Summarizing Logs with Pre-Trained Language Models

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin,

. . . .

Summarizing Logs with Pre-Trained Language Models

Ovidiu Victor Tătar

A thesis submitted to the
Faculty of Electrical Engineering and Computer Science
of the
Technical University of Berlin
in partial fulfillment of the requirements for the degree
Bachelor of Science in **Computer Science**

Berlin, Germany
10th May 2022

Main supervisor:

Prof. Dr. habil. Odej Kao, Technical University of Berlin

Research advisor:

Dr. Alexander Acker, Technical University of Berlin

Zusammenfassung

Das Verstehen und Eindämmung der Auswirkungen von Fehlern und Ausfällen ist ein unverzichtbarer Aspekt für den Betrieb moderner IT-Infrastrukturen, jedoch macht es alleine der Umfang großer Systeme schwierig, Probleme festzustellen und nachzuvollziehen. Sogenannte Logdateien beinhalten beträchtliche Mengen an Informationen bezüglich des Zustandes eines Systems, allerdings erfordert eine manuelle Untersuchung erhebliche Zeitressourcen und ein fundiertes Verständnis vieler Komponenten; mit steigender Komplexität des Systems wird dies nicht leichter. Um diese Analyse zu vereinfachen, wird in dieser Arbeit die Verwendung vortrainierter Modelle zur Verarbeitung menschlicher Sprache vorgeschlagen, um Logdaten zusammenzufassen. Die Annahme ist, dass eine kompaktere Beschreibung der relevanten Ererignisse eine manuelle Untersuchung einer Logdatei beschleunigen kann, und dass solche Systeme einen besseren Überblick bieten können, wenn Benutzer mit Logdaten interagieren. Weiterhin werden zwei Verfahren zur halb-automatischen Generierung von Zusammenfassungen vorgestellt, um Datensätze zu entwickeln, auf denen Vorgehen zur Zusammenfassung von Logdaten ausgewertet werden können. Der vorgeschlagene Ansatz ist in der Lage, ein bisher existierendes System zur Zusammenfassung von Logdaten auf mehreren zuvor untersuchten Datensätzen zu übertreffen. Insgesamt ist eine durchschnittliche Steigerung der ROUGE-1 F_1 Metrik um 12 Punkte zu beobachten. Dies zeigt, dass moderne Sprachverarbeitungsmodelle zumindest teilweise befähigt sind Logdaten zu verstehen, sodass sie in Zukunft Anwendung bei der Analyse von Logdateien finden können.

Abstract

Understanding and mitigating the impact of failures is essential to operating modern IT infrastructures, though their large scale alone makes it challenging to diagnose any problems. Logs supply crucial information on system state and behavior, but analyzing them manually requires substantial time investments and in-depth knowledge about the system; it becomes difficult for human operators to understand them with rising system complexity. To ease the analysis, we suggest using pre-trained *natural language processing* (NLP) models to summarize logs and propose two summarization tasks to construct reference summaries from logs semi-automatically. We assume that a concise representation of the relevant events in a log speeds up the analysis conducted by human operators. Additionally, such summarization systems may provide a better overview anywhere users interact with log-data. Our approach performs comparative to previous work on previously studied datasets for log summarization but outperforms it significantly on datasets the previous approach finds more difficult: Overall we achieve improved ROUGE-1 F_1 scores of 12 points on average. This shows that current NLP models are at least in part able to transfer their knowledge to the domain of log-data; Thus, their application may prove helpful to future research regarding log analysis.

Contents

List of Figures

List of Tables

List of Acronyms

AI artificial intelligence

AIOps artificial intelligence for IT operations

BOS beginning of sequence (referring to tokens for NLP models)

CPU central processing unit

EOS end of sequence (referring to tokens for NLP models)

FSL few-shot learning

GPU graphical processing unit

IT information technology

ML machine learning

MLM masked language modeling

MTTF mean time to failure

MTTR mean time to recovery

NLP natural language processing

Pre+FSL self-supervised pre-training & few-shot learning

Pre+ZSL self-supervised pre-training & zero-shot learning

RCA root cause analysis

ROUGE Recall-Oriented Understudy for Gisting Evaluation (name of a metric used in text summarization)

SEP separator (referring to tokens for NLP models)

seq2seq sequence-to-sequence (model type)

ZSL zero-shot learning

1 Introduction

With today's complex, increasingly distributed and decentralized IT systems, it becomes more and more challenging to determine when a failure occurs and tougher still to decide what caused said failure. As such it is inevitable that systems fail from time to time. Fulfilling the high availability requirements of modern services makes it necessary to mitigate the impact of failures and resolve problems swiftly when they emerge. To achieve this it is vital to understand the causes and effects of a failure. However, manual investigations often require extensive system knowledge and thus substantial time investment by both developers and infrastructure operators.

Computer-aided alternatives become a desirable solution. So called AIOps (*artificial intelligence for IT operations*) systems follow different automated approaches to aid system engineers in maintaining their systems; *Anomaly detection* aims to identify potentially erroneous system states and failures, while automated *root cause analysis* (RCA) tries to determine the faults and error events that led to the failure. To investigate failures, AIOps systems autonomously gather and process information from different sources.

One such information source are *log files*, which are text files where applications and operating systems inform about the system state. These are usually intended to be comprehensible to human operators and facilitate the process of debugging, allowing humans to gain insights into what went wrong [1]. Still, it remains up to the system (specifically its developers) which information is logged in which way. Considering that logs are unstructured text files intended to be human-readable, containing important information about the system state that developers and operators can access [1], they become increasingly difficult to analyze by humans with rising system complexity and logging volume. As logs contain rich information on the system state and events, yet much effort is needed to analyze them by hand, automatic solutions are becoming more popular.

The focus we approach this problem with is different from most solutions towards facilitating log analysis: We aim to construct a program with the ability to extract the most impactful information from a segment of log-data. This is an approach we will refer to as *log summarization*.

To us it seems appealing to facilitate the process of summarizing logs, as summaries may prove useful anywhere where a human interacts with log-data: A user attempting to examine unexpected application behavior on their own, an AIOps system presenting its results to a human operator, or indeed a developer or operator performing an in-depth investigation of a system failure. In this regard, previous research found that providing a concise, easily-readable representation of logs can be helpful for human operators and substantially speeds up their ability to inspect and analyze logs manually [2].

Moreover, log-driven AIOps solutions will often aim to identify portions of the logs related to a failure and differentiate them from log-data generated under normal execution conditions. A model that summarizes log-data may also be capable of understanding which portions of its input contain important information, such as any unexpected system behavior observed. Thus, models solving the problem of log summarization may also be able to transfer their knowledge to other tasks of automated log analysis.

Unfortunately, the unstructured nature of logs makes it hard for programs to handle log-data. As system developers write free-text portions in logs, we believe that applying methods that can understand the human language promises a high potential of gaining new insights, otherwise inaccessible by only focusing on the machine-readable parts of logs.

Recent advances in *natural language processing* (NLP) have made it possible to train models that show a general understanding of human language [3, 4], and models which generate near human-level summaries on vast datasets of news articles [5]. Additionally, current approaches quickly adapt to novel situations: Zhang *et al.* found their models beat previous state-of-the-art approaches on 6 out of 12 summarization datasets despite training with only 1000 examples [5].

While there have recently been some attempts to summarize logs automatically [6, 2], large pre-trained NLP models are yet to be studied in the domain of summarizing logs. The previous approach to textual summarization of log-data only considers NLP models specialized in the domain of log-data, not benefiting from the generalized language understanding of pre-trained models. In addition, their approach is evaluated on segments of 20 log lines [6]. However, large-scale distributed systems can generate over 500 thousand [7, 8] or even 120 million [9] lines per hour. Thus, approaches should also be evaluated on longer inputs.

We aim to contribute toward the goal of automated log analysis by generating summaries of log-data. Our contribution is threefold:

- We propose two summarization tasks based on the idea of summarizing failures, which can be applied to summarize sections of logs encompassing multiple hundreds of log-entries.
- We further tune pre-trained NLP models and evaluate them on these novel summarization tasks, as well as on a previously researched dataset containing human-written summaries of log data. Our best-performing model outperforms the state-of-the-art framework [6] on these previously researched datasets.
- We highlight several challenges that arise when applying pre-trained language models to the domain of logs.

Going forward, this thesis is structured as follows: We first introduce the reader to the study of reliable systems and the concepts of NLP in chapter 2. In chapter 3 we lay out important aspects regarding our contribution and then present our approach to log summarization. We then extensively evaluate our approach in chapter 4, first introducing evaluation metrics and datasets, then detailing the experiments conducted. Finally, we discuss related work in chapter 5 and conclude this thesis in chapter 6.

2 Background

2.1 Reliability Engineering

Until the 1960s, the quality targets of a component could be deemed fulfilled if the component in question was free of defects and failures at the time of development or manufacturing [10].

But with rising complexity and costs of failure " [t]he expectation today is that complex [. . .] systems are not only *free from defects and systemic failures* [. . .] when they are put into operation [. . .], but also *perform the required function failure free* for a stated time interval" [10, p. 1].

This is where reliability engineering comes in:

> The purpose of *reliability [. . .] engineering* is to develop methods and tools to *evaluate and demonstrate* reliability, maintainability, availability, and safety of components [. . .], as well as to *support* development and production engineers in *building in* these characteristics. [10, p. 1]

We say a system experiences a *failure* if it no longer performs its desired function and is thus in violation of its requirements [10].

A high *availability* has long been an essential requirement to many IT systems, critical to their quality of service [11]. Formally, availability represents the probability that a system is able to provide its service correctly at any given point in time [12]. According to Birolini [10] the average availability $\bar{A}$ of a system corresponds to:

$$\bar{A} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \tag{2.1}$$

Thus main contributors to availability are *mean time to failure* (MTTF), the average duration of time after which a system is expected to fail, starting from an initial operational state, and *mean time to recovery* (MTTR), the average duration of time needed to restore the system to the operational state without failure.

Therefore, a desirable goal is to keep MTTF high by proactively prohibiting the occurrence of failures while lowering the MTTR, for example by mitigating the impact of a failure.

One measure to achieve this is *root cause analysis* (RCA), as it is "designed to help identify not only what and how an [undesired] event occurred, but also why it happened. Only when investigators are able to determine why an event or failure occurred will they be able to specify workable corrective measures that prevent future events of the type observed." [13, p. 45] According to Rooney *et al.* RCA is a process involving the following steps:

1. Data collection.
2. Causal factor charting.
3. Root cause identification.
4. Recommendation generation and implementation.

[13, pp. 46–48]

The process of collecting data is the most time intensive, heavily influenced in its scope by the construction of the causal factor chart. Concurrently, investigators prepare and perform causal analyses, building a causal graph of events leading up to the failure, which later helps identify the major contributors to the unwanted event [13].

2.2 Artificial intelligence for IT operations and Log Analysis

Lying at the crossroads of development and operation, *DevOps* has seen widespread adoption in the industry in recent years, actively embraced by Tech Giants such as Google or Amazon [14, 15]. DevOps is a concept which thrives on increased collaboration between the traditionally separate IT departments of development, quality assurance and operation [15]. It can be understood as a paradigm where IT professionals from different backgrounds work as cross-functional teams as opposed to the conventional approach of independent functional silos [15, 14].

Aiello *et al.* understand rapid software deployment in accordance with business needs as an explicit goal of the set of principles and practices laid out by DevOps. There is a strong focus that deployment of changes happen automatically, increasing the reliability of a system by design [14].

In operations, DevOps is focused with maintaining the stability and performance of a system. As such it employs *logging* and monitoring frameworks and tools, that help to uncover and address potential problems before they affect customer experience [15]. These commonly present operators with visual dashboards that display a summary of the overall system state and automatically alert administrators if further investigation and action is needed [15].

This is where recently the idea of AIOps (*artificial intelligence for IT operations*) has started to gain some interest, with the number of related publications published each year consistently rising [16, 17]. As the concept of AIOps is still relatively young, there is not yet a widely agreed-upon definition of it [16, 18].

> In general [however], AIOps is about empowering software and service engineers to [. . .] build and operate services that are easy to support and maintain by using artificial intelligence and machine learning techniques. [18, p. 4]

> Loosely speaking, artificial intelligence (AI) is a branch of computer science that aims to build systems that require human intelligence. [19, p. 14]

Rijal *et al.* conducted a literature review of scientific and non-scientific work focused on AIOps [16]. They point out several reported benefits of AIOps, among which:

◇ time saving; IT professionals can spend less time on routine monitoring and more of it on innovative tasks

◇ improved collaboration between IT teams and other business units, lowering the effort needed to gain an overall understanding of a problem

◇ faster investigation of potential problems, thus predicting failures before they impact a service's quality (potentially longer MTTF)

◇ automatization of RCA and suggestions of possible solutions, leading to shorter MTTR

Overall, AIOps has the potential to increase the reliability and availability of an IT system by helping to identify problems and providing options in the decision-making process [20].

A significant amount of methods related to AIOps rely on logging systems as an information source [17]. The *analysis of logs* promises to provide great insights into a system's behaviors, as logs typically record detailed information about system state and any events influencing the system, enabling various diagnoses, from investigating performance bottlenecks to detecting security breaches [1, 8].

However, gathering information from logs remains challenging, as noted by Oliner *et al.* [1]. These are some of the identified challenges:

◇ A developer writes log messages in a particular context of the program source code, which is lost in the log file, making them difficult to understand.

◇ Log messages from different system components may be interleaved in a single log file.

◇ Log volume may be excessive in large systems, so it becomes impractical to record the management of *every* resource, such as lock objects.

◇ Different components may use heterogeneous formatting for logging.

From the perspective of AIOps, system administration can come in different levels of automatization and autonomy the AI system, starting from a regular workflow where the computer is merely a non-intelligent actor executing the commands of a human operator (level 1), ranging over AI-generated recommendations a human operator can choose to follow or ignore (level 2), all the way to AIOps systems that take actions on their own and can make self-determined changes to a system (level 5/6) [20].

A system summarizing logs could be considered (part of) a level 2 AIOps system, providing a human operator with a condensed overview of what happened. Especially for operators trying to investigate system failures, this may help to get a better understanding of the situation and ease communication to other teams unfamiliar with the details of the problem. Altogether, providing a concise, easily-readable representation of logs can be helpful to drastically speed up the analysis of log-data by human operators [2].

However, applying methods of artificial intelligence is not straightforward, because most general-purpose logs represent unstructured text, written by and designed for humans in order to better understand the systems behavior. Free-text log messages cannot be simply converted into numerical representations a computer can reason upon [1].
Here is where the field of NLP can play a vital role in log analysis.

2.3 Natural Language Processing

Natural language processing (NLP) is "an area of computer science that deals with methods to analyze, model, and understand human language." [19, p. 4]

During the early years of NLP (around the 1960s), research in this field was "dominated by a *rationalist* approach [...][,] characterized by the belief that a significant part of the knowledge in the human mind [...] is fixed in advance." [21, pp. 4–5] As such, rule-based systems saw great application in NLP and other research fields of artificial intelligence. Typically these had a lot of handcrafted knowledge and heuristics built into them [21, 19]. This situation is similar to that encountered in *log analysis* even today, where domain knowledge and heuristics empower tools to provide more accurate and effective diagnoses [1].

In the past decades though, an empiricist approach to NLP has gained traction, which "suggests that we can learn the complicated and extensive structure of language by specifying an appropriate general language model, and then inducing the values of parameters by applying statistical, pattern recognition, and machine learning methods" [21, p. 5].

Increasingly NLP thus applies methods originating in the field of *machine learning* (ML) [19]:

> The goal of ML is to "learn" to perform tasks based on examples (called "training data") without explicit instruction. This is typically done by creating a numeric representation [...] of the training data and using this representation to learn the patterns in those examples. [19, p. 15]

An area of ML that has lately seen great success in diverse areas of artificial intelligence research is *deep learning* [22, 19], which combines multiple parametric processing layers and updates internal parameters according to the optima of a given loss function. These models are able to learn from unstructured, large datasets and create representations of the underlying information [22]. Conventional machine learning methods require domain knowledge to extract structured data (*features*) from raw data, which could be used as the numeric representation for a learning algorithm to work on. Unlike these, deep learning models can learn suitable representations automatically from training on raw data [22]. As "language is inherently complex and unstructured" [19, p. 22], deep learning architectures "have become the status quo in NLP" [19, p. 23].

Language models One central component of many modern NLP approaches is to train a model to solve the task of language modeling, where the model is asked to predict the next word given a context of previous words. This is a well-studied problem for which statistical models can be developed. Implementations of such *language models* are not usually bound to this specific task. They provide a probability distribution over the sequences of words of a language and can thus be applied to diverse NLP tasks [21].

This concept can also be generalized to bidirectional language models, which are able to predict sequences of words missing in a given context of words, not just any subsequent words [3, 23, 24].

In machine learning, one differentiates between *unsupervised* learning algorithms, where the classification of an training example is unknown to an algorithm and it tries to understand the underlying patterns of the training data, and *supervised* ones where the algorithm requires additional *labels* for training, depicting the desired output for each particular input example [21, 19].

One advantage of using language models is that training can often be realized in an unsupervised manner, bypassing the requirement of a large corpus of labeled data that may be expensive to construct. A way to do this is to remove words randomly in a text and ask the model to guess which words are missing [3]. This style of learning approach is sometimes referred to as *self-supervised* [25], as a model still requires knowledge about the desired output for each input example. However, the desired output is simply the text that was withheld from the model's input. Therefore the labels for the self-supervised training are provided by the unlabeled data itself.

Dai *et al.* proposed using such self-supervised learning approaches to train a deep learning model and showed empirically that doing so significantly improves performance for supervised learning tasks compared to directly training a model on the same task [26]. From the perspective of the broader field of artificial intelligence, this is an application of *transfer learning*, where knowledge is first gained in a certain domain or task but is then applied (effectively) to other problems related to the original task [19]. It has now become standard practice to first *pre-train* a large model with millions of parameters on a self-supervised language modeling task using a large corpus of unlabeled text data, with the assumption that it forms some generalized knowledge about language which it can then transfer on the actual "downstream" task after some supervised learning (*fine-tuning*) [3, 4, 5]. These models can be seen as semi-supervised learners, as they combine unsupervised pre-training on a large dataset with supervised fine-tuning on a smaller dataset [19, 26].

Transformers In recent years the *transformer*-architecture [27] introduced by Vaswani *et al.* has been a central part of research in the field of NLP and machine learning in general.[1] Open-access transformer-based models such as those provided by HuggingFace's *Transformers*-library [28] can be accessed publicly and used in production systems.

Traditionally transformer architectures follow a *sequence-to-sequence* (seq2seq) [29] encoder-decoder structure, which combines an *encoder*, mapping input elements to internal continuous

[1] The original article has been cited over 40000 times according to Google Scholar: `https://scholar.google.de/scholar?cluster=2960712678066186980` (last accessed on 27th April 2022)

numerical vector representations, and an autoregressive *decoder*, generating output elements from internal vector representations and previous outputs. (The dependency on its previous outputs is what classifies the decoder as autoregressive.) Both components are separate language models mainly composed of several stacked multi-head self-attention layers [27].

For an in-depth explanation of the *self-attention* mechanism and its role in the transformer architecture the reader is referred to the original article [27], but all in all it allows the transformer architecture to investigate words by their context [19], while maintaining the possibility for direct dependencies between segments separated by longer text passages [27]. For many practical applications, self-attention layers exhibit a better computational complexity than the typical alternatives [27], allowing transformer models to be scaled up to bigger sizes with more layers and parameters than alternative deep learning architectures.

Naturally it is impractical for deep learning methods to work on arbitrary text in the form of character sequences directly. Instead, inputs and outputs generated by a transformer model are numerical vectors called *embeddings* [27]. Each *token* in a model's vocabulary can be represented as a unique embedding and the embedding outputted by the decoder represents a probability distribution over the most probable subsequent tokens [27], making it a language model. In the context of NLP, tokens often represent singular words or sentence components [21], but deep learning architectures need a fixed-size vocabulary and therefore need to be able to represent words not previously seen. More sophisticated approaches such as SentencePiece [30] allow words to be split into multiple tokens in a robust and unique manner [31]. Transforming sequences of tokens from or into actual text sequences is usually handled by a *tokenizer* separate from though related to the main transformer model [28].

Summarization One example of a typical sequence-to-sequence task is *summarization*, where a model is given an input sequence (the document to summarize) and is expected to produce an output sequence (the summary). In some domains *extracting* important sequences based on heuristics works well for writing summaries. For instance, when summarizing news articles, selecting the first three sentences (a heuristic known as Lead-3) forms an acceptable summary due to the layout bias present in news media [32]. Nevertheless, summarization remains a challenging task in NLP, especially on datasets requiring *abstractive* summarization, including novel words not present in the original document. Recent transformers have therefore significantly outperformed previous state-of-the-art models on many datasets [4, 5].

BERTSum [33] represents one of the earliest applications of transformer-based models to summarization tasks. Based on a pre-trained bidirectional encoder (BERT [3] more specifically), it introduced both an extractive summarization model using a classification-network on top of the encoder, and an abstractive sequence-to-sequence summarization model using an autoregressive decoder on top [33].

Since then several seq2seq transformer models have been proposed for abstractive summarization. Among the most performant ones are BART [4] and PEGASUS [5], both using architectures analogous to the seq2seq transformer architecture described in [27], but with differing pre-training objectives.

3 Contribution

We first explain some concepts surrounding logs and the processing of log-data (section 3.1). Using these basics, we define summarization tasks that serve as the foundation of our experiments (section 3.2). Afterward, we present the models we use for summarization (section 3.3) and give an in-depth overview of our approach (section 3.4). In the final sections of this chapter, we explain the commonly used cross-entropy as a function to optimize during training and how to interpret the related perplexity (section 3.5). We end with a brief explanation of how beam-search is employed to generate text, as well as the concept of a hyperparameter search (section 3.6).

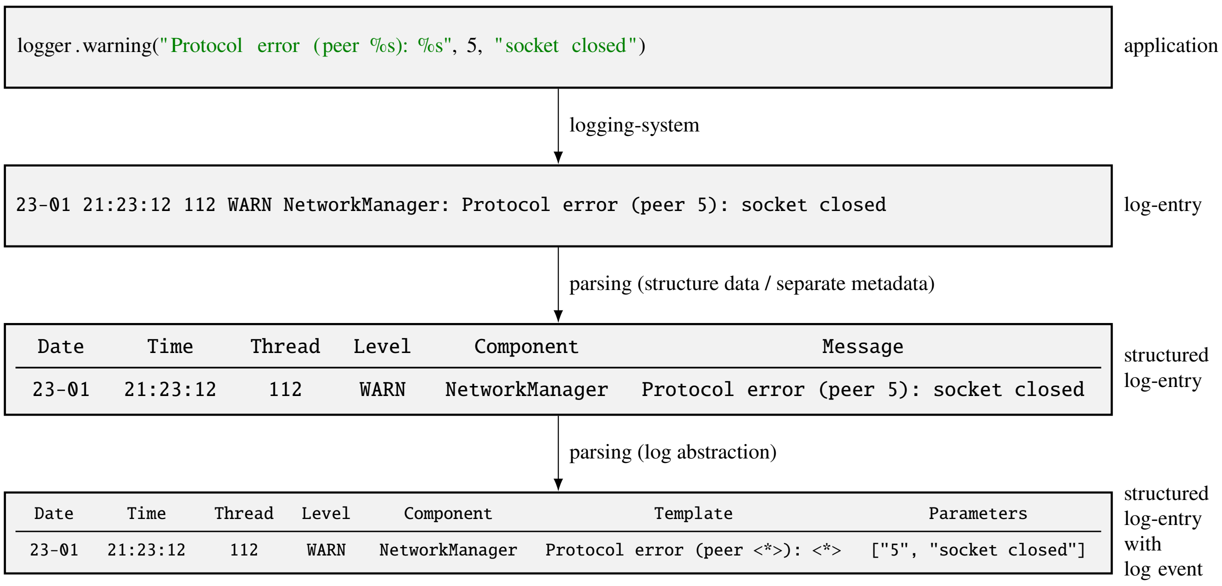
3.1 Logging Systems and Log Parsing

In its simplest form, a log is a text file to which a software system writes arbitrary messages with the goal to record information about its state and activities. The process of keeping logs is known as *logging*, while we will refer to the part of the software managing the logging operation as a *logging-system*. Most general-purpose logs represent unstructured text designed by developers to be human-readable, making it hard for computers to understand these free-text messages [1].

Since the system decides which information is included in a log and how it is formatted, programs analyzing log-data usually have to be individually configured for each system.

Nevertheless, different logging-systems generally exhibit some common properties that can be leveraged for automated log analysis. This is also illustrated using a hypothetical example in Figure 3.1: An application first makes a call to its logging-system, which will fill in *parameters* into given *templates* to produce a plain text message (*log message*) and add additional metadata such as timestamps, the ID of the executing thread or software-components issuing the log message, finally outputting the log-entry. In a sense, log parsers try to undo this process, usually by first separating metadata from the log message.

From a log parser's perspective, logs consist of *log-entries*, each usually contained in its own



```
logger.warning("Protocol error (peer %s): %s", 5, "socket closed")
```

```
23-01 21:23:12 112 WARN NetworkManager: Protocol error (peer 5): socket closed
```

Date	Time	Thread	Level	Component	Message
23-01	21:23:12	112	WARN	NetworkManager	Protocol error (peer 5): socket closed

Date	Time	Thread	Level	Component	Template	Parameters
23-01	21:23:12	112	WARN	NetworkManager	Protocol error (peer <*>): <*>	["5", "socket closed"]

application
logging-system
log-entry
parsing (structure data / separate metadata)
structured log-entry
parsing (log abstraction)
structured log-entry with log event

Figure 3.1: Example of how applications produce logs and how parsers can structure the contained information.

line and formatted in a predictable manner. These entries contain a plain text message (*log message*) as well as some metadata; parsers are able to use the underlying patterns of log-entries and extract the information into a structured format, such as CSV or JSON. Additionally, in a process called *log abstraction*, parsers separate static parts (*templates*) from dynamic parts (*parameters*) of the log message, categorizing log-entries by the static parts and thus assigning each log-entry to a *log event* [34].

We leverage the existing technology of log abstraction to determine log-entries with similar meaning as represented by log events, which play a central role in the log summarization task we propose.

3.2 Summarization Tasks

A program with the ability to extract the most important and impactful log-entries within a group of log-entries, would construct a summary of said log-entries. This is the approach we will refer to as *log summarization*.

While standardized tasks and datasets exist for text summarization in general,[1] the same cannot be said for the summarization of log-data.

Such a summarization task seems necessary to train or fine-tune performant models; Even more so as it seems plausible that log-data is linguistically different from news corpora and other domains that are frequently used in summarization benchmarks. Texts in log-entries often do not form complete sentences, may be unrelated to surrounding log-entries as opposed to continuous text, and may contain additional metadata. Furthermore, evaluating summarization

[1] For example XSum [35] and CNN/DailyMail [36] represent summarization datasets constructed from large news corpora. For a list of datasets and benchmarks see `https://paperswithcode.com/task/text-summarization`.

results without a dataset containing reference summaries is not practical as the quality of model outputs can then only be manually judged by *system experts* (developers, operators, maintainers, . . .).

Previous work related to producing summaries of log-data relied on manually written summaries; Meng *et al.* summarize 100 groups of 20 successive log-entries per dataset [6]. However, writing summaries for logs manually requires knowledge about the system, is susceptible to an author's biases and mistakes, represents a substantial time investment, and therefore does not seem feasible to create enough labeled data to train a supervised model.

As Meng *et al.* point out themselves, even system experts might not even be aware of all aspects of a diverse software system, as large-scale services are usually developed and operated by hundreds of IT professionals. In this case, developers or operators analyzing the log cannot have complete knowledge of all components and underlying contexts [6].

Additionally, 20 log-entries is a relatively short sequence, as large-scale distributed systems can generate over 500 thousand [7, 8] or even 120 million [9] log-entries per hour. As logs can get quite long, even only considering short periods of time which may be relevant to a failure (a few seconds of system runtime may already equate to hundreds or thousands of log-entries according to the above figures), we believe it to be necessary to evaluate summarization models on longer sequences of log-entries.

Hence we propose novel summarization tasks on log-data, which can be applied to new datasets using mostly automatic means. For constructing reference summaries, we combine only log-entries corresponding to certain log events. We have two methods for selecting relevant log events, which necessitate different assumptions from each log dataset. We define them as two separate summarization tasks:

common log events all events that are common between multiple occurrences of the same failure are selected for constructing the summary

non-normal log events all events that do not occur during normal execution are selected for constructing the summary

Generally speaking, both approaches to constructing reference summaries require a larger collection of logs and cannot be applied to individual, unrelated log-entries.

We first detail how we apply log parsing in the context of these tasks, then describe each summarization task in greater detail.

Log parsing at component-level Our approaches are based in large parts on the existing concept of log parsing, abstracting log messages into events, as it allows for comparing logs regardless of different time-stamps, threads, parameters and other variable parts of log-entries.

However, as opposed to applying log parsing globally, if log-entries originate from different components of a system, we choose to parse them separately: Even if two components produce the same log message, these will be handled as two different log events.
We believe this to be beneficial for the following two reasons:

1. Different components in a system represent different points of origin for a log-entry; A monitoring system running out of memory is usually more severe than a sensor driver doing

the same. Preserving this information by handling entries from different components distinctly seems appropriate.

2. If two system components produce the same message, it may be by coincidence; Components performing different tasks in a system may produce similar messages, but it seems unlikely that these represent the same underlying event or system-state. For instance, the trivial message `Successfully completed operation` has a different meaning for a database component compared to a component parsing user data. Therefore in the majority of cases, similar messages from different components will represent distinct events and should be treated as such.

The component representing the origin of a log-entry is often explicitly mentioned in logs, so handling each component separately comes at no additional cost for us; Nevertheless, in a setting where linking log-entries to their components is impracticable or our argumentation does not hold (e.g. components do not perform distinct tasks), parsing could also be applied globally without major loss of applicability.

3.2.1 Summarization based on *common log events*

This summarization task is based on the following prerequisite: *Logs corresponding to a failure are grouped by the underlying root cause.*
Each failure will have an underlying root cause that can be identified by system experts. When comparing logs from different logs of the same root cause, there will be parts of overlapping information. Using parsers such as *Drain* [37], events represented by entries in logs can be determined. This enables us to find the log events that are common between all logs with the same root cause (*common log events*). As these events are present every time the failure happens, we expect a high correlation factor between the occurrence of the failure and the common log events.

We hypothesize that the log-entries corresponding to these common log events are of high importance and describe the underlying error events and root causes. It follows that every summary of the logs contained in such a group of the same underlying root cause should contain at least parts of the information from the common log events.

In summary, we propose the following process for collecting reference summaries from logs using *common log events*:
Log-entries from multiple logs with the same root cause are categorized into log events using a log parser. Then log events that are common to all logs are computed as the largest intersection between the sets of events from each log. After applying an optional preprocessing step, the log-entries of each log represent the documents to summarize, while the log-entries whose log events correspond to the set of common events represent the reference summaries.

Given the log, the task is to produce a summary that should closely match the sequence of log-entries corresponding to the set of log events similar logs have in common. An overview of the process of collecting documents and reference summaries from logs is shown in Figure 3.2.

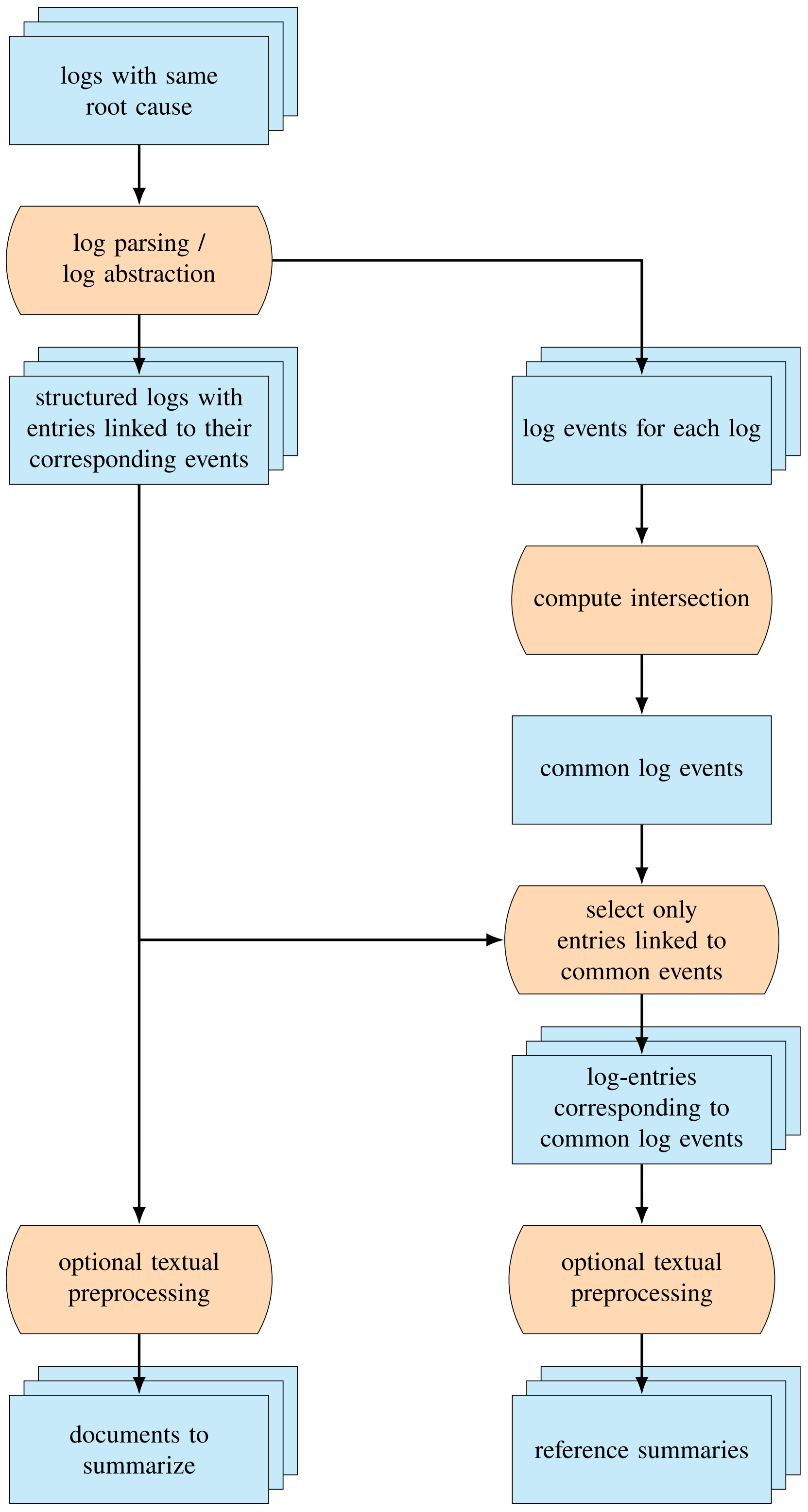

logs with same root cause
log parsing / log abstraction
structured logs with entries linked to their corresponding events
log events for each log
compute intersection
common log events
select only entries linked to common events
log-entries corresponding to common log events
optional textual preprocessing
optional textual preprocessing
documents to summarize
reference summaries

Figure 3.2: Overview of the process of gathering documents and reference summaries for the summarization task based on *common log events*. Data in the above image is marked in a light blue color, while processes are colored slightly orange.
A log parser is used to identify events corresponding to each log-entry, which can then be used to collect all events that occur in every log file (*common log events*). As a basis for reference summaries we then use the entries corresponding to these events, while all entries form the basis for documents to summarize.

3.2.2 Summarization based on *non-normal log events*

This summarization task is based on the following prerequisite: *Logs are categorized into whether any failure occurred (abnormal) or not (normal execution). Both categories of logs cover roughly the same usage scenario and granularity.*
This categorization may be easier to conduct, as no knowledge about underlying root causes or a preceding in-depth *root cause analysis* (RCA) is needed. Given a suitable monitoring system, this categorization could even happen automatically. Furthermore, by manually triggering failures, a large collection of logs adhering to the above criteria could be constructed.

Again, log parsing is employed to determine events represented by log-entries. This time, it identifies all log events that can occur during normal execution and distinguishes them from any log events that do not (*non-normal log events*).

We suspect that the most impactful information is contained in log-entries whose corresponding events do not occur during normal operation and that those log-entries are symptomatic of the failure. As we assume all other log-entries to be of lesser importance, normal and abnormal logs must roughly cover the same application flow; Otherwise, this approach will not work well.

For example, if the normal logs only contain communication with an API, but the abnormal logs record some sensor data, all log-entries observing sensor data will (wrongly) be assumed to be important.
Conversely, if the normal logs contain events more events than necessary, also encompassing events unrelated to the abnormal logs, the results can be just as poor. For instance, it is abnormal for a system with no network interfaces to attempt to create a connection to an external service, while it may be entirely normal for any other system to do so. If the category of normal logs contains log instances from both systems, attempted abnormal network connections may (wrongly) be interpreted as irrelevant.

Following our assumption, we determine that every summary of an abnormal log should contain at least parts of the information from log events that do not occur during normal operation. No summaries are generated for the logs without failures (normal logs).

In summary, we propose the following process for collecting reference summaries from abnormal logs using *non-normal log events*:
Log-entries from multiple logs are categorized into log events using a log parser. Then log events occurring during normal operation are computed as the union between the sets of events from each normal log. After applying an optional preprocessing step, the log-entries of each abnormal log represent the documents to summarize, while the abnormal log-entries whose log events do not correspond to the set of normal events represent the reference summaries.

Given an abnormal log, the task is to produce a summary that should closely match the sequence of log-entries corresponding to the set of these log events which are unique to abnormal logs. An overview of the process of collecting documents and reference summaries from logs is shown in Figure 3.3.

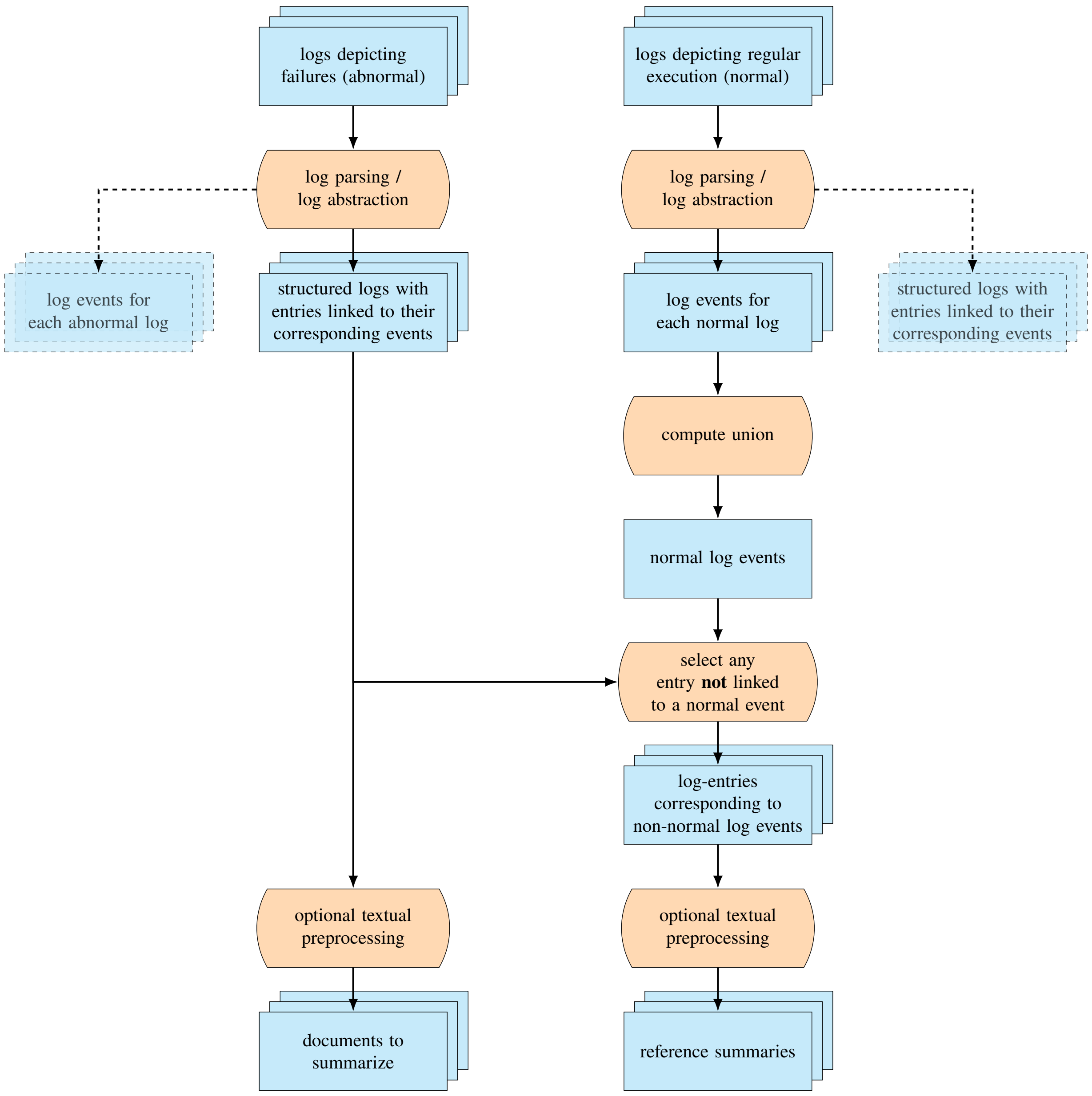

logs depicting failures (abnormal)
logs depicting regular execution (normal)
log parsing / log abstraction
log parsing / log abstraction
log events for each abnormal log
structured logs with entries linked to their corresponding events
log events for each normal log
structured logs with entries linked to their corresponding events
compute union
normal log events
select any entry **not** linked to a normal event
log-entries corresponding to non-normal log events
optional textual preprocessing
optional textual preprocessing
documents to summarize
reference summaries

Figure 3.3: Overview of the process of gathering documents and reference summaries for the summarization task based on *non-normal log events*. Data in the above image is marked in a light blue color, while processes are colored slightly orange.
The process of log parsing is used on both categories of logs, but only the events belonging to normal logs and the entries of the abnormal logs are of greater interest. Now we can identify which events can occur during normal operation of the system and collect all log-entries that *do not* correspond to normally occurring events as the basis of our reference summaries.

3.3 Summarization Models

Recently, substantial advances have been made towards automated abstractive summarization by employing *sequence-to-sequence* (seq2seq) transformer models. Among the most performant ones are BART [4] and PEGASUS [5], both using architectures analogous to the architecture described in [27].

As these models show good performance while maintaining a (relative speaking) straightforward model architecture, we chose these models as a baseline for our experiments. A short overview of the two models, their methodology, and their differences follows:

BART Performing well in multiple *natural language processing* (NLP) tasks, not only summarization, BART is an “allrounder”. It performs as well as specialized models in discriminative tasks and language understanding, comparatively well in dialogue generation, question answering and translation, and excels at summarization, outperforming previous models such as BERTSum [4].

BART has been pre-trained in the same environment as the RoBERTa-model, using text corpora from the domains of news articles, webpages, wiki articles. [38].

Lewis *et al.* consider and evaluate different approaches to what they call *document corruption* for pre-training their BART-model. After an empirical study they decide on two corruption methods used in the pre-training objective of the final model:

> **Text Infilling** A number of text spans are sampled, with span lengths drawn from a Poisson distribution ($\lambda = 3$). Each span is replaced with a *single* `[MASK]` token. 0-length spans correspond to the insertion of `[MASK]` tokens. [. . .]
>
> **Sentence Permutation** A document is divided into sentences based on full stops, and these sentences are shuffled in a random order.
>
> [4, p. 7873]

Therefore BART’s pre-training objective can be understood as a self-supervised learning task. The model is trained to predict which (possibly empty) text sequences are missing in a text where a mask-token has been inserted.

PEGASUS On the other hand, PEGASUS is a model specifically designed with abstractive summarization in mind. The final model outperforms all previous state-of-the-art models (including BART) on nearly all observed datasets [5]. In surveys where human judges were asked to assess the quality of summaries on a numeric rating scale, the authors found their PEGASUS-models generate summaries that are at least as good as (human-written) references in the XSum and CNN/DailyMail datasets [5].

Zhang *et al.* introduce a new pre-training objective tailored to the summarization task which they call *gap sentences generation*:

> In PEGASUS, important sentences are removed/masked from an input document and are generated [. . .] from the remaining sentences, similar to an extractive summary. [5, p. 11328]

It should be noted that contrary to BART, PEGASUS only generates the masked sentences, not the full reconstructed inputs [5].

Similar to BART, the pre-training objective used for PEGASUS is a self-supervised learning task. The model is trained to predict which sentences are missing in a text where a mask-token has been inserted instead. Furthermore, PEGASUS was also pre-trained in the domain of news articles and webpages, although the exact datasets used vary from those used with BART.

The difference lies with the selection of text sequences to be masked:
Different methods for selecting the sentences (random n, first n, $\ldots$) to mask have been evaluated empirically. For the final pre-trained models, sentences are ranked by importance by calculating ROUGE-1 F_1-scores between the selected sentence and the rest of the document, and a share of the top-ranked sentences are masked for pre-training. [5].

3.3.1 Extractive and Abstractive Summarization

Although we propose using models that would be considered *abstractive* summarization models, the summarization tasks we described in the previous section lead to reference-summaries which strongly encourage an *extractive* approach to summarization [32]:

Extractive summarization constructs summaries by combining fragments from the input-text. Extractive models can be constructed as binary classifiers, deciding whether to include a text fragment in the summary, or not.

Abstractive summarization is able to understand and paraphrase the information contained in the input-text and hence produce summaries of different phrasing and wording. Sequence-to-sequence models are fit for this task, as they are able to process any text input and produce arbitrary text output.

Abstractive models have the benefit of being able to produce summaries whose quality is not limited to that of the input text. Sequence-to-sequence models can be trained to perform well on multiple tasks simultaneously, not just for summarization, making them more versatile. For instance, BART outperformed not only previous summarization models, but also scored comparable to specialized models in a varaity language understanding and generation tasks [4].

Using pre-trained abstractive summarization models for our task is attractive since models like PEGASUS are already capable of producing outputs comparable in quality to summaries produced by humans for selected datasets. Zhang *et al.* conducted surveys where human judges were asked to assess the quality of summaries on a numeric rating scale and found their models generate summaries that are at least as good as references in the XSum and CNN/DailyMail datasets [5].

3.4 Approach

Our goal is to fine-tune and evaluate a *natural language processing* (NLP) model that is able to summarize log-data. Assuming the trained model can understand log-data well enough, it may be able to generalize well to other tasks in the context of log analysis. For a model to be able to generalize to other such tasks, it needs to be able to produce arbitrary textual outputs. Therefore we primarily focus on abstractive sequence-to-sequence models.

To evaluate models in their capability to summarize logs, we introduced summarization tasks on log-data in section 3.2, which we now implement with a concrete architecture:

1. We use the log parser Drain [37] as made publicly available[2] by IBM. Drain was previously found to be one of the state-of-the-art log parsers, both in terms of accuracy and efficiency [8]. It preprocesses log-entries using few (usually one or two) manually constructed regular expressions and then uses a parsing tree to group log-entries on the fly (*online parsing*) [37].

   IBM's version represents an updated and more feature-rich, robust adaptation of the original implementation, which in turn was provided by the LogPAI-team[3] in the context of their evaluation and benchmarks of log parsers [34, 8]. We personally contributed to IBM's implementation by providing it with a way to extract parameters from log messages more accurately by leveraging already specified regular expressions.

   Using Drain, we determine the log events contained in the logs of our datasets. During log parsing, we consider the log messages of different system components separately: Even if two components produce the same log message, these will be handled as two different log events.

2. We apply one of our two summarization tasks:

   summarization based on *common log events* We compute the log events that are present in all logs with the same root cause, select log-entries corresponding to these events and use them as a basis for reference summaries.

   summarization based on *non-normal log events* We compute the log events that are not present in any log during normal operation of a system (no failures), select log-entries corresponding to these events and use them as a basis for reference summaries.

3. We focus only on the log message contained in each log-entry and apply a preprocessing step laid out in subsection 3.4.2. This preprocessing is applied to both log messages we use as input for our models and to log messages forming reference summaries.

4. We split input documents and reference summaries into multiple parts, respecting the input size limitations of our models. As explained in subsection 3.4.3, we can do this without truncating whole documents or providing mismatched summaries.

5. We evaluate different summarization models on the new summarization tasks.

For better comparison with existing previous work, we also evaluate using the openly available manual summaries written by Meng *et al.* [6] in addition to our own tasks.

[2] The repository is available at: `https://github.com/IBM/Drain3`

[3] Located inside LogPAI's repository: `https://github.com/logpai/logparser/tree/e8d96cd4de1121c5d2b517982c6028cd06e643f1/logparser/Drain`

The next subsections first provide details on the models we use and the insights we aim to gain through experiments, then explain how we preprocess log messages and split documents into multiple parts.

3.4.1 Training and Evaluation of NLP models

As NLP models pre-trained on large text corpora have shown to build a baseline for language understanding of the English language [4, 3], we choose to employ sequence-to-sequence models pre-trained on NLP tasks. With English being known as a "lingua franca" of programming, the predominant language used in the log-data we study, we exclusively focus on models trained on English corpora.

For their good performance in other summarization domains and production-ready, open-access availability in the context of HuggingFace's *Transformers*-library [28], we choose BART [4] and PEGASUS [5] as bases for our models.
It should be noted that HuggingFace's implementations of models may vary from their original implementations. In our case, the differences do not lie with any implementation details described in the original papers.[4]

Because of the limited availability of reference summaries for the log summarization task, and to quantify the effect of pre-training and supervision regarding log-data, we evaluate models with differing levels of training on log-data:

***zero-shot learning* (ZSL)** The models are directly evaluated on the summarization task without further fine-tuning.

***few-shot learning* (FSL)** The models are fine-tuned on the summarization task and then evaluated.

***self-supervised pre-training & ZSL* (Pre+ZSL)** The models are pre-trained in a self-supervised manner on the log-data and then evaluated on the summarization task without further fine-tuning.

***self-supervised pre-training & FSL* (Pre+FSL)** The models pre-trained on the log-data are further fine-tuned on the supervised summarization task and then evaluated.

The distinction between *pre-training* and *fine-tuning* is one of convention, as both refer to training processes, however fine-tuning is tailored to a specific dataset or task. Given that we apply both pre-training and fine-tuning on the same kind of log-data, we feel the need to define how this terminology is used in the context of this work:

Pre-training refers to a training process, where the model inspects a given text in a self-supervised manner, learning to fill in information intentionally left out from it.

Fine-tuning on the other hand refers to a supervised training process, where the model is given a document as input and evaluated using a target text. In our case we refer to fine-tuning exclusively in the context of summarization, where the model is given a document to summarize and is evaluated using a reference summary.

[4] For PEGASUS, differences are documented in the following GitHub-issue: `https://github.com/huggingface/transformers/issues/6844` (last accessed on 22nd March 2022)

Exactly which models we train and at which levels of supervision is laid out in the two next segments.

Evaluation of fine-tuned summarization models from other domains for log summarization

We first intend to determine how similar the task of log summarization is compared to summarization in previously researched domains, respectively, if models can transfer their knowledge on summarization from other domains to log summarization. To this end, we examine the models laid out in Table 3.1 that have already been fine-tuned on different summarization datasets, and evaluate them in a zero-shot learning and few-shot learning setting. To better understand whether previous fine-tuning has any positive effects, we also need to consider how models not previously fine-tuned perform on our tasks. This is why we evaluate the large versions of the pre-trained baselines BART-Large (facebook/bart-large) and PEGASUS-Large (google/pegasus-large) in addition to the models laid out in Table 3.1.

If summarization on other domains is similar to log summarization, this would be beneficial to NLP models requiring large amounts of supervised training. One could train a model on an actively researched domain with higher availability of large summarization datasets and avoid overfitting models on the few existing log summaries.
Previous research has found that both model-architectures greatly benefit from learning even with few examples [39], we expect the model's performance to improve considerably after further fine-tuning.

XSum [35] and CNN/DailyMail [36] are popular examples of news datasets. These also represent the only datasets BART models were originally trained on for summarization. In general, XSum's summaries are shorter and more abstractive than CNN/DailyMail's, meaning they often include text sequences not present in the document to be summarized.

On the other hand, AESLC represents summarization of business emails, with summaries generated from subject lines. Here both documents and summaries are significantly shorter [40].

Finally, BigPatent is a dataset containing summaries of U.S. patent documents; summaries are more abstractive and often include recurring segments [41].

A simple comparison between these conventional summarization datasets and the log summarization datasets used by us is shown in Table 4.2 on page 38.

identifier	**HuggingFace's model**	**model described in**
BART-XSum	facebook/bart-large-xsum	[4]
BART-CNN	facebook/bart-large-cnn	[4]
PEGASUS-XSum	google/pegasus-xsum	[5]*
PEGASUS-CNN	google/pegasus-cnn_dailymail	[5]*
PEGASUS-AESLC	google/pegasus-aeslc	[5]*
PEGASUS-BigPatent	google/pegasus-big_patent	[5]*

* Models are based on the improved Mixed & Stochastic approach, described in the original paper as PEGASUS$_{\text{LARGE}}$-(mixed,stochastic), as stated here: `https://github.com/huggingface/transformers/issues/4918#issuecomment-673572058` (last accessed on 14th March 2022)

Table 3.1: Overview of models already fine-tuned for summarization we experiment with.

Effects of further pre-training on log-data

During pre-training, models are expected to predict text segments in the input data, which have been replaced ("masked") by a mask-token. The most basic form of this task where a single token is missing is known as *masked language modeling* (MLM) and was introduced for pre-training NLP models by Devlin *et al.* [3]. Since then, many alternative pre-training objectives have been proposed, building on this idea. Both models we use have been pre-trained using this basic idea, but in contrast to MLM, longer sequences of text can be missing; the exact method by which the masked text segments are chosen differ between BART and PEGASUS.

In the context of their empirical study Zhu *et al.* note that further pre-training is not always constructive towards improving a model's performance when fine-tuning for a specific task and domain later on. However, they do observe that further pre-training a model is especially effective when data used for fine-tuning is not available in large quantities [42].
Furthermore, Gururangan *et al.* find that further pre-training significantly improves performance when the domain a model was trained on differs from the domain a model is used in [43].

Given the difference between the model's training data (mostly web pages and news articles) and log-data, we see great importance in further pre-training the models and follow the self-supervised approaches chosen by the respective model.

As HuggingFace's *Transformers*-library [28] does not provide implementations for neither model's pre-training objectives, they have been implemented manually by us following the ideas laid out in the model's respective article and studying the code provided in the original projects.[5]

We follow BART's approach to pre-training using the *Text Infilling* method: Possibly empty token sequences with lengths drawn from a Poisson distribution ($\lambda = 3$) are removed and masked by inserting a mask-token [4]. Masked sequences can start at sub-tokens of words[6] and can span over multiple messages. The starting positions of the masked sequences are uniformly chosen.

Contrary to the original implementation, it is made sure that already inserted mask-tokens are not deleted when masked sequences overlap.[7] During pre-training model is then required to predict which (possibly empty) token sequences are missing in a text where a mask-token is present.

To quantify the performance boost to be gained from pre-training alone, we first evaluate the model in a ZSL setting with no further training and compare their performance after pre-training in a Pre+ZSL setting. Finally, we additionally fine-tune the pre-trained models on the actual summarization tasks (Pre+FSL) in order to investigate whether further pre-training is useful in the domain of log-data as compared to just fine-tuning the models.

[5] See `https://github.com/pytorch/fairseq/blob/fcca32258c8e8bcc9f9890bf4714fa2f96b6b3e1/fairseq/data/denoising_dataset.py` for BART's implementation

[6] The original paper does not mention whether word-boundaries are respected and the original implementation supports both variants. See also: `https://github.com/pytorch/fairseq/blob/fcca32258c8e8bcc9f9890bf4714fa2f96b6b3e1/fairseq/data/denoising_dataset.py#L104-L106`

[7] The original implementation may override existing masks as noted here `https://github.com/pytorch/fairseq/issues/3486#issuecomment-901689083` (last accessed on 30th March 2022)

After fine-tuning, at least BART models for summarization are not able to fill in masks anymore[8] and thus cannot directly achieve their pre-training objectives. We will not experiment with further pre-training models that are already fine-tuned for summarization. Since summarization models are not usually exposed to masks during fine-tuning, they are not encouraged to retain their ability to fill in masks; We would expect them to show worse pre-training performance than non fine-tuned models eitherway.

Thus, we cannot compare different variants of the same model, and decide to use the model BART-Base (facebook/bart-base) for this style of pre-training. Due to its smaller size, the training and evaluation are considerably faster.

3.4.2 Text Preprocessing

In the context of news summarization Kryściński *et al.* previously found that datasets contained noisy reference summaries, such as passages unrelated to the actual article, markup characters, or placeholders. Using simple regular expressions, they removed such noise from reference summaries [32].

Inspired by this, we also intend to simplify the text contained in log messages, in order to make the message more readable and more similar to actual text.

Logs often contain long *identifiers*, addresses of remote systems, **class names**, paths and other URIs, that may be problematic to tokenize in order to use them as inputs for sequence-to-sequence models. Critically, relative to their length, these sequences contain little information that NLP models can actually use. Additionally, log-entries originate from different components, making the formatting of log messages quite inconsistent when compared to regular text; Ranging from well-constructed sentences, to fragments with inconsistent use of lower- and uppercase, punctuation, and special characters. This is exemplified for the Hadoop log dataset in Table 3.2, which is later introduced in section 4.2.

Thus, we introduce a preprocessing/simplification step, so that models will not be expected to replicate many complex patterns present in log-data:

To simplify messages, we use their corresponding log templates and parameters determined during log parsing, process these parameters, and insert them back into the template to reconstruct simplified messages. We use several regular expressions to improve the detection of parameters and categorize parameters according to these during preprocessing.

Based on the idea of replacing long sequences that convey little useful information, we devise rules for different categories of parameters. If the parameter represents a(n)

filesystem path	we simplify it to the last word contained in the path,
URI	we simplify it to its URI-type, e.g. `http-URL` for a URI beginning with `http://`,
Java object path	we simplify it to the first word contained in the last path elements (usually the entire class-name),

[8] As noted here `https://huggingface.co/docs/transformers/v4.17.0/en/model_doc/bart#implementation-notes` (last accessed on 28th March 2022) and as can be verified by attempting to do so manually.

```
We launched 1 speculations.  Sleeping 15000 milliseconds.
```

```
Processing the event EventType:  CONTAINER_REMOTE_CLEANUP for container
container_1445062781478_0011_01_000002 taskAttempt attempt_1445062781478_0011_m_000000_0
```

```
Before Scheduling:  PendingReds:0 ScheduledMaps:0 ScheduledReds:0 AssignedMaps:9 AssignedReds:1
CompletedMaps:4 CompletedReds:0 ContAlloc:13 ContRel:0 HostLocal:12 RackLocal:0
```

```
Process Thread Dump:  Communication exception
12 active threads
Thread 21 (SpillThread):
State:  WAITING
Blocked count:  0
Waited count:  6
Waiting on java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject@2e498b1
Stack:
sun.misc.Unsafe.park(Native Method)
java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)
[...]
Thread 20 (org.apache.hadoop.hdfs.PeerCache@60ec1f20):
[...]
```

```
Error Recovery for block BP-1347369012-10.190.173.170-1444972147527:blk_1073742514_1708 in pipeline
10.190.173.170:50010, 10.86.164.9:50010:  bad datanode 10.86.164.9:50010
```

```
Releasing unassigned and invalid container Container:  [ContainerId:
container_1445062781478_0012_01_000012, NodeId:  MININT-75DGDAM1.fareast.corp.microsoft.com:51951,
NodeHttpAddress:  MININT-75DGDAM1.fareast.corp.microsoft.com:8042, Resource:  <memory:1024, vCores:1>,
Priority:  20, Token:  Token { kind:  ContainerToken, service:  10.86.165.66:51951 }, ].  RM may have
assignment issues
```

```
Processing split:  hdfs://msra-sa-41:9000/pageinput2.txt:0+134217728
```

```
Registering class org.apache.hadoop.mapreduce.v2.app.job.event.JobFinishEvent$Type for class
org.apache.hadoop.mapreduce.v2.app.MRAppMaster$JobFinishEventHandler
```

```
Extract jar:file:/D:/hadoop-2.6.0-localbox/share/hadoop/yarn/hadoop-yarn-common-2.6.0-SNAPSHOT.jar!/
webapps/mapreduce to C:\Users\msrabi\AppData\Local\Temp\2\Jetty_0_0_0_0_47462_mapreduce____j3iclo\webapp
```

Table 3.2: Examples of "noisy" log messages and inconsistent style across the Hadoop-dataset. *Identifiers*, addresses of remote systems, **class names**, paths and other URIs are highlighted in the above messages.

IP- or mad-address	we compute a corresponding short hash and replace it with a text like `remote host #<hash>`,
known identifier	we use a hash again, but indicate the type of object referenced like `<type>#<hash>`,
long list of numbers	we simplify it to the 3 numbers with highest occurrence,
hexadecimal number	we convert it into a decimal number,
long sequence of hex. characters	we shorten it,
number	we leave it as is,
otherwise	we simplify it to the longest word contained in the parameter as a fallback.

In addition to the rule-based message simplification laid out above, we also remove some complex patterns *before* log parsing: Patterns spanning multiple lines such as stack traces and information dumps represent information hard to parse and process. Some log messages contain additional information that may prefix any message, such as markers for debugging traces or additional meta-information such as timestamps. It is necessary to separate these from log messages before parsing them, at least with fixed-length templates such as those produced by Drain. The same log message will be assigned different events depending on whether additional information is present or not, inhibiting our summarization tasks from working as intended.

The results of our efforts are exemplified in Table 3.3. While our heuristical approach may destroy useful information in some cases and does not go as far enough to fix inconsistent styling in other cases, we believe our simplification strikes an acceptable balance and increases the readability overall.

`We launched 1 speculations. Sleeping 15000 milliseconds.`
`Processing the event EventType: CONTAINER for container container#17058 taskAttempt attempt#64013`
`Before Scheduling: PendingReds:0 ScheduledMaps:0 ScheduledReds:0 AssignedMaps:9 AssignedReds:1 CompletedMaps:4 CompletedReds:0 ContAlloc:13 ContRel:0 HostLocal:12 RackLocal:0`
`Process Thread Dump: Communication exception 12 active threads`
`Error Recovery for block block#68462 in pipeline remote host #28779, remote host #68312: bad datanode remote host #68312`
`Releasing unassigned and invalid container Container: [ContainerId: container#5224, NodeId: microsoft. NodeHttpAddress: microsoft. Resource: <memory:1024, vCores:1>, Priority: 20, Token: Token { kind: ContainerToken, service: remote host #80275 }, ]. RM may have assignment issues`
`Processing split: hdfs-URL:0+134217728`
`Registering class JobFinishEvent for class MRAppMaster`
`Extract jar:file:/jar-path!mapreduce-path to webapp-path`

Table 3.3: Simplified messages from the examples in Table 3.2.
The replacements of parameters previously highlighted in the examples are highlighted again, but there are also a few **paramters simplified using our fallback-rule**.

It is important to stress that our NLP models are only given the simplified messages as input. The reference summaries used in our summarization tasks are also composed of these simplified messages. Thus our approach does not evaluate how important the content of each log message itself is: For evaluation, it is not important which parts of the log message the model is expected to reproduce.
Nevertheless, we suspect that making log messages more readable while containing enough context for a model to be able to identify important log messages is beneficial to the model's summarization abilities.

As a final note, we believe that our preprocessing step could also be refined by making it mostly lossless to convert the simplified messages back to the original messages. For example, we could convert segments like `remote host #<hash>` back to IP-addresses using a unique mapping of the contained hash. While this could be beneficial to applications where the exact IP-address or identifiers are of high interest, we are not interested in preserving the exact messages as it is not essential to summarization, and will not further explore this idea going forward.

Feeding log messages to NLP models

NLP models are typically designed to process texts that are made up of multiple sentences and may not be able to handle some specifics of log-data well. We have already discarded any meta-information contained in log-entries and focused only on the log messages, but some challenges still remain in processing these messages.

Transformer-based NLP models typically use a tokenizer to convert texts into a sequence of tokens. Each token is assigned to a distinct vector, a so called *embedding*, that the model operates on. The first challenge comes with the tokenizer's ability to process previously unseen words, of which there may be many in log-data. For example, in the form of compound words.

In our experiments, the tokenizers used by both PEGASUS and BART were able to handle longer compound words composed of multiple English words: Mixed case (*camelCase* or *CapitalCamelCase*) compounds will be split into multiple lexical units. As an example, PEGASUS' tokenizer will split `NodeHttpAddress` into the 3 tokens `_Node`, `Http` and `Address`, denoting that `_Node` starts a new word, while the other tokens do not and are part of the previous word.

Therefore we decide against splitting such compounds manually in our preprocessing step. Compounds often refer to a singular entity or concept; splitting compounds into multiple lexical components will inevitably destroy this kind of information that may be useful for NLP models.

However, another problem is the one of separating multiple log messages syntactically in a model's input. In typical English texts, punctuation marks symbolize the semantic separation of text segments. These are absent in log-data; Here, linebreaks are usually used to separate log-entries and log messages may or may not make use of punctuation themselves.

Usually encoder-decoder models include special tokens to symbolize *beginning of sequence*s (`BOS`s) and *end of sequence*s (`EOS`s) [29] or sometimes a special *separator* (`SEP`) token [3]. But neither BART nor PEGASUS has been trained to use multiple `SEP`, `BOS`- or `EOS`-tokens to separate sentences in inputs: The primary purpose of `BOS`-/`EOS`-tokens is to give a model's decoder an input when predicting the first token (`BOS`-token) and the last token (`EOS`-token) [29].[9]

Devlin *et al.* did leverage the idea of `SEP`-tokens to separate *pairs* of sentences in the input for different kinds of tasks, where a model would need to consider two different text sequences (e.g. question answering or sentence entailment) [3]. BART later used their `BOS`-/`EOS`-tokens for the purpose of separation for these kinds of tasks [4].

So while the use of `BOS`-/`EOS`-tokens for separation of different text segments is not unheard of, neither BART nor PEGASUS have observed *several* such tokens in a singular input batch during pre-training. Sentences are separated simply by the tokens of punctuation symbols. PEGASUS' tokenizer *does* possess a special token for newlines [5], but this is done explicitly to be able to detect the separation of different paragraphs.[10]

The self-evident solution to syntactically separate log messages in input data is to present models with special separators during training. Unfortunately, since we want to employ and evaluate pre-trained models for log summarization, this is not an option for us.

[9] See also `https://huggingface.co/blog/encoder-decoder#background` (last accessed on 5th April 2022) for a more in-depth explanation.

[10] See implementation notes of the "Mixed & Stochastic" model here: `https://github.com/google-research/pegasus/tree/649a5978e45a078e1574ed01c92fc12d3aa05f7f#results-update`

Due to a lack of better alternatives, we use semicolons to separate log messages for BART models and either a semicolon or the newline-token to separate messages for PEGASUS models. This choice is backed up by a preliminary experiment we conducted in section 4.3.

Given that there does not seem to be a perfect solution to this problem, without previously training models to accept certain tokens as separators, we will abstain from evaluating a model's ability to insert line-breaks correctly between distinct log messages when evaluating its performance later on.

3.4.3 Respecting Input Size Limitations

One of the inevitable design limitations of NLP models based on the transformer architecture [27] is a fixed maximum input size. For both the BART and PEGASUS models we use, this limit lies at 1024 tokens. As can be seen in Table 4.2 on page 38 this is enough for many domains such as news articles, where the average word count (which is proportional to the token count) is not that high.

However, with our proposed summarization tasks, the maximum input limit is exceeded drastically on our datasets (the HADOOP and TELCOAPP datasets, more specifically). The usual approach also followed by BART and PEGASUS is to truncate documents at the model's maximum input size such that the remaining part is not considered for summarization. Fortunately, due to the nature of our summarization tasks, it is trivial to construct summaries for parts of a document; Hence we need not discard most of our input documents.

We first determine the relevant log events according to the respective summarization task described in section 3.2 for the entire sequence we summarize. Following this, we can divide the log-data into chunks that will fit within the given maximum input length, respecting the boundaries of individual log messages, and construct a summary for each chunk by selecting the log messages linked to those events. If a chunk has no relevant log messages and the summary is therefore empty, we ignore that chunk for evaluation and training. Crucially, the relevant log events are determined for the entire document beforehand, not each chunk.

As the tokenizer from PEGASUS uses a different vocabulary compared to BART's tokenizer, the boundaries where chunks are drawn differ slightly between models. This is problematic for direct comparison of the models on the summarization tasks.

To circumvent this, we can use one model's tokenizer to select chunks for both models: BART's tokenizer always needs a few more tokens to represent the same log-data in all our observed cases, so if the input fits into a BART model, it will also fit into a PEGASUS model. We would like to emphasize that *each model still uses its own tokenizer to process inputs*. BART's tokenizer is solely used to determine how to select chunks of input data that will fit into both models. While PEGASUS' maximum input limit is not always *fully* utilized, it allows us to compare both models' performances as they are evaluated on identical inputs and reference summaries.

However, we *do* truncate individual log messages longer than $\frac{1}{4}$ of the model's maximum input limit. Our reasoning is as follows: It is not meaningful to have the model summarize an individual log message; instead, we ensure that the model always receives some context by limiting the length of individual log messages. Furthermore, we would argue that long messages often convey less information relative to their length than multiple shorter log messages. This is because a log message usually only relates to a singular component of a system and denotes a singular event or system state. It is more important to consider multiple log messages than to fully consider every detail of a single message spanning multiple hundreds of tokens.

3.5 Perplexity and Cross-Entropy

When training a NLP model one needs a measure of the model's performance (*loss function*) in order to tweak model parameters with the goal of optimizing this measure. A widely-used choice is *cross-entropy* or indirectly *perplexity*; both BART and PEGASUS optimize cross-entropy during all training-phases [4, 5].

Cross-entropy understands a language model as a probability distribution $\mathbb{P}$ over a discrete set of tokens X forming the vocabulary of a language $\mathcal{L}$. Given a known context of n preceding tokens $t_0, \dots, t_{n-1} \in X$, the language model tries to approximate the language by providing a probability distribution $\mathbb{P}(t_n | t_0, \dots, t_{n-1})$ for the next token $t_n \in X$.

Given two probability distributions p, q their cross-entropy H is defined as

$$H(p, q) = -\sum p(x) \log_2 q(x) \tag{3.1}$$

according to Manning *et al.* [21]. By interpreting a text sample of k tokens $s_1, \dots, s_k \in X$ as a probability distribution (where the probability of s_i given $s_1, \dots, s_{i-1}$ is 1 and for all other tokens $t \neq s_i$ the probability is 0) the cross-entropy of a language model $\mathbb{P}$ on that sample can be defined as

$$\tilde{H}((s_1, \dots, s_k); \mathbb{P}) = -\sum_{i=1}^{k} \log_2 \mathbb{P}(s_i | s_1, \dots, s_{i-1}) \tag{3.2}$$

Normalizing this value by the length k of the text sample, results in a measure representing how "surprised" the language model is on average [21]. For practical applications, a lower cross-entropy tends to result in better performances but this is not necessarily the case: If the model simply grows confident for more obvious predictions (e.g. the letter at the beginning of a sentence is always capitalized) but fails to understand more sophisticated contexts, using cross-entropy as a loss function still rewards the model [21].

Perplexity PPL simply refers to the exponentiated normalized cross-entropy

$$\mathrm{PPL}(\mathbb{P}) = 2^{H_{\text{norm.}}(\mathbb{P})} \tag{3.3}$$

Intuitively, if a model shows a perplexity of n this can be understood as the model being as surprised on average as if it had to guess between n equally probable tokens at each step [21].

The concept of cross-entropy can also be extended to bidirectional language models (such as BART and PEGASUS), which can additionally consider a fixed-size context of m subsequent tokens $t_{n+1}, \dots, t_{n+m} \in X$. Here, the model combines the probability distributions for preceding and subsequent tokens in a joint-probability distribution [23, 24].
We use PyTorch's `CrossEntropyLoss`[11], which is modified to be able to ignore arbitrary tokens,

[11] `https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html` (last accessed on 10th March 2022)

making it suitable for evaluating bidirectional language models such as BART and PEGASUS. This is helpful to ignore mask tokens during pre-training and any padding during both training phases.

During both pre-training and fine-tuning we leverage the cross-entropy between the model output and the reference texts as a loss function to optimize.

Furthermore, we sometimes use perplexity to evaluate the performance. Since cross-entropy and, by extension, perplexity are dependent on the model's vocabulary, perplexity cannot be compared between models using different vocabularies and tokenizers. Thus, when comparing model performances, we may use perplexity exclusively to compare models of the same architecture.

3.6 Text Generation via Beam-Search

Graves introduced a fixed-width beam-search to neural text generation as a means to create a generalized model to produce an output sequence given another sequence as input, without specifying the length of outputs in advance [44]. As well as enabling arbitrary lengths of generated output text, the beam-search also allows the language model to consider the text generated by itself thus far at each prediction step.

Analogous to cross-entropy, beam-search understands a language model as a probability distribution $\mathbb{P}$ over a discrete set of tokens X forming the vocabulary of possible sequences. We make the simplification that input and output sequences use the same vocabulary, as is the case in our application. In a more generalized setting, beam-search is not limited to a singular vocabulary. Given a known context of n already generated tokens $t_0, \ldots, t_{n-1} \in X$ and an input sequence $i_0, \ldots, i_m \in X$, the language model provides a probability distribution $\mathbb{P}(t_n | t_0, \ldots, t_{n-1}; i_0, \ldots, i_m)$ for the next token $t_n \in X$.

During a beach-search of fixed width W, the most probable next tokens are determined using the language model and appended to the current predictions, similar to a greedy best-first search. Unlike the greedy search, the W most probable sequences, forming the set of *beams* B, are kept and updated:

1. For each beam $b \in B$ the W most probable next tokens are determined.
2. A new candidate is created for each of the W tokens by appending the token to the end of the beam b.
3. Only the W most probable candidates are kept and used as beams B for the next iteration step.

Thus a beam-search with width $W = 1$ is just a greedy best-first search, appending the most probable next token given the current prediction at each step. By increasing the width W of the beam-search, we can get sequences that are more likely overall but would not have been considered by a greedy search, as they were hidden behind the inclusion of a less likely token.

In the end, the beam $b \in B$ with highest score s is returned

$$s(b) = \frac{\log \mathbb{P}(|b|)}{|b|} \tag{3.4}$$

as described in [44].

Several extensions to the beam-search have been proposed to improve its performance for text generation, such as prohibiting the model from generating duplicated n-grams in order to increase quality of longer summaries [45], or introducing a length penalty parameter to gain better control of the length of the generated summary [46].

The implementation provided by HuggingFace's *Transformers*-library [28] we use, employs an exponential length penalty δ, such that:

$$s(b) = \frac{\log \mathbb{P}(|b|)}{|b|^{\delta}} \tag{3.5}$$

Thus $\delta > 1$ encourages the model to predict longer sequences, while $\delta < 1$ favors shorter outputs than the initially proposed method. Additionally a beam is deemed as complete, if the language model ever predicts an `EOS`-token as the next most probable token.[12]

Parameters controlling the length of generated texts directly influence the quality of generated texts as perceived by different metrics. As such it becomes desirable to find optimal values for such parameters.

In the context of machine learning, parameters like these which need to be specified by the user in order to maximize the utility of a learning algorithm, are known as *hyperparameters*. They are generally used to configure the algorithm and may have notable effects on a model's performance [47]. The process of finding hyperparameters that maximize a scoring-function is known as a *hyperparameter serach* [47]. Frameworks such as Optuna [48] can be used to automatically perform such searches.

During our experiments we use beam-search to generate text, and use the same number of beams as indicated by the original description of the models; 5 for all BART-models and 8 for all PEGASUS-models [4, 5]. We regularly employ small automatic hyperparameter searches using Optuna to find adequate values for the length penalty δ described above.

[12] See the implementation at: `https://github.com/huggingface/transformers/blob/3a71e94a929fae8c04bdad9129c461c920413a2d/src/transformers/generation_beam_search.py`

4 Evaluation

To begin with, we introduce the ROUGE metrics we use during evaluation and briefly explain how these values can be interpreted in section 4.1. Afterward, we present the three datasets used in this work, explain how we select training and test data, and compare them with summarization datasets from other domains in section 4.2.

Then we report on the concrete experiments conducted: First evaluating a preliminary experiment to determine suitable separators between log messages (section 4.3), second, an extensive evaluation of different summarization models (section 4.4), followed by an experiment to explore the effects of further pre-training on log-data (section 4.5). Finally, we present a direct comparison of the best-performing model with a previous log summarization framework (section 4.6).

At the end of this chapter, we discuss potential threats to validity we see with our work in section 4.7.

4.1 Using the ROUGE Metric for Evaluation

In the previous chapter we introduced several models and the tasks we evaluate them on, yet the question remains how to judge the performance of our models. As abstractive seq2seq models produce arbitrary text, one cannot directly determine a set of *true* and *false* predictions, to measure accuracy for instance.

Deciding whether a text is a good summary of another text is not a straightforward, objective question. Rather we have to rely on metrics such as the widely used collection of ROUGE-metrics [49], which has become a de facto standard for judging the performance of summarization models.[1]

Originally, *Recall-Oriented Understudy for Gisting Evaluation* (ROUGE) was introduced to provide a set of metrics to judge the quality of a summary automatically. This is achieved by

[1] See for example the benchmarks on `https://paperswithcode.com/task/text-summarization`.

comparing it to other reference summaries usually created by humans [49].

As such, it analyzes so-called n-grams between a *reference* text and a *candidate* text that is to be evaluated, and defines different metrics based on their matching n-grams. In the context of natural language processing pairs of words are usually referred to as bigrams [21]; consequently, n-grams are the generalization of this concept and simply refer to a sequence of n words or tokens.

We use a port of the original implementation provided by the Google Research Team.[2] Here, unigrams (n-grams with $n = 1$) are determined as continuous sequences of non-alphanumerical characters, which are then lowercased. n-grams are computed as sequences of these unigrams. Consequently, the metric analyzes the whole text at once and ignores sentence boundaries indicated by punctuation. Contrary to the original, Google's implementation does not support *stopword removal*, that is the removal of words that play an important role in the construction of English sentences, but are less likely to contribute important information [21]. Common examples for stop words are *the*, *from* or *could* [21].

In the following segments, we intend to explain the two families of ROUGE-metrics we use for evaluation in our experiments. Although ROUGE was designed to be able to compare a candidate summary to multiple reference summaries, we will only ever have a single reference summary to compare to, hence why we will not explain how to apply ROUGE with multiple references.

ROUGE-n Vital to the computation of ROUGE-n is the number of n-grams occurring both in the candidate and reference summary [49], which we will refer to as $\#\texttt{match}$. Given the sequence of n-grams in the candidate text C_n and reference text R_n, as well as a function $\#_X(t)$ counting the occurrences of an element t in a sequence X, we can determine $\#\texttt{match}$ to be

$$\#\texttt{match}(C_n, R_n) = \sum_{t \in C_n \cup R_n} \min(\#_{C_n}(t), \#_{R_n}(t)) \tag{4.1}$$

where $X \cup Y$ is the set-union of two sequences X and Y (X, Y are each transformed each into a set of their unique sequence elements, then the union of the resulting two sets is taken).

The ROUGE-n recall is then defined as

$$\text{ROUGE-}n_{\text{recall}} = \frac{\#\texttt{match}(C_n, R_n)}{|R_n|} \tag{4.2}$$

where $|X|$ refers to the length of a sequence X. Depending on the choice of n one gets different results; ROUGE-1 measures the overlap in words between reference and candidate summaries ($n = 1$), ROUGE-2 measures the overlap of word-pairs ($n = 2$), and so forth.

In [49], only the ROUGE-n recall metric is mentioned, but we are equally interested in the precision and F_1-measure of ROUGE-n, which were added in a later version of ROUGE.[3]

[2] The repository is available at: `https://github.com/google-research/google-research/tree/9f5bedb711b322125505336fac58ef6261911ca4/rouge`

[3] A copy of the updated implementation can be found at `https://github.com/andersjo/pyrouge/tree/3b6c415204dbc2c8360a01d92533441f4aae95eb/tools/ROUGE-1.5.5`; originally the script would be sent via email, as explained on its nowadays unreachable website `https://web.archive.org/web/20161205014058/http://www.berouge.com/Pages/default.aspx`.

The ROUGE-n precision is defined as

$$\text{ROUGE-}n_{\text{precision}} = \frac{\#\texttt{match}(C_n, R_n)}{|C_n|} \tag{4.3}$$

Suppose we understand the overlap in n-grams as an approximation of the information a reference and candidate summary have in common and declare only the information contained in a reference summary as "important". In that case, ROUGE-n recall tells us how much of the important information is contained in a candidate summary, while precision measures what ratio of a candidate summary is important.

Intuitively, a good summary usually contains a lot of the important information (*recall*) but is also to the point and does not contain much unnecessary information (*precision*). However, it would be beneficial to compare the recall and precision in a single, combined metric to get a final indicator of a summary's quality. To this end, we can employ the F_1-measure, which is the harmonic mean of recall and precision:

$$F_1 = 2 \cdot \frac{\text{recall} \cdot \text{precision}}{\text{recall} + \text{precision}} \tag{4.4}$$

Going forward, we will imply the use of the F_1-measure for ROUGE-n when we refer to *ROUGE-n scores* unless stated otherwise.
When reporting ROUGE-scores, we report them in percent, but omit the percent sign, e.g. a ROUGE-1 recall of 0.245 is reported as a ROUGE-1 recall-score of 24.5. This also applies to any ROUGE-L scores we mention, which we explain in the next segment.

ROUGE-L Essential to ROUGE-L is the concept of the *longest common subsequence* LCS. Given a sequence $X = (x_0, x_1, \ldots, x_n)$ we can write any arbitrary subsequence of X as $Z = (x_{i_0}, x_{i_1}, \ldots, x_{i_j})$ with $0 <= i_0 < i_1 < \ldots < i_j <= n$. Such a subsequence Z may contain less elements than X but preserves the order of elements.

The longest common subsequence $\texttt{LCS}(X, Y)$ of two sequences X, Y is the longest sequence Z that is both a subsequence of X and Y. Intuitively, if two sentences share a long common subsequence, they not only contain similar information but also follow related sentence structures [49]. ROUGE-L normalizes the LCS to the lengths of the texts that are compared.

Given the sequence of words (*unigrams*) in the candidate text C_1 and reference text R_1, as well as the function $\texttt{LCS}(X, Y)$ which determines the longest common subsequence of the sequences X and Y, *sentence-level* ROUGE-L recall and precision are defined as follows:

$$\text{ROUGE-LSent}_{\text{recall}} = \frac{\texttt{LCS}(C_1, R_1)}{|R_1|} \tag{4.5}$$

$$\text{ROUGE-LSent}_{\text{precision}} = \frac{\texttt{LCS}(C_1, R_1)}{|C_1|} \tag{4.6}$$

ROUGE-L makes the distinction between sentences and summaries and uses a different method to determine recall and precision at *summary-level* with the help of user-provided sentence boundaries. At the summary-level so-called *union LCS* is used [49]: The LCS is calculated between a sentence in a reference summary and each sentence of the candidate summary, and the elements of all individual subsequences are merged to obtain the largest sequence, that is still a subsequence of the reference summary.

Thus the information contained in a single reference sentence may be spread across the whole candidate summary without impacting the overall summary-level score. Summary-level ROUGE-L allows the candidate summary to be structured differently from the reference and is indifferent to the number of sentences a candidate summary has.

[49] initially proposes a weighted F-measure to combine ROUGE-L recall and precision, which is a generalization of the F_1-measure, but the most recent official implementation uses a F_1-measure by default.
As such we also use the F_1-measure implicitly, when we refer to *ROUGE-L scores* unless stated otherwise.

4.2 Datasets

In principle, any dataset containing logs that adhere to the characteristics laid out in section 3.2 could be used. The key property for summarization based on *common log events* to be applicable, is the existance of multiple logs which can be grouped by root cause. That way it is possible to extract log-entries corresponding to common log events, which are the base for any reference summaries.
Meanwhile, summarization based on *non-normal log events* can in principle be applied, if logs have been labeled as abnormal or not.

In the following paragraphs we present the datasets we use individually.

TelcoApp A global telecommunication and cloud provider has granted us access to a collection of logs that we will refer to as the TelcoApp-dataset. The logs arise from application crash reports when using enterprise software in a small distributed system, producing entries of heterogeneous formatting, content, and purpose. System experts have manually conducted a root cause analysis and grouped the logs by root cause. For each root cause, there are multiple scenarios from different system failures. Therefore this dataset is particularly suitable for the summarization based on *common log events*.

Since logs in the TelcoApp-dataset are very fine-grained, containing tens of thousands of log-entries for each minute of runtime, the relevant portions of the logs have to be determined. To this end, we are also provided with the timestamp of the failure, but its temporal resolution is too low, still potentially leaving us with thousands of log-entries.

To further delimit the relevant parts of the logs, we inspect the common log events in a time span around the failure. For some root causes, we manually determined anomalous log events for which we suspect they represent error events leading up to the failure. There may be multiple such events per log file.
Finally, we extract all entries around these log events using a manually defined threshold and use these log messages as the baseline for our summarization dataset. All in all, we gather log-data from two related root causes.

HADOOP First described by Lin *et al.*, the HADOOP-dataset contains logs produced by the Hadoop platform[4] when running two different applications (PageRank, WordCount) on a cluster spanning multiple computers. During the execution of each application, failures are provoked manually (machine shutdown, network disconnection, full disk), thus creating multiple scenarios with the same failures [50]. There are also several logs available from running the two applications during normal execution, in the absence of any failures.
This makes the HADOOP-dataset particularly suitable for our summarization task based on *non-normal log events*.

Interestingly, summarization based on *common log events* is not applicable here, even though each log is categorized by failure type: The common log events between each failure type are almost the same; there are only a handful of log events common between all logs where a network disconnection occurred, which do not also occur when the disk is full. Therefore the common log events do not represent events specific to that failure but rather to any execution environment; the common log events are quite dull and represent startup operations, progress indicators and other generic activity. However basing the summarization on non-normal log events, we can get a concise summary of the failure from each log.

The dataset also features a multitude of short logs from other nodes running in parallel with the node that experienced the failure; Often a node simply starts up, performs its task successfully and shuts down again. In these cases no events of interest occur, so we would end up with many empty summaries. To avoid this situation we choose to exclude any logs with fewer than 200 log-entries from the summarization task.

This method of manually triggering failures used in [50] may be well suited to further construct datasets for log summarization. However going forward we will not explore the creation of more log summarization datasets and instead employ existing datasets for training and evaluation. As such we will leverage the publicly accessible HADOOP-dataset provided in the context of Loghub [51] available at [52].

LOGSUMMARY Unlike the previous two datasets, the LOGSUMMARY-dataset is not constructed by us using one of our summarization tasks but instead contains manually written summaries by Meng *et al.* using various datasets from [52], originating from various distributed systems, supercomputers, and standalone applications [51].

They summarized 601 examples of 20 consecutive log messages selected uniformly from 6 datasets (not including HADOOP) [6]. 2 of these datasets were not reported upon in the original paper, but the summaries are available in their repository.[5] All in all, we call this collection of documents and summaries from all 6 (sub-)datasets *the* LOGSUMMARY-dataset.

Contrary to the previous two datasets we examine, summaries are not based on the presence of failures, and input data may represent normal system activity. On TELCOAPP and HADOOP we would not produce a summary in such cases. Here we are given summaries of the most relevant contents, even for log-data that can be expected to occur during normal operation. This makes the task of summarization on this dataset different from the failure-centric summarization on

[4] see `https://hadoop.apache.org/`

[5] `https://github.com/WeibinMeng/LogSummary/tree/469b1e25f4f759c0ace1c91e2513a4be08831ea7/data/summary/logs`

the other two datasets.

Although the summaries are human-written, they still are purely extractive: Segments deemed most important by the authors are copied from the log entries and are collected into each summary.

Compared to our other log datasets, LogSummary's documents and summaries are quite short as can be seen in Table 4.2, and summaries contain only parts of a log message as opposed to being a sequence of entire log messages.

4.2.1 Selection of Training and Evaluation Data

As pointed out by Manning *et al.*, evaluating a model on the same data as the one it was trained on is bad practice: The evaluation is only fair, if the model is asked to make predictions using data it has not seen before. Otherwise, a model can simply reconstruct the outputs it has observed during training, favoring *overfitted* models that did not actually learn to perform the desired task in a more general setting [21].

It is common practice to select a only small percentage of a dataset as test data to evaluate on [21]. This would result in too few test examples to draw empirically sound conclusions because our datasets do not contain many examples. Furthermore, summaries in our datasets are not very varied, which is to be expected considering one of our summarization tasks is based on log events *common* between all logs of the same root cause.

Instead, we decide to select a few representative examples for each dataset to train on and evaluate on the overwhelming remainder:

LogSummary We randomly select 5 examples from each sub-dataset for a total of 30 training examples and 571 examples for evaluation.

Hadoop We manually choose representative examples from each application and failure type; Per application 5 for the *disk full*-failure and 10 for the *machine shutdown* and *network disconnection*-failures resulting in 50 training examples and 1055 examples for evaluation.

TelcoApp We choose all chunks belonging to one of two log files, which leads to 16 training examples and 89 examples for evaluation.

As a practical use case for manual selection of training examples like on Hadoop and TelcoApp, we envision the following: An operator has identified log-data corresponding to an error or set of errors which may occur repeatedly during system operation, and would like to speed up any future analyses conducted on such failures or similar ones. Thus he trains an *expert model* which is able to summarize such log-data but may fail on more distinct logs.

To judge the representativeness and diversity of our training and test sets, we calculate the overlap in the vocabulary of words used in the summaries between the training and test sets, shown in Table 4.1. For comparison, we also include this ratio for the commonly used training and test splits on the XSum news article summarization dataset. As was to be expected, the overlap on the TelcoApp-dataset is large, as we use the common log events between the logs as summaries. On the other two datasets, our training examples contain a large part of the words used in test data, which speak for their representativeness. Nonetheless, there are still some examples with novel content keeping the comparison fair, especially in LogSummary, which shows similar relative overlap to the one observed on XSum.

LogSummary	**Hadoop**	**TelcoApp**	**XSum**
$\frac{94}{321} \approx 29.284\%$	$\frac{171}{260} \approx 65.769\%$	$\frac{584}{600} \approx 97.333\%$	$\frac{18562}{65619} \approx 28.288\%$

Table 4.1: Ratio of words common between summaries in the respective training and test sets.

During our experiments, we sometimes use a *validation set* to run trials, allowing us to judge the performance of models without evaluating on the whole test set. This validation set is a small subset of the actual test data, where we again manually selected examples that seem diverse but representative for the test data. Importantly, the performance of models on these validation sets should and will only be used to estimate the effects of different training parameters.

4.2.2 Comparison of Log Datasets to other Summarization Domains

Another difference is that contrary to all other summarization datasets listed in this section, our log-summarization datasets are *purely* extractive, meaning that they only contain text segments that are also present in the input document.

A “perfect” model could theoretically achieve a score of 100 across all ROUGE-metrics. This is not the case with other summarization domains, even the more extractive datasets such as CNN/DailyMail, where such an *extractive oracle* would only be able to achieve an average ROUGE-2 F_1-score of 83.19 [32]. It is important to keep this in mind when contextualizing the performance on our log-summarization datasets.

We provide an overview of our datasets and other summarization datasets in Table 4.2. These characteristics were calculated by us, but they roughly match the characteristics reported by the authors of the respective dataset, if mentioned at all. As described in subsection 3.4.3, the documents and summaries for Hadoop and TelcoApp are split into multiple parts to respect the input size limitations of our models. Thus we report these properties on both the whole documents and partitioned documents. Since we discarded any empty summaries, the count of unique tokens may differ. Finally, for LogSummary we decided to also display the information for each subdataset separately.

Inspired by [43], we additionally analyze the overlap in vocabulary between log data and text corpora from other domains in an attempt to quantify how they differ. Specifically, we analyze the datasets that were used to train the summarization models from Table 3.1 and our log summarization datasets. To obtain a larger sample of log-data, we analyze all log files for TelcoApp and Hadoop, not only the parts where a summary can be generated for.

We tokenize all documents by collecting all alphanumerical sequences but also by splitting mixed case (*camelCase* or *CapitalCamelCase*) tokens. These occur often in log data but are actually made up from multiple English words (e.g. `HTTPJobFinishEventHandler` becomes `http job finish event handler`). For the log datasets we use the simplified messages (see subsection 3.4.2). We collect the tokens, remove words that commonly occur in English texts (so called *stop words*) or occur less than 3 times overall and rank the remaining tokens by frequency. Figure 4.1 shows the ratio of the top 500 most frequent tokens shared between

Dataset	number of documents	number of unique tokens	avg. words per document	avg. words per summary
CNN/DailyMail [36]	≈ 312k	≈ 481k	786.6	55.1
XSum [35]	≈ 226k	≈ 271k	430.2	23.2
AESLC [40]	≈ 18k	≈ 48k	136.4	4.4
BigPatent [41]	≈ 1341k	≈ 3527k	3607.9	116.6
HADOOP (ours)	57	471	24616.3	12762.5
partitioned HADOOP (ours)	1105	463	546.0	440.5
TELCOAPP (ours)	10	1632	5590.2	2109.8
partitioned TELCOAPP (ours)	105	1553	456.2	190.0
LOGSUMMARY overall [6]	601	651	190.4	8.7
BGL [6]	100	277	276.3	7.4
HDFS [6]	100	46	161.9	5.9
HPC [6]	100	88	103.1	5.9
Proxifier [6]	100	33	233	10.2
Spark [6]	100	138	205.6	11.6
Zookeeper [6]	101	203	162.4	11

Table 4.2: Dataset sizes and characteristics for log summarization datasets and summarization datasets from other domains.

datasets. Comparing the top 10k most frequent tokens like in [43] would not lead to a fair comparison in our case, as some of our log datasets contain far fewer unique tokens, see LOGSUMMARY in Table 4.2.

CNN/DailyMail and XSum share the highest amount of frequent tokens, which is not unexpected as they are both datasets for news summarization. AESLC shares a considerable amount of frequent tokens with those domains, but otherwise the overlap between datasets is quite limited.

In general, the vocabulary of log datasets differs from that of other domains. Log datasets show highest similarities between themselves, but are quite diverse even among themselves. A model performing well on one of these datasets might therefore not perform well on the other log datasets. Interestingly, the vocabulary of BigPatent shows moderate similarity with TELCOAPP and LOGSUMMARY, which we find promising in respect to the transferability of log summarization.

4.3 Investigating the Effects of Separators on Output Quality

In subsection 3.4.2 on page 25 we highlighted some challenges that come with separating log messages from one another in the models' input; Log messages often do not form complete sentences and thus punctuation cannot reliably be used for the purpose of separating messages.

Without such a separator it will be impossible for us to tell where a model intends to end a log message and begin the next one, as well as it being difficult for models to distinguish the semantic content of individual log messages.

In this section, we aim to find a sequence of tokens that can adequately serve as separators for each model.

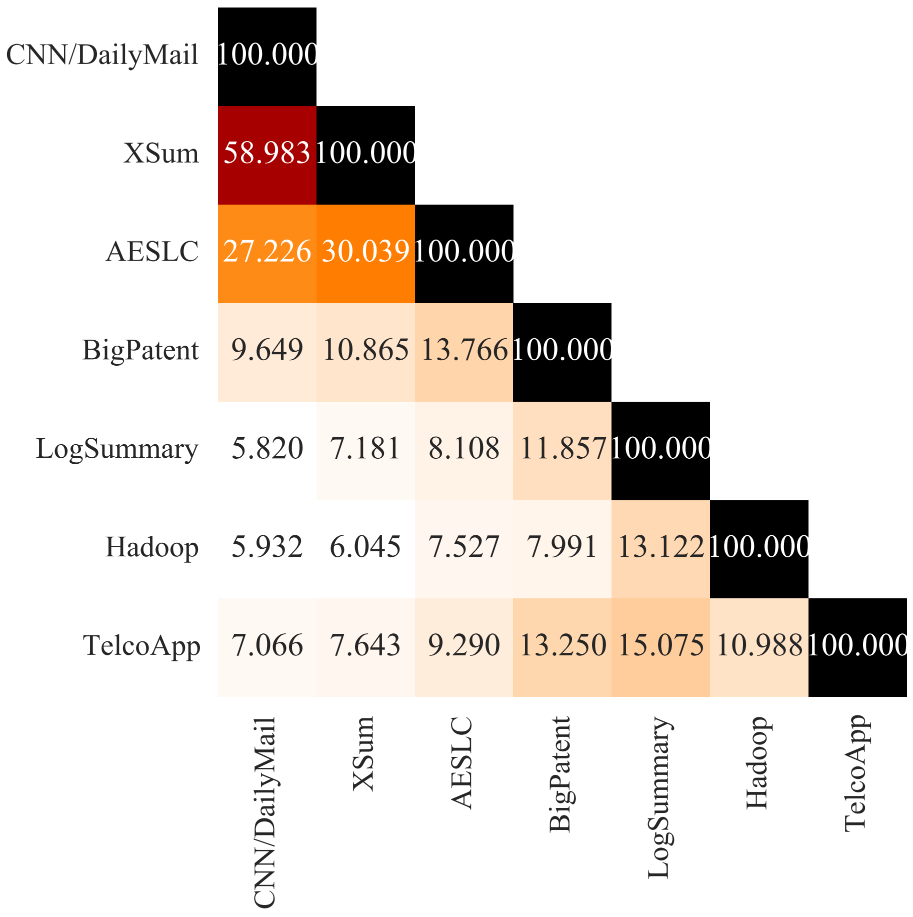


Figure 4.1: Top 500 most frequent tokens (excluding stop words and infrequent tokens) shared among datasets from different domains.

4.3.1 Experimental setup

Ideally, the separators we consider also have the function of separating text sequences with different semantic content in the natural language corpora the models have been trained on. We hypothesize that a model is more likely to understand that the additional tokens are used to syntactically separate log messages which are text segments of different semantic contents.

We consider the following options:

`BOS`-/`EOS` tokens These tokens are used to signify the beginning and end of the model's input. When fine-tuned for certain sentence-pair classification tasks Lewis *et al.* use their `EOS`-token as separator [4]. PEGASUS has a designated `EOS`-token only.

This approach is equivalent to passing each log messages separately to a model's tokenizer and then manually concatenating the results.

semicolons Semicolons may be used in written text; usually they are used to separate two related but independent text segments.[6] A semicolon used this way is typically separated by a

[6] See `https://www.grammarly.com/blog/semicolon/` (last accessed on 13th April 2022) for exemplified uses in the English language.

space from the next word. BART's tokenizer makes the distinction between a semicolon with a subsequent space or without, while PEGASUS' does not.

newlines PEGASUS is able to encode newlines in its input, which separate paragraphs in the pre-training data. The special newline-token also has the benefit of not being contained in any log messages, as newlines typically separate different log-entries. Thus we can guarantee that newlines are only used to separate different log messages, while the model should still be familiar with the semantic meaning of a newline.

For better comparison we also include a space as a baseline to compare model performances for when no special separator is employed.

We use BART-Large and PEGASUS-Large for this preliminary experiment, which have not been fine-tuned to any specific task, and train the models with the same setup as later in the FSL setting later described in section 4.4, apart from the separator used to join log messages in the model inputs and reference summaries. To be able to anticipate the effects of different separators on performance directly, we decide to evaluate the models' text generation capabilities and against investigating perplexity. Hence we perform a rough hyperparameter search to find "good-enough" parameters for the beam-search used to generate text.

4.3.2 Results

As the metric scores from this experiment are not meant to be comparable with the results from other experiments in this work, but rather to evaluate different separators for each model individually, we feel free to report the models' performance on a smaller, manually chosen validation set for each dataset. (36, 25 and 11 examples for LogSummary, Hadoop and TelcoApp respectively.) Additionally we only report mean F_1-scores for ROUGE-1, ROUGE-2 and sentence-level ROUGE-L since we are primarily focused on observing any relative improvements, not conducting a comprehensive evaluation of the models.

We chose sentence-level ROUGE-L rather than summary-level, as we are interested in the models ability to present the information in the same order as in the reference, which would be ignored by summary-level ROUGE-L.

The results for BART are reported in Table 4.3, while the results for PEGASUS are shown in Table 4.4. The best results are highlighted.

	ROUGE-1 / ROUGE-2 / ROUGE-LSent		
	LogSummary	**Hadoop**	**TelcoApp**
space (no syntactic separation)	71.6/63.2/68.2	35.5/31.6/34.1	**43.8/37.1/38.7**
`BOS`-/`EOS`-token pair	55.9/45.0/53.3	33.7/28.0/31.0	35.1/27.5/27.6
semicolon with space	67.2/58.1/59.4	36.5/30.1/34.8	37.2/29.2/29.8
semicolon without space	**72.4/65.7/70.4**	**47.6/40.2/45.2**	**43.8**/36.1/38.1

Table 4.3: Mean ROUGE-scores using BART-Large with different separators across all validation sets.

We also report the ratio of log messages ending in a punctuation mark (one of `.?!`) in Table 4.5, and the ratio of log messages starting with a capital letter in Table 4.6.

	ROUGE-1 / ROUGE-2 / ROUGE-LSent		
	LogSummary	Hadoop	TelcoApp
space (no syntactic separation)	69.4/60.7/65.4	31.6/28.7/31.5	26.5/20.4/24.2
EOS-token	51.8/42.3/51.8	21.9/18.9/21.8	7.6/ 5.4/ 7.1
semicolon	**77.0**/68.1/**72.7**	**33.0/30.0/32.7**	23.8/19.2/23.0
newline	76.9/**68.8**/71.8	23.6/20.2/23.4	**29.1/25.1/28.6**

Table 4.4: Mean ROUGE-scores using PEGASUS-Large with different separators across all validation sets.

LogSummary	Hadoop	TelcoApp
$\frac{311}{12020} \approx 2.587\%$	$\frac{233}{25931} \approx 0.899\%$	$\frac{592}{6079} \approx 9.738\%$

Table 4.5: Ratio of log messages ending in a punctuation mark across all datasets.

LogSummary	Hadoop	TelcoApp
$\frac{5479}{12020} \approx 45.582\%$	$\frac{23550}{25931} \approx 90.818\%$	$\frac{1970}{6079} \approx 32.407\%$

Table 4.6: Ratio of log messages starting with a capitalized token across all datasets.

4.3.3 Discussion

Generally, BOS-/EOS-tokens perform poorly as separators, in all cases performing worse than providing no syntactical separation of log messages. The poor performance can at least in part be explained by the fact that EOS-tokens signify to the beam-search that a beam is finished. When manually inspecting the generated summaries, we notice they are particularly short, leading to low recall, resulting in a low overall score.

In most cases, providing some form of separators significantly improved performance compared to using no explicit syntactical separation, confirming our assumption that it is important for models to be able to differentiate between different log messages. In settings where spaces performed comparatively well, we suspect that models may be able to use existing patterns concerning punctuation or capitalization to distinct between log messages.

PEGASUS We observe that PEGASUS-Large clearly benefits from the use of explicitly inserted separators on every dataset. This can be seen well on the LogSummary-dataset, where either option resulted in a minimum of a 9.79% increase in model performance after training (among all observed metrics) compared to separating subsequent log messages by a space.

Interestingly, not providing additional syntactical separation by using spaces as separators performs well with Hadoop. We believe this can in part be explained by the consistent use of capitalization in this dataset, as signified by the high rate of capital letters at the beginning of log messages. Our hypothesis is that PEGASUS-Large picked up on this pattern and it is sufficient for it to identify semantically different text segments.

On the other hand, the low amount of punctuation at the end of log messages seen in Hadoop may be the reason why using a punctuation-symbol such as semicolons performs better than using newlines in this specific case, whereas introducing extra punctuation performs worse in cases where punctuation is already present, such as TelcoApp.

BART With BART-Large, the benefits of introducing explicit separators seem less pronounced considering that spaces almost as well as the best choice for LogSummary and the best for TelcoApp.

However, we do also see a minimum of 27.22% performance increase for all considered metrics on the Hadoop-dataset when using semicolons without additional spaces compared to using spaces for separation.
This shows to us that BART-Large also benefits from being able to differentiate between independent log messages, at least in cases where there is a lack of punctuation at the end of messages.

Oddly enough, inserting a semicolon followed by a space between each log message, as one would do to separate independent text segments in a sentence, performs poorly. We are unsure why the model prefers semicolons without subsequent spaces, although we will highlight that the semicolon is represented by different tokens with different embeddings depending on whether a space is inserted or not. This means that the model did not necessarily learn that these two tokens have the same meaning for humans, instead there may be some biases present in the pre-training data that make the semicolon without a space a better syntactic separator of semantically independent text segments.

Conclusion It is impracticable to evaluate all possible tokens as separators. Hence, there may be some other sensible choices we did not consider that are better suited for use as separators. Furthermore, models previously fine-tuned may prefer different separators depending on the patterns seen in their domain and dataset. Nevertheless, performing such a preliminary study for each fine-tuned summarization model from Table 3.1 individually would be an impractical effort; instead we rely on the assumption that previously fine-tuned models inherit some knowledge on separators from their pre-trained bases.

Our goal was not to find optimal separators for our models, but to find ones that produce adequate results. Ideally, we would like to choose a single option per model going forward and use it consistently across all datasets. It would allow for models to be used in more universal settings.

We make an exception for this rule with PEGASUS-models: We choose newlines as separators for LogSummary and TelcoApp-datasets, but use semicolons on the Hadoop-dataset because we see a significant 39.74% increase among all observed metrics compared to newlines.

For BART-models we decide on semicolons as separators (without inserting a space after each semicolon) for all datasets, even if slightly outperformed by using regular spaces on TelcoApp.

As not introducing additional syntactic separation makes it harder to tell apart independent text segments in the summary and therefore decreases readability, we believe using spaces as separators is less helpful for any human operators using the model as it, even if the performance is comparatively good when considering ROUGE metrics.

4.4 Evaluation of Summarization Models from other Domains

As motivated in subsection 3.4.1, we aim to determine if models fine-tuned for summarization on other previously researched domains are able to transfer their knowledge on summarization to log-data.

For this, we evaluate all models laid out in Table 3.1 on page 20 in a ZSL and a FSL setting. Additionally, we include the baseline models BART-LARGE and PEGASUS-LARGE which have not been fine-tuned on summarization before. This way, we can see if previous fine-tuning for summarization makes a noticeable difference in performance, or if it suffices to fine-tune on the limited data available.

Under ideal circumstances we could simply let each of the models generate summaries on all our datasets. Using the ROUGE-metrics we could then quantify the performance of the models and conclude if any singular model outperforms the others.

During preliminary experiments, however, we find that the length of generated text heavily depends on the parameter chosen for the beam-search, such as constraints on the minimum and maximum amount of tokens generated and the *length penalty* in particular. The length of the summary directly influences its quality according to the ROUGE-metrics; As such we would need to find optimal values for such parameters influencing the generation of text in order to keep a fair comparison between models. Unfortunately, each model variant appears to have their own optima for such generation-parameters on every dataset.

Because finding optimal values for these parameters for each model is error-prone and cumbersome, we instead choose to initially focus on the perplexity for comparing different models, then compare generated summaries for selected models only.

Sometimes we employ boxplots to visualize our results; these are the classic "Tukey" boxplots [53], with a red line indicating the median of the distribution. Similarly, we use violin plots that visualize the data's distribution density, using the same indication of the median. Both types of plots and their origins are explained in [53].

We include intermediate results and frequently draw similar conclusions from different perspectives. We believe this to be necessary to justify the decisions we take in presenting the final results, but as a consequence, this section may be a bit long-winded.
As such we summarize any important intermediate conclusions during the final discussion of this experiment in subsection 4.4.5.

4.4.1 Experimental Setup

During the experiments in the next segments, we regularly evaluate models in two different settings with differing levels of supervision; With further fine-tuning on the training dataset or without:

***zero-shot learning* (ZSL) setting** As described in subsection 4.2.1 we defined a set of training examples for each dataset and use the rest for evaluation.

A zero-shot learning setting means that the models have previously seen zero training examples on the concrete task/dataset used; as such we simply run the models on the test datasets and evaluate their outputs, either through perplexity or through examining their summaries generated via beam-search.

Following Lewis *et al.* and Zhang *et al.* we apply a label smoothing factor of 0.1 for computing the cross-entropy loss [4, 5], which impacts the perplexity scores.

***few-shot learning* (FSL) setting** In the FSL setting we first train the models on a small training set, minimizing the cross-entropy between model predictions and reference summaries, and then evaluate it on the same test set as described above.

We use an effective batch-size of 8 and using a constant learning rate of $5 \cdot 10^{-5}$ we train for different amount of steps on each dataset:

LogSummary	$20 \cdot \lceil \frac{30}{8} \rceil = 80$
Hadoop	$15 \cdot \lceil \frac{50}{8} \rceil = 105$
TelcoApp	$25 \cdot \lceil \frac{16}{8} \rceil = 50$

where the number of steps is determined as

$$\text{\#steps} = \text{\#epochs} \cdot \left\lceil \frac{\text{\#training-examples}}{\text{batch-size}} \right\rceil \tag{4.7}$$

An epoch is understood as an iteration over the entire training dataset. We chose an adequate number of epochs using our experiences from preliminary trials, where we observed roughly at which point the generation performance starts to stagnate or even deteriorate on a validation set and chose an amount of epochs that works for *both* model architectures.

4.4.2 Analysis of Model Perplexities

We first evaluate the perplexity of the models only. Again, perplexities are not comparable between models with different vocabularies for the reasons laid out in section 3.5. However, we can compare models of the same architecture, e.g. BART-Large with BART-CNN as they use the same vocabulary and tokenizer.

The results for BART are shown in Table 4.7 for the zero-shot learning setting and in Table 4.8 for the few-shot learning setting.

	LogSummary	**Hadoop**	**TelcoApp**
BART-Large	869.696	19.356	31.837
BART-CNN	**89.208**	**5.447**	**7.373**
BART-XSum	412.338	10.316	11.844

Table 4.7: Perplexities of BART variants in a zero-shot learning setting.

For PEGASUS we report the zero-shot learning results in Table 4.9 and in Table 4.10 for the few-shot learning setting.

	LogSummary	Hadoop	TelcoApp
BART-Large	**15.547**	4.581	7.151
BART-CNN	17.236	**4.369**	**6.009**
BART-XSum	15.820	4.518	7.032

Table 4.8: Perplexities of BART variants in a few-shot learning setting.

	LogSummary	Hadoop	TelcoApp
PEGASUS-Large	1684.929	**100.972**	**168.063**
PEGASUS-CNN	563.526	136.606	173.432
PEGASUS-XSum	**495.076**	525.619	459.367
PEGASUS-AESLC	747.233	703.096	744.069
PEGASUS-BigPatent	1399.012	705.201	330.518

Table 4.9: Perplexities of PEGASUS variants in a zero-shot learning setting.

	LogSummary	Hadoop	TelcoApp
PEGASUS-Large	12.957	6.115	32.345
PEGASUS-CNN	**12.244**	**5.559**	**27.920**
PEGASUS-XSum	13.144	9.560	78.193
PEGASUS-AESLC	19.775	8.943	68.682
PEGASUS-BigPatent	60.724	29.735	108.614

Table 4.10: Perplexities of PEGASUS variants in a few-shot learning setting.

Generally, we notice that previous fine-tuning on summarization data does help when in a ZSL setting, but the differences between models shrink significantly after training on the few examples provided.

This is especially true for BART models, for which the largest relative deviation between the best and lowest scoring models after fine-tuning is approximately 11% on the LogSummary, 5% on the Hadoop and 19% on the TelcoApp-dataset. (Which is low compared to for example $\approx 255\%$ on the Hadoop dataset for ZSL.) While judging the perplexity alone is not enough to infer a models performance to generate good summaries, this is a strong indicator that previous training for summarization does not contribute considerably to a models ability to predict summaries on our specific log-data.

This also holds for PEGASUS, where PEGASUS-Large always shows the second-best performance, with an approximate relative derivation of 6%, 10% and 16% from PEGASUS-CNN on LogSummary, Hadoop and TelcoApp respectively.

However, we can see that using previously fine-tuned models can actually perform substantially worse than the baseline PEGASUS-Large. Especially the variant fine-tuned for summarization on BigPatent failed to achieve similar perplexities as the other variants, despite the higher overlap of vocabulary we found in subsection 4.2.2. As summaries in BigPatent tend to be abstractive [41], we suspect that the model is unable to quickly adapt to a more extractive task such as the ones presented by us. PEGASUS-BigPatent may improve further with more fine-tuning steps, but even so, PEGASUS-Large managed to adapt much quicker: Initially staring with a higher perplexity in the ZSL setting, but achieving a significantly better FSL result after the same amount of training-steps on LogSummary. This reinforces our assessment that further fine-tuning models previously trained on summarization tasks is not worth it for dissimilar domains such as log-data and that fine-tuning pre-trained models even with only a few examples is sufficient.

Still, for both models the best performing FSL model is most often the variant trained on the CNN/DailyMail dataset, albeit closely followed by the baselines BART-LARGE and PEGASUS-LARGE.

Extractive models tend to perform better on the CNN/DailyMail news-dataset than abstractive ones [32], which may explain why models fine-tuned on this dataset outperform those trained on other domains. After all, the summaries in all our log summarization datasets are purely extractive, hence do not include any novel words not seen in the input data or any paraphrasation of content. The slight difference in performance compared to the baselines not tuned for summarization might indicate that there is still some insight to gain concerning which text segments are relevant. We believe this is why we are seeing that BART-CNN performs comparatively well in a ZSL setting.

As noted before, lower perplexity usually correlates with better performance in applications, but if the model simply gets more confident in more obvious cases, this is not always the case. Hence we cannot be sure that the models showing the best perplexities are also the ones performing the best for summarization of our log-data. To select the best models, we conduct a few further trials before decide on the final models to evaluate.

4.4.3 Further Trials to determine the best Models

PEGASUS Due to the summary length heavily impacting the recall of the ROUGE metric, and the complications arising from optimizing the summary length using the parameters of the beam-search, we instead decide to compare only the ROUGE precision while setting the length of generated summaries to a small, fixed amount. For the PEGASUS models we decide on the following lengths:

- ◇ We let the models generate exactly 20 tokens on the TELCOAPP and HADOOP datasets.
- ◇ On the LOGSUMMARY dataset, we instruct the model to generate only 8 tokens, as summaries are much shorter in this dataset. 8 tokens corresponds to the median length of summaries using PEGASUS' tokenizer.

By comparing the precision only, we can judge what percentage of the few written tokens are actually relevant. Thus we hope to anticipate the performance of the models when scaled up to the full summarization task and select a well-performing model.

The mean F_1-scores for ROUGE-1, ROUGE-2 and sentence-level ROUGE-L are reported in Table 4.11 for the ZSL setting and in Table 4.12 for the FSL one.

We observe that the perplexity of a model does not perfectly match its performance in this constrained summarization setting: In the ZSL setting, PEGASUS-LARGE performs second-best on TELCOAPP despite previously showing the highest perplexity. On the other side of the spectrum, PEGASUS-BIGPATENT performs worse than its perplexity would indicate; for example in the TELCOAPP-dataset where the FSL BigPatent model performs about as good as

	ROUGE-1 / ROUGE-2 / ROUGE-LSent		
	LogSummary	**Hadoop**	**TelcoApp**
PEGASUS-Large	50.9/27.2/48.2	75.3/**72.6**/75.1	**43.3/34.2/42.9**
PEGASUS-CNN	**58.9/32.9/55.1**	**77.8**/70.8/**77.7**	39.5/27.2/37.9
PEGASUS-XSum	41.0/25.5/39.3	66.0/60.9/65.6	26.9/12.0/24.9
PEGASUS-AESLC	41.8/23.7/39.5	46.1/33.4/45.9	31.0/15.4/29.3
PEGASUS-BigPatent	15.7/ 8.5/14.9	25.3/ 5.0/25.1	14.3/ 2.0/11.7

Table 4.11: Mean ROUGE precisions of ZSL PEGASUS models in a constrained summarization task.

	ROUGE-1 / ROUGE-2 / ROUGE-LSent		
	LogSummary	**Hadoop**	**TelcoApp**
PEGASUS-Large	**82.5/70.7/78.3**	**94.2/93.4/94.1**	**49.4/38.6/48.5**
PEGASUS-CNN	81.0/69.6/77.9	93.5/82.7/89.6	44.3/32.3/43.1
PEGASUS-XSum	81.7/69.4/77.6	92.9/89.7/92.7	45.4/30.0/43.2
PEGASUS-AESLC	79.0/64.6/75.3	92.5/91.1/92.4	47.7/31.1/45.9
PEGASUS-BigPatent	49.8/34.1/48.5	46.4/29.7/44.7	32.9/19.5/29.9

Table 4.12: Mean ROUGE precisions of FSL PEGASUS models in a constrained summarization task.

the AESLC model in the ZSL setting, despite PEGASUS-BigPatent now exhibiting a perplexity that is about 7 times lower than ZSL PEGASUS-AESLC.

Unsurprisingly, the models perform much better in a FSL setting after being fine-tuned on a few examples. We explore the quality of these generated texts in an example from the LogSummary-dataset in Table 4.13.

input	**reference summary**
`Accepted socket connection from remote host #83617`	`Accepted socket connection`
`Connection request from old client remote host #83617 ; will be dropped if server is in r-o mode`	`Connection request`
`Client attempting to establish new session at remote host #83617`	`Established session timeout`
`Established session 238367163526545476 with negotiated timeout 10000 for client remote host #83617`	
`[...]` *(messages repeat with 4 different clients)*	

PEGASUS-Large	**PEGASUS-CNN**	**PEGASUS-XSum**	**PEGASUS-AESLC**	**PEGASUS-BigPat.**
ROUGE-LSent precision: 100	ROUGE-LSent precision: 60	ROUGE-LSent precision: 16.7	ROUGE-LSent precision: 40	ROUGE-LSent precision: 33.3
`Accepted socket connection Established session`	`Accepted socket connection Client attempting`	`Client attempting to establish new session`	`Connection request from old client`	`A socket request from old client`

Table 4.13: FSL predictions on an example from the *ZooKeeper*-subdataset of LogSummary.

We find that, at least with this example, models originating from a more abstractive summarization background prefer to include grammatically sound segments in their summary, as can be seen for PEGASUS-XSum and PEGASUS-BigPatent.

PEGASUS-BigPatent's summary is the most abstractive one, combining two different log messages into a single statement. This hurts its score on the ROUGE metric, even if the summary is analogous to the one from PEGASUS-AESLC, and explains why this model performs poorly regarding this metric.

However, this more abstractive approach is also more error-prone. In another example, Table 4.14, PEGASUS-BigPatent includes additional descriptive text, which does not fit the context of the log: `binding to port` is an action taken by the software producing the log, but similar to the summaries of patents in the BigPatent dataset, the model seems to interpret the log as a procedure that needs to be described.

Therefore we do not believe that abstractive models perform worse solely because the ROUGE metric prefers extractive summaries, but because they do not fully understand the log-data presented to them and fail to interpret it in a meaningful way; even more so as they have not been provided with abstractive summaries during fine-tuning. Ultimately we think a purely extractive summary in the style of the human-written summaries of this dataset will be more helpful to any human operator than a factually inconsistent abstractive description of what happened.

Hence why we decide to select the PEGASUS-Large model for further evaluation: Comparing Table 4.12 and Table 4.11, it outperforms every other variant in the FSL setting and is competitive in the ZSL setting. As can be seen from the examples, its summaries follow the extractive style of the human-written summaries.

PEGASUS-CNN would be another viable alternative, as it shows even better perplexity, but evidently it performs slightly worse in this constrained generation setting. We speculate this is the case because PEGASUS-Large can more quickly adapt to the summarization task, as it does not have any prior biases towards which parts of the input document need to be included in the summary. Previous research has shown that summarization of news articles is prone to substantial layout biases [32], speaking against the use of previously fine-tuned models such as PEGASUS-CNN, which is accustomed to summarizing news articles.

input	**reference summary**

`binding to port remote host #44607remote host #5137`
`tickTime set to 2000`
`minSessionTimeout set to -1`
`maxSessionTimeout set to -1`
`initLimit set to 10`
`Reading snapshot zookeeper`
`My election bind port :  remote host #44607remote host #47103`
`LOOKING`
`New election.  My id = 1 , proposed zxid = 30064771479`
`Notification :  1 ( n.leader ) , 30064771479 ( n.zxid ) , 1 ( n.round ) , LOOKING ( n.state ) , 1 ( n.sid )`
`, 7 ( n.peerEPoch ) , LOOKING ( my state )`
`Have smaller server identifier , so dropping the connection :  ( 2 , 1 )`
`Received connection request remote host #46154`
`Have smaller server identifier , so dropping the connection :  ( 3 , 1 )`
`Received connection request remote host #7574`
`Notification :  3 ( n.leader ) , 30064771479 ( n.zxid ) , 1 ( n.round ) , LOOKING ( n.state ) , 2 ( n.sid )`
`, 7 ( n.peerEPoch ) , LOOKING ( my state )`
`[...]` *(4 similar messages discarded in this example for brevity)*
`Notification :  3 ( n.leader ) , 30064771479 ( n.zxid ) , 1 ( n.round ) , FOLLOWING ( n.state ) , 2 ( n.sid`
`) , 7 ( n.peerEPoch ) , LOOKING ( my state )`
	`binding to port`
	`Reading snapshot`
	`Have smaller server`
	`identifier`
	`dropping the connection`
	`Received connection request`

PEGASUS-Large	**PEGASUS-CNN**	**PEGASUS-XSum**	**PEGASUS-AESLC**	**PEGASUS-BigPat.**
ROUGE-LSent precision: 100	ROUGE-LSent precision: 40	ROUGE-LSent precision: 60	ROUGE-LSent precision: 100	ROUGE-LSent precision: 50

`Have smaller server`
`identifier`
`Received`
	`bind to port remote host`	`My state`
		`Received connection request`
			`binding to port`
			`Received connection`
				`A method for binding to`
				`port`

Table 4.14: FSL predictions on a more diverse example from the *ZooKeeper*-subdataset of LogSummary.

BART For the BART variants, the performances on this constrained generation task do not differ significantly between models, so we chose to omit the results here. Instead, we did perform a hyperparameter search for all variants on the LogSummary task, where the generated summaries are the shortest, and thus the search can evaluate different parameters in a relatively timely manner. To further speed up the search, it is performed on a manually selected validation set and not the whole dataset, resulting in 20 trials to determine an optimal length penalty. The resulting ROUGE-2 and summary-level ROUGE-L metrics on the evaluation dataset are visualized in Figure 4.2 (ZSL) and Figure 4.3 (FSL).

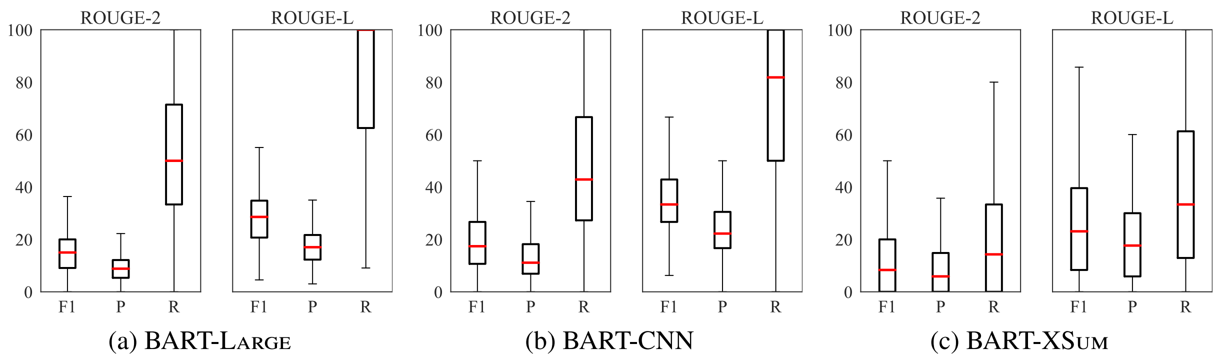


Figure 4.2: Boxplots showing ROUGE recall, precision and F_1-scores for different BART variants without further fine-tuning (ZSL) on the LogSummary dataset.

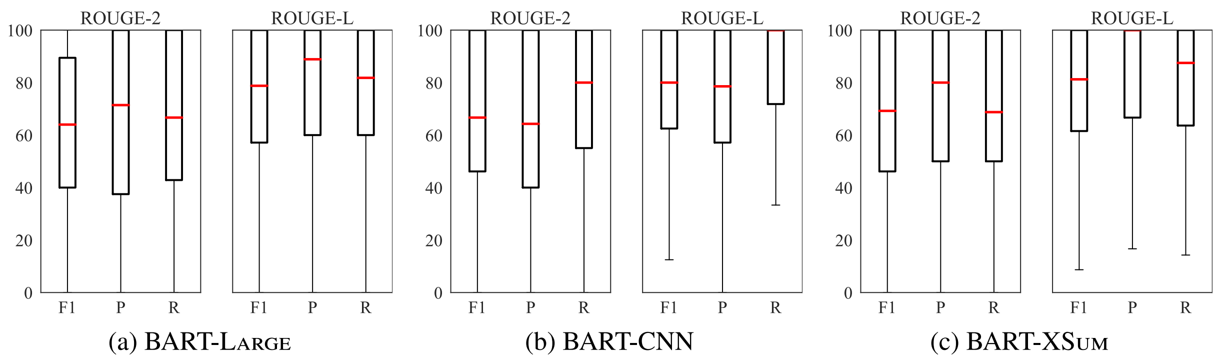


Figure 4.3: Boxplots showing ROUGE recall, precision and F_1-scores for different BART variants that have been further fine-tuned (FSL) on the LogSummary dataset.

In addition, we decide to qualitatively evaluate a summary generated in a ZSL setting, and compare it to a summary written after fine-tuning. To determine the models ability to select the log messages deemed relevant by the human-written reference, we choose an example whose ZSL-prediction scores nearest to the median of summary-level ROUGE-L recall on the *Spark*-subdataset of LogSummary. The precision would also measure the summary's conciseness, hence why we decide against using the F_1-score in this case. We show the input, human-written summary and BART-CNN's predictions in Table 4.15. BART-CNN is the BART-model that performed best in a ZSL setting according to perplexity. For the ZSL-prediction we highlight the portions of the generated summary that correspond to the reference.

The example in Table 4.15 shows that the model quickly adapts to the style of the reference summaries in the FSL setting. For this example in particular it is able to extract the same text segments as the human-written reference, albeit in a different order.

In the ZSL setting, the model correctly extracts some of the relevant log messages, however it does not simplify them in any way, leading to a longer summary. This is how we would want the model to behave on our other two datasets anyways, but here this partly explains why

input	**reference summary**
Block rdd stored as bytes in memory (estimated size 16.0 B , free 6.2 MB) Block rdd stored as bytes in memory (estimated size 16.0 B , free 6.2 MB) Times : total = 38 , boot = 16 , init = 22 , finish = 0 Times : total = 37 , boot = 20 , init = 17 , finish = 0 Times : total = 38 , boot = 16 , init = 22 , finish = 0 Times : total = 38 , boot = 16 , init = 22 , finish = 0 Times : total = 38 , boot = 21 , init = 17 , finish = 0 Finished task 14.0 in stage 1359.0 (TID 54654) . 2667 bytes result sent to driver [...] *(4 similar messages discarded in this example for brevity)* Got assigned task 54656 Running task 15.0 in stage 1359.0 (TID 54656) [...] *(3 similar messages discarded in this example for brevity)*	Running task Finished task Got assigned task Block stored in memory
BART-CNN ZSL	**BART-CNN FSL**
ROUGE-L recall: 54.54	ROUGE-L recall: 100
Block rdd ***stored*** as bytes ***in memory*** (estimated size 16.0 B, free 6.2 MB) Times : total = 38, boot = 16, init = 22, finish = 0 ***Finished task*** 14.0 in stage 1359.0 (TID 54654). 2667 bytes result sent to driver	Block stored in memory Finished task Running task Got assigned task

Table 4.15: Representative predictions on an example from the *Spark*-subdataset of LogSummary.

model-generated summaries in the ZSL setting do not perform as well concerning ROUGE precision and indirectly F_1-scores as in the FSL setting.

Nevertheless, the model also performs worse in recall; selecting a log message not considered important by the reference summary, and missing two segments that were considered relevant. This is a trend we can also observe when comparing the overall model performance from Figure 4.2 to Figure 4.3: BART-CNN performs much better regarding recall after fine-tuning.

Evidently all model variants perform significantly better concerning the overall F_1-scores after fine-tuning with a few examples. As we saw significant improvements in model perplexity, this was to be expected. Interestingly, on the LogSummary dataset this is mostly caused by a boost in precision, as recall was high even before further fine-tuning. As noted before, this jump in precision performance can be explained by the models' success in replicating the condensed style of the human-written summaries.

In the ZSL setting we can roughly see that the BART-CNN model performs best, which matches up with the observed perplexities, although the magnitude of differences between variants seem much less drastic.

Yet, the perplexities do not necessarily agree with the observed performance in the FSL setting, where BART-Large shows the overall least reliable F_1-scores despite scoring first in perplexity. Furthermore, the difference between the models seems to be mostly a trade-off between precision and recall, as the median F_1-scores mostly match up: BART-XSum performs best at precision, hinting at its ability to extract the most valuable information from each log message, while BART-CNN excels at recall.

Furthermore, we notice that model perplexity is also not comparable between different datasets, as FSL models perform better on the LogSummary-dataset than the Hadoop-dataset, but perplexities on the Hadoop-dataset are low compared the perplexity of the same models fine-tuned on LogSummary.

Overall, these results affirm our verdict that after training on a few examples, models previously fine-tuned for summarization do not perform significantly better on our summarization datasets than their baselines.

We decide to select the BART-CNN model for further evaluation, as its recall values are highest, which we take as an indicator that it manages to identify the most relevant log messages more reliably than the other the other two variants.

4.4.4 Final Results on Summarization Capabilities

Finally, we choose BART-CNN and PEGASUS-LARGE for thorough evaluation on all our datasets, a decision which we motivated in the previous section.

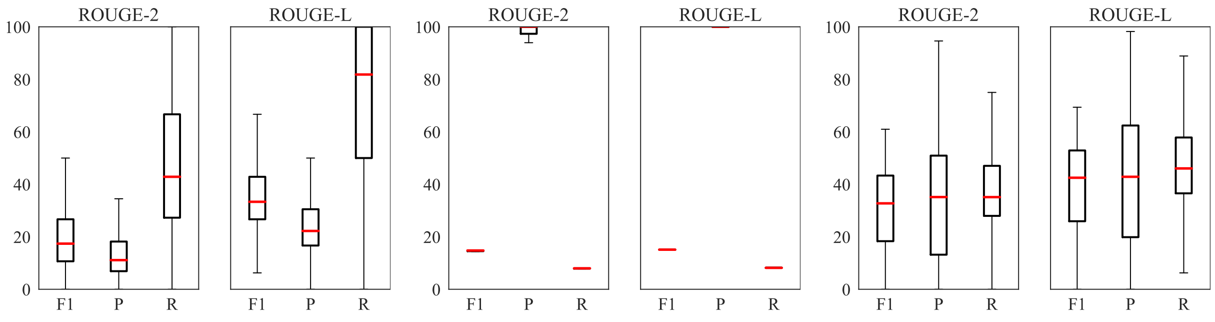
We perform a hyperparameter search comprising 20 trials for each model on a validation set for every dataset to determine an optimal length penalty for the beam-search. Additionally for the HADOOP-dataset, we disable the removal of duplicated trigrams of the beam-search for BART-CNN used in the original implementation [4]: The summaries in this dataset may contain many duplicated text segments, and prohibiting the model from generating such duplicate sequences will drastically limit its performance.

The mean F_1-scores for ROUGE-1 to ROUGE-4 as well as sentence-level ROUGE-L and summary-level ROUGE-L are reported in Table 4.19 on page 56 for the FSL scenario with the best results being highlighted.

As summaries in all our datasets are purely extractive, and solely include text segments that are also present in the input data, we should keep in mind that a "perfect" extractive model would be able to achieve a score of 100 on every ROUGE-metric.

A more detailed analysis of each models performance is made in the following two segments.

BART We examine the summarization capabilities in a ZSL setting and visualize the ROUGE-2 and summary-level ROUGE-L recall, precision and F_1-scores in Figure 4.4 for BART-CNN. For FSL the results are shown in Figure 4.5.



(a) LOGSUMMARY (b) HADOOP (c) TELCOAPP

Figure 4.4: Boxplots showing ROUGE recall, precision and F_1-scores of BART-CNN without further fine-tuning (ZSL) on all datasets.

While we previously found that fine-tuning improves performance substantially, this seems less clear when looking at the BART-model's performance on the TELCOAPP-dataset. The F_1 scores remained at almost the same level, though recall is noticeably higher; at the expense of a equally lower precision. Unfortunately, it seems that fine-tuning was not able to improve the performance on this dataset, though it should be noted that the model was able to extract 49.4%

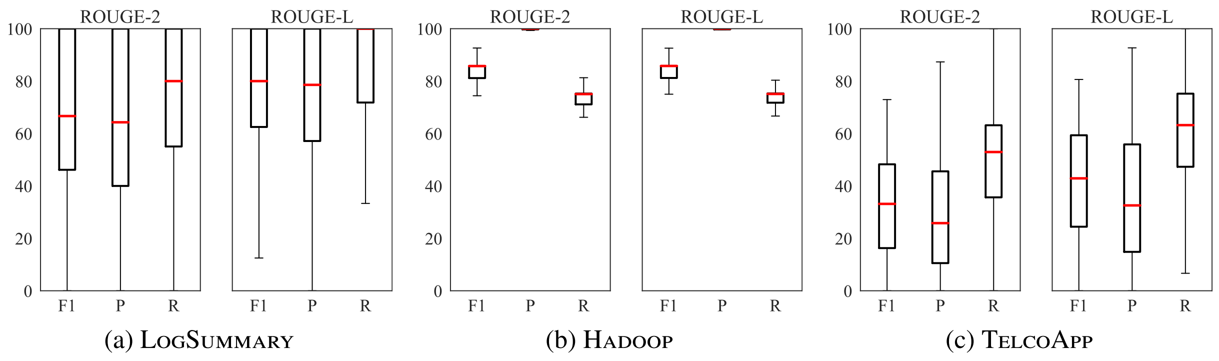


Figure 4.5: Boxplots showing ROUGE recall, precision and F_1-scores of BART-CNN with further fine-tuning (FSL) on all datasets.

of the information contained in the reference summary *without* any fine-tuning, according to the mean summary-level ROUGE-L recall. This is while keeping around 42.3% of the information in the summary relevant, according to the mean summary-level ROUGE-L precision.

We do observe however strict improvements on the Hadoop and LogSummary datasets, on Hadoop mostly due to a substantial improvement in recall. The results on both datasets seem promising.

However, the Hadoop-dataset is actually comprised of logs from 3 failure types, which are not present in equal proportions, while running one of two different applications. As the summaries vary in length and form between failure types, it becomes important to also consider the results for each failure separately. Thus we report the same metrics as before in Figure 4.4 for the ZSL case, and in Figure 4.5 for FSL.

Now we can see that the Hadoop-dataset includes mostly examples from a *network disconnection* failure, which overshadow the results on the two other failure types. The continued fine-tuning improved precision *and* recall, except for the *full disk* failure type where the continued training actually decreased the precision or recall in some cases.

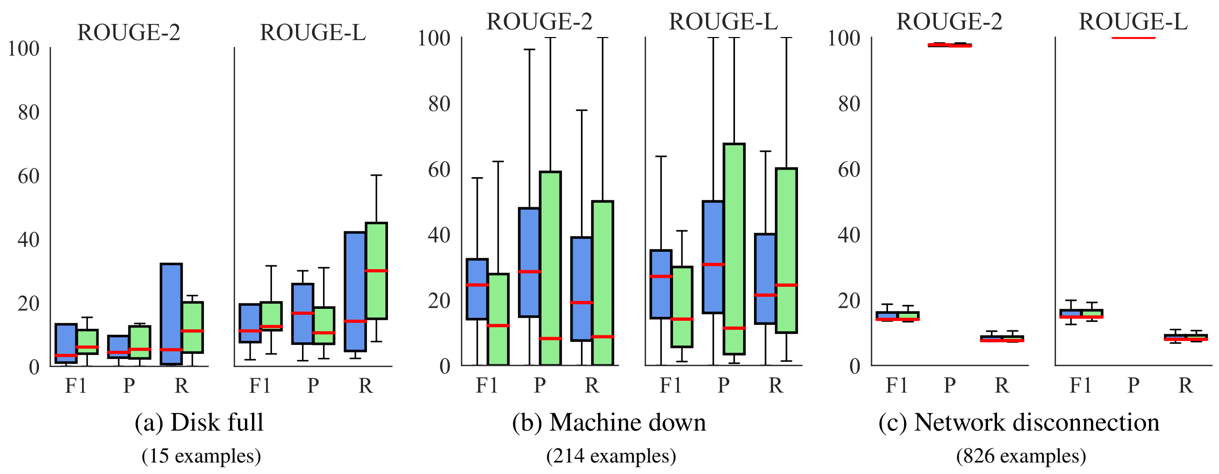


Figure 4.6: Boxplots showing ROUGE recall, precision and F_1-scores of BART-CNN without further fine-tuning (ZSL) on Hadoop categorized by failure type and application (PageRank on the left, WordCount on the right).

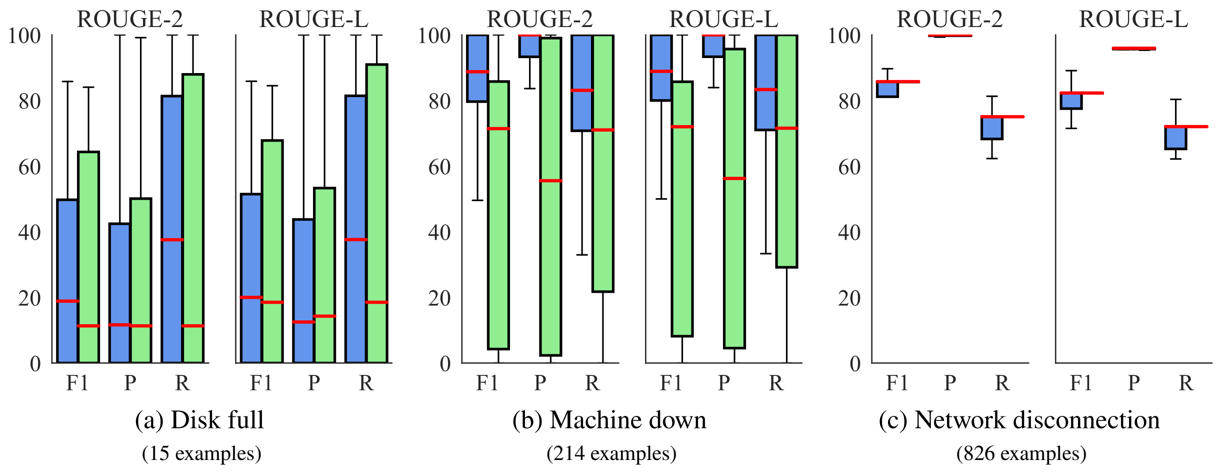


Figure 4.7: Boxplots showing ROUGE recall, precision and F_1-scores of BART-CNN with further fine-tuning (FSL) on Hadoop categorized by failure type and application (PageRank on the left, WordCount on the right).

It should be reiterated, that the training set contained 5 examples for each application on the *full disk* failure, leaving 11 examples from the WordCount application and 4 examples from the PageRank application in the test set for this failure type. Hence we might expect the models to do best in the FSL setting here, as it has trained with a higher proportion of the available examples compared to the other to failure types. However this is not at all what we observe: Both precision and recall vary widely and especially for the WordCount application little improvements could be made. It is not empirically sound to take conclusions concerning the performance difference between the applications on this failure type due to the imbalanced and low number of test examples available for each application.

An evaluation of the performance on the *machine down* failure is more meaningful, as the model has only seen 10 examples per application in this dataset. The test set then contains 181 examples for PageRank and 33 for WordCount. Again, this is quite imbalanced between applications, hence why we might expect the model to do better on the WordCount application, where it has been trained on a greater share of examples. Instead we observe that the model maintains its preference for the PageRank summaries from the ZSL-phase.

Regarding this same failure, for PageRank the model's summaries actually include 81.1% of the tokens contained in the non-normal log messages according to the mean summary-level ROUGE-L recall after pre-training. On the other hand, the summaries contain 16.7% tokens not present in the reference summaries according to the complement of the mean summary-level ROUGE-L precision. As such, the model does manage to identify over $\frac{4}{5}$ of the relevant information, while keeping the amount of irrelevant information close to $\frac{1}{6}$.

Though, one-sixth of *factually inconsistent* information would still be harmful to the quality of the summaries. Unfortunately ROUGE is unable to judge the origin or truthiness of the irrelevant portions of the summary. As such, and to further gain an intuitive understanding of the big variance in performance between the failure types, we also qualitatively examine one representative example for each failure.

We choose an example whose FSL-prediction scores nearest to the median of summary-level ROUGE-L F_1 score for each failure type, to get an idea of the model's average summarization ability. For comparison we also show the summary in the ZSL setting, but omit the input to the model for brevity.

The results are shown in Table 4.16 for the *full disk* failure, Table 4.17 for the *machine down* failure, and Table 4.18 for the *network disconnection* failure.

reference summary	**BART-CNN FSL**	**BART-CNN ZSL**
	ROUGE-L F_1: 18.52	ROUGE-L F_1: 11.88
`Task: attempt#59489 - exited : DiskChecker: Could not find any valid local directory for out-path Diagnostics report from attempt#59489: Error: DiskChecker: Could not find any valid local directory for out-path Diagnostics report from attempt#59489: Error: DiskChecker: Could not find any valid local directory for out-path 1 failures on node microsoft.`	`Diagnostics report from attempt#83820: Container killed by the ApplicationMaster. Container killed on request. Exit code is 137 Container exited with a non-zero exit code 137 Diagnostics report from attempt#97248: Container killed by the ApplicationMaster. Container killed on request. Exit code is 137 Container exited with a non-zero exit code 137`	`Diagnostics report from attempt#83820: Container killed by the ApplicationMaster. Exit code is 137 Container exited with a non-zero exit code 137. After Scheduling: PendingReds:0 ScheduledMaps:0 ScheduledReds:0 AssignedMaps:3 AssignedReds:1 CompletedMaps:8 CompletedReds:0 ContAlloc:19 ContRel:0 HostLocal:10 RackLocal:8.`

Table 4.16: Representative predictions on a full disk failure of HADOOP.

reference summary	**BART-CNN FSL**	**BART-CNN ZSL**
	ROUGE-L F_1: 88.89	ROUGE-L F_1: 22.43
`Retrying connect to server: microsoft. Already tried 15 time(s); maxRetries=45 [...]` *(message repeats 4 times with different retry-counts)*	`Retrying connect to server: microsoft. Already tried 15 time(s) maxRetries=45 [...]` *(messages repeat 3 times with different retry-counts)*	`Retrying connect to server: microsoft. Already tried 15 time(s) maxRetries=45 Progress of TaskAttempt attempt#14835 is : 0.23333333 MapCompletionEvents request from attempt#14835. startIndex 7 maxEvents 10000 [...]` *(message repeats 2 times)*

Table 4.17: Representative predictions on a machine down failure of HADOOP.

If we interpret the summaries for the full disk failure in Table 4.16, we notice that the model has learned that messages starting with `Diagnostics report from attempt` usually are important. Looking at the reference summary provided in that example, this does seem to be usually the case. Unfortunately, in this specific case this heuristic did not work out for the model, because the cancellation of an attempt is actually an benign activity, and the model failed to include any of the more relevant messages about disk failures.

In the case of the machine down failure in Table 4.18, the model missed to include one repetition of the relevant message and would have achieved a perfect score otherwise.

Similarly, for the network disconnection failure in Table 4.18 we can actually speculate that the FSL model would have correctly continued the message, however it was cutoff by a maximum length limit we imposed upon the beam-search. As this is the typical structure of a network disconnection failure, we can expect that the recall would actually improve on this failure type, if we had allowed the model to generate longer summaries. As it maintained a close to perfect precision on this failure until now, this would probably result in a strict improvement of F_1 scores.

reference summary	**BART-CNN FSL**	**BART-CNN ZSL**
	ROUGE-L F_1: 82.28	ROUGE-L F_1: 15.09
`Failed to renew lease for NONMAPREDUCE for 1389 seconds. Will retry shortly... NoRouteToHostException: No Route to Host from MININT-FNANLI5remote host #73234 to msra-sa-41:9000 failed on socket timeout exception: NoRouteToHostException: No route to host: no further information; For more details see: http-URL Caused by: NoRouteToHostException: No route to host: no further information Address change detected. Old: msra-sa-41remote host #94343 New: msra-sa-41:9000 [...]` *(messages repeat 6 times with incremented timeout)*	`Failed to renew lease for NONMAPREDUCE for 1389 seconds. Will retry shortly... NoRouteToHostException: No Route to Host from MININT-FNANLI5remote host #73234 to msra-sa-41:9000 failed on socket timeout exception: NoRouteToHostException: No route to host: no further information For more details see: http-URL Caused by: NoRouteToHostException: No route to host: no further information Address change detected. Old: msra-sa-41remote host #94343 New: msra-sa-41:9000 [...]` *(messages repeat 4 times with incremented timeout)* `Failed to renew lease for NONMAPREDUCE for 1394 seconds. Will retry shortly... NoRouteToHostException: No Route to Host from`	`Failed to renew lease for NONMAPREDUCE for 1389 seconds. Will retry shortly... NoRouteToHostException: No Route to Host from MININT-FNANLI5remote host #73234 to msra-sa-41:9000 failed on socket timeout exception: NoRouteToHostException: No route to host: no further information.`

Table 4.18: Representative predictions on a network disconnection failure of Hadoop.

Overall, the model does well in the last two failure types, where summaries often repeat a message multiple time, but does worse for the more diverse full disk failure. On the other hand, this is also where the model has experienced less training.

The ZSL model performs worse in all examples, managing to select some of the same messages as the model in the FSL setting, but also including a great portion of irrelevant messages.

Yet, we have to wonder whether this success in caused by the models summarization abilities, or if the summaries are just too trivial. For the network disconnection failure specifically, the repeated message which is part of the summary is quite long. As such they likely represent a majority of the input data.
We decide to investigate and calculate the ratio of log messages shared between the model inputs and the reference summaries, visualizing the results in Figure 4.8.

It is clear now that on the network disconnection failure we do not actually evaluate the model's performance to summarize, since the input and reference summaries are identical in most cases. Rather, the FSL model as just learned to replicate the relevant message as many times as needed, while the ZSL model just selects the first occurrence. However, the summarization task is meaningful on the other two failure types. Concerning the machine down failure type, we can now infer a cause for the difference in performance between the logs from the PageRank and WordCount applications: As indicated by the higher median, the summaries for the PageRank application share more log messages with the input documents. We assume this is one cause of the higher performance we observe for logs originating during the execution of the PageRank application.

All in all, the example predictions were factually consistent, including only log messages present in the input data, but sometimes selecting irrelevant ones. From manually inspecting other examples, we conclude that this is the situation in general.

PEGASUS Again, we visualize the ROUGE-2 and summary-level ROUGE-L recall, precision and F_1-scores in Figure 4.9 for PEGASUS-Large under the ZSL setting. For FSL the results

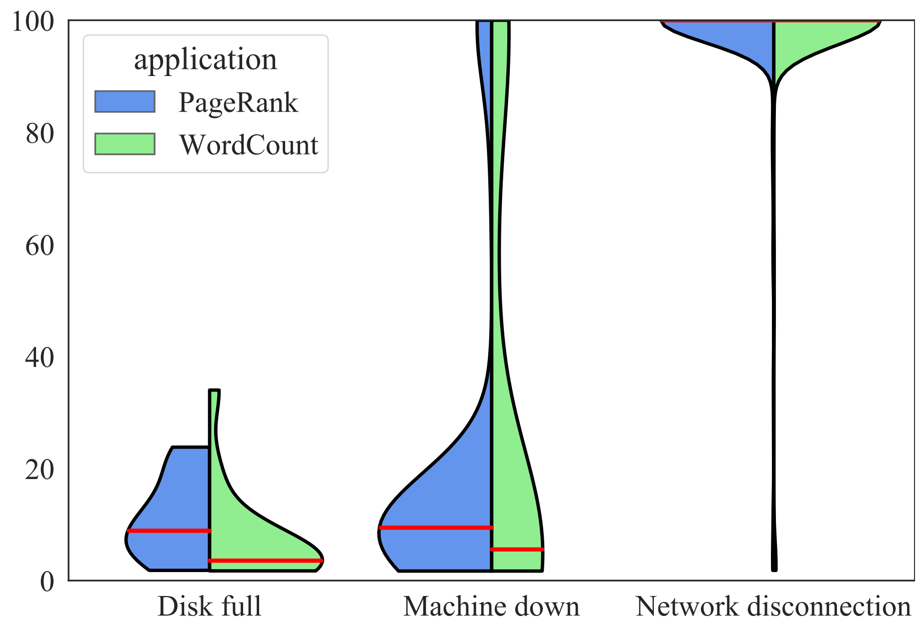


Figure 4.8: Violin plot showing the percentage of log messages shared between model inputs and the summaries on HADOOP.

are shown in Figure 4.10.

The fine-tuning for PEGASUS-LARGE actually helped on all datasets, even on TELCOAPP. However, one should point out that BART-CNN performed more than twice as good in a ZSL setting compared to PEGASUS-LARGE, when considering the median F_1-score for summary-level ROUGE-L. Even after fine-tuning this situation does not improve by much.

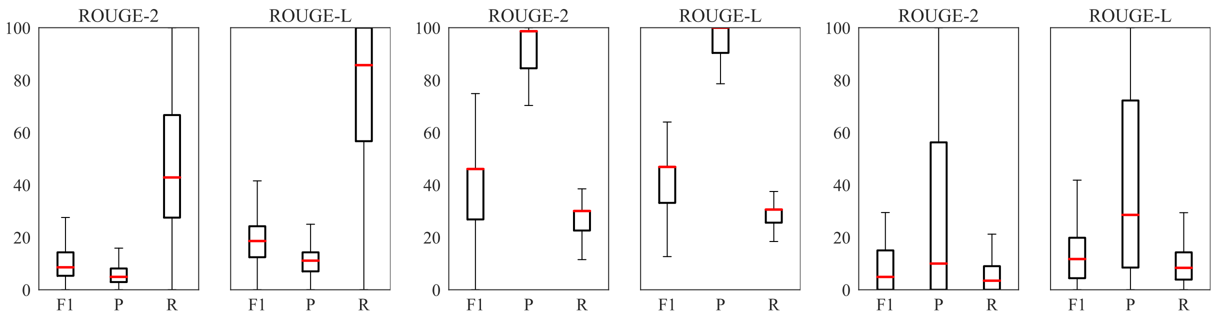
On the TELCOAPP dataset we observe that the PEGASUS model manages to achieve higher precision than the BART model in a FSL setting, at the expense of recall.

Finally, on the HADOOP dataset the achieved performances vary much more than with BART-CNN. This remains true when considering the results by failure type, shown in Figure 4.11 for the ZSL case, and in Figure 4.12 for FSL.

Altogether, BART-CNN outperforms PEGASUS-LARGE in all observed settings as can be seen in Table 4.19, where the best results have been highlighted. As such we decide to omit any further investigations of the performance of the PEGASUS model for the sake of brevity.

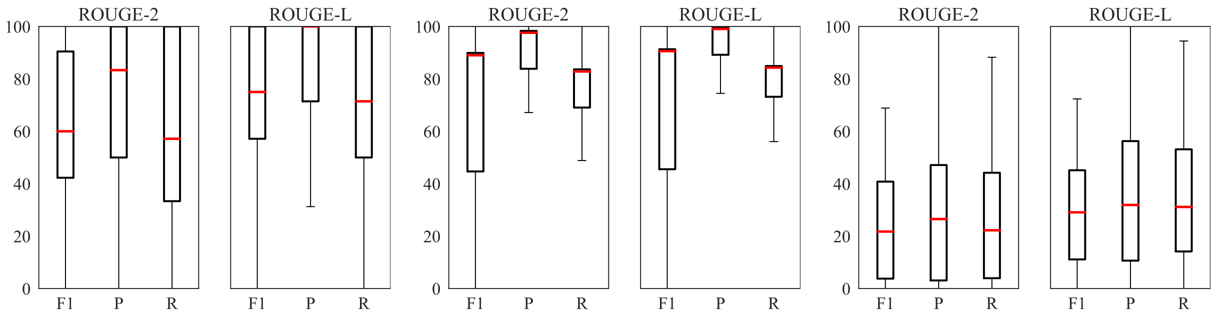
	BART-CNN / PEGASUS-LARGE		
	LOGSUMMARY	**HADOOP**	**TELCOAPP**
ROUGE-1	**76.3**/72.9	**75.8**/70.4	**40.5**/29.6
ROUGE-2	**66.0**/61.7	**75.6**/68.5	**33.0**/23.7
ROUGE-3	**52.8**/45.9	**75.4**/66.9	**29.2**/21.5
ROUGE-4	**42.0**/33.6	**75.3**/65.4	**26.1**/19.7
(sentence) ROUGE-L	**69.7**/67.9	**75.6**/69.4	**37.2**/26.9
(summary) ROUGE-L	**76.3**/72.9	**75.8**/70.3	**40.4**/29.4

Table 4.19: Mean ROUGE F_1-scores of BART-CNN (left) and PEGASUS-LARGE (right) in the FSL setting on all datasets.



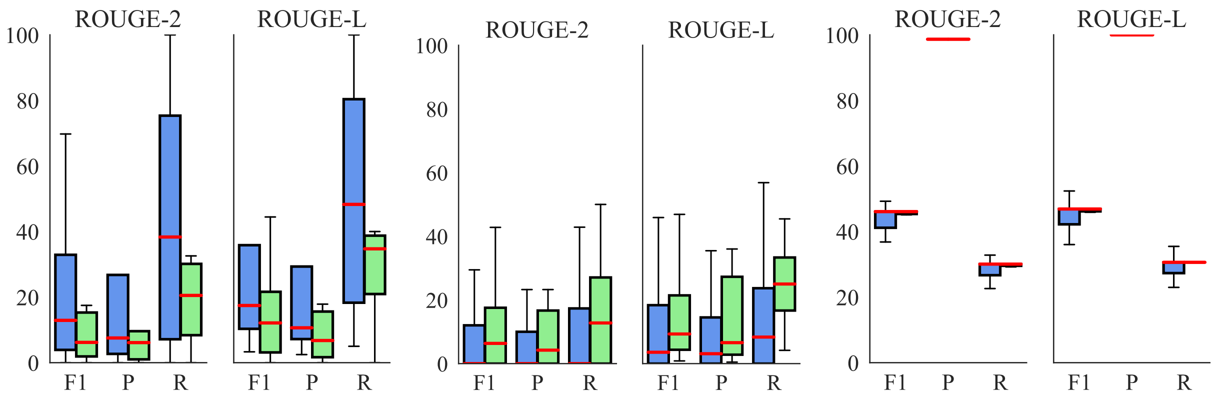
(a) LogSummary (b) Hadoop (c) TelcoApp

Figure 4.9: Boxplots showing ROUGE recall, precision and F_1-scores of PEGASUS-Large without further fine-tuning (ZSL) on all datasets.



(a) LogSummary (b) Hadoop (c) TelcoApp

Figure 4.10: Boxplots showing ROUGE recall, precision and F_1-scores of PEGASUS-Large with further fine-tuning (FSL) on all datasets.



(a) Disk full
(15 examples)

(b) Machine down
(214 examples)

(c) Network disconnection
(826 examples)

Figure 4.11: Boxplots showing ROUGE recall, precision and F_1-scores of PEGASUS-Large without further fine-tuning (ZSL) on Hadoop categorized by failure type and application (PageRank on the left, WordCount on the right).

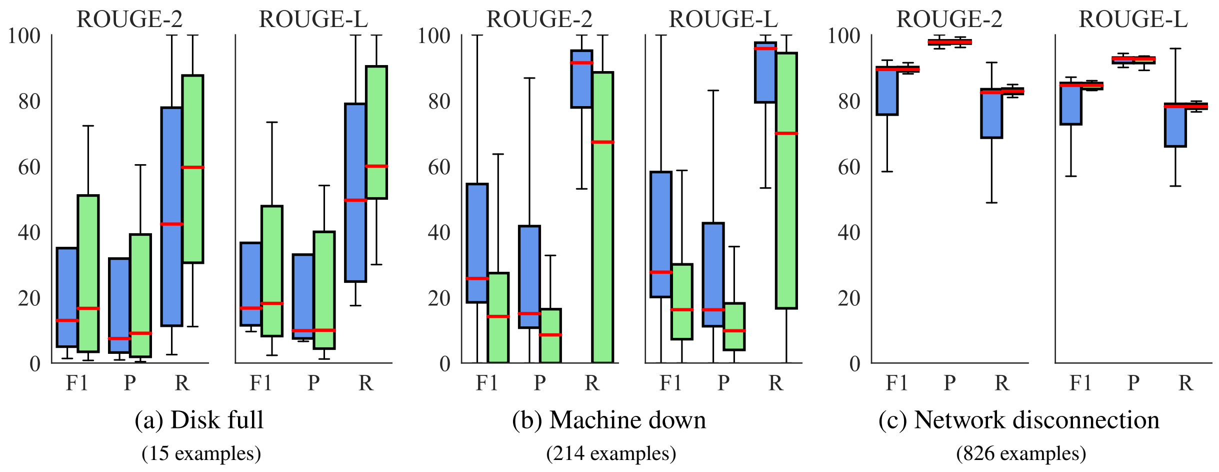


Figure 4.12: Boxplots showing ROUGE recall, precision and F_1-scores of PEGASUS-Large with further fine-tuning (FSL) on Hadoop categorized by failure type and application (PageRank on the left, WordCount on the right).

4.4.5 Discussion

We start the discussion of the final results by providing an overview of previous observations:

- ◇ Even fine-tuning on a fraction of the available examples greatly improves performance.
- ◇ Models previously fine-tuned for abstractive summarization (e.g. PEGASUS-BigPatent) perform poorly in our setting, as our datasets are purely extractive.
- ◇ The best models manage to keep summaries factually consistent, by only including segments previously seen in the input data.
- ◇ PEGASUS models fail to adapt to the domain of logs as fast as BART models do.
- ◇ Summarization of logs resulting from a network disconnection failure in Hadoop is not meaningful. These represent a majority of the examples in this dataset. Instead the results on the other two failure types are of greater relevance.

Given that PEGASUS outperformed BART on previous summarization datasets and it represents a model with more parameters, its poor performance in our experiments is somewhat unexpected. However, the difference between BART and PEGASUS is more pronounced on the XSum dataset, than the CNN/DailyMail dataset [4, 5], meaning that PEGASUS is generally better at abstractive summarization. Since our datasets are highly extractive, PEGASUS' performance improvements may simply be less significant. Furthermore, we observed that the perplexity of PEGASUS models converges slower than the one of BART models. As such PEGASUS models may need an increased number of fine-tuning epochs to reach their full potential.

On another note, one problem we frequently observed is a precision/recall-tradeoff: Optimizing one measure often comes at the cost of lowering the other. We see this between models originating from differing domains (e.g. the BART variants in Figure 4.3), or when comparing the ZSL BART-CNN model with the FSL one on TelcoApp. In the more general setting of information retrieval this is a well-studied problem, and has been shown to be unavoidable in many situations [54].

In some situations, it is beneficial to prioritize one measure over the other; receiving a short but precise description of the most important aspect of a segment of log data is helpful to gain an overview, but is not sufficient to analyze the causes of a problem. Conversely, on a longer span of log-data which shows high recall it requires more effort to gain a superficial understanding, but it can be analyzed more thoroughly.

For TelcoApp, even without fine-tuning BART-CNN was able to capture almost half of the information contained in the reference summary, but still a majority of the summary does not contain relevant information. This performance is not good enough to replace manual analysis of the log-data, but an operator may still consult the model to get a rough idea of the anomalous log messages before diving deeper into the details, speeding up the analysis conducted.

The summarization task on the Hadoop dataset (and to an extend on the TelcoApp dataset as well) could be interpreted as an *anomaly detection* task, where the model is asked to identify anomalous log messages: After all, this is essentially the premise of our summarization task based on non-normal log events.

On these types of inputs, the summary length may vary significantly, depending on how many anomalies are present. Not every failure produces the same amount of anomalous log messages (as can be seen with the Hadoop-dataset), and the model may be asked to investigate normally occurring portions of a log, where there is not a lot to report on. It may thus be tricky to find optimal values for the length penalty, affecting a model's performance.

Training a model as a binary classifier ("Is a given message relevant or not?") may be better suited for this task than an abstractive sequence-to-sequence architecture. One needs not to worry about the model introducing factual inconsistencies, or finding optimal values for the length penalty. Sequence-to-sequence architectures still have an advantage over traditional classifiers, in that they are naturally able to consider the surrounding *context* of a message to judge if it is important or not.

On the other hand, there may also be requirements for summarization similar to that in LogSummary, where summarization of benign activity is normal and represents a majority of the data presented to the model. Here the length of the summaries can be expected to be a stable proportion of the input length, and sensible values for hyperparameters like the length penalty may be chosen in advance. We analyze the performance on the LogSummary dataset in greater detail in section 4.6, however our results suggest that BART-CNN operates well in this kind of setting.

Last but not least, it is arguably not helpful for a human operator to receive a summary with the same type of log message repeated multiple times, as is the case in the Hadoop-dataset. As a further preprocessing in the summarization based on *non-normal log events*, log messages could be excluded from a summary whose log events occur more than once in the reference summary.

Since the model would not be encouraged to repeat the same group of anomalous log messages, this would result in more concise summaries, and possibly more accurate ones, because the model has less chances to introduce inconsistencies. By removing redundancy, the same amount of useful information could be presented to human operators, while the model would be able to scan over longer portions of the log without running into problems of input limitation. Shorter summaries also lead to less computation time required for the beam-search.

However, omitting this preprocessing step may be helpful in situations where the dynamic contents of a log message (its parameters) contains important information and dropping log messages of duplicated event type loses this information.

Conclusion We conclude that using models previously fine-tuned for summarization in other domains does not substantially contribute to better performances. Instead it seems sufficient to use a pre-trained model and fine-tune it directly on the limited log-data available. As is perhaps to be expected, summarization of log-data does not seem to have that much in common with other summarization domains. Although we studied a wide range of previously researched summarization domains, there may be other summarization datasets where this does not hold true, especially concerning extractive summarization tasks. Nevertheless, we suspect any benefits from previous fine-tuning to become less relevant when fine-tuning is scaled up to larger log summarization datasets.

Overall, the application of pre-trained NLP models to log data was a successful proof-of-concept, however further research is needed to construct better datasets for log summarization. Models are able to reproduce log data seen in their inputs, and are able to pick up distinctions between important segments of logs and less important ones. Our best-performing models still achieve some meaningful performances. For instance, BART-CNN FSL is able to retrieve 81.1% of the anomalous information contained in its inputs with high precision when observing the *machine down* failure of the PageRank application (according to summary-level ROUGE-L). On the LogSummary dataset, BART-CNN FSL shows promising results, reaching a summary-level ROUGE-L score of 76.3 out of a theoretically possible 100; The performance on this dataset is further explored in a later section.

4.5 Effects of further Pre-Training on Log-Data

Previous research suggests that further pre-training can significantly improve the performance on later downstream tasks, especially when data for fine-tuning is not available in larger quantities and the domain a model was trained on differs from the domain it is applied in [42, 43]. In this section we thus investigate the effects of further pre-training on the performance in log summarization.

As detailed on page 21 we implemented the self-supervised pre-training objective of BART, and apply it to the BART-Base model. It is infeasible to pre-train an already fine-tuned BART model such as BART-CNN. Hence we use the smaller BART-Base baseline model, because pre-training it is faster due to the reduced amount of parameters that are adjusted during training.

We decide to pre-train the model on our largest collection of logs: The logs forming the basis of the TelcoApp dataset. Incidentally, this allows us to examine if pre-training can help to improve the performance on the dataset our models found most challenging. Additionally, we test whether pre-training is helpful when we later fine-tune the model on a dataset distinct from the one using during pre-training. As such we also evaluate the models in the LogSummary dataset .

4.5.1 Experimental Setup

We evaluate BART-Base in four different settings:

***zero-shot learning* (ZSL)** The model is directly evaluated on the summarization task without further training.

***few-shot learning* (FSL)** The model is fine-tuned on the summarization task and then evaluated.

***self-supervised pre-training & ZSL* (Pre+ZSL)** The model is pre-trained in a self-supervised manner on the log-data and then evaluated on the summarization task without further fine-tuning.

***self-supervised pre-training & FSL* (Pre+FSL)** The model pre-trained on the log-data is further fine-tuned on the supervised summarization task and then evaluated.

The FSL fine-tuning is conducted in the same manner as described in section 4.4, including the same number of steps, batch-sizes and learning rates.

For pre-training we leverage our largest collection of logs: The logs forming the basis of the TelcoApp dataset. The log segments used as the basis of our summaries only represent a fraction of the provided dataset; As some manual analysis is still needed to identify the relevant segments of the logs where our summarization based on common log events could be applied on, we only produced reference summaries for a few log files. Before training, we remove the log files that formed the basis of our summaries, still leaving us with multiple Gigabytes of log-data. We do this to prevent the model from later copying inputs it has seen during pre-training. However, we do not remove logs which originate from similar root causes, so the logs seen during pre-training are still relevant and similar in style.

During pre-training, we use a comparatively large effective batch size of 8192, similar to the batch sizes used by PEGASUS and BART during pre-training [4, 5]. The reasoning provided by Lewis *et al.* is that previous work demonstrates the effectiveness of large batch sizes during pre-training [4]. Analogous to the fine-tuning phase, we use a constant learning rate of $5 \cdot 10^{-5}$. We pre-train for 14 epochs, the number of times the model trains on the entire log dataset, resulting in a total of 160 pre-training steps.

Compared to the 500 thousand steps the large BART variants have experienced during their pre-training, this is a small amount of steps, relatively speaking. The pre-training is still quite time-intensive for our computing setup, so we were unable to scale up pre-training to much more steps. However, due to the large batch sizes, the model has actually seen a considerable amount of log data, and may already employ its knowledge from its previous pre-training. The role of the continued pre-training is therefore to let the model become acquainted with the log-data, not to train the model from scratch.

4.5.2 Results

We visualize the cross-entropy loss of BART-Base during pre-training in Figure 4.13.

Additionally we record the cross-entropy loss on the evaluation data while fine-tuning and report it in Figure 4.14 for TelcoApp and in Figure 4.15 for LogSummary.

Finally, we present the results on the summarization datasets in Table 4.20 for TelcoApp and

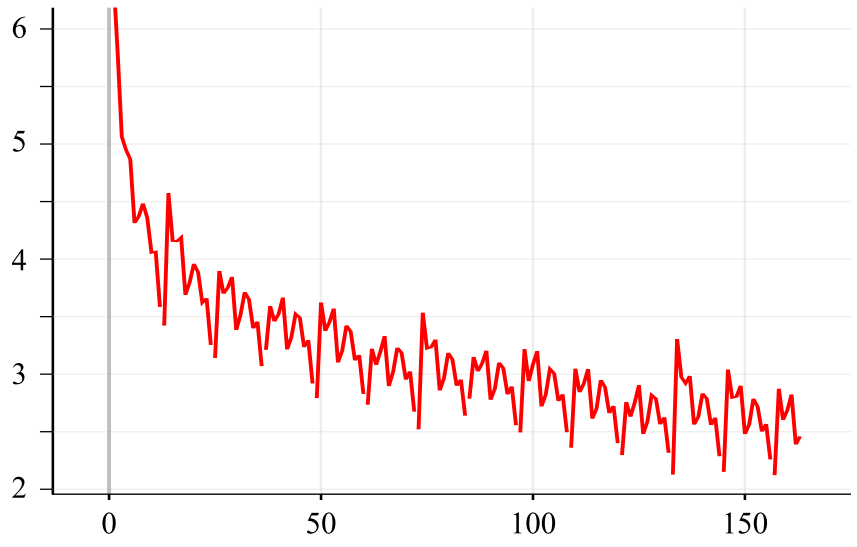


Figure 4.13: Cross-entropy loss during the pre-training of BART-Base.

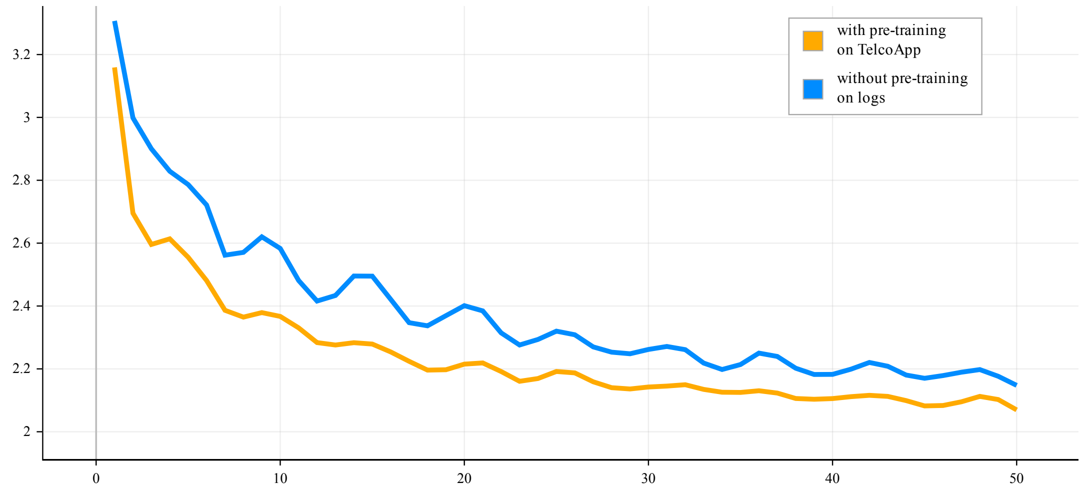


Figure 4.14: Cross-entropy loss on the test set during the fine-tuning of BART-Base and the variant pre-trained on TelcoApp.

in Table 4.21 for LogSummary. The F_1 scores of ROUGE-1, ROUGE-2 and sentence-level ROUGE-L are reported, and the best results highlighted between the model that experienced further pre-training and the one that did not. For comparison, we also include the results of BART-CNN from section 4.4.

	ROUGE-1 / ROUGE-2 / ROUGE-LSent		
	BART-Base	**BART-Base**	**BART-CNN**
	not pre-trained on log-data	further pre-trained on **TelcoApp**	**not** pre-trained on log-data
zero-shot learning (ZSL)	35.2/**26.3/29.2**	**36.4**/25.4/26.1	39.2/31.1/33.3
few-shot learning (FSL)	39.4/31.5/35.8	**39.9/32.1/35.9**	40.5/33.0/37.2

Table 4.20: Mean ROUGE F_1-scores on TelcoApp for the different BART variants.

We observe that pre-training alone does improve ZSL performance a bit, at least in the LogSummary dataset, but is simplify not enough to create a model able to produce meaningful

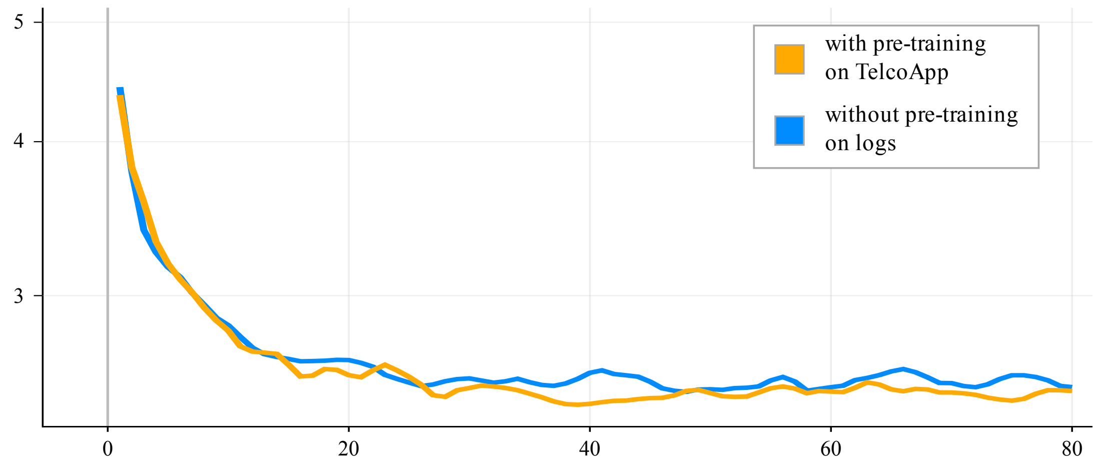


Figure 4.15: Cross-entropy loss on the test set during fine-tuning of BART-Base and the variant pre-trained on LogSummary.

	ROUGE-1 / ROUGE-2 / ROUGE-LSent		
	BART-Base **not** pre-trained on log-data	**BART-Base** further pre-trained on **TelcoApp**	**BART-CNN** **not** pre-trained on log-data
zero-shot learning (ZSL)	25.6/13.8/23.6	**26.8/14.4/24.3**	35.5/20.6/33.1
few-shot learning (FSL)	76.6/**66.4/71.1**	**76.9**/66.3/70.3	76.9/65.9/69.9

Table 4.21: Mean ROUGE F_1-scores of .

summaries. As the model has never been fine-tuned for summarization before, analogous to BART-Large, this is to be expected. The model has not yet learned that we expect it to write summaries.

All in all, the scores are quite similar after fine-tuning, even when compared to BART-CNN.

4.5.3 Discussion

As can be seen in Figure 4.13 the BART-Base model quickly adapts to the new pre-training data, with the cross-entropy halving within the first 4 epochs. By the time it reaches around 130 steps, the gains start to stagnate however. To achieve further significant gains, we believe it to be likely that the pre-training has to be scaled up to more steps (1000 - 10000).

Furthermore, the cross-entropy fluctuates considerably. Whenever an epoch of pre-training is finished and a new one starts, the cross-entropy spikes. This suggests that the log dataset is not homogeneous and the model partly forgets how to handle logs at the beginning of the dataset by the time it finishes an epoch. The model still is able to consistently lower its cross-entropy overall, but in the future one may want to shuffle the training data randomly to smooth out the pre-training process.

If we examine the performance of the pre-trained model in the summarization task on TelcoApp, we notice that the continued pre-training indeed helped the model better predict the given data; during fine-tuning it adapts more swiftly and maintains its head-start in cross-entropy even after the fine-tuning is finished.

Unfortunately, the positive impact of continued pre-training on cross-entropy remains confined to the TelcoApp dataset. On LogSummary, the pre-training in the domain of log-data did not help BART-Base to make significant performance gains, neither regarding perplexity nor the ROUGE metrics. This is likely due to the difference in style, format and content of various log datasets. Since LogSummary and TelcoApp are diverse and distinct from one another, the model was not able to transfer its newfound knowledge.

From the perspective of cross-entropy, the pre-training was successful for TelcoApp. This improvement in cross-entropy and perplexity however does not translate in a significant difference in summarization performance: Both the Pre+FSL model and the FSL baseline show near identical ROUGE-scores.

Ultimately, continued pre-training did not deliver on the performance improvements we initially expected from it. We speculate that continued pre-training can still be useful for other tasks than the ones we studied because the model was able to achieve lower perplexity much faster, even if pre-training did not help on our datasets.

Last but not least, we remark that after fine-tuning, the base model achieved similar performances compared to the BART-CNN model on both datasets. To us, this indicates not only that previous training in another summarization dataset is not that relevant to our application but also that smaller models may be able to achieve similar performances to variants with more parameters. Since smaller models consume less memory and are faster, both during training and text generation, this is a promising result.

4.6 Comparison with the LogSummary framework

Besides introducing a dataset of summarized log-data, Meng *et al.* also introduced their own framework for summarizing logs called *LogSummary* [6], which is explained in greater detail in chapter 5.

In this section, we aim to compare our results directly with this previous work. As the LogSummary framework is specifically laid out to only extract parts of any given log messages, it would perform badly on the two summarization datasets Hadoop and TelcoApp we introduced ourselves, where models are expected to reproduce entire log messages. Therefore we only compare on the LogSummary dataset, however we will differentiate between each subdataset as was done in [6].

4.6.1 Experimental Setup

We choose our best performing model (BART-CNN) and compare it with the LogSummary framework.

To keep the comparison fair, we need to skip our simplification step from subsection 3.4.2, as the framework works directly on the log messages without needing any simplification. As such, we take the baseline BART-CNN and fine-tune it on the raw log messages, otherwise using the same FSL setting as in the previous experiment. To find an optimal length penalty for the beam-search, we again employ an automatic hyperparameter search with 20 trials.

We take this as an opportunity to test how helpful our simplification step was for the model, by comparing the results with the BART-CNN model fine-tuned on the simplified messages.

For the LogSummary framework, we use the results as reported in [6]. For the two datasets (Spark and Zookeeper) not included in the original article, we manually run the LogSummary framework on these datasets and fix any errors encountered.[7] This also requires us to manually train two Log2Vec [55] models, which are used by the LogSummary framework.[8] As we could not find any specific information on the amount of data used to train the Log2Vec models in [6], we decide to use the `2k.log`-files provided by [51] for the respective datasets, as this is also the example data used in the Log2Vec repository.

To verify we conducted this process correctly, we evaluate the LogSummary framework on the HDFS dataset using their pipeline and receive mean ROUGE-1 scores of 54.7, 72.1 and 45.4 for F_1, precision and recall respectively. For the BGL dataset we obtain 77.4, 83.0 and 76.1. While these do differ from the scores reported in [6], they do so only by a small margin of 1 to 5 points. Therefore we assume we executed the LogSummary framework correctly and proceed to evaluate it on the two new datasets.

For this experiment only, we use a different implementation of the ROUGE metric, namely the one used by the LogSummary framework[9], to keep the comparison fair.

4.6.2 Results

Following [6], we primarily investigate values of the ROUGE-1 metric: The reasoning for comparing only the ROUGE-1 values provided by Meng *et al.* is that the different authors of each reference summary may have written words and phrases in different succession, ranking extracted segments by different priorities. The results are shown in Table 4.22 with the best results on each subdataset highlighted.

The results for the LogSummary framework are for the whole dataset, while the results of our model are calculated on the test set only. Since 95 out of 100 examples in each subdataset are in the test set, the results should still be comparable.

We also report the sentence-level ROUGE-L scores for our approach: In Table 4.23 we display the performance of our BART-CNN model on each subdataset of LogSummary, and provide the results of the BART-CNN fine-tuned on the simplified messages from section 4.4 for comparison.

Sentence-level ROUGE-L provides a measure to evaluate how much information overlap there is between the model generated summary and human-written references, additionally estimating the model's ability to preserve the order of extracted segments. Thus we can also assess how well the model can prioritize segments from the subjective ordering assigned to them by the authors of the summaries.

[7] LogSummary's repository is available at: `https://github.com/WeibinMeng/LogSummary`
[8] Log2Vec's repository is available at: `https://github.com/NetManAIOps/Log2Vec`
[9] That implementation is available at `https://github.com/pltrdy/rouge`.

	F_1 / precision / recall	
	LogSummary framework	**BART-CNN FSL** (ours)
BGL	**72.5/81.5**/70.3*	71.9/72.8/**75.3**
HDFS	53.8/75.9/43.2*	**77.2/80.2/75.5**
HPC	84.0/81.9/**91.1***	**84.2/82.8**/88.5
Proxifier	86.4/87.9/85.7*	**93.1/99.1/89.8**
Spark	59.4/68.9/53.1†	**79.5/78.2/83.8**
Zookeeper	50.7/58.9/46.0†	**74.1/73.6/79.5**
average	67.8/75.8/64.9	**80.0/81.1/82.1**

* Scores as reported by [6].
† Scores as identified by us from running the framework manually.

Table 4.22: Mean ROUGE-1 F_1, precision and recall on the subdatasets of LogSummary.

	F_1 / precision / recall	
	simplified messages	**regular messages**
BGL	59.0/51.8/**76.9**	**67.8/68.4**/71.3
HDFS	69.3/67.6/**73.3**	**69.4/71.9**/67.9
HPC	75.6/71.7/**86.0**	**81.1/79.7**/85.4
Proxifier	**95.0**/97.8/**93.2**	92.2/**98.0**/89.1
Spark	**68.3**/64.5/**75.8**	67.0/**65.9**/70.5
Zookeeper	**62.7**/60.0/**68.6**	61.5/**61.0**/66.3
average	71.6/68.9/**79.0**	**73.1/74.1**/75.1

Table 4.23: Mean sentence-level ROUGE-L F_1, precision and recall of the fine-tuned BART-CNN models on the subdatasets of LogSummary.

4.6.3 Discussion

Evaluation of our text simplification during preprocessing First, we briefly examine how BART-CNN handles the log-data without using our simplifications; To our surprise the model performs well, even surpassing the average performance of our previous BART-CNN model using the simplified messages. The model operating on the raw messages manages to achieve higher precision, only slightly falling of on recall.

However, any direct comparisons should be taken with a grain of salt: The length penalty the hyperparameter search found for BART-CNN operating on the raw messages encourages shorter summaries than the one for the previous model, likely being a major cause in the precision/recall-tradeoff we observe.

Still, on the HPC dataset both models achieve almost the same recall, but the model operating on raw messages is more precise. At the very least, this indicates that there are situations where our simplification step is not helpful. Further still, at least on the LogSummary dataset, the simplification step is likely not a major factor in increasing a model's performance: After further fine-tuning, BART-CNN is able to summarize raw log messages well enough.

While this means that our simplification step may have been unnecessary, the possibility that pre-trained NLP models may not require a simplification step based on error-prone regular expressions is actually very promising.

The necessity of simplification may change on datasets including more complex patterns; we believe it to be likely that simplifying long patterns is still important, like java stack traces or long file-paths present in the HADOOP and TELCOAPP datasets. These passages are especially long and contain little important information, displacing more important context. Altogether though, our simplification step can likely be scaled down without negatively impacting model performance.

Comparison with previous work In general, the BART-CNN fine-tuned on raw messages performs comparatively well. On the BGL dataset, the F_1-scores are similar, but the summarization framework is more precise. The contrary is true on the HPC dataset, where BART-CNN is more precise. On all other remaining datasets, BART-CNN strictly outperforms the previous work by at least 7 points regarding the F_1 measure. Overall, we achieve improved ROUGE-1 F_1 scores of 12 points on average.

Of course, risk of overfitting is present when using deep learning methods. As the LogSummary framework makes use of unsupervised learning algorithms, it is not affected by this problem. However, the vocabulary overlap of summaries between training data and test data is the lowest for the LOGSUMMARY dataset across all studied datasets (see Table 4.1 on page 37). We do not believe the improved performance of BART-CNN can be explained solely on the basis of overfitting.

Still, we previously only reported the overall vocabulary overlap of summaries, but there may be substantial differences between the subdatasets of LOGSUMMARY. Thus we also report the overlap for each subdataset in Table 4.24.

BGL	HDFS	HPC	Proxifier	Spark	ZooKeeper
$\frac{17}{139} \approx 12.230\%$	$\frac{10}{25} = 40.000\%$	$\frac{26}{66} \approx 39.394\%$	$\frac{17}{22} \approx 77.273\%$	$\frac{20}{61} = 32.787\%$	$\frac{22}{72} \approx 30.556\%$

Table 4.24: Ratio of words common between summaries in the respective training and test sets for all subdatasets of LOGSUMMARY.

Here we see a more diverse picture: The BGL dataset certainly brings down the average vocabulary overlap between training and test set. As the summaries on HDFS and Proxifier are less varied in their vocabulary, both in relative and absolute terms, it is possible that BART-CNN overfits on these datasets. However, the same model still performs well on the other more diverse datasets, showing that it is capable of applying its knowledge even on inputs different from its training data.

As opposed to the LogSummary framework, which required a separate Log2Vec model for each subdataset to achieve peak performance, we trained BART-CNN on all six datasets simultaneously. Our results suggest that BART-CNN is applicable even in situations where logs are generated from heterogeneous subsystems with different logging styles.

Last but not least, if we compare ROUGE-1 to the sentence-level ROUGE-L for BART-CNN, we observe that the reduction in recall and precision is not drastic, both falling by only 7 points. This means that BART-CNN was able to mostly imitate the style and subjective structure of the human-written summaries. We speculate the model can thus also be adapted to other extractive summary styles wished for in practical applications.

All in all, we showed that pre-trained NLP models can effectively summarize the contents of log-data, even with minimal fine-tuning required. As a seq2seq architecture BART-CNN not only outperforms previous work, but is also more flexible by design.

4.7 Threats to Validity

As a conclusion to this chapter, we discuss some aspects regarding the validity of our results and conclusions.

Sensitivity to hyperparameters The hyperparameters of the beam-search greatly influence the performance of our models. Previous research confirms this finding, as Murray *et al.* show that machine translation systems are sensitive to the length penalty used and that different tasks and datasets require different penalties [56].

Finding optimal values is challenging; The differences observed between the transformer models in our experiments could in part be caused by suboptimal values for these generation parameters. We tried to mitigate this in our experiments by using automatic hyperparameter searches and consulting multiple perspectives (including examining perplexity, which is unaffected by these parameters) to select the best-performing models. Yet, we cannot fully exclude that some of our findings are influenced by suboptimal hyperparameters, especially when the differences between performances is small, as was the case for the BART variants in section 4.4 or when we examined the effects of pre-training in section 4.5.

Problems of ROUGE as a measure of quality Ever since the publication of the ROUGE metrics, several shortcomings have been pointed out by the academic community:

- ◇ ROUGE is unaware of synonyms and expects summaries to closely match the reference summaries [57].
- ◇ In general, ROUGE only judges the overlap in vocabulary and longer text segments, but cannot judge fluency and factual consistency. These qualities need to be judged separately, for instance during trials with human judges [32].
- ◇ Surveys with human judges suggest ROUGE may be better fit for evaluating extractive models than abstractive models. For past model architectures Kryściński *et al.* investigated the correlation of ROUGE-scores to human judgments of quality:
  When considering extractive summaries only, ROUGE-scores were moderately correlated to human judgments of relevance, fluency, comprehensibility and factual coherence; such correlations remain less clear-cut for abstractive summaries [32].

Since abstractive models create novel text sequences, they may have a higher tendency to introduce factual inconsistencies [32], which ROUGE cannot judge. However, from manual inspection of the generated summaries, we generally observe that the summaries closely follow an extractive style in our case, not including novel sequences of words. As our datasets are purely extractive, they encourage this behavior.

We believe this to be beneficial to the applicability of ROUGE to measure the quality of summaries on our datasets, since many of the problems described above are directed towards comparing texts that are syntactically different, but have the same semantic contents. As both the generated summaries and the reference summaries closely follow the style of the inputs, this should be less of a concern.

Biases in logs In the past, summarization models trained on news articles showed a significant bias for the first three sentences of an article. This has been attributed to a layout bias in journalism where the first parts of an article usually contain more important information [32]. Thus *Lead-3*, selecting the first three sentences of an article, performs well as a heuristical summary for news articles.

Lead-3 as a heuristic has next to no meaning in the context of log-data, because messages in logs roughly follow a chronological order and are not ordered by significance. However, there may be other biases in the data that could be used influence performance and should be accounted for. A few examples come to mind:

◇ The setup operations executed at the startup of a system are likely to produce less important log-entries. If a model received only inputs that started at the beginning of a log, it may learn to ignore the first share of its input.

  On the Hadoop dataset it is the case that some of the model inputs start at the beginning of a log, where the startup operations happen, however the dataset also contains many input documents that start at other portions of a log.

◇ Similarly, due to the way we selected the relevant portions of log-data we use as inputs on the TelcoApp dataset, some of the model inputs necessarily start with log messages that are deemed important by the summarization task based on common log events. In these cases it may be advisable to perturb the inputs by including additional log messages at the beginning.

◇ Each logging system may exhibit their own biases, with certain formats representing more important messages. Consider for example messages starting with `Diagnostics report from attempt` in the Hadoop dataset. As we saw in Table 4.16 on page 54 these usually represent anomalous messages, but not in all cases. Operators may find such messages useful in order to identify problems, but they may facilitate overfitting NLP models.

◇ While there are many lists of *stop words*, which represent words that are less likely to convey important semantic information, it is unclear if these apply well to application logs.[10] Words present in stop word lists are often ignored when evaluating a model using a metric. For example, ROUGE can optionally omit such words when evaluating the quality of a summary.

◇ Log-entries often contain metadata such as severity-levels (error, debug, . . .). We suspect errors and warnings may contain a significantly greater share of important information than the average log-entry. Related to this assumption, in the past system maintainers

[10] See `https://github.com/igorbrigadir/stopwords` for an overview of different stop word lists.

used basic keyword searches (e.g. searching for words like *failure* or *warning*) to identify problematic log-lines [58].

◇ Another potentially important metadata is the software component where the log message originated from. Some components will produce more important messages than others, because they play different roles in the operation of the system. It is possible that components issuing few messages are more likely to write important messages, because they represent software execution paths seldomly followed.

To avoid such biases and improve their generality, our models did not receive any metadata as inputs. Still, our models could still be susceptible to the other biases mentioned and further biases not considered in these examples.

On the whole, biases present in log-data are not well researched, even though they are an important aspect influencing the performance and generality of models. In other summarization datasets, like CNN/DailyMail and XSum, such biases are well-researched and accounted for during the design and evaluation of a model. Consider for instance the removal of duplicated trigrams employed by BART on the CNN/DailyMail dataset, which past research has shown to be beneficial [4, 45]: In other summarization datasets that contain repetitions (like BigPatent [41]) this may be detrimental to the model's performance, but it is actually helpful here.

Threat to universality One actuality we frequently observed during our experiments is that logs can be quite diverse:

◇ Logs originating from different systems may use dissimilar vocabularies and show less overlap than other previously researched summarization domains. As can be seen in Figure 4.1 on page 39, the datasets from the news domain show a much larger overlap in frequently used words than logs do. Further still, logs share little vocabulary with common summarization datasets in general.

◇ Logs vary in degree of correct use of capitalization and punctuation, meaning that there is no perfect choice for a separator between log messages concerning the pre-trained models investigated.

◇ As seen in Table 4.24 on Table 4.24, even by randomly sampling only 5 out of 100 examples for training, one can get widely different overlaps in vocabulary for training and test sets, on logs originating from different systems. Some systems show multifaceted use of different words, while others consist of only a few different log events, making it easy to accidentally overfit models on these datasets.

◇ Different summarization tasks are applicable to different log datasets. We presented the conditions in which we believe our summarization tasks to identify meaningful log events, that can be used as the basis for reference summaries, yet the nature of the resulting summaries is quite different. Summaries based on *common log events* are by nature quite homogeneous when only considering a single root cause, while summaries based on *non-normal log events* can be more diverse.

Both our proposed summarization tasks are *failure-oriented*, as they require a failure to be present to be applicable, but they make use of different assumptions inspired from the log datasets they were applied on.

◇ Even logs originating from the same system can be very different, as we observed on the Hadoop dataset: During some failures a group of message is periodically written to the log, leading to large log files, which consist of mostly redundant information. Other failures only produce small amounts of more diverse anomalous messages, reducing the overall redundancy of the log.

This diversity is part of what makes summarization on log-data challenging. Even worse, it means that our models and summarization tasks may not be applicable to logs from other systems.

Log summarization cannot be understood as a singular task, rather requirements for summaries will vary from system to system. Even for the human-written summaries in LogSummary which all follow a similar style, the model's performance varies between log segments from different systems. Our findings may not apply to other log datasets.

In practice, this means that a model should be fine-tuned on each specific dataset it is used on in order to achieve peak performance. Thankfully, tuning a pre-trained transformer model with only a few examples can be enough to improve performance and achieve useful results, as we have demonstrated in our experiments. Additionally, a single model may be able to handle logs from heterogeneous systems, as our results in the LogSummary dataset suggest.

Lack of user studies As the idea of textual log summarization is relatively young, none of the datasets we employ have previously been studied in the context of a user study. While Oliner *et al.* previously found that presenting a condensed representation of log-data to users can greatly speed up their ability to analyze said data [1], it is not possible for us to determine if the same is true for the summaries studied by us.

On the LogSummary dataset, the summaries are human-written, implying at least a basic degree of quality. However, on the other two datasets we employed our own summarization tasks, which construct reference summaries using semi-automatic means. While we presented several arguments why we believe our summarization tasks to produce informative summaries, it is still unclear whether system experts would find the resulting reference summaries valuable.

5

Related Work

Published research in the context of log summarization is sparse; however, more generally, log analysis is an active field of research, with literature presening and demonstrating the effectiveness of different approaches to gaining insights from logs.

Log analysis One fascinating application of log analysis is in the context of Computer Security, where it can present an important tool to detect and comprehend attacks on IT infrastructures. Systems such as HOLMES or HERCULE try to reconstruct the attack and detect different stages present in cyberattacks: Starting from the infiltration of the attacker into the system, the attacker gaining a foothold and escalating their privileges, up to the collection and extraction of sensitive information and potential cleanup operations.

The HOLMES [59] system detects ongoing attacks in real-time by correlating different system events that are extracted from logs, then using databases of common attack principles and causality rules to differentiate benign activities from an attack and present a high-level view of the attack in the form of a graph. For an event or action to be part of the attack, there must be dependencies between it and suspicious activities from other attack phases.

HERCULE [60] is based on the observation that attack-related events are highly correlated and dependent on each other but are usually not related to other benign activities. They build a graph of events, connecting log-entries through a multi-edge, denoting a set of predetermined binary features, like accesses to the same files and web resources, occurrence within a fixed time window, or requests directed at the same remote hosts. They then apply a machine learning algorithm to collapse the multi-edged graph to a weighted graph and use community detection to identify communities of related activities. From these, communities of log-entries containing suspicious activities are identified as potential attacks and presented to the user.

However, attack detection and reconstruction are only one possible use case where the investigation of log-data proves helpful. Log analysis is also widely-employed in IT operations more generally, not only related to attack detection and reconstruction [1]. Similar to the reconstruction of an attack tracing anomalous events, automated *root cause analysis* (RCA) tries to identify the root-cause of a failure and any intermediate steps leading up to the failure.

Fu *et al.* present a system which facilitates RCA in log-data. They identify sequences of log events that frequently occur in the log, cluster different sequences by the number of events they share, and construct a dependency graph of related log events by comparing the chronological order of the events in each cluster. Given a set of log events that indicate the presence of a failure, their approach identifies possible causes for these events by tracing the dependencies in the graph. Ultimately, this dependency graph is presented to a system expert, who can use the graph to perform an in-depth investigation of the root-cause [61].

All these systems have in common that they present an overview of the problem (as a graph of related events) to a human operator. This is done to communicate the details of the problem, speed up the understanding of the problem and potential solutions [59, 60], but also to facilitate further investigations [61]. These dependency graphs can be interpreted as *visual* summaries involving log-data of the studied problems.

We believe *textual* summarization models, such as the ones presented by us, could in the future enhance such visual summaries of log-data by further summarizing groups of log events, further lowering the effor required to swiftly form a genral understanding of the detected problem.

Transformer-based approaches to log analysis One of the first uses of transformer-based models for log analysis was presented by Nedelkoski, Bogatinovski, Acker, *et al.* with their NuLog and Logsy architectures. *NuLog* [63] represents a self-supervised algorithm for learning log templates and is hence a log parser. The key idea is to use *masked language modeling* (MLM) to mask random words in a log message and instruct a transformer-based language model to predict the missing word. If the model guesses the word correctly, it is not a dynamic parameter and hence part of the template. NuLog outperforms Drain [37] and other previously studied log parsers in parsing accuracy, but also provides a numeric representation (*embeddings*) for each log message.

These numeric embeddings encode a log message as a whole and thus represent the semantic content of each message. If combined with a supervised binary classifier, this can yield a model for log-based *anomaly detection*, a log analysis task where one tries to identify which log-entries are not the result of a normal execution environment [63]. Additionally, an unsupervised anomaly detection model is presented, which is based on the number of correctly predicted missing words in each log message: The assumption is that NuLog will not be able to predict the missing words in anomalous messages as well as in normal ones.

Likewise, anomaly detection is the task where the *Logsy* [62] model was applied to. The idea is to produce numerical embeddings where normal log messages are close to each other while anomalous log messages are more distant. AA spherical learning objective is used to train the model to place normal log messages close to the center of a sphere and other messages at large distances from this center point. This hypersphere is part of the same dimensional space as the embedding vectors, with the center point defined as 0. The embedding's Euclidean distance from the center is then used as the *anomaly score* of a log message. In combination with a threshold ε, this distance is used to classify the log message; Embeddings inside the hypersphere with radius ε are classified as normal while all other embeddings are not.

During training, the model is presented log messages from the target system it is later applied on, and auxiliary log messages originating from different openly accessible log datasets. The messages from the target system are assumed to be mostly normal, while the auxiliary messages are anomalous by definition, because they do not originate from that system. Logsy can thus be trained as a supervised classifier, although the data from the target system does not need detailed labeling. Overall, with unsupervised training on one-fifth of the log Logsy achieves a recall of 90% while maintaining a precision of 26% on the BGL dataset, which is the most challenging among the 3 different previously studied log datasets. With the inclusion of 2.1% labeled training data from the BGL target, the precision improves drastically to 89%, although the recall slightly decreases to 72%.

On the HADOOP-dataset our models could be interpreted as models for anomaly detection, though we did not evaluate them as such, because they generate arbitrary text. Hence we are not directly able to measure the overlap in terms of log-entries, although in practice the models usually reproduced the input log messages exactly, barring the correct separation between distinct log messages. However, from the examples we have seen with the full disk failure in the HADOOP dataset, even after supervised fine-tuning, our models may struggle to compete with the high precision and recall of Logsy.

Sequence-to-sequence (Seq2seq) models suffer from the problem that the expected outputs in an anomaly detection task can widely vary in length, making it challenging to find adequate hyperparameters. Though, these architectures still have an advantage over traditional classifiers, in that they are naturally able to consider the surrounding *context* of a message to judge if it is important or not, while classifiers usually work on a line-by-line basis. This makes it possible for seq2seq models to judge, if a log-entry is anomalous in a given context (e.g. it is anomalous for a network disconnection to happen during regular system operation, but it is a routine status indication when a device is reconnected to another network as a result of a user-request), however the usefulness of this depends on the granularity of the presented logs and the existence of appropriate datasets. If the log-data is too fine-grained and contains interleaving messages from widely different components, the context of a log message is not that meaningful, especially considering the input size limitations of current transformer models.

Liu, Tong, Xu, *et al.* undertake a study with different language models (including the transformer-based encoder BERT [3]) on log-data and use the model's internal embeddings for log anomaly detection [64].

They find that embeddings produced by models pre-trained on log-data are more useful for anomaly detection than when pre-trained on general-purpose text. However, they find that pre-training BERT on log-data alone does not consistently outperform the original BERT-model trained on large corpora of natural language texts. Furthermore, introducing more diverse log-data during pre-training can actually degrade the embeddings' quality.

As opposed to Liu *et al.*, we followed the advice from [43] and did not pre-train a model from scratch: Even after previous training on general-purpose text, we observe that further pre-training transformer models on log-data can lower the cross-entropy loss on downstream tasks. Unfortunately this did not improve the summarization performance in our applications.

Nevertheless, our results from fine-tuning alone, suggest that even general-purpose transformer models possess some knowledge transferable to the domain of log analysis.

Summarization of logs In the context of *visual* log summarization Locke, Li, Chen, *et al.* introduced their "log analysis IDE" *LogAssist* [2], which summarizes the information contained

in logs and displays a condensed but expandable view to the user, which is 75.2% to 93.9% smaller than the raw logs [2].
LogAssist constructs workflows from logs, which group multiple related log-entries matching program-flow. Logs are first parsed into log events using Drain [37], then grouped by an ID (e.g. a thread-ID) and time-difference. Finally, redundant event sequences are condensed using n-gram modeling [2]. During a user study with 19 participants, they find that "LogAssist provides, on average, a 42% improvement in log analysis speed when compared to performing the same analysis on raw logs alone" [2, pp. 10–11], thus demonstrating the need for effective tools to present log-data in a concise way to human operators.

Contrary to our approach, LogAssist does not aim to generate a textual summary that includes important information from the logs but instead aims to provide users with a compact representation of logs that can be easily explored. We believe the n-gram modeling used to simplify redundant event sequences in their work could also be effectively applied as a preprocessing step for textual summarization, helping to produce condensed summaries in situations where important log events are duplicated many times such as in the HADOOP-dataset.

Textual summarization of logs Meng, Zaiter, Huang, *et al.* apply existing concepts of *natural language processing* (NLP) in their open-access framework *LogSummary* [6] to create textual summaries of log-data and are to our knowledge the first to do so in the context of a scientific publication.

Furthermore they introduce *LogIE*, an information extraction framework for logs that is an integral part of LogSummary; LogIE combines log templates, a rule-based system and an information extraction framework *OpenIE* to extract entity-relationship tuples from log messages. [6]

The rule-based system first extracts entity-value pairs using commonly occurring patterns in log-data, such as separator characters like colons between key-value pairs. Hence it would extract the entity-relationship triples (`Severity`, `is`, `HIGH`) and (`Reason`, `is`, `NetworkException`) from the log message:

```
Error transferring transaction to PersistanceManager!
Will not retry transfer. Severity: HIGH, Reason=NetworkException
```

Now OpenIE is applied to the remaining part of the message to extract further tuples of entities, events and their relations, like (`Error`, `transferring`, `transaction`) and (`Will not retry`, `transfer`).

LogSummary then leverages previous techniques to produce vector embeddings for each tuple and ranks them using TextRank [65]; the top-k tuples are selected as the summary.

As we focus mainly on the application of pre-trained transformer models on log-data as opposed to creating a comprehensive framework for summarizing logs, LogSummary address some additional challenges with log summarization that we did not consider:

1. *Accurate summaries:* As LogSummary is entirely an extractive summarization framework, it does not suffer from some of the problems laid out in section 4.7; Our abstractive models possess the ability to generate arbitrary text not present in the input documents, but with that comes the challenge of keeping factual consistency.

LogSummary is less likely to write a factually inconsistent summary as summaries are entirely made up of text segments present in the input data.

2. *High throughput requirements:* Large-scale distributed systems can generate over 500 thousand [7, 8] or even 120 million [9] log-entries per hour. As such, depending on the use case, it may be necessary for log summarization methods to be able to process large amounts of log-data.

   While LogIE can process several thousand lines of logs per second on a server CPU, our models require several seconds or minutes to generate a summary for hundreds of log-entries on two deep-learning GPUs. According to benchmarks conducted by HuggingFace transformer-based models may even run between 13 to 23 times slower on a CPU.[1]

   Due to the self-attention mechanism, computation time scales quadratically with respect to input-size for transformer models [27]. Furthermore, the computation time of the beam-search employed for generating text from a model's predictions scales with respect to the length of the text to generate [44]. In situations where relations between distant log-entries is less important, throughput can hence be significantly improved by feeding models smaller portions of log-data at a time or asking for shorter summaries.

   Numerous efforts exist to make transformer-based more time-efficient, ranging from optimizations on CPUs halving computation times [66], using mixed precision for training and interference on GPUs benefiting from higher throughput of operations [67] or approximating the self-attention mechanism at the core of transformers in a computationally more efficient way [68, 69]. Moreover, previous research found that smaller models may be trained to achieve similar results as larger versions [70].

   On the other hand, LogSummary uses TextRank to rank triples, which actually also exhibits quadratic time complexity with respect to input size when used on a complete graph,[2] as is the case with LogSummary. We hypothesize that LogSummary also slows down when scaled up to longer input data.

   Ultimately, we do not expect transformer models to be faster than conventional algorithms in the near future.

3. *Prioritization of important messages:* Our proposed summarization tasks discourage from changeing the order of events in the summary. However, LogSummary explicitly approximates the importance of each summarized entry and thus is able to construct summaries beginning with the most critical segments first.

4. *Removing redundancy in log-data:* Logs often contain multiple repetitions of similar log messages; if one such message is part of the summary, other similar log messages may not be as important. Our proposed summarization tasks do not remove redundant log-messages from summaries.
   We notice this problem with the Hadoop-dataset, where the disconnection from the

[1] `https://medium.com/huggingface/benchmarking-transformers-pytorch-and-tensorflow-e2917fb891c2` (last accessed on 6th April 2022)

[2] A complete graph of n nodes is known to have $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ edges, while the PageRank-based ranking method is known to have a time complexity of $\mathcal{O}(n + e)$ for a graph with n nodes and e edges. It follows that $\mathcal{O}(n + \binom{n}{2}) = \mathcal{O}(\binom{n}{2}) = \mathcal{O}(n \cdot (n-1)) = \mathcal{O}(n^2)$ is the complexity on a complete graph.

network causes the repetition of the same group of log-entries over and over again, leading to highly repetitive and unnecessarily long summaries.

LogSummary addresses this by simply removing duplicated entity-relationship tuples, preventing repetition in summaries.

5. *Summarization within log messages:* By only extracting entity-relationship tuples from log-entries, LogSummary also manages to summarize the contents of log messages, which our own summarization tasks do not take into consideration.

   Though, as our results show, models fine-tuned on the LOGSUMMARY-dataset can also approximate the extraction of entity-relationship tuples present in their manual summaries, as exemplified in Table 4.15 on page 50, and thus summarize the contents of individual messages.

Last but not least, the team at Zebrium Inc. recently presented how they use OpenAI's transformer-based GPT-3 [71] model for textual summarization of root cause reports on their platform [72]. GPT-3 is a unidirectional language model with the objective of text completion, which has been scaled up 175 billion parameters; 10 times the size of any previous research models [71]. For comparison: PEGASUS-LARGE uses 568 million parameters [5].

Using their existent unsupervised methods Zebrium first generate a root cause report, typically including 5 to 20 log events. They then write a specialized prompt for GPT-3 (including log messages from the report) that encourages the model to predict an expert's "plain English" description of what happened. The intended use case is to simplify a root cause report for users and operators that may not be fully familiar with the system.

The examples they present are impressive in our opinion: Summaries show high fluency and are reported to be mostly factually consistent, despite not further fine-tuning GPT-3 and the model essentially performing zero-shot learning summarization.

As neither GPT-3 nor Zebrium's root cause reports are openly available, it is not possible for us to verify their results or to objectively contextualize their approach. Nonetheless, we believe their use of pre-trained transformer-based models for summarization of log-data in production-systems exemplifies that there is great potential for further research in the area of log summarization (including abstractive summarization) using NLP methods.

6 Conclusion

In this thesis, we proposed using pre-trained *natural language processing* (NLP) models to create textual summaries of logs. Our assemption is that summarizing logs can speed up the manual analysis of log data and that such models may be able to generalize to other tasks concerning log analysis. This idea of *log summarization* is relatively new; hence only few prior works have been conducted in this domain. Furthermore, we proposed two different ways to construct reference summaries for log datasets semi-automatically.

We conducted an extensive evaluation, testing different aspects that could influence the performance of models, and ultimately highlighted multiple challenges one faces when applying language models previously pre-trained on other domains, among which: The need to introduce adequate separators between individual log messages, the difference from other summarization domains, and the heterogeneous nature of systems and their logs. Factors like these make the problem of summarization different in every dataset.

Our results are mixed: While our best-performing models were able to outperform the previous state-of-the-art log summarization framework on manually written summaries, we believe our own proposed datasets require further improvements to be helpful for the evaluation of summarization approaches. Here, our models are not precise enough to identify many problematic parts of a log and significantly facilitate human operators' work.

We suggest that future work should not only concentrate on developing new approaches to summarize of logs; rather, an in-depth examination of a multitude of different log datasets is needed. As research in this area is very sparse, we are unaware of any comprehensive analyses regarding the biases present in log-data that one should consider when training NLP models, nor any user studies assessing the value of textual summaries.

Nevertheless, we believe our approach represents a successful proof of concept: To our knowledge, it is one of the first applications of abstractive sequence-to-sequence models in the domain of analyzing logs. We demonstrate that pre-trained models are able to quickly adapt to many log summarization datasets with the help of supervised fine-tuning. Modern sequence-to-sequence models can correctly reproduce log-data and are able to identify a part of the relevant information contained in logs.

Bibliography

[1] A. Oliner, A. Ganapathi, and W. Xu, “Advances and challenges in log analysis”, *Communications of the ACM*, vol. 55, no. 2, pp. 55–61, Feb. 2012, ISSN: 0001-0782. DOI: 10.1145/2076450.2076466.

[2] S. Locke, H. Li, T.-H. Chen, W. Shang, and W. Liu, “LogAssist: Assisting log analysis through log summarization”, *IEEE Transactions on Software Engineering (Early Access)*, pp. 1–15, May 2021. DOI: 10.1109/TSE.2021.3083715.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding”, in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, vol. 1, Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423.

[4] M. Lewis, Y. Liu, N. Goyal, *et al.*, “BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension”, in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, Jul. 2020, pp. 7871–7880. DOI: 10.18653/v1/2020.acl-main.703.

[5] J. Zhang, Y. Zhao, M. A. Saleh, and P. J. Liu, “PEGASUS: Pre-training with extracted gap-sentences for abstractive summarization”, in *Proceedings of the 37th International Conference on Machine Learning*, H. D. III and A. Singh, Eds., ser. Proceedings of Machine Learning Research, vol. 119, PMLR, Jul. 2020, pp. 11 328–11 339. [Online]. Available: http://proceedings.mlr.press/v119/zhang20ae.html.

[6] W. Meng, F. Zaiter, Y. Huang, *et al.*, “Summarizing unstructured logs in online services”, 2020. arXiv: 2012.08938 [cs.SE].

[7] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting large-scale system problems by mining console logs”, in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, New York, NY, USA: Association for Computing Machinery, 2009, pp. 117–131, ISBN: 9781605587523. DOI: 10.1145/1629575.1629587.

[8] J. Zhu, S. He, J. Liu, *et al.*, “Tools and benchmarks for automated log parsing”, in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2019, pp. 121–130. DOI: 10.1109/ICSE-SEIP.2019.00021.

[9] H. Mi, H. Wang, Y. Zhou, M. R. Lyu, and H. Cai, “Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1245–1255, 2013. DOI: 10.1109/TPDS.2013.21.

[10] A. Birolini, “Basic concepts, quality and reliability (RAMS) assurance of complex equipment and systems”, in *Reliability Engineering: Theory and Practice*, 7th ed. Springer Science & Business Media, 2013, ch. 1, pp. 1–24, ISBN: 978-3-642-39535-2. DOI: 10.1007/978-3-642-39535-2.

[11] J. Gray and D. P. Siewiorek, “High-availability computer systems”, *Computer*, vol. 24, no. 9, pp. 39–48, Sep. 1991. DOI: 10.1109/2.84898.

[12] H. Kopetz and P. Veríssimo, “Real time and dependability concepts”, in *Distributed Systems*, S. Mullender, Ed., 2nd ed. USA: ACM Press/Addison-Wesley Publishing Co., 1993, pp. 411–446, ISBN: 0201624273.

[13] J. J. Rooney and L. N. V. Heuvel, “Root cause analysis for beginners”, *Quality Progress*, vol. 37, no. 7, pp. 45–53, Jul. 2004.

[14] B. Aiello and L. Sachs, “DevOps”, in *Agile Application Lifecycle Management: Using DevOps to drive process improvement*, 1st ed. Addison-Wesley Professional, 2016, ch. 1, pp. 213–232, ISBN: 9780321774101.

[15] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, “DevOps”, *IEEE Software*, vol. 33, no. 3, pp. 94–100, 2016. DOI: 10.1109/MS.2016.68.

[16] L. Rijal, R. Colomo-Palacios, and M. Sánchez-Gordón, “AIOps: A multivocal literature review”, in *Artificial Intelligence for Cloud and Edge Computing*, S. Misra, A. K. Tyagi, V. Piuri, and L. Garg, Eds. Springer International Publishing, 2022, pp. 31–50, ISBN: 978-3-030-80821-1. DOI: 10.1007/978-3-030-80821-1_2.

[17] P. Notaro, J. Cardoso, and M. Gerndt, “A systematic mapping study in AIOps”, in *Service-Oriented Computing – ICSOC 2020 Workshops*, H. Hacid, F. Outay, H.-y. Paik, *et al.*, Eds., Springer International Publishing, 2021, pp. 110–123, ISBN: 978-3-030-76352-7. DOI: https://doi.org/10.1007/978-3-030-76352-7_15.

[18] Y. Dang, Q. Lin, and P. Huang, “AIOps: Real-world challenges and research innovations”, in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 4–5. DOI: 10.1109/ICSE-Companion.2019.00023.

[19] S. Vajjala, B. Majumder, A. Gupta, and H. Surana, “NLP: A primer”, in *Practical Natural Language Processing: A Comprehensive Guide to Building Real-World NLP Systems*, 1st ed. O’Reilly Media, 2020, ch. 1, pp. 1–36, ISBN: 9781492054054.

[20] A. Gulenko, A. Acker, O. Kao, and F. Liu, “AI-governance and levels of automation for AIOps-supported system administration”, in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, 2020. DOI: 10.1109/ICCCN49398.2020.9209606.

[21] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999, ISBN: 0262133601.

[22] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning”, *Nature*, vol. 521, pp. 436–444, May 2015. DOI: 10.1038/nature14539.

[23] M. E. Peters, M. Neumann, M. Iyyer, *et al.*, “Deep contextualized word representations”, Mar. 2018. arXiv: 1802.05365 [cs.CL].

[24] M. E. Peters, W. Ammar, C. Bhagavatula, and R. Power, “Semi-supervised sequence tagging with bidirectional language models”, in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, vol. 1, Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 1756–1765. DOI: 10.18653/v1/P17-1161.

[25] A. Jaiswal, A. R. Babu, M. Z. Zadeh, D. Banerjee, and F. Makedon, “A survey on contrastive self-supervised learning”, *Technologies*, vol. 9, no. 1, 2, 2021, ISSN: 2227-7080. DOI: 10.3390/technologies9010002.

[26] A. M. Dai and Q. V. Le, “Semi-supervised sequence learning”, in *Advances in Neural Information Processing Systems*, vol. 28, Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper/2015/hash/7137debd45ae4d0ab9aa953017286b20-Abstract.html.

[27] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, “Attention is all you need”, in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[28] T. Wolf, L. Debut, V. Sanh, *et al.*, “Transformers: State-of-the-art natural language processing”, in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Association for Computational Linguistics, Oct. 2020, pp. 38–45. DOI: 10.18653/v1/2020.emnlp-demos.6.

[29] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks”, in *Advances in Neural Information Processing Systems*, vol. 27, 2014, pp. 3104–3112. [Online]. Available: https://proceedings.neurips.cc/paper/2014/hash/a14ac55a4f27472c5d894ec1c3c743d2-Abstract.html.

[30] T. Kudo and J. Richardson, “SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing”, in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 66–71. DOI: 10.18653/v1/D18-2012.

[31] T. Kudo, “Subword regularization: Improving neural network translation models with multiple subword candidates”, in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, vol. 1, Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 66–75. DOI: 10.18653/v1/P18-1007.

[32] W. Kryściński, N. S. Keskar, B. McCann, C. Xiong, and R. Socher, “Neural text summarization: A critical evaluation”, in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 540–551. DOI: 10.18653/v1/D19-1051.

[33] Y. Liu and M. Lapata, “Text summarization with pretrained encoders”, in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3730–3740. DOI: 10.18653/v1/D19-1387.

[34] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, “An evaluation study on log parsing and its use in log mining”, in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016, pp. 654–661. DOI: 10.1109/DSN.2016.66.

[35] S. Narayan, S. B. Cohen, and M. Lapata, “Don’t give me the details, just the summary! Topic-aware convolutional neural networks for extreme summarization”, in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 1797–1807. DOI: 10.18653/v1/D18-1206.

[36] R. Nallapati, B. Zhou, C. Nogueira dos Santos, Ç. Gùlçehre, and B. Xiang, *Abstractive text summarization using sequence-to-sequence RNNs and beyond*, Berlin, Germany, Aug. 2016. DOI: 10.18653/v1/K16-1028.

[37] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, “Drain: An online log parsing approach with fixed depth tree”, in *2017 IEEE International Conference on Web Services (ICWS)*, 2017, pp. 33–40. DOI: 10.1109/ICWS.2017.13.

[38] Y. Liu, M. Ott, N. Goyal, *et al.*, “RoBERTa: A robustly optimized BERT pretraining approach”, Jul. 2019. arXiv: 1907.11692 [cs.CL].

[39] T. R. Goodwin, M. E. Savery, and D. Demner-Fushman, “Flight of the PEGASUS? Comparing transformers on few-shot and zero-shot multi-document abstractive summarization”, in *Proceedings of the 28th International Conference on Computational Linguistics*, International Committee on Computational Linguistics, Dec. 2020, pp. 5640–5646. DOI: 10.18653/v1/2020.coling-main.494.

[40] R. Zhang and J. Tetreault, “This email could save your life: Introducing the task of email subject line generation”, in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 446–456. DOI: 10.18653/v1/P19-1043.

[41] E. Sharma, C. Li, and W. Lu, “BIGPATENT: A large-scale dataset for abstractive and coherent summarization”, in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 2204–2213. DOI: 10.18653/v1/P19-1212.

[42] Q. Zhu, Y. Gu, L. Luo, *et al.*, “When does further pre-training MLM help? An empirical study on task-oriented dialog pre-training”, in *Proceedings of the Second Workshop on Insights from Negative Results in NLP*, Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 54–61. DOI: 10.18653/v1/2021.insights-1.9.

[43] S. Gururangan, A. Marasović, S. Swayamdipta, *et al.*, “Don’t stop pretraining: Adapt language models to domains and tasks”, in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, Jul. 2020, pp. 8342–8360. DOI: 10.18653/v1/2020.acl-main.740.

[44] A. Graves, “Sequence transduction with recurrent neural networks”, Nov. 2012. arXiv: 1211.3711 [cs.NE].

[45] R. Paulus, C. Xiong, and R. Socher, "A deep reinforced model for abstractive summarization", May 2017. arXiv: 1705.04304 [cs.CL].

[46] Y. Wu, M. Schuster, Z. Chen, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation", Sep. 2016. arXiv: 1609.08144 [cs.CL].

[47] M. Claesen and B. D. Moor, "Hyperparameter search in machine learning", Apr. 2015. arXiv: 1502.02127 [cs.LG].

[48] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework", in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 2623–2631, ISBN: 9781450362016. DOI: 10.1145/3292500.3330701.

[49] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries", in *Text Summarization Branches Out*, Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013.

[50] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems", in *Proceedings of the 38th International Conference on Software Engineering Companion*, New York, NY, USA: Association for Computing Machinery, 2016, pp. 102–111, ISBN: 9781450342056. DOI: 10.1145/2889160.2889232.

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

[52] LogPAI Team, *Loghub*, version 7, Zenodo, Sep. 2021. DOI: 10.5281/zenodo.3227177.

[53] H. Wickham and L. Stryjewski, "40 years of boxplots", had.co.nz, Tech. Rep., 2012.

[54] M. Buckland and F. Gey, "The relationship between recall and precision", *Journal of the American Society for Information Science*, vol. 45, no. 1, pp. 12–19, 1994. DOI: 10.1002/(SICI)1097-4571(199401)45:1<12::AID-ASI2>3.0.CO;2-L.

[55] W. Meng, Y. Liu, Y. Huang, *et al.*, "A semantic-aware representation framework for online log analysis", in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, 2020, pp. 1–7. DOI: 10.1109/ICCCN49398.2020.9209707.

[56] K. Murray and D. Chiang, "Correcting length bias in neural machine translation", Aug. 2018. arXiv: 1808.10006 [cs.CL].

[57] K. Ganesan, "Rouge 2.0: Updated and improved measures for evaluation of summarization tasks", Tech. Rep., 2018. arXiv: 1803.01937 [cs.IR].

[58] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. K. Flora, "Automatic identification of load testing problems", in *2008 IEEE International Conference on Software Maintenance*, Oct. 2008, pp. 307–316. DOI: 10.1109/ICSM.2008.4658079.

[59] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. N. Venkatakrishnan, *HOLMES: Real-time APT detection through correlation of suspicious information flows*, 2019. arXiv: 1810.01594 [cs.CR].

[60] K. Pei, Z. Gu, B. Saltaformaggio, *et al.*, "HERCULE: Attack story reconstruction via community discovery on correlated log graph", in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16, Los Angeles, California, USA: Association for Computing Machinery, 2016, pp. 583–595, ISBN: 9781450347716. DOI: 10.1145/2991079.2991122.

[61] X. Fu, R. Ren, S. A. McKee, J. Zhan, and N. Sun, "Digging deeper into cluster system logs for failure prediction and root cause diagnosis", in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, 2014, pp. 103–112. DOI: 10.1109/CLUSTER.2014.6968768.

[62] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, "Self-attentive classification-based anomaly detection in unstructured logs", in *2020 IEEE International Conference on Data Mining (ICDM)*, 2020, pp. 1196–1201. DOI: 10.1109/ICDM50108.2020.00148.

[63] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, *Self-supervised log parsing*, 2020. arXiv: 2003.07905 [cs.LG].

[64] X. Liu, Y. Tong, A. Xu, and R. Akkiraju, “Using language models to pre-train features for optimizing information technology operations management tasks”, in *Service-Oriented Computing – ICSOC 2020 Workshops*, H. Hacid, F. Outay, H.-y. Paik, *et al.*, Eds., Springer International Publishing, 2021, pp. 150–161, ISBN: 978-3-030-76352-7.

[65] R. Mihalcea and P. Tarau, “TextRank: Bringing order into text”, in *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 404–411. [Online]. Available: `https://aclanthology.org/W04-3252`.

[66] A. Kogan and D. Dice, “Optimizing inference performance of transformers on CPUs”, Feb. 2021. arXiv: `2102.06621 [cs.CL]`.

[67] NVIDIA. “Train with mixed precision”. (Mar. 2022), [Online]. Available: `https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html` (visited on 04/06/2022).

[68] M. Zaheer, G. Guruganesh, A. Dubey, *et al.*, *Big bird: Transformers for longer sequences*, 2021. arXiv: `2007.14062 [cs.LG]`.

[69] Y. Xiong, Z. Zeng, R. Chakraborty, *et al.*, “Nyströmformer: A nystöm-based algorithm for approximating self-attention”, *Proc. Conf. AAAI Artif. Intell.*, vol. 35, no. 16, pp. 14 138–14 148, May 2021.

[70] S. Shleifer and A. M. Rush, “Pre-trained summarization distillation”, Oct. 2020. arXiv: `2010.13002 [cs.CL]`.

[71] T. Brown, B. Mann, N. Ryder, *et al.*, “Language models are few-shot learners”, in *Advances in Neural Information Processing Systems*, vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: `https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html`.

[72] L. Lancaster. “Using GPT-3 for plain language incident root cause from logs”. (Jan. 2021), [Online]. Available: `https://www.zebrium.com/blog/using-gpt-3-with-zebrium-for-plain-language-incident-root-cause-from-logs` (visited on 03/19/2022).