Data Minung und Big Data

Identifikation von Bottlenecks in der Produktion am Beispiel des Materialflusses

Alwine Schultze

April 2024

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

Eines der zentralen Ziele für Unternehmen im produzierenden Gewerbe ist die Effizienzsteigerung in den Produktionshallen. Vermeintliche Engpässe in diesen Hallen, sogenannte Bottlenecks (engl. Flaschenhals), führen zu Verzögerungen in den Produktionsabläufen, was wiederum die Produktivität im Allgemeinen vermindert. Corona ist ein Paradebeispiel für Engpässe in zahlreichen Produktionshallen. Besonders betroffen waren 2020-2021 Produkte, die chinesische Chips verbaut hatten. Aber auch die Automobilindustrie hatte mit Produktionsengpässen zu kämpfen, da zeitweise die Grenzen zwischen den Zulieferländern, wie Tschechien und Polen, geschlossen wurden.
Jedes Bottleneck in der Fertigungshalle bedeutet Umsatzeinbußen. Daher sollte die zeitnahe Identifikation und Beseitigung dieser von großer Bedeutung für das produzierende Gewerbe sein. Process-Mining und Anomaliedetection bieten hierfür innovative Ansätze.
Welche Techniken sich besonders eignen, wird in der zugrundeliegenden Arbeit am Beispiel eines Materialflusses analysiert.

1.2 Zielsetzung der Arbeit

Die vorliegende Arbeit konzentriert sich auf die Beantwortung der Forschungsfrage, wie mithilfe von Methoden des maschinellen Lernens einfach und effizient Bottlenecks bzw. Anomalien in der Produktion, anhand von Bewegungsdaten von Paletten identifiziert werden können. Dabei werden die folgenden Leitfragen beantwortet:

- Welche Methoden zur Identifikation von Bottlenecks gibt es?
- Wie könnte eine Umsetzung zur Detektion von Bottlenecks in einem Produktionsprozess aussehen?

1.3 Aufbau der Arbeit

Der Aufbau der Arbeit gliedert sich in drei Hauptabschnitte. Im ersten Drittel der Arbeit werden die *Grundlagen* von Process-Mining, Methoden zur Identifikation von Bottlenecks und die Motivation für die Untersuchung dieser Themen erläutert. Im Haupteil der Arbeit wird die konkrete *Umsetzung der Erkennung von Anomalien im Produktionsbetrieb anhand des Materialflusses* beschrieben. Dies beinhaltet die Analyse des Datensets, die Datenaufbereitung und das Feature-Engineering, die Process Discovery sowie die Anwendung von Machine Learning-Methoden wie dem LSTM-Modell zur Identifikation von Bottlenecks. Abschließend fasst das *Fazit* die Kernergebnisse zusammen und gibt einen Ausblick auf Optimierungspotentiale.

2 Grundlagen

2.1 Was versteht man unter Process-Mining?

Unter Process-Mining versteht man die Nutzung von Daten, die bereits in verschiedenen Systemen des Unternehmens gespeichert sind, um Erkenntnisse aus den tatsächlichen Prozessen zu gewinnen (vgl. Peters und Nauroth 2019, S. 3). Diese Daten stammen beispielsweise aus dem Materialfluss einer Produktionsanlage. Im Gegensatz zu Annahmen oder Anekdoten liefert Process Mining objektive, faktenbasierte Einblicke in den tatsächlichen Ablauf von Prozessen (vgl. Celonis 2024). Die dadurch gewonnenen Erkenntnisse können genutzt werden, um Ineffizienzen zu identifizieren und Verbesserungspotenziale abzuleiten, was sich wiederum positiv auf die Kosten und den Umsatz des Unternehmens auswirken kann. Beim Process-Mining können verschiedene Methoden angewendet werden. Gemäß Pawar (2023) zählen die folgenden zu den gängigsten Methoden:

Process Discovery Bei der *Process Discovery* (aus dem engl. Prozessfindung) geht es, wie der Name bereits sagt um die Identifikation eines bestehenden Prozesses. Dabei werden Ereignisprotokolle automatisch in ein Prozessmodell umgewandelt. Dabei werden auch Routenwahrscheinlichkeiten, häufige Pafe und Variationen im Prozessfluss aufgezeigt (vgl. Pawar 2023, S. 22).

Conformance Checking Das Ziel beim *Conformance Checking* (aus dem engl. Konformitätsprüfung) ist es einen Standard-Prozess-Fluss (a-priori-Modell) zu definieren und es mit dem gewonnenen Prozess aus der Realität abzugleichen. Dadurch können Abweichungen, aber auch Gemeinsamkeiten augedeckt werden. Die Konformitätsprüfung hilft also dabei Fälle zu entdecken, die nicht wie vorgesehen ablaufen (vgl. Pawar 2023, S. 24).

Enhancement / Process Re-engineering Auch beim *Enhancement* (aus dem engl. Anpassung) wird ein a-priori-Modell benötigt, jedoch aus einem anderen Grund. Bei dieser Methode liegt das Ziel darin, das zugrunde gelegte Modell zu ändern oder zu erweitern. Identifizierte Abweichungen im gelebten Prozess könnten durchaus seine Gültigkeit haben (vgl. Pawar 2023, S. 25).

Operational Support Die Methode *Operational Support* (aus dem engl. Operationale Unterstützung) hat das Ziel, wie der Name schon andeutet, Geschäftsoperationen zu unterstützen. Sie ermöglicht es, schwierig laufende Fälle zu identifizieren, zukünftige Entwicklungen zu antizipieren und korrigierende Maßnahmen zu empfehlen. Beim *Operational Support* kommt die künstliche Intelligenz mit Methoden des Maschinellen Lernens ins Spiel. Hier können die Methoden des Maschinellen Lernens folgende drei Aufgaben erfüllen: *Detect*, *Predict* und *Recommend* (vgl. Pawar 2023, S. 27).

Detect Maschinelles Lernen kann für das Aufspüren (engl. *Detect*) von Abweichungen vom a-priori-Modell zur Laufzeit verwendet werden. Beim Auftreten dieser, wird eine automatische Warnung vom System verschickt (vgl. Pawar 2023, S. 28).

Predict Bei einer *Prediction* (aus dem engl. Vorhersage) handelt es sich um die Vorhersage, was als nächstes im Prozess passieren könnte. Dabei wird der aktuelle Prozess mit ähnlichen Prozess-Durchläufen aus der Vergangenheit vergleichen, dies ermöglicht ein vorhersehen was als nächstes passieren könnte (vgl. Pawar 2023, S. 28).

Recommend *Recommend* (aus dem engl. empfehlen) ist eine Methode, die auf der Vorhersage basiert. Auf deren Basis werden dann entsprechende Empfehlungen für mögliche Gegenmaßnahmen ausgesprochen, die dazu führen, dass Probleme frühzeitig erkannt und behoben werden können (vgl. Pawar 2023, S. 29).
Doch welche der oben genannten Methoden eignen sich besonders um Bottlenecks in der Produktion zu identifizieren?

2.2 Methoden zur Identifizierung von Bottlenecks

In einem Produktionsprozess können Bottlenecks zum Beispiel Engpässe sein, die den Durchsatz in der Produktion begrenzen. Als möglichen Ursachen sind beispielsweise die Abnahme der Maschinenleistung, ein Mangel in der Materialversorgung oder Kapazitätsengpässe vorstellbar. Ein Bottleneck äußert sich in digitaler Form als Datenanomalie und kann mittels Machine Learning Methoden identifiziert werden, indem der Standardprozess einem Modell antrainiert wird und dieses anschließend Abweichungen im tatsächlichen Prozessverlauf erkennt.
Auch die Methode *Detect* des Prozess-Minings kann genutzt werden um zu kontrollieren, ob der Materialfluss seinen gewohnten Lauf nimmt. Dabei gleicht man den Materialfluss einer Palette in der Produktionshalle mit dem identifzierten a-priori-Modell ab. Bei abweichendem Materialfluss könnte der Produktionsleiter entsprechend vom System benachrichtigt werden.
Ein mögliches Szenario wären ungewöhnlich lange Verweildauern von Material oder Produkt an Maschinenplätzen. Zur Identifikation von Anomalien in Durchlaufzeiten eignen sich die Methode *Predict*. Bei dieser Methode lässt sich die Durchlaufzeit an Maschinen einem Modell gut antrainieren. Verhält sich der gelebte Prozess stark anders als vom Modell gelernt, so könnte eine Warnung vom System an den Produktionsleiter verschickt werden.
Für die hier zugrunde liegende Arbeit wurde sich für die Methode *Predict* für die Erkennung von Bottlenecks entschieden und umgesetzt.

3 Umsetzung der Erkennung von Anomalien im Produktionsbetrieb anhand des Materialflusses

3.1 Analyse des Datensets

Eine ausführliche Analyse des Datensets kann wertvolle Einblicke und Erkenntnisse liefern. Im vorliegenden Fall handelt es sich um einen Datensatz aus einer Produktionslinie über den Zeitraum über eine Woche, vom 30.11.2020 bis einschließlich 05.12.2020, mit über 59.000 Datenreihen.
Nach einer eingehenden Analyse in Excel, unter Berücksichtigung von eindeutigen Werten in den jeweiligen Spalten und nach Informationsgehalt der Daten, wurden folgende Spalten für ein Prozess-Mining und die Identifikation von Bottlenecks ausgewählt:

- Der Zeitstempel (**Timestamp**), in der die Palette den Bereich der Messung betreten hat, vermutlich den Bereich, der einer Machine zugesprochen wird.
- Die **Area** entspricht dem Bereich in den Produktionshallen. Zum Beispiel *Montage BAU 2 1.OG*, über diese Eigenschaft lassen sich Rückschlüsse auf die Reisezeit der Palette schließen. Insgesamt gibt es in dem Datensatz sechs Bereiche.
- Das Attribut **PLCName** hält den Maschinennamen (z.B. *RS71SPS1*) bereit. Insgesamt sind es 49 Maschinen im vorliegenden Datenset.
- Unter **HMI_MZEV_EW** verbirgt sich die Palettenbezeichnung, z.B. *200075*.

Bei dem vorliegenden Datensatz handelt es sich um unbereinigte Produktionsdaten, dieser muss zunächst noch entsprechend aufbereitet und um weitere Informaionen angereichert werden.

3.2 Datenaufbereitung und Feature-Engineering

Die Untersuchung auf Nullwerte im Datensatz erbrachte keine Ergebnisse. Allerdings wurden bei den Paletten zwei auffällige Werte identifiziert, *0* und *#WERT!*. Diese wurden aus dem Datensatz für die Process Discovery ausgeschlossen. Um den Materialfluss genau abbilden zu können, ist es außerdem sinnvoll, die *Herkunft* jeder Palette zu bestimmen. Außerdem kann die Bestimmung der *Verweildauer* einer Palette an einer Maschine zur Identifikation von Engpässen beitragen. Ein weiteres potenziell relevantes Feature, das aus dem Datensatz gewonnen werden kann, ist ein *Wechsel der Area*, da dies zusätzliche Informationen über die Reisedauer der Palette liefert. Zum Wechsel der Area kann es zudem sinnvoll erscheinen die jeweiligen Area Attribute *BAU* und *Etage* als einzelne Features zu extrahieren. Die Methoden aus Anhang A wurde für das Feature Engineering verwendet und führt die notwendigen Datentransformationen aus.

	Zeitpunkt	Area	Maschine	Palette	Herkunft	Verweildauer	Area_Wechsel	BAU	Etage
1	2020-12-01 06:56:44.000	Montage BAU 3 UG	F710SPS1	105600	UB75SPS2	92.050000	True	3	UG
2	2020-12-01 07:13:10.000	Montage BAU 2 1.OG	AS71SPS1	105600	F710SPS1	16.433333	True	2	OG
3	2020-12-01 07:18:14.000	Montage BAU 2 1.OG	AS72SPS1	105600	AS71SPS1	5.066667	False	2	OG
4	2020-12-01 07:32:51.000	Montage BAU 3 UG	F720SPS1	105600	AS72SPS1	14.616667	True	3	UG
5	2020-12-01 07:35:05.000	Montage BAU 2 1.OG	FS71SPS1	105600	F720SPS1	2.233333	True	2	OG

Abbildung 1: Daten nach Bereinigung und Anreicherung

Abbildung 1 zeigt den bereinigten und angereicherten Datensatz. In dieser Darstellung können die ersten fünf Palettenbewegungen der Palette *105600* mit allen Attributen, wie Verweildauer und Herkunft der Palette, nachvollzogen werden. Diese Informationen ermöglichen bereits eine Ableitung des Produktionsprozesses. Wie genau dies erfolgt, wird im nachfolgenden Kapitel erläutert.

3.3 Process Discovery

Wie bereits im Kapitel 2.1 erläutert, handelt es sich bei *Process Discovery* um einen Kernbreich des Process Minings, in dem der Prozess aus Ereignisprotokollen abgeleitet wird. Die aufbereiteten Daten aus Abbildung 1 sind bereits ausreichend, um daraus einen Prozess ableiten zu können. Die wichtigsten Eigenschaften sind dabei die *Maschine*, an der die Palette registriert wurde, und die *Herkunft* der Palette.

```
dfCounts = dfCleaned.groupby(['Herkunft', 'Maschine']).\
                size().reset_index(name='Count')

for index, row in dfCounts.iterrows():
    print(row['Herkunft'], '->', row['Maschine'], ':', row['Count'])

# Beispiel Output:
# AS71SPS1 -> AS72SPS1 : 1204
# AS71SPS1 -> F720SPS1 : 3
# AS71SPS1 -> FS71SPS2 : 5
# AS72SPS1 -> F720SPS1 : 1205
```

Durch das Zählen der Kombinationen von Herkunft und Maschine lässt sich die häufigste Prozessroute ermitteln. Mithilfe des im Anhang B.1 dargestellten Pythoncodes kann der Prozess, wie in Abbildung B.2 visualisiert werden. Die dick markierten Routen repräsentieren jeweils das a-priori-Modell. Die dünnen Linien zeigen dabei kleinere Abweichungen im Prozessverlauf auf.
Um das a-priori-Modell zu ermitteln, eignet sich der Pythoncode aus dem Anhang B.3. Mit diesem Code kann der Standardprozess identifiziert und in der Abbildung aus dem Anhang B.4 visualisiert werden. Damit ist der Schritt der *Process Discovery* abgeschlossen. Obwohl noch weitere

Methoden des Process-Minings, wie das *Conformance Checking* oder weitere, angewendet werden könnten, liegt der Fokus der vorliegenden Arbeit primär auf der Erkennung von Anomalien. Daher wird im nächsten Kapitel näher auf die Methode *Predict* eingegangen.

3.4 Predict

3.4.1 Bereinigung der Ausreißer

Für die Anomalieerkennung wird ein Attribut benötigt, das vorhergesagt werden soll. Im vorliegenden Fall wird die *Verweildauer* der Palette an der Maschine als zu prognostizierendes Attribut verwendet. Die Abbildung 2 zeigt die Verweildauern (y-Achse) der Paletten für jede Maschine (x-Achse).

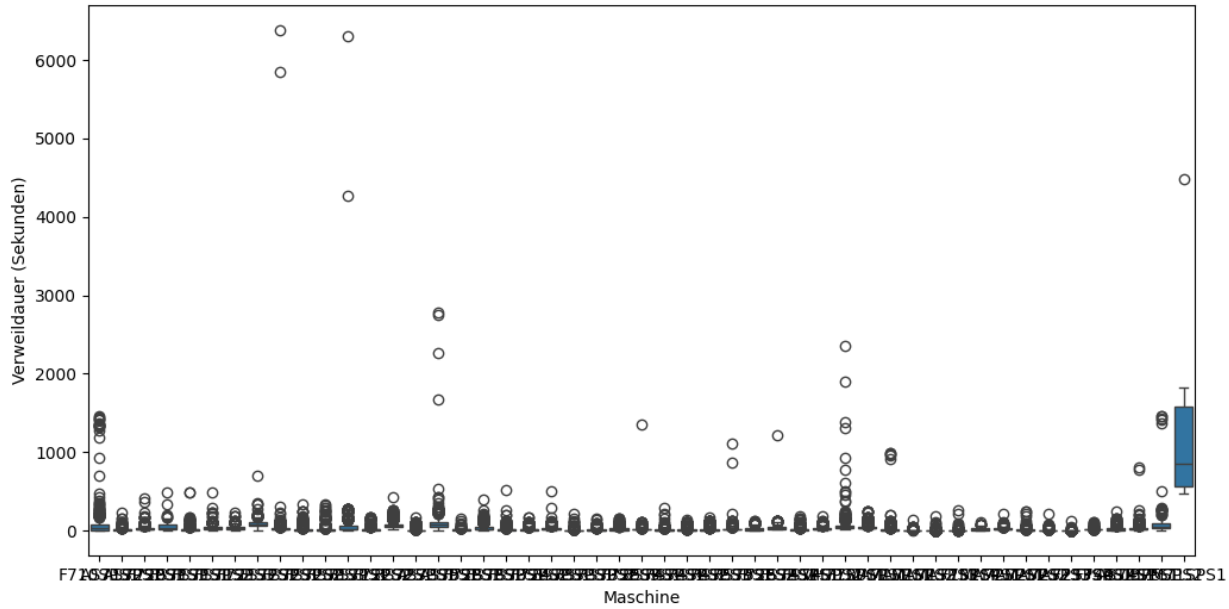

Abbildung 2: Aufteilung der Verweildauer pro Maschine

Es ist deutlich erkennbar, dass die üblichen Verweildauern sich im Bereich von wenigen Sekunden befinden. Allerdings treten auch Ausreißer nach oben auf, die Verweildauern von über 1000 Sekunden aufweisen. Um das Modell so zu trainieren, dass realistische Verweildauern erlernt werden können, müssen diese Ausreißer entfernt werden. Hierfür eignet sich die Methode des Interquartilbereichs (IQR), es ist ein statistisches Maß, das die Streuung der Daten um den Median beschreibt. Er wird verwendet, um Ausreißer in einem Datensatz zu identifizieren.
Dabei werden die Werte, in unserem Fall Verweildauern, im oberen und unteren Quartil betrachtet. Verweildauern außerhalb der Quantilbereiche werden als Ausreißer betrachtet und anschließend gezielt entfernt. Dies ermöglicht eine Bereinigung des Datensatzes und ermöglicht eine Fokussierung auf die Hauptdatenverteilung. Der folgende Python-Code zeigt, wie Ausreißer in den Verweildauern ermittelt und etnfernt werden.

```python
# Berechnung des Interquartilbereichs (IQR)
Q1 = df['Verweildauer'].quantile(0.25)
Q3 = df['Verweildauer'].quantile(0.75)
IQR = Q3 - Q1

# Definieren des unteren und oberen Schwellenwerts für Ausreißer
lower_threshold = Q1 - 1.5 * IQR
upper_threshold = Q3 + 1.5 * IQR

# Filtern der Einträge, die innerhalb des IQR liegen
df = df[(df['Verweildauer'] >= lower_threshold) &
        (df['Verweildauer'] <= upper_threshold)]
```

Nach dem Entfernen der Ausreißer aus dem Datensatz ergibt sich nun ein ausgewogeneres Bild der Verweildauern (vgl. Abbildung 3).

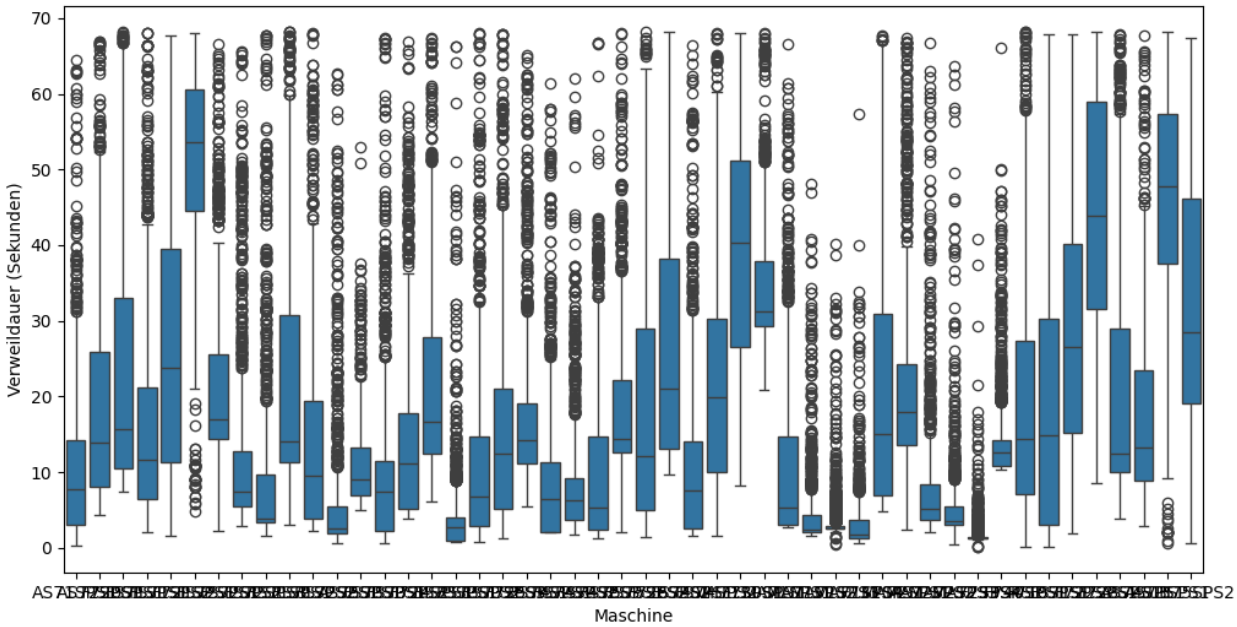

Abbildung 3: Aufteilung der Verweildauer pro Maschine nach Bereinigung der Ausreißer

Mit den nun vorliegenden Daten kann ein Modell trainiert werden, das die Verweildauer einer Palette anhand der anderen vorhandenen Eigenschaften vorhersagen kann.

3.4.2 Encoding der Eigenschaften

Da es sich bei dem vorliegenden Datensatz um Zeitreihendaten handelt, empfiehlt die Praxis die Verwendung eines Modells, das speziell für den Umgang mit Zeitreihen geeignet ist. Für die Lösung des vorliegenden Problems bietet sich das Long Short-Term Memory (LSTM) Modell an. Dieses Modell ist in der Lage, anhand von ganzen Sequenzen trainiert zu werden und kann somit

komplexe zeitliche Abhängigkeiten in den Daten erfassen und nutzen.
Vor dem Training müssen die relevanten Attribute in maschinenlesbare Features umgewandelt werden. Dieser Prozess beinhaltet die Transformation der Daten in ein Format, das von den Machine-Learning-Algorithmen verarbeitet werden kann.

```
1  # Encoding von kategorischen Variablen
2  le_maschine = LabelEncoder()
3  le_maschine.classes_ = sorted(dfML['Maschine'].unique())
4  dfML['Maschine_encoded'] = le_maschine.fit_transform(dfML['Maschine'])
5  dfML['Herkunft_encoded'] = le_maschine.fit_transform(dfML['Herkunft'])
6  le_bau = LabelEncoder()
7  dfML['BAU_encoded'] = le_bau.fit_transform(dfML['BAU'])
8  le_etage = LabelEncoder()
9  dfML['Etage_encoded'] = le_etage.fit_transform(dfML['Etage'])
10
11 # Encoding von Area_Wechsel
12 dfML['Area_Wechsel_encoded'] = dfML['Area_Wechsel'].astype(int)
13
14 # Encoding des Zeitpunkts
15 dfML['Zeitpunkt_encoded'] = pd.to_datetime(dfML['Zeitpunkt']).
16                             astype('int64')
17
18 # Encoding der Verweildauer ('Verweildauer')
19 sc_verweildauer = RobustScaler()
20 dfML['Verweildauer_scaled'] = sc_verweildauer.fit_transform(
21                         dfML['Verweildauer'].values.reshape(-1, 1))
```

Bei den Eigenschaften *Maschine*, *Herkunft*, *BAU* und *Etage* handelt es sich um kategorische Variablen. Diese Werte repräsentieren verschiedene Kategorien, haben jedoch keine natürliche Ordnung zwischen ihnen und können daher eigenständig verwendet werden. Da die Maschinenbezeichnung sowohl in der Spalte *Herkunft* als auch in der Spalte *Maschine* vorkommen, kann hier derselbe Encoder verwendet werden (vgl. Zeilen 2 bis 9).
Der *Area_Wechsel* wird lediglich als 0 oder 1 dargestellt, da es sich um einen einfachen boolschen Wert handelt (vgl. Zeile 12). Bei der Transformation des *Zeitpunkts* wird lediglich eine Umwandlung in einen Unix-Zeitstempel vorgenommen (vgl. Zeilen 14 und 15).
Für die Verweildauer, die vorhergesagt werden soll, wird eine Skalierung durchgeführt, bei der jeder Wert in einen Bereich zwischen -1 und 1 transformiert wird. Diese Skalierung auf einen begrenzten Bereich soll dazu beitragen, Ausreißer zu reduzieren und die Empfindlichkeit des Modells gegenüber extremen Werten zu verringern. Dies wiederum kann die Stabilität und Zuverlässigkeit der Vorhersagen verbessern (vgl. Zeilen 19-21). Mit den nun vorliegenden, maschinenverständlichen daten kann nun ein Modell trainiert werden, das die Verweildauer einer Palette anhand der

anderen vorhandenen Eigenschaften vorhersagen kann.

3.4.3 Training eines LSTM

Im Rahmen der Lösungskonzeption wurden die Methoden *train_test_split*, *calc_mse*, *plot_history* und *train* implementiert (vgl. Anhang C). Die Funktionalität der jeweiligen Funktionen wird in den Folgenden Abschnitten näher erläutert.

train_test_split Die Methode *train_test_split* (vgl. Anhang C.1) teilt die Eingabedaten *X* und die zugehörigen Zielvariablen *y* in Trainings- und Testsets auf, wobei die Daten in Sequenzen mit der angegebenen Länge *sequence_length* unterteilt werden. Die Aufteilung in Trainings- und Testdaten ist wichtig, um später das trainierte Modell anhand ungesehener Daten bewerten zu können. Für die Aufteilung werden zunächst Sequenzen von *X* erstellt (vgl. Zeile 3-4). Die Sequenzbildung ist wichtig, da es sich hier um Zeitreihendaten handelt und das Modell nicht einzelne Datenpunkte betrachten soll, sondern ganze Sequenzreihen. Anschließend werden die Daten in Trainings- und Testsets aufgeteilt, wobei 80% der Daten für das Training und 20% für Tests verwendet werden (vgl. Zeilen 6-10). Die Methode gibt die Trainings- und Testsets für *X* und *y* entsprechend zurück (vgl. Zeile 12).

calc_mse Die Methode *calc_mse* (vgl. Anhang C.2) berechnet den Mean Squared Error (MSE) zwischen den vorhergesagten Werten und den tatsächlichen Werten für das Testset. Zunächst werden Vorhersagen mit dem trainierten Modell durchgeführt (vgl. Zeile 2). Anschließend werden die Skalierungen der Vorhersagen und der wahren Werte rückgängig gemacht (vgl. Zeile 6), um den MSE in absoluten Einheiten (Sekunden) zu ermitteln (vgl. Zeile 9). Der MSE und der Root Mean Squared Error (RMSE), der die durchschnittliche Abweichung zwischen den vorhergesagten und den tatsächlichen Werten angibt, werden in der Console ausgegeben (vgl. Zeile 12 -14).

plot_history Die Methode *plot_history* (vgl. Anhang C.3) dient dazu, die Verläufe des Trainings- und Validierungsverlusts, die während des Trainings ermittelt werden, zu visualisieren. Dazu werden die Loss-Werte aus dem übergebenen *history*-Objekt verwendet und in einem Diagramm dargestellt. Dies ermöglicht eine visuelle Bewertung der Performance des Modells während des Trainingsprozesses.

train Die Methode *train* (vgl. Anhang C.4) dient dazu, das LSTM-Modell zu trainieren und zu evaluieren. Dabei werden die folgenden Parameter verwendet:

- Der Parameter *sequence_length* bestimmt die Länge der Eingabesequenzen für das LSTM-Modell. Mit der Sequenzlänge wird reguliert, wie viele aufeinanderfolgende Zeitschritte als

Eingabe von dem Modell betrachtet werden sollen. Sie ist entscheidend für die zeitliche Struktur der Daten.

- Die *features* bestimmen, welche Merkmale oder Variablen als Eingabe für das Modell verwendet werden sollen (z.B. Herkunft, Maschine, Bau, Etage). Mit ihnen kann festgelegt werden, welche Informationen dem Modell beim Training zur Verfügung stehen. Damit lässt sich die Lernfähigkeit und Leistung des Modell beeinflussen.

- Mit der *learning_rate* wird die Schrittweite festgelegt, mit der die Parameter des Modells während des Trainings angepasst werden. Damit wird die Geschwindigkeit, mit der das Modell lernt reguliert, dies widerum beeinflusst die Geschwindigkeit, mit der die Lernkurve konvergiert. Diese Einstellung kann direkt die Qualität der gefundenen Lösungen beeinflussen. Eine zu hoch gewählte Lernrate kann dazu führen, dass das Modell das optimale Minimum (Loss) überspringt, während eine zu niedrige Lernrate dazu führen kann, dass das Modell das Minimum in den angegebenen Schritten (Epochen) erst gar nicht erreicht.

- Mit dem *drop_out* wird der Prozentsatz der Neuronen festgelegt, die während des Trainings zufällig ausgeschaltet werden. Mit dem DropOut lässt sich Overfitting vermeiden, außerdem kann er dazu beitragen die Generalisierungsfähigkeit zu verbessern.

- Die *batch_size* gibt an, wie viele Datenpunkte mit einem Mal während des Trainings vom Modell verarbeitet werden sollen. Damit kann die Effizienz des Trainingsprozesses und der Speicherverbrauch beim Training direkt beeinflusst werden.

- Der *epochs*-Parameter bestimmt, wie oft das gesamte Trainingsdatenset während des Trainings maximal durchlaufen wird. Die Epochen regulieren somit die Anzahl der Trainingsschritte und damit die Anpassungsfähigkeit des Modells an die Trainingsdaten. Zu viele Epochen können zu einer Überanpassung (Overfitting) beim Modell führen.

- Die *estopping*-Variable wird für das EarlyStopping verwendet. Diese ermöglicht das Training zu stoppen, wenn sich die Leistung des Modells auf dem Validierungsdatensatz nicht verbessert. Sie reguliert die vorzeitige Beendigung des Trainings, damit lässt sich ebenfalls Overfitting verhindert werden.

- Mit dem Parameter *optimizer* wird der Algorithmus festgelegt, der während des Trainings verwendet wird, um die Modellparameter anzupassen. Der Optimizer reguliert direkt den Optimierungsprozess und beeinflusst die Konvergenzgeschwindigkeit sowie die Qualität der Lösung. In dem hier vorliegenden Anwendungsfall hat sich der *Adam*-Optimizer als geeignet herausgestellt, es können jedoch auch andere Optimizer verwendet werden. Das Besondere am Adam-Optimizer ist, dass er automatisch die Schrittgröße des Trainings anpasst und außerdem Informationen über vergangene Trainingsfortschritte speichert.

- Die *loss*-Funktion bestimmt, wie der Fehler zwischen den tatsächlichen und den vorhergesagten Werten berechnet wird. Sie reguliert damit direkt das Trainingsverhalten und beeinflusst die Optimierung des Modells, da während des Trainings versucht wird, den Loss-Wert so minimal wie möglich zu halten. Im hier zugrundegelegten Trainingsprozess ist die Überwachung des mittleren quadratischen Fehlers (*mean_squared_error*) eine geeignete Maßnahme zur Bewertung der Modellleistung.

Zunächst werden in der Methode *train* die Eingabedaten *X* und die zugehörigen Zielvariablen *y* aus dem DataFrame extrahiert und anschließend in Trainings- und Testsets aufgeteilt (vgl. Zeilen 3-7). Danach wird ein LSTM-Modell erstellt, das aus zwei LSTM-Schichten und einer Dichteschicht besteht. Die erste LSTM-Schicht hat dabei 64 Neuronen, von denen einige während des Trainings zufällig deaktiviert werden (*Dropout*, vgl. Zeilen 11-12). Für die zweite LSTM-Schicht wurden bewusst weniger Neuronen gewählt, nämlich die Hälfte, um die Modellkomplexität zu begrenzen und eine Überanpassung an die Trainingsdaten zu vermeiden (vgl. Zeile 13). Auch nach der zweiten LSTM-Schicht werden einige Neuronen deaktiviert (*Dropout*, vgl. Zeile 14). Bei dem Hyperparameter-Tuning wurden auch andere Konfigurationen mit mehr oder weniger Neuronen getestet. Jedoch führten mehr Neuronen zu keinem besseren Ergebnis, sondern nur zu einer deutlichen Verlängerung der Trainingszeit, auf bis zu 45 ms pro Schritt. Im Gegensatz dazu beträgt die Trainingszeit bei der finalen Konfiguration mit festgelegter Neuronenzahl lediglich 10 ms pro Schritt. Die *Dense*-Schicht aggregiert die Merkmale des Neuronen aus den LSTM-Schichten und führt anschließend die finale Vorhersage durch (vgl. Zeile 15).
Anschließend wird der *Adam*-Optimizer mit einer angepassten Lernrate definiert und das Modell wird mit der angegebenen Verlustfunktion *mean_squared_error* kompiliert (vgl. Zeilen 17 und 19). In den Zeilen 21-23 wird außerdem ein EarlyStopping-Callback definiert, um das Training zu beenden, sobald sich die Validierungsverluste nicht mehr verbessern.
Das Training des Modells findet in den Zeilen 25-27 statt. Hier werden neben den Trainingsdaten *X* und *y* auch die erforderlichen Parameter wie die Anzahl der Epochen, die Batchgröße, der Validierungssplit und der EarlyStopping-Callback übergeben.
Nach dem Training wird die Leistung des Modells anhand des Test Loss bewertet und ausgegeben (vgl. Zeilen 27-29). Der Test Loss gibt hier an, wie gut das Modell auf den Testdaten abschneidet. Er misst dabei die durchschnittliche Abweichung zwischen den vorhergesagten Werten des Modells und den tatsächlichen Werten der Testdaten.
Im Anschluss wird in der Methode *train* noch die Visualisierung des Trainings- und Validierungsverlustes visualisiert und die beiden Werte MSE und RMSE ausgegeben, diese dienen zur Bewertung der Leistung des Modells.
Während des Hyperparameter-Tunings wurden verschiedene Kombinationen von *sequence_length*, *features*, *learning_rate*, *drop_out*, *batch_size* und *estopping* ausprobiert. Die beste Kombination, die sich als am effektivsten erwiesen hat, ist die Folgende:

```
features = ['Maschine_encoded', 'Herkunft_encoded', 'BAU_encoded',
            'Etage_encoded']
model = train(sequence_length=25,
              features=features,
              learning_rate=0.01,
              drop_out=0.3,
              batch_size=64,
              epochs=50,
              estopping=15)
model.save('LSTM_Process_Mining_2.keras')
```

In Abbildung 4 wird das Ergebnis des Trainings dargestellt. Es wurden insgesamt 18 von 50 Epochen trainiert, da sich kein weiteres Absinken des Validierungsverlusts mehr abzeichnete.

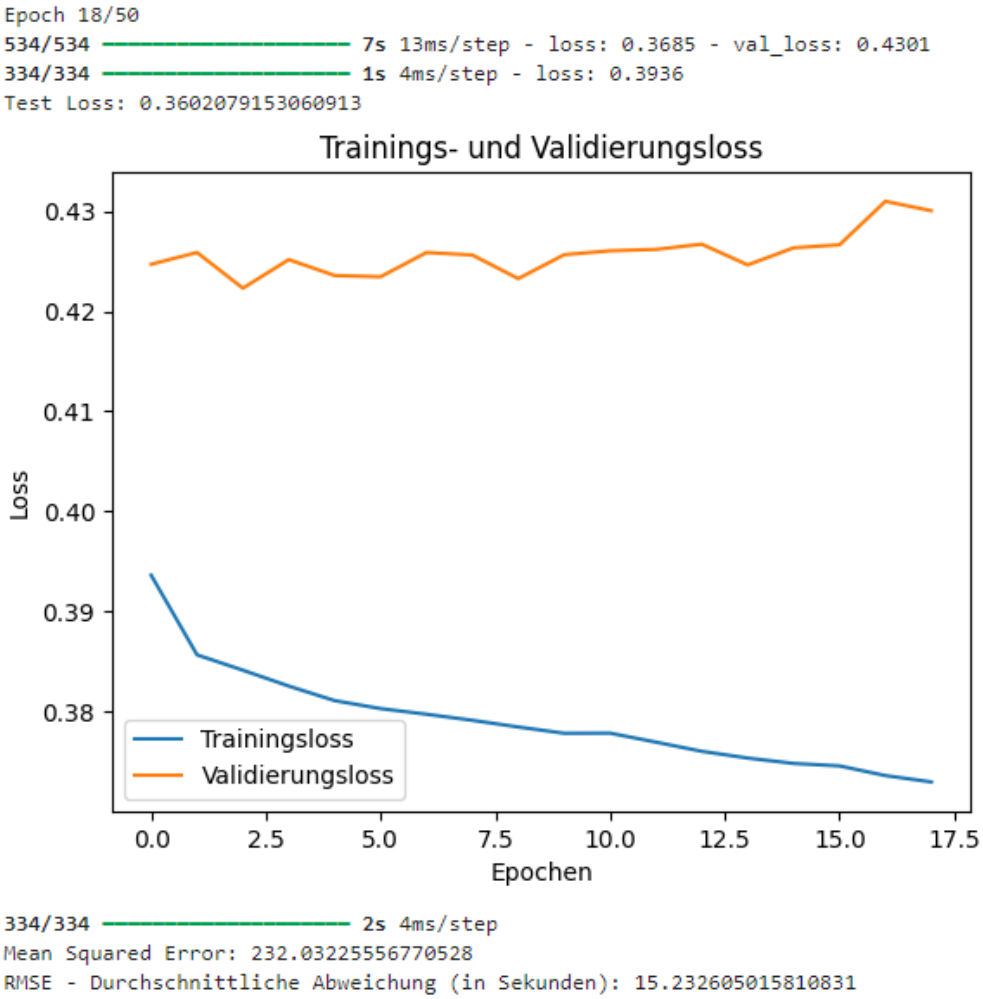

Abbildung 4: Trainingsresultat

Die grafische Darstellung zeigt einen kontinuierlichen Rückgang des Trainingsverlusts (blau) mit zunehmender Anzahl der Epochen, was darauf hinweist, dass das Modell während des Trainings

lernt und die Fehlerquote abnimmt. Der Validierungsverlust (orange) sinkt zunächst ebenfalls, beginnt aber nach etwa zehn Epochen wieder zu steigen, was auf Überanpassung (Overfitting) hinweist. Dies bedeutet, dass das Modell die Trainingsdaten zu gut lernt und daher Schwierigkeiten hat, auf neue Daten zu verallgemeinern. Das EarlyStopping sorgt jedoch dafür, dass das Training entsprechend abgebrochen wird, um eine Überanpassung zu vermeiden. Abschließend zeigt die Durchschnittliche Abweichung der vorhergesagten zu den tatsächlichen Werten, die bei 15.23 Sekunden liegt, die Leistung des Modells.
Mithilfe des trainierten Modells können nun die Anomalien bzw. Bottlenecks identifiziert bzw. vorhergesagt werden.

3.4.4 Anomalieerkennung mithilfe des LSTM-Modells

Nach dem erfolgreichen Training des LSTM-Modells liegt in diesem Unterkapitel der Fokus auf der Verwendung des Modells zur Identifizierung von Anomalien. Hierzu werden zunächst die Vorhersagen für das bestehende vollständige Datenset vom LSTM-Modell getätigt.
Für die Vorhersage der Werte wird die Methode *predict_values* (vgl. Anhang D) verwendet. Dieser Code führt eine Vorhersage für die Verweildauer von Paletten an Maschinen in einem Produktionsprozess durch, basierend auf einem trainierten LSTM-Modell. Zuerst werden die erforderlichen Merkmale ausgewählt und das trainierte Modell geladen (vgl. Zeilen 2 bis 4). Dann werden Sequenzen von Merkmalswerten mit einer festgelegten Länge erstellt (vgl. Zeilen 5-9). Für jede Sequenz wird eine Vorhersage mithilfe des Modells getätigt, und die Vorhersagewerte wieder in Sekunden umgewandelt (vgl. Zeile 10-17). Schließlich werden die vorhergesagten Werte in eine CSV-Datei gespeichert (vgl. Zeile 19).
Die Berechnung der Residuen, also die Abweichung zwischen tatsächlichem Wert und dem vorhergesagtem Wert, ist eine wichtige Information zur Erkennung von Anomalien. Der folgende Code löscht zunächst alle Zeilen des Datensets heraus, die keine Vorhersage besitzen, zum Beispiel, wenn die Sequenz für eine Vorhersage nicht vollständig vorlag. Außerdem werden die Residuen berechnet die statistischen Werte dieser ausgegeben:

```
1  dfML.dropna(subset=['Vorhersage'], inplace=True)
2  dfML['residuen'] = (dfML['Verweildauer'] - dfML['Vorhersage']).abs()
3
4  print("Durchschnitt:", dfML['residuen'].mean())
5  print("Median:", dfML['residuen'].median())
6  print("Standardabweichung:", dfML['residuen'].std())
7  print("Minimum:", dfML['residuen'].min())
8  print("Maximum:", dfML['residuen'].max())
9  print("25. Perzentil:", dfML['residuen'].quantile(0.25))
10 print("75. Perzentil:", dfML['residuen'].quantile(0.75))
11
```

```
# Output:
Durchschnitt: 20.219176935055543
Median: 12.321652475992838
Standardabweichung: 57.7451047827564
Minimum: 0.0003197987874337116
Maximum: 6289.3171980539955
25. Perzentil: 6.373351446787518
75. Perzentil: 15.870989004770916
```

Insgesamt wurden 58.288 Vorhersagen getroffen, die aussagekräftig für eine Analyse sind, da sie nicht Null sind. Dabei beträgt die durchschnittliche Abweichung etwa 20 Sekunden. Besonders auffällig ist der maximale Abweichungswert von über 6289 Sekunden, der wahrscheinlich auf eine Anomalie hinweist. Auch die Grafische Darstellung der Residuen in Abbildung 5 zeigt klar, dass kaum Abweichungen jenseits kleiner zweistelliger Nummern vorliegen.

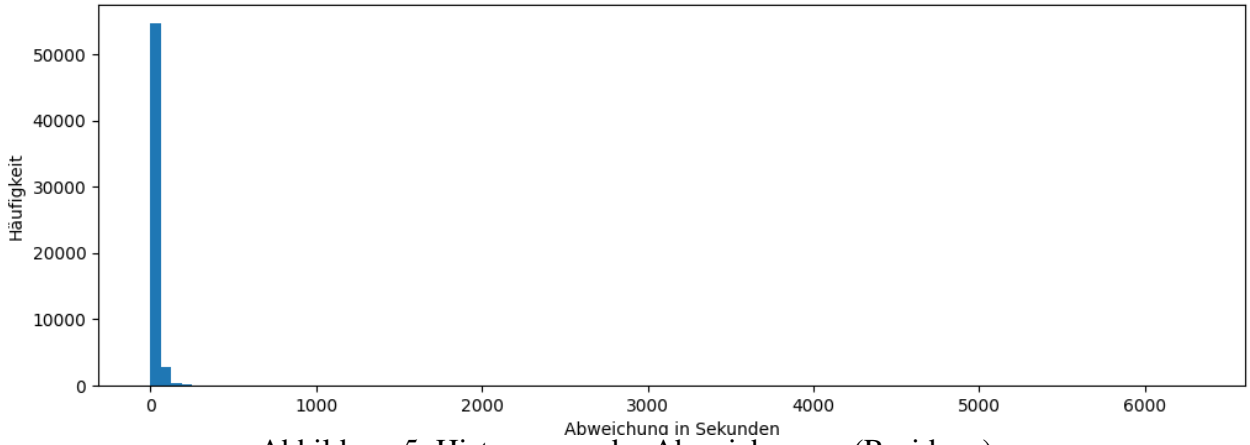

Abbildung 5: Histogramm der Abweichungen (Residuen)

Für eine zuverlässige Erkennung von Anomalien ist es entscheidend, eine klare Messlatte festzulegen, ab wann eine Abweichung als signifikant betrachtet wird. In diesem Kontext wurde in dieser Arbeit definiert, dass Anomalien als Werte betrachtet werden, die über dem 99. Perzentil der Residuen liegen. Dies bedeutet, dass die oberen 1% der Residuen als Anomalien eingestuft werden. Durch diese Festlegung eines Grenzwerts wird eine klare Schwelle geschaffen, um ungewöhnliche Ereignisse im Produktionsprozess zu identifizieren. Für die Implementierung dieses Ansatzes wurde der folgende Python-Code verwendet:

```
perzentil_99 = dfML['residuen'].quantile(0.99)

dfML['Anomalie'] = dfML['residuen'] > perzentil_99
dfML.to_csv('Predicted_values.csv', index=False)
```

Mit dem oben genannten Code werden alle Abweichungen, die etwas mehr als 134 Sekunden betragen, als Anomalie eingestuft. In Abbildung 6 werden diese Anomalien grafisch dargestellt,

um ihren zeitlichen Verlauf zu visualisieren und aufzuzeigen, wann sie im Produktionsprozess aufgetreten sind.

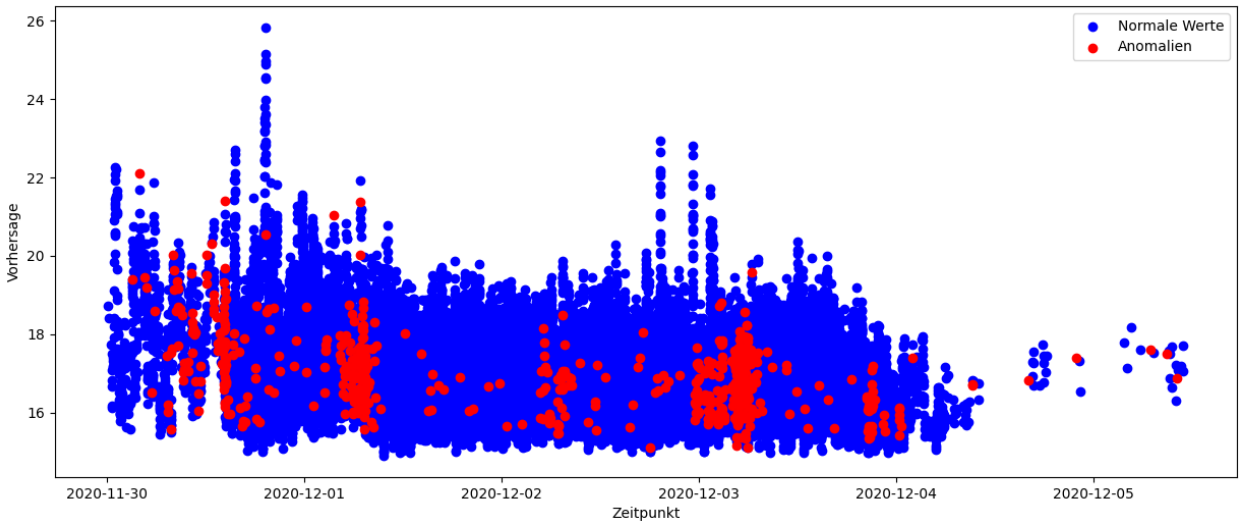

Abbildung 6: Darstellung der Anomalien über die Zeit

Man könnte nun das Modell und die hier vorgeschlagene Methode automatisieren und bei jeder Anoamlie, die hier in der Abbildung rot markiert ist eine Warnung an den Produktionsleiter rausschicken. Dies könnte jedoch dazu führen, dass dass der Produktionsleiter überflutet wird und die Warnungen nicht mehr ernst nimmt. Eine alternative Herangehensweise wäre, die Anomalien in einem bestimmten Zeitraum zu zählen und erst bei einer Häufung eine automatisierte Warnung zu versenden. Hier ist ein möglicher Python-Code für diese Art der Anomalieerkennung:

```
1  def filter_anomalien(df):
2      gruppen = df.set_index('Zeitpunkt').rolling('15Min').sum()
3      anomalie_fenster = gruppen[gruppen['Anomalie'] >= 5]
4      anomale_werte = df[df['Zeitpunkt'].isin(anomalie_fenster.index)]
5      return anomale_werte
6
7  anomale_werte = filter_anomalien(dfML[['Zeitpunkt', 'Vorhersage',
8                                         'Anomalie']].copy())
9
10 plt.figure(figsize=(15, 6))
11 plt.scatter(normale_werte['Zeitpunkt'], normale_werte['Vorhersage'],
12             label='Normale Werte', color='blue')
13 plt.scatter(anomale_werte['Zeitpunkt'], anomale_werte['Vorhersage'],
14             label='Anomalien', color='red')
```

In Zeile 2 werden die Gruppen mit einem rollierenden Zeitfenster von 15 Minuten gebildet und dabei Summen über die Spaltenwerte gebildet, was dazu führt, dass das Aufkommen von Anomalien entsprechend aggregiert wird. Wenn im angegebenen Zeitfenster von 15 Minuten mindestens

5 Anomalieausschläge auftreten, wird der Wert als anomal gekennzeichnet (vgl. Zeilen 3-4). Der Plot (vgl. Abbildung 7) wird mit den Code-Zeilen 10-14 generiert.

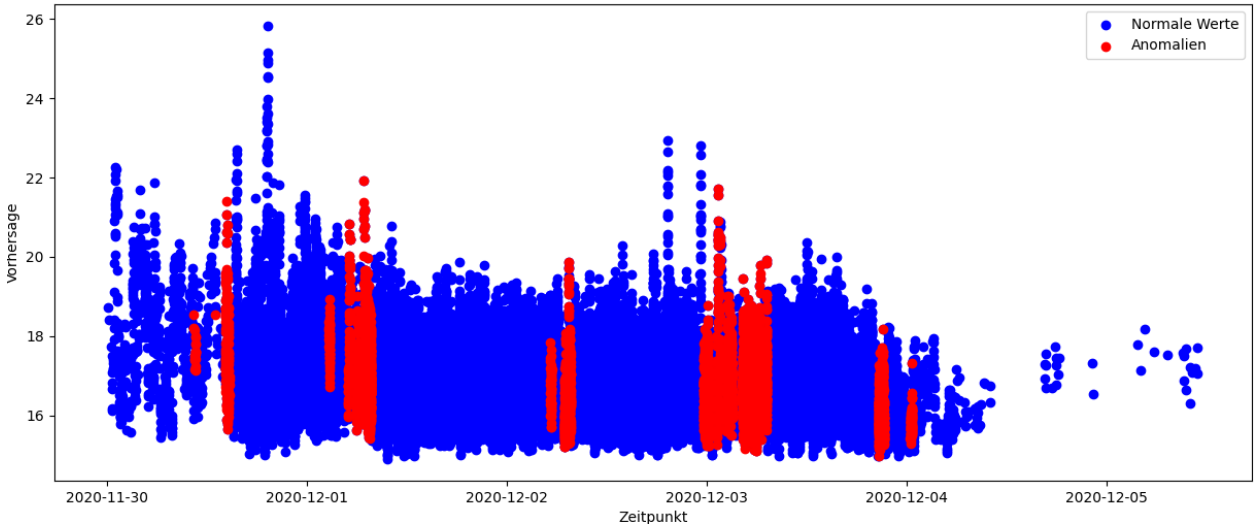

Abbildung 7: Darstellung der Anomalien mit rollierenden Zeitfenstern

Würde man diesen Ansatz verfolgen, so wäre es möglich die Anzahl der Warnmeldungen an den Produktionsleiter stark zu reduzieren und somit wirkliche Anomalien zu melden.

4 Fazit und Ausblick

Die vorliegende Arbeit zielt darauf ab, Bottlenecks in Produktionsprozessen durch die Kombination von Process-Mining und Anomaliedetection zu identifizieren. Eine detaillierte Analyse eines Datensatzes aus einer Produktionslinie über eine Woche konnte wertvolle Einblicke liefern. Während der Analyse wurden relevante Spalten des Datensatzes für das Prozess-Mining und die Bottleneck-Identifikation ausgewählt. Der Zeitstempel und andere eindeutige Werte wie Maschine, Herkunft und Etage ermöglichten es, mithilfe von Python-Code das a-priori-Modell zu ermitteln und grafisch darzustellen.
Die Arbeit beantwortet die Frage nach den verfügbaren Methoden zur Bottleneck-Identifikation und skizziert die Umsetzungsmöglichkeiten in einem Produktionsprozess. Dabei liegt ein Schwerpunkt auf Methoden des maschinellen Lernens, insbesondere auf dem Long Short-Term Memory (LSTM) Modell für Zeitreihendaten. Durch den Einsatz eines trainierten LSTM-Modells war es möglich, Abweichungen von normalen Betriebsabläufen zu erkennen, insbesondere anhand der Verweildauer einer Palette an einer Maschine. Warnungen können damit automatisch an den Produktionsleiter gesendet werden, um potenzielle Bottlenecks frühzeitig zu identifizieren und geeignete Maßnahmen zur Behebung dieser einzuleiten.
Um die Ergebnisse weiter zu verbessern, bietet sich die Integration zusätzlicher Datenquellen an, um ein umfassenderes Bild der Produktionsumgebung zu erhalten und die Genauigkeit der Anomalieerkennung zu erhöhen. Die Entwicklung einer Echtzeitüberwachungslösung auf Basis des trainierten LSTM-Modells könnte eine sofortige Erkennung von Anomalien und Echtzeitoptimierung ermöglichen. Eine Validierung der Methodik in realen Produktionsumgebungen könnte die praktische Wirksamkeit und Auswirkungen auf die Produktionsleistung zeigen.
Zusätzlich wäre es ratsam, einen alternativen Ansatz für die Anomaliedetection zu erwägen. Statt der Vorhersage der Verweildauer von Paletten an Maschinen könnte die Anzahl der Palettenbewegungen in einem rollierenden Zeitfenster verfolgt werden, und bei ungewöhnlichen Aktivitäten könnte eine Warnung an den Produktionsleiter ausgegeben werden. Dieser alternative Ansatz könnte die Detektion von Anomalien weiter verbessern und zusätzliche Einblicke in potenzielle Engpässe bieten.

Literaturverzeichnis

Pawar, Vaishali (2023). *Holistic Assessment of Process Mining in Indirect Procurement*. Springer Fachmedien Wiesbaden. ISBN: 978-3-658-41452-8.

Peters, Ralf und Markus Nauroth (2019). *Process-Mining - Geschäftsprozesse: smart, schnell und einfach*. Springer Fachmedien Wiesbaden. ISBN: 978-3-658-24169-8.

Online-Quellen

Celonis (2024). *Was ist Process Mining?* en. Zugriff am 18. März 2024. URL: `https://www.celonis.com/de/process-mining/what-is-process-mining/`.

Anhang

A Feature Engineering

```
dfFlow['Herkunft'] = None
dfFlow['Verweildauer'] = 0.0
dfFlow['Area_Wechsel'] = False

for index in range(1, len(dfFlow)):
    act = dfFlow.loc[index]
    herk = dfFlow.loc[index - 1]
    if act['Palette'] == herk['Palette']:
        dfFlow.loc[index, 'Herkunft'] = herk['Maschine']
        dfFlow.loc[index, 'Verweildauer'] =
                    (pd.to_datetime(act['Zeitpunkt']) -\
                    pd.to_datetime(herk['Zeitpunkt'])).total_seconds() / 60
        if herk['Area'] != act['Area']:
            dfFlow.loc[index, 'Area_Wechsel'] = True

def extract_bau_and_floor(area):
    # Muster für BAU X
    bau_pattern = re.compile(r'BAU (\d+)')
    # Muster für Etage (optional)
    floor_pattern = re.compile(r'(UG|OG|QM)')

    bau_match = bau_pattern.search(area)
    floor_match = floor_pattern.search(area)

    bau = bau_match.group(1) if bau_match else None
    floor = floor_match.group(1) if floor_match else 'Unbekannt'
    return bau, floor

dfCleaned['BAU'] = dfCleaned['Area'].apply(lambda x:
                                           extract_bau_and_floor(x)[0])
dfCleaned['Etage'] = dfCleaned['Area'].apply(lambda x:
                                             extract_bau_and_floor(x)[1])
```

B Process Discovery

B.1 Prozessschaubild - Code

```
G = pgv.AGraph(strict=False, directed=True)

G.graph_attr['rankdir'] = 'TB'
G.node_attr['shape'] = 'box'

# Schleife über alle verfügbaren Maschinen (dfMachineExists ist ein Set)
# Knoten für jede Maschine in die Grafik hinzufügen
for index, row in dfMachineExists.iterrows():
    text = row['Maschine'] + '\n(' + str(row['Total_Exits']) + ')'
    G.add_node(row['Maschine'], label=text)

# Ermittlung von Min/Max um die Strichstärke berechnen zu können
countMin = dfCounts['Count'].min()
countMax = dfCounts['Count'].max()

# Festlegen der Strichstärken (1 = Min, 5 = Max)
thickMin = 1.0
thickMax = 5.0

# Schleife über alle Verbindungen um die Kanten einzuzeichnen.
for index, row in dfCounts.iterrows():
    count = row['Count']
    thick = thickMin + (thickMax-thickMin) *
            float(count-countMin) / float(countMax-countMin)
    G.add_edge(row['Herkunft'], row['Maschine'],
               label=count, penwidth=thick)

G.draw('full_graph.png', prog='dot')
```

B.2 Prozessschaubild - Abbildung

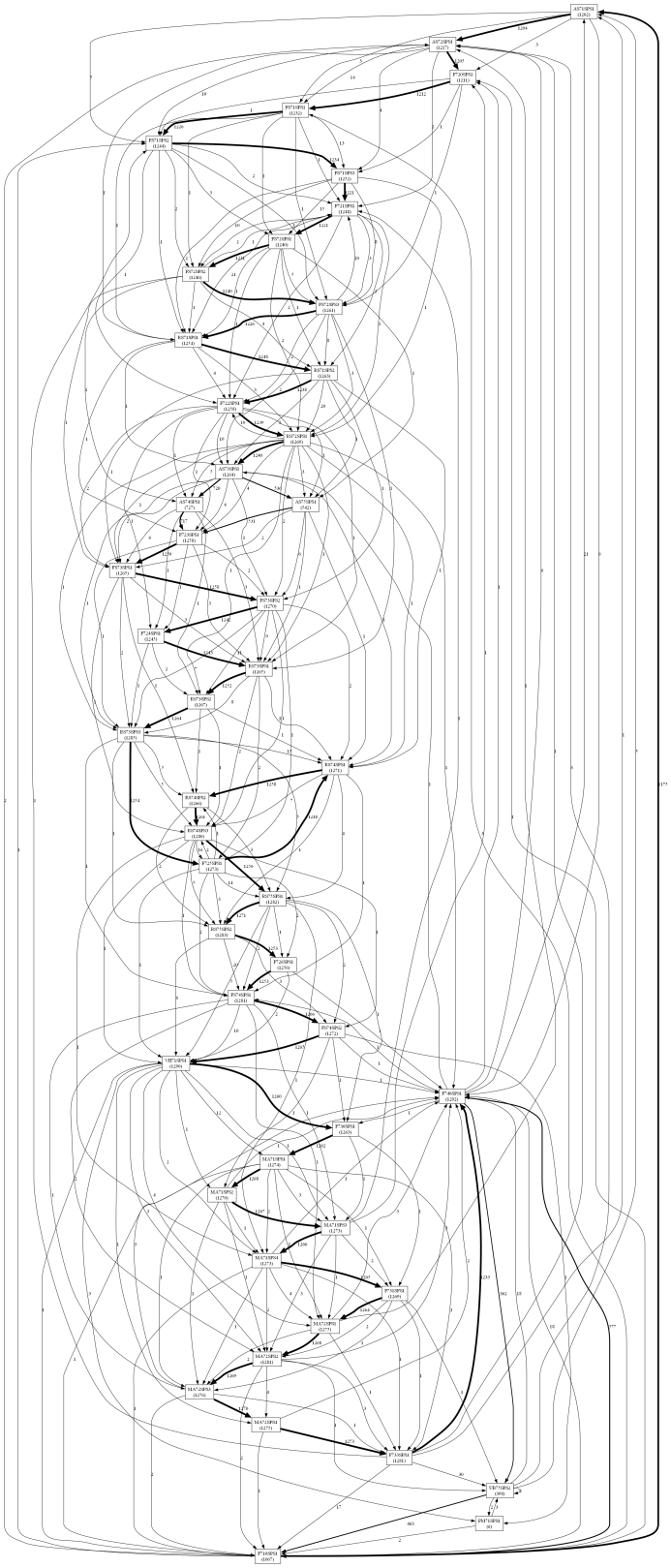

Abbildung B.1: Prozessschaubild

B.3 A-priori-Modell - Code

```
G = pgv.AGraph(strict=False, directed=True)

G.graph_attr['rankdir'] = 'TB'
G.node_attr['shape'] = 'box'

exitsMin = dfMachineExists['Total_Exits'].min()
exitsMax = dfMachineExists['Total_Exits'].max()

# Knoten und Kanten einzeichnen, wenn mind. 50 mal die Route von Paletten
# zurückgelegt wurde (Ungewöhnliche Bewegungen entfernen)
for index, row in dfMachineExists.iterrows():
    text = row['Maschine'] + '\n(' + str(row['Total_Exits']) + ')'
    grayScale = int(float(exitsMax - row['Total_Exits']) /
                    float(exitsMax - exitsMin) * 100.)
    fillColor = 'gray' + str(grayScale)
    fontcolor = 'black'
    if grayScale < 50:
        fontcolor = 'white'
    G.add_node(row['Maschine'], label=text, style='filled',
               fillcolor=fillColor, fontcolor=fontcolor)

# Die häufigsten Wege sollen entsprechend dünn bzw. dick ausgegeben werden.
countMin = 50
countMax = dfCounts['Count'].max()

for index, row in dfCounts.iterrows():
    count = row['Count']
    if count >= 25:
        thick = thickMin + (thickMax-thickMin) * float(count-countMin) / \
                float(countMax-countMin)
        G.add_edge(row['Herkunft'], row['Maschine'], label=count,
        penwidth=thick)

G.draw('reduced_graph.png', prog='dot')
```

B.4 A-priori-Modell - Abbildung

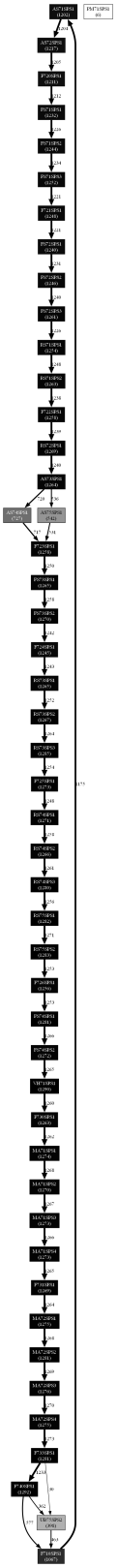

Abbildung B.2: A-priori Modell

C Trainingsmethoden

C.1 Methode *train_test_split*

Methode zur Umwandlung der Daten in Sequenzen für das LSTM-Modell.

```
def train_test_split(X, y, sequence_length):
    sequences = []
    for i in range(len(X) - sequence_length):
        sequences.append(X[i:i+sequence_length])

    # Aufteilen der Daten in Trainings- und Testsets
    X_train = np.array(sequences[:int(len(sequences)*0.8)])
    X_test = np.array(sequences[int(len(sequences)*0.8):])
    y_train = y[sequence_length:int(len(sequences)*0.8)+sequence_length]
    y_test = y[int(len(sequences)*0.8)+sequence_length:]

    return X_train, X_test, y_train, y_test
```

C.2 Methode *calc_mse*

Methode zur Berechnung des *Mean Squared Errors* und des *Root Mean Squared Error*

```
def calc_mse(model, X_test, y_test):
    predictions = model.predict(X_test)

    predictions_unscaled = sc_verweildauer.inverse_transform(predictions)

    y_test_unscaled = sc_verweildauer.inverse_transform(y_test.
                                                        reshape(-1, 1))

    mse_absolute = mean_squared_error(y_test_unscaled,
                                      predictions_unscaled)

    print('Mean Squared Error:', mse_absolute)
    print("RMSE - Durchschnittliche Abweichung (in Sekunden):",
          math.sqrt(mse_absolute))
```

C.3 Methode *plot_history*

Methode zur Visualisierung des Trainings- und Validierungsverlustes (loss)

```
def plot_history(history):
    plt.plot(history.history['loss'], label='Trainingsloss')
    plt.plot(history.history['val_loss'], label='Validierungsloss')
    plt.title('Trainings- und Validierungsloss')
    plt.xlabel('Epochen')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

C.4 Methode *train*

Methode zum Training eines LSTM mi zahlreichen Übergabeparametern, die besser im Text beschrieben sind.

```
def train(sequence_length, features, learning_rate, drop_out, batch_size,
          epochs, estopping, optimizer='adam', loss='mean_squared_error'):
    X = dfML[features].values
    y = dfML['Verweildauer_scaled'].values

    X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                        sequence_length)

    # LSTM-Modell-Aufbau
    model = Sequential()
    model.add(LSTM(units=64, return_sequences=True))
    model.add(Dropout(drop_out))
    model.add(LSTM(units=32))
    model.add(Dropout(drop_out))
    model.add(Dense(units=1))

    custom_optimizer = Adam(learning_rate=learning_rate)

    model.compile(optimizer=optimizer, loss=loss)

    early_stopping = EarlyStopping(monitor='val_loss',
                                   patience=estopping,
                                   restore_best_weights=True)

    history = model.fit(X_train, y_train, epochs=epochs,
                        batch_size=batch_size, validation_split=0.2,
                        callbacks=[early_stopping])

    score = model.evaluate(X_test, y_test)
```

```
30      print('Test Loss:', score)
31      plot_history(history)
32
33      calc_mse(model, X_test, y_test)
34      return model
```

D Anomalieerkennung

D.1 Methode *predict_values*

Methode zur Vorhersage der Verweildauern

```
def predict_values():
    features = ['Maschine_encoded', 'Herkunft_encoded', 'BAU_encoded',
                'Etage_encoded']
    model = load_model('LSTM_Process_Mining_2.keras')
    sequence_length = 25
    sequences = {}
    for feature in features:
        sequences[feature] = split_sequences(dfML[feature].values,
                                             sequence_length)
    for i in range(len(sequences[features[0]])):
        sequence = np.array([sequences[column][i] for column
                                                    in features]).T
        sequence = sequence.reshape(1, sequence_length, len(features))

        prediction = model.predict(sequence, verbose=0)
        dfML.at[i, 'Vorhersage'] = sc_verweildauer.inverse_transform(
                                        prediction[0][0].reshape(1, -1))[0]

    dfML.to_csv('Predicted_values.csv', index=False)
```