# AIOps-basierte Anomalie Erkennung in Logdaten moderner Software-Systeme

## Zwischenstandbericht

**Studierende*r:**

Muhammad Taha Imran

C2410838030

**Betreuer*in:**

Prof. Mugdim Bublin

# Contents

# 1. Einleitung

Moderne verteilte Softwaresysteme erzeugen große Mengen an Logdaten, die Informationen über Systemzustände, Fehler, Performanceprobleme und ungewöhnliches Verhalten enthalten. Aufgrund der hohen Datenmenge und Komplexität ist eine manuelle Analyse dieser Logs nicht praktikabel. Daher gewinnen automatisierte Verfahren zur Anomalieerkennung zunehmend an Bedeutung.

Ziel der Anomalieerkennung in Logdaten ist es, ungewöhnliche oder unerwartete Muster zu identifizieren, die auf Fehler, Fehlkonfigurationen oder sicherheitsrelevante Vorfälle hinweisen können. Klassische regelbasierte Ansätze stoßen hierbei schnell an ihre Grenzen, da sie schwer wartbar sind und nicht gut auf neue oder unbekannte Fehler reagieren.

Diese Masterarbeit untersucht den Einsatz von **Machine-Learning-basierten Anomalieerkennungsverfahren** für Logdaten, mit besonderem Fokus auf **unüberwachte und semi-überwachte Methoden**. Diese sind besonders relevant, da in realen Systemen meist nur wenige oder unzuverlässige Anomalielabels vorhanden sind.

Der vorliegende Zwischenstandsbericht beschreibt den aktuellen Arbeitsstand. Er gibt einen Überblick über die verwendeten Datensätze, die durchgeführten Vorverarbeitungsschritte, die implementierten Modelle sowie die bisher erzielten Ergebnisse und Erkenntnisse.

# 2. Problemstellung und Zielsetzung

## 2.1 Problemstellung

Die Anomalieerkennung in Logdaten ist mit mehreren Herausforderungen verbunden:

- Logdaten sind häufig unstrukturiert oder nur teilstrukturiert
- Anomalien treten selten auf und sind sehr heterogen
- Labels sind oft nicht vorhanden oder unvollständig
- Normales Systemverhalten kann sich im Laufe der Zeit verändern

Diese Eigenschaften erschweren den Einsatz klassischer überwachten Lernverfahren und machen robuste, adaptive Methoden erforderlich.

## 2.2 Zielsetzung

Die Ziele dieser Arbeit sind:

- Aufbereitung und Strukturierung von Roh-Logdaten für Machine Learning
- Implementierung verschiedener Anomalieerkennungsverfahren
- Quantitative und qualitative Bewertung der Verfahren
- Analyse der Interpretierbarkeit der erkannten Anomalien

Der Fokus des aktuellen Projektstandes liegt auf der **Implementierung, Validierung und explorativen Analyse**, nicht auf einer finalen Optimierung.

# 3. Verwendete Datensätze

## 3.1 HDFS-Logdatensatz

Der HDFS-Logdatensatz stammt aus einem verteilten Dateisystem und wird häufig in der Forschung zur Log-Anomalieerkennung verwendet. Er enthält:

- strukturierte Logtemplates
- Ereignisse in Form von Event-IDs
- Sequenzen von Events
- Ground-Truth-Labels für Anomalien

Durch die vorhandenen Labels eignet sich der Datensatz gut für eine **quantitative Evaluation** der Modelle.

## 3.2 Apache HTTP Server Logs

Zusätzlich werden Apache-HTTP-Server-Logs verwendet. Diese stellen ein realitätsnahes Szenario dar, da sie **keine expliziten Anomalielabels** enthalten.

Die Apache-Logs dienen dazu:

- das Modell in einer unlabeled Umgebung zu testen
- qualitative Analysen durchzuführen
- die Praxistauglichkeit der Methode zu bewerten

## 3.3 ZooKeeper Logs (vorbereitet/geplant)

ZooKeeper-Logs wurden ebenfalls vorbereitet und sollen in späteren Phasen verwendet werden, um die Generalisierbarkeit des Ansatzes zu überprüfen.

Zusätzlich wird im weiteren Verlauf der Arbeit geprüft, ob weitere geeignete Logdatensätze integriert werden können, um sowohl die Datenmenge als auch die

Vielfalt der betrachteten Systeme zu erhöhen. Eine Erweiterung der Datengrundlage könnte dazu beitragen, die Aussagekraft der Ergebnisse weiter zu verbessern und die Generalisierbarkeit der entwickelten Anomalieerkennungsverfahren zu stärken.

# 4. Vorverarbeitung und Feature Engineering

## 4.1 Log Parsing und Template-Zuordnung

Durch das Log Parsing wird die hohe Variabilität natürlicher Logtexte reduziert, wodurch die Lernbarkeit für datengetriebene Modelle verbessert wird.

Die Rohlogs werden zunächst geparst und in strukturierte Form überführt. Jede Logzeile wird einem **Logtemplate** zugeordnet, welches die semantische Bedeutung der Nachricht repräsentiert. Dadurch wird Rauschen reduziert und die Vergleichbarkeit von Logs verbessert.

## 4.2 Erstellung von Ereignissequenzen

Für die sequenzbasierte Modellierung werden Logeinträge zu **Event-Sequenzen** zusammengefasst. Jede Sequenz stellt einen zeitlichen Ablauf von Systemereignissen dar.

Sequenzen bilden die Grundlage für rekurrente neuronale Netze, da diese zeitliche Abhängigkeiten zwischen Ereignissen explizit modellieren können.

Beispiel:

[21, 21, 59, 59, 59, 59]

Diese Sequenzen erfassen das Systemverhalten wesentlich besser als einzelne Logeinträge.

## 4.3 Padding und Normalisierung

Da neuronale Netze feste Eingabelängen benötigen, werden Sequenzen:

- auf eine maximale Länge begrenzt
- mit Nullen aufgefüllt (Padding)

Für Apache-Logs wurde eine maximale Sequenzlänge von 50 verwendet, um Konsistenz mit dem HDFS-Datensatz zu gewährleisten.

Die Wahl einer einheitlichen Sequenzlänge ermöglicht einen konsistenten Modellvergleich zwischen unterschiedlichen Datensätzen.

## 4.4 Numerische Repräsentation

Die finalen Eingabedaten liegen als Tensoren der Form:

(N, Sequenzlänge, 1)

vor und sind damit für rekurrente neuronale Netze geeignet.

# 5. Implementierte Modelle

## 5.1 Isolation Forest (Baseline)

Als Baseline wurde ein Isolation-Forest-Modell implementiert. Dieses isoliert Anomalien basierend darauf, wie schnell sie in zufälligen Entscheidungsbäumen separiert werden können.

Eine Hyperparametersuche zeigte, dass das Modell brauchbare Ergebnisse erzielt, jedoch zeitliche Abhängigkeiten nur eingeschränkt erfassen kann.

Der Isolation Forest dient primär als Referenzmodell und erlaubt einen Vergleich zwischen sequenzbasierten und nicht-sequenzbasierten Ansätzen.

```
best_auc = 0.0
best_model = None
best_params = None

for params in itertools.product(
    param_grid["contamination"],
    param_grid["max_samples"],
    param_grid["n_estimators"],
    param_grid["max_features"],
    param_grid["bootstrap"],
):
    contamination, max_samples, n_estimators, max_features, bootstrap = params

    model = IsolationForest(
        contamination=contamination,
        max_samples=max_samples,
        n_estimators=n_estimators,
        max_features=max_features,
        bootstrap=bootstrap,
        random_state=42,
        n_jobs=-1,
    )

    model.fit(X_train)

    scores = -model.decision_function(X_eval)
    auc = roc_auc_score(y_eval, scores)

    if auc > best_auc:
        best_auc = auc
        best_model = model
        best_params = {
            "contamination": contamination,
            "max_samples": max_samples,
            "n_estimators": n_estimators,
            "max_features": max_features,
            "bootstrap": bootstrap,
        }

print("BEST ROC-AUC:", best_auc)
print("BEST PARAMS:", best_params)
```

## 5.2 LSTM Autoencoder

Autoencoder eignen sich besonders für Anomalieerkennung, da sie normales Verhalten gut rekonstruieren, während ungewöhnliche Muster zu erhöhten Rekonstruktionsfehlern führen. Als Hauptmodell wurde ein **LSTM Autoencoder** implementiert. Dieser besteht aus:

- einem Encoder, der Sequenzen in einen kompakten latenten Raum projiziert
- einem Decoder, der versucht, die ursprüngliche Sequenz zu rekonstruieren

Das Modell wird ausschließlich mit **normalen Sequenzen** trainiert.

```
[10]
EMBED_DIM = 32
LATENT_DIM = 64
BATCH_SIZE = 256
EPOCHS = 20
```

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import (
    Input, Embedding, LSTM, RepeatVector, TimeDistributed, Dense
)
from tensorflow.keras.callbacks import EarlyStopping

inputs = Input(shape=(MAX_LEN,))

x = Embedding(
    input_dim=VOCAB_SIZE,
    output_dim=EMBED_DIM
)(inputs)

encoded = LSTM(LATENT_DIM, activation="tanh")(x)

decoded = RepeatVector(MAX_LEN)(encoded)
decoded = LSTM(EMBED_DIM, activation="tanh", return_sequences=True)(decoded)

outputs = TimeDistributed(
    Dense(VOCAB_SIZE, activation="softmax")
)(decoded)

model = Model(inputs, outputs)
model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy"
)

model.summary()
[11]
```

```
Model: "functional"
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_layer (InputLayer) | (None, 20) | 0 |
| embedding (Embedding) | (None, 20, 32) | 960 |
| lstm (LSTM) | (None, 64) | 24,832 |
| repeat_vector (RepeatVector) | (None, 20, 64) | 0 |
| lstm_1 (LSTM) | (None, 20, 32) | 12,416 |
| time_distributed (TimeDistributed) | (None, 20, 30) | 990 |

```
Total params: 39,198 (153.12 KB)

Trainable params: 39,198 (153.12 KB)

Non-trainable params: 0 (0.00 B)
```

## 5.3 Training

Das Training erfolgt mit:

- Mean Squared Error als Verlustfunktion
- Early Stopping
- Validierungssplit von 10 %
- Die Trainings- und Validierungskurven zeigen eine stabile Konvergenz.

```
X_train, X_val = train_test_split(
    X_normal,
    test_size=0.2,
    random_state=42
)

X_eval = X_feat
y_eval = labels

print("Train normals:", X_train.shape)
print("Validation normals:", X_val.shape)
print("Evaluation set:", X_eval.shape)
```

✓ 0.1s

```
Train normals: (446578, 29)
Validation normals: (111645, 29)
Evaluation set: (575061, 29)
```

```
early_stop = EarlyStopping(
    monitor="loss",
    patience=3,
    restore_best_weights=True
)

history = model.fit(
    X_train_in,
    X_train_out,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    callbacks=[early_stop],
    shuffle=True
)
```

# 6. Anomalieerkennung

## 6.1 Rekonstruktionsfehler

Nach dem Training wird für jede Sequenz der Rekonstruktionsfehler berechnet:

$$\text{Fehler} = \frac{1}{T}\sum(x - \hat{x})^2$$

- **x** = ursprüngliche Eingabesequenz (z. B. Event-IDs / Zeitfenster)

- **x̂** = vom Autoencoder rekonstruierte Sequenz

- **T** = Länge der Sequenz (z. B. 50 Events)

Hohe Fehler deuten auf ungewöhnliches Verhalten hin.

## 6.2 Schwellenwertbestimmung

Die Wahl des Schwellenwerts stellt einen Trade-off zwischen Precision und Recall dar und hängt vom jeweiligen Anwendungsszenario ab.

Da Apache-Logs keine Labels enthalten, werden Schwellenwerte über **Perzentile der Fehlerverteilung** bestimmt (z. B. 95–99.9 %).

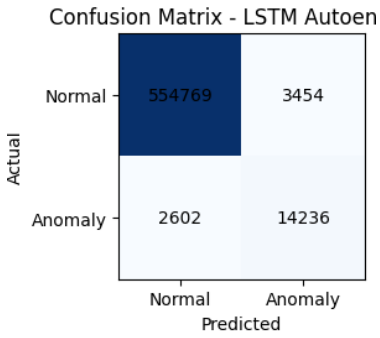
Ein Schwellenwert-Sweep wurde implementiert, um Precision, Recall und F1-Score zu analysieren.

## 6.3 Evaluationsmetriken

Verwendete Metriken:

- ROC-AUC
- Precision
- Recall
- F1-Score
- Konfusionsmatrix

```
ROC-AUC: 0.9207505687371418
Precision: 0.7863
Recall:    0.8451
F1-score:  0.8147
```




Zusätzliche Visualisierungen sowie ausgewählte Codeausschnitte befinden sich im Appendix.

# 7. Ergebnisse und Analyse

## 7.1 Ergebnisse auf HDFS

Der LSTM Autoencoder erreicht eine ROC-AUC von ca. **0.92** und übertrifft damit den Isolation Forest deutlich.

### 7.2 Ergebnisse auf Apache-Logs

Die Fehlerverteilung zeigt eine klare Trennung zwischen normalem Verhalten und Ausreißern. Die am stärksten auffälligen Sequenzen weisen auf:

- sehr repetitive Muster
- geringe Entropie
- ungewöhnliche Event-Wiederholungen

## 7.3 Entropieanalyse

Die berechnete Entropie der auffälligen Sequenzen ist häufig nahe null, was auf fehlerhafte oder blockierte Systemzustände hinweist.

In einem nächsten Schritt sollen die Ergebnisse auf einer feineren Label-Ebene systematisch verglichen werden. Hierzu ist geplant, weitere Logdatensätze in das Experiment aufzunehmen und die Modelle auf einer größeren und heterogeneren Datenbasis erneut zu trainieren. Darüber hinaus kann die Einbindung zusätzlicher

Modellansätze oder alternativer Architekturen dazu beitragen, die Robustheit und Generalisierbarkeit der Ergebnisse weiter zu erhöhen. Eine detaillierte Gegenüberstellung der Modelle unter erweiterten Datensätzen soll eine fundiertere Bewertung der jeweiligen Stärken und Schwächen ermöglichen.

## 8. Zwischenfazit

Der aktuelle Stand zeigt:

- erfolgreiche Implementierung mehrerer Modelle
- gute Ergebnisse auf gelabelten Daten
- plausible und interpretierbare Anomalien auf unlabeled Logs

Der Ansatz ist funktional und vielversprechend.

## 9. Nächste Schritte

Geplante Arbeiten:

- Anwendung auf ZooKeeper-Logs
- Vergleich der Anomalien zwischen Systemen
- Optimierung der Schwellenwertwahl
- Tiefergehende qualitative Analyse
- Integration in die finale Thesis

## 10. Fazit

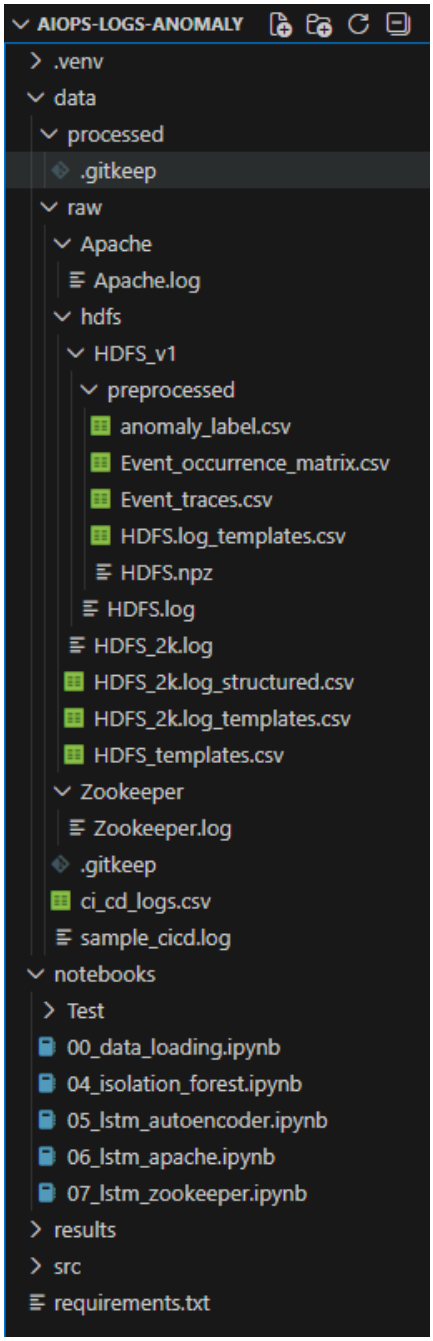
Die bisherigen Ergebnisse zeigen, dass LSTM-basierte Autoencoder eine effektive Methode zur Anomalieerkennung in Logdaten darstellen. Der Zwischenstand bestätigt die Machbarkeit des Ansatzes und bildet eine solide Grundlage für die weitere Arbeit.

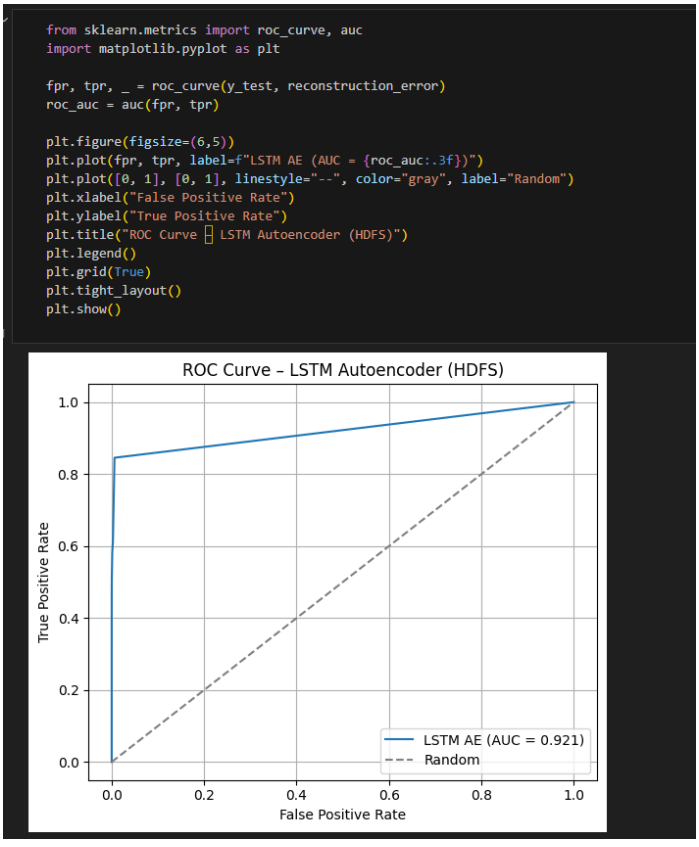
# Appendix

**Figure A.1:**
Projektstruktur und zentrale Notebooks zur Anomalieerkennung.

**Figure A.2:**
ROC-Kurve des LSTM-Autoencoders für den HDFS-Datensatz.

**Figure A.3:**
Vergleich zwischen tatsächlicher Anzahl an Anomalien und durch das Modell detektierten Anomalien.



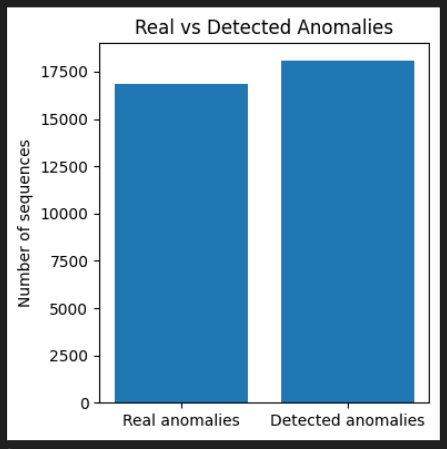



```python
import matplotlib.pyplot as plt

labels = ["Real anomalies", "Detected anomalies"]
values = [n_real_anomalies, n_pred_anomalies]

plt.figure(figsize=(4,4))
plt.bar(labels, values)
plt.ylabel("Number of sequences")
plt.title("Real vs Detected Anomalies")
plt.tight_layout()
plt.show()
```

0.0s




```python
# Ground truth counts
n_total = len(y_test)
n_real_anomalies = (y_test == 1).sum()
n_real_normal = (y_test == 0).sum()

# Model predictions
n_pred_anomalies = (y_pred == 1).sum()
n_pred_normal = (y_pred == 0).sum()

print(f"Total samples: {n_total}")
print(f"Real anomalies (ground truth): {n_real_anomalies}")
print(f"Real normal: {n_real_normal}")
print(f"Detected anomalies (model): {n_pred_anomalies}")
print(f"Detected normal (model): {n_pred_normal}")
```

0.0s

```
Total samples: 575061
Real anomalies (ground truth): 16838
Real normal: 558223
Detected anomalies (model): 18097
Detected normal (model): 556964
```