CSC-482-03-2238 - System Log Glossary Search (SLGS)

Author: Daniel Gonzalez

CSC 482 Speech and Language Processing Section 3 - Professor Foaad Khosmood

Introduction

I have chosen to undertake the "*System Log Glossary Search"* project, which stands out due to its complexity and direct applicability to real-world scenarios where software maintenance and error tracking are crucial. Within the context of an organization, teams often utilize particular system dependencies and operating systems that can introduce unique challenges and, at times elusive debugging scenarios, despite the potential presence of straightforward solutions (IE: change in Linux system dependencies).

This project aims to highlight these issues and ultimately improve the developer experiences. In today's software development landscape, Continuous Integration (CI) has become a common practice in industrial and open-source software development. Although Continous Integration (CI) has enhanced various facets of the software development process, issues that arise during CI builds present a significant challenge to development efficiency, as more time is dedicated to rectifying these issues, failed builds can considerably disrupt the development workflow and result in substantial financial expenses. The "*System Log Glossary Search"* project is pivotal in mitigating these challenges by furnishing a comprehensive tool for ingesting system logs and pinpointing the fundamental origins of CI failures. By integrating this project's capabilities into the CI pipeline, teams can streamline their workflows to other developers, minimizing periods of inactivity, and elevating the overall software quality.

Datasource

In order to fetch logs from github, I created a python script that leverages the GitHub API ro retrieve build logs from a list of top Java repositories. It starts by querying each repository's workflows to gather information about the defined workflows. Then, for each workflow, it fetches details about individual runs, including their statuses and associated commit information. The script collects and stores run logs for each run, allowing for a comprehensive analysis of the Continuous Integration (CI) processes within these repositories. This data retrieval process is automated and systematic, offering valuable insights into the performance and issues related to CI pipelines in the selected repositories, making it an essential tool for software development and maintenance tasks.

```
top_java_repos = [
    "google/guava", # 25 most starred repos
    "CyC2018/CS-Notes",
    "Snailclimb/JavaGuide",
    "iluwatar/java-design-patterns",
    "doocs/advanced-java",
    "macrozheng/mall",
    "spring-projects/spring-boot",
    "elastic/elasticsearch",
    "kdn251/interviews",
    "TheAlgorithms/Java",
    "azl397985856/leetcode",
    "google/guava",
    "ReactiveX/RxJava",
    "square/okhttp",
    "youngyangyang04/leetcode-master",
    "square/retrofit",
    "apache/dubbo",
    "apache/spark",
    "skylot/jadx",
    "PhilJay/MPAndroidChart",
    "jeecgboot/jeecg-boot",
    "autonomousapps/dependency-analysis-gradle-plugin", # common gradle plugins
    "modrinth/minotaur",
    "klawson88/liquiprime",
    "klawson88/liquigen",
    "bkmbigo/epit",
    "TanVD/kosogor",
    "kazurayam/inspectus4katalon-gradle-plugin",
    "robertfmurdock/jsmints",
    "steklopod/gradle-ssh-plugin",
    "gesellix/gradle-docker-plugin",
    "steklopod/gradle-ssh-plugin",
    "robertfmurdock/testmints",
    "DanySK/gradle-kotlin-qa",
    "JetBrains/kotlin", # common libs
```

Figure 1: Fetched Repositories

Although several error logs were provided from this scraping process, I also gained access to a substantial dataset from Thomas Raush's research titled "***An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software.***" This dataset comprises a comprehensive empirical study of CI build failures in 14 java-based open-source software projects, amounting to a total size of 4.2 gigabytes.

- **git-repositories.tar.gz**
  contains the Git repositories analyzed. Each commit that triggered a Travis-CI build (beginning from the time the continuous mining process was started) is tagged with the Travis-CI build ID
- **logcat-categories-and-patterns.tar.gz**
  contains the error categories of the analyzed log files that were coded from the exploratory analysis
- **projects.csv**
  maps the Travis-CI project ID to the GitHub project slug
- **travis-ci-logs.tar**
  contains for each project the analyzed log files that were scraped from Travis-CI. Each txt.gz file in the project folders contains the log of one Travis *job*. A build may have multiple jobs.

Figure 2: Java OSS Travis-CI Build Failure Dataset

SVO - System Description

Using the manually extracted data, the following SVO system is designed to extract and analyze specific java logs found from regex, specifically error logs, and perform Semenatic Subject-Verb-Object (SVO) extraction on these logs. The system comprises two main components: a log extraction process and an SVO extraction process.

Log Extraction Process:

- The log extraction process is initiated by specifying a source directory containing compressed log files in ZIP format.
- The system iterates through each ZIP file and extracts the text contents of files with the ".txt" extension found within.
- During extraction, the system employs regular expressions to identify log entries based on a predefined pattern. Log entries typically consist of timestamps, log levels (e.g., ERROR, WARN), and log messages.
- Extracted log entries are then appended to a global CSV file, associating each entry with its source file for reference.

SVO Extraction Process:

- The SVO extraction process utilizes the extracted log entries from the global CSV file.
- Each log entry is pre-processed to remove specific patterns (e.g., "WARNING |", "ERROR |", "WARNING:") to ensure cleaner analysis.
- The system utilizes the spaCy natural language processing library to tokenize and parse each log entry's text.
- It identifies Subject-Verb-Object (SVO) relationships within the log entries, extracting subjects, verbs, and objects.
- The extracted SVO relationships are then aggregated and saved to a CSV file for further analysis.
- Additionally, the system handles negations (e.g., "not," "never") and conjunctions (e.g., "and") in log entries to improve the accuracy of SVO extraction.

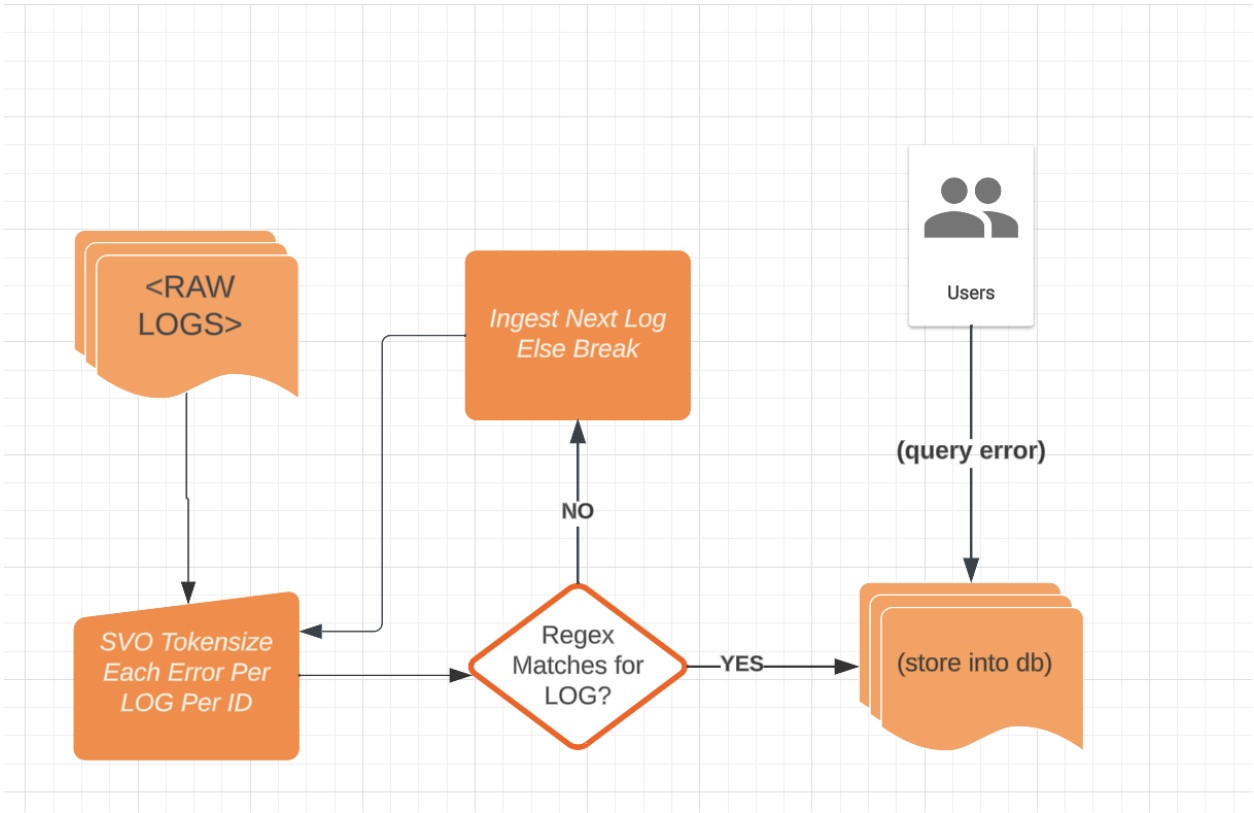

Figure 3: SVO - DataFlow Diagram

This system proxies a pipeline for log data extraction and semantic analysis, enabling users to hain insights into the structure and content of logs, partifcurally error logs. The extracted SVO relationships offer a structured representation of log information, aiding in automated analysis and error detection in log data.

Using this datastore, we can query for specific commits from the log files. The implementation script parses the log into SVO structures and returns the ID with the most matches. It provides a GitHub commit link based on the matched log entry to facilitate debugging and monitoring (provided by the ID).

```
→  CSC482-Log-NLP git:(main) python3 searchSVO.py sample_log_1.txt
From 1812 SVO matches, here is the link to a commit where the builds matched the errors provided:
https://github.com/square/okhttp/commit/4984568367caaf359b82c452bd28b5e192824d1c
→  CSC482-Log-NLP git:(main) python3 searchSVO.py sample_log_2.txt
From 209 SVO matches, here is the link to a commit where the builds matched the errors provided:
https://github.com/square/okhttp/commit/4984568367caaf359b82c452bd28b5e192824d1c
```

Figure 4: SVO - Implementation

Log Embeddings - System Description

Using the provided datasource, this system is designed to preprocess and embed log messages using BERT (Bidirectional Encoder Representations from Transformers) to capture semantic information and support various downstream natural language processing tasks. The system consists of two primary components: log preprocessing and BERT embedding generation.

Log Preprocessing:

- The log preprocessing component starts by loading log messages from a CSV file, presumed to contain log entries.
- Each log message undergoes preprocessing, including the removal of URLs and conversion to lowercase, ensuring cleaner and consistent input for embedding.
- Preprocessed log messages are then fed into the BERT embedding generation process for further analysis.

BERT Embedding Generation:

- The BERT embedding generation component employs a pre-trained BERT model and tokenizer, provided by the Hugging Face Transformers library.
- It iterates through each preprocessed log message, tokenizes it, and generates BERT embeddings.
- These embeddings serve as numerical representations of the log messages, capturing their contextual information.
- The embeddings are stored alongside their corresponding log messages in an output CSV file for use in subsequent analysis (39718 unique build logs).

This system facilitates the transformation of raw log messages into meaningful numerical embeddings, enabling error log analysis to be used for implementation by computing cosine similarity between samples in X and Y as shown below.

Figure 5: Embeddings - Implementation

Evaluation

In order to evaluate the accuracy of our embeddings we assess varying random log entries and query the entry using BERT embeddings. Our testing begins by loading the precomputed BERT embeddings from a CSV file and preprocessing provided log messages. Then we calculate the cosine simarility scores between the query log and all existing entries on the database, determining the highest similarity score for each query. The resulting average simailrity score serves as an evaluation metric for different sample sizes. As we scale to higher amounts we slightly dip in performance but this is likely due to similar error logs being rendered or unexpected parsing in the logs causing noise.

Results Matrix

Cosine Similarity	LOG COUNT (N = 25%)
1.0000	100
0.9998	1000
0.9837	5000
0.9711	50000

Figure 6: Embeddings - Results

Improvements

- Enhancement in data preprocessing, errors can arise in varies structures, Gradle, SL4J, and LOG4J each arise in there own format. Preprocessing for each, would reduce the data size and also be a helpful abstraction for the model to ingest from.
- Fine-tuning the BERT model on log-specific data or using domain-specific embeddings if available. LogBERT For example: https://arxiv.org/abs/2103.04475
- Implement optimizations for better performance, especially when dealing with a large number of logs. This might involve parallel processing or batch processing to speed up similarity computations. 4.4 Gigabytes of CSV is difficult to process independently and can be better distributed.
- Better embeddings assessment, which is not obvious since we need generated/artificial logs to conduct true testing.

Conclusion

This project proved an effective approach for querying simarilies in CI logs. Utilzing a communication of Pandas, SpaCy for semantics, and BERT embeddings for contextual comprehension of log messages, the system provides an approach to debugging and error tracing within an organization. It adeptly analyzes large datasets such as the one provide from Thomas Raush's research. Future enhancements could include refined data processing, log specific model fine-tuing, and parallel log processing which could deepen the system's analytical capabilities.

References:

Rausch, T. (2018). Java OSS Travis-CI Build Failure Dataset [Data set]. Zenodo. https://doi.org/10.5281/zenodo.1745638

Guo, H., Yuan, S., & Wu, X. (2021). LogBERT: Log Anomaly Detection via BERT. ArXiv. /abs/2103.04475