraham.com/spam.html>.
- [15] *Python Memory Profiler*. [https://pypi.python.org/pypi/memory\\_profiler](https://pypi.python.org/pypi/memory_profiler).
- [16] *Scrapy*. <https://scrapy.org/>.
- [17] *SMS Spam Filter*. <https://github.com/revantkumar/SMS-Spam-Classifier>.
- [18] *Spam Filter*. <https://github.com/jameshwang2013/SpamFilter>.
- [19] *Spam Filter*. <https://github.com/SunnyMarkLiu/NaiveBayesSpamFilter>.
- [20] *Tencent 2012 KDD Cup*. <http://www.kddcup2012.org/>.
- [21] *TextRank using NetworkX*. <http://stackoverflow.com/questions/9247538/textrank-complementing-pagerank-for-sentence-extraction-using-networkx/9247791>.
- [22] *Wikipedia Data Set*. <https://dumps.wikimedia.org/enwiki/20160601/>.
- [23] Joel Andersson, Johan Åkesson, and Moritz Diehl. 2012. CasADI: A symbolic package for automatic differentiation and optimal control. In *Recent Advances in Algorithmic Differentiation*. Springer, 297–307.
- [24] Christian Bischof, Lucas Roh, and Andrew Mauer-Oats. 1997. ADIC: an extensible automatic differentiation tool for ANSI-C. *Urbana* 51 (1997), 61802.
- [25] Soumen Chakrabarti. 2007. Dynamic personalized pagerank in entity-relation graphs. In *Proceedings of the 16th international conference on World Wide Web*. ACM, 571–580.
- [26] Hung-Hsuan Chen and C Lee Giles. 2013. ASCOS: an asymmetric network structure context similarity measure. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. ACM, 442–449.
- [27] Tianqi Chen, Linpeng Tang, Qin Liu, Diyi Yang, Saining Xie, Xuezhi Cao, Chunyang Wu, Enpeng Yao, Zhengyang Liu, Zhansheng Jiang, and others. 2012. Combining factorization model and additive forest for collaborative followee recommendation. *KDD CUP* (2012).
- [28] Juan José Conti and Alejandro Russo. 2010. A taint mode for python via a library. In *Nordic Conference on Secure IT Systems*. Springer, 210–222.
- [29] Sebastian Elbaum and David S. Rosenblum. 2014. Known Unknowns: Testing in the Presence of Uncertainty. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 833–836. DOI: <http://dx.doi.org/10.1145/2635868.2666608>
- [30] David A Fournier, Hans J Skaug, Johnoel Ancheta, James Ianelli, Arni Magnusson, Mark N Maunder, Anders Nielsen, and John Sibert. 2012. AD Model Builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models. *Optimization Methods and Software* 27, 2 (2012), 233–249.
- [31] Edward I George and Robert E McCulloch. 1993. Variable selection via Gibbs sampling. *J. Amer. Statist. Assoc.* 88, 423 (1993), 881–889.
- [32] Olivier Gevaert, Frank De Smet, Dirk Timmerman, Yves Moreau, and Bart De Moor. 2006. Predicting the prognosis of breast cancer by integrating clinical and microarray data with Bayesian networks. *Bioinformatics* 22, 14 (2006), e184–e190.
- [33] Saptarshi Ghosh, Bimal Viswanath, Farshad Kooti, Naveen Kumar Sharma, Gautham Korlam, Fabricio Benevenuto, Niloy Ganguly, and Krishna Phani Gummadi. 2012. Understanding and Combating Link Farming in the Twitter Social Network. In *Proceedings of the 21st International Conference on World Wide Web (WWW '12)*. ACM, New York, NY, USA, 61–70. DOI: <http://dx.doi.org/10.1145/2187836.2187846>
- [34] David Francis Gleich and Michael Saunders. 2009. Models and algorithms for pagerank sensitivity. *...*, Sept (2009).
- [35] G. H. Golub and C. Greif. 2006. An Arnoldi-type algorithm for computing page rank. *BIT Numerical Mathematics* 46, 4 (2006), 759–771. DOI: <http://dx.doi.org/10.1007/s10543-006-0091-y>
- [36] Andreas Griewank. 1991. Achieving Logarithmic Growth Of Temporal And Spatial Complexity In Reverse Automatic Differentiation. (1991).
- [37] Andreas Griewank. 2000. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.
- [38] Andreas Griewank and others. 1989. On automatic differentiation. *Mathematical Programming: recent developments and applications* 6, 6 (1989), 83–107.
- [39] Andreas Griewank, David Juedes, and Jean Utke. 1996. Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software (TOMS)* 22, 2 (1996), 131–167.
- [40] Vince Grolmusz. 2015. A note on the pagerank of undirected graphs. *Inform. Process. Lett.* 115, 6 (2015), 633–634.
- [41] GrouphLens. *MovieLens Datasets*. <http://grouplens.org/datasets/movielens/>.
- [42] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 784–795.
- [43] Robert Ikeda, Hyunjung Park, and Jennifer Widom. 2011. Provenance for Generalized Map and Reduce Workflows. In *CIDR*.
- [44] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titan: Data Provenance Support in Spark. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 216–227. DOI: <http://dx.doi.org/10.14778/2850583.2850595>
- [45] Glen Jeh and Jennifer Widom. 2002. SimRank: a measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 538–543.
- [46] Meng Jiang, Peng Cui, Rui Liu, Qiang Yang, Fei Wang, Wenwu Zhu, and Shiqiang Yang. 2012. Social contextual recommendation. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 45–54.
- [47] Charles E Kahn, Linda M Roberts, Katherine A Shaffer, and Peter Haddawy. 1997. Construction of a Bayesian network for mammographic diagnosis of breast cancer. *Computers in biology and medicine* 27, 1 (1997), 19–29.
- [48] Sepandar Kamvar, Taher Haveliwala, and Gens Golub. 2004. Adaptive methods for the computation of PageRank. *Linear Algebra Appl.* (2004).
- [49] Gershon Kedem. 1980. Automatic differentiation of computer programs. *ACM Transactions on Mathematical Software (TOMS)* 6, 2 (1980), 150–165.
- [50] Gyanendra Kumar, Neelam Duhan, and AK Sharma. 2011. Page ranking based on number of visits of links of Web page. In *Computer and Communication Technology (ICCT), 2011 2nd International Conference on*. IEEE, 11–14.
- [51] S. L. Lauritzen and D. J. Spiegelhalter. 1990. Readings in Uncertain Reasoning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems, 415–448. <http://dl.acm.org/citation.cfm?id=84628.85343>
- [52] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data>. (June 2014).
- [53] Dionysis Logothetis, Soumyarupa De, and Kenneth Yocum. 2013. Scalable lineage capture for debugging disc analytics. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 17.
- [54] Lu Lu, Xuanhua Shi, Yonglun Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. 2016. Lifetime-based Memory Management for Distributed Data Processing Systems. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 936–947. DOI: <http://dx.doi.org/10.14778/2994509.2994513>
- [55] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 281–292. DOI: <http://dx.doi.org/10.14778/2735508.2735517>
- [56] Rada Mihalcea and Paul Tarau. 2004. TextRank: Bringing order into texts. Association for Computational Linguistics.
- [57] Christian Murphy. 2010. *Metamorphic testing techniques to detect defects in applications without test oracles*. Columbia University.
- [58] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 1099–1110. DOI: <http://dx.doi.org/10.1145/1376616.1376726>
- [59] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The PageRank citation ranking: bringing order to the web. (1999).
- [60] Louis B Rall. 1981. Automatic differentiation: Techniques and applications. (1981).
- [61] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015. SSTIC*, 31–54.
- [62] M Berkay Sesen, Ann E Nicholson, Rene Banares-Alcantara, Timor Kadir, and Michael Brady. 2013. Bayesian networks for clinical decision support in lung cancer care. *Plos one* 8, 12 (2013), e82349.
- [63] Neelam Tyagi and Simple Sharma. 2012. Weighted Page Rank algorithm based on number of visits of Links of web page. *International Journal of Soft Computing and Engineering (IJSCSE) ISSN* (2012), 2231–2307.
- [64] Xiaoyuan Xie, Joshua W. K. Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and Validating Machine Learning Classifiers by Metamorphic Testing. *J. Syst. Softw.* 84, 4 (April 2011), 544–558. DOI: <http://dx.doi.org/10.1016/j.jss.2010.11.920>

[65] Wenpu Xing and Ali Ghorbani. 2004. Weighted pagerank algorithm. In *Communication Networks and Services Research, 2004. Proceedings. Second Annual Conference on*. IEEE, 305–314.

[66] Zhaogui Xu, Peng Liu, Xiangyu Zhang, and Baowen Xu. 2016. Python Predictive Analysis for Bug Detection. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 121–132. DOI : <http://dx.doi.org/10.1145/2950290.2950357>

[67] Fan Yang, Jinfeng Li, and James Cheng. 2016. Husky: Towards a More Efficient and Expressive Distributed Computing Framework. *Proc. VLDB Endow.* 9, 5 (Jan. 2016), 420–431. DOI : <http://dx.doi.org/10.14778/2876473.2876477>

[68] Jonathan S Yedidia, William T Freeman, Yair Weiss, and others. 2000. Generalized belief propagation. In *NIPS*, Vol. 13. 689–695.

[69] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[70] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. <http://dl.acm.org/citation.cfm?id=2228298.2228301>

[71] Hui Zhang, Ashish Goel, Ramesh Govindan, Kahn Mason, and Benjamin Van Roy. 2004. Making Eigenvector-Based Reputation Systems Robust to Collusion.. In *WAW (Lecture Notes in Computer Science)*, Stefano Leonardi (Ed.), Vol. 3243. Springer, 92–104.

[72] Xiao Jian Zhou and Ting Jiang. 2016. Enhancing Least Square Support Vector Regression with Gradient Information. *Neural Processing Letters* 43, 1 (2016), 65–83.