Navigating the Data Lake with Datamaran: Automatically Extracting Structure from Log Datasets

Yihan Gao  Silu Huang  Aditya Parameswaran
University of Illinois—Urbana-Champaign

ABSTRACT
Organizations routinely accumulate semi-structured log datasets generated as the output of code; these datasets remain unused and uninterpreted, and occupy wasted space—this phenomenon has been colloquially referred to as "data lake" problem. One approach to leverage these semi-structured datasets is to convert them into a structured relational format, following which they can be analyzed in conjunction with other datasets. We present Datamaran, an tool that extracts structure from semi-structured log datasets with no human supervision. Datamaran automatically identifies field and record endpoints, separates the structured parts from the unstructured noise or formatting, and can tease apart multiple structures from within a dataset, in order to efficiently extract structured relational datasets from semi-structured log datasets, at scale with high accuracy. Compared to other unsupervised log dataset extraction tools developed in prior work, Datamaran does not require the record boundaries to be known beforehand, making it much more applicable to the noisy log files that are ubiquitous in data lakes. In particular, Datamaran can successfully extract structured information from all datasets used in prior work, and can achieve 95% extraction accuracy on automatically collected log datasets from GitHub—a substantial 66% increase of accuracy compared to unsupervised schemes from prior work.

1 INTRODUCTION
Enterprises routinely collect semi-structured or partially structured log datasets in shared file systems such as HDFS. These datasets are typically generated automatically as log datasets output by programs, and often number in the billions, e.g., Google has 26B datasets in their shared file system [24]. This phenomenon of accumulation of log datasets within enterprises has recently been referred to as the "data lake" problem [23, 45, 46]. Unfortunately, the datasets in a data lake often remain unused, unstructured, and uninterpreted, and as they accumulate, they become unmanageable—recent work has characterized this data lake problem as one of the most important challenges facing large enterprises today [38, 45].

The first step to making these log datasets more useful is to convert them into a structured (relational) format. Once we have structured these datasets, we can then infer relationships across datasets, and put them to use to aid analysis, search, or browsing [9, 10, 17, 32, 40, 48, 49, 51]. The goal of this paper is to automatically, efficiently, and accurately extract structure from log datasets, enabling us to tap into and put to use the log datasets in large enterprise data lakes.

Why Not Use Prior Work? Given the vast volumes of related work on information extraction [39], one may be tempted to ask: doesn’t that solve the problem? Unfortunately, as we will describe in more detail in Section 6, much related work on general HTML wrapper induction, e.g., [16, 25, 26, 34–36], HTML list-based extraction, e.g., [22, 33], and others, e.g., [31, 41], requires training examples or a corpus of entities to be provided. A relatively smaller body of work exists on unsupervised extraction, from general HTML pages [6, 43, 44], and HTML lists [11, 13, 50]. The former crucially relies on the HTML DOM tree, opting to identify recurrent tree patterns; and the latter relies on having each list item corresponding to a record. Log datasets unfortunately do not correspond to a tree structure; and records in log datasets often span multiple lines, making it hard to identify record boundaries—also, there are often multiple types of records in log datasets, as well as noise or other formatting, making it hard to apply the HTML list techniques.

Perhaps the most related body of work is on log dataset extraction itself. Work from program synthesis has developed techniques to perform extraction or transformation from examples [19, 20, 27, 30], while some others [28, 37] requires users to provide the transformation steps; instead, we are opting for a fully unsupervised approach. Fisher et al. [15] took one step towards automation by only requiring that users provide record boundaries. The tool Recordbreaker [2] is a simple automated implementation of Fisher et al.’s technique that assumes that each record occupies the same number of lines. Indeed, this is far too drastic an assumption to retain applicability in a data lake scenario, as we will see.

Example 1.1 (Illustrative Example of Real Log Dataset). Compared with extracting structure from dataset with clearly defined cues, e.g., HTML tags, or record boundaries, there are some unique challenges when dealing with log datasets. Past work on data extraction from log files [2, 15], assumes that the data is already chunked beforehand using external tools. That is, the dataset has been partitioned into many small blocks, such that each block contains exactly one record. This chunking step is assumed to be straightforward (e.g., when each record contains exactly k lines), and prior work primarily focus on learning structure given the blocks. Unfortunately, most log datasets cannot be easily chunked, since there are usually multiple types of records, making the span of such records (i.e., the number of lines each record occupies) non-constant. Consider an example log dataset crawled from GitHub in Figure 1, in which there are two types of records consisting of 7 and 9 lines respectively, randomly interspersed with each other. Since the sequence of record types can be arbitrary, it is no longer possible to identify the boundaries...
of records using simple rules, rendering prior unsupervised log structure extraction algorithms non-applicable\(^\text{1}\).

**DATAMARAN: Automatic Log Structure Extraction.** In this paper, we present DATAMARAN\(^\text{2}\), an automatic log dataset structure extraction algorithm.

At a high-level, the idea behind DATAMARAN is simple: DATAMARAN identifies the correct structure of the dataset by looking for repeated patterns; we examine small portions of the dataset and use a hash-table to find repeated patterns covering a significant fraction of the dataset. All such patterns are then evaluated via some scoring function, such as the minimum description length [8] (Note, however, that DATAMARAN is general, and can adapt to any scoring modality, not just minimum description length). Finally, the best pattern is used to actually extract structured information from the dataset.

However, a naive implementation of this algorithm, as we will demonstrate, can lead to a huge blowup in the number of patterns considered, and therefore the time taken for extraction; as a result, DATAMARAN requires careful design and engineering to bound the computation at each step. We developed techniques to address the following challenges we encountered when applying the above high-level idea on log datasets:

- **Unknown Record Endpoints.** As described previously, identifying the boundaries of records is not straightforward; while the end-of-line character ‘\n’ is used for separating records in many datasets, it could also appear within records (i.e., multi-line records).
- **Unknown Field Endpoints.** When trying to detect repetitive patterns, it is necessary to first separate the formatting characters from the field values. This is not as easy in log datasets, due to the fact that commonly used formatting characters (e.g., the space character ‘ ‘) can also appear within field values (e.g., text fields).
- **Complex Structure.** There are often complex structures within records: for example, if a record contains a list of values, the number of values can vary from record to record, which makes even the underlying formatting vary between records, and therefore the same pattern not applying across the dataset. Indeed, like our example demonstrates, multiple record types may also exist within the dataset. Furthermore, substructures could also exist within the structures via nesting. This makes detecting repetitive patterns substantially more difficult.
- **Redundant Structure.** During the early stages of extraction we often find a number of different repetitive patterns; of which most are completely useless (e.g., the trivial pattern that extracts the entire dataset). The number of such patterns can blow up very quickly as the structure becomes complex: for example, the date component YYYY-MM-DD can be identified as either a single field or three different fields, and different combinations of such kind of choices would yield exponentially many patterns. We need an efficient method for filtering out most of the low-quality patterns without evaluating them.
- **Structure Semantics.** Structure extraction is not simply about identifying patterns that can partition the identified records into formatting components (or delimiters), and various pieces of information to be extracted, as the ultimate goal is to transform the log datasets into structured relational datasets. Finding an appropriate structure for this purpose (i.e., making sure that resulting structured datasets are interpretable to users) requires not only a good scoring metric, but also well-designed structure refinement techniques.

Overall, DATAMARAN can automatically extract structured datasets from log datasets without any human supervision. Compared to unsupervised adaptations of semi-supervised structure extraction systems [2, 15], DATAMARAN makes fewer assumptions regarding the structure of the dataset, and therefore is much more applicable towards extracting from log datasets: as shown in our experimental evaluation, DATAMARAN can successfully extract structure from all of the datasets used in Fisher et al.’s work [15], and can achieve 95% extraction accuracy on automatically collected log datasets from GitHub, while RecordBreaker [2] can only achieve 29% extraction accuracy on the same dataset collection — a substantial 66% increase. DATAMARAN is also efficient and scales well to large datasets: the average running time for small datasets (< 50MB) is less than 20 seconds; even for datasets of size more than 100MB, DATAMARAN can still complete extraction within a few minutes. The main time spent by DATAMARAN for large datasets is in extraction (which is eminently parallelizable), and identifying an appropriate structure can be done much faster.

**Paper Outline.** The rest of this paper is organized as follows:

- In Section 2, we formally define the problem of unsupervised structure extraction.
- In Section 3, we identify key assumptions that will help us solve the problem in a tractable manner. We also compare the assumptions made in our work with those in prior works to
demonstrate why DATAMARAN is better tailored towards structure extraction from log datasets.

- In Section 4, we present DATAMARAN, our structure extraction algorithm, and provide theoretical analysis on time complexity and correctness.
- In Section 5, we experimentally study the performance of our algorithm on 25 typical datasets and automatically collected log datasets from GitHub, and demonstrate the efficiency, effectiveness, and robustness of DATAMARAN for log dataset structure extraction.

2 PROBLEM DEFINITION

In this section, we formally define the problem of (unsupervised) structure extraction from log datasets and introduce related concepts. We begin by defining the concepts of record templates and instantiated records:

Definition 2.1 (Record Template/Instantiated Record). A record template is a string that contains one or more instances of the field placeholder character—a special type of character, denoted as F in this paper—along with other characters. An instantiated record is a string with no field placeholder character.

We say an instantiated record can be generated from a record template if it can be constructed by replacing field placeholder characters in the record template with strings containing no field placeholder characters.

Given an instantiated record and a record template, we can now define the concept of field values as follows:

Definition 2.2 (Field Values). For any pair of instantiated record R and record template RT, if R can be generated from RT then the replacement strings in R for the field placeholder characters are called the field values of R for RT. When the context is clear, we simply call them the field values of R or just the field values.

Figure 2: Record Template Illustration

Figure 3: Structural Uncertainty of Record Templates

These definitions are illustrated via an example in Figure 2. As we can see, the instantiated records on the left hand side are generated by replacing the placeholder character ‘F’ in the record template on the right hand side with concrete values. The data items replacing the placeholder character (e.g., 192, 168, 0, 1, . . .) are the field values to be extracted from the dataset.

25

Figure 4: Log dataset illustration

There are often many record templates that could correspond to a given dataset. Figure 3 illustrates an example wherein the corresponding record templates (i.e., the right hand side) are similar but not exactly the same. To characterize this scenario, we define the concept of a structure template:

Definition 2.3 (Structure Template). A structure template is a regular expression [42] for record templates. We say the record template RT can be generated from the structure template ST iff the regular expression of ST matches the string form of RT.

Intuitively, a structure template captures minor variations in the structure of records within a dataset via a regular expression. The bottom of Figure 3 shows an example structure template corresponding to the records in the top, capturing minor differences in the record templates such as one, two, or three arguments within parentheses.

Now, we define the concept of a log dataset:

Definition 2.4 (Log Dataset). A log dataset D = {T, S} consists of two components: the textual component T and the structural component S. S = {ST1, ST2, . . . , STk} is a collection of structure templates, and T is a text dataset with the following structure: T can be partitioned into several blocks separated by the end-of-line character ‘\n’, and each block is either an instantiated record generated from one of the structure templates in S, or corresponds to a noise string with no structure.

Figure 4 illustrates an example log dataset. The parts with a gray background are noise blocks, while the other parts are record blocks. Noise blocks have no structure within, and are not relevant to the structure extraction problem. The requirement that blocks are separated by end-of-line characters in Definition 2.4 is reasonable for log datasets: it seems to be a common practice for programmers to write ‘\n’ character at the end of every log line (it holds for every log dataset we have examined). Notice however, that it is not necessary for a record to span just one line, such as the example in Figure 1: we only require that the structured components and noise are clearly demarcated.

To formalize the structure extraction problem, we start with an intuitive formulation:

Problem 1 (Structure Extraction). For a log dataset D = {T, S} with only T observed but S unknown, recover S and the field values of the instantiated records in T.

Note that Problem 1 is not well-posed: for any given text component T, there are infinitely many potential structural components S such that the pair (T, S) obeys Definition 2.4 (for example, the simplest structure template "\n" can pair with any textual component to satisfy Definition 2.4). Most of these structures are unacceptable from an end-user’s point of view. In practice, the structure...
extraction algorithm needs to discover the most plausible one by designing a scoring system that assigns scores to \((T, S)\) pairs. The scoring system is intended to mimic human judgment: a better score implies that the structure is more plausible from an end-user’s point of view. We also adopt this approach in Datamaran, and the precise regularity score function \(F(T, S)\) we use will be discussed later. Thus, an optimization based formulation of the structure extraction problem is as follows:

**Problem 2 (Structure Extraction (Optimization)).** For a log dataset \(D = (T, S)\) with only \(T\) observed but \(S\) unknown, find \(S\) that optimizes a given regularity score function \(F(T, S)\), and extract all the field values of the instantiated records in \(T\).

3 **STRUCTURAL ASSUMPTIONS**

In Section 2, we formalized the structure extraction problem as finding the structural component \(S\), i.e., a collection of regular expressions, that best explains or generates the textual component \(T\), i.e., the one that achieves the highest regularity score \(F(T, S)\). However, in practice, it is computationally infeasible to search over the entire space of all possible regular expressions. Therefore, it is necessary for structure extraction systems—even semi-supervised ones—to make additional assumptions on the structural component \([15, 30]\). These assumptions restrict the search space of potential structure templates, thereby serving the following two purposes:

- To enforce human intuition upon the searching procedure. Structure templates following such assumptions are more likely to be the acceptable from an end-user’s point of view. In particular, log files have a regular repeating structure, since they were generated by a computer program and meant to contain all relevant information to be extracted via a computer program or script. Our assumptions serve to codify these principles.

- To reduce the complexity of search space of the structural component, making the structure extraction problem more tractable.

In Datamaran, we make three assumptions regarding the structure of the dataset. These three assumptions will be introduced in the rest of this section. The validity of these assumptions will be experimentally verified in Section 5.3 through examination of a large collection of log datasets in GitHub. We will also compare these assumptions with the ones made by RecordBreaker \([2]\), an unsupervised structure extraction algorithm, at the end of this section.

Note that the following three subsections correspond to the three assumptions made in Datamaran. Readers may skip to Section 4 first if desired, revisiting these subsections when the corresponding assumption is being referenced.

3.1 **Coverage Threshold Assumption**

Here is the first assumption, which is very intuitive:

**Assumption 1 (Coverage Threshold).** The coverage of every structure template \(ST_i \in S\) should be at least \(\alpha\%\). The coverage of structure template \(ST\) is defined as the total length (i.e., total number of characters) of the instantiated records of \(ST\).

**Explanation.** Assumption 1 states that log datasets don’t typically contain a large number of different structure templates within, and thereby each structure template should cover a significant portion of the dataset. Note that a structure template is itself a regular expression that can capture a multitude of record templates, so this is not a severe restriction. Assumption 1 makes this intuition explicit. The coverage threshold assumption allows us to prune out most unreasonable structure template candidates. We will discuss the impact of varying the parameter \(\alpha\) in our experiments.

3.2 **Non-Overlapping Assumption**

The second assumption we make is the following:

**Assumption 2 (Non-Overlapping).** For any structural template \(ST\) and any character \(c\), one of the following is true:

- for any record template \(RT\) generated from \(ST\), \(c \notin RT\).
- for any instantiated record \(R\) generated from \(ST\), no field values of \(R\) contains \(c\).

**Explanation.** Assumption 2 states that the formatting characters of records cannot be mixed with actual field values. Intuitively, this makes sense because in practice these records are manually extracted via scripts, and these scripts use delimiters to extract field values. To formally explain this, we first define some notation: we let RT-CharSet denote the set of characters in record templates, while F-CharSet denotes the set of characters in field values.

Under this notation, Assumption 2 can be simply stated as: For any structure template \(ST\), there exists two disjoint character sets \(A(ST)\) and \(B(ST)\), such that for any instantiated record \(R\) of \(ST\), we have \(RT-CharSet(R) \subseteq A(ST)\) and \(F-CharSet(R) \subseteq B(ST)\). In this paper, we further assume that \(RT-CharSet\) contains only special characters. In other words, we redefine a collection of special characters RT-CharSet-Candidate, and assume that \(RT-CharSet(R) \subseteq RT-CharSet-Candidate\) for all records \(R\).

Assumption 2 plays an important role in Datamaran: it allows us to extract the record template directly from an instantiated record given the corresponding character set of the record templates, and efficiently extract matches for a given structure template from the dataset.

**Justification of Assumption.** Assumption 2 is a relatively strong assumption. To compensate for this, the structural form assumption in Section 3.3 (discussed next) is sufficiently flexible such that even for many datasets that seemingly violate this assumption, we can still get reasonable results.

For example, consider the record template \(F, "F", \) \(F\). If the field value surrounded by the quotes contains the comma character, then \(RT-CharSet\) contains the comma character, then Assumption 2 would be violated. However, Datamaran will still be able to recognize several different record templates in the following, depending on the number of commas in the middle field value:

\[ F, "F", F \quad F, "F", F \quad F, "F", F \]

Since all the above record templates can be generated from the same structure template \(F, "(F,)\ast F", \) they will still be recognized as the same record type. We formalize the space of structure templates next.

3.3 **Structural Form Assumption**

The following assumption restricts the forms of structure templates:

**Assumption 3 (Structural Form).** Every structure template is a regular expression that has one of the following forms:
we can extract the structure from a dataset if it can be represented
in the canonical LL(1) parser in linear time.

3.4 Assumption Comparison

Here we compare the assumptions made in DATAMARAN with those in RecordBreaker [2]. The structural form assumption (Assumption 3) has an equivalent counterpart in RecordBreaker. RecordBreaker also makes a stronger version of Assumption 2, together with another additional assumption regarding record boundaries:

**Assumption 4 (Boundary).** The boundaries of records can be easily identified beforehand.

**Assumption 5 (Tokenization).** Each record can be tokenized beforehand, such that each token is either part of a field-value, or part of the structure template. In other words, in addition to Assumption 2, it is further assumed that RT-CharSet for all records are predetermined in advance:\(^3\):

\[
VR, \text{RT-CharSet}(R) = \text{RT-CharSet-Candidate}
\]

<table>
<thead>
<tr>
<th>Assumption</th>
<th>RecordBreaker</th>
<th>DATAMARAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverage Threshold</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Non-overlapping</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Structural Form</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Boundary</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Tokenization</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1: The Assumption Comparison Chart

Table 1 compares the assumptions in RecordBreaker and DATAMARAN. As discussed in the introduction, the two additional assumptions in RecordBreaker are rather restrictive for log datasets. This is further verified in our experiments: about 31% of the log datasets we automatically collected from GitHub (details in Section 5.3) do not satisfy these assumptions. In comparison, the additional assumption made in DATAMARAN is much milder: due to the coverage threshold assumption, we will only extract from “popular” structure templates rather than all of them. In most practical settings, such a restriction wouldn’t cause any problems.

4 THE DATAMARAN ALGORITHM

In Section 2, we defined the structure extraction problem as the problem of finding the structural component \(S\) that optimizes a given regularity score function \(F(T, S)\) given the observed textual component \(T\). Most prior unsupervised structure extraction algorithms [2, 13, 15] adopt the “partition-summarization” approach for this problem: the dataset is first partitioned into records using a rule that is provided to the algorithm; then, using the information gathered from these records, the algorithm tries to find a structure template that best summarizes them. However, as mentioned previously, it is very difficult to partition log datasets directly, and as a

\(^3\)The only difference between Fisher’s algorithm [15] and RecordBreaker [2] is the treatment of this assumption: Fisher et al. assume that RT-CharSet-Candidate is given by the user for each dataset; RecordBreaker compiled a predetermined character set, making their program unsupervised.
result prior approaches are unsuitable for extracting structure from log datasets.

Given the difficulty associated with identifying record boundaries, a different approach is used by Datamaran for structure extraction: Datamaran first generates a collection of structure template candidates directly from the dataset, then evaluates each of them to find the optimal one. Figure 6 illustrates the conceptual differences between Datamaran and earlier approaches. Compared to the prior "partition-summarization" approach, the advantage of Datamaran’s approach is that datasets are no longer required to be partitioned into records at the very first step. Concretely, Datamaran algorithm consists of the following three steps:

- **Generation.** The first step is to search for candidate structure templates that satisfy the coverage threshold assumption (Assumption 1).
- **Pruning.** The second step is to prune out most of the candidates found in the previous step, such that only a few candidates need to be actually evaluated against the original dataset. To achieve this, we designed an assimilation score function $G$, a built-in regularity score function that can be evaluated very efficiently. We then retain the candidates with highest assimilation score $G(T, S)$ for the final evaluation step.
- **Evaluation.** During the final step, we apply two structure refinement techniques to the remaining structure templates after the pruning step, and then evaluate their regularity score to find the one with the highest $F(T, S)$.

Figure 7 illustrates the three-step workflow of Datamaran. The first two steps of Datamaran (i.e., Generation and Pruning) are nontrivial. For the generation step, identifying the repetitive patterns within datasets is difficult, due to the potential structural variations of record templates (as described in Assumption 3) and the fact that field value boundaries are unknown. As for the pruning step, there are usually too many potential candidates after the generation step. Even with the coverage assumption (Assumption 1) under consideration, evaluating all the high coverage candidates against the dataset is still impractical. Designing an assimilation score function that can be evaluated efficiently while being able to prune out most of the low-quality candidates without complete evaluation is necessary but not straightforward.

The details of Datamaran algorithm will be discussed in the rest of this section: in Section 4.1, we describe the algorithm for efficiently finding structure templates satisfying the coverage threshold assumption (Assumption 1); in Section 4.2, we describe our approximation score function and discuss the intuition behind its design; in Section 4.3, we describe two structure refinement techniques that are applied during the evaluation step; in Section 4.4, we analyze the time complexity of Datamaran and characterize the conditions under which the correctness of Datamaran can be guaranteed. There are some additional algorithmic details of Datamaran that will not be discussed in this section due to page limitations, and they can be found in the appendix.

**The Regularity Scoring Function.** In Datamaran, we assume the regularity score function $F(T, S)$ is given, and we can access it through a function call. The design of Datamaran is independent of the choice of this scoring function: we can plug in any reasonable scoring function into Datamaran, and the algorithm would function as before. In this sense, the primary contribution of Datamaran is an efficient and scalable method to optimize any reasonable scoring function.

However, for completeness, we will present the details of the minimum description length [8] regularity score function that we use in our implementation in the appendix, and we demonstrate that it does well empirically in Section 5. That said, through the rest of this section, we assume the regularity score function is given and it mimics human judgment regarding the quality of structure templates.

**Notation.** Table 2 lists the notations used in Datamaran. The first 3 symbols are parameters in Datamaran, while the last 5 symbols represent dataset-dependent values. We will describe each of these parameters later on.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>The number of structure templates retained after the pruning step</td>
</tr>
<tr>
<td>$L$</td>
<td>The maximum span of records (i.e., the maximum number of lines each record can span)</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>The minimum coverage threshold for records</td>
</tr>
<tr>
<td>$n$</td>
<td>The total number of lines in the dataset</td>
</tr>
<tr>
<td>$K$</td>
<td>The number of structure templates retained after the generation step</td>
</tr>
<tr>
<td>$T_{data}$</td>
<td>The total size of the dataset</td>
</tr>
<tr>
<td>$S_{data}$</td>
<td>The amount of data sampled during all three steps</td>
</tr>
<tr>
<td>$c$</td>
<td>The number of special characters (i.e., characters in $RT$-CharSet-Candidate) appearing in the dataset</td>
</tr>
</tbody>
</table>

Table 2: Notation Summary

### 4.1 The Generation Step

In the generation step, we find structure templates satisfying Assumption 1 (i.e., those with at least $\alpha\%$ coverage). At a high level, this is achieved by finding repetitive patterns within the dataset. Specifically, Datamaran uses the following five steps to find structure templates with at least $\alpha\%$ coverage:

1. Enumerate possible values of $RT$-CharSet (i.e., the character set in the record templates), and for each such value of $RT$-CharSet, run through steps 2-5.
2. Enumerate all $O(nL)$ pairs of end-of-line characters ‘\n’ that are close to each other (i.e., at most $L$ lines are between them) in the textual component $T$. For each such pair, treat the content between each pair as an instantiated record, and run steps 3-4.
3. Extract the record template from the instantiated record using the value of $RT$-CharSet. Due to the non-overlapping assumption (Assumption 2), it is possible to separate the field values from instantiated records given the value of $RT$-CharSet.
4. Reduce the record template into a structure template (with the form defined in Assumption 3).
5. Store all of the structure templates generated in step 4 within a hash-table, and then find the ones that satisfy the coverage threshold assumption.

Figure 8 illustrates the workflow of the generation step. As we can see, the input is first transformed into many small potential
with pseudo-code, can be found in Section 8.1 in the appendix.

There are two different versions of the Variants of Generation:

- Exhaustive version enumerates all possible values while greedy version only enumerates a subspace of possible values. Intuitively, these two searching procedures represent a trade-off between accuracy and efficiency:
  - The exhaustive search is slower but gives us better extraction results.
  - The greedy version is faster but may miss some true structure templates.

The intuition behind our algorithmic design here is to get around the difficulty associated with identifying the record boundaries: since we are enumerating all possibilities, it is guaranteed that the actual structure template will appear enough times. On the other hand, we use the coverage threshold assumption (Assumption 1) to filter out noise (i.e., structure templates derived from noise blocks and incorrect boundaries). In this way, we can limit the total size of the candidate pool, while ensuring that the correct structure template still remains within the candidates.

The reader may have noticed that this step may be very expensive for large datasets, and we have used sampling method to ameliorate this. All of the technical details of the generation step, together with pseudo-code, can be found in Section 8.1 in the appendix.

**Variants of Generation:** There are two different versions of the first sub-step implemented in DATAMARAN: exhaustive version enumerates all possible values while greedy version searches only a subspace of possible values. Intuitively, these two searching procedures represent a trade-off between accuracy and efficiency: the exhaustive search is slower but gives us better extraction results. More details can be found in appendix.

### 4.2 The Pruning Step

Even with the coverage threshold assumption, there are often far too many structure template candidates remaining after the generation step. As a result, it is impossible to evaluate regularity score $F(T, S)$ for every single one. The purpose of the pruning step is to identify a small promising subset of these candidates to be evaluated in the final evaluation step, and discard the rest.

In the pruning step, we use assimilation score $G(T, S)$ to order the structure templates, so that only the best $M$ ones need to be evaluated explicitly in the evaluation step. The assimilation score estimates the amount of data “assimilated” by the structure template (i.e., the amount of data that can be explained by the structure template). Therefore, structure templates with a higher assimilation score are more likely to also have a higher regularity score.

Before we describe the actual design of our assimilation score function, it is helpful to first understand why there are so many structure templates remaining after the generation step. It turns out that most of the redundant structure templates come from two sources as demonstrated in Figure 9: (a) when the structure template consists of multiple lines (line 1-5 in Figure 9 left), any subset of such a structure template would also be captured by the generation step as a legitimate structure template (line 2-4 in Figure 9 right); (b) when the structure template uses multiple types of characters to separate the field values, simpler structure templates can be recognized if some of those characters are treated as field values as illustrated in Figure 9 (bottom).

Therefore, a good assimilation score should be able to distinguish both types of redundancies, and rank the true structure template(s) higher than the redundant ones. At the same time, it should be relatively lightweight to compute. To achieve this, our first component uses the coverage value of structure templates, since we have already computed it during the generation step. However, while the coverage value can effectively distinguish the first source of redundancy, it is not capable of distinguishing the second one. As a result, using the coverage value directly as the assimilation score will not serve our propose.

To address this shortcoming, we introduce another component into the assimilation score: the Non-Field-Coverage term, which is defined as the total coverage of the structure template minus the total coverage of all field values of the structure template (i.e., the total length covered by field values in the instantiated records). This term computes the total coverage achieved by “non-field” characters in the template, and can be effectively used to distinguish the second source of redundancy. The final assimilation score function $G(T, S)$ used in DATAMARAN is the following, which filters out all structure templates with either low coverage or low non-field-coverage.

$$ G(T, S) = \text{Cov}(T, S) \times \text{Non\_Field\_Cov}(T, S) $$
4.3 Structure Refinement

In order to further improve the extraction accuracy of DATAMARAN, we developed two techniques to refine the structure templates. These techniques are applied to all of the top M structure templates during the evaluation step: we revise these structure templates, and compare the revised structure templates against the original ones (using the regularity score function). We replace the original structure template with the revised ones when the score is improved.

4.3.1 Array Unfolding. During the generation step, all of the records are transformed into minimal structure templates, which allowed us to detect repetitive patterns within the dataset. However, there are cases where the minimum structure template is not the optimal structure template.

For instance, in comma-separated values files (*.csv files), all of the records have the form "F,F,F,....,F,F
" (i.e., a fixed number of field values separated by commas). There are two possible structure templates for these records: the plain struct-type "F,F,F,....,F,F
" and the array-type "(F,)\*F
". The plain struct-type template offers a better semantic interpretation in this case (since it explicitly states that the field values are of different types), and also leads to a better score \( F(T,S) \).

Unfortunately, the best structure template will not be found during the generation step in this case, since we have only considered the minimal structure templates in the generation step. To address this, we designed the array unfolding technique: for each array-type regular expression in the structure template, we attempt to unfold it by expanding it into a struct-type. Figure 10 demonstrates this process: the array-type regular expression at the top of the figure will be unfolded into one of the struct-type regular expression at the bottom of the figure. If any of these unfolded structure templates has a better score than the original one, the unfolding would be finalized.

Figure 10 also demonstrates partial unfolding, which is another type of array unfolding mechanism implemented in DATAMARAN. In this case, we expand the array-type regular expression while retaining the nondeterministic array-type suffix. Partial unfolding can be used to handle the cases where regular field values are “mixed in” with text field values, as in the following example:

```
Apr 24 04:02:24 srv7 snort shutdown succeeded
Apr 24 04:02:24 srv7 snort startup succeeded
Apr 24 14:44:28 srv7 Disabling nightly yum
```

In this example, the first four fields are regular fields, but the last one is a free-text field. The ideal structure template for this example is \( F,F,F,F,(F,)\*F
\), which can be obtained by applying partial unfolding to the minimum structure template \( (F,)\*F
\).

4.3.2 Structure Shifting. Typically, the regularity score function \( F(S,T) \) evaluates the quality of structure templates using statistics such as coverage value or minimum description length (see Section 8.1 in the appendix). For most cases, these kinds of score functions can distinguish good structure templates from bad ones. However, there is one ambiguity among structure templates that such a regularity score would fail to detect: the cyclic shifting of structure templates. Figure 11 illustrates this phenomenon: the regularity score \( F(T,S) \) of the shifted structure template (right hand side in Figure 11) and the score \( F(T,S) \) of the correct structure template (left hand side in Figure 11) are usually approximately equal to each other.

Unfortunately, the best structure template will not be found during the generation step in this case, since we have only considered the minimal structure templates in the generation step. To address this, we designed the array unfolding technique: for each array-type regular expression in the structure template, we attempt to unfold it by expanding it into a struct-type. Figure 10 demonstrates this process: the array-type regular expression at the top of the figure will be unfolded into one of the struct-type regular expression at the bottom of the figure. If any of these unfolded structure templates has a better score than the original one, the unfolding would be finalized.

Figure 10 also demonstrates partial unfolding, which is another type of array unfolding mechanism implemented in DATAMARAN. In this case, we expand the array-type regular expression while retaining the nondeterministic array-type suffix. Partial unfolding can be used to handle the cases where regular field values are “mixed in” with text field values, as in the following example:

```
```

In this example, the first four fields are regular fields, but the last one is a free-text field. The ideal structure template for this example is \( F,F,F,F,(F,)\*F
\), which can be obtained by applying partial unfolding to the minimum structure template \( (F,)\*F
\).

4.4 Theoretical Analysis
4.4.1 Time Complexity. Table 3 lists the time complexity of the three steps in DATAMARAN respectively. An explanation for the symbols can be found in Table 2. Note that for large datasets, we would utilize sampling for all the three steps (details can be found in Section 8.1), and therefore $S_{data}$ is upper-bounded by a large constant. In such cases, the running time of our algorithm is dominated by the actual data extraction procedure.

<table>
<thead>
<tr>
<th>Step</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generation Step</td>
<td>$O(S_{data}L^2)$ or $O(S_{data}L^2c)$</td>
</tr>
<tr>
<td>Pruning Step</td>
<td>$O(K \log K)$</td>
</tr>
<tr>
<td>Evaluation Step</td>
<td>$O(MS_{data})$</td>
</tr>
<tr>
<td>Data Extraction</td>
<td>$O(T_{data})$</td>
</tr>
</tbody>
</table>

Table 3: Time Complexity of the Three Steps in DATAMARAN

4.4.2 Correctness Guarantee. DATAMARAN is designed to tolerate noise blocks and variations within record structures and field values. From a practical perspective, it is very hard to argue the correctness of our algorithm when the field values and noise blocks can be arbitrary. Therefore, we characterized three conditions that are sufficient for guaranteeing the correctness of DATAMARAN:

**Theorem 4.1.** For a log dataset $D = \{T, S\}$ with only $T$ observed, if the following conditions are all met:

(a) One of the structure templates in $S$ (denote it as $ST_0$) has the highest coverage and non-field-coverage (defined in Section 4.2) among all structure templates.

(b) For at least $\alpha$% of the instantiated records, the minimum structure parameter for them is $ST_0$.

(c) $ST_0$ has the best regularity score among all structure templates.

Then, DATAMARAN is guaranteed to return $ST_0$ as the optimal structure template.

The proof can be found in Section 8.5 in the appendix. For most practical settings, condition (b) is automatically met. Condition (c) requires a carefully designed score function, which is not the focus of this paper. As for condition (a), intuitively it requires the structure templates in $S$ to be sufficiently different from each other, and the field values and noise blocks are sufficiently random. If all of these conditions are satisfied, then Theorem 4.1 would guarantee the correctness of DATAMARAN.

5 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate the performance of DATAMARAN. The experiments are conducted on two sets of datasets serving different purposes:

- Manually collected log datasets (Section 5.2). We collected 25 datasets, including the entire set of 15 datasets used by Fisher et al. [15] and 10 others from various sources (details in Section 5.2). These datasets cover a wide variety of structural formats and possess different characteristics (e.g., file size or structural complexity). We use these datasets to study various properties of DATAMARAN such as effectiveness, efficiency, parameter sensitivity, and scalability.

- GitHub log datasets (Section 5.3). We crawled a collection of 100 log datasets automatically from public GitHub repositories. These datasets reflect the properties of real-world data lakes. We use these datasets to study the properties of data lakes “in the wild”, as well as the utility of DATAMARAN in such settings. This collection of datasets can be viewed as a benchmark for further research.

**DATAMARAN Settings:** DATAMARAN is implemented in C++ and compiled under Visual Studio 2015. The default values for the three parameters in DATAMARAN are: $\alpha = 10\%$ (the coverage threshold parameter); $L = 10$ (the upper bound of record span); $M = 50$ (the number of remaining structure templates after the pruning step). These default values are used in all of our experiments except for Section 5.2.2 and 5.2.3, where we study the sensitivity of these parameters.

**RecordBreaker [2] Settings:** Despite our best attempts, we were unable to install or run the open-source version of RecordBreaker [2]. Therefore, we decided to faithfully reimplement RecordBreaker in C++ for our comparison. At the first step, RecordBreaker relies on a lexer to break up each record into tokens. We use the open source software Flex [1] as the lexer in our implementation. Accordingly, users need to write a Flex specification file tailored to their dataset in order to obtain a better tokenization scheme. We will compare against RecordBreaker in Section 5.3.

**Experiment Settings:** All experiments were conducted on a 64-bit Windows machine with 8-core Intel Xeon 3.40GHz CPU and 8GB RAM. All executions are single-threaded.

5.1 Evaluation Criteria

Recall that the structure extraction problem is not well-posed, and the validity of the extracted structure solely depends on the end-user. Therefore, for many datasets, there are usually multiple structures that can potentially be deemed as valid. For example, the dataset [01:05:02] 192.168.0.1 has at least the following 4 valid structure templates:

Thus, the evaluation of extraction accuracy is somewhat ambiguous. In this paper, we define the evaluation criteria by mimicking the practical scenario: for each dataset, we first identify as many intended extraction targets as possible (i.e., observable fields with potentially interesting information), then the extraction is considered successful if all of the intended extraction targets can be reconstructed by concatenating field values at specific positions. Figure 12 demonstrates an example successful extraction, in which we have two intended extraction targets (i.e., time and ip address), and DATAMARAN returns the structure template as shown in the figure. In this example, the extraction is considered successful because both intended extraction targets can be reconstructed by concatenating field values at specific positions. If, instead, the targets were extracted together, reconstructing them via concatenation would not be possible.

A more rigorous version of the above evaluation criteria can be found in Section 8.2 in appendix.
5.2 Manually Collected Datasets

The first 15 datasets in this collection come from Fisher et al.’s work [15]. Since Fisher’s collection lacks large or complex datasets (i.e., datasets containing multiple types of records or multi-line records), we also collected 10 additional datasets from the internet (e.g., the stack exchange data dump [3]) as well as from our genomics collaborators. The sources and characteristics of the 25 manually collected datasets can be found in Section 8.4 in the appendix.

Evaluation Goal. The goal of the experiments in this section is to study various properties of DATAMARAN. In Section 5.2.1, we demonstrate the extraction accuracy of DATAMARAN. In Section 5.2.2, we study the efficiency of DATAMARAN under various settings. Finally, in Section 5.2.3, we study the parameter sensitivity of DATAMARAN.

5.2.1 Extraction Accuracy. We used DATAMARAN to extract structures from the 25 datasets, and the extractions are successful for all 25 datasets based on the evaluation standard in Section 5.1. DATAMARAN correctly identified the record boundaries for all 25 datasets, without knowing the span of records and the position of noise blocks beforehand. For datasets containing multiple types of records, DATAMARAN can also correctly identify the type of each record.

Based on these results, we conclude that DATAMARAN is capable of extracting structure from a wide variety of datasets such that end-users could reconstruct any intended target field value using the extracted structures with little extra effort (in most cases no extra effort at all).

5.2.2 Running Time. We study the efficiency of DATAMARAN here. We first run DATAMARAN on the 25 datasets using the default parameters to study the connection between the characteristics of datasets (size/structural complexity) and the running time of DATAMARAN. Then, we vary the parameters to study their impact on the efficiency of DATAMARAN.

Running Time vs. Dataset Size: Figure 13a depicts the impact of the size of the dataset on the running time of DATAMARAN (using either exhaustive search or greedy search). The running time on small datasets (less than 50MB) is dominated by the generation and evaluation step. For these datasets, the average running time is 17 seconds for greedy search and 37 seconds for exhaustive search. It takes about 7 minutes for DATAMARAN to process the largest dataset here (with size 167MB), where the majority of the running time is spent on running the LL(1) parser [18] for the actual data extraction. Note that the running time of the three major steps of DATAMARAN is not affected by dataset size for large datasets (as discussed in Section 4.4.1). As we can see in Figure 13a, the extraction time is already dominated by the running time of LL(1) parser [18] (which is a necessary step for all structure extraction algorithms) even when the dataset is only moderately large (i.e., about 167MB). Further, this step is easily parallelizable. Therefore, we conclude that DATAMARAN is efficient enough in practice.

Running Time vs. Structural Complexity: Figure 13b depicts the impact of the structural complexity of the dataset on the running time of DATAMARAN. The structural complexity of datasets is characterized using the total number of structure templates with at least 10% coverage. In general, it takes a longer time for DATAMARAN to extract datasets with higher structural complexity, and the efficiency benefits of greedy search is more significant on these datasets.

Real Example. The above evaluation criteria can be better clarified via the following dataset, which is part of a real log dataset from GitHub:

```
```

Say we consider two intended extraction targets: the directory path and the list of parameter names. Here are some structures that are acceptable by the above criteria:

- F F F (F=F )xF=F\n is an acceptable structure template because the 3rd field value is exactly the directory path, and one of the field values within the array structure corresponds to the parameters.
- F-F-F F:F:F: (F/)xF (F=F )xF=F\n is also acceptable: even though the directory path is recognized as an array-type structure, the full directory path can still be reconstructed by concatenating the components within the array. In fact, this is the exact structure template found by DATAMARAN in our experiment.

On the other hand, the following structures are not acceptable by the evaluation standard:

- (F )xF\n is not an acceptable structure template because the directory path is no longer reconstructible. Note that it is not possible to only pick up field values at a specific position within an array-type structure (details can be found in appendix).
- F F F (F )xF\n is also not acceptable: in this structure template, the parameter names are "mixed up" with the parameter values, and therefore are not reconstructible.
5.3 GitHub Datasets

GitHub contains a large quantity of log datasets generated by programmers across the world. We collected 100\(^6\) of such datasets by uniformly sampling from the first 1000 search results using the following three criteria: (a) files end with “.log” (b) with length greater than 20000 (c) contains one of the following keywords: ”db,” “2016,” “system,” “query,” “user.” The datasets are sampled using computer-generated random numbers and chosen before any follow-up analysis is conducted, so it represents an unbiased subset of the whole dataset. The characteristics of these datasets are discussed in Section 5.3.1, and the experimental results are discussed in Section 5.3.2. The 100 sampled datasets constitute a new benchmark for structure extraction from log datasets, which will be released to public if this paper is accepted.

Evaluation Goal. The goal of the experiments in this section is to demonstrate the effectiveness of DATAMARAN on common log datasets “in the wild.” In Section 5.3.1, we study the characteristics of the log datasets in our sampled collection. In Section 5.3.2, we evaluate the extraction accuracy of DATAMARAN and compare with RecordBreaker [2].

5.3.1 Dataset Characteristics. The sampled datasets are categorized based on three criteria:

- whether the dataset contains multi-line records
- whether the dataset consists of multiple types of records
- whether the dataset has any structure at all

There are five possible labels of datasets based on the above criteria, which are listed in Table 4. The distribution of labels among the 100 sampled log files is shown in Figure 16a.

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S (Single-line)</td>
<td>Dataset consists of only single-line records.</td>
</tr>
<tr>
<td>M (Multi-line)</td>
<td>Dataset contains records spanning multiple lines.</td>
</tr>
<tr>
<td>NI (Non-Interleaved)</td>
<td>Dataset consists of only one type of records.</td>
</tr>
<tr>
<td>I (Interleaved)</td>
<td>Dataset contains more than one types of records.</td>
</tr>
<tr>
<td>NS (No Structure)</td>
<td>Dataset has no structure or its structure does not follow assumptions in Section 3.</td>
</tr>
</tbody>
</table>

Table 4: GitHub Dataset Labels

![Figure 16: GitHub Datasets: Characteristics and Accuracy](image)

In the following, we discuss several findings from Figure 16a:

- Validity of Structural Assumptions: 89% of datasets follow assumptions in Section 3, and 10% of the datasets has no structure at all (nothing can be extracted from these datasets), only 1% dataset have structure that cannot be described within the framework in Section 3. These statistics suggest that assumptions in Section 3 are well-justified for log datasets.
• **Necessity for Multi-line Record Handling:** 31% of datasets contains at least one type of record spanning multiple lines. The optimal structure in these datasets cannot be successfully extracted if the extraction system cannot handle multi-line records.

• **Necessity for Interleaved Records Handling:** 32% of datasets contains more than one type of records. If the extraction system cannot recognize the existence of multiple types of records, only one type of record can be extracted (the rest will be regarded as noise), resulting in information loss.

5.3.2 **Structure Extraction Accuracy.** We applied Datamaran to extract structured information from GitHub datasets. Figure 16b shows extraction accuracy for different types of datasets (based on the standard in Section 5.1). Overall, Datamaran successfully extracted structured information from 85 datasets. The accuracy is 95.5% if we exclude datasets with no structure.

As we can see in Figure 16b, Datamaran achieved 100% accuracy on single-line non-interleaved datasets, the simplest type of dataset. The accuracy of Datamaran for the other three types of datasets are 85.7%, 92.3% and 94.4% for exhaustive search, and 76.6%, 76.9%, 83.3% for greedy search. Therefore, we conclude that Datamaran is effective for most of the log datasets in practice. We also identified major causes for inaccurate extractions, which can be found in Section 8.3 in the appendix.

Figure 16b also shows the extraction accuracy of RecordBreaker [2] with default configurations and parameters for comparison. As we can see, RecordBreaker performs very poorly on log datasets with accuracy 56.8% and 7.1% on S(NI) and S(I) respectively and 0% on M(NI) and M(I), for a total of 29.2% accuracy, which is not very surprising: RecordBreaker is originally designed for well-structured datasets, and cannot handle the noise-heavy log datasets very well. Furthermore, the resulting structure templates depend a lot on the Flex configurations and the tuning of two parameters in RecordBreaker (i.e., MaxMass and MinCoverage). This is because Flex configurations decide the quality of tokenization, while the other two parameters determine the datatype (i.e., struct, array or union) for a given list of records. However, there are no generic configurations or parameter values that work for all datasets, which makes RecordBreaker less desirable in an unsupervised setting and incomparable to Datamaran.

Figure 16a and Figure 16b also demonstrates why prior work such as Recordbreaker [2] is not well-suited for extracting structure from log datasets: for any dataset containing multi-line records, the task of partitioning such dataset into collection of records is nontrivial (due to the presence of noise & the fact that record span is unknown). From Figure 16a, we see that at least 31%7 of datasets cannot be handled by RecordBreaker [2] as demonstrated by M(NI) and M(I) in Figure 16b.

6 **RELATED WORK.**

Our work is related to the vast bodies of work on general information extraction, as well as the more limited work on log dataset extraction, and string transformation. Sarawagi [39] provides an excellent summary of the information extraction area.

---

7This number is an underestimate since Assumption 1 can also be violated in some datasets

Example-Driven HTML Wrapper Induction. There has been a long line of work on inducing or learning a “wrapper” to extract content from HTML pages, e.g., [16, 25, 26, 29, 34–36]. The majority of these papers crucially rely on both the web-page structure in the form of the DOM, as well as on text (e.g., extract the piece of text immediately following “Price”). Examples are provided in the form of entities that belong to the concept class that are to be extracted, or in the form of explicit annotations (e.g., this location contains an item of interest to be extracted). Often, the eventual relational schema is known in advance. Some papers do not rely on the HTML structure, opting instead to use NLP [5, 14]. In our case, we do not require any seed entities or annotations.

Unsupervised HTML Wrapper Induction. A few papers attempt to extract from HTML pages directly, without requiring any training examples [6, 43, 44, 47]. All of these papers rely on repetitiveness within a page, or the redundancy across similar pages to separate the content from the template. The rules that are inferred are strongly dependent on the HTML DOM tree structure; in our case, we do not have the luxury of HTML tags to distinguish between records or fields, relying only on text-formatting instead.

Other Forms of Extraction. There has been some work on extraction from other forms of documents, or portions of Web documents. For example, Senellart et al. [41] extract entities from the hidden web, using a few seed example concepts, while some papers [4, 12, 31, 52] extract entities and attributes from text files (such as recipes [12]) using a knowledge base.

List extraction, i.e., extraction from lists on the web is another area that has seen some work [11, 13, 22, 33, 50]. Some of these papers require both the eventual relational schema as well as candidate examples to be provided [22, 33]. Some papers attempt fully-automated list extraction [11, 13, 50]. These papers make the crucial assumption of each record corresponding to a single list item, making it easy to extract the boundaries of the records. Our space of datasets—log files—do not admit any such assumption.

Log Dataset Extraction and Transformation. Wrangler [21, 28] supports the interactive specification of log dataset cleaning operations, drawing from the transformation operations described in Potter’s Wheel [37]. Instead of operator specification, other work relies on user-provided input-output examples [7, 19, 20, 27, 30] to transform one semi-structured dataset to another. In our case, we do not require any intervention from the user.

Finally, the PADs project [15] relies on user-provided chunker and tokenizer to identify the boundaries of records and field values. The only unsupervised log dataset extraction system that we have seen is Recordbreaker [2]: as we have seen it can only handle single-line records, and its fixed lexer configuration might not be flexible enough for all kinds of log datasets in practice.

7 **CONCLUSIONS.**

We presented Datamaran, a completely unsupervised automatic structure extraction tool specifically tailored towards log datasets. We formally defined the structure extraction problem as an optimization problem, where we are given a regularity score function that reflects human judgment, and we search for the best structure template that optimizes this regularity score function. Datamaran algorithm consists of three major steps: the generation step
searches for structure templates satisfying the minimum coverage threshold assumption; the pruning step orders them using an assimilation score function; the evaluation step evaluates the structure templates remaining after the pruning step, and further refines them to achieve even better score; We experimentally evaluated DATAMARAN on a collection of representative datasets and a large collection of log files crawled from GitHub. The experimental results demonstrate that DATAMARAN can efficiently and correctly extract structures from all representative datasets and 95.5% of the GitHub datasets, and is robust with respect to parameter choices, while RecordBreaker can only extract 29.2% from the same dataset collection.

REFERENCES
would only enumerate 128. Thus, the exhaustive search would enumerate the necessary subsets. On the other hand, the greedy search procedure is guaranteed to find the correct set and gradually adds new characters into it: RT-CharSet-

Candidate \( c \) that appeared in the dataset. The exhaustive search would enumerate all \( 2^c \) subsets. On the other hand, the greedy search procedure would only enumerate \( O(c^2) \) of them. The greedy search procedure operates in the following way: initially, RT-CharSet is set to be empty; then in each step, one of the characters in RT-CharSet-Candidate is added to RT-CharSet; the decision for choosing which character to add is made greedily by choosing the character generating the structure template with highest approximation score.

The following example helps illustrate the two searching procedures. Consider a dataset with the following structure template:

\[
[F : F : F] F (F, F)
\]

There are 7 special characters in total: \([, \): \), space character. Thus, the exhaustive search would enumerate 128 possible subsets for this example. As for the greedy search, it starts from the empty set and gradually adds new characters into it:

1. in the first step, it enumerates all the subsets containing only one character, and computes the corresponding structure templates (i.e., invoking steps 2-5).
2. it then decides which subset to proceed based on which one has the structure template with the highest approximation score (for this example, it is \( "F : F : F" \)).
3. then in the second step, it enumerates all 6 subsets consisting of the character ‘:\’ and one additional character.
4. this procedure repeats until either the subset is full or we can no longer find any structure template with at least \( \alpha \% \) coverage.

It is easy to see that, for this example, the maximum number of subsets that the greedy search would have enumerated is 29 (also counting the empty subset here). On the other hand, the exhaustive search would have enumerated 128 subsets. Note that if the field values do not contain any special characters in RT-CharSet-Candidate, then the correct RT-CharSet would contain all characters in RT-CharSet-Candidate that appeared in the dataset. In this case, the greedy search procedure is guaranteed to find the correct RT-CharSet since it will always consider the full subset at the end of the searching procedure.

Second step: We only enumerate pairs of end-of-line characters \( \backslash n \) that are close to each other: the number of lines between them is at most \( L \). For example, when \( L = 1 \), we will only enumerate contents between the \( i \)th and \( (i + 1) \)th \( \backslash n \) character. In this way, we can ensure that the number of pairs that need to be enumerated is linear to the length of the dataset inspected in the generation step.

Third step: The non-overlapping assumption (Assumption 2) states that there exists two disjoint sets of characters \( A \) and \( B \), such that for any instantiated record \( R \), \( RT CharSet(R) \) (i.e., the record template character set) is a subset of \( A \), and \( F CharSet(R) \) (i.e., the field value character set) is a subset of \( B \). By this assumption, the record template can be uniquely extracted from any of its instantiated records given the value of \( A \) and \( B \).

For example, if \( A = \{ \, , \, , \, \, \, \backslash n \} \) then the instantiated record \( 1, 2, 3, 45, 6, 78, 9, a, b, c, d\backslash n \) can be transformed into the record template \( F, F, F, F, F, F, F, F, F, F, F, F, F, F\backslash n \) by replacing characters not in \( A \) with the field placeholder.

Fourth step: We identify the corresponding minimum structure template that can generate each extracted record template. This is achieved by repeatedly reducing repeated patterns into array regular expressions. For example, the record template \( F, F, F, F, F, F, F, F, F, F, F, F, F, F\backslash n \) is reduced into the structure template \( \{ F, F \} \times F\backslash n \). If there are conflicting reduction steps (i.e., reduction steps that cannot be performed simultaneously), we choose one of them arbitrarily. The reduction process only guarantees to find a minimal structure template (i.e., structure template that cannot be reduced further), which means that not all instantiated records are correctly reduced back to the corresponding structure template. As a result, the coverage estimate during the generation step is an underestimate. However, in our experiments, the initial coverage estimate is usually not far away from the correct value, and therefore is still well above the \( \alpha \% \) threshold.

Fifth step: We store all of the structure templates in a hash-table, and maintain the total coverage of all structure templates associated with each hash-bin. For all hash-bins with less than \( \alpha \% \) total coverage, the associated structure templates are discarded.

The pseudocode of the generation step can be found in Algorithm 1. Two searching procedures correspond to function GreedySearch and ExhaustiveSearch respectively. The function GenST finds structure templates with at least \( \alpha \% \) coverage given the value of RT-CharSet.

8.1.2 Handling Multiple Structure Templates. In the cases where there are more than one type of record in the dataset, we repeat the entire structure detection process (Generation-Pruning-Evaluation) for multiple times. After each iteration, we retrieve the parts of the dataset that are not explained by the previous structure. These parts are concatenated together, and we run the entire procedure on it again.

8.1.3 Sampling Technique. In the actual implementation of DATAMARAN, sampling is used instead of simply scanning through the entire dataset in both the generation and evaluation step. For large datasets, scanning the whole dataset during these steps could be
Algorithm 1 The Generation Step

function GenST(char_set)

    n ← Total Number of Lines
    for i ← 1 to n do
        for j ← 1 to i + 1 do
            left_boundary ← i
            right_boundary ← j
            r ← ExtractRecord(left_boundary, right_boundary)
            rt ← ExtractRecordTemplate(r, char_set)
            st ← GenerateStructureTemplate(rt)
            k ← ComputeHashKey(st)
            cost(k) ← cost(k) + length(r)
            st_set(k) ← st_set(k) ∪ {r}
        end for
    end for

    ST_set ← {}
    for char_set ⊆ char_candidates do
        ST_set ← ST_set ∪ GenST(char_set)
    end for

    return ST_set

end function

8.1.4 Default Regularity Score Function. We implemented a simple default regularity score function based on the minimum description length principle [8]: we design a record generation procedure from the structure template, and compute the total amount of additional information needed to describe the instantiated records using the structure template. The final score is the total amount of information needed for describing all the instantiated records, plus the additional information needed to describe the noise blocks. For completeness, we describe the details of this score function in the following.

Describing the record using the structure template is straightforward given the structural form assumption (Assumption 3):

- For arrays, we describe the number of repetitions, then describe each repetition individually.

- For structs, we describe each component individually.

- For fields, the description scheme depends on its value type.

For the field value description, we utilize the group partitioning of field values in Section 3.3, and associate each group with one of the following four value-types: enumerated type, integer, real number, or string. The description schemes for field values depend on the data-type—which can be determined by analyzing the field values in the group; the details of these schemes are listed as follows:

- The enumerated type fields are described using \([\log_2(n_{value})]\) bits, where \(n_{value}\) is the total number of unique values in this field.

- The integer fields are described using \([\log_2(\text{max}_{value} - \text{min}_{value} + 1)]\) bits, where \(\text{max}_{value}\) and \(\text{min}_{value}\) are the upper bound and lower bound of the field value, which can be determined by scanning through the dataset.

- The real number fields are described using \([\log_2(\text{exp}_{value} \times \text{min}_{value})]\) bits, where \(\text{max}_{value}\) and \(\text{min}_{value}\) are the same as above, and \(\text{exp}\) is the maximum number of digits after the decimal point.

- The string fields are described directly using \((\text{len}(s) + 1) \times 8\) bits, where \(\text{len}(s)\) is the length of the field value. The +1 term is to include the end-of-string `\0` character, and each character needs 8 bits to describe.

Using the description schemes above, the total description length can be computed as follows:

\[
D(\text{dataset}) = \text{len}(\text{ST}) \times 8 + 32 + m + \sum_{i=1}^{m} D(\text{block}_i) \text{ bits}
\]

The first \(\text{len}(\text{ST})\times 8\) bits describe the the structure template, and the next \(32 + m\) bits describe the total number of blocks in the dataset and whether each block is a noise block or a record. \(D(\text{block}_i)\) is the description length of ith block: for noise blocks, it is simply the block length times 8; for record blocks, we compute its description length accordingly.

The pseudocode for computing the description score can be found in Algorithm 2, with the following 3 steps:

1. (1) extract all the instantiated records from the dataset.
2. (2) estimate the data-type parameters from the extracted records.
3. (3) compute the description length using the formulae above.

Algorithm 2 The Evaluation Step

function EvalST(ST)

    (RecordBlocks, NoiseBlocks) ← ParseData(ST)
    Determine the data types of field values
    Learn the distributional parameters

    TotalDL ← \(\text{len}(\text{ST})\times 8 + 32 + \text{NumBlocks}\)

    for record ∈ RecordBlocks do
        RT ← GetRecordTemplate(record)
        TotalDL ← TotalDL + D(RT)
    end for

    for block ∈ NoiseBlocks do
        TotalDL ← TotalDL + \(\text{len}(\text{block})\times 8\)
    end for

    return TotalDL

end function
8.2 Formal Evaluation Standard

In order to rigorously define the evaluation standard, we first define the concept of transformation script:

Definition 8.1 (Transformation Script). A transformation script of structure template ST is a mapping function from instantiated records of ST to strings. Depending on the type of ST (defined in Assumption 2), the transformation script $T_{ST}$ must have one of the following formats:

- If the root node of ST is struct-type (see Assumption 3), then $T_{ST}$ must have the following format:
  \[ T_{ST}(R) = T_{C_1}(R_1) + T_{C_2}(R_2) + \ldots + T_{C_k}(R_k) \]
  where $C_1, \ldots, C_k$ are the children nodes of ST, and $T_{C_1}, \ldots, T_{C_k}$ are transformation scripts of them. $R_1, \ldots, R_k$ are the parts of R corresponding to $C_1, \ldots, C_k$ respectively.

- If the root node of ST is array-type (see Assumption 3), then $T_{ST}$ must have the following format:
  \[ T_{ST}(R) = U(T_{C_1}(R_1) + T_{C_2}(R_2) + \ldots + T_{C_k}(R_k)) \]
  where $C$ is the child node of ST corresponding to the repetitive pattern, and $R_1, \ldots, R_k$ are the instantiated repetitions of $C$ in $. T_{C}$ is a transformation script of $C$, and $U(str)$ is a simple function defined as follows:
  \[ U(str) = s_1 + SubStr(str, a, len(str) - b) + s_2 \]
  where $a, b, s_1, s_2$ are constants that depends only on ST (they are fixed for all instantiated records R of ST). Note that we allow the extreme case where $a = b = \infty$ such that $U(str) = s_1 + s_2$ regardless of the value of $str$.

- If the root node is a simple string consisting of $k$ field values, then $T_{ST}$ must have the following format:
  \[ T_{ST}(R) = U_1(F_1) + U_2(F_2) + \ldots + U_k(F_k) \]
  where $F_1, \ldots, F_k$ are the field values of R, and $U_1, \ldots, U_k$ are simple functions defined as follows:
  \[ U_i(str) = s_{1,i} + SubStr(str, a_i, len(str) - b_i) + s_{2,i} \]
  where $a_i, b_i, s_{1,i}, s_{2,i}$ are constants that depends only on ST (they are fixed for all instantiated records R of ST).

Intuitively, the transformation script represents an “aggregation” of all the information from leaf nodes of the tree. To understand this, we can assume that messages are being passed from leaf nodes to the root along the tree path: each message contains one particular field value, and originates from the corresponding leaf node of the tree. At each tree node, we will perform the following actions in sequence: (a) gather messages received from its children nodes; (b) concatenate all the messages; (c) make changes to the result by adding/dropping prefix/suffix; and (d) send the resulting string to its parent node. It is easy to see that the above message-passing formulation is essentially equivalent to Definition 8.1.

Given the concept of transformation script, we can formally define the following rigorous standard for accuracy evaluation:

- For each dataset, we first identify all the observable structure templates (i.e., following Assumption 3) within the dataset.
- Within each identified structure template, we identify as many “key” fields as possible. Intuitively, “key” fields are the field values that contain potentially interesting information.
- For each “key” field $K$, we consider it successfully extracted if there exists a transformation script $T_{ST,K}$ of the corresponding structure template ST, such that the correct field value can be reconstructed for all instantiated records of ST by applying $T_{ST,K}$ to the corresponding extracted structure.
- The structure extraction for each dataset is considered successful if all of the “key” field values are successfully extracted.

8.3 Causes for Inaccurate Extraction

Here we describe the causes for inaccurate extraction for GitHub log datasets (Section 5.3). There are 4 log files where even the exhaustive search version of DATAMARAN failed to find a valid structure. In the following, we list the two causes for these inaccurate extractions, and discuss the potential ways to address them.

Fail to recognize “long” records: The maximum range of records is set to be 10 lines during the experiments. In some datasets, there are some extremely “long” records that exceeds this limit. If we increase the range limit, the efficiency of DATAMARAN would suffer. As the records in practice can be arbitrarily long, we are still unsure of methods that can completely solve this problem.

The greedy approach for interleaved datasets: In DATAMARAN, we handle interleaved datasets by repeatedly applying the algorithm on the dataset. However, this greedy procedure does not always find the correct structure for interleaved dataset. Instead, sometimes we would find structure templates with characteristics of multiple types of records. The following example illustrates this phenomenon.

Suppose we have two types of records with templates:

\[ F: (F)^*F \]

\[ F: F F F \]

\[ F: F F F F F F \]

When this generic structure template has a lower regularity score compared to the two correct record templates.

8.4 Sources and Characteristics of Manually Collected Datasets

Table 5 lists the sources and characteristics of the 25 manually collected datasets. The first 15 datasets are from Fisher et al.’s paper [15] (marked with “*” in Table 5).

8.5 Proof of Theorem 4.1

Proof. First of all, condition (b) ensures that $ST_0$ can be found during the generation step. Then, using condition (a), we can ensure that $ST_0$ to be the top structure template during the pruning step. Finally, condition (c) ensures that $ST_0$ will be chosen during the evaluation step. Combining all arguments, we can see that DATAMARAN is guaranteed to return $ST_0$ as the optimal structure template.
<table>
<thead>
<tr>
<th>Data source</th>
<th>File size(MB)</th>
<th># of rec. types</th>
<th>Max rec. span</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction records</td>
<td>0.07</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>comma-sep records</td>
<td>0.02</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>log file of Mac ASL</td>
<td>0.28</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Mac OS boot log</td>
<td>0.02</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>crash log</td>
<td>0.05</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>crash log (modified in [15])</td>
<td>0.05</td>
<td>1</td>
<td>(3)</td>
</tr>
<tr>
<td>netstat output</td>
<td>0.01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>personal income records</td>
<td>0.01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>US railroad info</td>
<td>0.01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>application log</td>
<td>0.06</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LoginWindow server log</td>
<td>0.05</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>pkg install log</td>
<td>0.02</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Thailand district info</td>
<td>0.19</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>stackexchange xml data</td>
<td>20</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>vcf genetic format</td>
<td>167.4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fastq genetic format</td>
<td>29.9</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>blog xml data</td>
<td>0.06</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>log file (1)</td>
<td>0.03</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>log file (2)</td>
<td>0.01</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>log file (3)</td>
<td>0.19</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>log file (4)</td>
<td>0.07</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>log file (5)</td>
<td>0.09</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5: Sources and characteristics of manually collected datasets.