h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

Fachbereich Informatik & Fachbereich
Mathematik und Naturwissenschaften

Automatisierte Fehlererkennung einer Microservice-Anwendung basierend auf Log-Dateien

Abschlussarbeit zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

im Studiengang Data Science

vorgelegt von

Valerie Restat

Matrikelnummer: 741423

Referentin	:	Prof. Dr. Uta Störl
Korreferentin	:	Prof. Dr. Antje Jahn

Ausgabedatum	:	31.08.2020
Abgabedatum	:	12.02.2021

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, 12. Februar 2021

Valerie Restat

ZUSAMMENFASSUNG

Log-Dateien dokumentieren das Verhalten einer Anwendung oder eines Systems, was deren Analyse zu einem Schlüsselfaktor für die Sicherheit, Stabilität und Nutzbarkeit eines Systems macht. Dabei gibt es in vielen Applikationen in den Log-Dateien Fehler oder Warnungen, bei denen kein Handlungsbedarf besteht. Die hohe Anzahl solcher Meldungen im Vergleich zur geringen Menge an tatsächlichen Fehlern führt dazu, dass sich die Analyse sehr aufwändig gestaltet. Diese Problemstellung findet sich auch bei der ORDIX AG bei einem Mircoservice für interne Zwecke wieder. Ziel dieser Arbeit war deshalb die automatisierte Fehlererkennung basierend auf den Log-Dateien.

Zu diesem Zweck musste zunächst eine geeignete Methode zur Vorverarbeitung entwickelt werden, um aus den unstrukturierten Log-Dateien Features zu gewinnen, die für den Einsatz von Machine-Learning-Modellen zur Fehlererkennung verwendet werden können. Hierfür wurde aus einer Log-Meldung der konstante Teil, der sogenannte Log-Key, extrahiert.

Anschließend wurde die Fehlererkennung betrachtet und zu diesem Zweck diverse Supervised-, Semi-Supervised- und Unsupervised-Methoden untersucht. Dabei hat sich gezeigt, dass die Unsupervised-Verfahren nicht für den Anwendungsfall dieser Arbeit geeignet sind, sondern in erster Linie für Anwendungen, in denen die Daten mittels numerischer Werte verglichen und so Fehler identifiziert werden können. Im Rahmen dieser Arbeit war dies jedoch nicht gegeben, weshalb jene Verfahren in der Umsetzung nicht berücksichtigt wurden.

Für die Evaluierung wurden die nicht erkannten Fehler sowie die Meldungen, die inkorrekterweise als Fehler klassifiziert wurden, berücksichtigt. Die Supervised-Verfahren erzielten sehr gute Ergebnisse, insbesondere durch den Einsatz des Decision Tree konnten fast alle Meldungen korrekt klassifziert werden. Im Bereich der Semi-Supervised-Fehlererkennung konnten mittels eines Long Short-Term Memory alle Fehler erkannt werden. Das Modell wies jedoch, im Vergleich zu den Supervised-Methoden, eine höhere Anzahl an Meldungen auf, die inkorrekt als Fehler klassifiziert wurden.

Es wurde gezeigt, dass sowohl der Supervised- als auch der Semi-Supervised-Ansatz für eine automatisierte Fehlererkennung auf Basis der Log-Dateien des Microservice eingesetzt werden kann. Bei den Modellen des Supervised Learnings besteht der Aufwand dabei in der manuellen Vergabe der Labels, während er bei der Semi-Supervised-Fehlererkennung in der manuellen Analyse der Modellergebnisse liegt.

ABSTRACT

Log files protocol a system's behavior and are therefore a valuable source of information about the security, stability, and usability of a system. Nevertheless, many applications display errors or warnings in their log files, even though there no action is required. The large number of such messages compared to the few actual errors leads to a very time-consuming need of manual error message screening. This problem is also found at the ORDIX AG at a microservice application. For this reason, the goal of this thesis was an automated error detection based on the log files.

For this purpose a suitable preprocessing method was first needed to extract features from the unstructured logs to then be used for machine learning models to detect the errors. Thus, the log key, which is the constant part of a log message, was extracted.

For the error detection various methods for supervised, semi-supervised and unsupervised error detection were examined. However, unsupervised models proved to be unsuitable for the context of this work since they primarily rely on analysis of numerical values. As this was not given in the context of this work, those procedures were not considered for implementation.

For the evaluation, the number of detected errors and the messages that were incorrectly classified as errors were taken into account. The supervised methods achieve very good results, in particular the decision tree, which classified almost all messages correctly. In the area of semi-supervised anomaly detection, all errors could be detected using a long short-term memory, but a larger number of messages was incorrectly classified as errors compared to the supervised methods.

It has been shown that both, the supervised and the semi-supervised approach, are suitable for automated error detection based on the microservice's log files. For the supervised models, the effort is in the manual labelling, while for the semi-supervised error detection, it is in the manual analysis of the model results.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

ABKÜRZUNGSVERZEICHNIS

BSG Basic Signature Generation

CNN Convolutional Neural Network

FN False Negative

FP False Positive

IPLoM Iterative Partitioning Log Mining

LCS Longest Common Subsequence

LSTM Long Short-Term Memory

NLP Natural Language Processing

PCA Principal Component Analysis

TN True Negative

TP True Positive

Spell Streaming structured Parser for Event Logs using LCS

SVM Support Vector Machine

RNN Recurrent Neural Network

Teil I

THESIS

1 EINLEITUNG

1.1 MOTIVATION

In Log-Dateien wird das Verhalten einer Anwendung oder eines Systems dokumentiert, was deren Analyse zu einem Schlüsselfaktor für die Sicherheit, Stabilität und Nutzbarkeit macht. Dabei gibt es in vielen Applikationen in den Log-Dateien Fehler oder Warnungen, bei denen kein Handlungsbedarf besteht. Die hohe Anzahl solcher Meldungen im Vergleich zur geringen Menge an tatsächlichen Fehlern führt dazu, dass sich die Analyse sehr aufwändig gestaltet und viel Zeit kostet. Bei gekaufter Software oder Open-Source-Software entfällt, im Gegensatz zur Eigenentwicklung, zudem der Einfluss auf die Meldungen.

Diese Ausgangssituation findet sich auch bei der ORDIX AG wieder. Dort werden für interne Anwendungen Microservices eingesetzt, deren Entwickler*innen die Aufgabe haben, die Log-Dateien regelmäßig zu prüfen, um Fehler oder falsches Programmverhalten zu identifizieren. Dabei ist aufgefallen, dass es viele Fehler gibt, die „falsche“ Fehler sind. Dies sind beispielsweise Log-Zeilen mit Loglevel *ERROR*, bei denen jedoch kein Handlungsbedarf besteht. Dies führt auch hier dazu, dass sich die Analyse sehr aufwändig gestaltet und die Entwickler*innen viel Zeit kostet. Aus diesem Grund soll das Auffinden von „echten“ Fehlern, also Fehlern mit Handlungsbedarf, automatisiert werden.

Eine Automatisierung wird allerdings erschwert durch die meist unstrukturierten Log-Dateien. Log-Zeilen werden überwiegend mittels Print-Statements im Sourcecode erzeugt. Auch wenn diese eine grobe Struktur aufweisen, beispielsweise immer beginnend mit einem Timestamp, besteht der Großteil der Nachricht aus unstrukturiertem Text. Dieser variiert je nach Anwendung und Entwickler*in, weshalb Log-Dateien je nach Kontext eine hohe Heterogenität aufweisen.

Eine weitere Herausforderung stellt die Unausgeglichenheit der Daten dar. Im Normalfall treten bei einer Anwendung nur wenige Fehler auf. Die Menge der fehlerfreien Daten ist somit wesentlich größer als die der Fehler. Dieser Umstand wird als *Imbalanced Data* bezeichnet und erschwert die Fehlersuche zusätzlich. Des Weiteren sind, wie beschrieben, nicht alle Meldungen, die als Fehler gekennzeichnet sind, echte Fehler. Gleichzeitig können auch vermeintlich unkritische Meldungen auf ein Fehlverhalten des Systems hindeuten. Diese Umstände und die zunehmende Menge an Daten führt dazu, dass eine manuelle Analyse der Dateien mit einem hohen Zeitaufwand verbunden ist und ein ausgeprägtes Expert*innenwissen erfordert. Eine reine regelbasierte Suche auf Basis des Loglevels ist meist nicht ausreichend. [MP08]

Aus diesem Grund wird in dieser Arbeit untersucht, wie eine Automatisierung mithilfe von Machine-Learning-Methoden möglich ist, und analysiert, welche Verfahren für die beschriebene Problemstellung gut geeignet sind.

1.2 ZIEL DER ARBEIT

Ziel dieser Arbeit ist die automatisierte Fehlererkennung einer Microservice-Anwendung basierend auf den Log-Dateien.

Von einem Mircoservice wurden die Log-Dateien eines Monats zur Verfügung gestellt (Anhang A.1). Auf den Sourcecode kann hingegen nicht zugegriffen werden. Die Daten wurden von einem Entwickler mit Labels versehen, dabei wurden die echten Fehler entsprechend markiert. In der Praxis sind meist keine ausreichenden Labels vorhanden, durch die manuelle Vergabe können in dieser Arbeit jedoch Supervised-Vefahren, die diese Labels benötigen, und Unsupervised-Verfahren, die keine Labels benötigen, gegenübergestellt werden. Für die Unsupervised-Verfahren werden die Labels für die Evaluation verwendet. Auch ein Semi-Supervised-Verfahren, bei dem ein Modell nur auf fehlerfreien Daten trainiert wird, ist mit den Daten möglich.

Um ein Machine-Learning-Modell zur Fehlererkennung zu entwickeln, muss zunächst eine Vorverarbeitung der Daten erfolgen, das sogenannte Log-Parsing. Die Log-Dateien liegen in unstrukturierter Textform vor und weisen eine hohe Heterogenität auf. Aus diesem Grund muss zunächst die Fragestellung betrachtet werden, welches Log-Parsing-Verfahren am besten geeignet ist.

Auf den daraus gewonnenen Features kann im nächsten Schritt ein Modell zur Fehlererkennung trainiert werden. Sowohl bei der Auswahl des Modells als auch bei den Metriken zur Beurteilung der Vorhersagegüte müssen die Besonderheiten von Imbalanced Data, also dem Ungleichgewicht der Daten, beachtet werden, da die Menge echter Fehler gering ist. Hieraus ergibt sich die Fragestellung, wie ein Modell den genannten Herausforderungen gerecht werden kann.

Aufgrund der vorhandenen Labels können Supervised-, Unsupervised- und Semi-Supervised-Modelle verglichen werden. Es wird deshalb die Fragestellung betrachtet, ob der Aufwand zur Vergabe von Labels, die in der Praxis meist nicht vorliegen, bei den Supervised-Modellen in einer besseren Vorhersagegüte resultiert.

Somit ergeben sich drei Fragestellungen, die an dieser Stelle noch einmal zusammengefasst werden:

1. Welches Log-Parsing-Verfahren ist für die Daten am besten geeignet?
2. Welches Modell ist für die automatisierte Fehlererkennung des Microservice am besten geeignet?
3. Spiegelt sich der Aufwand zur Vergabe von Labels in der Vorhersagegüte der Supervised-Modelle wider?

Dahingegen zielt diese Arbeit nicht darauf ab, ein generisches Framework zur automatisierten Fehlererkennung für diverse Anwendungsgebiete zu

entwickeln. Es wird konkret der Anwendungsfall des Microservice betrachtet. Die Verfahren können jedoch für ähnliche Systeme adaptiert werden. Zudem erfolgt eine Diskussion, welche Methoden für welchen Anwendungsfall geeignet sind.

1.3 AUFBAU DER ARBEIT

Zu Beginn werden in Kapitel 2 die Grundlagen erläutert, die zum Verständnis dieser Arbeit notwendig sind. In Kapitel 3 werden verwandte Arbeiten vorgestellt. Anschließend werden in Kapitel 4 zunächst die zugrundeliegende Daten und die Anforderungen des Systems analysiert. Daraufhin wird das Log-Parsing betrachtet. Hierfür werden die gängigen Methoden, wie beispielsweise Spell [DL17], untersucht und eine passende Vorgehensweise konzipiert, um aus einer rein textuellen Form Features zu erzeugen. Diese können dann für das Trainieren der Modelle verwendet werden. Hierbei werden unterschiedliche Verfahren erarbeitet, die die oben genannten Fragestellungen adressieren. Abschließend erfolgt in Kapitel 6 eine Zusammenfassung der Ergebnisse dieser Arbeit sowie ein Ausblick auf mögliche aufbauende Aspekte, die in zukünftigen Arbeiten untersucht werden könnten.

2 GRUNDLAGEN

In diesem Kapitel werden die Grundlagen zu Log-Dateien vorgestellt sowie grundlegende Prinzipien von Machine Learning erläutert.

2.1 LOG-DATEIEN

Log-Dateien dokumentieren das Verhalten einer Anwendung oder eines Systems, was die Analyse dieser Dateien zu einem Schlüsselfaktor für Sicherheit, Stabilität und Nutzbarkeit macht. [Guo+19] Die Relevanz von Logging zeigt sich auch durch die Verwendung in diversen Aufgaben des Softwaresystem-Managements, wie z.B. Anomaliedetektion, Fehler-Diagnose, Performance-Analyse und weiteren. [Fu+14] Durch eine stets wachsende Systemlandschaft nimmt auch die Menge an Log-Dateien weiter zu. Dies führt dazu, dass das Durchsuchen und Analysieren zu einer aufwändigen und fehlerbehafteten Aufgabe geworden ist. Die automatisierte Analyse von Log-Dateien hat sich deshalb zu einem wichtigen Aspekt entwickelt, der auch in der Forschung und Literatur zunehmend an Bedeutung gewinnt. [MZHM09]

2.1.1 *Allgemein*

System-Logs können Millionen von Zeilen enthalten, die jeweils aus einem Text bestehen, der mit einem Print- oder Log-Statement im Sourcecode korrespondiert. [Guo+19]. Diverse Bibliotheken bieten Interfaces zur Unterstützung und Vereinfachung von Logging, trotzdem bleibt es die Entscheidung des Entwicklers/der Entwicklerin, was, wann und wo geloggt wird. [SKK12]

Log-Anweisungen entsprechen meist dem folgenden Muster [Fu+14]:

Logger.log(Loglevel, "Log-Meldung %s", Parameter) (2.1)

Im Folgenden werden exemplarisch ein Log-Statement (a) sowie zwei zugehörige Log-Einträge (b, c) gezeigt.

(a) Logger.log(info, „Image saved in %s seconds“, sec)
(b) 2020-06-10 07:00:00 INFO Image saved in 2 seconds
(c) 2020-06-10 07:00:00 INFO Image saved in 3 seconds (2.2)

Wie zu erkennen, setzt sich ein Log-Eintrag aus zwei Teilen zusammen, den Metadaten (*2020-06-10 07:00:00 INFO*) und der Log-Meldung (*Image saved in * seconds*). Die Metadaten enthalten diverse Informationen, wie Datum und Uhrzeit, den Loglevel oder auch Informationen zur Klasse oder Funktion, in der die Meldung erzeugt wurde. Dahingegen bestehen die Log-Meldungen

aus einem konstanten Text und einem oder mehreren variablen Parameter. [YPZ12] Durch Event-Extrahierung, wie z.B. Log-Parsing, werden die unstrukturierten Log-Zeilen in strukturierte Events eines bestimmten Formats konvertiert, auf denen anschließend Analysen durchgeführt werden können. [Guo+19] Das Log-Parsing wird im Folgenden genauer erläutert.

2.1.2 *Log-Parsing*

Log-Parsing kann in zwei Bestandteile aufgeteilt werden: Das Parsen der gesamten Logzeile, zur Trennung von Metadaten und Meldung, und das Parsen der Meldung, zur Trennung des konstanten und des variablen Teils. Alle Meldungen, die vom gleichen Print-Statement erzeugt wurden, besitzen den gleichen konstanten Text, auch genannt Log-Typ [DL17], Log-Key [Fu+09] [Du+17], Signatur [Guo+19], Muster [MZHM09] oder Event-Typ [Lin+16] [MP08]. Im Folgenden wird der Begriff Log-Key verwendet. Beispielsweise besitzen die Log-Einträge in Beispiel 2.2 den gleichen Log-Key (*Image saved in * seconds*). [Fu+09]

Ein häufig gewählter Ansatz, um unstrukturierten Text in ein vorgegebenes Format zu konvertieren, ist das Parsen mittels regulärer Ausdrücke, die zuvor manuell definiert werden müssen, wie in 2.3 veranschaulicht. [Fu+09]

Log-Message:	*Image saved in 2 seconds*	
Regulärer Ausdruck:	*Image saved in ([0-9]+) seconds*	
Log-Key:	*Image saved in * seconds*	
Parameter:	*2*	(2.3)

Ohne ein umfangreiches Domain-Wissen können jedoch nicht alle möglichen Muster abgebildet werden. Aufgrund der Tatsache, dass die Meldungen aus freiem Text bestehen, ist es nicht trivial, die Parameter zu identifizieren und somit den Log-Key zu ermitteln. Wenn nicht auf den Sourcecode oder Expert*innenwissen zugegriffen werden kann, sind sowohl Log-Key als auch die Position der Parameter im Text unbekannt. [Fu+09]. Zudem können die regulären Ausdrücke nur auf die spezifische Applikation angewandt werden, für die sie entwickelt wurden. Um diese Einschränkung zu umgehen, existieren Log-Parsing-Methoden, die auf String-Vergleichen basieren. [DL17] Ziel dieser Methoden ist die Identifizierung des Log-Leys anhand von identischen Substrings. Hierfür werden Metriken zur Bestimmung der Ähnlichkeit von Strings eingesetzt, von denen in der Literatur unterschiedliche Varianten genutzt werden. Nachfolgend werden die beiden Metriken erläutert, die am meisten verbreitet sind.

LONGEST COMMON SUBSEQUENCE: Ziel der *Longest Common Subsequence (LCS)* ist es, unter Berücksichtigung der Reihenfolge die maximale Anzahl gleicher Symbole zu finden. [BHR00]

Sei $\sum$ ein Alphabet und $A = \{a_1, ..., a_m\}$ eine Sequenz, sodass $a_i \in \sum$, dann ist eine Subsequenz von A definiert als $\{a_{x1}, ..., a_{xk}\}$, wobei $\forall x_i$:

$x_i \in \mathbb{Z}^+$ und $1 \leq x_1 \leq ... \leq x_k \leq m$. Ferner sei $B = \{b_1, ..., b_n\}$ eine Sequenz, sodass $b_j \in \sum$. Ist die Subsequenz γ sowohl in A als auch in B enthalten, ist sie eine gemeinsame Subsequenz von A und B. Das LCS-Problem besteht darin, die längste mögliche Subsequenz γ zu finden. [DL17]
Zur Lösung des Problems existieren mehrere Varianten, Formel 2.4 zeigt das Basis-Konzept der Berechnung der LCS L für zwei Strings $A = \{a_1, ..., a_m\}$ und $B = \{b_1, ..., b_n\}$. [BHR00]

$$L(a_i, b_j) = \begin{cases} 0 & \text{if } a_i = 0 \text{ or } b_j = 0 \\ L(a_{i-1}, b_{j-1}) + 1 & \text{if } a_i = b_j \\ max\{L(a_{i-1}, b_j), L(a_i, b_{j-1})\} & \text{if } a_i \neq b_j \end{cases} \tag{2.4}$$

A_0 bzw. B_0 beschreibt dabei die leere Menge. Der Algorithmus wird in Tabelle 2.1 exemplarisch für die Sequenzen $A = abcd$ und $B = acde$ ausgeführt, die längste gemeinsame Subsequenz ist acd mit der Länge 3.

	∅	**a**	**b**	**c**	**d**
∅	0	0	0	0	0
a	0	1	1	1	1
c	0	1	1	2	2
d	0	1	1	2	3
e	0	1	1	2	3

Tabelle 2.1: Beispiel des LCS-Algorithmus

LEVENSHTEIN-DISTANZ: Die Levenshtein-Distanz, teilweise auch Edit Distance genannt, ist eine Metrik zur Berechnung der Differenz zweier Sequenzen. Sie gibt die minimale Anzahl an Operationen an, die notwendig sind, um Sequenz 1 in Sequenz 2 zu transformieren. Erlaubte Operationen sind das Einfügen, Ersetzen und Löschen eines Zeichens. [MVM09]
Für zwei Sequenzen $A = \{a_1, ..., a_m\}$ und $B = \{b_1, ..., b_n\}$ wird die Distanz D wie in Formel 2.5 berechnet [SM01].

$$D(a_i, b_j) = \begin{cases} \max(a_i, b_j) & \text{if } \min(a_i, b_j) = 0 \\ \min \begin{cases} D(a_{i-1}, b_j) + 1 \\ D(a_i, b_{j-1}) + 1 \\ D(a_{i-1}, b_{j-1}) + 1_{a_i \neq b_j} \end{cases} & \text{otherwise} \end{cases} \tag{2.5}$$

Tabelle 2.2 zeigt die exemplarische Anwendung des Algorithmus für die Sequenzen $A = abcd$ und $B = acde$. Die Levenshtein-Distanz beträgt 2.

	a	**b**	**c**	**d**
a	0	1	2	3
c	1	1	1	2
d	2	2	2	1
e	3	3	3	2

Tabelle 2.2: Beispiel zur Berechnung der Levenshtein-Distanz

Diese beiden beschriebenen Metriken bilden die Grundlage für viele Methoden des automatisierten Log-Parsings von Meldungen. In der Literatur existieren unterschiedliche Verfahren, die in zwei nachfolgend beschriebene Gruppen unterteilt werden können.

OFFLINE-LOG-PARSING: Alle vorhandenen Logs werden in einem Batch-Prozess verarbeitet, wie beispielsweise bei den Algorithmen IPLoM (*Iterative Partitioning Log Mining*) [MZHM09] und LogSig [TLP11].

ONLINE-LOG-PARSING: Die einzelnen Meldungen werden im Streaming-Verfahren analysiert, ein Offline-Training ist nicht erforderlich. Spell (*Streaming structured Parser for Event Logs using LCS*) [DL17], BSG (*Basic Signature Generation*) [Guo+19] und der Online Dictionary Creation Algorithm [Aha+09] sind Verfahren, die hier zu nennen sind.

Im Folgenden werden die oben genannten Log-Parsing-Methoden genauer erläutert:

IPLoM

IPLoM steht für *Iterative Partitioning Log Mining* und besteht aus vier Schritten. [MZHM09]

1. PARTITIONIERUNG MITTELS TOKEN COUNT: Es wird die Annahme getroffen, dass alle Log-Meldungen des gleichen Log-Keys auch die gleiche Länge (Anzahl Token/Wörter) besitzen. Deshalb werden die Meldungen zunächst nach Anzahl der Tokens sortiert. [MZHM09]

2. PARTITIONIERUNG MITTELS TOKEN-POSITION: Nach dem vorherigen Schritt hat jede Partition die gleiche Anzahl an Token und die einzelnen Zeilen können deshalb als n-Tuples betrachtet werden, wobei n die Anzahl Token darstellt. Es wird die Annahme getroffen, dass die Spalte mit den wenigsten unterschiedlichen Wörtern als konstant betrachtet werden kann und somit Bestandteil des Log-Keys sind. [MZHM09] Abbildung 2.1 stellt diesen Schritt dar.

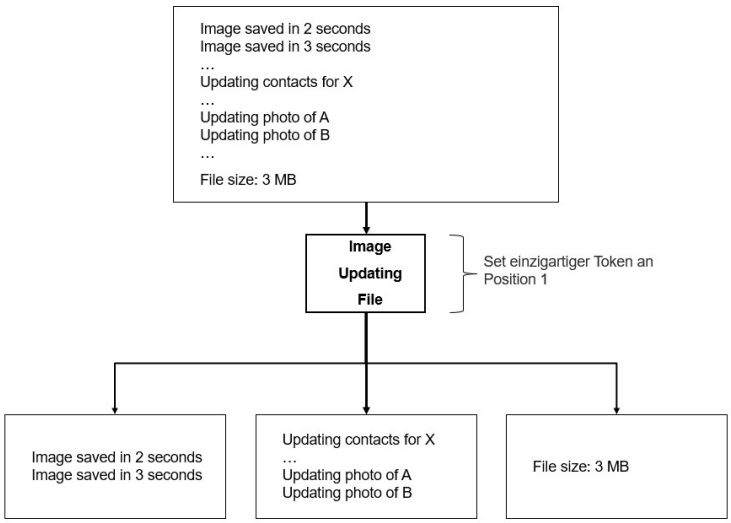


Abbildung 2.1: IPLoM - Schritt 2 (nach [MZHM09])

3. PARTITIONIERUNG MITTELS DER SUCHE NACH BIJEKTIONEN: In diesem Schritt, dem letzten der Partitionierung, wird nach bijektiven Beziehungen zwischen einem Set von einzigartigen Token in zwei Positionen gesucht. Die ersten beiden Token-Positionen mit der am häufigsten vorkommenden Anzahl größer als Eins werden ausgewählt, wie in Abbildung 2.2 dargestellt. Die Token *photo* und *of* haben eine 1-1-

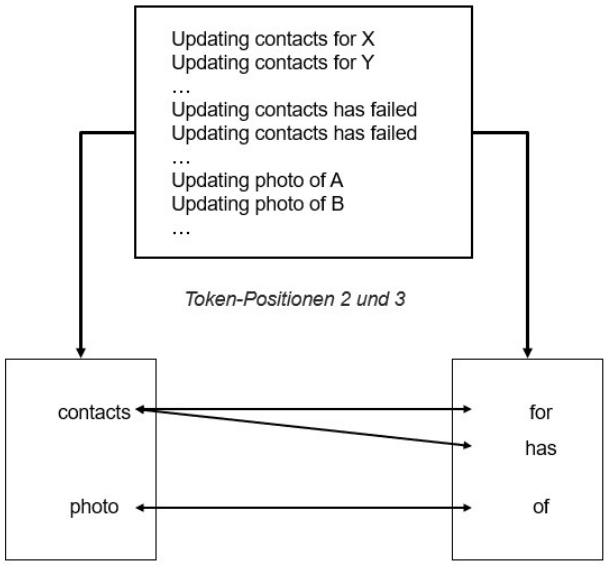


Abbildung 2.2: IPLoM - Schritt 3 (nach [MZHM09])

Beziehung, da in allen Zeilen, bei denen sich an Position 2 das Wort *photo* befindet, *of* an Position 3 steht und umgekehrt. Dahingegen be-

sitzt *contacts* ein 1-M-Beziehung mit den Token *for* und *has*. Exisitiert ein bijektive Beziehung zwischen zwei Elementen, werden sie einer neuen Partition zugeordnet. Im Falle einer 1-M- und M-1-Beziehung kann die M-Seite sowohl Parameter repräsentieren als auch konstante Werte, die zu unterschiedlichen Log-Keys korrespondieren. Die Entscheidung wird anhand des Verhältnisses zwischen einzigartigen Werten der Token-Position und der Anzahl Zeilen, die diese Werte enthalten, getroffen. [MZHM09]

4. SUCHE NACH CLUSTER-BESCHREIBUNG/FORMATIERUNG DER ZEILEN: Im Anschluss an die Partitionierung wird die Annahme getroffen, dass jede Partition ein Cluster repräsentiert und somit alle Nachrichten einer Partition den gleichen Log-Key besitzen. Hierfür wird wiederum die Anzahl einzigartiger Tokens pro Position gezählt. Gibt es nur einen Wert an einer Position, wird dieser als konstanter Wert und somit als Teil des Log-Keys betrachtet. Andernfalls wird die Position durch eine Wildcard ersetzt. [MZHM09] Abbildung 2.3 beschreibt das Vorgehen.

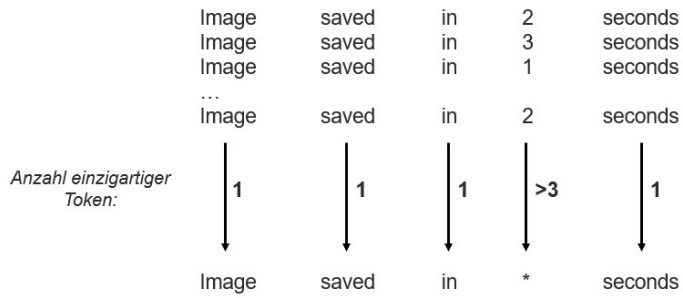


Abbildung 2.3: IPLoM - Schritt 4 (nach [MZHM09])

LogSig

Bei diesem Algorithmus werden drei Schritte angewandt, um die Log-Meldungen zu gruppieren. [TLP11]

1.GENERIERUNG VON TERM-PAAREN: Zunächst wird jede Nachricht in ein Set von Begriffen konvertiert, wobei die Reihenfolge beibehalten wird. Dies wird an folgender Nachricht exemplarisch aufgezeigt:

```
File size: 3 MB
```

Durch die Konvertierung wird folgendes Set an Begriffen erzeugt:
`(File, size:),(File, 3),(File, MB),(size:, 3),(size:, MB), (3, MB)`
Bei der Aufteilung bleibt die Reihenfolge erhalten, jedoch können auf den Paaren schneller Berechnungen durchgeführt werden als auf Wort-Sequenzen unterschiedlicher Länge. [TLP11]

2. PARTITIONIERUNG DER LOG-MELDUNGEN: Im nächsten Schritt werden k Gruppen gesucht, sodass jede Gruppe möglichst viele gemeinsame

Paare beinhaltet. Der Parameter *k* muss zu Beginn manuell festgelegt werden, das Ergebnis des Algorithmus hängt also stark von der Wahl von *k* ab. Diese muss entweder mittels empirischer Versuche erfolgen oder basierend auf Expert*innenwissen. [TLP11]

3. GENERIERUNG DES LOG-KEYS: Im letzten Schritt werden die Log-Keys, basierend auf den meisten gemeinsamen Paaren pro Gruppe, konstruiert. [TLP11]

Spell

Spell steht für *Streaming structured Parser for Event Logs using LCS* und basiert, wie der Name bereits sagt, auf dem LCS-Algorithmus. Bei dem Verfahren handelt es sich um einen Online-Algorithmus, der eine Liste mit den Log-Keys erstellt, hier genannt *LCSMap*. Diese besteht aus Konstrukten namens *LCSObjects*, die den Log-Key (*LCSSeq*) sowie eine Liste der zugehörigen Zeilen-IDs (*lineIDs*) beinhalten. Der Ablauf des Verfahrens wird in Abbildung 2.4 dargestellt. [DL17]

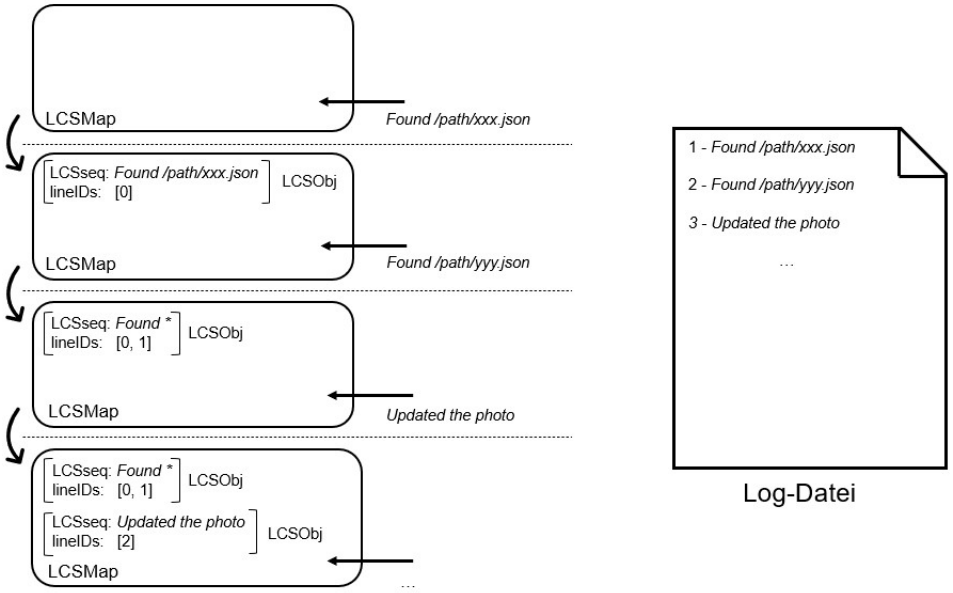


Abbildung 2.4: Funktionsweise von Spell

Zu Beginn ist die *LCSMap* leer. Mit der ersten Meldung wird ein *LCSObject* angelegt, das als *LCSSeq* diese Meldung beinhaltet. Zudem wird die Zeilen-ID in *lineIDs* gespeichert. Beim Eintreffen der nächsten Meldung wird diese mit den bestehenden Log-Keys verglichen. Als String-Metrik wird der LCS-Algorithmus verwendet. Liegt der Wert über einem vordefinierten Threshold, wird *LCSSeq* aktualisiert und die Zeilen-ID in die Liste eingefügt. Andernfalls wird ein neues *LCSObject* angelegt. Der Threshold muss dabei manuell gewählt werden und basiert entweder auf Expert*innenwissen oder muss durch empirische Versuche optimiert werden. [DL17]

BSG

Ein anderer Online-Algorithmus ist Basic Signature Generation (BSG), der im Gegensatz zu den oben genannten Verfahren auf der Levenshtein-Distanz basiert. [Guo+19] Dieser besteht aus drei Schritten:

1. DOMAIN KNOWLEDGE PREPROCESSING: Zunächst kann eine Vorverarbeitung anhand von Expert*innenwissen erfolgen, wie beispielsweise das Filtern von IP-Adressen. Dafür werden reguläre Ausdrücke verwendet. Der Schritt basiert auf der Annahme, dass einfache reguläre Ausdrücke wenig menschlichen Aufwand erfordern, jedoch eine große Auswirkung auf die Qualität des Algorithmus haben können. [Guo+19]

2. BAGS OF LOG GENERATION: Dieser Schritt ergibt sich aus den Recherchen in He et al. [He+17] und Makanju et al. [MZHM12], die zeigen, dass das Gruppieren von Meldungen der gleichen Länge die Resultate verbessert. Aus diesem Grund werden alle Meldungen mit der gleichen Anzahl von Wörtern in einen sogenannten *Bag* sortiert. Die Log-Key-Extrahierung erfolgt dann pro Bag. Es kann allerdings vorkommen, dass Meldungen unterschiedlicher Länge den gleichen Log-Key besitzen. In Schritt 3 können deshalb unterschiedliche Bags zusammengeführt werden. [Guo+19]

3. BASIC SIGNATURE GENERATION: Die Log-Meldungen liegen nun in Bags fester Länge vor, weshalb hier die Levenshtein-Distanz für den String-Vergleich verwendet wird. In diesem Schritt werden diverse Objekte namens *SigMaps* erzeugt. In diesen wird, ähnlich wie bei Spell, eine Liste erzeugt, in der die Log-Keys (hier Signaturen genannt) und die zugehörigen Zeilen-IDs in einem *SigObj* gespeichert werden. Der Ablauf wird in Abbildung 2.5 dargestellt. Bei jeder neu ankommenden Nachricht wird zunächst die Anzahl der Wörter gezählt und die Meldung dementsprechend in die zugehörige *SigMap* eingeordnet. Anschließend wird, wie bei Spell, die Meldung mit den vorhandenen Signaturen verglichen, allerdings basierend auf der Levenshtein-Distanz. Ist dieser Wert geringer als ein vordefinierter Threshold, wird die Signatur des entsprechenden *SigObj* aktualisiert und die Zeilen-ID hinzugefügt. Andernfalls wird ein neues *SigObj* erstellt. Anschließend werden identische Siganturen aus *SigMaps* unterschiedlicher Länge zusammengeführt. [Guo+19]

Online Dictionary Creation Algorithm

Bei diesem Algorithmus wird die Ähnlichkeit von zwei Strings msg_1 und msg_2 mithilfe folgender Metrik berechnet:

$$< msg_1, msg_2 >= \frac{n_{12}}{\sqrt{n_1 \cdot n_2}}$$

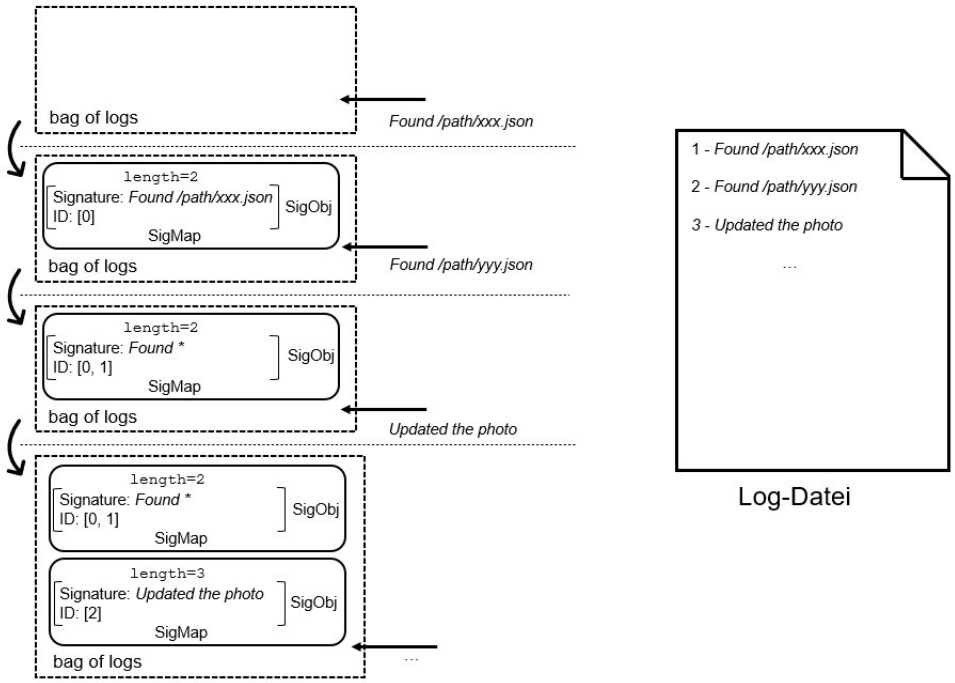


Abbildung 2.5: Funktionsweise des BSG

Dabei ist n_{12} die Anzahl identischer Wörter von beiden Meldungen unter Berücksichtigung der Reihenfolge. Zur Berechnung wird LCS verwendet. n_1 und n_2 sind die Anzahl der Wörter aus msg_1, bzw. msg_2. Der Algorithmus beginnt mit einem leeren Set, in dem Schritt für Schritt neue Cluster hinzugefügt werden. Die erste Meldung erzeugt ein erstes Cluster. Jede neue Nachricht (hier Event genannt) wird mit den Meldungen, die die vorhandenen Cluster repräsentieren, in der Reihenfolge verglichen, in der die Cluster erstellt wurden. Die Meldung wird dem ersten Cluster zugeordnet, zu dem ein vordefinierter Threshold überschritten wird. Dieses Vorgehen garantiert, dass zwei Meldungen, die den gleichen Log-Key besitzen, auch immer dem gleichen Cluster zugeordnet werden, unabhängig von dem Zeitpunkt, zu dem die Meldung auftritt. Wird der Schwellwert für keine der vorhandenen Meldungen erfüllt, wird ein neues Cluster angelegt und das aktuelle Event wird zur repräsentativen Meldung des Clusters. [Aha+09]

2.2 ANOMALIEDETEKTION

Die Fehlererkennung, die Ziel dieser Arbeit ist, lässt sich in den Kontext der Anomaliedetektion einordnen. Anomaliedetektion ist eine sehr wichtige Aufgabe, die in zahlreichen Anwendungsfällen eine Rolle spielt, unter anderem in der Intrusion Detection, Fraud Detection, Bildverarbeitung sowie im medizinischen Bereich und dem Gesundheitswesen. Aus diesem Grund ist es auch ein viel betrachtetes Thema in der Literatur. [HA04] [CBK09] Allgemein beschreibt es die Problemstellung, Muster in Daten zu finden, die vom erwarteten Verhalten abweichen. Solch abweichende Muster werden als An-

omalie bezeichnet. [CBK09] Im Kontext dieser Arbeit stellen die Fehler des Microservice Anomalien dar, da diese vom erwarteten Programmverhalten abweichen.

2.2.1 *Arten von Anomalien*

Je nach Anwendungsfall gibt es unterschiedliche Arten von Anomalien. Diese lassen sich in drei Kategorien unterteilen.

Punktanomalien

Wenn individuelle Datensätze vom Rest der Datenmenge abweichen, wird dies als Punktanomalie bezeichnet. Punktanoamlien sind die einfachste Art von Anomalien. Ein Beispiel wird in Abbildung 2.6 gezeigt. Es werden die

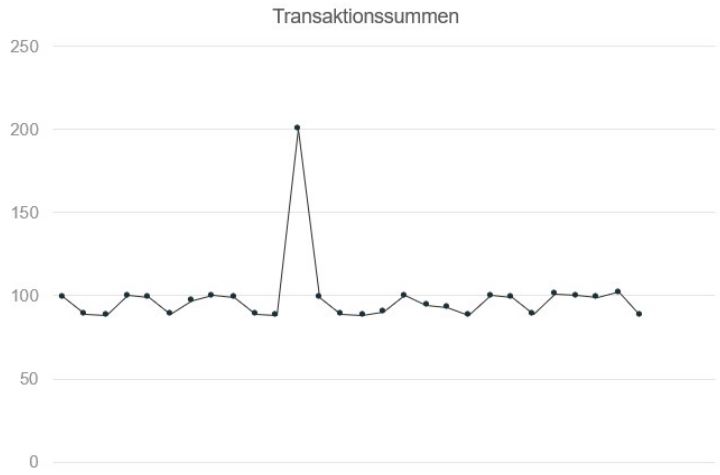


Abbildung 2.6: Beispiel für eine Punktanomalie [CBK09]

Beträge von Transaktionen abgebildet. Eine Summe weicht stark von den übrigen Summen ab und stellt somit eine Punktanomalie dar. [CBK09]

Kontextuelle Anomalien

Kontextuelle Anomalien sind Datenpunkte, die je nach Kontext als normal oder als Anomalie angesehen werden. [AA19] Zur Erläuterung kann folgendes Beispiel herangezogen werden, das in Abbildung 2.7 zu sehen ist. Die Temperatur zum Zeitpunkt t_2 ist die gleiche wie zum Zeitpunkt t_1. Dabei ist eine niedrige Temperatur im Dezember zu erwarten, im Juni jedoch nicht. Somit ist die Temperatur in t_2 aufgrund des Kontextes (der Zeit) eine Anomalie, in t_1 hingegen nicht.

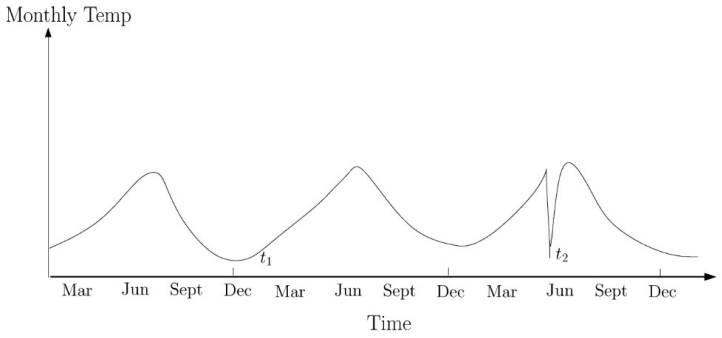


Abbildung 2.7: Beispiel für eine kontextuelle Anomalie [CBK09]

Kollektive Anomalien

Eine Sammlung von Datenpunkten wird als kollektive Anomalie bezeichnet, wenn die einzelnen Punkte zwar normal sind, ihr Auftreten zusammen hingegen als anomal angesehen wird. Abbildung 2.8 zeigt diese Art von Anomalien. Hier wird ein Elektrokardiogramm eines Menschen abgebildet.

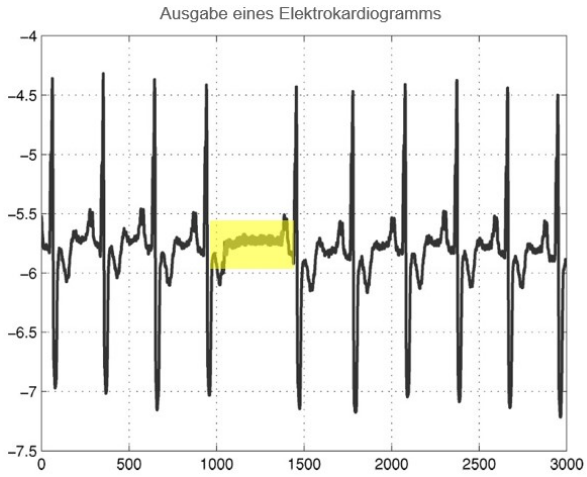


Abbildung 2.8: Beispiel für eine kollektive Anomalie [CBK09] (Die Abbildung wurde angepasst.)

Die hervorgehobenen Werte (von ca. 1000 bis 1500) sind für sich betrachtet nicht unnormal. Das Auftreten dieser Werte über den ungewöhnlich langen Zeitraum hinweg stellt jedoch eine Anomalie dar. [CBK09]

2.2.2 *Verfahren zur Anomaliedetektion*

Trotz der großen Bedeutung und der zahlreichen Anwendungsfälle stellt Anomaliedetektion weiterhin eine große Herausforderung im Bereich von Machine Learning dar. Aufgrund der Vielschichtigkeit durch unterschiedliche Arten von Anomalien, Daten und Problemstellungen existiert kein bester Algorithmus, der alle Varianten abdeckt. In Abhängigkeit der Anforderun-

gen des individuellen Anwendungsfalls muss aus einer Vielzahl möglicher Verfahren das geeignete gewählt werden. [SBK19]

Die Verfahren zur Anomaliedetektion können in drei Teilbereiche gegliedert werden, die im Folgenden erläutert werden.

Supervised Learning

Beim Supervised Learning werden dem Algorithmus sowohl die Daten als auch die dazugehörigen Labels zur Verfügung gestellt, sprich der erwartete Output. Ein Modell lernt dann die Zusammenhänge zwischen Input und Output und kann dadurch Vorhersagen auf neuen, noch ungesehenen Daten treffen. [MG16] Eine große Herausforderung der Supervised-Anomaliedetektion besteht darin, gelabelte Daten für alle möglichen Arten von Anomalien zu erhalten. Dabei ist zu Beginn möglicherweise nicht jede Ausprägung von Anomalien bekannt, weshalb Supervised-Anomaliedetektion oft einen hohen manuellen Aufwand und Expert*innenwissen erfordert. [CBK09]

Semi-Supervised Learning

Bei der Semi-Supervised-Anomaliedetektion wird vorausgesetzt, dass alle Instanzen des Trainings-Datensatzes der normalen Klasse angehören und somit keine Anomalien beinhalten. Aus diesem Grund werden keine Labels von Anomalien benötigt, was einen Vorteil gegenüber den Supervised-Verfahren darstellt. [CBK09]

Unsupervised Learning

Beim Unsupervised Learning werden keine Labels benötigt. Ziel dieser Techniken ist es, Strukturen und Muster in den Daten zu erkennen und zu prüfen, welche Datenpunkte von diesen abweichen. Verfahren der Unsupervised-Anomaliedetektion basieren dabei auf der Annahme, dass die normalen Daten wesentlich stärker vertreten sind als die Anomalien. [CBK09]

2.3 IMBALANCED DATA

Ein bekanntes Problem des Machine Learnings ergibt sich durch *Imbalanced Data*, also unbalancierte bzw. unausgeglichene Daten, und das damit verbundene *Class Imbalance Problem*. Dieses tritt auf, wenn in einer Klassifikations-Fragestellung eine Klasse überrepräsentiert ist. Klassische Verfahren neigen in diesem Fall dazu, die unterrepräsentierte Klasse zu ignorieren, weshalb sich eine Unbalanciertheit der Daten negativ auf die Vorhersagegüte der Modelle auswirkt. [CJK04]

Dies stellt auch eine Herausforderung bei der Anomaliedetektion dar, da Anomalien im Normalfall stark unterrepräsentiert sind und somit nur einen geringen Anteil an der Gesamtmenge der Daten bilden. Insbesondere die meisten Supervised-Verfahren erwarten jedoch einen ausgeglichenen Datensatz. [MK12]

Ein Beispiel dafür wird in He et al. [HG09] im Zusammenhang mit dem *Mammography Data Set* beschrieben. Dieses beinhaltet Bilder einer Mammographie und wird häufig als Beispiel für Imbalanced Data verwendet. Die Bilder sind mit den Labels *Positive* für gesunde Patientinnen und *Negative* für kranke Patientinnen versehen. Die Anzahl gesunder Patientinnen übersteigt dabei mit 10.923 stark die der kranken, von denen nur 260 vertreten sind. Viele Klassifizierer tendieren dazu, fast 100% der überrepräsentierten Klasse richtig vorherzusagen, hingegen jedoch nur 0-10% der Minderheiten-Klasse. In dem konkreten Beispiel würde das bedeuten, dass wenn 10% der negativen Klasse richtig vorhergesagt werden, 234 kranke Patientinnen fälschlicherweise als gesund klassifiziert würden. [HG09]

Für die Lösung des *Class Imbalance Problem* werden in der Literatur unterschiedliche Ansätze aufgeführt.

Eine Möglichkeit ist das Ausgleichen der Unbalanciertheit durch Sampling, dies geschieht bereits vor dem Lernen des Modells. Durch Oversampling werden weitere Daten der unterrepräsentierten Klasse erzeugt und durch Undersampling die Daten der überrepräsentierten Klasse reduziert. [CJK04]

Ein Lösungsansatz während des Lernens sind kostensensitive Methoden und das One-Class Learning. Kostensensitive Methoden berücksichtigen die Unausgeglichenheit in den Daten bei den Kosten, die durch Fehlklassifizierung entstehen. Dabei wird die Fehlklassifizierung von Daten der unterrepräsentierten Klasse stärker gewichtet als die der überrepräsentierten Klasse. Beim One-Class Learning wird hingegen nicht zwischen den Instanzen der über- und unterrepräsentierten Klasse unterschieden, sondern stattdessen nur eine Klasse für alle Daten verwendet. Dabei wird die Ähnlichkeit der Daten gemessen und mittels eines Schwellwertes die Klassifizierung vorgenommen. Datenpunkte, die dem Großteil der Daten nicht ähnlich sind, werden somit als Instanz der unterrepräsentierten Klasse klassifiziert. [HG09]

Ein wichtiger Aspekt bei der Arbeit mit unbalancierten Daten ist die Wahl der geeigneten Metrik. Wird ein binäres Klassifikations-Problem mit der unterrepräsentierten Klasse als positive Klasse und der überrepräsentierten Klasse als negative Klasse betrachtet, lassen sich die Metriken zur Bestimmung der Vorhersagegüte an der Konfusionsmatrix aus Tabelle 2.3 ableiten.

		Wahre Klasse	
		Positiv	Negativ
Vorhersage	Positiv	True Positive (TP)	False Positive (FP)
	Negativ	False Negative (FN)	True Negative (TN)

Tabelle 2.3: Konfusionsmatrix

Eine häufig verwendete Metrik im Machine Learning ist die Accuracy. Diese lässt sich wie folgt berechnen [HG09]:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TN} + \text{FN} + \text{TP} + \text{FP}} \tag{2.6}$$

Ist allerdings beispielsweise ein Datensatz mit 1.000 Zeilen gegeben, von denen 990 der negativen Klasse angehören und 10 der positiven, kann bereits eine Accuracy von 99% erreicht werden, wenn für jede Zeile die negative Klasse gewählt wird. Somit ist die Accuracy nicht für unausgeglichene Daten geeignet. Dieser Effekt lässt sich mithilfe der Konfusionsmatrix erklären. Die linke Spalte repräsentiert die positiven Datenpunkte und die rechte die negativen, somit beschreibt der Anteil der beiden Spalten die Verteilung der Klassen. Metriken, die Werte aus beiden Spalten verwenden, sind aus diesem Grund sensitiv gegenüber Unausgeglichenheiten. Wie in Formel 2.6 zu erkennen ist, verwendet die Accuracy Informationen aus beiden Spalten. Dadurch steht die Metrik in Abhängigkeit zur Verteilung der zugrundeliegenden Daten. Änderungen in der Verteilung führen zu Änderungen der Metrik, ohne dass die Vorhersagegüte des Klassifizierers variiert. [HG09]

Aus diesem Grund kommen im Kontext von Imbalanced Data meist andere Metriken zum Einsatz, die im Folgenden erläutert werden.

Die Precision gibt an, welcher Anteil aller positiv vorhergesagten Beispiele korrekterweise als positiv klassifiziert wurde und misst somit die Genauigkeit. Sie ist wie folgt definiert [HG09]:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.7}$$

Recall (auch Sensitivity gennant) ist eine Metrik zur Ermittlung der Trefferquote, die prüft, wie viele Datenpunkte der positiven Klasse richtig vorhergesagt wurden und wird mittels folgender Gleichung berechnet [HG09]:

$$\text{Recall/Sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.8}$$

Dabei ist die Precision empfindlich gegenüber der Verteilung der Daten, da die Formel Werte aus beiden Spalten der Konfusionsmatrix einschließt, Recall jedoch nicht. Dies verdeutlicht Abbildung 2.9.

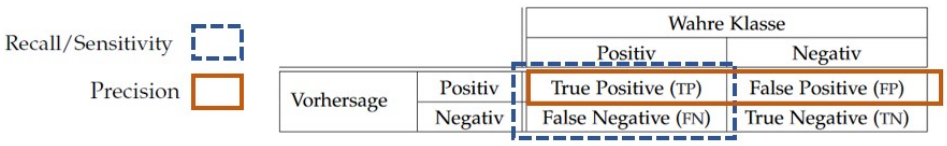


Abbildung 2.9: Recall und Precision

Eine alleinige Betrachtung des Recalls ist allerdings nicht hinreichend, da dadurch keine Aussage darüber getroffen wird, wie viele Beispiele inkorrekterweise positiv vorhergesagt wurden. Dies ist neben der Precision auch mithilfe der Specificity möglich, die angibt, mit welcher Wahrscheinlichkeit

ein negativer Datenpunkt korrekt als negativ vorhergesagt wird. [HG09] Die Definition ist wie folgt :

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}} \tag{2.9}$$

Die Specificity ist ebenfalls unabhängig von der Datenverteilung, wie Abbildung 2.10 zeigt.

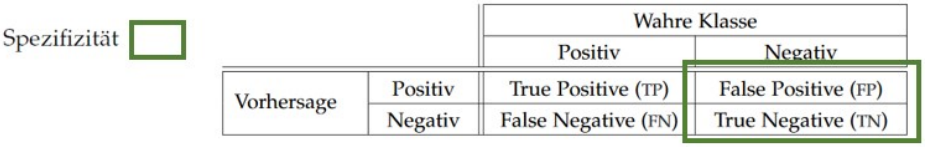


Abbildung 2.10: Specificity

Im Kontext von Imbalanced Data wird zur Beurteilung eines Klassifizierers meist die kombinierte Betrachtung von Recall und Precision verwendet, wie beispielsweise mithilfe des F-Measures [HG09]:

$$\text{F-Measure} = \frac{(1 + \beta^2) * Recall * Precision}{\beta^2 * Recall + Precision} \tag{2.10}$$

Dabei beschreibt β den Koeffizienten, mit dem die relative Wichtigkeit von Recall gegenüber Precision definiert werden kann. Standardmäßig ist $\beta = 1$. [HG09]

3

RELATED WORK

Neben regelbasierten Ansätzen werden in der Literatur mehrere Aspekte der Logfile-Analyse betrachtet, darunter Anomaliedetektion, Fehler-Diagnose, Programm-Verifikation und Performance-Vorhersage. [He+16]

In Xu et al. [Xu+10] wird eine Methode vorgestellt, die automatisiert Laufzeitprobleme von Systemen ermitteln kann. Für das Parsen der Logdateien werden Sourcecode-Analysen hinzugezogen und daraus neue Features erstellt. Diese werden dann mit der Unsupervised-Methode der Principal Component Analysis (PCA) analysiert. Zudem erfolgt eine Visualierung mittels Entscheidungsbäumen, um eine bessere Nachvollziehbarkeit für Administrator*innen und Entwickler*innen zu gewährleisten.

Die beschriebene Problematik der hohen Anzahl falscher Fehler, die die Suche nach echten Fehlern erschwert, tritt auch in anderen Bereichen auf, wie zum Beispiel in der Network Intrusion Detection. In Meng et al. [MK12] wird ein adaptiver False-Alarm-Filter vorgestellt, der sechs verschiedene Methoden des Machine Learnings (u.a. Support Vector Machine (SVM), Naive Bayes und Decision Tree) vergleicht. In Abhängigkeit des Kontexts wählt der Filter den geeigneten Algorithmus und kann somit die False Alarms um bis zu 80% reduzieren. Dabei sind diese Logs aber nicht derart unstrukturiert, wie es bei den vorhandenen Daten der Microservices der Fall ist.

Eine Methode zur Vorhersage von Fehlverhalten wird in Fulp et al. [FFH08] vorgestellt, in der eine SVM als Supervised-Modell eingesetzt wird. Bei Supervised-Modellen ist allerdings zu beachten, dass ausreichende Labels in der Praxis meist nicht zur Verfügung stehen. SVMs können auch als One-Class SVM in einem Semi-Supervised-Modus zur Anomaliedetektion eingesetzt werden. Wang et al. [WWM04] beschreibt einen solchen Ansatz. Eine SVM wird auf fehlerfreien Daten trainiert, um in neuen Daten Anomalien erkennen zu können, die dann als Fehler gekennzeichnet werden.

In He et al. [He+16] wird die Problemstellung der Fehlererkennung ebenfalls im Kontext von Anomaliedetektion betrachtet. Hierfür werden drei Supervised- und drei Unsupervised-Modelle miteinander verglichen.

In Vaarandi et al. [VKP16] wird ein Tool namens LogCluster vorgestellt, das mittels Clustering Ausreißer in Security- und Event-Logs identifiziert.

Ein Teilbereich von Machine Learning, Deep Learning, gewinnt zunehmend an Bedeutung. In Du et al. [DL17] wird ein Framework namens Deeplog vorgestellt, das als Trainingsmodell Long Short-Term Memory (LSTM), eine Spezialisierung von Recurrent Neural Network (RNN), verwendet, um Anomalien in einem Log-Execution-Path zu finden.

In Lu et al. [Lu+18] wird mit einem Convolutional Neural Network (CNN) ein Supervised-Modell verwendet, das laut den dort vorgestellten Ergebnissen eine bessere Accuracy erzielt, als andere Arten von neuronalen Netzen.

Allerdings stellt Accuracy keine geeignete Metrik für unbalancierte Datensätze dar.

Oben beschriebene Methoden erfüllen nicht die gleichen Voraussetzungen, die im Zuge der genannten Problemstellung aufgetreten sind, weil beispielsweise die Daten eine weniger hohe Heterogenität aufweisen oder die Fragestellung nicht im Kontext von *Imbalanced Data* betrachtet wird. Aus diesem Grund können sie in dieser Arbeit nicht ohne Weiteres verwendet werden, was die Entwicklung einer eigenen Lösung erforderlich macht. Allerdings gibt es vielversprechende Ansätze, wie den Einsatz von LSTMs in Du et al. [DL17], die im Rahmen dieser Arbeit näher untersucht werden.

4 KONZEPTION

In diesem Kapitel wird die Konzeption des Log-Parsings und der Modelle zur Fehlererkennung beschrieben. Dafür wird zunächst die verwendete Datengrundlage analysiert und daraus Anforderungen an die Vorverarbeitung und die Modelle zur Anomaliedetektion abgeleitet. Die Verwendbarkeit der Log-Parsing-Methoden, die in Kapitel 2.1.2 vorgestellt wurden, wird anhand dieser Anforderungen analysiert. Zudem werden geeignete Modelle zur Anomaliedetektion von Log-Dateien in Anlehnung an die in der Literatur gängigen Verfahren konzipiert.

4.1 DATENGRUNDLAGE

Im Rahmen dieser Arbeit wurden Log-Dateien einer Microservice-Anwendung der ORDIX AG zur Synchronisation von Kontakten verwendet. Die Anwendung wird von jedem Mitarbeiter/jeder Mitarbeiterin genutzt und synchronisiert für jeden einzelnen/jede einzelne die Kontakte aller Mitarbeiter*innen. Auf den Sourcecode kann nicht zugegriffen werden.

4.1.1 *Datenmenge und Zeitraum*

Die zugrundeliegende Datenmenge ist essentiell für die Qualität der Anomaliedetektion, da ein repräsentativer Zeitraum entscheidend für die Vorhersagegüte der Modelle ist. Im Kontext dieser Arbeit wurden für das Trainieren der Modelle Log-Dateien im Umfang eines Monats zur Verfügung gestellt, vom 11. Mai 2020 bis zum 10. Juni 2020. Der Datensatz umfasst 220.077 Log-Zeilen. Die ersten drei Wochen des verwendeten Datensatzes sind Daten des Normalzustands und somit fehlerfreie Daten. Die letzte Woche hingegen beinhaltet Fehler.

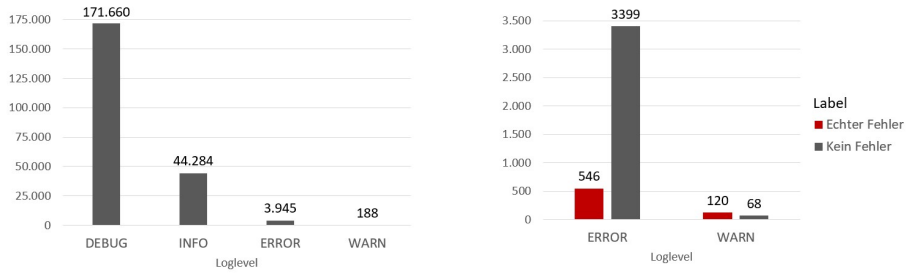
Je nach Anwendung kann es zu unterschiedlich starken Auslastungen in Abhängigkeit des Zeitraums kommen. Dabei lassen Schwankungen der Auslastung in manchen Kontexten auf Fehler schließen, in anderen jedoch nicht.

In Xu et al. [Xu+10] wird beispielsweise ein System beschrieben, bei dem zu jedem Zeitpunkt unterschiedlich hohe Workloads auftreten können. Dort werden unter anderem die Log-Dateien des Hadoop Distributed File Systems analysiert. Die Anzahl der Log-Meldungen spielt in diesem Zusammenhang keine Rolle, da sie von der Auslastung des Systems abhängt, die stark variieren kann, ohne dass ein Fehler vorliegt. [Xu+10]

In anderen Systemen hingegen kann eine erhöhte Auslastung ein Indikator für eine Anomalie sein, wie beispielsweise im Bereich der Network Intrusion Detection, wenn ein Angreifer sich unrechtmäßigen Zugang zu Systemen verschafft hat und dort für erhöhte Auslastung sorgt. [Lan+18]

Die Daten, die im Rahmen dieser Arbeit verwendet werden, stammen, wie beschrieben, von einer Microservice-Anwendung zur Synchronisation von Kontakten. Die Auslastung des Systems hängt von der Häufigkeit der Synchronisationen ab, die auch im Normalzustand stets variiert, und steht somit nicht im Zusammenhang mit möglichen Fehlern.

Wie erläutert, liegt in den Daten eine Unausgeglichenheit vor. Dies wird in Abbildung 4.1 verdeutlicht.



(a) Anzahl der Meldungen pro Loglevel

(b) Anzahl der Meldungen mit Loglevel *WARN* oder *ERROR* pro Label

Abbildung 4.1: (a) zeigt die Anzahl der Meldungen pro Loglevel, (b) die Meldungen mit Loglevel *WARN* oder *ERROR*, gruppiert nach Label. Meldungen mit dem Loglevel *DEBUG* und *INFO* beinhalten keine Fehler und werden deshalb in (b) nicht berücksichtigt.

Wie zu erkennen ist, sind die meisten Nachrichten vom Loglevel *DEBUG* oder *INFO*. Nur wenige entsprechen dem Loglevel *ERROR* oder *WARN*, wovon wiederum nur wenige echte Fehler sind, lediglich 0,003% der Gesamtdaten. Dieses starke Ungleichgewicht der Daten muss bei der Umsetzung und Evaluierung berücksichtigt werden.

4.1.2 *Struktur*

Die Log-Meldungen werden durch ein Print-Statement erzeugt und bestehen aus unstrukturiertem Text. Alle Log-Zeilen folgen dem gleichen Aufbau (aus Datenschutzgründen wird der Klassenname nicht angegeben):

```
<Datum> <Uhrzeit> <Loglevel> [<Thread>] [<Klasse>] - <Meldung>
```

Beispiel:

```
2020-05-11 10:30:20,944 INFO [scheduling-1] [class] - Aktualisieren
    abgeschlossen.
```

Somit ergeben sich daraus die *Meldung* sowie die Metadaten *Datum*, *Uhrzeit*, *Loglevel*, *Thread* und *Klasse*. Die Klasse, der Thread sowie der Loglevel korrespondieren stets mit der Meldung. Es kann somit keine Meldung mehreren Klassen oder Threads zugeordnet werden.

Ein Auszug aus den Log-Dateien wird in Abbildung 4.2 dargestellt. Zur besseren Lesbarkeit wurden die Infos zu Klasse und Thread entfernt. Die markierten Teile der Log-Meldung stellen den variablen Anteil dar. Aus Da-

tenschutzgründen wurden ORDIX-spezifische Informationen ersetzt, der Datentyp wurde jedoch beibehalten.

```
2020-05-13 05:00:10,277 DEBUG - Found /path/yyy.json
2020-05-13 05:00:10,278 ERROR - Exception beim Lesen der Konfigurationsdatei für yyy - Standardwerte werden verwendet
2020-05-13 05:00:10,278 DEBUG - No contact folder found for yyy - using default (X)
2020-05-13 05:00:10,278 DEBUG - Bearer Token URL: <URL> Body: client_secret=***&scope=***
2020-05-13 05:00:10,685 DEBUG - Got a new bearer token , expires_in 3599 sec
2020-05-13 05:00:10,685 DEBUG - access_token=***...
2020-05-13 05:00:10,852 INFO - Contact Folder existiert bereits, daher wird nun die dazugehörige ID verwendet: <ID>
2020-05-13 05:00:10,852 DEBUG - Updating yyy-ordix's photo of IDbbb
2020-05-13 05:00:11,040 DEBUG - Updated the photo
2020-05-13 05:00:11,040 DEBUG - Found /path/xxx.json
2020-05-13 05:00:11,040 DEBUG - Updating xxx-ordix's photo of IDaaa
2020-05-13 05:00:11,608 DEBUG - Updated the photo
```

Abbildung 4.2: Auszug der Log-Dateien

Wie in der Abbildung zu erkennen ist, bestehen die Meldungen zum Großteil aus Text, sowohl in englischer als auch in deutscher Sprache. Der variable Anteil der Log-Meldungen bezeichnet zum Großteil die Kennung eines Mitarbeiters/einer Mitarbeiterin und besteht somit ebenfalls aus Text.

4.1.3 *Fehlerarten*

Wie in Kapitel 2.2.1 gezeigt, können Anomalien in drei unterschiedliche Kategorien aufgeteilt werden. Im Kontext dieser Arbeit lassen sich die Log-Meldungen wie folgt in diese Kategorien einordnen:

A. Punktanomalien: Log-Meldungen, deren Auftreten zu jedem Zeitpunkt einen Fehler darstellt.

B. Kontextuelle Anomalien: Log-Meldungen, die zu einer bestimmten Uhrzeit einen Fehler bedeuten, zu einer anderen Uhrzeit jedoch als normales Programmverhalten gewertet werden können.

C. Kollektive Anomalien: Log-Meldungen, die individuell betrachtet keinen Fehler bedeuten, deren Aufkommen zusammen (z.B. ungewöhnlich häufig pro Tag) auf ein inkorrektes Programmverhalten schließen lässt.

In den zugrundeliegenden Daten sind die meisten Fehler vom Typ A, Log-Einträge, deren Auftreten immer einen Fehler bedeutet. Daneben existieren aber auch Fehler der Kategorie B, die in Abhängigkeit zur Uhrzeit stehen. Fehler vom Typ C sind in den Trainingsdaten nicht vorhanden, es kann jedoch passieren, dass diese in zukünftigen Log-Dateien auftreten werden.

4.2 LOG-PARSING

In diesem Kapitel werden die Anforderungen an das Log-Parsing analysiert. In Abhängigkeit der in Kapitel 4.1 vorgestellten Eigenschaften der Daten wird aus den Log-Parsing-Methoden, die in Kapitel 2.1.2 vorgestellt wurden, das passende Verfahren gewählt. Eine Möglichkeit zur Extrahierung

von Log-Keys ist das Erstellen von regulären Ausdrücken, das jedoch ein umfangreiches Expert*innenwissen voraussetzt und deshalb hier nicht näher betrachtet wird.

Log-Parsing-Methoden lassen sich, wie in Kapitel 2.1.2 beschrieben, in zwei Kategorien aufteilen: Offline-Log-Parsing, bei dem alle vorhandenen Logs in einem Batch-Prozess verarbeitet werden und Online-Log-Parsing, bei dem die einzelnen Meldungen im Streaming-Verfahren analysiert werden. Die Log-Dateien, die im Rahmen dieser Arbeit verwendet werden, werden nur eine begrenzte Zeit vorgehalten. Aus diesem Grund besteht die Anforderung, dass das Log-Parsing nicht erneut von Beginn an ausgeführt werden soll, wenn neue Log-Dateien hinzukommen, da sonst stets alle Log-Dateien vorhanden sein müssen. Diese Anforderung erfüllen nur die Methoden des Online-Log-Parsings, weshalb nur die Verfahren der linken Spalte aus Tabelle 4.1 im Folgenden betrachtet werden.

Online-Log-Parsing	Offline-Log-Parsing
Spell	IPLoM
BSG	LogSig
Online Dictionary Creation Algorithm	

Tabelle 4.1: Log-Parsing-Verfahren

Wie in Kapitel 4.1.2 beschrieben, bestehen die einzelnen Log-Meldungen aus unstrukturiertem Text in deutscher und englischer Sprache. In Vorbereitung zur Identifikation der Log-Keys werden die Meldungen anhand des Leerzeichens getrennt. Die Metriken zum String-Vergleich werden somit auf Basis der Wörter und nicht auf den einzelnen Zeichen angewendet. Ein Vergleich der Zeichen ist in diesem Kontext nicht sinnvoll, weil es Parameter gibt, die mit bis zu 160 Zeichen deutlich länger sind als die anderen Worte. Bei einem Vergleich von Zeichen würden deshalb diese Parameter die größte Übereinstimmung erreichen und fälschlicherweise als Log-Key extrahiert werden. Dies wird im folgenden Beispiel 4.1 verdeutlicht.

Log-Meldung 1:	*Contact mit der id: <Parameter_1> wird aktualisiert.*	
Log-Meldung 2:	*Caching contact ID <Parameter_1>*	
Log-Meldung 3:	*Contact mit der id: <Parameter_2> wird aktualisiert.*	
Log-Meldung 4:	*Caching contact ID <Parameter_2>*	(4.1)

Wie beschrieben haben die Parameter hier eine Länge von bis zu 160 Zeichen. Der Rest der Meldung besteht (mit Leerzeichen) lediglich aus 38 (*Contact mit der id: wird aktualisiert.*), bzw. 19 Zeichen (*Caching contact ID*). Im Folgenden (4.2) sind die Log-Keys veranschaulicht, in Abhängigkeit davon, ob der String-Vergleich auf Zeichen-Basis oder auf Wort-Basis vorgenommen wird:

Vergleich auf Zeichen-Basis:
Log-Key 1: * <*Parameter_1*> *
Log-Key 2: * <*Parameter_2*> *

Vergleich auf Wort-Basis:
Log-Key 1: *Contact mit der id: * wird aktualisiert.*
Log-Key 2: *Caching contact ID ** (4.2)

Es wird deutlich, dass ein Vergleich anhand der Zeichen zu inkorrekten Ergebnissen führt. Aus diesem Grund wird ein Vergleich auf Wort-Basis vorgenommen.

Die Semantik der Log-Meldungen wird nicht berücksichtigt, da viele Log-Meldungen ohne externen Kontext mehrdeutig sind, wie in Oliner et al. [OS07] gezeigt wurde. Des Weiteren wird auch der Wortstamm nicht berücksichtigt, da dies zu inkorrekten Ergebnissen führen kann, wie Beispiel 4.3 zeigt.

Log-Message 1:	*Updating photo cache*	
Log-Message 2:	*Updated photo cache*	(4.3)

Die beiden Meldungen unterscheiden sich nur im ersten Wort, das in beiden Fällen eine Konjugation des Wortes *update* ist. Jedoch stellt die erste Meldung den Beginn des Update-Prozesses dar und die zweite Meldung das Ende. Somit wäre es inkorrekt, sie dem gleichen Log-Key zuzuordnen. Log-Meldungen sind nicht mit anderen geschriebenen Texten vergleichbar. Sie dienen üblicherweise nur der Fehlersuche, sprachliche Feinheiten wie Numerus oder Tempus werden meist nicht berücksichtigt. Sind zwei Log-Meldungen nach Extrahierung der Parameter ähnlich, aber nicht identisch, werden sie im Code üblicherweise auch von unterschiedlichen Log-Anweisungen erzeugt, was dafür spricht, sie unterschiedlichen Log-Keys zuzuordnen.

Einzelne Log-Zeilen beinhalten den Stacktrace der Fehlermeldung. Ein Beispiel wird in Abbildung 4.3 gezeigt. Aus Gründen der Übersichtlichkeit wird dieser nur ausschnittsweise dargestellt.

Der Stacktrace wird dabei stets nach einem Zeilenumbruch an die Log-Meldung angehangen und kann somit identifiziert werden. Er ist jedoch nur für die spätere Fehler-Analyse relevant und nicht für die Extrahierung des Log-Keys, weshalb er in der Vorverarbeitung nicht berücksichtigt wird. Stattdessen wird die Meldung nach dem Zeilenumbruch abgeschnitten.

Abbildung 4.4 zeigt die Anzahl der Log-Meldungen pro Länge der Meldung, was nach dem Trennen anhand des Leerzeichens und dem Filtern des Stacktrace gleichbedeutend mit der Anzahl der Wörter ist. Es ist zu erkennen, dass ein Großteil der Meldungen (über 85%) aus sieben oder weniger

```
2020-05-13 05:00:17,401 ERROR [scheduling-1] [or.sp.sc.su.TaskUtils$LoggingErrorHandler] - Unexpected error occurred in scheduled task.
org.springframework.orm.jpa.JpaSystemException: Unable to acquire JDBC Connection...
        at org.springframework.orm.jpa.vendor.HibernateJpaDialect....
        at org.springframework.orm.jpa.vendor.HibernateJpaDialect....
        at org.springframework.orm.jpa.AbstractEntityManagerFactoryBean....
        at org.springframework.dao.support.ChainedPersistenceExceptionTranslator...
        at org.springframework.dao.support.DataAccessUtils...
        at org.springframework.dao.support.PersistenceExceptionTranslationInterceptor...
        at org.springframework.aop.framework...
        at org.springframework.data.jpa.repository.support.CrudMethodMetadataPostProcessor$CrudMethodMetadataPopulatingMethod...
        at org.springframework.aop.framework.ReflectiveMethodInvocation...
        at org.springframework.aop.interceptor.ExposeInvocationInterceptor...
        at org.springframework.aop.framework.ReflectiveMethodInvocation...
        at org.springframework.data.repository.core.support.SurroundingTransactionDetectorMethodInterceptor...
        at org.springframework.aop.framework.ReflectiveMethodInvocation...
        at org.springframework.aop.framework.JdkDynamicAopProxy...
        at com.sun.proxy...
        at ...
        at...
        at sun.reflect.GeneratedMethodAccessor157...
        at sun.reflect.DelegatingMethodAccessorImpl...
        at java.lang.reflect.Method...
        at org.springframework.scheduling.support.ScheduledMethodRunnable...
        at org.springframework.scheduling.support.DelegatingErrorHandlingRunnable...
        at org.springframework.scheduling.concurrent.ReschedulingRunnable...
        at java.util.concurrent.Executors$RunnableAdapter...
        at java.util.concurrent.FutureTask...
        at java.util.concurrent.ScheduledThreadPoolExecutor...
        at java.util.concurrent.ScheduledThreadPoolExecutor...
        at java.util.concurrent.ThreadPoolExecutor...
        at java.util.concurrent.ThreadPoolExecutor...
        at java.lang.Thread...
Caused by: org.hibernate.exception.GenericJDBCException: Unable to acquire JDBC Connection
...
```

Abbildung 4.3: Beispiel für einen Stacktrace

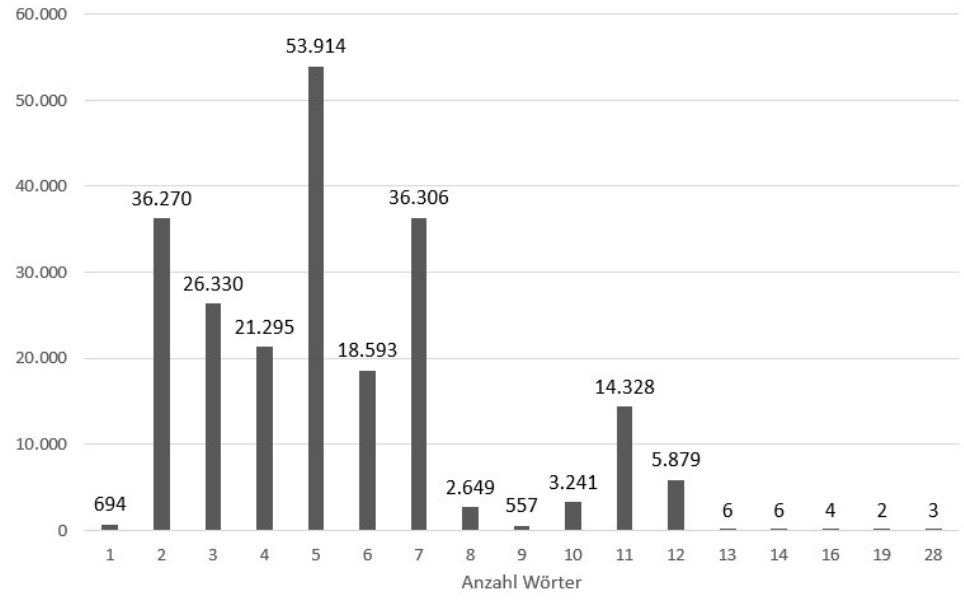


Abbildung 4.4: Länge der Log-Meldungen

Wörtern besteht und somit relativ kurz ist. Ca. 17% aller Meldungen besitzen sogar nur ein oder zwei Wörter. Bei den Log-Meldungen mit zwei Wörtern stellt bei einem Großteil ein Wort den Log-Key dar und das andere den Parameter. Dies muss bei der Wahl des Thresholds für die Methoden der Vorverarbeitung berücksichtigt werden. Bei Meldungen, die nur aus einem Wort bestehen, kann kein Parameter extrahiert werden, weshalb das Wort selbst als Log-Key betrachtet wird.

Somit ergeben sich folgende Aspekte für die Vorverarbeitung:

- Die Metriken zum String-Vergleich werden auf Basis der Wörter und nicht auf Basis der einzelnen Zeichen angewendet. Dies ist sowohl mit dem LCS-Algorithmus als auch mit der Levenshtein-Distanz möglich.
- Die Semantik und Grammatik der einzelnen Wörter wird nicht berücksichtigt.
- Zu Beginn wird der Stacktrace, den einzelne Meldungen besitzen, entfernt.
- Bei der Wahl des Thresholds muss die Anzahl der Wörter beachtet werden, die in Abbildung 4.4 dargestellt ist.

Alle Online-Log-Parsing-Verfahren sind im Rahmen dieser Arbeit geeignet:

1. **Spell** (LCS-Algorithmus)
2. **BSG** (Levenshtein-Distanz)
3. **Online Dictionary Creation Algorithm** (LCS-Algorithmus)

Im Folgenden erfolgt die Konzeption dieser drei Methoden unter Berücksichtigung der oben genannten Aspekte.

Spell

Wie in Kapitel 2.1.2 beschrieben, wird bei diesem Algorithmus eine sogenannte *LCSMap* aufgebaut. Dies geschieht anhand eines String-Vergleichs auf Basis des LCS-Algorithmus. Hierfür muss ein Threshold gewählt werden, nach welchem zwei Meldungen dem gleichen Objekt in der LCSMap zugeordnet werden. Der Threshold gibt also an, wie viele Wörter zweier Log-Zeilen übereinstimmen müssen. Die Anzahl der übereinstimmenden Wörter wird durch den LCS-Algorithmus ermittelt.

Die mögliche Anzahl an Wörtern, die übereinstimmen können, steht in Abhängigkeit zur Länge der Log-Meldung. Aus diesem Grund ist es nicht sinnvoll, einen statischen Threshold zu wählen, sondern diesen in Abhängigkeit zu der Länge der Log-Zeilen zu setzen. In dieser Arbeit wurde deshalb folgender Ansatz gewählt: Für eine Meldung m wird der Threshold t wie folgt berechnet:

$$t = len(m) * a \tag{4.4}$$

Somit beschreibt a den Anteil der Log-Keys, der übereinstimmen muss. Beispielsweise würde ein a von 0,75 bedeuten, dass 75% der Wörter zweier Log-Meldungen identisch sein müssen, damit sie dem gleichen Objekt der LCS-Map zugeordnet werden. Werden zwei Log-Zeilen unterschiedlicher Länge verglichen, so ist die Wortanzahl der längeren Log-Zeile entscheidend.

Der Threshold t variiert stets in Abhängigkeit der Länge der Meldungen, weshalb zur besseren Übersicht im Folgenden stets der Anteil a als Threshold referenziert wird.

In Kapitel 4.2 wurde gezeigt, dass ca. 16% der Meldungen aus zwei Wörtern bestehen. Bei einem Großteil dieser Meldungen handelt es sich dabei bei einem Wort um den Parameter und bei dem anderen um den Log-Key. Um Log-Key und Parameter trennen zu können, wird der Threshold deshalb auf 0,5 gesetzt, somit müssen 50% der Wörter übereinstimmen, was genau dem einen Wort des Parameters entspricht.

Bei längeren Nachrichten könnte es sinnvoll sein, den Threshold höher zu wählen. Deshalb werden in in Kapitel 5.1 für Log-Meldungen, die aus drei oder mehr Wörtern bestehen, unterschiedliche Thresholds eingesetzt und evaluiert.

BSG

Beim BSG-Algorithmus erfolgt im ersten Schritt, wie in Kapitel 2.1.2 beschrieben, eine Bereinigung der Log-Zeilen mittels Expert*innenwissen. Der Schritt wird hier allerdings übersprungen, da keine einfachen Regeln, wie beispielsweise das Ersetzen von IP-Adressen, angwendet werden können. Eine Bereinigung würde eine umfangreiche Analyse voraussetzen, die dem Ziel dieses Schrittes widerspricht.

Im nächsten Schritt erfolgt die Einordung in Bags. Wie bereits erläutert, werden die Log-Meldungen in dieser Arbeit auf Wort-Basis miteinander verglichen. Aus diesem Grund erfolgt die Einordnung in die Bags anhand der Anzahl der Wörter. Mittels dieser Bags werden im dritten und letzten Schritt die SigMaps erzeugt.

Als Metrik zum String-Vergleich wird beim BSG die Levenshtein-Distanz in Bezug auf die Wörter genutzt. Je geringer die Distanz, desto weniger Veränderungen müssen vorgenommen werden und desto ähnlicher sind sich zwei Meldungen. Auch hier muss für die richtige Einordnung ein Threshold festgelegt werden, der die Länge der Nachrichten berücksichtigt. Somit wird auch bei diesem Verfahren der Threshold als Anteil eingsetzt, der beschreibt, wie viele Wörter einer Log-Meldung verändert werden müssen. Ein Threshold von 0,75 würde bedeuten, dass mindestens 75% der Meldungen verändert werden darf, um diese dem gleichen Log-Key zuzuordnen.

Analog zu dem Vorgehen bei Spell wird der Threshold für Zeilen, die aus zwei Wörtern bestehen, auf 0,5 gesetzt. Somit dürfen 50% der Zeile, was einem Wort entspricht, verändert werden, um trotzdem noch dem gleichen Log-Key zugeordnet zu werden. Für Meldungen, die aus mehr als zwei Wörtern bestehen, werden auch hier unterschiedliche Varianten des Thresholds evaluiert, die Ergebnisse werden in Kapitel 5.1 vorgestellt. Durch die Einord-

nung in die Bags werden nur Nachrichten der gleichen Länge miteinander verglichen.

Online Dictionary Creation Algorithm

Bei der Einordnung in Cluster nach dem Online Dictionary Creation Algorithm wird eine normierte Variante des LCS-Algorithmus als Metrik zum String-Vergleich verwendet. Aus diesem Grund wurde sich bei der Wahl des Thresholds am Vorgehen des Spell-Verfahrens orientiert. Bei Nachrichten der Länge Zwei wurde der Threshold auf 0,5 gesetzt, für alle Nachrichten mit mehr als zwei Wörtern wurden unterschiedliche Werte für den Threshold evaluiert. Da die Normierung jedoch in Abhängigkeit der Länge der Meldungen steht, muss dies beim Threshold nicht erfolgen, sodass dieser hier statisch gewählt wird. Die Ergebnisse der Evaluierung werden in Kapitel 5.1 vorgestellt.

4.3 MODELLAUSWAHL

In diesem Schritt erfolgt die Modellauswahl. Dafür werden gängige Methoden der Anomaliedetektion von Log-Dateien analysiert und geprüft, ob sie den definierten Anforderungen gerecht werden.

Aus Kapitel 4.1 ergeben sich folgende Anforderungen an das zu entwickelnde Modell, das auf den vorverarbeiteten Daten und somit den extrahierten Log-Keys aufsetzt:

- Die Vorhersage darf nicht von der Auslastung des Systems abhängen.
- Punktanomalien müssen erkannt werden.
- Kontextuelle Anomalien müssen erkannt werden.

Das zu entwickelnde Modell muss all diesen Anforderungen gerecht werden. Zudem muss der Aspekt der unausgeglichenen Daten berücksichtigt werden. Wie beschrieben kann das Ungleichgewicht nicht im Voraus durch Sampling ausgeglichen werden. Für einige Modelle können spezielle Parameter gesetzt werden, durch die das Ungleichgewicht beim Training berücksichtigt wird. Diese werden im Zusammenhang mit den anderen Hyperparametern im Kapitel 5 beschrieben. Des Weiteren steht dieser Aspekt im Zuge der Evaluierung bei der Beurteilung der Modellqualität im Fokus. Hier wird die Qualitätsanforderung beschrieben und im Zuge dessen erfolgt die Wahl der geeigneten Metriken. Die Laufzeit spielt keine Rolle, da die Analyse der Log-Dateien wöchentlich erfolgt und der Performance-Aspekt deshalb nicht von Bedeutung ist. Für zeitkritische Anwendungen ist dies jedoch ein wichtiger Punkt, der in zukünftigen Arbeiten näher untersucht werden könnte.

Zunächst wird analysiert, ob ein Modell theoretisch die geforderten Fehlerarten erkennen kann. Ob dies bei den vorhandenen Daten auch praktisch der Fall ist, wird ebenfalls in der Evaluierung geprüft.

Um einen Vergleich zwischen den unterschiedlichen Bereichen gewährleisten zu können, werden sowohl Supervised- als auch Semi-Supervised- und Unsupervised-Verfahren berücksichtigt.

4.3.1 *Feature Selection*

Wie in Abschnitt 4.1 beschrieben, können folgende Features direkt aus einer Log-Zeile generiert werden: *Datum*, *Uhrzeit*, *Loglevel*, *Thread*, *Klasse* und die *Meldung*, aus der im Log-Parsing-Schritt der Log-Key extrahiert wurde. Aufgrund der Tatsache, dass Klasse, Thread und Loglevel mit der Meldung korrespondieren und somit keine weiteren Informationen in das Modell bringen können, wurden sie im Trainingsprozess nicht berücksichtigt. Auch der Wochentag hat keine Auswirkung auf die Anzahl der Fehler, da Fehler an jedem Tag der Woche auftreten können. Allerdings spielt die Uhrzeit eine Rolle, weil diese den Kontext zu einer kontextuellen Anomalie liefert. Es gibt Meldungen, die in einem gewissen Zeitraum vorkommen dürfen, ohne dass dies einen Fehler bedeutet. In einem anderen Zeitraum dürfen diese Meldungen allerdings nicht auftreten. Aus diesem Grund wird aus der Uhrzeit die Informationen zur Stunde extrahiert und diese als Feature verwendet.

In den vorliegenden Daten können aktuell nur Meldungen mit Loglevel *ERROR* oder *WARN* einen Fehler bedeuten. Es wäre deshalb denkbar, die Daten vor dem Training entsprechend zu filtern. Dieser Ansatz wurde jedoch aus unterschiedlichen Gründen nicht verfolgt. Zum einen können auch die Meldungen mit Loglevel *INFO* und *DEBUG* wichtige Informationen über den Programmablauf enthalten. Zum anderen könnte es in zukünftigen Daten auch vorkommen, dass eine Abfolge solcher Meldungen auf einen inkorrekten Programmablauf hindeutet.

In Xu et al. [Xu+10] basieren die verwendeten Features auf Zustandsvariablen und Identifikatoren, die aus den Parametern extrahiert werden. Dies wird an folgendem Beispiel (4.5) [Xu+10] erläutert:

Log-Meldung 1:	*starting: xact 325 is COMMITTING*	
Log-Meldung 2:	*starting: xact 346 is ABORTING*	(4.5)

Zustandsvariablen sind Labels, die eine Menge möglicher Zustände beschreiben, die ein Parameter annehmen kann, wie beispielsweise *ABORTING* und *COMMITTING* in Beispiel 4.5. Identifikatoren sind Parameter, mit denen ein Objekt identifiziert wird, das vom Programm verändert wurde. Beispiele hierfür können unter anderem Transaktions-IDs (wie *325* und *346* in Beispiel 4.5) sein. [Xu+10]

Zustandsvariablen besitzen üblicherweise wenig einzigartige Werte, da ein Objekt nur eine begrenzte Anzahl von Zuständen annehmen kann. Dahingegen können Identifikatoren viele unterschiedliche Werte annehmen. Somit kann anhand der Häufigkeit der Parameter bestimmt werden, ob es sich um eine Zustandsvariable oder einen Identifikator handelt. [Xu+10]

Aus den Zustandsvariablen wird ein Vektor erstellt, in dem das Verhältnis der Zustände dokumentiert wird. Als Beispiel wird an dieser Stelle aufgeführt, dass das Verhältnis der Zustände *ABORTING* und *COMMITTING* im normalen Programmverhalten stabil ist, sich jedoch im Fehlerfall stark ändert. Dabei ist nicht die tatsächliche Anzahl der Zustände wichtig, die von der Auslastung des Systems abhängt, sondern nur das Verhältnis zwischen den unterschiedlichen Werten. Die Identifikatoren werden genutzt, um die Log-Meldungen zu gruppieren und so beispielsweise in Abhängigkeit der Transaktions-ID getrennt zu analysieren. [Xu+10]

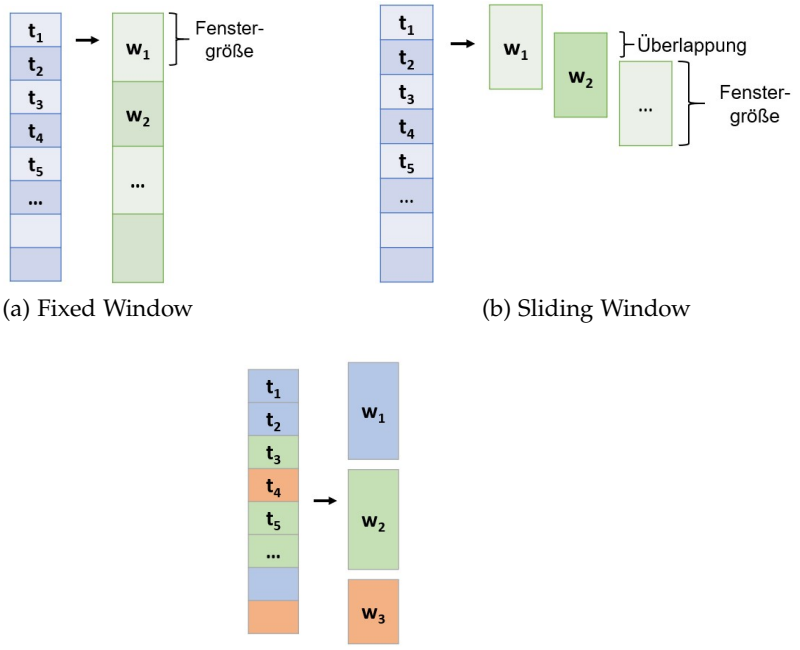
In den Daten, die im Rahmen dieser Arbeit verwendet werden, existieren jedoch keine Identifikatoren. Es wäre zwar denkbar, die Synchronisation pro Mitarbeiter*in getrennt zu analysieren, jedoch lassen sich die entsprechenden Kennungen nicht automatisiert aus den Log-Dateien extrahieren. Dies wird im Folgenden an einem Beispiel (4.6) veranschaulicht.

Log-Meldung:	*Updating aaa in bbb's folder*	
Parameter 1:	*aaa*	
Parameter 2:	*bbb's*	(4.6)

Wie zu sehen ist, besteht die Nachricht aus zwei Parametern, die beide einer Kennung entsprechen. Die erste Kennung (*aaa*) steht für Mitarbeiter*in A, dessen/deren Kontakt aktualisiert wird. Die zweite Kennung (*bbb*) entspricht Mitarbeiter*in B, dessen/deren Kontakteordner synchronisiert wird. Es wird deutlich, dass ohne ein umfangreiches Expert*innenwissen, keine Unterscheidung der Parameter vorgenommen werden kann. Die Synchronisation wird für alle Mitarbeiter*innen vorgenommen und dabei werden pro Mitarbeiter*in wiederum die Kontakte aller Mitarbeiter*innen synchronisiert. Aus diesem Grund können auch anhand der Häufigkeit der Werte keine Identifikatoren erkannt werden. Des Weiteren können auch keine Zustandsvariablen identifiziert werden, da in den Log-Dateien keine Zustände beschrieben werden. Dies muss bei der Konzeption der Modelle berücksichtigt werden.

In diversen Modellen zur Anomaliedetektion werden die Anomalien nicht pro Log-Meldung geprüft, sondern es wird ein Gruppe von Log-Zeilen analysiert. [Xu+10] [Lou+10] [Lin+16] [Lan+18] Zur Erstellung der Gruppen stehen drei verschiedene Methoden zur Verfügung, die in Abbildung 4.5 veranschaulicht werden: *Fixed Window*, *Sliding Window* und *Session Window*. [He+16]

Fixed Window (4.5a) und *Sliding Window* (4.5b) basieren auf der Uhrzeit. Die Größe des Fensters entspricht somit einer festen Zeitspanne, die immer konstant ist, beispielsweise eine Stunde. Beim *Sliding Window* überlappen sich, im Gegensatz zum *Fixed Window*, die Fenster. Die Überlappung hängt von der gewählten Schrittgröße ab und ist in der Regel kleiner als die Fenstergröße. Beispielsweise kann bei einer Fenstergröße von einer Stunde eine Schrittgröße von 30 Minuten gewählt werden. Innerhalb des Zeitfensters wird gezählt, wie häufig die einzelnen Log-Keys auftreten und somit ein



(a) Fixed Window

(b) Sliding Window

(c) Session Window

Abbildung 4.5: Vergleich von Window-Techniken

Event-Count-Vektor gewonnen. Alle Fenster bzw. Vektoren zusammen ergeben dann eine *Event-Count-Matrix*, die für das Training der Modelle verwendet werden kann.

Session Windows (4.5c) hingegen basieren nicht auf der Uhrzeit, sondern die Fenster ergeben sich durch eine Gruppierung anhand eines Identifikators, wie beispielsweise einer Transaktions-ID. Wie bereits erläutert, können in dieser Arbeit keine Identifikatoren aus den Daten gewonnen werden, weshalb die Verwendung von *Session Windows* nicht möglich ist.

Bei der Verwendung der Event-Count-Matrix verschiebt sich der Fokus wie beschrieben von einer zeilenbasierten Analyse der Log-Dateien auf einen festgelegten Zeitraum. In Xu et al. [Xu+10] wird argumentiert, dass eine Gruppe von Meldungen einen besseren Indikator zur Fehleranalyse darstellt als eine einzelne Meldung, weil manche Anomalien sich nur durch unvollständige Sequenzen von Meldungen äußern. Dies trifft auch für kollektive Anomalien zu, bei denen eine einzelne Meldung keinen Fehler darstellt, sondern erst eine längere Sequenz dieser Meldungen. Für Punktanomalien ist diese Aussage jedoch nicht korrekt, da hier einzelne Meldung unabhängig von anderen einen Fehler bedeuten.

Bei der Erstellung der Event-Count-Matrix geht zudem die Information zur Uhrzeit verloren, die den Kontext für kontextuelle Anomalien darstellt. Eine Anforderung an das Modell ist, dass alle Arten von Fehlern erkannt werden müssen, weshalb sich der Einsatz der Event-Count-Matrix für diesen Anwendungsfall nicht eignet. Allerdings wäre es denkbar, die Event-Count-Matrix zusätzlich zur zeilenbasierten Analyse hinzuziehen. Dies würde die Vorteile beider Ansätze vereinen. Im Kontext dieser Arbeit sind der Großteil

der Fehler Punktanomalien, kollektive Anomalien existieren nicht in den Daten und somit kann die zusätzliche Verwendung der Event-Count-Matrix nicht evaluiert werden. Aus diesem Grund wurde sich für eine zeilenbasierte Analyse der Log-Dateien entschieden. Die Verwendung von einer Kombination der Möglichkeiten wird jedoch im Ausblick diskutiert.

4.3.2 *Supervised-Anomaliedetektion*

Wie in Kapitel 2.2.2 beschrieben, erzielen Supervised-Verfahren häufig bessere Vorhersagen als Semi-Supervised- oder Unsupervised-Verfahren, weil durch die Labels mehr Informationen mit in das Modell gegeben werden können. Allerdings liegen in der Praxis nur sehr selten Labels vor und der Prozess zur Vergabe der Labels ist häufig mit einem sehr hohen Aufwand verbunden. [MG16] Im Rahmen dieser Arbeit steht, wie in Kapitel 4.1.1 beschrieben, ein gelabelter Datensatz zur Verfügung, somit können Supervised-Verfahren verwendet werden.

Klassische Verfahren

Im Kontext dieser Arbeit kann die Fehlererkennung allgemein als binäres Klassifizierungsproblem betrachtet werden, bei dem vorhergesagt wird, ob es sich bei einer Log-Zeile um einen Fehler handelt oder nicht.

Zu den klassischen Verfahren, die in diesem Bereich häufig eingesetzt werden, gehören die Support Vector Machine (SVM), die logistische Regression und der Decision Tree. Diese Modelle werden auch in He et al. [He+16] für eine Log-Analyse zur Anomaliedetektion eingesetzt. Für das Training wird dort die Event-Count-Matrix verwendet. Alle drei Modelle konnten auf den dort verwendeten Daten gute Ergebnisse erzielen und werden deshalb auch in dieser Arbeit untersucht. Dabei werden aus den oben genannten Gründen statt einer Event-Count-Matrix die Features der einzelnen Log-Zeilen verwendet.

Long Short-Term Memory

Die Problemstellung der Identifikation von Fehlern basierend auf Log-Meldungen kann auch als solche forumliert werden, in der Sequenzen von Wörtern eines festen Vokabulars Wahrscheinlichkeiten zugeordnet werden. Dies ist bekannt aus dem Bereich des Natural Language Processing (NLP). Übertragen auf den Kontext dieser Arbeit kann ein Log-Key als Wort des Vokabulars V angesehen werden. V beschreibt dabei alle möglichen Log-Keys.

Ein etabliertes und häufig verwendetes Modell im Bereich von NLP und Zeitreihenanalyse ist das Long Short-Term Memory (LSTM). [Du+17] [Gre+17] Ursprünglich von Hochreiter et al. [HS97] vorgestellt und anschließend von Gers et al. [GSC00] weiterentwickelt, eignet es sich besonders dafür, Muster in sequentiellen und zeitlichen Daten zu lernen. [Gre+17]

Die Architektur eines LSTM wird in Abbildung 4.6 gezeigt. Die Funktions-

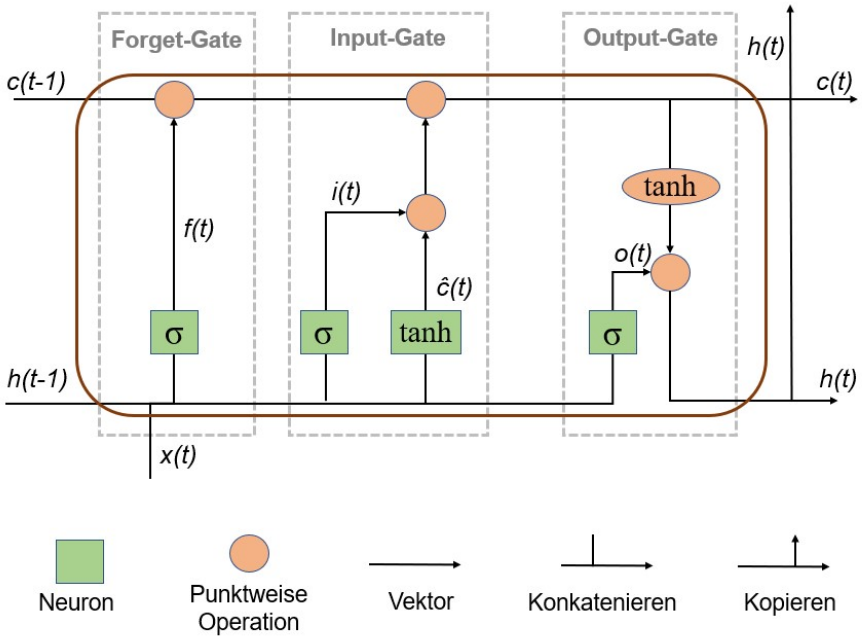


Abbildung 4.6: Architektur eines LSTM (nach [Yu+19])

weise lässt sich mathematisch durch folgende Gleichungen darstellen:

$$
\begin{aligned}
f(t) &= \sigma(W_{fh}h_{t-1} + W_{fx}x_t + b_f) && \text{(Forget-Gate)}\\
f(t) &= \sigma(W_{fh}h_{t-1} + W_{fx}x_t + b_f) && \text{(Forget-Gate)}\\
i(t) &= \sigma(W_{ih}h_{t-1} + W_{ix}x_t + b_i) && \text{(Input-Gate)}\\
\hat{c}(t) &= tanh(W_{\hat{c}h}h_{t-1} + W_{\hat{c}x}x_t + b_{\hat{c}}) && \text{(Kandidaten für Zellstatus)}\\
c(t) &= f(t) \cdot c(t-1) + i(t) \cdot \hat{c}(t) && \text{(Zellstatus)}\\
o(t) &= \sigma(W_{oh}h_{t-1} + W_{ox}x_t + b_o) && \text{(Output-Gate)}\\
h(t) &= o(t) \cdot tanh(c(t)) && \text{(Versteckter Zellstatus)}
\end{aligned}
\tag{4.7}
$$

Dabei ist x_t der aktuelle Teil der Sequenz, W_f, W_h, W_i und $W_{\hat{c}}$ sind die jeweiligen Gewichte sowie b_f, b_h, b_i und $b_{\hat{c}}$ die Bias-Vektoren. Der Operator $\cdot$ steht für die punktweise Mulitplikation zweier Vektoren. σ und $tanh$ sind die Aktivierungsfunktionen, die sich wie folgt berechnen lassen:

$$
\begin{aligned}
\sigma(z) &= \frac{1}{1+e^{-z}}\\
tanh(z) &= \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}
\end{aligned}
\tag{4.8}
$$

Wird der Status einer Zelle aktualisiert, wird mittels des Forget-Gates entschieden, welche Information des Zellstatus behalten wird. $f(t) = 1$ bedeutet, dass die Informationen behalten werden, während bei einem $f(t) = 0$ alle Informationen gelöscht werden. Im Input-Gate wird entschieden, welche neuen Informationen im Zellstatus gespeichert werden und im Output-Gate

wird entschieden, welche Informationen basierend auf dem aktuellen Zellstatus als Output weitergegeben werden. [Yu+19]

In Du et al. [Du+17] wird die Annahme getroffen, dass die Zahl an Print-Statements im Sourcecode konstant ist. Dadurch ist auch die Anzahl an Log-Keys, n, konstant und somit die Größe des Vokabulars V, mit $V = v_1, v_2, ..., v_n$ der Menge an möglichen Log-Keys. Nach dem Log-Parsing liegen Sequenzen von Log-Keys vor, die die Ausführung des Programms widerspiegeln. Dabei ist k_i der Log-Key an der Position i in der Log-Key-Sequenz. Zudem ist k_i einer der möglichen n Log-Keys aus V. Durch die Sequenzen wird der Programmfluss des Microservice dargestellt und deshalb ist k_i stark von den m vorherigen Log-Keys der Sequenz, also $k_{i-1}, ..., k_{i-m}$, abhängig.

Mittels eines LSTM kann ein Modell zur Anomaliedetektion trainiert werden, das diese Sequenzen lernt. Der Input ist dabei eine Sequenz von Log-Keys und der Output die Wahrscheinlichkeit, dass es sich beim letzten Log-Key der Sequenz um einen Fehler handelt. Dieses Prinzip wird in Abbildung 4.7 veranschaulicht.

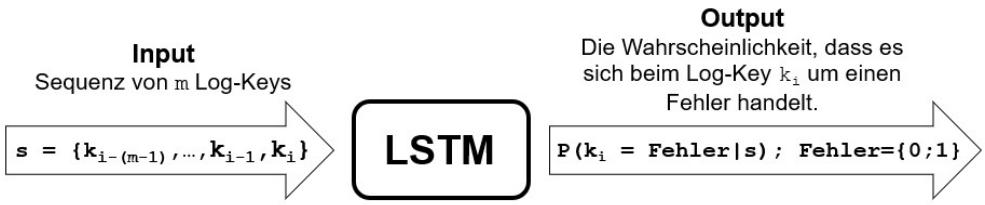


Abbildung 4.7: Supervised-LSTM zur Anomaliedetektion (nach [Du+17])

Es sei beispielsweise folgende Sequenz von Log-Keys gegeben, in der die Fehler rot markiert sind:

$$\{v_1, v_{12}, v_4, v_7, v_9, v_{11}, v_3\} \tag{4.9}$$

Bei einem $m = 3$ würden für das Training des LSTM folgende Kombinationen aus Input-Sequenz und Label verwendet werden, wobei eine 0 dem Label *Kein Fehler* entspricht und eine 1 dem Label *Fehler*:

$$\begin{aligned}
\{v_1, v_{12}, v_4\} &\Rightarrow 0 \\
\{v_{12}, v_4, v_7\} &\Rightarrow 1 \\
\{v_4, v_7, v_9\} &\Rightarrow 0 \\
\{v_7, v_9, v_{11}\} &\Rightarrow 0 \\
\{v_9, v_{11}, v_3\} &\Rightarrow 1
\end{aligned} \tag{4.10}$$

Die optimale Länge der Input-Sequenzen, m, wird in Kapitel 5 evaluiert.

Neben der klassischen Variante kann dieses Modell auch als bidirektionales LSTM eingesetzt werden. Bei der bidirektionalen Variante werden nicht nur die vorhergehenden Log-Keys, sondern auch die nachfolgenden berücksichtigt. Im betrachteten Anwendungsfall ist dies möglich, da die Analyse

der Log-Dateien wöchentlich erfolgt und somit alle benötigten Log-Zeilen vorliegen.

4.3.3 *Semi-Supervised-Anomaliedetektion*

Beim Semi-Supervised Learning zur Anomaliedetektion wird vorausgesetzt, dass alle Instanzen des Trainings-Datensatzes der normalen Klasse angehören und somit keine Anomalien beinhalten. Die vorliegenden Log-Dateien beinhalten, wie in Kapitel 4.1.1 beschrieben, drei fehlerfreie Wochen und eignen sich somit für den Einsatz von Semi-Supervised Learning.

Long Short-Term Memory

Das LSTM kann nicht nur als Supervised-Verfahren verwendet werden, sondern es eignet sich auch für den Einsatz zur Semi-Supervised-Anomaliedetektion, wie in Du et al. [Du+17] beschrieben. Für das Training werden dabei ausschließlich Daten aus dem Normalzustand verwendet, was bedeutet, dass das Modell den fehlerfreien Ablauf des Systems lernt.

Im Gegensatz zum Einsatz des LSTM als Supervised-Verfahren stehen die Labels hier nicht zur Verfügung. Deshalb wird die Problemstellung im Kontext der Multiklassifikation betrachtet, bei der jeder Log-Key eine eigene Klasse darstellt. Der Input ist weiterhin eine Sequenz von Log-Keys mit der Länge m, der Output nun jedoch eine Wahrscheinlichkeitsverteilung über die n Log-Keys aus V, die die Wahrscheinlichkeit darstellt, dass der nächste Log-Key der Key $v_j \in V$ ist. Das Prinzip wird in Abbildung 4.8 veranschaulicht.

Abbildung 4.8: Semi-Supervised-LSTM zur Anomaliedetektion (nach [Du+17])

Es sei beispielsweise folgende Sequenz von Log-Keys gegeben:

$$\{v_1, v_{12}, v_4, v_7, v_9, v_{11}, v_3\} \tag{4.11}$$

Bei einem $m = 3$ würden für das Training des LSTM folgende Kombinationen aus Input-Sequenz und Label verwendet werden:

$$\begin{aligned} \{v_1, v_{12}, v_4\} &\Rightarrow v_7 \\ \{v_{12}, v_4, v_7\} &\Rightarrow v_9 \\ \{v_4, v_7, v_9\} &\Rightarrow v_{11} \\ \{v_7, v_9, v_{11}\} &\Rightarrow v_3 \end{aligned} \tag{4.12}$$

Die optimale Länge der Input-Sequenzen, m, wird in Kapitel 5 evaluiert.

Liegt der tatsächliche Log-Key nicht innerhalb der x Log-Keys mit der höchsten Wahrscheinlichkeit, bedeutet das einen Fehler, andernfalls entspricht es dem normalen Programmverhalten. [Du+17] Das optimale x wird im folgenden Kapitel 5 ermittelt.

Das Vorgehen in Du et al. [Du+17] sieht vor, dass jede Log-Zeile in den Log-Key und die Liste der zugehörigen Parameter unterteilt wird. Anschließend werden Sequenzen von Log-Keys verwendet, um mittels eines LSTM wie beschrieben ein Modell zur Anomaliedetektion zu trainieren. Zusätzlich werden auch die Parameter zur Anomaliedetektion hinzugezogen. Wird anhand eines Log-Keys keine Anomalie erkannt, werden im nächsten Schritt die Parameter geprüft. [Du+17] Wie bereits erläutert, können in dieser Arbeit anhand der Parameter keine Zustandsvariablen identifiziert werden, die einen Rückschluss auf Anomalien erlauben. Aus diesem Grund werden nur die Log-Keys zur Anomaliedetektion verwendet.

Wie beim Einsatz des LSTM als Supervised-Verfahren kann auch für die Semi-Supervised-Anomaliedetektion sowohl die klassische Variante als auch das bidirektionale Modell zum Einsatz kommen.

One-Class Support Vector Machine

Die One-Class Support Vector Machine zur Anomaliedetektion ist eine Variante der SVM, bei der angenommen wird, dass alle Trainingsdaten nur ein Label haben, im Kontext dieser Arbeit das Label *Kein Fehler*. Somit lernt die One-Class SVM anhand der normalen Daten eine Grenze um diese. Alle Daten, die außerhalb der erlernten Grenze liegen, werden als Anomalien deklariert. [CBK09] Dieses Modell ist ein Verteter des One-Class Learnings, das für unbalancierte Daten geeignet ist.

Für dieses Modell muss jedoch anhand von numerischen Werten eine Ähnlichkeit messbar sein, um die Grenze um die normalen Daten bestimmen zu können. Im vorliegenden Datensatz gibt es keine numerischen Werte, die verglichen werden können. Der Unterschied zwischen den Log-Meldungen auf Basis des String-Vergleichs, der mittels der Levenshtein-Distanz oder des LCS-Algorithmus berechnet werden kann, lässt keinen Rückschluss dar-

auf zu, ob es sich bei einer Meldung um einen Fehler handelt, wie folgendes Beispiel veranschaulicht:

Kein Fehler:	*CoreHttpProvider[send] - 203 [...]*
Fehler:	*CoreHttpProvider[send] - 203 Error during http request*

(4.13)

Wie zu erkennen ist, sind die Meldungen sich relativ ähnlich, jedoch bedeutet die erste Meldung keinen Fehler, die zweite hingegen schon.

Der Einsatz der Event-Count-Matrix ist für dieses Modell geeignet, jedoch können damit, wie erläutert, keine kontextuellen Anomalien erkannt werden. Zudem wird angenommen, dass einzelne Punktanomalien keine signifikanten Änderungen in den Daten erzeugen. Diese Annahme könnte in zukünftigen Arbeiten überprüft werden. Mit dem LSTM wurde jedoch bereits ein Semi-Supervised-Verfahren identifiziert, das alle Anforderungen erfüllt, weshalb die One-Class SVM hier nicht berücksichtigt wird.

4.3.4 *Unsupervised-Anomaliedetektion*

Der große Vorteil des Unsupervised Learnings liegt darin, dass für das Training keine Labels benötigt werden. Jedoch sind Unsupervised-Verfahren häufig schwieriger zu erstellen und zu evaluieren. [MG16] Wie in Kapitel 2.2.2 beschrieben, wird Unsupervised Learning im Kontext der Anomaliedetektion wie folgt betrachtet: Die Daten werden verglichen und Muster gesucht. Die Datenpunkte, die von diesen Mustern abweichen, werden als Anomalie deklariert.

In der Literatur werden unterschiedliche Verfahren des Unsupervised Learnings im Kontext der Logfile-Analyse beschrieben. Diese werden im Folgenden erläutert und für jedes Verfahren wird die Anwendbarkeit auf die Daten sowie die Erfüllung der Anforderungen geprüft.

Cluster-Evolution

In Landauer et al. [Lan+18] wird das Verfahren der Cluster-Evolution vorgestellt, in der für feste Zeitfenster Cluster-Maps erstellt werden und die Veränderung dieser Maps mit Methoden der Zeitreihenanalyse untersucht wird. Der Ablauf ist in Abbildung 4.9 veranschaulicht. Die Schritte 1-4 können als Clustering-Verfahren zusammengefasst werden. Zunächst werden die Log-Zeilen eingelesen und an den Vorverarbeitungsschritt übergeben. An dieser Stelle können die Strings bereinigt werden, wie beispielsweise durch das Ersetzen nicht darstellbarer Zeichen oder das Entfernen von Zeichen, die außerhalb des Wertebereichs 32-126 der ASCII-Tabelle liegen. Anschließend wird iterativ eine Cluster-Map aufgebaut, indem jede neue Log-Zeile entweder einem bestehenden Cluster zugeordnet wird oder ein neues erzeugt. Die Einordnung in Cluster erfolgt mittels eines String-Vergleichs auf Basis der Levenshtein-Distanz. Jede Cluster-Map ist dabei nur ein statisches Bild des aktuellen Zeitfensters. Deshalb werden, ebenfalls mittels der Levenshtein-

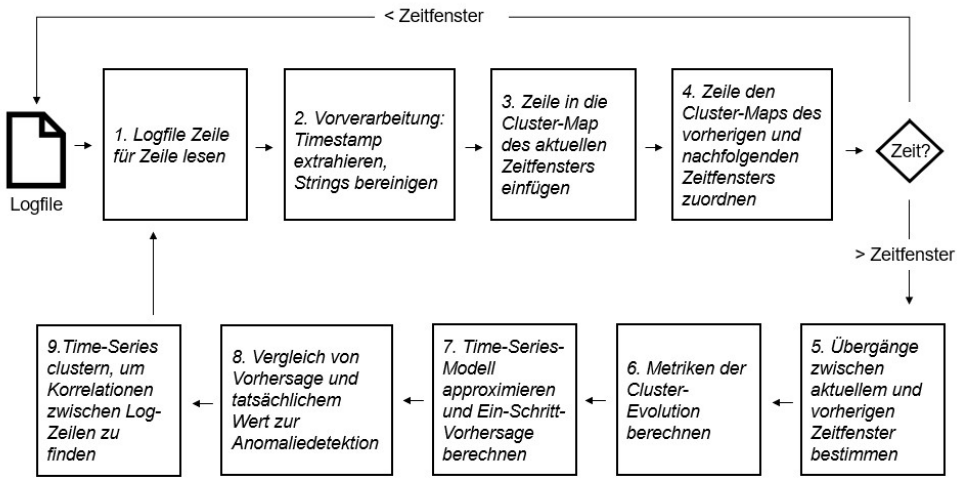


Abbildung 4.9: Ablauf der Anomaliedetektion mittels Cluster-Evolution (nach [Lan+18])

Distanz, die Verbindungen zu den Cluster-Maps der benachbarten Zeitfenster allokiert. [Lan+18]

Nachdem der Prozess des Clustering-Verfahrens für ein Zeitfenster durchlaufen wurde, werden in Schritt 5 mithilfe der Allokationen aus Schritt 4 die Beziehungen zwischen den Clustern benachbarter Zeitfenster analysiert. Dadurch kann bestimmt werden, welches Cluster aus dem aktuellen Zeitfenster aus welchem Cluster des vorherigen Zeitfensters resultiert. Für diese Beziehung werden in Schritt 6 Mertiken der Cluster-Evolution berechnet. Ein Beispiel für eine solche Metrik ist die Information, ob ein Cluster stabil ist, was bedeutet, dass die Log-Zeilen, die zum aktuellen Cluster gehören, sich im gleichen Cluster des vorherigen Zeitfensters wiederfinden. Die Metriken werden anschließend in Schritt 7 für die Erstellung eines Modells für die Zeitreihenanalyse verwendet und eine Ein-Schritt-Vorhersage für die Metriken des nachfolgenden Zeitfensters getroffen. Die Vorhersage der Metrikwerte aus dem vorherigen Zeitfenster wird in Schritt 8 zur Anomaliedetketion verwendet. Weicht der tatsächlich berechnete Wert zu weit von dem vorhergesagten ab, wird dies als Anomalie deklariert. Abschließend werden in Schritt 9 die Korrelationen zwischen den Zeitreihen überwacht. Unerwartete Änderungen dieser Korrelationen indizieren ein unerwartetes Verhalten des Systems und somit eine Anomalie. [Lan+18]

Das beschriebene Verfahren wird allerdings im Kontext von Intrusion-Detection-Systemen angewendet. Im Szenario, das dort als Beispiel referenziert wird, wird das Surfverhalten eines Angreifers analysiert, der sich mittels Social Engineerings unbefugten Zugang beschafft hat. Die Aktionen des Angreifers stimmen in der Häufigkeit nicht mit dem Gesamtverhalten der anderen Nutzer überein. Dies führt dazu, dass die Cluster sich stark vergrößern, was eine Anomalie bedeutet. [Lan+18] Bei den Log-Dateien des Mircoservice erlaubt eine erhöhte Auslastung wie beschrieben jedoch keinen

Rückschluss auf einen Fehler. Zudem werden bei diesem Verfahren keine weiteren Informationen, wie z.B. die Uhrzeit, berücksichtigt, weshalb keine Erkennung von kontextuellen Anomalien möglich ist. Somit wird dieser Ansatz, auch wenn er in anderen Anwendungsbereichen sehr nützlich sein kann, aus den genannten Gründen in dieser Arbeit nicht verwendet.

Log-Clustering

In Lin et al. [Lin+16] wird der Ansatz des Log-Clusterings beschrieben. Der Ablauf wird in Abbildung 4.10 gezeigt. Es wird zwischen zwei Phasen unter-

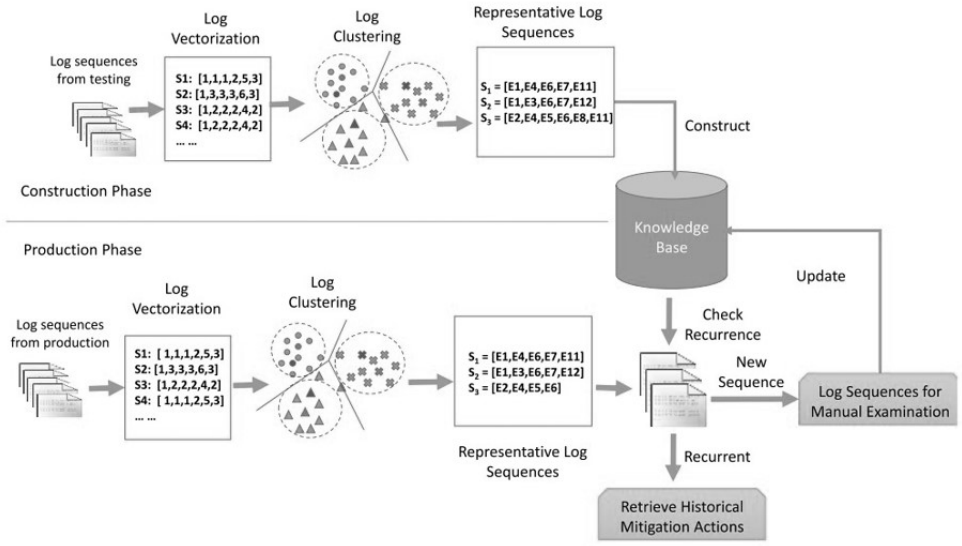


Abbildung 4.10: Ablauf des Log-Clusterings [Lin+16]

schieden, der *Construction Phase* und der *Production Phase*. In der Construction Phase werden Log-Sequenzen einer Testumgebung verwendet, um Cluster zu bilden. Zunächst werden die Sequenzen in Vektoren konvertiert und in Cluster eingeordnet. Für jedes Cluster wird von Expert*innen eine repräsentative Sequenz ausgesucht und diese als Wissensgrundlage in der sogenannten *Knowledge Base* gespeichert. Anschließend werden in der Production Phase Log-Sequenzen aus der Produktionsumgebung analysiert. Es erfolgt zunächst die Vektorisierung und das Clustering der Sequenzen, analog zur Construction Phase. Dann werden wiederum die repräsentativen Sequenzen pro Cluster ermittelt und mit der Knowledge Base verglichen. Somit müssen Entwickler*innen nur jene repräsentativen Log-Sequenzen manuell untersuchen, die nicht in der Knowledge Base zu finden sind, welche kontinuierlich aktualisiert wird.

Der Aufbau der Knowledge Base verlangt allerdings das Labeling von Expert*innen. Somit kann dieses Verfahren, auch wenn es in He et al. [He+16] als solches deklariert wurde, nicht als Unsupervised Learning bezeichnet werden. Es könnte als Unterstützung für das Labeling verwendet werden, dabei muss jedoch berücksichtigt werden, dass keine Erkennung von kontextuellen oder kollektiven Anomalien möglich ist.

Im Rahmen dieser Arbeit wurde das Labeling bereits vorgenommen. Zudem kann das Modell weder kollektive noch kontextuelle Anomalien erkennen, weshalb das Log-Clustering in dieser Arbeit nicht eingesetzt wird.

Principal Component Analysis

In Xu et al. [Xu+10] wird die Principal Component Analysis (PCA) als Unsupervised-Verfahren eingesetzt. PCA bzw. Hauptkomponentenanalyse ist eine statistische Methode zur Dimensionsreduktion, die Muster in hochdimensionalen Daten sucht. Dies geschieht, indem die Daten in ein neues Koordinatensystem projiziert werden, das aus k Dimensionen besteht, den k Hauptkomponenten. Sie reflektieren die Kovarianz zwischen den ursprünglichen Koordinaten. Dabei ist k kleiner als die Anzahl der Dimensionen in den Ausgangsdaten. [Xu+10]

Um die k Hauptkomponenten zu berechnen, werden die Komponenten gesucht, die die größte Varianz zwischen den hochdimensionalen Daten erfassen, wodurch möglichst viel Informationen der ursprünglichen Daten erhalten bleibt. [He+16]

Für die Anomaliedetektion wird eine Event-Count-Matrix erstellt und mittels PCA die Muster in den Dimensionen zwischen den einzelnen Zeilen der Matrix, den Event-Count-Vektoren, gesucht. Dafür wird im ursprünglichen n-dimensionalen Raum der „normale“ k-dimensionale Raum S_d erstellt. Die übrigen $(n-k)$ Dimensionen formen den „abnormalen“ Raum S_α.

Um zu bestimmen, ob ein Event-Count-Vektor y abnorml ist, wird dessen Distanz zum normalen Raum S_d berechnet. Dies geschieht mittles des quadratischen Vorhersagefehlers, der sich wie folgt berechnet:

$$\|y_\alpha\|^2 \tag{4.14}$$

Dabei ist y_α die Projektion eines Event-Count-Vektors y zu S_α und wird durch folgende Gleichung berechent:

$$y_\alpha = (1 - PP^T)y \tag{4.15}$$

$P = [v_1, v_2, ..., v_k]$ sind dabei die ersten k Hauptkomponenten.

Ein Event-Count-Vektor y wird als abnormal deklariert, wenn der quadratische Vorhersage-Fehlers einen Threshold Q_α überschreitet:

$$\|y_\alpha\|^2 > Q_\alpha \tag{4.16}$$

Der Threshold Q_α ist die Residuen-Funktion des Vohersagefehlers mit einem Konfidenz-Level von $(1-\alpha)$. Für die Wahl von α wird sich in Xu et al. [Xu+10] an verwandten Arbeiten orientiert und der Standard-Empfehlung von $\alpha = 0,001$ gefolgt.

Wie in Kapitel 4.3.1 beschrieben, werden in Xu et al. [Xu+10] Zustandsvariablen und Identifikatoren für den Aufbau der Event-Count-Matrix verwendet, was im Kontext dieser Arbeit nicht möglich ist. Im Rahmen dieser Arbeit kann die Event-Count-Matrix deshalb nur basierend auf den Daten

der *Fixed Windows* bzw. *Sliding Windows* erstellt werden, wie es auch in He et al. [He+16] umgesetzt wird. Allerdings wird dort auch darauf hingewiesen, dass die PCA sehr sensitiv gegenüber Veränderungen in den Daten ist und die Qualität der Vorhersage, in Abhängigkeit der verwendeten Datensätze, stark variiert. [He+16] Die Werte der Event-Count-Matrix müssen hier skaliert werden, damit die Ergebnisse der PCA nicht von der Auslastung des Systems abhängen. Jeder Wert eines Event-Count-Vektors, was einer Zeile der Matrix entspricht, wird durch die Anzahl der Elemente der Zeile geteilt, wie in folgendem Beispiel veranschaulicht:

Zeile vor Skalierung:	[1, 10, 2, 5, 2]
Anzahl Elemente:	1 + 10 + 2 + 5 +2 = 20
Zeile nach Skalierung:	[0.05, 0.5, 0.1, 0.25, 0.1]

Durch den Einsatz der Event-Count-Matrix geht jedoch, wie in Kapitel 4.3.1 erläutert, die Information zur Uhrzeit verloren, weshalb keine kontextuellen Anomlien erkannt werden können. Für deren korrekte Identifizierung wäre eine nachträgliche Prüfung der Uhrzeit benötigt. Zudem erzeugen einzelne Punktanomalien keine signifikanten Änderungen in den Daten, sodass diese nicht mittels der PCA identifiziert werden können. Es können jedoch wie beschrieben keine anderen numerischen Features aus den Daten extrahiert werden, die einen Rückschluss auf ein fehlerhaftes Programverhalten geben. Aus diesem Grund erfüllt die PCA nicht alle definierten Anforderungen.

Invariants Mining

In Lou et al. [Lou+10] wird *Invariants Mining* verwendet, um Fehler erkennen zu können. Allgemein bezeichnen die Invarianten eines Programms Bedingungen, die immer vor und nach der Ausführung erfüllt werden, unabhängig vom Input oder der Auslastung. Sie können für unterschiedliche Aspekte definiert werden, wie beispielsweise System-Metriken wie die CPU oder auch mittels Programm-Variablen. Zudem können auch aus dem Ablauf eines Programms Invarianten abgeleitet werden. [Lou+10]

Beispielsweise wird eine Datei im normalen Programmablauf wieder geschlossen, nachdem sie geöffnet wurde. Somit sollte die Anzahl der Meldungen zu „Open File“ mit der Anzahl von „Close File“ übereinstimmen. Ist dies nicht der Fall, ist die Invariante nicht erfüllt, was auf einen Fehler im Programm hindeutet. [He+16]

In Lou et al. [Lou+10] wird die Annahme getroffen, dass Log-Meldungen ausreichend Informationen über die Ausführung eines Systems bereitstellen, weshalb mittels Analyse dieser Meldungen Invarianten des Programms ermittelt werden können. [Lou+10] Ein Beispiel zeigt Abbildung 4.11. Dabei

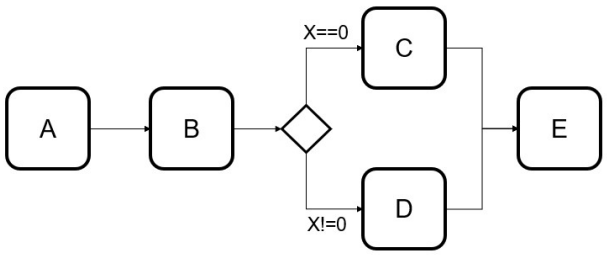


Abbildung 4.11: Beispielhafter Programmablauf (nach [Lou+10])

sollten, unabhängig von der Ausführung des Programms, beispielsweise folgende Gleichungen immer erfüllt werden:

$$c(A) = c(B) = c(E)$$
$$c(B) = c(C) + c(D) \tag{4.17}$$

$c(A), ..., c(E)$ entsprechen der Anzahl der Log-Meldungen $A, ..., E$. Die Gleichungen korrespondieren mit den jeweiligen Invarianten des Programmablaufs und ihre Gültigkeit wird nicht vom Input oder der Auslastung beeinflusst. Invariants Mining eignet sich deshalb als Methode zur Anomaliedetektion, weil Fehler meist zu einem anderen Programmablauf führen als im Normalzustand. Eine Verletzung der Invarianten bedeutet somit eine Anomalie. [Lou+10]

Der Ablauf der Methode wird in Abbildung 4.12 veranschaulicht.

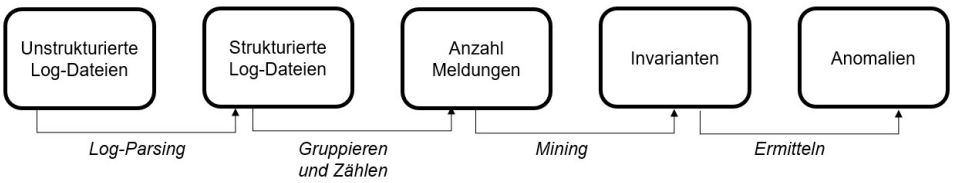


Abbildung 4.12: Ablauf des Invariants Mining (nach [Lou+10])

Beim Log-Parsing erfolgt die Trennung von Log-Key und Parametern. Anschließend werden Identifikatoren ermittelt und anhand dieser das Gruppieren vorgenommen und pro Gruppe die Anzahl Meldungen jedes Log-Keys gezählt, wodurch ein Message-Count-Vektor entsteht. Mithilfe dieses Vektors werden die Invarianten ermittelt. Eine Gruppe von Log-Meldungen, die diese Invarianten verletzt, gilt als Anomalie. [Lou+10]

Die Invarianten werden über einen sogenannten *Invariant Space*, zu Deutsch Invarianten-Raum, ermittelt. Eine invariante lineare Beziehung kann als in-

variante Gleichung dargestellt werden. Bei n unterschiedlichen Log-Keys wird diese Beziehung wie folgt formuliert:

$$a_0 + a_1 x_1 + ... + a_n x_n = 0 \tag{4.18}$$

x_i beschreibt dabei die Anzahl der Log-Meldungen, deren Log-Key-Index i ist. Die Koeffizienten der Gleichung können durch den Vektor θ repräsentiert werden, der sich wie folgt definiert:

$$\theta = [a_0, a_1, ..., a_n]^T \tag{4.19}$$

Eine Invariante kann mithilfe von θ dargestellt werden. Beispielsweise entspricht die zweite Gleichung aus 4.17 folgendem Vektor:

$$\theta = [0, 0, 1, -1, -1, 0]^T \tag{4.20}$$

Dabei besitzen die Log-Keys $A - E$ jeweils den Index $1 - 5$. Unabhängige Vektoren können mit unabhängigen linearen Beziehungen korrespondieren und stellen somit unterschiedliche Invarianten dar. [Lou+10]

Die Invarianten werden mithilfe der Event-Count-Matrix dargestellt. Jede Zeile der Matrix repräsentiert, wie in Kapitel 4.3.1 erläutert, die Anzahl der unterschiedlichen Log-Keys in einem gewissen Zeitraum. Diese Anzahl kann durch x_{ij} dargestellt werden, was der Anzahl des j-ten Log-Keys in der i-ten Zeile der Event-Count-Matrix entspricht. Wenn keine der Sequenzen einen Fehler beinhaltet und alle Invarianten erfüllt werden, kann folgende Gleichung erstellt werden [Lou+10]:

$$a_0 + a_1 x_{i1} + ... + a_n x_{in} = 0, \forall i = 1, ..., m \tag{4.21}$$

Sei X wie folgt:

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & ... & x_{1n} \\ 1 & x_{21} & x_{22} & ... & x_{2n} \\ ... & & & & \\ 1 & x_{m1} & x_{m2} & ... & x_{mn} \end{bmatrix} \tag{4.22}$$

Dann kann die Formel aus 4.21 in eine Matrix transformiert werden:

$$X\theta = 0 \tag{4.23}$$

Jeder Invarianten-Vektor muss die Gleichung 4.23 erfüllen. Es lassen sich zwei Teilräume von X ableiten: die Spannweite der Zeilenvektoren und der Nullraum, welcher das orthogonale Komplement zu der Spannweite ist und im betrachteteten Zusammenhang *Invarianten-Raum* genannt wird. [Lou+10]

Mittels Singulärwertzerlegung wird der Invarianten-Raum geschätzt, wodurch die Anzahl der r Invarianten bestimmt wird. Anschließend werden die Invarianten mittels einer Brute-Force-Methode ermittelt und mithilfe eines Thresholds geprüft, wie viel Prozent der Log-Sequenzen der jeweiligen

Invarianten entsprechen. Dies wird so lange wiederholt, bis r Invarianten gefunden wurden. [He+16]

In der Praxis kann es vorkommen, dass in den Log-Sequnzen, die zur Bestimmung des Invarianten-Raums verwendet werden, bereits Fehler enthalten sind. Dies führt dazu, dass die Gleichung aus 4.21 nicht immer erfüllt wird. In Lou et al. [Lou+10] wird jedoch die Annahme getroffen, dass solche Fehler nur einen sehr geringen Anteil (unter 5%) der gesamten Log-Dateien ausmachen, wodurch die Invarianten gefunden werden können, indem die dünnbesetzte Auflösung der Gleichung aus 4.23 durchsucht wird. Dies geschieht durch die Minimierung von $\|X\theta\|_0$, was der Anzahl der Log-Sequenzen entspricht, die die Invariante θ verletzen. [Lou+10]

In den hier verwendeten Log-Dateien machen die Fehler, wie in Kapitel 4.1 beschrieben, nur einen sehr geringen Anteil der Daten aus, weshalb diese Annahme erfüllt ist.

Die Minimierung von $\|X\theta\|_0$ ist ein NP-hartes Problem, was die Suche der Invarianten sehr zeitaufwändig macht. In Lou et al. [Lou+10] wird sich deshalb der Umstand zu Nutze gemacht, dass die Log-Meldungen in den meisten Systemen gruppiert werden können, beispielsweise anhand der Transaktions-ID, was die Anzahl an unterschiedlichen Log-Keys und den Suchraum des Algorithmus stark reduziert. Der Ansatz zielt explizit nicht auf Systeme ab, für die keine Gruppierung möglich ist, da auch ohne diese die Berechnung der Invarianten sehr aufwändig ist. [Lou+10]

Zudem wird auch bei diesem Verfahren keine weitere Information wie die Uhrzeit berücksicht, weshalb eine Identifizierung der kontextuellen Anomalien nicht möglich ist. Eine Prüfung des Kontextes müsste nachträglich erfolgen. Das Verfahren ist somit nicht für diese Arbeit geeignet.

4.3.5 *Zusammenfassung*

Zum Abschluss des Kapitels werden die vorgestellten Methoden zusammengefasst und eine Übersicht über die analysierten Anforderungen gegeben. Tabelle 4.2 zeigt das Ergebnis.

Im Bereich der Supervised-Verfahren werden die klassischen Verfahren SVM, logistische Regression und Decision Tree eingesetzt. Zudem kommt ein LSTM zum Einsatz, das die Log-Sequenzen und somit den Programmablauf lernt. Die Problemstellung wird hier im Kontext der binären Klassifikation betrachtet, mit den Labels *Kein Fehler* und *Fehler*.

Für die Semi-Supervised-Anomaliedetektion wird ausschließlich ein LSTM zur Multiklassifikation verwendet, das für eine Sequenz von Log-Keys den nachfolgenden Log-Key vorhersagt. Die One-Class SVM wird im Rahmen dieser Arbeit nicht umgesetzt, da sie nicht alle Anforderungen erfüllt.

Im Unsupervised-Bereich konnten keine Modelle konzipiert werden, die allen Anforderungen gerecht werden. Solche Verfahren suchen in den Daten nach Mustern und benötigen hierfür numerische Werte, um bestimmen zu können, ob Datenpunkte von diesen Mustern abweichen und somit einen Fehler bedeuten. Aus diesem Grund sind sie für Systeme geeignet, in denen

	Anforderungen		
	Unabhängig von der Auslastung	Punktanomalien werden erkannt	Kontextuelle Anomalien werden erkannt
Supervised			
SVM	Ja	Ja	Ja
Logistische Regression	Ja	Ja	Ja
Decision Tree	Ja	Ja	Ja
LSTM	Ja	Ja	Ja
Semi-Supervised			
LSTM	Ja	Ja	Ja
One-Class SVM	Ja	Nein	Nein
Unsupervised			
Cluster-Evolution	Nein	Ja	Nein
PCA	Ja	Nein	Nein
Invariants Mining	Ja	Ja	Nein

Tabelle 4.2: Ergebnisse der Anforderungsanalyse

eine ungewöhnlich hohe Auslastung oder ein unerwartet hohes Auftreten einzelner Meldungen auf einen Fehler schließen lassen. Mittels Invariants Mining können zudem jene Fehler erkannt werden, die durch ausbleibende Meldungen entstehen, wenn beispielsweise zu einer Meldung „Open File“ das zugehörige „Close File“ fehlt. Die genannten Aspekte treten jedoch nicht im Kontext dieser Arbeit auf, weshalb die Verfahren im folgenden Kapitel nicht berücksichtigt werden. Sie könnten jedoch in zukünftigen Arbeiten anhand einer passenden Datenmenge untersucht werden.

Das Log-Clustering-Verfahren taucht in der Tabelle nicht auf, weil es durch das Erstellen der Knowledge-Base eine Vergabe von Labels durch Expert*innen benötigt und somit nicht als Unsupervised-Verfahren bezeichnet werden kann. Zudem ist mit dieser Methode ausschließlich eine Identifikation von Punktanomalien möglich, weshalb sie im Folgenden nicht berücksichtigt wird.

In Tabelle 4.3 sind zusammenfassend die Modelle aufgeführt, die im nächsten Kapitel umgesetzt und evaluiert werden.

Supervised	Semi-Supervised	Unsupervised
SVM Logistische Regression Decision Tree LSTM	LSTM	- Nicht geeignet -

Tabelle 4.3: Übersicht über die verwendeten Modelle

Dabei muss die Unausgeglichenheit der Daten beachtet werden. Zudem wird geprüft, ob wirklich beide Fehlertypen identifiziert werden.

5 UMSETZUNG UND EVALUIERUNG

In diesem Kapitel erfolgt die Umsetzung und Evaluierung der Log-Parsing-Methoden und der Modelle zur Fehlererkennung, wie sie in Kapitel 4 konzipiert wurden. Dabei werden zunächst die Ergebnisse der unterschiedlichen Log-Parsing-Verfahren analysiert und in Abhängigkeit dieser das am besten geeignete Verfahren ausgewählt, mit dem die Daten vorverarbeitet werden. Diese vorverarbeiteten Daten werden anschließend für das Training der Modelle zur Fehlererkennung verwendet. Es werden die Metriken zur Modellbewertung beschrieben und in Abhängigkeit derer die Ergebnisse analysiert und verglichen.

5.1 LOG-PARSING

Im ersten Schritt erfolgt die Vorverarbeitung der Daten, deren Ablauf in Abbildung 5.1 dargestellt ist. Die Verfahren, die anhand der Anforderungs-

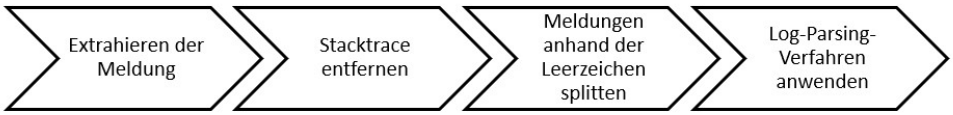


Abbildung 5.1: Ablauf der Vorverarbeitung

analyse in Kapitel 4.2 ausgewählt wurden, werden implementiert und die Genauigkeit wird verglichen. Hierfür wurden für den Monat, der als Datengrundlage für das Training der Modelle verwendet wird, manuell die Log-Keys erstellt. Daraus resultierten 151 einzigartige Log-Keys. Die Ergebnissen werden mit den unterschiedlichen Methoden verglichen. Entscheidend für die Wahl des Log-Parsing-Verfahrens ist somit die Anzahl der korrekt zugeordneten Log-Zeilen.

5.1.1 *Verfahren*

Folgende Verfahren werden im Rahmen dieser Arbeit analysiert:

- Spell
- BSG
- Online Dictionary Creation Algorithm

Im Folgenden werden die Umsetzung und die Evaluierung dieser Verfahren beschrieben und im Anschluss die Ergebnisse verglichen und bewertet.

Spell

Wie in Kapitel 4.2 beschrieben, wird für Nachrichten bestehend aus weniger als drei Wörtern ein Threshold von 0,5 gewählt. Für alle anderen Nachrichten wurden diverse Thresholds evaluiert. Die Ergebnisse werden in Tabelle 5.1 dargestellt.

Threshold	Richtig zugeordnet		Falsch zugeordnet		Anzahl einzigartiger Log-Keys
	Anzahl Zeilen	Prozent	Anzahl Zeilen	Prozent	
0,40	123.664	56,19%	96.413	43,80%	141
0,45	180.092	81,18%	39.985	18,16%	143
0,50	180.092	81,18%	39.985	18,16%	143
0,55	192.182	87,32%	27.895	12,68%	156
0,60	186.900	84,92%	33.177	15,07%	157

Tabelle 5.1: Ergebnisse Spell: Die Tabelle zeigt die Anzahl korrekt bzw. nicht korrekt zugeordneter Zeilen, sowohl die absoluten Zahlen als auch die Angabe in Prozent.

Wie zu erkennen ist, liefert ein Threshold von 0,55 die besten Ergebnisse, da dadurch die meisten Log-Meldungen korrekt erkannt werden können. Weitere Verbesserungen können erzielt werden, wenn die Log-Meldungen vor Beginn der Vorverarbeitung alphabetisch sortiert werden. Die Ergebnisse zeigt Tabelle 5.2.

Threshold	Richtig zugeordnet		Falsch zugeordnet		Anzahl einzigartiger Log-Keys
	Anzahl Zeilen	Prozent	Anzahl Zeilen	Prozent	
0,40	123.664	56,19%	96.413	43,80%	141
0,45	180.092	81,18%	39.985	18,16%	143
0,50	182.926	83,12%	37.151	16,88%	138
0,55	214.172	97,32%	5.905	2,68%	148
0,60	208.890	94,92%	11.187	5,08%	149

Tabelle 5.2: Ergebnisse Spell sortiert: Die Tabelle zeigt die Anzahl korrekt bzw. nicht korrekt zugeordneter Zeilen, sowohl die absoluten Zahlen als auch die Angabe in Prozent.

Dies widerspricht zwar dem Ansatz des Online-Log-Parsings, wie in Kapitel 4.2 beschrieben, allerdings kann es im Kontext dieser Arbeit trotzdem verwendet werden, da die Sortierung nur für den initialen Aufbau der LCS-Map benötigt wird. Neu eintreffende Meldungen müssen nicht sortiert werden, um den Algorithmus anwenden zu können und somit müssen auch die Log-Dateien nicht vorgehalten werden.

Aus diesem Grund wird sich dafür entschieden, die Meldungen für den initialen Aufbau der LCSMap alphabetisch zu sortieren. Mit einem Threshold von 0,55 können die meisten Zeilen zugeordnet werden. Die Wahl dieses Thresholds führt zwar dazu, dass beispielsweise die Meldungen „Update photo cache“ und „Updated photo cache“ dem gleichen Log-Key zugeordnet werden („* photo cache“), dies lässt sich jedoch nicht vermeiden, da die meisten anderen Meldungen, die aus drei Wörtern bestehen, einen Parameter enthalten, der identifiziert werden muss. Es wäre denkbar, solche Aspekte, die bei einer eingehenderen Analyse erkannt werden, mittels eines regulären Ausdrucks gesondert zu behandeln. An dieser Stelle wird jedoch darauf verzichtet, da das Verfahren ohne Expert*innenwissen angewendet werden soll.

BSG

Auch hier werden verschiedene Werte für den Threshold der Nachrichten mit mehr als zwei Wörtern evaluiert. Die Ergebnisse werden in Tabelle 5.3 gezeigt.

Threshold	Richtig zugeordnet		Falsch zugeordnet		Anzahl einzigartiger Log-Keys
	Anzahl Zeilen	Prozent	Anzahl Zeilen	Prozent	
0,40	214.633	97,526%	5.444	2,474%	152
0,45	219.915	99,926%	162	0,074%	151
0,50	219.927	99,932%	150	0,068%	145
0,55	219.927	99,932%	150	0,068%	145
0,60	204.506	92,925%	15.571	7,075%	142

Tabelle 5.3: Ergebnisse BSG: Die Tabelle zeigt die Anzahl korrekt bzw. nicht korrekt zugeordneter Zeilen, sowohl die absoluten Zahlen als auch die Angabe in Prozent.

In diesem Fall liefert ein Threshold zwischen 0,5 und 0,55 die besten Ergebnisse. Dies führt auch hier dazu, dass die Meldungen „Update photo cache“ und „Updated photo cache“ dem gleichen Log-Key zugeordnet werden, was sich jedoch wie beschrieben nicht vermeiden lässt.

Für die Meldungen mit zwei Wörtern wurde bereits ein Threshold von 0,5 festgelegt, weshalb sich auch bei längeren Meldungen für diesen Wert entschieden wird, um unnötige Komplexität zu vermeiden. Eine alphabetische Sortierung der Meldungen bringt keine Verbesserung, weshalb die Ergebnisse hier nicht aufgeführt wurden.

Online Dictionary Creation Algorithm

Die Ergebnisse des Online Dictionary Creation Algorithm in Abhängigkeit des Thresholds sind in Tabelle 5.4 dargestellt.

Threshold	Richtig zugeordnet		Falsch zugeordnet		Anzahl einzigartiger Log-Keys
	Anzahl Zeilen	Prozent	Anzahl Zeilen	Prozent	
0,40	160.379	72,87%	59.698	27,13%	130
0,45	166.698	75,75%	53.373	24,25 %	135
0,50	166.704	75,75%	53.373	24,25%	136
0,55	188.129	85,48%	31.948	14,52%	143
0,60	160.379	72,87%	59.698	27,13%	149

Tabelle 5.4: Ergebnisse Online Dictionary Creation Algorithm: Die Tabelle zeigt die Anzahl korrekt bzw. nicht korrekt zugeordneter Zeilen, sowohl die absoluten Zahlen als auch die Angabe in Prozent.

Auch hier kann eine Verbesserung durch eine initiale Sortierung der Meldungen erzielt werden. Die Ergebnisse zeigt Tabelle 5.5.

Threshold	Richtig zugeordnet		Falsch zugeordnet		Anzahl einzigartiger Log-Keys
	Anzahl Zeilen	Prozent	Anzahl Zeilen	Prozent	
0,40	129.193	58,70%	90.884	41,30%	129
0,45	140.866	64,01%	79.211	35,99%	135
0,50	140.870	64,01%	79.207	35,99%	138
0,55	193.406	88,48%	26.671	12,12%	144
0,60	188.1214	85,48%	31.963	14,52%	149

Tabelle 5.5: Ergebnisse Online Dictionary Creation Algorithm sortiert: Die Tabelle zeigt die Anzahl korrekt bzw. nicht korrekt zugeordneter Zeilen, sowohl die absoluten Zahlen als auch die Angabe in Prozent.

Wie schon bei Spell können mit einem Threshold von 0,55 die meisten Zeilen richtig zugeordnet werden.

5.1.2 *Vergleich und Diskussion der Ergebnisse*

Für die Wahl der geeigneten Log-Parsing-Methode ist die Anzahl korrekt zugeordneter Log-Zeilen entscheidend. Tabelle 5.6 und Abbildung 5.2 zeigen die Ergebnisse der einzelnen Verfahren im Vergleich. Zur besseren Übersichtlichkeit wird der Online Dictionary Creation Algorithm an dieser Stelle mit ODCA abgekürzt.

Wie zu erkennen ist zeigt der BSG-Algorithmus insgesamt die besten Ergebnisse. Dies bestätigt die Recherchen von He et al. [He+17] und Makanju et al. [MZHM12], die zeigen, dass das Gruppieren von Meldungen der gleichen Länge die Resultate verbessert.

	Threshold	Richtig zugeordnet		Falsch zugeordnet	
		Anzahl Zeilen	Prozent	Anzahl Zeilen	Prozent
Spell (sortiert)	0,55	214.172	97,30%	5.905	2,70%
BSG	0,50	219.915	99,93%	162	0,07%
ODCA (sortiert)	0,55	193.406	87,88%	26.671	12,12%

Tabelle 5.6: Vergleich der Log-Parsing-Methoden: Es werden die jeweils besten Ergebnisse pro Verfahren dargestellt.

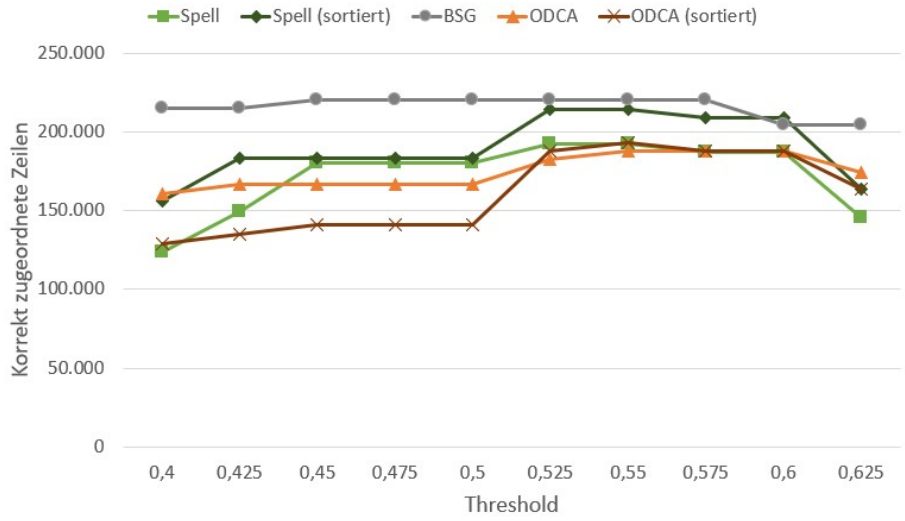


Abbildung 5.2: Vergleich der Log-Parsing-Methoden: Es wird die Anzahl der korrekt zugeordneten Zeilen in Abhängigkeit des Thresholds abgebildet.

Spell liefert zudem bessere Ergebnisse als der Online Dictionary Creation Algorithm. Dies lässt darauf schließen, dass eine individuelle Wahl des Thresholds in Abhängigkeit der Länge der Meldungen besser für die Daten geeignet ist als die Verwendung einer normierten Metrik, wie es beim Online Dictionary Creation Algorithm der Fall ist.

Zur weiteren Analyse wurde Spell mit der Levenshtein-Distanz und BSG mit dem LCS-Algorithmus als String-Metrik evaluiert. Dies konnte jedoch keine Verbesserungen bringen, weshalb die Ergebnisse hier nicht aufgelistet sind. Dadurch, dass die Verwendung des LCS-Algorithmus keine signifikanten Änderungen der Ergebnisse im Vergleich zum Einsatz der Levenshtein-Distanz brachte, wird daraus geschlossen, dass beide Metriken gleichermaßen für diesen Anwendungsfall geeignet sind.

Als weiteren Aspekt zur Bewertung des Log-Parsings kann die Laufzeit herangezogen werden. Durch die Unterteilung in Bags beim BSG-Algorithmus

muss eine Nachricht nur mit denen des gleichen Bags verglichen werden, weshalb dieser eine geringere Laufzeit hat als Spell und der Online Dictionary Creation Algorithm. Jedoch spielt diese im Kontext dieser Arbeit nur eine untergeordnete Rolle, da die Analyse der Log-Dateien wöchentlich erfolgt und die Genauigkeit der Methoden deshalb im Vordergrund steht. Aus diesem Grund wurden keine Performance-Optimierungen vorgenommen. Ohne diese ist jedoch kein valider Performance-Vergleich zwischen den Methoden möglich und deshalb wird der Aspekt im Kontext dieser Arbeit nicht berücksichtigt, könnte jedoch in zukünftigen Arbeiten näher untersucht werden.

Aufgrund der hier aufegführten Analysen wird sich dafür entschieden, den BSG-Algorithmus mit einem Threshold von 0,5 für das Log-Parsing zu verwenden. Mit dieser Implementierung können die meisten Log-Zeilen korrekt zugeordnet werden.

5.2 MODELLE ZUR FEHLERERKENNUNG

Im nächsten Schritt werden die Modelle, die in Kapitel 4.3 konzipiert wurden, umgesetzt und evaluiert. Für das Training der Modelle werden die Daten verwendet, die in Kapitel 4.1 beschrieben wurden.

Für die Evaluation und Beurteilung der Qualität der Modelle wurden die Log-Dateien zwei weiterer Monate von den Entwickler*innen zur Verfügung gestellt. Der erste Datensatz (im Folgenden als *Eval1* referenziert) umfasst 164.014 Zeilen, davon sind 241 echte Fehler. Der zweite Datensatz (im Folgenden als *Eval2* referenziert) beinhaltet 58.443 Zeilen, wovon 149 echte Fehler sind. Die Fehler beinhalten dabei Punktanomalien und kontextuelle Anomalien, kollektive Anomalien sind nicht vorhanden.

5.2.1 *Bewertungskriterien*

Im Rahmen dieser Arbeit werden die echten Fehler als *Positives* und die fehlerfreien Meldungen als *Negatives* deklariert. Eine Meldung, die vom Entwickler als Fehler markiert und auch vom Modell als solcher vorhergesagt wird, ist somit ein *True Positive (TP)*. Eine vollständige Übersicht zeigt die Konfusionsmatrix in Tabelle 5.7.

		Vom Entwickler vergebenes Label	
		Fehler	Kein Fehler
Vorhersage	Fehler	True Positive (TP)	False Positive (FP)
	Kein Fehler	False Negative (FN)	True Negative (TN)

Tabelle 5.7: Konfusionsmatrix

Um dem Aspekt der Imbalanced Data gerecht zu werden, müssen für die Evaluierung eine geeignete Metrik verwendet werden. Laut den Entwick-

ler*innen ist es besonders wichtig, dass die echten Fehler erkannt werden. Gleichzeitig soll jedoch die Zahl der Falschmeldungen nicht zu hoch werden, damit der Aufwand der daraus resultierenden manuellen Analyse möglichst gering gehalten wird. Das bedeutet, dass die Zahl der FN so gering wie möglich sein muss, ohne dass die Zahl der FP zu stark ansteigt.

Die erste Anforderung, dass die echten Fehler korrekt erkannt werden sollen, kann durch den Recall gemessen werden, der unabhängig von der Klassenverteilung ist. Die zweite Anforderung, dass die Anzahl der Falschmeldungen nicht zu hoch werden soll, kann durch die Precision oder die Specificity gemessen werden. Um jedoch beide Anforderungen kombiniert mittels einer einzelnen Metrik bewerten zu können, wird sich dafür entschieden, den F-Measure als Bewertungskriterium zu verwenden. Der Aspekt, dass die erste Anforderung von den Entwickler*innen deutlich höher priorisiert wurde, kann durch die Wahl eines geeigneten β berücksichtigt werden. Als Beispiel wurde von den Entwickler*innen folgender Vergleich aufgeführt:

A. Es werden zehn Fehler nicht erkannt (10 FN), dafür werden keine Meldungen inkorrekterweise als Fehler klassifiziert (0 FP).

B. Alle Fehler werden erkannt (0 FN), dafür werden 100 Meldungen inkorrekterweise als Fehler klassifiziert (100 FP).

An dieser Stelle wäre Variante B vorzuziehen, auch wenn die Anzahl falsch klassifizierter Meldungen somit insgesamt höher wäre. Werden die beiden Varianten auf den Trainingsdatensatz übertragen, ergeben sich die folgenden F-Measure, die in Tabelle 5.8 dargestellt sind.

	TN	FN	TP	FP	$\beta = 2$	$\beta = 3$	$\beta = 4$
Variante A	219.411	10	656	0	0,988	0,986	0,986
Variante B	219.311	0	666	100	0,971	0,986	0,991

Tabelle 5.8: F-Measure für verschiedene β

Wie zu erkennen ist, erreicht Variante B erst mit einem $\beta = 4$ einen besseren F-Measure. Aus diesem Grund wird im Rahmen dieser Arbeit der F-Measure mit einem $\beta = 4$ verwendet, im Folgenden F4-Measure genannt.

5.2.2 *Umsetzung und Ergebnisse*

In diesem Kapitel wird die Umsetzung und Evaluierung der Modelle beschrieben, wie sie in Kapitel 4.3 konzipiert wurden. Tabelle 5.9 zeigt noch einmal die Modelle und die Einordnung in Supervised-, Semi-Supervised- und Unsupervised-Verfahren.

Supervised	Semi-Supervised	Unsupervised
SVM Logistische Regression Decision Tree LSTM	LSTM	- Nicht geeignet -

Tabelle 5.9: Übersicht über die verwendeten Modelle

5.2.2.1 *Supervised Learning*

Wie beschrieben werden die Verfahren auf dem vorverarbeiteten Datensatz angewendet, der die Log-Keys enthält, die mittels des BSG-Algorithmus extrahiert wurden.

Der Datensatz, der für das Training zur Verfügung steht, wird zufällig in 20% Testdaten (44.016 Zeilen) und 80% Trainingsdaten (176.061 Zeilen) unterteilt. Bei der Erstellung des Testdatensatzes muss beachtet werden, dass durch eine zufällige Aufteilung der Log-Zeilen die Informationen über kollektive Anomalien verloren gehen. Bei der Wahl eines festen Zeitabschnitts kann es wiederum vorkommen, dass keine kontextuellen Anomalien enthalten sind. Kollektive Anomalien lagen hier nicht vor, weshalb eine zufällige Aufteilung gewählt wird. Im Anschluss an das Training wird eine Evaluierung auf den Datensätzen Eval1 und Eval2 vorgenommen.

Klassische Verfahren

Für die Implementierung der klassischen Verfahren wird die weit verbreitete Machine-Learning-Bibliothek scikit-learn[1] genutzt. Das Training erfolgt, wie in Kapitel 4.3.1 erläutert, mit den Features *Stunde* und *Log-Key*. Dafür muss der Log-Key zunächst in einen numerischen Wert transformiert werden. Eine gängige Methode, um textuelle Attribute in numerische Werte umzuwandeln, ist das sogenannte *Feature-Hashing*.

Alternativ wäre es auch möglich, den Log-Key mittels One-Hot-Kodierung vorzuverarbeiten. Dies erhöht jedoch die Dimensionalität und somit die Komplexität der Modelle. Deshalb wird sich für die Variante des Feature-Hashings entschieden.

Für alle Modelle wird eine Hyperparameteroptimierung mittels Cross-Validierung durchgeführt, wofür die Klasse *GridSearchCV* von scikit-learn verwendet wird, die eine Suche über spezifizierte Parameter ausführt. Der Suchraum ist jeweils pro Modell angegeben. Bei numerischen Werten wird zudem der Bereich um den besten Wert der Hyperparameteroptimierung manuell untersucht, um den besten Wert zu finden. Eine Beschreibung der Hyperparameter ist Anhang B.2.1 zu entnehmen.

In scikit-learn gibt es die Möglichkeit über den Parameter *class_weight* dem Algorithmus, in Abhängigkeit der Klassenverteilung, eine Gewichtung mit-

1 https://scikit-learn.org/

zugeben. Dies eignet sich für unbalancierte Datensätze und wird deshalb im Kontext dieser Arbeit verwendet.

Im Folgenden werden die Ergebnisse der Evaluierung dargestellt. Dabei werden zunächst die Resultate der Hyperparameteroptimierung und anschließend die spezifizierten Metriken zur Modellbewertung gezeigt.

Support Vector Machine

Tabelle 5.10 zeigt die verwendeten Hyperparameter und den Wertebereich, der analysiert wird.

Hyperparameter	Wertebereich
Kernel	linear, poly, rbf, sigmoid
gamma	auto, scale
C	1, 10, 100, 1000, 5000
class_weight	{Kein Fehler: 0.1, Echter Fehler: 0.9}, {Kein Fehler: 0.003, Echter Fehler: 0.997}, balanced
tol	0.0001, 0.001, 0.05, 0.5, 0.9, 0.999

Tabelle 5.10: Hyperparameter der SVM

Die Hyperparameteroptimierung ergibt folgendes Modell:

```
SVC(
    gamma="auto",
    class_weight={"Kein Fehler": 0.003, "Echter Fehler": 0.997},
    kernel="rbf",
    C=389,
    tol=0.999
)
```

Tabelle 5.11 zeigt die Ergebnisse entsprechend der optimalen Hyperparameter.

	TN	FN	TP	FP	F4-Measure
Eval1	163.749	0	241	24	0,994
Eval2	58.293	0	149	1	0,999

Tabelle 5.11: Ergebnisse der SVM

Die SVM kann alle echten Fehler korrekt erkennen (keine FN). Jedoch werden manche fehlerfreien Meldungen inkorrekt als Fehler vorhergesagt. Dies betrifft in erster Linie die kontextuellen Anomalien. Während die echten Fehler richtig klassifziert werden, werden die Meldungen, die aufgrund der Uhrzeit keinen echten Fehler darstellen, ebenfalls als Fehler vorhergesagt (FP). Dieser Umstand ist der Wahl der Hyperparameter geschuldet. Bei einer

anderen Wahl der Werte konnten die kontextuellen Anomalien korrekt zugeordnet werden, dabei stieg jedoch die Anzahl der nicht erkannten Fehler (FN), was zu einem schlechteren F4-Measure führt.

Logistische Regression

Für die logistische Regression wurden die Hyperparameter, wie in Tabelle 5.12 abgebildet, evaluiert. Zudem wurde die maximale Anzahl an Iterationen erhöht.

Hyperparameter	Wertebereich
solver	lbfgs, liblinear, newton-cg
C	1, 10, 100, 1000, 5000
class_weight	{Kein Fehler: 0.1, Echter Fehler: 0.9}, {Kein Fehler: 0.003, Echter Fehler: 0.997}, balanced
tol	0.0001, 0.001, 0.05, 0.5, 0.9, 0.999

Tabelle 5.12: Hyperparameter der logistischen Regression

Als Resultat der Hyperparameteroptimierung wurde folgendes Modell verwendet:

```
LogisticRegression(
    solver='liblinear',
    max_iter=5000,
    class_weight={"Kein Fehler": 0.003, "Echter Fehler": 0.997},
    C=5000
)
```

In Tabelle 5.13 sind die zugehörigen Ergebnisse dargestellt.

	TN	FN	TP	FP	F4-Measure
Eval1	163.740	1	240	33	0,988
Eval2	58.283	0	149	6	0,996

Tabelle 5.13: Ergebnisse der logistischen Regression

Bei der logistischen Regression können im Datensatz Eval2 alle echten Fehler korrekt erkannt werden. Im Datensatz Eval1 kann hingegen ein Fehler nicht identifiziert werden. Bei beiden Datensätzen wurden zudem Zeilen fälschlicherweise als Fehler klassifiziert, trotzdem kann ein F4-Measure über 0,98 erreicht werden.

Allerdings konnte bei keinem der Datensätze die kontextuellen Anomalie richtig zugeordnet werden, da alle entsprechenden Meldungen als Fehler klassifiziert wurden. Dies traf auch für die Meldungen zu, die in dem Zeitraum liegen, in dem sie keinen echten Fehler bedeuten.

Decision Tree

Für den Decision Tree wurden verschiedene Werte für die Gewichtung der Klassen evaluiert, wie in Tabelle 5.14 abgebildet.

Hyperparameter	Wertebereich
class_weight	{Kein Fehler: 0.1, Echter Fehler: 0.9}, {Kein Fehler: 0.003, Echter Fehler: 0.997}, balanced

Tabelle 5.14: Hyperparameter des Decision Trees

Die besten Ergebnisse lieferte folgendes Modell:

```
DecisionTreeClassifier(
    class_weight={"Kein Fehler": 0.003, "Echter Fehler": 0.997}
)
```

Diese sind in Tabelle 5.15 abgebildet.

	TN	FN	TP	FP	F4-Measure
Eval1	163.772	0	241	1	0,999
Eval2	58.294	0	149	0	1,000

Tabelle 5.15: Ergebnisse des Decision Trees

Wie anhand der Tabelle zu erkennen ist, sind beim Decision Tree alle Vorhersagen für den Datensatz Eval2 zutreffend. Beim Datensatz Eval1 wird lediglich eine Zeile inkorrekterweise als Fehler klassifiziert (FP). Jedoch werden beiden Datensätzen alle echten Fehler korrekt erkannt (keine FN).

Long Short-Term Memory

Für den Einsatz dieses Modells kommen wie beschrieben sowohl das klassische LSTM als auch die bidirektionale Variante in Frage. Aus diesem Grund wurden beide Verfahren im Rahmen dieser Arbeit evaluiert. Für die Umsetzung wurde Keras[2] verwendet, eine weit verbreitete Bibliothek für die Arbeit mit neuronalen Netzen.

LSTMs können nur mit numerischen Werten arbeiten, die zwischen 0 und 1 normiert sein sollten. Um dem Modell keine Ordnung vorzugeben, die nicht existiert, wurden den Log-Keys im Rahmen dieser Arbeit keine Zahlen zugeordnet, sondern diese mittels One-Hot-Kodierung vorverarbeitet. Mithilfe dieser Methode entstehen n Spalten, von denen jede einen der möglichen n Log-Keys aus V repräsentiert. Jedem Log-Key k_i wird in der Spalte, in der $k_i = v_j$ eine 1 eingetragen, andernfalls eine 0. Abbildung 5.3 veranschaulicht das Prinzip der One-Hot-Kodierung.

2 https://keras.io/

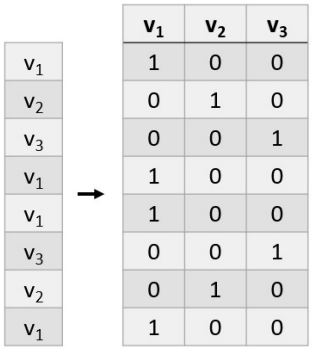


Abbildung 5.3: Beispiel One-Hot-Kodierung

Dies erhöht zwar die Dimensionalität, jedoch müssen die Log-Keys für die Multiklassifikation eindeutig zugeordnet werden können. Aus diesem Grund wurde an dieser Stelle kein Feature-Hashing verwendet.

Um die Länge der Sequenzen, m, zu bestimmmen, wurden unterschiedliche Werte evauliert. Zunächst wurde $m = 1$ gesetzt und dann schrittweise erhöht. Die Evaluierung hat gezeigt, dass sowohl für das klassische LSTM als auch für die bidirektionale Variante ein $m = 5$ die besten Ergebnisse liefert.

In Du et al. [Du+17] werden beim Semi-Supervised LSTM drei Hidden Layer verwendet. Aus diesem Grund wurde sich an dieser Größenordnung orientiert und zunächst eine einfache Architektur mit einem, dann mit zwei und dann mit drei Hidden Layern verwendet. Die Erhöhung der Anzahl an Hidden Layern konnte jedoch keine Verbesserungen erzielen, weshalb sich dafür entschieden wurde, die Architektur mit der geringeren Komplexität zu wählen. Dies entspricht sowohl bei der normalen Variante als auch beim bidirektionalen LSTM einem Hidden Layer (1.024 Neuronen) und einem Dropout Layer (Dropout-Rate von 0,2). Als Verlustfunktion wird die binäre Kreuzentropie verwendet, da diese dem Standard bei binärer Klassifikation entspricht. Bei neuronalen Netzen bestehen umfangreiche Möglichkeiten zur Wahl der Hyperparameter und der Architektur. Im Anhang B.2.2 ist dokumentiert, welche unterschiedlichen Varianten hier untersucht wurden.

Die Ergebnisse der Modelle, entsprechend der gewählten Architektur und Hyperparameter, werden in Tabelle 5.16 dargestellt, das bidirektionale LSTM wird zur besseren Übersichtlichkeit mit BLSTM abgekürzt.

		TN	FN	TP	FP	F4-Measure
LSTM	Eval1	163.768	7	234	2	0,972
	Eval2	58.290	6	141	0	0,961
BLSTM	Eval1	163.768	7	234	1	0,972
	Eval2	58.289	0	149	1	0,999

Tabelle 5.16: Ergebnisse des Supervised LSTM

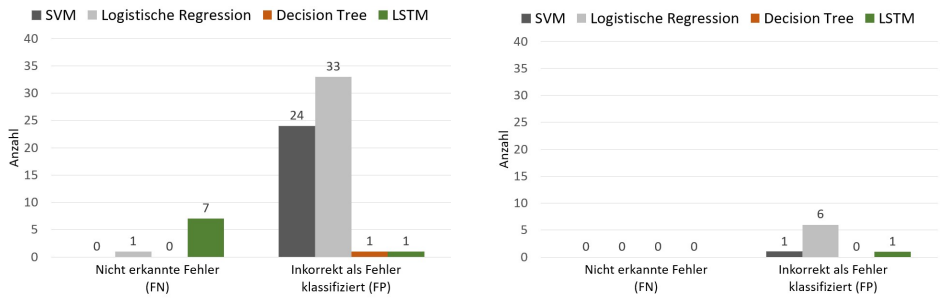
Wie zu erkennen ist, erreichen beide Varianten auf beiden Datensätzen einen F4-Measure über 0,96. Jedoch können nicht alle Fehler korrekt erkannt werden. Die birektionale Variante erzielt einen etwas besseren F4-Measure, da mehr Fehler korrekt erkannt werden (TP). Im Folgenden wird deshalb stets dieses Modell referenziert.

Vergleich

In Tabelle 5.17 und Abbildungen 5.4 sind die Ergebnisse der Supervised-Verfahren zur Anomaliedetektion zusammengefasst.

		FN	FP	F4-Measure
Eval1	SVM	0	24	0,994
	Logistische Regression	1	33	0,988
	Decision Tree	0	1	0,999
	LSTM	7	1	0,972
Eval2	SVM	0	1	0,999
	Logistische Regression	0	6	0,996
	Decision Tree	0	0	1,000
	LSTM	0	1	0,999

Tabelle 5.17: Ergebnisse der Supervised-Verfahren



(a) Datensatz Eval1 (b) Datensatz Eval2

Abbildung 5.4: Vergleich der Supervised-Verfahren

Wie zu erkennen ist, erzielt der Decision Tree die besten Ergebnisse, mit einem F4-Measure von 0,999 auf dem Datensatz Eval1 und einem F4-Measure von 1,0 auf dem Datensatz Eval2. Zudem können alle Fehler erkannt werden (keine FN). Die SVM kann ebenfalls alle Fehler erkennen, hat jedoch aufgrund der höheren Anzahl an FP einen etwas geringeren F4-Measure. Die logistische Regression weist auf beiden Datensätzen die höchste Anzahl von FP auf. Trotzdem erzielt das LSTM aufgrund der höheren Anzahl an FP insgesamt schlechtere Ergebnisse. Auch wenn der F4-Measure beim Datensatz Eval2 um 0,003 besser ist als der der logistischen Regression, ist insbesondere auf dem Datensatz Eval2 die Anzahl nicht erkannter Fehler höher. Dies

ist auch in Abbildung 5.4a zu erkennen und führt zu einem schlechteren F4-Measure.

Die erkannten Anomalien pro Modell sind in Tabelle 5.18 gezeigt.

	Erkannte Anomalien	
	Punktanomalien	Kontextuelle Anomalien
SVM	Ja	Nein
Logistische Regression	Ja	Nein
Decision Tree	Ja	Ja
LSTM	Ja	Ja

Tabelle 5.18: Erkannte Anomalien der Supervised-Verfahren

Die SVM und die logistische Regression identifizieren die kontextuellen Anomalien nicht korrekt, der Decsion Tree und das LSTM hingegen schon.

Wie die diskutierten Ergebnisse zeigen, ist der Decision Tree für diesen Anwendungsfall das beste Modell der Supervised-Verfahren. Mit diesem Modell können jedoch, wie auch mit der SVM und der logistische Regression, keine kollektiven Anomalien erkannt werden, da die Vorhersage zeilenbasiert erfolgt. Hier wäre eine weitere Analyse, beispielsweise mithilfe der Event-Count-Matrix, notwendig. Das LSTM basiert auf Sequenzen von Log-Zeilen, weshalb davon ausgegangen werden kann, dass kollektive Anomalien identifiziert werden können. Diese Annahme kann allerdings aufgrund der unzureichenden Datengrundlage nicht evaluiert werden.

5.2.2.2 *Semi-Supervised Learning*

Im Folgenden werden die Semi-Supervised-Anomaliedetektion untersucht. Für das Training werden die drei Wochen verwendet, die den fehlerfreien Zustand beschreiben, auf der vierten Woche erfolgt der Test. Anschließend wird die Validierung anhand der Evaluierungsdatensätze vorgenommen.

Long Short-Term Memory

Auch hier kann sowohl die normale Variante als auch das birektionale LSTM verwendet werden. Für die Umsetzung wurde ebenfalls Keras verwendet und die Log-Keys wurden mittels One-Hot-Kodierung vorverarbeitet.

Um die Länge der Sequenzen zu bestimmen, wurde analog zum Vorgehen des Supervised-Verfahrens zunächst $m = 1$ gesetzt und dann schrittweise erhöht. Die besten Ergebnisse konnten bei einem $m = 3$ erzielt werden.

Für die Klassifizierung muss zudem ein x gewählt werden, das beschreibt, unter welchen x Log-Keys mit der höchsten Wahrscheinlichkeit sich die Vorhersage befinden muss, um als fehlerfrei bestimmt zu werden. Die Analyse hat gezeigt, dass mit einem $x = 5$ die besten Ergebnisse erzielt werden.

Die Wahl der Hyperparameter und der Architektur wurde analog zu der Vorgehensweise beim Supervised LSTM durchgeführt. Dies resultierte sowohl bei der normalen Variante als auch beim bidirektionalen LSTM in einer Architektur mit einem Hidden Layer (512 Neuronen) und einem Dropout Layer (Dropout-Rate von 0,2). Als Verlustfunktion wird die kategorische Kreuzentropie verwendet, da es sich hier um eine Multiklassifikation handelt.

Tabelle 5.19 zeigt die Ergebnisse, das bidirektionale LSTM wird an dieser Stelle wieder mit BLSTM abgekürzt.

		TN	FN	TP	FP	F4-Measure
LSTM	Eval1	154.857	0	241	8.913	0,315
	Eval2	55.501	0	149	2.790	0,476
BLSTM	Eval1	154.851	0	241	8.919	0,315
	Eval2	55.483	0	149	2.808	0,474

Tabelle 5.19: Ergebnisse des Semi-Supervised LSTM

Die normale Variante erzielt an dieser Stelle einen leicht besseren F4-Measure als das bidirektionale Modell, da etwas weniger Log-Meldungen inkorrekt als Fehler klassifiziert werden.

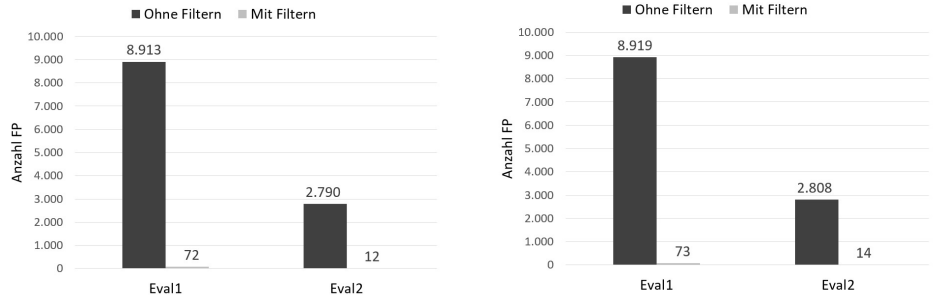
Wie zu sehen ist, kann das Modell alle echten Fehler erkennen. Die Information zur Uhrzeit, die den Kontext für die kontextuellen Anomalien liefert, wurde nicht mit in das Modell gegeben. Trotzdem können diese erkannt werden - sowohl die Meldungen, die einen Fehler darstellen als auch jene, die keinen Fehler bedeuten. Dies bestätigt die Annahme, dass eine Log-Meldung in erster Linie von den vorhergehenden Meldungen abhängt und sich auch über diese ein entsprechender Kontext feststellen lässt. Daraus lässt sich ableiten, dass dieses Modell auch für kollektive Anomalien geeignet ist, auch wenn das im Rahmen dieser Arbeit nicht evaluiert werden kann.

Dahingegen hat das Modell jedoch eine vergleichsweise hohe Anzahl an FP, weil es viele Nachrichten als Fehler deklariert, die keine solchen sind. Dies schlägt sich auch im geringen F4-Measure nieder. Die kontextuellen Anomalien können allerdings korrekt zugeordnet werden, sowohl die Meldungen, die einen Fehler darstellen, als auch die fehlerfreien Meldungen. In Absprache mit dem zuständigen Entwickler können zur Reduzierung der FP die vorhergesagten Fehler nach Loglevel gefiltert und nur *ERROR* und *WARN* berücksichtigt werden, da auch nur diese Loglevel beim Vergeben der Labels betrachtet wurden. Die daraus resultierenden Ergebnisse sind in Tabelle 5.20 Abbildung 5.5 veranschaulicht.

Wie zu erkennen ist, kann die Anzahl FP deutlich verringert werden, was zu einem höheren F4-Measure führt. Für weitere Analysen könnte es auch hilfreich sein, die Log-Meldungen vom Typ *DEBUG* oder *INFO* zu berücksichtigen, da diese auf ein ungewöhnliches Programmverhalten hindeuten könnten. Dies muss jedoch von einer Person mit Expert*innenwissen vorgenommen werden und geschieht im Rahmen dieser Arbeit nicht.

		FN	FP	F4-Measure
LSTM				
Eval1	Alle Loglevel	0	8.913	0,315
	Nur *ERROR* und *WARN*	0	72	0,983
Eval2	Alle Loglevel	0	2.790	0,476
	Nur *ERROR* und *WARN*	0	12	0,995
BLSTM				
Eval1	Alle Loglevel	0	8.919	0,315
	Nur *ERROR* und *WARN*	0	73	0,982
Eval2	Alle Loglevel	0	2.808	0,474
	Nur *ERROR* und *WARN*	0	14	0,995

Tabelle 5.20: Ergebnisse des Semi-Supervised LSTM - vor und nach dem Filtern entsprechend des Loglevels



(a) LSTM (b) Bidirektionales LSTM

Abbildung 5.5: Anzahl der Meldungen, die inkorrekterweise als Fehler vorhergesagt wurden - vor und nach dem Filtern nach Loglevel

Im Folgenden wird lediglich die normale Variante berücksichtigt, da diese etwas bessere Ergebnisse erzielt.

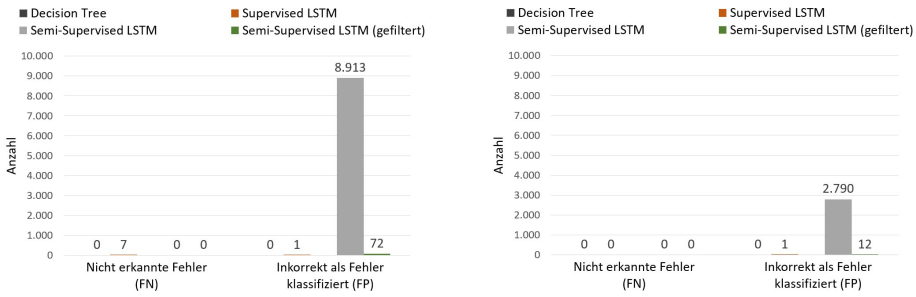
5.2.2.3 *Vergleich und Diskussion der Ergebnisse*

In diesem Abschnitt werden die Ergebnisse der Supervised- und Semi-Supervised-Anomaliedetektion verglichen. Aus dem Bereich der Supervised-Modelle werden der Decision Tree und das LSTM zur Analyse der Ergebnisse herangezogen, da diese beiden Modelle, im Gegensatz zu der SVM und der logistischen Regression, die kontextuellen Anomalien korrekt identifizieren können. Der Decision Tree erzielt außerdem die besten Ergebnisse, hier ist jedoch anzunehmen, dass keine kollektiven Anomalien erkannt werden. Beim LSTM wird angenommen, dass alle drei Fehlerarten identifiziert werden können. Aus diesem Grund wurde sich dafür entschieden, diese beiden Modelle im Vergleich zu berücksichtigen. Zudem kann dadurch eine Vergleichbarkeit zwischen dem LSTM in der Supervised-Variante und dem Semi-Supervised

LSTM hergestellt. Für die Semi-Supervised-Anomaliedetektion kommt lediglich das LSTM in Frage. Die Ergebnisse werden in Tabelle 5.21 und Abbildung 5.6 gegenübergestellt.

		FN	FP	F4-Measure
Supervised				
Eval1	Decision Tree	0	1	0,999
	LSTM	7	1	0,972
Eval2	Decision Tree	0	0	1,000
	LSTM	0	1	0,999
Semi-Supervised				
Eval1	LSTM	0	8.913	0,315
	LSTM (gefiltert)	0	72	0,983
Eval2	LSTM	0	2.970	0,476
	LSTM (gefiltert)	0	12	0,995

Tabelle 5.21: Vergleich der Supervised- und Semi-Supervised-Verfahren



(a) Datensatz *Eval1* (b) Datensatz *Eval2*

Abbildung 5.6: Vergleich der Supervised- und Semi-Supervised-Verfahren

Wie zu erkennen ist, können die Verfahren des Supervised Learnings einen besseren F4-Measure erzielen als das Semi-Supervised LSTM ohne Filtern der Ergebnisse nach Loglevel, da sie deutlich weniger FP aufweisen. Durch das Filtern kann jedoch ein vergleichbarer F4-Measure erreicht werden.

Mit dem Decision Tree und dem Semi-Supervised LSTM können sowohl mittels eines Supervised- als auch eines Semi-Supervised-Verfahrens alle Fehler erkannt werden (keine FN).

Für das Training der Supervised-Verfahren wird ein umfangreicher Datensatz mit Labels benötigt, der alle möglichen Fehlerszenarien abdeckt. Dies ist häufig mit großem Aufwand verbunden. Für den Einsatz des LSTM als Semi-Supervised-Methode werden hingegen lediglich die Daten des Normalzustands ohne Fehler benötigt. Dabei sollte auch hier ein Datensatz zur

Verfügung stehen, der den Programmablauf der Anwendung umfangreich abdeckt. Der Aufwand für das Vergeben der Labels entfällt jedoch. Wie in Abbildung 5.6 zu sehen, kann dieses Modell alle echten Fehler korrekt erkennen (keine FN), was die wichtigste Anforderung an die Modelle war. Im Vergleich zu den Methoden der Supervised-Anomaliedetektion ist allerdings die Anzahl der Meldungen, die inkorrekterweise als Fehler vorhergesagt werden (FP), deutlich höher. Durch ein Filtern dieser Meldungen entsprechend dem Loglevel kann die Anzahl jedoch, wie beschrieben, signifikant verringert werden.

Somit steht der Aufwand für die Vergabe der Labels bei der Supervised-Anomaliedetektion dem Aufwand zur manuellen Analyse der Modellergebnisse im Bereich der Semi-Supervised-Anomaliedetektion gegenüber. Stehen bereits Labels zur Verfügung, empfiehlt es sich, Supervised-Modelle zur Fehlererkennung einzusetzen. Soll ein Modell jedoch initial ohne umfangreiches Expert*innenwissen trainiert werden, ist der Einsatz des Semi-Supervised LSTM die bessere Wahl.

Im Kontext dieser Arbeit lagen zunächst keine gelabelten Daten vor, weshalb ein Entwickler des Microservice dies für den gesamten Trainingsdatensatz übernehmen musste. Dieser Aufwand wird größer bewertet als die Analyse der Modellergebnisse, weshalb für diesen Kontext das Semi-Supervised LSTM als das am besten geeignete Modell betrachtet wird.

Wie schon beim Log-Parsing ist der Aspekt der Performance im Kontext dieser Arbeit nicht relevant, da die Analyse der Log-Dateien einmal wöchentlich erfolgt. Alle Modelle konnten für die angegebene Datenmenge mindestens stündlich Vorhersagen treffen. Für zeitkritische Anwendungen könnte in zukünftigen Arbeiten jedoch eine genauere Untersuchung erfolgen.

Die Analyse der Modelle hat gezeigt, dass eine Reduzierung der FN stets mit einer Erhöhung der FP einhergeht. Wie in Landauer et al. [Lan+18] beschrieben, ist dies eine Problematik, die alle Techniken der Anomaliedetektion betrifft. An dieser Stelle wird ein aktuelles und umfangreiches Expert*innenwissen benötigt, um das Modell in der Unterscheidung zwischen normalem Verhalten und tatsächlicher Anomalie zu unterstützen. [Lan+18]

5.2.3 *Inkrementelles Lernen*

Die Daten, die für das Training verwendet werden, decken häufig nicht alle Programmabläufe ab, insbesondere sind meist nicht alle möglichen Fehler enthalten. Aus diesem Grund ist es unter Umständen notwendig, dass das Modell inkrementell lernt. Dies ermöglicht es Expert*innen, Feedback zu geben, wenn beispielsweise eine Meldung inkorrekterweise als Fehler klassifiziert wurde. Dies spielt insbesondere für die Semi-Supervised- und Unsupervised-Verfahren eine Rolle, die ohne Labels verwendet werden und somit kein Expert*innenwissen in das Training fließt.

Inkrementelles Lernen ist eine komplexe Aufgabe im Bereich Machine Learning, die mit folgenden Herausforderungen einhergeht:

CAPACITY SATURATION: Jedes Modell hat nur ein gewisse Kapazität. Soll neues Wissen gespeichert werden, ohne dass altes Wissen verloren geht, muss diese Kapazität im laufenden Betrieb erhöht werden. [SCB20]

CATASTROPHIC FORGETTING: Mit jedem neuen Training steigt die Wahrscheinlichkeit, dass altes Wissen vergessen wird. Ein möglicher Grund dafür ist die Capacity Saturation, aber auch bei ausreichender Kapazität kann durch neues Training altes Wissen verloren gehen. [SCB20]

Dies ist auch bekannt als *stability-plasticity-dilemma*, das Dilemma, neue Informationen zu lernen, ohne altes Wissen zu vergessen. Ein stabiles Modell behält das gelernte Wissen, kann jedoch nur bis zu einem gewissen Zeitpunkt neue Informationen lernen. Dahingegen lernt ein plastisches Modell neue Informationen auf Kosten des vorherigen Wissens. [Pol+01] Diese Aspekte müssen beim inkrementellen Lernen eines Modells beachtet werden. Eine Alternative, um mit neuen Daten umzugehen, wäre ein erneutes Training mit der gesamten Datenmenge. Hierfür müssen jedoch alle Daten vorgehalten werden, weshalb dieser Ansatz in der Praxis meist nicht praktikabel ist.

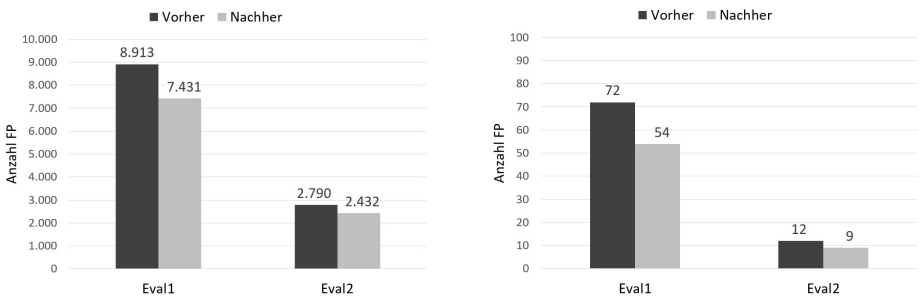
In Du et al. [Du+17] wird, wie beschrieben, angenommen, dass alle Log-Keys zu Beginn des Trainings bekannt sind. Falls nicht auf den Sourcecode zugegriffen werden kann, muss allerdings, um dieser Anforderung gerecht zu werden, ein sehr umfangreicher Datensatz für das Training zur Verfügung gestellt werden. Bei einer Weiterentwicklung des Systems kann es zudem passieren, dass sich Log-Keys ändern.

Bei dem LSTM, das im Kontext dieser Arbeit verwendet wurde, wurden Log-Keys, die im Training nicht bekannt waren, in der One-Hot-Kodierung durch eine Reihe mit ausschließlich Nullen repräsentiert. Dies stellt sicher, dass das Modell weiß, dass es diesen Log-Key noch nicht gesehen hat. Allerdings werden somit alle neuen Log-Keys durch Nullen dargestellt, weshalb keine Unterscheidung zwischen neuen Log-Keys möglich ist. Hierfür wäre ein erneutes Trainieren des Modells mit der gesamten Datenmenge erfoderlich. Dieser Aspekt unterstreicht noch einmal die Notwendigkeit eines umfangreichen Trainingsdatensatzes.

In Du et al. [Du+17] können Expert*innen Feedback geben, das an das Modell zurückgegeben wird. Dabei muss das Modell nicht neu trainiert werden, sondern es werden lediglich die vom Modell falsch klassifizierten Zeilen als neue Trainingsdaten verwendet, um das Modell inkrementell zu erweitern.

Im Rahmen dieser Arbeit wurde dieser Ansatz untersucht. Die Methoden der Supervised-Anomaliedetektion konnten bereits die meisten Zeilen korrekt zuordnen, weshalb sich dafür entschieden wurde, zur Analyse das Semi-Supervised LSTM zu verwenden. Die Meldungen, die im Training inkorrekt als Fehler vorhergesagt wurden, werden verwendet, um das Modell inkrementell zu erweitern. Die Ergebnisse sind in Abbildung 5.7 dokumentiert.

Wie zu sehen ist, kann durch die inkrementelle Erweiterung auf beiden Datensätzen der Evaluierung die Anzahl der Meldungen, die fälschlicher-



(a) Ohne Filtern (b) Mit Filtern

Abbildung 5.7: Ergebnisse des inkrementellen Lernens

weise als Fehler vohergesagt werden, reduziert werden (FP). Die Anzahl nicht erkannter Fehler ist dabei nicht gestiegen.

An dieser Stelle sei jedoch erwähnt, dass dies nur eine sehr oberflächliche Untersuchung darstellt. In Schak et al. [SG19] wird der Effekt des Catastrophic Forgetting beim Einsatz von LSTMs untersucht. Das Ergebnis der Studie zeigt, dass der gennante Effekt bei allen betrachteten LSTM-basierten Klassifzierern auftrat. [SG19] Aus diesem Grund sollte dieser Aspekt in zukünftigen Arbeiten näher untersucht werden, ist jedoch aus Gründen des Umfangs nicht Thema dieser Arbeit.

6 FAZIT UND AUSBLICK

In diesem Kapitel erfolgt eine Zusammenfassung der Ergebnisse dieser Arbeit. Zudem wird ein Ausblick auf potentiell aufbauende Untersuchungsaspekte gegeben.

6.1 FAZIT

Ziel dieser Arbeit war die automatisierte Fehlererkennung einer Microservice-Anwendung basierend auf Log-Dateien. Zu diesem Zweck wurden drei Fragestellungen untersucht:

1. Welches Log-Parsing-Verfahren ist für die Daten am besten geeignet?
2. Welches Modell ist für die automatisierte Fehlererkennung des Microservice am besten geeignet?
3. Spiegelt sich der Aufwand zur Vergabe von Labels in der Vorhersagegüte der Supervised-Modelle wider?

Die erste Fragestellung ist von Bedeutung, um aus den unstrukturierten Log-Dateien Features für den Einsatz der Modelle zur Fehlererkennung zu gewinnen. Hierfür wurden die Log-Meldungen in den konstanten Teil, den sogenannten Log-Key, und den variablen Teil, die sogenannten Parameter, getrennt. Die extrahierten Log-Keys konnten im Anschluss von den Modellen verwendet werden.

Aus der Analyse der Log-Dateien des Microservice ergaben sich folgende Aspekte für die Vorverarbeitung:

- Die Metriken zum String-Vergleich wurden auf Basis der Wörter und nicht auf Basis der einzelnen Zeichen angewendet.
- Die Semantik und Grammatik der einzelnen Wörter wurde nicht berücksichtigt.
- Zu Beginn wurde der Stacktrace, den einzelne Meldungen besitzen, entfernt.
- Bei der Wahl des Thresholds für den Einsatz der String-Metriken musste die Länge der Wörter beachtet werden.

Zur Extrahierung der Log-Keys wurden drei Methoden, die die analysierten Anforderungen erfüllen, umgesetzt und evaluiert. Es wurde gezeigt, dass der BSG-Algorithmus mit einem Threshold von 0,5 die besten Ergebnisse erzielen konnte, da hiermit die meisten Log-Zeilen korrekt zugeordnet werden konnten.

Im Anschluss an die Vorverarbeitung erfolgte zur Untersuchung der weiteren Fragestellungen die Modellauswahl zur Fehlererkennung und der Vergleich der Supervised-, Semi-Supervised- und Unsupervised-Methoden.

Für die Modelle wurden in Abhängigkeit der Microservice-Anwendung und den daraus resultierenden Log-Dateien drei Anforderungen abgeleitet:

- Die Vorhersage darf nicht von der Auslastung des Systems abhängen.
- Punktanomalien müssen erkannt werden.
- Kontextuelle Anomalien müssen erkannt werden.

Die Supervised-Verfahren wurden im Kontext der Problemstellung einer binären Klassifikation betrachtet, bei der geprüft wird, ob es sich bei einer Log-Zeile um einen Fehler handelt oder nicht. Dafür wurden drei klassische Verfahren zur binären Klassifikation betrachet: Die Support Vector Machine (SVM), die logistische Regression und der Decision Tree, die alle drei die definierten Anforderungen erfüllen. Zudem wurde ein Long Short-Term Memory (LSTM) untersucht, das inbesondere für sequentielle Daten gut geeignet ist. Auch dieses Modell erfüllt alle Anforderungen.

Im Bereich der Semi-Supervised-Anomaliedetektion wurde ebenfalls ein LSTM untersucht. Labels stehen bei diesen Verfahren nicht zur Verfügung, weshalb die Problemstellung hier im Kontext der Multiklassifikation betrachtet wurde. Jeder Log-Key stellt dabei eine eigene Klasse dar. Für eine Sequenz von Log-Keys wird die Wahrscheinlichkeit für den darauffolgenden über alle möglichen Log-Keys berechnet. Dieses Modell erfüllt ebenfalls die erforderlichen Anforderungen. Zudem wurde eine One-Class SVM betrachtet, da dieses Modell gut für unausgeglichene Daten geeignet ist. Mit diesem Verfahren können jedoch keine kontextuellen Anomalien erkannt werden, weshalb es nicht alle Anforderungen erfüllt.

Die Analyse hat gezeigt, dass Unsupervised-Verfahren zur Fehlererkennung für den betrachteten Anwendungsfall nicht geeignet sind. Solche Verfahren suchen in den Daten nach Mustern und deklarieren Datenpunkte, die von diesen Mustern abweichen, als Anomalie. Aus diesem Grund eignen sich Unsupervised-Verfahren in erster Linie für Anwendungen, in denen die Daten mittels numerischer Werte verglichen werden und so Fehler identifiziert werden können. Dies kann beispielsweise über die Auslastung des Systems erfolgen oder über die Analyse der Cluster-Evolution. Es lässt sich daraus schließen, dass Unsupervised-Verfahren deshalb für die Identifizierung von kollektiven Anomalien eingesetzt werden können. Als geeignete Verfahren sind hier wiederum die Cluster-Evolution, aber auch das Invariants Mining zu nennen. Invariants Mining eignet sich zudem für Daten, in denen ausbleibende Meldungen auf ein Fehlerverhalten der Anwendung hindeuten.

Für die Umsetzung wurden nur jene Verfahren berücksichtigt, die alle Anforderungen erfüllen konnten. Die korrekte Erkennung der Fehler war von höchster Bedeutung für die Beurteilung der Modellqualität. Zugleich durfte die Anzahl der Meldungen, die inkorrekterweise als Fehler klassifiziert wurden, nicht so groß sein, dass deren manuelle Analyse zu aufwändig wird.

Im Bereich der Supervised-Anomaliedetektion erzielte bei den klassischen Verfahren der Decision Tree die besten Ergebnisse. Die SVM und die logistische Regression konnten die kontextuellen Fehler nicht korrekt erkennen. Durch eine veränderte Auswahl der Hyperparameter konnte zwar auch bei der SVM eine korrekte Identifzierung der kontextuellen Anomalien erfolgen, jedoch führte dies zu einem Anstieg der nicht erkannten Fehler. Auch durch den Einsatz des LSTM konnten gute Ergebnisse erzielt werden, jedoch wurden nicht alle Fehler erkannt.

Bei der Semi-Supervised-Anomaliedetektion konnten alle Fehler vom LSTM identifiziert werden. Jedoch trat hier, im Vergleich zu den Supervised-Modellen, eine höhere Anzahl an Meldungen auf, die fälschlicherweise als Fehler zugeordnet wurden. Die Anzahl solcher Meldungen bewegte sich jedoch in einem Rahmen, der den Aufwand einer manuellen Analyse nicht übersteigt. Aus diesem Grund ist das Modell für den beschriebenen Anwendungsfall geeignet.

Somit wurde gezeigt, dass zur automatisierten Fehlererkennung der Microservice-Anwendung sowohl Supervised- als auch Semi-Supervised-Modelle eingesetzt werden können. Für die Supervised-Modelle müssen zu Beginn manuell Labels vergeben werden, was häufig mit einem hohen Aufwand einhergeht. Dies ist bei den Semi-Supervised-Verfahren nicht der Fall, hier besteht der Aufwand jedoch in einer höheren manuellen Analyse der Modellergebnisse. Dieser kann mittels eines Expert*innen-Feedbacks über inkrementelles Lernen reduziert werden, jedoch muss in zukünftigen Arbeiten näher untersucht werden, inwiefern vorheriges Wissen durch ein inkrementelles Lernen verloren geht. Im Rahmen dieser Arbeit wird der Aufwand für die manuelle Analyse der Ergebnisse des Semi-Supervised-Modells geringer bewertet als der initiale Aufwand zur Vergabe der Labels. Somit ist der Einsatz des LSTM im Kontext der Multiklassifikation das Modell, das für den beschriebenen Anwendungsfall am besten geeignet ist.

6.2 AUSBLICK

Bei den entwickelten Methoden lag der Fokus auf einer Qualitätsbewertung mittels Genauigkeit. So wurden für die Log-Parsing-Methoden die Ergebnisse ausschließlich anhand der Anzahl korrekt zugeordneter Log-Zeilen geprüft. Des Weiteren wäre auch eine Bewertung hinsichtlich der Laufzeit der Methoden denkbar. Im Zuge dessen könnten Peformance-Optimierungen der einzelnen Verfahren näher untersucht werden. Dies kann jedoch lediglich als erweitertes Qualitätsmerkmal hinzugezogen werden. Die korrekte Zurodnung der Log-Zeilen bleibt für die Vorhersagegüte der Modelle weiterhin essentiell.

Auch für die Bewertung der Modelle zur Fehlererkennung wäre eine zusätzliche Berücksichtigung der Performance denkbar. Die Analyse der Log-Dateien der Microservice-Anwendung erfolgt jedoch einmal wöchentlich, weshalb die Zeitkomponente keine entscheidende Rolle spielt. Für zeitkri-

tische Anwendungen kann die Qualitätsbewertung in Abhängigkeit der Performance der Modelle jedoch von Bedeutung sein.

Wie beschrieben traten im Rahmen dieser Arbeit nur Punktanomalien und kontextuelle Anomalien auf. Für weitere Analysen wäre es denkbar, die entwickelten Verfahren der Supervised- und Semi-Supervised-Anomaliedetektion dahingehend zu erweitern, dass auch kollektive Anomalien erkannt werden können. Dies spielt insbesondere für die klassischen Verfahren des Supervised Learnings eine Rolle, die zeilenbasiert eingesetzt wurden. Es ist denkbar, dass im Anschluss an die Vorhersage der Modelle zur Identifizierung der kollektiven Anomalien eine weitere Prüfung mittels der Event-Count-Matrix durchgeführt werden könnte. Das LSTM lernt, sowohl in der Supervised- als auch in der Semi-Supervised-Variante, Sequenzen in den Daten, weshalb davon ausgegangen werden kann, dass dieses Modell kollektive Anomalien erkennen kann. Dies könnte in weiteren Analysen, in denen Daten mit kollektiven Anomalien vorliegen, geprüft werden.

Die Verfahren des Unsupervised Learnings erfüllten nicht die definierten Anforderungen. Zukünftige Arbeiten könnten Anwendungsfälle untersuchen, für die solche Verfahren geeignet sind, um die Methoden, die in dieser Arbeit vorgestellt wurden, besser bewerten zu können.

Der Aspekt des inkrementellen Lernens konnte im Umfang dieser Arbeit nur oberflächlich betrachtet werden. Zukünftige Arbeiten könnten hier einen stärkeren Fokus setzen und untersuchen, wie ein Modell inkrementell erweitert werden kann, ohne bereits erlerntes Wissen zu verlieren.

Teil II

APPENDIX

A

DATENGRUNDLAGE

A.1 AUSZUG DER LOG-DATEIEN

Im Folgenden ist ein Auszug der Log-Dateien dargestellt. Aus Datenschutzgründen wurden ORDIX-spezifische Informationen ersetzt. Der Datentyp wurde beibehalten.

```
2020-05-13 05:00:00,000 DEBUG [scheduling-1] [de.or.co.se.
    ChangedPhotoService] - Updating photo cache
2020-05-13 05:00:00,017 INFO [scheduling-1] [class1] - getSmallJpegPhoto
     aaa
2020-05-13 05:00:00,095 INFO [scheduling-1] [class2] - getSmallJpegPhoto
     bbb
2020-05-13 05:00:00,166 INFO [scheduling-1] [class2] - getSmallJpegPhoto
     ccc
2020-05-13 05:00:00,232 INFO [scheduling-1] [class2] - getSmallJpegPhoto
     ddd
...
2020-05-13 05:00:10,277 DEBUG [scheduling-1] [class2] - Found /path/yyy.
    json
2020-05-13 05:00:10,278 ERROR [scheduling-1] [class3] - Exception beim
    Lesen der Konfigurationsdatei f?r yyy - Standardwerte werden
    verwendet
2020-05-13 05:00:10,278 DEBUG [scheduling-1] [class3] - No contact
    folder found for yyy - using default (X)
2020-05-13 05:00:10,278 DEBUG [scheduling-1] [class4] - Bearer Token URL
    : <URL> Body: client_secret=***&scope=***
2020-05-13 05:00:10,685 DEBUG [scheduling-1] [class4] - Got a new bearer
     token , expires_in 3599 sec
2020-05-13 05:00:10,685 DEBUG [scheduling-1] [class4] - access_token
    =***...
2020-05-13 05:00:10,852 INFO [scheduling-1] [class5] - Contact Folder
    existiert bereits, daher wird nun die dazugeh?rige ID verwendet: <ID
    >
2020-05-13 05:00:10,852 DEBUG [scheduling-1] [class1] - Updating yyy-
    ordix's photo of IDbbb
2020-05-13 05:00:11,040 DEBUG [scheduling-1] [class1] - Updated the
    photo
2020-05-13 05:00:11,040 DEBUG [scheduling-1] [class3] - Found /path/xxx.
    json
2020-05-13 05:00:11,040 DEBUG [scheduling-1] [class1] - Updating xxx-
    default's photo of IDaaa
2020-05-13 05:00:11,608 DEBUG [scheduling-1] [class1] - Updated the
    photo
...
2020-05-13 05:00:17,366 DEBUG [scheduling-1] [class1] - Updated photo
    cache
```

```
2020-05-13 05:00:17,366 INFO [scheduling-1] [class6] - Prozess zur L?
    schung von ausgeschiedenen Kontakten beginnt.
2020-05-13 05:00:17,400 WARN [scheduling-1] [class7] - SQL Error: -425,
    SQLState: IX000
2020-05-13 05:00:17,400 ERROR [scheduling-1] [class7] - Database is
    currently opened by another user.
2020-05-13 05:00:17,400 WARN [scheduling-1] [class7] - SQL Error: -107,
    SQLState: IX000
2020-05-13 05:00:17,400 ERROR [scheduling-1] [class7] - ISAM error:
    record is locked.
2020-05-13 05:00:17,401 ERROR [scheduling-1] [class8] - Unexpected error
     occurred in scheduled task.
org.springframework.orm.jpa.JpaSystemException: Unable to acquire JDBC
    Connection; nested exception is org.hibernate.exception.
    GenericJDBCException: Unable to acquire JDBC Connection
        at org.springframework.orm.jpa.vendor.HibernateJpaDialect.
            convertHibernateAccessException(HibernateJpaDialect.java
            :351) ~[spring-orm-5.1.6.RELEASE.jar!/:5.1.6.RELEASE]
        at org.springframework.orm.jpa.vendor.HibernateJpaDialect.
            translateExceptionIfPossible(HibernateJpaDialect.java:253)
            ~[spring-orm-5.1.6.RELEASE.jar!/:5.1.6.RELEASE]
        at org.springframework.orm.jpa.AbstractEntityManagerFactoryBean.
            translateExceptionIfPossible(
            AbstractEntityManagerFactoryBean.java:527) ~[spring-orm
            -5.1.6.RELEASE.jar!/:5.1.6.RELEASE]
    ...
2020-05-13 06:00:00,006 DEBUG [scheduling-1] [class3] - Found xxx.json
2020-05-13 06:00:00,007 DEBUG [scheduling-1] [class3] - Found /path/xxx.
    json
2020-05-13 06:00:00,007 DEBUG [scheduling-1] [class3] - Found yyy.json
2020-05-13 06:00:00,007 DEBUG [scheduling-1] [class3] - Found /path/yyy.
    json
2020-05-13 06:00:00,007 ERROR [scheduling-1] [class3] - Exception beim
    Lesen der Konfigurationsdatei f?r yyy - Standardwerte werden
    verwendet
2020-05-13 06:00:00,007 DEBUG [scheduling-1] [class3] - No contact
    folder found for yyy - using default (X)
2020-05-13 06:00:00,007 DEBUG [scheduling-1] [class4] - Bearer Token URL
    : <URL> Body: client_secret=***&scope=***
2020-05-13 06:00:00,497 DEBUG [scheduling-1] [class4] - Got a new bearer
     token , expires_in 3599 sec
2020-05-13 06:00:00,497 DEBUG [scheduling-1] [class4] - access_token
    =*...
2020-05-13 06:00:00,641 INFO [scheduling-1] [class5] - Contact Folder
    existiert bereits, daher wird nun die dazugeh?rige ID verwendet:
    IDaaa
2020-05-13 06:00:00,641 INFO [scheduling-1] [class6] - Aktualisieren der
     Contacts f?r xxx,yyy beginnt.
2020-05-13 06:00:00,641 INFO [scheduling-1] [class9] - Refreshing cache
    for changed contacts
2020-05-13 06:00:00,641 INFO [scheduling-1] [class10] - Updating
    contacts of xxx
```

```
2020-05-13 06:00:00,641 DEBUG [scheduling-1] [class10] - Using delta
    link <Link>
2020-05-13 06:00:00,883 DEBUG [scheduling-1] [class10] - Found 3 changed
     contacts for xxx
2020-05-13 06:00:00,883 DEBUG [scheduling-1] [class9] - Loading all
    contacts from databases ... this might take some time
2020-05-13 06:00:00,883 DEBUG [scheduling-1] [class11] - Getting
    contacts from Table1
2020-05-13 06:00:00,952 WARN [scheduling-1] [class7] - SQL Warning Code:
     0, SQLState: 01I01
2020-05-13 06:00:00,952 WARN [scheduling-1] [class7] - Database has
    transactions
2020-05-13 06:00:00,952 WARN [scheduling-1] [class7] - SQL Warning Code:
     0, SQLState: 01I04
2020-05-13 06:00:00,952 WARN [scheduling-1] [class7] - Database selected
2020-05-13 06:00:00,952 DEBUG [scheduling-1] [class12] - Getting
    contacts from Service1
2020-05-13 06:00:01,000 DEBUG [scheduling-1] [class13] - Getting
    contacts from Table2
2020-05-13 06:00:01,527 DEBUG [scheduling-1] [class13] - Found 252
    workers, conversion might take some time
2020-05-13 06:00:18,001 DEBUG [scheduling-1] [class14] -
    Mitarbeiterkategorie von XXX ist NN - es wird keine Position
    gespeichert
2020-05-13 06:00:18,069 DEBUG [scheduling-1] [class14] -
    Mitarbeiterkategorie von YYY ist NN - es wird keine Position
    gespeichert
2020-05-13 06:00:18,138 DEBUG [scheduling-1] [class14] -
    Mitarbeiterkategorie von ZZZ ist NN - es wird keine Position
    gespeichert
2020-05-13 06:00:18,205 DEBUG [scheduling-1] [class9] - Q wird durch die
     Konfiguration als Kontakt ausgeschlossen, springe zum n?chsten
    Mitarbeiter
..
2020-05-13 06:00:21,463 INFO [scheduling-1] [class10] -
    updateAllContactsForOneWorker() wurde beendet.
2020-05-13 06:00:21,463 INFO [scheduling-1] [class6] - Aktualisieren
    abgeschlossen.
...
```

B

DETAILS ZUR UMSETZUNG

B.1 LOG-PARSING

Listing B.1: Implementierung des LCS-Algorithmus auf Basis der Wörter

```
def lcs_algorithmus(a, b):
    tbl = [[0 for _ in range(len(b) + 1)] for _ in range(len(a) + 1)]
    for i, x in enumerate(a):
        for j, y in enumerate(b):
            tbl[i + 1][j + 1] = tbl[i][j] + 1 if x == y else max(tbl[i +
                1][j], tbl[i][j + 1])
    res = []
    i, j = len(a), len(b)
    while i and j:
        if tbl[i][j] == tbl[i - 1][j]:
            i -= 1
        elif tbl[i][j] == tbl[i][j - 1]:
            j -= 1
        else:
            res.append(a[i - 1])
            i -= 1
            j -= 1
    return len(res[::-1]), res[::-1]
```

Listing B.2: Implementierung der Levenshtein-Distanz auf Basis der Wörter

```
def levenshtein_distanz(seq1, seq2):
    size_x = len(seq1) + 1
    size_y = len(seq2) + 1
    matrix = np.zeros ((size_x, size_y))
    for x in range(size_x):
        matrix [x, 0] = x
    for y in range(size_y):
        matrix [0, y] = y

    for x in range(1, size_x):
        for y in range(1, size_y):
            if seq1[x-1] == seq2[y-1]:
                matrix [x,y] = min(
                    matrix[x-1, y] + 1,
                    matrix[x-1, y-1],
                    matrix[x, y-1] + 1
                )
            else:
                matrix [x,y] = min(
                    matrix[x-1,y] + 1,
```

```
                matrix[x-1,y-1] + 1,
                matrix[x,y-1] + 1
            )
    return (matrix[size_x - 1, size_y - 1])
```

B.2 BESCHREIBUNG DER HYPERPARAMETER

B.2.1 *Klassische Verfahren*

Hyperparameter	Beschreibung
Kernel	Genutzter Kernel
gamma	Kernel-Koeffizient für die Kernel linear, poly, rbf, sigmoid. Wenn gamma='scale', dann gamma=1 / (Anzahl der Features * Varianz von X) Wenn gamma='auto', dann gamma=1 / Anzahl der Features
C	Regularisierungsparameter Die Stärke der Regularisierung ist umgekehrt proportional zu C. C muss immer positiv sein. Die Strafe ist eine quadrierte L2-Strafe.
class_weight	Den Klassen zugeordnete Gewichte in der Form {Klassenname: Gewicht} Ist der Parameter nicht gesetzt, werden alle Klassen gleich gewichtet. Bei der Verwendung von *balanced* werden die Klassengewichte automatisch angepasst, umgekehrt proportional zu den Häufigkeiten der Klassen des Inputs.
tol	Toleranz für das Stop-Kriterium

Tabelle B.1: Beschreibung der Hyperparameter der SVM

Hyperparameter	Wertebereich
solver	Algorithmus, der für das Optimierungsproblem verwendet wird.
C	Regularisierungsparameter Die Stärke der Regularisierung ist umgekehrt proportional zu C. C muss immer positiv sein. Die Strafe ist eine quadrierte L2-Strafe.
class_weight	Den Klassen zugeordnete Gewichte in der Form {Klassenname: Gewicht} Ist der Parameter nicht gesetzt, werden alle Klassen gleich gewichtet. Bei der Verwendung von *balanced* werden die Klassengewichte automatisch angepasst, umgekehrt proportional zu den Häufigkeiten der Klassen des Inputs.
tol	Toleranz für das Stop-Kriterium

Tabelle B.2: Beschreibung der Hyperparameter der logistischen Regression

Hyperparameter	Wertebereich
class_weight	Den Klassen zugeordnete Gewichte in der Form {Klassenname: Gewicht} Ist der Parameter nicht gesetzt, werden alle Klassen gleich gewichtet. Bei der Verwendung von *balanced* werden die Klassengewichte automatisch angepasst, umgekehrt proportional zu den Häufigkeiten der Klassen des Inputs.

Tabelle B.3: Beschreibung der Hyperparameter des Decision Trees

B.2.2 *Long Short-Term Memory*

Die hier aufgelisteten Parameter wurden sowohl für die normale Variante als auch für die bidirektionale Alternative verwendet.

Parameter	Wertebereich
Dropout-Rate	0.3, 0.2, 0.1, 0.05
Anzahl Neuronen in einem Hidden Layer	8, 64, 128, 256, 512, 1024

Tabelle B.4: Architektur des LSTM

Dropout ist eine Regularisierungsmethode für neuronale Netze, um Overfitting vorzubeugen. Dadurch wird beim Training eine gewissen Anzahl an Neuronen eines Layers „ausgeschaltet“. Die konkrete Anzahl wird durch die

Dropout-Rate angegeben. Eine Dropout-Rate von 0,2 bedeutet beispielsweise, dass 20% der Neuronen eines Layers nicht berücksichtigt werden.

Parameter	Wertebereich
Optimizer	Adam, RMSprop, SGD
Lernrate	5, 10, 20

Tabelle B.5: Einstellungen zur Optimierung des LSTM

Eine häufig verwendetes Verfahren, mit dem neuronale Netze lernen ist der Backpropagation-Algorithmus. Der Optimizer ist die spezifische Implementierung des Backpropagation-Algorithmus von Keras. Die Lernrate des Optimizers bestimmt dabei die Größe der Schritte im Backpropagation-Verfahren.

Parameter	Wertebereich
Batch-Size	128, 256, 512, 1024
Lernrate	0.05, 0.025, 0.01, 0.0025, 0.005

Tabelle B.6: Einstellungen zur *fit()*-Methode des LSTM

Mittels der *fit()*-Methode wird das Training des neuronalen Netzes ausgeführt. Dieser Methode können unterschiedliche Parameter mitgegeben werden. Mittels der Batch-Size wird die Anzahl der Einträge im Datensatz bestimmt, die für einen Schritt des Gradientenabstiegs verwendet wird. Je mehr Dateneinträge für einen Schritt verwendet werden, desto genauer ist der berechnete Gradient. Insgesamt steigt allerdings auch der Berechnungsaufwand. Die Anzahl Epochen gibt die Anzahl der vollständigen Iterationen über den Trainingsdatensatz an.

[Aha+09] Michal Aharon, Gilad Barash, Ira Cohen und Eli Mordechai. "One Graph Is Worth a Thousand Logs: Uncovering Hidden Structures in Massive System Event Logs". In: *Machine Learning and Knowledge Discovery in Databases*. Hrsg. von Wray Buntine, Marko Grobelnik, Dunja Mladenić und John Shawe-Taylor. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 227–243. ISBN: 978-3-642-04180-8.

[AA19] Sridhar Alla und Suman Kalyan Adari. *Beginning Anomaly Detection Using Python-Based Deep Learning*. Springer, 2019.

[BHR00] Lasse Bergroth, Harri Hakonen und Timo Raita. "A survey of longest common subsequence algorithms". In: *Proceedings - 7th International Symposium on String Processing and Information Retrieval, SPIRE 2000* (2000), S. 39–48. DOI: 10.1109/SPIRE.2000.878178.

[CBK09] Varun Chandola, Arindam Banerjee und Vipin Kumar. "Anomaly Detection: A Survey". In: *ACM computing surveys (CSUR).* 41.3 (Juli 2009), S. 1–58. ISSN: 0360-0300. DOI: 10.1145/1541880.1541882. URL: https://doi.org/10.1145/1541880.1541882.

[CJK04] Nitesh V. Chawla, Nathalie Japkowicz und Aleksander Kotcz. "Editorial: Special Issue on Learning from Imbalanced Data Sets". In: *ACM SIGKDD Explorations Newsletter* 6.1 (2004), S. 1–6. ISSN: 1931-0145. DOI: 10.1145/1007730.1007733.

[DL17] Min Du und Feifei Li. "Spell: Streaming parsing of system event logs". In: *Proceedings - IEEE International Conference on Data Mining, ICDM* (2017), S. 859–864. ISSN: 15504786. DOI: 10.1109/ICDM.2016.160.

[Du+17] Min Du, Feifei Li, Guineng Zheng und Vivek Srikumar. "DeepLog: Anomaly detection and diagnosis from system logs through deep learning". In: *Proceedings of the ACM Conference on Computer and Communications Security* (2017), S. 1285–1298. ISSN: 15437221. DOI: 10.1145/3133956.3134015.

[Fu+09] Qiang Fu, Jian Guang Lou, Yi Wang und Jiang Li. "Execution anomaly detection in distributed systems through unstructured log analysis". In: *Proceedings - IEEE International Conference on Data Mining, ICDM* (2009), S. 149–158. ISSN: 15504786. DOI: 10.1109/ICDM.2009.60.

[Fu+14] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang und Tao Xie. "Where do developers log? An empirical study on logging practices in industry". In: *36th International Conference on Software Engineering, ICSE Companion 2014 - Proceedings* (2014), S. 24–33. DOI: 10.1145/2591062.2591175.

[FFH08] Ew Fulp, Ga Fink und Jn Haack. "Predicting Computer System Failures Using Support Vector Machines." In: *Proceedings of the First USENIX conference on Analysis of system logs* (2008), S. 5–5. URL: http://static.usenix.org/event/wasl/tech/full_papers/fulp/fulp_html/wasl08f.html.

[GSC00] Felix Gers, Jürgen Schmidhuber und Fred Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Neural computation* 12 (Okt. 2000), S. 2451–71. DOI: 10.1162/089976600300015015.

[Gre+17] Klaus Greff, Rupesh K. Srivastava, Jan Koutnik, Bas R. Steunebrink und Jürgen Schmidhuber. "LSTM: A Search Space Odyssey". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10 (2017), S. 2222–2232. ISSN: 21622388. DOI: 10.1109/TNNLS.2016.2582924. arXiv: 1503.04069.

[Guo+19] Shuting Guo, Zheng Liu, Wenyan Chen und Tao Li. "Event extraction from streaming system logs". In: *Lecture Notes in Electrical Engineering*. Bd. 514. 2019, S. 465–474. ISBN: 9789811310553. DOI: 10.1007/978-981-13-1056-0_47.

[HG09] Haibo He und Edwardo A Garcia. "Learning from imbalanced data". In: *IEEE Transactions on knowledge and data engineering* 21.9 (2009), S. 1263–1284.

[He+17] Pinjia He, Jieming Zhu, Zibin Zheng und Michael R. Lyu. "Drain: An Online Log Parsing Approach with Fixed Depth Tree". In: *Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017* (2017), S. 33–40. DOI: 10.1109/ICWS.2017.13.

[He+16] Shilin He, Jieming Zhu, Pinjia He und Michael R. Lyu. "Experience Report: System Log Analysis for Anomaly Detection". In: *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*. IEEE Computer Society, 2016, S. 207–218. DOI: 10.1109/ISSRE.2016.21. URL: https://doi.org/10.1109/ISSRE.2016.21.

[HS97] Sepp Hochreiter und Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), S. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[HA04] Victoria Hodge und Jim Austin. "A survey of outlier detection methodologies". In: *Artificial intelligence review* 22.2 (2004), S. 85–126. DOI: 10.1023/B:AIRE.0000045502.10941.a9. URL: https://doi.org/10.1023/B:AIRE.0000045502.10941.a9.

[Lan+18] Max Landauer, Markus Wurzenberger, Florian Skopik, Giuseppe Settanni und Peter Filzmoser. "Dynamic log file analysis: An unsupervised cluster evolution approach for anomaly detection". In: *Computers and Security* 79 (2018), S. 94–116. ISSN: 01674048. DOI: 10.1016/j.cose.2018.08.009. URL: https://doi.org/10.1016/j.cose.2018.08.009.

[Lin+16] Qingwei Lin, Hongyu Zhang, Jian Guang Lou, Yu Zhang und Xuewei Chen. "Log clustering based problem identification for online service systems". In: *Proceedings - International Conference on Software Engineering* (2016), S. 102–111. ISSN: 02705257. DOI: 10.1145/2889160.2889232.

[Lou+10] Jian Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu und Jiang Li. "Mining invariants from console logs for system problem detection". In: *Proceedings of the 2010 USENIX Annual Technical Conference, USENIX ATC 2010* (2010), S. 231–244.

[Lu+18] Siyang Lu, Xiang Wei, Yandong Li und Liqiang Wang. "Detecting anomaly in big data system logs using convolutional neural network". In: *Proceedings - IEEE 16th International Conference on Dependable, Autonomic and Secure Computing, IEEE 16th International Conference on Pervasive Intelligence and Computing, IEEE 4th International Conference on Big Data Intelligence and Computing and IEEE 3* (2018), S. 159–165. DOI: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00037.

[MZHM09] Adetokunbo Makanju, A. Nur Zincir-Heywood und Evangelos E. Milios. "Clustering event logs using iterative partitioning". In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2009), S. 1255–1263. DOI: 10.1145/1557019.1557154.

[MZHM12] Adetokunbo Makanju, A. Nur Zincir-Heywood und Evangelos E. Milios. "A lightweight algorithm for message type extraction in system application logs". In: *IEEE Transactions on Knowledge and Data Engineering* 24.11 (2012), S. 1921–1936. ISSN: 10414347. DOI: 10.1109/TKDE.2011.138.

[MP08] Leonardo Mariani und Fabrizio Pastore. "Automated identification of failure causes in system logs". In: *Proceedings - International Symposium on Software Reliability Engineering, ISSRE* (2008), S. 117–126. ISSN: 10719458. DOI: 10.1109/ISSRE.2008.48.

[MK12] Yuxin Meng und Lam-for Kwok. "Adaptive False Alarm Filter Using Machine Learning in Intrusion Detection". In: *Practical Applications of Intelligent Systems*. Hrsg. von Yinglin Wang und Tianrui Li. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 573–584. ISBN: 978-3-642-25658-5. DOI: https://doi.org/10.1007/978-3-642-25658-5_68.

[MVM09] Frederic P. Miller, Agnes F. Vandome und John McBrewster. *Levenshtein Distance: Information Theory, Computer Science, String (Computer Science), String Metric, Damerau?Levenshtein Distance, Spell Checker, Hamming Distance*. Alpha Press, 2009. ISBN: 6130216904.

[MG16] Andreas C. Müller und Sarah Guido. *Introduction to Machine Learning with Python*. O'Reilly Media, Inc., 2016. ISBN: 978-1-449-36941-5.

[OS07] Adam Oliner und Jon Stearley. "What supercomputers say: A study of five system logs". In: *Proceedings of the International Conference on Dependable Systems and Networks* (2007), S. 575–584. DOI: 10.1109/DSN.2007.103.

[Pol+01] Robi Polikar, Lalita Upda, Satish S Upda und Vasant Honavar. "Learn++: An incremental learning algorithm for supervised neural networks". In: *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)* 31.4 (2001), S. 497–508.

[SBK19] Alireza Vafaei Sadr, Bruce A. Bassett und Martin Kunz. "A flexible framework for anomaly detection via dimensionality reduction". In: *2019 6th International Conference on Soft Computing and Machine Intelligence, ISCMI 2019* (2019), S. 106–110. DOI: 10.1109/ISCMI47871.2019.9004400. arXiv: 1909.04060.

[SG19] Monika Schak und Alexander Gepperth. "A Study on Catastrophic Forgetting in Deep LSTM Networks". In: *Artificial Neural Networks and Machine Learning – ICANN 2019: Deep Learning*. Hrsg. von Igor V. Tetko, Věra Kůrková, Pavel Karpov und Fabian Theis. Cham: Springer International Publishing, 2019, S. 714–728. ISBN: 978-3-030-30484-3.

[SCB20] Shagun Sodhani, Sarath Chandar und Yoshua Bengio. "Toward training recurrent neural networks for lifelong learning". In: *Neural Computation* 32.1 (2020), S. 1–35. ISSN: 1530888X. DOI: 10.1162/neco_a_01246. arXiv: arXiv:1811.07017v3.

[SKK12] Janusz Sosnowski, Marcin Kubacki und Henryk Krawczyk. "Monitoring event logs within a cluster system". In: *Advances in Intelligent and Soft Computing* 170 AISC (2012), S. 257–271. ISSN: 18675662. DOI: 10.1007/978-3-642-30662-4-17.

[SM01] R. William Soukoreff und I. Scott MacKenzie. "Measuring errors in text entry tasks". In: (2001), S. 319. DOI: 10.1145/634253.634256.

[TLP11] Liang Tang, Tao Li und Chang Shing Perng. "LogSig: Generating system events from raw textual logs". In: *International Conference on Information and Knowledge Management, Proceedings* (2011), S. 785–794. DOI: 10.1145/2063576.2063690.

[VKP16] Risto Vaarandi, Markus Kont und Mauno Pihelgas. "Event log analysis with the LogCluster tool". In: *Proceedings - IEEE Military Communications Conference MILCOM* (2016), S. 982–987. DOI: 10.1109/MILCOM.2016.7795458.

[WWM04] Yanxin Wang, Johnny Wong und Andrew Miner. "Anomaly intrusion detection using one class SVM". In: *Proceedings fron the Fifth Annual IEEE System, Man and Cybernetics Information Assurance Workshop, SMC* (2004), S. 358–364. DOI: 10.1109/iaw.2004.1437839.

[Xu+10] Wei Xu, Ling Huang, Armando Fox, David Patterson und Michael I. Jordan. "Detecting large-scale system problems by mining console logs". In: *ICML 2010 - Proceedings, 27th International Conference on Machine Learning* (2010), S. 37–44.

[Yu+19] Yong Yu, Xiaosheng Si, Changhua Hu und Jianxun Zhang. "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures". In: *Neural Computation* 31.7 (2019). PMID: 31113301, S. 1235–1270. DOI: 10.1162/neco_a_01199. eprint: https://doi.org/10.1162/neco_a_01199. URL: https://doi.org/10.1162/neco_a_01199.

[YPZ12] Ding Yuan, Soyeon Park und Yuanyuan Zhou. "Characterizing logging practices in open-source software". In: *Proceedings - International Conference on Software Engineering* (2012), S. 102–112. ISSN: 02705257. DOI: 10.1109/ICSE.2012.6227202.