On the Naturalness and Localness of Software Logs

Sina Gholamian
University of Waterloo
Waterloo, ON, Canada
sgholamian@uwaterloo.ca

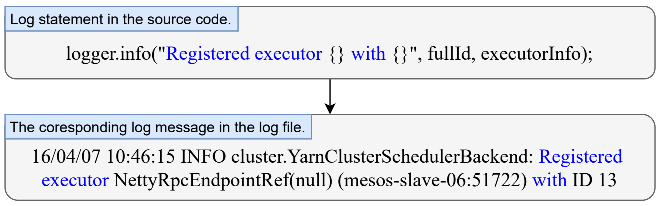
Paul A. S. Ward
University of Waterloo
Waterloo, ON, Canada
pasward@uwaterloo.ca

***Abstract*—Logs are an essential part of the development and maintenance of large and complex software systems as they contain rich information pertaining to the dynamic content and state of the system. As such, developers and practitioners rely heavily on the logs to monitor their systems. In parallel, the increasing volume and scale of the logs, due to the growing complexity of modern software systems, renders the traditional way of manual log inspection insurmountable. Consequently, to handle large volumes of logs efficiently and effectively, various prior research aims to automate the analysis of log files. Thus, in this paper, we begin with the hypothesis that log files are natural and local and these attributes can be applied for automating log analysis tasks. We guide our research with six research questions with regards to the naturalness and localness of the log files, and present a case study on anomaly detection and introduce a tool for anomaly detection, called *ANALOG*, to demonstrate how our new findings facilitate the automated analysis of logs.**

***Index Terms*—software systems, logging statements, log files, entropy, natural language processing, naturalness, localness, natural language processing (NLP), anomaly detection**

I. Introduction

Logging is an everyday programming practice and of great importance in modern software development, as software logs are widely used in various software maintenance tasks. Based on its granularity, logging allows developers and practitioners to investigate the inner-workings of software systems, and track down problems as they arise. Because of the rich information that resides in logs and the pervasiveness of logging, logs enable a wide range of tasks such as system provisioning, debugging, management, maintenance, and troubleshooting. Examples of prior research threads associated with logs include analyzing user statistics [46], identifying performance anomalies [24], [53], diagnosing system errors and crashes [66], [68], and ensuring application security [58]. Fig. 1 illustrates an example of a logging statement and its end product written to a log file[1].


Log statement in the source code.
The coresponding log message in the log file.

```
logger.info("Registered executor {} with {}", fullId, executorInfo);
```

```
16/04/07 10:46:15 INFO cluster.YarnClusterSchedulerBackend: Registered
executor NettyRpcEndpointRef(null) (mesos-slave-06:51722) with ID 13
```

Fig. 1: A log example from an Apache Spark application.

A Log statement is commonly supplemented with a verbosity level (*e.g.*, error/debug/info), a constant part (*e.g.*, *"Registered executor with"* in Fig. 1; also called *'log statement description'*), and a variable part (*e.g.*, *fullId* and *executorInfo* in Fig. 1). Additionally, logging libraries and wrappers such as Log4j [10] accompany the log statement with extra information as it is written to the log file, such as its timestamp and the component generating the log. Due to the free-form text format of the log messages written by developers, it is often an involved task to extract meaning from the log messages. Furthermore, the dynamic nature of variables, which may yield a different output in each iteration of the program, brings additional irregularity and adds to the complexity. In a parallel angle, due to the sheer volume of logs that modern software systems routinely produce, in the scale of tens of Gigabytes of data per hour for a commercial cloud application [38], [52], [76], it is unfeasible to inspect log messages for crucial diagnostic information with traditional methods such as manual checks or searches with search and `grep` scripts. As such, although tremendous system diagnosis and maintenance advantage is beclouded in the logs, how to effectively and efficiently analyze the logs remains a great challenge, and subsequently, automatic log analysis tools and approaches are highly sought after [76].

To fill in this gap and pave the way towards the goal of automated log analysis, prior research has proposed multiple avenues of work to automate analysis, which relies on finding patterns in logs. The analysis usually starts with log parsing [76], which aims to extract structured templates from unstructured log data. Following this step, prior research has applied a wide variety of pattern mining and machine learning approaches, to name a few, PCA-based dimension reduction [56], execution path and variable value inference [68], learning model [53], event correlation graphs [30], and temporal correlation mining [26].

Despite the presence of prior log analysis approaches, we believe prior research has not fully utilized the natural language attributes of software systems [41], [64] for log file analysis. Therefore, in this paper, we focus on the natural language processing (NLP) characteristics of log files, and our main hypothesis is:

Logs, as an artifact of software systems, similar to programming languages and source code, are natural and locally repetitive and predictable. Thus, natural language models can capture these attributes and leverage them for automated analysis of log files.

[1]We use *logs* and *log files* interchangeability.

Prior research such as [41] and [64] have shown that software's source code, similar to natural language ($\mathcal{NL}$) corpus, is repetitive, predictive, and local. As such, statistical language models yield promising results once applied in software engineering tasks. Authors in [41] and later in [64] showed that n-gram language models perform well in the source code's modeling and leveraged this fact to propose a cache language model for code suggestion. Most recently, He *et al.* [35] showed that logging descriptions within the source code also follow the natural language repetitiveness observed in the software systems. In this paper, inspired by the prior work, we investigate if log files possess NLP characteristics and how to leverage this feature for automated analysis of logs.

There are several findings and implications stemming from our study. We observed that log files show a high degree of repetitiveness and regularity (Finding 1), and the regularity is project-dependent (Finding 2). Zipf's law shows the high-rank tokens happen more often in the logs than in English text (Finding 3). Findings 4 and 5 shed light on the log localness and confirm that logs are *endemic* and *specific*. At last, our NLP-based anomaly detection approach achieves high F-Measure and Balance accuracy scores, which illustrates an application of NLP in log analysis (Finding 6). In summary, our paper makes the following contributions:

- We conduct the first empirical study on the utilization of natural language for log files by evaluating eight system logs and two English corpora. We have provided our dataset to encourage and facilitate further research [4].
- We demonstrate the naturalness and localness of logs through several research questions (RQs) by utilizing n-gram models and self- and cross-project entropy calculations.
- We introduce *ANALOG*, an NLP-based log file anomaly detector, to illustrate the potential benefits of NLP in log analysis.

We organize the rest of this paper as follows. Section II reviews the background and motivation. Section III presents our research questions (RQs), and we quantitatively analyze the naturalness and localness of logs in Sections IV and V. Section VI demonstrates the use of language models (LMs) in a case study for anomaly detection. Related work and threats to validity are in Sections VII and VIII. Finally, we conclude the paper with some avenues for future work in Section IX.

II. Background and Motivation

Natural language processing (NLP) models (*e.g.*, the n-gram model [5]) are statistical models that *estimate* and *evaluate* the probability of a sequence of words or tokens. During *estimation*, the model assigns a probability to sequences of words (or tokens) with *maximum likelihood estimation* (MLE). During *evaluation*, the model predicts the probability of whether the sequence under test belongs to the training corpus. The predictable and repetitive characteristics of common English corpora, which statistical NLP techniques extract and model, have been the driving force of various successful tasks, such as speech recognition [16] and machine translation [51]. Subsequently, software engineering researchers [14], [41], [64] have shown that software systems are even more predictable and repetitive than common English text, and language models perform better on software engineering tasks than English text. As such, tasks such as code completion [63] and code suggestion [18] utilize n-gram models for their predictions. Recently, He *et al.* [35] showed that log statements' descriptions (LSDs) within the source code (Fig. 1) also follow natural language characteristics. Because LSDs in the source code are considered in isolation and they do not completely determine what is in the log files, which is an arbitrary sequence (*i.e.*, not isolated) of log prints with LSDs and, additionally, the dynamic runtime value of the variables included by virtue of the execution of log statements, the naturalness of LSDs in the source code does not guarantee the naturalness of log files. Thus, our goal is to validate the NLP attributes of logs and foster research and employment of NLP methods for automated log analysis.

N-gram language models. Formally, considering a sequence of tokens in the corpus under consideration (in our case, log files), $S = a_1, a_2, ..., a_N$, the n-gram model statistically estimates how likely a token is to follow preceding tokens. Thus, the probability of the sequence is estimated based on the product of a series of conditional probabilities [41]:

$$P_\theta(S) = P_\theta(a_1)P_\theta(a_2|a_1)P_\theta(a_3|a_1a_2)....P_\theta(a_N|a_1...a_{N-1}) \quad (1)$$

which is equal to:

$$P_\theta(S) = P_\theta(a_1).\prod_{i=2}^{N} P_\theta(a_i|a_{i-1}, a_{i-2}, ..., a_1) \quad (2)$$

where a_1 to a_N are tokens of the sequence S and the distribution of θ is estimated from the training set with MLE[2]. The metric to assess the performance of an n-gram model $\mathcal{M}$ is *perplexity* (PP), which is the inverse probability of the test sequence:

$$PP_\mathcal{M}(S) = \sqrt[N]{\frac{1}{P(a_1).\prod_{i=2}^{N} P(a_i|a_{i-1}, a_{i-2}, ..., a_1)}} \quad (3)$$

For model $\mathcal{M}$'s performance evaluation, perplexity and its log-transformed version, *cross-entropy*[3], $H_\mathcal{M}$ [3] are often used interchangeably: $H_\mathcal{M}(S) = \log_2 PP_\mathcal{M}(S)$. Equation 3 explains that the higher the probability of a sequence, the lower the PP value will be. In other words, the lower the *perplexity* (and likewise *cross-entropy*), the less surprising the new token sequence is for the model $\mathcal{M}$, and hence the higher probability that the token sequence under investigation belongs to the same corpus. Thus, an appropriately trained n-gram model will predict with low probabilities (and high *perplexity* values) if it deems that the content of a test sequence does not belong to the corpus used for training.

Motivation. Encouraged by the prior work in the utilization of NLP for software engineering systems, in this work, we investigate the natural language characteristics of log files. More specifically, we aim to answer the question

[2]For simplicity, we drop the θ in the notation onward.

[3]Often, simply called *entropy*.

of *"are log files natural and local?"*. Intuitively, if there are discernible repetitiveness and predictability in logs, a suitably trained language model should yield acceptable performance in distinguishing the log messages belonging to a specific system's log from the ones which look unfamiliar. We speculate this feature will benefit automated analysis of logs, and we aim to use it for software engineering tasks such as anomaly detection since anomalous log messages generally look different from normal logs. For this purpose, we exploit the n-gram language models, measure the perplexity and entropy values for different logs, and analyze our findings from the RQs.

III. Natural Language Processing for Logs

To investigate whether log files are natural and local, we analyze both natural English corpora and a collection of logs from various systems available online and applied by prior research [38], [67]. We guide our research with the following research questions (RQs) for the **naturalness** of logs:

- ***RQ1:*** *does a natural repetitiveness and regularity exist in log files?*
- ***RQ2:*** *is the regularity that the statistical language models capture merely log-nature specific, or is it also project-specific?*
- ***RQ3:*** *how does Zipf's law capture the repetitiveness of high-rank tokens in log files?*

Next, for **localness** of logs, we investigate:

- ***RQ4:*** *are log n-grams endemic to their projects*[4]*?*
- ***RQ5:*** *are log n-grams specific to their projects?*

Finally, we provide a case study of the applications of NLP attributes of logs in the final RQ:

- ***RQ6:*** *how the logs' NLP characteristics*[5] *can help with automated analysis of log files?*

Clarifying the above research questions with quantitative analysis for logs and English corpora enables us to find supporting evidence on our hypothesis on the naturalness and localness of logs, which we later use for anomaly detection. We continue with the description of the dataset that we used in our analyses.

Data description. We utilize the data publicly available by prior research [38], [67] as the base for our analyses. In summary, we select eight log files from a wide range of computing systems and two English corpora. We briefly review each of the analyzed logs in the following. ❶ **HDFS.** The HDFS [11] log is generated from a Hadoop cluster. HDFS is a distributed and resilient-to-failure file system for commodity servers, which brings in reliability. Prior research [67], [76] has used this data set for log parsing and compression. ❷ **Spark.** Apache Spark [9] is a popular and efficient big-data processing framework. This log data is aggregated from event logs of a spark cluster of 32 machines. Prior research has utilized this data for log message pattern extraction [37], [76]. ❸ **Firewall.** Firewall log is the collection of firewall process activities during the network operation of operating systems, such as Windows's firewall process. Prior work has used this type of data for frequent pattern extraction and compression [34], [67]. ❹ **Windows.** The Windows log belongs to Windows 7's component-based servicing (CBS) initially stored at *C:/Windows/Logs/CBS*, which collects logs on package and driver installations and updates. It is used for log parsing and anomaly detection in prior studies [38], [76]. ❺ **Linux Syslog.** The Linux Syslog is the standard logging system in Linux which collects logs from various concurrently running applications on a single machine, such as networking and *cron* logs, and Linux kernel logs. Syslog is used to analyze compression on log data in prior research [67]. ❻ **Thunderbird.** The Thunderbird dataset is collected from a Linux supercomputer cluster from Sandia National Labs (SNL), which is the aggregation of Syslogs from different machines. This dataset is used in prior work for log compression and log analysis [55], [67]. ❼ **Liberty.** The Liberty log is an aggregated dataset of Syslog-based events on a supercomputer system, which was examined for log analysis and alert detection in previous studies [55], [57]. ❽ **Spirit.** Similar to Thunderbird and Liberty, Spirit is also a supercomputer log data from the Spirit supercomputer system, used for gaining insight on failure diagnosis of large-scale systems [55].

Besides the log data, we also analyze two natural language corpora: ❶ **Gutenberg.** Project Gutenberg corpus [6] is a collection of over 60,000 English books [7]. The Gutenberg corpus has been studied to examine the performance of natural language tools [19], and NLP evaluation of software systems [35], [41]. ❷ **Wiki.** The Wikipedia corpus is the data collected from English articles of Wikipedia. This data has been previously applied to evaluate the performance of different text compressors [32].

Logs	**Category**	**Lines**	**Tokens**	
			Total	Unique
HDFS (HS)	Distributed system	7527525	259293940	875434
Spark (SP)	Distributed system	13221789	308093706	673725
Firewall (FW)	Operating system task	4763441	418012413	2655410
Windows (WD)	Operating system	4408109	338718867	37023
LinuxSyslog (LS)	Operating system	5315580	348036971	454755
Thunderbird (TB)	Supercomputer	6018428	357613165	102699
Liberty (LB)	Supercomputer	7316856	352384327	216946
Spirit (ST)	Supercomputer	7983346	366526309	320426
English corpora	**Description**	**Lines**	**Tokens**	
			Total	Unique
Gutenberg	English books	20867266	232266714	370164
Wiki	English articles	5815221	173380056	3963045

TABLE I: System logs and English corpora statistics.

Table I summarizes the system logs and the English corpora that we studied. We categorize logs based on their domains such as *distributed system, operating system (OS), OS task, and supercomputer*. In order to make a fair comparison among different log files and between log files and natural language corpora, all the files are curtailed at the same size of 1 GB. We show the number of lines, calculated using Unix *"wc -l"* on each file, and the number of tokens. For tokens,

[4]We use *project* and *system* interchangeably, as software systems are commonly referred to as projects during their developments, *e.g.*, Apache Hadoop project.

[5]We use *'natural language characteristics or NLP characteristics'* to cover both of *naturalness* and *locallness*.

we have listed the total number of tokens, and the unique tokens count for each file.

Data preprocessing. Prior to the application of NLP models, we require performing some pre-processing steps on the raw log data and English corpora. In order to be case-insensitive, we initially normalize all the letters to lower cases. We then tokenize all the data to extract words and special symbols. For n-gram model training and testing, we use Kenlm [39], [40] library. For different analyses that follow later in this paper, we create n-gram models for different sizes of n in the range of $n \in (1, 10)$ for each data file, *i.e.*, for unigrams (single tokens), bigrams (pairs of adjacent tokens), and so on.

IV. Naturalness of Logs

In this section, we investigate the naturalness of log files guided with a set of RQs which follows.

A. *RQ1: does a natural repetitiveness and regularity exist in log files?*

Prior research has leveraged the natural predictability of n-grams in English corpora in applications, such as speech and handwriting recognition [16], [60], and machine translation [51]. This observation is also noteworthy for artifacts of software systems, such as user documents, repositories, and logs. It is beneficial to explore whether similar repetitiveness and predictability of n-grams exist in log data as it benefits the automated analysis of logs, and it is the focus of our research.

We showed in Equation 2 the probability estimated by the n-gram model $\mathcal{M}$ for sequence $S = a_1, a_2, ..., a_N$. For simplifying the computation, n-gram models often assume a *Markov* property, which states that for a language model of order n, token occurrences are approximated only by the $n-1$ tokens that precede the token under consideration [42]. For example, for a 5-gram model, the probability of a_i appearing after the sequence of $a_1, a_2, ..., a_{i-1}$ is approximated by the probability of the four prior tokens:

$$P(a_i|a_1...a_{i-1}) \cong P(a_i|a_{i-4}a_{i-3}a_{i-2}a_{i-1}) \tag{4}$$

To calculate the probabilities, the NLP model is estimated on a training set using the maximum likelihood-based frequency-counting of token sequences. Therefore, we estimate the probability of a_i, i_{th} element in Sequence S, which follows tokens $a_{i-1}, a_{i-2}, ..., a_{i-n+1}$ and order n model with:

$$p(a_i|a_{i-1}a_{i-2}...a_{i-n+1}) = \frac{count(a_i a_{i-1} a_{i-2}...a_{i-n+1})}{count(a_{i-1}a_{i-2}...a_{i-n+1})} \tag{5}$$

Based on this estimation, the *cross-entropy* (or *entropy*, used interchangeably), $H_{\mathcal{M}}$, evaluated for model $\mathcal{M}$ for a sequence of n tokens is:

$$H_{\mathcal{M}} = -\frac{1}{N}\sum_{n=1}^{N} \log_2 p(a_i|a_{i-1}a_{i-2}...a_{i-n+1}) \tag{6}$$

A subtle detail worth mentioning about the n-gram model is that, in practice, the n-gram model often encounters some unseen sequences during prediction. This results in the probability $p(a_i|a_{i-1}a_{i-2}...a_{i-n+1}) = 0$, and undefined values for $H_{\mathcal{M}}$. Smoothing is a technique that handles such cases and assigns reasonable probabilities to unseen n-grams. In this paper, we use Modified Kneser-Ney Smoothing [43] available with Kenlm [40], which is a standard smoothing technique and gives acceptable results for software corpora [41], [69]. On the other side of the spectrum, a perfect n-gram model correctly predicts all the next tokens, *i.e.*, $p(a_i) = 1$, and therefore, $H_{\mathcal{M}} = 0$. In general, lower entropy values imply that the n-gram model is more effective in predicting the sequence of tokens and capturing the regularity and repetitiveness of the corpus.

Experiment. To evaluate the repetitiveness and regularity in logs, we measure cross-entropy by averaging over 10-fold cross-validation: we randomly select 10 MB from each log file, ~59,000 lines of logs, which falls well within a $\pm 2.5\%$ margin of error and 95% confidence interval [31], [44]. We then organize each of the log files and English corpus to a 90%–10% train-test split at ten random locations, and train the n-gram model for different values of $n \in (1, 10)$ on the 90% train split, and then test it on the remaining 10% split by measuring the average cross-entropy with Formula 6.

Result. In Figure 2, boxplots display the cross-entropy results for system log files, and the blue line shows the average cross-entropy for the two English corpora. Comparing the values of entropy for log data and English corpora provides an intuitive understanding of the repetitiveness and regularity of tokens in log data and common English. The horizontal axis shows the n-gram model trained with different numbers of n and the vertical axis presets the entropy. Both the single line and boxplot manifest similar trends: as the order of n-gram increases, the entropy values decrease, which implies that using larger values of n (*viz.*, more preceding tokens in Formula 4) yields more accurate predictions for both log data and English Corpora. 4- or 5-gram models are the optimal choice for the studied logs considering the trade-off between the entropy and model memory usage, as cross-entropy saturates around $n \in (4, 5)$. The English corpora entropy starts at 10.19 for 1-gram models and trails down to 8.09 for 10-gram models, compared to logs entropies median that falls just below 1.8. Thus, quantitatively, the log data manifests lesser entropies, and as such, less perplexing to predict when compared to English corpora. This observation paves the way to utilize n-gram models for automated analysis of logs. Additionally, our findings are consistent with prior research on the naturalness of software source code [35], [41].

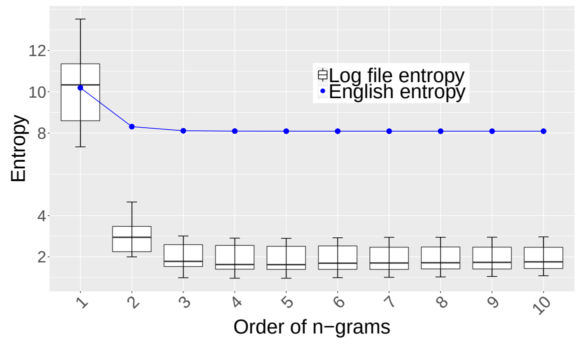


Fig. 2: Entropy values for a sequence of n-grams for log data (boxplot) and English corpora (blue line).

B. *RQ2: is the regularity that the statistical language model captures merely log-nature specific, or is it also project-specific?*

In RQ1, we showed LMs accurately capture the repetitiveness within log files even better than English corpora. However, as we know, software systems, and subsequently, their artifacts, such as log files, have a smaller set of vocabulary when compared to the English language [41]. Thus, there exists a valid concern that the lower entropy values for logs might be the outcome of their limited vocabulary and not their regularity and repetitiveness. If this concern proves to be valid, *viz.*, if the captured regularity from the logs is solely the result of their limited vocabulary, then we should observe similar lower entropies for cross-project evaluation. In other words, if we train the n-gram model on one log file, and then test it on another system's log, we should observe comparable entropies with inner-project entropies. As such, in RQ2, we investigate this concern by training and testing the model on different projects.

Experiment. In this experiment, we measure self- and cross-project entropy values by averaging over 10-fold cross-validation. We randomly sample 10 MB of data from the available 1 GB for each system and English corpora. Without loss of generality [44], we selected this sample size to make the training overhead manageable. Additionally, to confirm the results, we run some limited experiments with various sizes to confirm the results are consistent. Similar to RQ1, we split each of the samples to a 90%–10% split at ten random locations, train the n-gram a 5-gram model on the 90%, and then test it on the remaining 10%, and measured the average self entropy. As observed in RQ1, entropy values stabilize beyond 5-gram models. As such, we use 5-gram models to reach faster training and save on the memory footprint. To calculate the cross-project entropy, once we train the project on one system, we test it on all the other projects and average the values of entropies through 10-fold cross-validation, to minimize the risk of over-fitting [54].

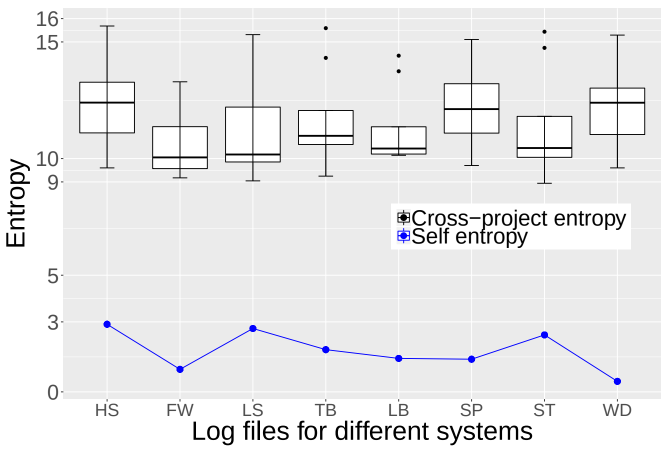


Fig. 3: Entropy values for 5-grams cross-project versus self-project.

Result. Fig. 3 shows the self- and cross-project entropies. The x-axis lists the different logs, and for each log, the boxplot shows the range of cross-project entropies with the other seven projects. The blue line at the bottom shows the average *self entropy* of the project against itself (*i.e.*, training and testing on the same system's log). From this plot, the self-entropy values are always lower, indicating that the repetitive n-gram patterns are not the artifact of limited log vocabulary but because of the regularity and repetitiveness in each system. This regularity is different across different projects, and hence, we observe higher cross-project entropies, implying that the n-gram patterns noticeably disagree across project logs.

C. *RQ3: how does Zipf's law capture the repetitiveness of high-rank tokens in log files?*

Zipf's [61], [77] law is an empirical theorem which states that given a large sample of words in a corpus, the probability of any word is inversely proportional to its rank in the corpus. Thus, the word rank r has a probability proportional to $\frac{1}{r}$. In other words, the rank of the word (r) times its probability $p(r)$ is approximately a constant ($r \times p(r) \simeq const$). For the classic version of Zipf's law we have: $p(r) = \frac{\frac{1}{r}}{\sum_{n=1}^{N}(\frac{1}{N})}$, where N is the population of unique tokens. Considering that Zipf's law is not an exact value but a statistical measurement, it gives an intuitive understanding that the repetition of the high-rank tokens occupies what percentage of the document. For example, in English, the top 50 words accumulate to 35–50% of total word occurrences [70].

Experiment & Result. We measure the token frequency (TF) for logs and English text and then sort the tokens based on their frequencies to study Zipf's law for log files. Figure 4 plots the rank of tokens on the horizontal axis and the frequency counts on the vertical axis for average of Logs (red) and English text (blue) and gives us an idea of the TF distribution in a given document. Logs's TF starts higher than English and drops below as the rank increases. The top-50 tokens contribute to 70% and 51% of the entire document for Logs and English, respectively. This result is encouraging for more accurate adaptation of LMs for logs, as prior research [61] has suggested that smoothing algorithms for n-gram models, such as Good-Turing [25], and Kneser-Ney [25], could lead to better smoothing with more predictive high-rank tokens.

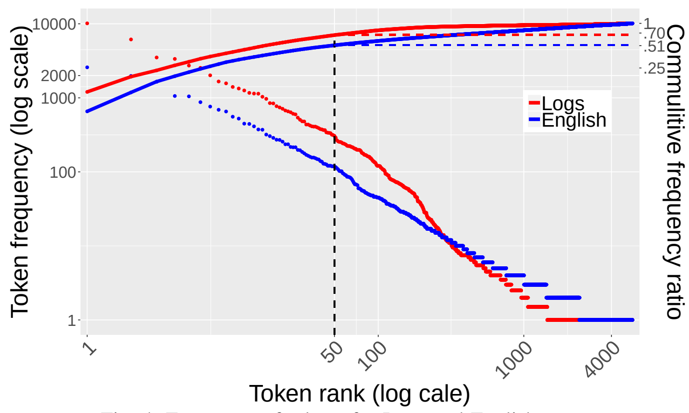


Fig. 4: Frequency of tokens for Logs and English text.

V. Localness of Logs

In this section, we investigate the localness of log files. The localness attribute builds upon the naturalness of logs, such that besides being repetitive and predictable, logs tend to take on a specific form of repetitiveness in the *local context*. Here, a local context is a system's logs versus other systems. For this purpose, we investigate *endemism* and *specificity* of n-grams in different logs.

A. ***RQ4: are log n-grams endemic to their projects?***

The n-grams that only appear in a single project log file (*i.e.*, *local context*) are called endemic, *i.e.*, they are endemic to that specific project's log file. In this RQ, we investigate whether there exist n-grams that are found exclusively in a local context and are endemic to a system. More specifically, we investigate what percentage of n-grams happen only in one system logs. Table II lists the average percentage of the endemic n-grams in the log files and English corpora. For both system logs and English corpora, the first row with (Freq $\geqslant 1$) shows the percentage of endemic n-grams that appear at least once in a system, and the second row with (Freq $\geqslant 2$) represents the percentage of endemic n-grams that appear at least twice in a system logs. For example, 90.24% and 80.74% of 2-grams with (Freq $\geqslant 1$) are endemic for system logs and English corpora, respectively. Similarly, 39.20% and 8.57% of 5-grams with (Freq $\geqslant 2$) are endemic for system logs and English corpora, respectively. The percentage of the endemic n-grams for (Freq $\geqslant 2$) generally increases for higher orders of *n*, because it becomes less likely to observe a large sequence of tokens repeatedly. For (Freq $\geqslant 2$), we observed that for $n \geqslant 3$, because there is a very limited number of n-grams that appear more than once compared to the total number of n-grams in the same order of *n*, the percentage shrinks slightly onwards. By comparison, the percentage of the endemic n-grams in English corpora is noticeably lower than system logs. This observation validates the hypothesis that log files are *local in the context of n-grams*, even to a higher extent than natural language corpora. As such, we point out, similar to prior research [64], we can leverage this localness attribute to improve the performance of language models for logs by augmenting them with caching mechanism and store the n-grams in the local context for quick access.

	Freq.	1-gram	2-gram	3-gram	5-gram	8-gram	10-gram
System logs	⩾1	69.09%	90.24%	95.44%	97.37%	99.60%	99.79%
	⩾2	38.02%	49.63%	48.59%	39.20%	30.66%	26.00%
English corpora	⩾1	56.74%	80.74%	93.13%	99.63%	99.97%	99.98%
	⩾2	26.38%	21.79%	15.68%	8.57%	6.34%	5.77%

TABLE II: System logs and English endemic n-gram stats.

B. ***RQ5: are log n-grams specific to their projects?***

In RQ4, we experimented with endemic n-grams which happen only in one system's log. Although endemism provides insight into exclusive n-grams, we still lack insight into the overall distribution of non-endemic n-grams. This is where *specificity* comes into effect. Specificity explains whether non-endemic n-grams favor specific system logs, *viz.*, whether non-endemic n-grams happen more often in one system than another, which sheds light on the *system-wide locality* of n-grams. For example, if n-gram σ is uniformly distributed across different log files, then all the files contain an equal number of the n-gram σ. Conversely, the more skewed the distribution becomes, the more *specific* and *localized* the n-gram σ becomes to a *specific* log file. For example, an n-gram that happens 100 times in all the log files, if it happens 93 times in one file and only once in the rest of the files, is highly skewed. For a set of log files $F = \{f_1, f_2, ..., f_{|F|}\}$, each non-endemic n-gram σ can happen in more than one file with probability distribution of $p(f_i(\sigma))$:

$$p(f_i(\sigma)) = \frac{\texttt{count (n-gram } \sigma \texttt{ in } f_i\texttt{)}}{\texttt{count (n-gram } \sigma \texttt{ in Set F)}} \tag{7}$$

As such, to measure the skewness of the distribution of n-gram σ in the set of log files F, defined as *locality entropy* [35], [64], $H_{\mathcal{L}}(\sigma)$, the formula is as follows:

$$H_{\mathcal{L}}(\sigma) = -\sum_{f_i \in F} p(f_i(\sigma)) \log_2 p(f_i(\sigma)) \tag{8}$$

Similar to *cross-entropy* values, the more skewed the distribution of the non-endemic n-grams is, the lower the *locality entropy* values will be. In the extreme case, *i.e.*, $H_{\mathcal{L}}(\sigma) == 0$, then σ is an *endemic* n-gram, and it happens only in one system logs. $H_{\mathcal{L}}(\sigma)$ reaches its max (*i.e.*, $\log_2|F|$) when the n-gram σ is uniformly distributed across F, *i.e.*, all the files in F contain equal number of n-gram σ.

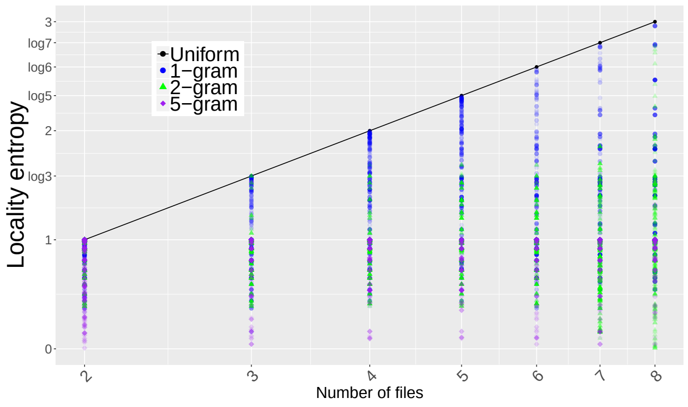


Fig. 5: Distribution of entropies for non-endemic n-grams, grouped by the number of files. "Uniform" represents that n-grams are distributed uniformly in the files.

Fig. 5 shows the values of *locality entropy* for different n-gram orders for file sizes, $|F| \in (2, 8)$. The horizontal axis represents the number of files ($|F|$) that contain the non-endemic n-grams, and the vertical axis shows the *locality entropy*, $H_{\mathcal{L}}$. The n-gram with varying orders, which are marked by different colors, share similar trends. According to this figure, non-endemic n-grams' entropies commonly fall below the uniform distribution (the solid black line in the figure), and as the number of files and the order of n-grams increases, the skewness becomes more apparent, which affirms the local tendencies of n-grams.

VI. RQ6: Log File Anomaly Detection

In the previous sections, we demonstrated the NLP characteristics of the log files. Based on our findings, we believe it is a valuable effort to explore how the NLP findings would help in the automated analysis of log files. As such, in the following, we focus on anomaly detection (AD) through log files by utilizing NLP. As modern large-scale computer systems continue to expand and service millions of users, their dependability becomes even more critical, and any noticeable downtime could result in millions of revenue and quality-of-service loss [1], [8], [38]. Consequently, to improve reliability, online observation of running systems for unforeseen abnormal behavior and potential problems has been the focus of prior research [20], [28], [29], [37], [48], [49], [65]. As log files contain information on the dynamic state of the system, prior research has utilized logs for AD [45], [49], [71], [74]. Log-based AD's goal is to accurately detect runtime system anomalies by processing the rich data gathered in the log files. AD is often modeled as a binary (*i.e., yes or no*) decision problem such that the input to the AD algorithm is a vector (or matrix) of events or time intervals, and the algorithm decides whether each event or interval is normal or abnormal. Encouraged by our findings from the naturalness and localness of the logs, we introduce ***ANALOG*** (**A**nomaly detection with **NA**tural language and **LOG**s) in the following.

Approach. We propose to utilize NLP techniques and consider log files as natural language sequences. Figure 6 shows the steps involved in our ANALOG approach:

- Initially, we collect the log files for the system under analysis during its normal behavior, *i.e.*, no anomalies. Similar to the approach in RQ1, we divide parts of the logs for training and testing of the n-gram model. We then preprocess and tokenize the logs and feed them into the n-gram model.
- Next, we *train* the n-gram model on the training data for an order of n which results in a good balance of performance and memory footprint. In our analysis, values in the range of $n \in (4, 5)$ are sufficient. During the training phase, we establish a ***baseline*** for entropy values for each system. Our analysis shows that normal log sequences result in entropy values, which are *'distinguishably'* lower than entropy values for abnormal log sequences. In this step, we also establish a ***threshold (Th)*** for the maximum value of normal entropies.
- Later, during the *testing phase*, we look for anomalies as the test log data is fed into the NLP model, and if no anomaly is detected, we achieve comparable to baseline perplexity or entropy values. On the other hand, if the test log data contains abnormal sequences, which appears surprising and perplexing to the n-gram model, the entropy values will be higher than the established baseline. Then these log sequences are singled out for further analysis by practitioners or developers.

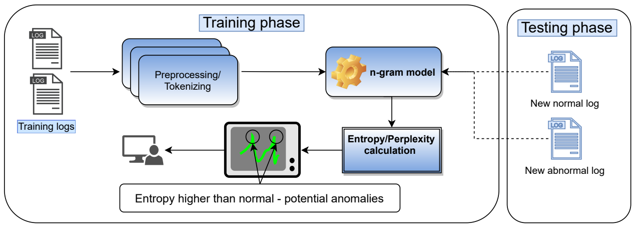

Training phase
Testing phase
Training logs
Preprocessing/ Tokenizing
n-gram model
Entropy/Perplexity calculation
New normal log
New abnormal log
Entropy higher than normal - potential anomalies

Fig. 6: ANALOG steps for anomaly detection through log files.

Usage scenario. The way we imagine practitioners and developers will use our approach for anomaly detection is as follows: during the ***training phase***, initially, we acquire a set of normal operational log files to train the n-gram model. This process also involves choosing a proper order for the n-gram model (*e.g.*, $n \in (4, 5)$), and detecting the baseline values of entropies. Based on the baseline range of entropy values, we use the *Hampel Filter* approach (Section VI-A) to assign a *threshold (Th)* for alarming an anomaly in case the entropy values go beyond *Th* during the testing phase. While in the *testing phase*, as the system is running and continuously generating new log records, we define an *observation window (ow)* for the recently generated logs. The length (in bytes) of *ow* can vary from a single log message up to chuck of log file (*e.g.*, 16 KB), depending on the volume of the logs and the desired granularity of the log analysis. During the continuous testing, the content of *ow* is fed into the NLP model, and the entropies *(E)* are evaluated. If $E > Th$, this chunk of log data *ow* is isolated as it potentially contains an anomalous log message. Because this usage scenario is an ***online*** approach, the *ow* moves to the next chuck of the log file as soon as new log messages are available. Regarding the granularity of *ow*, ideally, we will be able to test each line of the log file against the NLP model, as it is written to the storage medium, as the testing is quite fast compared to the training of the n-gram model. An incremental enhancement that can be implemented is to periodically retrain the NLP model on the most recent log data once every cycle, such as daily, overnight, weekly, to align the model with recent normal log changes, if any.

Figure 7 shows entropy values (*i.e.*, $\log(perplexity)$) for distinct orders of n-grams for both normal and abnormal log windows. Solid lines show normal log windows, and dashed lines represent anomalous log windows for 2-, 3-, and 5-gram models. The horizontal axis shows the size of the log window (in KB) that we inspect for perplexity calculation. For creating an abnormal log window, we synthetically inject an error message into the log file. Across the board, once there is an anomaly, the perplexity, and therefore entropy values increase, which signals that we have detected an anomalous log window.

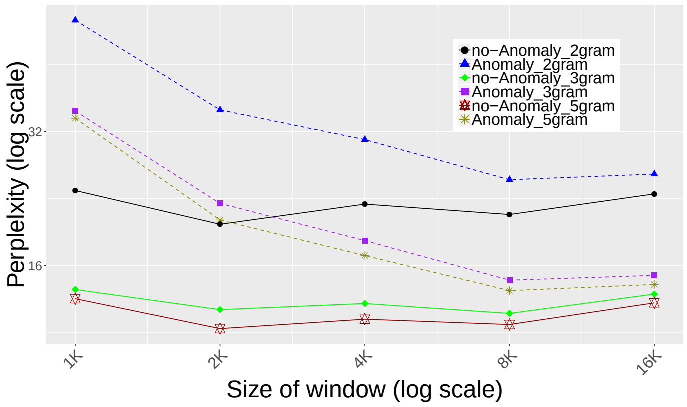


Fig. 7: Anomaly detection with perplexity values for HDFS.

A. Hampel Filter for Threshold Selection

Hampel filter [59] is a decision filter (*viz., yes or no*) that detects anomalies in the data vector if they lie far enough from the median to be deemed as an outlier. This filter depends on both the entropy vector width and an additional tuning parameter *t*, which explains the margins from the median. For a vector of entropy values ($E = \{e_1, e_2, ..., e_n\}$) measured from the training data, we can calculate the threshold (Th) according to the following formula: $Th = median(E) + t \times MAD$, where MAD stands for *median absolute deviation* and is defined as the median of the absolute deviations from the data's median $\tilde{E} = median(E)$: $MAD = median(|e_i - \tilde{E}|)$. The value of t is chosen based on the range of values in E, and according to our experiment on the eight projects, it falls in the range of $t \in [1, 8]$. As such, threshold Th is in the range of:

$$\tilde{E} + 1 \times MAD \leq Th \leq \tilde{E} + 8 \times MAD \tag{9}$$

Table III summarizes the *t* values for different systems. System abbreviations are from Table I. For each project, we evaluated different values of thresholds, and we chose the value of *t*, as an agreeable practice [62], to balance *Precision* and *Recall* and achieve the highest *F-Measure*, which we discuss in the following section. It is admissible to use different values of *t* to achieve a particular goal, such as maximizing *Precision* or *Recall*.

System	HS	SP	FW	WD	LS	TB	LB	ST
t	3.0	4.5	8.0	6.0	3.1	1.7	2.8	1.4

TABLE III: Values of t for different systems.

B. Evaluation

To evaluate the accuracy of anomaly detection approaches, we use *Precision, Recall, Fallout, F-measure, and Balanced Accuracy*, which are the commonly used metrics for evaluating anomalies in prior work [38], [74], [75]. These metrics are based on the confusion matrix [2], composed of four values: ❶ true positive (t_p) is the correctly identified abnormal log windows by the n-gram model. ❷ False positive (f_p) is the number of incorrectly identified normal log windows as abnormal ones. ❸ False negative (f_n) is the number of incorrectly identified abnormal log windows as normal ones. ❹ True negative (t_n) is the correctly identified not abnormal (*i.e.*, normal) log windows by the n-gram model.

Based on the confusion matrix, the definitions of the metrics are: ***Precision*** is the percentage of log test windows that are correctly identified as anomalies over all the log test windows that are identified as anomalies: $\frac{t_p}{t_p+f_p}$. ***Recall*** (or *true positive rate*) is the percentage of log test windows that are correctly identified as anomalies over all log windows containing anomalies: $\frac{t_p}{t_p+f_n}$. ***Fallout*** (or *false positive rate*) is the percentage of log test windows that are incorrectly identified as anomalies over all normal log windows: $\frac{f_p}{f_p+t_n}$. ***F-Measure*** is the harmonic mean of Precision and Recall: $\frac{2 \times precision \times recall}{precision + recall}$. ***Balanced Accuracy (BA)*** is the average of the proportion of anomalous and normal log windows that are correctly classified: $\frac{1}{2} \times (\frac{TP}{TP+FN} + \frac{TN}{TN+FP})$. BA is important for our analysis as only 10% of the synthetically generated cases are anomalous, *e.g.*, t_n is ~10X greater than t_p. In case there is an imbalance in the data (anomalous scenarios happen less often than normal scenarios [38]), BA is widely used [73], [75] for evaluation because it avoids over-optimism that traditional accuracy might suffer from. We perform two sets of experiments: 1) **synthetic anomalies**, 2) **comparison with PCA** [65].

Synthetic anomalies. For each log data, we randomly sample 512 MB of logs. We then train a 5-gram model on 90% of the data and keep the rest for testing. Next, we split the test data into 400 sequential samples of 4KB log windows. The idea is to simulate the continuous generation of log windows while the system is running. We then randomly inject anomalous messages in 10% ($anomaly_{freq.} \ll normal_{freq.}$) of the test windows. We calculate the entropy for each log window and detect an anomaly if the entropy is higher than the baseline threshold (Th).

Comparison with PCA. In this part, we compare our approach with the PCA-based anomaly detection and their provided labeled dataset [65], which is the log of HDFS system operations on various blocks. Although plenty of logs are available, this dataset is one of the few available labeled ones. Initially, to make the log parsing and template extraction manageable (required for [65]), we randomly select 20 MB of log data (~105K log lines and within 95% confidence [44]) and parse them with IPLoM [50] to extract event log templates belonging to each block (*i.e.*, session; denoted by *blk_-id* in the logs). Then, for each block, we create the sequence of events, *i.e.*, log window, and for each window, a label of normal or abnormal is provided with the dataset. We then split the data in 90%-10% for train-test and evaluate both PCA and ANALOG. PCA, like the majority of anomaly detection methods, requires both normal and abnormal samples to fit log windows into two sub-spaces, *viz.*, normal space, S_n, and anomaly space, S_a. ANALOG does not require historical abnormal instances, as it builds the n-gram model based on the normal instances, which provides a clear advantage for ANALOG over the majority of the anomaly detection methods [20], [23], [47], [65], which rely on witnessing anomalous instance in the training data to function.

C. Results

Synthetic anomalies. Figure 8 shows the entropy vector values for the logs of eight systems in our study. The x-axes show the log windows, and the y-axes represent the perplexity (*PP*) values. The horizontal blue line shows the overall trend-line for the *PP* values of the 400 log windows. The random sudden spikes in the *PP* values indicate that the n-gram model considers the content of these log windows surprising and not similar to the normal logs. As such, it identifies them as abnormal. Table IV shows the result of our anomaly detection for different systems. Figure 9 plots *receiver operating characteristic* (ROC) graph, which is the *true positive rate* against *false positive rate*. ROC illustrates the ability of our approach in classifying anomalies. The black line corresponds to randomly classifying normal and abnormal log windows, and the red dot on the top left corner shows the perfect classifier (PC), and the blue labels show the performance of ANALOG for eight evaluated projects (two labels, *SP* and *ST*, are overlapping). Our

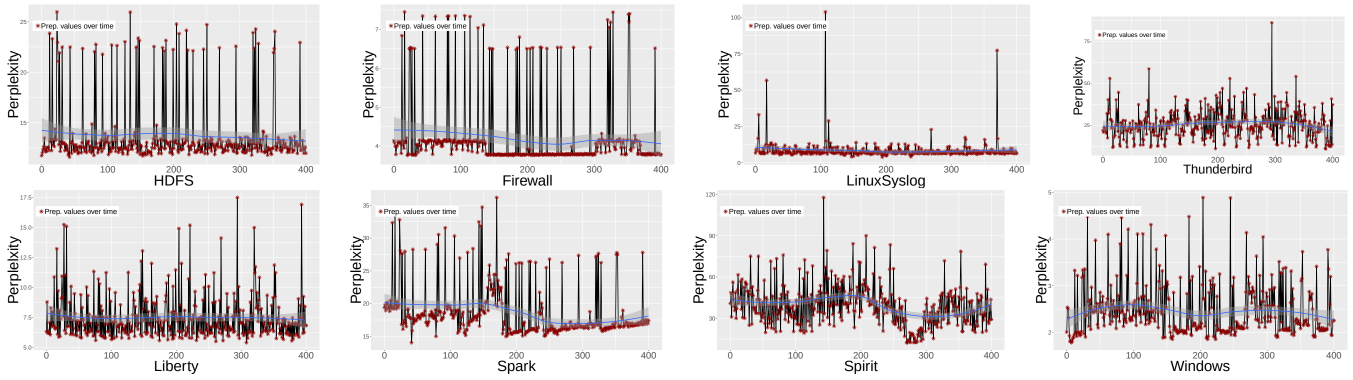


Fig. 8: Perplexity values for different projects.

System	tp	fp	tn	fn	Precision %	Recall %	Fallout %	F-Measure %	Balanced Accuracy %
HDFS (HS)	41	4	356	0	91.11	100	1.12	95.35	94.44
Spark (SP)	41	6	354	0	87.23	100	1.67	93.18	99.17
Firewall (FW)	41	3	357	0	93.18	100	0.83	96.47	99.58
Windows (WD)	27	6	354	14	81.81	65.85	1.67	72.97	82.09
LinuxSyslog (LS)	35	29	331	6	54.69	85.37	8.05	66.70	88.66
Thunderbird (TB)	23	24	336	18	48.93	56.10	6.67	52.27	74.72
Liberty (LB)	33	15	345	8	68.75	80.49	4.17	74.16	88.16
Spirit (ST)	28	22	338	13	56.00	68.29	1.67	61.54	81.09
Average	34	14	346	7	72.71	82.01	3.23	76.58	88.49

TABLE IV: Performance of anomaly detection with NLP models for different system logs.

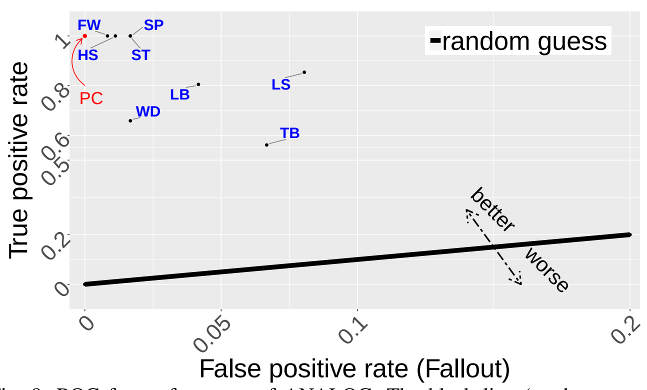
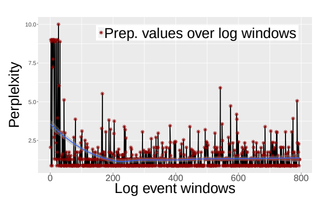
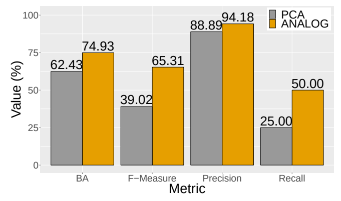


Fig. 9: ROC for performance of ANALOG. The black line (random guess) shows $y = x$, *i.e.*, 45°angle; however, the x-axis is stretched to provide a better separation on the ROC values of the evaluated projects.



(a) Perplexity values for log windows. Anomalous logs are presented at the beginning of the graph to show how overall perplexity trend changes for abnormal against normal logs.



(b) This chart compares the values for *Balanced Accuracy (BA)*, *F-Measure*, *Recall*, and *Precision* for ANALOG vs. PCA.

Fig. 10: PCA versus ANALOG performance comparison.

evaluations all fall on top of the random line, and closer to PC.

Comparison with PCA. In the second part of the experimentation, we compare our approach with the PCA-based anomaly detection approach proposed in [65]. Fig.10a shows the perplexity values for 795 test samples of log windows, with 32 anomalies gathered on the left side of the chart, and 763 normal samples. As observed from the blue trend line, the perplexity is lower for normal samples. Fig.10b summarizes the evaluation metrics for ANALOG, which are higher when compared with PCA across the board. We rationalize that because ANALOG assigns probabilities for the sequences of log events, *i.e.*, the sequences of n-grams, it can better distinguish normal and abnormal sequences when compared to PCA. We believe our anomaly detection results in Table IV and Fig. 10 are encouraging, and we expect further adaptation and continuation of our NLP-based research for other software engineering tasks.

Summary of the findings. Lastly, Table V provides a synopsis of our main findings and their implications from studying the NLP characteristics of logs.

VII. Related Work

NLP in software engineering tasks. Prior work has considered natural language processing techniques for software engineering (SE) tasks. Initial SE research in this field [31], [41] showed that the software's source code is regular and repetitive. As such, these works aimed to represent the sequences of the source code with n-gram language models. Consequently, applications of this representation emerged in software-related tasks such as bug reports [33], identifier name, class name, and next token code suggestion [12], [13], [18]. Following the work of Hindle *et al.* [41], Tu *et al.* [64] explored the idea of the localness of the source code and leveraged it to improve the performance of n-gram language models by introducing a local cache. Focusing on the source code's logging statements, He et. [35] investigated the NLP features of logging descriptions and used them to answer the question of *"what to log?"*. Inspired by and orthogonal to prior work, in this research, we utilize n-gram models to

Research questions (RQs)	Findings (Fs)	Implications (Is)
RQ1: does a natural repetitiveness and regularity exist in log files?	**F1:** language n-gram models capture a high degree of repetitiveness in software systems' logs, even to a higher degree than in common English corpora.	**I1:** compared with common English, the repetitiveness of log data can be better captured by statistical language models.
RQ2: is the regularity that the statistical language models capture merely because of the limited language of logs, or it is also a project-specific characteristic?	**F2:** n-gram-based statistical language models capture a high level of regularity and repetitiveness in within-project analysis and less cross-project regularity.	**I2:** the lower entropy values of log data are the outcome of the predictable repetitiveness within each system's logs, and not the limited language of logs.
RQ3: how does Zipf's law capture the repetitiveness of high-rank tokens in log files?	**F3:** from Zipf's law calculation, log files illustrate a higher concentration of high-rank tokens when compared to the English corpora.	**I3:** logs are more predictable, and language models potentially perform better in predicting the next token of a sequence when applied on log data.
RQ4: are log n-grams *endemic* to their projects?	**F4:** a significant percentage of endemic n-grams are repetitively used in the local context of logs.	**I4:** log files are locally endemic. Endemic n-grams in logs are the artifact of endemic n-grams in the source code and software itself.
RQ5: are log n-grams *specific* to their projects?	**F5:** each system tends to use its own set of n-grams more frequently, which is reflected in its logs.	**I5:** log files are locally specific. **F4** and **F5** enable more efficient analysis of logs with localized caching models.
RQ6: how the logs' naturalness and localness can help with the automated analysis of the log files?	**F6:** with an n-gram model that is trained on normal logs, abnormal logs result in higher-than-normal entropy values during testing.	**I6:** this finding opens up an avenue of research to utilize NLP attributes for automated log analysis tasks, such as anomaly detection.

TABLE V: Summary of RQs and our findings.

characterize the naturalness and localness of log files and to improve automated log analysis by leveraging these attributes.

Automated log analysis. As software logs contain rich information on the runtime state of the systems, their analysis has been the focal point of various prior research to improve systems' reliability and user experience, such as anomaly detection [29], [65], user statistics [46], fault detection and diagnosis [78]. These works typically employ data mining and learning algorithms to efficiently analyze a large scale of logs, which also involves log collection [27] and log parsing [36]. For example, Xu *et al.* [65] leveraged a dimension reduction algorithm (PCA) to distinguish normal and abnormal events in distributed systems. In this paper, however, we propose to uncover the NLP characteristics of logs, and we leverage them for benefiting automatic log analysis.

Anomaly detection with logs. Anomaly detection refers to the task of uncovering events that do not follow the expected behavior in the system [22]. Possibly, an anomaly in the system might turn to a fault, an error, and eventually to a system failure, if left untreated [15]. As such, log files, because of their rich content of runtime information and events, are widely utilized for computer systems' anomaly detection [20], [21], [28], [74]. Different from the prior work, we present an NLP-based method to investigate anomalies in software logs. As an advantage, because we utilize NLP features of logs, our approach does not require a log parsing step, which is the first step of every log analysis approach [36]. Two main categories of anomaly detection include supervised (*e.g.*, [20], [28], [29]) and unsupervised (*e.g.*, [48], [49], [65]) methods. In industry settings, a system may encounter only very few anomalies per year [38]. As such, a long list of prior methods [17], [20], [47], [65], [72], which relies on seeing anomalous instances in the training data, becomes ineffective. We believe because ANALOG does not require historical abnormal instances, as it builds the n-gram model based on the normal instances, it has an edge on such approaches.

VIII. Threats to Validity and Discussion

External threats to the validity reflect on the generalization of our work to other such software projects and log files. In this research, we conducted our NLP analysis on logs of eight software systems. We picked the systems from different domains and assumed our approach is independent of the underlying programming language, source code, and the software architecture that the system is implemented with. However, since other software systems may follow different logging practices, our findings may not accurately extend and generalize to logs of such other systems. Regarding ***internal threats*** to the validity, we rely on the accuracy of the n-gram model to calculate the entropies. Additionally, the threshold selection, and false positive and false negative values can affect the accuracy of our anomaly detection approach. Finally, due to the limited availability of labeled datasets, although we conducted experimentation on HDFS and synthetic anomaly datasets, future opportunities that give access and enable us to evaluate ANALOG on logs of large-scale and enterprise software systems would further solidify our findings.

IX. Conclusion and Future Work

This paper explores the natural language attributes of software logs. Guided by a set of research questions, our findings confirm that log files, as an artifact of software systems, are natural and local, even more so than common English corpora. We show how the NLP characteristics of log files can be leveraged for log analysis tasks, and we present *ANALOG*, an anomaly detection tool that is built upon NLP features and outperforms the prior work.

In this research, our primary focus has been to show the NLP characteristics of the logs and illustrate the potential applications of NLP for log analysis while not limiting our approach to anomaly detection techniques. Therefore, as a future direction, besides anomaly detection, we look into extending our work to other software analysis tasks, such as extracting system security infringements from the system and network logs. Additionally, we also aim to leverage deep learning (DL) language models to potentially improve our approach and have a comparison against DL anomaly detection approaches.

References

[1] Amazon's one hour of downtime on Prime Day may have cost it up to $100 million in lost sales. https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7.

[2] Confusion matrix. https://en.wikipedia.org/wiki/Confusion_matrix.

[3] cross-entropy. http://en.wikipedia.org/wiki/Cross_entropy.

[4] Data for log naturalness analysis. https://github.com/sgholamian/naturalness_of_software_logs.

[5] N-gram model. https://en.wikipedia.org/wiki/N-gram.

[6] Project Gutenburg. https://www.gutenberg.org/.

[7] Project Gutenburg on Wikipedia. https://en.wikipedia.org/wiki/Project_Gutenberg.

[8] When It Goes Down, Facebook Loses $24,420 Per Minute. https://www.theatlantic.com/technology/archive/2014/10/facebook-is-losing-24420-per-minute/382054/.

[9] Apache Spark. https://spark.apache.org/, 2019.

[10] The Apache Software Foundation. logging services project. http://logging.apache.org/, 2020.

[11] Hadoop Distributed File System. hhttps://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2020.

[12] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

[13] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49, 2015.

[14] M. Allamanis and C. Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 472–483, 2014.

[15] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[16] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio. End-to-end attention-based large vocabulary speech recognition. In *2016 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 4945–4949. IEEE, 2016.

[17] C. Bertero, M. Roy, C. Sauvanaud, and G. Trédan. Experience report: Log mining using natural language processing and application to anomaly detection. In *28th International Symposium on Software Reliability Engineering (ISSRE 2017)*, page 10p, 2017.

[18] A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel. Learning python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307*, 2016.

[19] S. Bird, E. Klein, and E. Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[20] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*, pages 111–124, 2010.

[21] J. Breier and J. Branišová. A dynamic rule creation based anomaly detection method for identifying security breaches in log records. *Wireless Personal Communications*, 94(3):497–511, 2017.

[22] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009.

[23] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 36–43. IEEE, 2004.

[24] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 217–231, 2014.

[25] K. W. Church and W. A. Gale. A comparison of the enhanced good-turing and deleted estimation methods for estimating probabilities of english bigrams. *Computer Speech & Language*, 5(1):19–54, 1991.

[26] S. Di, R. Gupta, M. Snir, E. Pershey, and F. Cappello. Logaider: A tool for mining potential correlations of hpc log events. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 442–451. IEEE, 2017.

[27] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 139–150, 2015.

[28] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[29] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 149–158. IEEE, 2009.

[30] X. Fu, R. Ren, J. Zhan, W. Zhou, Z. Jia, and G. Lu. Logmaster: Mining event correlations in logs of large-scale cluster systems. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 71–80. IEEE, 2012.

[31] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156, 2010.

[32] E. Gabrilovich and S. Markovitch. Wikipedia-based semantic interpretation for natural language processing. *Journal of Artificial Intelligence Research*, 34:443–498, 2009.

[33] S. Haiduc, V. Arnaoudova, A. Marcus, and G. Antoniol. The use of text retrieval and natural language processing in software engineering. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 898–899, 2016.

[34] K. Hätönen, J. F. Boulicaut, M. Klemettinen, M. Miettinen, and C. Masson. Comprehensive log compression with frequent patterns. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 360–370. Springer, 2003.

[35] P. He, Z. Chen, S. He, and M. R. Lyu. Characterizing the natural language descriptions in software logging statements. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 178–189. ACM, 2018.

[36] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu. An evaluation study on log parsing and its use in log mining. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 654–661. IEEE, 2016.

[37] S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218. IEEE, 2016.

[38] S. He, J. Zhu, P. He, and M. R. Lyu. Loghub: A large collection of system log datasets towards automated log analytics. *arXiv preprint arXiv:2008.06448*, 2020.

[39] K. Heafield. Kenlm: Faster and smaller language model queries. In *Proceedings of the sixth workshop on statistical machine translation*, pages 187–197, 2011.

[40] K. Heafield, I. Pouzyrevsky, J. H. Clark, and P. Koehn. Scalable modified kneser-ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 690–696, 2013.

[41] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[42] D. Jurafsky. *Speech & language processing*. Pearson Education India, 2000.

[43] P. Koehn. *Statistical machine translation*. Cambridge University Press, 2009.

[44] R. V. Krejcie and D. W. Morgan. Determining sample size for research activities. *Educational and psychological measurement*, 30(3):607–610, 1970.

[45] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 251–261, 2003.

[46] G. Lee, J. Lin, C. Liu, A. Lorek, and D. Ryaboy. The unified logging infrastructure for data analytics at twitter. *arXiv preprint arXiv:1208.4171*, 2012.

[47] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo. Failure prediction in ibm bluegene/l event logs. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 583–588. IEEE, 2007.

[48] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 102–111. IEEE, 2016.

[49] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining invariants from console logs for system problem detection. In *USENIX Annual Technical Conference*, pages 1–14, 2010.

[50] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 1255–1264, New York, NY, USA, 2009. ACM.

[51] J. B. Marino, R. E. Banchs, J. M. Crego, A. de Gispert, P. Lambert, J. A. Fonollosa, and M. R. Costa-jussà. N-gram-based machine translation. *Computational linguistics*, 32(4):527–549, 2006.

[52] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, 2013.

[53] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 353–366, 2012.

[54] A. Y. Ng et al. Preventing" overfitting" of cross-validation data. In *ICML*, volume 97, pages 245–253. Citeseer, 1997.

[55] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 575–584. IEEE, 2007.

[56] A. J. Oliner and A. Aiken. Online detection of multi-component interactions in production systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 49–60. IEEE, 2011.

[57] A. J. Oliner, A. Aiken, and J. Stearley. Alert detection in system logs. In *2008 Eighth IEEE International Conference on Data Mining*, pages 959–964. IEEE, 2008.

[58] A. Oprea, Z. Li, T.-F. Yen, S. H. Chin, and S. Alrwais. Detection of early-stage enterprise infection by mining large-scale log data. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 45–56. IEEE, 2015.

[59] R. K. Pearson, Y. Neuvo, J. Astola, and M. Gabbouj. Generalized hampel filters. *EURASIP Journal on Advances in Signal Processing*, 2016(1):1–18, 2016.

[60] T. Plötz and G. A. Fink. Markov models for offline handwriting recognition: a survey. *International Journal on Document Analysis and Recognition (IJDAR)*, 12(4):269, 2009.

[61] D. M. Powers. Applications and explanations of zipf's law. In *New methods in language processing and computational natural language learning*, 1998.

[62] D. M. Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*, 2020.

[63] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[64] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014.

[65] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132. ACM, 2009.

[66] X. Xu, L. Zhu, I. Weber, L. Bass, and D. Sun. Pod-diagnosis: Error diagnosis of sporadic operations on cloud applications. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 252–263. IEEE, 2014.

[67] K. Yao, H. Li, W. Shang, and A. E. Hassan. A study of the performance of general compressors on log files. *Empirical Software Engineering*, 25(5):3043–3085, 2020.

[68] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 143–154, 2010.

[69] D. Yuan, S. Park, and Y. Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, pages 102–112. IEEE Press, 2012.

[70] C. Zhai and S. Massung. *Text data management and analysis: a practical introduction to information retrieval and text mining*. Association for Computing Machinery and Morgan & Claypool, 2016.

[71] B. Zhang, H. Zhang, P. Moscato, and A. Zhang. Anomaly detection via mining numerical workflow relations from logs. In *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pages 195–204. IEEE, 2020.

[72] K. Zhang, J. Xu, M. R. Min, G. Jiang, K. Pelechrinis, and H. Zhang. Automated it system failure prediction: A deep learning approach. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1291–1300. IEEE, 2016.

[73] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 644–653. IEEE, 2005.

[74] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, et al. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 807–817, 2019.

[75] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 415–425. IEEE Press, 2015.

[76] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 121–130. IEEE, 2019.

[77] G. K. Zipf. *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio Books, 2016.

[78] D.-Q. Zou, H. Qin, and H. Jin. Uilog: Improving log-based fault diagnosis by log analysis. *Journal of computer science and technology*, 31(5):1038–1052, 2016.