Technology
Arts Sciences
TH Köln

DLR

Event-Log Analyse mittels Clustering und Mustererkennung

MASTERARBEIT

ausgearbeitet von:

Sebastian Wiesendahl

eingereicht zur Erlangung des akademischen Grades

MASTER OF SCIENCE (M.SC.)

vorgelegt an der

TECHNISCHEN HOCHSCHULE KÖLN
CAMPUS GUMMERSBACH
FAKULTÄT FÜR INFORMATIK UND INGENIEURWISSENSCHAFTEN

im Studiengang

INFORMATIK / COMPUTER SCIENCE

Erstprüfer/in: Prof. Dr. Heide Faeskorn-Woyke
TH Köln

Zweitprüfer/in: Dr. rer. nat. Brigitte Boden
DLR

Gummersbach, Oktober 2018

Kontakt: Sebastian Wiesendahl
sebastian.wiesendahl@smail.th-koeln.de

Prof. Dr. Heide Faeskorn-Woyke
Technische Hochschule Köln (TH Köln)
Institut für Informatik
Steinmüllerallee 1
51643 Gummersbach
heide.faeskorn-woyke@th-koeln.de

Dr. rer. nat. Brigitte Boden
Deutsches Zentrum für Luft- und Raumfahrt e. V. (DLR)
Simulations- und Softwaretechnik
Intelligente und verteilte Systeme
Linder Höhe
51147 Köln
brigitte.boden@dlr.de

Kurzfassung

Die Analyse von Log-Dateien als Spezialfall des Text-Mining dient in der Regel dazu Laufzeitfehler oder Angriffe auf ein Systems nachzuvollziehen. Gegen erkannte Fehlerzustände können Maßnahmen ergriffen werden, um diese zu vermeiden. Muster in semi-strukturierten Log-Dateien aus dynamischen Umgebungen zu erkennen, ist komplex und erfordert einen mehrstufigen Prozess. Zur Analyse werden die Log-Dateien in einen strukturierten Event-Log (*event log*) überführt. Diese Arbeit bietet dem Anwender ein Werkzeug, um häufige (*frequent*) oder seltene (*rare*) Ereignisse (*events*), sowie temporale Muster (*temporal patterns*) in den Daten zu erkennen. Dazu werden verschiedene Techniken des Data-Mining miteinander verbunden. Zentrales Element dieser Arbeit ist das Clustering mittels verschiedener Dimensionsreduktionstechniken. Es wurde untersucht, ob durch Neuronale Netze mittels unüberwachtem Lernen durch *Autoencoder* (AE) geeignete Repräsentationen (*embeddings*) von Ereignissen erstellt werden können, um syntaktisch und semantisch ähnliche Instanzen zusammenzufassen. Dazu wurde ein tiefes Neuronales Netz entwickelt (DeepKATE), das auf KATE [13] basiert und als allgemeines Dimensionsreduktions Werkzeug fungiert. Dies dient zur Einbettung von Ereignissen, Erkennung von Ausreißern (*outlier detection*), sowie zur Inferenz einer nachvollziehbaren textuellen Repräsentation durch *Regular Expressions* (RE). Um verborgene temporale Muster in den Daten zu finden, werden diese auf *Auftritte* von *seriellen Episoden* untersucht. Durch *Episode Mining* (EM) können alle *minimalen Auftritte serieller Episoden* in einer Sequenz gefunden werden. Der enorme Suchraum erfordert effektive und effiziente Algorithmen, um in angemessener Zeit Ergebnisse zu erzielen. Das Clustering dient ebenfalls zur Reduktion (*pruning*) des Suchraums für das Episode Mining. Um die Menge der Ergebnisse einzuschränken wurden verschiedene Strategien des Episode Mining auf ihre praktische Tauglichkeit hin untersucht. Das Episode Mining findet unter Anwendung verschiedener Kriterien (*constrains*) statt: über den *Minimum Support* (*Frequent Episode Mining* - FEM) und die *Nützlichkeit* (*High Utility Episode Mining* - HUEM), sowie drei weiteren Kriterien. Die entstandenen Episode Mining Algorithmen MV-Span und MT-Span basieren auf unterschiedlichen Strategien des Präfix-Wachstums (*prefix-growth*). MV-Span nutzt die Strategie einer vertikalen Projektion, wobei MT-Span auf TSpan [25] aufbaut.

Schlüsselworte: event log, clustering, clustering validation index, neural networks, autoencoder, outlier detection, constrained pattern mining, serial episodes, high utility episode mining

Inhaltsverzeichnis

Kurzfassung iii

Inhaltsverzeichnis iv

Abbildungsverzeichnis vi

Tabellenverzeichnis vii

Abkürzungsverzeichnis ix

1 Einleitung 1
- 1.1 Motivation 1
- 1.2 Kapitelübersicht 1

2 Grundlagen 3
- 2.1 Event-Log 3
- 2.2 Sprachverarbeitung (NLP) 6
  - 2.2.1 Einbettung (Embedding) 7
- 2.3 Künstliche Neuronale Netze (KNN) 8
  - 2.3.1 Architektur 9
    - Autoencoder (AE) 10
    - Long Short-Term Memory (LSTM) 12
  - 2.3.2 Lernen 13
    - Aktivierungsfunktion 13
    - Zielfunktion 13
    - Fehlerrückführung (Backpropagation) 16
    - Automatisches Differenzieren (AD) 17
- 2.4 Clustering 18
  - 2.4.1 Clustering Auswertung (Clustering Validation) 19
- 2.5 Mustererkennung in Sequenzen 23
  - 2.5.1 Frequent Episode Mining (FEM) 23
  - 2.5.2 High Utility Episode Mining (HUEM) 26
  - 2.5.3 Episode Rule Mining 26

3 Methodik 28
- 3.1 Anforderungen und Spezifikation 28
  - 3.1.1 Anforderungen 28
  - 3.1.2 Spezifikation 28
- 3.2 Algorithmen und Datenstrukturen 29
  - 3.2.1 Datenstrukturen 30
  - 3.2.2 Event-Log Parser 31
  - 3.2.3 Modellierung Neuronaler Netze 33
    - DeepKATE 36
  - 3.2.4 Clustering 39

3.2.5 Detektion von Anomalien 40
3.2.6 Log-Key Inferenz 40
3.2.7 Episoden Mining (EM) 42
MV-Span 42
MT-Span 46
3.2.8 Eventvoraussage 49

4 Ergebnisse **51**
4.1 DeepKATE 51
4.2 Episode Mining 55

5 Zusammenfassung **56**
5.1 Fazit 56

Literatur **59**

DeepKATE **67**

Anforderungen **73**

Sourcecode und Ergebnisse **75**

Eidesstattliche Erklärung **76**

Abbildungsverzeichnis

2.1 Autoencoder 11
2.2 Aktivierungsfunktionen 14
2.3 Backpropagation 17

3.1 Informationsfluss zwischen Komponenten von *LogClustering.jl* 29
3.2 Komplement der Funktion Repel 39

4.1 Korrelationsmatrix verschiedener *Clustering Validity Indices* 52
4.2 Einbettung mittels DeepKATE (CICIDS2017) 54
4.3 MV-Span vs. MT-Span (synthetischer Datensatz) 55

Tabellenverzeichnis

2.1 Eine *Event Sequenz* mit 16 Events. 25
2.2 Eine Zuordnung zwischen Eventtyp und externer Nützlichkeit 26

Quellcodeverzeichnis

2.1 One-Hot-Encoding Beispiel . . . 7
3.1 Event-Log Parser . . . 31
3.2 Tokenisierung geparster Log-Zeilen zu Log-Keys und Log-Attributen . 32
3.3 Fortlaufende Zeit von Events extrahieren . . . 33
3.4 Frequenznormalisierung mittels maximalem Auftreten . . . 34
3.5 KATE Layer Struktur . . . 34
3.6 KATE Layer Verhalten . . . 35
3.7 DeepKATE Modell mittels Flux.jl . . . 37
3.8 DeepKATE Zielfunktion mittels Flux.jl . . . 38
3.9 Hilfsprozedur Repel . . . 38
3.10 Log-Key Inferenz . . . 40
3.11 MV-Span-Wachstum . . . 43
3.12 MV-Span Algorithmus . . . 45
3.13 MT-Span S-Concatenation . . . 47
3.14 MT-Span Algorithmus . . . 49

Abkürzungsverzeichnis

Abkürzungen

AAE Adversarial Autoencoder

AD Automatische Differenzierung

AE Autoencoder

CVI Clustering Validity Index (Metrik oder Maß zur Güte eines Clustering)

DM Data Mining

EM Episode Mining (Mustererkennung von Episoden)

FEM Frequent Episode Mining (Mustererkennung von Episoden mit einer Häufigkeits-Bedingung)

GloVe Global Vectors for Word Representation

HUEM High Utility Episode Mining (Mustererkennung von Episoden hoher Nützlichkeit)

ID Identifier oder *inter-cluster density* (siehe *Dense_bw* 2.4.1)

KNN Künstliches neuronales Netz

LDA latent Dirichlet allocation (dreischichtiges generatives Wahrscheinlichkeitsmodell)

LSTM Long short-term memory (siehe 2.3.1)

MD mean deviation from the median (mittlere absolute Abweichung vom Median)

ML Machine Learning (Maschinelles Lernen)

MSE mean squared error (mittlere quadratische Abweichung)

NLP Natural Language Processing (maschinelle Verarbeitung natürlicher Sprache)

RE Regular Expression (Regulärer Ausdruck)

ReLU Rectified Linear Unit

RNN Rekurrentes Neuronales Netz (Recurrent Neural Network) (siehe 2.3.1)

t-SNE t-Distributed Stochastic Neighbor Embedding

Tf-idf term frequency–inverse document frequency

UMAP Uniform Manifold Approximation and Projection for Dimension Reduction

VAE Variational Autoencoder

Kapitel 1

Einleitung

1.1 Motivation

Das Auffinden von ungewöhnlichen Ereignissen, sowie das Erkennen von regelmäßigen Mustern in Log-Dateien ist für einen Analysten eine große Herausforderung, da die zu verarbeitenden Datenmengen stetig steigen und der Inhalt von Log-Dateien meist semi-strukturiert ist.

Diese Arbeit soll es ermöglichen sowohl in großen Mengen von Log-Events Ausreißer ausfindig zu machen, sowie auftretende Muster durch generische Repräsentation wiederzugeben. Mit Mustern sind zusammenfassende Regeln (Regulärer Ausdruck) für ähnliche Log-Zeilen gemeint, die einer einzelnen Gruppe zugeordnet werden – einem *Cluster*. Neben dem Bilden von Clustern – dem *Clustering* – werden mittels temporaler Mustererkennung im Event-Log als Sequenz (*Pattern Mining*) zeitliche Muster von Interesse aufgedeckt.

In dieser Arbeit werden vier Probleme angegangen. Zum einen das Problem Log-Events variabler Länge sowohl syntaktisch als auch semantisch miteinander zu vergleichbar zu machen. Zum anderen sollen ähnliche Log-Events durch Clustering so partitioniert werden, dass eindeutige *reguläre Ausdrücke* (RE) aus den Instanzen eines Clusters inferiert werden können. Sowie schließlich das Problem in der großen Menge von möglichen zeitlichen Mustern relevante Muster durch *High Utility Episode Mining* (HUEM) zu erkennen.

Für diese Arbeit sind Arbeiten aus verschiedenen Disziplinen relevant. Zunächst sollen die Attribute der Log-Events durch Dimensionsreduktionstechniken eingebettet werden. Es wird untersucht, inwiefern sich Autoencoder unterschiedlicher Architekturen (siehe 2.3.1) zur Einbettung textueller Log-Events eignen, um diese schließlich syntaktisch und semantisch zu Clustern.

1.2 Kapitelübersicht

Im folgendem Kapitel 2 wird auf die Grundlagen der Arbeit eingegangen. In dem Kapitel werden zum einen formale Problemdefinition gegeben. Zum anderen findet eine Literaturuntersuchung statt, die verschiedene aktuelle und relevante Arbeiten in den Zusammenhang mit dieser Arbeit stellt.

Im dritten Kapitel 3 werden die Anforderungen an die Arbeit als Softwareprojekt, sowie dessen konkrete Umsetzung durch eine Spezifikation definiert. Danach werden, die in der Arbeit angewandten Algorithmen und Datenstrukturen für die unterschiedlichen Schritte des *Data Mining*-Prozesses (DM) erläutert. Die Ansätze der Arbeit zur Problemlösung werden vorgestellt und als Experimente definiert. Dabei werden verschiedene Modelle des Maschinellen Lernens mittels künstlicher

Neuronaler Netze, des Clustering und der Mustererkennung in Sequenzen angewandt.

Das vierte Kapitel 4 befasst trägt die Ergebnisse der verschiedenen Versuchsaufbauten zusammen. Es wird gezeigt, wie sich die Modelle auf unterschiedlichen Datensätzen verhalten und ob diese dazu in der Lage sind generische Muster in Log-Dateien zu finden. Die Ergebnisse werden dabei sowohl quantitativ, durch verschiedene Metriken, sowie qualitativ an Hand anekdotischer Beispiele ausgewertet. Eine qualitative Auswertung findet dort statt, wo eine objektive Betrachtung nicht möglich ist.

Das fünfte Kapitel 5 fasst die Erkenntnisse der Arbeit zusammen. Es wird auf gelöste, sowie ungelöste Probleme eingegangen. Zudem wird neben dem Fazit auf mögliche Erweiterungen und Forschungsgegenstände hingewiesen.

Kapitel 2

Grundlagen

In diesem Kapitel werden grundlegende Begriffe definiert und relevante Techniken und Methoden erläutert. Diese dienen als Basis für die Methodik (Kapitel 3) in dem die Techniken und Methoden mittels konkreter oder abgewandelter Modelle und Algorithmen in Zusammenhang zur Arbeit gebracht werden.

Da in dieser Arbeit durch den KDD Prozess viele unterschiedliche Teilgebiete genutzt werden, biete ich dem Leser in diesem Kapitel die Möglichkeit sich einen Überblick zu verschaffen und Definitionen zu harmonisieren, um diese in der Arbeit anzuwenden.

In den folgenden Abschnitten werden zunächst die Eigenheiten der Teilgebiete hervorgehoben und durch eine jeweils kurze Literaturuntersuchung ergänzt, um aufzuzeigen, welche Verfahren *state-of-the-art* sind.

Im nächsten Abschnitt wird auf den Untersuchungsgegenstand der Log-Dateien eingegangen. Es wird der Unterschied zwischen einer unstrukturierten Log-Datei und der abgeleiteten strukturierten Form eines *Event-Logs* dargestellt.

2.1 Event-Log

Log-Dateien (*Log*) fallen in verschieden Formen und durch unterschiedlichste System an. Sie dienen zur Nachvollziehbarkeit von Ereignissen (*Event*) in einer Anwendung, indem relevante Ereignisse in einer aufgezeichnet werden - dieser Vorgang wird als *Logging* bezeichnet. Log-Dateien folgen dabei in der Regel einem anwendungsspezifischem Schema. Dies ermöglicht die maschinelle Verarbeitung durch andere Systeme, bietet jedoch eine große Herausforderung, wenn Log-Dateien verschiedener Systeme verarbeitet werden sollen.

Diese Arbeit befasst sich hauptsächlich mit Log-Dateien der RCE (Remote Component Environment) Anwendung, welches durch den DLR entwickelt wird [70]. RCE ist eine verteilte Integrationsumgebung für komplexe Systeme. Sie dient zum Entwurf, Analyse und Optimierung z.B. von Flugzeugen, Schiffen oder Satelliten. Den Anwendern wird dabei ermöglicht eigene Systeme zu integrieren und mittels RCE zu *Workflows* zu verbinden. Die Software wird als Open Source Projekt zur Verfügung gestellt [1].

Bei einem *Event-Log* handelt es sich um eine strukturierte Datenstruktur einer *Log-Datei*, die aus der *semi-strukturierten* Log-Datei (Quelle) abgeleitet wird.

Definition 2.1 (Log-Zeile)**.** Bei einer *Log-Zeile* handelt es sich um eine Zeile einer *Log-Nachricht* in einem *Log*. Dabei kann eine *Log-Nachricht* mehrere Zeilen umfassen.

[1] Allgemeine Informationen und Veröffentlichungen von RCE sind auf der Projektseite abrufbar: `http://rcenvironment.de/`. Sie Software wird zudem auf der Open Source Plattform Github zur Verfügung gestellt: `https://github.com/rcenvironment/rce`

In dieser Arbeit wird die Analyse auf der Ebene von *Log-Zeilen* und nicht etwa *Log-Nachrichten* untersucht.

Beispiel 2.1. Eine *Log-Zeile* der RCE Anwendung:

```
2016-01-28 12:07:27,624 DEBUG - de.rcenvironment.core.datamanagement.internal.
    FileDataServiceImpl - Finished uploading 652261 bytes for upload id ea1b6ca4
    -4152-49ed-ac81-51c08c29ff05; polling for remote data reference
```

Definition 2.2 (Event-Typ)**.** Bei einem *Event-Typen* handelt es sich um ein Token, dass die Zugehörigkeit einer *Log-Nachricht* (bzw. *Log-Zeile*) zu einer anwendungsspezifischen Klasse ausdrückt.

Definition 2.3 (Log-Level)**.** Beim Logging gibt es hierarchische Kategorisierungen nach *Log-Level*. Diese stellen sicher, dass Events bestimmten Typs, in eine Log-Datei übertragen werden, falls ein solches Event in der Anwendung auftritt, dass dem aktuellen *Log-Level* entspricht.

Beispiel 2.2. Eine typische *Log-Level* Hierarchie in Java ist z.B. durch "log4j"[39] gegeben:

ALL < **TRACE** < **DEBUG** < **INFO** < **WARN** < **ERROR** < **FATAL** < **OFF**

In Python ist die Hierarchie z.B. in der *builtin* Bibliothek "logging"[1] wie folgt definiert:

NOTSET < **DEBUG** < **INFO** < **WARNING** < **ERROR** < **CRITICAL**

Wie man an den Beispielen (2.2) erkennen kann, gibt es weder eine einheitliche Definition für *Event-Typen*, noch für *Log-Level*. Zudem bieten viele Logging-Bibliotheken die Möglichkeit eigene Typen zu definieren und einer Log-Level Hierarchie hinzuzufügen.

Definition 2.4 (Log-Key)**.** Ein *Log-Key* ist eine Repräsentation einer Log-Zeile. Dieser enthält statische und variable Teile. Dabei wird der statische Teil einer Log-Zeile explizit als Zeichenfolge dargestellt und variable Teile mit einem *Platzhalter* Token, meist $*$.

Beispiel 2.3. Der *Log-Key* für gegebene *Log-Zeile* (2.1) könnte, wie folgt aussehen:

```
* DEBUG - * - Finished uploading * bytes for upload id *; polling for remote data
    reference
```

Oft ist das originale Format eines *Log-Keys* unbekannt, wenn der Quellcode einer Anwendung nicht vorliegt. Es gilt daher das Format eines *Log-Keys* zu approximieren und durch Heuristiken zu erschließen, welche Teile variabel sind und die Attribute (siehe Definition 2.6) ergeben. Die Begriffe des *Log-Keys* und *Log-Attributen* wurden von [20] übernommen.

Definition 2.5 (Beschreibender Log-Key)**.** Bei einem *beschreibenden Log-Key* handelt es sich um eine Erweiterung von *Log-Keys*. Jeder variable Teil ($*$) wird durch einen eindeutigen Bezeichner (*Label*) ersetzt, das einen Regulären Ausdruck (*Regular Expression* - RE) repräsentiert.

Beispiel 2.4. Ein *beschreibenden Log-Key* mit vier *Label*:

```
%TIMESTAMP% DEBUG - %URI% - Finished uploading %INT% bytes for upload id %HEX_ID%;
    polling for remote data reference
```

Die Zuordnung zwischen *Label* und *Regulärem Ausdruck* ist vom Anwendungsfall abhängig. Der Reguläre Ausdruck für das `%TIMESTAMP%` *Label* könnte wie folgt aussehen: `r"\b\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2},\d{3}\b"`. Neben dieser konkreten RE gibt es viele weitere Möglichkeiten einen validen Ausdruck zu erstellen, welcher die selbe Zielmenge erfasst.

Definition 2.6 (Attribute eines Log-Keys)**.** Die *Attribute* (*parameter values*) eines *Log-Keys* sind ein n-dimensionaler Vektor, wobei n der Anzahl der Variablen im zugehörigen *Log-Key* entspricht. Jede Komponente der Attribute stellt einen Wert dar, der sich aus der zugehörigen *Log-Zeile* des *Log-Keys* ergibt.

Beispiel 2.5. Vier *Attribute* des *Log-Keys* aus Beispiel 2.3 oder des *beschreibenden Log-Keys* aus Beispiel 2.3 für die *Log-Zeile* aus Beispiel 2.1:

```
("2016-01-28 12:07:27,624", "de.rcenvironment.core.datamanagement.internal.
    FileDataServiceImpl", 652261, "ea1b6ca4-4152-49ed-ac81-51c08c29ff05")
```

Definition 2.7 (Event-Log)**.** Ist ein geordnete Menge von Tupeln (i, k, p, v), bestehend aus dem Index einer Log-Zeile i in einer Log-Datei, dem zugeordnetem Identifier (ID) eines *Log-Keys* k, dem *Log-Key* p selbst und den Attributen v des *Log-Keys*.

Die Zuordnung zwischen *Log-Key* und Log-Zeile ist nicht eindeutig, wenn ein Token aus mehreren Teilen besteht, wie etwa hier der Zeitstempel im Beispiel der Log-Zeile. Um von einer Log-Zeile auf einen *Log-Key* zu schließen ist eine Heuristik nötig, die eine solche Zuordnung vornimmt.

Andere Arbeiten, die sich mit dem verarbeiten von *Event-Logs* auseinander gesetzt haben sind unter anderem folgende.

Mit dem Parsen von Log-Dateien und der Generierung von *Log-Keys* haben sich bereits viele Arbeiten befasst. Dabei kann der Anwender in der Regel Reguläre Ausdrücke definieren, welche die *Attribute* einer *Log-Zeile* erkennen. Es wird dabei zwischen *offline* (*batch*) und *online* Verarbeitung unterschieden. Aus der Familie der online verarbeitenden Systemen wird durch [20] Spell [19] als *state-of-the-art* System genannt. Der Parser Spell steht nicht frei zur Verfügung.

Aus der Familie der offline verarbeitenden Systemen schaffte das *Simple Logfile Clustering Tool* (SLCT) [75] eine Grundlage, die durch LogClusterP [77, 76] (wobei P für die Programmiersprache Perl steht) und LogClusterC [88] (C, für die Programmiersprache C) weiter vorangetrieben wurde. Neben dem Clustering von *Log-Keys* durch Fuzzy-Sets bieten diese Werkzeuge dem Anwender die Möglichkeit Ausreißer unter den *Log-Keys* zu detektieren.

Zudem nutzt [77] *Log-Keys* mit Quantoren (*quantifier*), welche die Menge an Token bestimmen, die für einen Platzhalter auftreten dürfen. Die Token werden durch einen vom Nutzer definierten Regulären Ausdruck voneinander getrennt (Standard: `r"\s+"`).

Beispiel 2.6. Ein quantifizierter *Log-Key* könnte, wie folgt aussehen (nach [77]):

```
User *{1,1} login from *{1,1}
```

Diese Eigenschaft wird genutzt um überlappende Cluster (*soft* Clustering - siehe Abschnitt 2.4) zu ermöglichen und die Auftritte (*support*) von *Log-Keys* zu vereinigen. Folgende spezifischere *Log-Keys* werden durch den *quantifitzierten Log-Key* aus Beispiel 2.6 nach [77] ebenfalls abgedeckt: `User bob login from 10.1.1.1` oder `User *{1,1} login from 10.1.1.1`.

DeepLog [20] ist ein umfassendes System, welches sich mit der Erkennung temporaler Muster und der Ableitung von *Workflows* befasst. Bei einem Workflow handelt es sich um eine *Komplexe Event Sequenz* (*complex event sequence*), also einen gerichteten Graphen mit *parallelen* und *seriellen* Events (siehe Abschnitt 2.5). Es werden endliche Automaten (finite state automaton - FSA,FSM) gebildet, um einen *Workflow* erkennen zu können. Dabei werden LSTM-Netze (siehe Abschnitt 2.3.1) genutzt, um ein Modell der temporalen Abläufe zu erstellen. Die trainierten Netze werden dienen dazu ein nächstes Event vorauszusagen. Falls die Voraussage und der tatsächliche Auftritt eines Events übereinstimmen, wird dieses Kriterium dazu genutzt einen bestehenden *Workflow* zu erweitern. Falls die Voraussage nicht übereinstimmt, wird die Konstruktion eines neuen *Workflows* begonnen.

Im nächsten Abschnitt wird darauf eingegangen, wie Textanalyse mittels Sprachverarbeitung betrieben werden kann. Die Annahme dahinter ist, Log-Dateien als Text zu interpretieren und mit Mitteln der Sprachverarbeitung semantisch und syntaktisch zugänglich zu machen.

2.2 Sprachverarbeitung (NLP)

Die maschinelle Verarbeitung natürlicher Sprache (*Natural Language Processing* - NLP) ist ein weites Forschungsfeld. Sprache zeichnet sich neben ihrer Grammatik vor allem dadurch aus, dass sie besonders dünn besetzt ist, da nicht jede beliebige Zeichenfolge erlaubte Worte oder Sätze bilden. Dies spiegelt sich in der Häufigkeitsverteilung von Buchstaben, Worten oder ganzen Sätzen wider, die meist einer Zipf-Verteilung (diskrete Exponentialverteilung) folgen. Unter Anwendung von Vorwissen bietet Sprache daher ein hohes Maß zur Komprimierung, durch die Wahl einer geeigneten Kodierung in Bezug auf ihre tatsächliche Verteilung.

Ein Satz besteht aus Worten (*Token*), die wiederum aus Zeichen eines endlichen *Alphabets A* bestehen. Binär kann ein Satz als dünn-besetzter dreidimensionaler Vektorraum (*sparse representation*) über alle Zeichen des Alphabets dargestellt werden, wie beispielhaft in 2.2 als ineinander verschachtelte Vektoren. Diese Kodierung wird *One-Hot-Encoding* genannt, da jedes aktive Bit ein Zeichen aus dem Alphabet repräsentiert.

```
[
  [
    [0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0]
  ],[
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]
  ],[
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0],
    [0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
```

```
    [0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0]
  ]
]
```

QUELLCODE 2.1: *One-Hot-Encoding* für die Worte (*Token*) "hallo", ", ", "welt" mit dem Alphabet: [a,b,c,d,e,f,g,h,i,j,k, l,m,n,o,p,q,r,s,t,u,v,w,x,y,z," ",","].

Eine solche Darstellung ist nicht nur ineffizient, sondern bietet eine besondere Herausforderung beim Maschinellen Lernen, da sie, wie viele andere Textkodierungen, dem Fluch der Dimensionalität unterliegt. Das bedeutet, dass mit Zunahme der Dimensionen die messbaren Abstände von Instanzen im Raum immer geringer werden. Der Suchraum wächst mit jeder Dimension exponentiell. Je nach Repräsentation eines Texts, entspricht die Dimensionalität der Anzahl der Zeichen im *Alphabet* oder etwa der Anzahl einzigartiger Worte im Text, dem *Vokabular* V. Vorteil von von dünn-besetzten Kodierungen ist, dass diese eindeutig sind. Beim Maschinellen Lernen gilt es ein Kompromiss zu finden, zwischen möglichst eindeutigen Kodierungen, die jedoch nicht übermäßig dimensional sind.
Wird in dieser Arbeit von der *Zeichenebene* gesprochen, ist damit gemeint, dass der Text durch Zeichen des Alphabets kodiert ist. Die *Wortebene* meint, dass ein Wort durch einen skalaren Wert, wie dem *Index* im Vokabular oder seiner *Frequenz* im Text repräsentiert wird. Werden Worte oder Sätze durch einen n-dimensionalen Vektor repräsentiert wird von einer Einbettung (*distributed representation*) gesprochen.
Das Spektrum der Kodierung von Text reicht daher von dünn-besetzen Verfahren, wie dem *One-Hot-Encoding*, über n-dimensionale verteile Einbettungen, wie z.B. *Word2Vec* bis hin zu 1-dimensionalen Verfahren, wie etwa der Darstellung der Auftritte (*Frequenz*) eines Wortes im Text als Skalar, wie z.B. dem Tf-idf-Maß.
Im nächsten Abschnitt wird darauf eingegangen, welche Vor- und Nachteile die Einbettung von Texten für das Maschinelle Lernen mit sich bringen.

2.2.1 Einbettung (Embedding)

Das Problem Text unterschiedlicher Länge durch eine Repräsentation fester Länge darzustellen wird Einbettung genannt. Eine Einbettung kann dabei auf verschiedenen semantischen Ebenen stattfinden, wie Worten, Sätzen oder ganzen Dokumenten. In dieser Arbeit wird untersucht, wie mittels verschiedener Verfahren eine hinreichend gute Repräsentation von Log-Zeilen (Sätzen) erzeugt werden kann, um diese in syntaktisch und semantisch kohärente Cluster zu überführen (siehe Abschnitt 2.4).
Formal ausgedrückt: Eine Sequenz S der Länge N soll in eine Sequenz S' der Länge M überführt werden, wie in 2.1 definiert.

$$S \rightarrow S' \tag{2.1}$$

Im Allgemeinen ist eine Einbettung eine Reduktion der Dimensionalität, eine dichtere Repräsentation. Die Herausforderung liegt darin, eine Repräsentation zu erzeugen, die sowohl syntaktisch, als auch semantisch generalisiert. Syntaktisch ähnliche Worte sind z.B. "bauch" und "rauch" oder semantisch "zwei" und "2". Wie ähnlich sich zwei Worte sind, hängt von der Definition des Ähnlichkeitsmaßes ab und

ist daher subjektiv durch die Wahl des Maßes, wie etwa der *Hamming*- oder *Levenshtein*-Distanz, sowie der *Kosinus-Ähnlichkeit*.
Der Vorteil von Einbettung gegenüber der Repräsentation durch die Häufigkeit eines Tokens (Wort) im Text, seiner *Frequenz*, liegt darin, dass diese weniger Kollisionen aufweisen. Werden Token durch ihre Frequenz dargestellt, können gleich häufig vorkommende Worte, die jedoch syntaktisch, als auch semantisch eine große Entfernung zueinander haben, durch den selben skalaren Wert wiedergegeben werden.
Der Nachteil von Einbettung ist, dass diese nicht immer eindeutig sind. Dies trifft besonders für niedrigdimensionale Einbettungen zu, wenn Instanzen verschiedener Klassen im eingebetteten Raum nahe beieinander liegen.

Im nächsten Abschnitt wird auf die Grundlagen von Neuronalen Netzen eingegangen. Da diese zur Dimensionsreduktion (*Einbettung*) mittels *Multi-Layer Perceptrons* (MLP) und zur Vorhersage von Events in Sequenzen mittels rückgekoppelter Netze (RNN) genutzt wurden.

2.3 Künstliche Neuronale Netze (KNN)

Ein Künstliches Neuronales Netz (KNN) ist ein Modell des Maschinellen Lernens (ML), welches als universelle Funktionsapproximation angewandt werden kann.

Sei $\mathcal{D} = (X, Y)$ ein Datensatz mit $N_{\mathcal{D}}$ Instanzen (*Samples*), wobei zugleich $x \in \mathbb{R}^n = X$ (Definitionsmenge) und $y \in \mathbb{R}^m = Y$ (Zielmenge) einer Funktion f sind. Dann wird durch ein Neuronales Netz versucht, die Funktion f zu approximieren, welche den Zusammenhang zwischen $f(x) = y$ abbildet. Gesucht ist eine Funktion g, die sich f beliebig genau annähert [15, 28]:

$$min\, \mathcal{L} = \underset{\Theta}{argmin}\, E(g(\Theta, x), f(x)) < \epsilon \;\; \forall\, x \in X \wedge \epsilon > 0 \tag{2.2}$$

Bei der Optimierung eines neuronalen Netzes werden die Parameter Θ so angepasst, dass die Gleichung 2.2 erfüllt wird. Das Abbruchkriterium ϵ wird frei gewählt. E ist eine Zielfunktion, wie in Abschnitt 2.3.2 dargestellt. An die Funktionen f und g sind weitere Bedingungen geknüpft, die durch [15, 28] im *universal approximation theorem* erläutert werden. Grundsätzlich entspricht die Form von g der, wie in Gleichung 2.5 dargestellt. Dabei umfasst $\Theta = \{\theta | \theta \in W \vee b\}$ sowohl die Gewichte W, als auch den Bias b, wenn dieser vorhanden ist.
Handelt es sich bei der Zielmenge Y um eine Wahrscheinlichkeitsverteilung von Klassen (*Labels*) wird von einer *Klassifikation* gesprochen, anderenfalls ist Y das *Bild* von f. Im zweiten Fall wird von der *Regression* gesprochen. Bei der Klassifikation wird die Wahrscheinlichkeit der Zugehörigkeit der Eingabedaten x zu einer Klasse bestimmt, wenn f eine Wahrscheinlichkeitsfunktion ist. Bei der Regression wird je Instanz $x \in X$ aus der Definitionsmenge ein Vektor $y \in R^m$ (Bild) der Funktion f approximiert.
Ein KNN orientiert sich stark abstrahiert an der Funktionsweise von biologischen Neuronalen Netzen. Ein Neuronales Netz besteht aus L Schichten, die nacheinander verschaltet sind. Jede Schicht a^l, mit $l \in 1..L$ weist in der Regel vier Teile auf (a, W, b, σ) auf, die in einer differenzierbaren Funktion miteinander komponiert werden.
Ein Gewicht $w^l_{jk} \in \mathbb{R}$ stellt eine Verbindung von einem Neuron j zu einem Neuron k einer vorherigen Schicht $l - 1$ dar. Es kann ein Signal verstärken, abschwächen oder negieren. Die *Eingabe* einer Schicht l sei als Vektor a mit K Komponenten gegeben.

Wenn für jedes Neuron j ein Gewicht zu jeder Komponente k der *Eingabe* a vorliegt, spricht man von einem *Fully-Connected-Layer* (kurz *Dense-Layer*), andernfalls liegt ein *Sparse-Layer* vor. Ist $l < L$ spricht man von einem *Hidden-Layer* innerhalb des Netzes. Die Anzahl der Neuronen einer Schicht l sei durch J gegeben. Dann können die Gewichte einer Schicht als Matrix der Form $W_{J\times K}$ (*Gewichtsmatrix*) dargestellt werden, wobei K der Anzahl der Verbindungen zur vorherigen Schicht $l-1$ entspricht. Für die Eingabeschicht a^1 gilt $K := |x|$, für $x \in X$.
Im zweiten Schritt, wird für jedes Neuron j die *gewichtete Eingabe* als Skalar zusammengefasst. Diese Übertragungsfunktion ist die Summe der *gewichteten Eingabe*.
Der dritte Schritt wird durch den Bias b vorgegeben, der angibt, wie aktiv jedes Neuron einer Schicht ist. Ein höherer Bias führt zu einer starken *Aktivierung* und verstärkt somit die Eingabe. Ein neutraler Bias bedeutet, dass die Aktivierung eines Neurons lediglich von der Eingabe a und der Gewichtung w abhängt. Ein negativer Bias wirkt inhibitorisch und lässt lediglich sehr starke Signale hindurch.

$$z_j^l = \sum_{k\in K} w_{jk}^l a_k^{l-1} + b_j^l \tag{2.3}$$

Als letzter Schritt wird eine nichtlineare Aktivierungsfunktion σ auf die *gewichtete Eingabe* z_j^l eines Neurons angewandt, um die Annäherung nichtlinearer Zusammenhänge der Eingabedaten X zu ermöglichen. Die Aktivierung a_j^l eines Neurons j aus der Schicht l sei definiert als:

$$a_j^l = \sigma^l(z_j^l) \tag{2.4}$$

Die *gewichtete Eingabe* für die Aktivierungsfunktion σ kann effizient als Matrixmultiplikation zwischen den Gewichten W^l und der Eingabe a^{l-1} plus möglichem Bias b^l berechnet werden. Dies führt zur rekursiven Definition 2.5 einer Schicht a^l einer Schicht l:

$$a^l = \sigma^l(z^l) = \sigma^l(W^l a^{l-1} + b^l) \tag{2.5}$$

2.3.1 Architektur

Die Topologie eines Netzes hat starken Einfluss für die Anwendbarkeit eines Neuronalen Netzes auf ein gegebenes Optimierungsproblem. Es wurde gezeigt, dass ein Netz mit nur einem *Hidden-Layer*, tiefen Netzen gegenüber äquivalent ist. Nach dem *universal approximation theorem*) [15, 28] sind sie dazu in Lage jede Funktion anzunähern[2]. In der Praxis hat sich jedoch herausgestellt, dass tiefe Netzwerke (*Deep-Nets*) auf Grund ihrer Architektur einfacher Lernen [6]. In den letzten Jahren konnten mittels Deep-Learning in verschiedenen Disziplinen starke Erfolge verbucht werden. *Deep-Nets* werden z.B. im Bereich des der Bildverarbeitung oder beim NLP eingesetzt. Die Architektur eines Netzes spielt dabei eine entscheidende Rolle. Bestimme Architekturen eignen sich mehr als andere für gegebene Problemstellungen. Das Auffinden einer geeigneten Architektur eines Netzes ist nicht trivial.

Für diese Arbeit sind insbesondere zwei Architekturen relevant, die im Folgenden vorgestellt werden.

[2] Unter http://neuralnetworksanddeeplearning.com/chap4.html wird interaktiv erklärt, wie KNNs lernen.

Autoencoder (AE)

Ein *Autoencoder* definiert sich dadurch, dass er beim unüberwachten Lernen (*unsupervised*) mit Neuronalen Netzen genutzt wird. Ein unüberwachtes Optimierungproblem liegt dann vor, wenn die Zielmenge Y_u eines Datensatzes $\mathcal{D}_u$ der Verteilung der ursprünglichen Daten X_u entspringt. Somit gilt, dass $Y_u \subseteq X_u$ und $\mathcal{D}_u = (X_u, Y_u)$, wobei der Index u für *unsupervised* steht. Es können daher Daten verarbeitet werden, die nicht annotiert sind. Y_u kann im einfachsten Fall gleich X_u gesetzt werden. Oft wird bei sequentiellen Datensätzen Y_u um eine Instanz verschoben ($y_u^n = x_u^{n-1} | x^n \in X_u$), so dass dem Netz vereinfacht wird, Beziehungen (*Co-Occurrence*) zwischen Instanzen zu erkennen.

Ein Autoencoder g_{ae} zeichnet sich dadurch aus, dass er dem *Encoder-Decoder* Modell folgt. Er besteht aus zwei Teilen: Dem Encoder e und Decoder d, wie in Gleichung 2.6 zu sehen:

$$g_{ae}(x) = e \circ d = d(e(x)) \tag{2.6}$$

Formal versucht ein Autoencoder den Rekonstruktionsfehler E_r zu minimieren, wie in Gleichung 2.7 definiert ist.

$$min\, \mathcal{L} = min\, E_r(g_{ae}(x_u^n), y_u^n) \;\; \forall x_u^n \in X_u, y_u^n \in Y_u \tag{2.7}$$

E_r ist oft die mittlere quadratische Abweichung (siehe Abschnitt 2.3.2) oder die Kreuzentropie (siehe Abschnitt 2.3.2). Ein Autoencoder versucht daher die Identitätsfunktion zwischen der Definitions- und Zielmenge der Daten zu lernen.
Die Ein- und Ausgabeschicht des Autoencoders haben die selbe Dimensionalität m_u, um entweder eine Regression oder Klassifikation über den Zielbereich Y_u des Datensatzes durchzuführen. Zudem weist ein Autoencoder meist eine symmetrische Topologie auf. Der Encoder bettet die Eingabedaten in einen latenten Raum l_u ein. Der Decoder fungiert als *Generator*, um eine Instanz aus einer latenten Einbettung wieder herzustellen. Dabei komprimiert der Encoder in der Regel die Daten, so dass die eingebettete Dimension $l_u < m_u$ ist. Abbildung 2.1 zeigt zwei stilisierte Autoencoder Architekturen.
Die Idee des *Encoder-Decoder* Modells ist, dass ein Autoencoder mit einer endlichen Menge an Neuronen im latenten Raum dazu gezwungen ist, die auszeichnenden Merkmale der Daten zu kodieren, um diese anschließend durch den Decoder möglichst fehlerfrei wieder herstellen zu können. Autoencoder können als Dimensionsreduktions Technik, wie eine Principal Component Analysis (PCA) eingesetzt werden. Einer PCA gegenüber kann ein Autoencoder wegen der Aktivierungsfunktion (siehe Abschnitt 2.3.2) auch nicht-lineare Zusammenhänge erkennen. Ein Autoencoder kann als allgemeines Dimensionsreduktionswerkzeug genutzt werden, wenn der latente Raum weniger Dimensionen aufweist, als die Eingabe - *undercomplete*. Andernfalls spricht man von einem *overcomplete* Autoencoder, wobei das Verhalten der *Hidden-Units* oft durch Bedingungen eingeschränkt wird, um sinnvolle Repräsentationen zu lernen (z.B. *k-Sparse Autoencoders* (KSAE) [48] oder *Stacked Denoising Autoencoders* (SdA / DAE) [79]).

Autoencoder werden bereits erfolgreich in der Bildverarbeitung eingesetzt, wie *Variational Autoencoders* (VAE) [33], *k-Sparse Autoencoders* (KSAE) [48], Trainingsmethoden für *Convolutional Autoencoders* [47], *Adversarial Autoencoder* (AAE) [49]. Durch [83] konnte mittels *Deep Embedded Clustering* (DEC) gezeigt werden, dass der Schritt der Einbettung und des Clustering in der Lernphase kombiniert werden

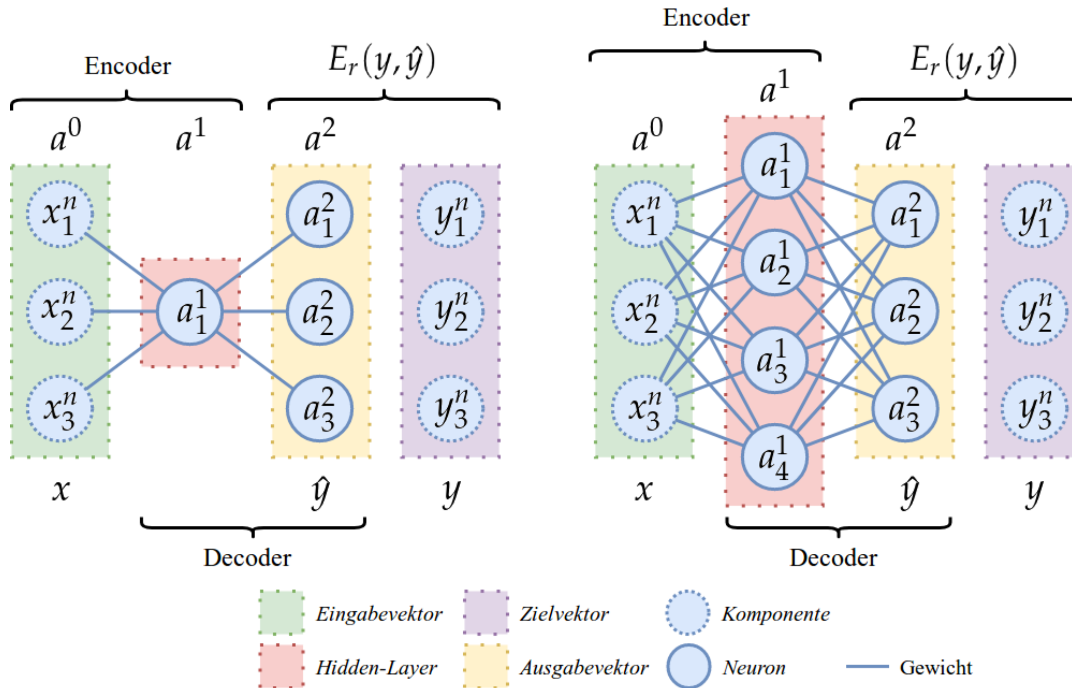


ABBILDUNG 2.1: Zwei Autoencoder - links (*undercomplete*), rechts (*overcomplete*). Der linke Autoencoder weist nur wenige *hidden-units* auf, wodurch der latente Raum ($a^1 \in \mathbb{R}^1$) niedrigdimensional ist. Der rechte Autoencoder weist hingegen einen hoch-dimensionalen latenten Raum ($a^1 \in \mathbb{R}^4$) auf. Beide Autoencoder sind flache (*shallow*) Netze, da sie jeweils lediglich ein *Hidden-Layer* aufweisen. Der *Encoder* ist jeweils ($a^1 \circ a^0$), der *Decoder* ($a^2 \circ a^1$). E_r ist der Rekonstruktionsfehler zwischen y (*ground truth*) und der Vorhersage durch den Autoencoder $\hat{y}$ (*prediction*).

kann.

Verschiedene Arbeiten haben gezeigt, dass Vektorrepräsentationen (Einbettung) dazu geeignet sind, sowohl syntaktische als auch semantische Regelmäßigkeiten von Worten, Sätzen oder ganzen Dokumenten abzubilden. Lange Zeit, wurden vor allem klassische Verfahren, wie *Latent Semantic Analysis* (LSA) [17] oder stochastische Verfahren, wie *Latent Dirichlet Allocation* (LDA) [44] eingesetzt. Zuletzt wurden vermehrt Autoencoder zur Generierung von Wortrepräsentationen genutzt, die im nächsten Absatz vorgestellt werden.

Neben tiefen Netzen, konnte ebenfalls mittels flacher Netze (*shallow nets*), gezeigt werden, dass diese dazu in der Lage sind semantische kohärente Text-Einbettungen zu generieren. *Word2Vec* [57] oder *fastText* [11, 31]. Mit *GloVe* [65] wurden globale Matrix-Faktorisierung und lokale Kontextfenster kombiniert. Mittels KATE [13] konnte gezeigt werden, dass kompetitive Schichten dazu in der Lage sind, mit einer höheren Genauigkeit (*accuracy*) bei der Klassifikation von Text abschneidet als folgender Verfahren, die in dem Artikel benannt werden: *Latent Dirichlet allocation* (LDA) [16], *Deep belief network* (DBN) [45], DocNADE [36], NVDM [55], Word2Vec [57], Doc2Vec [38], AE (einfache Autoencoder), *denoising autoencoder* (DAE) [79], *Contracting Autoencoders* (CAE) [69], *Variational Autoencoders* (VAE) [33], *k-Sparse*

Autoencoders (KSAE) [48]. In [81] wurden mehrere Einbettungsmethoden gegeneinander gestellt und festgestellt, dass einfache Durchschnittsbildende Modelle kompetitiv mit hochspezialisierten Modellen, wie LSTMs sind.

Neben t-SNE [46], dass entwickelt wurde um Datensätze hoher Dimensionen in zwei oder drei Dimensionen zu visualisieren, wird durch UMAP [52] ein Werkzeug zur allgemeinen Dimensionsreduktion zur Verfügung gestellt. UMAP übertrifft t-SNE in mehreren Hinsichten, wie dem Bewahren von globalen Strukturen und der Laufzeit.

Im nächsten Abschnitt wird eine Architektur vorgestellt, die dazu in der Lage ist, sequentielle Daten zu verarbeiten und Langzeitabhängigkeiten zwischen Instanzen zu lernen, was mittels einfachen *Feed-Forward*-Netzen (*Multi-Layer-Perceptron* - MLP) nicht möglich ist.

Long Short-Term Memory (LSTM)

Ein *Long Short-Term Memory* (LSTM) ist eine Erweiterung von Rekurrenten Neuronalen Netzen (RNN). Ein RNN ist ein rückgekoppeltes *Layer*, das Verbindungen zu sich selbst (t) oder vorangehenden Schichten ($t-1$) erlaubt. Es kann daher Sequenzen variabler Länge auf Zeichen- oder Wortebene verarbeiten, wobei jedes Zeichen oder zuvor eingebetettes Wort nacheinander als *Samples* präsentiert werden. LSTMs wurden erstmals 1997 durch [27] beschrieben, um fundamentale Nachteile von RNNs beim lernen von Langzeitabhängigkeiten zu lösen. In seiner Struktur ist es ein 5-Tupel (W, U, b, h, c). Wobei in einem LSTM-Layer vier Schaltungen (*gates*) genutzt werden, um sowohl die Eingabe x_t, die als c_t modifizert das Netz verlässt, als auch den Zustand von h zu manipulieren.

Ein LSTM verarbeitet eine Sequenz von Eingabe- und Zieldaten $(x_1, y_1), ..., (x_m, y_m)$. Das Verhalten eines LSTM-Layers ist definiert sich, wie folgt (angelehnt an [62]):

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{2.8}$$

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{2.9}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{2.10}$$

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{2.11}$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \tag{2.12}$$

$$h_t = o_t \circ \tanh(c_t) \tag{2.13}$$

Wobei folgender Definitionsbereich angenommen wird:

$x_t \in \mathbb{R}^d$: Eingabevektor der Daten

$h_{t-1} \in \mathbb{R}^l$: Eingabevektor des vorherigen *hidden-state* oder h_0 als Anfangswert.

$i_t, f_t, o_t, \tilde{c}_t \in \mathbb{R}^l$: Aktivierungsvektoren: *input gate*, *forget gate*, *output gate* und implizites *state gate*

$c_t, h_t \in \mathbb{R}^l$: Ausgabevektor: interner *cell state*, *hidden-state*

$W_g \in \mathbb{R}^{l \times d}$, $U_g \in \mathbb{R}^{l \times h}$ and $b_g \in \mathbb{R}^l$: Gewichtsmatrizen und Biase, wobei $g \in \{i, f, o, c\}$. Diese können zusammengefasst werden zu: $W \in \mathbb{R}^{4l \times d}$, $U \in \mathbb{R}^{4l \times d}$ und $b \in \mathbb{R}^{4l}$.

$\circ, +$: komponentenweises Produkt (*Hadamard-Produkt*) oder Addition

Der *Feed-Forward* Schritt für ein LSTM ergibt sich aus Gleichungen 2.8 bis 2.13:

$$(h_t, c_t) = LSTM(W_{t-1}, U_{t-1}, b_{t_1}, h_{t-1}, c_{t-1}, x_t) \tag{2.14}$$

LSTMs haben sich in letzten Jahren, im Bereich der *Sequence-to-Sequence* Modellierung für Übersetzungen aller Art bewährt [32]. Aktuell wird diese Forschung von Übersetzungsproblemen durch Modelle untersucht, die Aufmerksamkeit (*attention*) als Konzept nutzen und dabei die Leistung von LSTMs übertreffen können [78].

2.3.2 Lernen

Wenn ein KNN lernt, wird in der Regel Einfluss auf die frei wählbaren Parameter W, b, σ (siehe Gleichung 2.5) eines Netzes genommen. Das Netz wird solange optimiert, bis es eine gewünschte Güte an Ergebnissen produziert. Dabei kommen Gradientenabstiegsverfahren zum Einsatz, welche die optimalen Parameter des Modells für eine gegebene Zielfunktion bestimmen sollen.

Aktivierungsfunktion

Im Kontext der Sprachverarbeitung werden üblicherweise folgende nichtlineare Funktionen als Aktivierungsfunktionen σ verwendet, wie in Abbildung 2.2 zu sehen ist.

Eine lineare Aktivierungsfunktion führt dazu, dass die Komposition der Schichten zu einer Einzigen zusammengefasst werden kann. Ein solches Netz ist lediglich dazu in der Lage, die Eingabedaten durch eine Hyperebene[3] voneinander zu trennen. Um nichtlineare Zusammenhänge in Daten erfassen zu können werden daher die verschiedenen nichtlinearen Aktivierungsfunktionen eingesetzt. Ob eine Aktivierungsfunktion für eine bestimmte Architektur des Netzwerks und der zu lernenden Muster in den Daten geeignet ist, variiert stark und kann oftmals lediglich durch Ausprobieren herausgefunden werden. In tiefen Netzwerken werden häufig *Rectified Linear Units* (ReLU) angewendet, da diese das Problem des *Vanishing-Gradient* bei der Fehlerrückführung (siehe Abschnitt 2.3.2) weitgehend vermeiden können [23].

Zielfunktion

Um den Unterschied zwischen einer erwarteten Grundgesamtheit P (*ground truth*) und einer Vorhersage Q (*prediction*) zu bestimmen, benötigt man ein aussagekräftiges Maß (meist eine echte Metrik).

[3]Das Verhalten eines tiefen *Feed-Forward* Netzes kann interaktiv unter http://playground.tensorflow.org/ exploriert werden. Hier kann man z.B. anschaulich nachvollziehen, wie die Architektur des Netzwerkes hinfällig ist, wenn die Identität als lineare Aktivierungsfunktion eingesetzt wird.

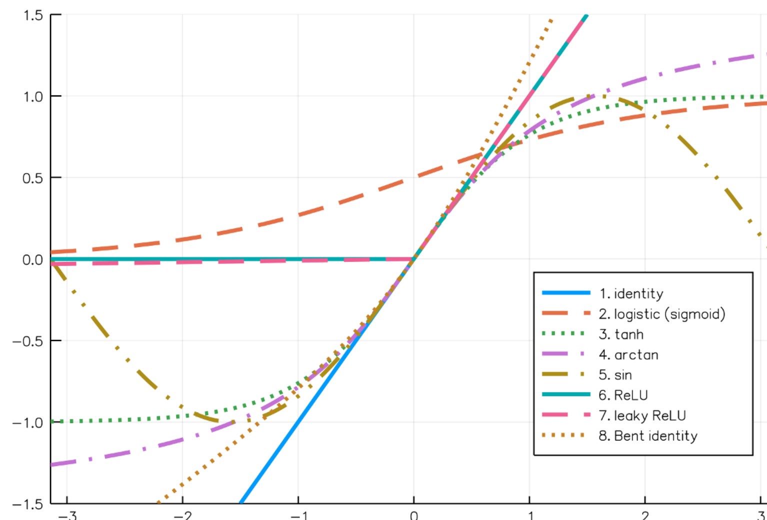


ABBILDUNG 2.2: Verschiedene nichtlineare Aktivierungsfunktionen, mit Ausnahme der Identität (1). Die Logistische Aktivierungsfunktion (2), stellt wegen ihres Definitionsbereichs eine weitere Ausnahme dar. Sie wird häufig zur Normalisierung genutzt, um explodierende Gewichte (W) zu vermeiden oder in der Ausgabeschicht eines KNNs verwendet. Die anderen Aktivierungsfunktionen (3-8) bieten unterschiedliche Eigenschaften bei der Fehlerrückführung (siehe: 2.3.2) und ihrer asymptotischen Laufzeit.

Eine *Zielfunktion* bzw. *Fehlerfunktion* (im Englischen: *loss function*, *error function*, *distance function*, *objective function*) drückt die Distanz von zwei gegebenen Verteilungen (meist als Vektoren repräsentiert) zueinander als skalaren Wert aus, der als *Fehler* (*error*, *loss*) bezeichnet wird. Dadurch wird die Vergleichbarkeit zwischen der Grundwahrheit und der Vorhersage eines Modells hergestellt, so dass überprüft werden kann, ob ein gegebenes Optimierungsproblem bereits hinreichend konvergiert ist.
Grundsätzlich können die Metriken unterschiedliche asymptotische Laufzeiten aufweisen. Daher ist es nicht nur von Relevanz eine aussagekräftige Metrik zu wählen, sondern gleichzeitig eine günstige, um Rechenzeit einzusparen. Diese beiden Anforderungen können sich diametral gegenüberstehen und müssen gegeneinander abgewogen werden. Dadurch, dass eine Zielfunktion für jede Iteration der Optimierung genutzt wird, nimmt sie einen erheblichen Anteil der gesamten Rechenzeit neben der Fehlerrückführung (siehe 2.3.2) bei künstlichen Neuronalen Netzen oder anderen numerischen Optimierungsproblemen ein.
In den folgenden Abschnitten werden verschiedene Zielfunktionen vorgestellt, die für das NLP besonders relevant sind:

Mittlere quadratische Abweichung

Die *mittlere quadratische Abweichung* (Eng.: *mean squared error* - MSE) gibt die Streuung um einen Punkt $\tilde{P}$ (Erwartungswert) wider. Der MSE ist die am häufigsten eingesetzte Fehlerfunktion für Feed-Forward Netze.
Für gegebene Daten $X \in \mathbb{R}^k$ von der Wahrscheinlichkeitsfunktionen P (Grundwahrheit) und Q (Vorhersage) ist der MSE, wie folgt in Gleichung 2.15 definiert, wobei $\tilde{P}(X)$ den beschreibenden Punkt (den Mittelwert) der Wahrscheinlichkeitsverteilung

von P darstellt und $n = |X|$:

$$D_{MSE}(P\|Q) = MSE(P, Q) = \frac{1}{n}\sum_{x\in X}(Q(x) - \tilde{P}(X))^2 \tag{2.15}$$

Wird statt dem Mittelwert der Median genutzt, spricht man vom der mittleren absoluten Abweichung vom Median (*mean deviation from the median* - MD). Der MSE wird auch häufig mit *L2 loss* bezeichnet.

Entropie
Die *Entropie* (2.16) gibt den durchschnittlichen Informationsgehalt I eines Ereignisses $A \in \Omega$ aus einer Wahrscheinlichkeitsverteilung P an. Sie ist ein Maß für Unordnung. Eine hohe Entropie weist auf eine hohe Unordnung der Verteilung P hin. Eine niedrige Entropie drückt aus, dass Ereignisse Ω aus der Verteilung P vorhersagbar sind, da sie im Durchschnitt nur wenig Information enthalten, weil sie wenig variieren.

$$H(P) = \sum_{x\in X} P(x)\, I(P(x)) = -\sum_{x\in X} P(x)\log_2 P(x) \tag{2.16}$$

Kullbach-Leibler-Divergenz (Information Gain)
Die *Kullbach-Leibler-Divergenz* (kurz: KL-Divergenz) im Englischen mit *Information Gain* bezeichnet ist ein Unähnlichkeitsmaß. Sie gibt den Unterschied zweier Verteilungen wider. Die Funktion ist nicht symmetrisch, daher dürfen gegebene diskrete Wahrscheinlichkeitsfunktionen P (Grundwahrheit) und Q (Vorhersage) bzw. konkrete Wahrscheinlichkeitsverteilungen nicht miteinander vertauscht werden. In der Informationstheorie ist sie ein Maß für die Effizienz einer Kodierung auf Basis von Q, wenn sie P folgt [35]:

$$D_{KL}(P\|Q) = KL(P, Q) = \sum_{x\in X} P(x)\cdot\log\frac{P(x)}{Q(x)} \tag{2.17}$$

Transinformation (*Mutual Information*)
Die Transinformation kann mittels der KL-Divergenz ausgedrückt werden. Sie beschreibt das Verhältnis der bivariaten Verteilung $p(x, y)$ im Verhältnis zu ihren einzelnen diskreten Randverteilungen $p(x)$ und $p(y)$.

$$I(X, Y) = D_{KL}(p(x, y)\|p(x)p(y)) = \sum_{x\in X}\sum_{y\in Y} p(x, y)\cdot\log_2\left(\frac{p(x, y)}{p(x)p(y)}\right) \tag{2.18}$$

Kreuzentropie
Die *Kreuzentropie* (auf Englisch: *cross entropy* oder *log loss*) ist eine Fehlerfunktion, die sowohl im Kontext des überwachten, als auch unüberwachten Lernens häufig verwendet wird. Sie gibt die durchschnittliche Länge einer Nachricht Q in Bezug auf eine Grundgesamtheit P aller Nachrichten an [5, S. 141].
Die Kreuzentropie (Gleichung 2.19) ist die Erweiterung der *Entropie* um die *Kullbach-Leibler-Divergenz* (Gleichung 2.17):

$$D_{CE}(P\|Q) = H(P) + D_{KL}(P\|Q) = -\sum_{x\in X} P(x)\log_2 Q(x) \tag{2.19}$$

Wenn $Q = P$, dann gibt sie lediglich die Entropie H wieder, die optimale Angleichung von Q an P, da dann $D_{KL}(P\|Q) = 0$ gilt.

Fehlerrückführung (Backpropagation)

Sei $E(w)$ eine Fehlerfunktion, wie im vorherigen Abschnitt 2.3.2 beschrieben und die Parameter des Modells mit w gegeben. Die Parameter w sind als n-dimensionaler Vektor gegeben und können neben den Gewichten auch den Bias enthalten, der dann als $w_j^l 0$ bezeichnet wird. Wenn ein Modell von weiteren Parametern abhängig ist, werden diese dem Vektor hinzugefügt. Dann ist $\nabla E(w)$ der Gradient der Fehlerfunktion in Bezug auf die Parameter w, der in Richtung des Anstiegs des Fehlers zeigt. Der gesamte Fehler ergibt sich aus der Betrachtung aller *Samples* der Trainingsdaten:

$$E(w) = \sum_{n=1}^{N_{\mathcal{D}}} E_n(w) \tag{2.20}$$

Ebenso ergibt sich der Gradient im Bezug auf gegebene Parameter aus der Summe der Gradienten für jedes *Sample*. Nach [9, S. 239]:

$$\nabla E(w) = \frac{\partial E}{\partial w_{jk}} = \sum_{n=1}^{N_{\mathcal{D}}} \frac{\partial E_n}{\partial w_{jk}} \tag{2.21}$$

Der sequentielle bzw. stochastische Gradientenabstieg (*stochastic gradient decent* - SGD) ist eine *online* Methode des Gradientenabstiegs . Sei τ der Index der durchlaufenen Iterationen und η die Lernrate. Dann ergibt sich die rekursive Definition des Gradientenabstiegs, wie folgt [9, S. 240]:

$$w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E_n(w^{(\tau)}) \tag{2.22}$$

Die Parameter werden in negativer Richtung des Gradienten geändert, um die gegebene Fehlerfunktion minimieren zu können. Es kann beispielsweise nicht nur ein *Sample* pro Iteration einfließen, sondern auch $2..N_{\mathcal{D}} - 1$ *Samples*, die gleichzeitig im *mini-batch*-Modus trainiert werden. Wird der Fehler in Bezug auf alle *Samples* $N_{\mathcal{D}}$ berechnet, beschreibt dies den Gradientenabstieg im *batch*-Modus mit nur einer Iteration über den gesamten Datensatz. Das *mini-batch* Verfahren stellt daher einen Kompromiss zwischen *online* und *batch* Verfahren dar, wobei es gilt eine geeignete *batch*-Größe im Bezug auf das gegebene Optimierungsproblem zu wählen.
Der Gradientenabstieg wurde bereits in verschiedene Optimierungsverfahren umgesetzt. Ziel des Optimierungsverfahrens ist es sich auf der Oberfläche der Fehlerfunktion, die durch den Definitionsbereichs der Parameter w aufgespannt wird, effizient und effektiv in Richtung des globalen Minimums zu bewegen. Dabei ist z.B. ADAM ein effektives Optimierungsverfahren, welches AdaGrad oder einfachem SGD gegenüber vorzuziehen ist[34].

Was alle Verfahren gemein haben ist, dass sie den Gradienten der Fehlerfunktion als rekursive Funktion zur Verfügung gestellt bekommen, der *Backpropagation*, wie in Abbildung 2.3 dargestellt.
Der Fehler in Bezug auf die Parameter w und gegebenem *Sample* besteht aus zwei Komponenten: dem Fehler δ_j (*back-propagation*) für Schicht j und dem Ergebnis aus der a_i (*forward-propation*) der Vorgängerschicht $i := j - 1$ (nach [9, S. 242]):

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial zj} \frac{\partial z_j}{\partial w_{ji}} = \delta_j a_i \tag{2.23}$$

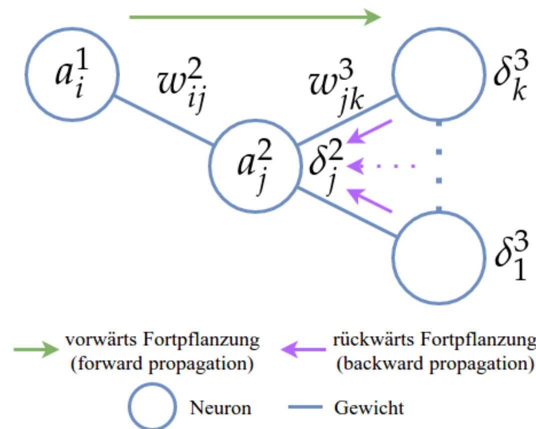


ABBILDUNG 2.3: Visualisierung der Fehlerrückführung (*Backpropagation*) in einem KNN für ein Neuron j. Angelehnt an [9, S. 244].

Für die Ausgabeschicht gilt der Rekursionsanfang aus der Schätzung (Vorhersage) durch das Modell $\hat{y}$ ($Q(x)$) für gegebene *Samples* im Bezug auf die Grundgesamtheit y ($P(x)$), mit $y \in \mathcal{D}$ (*Labels*):

$$\delta_k = \hat{y}_k - y_k = Q(x) - P(x) \tag{2.24}$$

Die Formel für *Backpropagation* nach [9, S. 244] für alle *Hidden-Layer* ist als Rekursionsschirtt, wie folgt anzuwenden:

$$\delta_j = \sigma^{'}(z_j) \sum_{k \in K} w_{jk} \delta_k \tag{2.25}$$

Automatisches Differenzieren (AD)

Beim Automatischen Differenzieren (AD) handelt es sich um ein Verfahren, dass auf eine Funktion die als prozedurales Programm gegeben ist angewendet werden kann. Es kann der exakte Gradient an einer gegebenen Stelle (Anfangswert) der Funktion bestimmt werden. Das AD beruht darauf, dass für jeden Teilausdruck (*Expression*) der Prozedur der Gradient berechnet wird, die für definierte Primitiven einfach zu berechnen sind. Bei der AD wird jedem Wert sein Gradient zugeordnet so, dass sich ein *augmented* Tupel aus dem Wert x und seinem Gradienten $\triangle$ ergibt:

$$x \rightarrow (x, \triangle) \tag{2.26}$$

Praktisch kann dies mittels Sammeln der Rechenschritte auf einem Band (*tape*) umgesetzt werden. Beim Durchlaufen der *forward-propagation* werden dabei alle Ausdrücke gesammelt und auf das Band geschrieben. Bei einem Band handelt es sich um einen *Computational Graph* [56]. Um den Gradienten der Ausdrücke zu berechnen, kann das Band auf unterschiedliche Arten durchlaufen werden. Vorwärts im *forward-mode*, rückwärts im *reverse-mode* oder aus einer Kombination beider Verfahren.
Die *Backpropagation* ist ein spezieller Fall des Automatischen Differenzierens im *reverse-mode* [7]. AD ist somit ein geeignetes Verfahren, um den Gradienten einer Fehlerfunktion effizient und flexibel zu berechnen. In den Schichten eines Netzes oder der Fehlerfunktion selbst können dadurch komplexe Prozeduren definiert werden, die sowohl *branches* durch Wenn-Dann-Abfragen, Schleifen oder sogar Bäume

nutzen. Ohne AD müssten die Ableitungen analytisch mittels symbolischer oder numerisch mittels numerischer Differenzierung beigebracht werden, was jeweils sehr aufwändig und fehleranfällig ist.
Das Verfahren der *reverse-mode* AD ist bereits seit den 1970er bekannt [24], findet jedoch erst seit wenigen Jahren vermehrt Anwendung in verschiedenen ML Frameworks, wie z.B. Tensorflow [2], Keras mit entsprechendem *Backend* [14], PyTorch [63], Knet [85], Flux [30] u.v.m.. AD ermöglicht eine erhebliche Reduktion des Kosten-Nutzen Aufwands bei der Implementierung ungewöhnlicher Neuronaler Netze und erlaubt neue Topologien und Zielfunktionen zu testen.

2.4 Clustering

Bei der *Clusteranalyse* (*Clustering*) handelt es sich um eine Zuordnung von Instanzen eines Datensatzes zu einer oder mehreren Gruppen (*Clustern*). Ziel ist eine Gruppierung zu finden, sodass der *Inter-Cluster* Abstand minimal ist und der *Intra-Cluster* Abstand maximal ist.
Formal Grundlage für die Clusteranalyse bieten folgende Begrifflichkeiten (angelehnt an [86, S. 426]):

Definition 2.8 (Cluster)**.** Ein Cluster $C_i = \{x_1, ..., x_j\}$ ist eine ungeordnete Menge mit j Mitgliedern (*members*), wobei $x \in X$ einem Datensatz mit n Instanzen entspringt und $1 \leq j \leq n$ ist.

Definition 2.9 (Clustering)**.** Ein *Clustering* $\mathcal{C}$ besteht aus k *Clustern*, so dass jeder Punkt (*Instanz*, *Objekt*) eines Datensatzes X einem Cluster zugeordnet ist.

$$\mathcal{C} = \{C_1, ..., C_k\} \;\; \forall x \in X \tag{2.27}$$

Definition 2.10 (Partitionierung)**.** Eine *Partitionierung* $\mathcal{T}$ besteht aus r *Partitionen*. Eine *Partitionierung* stellt die Grundgesamtheit der Daten dar. Eine *Partition* ist äquivalent zu einem *Cluster*.

$$\mathcal{T} = \{T_1, ..., T_r\} \;\; \forall x \in X \tag{2.28}$$

Wenn die Partitionierung der Daten bekannt ist, wird $k := r$ gesetzt. Ist r unbekannt, muss k geschätzt werden. Die korrekte Schätzung von k ist nicht trivial [8].

Des Weiteren wird zwischen weichen (*soft*) und harten (*hard*) Verfahren unterschieden. Harte Clustering-Verfahren unterteilen die Daten in disjunkte Cluster. Weiche Verfahren hingegen lassen zu, dass eine Instanz mit einer Zugehörigkeit (*membership degree*) mehreren Clustern angehört. Eine weitere Variante ist, eine Instanz mehreren Clustern vollständig zuzuordnen.

Clustering Methoden lassen sich in verschiedene Ansätze unterteilen. Im folgenden benenne ich verschiedene repräsentative Clusteringverfahren. Die Liste erhebt keinen Anspruch auf Vollständigkeit, es werden klassische Verfahren benannt:

- Clusteranalyse mittels Repräsentanten (*Representative-based Clustering*)
  - k-Means nach [43] ist ein Clustering Verfahren, dass einen Datensatz in Voronoi-Zellen unterteilt. Der Mittelpunkt einer Voronoi-Zelle dient als Repräsentant eines Clusters.
- Dichtebasierte Clusteranalyse (*Density-based Clustering*)

  - DBSCAN ist ein dichtebasiertes Clusteringverfahren, das dazu entwickelt wurde Cluster beliebiger Form aufzufinden [21].

- Unterraum Clustering (*Subspace Clustering*)
  - CLIQUE

- Probabilistisches Clustering (*Fuzzy-Clustering*)
  - EM-Algorithmus
  - Fuzzy-C-Means

- Hierarchische Clusteranalyse (*Hierarchical clustering*)
  - Brown Clustering: Ist ein hierarchisches Clustering-Verfahren mit Binärbäumen und Wort-Bigrammen als Blätter des Binärbaums. Es wird die Transinformation (siehe Gleichung 2.18) zwischen der Auftrittswahrscheinlichkeiten eines Bigramms zu den einzelnen Auftrittswahrscheinlichkeiten der Worte im Bigramm maximiert [12, 18].

- Spektralclustering (*Spectralclustering*): Spektrale Clustering-Verfahren nutzen die Eigenwerte (Spektrum) der Distanzmatrix eines Datensatzes um die Daten in einen niedrigdimensionalen Raum zu transformieren. Der eingebettete Raum kann anschließend mit Clustering-Verfahren, wie etwa k-Means in ein Clustering überführt werden [10]. Die Einträge der Distanzmatrix werden als Gewichte eines Nachbarschaftsgraphen interpretiert. Kanten von zu weit voneinander entfernten Punkten werden entfernt, was zur Dimensionsreduktion führt.

Dabei sind die Methoden nicht immer klar voneinander abzutrennen, da oftmals hybride Verfahren angewandt werden, um die Nachteile eines bestimmten Verfahrens auszugleichen.

2.4.1 Clustering Auswertung (Clustering Validation)

Dadurch, dass Clusteringverfahren sehr unterschiedlich sein können und meist parametrisiert sind, können objektive Metriken dazu beitragen, die Güte oder die Qualität eines Clustering zu bestimmen und dadurch vergleichbar zu machen. Es werden drei Arten von Validitätsmetriken unterschieden [86, S. 425]:

- **Interne**: Bestimmen die Güte anhand der vorliegenden Daten mittels Distanzmetriken. Sie können für unüberwachte Lernverfahren genutzt werden.

- **Externe**: Bestimmen die Güte anhand vorgegebenen Wissens ($\mathcal{T}$). Sie sind der Klasse der überwachten Verfahren zuzuordnen.

- **Vergleichende**: Bestimmen die Ähnlichkeit zweier *Clustering* $\mathcal{C}$ und $\mathcal{C}'$. Dienen meistens zur Kontrolle eines Algorithmus unter Anwendung verschiedener Parameter.

Diese Arbeit nutzt sowohl interne, als auch externe Verfahren, um die Ergebnisse der Verschiedenen Clustering-Verfahren miteinander und auf verschiedenen Datensätzen zu vergleichen. Voraussetzung ist, dass gegebene Daten überhaupt Gruppen von Instanzen enthalten, die sich ähneln. Ein Clustering Validity Index (CVI) ist ein

Indikator für die Güte eines Clustering. Oftmals ist es eine *Metrik*, die der Symmetrie und Dreiecksungleichung genügt, ansonsten wird von einem *Ähnlichkeitsmaß* bzw. *Unähnlichkeitsmaß* gesprochen. CVIs unterscheiden sich stark in ihrer Komplexität und neigen dazu, bestimmte Clustering-Methoden zu bevorzugen, wenn sich die Metrik zu Erstellung eines Clustering und eines CVIs ähneln. Wie [59] zu sehen ist, weist jede Metrik gegenüber bestimmten Clustering-Methoden oder Datensätzen Vor- und Nachteile auf.

Um in dieser Arbeit die Güte eines Clustering objektiv zu bewerten, verwende ich eine Kombination verschiedener Verfahren. Dies dient dazu die Nachteile eines einzelnen Verfahrens durch ein anderes ausgleichen zu können. Zum einen wird das intuitive Verfahren des *BetaCV* [86, S. 441] [68] genutzt, dass die innere Kohärenz eines Clusters in Kontrast zur Entfernung aller anderen Cluster setzt. Zum anderen wird der *S_Dbw* [26] [42] Index genutzt, um die Verteilung (*Scattering*), als auch die Dichte (*Density*) in eines Clustering zu bestimmen. Auf die beiden Maße des *BetaCV* und *S_Dbw* wird im Abschnitt 2.4.1 näher eingegangen, da diese in der Arbeit selbst implementiert wurden. Für die anderen Metriken und Maße wurden bestehende Implementierungen genutzt.

Für Paarweise-Metriken $(x_i, x_j) \in \mathcal{D}$ werden vier Fälle unterschieden (für $y \in \mathcal{T}$ und $\hat{y} \in \mathcal{C}$), nach [86, S. 434]:

- *True Positives* (TP): Punkte, die sowohl zum Cluster C_i, als zur Partition P_i gehören: $TP = \{(x_i, x_j) \mid y_i = y_j \, und \, \hat{y}_i = \hat{y}_j\}$
- *False Negatives* (FN): Punkte, die nicht zum selben Cluster C_i, jedoch zur selben zur Partition P_i gehören: $FN = \{(x_i, x_j) \mid y_i = y_j \, und \, \hat{y}_i \neq \hat{y}_j\}$
- *False Positives* (FP): Punkte, die zum selben Cluster C_i, jedoch nicht zur selben zur Partition P_i gehören: $FP = \{(x_i, x_j) \mid y_i \neq y_j \, und \, \hat{y}_i = \hat{y}_j\}$
- *True Negatives* (TN): Punkte, die weder zum Cluster C_i, noch zur Partition P_i gehören: $TN = \{(x_i, x_j) \mid y_i \neq y_j \, und \, \hat{y}_i \neq \hat{y}_j\}$

Es gilt die Identität für alle Paare (x_i, x_j) der Punkte von $\mathcal{C}$ und $\mathcal{T}$:

$$N = TP + FN + FP + TN \tag{2.29}$$

Im Folgenden werden verschiedene CVIs aufgelistet, die in der Arbeit genutzt wurden. Diese werden nach den drei genannten Kategorien (*Intern*, *Extern*, *Vergleichend*) unterschieden:

Externe Maße:

- Die Homogenität (*homogeneity*) gibt an, ob Cluster lediglich Mitglieder (*member*) einer Klasse enthalten.
- Die Vollständigkeit (*completeness*) gibt an, ob Instanzen einer Klasse möglichst einem Cluster zugeordnet sind.
- *Variation of Information* (VI), „misst den Anteil der Zu- oder Abnahme an Information beim Wechseln von einem Clustering $\mathcal{C}$ zu einem Clustering $\mathcal{C}'$“ [54].

- *Accuracy* gibt den Anteil der korrekt klassifizierten Klassen (Cluster) wieder [72]. Diese Metrik kann lediglich genutzt werden, wenn ein Clustering $\mathcal{C}$ und eine Partition $\mathcal{T}$ die gleiche Anzahl an Clustern aufweisen.
- *F1-Score* gibt einen gewichteten Durchschnitt von *Precision* und *Recall* an [73].
- *V-Measure* gibt den harmonischen Durchschnitt zwischen Homogenität und Vollständigkeit an [74].
- *adjusted Mutual information* (AMI) [80] (Maß, keine Metrik)
- *Rand-Index*: $Rand = \frac{TP+TN}{N}$ [86, S. 435] [66].
  Neben dem *Rand*-Index gibt es weitere abgeleitete Metriken und Maße, welche diesen auf verschiedene Arten korrigieren:
  - *adjusted Rand-Index* (AR): Der bereinigte Rand-Index ist eine Korrektur für eine kleine Anzahl an Punkten N. Er kann Werte zwischen 0 und 1 (für identische Clustering) annehmen [53, S. 174].
  - *Hubert's Index*: Ist eine Korrektur des Rand-Indexes mit probabilistischer Interpretation. Der Index ist auf den Wertebereich ± 1 beschränkt. Statt Tupeln werden Tripel von Instanzen miteinander verglichen [29].
  - *Mirkin's Index*: „Die Mirkin Metrik ist eine bereinigte Form des Rand-Indexes“ [53]. Für gleiche Clustering $\mathcal{C} = \mathcal{C}'$ ist der Index 0, sonst positiv.

Für die Homogenität und Vollständigkeit gelten Werte nahe 1 als positiv zu bewerten.
Relative Maße

- *Silhouette* gibt an, wie gut ein Punkt innerhalb eines Clusters liegt, im Gegensatz zu anderen Clustern [40].

Interne Maße:
BetaCV

Bei *BetaCV* (siehe Gleichung 2.35 nach [86, S. 441]) handelt es sich um ein Ähnlichkeitsmaß, welches mittels einer Distanzmetrik ($w(x_i, x_j)$) berechnet werden kann. Die Distanz zwischen zwei Punkten (x_i, x_j) wird auch als Gewicht (*weight*) bezeichnet. Dadurch, dass das *BetaCV*-Maß auf der Distanzmatrix (*distance matrix*, *proximity matrix*, siehe Gleichung 2.30) aufbaut, handelt es sich um ein internes Maß zur Bewertung der Güte eines Clustering.

$$W(S,R) = \sum_{x_i \in S} \sum_{x_j \in R} w_{ij} \tag{2.30}$$

Das Maß misst das bereinigte Verhältnis der *Intra-Cluster* (W_{in}) Abstände im Verhältnis zu den *Inter*-Cluster (W_{out}) Abständen aller Cluster.

Intra-Cluster Abstand

$$W_{in} = \frac{1}{2} \sum_{i=1}^{k} W(C_i, C_i) \tag{2.31}$$

Inter-Cluster Abstand

$$W_{out} = \frac{1}{2} \sum_{i=1}^{k} W(C_i, \bar{C}_i) = \sum_{i=1}^{k-1} \sum_{j>1} W(C_i, C_j) \tag{2.32}$$

Anzahl der Gewichte (N_{in}, N_{out}) für den *Intra-* und *Inter-Cluster* Abstand:

$$N_{in} = \sum_{i=1}^{k} \binom{n_i}{2} = \frac{1}{2} \sum_{i=1}^{k} n_i(n_i - 1) \tag{2.33}$$

$$N_{out} = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} n_i \cdot n_j = \frac{1}{2} \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} n_i \cdot n_j \tag{2.34}$$

Mittels Gleichungen 2.31, 2.32, 2.33, 2.34 kann das *BetaCV*-Maß (siehe Gleichung 2.35) definiert werden:

$$BetaCV = \frac{W_{in}/N_{in}}{W_{out}/N_{out}} = \frac{N_{out}}{N_{in}} \cdot \frac{W_{in}}{W_{out}} \tag{2.35}$$

S_Dbw

Bei dem *S_Dbw*-Index (echte Metrik) handelt es sich um ein Ähnlichkeitsmaß, dass zum einen die Varianz einzelner Cluster ins Verhältnis zur Varianz der Punkte des gesamten Datensatzes stellt (*Scattering* - siehe Gleichung 2.38) und zum anderen *die inter-cluster density* (ID - siehe Gleichung 2.40) berücksichtigt. Nach [42] ist der Varianz-Vektor, wie folgt definiert:

$$\nu^{\{k\}} = (Var(\nu_1^{\{k\}}), ..., Var(\nu_p^{\{k\}})), wobei \;\; p \in D \tag{2.36}$$

$$\sigma = \frac{1}{K} \sqrt{\sum_{k=1}^{K} \|\nu^{\{k\}}\|} \tag{2.37}$$

Das *Scattering* (Gleichung 2.38) setzt die Varianz eines Clusters ins Verhältnis zur Varianz des gesamten Datensatzes. Es ist, wie folgt bei [42] definiert:

$$Scattering = \frac{1}{n_c} \sum_i \frac{\|\sigma(C_i)\|}{\|\sigma(D)\|} = \frac{1}{n_c} \sum_i \frac{(C_i^T \cdot C_i)^{1/2}}{(D_i^T \cdot D_i)^{1/2}} \tag{2.38}$$

Nach [26] wird die *density* durch die Anzahl von Punkten bestimmt, die innerhalb eines Radius r in einer p-dimensionalen Sphäre liegen. Der Radius r wird durch die Standardabweichung der Punkte des Datensatzes vorgegeben.

$$density(u) = |\, \{\, x \mid \; \|x - u\| \leq r \;\; \forall x \in X \} \,| \tag{2.39}$$

Die *inter-cluster density* (ID) ist nach [26], wie folgt definiert:

$$Dens_bw = \frac{1}{n_c(n_c - 1)} \sum_{i=1}^{n_c} \sum_{\substack{j=1 \\ i \neq j}} \frac{density(u_{ij})}{max\{density(v_i), density(v_j)\}} \tag{2.40}$$

$$S_Dbw = Scattering + Dens_bw \tag{2.41}$$

Da die Berechnung des *Dens_bw* Teilindexes sehr rechenintensiv ist und bei nicht überlappenden Clustern nicht zum tragen kommt, kann dieser vernachlässigt werden. Daher wird eine Kombination des *BetaCV* Indexes und des *Scatterings* verwendet, um die Güte der verschiedenen Ergebnisse zu evaluieren.

2.5 Mustererkennung in Sequenzen

Beim *Frequent Episoden Mining* (FEM), welches zuerst formal durch [50, 51] beschrieben wurde, handelt es sich um eine Abwandlung des *Sequential Pattern Mining* (SPM). Im Gegensatz zum SPM wird beim FEM eine Sequenz auf Muster untersucht und nicht etwa mehrere Sequenzen auf das Auftreten gleicher Muster.
Im folgenden wird auf das *Episode Mining* mittels Bedingungen (*constraints*) eingegangen. Es werden Begriffe definiert, die für das Verständnis des *Episode Mining* unter Berücksichtigung von minimalen Auftritten und weiteren Bedingungen nötig sind. Zunächst folge eine Abgrenzung zum *Episode Mining* mittels fester Fenstergröße.

2.5.1 Frequent Episode Mining (FEM)

Die ersten Algorithmen zum Frequent Episode Mining (FEM) wurden durch [51] mit WINEPI und MINEPI erarbeitet. Dabei handelt es sich um zwei Algorithmen, die den Suchraum mittels Beschränkungen (*constraints*) verringern (*pruning*). Ziel ist es häufige Episoden zu finden. Wobei in dieser Arbeit die Häufigkeit mittels Support (siehe Definitiont: 2.17) gemeint ist und mit Episoden, serielle Episoden (siehe Definition: 2.14) gesucht werden.

WINEPI nutzt die Angabe einer festen Fenstergröße (*window width* - kurz: win) um den Suchraum zu beschränken. Eine Episode gilt als gefunden, wenn sie mindestens einmal innerhalb eines Fensters (w - siehe Definition 2.13) auftritt - Mehrfachauftritte werden nicht berücksichtigt - und diese häufig (*frequent*) ist. Eine Episode ist beim WINEPI-Ansatz häufig, wenn ihre Frequenz (*frequency* - kurz: fr) größer als die durch den Anwender geforderte Mindestfrequenz (min_fr) ist: $fr \geq min_fr$. Gleichung 2.42 nach [51] beschriebt die Frequenz unter Berücksichtigung der Fenstergröße win:

$$fr(\alpha, S, win) = \frac{|\{w \in \mathcal{W}(S, win) | \alpha \, tritt \, in \, w \, auf\}|}{|\mathcal{W}(S, win)|} \tag{2.42}$$

Wobei $\mathcal{W}(S, win)$ die Menge alle Fenster w mit der Größe (*width*) win ist. Die Menge aller häufigen Episoden hinsichtlich der Konditionen win und min_fr wird, wie folgt bezeichnet: $\mathcal{F}_{win}(S, win, min_fr)$. Dieser Ansatz hat jedoch einen Nachteil: „Der Algorithmus konnte die doppelte Zählung von Auftritten einer Episode nicht vermeiden“ [67]. Daher wurde die MINEPI-Strategie entwickelt.

MINEPI nutzt minimale Auftritte (siehe Definition 2.16) einer Episode, um lediglich solche zu finden, die häufig (*frequent*) nach dem *Minimum Support* (min_sup) Kriterium sind: $support(\alpha) \geq min_sup$. Der MINEPI Ansatz unterscheidet sich zu WINEPI, wie folgt: „(1) Es gibt keine feste Fenstergröße und (2) ein Fenster darf mehrere minimale Auftritte enthalten“ [51, S. 273]. Statt dem Konzept der Frequenz folgt die MINEPI-Strategie dem Support (siehe Definition 2.17). In dieser Arbeit werden Algorithmen erweitert, die der MINEPI-Strategie insofern folgen, dass sie die minimalen Auftritte einer Episode berücksichtigen.

Definition 2.11 (Event)**.** Sei $\mathcal{E}$ die Menge aller Eventtypen. Ein Event $E = (e, t)$ ist ein Paar, wobei $e \in \mathcal{E}$ der Eventtyp ist und $t \in \mathbb{N}^+$ den Zeitpunkt des Auftritts (*occurrence*) eines Events darstellt [51, 67].

Definition 2.12 (Event Sequenz)**.** Eine *Event Sequenz* ist ein Tupel $ES = (s, I)$, wobei $I = [T_s, T_e]$ ein Intervall ganzer Zahlen ist. $s = \langle (e_1, t_1), (e_2, t_2), ..., (e_n, t_n) \rangle$ ist eine geordnete Menge von Events E, wobei jedem Eventtyp e ein Zeitpunkt $t \in \mathbb{N}^+$ zugeordnet ist, so dass $T_s \leq t_i < t_j \leq t_n < T_e$ [51, 67]. Die *Länge* $|ES|$ einer Event Sequenz ist gleich der Anzahl enthaltener Events in s. Des Weiteren wird T_s als Anfang (*starting time*) und T_e als Ende (*ending time*) bezeichnet [51]. Mit der Notation $ES[i]$ ist ein das Event $E_i = (e_i, t_i)$ an der Stelle i in s gemeint.

Beispiel 2.7. Sei folgende Zuordnung eine Event Sequenz $ES = (s, [1, 5])$:

e_i	A	B	C	E
t_i	1	2	3	4

Dann ist $s = \langle (A, 1), (B, 2), (C, 3), (E, 4) \rangle$ und die *Länge* |ES| = 4.

Beispiel 2.8. Sei folgende Zuordnung eine Event Sequenz $ES = (s, [1, 5])$:

e_i	A	B		E
t_i	1	2	3	4

Dann ist $s = \langle (A, 1), (B, 2), (E, 4) \rangle$ und die *Länge* |ES| = 3.

Definition 2.13 (Fenster - *Window*)**.** Ein Fenster w beschreibt eine Subsequenz ES' im Intervall $I' = [T'_s, T'_e]$ einer *Event Sequenz* $ES = (s, [T_s, T_e])$. Wobei $T'_s < T_e$ und $T'_e > T_s$. $w(ES, I') = \langle\, ES[i] \mid \forall\, i \in [T'_s, ..., T'_e - 1] \,\rangle$. Die Zeitspanne $T'_e - T'_s$ beschreibt die *Größe* (*width*) eines Fensters [51].

Beispiel 2.9. Das Fenster w(ES, [4, 8]) für die *Event Sequenz* ES aus Tabelle 2.1 ergibt: $\langle (E, 4), (D, 5), (G, 6), (E, 7) \rangle$ und hat die *Größe* 4. Die Angabe von $T'_e = 8$ ist exklusive. Das Event $E_8 = (A, 8)$ ist nicht im Fenster enthalten.

Nach [51] wird in zwischen *parallelen* und *seriellen* Episoden unterschieden. Parallele Episoden liegen dann vor, wenn ihre Events gleichzeitig auftreten und keiner Ordnung folgen. In dieser Arbeit werden serielle Episoden betrachtet, die einer totalen Ordnung folgen. Wird in dieser Arbeit lediglich von einer Episode gesprochen ist damit eine serielle Episode gemeint. Serielle Episoden sind, wie folgt definiert:

Definition 2.14 (Serielle Episode - *Serial Episode*)**.** Eine serielle Episode α ist eine nicht leere Menge von Eventtypen $e \in \mathcal{E}$ totaler Ordnung: $\alpha = \langle e_1, e_2, ..., e_n \rangle$, wobei e_i vor e_j auftritt, für alle $1 \leq i < j \leq n$ [82]. Die Länge einer Episode ergibt sich aus der Anzahl enthaltener Events.

Beispiel 2.10. Die Notationen für eine Episode $\alpha = \langle A, B, C \rangle$ oder α_{ABC} meinen das Selbe.

Der Unterschied zwischen einem *Fenster* und dem *Auftritt* einer *seriellen Episode* wird gut durch [58] ausgedrückt: Ein *Fenster* ist eine konsekutive (*consecutive*) Subsequenz einer *Event Sequenz ES*, wogegen der *Auftritt* einer *seriellen Episode* eine nicht konsekutive (*non-consecutive*) Subsequenz von *SE* ist. Oft werden die Begriffe Episode und *Auftritt* einer Episode synonym genutzt, meinen jedoch verschiedenes. Eine Episode ist generisch und steht für sich, wogegen der *Auftritt* einer Episode sich auf eine *Event Sequenz* bezieht, also konkrete Subsequenz in einem bestimmten Intervall ist. Im folgenden wird ein *Auftritt* einer Episode formal definiert:

Definition 2.15 (Auftritt - *Occurrence*)**.** Der *Auftritt* einer Episode α in einer *Event Sequenz ES* ist ein Intervall $I = [T_s, T_e]$: $occ(\alpha) = [T_s, T_e]$. Eine Episode α tritt auf, wenn (1) die Folge der Events $E_i, ..., E_n$ einer Episode α im Fenster $w(ES, [T_s, T_e + 1])$ auftritt und (2) das erste Event $E_1 = (e_1, t_1)$ an der Stelle T_s und das letzte Event $E_n = (e_n, t_n)$ an der Stelle T_e auftritt. Die *Menge der Auftritte* (*oSet*) einer Episode ergibt sich aus allen Auftritten von α in *ES* [67].

Beispiel 2.11. Für die Episoden $\langle A, B\rangle$, $\langle E, E\rangle$, $\langle A, B, E\rangle$ gelten folgende Auftritte für die *Event Sequenz ES* aus Tabelle 2.1:

$$oSet(\alpha_{AB}) = \{[1,2], [1,9], [1,15], [8,9], [8,15], [14,15]\}$$
$$oSet(\alpha_{EE}) = \{[4,7], [4,12], [4,13], [7,12], [7,13], [12,13]\}$$
$$oSet(\alpha_{ABE}) = \{[1,4], [1,7], [1,12], [1,13], [8,12], [8,13]\}$$

Die Aussage „Der Minimale Auftritt einer Episode α ist ein Zeitintervall $[t_s, t_e]$, in dem die Episode auftritt und in keinem echten Subintervall von $[t_s, t_e]$ auftritt“ [67] lässt sich formal wie folgt beschreiben:

Definition 2.16 (Minimaler Auftritt - *Minimal Occurrence*)**.** Eine Episode α tritt in einer *Event Sequenz ES* minimal auf, wenn (1) α in einem Intervall $[T_s, T_e]$ auftritt und (2) für gegebenes Intervall $[T_s, T_e]$ kein Subintervall $[T'_s, T'_e]$ gebildet werden kann, so dass α im Fenster $w(S, [T'_s, T'_e + 1])$ auftritt, wobei für ein Subintervall gilt, dass $T_s \leq T'_s$ und $T'_e \leq T_e$ ist. Der minimale Auftritt einer Episode ist ein Intervall, das die Bedingungen erfüllt und schreibt sich: $mo(\alpha) = [T_s, T_e]$. Die *Menge der minimalen Auftritte* (*moSet*) einer Episode ergibt sich aus allen minimalen Auftritten einer Episode, so dass sich Ende und Anfang von zwei minimalen Auftritten einer Episode überschneiden dürfen: $T^i_e \leq T^j_s$ und $1 \leq i < j \leq |moSet|$, wobei $|moSet|$ der Anzahl aller minimalen Auftritte von α in *ES* entspricht [82].

Beispiel 2.12. Minimale Auftritte der Episoden $\langle A, B\rangle$, $\langle E, E\rangle$, $\langle A, B, E\rangle$ in *ES* aus Tabelle 2.1:

$$moSet(\alpha_{AB}) = \{[1,2], [8,9], [14,15]\}$$
$$moSet(\alpha_{EE}) = \{[4,7], [7,12], [12,13]\}$$
$$moSet(\alpha_{ABE}) = \{[1,4], [8,12]\}$$

Definition 2.17 (Support)**.** Der *Support* einer Episode ist gleich der Anzahl der minimalen Auftritte einer Episode α: $support(\alpha) = |moSet|$.

Der Support bringt eine Eigenschaft mit sich, die in der Literatur als *Downward Closure Property* benannt wird. Um diese Eigenschaft aufzuzeigen, muss zunächst definiert werden, was eine häufige Episode (*frequent episode*) ist. Folgende Definitionen (2.18, 2.19, 2.20) sind durch [82] vorgegeben:

Definition 2.18 (*Frequent Episode*)**.** Eine Episode wird nur dann als *frequent* bezeichnet, wenn dessen Support nicht geringer als ein Schwellwert *min_sup* (*minimum Support*) ist. Andernfalls ist eine Episode als *infrequent* zu bezeichnen.

e_i	A	B	C	E	D	G	E	A	B	C	F	E	E	A	B	D
t_i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

TABELLE 2.1: Eine *Event Sequenz ES* mit 16 Events, gegebener Menge an Eventtypen $\mathcal{E} = \{A, B, C, D, E, F, G\}$ und geordneten Events $E_i = (e_i, t_i)$, wobei $i \in \mathbb{N}^+$, $e_i \in \mathcal{E}$ und $t_i \in \mathbb{N}^+$.

e	A	B	C	D	E	F	G
u_{ext}	1.0	1.0	0.5	1.0	1.5	1.0	1.0

TABELLE 2.2: Eine Zuordnung zwischen Eventtypen $e \in \mathcal{E}$ und externer Nützlichkeit u_{ext}.

Definition 2.19 (*Frequent Episode Mining*). Für eine gegebene Event Sequenz ES und einen Schwellwert min_sup (*minimum support*), besteht das Problem des *Frequent Episode Mining* darin alle Episoden zu finden, die keinen geringen Support als das min_sup Kriterium aufweisen.

Definition 2.20 (*Downward Closure Property*). (1) Für jede *frequent* Episode gilt, dass dessen Subepisoden ebenfalls *frequent* sind. (2) Für jede *infrequent* Episode gilt, dass dessen Superepisoden ebenfalls *infrequent* sind.

2.5.2 High Utility Episode Mining (HUEM)

Das High Utility Episode Mining (HUEM) unterscheidet sich durch das FEM dadurch, dass eine Bedingung eingeführt wird, die als Nützlichkeit (*Utility*) bezeichnet wird. Ein Problem des FEM ist, dass zwar häufige Muster gefunden werden, jedoch Muster mit potenziell hohem Informationsgehalt, die selten auftreten, nicht gefunden werden. Gerade seltene Muster mit einem hohen Informationsgehalt sind jedoch oft von hohem Interesse. Dieses Problem soll durch das *utility pattern mining* (UPM) angegangen werden und findet in unterschiedlichsten Bereichen Anwendung, die seltene Muster mit hohem Informationsgehalt suchen.
, wie in folgender Definition dargelegt:

Definition 2.21 (Nützlichkeit - *Utility*). Die Nützlichkeit einer Episode α ergibt sich aus der Summe der externen Nützlichkeiten multipliziert mit der internen Nützlichkeit jedes Events einer Episode: $u(\alpha) = \sum_{i=1}^{n} u_{int}(e) \times u_{ext}(e)$, wobei n die Länge der Episode ist und $e \in E$ der Menge der Eventtypen.

Die Zuordnung der externen Nützlichkeit wird wie in Tabelle 2.2 vorgegeben. Im Szenario einer „Warenkorbanalyse“ (*Market Basket Analysis*) entspricht die externe Nützlichkeit dem Profit (*profit*) eines Produktes. Die interne Nützlichkeit würde im selben Szenario der Anzahl (*quantity*) gekaufter Produkte entsprechen [67]. Der Fall interner Nützlichkeit von Events wird in dieser Arbeit nicht berücksichtigt, da jedes Event in einem Event-Log einmalig auftritt, $u_{int} := 1$.
Analog zur Definition des *Frequent Episode Mining* (2.19) ergibt sich für das *High Utility Episode Mining* (HUEM) folgende Definition:

Definition 2.22 (*High Utility Episode Mining*). Für eine gegebene Event Sequenz ES und einen Schwellwert $min_utility$ (*minimum utility*), besteht das Problem des *High Utility Episode Mining* darin alle Episoden zu finden, die keine geringe Nützlichkeit (*Utility*) als das $min_utility$ Kriterium aufweisen.

Im nächsten Abschnitt wird definiert, wie aus Sequenzen bzw. Episoden Regel abgeleitet werden können.

2.5.3 Episode Rule Mining

Das ableiten von Regeln aus Sequenzen oder Episoden erweitert die Assoziationsanalyse. Durch das bilden von Regeln können nicht nur Korrelationen, sondern Implikationen erschlossen werden. Dadurch können Ursachen eines Ereignisses bestimmt werden.

Eine Regel ist eine Zuordnung mit linker (*Prämisse*, *antecedent* - unabhängige Variable) und rechter (*Konsequenz*, *consequent* - abhängige Variable) Seite. Prämisse und Konsequenz sind aufeinander folgende disjunkte Subsepisoden einer Epsiode α. Die rechte Seite γ ist eine Episode, die nach der Episode β auftritt.

$$\alpha_{rule} = \beta \rightarrow \gamma \ \ , wobei \, \beta \preceq \gamma \tag{2.43}$$

Metriken für *Association Rules* nach [86, S. 220], die ebenso für das Rule Mining von *Episode Rules* zutreffen:

Support

$$support_{seq}(A) = P(A) \tag{2.44}$$

$$support_{rule}(A \rightarrow B) = support_{rule}(AB) = |AB| \tag{2.45}$$

$$relative_support_{rule}(A \rightarrow B) = P(A \cup B) = \frac{support_{rule}(AB)}{|D|} \tag{2.46}$$

Confidence

$$confidence_{rule}(A \rightarrow B) = P(B/A) = \frac{P(A \cup B)}{P(A)} \tag{2.47}$$

Die in diesem Kapitel genannten Definitionen finden im Kapitel der Methodik 3 in Form verschiedener aufeinander aufbauender Verfahren und Algorithmen Anwendung.

Kapitel 3

Methodik

In diesem Kapitel wird auf die konkreten Herausforderungen der Arbeit eingegangen. Diese werden zunächst als Anforderungen festgehalten und in eine Spezifikation überführt.
Danach wird auf die verschieden Techniken und Methoden eingegangen, die zum Einsatz kommen. In diesem Kapitel werden die Begriffe und Definitionen aus den Grundlagen herangezogen (siehe Kapitel 2).

3.1 Anforderungen und Spezifikation

3.1.1 Anforderungen

Im Rahmen der Arbeit wurden zu Beginn funktionale Anforderungen erhoben, um den Rahmen der Arbeit weiter einzuschränken. Diese dienten vielmehr zu Orientierung, um festzulegen, mit welchen Forschungsfragen sich die Arbeit auseinander setzen sollte, als diese vollständig abzudecken. Ziel der Arbeit ist nicht gewesen eine vollständige Anwendung zu schaffen, sondern ein Wege aufzuzeigen, wie in einer Log-Datei mittels verschiedener Verfahren Muster aufgedeckt werden können.

Die Anforderungen folgen dem Schema:

Als *PERSON* [muss/kann] ich *AKTION*, um *GRUND*.

Dadurch, dass für jede Anforderung eine Begründung angegeben wurde, musste sich alle Beteiligten mit der Sinnhaftigkeit und Relevanz einer geforderten Anforderung auseinandersetzen. Dies sollte als normativer Filter dienen, um zum einen die Qualität der Anforderungen zu erhöhen und zum anderen die Quantität einzuschränken. Die Liste der Anforderungen ist als Anhang 5.1 beigefügt.

3.1.2 Spezifikation

Die Anforderungen wurden in ein Diagramm des Informationsflusses (siehe Abbildung 3.1) überführt, der durch die Komponenten der Arbeit ermöglicht werden soll. Die Abbildung zeit, dass Log-Dateien in einen Event-Log überführt werden, bevor diese weiterverarbeitet werden. Dadurch ist gewährleistet, dass Log-Keys und Log-Attribute getrennt voneinander verarbeitet werden können. Dieser Ansatz ermöglicht numerische Log-Attribute (wie Ordinalzahlen, Intervalle oder Verhältniszahlen) von nominalen Attributen zu unterscheiden.

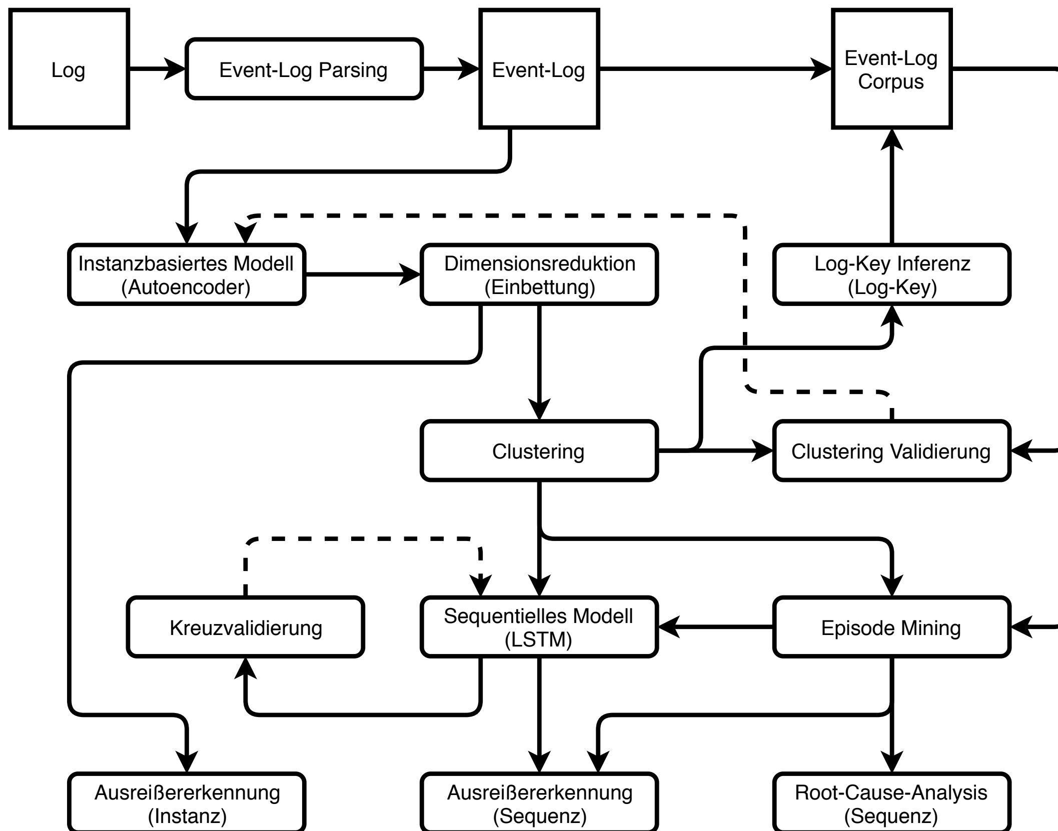


ABBILDUNG 3.1: Angestrebter Informationsfluss zwischen Komponenten, die als Bibliothek zur Verfügung gestellt werden.

3.2 Algorithmen und Datenstrukturen

In diesem Abschnitt werden die verschiedenen Algorithmen der Schritte des KDD Prozess' dargestellt, die für die Nachvollziehbarkeit der Arbeit notwendig sind.

Pseudocode

Zunächst stelle ich grundlegende Datenstrukturen der Arbeit dar. Die Arbeit wurde hauptsächlich in der Programmiersprache Julia implementiert, die eine ähnliche Syntax zu Matlab und Lisp aufweist. Die Syntax von Julia weist einige Eigenheiten auf, die hier teilweise im Pseudocode der Algorithmen übernommen wurden.

Symbole werden mit einem Doppelpunkt definiert (z.B. `:key` oder `:value`). Indexangaben für Intervalle sind inklusive. Die Syntax `1:5` expandiert zu `[1,2,3,4,5]`. Zudem gibt die Funktion *length* über ein Intervall die Anzahl der enthaltenen Punkte wieder (z.B. `length(1:5)` ist 5). Die Notation `Array[1:end]` meint alle Werte im Intervall vom ersten bis zum einschließlich letzten Element des Arrays. Es gilt, dass eine leere Menge wiedergegeben wird, wenn `end` kleiner als die linke Seite des Intervalls ist. Die Funktion *eachindex* gibt ein Intervall über jeden Index einer aufzählbaren Datenstruktur wieder. Die Funktion *enumerate* verhält sich, wie ein Generator in Python und gibt ein Tupel aus Index und den Wert an der Stelle des Indexes in einem Array wieder. Zum Beispiel ist

```
for (i, value)in enumerate(array)... end
```

eine *For-Each* Schleife über jeden Index i und Wert $value$. Dieselbe Syntax ist für Hashtabellen anwendbar, wobei der erste Wert des Tupels der Schlüssel ist und der Zweite der zugeordnete Wert $value$ zum Schlüssel:

```
for (key, value)in dict ... end
```

Werte einer Matrix `M` können mit einer kommaseparierten Syntax gelesen und modifiziert werden. Dabei gibt die Syntax `M[Zeile, Spalte]` das Element an der Stelle von $Zeile$ und $Spalte$ wieder. Des Weiteren können mittels `:` alle Elemente einer Zeile oder Spalte angesprochen werden. Für ein Array `A` gibt der Ausdruck `A[:]` das gesamte Array wieder. Für eine Matrix `M` mit m Zeilen und n Spalten gibt die Syntax $M[:]$ alle Elemente in einem Array der Länge $n * m$ zurück. `M[i]` gibt das Element an erster Stelle $i = m * n$ wieder. Mit der Syntax `M[Zeile,:]` oder `M[:,Spalte]` können alle Elemente einer Zeile oder Spalte angesprochen werden.
Neben der Syntax für den Umgang mit Datenstrukturen habe ich zudem die Julia Syntax für komponentenweise Operatoren- und Funktionsaufrufe übernommen. Die Syntax besteht darin, einen `.` vor einen Operator oder eine Funktion zu schreiben. Dies entspricht einer Abbildung von jedem Element eines Arrays oder einer Matrix auf die Zielmenge des Operators oder der Funktion. Zum Beispiel kann ein Hadamard-Produkt für zwei Matrizen $A \circ B$ dadurch, wie folgt geschrieben werden: `A .* B`.
Mit dem Schlagwort **`function`** sind pure Funktionen gemeint. Eine Prozedur (**`procedure`**) meint hingegen eine Methode, die Teile der Eingabe mutiert und in der Regel keinen Rückgabewert aufweist. Prozeduren wird in dieser Arbeit (und als Konvention in Julia) zusätzlich ein `!` im Namen nachgestellt (Bsp. `name!`).

3.2.1 Datenstrukturen

Zur Verarbeitung von Log-Dateien wurden die folgenden Datenstrukturen iterativ-inkrementell implementiert. Ein explizites Entity-Relationship-Modell oder Datenmodell wurde nicht erarbeitet. Die Datenstrukturen werden binär im BSON Format im Dateiverzeichnis persistiert.

Ein `Label` (*Label*) ist ein Container für einen beliebigen Token (String), der einem *LogAttr* zugeordnet werden kann, um diesen eine Bedeutung zuzuordnen:

```
1 struct Label
2     label: String
3 end
```

Ein *Label* kann z.B. ein *Log-Level* Token sein oder andere Konzepte, wie einen Zeitstempel dartsellen.

`LogKey` (*LogKey*):
Ein `LogKey` ist ein Array von Strings. Wobei die einzelnen Strings die Token eines *Log-Keys* nach der Tokenisierung (siehe: Algorithmus 3.2) sind.

`LogAttr` (*Log-Attribut*):
Ein `LogAttr` repräsentiert eine vom Parser aufgefundene Primitive und trägt die folgenden Attribute:

```
1 struct LogAttr
2     value: T
3     source: String
4     label: Label
5     occurrence: Interval
6 end
```

Für ein `LogAttr` gilt, dass *T* ein beliebiger Typ sein kann. `source` ist ein Ausschnitt aus der ursprünglichen Log-Zeile im Bereich des Auftritts (*occurrence*). Das `label` Attribut ist ein zugeordnetes *Label*. `occurrence`, wobei sich das Intervall auf den Auftritt in der ursprünglichen *Log-Zeile* bezieht.

`EventLog` (*Event-Log*):
Ein *Event-Log* ist als geordnete *Hashtable* (*DataStructures.OrderedDict*) implementiert. Dabei trägt ein *Event-Log* mindestens zwei Schlüssel: `:keys` und `:values`, um der Definition 2.7 für einen *Event-Log* gerecht zu werden. Wobei dem Schlüssel `:keys` die *Log-Keys* als Array von Strings zugeordnet sind. Dem Schlüssel `:values` sind die Log-Attribute als Array der `LogAttr` Datenstruktur zugeordnet (siehe `LogAttr` 3.2.1).

`EventCorpus` (*Event-Corpus*):
Ein *Event-Corpus* umfasst die Menge aller erfassten *Log-Keys* und weist diesen eine eindeutige ID zu. Ein `EventCorpus` ist als geordnete *Hashtable* (*DataStructures.OrderedDict*) implementiert, wobei der Schlüssel ein `LogKey` ist. Jedem Schlüssel ist ein eindeutiger Integerwert zugeordnet.

3.2.2 Event-Log Parser

Um Log-Dateien zu verarbeiten werden diese zunächst in einen Event-Log überführt. Algorithmus 3.1 beschreibt einen rekursiven Parser über die eine Log-Zeile einer Log-Datei.

```
1  function parse_line
2      Input:  line   : String,
3              labels: Array of triples of (label, regular expression, parse function),
4              slice : Intervall (start:stop)
5      Output: parsed: Array of tuples of (occurrence, value)
6
7      if length(line) = 0
8          return []
9      end
10
11     if slice.start = 0
12         slice := 1:length(line)
13     end
14
15     sub := substring(line, slice.start, slice.stop)
16
17     if length(labels) = 0
18         return [(slice, sub)]
19     end
20
21     parsed := []
22     (label, re, parse_value) = labels[1]
23     rest := labels[2:end]
24     matches := matchall(re, sub)
25
26     if length(matches) = 0
27         append(parsed, parse_line(line, rest, slice))
28     end
29
30     for i in eachindex(matches)
31         m := matches[i]
32
33         if i = 1
34             from := slice.start
35             to := slice.start + m.offset - 1
36             if length(from:to) > 0
37                 append(parsed, parse_line(line, rest, from:to))
38             end
```

```
39            end
40
41            from := slice.start + m.offset
42            to := slice.start + m.offset + m.endof - 1
43            push(parsed, (from:to, LogAttr(parse_value(m), m, label, from:to)))
44
45            if i < length(matches)
46                from := slice.start + m.offset + m.endof
47                to := slice.start + matches[i+1].offset -1
48                if length(from:to) > 0
49                    append(parsed, parse_line(line, rest, from:to))
50                end
51            else # i = length(matches)
52                from := slice.start + m.offset + m.endof
53                to := slice.stop
54                if length(from:to) > 0
55                    append(parsed, parse_line(line, rest, from:to))
56                end
57            end
58        end
59
60        return parsed
61    end
```

QUELLCODE 3.1: Der Event-Log Parser parst eine Zeile rekursiv, um darin Log-Attribute (*LogAttr*) zu finden. Der Parser beachtet dabei die Rangordnung der übergebenen *Label* (*labels*), um komplexe Attribute vor einfachen abzugleichen (*match*).

Die Funktion *substring* in Zeile 15 bildet eine Subsequenz (*Substring*) eines Strings, wobei die Angabe der Indizes inklusive ist. Jeder `SubString` trägt die Attribute `offset` und `endof`, die sich auf den Beginn und das Ende des Auftritts im Quellstring beziehen. Die Angaben sind ebenfalls inklusive. Die Funktion *matchall* in Zeile 24 gibt ein Array der Auftritte des gesuchten regulären Ausdrucks wieder, wobei es sich um ein Array von `SubString` handelt.

Der Parser funktioniert, wie folgt: Zunächst wird nach Auftritten des derzeitigen *Label* (Zeile 22) in einer *Log-Zeile* gesucht (siehe Zeile 24) - mittels zugeordnetem regulärem Ausdruck `re` (Zeile 22). Für jeden Auftritt wird sowohl auf der linken (Zeile 37), als der rechten Seite (Zeile 49 oder 55) nach weiteren Auftritten der übrigen *Label* (*rest* - Zeile 23) gesucht. Dies wird solange wiederholt, bis entweder die linke oder rechte Seite eines Auftritts oder die Menge der restlichen *Label* leer sind. Zeilen 26-28 stellen sicher, dass nach allen *Label* gesucht wird. Es wird weiter gesucht, auch wenn das derzeitige *Label* nicht auftritt. Das Ergebnis Array kann somit sowohl Instanzen von `SubString`, als auch `LogAttr` als Wert an zweiter Stelle im Tupel (*value*) enthalten. Der Algorithmus behält dabei die Reihenfolge des Auftretens der *Log-Attribute*, sowie der *Substrings* bei, indem gefundene *Log-Attribute* erst nach dem Hinzufügen der linken Seite (Zeile 37) dem Ergebnis Array *parsed* angehängt werden (Zeile 43).

Folgender Algorithmus wird nach dem Parsen von Log-Zeilen angewandt, um eine Log-Zeile zu tokenisieren. Dabei kann der Nutzer einen regulären Ausdruck (*splitter*) angeben, der vorgibt, an welchen Zeichen eine Zeile getrennt werden soll. Anders als oftmals in der NLP üblich, behalte ich die Trennzeichen bei, da diese syntaktische Information tragen, die zur Unterscheidung von Log-Zeilen beitragen kann. Es wird z.B. zwischen einem (`" "`) oder mehreren Leerzeichen (`"   "`) unterschieden.

```
1   function tokenize
2       Input: parsed: a parsed log-line,
3              splitter: regular expression, default r"\s+"
4       Output: a tuple of (log_key: Array of String, log_vals: Array of LogAttr)
```

```
5
6       log_key := []
7       log_vals := []
8       for (pos, token) in parsed
9           if token is a LogAttr
10              push(log_key, token.label.label)
11              push(log_vals, token)
12          else
13              append(log_key, NLP.split_and_keep_splitter(token, splitter))
14          end
15      end
16      (log_key, log_vals)
17  end
```

QUELLCODE 3.2: Um eine geparste Log-Zeile in einen Log-Key mit Log-Attributen zu überführen, werden *Substrings* durch die Angabe eines *splitters* in *Token* aufgeteilt.

Für die Ausreißeruntersuchung kann ein *Event-Log* um weitere Attribute neben den Log-Attributen erweitert werden. Nach dem Vorbild von [20] kann für jedes Event die Zeitdifferenz zum vorherigen Event ermittelt werden. Im Algorithmus 3.3 wird beschrieben, wie jedem Event ein Integer Wert zugeordnet wird. Dieser ergibt sich aus der Differenz zum letzten Auftreten eines Zeitstempels (*Datetime*).

```
1   function extract_time
2       Input: log_values: Array of LogAttr
3       Output: log_times: Array of Integer
4
5       log_times := fill(0, length(log_values))
6       last := nothing
7       for i in eachindex(log_values)
8           if length(log_values[i]) > 0
9               value := log_values[i][1].value
10              if value is a DateTime
11                  if last = nothing
12                      last := value
13                  end
14                  log_times[i] := value - last # time diff
15                  last = value
16              end
17          end
18      end
19      log_times
20  end
```

QUELLCODE 3.3: Der Algorithmus extrahiert die Zeitdifferenz von einem Event zum nächsten. Falls ein Event keinen Zeitstempel (*DateTime*) aufweist, wird davon ausgegegangen, dass die dieser zeitlich zum letzten Event mit einem Zeitstempel stattgefunden hat.

3.2.3 Modellierung Neuronaler Netze

Bevor Daten mit einem Neuronalen Netz verarbeitet werden können, müssen die Daten zur numerischen Verarbeitung aufbereitet werden. Die Idee von tiefen Netzen ist, dass diese dazu in der Lage sind, abstrakte Merkmale (*Features*) der Daten aufzugreifen. Wie im Abschnitt 2.3 beschrieben, baut ein Neuronales Netz ein internes Modell der Eingabedaten auf, indem die Gewichte des Netzes angepasst werden.
In dieser Arbeit wird versucht, ein interne Repräsentation von *Log-Keys* aufzubauen. Da dessen Quelle oft nicht zur Verfügung steht, müssen diese inferiert werden. Die Arbeit bewegt sich daher in einem *unüberwachtem* Lernszenario.
Wie im Abschnitt 2.2 dargestellt, bewegt sich die Repräsentation von Sprache zwischen zwei Extrema. Auf einer Seite können die Daten mit einer dünn-besetzten

Repräsentation kodiert werden. Auf der anderen Seite mit einer verteilten Repräsentation. Beide Ansätze haben Vor- und Nachteile, wobei ich mich in dieser Arbeit für folgende Repräsentation entschieden habe, wie im Algorithmus 3.4 dargestellt.

```
1  function normalize
2      Input:  document::NLP.Document, vocab::NLP.TermCount,
3              padding := 0.0, W := 1000
4      Output: N::Normalized Frequency Matrix with size LxW
5
6      max_count := maximum(values(vocab))
7      L = length(document)
8      N = fill(padding, (L,W))
9      for l in 1:L
10         for w in 1:W
11             if w <= length(document[l])
12                 word := document[l][w]
13                 N[l,w] := vocab[word] / max_count
14             else
15                 break
16             end
17         end
18     end
19     return N
20 end
```

QUELLCODE 3.4: Frequenznormalisierung mittels maximalem Auftreten. Die normalisierte Frequenzmatrix N ist eine rechtsseitig aufgefüllte Matrix mit Füllwert *padding*.

Die Funktion 3.4 nimmt eine verschachtelte Repräsentation eines Dokuments entgegen, ein *NLP.Document*. Dabei handelt es sich um eine Liste von Zeilen, die wiederum aus einer Liste von Worten besteht. Ein *NLP.TermCount* ist eine Abbildung repräsentiert als *Hashtable* von einem Wort (*word*) zu der Anzahl des Auftretens des Wortes im Dokument ($vocab[word]$). Die normalisierte Frequenzmatrix N ist eine rechtsseitig aufgefüllte Matrix mit Füllwert *padding*.

Die Modellierung der Neuronalen Netze erfolgte mit Hilfe durch das Framework *Flux* [30] für maschinelles Lernen in Julia. Das Framework ist erweiterbar und weist die Möglichkeit auf eigene Schichten (*Layer*) für Neuronale Netze zu definieren.
Ein Layer weist dabei zwei Teile auf. Zum einen ist es eine Datenstruktur mit Attributen, wie in Quellcode 3.5 dargestellt. Zum anderen weist ein *Layer* Verhalten auf, wie im Algorithmus 3.6 für KATE dargestellt.

```
1  struct KCompetetive
2    sigma: Function # activation function
3    W: Matrix       # weights
4    b: Array        # bias
5    k: Integer      # k winner neurons
6    alpha: Float    # boost
7    active: Boolean # training vs. test mode
8  end
```

QUELLCODE 3.5: Ein KATE Layer, wie nach [13] definiert.

Ein Kate Layer zeichnet sich durch drei Besonderheiten aus. Zum ersten kann durch das Attribut k festgelegt werden, auf wie viele Neuronen die Aktivierungen aller anderen Neuron verteilt wird. Zum zweiten gibt es einen Hyperparameter $alpha$, der durch [13] empirisch ermittelt wurde und auf 6.26 festgelegt ist. Zum dritten handelt es sich um eine Schicht, die sich während der Trainingsphase anders als in der Testphase verhält. Es ist somit ein *stateful Layer* und trägt daher das Attribut *active*, welches für die Trainingsphase auf *wahr* gesetzt ist, sonst *falsch*. Die drei anderen

Attribute *sigma*, *W* und *b* entsprechen σ, *W* und *b* aus Gleichung 2.5. Für ein KATE Layer ist die Funktion σ der *tanh*, wie Gleichung 3.1 nach [13] zu sehen ist.

$$z = tanh(Wx + b) \tag{3.1}$$

In der Testphase verhält sich ein KATE *Layer*, wie ein *dense Layer* aus Gleichung 2.5, wie man gut in Zeile 8 des Algorithmus' 3.6 sehen kann. Falls man sich jedoch in der Trainingsphase befindet, verhält es sich kompetitiv, wie in [13] dargelegt. Die *Energie* (nach Gleichung 3.2 [13]), einer Untermenge von Neuronen *H* wird auf die Hälfte ($k/2$) „Gewinner" Neuronen verteilt. Die *Energie* stellt das gesamte Aktivierungspotential der Neuronen in der Untermenge *H* dar.

$$E(H) =_{h_i \in H} |z_i| \tag{3.2}$$

In einem KATE Layer werden zunächst zwei Mengen von Neuronen gebildet. Zum einen die Neuronen mit positiver Aktivierung (*ps*) und die negativen (*ns*), wie in den Zeilen 18-24 im Algorithmus 3.6 zu sehen ist. Danach werden die beiden Mengen nach ihrem Potential geordnet (siehe Zeilen 25-28). Für die positiven Neuronen wird die Hälfte der Gewinner mit $\lceil k/2 \rceil$ bestimmt. Für die negativen Neuronen wird die Hälfte der Gewinner mit $\lfloor k/2 \rfloor$ bestimmt. In den Zeilen 34-45 und 47-58 ist definiert, wie die Energie der „Verlierer" Neuronen zu dem bestehende Aktivierungspotential unter Anwendung des Hyperparameters *alpha* hinzu addiert werden.

```
1  function k_competitive
2      Input: a: KCompetetive layer,
3             x: Array of Float
4      Output: z: Array of Float
5
6      σ, W, b, k, α = a.sigma, a.W, a.b, a.k, a.alpha
7
8      z := σ.(W*x .+ b)
9
10     # only apply K-Competetion in the training phase
11     if not a.active
12         return z
13     end
14
15     ps := [] # Array of tuples of Integer and Float
16     ns := [] # Array of tuples of Integer and Float
17
18     for (i, activation) = enumerate(z)
19         if activation >= 0
20             push(ps, (i, activation))
21         else
22             push(ns, (i, activation))
23         end
24     end
25     # sort in ascending order by the second value of the tuple
26     ps := sort(ps, by:=last)
27     # sort in descending order by the second value of the tuple
28     ns := sort(ns, by:=last, reverse:=true)
29     P := length(ps)
30     N := length(ns)
31
32     ẑ := z
33
34     p := P - ceil(k/2) # round up to Integer
35     if p > 0
36         E_pos := sum(map(second, ps[1:p]))
37         for i in p+1:P
38             # positive winners
39             ẑ[first(ps[i])] := ẑ[first(ps[i])] + (α * E_pos)
40         end
41         for i in 1:p
42             # positive losers
```

```
43              ẑ[first(ps[i])] := 0.0
44          end
45      end
46
47      n := N - floor(k/2) # round down to Integer
48      if n > 0
49          E_neg := sum(map(second, ns[1:n]))
50          for i in n+1:N
51              # negative winners
52              ẑ[first(ns[i])] := ẑ[first(ns[i])] + (α * E_neg)
53          end
54          for i in 1:n
55              # negative losers
56              ẑ[first(ns[i])] := 0.0
57          end
58      end
59
60      return ẑ
61  end
```

QUELLCODE 3.6: Ein KATE Layer, wie nach [13] definiert.

Die Funktionen *ceil* und $floor$ (Zeile 34 und 47) runden zu Integer, wobei für Reste $x \leq 0.5$ abgerundet wird. Die Funktionen $first$ und *last* geben jeweils das erste oder letzte Element einer geordneten Menge wieder. Für ein Tupel, wie in Zeile 26 gibt die Funktionen *last* das zweite Element des Tupels aus.

DeepKATE

In diesem Abschnitt gehe ich darauf ein, wie das von mir erstellte Modell DeepKATE zur allgemeinen Dimensionsreduktion genutzt werden kann. Die Idee von DeepKATE ist, die Vorteile der kompetitiven Aktivierung von Neuronen von KATE mit einem tiefen Netzwerk zu verknüpfen, um eine gruppierende Einbettung im latenten Raum zu erzeugen. Das Hinzufügen mehrerer Schichten soll ermöglichen, dass das Netzwerk einfacher abstrakte Muster in den Eingabedaten erkennen kann. Damit das Netzwerk die Daten nicht auswendig lernt (*overfitting*) wird die *dropout*-Strategie während des Trainings angewandt.
Der Quellcode 3.7 zeigt die DeepKATE-Architektur als Autoencoder. n gibt die Dimensionalität der Eingabedaten ($\mathbb{R}^n_{\mathcal{D}}$) an. l gibt die Dimensionen des latenten Raums ($\mathbb{R}^l_{latent}$) an. Der erste Parameter eines Flux-Layers gibt in der Regel die Eingabedimensionen, der zweite die Ausgabedimensionen an. Dritter Parameter für *Dense*- und *KCompetetive*-Layer ist die Aktivierungsfunktion. Um die Datenpunkte gleichmäßig im latenten Raum zu verteilen, wird die Ausgabeschicht des Encoders mit einem Sinus aktiviert (siehe Zeile 6). Die beiden *Dropout*-Layer (Zeile 4 und 9), sowie die die *KCompetetive*-Layer (Zeile 2 und 5) verhalten sich während des Trainings anders als in der Testphase.
Ein *Dropout*-Layer[1] setzt Eingabewerte zufällig (mit Wahrscheinlichkeit p) auf 0 oder skaliert andernfalls die Eingabewerte mittels $1/(1-p)$. Das führt dazu, dass das neuronale Netz dazu gezwungen ist selbst gleiche Instanzen der Eingabedaten anhand anderer Neuronen zu identifizieren. Generell werdend dadurch alternative Verbindungen zwischen den Schichten gefördert, um die Erkennung robuster zu machen. Im Falle von DeepKATE als Autoencoder wird das Netzwerk dazu angeregt Verbindungen (Gewichte) mit mehreren Bedeutungen zu überlagern. In der Testphase hingegen wird die Regularisierung nicht genutzt, sodass die Eingabedaten unverändert weitergereicht werden.

[1]Implementierung des *dropout*-Kernels in Flux `https://github.com/FluxML/Flux.jl/blob/4045c322d520b2bd63efe0a46b072d5f03eeccf4/src/layers/normalise.jl#L34`

Im Kontrast dazu wendet ein *KCompetetive*-Layer ein gegenteiliges Konzept an. Da ein *KCompetetive*-Layer die Energie aller Neuronen auf k „Gewinner“ Neuronen verteilt, wird eine Spezialisierung von Neuronen gefördert. Daher würde es z.B. wenig Sinn ergeben, ein *Dropout*-Layer direkt nach einem *KCompetetive*-Layer zu schalten, da dadurch die wenigen Aktivierungen weiter reduziert würden. Andererseits erwarte ich jedoch, dass die Spezialisierung von Neuronen durch das Vorschalten eines *Dropout*-Layers vor ein *KCompetetive*-Layer die Spezialisierung weiter fördert. Schaltet man ein *Dropout*-Layer vor ein *KCompetetive*-Layer nehme ich an, dass dieses dazu angeregt wird, komplexe Muster als Grundlage zur Entscheidungsfindung heranzuziehen. Die Idee dabei ist, das vermieden werden soll lediglich eine Entscheidung anhand weniger trivialer Merkmale der Eingabedaten zu treffen.

```
1  m = Chain(
2      KATE.KCompetetive(n, 100, tanh, k=25),
3      Dense(100, 20, sigmoid),
4      Dropout(0.4),
5      KATE.KCompetetive(20, 5, tanh, k=1),
6      Dense(5, 1, sin),
7      Dense(1, 5, sigmoid),
8      Dense(5, 20, sigmoid),
9      Dropout(0.4),
10     Dense(20, 100, sigmoid),
11     Dense(100, n, sigmoid),
12 )
```

QUELLCODE 3.7: DeepKATE Modell mittels Flux.jl. DeepKATE besteht aus jeweils fünf Encoder- und Decoder-Layer. Sowohl der Encoder, als auch der Decoder nutzen in der Trainingsphase ein Dropout-Layer.

Der *Encoder* (Zeilen 2-6) des Netzes versucht, die speziellen Eigenschaften der Layer so zu kombinieren, dass die Eingabedaten im latenten Raum gruppiert werden. Das Verhalten des Encoders ist komplex, wird jedoch stark durch die geringe Anzahl der Neuronen, sowie der *Dropout*-Layer beschränkt. Der *Decoder* (Zeilen 7-11) hingegen ist schlicht gehalten und weist neben den vier *Dense*-Layer mit sigmoider Aktivierungsfunktion ein *Dropout*-Layer auf.
Die Idee den Sinus als Aktivierungsfunktion in der Ausgabeschicht des *Encoders* (Zeile 6) zu nutzen, beruht auf der Annahme, dass es wünschenswert ist, Instanzen räumlich voneinander zu trennen, auch wenn diese gegen die selbe Aktivierung konvergieren, sich jedoch in ihrer Stärke um Größenordnungen unterscheiden können.

Zielfunktion

Die Zielfunktion von DeepKATE (Algorithmus 3.8) sieht vor, dass gleichzeitig drei Ziele verfolgt werden (*Pareto-Optimierung*). Zum einen, wird wie bei KATE [13] die Kreuzentropie genutzt, um den Rekonstruktionsfehler (*ce*) der Eingabe $b := x_n$ zu berechnen. Zum anderen, werden zwei weitere Ziele vorgegeben, die auf dem latenten Raum operieren. Die Idee der beiden Teilziele (*prev*, *succ*) ist, verschiedenartige Instanzen im latenten Raum durch das Training weiter von einander zu trennen. Dazu werden neben der eigentlichen *Forwardpropagation* (Zeilen 14-15) zwei weitere *Forward*-Schritte mittels einer Kopie des Modells durchgeführt, um latente Einbettungen der Eingaben $a := x_{n-1}$ und $c := x_{n+1}$ zu generieren (Zeilen 8-11). Das Ziel ist es die Einbettung (*embedded*) der Instanz b jeweils dem Komplement der Einbettungen anzunähern ($-target_a$ und $-target_c$).

Bei der Einbettung des latenten Raums handelt es sich um einen mehrdimensionalen Sinus (siehe Zeile 6 des Modells 3.7). Bei dem Komplement von *target_a* und *target_b* handelt es sich daher um die Umkehrung der Amplitude der Einbettung. Da der latente Raum keine Kategorien darstellt, wird auf dem latenten Raume eine Regression mittels *Mean-Squared-Error* (MSE - siehe Gleichung 2.15) durchgeführt (siehe Zeilen 19 und 20). Die Pareto-Optimierung findet unter der Annahme statt, dass dem Netzwerk unterschiedliche Punkte präsentiert werden. Falls jedoch für *a* und *c* die selben Instanzen präsentiert werden, gleichen sich die beiden Teilziele (*prev*, *succ*) aus. Das dritte Teilziel der Minimierung der Kreuzentropie als Rekonstruktionsfehler kommt immer zum Tragen.

```
1  function deep_kate_loss
2      Input:  m:      flux model              # neural net
3              lat:    integer                 # index of latent layer
4              a:      tracked array of float  # x_{n-1} (predecessor instance)
5              b:      tracked array of float  # x_n   (instance)
6              c:      tracked array of float  # x_{n+1} (successor instance)
7      Output: loss:   tracked float           # multi-objective
8
9      # prepare latent targets (forward-propagation)
10     cm := deepcopy(m)
11     Flux.testmode!(m)
12     target_a := Flux.data(cm[1:lat](a))
13     target_c := Flux.data(cm[1:lat](c))
14
15     # compute forward-propagation
16     embedded   := m[1:lat](b)
17     prediction := m[lat+1:end](embedded)
18
19     # compute errors
20     prev := Flux.mse(embedded, -target_a)
21     succ := Flux.mse(embedded, -target_c)
22     ce   := Flux.crossentropy(prediction, b)
23
24     # merge for multi-objective optimization
25     prev + ce + succ
26 end
```

QUELLCODE 3.8: DeepKATE Zielfunktion mittels Flux.jl. Die Zielfunktion ist eine Pareto-Optimierung von drei Teilzeilen: $prev, succ, ce$.

Die Zielfunktion weist zudem die Besonderheit auf, dass die Kopie des Modells (*cm*) in dem Modus zum Testen versetzt wird. Dies hat den Vorteil, dass die Berechnung der Einbettung ohne Umverteilung der Energie (siehe Algorithmus 3.6) erfolgt. Die Einbettungen (*target_a* und *target_c*) können daher effizient berechnet werden.

Die *Backpropagation* wird nach jeder Präsentation eines Tripels (x_{n-1}, x_n, x_{n+1}) auf dem erstellten *Computational Graph* (siehe Abschnitt 2.3.2) durchgeführt.

Eine Steigerung der Umkehrung der Amplitude der Einbettung, kann mittels einer Rechteckfunktion wie in Algorithmus 3.9 umgesetzt werden. Idee hierbei ist, dass eine Instanz an den Rand des Definitionsbereichs der Zielmenge des Sinus gedrückt wird. Der Abstand einer eingebetteten Instanz x_n (*embedded*) zum Ziel (*target*) wird durch die Funktion *repel* maximiert. Ich nehme an, dass das Lernen der beiden Teilziele (*prev*, *succ*) aus Algorithmus 3.8 dadurch weiter beschleunigt werden kann.

```
1  procedure repel!
2      Input:  xs:     array of float          # input signal
3              pivot:  float (default 0.0)     # origin
4      Output: xs:     array of float          # repelled input
5
6      for i in eachindex(xs)
```

```
7          if xs[i] >= pivot
8              xs[i] := ceil(xs[i])
9          else
10             xs[i] := floor(xs[i])
11         end
12     end
13     return xs
14 end
```

QUELLCODE 3.9: Durch die Prozedur *repel*, wird das Eingabesignal des Sinus an die Grenzen des Definitionsbereichs der Zielmenge $[-1, 1]$ gedrückt. Das Komplement $-$*repel* maximiert den Abstand zu zum Sinus, indem zugleich die Amplitude umgekehrt wird.

Es Abschnitt 4.1 wurde untersucht, ob die Hilfsfunktion *repel* sich dazu eignet, das Optimierungsproblem nicht bloß zu beschleunigen, sondern das gruppierende Verhalten von DeepKATE zu verstärken.
Die Abbildung 3.2 zeigt, wie sich das Komplement einer Einbettung *target*, welcher ein Sinus ist, an den Rand Definitionsbereichs der Zielmenge $[-1, 1]$ gedrückt wird.

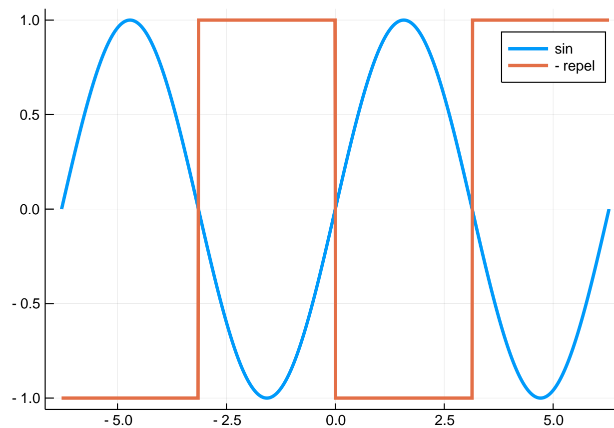


ABBILDUNG 3.2: Bei der Funktion Repel (3.9) handelt es sich um eine unstetige Rechteckfunktion. Wird die Amplitude des Sinus umgekehrt, dann wird folglich das Optimierungsproblem (*prev* oder *succ*) in Algorithmus 3.8 umgekehrt.

3.2.4 Clustering

Dadurch, dass die hochdimensionalen Daten in dieser Arbeit zunächst durch einen Autoencoder in einen niedrigdimensionalen Latenten-Raum eingebettet werden, können Clustering-Verfahren angewandt werden, die für wenige Dimensionen geeignet sind. Dies hat den Vorteil, dass nicht nur die eingebetteten Daten visualisiert werden können, sondern ein Clustering einfach nachvollzogen werden kann. Dem Anwender steht neben objektiven Metriken und daher zudem der subjektive Eindruck zur qualitativen Analyse zur Verfügung.

Folgende Clusteringverfahren wurden in der Arbeit genutzt, da dessen Skalierbarkeit im Vergleich zu anderen besonders hoch ist [60] und diese sich dazu eignen niedrigdimensionale Daten mittels Distanzmetriken in ein Clustering zu überführen:

- k-Means

- Fuzzy-C-means
- DBSCAN

Es wurden Implementierungen der Software Pakete *Clustering.jl* [61] und der Scikit-Learn Bibliothek [64] genutzt.

3.2.5 Detektion von Anomalien

Die Detektion von Anomalien mittels Autoencoder kann auf verschiedene Weisen erfolgen. In der Regel wird ein *Anomaly Score* genutzt, der Anhand der Rekonstruktion der Daten errechnet wird. Neben der deterministischen Berechnung des Rekonstruktionsfehlers, kann dies z.B. auch stochastisch geschehen, wenn stattdessen eine Rekonstruktionswahrscheinlichkeit berechnet wird [3]. Zudem ist es möglich mittels einer Distanzmetrik Anomalien durch Clustering zu bestimmen.
Um einzelne Punkte der Daten als Ausreißer zur erkennen, verwende ich mehrere Metriken. Diese werden gewichtet zu einer *Meta*-Metrik zusammengeführt.

Absoluter Rekonstruktionsfehler

Der absolute Rekonstruktionsfehler ergibt sich aus der Differenz der Vorhersage ($\hat{y}$) zur Grundgesamtheit (y), wie in Gleichung 3.3 dargestellt.

$$E_{abs} = \sum_{n \in N} |y(x_n) - \hat{y}(x_n)| \tag{3.3}$$

3.2.6 Log-Key Inferenz

Bei Algorithmus 3.10 handelt es sich um einen naiven Ansatz aus einer Menge von Log-Zeilen eine generische Repräsentation zu erzeugen. Der Ansatz ist naiv, da dieser weder die Syntax, noch die Semantik der Beispiele berücksichtigt, aus denen ein regulärer Ausdruck inferiert werden soll. Es wird davon ausgegangen, dass sich die Menge der durch das Clustering erstellten Gruppen (Cluster) bereits so homogen sind, dass gleiche Token an gleicher Stelle auftreten.

```
1  function infer(
2      samples::Vector{Vector{String}};
3      replacements::Union{Vector{String},Nothing} = nothing,
4      label::Regex = r"%[0-9A-Z\_]*?%",
5      asterix::String = ".*?",
6      regex = false)
7
8      max = 0
9      for s in eachindex(samples)
10         sample = samples[s]
11         if length(sample) > max
12             max = length(sample)
13         end
14     end
15
16     function push_word!(set, word)
17         if replacements != nothing && any(p->contains(word, p), replacements)
18             escaped = RegExp.escape(word)
19             push!(set, replace(escaped, label, asterix))
20         else
21             push!(set, RegExp.escape(word))
22         end
23     end
24
25     groups = OrderedDict{Int64,OrderedSet{String}}()
```

```
26      for s in eachindex(samples)
27          for w in eachindex(samples[s])
28              word  = samples[s][w]
29              if haskey(groups, w)
30                  push_word!(groups[w], word)
31              else
32                  groups[w] = OrderedSet{String}()
33                  push_word!(groups[w], word)
34              end
35          end
36      end
37
38      joins = Vector{String}(max)
39      for g in eachindex(groups)
40          group = collect(groups[g])
41          if length(group) > 1
42              joins[g] = string("(",join(group,"|"),")")
43          else
44              joins[g] = join(group)
45          end
46      end
47
48      joined = strip(join(joins))
49
50      if regex
51          try return Regex(joined) catch; warn(joined) end
52      else
53          return joined
54      end
55  end
```

QUELLCODE 3.10: Dieser Algorithmus inferiert aus einer Menge von Samples (*augmented Log-Keys*) eine Repräsentation als regulären Ausdruck.

Von Zeile 8 bis 14 wird die maximale Länge aller Samples ermittelt. Dies dient dazu in Zeile 38 einen Vektor fester Größe zu allozieren und mittels Index ansprechen zu können. Im zweiten Schritt, werden Metazeichen der Eingabemenge deaktiviert. Gegebenenfalls werden danach zudem zu ersetzende Worte durch eine gewünschte Repräsentation ersetzt. Dies ermöglicht es einen beschreibenden Log-Key in einen generischen umzuwandeln. Ersetzt werden alle Token, die in der Menge der Ersetzungen (*replacements*) vorkommen durch einen Platzhalter (*asterix*). Standardeinstellung für den *asterix* Ausdruck ist eine „genügsame“ (*non-greedy*) beliebige Zeichenfolge (Kleensche-Hülle eines beliebigen Zeichens). Ein genügsamer Quantor bricht die Suche nach dem ersten Auftritt (*Match*) eines akzeptierten Zustands ab.
Nachdem die Worte in den Zeilen vorverarbeitet wurden, werden die Worte aller Samples an der selben Stelle zu einer Gruppe zusammengefasst.
Des Weiteren berücksichtigt der Algorithmus keine Mehrfachauftritte von Token mittels Quantoren, wie dies z.B. durch [76] vorgenommen wird (siehe Beispiel 2.6).

Beispiel 3.1. Aus gegebener Menge beschreibender Log-Keys:

```
[
    String["%RCE_DATETIME%", " ", "DEBUG", " ", "-", " ", "de", ".", "rcenvironment
        ", ".", "core", ".", "communication", ".", "transport", ".", "jms", ".", "
        activemq"],
    String["%RCE_DATETIME%", " ", "DEBUG", " ", "-", " ", "de", ".", "rcenvironment
        ", ".", "core", ".", "communication", ".", "transport", ".", "jms", ".", "
        common"]
]
```

inferiert der Algorithmus folgenden regulären Ausdruck:

```
r"%RCE_DATETIME% DEBUG \- de\.rcenvironment\.core\.communication\.transport\.jms\.(
    activemq|common)"
```

Wie in Beispiel 3.1 zu sehen ist, werden Metazeichen regulärer Ausdrücke durch einen Backslash mittels *escape* Funktion deaktiviert (siehe Zeile 18, 21). Welche Zeichen tatsächlich Metazeichen von regulären Ausdrücken sind, ist sowohl vom Kontext, als auch der Implementierung abhängig. Die beiden Token „activemq“ und „common“ wurden zu einer Gruppe zusammengeführt, da diese an der selben Stelle eines Samples in der Eingabemenge vorkommen.
Im nächsten Abschnitt wird auf das Episode Mining eingegangen, welches dazu dient serielle Episoden, wie in Abschnitt (2.5) dargestellt aufzufinden.

3.2.7 Episoden Mining (EM)

Um temporale Zusammenhänge von *Events* in *Event-Logs* aufzuzeigen, wurden zwei Verfahren implementiert. Ziel des Episode Mining (EM) ist, serielle Episoden zu generieren, um eine Ursachen-Wirkungs-Analyse (Root-Cause-Analysis) mittels *Episode-Rule Mining* zu betreiben. Dazu werden aus den gefundenen seriellen Episoden, welche durch den Nutzer vorgegebene Suchkriterien (*constraints*) erfüllen, in einem zweiten Schritt in Regeln von Episoden (*Episode-Rules*) überführt.
Bei dem einen Verfahren handelt es sich um einen Algorithmus mit Tiefensuche auf einer projizierten Vertikalen Datenbank, welcher sich an das Vorgehen von *SPADE* [87] anlehnt. Bei dem anderen handelt es sich um die Übertragung von *TSpan* [25] auf einen einfachen *Event-Log* mittels *EWU* Bedingung.

Beide Algorithmen umfassen dabei die folgende Kriterien:

- min_sup: Minimum Support, Bedingung über den minimalen Auftritt einer seriellen Episode.
- $min_utililty$: relative Nützlichkeit, Bedingung zu über die Mindestnützlichkeit einer seriellen Episode.
- $max_repetitions$: maximal erlaubte Wiederholungen eines Eventtyps in einer seriellen Episode.
- max_gap: maximal erlaubte Zeitsprünge zwischen Events.
- $max_time_duration$ (mtd): maximale Musterlänge + 1.

Unter der Angabe der folgenden Anfangswerte, kann weiter Einfluss auf Suchraum genommen werden:

- $prefixes$: Präfixe, die erweitert werden sollen.
- $utilities$: Externe Nützlichkeit, Zuordnung für jeden Eventtyp ($profit$).

Dadurch, dass zu erweiternde *Präfixe* als Anfangswert angegeben werden können, kann gezielt nach bestimmten Konsequenzen gesucht werden. Dazu muss die originale Sequenz umgekehrt werden, wobei eine gesuchte Konsequenz zur Prämisse wird. Durch erneutes Umkehren gefundener Episoden kann ermittelt werden, welche Ereignisse zu einer bestimmten Konsequenz geführt haben.

MV-Span

Der MV-Span Alogrithmus besteht aus zwei Teilen: Zum einen der Prozedur für das Präfix-Wachstum (*grow_mv_span*! - Algorithmus 3.11) mittels Tiefensuche auf einer

projezierten Vertikalen Datenbank. Zum anderen aus der Funktion (Algorithmus 3.12), welche den rekursiven Algorithmus *grow_mv_span*! nutzt, um die Ergebnismenge über alle Episoden aufbaut, welche die vorgegebenen Kriterien erfüllen. Die Idee der vertikalen Projektion besteht darin lediglich solche Eventtypen als Kandidaten zu betrachten, die tatsächlich das bisher gefundene *Präfix* erweitern können.

```
procedure grow_mv_span!

    Input:
        db::OrderedDict{Vector{Int},Vector{Vector{Int}}},
        prefixes::Vector{Vector{Int}},
        sequence::Vector{Int},
        len::Int64,
        vertical::Dict{Int,Vector{Int}},
        alphabet::Vector{Int};

        utility_measure: Union{Nothing, Symbol} = :external, #
            um ∈ {: external,: local,: average}
        utilities: Union{Nothing, Dict{Int64,Int64}} = nothing,
        total_utility = 0,      # total sequence utitility
        min_utility   = 0.0,    # relative minimum utility
        min_sup       = 1,      # absolute minimum support
        max_repetitions = 0,    # n <= 0 → endless, n > 0 → maximal neventtype repititions
        overlapping   = false,  # false → minimal occurrences
        max_gap       = 0,      # −1 → endless, 0 → contiguous, max_gap > 0 → non contiguous
        max_time      = -1,     # −1 → endless, mtd → maximal episode length − 1
        result_set    = :all,   # result_set ∈ {: all,: closed}
        depth         = 0       # episode length − 1

    Output:
        db::OrderedDict{Vector{Int},Vector{Vector{Int}}}

    for pattern in prefixes

        if max_time > -1 && length(pattern) > max_time
            continue
        end

        support = length(db[pattern])
        if support < min_sup
            delete!(db, pattern)
            continue
        end

        if utility_measure != nothing
            if utility_measure == :external && external_utility(utilities,
                total_utility, pattern) < min_utility
                delete!(db, pattern)
                continue
            elseif utility_measure == :average  && avg_utility(utilities,
                total_utility, len, pattern) < min_utility
                delete!(db, pattern)
                continue
            elseif utility_measure == :local && local_utility(utilities, pattern) <
                min_utility
                delete!(db, pattern)
                continue
            end
        end

        foundat_all = Set{Int64}()
        for s_ext in alphabet
            if max_repetitions > 0
                if max_repetitions == 1
                    if s_ext in pattern
                        continue
                    end
                else
                    if count(e->e == s_ext, pattern) >= max_repetitions
                        continue
                    end
```

```
62                  end
63              end
64              foundat = Vector{Int64}()
65              s_extension = Array{Int64}(undef, depth+2)
66              s_extension[1:depth+1] = pattern
67              s_extension[end] = s_ext
68
69              for i in 1:support
70                  start = db[pattern][i][end] + 1
71                  stop = len
72                  if max_gap >= 0
73                      stop = min(len, start + max_gap)
74                  end
75
76                  for candidate in @views vertical[s_ext]
77                      if candidate > stop
78                          break
79                      end
80
81                      if candidate >= start
82                          occurence = Array{Int64}(undef, depth+2)
83                          occurence[1:depth+1] = db[pattern][i]
84                          occurence[end] = candidate
85                          if haskey(db, s_extension)
86                              if overlapping
87                                  push!(db[s_extension], occurence)
88                                  push!(foundat, i)
89                              elseif db[s_extension][end][end] <= db[pattern][i][1]
90                                  push!(db[s_extension], occurence)
91                                  push!(foundat, i)
92                              end
93                          else
94                              db[s_extension] = [occurence]
95                              push!(foundat, i)
96                          end
97                      end
98                  end
99              end
100
101             if length(foundat) >= min_sup
102                 union!(foundat_all, foundat)
103
104                 mv_span(
105                     db, [s_extension],
106                     sequence, len, vertical, alphabet;
107                     utility_measure = utility_measure,
108                     utilities = utilities,
109                     total_utility = total_utility,
110                     min_utility = min_utility,
111                     min_sup = min_sup,
112                     max_repetitions = max_repetitions,
113                     overlapping = overlapping,
114                     max_gap = max_gap,
115                     max_time = max_time,
116                     result_set = result_set,
117                     depth = depth + 1)
118             else
119                 delete!(db, s_extension)
120             end
121         end
122
123         if result_set == :closed
124             d = 0
125             for i in sort(collect(foundat_all))
126                 if i > d
127                     deleteat!(db[pattern],i-d)
128                     d += 1
129                 end
130             end
131             support = length(db[pattern])
132         end
133
```

```
134         support = length(db[pattern])
135         if support < min_sup
136             delete!(db, pattern)
137         end
138     end
139
140     return db
141 end
```

QUELLCODE 3.11: Der Algorithmus *MV-Span-Wachstum* (*grow_mv_span*) generiert für gegebene Präfixe ($prefixes$) serielle Erweiterungen unter Berücksichtigung der gegebenen Bedingungen.

Dadurch, dass in Zeile 77 für ein *break* geprüft wird, ob $candidate > stop$ ist, gilt für Zeile 81, dass $candidate <= stop$ ist.

```
1  function mv_span
2
3      Input:
4          sequence::Vector{Int};
5          prefixes::Union{Nothing, Vector{Vector{Int}}} = nothing,
6          utilities::Union{Nothing, Dict{Int,Int}} = nothing,
7          utility: Symbol = :external, # {:external, :local, :average}
8          min_utility         = 0.0,   # rel min utility
9          min_sup             = 1,     # abs. min support
10         max_repetitions     = 0,     # max_rep < 1 --> endless
11         max_gap             = 0,     # max_gap == -1 --> endless
12         max_time_duration   = -1,    # mtd == -1 --> endless
13         min_occurrences     = true,  # true --> only minimal occurences
14         result_set          = :all   # {:all, :closed}
15
16     Output:
17         db::OrderedDict{Vector{Int64},Vector{Vector{Int64}}}
18
19     seq_len = length(sequence)
20     vertical = Index.invert(sequence)
21     vertical = filter(kv->length(kv[2]) >= min_sup, vertical)
22     alphabet = map(kv->kv[1], sort(collect(vertical),
23                                by = kv -> length(kv[2]),
24                                rev=true))
25     total_utility = 0
26     if utilities != nothing
27         total_utility = sum(e->utilities[e], sequence)
28     end
29
30     db = OrderedDict{Vector{Int64},Vector{Vector{Int64}}}()
31     for (key, values) in collect(vertical)
32         db[[key]] = collect(map(v->[v], values))
33     end
34
35     if prefixes == nothing
36         prefixes = sort(collect(keys(db)), by = k -> length(db[k]))
37     end
38
39     grow_mv_span!(
40         db,
41         prefixes,
42         sequence,
43         seq_len,
44         vertical,
45         alphabet;
46         utility_measure     = utility,
47         utilities           = utilities,
48         total_utility       = total_utility,
49         min_utility         = min_utility,
50         min_sup             = min_sup,
51         max_repetitions     = max_repetitions + 1,
52         overlapping         = !min_occurrences,
53         max_gap             = max_gap,
54         max_time_duration   = max_time_duration,
55         set                 = result_set)
56
```

```
57      # filter 1-event episodes
58      if min_occurrences
59          for k in keys(db)
60              if length(k) == 1
61                  delete!(db, k)
62              end
63          end
64      end
65
66      return db
67  end
```

QUELLCODE 3.12: Der MV-Span Algorithmus ist an SPADE angelehnt und nutzt eine pseudo projizierte *vertikale Datenbank*.

Der Algorithmus 3.12 nutzt eine vertikale Datenbank der Auftritte aller Events (*vertical*) für eine gegebene Sequenz (*sequence*). Die Idee des Algorithmus ist es lediglich Kandidaten zu generieren, die tatsächlich in der Sequenz (*sequence*) vorkommen. Der Suchraum kann durch die 5 Bedingungen (*min_sup*, *min_utility*, *max_repetitions*, *max_gap*, *max_time_duration*) und Anfangswerte der gesuchten Präfixe (*prefixes*) und Nützlichkeiten (*utilities*) beeinflusst werden werden.
Es wurden 3 Nützlichkeitsfunktionen implementiert, welche die externe Nützlichkeit unterschiedlich gewichtet als Filter nutzen.

MT-Span

Bei MT-Span (siehe Algorithmus 3.14) handelt es sich um eine in dieser Arbeit erstellte Abwandlung des TSpan Algorithmus [25], welcher wiederum auf UP-Span [82] und USpan [84] aufbaut. Dabei wurden die Strategien *EWU* (*Episode-Weighted Utilization*) nach [82] und *IESC* (*Improved estimation of EWU for S-Concatenation*) nach [25] für *serielles Episode Mining* auf *komplexen Event-Sequenzen* für die Mustererkennung auf *einfachen Event-Sequenzen* angepasst. Für die EWU-Strategie gilt, dass diese die *Downward Closure Property* (siehe Definition 2.20) erfüllt [25].

Im Gegensatz zu MV-Span wird bei MT-Span keine projizierte Datenbank genutzt, um Kandidaten zu generieren. Stattdessen wird das Vorgehen des Präfixwachstums (*prefix-growth*, bzw. *pattern-growth*), wie bei TSpan übernommen. TSpan erweitert dabei sowohl parallele Episoden mittels *I-Concatenation*, sowie serielle Episoden durch die *S-Concatenation*, wie bei USpan [84]. MT-Span nutzt von beiden Verfahren lediglich die *S-Concatenation* (siehe Algorithmus 3.13) für serielles Episoden Mining.

Das generelle Vorgehen von TSpan sieht vor, mittels EWU und IESC die obere Grenze (*upper bound*) der Nützlichkeit (*utililty*) einer Episode abzuschätzen, um dadurch den Suchraum effizient einzuschränken. Die EWU-Strategie wird dabei als Indikator genutzt, ob eine Episode auf ihre tatsächliche Nützlichkeit (*utility*) geprüft wird. IESC hingegen gibt an, ob der Suchraum expandiert wird. Dieses Vorgehen wurde in MT-Span übernommen und weiter ergänzt, wie im folgenden Absatz erklärt wird.

Der *Prefix-Growth* Algorithmus nach [25] wurde neben der Nützlichkeit (u) und der maximalen Dauer (MTD), um drei weitere Bedingungen erweitert. Zum einen wird mittels *Minimum Support* (*min_sup*) sichergestellt, dass nur Kandidaten (β, bzw. `beta`) in die Ergebnismenge (*hueSet* - *High Utility Episode Set*) aufgenommen werden, die diesen tatsächlich erfüllen (siehe Zeile 84). Zum anderen wird mittels *min_sup* der Suchraum zusätzlich (siehe Zeile 91) eingeschränkt. Eventtypen, die den *Minimum Support* nicht erfüllen, werden ebenfalls ignoriert, wie in Zeile 32

zu sehen ist. Die *max_repitition*s Bedingung legt fest, wie oft ein Eventtyp in einer seriellen Episode auftreten darf. Tritt eine Eventtyp bereits maximal nach dem *max_repititions* auf, wird die Erweiterung des aktuellen Kandidaten β abgebrochen (siehe Zeile 47). Falls $|mo(\beta)| \geq |\alpha| + max_gap$ ist, wird das Intervall $mo(\beta)$ verworfen und nicht weiter untersucht (siehe Zeile 28).

```
1  procedure s_concatenation!
2
3      Input:
4          sequence::Vector{Int64},
5          supports::Vector{Int64},
6          utilities::Vector{Float64},
7          prefix::Vector{Int64},
8          moSet::Dict{Vector{Int64},Vector{Intervall}},  # minimal occurrences
9          hueSet::Dict{Vector{Int64},Vector{Intervall}}, # high utility episodes
10         max_time_duration::Int64,
11         min_sup::Int64,
12         min_utililty::Float64,
13         total_utility::Float64,
14         max_repetitions::Int64,
15         max_gap::Int64
16
17     Output:
18         moSet::Dict{Vector{Int64},Vector{Intervall}},  # minimal occurrences
19         hueSet::Dict{Vector{Int64},Vector{Intervall}}  # high utility episodes
20
21     l = length(prefix)
22
23     for range in moSet[prefix]
24         I = range.stop+1:min(range.start+max_time_duration+1,length(sequence))
25         for t in I
26             moBeta::Intervall = range.start:t
27
28             if max_gap >= 0 && length(moBeta) >= l + max_gap
29                 break
30             end
31
32             if supports[sequence[t]] < min_sup
33                 continue
34             end
35
36             beta = Vector{Int64}(undef, l+1)
37             beta[1:l] = prefix
38             beta[end] = sequence[t]
39
40             if max_repetitions >= 0
41                 c = 0
42                 for e in beta
43                     if e == sequence[t]
44                         c += 1
45                     end
46                 end
47                 if c-1 > max_repetitions
48                     break
49                 end
50             end
51
52             if !haskey(moSet, beta)
53                 moSet[beta] = Int64[]
54             end
55
56             # TSpan: M = {mo | mo ∈ moSet(β) and mo ⊆ mo(β)}
57             M = Vector{Intervall}()
58             for mo in moSet[beta]
59                 if mo.start >= moBeta.start && mo.stop <= moBeta.stop
60                     push!(M, mo)
61                 end
62             end
63             if isempty(M)
64                 # TSpan: N = {mo | mo ∈ moSet(β) and mo(β) ⊂ mo}
65                 N = Vector{Intervall}()
```

```
66                 for mo in moSet[beta]
67                     if moBeta.start > mo.start && moBeta.stop <= mo.stop && mo.stop
                           <= moBeta.start
68                         push!(N, mo)
69                     end
70                 end
71                 if !isempty(N)
72                     filtered = Vector{Intervall}()
73                     for mo in moSet[beta]
74                         if !(mo in N)
75                             push!(filtered,mo)
76                         end
77                     end
78                     moSet[beta] = filtered
79                     push!(moSet[beta], moBeta)
80                 else
81                     push!(moSet[beta], moBeta)
82 
83                     # UP-Span: EWU(β) ≥ min_utility
84                     if support(moSet, beta) >= min_sup && ewu(total_utility,
                           utilities, beta, moSet) >= min_utililty
85                         # check exact utility
86                         if relative_utility(total_utility, utilities, beta) >=
                               min_utililty
87                             hueSet[beta] = moSet[beta]
88                         end
89 
90                         # TSpan: IESC(α, SES) ≥ min_utility
91                         if support(moSet, prefix) >= min_sup && iesc(total_utility,
                               utilities, prefix, Set{Int64}(sequence[I])) >=
                               min_utililty
92                             s_concatenation!(
93                                 sequence,
94                                 supports,
95                                 utilities,
96                                 beta,
97                                 moSet,
98                                 hueSet,
99                                 max_time_duration,
100                                min_sup,
101                                min_utililty,
102                                total_utility,
103                                max_repetitions,
104                                max_gap)
105                        end #if (line 91)
106                    end # if (line 84)
107                end # if (line 71)
108            end # if (line 63)
109        end # for (line 25)
110    end # for (line 23)
111 end # procedure
```

QUELLCODE 3.13: Die *S-Concatenation* erweitert Präfixe (*prefix-growth*) um valide Kandidaten, unter Berücksichtigung der EWU-Strategie und weiteren Konditionen.

Nach [25] kann die Ergebnismenge (*hueSet*) iterativ aufgebaut werden. Der rekursive Aufruf (siehe Zeile 104) kann unabhängig von anderen Präfixen stattfinden, was zudem eine Parallelisierung der Suche über alle Präfixe ermöglicht. Die Parallelisierung kann in Zeile 35 des MT-Span Algorithmus ansetzen.

MT-Span (siehe Algorithmus 3.14) ist ein *Prefix-Growth* Algorithmus für *einfache Event-Sequenzen*. Er ist an TSpan angelehnt und erweitert diesen um die zusätzlichen Bedingungen (*min_sup*, *max_repititions*, *max_gap*). Das eigentliche Wachstum wird mittels der *S-Concatenation* (siehe Algorithmus 3.13 durchgeführt. Der MT-Span Algorithmus dient als Einstiegspunkt (Rekursionsanfang) und stellt sicher, dass nur Präfixe erweitert werden, die sowohl das *min_sup* Kriterium erfüllen, als auch die

relative Mindestnützlichkeit (*min_utility*) mittels *IESC* nach [25] erfüllen (siehe Zeile 37).

```
1  function mt_span
2
3      Input:
4          sequence::Vector{Int64},
5          utilities::Vector{Float64},
6          max_time_duration::Int64 = 0, # mtd
7          min_sup::Int64 = 1,
8          min_utililty::Float64 = 0.0,
9          max_repetitions::Int64 = -1,
10         max_gap::Int64 = -1;
11         prefixes::Union{Symbol,Vector{Vector{Int64}}} = :all)
12
13     Output:
14         moSet::Dict{Vector{Int64},Vector{Intervall}},  # minimal occurrences
15         hueSet::Dict{Vector{Int64},Vector{Intervall}}  # high utility episodes
16
17     vertical::Dict{Int,Vector{Int64}} = Index.invert(sequence)
18     k_max = maximum(keys(vertical))
19     supports::Vector{Int64} = fill(0,k_max)
20     for (k,v) in vertical
21         supports[k] = length(v)
22     end
23     vertical = filter(kv -> length(kv[2]) >= min_sup, vertical)
24
25     moSet = Dict{Vector{Int64},Vector{Intervall}}(
26         map(kv -> [kv[1]] => map(v -> v:v, kv[2]), collect(vertical)))
27     hueSet = Dict{Vector{Int64},Vector{Intervall}}()
28
29     tu = total_utility(sequence, utilities)
30
31     if prefixes == :all
32         prefixes = deepcopy(sort(collect(keys(moSet)), by = p -> p[1]))
33     end
34
35     for i = 1:length(prefixes)
36         prefix = prefixes[i]
37         if support(moSet, prefix) >= min_sup && iesc(tu, utilities, prefix, prefix)
               >= min_utililty
38             s_concatenation!(
39                 sequence,
40                 supports,
41                 utilities,
42                 prefix,
43                 moSet,
44                 hueSet,
45                 max_time_duration,
46                 min_sup,
47                 min_utililty,
48                 tu,
49                 max_repetitions,
50                 max_gap)
51         end
52     end
53
54     moSet, hueSet
55 end
```

QUELLCODE 3.14: MT-Span ist ein *Prefix-Growth* Algorithmus für *einfache Event-Sequenzen*. Er ist an TSpan angelehnt und erweitert diesen um die zusätzlichen Bedingungen (*min_sup*, *max_repititions*, *max_gap*).

3.2.8 Eventvoraussage

Um Events einer Sequenz vorauszusagen, wurden rekurrente Netze genutzt, wie in Abschnitt 2.3.1 beschrieben.

Um mittels *Flux.jl* auch Bidirektionale LSTMs (Bi-LSTM) erstellen zu können, wurde die Datenstruktur *Parallel* (3.2.8) geschaffen. Diese ermöglicht das Anwenden des MapReduce-Paradigmas, um die Ergebnisse unabhängiger Netze (Layer) zusammen zuführen.

```
1 struct Parallel
2     layers::Vector{L}
3     map::Vector{Function}
4     reduce::Function
5 end
```

Die Datenstruktur *Parallel* hält sowohl Layer unabhängiger Netze vor, als auch Mapping Funktionen für jedes Layer. Wobei ein Layer L ein rekurrentes Netz, wie bspw. ein LSTM sein kann. Zudem wird mittels *reduce* Funktion vorgegeben, wie die Ergebnisse der verschiedenen Layer zusammengeführt werden. Es wurden zwei Ansätze implementiert. Zum einen können die Ergebnisse überlagert werden und mittels Mittelwerts-Funktion (*average*) zusammen geführt werden. Zum anderen können diese zu einem Vektor konkateniert (*concat*) werden.

Das Verhalten eines *Parallel*-Layers ist, wie in Funktion 3.2.8 definiert. Für jedes *Layer* wird eine Mapping Funktion auf die Eingabedaten angewandt, bevor diese von dem Layer verarbeitet werden. Dies ermöglicht es z.B. eine Sequenz von Events umzukehren, bevor diese durch ein Layer verarbeitet wird. Standardmäßig, wie hier nicht zu sehen ist, wird die Identität der Daten weitergereicht.

```
1 function (p::Parallel)(xs)
2     layers, map, reduce = p.layers, p.map, p.reduce
3     mapped = Base.map(l-> layers[l](map[l](xs)), eachindex(layers))
4     reduce(mapped)
5 end
```

Im Anschluss auf die Mapping-Phase folgt das Zusammenführen mittels *Reduce*-Funktion. Die resultierende Vektorgröße ist von der *Reduce*-Funktion abhängig.
Neben dem Bi-LSTMs wurde ein *peephole* LSTM für das *Flux.jl* Framework nach [22] implementiert.

Kapitel 4

Ergebnisse

Es wurden verschiedene Experimente durchgeführt, um sowohl die DeepKATE Architektur, als auch die vorgeschlagenen Episode Mining Algorithmen MV-Span und MT-Span zu evaluieren.

4.1 DeepKATE

In der Arbeit wurden zwei Datensätze herangezogen, um das DeepKATE Modell zu evaluieren.
Im Rahmen der Arbeit wurde ein interner Datensatz des DLR von Log-Dateien der RCE Anwendung zusammengetragen. Dieser umfasst 70 Log-Dateien mit durchschnittlich 15225 Log-Zeilen. Im Median beträgt die Länge der Log-Dateien 1423 Log-Zeilen. Der Datensatz enthält sensible Informationen und kann daher nicht öffentlich zugänglich gemacht werden. Aus diesem Grund wurde das DeepKATE Modell ebenfalls mit dem ISCX CICIDS2017 Datensatz [71] für Maschinelles Lernen zur Ausreißererkennung von Netzwerkverkehrsdaten evaluiert. Der CICIDS2017 Datensatz enthält für das maschinelle Lernen aufbereitete CSV-Dateien, die 80 Attribute enthalten. Die Attribute wurden von [71] mittels CICFlowMeter [37] aus Echtzeitmitschnitten des Netzwerkverkehrs (PCAP-Dateien) von verschiedenen Angriffsszenarien gewonnen.

DeepKATE - RCE Datensatz

In den Abbildungen (1, 2, 3, 4, 5, 6) des Anhangs sind verschiedene Epochen der Einbettung mittels DeepKATE dargestellt. Die Dimension des latenten Raums kann bei der Einbettung frei gewählt werden. Es werden Einbettungen einer Log-Datei aus dem RCE Datensatz gezeigt, die für das DeepKATE-Modell typisch sind. Dabei handelt es sich um verschiedene Einbettungen der selben Log-Datei (6073 Log-Zeilen) in einen zwei- und dreidimensionalen latenten Raum. Die Einbettungen wurden mittels DBSCAN in ein Clustering überführt, wobei der Parameter des Radius' (*epsilon*) variiert wurde. Durch einen größeren Radius werden weniger Cluster gebildet, wie in den Abbildungen zu sehen ist. Die Anzahl der Cluster ist auf der rechten Skala der Abbildungen (1, 2, 3, 4, 5, 6) zu entnehmen.

Die Abbildung 4.1 zeigt eine Korrelationsmatrix verschiedener Parameter, der Einbettung und des Clusterings, im Verhältnis zum Rekonstruktionsfehler (*error*) und verschiedener CVIs. Dabei ist besonders der Radius (*epsilon*) des DBSCAN Verfahrens von Interesse, da dieser indirekt die Anzahl resultierender Cluster bestimmt. Der DBSCAN-Parameter der mindest Nachbarschaftsgröße (*min_neighbors*) ist bei der Erkennung von Ausreißern hinderlich. Wenn die Mindestgröße der Cluster

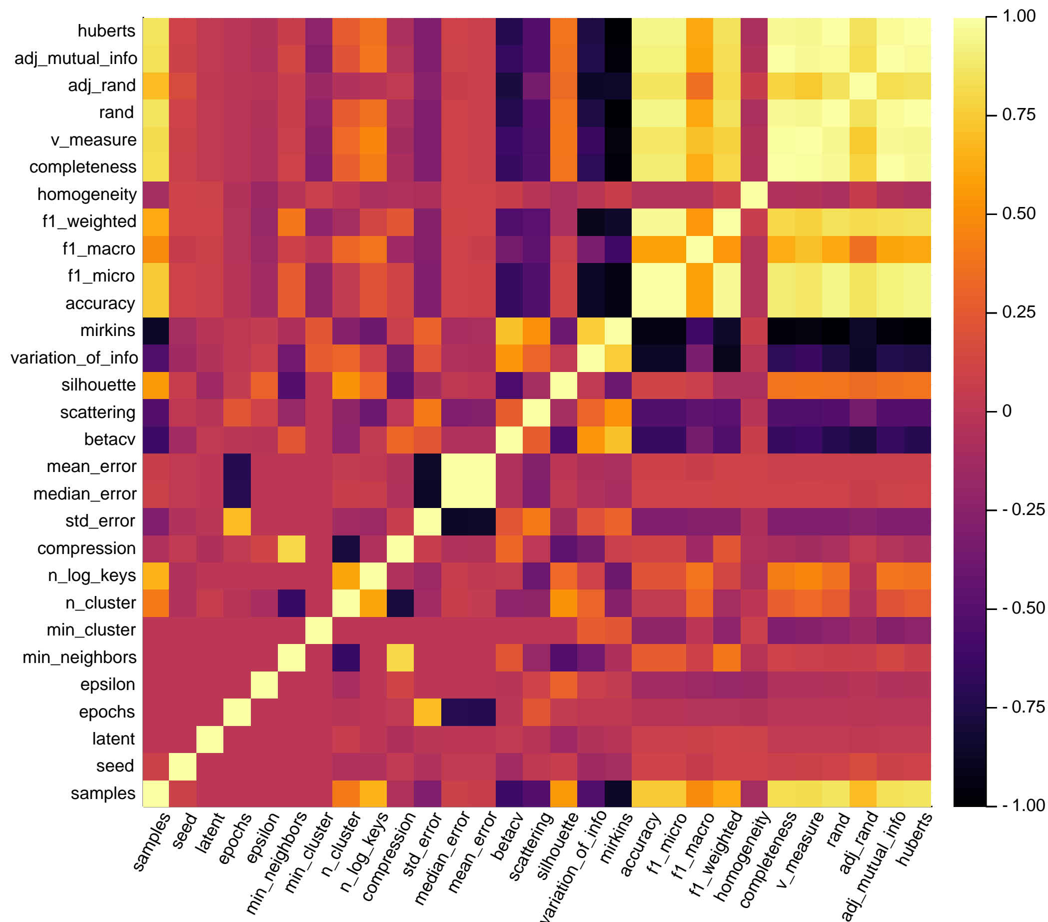


ABBILDUNG 4.1: Korrelationsmatrix verschiedener *Clustering Validity Indices* (CVI) von 32400 Kombinationen verschiedener Parameter bei der Einbettung mittels DeepKATE (*seed*, *latent*) und dem Clustering mittels DBSCAN (*epsilon*, $min_{n}eighbors$).

größer als 1 gesetzt wird, werden mögliche Ausreißer-Instanzen zwingend mit anderen Instanzen zu einem Cluster zusammengeführt, die möglicherweise keinen Ausreißer darstellen, sondern im latenten Raum weit entfernt sein können.

In Abbildung 4.1 ist ebenfalls zu sehen, das die CVIs *BetaCV* und das *Scattering* stark mit der Anzahl der Samples korrelieren, was in der Regel unerwünscht ist. Es deutet darauf hindeutet, dass die Metriken besonders für wenige Samples nicht aussagekräftig sind. Die Annahme mittels *BetaCV* oder des *Scattering* als interne Validierungsmaße, nicht nur die Güte resultierender Clusterings zu bestimmen, sondern zudem indirekt die Güte von DeepKATE als Dimensionsreduktionsverfahren evaluieren zu können, ist in Frage zu stellen.

Des Weiteren kann der Abbildung 4.1 entnommen werden, dass der Rekonstruktionsfehler negativ mit der Anzahl der Epochen des Trainings des DeepKATE Modells korreliert. Dadurch, dass die Daten mit dem Algorithmus 3.4 normalisiert

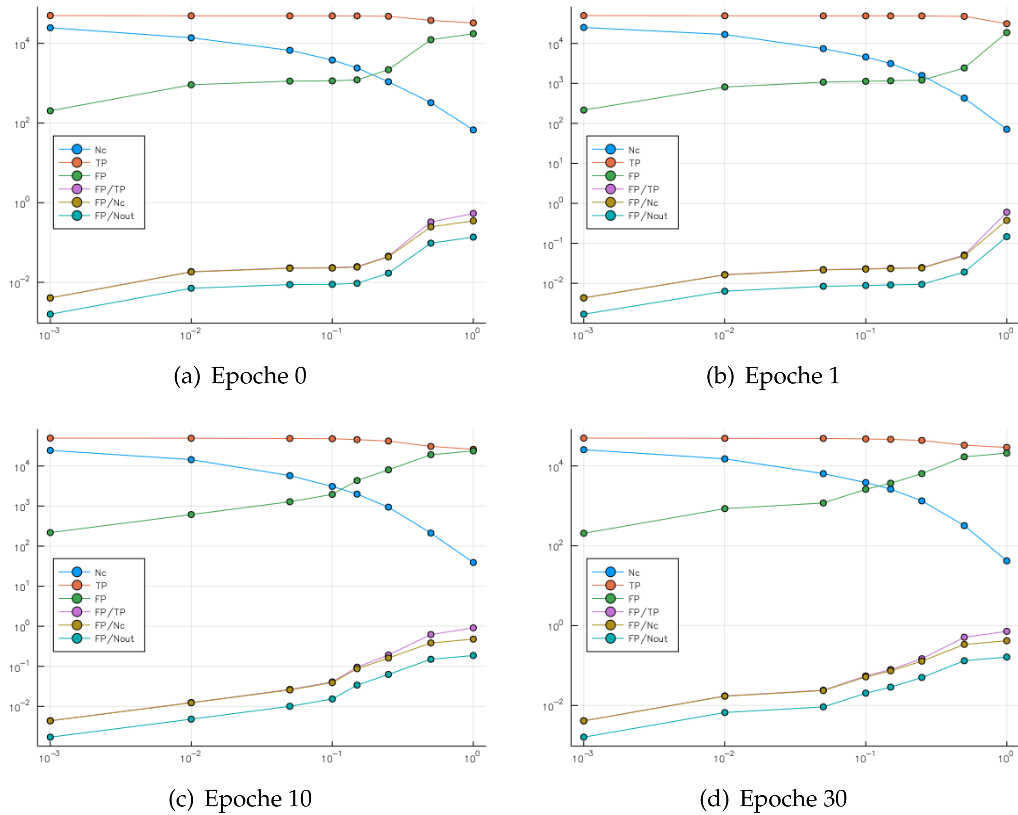
werden, wird durch DeepKATE eine Regression auf den Daten durchgeführt. Die benannte Korrelation deutet daraufhin, dass das DeepKATE-Verfahren in der Regel gegen einen *pareto-optimalen* Zustand konvergiert. Zudem kann der Abbildung 4.1 entnommen werden, dass sich die Standardabweichung des Rekonstruktionsfehlers (*std_error*) mit der Anzahl der Epochen des Trainings erhöht. Dies stellt ein interessantes Verhalten dar, da Instanzen, die zu anderen eine große Distanz aufweisen, mit der Dauer des Trainings weiter von diesen getrennt werden (im Sinne des Rekonstruktionsfehlers, nicht des latenten Raums). Dadurch ist es einfacher solche Instanzen von anderen mittels einer Distanzmetrik zu unterscheiden. Neben dem Rekonstruktionsfehler, kann daher auch der Abstand einer Instanz zu einem Repräsentanten gegebener Daten (z.B. zum Schwerpunkt oder Mittelpunkt) als Maß zur Detektion von Ausreißer-Instanzen herangezogen werden. Eine weitere Möglichkeit Ausreißer-Instanzen festzumachen, besteht darin die Anzahl der Mitglieder eines Clusters als Kriterium heranzuziehen.

DeepKATE - ISCX CICIDS2017 Datensatz

Im Umgang mit dem CICIDS2017 Datensatz hat sich gezeigt, dass durch die Präsentation vieler Instanzen während des Trainings die Qualität der Einbettung im Bezug auf die Herausarbeitung von Clustern stark eingeschränkt wird. Statt in der Trainingsphase viele Instanzen für das Training heranzuziehen, kann auch mit nur wenigen *Samples* ein latenter Raum erzeugt werden, der durch das Clustering mittels DBSCAN gute Ergebnisse erzeugt. Die Güte bezieht sich hier auf die Homogenität der gebildeten Cluster.

Abbildung 4.2 zeigt die Auswertung von der Einbettungen (siehe Abbildungen 7, 8, 9, 10 des Anhangs) von DDoS-Attacken aus dem CICIDS2017 Datensatz (`Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv`) mit 225745 Instanzen.

Aus der Abbildung 4.2 kann geschlossen werden, dass DeepKATE mit weiterem Training Ausreißer (`DDoS` Label) nicht zwingend von normalen Instanzen (`BENIGN` Label) im latenten Raum trennt. Bei der Durchführung desselben Aufbaus mit anderen initialen Gewichten des Modells, konnte ein ähnliches Verhalten beobachtet werden. Oft hat der initiale Zustand einen latenten Raum erzeugt, der DDoS-Instanzen besser von normalen Instanzen getrennt hat, als ein latenter Raum nach dem Training von DeepKATE. Als *True Positives* (TP) wurden in diesem Experiment alle Cluster markiert, die lediglich Instanzen einer Klasse nach dem CICIDS2017 Label aufwiesen. Mit *False Positives* (FP) sind hier Clutser gemeint, die Instanzen mehrerer Klassen nach dem CICIDS2017 Label aufwiesen.



(a) Epoche 0

(b) Epoche 1

(c) Epoche 10

(d) Epoche 30

ABBILDUNG 4.2: Einbettung von `Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv` zwischen 0 und 30 Epochen mittels DeepKATE. Die Legende ist, wie folgt zu Interpretieren: Anzahl der Cluster (Nc), korrekte Klassifikation (TP), falsche Klassifikation (FP), Verhältnis von falsch zu korrekt klassifizierten Instanzen (FP/TP), Verhältnis von falsch klassifizierten Instanzen zur Anzahl der Cluster (FP/Nc), Verhältnis von falsch klassifizierten Instanzen zur Anzahl von Ausreißer Instanzen ($FP/Nout$).

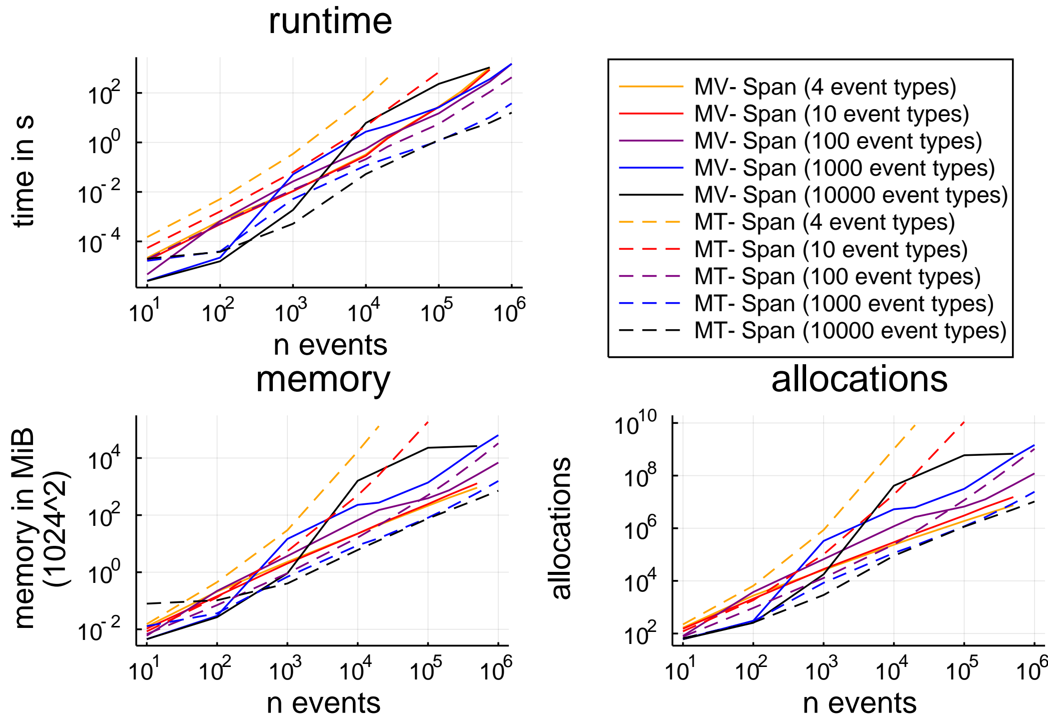


ABBILDUNG 4.3: Vergleich zwischen MV-Span und MT-Span durch synthetische Datensätze mit e gleichverteilten Eventtypen über n Events. Parameter (MV-Span und MT-Span): $min_sup = 2$, $min_utility = 0.0$, $max_reps = 2$, $max_gap = 0$, $max_time_duration = 20$. Beide Skalen sind jeweils logarithmisch dargestellt.

4.2 Episode Mining

Um die beiden erstellten Algorithmen MV-Span und MT-Span miteinander zu vergleichen, wurden synthetische Datensätze erstellt. Ein synthetischer Datensatz besteht dabei aus e gleichverteilten Eventtypen über n Events.

Experiment 1

Folgende Abbildung 4.3 zeigt die Laufzeit (*runtime*), Speichernutzung (*memory*) und Anzahl an Allokationen (*allocations*) für MV-Span und MT-Span. Das Experiment wurde mit folgenden Parameter (MV-Span und MT-Span) durchgeführt: $min_sup = 2$, $min_utility = 0.0$, $max_reps = 2$, $max_gap = 0$, $max_time_duration = 20$. Die externe Nützlichkeit wurde für jedes Event auf 1 gesetzt, da MT-Span die Gesamtnützlichkeit (*totalutility*) einer Sequenz in berücksichtigt und diese nicht 0 sein darf.

MT-Span kann im Gegensatz zu MV-Span besser lange Sequenzen verarbeiten.
Das Experiment zeigt mehrere Sachverhalte auf. Zum einen die lange Laufzeit von MT-Span für nur wenige Eventtypen (z.B. 4). Zum anderen die exponentielle Speichernutzung von MT-Span. Ein weiterer Punkt ist, dass für viele Eventypen (z.B. 1000) MV-Span zwar weniger Speicher benötigt, jedoch eine längere Laufzeit, als MT-Span aufweist.

Kapitel 5

Zusammenfassung

In der Arbeit wurden die drei Themengebiete des Clustering mittels Autoencoder, des Episoden Minings mittels High-Utility-Episode-Mining (HUEM) Algorithmen und der Voraussage von nächsten Events in Sequenzen mittels LSTMs bearbeitet. Ziel der Arbeit war es sowohl einzelne Ausreißer-Events in Event-Logs, als auch ungewöhnliche Abfolgen von Events aufzufinden. Wobei es galt Event-Logs aus verschiedenen Quellen vergleichbar zu machen.

Folgende Probleme konnten durch die erarbeiteten Modelle und Heuristiken gelöst werden:

- Semantische Einbettung hochdimesionaler Events in niedrigdimensionale latente Räume mittels der Autoencoder Architektur DeepKATE ist möglich.
- Robustes Clustering mittels dichtebasiertem Verfahren (DBSCAN) konnte erfolgreich auf die Einbettungen von DeepKATE angewandt werden.
- Ausreißererkennung von Instanzen mittels Rekonstruktionsfehler ist möglich. Es erfordert ein Verfahren, um unüberwachtes Lernen in semi-überwachtes zu überführen.
- Mittels *constrained Episode Mining* konnte gezeigt werden, dass zusätzliche Kriterien dabei helfen repetitive Muster zu ignorieren und dadurch mehr Muster mit hohem Informationsgehalt gefunden werden.

Folgende Fragestellungen konnten nicht beantwortet werden:

- Ob das vorgestellte Modell DeepKATE sich als allgemeines Dimensionreduktions Modell eignet, bleibt offen und gilt es weiter zu untersuchen.
- Einbettungen von DeepKATE als Eingabe für rekurrente Netze zu nutzen wurde nicht untersucht. Eine Repräsentation von Events durch Einbettungen, statt dünn-besetzter Kodierungsverfahren, hat den Vorteil einer einheitlichen Eingabegröße für Neuronale Netze.

5.1 Fazit

In der Arbeit wurden verschiedene Techniken und Methoden angewandt. Die Heterogenität der genutzten Systeme war dabei ein großes Hindernis. Zum einen, da diese keine einheitlichen Schnittstellen bieten und zum anderen, da die Interprozess-Kommunikation ein bisher nicht ausreichend gelöstes Problem darstellt. An vielen stellen schaffen *Wrapper*-Bibliotheken Abhilfe, um die Interprozess-Kommunikation zu vereinfachen. Die Programmiersprache Julia als Hostsprache zu nutzen hat Vor-

und Nachteile. Julia bietet *Wrapper*-Bibliotheken für andere Sprachen, wie etwa die *PyCall*- oder *JavaCall*-Bibliothek an. Bisher wenig genutzte Konzepte für „Skriptsprachen“, wie *Multiple-Dispatch*, sowie *Makros* ermöglichen übersichtlichen und leicht verständlichen Programmcode. Des Weiteren gilt, dass mit der Julia Version 0.7 eingeführten Paketmanagement System, auf einfache Weise reproduzierbare Umgebungen geschaffen werden können, was eine transparente und nachvollziehbare Forschung ermöglicht. Mir war es beispielsweise nicht möglich die nötige Umgebung in Python für KATE[1] soweit herzustellen, dass es möglich gewesen wäre die Ergebnisse von [13] zu reproduzieren. Grund dafür waren fehlende Angaben der genutzten Paketversionen. Nachteil von Julia ist, dass bisher keine *Executables* erstellt werden können. Ein Anwender muss daher vor der Nutzung Julia-Skripte und -Bibliotheken selber kompilieren, was einen nicht unerheblichen Zeitaufwand beim iterativen Entwurf von Prototypen einnehmen kann. Dieser Umstand gilt natürlich für alle kompilierten Sprachen, allerdings fällt er besonders ins Auge, wenn der Versuch unternommen wird Julia, statt interpretierter Sprachen zu nutzen.

Durch die Erstellung des rekursiven Parsers, wie in Abschnitt 3.2.2 beschrieben, können Events aus Log-Dateien in ein einheitliches Format für *Event-Logs* überführt werden. Der Parser überführt Log-Zeilen in Log-Keys und Log-Attribute, wobei die Log-Keys bereits natürliche Cluster darstellen. Durch Benutzerdefinierte Primitiven (*Reguläre Ausdrücke*) können Log-Zeilen deterministisch in generische Muster überführt werden. Vorteil des vorgestellten Parsers ist, dass dieser einfach für *Streaming*-Umgebungen übernommen werden kann. Nachteil des Ansatzes ist, dass dieser stark vom Nutzer abhängt. Daher nehme ich an, dass erfahrenere Analysten von Log-Dateien eher dazu in der Lage sind, geeignete Primitiven zu definieren. Um dieses Problem zu verringern, wurden einige Primitiven vordefiniert. Um einen geeigneten Katalog von Primitiven zu erstellen bedarf es einer quantitativen Untersuchung von häufig genutzten Primitiven verschiedener Systeme.

Durch die Nutzung eines noch jungen Frameworks, wie *Flux.jl* [30] für die Modelle Neuronaler Netze war ich mit mehreren Problemen konfrontiert. Zum einen bietet die Bibliothek keinen einheitlichen Weg Modelle zu erstellen und zu testen. Zum anderen leiden auch andere Frameworks darunter, dass Architekturen Neuronaler Netze und trainierter Netze nicht einfach transferiert werden können. Projekte, wie *ONNX* sollen dabei Helfen die Interoperabilität Neuronaler Netze zwischen verschiedenen Frameworks herzustellen, werden oft jedoch nur teilweise unterstützt. So ist es z.B. in *Flux.jl* möglich ONNX Modelle zu laden, jedoch eigene Modelle nicht als *ONNX* zu exportieren. Es konnte jedoch gezeigt werden, dass auch ungewöhnliche Zielfunktionen (siehe 3.8) mittels *Flux.jl* umgesetzt werden können. In seiner Flexibilität, was den Entwurf von Architekturen Neuronaler Netze betrifft, eignet sich das Framework daher durch die Nutzung der AD, neuartige Architekturen (wie DeepKATE) umzusetzen oder andere zu reproduzieren (z.B. KATE, LSTM, Peephole-LSTM, Bi-LSTM). Dabei konnte im Falle von rekurrenten Netzen mittels *Flux.jl* jedoch keine Hardwarebeschleunigung mittels GPUs durch das vorgestellte MapReduce-Verfahren genutzt werden. Eine Erweiterung der vorgestellten Architektur für bidirektionale rekurrente Netze, sodass die Layer tatsächlich parallel und hardwarebeschleunigt berechnet werden können, könnte ein interessantes Projekt sein, von dem alle Nutzer des *Flux.jl* Frameworks profitieren würden.

[1] `https://github.com/hugochan/KATE`

Wie Dimensionsreduktionstechniken objektiv zu bewerten sind, bleibt eine offene Frage. Der Ansatz die Leistungsfähigkeit indirekt über *Clustering Validity Indices* bewerten zu wollen, war so wie hier in der Arbeit durchgeführt nicht aussagekräftig. Nichtsdestotrotz konnte das DeepKATE-Modell gute Ergebnisse bei der Einbettung von Events eines *Event-Logs* liefern. Eine erkennbare Eigenschaft von Deep-KATE ist natürliche Gruppen zu bilden und solche Gruppen sphärisch anzuordnen (konvexe Cluster). Dabei können jedoch erhebliche Überschneidungen von Instanzen unterschiedlicher Klassen auftreten, wodurch es nötig ist Clusteringverfahren auf den latenten Raum anzuwenden, die gegenüber Ausreißern unempfindlich sind. Es hat sich gezeigt, dass dichtebasierte Verfahren, wie etwa DBSCAN dazu geeignet sind. Das klassische Verfahren des k-means hingegen, eignet sich nicht, um den erzeugten latenten Raum von Deep-KATE so zu clustern, dass eine hohe Homogenität und Vollständigkeit erreicht wird. Um die gruppierenden Eigenschaften des Verfahrens zu verbessern, könnte folgendes ausprobiert werden: Statt unähnliche Instanzen des latenten Raums lediglich voneinander zu trennen, könnten versucht werden, ähnliche Instanzen mittels einer geeigneten Zielfunktion zusätzlich anzunähern.

Die Inferenz von Log-Keys aus Clustern des vorgestellten naiven Ansatzes (siehe Algorithmus 3.10) liefert gute qualitative Ergebnisse, solange keine Verschiebung gleicher Token in den Log-Key Instanzen auftreten. Die Erweiterung des Verfahrens um weitere Strategien, zum Gruppieren benachbarter gleicher Token ist erforderlich um *Log-Keys* mit Quantoren zu erstellen. Ich sehe dabei die Möglichkeit Token, die nicht in allen Instanzen eines Clusters auftreten, über einen Schwellwert zu filtern und somit generische Log-Keys mit Quantoren generieren zu können.

Episoden Mining zu betreiben, um anschließend daraus Episode Rules zu generieren ist ineffizient. Statt Episode Mining zu betreiben, um daraus *Episode Rules* zu generieren, bieten Verfahren des Episode Rule Mining, wie durch MIPER (Mining Precise-positioning Episode Rules) [4] oder UBER-Mine (Utility-Based Episode Rules) [41], das Potenzial bei gleicher Laufzeit mehr Muster aufzufinden. Insbesondere wurden in dieser Arbeit die Auswirkungen durch die Nutzung von Hash-Tables, statt Tries nicht berücksichtigt. Durch den Gebrauch angepasster Datenstrukturen für den Zweck des Episode Rule Mining sind weitere Erfolge zu erwarten. Die Verbindung von MIPER mit weiteren Kriterien wäre aus meiner Sicht ein interessanter Forschungsgegenstand - z.B. als High utility Episode Rule Mining (HUERM).

Literatur

[1] *16.6. logging — Logging facility for Python — Python 3.7.0 documentation*. URL: https://docs.python.org/3/library/logging.html#logging-levels (besucht am 03. 08. 2018).

[2] Martín Abadi, Ashish Agarwal, Paul Barham und et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[3] An, Jinwon; Cho, Sungzoon. „Variational Autoencoder based Anomaly Detection using Reconstruction Probability“. SNU Data Mining Center. Dez. 2015. URL: http://dm.snu.ac.kr/static/docs/TR/SNUDM-TR-2015-03.pdf (besucht am 28. 06. 2018).

[4] X. Ao u. a. „Mining Precise-Positioning Episode Rules from Event Sequences“. In: *IEEE Transactions on Knowledge and Data Engineering* 30.3 (März 2018), S. 530–543. ISSN: 1041-4347. DOI: 10.1109/TKDE.2017.2773493.

[5] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 1. O'Reilly Media, März 2017. ISBN: 978-1-4919-6229-9.

[6] Lei Jimmy Ba und Rich Caurana. „Do Deep Nets Really Need to be Deep?“ In: *CoRR* abs/1312.6184 (2013). arXiv: 1312.6184. URL: http://arxiv.org/abs/1312.6184.

[7] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul und Jeffrey Mark Siskind. „Automatic Differentiation in Machine Learning: a Survey“. In: *Journal of Machine Learning Research* 18.153 (2018), S. 1–43. URL: http://jmlr.org/papers/v18/17-468.html (besucht am 01. 08. 2018).

[8] Hila Becker. „A Survey of Correlation Clustering“. en. In: (Mai 2005), S. 10.

[9] Christopher Bishop. *Pattern Recognition and Machine Learning*. en. Information Science and Statistics, Springer Nature Textbooks – HE site. New York: Springer-Verlag, 2006. ISBN: 978-0-387-31073-2. URL: //www.springer.com/us/book/9780387310732 (besucht am 30. 07. 2018).

[10] Matthew B. Blaschko und Christoph H. Lampert. „Correlational spectral clustering“. en. In: IEEE, Juni 2008, S. 1–8. ISBN: 978-1-4244-2242-5. DOI: 10.1109/CVPR.2008.4587353. URL: http://ieeexplore.ieee.org/document/4587353/ (besucht am 09. 08. 2018).

[11] Piotr Bojanowski, Edouard Grave, Armand Joulin und Tomas Mikolov. „Enriching Word Vectors with Subword Information“. In: *arXiv preprint arXiv:1607.04606* (2016).

[12] Peter F. Brown u. a. „Class-Based n-gram Models of Natural Language“. In: *Computational Linguistics* 18.4 (1992), S. 467–479.

[13] Y. Chen und M. J. Zaki. „KATE: K-Competitive Autoencoder for Text“. In: *ArXiv e-prints* (Mai 2017). arXiv: 1705.02033 [stat.ML].

[14] François Chollet u. a. *Keras*. https://keras.io. 2015.

[15] G. Cybenko. „Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (Dez. 1989), S. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[16] Rajarshi Das, Manzil Zaheer und Chris Dyer. „Gaussian LDA for Topic Models with Word Embeddings". In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, 2015. URL: http://www.aclweb.org/anthology/P15-1077.

[17] Scott Deerwester u. a. „Indexing by latent semantic analysis". In: *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE* 41.6 (1990), S. 391–407.

[18] Leon Derczynski, Sean Chester und Kenneth S. Bøgh. „Tune Your Brown Clustering, Please". In: *Recent Advances in Natural Language Processing, RANLP 2015, 7-9 September, 2015, Hissar, Bulgaria*. 2015, S. 110–117. URL: http://aclweb.org/anthology/R/R15/R15-1016.pdf.

[19] M. Du und F. Li. „Spell: Streaming Parsing of System Event Logs". In: *2016 IEEE 16th International Conference on Data Mining (ICDM)*. Dez. 2016, S. 859–864. DOI: 10.1109/ICDM.2016.0103.

[20] Min Du, Feifei Li, Guineng Zheng und Vivek Srikumar. „DeepLog: Anomaly Detection and Diagnosis from System Logs Through Deep Learning". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: ACM, 2017, S. 1285–1298. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134015. URL: http://doi.acm.org/10.1145/3133956.3134015 (besucht am 02. 08. 2018).

[21] Martin Ester, Hans-Peter Kriegel, Jörg Sander und Xiaowei Xu. „A density-based algorithm for discovering clusters in large spatial databases with noise". In: AAAI Press, 1996, S. 226–231.

[22] Felix A. Gers und Jürgen Schmidhuber. „Recurrent Nets that Time and Count". In: *IJCNN (3)*. 2000, S. 189–194. DOI: 10.1109/IJCNN.2000.861302. URL: https://doi.org/10.1109/IJCNN.2000.861302.

[23] Xavier Glorot, Antoine Bordes und Yoshua Bengio. „Deep Sparse Rectifier Neural Networks". en. In: (2011), S. 9.

[24] Andreas Griewank. „Who Invented the Reverse Mode of Differentiation?" In: *Documenta Matematica* Extra Volume ISMP (2012) (2012), S. 389–400. URL: https://www.math.uni-bielefeld.de/documenta/vol-ismp/52_griewank-andreas-b.pdf (besucht am 01. 08. 2018).

[25] Guangming Guo u. a. „High Utility Episode Mining Made Practical and Fast". In: *Advanced Data Mining and Applications - 10th International Conference, ADMA 2014, Guilin, China, December 19-21, 2014. Proceedings*. 2014, S. 71–84. DOI: 10.1007/978-3-319-14717-8_6. URL: https://doi.org/10.1007/978-3-319-14717-8_6.

[26] M. Halkidi und M. Vazirgiannis. „Clustering validity assessment: finding the optimal partitioning of a data set". In: *Proceedings 2001 IEEE International Conference on Data Mining*. Nov. 2001, S. 187–194. DOI: 10.1109/ICDM.2001.989517.

[27] Sepp Hochreiter und Jürgen Schmidhuber. „Long Short-Term Memory“. In: *Neural Comput.* 9.8 (Nov. 1997), S. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735 (besucht am 02. 08. 2018).

[28] Kurt Hornik. „Approximation capabilities of multilayer feedforward networks“. In: *Neural Networks* 4.2 (Jan. 1991), S. 251–257. ISSN: 0893-6080. DOI: 10.1016/0893-6080(91)90009-T. URL: http://www.sciencedirect.com/science/article/pii/089360809190009T (besucht am 12. 07. 2018).

[29] Lawrence Hubert und Phipps Arabie. „Comparing partitions“. en. In: *Journal of Classification* 2.1 (Dez. 1985), S. 193–218. ISSN: 0176-4268, 1432-1343. DOI: 10.1007/BF01908075. URL: https://link.springer.com/article/10.1007/BF01908075 (besucht am 18. 08. 2018).

[30] Mike Innes. „Flux: Elegant machine learning with Julia“. In: *Journal of Open Source Software* 3.25 (Mai 2018), S. 602. ISSN: 2475-9066. DOI: 10.21105/joss.00602. URL: http://joss.theoj.org/papers/10.21105/joss.00602 (besucht am 01. 08. 2018).

[31] Armand Joulin, Edouard Grave, Piotr Bojanowski und Tomas Mikolov. „Bag of Tricks for Efficient Text Classification“. In: *arXiv preprint arXiv:1607.01759* (2016).

[32] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. Mai 2015. URL: https://karpathy.github.io/2015/05/21/rnn-effectiveness/ (besucht am 12. 07. 2018).

[33] D. P Kingma und M. Welling. „Auto-Encoding Variational Bayes“. In: *ArXiv e-prints* (Dez. 2013). arXiv: 1312.6114 [stat.ML].

[34] Diederik P. Kingma und Jimmy Ba. „Adam: A Method for Stochastic Optimization“. In: *arXiv:1412.6980 [cs]* (Dez. 2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980 (besucht am 03. 02. 2018).

[35] *Kullback-Leibler information - Encyclopedia of Mathematics*. URL: https://www.encyclopediaofmath.org/index.php/Kullback%e2%80%93Leibler_information (besucht am 01. 10. 2018).

[36] Hugo Larochelle und Stanislas Lauly. „A Neural Autoregressive Topic Model“. In: *Advances in Neural Information Processing Systems 25*. Hrsg. von F. Pereira, C. J. C. Burges, L. Bottou und K. Q. Weinberger. Curran Associates, Inc., 2012, S. 2708–2716. URL: http://papers.nips.cc/paper/4613-a-neural-autoregressive-topic-model.pdf.

[37] Arash Habibi Lashkari, Gerard Draper Gil, Mohammad Saiful Islam Mamun und Ali A. Ghorbani. „Characterization of Tor Traffic using Time based Features“. In: *Proceedings of the 3rd International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,* INSTICC. SciTePress, 2017, S. 253–262. ISBN: 978-989-758-209-7. DOI: 10.5220/0006105602530262.

[38] Quoc V. Le und Tomas Mikolov. „Distributed Representations of Sentences and Documents“. In: *CoRR* abs/1405.4053 (2014). arXiv: 1405.4053. URL: http://arxiv.org/abs/1405.4053.

[39] *Level (Apache Log4j 1.2.17 API)*. 2012. URL: https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Level.html (besucht am 03. 08. 2018).

[40] Dahua Lin und et al. *Silhouettes — Clustering 0.3.0 documentation*. URL: `https://clusteringjl.readthedocs.io/en/latest/silhouette.html` (besucht am 01. 10. 2018).

[41] Yu-Feng Lin, Cheng-Wei Wu, Chien-Feng Huang und Vincent S. Tseng. „Discovering utility-based episode rules in complex event sequences“. In: *Expert Systems with Applications* 42.12 (2015), S. 5303 –5314. ISSN: 0957-4174. DOI: `https://doi.org/10.1016/j.eswa.2015.02.022`. URL: `http://www.sciencedirect.com/science/article/pii/S0957417415001219`.

[42] Yanchi Liu u. a. „Understanding of Internal Clustering Validation Measures“. en. In: IEEE, Dez. 2010, S. 911–916. ISBN: 978-1-4244-9131-5. DOI: `10.1109/ICDM.2010.35`. URL: `http://ieeexplore.ieee.org/document/5694060/` (besucht am 08. 08. 2018).

[43] S. Lloyd. „Least squares quantization in PCM“. In: *IEEE Transactions on Information Theory* 28.2 (März 1982), S. 129–137. ISSN: 0018-9448. DOI: `10.1109/TIT.1982.1056489`.

[44] David M. Blei, Andrew Y. Ng, Michael Jordan und John Lafferty. „Latent Dirichlet Allocation“. In: *Journal of Machine Learning Research 3 (2003) 993-1022 Submitted 2/02; Published 1/03* (Feb. 2003), S. 29.

[45] Lars Maaloe, Morten Arngren und Ole Winther. „Deep Belief Nets for Topic Modeling“. In: *CoRR* abs/1501.04325 (2015). arXiv: `1501.04325`. URL: `http://arxiv.org/abs/1501.04325`.

[46] L.J.P van der Maaten und G.E. Hinton. „Visualizing High-Dimensional Data Using t-SNE“. In: *Journal of Machine Learning Research* 9: 2579–2605 (Nov. 2008).

[47] Alireza Makhzani und Brendan J. Frey. „A Winner-Take-All Method for Training Sparse Convolutional Autoencoders“. In: *CoRR* abs/1409.2752 (2014). arXiv: `1409.2752`. URL: `http://arxiv.org/abs/1409.2752`.

[48] Alireza Makhzani und Brendan J. Frey. „k-Sparse Autoencoders“. In: *CoRR* abs/1312.5663 (2013). arXiv: `1312.5663`. URL: `http://arxiv.org/abs/1312.5663`.

[49] Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly und Ian J. Goodfellow. „Adversarial Autoencoders“. In: *CoRR* abs/1511.05644 (2015). arXiv: `1511.05644`. URL: `http://arxiv.org/abs/1511.05644`.

[50] Heikki Mannila, Hannu Toivonen und A. Inkeri Verkamo. „Discovering Frequent Episodes in Sequences“. In: *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95), Montreal, Canada, August 20-21, 1995*. 1995, S. 210–215. URL: `http://www.aaai.org/Library/KDD/1995/kdd95-024.php`.

[51] Heikki Mannila, Hannu Toivonen und A. Inkeri Verkamo. „Discovery of Frequent Episodes in Event Sequences“. In: *Data Min. Knowl. Discov.* 1.3 (1997), S. 259–289. DOI: `10.1023/A:1009748302351`. URL: `https://doi.org/10.1023/A:1009748302351`.

[52] Leland McInnes und John Healy. „UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction“. In: *arXiv:1802.03426 [cs, stat]* (Feb. 2018). arXiv: 1802.03426. URL: `http://arxiv.org/abs/1802.03426` (besucht am 07. 03. 2018).

[53] Marina Meilă. „Comparing Clusterings by the Variation of Information“. In: *Learning Theory and Kernel Machines*. Hrsg. von Bernhard Schölkopf und Manfred K. Warmuth. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, S. 173–187. ISBN: 978-3-540-45167-9.

[54] Marina Meilă. „Comparing clusterings—an information based distance“. In: *Journal of Multivariate Analysis* 98.5 (Mai 2007), S. 873–895. ISSN: 0047-259X. DOI: 10.1016/j.jmva.2006.11.013. URL: http://www.sciencedirect.com/science/article/pii/S0047259X06002016 (besucht am 09. 08. 2018).

[55] Yishu Miao, Lei Yu und Phil Blunsom. „Neural Variational Inference for Text Processing“. In: *CoRR* abs/1511.06038 (2015). arXiv: 1511.06038. URL: http://arxiv.org/abs/1511.06038.

[56] Michael Collins. „Computational Graphs, and Backpropagation“. Dez. 2017. URL: http://www.cs.columbia.edu/~mcollins/ff2.pdf (besucht am 10. 09. 2018).

[57] Tomas Mikolov, Kai Chen, Greg Corrado und Jeffrey Dean. „Efficient Estimation of Word Representations in Vector Space“. In: *CoRR* abs/1301.3781 (2013). arXiv: 1301.3781. URL: http://arxiv.org/abs/1301.3781.

[58] Hideyuki Ohtani, Takuya Kida, Takeaki Uno und Hiroki Arimura. „Efficient Serial Episode Mining with Minimal Occurrences“. In: *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*. ICUIMC '09. New York, NY, USA: ACM, 2009, S. 457–464. ISBN: 978-1-60558-405-8. DOI: 10.1145/1516241.1516320. URL: http://doi.acm.org/10.1145/1516241.1516320 (besucht am 17. 08. 2018).

[59] o.V. *2.3. Clustering — scikit-learn 0.19.2 documentation*. URL: http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation (besucht am 08. 08. 2018).

[60] o.V. *2.3. Clustering — scikit-learn 0.19.2 documentation*. URL: http://scikit-learn.org/stable/modules/clustering.html (besucht am 08. 08. 2018).

[61] o.V. *A Julia package for data clustering*. original-date: 2012-11-24T15:47:36Z. Sep. 2018. URL: https://github.com/JuliaStats/Clustering.jl (besucht am 02. 10. 2018).

[62] Hamid Palangi u. a. „Deep Sentence Embedding Using Long Short-Term Memory Networks: Analysis and Application to Information Retrieval“. In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 24.4 (Apr. 2016). arXiv: 1502.06922, S. 694–707. ISSN: 2329-9290, 2329-9304. DOI: 10.1109/TASLP.2016.2520371. URL: http://arxiv.org/abs/1502.06922 (besucht am 31. 05. 2018).

[63] Adam Paszke u. a. „Automatic differentiation in PyTorch“. In: (2017).

[64] F. Pedregosa u. a. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830.

[65] Jeffrey Pennington, Richard Socher und Christopher D. Manning. „GloVe: Global Vectors for Word Representation“. In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, S. 1532–1543. URL: http://www.aclweb.org/anthology/D14-1162.

[66] William M. Rand. „Objective Criteria for the Evaluation of Clustering Methods“. In: *Journal of the American Statistical Association* 66.336 (1971), S. 846–850. DOI: 10.1080/01621459.1971.10482356. eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1971.10482356. URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1971.10482356.

[67] Sonam Rathore, Siddharth Dawar, Vikram Goyal und Dhaval Patel. „Top-K High Utility Episode Mining from a Complex Event Sequence“. In: *21st International Conference on Management of Data, COMAD 2016, Pune, India, March 11-13, 2016*. 2016, S. 56–63. URL: http://comad.in/comad2016/proceedings/paper_19.pdf.

[68] B Rieck und H Leitte. „Exploring and Comparing Clusterings of Multivariate Data Sets Using Persistent Homology: Supplementary materials“. en. In: (2016), S. 8.

[69] Salah Rifai u. a. „Contracting auto-encoders: Explicit invariance during feature extraction“. In: *In Proceedings of the Twenty-eight International Conference on Machine Learning (ICML11*. 2011.

[70] Doreen Seider u. a. „Open Source Software Framework for Applications in Aeronautics and Space“. In: *IEEE Aerospace Conference*. März 2012. URL: https://elib.dlr.de/77442/.

[71] Iman Sharafaldin, Arash Habibi Lashkari und Ali A. Ghorbani. „Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization“. In: *Proceedings of the 4th International Conference on Information Systems Security and Privacy, ICISSP 2018, Funchal, Madeira - Portugal, January 22-24, 2018.* Hrsg. von Paolo Mori, Steven Furnell und Olivier Camp. SciTePress, 2018, S. 108–116. ISBN: 978-989-758-282-0. DOI: 10.5220/0006639801080116. URL: https://doi.org/10.5220/0006639801080116.

[72] *sklearn.metrics.accuracy_score — scikit-learn 0.20.0 documentation*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html# (besucht am 01. 10. 2018).

[73] *sklearn.metrics.f1_score — scikit-learn 0.20.0 documentation*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html (besucht am 01. 10. 2018).

[74] *sklearn.metrics.v_measure_score — scikit-learn 0.20.0 documentation*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.metrics.v_measure_score.html (besucht am 01. 10. 2018).

[75] R. Vaarandi. „A data clustering algorithm for mining patterns from event logs“. In: *Proceedings of the 3rd IEEE Workshop on IP Operations Management (IPOM 2003) (IEEE Cat. No.03EX764)*. Okt. 2003, S. 119–126. DOI: 10.1109/IPOM.2003.1251233.

[76] Risto Vaarandi, Markus Kont und Mauno Pihelgas. „Event log analysis with the LogCluster tool“. In: *2016 IEEE Military Communications Conference, MILCOM 2016, Baltimore, MD, USA, November 1-3, 2016*. 2016, S. 982–987. DOI: 10.1109/MILCOM.2016.7795458. URL: https://doi.org/10.1109/MILCOM.2016.7795458.

[77] Risto Vaarandi und Mauno Pihelgas. „LogCluster - A data clustering and pattern mining algorithm for event logs“. In: *11th International Conference on Network and Service Management, CNSM 2015, Barcelona, Spain, November 9-13, 2015*. 2015, S. 1–7. DOI: 10.1109/CNSM.2015.7367331. URL: https://doi.org/10.1109/CNSM.2015.7367331.

[78] Ashish Vaswani u. a. „Attention Is All You Need“. In: *arXiv:1706.03762 [cs]* (Juni 2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762 (besucht am 01. 06. 2018).

[79] Pascal Vincent u. a. „Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion“. In: *Journal of Machine Learning Research* 11 (2010), S. 3371–3408. URL: http://portal.acm.org/citation.cfm?id=1953039.

[80] Nguyen Xuan Vinh, Julien Epps und James Bailey. „Information Theoretic Measures for Clusterings Comparison: Is a Correction for Chance Necessary?“ In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML '09. New York, NY, USA: ACM, 2009, S. 1073–1080. ISBN: 978-1-60558-516-1. DOI: 10.1145/1553374.1553511. URL: http://doi.acm.org/10.1145/1553374.1553511 (besucht am 09. 08. 2018).

[81] John Wieting, Mohit Bansal, Kevin Gimpel und Karen Livescu. „Towards Universal Paraphrastic Sentence Embeddings“. In: *arXiv:1511.08198 [cs]* (Nov. 2015). arXiv: 1511.08198. URL: http://arxiv.org/abs/1511.08198 (besucht am 31. 05. 2018).

[82] Cheng-Wei Wu, Yu-Feng Lin, Philip S. Yu und Vincent S. Tseng. „Mining high utility episodes in complex event sequences“. In: *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*. 2013, S. 536–544. DOI: 10.1145/2487575.2487654. URL: http://doi.acm.org/10.1145/2487575.2487654.

[83] Junyuan Xie, Ross B. Girshick und Ali Farhadi. „Unsupervised Deep Embedding for Clustering Analysis“. In: *CoRR* abs/1511.06335 (2015). arXiv: 1511.06335. URL: http://arxiv.org/abs/1511.06335.

[84] Junfu Yin, Zhigang Zheng und Longbing Cao. „USpan: an efficient algorithm for mining high utility sequential patterns“. In: *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*. Hrsg. von Qiang Yang, Deepak Agarwal und Jian Pei. ACM, 2012, S. 660–668. ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339636. URL: http://doi.acm.org/10.1145/2339530.2339636 (besucht am 11. 09. 2018).

[85] Deniz Yuret. „Knet: beginning deep learning with 100 lines of Julia“. In: *Machine Learning Systems Workshop at NIPS 2016*. 2016.

[86] Mohammed J. Zaki und Jr. Wagner Meira. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, Mai 2014. ISBN: 9780521766333.

[87] Mohammed Javeed Zaki. „SPADE: An Efficient Algorithm for Mining Frequent Sequences“. In: *Machine Learning* 42.1/2 (2001), S. 31–60. DOI: 10.1023/A:1007652502315. URL: https://doi.org/10.1023/A:1007652502315.

[88] C. Zhuge und R. Vaarandi. „Efficient Event Log Mining with LogClusterC". In: *2017 ieee 3rd international conference on big data security on cloud (bigdatasecurity), ieee international conference on high performance and smart computing (hpsc), and ieee international conference on intelligent data and security (ids)*. Mai 2017, S. 261–266. DOI: 10.1109/BigDataSecurity.2017.26.

DeepKATE

In folgenden Abbildungen sind verschiedene Epochen der Einbettung mittels DeepKATE dargestellt. Die Dimension des latenten Raums kann frei gewählt werden. Es werden beispielhaft Einbettungen des RCE-Datensatzes gezeigt. Dabei handelt es sich um Einbettungen der selben Log-Datei (6073 Log-Zeilen) in einen zwei- und dreidimensionalen latenten Raum. Die Einbettungen wurden mittels DBSCAN in ein Clustering überführt, wobei der Parameter Radius (*epsilon*) variiert wurde. Durch einen größeren Radius werden weniger Cluster gebildet, wie in den Abbildungen zu sehen ist.

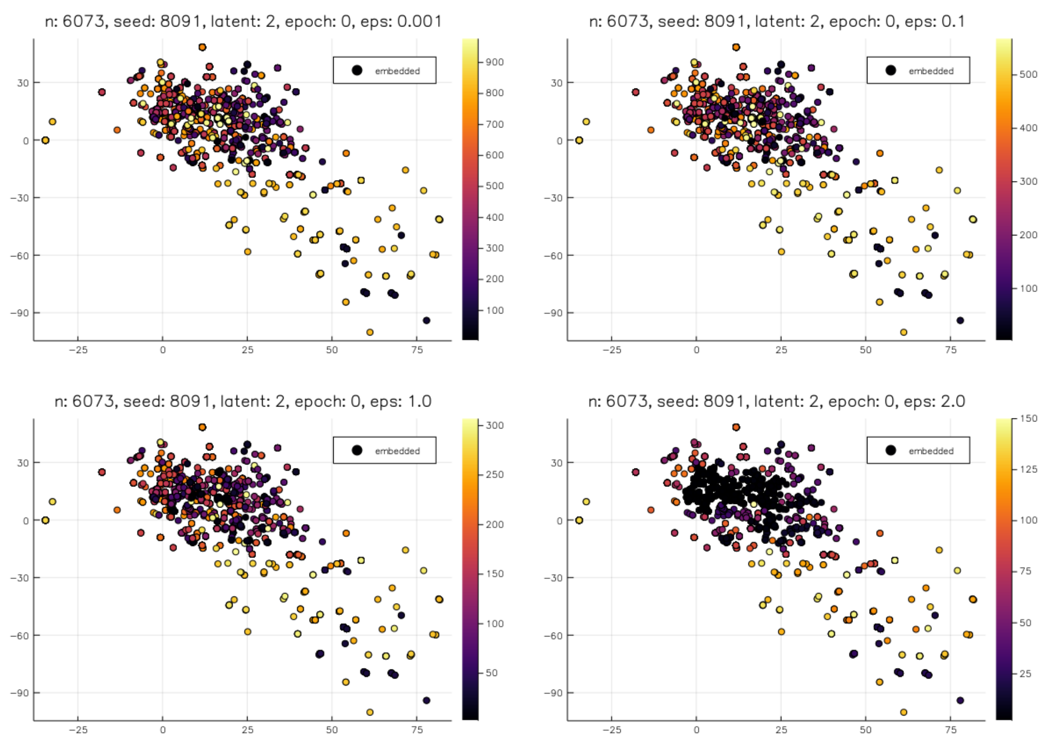


ABBILDUNG 1: 2d Einbettung ohne Training (RCE).

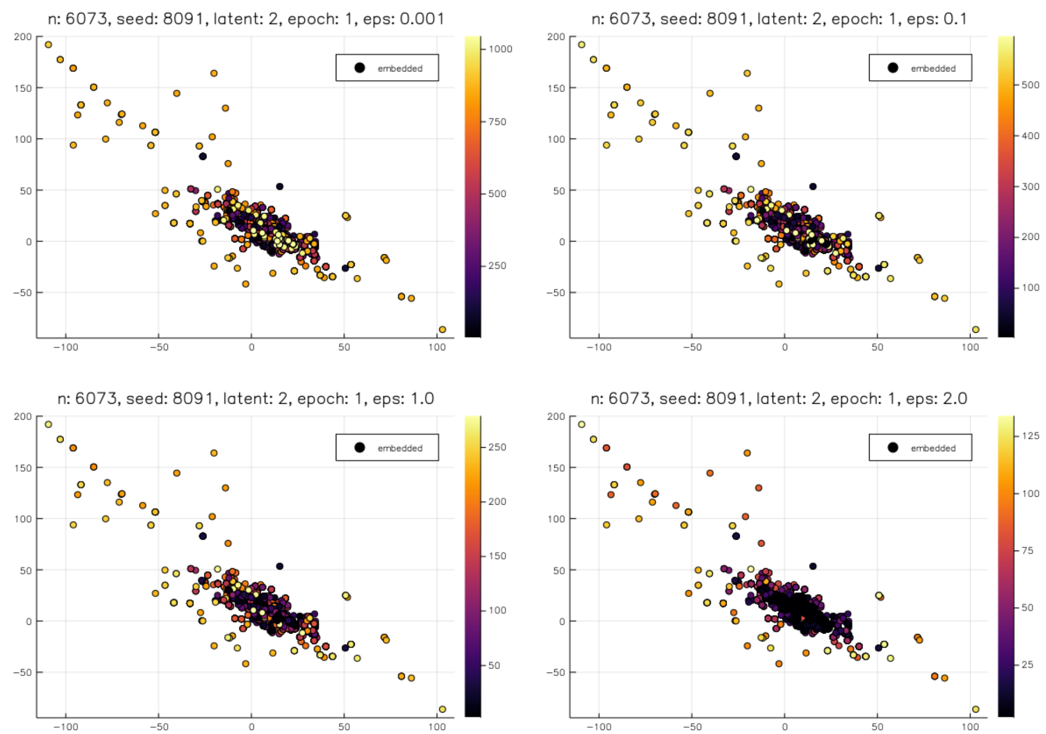


ABBILDUNG 2: 2d Einbettung nach 1 Epoche (RCE).

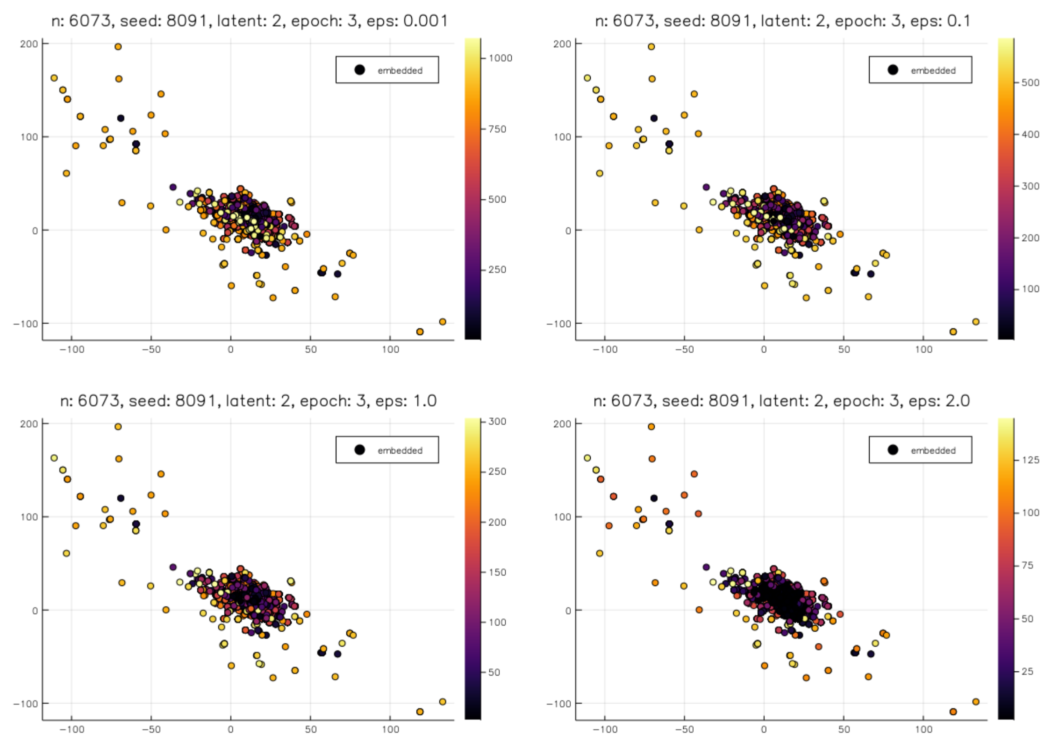


ABBILDUNG 3: 2d Einbettung nach 3 Epochen (RCE).

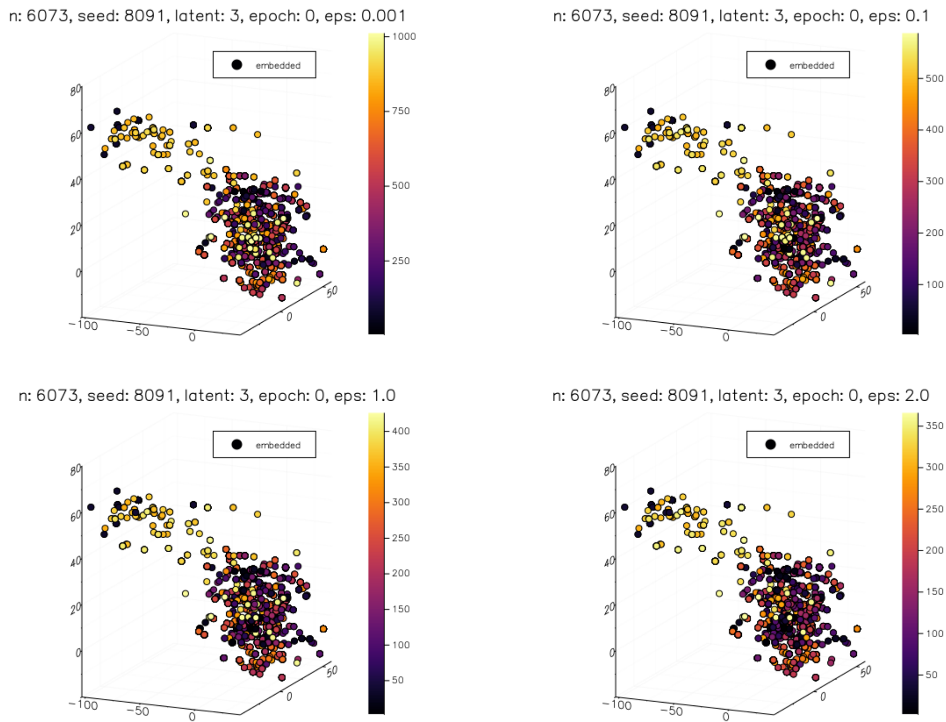


ABBILDUNG 4: 3d Einbettung nach 0 Epochen (kein Training).

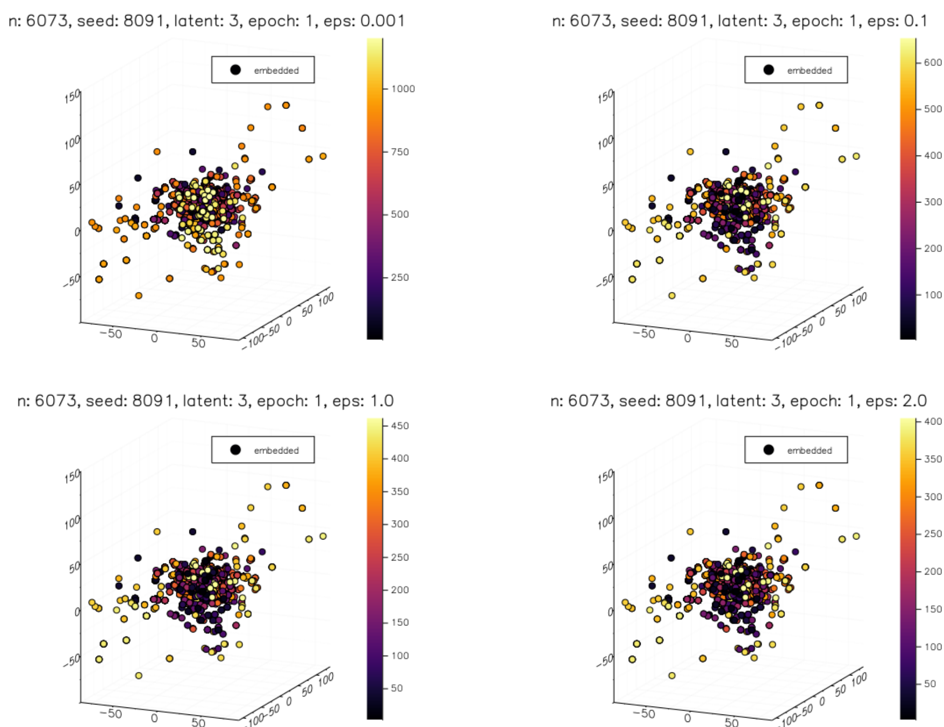


ABBILDUNG 5: 3d Einbettung nach 1 Epoche (RCE).

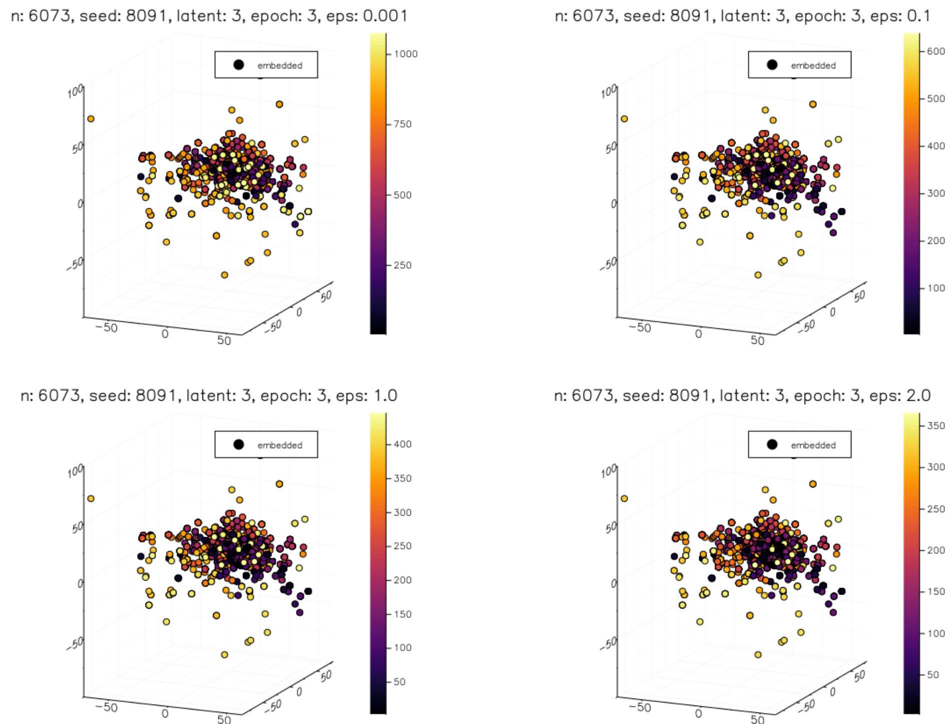


ABBILDUNG 6: 3d Einbettung nach 3 Epochen (RCE).

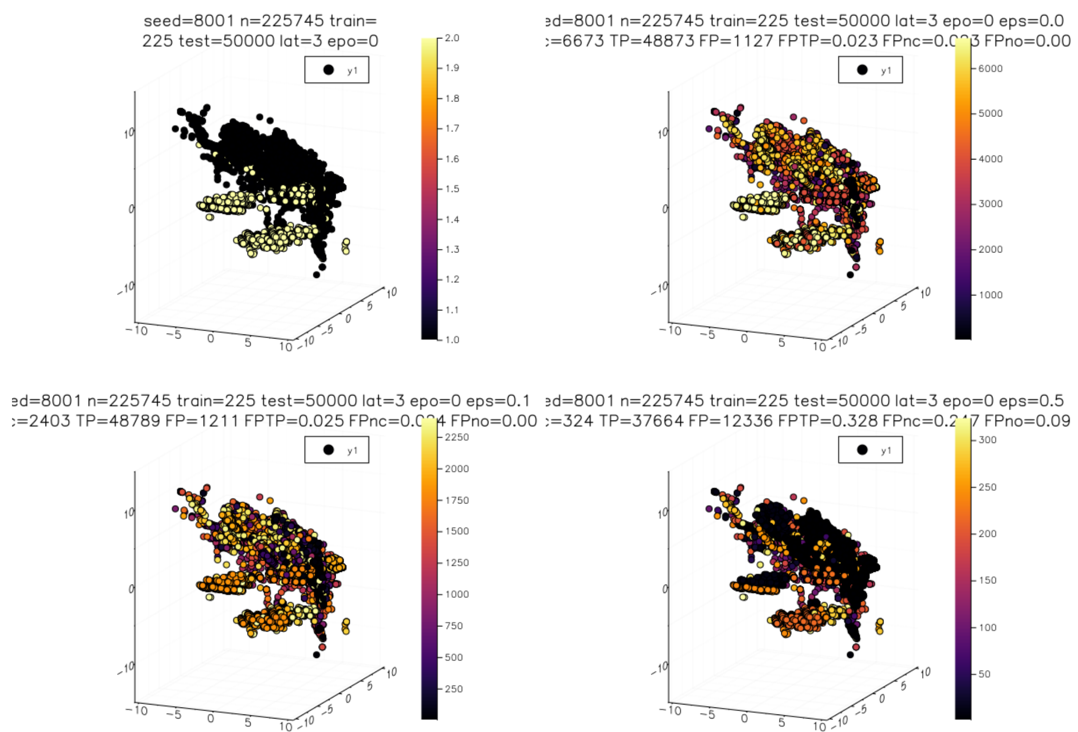


ABBILDUNG 7: Einbettung ohne Training (CICIDS2017).

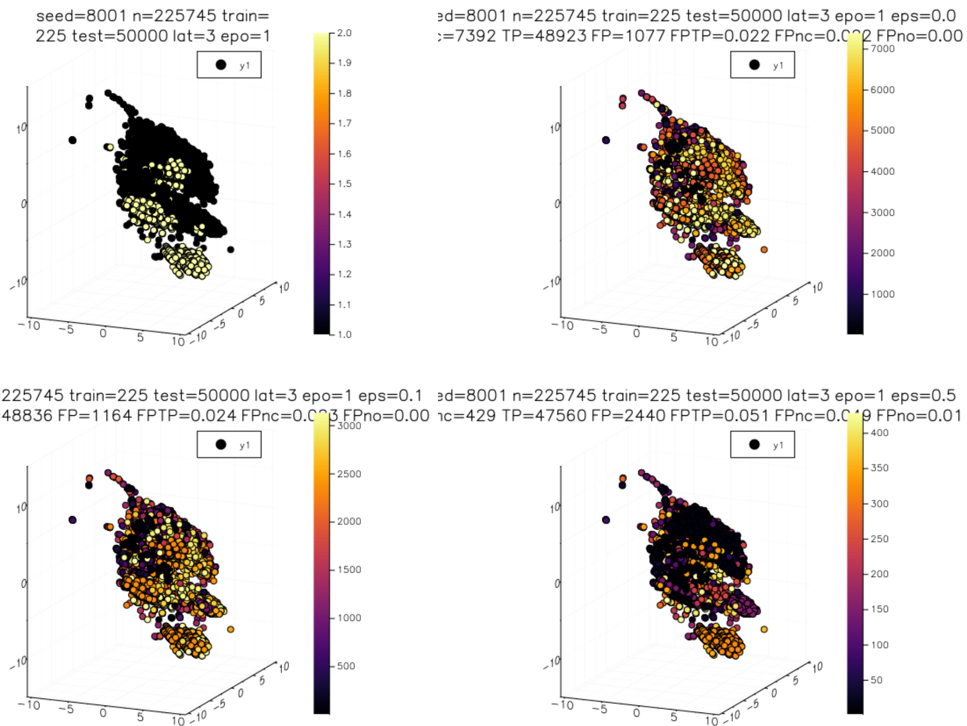


ABBILDUNG 8: Einbettung nach einer Epoche (CICIDS2017).

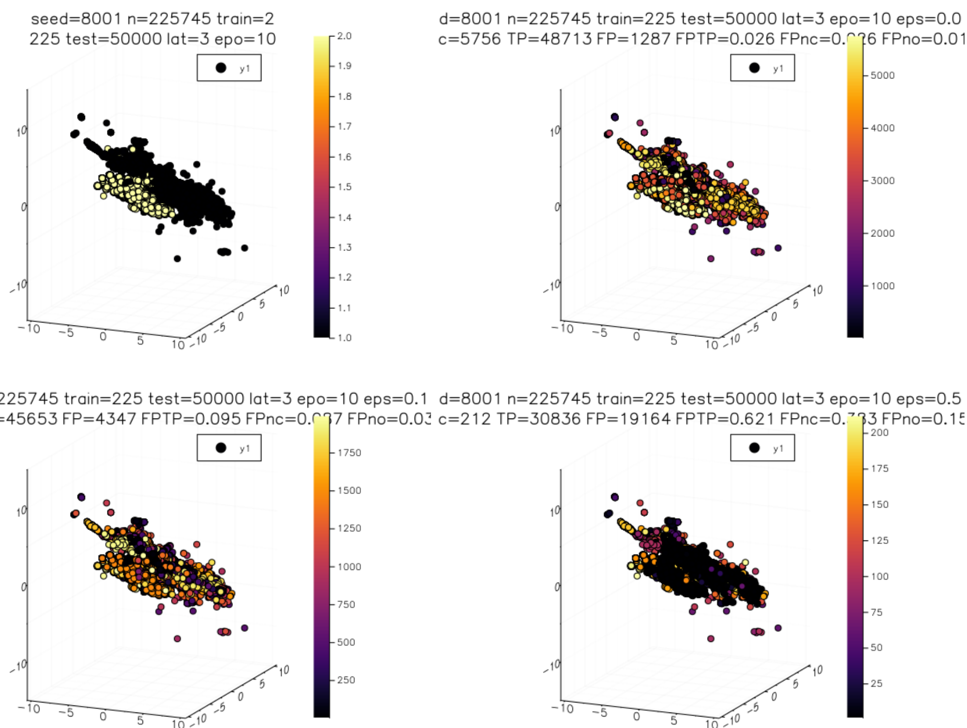


ABBILDUNG 9: Einbettung nach 10 Epochen (CICIDS2017).

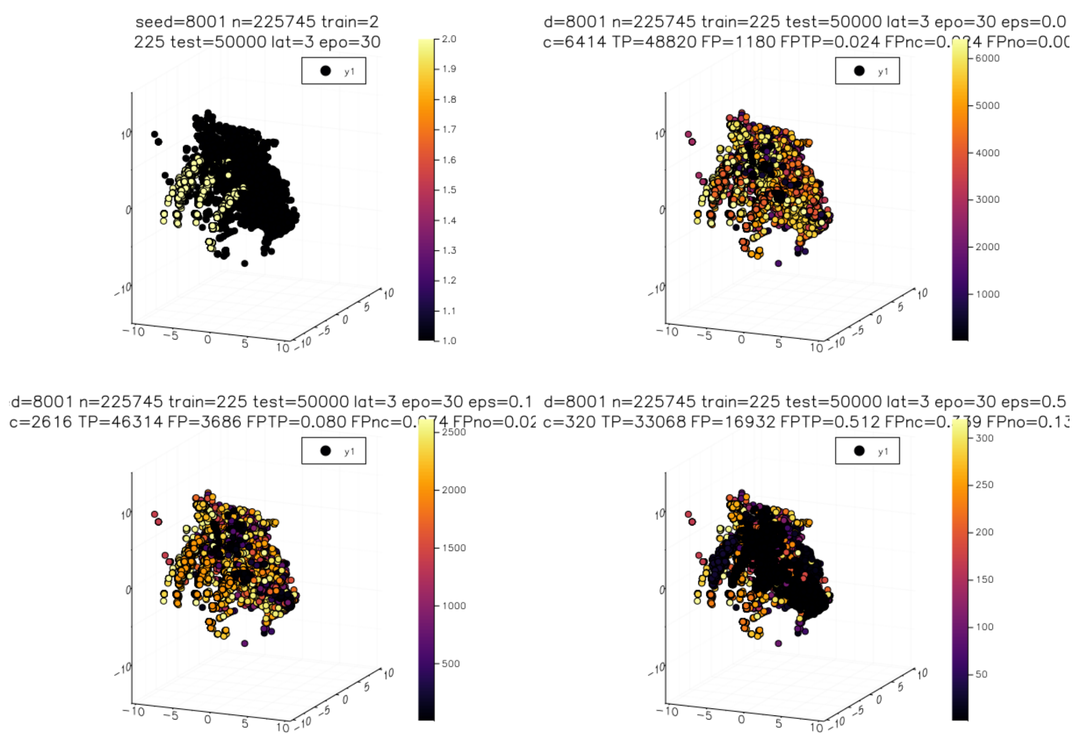


ABBILDUNG 10: Einbettung nach 30 Epochen (CICIDS2017).

Anforderungen

Preprocessing

Template-Preprocessing:
[F-Prep-1] Als Anwender [muss] ich eine Log-Datei durch die Angabe von Templates in einen Event-Log transformieren können, um eine Log-Datei in eine strukturierte Form zu über führen.

Clustering

Clustering / Outlier:

[F-Cluster-1] Als Anwender [muss] ich einen Event-Log in ein Clustering überführen können, um Log-Zeilen (Log-Keys) einen Identifier zuordnen zu können.
[F-Cluster-2a] Als Anwender [muss] ich den Support eines Clusters ermitteln können, um Ausreißer und häufige Klassen zu unterscheiden.
[F-Cluster-2b] Als Anwender [muss] ich ein Clustering nach dem Support sortieren können, um Ausreißer und häufige Klassen zu unterscheiden.
[F-Cluster-2c] Als Anwender [muss] ich ein Clustering nach dem Support filtern können, um Ausreißer und häufige Klassen zu unterscheiden.

Outlier-Kriterien:
[F-Cluster-3] Als Anwender [kann] ich durch Angabe von Bedingungen die Menge der Ausreißer näher bestimmen, um spezifische Log-Keys zu generieren.

Clustering Validierung:
[F-Cluster-1] Als Anwender [muss] ich mittels verschiedener Metriken das Clustering validieren können, um die schwächen einer Metrik durch andere ausgleichen zu können.
[F-Cluster-2a] Als Anwender [kann] ich visuelles Feedback des Clustering durch eine Metrik bekommen, um eine Übersicht zur Varianz eines Event-Logs zu bekommen.
[F-Cluster-2b] Als Anwender [kann] ich visuelles Feedback des Clustering durch die Kombination von Metriken bekommen, um eine Übersicht zur Varianz eines Event-Logs zu bekommen.

Repräsentations-Management

RegExp Generieren (init)
[F-Repr-1] Als Anwender [muss] ich aus der Menge der von Log-Events eines Clusters einen generischen *Regulären Ausdruck* inferieren können, um aus einer

Clusterzuordnung eine menschenlesbare Repräsentation zu erzeugen.

RegExp Speichern (write)
[F-Repr-2] Als Anwender [muss] ich zeitliche Muster eines Event-Logs abspeichern können, um diese in anderen Event-Logs wiedererkennen zu können.

RegExp Lesen-und-Anwenden (read)
[F-Repr-3] Als Anwender [muss] ich zeitliche Muster auf einen Event-Log anwenden können, um wiedererkannte Muster anzuzeigen.

RegExp ändern (update)
[F-Repr-4a] Als Anwender [muss] ich zeitliche Muster ändern können, um eine Repräsentation für einen Anwendungsfall anpassen zu können.
[F-Repr-4b] Als Anwender [muss] ich zeitliche Muster duplizieren und dabei ändern können, um eine Repräsentation für einen Anwendungsfall anpassen zu können.

RegExp löschen (delete)
[F-Repr-5] Als Anwender [muss] ich zeitliche Muster aus der Menge aller gespeicherten Muster entfernen können, um keine unnötigen Muster anzusammeln.

Mustererkennung

[F-Muster-1] Als Anwender [muss] ich Episoden aus den zeitlichen Mustern generieren können, um die Menge der Muster (Frequent Pattern) zu reduzieren und visuell als Graphen aufzubereiten.
[F-Muster-2a] Als Anwender [muss] es mir möglich sein für jede Log-Zeile ihre Cluster-Zuordnung und Repräsentation anzeigen zu lassen, um Verwechslungen bei der Auswahl eines zeitlichen Musters zu vermeiden.
[F-Muster-2b] Als Anwender [kann] ich mir den Identifier und die Frequenz eines Clusters aus der Menge der Cluster für einen Event-Log anzeigen lassen können, um damit der Anwender abschätzen kann, wie häufig ein Cluster ist.

Sourcecode und Ergebnisse

Die im Rahmen der Arbeit erstellte Bibliothek *LogClustering.jl* für die Programmiersprache Julia ist hier in Form eines Speichermediums hinterlegt. Neben dem Sourcecode der Bibliothek sind die Experimente und dessen Ergebnisse ebenfalls hinterlegt, soweit diese keine sensiblen Informationen des RCE-Datensatzes enthalten. Eine Beschreibung der Struktur des Projekts und wie man es nutzen kann, ist in Form einer *README*-Datei im Wurzelverzeichnis des Projekts zu finden. Als Versionsverwaltungssystem für den Sourcecode kam *Git* zum Einsatz.

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, den

Ort, Datum

Rechtsverbindliche Unterschrift: Sebastian Wiesendahl