# A System for Automatic Information Extraction from Log Files

by

Anubhav Chhabra

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the degree of
Master of Computer Science
in
Electrical and Computer Engineering
University of Ottawa
Ottawa, Ontario, Canada

## Abstract

The development of technology, data-driven systems and applications are constantly revolutionizing our lives. We are surrounded by digitized systems/solutions that are transforming and making our lives easier. The criticality and complexity behind these systems are immense. So as to meet user satisfaction and keep up with the business needs, these digital systems should possess high availability, minimum downtime, and mitigate cyber attacks. Hence, system monitoring becomes an integral part of the lifecycle of a digital product/system. System monitoring often includes monitoring and analyzing logs outputted by the systems containing information about the events occurring within a system. The first step in log analysis generally includes understanding and segregating the various logical components within a log line, termed log parsing.

Traditional log parsers use regular expressions and human-defined grammar to extract information from logs. Human experts are required to create, maintain and update the database containing these regular expressions and rules. They should keep up with the pace at which new products, applications and systems are being developed and deployed, as each unique application/system would have its own set of logs and logging standards. Logs from new sources tend to break the existing systems as none of the expressions match the signature of the incoming logs. The reasons mentioned above make the traditional log parsers time-consuming, hard to maintain, prone to errors, and not a scalable approach. On the other hand, machine learning based methodologies can help us develop solutions that automate the log parsing process without much intervention from human experts. NERLogParser [58] is one such solution that uses a Bidirectional Long Short Term Memory

(BiLSTM) architecture to frame the log parsing problem as a Named Entity Recognition (NER) problem. There have been recent advancements in the Natural Language Processing (NLP) domain with the introduction of architectures like Transformer [64] and Bidirectional Encoder Representations from Transformers (BERT) [17]. However, these techniques have not been applied to tackle the problem of information extraction from log files. This gives us a clear research gap to experiment with the recent advanced deep learning architectures.

This thesis extensively compares different machine learning based log parsing approaches that frame the log parsing problem as a NER problem. We compare 14 different approaches, including three traditional word-based methods: Naive Bayes, Perceptron and Stochastic Gradient Descent; a graphical model: Conditional Random Fields (CRF); a pre-trained sequence-to-sequence model for log parsing: NERLogParser [58]; an attention-based sequence-to-sequence model: Transformer Neural Network; three different neural language models: BERT, RoBERTa and DistilBERT; two traditional ensembles and three different cascading classifiers formed using the individual classifiers mentioned above. We evaluate the NER approaches using an evaluation framework that offers four different evaluation schemes that not just help in comparing the NER approaches but also help us assess the quality of extracted information.

The primary goal of this research is to evaluate the NER approaches on logs from new and unseen sources. To the best of our knowledge, no study in the literature evaluates the NER methodologies in such a context. Evaluating NER approaches on unseen logs helps us understand the robustness and the generalization capabilities of various methodologies. To carry out the experimentation, we use In-Scope and Out-of-Scope datasets. Both the datasets originate from entirely different sources and are entirely mutually exclusive. The

In-Scope dataset is used for training, validation and testing purposes, whereas the Out-of-Scope dataset is purely used to evaluate the robustness and generalization capability of NER approaches.

To better deal with logs from unknown sources, we propose Log Diversification Unit (LoDU), a unit of our system that enables us to carry out log augmentation and enrichment, which helps make the NER approaches more robust towards new and unseen logs. We segregate our final results on a use-case basis where different NER approaches may be suitable for various applications. Overall, traditional ensembles perform the best in parsing the Out-of-Scope log files, but they may not be the best option to consider for real-time applications. On the other hand, if we want to balance the trade-off between performance and throughput, cascading classifiers can be considered the go-to solution.

**Acknowledgements**

First and foremost, I would like to thank my supervisors Professor Paula Branco, Professor Guy-Vincent Jourdan and Professor Herna Viktor. I am really grateful to have been carrying out my research work under the supervision and guidance of such supportive and knowledgeable people. I want to express my gratitude to Professor Guy-Vincent Jourdan and Professor Paula Branco, who convinced me to shift from a course-based to a research-based program. It was one of the best decisions that I have ever made.

I want to thank my family: my mom, dad and brother Rahul for their unconditional support in the past years. My friends have been integral to this journey, so I would like to thank Kanwar, Anushree, Dilpreet, and Geetpal. I was lucky to have Kanwar to encourage me and help me make decisions. Finally, I would like to thank Anushree for believing in me and being rock-solid support throughout this adventure.

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

**BERT** Bidirectional Encoder Representations from Transformers. iii, 5, 14, 22, 31, 41–44, 46, 47, 49, 51, 56, 66, 68, 69, 80, 83, 84, 86, 89, 96, 99, 105, 106, 108, 114–117, 120, 121, 124

**BiLSTM** Bidirectional Long Short Term Memory. ii, iii, 4, 16–18, 124

**CB Cascade** CRF-BERT Cascade. 51, 66, 86, 89, 92, 96, 99, 103, 108, 118

**CBT Cascade** CRF-BERT-Transformer Cascade. 51, 66, 86, 89, 92, 96, 99, 102, 108, 116, 118

**Cisco ASA** Cisco Adaptive Security Appliance. 62, 89, 101, 103, 106, 116

**Cisco IOS** Cisco Internetwork Operating System. 62, 89, 92, 101, 103, 106, 113, 116, 117

**CNN** Convolutional Neural Network. 16

**CPU** Central Processing Unit. 81

**CRF** Conditional Random Fields. iii, 16, 19, 22, 33–35, 46, 47, 49–51, 66, 69, 80, 83, 86, 89, 91–93, 98, 99, 105, 108, 113–117, 119, 120, 124

**CSV** Comma Separated Values. 57, 60

**CT Cascade** CRF-Transformer Cascade. 51, 66, 86, 90–92, 96, 99, 100, 103, 108, 113, 115, 118

**ENT** Entity. 16

**GB** Gigabyte. 81

**GloVe** Global Vectors. 17

**GPU** Graphical Processing Unit. 81

**IOB** Inside-Outside-Beginning. 16, 17, 21–23, 33–35, 37, 44, 54, 56, 57, 62, 69, 71, 73, 74, 80

**IP** Internet Protocol. 17, 25, 29, 35, 60

**IPLoM** Iterative Partitioning for Log Mining. 12

**IQR** Inter Quartile Range. 90, 91

**JSON** Javascript Object Notation. 2, 10, 54

**LAnoBERT** Log Anomaly BERT. 14, 15

**LenMa** Length Matters. 13

**LoDU** Log Diversification Unit. iv, 6, 22, 23, 30, 56, 59, 63, 81, 82, 105, 106, 112, 122

**LoPU** Log Parsing Unit. 22, 30, 31, 56, 59, 62, 63, 81

**LSTM** Long Short Term Memory. 16, 17, 38

**ML** Machine Learning. 3, 5, 22, 66, 121

**MLM** Masked Language Modelling. 42, 43

**MV Ensmbl** Majority Voting Ensemble. 47, 66, 86, 89, 91, 95, 114, 119

**NER** Named Entity Recognition. iii, iv, 3–8, 15–21, 31, 33, 34, 47, 51, 56, 57, 59, 66, 72–74, 76, 77, 80, 82, 86, 87, 90, 91, 93–95, 97, 105–107, 109, 112–114, 117–125

**NERLogParser** Named Entity Recognition Log Parser. 3–5, 17–19, 22, 46, 59, 66, 69, 86, 89, 91, 93, 96, 98, 108, 113, 120

**NLP** Natural Language Processing. iii, 4–6, 8, 14, 16, 18, 34, 38, 41, 42

**NSP** Next Sentence Prediction. 42, 43

**RAM** Random Access Memory. 81

**RNN** Recurrent Neural Network. 38

**RoBERTa** Robustly Optimized BERT Approach. 5, 31, 43, 66, 80, 89, 91, 96, 108, 111, 121

**SGD** Stochastic Gradient Descent. 31, 66, 86, 89, 111, 118

**SIEM** Security Information and Event Management. 2

**SLURM** Simple Linux Utility for Resource Management. 80

**SV Ensmbl** Soft Voting Ensemble. 47, 66, 86, 89–91, 93, 95, 98, 108, 114, 115, 119

**WAE** Windows Application Events. 62, 89, 93, 102, 103, 113, 116, 117

**WSE** Windows Security Events. 62, 89, 98, 102, 116

**WSYE** Windows System Events. 62, 89, 101, 103, 113, 117

# Chapter 1

# Introduction

The development in technology and the data world is constantly revolutionizing our lives. A plethora of industries have moved and are moving towards digitization. Watching your favourite movie on NetFlix, getting your groceries delivered to your doorstep using Instacart, booking a taxi service using Uber, and digitizing your home using Alexa are some examples that many of us enjoy in our day to day lives. We are unknowingly surrounded by solutions that make our lives easier by providing services with a single command. The shift toward cloud computing and distributed systems has helped these solutions scale globally. The criticality and complexity behind these digital systems are immense. To meet the business needs and ensure high availability, these systems should possess minimum downtime. There could be various reasons for the downtime, including deploying a buggy code into the production environment, the server not being able to scale to the number of incoming requests, network outages, cyber attacks, and power outages, to name a few. System health and status monitoring thus become an integral part of the routine in the Information Technology industry. System monitoring generally involves monitoring the logs outputted by the system to reproduce bugs and carry out root cause analysis. Even the modern-day Security Information and Event Management (SIEM) tools rely on logs to detect and prevent cyber attacks.

A log is an unstructured piece of textual data outputted by a system, device, or application containing information about the events taking place within a system throughout its lifecycle. The logging of events could happen on a device's storage or remotely over another server. The content within logs may change based on the event taking place and the source from which it originates. Systems generally log information about the events in a file with a .log extension. The analysis of log files commences with the parsing of the log instances. Log parsing involves the extraction of logical units from log instances. It is difficult for developers, analysts and operatives to efficiently investigate vast amounts of unstructured text. Thus, there is a need for a structured format. Searching for a keyword is one of the most common operations performed on logs. For example, searching for a particular transaction ID for which the system broke. Searching through such amounts of unstructured text is a challenge for operatives involved in the investigation. Parsed logs possess a Javascript Object Notation (JSON) like structured format containing key-value pairs, making the searching operation efficient. The SIEM tools require the logs to be parsed, and the extracted information act as features for each log instance. The features extracted from the key-value pairs are then used to prevent attacks, followed by the appropriate steps.

Most traditional log parsing approaches focus on building regular expressions, rules or grammars to carry out information extraction [5,8,51]. We need a team of experts to build regular expressions or rules for each type of log present in the system, and this process needs to be repeated for all the systems present. Regular expressions or rules need to be regularly maintained because of the addition or removal of applications or systems from the environment. Since there is substantial human intervention, the process becomes prone to errors. A single error in the rules can cause the system to break and be challenging to debug. Hence, the entire process is costly in terms of time and human effort. Machine learning based solutions can be used to extract information from log files. Named Entity Recognition Log Parser (NERLogParser) [58] is one such solution which framed the log parsing problem as a Named Entity Recognition (NER) problem. All the tokens present in the logs are labelled with tags, and supervised learning is carried out. The use of such Machine Learning (ML) driven automated solutions removes human intervention and saves time and effort, making the process a lot more efficient and faster.

This thesis is focused on experimenting with a wide variety of machine learning and deep learning methods to solve the problem of information extraction from log files. The primary goal of this work is to improve the generalization capabilities of log parsers so that they do not break when prompted with logs from unseen sources. The remainder of this chapter discusses the motivation and the objective of the research work.

## 1.1 Motivation

Most of the work in the field of log parsing revolves around the use of regular expressions and human-defined grammar or rules. Maintaining a database of regular expressions or rules and regularly updating it is an overhead and is prone to errors. Domain Experts need to be allocated specifically to this task as it requires intense effort and time. We need to make sure after each deployment cycle that all the logging standards are being followed. Logging methods and standards also need to be reviewed in order to protect the log parser from breaking. In some cases, there can be a need to introduce organization-wide standards.

A number of studies have been performed around generating message templates and performing frequent pattern recognition [27, 43, 55, 63]. These approaches work on mining a set of patterns/templates containing frequent words and wildcards ("*"). The wildcards are used to denote placeholders and are positions in the template where not-so-frequent tokens are present. The problem with these approaches is that they cannot be directly used or even extended further to perform information extraction. The wildcard symbol does not help in determining which type of entities might be present.

A relatively small number of studies have been carried out to perform information extraction from log files using Machine Learning and Deep Learning techniques. NER-LogParser makes use of a Bidirectional Long Short Term Memory (BiLSTM) based architecture to carry out NER for log files. There have been recent advancements in the Natural Language Processing (NLP) domain with attention-based systems and related language models. This gives us an opportunity and room for experimentation and research by studying these recent techniques to information extraction from log files.

## 1.2 Objectives

Most of the work with log files is focused on message template generation and is further extended to the problem of anomaly detection. Recent NLP techniques like Transformer [64] and Bidirectional Encoder Representations from Transformers (BERT) [17] have also been used to carry out anomaly detection from log files [25, 26, 38, 73], but there are no studies in the literature supporting information extraction from logs using these architectures. We extend the idea of NER used in [58] to carry out information extraction from log files using recent NLP-based models like transformer neural network, BERT, Robustly Optimized BERT Approach (RoBERTa) and more. Mentioned below are the various goals of our research work:

- Goal 1: To perform an extensive comparison between various ML-based NER systems used in the context of information extraction from log files. We experiment and compare a total of 14 different approaches, including word-based baselines like Naive Bayes, graph-based models like Conditional Random Fields, a pre-trained sequence-to-sequence model for log parsing: NERLogParser, sequence-to-sequence models like transformer, language models like BERT, ensembles and cascading classifiers of these individual approaches.

- Goal 2: To understand the generalization capability of various NER approaches used to extract information from log files. We evaluate the NER approaches for their generalization capabilities to check how well the systems are able to extract information from logs in a format unseen in the training phase. We use two different datasets to carry out the evaluation process: 1) In-Scope data; and 2) Out-of-Scope data. The In-Scope data is used to train-validate-test the various NER approaches. On the other hand, the Out-of-Scope data contains logs originating from unseen sources and is used to test how well the NER systems generalize. The generalization aspect is an important factor to consider to understand whether the system can adapt well to new incoming logs.

- Goal 3: To develop a system that tackles different challenges of information extraction from logs. We propose Log Diversification Unit (LoDU), a component of our system responsible for diversifying and enriching logs. This component carries out log augmentation and helps increase the generalization capabilities of various NER approaches. We also develop the Delimiter Classification Unit, responsible for preprocessing the logs to remove unwanted delimiters, which could become a bottleneck for log parsing systems containing logs with various delimiters.

We employ the framework proposed in [52] to evaluate the NER systems. The evaluation framework uses four different schemes to evaluate the systems resulting in 4 different sets of precision, recall and F1 scores. Each evaluation scheme gives a different perspective of looking at the extraction results. Using the framework, we are able to compare the systems based on their extraction capabilities and are also able to understand the reasons why the system is not able to perform well. This thesis intends to employ the latest advancements in NLP to address the problem of information extraction from log files. This work also revolves around verifying and improving upon the generalization capabilities of log parsing NER systems.

## 1.3 Publications

We have published a paper out of this research:

- [10] Anubhav Chhabra, Paula Branco, Guy-Vincent Jourdan, and Herna L. Viktor. "An Extensive Comparison of Systems for Entity Extraction from Log Files." In International Symposium on Foundations and Practice of Security, pp. 376-392. Springer, Cham, 2022.

We plan to write an additional journal paper and are working on the same.

## 1.4 Organization

The remainder of this thesis is structured as follows. Chapter 2 discusses the related work in the field of log analysis to help the reader understand what advancements are taking place considering the logs. We then discuss NER and how it can be used to tackle information extraction from logs in detail. Chapter 3 explains the complete system design and its various components. We describe in detail how different algorithms are employed to carry out NER in log files. We also look at what techniques can be employed to make the classifier more generalized to new and unseen logs. Chapter 4 presents the experimental design of the system and throws light on the used datasets, hyperparameter optimization and evaluation strategy. The results and analysis are presented in Chapter 5. Finally, Chapter 6 provides the main conclusions and presents improvements and directions for future work of this research.

# Chapter 2

# Related Work

This chapter discusses the relevant work performed in the field of log analysis. It begins by throwing light on the diverse nature of logging standards. We then examine traditional information extraction methods, including regular expression-based extractions. It is followed by a discussion of various studies that suggest approaches to perform message template extraction. We also discuss the recent advancements in NLP and how it has benefited log analysis. The chapter then discusses NER in detail and how it can be framed to tackle log parsing. Finally, we discuss the possible areas of improvement and research gaps that should be addressed.

## 2.1 Diversity in Logging Practices

Logs can be used to cater to multiple purposes like real-time monitoring of systems, anomaly detection, and even calculating statistics related to the health and life of the system. There have been protocols in place to make the process of logging standardized across various use cases. But due to the dissimilar requirements of different use cases, these protocols may become bottlenecks. Hence, these are not followed for all the use cases. One such standardization is the Syslog protocol [23]. It lays out the various rules and formats that provide use case specific information to be plugged in a structured way.

It is essential to log information about the events happening within a system. Various organizations and communities also work on setting up best practices related to logging, which the developers may or may not follow. Below mentioned are some of the common practices while logging information about events.

- Using standard logging libraries to log information about events.
- Logging with proper usage of severity levels of events.
- Using a common language like English for logging across various systems.
- Using a common structured or unstructured standard for logging messages across all the systems.

The standards mentioned above are not always followed and may vary across teams and organizations. For example, some developers may choose to use print statements to log the information about events rather than using logging libraries. Logging generally includes the severity level of messages; INFO, DEBUG, and CRITICAL are some of the severity levels of the logged events. Some developers may choose not to use these levels properly and could log CRITICAL messages at an INFO level. Some developers may want to violate the use of English language while logging and could use their native language to log the information. Another approach that can vary across developers is logging information in a machine-parseable format. Some developers could log information in JSON-like key-value pair format, while others stick to the semi-structured text.

Hence, such a wide variety of practices, personal choices, and no adherence to the practices across teams/organizations lead to a wide diversity in the structure, content and other properties of logs. This makes the information extraction task even more challenging.

## 2.2 Traditional Methods for Information Extraction

Most of the traditional methods for information extraction from log files rely either on regular expressions or human-made rules/grammar. This section will discuss some of these works, how the information extraction is performed, and the typical issues that arise while using these approaches.

Chen et al. [8] worked on finding the correlation of the event logs. They use five sources of logs in their work, including Door Logs, Windows 2000 Security logs, Apache Server Logs, Browser Logs, and SysLogs. The logs from different sources are converted to a canonical representation set out by the authors. The canonical representation consists of 11 fields to be extracted from the event logs. These canonical representations are then stored in a database which is finally used to perform the correlation analysis. The conversion of the raw logs to the canonical representation includes the use of pre-parsers and parsers. The pre-parsers include rules related to the pre-processing of log instances. On the other hand, the parsers are based on Java-based regular expressions. Various sets of regular expressions are developed to tackle the parsing of multiple log files, which help extract the 11 fields to form the canonical representation.

Schatz et al. [51] proposed automated detection of forensic scenarios based on the information gathered from the log files. Three different log sources, including Windows 2000 System logs, Apache Logs and Door Logs, are used to create a knowledge base. Since the unstructured and heterogeneous logs cannot be inserted into the knowledge base, authors use parsers to extract the information and represent logs from all sources in a standardized manner. The parsers used to extract the information are purely based on the use of regular expressions. The structured logs are then pushed to the Jena framework used to create the knowledge base.

Traditional methods perform well at parsing the logs for which the regular expressions and rules are already present in the system. But, as soon as a new log is made incident, the system breaks leading to poor extraction of information. Even a small update in the system can lead the parser to collapse. For systems relying on traditional methods, regular human intervention is needed to maintain and update the regular expressions. Hence, it becomes a costly process in terms of human labour and time. Due to the reasons mentioned above, traditional methods may not be a scalable approach for the rapidly growing distributed and complex systems.

## 2.3 Message Template Generation

Log parsing is an ambiguous term used for different tasks. Many studies employ this term to refer to the template generation process while few others refer to information extraction from log files. Most of the solutions that carry out template generation use unsupervised learning methodologies. This section will discuss the various unsupervised log parsing approaches that already exist and why they are not suitable for our use case.

Vaarandi et al. [63] proposed a novel data clustering algorithm and a tool for mining frequent patterns from log files. The main idea behind the algorithm is based on the frequency of words present in the log files. The author studies that most words in log files are infrequent, whereas a small fraction of terms is frequently present. Using this characteristic of log files, the algorithm performs a density-based clustering. A total of three iterations are performed over the log files. The first pass over the log files mines the frequent words considering their absolute position; the second pass prepares the cluster candidates, including grouping the frequent words to form a template. The final iteration helps select the cluster candidates having an occurrence greater than a user-defined threshold. The final template consists of wildcards ("*"), representing the variables or infrequent words.

The template generation problem can also be solved using hierarchical partitioning of log instances based on their properties. Makanju et al. [43] proposed Iterative Partitioning for Log Mining (IPLoM) , a novel algorithm that performs better than [63] and does not depend on the frequency of words. The algorithm consists of 4 steps, including three iterative partitioning steps and a template generation step. The first partitioning step divides the log instances based on the token count such that the logs with the exact token count are clustered together. It is followed by partitioning each cluster based on the token position. The token position with the least number of unique values is chosen, forming groups based on each unique value. The third partitioning is based on finding the most occurring pair of tokens. The final step is the generation of templates containing wildcards.

Length Matters (LenMa) [55] uses the length of tokens present in the logs to develop a novel online clustering algorithm. It performs a single pass over the logs to group the log instances and generates log message templates in an online fashion. It starts by preparing a word length vector and a word vector for the new incoming log instance. The cosine similarity score is computed between the word length vector and all the clusters with the same word length. If the similarity score is greater than a threshold, then the incoming log is a match for the cluster. Otherwise, a new cluster is formed using the new log instance. The cluster vectors are updated every time a new instance matches the signature. The cluster's word length vector is used to calculate the cosine similarity, whereas the cluster's word vector generates message templates.

Xu et al. [70] performed source code analysis, which included iterating the entire source code to form log message templates. Once the message templates have been built, the real-time logs are matched with every log template to extract the information. The templates contain wildcards/placeholders which can be used to capture the information from the incoming log instance. Many more solutions tackle the log parsing problem by generating templates, including Drain [27] and Spell [20]. Drain makes the use of regular expressions and a fixed depth tree, whereas Spell makes the use of the Longest Common Subsequence approach.

All the message template generation approaches are suited well for tasks like pattern matching, clustering of logs, and even carrying out log anomaly detection through the use of these templates. These approaches generate wildcards which represent placeholders in the templates. Knowing what field the wildcard represents is not handled by these approaches, and hence, these do not hold value in the information extraction domain.

## 2.4 Attention based Log Analysis

There have been recent advancements in the area of NLP with the introduction of attention-based mechanisms such as the Transformer Neural Network [64], BERT [17] and other large language models. These techniques have been widely adopted across various domains but have not been widely studied for log analysis. Only a few studies carry out anomaly detection using BERT as the underlying architecture.

LogBERT [25] is a self-supervised framework that helps predict anomalies present in a log sequence. It uses the underlying principle of BERT but tweaks the objective functions that need to be minimized. LogBERT is pre-trained using two tasks: 1) Mask Language Modeling of log sequences depicting normal behaviour, which helps the model understand the context within a log sequence; 2) Volume of hypersphere minimization, which focuses on bringing the normal log sequences closer, whereas the anomalous logs can far apart in embedding space. Log Anomaly BERT (LAnoBERT) [38] is another study showing how BERT can be used to detect log anomalies. LAnoBERT is different from LogBERT as, rather than considering log sequences, LAnoBERT considers single log instances irrespective of their temporal order. Pre-training of LAnoBERT is carried out using Masked

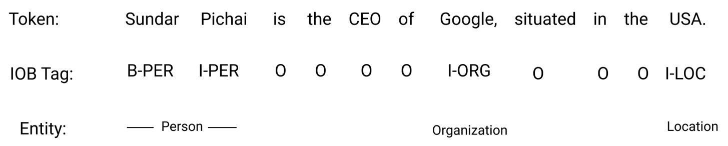
Language Modeling of the normal log instances. Masked tokens are predicted using the pre-trained model, and the predictions' probabilities are calculated. Since LAnoBERT is trained on normal log instances, the confidence of predicting masked tokens in normal cases is high. In contrast, this behaviour is not observed for the logs which haven't been seen during the pre-training phase. Masked tokens present in the anomalous logs are not predicted with high confidence; hence, this property can be used to detect anomalies in the log data.

Current attention-based log analysis studies do not cover the information extraction aspect. There hasn't been any formal work that uses these methods to perform information extraction from log files. However, deep learning techniques have been employed to tackle information extraction in [58].

## 2.5 NER for Log Parsing

NER is the process of extracting named entities from a piece of text [49]. The initial idea of NER was developed to extract named entities like person, organization and location. NER includes annotating sections of texts with pre-defined class labels that we want to identify. Formally framed, NER can be considered a multiclass classification task where the primary goal is to predict which text sections belong to which categories. Over the years, the idea of NER has evolved to solve a wide variety of information extraction and text labelling tasks. It has been actively employed in medical domains [53,68] and has also been extended to languages other than English [4,9].

Since classification is a supervised learning task, we need labelled pieces of text to perform NER. Every token in the text is tagged with a respective label using a standardized process. The most widely accepted labelling process is the Inside-Outside-Beginning (IOB) tagging. IOB tagging includes labelling tokens using three different tags: B-ENT, I-ENT and O, where ENT is the acronym for the named entity. If the named entity spans a single token, it is tagged as I-ENT. On the other hand, if a named entity spans multiple tokens, then the first token is tagged as B-ENT, followed by I-ENT for all the tokens contained in the entity. The O tag is used to label all the tokens that do not belong to any pre-defined categories. Figure 2.1 shows the IOB tagging of a sentence having three named entities.



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Token: | Sundar | Pichai | is | the | CEO | of | Google, | situated | in | the | USA. |
| IOB Tag: | B-PER | I-PER | O | O | O | O | I-ORG | O | O | O | I-LOC |
| Entity: | —— Person | —— | | | | | Organization | | | | Location |

Figure 2.1: Example Depicting IOB tagging

Studies have been carried out to perform NER using traditional machine learning classifiers like Naive Bayes and more [7, 21, 62]. Deep Learning techniques have also been widely studied and employed for applications involving NER. Chiu et al. [13] employed Long Short Term Memory (LSTM) and Convolutional Neural Network (CNN) to perform NER, whereas [36] proposed a combination of BiLSTM and Conditional Random Fields (CRF). Recent advancements in NLP and attention-based mechanisms have enabled the language models to cater to token classification tasks such as NER [17, 42, 50].

The concept of NER can be extended to the log parsing problem. Log parsing is the information extraction process from log instances; similarly, NER is the process of extracting named entities from text. Hence, the overlap in the primitive goal of both the problem statements lets us frame the log parsing problem as a NER problem. Log records contain text sections representing various fields/entities like timestamp, IP address, hostname, service and more. Each of these fields may span one or more tokens in length, allowing the IOB tagging to be carried out. Every token in the log instance must be labelled with its respective IOB tag. Once the data is prepared in such a format, different classifiers can be trained and evaluated.

NERLogParser [58] applies NER to extract information from log files and support digital forensics. NERLogParser is an automatic tool capable of parsing logs from various sources. It comes bundled as a pre-trained model in a python package. The log parser is pre-trained on a dataset containing over 300,000 log instances spanning eleven different sources. The NER problem is further framed as a sequence-to-sequence learning task, which aims to convert an input sequence to an output sequence. In the context of log parsing, the input sequence is the sequence of tokens present in the logs, whereas the output sequence contains the respective IOB tags. NERLogParser's underlying architecture relies on BiLSTM. BiLSTM [24] consists of two different LSTMs, one in the forward direction (left-to-right) and the other in the backward order (right-to-left), giving the architecture the ability to learn the context of words from both the directions.

Since words cannot be directly fed to the BiLSTM architecture, authors use the Global Vectors (GloVe) word embeddings to map words to vectors. The GloVe word embedding [46] is similar to a lookup dictionary which contains numerical vectors for different words. The idea of word embeddings reflects that words with similar meanings should be close in their vector space while dissimilar words should be far apart. In this use case, NERLogParser makes the use of glove.6B embedding with a size of 300 dimensions. NERLogParser also uses character-level embeddings to deal with unknown tokens as used in [36]. The word embedding and the character-level embedding of the log token are concatenated, and the resultant vector is fed to the BiLSTM network for training. Figure 2.2 shows the underlying architecture of NERLogParser. The authors divided the log data into a 60-20-20 split for training, validation and testing sets. To carry out evaluation on the testing set, the authors use precision, recall, F1 score and accuracy to compare NERLogParser with other traditional machine learning approaches where NERLogParser outperforms the other methodologies.

## 2.6 Current Gaps in Information Extraction from Logs

Recent NLP advancements, including neural machine translation [3, 64] and language models [17, 42, 50], have impacted a wide variety of applications. Studies have improved the architectures in terms of performance metrics, training times and even inference times on benchmarking datasets. The introduction of the attention mechanism in [3] helps unfold how different tokens rely on each other within a sentence. The transformer neural network [64] removes the use of recurrent units and relies entirely on the attention mechanism resulting in faster training times. These architectures frame the neural machine translation as a sequence-to-sequence problem and improve over the previously existing state-of-the-art methodologies. NERLogParser uses the BiLSTM architecture to frame the NER problem as a sequence-to-sequence learning problem. The attention-based methodologies discussed above show state-of-the-art performance on the machine translation tasks, but their per-

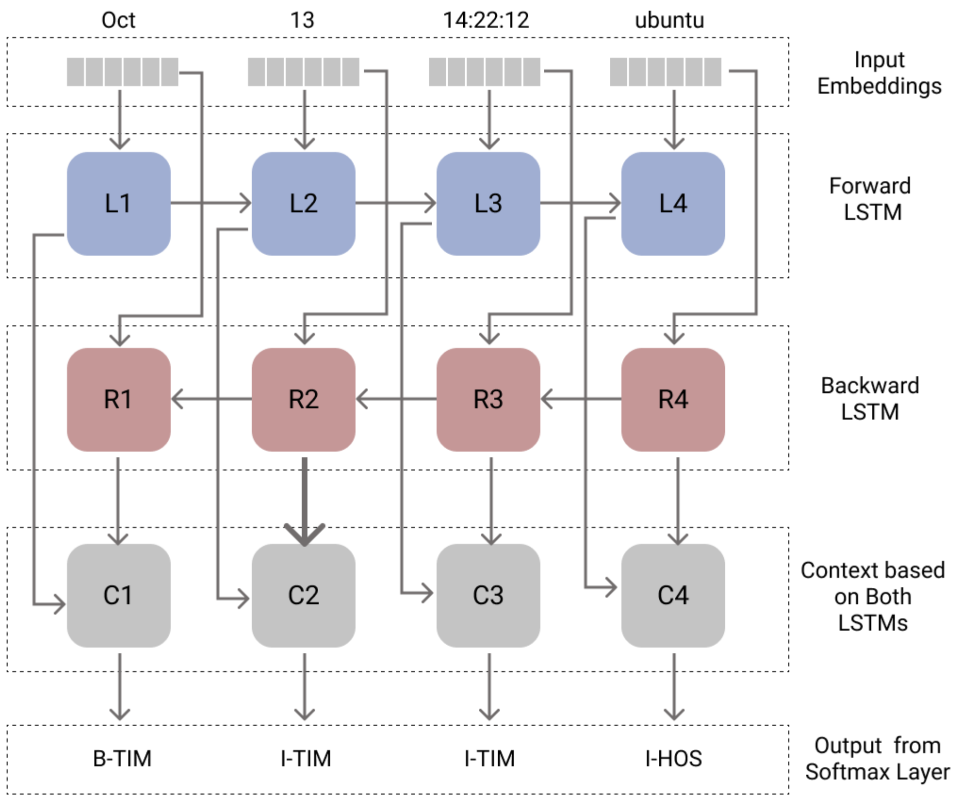


Figure 2.2: Architecture of NERLogParser

formance efficiency has not been evaluated in our application's context. Hence, there is a possible gap for further research in this area. Another area which has not been explored is the application of language models to extract information from log files. The advancements in language models let us carry out multiple tasks, including NER. No formal study applies or evaluates the language models for information extraction, allowing us to extend this idea further.

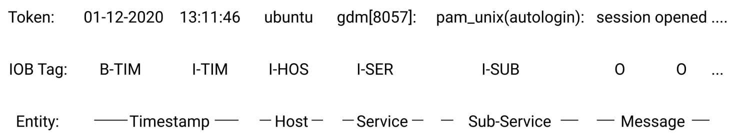
NERLogParser compares its performance with standard machine learning methods, including Naive Bayes and Perceptron, to name a few. CRF [65] classifier is a methodology that has been used to carry out NER, but the authors do not consider it while performing the comparison. A clear research gap exists in conducting a systematic and detailed comparison study evaluating multiple non-traditional classifiers. We employ various methodologies and perform a detailed comparison between NER approaches. Another motivation behind this work is to evaluate the generalizability of a log parser. To the best of our knowledge, no study in the literature evaluates the methodologies in this aspect. The primary goal of our work is to develop a system capable of parsing the logs originating from new and unseen sources. We discuss the system development and evaluation in Chapters 3 and 4, respectively.

# Chapter 3

# System Design

## 3.1 System Architecture

The problem of log parsing can be addressed using supervised machine learning as our goal is to classify a section of text belonging to a particular class. We frame the log parsing problem as a NER problem. To carry out NER, we need to have the data in a specific IOB tagged format as used in [49] and need to train and test machine learning models using the IOB-tagged logs. Figure 3.1 depicts and example of IOB tagging for a log instance.



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Token: | 01-12-2020 | 13:11:46 | ubuntu | gdm[8057]: | pam_unix(autologin): | session | opened | .... |
| IOB Tag: | B-TIM | I-TIM | I-HOS | I-SER | I-SUB | O | O | ... |
| Entity: | —— Timestamp —— | | — Host — | — Service — | — Sub-Service — | — Message — | | |

Figure 3.1: Example Depicting IOB tagging for logs.

The systems discussed in the previous chapter based on regex, grammars [8, 51] and other ML approaches [58] do not consider the wide variety and constant evolution of logs. Such systems could break when there is even a minute difference in the log format. We carry out various experiments to support this statement in section 5.5. Our primary goal in this work is to develop an end-to-end system capable of performing well with logs from previously seen sources and logs from unseen sources.

The information extraction system that we propose is divided into a total of 4 units. These are: 1) Log Diversification Unit (LoDU), 2) Log Parsing Unit (LoPU), 3) Delimiter Classification Unit, and 4) IOB Tag Stitching Unit. Figure 3.2 reflects the complete end-to-end flow of the system containing all the mentioned units. These units are completely decoupled and can be used as stand-alone units without dependencies. These units include different modules responsible for carrying out different operations on the log data.

Since our primary focus is on performing well on logs from new and unseen logs, we introduce LoDU. The raw IOB-tagged logs are firstly passed through LoDU, which is responsible for enriching the logs and adding randomness/diversity to the logs. The enriched and diversified logs are then used to train the machine learning models present in LoPU. LoPU consists of diverse models including traditional word-based baselines such as Naive Bayes, graph based models like CRF [65], sequence-to-sequence model: Transformer Neural Network [64], language models including BERT [17], ensembles and complex cascades of many models. NERLogParser is used as a pre-trained log parsing tool and is directly used as a log parser and evaluation is performed.

At the time of inference, the logs are first passed through the Delimiter Classification Unit, which converts logs into space-separated tokens and eventually fed to the LoPU. The trained models are then used to predict IOB tags for the testing logs, which are then fed to the IOB tag stitching unit to obtain the entities. We provide more details on these four key units in the upcoming sections.

## 3.2 Log Diversification Unit (LoDU)

Our goal is to design a system capable of performing well not only on logs from seen sources but also on logs that originate from a set of new and unseen sources. Building Machine Learning models on a dataset that is just constrained to a few sources could lead to overfitting [18] and would result in models not generalizing well to logs from unknown sources. Data Augmentation techniques can help boost the performance of text classification systems [67]. LoDU performs data augmentation by adding new synthetic logs to the existing log dataset and providing randomness and diversity. LoDU takes the raw logs and outputs augmented logs that are fed to the Log Parsing Unit to train different machine learning models. LoDU consists of two modules: 1) Entity Enrichment Module and 2) Log Pre-processing Module. We will discuss these modules in detail in the upcoming subsections.

### 3.2.1 Entity Enrichment Module

The first component of LoDU is the Entity Enrichment Module. The Entity Enrichment Module takes care of adding diversity to the logs by introducing/replacing entities within the logs. The enriched logs help achieve a training set that could be representative for

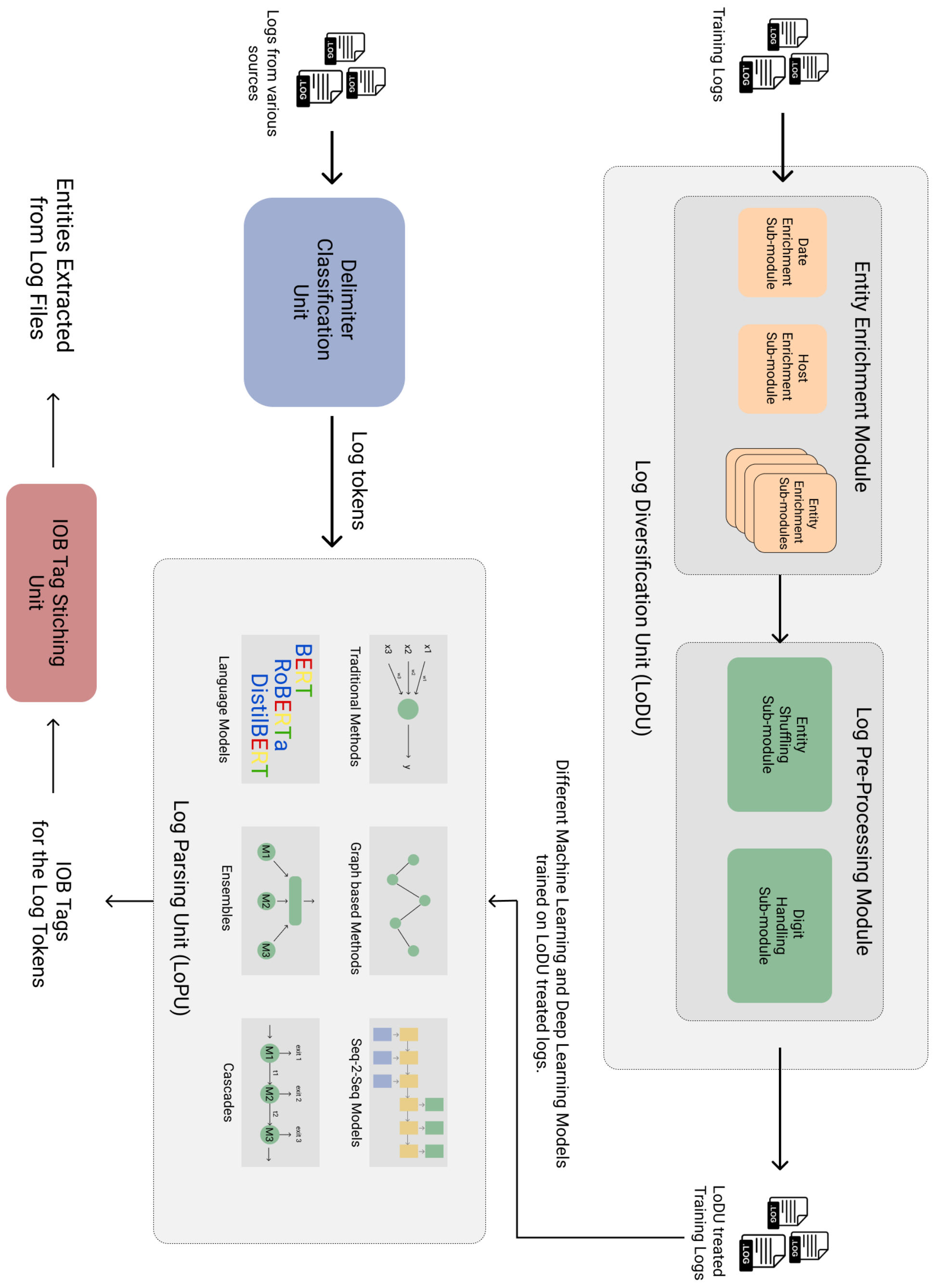

Training Logs
LOG
Log Diversification Unit (LoDU)
Entity Enrichment Module
Date Enrichment Sub-module
Host Enrichment Sub-module
Entity Enrichment Sub-modules
Log Pre-Processing Module
Entity Shuffling Sub-module
Digit Handling Sub-module
LoDU treated Training Logs
Different Machine Learning and Deep Learning Models trained on LoDU treated logs.
Logs from various sources
Delimiter Classification Unit
Log tokens
Log Parsing Unit (LoPU)
Traditional Methods
Graph based Methods
Seq-2-Seq Models
Language Models
BERT
RoBERTa
DistilBERT
Ensembles
Cascades
M1
M2
M3
exit 1
exit 2
exit 3
x1
x2
x3
y
IOB Tags for the Log Tokens
IOB Tag Stiching Unit
Entities Extracted from Log Files

Figure 3.2: System Architecture

new logs and could help avoid overfitting [72]. Figure 3.3 depicts the Entity Enrichment Module, which consists of two different ways of entity enrichment as listed below.

- Entity Gazetteer: The first way to achieve entity enrichment is to define an entity gazetteer. Entity gazetteer is a pre-defined list of entity values that is used as a replacement for the entity values present in the training logs. An example of such a list is a month gazetteer that can be used to diversify the logs which just contain timestamp having a single month.
- New Entity Generation Logic: Another way to add/replace entity values could be to dynamically generate values rather than using a fixed list of values. This approach could be suitable for cases where the entities could have a huge number of possible values. For example, entities like IP addresses could be dynamically generated using stored procedures.

The same concept can be applied to enrich any number of entities to achieve diversity. We apply the entity enrichment module to two different entities and explain them in the following sub-sections.

#### 3.2.1.1 Timestamp Enrichment Sub-module

Logs originate from a variety of sources and include information relating to events that take place within a system, application or device. Depending upon the source and the events taking place, the entities within a log change. These entities can be but are not limited to hostname, IP address, timestamp, and more. Generally, timestamp is the only

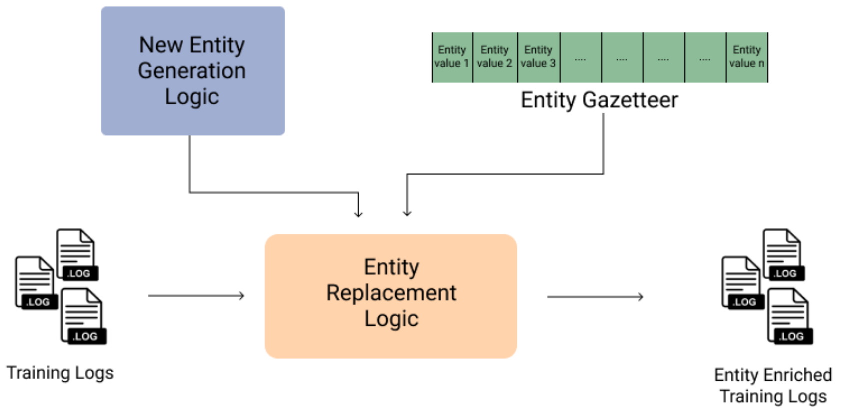


Figure 3.3: Entity Enrichment Module

common entity present in the logs, irrespective of source and event. Though it is common across logs from different sources, it is heterogeneous in nature [16] and varies a lot in terms of format and when the information has been logged. There are several timestamp formats depending upon the precision of time that we want to record. Some may contain a 12-hour time format, while some could be following a 24-hour time format. Some formats could contain months in numeric form, while others could have a string representation. A few formats could contain granularity of up to seconds; on the other hand, a few do not represent the seconds. There could even be formats where the year is not present; just the month and day are enough to represent the timestamp of the event. All of the formats are use-case dependent and are generally in the hands of System Administrators, DevOps Engineers and developers. Hence, the timestamp format could vary across organizations, departments and even across development teams.

Since there are such diverse timestamps present in the logs, it can be challenging to intelligently predict them, given the fact that the logs are constrained to a few sources and are collected over a short period. One way to tackle this situation is to collect the logs from many sources and over a long period (so that the timestamp includes all the months/days and a few years). A better and quicker way to tackle this challenge is by diversifying and enriching the timestamps present in our training data. This way, we achieve a training set with enough diversification such that the models understand the concept of the timestamp in general rather than using an extensive dataset with all the possible permutations and combinations of timestamps. The diversity in timestamp can be introduced in two different ways: 1) by introducing new and unseen timestamp formats into the data; and/or 2) by introducing new and unseen timestamp values into the data.

Table 3.1: Timestamp Formats present in the Timestamp Enrichment Sub-module

| **Time format** | **Year Present** | **Timestamp Format** |
|---|---|---|
| 24 hour format | Yes | <d B Y H:M:S>, <d b Y H:M:S>, <d-B-Y H:M:S>, <d-b-Y H:M:S>, <d.m.Y H:M:S>, <d/m/Y H:M:S>, <d-m-Y H:M:S>, <m.d.Y H:M:S>, <m/d/Y H:M:S>, <m-d-Y H:M:S>, <Y.m.d H:M:S>, <Y/m/d H:M:S>, <Y-m-d H:M:S>, <d/B/Y H:M:S>, <d/b/Y H:M:S> |
| 24 hour format | No | <d B Y I:M:S p>, <d b Y I:M:S p>, <d-B-Y I:M:S p>, <d-b-Y I:M:S p>, <d.m.Y I:M:S p>, <d/m/Y I:M:S p>, <d-m-Y I:M:S p>, <m.d.Y I:M:S p>, <m/d/Y I:M:S p>, <m-d-Y I:M:S p>, <Y.m.d I:M:S p>, <Y/m/d I:M:S p>, <Y-m-d I:M:S p>, <d/B/Y I:M:S p>, <d/b/Y I:M:S p> |
| 12 hour format | Yes | <d B H:M:S>, <d b H:M:S>, <d-B H:M:S>, <d-b H:M:S>, <d.m H:M:S>, <d/m H:M:S>, <d-m H:M:S>, <m.d H:M:S>, <m/d H:M:S>, <m-d H:M:S>, <d/B H:M:S>, <d/b H:M:S> |
| 12 hour format | No | <d B I:M:S p>, <d b I:M:S p>, <d-B I:M:S p>, <d-b I:M:S p>, <d.m I:M:S p>, <d/m I:M:S p>, <d-m I:M:S p>, <m.d I:M:S p>, <m/d I:M:S p>, <m-d I:M:S p>, <d/B I:M:S p>, <d/b I:M:S p> |

The entire set of training logs is passed through the timestamp enrichment sub-module to obtain a timestamp-enriched version of the dataset with a total of 54 timestamp formats. Table 3.1 depicts all the timestamp formats produced by the timestamp enrichment sub-module.

#### 3.2.1.2 Hostname Enrichment Sub-module

Hostname is another common entity found in logs of applications/devices that operate over a network. It is generally an alphanumeric string uniquely identifying a computer system within a local network. The alphanumeric nature of the hostname entity makes the prediction challenging as the alphanumeric list is inexhaustible. Just a few hostnames in the training dataset can lead to over-fitting [72] as the prediction models could memorize the hostname string literals. Adding diversity to the hostnames is thus essential as it could make the models generalizable about predicting hostnames.

A preliminary analysis is run on the training logs to find out the diversity in the hostnames. A total of 4 hostnames are found in the logs: ubuntu, ps3, nssal-ps3, victoria. Such few hostnames can lead to overfitting, thus there is a need to diversify the hostnames as well. To carry out the enrichment process for hostnames, we use a hostname gazetteer containing one million hostnames and randomly replace the four hostnames with random hostnames present in the gazetteer. The resultant enriched dataset contains over 65K hostnames which could avoid over-fitting.

### 3.2.2 Log Pre-processing Module

The Log Pre-Processing Module encapsulates the Entity Shuffling and Digit Handling Sub-modules. These sub-modules are responsible for introducing diversification and avoiding overfitting. The upcoming subsubsections discuss these sub-modules in detail.

#### 3.2.2.1 Entity Shuffling Sub-module

Logs from similar sources may not always have the same order of entities. For example, the logs of a server may have an IP address present after the request timestamp, whereas a different server may have an IP address prior to the timestamp. This change in order is evident because of the different configurations that the logs can possess. Hence, the sequential order of entities in a log is prone to change because of human intervention.

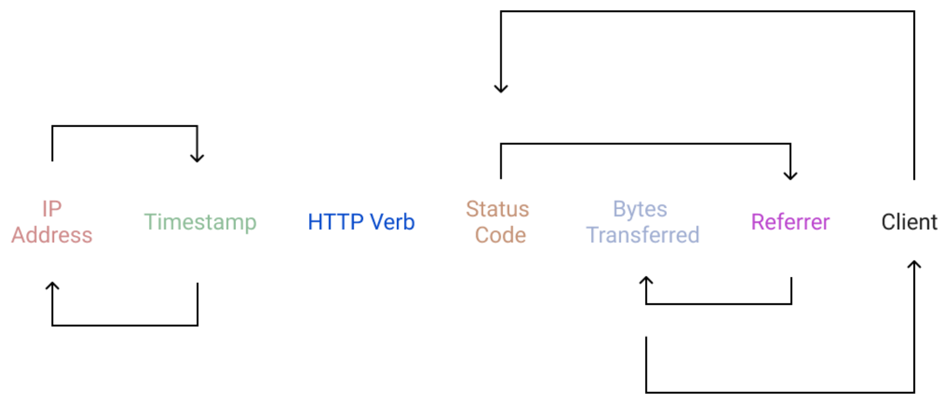


Figure 3.4: Example depicting Entity Shuffling

Building Machine Learning models on a fixed set of sequences for a single source may lead to overfitting. Such models would not be able to generalize well when prompted with unseen logs from various sources. Adding diversification to the logs in terms of the sequence of entities can help us generalize better. As shown in Figure 3.4, the Entity Shuffling Sub-module shuffles the order of the entities present in the log, providing a more rich set of logs in terms of sequential order. The shuffling adds another level of diversification and randomness to the logs by introducing modified logs with different entity sequences.

#### 3.2.2.2 Digit Handling Sub-module

Many entities present in a log can be purely numeric like date, year, log level and number of bytes transferred, to name a few. Entities like date and log level possess a fixed number of values, but this is not the case with all the numeric entities present in the log. Some numeric entities may possess an inexhaustible list of values that could make the learning process difficult. Machine Learning algorithms work with tokens/textual data by maintaining a set of training vocabulary. Since the list of possible values that the numeric entities can possess is inexhaustible, the size of the vocabulary could increase enormously which hinders the learning process. To tackle this situation every digit present in the log is converted to a special <num> token. Figure 3.5 shows an example of a pre-processed log.

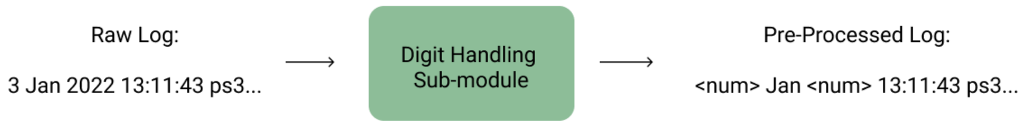

Raw Log:
3 Jan 2022 13:11:43 ps3...
Digit Handling Sub-module
Pre-Processed Log:
<num> Jan <num> 13:11:43 ps3...

Figure 3.5: Digit Handling Sub-Module

## 3.3 Log Parsing Unit

Once the logs have been augmented and enriched using LoDU, these are passed to the Log Parsing Unit (LoPU). LoPU is the core component of the system as it takes care of all the models' training and serves the log parsing requests during the inference phase. LoPU consists of various models, including word-based baselines: Naive Bayes [47], Perceptron [48], and Stochastic Gradient Descent (SGD) [6], graph-based model: Conditional Random Fields [65], sequence-to-sequence model: Transformer Neural Network [64], neural language models like BERT [17], RoBERTa [42] and DistilBERT [50], ensembles of the standalone models and even more complex models like cascading classifiers. We experiment with a total of 14 models, all of which treat the log parsing problem as a NER problem. Since models vary a lot in terms of perspective and fundamentals, the input to these models could differ too. Each model present in LoPU has its pre-processor that converts the log data to a model-friendly format, and this is used to carry out the training process.

The presence of multiple models in LoPU gives users and domain experts the ability to switch between models and use them on a use-case basis. For example, ensemble-based classifiers can have a huge memory footprint and may even take time during the inference phase. Because of this reason, ensembles may not be a good fit for real-time use-cases. Such models are not a good fit for environments that demand less computing and memory usage. On the other hand, ensembles might be the best option if a use case allows such bandwidth in terms of time and memory. Word-based classifiers like Naive Bayes may not possess high performance. Still, they are included in the LoPU because the traditional word-based methodologies can be considered a decent baseline to compare other classifiers/approaches. Further, this section discusses the traditional word-based methods followed by other complex NER approaches.

### 3.3.1 Traditional Methods

Traditional methods are the word-based or token-based methods used as baselines in our work. These methods are purely based on the vocabulary (set of tokens) seen during the training phase. Based on the knowledge learnt from the vocabulary, the inference is carried out. The token-based classifiers do not take context into account, which means the presence of neighbouring words does not impact the prediction of a particular token, and hence these classifiers are purely based on the training vocabulary. Many standard machine learning algorithms could be used as baselines, but we decided to choose Naive Bayes [47], Stochastic Gradient Descent [6], and Perceptron [48]. These classifiers are trained on a token-tag mapping, as shown in Table 3.2.

Table 3.2: Token to Tag Mapping

| **Token** | **Tag** |
|---|---|
| 01-12-2020 | B-TIM |
| 13:11:46 | I-TIM |
| ubuntu | I-HOS |
| gdm[8057]: | I-SER |
| pam_unix(autologin): | I-SUB |
| session | O |
| opened | O |

All the training logs are traversed and the token-tag pairs are extracted. As seen in the Table, tokens are the space-separated words present in the log instance, and the tag represents the corresponding IOB tag. The NER task for the baseline methods is simply a multiclass classification problem where the algorithms have to pick a tag for each token present. The tokens and tags possess a one-to-many relationship as the same token in the log can belong to multiple entities and hence have different tags; this can be seen in Table 3.3. The token "May" has 3 different tag mappings: B-TIM. I-TIM and O, this one-to-many mapping could make the learning process harder as there is no other factor to determine the entity to which the token belongs. The token-tag mapping is converted to a one-hot-encoded form, resulting in a sparse matrix. This sparse matrix is finally used to train the token-based baseline classifiers.

Table 3.3: One-to-Many relationship between the token and tags

| **Log Instance** | **IOB tags** | **Mapping** |
|---|---|---|
| May 14, 2021 | B-TIM I-TIM I-TIM | May:B-TIM |
| 14 May 2021 | B-TIM I-TIM I-TIM | May:I-TIM |
| Session closing for user May | O O O O O | May:O |

### 3.3.2 Graph based Methods

Conditional Random Fields (CRF) [65] is an undirected graphical model that finds its significance in the information extraction domain. CRF is a heavily used model in the field of NLP to tackle sequence-related tasks such as sequence labelling problems. It has been utilized in medical applications, including Biomedical NER, Disorder NER, and Disease NER, to name a few [40, 53, 68]. CRF has also been used to carry out NER in languages other than English [4,9]. In its primary sense, NER for logs is a sequence labelling problem where the input is a sequence of log tokens, and we need to label each token with an IOB tag. The sequential and ordered nature of entities within a log makes CRF a sound fit to drive NER. Unlike the traditional baseline methods, CRF takes context into account by looking into the neighbouring words and IOB tags, making it a better fit for predicting sequences.

CRF defines the conditional probability of an output sequence $Z_{1:n}$ given an input sequence $X_{1:n}$, as shown in Equation 3.1.

$$p(Z_{1:n}|X_{1:n}) = \frac{1}{K} exp(\sum_{n=1}^{N}\sum_{i=1}^{F} \lambda_i f_i(Z_{n-1}, Z_n, X_{1:n}, n)) \tag{3.1}$$

where $K = \sum_{Z_{1:n}} exp(\sum_{n=1}^{N}\sum_{i=1}^{F} \lambda_i f_i(Z_{n-1}, Z_n, X_{1:n}, n))$ is a normalization factor which is used to bring the value of probability between 0 and 1, $f_i$ are feature functions and $\lambda_i$ are the weights assigned to the feature functions, $F$ is the total number of feature functions considered, $N$ is the total number of tokens present in a log instance, and $n$ is the current index being considered. In the equation above, we observe that for each token index $n$, a weighted summation of all the feature functions $f_i$ is performed using weights $\lambda_i$. This is repeated for all the $N$ indices and then finally divided by a normalization factor $K$ which brings down the probability value between 0 and 1. This process is continued for all the log sequences present in the training dataset.

The whole idea of CRF revolves around feature functions. Feature function $f_i$ is a function that takes in input sequence tokens, word-level features for these tokens, current index of consideration and the IOB tags; and returns a real value. All of these functions carry weights $\lambda_i$ that are to be learnt at the time of training. Table 3.4 depict the various feature functions used to train the CRF classifier. The feature functions depend on the use case and may change depending upon the problem being solved. Different properties of logs and characteristics of various entities were studied and analysed to come up with the final list of feature functions. To take context into account, for an index $i$, feature functions of the $i^{th}$, $(i+1)^{th}$, and $(i-1)^{th}$ are computed and used in training. These features are engineered by taking into account the characteristics of logs, for example, not all the words in logs are part of the English vocabulary. Similarly, entities like IP address are of fixed length and contain periods in between. Such characteristics are used to engineer approximately 17 feature functions.

During the training phase, we compute the conditional probability for all the sequences present in the training dataset and try to maximize the conditional likelihood objective function using gradient descent. This way, we find the optimal set of learnable weights $\lambda_i$ and each feature function is assigned a weight/importance. Equation 3.2 depicts the conditional likelihood computed for $m$ sequences present in the training dataset.

$$\sum_{j=1}^{m} log(p(z^j|x^j)) \tag{3.2}$$

where $z^j$ and $x^j$ depict the $j^{th}$ output and input sequence respectively.

| **Word-Based Functions** | **Return Type** | **Description** |
|---|---|---|
| is_pure_digit | Boolean | 1 if the token is pure digit else 0. |
| is_pure_alpha | Boolean | 1 if the token is pure alphabetic else 0. |
| is_alphanumeric | Boolean | 1 if the token is alphanumeric else 0. |
| contains_slash | Boolean | 1 if the token contains slash else 0. |
| contains_period | Boolean | 1 is the token contains period else 0. |
| num_slash | Numeric | The number of slashes present in the token. |
| num_period | Numeric | The number of periods present in the token. |
| length_of_word | Numeric | Total number of chars in the token. |
| is_english_word | Boolean | 1 if the token is present in English Dictionary else 0. |
| is_upper | Boolean | 1 if the token is uppercase else 0. |
| is_title | Boolean | 1 if the token is titled. |
| contains_num | Boolean | 1 if the token contains number else 0. |
| contains_symbols | Boolean | 1 if the token contains any symbols else 0. |
| word.lower | String | The lower-cased token. |
| word_suffix | String | The suffix of the token. |
| is_eos | Boolean | 1 if the token is end of the sentence else 0. |
| is_bos | Boolean | 1 if the token is beginning of sentence. |

Table 3.4: Feature Functions used to build CRF

### 3.3.3 Sequence to Sequence Models

The problem of log parsing can also be framed as a sequence-to-sequence task. A sequence-to-sequence model involves mapping a variable-length input sequence to a variable-length output sequence. For a log parsing use case, log tokens can be considered an input sequence, and the related IOB tags can be regarded as the output sequence. A model can be trained using a dataset present in such a format. Depending upon the length of the input and output sequences, several architectures have been proposed to solve the sequence-to-sequence problem, and Figure 3.6 depicts two such architectures. The first architecture represents an encoder-decoder model in which the length of the input and output sequences may vary. In contrast, the second architecture can be employed in use cases where the lengths of both sequences match.

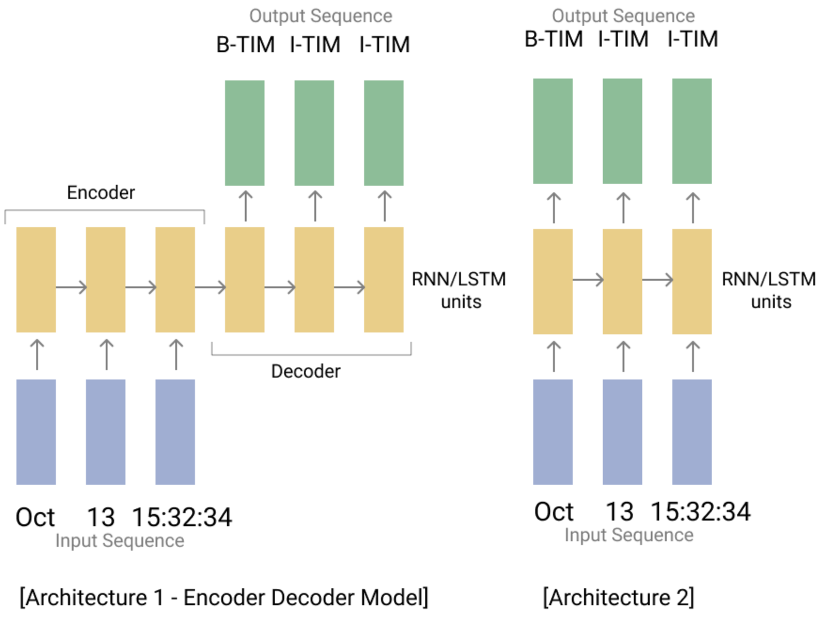


Figure 3.6: NER Sequence to Sequence Models

The encoder-decoder architecture has been widely used in NLP tasks and has shown remarkable results in the field of machine translation. Both Recurrent Neural Network (RNN) [61] and Long Short Term Memory (LSTM) [29] have been used to solve the problem of machine translation [14,60]. Irrespective of the recurrent units, the encoder is responsible for reading the input sequence one token at a time and encoding information into a fixed-length context vector that is passed to the decoder block, which outputs the tokens of the output sequence one step at a time. This architecture faces difficulties for use-cases involving long-length sequences as the fixed-length context vector cannot carry enough knowledge to decode long-length sequences [3].

Bahdanau et. al [3] introduced the notion of attention, which made decoding long-length sequences easier. The authors presented an architecture that disregards using a single fixed-length context vector and instead passes a set of attention weights to the RNN/LSTM unit at each time step. The fixed-length context vector carries a hidden representation of the entire sentence used for decoding sentences. In contrast, the notion of attention helped the model search for input sequence segments relevant to the target word being predicted.

Another solution in the sequence-to-sequence domain is the transformer neural network [64]. The transformer neural network purely relies on the attention mechanism and eliminates the need for any recurrent units like RNN or LSTM. The elimination of recurrent units leads to faster training times as the input sequences can be fed all at once [64]. The transformer neural network has an encoder-decoder architecture as shown in Figure 3.7. We discuss about the architecture in detail below.

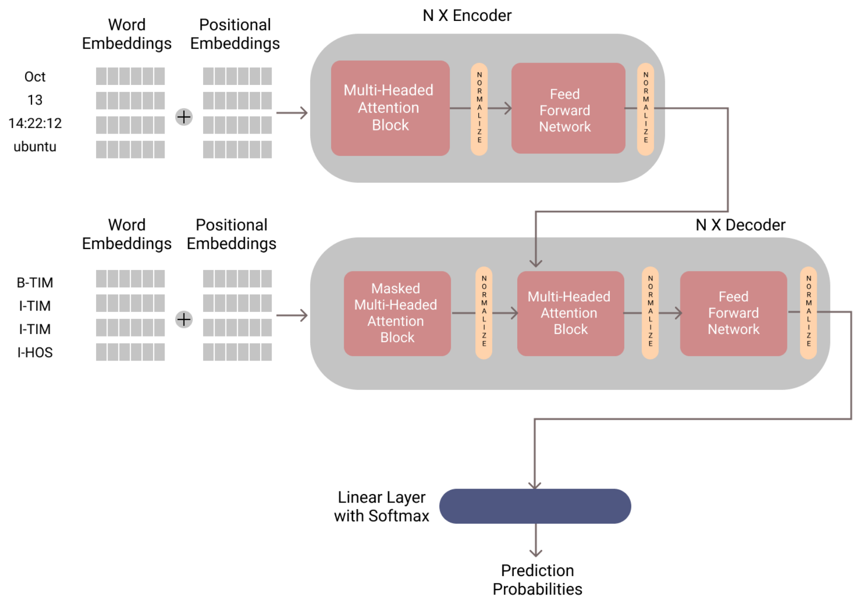


Figure 3.7: Transformer Neural Network

**Encoder**: The encoder consists of two main components: a multi-headed attention block and a feed forward network. The word and positional embeddings for all the tokens in the log sequence are calculated and are fed to the multi-headed attention block. Positional embeddings are used to educate the transformer about the relative positions of tokens within a sequence. The multi-headed attention block is responsible for computing self-attention, a set of weights which carry information about how different words in a sequence are contextually related. Once the self-attention is computed, the normalized output is passed through a feed-forward network and the output is then again normalized. The normalization process helps transformer reduce the training time and also ensure that there is no weight explosion taking place.

**Decoder**: The decoder consists of a total of 3 primary components: 1) Masked multi-headed attention block; 2) Multi-headed attention block, and 3) Feedforward network. Both the attention blocks operate entirely differently; the masked multi-headed attention block computes the self-attention for the target sequence. Masking is done to achieve the training in parallel and avoid cheating at the time of training. All the blocks are connected one after the other with a normalization layer in between. The normalized output from masked multi-headed attention and the encoder is fed to the multi-headed attention block. This block calculates attention weights which carries information about which segment of the input sequence should be focused on while predicting a particular target word. The normalized attention weights are then fed to the feed-forward network. The normalized output from the decoder is fed to a linear layer with a softmax activation function which provides us with the target token probabilities.

### 3.3.4 Language Models

Language models are a vital part of any NLP pipeline. These models help us determine the probability of a sequence of words. This probability measure carries information about the context of the words in a sequence. Language models are generally trained over a large corpus of text and learn about the arrangements and context from the data itself. We can model language either by using: 1) Statistical Language Models; or 2) Neural Language Models. In this subsection, we will discuss three different neural language models to see how they can be used to tackle the problem of log parsing.

#### 3.3.4.1 BERT

Bidirectional Encoder Representations from Transformers (BERT) [17] is a language model based on the transformer neural network [64]. The transformer neural network consists of an encoder and a decoder. BERT is developed by stacking multiple encoders, one over the other. The output of one encoder is fed into the input of the consecutive encoders, and the final output is made incident to a fully-connected layer. Figure 3.8 depicts the stacking of numerous encoders. There are two different variants of BERT depending upon the number of encoders being stacked together: BERT Base and BERT Large. The base variant uses 12 encoders, whereas the larger variant uses 24. BERT has been widely used in solving various NLP problems and has shown state-of-the-art results for 11 different tasks, including Sentence Classification, Sentence Pair Classification, Question Answering, Sentence Tagging and more.

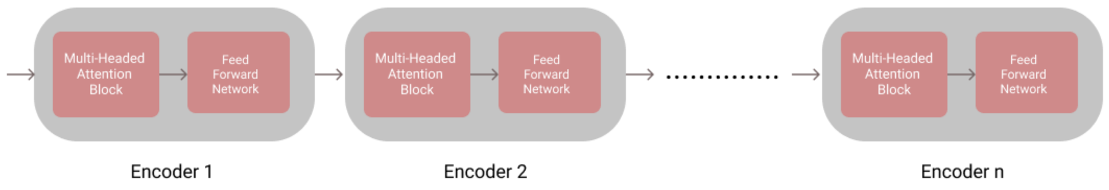


Figure 3.8: Stacking Multiple Encoders to form BERT

BERT training is carried out in two phases: 1) Pre-training; and 2) Fine-tuning . Next we describe these phases in detail.

1. **Pre-training Phase**: BERT is pre-trained on the complete Wikipedia English and BookCorpus data, having a total vocabulary of over 3,000M words. BERT learns language by pre-training on two different unsupervised tasks: 1) Masked Language Modelling (MLM) and 2) Next Sentence Prediction (NSP). In MLM, 15% of the tokens in a sentence are masked/hidden, and BERT learns to predict these masked tokens using the context from nearby words. So basically, in MLM, we are teaching language to BERT using a "Fill in the Blanks" strategy. The NSP task includes predicting whether a sentence can follow another sentence or not. Both these tasks help BERT learn about context; MLM helps BERT learn the intra-sentence context, whereas NSP helps it understand the inter-sentence context. Pre-training of BERT includes minimizing the aggregated loss from both the mentioned tasks.

2. **Fine-tuning Phase**: Once the pre-training of BERT is completed, it is ready to be used for several NLP problems. The basic idea behind fine-tuning BERT is to connect a dense layer at the output end and train it with the dataset of interest. The dense layer head connected to BERT varies depending upon the task being solved. Different NLP tasks require different outputs orientation, and accordingly, the heads can be changed. At a higher level, we can say that BERT tries to understand language in the pre-training phase while it adapts to the problem statement in the fine-tuning stage.

#### 3.3.4.2 RoBERTa

Robustly Optimized BERT Approach (RoBERTa) [42] is a variant of BERT developed at Facebook, which uses a different and improved pre-training methodology. It is trained on enormous amounts of text compared to BERT, using more computational power. The training methodology of RoBERTa is different in two ways:

1. RoBERTa completely removes the NSP task used for pre-training.

2. RoBERTa makes use of dynamic masking in the MLM task. In MLM, a set of tokens are masked, and the model is asked to predict these tokens. BERT uses static masking, in which the same tokens are masked after each epoch, whereas, RoBERTa uses dynamic masking, which hides different tokens after each epoch.

#### 3.3.4.3 DistilBERT

Both BERT and RoBERTa are huge in size, making them computationally expensive and have significant inference times. These models may not be the preferred choice in real-time applications because of memory footprints and time-taken. On the other hand, language models which are equally performant and smaller in size could be an excellent fit for such applications. One of such language models is the DistilBERT [50]. DistilBERT, as the name suggests, is a distilled version of BERT. It considers the BERT-base architecture and trims some of the layers from it. The resultant architecture is a faster and smaller variant without much degradation in the model's performance. DistilBERT is formed using the process of knowledge distillation [28] used to compress the size of a neural network using a teacher-student architecture. Table 3.5 depicts a comparison between the language models based of different measures.

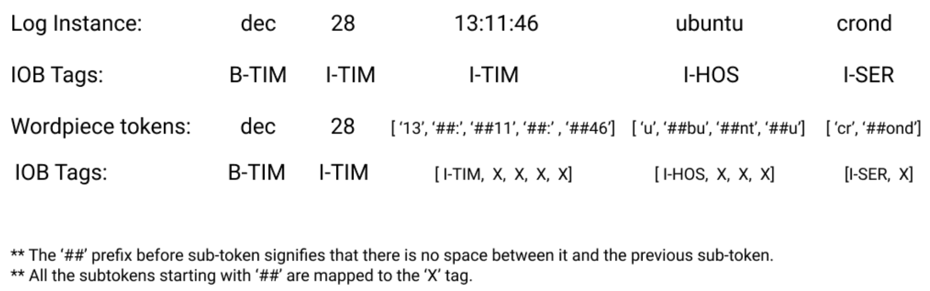
| **Model** | **No. of Weights** | **Data** | **Methodology** |
|---|---|---|---|
| BERT-Base | 110 million | 16 GB (Wikipedia + Book Corpus) | MLM and NSP |
| BERT-Large | 340 million | 16 GB (Wikipedia + Book Corpus) | MLM and NSP |
| RoBERTa-Base | 110 million | 160 GB (BERT data + additional) | Dynamic MLM |
| RoBERTa-Large | 340 million | 160 GB (BERT data + additional) | Dynamic MLM |
| DistilBERT | 66 million | 16 GB (BERT data) | MLM, NSP using Knowledge Distillation |

Table 3.5: Comparison of different Language Models

For our use case, we employ the language models for a sentence tagging task in the context of logs. Each log can be considered a sequence of tokens, and each token has its corresponding IOB tag. Log sequences are pre-processed using a tokenizer(different for each language model) to make the data compatible with the language model, resulting in a subword-tokenized format. Figure 3.9 shows the BERT input in a subword-tokenized form. The token's IOB tag is mapped to the first subword, while the remaining subwords are mapped to the tag "X." Once the subword tokens are prepared, the subword tokens and the IOB tags are converted to integer-based indices and are padded to a fixed length.

The input and output sequences are then fed to the language model to perform a full fine-tuning of the model.

Full fine-tuning includes changing the weights of the entire language model. Performing a full fine-tuning can be costly in terms of storage and memory as we need to store a new model for every downstream task that we perform. On the other hand, there are ways which keep the weights of the language model frozen, but optimize a small task-related vector. Prefix-tuning [41] and Prompt tuning [39] are two examples of recent advancements that allow to achieve this.



| | | | | | |
|---|---|---|---|---|---|
| Log Instance: | dec | 28 | 13:11:46 | ubuntu | crond |
| IOB Tags: | B-TIM | I-TIM | I-TIM | I-HOS | I-SER |
| Wordpiece tokens: | dec | 28 | [ ‘13’, ‘##:’, ‘##11’, ‘##:’ , ‘##46’] | [ ‘u’, ‘##bu’, ‘##nt’, ‘##u’] | [ ‘cr’, ‘##ond’] |
| IOB Tags: | B-TIM | I-TIM | [ I-TIM, X, X, X, X] | [ I-HOS, X, X, X] | [I-SER, X] |

** The ‘##’ prefix before sub-token signifies that there is no space between it and the previous sub-token.
** All the subtokens starting with ‘##’ are mapped to the ‘X’ tag.

Figure 3.9: Example of a sub-tokenised input for BERT

### 3.3.5 Ensembles

Ensembles are a class of machine learning models formed by combining two or more machine learning models. The concept of ensembles is closely based on the human group decision-making process in which an opinion by a human is analogous to the classification by a classifier [30]. The intuition behind ensembles is that each individual classifier in a group has its own perspective of looking at the problem being solved and learning it.

Ensembles aggregate the predictions from participating models to output a final prediction. Combining the predictions from different models leads to depreciation in prediction errors and, hence, a boost in prediction accuracy [19, 35]. The aggregation of predictions from multiple models can be carried out in various ways. The most straightforward way is to perform majority voting. In majority voting, we vote between the classifier's predictions and move ahead with the output having a maximum count of votes. Another way is to look at the prediction probabilities for all the classifiers and average them out; this is called soft voting. We can also assign weights to different classifiers such that a particular classifier holds more dominance in predicting an output. These weights can either be learned or set by a domain expert.

Ensembles can be formed in two ways, including 1) Training multiple classifiers of the same type on different versions of data; and 2) Training different types of algorithms on the same dataset. The ensembles that use the same algorithm are called homogeneous ensembles, whereas those using different algorithms are termed hybrid/heterogeneous ensembles. Both these types of ensembles have been applied and heavily exploited in a wide variety of domains, including medical [15, 45, 71] and cybersecurity [11, 33, 34, 54] to name a few.

We chose to form the ensemble using different learning algorithms rather than training the same algorithm on different subsets of logs. We select a total of 3 different algorithms from different categories: a graphical model: CRF, a language model: BERT and a sequence-to-sequence model: Transformer Neural Network. The traditional word-based classifiers are not chosen because of their poor performance and inability to contribute to the ensemble. NERLogParser is not chosen as it is not trained on the augmented logs and is just used to evaluate generalizability. Each of the approaches used in creating the ensemble tackles the problem of Named Entity Recognition differently, and the underlying principles of each algorithm vary. CRF is based on the feature functions that consider the word-level features, as shown in Table 3.4. BERT is a language model capable of understanding the English language, and the inference depends on the sub-tokenized version of the log, as depicted in Figure 3.9. On the other hand, the transformer neural network tackles the NER problem by framing it into a word-based sequence-to-sequence problem. Using such diverse principles within an ensemble could reduce prediction errors while performing NER. Figure 3.10 depicts the Majority Voting Ensemble (MV Ensmbl) and Soft Voting Ensemble (SV Ensmbl) using CRF, BERT and Transformer. The predictions of the individual classifiers are aggregated using a voting mechanism in majority voting, whereas the prediction probabilities are averaged in the case of a soft vote.

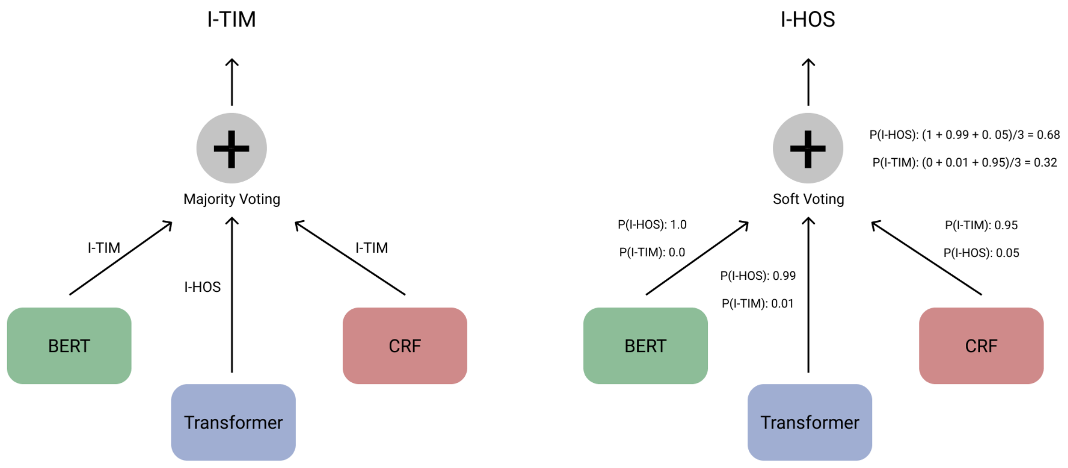


Figure 3.10: MV Ensmbl and SV Ensmbl using CRF, Transformer and BERT

### 3.3.6 Cascades

As discussed in the previous subsection, traditional ensembles may boost performance by taking multiple algorithms/opinions into consideration, but this increase in performance comes at the cost of:

1. Computational Expensiveness.
2. Increased Complexity of Models.
3. Less Interpretability of Ensemble.
4. Increase in Training and Inference Times.

Since log parsing finds its importance in software monitoring and system security, there is quite a high chance that the tools using log parsers would require a massive throughput in real-time. Traditional ensemble-based solutions may help boost the performance but may not be a good fit for such use cases because of the enormous inference times and computation [66]. For such use cases, we want solutions comparable to traditional ensembles in terms of performance but, on the other hand, possess less inference time and compute.

One of such solutions is a cascade/cascading classifier [2]. The Cascading Classifier is a class of ensembles that uses two or more classifiers but operates differently from traditional ensembles. Traditional ensembles are based on the aggregation of classifiers' predictions, whereas cascades are not. Cascades follow linear fashioned cascading from one classifier to another to transfer control flow. Figure 3.11 shows the architecture of a cascading classifier. The participating classifiers in a cascade are arranged in a fashion that the least computationally expensive model comes first and is followed by classifiers that are more computationally expensive.

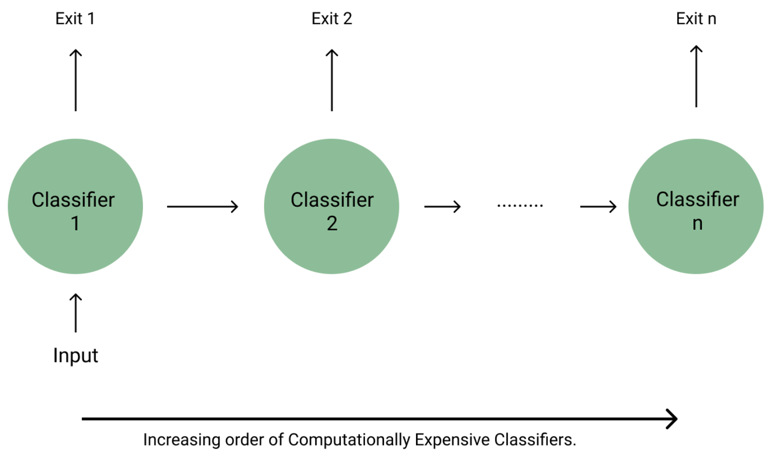


Figure 3.11: Architecture of a Cascade

Cascades are driven by policies/rules which govern the control flow. These policies are the basis on which an exit is made from the cascade. The policies in a cascading classifier can vary from architecture to architecture, but one of the most common policies is based on confidence thresholds. In such a policy, we compare a classifier's prediction probability with a pre-defined confidence threshold; based on the comparison, we either pass the control to the next classifier or exit the cascade.

To build a cascade for our use case, we chose:

1. The same pre-trained classifiers as used in the ensembles: CRF, BERT and Transformer.

2. A soft voting node at the end as a fallback for all the classifiers.

3. A confidence threshold-based policy.

CRF is the least computationally expensive out of the three classifiers, whereas the Transformer Neural Network is the most expensive. We arrange the classifiers in an order of increasing complexity. We also adjust our architecture by adding a soft voting node at the end of the cascade; this means we have a soft-voting ensemble as our terminal fallback. Since we have four nodes in our cascade, the total thresholds to be decided are three. For simplicity, rather than choosing three thresholds for each step, we prefer a common threshold that can be applied to the whole cascade. Figure 3.12 shows the cascading classifier built for our use case.

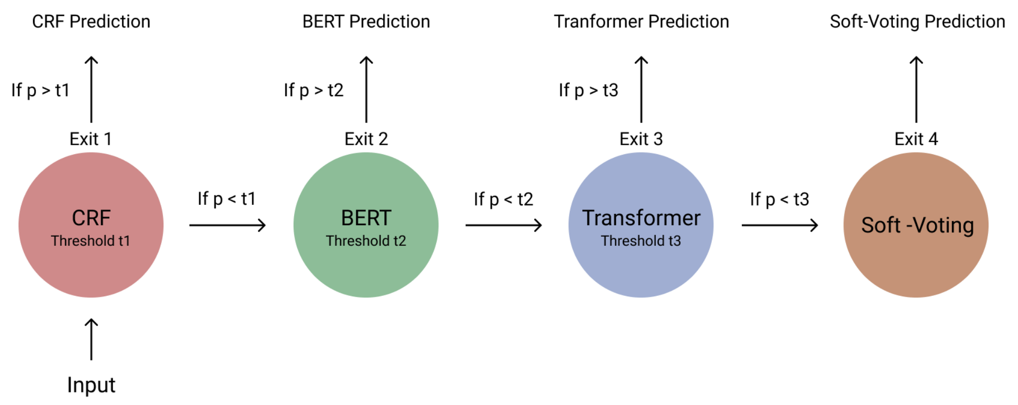


Figure 3.12: Cascade Architecture built using CRF, BERT and Transformer.

We experiment with a total of 11 thresholds ranging in [0, 1], compare the results and choose the threshold giving the best performance. A threshold-based analysis of various cascades is present in Section 5.4. A zero threshold implies that control would be constrained just to the CRF classifier and won't be cascaded ahead. On the other hand, the threshold value of 1 leads to the transfer of control to the soft voting ensemble. The cascade becomes a computationally expensive algorithm as we keep on increasing the threshold towards one. We also consider two more cascade models: a fusion of CRF-BERT (CB Cascade) and CRF-Transformer (CT Cascade). These architectures are evaluated to understand whether the smaller cascades can match the performance of the CRF-BERT-Transformer Cascade (CBT Cascade). The CB Cascade has three nodes: CRF, BERT and soft-voting node. Similarly, the CT Cascade is built using the CRF, Transformer and soft-voting node. The same threshold values and experimental settings are used in CB Cascade and CT Cascade as in the CBT Cascade.

## 3.4 Delimiter Classification Unit

Entities within a log may not always be space-separated. There can be a presence of some other special symbols that act as a delimiter to separate the entities. The significance of a delimiter is to separate the entities so that analytics can be easily performed. This delimiter is configurable and may change across different applications, devices, development teams, or organizations. Table 3.6 shows an example of log entities separated by different delimiters.

The machine learning models discussed in the previous section operate over space-separated log tokens. For the models to perform NER correctly, all the log entities should be space-separated. The presence of a delimiter other than space causes the entities to concatenate together, thus forming a single token. This concatenation of the tokens leads

| **Log** | **Delimiter Present** |
|---|---|
| 3 Jan 13:11:43 ps3 NetworkManager: <info> Trying to start... | Space |
| 3 Jan 13:11:43\|ps3\|NetworkManager:\|<info>\|Trying to start... | Pipe (\|) |
| 3 Jan 13:11:43\tps3\tNetworkManager:\t<info>\tTrying to start... | Tab (\t) |
| 3 Jan 13:11:43;ps3;NetworkManager:;<info>;Trying to start... | Semi-colon (;) |

Table 3.6: Log Entities separated by different delimiters

to wrong predictions as the model could miss a few tokens because of the delimiter present between the entities. Hence, there is a need for a solution that converts the delimiter separated logs to purely space-separated tokens; this is handled by the Delimiter Classification Unit as shown in Figure 3.13.

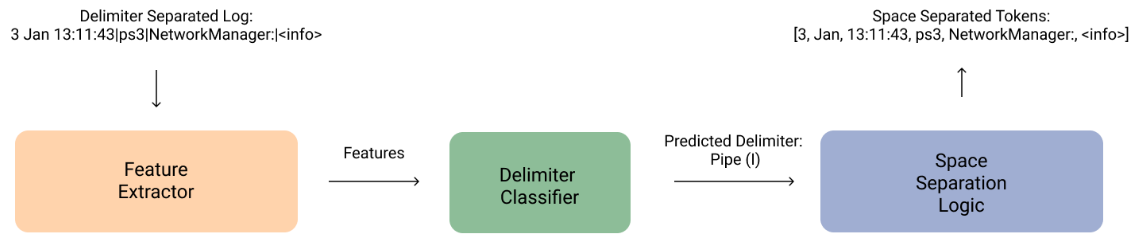


Figure 3.13: Delimiter Classification Unit

The Delimiter Classification Unit consists of three different components as listed below:

- **Feature Extractor**: Machine Learning algorithms cannot be directly applied to the logs, and thus we need to extract features from the log dataset using feature

| **Feature Name** | **Description** |
|---|---|
| num_chars | The total number of characters present in the log instance. |
| delim_freq | The total number of occurrences of a delimiter being considered. |
| delim_first_idx | The index of the first occurrence of the delimiter being considered. |
| delim_last_idx | The index of the last occurrence of the delimiter being considered. |

Table 3.7: Features used for building the delimiter classifier

engineering. Each log in the dataset is iterated, and different character-level features are extracted. Table 3.7 shows the different features and their description that are computed for every log instance present. The delim_freq, delim_first_idx and delim_last_idx features are calculated for all the possible delimiters being considered.

- **Delimiter Classifier**: Once the features are successfully extracted from the training logs using the feature extractor, the entire dataset can be used to train multiple machine learning models. Different machine learning models like Decision Tree, Random Forest, K Nearest Neighbors and more can be used to prepare the classifier. This classifier is then used to predict the delimiter in real-time.

- **Space Separation Logic**: The Log Parsing Unit expects the input formatted as space-separated log tokens. This component uses the predicted delimiter to generate input data in the required format.

## 3.5 IOB Tag Stitching Unit

The IOB Tag Stitching Unit is responsible for converting the IOB tags to entities. The log parsing unit outputs IOB tags for each corresponding log token, which is fed as an input to the IOB tag stitching unit. The tags having the same suffix are stitched together as an entity. For example, B-TIM and I-TIM tags are stitched together to form the timestamp entity. On the other hand, the stitching process operates slightly differently for the language models because of the sub-tokenized token input. Stitching is a two iteration process for the sub-tokenized input: 1) Every tag that is followed by a series of X tags is stitched to form a single IOB tag. 2) Tags having the same suffix are stitched together. Once the IOB tag stitching is complete to form entities, the result is converted to a JSON like key-value representation. Figure 3.14 clearly depicts the IOB tag stitching unit.

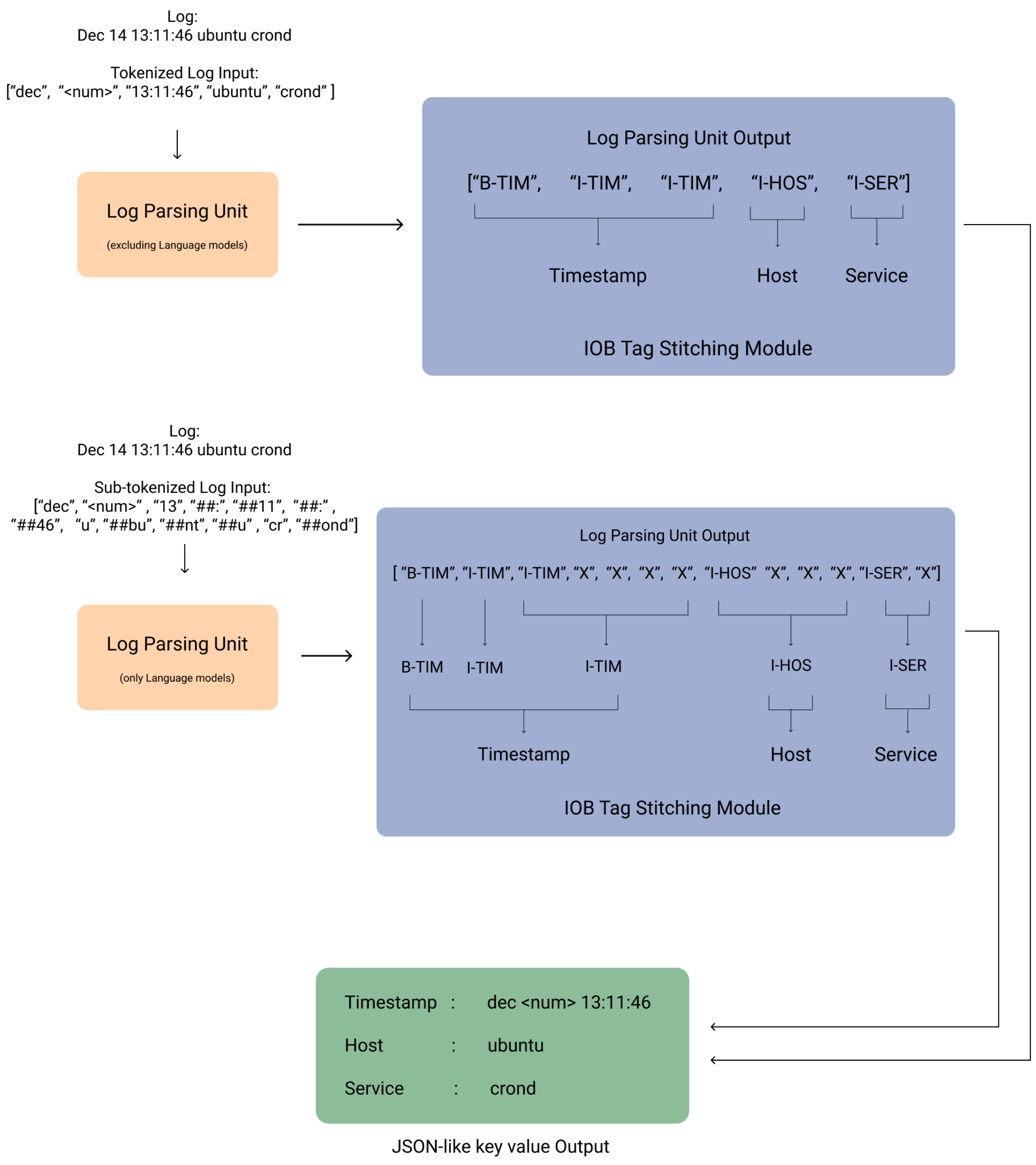

Log:
Dec 14 13:11:46 ubuntu crond
Tokenized Log Input:
[“dec”, “<num>”, “13:11:46”, “ubuntu”, “crond” ]
Log Parsing Unit
(excluding Language models)
Log Parsing Unit Output
[“B-TIM”, “I-TIM”, “I-TIM”, “I-HOS”, “I-SER”]
Timestamp
Host
Service
IOB Tag Stitching Module
Log:
Dec 14 13:11:46 ubuntu crond
Sub-tokenized Log Input:
[“dec”, “<num>” , “13”, “##:”, “##11”, “##:” , “##46”, “u”, “##bu”, “##nt”, “##u” , “cr”, “##ond”]
Log Parsing Unit
(only Language models)
Log Parsing Unit Output
[ “B-TIM”, “I-TIM”, “I-TIM”, “X”, “X”, “X”, “X”, “I-HOS” “X”, “X”, “X”, “I-SER”, “X”]
B-TIM
I-TIM
I-TIM
I-HOS
I-SER
Timestamp
Host
Service
IOB Tag Stitching Module
Timestamp : dec <num> 13:11:46
Host : ubuntu
Service : crond
JSON-like key value Output

Figure 3.14: IOB Tag Stitching Unit

# Chapter 4

# Experimental Design

In the previous chapter, we saw how the log parsing problem could be framed as a NER problem using different classifiers. We discussed LoDU and how it performs log augmentation and enrichment techniques that could help tackle log parsing for a wide variety of logs. We also explored LoPU and the different classifiers it considers to carry out NER. These classifiers included baseline solutions like Naive Bayes, graph-based methods like Conditional Random Fields, sequence-to-sequence methods like Transformer Neural Network, neural language models like BERT, and committee-based approaches like Ensembles and Cascades. We also discussed the other units, including the Delimiter Classification Unit and the IOB Tag Stitching Unit. In this chapter, we will be discussing the experimental design and settings used to carry out the experiments. The chapter is organized into multiple sections discussing the datasets used, machine learning algorithms and hyperparameter optimization, the strategy used to evaluate the NER solutions and system and hardware requirements to carry out the experiments.

| **Input Sequence** | **Output Sequence** |
|---|---|
| $[Dec, 11, 13:11:46, ubuntu, gdm[8057], ....]$ | $[B-TIM, I-TIM, I-TIM, I-HOS, I-SER, ....]$ |
| $[220.181.108.182, -, -, [01/Jan/2017:08:44:12, -800], ....]$ | $[I-IPA, I-DAS, I-AUT, B-TIM, I-TIM, ....]$ |

Table 4.1: CSV Files Structure

## 4.1 Datasets

We have already discussed that the problem of log parsing can be framed as a NER problem. The NER problem is a use case that falls under the umbrella of supervised learning. Supervised learning includes training a classifier with a pair of input and output. In the context of logs, a single log instance can be considered an input, and the output should be a value that indicates which set of tokens represent which entities. A sophisticated way to achieve this is to have IOB tags for each token present in the log instance as used in [58]. The dataset should have a structure consisting of pairs of input and output sequences; input sequences representing the log tokens and the output sequences depicting the respective IOB tags for the tokens. Table 4.1 shows the structure of the CSV files that we create for our datasets.

The log files do not inherently contain IOB tags, but these tags are vital and are needed to carry out NER. We can form regular expressions or grammar to perform labelling of the dataset. Logs can be derived from various sources and can have completely different structures and content. The varying structure and scope of logs make it hard to write a standard grammar or regular expression that tackles various logs. To simplify the entire process, we develop one or more grammar for each source. Sources that have an exact similar structure tend to follow a standard grammar for all the logs, but there could be a few sources that have logs utterly different in terms of structure. We develop multiple grammars for some sources to handle such situations and simplify the labelling strategy. Figure 4.1 shows an example of grammar for a log belonging to a particular log file.

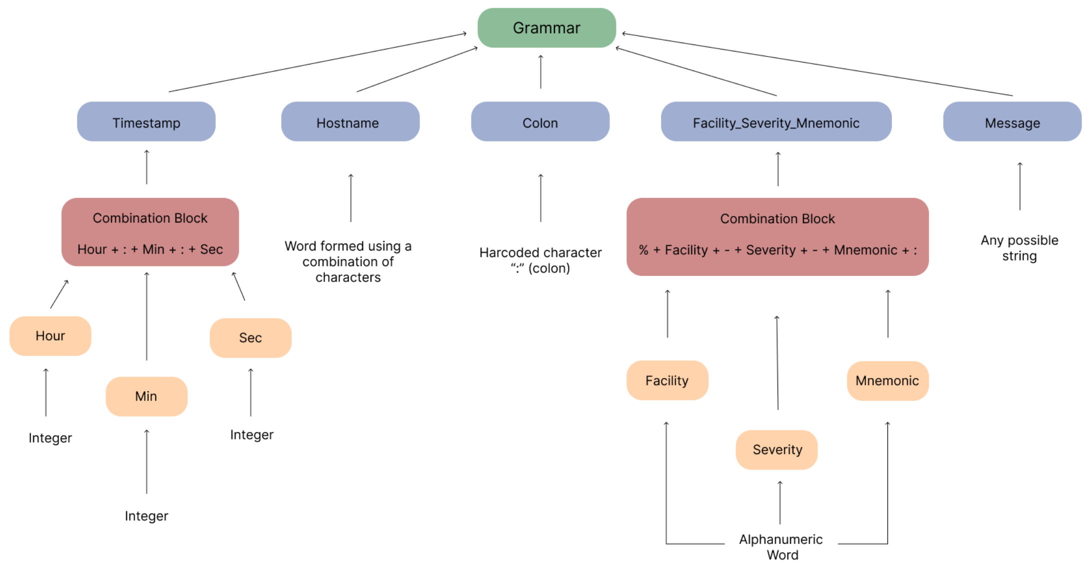


Figure 4.1: Example of a grammar used to Tag Entities

In this era of data-driven applications and large-scale distributed systems, millions of events are being logged per second. The number of logs can quickly grow to hundreds of billions if we consider logs being ingested from numerous sources over a long period of time. Collecting logs from such an environment and in such a volume guarantees diverse nature, structure and content but poses a few challenges. One of these challenges is the training of machine learning and deep learning models, as having such a quantity of data requires substantial computing resources and time. To solve this problem, we use a small dataset of logs and treat the logs using the LoDU to enrich and diversify in terms of structure and content. The idea of log diversification extends well to use-cases that include logs from unseen sources.

We have divided our datasets into two different categories based on whether they are used to train-validate-test the machine learning models or only test. These are 1) The In-Scope dataset and 2) The Out-of-Scope dataset. The In-Scope dataset is used to train all the classifiers present in the LoPU; it is also used for validation and testing purposes. In contrast, the Out-of-Scope dataset consists of logs originating from entirely different sources than the In-Scope dataset. It is used to test how well the machine learning models perform with logs from new and unseen sources. The Out-of-Scope dataset gives a reasonable estimate of the generalizability of the trained models. In the upcoming subsection, we discuss the In-Scope and the Out-of-Scope Datasets in detail.

### 4.1.1 In-Scope Dataset

The In-Scope dataset is used to train, validate and test the various NER solutions we prepare. This dataset is a subset of the logs used to train the NERLogParser in [58]. We contacted the author of NERLogParser [58] to collect the entire dataset and were able to gather a fair number of the sources but missed some of the sources including the "Honeynet Challenge logs". The logs in the gathered dataset originate from 10 different sources, which include: Auth Logs, Bluegene Logs, Daemon Logs, Debug Logs, Dmesg Logs, Kernel Logs, Message Logs, Proxifier Logs, Web Logs, and Zookeeper Logs. Logs from all the sources mentioned above differ in the following terms: 1) The event being

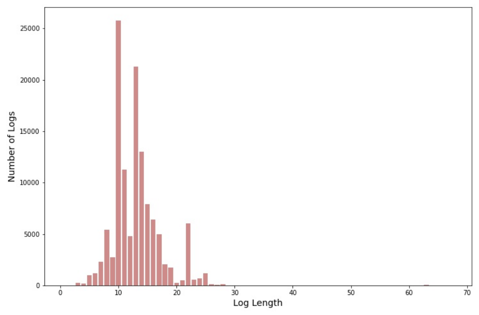


Figure 4.2: In-Scope Length Distribution.

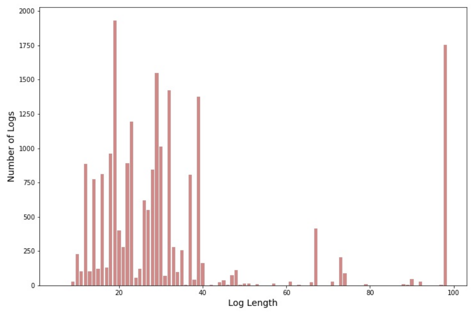


Figure 4.3: OOS Length Distribution

logged, 2) The Application/device from where the event is being logged, 3) The structure of the log, and 4) The types of entities present in the log. All the log instances from the mentioned sources are aggregated in a single CSV file to form the In-Scope dataset spanning over 120,000 log instances. Table 4.2 depicts the distribution number of logs and entities grouped by the source.

The In-Scope dataset contains a total of 23 unique entities, including timestamp, hostname, service, sub-service, IP address and more. Dmesg logs contain three entities, which is the least, whereas weblogs comprise nine entities, the highest amongst all the sources. The length of the log sequences in the dataset lies in the range [1, 67]. Figure 4.2 shows the distribution plot of the sequence length of the logs present in the In-Scope dataset. To carry out the training and evaluation process, we divide the In-Scope dataset into three parts: a training set, a validation set and a testing set using a 60-20-20 split resulting in $\sim 73,000$ log instances in the training set and $\sim 24,000$ logs in each of the remaining sets.

Table 4.2: Distribution of the number of logs and entities for the In-Scope dataset.

| **Log File** | **No.Instances** | **No.Entities** | **Entities present in the logs** |
|---|---|---|---|
| Auth | 16669 | 5 | timestamp, hostname, service, subservice, message |
| Bluegene | 10001 | 8 | socket, number, timestamp, core, source, service, level, message |
| Daemon | 9809 | 5 | timestamp, hostname, service, subservice, message |
| Debug | 1722 | 6 | timestamp, hostname, service, unix_time, subservice, message |
| Dmesg | 7218 | 3 | unix_time, subservice, message |
| Kernel | 34246 | 6 | timestamp, hostname, service, unix_time, subservice, message |
| Message | 11338 | 6 | timestamp, hostname, service, unix_time, subservice, message |
| Proxifier | 10107 | 6 | timestamp, service, arch, domain_or_ip, status, message |
| Web | 10883 | 9 | ip_address, dash, auth, timestamp, command, status_code, num_bytes, referrer, client_agent |
| Zookeeper | 10000 | 5 | timestamp, dash, status, job, message |
| **Total** | **121993** | | |

### 4.1.2 Out-of-Scope Dataset

Our work's primary novelty is supported and rooted in the usage of the Out-of-Scope dataset. The Out-of-Scope dataset is solely used for evaluating the generalizability of the classifiers trained in the LoPU. None of the logs present in this dataset has been seen during the training or the validation phase. These logs originate from 9 different applications, servers, systems and devices, including Cisco Adaptive Security Appliance (Cisco ASA), Cisco Internetwork Operating System (Cisco IOS), Linux Secure, Linux Apache, Linux Secure, two different NGINX servers, Windows Application Events (WAE), Windows Security Events (WSE), and Windows System Events (WSYE). None of these sources are shared with the In-Scope dataset, and thus, the In-Scope and Out-of-Scope datasets are mutually exclusive in nature. Initially, we had only a single NGINX log file which was later split into two variants (v1 and v2) as logs from two different NGINX servers were present in the file.

The logs are collected from sources deployed within the premises of the University of Ottawa. The number of events logged from these sources is enormous, but we decided to use only a small subset from each source, contributing to a total of $\sim 21,000$ log instances. The logs present in this dataset differ in structure and entities present; Figure 4.3 shows a distribution of the length of logs present in the dataset. To evaluate the performance of different classifiers on the Out-of-Scope dataset, we need a golden standard to compare the predictions. We analyze the structure of all the log files and develop different grammar to prepare the IOB tags for the logs. The total number of unique entities present in the Out-of-Scope logs is 37, whereas the In-Scope dataset has 23. Out of these sets of entities, only 11 are common, and we evaluate our approaches only on these entities. Table 4.3 shows the distribution of the number of logs and entities across various log files collected.

### 4.1.3 LoDU treated In-Scope Dataset

To develop a solution that generalizes well to new and unseen logs, we need to make sure that the training dataset is diverse enough and thoroughly understands the outline of each entity present in the logs. We use LoDU to add a diversity vertical to our solution. The training set of the In-Scope dataset is treated with LoDU to carry our enrichment and diversification of logs. The design of LoDU is not influenced by the logs present in the Out-of-Scope dataset. In other words, none of the logs present in the Out-of-Scope dataset were seen while preparing augmentation techniques for the training logs. Table 4.4 shows some of the dataset metrics before and after the treatment. The number of date formats increases from 5 to 54, the number of hosts increases from 4 to $\sim 58,000$, and the number of unique sequences rises from 244 to 15235. Figure 4.4 represents the distribution of entities present in the LoDU treated logs. The distribution of entities is not balanced within the logs. Entities like timestamp (TIM), host (HOS), service (SER) and subservice (SUB) are present in more than 40,000 log instances. On the other hand, entities like arch (ARC) are not well represented and is present in less than 50 log instances. The diversity added by LoDU is quite considerate and increase the total number of total instances from $\sim 73,000$ to $\sim 110,000$. Finally the LoDU treated logs are used to train all the classifiers present in the LoPU.

Table 4.3: Distribution of number of logs and entities for the Out-of-Scope dataset.

| **Log File** | **No.Instances** | **No.Entities** | **Entities present in the logs** |
|---|---|---|---|
| Cisco ASA | 1691 | 5 | timestamp, hostname, colon, facility_severity_mnemonic, message |
| Cisco IOS | 2999 | 6 | timestamp, hostname, colon, service, facility_severity_mnemonic, message |
| Linux Secure | 3000 | 13 | timestamp, hostname, service, ip_address, dash, auth, http_request_timestamp, http_command, status_code, num_bytes, referrer, client_agent, message |
| Linux Apache | 3000 | 9 | ip_address, dash, auth, timestamp, http_command, status_code, num_bytes, referrer, client_agent |
| NGINX v1 | 2954 | 17 | ip_address, dash, auth, timestamp, http_command, status_code, num_bytes, referrer, client_agent, http_x_forwarded_for, request_time, upstream_response_time, scheme, scheme_protocol, url, http_range, sent_http_x_varnish_cache |
| NGINX v2 | 46 | 17 | ip_address, dash, auth, timestamp, http_command, status_code, num_bytes, referrer, client_agent, http_x_forwarded_for, request_time, upstream_response_time, scheme, scheme_protocol, url, http_range, sent_http_x_varnish_cache |
| Win App Events | 1392 | 12 | timestamp, logname, source_name, event_code, event, event_name, computer_name, task_category, op_code, keywords, record_number, message |
| Win Sys Events | 3000 | 15 | timestamp, logname, source_name, event_code, event, event_name, computer_name, user, security_identifier, security_identifier_type, task_category, op_code, record_number, keywords, message |
| Win Sec Events | 3000 | 12 | timestamp, logname, source_name, event_code, event, event_name, computer_name, task_category, op_code, record_number, keywords, message |
| **Total** | **21082** | | |

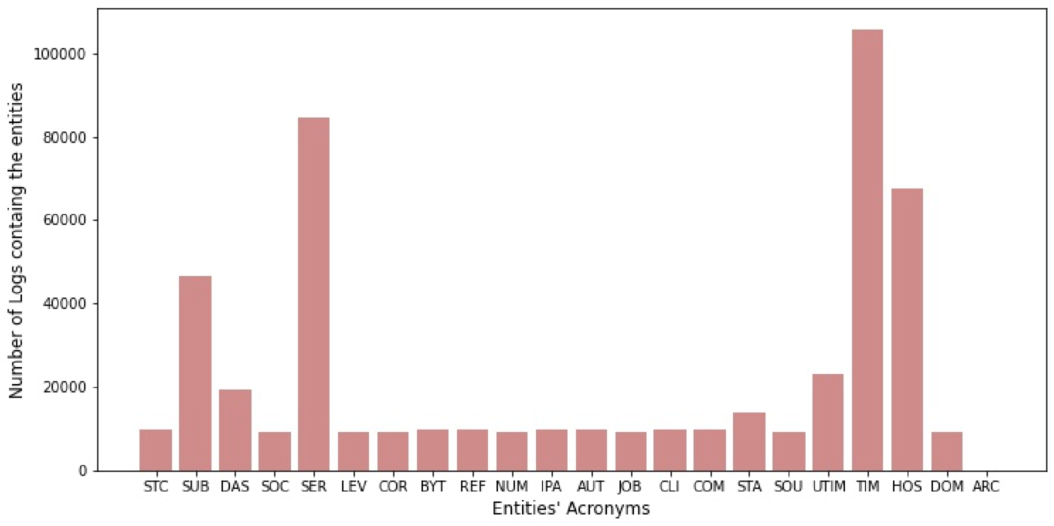


Figure 4.4: Distribution of Entities in LoDU treated logs.

| **Metric** | **Sub-module used** | **Pre-treatment Value** | **Post-treatment Value** |
|---|---|---|---|
| No. of Date Formats | Date-Enrichment Sub-module | 5 | 54 |
| No. of Hostnames | Host-Enrichment Sub-module | 4 | 58388 |
| No. of unique Entity Sequences | Entity Shuffling Module | 244 | 15235 |

Table 4.4: Comparison of Metrics before and after treatment of logs by LoDU.

## 4.2 ML Algorithms and Hyperparameter Optimization

### 4.2.1 ML Algorithms

We use 14 different ML-based approaches to carry out NER. These include models having different underlying principles and include three traditional word-based methodologies: Naive Bayes, SGD, and Perceptron; a graphical model: CRF, two sequence-to-sequence models: NERLogParser, and Transformer Neural Network; three language models: BERT, RoBERTa, and DistilBERT; two traditional ensembles: MV Ensmbl and SV Ensmbl; three cascading classifiers: CBT Cascade, CB Cascade, and CT Cascade. Each methodology accepts data in its own compatible format, and the raw log sequences need to be pre-processed and converted to the desired format. For example, CRF relies on data present in the form of feature functions; Traditional word-based methodologies rely on token-to-tag mapping. All of the classifiers and their respective compatible data format has already been discussed in Section 3.3. In the next subsection, we discuss hyperparameter optimization and how it is carried out for the various algorithms discussed.

### 4.2.2 Hyperparameter Optimization

Hyperparameters are a set of attributes or properties belonging to a machine learning or deep learning model that: 1) Helps them in the decision-making process during the training and inference time; and 2) Helps set up the model's architecture and change the way they operate. Hyperparameters allow us to configure the internals of a machine or deep learning classifiers. These can be considered the knobs of an algorithm that can help tweak the model's behaviour.

Hyperparameter optimization is the process that tries to scout for hyperparameter values that result in finding the global maxima/minima of the performance metric being considered. There are several ways to carry out the hyperparameter optimization process. Some of these are 1) Manual Search, 2) Grid Search, 3) Random Search, and 4) Bayesian Optimization. All of these approaches are discussed below.

- **Manual Search**: Manual search includes searching for the best hyperparameters manually. Generally, a domain expert would look at the performance of different models and, based on the metrics, would hand tune the hyperparameters till a satisfactory performance metric is obtained.

- **Grid Search**: Grid Search requires defining a search space for each hyperparameter. The performance metric is calculated for all the possible combinations of hyperparameters, and the best hyperparameters are returned. If we have two hyperparameters with a search space of $n$ values each, then $n^2$ is the number of possible combinations. The machine learning model would be trained and evaluated $n^2$ times, and the best hyperparameters are returned.

- **Random Search**: While performing a random search of hyperparameters, we need to specify the search space for each input and the total number of iterations that we want to run. After each iteration, models are trained and evaluated using random combinations of different hyperparameters. Once the number of iterations is exhausted, the best-performing hyperparameters are selected.

- **Bayesian Optimization**: Unlike Grid Search and Random Search, Bayesian Optimization does not make use of all possible/random combinations of hyperparameters. Instead, it follows a statistically guided approach to search for the best performing hyperparameters [22]. We have used this approach to carry out the hyperparameter optimization for our use case. The following subsection discusses Bayesian Optimization in detail.

#### 4.2.2.1 Bayesian Optimization

Bayesian Optimization treats every algorithm as a black box, looks at the input-output pairs and tries to model the hidden objective function. It prepares a probabilistic model that approximates the objective function. Computing the objective function using all the possible values in the input space is a costly process. Instead, Bayesian Optimization lets us approximate the objective function using less number of trials. Once the function has been approximated, the global minima/maxima can be calculated, and so can be the hyperparameters. The underlying objective function can be predicted by evaluating the model at each combination of hyperparameters and is easily achievable using grid search. Although this approach can expose the hidden objective function, it is not a scalable approach in the era of deep learning based algorithms. Exhausting all the hyperparameter values for training complex models like BERT and Transformer may require enormous time and resource consumption. We can avoid such brute force approaches by using educated and statistically backed input guesses. Bayesian Optimization is one such educated approach that develops a probabilistic model that helps in a guided search of parameters. Bayesian Optimization tries to approximate the objective function by modelling a surrogate function. Figure 4.5

shows the iterative modelling of the surrogate function. The blue colour function depicts the real objective function that the surrogate function tries to mimic. Red stars denote the points where the black box evaluation has been carried out. In part A, the surrogate function is modelled using five observations and does not quite match the shape of the underlying objective function. New hyperparameter inputs are statistically calculated and are evaluated. This is followed by the re-modelling of the surrogate function in part B. Finally, after a certain number of iterations, we observe that the surrogate function has almost approximated the hidden objective function in part C. The approximated function gives us a fair idea about the best-performing hyperparameters.

We set a search space for each hyperparameter that defines a domain within which the value should be searched. A total of 20 trials are carried out to approximate the best performing hyperparameters for each algorithm used. The best-performing hyperparameters are chosen, and the resultant models can be used for testing and deployment purposes. Hyperparameter optimization is not perfromed for NERLogParser, ensembles and cascades. NERLogParser is used as a pre-trained model for log parsing and the pre-trained frozen model weights are used. On the other hand, the best performing individual models (CRF, BERT, Transformer) are used to form the ensembles and cascades.

## 4.3 Evaluation Strategy

Each trained classifier outputs a sequence of IOB tags when prompted with a log sequence. A trivial approach to evaluate the log parsing systems is to compare the golden standard IOB tags with the predicted IOB tags. Each IOB tag is treated as a separate class, and

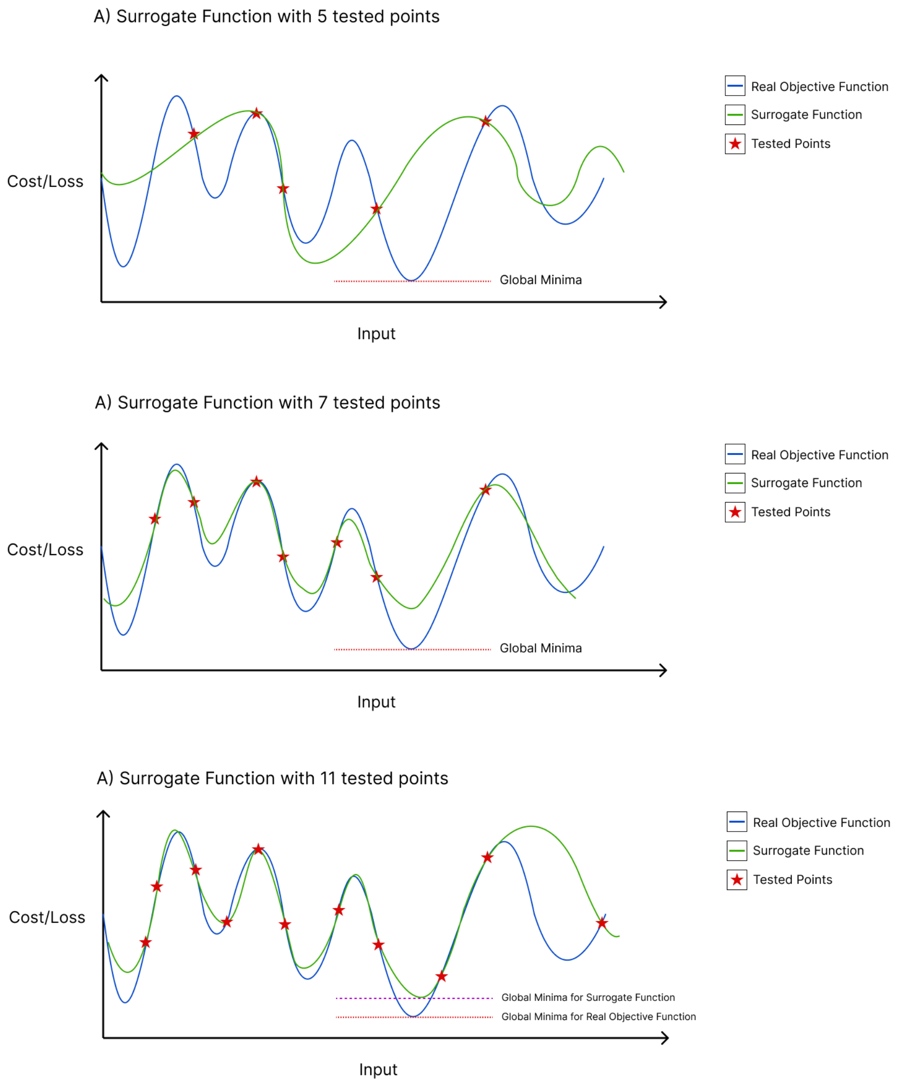
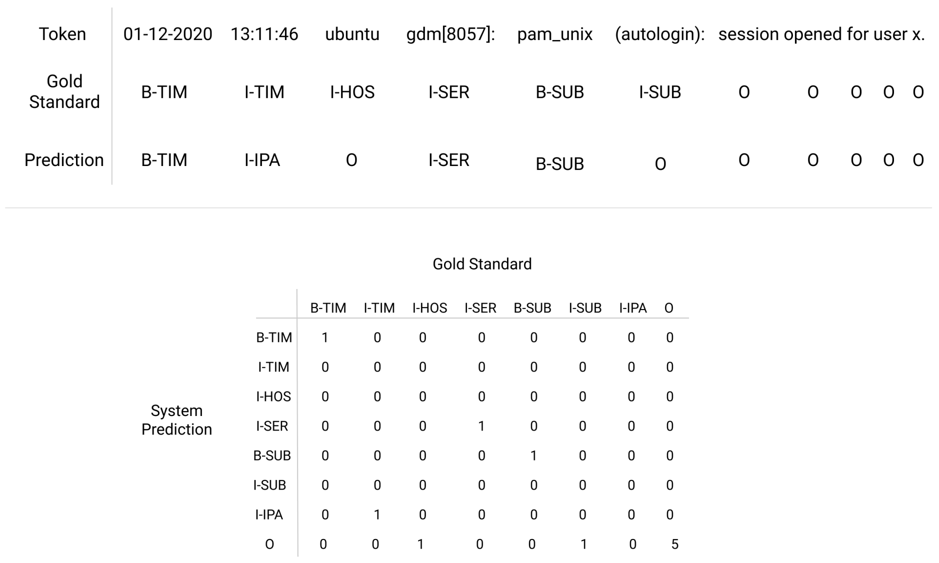


Figure 4.5: Modelling of Surrogate Function

the evaluation becomes similar to a multiclass classification problem use case. Standard evaluation metrics like accuracy, precision, recall and F1 score can be computed for each classifier and used for comparison purposes. Figure 4.6 shows how a confusion matrix can be prepared to calculate the standard metrics. Accuracy can be calculated using the formula mentioned in Equation 4.1 where N represents the total number of IOB tags and $M_{ij}$ depicts the element present in the $i^{th}$ row, $j^{th}$ column of the confusion matrix.



| Token | 01-12-2020 | 13:11:46 | ubuntu | gdm[8057]: | pam_unix | (autologin): | session opened for user x. | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Gold Standard | B-TIM | I-TIM | I-HOS | I-SER | B-SUB | I-SUB | O | O | O | O | O |
| Prediction | B-TIM | I-IPA | O | I-SER | B-SUB | O | O | O | O | O | O |

Gold Standard

| System Prediction | | B-TIM | I-TIM | I-HOS | I-SER | B-SUB | I-SUB | I-IPA | O |
|---|---|---|---|---|---|---|---|---|---|
| | B-TIM | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | I-TIM | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | I-HOS | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | I-SER | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | B-SUB | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| | I-SUB | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | I-IPA | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | O | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 5 |

Figure 4.6: Token level evaluation and calculation of Confusion Matrix

$$\frac{\sum_{i=1}^{N} \sum_{j=1}^{N} M_{ij(i==j)}}{\sum_{i=1}^{N} \sum_{j=1}^{N} M_{ij}} \tag{4.1}$$

Using Equation 4.1 we obtain an accuracy of $8/11 = 0.7272$ for the example displayed in Figure 4.6. The accuracy score obtained here is fair, but if we look at an entity level (across span of tokens forming an entity), we notice that only the service entity (having I-SER tag) is correctly predicted. All the other entities, including timestamp, hostname, subservice and message, are incorrect across the complete span of tokens. Hence, this reflects that the evaluation methodology that we discussed above is not suitable for assessing the performance of the NER system as it may not reflect correctly the performance in some cases. The evaluation methodology mentioned above is based on a tag level and not on an entity level and thus can be misleading in some situations. Thus, we do not abide by this methodology. Since the end goal of a NER system is to extract the entities present within a sequence correctly, the evaluation should also be based on the entity level. Using an entity-level evaluation framework could make it easier to compare NER systems without misleading conclusions as in tag-based evaluation schemes. Apart from the framework, we also need to tackle some use-case related concerns like:

- Should we consider computing accuracy for the task at hand?
- If we consider the precision/recall framework, is precision more relevant than recall for our use case, or do both carry equal weight?
- What other contextual metrics should we consider apart from traditional ones?
- Is explainability important in our use case?

Answering the above questions gives us more clarity regarding what is expected from the system. Since entities may not be present in an equal ratio, it is wise not to choose accuracy as a measure of comparison. The trade-off between precision and recall is affected by the number of false positives and false negatives present. Minimizing the false positives increases precision, whereas minimizing the false negatives boosts the recall. In most cases, minimizing the count of one of these comes at the cost of increasing the other counter. In most cybersecurity problems, either of the two carries more weight. For example, precision is more relevant in spam detection, whereas, in intrusion detection, recall can be considered a better metric. In the log parsing use case, both the false positives and false negatives are relevant, and hence we choose the F1 score as the basis for comparing log parsing systems. Another essential metric to consider for this use case is the time taken in parsing the log. The time metric helps us identify whether the developed solution can be deployed in real-time scenarios. It can also be termed the system's throughput and estimates logs parsed per second. Explainability is another vital aspect for dissecting the working of machine learning models. The medical domain is one of the fields where explainability matters, but it does not hold significant importance in the log parsing use case; we are just interested in getting out logs parsed correctly.

Before taking a look at the evaluation framework, we need to understand all the possible scenarios that can take place in a NER-based log parsing system. Below are the six scenarios that can arise while a system is at work:

1. **System perfectly matches tokens and entity type**: The system predicts correct IOB tags for each of the tokens present in the sequence.

2. **System hypothesizes an entity**: The system predicts an entity in place of an 'O' tag.

3. **System misses an entity**: The system predicts one or more 'O' tags in place of valid IOB tags depicting an entity.

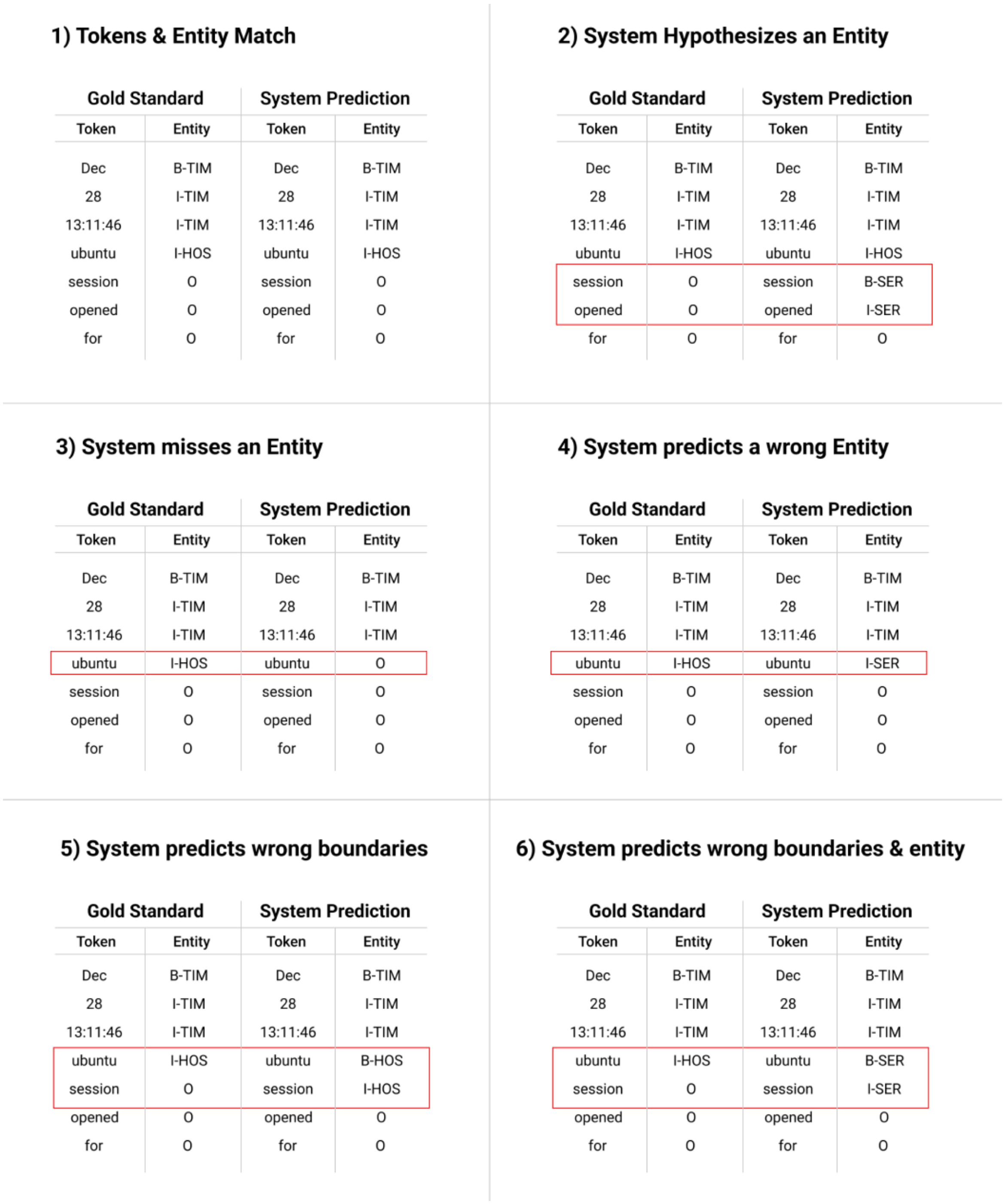
4. **System predicts a wrong entity**: The system misunderstands the concept of an

entity and predicts it as a different entity. The IOB tags of one entity are placed at the position of some other entity.

5. **System predicts wrong boundaries**: The predicted entity is correct but, the offset of IOB tags is misplaced.

6. **System predicts wrong boundaries and entity**: System predicts both the entity and the offset incorrectly.

Figure 4.7 shows an example representing all the possible prediction scenarios mentioned above. Scenarios 1-4 represent correct and incorrect predictions having an exact boundary match. On the other hand, scenarios 5 and 6 reflect partial matching. Partial matching includes the cases where a certain set of tokens get misclassified or gets drifted toward the neighbouring entities. Partial matches can easily occur within a NER system and carry information about an entity's underlying knowledge/concept. Metrics capturing partial matches can help us improve the existing systems by taking the necessary steps. An evaluation system considering partial matching can be robust and constructive. Hence, an evaluation framework should cater to the following for our use case: 1) It offers detailed metrics that help us understand how the systems are working at an entity level, 2) It offers metrics that are comparable while comparing two or more NER systems, and 3) It offers metrics that help us understand the areas where the System is performing inefficiently.



**1) Tokens & Entity Match**

| Gold Standard | | System Prediction | |
|---|---|---|---|
| Token | Entity | Token | Entity |
| Dec | B-TIM | Dec | B-TIM |
| 28 | I-TIM | 28 | I-TIM |
| 13:11:46 | I-TIM | 13:11:46 | I-TIM |
| ubuntu | I-HOS | ubuntu | I-HOS |
| session | O | session | O |
| opened | O | opened | O |
| for | O | for | O |

**2) System Hypothesizes an Entity**

| Gold Standard | | System Prediction | |
|---|---|---|---|
| Token | Entity | Token | Entity |
| Dec | B-TIM | Dec | B-TIM |
| 28 | I-TIM | 28 | I-TIM |
| 13:11:46 | I-TIM | 13:11:46 | I-TIM |
| ubuntu | I-HOS | ubuntu | I-HOS |
| session | O | session | B-SER |
| opened | O | opened | I-SER |
| for | O | for | O |

**3) System misses an Entity**

| Gold Standard | | System Prediction | |
|---|---|---|---|
| Token | Entity | Token | Entity |
| Dec | B-TIM | Dec | B-TIM |
| 28 | I-TIM | 28 | I-TIM |
| 13:11:46 | I-TIM | 13:11:46 | I-TIM |
| ubuntu | I-HOS | ubuntu | O |
| session | O | session | O |
| opened | O | opened | O |
| for | O | for | O |

**4) System predicts a wrong Entity**

| Gold Standard | | System Prediction | |
|---|---|---|---|
| Token | Entity | Token | Entity |
| Dec | B-TIM | Dec | B-TIM |
| 28 | I-TIM | 28 | I-TIM |
| 13:11:46 | I-TIM | 13:11:46 | I-TIM |
| ubuntu | I-HOS | ubuntu | I-SER |
| session | O | session | O |
| opened | O | opened | O |
| for | O | for | O |

**5) System predicts wrong boundaries**

| Gold Standard | | System Prediction | |
|---|---|---|---|
| Token | Entity | Token | Entity |
| Dec | B-TIM | Dec | B-TIM |
| 28 | I-TIM | 28 | I-TIM |
| 13:11:46 | I-TIM | 13:11:46 | I-TIM |
| ubuntu | I-HOS | ubuntu | B-HOS |
| session | O | session | I-HOS |
| opened | O | opened | O |
| for | O | for | O |

**6) System predicts wrong boundaries & entity**

| Gold Standard | | System Prediction | |
|---|---|---|---|
| Token | Entity | Token | Entity |
| Dec | B-TIM | Dec | B-TIM |
| 28 | I-TIM | 28 | I-TIM |
| 13:11:46 | I-TIM | 13:11:46 | I-TIM |
| ubuntu | I-HOS | ubuntu | B-SER |
| session | O | session | I-SER |
| opened | O | opened | O |
| for | O | for | O |

Figure 4.7: An Example depicting different Prediction Scenarios.

### 4.3.1 Performance Evaluation Framework

Chichor et al. [12] introduced five different scoring categories that help capture the different types of errors made by an NER system and are based on comparing the gold standard with the prediction made by the system. These scoring categories can be used to calculate the precision, recall and F1-score of any NER system and these namely are:

- Correct (**COR**): The prediction is same as the gold standard.
- Incorrect (**INC**): The prediction and the gold standard do not match.
- Partial (**PAR**): The prediction and the gold standard match partially.
- Missing (**MIS**): The gold standard is not captured by the system.
- Spurious (**SPU**): The predicted entity is not present in the gold standard.
- Actual (**ACT**): $|COR| + |INC| + |PAR| + |SPU|$
- Possible (**POS**): $|COR| + |INC| + |PAR| + |MIS|$

These scoring categories capture various aspects including full matches, partial matches, incorrect matches as well as missing entities. We employ the evaluation framework developed in [52] that uses the above mentioned scoring categories to compare the NER systems across two axes: tokens and entities. The framework offers four different evaluation schemes that let us assess the performance of the system in various scenarios and attributes. These include:

- **Strict Evaluation Scheme**: Includes token and entity-type matching. Evaluation takes place across both the axes.
- **Exact Evaluation Scheme**: Includes evaluation across one axis: tokens. Performs token matching, irrespective of the predicted entities.
- **Partial Evaluation Scheme**: Includes partial token matching, irrespective of the predicted entities.
- **Entity-Type Evaluation Scheme**: Includes entity-type matching, irrespective of the tokens.

Each of the evaluation schemes mentioned above has its own set of precision, recall and F1 score. These metrics over four evaluation schemes allow us to better understand the working of the NER systems. The strict evaluation scheme is the most favourable for comparing systems as it considers both axes (tokens and entities). The exact and partial evaluation schemes consider just the tokens. On the other hand, the entity-type evaluation only looks at the entities. The exact, partial and entity type evaluation schemes give a different perspective on the problem being solved. These are not used to compare the systems but help analyze and improve the systems. A NER system with a high F1 score in the strict evaluation scheme will reflect a high score in the rest of the three evaluation schemes. On the other hand, if a system has a high F1 score in the exact evaluation scheme but a comparatively low score in strict evaluation, it suggests that the boundary matches of the tokens are correct. However, the prediction of entities might be incorrect. A low F1 score in the strict and exact evaluation scheme but a high F1 score in partial evaluation indicates that the system outputs many boundary mismatches. Equations 4.2 and 4.3 depict the precision and recall for strict and exact evaluation schemes whereas Equations 4.4 and 4.5 state the same metrics for partial and entity type evaluation schemes.

$$Precision_A = \frac{|COR|}{|ACT|} \tag{4.2}$$

$$Recall_A = \frac{|COR|}{|POS|} \tag{4.3}$$

$$Precision_B = \frac{|COR| + (0.5 * |PAR|)}{|ACT|} \tag{4.4}$$

$$Recall_B = \frac{|COR| + (0.5 * |PAR|)}{|POS|} \tag{4.5}$$

Table 4.5 depicts an example showing the various scoring categories for the different evaluation schemes considered in the framework. Each row in the table represents an entity-tokens pair which is scored using the scoring categories already discussed. The scoring is performed for all the entity-tokens pair present in all the logs. By performing summation of various scoring categories across all the evaluation schemes we can calculate the precision and recall using the Equations 4.2-4.5. For the strict evaluation we perform summation over the column and get the following: $|COR| = 1$, $|INC| = 3$, $|PAR| = 0$, $|MIS| = 1$, $|SPU| = 0$, $|ACT| = |COR| + |INC| + |PAR| + |SPU| = 4$, and $|POS| = |COR| + |INC| + |PAR| + |MIS| = 5$. With the values obtained for different scoring categories we calculate the precision and recall for the strict evaluation scheme using Equations 4.2 and 4.3: $Precision_A = \frac{|COR|}{|ACT|} = \frac{1}{4} = 0.25$ and $Recall_A = \frac{|COR|}{|POS|} = \frac{1}{5} = 0.2$. The same methodology can be applied across all the evaluation schemes and the evaluation metrics can be computed for each.

Table 4.5: An example of the scoring categories for the different evaluation schemes.

| Ground Truth | | System's Prediction | | Evaluation Schemes | | | |
|---|---|---|---|---|---|---|---|
| **Entity** | **Tokens** | **Entity** | **Tokens** | **Strict** | **Exact** | **Partial** | **Entity Type** |
| Time | Dec 28 18:12:51 | Time | Dec 28 | INC | INC | PAR | COR |
| Host | ubuntu | Host | 18:12:51 ubuntu | INC | INC | PAR | COR |
| Service | cron[4154]: | Time | cron[4154]: | INC | COR | COR | INC |
| Sub-Service | pam_unix(cron:session): | Sub-Service | pam_unix(cron:session): | COR | COR | COR | COR |
| Message-Type | <info> | | | MIS | MIS | MIS | MIS |

### 4.3.2 Timing Evaluation

Timing is another critical factor in evaluating and comparing various systems. Log parsers find their application in security and monitoring systems. The nature of the systems can vary depending upon whether they process logs in real-time or not. If the systems are bound to make operations and take action in real-time, then the timing constraint carries utmost importance. For a real-time system, we would like to have a log-parser that could match the throughput of the parent system. Log parsers that require massive time in processing and outputting the entities are not a suitable fit for such real-time applications. On the other hand, if the use case being solved is time-independent, we may have room to explore approaches that could increase the performance at the cost of consuming more time. In both the cases discussed, time can be used to compare the log parsing systems. To make a fair comparison, we need to ensure that the measurement of time is standardized across all the approaches being considered. We considered the following processes while measuring the time: 1) Preprocessing data to model suitable format, 2) Predicting Delimiters, 3) Making IOB tag predictions, and 4) Saving the predictions in a csv file.

## 4.4 Software Tools and Libraries

All the experiments are carried out using Python v3.7.0 and compatible data processing and machine learning based libraries. Numpy v1.21.2 and Pandas v1.1.5 are used to load, preprocess and perform transformations on data. Pyparsing v2.4.7 is used to develop grammars to parse the logs and tag them with the IOB tags. All the traditional word-based NER approaches are implemented using scikit-learn v0.24.2. CRF is implemented using the sklearn-crfsuite v0.3.6 package. PyTorch v1.9.1 and torchtext v0.6.0 are used to implement the transformer neural network and other internals required to carry out the training and inference. Pre-trained language models, including BERT, RoBERTa and DistilBERT, are imported from the HuggingFace library. Bayesian Optimization is performed using the optuna v2.10 library [1].

## 4.5 System and Hardware Requirements

All the experiments, including training and inference, are carried out on a Linux machine on a cedar cluster belonging to Compute Canada. The training and inference scripts are submitted as jobs using Simple Linux Utility for Resource Management (SLURM). We use 16 GB RAM, a v100 GPU and a 4-core CPU for each job across all the experiments performed.

## 4.6 Summary

We discussed two datasets: the In-Scope and Out-of-Scope datasets. The In-Scope dataset is used to train-validate-test, whereas the Out-of-Scope dataset originates from a completely different distribution/source and is used to test the trained system to get an idea about the generalizability of the trained classifier. Since the end goal is to develop a system that can cater to many new and unseen logs, we treat the In-Scope logs with LoDU. The LoDU helps diversify and enrich the In-Scope logs that are finally used to train the classifiers present in the LoPU. We perform hyperparameter optimization using bayesian statistics to get the best-performing models. Bayesian Optimization provides a guided approach to approximating best-performing hyperparameters using less number of trials. We use an evaluation framework that offers multiple evaluation schemes: Strict Evaluation, Exact Evaluation, Partial Evaluation, and Entity-Type Evaluation to evaluate the trained models. These multiple perspectives help us evaluate, compare and improve the log parsing systems. Taking into account the log parsers' real-time usage, we also measure the timing attribute. Once all the evaluation metrics are in hand, statistical tests and rankings can be easily carried out. The next chapter will showcase the results obtained using the experimental design mentioned in this chapter.

# Chapter 5

# Experimental Results and Analysis

This chapter discusses the results and analysis for the experimental design covered in the previous chapter. We start by discussing the hyperparameter search and present the best-performing hyperparameter values for each algorithm. Then, we report the performance metrics using four different evaluation schemes for the In-Scope dataset. Similarly, we consider the various log files in the Out-of-Scope dataset and evaluate them using the same evaluation framework. A time-based analysis is also performed for the Out-of-Scope log files. This is followed by performing a threshold analysis for different cascade classifiers that we consider. Then we discuss the impact of LoDU and Delimiter Classification Unit on the performance of a few NER classifiers. We then perform various statistical tests on the results obtained. Finally, a synthesis and discussion section is presented.

## 5.1 Hyperparameter Search and Analysis

Table 5.1 shows the various hyperparameters for each algorithm selected using Bayesian Optimization. It also depicts the datatype, search space and selected value for each hyperparameter. Figure 5.1 depicts the hyperparameter importance and contour plots for CRF, Transformer and BERT. The hyperparameter importance signifies the contribution of each hyperparameter in achieving the best performance. In other words, it can be considered a weight that indicates how a hyperparameter directly impacts the performance of a classifier. The top two contributors are selected, and a contour plot is plotted. The contour plot helps map the relationship between the hyperparameters and their impact on the performance. The axes of the contour plot depict the two hyperparameters, whereas the colour of the plot represents either the strict F1 score or loss value. The grey region is the area of our interest, whereas we want to avoid choosing hyperparameters which result in a blue region.

Considering the contour plots, we can observe that Bayesian Optimization has helped us intelligently select the hyperparameters. This is because there are fewer trials in the darker region, whereas most of the trials are present in the grey area of the plot. Another interesting point is that the search space for the hyperparameters in the contour plot has been explored thoroughly. The trials are evenly distributed and are not present in a single region of consideration. For CRF, 'max iterations' is the dominant hyperparameter in terms of importance. As seen in the contour plot B), trials with max iterations of over 150 are located in the grey region. Moving beneath, we observe different shades of blue. Similarly, we select the top two hyperparameters for the transformer: Learning Rate and

No. Encoder/Decoder. From contour plot D), we observe that the left part of the plot is primarily grey, whereas when the learning rate increases, we observe different shades of blue irrespective of the no. of encoder/decoder. For BERT, only the learning rate holds primary importance as a hyperparameter, and hence we do not plot a contour plot.

Table 5.1: Hyperparameter Search Space for All Classifiers.

| **Model** | **Name** | **Datatype** | **Search-Space** | **Selected Value** |
|---|---|---|---|---|
| Naive Bayes | Alpha | Float | low=0, high=5, step=0.1 | 0.1 |
| Perceptron | Penalty | Categorical | ['l1', 'l2'] | l2 |
| | Max Iterations | Int | low=50, high=1000, step=50 | 400 |
| SGD | Loss | Categorical | ['hinge', 'log'] | hinge |
| | Penalty | Categorical | ['l1', 'l2'] | l2 |
| | Max Iterations | Int | low=50, high=1000, step=50 | 400 |
| CRF | C1 | Log Uniform | low=1e-3, high=10 | 0.5075814 |
| | C2 | Log Uniform | low=1e-3, high=10 | 0.0265048 |
| | Max Iterations | Int | low=20, high=200, step=10 | 170 |
| Transformer | Learning Rate | Log Uniform | low=5e-5, high=5e-4 | 0.0001447 |
| | Forward Expansion | Categorical | [4, 64, 128, 256, 512, 1024, 2048] | 2048 |
| | Dropout | Uniform | low=0.1, high=0.3 | 0.2197131 |
| | Embedding Size | Categorical | [128, 256, 512] | 512 |
| | No. Heads | Categorical | [1, 2, 4, 8] | 8 |
| | No. Encoder/Decoder | Int | low=1, high=6, step=1 | 3 |
| BERT, RoBERTa, DistilBERT | Epochs | Int | low=2, high=4, step=1 | 4, 4, 3 |
| | Batch Size | Categorical | [16, 32] | 32, 16, 16 |
| | Learning Rate | Categorical | [3e-4, 1e-4, 5e-5, 3e-5] | 3e-5, 3e-5, 3e-5 |

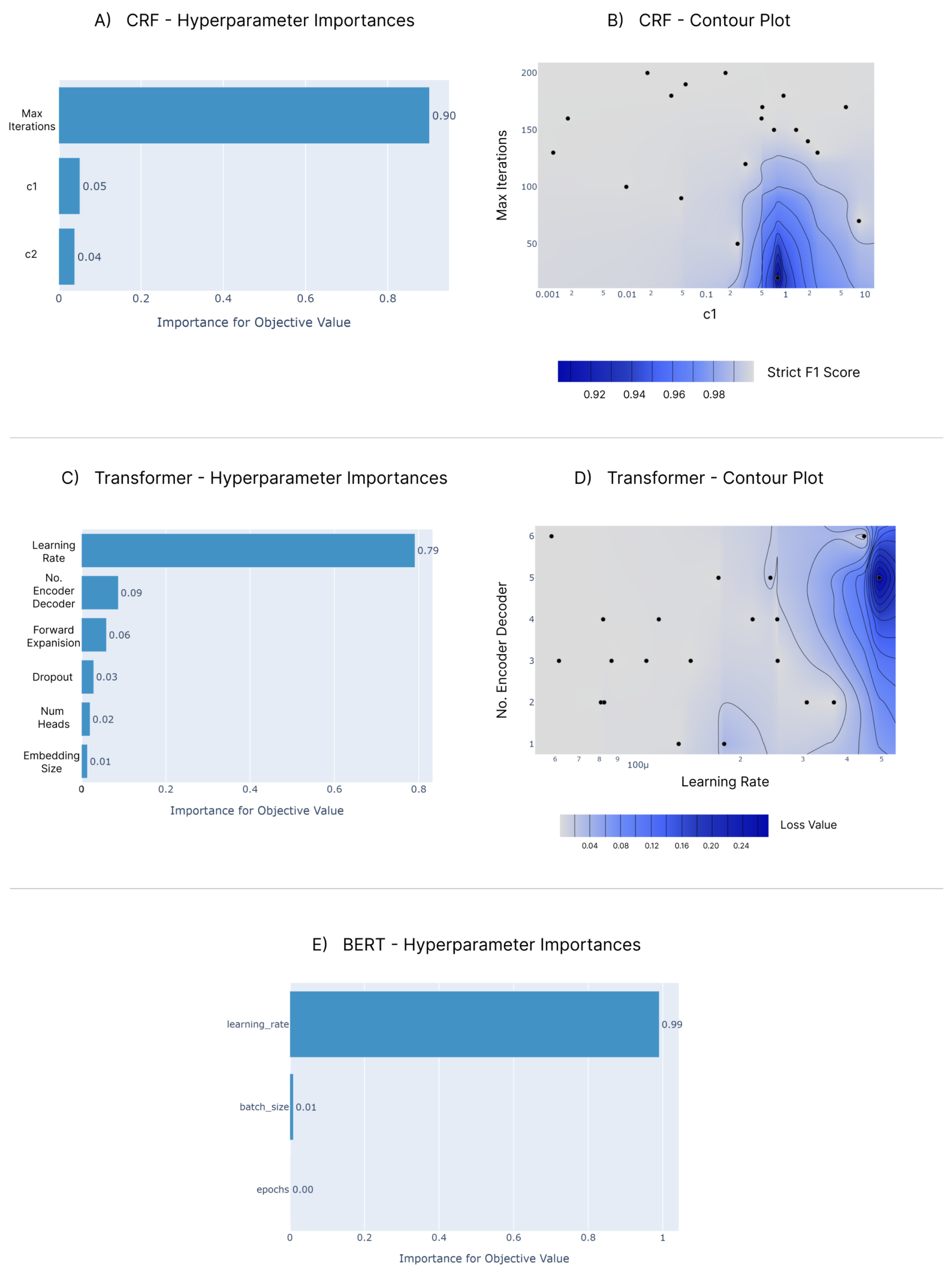


Figure 5.1: Hyperparameter Importances and Contour Plots for CRF, Transformer and BERT

## 5.2 In-Scope Data Results

This section presents the In-Scope data results where we compare the various NER approaches applied to the In-Scope data. Table 5.2 depicts the precision, recall, and F1 score(rounded of to 5 decimal places) for different evaluation schemes across various NER approaches considered.

All the traditional word-based methods did not perform well, having a strict F1 score value of less than 0.66. Perceptron performed the worst of all methodologies with a strict F1 score of 0.35, even worse than the random guessing baseline. In comparison, Naive Bayes and SGD performed slightly better than random guessing, with a strict F1 score greater than 0.6. All the other models, including CRF, Transformer, NERLogParser, Language models, ensembles, and cascades performed exceptionally well with a strict F1 score of over 0.998. The CB Cascade and CBT Cascade perform the best with the highest strict F1 score of 0.99992. This is followed by the Majority Voting Ensemble (MV Ensmbl), Soft Voting Ensemble (SV Ensmbl) and the CT Cascade having a strict F1 score of 0.99991. All the language models achieved a strict F1 score $>$ 0.9996, with BERT having the best performance. Out of all the NER approaches excluding word-based baselines, Transformer showed the least performance with a strict F1 score of 0.99897.

Such high values of strict F1 scores indicate a near-to-perfect NER with just a few errors in predicting entities. Figure 5.2 shows a bar chart comparing the strict F1 scores of the aforementioned NER approaches. Cascades and Ensembles are the two best performing types of NER approaches. We prefer cascading classifiers over ensembles as they balance the trade-off between time and performance. Table 5.2 reports all the evaluation schemes,

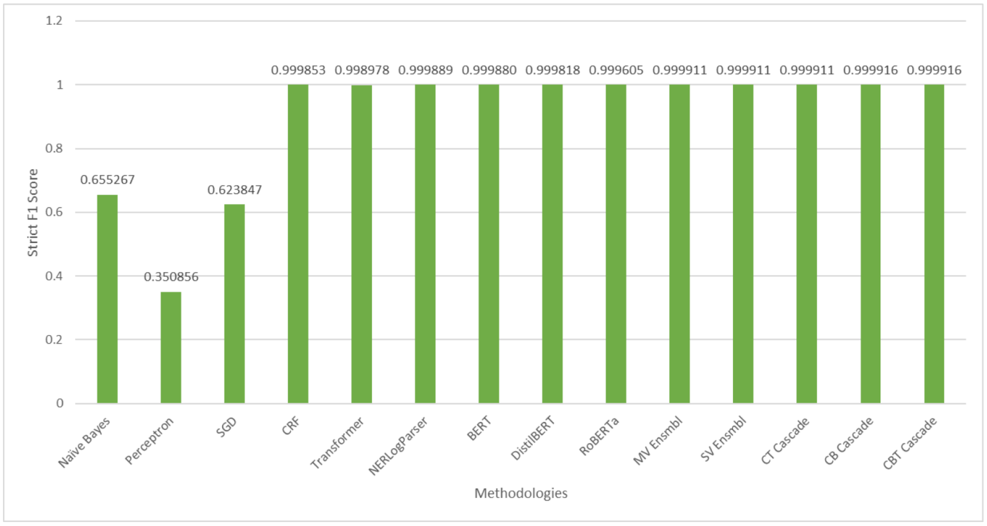


Figure 5.2: Strict F1 scores obtained using various NER approaches for In-Scope test data.

but only the strict evaluation scheme is heavily used. The other evaluation schemes do not play a significant role in use cases where the strict F1 score is close to 1. The other evaluation schemes are considered when we have to dig deeper to determine the areas where the NER approach is not performing well.

## 5.3 Out-of-Scope Data Results

We primarily focus on evaluating the NER approaches on the Out-of-Scope data to understand how well the log augmentation techniques work. All the NER approaches are applied to the different Out-of-Scope log files, and the various metrics are recorded. Table 5.3 depicts the precision, recall and F1 score for the strict evaluation scheme for all the log files present in the Out-of-Scope data. The results are discussed further below.

All the traditional methods perform poorly, having an F1 score of less than 0.1 for all

Table 5.2: Performance results for In-Scope data.

| Model | Metric | Strict | Exact | Partial | Entity-type |
|---|---|---|---|---|---|
| **Naive Bayes** | Precision | 0.70017 | 0.72445 | 0.73292 | 0.71711 |
| | Recall | 0.61576 | 0.63711 | 0.64456 | 0.63066 |
| | F1-score | 0.65526 | 0.67798 | 0.68590 | 0.67111 |
| **Perceptron** | Precision | 0.29729 | 0.46288 | 0.46498 | 0.30151 |
| | Recall | 0.42795 | 0.66631 | 0.66934 | 0.43402 |
| | F1-score | 0.35085 | 0.54627 | 0.54875 | 0.35583 |
| **SGD** | Precision | 0.66569 | 0.71016 | 0.72528 | 0.69593 |
| | Recall | 0.58694 | 0.62615 | 0.63948 | 0.61361 |
| | F1-score | 0.62384 | 0.66552 | 0.67968 | 0.65218 |
| **CRF** | Precision | 0.99976 | 0.99979 | 0.99978 | 0.99976 |
| | Recall | 0.99993 | 0.99996 | 0.99995 | 0.99993 |
| | F1-score | 0.99985 | 0.99987 | 0.99987 | 0.99985 |
| **Transformer** | Precision | 0.99927 | 0.99965 | 0.99970 | 0.99937 |
| | Recall | 0.99868 | 0.99907 | 0.99912 | 0.99879 |
| | F1-score | 0.99897 | 0.99936 | 0.99941 | 0.99908 |
| **NERLogParser** | Precision | 0.99988 | 0.99988 | 0.99988 | 0.99988 |
| | Recall | 0.99989 | 0.99989 | 0.99989 | 0.99989 |
| | F1-score | 0.99989 | 0.99989 | 0.99989 | 0.99989 |
| **BERT** | Precision | 0.99987 | 0.99988 | 0.99988 | 0.99987 |
| | Recall | 0.99988 | 0.99989 | 0.99989 | 0.99988 |
| | F1-score | 0.99988 | 0.99989 | 0.99988 | 0.99988 |
| **RoBERTa** | Precision | 0.99953 | 0.99954 | 0.99955 | 0.99956 |
| | Recall | 0.99967 | 0.99967 | 0.99968 | 0.99969 |
| | F1-score | 0.99960 | 0.99960 | 0.99961 | 0.99963 |
| **DistilBERT** | Precision | 0.99983 | 0.99983 | 0.99983 | 0.99983 |
| | Recall | 0.99980 | 0.99980 | 0.99980 | 0.99980 |
| | F1-score | 0.99981 | 0.99981 | 0.99981 | 0.99981 |
| **MV Ensmbl** | Precision | 0.99992 | 0.99992 | 0.99992 | 0.99992 |
| | Recall | 0.99989 | 0.99989 | 0.99989 | 0.99989 |
| | F1-score | 0.99991 | 0.99991 | 0.99991 | 0.99991 |
| **SV Ensmbl** | Precision | 0.99992 | 0.99992 | 0.99992 | 0.99992 |
| | Recall | 0.99990 | 0.99990 | 0.99990 | 0.99991 |
| | F1-score | 0.99991 | 0.99991 | 0.99991 | 0.99992 |
| **CT Cascade** | Precision | 0.99990 | 0.99991 | 0.99991 | 0.99990 |
| | Recall | 0.99992 | 0.99993 | 0.99993 | 0.99992 |
| | F1-score | 0.99991 | 0.99992 | 0.99992 | 0.99991 |
| **CB Cascade** | Precision | 0.99988 | 0.99988 | 0.99988 | 0.99988 |
| | Recall | 0.99995 | 0.99995 | 0.99995 | 0.99995 |
| | F1-score | 0.99992 | 0.99992 | 0.99992 | 0.99992 |
| **CBT Cascade** | Precision | 0.99989 | 0.99989 | 0.99989 | 0.99989 |
| | Recall | 0.99994 | 0.99994 | 0.99994 | 0.99994 |
| | F1-score | 0.99992 | 0.99992 | 0.99992 | 0.99992 |

the Out-of-Scope log files except the Naive Bayes - Cisco ASA pair. The traditional word-based methods could not parse any of the windows-based log files resulting in an F1 score of 0. Perceptron performs the worse out of all the word-based baselines getting an F1 score of 0 in 6 out of 9 log files. This behaviour is followed by SGD scoring 0 in parsing 4 log files. Since NERLogParser is used as a pre-trained log parsing tool, its performance vary a lot. NERLogParser is unable to parse the windows-based log files obtaining a strict F1 score of 0. The MV Ensmbl, SV Ensmbl, CB Cascade and CBT Cascade models get well adapted to the windows-based log files and perform the best parsing the same. These approaches achieve an F1 score $> 0.96$ and 0.9 for WAE and WSYE, respectively. The WSE log file is not parsed entirely, and all the approaches mentioned above get an F1 score ranging between 0.6 and 0.77. All the individual stand-alone classifiers face challenges scoring a high F1 score for the windows-based log files.

There is no model that best parses all of the Out-of-Scope log files. Different models work best for different log files. A summary of the same is provided below. The transformer neural network works the best for parsing the Cisco ASA log file with an F1 score of $\sim 0.95$. The CB Cascade achieves an F1 score of $\sim 0.84$ for the Linux Secure log file. The NGINX v1 file is best parsed using the CRF model and achieves an F1 score of $\sim 0.83$. None of the classifiers performed well in parsing the NGINX v2 log file, as the highest F1 score achieved is 0.44818 using the NERLogParser. The WAE log file is almost entirely parsed by the CBT Cascade with an F1 score greater than 0.99, making just a few errors in parsing the entities. The RoBERTa language model performs the best in parsing Linux Apache and WSE log files with $\sim 0.97$ and $\sim 0.78$ F1 scores, respectively. Cisco IOS and WSYE log files are best parsed using the MV Ensmbl with an F1 score greater than 0.9. BERT, DistilBERT, SV Ensmbl and CT Cascade did not perform best for any of the individual log files.

Table 5.3: Strict Evaluation for Out-of-Scope Data.

| **Log files** | **Metric** | **Cisco ASA** | **Cisco IOS** | **Linux Secure** | **Linux Apache** | **NGINXx v1** | **NGINX v2** | **WAE** | **WSYE** | **WSE** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Naive Bayes** | Precision | 0.42381 | 0.09362 | 0.08945 | 0.00237 | 0.00445 | 0.05775 | 0 | 0 | 0 |
| | Recall | 0.5 | 0.08510 | 0.04874 | 0.00148 | 0.00135 | 0.04948 | 0 | 0 | 0 |
| | F1-score | 0.45876 | 0.08915 | 0.06310 | 0.00182 | 0.00207 | 0.0533 | 0 | 0 | 0 |
| **Perceptron** | Precision | 0 | 0 | 0.02308 | 0.00293 | 0 | 0.06762 | 0 | 0 | 0 |
| | Recall | 0 | 0 | 0.01338 | 0.00148 | 0 | 0.04948 | 0 | 0 | 0 |
| | F1-score | 0 | 0 | 0.01694 | 0.00197 | 0 | 0.05714 | 0 | 0 | 0 |
| **SGD** | Precision | 0 | 0.01755 | 0.06123 | 0.00293 | 0.00450 | 0.06738 | 0 | 0 | 0 |
| | Recall | 0 | 0.01562 | 0.02947 | 0.00148 | 0.00135 | 0.04948 | 0 | 0 | 0 |
| | F1-score | 0 | 0.01653 | 0.03979 | 0.00197 | 0.00207 | 0.05714 | 0 | 0 | 0 |
| **CRF** | Precision | 0.96463 | 0.64672 | 0.79183 | 0.9374 | 0.77894 | 0.41278 | 0.5009 | 0.56116 | 0.33564 |
| | Recall | 0.5 | 0.65556 | 0.78627 | 0.8781 | 0.88958 | 0.46927 | 1 | 1 | 1 |
| | F1-score | 0.65861 | 0.65111 | 0.78904 | 0.90679 | **0.83059** | 0.43922 | 0.66747 | 0.71890 | 0.502597 |
| **Transformer** | Precision | 0.914553 | 0.85191 | 0.63514 | 0.95965 | 0.70481 | 0.38972 | 0.33333 | 0.33467 | 0.33333 |
| | Recall | 0.981076 | 0.96462 | 0.56342 | 0.93748 | 0.88640 | 0.50838 | 1 | 1 | 1 |
| | F1-score | **0.946648** | 0.90477 | 0.59713 | 0.94844 | 0.78525 | 0.44121 | 0.5 | 0.50150 | 0.5 |
| **NERLogParser** | Precision | 0.66667 | 0.58878 | 0.9999 | 0.97521 | 0 | 0.41787 | 0 | 0 | 0 |
| | Recall | 1 | 0.68887 | 0.52822 | 0.95957 | 0 | 0.48324 | 0 | 0 | 0 |
| | F1-score | 0.8 | 0.63491 | 0.69126 | 0.96733 | 0 | **0.44818** | 0 | 0 | 0. |
| **BERT** | Precision | 0.50432 | 0.70477 | 0.66253 | 0.94601 | 0.44677 | 0.37773 | 0.62534 | 0.48828 | 0.42241 |
| | Recall | 0.5 | 0.82279 | 0.62428 | 0.92132 | 0.89274 | 0.53073 | 1 | 1 | 1 |
| | F1-score | 0.50215 | 0.75922 | 0.64284 | 0.9335 | 0.59552 | 0.44135 | 0.76949 | 0.65616 | 0.59394 |
| **RoBERTa** | Precision | 0.42127 | 0.70173 | 0.62283 | 0.97474 | 0.57604 | 0.36714 | 0.50417 | 0.52420 | 0.64047 |
| | Recall | 0.5 | 0.80900 | 0.57493 | 0.9695 | 0.88743 | 0.49589 | 1 | 1 | 1 |
| | F1-score | 0.45727 | 0.75156 | 0.59792 | **0.97211** | 0.69861 | 0.42191 | 0.67036 | 0.68783 | **0.78084** |
| **DistilBERT** | Precision | 0.31361 | 0.32528 | 0.25008 | 0.8965 | 0.3881 | 0.35141 | 0.40595 | 0.39856 | 0.46026 |
| | Recall | 0.5 | 0.55144 | 0.25053 | 0.89269 | 0.62763 | 0.48883 | 1 | 1 | 1 |
| | F1-score | 0.38545 | 0.40919 | 0.25031 | 0.89459 | 0.47962 | 0.40888 | 0.57747 | 0.56996 | 0.63038 |
| **MV Ensmbl** | Precision | 1 | 0.94450 | 0.81533 | 0.95652 | 0.73350 | 0.39651 | 0.9336 | 0.98437 | 0.49875 |
| | Recall | 0.5 | 0.93040 | 0.72451 | 0.93180 | 0.90754 | 0.50838 | 1 | 1 | 1 |
| | F1-score | 0.66667 | **0.93739** | 0.76724 | 0.944 | 0.81129 | 0.44553 | 0.96566 | **0.99222** | 0.66555 |
| **SV Ensmbl** | Precision | 0.98623 | 0.84612 | 0.80107 | 0.95873 | 0.62473 | 0.38477 | 0.95868 | 0.88183 | 0.61690 |
| | Recall | 0.50827 | 0.90180 | 0.77432 | 0.9359 | 0.90573 | 0.52235 | 1 | 1 | 1 |
| | F1-score | 0.67082 | 0.87307 | 0.78747 | 0.94718 | 0.73943 | 0.44313 | 0.9789 | 0.93720 | 0.76306 |
| **CT Cascade** | Precision | 0.96349 | 0.64105 | 0.78930 | 0.94568 | 0.6997 | 0.38771 | 0.52727 | 0.57714 | 0.42959 |
| | Recall | 0.77262 | 0.68898 | 0.77749 | 0.92263 | 0.88568 | 0.51117 | 1 | 1 | 0.99966 |
| | F1-score | 0.85756 | 0.66415 | 0.78335 | 0.93401 | 0.78178 | 0.44096 | 0.69047 | 0.73188 | 0.60094 |
| **CB Cascade** | Precision | 0.99529 | 0.67423 | 0.83554 | 0.95654 | 0.50322 | 0.386503 | 0.967338 | 0.89874 | 0.51203 |
| | Recall | 0.5 | 0.68554 | 0.85512 | 0.93323 | 0.906116 | 0.52793 | 1 | 1 | 1 |
| | F1-score | 0.66561 | 0.679841 | **0.84521** | 0.94474 | 0.64708 | 0.44628 | 0.98339 | 0.94667 | 0.67727 |
| **CBT Cascade** | Precision | 0.99562 | 0.67783 | 0.83181 | 0.95971 | 0.51463 | 0.38383 | 0.98444 | 0.84033 | 0.42863 |
| | Recall | 0.53814 | 0.71092 | 0.85811 | 0.93631 | 0.90531 | 0.53072 | 1 | 1 | 1 |
| | F1-score | 0.69865 | 0.69398 | 0.84476 | 0.94786 | 0.65622 | 0.44548 | **0.99216** | 0.91324 | 0.60006 |

Figure 5.3 depicts the box plot for strict F1 scores across various NER approaches. We use the box plot to compare the Medians, Inter Quartile Range (IQR) and the overall distribution of strict F1 scores. The traditional methods have a really low strict F1 score,

so we do not consider them for comparison. The MV Ensmbl and SV Ensmbl have a median value of $\sim 0.8$, whereas, for the cascading classifiers, it is $\sim 0.7$. The median for NERLogParser and DistilBERT lies between 0.4 and 0.5. The individual classifiers, including CRF, transformer and language models, have a median value of $< 0.7$. The CT Cascade has the smallest IQR, but it does not have the highest performing median across all the other approaches; and hence it cannot be considered the best. The IQR for all the language models is somewhat similar, with RoBERTa being the best performing and DistilBERT the worst. NERLogParser has the highest IQR, making it the most unstable approach. We also consider the extreme values and the length of whiskers to understand the overall spread of F1 scores. The lower end of the lower whisker denotes the minimum value. For most of the approaches, the lower end of the whisker is positioned between 0.4 and 0.5. This is because of all the approaches' inability to parse the NGINX v2 file. The overall spread of F1 scores is similar for majority of the NER approaches except the CT Cascade, CRF, DistilBERT and NERLogParser.

We have been using the strict evaluation scheme to compare the NER systems. The other evaluation schemes can not be directly used to compare the approaches. Still, they can provide a complementary set of metrics which help us understand and improve the systems that are not performing well. Here, we discuss the exact evaluation scheme and how we use it to understand the internals of a NER system. Table 5.4 shows the precision, recall and F1 score calculated using the exact evaluation scheme for different NER approaches across all the Out-of-Scope log files. The exact evaluation scheme calculates the precision, recall and F1 score by only considering token matching, irrespective of the entity-type matching. A high strict F1 score would also mean that we have a high exact F1 score. On

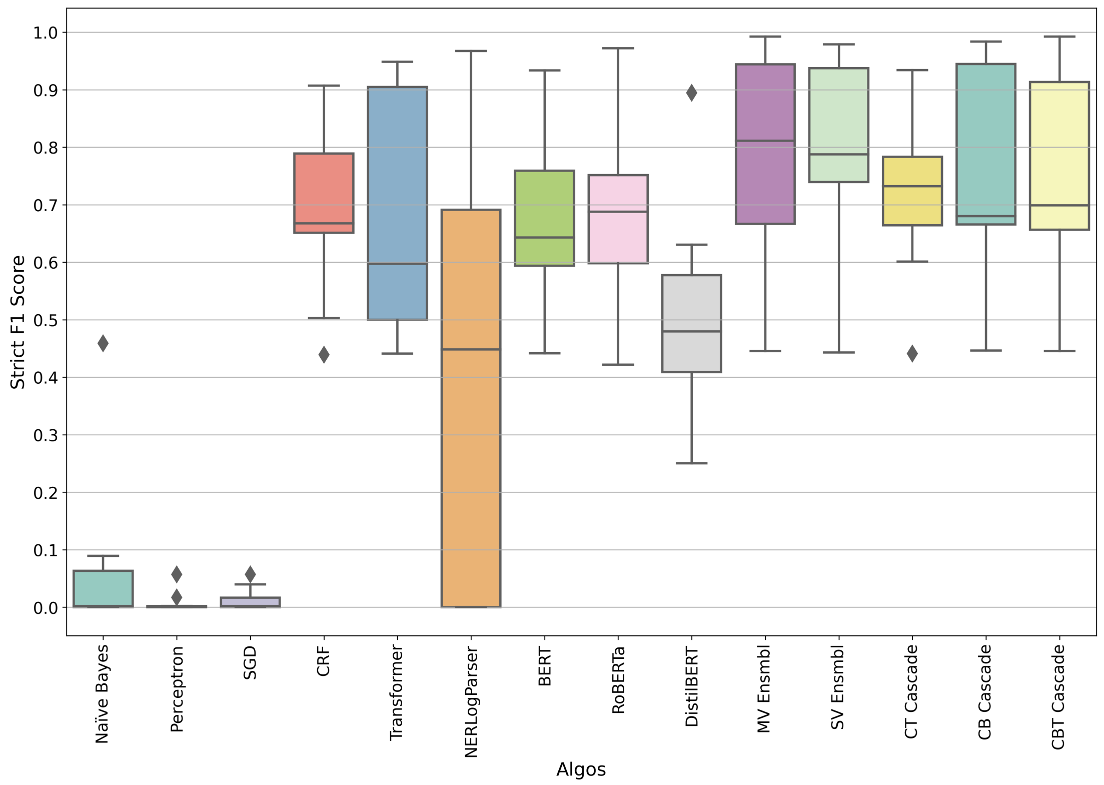


Figure 5.3: Box Plot for Strict F1 scores.

the other hand, the reverse does not hold. We will now look at the cases where the strict F1 score is low, but a high exact F1 score is present. For the Cisco IOS log file, CRF, CT Cascade, CB Cascade and CBT Cascade have a strict F1 score below 0.7. At the same time, the exact F1 score for all the approaches mentioned above is above 0.95. Similarly, for Linux Secure log file, the CB Cascade and the CBT Cascade have a strict F1 score of 0.84, but the exact metric score is 0.97.

Such high values of exact F1 score suggest that the token grouping and matching occurs ideally with just a few boundary mismatches. When the high exact F1 scores complement the low strict F1 scores, we understand that the entity prediction is not perfect. This scenario is generally reflected in the entity-type F1 score (table present in the Appendix section). Cases with high exact F1 scores and low entity-type scores could also mean that the NER approach has learned to mispredict an Entity A as Entity B. For example, a timestamp can always be predicted as an IP Address. This boosts exact evaluation metrics but dips the entity-type evaluation metrics, leading to an overall drop in the strict F1 score. Partial and entity-type evaluation schemes can be used similarly and help make multiple performance inferences; tables depicting these schemes are present in the appendix section.

The time required to parse the log files is also an essential factor in selecting NER approaches. Different use-cases allow different timing thresholds, and thus it is crucial to keep timing into account. Table 5.5 shows the inference times of all the NER approaches across all the Out-of-Scope log files. All the timing values mentioned in the table are in seconds. Timing precision of up to milliseconds has not been considered because of the gigantic difference between the inference times of different NER approaches. CRF achieves the least inference time across all the Out-of-Scope log files. On the other hand, SV Ensmbl takes the most time because of the sequential processing of three different approaches. The CRF is almost 100 times faster than the ensembles in some cases. For example, CRF takes 10 seconds to parse the complete WAE file, whereas the SV Ensmbl takes 1018 seconds. The overall trend of inference times that can be observed in the table is as follows: CRF < Language Models < Traditional Methods < NERLogParser < Cascades < Transformer < Ensembles.

Table 5.4: Exact Evaluation for Out-of-Scope Data.

| **Log files** | **Metric** | **Cisco ASA** | **Cisco IOS** | **Linux Secure** | **Linux Apache** | **NGINX v1** | **NGINX v2** | **WAE** | **WSYE** | **WSE** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Naive Bayes** | Precision | 0.42381 | 0.09362 | 0.12411 | 0.04359 | 0.46983 | 0.21580 | 0 | 0 | 0 |
| | Recall | 0.5 | 0.08510 | 0.06763 | 0.02723 | 0.14232 | 0.18489 | 0 | 0 | 0 |
| | F1-score | 0.45876 | 0.08915 | 0.08755 | 0.03352 | 0.21847 | 0.19915 | 0 | 0 | 0 |
| **Perceptron** | Precision | 0 | 0.0061 | 0.06818 | 0.02838 | 0.47504 | 0.15302 | 0 | 0 | 0 |
| | Recall | 0 | 0.00815 | 0.03953 | 0.01436 | 0.14232 | 0.11197 | 0 | 0 | 0 |
| | F1-score | 0 | 0.00698 | 0.05005 | 0.01907 | 0.21903 | 0.12932 | 0 | 0 | 0 |
| **SGD** | Precision | 0 | 0.01755 | 0.10048 | 0.02837 | 0.47504 | 0.15248 | 0 | 0 | 0 |
| | Recall | 0 | 0.01562 | 0.04836 | 0.01436 | 0.14232 | 0.11197 | 0 | 0 | 0 |
| | F1-score | 0 | 0.01653 | 0.06529 | 0.01907 | 0.21903 | 0.12912 | 0 | 0 | 0 |
| **CRF** | Precision | 0.96463 | 0.98629 | 0.84631 | 0.95226 | 0.78049 | 0.48157 | 0.5009 | 0.56116 | 0.33564 |
| | Recall | 0.5 | 0.99977 | 0.84036 | 0.89202 | 0.89135 | 0.54748 | 1 | 1 | 1 |
| | F1-score | 0.65861 | 0.99298 | 0.84332 | 0.92116 | 0.83224 | 0.51241 | 0.66747 | 0.71890 | 0.502597 |
| **Transformer** | Precision | 0.91455 | 0.85191 | 0.89261 | 0.97647 | 0.70791 | 0.44967 | 0.33333 | 0.33467 | 0.33333 |
| | Recall | 0.98107 | 0.96462 | 0.79181 | 0.95392 | 0.89029 | 0.58659 | 1 | 1 | 1 |
| | F1-score | 0.94664 | 0.90477 | 0.83919 | 0.96507 | 0.7887 | 0.50909 | 0.5 | 0.50150 | 0.5 |
| **NERLogParser** | Precision | 0.66667 | 0.58878 | 0.9999 | 0.98086 | 0.34122 | 0.4855 | 0 | 0 | 0 |
| | Recall | 1 | 0.68887 | 0.52822 | 0.96513 | 0.14387 | 0.56145 | 0 | 0 | 0 |
| | F1-score | 0.8 | 0.63491 | 0.69126 | 0.97293 | 0.2024 | 0.52072 | 0 | 0 | 0 |
| **BERT** | Precision | 0.50432 | 0.70605 | 0.84768 | 0.97262 | 0.44770 | 0.4334 | 0.62534 | 0.48828 | 0.42241 |
| | Recall | 0.5 | 0.82428 | 0.79874 | 0.94723 | 0.89460 | 0.60893 | 1 | 1 | 1 |
| | F1-score | 0.50215 | 0.76060 | 0.82248 | 0.95976 | 0.59676 | 0.50638 | 0.76949 | 0.65616 | 0.59394 |
| **RoBERTa** | Precision | 0.42127 | 0.70233 | 0.73626 | 0.99217 | 0.58089 | 0.42393 | 0.50417 | 0.52420 | 0.64047 |
| | Recall | 0.5 | 0.80969 | 0.67964 | 0.98684 | 0.89489 | 0.57260 | 1 | 1 | 1 |
| | F1-score | 0.45727 | 0.75220 | 0.70682 | 0.9895 | 0.70448 | 0.48717 | 0.67036 | 0.68783 | 0.78084 |
| **DistilBERT** | Precision | 0.62296 | 0.33550 | 0.59642 | 0.93824 | 0.44730 | 0.43172 | 0.40595 | 0.39856 | 0.46026 |
| | Recall | 0.99319 | 0.56878 | 0.59749 | 0.93425 | 0.72338 | 0.60055 | 1 | 1 | 1 |
| | F1-score | 0.76567 | 0.42205 | 0.59695 | 0.93624 | 0.55279 | 0.50233 | 0.57747 | 0.56996 | 0.63038 |
| **MV Ensmbl** | Precision | 1 | 0.94450 | 0.93743 | 0.97046 | 0.73485 | 0.45751 | 0.9336 | 0.98437 | 0.49875 |
| | Recall | 0.5 | 0.93040 | 0.833 | 0.94538 | 0.90921 | 0.58659 | 1 | 1 | 1 |
| | F1-score | 0.66667 | 0.93739 | 0.88214 | 0.95775 | 0.81279 | 0.51407 | 0.96566 | 0.99222 | 0.66555 |
| **SV Ensmbl** | Precision | 0.98623 | 0.85592 | 0.93267 | 0.97326 | 0.62604 | 0.44238 | 0.95868 | 0.88183 | 0.61690 |
| | Recall | 0.50827 | 0.91225 | 0.90153 | 0.95008 | 0.90763 | 0.60055 | 1 | 1 | 1 |
| | F1-score | 0.67082 | 0.88319 | 0.91684 | 0.96153 | 0.74099 | 0.50947 | 0.9789 | 0.93720 | 0.76306 |
| **CT Cascade** | Precision | 0.96607 | 0.92274 | 0.88435 | 0.96071 | 0.70829 | 0.44915 | 0.52727 | 0.57714 | 0.42959 |
| | Recall | 0.77469 | 0.99173 | 0.87112 | 0.93730 | 0.89656 | 0.59217 | 1 | 1 | 0.99966 |
| | F1-score | 0.85986 | 0.95599 | 0.87768 | 0.94886 | 0.79139 | 0.51084 | 0.69047 | 0.73188 | 0.60094 |
| **CB Cascade** | Precision | 0.99529 | 0.97232 | 0.95966 | 0.97112 | 0.50423 | 0.44376 | 0.967338 | 0.89874 | 0.51203 |
| | Recall | 0.5 | 0.98863 | 0.98215 | 0.94745 | 0.90792 | 0.60614 | 1 | 1 | 1 |
| | F1-score | 0.66561 | 0.98041 | 0.97077 | 0.95914 | 0.64837 | 0.51239 | 0.98339 | 0.94667 | 0.67727 |
| **CBT Cascade** | Precision | 0.99562 | 0.92860 | 0.95553 | 0.97355 | 0.51566 | 0.4404 | 0.98444 | 0.84033 | 0.42863 |
| | Recall | 0.53814 | 0.97392 | 0.98574 | 0.94982 | 0.90712 | 0.60893 | 1 | 1 | 1 |
| | F1-score | 0.69865 | 0.95072 | 0.97040 | 0.96154 | 0.65753 | 0.51113 | 0.99216 | 0.91324 | 0.60006 |

### 5.3.1 Ranking Results

Different approaches worked well for different log files in the Out-of-Scope dataset. We did not have a single NER approach that performed the best for all the Out-of-Scope log files. We also did not observe any single approach doing well for a majority of log files. Hence, we perform ranking on the strict F1 scores to understand which approach(es) performs

Table 5.5: Inference times of NER approaches for Out-of-Scope Log Files.

| | Cisco ASA | Cisco IOS | Linux Secure | Linux Apache | NGINX v1 | NGINX v2 | WAE | WSYE | WSE |
|---|---|---|---|---|---|---|---|---|---|
| Naïve Bayes | 36 | 60 | 53 | 56 | 61 | 0 | 23 | 58 | 57 |
| Perceptron | 35 | 58 | 50 | 54 | 48 | 2 | 36 | 68 | 66 |
| SGD | 31 | 58 | 61 | 62 | 62 | 1 | 25 | 50 | 53 |
| CRF | 8 | 15 | 17 | 15 | 24 | 0 | 10 | 21 | 37 |
| Transformer | 213 | 339 | 508 | 391 | 604 | 6 | 418 | 1127 | 1463 |
| NERLogParser | 49 | 62 | 110 | 159 | 179 | 1 | 72 | 159 | 285 |
| BERT | 12 | 23 | 28 | 30 | 33 | 0 | 11 | 32 | 31 |
| RoBERTa | 11 | 23 | 27 | 27 | 28 | 0 | 11 | 28 | 30 |
| DistilBERT | 11 | 19 | 23 | 24 | 26 | 0 | 11 | 26 | 34 |
| MV Ensmbl | 456 | 765 | 1079 | 827 | 1576 | 13 | 953 | 1924 | 2333 |
| SV Ensmbl | 465 | 777 | 1117 | 853 | 1689 | 13 | 1018 | 2081 | 2560 |
| CT Cascade | 35 | 73 | 88 | 115 | 201 | 1 | 55 | 106 | 284 |
| CB Cascade | 200 | 183 | 346 | 358 | 356 | 5 | 170 | 363 | 382 |
| CBT Cascade | 249 | 423 | 434 | 444 | 492 | 7 | 210 | 478 | 786 |

the best. We follow an average ranking method that includes assigning an average rank whenever there is a tie. Table 5.6 shows the ranking results of various NER approaches across the Out-of-Scope log files. We compute an average rank and standard deviation of ranks using which we compare all the approaches. We also plot line charts depicting the ranks of various approaches, as shown in Figure 5.4.

The MV Ensmbl performs the best with an average rank of 3.722. The SV Ensmbl follows with an average rank of 3.889. Comparing the spread of ranks using the line chart, we can say that although the average rank of the MV Ensmbl is less, the SV Ensmbl is more stable. Stability is even more evident while comparing the standard deviation values; SV Ensmbl has a standard deviation of 1.099, whereas the MV Ensmbl seems to be a bit

Table 5.6: Ranking of NER approaches for Out-of-Scope Log Files.

| | Cisco ASA | Cisco IOS | Linux Secure | Linux Apache | NGINX v1 | NGINX v2 | WAE | WSYE | WSE | avg_rank | std |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Naïve Bayes | 10.0 | 12.0 | 12.0 | 13.0 | 11.5 | 14.0 | 12.5 | 12.5 | 12.5 | 12.222222 | 1.030402 |
| Perceptron | 13.5 | 14.0 | 14.0 | 13.0 | 13.5 | 12.5 | 12.5 | 12.5 | 12.5 | 13.111111 | 0.613631 |
| SGD | 13.5 | 13.0 | 13.0 | 13.0 | 11.5 | 12.5 | 12.5 | 12.5 | 12.5 | 12.666667 | 0.527046 |
| CRF | 8.0 | 9.0 | 3.0 | 10.0 | 1.0 | 9.0 | 8.0 | 6.0 | 9.0 | 7.000000 | 2.905933 |
| Transformer | 1.0 | 2.0 | 10.0 | 3.5 | 3.0 | 7.0 | 10.0 | 10.0 | 10.0 | 6.277778 | 3.659926 |
| NERLogParser | 3.0 | 10.0 | 7.0 | 2.0 | 13.5 | 1.0 | 12.5 | 12.5 | 12.5 | 8.222222 | 4.779070 |
| BERT | 9.0 | 4.0 | 8.0 | 8.5 | 9.0 | 7.0 | 5.0 | 8.0 | 8.0 | 7.388889 | 1.662959 |
| RoBERTa | 11.0 | 5.0 | 9.0 | 1.0 | 6.0 | 10.0 | 7.0 | 7.0 | 1.0 | 6.333333 | 3.366502 |
| DistilBERT | 12.0 | 11.0 | 11.0 | 11.0 | 10.0 | 11.0 | 9.0 | 9.0 | 5.0 | 9.888889 | 1.968894 |
| MV Ensmbl | 6.0 | 1.0 | 6.0 | 7.0 | 2.0 | 2.5 | 4.0 | 1.0 | 4.0 | 3.722222 | 2.122775 |
| SV Ensmbl | 5.0 | 3.0 | 4.0 | 5.0 | 5.0 | 5.0 | 3.0 | 3.0 | 2.0 | 3.888889 | 1.099944 |
| CT Cascade | 2.0 | 8.0 | 5.0 | 8.5 | 4.0 | 7.0 | 6.0 | 5.0 | 6.0 | 5.722222 | 1.901916 |
| CB Cascade | 7.0 | 7.0 | 1.5 | 6.0 | 8.0 | 2.5 | 2.0 | 2.0 | 3.0 | 4.333333 | 2.460804 |
| CBT Cascade | 4.0 | 6.0 | 1.5 | 3.5 | 7.0 | 4.0 | 1.0 | 4.0 | 7.0 | 4.222222 | 2.029109 |

unstable with a value of 2.122. Traditional methods performed the worst and achieved an average rank of over 12. Out of all the language models, RoBERTa performed the best with an average rank of 6.333, whereas DistilBERT performed the worst, having an average rank 9.889. BERT achieved an average rank of 7.389, worse than RoBERTa. Comparing BERT and RoBERTa using the line chart, we can say that BERT is more stable than RoBERTa as the rank spread is wide for RoBERTa than BERT. Even the standard deviation of RoBERTa is double that of BERT. NERLogParser is the most unstable approach as it achieves a standard deviation of 4.779 and ranks ranging between 1 and 13.5. Cascading classifiers show promising results. The CBT Cascade achieves an average rank of 4.222, close to both the ensembles. The CB Cascade and CT Cascade got an average rank of 4.333 and 5.722, respectively. Overall, the cascading classifiers are less stable than the ensembles.

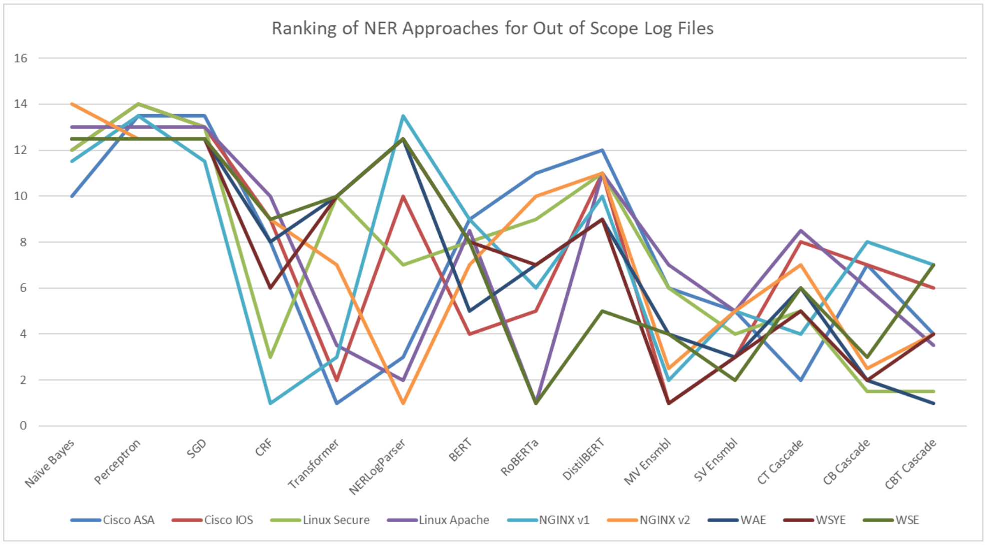


Figure 5.4: Ranking of NER Approaches for Out-of-Scope Log Files.

Table 5.7: Ranking of inference times of NER approaches for Out-of-Scope Log Files.

| | Cisco ASA | Cisco IOS | Linux Secure | Linux Apache | NGINXv1 | NGINX v2 | WAE | WSYE | WSE | Avg_rank | Std Dev |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Naïve Bayes | 8 | 7 | 6 | 6 | 6 | 3 | 5 | 6 | 6 | 5.888889 | 1.364225 |
| SGD | 5 | 5.5 | 7 | 7 | 7 | 7 | 6 | 5 | 5 | 6.055556 | 0.950146 |
| Perceptron | 6.5 | 5.5 | 5 | 5 | 5 | 9 | 7 | 7 | 7 | 6.333333 | 1.346291 |
| CRF | 1 | 1 | 1 | 1 | 1 | 3 | 1 | 1 | 4 | 1.555556 | 1.130388 |
| Transformer | 11 | 11 | 12 | 11 | 12 | 11 | 12 | 12 | 12 | 11.555556 | 0.527046 |
| NERLogParser | 9 | 8 | 9 | 8 | 8 | 7 | 9 | 9 | 9 | 8.444444 | 0.726483 |
| BERT | 4 | 3.5 | 4 | 4 | 4 | 3 | 3 | 4 | 2 | 3.500000 | 0.707107 |
| RoBERTa | 2.5 | 3.5 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 2.777778 | 0.712000 |
| DistilBERT | 2.5 | 2 | 2 | 2 | 2 | 3 | 3 | 2 | 3 | 2.388889 | 0.485913 |
| MV Ensmbl | 13 | 13 | 13 | 13 | 13 | 13.5 | 13 | 13 | 13 | 13.055556 | 0.166667 |
| SV Ensmbl | 14 | 14 | 14 | 14 | 14 | 13.5 | 14 | 14 | 14 | 13.944444 | 0.166667 |
| CT Cascade | 6.5 | 9 | 8 | 9 | 9 | 7 | 8 | 8 | 8 | 8.055556 | 0.881917 |
| CB Cascade | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10.000000 | 0.000000 |
| CBT Cascade | 12 | 12 | 11 | 12 | 11 | 12 | 11 | 11 | 11 | 11.444444 | 0.527046 |

We also perform ranking on the inference times of the various NER approaches. Table 5.7 shows the ranks of various NER approaches' inference times with their average ranks

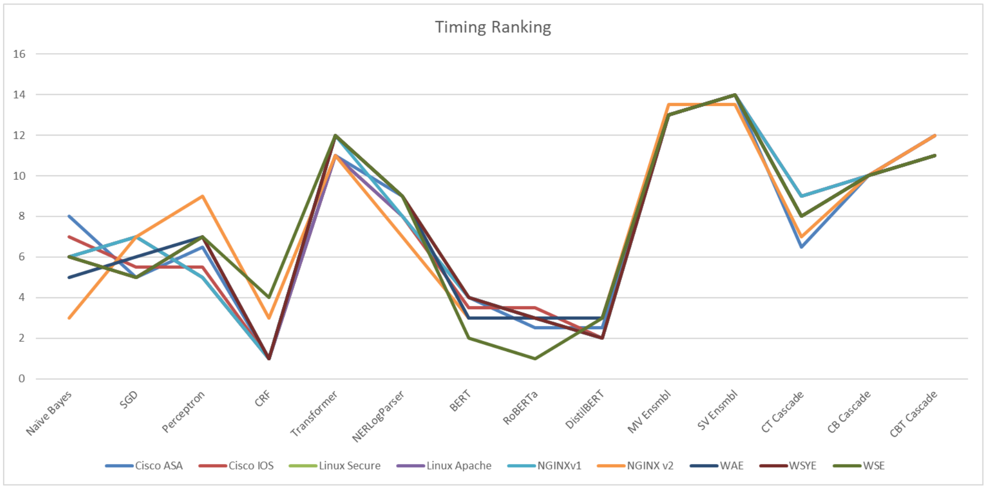


Figure 5.5: Ranking of Inference Time of NER Approaches for Out-of-Scope Log Files.

and standard deviations. Figure 5.5 complements the data shown in Table 5.7 by plotting a line chart for the same. Ranks are not widely spread and do not vary a lot across the log files. The line chart for the inference times' ranks is relatively uniform compared to the line chart of strict F1 score. CRF achieves a rank of 1 for every Out-of-Scope log file except the WSE; the average rank for CRF is 1.55. In comparison, the SV Ensmbl has the highest average rank of 13.944. Looking at Figure 5.5, we observe a general trend in inference time rank as follows: CRF $<$ Language models $<$ Traditional Methods $<$ NERLogParser $<$ Cascading Classifiers $<$ Transformer $<$ Ensembles.

Cascades depend on a threshold-based policy, which governs the transfer of control from one classifier to the other. The cascades used in this section were the best performing cascades out of a group of cascades formed using different thresholds. We discussed the best performing cascades in the results but did not highlight the threshold selection process. A threshold-based analysis for all three cascades is present in the following section, where we evaluate and compare the various cascades formed using different values of thresholds.

## 5.4 Cascade Threshold Analysis

As discussed in Chapter 3, cascading classifiers consist of multiple individual models and work by transferring control from one model to another based on a pre-defined threshold-based policy. This section discusses and analyses the effect of various threshold values on the performance of the cascading classifiers.

### 5.4.1 In-Scope Data - Cascade Threshold Analysis

We consider three different cascade classifiers, including the following combinations: 1) CRF + BERT + Transformer (CBT Cascade), 2) CRF + Transformer (CT Cascade), and 3) CRF + BERT (CB Cascade). We experiment with 11 threshold values for each of the cascading classifiers mentioned above. As a result we have a total of 33 cascade models (11 for each combination) including the following thresholds: [0, 0.25, 0.5, 0.6, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1]. Figure 5.6 depicts the relationship between the strict F1 score and inference time of the different cascading classifiers built with the abovementioned thresholds. The points plotted over the line chart represent the threshold values.

As we increase the threshold from 0 to 0.95, we observe an increase in the strict F1 score but not a significant increase in the inference time. On the other hand, we notice a plateau formation when we move from 0.95 to 1. There is no significant difference between the strict F1 score obtained using 0.95 and 1 threshold values. The inference time shoots up

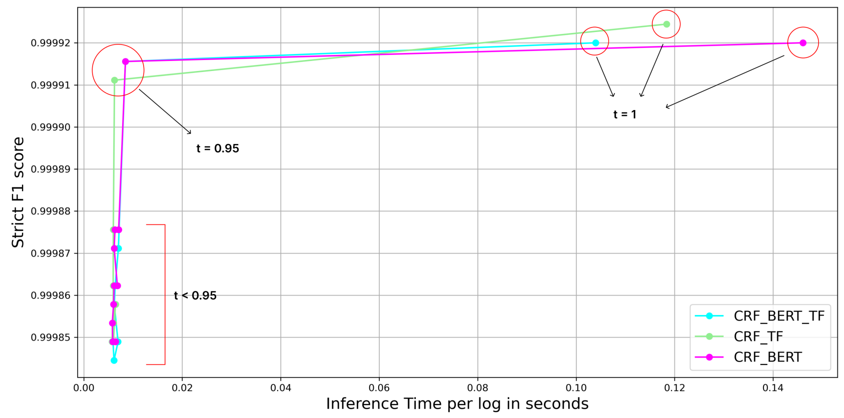


Figure 5.6: In-Scope Data - Cascade Performance Result

as we approach a threshold of 1 as the cascade model converges to a soft voting classifier. The performance of all the cascading classifiers peak at $t = 1$ but we choose $t = 0.95$ as the inference time is approximately ten times lesser.

### 5.4.2 Out-of-Scope Data - Cascade Threshold Analysis

Once we are done finding the best-performing thresholds using the In-Scope dataset, we further decide to conduct the same experiments on the Out-of-Scope dataset to confirm whether the threshold behaviour is identical across the datasets. We plot line graphs depicting the strict F1 scores and inference times for different Out-of-Scope log files; Figure 5.7 represents the same for the CT Cascade. The points over the line graph represent the various thresholds, and the threshold with a red marker denotes the peak threshold point. The behaviour of the cascade models varies tremendously across all the nine Out-of-Scope log files. It is challenging to generalize whether the performance experiences an upward or a downward trend as the threshold increases.

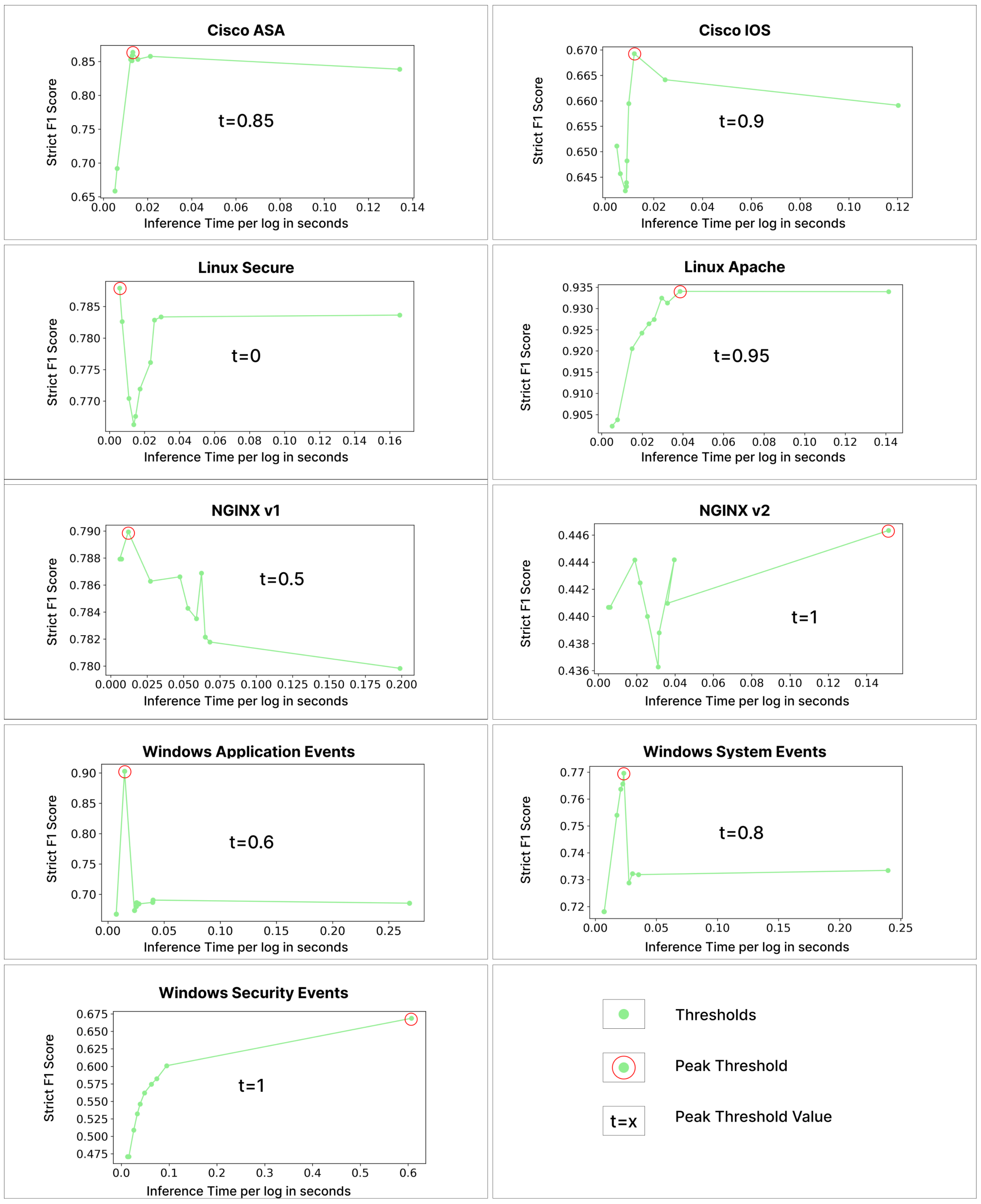


Figure 5.7: Out-of-Scope Data - CRF-Transformer Cascade Performance Result

Cisco ASA, Cisco IOS, Linux Apache and Windows System Events (WSYE) have a

Table 5.8: Out-of-Scope Data: CRF-Transformer Cascade Threshold Analysis

| | Cisco ASA | Cisco IOS | Linux Secure | Linux Apache | NGINX v1 | NGINX v2 | WAE | WSYE | WSE | Avg Rank | Std Dev |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0.00** | 0.65862 | 0.65112 | **0.78793** | 0.90232 | 0.78793 | 0.44068 | 0.66747 | 0.71813 | 0.47099 | 7.500000 | 3.570714 |
| **0.25** | 0.65862 | 0.65112 | **0.78793** | 0.90232 | 0.78793 | 0.44068 | 0.66747 | 0.71813 | 0.47099 | 7.500000 | 3.570714 |
| **0.50** | 0.69194 | 0.64567 | 0.78263 | 0.90381 | **0.78994** | 0.44068 | 0.66747 | 0.71813 | 0.47099 | 7.777778 | 2.905933 |
| **0.60** | 0.85306 | 0.64232 | 0.77042 | 0.92055 | 0.78629 | 0.44416 | **0.90272** | 0.75399 | 0.50878 | 6.222222 | 3.153481 |
| **0.70** | 0.85866 | 0.64316 | 0.76627 | 0.92420 | 0.78662 | 0.44248 | 0.67328 | 0.76369 | 0.53229 | 6.333333 | 3.082207 |
| **0.75** | 0.85490 | 0.64391 | 0.76759 | 0.92640 | 0.78428 | 0.44000 | 0.68003 | 0.76564 | 0.54620 | 6.666667 | 2.549510 |
| **0.80** | 0.85129 | 0.64821 | 0.77192 | 0.92742 | 0.78351 | 0.43627 | 0.68622 | **0.76961** | 0.56201 | 6.222222 | 2.862594 |
| **0.85** | **0.86361** | 0.65948 | 0.77612 | 0.93244 | 0.78689 | 0.43879 | 0.68420 | 0.72880 | 0.57438 | 5.111111 | 2.848001 |
| **0.90** | 0.85323 | **0.66927** | 0.78285 | 0.93129 | 0.78215 | 0.44417 | 0.68673 | 0.73224 | 0.58224 | 4.222222 | 2.386304 |
| **0.95** | 0.85756 | 0.66416 | 0.78335 | **0.93402** | 0.78178 | 0.44096 | 0.69048 | 0.73189 | 0.60094 | **4.000000** | 2.915476 |
| **1.00** | 0.83854 | 0.65912 | 0.78364 | 0.93394 | 0.77983 | **0.44634** | 0.68540 | 0.73346 | **0.66875** | 4.444444 | 3.320810 |

peak threshold value ranging between [0.8, 0.95]. The best-performing threshold for Linux Secure is t=0. On the other hand, it is t=1 for NGINX v2 and Windows Security Events (WSE) log files. NGINX v1 and Windows Application Events (WAE) log files have a peak threshold of 0.5 and 0.6, respectively. It isn't easy to visually judge and select a common threshold value that works well for all the log files present in the Out-of-Scope dataset. Hence, we rank thresholds across various Out-of-Scope log files and compute an average rank for each threshold point. Table 5.8 shows the strict F1 scores for various thresholds across different Out-of-Scope log files; it also shows the average rank and the standard deviation of rank across multiple files. Threshold t=0.95 performs the best with an average rank of 4.0, whereas t=0 and 0.25 perform the worst with an average rank of 7.5.

The same set of experiments are performed using the CBT Cascade. Compared to the

CT Cascade, a completely different set of best-performing thresholds is obtained in these experiments. Figure 5.8 shows the strict F1 scores vs inference time trend as the threshold value increases from 0 to 1. By visual judgement and analysis, we cannot generalize the trend being followed across all the log files. We notice the formation of different line graph shapes. The ideal form that is expected is a plateau such that the strict F1 score saturates as we reach a threshold of 1 and the time keeps increasing. Cisco ASA, Cisco IOS, Linux Secure, and WSYE log files experience a hill formation. NGINX v1 and v2 log files face a decrease followed by an increase in strict F1 score resulting in a valley-like shape. We also notice plateaus in the case of Linux Apache and WAE logs. Hence, we cannot select a standard threshold that best performs across all the considered log files.

As a result, we again perform the exact ranking mechanism. The threshold with the minimum average rank across all the log files is selected as the best-performing value. Table 5.9 depicts the strict F1 scores and the average ranks of the cascade models trained with different threshold values for all Out-of-Scope log files. We observe that the cascade model with threshold t=0.95 has an average rank of 2.77, which is the least compared to the other values. On the other hand, t=0.75 has the least standard deviation, but the average rank is relatively higher when compared to t=0.95 and therefore, we select the t=0.95 threshold as the best-performing value.

Similarly, we consider the CB Cascade and perform the same set of experiments. We find out that threshold $t = 0.9$ performs the best. It achieves an average rank of 3.11 with a standard deviation of 2.315.

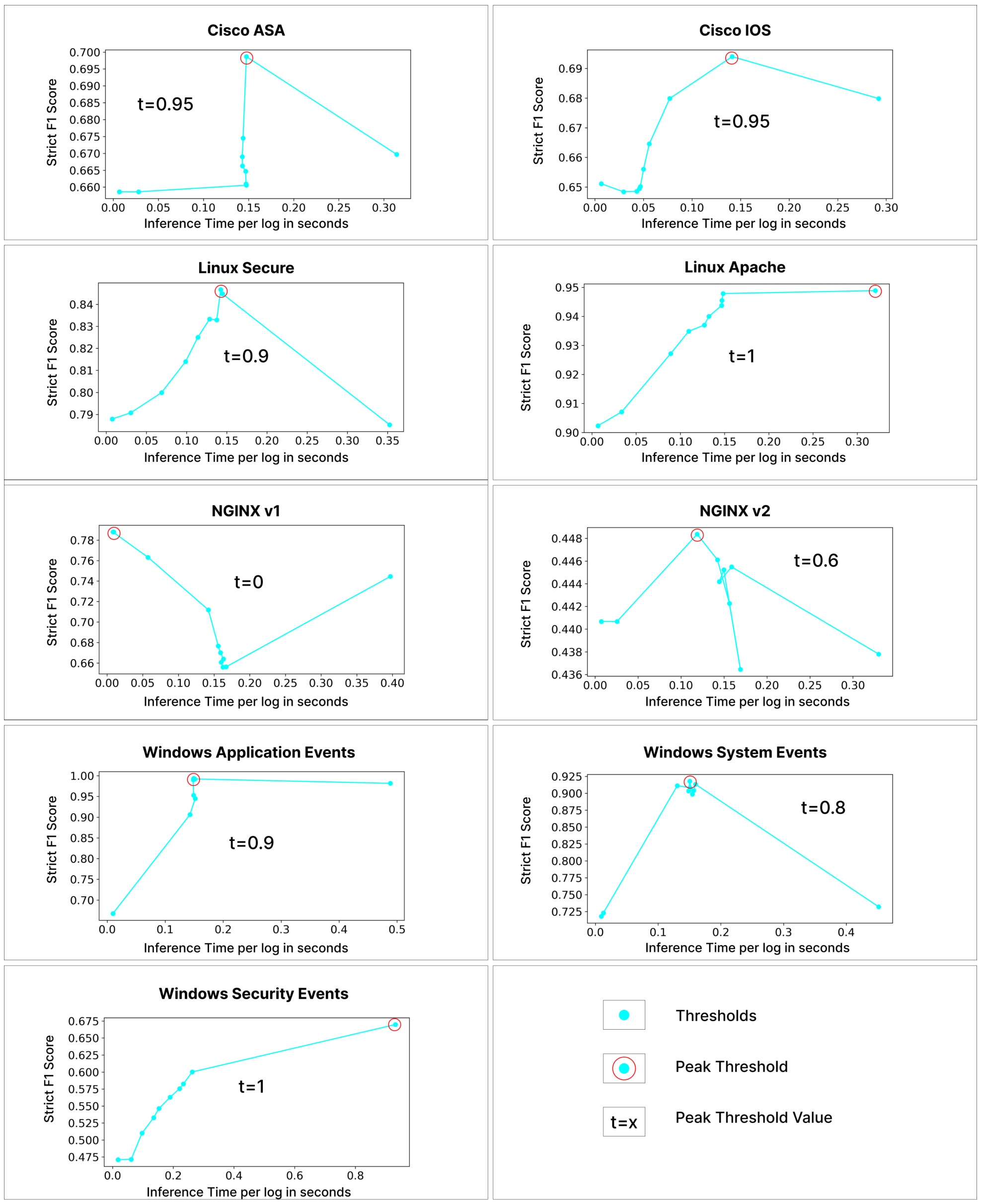


Figure 5.8: Out-of-Scope Data - CBT Cascade Performance Result

Table 5.9: Out-of-Scope Data: CBT Cascade Threshold Analysis

| | Cisco ASA | Cisco IOS | Linux Secure | Linux Apache | NGINX v1 | NGINX v2 | WAE | WSYE | WSE | Avg Rank | Std Dev |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0.00** | 0.65862 | 0.65112 | 0.78793 | 0.90232 | **0.78793** | 0.44068 | 0.66747 | 0.71813 | 0.47099 | 8.555556 | 2.973260 |
| **0.25** | 0.65862 | 0.65112 | 0.78793 | 0.90232 | 0.78793 | 0.44068 | 0.66747 | 0.71813 | 0.47099 | 8.555556 | 2.973260 |
| **0.50** | 0.65862 | 0.64842 | 0.79075 | 0.90710 | 0.76318 | 0.44068 | 0.66747 | 0.72324 | 0.47155 | 8.555556 | 2.297341 |
| **0.60** | 0.66055 | 0.64857 | 0.79991 | 0.92717 | 0.71189 | **0.44836** | 0.90596 | 0.91116 | 0.51016 | 6.444444 | 2.877113 |
| **0.70** | 0.66094 | 0.64948 | 0.81396 | 0.93489 | 0.67669 | 0.44612 | 0.94501 | 0.90824 | 0.53267 | 6.111111 | 2.027588 |
| **0.75** | 0.66471 | 0.65015 | 0.82503 | 0.93700 | 0.66998 | 0.44226 | 0.95277 | 0.90334 | 0.54620 | 6.222222 | 0.833333 |
| **0.80** | 0.66627 | 0.65603 | 0.83330 | 0.94002 | 0.66398 | 0.43645 | 0.99004 | **0.91783** | 0.56290 | 5.222222 | 2.862594 |
| **0.85** | 0.66902 | 0.66454 | 0.83291 | 0.94372 | 0.66069 | 0.44523 | 0.99110 | 0.89847 | 0.57559 | 4.777778 | 1.922094 |
| **0.90** | 0.67449 | 0.67986 | **0.84668** | 0.94550 | 0.65598 | 0.44418 | **0.99358** | 0.90416 | 0.58241 | 3.666667 | 3.122499 |
| **0.95** | **0.69866** | **0.69399** | 0.84476 | 0.94787 | 0.65623 | 0.44549 | 0.99216 | 0.91324 | 0.60006 | **2.777778** | 2.773886 |
| **1.00** | 0.66967 | 0.67985 | 0.78531 | **0.94888** | 0.74450 | 0.43780 | 0.98166 | 0.73220 | **0.66987** | 5.111111 | 3.723051 |

## 5.5 Impact of LoDU and Delimiter Classification Unit

In this section, we discuss the results of the system without LoDU and the Delimiter Classification Unit as used in [10]. Without the two mentioned units, there is no enrichment and augmentation within the system, and the system relies on learning from the provided logs. Table 5.10 shows the strict precision, recall and F1 score for CRF, Transformer and BERT achieved across all the Out-of-Scope log files. In this experimentation, the NGINX v1 and v2 files were combined, and the results were reported. The CRF model is unable to learn the underlying concepts of entities and got a strict F1 score of 0 for 6 out of the 8 Out-of-Scope log files. All the NER approaches failed to extract the timestamps from the windows log files and get a strict F1 score of 0. A similar performance behaviour is observed for the NGINX log files, which could not be parsed either, and all the NER approaches

Table 5.10: Strict Evaluation for Out-of-Scope Data without LoDU and Delimiter Classification Unit.

| **Log files** | **Metric** | **CRF** | **Transformer** | **BERT** |
|---|---|---|---|---|
| **Cisco ASA** | Precision | 0 | 0.6666 | 0.997 |
| | Recall | 0 | 0.9923 | 0.997 |
| | F1-score | 0 | 0.7975 | 0.997 |
| **Cisco IOS** | Precision | 0 | 0.5688 | 0.00053 |
| | Recall | 0 | 0.8629 | 0.00066 |
| | F1-score | 0 | 0.6856 | 0.00059 |
| **Linux Secure** | Precision | 0.9696 | 0.9698 | 0.7028 |
| | Recall | 0.5348 | 0.4407 | 0.6193 |
| | F1-score | 0.6894 | 0.6060 | 0.6584 |
| **Linux Apache** | Precision | 0.9598 | 0.9993 | 0.9821 |
| | Recall | 0.6841 | 0.9907 | 0.9711 |
| | F1-score | 0.7988 | 0.9950 | 0.9766 |
| **NGINX (v1+v2)** | Precision | 0 | 0.0351 | 0.0121 |
| | Recall | 0 | 0.0060 | 0.0067 |
| | F1-score | 0 | 0.0102 | 0.0086 |
| **WAE, WSYE, WSE** | Precision | 0 | 0 | 0 |
| | Recall | 0 | 0 | 0 |
| | F1-score | 0 | 0 | 0 |

achieve a strict F1 score of close to 0. The Linux Apache log file is parsed the best out of all the log files with a strict F1 score above 0.8. BERT achieves a near-to-perfect result in parsing the Cisco ASA logs, whereas it suffers in extracting information from the Cisco IOS file. Comparing the results obtained here with those after treating the logs with LoDU and Delimiter Classification Unit, we observe a substantial increase in the strict F1 scores across most Out-of-Scope log files. We can infer that training the NER approaches on the augmented and enriched log dataset give the algorithms a comprehensive view of the value that the entities can possess.

## 5.6 Statistical Tests

We perform a set of non-parametric statistical tests on the resultant data to check our results' statistical significance. We compare every possible pair of NER approaches using the Wilcoxon Signed Rank Test. We then perform Friedman's t-test, which is followed by Nemenyi's post hoc test. Further subsections discuss each statistical test in detail.

### 5.6.1 Wilcoxon Signed Rank Test

The Null hypothesis for the Wilcoxon Signed Rank Test for our use case is as follows: Given two NER approaches, A and B, both A and B perform equally. There are a total of ${}^{14}c_2$ pair combinations possible with 14 NER approaches. We perform the test on each pair and calculate its corresponding p-value. Figure 5.9 shows a heat map depicting the p-values of each possible pair of NER approaches. The heatmap is symmetric; that is, the lower left and the upper right triangles of the heatmap represent the same information. Figures 5.10 and 5.11 show the binary representation of whether there is a statistical difference between the two approaches at 95 percent and 99 percent confidence, respectively. The black cells of the heatmap depict no statistical difference between the two NER approaches. At the same time, the light-coloured cells represent the pair of NER approaches that are statistically different.

At 95% confidence, 47 pairs of algorithms possess statistical differences, whereas the number drops to 38 at 99% confidence. Out of these combinations, at least 30 pairs included at least one traditional method at both confidence levels. For the 95% confidence level,

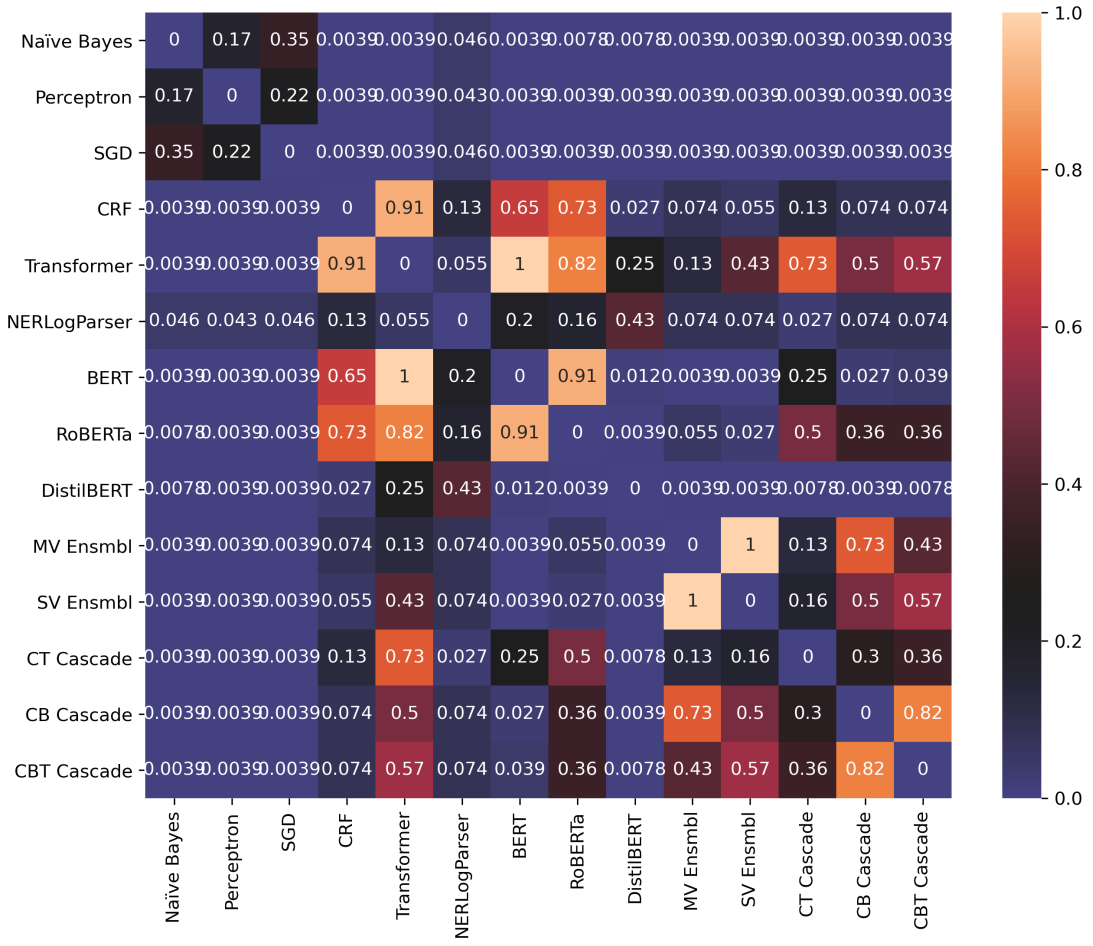


Figure 5.9: Heat Map depicting P values for Wilcoxon's Signed Rank Test.

the following pairs had statistically significant differences: DistilBERT-CRF, DistilBERT-BERT, DistilBERT-RoBERTa, DistilBERT-Ensembles, DistilBERT-Cascades, BERT Ensembles, BERT-CB Cascade, BERT-CBT Cascade, RoBERTa-SV Ensmbl and NERLogParser-CT Cascade. At the same time, for 99% confidence, the following pairs had a statistically significant performance difference: BERT-Ensembles, DistilBERT-RoBERTa, DistilBERT-Ensembles and DistilBERT-Cascades.

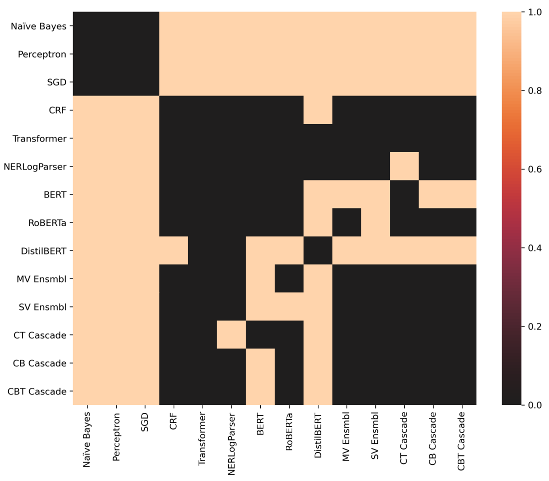


Figure 5.10: Wilcoxon Test - 95% conf.

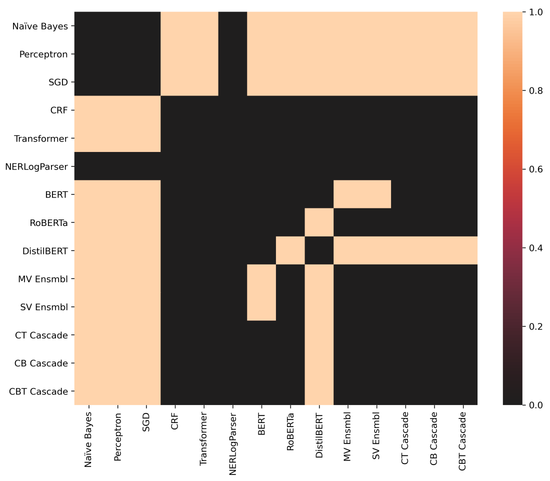


Figure 5.11: Wilcoxon Test - 99% conf.

### 5.6.2 Friedman's t Test and Nemenyi's Post Hoc Test

The null hypothesis for the Friedman's t-test applied to our use case is as follows: Given a group of NER approaches, all perform equally. Unlike the Wilcoxon Signed Rank test, Friedman's t-test is not used for pairs but computes a p-value for the entire set of NER approaches. The calculated p-value has a value of 2.4827e-10, which is less than 0.05 and 0.01, so we can say that all the NER methodologies do not perform equally at both the confidence levels. Since the p-value of the test is statistically significant, we can proceed with the Nemenyi's post hoc to determine which methodologies differ. Figure 5.12 shows the heatmap depicting the p values for the Nemenyi's Friedman test. Similar to the Wilcoxon Signed Rank Test, we can compare the p values to 0.01 or 0.05 to check the statistical significance at the respective confidence levels.

Figure 5.13 shows a rank graph depicting the average ranks of different NER methodologies over a number line. CD refers to the critical difference and is a measure used to

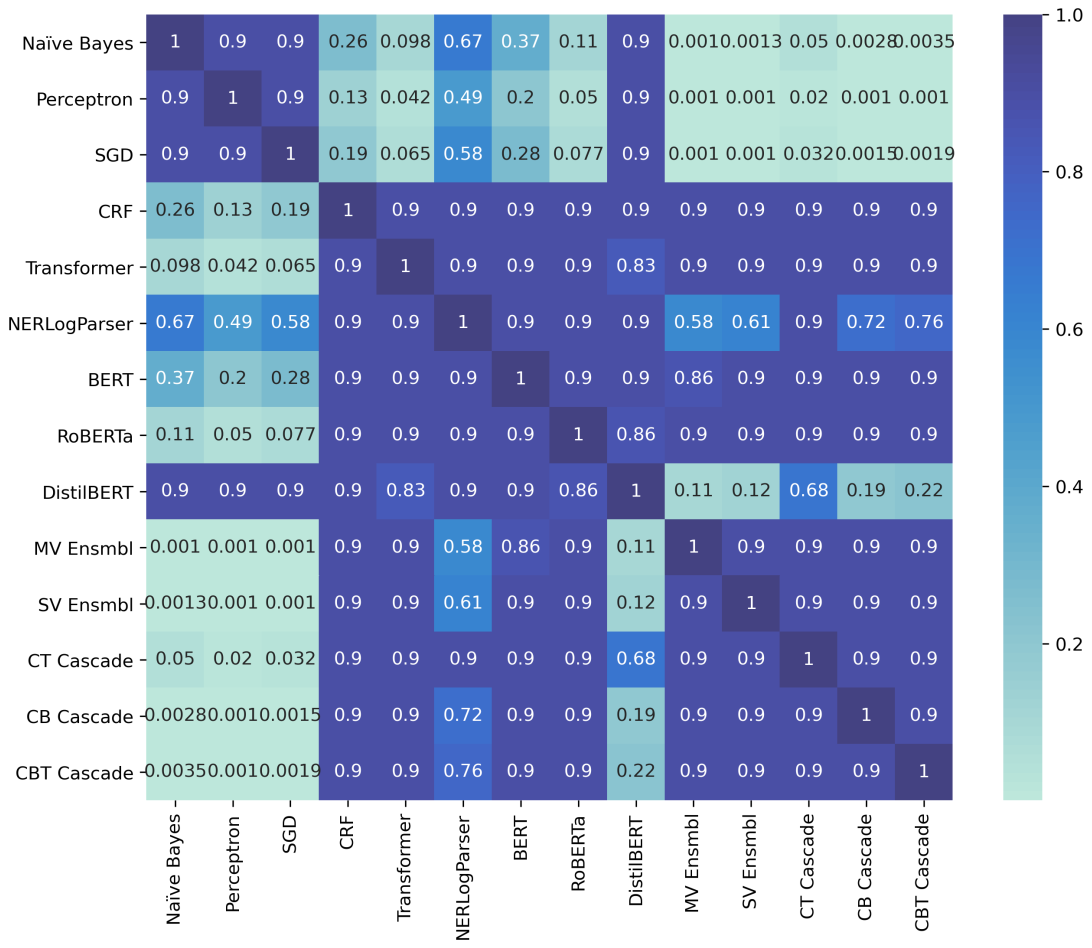


Figure 5.12: Heat Map depicting P values for Nemenyi's Friedman Test.

identify whether two approaches are statistically significant or not. The value of the critical difference at 95% confidence is calculated to be 6.6134. The bold black horizontal lines are used for comparing two or more approaches and are of the same length as critical difference. Suppose the difference between the average ranks of any two approaches is greater than the critical difference. In that case, there is a statistically significant difference between the performance of the two methodologies. After referring to the heatmap and the rank graph, we can observe that there is a statistically significant difference between the following pairs: Traditional methods - Ensembles, Traditional methods - cascades, Transformer-Perceptron and RoBERTa-Perceptron.

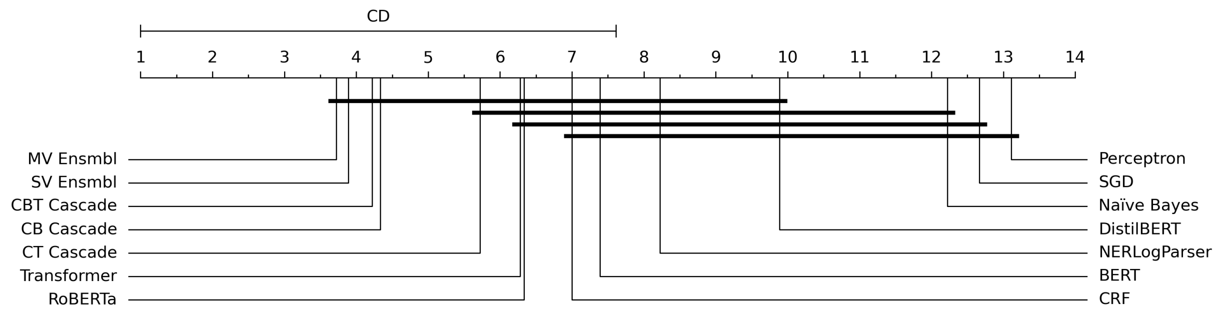


Figure 5.13: Nemenyi's Post Hoc Test depicting Ranking and Critical Difference

## 5.7 Synthesis and Discussion

**In-Scope Data Results:** The traditional word-based baseline methods, including Naive Bayes and SGD, perform decently high with a strict F1 score of over 60%. Both the approaches surpass the random guessing baseline. The underlying principle of these approaches is to learn just by using the tokens without knowing the surrounding context. Getting such a high score without knowing the context points towards an overlap of the vocabulary of training and testing datasets. We investigated and found out that over 63% of tokens present in the test logs are also a part of the training vocabulary. This percentage of overlap justifies the strict F1 score achieved by the baseline learners. Despite having a heavy overlap in the vocabulary, perceptron suffers in the learning process and reaches a strict F1 score of 0.35.

In comparison, all the other approaches which take context into account perform well, with a strict F1 score of over 0.998. A strict F1 score close to 1 depicts the ability of the parser to parse the entire log file with just a few entity mismatches. There is no significant difference between the performance of all these models. To dig down into such high F1 scores, we analyzed the number of unique entity sequences present in the training and testing datasets. The test data contains 24,399 log instances with a total of 226 unique entity sequences. The LoDU treated augmented training dataset includes 15,235 different entity sequences. The intersection between the two sets is really high, with 223 entity sequences. Over 98% of the entity sequences present in the testing set are present in the training dataset. This commonality in the structure of logs across the training and testing dataset justifies achieving high strict F1 scores.

**Out-of-Scope Results:** Most NER approaches worked well for parsing the Linux Apache log file with a strict F1 score $> 0.9$. Interestingly, we achieve such high performance even on a log file that originates from an entirely different distribution. We analyzed the structure of the Linux Apache log files and found out that it is very similar to the structure of the Web Logs present in the training dataset. In contrast, we observe the opposite behaviour with the NGINX v2 log file. All the NER methodologies could achieve a maximum strict F1 score of $\sim 0.45$, even worse than the random guessing baseline. The NGINX v1 log file is better parsed than the NGINX v2 file and gets a strict F1 score $> 0.6$ for most of the methodologies having the best performing model present at a score of 0.83. We compared the two versions of NGINX files and found that both are structurally similar except for the presence of two hyphens (-) within the logs of the NGINX v2 file. The presence of hyphens could be the reason for a dip in NER models' performance for the second version of the log file. The traditional word-based methods could not parse any windows-based log files and got a strict F1 score of 0. NERLogParser achieves a strict F1 score of 0 for all the windows-based log files. NERLogParser is not trained on the augmented data and is used as a pre-trained model and hence, it does not perform well on some of the Out-of-Scope log files. The Transformer model achieved a strict F1 score of  0.5 for all the windows files. We investigated the predictions and found that the Transformer model introduces a lot of false positives by predicting 'O' as 'I-HOS', and 'I-SER'. Ensembles work the best in parsing the windows log files, reaching a strict F1 score $> 0.9$ for WAE and WSYE whereas $>0.65$ for the Security Events (WSE) log files. Even the cascade models, except the CT Cascade, achieved similar performance to the ensembles for parsing the windows-based log files. It is clear from the results that the Transformer model does not play an essential role in extracting information from the windows based log files as it introduces a lot of false positives. Another interesting observation is that all the NER approaches except the traditional methods and NERLogParser score a recall of 1 for all the windows-based log files. This vast difference between the precision and recall is because the count of $|ACT|$ is greater than the count of $|POS|$ (Equations 4.2, 4.3). For the Linux Secure log file, CRF achieves a strict F1 score of  0.79, whereas the language models could not perform well, having a score $< 0.65$. We can thus infer that the feature functions of the CRF better capture the dependencies than the attention-based language models in parsing log files structurally equivalent to Linux Secure. On the opposite end, we see an inverted pattern for the Cisco IOS file where the attention-based language models outperform the feature function-based CRF model.

The selection of the best-performing model is challenging. Choosing the model with the best strict F1 score would not be wise. There are a lot of factors that we need to take into account, including 1) Performance on the In-Scope data, 2) Average rank achieved on the Out-of-Scope log files, 3) The stability/standard deviation of the model, and 4) Throughput of the NER approach

The use-case of the log parser is also an essential factor which could help us decide which model to choose. If we are required to parse the logs in real-time, ensembles are probably not an option to choose. On the other hand, if we need the best results, ensembles could give us the best parsing capabilities. The CRF model can be the best choice in use cases involving real-time streaming and processing. The throughput of the CRF model is $\sim 50$ times faster than the SV Ensmbl. To balance the trade-off between time and performance, cascades can be employed. Cascading classifiers could be a viable option, as the throughput is approximately 2 to 8 times faster than the ensembles without a significant compromise in strict F1 score. From one perspective, the MV Ensmbl achieves the least average rank of 3.722 in parsing the Out-of-Scope log files, and it should be considered the best performing model. Whereas, if we look at the standard deviation, the SV Ensmbl is more stable in performance with a standard deviation of 1.099 and a comparable average rank of 3.889.

**Cascade Results:** We experiment with a total of three different cascading classifiers. We selected models having completely different underlying principles to tackle the same problem statement. We decided to form the cascades out of different combinations of CRF, BERT and Transformer. CRF is chosen to be the first model in all three cascades because of the high performance and less inference time. The motivation and end goal of developing cascades is to achieve similar performance to ensembles and possess a better throughput. Different levels of thresholds are used to perform experimentations to understand the effect of the level of participation of various models in the prediction process.

**In-Scope Data Cascade Results:** For the In-Scope data, we start with a threshold of 0 (individual CRF model), keep increasing the threshold to 1 (soft voting classifier including all three models), and reach a strict F1 score of  0.99992 for all the three cascades. At threshold $t = 0.95$, we achieve almost the same performance as the SV Ensmbl but a ten times higher throughput. This happens because the cascading classifiers include knowledge from all the models on a need and conditional basis. The high throughput and performance equivalent to SV Ensmbl make the cascades better suited for real-time applications.

At $t = 0.95$ threshold, we fall back to the BERT/Transformer model only if a prediction's confidence score is less than 0.95. With just a stand-alone CRF model (t=0), we achieve a high strict F1 score of  0.99985. We observe that $t = 0.95$ helps boost the strict F1 score to  0.99992. BERT and Transformer models could be confident in predicting entities for cases where CRF is not confident enough (confidence score $< 0.95$). The high throughput of cascades at t=0.95 can be justified by the high confidence score obtained by CRF for most of the instances. The high confidence score of CRF doesn't allow it to fall back to the other participating models in the cascade and hence the throughput of the cascade is ten times faster than the SV Ensmbl.

**Out-of-Scope Data Cascade Result:** On the other hand, all three cascades perform entirely differently for every log file present in the Out-of-Scope dataset. Further below, we discuss the performance of various cascades in detail. The CT Cascade best parsed 6 out of 9 files at a threshold value other than 0 (stand-alone CRF model) or 1 (SV Ensmbl). We observed no standard threshold value that performs the best across all the log files, and hence we move ahead with performing a ranking on various log files across different threshold values. For Cisco ASA, the stand-alone CRF model achieves a strict F1 score of 0.65, and the Transformer model scores 0.95 (Table 5.3). Referring to Table 5.8, the cascade performs best in parsing Cisco ASA at $t = 0.85$, achieving a strict F1 score of 0.86, lying well between the performance metrics of the two participating models. The infused knowledge of Transformer in cases where CRF is not confident helps boost the score of the entire system and increases it by 0.2. Similar behaviour can be observed in Linux Apache, WAE, and WSE log files, where falling back to the Transformer model helps boost the performance.

In contrast, the infusion of knowledge from the Transformer Neural Network does not help in improving the parsing quality of Cisco IOS, Linux Secure, NGINX v1 and NGINX v2 log files. This could be justified as the CRF model could be very confident in predicting even the wrong entities, which prevents it from transferring control to the Transformer model. Overall, there is a decreasing trend in the average rank as we increase the threshold. The average rank decreases from 7.5 to 4.4 as we increase the threshold from 0 to 1. The minimum ranking threshold is $t = 0.95$ with an average rank of 4.

The CBT Cascade best parses 6 out of 9 log files at a threshold other than 0 or 1. For the Cisco ASA log file, the strict F1 score increases from 0.65 to 0.69 (Table 5.9) when the threshold increases from 0 to 0.95. The individual transformer model achieves a strict F1 score of 0.94, but the cascade suffers from using the model's knowledge. The infusion of knowledge doesn't occur as the Transformer is placed last on the cascading ladder. The total number of control diversions to the transformer model could be less than that of the BERT model. Even a threshold value of 1 results in a strict F1 score of 0.66

because the CRF and BERT could be confident about predicting wring entities. Hence, it overpowers the knowledge carried by the Transformer. The same pattern is observed for the Cisco IOS log file. For the Linux Secure log file, CRF achieved a strict F1 score of  0.78, whereas BERT and Transformer got  0.64 and  0.59, respectively. The cascade of these models achieves a strict F1 score of  0.84 at a threshold of 0.8. This increase in performance could be because BERT and Transformer better predict entities where the CRF model suffers. The individual participant models of the cascade did not perform well in parsing the windows log files, with Transformer getting a strict F1 score of  0.5 across all the three files. Even with a low score achieved by Transformer, the cascade model managed to achieve high strict F1 scores of over 0.9 for WAE and WSYE log files. This high performance by the cascade can be justified by the meagre participation of the Transformer in the prediction-making process. The confidence scores of CRF and BERT could be high enough to prevent the control from being transferred to the Transformer model.

## 5.8 Summary

The experimentation started by using Bayesian Optimization to conduct the hyperparameter search for all NER approaches. We performed 20 trials to best estimate the underlying objective function for each NER approach. We found that various hyperparameters had different importances in contributing to the best-performing classifier. For instance, 1) max iterations for CRF, 2) learning rate, number of encoders-decoders for Transformer, and 3) learning rate for BERT contributed the most in finding the best performing variant of the model. The best hyperparameters for each model were observed and reported.

Next, we performed a cascade threshold analysis where we experimented with three different cascades and eleven different levels of thresholds. A total of 33 different models are used to parse all the log data. The threshold value of 1 performed the best for parsing the In-Scope log data, but we decided to choose $t = 0.95$ because of the high throughput and equivalent performance. The same analysis is performed on the Out-of-Scope log files. Since different threshold values worked best for different Out-of-Scope log files, we performed a ranking on the strict F1 scores and selected the model with the least average rank for each of the cascades. We observed that $t = 0.95$, $t = 0.9$ and $t = 0.95$ worked the best for CT Cascade, CB Cascade and CBT Cascade respectively.

Once the best-performing cascades were selected, an extensive comparison between 14 different NER approaches was performed. We started by considering the In-Scope log data. The precision, recall and F1 score was reported for four different evaluation schemes. For the In-Scope data, even the traditional word-based methods (Naive Bayes and SGD) achieved a strict F1 score $> 0.6$. The high performance by the baselines was investigated, and it was found that there was a heavy overlap between the tokens present in the training and testing log data. The other NER approaches performed nearly perfectly, with a strict F1 score of over 0.99. The high structural similarity of logs present in the training and testing dataset was responsible for such high-performance results. The CB Cascade and CBT Cascade were the two best performing models for the In-Scope data having a strict F1 score of 0.99992.

The same metrics were collected for each log file present in the Out-of-Scope log dataset.

Most NER approaches best parsed the Linux Apache log file with a strict F1 score $> 0.9$. The NER approaches suffered in parsing the NGINX v2 log file having a strict F1 score $< 0.45$. It was found that different NER approaches worked best for different Out-of-Scope log files. We performed a ranking across all the log files to choose the best-performing model. The MV Ensmbl achieved the least average rank of 3.722, followed by the SV Ensmbl with an average rank of 3.889. The ranking process was also carried out on the inference times of the NER approaches. CRF was the best-performing model in terms of inference time, whereas the Ensembles were the worst. Different use cases were considered, and various attributes of the NER approaches were compared to select a best-performing model relevant to each use case. Traditional ensembles could be the go-to models if the log parser needs to be used in applications that analyze the log's patterns and content, and no immediate action needs to be taken. In systems that require real-time streaming and processing of logs, it would be unrealistic to deploy ensemble-based solutions. Despite the high average rank, the stand-alone CRF model could better fit the real-time systems. Overall, we saw that there is a trade-off between the performance and inference times in our experimentations and hence, Cascades can be a go-to option as they try to balance both.

# Chapter 6

# Conclusions

In this thesis, we focused on developing a robust system to carry out information extraction from log files. The information extraction problem was framed as a NER problem. The primary goal of this research work was to evaluate the robustness of information extraction systems when prompted with previously unseen data. We developed augmentation and enrichment techniques that helped improve the generalization capabilities of the NER systems. A total of 14 NER approaches were implemented and evaluated, including word-based baselines like Naive Bayes, graphical model: CRF, attention-based sequence-to-sequence: Transformer and NERLogParser, language models like BERT, ensembles and cascades of individual models. A rigorous comparison between these NER approaches was conducted using a multiple-perspective evaluation framework. This chapter further discusses the contributions of our research work and areas where future work can be carried out.

## 6.1 Contributions

In this thesis we provided the following main contributions:

- **Extensive experimental comparison of ML-based NER methodologies for log parsing:** We experiment with 14 different NER approaches ranging from simple baseline word-based approaches to complex deep learning based language models. Out of all the NER methodologies, four are purely attention-based, including Transformer, BERT, RoBERTa and DistilBERT. We also experiment with Majority Voting and Soft Voting Ensembles to utilize knowledge from multiple individual classifiers. To balance the trade-off between performance and inference time, we also include cascading classifiers, a class of ensembles that operate on conditional threshold-based policies. We employ an evaluation framework that offers multiple evaluation schemes, helping us compare and assess the quality of information extraction.

- **Evaluate robustness and generalization capabilities of information extraction systems for log files:** No formal studies evaluate the robustness or generalization capabilities of information extraction systems. The primary contribution of our research work is the evaluation of the NER approaches using previously unseen data. The assessment of NER systems using new or previously unseen data gives us an idea about the generalization capabilities and robustness of the systems. We considered two different datasets to carry out the evaluation: The In-Scope data and the Out-of-Scope data. The In-Scope data is used to train-validate-test the NER approaches, while the latter contains logs originating from entirely different sources and is solely used for evaluation purposes.

- **Text augmentation techniques specific to log files:** Another gap in the research which has not gained much attention is data augmentation for log files. There have been applied studies which have developed and experimented with various data augmentation techniques for textual data [56, 67]. These data augmentation techniques cannot be extended to log files as these are specific to English language classification tasks. In contrast, the nature and structure of log files differ a lot from the English language. We propose the LoDU, which takes care of the enrichment and diversification of log instances. This unit is responsible for entity enrichment, structural enrichment, and other pre-processing tasks specific to log files.

Overall, this thesis fills in many research gaps and previously undocumented areas, focusing on improving the generalization capabilities of NER systems for log files.

## 6.2 Areas of Improvement and Future Work

The first and foremost improvement for further research lies in the number of logs being considered. The experimental work presented in this thesis considers a dataset with 120K log instances. Modern-day distributed systems can emit such amounts of log lines in a few moments. Thus, we can say that this setting does not reflect real-life scenarios closely. Considering a dataset with various log sources collected over a considerable time period can help solve this problem. Another aspect which we can consider in this work is the structural differences between logs. Rather than collecting many logs from sources with structural similarities, we can collect smaller chunks from multiple sources having structural differences. This can help us introduce diversity inherently within the dataset. The experimental setting used in this thesis divides the In-Scope dataset into training-validation-test set using a random 60-20-20 split. This can be extended by exploring the correlation between various logs based on their origin, content and features like length to have a better representative split.

The current In-Scope dataset consists of logs containing 23 different entities. On the other hand, the Out-of-Scope dataset contains 37 unique entities. Out of these sets of entities, 11 belong to the intersection. We evaluate the NER approaches based on these intersecting entities. We can work on collecting a dataset with a comparatively larger set of entities in the future. Increasing the size of intersecting entities is also an area of interest that can be explored in the future work. The main idea is to create datasets that closely reflect the modern-day distributed systems.

We train our NER approaches on the In-Scope dataset and use the Out-of-Scope dataset to evaluate the robustness. The key distinction between both datasets is that they originate from different sources. In the future work, we can better define the notion of distinction for logs which could include features relating to log origin and log content. We can also formulate a similarity score for log instances and can use datasets with different similarity scores to evaluate the robustness and generalization capability of NER approaches.

Carrying out log augmentation much more extensively may benefit the system as well. We performed enrichment on just two entities to portray the idea and results in this work. This can be scaled to several entities so that the NER approaches get a decent understanding of the concept of entities. The log augmentation can be improved by introducing gibberish or symbols in between log instances; this way, the NER approaches could distinguish between the gibberish and the entities. All the NER approaches could not parse the NGINX v2 log file with a strict F1 score $< 0.45$ because of the presence of a hyphen ("-") in the log instances. The training data was missing log samples which contained examples with such symbol values. Hence, log augmentation techniques which include random symbols or gibberish can be helpful in such situations. The augmentation techniques that we discussed above are at a human readable entity level, there are also augmentation techniques that rely on higher level representations including sentence-level embeddings. Mixup [59] is one of such techniques and it does not require any domain knowledge for carrying out augmentation. Mixup has also been used in the context of performing NER for English and Chinese language [69].

Currently in our work, we have considered CRF to be an individual model in ensembles. We can also use the CRF as a top head present over other complex networks to enforce the sequence of the entities present in the logs. Misawa et al. [44] and Souza et al. [57] use CRF over BiLSTM and BERT to carry out Japanese and Portuguese NER respectively. Exploring the use of different threshold values within the same cascading classifier could be valuable in studying as well. Currently, the cascade classifiers use a single threshold value throughout the classifier. A single threshold limits the possibility of searching through all the possible combinations of contributions of the participating classifiers. The use of cascades proved to help decrease the inference time while maintaining the performance level in an offline setting. We can consider evaluating all the NER approaches, including cascades in an environment with streaming data that closely reflects real-time distributed systems.

The NER approaches employed in this work are evaluated and compared on the basis of strict F1 score, stability and inference time. Another interesting perspective could be to compare the models on the basis of their calibration. In this work, we framed the log parsing problem as a Named Entity Recognition (NER) task. We can also frame this problem as a question-answering task. Language models like SpanBERT [31] are better trained to tackle tasks involving predicting spans of text and can be used to extract information from logs.

With the help of Delimiter Classification Unit, we are able to handle logs containing delimiters other than white-space. In the future, we can completely remove the need of Delimiter Classification Unit and make the system inherently delimiter independent. Language models like ALBERT [37] employ a language independent tokenizer: SentencePiece [32]. Making the use of ALBERT and SentencePiece in the context of logs can make the system inherently delimiter independent. Finally, with the recent advancements in modelling language models and tokenizers, we can consider working on pre-training a language model using logs. Such a language model pre-trained on vast amounts of logs can be used for purposes like anomaly detection, temporal ordering of logs, information extraction, etc.

# References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[2] Ethem Alpaydin and Cenk Kaynak. Cascading classifiers. *Kybernetika*, 34(4):369–374, 1998.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] Yassine Benajiba and Paolo Rosso. Arabic named entity recognition using conditional random fields. In *Proc. of Workshop on HLT & NLP within the Arabic World, LREC*, volume 8, pages 143–153. Citeseer, 2008.

[5] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceed-*

*ings of the 36th International Conference on Software Engineering*, pages 468–479, 2014.

[6] Léon Bottou et al. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8):12, 1991.

[7] Xavier Carreras, Lluis Marquez, and Lluis Padro. Learning a perceptron-based named entity chunker via online recognition feedback. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003*, pages 156–159, 2003.

[8] Kevin Chen, Andrew Clark, Olivier De Vel, and George Mohay. Ecf-event correlation for forensics. In *First Australian Computer, Network and Information Forensics Conference*, pages 1–10. We-B Centre. com, 2003.

[9] Wenliang Chen, Yujie Zhang, and Hitoshi Isahara. Chinese named entity recognition with conditional random fields. In *Proceedings of the Fifth SIGHAN Workshop on Chinese Language Processing*, pages 118–121, 2006.

[10] Anubhav Chhabra, Paula Branco, Guy-Vincent Jourdan, and Herna L Viktor. An extensive comparison of systems for entity extraction from log files. In *International Symposium on Foundations and Practice of Security*, pages 376–392. Springer, 2022.

[11] Anubhav Chhabra, Tirumala Sree Akhil Nandyala, and Paula Branco. Heal: Heterogeneous ensemble and active learning framework.

[12] Nancy Chinchor and Beth M Sundheim. Muc-5 evaluation metrics. In *Fifth Message Understanding Conference (MUC-5): Proceedings of a Conference Held in Baltimore, Maryland, August 25-27, 1993*, 1993.

[13] Jason PC Chiu and Eric Nichols. Named entity recognition with bidirectional lstm-cnns. *Transactions of the association for computational linguistics*, 4:357–370, 2016.

[14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[15] Darya Chyzhyk, Manuel Grana, Alexandre Savio, and Josu Maiora. Hybrid dendritic computing with kernel-lica applied to alzheimer's disease detection in mri. *Neurocomputing*, 75(1):72–77, 2012.

[16] Biplob Debnath, Mohiuddin Solaimani, Muhammad Ali Gulzar Gulzar, Nipun Arora, Cristian Lumezanu, Jianwu Xu, Bo Zong, Hui Zhang, Guofei Jiang, and Latifur Khan. Loglens: A real-time log analysis system. In *2018 IEEE 38th international conference on distributed computing systems (ICDCS)*, pages 1052–1062. IEEE, 2018.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM computing surveys (CSUR)*, 27(3):326–327, 1995.

[19] Huong TX Doan and Giles M Foody. Increasing soft classification accuracy through the use of an ensemble of classifiers. *International Journal of Remote Sensing*, 28(20):4609–4623, 2007.

[20] Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864. IEEE, 2016.

[21] Oren Etzioni, Michele Banko, Stephen Soderland, and Daniel S Weld. Open information extraction from the web. *Communications of the ACM*, 51(12):68–74, 2008.

[22] Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

[23] Rainer Gerhards et al. The syslog protocol. Technical report, RFC 5424, March, 2009.

[24] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm networks. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 4, pages 2047–2052. IEEE, 2005.

[25] Haixuan Guo, Shuhan Yuan, and Xintao Wu. Logbert: Log anomaly detection via bert. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[26] Hongcheng Guo, Xingyu Lin, Jian Yang, Yi Zhuang, Jiaqi Bai, Bo Zhang, Tieqiao Zheng, and Zhoujun Li. Translog: A unified transformer-based framework for log anomaly detection. *arXiv preprint arXiv:2201.00016*, 2021.

[27] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*, pages 33–40. IEEE, 2017.

[28] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network (2015). *arXiv preprint arXiv:1503.02531*, 2, 2015.

[29] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[30] Kuo-Wei Hsu. A theoretical analysis of why hybrid ensembles work. *Computational intelligence and neuroscience*, 2017, 2017.

[31] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics*, 8:64–77, 2020.

[32] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

[33] P Arun Raj Kumar and S Selvakumar. Distributed denial of service attack detection using an ensemble of neural classifier. *Computer Communications*, 34(11):1328–1341, 2011.

[34] P Arun Raj Kumar and S Selvakumar. Detection of distributed denial of service attacks using an ensemble of adaptive and hybrid neuro-fuzzy systems. *Computer Communications*, 36(3):303–319, 2013.

[35] Ludmila I Kuncheva, Marina Skurichina, and Robert PW Duin. An experimental study on diversity for bagging and boosting with linear classifiers. *Information fusion*, 3(4):245–258, 2002.

[36] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*, 2016.

[37] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[38] Yukyung Lee, Jina Kim, and Pilsung Kang. Lanobert: System log anomaly detection based on bert masked language model. *arXiv preprint arXiv:2111.09564*, 2021.

[39] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

[40] Dingcheng Li, Guergana Savova, and Karin Kipper. Conditional random fields and support vector machines for disorder named entity recognition in clinical texts. In *Proceedings of the workshop on current trends in biomedical natural language processing*, pages 94–95, 2008.

[41] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[42] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[43] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD*

*international conference on Knowledge discovery and data mining*, pages 1255–1264, 2009.

[44] Shotaro Misawa, Motoki Taniguchi, Yasuhide Miura, and Tomoko Ohkuma. Character-based bidirectional lstm-crf with words and characters for japanese named entity recognition. In *Proceedings of the first workshop on subword and character level models in NLP*, pages 97–102, 2017.

[45] Loris Nanni. Ensemble of classifiers for protein fold recognition. *Neurocomputing*, 69(7-9):850–853, 2006.

[46] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[47] Irina Rish et al. An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

[48] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[49] Erik F Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050*, 2003.

[50] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[51] Bradley Schatz, George Mohay, and Andrew Clark. Rich event representation for computer forensics. In *Proceedings of the Fifth Asia-Pacific Industrial Engineering and Management Systems Conference (APIEMS 2004)*, volume 2, pages 1–16. Queensland University of Technology Publications, 2004.

[52] Isabel Segura-Bedmar, Paloma Martínez Fernández, and María Herrero Zazo. Semeval-2013 task 9: Extraction of drug-drug interactions from biomedical texts (ddiextraction 2013). Association for Computational Linguistics, 2013.

[53] Burr Settles. Biomedical named entity recognition using conditional random fields and rich feature sets. In *Proceedings of the international joint workshop on natural language processing in biomedicine and its applications (NLPBA/BioNLP)*, pages 107–110, 2004.

[54] Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *information security technical report*, 14(1):16–29, 2009.

[55] Keiichi Shima. Length matters: Clustering system log messages using length of words. *arXiv preprint arXiv:1611.03213*, 2016.

[56] Connor Shorten, Taghi M Khoshgoftaar, and Borko Furht. Text data augmentation for deep learning. *Journal of big Data*, 8(1):1–34, 2021.

[57] Fábio Souza, Rodrigo Nogueira, and Roberto Lotufo. Portuguese named entity recognition using bert-crf. *arXiv preprint arXiv:1909.10649*, 2019.

[58] Hudan Studiawan, Ferdous Sohel, and Christian Payne. Automatic log parser to support forensic analysis. 2018.

[59] Lichao Sun, Congying Xia, Wenpeng Yin, Tingting Liang, Philip S Yu, and Lifang He. Mixup-transformer: dynamic data augmentation for nlp tasks. *arXiv preprint arXiv:2010.02394*, 2020.

[60] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[61] Ah Chung Tsoi. Recurrent neural network architectures: an overview. *International School on Neural Networks, Initiated by IIASS and EMFCSC*, pages 1–26, 1997.

[62] Yoshimasa Tsuruoka, Jun'ichi Tsujii, and Sophia Ananiadou. Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 477–485, 2009.

[63] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*, pages 119–126. Ieee, 2003.

[64] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[65] Hanna M Wallach. Conditional random fields: An introduction. *Technical Reports (CIS)*, page 22, 2004.

[66] Xiaofang Wang, Dan Kondratyuk, Kris M Kitani, Yair Movshovitz-Attias, and Elad Eban. Multiple networks are more efficient than one: Fast and accurate models via ensembles and cascades. *arXiv preprint arXiv:2012.01988*, 2020.

[67] Jason Wei and Kai Zou. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196*, 2019.

[68] Qikang Wei, Tao Chen, Ruifeng Xu, Yulan He, and Lin Gui. Disease named entity recognition by combining conditional random fields and bidirectional recurrent neural networks. *Database*, 2016, 2016.

[69] Linzhi Wu, Pengjun Xie, Jie Zhou, Meishan Zhang, Chunping Ma, Guangwei Xu, and Min Zhang. Robust self-augmentation for named entity recognition with meta reweighting. *CoRR*, 2022.

[70] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael 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, 2009.

[71] Tao Yang, Vojislav Kecman, Longbing Cao, Chengqi Zhang, and Joshua Zhexue Huang. Margin-based ensemble classifier for protein fold recognition. *Expert Systems with Applications*, 38(10):12348–12355, 2011.

[72] Xue Ying. An overview of overfitting and its solutions. In *Journal of Physics: Conference Series*, volume 1168, page 022022. IOP Publishing, 2019.

[73] Chunkai Zhang, Xinyu Wang, Hongye Zhang, Hanyu Zhang, and Peiyi Han. Log sequence anomaly detection based on local information extraction and globally sparse transformer model. *IEEE Transactions on Network and Service Management*, 18(4):4119–4133, 2021.

# APPENDICES

Table 1: Partial Evaluation for Out-of-Scope Data.

| **Log files** | **Metric** | **Cisco ASA** | **Cisco IOS** | **Linux Secure** | **Linux Apache** | **NGINX v1** | **NGINX v2** | **WAE** | **WSYE** | **WSE** |
|---|---|---|---|---|---|---|---|---|---|---|
| | Precision | 0.42381 | 0.28395 | 0.24815 | 0.15804 | 0.70252 | 0.30851 | 0.12903 | 0.10812 | 0.04235 |
| **Naive Bayes** | Recall | 0.5 | 0.25813 | 0.13522 | 0.09874 | 0.21281 | 0.26432 | 0.25036 | 0.25 | 0.23317 |
| | F1-score | 0.45876 | 0.27042 | 0.17505 | 0.12155 | 0.32667 | 0.28471 | 0.17029 | 0.15095 | 0.07168 |
| | Precision | 0.0 | 0.08156 | 0.15643 | 0.16971 | 0.71032 | 0.26157 | 0.01626 | 0.02501 | 0.01462 |
| **Perceptron** | Recall | 0.0 | 0.10894 | 0.0907 | 0.08587 | 0.21281 | 0.19141 | 0.06178 | 0.06517 | 0.0665 |
| | F1-score | 0.0 | 0.09328 | 0.11482 | 0.11404 | 0.32751 | 0.22105 | 0.02575 | 0.03615 | 0.02396 |
| | Precision | 0.21196 | 0.25106 | 0.26067 | 0.16968 | 0.71032 | 0.26064 | 0.04395 | 0.04141 | 0.0 |
| **SGD** | Recall | 0.25 | 0.22338 | 0.12546 | 0.08587 | 0.21281 | 0.19141 | 0.08477 | 0.09467 | 0.0 |
| | F1-score | 0.22941 | 0.23642 | 0.1694 | 0.11403 | 0.32751 | 0.22072 | 0.05789 | 0.05762 | 0.0 |
| | Precision | 0.96463 | 0.98629 | 0.84759 | 0.96087 | 0.82253 | 0.5516 | 0.5009 | 0.56117 | 0.33565 |
| **CRF** | Recall | 0.5 | 0.99977 | 0.84163 | 0.90009 | 0.93935 | 0.62709 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.65862 | 0.99298 | 0.8446 | 0.92949 | 0.87707 | 0.58693 | 0.66747 | 0.71891 | 0.5026 |
| | Precision | 0.91455 | 0.85191 | 0.90156 | 0.9824 | 0.74519 | 0.51499 | 0.33333 | 0.33467 | 0.33333 |
| **Transformer** | Recall | 0.98108 | 0.96463 | 0.79975 | 0.95971 | 0.93717 | 0.67179 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.94665 | 0.90477 | 0.84761 | 0.97093 | 0.83022 | 0.58303 | 0.5 | 0.5015 | 0.5 |
| | Precision | 0.66667 | 0.58878 | 0.9999 | 0.9865 | 0.34128 | 0.57125 | 0.25 | 0.25 | 0.25 |
| **NERLogParser** | Recall | 1 | 0.68887 | 0.52822 | 0.97067 | 0.14389 | 0.66061 | 0.5 | 0.5 | 0.5 |
| | F1-score | 0.8 | 0.63491 | 0.69126 | 0.97852 | 0.20243 | 0.61269 | 0.33333 | 0.33333 | 0.33333 |
| | Precision | 0.50432 | 0.76396 | 0.85017 | 0.98025 | 0.41395 | 0.48211 | 0.62534 | 0.48828 | 0.42242 |
| **BERT** | Recall | 0.5 | 0.89189 | 0.80109 | 0.95466 | 0.77128 | 0.67737 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.50215 | 0.82298 | 0.8249 | 0.96728 | 0.53875 | 0.5633 | 0.76949 | 0.65617 | 0.59394 |
| | Precision | 0.42128 | 0.73406 | 0.75473 | 0.99321 | 0.61062 | 0.48682 | 0.50417 | 0.5242 | 0.64048 |
| **RoBERTa** | Recall | 0.5 | 0.84627 | 0.69669 | 0.98787 | 0.94069 | 0.65753 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.45727 | 0.78618 | 0.72455 | 0.99053 | 0.74054 | 0.55944 | 0.67036 | 0.68784 | 0.78084 |
| | Precision | 0.62296 | 0.4353 | 0.67534 | 0.95767 | 0.50092 | 0.48896 | 0.40595 | 0.39857 | 0.46026 |
| **DistilBERT** | Recall | 0.9932 | 0.73796 | 0.67655 | 0.9536 | 0.81009 | 0.68017 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.76567 | 0.54759 | 0.67594 | 0.95563 | 0.61905 | 0.56893 | 0.57747 | 0.56996 | 0.63038 |
| | Precision | 0.99823 | 0.88927 | 0.91816 | 0.97421 | 0.73274 | 0.53091 | 0.94952 | 0.98457 | 0.5963 |
| **MV Ensmbl** | Recall | 0.5 | 0.8956 | 0.77882 | 0.94409 | 0.89855 | 0.67179 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.66627 | 0.89242 | 0.84277 | 0.95891 | 0.80722 | 0.59309 | 0.97411 | 0.99223 | 0.7471 |
| | Precision | 0.98902 | 0.88486 | 0.93535 | 0.97978 | 0.65471 | 0.50945 | 0.96 | 0.87796 | 0.6192 |
| **SV Ensmbl** | Recall | 0.50621 | 0.94177 | 0.90178 | 0.9559 | 0.9493 | 0.67737 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.66967 | 0.91243 | 0.91826 | 0.96769 | 0.77495 | 0.58153 | 0.97959 | 0.93502 | 0.76482 |
| | Precision | 0.96608 | 0.92648 | 0.88668 | 0.97122 | 0.74514 | 0.51165 | 0.52727 | 0.57715 | 0.42967 |
| **CT Cascade** | Recall | 0.77469 | 0.99575 | 0.87342 | 0.94755 | 0.9432 | 0.67458 | 1.0 | 1.0 | 0.99983 |
| | F1-score | 0.85986 | 0.95987 | 0.88 | 0.95924 | 0.83255 | 0.58193 | 0.69048 | 0.73189 | 0.60104 |
| | Precision | 0.99529 | 0.97442 | 0.96334 | 0.97897 | 0.52754 | 0.5 | 0.96734 | 0.89874 | 0.51203 |
| **CB Cascade** | Recall | 0.5 | 0.99075 | 0.98592 | 0.95511 | 0.94989 | 0.68296 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.66562 | 0.98252 | 0.9745 | 0.96689 | 0.67835 | 0.57733 | 0.9834 | 0.94667 | 0.67728 |
| | Precision | 0.99562 | 0.93764 | 0.95875 | 0.98097 | 0.5398 | 0.49394 | 0.98444 | 0.84034 | 0.42863 |
| **CBT Cascade** | Recall | 0.53814 | 0.9834 | 0.98906 | 0.95706 | 0.94959 | 0.68296 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.69866 | 0.95998 | 0.97367 | 0.96887 | 0.68832 | 0.57327 | 0.99216 | 0.91324 | 0.60006 |

Table 2: Entity Type Evaluation for Out-of-Scope Data.

| **Log files** | **Metric** | **Cisco ASA** | **Cisco IOS** | **Linux Secure** | **Linux Apache** | **NGINX v1** | **NGINX v2** | **WAE** | **WSYE** | **WSE** |
|---|---|---|---|---|---|---|---|---|---|---|
| | Precision | 0.42381 | 0.47429 | 0.33751 | 0.23127 | 0.00446 | 0.24316 | 0.25805 | 0.21623 | 0.08471 |
| **Naive Bayes** | Recall | 0.5 | 0.43115 | 0.18391 | 0.1445 | 0.00135 | 0.20833 | 0.50072 | 0.5 | 0.46633 |
| | F1-score | 0.45876 | 0.45169 | 0.23809 | 0.17787 | 0.00207 | 0.2244 | 0.34058 | 0.3019 | 0.14337 |
| | Precision | 0.0 | 0.0 | 0.19957 | 0.28559 | 0.0 | 0.2847 | 0.03253 | 0.05002 | 0.02923 |
| **Perceptron** | Recall | 0.0 | 0.0 | 0.11571 | 0.1445 | 0.0 | 0.20833 | 0.12356 | 0.13033 | 0.133 |
| | F1-score | 0.0 | 0.0 | 0.14649 | 0.1919 | 0.0 | 0.2406 | 0.0515 | 0.07229 | 0.04793 |
| | Precision | 0.42392 | 0.48457 | 0.38162 | 0.28554 | 0.00451 | 0.28369 | 0.0879 | 0.08282 | 0.0 |
| **SGD** | Recall | 0.5 | 0.43115 | 0.18367 | 0.1445 | 0.00135 | 0.20833 | 0.16954 | 0.18933 | 0.0 |
| | F1-score | 0.45883 | 0.4563 | 0.24799 | 0.19189 | 0.00208 | 0.24024 | 0.11577 | 0.11524 | 0.0 |
| | Precision | 0.96463 | 0.64673 | 0.7944 | 0.95451 | 0.86166 | 0.51843 | 0.5009 | 0.56117 | 0.33565 |
| **CRF** | Recall | 0.5 | 0.65556 | 0.78881 | 0.89413 | 0.98405 | 0.58939 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.65862 | 0.65112 | 0.79159 | 0.92333 | 0.9188 | 0.55163 | 0.66747 | 0.71891 | 0.5026 |
| | Precision | 0.91455 | 0.85191 | 0.65118 | 0.97147 | 0.77298 | 0.46039 | 0.33333 | 0.33467 | 0.33333 |
| **Transformer** | Recall | 0.98108 | 0.96463 | 0.57764 | 0.94903 | 0.97212 | 0.60056 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.94665 | 0.90477 | 0.61221 | 0.96012 | 0.86119 | 0.52121 | 0.5 | 0.5015 | 0.5 |
| | Precision | 0.66667 | 0.58878 | 0.9999 | 0.98568 | 0 | 0.52173 | 0.5 | 0.5 | 0.5 |
| **NERLogParser** | Recall | 1 | 0.68887 | 0.52822 | 0.96986 | 0 | 0.60335 | 1 | 1 | 1 |
| | F1-score | 0.8 | 0.63491 | 0.69126 | 0.97770 | 0 | 0.55958 | 0.66667 | 0.66667 | 0.66667 |
| | Precision | 0.50432 | 0.82059 | 0.66751 | 0.96119 | 0.42848 | 0.41948 | 0.62534 | 0.48828 | 0.42242 |
| **BERT** | Recall | 0.5 | 0.95799 | 0.62897 | 0.9361 | 0.79835 | 0.58939 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.50215 | 0.88398 | 0.64767 | 0.94848 | 0.55766 | 0.49013 | 0.76949 | 0.65617 | 0.59394 |
| | Precision | 0.42128 | 0.76519 | 0.65976 | 0.97625 | 0.62275 | 0.41582 | 0.50417 | 0.5242 | 0.64048 |
| **RoBERTa** | Recall | 0.5 | 0.88216 | 0.60902 | 0.971 | 0.95937 | 0.56164 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.45727 | 0.81953 | 0.63338 | 0.97362 | 0.75525 | 0.47786 | 0.67036 | 0.68784 | 0.78084 |
| | Precision | 0.31361 | 0.51349 | 0.40781 | 0.93537 | 0.48665 | 0.40964 | 0.40595 | 0.39857 | 0.46026 |
| **DistilBERT** | Recall | 0.5 | 0.87052 | 0.40854 | 0.93139 | 0.78701 | 0.56983 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.38546 | 0.64596 | 0.40818 | 0.93338 | 0.60141 | 0.47664 | 0.57747 | 0.56996 | 0.63038 |
| | Precision | 0.99823 | 0.90877 | 0.80367 | 0.96972 | 0.76263 | 0.47241 | 0.94952 | 0.98457 | 0.5963 |
| **MV Ensmbl** | Recall | 0.5 | 0.91524 | 0.6817 | 0.93974 | 0.93519 | 0.59777 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.66627 | 0.91199 | 0.73768 | 0.9545 | 0.84014 | 0.52774 | 0.97411 | 0.99223 | 0.7471 |
| | Precision | 0.98902 | 0.90191 | 0.80733 | 0.97274 | 0.68067 | 0.45168 | 0.96 | 0.87796 | 0.6192 |
| **SV Ensmbl** | Recall | 0.50621 | 0.95992 | 0.77836 | 0.94903 | 0.98695 | 0.60056 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.66967 | 0.93001 | 0.79258 | 0.96074 | 0.80568 | 0.51559 | 0.97959 | 0.93502 | 0.76482 |
| | Precision | 0.9635 | 0.64854 | 0.79289 | 0.96226 | 0.77048 | 0.45339 | 0.52727 | 0.57715 | 0.42974 |
| **CT Cascade** | Recall | 0.77262 | 0.69703 | 0.78103 | 0.93881 | 0.97528 | 0.59777 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.85756 | 0.67191 | 0.78692 | 0.95039 | 0.86087 | 0.51566 | 0.69048 | 0.73189 | 0.60114 |
| | Precision | 0.99529 | 0.67841 | 0.8429 | 0.97224 | 0.54778 | 0.44172 | 0.96734 | 0.89874 | 0.51203 |
| **CB Cascade** | Recall | 0.5 | 0.68979 | 0.86265 | 0.94854 | 0.98633 | 0.60335 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.66562 | 0.68405 | 0.85266 | 0.96025 | 0.70437 | 0.51004 | 0.9834 | 0.94667 | 0.67728 |
| | Precision | 0.99562 | 0.6959 | 0.83825 | 0.97455 | 0.56088 | 0.43434 | 0.98444 | 0.84034 | 0.42863 |
| **CBT Cascade** | Recall | 0.53814 | 0.72987 | 0.86474 | 0.9508 | 0.98668 | 0.60056 | 1.0 | 1.0 | 1.0 |
| | F1-score | 0.69866 | 0.71248 | 0.85129 | 0.96253 | 0.7152 | 0.5041 | 0.99216 | 0.91324 | 0.60006 |