Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Vergleich von Verfahren des maschinellen Lernens zur Erkennung von Lahmheit bei Milchkühen

Comparison of machine learning methods for detecting lameness in dairy cows

Veit Zoche-Golob

Master-Abschlussarbeit

Betreuer:
Prof. Dr. Hans-Peter Beise (Hochschule Trier) und
Prof. Dr. Vitaly Belik (Freie Universität Berlin)

Wallersheim, 17.10.2022

Danksagung

Für mich stand schon recht früh in meinem Informatikstudium fest, dass ich mich für die Masterarbeit mit maschinellem Lernen beschäftigen möchte — ohne, dass ich bereits ein konkretes Thema gehabt hätte. Bis ich dann tatsächlich ein Thema gefunden hatte, dass ich bearbeiten konnte, dauerte es eine ganze Weile und es benötigte auch einen zweiten Anlauf. Daher bin ich meinem Betreuer Prof. Dr. Hans-Peter Beise sehr dankbar, dass er mein Vorhaben von Anfang an unterstützte und die Geduld aufbrachte, es solange zu begleiten. Seine einführenden Hinweise und Literaturempfehlungen halfen mir sehr, einen Einstieg in die verschiedenen Methoden des maschinellen Lernens zu finden.

Mein zweiter Betreuer, Prof. Dr. Vitaly Belik, verhalf mir dann zu einem konkreten Anwendungsfall für meine vagen Vorstellungen von meiner Masterarbeit. Bei ihm bedanke ich mich besonders für seine Mühen, mir Zugang zu den Daten aus dem Projekt Klauen*fit*net zu ermöglichen, die Unterstützung bei der Datenaufbereitung und die Vorschläge für interessante Klassifikationsansätze.

Mein Dank gilt auch den Projektbeteiligten von Klauen*fit*net und Klauen*fit*net 2.0 für die Erlaubnis, Daten aus dem Projekt für meine Masterarbeit zu verwenden und das Schema zur Bewegungsbeurteilung in meiner Masterarbeit zur Illustration wiederzugeben.

Beim HPC-Dienst der ZEDAT, Freie Universität Berlin, möchte ich mich für die zur Verfügung gestellte Rechenzeit bedanken und für die hilfreichen Tipps, wenn mit den Jobs etwas nicht so ganz optimal lief.

Für Korrekturen und Anmerkungen zum Manuskript bedanke ich mich sehr bei Dr. Rosemarie Quiring-Zoche und Dr. Anna-Linda Golob.

Ganz besonders danke ich Anna-Linda, Freyja und Arvid für die Unterstützung während der ganzen langen Zeit und die große Rücksichtnahme auf meine zusätzliche Arbeit.

Wallersheim, 17. Oktober 2022

Veit Zoche-Golob

Kurzfassung

Lahmheit ist eine der bedeutendsten Gesundheitsbeeinträchtigungen bei Milchkühen. Es ist sehr zeitaufwändig, lahme Kühe rechtzeitig und zuverlässig zu erkennen. Daher werden Systeme zur automatischen Lahmheitserkennung entwickelt. Solche Systeme können über unterschiedliche Sensoren wie Waagen oder Kameras das Gangbild der Kühe erfassen. Alternativ werden bereits existierende Daten über das Verhalten der Kühe verwendet, die aus Systemen zur automatischen Brunsterkennung stammen. Diese Daten sind meistens bereits aggregiert und werden als *indirekte Variablen* bezeichnet. Ein zentraler Bestandteil der Systeme zur automatischen Erkennung von Lahmheiten bei Milchkühen ist der Algorithmus, der die gemessenen Daten in „lahm“ oder „nicht lahm“ klassifiziert.

Das Ziel meiner vorliegenden Arbeit war es, verschiedene Klassifikationsalgorithmen zu vergleichen zur Erkennung von Lahmheit an Hand von indirekten Variablen zur Aktivität, zur Leistung und zu Eigenschaften der Kühe. Für meine Arbeit standen mir Daten aus dem Projekt Klauen*fit*net zur Verfügung mit den indirekten Variablen Laktationsnummer, Laktationstag, Tagesgemelk, durchschnittliche Schrittfrequenz und durchschnittliche Liegedauer pro Liegevorgang sowie den Labels „lahm“ oder „nicht lahm“ aus der Gangbeurteilung der Kühe. Die Werte der Variablen waren im Beobachtungszeitraum täglich erhoben worden. Als Merkmale zur Klassifikation wurden sowohl zeitabhängige Variablen wie die tägliche Aktivität und Milchmenge als auch (im Beobachtungszeitraum) konstante Variablen wie die Laktationsnummer verwendet. Da das Gangbild der Kühe nur etwa in 14-tägigen Abständen beurteilt worden war, waren die Labels der Beobachtungen unvollständig und standen für überwachte Lernverfahren nur eingeschränkt zur Verfügung.

In den Vergleich habe ich sowohl diskriminative als auch generative Ansätze zur Klassifikation von multivariaten Zeitreihen einbezogen. Neben Verfahren des klassischen maschinellen Lernens, wie Random Forests und Support Vector Machines (SVMs), wurden auch Methoden des Deep Learning, wie Multilayer Perceptrons (MLPs), Convolutional Neural Networks (CNNs) und Recurrent Neural Networks (RNNs), eingesetzt. In den diskriminativen Ansätzen wurden mittels Feature Engineering zusätzliche Merkmale erzeugt. Bei den Ende-zu-Ende-Ansätzen wurden die endgültigen Merkmale zur Klassifizierung in Deep-Learning-Modellen während des Trainings eines diskriminativen Klassifizierers erlernt. Damit die un-

gelabelten Daten ebenfalls für das Training der Klassifizierer verwendet werden konnten, wurden in generativen Ansätzen Deep-Learning-Modelle mit Autoencodern unüberwacht vortrainiert. Insgesamt umfasste der Vergleich neun Ansätze: drei Ansätze mit Feature Engineering (FE-SVM, FE-RF, FE-MLP), drei Ende-zu-Ende-Ansätze (E2E-MLP, E2E-CNN und E2E-GRU) sowie drei generative Ansätze mit unüberwachtem Vortraining (AE-MLP, AE-CNN, AE-GRU). Zur Bewertung der Klassifikationsleistungen der verschiedenen Ansätze wurden die Metriken Genauigkeit, Relevanz, Sensitivität und Spezifität berechnet. Zusätzlich erfasste ich die Laufzeiten für das Training und die Klassifikation sowie den benötigten Arbeitsspeicher. Die neun Ansätze wurden in zehnfacher Kreuzvalidierung mit jeweils denselben Teildatensätzen getestet. Für die Analyse der Unterschiede zwischen den erwarteten Klassifikationsleistungen der neun untersuchten Ansätze entwickelte ich ein grafisches Modell als Bayes'sches multivariates lineares Modell. So konnten die Unterschiede zwischen den Klassifikationsansätzen in den vier Vergleichsmetriken gleichzeitig analysiert werden.

Der verwendete Datensatz enthielt 727.008 Beobachtungen. Zu 17.114 Beobachtungen davon lagen Labels aus den Gangbeurteilungen vor. Die Labels verteilten sich zu 48,0 % und 52,0 % auf die Klassen „lahm“ bzw. „nicht lahm“. Die erwarteten Leistungen aller neun untersuchten Klassifikationsansätze waren sehr ähnlich. Lediglich die Ansätze E2E-MLP und AE-MLP hatten deutlich schlechtere Klassifikationsergebnisse. Insgesamt war die erwartete mittlere Klassifikationsleistung nur mäßig gut mit einer Genauigkeit und einer Relevanz von 0,71 sowie einer Sensitivität und einer Spezifität von 0,65 bzw. 0,75. Am besten klassifizierten FE-SVM, E2E-GRU, E2E-CNN und AE-CNN. Allerdings erreichte keiner dieser Ansätze eine erwartete Genauigkeit oder Relevanz von wenigstens 0,75. Der Arbeitsspeicherbedarf für das Training der Modelle lag bei allen untersuchten Ansätzen unter 3 GB, wobei nur die generativen Ansätze mehr als 1,5 GB benötigten. Bei allen diskriminativen Klassifikationsansätzen lag die Trainingsdauer unter fünf Minuten; alle generativen Ansätze mit unüberwachtem Vortraining benötigten deutlich mehr Zeit zum Erlernen der Modellparameter.

Für den praktischen Einsatz in Systemen zu automatischen Lahmheitserkennung an Hand von indirekten Variablen zur Aktivität, zur Leistung und zu den Eigenschaften der Kühe waren die erwarteten Klassifikationsleistungen aller neun Ansätze nicht ausreichend. Es besteht deshalb der Bedarf, weitere Modelle für diesen Anwendungsfall zu evaluieren und zu optimieren. Das größte Potential scheinen Ansätze mit CNN und RNN zu haben, da diese tiefen neuronalen Netze sehr flexibel an die spezifischen Anwendungsfälle angepasst werden können. Der Aufwand und der Ressourcenbedarf für die Entwicklung und das Training generativer Klassifikationsansätze standen in meinen Untersuchungen in keinem günstigen Verhältnis zum Zugewinn an Klassifikationsgenauigkeit gegenüber Ende-zu-Ende-Ansätzen mit vergleichbaren Modellen. Daher erscheint es nicht sinnvoll, für die automatische Lahmheitserkennung bei Milchkühen Ansätze mit unüberwachtem Vortraining zu entwickeln, solange ausreichend große Datensätze zum Training zur Verfügung stehen.

Abstract

Lameness is one of the most important health disorders of dairy cows. It is very time-consuming to detect lame cows reliably and early enough. For this reason, automatic lameness detection systems are being developed. Such systems are able to capture the cows' gait by different sensors like scales or cameras. Alternatively, existing data originating from automatic heat detection systems are used that describe the cows' behavior. These data are usually already aggregated and are called *indirect variables*. An essential part of automatic lameness detection systems is the algorithm which classifies the data that were measured in „lame“ and „not lame“.

The objective of my present thesis was to compare different classification algorithms to detect lameness from indirect variables describing the activity, the performance and characteristics of the cows. For my thesis, data of the project Klauen*fit*net were available including the indirect variables lactation number, days in milk, daily milk yield, average steps per hour and average lying duration per lying bout, and the labels „lame“ or „not lame“ according to the results of assessments of the cows' gait. The values of the variables were collected on a daily basis throughout the project period. The classification was based on time-dependent features like daily activity and milk yield as well as constant (for the observation period) variables like lactation number. The cows' gait had only been assessed approximately every two weeks. Therefore, not all observations were labelled and available for supervised learning methods.

I included discriminative and generative approaches for multivariate timeseries classification into the comparison. Additional to classical machine learning methods like random forests and Support Vector Machines (SVMs), deep learning methods like Multilayer Perceptrons (MLPs), Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) were applied. For the discriminative approaches, extra features were created by feature engineering. In the end-to-end approaches, the final features used for classification were learnt in deep learning models while the discriminative classifier was trained. In order to make the unlabeled data available for training the classifiers, unsupervised pretraining with autoencoders was applied to deep learning models in generative approaches. The comparison consisted in total of nine approaches: three approaches with feature engineering (FE-SVM, FE-RF, FE-MLP), three end-to-end approaches (E2E-MLP,

E2E-CNN, E2E-GRU), and three generative approaches with unsupervised pretraining (AE-MLP, AE-CNN, AE-GRU). The metrics accuracy, precision, sensitivity and specifity were calculated to assess the classification performance of the different approaches. In addition, I recorded the training and classification run times, and the amount of main memory that was required. The nine approaches were tested in ten-fold cross-validation based on the same splits of the data. For the analysis of the differences between the expected classification performances of the nine approaches under examination, I developed a graphical model as a Bayesian multivariate linear model. This solution made it possible to analyse the differences of the classification approaches in the four metrics simultaneously.

The used data set contained 727,008 observations. Of these, 17,114 observations had labels from the gait assessments. The labels were distributed to 48.0% and to 52.0% into the classes „lame“ and „not lame“, repectively. Expected preformances were similar among all nine classification approaches under consideration. Just the approaches E2E-MLP and AE-MLP had significantly worse classification results. Overall, the mean expected classification performance was only moderate with accuracy and precision of 0.71, 0.65 sensitivity, and 0.75 specifity. FE-SVM, E2E-GRU, E2E-CNN and AE-CNN classified the best. However, none of these approaches achieved an expected accuracy or precision of at least 0.75. Less than 3 GB of main memory were required for the training of all examined models, but solely the generative approaches needed more than 1.5 GB. Training duration of all discriminative classification approaches was below five minutes whereas all generative approaches with unsupervised pretraining needed considerably more time to learn the model parameters.

The expected classification performance of all nine approaches were not sufficient for practical application in automatic lameness detection systems using indirect variables of activity, performance and characteristics of the cows. Consequently, evaluation and optimisation of further models for this use case are required. Approaches with CNN and RNN seem to have the biggest potential because these deep neural networks can be adapted very flexibly to the specific use cases. The ratio of the effort and the ressources needed to develop and train generative classification approaches to the surplus in classification accuracy, compared to end-to-end approaches with similar models, was not beneficial in my studies. Therefore, developing approaches with unsupervised pretraining for the automatic detection of lameness in dairy cows does not seem to be sensible as long as enough data is available for training.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

AUC Area Under The Curve
CNN Convolutional Neural Network
CPU Central Processing Unit
ELU Exponential Linear Unit
GPU Graphics Processing Unit
GRU Gated Recurrent Unit
LSTM Long Short-Term Memory Neuronal Network
MLP Multilayer Perceptron
ROC Receiver Operator Characteristic
ROPE Region Of Practical Equivalence
RNN Recurrent Neural Network
SELU Scaled Exponential Linear Unit
SVM Support Vector Machine

1

Einleitung

Neben Fruchtbarkeitsproblemen und Euterentzündungen gehört Lahmheit zu den bedeutendsten gesundheitlichen Beeinträchtigungen von Milchkühen. Lahmheit ist eine „Störung des koordinierten Bewegungsablaufs an einer oder an mehreren Gliedmaßen [...]. Sie ist zurückzuführen auf schmerzhafte Prozesse (Entzündungen), mechan. Beeinträchtigungen der Gelenkfunktionen oder auf den Ausfall von Impulsen in den peripheren und zentralen motor. Nerven (Lähmungen).“ [WR00, S. 834]. Bei Milchkühen ist Lahmheit meistens von schmerzhaften Läsionen an den hinteren Klauen verursacht, die häufigsten Veränderungen sind Sohlengeschwüre, Weiße-Linie-Defekte und Dermatitis digitalis [ABH10].

Eine systematische Literaturstudie fand 128 Faktoren, die mit Lahmheit zusammenhängen. Darunter waren tierindividuelle Risikofaktoren wie die Anzahl von Laktationen oder der Laktationsstand aber auch Eigenschaften des Betriebs wie die Herdengröße [Oeh+19]. Da die Gesundheit der Klauen von der mechanischen Belastung, dem Stoffwechsel der Kuh und dem Keimdruck in der Umgebung abhängt, gehören die Gestaltung des Kuhstalls (insbesondere der Lauf- und Liegeflächen) und dessen Sauberkeit, die Fütterung und das Wasserangebot sowie die Reinigung und Pflege der Klauen zu den Bereichen mit großem Einfluss auf die Häufigkeit von Lahmheit bei Milchkühen [Cha+13; GGO18].

In Deutschland gehört Lahmheit zu den häufigsten krankheitsbedingten Ursachen, weshalb Milchkühe i. d. R. zur Schlachtung aus ihren Herden entfernt werden [FOS21; Pra20]. Eine Prävalenzstudie in niedersächsischen Milchkuhherden fand nur 48,1 % der Kühe mit unverändertem Gangbild, 17,8 % der Kühe waren klinisch lahm [Sch15]. Die mediane Prävalenz von Lahmheit innerhalb deutscher Milchkuhherden wurde in einer aktuellen Querschnittstudie mit 23,1 %, 39,1 % und 23,2 % für Nord-, Ost- bzw. Süddeutschland ermittelt [Jen+22]. Aus anderen Ländern wurden ähnliche Häufigkeiten berichtet, z. B. 20 % aus Alberta, Kanada [vHuy+20] und 31,6 % aus England und Wales [GGO18].

Lahmheit bei Milchkühen ist mit deutlichen Kosten für den landwirtschaftlichen Betrieb verbunden. Der Großteil der Kosten wird durch eine verringerte Fruchtbarkeit verursacht. Bereits geringgradige Lahmheit reduziert die Konzeptionsrate deutlich [Wie05]. Die schlechtere Fruchtbarkeit von lahmen Kühen wird hauptsächlich durch eine geringe Brunstintensität verursacht, was wiederum zu einer schlechteren Erkennung der Brunst und damit zu weniger erfolgreichen Be-

samungen führt. Die Brunstintensität lahmer Kühe kann einerseits durch niedrigere Hormonkonzentration verursacht werden, aber auch durch die physischen Einschränkungen in Folge der Lahmheit [Roe+10]. Desweiteren hat Lahmheit negative Auswirkungen auf die Milchleistung, was geringen Ertrag verglichen mit nicht-lahmen Kühen bedeutet [Kof+21; Pue+21; Vil+19].

Neben den wirtschaftlichen Auswirkungen hat Lahmheit nicht zuletzt gravierende Auswirkungen auf das Wohlergehen und das Verhalten der Milchkühe. Zusätzlich dazu, dass sie unter Schmerzen leiden, können lahme Kühe ihr natürliches Verhaltenrepertoire nicht mehr in ausreichendem Maße ausüben, was sich in verändertem Freß-, Liege- und Sozialverhalten äußert [Wei+18; WS17; Hut+21]. Lahme Kühe laufen weniger und liegen länger am Stück [Hut+21; Ito+10; YGB12].

Frühe Erkennung und Behandlung von Lahmheit reduziert die negativen Auswirkungen deutlich und verringert die Prävalenz von Lahmheit innerhalb der Herde. Die visuelle Beurteilung des Gangbilds der Kühe durch Betriebspersonal dient aktuell meistens zur Erkennung von lahmen Kühen [Dut+18]. Allerdings ist die Gangbeurteilung sehr arbeitsaufwändig und stark subjektiv, nicht sehr genau und schlecht reproduzierbar [Kan+20]. Kanadische Milchproduzentinnen und -produzenten sahen Lahmheit als das zweitwichtigste Gesundheitsproblem von Milchkühen an, das es zu kontrollieren galt. Die mit der Lahmheit verbundenen Schmerzen der Kühe waren ihnen ebenso bewusst wie die damit verbundenen Kosten [Hig+17]. Allerdings ist die Erkennungsquote durch Betriebspersonal nicht sehr hoch [Hig+17; Dut+20]. Ein falsches Verständnis davon, wie ein unverändertes Gangbild aussieht und was bereits Lahmheit ist, sowie die Verwendung ungeeigneter bzw. ungenauer Anzeichen für Lahmheit könnten Ursachen für die geringe Erkennungsrate sein, da die visuelle Gangbeurteilung Fachwissen und regelmäßiges Training erfordert [Hig+17]. Neben Schulungen könnten daher besonders technische Hilfsmittel die Erkennungsrate von Lahmheit bei Milchkühen deutlich erhöhen und so dazu beitragen, dass lahme Kühe früher behandelt werden und die Prävalenz von Lahmheit reduziert wird [Kan+20]. Sensoren, die an den Kühen befestigt sind, würden von dem meisten Milchproduzentinnen und -produzenten gegenüber Sensoren in Laufgängen und gegenüber Kameras bevorzugt [vdGuc+17]. Hinzu kommt, dass in vielen Milchkuhherden bereit Sensoren an den Kühen befestigt sind, um über die Überwachung der Aktivität der Kühe den Zeitpunkt der Brunst zu bestimmen [RH18]. Mit diesen Systemen werden bereits Aktivitätsdaten wie die Schrittfrequenz sowie die Liegehäufigkeit und -dauer bestimmt, die sich möglicherweise auch zur Erkennung von Lahmheit eignen.

1.1 Ansätze zur Automatisierung der Lahmheitserkennung bei Milchkühen

Ein System zur automatischen Lahmheitserkennung bei Milchkühen besteht aus mehreren Schichten [vgl. AFS19; OLe+20]:

1. der Erfassungsschicht mit Sensoren, die regelmäßig Rohdaten über die einzelnen Kühe liefern;

2. der Verarbeitungsschicht, in der sowohl die Rohdaten vorverarbeitet und ggf. zu Merkmalen zusammengefasst werden, die für die Lahmheitserkennung aussagekräftig sind, als auch die Beobachtungen der Kühe klassifiziert werden; und
3. der Ausgabeschicht zur Bereitstellung der Information über den Zustand der Kühe.

1.1.1 Erfassungsschicht

Ein System zur automatischen Lahmheitserkennung wird maßgeblich von der Art der Sensoren in der Erfassungsschicht bestimmt, für die so unterschiedliche Techniken wie Drucksensoren, Kameras oder Beschleunigungssensoren erprobt wurden [OLe+20]. Die Daten, die in der Erfassungsschicht erzeugt werden, wurden wiederholt in drei Kategorien eingeteilt [Sch+14; AFS19; Qia+21].

Kinetische Variablen

Kinetische Variablen beschreiben die Kräfte, die durch die Klauen der Kühe im Stand oder beim Gehen auf den Boden ausgeübt werden [Sch+14]. Die Kräfte können dabei in einer oder in mehreren Richtungen gemessen werden, sodass nicht nur der Druck nach unten auf den Boden, sondern auch horizontal wirkende Kräfte und Rotations- sowie Scherkräfte bestimmt werden können [AFS19; Qia+21]. Zur Erfassung werden Systeme mit zwei oder vier Sensoren (Waagen) oder druckempfindliche Folien im Boden von Laufgängen oder auch von Boxen von automatischen Melksystemen installiert [Sch+14; AFS19]. Durch die Verwendung von mehreren Sensoren können auch die Unterschiede zwischen den verschiedenen Beinen einer Kuh erfasst werden. Kinetische Variablen können den Stand und den Gang von Kühen sehr genau beschreiben. Daher lassen sich mit ihnen Lahmheiten mit hoher Genauigkeit erkennen [Sch+14]. Beispielsweise verwendeten Pastell und Kujala [PK07] vier Waagen in der Box eines automatischen Melksystems, welche die Druckbelastungen durch die vier Gliedmaßen der Kühe während des Melkens erfassten. Auf Grundlage der so erhobenen Daten gelang es ihnen, ein System zu entwickeln, das lahme Kühe mit einer Genauigkeit von 0,96 erkannte.

Allerdings können kinetische Variablen nicht kontinuierlich für alle Kühe erhoben werden, weil die entsprechenden Sensoren nur an bestimmten Stellen im Stall installiert werden können. Der Aufwand für die Installation der Sensoren in Kuhställen ist recht hoch, und nicht jeder Stall eignet sich für solche Systeme. Darüber hinaus müssen die Klauen richtig auf den Sensoren platziert werden, um verwendbare Daten zu erzeugen [AFS19], was nicht immer erreicht werden kann, wenn die Kühe sich artgerecht frei bewegen können.

Kinematische Variablen

Kinematische Variablen beschreiben räumlichen und zeitliche Veränderungen der Gliedmaßen oder allgemein bestimmter Körperteile der Kühe [Sch+14]. Dazu gehören u. a. Schrittlänge, Schrittfrequenz, Winkel von Gelenken in der Stützpha-

se und Kopfbewegungen. Zur Messung kinematischer Variablen wurden unterschiedliche Sensoren untersucht wie Kameras, 3D-Kameras, druckempfindliche Folien, Beschleunigungssensoren [AFS19], aber auch Radar [Bus+19] und Mikrofone [Vol+21].

Die enorme Weiterentwicklung der Bildverarbeitung und der computergestützten Mustererkennung ist sicher ein Grund dafür, dass in den letzten Jahren in vielen Studien versucht wurde, Bilddaten zur Lahmheitserkennung zu verwenden. Ein Schwerpunkt lag bisher auf der automatisierten Analyse der Rückenlinie [Pou+10; vHer+16; FSH18; Jia+22], aber auch die Bewegungen der Gliedmaßen wurden aus Bilddaten analysisert [Son+08; KZL20]. Durch die Analyse kinematischer Variablen aus Bilddaten können Lahmheiten mit sehr hoher Genauigkeit erkannt werden. Sowohl Kang, Zhang und Liu [KZL20] als auch Jiang u. a. [Jia+22] geben Klassifikationsgenauigkeiten von über 0,95 an (0,96 bzw. 0,97). Neben den spezifischen Problemen der Bildaufnahme und -analyse wie den Lichtverhältnissen und ähnlichen Farben und Strukturen in Vorder- und Hintergrund, die die Lahmheitserkennung *in praxi* erschweren, findet sich auch nicht in allen Ställen ein geeigneter Platz für die Kameras. Schließlich liefern stationäre Kameras auch nicht kontinuierlich Daten zu allen Kühen.

Beschleunigungssensoren, die an einem oder mehreren Beinen der Kühe oder am Hals angebracht sind, können ebenfalls kinematische Variablen messen. Sie werden bereits in vielen Betrieben zur Brunsterkennung eingesetzt. Diese Art der Sensoren würde von Landwirten und Landwirtinnen gegenüber den anderen Systemen bevorzugt [OLe+20]. Da die Beschleunigungssensoren an den Kühen angebracht sind, können sie kontinuierlich Daten zu allen Kühen liefern. Die zeitliche Auflösung der Messungen kann sich dabei stark unterscheiden, von unter 40 Hz bis zu 400 Hz [AFS19]. Die Autoren Alsaaod, Fadul und Steiner [AFS19] waren der Ansicht, dass 3D-Beschleunigungssensoren mit geringer Messfrequenz sich am besten für den Einsatz zur automatischen Lahmheitserkennung in Produktivsystemen eignen. In einer Literaturstudie beschrieben O'Leary u. a. [OLe+20] den Stand der Lahmheitserkennung mit Daten von Beschleunigungssensoren. Sie berichteten von Studien mit Lahmheitserkennungsystemen, die nach ihrer Ansicht praxistauglich waren, und die Lahmheiten mit einer Genauigkeit von bis zu 0,91 erkannten. Prototypen von Systemen, die nahe an der Praxistauglichkeit waren, wurden beispielsweise von Haladjian u. a. [Hal+18] und Taneja u. a. [Tan+20] vorgestellt mit Erkennungsgenauigkeiten von bis zu 0,91 bzw. von 0,87.

Aus den Daten von Beschleunigungssensoren können Variablen abgeleitet werden, die bestimmte Verhaltensweisen der Milchkühe beschreiben wie Stehen, Gehen, Liegen, Wiederkauen und Fressen [Mar+09]. Die so erzeugten Merkmale lassen sich wiederum zur Lahmheitserkennung weiter analysieren. Allerdings ist der Bezug dieser indirekten Variablen zum Vorkommen von Lahmheit geringer als von kinematischen Variablen, die direkte Bewegungen der Kühe beschreiben [OLe+20].

Indirekte Variablen

Indirekte Variablen sind Variablen, die die Kühe selbst oder ihr Verhalten beschreiben (z. B. Alter und Laktationsstand bzw. Liegedauer oder Wiederkaufrequenz)

oder mit denen die Leistung der Kühe gemessen wird, wie die Milchmenge pro Tag, und die als Indikatoren für Lahmheit verwendet werden können [Sch+14]. Diese Merkmale werden nicht von Sensoren in erster Linie für die Lahmheitserkennung erzeugt, sondern wurden zu einem anderen Zweck wie zur Brunsterkennung oder zur Leistungsbeurteilung erhoben und liegen bereits vor. Die Übersichtsstudie von O'Leary u. a. [OLe+20] beschrieb auch die Zusammenhänge von indirekten Merkmalen und dem Vorkommen von Lahmheit. Zwar wurde die Leistungsfähigkeit bei der Lahmheiterkennung von Kombinationen aus indirekten Variablen auch ohne kinetische oder kinematische Merkmale untersucht, aber diese Ansätze waren nicht sehr erfolgversprechend. Beispielsweise konnten Shahinfar u. a. [Sha+21] basierend auf rein indirekten Variablen Klassifikationsgenauigkeiten von bis zu 0,88 erzielen, allerdings mit geringer Sensitivität von höchstens 0,27. Meistens wurden indirekte Merkmale mit kinematischen Variablen kombiniert, um die Erkennung von Lahmheiten robuster und genauer zu machen [Qia+21]. Van Nuffel u. a. [vNuf+16] kamen zu dem Schluss, dass Systeme zur Lahmheitserkennung genauer werden, wenn sie Variablen wie Alter und Laktationsstatus der einzelnen Kühe zusätzlich zu kinematischen Variablen berücksichtigen. In der Studie von van Hertem u. a. [vHer+16] erreichten Modelle die größte Genauigkeit, die kamerabasierte kinematische Variablen und indirekte Variablen bei der Erkennung von Lahmheit berücksichtigten.

1.1.2 Verarbeitungsschicht

In der Verarbeitungsschicht findet die Vorverarbeitung der Rohdaten und anschließend die Klassifikation der Beobachtungen statt. Die Rohdaten werden zunächst zu Merkmalen zusammengefasst, die dann einem Klassifizierer zugeführt werden, der die Beobachtungen entsprechend der verwendeten Kategorien in „lahm“ oder „nicht lahm“ einteilt. Die Merkmale können z. B. die Schrittfrequenz oder die Krümmung der Rückenlinie sein. Die Datenvorverarbeitung hängt natürlich einerseits stark von den verwendeten Sensoren bzw. den Rohdaten ab und andererseits von den Erfordernissen der Klassifikationsalgorithmen, die zum Einsatz kommen. Werden ausschließlich indirekte Variablen analysiert, deren Beobachtungen aus anderen Systemen übernommen wurden, beschränkt sich die Vorverarbeitung möglicherweise auf das Zusammenführen der Daten aus unterschiedlichen Quellen.

Da die Sensoren kontinuierlich oder zumindest wiederholt Daten von den Kühen erfassen, liegen diese als Zeitreihen in unterschiedlicher zeitlicher Auflösung vor. Die Messwerte werden nicht nur räumlich (Bildpunkte oder Beschleunigung in unterschiedlichen Dimensionen), sondern auch zeitlich zusammengefasst, um aussagekräftige Merkmale zur Lahmheitserkennung zu generieren. Häufig erfolgt die Aggregation über mehrere Schritte und resultiert in Merkmalen mit einer zeitlichen Auflösung von einem Tag [OLe+20]. Beispielsweise fasste das kamerabasierte System, das von Piette u. a. [Pie+20] beschrieben wurde, die ursprünglichen Bilder zuerst zu einem Wert des Merkmals *Rückenkrümmung* pro Erfassung zusammen. Da die Kühe nach jedem Melken erfasst wurden und zweimal täglich gemolken wurden, lagen somit zuerst zwei Werte pro Tag vor, die dann zu einem Durch-

schnittswert pro Tag zusammengefasst wurden. Bei kontinuierlich erfassten Daten von Beschleunigungssensoren erfolgt eine erste Zusammenfassung für kürzere Intervalle wie z. B. 15 min [BOS21] oder eine Stunde [Gri+19]. Anschließend erfolgt ebenfalls eine Zusammenfassung pro Tag, bevor die Merkmale zur Klassifizierung verwendet werden.

Zur Vorverarbeitung der Daten wurden bisher fast ausschließlich algorithmische Verfahren verwendet, sowohl zur Vorverarbeitung von Bilddaten [vHer+16; FSH18] als auch von anderen Sensordaten [Als+12; vHer+13; Bee+16]. In neueren Studien wurden aber auch trainierbare Methoden verwendet, um die Merkmale zur Klassifizierung zu erzeugen [KZL20; Wu+20; Jia+22]. Bei diesen Studien erfolgte ebenfalls zuerst eine Vorverarbeitung und Zusammenfassung zu interpretierbaren Merkmalen wie Schrittlänge [Wu+20] oder Krümmung der Rückenlinie [Jia+22]. Echte Ende-zu-Ende-Verfahren, bei denen Vorverarbeitung und Klassifizierer zusammen die Rohdaten in einer geschlossenen Abfolge verarbeiten, ohne dass ggf. interpretierbare Merkmale erzeugt werden, wurden bisher nur selten beschrieben. Sowohl Wu u. a. [Wu+20] als auch Jiang u. a. [Jia+22] verglichen Verfahren, die zuerst interpretierbare Merkmale erzeugten, welche anschließend klassifiziert wurden, mit Ende-zu-Ende-Verfahren. In der Studie von Jiang u. a. [Jia+22] erwiesen sich die Vorhersagen des zweistufigen Verfahrens als genauer als die der getesten Ende-zu-Ende-Verfahren. Im Gegensatz dazu war der reine Deep-Learning-Ansatz bei Wu u. a. [Wu+20] etwas besser als das zweistufige Verfahren. Allerdings weisen die Autoren und Autorinnen darauf hin, dass Ende-zu-Ende-Verfahren schwieriger zu interpretieren bzw. deren Vorhersagen kaum nachvollziehbar sind.

Klassifizierer

Die Auswahl der Methode, mit der die Beobachtungen nach dem Vorliegen von Lahmheit klassifiziert werden, ist ein zentraler Punkt zur automatischen Erkennung von Lahmheit. Bisher hat sich dafür noch kein Standardverfahren etabliert. Allerdings ist auch nicht bekannt, wie groß der Einfluss der Klassifizierungsmethode auf die Leistungsfähigkeit des Gesamtsystems ist [OLe+20].

Werden Bilddaten als Grundlage zur automatischen Lahmheitserkennung verwendet, wird in Publikationen teilweise weniger Wert auf die Art und Beschreibung des Klassifizierers gelegt als auf die Kameraplatzierung und -technik und die Vorverarbeitung. So wurden von Poursaberi u. a. [Pou+10] und Fiolka, Schächter und Heinskill [FSH18] die Klassifizierer nicht genannt. Die Bandbreite beschriebener Klassifizierer reicht von Grenzwerten, die mittels Gittersuche festgelegt wurden [Pie+20] über logistische Regressionsmodelle [vHer+16] bis zu Long Short-Term Memory Neuronal Networks (LSTMs) und Varianten davon [Wu+20; Jia+22]. Zwar wurde von Wu u. a. [Wu+20] mit LSTM eine Genauigkeit von nahezu 0,99 erreicht, aber in derselben Studie schnitten Klassifikationsalgorithmen wie SVMs und k-Nearest Neighbours mit Genauigkeiten von 0,96 bzw. 0,95 nicht wesentlich schlechter ab. Der durch Ausprobieren festgelegte Grenzwert von Piette u. a. [Pie+20] (Genauigkeit: 0,79) übertraf die Klassifikationsgenauigkeit der logistischen Regression von van Hertem u. a. [vHer+16] (Genauigkeit: 0,60).

Laut der Literaturübersicht von O'Leary u. a. [OLe+20] sind logistische Regression und SVMs die bedeutendsten Verfahren bei der Verwendung von Daten aus Beschleunigungssensoren. Eine eigene Literaturrecherche zur Bewertung von Klassifikationsalgorithmen in der automatischen Lahmheitserkennung bei Milchkühen am 28. Mai 2021 in Pubmed[1] und GoogleScholar[2] konnte diese Einschätzung teilweise bestätigen. Die Literatursuche beschränkte sich auf Studien, die indirekte Variablen zur Lahmheitserkennung verwendeten, welche die Aktivität bzw. das Verhalten der Kühe beschreiben. Zwar habe ich nur eine Studie gefunden, in der eine SVM verwendet wurde, aber fünf der elf gefundenen Studien verwendeten logistische Regressionsmodelle in verschiedenen Varianten (Tabelle 1.1 auf der nächsten Seite). In den Studien zur Lahmheitserkennung aus Aktivitätsdaten erreichten die Klassifizierer nicht ganz die Leistungen wie in den Studien, in denen Bilddaten verwendet wurden. Es wurden bisher keine Deep-Learning-Verfahren zur Klassifizierung verwendet, um auf der Grundlage von Aktivitätsdaten Lahmheit bei Milchkühen zu erkennen.

1.1.3 Ausgabeschicht

Die Ausgabeschicht informiert die Landwirtin bzw. den Landwirt über das Auftreten von Lahmheit bei den einzelnen Kühen. In den wenigsten wissenschaftlichen Publikationen zur automatischen Lahmheitserkennung bei Milchkühen wurde diese Schicht beschrieben oder wurden Entwürfe für diese Schicht vorgestellt. Auch in den Übersichtsarbeiten von Alsaaod, Fadul und Steiner [AFS19] und O'Leary u. a. [OLe+20] finden sich keine Angaben zu Entwürfen der Ausgabeschicht. Die meisten Systeme, die in wissenschaftlichen Publikationen vorgestellt wurden, waren noch nicht praxisreif [AFS19], oder es wurden nur einzelne Aspekte der Automatisierung der Lahmheitserkennung untersucht. Die Arbeiten von Haladjian u. a. [Hal+18] und Taneja u. a. [Tan+20] stellen insofern Ausnahmen dar, als sie Prototypen kompletter Systeme von der Erfassungs- bis zur Ausgabeschicht umfassten.

Allerdings gaben Haladjian u. a. [Hal+18] nur Ausblicke und Ideen zur Ausgabeschicht in ihrer Beschreibung an. Sie schlugen vor, dass die Klassifikationsergebnisse über die Melktechnik erfasst werden sollten. Die Bewertungen pro Kuh und Tag könnten dann in einer Erweiterung der Melktechnik- oder der Herdenmanagementsoftware angezeigt werden.

Im Gegensatz dazu erwähnten Taneja u. a. [Tan+20] einen Prototypen für ein Cloud Dashboard und eine mobile Software zur Kommunikation der Klassifikationsergebnisse. Sie sahen bereits Schnittstellen vor, um ihr System für Klassifikationsergebnisse aus anderen Systemen (z. B. zur automatischen Erkennung von Euterentzündungen) oder andere Software zur Benachrichtigung der Landwirtinnen bzw. Landwirte zu öffnen.

[1] `https://pubmed.ncbi.nlm.nih.gov/`
[2] `https://scholar.google.de/`

Tabelle 1.1: Literaturübersicht über die Klassifikationsleistungen verschiedener Ansätze zur Lahmheitserkennung bei Milchkühen mit indirekten Variablen aus Aktivitätsdaten

Klassifikationsalgorithmus	Leistung			Quelle
	Genauigkeit	Sensitivität	Spezifität	
SVM	0,76			[Als+12]
Logistische Regression	0,86	0,89	0,85	[vHer+13]
Additive logistische Regression mit Regressionbäumen		0,41/0,57	[i]	[Kam+13]
Statistische Kontrollkarten		0,88	0,61	[MTK13]
Dynamisches lineares Modell		0,80	0,90	[dMol+13]
Partial Least Squares Discriminant Analysis	0,77/0,81[ii]	0,79/0,79[ii]	0,77/0,83[ii]	[Gar+14]
Logistische Regression		0,93	0,92	[Bee+16]
Elastic Net Logistic Regression		1,0	0,8	[Sch+17]
k-Nearest Neighbours	0,87[iii]	0,90[iii]	0,73[iii]	[Bya+19]
Elastic Net Logistic Regression		0,94	0,81	[Gri+19]
Gradient Boosted Decision Trees	0,78	0,78	0,78	[BOS21]

[i] Bei festgelegter Spezifität von 0,9 bzw. 0,8.
[ii] Für Kühe in der ersten bzw. zweiten Laktation.
[iii] Drei Tage vor dem Auftreten sichtbarer Lahmheit.

1.2 Aufgabenbeschreibung und Ziele

Wie von Wolpert [Wol96] gezeigt, ist kein Klassifikationsmodell grundsätzlich besser als ein anderes, wenn der Vergleich ohne Annahmen über die verwendeten Daten durchgeführt wird. Diese Feststellung wurde als *No Free Lunch Theorem* bekannt. Es gibt keine andere Möglichkeit, gut geeignete Klassifikationsansätze für eine bestimmte Aufgabe zu finden, als unterschiedliche Ansätze auszuprobieren. Da nicht alle möglichen Ansätze ausprobiert werden können, ist es wichtig, vielversprechende Kandidaten auszuwählen durch Kenntnis sowohl der jeweiligen Klassifikationsverfahren als auch der Eigenschaften der Daten und der fachlichen Hintergründe [vgl. Gér20, S. 34; DHS01, S. 457–458].

Das Ziel meiner vorliegenden Arbeit war der Vergleich von verschiedenen Klassifikationsalgorithmen zur Erkennung von Lahmheit an Hand von indirekten Variablen zur Aktivität, zur Leistung und zu Eigenschaften der Kühe. In den Vergleich sollten Verfahren des klassischen maschinellen Lernens wie baumbasierte Verfahren und SVMs, aber auch Methoden des Deep Learning einbezogen werden, um wenigstens ein Verfahren zu identifizieren, welches gut zu den zur Verfügung ste-

henden Daten passt [vgl. DHS01, S. 458]. Es sollte mit der Klassifikation (und ggf. zusätzlich notwendigen Datenvorverarbeitungsschritten) ein einzelner Aspekt innerhalb der Verarbeitungsschicht der automatischen Lahmheitserkennung untersucht werden unabhängig von möglichen Gesamtsystemen, aber bezogen auf eine konkrete Auswahl von Merkmalen.

Im Rahmen des Projekts Klauenfitnet[3] waren auf sieben deutschen Milchviehbetrieben jeweils über ein halbes Jahr Daten erhoben worden zu insgesamt etwa 3500 Milchkühen [fLei19]. Aus diesem Projekt standen die Daten zur täglichen Aktivität der Kühe (Schrittfrequenz, Liegehäufigkeit und -dauern), die tägliche Milchleistung sowie die Rasse, Laktationsnummer und die Kalbedaten zur Verfügung. Ergänzt wurden diese Daten durch die Ergebnisse der Gangbeurteilung nach Sprecher, Hostetler und Kaneene [SHK97] in zweitwöchentlichen Abständen.

Die Ergebnisse der Gangbeurteilung sollten als Labels zum Trainieren und Testen verschiedener Klassifikationsmodelle dienen, die die täglichen Beobachtungen der Variablen zur Aktivität und Leistung der Kühe verarbeiten. Die Kombination von kinematischen Variablen wie der Schrittfrequenz mit der Milchleistung sollten Erkennung von Lahmheit bei Milchkühen robuster und genauer machen [Qia+21]. Die Genauigkeit sollte durch die Kombination mit kuhindividuellen Merkmalen wie der Laktationsnummer und dem Laktationsstatus weiter erhöht werden [vNuf+16]. Als Merkmale zur Klassifikation sollten somit sowohl zeitabhängige Variablen wie die tägliche Aktivität und Milchmenge als auch (im Beobachtungszeitraum) konstante Variablen wie die Laktationsnummer verwendet werden. Die Beobachtungen bestanden damit aus einer Kombination aus mehreren Zeitreihen und Einzelwerten. Da nur etwa in 14-tägigen Abständen das Gangbild beurteilt worden war, waren die Labels der Beobachtungen unvollständig und standen für überwachte Lernverfahren nur eingeschränkt zur Verfügung. Neben Verfahren zur Zeitreihenklassifikation aus dem Bereich des überwachten Lernens sollten daher zusätzlich halbüberwachte Lernverfahren einbezogen werden, um die Beobachtungen mit fehlenden Labels zum Training der Klassifikationsmodelle nutzen zu können. Die Klassifikationsleistung und der Resourcenbedarf der trainierten Klassifikationsmodelle sollten anschließend verglichen werden.

1.2.1 Verwandte Arbeiten

Klassifikationsmodelle können entweder allgemein oder bezogen auf einen speziellen Anwendungsfall verglichen werden. Das Ziel eines allgemeinen Vergleichs ist es, die theoretischen Leistungsfähigkeiten verschiedener Modelle zu beurteilen. Dabei erfolgt der Vergleich i. d. R. mit sehr unterschiedlichen Datensätzen, um die Leistungsfähigkeiten für möglichst viele unterschiedliche Anwendungsszenarien einzubeziehen. Als Maße der Leistungsfähigkeit bieten sich in diesem Fall grenzwertunabhängige Metriken an wie die Area Under The Curve (AUC), die Fläche unter der Receiver Operator Characteristic (ROC) [Hum+20]. Im zweiten Fall werden verschiedene Klassifikationsmodelle verglichen, um für eine konkrete Anwendung das geeignetste zu identifizieren. Daher wird in diesem Fall häufig nur

[3] `https://www.klauenfitnet.de/programm/klauenfitnet-10/`, abgerufen am 17. Mai 2022

ein Datensatz für den Vergleich verwendet. Der Vergleich erfolgt sinnvollerweise über grenzwertabhängige Metriken wie Genauigkeit, Sensitivität und Spezifität [Hum+20].

Allgemeine Vergleiche von Klassifikationsmodellen für Zeitreihen

Vergleiche der theoretischen Leistungsfähigkeit verschiedener Modelle ohne konkreten Anwendungsbezug werden oft durchgeführt, wenn eine neue Modellvariante vorgestellt wird. Die neue Modellvariante wird mit bereits publizierten und angewandten Modellen verglichen, um einordnen zu können, ob sie einen Fortschritt darstellt. Beispielsweise verglichen Dempster, Petitjean und Webb [DPW20], Fawaz u. a. [Faw+20] und Khan u. a. [Kha+20] ihre neuen Modellvarianten jeweils mit den bis dahin leistungsfähigsten Klassifikationsmodellen für Zeitreihen. Allerdings wird in der vorliegenden Arbeit keine neue Modellvariante vorgestellt, daher werden diese Beispiele nicht näher ausgeführt.

Reine Vergleiche von Zeitreihenklassifizierern ohne die Präsentation eines neuen Modells finden sich in den Artikeln von Fawaz u. a. [Faw+19] und [Rui+21]. Die Studie von Fawaz u. a. [Faw+19] beschränkte sich auf den Vergleich von diskriminativen (unterscheidenen) Deep-Learning-Modellen zur Zeitreihenklassifizierung in Ende-zu-Ende-Ansätzen. Jedes Modell wurde jeweils zehnmal mit unterschiedlichen Datensätzen trainiert (85 Datensätze mit univariaten Zeitreihen und 12 Datensätze mit multivariaten Zeitreihen). Die Aufteilung in Trainings- und Testdaten war für alle Durchläufe eines Datensatzes immer dieselbe, allerdings wurden die Modelle jedes Mal mit anderen zufälligen Gewichten initialisert. Als Vergleichsmetrik wurde die Genauigkeit bestimmt und mittels Critical-Difference-Diagrammen nach Demšar [Dem06] für den Vergleich mehrerer Modelle mit mehreren Datensätzen beurteilt.

Einen anderen Schwerpunkt setzten Ruiz u. a. [Rui+21] in ihrem Vergleich, den sie auf Klassifikationsmodelle für multivariate Zeitreihen beschränkten. In ihren Vergleich bezogen sie neben Deep-Learning-Modellen auch Ansätze ein, die auf Shapelets oder Bags Of Words basierten. Die Modelle wurden mit 26 verschiedenen Datensätzen trainiert und getestet. Jedes der Modelle wurde auf jeden Datensatz 30 Mal angewandt, wobei für jeden Durchlauf eine unterschiedliche Aufteilung in Trainings- und Testdaten erfolgte. Zum Vergleich wurden verschiedene Metriken berechnet einschließlich Genauigkeit und AUC. Mit statistischen Tests wie dem Wilcoxon-Rank-Test und Critical-Difference-Diagrammen wurden mögliche Unterschiede zwischen den Leistungen der verschiedenen Klassifikationsmodelle untersucht. Zusätzlich wurden die Laufzeiten und der Arbeitsspeicherbedarf für das Training der unterschiedlichen Modelle deskriptiv verglichen.

Im Gegensatz zu den soeben vorgestellten Studien sollte in der vorliegenden Arbeit kein allgemeiner Vergleich der theoretischen Leistungsfähigkeit verschiedener Modelle zur Zeitreihenklassifizierung erfolgen, sondern es sollten unterschiedliche Ansätze hinsichtlich ihrer Eignung als Klassifizierer in einem System zur automatischen Lahmheitserkennung von Milchkühen beurteilt werden. Die Anforderungen wurden weiter konkretisiert, indem die ursprünglichen Merkmale, an Hand derer

die Klassifizierung erfolgen sollte, durch die vorhandenen Daten bereits festgelegt waren. Daher sind die Methoden und Ergebnisse von Fawaz u. a. [Faw+19] und Ruiz u. a. [Rui+21] nur eingeschränkt mit denjenigen der vorliegenden Arbeit vergleichbar.

Vergleiche von Klassifikationsmodellen zur automatisierten Lahmheitserkennung bei Milchkühen

Bisher wurden kaum Vergleichsstudien von Klassifikationsmodellen für die Analyse von Aktivitätsdaten zur Lahmheitserkennung bei Milchkühen veröffentlicht. Meistens wurden die zur Verfügung stehenden Daten mit einem Modell klassifiziert und Metriken für dessen Leistungsfähigkeit berechnet. In Tabelle 1.1 auf Seite 8 finden sich Beispiele dafür. Natürlich ist es wahrscheinlich, dass in den Vorbereitungen der dort aufgeführten Arbeiten auch unterschiedliche Modellansätze ausprobiert worden waren, aber diese Versuche sind nicht in den Veröffentlichungen enthalten. In ihrer Publikation erwähnten Taneja u. a. [Tan+20] immerhin, dass sie verschiedene Klassifizierer ausprobiert hatten (SVM, Random Forest, k-Nearest Neighbours und Gradient Boosted Decision Trees). Diese wurden mittels ihrer Genauigkeit verglichen. Allerdings wurden weder die Vergleichsmethodik noch die Ergebnisse vollständig beschrieben.

Die Studien von Lasser u. a. [Las+21] und Shahinfar u. a. [Sha+21] verglichen zwar Klassifikationsmodelle zur Lahmheitserkennung bei Milchkühen an Hand von indirekten Variablen, allerdings wurden bei beiden keine Aktivitätsdaten einbezogen. Lasser u. a. [Las+21] untersuchten die Klassifikationsleistung von logistischer Regression, Random Forests und Gradient Boosted Decision Trees zur Erkennung von acht Erkrankungen von Milchkühen, darunter auch Lahmheit. Dazu verwendeten sie etliche indirekte Variablen aus verschiedenen Bereichen wie Leistungsdaten und Eigenschaften der Kühe, aber auch Eigenschaften der Betriebe. Die Klassifikationsmodelle wurden mit vier Datensätzen mit unterschiedlichen Merkmalskombinationen und unterschiedlicher Auswahl der beobachteten Kühe trainiert und getestet. Pro Datensatz erfolgte eine zehnfache Kreuzvalidierung auf Beobachtungs-, aber auch auf Einzeltier- und Herdenebene. Mit verschiedenen grenzwertabhängigen Metriken wie der Genauigkeit wurden die Leistungen der Klassifizierer deskriptiv verglichen.

Ebenfalls im Vergleich enthalten waren bei der Studie von Shahinfar u. a. [Sha+21] logistische Regression und Random Forests, zusätzlich wurden MLPs untersucht. Ziel dieser Studie war die Erkennung von Lahmheit bei Milchkühen auf der Grundlage von Leistungsdaten und körperlichen Merkmalen, die für die Zuchtwertschätzung erhoben worden waren. Die Modelle wurden in einer zehnfachen Kreuzvaldierung mit einem Datensatz trainiert und getestet. Dabei wurden jeweils 20 % der Daten jeder Herde als Testdaten verwendet. Zum Vergleich wurden verschiedene grenzwertabhängige Metriken wie die Genauigkeit aber auch AUC berechnet. Der Vergleich erfolgte mittels Tukey's Honestly-Significant-Difference-Test.

Da zwar sowohl Lasser u. a. [Las+21] als auch Shahinfar u. a. [Sha+21] indirekte Variablen zur automatischen Lahmheitserkennung heranzogen, dabei aber Aktivitätsdaten nicht unberücksichtigt ließen, die beispielsweise im Zusammenhang mit der Brunsterkennung erhoben wurden, sind auch diese beiden Arbeiten nicht direkt mit der vorliegenden Studie vergleichbar.

1.2.2 Besonderheiten dieser Arbeit

In der vorliegenden Arbeit wurden verschiedenene Klassifikationsmodelle verglichen zur Erkennung von Lahmheit an Hand von indirekten Variablen zur Aktivität, zur Leistung und zu Eigenschaften der Kühe. Dabei wurden sowohl diskriminative als auch generative Ansätze zur Klassifikation von multivariaten Zeitreihen einbezogen. Außerdem wurden neben Verfahren des klassischen maschinellen Lernens, wie baumbasierten Verfahren und SVMs, auch Methoden des Deep Learning, wie CNNs und RNNs, berücksichtigt. Um zusätzliches Wissen in die Trainingsdaten einzubeziehen, wurden in diskriminativen Ansätzen mittels Feature Engineering zusätzliche Merkmale erzeugt [vgl. vRMB+20]. Diese zusätzlichen Merkmale dienten dann zusammen mit den ursprünglichen Merkmalen zur Klassifizierung der Datensätze mit Methoden des klassischen maschinellen Lernens sowie mit MLP. Bei Ende-zu-Ende-Ansätzen erfolgte die Entwicklung der endgültigen Merkmale zur Klassifizierung während des und zusammen mit dem Training eines diskriminativen Klassifizierers in Deep-Learning-Modellen [Faw+19]. Bei vielen der zur Verfügung stehenden Daten fehlten die Labels, d. h. die Angabe, ob die betreffende Kuh am beobachteten Tag lahm war oder nicht. Um diese ungelabelten Daten ebenfalls für das Training der Klassifizierer zu verwenden, wurden in generativen Ansätzen Deep-Learning-Modelle mit Autoencodern unüberwacht vortrainiert [vgl. Gér20, S. 352]. Zur Bewertung der Klassifikationsleistung der verschiedenen Ansätze und Modelle wurden grenzwertabhängige Metriken (Genauigkeit, Relevanz, Sensitivität und Spezifität) berechnet, weil die Leistungsbeurteilung für eine konkrete Aufgabe mit festgelegten Merkmalen erfolgte [vgl. Hum+20]. Der Vergleich aller vier Metriken erfolgte mit einem Bayes'schen multivariaten Modell wie von Benavoli u. a. [Ben+17] propagiert. Die Laufzeit zum Training der Modelle und der Arbeitsspeicherbedarf wurden in die Studie einbezogen, weil zur Bewertung der Eignung verschiedener Klassifikationsansätze für die automatische Lahmheitserkennung bei Milchkühen nicht nur die Klassifikationsleistung wichtig ist, sondern auch der Ressourcenbedarf der Verfahren.

1.3 Übersicht über die folgenden Kapitel

Der Rest der vorliegenden Arbeit gliedert sich in die Kapitel *Material und Methoden*, *Ergebnisse* und *Diskussion*.

Das nächste Kapitel *Material und Methoden* beginnt mit einer Beschreibung, wie die Daten erhoben und zusammengeführt wurden, die für den Vergleich der Klassifikationsansätze verwendet wurden. Anschließend werden die untersuchten

Klassifikationsansätze erläutert. Im dritten Abschnitt des Kapitels werden die Kriterien dargestellt, an Hand derer die Ansätze verglichen wurden, sowie der experimentelle Aufbau und das Softwareframework für den Vergleich detailliert beschrieben. Zum Abschluss des Kapitels wird das statistische Modell vorgestellt, mit dem die Leistungen der Klassifikationsansätze analysiert wurden.

Zu Beginn des Kapitels *Ergebnisse* werden die verwendeten Daten und die Hyperparameter der Klassifikationsmodelle beschrieben. Darauf folgen die Beschreibungen der Klassifikationsleistungen, die von den untersuchten Ansätzen erzielt wurden, ihres Ressourcenbedarfs und schließlich die Schätzungen des statistischen Modells für die zu erwartenden Klassifikationsleistungen.

Im Kapitel *Diskussion* werden die Innovationen und Einschränkungen der vorliegenden Arbeit in den Kontext bisher publizierter Studien eingeordnet. Die Ergebnisse der Arbeit werden in Beziehung zueinander gesetzt und mögliche Erklärungen werden erörtert. Zusätzlich werden Anregungen für weitere Studien gegeben. Mit einer Zusammenfassung der wesentlichen Schlussfolgerungen aus den Untersuchungen endet das Kapitel.

2

Material und Methoden

Im diesem Kapitel werden zuerst die Erhebung und Vorbereitung der verwendeten Daten beschrieben. Anschließend werden die untersuchten Ansätze zur Klassifikation im Detail dargestellt. Schließlich werden die Kriterien und der experimentelle Aufbau zur Bestimmung der Leistungen der Klassifikationsansätze erläutert. Den Abschluss bildet die genaue Beschreibung des statistischen Modells zum Vergleich der Klassifikationsansätze.

2.1 Verwendete Daten

Für den Vergleich verschiedener Ansätze zur multivariaten Zeitreihenklassifikation für die automatische Lahmheitserkennung im Rahmen dieser Arbeit standen Daten zur Verfügung, die während des Projekts „Entwicklung einer Dienstleistung zur Verbesserung der Klauengesundheit von Milchkühen durch Vernetzung und Verdichtung von Daten für das Tiergesundheitsmanagement“ (Klauen*fit*net)[1] erhoben worden waren. Das Projekt Klauen*fit*net hatte eine Laufzeit von 1. März 2015 bis 31. August 2018 und war vom Bundesministerium für Ernährung und Landwirtschaft über den Projektträger Bundesanstalt für Ernährung im Rahmen des Förderprogramms Innovationsförderung unterstützt worden. Die Projektkoordination erfolgte durch den Deutschen Verband für Leistungs- und Qualitätsprüfungen e.V. (DLQ) [fLei19].

2.1.1 Klauen*fit*net-Daten

Während des Projekts Klauen*fit*net waren Daten auf sieben Milchviehbetrieben in Ost- und Süddeutschland mit sehr unterschiedlichen Herdengrößen und -zusammensetzungen erhoben worden. Alle Betriebe hatten die ermolkene Milchmenge pro Kuh und Tag (das Tagesgemelk) erfasst. Pro Kuh war ein Bein mit einem Differential-Präzisionspedometer der Firma Lemmer Fullwood GmbH, Lohmar versehen gewesen, das Beschleunigungssensoren enthält. Die Differential-Präzisionspedometer hatten die Datengrundlage für die Berechnung der täglichen Aktivitätsdaten (durchschnittliche Schrittfrequenz und Liegedauer) geliefert. Die

[1] `https://www.klauenfitnet.de/`, besucht am 24. Mai 2022

Tagesgemelke und Aktivitätsdaten waren über das FULLEXPERT® Herdenmanagementsystem[2] von Lemmer Fullwood bereitgestellt worden. Zusätzlich waren für alle Kühe Daten aus den Milchleistungsprüfungen abgerufen worden (u. a. Rasse, Kalbedaten und Laktationsnummern) [fLei19].

In allen Herden war der Gang der Kühe nach Sprecher, Hostetler und Kaneene [SHK97] visuell beurteilt worden durch eine Tierärztin und einen Tierarzt, die ihre Bewertung wiederholt abgestimmt hatten. Für die Gangbeurteilung hatten sie ein einheitliches Schema mit Beispielbildern für jede Bewegungsnote verwendet (Abbildung 2.1 auf der nächsten Seite). Die Gangbeurteilung war in jeder Herde für sechs Monate ab Oktober 2015 in 14-tägigem Abstand durchgeführt und die Ergebnisse waren für jede Kuh elektronisch gespeichert worden. Insgesamt konnten nach Abschluss der Datenerhebung Beobachtungen von ca. 3.500 Milchkühen genutzt werden [fLei19].

Für die vorliegende Arbeit standen die Klauen*fit*net-Daten in verschiedenen Textdateien zur Verfügung. Die Daten zu den Merkmalen *Betrieb*, *Kuh*, *Laktationsnummer*, *Laktationstag*, *Tagesgemelk*, *durchschnittliche Schrittfrequenz pro Tag* und *Liegedauer pro Liegevorgang* sowie die Ergebnisse der Gangbeurteilung wurden in einem Datensatz zusammengefasst. Dabei wurden die Beobachtungen von einem Betrieb ausgeschlossen, weil nur sehr wenige Werte der Liegedauer vorhanden waren. Die Ergebnisse der Gangbeurteilung, die auf einer fünfstufigen Skala vorlagen, wurden als Klassenlabels dichotomisiert. Beobachtungen mit einer Note von drei bis fünf wurden als „lahm“ eingestuft, Beobachtungen mit den Noten eins und zwei als „nicht lahm“ (siehe Abbildung 2.1 auf der nächsten Seite).

Herausforderungen der Daten

Eine Beobachtung wurde durch einen Laktationstag in einer Laktation(snummer) einer Kuh definiert. Die einzelnen Beobachtungen konnten nicht als unabhängig voneinander behandelt werden, sondern sie lagen in einer hierarchischen Struktur mit zeitlicher und räumlicher Gruppierung vor. Die Laktationstage bildeten Zeitreihen innerhalb einer Laktation einer Kuh. Pro Kuh konnten wiederum mehrere Laktationen beobachtet worden sein. Schließlich waren die Kühe innerhalb der Betriebe räumlich gruppiert. Es lagen keine Beobachtungen von ein und derselben Kuh aus unterschiedlichen Betrieben vor. Um die Struktur der Daten und die möglichen unbeobachteten Einflüsse des Laktationsstadiums und der Laktationsnummer zu berücksichtigen, wurden die Merkmale *Laktationstag* und *-nummer* in die Modelle aufgenommen. Für die Merkmale *Kuh* und *Betrieb* erschien dieses Vorgehen nicht sinnvoll, weil ein System zur automatischen Lahmheitserkennung Kühe eines einzelnen Betriebs beurteilen muss, ohne Vorkenntnisse über die Kühe oder den Betrieb zu haben. Es wurde daher angestrebt, die Modelle unabhängig vom Merkmal *Kuh* zu trainieren. Dazu wurden die Daten auf der Ebene der Kühe in Trainings- und Testdaten eingeteilt, d. h. sämtliche Beobachtungen einer Kuh wurden entweder dem Trainings- oder dem Testdatensatz zugewiesen. Um die

[2] `https://www.lemmer-fullwood.info/loesungen/herdenmanagement/fullexpert/`, abgerufen am 2022-05-30

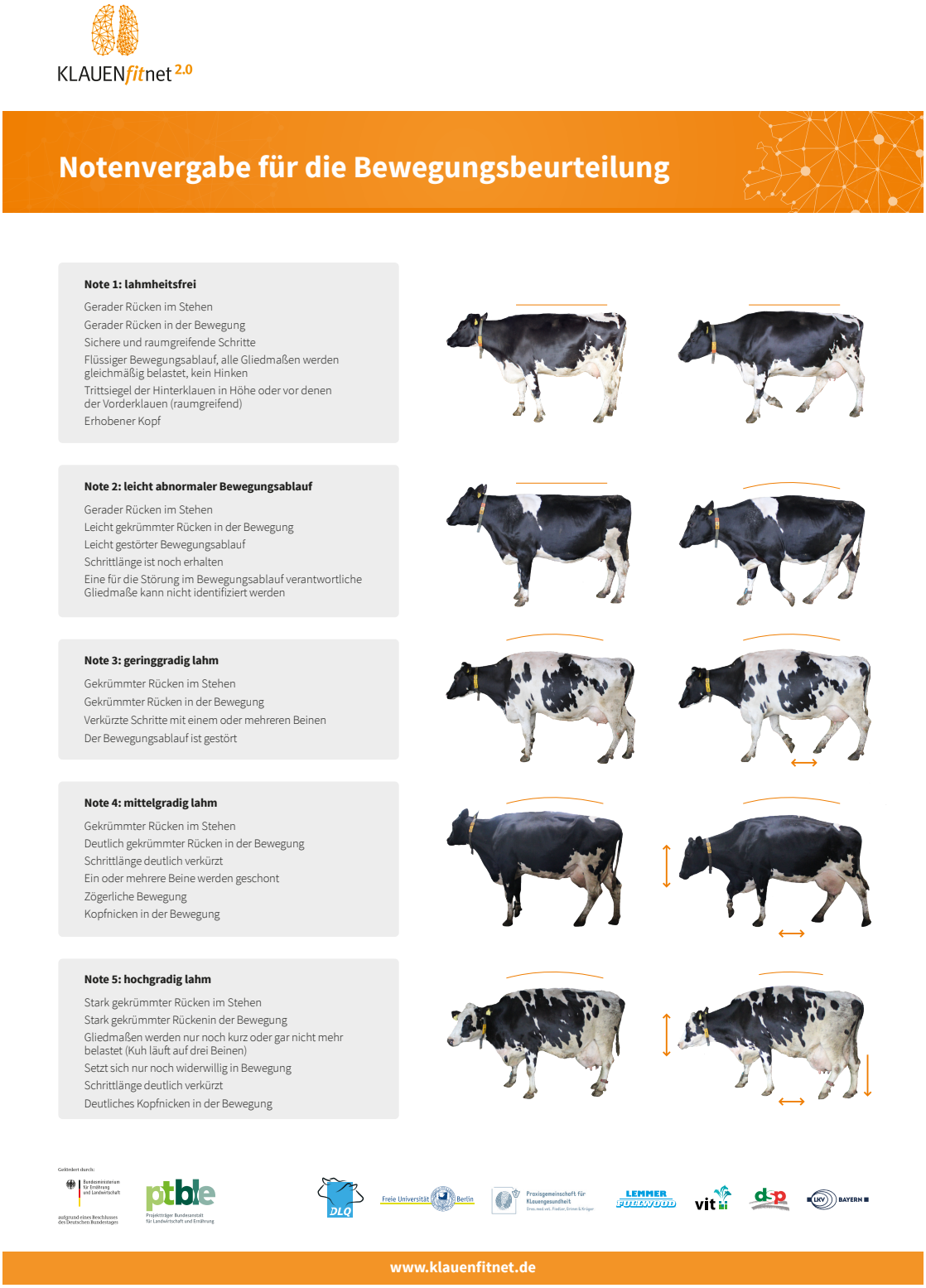

Abbildung 2.1: Schema zur Gangbeurteilung bei Kühen nach Sprecher, Hostetler und Kaneene [SHK97] im Rahmen des Projekts Klauen*fit*net [Kla, S. 2, Abdruck genehmigt am 2022-07-04]

(unbeobachteten) Einflüsse der Betriebe möglichst gering zu halten, wurden nur die Daten von Betrieben mit Herden aus mindestens 75 % schwarzbunten Kühen verwendet. Daten von zwei Betrieben mit überwiegend Fleckviehkühen wurden ausgeschlossen. Die verbliebenen Beobachtungen wiesen nur geringe Unterschiede zwischen den Betrieben auf.

Die Klauen*fit*net-Daten waren aus unterschiedlichen Quellen zusammengeführt worden (Milchmengenerfassung, Aktivitätsmessung, Milchleistungsprüfung und Gangbeurteilung). Nicht in allen Quellen waren die Daten zuvor Plausibilitätsprüfungen unterzogen worden. Zusätzlich konnten sich durch die Zusammenführung und Berechnung des Laktationstags aus Beobachtungsdatum und Kalbedatum falsche Werte ergeben. Daher lagen auch Beobachtungen mit fachlich unplausiblen Werten vor. Alle Beobachtungen mit einem Laktationstag über dem Minimum von 500 und dem Laktationstag der letzten Beobachtung mit dokumentiertem Tagesgemelk innerhalb der Laktation einer Kuh wurden entfernt. Anschließend wurden alle Laktationen einer Kuh mit weniger als 27 Beobachtungen vollständig ausgeschlossen. Extreme Werte der Schrittfrequenz und der Liegedauer unter bzw. über dem 0,005- bzw. 0,995-Quantil wurden durch die entsprechenden Quantile ersetzt.

Damit die Zeitreihen der Merkmalswerte klassifiziert werden konnten, mussten sie vollständig vorliegen, d. h. es durfte keine Lücken durch fehlende Werte innerhalb einer Reihe von Beobachtungen geben. Um diese Voraussetzung zu erfüllen, wurden zuerst alle Beobachtungen von Laktationen ausgeschlossen, bei denen nicht jeweils für wenigstens 67 % der Beobachtungen Werte für das Tagesgemelk, die Schrittfrequenz und die Liegedauer vorlagen. Anschließend wurden fehlende Werte dieser Merkmale mittels linearer Interpolation imputiert. Es wurden maximal zehn fehlende Werte in Folge interpoliert. Zuletzt wurde pro Laktation einer Kuh nur die längste zusammenhängende Folge von Beobachtungen mit vollständigen Werten behalten, die mindestens 27 Tage umfasste.

Für die Klassifikation standen Beobachtungen der fünf Variablen *Laktationsnummer* (`Lactation`), *Laktationstag* (`DaysInMilk`), *Tagesgemelk* (`MilkYield`), *durchschnittliche Schrittfrequenz* (`StepsPerHour`) und *durchschnittliche Liegedauer pro Liegevorgang* (`LyingDuration`) zur Verfügung. Weil die Gangbeurteilung nur etwa alle zwei Wochen erfolgt war, waren nicht alle Beobachtungen mit einem der Labels „lahm“ oder „nicht lahm“ markiert.

Die Vorbereitung der Daten erfolgte mit Python Version 3.8[3] [vRD09]. Neben den Standardbibliotheken wurden die Bibliotheken pandas Version 1.2[4] [Reb+21] und numpy Version 1.20[5] [Har+20] verwendet.

2.1.2 Zeitfenster

Zur Klassifikation wurden nicht die gesamten Zeitreihen pro Kuh und Laktation auf einmal verwendet, stattdessen wurden aus den Zeitreihen überlappende Zeitfenster derselben Länge gebildet. Dabei wurde pro Tag (pro Kuh und Laktation)

[3] `https://www.python.org/`, abgerufen am 2022-05-30
[4] `https://pandas.pydata.org`, abgerufen am 2022-07-16
[5] `https://numpy.org/`, abgerufen am 2022-07-16

ein Zeitfenster gebildet. Die Klassifikationsmodelle verwendeten jedes Zeitfenster als eine Beobachtung und wurden entsprechend trainiert und evaluiert [vgl. Hir21, S. 186].

Die Zeitfenster bestanden aus 27 aufeinanderfolgenden Tagen, wobei das Ergebnis der Gangbeurteilung am letzten Tag als Label für das gesamte Zeitfenster diente. Ergebnisse früherer Gangbeurteilungen in einem Zeitfenster wurden ignoriert. Die Klassifikationsaufgabe bestand folglich darin, aus den Merkmalswerten von 27 aufeinanderfolgenden Tagen das Vorliegen von Lahmheit am letzten (27.) Tag zu erkennen.

Für die Zeitfenster wurde eine Länge von 27 Tagen gewählt, damit mindestens ein kompletter Brunstzyklus der Kühe in jedem Zeitfenster enthalten war und sich keine Unterschiede zwischen den Zeitfenstern durch zyklusbedingte Schwankungen der Leistung und des Verhaltens ergaben. Die Zykluslänge von schwarzbunten Milchkühen wurde mit 22 ± 5 [OS51] bzw. 21 ± 3 [Gru99, S. 4] angegeben (jeweils Mittelwert $\pm$ Standardabweichung). Mit der Wahl einer Fensterlänge von 27 Tagen sollte also bei über 95 % der Kühe in jedem Zeitfenster mindestens ein Brunstzyklus enthalten sein.

In einem Zeitfenster lagen neben den Wertereihen der drei zeitabhängigen Merkmale `MilkYield`, `StepsPerHour` und `LyingDuration` auch 27 Werte des Merkmals `DaysInMilk` vor. Allerdings enthielten die 27 Werte dieses Merkmals nicht mehr Information als ein einziger Wert pro Zeitfenster, weil sich die übrigen Werte daraus direkt linear ergaben. Daher wurde nur der Laktationstag des letzten Tags pro Zeitfenster verwendet. Vom Merkmal `Lactation` wurde ebenfalls nur ein Wert pro Zeitfenster verwendet, weil es innerhalb eines Zeitfensters konstant war. Eine Beobachtung, die zum Training der Modelle verwendet wurde, bestand somit aus einem Zeitfenster der Länge 27 mit Wertereihen der drei Merkmale `MilkYield`, `StepsPerHour` und `LyingDuration` und zwei einzelnen Werten (`DaysInMilk`, `Lactation`).

Visualisierung nach Dimensionsreduktion mit t-distributed Stochastic Neighbor Embedding

Zu einer ersten visuellen Analyse wurden die Daten mit Gangbeurteilung in einem zweidimensionalen Streudiagramm dargestellt. Dazu wurden pro Beobachtung die jeweiligen Werte der konstanten Merkmale `Lactation` und `DaysInMilk` und die drei Zeitfenster der Merkmale `MilkYield`, `StepsPerHour` und `LyingDuration` zu einem Vektor zusammengefasst. Im so erstellten Datensatz mit einer Zeile pro Beobachtung und $2 + 3 \cdot 27 = 83$ Spalten, wurden die Werte in den Spalten zuerst jeweils standardisiert. Anschließend wurden die 83-dimensionalen Beobachtungen mittels t-distributed Stochastic Neighbor Embedding [vdMH08] auf eine zweidimensionale Fläche abgebildet. Die Transformation erfolgte mit der entsprechenden Funktion der Python-Bibliothek scikit-learn Version 0.24[6] [Ped+11]. Zur Visualisierung wurde die Python-Bibliothek plotnine[7] [Kib+21] verwendet. Dabei wurden

[6] `https://scikit-learn.org/0.24/`, abgerufen am 2022-05-30
[7] `https://plotnine.readthedocs.io/en/stable/`, abgerufen am 2022-05-30

die Beobachtungen entsprechend ihrer Labelklasse („lahm“ oder „nicht lahm“) markiert.

2.1.3 Aufteilung in Trainings- und Testdaten

Bei der Beurteilung der Eignung von Klassifizierungsansätzen und deren Vergleich ist man nicht daran interessiert, wie gut sich ein Verfahren an einen Datensatz anpassen lässt, sondern wie gut das trainierte Modell unbekannte Beobachtungen klassifizieren kann. Das Ziel jeder Evaluation eines Klassifikationsverfahrens ist es also, den erwarteten, auf unbekannte Beobachtungen aus einer Zielpopulation verallgemeinerten Klassifikationsfehler zu bestimmen [vgl. DHS01, S. 483]. Es hat sich bewährt, etwa 10 % der Daten als Testdaten zu verwenden und vom Training auszuschließen [DHS01, S. 484]. Zur Bestimmung des Klassifikationsfehlers von unbekannten Beobachtungen werden dann die Klassifikationsergebnisse des trainierten Modells für die Testdaten mit den entsprechenden Labels verglichen. Der gesamte Datensatz (incl. der Daten ohne Gangbeurteilung) wurde für die vorliegende Arbeit auf der Ebene der Kühe zufällig in zehn etwa gleich große Teile geteilt und in separaten Dateien gespeichert. Die Zugehörigkeit zu einem der drei Betriebe wurde dabei ignoriert. So stand für eine zehnfache Kreuzvalidierung der verschiedenen Klassifikationsansätze immer dieselbe Aufteilung in Trainings- und Testdaten zur Verfügung.

2.2 Untersuchte Ansätze zur Klassifikation von multivariaten Zeitreihen

Für die Merkmale `MilkYield`, `StepsPerHour` und `LyingDuration` lagen jeweils ein Meßwert pro Tag im Beobachtungszeitraum vor. Diese Daten befanden sich in diskreten Zeitreihen.

Bei Zeitreihendaten handelt es sich um zeitlich geordnete Daten. Es wird angenommen, dass die Reihenfolge der Werte selbst Information enthält. Diese Information soll es in gewissem Unfang ermöglichen, aus der Abfolge beobachteter Werte auf zukünftige Werte zu schließen [Hir21, S. 2–3]. Liegen zu jedem Beobachtungszeitpunkt Werte von mehreren Merkmalen vor, welche zusammen betrachtet werden sollen (wie im vorliegenden Fall), so spricht man von multivariaten Zeitreihen [Rui+21]. Bei der Analyse multivariater Zeitreihendaten sollten sowohl die zeitlichen Beziehungen innerhalb der einzelnen Merkmalsdatenreihen als auch die Beziehungen zwischen den Werten der verschiedenen Merkmale zum gleichen Zeitpunkt berücksichtigt werden.

Als Zeitreihenklassifikation bezeichnet man die Einteilung einzelner oder mehrerer Zeitreihen zusammen in Klassen. Sie wurde als eine der anspruchsvollsten Herausforderungen bei der Gewinnung von Wissen aus Daten bezeichnet [Faw+19]. Zur Klassifikation von multivariaten Zeitreihen können sowohl allgemeine Methoden als auch spezielle Zeitreihenklassifikationsmethoden eingesetzt werden [WYO17; Faw+19; Rui+21].

Die multivariaten Zeitreihen bestehend aus `MilkYield`, `StepsPerHour` und `LyingDuration` aus den Klauen*fit*net-Daten sollten unter Verwendung der zusätzlichen Information aus den konstanten Merkmalen `DaysInMilk` und `Lactation` in die Labelklassen „lahm“ und „nicht lahm“ aus der Gangbeurteilung eingeteilt werden. In meiner vorliegenden Arbeit habe ich diskriminative Modelle sowohl in Ansätzen mit Feature Engineering als auch in Ende-zu-Ende-Ansätzen untersucht. Darüber hinaus wurden generative Modelle zum Vortraining von Klassifikationsmodellen mit selbsterlernten Merkmalen einbezogen. Die untersuchten Klassifikationsansätze gliedern sich wie folgt:

- Diskriminative Klassifikationsansätze
  - Ansätze mit Feature Engineering
    - Support Vector Machine (SVM)
    - Random Forest
    - Multilayer Perceptron (MLP)
  - Ende-zu-Ende-Ansätze
    - MLP
    - Convolutional Neural Network (CNN)
    - Gated Recurrent Unit (GRU)
- Generative Klassifikationsansätze
  - MLP
  - CNN
  - GRU

Mit SVM und Random Forests wurden dabei zwei Methoden des klassischen maschinellen Lernens verwendet, während die Modelle der übrigen Ansätze alle auf künstlichen neuronalen Netzen basierten.

2.2.1 Diskriminative Klassifikationsansätze

Ein Klassifizierer, der direkt mit Zeitreihen- bzw. Merkmalsdaten und entsprechenden Labels trainiert wird und eine Labelklasse zurückgibt, wird als *diskriminatives Modell* bezeichnet [Faw+19]. In diskriminativen Ansätzen wird das Klassifikationsproblem direkt abgebildet, bzw. es wird direkt die Zuordnung von Eingabemerkmalen zu Klassenlabels erlernt. Dafür wird während des Trainings die bedingte Wahrscheinlichkeit $p(y|\boldsymbol{x})$ für das Vorliegen der Labelklasse y bei gegebenem Merkmalsvektor $\boldsymbol{x}$ optimiert. Durch Anpassung der Entscheidungsgrenze für die Klassenzuordnung basierend auf $p(y|\boldsymbol{x})$ wird der Klassifikationsfehler minimiert [NJ01].

Feature engineering

Um die Klassifikationsleistung von Modellen des klassischen maschinellen Lernens und eines Multilayer Perceptron zu verbessern, wurde zusätzliches Fachwissen einbezogen. Nach von Rüden, Mayer, Beckh u. a. [vRMB+20] ist die Datenvorverarbeitung ein gängiger Ansatzpunkt, um Wissen in das maschinelle Lernen einzube-

ziehen, welches nicht in Form von Daten aus Beobachtungen vorliegt. Expertenwissen und aus den Ergebnissen anderer Studien abgeleitete Erkenntnisse können beim sogenannten *Feature Engineering* herangezogen werden, um aus vorliegenden Daten zusätzliche Merkmale zu erzeugen.

Lahme Kühe haben ein verändertes Niveau der Geh- und Liegeaktivität als gesunde Kühe [Bya+19, Abb. 5]. Insbesondere liegen sie länger pro Liegevorgang [Ito+10; YGB12]. Allerdings zeigen sie auch eine größere Variabilität der Länge pro Liegevorgang [Ito+10]. Überhaupt unterscheiden sich die Tagesschwankungen der Aktivität von lahmen und gesunden Kühen [Bya+19, Abb. 5]. Um die größere Variablität und das veränderte Muster der Aktivitätsschwankungen von lahmen Kühen abzubilden, wurden die Standardabweichungen und Spektren der zeitabhängigen Variablen (`MilkYield`, `StepsPerHour` und `LyingDuration`) pro Zeitfenster als zusätzliche Merkmale erzeugt. Durch die Transformation in den Spektralraum können zyklische Schwankungen der Merkmalswerte besser analysiert werden. Unterschiede zwischen den Tagesschwankungen der Leistung und Aktivität von lahmen und nicht lahmen Kühen können dadurch besser als Merkmale für die Klassifikation verwendet werden. Die Standardabweichungen wurden als Merkmale der Variabiliät der Werte des jeweiligen ursprünglichen Merkmals hinzugefügt. Jede Beobachtung zum Training der Modelle bestand folglich aus einem Vektor mit 167 Elementen, die sich aus den Merkmalen ergaben: `DaysInMilk`, `Lactation`, den Zeitfenstern und Spektren der drei zeitabhängigen Variablen (jeweils zweimal 27 Merkmale) sowie den Standardabweichungen der Werte in den drei Zeitfenstern. Folgende Modelle wurden in den Klassifikationsansätzen mit Feature Engineering verwendet: ein Random Forest (FE-RF), eine SVM (FE-SVM) und ein MLP (FE-MLP).

FE-RF

Random Forests zur Klassifikation bestehen aus einem Ensemble von Entscheidungsbäumen, deren Klassifikation jeweils mit einer Stimme für das gemeinsame Klassifikationsergebnis zählt. Jeder Entscheidungsbaum wird mit einer anderen Zufallsauswahl der verfügbaren Beobachtungen trainiert, wobei die Auswahlen unabhängig und identisch verteilt sein müssen [Bre01].

Als zusätzliches Zufallselement werden für die Entwicklung der Knoten der Bäume nicht jedes Mal das Beste aus allen Merkmalen gesucht, sondern nur aus einer kleineren zufälligen Auswahl aus den Merkmalen [Bre01]. Die so entstehenden Entscheidungsbäume des Ensembles unterscheiden sich stärker voneinander, was die Varianz der Vorhersage reduziert [Gér20, S. 200].

In der vorliegenden Arbeit wurde ein Random Forest zur Klassifikation der Daten mit den zusätzlich erzeugten Merkmalen trainiert. Die Zufallsauswahlen für das Training der Entscheidungsbäume wurden mittels Bagging erzeugt, d. h. es wurden Beobachtungen aus dem Trainingsdatensatz mit Zurücklegen gezogen. Für jeden Baum wurden so viele Beobachtungen aus dem Trainingsdatensatz gezogen, wie insgesamt darin enthalten waren. Zur Entscheidung über die Teilung eines Knotens wurde nicht das beste aus allen 167 Merkmalen verwendet, sondern nur das beste Merkmal aus einer jeweils zufällig gewählten Untermenge der Größe

$\sqrt{167}$. Die einzelnen Entscheidungsbäume im Ensemble wurden mit dem CART-Algorithmus [DHS01, S. 396–411] entwickelt. Als Metrik zur Auswahl der Knoten wurde die Gini-Unreinheit [DHS01, S. 399] eingesetzt.

Die Anzahl der Entscheidungsbäume und das Minimum von Beobachtungen in einem Blatt (welches die maximale Tiefe der Bäume bestimmte) wurden mittels Gittersuche optimiert. Die Anzahl der Bäume wurde zwischen dem Ressourcenbedarf und der Klassifikationsleistung abgewogen. Das Minimum von Beobachtungen pro Blatt diente zur Regularisierung des Modells [Gér20, S. 183–185].

FE-SVM

Als weiterer Modelltyp des klassischen maschinellen Lernens wurde eine *Support Vector Machine (SVM)* zur Klassifikation der Beobachtungen nach Feature Engineering evaluiert. Eine SVM sucht zur Klassifikation der Beobachtungen die optimale Hyperebene im Merkmalsraum, welche die Beobachtungen entsprechend der Klassen aufteilt. Die Hyperebene wird lediglich durch die Datenpunkte jeder Klasse bestimmt, die ihr jeweils am nächsten liegen. Diese Beobachtungen bezeichnet man als Stützvektoren (support vectors) [DHS01, S. 259–265].

Die verwendeten Daten waren im Merkmalsraum selbst nicht gut linear separierbar, deshalb wurden sie in ähnlichkeitsbasierte Merkmale transformiert, und die Separierung mittels Hyperebene und Stützvektoren erfolgte in diesem neuen Merkmalsraum. Die Transformation in ähnlichkeitsbasierte Merkmale erfolgte über einen Kernel mit gaußscher radialer Basisfunktion [HTF09, S. 172–174 und S. 424–426].

Da SVMs sehr empfindlich auf unterschiedliche Skalen bei den Merkmalen reagieren [Gér20, S. 156], wurden die Merkmale vor dem Training jeweils standardisiert durch Abzug des Mittelwerts und Division durch die Standardabweichung. Der Regularisierungshyperparameter C der SVM wurde per Gittersuche optimiert.

FE-MLP

Multilayer Perceptrons (MLPs) behandeln das Problem der linearen Separierbarkeit der Beobachtungen auf andere Weise. Im Gegensatz zur SVM mit einem Kernel mit gaußscher radialer Basisfunktion werden nicht die Merkmalsdaten auf eine vorher festgelegte Art transformiert, sondern das MLP lernt, die Beobachtungen optimal nichtlinear in einen latenten Merkmalsraum abzubilden, in dem sie dann wiederum linear separiert werden [DHS01, S. 282–284].

Ein MLP besteht aus einer Eingabeschicht, beliebig vielen verborgenen Schichten und einer Ausgabeschicht. Die Schichten bestehen jeweils aus einem oder mehreren künstlichen Neuronen, die jedes mit allen Neuronen der folgenden Schicht verbunden sind. In jedem Neuron werden die Signale, die von der vorherigen Schicht empfangen werden, gewichtet und zusammen durch die Aktivierungsfunktion in eine Ausgabe umgewandelt. Die Ausgabe wird dann an alle nachfolgenden Neuronen weitergegeben. Die Gewichte aller Eingänge an allen Neuronen werden während des Trainings mit Hilfe einer Variante des Gradientenabstiegsverfahrens erlernt [Gér20, S. 290–291].

Nachdem in einem Trainingsschritt die Signale von der Eingabeschicht bis zur Ausgabeschicht durch das Netz gelaufen sind, wird die Klassifikation entsprechend der Ausgabe vorgenommen und mit den Labels der Trainingsdaten verglichen. Daraus wird der Trainingsfehler in diesem Schritt berechnet. Der Backpropagation-Algorithmus dient dazu, den Trainingsfehler von der Ausgabe bis zur Eingabe durchzureichen [DHS01, S. 288–293]. In jeder Schicht werden die Gradienten berechnet und die Gewichte mit dem Optimierer entsprechend der jeweiligen Lernrate angepasst.

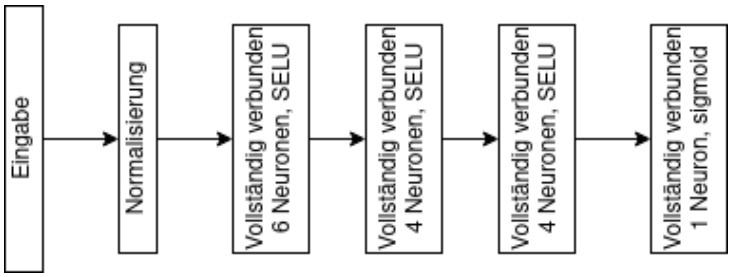

Abbildung 2.2: Multilayer Perceptron (MLP) im Klassifikationsansatz FE-MLP

Als MLP habe ich entsprechend der Empfehlung von Géron [Gér20, S. 375] ein selbstnormalisierendes vollständig verknüpftes neuronales Netz erstellt und trainiert (Abbildung 2.2). Zur Selbstnormalisierung mussten die Eingangsmerkmale standardisiert vorliegen [Gér20, S. 340], d. h. einen Mittelwert von 0 und eine Standardabweichung von 1 aufweisen. Deshalb enthielt das Netz als erste verborgene Schicht eine Normalisierungsschicht. Anschließend folgten drei Schichten mit sechs, vier und vier vollständig verknüpften Neuronen, welche die selbstnormalisierende Scaled Exponential Linear Unit (SELU) Aktivierungsfunktion [Kla+17] verwendeten (Abbildung 2.2). Die Gewichte der Neuronen in den verborgenen Schichten wurden mit Werten aus einer Normalverteilung nach LeCun initialisiert [Gér20, S. 335–336] wie es für ein selbstnormalisierendes Netz erforderlich ist [Kla+17]. Die Ausgabeschicht zur Klassifizierung bestand aus einem vollständig verknüpften Neuron mit Sigmoid-Aktivierungsfunktion [Gér20, S. 296], das mit Werten aus einer Gleichverteilung nach Glorot initialisiert wurde [GB10]. Als Verlustfunktion und Optimierer wurden die binäre Kreuzentropie [Gér20, S. 151–153 und S. 296] und Nadam [Doz16] verwendet. Die Hyperparameter Lernrate und Batchgröße wurden mittels Gittersuche optimiert.

Ende-zu-Ende-Ansätze

Im Gegensatz zum Feature Engineering erlernen *Ende-zu-Ende-Modelle* selbstständig die Merkmale für die Klassifizierung aus den Rohdaten während der diskriminative Klassifikator trainiert wird. Dieser Ansatz erfordert kein zusätzliches Fachwissen zur Entwicklung von Merkmalen [Nwe+18].

Folgende Ende-zu-Ende-Modelle habe ich für die vorliegende Arbeit untersucht: ein MLP (E2E-MLP), ein CNN (E2E-CNN) und ein neuronales Netz mit GRUs

(E2E-GRU). Außer den in den Klauen*fit*net-Daten vorhandenen Variablen wurden bei diesen Ansätzen keine zusätzlichen Merkmale manuell erzeugt. Während dem MLP pro Beobachtung ein Merkmalsvektor der Länge 83 übergeben wurde, erforderten das CNN und das Netz mit GRUs getrennte Eingangsschichten für die konstanten Merkmale (`Lactation` und `DaysInMilk`) und die Zeitfenster aus den drei Zeitreihen (`MilkYield`, `StepsPerHour` und `LyingDuration`). Diese Modelle konnten daher nicht als rein sequentielle Netze entwickelt werden [vgl. Hir21, S. 237–241].

Wie auch bei FE-MLP wurden bei allen Ende-zu-Ende-Modellen die binäre Kreuzentropie [Gér20, S. 151–153 und S. 296] als Verlustfunktion und der Nadam-Optimierer [Doz16] verwendet. Die Lernraten und die Batchgrößen wurden ebenfalls mit Gittersuchen optimiert.

E2E-MLP

Es wurde ein Multilayer Perceptron mit derselben Architektur im Ende-zu-Ende-Ansatz verwendet, wie sie für den entsprechenden Ansatz mit Feature Engineering (FE-MLP) auf der vorherigen Seite beschrieben wurde.

E2E-CNN

Convolutional Neural Networks (CNNs) sind künstliche neuronale Netze, die mindestens eine Faltungsschicht enthalten. Die Neuronen einer Faltungsschicht erhalten ihre Eingangssignale nicht von allen Merkmalen der Eingabe bzw. von allen Neuronen der vorherigen Schicht, sondern nur aus ihrem Wahrnehmungsfeld, dessen Größe durch die Kernelgröße festgelegt wird. Dadurch werden nur räumlich (bei Bildern) oder zeitlich (bei Zeitreihen) näher zusammenliegende Werte von einem Neuron berücksichtigt. Bei CNN werden die Gewichte der Eingangssignale für ein Wahrnehmungsfeld als *Filter* bezeichnet. Derselbe Filter wird von allen Neuronen verwendet, die notwendig sind, um eine vollständige Abbildung der Eingangssignale (*Feature Map*) zu erzeugen. In einer Faltungsschicht können mehrere unterschiedliche Filter angewandt werden. Die Zahl der Neuronen vervielfacht sich entsprechend um die Zahl der Filter. Werden mehr als ein Filter in einer Faltungsschicht eingesetzt, erhöht sich die Dimensionalität der Ausgabe um eins. Die zusätzliche Dimension der Ausgabe wird durch den Stapel der Feature Maps gebildet. Jeder zusätzliche Filter fügt der Ausgabe einer Faltungsschicht eine weitere Feature Map hinzu [Gér20, S. 451–456].

Das untersuchte CNN im Ende-zu-Ende-Ansatz war ein neuronales Netz mit zwei Eingangsschichten und einer Ausgabeschicht (Abbildung 2.3 auf der nächsten Seite). Der Pfad für die konstanten Merkmale bestand aus einer Normalisierungsschicht und einer vollständig verknüpften Schicht mit zwei Neuronen. Für die zeitabhängigen Merkmale folgte auf eine Normalisierungsschicht ein Fully Convolutional Neural Network wie von Wang, Yan und Oates [WYO17] veröffentlicht. Das FullyCNN bestand aus drei Blöcken mit jeweils einer eindimensionalen Faltungsschicht [Gér20, S. 524], Batch Normalisierung und Aktivierungsfunktion. Die Faltungsschichten hatten 128, 256 und 128 Filter mit den Kernelgrößen

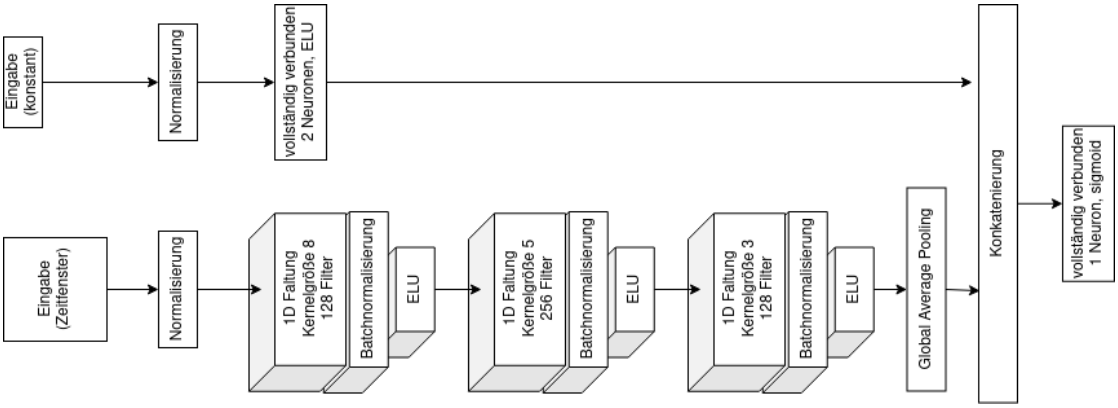

Abbildung 2.3: Zeitreihenanalyse mit Convolutional Neural Network (CNN) im Klassifikationsansatz E2E-CNN

von acht, fünf und drei und Zero Padding [Gér20, S. 452]. Den Abschluss des FullyCNN bildete eine Global Average Pooling Schicht. Die Ausgaben der beiden Eingangspfade wurden in einer Konkatenationsschicht verbunden. Die Klassifikation erfolgte schließlich durch eine Ausgabeschicht mit einem Neuron und Sigmoid-Aktivierungsfunktion wie bei FE-MLP auf Seite 23 beschrieben. Den Empfehlungen von Géron [Gér20, 340 und 336] folgend, war die Aktivierungsfunktion der verborgenen Schichten immer eine Exponential Linear Unit (ELU) [CUH16], und die Gewichte wurden durch Ziehung aus einer Normalverteilung nach He u. a. [He+15] initialisiert.

E2E-GRU

Neuronale Netze mit Gated Recurrent Units (GRUs) [Cho+14] gehören zu den *Recurrent Neural Networks (RNNs)*. RNNs wurden speziell für die Analyse von Daten in Zeitreihen entwickelt. Im Gegensatz zu den MLPs werden bei RNNs die Signale nicht nur von einer Schicht an die nächste weitergereicht, sondern auch an Neuronen in derselben Schicht (oder in vorherigen Schichten). Die Weiterleitung innerhalb derselben Schicht erfolgt so, dass ein Neuron ein Signal vom Wert eines Merkmals zu einem bestimmten Zeitpunkt aus der vorherigen Schicht erhält und ein Signal von den zeitlich vorangegangenen Werten desselben Merkmals von einem Neuron in derselben Schicht. Auf diese Weise werden die zeitlichen Abhängigkeiten der Merkmalswerte in der Netzstruktur berücksichtigt. Es wurden spezielle Formen von künstlichen Neuronen entwickelt, um die Signale von den zeitlich früheren Merkmalswerten und die Signale vom aktuellen Merkmalswert sinnvoll zu kombinieren. GRUs sind eine Form, die besonders für die Verarbeitung von kurzen und mittleren Zeitfenstern geeignet ist. Für jeden Wert desselben Merkmals innerhalb des Zeitfensters wird dasselbe Gewicht verwendet, so dass die Zahl der zu erlernenden Gewichte begrenzt bleibt [Hir21, S. 185–191].

Eine GRU-Schicht gibt die Ausgaben der letzten Zeitschritte pro Merkmal zurück oder die Ausgaben aller Zeitschritte pro Merkmal. Werden für alle Zeitschritte Werte ausgegeben, kann eine folgende Schicht die bereits verarbeiteten Daten wei-

ter analysieren. Ähnlich wie eine Schicht eines CNN mehrere Filter enthalten kann, können in einer Schicht aus GRUs mehrere Sätze von Gewichten eingesetzt werden [Hir21, S. 200–201]. Dadurch wird der Ausgabe wie bei CNN eine zusätzliche Dimension aus dem Stapel von Feature Maps hinzugefügt.

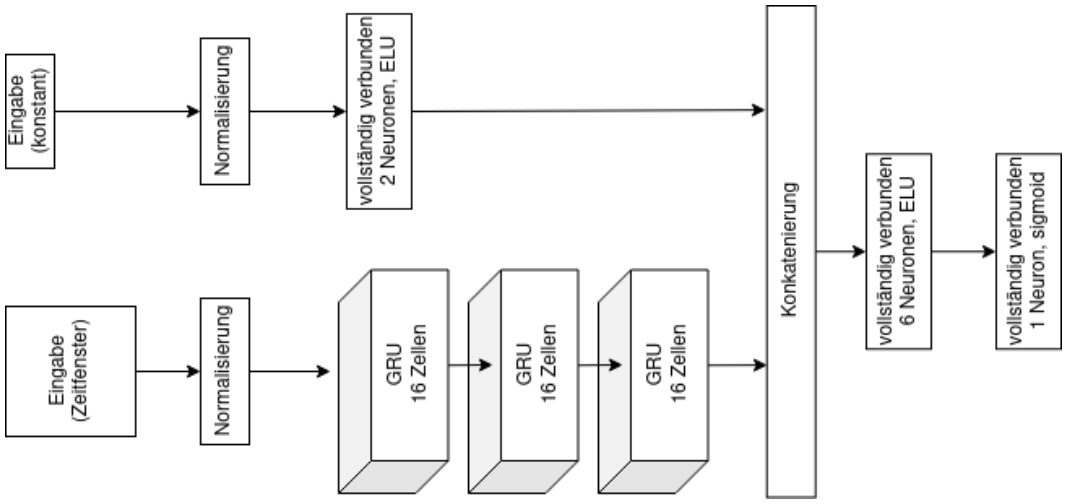

Abbildung 2.4: Zeitreihenanalyse mit Gated Recurrent Units (GRUs) im Klassifikationsansatz E2E-GRU

Das Ende-zu-Ende-Modell mit GRUs hatte wie E2E-CNN zwei Eingabeschichten und eine Ausgabeschicht (Abbildung 2.4). Eine Normalisierungsschicht sowie eine vollständig verknüpfte Schicht mit zwei Neuronen bildeten auch hier den Pfad für die beiden konstanten Merkmale. Ebenfalls mit einer Normalisierungsschicht begann der Pfad für die drei zeitabhängigen Merkmale. Darauf folgten drei GRU-Schichten mit jeweils 16 Zellen, wovon die ersten beiden vollständige Sequenzen zurückgaben. Die Architektur des Pfads für die Zeitfenster folgte dem Beispiel auf Seite 201 im Lehrbuch von Hirschle [Hir21]. Die getrennten Pfade mündeten in einer Konkatenationsschicht, auf die eine gemeinsame, vollständig verknüpfte Schicht mit sechs Neuronen folgte. Eine Klassifizierungsschicht entsprechend der Ausgabeschicht von FE-MLP auf Seite 23 schloss das künstliche neuronale Netz ab. Die beiden verborgenen vollständig verknüpften Schichten hatten eine ELU-Aktivierungsfunktion [CUH16] und wurden nach He u. a. [He+15] initialisiert.

2.2.2 Generative Ansätze

In *generativen Klassifikationsansätzen* wird die gemeinsame Wahrscheinlichkeit $p(y, \boldsymbol{x})$ der Labelklasse y und des Merkmalsvektors $\boldsymbol{x}$ modelliert. Für die eigentliche Klassifikation wird dann unter Verwendung des Satzes von Bayes die bedingte Wahrscheinlichkeit $p(y|\boldsymbol{x})$ berechnet und die Klasse y mit der größten bedingten Wahrscheinlichkeit vorhergesagt [NJ01].

Für die vorliegende Arbeit bestanden die generativen Ansätze allerdings darin, zunächst Modelle unüberwacht zu trainieren, um Abbildungen der Merkmale

zu erlernen. Anschließend wurden die eigentlichen Klassifikatoren auf Grundlage dieser Abbildungen trainiert. Für das unüberwachte Training konnten alle verfügbaren Beobachtungen einschließlich derer ohne Labels verwendet werden, sodass wesentlich mehr Daten zur Verfügung standen. Das abschließende Training des Klassifikators erfolgte dann nur mit den Beobachtungen, für welche Labels vorhanden waren [vgl. LKL14].

Im Zusammenhang mit Deep-Learning-Modellen bezeichnet man diese Verwendung generativer Modelle als *unüberwachtes Vortrainieren*. Es handelt sich um eine Variante des Transfer Learnings, bei der ein großer Teil der Schichten des endgültigen Klassifikationsmodells als Bestandteil des generativen Modells vortrainiert wird. Lediglich die Gewichte der oberen Schichten, die für die Klassifikation notwendig sind, können nicht vortrainiert werden. Auf diese Weise müssen beim Training des Klassifikators nicht mehr alle Gewichte von Anfang an gelernt werden, sondern sie sind bereits mit Werten vorbelegt, die zum vorliegenden Problem passen [Gér20, S. 348–352].

Zur Erkennung von Lahmheiten bei Milchkühen aus Aktivitäts- und Leistungsdaten erfolgte das unüberwachte Vortraining für die vorliegende Arbeit mit gestapelten Autoencodern, wobei der vollständige Datensatz einschließlich der ungelabelten Beobachtungen verwendet wurde. *Gestapelte Autoencoder* sind neuronale Netze, die im Zentrum eine latente Abstraktion der Eingaben erlernen (Abbildung 2.5 auf der nächsten Seite). Sie werden so trainiert, dass ihre Ausgabe möglichst genau der Eingabe entspricht [Gér20, S. 575–577].

Für das überwachte Klassifikationstraining wurden die Decoder durch die entsprechenden Klassifikatoren ersetzt und nur die gelabelten Daten verwendet (Abbildung 2.5 auf der nächsten Seite). Die Klassifikatoren waren immer MLPs mit zwei verborgenen Schichten sowie einer Ausgabeschicht mit einem Neuron und Sigmoid-Aktivierungsfunktion wie bei FE-MLP auf Seite 23.

Es wurden generative Ansätze mit ähnlichen Modellen wie in den Ende-zu-Ende-Ansätzen untersucht: mit einem MLP (AE-MLP), einem CNN (AE-CNN) und einem GRU-Netz (AE-GRU). Wie bei den Ende-zu-Ende-Modellen konnten auch hier die Modelle der Ansätze mit CNN und GRU nicht als rein sequentielle Netze erstellt werden, weil die beiden konstanten Merkmale und die drei Zeitfenster unterschiedlich dimensioniert übergeben werden mussten (als Vektor bzw. als Matrix pro Beobachtung). Daher hatten die Encoder zwei Eingabeschichten und eine Ausgabeschicht (Abbildung 2.5 auf der nächsten Seite). Die Decoder von AE-CNN und AE-GRU hatten umgekehrt eine Eingangs- und zwei Ausgabeschichten. In den Ausgabeschichten aller drei Decoder wurden keine Aktivierungsfunktionen verwendet, weil die Decoder die ursprünglichen Merkmalswerte in Regressionsaufgaben aus der latenten Repräsentation wieder herstellen sollten. Sie wurden mit Werten aus einer Gleichverteilung nach Glorot und Bengio [GB10] initialisiert.

Beim unüberwachten Vortraining der Autoencoder wurde die von Hastie, Tibshirani und Friedman [HTF09, S. 349–350] adaptierte Verlustfunktion nach Huber [Hub64] eingesetzt, weil es sich um Regressionsaufgaben handelte. Für das anschließende Klassifikationstraining wurde wiederum die binäre Kreuzentropie als Verlustfunktion eingesetzt. Für alle Modelle wurde der Nadam-Optimierer [Doz16]

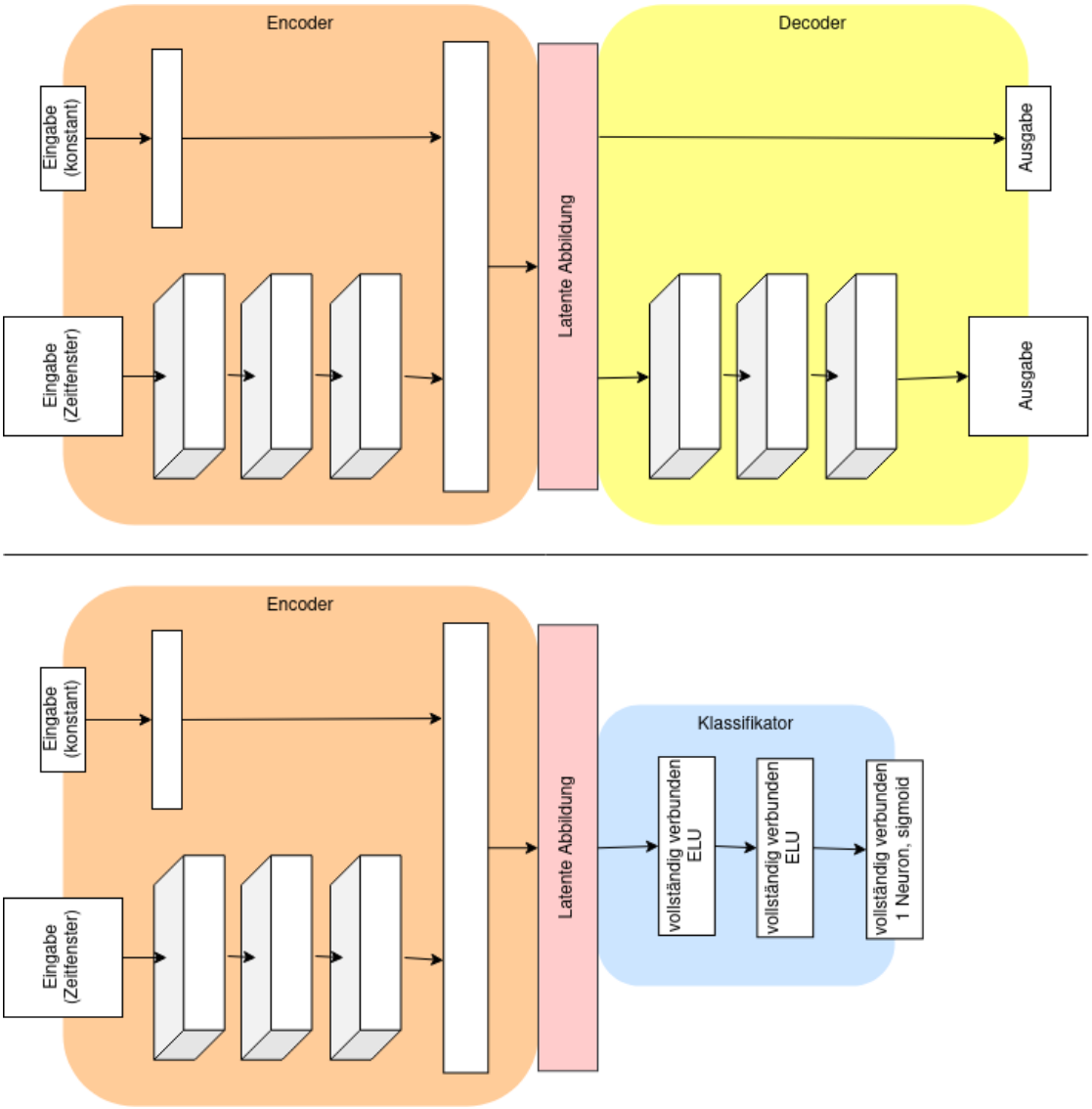

Abbildung 2.5: Verwendung von Autoencodern in den generativen Ansätzen (oben: Autoencoder mit 2 Ein- und 2 Ausgängen für das unüberwachte Vortraining, unten: Ersatz des Decoders durch den Klassifikator für die Klassifikation)

verwendet. Die Lernraten und die Batchgrößen wurden mit Gittersuchen optimiert. Die Normalisierung der Merkmalswerte erfolgte diesmal außerhalb der Modelle.

AE-MLP

Auch beim generativen Ansatz wurden selbstnormalisierende vollständig verknüpfte neuronale Netze mit SELU-Aktivierungsfunktion und Initialisierung nach LeCun erstellt [vgl. Kla+17]. Durch das unüberwachte Vortraining standen wesentlich mehr Beobachtungen zum Anlernen der Modellparameter zur Verfügung als bei den diskriminativen Ansätzen. Daher konnte das MLP im generativen Ansatz mehr und deutlich breitere Schichten enthalten als die MLPs mit Feature Engi-

neering und im Ende-zu-Ende-Ansatz. Der Encoder bestand aus vier verborgenen Schichten mit 40, 20, 20 und 20 Neuronen und einer Ausgabeschicht mit fünf Neuronen für die latente Merkmalsrepräsentation. Im Decoder waren die gleichen verborgenen Schichten in umgekehrter Reihenfolge enthalten, gefolgt von einer Ausgabeschicht mit 83 vollständig verknüpften Neuronen. Die beiden verborgenen Schichten des Klassifikators enthielten jeweils drei Neuronen.

AE-CNN

Bei AE-CNN handelte es sich um einen „Convolutional Auto-Encoder“ [Mas+11]. Der Encoder von AE-CNN entsprach dem E2E-CNN-Modell bis zur Konkatenationsschicht (siehe Abbildung 2.3 auf Seite 25). Der Decoder wurde als Umkehrung dieses Modells erstellt. Für die konstanten Merkmale bestand der Ausgabepfad lediglich aus einer vollständig verknüpften Ausgabeschicht mit zwei Neuronen. Der Decoderpfad für die Zeitfenster bestand aus drei Blöcken mit jeweils einer transponierten eindimensionalen Faltungsschicht (von Zeiler u. a. [Zei+10] als „Deconvolutional Network layer“ beschrieben), Batch Normalisierung [IS15] und Aktivierungsfunktion. Die transponierten Faltungsschichten enthielten 128, 256 und 128 Filter mit den Kernelgrößen drei, fünf und acht ohne Zero Padding. Die Ausgabeschicht war ebenfalls eine transponierte Faltungsschicht mit 3 Filtern der Kernelgröße 14 und ohne Padding. Dadurch konnten die ursprünglichen Dimensionen der Zeitfenster wiederhergestellt werden. Im Klassifikator, der im überwachten Training den Decoder ersetzte, wurden jeweils 15 Neuronen in den verborgenen Schichten eingesetzt. Alle verborgenen Schichten von Encoder, Decoder und Klassifikator enthielten die ELU-Aktivierungsfunktion [CUH16] und wurden durch Ziehung aus einer Normalverteilung nach He u. a. [He+15] initialisiert.

AE-GRU

Auch im rekurrenten Autoencoder des generativen Ansatzes AE-GRU entsprach der Encoder dem rekurrenten Ende-zu-Ende-Modell E2E-GRU ohne die Ausgabeschicht (siehe Abbildung 2.4 auf Seite 26). Wie bei AE-CNN bestand der Decoder von AE-GRU aus einer Umkehrung des Encoders. Für die konstanten Merkmale bestand der Decoderpfad ebenfalls lediglich aus einer Ausgabeschicht mit zwei vollständig verknüpften Neuronen. Im Decoderpfad für die Zeitfenster wurde der Eingangsvektor (die latente Repräsentation der Merkmale) zuerst entsprechend der Länge der Zeitfenster 27-mal wiederholt. Darauf folgten drei Schichten mit jeweils 16 GRUs, die alle die gesamten Sequenzen zurückgaben. Eine Schicht mit drei vollständig verknüpften Neuronen bildete die Ausgabeschicht, die zeitverteilt auf jeden Schritt eines Zeitfensters separat angewandt wurde [vgl. Gér20, S. 584]. Die beiden verborgenen Schichten des Klassifikator-MLP bestanden jeweils aus vier Neuronen mit ELU-Aktivierungsfunktion [CUH16]. Initialisiert wurden sie mit normalverteilten Werten nach He u. a. [He+15].

2.3 Vergleich der Klassifikationsansätze

Das Hauptziel der vorliegenden Arbeit war die Beantwortung der Frage, welcher der untersuchten Klassifizierungsansätze am besten zum Einsatz in der automatischen Lahmheitserkennung von Milchkühen an Hand von Leistungs- und Aktivitätsdaten geeignet ist. Dazu sollten die Klassifikationsleistungen der Ansätze und ihr Ressourcenbedarf verglichen werden. Die Klassifikationsleistungen der Modelle sollten sich aber nicht nur auf den vorliegenden Klauen*fit*net-Datensatz beziehen, sondern auf ähnliche Datensätze verallgemeinert werden. Aus diesem Grund sollten die erwarteten (verallgemeinerten) Klassifikationsleistungen der Ansätze verglichen werden [vgl. DHS01, S. 482]. Die Klassifikationsleistungen wurden mit verschiedenen Metriken bestimmt.

2.3.1 Vergleichskriterien

Es wurden grenzwertabhängige Metriken zur Bewertung der Klassifikationsleistungen verwendet, wie von Humm u. a. [Hum+20] für konkrete Aufgabenstellungen mit festgelegten Merkmalen empfohlen und in Abschnitt 1.2.1 auf Seite 9 bereits diskutiert. Um eine Abwägung zwischen den erwarteten Wahrscheinlichkeiten von falsch-positiven und von falsch-negativen Klassifizierungen bei der Auswahl des Klassifikationsansatzes zu ermöglichen [vgl. Gér20, S. 96], wurden die unterschiedlichen Metriken Genauigkeit, Relevanz, Sensitivität und Spezifität zu Bewertung herangezogen. Zur Beurteilung des Ressourcenbedarfs der untersuchten Klassifikationsansätze wurden jeweils

- der maximal benötigte Arbeitsspeicher für Training und Validierung,
- die Dauer des gesamten Trainings und
- der Zeitbedarf für die Klassifikation während der Validierung

bestimmt.

Genauigkeit

Die Genauigkeit (englisch: accuracy) ist die erwartete Wahrscheinlichkeit korrekter Klassifizierungen und wird geschätzt über den Anteil korrekter Klassifizierungen der Testdaten [vgl. Gér20, S. 93; DHS01, S. 483]. Als sehr weit verbreitete Metrik wurde sie beispielsweise in den Arbeiten von Fawaz u. a. [Faw+19], Ruiz u. a. [Rui+21], Lasser u. a. [Las+21] und Shahinfar u. a. [Sha+21] verwendet. Allerdings ist die Genauigkeit anfällig für unausgeglichene Häufigkeitsverteilungen der Zielklassen in den Daten [Hum+20]: bei selten auftretenden Labels würde ein Klassifikator, der alle Beobachtungen als negativ einstuft, eine hohe Genauigkeit erzielen. Aus diesem Grund empfahlen O'Leary u. a. [OLe+20] für den Vergleich von Klassifikationsansätzen für die automatische Lahmheitserkennung bei Milchkühen die Relevanz zusammen mit der Sensitivität und der Spezifität.

Relevanz

Die Relevanz wird auch als Precision oder positiv-prädiktiver Wert bezeichnet, und ist definiert als die bedingte Wahrscheinlichkeit für das tatsächliche Vorliegen eines Merkmals, gegeben dass die Klassifikation entsprechend lautet. Ein einfacher Schätzer für die Relevanz ist der Anteil richtig Positiver an allen positiven Klassifizierungen:

$$\widehat{\text{Relevanz}} = \frac{\text{RP}}{\text{RP} + \text{FP}}$$

(RP: Anzahl richtig positiver Klassifizierungen; FP: Anzahl falsch positiver Klassifizierungen).
Sie ist abhängig von der Häufigkeit des Zielmerkmals in der Zielpopulation [vgl. Wei08, S. 643–644] und sollte zusammen mit der Sensitivität interpretiert werden [Gér20, S. 95].

Sensitivität

Die auch Recall genannte Sensitivität ist die bedingte Wahrscheinlichkeit einer positiven Klassifizierung, wenn das Zielmerkmal tatsächlich vorliegt. Sie kann geschätzt werden als Anteil der tatsächlich Positiven, die korrekt klassifiziert wurden [Gér20, S. 95]:

$$\widehat{\text{Sensitivität}} = \frac{\text{RP}}{\text{RP} + \text{FN}}$$

(RP: Anzahl richtig positiver Klassifizierungen; FN: Anzahl falsch negativer Klassifizierungen).

Spezifität

Die bedingte Wahrscheinlichkeit einer negativen Klassifizierung, wenn das Zielmerkmal tatsächlich nicht vorliegt, wird als Spezifität bezeichnet. Sie wird aus dem Anteil negativer Klassifizierungen an den tatsächlich Negativen geschätzt [Wei08, S. 644]:

$$\widehat{\text{Spezifität}} = \frac{\text{RN}}{\text{RN} + \text{FP}}$$

(RN: Anzahl richtig negativer Klassifizierungen; FP: Anzahl falsch positiver Klassifizierungen).

2.3.2 Experimenteller Aufbau

Vor dem ersten Erlernen von Modellparametern wurden die verfügbaren Daten in Trainings- und Testdaten aufgeteilt, wie in Abschnitt 2.1.3 auf Seite 19 beschrieben. Wird ein Modell ausschließlich mit einem Teil der Daten trainiert, so können anschließend seine Klassifikationen der Testdaten, also der Beobachtungen, die beim Lernen der Modellparameter ausgeschlossen waren, mit den tatsächlichen Labels verglichen werden. Diese Validierung des trainierten Modells mit unbekannten Daten ermöglicht laut Duda, Hart und Stork [DHS01, S. 483] eine Schätzung

des erwarteten (verallgemeinerten) Klassifikationsfehlers. Der erwartete Klassifikationsfehler ist für die Bewertung der Leistungsfähigkeit eines Modells relevanter als der Klassifikationsfehler bei den Trainingsdaten, weil man in der Praxis ebenfalls in erster Linie an der Klassifizierung neuer Beobachtungen interessiert ist.

Vor dem endgültigen Training der zu vergleichenden Modelle wurden für jeden Ansatz ein oder mehrere Hyperparameter optimiert. Im Abschnitt 2.2 werden für alle Ansätze die jeweils optimierten Hyperparameter erwähnt. Die Optimierung der Hyperparameter erfolgte mittels Gittersuche über zuvor festgelegte Wertemengen. Bei einer Gittersuche wird das Modell mit allen Wertekombinationen der Hyperparametermengen trainiert, und anschließend wird die Kombination mit dem geringsten Klassifikationsfehler ausgewählt [Gér20, S. 78]. Für die vorliegende Arbeit habe ich über eine vierfache Kreuzvalidierung [DHS01, S. 483–484] die endgültigen Hyperparameterwerte ausgewählt. Dabei wurden pro Klassifikationsansatz diejenigen Werte gesucht, mit welchen jeweils die höchste mittlere Test-Genauigkeit erreicht wurde, ohne dass Konvergenzprobleme beim Training auftraten. Um weitgehend zu vermeiden, dass später in der Kreuzvalidierung zur endgültigen Bewertung der Klassifikationsleistung auf Daten getestet würde, die bereits während der Hyperparameteroptimierung verwendet wurden, wurden nur die Daten von ca. 40 % der Laktationen für die Optimierung herangezogen.

Anschließend wurden die Klassifikationsmodelle mit den optimierten Hyperparametern evaluiert. Die Evaluation erfolgte mittels zehnfacher Kreuzvalidierung, die von Cover [Cov69] eingeführt worden war. Dafür wurde jedes Modell jeweils mit neun der zehn Teildatensätze trainiert und dann die Klassifikationsleistung auf unbekannten Daten mit dem verbliebenen zehnten Teildatensatz getestet. Das wurde pro Klassifikationsansatz zehnmal wiederholt, so dass jeder Teildatensatz einmal als Testdatensatz verwendet wurde. Diese Variante der Kreuzvalidierung wird als Leave-One-Group-Out-Kreuzvalidierung bezeichnet. Laut Hastie, Tibshirani und Friedman [HTF09, S. 257] liefert die Kreuzvalidierung eine brauchbare Schätzung des erwarteten (verallgemeinerten) Klassifikationsfehlers. Allerdings sollte nicht nur der Punktschätzer der erwarteten Klassifikationsleistung berichtet werden, sondern auch der Standardfehler des Schätzers, da die Schätzung mittels der Kreuzvalidierung eine deutliche Variabiliät aufweisen kann [HTF09, S. 249].

Für die Gittersuche zur Hyperparameteroptimierung und die Kreuzvalidierung der neun Klassifikationsmodelle war es erforderlich, wiederholt dieselben Abläufe zu programmieren mit jeweils nur geringen Änderungen. Damit der gesamte experimentelle Aufbau nachvollziehbar, anpassbar und konsistent für alle neun Klassifikationsansätze war, habe ich ein Softwareframework entwickelt. So wurde die wiederholte Programmierung derselben Codezeilen vermieden und es war möglich, für jeden Ansatz nur den jeweils spezifischen Code zu programmieren.

High Performance Computing System

Sämtliche Experimente wurden auf dem High Performance Computing System Curta der Freien Universiät Berlin durchgeführt. Das System wurde von Bennett, Melchers und Proppe [BMP20] im Detail beschrieben. Kurz gesagt, besteht Curta aus

170 Standardrechenknoten und 12 Knoten mit Graphics Processing Units (GPUs). Jeder Knoten besitzt zwei Hauptprozessoren (Central Processing Unit (CPU)) mit jeweils 16 Rechenkernen (Intel® Xeon Gold 6130). Die GPU-Knoten haben zusätzlich jeweils zwei Grafikprozessoren (nvidia® Geforce 1080Ti). Als Betriebssystem kommt CentOS 7 Linux[8] zum Einsatz. Die Jobsteuerung erfolgt über Bash-Skripte mit dem Ressourcenmanager slurm[9]. Über slurm wurde für die vorliegende Arbeit auch der Arbeitsspeicherbedarf für das Training der Modelle erfasst. Mittels conda[10] wurde die verwendete Software in virtuellen Umgebungen bereitgestellt. Auf Curta wurde über ein virtuelles privates Netzwerk mittels Secure Shell (SSH) zugegriffen.

Verwendete Software

Der Programmcode für das Training und die Evaluierung der Klassifikationsansätze wurde in Python Version 3.8[11] [vRD09] geschrieben. Die Bibliothek numpy Version 1.20[12] [Har+20] wurde für die Berechnung der Spektren in den Ansätzen mit Feature Engineering verwendet. Für die Leave-One-Group-Out-Kreuzvalidierung während der Hyperparameteroptimierung sowie für die Random Forests und die SVMs kam die Bibliothek scikit-learn in Version 0.24[13] [Ped+11] zum Einsatz. Alle neuronalen Netze (MLP, CNN, GRU) wurden mit TensorFlow Version 2.4[14] [Aba+15; Aba+16; Ten21] entwickelt.

Wegen einer Inkompatibilität der Implementierung von GRU-Schichten in TensorFlow 2.4 und numpy 1.20 [PHJ+21] musste bei den Ansätzen E2E-GRU und AE-GRU auf die numpy-Version 1.19 ausgewichen werden.

Entwurf der Software für den Vergleich

Die Schnittstelle `TrainTestInterface` legte die Methodenköpfe der wesentlichen Methoden der Klassen zur Hyperparameteroptimierung (`Searcher`-Klassen) und zur Evaluierung (`Evaluator`-Klassen) fest (siehe Abbildung 2.6 auf der nächsten Seite). Mittels der Methode `import_data` sollten die erforderlichen Daten aus den vorbereiteten Dateien eingelesen werden. Die Klassenmethode `window_function`, deren Aufgabe es war, die Werte jeder einzelnen Beobachtung ggf. zu transformieren und das Datenformat anzupassen, sollte wiederum in einer Schleife über die Beobachtungen aus `import_data` heraus aufgerufen werden. Die Methode `prepare_fit` diente schließlich zur eigentlichen Modellerstellung und zum Training und Testen der Modelle. Außerdem sollten darin die für Training und Vorhersagen benötigten Zeiten erfasst und zusammen mit den Vergleichsmetriken (Genauigkeit, Relevanz, Sensitivität und Spezifität) ausgegeben werden. Der modular verschachtelte Entwurf der Methoden ermöglichte es, die für mehrere Ansätze benötigte

[8] `https://www.centos.org/`, abgerufen am 2022-07-04
[9] `https://slurm.schedmd.com`, abgerufen am 2022-07-04
[10] `https://anaconda.org/anaconda/conda`, abgerufen am 2022-08-03
[11] `https://www.python.org/`, abgerufen am 2022-05-30
[12] `https://numpy.org/`, abgerufen am 2022-07-16
[13] `https://scikit-learn.org/`, abgerufen am 2022-07-16
[14] `https://www.tensorflow.org/`, abgerufen am 2022-07-16

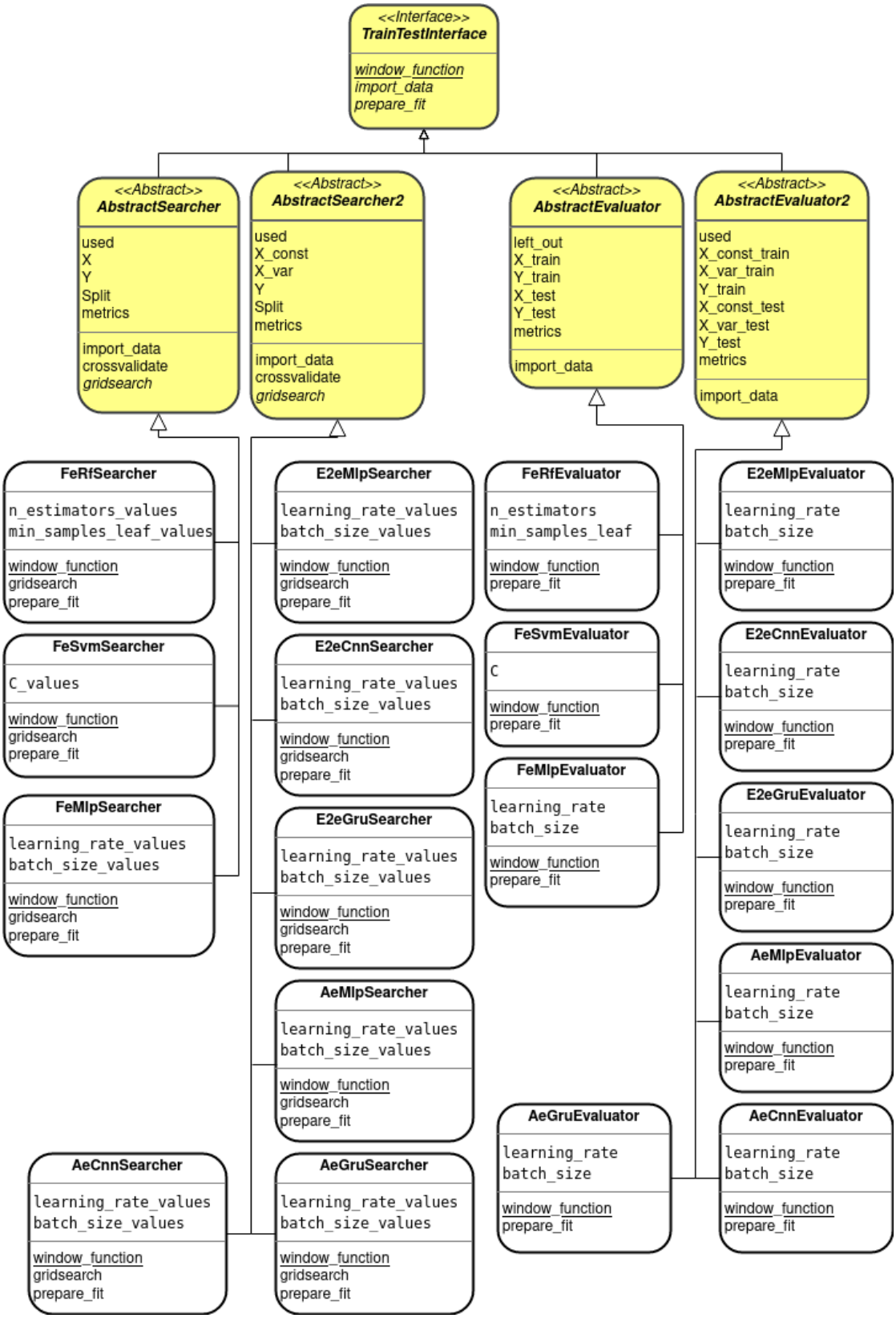

Abbildung 2.6: UML2-Klassendiagramm des Softwareframeworks zur Hyperparameteroptimierung und zur Evaluierung der Leistungen der Klassifikationsansätze

Methode `import_data` in abstrakten Klassen zu implementieren. So mussten nur die spezifischen Aufgaben in `window_function` und `prepare_fit` in den jeweiligen konkreten Spezialklassen pro Klassifikationsansatz programmiert werden.

Die Gittersuche mit Kreuzvalidierung zur Hyperparameteroptimierung wurde ähnlich verschachtelt entworfen (Abbildung 2.6 auf der vorherigen Seite). Die Schleife über die Teildatensätze für die Kreuzvalidierung sollte in den abstrakten Klassen `AbstractSearcher` und `AbstractSearcher2` als Methode `crossvalidate` implementiert werden. In jedem Schleifendurchlauf in `crossvalidate` sollte dann die modellspezifische Methode `prepare_fit` aufgerufen werden. Die Methode `gridsearch` mit den äußeren Schleifen über die möglichen Hyperparameterwerte musste ebenfalls modellspezifisch umgesetzt werden. Für jede mögliche Kombination der Hyperparameterwerte sollte dann `crossvalidate` aufgerufen werden.

Für die Evaluierung der Klassifikationsmodelle nach der Hyperparameteroptimierung sollte die Kreuzvalidierung nicht über Methoden realisiert werden, sondern über ein Job-Array[15], indem der Index des Teildatensatzes, welcher jeweils nur zum Testen verwendet werden sollte, als Argument über das Bash-Skript an das ausführbare Python-Skript übergeben werden sollte.

Die Vorbereitung der Daten für die Klassifikationsmodelle mit zwei Eingangsschichten der Ende-zu-Ende-Ansätze und der generativen Ansätze (siehe Abschnitte 2.2.1 und 2.2.2) erforderte separate Attribute für die konstanten Merkmale und die Zeitfenster. In den abstrakten Klassen `AbstractSearcher2` und `AbstractEvaluator2` sollten die entsprechenden Anpassungen der Attribute und der Methoden `import_data` vorgenommen werden (Abbildung 2.6 auf der vorherigen Seite).

Abstrakte Klassen

In den abstrakten Klassen wurden die Köpfe der Methoden definiert und Attribute und Methoden implementiert, die für die Untersuchung mehrerer Klassifikationsansätze benötigt wurden.

Modul `TrainTestInterface`

Python sieht von sich aus kein Schlüsselwort zur Schnittstellendefinition vor. Es ermöglicht aber, sehr leicht informelle Schnittstellen mittels *Duck Typing* zu erstellen. Beim Duck Typing wird darauf verzichtet, den Typ eines Objekts zu testen, stattdessen werden dessen Methoden und Attribute direkt verwendet. Es wird davon ausgegangen, dass ein Objekt, welches die gewünschten Attribute und Methoden hat, auch vom erforderlichen Typ ist. Dadurch wird es einfacher, durch Polymorphie das wiederholte Schreiben derselben Quellcodezeilen zu vermeiden [Pyt22, Glossary zu „duck-typing“]. Eine informelle Schnittstelle definiert die Köpfe der Methoden, die in den Nachkommensklassen zur konkreten Implementierung überschrieben werden können, ohne dass diese Implementierung zwingend erforderlich ist [Mur].

[15] `https://www.fu-berlin.de/sites/high-performance-computing/Dokumentation/Ressourcen-Manager/Job-Skripte/Job-Array/index.html`, abgerufen am 2022-07-05

Im Modul `TrainTestInterface` wurde die Schnittstelle `TrainTestInterface` informell umgesetzt (siehe Listing A.1 auf Seite 94 im Anhang). Die Methode `import_data` wurde als leere Methode erstellt. Damit sie in Methoden von abstrakten Nachkommensklassen verwendet werden konnten, lieferten die Methoden `window_function` und `perpare_fit` Rückgabewerte im erwarteten Format. Dadurch war es möglich, Methoden von abstrakten Klassen zu testen, ohne sämtliche Methoden bereits vollständig zu programmieren.

Module `AbstractSearcher` und `AbstractSearcher2`

In den abstrakten Nachkommensklassen von `TrainTestInterface`, `AbstractSearcher` und `AbstractSearcher2`, wurden die Gittersuchen mit Kreuzvalidierung für die Optimierung der Hyperparameter für die Klassifikationsansätze mit Feature Engineering bzw. für die Ende-zu-Ende-Ansätze und die generativen Ansätze vorbereitet und das Einlesen der Daten implementiert. Die Methode `import_data` wurde so umgesetzt, dass mehrere Teildatensätze parallel eingelesen und vorbereitet werden konnten (siehe Listings A.2 auf Seite 96 und A.3 auf Seite 100 im Anhang). Dafür wurde auf die Klasse `Pool`[16] aus dem Pythonmodul `multiprocessing.pool` zurückgegriffen. Ein `Pool` besteht aus mehreren Prozessen, die für die synchrone und asynchrone Bearbeitung von Aufgaben zur Verfügung stehen (äquivalent zu Threads in Java). Eine Besonderheit der Klasse `Pool` ist es, dass sie die Möglichkeit bietet, die Eingabedaten auf die unterschiedlichen Prozesse zu verteilen. Damit erlaubt sie Datenparallelität bei der Verarbeitung, was gerade beim Einlesen und Vorbereiten großer Datenmengen wie für die vorliegende Arbeit eine deutliche Beschleunigung ermöglicht. Für die datenparallele Verarbeitung wurde das Einlesen eines einzelnen Teildatensatzes in die separate Methode `_get_split` ausgelagert, die wiederum die zusätzliche Methode `_get_reshaped_data` aufrief, um die Merkmalswerte pro Beobachtung in der Form zurückzugeben, die von den Klassifikationsansätzen benötigt wurde. In `_get_reshaped_data` wurden die Zeitfenster aus den Zeitreihen der Variablen `MilkYield`, `StepsPerHour` und `LyingDuration` pro Kuh und Laktation gebildet und mit den entsprechenden Werten der konstanten Merkmale `Lactation` und `DaysInMilk` sowie gegebenenfalls einem Label zu einer Beobachtung verknüpft. Um die Variablenwerte in die ansatzspezifische Form zu bringen und bei Ansätzen mit Feature Engineering zusätzliche Merkmale zu erzeugen, wurde in `_get_reshaped_data` die Methode `window_function` auf die Merkmale jeder Beobachtung angewandt. `window_function` selbst wurde in `AbstractSearcher` und `AbstractSearcher2` nicht überschrieben. Die eingelesenen Daten wurden schließlich in den Attributen für die Merkmalswerte `AbstractSearcher.X` bzw. `AbstractSearcher2.X_const` und `AbstractSearcher2.X_var` sowie für die Labels (`Y` in beiden Klassen) gespeichert.

Für die Gittersuche zur Hyperparameteroptimierung wurde bereits die Methode `gridsearch` angelegt, die allerdings in den konkreten Klassen über-

[16] `https://docs.python.org/3.8/library/multiprocessing.html#multiprocessing.pool.Pool`, abgerufen am 2022-07-12

schrieben werden musste, um Suchen über die spezifischen Hyperparameterräume durchzuführen. Als Ergebnis einer Gittersuche sollte `gridsearch` die Mittelwerte und Standardabweichungen der Vergleichskriterien aus der Kreuzvalidierung pro Tupel von Hyperparameterwerten im Attribut `metrics` ablegen. Jedes Tupel von Hyperparameterwerten wurde zur Kreuzvalidierung an die Methode `crossvalidate` übergeben. Mit einem Objekt der Klasse `sklearn.model_selection.LeaveOneGroupOut`[17] wurden für jeden Kreuzvalidierungsdurchgang die Daten eines Teildatensatzes als Testdaten ausgewählt. Die übrigen Daten wurden als Trainingsdaten verwendet. Trainings- und Testdaten wurden an die Methode `prepare_fit` übergeben, die in einer konkreten Klasse spezifisch für jeden Klassifikationsansatz implementiert werden musste. Von `prepare_fit` erhielt `crossvalidate` die Werte der Vergleichskriterien (Genauigkeit, Relevanz, Sensitivität, Spezifität, Trainings- und Testzeit) dieses Durchgangs zurück. Die Mittelwerte und Standardabweichungen der Vergleichskriterien aller Kreuzvalidierungsdurchgänge wurden dann wiederum von `crossvalidate` zurückgegeben.

Die Klasse `AbstractSearcher` diente zur Hyperparameteroptimierung für die Klassifikationsansätze mit Feature Engineering (vgl. Abbildung 2.6 auf Seite 34). In diesem Fall wurden alle Merkmalswerte pro Beobachtung in einem Vektor zusammengefasst. Beobachtungen ohne Label konnten nicht verarbeitet werden und wurden ausgeschlossen. Bei einigen Ende-zu-Ende- und generativen Ansätzen mussten die Werte der zeitabhängigen Merkmale pro Beobachtung als Matrix getrennt vom Vektor der Werte der konstanten Merkmale an die Modelle übergeben werden. Außerdem sollten in den generativen Ansätzen auch Beobachtungen ohne Labels verwendet werden. Aus diesen Gründen gab es für diese Ansätze die Klasse `AbstractSearcher2`, in der die Daten der konstanten und der zeitabhängigen Merkmale in getrennten Attributen gespeichert wurden und unterschiedlich behandelt werden konnten. Dazu mussten über `import_data` und `_get_split` die Positionen der Spalten der konstanten bzw. zeitabhängigen Merkmalswerte in den Eingabedaten an `_get_reshaped_data` übergeben werden. Auf demselben Weg wurde die Information übermittelt, ob alle oder nur die mit einem Label versehenen Beobachtungen verwendet werden sollten. Natürlich musste auch `crossvalidate` in `AbstractSearcher2` separat implementiert werden, um die getrennten Attribute für konstante und zeitabhängige Merkmalswerte zu verwenden.

Module `AbstractEvaluator` und `AbstractEvaluator2`

In den abstrakten Klassen `AbstractEvaluator` und `AbstractEvaluator2` als Nachkommen von `TrainTestInterface` für die Evaluierung der Klassifikationsmodelle wurde jeweils die Methode `import_data` überschrieben (siehe Abbildung 2.6 auf Seite 34 sowie Listings A.4 auf Seite 105 bzw. A.5 auf Seite 108 im Anhang). Bei der Evaluierung der Modelle mit den optimierten Hyperparametern erfolgte die Kreuzvalidierung nicht über Methoden sondern mittels Übergabeparametern

[17] `https://scikit-learn.org/0.24/modules/generated/sklearn.model_selection.LeaveOneGroupOut.html`, abgerufen am 2022-07-12

bei der Ausführung der Python-Skripte in einem slurm-Job-Array wie auf Seite 35 beschrieben. Daher musste die Information über die Teildatensätze für die Kreuzvalidierung nicht in Attributen gespeichert werden, wie es bei `AbstractSearcher` und `AbstractSearcher2` der Fall war, sondern es wurden nur die Daten für einen Kreuzvalidierungsdurchgang in Attributen gespeichert (siehe Abbildung 2.6 auf Seite 34). Wie in den Klassen für die Hyperparameteroptimierung wurde das Einlesen eines Teildatensatzes in die Methode `_get_split` ausgelagert zur datenparallelen Ausführung mit einem `multiprocessing.pool.Pool` (siehe S. 36). Ebenfalls wie dort erfolgte die Zusammenstellung der Werte für eine Beobachtung mit der Methode `_get_reshaped_data`. Allerdings wurden von den Methoden der Evaluierungsklassen im Gegensatz zu den Methoden von `AbstractSearcher` und `AbstractSearcher2` keine Informationen über den Teildatensatz einer Beobachtung zurückgegeben. Ein weiterer Unterschied zwischen den Klassen zur Hyperparameteroptimierung und zur Evaluierung war, dass bei letzteren nur das Einlesen der Trainingsdaten parallelisiert wurde. Der einzelne Teildatensatz, der zum Testen verwendet wurde, wurde separat eingelesen.

Die Unterschiede zwischen `AbstractEvaluator` und `AbstractEvaluator2` entsprachen denen zwischen `AbstractSearcher` und `AbstractSearcher2` bezüglich der gemeinsamen oder getrennten Behandlung von Werten konstanter und zeitabhängiger Merkmale und bezüglich der Möglichkeit, Beobachtungen ohne Label einzubeziehen.

Softwaretests der abstrakten Klassen

Vor der Implementierung der konkreten Klassen für die spezifischen Modelle wurden die Methoden zum Datenimport der vier abstrakten Klassen separat getestet. Für die Softwaretests wurden jeweils kleinere Datensätze erzeugt aus den Zeilen 100 bis 500 der zehn Teildatensätze. Die Methoden `AbstractSearcher.import_data` und `AbstractSearcher2.import_data` wurden auf vier dieser kleineren Datensätze getestet. Bei den Tests der Importmethoden von `AbstractEvaluator` und `AbstractEvaluator2` wurde einer der kleineren Datensätze als Testdatensatz eingelesen, die übrigen zusammen als Trainingsdatensätze. Es wurde der Datenimport für die Evaluierung sowohl aller als auch nur der Beobachtungen mit Labels getestet. Um die Softwaretests zu bestehen, mussten die Größen in allen Dimensionen der Attribute für die Merkmalswerte, Labels und ggf. Teildatensätze von Objekten der vier Klassen nach dem Einlesen der Daten mit den entsprechenden Größen übereinstimmen, die vorher durch Gruppierung und Filtern der Daten in interaktiven Pythonsitzungen bestimmt wurden. Der Programmcode der abstrakten Klassen wurde solange korrigiert, bis die Tests bestanden waren.

Modul `winfunc`

Das Modul `winfunc` (Listing A.6 auf Seite 111 im Anhang) stellte die Funktion `add_spectra_and_sd` zur Verfügung, die innerhalb der Methode `window_function` der konkreten Klassen für die Klassifikationsansätze mit Feature

Engineering verwendet wurde. Die Funktion `add_spectra_and_sd` erweiterte den übergebenen Merkmalsvektor einer Beobachtung um die Spektren und Standardabweichungen der Werte der zeitabhängigen Merkmale und gab den so vergrößerten Vektor zurück. Zur Berechnung der Spektren wurde die eindimensionale diskrete Fouriertransformation jeweils auf die Werte eines Merkmals in einem Zeitfensters angewandt, und zwar in Form des Fast Fourier Transfrom Algorithmus [CT65], wie er im Modul `fft` des Pythonpakets `numpy` [Har+20] implementiert ist[18]. Anschließend wurde die Komponente für die Frequenz null ins Zentrum des Spektrums verschoben. Um im reellen Zahlenraum zu bleiben, wurden die absoluten Werte des Spektrums zurückgegeben.

Konkrete Klassen in ausführbaren Skripten

Der vollständige Quellcode der Skripte befindet sich im Anhang ab Seite 113. In den Skripten wurden zunächst die konkreten Nachkommensklassen der abstrakten Klassen implementiert. Die konkreten Klassen für die Ansätze mit Feature Engineering überschrieben die Klassenmethode `window_function`, die sie von `TrainTestInterface` geerbt hatten, mit einer Version, welche `winfunc.add_spectra_and_sd` verwendete, um zusätzliche Merkmale zu erzeugen. In den konkreten Klassen der Ansätze E2E-MLP und AE-MLP wurde `window_function` ebenfalls überschrieben, um die Merkmalswerte pro Beobachtung in einem Vektor bereitzustellen. Zur Gittersuche der Hyperparameterwerte wurden in den konkreten Klassen die von `AbstractSearcher` bzw. von `AbstractSearcher2` geerbten Methoden `gridsearch` überschrieben, um sie für die modellspezifischen Hyperparameter zu programmieren. Alle konkreten Klassen enthielten eine spezifische Version der Methode `prepare_fit`, mit welcher die von `TrainTestInterface` geerbte Verion überschrieben wurde. Innerhalb von `prepare_fit` wurden die Daten standardisiert (außer bei FE-RF) und die Modelle erstellt, trainiert und evaluiert. Die Pseudozufallszahlengeneratoren wurden immer mit demselben Wert initialisiert, um bei verschiedenen Durchläufen der Skripte immer dieselben Werte zu erzeugen. Während des Trainings und der Evaluierung wurden die benötigten Zeiten erfasst. Nachdem die Vorhersagen des trainierten Modells für die jeweiligen Testdaten geprüft waren, wurden die Vergleichsmetriken Genauigkeit, Relevanz, Sensitivität und Spezifität berechnet und mit den erfassten Zeiten zurückgegeben.

Nach der Definition der jeweiligen konkreten Klasse wurden im Skript die erforderlichen Variablenwerte festgelegt und ein Objekt der konkreten Klasse erzeugt. Beim Aufruf der Skripte zur Modellevaluierung wurde ein Parameter übergeben, der den Teildatensatz identifizierte, welcher als Testdatensatz verwendet wurde. Mit der geerbten Methode `import_data` wurden dem Objekt die Daten hinzugefügt. Anschließend wurden die Gittersuche zur Hyperparameteroptimierung durchgeführt oder durch direkten Aufruf von `prepare_fit` das jeweilige Modell einmal

[18] `https://numpy.org/doc/1.20/reference/generated/numpy.fft.fft.html`, abgerufen am 2022-07-13

trainiert und getestet. Die Ergebnisse wurden abschließend in Textdateien gespeichert (während der Hyperparametersuche) bzw. ausgegeben (für die Evaluierung).

Besonderheiten bei den neuronalen Netzen

Beim Training aller neuronalen Netze (FE-MLP, sämtliche E2E- und AE-Modelle) wurde Early Stopping[19] zur Regularisierung eingesetzt, um eine Überanpassung an die Trainingsdaten zu verhindern [vgl. GBC16, S. 241–249]. Dadurch wurde das Training beendet, wenn der Validierungsverlust für eine festgelegte Zahl von Epochen nicht sank.

Die neuronalen Netze der Ende-zu-Ende- und der generativen Ansätze wurden zusätzlich mit einer gesteuerten Lernrate trainiert. Wenn für eine bestimmte Zahl von Epochen der Validierungsverlust nur wenig abnahm, wurde die Lernrate deutlich reduziert[20]. Auf diese Weise sollten die Modelle schneller gute Parameterwerte lernen [Gér20, S. 363].

Unüberwachtes Vortraining in den generativen Ansätzen

Bei den generativen Ansätzen wurde zunächst der Autoencoder mit allen Trainingsdaten trainiert, wie in Listing 2.1 beispielhaft dargestellt. Anschließend wurde der Decoder durch den Klassifikator ersetzt. Bevor das so entstandene Modell mit den gelabelten Beobachtungen trainiert wurde, wurden die bereits gelernten Gewichte des Encoders eingefroren, sodass zunächst nur die Schichten des Klassifikators trainiert wurden und sich die Gewichte des Encoders nicht in Folge des anfänglich hohen Verlusts zu sehr veränderten. Erst nach diesem Trainingsdurchlauf wurden die Gewichte des Encoders wieder aufgetaut und im abschließenden Training des gesamten Modells für die Klassifikationsaufgabe angelernt. Für den abschließenden Trainingsdurchlauf wurde allerdings die Lernrate reduziert [vgl. Gér20, S. 350–353].

Listing 2.1: Python-Code für das unüberwachte Vortraining mit einem Autoencoder und das Training des Klassifikators im generativen Ansatz AE-GRU (vereinfachter Code)

```
# -----------------
# Encoder erstellen
# -----------------
input_const = Input(shape=x_const_train.shape[1:])
layer_const = Dense(2,activation='elu',
  kernel_initializer='he_normal')(input_const)
input_var = Input(shape=x_var_train.shape[1:])
rec1_var = GRU(units=16,return_sequences=True)(input_var)
rec2_var = GRU(units=16,return_sequences=True)(rec1_var)
rec3_var = GRU(units=16)(rec2_var)
join = concatenate([layer_const,rec3_var])
# Latente Abbildung
dense = Dense(6,activation='elu',kernel_initializer='he_normal')(join)
# -----------------
```

[19] `https://www.tensorflow.org/versions/r2.4/api_docs/python/tf/keras/callbacks/EarlyStopping`, abgerufen am 2022-07-13

[20] `https://www.tensorflow.org/versions/r2.4/api_docs/python/tf/keras/callbacks/ReduceLROnPlateau`, abgerufen am 2022-07-13

```
# Decoder erstellen
# -----------------
output_decoder_const = Dense(2)(dense)
de_rep = RepeatVector(27)(dense)
de_rec1_var = GRU(units=16,return_sequences=True)(de_rep)
de_rec2_var = GRU(units=16,return_sequences=True)(de_rec1_var)
de_rec3_var = GRU(units=16,return_sequences=True)(de_rec2_var)
output_decoder_var = TimeDistributed(Dense(3))(de_rec3_var)
# ----------------------
# Autoencoder trainieren
# ----------------------
ae = Model([input_const,input_var],
  [output_decoder_const,output_decoder_var])
ae.compile(loss='huber_loss',
  optimizer=Nadam(learning_rate=learning_rate))
ae.fit([x_const_train,x_var_train],[x_const_train,x_var_train],
  validation_data=([x_const_test,x_var_test],[x_const_test,x_var_test]),
  batch_size=batch_size,epochs=n_epochs,
  callbacks=[ReduceLROnPlateau(),EarlyStopping()])
# ------------------------------------------------
# Reduktion der Daten auf Beobachtungen mit Label
# ------------------------------------------------
x_const_train_ = x_const_train[idx_labels_train]
x_var_train_ = x_var_train[idx_labels_train]
x_const_test_ = x_const_test[idx_labels_test]
x_var_test_ = x_var_test[idx_labels_test]
y_train_ = (y_train[idx_labels_train]).astype(int)
y_test_ = (y_test[idx_labels_test]).astype(int)
# ------------------------------------
# Decoder durch Klassifikator ersetzen
# ------------------------------------
c1 = Dense(4,kernel_initializer='he_normal',activation='elu')(dense)
c2 = Dense(4,kernel_initializer='he_normal',activation='elu')(c1)
out_layer = Dense(1,activation='sigmoid')(c2)
model = Model([input_const,input_var],out_layer)
# ------------------------
# Klassifikator trainieren
# ------------------------
# Trainierte Gewichte des Encoders einfrieren
for layer in model.layers[:-1]:
    layer.trainable = False
# Klassifikator zum ersten Mal trainieren
model.compile(loss='binary_crossentropy',
  optimizer=Nadam(learning_rate=learning_rate),metrics=['accuracy'])
model.fit([x_const_train_,x_var_train_],y_train_,
  validation_data=([x_const_test_,x_var_test_],y_test_),
  batch_size=batch_size,epochs=n_epochs,
  callbacks = [ReduceLROnPlateau(),EarlyStopping()])
# Gewichte des Encoders auftauen
for layer in model.layers[:-1]:
    layer.trainable = True
# Klassifikator mit reduzierter Lernrate endgültig trainieren
model.compile(loss='binary_crossentropy',
  optimizer=Nadam(learning_rate=learning_rate/10),metrics=['accuracy'])
model.fit([x_const_train_,x_var_train_],y_train_,
  validation_data=([x_const_test_,x_var_test_],y_test_),
  batch_size=batch_size,epochs=n_epochs,
  callbacks=[ReduceLROnPlateau(),EarlyStopping()])
```

2.3.3 Hyperparameteroptimierung, Training und Validierung der Modelle

Zur Optimierung der Hyperparameter wurden wie beschrieben mittels Gittersuche aus den Suchmengen diejenigen Werte ausgewählt, mit denen die höchste Genau-

igkeit zu erwarten war. Es wurden die Hyperparameterwerte für die endgültige Modellevaluierung verwendet, mit denen in den Kreuzvalidierungen während der Gittersuchen die größten mittleren Test-Genauigkeiten erzielt wurden. Bei den neuronalen Netzen (MLP, CNN, GRU) wurden zusätzlich mittels TensorBoard[21] Lernkurven während des Trainings der Modelle erstellt und visuell beurteilt [vgl. Smi18]. Dabei sollte insbesondere erkannt werden, ob die Parameterwerte des Modells während des Trainings konvergierten [vgl. Gér20, S. 362–363].

Bei der Evalierung der Modelle wurde die verwendete Hardware an die Möglichkeiten angepasst, die von den verwendeten Softwarebibliotheken angeboten wurden. So wurden zwar allen Ansätzen für die Evaluierung mindestens vier CPUs zur Verfügung gestellt, um den Datenimport zu beschleunigen. Für die Evaluierung von FE-RF wurden aber fünf CPUs zur Verfügung gestellt, weil `sklearn.ensemble.RandomForestClassifier` es ermöglicht, die Bäume eines Random Forest parallel auf mehreren Prozessoren zu lernen[22]. Im Gegensatz dazu bietet `sklearn.svm.SVC` keine Möglichkeit der Parallelisierung des Trainings einer SVM[23]. Anders als scikit-learn bietet TensorFlow die Möglichkeit, das Training neuronaler Netze durch die Nutzung von GPUs zu beschleunigen [Gér20, S. 689–702]. Daher wurden alle neuronalen Netze (FE-MLP, sämtliche E2E- und AE-Ansätze) auf GPU-Knoten evaluiert. Pro Evaluierung wurde eine GPU verwendet.

2.3.4 Schließende Statistik

Nur für die Metriken der Klassifikationsleistung (erwartete Genauigkeit, Relevanz, Sensitivität und Spezifität) wurde eine schließende statistische Analyse durchgeführt. Die verwendeten Vergleichmetriken sind nicht unabhängig voneinander. Wenn die Häufigkeit des Zielmerkmals in der Zielpopulation bekannt ist, können sie ineinander umgerechnet werden [vgl. Wei08, S. 642–646]. Allerdings ist die Häufigkeit von Lahmheit bei schwarzbunten Milchkühen in intensiven Milchproduktionsbetrieben in Deutschland nicht bekannt. Aufgrund der Abhängigkeiten von einander sollten Genauigkeit, Relevanz, Sensitivität und Spezifität als abhängige Variablen in einem gemeinsamen multivariaten Modell analysiert werden. Wie von Benavoli u. a. [Ben+17] empfohlen, wurde ein (grafisches) Bayes'sches multivariates lineares Modell erstellt. Die Annahme, dass die Werte der abhängigen Variablen im relevanten Bereich hinreichend normalverteilt waren, wurde vorher visuell mittels Quantil-Quantil-(Q-Q)-Grafiken überprüft.

Die Werte der Vergleichsmetriken, die für unterschiedliche Klassifikationsansätze in Kreuzvalidierungsdurchgängen mit demselben Teildatensätzen ermittelt wurden, waren ebenfalls nicht unabhängig voneinander. Der jeweils als Testdaten verwendete Teildatensatz wurde daher als Confounder in das statistische Modell aufgenommen. Für das Bayes'sches multivariate lineare Modell zum Vergleich der

[21] `https://www.tensorflow.org/tensorboard`, abgerufen am 2022-07-16

[22] `https://scikit-learn.org/0.24/modules/generated/sklearn.ensemble.RandomForestClassifier.html`, abgerufen am 2022-07-16

[23] `https://scikit-learn.org/0.24/modules/generated/sklearn.svm.SVC.html`, abgerufen am 2022-07-16

Klassifikationsansätze wurde das Modell für eine Bayes'sche multifaktorielle Analysis of Variance von Kruschke [Kru11, S. 519–520] auf ein multivariates Modell erweitert. Allerdings wurde darauf verzichtet, eine Interaktion zwischen Klassifikationansatz und Teildatensatz zu modellieren, weil angenommen wurde, dass sich die Effekte der beiden bestimmenden Variablen auf die abhängigen Variablen lediglich addierten [vgl. Kru11, S. 516].

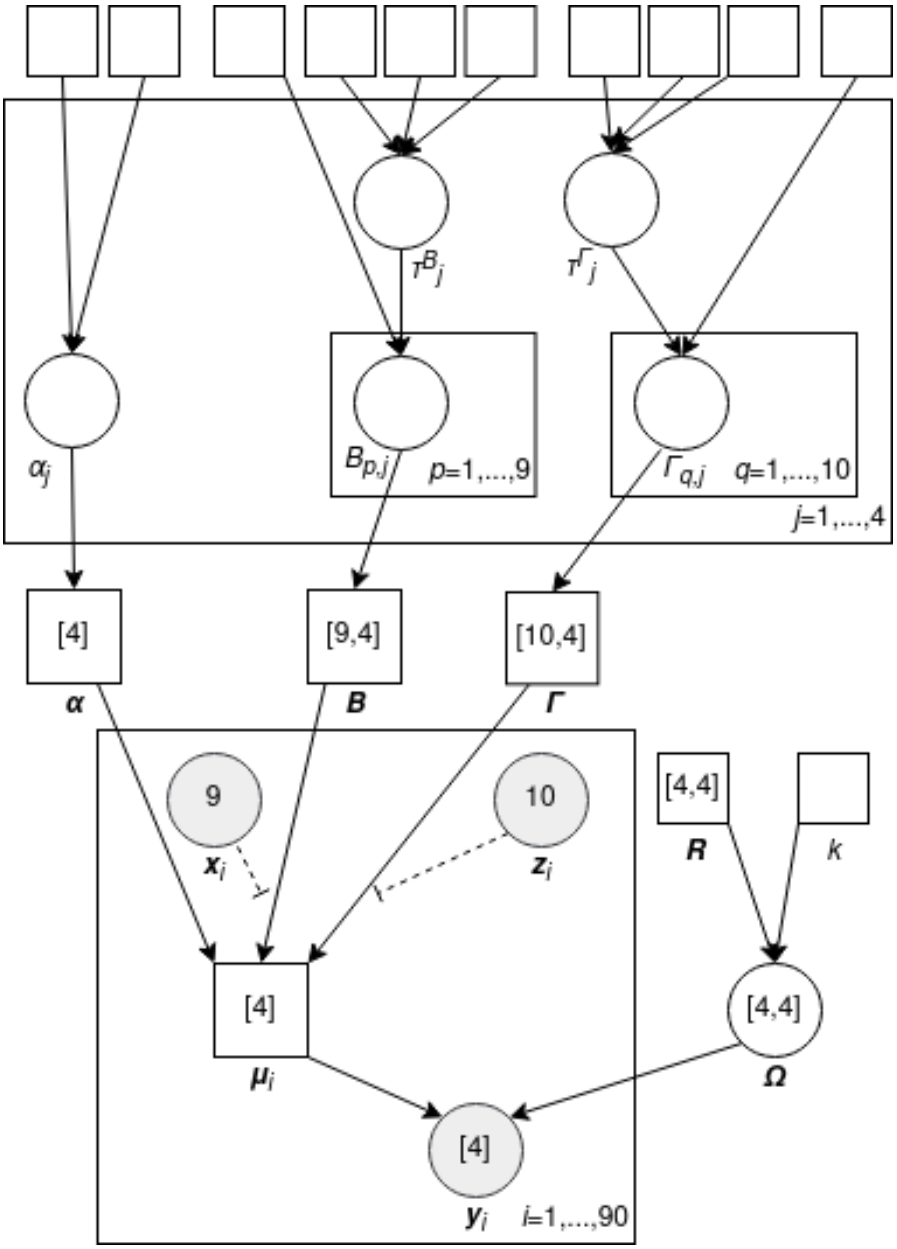

Abbildung 2.7: Bayes'sches multivariates lineares Modell zum Vergleich der Klassifikationsansätze in Plattennotation [erweitert nach Bun94]

Abbildung 2.7 auf der vorherigen Seite gibt einen Überblick über das Bayes'sches multivariate lineare Modell in erweiterter Plattennotation nach Buntine [Bun94]. Der Vektor $\boldsymbol{y}_i = (y_{1i}, y_{2i}, y_{3i}, y_{4i}), i = 1, \ldots, n$, fasste die Werte der vier Vergleichsmetriken (Genauigkeit, Relevanz, Sensitivität und Spezifität) als abhängige Variable pro Kreuzvalidierungsdurchgang und Klassifikationsansatz zusammen. Bei jeweils zehn Kreuzvalidierungsdurchgängen von neun Klassifikationsansätzen waren $n = 10 \cdot 9 = 90$ Beobachtungen für die schließende Statistik verfügbar. Der Klassifikationsansatz einer Beobachtung i wurde durch den Vektor $\boldsymbol{x}_i$ mit neun Elementen in One-Hot-Kodierung [Gér20, S. 68] dargestellt. Jedes Vektorelement entsprach einem Klassifikationsansatz, wobei nur das Element des bei i beobachteten Ansatzes den Wert 1 hatte, während alle übrigen Elemente den Wert 0 hatten. Entsprechend wurde der zum Testen verwendete Teildatensatz bzw. der Kreuzvalidierungsdurchgang durch den Vektor $\boldsymbol{z}_i$ mit zehn Elementen kodiert.

Als Likelihood-Funktion des Bayes'schen Modells wurde die Dichtefunktion einer multivariaten Normalverteilung mit den Parametern $\boldsymbol{\mu}$ und $\boldsymbol{\Omega}$ verwendet, d. h. es wurde angenommen, dass $\boldsymbol{y}$ multivariat normalverteilt war (Formel 2.1).

$$\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Omega}) \tag{2.1}$$

$$\boldsymbol{\mu} = \boldsymbol{\alpha} + \boldsymbol{x} \cdot \boldsymbol{B} + \boldsymbol{z} \cdot \boldsymbol{\Gamma} \tag{2.2}$$

Der Mittelwertsvektor $\boldsymbol{\mu}$ errechnete sich als Summe aus dem Vektor der Regressionskonstanten $\boldsymbol{\alpha}$ und den Vektor-Matrix-Produkten $\boldsymbol{x} \cdot \boldsymbol{B}$ und $\boldsymbol{z} \cdot \boldsymbol{\Gamma}$ der bestimmenden Variablen Klassifikationsansatz ($\boldsymbol{x}$) und Testdatensatz ($\boldsymbol{z}$) mit ihren jeweiligen Koeffizienten (Formel 2.2). Die Koeffizientenmatrizen $\boldsymbol{B}$ und $\boldsymbol{\Gamma}$ bestanden aus neun bzw. zehn Zeilen und vier Spalten.

Verteilungen der a-priori-Parameterwerte

Die Werte der 4×4 Präzisionsmatrix $\boldsymbol{\Omega}$ der multivariaten Normalverteilung folgten *a priori* einer Wishart-Verteilung mit einer Diagonalmatrix der Seitenlänge vier als Skalierungsmatrix $\boldsymbol{R}$ und $k = 5$ Freiheitsgraden. Durch diese Festlegung waren die Korrelationen zwischen den abhängigen Variablen *a priori* gleichverteilt zwischen -1 und $+1$ [vg. LL12, S. 111].

$$\boldsymbol{\Omega} \sim \text{Wishart}(\boldsymbol{R}, k) \tag{2.3}$$

Die Beziehungen der *a-priori*-Verteilungen der Modellparameter und ihrer Hyperparameter sind in Abbildung 2.7 auf der vorherigen Seite verdeutlicht. Die vier Elemente von $\boldsymbol{\alpha}$ waren *a priori* unabhängig normalverteilt mit Mittelwert null und einer Präzision von 0,001 (Formel 2.4). Für jeden der neun Klassifikationsansätze p waren die vier Elemente $\beta_{pj}, j = 1, \ldots, 4$, der entsprechenden Zeile $\boldsymbol{B}_{p,\bullet}$ jeweils *a priori* normalverteilt mit Mittelwert null und derselben Präzision τ_j^B (Formel 2.5 auf der nächsten Seite). Entsprechend waren die vier Elemente $\gamma_{qj}, j = 1, \ldots, 4$, der Zeile $\boldsymbol{\Gamma}_{q,\bullet}$ für einen Teildatensatz q jeweils *a priori* normalverteilt mit Mittelwert null und derselben Präzision τ_j^Γ (Formel 2.6 auf der nächsten Seite).

$$\alpha_j \sim \mathcal{N}(0,\ 0,001)\ ,\ j = 1, \ldots, 4 \tag{2.4}$$

$$B_{p,j} \sim \mathcal{N}(0, \tau_j^B)\ ,\ p = 1, \ldots, 9\ ,\ j = 1, \ldots, 4 \tag{2.5}$$

$$\Gamma_{q,j} \sim \mathcal{N}(0, \tau_j^\Gamma)\ ,\ q = 1, \ldots, 10\ ,\ j = 1, \ldots, 4 \tag{2.6}$$

Die jeweils vier Elemente der Hyperparametervektoren $\boldsymbol{\tau^B}$ und $\boldsymbol{\tau^\Gamma}$ waren *a priori* gefaltet t-verteilt mit Mittelwert null, Präzision 0,001 und zwei Freiheitsgraden wie von Gelman [Gel06] und Kruschke [Kru11, S. 493–495] empfohlen (Abbildung 2.7 auf Seite 43 und Formel 2.7). Die „Faltung“ der t-Verteilung wurde durch die Betragsbildung erreicht.

$$\tau_j^B \sim |t(0,\ 0,001,\ 2)|\ ,\ \tau_j^\Gamma \sim |t(0,\ 0,001,\ 2)|\ ,\ j = 1, \ldots, 4 \tag{2.7}$$

Schätzung der a-posteriori-Parameterverteilungen

Nach Standardisierung der Werte der abhängigen Variablen wurden die *a-posteriori*-Verteilungen der Modellparameter mittels Markov Chain Monte Carlo und Gibbs Sampling geschätzt. Dabei wurden vier Sampling-Ketten gebildet mit unterschiedlichen Startwerten und mit mindestens jeweils 100.000 Schritten nach einer Burn-in-Phase von 4.000 Schritten. Die Sampling-Ketten wurden anschließend auf jeden zehnten Schritt ausgedünnt. Mit Gelman-Rubin- [GR92] und Raftery-Lewis-Diagnostiken [RL92] wurde geprüft, ob die Markov-Chain-Monte-Carlo-Ziehungen konvergierten. Falls notwendig, wurden den Sampling-Ketten weitere Schritte hinzugefügt, bis Konvergenz erreicht war. Zusätzlich wurde der Verlauf der gezogenen Werte der Variablen für den Klassifikationsansatz in Grafiken über die Schritte und in Autokorrelationsgrafiken visuell beurteilt. Anschließend wurden die Modellannahmen in einer Residuenanalyse überprüft. Die Residuen jeder abhängigen Variablen wurden in Q-Q-Grafiken auf Normalverteilung getestet. In Abbildungen der Residuen über die vorhergesagten Werte pro abhängiger Variablen (Tukey-Anscombe-Grafiken) wurde auf Trends und Heteroskedastizität untersucht.

Die Werte der abhängigen Variablen aus den *a-posteriori*-Verteilungen wurden auf die ursprüngliche Skala zurücktransformiert. Die *a-posteriori*-Verteilungen der abhängigen Variablen wurden durch den Median und durch die Grenzen des Intervalls mit 95 % der Dichte der Verteilung beschrieben.

Tests auf Unterschiede zwischen den Klassifikationsansätzen

Zuerst wurde analyisert, ob sich die Verteilungen von Genauigkeit, Relevanz, Sensitivität und Spezifität eines Klassifikationsansatzes *a posteriori* jeweils von den übrigen unterscheiden. Dazu wurden für jede Vergleichsmetrik die Differenzen zwischen den *a-posteriori*-Verteilungen der untersuchten Klassifikationsansätze und der des theoretischen Mittelwerts mit einer Region Of Practical Equivalence (ROPE) von $0 \pm 0,01$ verglichen. Wenn weniger als 2,5 % der Verteilungsdichte der Differenz einer Vergleichsmetrik eines Klassifikationsansatzes innerhalb der ROPE lag, wurde der Schluss gezogen, dass sich dieser Ansatz signifikant von den übrigen unterschied in Bezug auf die jeweilige Vergleichsmetrik.

Zusätzlich wurden die *a-posteriori*-Verteilungen von Genauigkeit, Relevanz, Sensitivität und Spezifität aller Klassifikationsansätze jeweils paarweise verglichen.

Zwei Klassifikationsansätze wurden als signifikant unterschiedlich in Bezug auf eine Vergleichsmetrik angesehen, wenn die Dichte der Differenz der Verteilungen der Klassifikationsansätze für diese Metrik zu mehr als 2,5 % außerhalb einer ROPE von $-0,01$ bis $+0,01$ lag. Lag sie zu mehr als 97,5 % innerhalb der ROPE, so wurde geschlussfolgert, dass die beiden Klassifikationsansätze bezüglich dieser Metrik gleichwertig waren. Wie von Makowski u. a. [Mak+19] empfohlen, wurde immer der Anteil von der gesamten Dichte der Differenzverteilungen berücksichtigt.

Implementierung

Das Bayes'sche multivariate lineare Modell zum Vergleich der Klassifikationsansätze wurde in JAGS Version 4.3.0[24] [Plu03] programmiert (Listing 2.2). Die Beschreibung der Evaluationsergebnisse und die statistische Analyse erfolgte in R Version 4.2.1[25] [R C22]. Mit dem Paket runjags Version 2.2.1-7[26] [Den16] wurde JAGS aus R aufgerufen. Die Vergleiche der *a-posteriori*-Verteilungen mit der jeweiligen ROPE erfolgten mit dem Paket bayestestR Version 0.12.1[27] [MBL19]. In R wurden die Daten mit Hilfe der Funktionen des Pakets tidyverse[28] [Wic+19] vorbereitet. Die Grafiken wurden mit dem Paket ggplot2[29] [Wic16] erstellt.

Listing 2.2: JAGS-Code des Bayes'schen multivariaten linearen Modells zum Vergleich der Klassifikationsansätze

```
model {
  # ----------
  # Likelihood
  # ----------
  # für alle Beobachtungen
  for (i in 1:n) {
    # für die 4 Vergleichsmetriken separat
    for (j in 1:4) {
      # Mittelwert als Summe aus Regressionskonstante und den
      # Koeffizienten der bestimmenden Variablen
      # entsprechend deren Ausprägungen
      mu[i,j] <- alpha[j] + Beta[x[i],j] + Gamma[z[i],j]
    }
    ## Likelihood-Funktion: multivariate Normalverteilung
    y[i,1:4] ~ dmnorm(mu[i,], Omega)
  }
  # -----------------------
  # a-priori-Parameterwerte
  # -----------------------
  # für die 4 Vergleichsmetriken separat
  for (j in 1:4) {
    ## Regressionskonstante
    alpha[j] ~ dnorm(0, 0.001)
    ## Koeffizienten für die 9 Klassifikationsansätze
    for (p in 1:9) {
      Beta[p,j] ~ dnorm(0, tauBeta[j])
    }
    ## Koeffizienten für die 10 Teildatensätze
    for (q in 1:10) {
```

[24] https://mcmc-jags.sourceforge.io/, abgerufen am 2022-07-02
[25] https://www.r-project.org/, abgerufen am 2022-07-16
[26] https://cran.r-project.org/package=runjags, abgerufen am 2022-07-16
[27] https://cran.r-project.org/package=bayestestR, abgerufen am 2022-07-16
[28] https://cran.r-project.org/package=tidyverse, abgerufen am 2022-07-16
[29] https://cran.r-project.org/package=ggplot2, abgerufen am 2022-07-16

```
30        Gamma[q,j] ~ dnorm(0, tauGamma[j])
31      }
32      ## Hyperparameter tauBeta aus gefalteter t-Verteilung
33      tauBeta[j] <- 1 / pow(sBeta[j], 2)
34      # Betragsbildung ("Faltung")
35      sBeta[j] <- abs(sBetat[j]) + 0.1
36      # t-Verteilung
37      sBetat[j] ~ dt(0, 0.001, 2)
38      ## Hyperparameter tauGamma aus gefalteter t-Verteilung
39      tauGamma[j] <- 1 / pow(sGamma[j], 2)
40      # Betragsbildung ("Faltung")
41      sGamma[j] <- abs(sGammat[j]) + 0.1
42      # t-Verteilung
43      sGammat[j] ~ dt(0, 0.001, 2)
44    }
45    ## Präzisionsmatrix der multivariaten Normalverteilung
46    Omega ~ dwish(R, k)
47  }
```

3

Ergebnisse

Das Kapitel *Ergebnisse* beginnt mit der Beschreibung der verwendeten Daten und der Hyperparameterwerte, welche schließlich beim Vergleich der Klassifikationsansätze verwendet wurden. Anschließend werden die Ergebnisse der Evaluierungen der einzelnen Ansätze vorgestellt. Die Ergebnisse der schließenden Statistik über den Vergleich der Klassifikationsansätze bilden den Abschluss des Kapitels.

3.1 Beschreibung der verwendeten Daten

Tabelle 3.1: Zusammenfassung der Merkmalswerte der verwendeten Klauen*fit*net-Daten

Merkmal (Einheit)	Minimum	1. Quartil	2. Quartil	3. Quartil	Maximum
`Lactation`	1	1	2	3	10
`DaysInMilk` [d]	0	103	185	268	500
`MilkYield` [kg]	0,0	24,2	30,7	37,4	107,7
`StepsPerHour` [h^{-1}]	49	96	118	145	465
`LyingDuration` [min]	15	57	71	88	188

`Lactation`: Laktationsnummer, `DaysInMilk`: Laktationstag, `MilkYield`: Tagesgemelk, `StepsPerHour`: durchschnittliche Schrittfrequenz, `LyingDuration`: durchschnittliche Liegedauer pro Liegevorgang

Der verwendete Datensatz enthielt 727.008 Beobachtungen (Kuh-Tage) mit vollständigen Merkmalswerten aus 3.373 Laktationen von 2.528 Kühen aus vier Betrieben. Davon lagen zu 17.114 Beobachtungen aus 2.612 Laktationen Ergebnisse von Gangbeurteilungen als Klassenlabels vor. Bei 48,0 % der gelabelten Beobachtungen wurde eine Lahmheit diagnostiziert. Damit waren die Gangbeurteilungen nahezu balanciert. Tabelle 3.1 gibt einen Überblick über die Werte der Merkmale Laktationsnummer (`Lactation`), Laktationstag (`DaysInMilk`), Tagesgemelk (`MilkYield`), durchschnittliche Schrittfrequenz (`StepsPerHour`) und durchschnittliche Liegedauer pro Liegevorgang (`LyingDuration`).

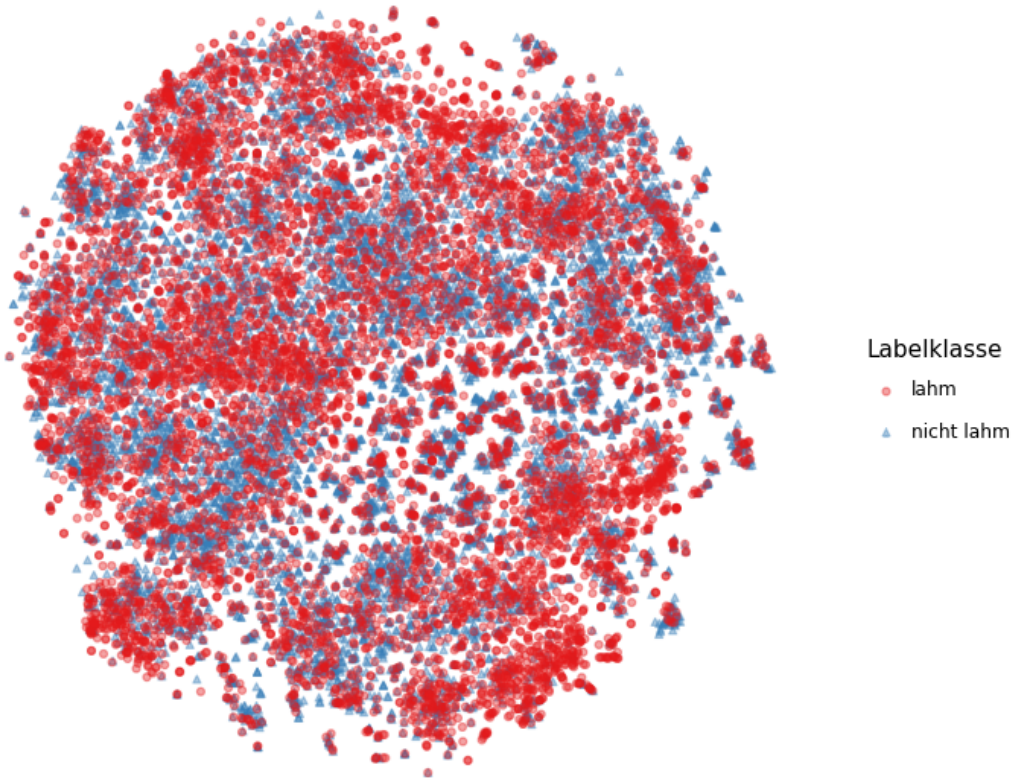

Abbildung 3.1: Visualisierung der verwendeten Daten mit Klassenlabels nach Dimensionsreduktion mittels t-distributed Stochastic Neighbor Embedding

In Abbildung 3.1 sind die verwendeten Daten nach Dimensionsreduktion mit t-distributed Stochastic Neighbor Embedding abgebildet. Eine Beobachtung bestand vor der Dimensionsreduktion aus einem 83-elementigen Vektor aus den Werten der konstanten Merkmale `Lactation` und `DaysInMilk` und den drei Zeitfenstern der Länge 27 für die Merkmale `MilkYield`, `StepsPerHour` und `LyingDuration`. Die Beobachtungen sind entsprechend ihrer Labels farblich markiert. In dieser Abbildung gibt es keine deutlich unterscheidbaren Häufungen der Beobachtungen der beiden Labelklassen „lahm“ und „nicht lahm“.

Die Aufteilung des gesamten Datensatzes in zehn Teildatensätze auf Ebene der Kühe ist in Tabelle 3.2 auf der nächsten Seite dargestellt. Alle Teildatensätze enthielten annähernd dieselbe Anzahl von Beobachtungen. Die Labelklassen waren allerdings nicht in allen Teildatensätzen gleichermaßen ausbalanciert. Die Teildatensätze zwei, drei, vier und sieben enthielten deutlich weniger Beobachtungen, die als „lahm“ markiert waren, als die anderen Teildatensätze.

Tabelle 3.2: Teilung der verwendeten Daten auf der Ebene der Kühe in zehn Teildatensätze für die Kreuzvalidierung der Klassifikationsansätze

Teildatensatz	Kühe	Beobachtungen	Häufigkeit der Labelklasse „lahm“
0	253	72.025	48,9 %
1	253	73.135	49,7 %
2	253	74.101	44,2 %
3	253	72.246	45,5 %
4	253	70.253	45,8 %
5	253	71.586	50,4 %
6	253	72.760	52,4 %
7	253	74.629	44,5 %
8	253	72.629	50,7 %
9	251	73.644	48,0 %

3.2 Hyperparameter der Klassifikationsmodelle

Die Hyperparameterwerte, die in den Gittersuchen in den besten erwarteten Genauigkeiten resultierten, sind für alle Klassifikationsansätze in Tabelle 3.3 auf der nächsten Seite angegeben zusammen mit den jeweiligen Wertemengen, aus denen gesucht wurde. Für AE-GRU wurde in der Gittersuche einer Batchgröße von 32 die höchste mittlere Test-Genauigkeit erreicht. Allergings divergierte das Modell während der anschließenden Evaluierung bei diesem Wert. Daher wurde für die endgültige Evaluierung auf eine Batchgröße von 512 ausgewichen, welche in der Gittersuche die zweitbeste erwartete Genauigkeit ergeben hatte.

3.3 Beschreibung der Evaluierungsergebnisse

Die Ergebnisse für alle Klassifikationsansätze und alle Kreuzvalidierungsdurchläufe sind in Tabelle B.1 auf Seite 174 im Anhang wiedergegeben. Die höchste Genauigkeit wurde mit 0,7563 von E2E-GRU beim Test mit Teildatensatz null erzielt, die niedrigste von E2E-MLP beim Test mit Teildatensatz drei mit 0,6461. Die Spannweite der erreichten Relevanz war etwas größer. Sie reichte von 0,6154 beim Test von AE-MLP auf Teildatensatz zwei bis 0,7856 beim Test von FE-SVM auf Teildatensatz null. Insgesamt wurde eine maximale Spezifität von 0,8280 und eine maximale Sensitivität von 0,7711 berechnet (Test von FE-SVM auf Teildatensatz null bzw. von E2E-CNN auf Teildatensatz acht). Die niedrigste Spezifität betrug 0,6468 (E2E-CNN, Teildatensatz acht), die niedrigste Sensitivität war 0,5497 (AE-MLP, Teildatensatz neun). Die Trainingsdauer reichte von unter sieben Sekunden beim Training von E2E-MLP mit den Daten ohne Teildatensatz acht bis über 22.000 sec beim Training von AE-CNN mit den Teildatensätzen eins bis neun. Im Gegensatz dazu unterschieden sich die Zeiten, die zur Klassifikation des Testdatensatzes benötigt wurden, deutlich weniger. Die Klassifikationen benötigten lediglich

Tabelle 3.3: Hyperparameterwerte für die neun Klassifikationsansätze in den Gittersuchen

Ansatz	Hyperparameter	Suchmenge	Ausgewählter Wert
FE-RF	`n_estimators`	{3.000; 3.500; 4.000; 4.500; 5.000}	4.500
	`min_samples_leaf`	{1; 5; 10; 20}	5
FE-SVM	`C`	{0,01; 0,1; 0,5; 1,0; 1,5; 3,0; 5,0; 10,0; 100,0}	0,5
FE-MLP	`learning_rate`	{0,0005; 0,001; 0,005; 0,01; 0,05}	0,0005
	`batch_size`	{8; 16; 32; 64; 128}	32
E2E-MLP	`learning_rate`	{0,0005; 0,001; 0,005; 0,01; 0,05}	0,005
	`batch_size`	{8; 16; 32; 64; 128}	128
E2E-CNN	`learning_rate`	{0,0005; 0,001; 0,005; 0,01; 0,05}	0,05
	`batch_size`	{8; 16; 32; 64; 128}	32
E2E-GRU	`learning_rate`	{0,0005; 0,001; 0,005; 0,01; 0,05}	0,001
	`batch_size`	{8; 16; 32; 64; 128}	32
AE-MLP	`learning_rate`	{0,001; 0,005; 0,01; 0,05}	0,001
	`batch_size`	{32; 128; 512}	128
AE-CNN	`learning_rate`	{0,001; 0,005; 0,01; 0,05}	0,001
	`batch_size`	{32; 128; 512}	32
AE-GRU	`learning_rate`	{0,001; 0,005; 0,01; 0,05}	0,005
	`batch_size`	{32; 128; 512}	32

ein paar Hunderstel Sekunden bis höchstens sechs Sekunden. Während des Trainings und des Testens benötigten die Klassifikationsansätze zwischen etwa einem halben und knapp drei Gigabyte Arbeitsspeicher.

3.3.1 Klassifikationsleistungen

Da für alle Klassifikationsansätze in der Kreuzvalidierung dieselben Teildatensätze verwendet wurden, erfolgte die Untersuchung der Klassifikationsleistungen sowohl vergleichend über die Klassifikationsansätze als auch vergleichend über die Teildatensätze.

Beobachtete Leistungen der Klassifikationsansätze

Die beobachteten Leistungen bei der Klassifikation unbekannter Daten sind für die neun Klassifikationsansätze in Tabelle 3.4 auf der nächsten Seite jeweils als Median und Interquartilsabstand zusammengefasst. Die Test-Genauigkeiten aller Ansätze mit Ausnahme von AE-MLP und E2E-MLP waren sehr ähnlich (siehe

Tabelle 3.4: Zusammengefasste Leistungen [Median (Interquartilsabstand)] bei bei der Klassifikation unbekannter Testdatensätze während der Evaluierung mit zehnfacher Kreuzvaldierung

Klassifikationsansatz	Genauigkeit	Relevanz	Sensitivität	Spezifität
AE-CNN	0.712 (0.013)	0.714 (0.043)	0.683 (0.039)	0.762 (0.085)
AE-GRU	0.706 (0.016)	0.712 (0.026)	0.672 (0.027)	0.746 (0.022)
AE-MLP	0.673 (0.026)	0.678 (0.025)	0.598 (0.057)	0.718 (0.021)
E2E-CNN	0.715 (0.013)	0.704 (0.032)	0.672 (0.048)	0.756 (0.072)
E2E-GRU	0.721 (0.022)	0.72 (0.038)	0.658 (0.025)	0.767 (0.039)
E2E-MLP	0.677 (0.026)	0.658 (0.042)	0.64 (0.054)	0.702 (0.02)
FE-MLP	0.704 (0.021)	0.711 (0.029)	0.663 (0.022)	0.752 (0.025)
FE-RF	0.721 (0.024)	0.717 (0.037)	0.653 (0.032)	0.769 (0.019)
FE-SVM	0.717 (0.021)	0.729 (0.024)	0.658 (0.029)	0.772 (0.017)

Abbildung 3.2 auf der nächsten Seite). Gleiches galt für die Test-Relevanz und in geringerem Maße auch für die Test-Sensitivität und Test-Spezifität. AE-MLP und E2E-MLP schnitten in allen Vergleichen der Klassifikationsleistungen am schlechtesten ab, AE-MLP mit einer schlechtern Sensitivität, E2E-MLP mit einer schlechteren Spezifität. Die höchsten mediane Test-Genauigkeiten erreichten E2E-GRU und FE-RF, allerdings mit weniger als 0,005 Abstand zu FE-SVM. Diese drei Ansätze hatten auch die höchste mediane Test-Relevanz, allerdings in anderer Reihenfolge. Die größte mediane Test-Relevanz von 0,729 erreichte FE-SVM mit Abständen von 0,009 zu E2E-GRU und von 0,012 zu FE-RF. Diese drei Ansätze hatten ebenfalls die höchsten medianen Test-Spezifitäten (FE-SVM: 0,772; FE-RF: 0,769; E2E-GRU: 0,767).

Ganz andere Klassifikationsansätze lieferten die höchsten medianen Test-Sensitivitäten von 0,683 (AE-CNN) und von 0,672 (AE-GRU und E2E-CNN). Die medianen Test-Sensitivitäten der drei Ansätze mit den höchsten medianen Test-Genauigkeiten und -Relevanzen (E2E-GRU, FE-RF und FE-SVM) lagen in der unteren Hälfte (Tabelle 3.4). AE-CNN und E2E-CNN erreichten nicht nur die besten medianen Test-Sensitivitäten, sondern hatten auch mediane Test-Spezifitäten auf dem vierten und fünften Rang (0,762 bzw. 0,756).

Beobachtete Klassifikationsleistungen pro Testdatensatz

Die erzielten Klassifikationsleistungen unterschieden sich deutlich voneinander, je nachdem, welcher Teildatensatz jeweils zum Testen verwendet worden war (Abbildung 3.3 auf Seite 54). Auf Teildatensatz neun wurden die höchsten medianen Leistungen erreicht. Mit den Testdatensätzen sechs und sieben wurden ebenfalls hohe mediane Klassifikationsleistungen erreicht. Am schlechtesten schnitten die Klassifikationen auf den Testdatensätzen drei und acht ab.

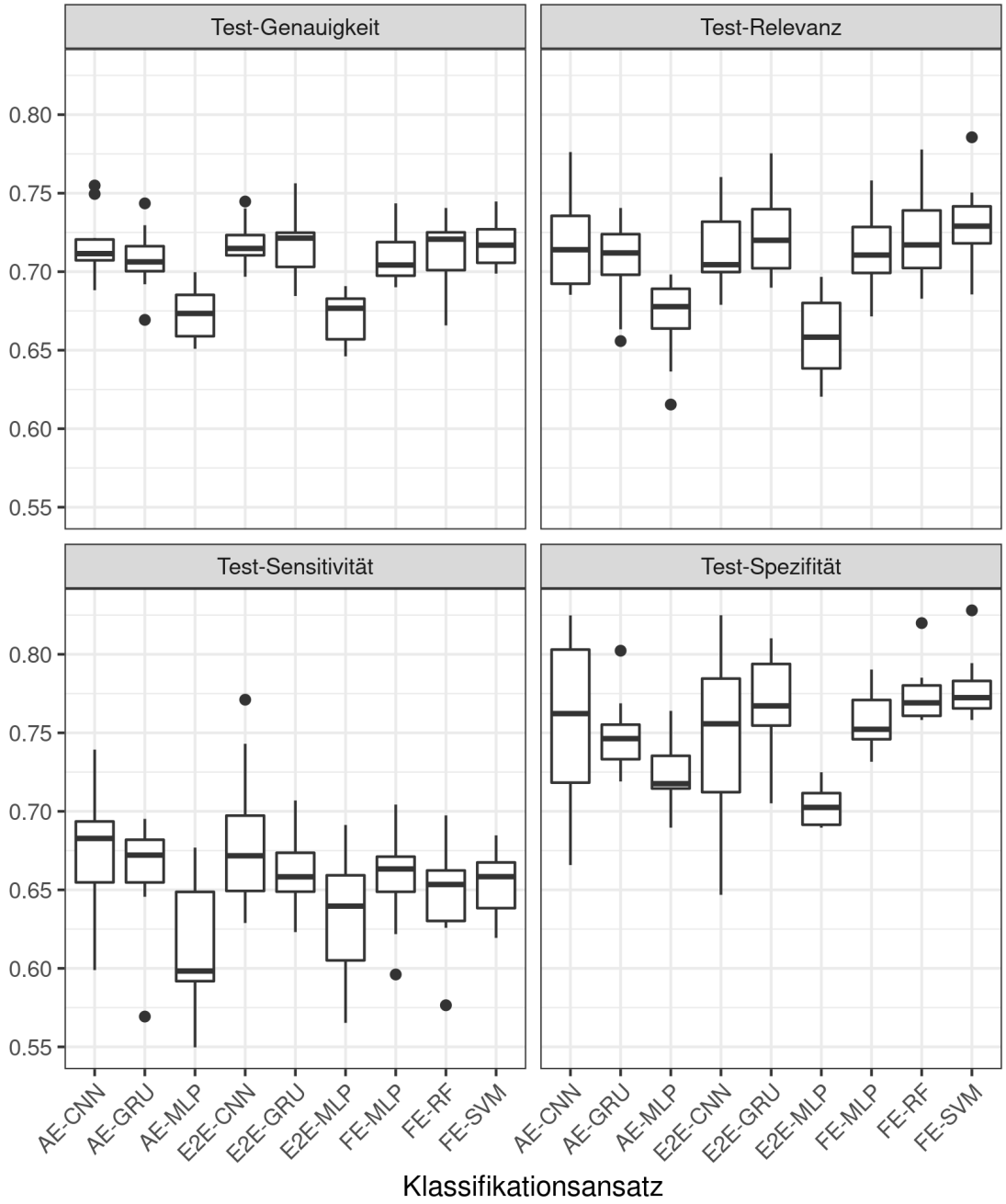

Abbildung 3.2: Beobachtete Leistungen der untersuchten Klassifikationsansätze in den zehn Kreuzvalidierungsdurchläufen

3.3.2 Ressourcenbedarf

Die untersuchten Klassifikationsansätze wiesen sehr starke Unterschiede auf hinsichtlich ihres Bedarfs an Ressourcen für das Training und das Testen (Abbildung 3.4 auf Seite 55 und Tabelle 3.5 auf Seite 57). Im Gegensatz zu den Klassifikationsleistungen zeigten sich beim Ressourcenbedarf die Gruppen der Klas-

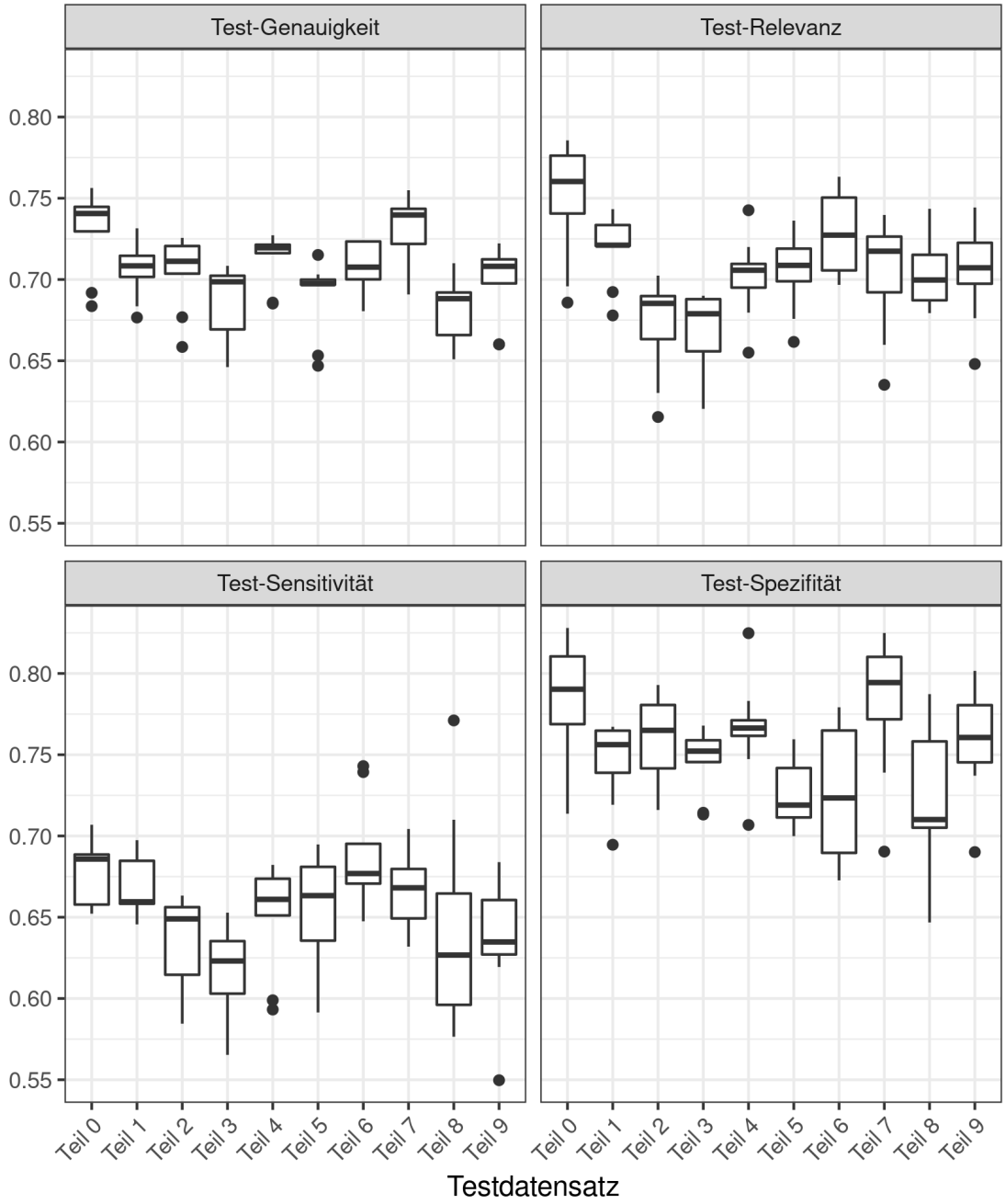

Abbildung 3.3: Beobachtete Leistungen bei den Klassifizierungen der zehn Testdatensätze über alle Ansätze zusammengefasst

sifikationsansätze. Die generativen Ansätze (AE-MLP, AE-CNN und AE-GRU), die große neuronale Netze und ein unüberwachtes Vortraining enthielten, benötigten die zwei- bis dreifache Menge an Arbeitsspeicher und zehn- bis hundertmal mehr Zeit zum Erlernen der Parameter. Unbekannte Datensätze wurden nach dem Training von ihnen dann allerdings sehr schnell klassifiziert.

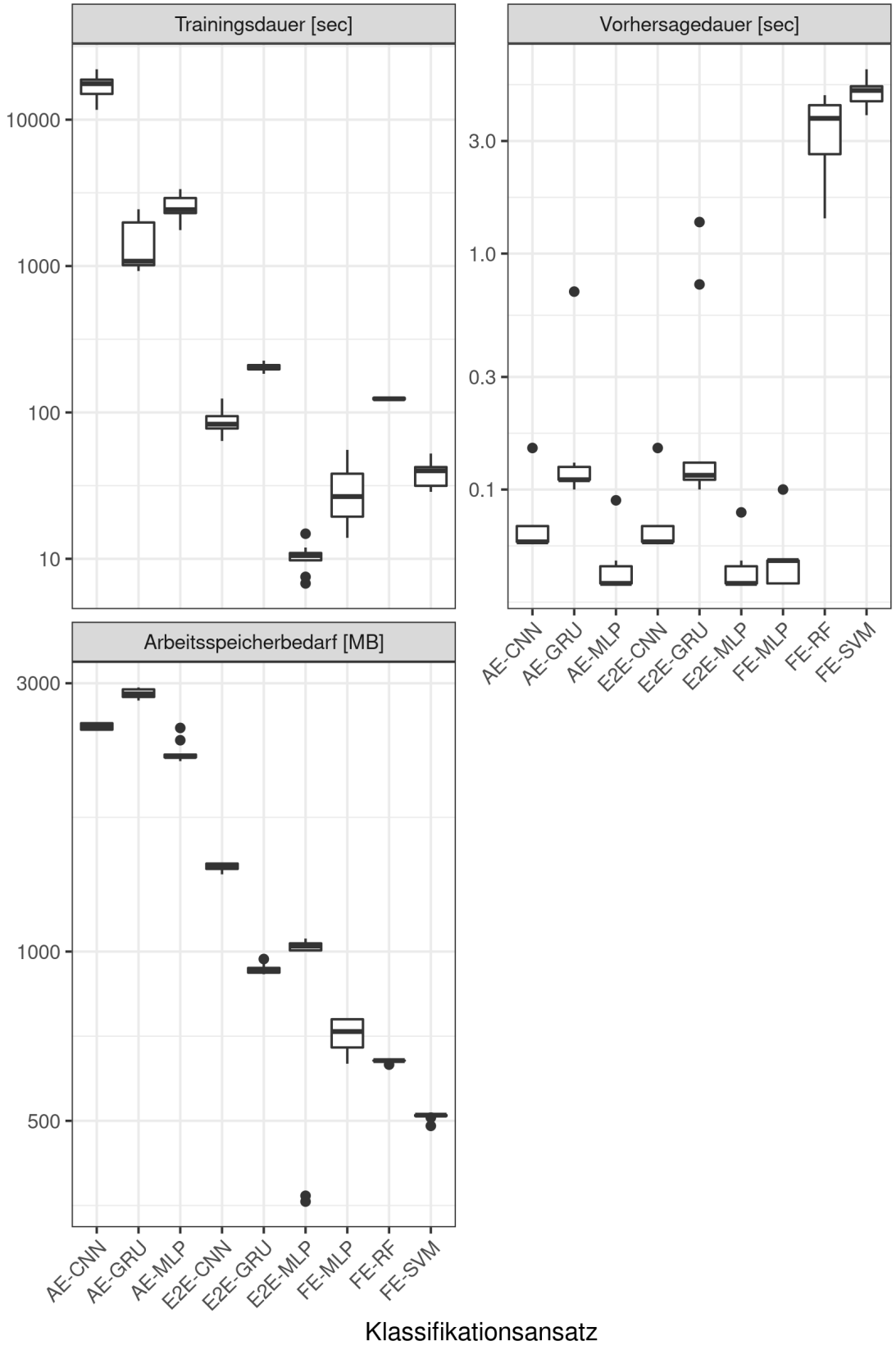

Abbildung 3.4: Laufzeiten und Arbeitsspeicherbedarf der untersuchten Klassifikationsansätze in den zehn Kreuzvalidierungsdurchläufen

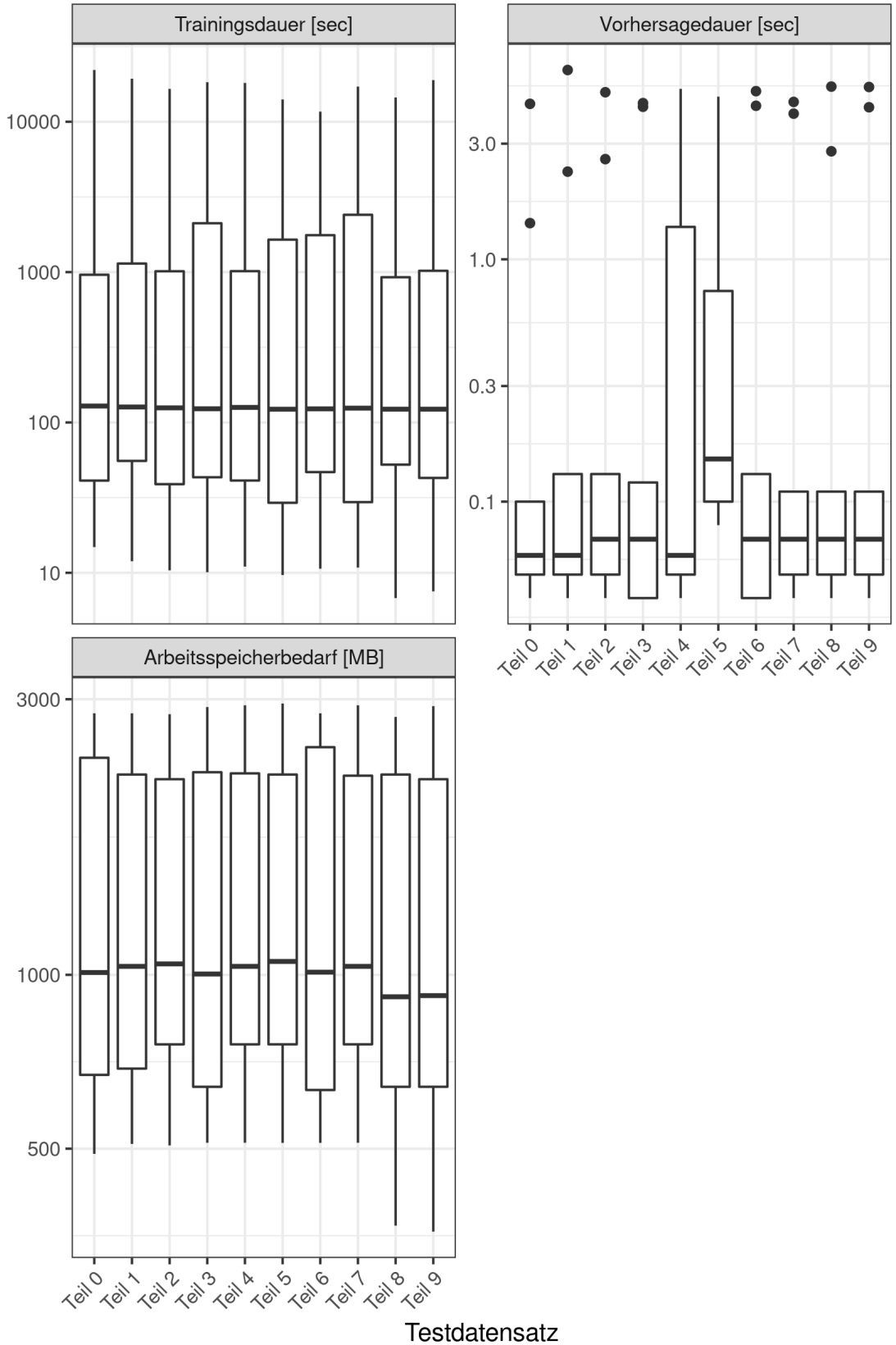

Abbildung 3.5: Laufzeiten und Arbeitsspeicherbedarf für Training und Klassifikation pro Testdatensatz über alle Ansätze zusammengefasst

Tabelle 3.5: Ressourcenbedarf [Median (Interquartilsabstand)] während der Evaluierung mit zehnfacher Kreuzvaldierung

Klassifikationsansatz	Trainingsdauer [sec]	Vorhersagedauer [sec]	Arbeitspeicher [MB]
AE-CNN	17626.56 (3752.91)	0.06 (0.01)	2513.92 (66.56)
AE-GRU	1080.3 (982.81)	0.11 (0.01)	2872.32 (89.6)
AE-MLP	2427.43 (613.91)	0.04 (0.01)	2222.08 (25.6)
E2E-CNN	83.08 (16.7)	0.06 (0.01)	1418.24 (30.72)
E2E-GRU	200.62 (13.37)	0.11 (0.02)	927.28 (18)
E2E-MLP	10.52 (1.19)	0.04 (0.01)	1022.43 (29.47)
FE-MLP	26.88 (18.93)	0.05 (0.01)	720.9 (82.59)
FE-RF	123.9 (2.98)	3.75 (1.62)	639.83 (0.16)
FE-SVM	39.97 (10.6)	4.92 (0.69)	511.79 (1.81)

Die Ansätze mit Feature Engineering und Klassifikationsmethoden aus dem klassischen maschinellen Lernen (FE-RF und FE-SVM) hatten hingegen nur einen sehr geringen Arbeitsspeicherbedarf und benötigten im Median 40 sec bis zwei Minuten, um die Parameter zu lernen. Allerdings dauerte es bei ihnen etwa 100-mal solange wie bei den Ansätzen, die auf neuronalen Netzen basierten, um unbekannte Testdaten zu klassifizieren. Selbst der maximale Zeitbedarf für die Klassifikation eines Testdatensatzes hatte aber nur wenig über 5 sec betragen (siehe Tabelle B.1 auf Seite 174 im Anhang).

Der Arbeitsspeicherbedarf von FE-MLP und der Ende-zu-Ende-Klassifikationsansätze lag zwischen diesen beiden Gruppen. Ihre Trainingsdauer war ähnlich der von FE-RF und FE-SVM, während die Zeiten, welche sie für die Klassifikation der Testdaten benötigten, denjenigen der generativen Ansätze ähnelten (Abbildung 3.4 auf Seite 55).

Der Zeitbedarf für das Lernen der Modellparameter und der benötigte Arbeitspeicherplatz waren nahezu konstant über alle Teildatensätze (Abbildung 3.5 auf der vorherigen Seite). Es dauert allerdings länger als bei den übrigen Teildatensätzen, die Daten der Testdatensätze vier und fünf zu klassifizieren.

3.4 Multivariates Bayes'sches lineares Modell

Die Schätzung der *a-posteriori*-Verteilungen der Modellparameter mittels Markov Chain Monte Carlo und Gibbs Sampling konvergierte innerhalb der ursprünglich eingegebenen 100.000 Schritte pro Kette. Pro abhängiger Variablen waren die Residuen normalverteilt und visuell wurden keine Hinweise auf Heteroskedastizität oder deutliche Trends in den Residuen gefunden.

Tabelle 3.6 auf der nächsten Seite zeigt die Schätzungen des Bayes'schen multivariaten linearen Modells für die erwarteten Werte der vier Vergleichsmetriken

Genauigkeit, Relevanz, Sensitivität und Spezifität für einen Klassifikationsansatz unabhängig vom Teildatensatz, der zum Testen verwendet wird (theoretisches „Gesamtmittel"). Die entsprechenden Schätzungen der Klassifikationsleistungen der untersuchten Ansätze sind als Median und Intervall mit 95 % der Dichte der *a-posteriori*-Verteilung in Tabelle 3.7 dargestellt. Die *a-posteriori*-Verteilungen der Werte für die Klassifikationsansätze von Genauigkeit, Relevanz, Sensitivität und Spezifität werden mit den entsprechenden Verteilungen der „Gesamtmittel" in Abbildung 3.6 auf Seite 60, Abbildung 3.7 auf Seite 61, Abbildung 3.8 auf Seite 62 bzw. Abbildung 3.9 auf Seite 63 verdeutlicht.

Tabelle 3.6: Erwartete Klassifikationsleistung unabhängig vom Klassifikationsansatz und vom Testdatensatz als Median und 95 %-Intervall der höchsten Dichte der *a-posteriori*-Verteilungen

	Genauigkeit	Relevanz	Sensitivität	Spezifität
Gesamtmittel	0,705 [0,703; 0,707]	0,706 [0,703; 0,709]	0,654 [0,647; 0,66]	0,751 [0,746; 0,756]

Tabelle 3.7: Erwartete Leistungen der neun untersuchten Klassifikationsansätze als Median und 95 %-Intervall der höchsten Dichte der *a-posteriori*-Verteilungen

Klassifikationsansatz	Genauigkeit	Relevanz	Sensitivität	Spezifität
AE-CNN	0,716 [0,711; 0,722]	0,717 [0,708; 0,726]	0,669 [0,652; 0,687]	0,755 [0,741; 0,77]
AE-GRU	0,708 [0,702; 0,713]	0,706 [0,697; 0,715]	0,66 [0,643; 0,677]	0,75 [0,735; 0,764]
AE-MLP	0,675 [0,67; 0,682]	0,673 [0,663; 0,682]	0,62 [0,601; 0,639]	0,725 [0,71; 0,741]
E2E-CNN	0,717 [0,711; 0,723]	0,715 [0,706; 0,724]	0,676 [0,658; 0,694]	0,751 [0,736; 0,765]
E2E-GRU	0,717 [0,711; 0,723]	0,722 [0,713; 0,731]	0,661 [0,644; 0,678]	0,767 [0,752; 0,781]
E2E-MLP	0,672 [0,667; 0,678]	0,663 [0,654; 0,673]	0,634 [0,616; 0,652]	0,707 [0,692; 0,723]
FE-MLP	0,71 [0,704; 0,715]	0,712 [0,703; 0,721]	0,657 [0,641; 0,674]	0,758 [0,743; 0,772]
FE-RF	0,713 [0,707; 0,719]	0,721 [0,712; 0,73]	0,649 [0,633; 0,666]	0,771 [0,756; 0,785]
FE-SVM	0,718 [0,712; 0,724]	0,727 [0,718; 0,736]	0,655 [0,639; 0,672]	0,775 [0,76; 0,79]

Die theoretischen Mittelwerte der Klassifikationsansätze von Genauigkeit und Relevanz lagen im Median knapp über 0,7. Insgesamt fiel es allen Ansätzen leichter, Beobachtungen von nicht lahmen Kühen richtig zu klassifizieren, als Beobachtungen von lahmen Kühen als solche zu erkennen. Die mittlere Spezifität war im Median 0,1 höher als die mittlere Sensitivität (Tabelle 3.6).

Die höchsten Genauigkeiten wurden für FE-SVM (Median: 0,718) sowie für E2E-CNN und E2E-GRU (beide mit Median 0,717) geschätzt. Für FE-SVM, E2E-GRU und FE-RF wurden die größten Relevanz-Werte durch das Bayes'sche multivariate lineare Modell geschätzt (Median: 0,727; 0,722; 0,721). Die Schätzungen für die Sensitivität waren am höchsten bei E2E-CNN (Median: 0,676), gefolgt von AE-CNN (Median: 0,669) und E2E-GRU (Median: 0,661). Die Spezifität wurde im Median am höchsten geschätzt für FE-SVM (0,775), FE-RF (0,771) und E2E-

GRU (0,767). Bei allen Vergleichsmetriken wurden für AE-MLP und E2E-MLP die niedrigsten Werte geschätzt (Tabelle 3.7 auf der vorherigen Seite).

Tabelle 3.8 zeigt die vorhergesagten Klassifikationsleistungen für die Testdatensätze unabhängig vom Klassifikationsansatz. Für Teildatensatz null wurden die höchste Genauigkeit, Relevanz und Spezifität und die zweithöchste Sensitivität geschätzt. Die nächsthöheren Genauigkeiten wurden für die Klassifikation der Teildatensätze sieben, vier, eins und zwei geschätzt.

Tabelle 3.8: Erwartete Klassifikationsleistungen pro Testdatensatz unabhängig vom Klassifikationsansatz als Median und 95 %-Intervall der höchsten Dichte der *a-posteriori*-Verteilungen

Testdatensatz	Genauigkeit	Relevanz	Sensitivität	Spezifität
Teil 0	0,729 [0,722; 0,735]	0,747 [0,737; 0,757]	0,676 [0,657; 0,694]	0,78 [0,764; 0,795]
Teil 1	0,706 [0,7; 0,712]	0,72 [0,711; 0,729]	0,664 [0,647; 0,682]	0,747 [0,732; 0,762]
Teil 2	0,706 [0,7; 0,712]	0,672 [0,663; 0,681]	0,638 [0,62; 0,656]	0,757 [0,742; 0,772]
Teil 3	0,689 [0,683; 0,696]	0,669 [0,66; 0,679]	0,622 [0,603; 0,641]	0,745 [0,729; 0,76]
Teil 4	0,713 [0,707; 0,719]	0,701 [0,692; 0,71]	0,653 [0,635; 0,67]	0,764 [0,749; 0,779]
Teil 5	0,691 [0,685; 0,697]	0,707 [0,698; 0,716]	0,653 [0,635; 0,67]	0,73 [0,715; 0,745]
Teil 6	0,705 [0,699; 0,711]	0,73 [0,72; 0,739]	0,68 [0,662; 0,699]	0,732 [0,716; 0,747]
Teil 7	0,729 [0,723; 0,735]	0,703 [0,693; 0,712]	0,666 [0,648; 0,684]	0,778 [0,762; 0,793]
Teil 8	0,683 [0,677; 0,689]	0,708 [0,698; 0,718]	0,644 [0,626; 0,662]	0,723 [0,708; 0,738]
Teil 9	0,7 [0,694; 0,706]	0,704 [0,695; 0,713]	0,64 [0,622; 0,657]	0,755 [0,74; 0,77]

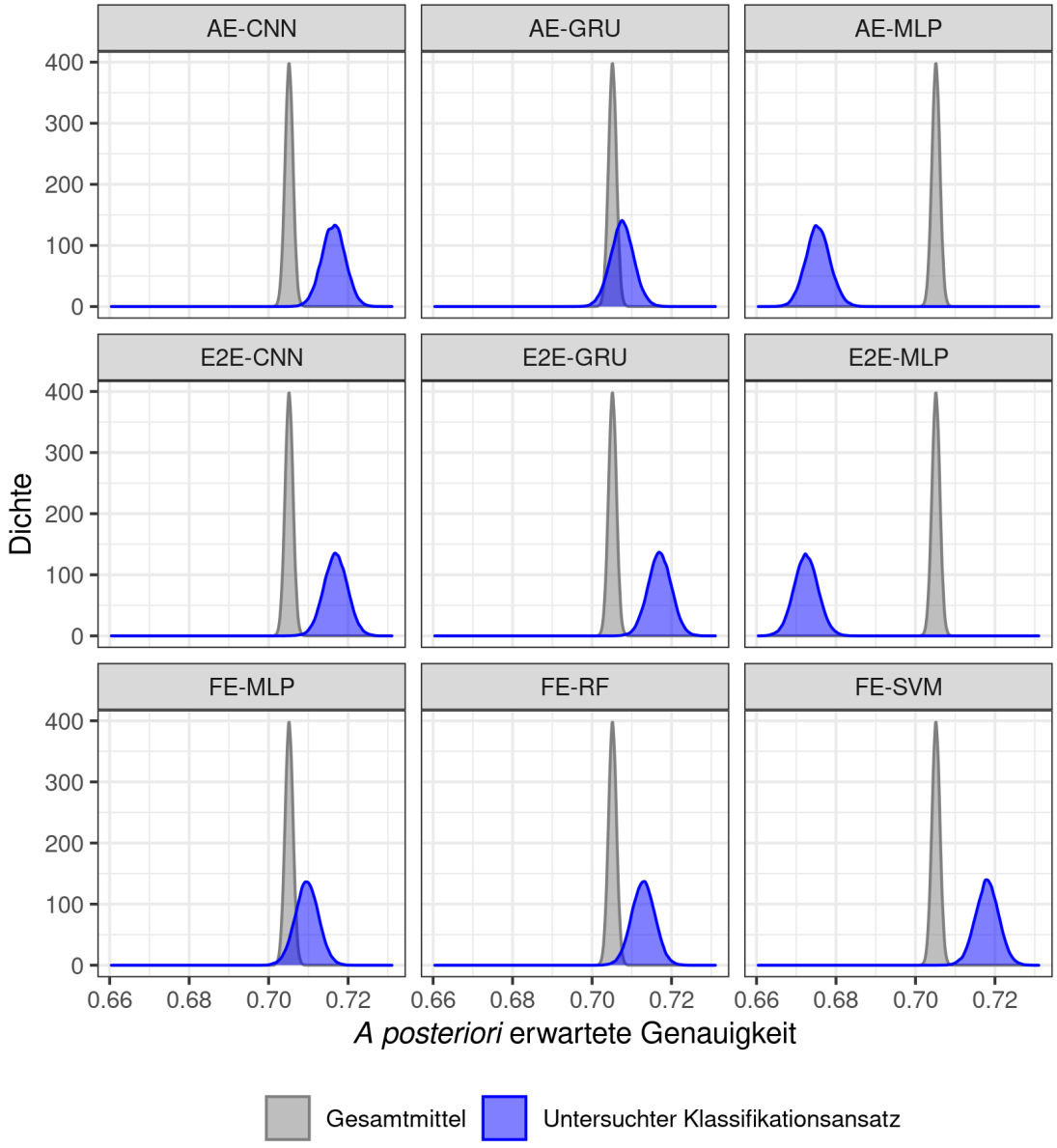

Abbildung 3.6: *A posteriori* erwartete Genauigkeit pro Klassifikationsansatz im Vergleich zum Gesamtmittel

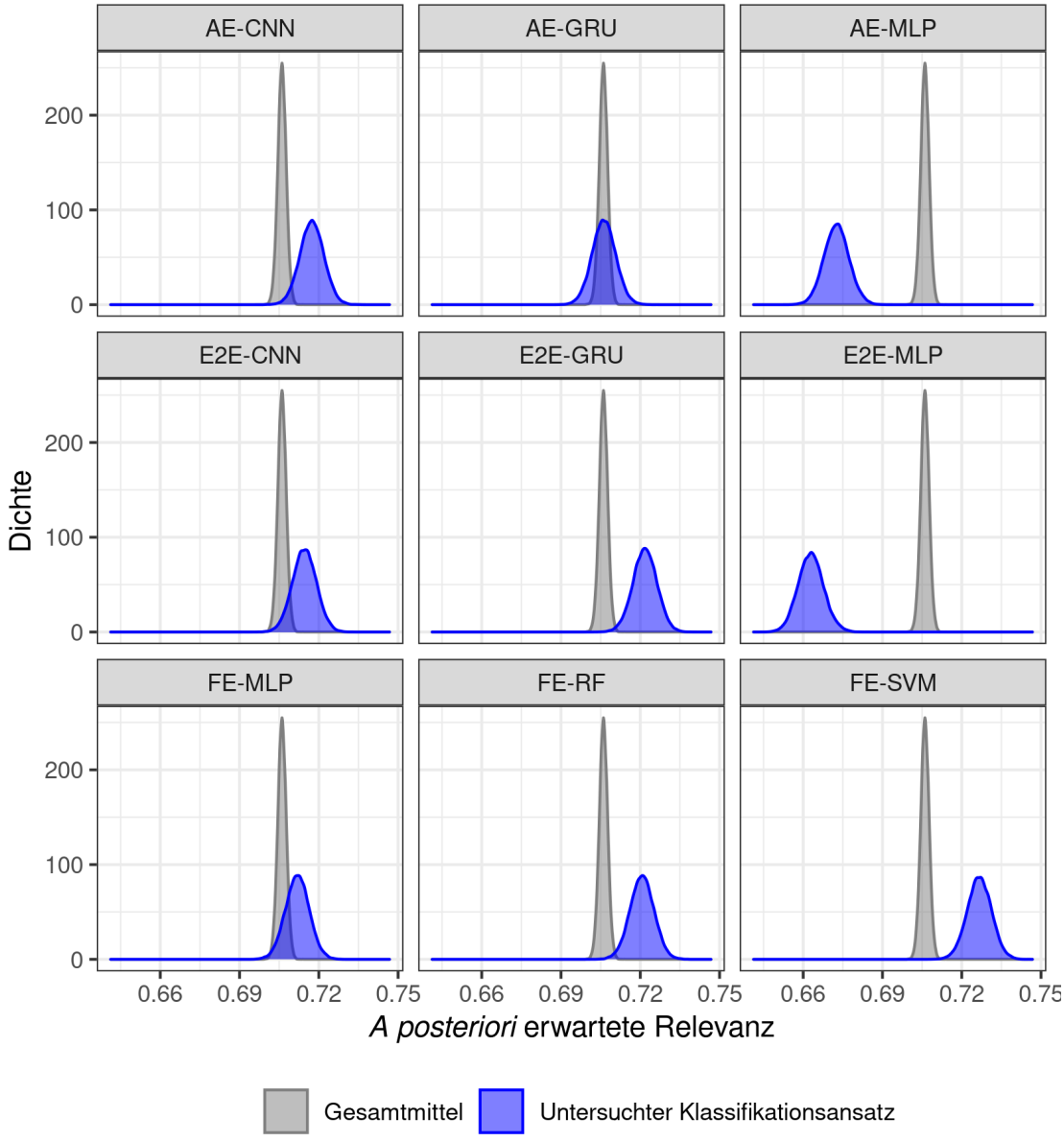

Abbildung 3.7: *A posteriori* erwartete Relevanz pro Klassifikationsansatz im Vergleich zum Gesamtmittel

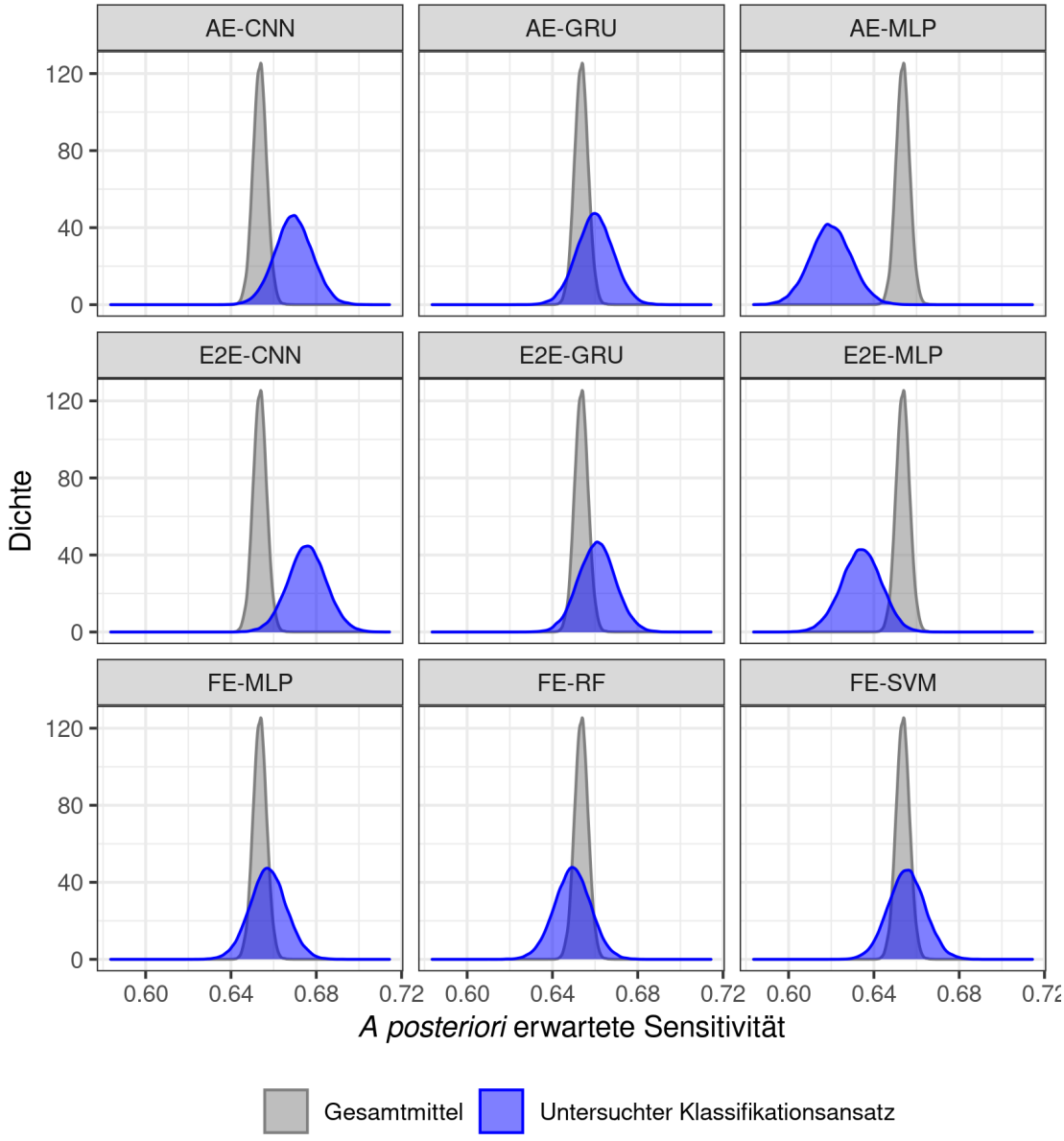

Abbildung 3.8: *A posteriori* erwartete Sensitivität pro Klassifikationsansatz im Vergleich zum Gesamtmittel

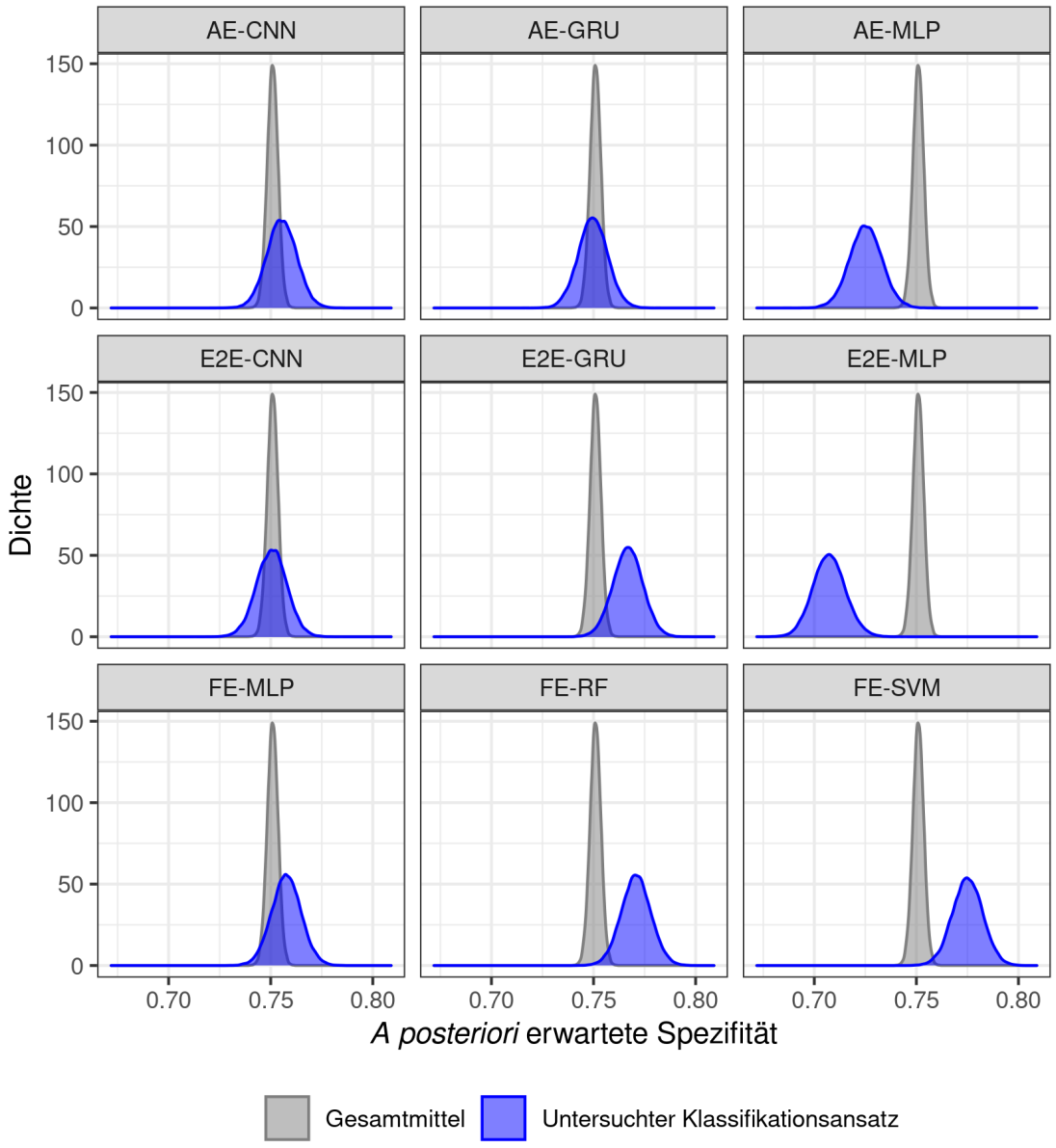

Abbildung 3.9: *A posteriori* erwartete Spezifität pro Klassifikationsansatz im Vergleich zum Gesamtmittel

3.4.1 Allgemeiner Vergleich der Klassifikationsansätze

Die *a-posteriori*-Verteilungen der Differenzen zwischen den geschätzten Werten der Vergleichsmetriken der untersuchten Klassifikationsansätze und dem jeweiligen theoretischen Mittelwert sind zusammen mit der ROPE von $0 \pm 0,01$ in den Abbildungen 3.10 auf der nächsten Seite, 3.11 auf Seite 66, 3.12 auf Seite 67 und 3.13 auf Seite 68 dargestellt. Tabelle 3.9 fasst den Vergleich der Differenzen mit der ROPE zusammen.

Nur die Leistungen von AE-MLP, E2E-MLP und FE-SVM unterschieden sich signifikant von den mittleren Leistungen der untersuchten Ansätze (Tabelle 3.9). AE-MLP schnitt in Bezug auf alle Vergleichsmetriken schlechter ab (Abbildungen 3.10 auf der nächsten Seite, 3.11 auf Seite 66, 3.12 auf Seite 67 und 3.13 auf Seite 68). Die Ergebnisse von E2E-MLP waren ähnlich schlecht. Lediglich die erwartete Sensitivität von E2E-MLP lag näher am theoretischen Mittelwert (Abbildung 3.8 auf Seite 62). Im Gegensatz dazu erwiesen sich die Relevanz und die Spezifität von FE-SVM besser als die theoretischen Mittelwerte (Abbildungen 3.7 auf Seite 61 und 3.11 auf Seite 66 bzw. Abbildungen 3.9 auf der vorherigen Seite und 3.13 auf Seite 68).

Tabelle 3.9: Anteile der Differenzen zwischen den *a-posteriori*-Verteilungen der erwarteten Klassifikationsleistungen der untersuchten Ansätze und der des Gesamtmittels, die innerhalb der Region Of Practical Equivalence (ROPE) von $0 \pm 0,01$ lagen

Klassifikationsansatz	Differenz in der ROPE			
	Genauigkeit	Relevanz	Sensitivität	Spezifität
AE-CNN	33,4 %	37,8 %	23,6 %	77,3 %
AE-GRU	99,6 %	98,2 %	66 %	85,7 %
AE-MLP	0 % *	0 % *	0,6 % *	1,8 % *
E2E-CNN	26,8 %	64 %	7,4 %	85,1 %
E2E-GRU	24,6 %	9 %	62 %	19,2 %
E2E-MLP	0 % *	0 % *	13,5 %	0 % *
FE-MLP	97,7 %	84,3 %	74,5 %	68,4 %
FE-RF	79,2 %	13,9 %	73,5 %	7,2 %
FE-SVM	14,7 %	0,8 % *	77,8 %	2,2 % *

* Signifikanter Unterschied

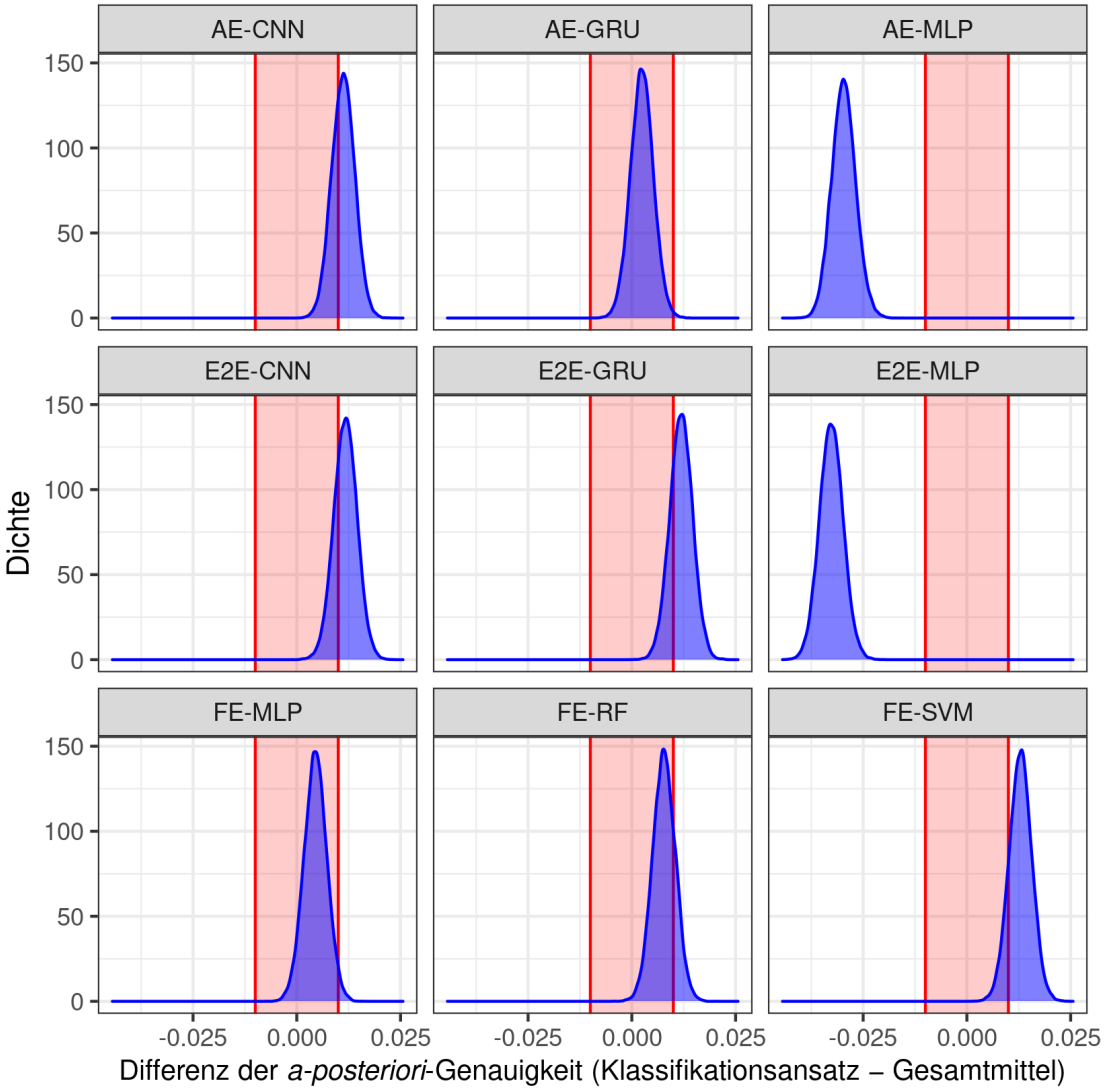

Abbildung 3.10: Differenz der *a posteriori* erwarteten Genauigkeit pro Klassifikationsansatz zum Gesamtmittel (Region Of Practical Equivalence (ROPE) in rot)

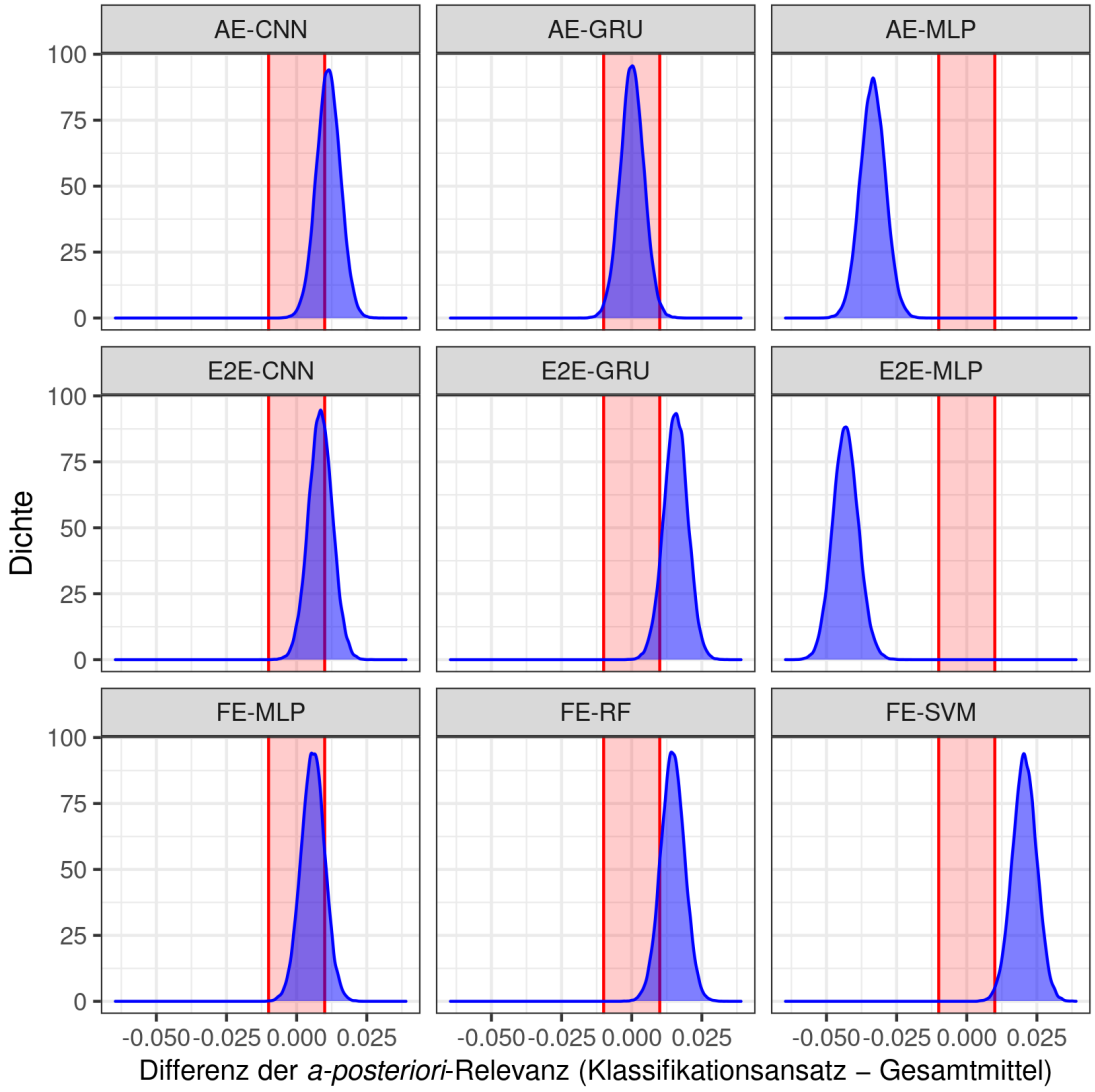

Abbildung 3.11: Differenz der *a posteriori* erwarteten Relevanz pro Klassifikationsansatz zum Gesamtmittel (Region Of Practical Equivalence (ROPE) in rot)

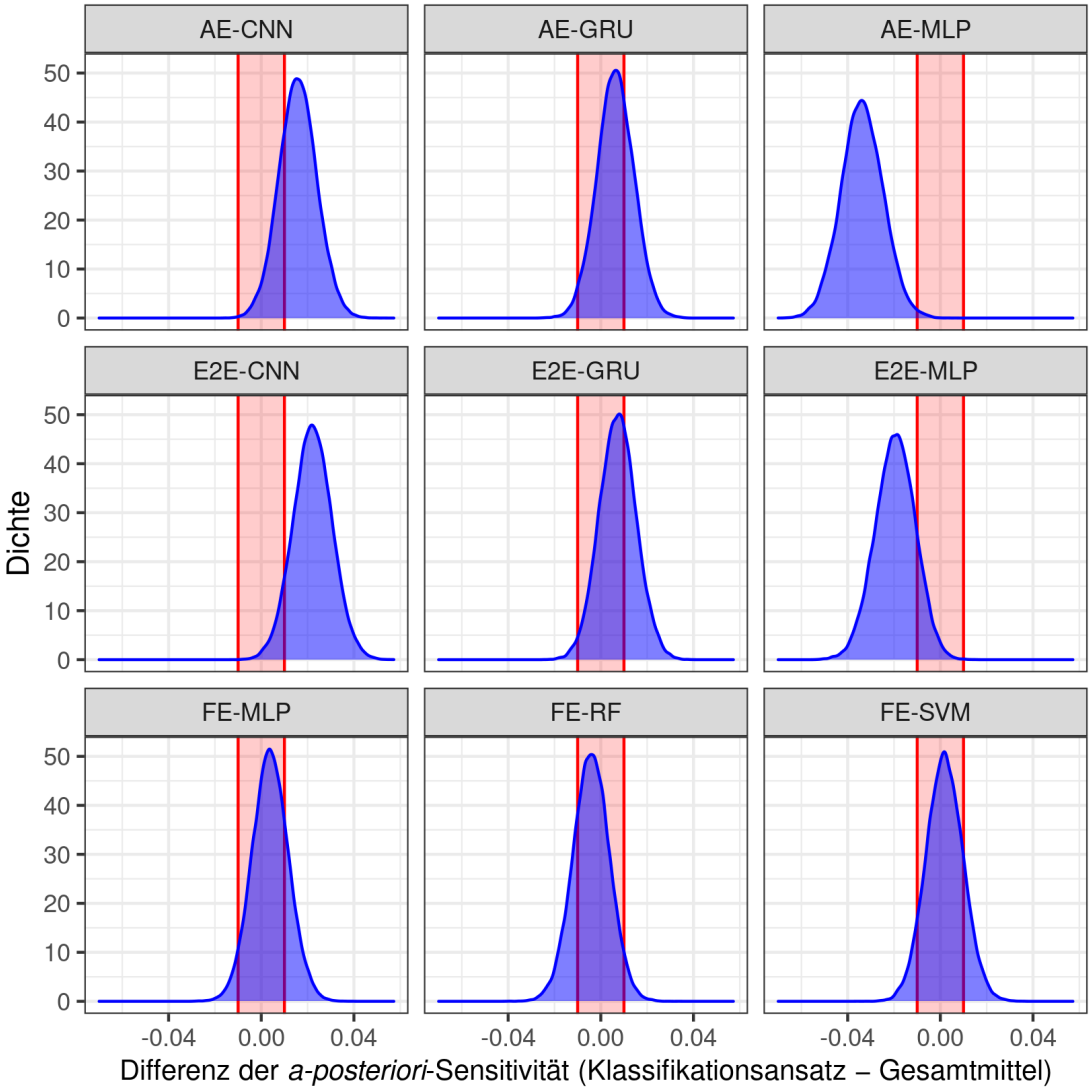

Abbildung 3.12: Differenz der *a posteriori* erwarteten Sensitivität pro Klassifikationsansatz zum Gesamtmittel (Region Of Practical Equivalence (ROPE) in rot)

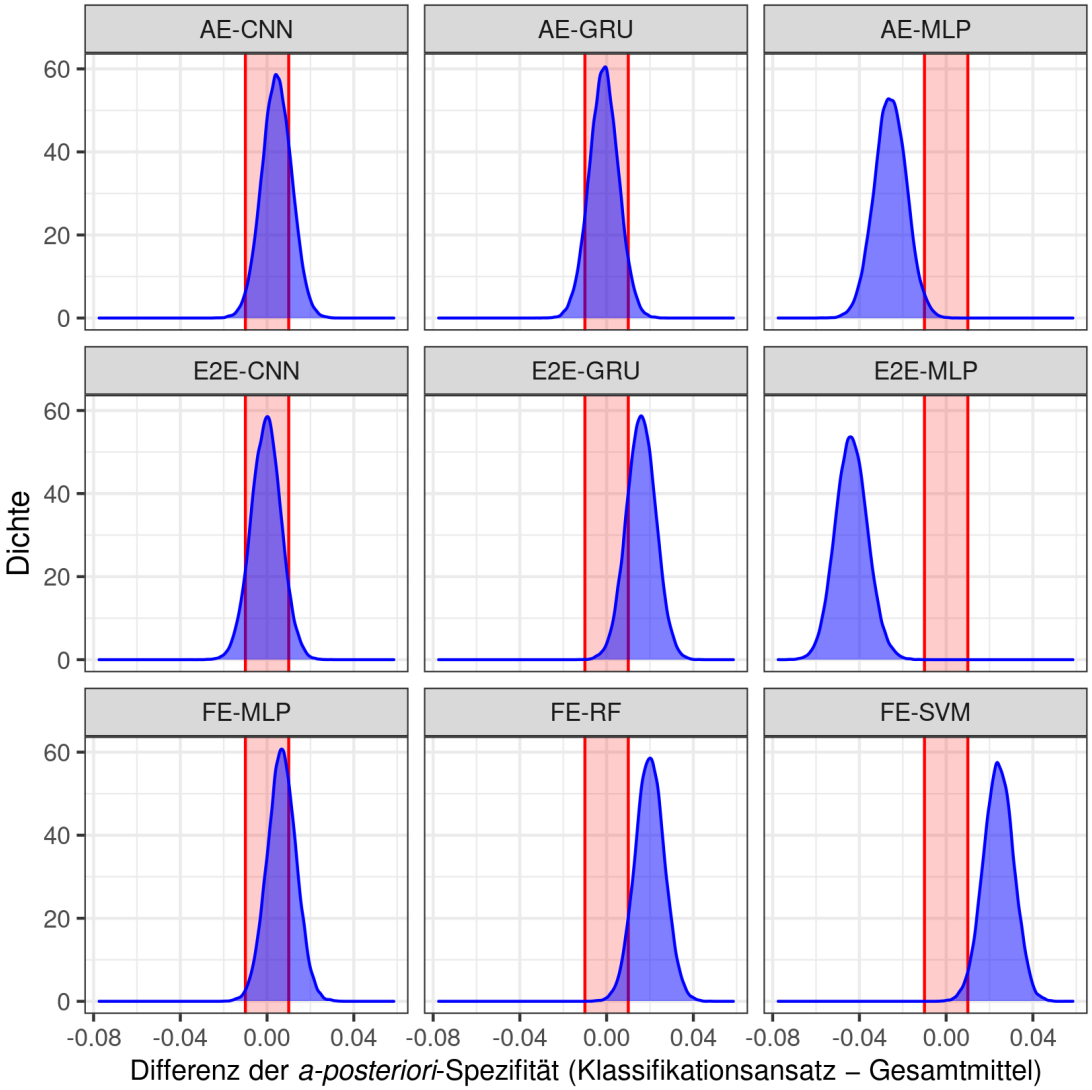

Abbildung 3.13: Differenz der *a posteriori* erwarteten Spezifität pro Klassifikationsansatz zum Gesamtmittel (Region Of Practical Equivalence (ROPE) in rot)

3.4.2 Paarweiser Vergleich der Klassifikationsansätze

Die Ergebnisse der paarweisen Vergleiche der vorhergesagten Klassifikationsleistungen aller Ansätze sind in Tabelle 3.10 auf der nächsten Seite zu finden als Anteile der Differenzverteilungen, die innerhalb der ROPE von $-0,01$ bis $+0,01$ lagen. Die paarweisen Differenzen der *a-posteriori*-Verteilungen der Genauigkeit von AE-CNN, E2E-CNN, E2E-GRU und FE-SVM lagen zu mehr als 97,5 % innerhalb der ROPE. Durch die gerundete Darstellung in Tabelle 3.10 auf der nächsten Seite ist der Anteil der Differenz der Genauigkeiten von AE-CNN und FE-SVM, der innerhalb der ROPE lag, nur mit 97,5 % angegeben. Die Genauigkeit dieser vier Ansätze wurde daher als gleich angesehen.

Die Klassifikationsleistungen von AE-MLP unterschieden sich von denen aller anderer Verfahren außer von E2E-MLP mindestens in Bezug auf drei der vier Vergleichsmetriken. Dasselbe galt für die Leistungen von E2E-MLP. Allerdings konnte entsprechend der verwendeten Regeln keine Entscheidung darüber getroffen werden, ob die Leistungen der beiden Ansätze (AE-MLP und E2E-MLP) gleich waren oder nicht (siehe Seite 46 in Abschnitt 2.3.4).

Tabelle 3.10: Anteile der paarweisen Differenzen zwischen den *a-posteriori*-Verteilungen der erwarteten Klassifikationsleistungen der untersuchten Ansätze, die innerhalb der Region Of Practical Equivalence (ROPE) von $0 \pm 0,01$ lagen

Klassifikationsansätze	Differenz in der ROPE			
	Genauigkeit	Relevanz	Sensitivität	Spezifität
AE-CNN - FE-SVM	97,5 % o	54,4 %	34,4 %	16,8 %
AE-CNN - FE-RF	94,5 %	84 %	19,7 %	29,1 %
AE-CNN - FE-MLP	78,9 %	75,5 %	40,4 %	66,8 %
AE-CNN - E2E-MLP	0 % *	0 % *	2,4 % *	0 % *
AE-CNN - E2E-GRU	98,3 % o	80,2 %	49,7 %	42,7 %
AE-CNN - E2E-CNN	98,5 % o	85 %	53,9 %	62,6 %
AE-CNN - AE-MLP	0 % *	0 % *	0,2 % *	3,2 %
AE-CNN - AE-GRU	62 %	42,5 %	46,7 %	60,4 %
AE-GRU - FE-SVM	45,4 %	5,2 %	56,6 %	6,4 %
AE-GRU - FE-RF	87,4 %	24 %	44,2 %	13,3 %
AE-GRU - FE-MLP	96,9 %	75,3 %	59,5 %	54,5 %
AE-GRU - E2E-MLP	0 % *	0 % *	10,2 %	0,1 % *
AE-GRU - E2E-GRU	55,5 %	19 %	60,4 %	23,2 %
AE-GRU - E2E-CNN	57 %	60,4 %	30 %	67,9 %
AE-GRU - AE-MLP	0 % *	0 % *	1,1 % *	8,6 %
AE-MLP - FE-SVM	0 % *	0 % *	2,3 % *	0 % *
AE-MLP - FE-RF	0 % *	0 % *	6 %	0,1 % *
AE-MLP - FE-MLP	0 % *	0 % *	1,6 % *	1,7 % *
AE-MLP - E2E-MLP	95,3 %	51,8 %	34,1 %	21,5 %
AE-MLP - E2E-GRU	0 % *	0 % *	0,8 % *	0,2 % *
AE-MLP - E2E-CNN	0 % *	0 % *	0,1 % *	8,2 %
E2E-CNN - FE-SVM	97,8 % o	37,1 %	19,2 %	7,8 %
E2E-CNN - FE-RF	92,7 %	72,5 %	8,9 %	15,5 %
E2E-CNN - FE-MLP	75,1 %	85,4 %	23,7 %	56,5 %
E2E-CNN - E2E-MLP	0 % *	0 % *	0,9 % *	0,2 % *
E2E-CNN - E2E-GRU	98,3 % o	66,7 %	32,3 %	26,1 %
E2E-GRU - FE-SVM	98,1 % o	78,2 %	55,8 %	53,2 %
E2E-GRU - FE-RF	92,5 %	88,2 %	41,7 %	64,5 %
E2E-GRU - FE-MLP	73,7 %	50,5 %	58,1 %	50,5 %
E2E-GRU - E2E-MLP	0 % *	0 % *	8,9 %	0 % *
E2E-MLP - FE-SVM	0 % *	0 % *	18,3 %	0 % *
E2E-MLP - FE-RF	0 % *	0 % *	31,9 %	0 % *
E2E-MLP - FE-MLP	0 % *	0 % *	13,9 %	0 % *
FE-MLP - FE-SVM	65,2 %	22,4 %	59,9 %	22,8 %
FE-MLP - FE-RF	94,7 %	57,5 %	50,9 %	36,4 %
FE-RF - FE-SVM	88,3 %	73 %	55,2 %	64 %

* Signifikanter Unterschied

o Kein Unterschied

4

Diskussion

Lahmheit ist eine der bedeutendsten Gesundheitsbeeinträchtigungen bei Milchkühen. Da ihre Erkennung sehr zeitaufwändig ist, werden Systeme zur automatischen Lahmheitserkennung erforscht. Der Algorithmus, mit dem die gemessenen Daten klassifiziert und so letztendlich lahme Kühe erkannt werden, ist ein essentieller Bestandteil dieser Systeme. In der vorliegenden Arbeit habe ich verschiedene Klassifikationsansätze zur Erkennung von Lahmheit an Hand von indirekten Variablen zur Aktivität, zur Leistung und zu Eigenschaften der Kühe verglichen. Die Leistungsfähigkeit der Ansätze bewertete ich mit den grenzwertabhängigen Metriken Genauigkeit, Relevanz, Sensitivität und Spezifität. Zusätzlich erfasste ich die für Training und Vorhersage benötigten Zeiten und den benötigten Arbeitsspeicher.

Neun Ansätze hatte ich in den Vergleich aufgenommen: drei Ansätze mit Feature Engineering (FE-SVM, FE-RF, FE-MLP), drei Ende-zu-Ende-Ansätze (E2E-MLP, E2E-CNN und E2E-GRU) sowie drei generative Ansätze mit unüberwachtem Vortraining (AE-MLP, AE-CNN, AE-GRU). Die neun Ansätze wurden in zehnfacher Kreuzvalidierung mit jeweils denselben Teildatensätzen getestet. Nach meinem Kenntnisstand wurde bisher keine Studie publiziert, in der die Klassifikationsleistungen von tiefen neuronalen Netzen wie CNNs und RNNs zur Erkennung von Lahmheit bei Milchkühen aus Aktivitätsdaten untersucht worden war. Gleiches gilt für generative Klassifikationsansätze. Damit ist meine Arbeit die erste Studie, die die Leistungsfähigkeit dieser Ansätze und Modelltypen, Lahmheiten von Milchkühen auf der Grundlage von Aktivitätsdaten zu erkennen, systematisch untersucht. Allerdings wurden in der vorliegenden Arbeit nur allgemeine Modelltypen des maschinellen Lernens untersucht, wenn auch mit leichten Anpassungen für multivariate Zeitreihen und andere Besonderheiten der verwendeten Daten. Spezifische Modelle für die Klassifizierung (multivariater) Zeitreihen habe ich ausgelassen, weil selbst die allgemeinen Modelltypen bisher nur selten bzw. noch nie mit vergleichbaren Daten zur Erkennung von Lahmheiten bei Milchkühen eingesetzt wurden. Einen Überblick und eine Einordnung der Leistungsfähigkeit spezifischer Ansätze wie z. B. Dynamic Time Warping, Hierarchical Vote Collective of Transformation-based Ensembles und Random Convolutional Kernel Transform wurde in der Vergleichsstudie von Ruiz u. a. [Rui+21] gegeben. Im Bereich der tiefen neuronalen Netze gibt es ebenfalls spezielle Architekturen zur Zeitreihenklassifikation. Zu diesen Architekturen gehören ResNet [WYO17] und InceptionTime

[Faw+20] sowie hybride Modelle aus CNN und RNN. In den hybriden Modellen werden mittels CNN Muster in der Zeitreihen erkannt und anschließend in darauf folgenden rekurrenten Schichten in den zeitlichen Kontext eingeordnet [Hir21, S. 208] oder die Daten werden CNN und RNN parallel übergeben und die jeweiligen Ergebnisse werden für die Klassifikation konkateniert [vgl. Kha+20]. Stark erweiterte hybride Ansätze wie beispielsweise TapNet enthalten nicht nur verschiedene Typen neuronaler Netze, sondern auch Modelle des klassischen maschinellen Lernens [Rui+21]. Eine Bewertung der Eignung all dieser speziellen Ansätze für die automatische Lahmheitserkennung bei Milchkühen aus Aktivitätsdaten hätte den Rahmen meiner Arbeit gesprengt. Die dafür notwendigen Untersuchungen bleiben damit weiteren Studien überlassen.

Für diese Arbeit standen mir bereits erhobene Daten aus dem Projekt Klauen*fit*net[1] zur Verfügung mit den indirekten Variablen *Laktationsnummer*, *Laktationstag*, *Tagesgemelk*, *durchschnittliche Schrittfrequenz* und *durchschnittliche Liegedauer pro Liegevorgang* sowie den Labels „lahm“ oder „nicht lahm“ aus der Gangbeurteilung der Kühe. Die Werte der Variablen waren im Beobachtungszeitraum täglich erhoben worden. Der Gang der Kühe war ungefähr alle 14 Tage beurteilt worden. Eine Einschränkung haben alle Klassifikationsmodelle zur automatischen Lahmheitserkennung, die mit Labels trainiert werden, die nicht wenigstens täglich erhoben wurden: Wenn die Labels nicht täglich erhoben wurden, d. h. der Gang der Kühe seltener als täglich beurteilt wurde, dann kann der Zeitpunkt, an dem eine Kuh lahm wurde, nur ungenau bestimmt werden. In den meisten Fällen dürfte er zwischen zwei Gangbeurteilungen liegen. Da in diesem Fall unbekannt ist, wie lange eine Kuh schon lahm ist, die als lahm erkannt wird, fehlen diese Informationen in den Daten und können nicht bei der Klassifizierung berücksichtigt werden. Die Modelle können nicht lernen, speziell frisch lahme (bzw. lahm werdende) Kühe zu erkennen, sie können lediglich lernen, Kühe zu erkennen, die bereits lahm sind. Diese Einschränkung könnte die Sensitivität der Klassifikationsansätze einschränken, weil möglicherweise die Muster in den Zeitfenstern der Merkmale von bereits länger lahmen Kühen, die sich an diese Beeinträchtigung gewöhnt haben, eher denen von nicht-lahmen Kühen ähneln als jenen von frisch lahmen Kühen. In den Studien, die mit vergleichbaren Daten arbeiteten, wie sie in der vorliegenden Arbeit verwendet wurden (Tabelle 1.1 auf Seite 8), erfolgte die Gangbeurteilung ebenfalls seltener als täglich. Lediglich für die beiden Studien von van Hertem u. a. [vHer+13] und Kamphuis u. a. [Kam+13] wurde täglich nach lahmen Kühen gesucht. Es wurde in diesen Fällen aber nicht das Gangbild von allen Kühen täglich beurteilt, sondern die Kühe wurden als „nicht-lahm“ betrachtet, solange sie nicht als „lahm“ auffielen. Auf diese Weise wurden aber leicht Kühe übersehen, die nur gering- oder mittelgradig lahm waren, was wiederum die Klassifikation durch die trainierten Modelle beeinflusst haben könnte [vHer+13].

In der Literatur zu den Themenbereichen dieser Arbeit (automatische Lahmheitserkennung und Zeitreihenklassifikation) konnte ich keine vergleichbaren Studien finden, die der Empfehlung von Benavoli u. a. [Ben+17] folgten und die Leistun-

[1] `https://www.klauenfitnet.de/`, besucht am 24. Mai 2022

gen der untersuchten Ansätze mit einem Bayes'schen Modell untersuchten. Obwohl in einigen Studien wie in meiner Arbeit der Vergleich der Klassifikationsansätze über mehrere Metriken erfolgte [Las+21; Rui+21; Sha+21], wurde in keiner dieser Arbeiten ein multivariates Modell zu statistischen Analyse verwendet. Damit ist die vorliegende Arbeit meines Wissens die erste Vergleichsstudie, bei der mit einem Bayes'schen multivariaten linearen Modell die Klassifikationsleistungen der untersuchten Ansätze ausgewertet wurden.

4.1 Klassifikationsleistung

Die Leistungen aller neun Klassifikationsansätze waren sehr ähnlich, lediglich die Ansätze mit MLP ohne Feature Engineering (E2E-MLP und AE-MLP) hatten deutlich schlechtere Klassifikationsergebnisse. Am besten klassifizierten FE-SVM, E2E-GRU, E2E-CNN und AE-CNN (Tabelle 3.7 auf Seite 58). Die Ansätze mit CNN (E2E-CNN und AE-CNN) hatten die höchste Sensitivität, erkannten also am besten lahme Kühe als lahm (Abbildung 3.8 auf Seite 62). FE-SVM und E2E-GRU klassifizierten hingegen nicht lahme Kühe häufiger richtigerweise als nicht lahm, ihre Spezifität war höher (Abbildung 3.9 auf Seite 63).

Die beiden Ansätze mit MLP ohne Feature Engineering (E2E-MLP und AE-MLP) klassifizierten deutlich schlechter als die anderen Ansätze (Tabelle 3.7 auf Seite 58). Da die erwartete Klassifikationsleistung von FE-MLP allerdings nicht schlechter war als das Mittel aller Ansätze, liegt die Vermutung nahe, dass die Features, die bei E2E-MLP und AE-MLP für die Klassifikation erlernt wurden, nicht so aussagekräftig waren wie die zusätzlich manuell bereitgestellten bei FE-MLP. Abgesehen von AE-MLP und E2E-MLP, gab es keine merklichen Unterschiede zwischen den Ansätzen mit Feature Engineering und den Ende-zu-Ende-Ansätzen sowie den generativen Ansätzen. Insgesamt schienen die „handgemachten" Features ähnlich aussagekräftig zu sein wie die erlernten, allerdings erreichten die Ansätze mit erlernten Features eine etwas höhere Sensitivität (Abbildung 3.8 auf Seite 62). Eine mögliche Begründung für das relativ schlechte Abschneiden von E2E-MLP und AE-MLP ist daher, dass es sich in beiden Fällen nur um kleine neuronale Netze handelte (E2E-MLP mit 14, AE-MLP mit 111 Neuronen in allen verborgenen Schichten zusammen), die möglicherweise nicht ausreichend differenzierende Merkmale zur Klassifikation erlernen konnten. Zum Vergleich: Das MLP, welches von Wang, Yan und Oates [WYO17] zur Klassifikation von Zeitreihen verwendet wurde, hatte insgesamt 1500 Neuronen in drei verborgenen Schichten. Gegen diese Erklärung sprechen allerdings die geringen Unterschiede zwischen den Klassifikationsleistungen von E2E-MLP und denen von AE-MLP trotz des deutlich größeren neuronalen Netzes in AE-MLP (Tabelle 3.10 auf Seite 70). Darüber hinaus konnten E2E-MLP und AE-MLP die zeitliche Information aus den Zeitreihen nicht verwenden im Gegensatz zu den entsprechenden Ansätzen mit CNN und GRU, weil für die vollständig verbundenen Schichten pro Beobachtung ein Vektor aller Features übergeben werden musste. Die zeitliche Information, welche in den als Matrizen übergebenen Beobachtungen in der Reihenfolge der Werte lag, ging durch die Um-

wandlung in Vektoren verloren. Aus diesem Grund bieten sich CNN und RNN für die Klassifikation von Zeitreihen besonders an [Faw+19].

Die Länge der Zeitfenster von 27 Zeitschritten (Tagen), die in der vorliegenden Arbeit verwendet wurde, war eher kurz. Das könnte ein Grund dafür sein, dass sich bei der Klassifizierung der Zeitreihen kaum Leistungsunterschiede zeigten zwischen den Ansätzen mit CNN und denjenigen mit GRU. Recurrent Neural Network, zu denen GRU gehören, wurden speziell für die Analyse von Zeitreihendaten entwickelt. Sie können sehr gut den Gesamtverlauf der Werte eines Merkmals im Zeitfenster bzw. den zeitlichen Kontext eines Merkmalswerts berücksichtigen [Hir21, S. 208]. Im Gegensatz dazu lernen CNN, das Vorkommen von Mustern in sehr kurzen Zeitabschnitten zu erkennen. Dabei berücksichtigen sie den zeitlichen Kontext bzw. die zeitliche Lage des Musters innerhalb des Zeitfensters nicht. Die vorherige Entwicklung des Merkmalswerts und dessen Verlauf über das gesamte Zeitfenster werden von CNN ignoriert [Hir21, S. 208]. Bei der Kürze der Zeitfenster, die in der vorliegenden Arbeit verwendet wurden, scheinen diese Unterschiede kaum ins Gewicht gefallen zu sein.

Bei allen untersuchten Ansätzen war die Sensitivität 0,07 bis 0,12 geringer als die Spezifität (Tabelle 3.7 auf Seite 58). Da die Hyperparameteroptimierung an Hand der Genauigkeit erfolgte, ist nicht auszuschließen, dass die leichte Imbalance der Daten in bezug auf die Labelklassen eine Optimierung zu Gunsten der Spezifität verursacht hat. Falls in einem Datensatz eine Klasse deutlich überwiegt, führt die bevorzugte Zuordung der Beobachtungen zu dieser Klasse durch ein Klassifikationsmodell zu einer höheren Genauigkeit. Betrachtet man die Ergebnisse bezogen auf die Testdatensätze in den Kreuzvalidierungsdurchgängen, kann man Hinweise darauf finden, dass in der vorliegenden Studie die Verteilung der Labelklassen in den Teildatensätzen möglicherweise geringen Einfluss auf die Klassifikationsleistungen gehabt haben könnte. Wurde ein Teildatensatz mit etwas weniger Beobachtungen von Lahmheit (Teildatensätze zwei, drei, vier und sieben in Tabelle 3.2 auf Seite 50) nicht zum Training verwendet, führte das nicht zu auffälliger Genauigkeit, Sensitivität oder Spezifität, lediglich die Relevanz war niedriger als beim Auslassen eines anderen Teildatensatzes (Tabelle 3.8 auf Seite 59). In den Kreuzvalidierungsdurchgängen, in denen Teildatensätze mit häufigeren Beobachtungen von Lahmheit (Teildatensätze fünf, sechs und acht in Tabelle 3.2 auf Seite 50) als Testdaten verwendet wurden, wurden die geringsten Spezifitäten erzielt (Tabelle 3.8 auf Seite 59). Die Unterschiede zu den Klassifikationsleistungen in den übrigen Kreuzvalidierungsdurchgängen waren aber nur gering. Insgesamt waren die in der vorliegenden Studie verwendeten Daten nahezu balanciert (48 % der gelabelten Beobachtungen hatten das Label „lahm“), sodass ich davon ausgehe, dass dieser Effekt bei meiner Studie nur einen untergeordneten Einfluss auf die Gesamtergebnisse hatte. Das Verhältnis von Sensitivität zu Spezifität lässt sich einfach nachträglich an die Erfordernisse anpassen, indem die Schwelle für die Entscheidung zwischen den Klassen angepasst wird. Tatsächlich empfahlen O'Leary u. a. [OLe+20] für Systeme zur automatischen Lahmheitserkennung bei Milchkühen eine höhere Spezifität als Sensitivität, weil sie einen geringen Anteil fälschlicherweise als lahm klassifizierter Kühe für wichtiger halten als einen sehr hohen Anteil lah-

mer Kühe, die als solche erkannt werden. Sie argumentieren, dass eine geringe Spezifität durch viele falsche Benachrichtigungen schnell zur Frustration bei den Anwenderinnen und Anwendern führen würde.

Die Visualisierung der Daten nach Dimensionsreduktion mit t-distributed Stochastic Neighbor Embedding ließ keine gute Separierbarkeit der Labelklassen vermuten (Abbildung 3.1 auf Seite 49). Insgesamt waren die Klassifikationsleistungen in der vorliegenden Arbeit daher erwartungsgemäß nur mäßig gut mit Genauigkeiten und Relevanzen um 0,71 sowie Sensitivität und Spezifität von etwa 0,65 bzw. 0,75 (Tabellen 3.6 auf Seite 58 und 3.7 auf Seite 58). Damit erwiesen sich die Ergebnisse der untersuchten Klassifikationsansätze bei den vorliegenden Daten als deutlich schlechter als die von O'Leary u. a. [OLe+20] für praxistaugliche Systeme empfohlenen Untergrenzen bei einer Sensitivität von 0,90 und einer Spezifität von 0,99. Die erzielten Klassifikationsleistungen waren in vorliegenden Studie schlechter als in den meisten Studien in Tabelle 1.1 auf Seite 8, die auch indirekte Variablen aus Aktivitätsdaten verwendeten. Nur zwei Publikationen konnte ich finden, in denen vergleichbare Merkmale (indirekte Variablen aus Aktivitätsdaten) zur automatischen Lahmheitserkennung verwendet wurden, und die mit ähnlichen Klassifikationsmodellen arbeiteten wie in meiner Arbeit. Es handelte sich um die Artikel von Alsaaod u. a. [Als+12] und Borghart, O'Grady und Somers [BOS21] (Tabelle 1.1 auf Seite 8). Im ersten Artikel wurde eine Studie beschrieben, in der ausschließlich Aktivitätsdaten von jeweils einem Tag zur Klassifikation herangezogen wurden. Jedoch wurden mittels Feature Engineering Merkmale erzeugt, welche die kuhindividuelle Abweichung der aktuellen Merkmalswerte vom jeweiligen Referenzwert ohne Lahmheit beschrieben. Zur Klassifikation wurde eine SVM mit Kernel mit gaußscher radialer Basisfunktion verwendet wie in meiner Studie. Diese erreichte in zehnfacher Kreuzvalidierung eine Genauigkeit von 0,76 und eine Relevanz von 0,77 [Als+12]. In der vorliegenden Arbeit erreichte FE-SVM eine Genauigkeit von 0,72 und eine Relevanz von 0,73 (Tabelle 3.7 auf Seite 58). Die zweite vergleichbare Untersuchung wurde im Artikel von Borghart, O'Grady und Somers [BOS21] beschrieben. Darin wurden ebenfalls keine Zeitreihen analysiert sondern nur Daten einzelner Tage. Neben Variablen, welche das Liegen und Gehen beschreiben, wurden auch solche einbezogen, die sich auf das Fressen und Wiederkauen bezogen. Die Klassifikation erfolgte in der Studie mittels Gradient Boosted Decision Trees. Gradient Boosted Decision Trees sind wie Random Forests ebenfalls Ensembles aus Entscheidungsbäumen. Allerdings werden bei Gradient Boosted Decision Trees die einzelnen Bäume nacheinander (additiv) mit dem Restfehler des vorangegangenen Baums trainiert [Fri01], während bei Random Forests die Bäume unabhängig voneinander trainiert werden [Bre01]. Das Modell von Borghart, O'Grady und Somers [BOS21] erreichte bei der einfachen Klassifikation unbekannter Testdaten eine Genauigkeit, Sensitivität und Spezifität von 0,78. Der vergleichbare Klassifikationsansatz FE-RF erreichte in der vorliegenden Arbeit eine Genauigkeit von 0,71, eine Sensitivität von 0,65 und eine Spezifität von 0,77 (Tabelle 3.7 auf Seite 58). Die in Tabelle 1.1 auf Seite 8 aufgeführten Studien hatten allerdings andere Ziele als die vorliegende Arbeit. Bei diesen Studien sollte jeweils ein einzelnes möglichst gutes Klassifikationsmodell erstellt werden. Daher wurden

diese Modelle vermutlich mit wesentlich größerem Aufwand ausgewählt und optimiert als die Modelle in meiner Studie, deren Ziel der Vergleich verschiedener Klassifikationsansätze war.

Die Klassifikationsleistungen der untersuchten Ansätze könnten wahrscheinlich verbessert werden, indem zusätzliche Features verwendet werden. Variablen, die die Jahreszeit und die einzelne Kuh beschreiben, sind erfolgversprechende Optionen. Die Jahreszeit (bzw. die Witterung) beeinflusst möglicherweise das Auftreten von Lahmheit und war in der Studie von Grimm u. a. [Gri+19] im endgültigen Modell enthalten. In den Klassifikationsmodellen der vorliegenden Arbeit konnte die Jahreszeit nicht verwendet werden, weil die Gangbeurteilung nicht in allen Betrieben in der selben Jahreszeit erfolgt war und weder Beobachtungen für ein volles Jahr noch Wetterdaten vorlagen. Daher konnte ein möglicher Zusammenhang zwischen der Jahreszeit und dem Ergebnis der Gangbeurteilung nicht vom Zusammenhang zwischen dem Betrieb und dem Ergebnis der Gangbeurteilung unterschieden werden. Die Ausprägung des Verhaltens (bzw. der Aktivität) ist abhängig von der individuellen Kuh [dMol+13], d. h. die Variabilität der Aktivitätsmerkmale kann größer sein zwischen den einzelnen Kühen als zwischen lahmen und nicht-lahmen Kühen [Als+12]. Eine Identifikation der einzelnen Kuh als Feature in Klassifikationsmodellen zur automatischen Lahmheitserkennung zu verwenden, wäre *in praxi* aufwändig umzusetzen, weil es notwendig wäre, dass Modell mit gelabelten Daten jeder Kuhherde, in der das System angewandt werden soll, separat zu trainieren [vgl. Als+12; Tan+20]). Dieser Aufwand würde die Akzeptanz der Anwenderinnen und Anwender für das System möglicherweise reduzieren und es ist fraglich, ob in allen Herden genug Trainingsdaten erhoben werden könnten.

4.2 Ressourcenbedarf

Der Arbeitsspeicherbedarf für das Training der Modelle lag bei allen neun Ansätzen in der vorliegende Arbeit unter 3 GB (Tabelle 3.5 auf Seite 57). Die Klassifikationszeiten lagen außer bei FE-RF und FE-SVM deutlich unter einer Sekunde, bei diesen beiden bei wenigen (drei bis fünf) Sekunden. Bei allen diskriminativen Klassifikationsansätzen lag die Trainingsdauer unter fünf Minuten; alle generativen Ansätze mit unüberwachtem Vortraining benötigten deutlich mehr Zeit zum Erlernen der Modellparameter (Tabelle 3.5 auf Seite 57). Eine längere Trainingsdauer stellt für den praktischen Einsatz kein allzu großes Problem dar, da das Training nur selten erfolgen muss und auf zentralen Hochleistungsrechnern durchgeführt werden könnte. Daher wären allen Ansätze nach ihrem Ressourcenbedarf *in praxi* einsetzbar. Die kleineren Ansätze, welche keine GPU benötigen (z. B. FE-SVM) könnten auch auf einfachen Personal Computern mit einer Konfiguration, wie sie von Byabazaire u. a. [Bya+19] beschrieben wurde (mit Dual Core CPU mit 2,3 GHz und 4,0 GB Arbeitsspeicher), trainiert werden.

Der gemessene Ressourcenbedarf der unterschiedlichen Klassifikationsansätze ist nicht gänzlich vergleichbar, weil nicht alle Ansätze auf derselben Hardware trainiert und getestet wurden. Vielmehr wurde die verwendete Hardware den un-

terschiedlichen Möglichkeiten der jeweils eingesetzten Softwarepakete angepasst, um die Trainingszeiten möglichst zu reduzieren. Für alle neuronalen Netze wurde TensorFlow verwendet, das es ermöglicht, wesentliche Teile der Berechnungen für das Training und die Vorhersage auf eine GPU auszulagern. Daher wurden alle Ansätze mit neuronalen Netzen unter Verwendung einer GPU evaluiert. Ohne Einsatz der GPU wären die Trainingszeiten dieser Ansätze nochmals deutlich länger. Die relativ kurzen Klassifikationszeiten dieser Ansätze (Abbildung 3.4 auf Seite 55) im Vergleich zu den Ansätzen ohne neuronale Netze (FE-SVM und FE-RF) können vermutlich zumindest teilweise durch diese Unterschiede der verwendeten Hardware erklärt werden. Das Training von Random Forests lässt sich in der Implementierung von scikit-learn sehr einfach auf mehreren CPU-Kernen parallelisieren. Für das Training und die Klassifikation wurden daher für FE-RF fünf CPU-Kerne bereitgestellt. Bei allen anderen Ansätzen wurden lediglich vier Kerne verwendet, welche allerdings in erster Linie für den parallelisierten Datenimport benötigt wurden. Ohne Parallelisierung wäre die Trainingsdauer von FE-RF vermutlich etwa fünfmal so lang wie sie aktuell gemessen wurde. Damit wäre sie länger als die Trainingsdauern der Ende-zu-Ende-Ansätze, bei denen eine GPU genutzt wurde.

Die Evaluierung der Modelle erfolgte nicht immer unter alleiniger Nutzung der jeweiligen Knoten des High Performance Computing Systems (siehe S. 32 im Abschnitt 2.3.2). Es kann nicht ausgeschlossen werden, dass die entsprechenden CPUs gleichzeitig auch von anderen Jobs verwendet wurden, was die Bestimmung von Trainings- und Vorhersagezeiten beeinflusst haben könnte.

Auf Grund der genannten Einschränkungen sind die Ergebnisse meiner Arbeit zum Ressourcenbedarf der untersuchten Klassifikationsansätze nur als Hinweise zu betrachten. Da sie wegen der spezifischen Hardwarekonfigurationen schlecht verallgemeinerbar sind, wurde keine entsprechende schließende Statistik angefertigt. In keiner der Publikationen über Studien mit vergleichbaren Fragestellungen (siehe Abschnitt 1.2.1 auf Seite 11) wird der Ressourcenbedarf der verwendeten Klassifikationsmodelle erwähnt. Lediglich Ruiz u. a. [Rui+21] gaben in ihrem Artikel über einen allgemeinen Vergleich von Modelltypen zur Klassifikation von multivariaten Zeitreihen den jeweils benötigten Arbeitsspeicher und die Laufzeiten der Ansätze an. Sie verzichteten ebenso wie ich auf die Generalisierung dieser Angaben durch eine schließende Statistik. Daher sind die Ergebnisse meiner Arbeit zum Ressourcenbedarf der untersuchten Klassifikationsansätze trotz der genannten Einschränkungen hilfreich für weitere Studien und die Weiterentwicklung von Systemen zur automatischen Lahmheitserkennung bei Milchkühen.

4.3 Gegenüberstellung der Klassifikationsleistung und des Ressourcenbedarfs

Die benötigten Trainingszeiten sind in Abbildung 4.1 auf der nächsten Seite in Beziehung zu den jeweils erreichten Klassifikationsleistungen beschreibend dargestellt. Das Verhältnis zwischen Leistung und Trainingszeit war für AE-MLP am ungünstigsten und für FE-SVM am besten. Lediglich in Bezug auf die erreichte

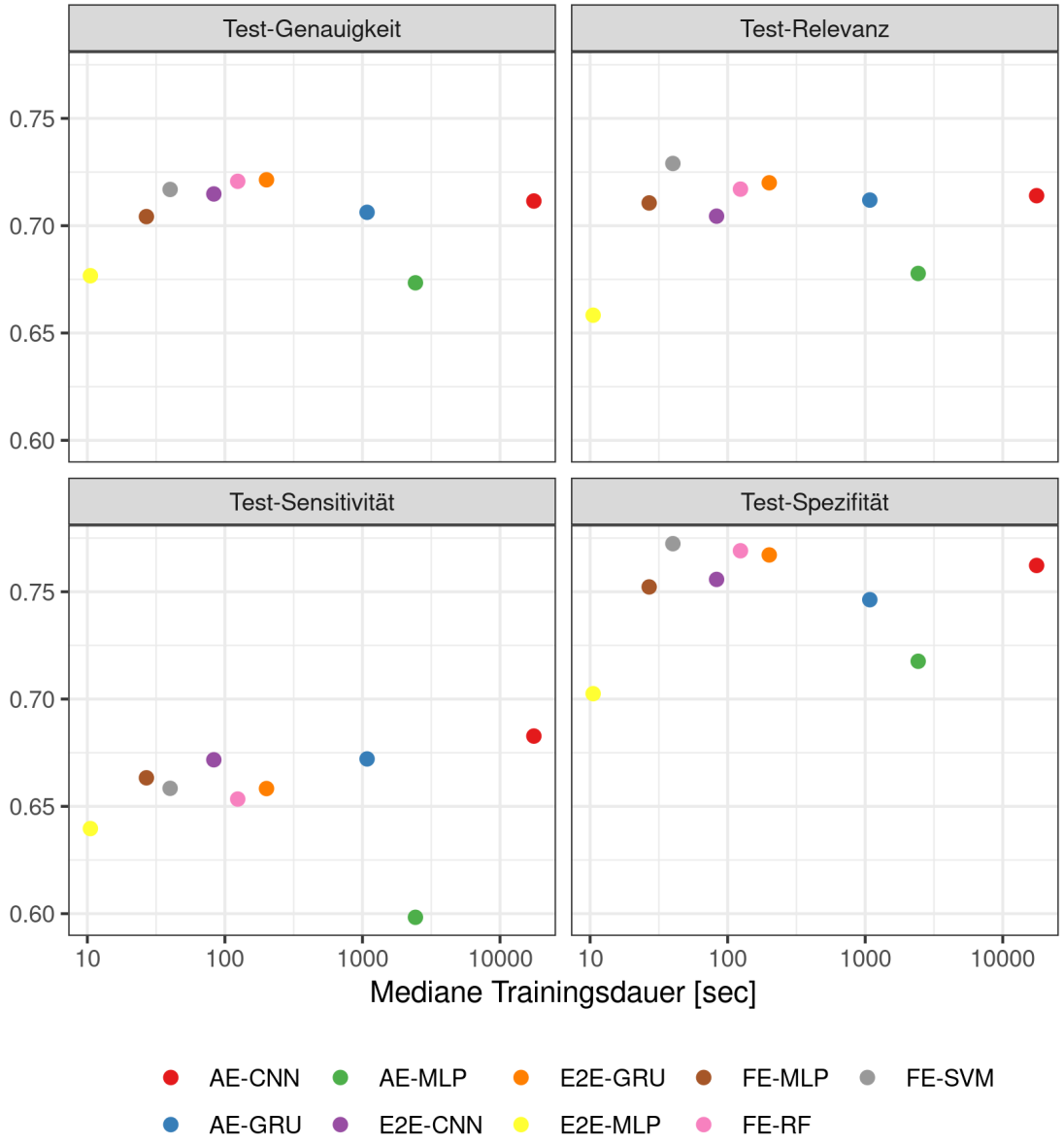

Abbildung 4.1: Beziehung von medianen Leistungen und medianer Trainingszeit je Klassifikationsansatz

Sensitivität schnitt FE-MLP günstiger ab. Bei etwas kürzerer Trainingszeit erreichte FE-MLP eine etwas bessere Sensitivität als FE-SVM.

Die Ende-zu-Ende-Ansätze, bei denen die tatsächlich für die Klassifikation verwendeten Features erlernt wurden, waren den Ansätzen mit Feature Engineering, bei denen „manuell“ zusätzliche Features erzeugt wurden, nicht grundsätzlich überlegen im Bezug auf die Klassifikationsleistung. Allerdings waren sie deutlich aufwändiger zu trainieren. Diese Beobachtung veranlasst mich zur These, dass sich durch die Einbeziehung von zusätzlichem Wissen in ein Klassifikationsmodell bei gleicher Vorhersageleistung des Modells Rechenzeit zum Erlernen der Modellpara-

meter einsparen lässt. In dem meisten Fällen dürfte das zusätzliche Wissen dem Modell aber hinzugefügt werden, um dessen Vorhersageleistung zu verbessern wenn keine ausreichenden Trainingsdaten zur Verfügung stehen [vgl. vRMB+20].

In meiner Untersuchung konnte ich nicht feststellen, dass generative Ansätze mit unüberwachtem Vortraining die Klassifikationsleistung verbessern, indem mehr Daten für das Training verwendet werden können, wie es von Géron [Gér20, S. 348–352] angeregt wurde (vgl. Tabelle 3.7 auf Seite 58). Bei den untersuchten künstlichen neuronalen Netzen handelte es sich um relativ kleine Netze mit entsprechend wenigen zu erlernenden Parametern. Vermutlich waren im vorliegenden Fall bereits allein die gelabelten Beobachtungen als Trainingsdaten ausreichend, um die Modellparameter zu erlernen, sodass durch das unüberwachte Vortraining mit allen Beobachtungen einschließlich der ungelabelten keine weitere Verbesserung erreicht werden konnte. Allerdings hatten die generativen Ansätze einen erheblich höheren Ressourcenbedarf als die entsprechenden Ende-zu-Ende-Ansätze (Tabelle 3.5 auf Seite 57). Bei den generativen Ansätzen wurde der Ressourcenbedarf insgesamt bestimmt, d. h. es wurde neben dem überwachten Klassifikationstraining auch das unüberwachtes Vortraining berücksichtigt. Dadurch erklären sich die deutlich längeren Trainingszeiten und auch der größere Arbeitsspeicherbedarf, weil die Autoencoder deutlich mehr zu erlernende Parameter enthielten als die Klassifikationsnetze. Wenn keine deutlich größeren künstlichen neuronalen Netze für die Erkennung von Lahmheit bei Milchkühen eingesetzt werden sollen und wenn ähnlich viele Trainingsdaten wie für die vorliegende Arbeit verfügbar sind, dann erscheint der Aufwand für die Entwicklung und das Training generativer Ansätze nicht sinnvoll für diesen Anwendungsfall.

4.4 Schlussfolgerungen

In den Vergleich verschiedener Klassifikationsansätze zur Erkennung von Lahmheit bei Milchkühen an Hand von indirekten Variablen zur Aktivität, zur Leistung und zu den Eigenschaften der Kühe wurden in der vorliegenden Arbeit neben Ansätzen den klassischen maschinellen Lernens (Random Forest und SVM) und MLP nach Feature Engineering sowohl Ende-zu-Ende-Ansätze mit MLP, CNN und GRU als auch generative Ansätze einbezogen. Neben der erwarteten Klassifikationsleistung wurde der Bedarf an Ressourcen für das Training und die Vorhersage bei den neun Ansätzen untersucht. Keiner der untersuchten Ansätze erwies sich als zu aufwändig für den Praxiseinsatz, d. h. der Ressourcenbedarf aller Ansätze war so gering, dass sie auf leistungsfähigeren Personal Computern (ggf. mit GPU) trainiert werden könnten.

Allerdings waren die erwarteten Klassifikationsleistungen der verwendeten Modelle zwar insgesamt akzeptabel, aber für mögliche Anwendungen in der Praxis noch deutlich zu schlecht. Es besteht deshalb der Bedarf, weitere Modelle für diesen Anwendungsfall zu evaluieren und zu optimieren. Die SVM im Ansatz mit Feature Engineering erreichte zwar die besten Klassifikationsleistungen bei sehr geringem Ressourcenbedarf (Laufzeit und Arbeitsspeicherbelegung), doch mit ei-

ner Genauigkeit von 0,72 ist das Modell nicht geeignet für die direkte Verwendung in Systemen zur automatischen Lahmheitserkennung. Dieses Modell bietet aber nicht viele Möglichkeiten für Anpassungen, die zu deutlichen Verbesserungen der Klassifikationsleistung führen könnten. Die Ende-zu-Ende-Ansätze mit tiefen neuronalen Netzen klassifizierten vergleichbar gut. Deshalb schlage ich vor, in weiteren Studien über Klassifikatoren für die Lahmheitserkennung aus Aktivitätsdaten in erster Linie solche Ansätze mit CNN, RNN oder mit Kombinationen daraus einzubeziehen. Durch die enorme Flexibilität tiefer neuronaler Netze bieten diese Ansätze ein riesiges Anpassungspotential an die spezifischen Anwendungsfälle. Ein sehr interessanter Ansatz wäre es, ein tiefes neuronales Netz mit CNN und/oder RNN zum Erlernen der relevanten Merkmale zu verwenden und die eigentliche Klassifikation dann durch eine SVM durchführen zu lassen. Trotz des Aufwands, die Klassifikationsmodelle für jede Herde spezifisch trainieren zu müssen, scheint es mir lohnenswert, Möglichkeiten zu prüfen, wie ein Merkmal für die individuellen Kühe bei der Klassifikation berücksichtigt werden könnte, da sich auf diese Weise die Unterschiede zwischen den Kühen besser im Modell abbilden lassen.

Der Aufwand und der Ressourcenbedarf für die Entwicklung und das Training generativer Klassifikationsansätze standen in meiner Untersuchung zur Lahmheitserkennung bei Milchkühen aus Aktivitätsdaten in keinem guten Verhältnis zum Zugewinn an Klassifikationsgenauigkeit gegenüber Ende-zu-Ende-Ansätzen mit vergleichbaren Modellen. Es erscheint daher nicht sinnvoll, Ansätze mit unüberwachtem Vortraining für diesen Anwendungsfall zu entwickeln — es sei denn, es sollen wesentlich größere neuronale Netze verwendet werden als es in der vorliegenden Arbeit der Fall war.

Für den Vergleich der Leistungsfähigkeit mehrerer Modelle gleichzeitig an Hand verschiedener Kriterien, die nicht unabhängig voneinander sind, ist das in der vorliegenden Arbeit beschriebene Bayes'sche multivariate lineare Modell gut geeignet. Es sollte statistischen Tests bzw. Modellen vorgezogen werden, bei welchen die Leistungen nach jedem Kriterium jeweils separat verglichen werden, da nur in einem gemeinsamen statistischen Modell die Abhängigkeiten zwischen den Kriterien berücksichtigt werden können.

Literaturverzeichnis

[Aba+15] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado u. a. *TensorFlow: Large-scale machine learning on heterogeneous systems*. 2015. URL: https://www.tensorflow.org/.

[Aba+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin u. a. „TensorFlow: A System for large-scale machine learning“. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, S. 265–283. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[AFS19] M. Alsaaod, M. Fadul und A. Steiner. „Automatic lameness detection in cattle“. In: *Veterinary Journal* 246 (2019), S. 35–44. DOI: 10.1016/j.tvjl.2019.01.005.

[Als+12] M. Alsaaod, C. Römer, J. Kleinmanns, K. Hendriksen, S. Rose-Meierhöfer, L. Plümer und W. Büscher. „Electronic detection of lameness in dairy cows through measuring pedometric activity and lying behavior“. In: *Applied Animal Behaviour Science* 142.3 (2012), S. 134–141. DOI: 10.1016/j.applanim.2012.10.001.

[ABH10] S. Archer, N. Bell und J. Huxley. „Lameness in UK dairy cows: A review of the current status“. In: *In Practice* 32.10 (2010), S. 492–504. DOI: 10.1136/inp.c6672.

[Bee+16] G. Beer, M. Alsaaod, A. Starke, G. Schuepbach-Regula, H. Müller, P. Kohler und A. Steiner. „Use of extended characteristics of locomotion and feeding behavior for automated identification of lame dairy cows“. In: *PloS One* 11.5 (2016), e0155796. DOI: 10.1371/journal.pone.0155796.

[Ben+17] A. Benavoli, G. Corani, J. Demšar und M. Zaffalon. „Time for a change: A tutorial for comparing multiple classifiers through Bayesian analysis“. In: *Journal of Machine Learning Research* 18.77 (2017), S. 1–36.

[BMP20] L. Bennett, B. Melchers und B. Proppe. *Curta: A general-purpose high-performance computer at ZEDAT, Freie Universität Berlin*. 2020. DOI: 10.17169/refubium-26754.

[BOS21] G.M. Borghart, L.E. O'Grady und J.R. Somers. „Prediction of lameness using automatically recorded activity, behavior and production data in post-parturient Irish dairy cows". In: *Irish Veterinary Journal* 74.1 (2021), S. 4. DOI: 10.1186/s13620-021-00182-6.

[Bre01] L. Breiman. „Random forests". In: *Machine Learning* 45.1 (2001), S. 5–32. DOI: 10.1023/A:1010933404324.

[Bun94] W.L. Buntine. „Operations for learning with graphical models". In: *Journal of Artificial Intelligence Research* 2 (1994), S. 159–225. DOI: 10.1613/jair.62.

[Bus+19] V. Busin, L. Viora, G. King, M. Tomlinson, J. LeKernec, N. Jonsson und F. Fioranelli. „Evaluation of lameness detection using radar sensing in ruminants". In: *The Veterinary Record* 185.18 (2019), S. 572. DOI: 10.1136/vr.105407.

[Bya+19] J. Byabazaire, C. Olariu, M. Taneja und A. Davy. „Lameness detection as a service: Application of machine learning to an internet of cattle". In: *2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC)*. Jan. 2019, S. 1–6. DOI: 10.1109/CCNC.2019.8651681.

[Cha+13] N. Chapinal, A.K. Barrientos, M.A.G. von Keyserlingk, E. Galo und D.M. Weary. „Herd-level risk factors for lameness in freestall farms in the northeastern United States and California". In: *Journal of Dairy Science* 96.1 (2013), S. 318–328. DOI: 10.3168/jds.2012-5940.

[Cho+14] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk und Y. Bengio. „Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, S. 1724–1734. DOI: 10.3115/v1/D14-1179.

[CUH16] D.-A. Clevert, T. Unterthiner und S. Hochreiter. „Fast and accurate deep network learning by exponential linear units (ELUs)". In: *Proceedings of the International Conference on Learning Representations*. 2016. DOI: 10.48550/ARXIV.1511.07289.

[CT65] J.W. Cooley und J.W. Tukey. „An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* 19.90 (1965), S. 297–301. DOI: 10.1090/S0025-5718-1965-0178586-1.

[Cov69] T.M. Cover. „Learning in pattern recognition". In: *Methodologies of pattern recognition*. Hrsg. von S. Watanabe. New York: Academic Press, 1969, S. 111–132. DOI: 10.1016/B978-1-4832-3093-1.50012-2.

[dMol+13] R.M. de Mol, G. André, E.J.B. Bleumer, J.T.N. van der Werf, Y. de Haas und C.G. van Reenen. „Applicability of day-to-day variation in behavior for the automated detection of lameness in dairy cows".

In: *Journal of Dairy Science* 96.6 (2013), S. 3703–3712. DOI: 10.3168/jds.2012-6305.

[DPW20] A. Dempster, F. Petitjean und G.I. Webb. „ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels“. In: *Data Mining and Knowledge Discovery* 34.5 (2020), S. 1454–1495. DOI: 10.1007/s10618-020-00701-z.

[Dem06] J. Demšar. „Statistical comparisons of classifiers over multiple data sets“. In: *Journal of Machine Learning Research* 7 (2006), S. 1–30.

[Den16] M.J. Denwood. „Runjags : An R package providing interface utilities, model templates, parallel computing methods and additional distributions for MCMC models in JAGS“. In: *Journal of Statistical Software* 71.9 (2016). DOI: 10.18637/jss.v071.i09.

[Doz16] T. Dozat. „Incorporating Nesterov momentum into Adam“. In: *ICLR 2016 - Workshop Track*. International Conference on Learning Representations. San Juan, Puerto Rico, 2. Mai 2016. URL: https://cs229.stanford.edu/proj2015/054%5C_report.pdf (besucht am 08. 06. 2022).

[DHS01] R.O. Duda, P.E. Hart und D.G. Stork. *Pattern classification*. 2. Aufl. New York: Wiley, 2001.

[Dut+18] K.J. Dutton-Regester, T.S. Barnes, J.D. Wright, J.I. Alawneh und A.R. Rabiee. „A systematic review of tests for the detection and diagnosis of foot lesions causing lameness in dairy cows“. In: *Preventive Veterinary Medicine* 149 (2018), S. 53–66. DOI: 10.1016/j.prevetmed.2017.11.003.

[Dut+20] K.J. Dutton-Regester, T.S. Barnes, J.D. Wright und A.R. Rabiee. „Lameness in dairy cows: Farmer perceptions and automated detection technology“. In: *Journal of Dairy Research* 87.S1 (2020), S. 67–71. DOI: 10.1017/S0022029920000497.

[Faw+19] H.I. Fawaz, G. Forestier, J. Weber, L. Idoumghar und P.-A. Muller. „Deep learning for time series classification: A review“. In: *Data Mining and Knowledge Discovery* 33.4 (2019), S. 917–963. DOI: 10.1007/s10618-019-00619-1.

[Faw+20] H.I. Fawaz, B. Lucas, G. Forestier, C. Pelletier, D.F. Schmidt, J. Weber, G.I. Webb, L. Idoumghar, P.-A. Muller und F. Petitjean. „InceptionTime: Finding AlexNet for time series classification“. In: *Data Mining and Knowledge Discovery* 34.6 (2020), S. 1936–1962. DOI: 10.1007/s10618-020-00710-y.

[FSH18] T. Fiolka, F. Schächter und J. Heinskill. „Automatische Rückenlinienanalyse bei Milchkühen aus Bilddaten“. In: *24. Workshop Computer-Bildanalyse in der Landwirtschaft*. 2018, S. 71.

[FOS21] M. Freitag, S. Oehler und K.F. Stock. „Nutzungsdauer und Abgangsursachen von Milchkühen - Neues aus NRW“. In: *Forum angewandte Forschung in der Rinder- und Schweinefütterung*. Fulda, 21–22. März 2017, S. 95–98.

[Fri01] J.H. Friedman. „Greedy function approximation: A gradient boosting machine“. In: *The Annals of Statistics* 29.5 (2001), S. 1189–1232. DOI: 10.1214/aos/1013203451.

[fLei19] Deutscher Verband für Leistungs- und Qualitätsprüfungen e.V. *Entwicklung einer Dienstleistung zur Verbesserung der Klauengesundheit von Milchkühen durch Vernetzung und Verdichtung von Daten für das Tiergesundheitsmanagement (KLAUENfitnet) - Teilprojekt 1: Abschlussbericht*. 2019. DOI: 10.2314/GBV:1067853286.

[Gar+14] E. Garcia, I. Klaas, J.M. Amigo, R. Bro und C. Enevoldsen. „Lameness detection challenges in automated milking systems addressed with partial least squares discriminant analysis“. In: *Journal of Dairy Science* 97.12 (2014), S. 7476–7486. DOI: 10.3168/jds.2014-7982.

[Gel06] A. Gelman. „Prior distributions for variance parameters in hierarchical models“. In: *Bayesian Analysis* 1 (2006), S. 515–533.

[GR92] A. Gelman und D.B. Rubin. „Inference from iterative simulation using multiple sequences“. In: *Statistical Science* 7.4 (1992), S. 457–472.

[Gér20] A. Géron. *Praxiseinstieg Machine Learning mit Scikit-Learn, Keras und TensorFlow: Konzepte, Tools und Techniken für intelligente Systeme*. Übers. von K. Rother und T. Demmig. 2. Aufl. Heidelberg: O'Reilly, 2020.

[GB10] X. Glorot und Y. Bengio. „Understanding the difficulty of training deep feedforward neural networks“. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Bd. 9. Sardinia, Italy: JMLR Workshop und Conference Proceedings, 31. März 2010, S. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html (besucht am 08. 06. 2022).

[GBC16] I. Goodfellow, Y. Bengio und A. Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org (besucht am 24. 09. 2022).

[GGO18] B.E. Griffiths, D. Grove White und G. Oikonomou. „A cross-sectional study into the prevalence of dairy cattle lameness and associated herd-level risk factors in England and Wales“. In: *Frontiers in Veterinary Science* 5 (2018), S. 65. DOI: 10.3389/fvets.2018.00065.

[Gri+19] K. Grimm, B. Haidn, M. Erhard, M. Tremblay und D. Döpfer. „New insights into the association between lameness, behavior, and performance in Simmental cows“. In: *Journal of Dairy Science* 102.3 (2019), S. 2453–2468. DOI: 10.3168/jds.2018-15035.

[Gru99] E. Grunert. „Sexualzyklus“. In: *Fertilitätsstörungen beim weiblichen Rind*. Hrsg. von E. Grunert, D. Ahlers und M. Berchtold. 3. Aufl. Berlin: Parey, 1999.

[Hal+18] J. Haladjian, J. Haug, S. Nüske und B. Bruegge. „A wearable sensor system for lameness detection in dairy cattle“. In: *Multimo-*

dal Technologies and Interaction 2.2 (2018), S. 27. DOI: 10.3390/mti2020027.

[Har+20] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau u. a. „Array programming with NumPy“. In: *Nature* 585.7825 (2020), S. 357–362. DOI: 10.1038/s41586-020-2649-2.

[HTF09] T. Hastie, R. Tibshirani und J.H. Friedman. *The elements of statistical learning: Data mining, inference, and prediction.* 2. Aufl. Springer series in statistics. New York: Springer, 2009.

[He+15] K. He, X. Zhang, S. Ren und J. Sun. „Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification“. In: *2015 IEEE International Conference on Computer Vision (ICCV).* Santiago, Chile: IEEE, Dez. 2015, S. 1026–1034. DOI: 10.1109/ICCV.2015.123.

[Hig+17] J.H. Higginson Cutler, J. Rushen, A.M. de Passillé, J. Gibbons, K. Orsel, E. Pajor, H.W. Barkema, L. Solano, D. Pellerin, D. Haley und E. Vasseur. „Producer estimates of prevalence and perceived importance of lameness in dairy herds with tiestalls, freestalls, and automated milking systems“. In: *Journal of Dairy Science* 100.12 (2017), S. 9871–9880. DOI: 10.3168/jds.2017-13008.

[Hir21] J. Hirschle. *Machine Learning für Zeitreihen: Einstieg in Regressions-, ARIMA- und Deep Learning-Verfahren mit Python.* München: Hanser, 2021.

[Hub64] P.J. Huber. „Robust estimation of a location parameter“. In: *The Annals of Mathematical Statistics* 35.1 (1964), S. 73–101. DOI: 10.1214/aoms/1177703732.

[Hum+20] B. Humm, H. Bense, J. Bock, M. Classen, O. Halvani, C. Herta, T. Hoppe, O. Juwig und M. Siegel. „Applying machine intelligence in practice: Selected results of the 2019 Dagstuhl Workshop on Applied Machine Intelligence“. In: *Informatik Spektrum* 43.2 (2020), S. 137–144. DOI: 10.1007/s00287-020-01259-2.

[Hut+21] P.R. Hut, M.M. Hostens, M.J. Beijaard, F.J.C.M. van Eerdenburg, J.H.J.L. Hulsen, G.A. Hooijer, E.N. Stassen und M. Nielen. „Associations between body condition score, locomotion score, and sensor-based time budgets of dairy cattle during the dry period and early lactation“. In: *Journal of Dairy Science* 104.4 (2021), S. 4746–4763. DOI: 10.3168/jds.2020-19200.

[IS15] S. Ioffe und C. Szegedy. „Batch normalization: Accelerating deep network training by reducing internal covariate shift“. In: *Proceedings of the 32nd International Conference on Machine Learning.* PMLR, 1. Juni 2015, S. 448–456. URL: https://proceedings.mlr.press/v37/ioffe15.html (besucht am 08. 06. 2022).

[Ito+10] K. Ito, M.A.G. von Keyserlingk, S.J. LeBlanc und D.M. Weary. „Lying behavior as an indicator of lameness in dairy cows“. In: *Journal of*

Dairy Science 93.8 (2010), S. 3553–3560. DOI: 10.3168/jds.2009-2951.

[Jen+22] K.C. Jensen, A.W. Oehm, A. Campe, A. Stock, S. Woudstra, M. Feist, K.E. Müller, M. Hoedemaker und R. Merle. „German farmers' awareness of lameness in their dairy herds". In: *Frontiers in Veterinary Science* 9 (2022). DOI: 10.3389/fvets.2022.866791.

[Jia+22] B. Jiang, H. Song, H. Wang und C. Li. „Dairy cow lameness detection using a back curvature feature". In: *Computers and Electronics in Agriculture* 194 (2022), S. 106729. DOI: 10.1016/j.compag.2022.106729.

[Kam+13] C. Kamphuis, E. Frank, J.K. Burke, G.A. Verkerk und J.G. Jago. „Applying additive logistic regression to data derived from sensors monitoring behavioral and physiological characteristics of dairy cows to detect lameness". In: *Journal of Dairy Science* 96.11 (2013), S. 7043–7053. DOI: 10.3168/jds.2013-6993.

[KZL20] X. Kang, X.D. Zhang und G. Liu. „Accurate detection of lameness in dairy cattle with computer vision: A new and individualized detection strategy based on the analysis of the supporting phase". In: *Journal of Dairy Science* 103.11 (2020), S. 10628–10638. DOI: 10.3168/jds.2020-18288.

[Kan+20] K. Kaniyamattam, J. Hertl, G. Lhermie, U. Tasch, R. Dyer und Y.T. Gröhn. „Cost benefit analysis of automatic lameness detection systems in dairy herds: A dynamic programming approach". In: *Preventive Veterinary Medicine* 178 (2020), S. 104993. DOI: 10.1016/j.prevetmed.2020.104993.

[Kha+20] M. Khan, H. Wang, A. Ngueilbaye und A. Elfatyany. „End-to-end multivariate time series classification via hybrid deep learning architectures". In: *Personal and Ubiquitous Computing* (2020). DOI: 10.1007/s00779-020-01447-7.

[Kib+21] H. Kibirige, G. Lamp, J. Katins, Gdowding, Austin, Matthias-K, T. Funnell u. a. *has2k1/plotnine: v0.8.0*. Version 0.8.0. 25. März 2021. DOI: 10.5281/ZENODO.4636791.

[Kla+17] G. Klambauer, T. Unterthiner, A. Mayr und S. Hochreiter. „Self-normalizing neural networks". In: *Advances in neural information processing systems (NIPS)*. 31st International Conference on Neural Information Processing Systems. Long Beach, CA, USA, 2017, S. 972–981. DOI: 10.48550/ARXIV.1706.02515.

[Kla] Klauenfitnet 2.0. *Anleitung zur Bewegungsbeurteilung*. URL: https://www.klauenfitnet.de/projektnews/download/ (besucht am 06. 04. 2022).

[Kof+21] J. Kofler, B. Fürst-Waltl, M. Dourakas, F. Steininger und C. Egger-Danner. „Impact of lameness on milk yield in dairy cows in Austria – Results from the Efficient-Cow-Project". In: *Schweizer Archiv für Tierheilkunde* 163.2 (2021), S. 123–138. DOI: 10.17236/sat00290.

[Kru11] J.K. Kruschke. *Doing Bayesian data analysis: A tutorial with R and BUGS*. 1. Aufl. Burlington, USA: Elsevier Academic Press, 2011.

[LKL14] M. Längkvist, L. Karlsson und A. Loutfi. „A review of unsupervised feature learning and deep learning for time-series modeling“. In: *Pattern Recognition Letters* 42 (2014), S. 11–24. DOI: 10.1016/j.patrec.2014.01.008.

[Las+21] J. Lasser, C. Matzhold, C. Egger-Danner, B. Fuerst-Waltl, F. Steininger, T. Wittek und P. Klimek. „Integrating diverse data sources to predict disease risk in dairy cattle — a machine learning approach“. In: *Journal of Animal Science* 99.11 (2021), skab294. DOI: 10.1093/jas/skab294.

[LL12] E. Lesaffre und A.B. Lawson. *Bayesian Biostatistics*. Chichester, England: John Wiley & Sons, 2012.

[MBL19] D. Makowski, M. Ben-Shachar und D. Lüdecke. „bayestestR: Describing effects and their uncertainty, existence and significance within the Bayesian framework“. In: *Journal of Open Source Software* 4.40 (2019), S. 1541. DOI: 10.21105/joss.01541.

[Mak+19] D. Makowski, M.S. Ben-Shachar, S.H.A. Chen und D. Lüdecke. „Indices of effect existence and significance in the Bayesian framework“. In: *Frontiers in Psychology* 10 (2019), S. 2767. DOI: 10.3389/fpsyg.2019.02767.

[Mar+09] P. Martiskainen, M. Järvinen, J.-P. Skön, J. Tiirikainen, M. Kolehmainen und J. Mononen. „Cow behaviour pattern recognition using a three-dimensional accelerometer and support vector machines“. In: *Applied Animal Behaviour Science* 119.1 (2009), S. 32–38. DOI: 10.1016/j.applanim.2009.03.005.

[Mas+11] J. Masci, U. Meier, D. Cireşan und J. Schmidhuber. „Stacked convolutional auto-encoders for hierarchical feature extraction“. In: *Artificial Neural Networks and Machine Learning — ICANN 2011*. Hrsg. von T. Honkela, W. Duch, M. Girolami und S. Kaski. Bd. 6791. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, S. 52–59. DOI: 10.1007/978-3-642-21735-7_7.

[MTK13] B. Miekley, I. Traulsen und J. Krieter. „Principal component analysis for the early detection of mastitis and lameness in dairy cows“. In: *Journal of Dairy Research* 80.3 (2013), S. 335–343. DOI: 10.1017/S0022029913000290.

[Mur] W. Murphy. *Implementing an interface in Python*. Real Python. URL: https://realpython.com/python-interface/ (besucht am 06. 07. 2022).

[NJ01] A. Ng und M. Jordan. „On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes“. In: *Advances in Neural Information Processing Systems*. Hrsg. von T. Dietterich, S. Becker und Z. Ghahramani. Bd. 14. MIT Press, 2001. URL: https://proceedings.neurips.cc/paper/2001/file/7b7a53e239400a13bd6be6c91c4f6c4e-Paper.pdf.

[Nwe+18] H.F. Nweke, Y.W. Teh, M.A. Al-garadi und U.R. Alo. „Deep learning algorithms for human activity recognition using mobile and wearable sensor networks: State of the art and research challenges“. In: *Expert Systems with Applications* 105 (2018), S. 233–261. DOI: 10.1016/j.eswa.2018.03.056.

[OLe+20] N.W. O'Leary, D.T. Byrne, A.H. O'Connor und L. Shalloo. „Invited review: Cattle lameness detection with accelerometers“. In: *Journal of Dairy Science* 103.5 (2020), S. 3895–3911. DOI: 10.3168/jds.2019-17123.

[Oeh+19] A.W. Oehm, G. Knubben-Schweizer, A. Rieger, A. Stoll und S. Hartnack. „A systematic review and meta-analyses of risk factors associated with lameness in dairy cows“. In: *BMC Veterinary Research* 15.1 (2019), S. 346. DOI: 10.1186/s12917-019-2095-2.

[OS51] D. Olds und D.M. Seath. „Repeatability of the estrous cycle length in dairy cattle“. In: *Journal of Dairy Science* 34.7 (1951), S. 626–632. DOI: 10.3168/jds.S0022-0302(51)91757-2.

[PK07] M.E. Pastell und M. Kujala. „A probabilistic neural network model for lameness detection“. In: *Journal of Dairy Science* 90.5 (2007), S. 2283–2292. DOI: 10.3168/jds.2006-267.

[Ped+11] F. Pedregosa, G. Varoquaux, A. Gramfort u. a. „Scikit-learn: Machine Learning in Python“. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830.

[PHJ+21] J.N. Philipp, G. Hylander, V. Janapati u. a. *Cannot convert a symbolic tensor (gru/strided_slice:0) to a numpy array.* 18. Feb. 2021. URL: https://github.com/tensorflow/tensorflow/issues/47242 (besucht am 03. 08. 2022).

[Pie+20] D. Piette, T. Norton, V. Exadaktylos und D. Berckmans. „Individualised automated lameness detection in dairy cows and the impact of historical window length on algorithm performance“. In: *Animal* 14.2 (2020), S. 409–417. DOI: 10.1017/S1751731119001642.

[Plu03] M. Plummer. „JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling“. In: *3rd International Workshop on Distributed Statistical Computing (DSC 2003)* 124 (Apr. 2003).

[Pou+10] A. Poursaberi, C. Bahr, A. Pluk, A. van Nuffel und D. Berckmans. „Real-time automatic lameness detection based on back posture extraction in dairy cattle: Shape analysis of cow with image processing techniques“. In: *Computers and Electronics in Agriculture* 74.1 (2010), S. 110–119. DOI: 10.1016/j.compag.2010.07.004.

[Pra20] PraeRi. *Tiergesundheit, Hygiene und Biosicherheit in deutschen Milchkuhbetrieben – Eine Prävalenzstudie (PraeRi).* 2020. URL: https://ibei.tiho-hannover.de/praeri/pages/69#_AB (besucht am 09. 04. 2022).

[Pue+21] M.A. Puerto, E. Shepley, R.I. Cue, D. Warner, J. Dubuc und E. Vasseur. „The hidden cost of disease: II. Impact of the first incidence of lameness on production and economic indicators of primiparous

dairy cows“. In: *Journal of Dairy Science* 104.7 (2021), S. 7944–7955. DOI: 10.3168/jds.2020-19585.

[Pyt22] Python Software Foundation. *Python 3.8.13 documentation*. 2022. URL: https://docs.python.org/3.8/index.html (besucht am 03. 08. 2022).

[Qia+21] Y. Qiao, H. Kong, C. Clark, S. Lomax, D. Su, S. Eiffert und S. Sukkarieh. „Intelligent perception-based cattle lameness detection and behaviour recognition: A review“. In: *Animals* 11.11 (2021), S. 3033. DOI: 10.3390/ani11113033.

[R C22] R Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. Vienna, Austria, 2022. URL: https://www.R-project.org/.

[RL92] A.E. Raftery und S.M. Lewis. „Comment: One long run with diagnostics: Implementation strategies for Markov Chain Monte Carlo“. In: *Statistical science* 7.4 (1992), S. 493–497.

[Reb+21] J. Reback, jbrockmendel, W. McKinney, J. van den Bossche, T. Augspurger u. a. *pandas-dev/pandas: Pandas 1.2.5*. Version 1.2.5. Juni 2021. DOI: 10.5281/zenodo.5013202.

[RH18] S. Reith und S. Hoy. „Review: Behavioral signs of estrus and the potential of fully automated systems for detection of estrus in dairy cattle“. In: *Animal: An International Journal of Animal Bioscience* 12.2 (2018), S. 398–407. DOI: 10.1017/S1751731117001975.

[Roe+10] J. Roelofs, F. López-Gatius, R.H.F. Hunter, F.J.C.M. van Eerdenburg und Ch. Hanzen. „When is a cow in estrus? Clinical and practical aspects“. In: *Theriogenology* 74.3 (2010), S. 327–344. DOI: 10.1016/j.theriogenology.2010.02.016.

[Rui+21] A.P. Ruiz, M. Flynn, J. Large, M. Middlehurst und A. Bagnall. „The great multivariate time series classification bake off: A review and experimental evaluation of recent algorithmic advances“. In: *Data Mining and Knowledge Discovery* 35.2 (2021), S. 401–449. DOI: 10.1007/s10618-020-00727-3.

[Sch+17] K. Schindhelm, I. Lorenzini, M. Tremblay, D. Dopfer, S. Reese und B. Haidn. „Automatically recorded performance and behaviour parameters as risk factors for lameness in dairy cattle“. In: *Chemical Engineering Transactions* 58 (2017), S. 583–588. DOI: 10.3303/CET1758098.

[Sch+14] A. Schlageter-Tello, E.A.M. Bokkers, P.W.G. Groot Koerkamp, T. van Hertem, S. Viazzi, C.E.B. Romanini, I. Halachmi, C. Bahr, D. Berckmans und K. Lokhorst. „Manual and automatic locomotion scoring systems in dairy cows: A review“. In: *Preventive Veterinary Medicine* 116.1 (2014), S. 12–25. DOI: 10.1016/j.prevetmed.2014.06.006.

[Sch15] A. Schranner. „Prävalenzen von Lahmheiten bei Milchkühen in niedersächsischen Milchviehbetrieben“. Hannover: Tierärztliche Hochschule Hannover, 2015.

[Sha+21] S. Shahinfar, M. Khansefid, M. Haile-Mariam und J.E. Pryce. „Machine learning approaches for the prediction of lameness in dairy cows“. In: *Animal* 15.11 (2021), S. 100391. DOI: 10.1016/j.animal.2021.100391.

[Smi18] L.N. Smith. „A disciplined approach to neural network hyper-parameters: Part 1 – Learning rate, batch size, momentum, and weight decay“. In: *arXiv* (24. Apr. 2018). URL: http://arxiv.org/abs/1803.09820 (besucht am 29. 12. 2021).

[Son+08] X. Song, T. Leroy, E. Vranken, W. Maertens, B. Sonck und D. Berckmans. „Automatic detection of lameness in dairy cattle — Vision-based trackway analysis in cow's locomotion“. In: *Computers and Electronics in Agriculture*. Smart Sensors in precision livestock farming 64.1 (2008), S. 39–44. DOI: 10.1016/j.compag.2008.05.016.

[SHK97] D.J. Sprecher, D.E. Hostetler und J.B. Kaneene. „A lameness scoring system that uses posture and gait to predict dairy cattle reproductive performance“. In: *Theriogenology* 47.6 (1997), S. 1179–1187. DOI: 10.1016/S0093-691X(97)00098-8.

[Tan+20] M. Taneja, J. Byabazaire, N. Jalodia, A. Davy, C. Olariu und P. Malone. „Machine learning based fog computing assisted data-driven approach for early lameness detection in dairy cattle“. In: *Computers and Electronics in Agriculture* 171 (2020), S. 105286. DOI: 10.1016/j.compag.2020.105286.

[Ten21] TensorFlow Developers. *TensorFlow*. Version 2.4.4. Nov. 2021. DOI: 10.5281/zenodo.5637331.

[vdGuc+17] T. van de Gucht, W. Saeys, A. van Nuffel, L. Pluym, K. Piccart, L. Lauwers, J. Vangeyte und S. van Weyenberg. „Farmers' preferences for automatic lameness-detection systems in dairy cattle“. In: *Journal of Dairy Science* 100.7 (2017), S. 5746–5757. DOI: 10.3168/jds.2016-12285.

[vdMH08] L. van der Maaten und G. Hinton. „Visualizing data using t-SNE.“ In: *Journal of Machine Learning Research* 9.11 (2008), S. 2579–2605.

[vHer+16] T. van Hertem, C. Bahr, A. Schlageter Tello, S. Viazzi, M. Steensels, C.E.B. Romanini, C. Lokhorst, E. Maltz, I. Halachmi und D. Berckmans. „Lameness detection in dairy cattle: single predictor v. multivariate analysis of image-based posture processing and behaviour and performance sensing“. In: *Animal* 10.9 (2016), S. 1525–1532. DOI: 10.1017/S1751731115001457.

[vHer+13] T. van Hertem, E. Maltz, A. Antler, C.E.B. Romanini, S. Viazzi, C. Bahr, A. Schlageter-Tello, C. Lokhorst, D. Berckmans und I. Halachmi. „Lameness detection based on multivariate continuous sensing of milk yield, rumination, and neck activity“. In: *Journal of Dairy Science* 96.7 (2013), S. 4286–4298. DOI: 10.3168/jds.2012-6188.

[vHuy+20] M. van Huyssteen, H.W. Barkema, S. Mason und K. Orsel. „Association between lameness risk assessment and lameness and foot lesion prevalence on dairy farms in Alberta, Canada“. In: *Journal of Dairy*

Science 103.12 (2020), S. 11750–11761. DOI: 10.3168/jds.2019-17819.

[vNuf+16] A. van Nuffel, T. van de Gucht, W. Saeys, B. Sonck, G. Opsomer, J. Vangeyte, K.C. Mertens, B. De Ketelaere und S. van Weyenberg. „Environmental and cow-related factors affect cow locomotion and can cause misclassification in lameness detection systems". In: *Animal* 10.9 (2016), S. 1533–1541. DOI: 10.1017/S175173111500244X.

[vRD09] G. van Rossum und F.L. Drake. *Python 3 reference manual*. Scotts Valley, CA: CreateSpace, 2009.

[Vil+19] M. Villettaz Robichaud, J. Rushen, A.M. de Passillé, E. Vasseur, K. Orsel und D. Pellerin. „Associations between on-farm animal welfare indicators and productivity and profitability on Canadian dairies: I. On freestall farms". In: *Journal of Dairy Science* 102.5 (2019), S. 4341–4351. DOI: 10.3168/jds.2018-14817.

[Vol+21] N. Volkmann, B. Kulig, S. Hoppe, J. Stracke, O. Hensel und N. Kemper. „On-farm detection of claw lesions in dairy cows based on acoustic analyses and machine learning". In: *Journal of Dairy Science* 104.5 (2021), S. 5921–5931. DOI: 10.3168/jds.2020-19206.

[vRMB+20] L. von Rüden, S. Mayer, K. Beckh u. a. *Informed machine learning — A taxonomy and survey of integrating knowledge into learning systems*. 2020. URL: http://publica.fraunhofer.de/dokumente/N-593257.html (besucht am 24. 09. 2022).

[WYO17] Z. Wang, W. Yan und T. Oates. „Time series classification from scratch with deep neural networks: A strong baseline". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. Mai 2017, S. 1578–1585. DOI: 10.1109/IJCNN.2017.7966039.

[Wei+18] H.C. Weigele, L. Gygax, A. Steiner, B. Wechsler und J.-B. Burla. „Moderate lameness leads to marked behavioral changes in dairy cows". In: *Journal of Dairy Science* 101.3 (2018), S. 2370–2382. DOI: 10.3168/jds.2017-13120.

[Wei08] N.S. Weiss. „Clinical epidemiology". In: *Modern epidemiology*. Hrsg. von K.J. Rothman, S. Greenland und T.L. Lash. 3. Aufl. Philadelphia: Lippincott Williams & Wilkins, 2008, S. 641–651.

[WS17] H.R. Whay und J.K. Shearer. „The impact of lameness on welfare of the dairy cow". In: *Veterinary Clinics of North America: Food Animal Practice* 33.2 (2017), S. 153–164. DOI: 10.1016/j.cvfa.2017.02.008.

[Wic16] H. Wickham. *ggplot2: Elegant graphics for data analysis*. New York: Springer-Verlag, 2016. URL: https://ggplot2.tidyverse.org.

[Wic+19] H. Wickham, M. Averick, J. Bryan, W. Chang, L. D'Agostino McGowan, R. François, G. Grolemund u. a. „Welcome to the tidyverse". In: *Journal of Open Source Software* 4.43 (2019), S. 1686. DOI: 10.21105/joss.01686.

[Wie05] D. Wiedenhöft. „Einfluss von Lahmheiten auf die Fruchtbarkeitsleistung von Milchkühen". Hannover: Tierärztliche Hochschule Hanno-

ver, 2005. URL: https://elib.tiho-hannover.de/receive/etd_mods_00002180?q=wiedenh%C3%B6ft (besucht am 11. 04. 2022).

[WR00] E. Wiesner und R. Ribbeck. *Lexikon der Veterinärmedizin*. 4. Aufl. Stuttgart: Enke, 2000.

[Wol96] D.H. Wolpert. „The lack of a priori distinctions between learning algorithms“. In: *Neural Computation* 8.7 (1996), S. 1341–1390. DOI: 10.1162/neco.1996.8.7.1341.

[Wu+20] D. Wu, Q. Wu, X. Yin, B. Jiang, H. Wang, D. He und H. Song. „Lameness detection of dairy cows based on the YOLOv3 deep learning algorithm and a relative step size characteristic vector“. In: *Biosystems Engineering* 189 (2020), S. 150–163. DOI: 10.1016/j.biosystemseng.2019.11.017.

[YGB12] C. Yunta, I. Guasch und A. Bach. „Short communication: Lying behavior of lactating dairy cows is influenced by lameness especially around feeding time“. In: *Journal of Dairy Science* 95.11 (2012), S. 6546–6549. DOI: 10.3168/jds.2012-5670.

[Zei+10] M.D. Zeiler, D. Krishnan, G.W. Taylor und R. Fergus. „Deconvolutional networks“. In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. Juni 2010, S. 2528–2535. DOI: 10.1109/CVPR.2010.5539957.

A

Quellcode

A.1 Abstrakte Klassen

Listing A.1: Modul `TrainTestInterface`

```
"""
Informal interface for evaluating different ML methods.

This module requires the following packages installed:

    * `numpy`
    * `pandas`

This module exports:

    * TrainTestInterface
"""

from abc import ABC, abstractmethod
import pandas as pd
import numpy as np


class TrainTestInterface(ABC):
    """
    Informal interface.
    """


    @abstractmethod
    def import_data(self,data_folder,n_proc,**kwargs):
        """
        Import data into attributes.

        Parameters
        ----------
        data_folder : string
            path to folder containing the data files
        n_proc : int
            number of processes to use
        **kwargs :
            additional parameters for window_function()

        Returns
        -------
        nothing
        """
        pass


    def prepare_fit(self,x_train,y_train,x_test,y_test,**kwargs):
        """
        Prepare data and model and fit model to data.

        Includes seeding of PRNG, preprocessing of the data, building,
        training and testing of the model.

        Parameters
        ----------
        x_train : np.array
            features for training
        y_train : 1d-np.array
            labels for training
        x_test : np.array
            features for testing
        y_test : 1d-np.array
            labels for testing
        **kwargs :
            hyperparameters to prepare the data or of the model
```

```
66          Returns
67          -------
68          metrics : dict
69              metrics used to evaluate the model
70          """
71          metrics = {
72              'Accuracy' : np.nan,
73              'Train_Accuracy' : np.nan,
74              'Precision' : np.nan,
75              'Sensitivity' : np.nan,
76              'Specifity' : np.nan,
77              'Time_train' : np.nan,
78              'Time_test' : np.nan
79          }
80          return metrics


83      @classmethod
84      def window_function(cls,window_data,**kwargs):
85          """
86          Transform time window data.

88          Parameters
89          ----------
90          window_data : array
91              data of one time window with shape=[time steps,features]
92          **kwargs :
93              additional parameters

95          Returns
96          -------
97          transformed : array
98              the transformed data
99          """
100         return window_data
```

Listing A.2: Modul `AbstractSearcher`

```
"""
Abstract class to search hyperparameter values.

This module requires the following packages installed:

    * `pickle`
    * `numpy`
    * `pandas`
    * `multiprocessing`
    * `sklearn.model_selection`

This module exports:

    * AbstractSearcher
"""

import pickle
import pandas as pd
import numpy as np
from multiprocessing import Pool
from sklearn.model_selection import LeaveOneGroupOut
from src.TrainTestInterface import TrainTestInterface

class AbstractSearcher(TrainTestInterface):
    """
    Abstract class to search hyperparameter values.

    Following methods need to be adapted to specific models:
        * self.gridsearch()
        * self.prepare_fit()
        * cls.window_function()

    Attributes
    ----------
    used : list
        indices of splits to be used for cross-validation
    logo :
        Leave One Group Out cross-validator
    X : np.array
        features
    Y : 1d-np.array
        labels
    Split : 1d-np.array
        split indicator for cross-validation
    metrics : pd.DataFrame
        hyperparameters and corresponding aggregated metrics
    """


    def __init__(self, used):
        """
        Parameters
        ----------
        used : list
            indices of splits to be used for cross-validation
        """
        self.used = used
        self.logo = LeaveOneGroupOut()
        self.X = None
        self.Y = None
        self.Split = None
        self.metrics = None


    def _get_split(self,index,data_folder,extraArgs):
```

```
66          """
67          Get reshaped data of one data split.

69          Parameters
70          ----------
71          index : int
72              index of data split to use
73          data_folder : string
74              path to folder containing the data files
75          extraArgs : list
76              parameters passed to _get_reshaped_data()

78          Returns
79          -------
80          split : tuple
81              (data,X,Y,Split)
82          """
83          with open(data_folder+'split'+str(index)+'.bin','rb') as file:
84              data = pickle.load(file)
85              data = pd.concat([data,pd.DataFrame({'Split' : np.repeat(↩
                    index,data.shape[0])},index=data.index)],axis=1)
86          X,Y,Split = self._get_reshaped_data(data.reset_index(),**↩
                extraArgs)
87          return(data,X,Y,Split)


90      def _get_reshaped_data(self,data,groups,feature_names,label_name,↩
            window_length,obs_shape,**kwargs):
91          """
92          Get reshaped data as time windows.

94          Notes
95          -----
96          Keeps only labelled observations.

98          Parameters
99          ----------
100         data : pd.DataFrame
101             data to work on
102         groups : list
103             names of grouping columns
104         feature_names : list
105             names of features as used in data
106         label_name : string
107             column name of label
108         window_length : int
109             length of time windows used for training and classification
110         obs_shape : tuple
111             shape of feature values of one observation
112         **kwargs :
113             additional parameters for window_function()

115         Returns
116         -------
117         data : tuple
118             (X,Y,Split)
119         """
120         X = np.empty((0,)+obs_shape)
121         Y = np.empty(0)
122         Split = np.empty(0)
123         idx_labelled = data.loc[pd.notna(data[label_name]),:].index.↩
                to_numpy()
124         idx_labelled = idx_labelled[idx_labelled >= window_length-1]
125         for i in idx_labelled:
126             grp_out = data.loc[i,groups[0]]
127             grp_in = data.loc[i,groups[1]]
128             data_i = data.loc[(i-window_length+1):i,:]
```

```
129             if any(data_i[groups[0]] != grp_out) or any(data_i[groups↩
                    [1]] != grp_in):
130                 continue
131             window = data_i.loc[(i-window_length+1):i,feature_names].↩
                    to_numpy()
132             window = self.window_function(window,**kwargs)
133             X = np.concatenate((X,window[np.newaxis]))
134             Y = np.concatenate((Y,data_i.loc[i,[label_name]]))
135             Split = np.concatenate((Split,data_i.loc[i,['Split']]))
136         return (X,Y,Split)


139     def import_data(self,data_folder,n_proc,**kwargs):
140         """
141         Import data into attributes.

143         Parameters
144         ----------
145         data_folder : string
146             path to folder containing the data files
147         n_proc : int
148             number of processes to use
149         **kwargs :
150             parameters for self._get_reshaped_data()

152         Returns
153         -------
154         nothing
155         """
156         arglst = [(i,data_folder,kwargs) for i in self.used]
157         with Pool(n_proc) as pool:
158             results = pool.starmap(self._get_split, arglst)
159         results = list(zip(*results))
160         data = pd.concat(list(results[0]))
161         self.X = np.concatenate(results[1])
162         self.Y = np.concatenate(results[2])
163         self.Split = np.concatenate(results[3])


166     def crossvalidate(self,**kwargs):
167         """
168         Cross-validate by leaving out one split per fold.

170         Calls self.prepare_fit() per fold.

172         Parameters
173         ----------
174         **kwargs :
175             parameters passed to self.prepare_fit(),
176             except for x_train,y_train,x_test,y_test

178         Returns
179         -------
180         metrics : dict
181             Means and standard deviations of metrics
182         """
183         results = np.empty((0,7))
184         for idx_train, idx_test in self.logo.split(X=self.X,groups=self.↩
                Split):
185             x_train = np.take(self.X,idx_train,axis=0)
186             x_test = np.take(self.X,idx_test,axis=0)
187             y_train = np.take(self.Y,idx_train,axis=0)
188             y_test= np.take(self.Y,idx_test,axis=0)
189             split_result = self.prepare_fit(x_train,y_train,x_test,↩
                    y_test,**kwargs)
190             results = np.vstack((
191                 results,
192                 np.array([
```

```
193                     split_result['Accuracy'],
194                     split_result['Train_Accuracy'],
195                     split_result['Precision'],
196                     split_result['Sensitivity'],
197                     split_result['Specifity'],
198                     split_result['Time_train'],
199                     split_result['Time_test']
200                 ])
201             ))
202         means = np.mean(results,axis=0)
203         stds = np.std(results,axis=0)
204         metrics = {
205             'Accuracy_mean' : means[0],'Accuracy_sd' : stds[0],
206             'Train_Accuracy_mean' : means[1],'Train_Accuracy_sd' : stds↩
                    [1],
207             'Precision_mean' : means[2],'Precision_sd' : stds[2],
208             'Sensitivity_mean' : means[3],'Sensitivity_sd' : stds[3],
209             'Specifity_mean' : means[4],'Specifity_sd' : stds[4],
210             'Time_train_mean' : means[5],'Time_train_sd' : stds[5],
211             'Time_test_mean' : means[6],'Time_test_sd' : stds[6]
212         }
213         return metrics


216     def gridsearch(self,**kwargs):
217         """
218         Grid search of hyperparameter values.

220         Parameters
221         ----------
222         **kwargs :
223             parameters passed to self.crossvalidate()

225         Returns
226         -------
227         metrics : pd.DataFrame
228             hyperparameters and corresponding metrics (mean, sd) ↩
                    resulting
229             from calls to self.crossvalidate()
230         """
231         self.metrics = pd.DataFrame(columns= [
232             'Accuracy_mean','Accuracy_sd',
233             'Train_Accuracy_mean','Train_Accuracy_sd',
234             'Precision_mean','Precision_sd',
235             'Sensitivity_mean','Sensitivity_sd',
236             'Specifity_mean','Specifity_sd',
237             'Time_train_mean','Time_train_sd',
238             'Time_test_mean','Time_test_sd'
239         ])
240         self.metrics = self.metrics.append(self.crossvalidate(**kwargs),↩
                ignore_index=True)
241         return self.metrics
```

Listing A.3: Modul `AbstractSearcher2`

```
"""
Abstract class to search hyperparameter values for models which depend on
the TF functional API.

This module requires the following packages installed:

    * `pickle`
    * `numpy`
    * `pandas`
    * `multiprocessing`
    * `sklearn.model_selection`

This module exports:

    * AbstractSearcher2
"""

import pickle
import pandas as pd
import numpy as np
from multiprocessing import Pool
from sklearn.model_selection import LeaveOneGroupOut
from src.TrainTestInterface import TrainTestInterface

class AbstractSearcher2(TrainTestInterface):
    """
    Abstract class to search hyperparameter values.

    Following methods need to be adapted to specific models:
        * self.gridsearch()
        * self.prepare_fit()
        * cls.window_function()

    Attributes
    ----------
    used : list
        indices of splits to be used for cross-validation
    logo :
        Leave One Group Out cross-validator
    X_const : np.array
        constant features
    X_var : np.array
        time-dependent features
    Y : 1d-np.array
        labels
    Split : 1d-np.array
        split indicator for cross-validation
    metrics : pd.DataFrame
        hyperparameters and corresponding aggregated metrics
    """


    def __init__(self,used):
        """
        Parameters
        ----------
        used : list
            indices of splits to be used for cross-validation
        """
        self.used = used
        self.logo = LeaveOneGroupOut()
        self.X_const = None
        self.X_var = None
        self.Y = None
```

```
65          self.Split = None
66          self.metrics = None


69      def _get_split(self,index,data_folder,extraArgs):
70          """
71          Get reshaped data of one data split.

73          Parameters
74          ----------
75          index : int
76              index of data split to use
77          data_folder : string
78              path to folder containing the data files
79          extraArgs : list
80              parameters passed to _get_reshaped_data()

82          Returns
83          -------
84          split : tuple
85              (data,X_const,X_var,Y,Split)
86          """
87          with open(data_folder+'split'+str(index)+'.bin','rb') as file:
88              data = pickle.load(file)
89              data = pd.concat([data,pd.DataFrame({'Split' : np.repeat(↩
                    index,data.shape[0])},index=data.index)],axis=1)
90          X_const,X_var,Y,Split = self._get_reshaped_data(data.reset_index↩
                ()),**extraArgs)
91          return(data,X_const,X_var,Y,Split)


94      def _get_reshaped_data(self,data,groups,feature_names,label_name,↩
            window_length,window_shape,idx_constants,idx_time_deps,↩
            labelled_only,**kwargs):
95          """
96          Get reshaped data as time windows.

98          Notes
99          -----
100         Keeps only labelled observations.

102         Parameters
103         ----------
104         data : pd.DataFrame
105             data to work on
106         groups : list
107             names of grouping columns
108         feature_names : list
109             names of features as used in data
110         label_name : string
111             column name of label
112         window_length : int
113             length of time windows used for training and classification
114         window_shape : tuple
115             shape of time window of one observation
116         idx_constants : list
117             indices of constant features
118         idx_time_deps : list
119             indices of time dependent features
120         labelled_only : bool
121             shall only labelled observations be used?
122         **kwargs :
123             additional parameters for window_function()

125         Returns
126         -------
127         data : tuple
128             (X_const,X_var,Y,Split)
```

```
129             """
130             X_const = np.empty((0,len(idx_constants)))
131             X_var = np.empty((0,)+window_shape)
132             Y = np.empty(0)
133             Split = np.empty(0)
134             names_const = [feature_names[i] for i in idx_constants]
135             names_var = [feature_names[i] for i in idx_time_deps]
136             if labelled_only:
137                 idx = data.loc[pd.notna(data[label_name]),:].index.to_numpy↩
                        ()
138             else:
139                 idx = data.index.to_numpy()
140             idx = idx[idx >= window_length-1]
141             for i in idx:
142                 grp_out = data.loc[i,groups[0]]
143                 grp_in = data.loc[i,groups[1]]
144                 data_i = data.loc[(i-window_length+1):i,:]
145                 if any(data_i[groups[0]] != grp_out) or any(data_i[groups↩
                        [1]] != grp_in):
146                     continue
147                 window = data_i.loc[(i-window_length+1):i,names_var].↩
                        to_numpy()
148                 window = self.window_function(window,**kwargs)
149                 X_var = np.concatenate((X_var,window[np.newaxis]))
150                 X_const = np.concatenate((X_const,data_i.loc[i,names_const].↩
                        to_numpy()[np.newaxis]))
151                 Y = np.concatenate((Y,data_i.loc[i,[label_name]]))
152                 Split = np.concatenate((Split,data_i.loc[i,['Split']]))
153             return (X_const.astype(float),X_var.astype(float),Y,Split)


156         def import_data(self,data_folder,n_proc,**kwargs):
157             """
158             Import data into attributes.

160             Parameters
161             ----------
162             data_folder : string
163                 path to folder containing the data files
164             n_proc : int
165                 number of processes to use
166             **kwargs :
167                 parameters for self._get_reshaped_data()

169             Returns
170             -------
171             nothing
172             """
173             arglst = [(i,data_folder,kwargs) for i in self.used]
174             with Pool(n_proc) as pool:
175                 results = pool.starmap(self._get_split, arglst)
176             results = list(zip(*results))
177             data = pd.concat(list(results[0]))
178             self.X_const = np.concatenate(results[1])
179             self.X_var = np.concatenate(results[2])
180             self.Y = np.concatenate(results[3])
181             self.Split = np.concatenate(results[4])


184         def crossvalidate(self,**kwargs):
185             """
186             Cross-validate by leaving out one split per fold.

188             Calls self.prepare_fit() per fold.

190             Parameters
191             ----------
192             **kwargs :
```

```
193             parameters passed to self.prepare_fit(),
194             except for x_train,y_train,x_test,y_test

196         Returns
197         -------
198         metrics : dict
199             Means and standard deviations of metrics
200         """
201         results = np.empty((0,7))
202         for idx_train, idx_test in self.logo.split(X=self.X_const,groups↩
                =self.Split):
203             x_const_train = np.take(self.X_const,idx_train,axis=0)
204             x_var_train = np.take(self.X_var,idx_train,axis=0)
205             x_const_test = np.take(self.X_const,idx_test,axis=0)
206             x_var_test = np.take(self.X_var,idx_test,axis=0)
207             y_train = np.take(self.Y,idx_train,axis=0)
208             y_test= np.take(self.Y,idx_test,axis=0)
209             split_result = self.prepare_fit(x_const_train,x_var_train,↩
                    y_train,x_const_test,x_var_test,y_test,**kwargs)
210             results = np.vstack((
211                 results,
212                 np.array([
213                     split_result['Accuracy'],
214                     split_result['Train_Accuracy'],
215                     split_result['Precision'],
216                     split_result['Sensitivity'],
217                     split_result['Specifity'],
218                     split_result['Time_train'],
219                     split_result['Time_test']
220                 ])
221             ))
222         means = np.mean(results, axis=0)
223         stds = np.std(results, axis=0)
224         metrics = {
225             'Accuracy_mean' : means[0], 'Accuracy_sd' : stds[0],
226             'Train_Accuracy_mean' : means[1], 'Train_Accuracy_sd' : stds↩
                    [1],
227             'Precision_mean' : means[2], 'Precision_sd' : stds[2],
228             'Sensitivity_mean' : means[3], 'Sensitivity_sd' : stds[3],
229             'Specifity_mean' : means[4], 'Specifity_sd' : stds[4],
230             'Time_train_mean' : means[5], 'Time_train_sd' : stds[5],
231             'Time_test_mean' : means[6], 'Time_test_sd' : stds[6]
232         }
233         return metrics


236     def gridsearch(self,**kwargs):
237         """
238         Grid search of hyperparameter values.

240         Parameters
241         ----------
242         **kwargs :
243             hyperparameters passed to self.crossvalidate()

245         Returns
246         -------
247         metrics : pd.DataFrame
248             hyperparameters and corresponding metrics (mean, sd)
249             resulting from calls to self.crossvalidate()
250         """
251         self.metrics = pd.DataFrame(columns= [
252             'Accuracy_mean', 'Accuracy_sd',
253             'Train_Accuracy_mean', 'Train_Accuracy_sd',
254             'Precision_mean', 'Precision_sd',
255             'Sensitivity_mean', 'Sensitivity_sd',
256             'Specifity_mean', 'Specifity_sd',
257             'Time_train_mean', 'Time_train_sd',
```

```
258             'Time_test_mean', 'Time_test_sd'
259         ])
260         self.metrics = self.metrics.append(self.crossvalidate(**kwargs),↩
                ignore_index = True)
261         return self.metrics


264     def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,↩
            x_var_test,y_test,**kwargs):
265         """
266         Prepare data and model and fit model to data.

268         Includes seeding of PRNG, preprocessing of the data, building,
269         training and testing of the model.

271         Parameters
272         ----------
273         x_const_train : np.array
274             constant features for training
275         x_var_train : np.array
276             time-dependent features for training
277         y_train : 1d-np.array
278             labels for training
279         x_const_test : np.array
280             constant features for testing
281         x_var_test : np.array
282             time-dependent features for testing
283         y_test : 1d-np.array
284             labels for testing
285         **kwargs :
286             (hyper)parameters to prepare the data or of the model

288         Returns
289         -------
290         metrics : dict
291             metrics used to evaluate the model
292         """
293         metrics = {
294             'Accuracy' : np.nan,
295             'Train_Accuracy' : np.nan,
296             'Precision' : np.nan,
297             'Sensitivity' : np.nan,
298             'Specifity' : np.nan,
299             'Time_train' : np.nan,
300             'Time_test' : np.nan
301         }
302         return metrics
```

Listing A.4: Modul `AbstractEvaluator`

```
"""
Abstract class to evaluate a model.

This module requires the following packages installed:

    * `pickle`
    * `numpy`
    * `pandas`
    * `multiprocessing`

This module exports:

    * AbstractEvaluator
"""

import pickle
import pandas as pd
import numpy as np
from multiprocessing import Pool
from src.TrainTestInterface import TrainTestInterface

class AbstractEvaluator(TrainTestInterface):
    """
    Abstract class to evaluate a model.

    Following methods need to be adapted to specific models:
        * self.prepare_fit()
        * cls.window_function()

    Attributes
    ----------
    left_out : int
        index of split used as test data
    X_train : np.array
        training features
    Y_train : 1d-np.array
        training labels
    X_test : np.array
        test features
    Y_test : 1d-np.array
        test labels
    metrics : pd.DataFrame
        test metrics
    """


    def __init__(self,left_out):
        """
        Parameters
        ----------
        left_out : int
            index of split used as test data
        """
        self.left_out = left_out
        self.X_train = None
        self.Y_train = None
        self.X_test = None
        self.Y_test = None
        self.metrics = None


    def _get_split(self,index,data_folder,extraArgs):
        """
        Get reshaped data of one data split.
```

```
66          Parameters
67          ----------
68          index : int
69              index of data split to use
70          data_folder : string
71              path to folder containing the data files
72          extraArgs : list
73              parameters passed to _get_reshaped_data()

75          Returns
76          -------
77          split : tuple
78              (data,X,Y)
79          """
80          with open(data_folder+'split'+str(index)+'.bin','rb') as file:
81              data = pickle.load(file)
82          X,Y = self._get_reshaped_data(data.reset_index(),**extraArgs)
83          return(data,X,Y)


86      def _get_reshaped_data(self,data,groups,feature_names,label_name,↩
            window_length,obs_shape,**kwargs):
87          """
88          Get reshaped data as time windows.

90          Notes
91          -----
92          Keeps only labelled observations.

94          Parameters
95          ----------
96          data : pd.DataFrame
97              data to work on
98          groups : list
99              names of grouping columns
100         feature_names : list
101             names of features as used in data
102         label_name : string
103             column name of label
104         window_length : int
105             length of time windows used for training and classification
106         obs_shape : tuple
107             shape of feature values of one observation
108         **kwargs :
109             additional parameters for window_function()

111         Returns
112         -------
113         data : tuple
114             (X,Y)
115         """
116         X = np.empty((0,)+obs_shape)
117         Y = np.empty(0)
118         idx_labelled = data.loc[pd.notna(data[label_name]),:].index.↩
                to_numpy()
119         idx_labelled = idx_labelled[idx_labelled >= window_length-1]
120         for i in idx_labelled:
121             grp_out = data.loc[i,groups[0]]
122             grp_in = data.loc[i,groups[1]]
123             data_i = data.loc[(i-window_length+1):i,:]
124             if any(data_i[groups[0]] != grp_out) or any(data_i[groups↩
                    [1]] != grp_in):
125                 continue
126             window = data_i.loc[(i-window_length+1):i,feature_names].↩
                    to_numpy()
127             window = self.window_function(window,**kwargs)
128             X = np.concatenate((X,window[np.newaxis]))
129             Y = np.concatenate((Y,data_i.loc[i,[label_name]]))
```

```
130         return (X,Y)


133     def import_data(self,data_folder,n_proc,**kwargs):
134         """
135         Import data into attributes.

137         Parameters
138         ----------
139         data_folder : string
140             path to folder containing the data files
141         n_proc : int
142             number of processes to use
143         **kwargs :
144             parameters for self._get_reshaped_data()

146         Returns
147         -------
148         nothing
149         """
150         # training data
151         train_splits = np.arange(10)
152         train_splits = np.concatenate((train_splits[:self.left_out],↩
                train_splits[self.left_out+1:]))
153         arglst = [(i,data_folder,kwargs) for i in train_splits]
154         with Pool(n_proc) as pool:
155             results = pool.starmap(self._get_split, arglst)
156         results = list(zip(*results))
157         data = pd.concat(list(results[0]))
158         self.X_train = np.concatenate(results[1])
159         self.Y_train = np.concatenate(results[2])
160         # test data
161         data = 0
162         with open(data_folder+'split'+str(self.left_out)+'.bin','rb') as↩
                file:
163                 data = pickle.load(file)
164         self.X_test,self.Y_test = self._get_reshaped_data(data.↩
                reset_index(),**kwargs)
```

Listing A.5: Modul `AbstractEvaluator2`

```
"""
Abstract class to evaluate a model depending on the TF functional API.

This module requires the following packages installed:

    * `pickle`
    * `numpy`
    * `pandas`
    * `multiprocessing`

This module exports:

    * AbstractEvaluator2
"""

import pickle
import pandas as pd
import numpy as np
from multiprocessing import Pool
from src.TrainTestInterface import TrainTestInterface

class AbstractEvaluator2(TrainTestInterface):
    """
    Abstract class to evaluate a model.

    Following methods need to be adapted to specific models:
        * self.prepare_fit()
        * cls.window_function()

    Attributes
    ----------
    left_out : int
        index of split used as test data
    X_const_train : np.array
        constant training features
    X_var_train : np.array
        time-depending training features
    Y_train : 1d-np.array
        training labels
    X_const_test : np.array
        constant test features
    X_var_test : np.array
        time-depending test features
    Y_test : 1d-np.array
        test labels
    metrics : pd.DataFrame
        test metrics
    """


    def __init__(self,left_out):
        """
        Parameters
        ----------
        left_out : int
            index of split used as test data
        """
        self.left_out = left_out
        self.X_const_train = None
        self.X_var_train = None
        self.Y_train = None
        self.X_const_test = None
        self.X_var_test = None
        self.Y_test = None
        self.metrics = None
```

```
68      def _get_split(self,index,data_folder,extraArgs):
69          """
70          Get reshaped data of one data split.

72          Parameters
73          ----------
74          index : int
75              index of data split to use
76          data_folder : string
77              path to folder containing the data files
78          extraArgs : list
79              parameters passed to _get_reshaped_data()

81          Returns
82          -------
83          split : tuple
84              (data,X,Y)
85          """
86          with open(data_folder+'split'+str(index)+'.bin','rb') as file:
87              data = pickle.load(file)
88          X_const,X_var,Y = self._get_reshaped_data(data.reset_index(),**↩
                extraArgs)
89          return(data,X_const,X_var,Y)


92      def _get_reshaped_data(self,data,groups,feature_names,label_name,
93              window_length,window_shape,idx_constants,idx_time_depst,
94              labelled_only,):
95          """
96          Get reshaped data as time windows.

98          Parameters
99          ----------
100         data : pd.DataFrame
101             data to work on
102         groups : list
103             names of grouping columns
104         feature_names : list
105             names of features as used in data
106         label_name : string
107             column name of label
108         window_length : int
109             length of time windows used for training and classification
110         window_shape : tuple
111             shape of time window of one observation
112         idx_constants : list
113             indices of constant features
114         idx_time_deps : list
115             indices of time dependent features
116         labelled_only : bool
117             shall only labelled observations be used?
118         **kwargs :
119             additional parameters for window_function()

121         Returns
122         -------
123         data : tuple
124             (X_const,X_var,Y)
125         """
126         X_const = np.empty((0,len(idx_constants)))
127         X_var = np.empty((0,)+window_shape)
128         Y = np.empty(0)
129         names_const = [feature_names[i] for i in idx_constants]
130         names_var = [feature_names[i] for i in idx_time_deps]
131         if labelled_only:
```

```
132             idx = data.loc[pd.notna(data[label_name]),:].index.to_numpy↩
                    ()
133         else:
134             idx = data.index.to_numpy()
135         idx = idx[idx >= window_length-1]
136         for i in idx:
137             grp_out = data.loc[i,groups[0]]
138             grp_in = data.loc[i,groups[1]]
139             data_i = data.loc[(i-window_length+1):i,:]
140             if any(data_i[groups[0]] != grp_out) or any(data_i[groups↩
                    [1]] != grp_in):
141                 continue
142             window = data_i.loc[(i-window_length+1):i,names_var].↩
                    to_numpy()
143             window = self.window_function(window,**kwargs)
144             X_var = np.concatenate((X_var,window[np.newaxis]))
145             X_const = np.concatenate((X_const,data_i.loc[i,names_const].↩
                    to_numpy()[np.newaxis]))
146             Y = np.concatenate((Y,data_i.loc[i,[label_name]]))
147         return (X_const.astype(float),X_var.astype(float),Y)


150     def import_data(self,data_folder,n_proc,**kwargs):
151         """
152         Import data into attributes.

154         Parameters
155         ----------
156         data_folder : string
157             path to folder containing the data files
158         n_proc : int
159             number of processes to use
160         **kwargs :
161             parameters for self._get_reshaped_data()

163         Returns
164         -------
165         nothing
166         """
167         # training data
168         train_splits = np.arange(10)
169         train_splits = np.concatenate((train_splits[:self.left_out],↩
                train_splits[self.left_out+1:]))
170         arglst = [(i,data_folder,kwargs) for i in train_splits]
171         with Pool(n_proc) as pool:
172             results = pool.starmap(self._get_split,arglst)
173         results = list(zip(*results))
174         data = pd.concat(list(results[0]))
175         self.X_const_train = np.concatenate(results[1])
176         self.X_var_train = np.concatenate(results[2])
177         self.Y_train = np.concatenate(results[3])
178         # test data
179         data = 0
180         with open(data_folder+'split'+str(self.left_out)+'.bin','rb') as↩
                file:
181             data = pickle.load(file)
182         self.X_const_test,self.X_var_test,self.Y_test = self.↩
                _get_reshaped_data(data.reset_index(),**kwargs)
```

Listing A.6: Modul `winfunc`

```
"""
Functions to transform time window data.

These functions can be used wrapped in
`TrainTestInterface.window_function()` when the interface is implemented
for specific machine learning approaches.

This module requires the following packages installed:

    * `numpy`

This module exports:

    * add_spectra_and_sd()
"""

import numpy as np


def add_spectra_and_sd(window_data,idx_constants,idx_time_deps):
    """
    Add spectra and standard deviations of columns in `window_data`.

    Transformation of time window data for feature engineering approach.

    Parameters
    ----------
    window_data : np.array
        data of one time window with shape=[time steps,features]
    idx_constants : list
        indices of constant features
    idx_time_deps : list
        indices of time dependent features

    Returns
    -------
    transformed : np.array (1d)
        the transformed and flattened window data
    """
    time_deps = window_data[:,idx_time_deps]
    spectra = np.abs(np.fft.fftshift(np.fft.fft(time_deps,axis=0)))
    stdevs = np.std(time_deps,axis=0)
    constants = window_data[0,idx_constants]
    transformed = np.hstack((time_deps,spectra)).flatten()
    transformed = np.hstack((constants,transformed,stdevs))
    return transformed
```

A.2 Ausführbare Skripte — Ansätze mit Feature Engineering

Listing A.7: Skript rf_search.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ----------------------
import time
import numpy as np
import pandas as pd
import sklearn as sk
from sklearn.model_selection import LeaveOneGroupOut
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractSearcher import AbstractSearcher
from src.winfunc import add_spectra_and_sd

# ----------------------
# Define specific class
# ----------------------
class RfSearcher(AbstractSearcher):

    def __init__(self,used,n_estimators_values,min_samples_leaf_values):
        self.used = used
        self.n_estimators_values = n_estimators_values
        self.min_samples_leaf_values = min_samples_leaf_values
        self.logo = LeaveOneGroupOut()
        self.X = None
        self.Y = None
        self.Split = None
        self.metrics = None

    @classmethod
    def window_function(cls,window_data,**kwargs):
        transformed = add_spectra_and_sd(window_data,kwargs['idx_constants'],kwargs['idx_time_deps'])
        return transformed

    def gridsearch(self,**kwargs):
        self.metrics = pd.DataFrame(columns= [
            'n_estimators','min_samples_leaf',
            'Accuracy_mean','Accuracy_sd',
            'Precision_mean','Precision_sd',
            'Sensitivity_mean','Sensitivity_sd',
            'Specifity_mean','Specifity_sd',
            'Time_train_mean','Time_train_sd',
            'Time_test_mean','Time_test_sd'
        ])
        for n_estimators in self.n_estimators_values:
            for min_samples_leaf in self.min_samples_leaf_values:
                cv_result = self.crossvalidate(n_estimators=n_estimators,min_samples_leaf=min_samples_leaf,n_jobs=n_jobs)
                cv_result['n_estimators'] = n_estimators
                cv_result['min_samples_leaf'] = min_samples_leaf
                self.metrics = self.metrics.append(cv_result,ignore_index=True)
        self.metrics = self.metrics.sort_values(by=['Accuracy_mean'],ascending=False)
        return self.metrics

    def prepare_fit(self,x_train,y_train,x_test,y_test,n_estimators,min_samples_leaf,n_jobs):
        metrics = {
            'Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
            'Specifity' : np.nan,
```

```
60              'Time_train' : np.nan,
61              'Time_test' : np.nan
62          }
63          rf_cl = RandomForestClassifier(max_features='sqrt', random_state↩
                =42, n_jobs=n_jobs,n_estimators=n_estimators, ↩
                min_samples_leaf=min_samples_leaf)
64          y_train_ = y_train.astype(int)
65          y_test_ = y_test.astype(int)
66          # train model
67          time_start = time.time()
68          rf_cl.fit(x_train,y_train_)
69          metrics['Time_train'] = time.time()-time_start
70          pred_train = rf_cl.predict(x_train)
71          metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
72          # test model
73          time_start = time.time()
74          pred_test = rf_cl.predict(x_test)
75          metrics['Time_test'] = time.time()-time_start
76          # score
77          metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
78          metrics['Precision'] = precision_score(y_test_,pred_test)
79          metrics['Sensitivity'] = recall_score(y_test_,pred_test)
80          cm = confusion_matrix(y_test_,pred_test)
81          metrics['Specifity'] = cm[0,0]/(cm[0,1]+cm[0,0])
82          return metrics

84  # ----------
85  # Run search
86  # ----------
87  n_jobs = 5
88  used = [1,4,6,9]
89  groups = ['CowId','Lactation']
90  feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
91  label_name = 'Lscore'
92  window_length = 27
93  obs_shape= (167,)
94  n_estimators_values = [3000,3500,4000,4500,5000]
95  min_samples_leaf_values = [1,5,10,20]
96  name = 'fe-rf_search'
97  ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
98  output = name+'_'+ts
99  data_folder = '../data/'

101 classifier = RfSearcher(used,n_estimators_values,min_samples_leaf_values↩
        )
102 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        obs_shape=obs_shape,idx_constants=[0,1],idx_time_deps=[2,3,4])
103 results = classifier.gridsearch(n_jobs=n_jobs)

105 results.to_csv(output+'.csv')
```

Listing A.8: Skript rf_eval.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import sys
import time
import pickle
import numpy as np
import pandas as pd
import sklearn as sk
from datetime import datetime
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractEvaluator import AbstractEvaluator
from src.winfunc import add_spectra_and_sd

# ---------------------
# Define specific class
# ---------------------
class RfEvaluator(AbstractEvaluator):

    def __init__(self,left_out,n_estimators,min_samples_leaf):
        self.left_out = left_out
        self.n_estimators=n_estimators
        self.min_samples_leaf=min_samples_leaf
        self.X_train = None
        self.Y_train = None
        self.X_test = None
        self.Y_test = None
        self.metrics = None

    @classmethod
    def window_function(cls,window_data,**kwargs):
        transformed = add_spectra_and_sd(window_data,kwargs['idx_constants'],kwargs['idx_time_deps'])
        return transformed

    def prepare_fit(self,x_train,y_train,x_test,y_test,n_estimators,min_samples_leaf,n_jobs):
        self.metrics = pd.DataFrame(columns= [
            'n_estimators','min_samples_leaf',
            'Accuracy',
            'Precision',
            'Sensitivity',
            'Specifity',
            'Time_train',
            'Time_test'
        ])
        metrics = {
            'Accuracy' : np.nan,
            'Train_Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
            'Specifity' : np.nan,
            'Time_train' : np.nan,
            'Time_test' : np.nan
        }
        # build model
        self.model = RandomForestClassifier(max_features='sqrt',random_state=42,n_jobs=n_jobs,n_estimators=n_estimators,min_samples_leaf=min_samples_leaf)
        y_train_ = y_train.astype(int)
        y_test_ = y_test.astype(int)
        # train model
```

```
61          time_start = datetime.now()
62          self.model.fit(x_train,y_train_)
63          metrics['Time_train'] = datetime.now()-time_start
64          pred_train = self.model.predict(x_train)
65          metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
66          # test model
67          time_start = datetime.now()
68          pred_test = self.model.predict(x_test)
69          metrics['Time_test'] = datetime.now()-time_start
70          # score
71          metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
72          metrics['Precision'] = precision_score(y_test_,pred_test)
73          metrics['Sensitivity'] = recall_score(y_test_,pred_test)
74          cm = confusion_matrix(y_test_,pred_test)
75          metrics['Specifity'] = cm[0,0]/(cm[0,1]+cm[0,0])
76          result = metrics.copy()
77          result['n_estimators'] = n_estimators
78          result['min_samples_leaf'] = min_samples_leaf
79          self.metrics = self.metrics.append(result,ignore_index=True)
80          return metrics

82  # --------------
83  # Run evaluation
84  # --------------
85  n_jobs=5
86  left_out = int(sys.argv[1])
87  groups = ['CowId','Lactation']
88  feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
89  label_name = 'Lscore'
90  window_length = 27
91  obs_shape = (167,)
92  n_estimators = 4500
93  min_samples_leaf = 5
94  name = 'fe-rf_eval'+str(left_out)
95  ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
96  output = name+'_'+ts
97  data_folder = '../data/'

99  classifier = FeRfEvaluator(left_out,n_estimators,min_samples_leaf)
100 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        obs_shape=obs_shape,idx_constants=[0,1],idx_time_deps=[2,3,4])
101 result = classifier.prepare_fit(x_train=classifier.X_train,y_train=↩
        classifier.Y_train,x_test=classifier.X_test,y_test=classifier.Y_test,↩
        n_estimators=n_estimators,min_samples_leaf=min_samples_leaf,n_jobs=↩
        n_jobs)

103 print(f"\nEvaluate␣FE-RF␣on␣left_out={left_out}␣with␣parameters:\n↩
        ----------------------\n\n")
104 print(f"n_estimators␣=␣{n_estimators}")
105 print(f"min_samples_leaf␣=␣{min_samples_leaf}")
106 print('\n-----\n')
107 print(f"Output:␣{output}")
108 print('\n-----\n')
109 print(f"Accuracy␣:␣{result['Accuracy']}␣(Training␣accuracy␣:␣{result['↩
        Train_Accuracy']})")
110 print(f"Precision␣:␣{result['Precision']}")
111 print(f"Sensitivity␣:␣{result['Sensitivity']}")
112 print(f"Specifity␣:␣{result['Specifity']}")
113 print(f"Time_train␣:␣{result['Time_train']}")
114 print(f"Time_test␣:␣{result['Time_test']}")
```

Listing A.9: Skript svm_search.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import time
import numpy as np
import pandas as pd
import sklearn as sk
from sklearn.model_selection import LeaveOneGroupOut
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractSearcher import AbstractSearcher
from src.winfunc import add_spectra_and_sd

# ---------------------
# Define specific class
# ---------------------
class SvmSearcher(AbstractSearcher):

    def __init__(self,used,kernel_values,C_values):
        self.used = used
        self.kernel_values = kernel_values
        self.C_values = C_values
        self.logo = LeaveOneGroupOut()
        self.X = None
        self.Y = None
        self.Split = None
        self.metrics = None

    @classmethod
    def window_function(cls,window_data,**kwargs):
        transformed = add_spectra_and_sd(window_data,kwargs['idx_constants'],kwargs['idx_time_deps'])
        return transformed

    def gridsearch(self,**kwargs):
        self.metrics = pd.DataFrame(columns= [
            'kernel','C',
            'Accuracy_mean','Accuracy_sd',
            'Precision_mean','Precision_sd',
            'Sensitivity_mean','Sensitivity_sd',
            'Specifity_mean','Specifity_sd',
            'Time_train_mean','Time_train_sd',
            'Time_test_mean','Time_test_sd'
        ])
        for kernel in self.kernel_values:
            for C in self.C_values:
                cv_result = self.crossvalidate(kernel=kernel,C=C,n_jobs=n_jobs)
                cv_result['kernel'] = kernel
                cv_result['C'] = C
                self.metrics = self.metrics.append(cv_result,ignore_index=True)
        self.metrics = self.metrics.sort_values(by=['Accuracy_mean'],ascending=False)
        return self.metrics

    def prepare_fit(self,x_train,y_train,x_test,y_test,kernel, C,n_jobs):
        metrics = {
            'Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
```

```
60              'Specifity' : np.nan,
61              'Time_train' : np.nan,
62              'Time_test' : np.nan
63          }
64          stdsca = StandardScaler()
65          svm_cl = SVC(degree=3,gamma='scale',random_state=42,coef0=0,↩
                cache_size=1000,kernel=kernel, C=C)
66          y_train_ = y_train.astype(int)
67          y_test_ = y_test.astype(int)
68          x_train_ = stdsca.fit_transform(x_train)
69          x_test_ = stdsca.transform(x_test)
70          # train model
71          time_start = time.time()
72          svm_cl.fit(x_train_,y_train_)
73          metrics['Time_train'] = time.time()-time_start
74          pred_train = svm_cl.predict(x_train_)
75          metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
76          # test model
77          time_start = time.time()
78          pred_test = svm_cl.predict(x_test_)
79          metrics['Time_test'] = time.time()-time_start
80          # score
81          metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
82          metrics['Precision'] = precision_score(y_test_,pred_test)
83          metrics['Sensitivity'] = recall_score(y_test_,pred_test)
84          cm = confusion_matrix(y_test_,pred_test)
85          metrics['Specifity'] = cm[0,0]/(cm[0,1]+cm[0,0])
86          return metrics

88  # ----------
89  # Run search
90  # ----------
91  n_jobs = 4
92  used = [1,4,6,9]
93  groups = ['CowId','Lactation']
94  feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
95  label_name = 'Lscore'
96  window_length = 27
97  obs_shape= (167,)
98  kernel_values = ['rbf']
99  C_values = [0.01,0.1,0.5,1,1.5,3,5,10,100]
100 name = 'fe-svm_search'
101 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
102 output = name+'_'+ts
103 data_folder = '../data/'

105 classifier = SvmSearcher(used,kernel_values,C_values)
106 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        obs_shape=obs_shape,idx_constants=[0,1],idx_time_deps=[2,3,4])
107 results = classifier.gridsearch(n_jobs=n_jobs)

109 results.to_csv(output+'.csv')
```

Listing A.10: Skript svm_eval.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ----------------------
import sys
import time
import pickle
import numpy as np
import pandas as pd
import sklearn as sk
from datetime import datetime
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractEvaluator import AbstractEvaluator
from src.winfunc import add_spectra_and_sd

# ----------------------
# Define specific class
# ----------------------
class SvmEvaluator(AbstractEvaluator):

    def __init__(self,left_out,kernel,C):
        self.left_out = left_out
        self.kernel=kernel
        self.C=C
        self.X_train = None
        self.Y_train = None
        self.X_test = None
        self.Y_test = None
        self.metrics = None

    @classmethod
    def window_function(cls,window_data,**kwargs):
        transformed = add_spectra_and_sd(window_data,kwargs['idx_constants'],kwargs['idx_time_deps'])
        return transformed

    def prepare_fit(self,x_train,y_train,x_test,y_test,kernel,C,n_jobs):
        self.metrics = pd.DataFrame(columns= [
            'kernel','C',
            'Accuracy',
            'Train_Accuracy',
            'Precision',
            'Sensitivity',
            'Specifity',
            'Time_train',
            'Time_test'
        ])
        metrics = {
            'Accuracy' : np.nan,
            'Train_Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
            'Specifity' : np.nan,
            'Time_train' : np.nan,
            'Time_test' : np.nan
        }
        stdsca = StandardScaler()
        y_train_ = y_train.astype(int)
        y_test_ = y_test.astype(int)
        x_train_ = stdsca.fit_transform(x_train)
        x_test_ = stdsca.transform(x_test)
        # build model
```

```
64          self.model = SVC(degree=3,gamma='scale',random_state=42,coef0=0,↩
                cache_size=1000,kernel=kernel,C=C)
65          # train model
66          time_start = datetime.now()
67          self.model.fit(x_train_,y_train_)
68          metrics['Time_train'] = datetime.now()-time_start
69          pred_train = self.model.predict(x_train_)
70          metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
71          # test model
72          time_start = datetime.now()
73          pred_test = self.model.predict(x_test_)
74          metrics['Time_test'] = datetime.now()-time_start
75          # score
76          metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
77          metrics['Precision'] = precision_score(y_test_,pred_test)
78          metrics['Sensitivity'] = recall_score(y_test_,pred_test)
79          cm = confusion_matrix(y_test_,pred_test)
80          metrics['Specifity'] = cm[0,0]/(cm[0,1]+cm[0,0])
81          result = metrics.copy()
82          result['kernel'] = kernel
83          result['C'] = C
84          self.metrics = self.metrics.append(result,ignore_index=True)
85          return metrics

87  # --------------
88  # Run evaluation
89  # --------------
90  n_jobs=4
91  left_out = int(sys.argv[1])
92  groups = ['CowId','Lactation']
93  feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
94  label_name = 'Lscore'
95  window_length = 27
96  obs_shape = (167,)
97  kernel = 'rbf'
98  C = 0.5
99  name = 'fe-svm_eval'+str(left_out)
100 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
101 output = name+'_'+ts
102 data_folder = '../data/'

104 classifier = FeSvmEvaluator(left_out,kernel,C)
105 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        obs_shape=obs_shape,idx_constants=[0,1],idx_time_deps=[2,3,4])
106 result = classifier.prepare_fit(x_train=classifier.X_train,y_train=↩
        classifier.Y_train,x_test=classifier.X_test,y_test=classifier.Y_test,↩
        kernel=kernel,C=C,n_jobs=n_jobs)

108 print(f"\nEvaluate␣FE-SVM␣on␣left_out={left_out}␣with␣parameters:\n↩
        ----------------------\n\n")
109 print(f"kernel␣=␣{kernel}")
110 print(f"C␣=␣{C}")
111 print('\n-----\n')
112 print(f"Output:␣{output}")
113 print('\n-----\n')
114 print(f"Accuracy␣:␣{result['Accuracy']}␣(Training␣accuracy␣:␣{result['↩
        Train_Accuracy']})")
115 print(f"Precision␣:␣{result['Precision']}")
116 print(f"Sensitivity␣:␣{result['Sensitivity']}")
117 print(f"Specifity␣:␣{result['Specifity']}")
118 print(f"Time_train␣:␣{result['Time_train']}")
119 print(f"Time_test␣:␣{result['Time_test']}")
```

Listing A.11: Skript fe-mlp_search.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import os
import time
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import LeaveOneGroupOut
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractSearcher import AbstractSearcher
from src.winfunc import add_spectra_and_sd

# ---------------------
# Define specific class
# ---------------------
class FeMlpSearcher(AbstractSearcher):

    def __init__(self,used,n_neurons_1st,n_neurons_other,n_layers_other,learning_rate_values,batch_size_values):
        self.used = used
        self.n_neurons_1st = n_neurons_1st
        self.n_neurons_other = n_neurons_other
        self.n_layers_other = n_layers_other
        self.learning_rate_values = learning_rate_values
        self.batch_size_values = batch_size_values
        self.logo = LeaveOneGroupOut()
        self.X = None
        self.Y = None
        self.Split = None
        self.metrics = None

    @classmethod
    def window_function(cls,window_data,**kwargs):
        transformed = add_spectra_and_sd(window_data,kwargs['idx_constants'],kwargs['idx_time_deps'])
        return transformed

    def gridsearch(self,**kwargs):
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy_mean', 'Accuracy_sd',
            'Precision_mean', 'Precision_sd',
            'Sensitivity_mean', 'Sensitivity_sd',
            'Specifity_mean', 'Specifity_sd',
            'Time_train_mean', 'Time_train_sd',
            'Time_test_mean', 'Time_test_sd'
        ])
        for learning_rate in self.learning_rate_values:
            for batch_size in self.batch_size_values:
                cv_result = self.crossvalidate(learning_rate=learning_rate,batch_size=batch_size,log_folder=log_folder)
                cv_result['learning_rate'] = learning_rate
                cv_result['batch_size'] = batch_size
                self.metrics = self.metrics.append(cv_result,ignore_index=True)
        self.metrics = self.metrics.sort_values(by=['Accuracy_mean'],ascending=False)
        return self.metrics
```

```
58      def prepare_fit(self,x_train,y_train,x_test,y_test,learning_rate,↩
            batch_size,log_folder):
59          y_train_ = y_train.astype(int)
60          y_test_ = y_test.astype(int)
61          metrics = {
62              'Accuracy' : np.nan,
63              'Precision' : np.nan,
64              'Sensitivity' : np.nan,
65              'Specifity' : np.nan,
66              'Time_train' : np.nan,
67              'Time_test' : np.nan
68          }
69          kernel_initializer = 'lecun_normal'
70          activation = 'selu'
71          n_neurons_1st = self.n_neurons_1st
72          n_neurons_other = self.n_neurons_other
73          n_layers_other = self.n_layers_other
74          n_epochs = 1000
75          # set random seed for reproducible results
76          tf.random.set_seed(42)
77          # data normalization layer
78          norm_layer = keras.layers.experimental.preprocessing.↩
                Normalization()
79          norm_layer.adapt(x_train)
80          # build model
81          model = keras.models.Sequential()
82          model.add(keras.layers.InputLayer(input_shape=x_train.shape[1:])↩
                )
83          model.add(norm_layer)
84          model.add(keras.layers.Dense(
85              n_neurons_1st,
86              kernel_initializer=kernel_initializer,
87              activation=activation
88          ))
89          for layer in range(n_layers_other):
90              model.add(keras.layers.Dense(
91                  n_neurons_other,
92                  kernel_initializer=kernel_initializer,
93                  activation=activation
94              ))
95          model.add(keras.layers.Dense(1, activation='sigmoid'))
96          model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
97          run_log_folder = 'N' + str(y_train.shape[0])
98          run_log_folder = os.path.join(log_folder,run_log_folder)
99          # train model
100         time_start = time.time()
101         model.fit(
102             x_train,y_train_,
103             validation_data=(x_test,y_test_),
104             batch_size=batch_size,epochs=n_epochs,
105             callbacks = [
106                 keras.callbacks.TensorBoard(run_log_folder),
107                 keras.callbacks.EarlyStopping(patience=5,↩
                        restore_best_weights=True)
108             ],
109             verbose=0
110         )
111         metrics['Time_train'] = time.time()-time_start
112         pred_train = (model.predict(x_train) > 0.5).astype(int)
113         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
114         # test model
115         time_start = time.time()
116         pred_test = (model.predict(x_test) > 0.5).astype(int)
117         metrics['Time_test'] = time.time()-time_start
118         # score
119         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
```

```
120         metrics['Precision'] = precision_score(y_test_,pred_test)
121         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
122         cm = confusion_matrix(y_test_,pred_test)
123         metrics['Specifity'] = cm[0,0]/(cm[0,1]+cm[0,0])
124         return metrics

126 # ----------
127 # Run search
128 # ----------
129 used = [1,4,6,9]
130 groups = ['CowId','Lactation']
131 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
132 label_name = 'Lscore'
133 window_length = 27
134 obs_shape= (167,)
135 n_neurons_1st = 6
136 n_neurons_other = 4
137 n_layers_other = 2
138 learning_rate_values = [0.0005,0.001,0.005,0.01,0.05]
139 batch_size_values = [8,16,32,64,128]
140 name = 'fe-mlp_search'
141 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
142 output = name+'_'+ts
143 data_folder = '../data/'
144 log_folder = '../tb_log/'
145 log_folder = os.path.join(log_folder,output)

147 classifier = FeMlpSearcher(used,n_neurons_1st,n_neurons_other,↩
        n_layers_other,learning_rate_values,batch_size_values)
148 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        obs_shape=obs_shape,idx_constants=[0,1],idx_time_deps=[2,3,4])
149 results = classifier.gridsearch(log_folder=log_folder)

151 results.to_csv(output+'.csv')
```

Listing A.12: Skript fe-mlp_eval.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import sys
import os
import time
import pickle
import numpy as np
import pandas as pd
import tensorflow as tf
from datetime import datetime
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractEvaluator import AbstractEvaluator
from src.winfunc import add_spectra_and_sd

# ---------------------
# Define specific class
# ---------------------
class FeMlpEvaluator(AbstractEvaluator):

    def __init__(self,left_ou,n_neurons_1st,n_neurons_other,n_layers_other,learning_rate,batch_size):
        self.left_out = left_out
        self.n_neurons_1st = n_neurons_1st
        self.n_neurons_other = n_neurons_other
        self.n_layers_other = n_layers_other
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.X_train = None
        self.Y_train = None
        self.X_test = None
        self.Y_test = None
        self.metrics = None

    @classmethod
    def window_function(cls,window_data,**kwargs):
        transformed = add_spectra_and_sd(window_data,kwargs['idx_constants'],kwargs['idx_time_deps'])
        return transformed

    def prepare_fit(self,x_train,y_train,x_test,y_test,learning_rate,batch_size,log_folder):
        y_train_ = y_train.astype(int)
        y_test_ = y_test.astype(int)
        self.metrics = pd.DataFrame(columns= [
            'learning_rate','batch_size',
            'Accuracy',
            'Train_Accuracy',
            'Precision',
            'Sensitivity',
            'Specifity',
            'Time_train',
            'Time_test'
        ])
        metrics = {
            'Accuracy' : np.nan,
            'Train_Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
            'Specifity' : np.nan,
            'Time_train' : np.nan,
            'Time_test' : np.nan
```

```
62          }
63          kernel_initializer = 'lecun_normal'
64          activation = 'selu'
65          n_neurons_1st = self.n_neurons_1st
66          n_neurons_other = self.n_neurons_other
67          n_layers_other = self.n_layers_other
68          n_epochs = 1000
69          # set random seed for reproducible results
70          tf.random.set_seed(42)
71          # data normalization layer
72          norm_layer = keras.layers.experimental.preprocessing.↩
                Normalization()
73          norm_layer.adapt(x_train)
74          # build model
75          model = keras.models.Sequential()
76          model.add(keras.layers.InputLayer(input_shape=x_train.shape[1:])↩
                )
77          model.add(norm_layer)
78          model.add(keras.layers.Dense(
79              n_neurons_1st,
80              kernel_initializer=kernel_initializer,
81              activation=activation
82          ))
83          for layer in range(n_layers_other):
84              model.add(keras.layers.Dense(
85                  n_neurons_other,
86                  kernel_initializer=kernel_initializer,
87                  activation=activation
88              ))
89          model.add(keras.layers.Dense(1, activation='sigmoid'))
90          model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
91          run_log_folder = 'N' + str(y_train.shape[0])
92          run_log_folder = os.path.join(log_folder, run_log_folder)
93          # train model
94          time_start = datetime.now()
95          model.fit(
96              x_train,y_train_,
97              validation_data=(x_test,y_test_),
98              batch_size=batch_size,epochs=n_epochs,
99              callbacks = [
100                 keras.callbacks.TensorBoard(run_log_folder),
101                 keras.callbacks.EarlyStopping(patience=5,↩
                        restore_best_weights=True)
102             ],
103             verbose=0
104         )
105         metrics['Time_train'] = datetime.now()-time_start
106         pred_train = (model.predict(x_train) > 0.5).astype(int)
107         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
108         # test model
109         time_start = datetime.now()
110         pred_test = (model.predict(x_test) > 0.5).astype(int)
111         metrics['Time_test'] = datetime.now()-time_start
112         # score
113         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
114         metrics['Precision'] = precision_score(y_test_,pred_test)
115         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
116         cm = confusion_matrix(y_test_,pred_test)
117         metrics['Specifity'] = cm[0,0]/(cm[0,1]+cm[0,0])
118         result = metrics.copy()
119         result['learning_rate'] = learning_rate
120         result['batch_size'] = batch_size
121         self.metrics = self.metrics.append(result,ignore_index=True)
122         return metrics

124 # --------------
```

```
125 # Run evaluation
126 # --------------
127 left_out = int(sys.argv[1])
128 groups = ['CowId','Lactation']
129 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
130 label_name = 'Lscore'
131 window_length = 27
132 obs_shape = (167,)
133 n_neurons_1st = 6
134 n_neurons_other = 4
135 n_layers_other = 2
136 learning_rate = 0.0005
137 batch_size = 32
138 name = 'fe-mlp_eval'+str(left_out)
139 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
140 output = name+'_'+ts
141 data_folder = '../data/'
142 log_folder = '../tb_log/'
143 log_folder = os.path.join(log_folder, output)

145 classifier = FeMlpEvaluator(left_out,n_neurons_1st,n_neurons_other,↩
        n_layers_other,learning_rate,batch_size)
146 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        obs_shape=obs_shape,idx_constants=[0,1],idx_time_deps=[2,3,4])
147 result = classifier.prepare_fit(x_train=classifier.X_train,y_train=↩
        classifier.Y_train,x_test=classifier.X_test,y_test=classifier.Y_test,↩
        learning_rate=learning_rate,batch_size=batch_size,log_folder=↩
        log_folder)

149 print(f"\nEvaluate␣FE-MLP␣on␣left_out={left_out}␣with␣parameters:\n↩
        ----------------------\n\n")
150 print(f"learning_rate␣=␣{learning_rate}")
151 print(f"batch_size␣=␣{batch_size}")
152 print('\n-----\n')
153 print(f"Output:␣{output}")
154 print('\n-----\n')
155 print(f"Accuracy␣:␣{result['Accuracy']}␣(Training␣accuracy␣:␣{result['↩
        Train_Accuracy']})")
156 print(f"Precision␣:␣{result['Precision']}")
157 print(f"Sensitivity␣:␣{result['Sensitivity']}")
158 print(f"Specifity␣:␣{result['Specifity']}")
159 print(f"Time_train␣:␣{result['Time_train']}")
160 print(f"Time_test␣:␣{result['Time_test']}")
```

A.3 Ausführbare Skripte — Ende-zu-Ende-Ansätze

Listing A.13: Skript e2e-mlp_search.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import os
import time
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import LeaveOneGroupOut
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractSearcher2 import AbstractSearcher2

# ---------------------
# Define specific class
# ---------------------
class E2eMlpSearcher(AbstractSearcher2):

    def __init__(self,used,n_neurons_1st,n_neurons_other,n_layers_other,learning_rate_values,batch_size_values):
        self.used = used
        self.n_neurons_1st = n_neurons_1st
        self.n_neurons_other = n_neurons_other
        self.n_layers_other = n_layers_other
        self.learning_rate_values = learning_rate_values
        self.batch_size_values = batch_size_values
        self.logo = LeaveOneGroupOut()
        self.X_const = None
        self.X_var = None
        self.Y = None
        self.Split = None
        self.metrics = None

    @classmethod
    def window_function(cls,window_data,**kwargs):
        return window_data.flatten()

    def gridsearch(self,**kwargs):
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy_mean', 'Accuracy_sd',
            'Precision_mean', 'Precision_sd',
            'Sensitivity_mean', 'Sensitivity_sd',
            'Specifity_mean', 'Specifity_sd',
            'Time_train_mean', 'Time_train_sd',
            'Time_test_mean', 'Time_test_sd'
        ])
        for learning_rate in self.learning_rate_values:
            for batch_size in self.batch_size_values:
                cv_result = self.crossvalidate(learning_rate=learning_rate,batch_size=batch_size,log_folder=log_folder)
                cv_result['learning_rate'] = learning_rate
                cv_result['batch_size'] = batch_size
                self.metrics = self.metrics.append(cv_result,ignore_index=True)
        self.metrics = self.metrics.sort_values(by=['Accuracy_mean'],ascending=False)
        return self.metrics

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,x_var_test,y_test,learning_rate,batch_size,log_folder):
        x_train = np.hstack((x_const_train,x_var_train))
```

```
59          x_test = np.hstack((x_const_test,x_var_test))
60          y_train_ = y_train.astype(int)
61          y_test_ = y_test.astype(int)
62          metrics = {
63              'Accuracy' : np.nan,
64              'Precision' : np.nan,
65              'Sensitivity' : np.nan,
66              'Specifity' : np.nan,
67              'Time_train' : np.nan,
68              'Time_test' : np.nan
69          }
70          kernel_initializer = 'lecun_normal'
71          activation = 'selu'
72          n_neurons_1st = self.n_neurons_1st
73          n_neurons_other = self.n_neurons_other
74          n_layers_other = self.n_layers_other
75          n_epochs = 1000
76          # set random seed for reproducible results
77          tf.random.set_seed(42)
78          # data normalization layer
79          norm_layer = keras.layers.experimental.preprocessing.↩
                Normalization()
80          norm_layer.adapt(x_train)
81          # build model
82          model = keras.models.Sequential()
83          model.add(keras.layers.InputLayer(input_shape=x_train.shape[1:])↩
                )
84          model.add(norm_layer)
85          model.add(keras.layers.Dense(
86              n_neurons_1st,
87              kernel_initializer=kernel_initializer,
88              activation=activation
89          ))
90          for layer in range(n_layers_other):
91              model.add(keras.layers.Dense(
92                  n_neurons_other,
93                  kernel_initializer=kernel_initializer,
94                  activation=activation
95              ))
96          model.add(keras.layers.Dense(1,activation='sigmoid'))
97          model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
98          run_log_folder = 'N'+str(y_train.shape[0])
99          run_log_folder = os.path.join(log_folder,run_log_folder)
100         # train model
101         time_start = time.time()
102         model.fit(
103             x_train,y_train_,
104             validation_data=(x_test,y_test_),
105             batch_size=batch_size,epochs=n_epochs,
106             callbacks = [
107                 keras.callbacks.TensorBoard(run_log_folder),
108                 keras.callbacks.ReduceLROnPlateau(patience=10,min_delta↩
                        =0.00001),
109                 keras.callbacks.EarlyStopping(patience=20,↩
                        restore_best_weights=True)
110             ],
111             verbose=0
112         )
113         metrics['Time_train'] = time.time()-time_start
114         pred_train = (model.predict(x_train) > 0.5).astype(int)
115         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
116         # test model
117         time_start = time.time()
118         pred_test = (model.predict(x_test) > 0.5).astype(int)
119         metrics['Time_test'] = time.time()-time_start
120         # score
```

```
121         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
122         metrics['Precision'] = precision_score(y_test_,pred_test)
123         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
124         cm = confusion_matrix(y_test_,pred_test)
125         metrics['Specifity'] = cm[0,0] / ( cm[0,1] + cm[0,0] )
126         return metrics

128 # ----------
129 # Run search
130 # ----------
131 used = [1,4,6,9]
132 groups = ['CowId','Lactation']
133 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
134 label_name = 'Lscore'
135 window_length = 27
136 window_shape=(81,)
137 n_neurons_1st = 6
138 n_neurons_other = 4
139 n_layers_other = 2
140 learning_rate_values = [0.0005,0.001,0.005,0.01,0.05]
141 batch_size_values = [8,16,32,64,128]
142 name = 'e2e-mlp_search'
143 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
144 output = name+'_'+ts
145 data_folder = '../data/'
146 log_folder = '../tb_log/'
147 log_folder = os.path.join(log_folder,output)

149 classifier = E2eMlpSearcher(used,n_neurons_1st,n_neurons_other,↩
        n_layers_other,learning_rate_values,batch_size_values)
150 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=True)
151 results = classifier.gridsearch(log_folder=log_folder)

153 results.to_csv(output+'.csv')
```

Listing A.14: Skript e2e-mlp_eval.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import sys
import os
import time
import pickle
import numpy as np
import pandas as pd
import tensorflow as tf
from datetime import datetime
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,↩
    confusion_matrix
from src.AbstractEvaluator2 import AbstractEvaluator2

# ---------------------
# Define specific class
# ---------------------
class E2eMlpEvaluator(AbstractEvaluator2):

    def __init__(self,left_out,n_neurons_1st,n_neurons_other,↩
        n_layers_other,learning_rate,batch_size):
        self.left_out = left_out
        self.n_neurons_1st = n_neurons_1st
        self.n_neurons_other = n_neurons_other
        self.n_layers_other = n_layers_other
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.X_const_train = None
        self.X_var_train = None
        self.Y_train = None
        self.X_const_test = None
        self.X_var_test = None
        self.Y_test = None
        self.metrics = None

    @classmethod
    def window_function(cls,window_data,**kwargs):
        return window_data.flatten()

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,↩
        x_var_test,y_test,learning_rate,batch_size,log_folder):
        x_train = np.hstack((x_const_train,x_var_train))
        x_test = np.hstack((x_const_test,x_var_test))
        y_train_ = y_train.astype(int)
        y_test_ = y_test.astype(int)
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy',
            'Train_Accuracy',
            'Precision',
            'Sensitivity',
            'Specifity',
            'Time_train',
            'Time_test'
        ])
        metrics = {
            'Accuracy' : np.nan,
            'Train_Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
            'Specifity' : np.nan,
            'Time_train' : np.nan,
```

```
63              'Time_test' : np.nan
64          }
65          kernel_initializer = 'lecun_normal'
66          activation = 'selu'
67          n_neurons_1st = self.n_neurons_1st
68          n_neurons_other = self.n_neurons_other
69          n_layers_other = self.n_layers_other
70          n_epochs = 1000
71          # set random seed for reproducible results
72          tf.random.set_seed(42)
73          # data normalization layer
74          norm_layer = keras.layers.experimental.preprocessing.↩
                Normalization()
75          norm_layer.adapt(x_train)
76          # build model
77          model = keras.models.Sequential()
78          model.add(keras.layers.InputLayer(input_shape=x_train.shape[1:])↩
                )
79          model.add(norm_layer)
80          model.add(keras.layers.Dense(
81              n_neurons_1st,
82              kernel_initializer=kernel_initializer,
83              activation=activation
84          ))
85          for layer in range(n_layers_other):
86              model.add(keras.layers.Dense(
87                  n_neurons_other,
88                  kernel_initializer=kernel_initializer,
89                  activation=activation
90              ))
91          model.add(keras.layers.Dense(1,activation='sigmoid'))
92          model.compile(
93              loss='binary_crossentropy',
94              optimizer=keras.optimizers.Nadam(learning_rate=learning_rate↩
                    ),
95              metrics=['accuracy']
96          )
97          run_log_folder = 'N'+str(y_train.shape[0])
98          run_log_folder = os.path.join(log_folder,run_log_folder)
99          # train model
100         time_start = datetime.now()
101         model.fit(
102             x_train,y_train_,
103             validation_data=(x_test,y_test_),
104             batch_size=batch_size,epochs=n_epochs,
105             callbacks = [
106                 keras.callbacks.TensorBoard(run_log_folder),
107                 keras.callbacks.ReduceLROnPlateau(patience=10,min_delta↩
                        =0.00001),
108                 keras.callbacks.EarlyStopping(patience=20,↩
                        restore_best_weights=True)
109             ],
110             verbose=0
111         )
112         metrics['Time_train'] = datetime.now()-time_start
113         pred_train = (model.predict(x_train) > 0.5).astype(int)
114         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
115         # test model
116         time_start = datetime.now()
117         pred_test = (model.predict(x_test) > 0.5).astype(int)
118         metrics['Time_test'] = datetime.now()-time_start
119         # score
120         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
121         metrics['Precision'] = precision_score(y_test_,pred_test)
122         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
123         cm = confusion_matrix(y_test_,pred_test)
124         metrics['Specifity'] = cm[0,0] / ( cm[0,1] + cm[0,0] )
125         result = metrics.copy()
```

```
126         result['learning_rate'] = learning_rate
127         result['batch_size'] = batch_size
128         self.metrics = self.metrics.append(result, ignore_index = True)
129         return metrics

131 # --------------
132 # Run evaluation
133 # --------------
134 left_out = int(sys.argv[1])
135 groups = ['CowId','Lactation']
136 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
137 label_name = 'Lscore'
138 window_length = 27
139 window_shape=(81,)
140 n_neurons_1st = 6
141 n_neurons_other = 4
142 n_layers_other = 2
143 learning_rate = 0.005
144 batch_size = 128
145 name = 'e2e-mlp_eval'+str(left_out)
146 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
147 output = name+'_'+ts
148 data_folder = '../data/'
149 log_folder = '../tb_log/'
150 log_folder = os.path.join(log_folder, output)

152 classifier = E2eMlpEvaluator(left_out,n_neurons_1st,n_neurons_other,↩
        n_layers_other,learning_rate,batch_size)
153 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=True)
154 result = classifier.prepare_fit(x_const_train=classifier.X_const_train,↩
        x_var_train=classifier.X_var_train,y_train=classifier.Y_train,↩
        x_const_test=classifier.X_const_test,x_var_test=classifier.X_var_test↩
        ,y_test=classifier.Y_test,learning_rate=learning_rate,batch_size=↩
        batch_size,log_folder=log_folder)

156 print(f"\nEvaluate␣E2E-MLP␣on␣left_out={left_out}␣with␣parameters:\n↩
        ----------------------\n\n")
157 print(f"learning_rate␣=␣{learning_rate}")
158 print(f"batch_size␣=␣{batch_size}")
159 print('\n-----\n')
160 print(f"Output:␣{output}")
161 print('\n-----\n')
162 print(f"Accuracy␣:␣{result['Accuracy']}␣(Training␣accuracy␣:␣{result['↩
        Train_Accuracy']})")
163 print(f"Precision␣:␣{result['Precision']}")
164 print(f"Sensitivity␣:␣{result['Sensitivity']}")
165 print(f"Specifity␣:␣{result['Specifity']}")
166 print(f"Time_train␣:␣{result['Time_train']}")
167 print(f"Time_test␣:␣{result['Time_test']}")
```

Listing A.15: Skript e2e-cnn_search.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import os
import time
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import LeaveOneGroupOut
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractSearcher2 import AbstractSearcher2
from src.winfunc import transpose

# ---------------------
# Define specific class
# ---------------------
class E2eCnnSearcher(AbstractSearcher2):

    def __init__(self,used,learning_rate_values,batch_size_values):
        self.used = used
        self.learning_rate_values = learning_rate_values
        self.batch_size_values = batch_size_values
        self.logo = LeaveOneGroupOut()
        self.X_const = None
        self.X_var = None
        self.Y = None
        self.Split = None
        self.metrics = None

    def gridsearch(self,**kwargs):
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy_mean', 'Accuracy_sd',
            'Precision_mean', 'Precision_sd',
            'Sensitivity_mean', 'Sensitivity_sd',
            'Specifity_mean', 'Specifity_sd',
            'Time_train_mean', 'Time_train_sd',
            'Time_test_mean', 'Time_test_sd'
        ])
        for learning_rate in self.learning_rate_values:
            for batch_size in self.batch_size_values:
                cv_result = self.crossvalidate(learning_rate=learning_rate,batch_size=batch_size,log_folder=log_folder)
                cv_result['learning_rate'] = learning_rate
                cv_result['batch_size'] = batch_size
                self.metrics = self.metrics.append(cv_result, ignore_index=True)
        self.metrics = self.metrics.sort_values(by=['Accuracy_mean'],ascending=False)
        return self.metrics

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,x_var_test,y_test,learning_rate,batch_size,log_folder,**kwargs):
        y_train_ = y_train.astype(int)
        y_test_ = y_test.astype(int)
        metrics = {
            'Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
            'Specifity' : np.nan,
            'Time_train' : np.nan,
```

```
60              'Time_test' : np.nan
61          }
62          kernel_initializer = 'he_normal'
63          activation = 'elu'
64          n_epochs = 1000
65          # set random seed for reproducible results
66          tf.random.set_seed(42)
67          # data normalization layers
68          norm_layer_const = keras.layers.experimental.preprocessing.↩
                Normalization()
69          norm_layer_const.adapt(x_const_train)
70          norm_layer_var = keras.layers.experimental.preprocessing.↩
                Normalization()
71          norm_layer_var.adapt(x_var_train)
72          # build model
73          input_const = keras.layers.Input(shape=x_const_train.shape[1:])
74          norm_const = norm_layer_const(input_const)
75          layer_const = keras.layers.Dense(2,activation=activation,↩
                kernel_initializer=kernel_initializer)(norm_const)
76          input_var = keras.layers.Input(shape=x_var_train.shape[1:])
77          norm_var = norm_layer_var(input_var)
78          conv1_var = keras.layers.Conv1D(
79              filters=128,kernel_size=8,
80              kernel_initializer=kernel_initializer,
81              use_bias=False,
82              padding='same'
83          )(norm_var)
84          bn1_var = keras.layers.BatchNormalization()(conv1_var)
85          act1_var = keras.layers.ELU()(bn1_var)
86          conv2_var = keras.layers.Conv1D(
87              filters=256,kernel_size=5,
88              kernel_initializer=kernel_initializer,
89              use_bias=False,
90              padding='same'
91          )(act1_var)
92          bn2_var = keras.layers.BatchNormalization()(conv2_var)
93          act2_var = keras.layers.ELU()(bn2_var)
94          conv3_var = keras.layers.Conv1D(
95              filters=128,kernel_size=3,
96              kernel_initializer=kernel_initializer,
97              use_bias=False,
98              padding='same'
99          )(act2_var)
100         bn3_var = keras.layers.BatchNormalization()(conv3_var)
101         act3_var = keras.layers.ELU()(bn3_var)
102         features_var = keras.layers.GlobalAveragePooling1D()(act3_var)
103         join = keras.layers.concatenate([layer_const,features_var])
104         out_layer = keras.layers.Dense(1,activation='sigmoid')(join)
105         model = keras.models.Model([input_const,input_var],out_layer)
106         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
107         run_log_folder = 'N'+str(y_train.shape[0])
108         run_log_folder = os.path.join(log_folder,run_log_folder)
109         # train model
110         time_start = time.time()
111         model.fit(
112             [x_const_train,x_var_train],y_train_,
113             validation_data=([x_const_test,x_var_test],y_test_),
114             batch_size=batch_size,epochs=n_epochs,
115             callbacks = [
116                 keras.callbacks.TensorBoard(run_log_folder),
117                 keras.callbacks.ReduceLROnPlateau(patience=10,min_delta↩
                        =0.00001),
118                 keras.callbacks.EarlyStopping(patience=20,↩
                        restore_best_weights=True)
119             ],
120             verbose=0
```

```
121         )
122         metrics['Time_train'] = time.time()-time_start
123         pred_train = (model.predict([x_const_train,x_var_train]) > 0.5).↩
                astype(int)
124         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
125         # test model
126         time_start = time.time()
127         pred_test = (model.predict([x_const_test,x_var_test]) > 0.5).↩
                astype(int)
128         metrics['Time_test'] = time.time()-time_start
129         # score
130         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
131         metrics['Precision'] = precision_score(y_test_,pred_test)
132         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
133         cm = confusion_matrix(y_test_,pred_test)
134         metrics['Specifity'] = cm[0,0] / ( cm[0,1] + cm[0,0] )
135         return metrics

137 # ----------
138 # Run search
139 # ----------
140 used = [1,4,6,9]
141 groups = ['CowId','Lactation']
142 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
143 label_name = 'Lscore'
144 window_length = 27
145 window_shape = (27,3)
146 learning_rate_values = [0.0005,0.001,0.005,0.01,0.05]
147 batch_size_values = [8,16,32,64,128]
148 name = 'e2e-cnn_search'
149 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
150 output = name+'_'+ts
151 data_folder = '../data/'
152 log_folder = '../tb_log/'
153 log_folder = os.path.join(log_folder,output)

155 classifier = E2eCnnSearcher(used,learning_rate_values,batch_size_values)
156 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=True)
157 results = classifier.gridsearch(log_folder=log_folder)

159 results.to_csv(output+'.csv')
```

Listing A.16: Skript e2e-cnn_eval.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import sys
import os
import time
import pickle
import numpy as np
import pandas as pd
import tensorflow as tf
from datetime import datetime
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractEvaluator2 import AbstractEvaluator2

# ---------------------
# Define specific class
# ---------------------
class E2eCnnEvaluator(AbstractEvaluator2):

    def __init__(self,left_out,learning_rate,batch_size):
        self.left_out = left_out
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.X_const_train = None
        self.X_var_train = None
        self.Y_train = None
        self.X_const_test = None
        self.X_var_test = None
        self.Y_test = None
        self.metrics = None

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,x_var_test,y_test,learning_rate,batch_size,log_folder):
        y_train_ = y_train.astype(int)
        y_test_ = y_test.astype(int)
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy',
            'Train_Accuracy',
            'Precision',
            'Sensitivity',
            'Specifity',
            'Time_train',
            'Time_test'
        ])
        metrics = {
            'Accuracy' : np.nan,
            'Train_Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
            'Specifity' : np.nan,
            'Time_train' : np.nan,
            'Time_test' : np.nan
        }
        kernel_initializer = 'he_normal'
        activation = 'elu'
        n_epochs = 1000
        # set random seed for reproducible results
        tf.random.set_seed(42)
        # data normalization layers
        norm_layer_const = keras.layers.experimental.preprocessing.Normalization()
```

```
63          norm_layer_const.adapt(x_const_train)
64          norm_layer_var = keras.layers.experimental.preprocessing.↩
                Normalization()
65          norm_layer_var.adapt(x_var_train)
66          # build model
67          input_const = keras.layers.Input(shape=x_const_train.shape[1:])
68          norm_const = norm_layer_const(input_const)
69          layer_const = keras.layers.Dense(2,activation=activation,↩
                kernel_initializer=kernel_initializer)(norm_const)
70          input_var = keras.layers.Input(shape=x_var_train.shape[1:])
71          norm_var = norm_layer_var(input_var)
72          conv1_var = keras.layers.Conv1D(
73              filters=128,kernel_size=8,
74              kernel_initializer=kernel_initializer,
75              use_bias=False,
76              padding='same'
77          )(norm_var)
78          bn1_var = keras.layers.BatchNormalization()(conv1_var)
79          act1_var = keras.layers.ELU()(bn1_var)
80          conv2_var = keras.layers.Conv1D(
81              filters=256,kernel_size=5,
82              kernel_initializer=kernel_initializer,
83              use_bias=False,
84              padding='same'
85          )(act1_var)
86          bn2_var = keras.layers.BatchNormalization()(conv2_var)
87          act2_var = keras.layers.ELU()(bn2_var)
88          conv3_var = keras.layers.Conv1D(
89              filters=128,kernel_size=3,
90              kernel_initializer=kernel_initializer,
91              use_bias=False,
92              padding='same'
93          )(act2_var)
94          bn3_var = keras.layers.BatchNormalization()(conv3_var)
95          act3_var = keras.layers.ELU()(bn3_var)
96          features_var = keras.layers.GlobalAveragePooling1D()(act3_var)
97          join = keras.layers.concatenate([layer_const,features_var])
98          out_layer = keras.layers.Dense(1,activation='sigmoid')(join)
99          model = keras.models.Model([input_const,input_var],out_layer)
100         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
101         run_log_folder = 'N'+str(y_train.shape[0])
102         run_log_folder = os.path.join(log_folder,run_log_folder)
103         # train model
104         time_start = datetime.now()
105         model.fit(
106             [x_const_train,x_var_train],y_train_,
107             validation_data=([x_const_test,x_var_test],y_test_),
108             batch_size=batch_size,epochs=n_epochs,
109             callbacks = [
110                 keras.callbacks.TensorBoard(run_log_folder),
111                 keras.callbacks.ReduceLROnPlateau(patience=10,min_delta↩
                        =0.00001),
112                 keras.callbacks.EarlyStopping(patience=20,↩
                        restore_best_weights=True)
113             ],
114             verbose=0
115         )
116         metrics['Time_train'] = datetime.now()-time_start
117         pred_train = (model.predict([x_const_train,x_var_train]) > 0.5).↩
                astype(int)
118         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
119         # test model
120         time_start = datetime.now()
121         pred_test = (model.predict([x_const_test,x_var_test]) > 0.5).↩
                astype(int)
122         metrics['Time_test'] = datetime.now()-time_start
```

```
123         # score
124         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
125         metrics['Precision'] = precision_score(y_test_,pred_test)
126         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
127         cm = confusion_matrix(y_test_,pred_test)
128         metrics['Specifity'] = cm[0,0]/(cm[0,1]+cm[0,0])
129         result = metrics.copy()
130         result['learning_rate'] = learning_rate
131         result['batch_size'] = batch_size
132         self.metrics = self.metrics.append(result,ignore_index=True)
133         return metrics

135 # --------------
136 # Run evaluation
137 # --------------
138 left_out = int(sys.argv[1])
139 groups = ['CowId','Lactation']
140 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
141 label_name = 'Lscore'
142 window_length = 27
143 window_shape = (27,3)
144 learning_rate = 0.05
145 batch_size = 32
146 name = 'e2e-cnn_eval'+str(left_out)
147 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
148 output = name+'_'+ts
149 data_folder = '../data/'
150 log_folder = '../tb_log/'
151 log_folder = os.path.join(log_folder, output)

153 classifier = E2eCnnEvaluator(left_out,learning_rate,batch_size)
154 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=True)
155 result = classifier.prepare_fit(x_const_train=classifier.X_const_train,↩
        x_var_train=classifier.X_var_train,y_train=classifier.Y_train,↩
        x_const_test=classifier.X_const_test,x_var_test=classifier.X_var_test↩
        ,y_test=classifier.Y_test,learning_rate=learning_rate,batch_size=↩
        batch_size,log_folder=log_folder)

157 print(f"\nEvaluate␣E2E-CNN␣on␣left_out={left_out}␣with␣parameters:\n↩
        ----------------------\n\n")
158 print(f"learning_rate␣=␣{learning_rate}")
159 print(f"batch_size␣=␣{batch_size}")
160 print('\n-----\n')
161 print(f"Output:␣{output}")
162 print('\n-----\n')
163 print(f"Accuracy␣:␣{result['Accuracy']}␣(Training␣accuracy␣:␣{result['↩
        Train_Accuracy']})")
164 print(f"Precision␣:␣{result['Precision']}")
165 print(f"Sensitivity␣:␣{result['Sensitivity']}")
166 print(f"Specifity␣:␣{result['Specifity']}")
167 print(f"Time_train␣:␣{result['Time_train']}")
168 print(f"Time_test␣:␣{result['Time_test']}")
```

Listing A.17: Skript e2e-gru_search.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
# requires numpy=1.19
# ---------------------
import os
import time
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import LeaveOneGroupOut
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractSearcher2 import AbstractSearcher2
from src.winfunc import transpose

# ---------------------
# Define specific class
# ---------------------
class E2eGruSearcher(AbstractSearcher2):

    def __init__(self,used,learning_rate_values,batch_size_values):
        self.used = used
        self.learning_rate_values = learning_rate_values
        self.batch_size_values = batch_size_values
        self.logo = LeaveOneGroupOut()
        self.X_const = None
        self.X_var = None
        self.Y = None
        self.Split = None
        self.metrics = None

    def gridsearch(self,**kwargs):
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy_mean', 'Accuracy_sd',
            'Precision_mean', 'Precision_sd',
            'Sensitivity_mean', 'Sensitivity_sd',
            'Specifity_mean', 'Specifity_sd',
            'Time_train_mean', 'Time_train_sd',
            'Time_test_mean', 'Time_test_sd'
        ])
        for learning_rate in self.learning_rate_values:
            for batch_size in self.batch_size_values:
                cv_result = self.crossvalidate(learning_rate=learning_rate,batch_size=batch_size,log_folder=log_folder)
                cv_result['learning_rate'] = learning_rate
                cv_result['batch_size'] = batch_size
                self.metrics = self.metrics.append(cv_result,ignore_index=True)
        self.metrics = self.metrics.sort_values(by=['Accuracy_mean'],ascending=False)
        return self.metrics

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,x_var_test,y_test,learning_rate,batch_size,log_folder,**kwargs):
        y_train_ = y_train.astype(int)
        y_test_ = y_test.astype(int)
        metrics = {
            'Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
```

```
60              'Specifity' : np.nan,
61              'Time_train' : np.nan,
62              'Time_test' : np.nan
63          }
64          n_epochs = 1000
65          # set random seed for reproducible results
66          tf.random.set_seed(42)
67          # data normalization layers
68          norm_layer_const = keras.layers.experimental.preprocessing.↩
                Normalization()
69          norm_layer_const.adapt(x_const_train)
70          norm_layer_var = keras.layers.experimental.preprocessing.↩
                Normalization()
71          norm_layer_var.adapt(x_var_train)
72          # build model
73          input_const = keras.layers.Input(shape=x_const_train.shape[1:])
74          norm_const = norm_layer_const(input_const)
75          layer_const = keras.layers.Dense(2,activation='elu',↩
                kernel_initializer='he_normal')(norm_const)
76          input_var = keras.layers.Input(shape=x_var_train.shape[1:])
77          norm_var = norm_layer_var(input_var)
78          rec1_var = keras.layers.GRU(units=16,return_sequences=True
79          )(norm_var)
80          rec2_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                rec1_var)
81          rec3_var = keras.layers.GRU(units=16)(rec2_var)
82          join = keras.layers.concatenate([layer_const,rec3_var])
83          dense =keras.layers.Dense(6,activation='elu',kernel_initializer=↩
                'he_normal')(join)
84          out_layer = keras.layers.Dense(1, activation='sigmoid')(dense)
85          model = keras.models.Model([input_const,input_var],out_layer)
86          model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
87          run_log_folder = 'N' + str(y_train.shape[0])
88          run_log_folder = os.path.join(log_folder,run_log_folder)
89          # train model
90          time_start = time.time()
91          model.fit(
92              [x_const_train,x_var_train],y_train_,
93              validation_data=([x_const_test,x_var_test],y_test_),
94              batch_size=batch_size,epochs=n_epochs,
95              callbacks = [
96                  keras.callbacks.TensorBoard(run_log_folder),
97                  keras.callbacks.ReduceLROnPlateau(patience=10,min_delta↩
                        =0.00001),
98                  keras.callbacks.EarlyStopping(patience=20, ↩
                        restore_best_weights=True)
99              ],
100             verbose=0
101         )
102         metrics['Time_train'] = time.time()-time_start
103         pred_train = (model.predict([x_const_train,x_var_train]) > 0.5).↩
                astype(int)
104         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
105         # test model
106         time_start = time.time()
107         pred_test = (model.predict([x_const_test,x_var_test]) > 0.5).↩
                astype(int)
108         metrics['Time_test'] = time.time()-time_start
109         # score
110         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
111         metrics['Precision'] = precision_score(y_test_,pred_test)
112         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
113         cm = confusion_matrix(y_test_,pred_test)
114         metrics['Specifity'] = cm[0,0]/(cm[0,1]+cm[0,0])
115         return metrics
```

```
117 # ----------
118 # Run search
119 # ----------
120 used = [1,4,6,9]
121 groups = ['CowId','Lactation']
122 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
123 label_name = 'Lscore'
124 window_length = 27
125 window_shape = (27,3)
126 learning_rate_values = [0.0005,0.001,0.005,0.01,0.05]
127 batch_size_values = [8,16,32,64,128]
128 name = 'e2e-gru_search'
129 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
130 output = name+'_'+ts
131 data_folder = '../data/'
132 log_folder = '../tb_log/'
133 log_folder = os.path.join(log_folder,output)

135 classifier = E2eGruSearcher(used,learning_rate_values,batch_size_values)
136 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=True)
137 results = classifier.gridsearch(log_folder=log_folder)

139 results.to_csv(output+'.csv')
```

Listing A.18: Skript e2e-gru_eval.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# --------------------
# requires numpy=1.19
# --------------------
import sys
import os
import time
import pickle
import numpy as np
import pandas as pd
import tensorflow as tf
from datetime import datetime
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractEvaluator2 import AbstractEvaluator2

# ----------------------
# Define specific class
# ----------------------
class E2eGruEvaluator(AbstractEvaluator2):

    def __init__(self,left_out,learning_rate,batch_size):
        self.left_out = left_out
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.X_const_train = None
        self.X_var_train = None
        self.Y_train = None
        self.X_const_test = None
        self.X_var_test = None
        self.Y_test = None
        self.metrics = None

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,x_var_test,y_test,learning_rate,batch_size,log_folder):
        y_train_ = y_train.astype(int)
        y_test_ = y_test.astype(int)
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy',
            'Train_Accuracy',
            'Precision',
            'Sensitivity',
            'Specifity',
            'Time_train',
            'Time_test'
        ])
        metrics = {
            'Accuracy' : np.nan,
            'Train_Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
            'Specifity' : np.nan,
            'Time_train' : np.nan,
            'Time_test' : np.nan
        }
        n_epochs = 1000
        # set random seed for reproducible results
        tf.random.set_seed(42)
        # data normalization layers
        norm_layer_const = keras.layers.experimental.preprocessing.Normalization()
```

```
63          norm_layer_const.adapt(x_const_train)
64          norm_layer_var = keras.layers.experimental.preprocessing.↩
                Normalization()
65          norm_layer_var.adapt(x_var_train)
66          # build model
67          input_const = keras.layers.Input(shape=x_const_train.shape[1:])
68          norm_const = norm_layer_const(input_const)
69          layer_const = keras.layers.Dense(2,activation='elu',↩
                kernel_initializer='he_normal')(norm_const)
70          input_var = keras.layers.Input(shape=x_var_train.shape[1:])
71          norm_var = norm_layer_var(input_var)
72          rec1_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                norm_var)
73          rec2_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                rec1_var)
74          rec3_var = keras.layers.GRU(units=16)(rec2_var)
75          join = keras.layers.concatenate([layer_const,rec3_var])
76          dense =keras.layers.Dense(6,activation='elu',kernel_initializer=↩
                'he_normal')(join)
77          out_layer = keras.layers.Dense(1, activation='sigmoid')(join)
78          model = keras.models.Model([input_const,input_var],out_layer)
79          model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
80          run_log_folder = 'N' + str(y_train.shape[0])
81          run_log_folder = os.path.join(log_folder,run_log_folder)
82          # train model
83          time_start = datetime.now()
84          model.fit(
85              [x_const_train,x_var_train],y_train_,
86              validation_data=([x_const_test,x_var_test],y_test_),
87              batch_size=batch_size,epochs=n_epochs,
88              callbacks = [
89                  keras.callbacks.TensorBoard(run_log_folder),
90                  keras.callbacks.ReduceLROnPlateau(patience=10,min_delta↩
                        =0.00001),
91                  keras.callbacks.EarlyStopping(patience=20,↩
                        restore_best_weights=True)
92              ],
93              verbose=0
94          )
95          metrics['Time_train'] = datetime.now()-time_start
96          pred_train = (model.predict([x_const_train,x_var_train]) > 0.5).↩
                astype(int)
97          metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
98          # test model
99          time_start = datetime.now()
100         pred_test = (model.predict([x_const_test,x_var_test]) > 0.5).↩
                astype(int)
101         metrics['Time_test'] = datetime.now()-time_start
102         # score
103         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
104         metrics['Precision'] = precision_score(y_test_,pred_test)
105         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
106         cm = confusion_matrix(y_test_,pred_test)
107         metrics['Specifity'] = cm[0,0]/(cm[0,1]+cm[0,0])
108         result = metrics.copy()
109         result['learning_rate'] = learning_rate
110         result['batch_size'] = batch_size
111         self.metrics = self.metrics.append(result,ignore_index=True)
112         return metrics

114 # --------------
115 # Run evaluation
116 # --------------
117 left_out = int(sys.argv[1])
118 groups = ['CowId','Lactation']
```

```
119 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
120 label_name = 'Lscore'
121 window_length = 27
122 window_shape = (27,3)
123 learning_rate = 0.001
124 batch_size = 32
125 name = 'e2e-gru_eval'+str(left_out)
126 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
127 output = name+'_'+ts
128 data_folder = '../data/'
129 log_folder = '../tb_log/'
130 log_folder = os.path.join(log_folder, output)

132 classifier = E2eGruEvaluator(left_out,learning_rate,batch_size)
133 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=True)
134 result = classifier.prepare_fit(x_const_train=classifier.X_const_train,↩
        x_var_train=classifier.X_var_train,y_train=classifier.Y_train,↩
        x_const_test=classifier.X_const_test,x_var_test=classifier.X_var_test↩
        ,y_test=classifier.Y_test,learning_rate=learning_rate,batch_size=↩
        batch_size,log_folder=log_folder)

136 print(f"\nEvaluate␣E2E-GRU␣on␣left_out={left_out}␣with␣parameters:\n↩
        ----------------------\n\n")
137 print(f"learning_rate␣=␣{learning_rate}")
138 print(f"batch_size␣=␣{batch_size}")
139 print('\n-----\n')
140 print(f"Output:␣{output}")
141 print('\n-----\n')
142 print(f"Accuracy␣:␣{result['Accuracy']}␣(Training␣accuracy␣:␣{result['↩
        Train_Accuracy']})")
143 print(f"Precision␣:␣{result['Precision']}")
144 print(f"Sensitivity␣:␣{result['Sensitivity']}")
145 print(f"Specifity␣:␣{result['Specifity']}")
146 print(f"Time_train␣:␣{result['Time_train']}")
147 print(f"Time_test␣:␣{result['Time_test']}")
```

A.4 Ausführbare Skripte — Generative Ansätze

Listing A.19: Skript ae-mlp_search.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import os
import time
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import LeaveOneGroupOut
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractSearcher2 import AbstractSearcher2

# ---------------------
# Define specific class
# ---------------------
class AeMlpSearcher(AbstractSearcher2):

    def __init__(self,used,n_neurons_1st,n_neurons_other,n_layers_other,learning_rate_values,batch_size_values):
        self.used = used
        self.n_neurons_1st = n_neurons_1st
        self.n_neurons_other = n_neurons_other
        self.n_layers_other = n_layers_other
        self.learning_rate_values = learning_rate_values
        self.batch_size_values = batch_size_values
        self.logo = LeaveOneGroupOut()
        self.X_const = None
        self.X_var = None
        self.Y = None
        self.Split = None
        self.metrics = None

    @classmethod
    def window_function(cls,window_data,**kwargs):
        return window_data.flatten()

    def gridsearch(self,**kwargs)
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy_mean', 'Accuracy_sd',
            'Precision_mean', 'Precision_sd',
            'Sensitivity_mean', 'Sensitivity_sd',
            'Specifity_mean', 'Specifity_sd',
            'Time_train_mean', 'Time_train_sd',
            'Time_test_mean', 'Time_test_sd'
        ])
        for learning_rate in self.learning_rate_values:
            for batch_size in self.batch_size_values:
                cv_result = self.crossvalidate(learning_rate=learning_rate,batch_size=batch_size,log_folder=log_folder)
                cv_result['learning_rate'] = learning_rate
                cv_result['batch_size'] = batch_size
                self.metrics = self.metrics.append(cv_result,ignore_index=True)
        self.metrics = self.metrics.sort_values(by=['Accuracy_mean'],ascending=False)
        return self.metrics

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,x_var_test,y_test,learning_rate,batch_size,log_folder):
        x_train = np.hstack((x_const_train,x_var_train))
```

```
59          x_test = np.hstack((x_const_test,x_var_test))
60          # normalization outside of the model
61          normalizer = keras.layers.experimental.preprocessing.↩
                Normalization()
62          normalizer.adapt(x_train)
63          x_train = normalizer(x_train).numpy()
64          x_test = normalizer(x_test).numpy()
65          idx_labels_train = np.nonzero(~np.isnan(y_train.astype(float)))
66          idx_labels_test = np.nonzero(~np.isnan(y_test.astype(float)))
67          metrics = {
68              'Accuracy' : np.nan,
69              'Precision' : np.nan,
70              'Sensitivity' : np.nan,
71              'Specifity' : np.nan,
72              'Time_train' : np.nan,
73              'Time_test' : np.nan
74          }
75          kernel_initializer = 'lecun_normal'
76          activation = 'selu'
77          n_neurons_1st = self.n_neurons_1st
78          n_neurons_other = self.n_neurons_other
79          n_layers_other = self.n_layers_other
80          n_epochs = 1000
81          patience_reduce = 10
82          patience_stop = 50
83          # set random seed for reproducible results
84          tf.random.set_seed(42)
85          # build autoencoder
86          encoder = keras.models.Sequential()
87          encoder.add(keras.layers.InputLayer(input_shape=x_train.shape↩
                [1:]))
88          encoder.add(keras.layers.Dense(
89              n_neurons_1st,
90              kernel_initializer=kernel_initializer,
91              activation=activation
92          ))
93          for layer in range(n_layers_other):
94              encoder.add(keras.layers.Dense(
95                  n_neurons_other,
96                  kernel_initializer=kernel_initializer,
97                  activation=activation
98              ))
99          encoder.add(keras.layers.Dense(
100             5,
101             kernel_initializer=kernel_initializer,
102             activation=activation
103         ))
104         decoder = keras.models.Sequential()
105         for layer in range(n_layers_other):
106             decoder.add(keras.layers.Dense(
107                 n_neurons_other,
108                 kernel_initializer=kernel_initializer,
109                 activation=activation
110             ))
111         decoder.add(keras.layers.Dense(
112             n_neurons_1st,
113             kernel_initializer=kernel_initializer,
114             activation=activation
115         ))
116         decoder.add(keras.layers.Dense(np.sum(x_train.shape[1:])))
117         ae = keras.models.Sequential([encoder,decoder])
118         ae.compile(loss='huber_loss',optimizer=keras.optimizers.Nadam(↩
                learning_rate=learning_rate))
119         run_log_folder = 'ae_N'+str(y_train.shape[0])
120         run_log_folder = os.path.join(log_folder,run_log_folder)
121         # train autoencoder
122         time_start = time.time()
123         ae.fit(
```

```
124             x_train,x_train,
125             validation_data=(x_test,x_test),
126             batch_size=batch_size,
127             epochs=n_epochs,
128             callbacks=[
129                 keras.callbacks.TensorBoard(run_log_folder),
130                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
131                 keras.callbacks.EarlyStopping(patience=patience_stop,↩
                        restore_best_weights=True)
132             ],
133             verbose=0
134         )
135         # reduce to labeled data
136         x_train_ = x_train[idx_labels_train]
137         x_test_ = x_test[idx_labels_test]
138         y_train_ = (y_train[idx_labels_train]).astype(int)
139         y_test_ = (y_test[idx_labels_test]).astype(int)
140         # build model
141         classifier = keras.models.Sequential()
142         classifier.add(keras.layers.Dense(3,kernel_initializer=↩
                kernel_initializer,activation=activation))
143         classifier.add(keras.layers.Dense(3,kernel_initializer=↩
                kernel_initializer,activation=activation))
144         classifier.add(keras.layers.Dense(1,activation='sigmoid'))
145         model = keras.models.Sequential([encoder,classifier])
146         # freeze encoder's weights
147         for layer in encoder.layers:
148             layer.trainable = False
149         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
150         run_log_folder = 'inter_N'+str(y_train.shape[0])
151         run_log_folder = os.path.join(log_folder,run_log_folder)
152         # train model 1st time
153         model.fit(
154             x_train_,y_train_,
155             validation_data=(x_test_,y_test_),
156             batch_size=batch_size,
157             epochs=n_epochs,
158             callbacks=[
159                 keras.callbacks.TensorBoard(run_log_folder),
160                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
161                 keras.callbacks.EarlyStopping(patience=patience_stop,↩
                        restore_best_weights=True)
162             ],
163             verbose=0
164         )
165         # unfreeze encoder's weights
166         for layer in encoder.layers:
167             layer.trainable = True
168         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate/10),metrics=['↩
                accuracy'])
169         run_log_folder = 'final_N'+str(y_train.shape[0])
170         run_log_folder = os.path.join(log_folder, run_log_folder)
171         # train model finally
172         model.fit(
173             x_train_,y_train_,
174             validation_data=(x_test_,y_test_),
175             batch_size=batch_size,
176             epochs=n_epochs,
177             callbacks=[
178                 keras.callbacks.TensorBoard(run_log_folder),
179                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
```

```
180                 keras.callbacks.EarlyStopping(patience=patience_stop,↩
                        restore_best_weights=True)
181             ],
182             verbose=0
183         )
184         metrics['Time_train'] = time.time()-time_start
185         pred_train = (model.predict(x_train_) > 0.5).astype(int)
186         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
187         # test model
188         time_start = time.time()
189         pred_test = (model.predict(x_test_) > 0.5).astype(int)
190         metrics['Time_test'] = time.time()-time_start
191         # score
192         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
193         metrics['Precision'] = precision_score(y_test_,pred_test)
194         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
195         cm = confusion_matrix(y_test_,pred_test)
196         metrics['Specifity'] = cm[0,0] / ( cm[0,1] + cm[0,0] )
197         return metrics

199 # ----------
200 # Run search
201 # ----------
202 used = [1,4,6,9]
203 groups = ['CowId','Lactation']
204 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
205 label_name = 'Lscore'
206 window_length = 27
207 window_shape=(81,)
208 n_neurons_1st = 40
209 n_neurons_other = 20
210 n_layers_other = 3
211 learning_rate_values = [0.001,0.005,0.01,0.05]
212 batch_size_values = [32,128,512]
213 name = 'ae-mlp_search'
214 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
215 output = name+'_'+ts
216 data_folder = '../data/'
217 log_folder = '../tb_log/'
218 log_folder = os.path.join(log_folder,output)

220 classifier = AeMlpSearcher(used,n_neurons_1st,n_neurons_other,↩
        n_layers_other,learning_rate_values,batch_size_values)
221 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=False)
222 results = classifier.gridsearch(log_folder=log_folder)

224 results.to_csv(output+'.csv')
```

Listing A.20: Skript ae-mlp_eval.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import sys
import os
import time
import pickle
import numpy as np
import pandas as pd
import tensorflow as tf
from datetime import datetime
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractEvaluator2 import AbstractEvaluator2

# ---------------------
# Define specific class
# ---------------------
class AeMlpEvaluator(AbstractEvaluator2):

    def __init__(self,left_out,n_neurons_1st,n_neurons_other,n_layers_other,learning_rate,batch_size):
        self.left_out = left_out
        self.n_neurons_1st = n_neurons_1st
        self.n_neurons_other = n_neurons_other
        self.n_layers_other = n_layers_other
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.X_const_train = None
        self.X_var_train = None
        self.Y_train = None
        self.X_const_test = None
        self.X_var_test = None
        self.Y_test = None
        self.metrics = None

    @classmethod
    def window_function(cls,window_data,**kwargs):
        return window_data.flatten()

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,x_var_test,y_test,learning_rate,batch_size,log_folder):
        x_train = np.hstack((x_const_train,x_var_train))
        x_test = np.hstack((x_const_test,x_var_test))
        # normalization outside of the model
        normalizer = keras.layers.experimental.preprocessing.Normalization()
        normalizer.adapt(x_train)
        x_train = normalizer(x_train).numpy()
        x_test = normalizer(x_test).numpy()
        idx_labels_train = np.nonzero(~np.isnan(y_train.astype(float)))
        idx_labels_test = np.nonzero(~np.isnan(y_test.astype(float)))
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy',
            'Train_Accuracy',
            'Precision',
            'Sensitivity',
            'Specifity',
            'Time_train',
            'Time_test'
        ])
        metrics = {
```

```
62              'Accuracy' : np.nan,
63              'Train_Accuracy' : np.nan,
64              'Precision' : np.nan,
65              'Sensitivity' : np.nan,
66              'Specifity' : np.nan,
67              'Time_train' : np.nan,
68              'Time_test' : np.nan
69          }
70          kernel_initializer = 'lecun_normal'
71          activation = 'selu'
72          n_neurons_1st = self.n_neurons_1st
73          n_neurons_other = self.n_neurons_other
74          n_layers_other = self.n_layers_other
75          n_epochs = 1000
76          patience_reduce = 10
77          patience_stop = 50
78          # set random seed for reproducible results
79          tf.random.set_seed(42)
80          # build autoencoder
81          encoder = keras.models.Sequential()
82          encoder.add(keras.layers.InputLayer(input_shape=x_train.shape↩
                [1:]))
83          encoder.add(keras.layers.Dense(
84              n_neurons_1st,
85              kernel_initializer=kernel_initializer,
86              activation=activation
87          ))
88          for layer in range(n_layers_other):
89              encoder.add(keras.layers.Dense(
90                  n_neurons_other,
91                  kernel_initializer=kernel_initializer,
92                  activation=activation
93              ))
94          encoder.add(keras.layers.Dense(
95              5,
96              kernel_initializer=kernel_initializer,
97              activation=activation
98          ))
99          decoder = keras.models.Sequential()
100         for layer in range(n_layers_other):
101             decoder.add(keras.layers.Dense(
102                 n_neurons_other,
103                 kernel_initializer=kernel_initializer,
104                 activation=activation
105             ))
106         decoder.add(keras.layers.Dense(
107             n_neurons_1st,
108             kernel_initializer=kernel_initializer,
109             activation=activation
110         ))
111         decoder.add(keras.layers.Dense(np.sum(x_train.shape[1:])))
112         ae = keras.models.Sequential([encoder,decoder])
113         ae.compile(loss='huber_loss',optimizer=keras.optimizers.Nadam(↩
                learning_rate=learning_rate))
114         run_log_folder = 'ae_N'+str(y_train.shape[0])
115         run_log_folder = os.path.join(log_folder, run_log_folder)
116         # train autoencoder
117         time_start = datetime.now()
118         ae.fit(
119             x_train,x_train,
120             validation_data=(x_test,x_test),
121             batch_size=batch_size,
122             epochs=n_epochs,
123             callbacks = [
124                 keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
125                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
```

```
126                 keras.callbacks.EarlyStopping(patience=patience_stop,↩
                        restore_best_weights=True)
127             ],
128             verbose=0
129         )
130         # reduce to labeled data
131         x_train_ = x_train[idx_labels_train]
132         x_test_ = x_test[idx_labels_test]
133         y_train_ = (y_train[idx_labels_train]).astype(int)
134         y_test_ = (y_test[idx_labels_test]).astype(int)
135         # build model
136         classifier = keras.models.Sequential()
137         classifier.add(keras.layers.Dense(3,kernel_initializer=↩
                kernel_initializer,activation=activation))
138         classifier.add(keras.layers.Dense(3,kernel_initializer=↩
                kernel_initializer,activation=activation))
139         classifier.add(keras.layers.Dense(1,activation='sigmoid'))
140         model = keras.models.Sequential([encoder,classifier])
141         # freeze encoder's weights
142         for layer in encoder.layers:
143             layer.trainable = False
144         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
145         run_log_folder = 'inter_N'+str(y_train.shape[0])
146         run_log_folder = os.path.join(log_folder,run_log_folder)
147         # train model 1st time
148         model.fit(
149             x_train_,y_train_,
150             validation_data=(x_test_,y_test_),
151             batch_size=batch_size,
152             epochs=n_epochs,
153             callbacks = [
154                 keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
155                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
156                 keras.callbacks.EarlyStopping(patience=patience_stop,↩
                        restore_best_weights=True)
157             ],
158             verbose=0
159         )
160         # unfreeze encoder's weights
161         for layer in encoder.layers:
162             layer.trainable = True
163         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate/10),metrics=['↩
                accuracy'])
164         run_log_folder = 'final_N' + str(y_train.shape[0])
165         run_log_folder = os.path.join(log_folder,run_log_folder)
166         # train model finally
167         model.fit(
168             x_train_,y_train_,
169             validation_data=(x_test_,y_test_),
170             batch_size=batch_size,
171             epochs=n_epochs,
172             callbacks = [
173                 keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
174                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
175                 keras.callbacks.EarlyStopping(patience=patience_stop,↩
                        restore_best_weights=True)
176             ],
177             verbose=0
178         )
179         metrics['Time_train'] = datetime.now()-time_start
180         pred_train = (model.predict(x_train_) > 0.5).astype(int)
```

```
181         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
182         # test model
183         time_start = datetime.now()
184         pred_test = (model.predict(x_test_) > 0.5).astype(int)
185         metrics['Time_test'] = datetime.now()-time_start
186         # score
187         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
188         metrics['Precision'] = precision_score(y_test_,pred_test)
189         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
190         cm = confusion_matrix(y_test_,pred_test)
191         metrics['Specifity'] = cm[0,0] / ( cm[0,1] + cm[0,0] )
192         result = metrics.copy()
193         result['learning_rate'] = learning_rate
194         result['batch_size'] = batch_size
195         self.metrics = self.metrics.append(result,ignore_index=True)
196         return metrics
197 # --------------
198 # Run evaluation
199 # --------------
200 left_out = int(sys.argv[1])
201 groups = ['CowId','Lactation']
202 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
203 label_name = 'Lscore'
204 window_length = 27
205 window_shape=(81,)
206 n_neurons_1st = 40
207 n_neurons_other = 20
208 n_layers_other = 3
209 learning_rate = 0.001
210 batch_size = 128
211 name = 'ae-mlp_eval'+str(left_out)
212 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
213 output = name+'_'+ts
214 data_folder = '../data/'
215 log_folder = '../tb_log/'
216 log_folder = os.path.join(log_folder, output)

218 classifier = AeMlpEvaluator(left_out,n_neurons_1st,n_neurons_other,↩
        n_layers_other,learning_rate,batch_size)
219 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=False)
220 result = classifier.prepare_fit(x_const_train=classifier.X_const_train,↩
        x_var_train=classifier.X_var_train,y_train=classifier.Y_train,↩
        x_const_test=classifier.X_const_test,x_var_test=classifier.X_var_test↩
        ,y_test=classifier.Y_test,learning_rate=learning_rate,batch_size=↩
        batch_size,log_folder=log_folder)

222 print(f"\nEvaluate␣AE-MLP␣on␣left_out={left_out}␣with␣parameters:\n↩
        ----------------------\n\n")
223 print(f"learning_rate␣=␣{learning_rate}")
224 print(f"batch_size␣=␣{batch_size}")
225 print('\n-----\n')
226 print(f"Output:␣{output}")
227 print('\n-----\n')
228 print(f"Accuracy␣:␣{result['Accuracy']}␣(Training␣accuracy␣:␣{result['↩
        Train_Accuracy']})")
229 print(f"Precision␣:␣{result['Precision']}")
230 print(f"Sensitivity␣:␣{result['Sensitivity']}")
231 print(f"Specifity␣:␣{result['Specifity']}")
232 print(f"Time_train␣:␣{result['Time_train']}")
233 print(f"Time_test␣:␣{result['Time_test']}")
```

Listing A.21: Skript ae-cnn_search.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import os
import time
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import LeaveOneGroupOut
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractSearcher2 import AbstractSearcher2

# ---------------------
# Define specific class
# ---------------------
class AeCnnSearcher(AbstractSearcher2):

    def __init__(self,used,learning_rate_values,batch_size_values):
        self.used = used
        self.learning_rate_values = learning_rate_values
        self.batch_size_values = batch_size_values
        self.logo = LeaveOneGroupOut()
        self.X_const = None
        self.X_var = None
        self.Y = None
        self.Split = None
        self.metrics = None

    def gridsearch(self,**kwargs):
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy_mean', 'Accuracy_sd',
            'Precision_mean', 'Precision_sd',
            'Sensitivity_mean', 'Sensitivity_sd',
            'Specifity_mean', 'Specifity_sd',
            'Time_train_mean', 'Time_train_sd',
            'Time_test_mean', 'Time_test_sd'
        ])
        for learning_rate in self.learning_rate_values:
            for batch_size in self.batch_size_values:
                cv_result = self.crossvalidate(learning_rate=learning_rate,batch_size=batch_size,log_folder=log_folder)
                cv_result['learning_rate'] = learning_rate
                cv_result['batch_size'] = batch_size
                self.metrics = self.metrics.append(cv_result,ignore_index=True)
        self.metrics = self.metrics.sort_values(by=['Accuracy_mean'],ascending=False)
        return self.metrics

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,x_var_test,y_test,learning_rate,batch_size,log_folder):
        # normalization outside of the model
        norm_const = keras.layers.experimental.preprocessing.Normalization()
        norm_const.adapt(x_const_train)
        x_const_train = norm_const(x_const_train).numpy()
        x_const_test = norm_const(x_const_test).numpy()
        norm_var = keras.layers.experimental.preprocessing.Normalization()
        norm_var.adapt(x_var_train)
```

```
58          x_var_train = norm_var(x_var_train).numpy()
59          x_var_test = norm_var(x_var_test).numpy()
60          idx_labels_train = np.nonzero(~np.isnan(y_train.astype(float)))
61          idx_labels_test = np.nonzero(~np.isnan(y_test.astype(float)))
62          metrics = {
63              'Accuracy' : np.nan,
64              'Precision' : np.nan,
65              'Sensitivity' : np.nan,
66              'Specifity' : np.nan,
67              'Time_train' : np.nan,
68              'Time_test' : np.nan
69          }
70          kernel_initializer = 'he_normal'
71          activation = 'elu'
72          n_epochs = 1000
73          patience_reduce = 10
74          patience_stop = 50
75          # set random seed for reproducible results
76          tf.random.set_seed(42)
77          # build encoder
78          input_const = keras.layers.Input(shape=x_const_train.shape[1:])
79          layer_const = keras.layers.Dense(2,activation=activation,↩
                kernel_initializer=kernel_initializer)(input_const)
80          input_var = keras.layers.Input(shape=x_var_train.shape[1:])
81          conv1_var = keras.layers.Conv1D(
82              filters=128,kernel_size=8,
83              kernel_initializer=kernel_initializer,
84              use_bias=False,
85              padding='same'
86          )(input_var)
87          bn1_var = keras.layers.BatchNormalization()(conv1_var)
88          act1_var = keras.layers.ELU()(bn1_var)
89          conv2_var = keras.layers.Conv1D(
90              filters=256,kernel_size=5,
91              kernel_initializer=kernel_initializer,
92              use_bias=False,
93              padding='same'
94          )(act1_var)
95          bn2_var = keras.layers.BatchNormalization()(conv2_var)
96          act2_var = keras.layers.ELU()(bn2_var)
97          conv3_var = keras.layers.Conv1D(
98              filters=128,kernel_size=3,
99              kernel_initializer=kernel_initializer,
100             use_bias=False,
101             padding='same'
102         )(act2_var)
103         bn3_var = keras.layers.BatchNormalization()(conv3_var)
104         act3_var = keras.layers.ELU()(bn3_var)
105         features_var = keras.layers.GlobalAveragePooling1D()(act3_var)
106         join = keras.layers.concatenate([layer_const,features_var])
107         # build decoder
108         output_decoder_const = keras.layers.Dense(2)(join)
109         de_reshape1_var = keras.layers.Reshape((1,130))(join)
110         deconv1_var = keras.layers.Conv1DTranspose(
111             filters=128,kernel_size=3,
112             kernel_initializer=kernel_initializer,
113             use_bias=False,
114             padding='valid'
115         )(de_reshape1_var)
116         de_bn1_var = keras.layers.BatchNormalization()(deconv1_var)
117         de_act1_var = keras.layers.ELU()(de_bn1_var)
118         deconv2_var = keras.layers.Conv1DTranspose(
119             filters=256,kernel_size=5,
120             kernel_initializer=kernel_initializer,
121             use_bias=False,
122             padding='valid'
123         )(de_act1_var)
124         de_bn2_var = keras.layers.BatchNormalization()(deconv2_var)
```

```
125         de_act2_var = keras.layers.ELU()(de_bn2_var)
126         deconv3_var = keras.layers.Conv1DTranspose(
127             filters=128,kernel_size=8,
128             kernel_initializer=kernel_initializer,
129             use_bias=False,
130             padding='valid'
131         )(de_act2_var)
132         de_bn3_var = keras.layers.BatchNormalization()(deconv3_var)
133         de_act3_var = keras.layers.ELU()(de_bn3_var)
134         output_decoder_var = keras.layers.Conv1DTranspose(
135             filters=3,kernel_size=14,padding='valid',
136             activation=None, kernel_initializer='glorot_uniform'
137         )(de_act3_var)
138         # combine to autoencoder
139         ae = keras.models.Model([input_const,input_var],[↩
                output_decoder_const,output_decoder_var])
140         ae.compile(loss='huber_loss',optimizer=keras.optimizers.Nadam(↩
                learning_rate=learning_rate))
141         run_log_folder = 'ae_N'+str(y_train.shape[0])
142         run_log_folder = os.path.join(log_folder,run_log_folder)
143         # train autoencoder
144         time_start = time.time()
145         ae.fit(
146             [x_const_train,x_var_train],[x_const_train,x_var_train],
147             validation_data=([x_const_test,x_var_test],[x_const_test,↩
                    x_var_test]),
148             batch_size=batch_size,epochs=n_epochs,
149             callbacks = [
150                 keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
151                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
152                 keras.callbacks.EarlyStopping(patience=patience_stop,↩
                        restore_best_weights=True)
153             ],
154             verbose=0
155         )
156         # reduce to labeled data
157         x_const_train_ = x_const_train[idx_labels_train]
158         x_var_train_ = x_var_train[idx_labels_train]
159         x_const_test_ = x_const_test[idx_labels_test]
160         x_var_test_ = x_var_test[idx_labels_test]
161         y_train_ = (y_train[idx_labels_train]).astype(int)
162         y_test_ = (y_test[idx_labels_test]).astype(int)
163         # build classifier
164         c1 = keras.layers.Dense(15,kernel_initializer=kernel_initializer↩
                ,activation=activation)(join)
165         c2 = keras.layers.Dense(15,kernel_initializer=kernel_initializer↩
                ,activation=activation)(c1)
166         out_layer = keras.layers.Dense(1,activation='sigmoid')(c2)
167         model = keras.models.Model([input_const,input_var],out_layer)
168         # freeze encoder's weights
169         for layer in model.layers[:-1]:
170             layer.trainable = False
171         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
172         run_log_folder = 'inter_N' + str(y_train.shape[0])
173         run_log_folder = os.path.join(log_folder,run_log_folder)
174         # train model
175         model.fit(
176             [x_const_train_,x_var_train_],y_train_,
177             validation_data=([x_const_test_,x_var_test_],y_test_),
178             batch_size=batch_size,epochs=n_epochs,
179             callbacks = [
180                 keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
```

```
181                     keras.callbacks.ReduceLROnPlateau(patience=↩
                            patience_reduce,min_delta=0.00001),
182                     keras.callbacks.EarlyStopping(patience=patience_stop,↩
                            restore_best_weights=True)
183                 ],
184                 verbose=0
185             )
186             # unfreeze encoder's weights
187             for layer in model.layers[:-1]:
188                 layer.trainable = True
189             model.compile(loss='binary_crossentropy',optimizer=keras.↩
                    optimizers.Nadam(learning_rate=learning_rate/10),metrics=['↩
                    accuracy'])
190             run_log_folder = 'final_N'+str(y_train.shape[0])
191             run_log_folder = os.path.join(log_folder,run_log_folder)
192             # train model
193             model.fit(
194                 [x_const_train_,x_var_train_],y_train_,
195                 validation_data=([x_const_test_,x_var_test_],y_test_),
196                 batch_size=batch_size,epochs=n_epochs,
197                 callbacks = [
198                     keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                            =0),
199                     keras.callbacks.ReduceLROnPlateau(patience=↩
                            patience_reduce,min_delta=0.00001),
200                     keras.callbacks.EarlyStopping(patience=patience_stop,↩
                            restore_best_weights=True)
201                 ],
202                 verbose=0
203             )
204             metrics['Time_train'] = time.time()-time_start
205             pred_train = (model.predict([x_const_train_,x_var_train_]) > ↩
                    0.5).astype(int)
206             metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
207             # test model
208             time_start = time.time()
209             pred_test = (model.predict([x_const_test_,x_var_test_]) > 0.5).↩
                    astype(int)
210             metrics['Time_test'] = time.time()-time_start
211             # score
212             metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
213             metrics['Precision'] = precision_score(y_test_,pred_test)
214             metrics['Sensitivity'] = recall_score(y_test_,pred_test)
215             cm = confusion_matrix(y_test_,pred_test)
216             metrics['Specifity'] = cm[0,0] / ( cm[0,1] + cm[0,0] )
217             return metrics

219 # ----------
220 # Run search
221 # ----------
222 used = [1,4,6,9]
223 groups = ['CowId','Lactation']
224 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
225 label_name = 'Lscore'
226 window_length = 27
227 window_shape=(27,3)
228 learning_rate_values = [0.001,0.005,0.01,0.05]
229 batch_size_values = [32,128,512]
230 name = 'ae-cnn_search'
231 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
232 output = name+'_'+ts
233 data_folder = '../data/'
234 log_folder = '../tb_log/'
235 log_folder = os.path.join(log_folder,output)

237 classifier = AeCnnSearcher(used,learning_rate_values,batch_size_values)
```

```
238 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=False)
239 results = classifier.gridsearch(log_folder=log_folder)

241 results.to_csv(output+'.csv')
```

Listing A.22: Skript ae-cnn_eval.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ----------------------
import sys
import os
import time
import pickle
import numpy as np
import pandas as pd
import tensorflow as tf
from datetime import datetime
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractEvaluator2 import AbstractEvaluator2

# ----------------------
# Define specific class
# ----------------------
class AeCnnEvaluator(AbstractEvaluator2):

    def __init__(self,left_out,learning_rate,batch_size):
        self.left_out = left_out
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.X_const_train = None
        self.X_var_train = None
        self.Y_train = None
        self.X_const_test = None
        self.X_var_test = None
        self.Y_test = None
        self.metrics = None

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,x_var_test,y_test,learning_rate,batch_size,log_folder):
        # normalization outside of the model
        norm_const = keras.layers.experimental.preprocessing.Normalization()
        norm_const.adapt(x_const_train)
        x_const_train = norm_const(x_const_train).numpy()
        x_const_test = norm_const(x_const_test).numpy()
        norm_var = keras.layers.experimental.preprocessing.Normalization()
        norm_var.adapt(x_var_train)
        x_var_train = norm_var(x_var_train).numpy()
        x_var_test = norm_var(x_var_test).numpy()
        idx_labels_train = np.nonzero(~np.isnan(y_train.astype(float)))
        idx_labels_test = np.nonzero(~np.isnan(y_test.astype(float)))
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy',
            'Train_Accuracy',
            'Precision',
            'Sensitivity',
            'Specifity',
            'Time_train',
            'Time_test'
        ])
        metrics = {
            'Accuracy' : np.nan,
            'Train_Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
            'Specifity' : np.nan,
```

```
62              'Time_train' : np.nan,
63              'Time_test' : np.nan
64          }
65          kernel_initializer = 'he_normal'
66          activation = 'elu'
67          n_epochs = 1000
68          patience_reduce = 10
69          patience_stop = 50
70          # set random seed for reproducible results
71          tf.random.set_seed(42)
72          # build encoder
73          input_const = keras.layers.Input(shape=x_const_train.shape[1:])
74          layer_const = keras.layers.Dense(2,activation=activation,↩
                kernel_initializer=kernel_initializer)(input_const)
75          input_var = keras.layers.Input(shape=x_var_train.shape[1:])
76          conv1_var = keras.layers.Conv1D(
77              filters=128,kernel_size=8,
78              kernel_initializer=kernel_initializer,
79              use_bias=False,
80              padding='same'
81          )(input_var)
82          bn1_var = keras.layers.BatchNormalization()(conv1_var)
83          act1_var = keras.layers.ELU()(bn1_var)
84          conv2_var = keras.layers.Conv1D(
85              filters=256,kernel_size=5,
86              kernel_initializer=kernel_initializer,
87              use_bias=False,
88              padding='same'
89          )(act1_var)
90          bn2_var = keras.layers.BatchNormalization()(conv2_var)
91          act2_var = keras.layers.ELU()(bn2_var)
92          conv3_var = keras.layers.Conv1D(
93              filters=128,kernel_size=3,
94              kernel_initializer=kernel_initializer,
95              use_bias=False,
96              padding='same'
97          )(act2_var)
98          bn3_var = keras.layers.BatchNormalization()(conv3_var)
99          act3_var = keras.layers.ELU()(bn3_var)
100         features_var = keras.layers.GlobalAveragePooling1D()(act3_var)
101         join = keras.layers.concatenate([layer_const,features_var])
102         # build decoder
103         output_decoder_const = keras.layers.Dense(2)(join)
104         de_reshape1_var = keras.layers.Reshape((1,130))(join)
105         deconv1_var = keras.layers.Conv1DTranspose(
106             filters=128,kernel_size=3,
107             kernel_initializer=kernel_initializer,
108             use_bias=False,
109             padding='valid'
110         )(de_reshape1_var)
111         de_bn1_var = keras.layers.BatchNormalization()(deconv1_var)
112         de_act1_var = keras.layers.ELU()(de_bn1_var)
113         deconv2_var = keras.layers.Conv1DTranspose(
114             filters=256,kernel_size=5,
115             kernel_initializer=kernel_initializer,
116             use_bias=False,
117             padding='valid'
118         )(de_act1_var)
119         de_bn2_var = keras.layers.BatchNormalization()(deconv2_var)
120         de_act2_var = keras.layers.ELU()(de_bn2_var)
121         deconv3_var = keras.layers.Conv1DTranspose(
122             filters=128,kernel_size=8,
123             kernel_initializer=kernel_initializer,
124             use_bias=False,
125             padding='valid'
126         )(de_act2_var)
127         de_bn3_var = keras.layers.BatchNormalization()(deconv3_var)
128         de_act3_var = keras.layers.ELU()(de_bn3_var)
```

```
129         output_decoder_var = keras.layers.Conv1DTranspose(
130             filters=3,kernel_size=14,padding='valid',
131             activation=None, kernel_initializer='glorot_uniform'
132         )(de_act3_var)
133         # combine to autoencoder
134         ae = keras.models.Model([input_const,input_var],[output_decoder_const,output_decoder_var])
135         ae.compile(loss='huber_loss',optimizer=keras.optimizers.Nadam(learning_rate=learning_rate))
136         run_log_folder = 'ae_N'+str(y_train.shape[0])
137         run_log_folder = os.path.join(log_folder,run_log_folder)
138         # train autoencoder
139         time_start = datetime.now()
140         ae.fit(
141             [x_const_train,x_var_train],[x_const_train,x_var_train],
142             validation_data=([x_const_test,x_var_test],[x_const_test,x_var_test]),
143             batch_size=batch_size,epochs=n_epochs,
144             callbacks = [
145                 keras.callbacks.TensorBoard(run_log_folder,profile_batch=0),
146                 keras.callbacks.ReduceLROnPlateau(patience=patience_reduce,min_delta=0.00001),
147                 keras.callbacks.EarlyStopping(patience=patience_stop,restore_best_weights=True)
148             ],
149             verbose=0
150         )
151         # reduce to labeled data
152         x_const_train_ = x_const_train[idx_labels_train]
153         x_var_train_ = x_var_train[idx_labels_train]
154         x_const_test_ = x_const_test[idx_labels_test]
155         x_var_test_ = x_var_test[idx_labels_test]
156         y_train_ = (y_train[idx_labels_train]).astype(int)
157         y_test_ = (y_test[idx_labels_test]).astype(int)
158         # build classifier
159         c1 = keras.layers.Dense(15,kernel_initializer=kernel_initializer,activation=activation)(join)
160         c2 = keras.layers.Dense(15,kernel_initializer=kernel_initializer,activation=activation)(c1)
161         out_layer = keras.layers.Dense(1,activation='sigmoid')(c2)
162         model = keras.models.Model([input_const,input_var],out_layer)
163         # freeze encoder's weights
164         for layer in model.layers[:-1]:
165             layer.trainable = False
166         model.compile(loss='binary_crossentropy',optimizer=keras.optimizers.Nadam(learning_rate=learning_rate),metrics=['accuracy'])
167         run_log_folder = 'inter_N'+str(y_train.shape[0])
168         run_log_folder = os.path.join(log_folder,run_log_folder)
169         # train model
170         model.fit(
171             [x_const_train_,x_var_train_],y_train_,
172             validation_data=([x_const_test_,x_var_test_],y_test_),
173             batch_size=batch_size,epochs=n_epochs,
174             callbacks = [
175                 keras.callbacks.TensorBoard(run_log_folder,profile_batch=0),
176                 keras.callbacks.ReduceLROnPlateau(patience=patience_reduce,min_delta=0.00001),
177                 keras.callbacks.EarlyStopping(patience=patience_stop, restore_best_weights=True)
178             ],
179             verbose=0
180         )
181         # unfreeze encoder's weights
182         for layer in model.layers[:-1]:
183             layer.trainable = True
```

```
184         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate/10),metrics=['↩
                accuracy'])
185         run_log_folder = 'final_N' + str(y_train.shape[0])
186         run_log_folder = os.path.join(log_folder,run_log_folder)
187         # train model 1st time
188         model.fit(
189             [x_const_train_,x_var_train_],y_train_,
190             validation_data=([x_const_test_,x_var_test_],y_test_),
191             batch_size=batch_size,epochs=n_epochs,
192             callbacks = [
193                 keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
194                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
195                 keras.callbacks.EarlyStopping(patience=patience_stop, ↩
                        restore_best_weights=True)
196             ],
197             verbose=0
198         )
199         metrics['Time_train'] = datetime.now()-time_start
200         pred_train = (model.predict([x_const_train_,x_var_train_]) > ↩
                0.5).astype(int)
201         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
202         # test model
203         time_start = datetime.now()
204         pred_test = (model.predict([x_const_test_,x_var_test_]) > 0.5).↩
                astype(int)
205         metrics['Time_test'] = datetime.now()-time_start
206         # score
207         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
208         metrics['Precision'] = precision_score(y_test_,pred_test)
209         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
210         cm = confusion_matrix(y_test_,pred_test)
211         metrics['Specifity'] = cm[0,0] / ( cm[0,1] + cm[0,0] )
212         result = metrics.copy()
213         result['learning_rate'] = learning_rate
214         result['batch_size'] = batch_size
215         self.metrics = self.metrics.append(result, ignore_index = True)
216         return metrics

218 # --------------
219 # Run evaluation
220 # --------------
221 left_out = int(sys.argv[1])
222 groups = ['CowId','Lactation']
223 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
224 label_name = 'Lscore'
225 window_length = 27
226 window_shape= (27,3)
227 learning_rate = 0.001
228 batch_size = 32
229 name = 'ae-cnn_eval'+str(left_out)
230 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
231 output = name+'_'+ts
232 data_folder = '../data/'
233 log_folder = '../tb_log/'
234 log_folder = os.path.join(log_folder, output)

236 classifier = AeCnnEvaluator(left_out,learning_rate,batch_size)
237 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=False)
238 result = classifier.prepare_fit(x_const_train=classifier.X_const_train,↩
        x_var_train=classifier.X_var_train,y_train=classifier.Y_train,↩
        x_const_test=classifier.X_const_test,x_var_test=classifier.X_var_test↩
```

```
        ,y_test=classifier.Y_test,learning_rate=learning_rate,batch_size=↩
        batch_size,log_folder=log_folder)

240 print(f"\nEvaluate␣AE-CNN␣on␣left_out={left_out}␣with␣parameters:\n↩
        ----------------------\n\n")
241 print(f"learning_rate␣=␣{learning_rate}")
242 print(f"batch_size␣=␣{batch_size}")
243 print('\n-----\n')
244 print(f"Output:␣{output}")
245 print('\n-----\n')
246 print(f"Accuracy␣:␣{result['Accuracy']}␣(Training␣accuracy␣:␣{result['↩
        Train_Accuracy']})")
247 print(f"Precision␣:␣{result['Precision']}")
248 print(f"Sensitivity␣:␣{result['Sensitivity']}")
249 print(f"Specifity␣:␣{result['Specifity']}")
250 print(f"Time_train␣:␣{result['Time_train']}")
251 print(f"Time_test␣:␣{result['Time_test']}")
```

Listing A.23: Skript ae-gru_search.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import os
import time
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import LeaveOneGroupOut
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractSearcher2 import AbstractSearcher2

# ---------------------
# Define specific class
# ---------------------
class AeGruSearcher(AbstractSearcher2):

    def __init__(self,used,learning_rate_values,batch_size_values):
        self.used = used
        self.learning_rate_values = learning_rate_values
        self.batch_size_values = batch_size_values
        self.logo = LeaveOneGroupOut()
        self.X_const = None
        self.X_var = None
        self.Y = None
        self.Split = None
        self.metrics = None

    def gridsearch(self,**kwargs):
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy_mean', 'Accuracy_sd',
            'Precision_mean', 'Precision_sd',
            'Sensitivity_mean', 'Sensitivity_sd',
            'Specifity_mean', 'Specifity_sd',
            'Time_train_mean', 'Time_train_sd',
            'Time_test_mean', 'Time_test_sd'
        ])
        for learning_rate in self.learning_rate_values:
            for batch_size in self.batch_size_values:
                cv_result = self.crossvalidate(learning_rate=learning_rate,batch_size=batch_size,log_folder=log_folder)
                cv_result['learning_rate'] = learning_rate
                cv_result['batch_size'] = batch_size
                self.metrics = self.metrics.append(cv_result, ignore_index = True)
        self.metrics = self.metrics.sort_values(by=['Accuracy_mean'],ascending=False)
        return self.metrics

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,x_var_test,y_test,learning_rate,batch_size,log_folder,**kwargs):
        # normalization outside of the model
        norm_const = keras.layers.experimental.preprocessing.Normalization()
        norm_const.adapt(x_const_train)
        x_const_train = norm_const(x_const_train).numpy()
        x_const_test = norm_const(x_const_test).numpy()
        norm_var = keras.layers.experimental.preprocessing.Normalization()
        norm_var.adapt(x_var_train)
```

```
58          x_var_train = norm_var(x_var_train).numpy()
59          x_var_test = norm_var(x_var_test).numpy()
60          idx_labels_train = np.nonzero(~np.isnan(y_train.astype(float)))
61          idx_labels_test = np.nonzero(~np.isnan(y_test.astype(float)))
62          metrics = {
63              'Accuracy' : np.nan,
64              'Precision' : np.nan,
65              'Sensitivity' : np.nan,
66              'Specifity' : np.nan,
67              'Time_train' : np.nan,
68              'Time_test' : np.nan
69          }
70          n_epochs = 1000
71          patience_reduce = 10
72          patience_stop = 50
73          # set random seed for reproducible results
74          tf.random.set_seed(42)
75          # build encoder
76          input_const = keras.layers.Input(shape=x_const_train.shape[1:])
77          layer_const = keras.layers.Dense(2,activation='elu',↩
                kernel_initializer='he_normal')(input_const)
78          input_var = keras.layers.Input(shape=x_var_train.shape[1:])
79          rec1_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                input_var)
80          rec2_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                rec1_var)
81          rec3_var = keras.layers.GRU(units=16)(rec2_var)
82          join = keras.layers.concatenate([layer_const,rec3_var])
83          dense =keras.layers.Dense(6,activation='elu',kernel_initializer=↩
                'he_normal')(join)
84          # build decoder
85          output_decoder_const = keras.layers.Dense(2)(dense)
86          de_rep = keras.layers.RepeatVector(27)(dense)
87          de_rec1_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                de_rep)
88          de_rec2_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                de_rec1_var)
89          de_rec3_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                de_rec2_var)
90          output_decoder_var = keras.layers.TimeDistributed(keras.layers.↩
                Dense(3))(de_rec3_var)
91          # combine to autoencoder
92          ae = keras.models.Model([input_const,input_var],[↩
                output_decoder_const,output_decoder_var])
93          ae.compile(loss='huber_loss',optimizer=keras.optimizers.Nadam(↩
                learning_rate=learning_rate))
94          run_log_folder = 'ae_N'+str(y_train.shape[0])
95          run_log_folder = os.path.join(log_folder,run_log_folder)
96          # train autoencoder
97          time_start = time.time()
98          ae.fit(
99              [x_const_train,x_var_train],[x_const_train,x_var_train],
100             validation_data=([x_const_test,x_var_test],[x_const_test,↩
                    x_var_test]),
101             batch_size=batch_size,epochs=n_epochs,
102             callbacks = [
103                 keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
104                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
105                 keras.callbacks.EarlyStopping(patience=patience_stop, ↩
                        restore_best_weights=True)
106             ],
107             verbose=0
108         )
109         # reduce to labeled data
110         x_const_train_ = x_const_train[idx_labels_train]
111         x_var_train_ = x_var_train[idx_labels_train]
```

```
112         x_const_test_ = x_const_test[idx_labels_test]
113         x_var_test_ = x_var_test[idx_labels_test]
114         y_train_ = (y_train[idx_labels_train]).astype(int)
115         y_test_ = (y_test[idx_labels_test]).astype(int)
116         # build classifier
117         c1 = keras.layers.Dense(4,kernel_initializer='he_normal',↩
                activation='elu')(dense)
118         c2 = keras.layers.Dense(4,kernel_initializer='he_normal',↩
                activation='elu')(c1)
119         out_layer = keras.layers.Dense(1,activation='sigmoid')(c2)
120         model = keras.models.Model([input_const,input_var],out_layer)
121         # freeze encoder's weights
122         for layer in model.layers[:-1]:
123             layer.trainable = False
124         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
125         run_log_folder = 'inter_N' + str(y_train.shape[0])
126         run_log_folder = os.path.join(log_folder,run_log_folder)
127         # train model 1st time
128         model.fit(
129             [x_const_train_,x_var_train_],y_train_,
130             validation_data=([x_const_test_,x_var_test_],y_test_),
131             batch_size=batch_size,epochs=n_epochs,
132             callbacks = [
133                 keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
134                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
135                 keras.callbacks.EarlyStopping(patience=patience_stop,↩
                        restore_best_weights=True)
136             ],
137             verbose=0
138         )
139         # unfreeze encoder's weights
140         for layer in model.layers[:-1]:
141             layer.trainable = True
142         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate/10),metrics=['↩
                accuracy'])
143         run_log_folder = 'final_N'+str(y_train.shape[0])
144         run_log_folder = os.path.join(log_folder,run_log_folder)
145         # train model finally
146         model.fit(
147             [x_const_train_,x_var_train_],y_train_,
148             validation_data=([x_const_test_,x_var_test_],y_test_),
149             batch_size=batch_size,epochs=n_epochs,
150             callbacks = [
151                 keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
152                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
153                 keras.callbacks.EarlyStopping(patience=patience_stop, ↩
                        restore_best_weights=True)
154             ],
155             verbose=0
156         )
157         metrics['Time_train'] = time.time()-time_start
158         pred_train = (model.predict([x_const_train_,x_var_train_]) > ↩
                0.5).astype(int)
159         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
160         # test model
161         time_start = time.time()
162         pred_test = (model.predict([x_const_test_,x_var_test_]) > 0.5).↩
                astype(int)
163         metrics['Time_test'] = time.time()-time_start
164         # score
165         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
```

```
166         metrics['Precision'] = precision_score(y_test_,pred_test)
167         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
168         cm = confusion_matrix(y_test_,pred_test)
169         metrics['Specifity'] = cm[0,0] / ( cm[0,1] + cm[0,0] )
170         return metrics

172 # ----------
173 # Run search
174 # ----------
175 used = [1,4,6,9]
176 groups = ['CowId','Lactation']
177 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
178 label_name = 'Lscore'
179 window_length = 27
180 window_shape=(27,3)
181 learning_rate_values = [0.001,0.005,0.01,0.05]
182 batch_size_values = [32,128,512]
183 name = 'ae-gru_search'
184 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
185 output = name+'_'+ts
186 data_folder = '../data/'
187 log_folder = '../tb_log/'
188 log_folder = os.path.join(log_folder,output)

190 classifier = AeGruSearcher(used,learning_rate_values,batch_size_values)
191 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=False)
192 results = classifier.gridsearch(log_folder=log_folder)

194 results.to_csv(output+'.csv')
```

Listing A.24: Skript ae-gru_eval.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# ---------------------
import sys
import os
import time
import pickle
import numpy as np
import pandas as pd
import tensorflow as tf
from datetime import datetime
from tensorflow import keras
from sklearn.metrics import accuracy_score,precision_score,recall_score,confusion_matrix
from src.AbstractEvaluator2 import AbstractEvaluator2

# ---------------------
# Define specific class
# ---------------------
class AeGruEvaluator(AbstractEvaluator2):

    def __init__(self,left_out,learning_rate,batch_size):
        self.left_out = left_out
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.X_const_train = None
        self.X_var_train = None
        self.Y_train = None
        self.X_const_test = None
        self.X_var_test = None
        self.Y_test = None
        self.metrics = None

    def prepare_fit(self,x_const_train,x_var_train,y_train,x_const_test,x_var_test,y_test,learning_rate,batch_size,log_folder):
        # normalization outside of the model
        norm_const = keras.layers.experimental.preprocessing.Normalization()
        norm_const.adapt(x_const_train)
        x_const_train = norm_const(x_const_train).numpy()
        x_const_test = norm_const(x_const_test).numpy()
        norm_var = keras.layers.experimental.preprocessing.Normalization()
        norm_var.adapt(x_var_train)
        x_var_train = norm_var(x_var_train).numpy()
        x_var_test = norm_var(x_var_test).numpy()
        idx_labels_train = np.nonzero(~np.isnan(y_train.astype(float)))
        idx_labels_test = np.nonzero(~np.isnan(y_test.astype(float)))
        self.metrics = pd.DataFrame(columns= [
            'learning_rate', 'batch_size',
            'Accuracy',
            'Train_Accuracy',
            'Precision',
            'Sensitivity',
            'Specifity',
            'Time_train',
            'Time_test'
        ])
        metrics = {
            'Accuracy' : np.nan,
            'Train_Accuracy' : np.nan,
            'Precision' : np.nan,
            'Sensitivity' : np.nan,
            'Specifity' : np.nan,
```

```
62              'Time_train' : np.nan,
63              'Time_test' : np.nan
64          }
65          n_epochs = 1000
66          patience_reduce = 10
67          patience_stop = 50
68          # set random seed for reproducible results
69          tf.random.set_seed(42)
70          # build encoder
71          input_const = keras.layers.Input(shape=x_const_train.shape[1:])
72          layer_const = keras.layers.Dense(2,activation='elu',↩
                kernel_initializer='he_normal')(input_const)
73          input_var = keras.layers.Input(shape=x_var_train.shape[1:])
74          rec1_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                input_var)
75          rec2_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                rec1_var)
76          rec3_var = keras.layers.GRU(units=16)(rec2_var)
77          join = keras.layers.concatenate([layer_const,rec3_var])
78          dense = keras.layers.Dense(6,activation='elu',kernel_initializer↩
                ='he_normal')(join)
79          # build decoder
80          output_decoder_const = keras.layers.Dense(2)(dense)
81          de_rep = keras.layers.RepeatVector(27)(dense)
82          de_rec1_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                de_rep)
83          de_rec2_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                de_rec1_var)
84          de_rec3_var = keras.layers.GRU(units=16,return_sequences=True)(↩
                de_rec2_var)
85          output_decoder_var = keras.layers.TimeDistributed(keras.layers.↩
                Dense(3))(de_rec3_var)
86          # combine to autoencoder
87          ae = keras.models.Model([input_const,input_var],[↩
                output_decoder_const,output_decoder_var])
88          ae.compile(loss='huber_loss',optimizer=keras.optimizers.Nadam(↩
                learning_rate=learning_rate))
89          run_log_folder = 'ae_N'+str(y_train.shape[0])
90          run_log_folder = os.path.join(log_folder,run_log_folder)
91          # train autoencoder
92          time_start = datetime.now()
93          ae.fit(
94              [x_const_train,x_var_train],[x_const_train,x_var_train],
95              validation_data=([x_const_test,x_var_test],[x_const_test,↩
                    x_var_test]),
96              batch_size=batch_size,epochs=n_epochs,
97              callbacks = [
98                  keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
99                  keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
100                 keras.callbacks.EarlyStopping(patience=patience_stop,↩
                        restore_best_weights=True)
101             ],
102             verbose=0
103         )
104         # reduce to labeled data
105         x_const_train_ = x_const_train[idx_labels_train]
106         x_var_train_ = x_var_train[idx_labels_train]
107         x_const_test_ = x_const_test[idx_labels_test]
108         x_var_test_ = x_var_test[idx_labels_test]
109         y_train_ = (y_train[idx_labels_train]).astype(int)
110         y_test_ = (y_test[idx_labels_test]).astype(int)
111         # build classifier
112         c1 = keras.layers.Dense(4,kernel_initializer='he_normal',↩
                activation='elu')(dense)
113         c2 = keras.layers.Dense(4,kernel_initializer='he_normal',↩
                activation='elu')(c1)
```

```
114         out_layer = keras.layers.Dense(1,activation='sigmoid')(c2)
115         model = keras.models.Model([input_const,input_var],out_layer)
116         # freeze encoder's weights
117         for layer in model.layers[:-1]:
118             layer.trainable = False
119         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate),metrics=['↩
                accuracy'])
120         run_log_folder = 'inter_N'+str(y_train.shape[0])
121         run_log_folder = os.path.join(log_folder, run_log_folder)
122         # train model 1st time
123         model.fit(
124             [x_const_train_,x_var_train_],y_train_,
125             validation_data=([x_const_test_,x_var_test_],y_test_),
126             batch_size=batch_size,epochs=n_epochs,
127             callbacks = [
128                 keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
129                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
130                 keras.callbacks.EarlyStopping(patience=patience_stop,↩
                        restore_best_weights=True)
131             ],
132             verbose=0
133         )
134         # unfreeze encoder's weights
135         for layer in model.layers[:-1]:
136             layer.trainable = True
137         model.compile(loss='binary_crossentropy',optimizer=keras.↩
                optimizers.Nadam(learning_rate=learning_rate/10),metrics=['↩
                accuracy'])
138         run_log_folder = 'final_N'+str(y_train.shape[0])
139         run_log_folder = os.path.join(log_folder, run_log_folder)
140         # train model finally
141         model.fit(
142             [x_const_train_,x_var_train_],y_train_,
143             validation_data=([x_const_test_,x_var_test_],y_test_),
144             batch_size=batch_size,epochs=n_epochs,
145             callbacks = [
146                 keras.callbacks.TensorBoard(run_log_folder,profile_batch↩
                        =0),
147                 keras.callbacks.ReduceLROnPlateau(patience=↩
                        patience_reduce,min_delta=0.00001),
148                 keras.callbacks.EarlyStopping(patience=patience_stop, ↩
                        restore_best_weights=True)
149             ],
150             verbose=0
151         )
152         metrics['Time_train'] = datetime.now()-time_start
153         pred_train = (model.predict([x_const_train_,x_var_train_]) > ↩
                0.5).astype(int)
154         metrics['Train_Accuracy'] = accuracy_score(y_train_,pred_train)
155         # test model
156         time_start = datetime.now()
157         pred_test = (model.predict([x_const_test_,x_var_test_]) > 0.5).↩
                astype(int)
158         metrics['Time_test'] = datetime.now()-time_start
159         # score
160         metrics['Accuracy'] = accuracy_score(y_test_,pred_test)
161         metrics['Precision'] = precision_score(y_test_,pred_test)
162         metrics['Sensitivity'] = recall_score(y_test_,pred_test)
163         cm = confusion_matrix(y_test_,pred_test)
164         metrics['Specifity'] = cm[0,0] / ( cm[0,1] + cm[0,0] )
165         result = metrics.copy()
166         result['learning_rate'] = learning_rate
167         result['batch_size'] = batch_size
168         self.metrics = self.metrics.append(result, ignore_index = True)
169         return metrics
```

```
171 # ---------------
172 # Run evaluation
173 # ---------------
174 left_out = int(sys.argv[1])
175 groups = ['CowId','Lactation']
176 feature_names = ['Lactation','DaysInMilk','MilkYield','StepsPerHour','↩
        LyingDuration']
177 label_name = 'Lscore'
178 window_length = 27
179 window_shape= (27,3)
180 learning_rate = 0.005
181 batch_size = 32
182 name = 'ae-gru_eval'+str(left_out)
183 ts = time.strftime('%Y-%m-%d_%H%M%S',time.localtime())
184 output = name+'_'+ts
185 data_folder = '../data/'
186 log_folder = '../tb_log/'
187 log_folder = os.path.join(log_folder, output)

189 classifier = AeGruEvaluator(left_out,learning_rate,batch_size)
190 classifier.import_data(data_folder,n_proc=4,groups=groups,feature_names=↩
        feature_names,label_name=label_name,window_length=window_length,↩
        window_shape=window_shape,idx_constants=[0,1],idx_time_deps=[2,3,4],↩
        labelled_only=False)
191 result = classifier.prepare_fit(x_const_train=classifier.X_const_train,↩
        x_var_train=classifier.X_var_train,y_train=classifier.Y_train,↩
        x_const_test=classifier.X_const_test,x_var_test=classifier.X_var_test↩
        ,y_test=classifier.Y_test,learning_rate=learning_rate,batch_size=↩
        batch_size,log_folder=log_folder)

193 print(f"\nEvaluate␣AE-GRU␣on␣left_out={left_out}␣with␣parameters:\n↩
        ----------------------\n\n")
194 print(f"learning_rate␣=␣{learning_rate}")
195 print(f"batch_size␣=␣{batch_size}")
196 print('\n-----\n')
197 print(f"Output:␣{output}")
198 print('\n-----\n')
199 print(f"Accuracy␣:␣{result['Accuracy']}␣(Training␣accuracy␣:␣{result['↩
        Train_Accuracy']})")
200 print(f"Precision␣:␣{result['Precision']}")
201 print(f"Sensitivity␣:␣{result['Sensitivity']}")
202 print(f"Specifity␣:␣{result['Specifity']}")
203 print(f"Time_train␣:␣{result['Time_train']}")
204 print(f"Time_test␣:␣{result['Time_test']}")
```

B

Einzelergebnisse

Tabelle B.1: Einzelergebnisse der Evaluierung aller neun Klassifikationsansätze

Test-datensatz	Ansatz	Genauigkeit (bei der Klassifikation von Testdaten)	Relevanz	Sensitivität	Spezifität	Trainings-dauer [sec]	Vorhersage-dauer [sec]	Arbeitspeicher-bedarf [MB]
Teil 0	FE-RF	0,7406	0,7778	0,6578	0,8199	128,62	1,41	639,86
Teil 1	FE-RF	0,7315	0,7433	0,6974	0,7648	126,72	2,30	639,73
Teil 2	FE-RF	0,7256	0,7012	0,6490	0,7851	125,07	2,59	639,95
Teil 3	FE-RF	0,6986	0,6828	0,6258	0,7589	123,32	4,26	639,94
Teil 4	FE-RF	0,7195	0,7057	0,6638	0,7664	125,90	3,51	639,75
Teil 5	FE-RF	0,6974	0,7262	0,6356	0,7595	122,48	4,69	639,79
Teil 6	FE-RF	0,7234	0,7632	0,6707	0,7792	123,17	4,30	629,43
Teil 7	FE-RF	0,7219	0,6921	0,6580	0,7718	124,49	3,99	639,80
Teil 8	FE-RF	0,6658	0,7115	0,5765	0,7582	122,55	2,79	639,87
Teil 9	FE-RF	0,7081	0,7226	0,6284	0,7805	122,56	4,24	639,96
Teil 0	FE-MLP	0,7392	0,7581	0,6858	0,7903	30,55	0,05	671,05
Teil 1	FE-MLP	0,7084	0,7254	0,6595	0,7562	55,50	0,04	688,07
Teil 2	FE-MLP	0,7112	0,6720	0,6633	0,7483	23,20	0,05	757,87
Teil 3	FE-MLP	0,6907	0,6715	0,6218	0,7478	43,20	0,04	635,15
Teil 4	FE-MLP	0,7214	0,7046	0,6737	0,7616	19,17	0,05	757,90
Teil 5	FE-MLP	0,6974	0,7127	0,6633	0,7316	13,91	0,10	757,98
Teil 6	FE-MLP	0,7001	0,7295	0,6634	0,7390	40,54	0,04	631,78
Teil 7	FE-MLP	0,7435	0,7085	0,7043	0,7740	17,71	0,05	757,95
Teil 8	FE-MLP	0,6901	0,7435	0,5961	0,7873	31,76	0,05	736,25
Teil 9	FE-MLP	0,6976	0,6974	0,6452	0,7453	20,16	0,04	705,55
Teil 0	FE-SVM	0,7447	0,7856	0,6578	0,8280	41,06	4,38	489,79
Teil 1	FE-SVM	0,7265	0,7418	0,6847	0,7672	38,42	6,04	509,61
Teil 2	FE-SVM	0,7187	0,6855	0,6590	0,7650	38,87	4,89	506,61
Teil 3	FE-SVM	0,7047	0,6891	0,6353	0,7623	28,67	4,41	511,97
Teil 4	FE-SVM	0,7272	0,7200	0,6610	0,7831	41,09	5,05	511,98
Teil 5	FE-SVM	0,7151	0,7362	0,6721	0,7582	29,23	3,86	511,79
Teil 6	FE-SVM	0,7076	0,7504	0,6475	0,7714	46,76	4,94	511,80
Teil 7	FE-SVM	0,7397	0,7174	0,6696	0,7944	29,52	4,46	511,97
Teil 8	FE-SVM	0,6988	0,7410	0,6267	0,7734	52,39	5,16	511,92
Teil 9	FE-SVM	0,7050	0,7218	0,6194	0,7829	42,76	5,14	511,74
Teil 0	E2E-CNN	0,7447	0,7603	0,6985	0,7890	124,36	0,06	1.372,16
Teil 1	E2E-CNN	0,7134	0,7335	0,6595	0,7660	67,54	0,06	1.433,60
Teil 2	E2E-CNN	0,7212	0,7024	0,6289	0,7929	87,47	0,07	1.423,36
Teil 3	E2E-CNN	0,7016	0,6789	0,6487	0,7455	76,01	0,06	1.392,64
Teil 4	E2E-CNN	0,7162	0,7060	0,6511	0,7712	82,87	0,06	1.413,12
Teil 5	E2E-CNN	0,6968	0,6989	0,6936	0,7000	83,05	0,15	1.433,60
Teil 6	E2E-CNN	0,7240	0,7269	0,7430	0,7039	96,74	0,06	1.433,60
Teil 7	E2E-CNN	0,7403	0,7377	0,6319	0,8249	63,83	0,06	1.402,88
Teil 8	E2E-CNN	0,7100	0,6931	0,7711	0,6468	83,11	0,07	1.433,60
Teil 9	E2E-CNN	0,7117	0,7029	0,6839	0,7371	101,46	0,07	1.402,88
Teil 0	E2E-MLP	0,6836	0,6858	0,6522	0,7137	14,84	0,05	1.009,25
Teil 1	E2E-MLP	0,6766	0,6779	0,6583	0,6946	11,94	0,05	1.034,24
Teil 2	E2E-MLP	0,6768	0,6302	0,6032	0,7249	10,39	0,04	1.044,48
Teil 3	E2E-MLP	0,6461	0,6204	0,5653	0,7132	10,09	0,04	1.003,28
Teil 4	E2E-MLP	0,6852	0,6550	0,6596	0,7068	11,00	0,04	1.034,24
Teil 5	E2E-MLP	0,6469	0,6616	0,6040	0,6999	9,66	0,08	1.054,72
Teil 6	E2E-MLP	0,6805	0,6967	0,6720	0,6896	10,65	0,04	1.010,62
Teil 7	E2E-MLP	0,6908	0,6352	0,6913	0,6904	10,84	0,04	1.034,24
Teil 8	E2E-MLP	0,6559	0,6808	0,6083	0,7051	6,79	0,04	367,93
Teil 9	E2E-MLP	0,6601	0,6480	0,6271	0,6901	7,53	0,04	359,26

Fortsetzung auf der nächsten Seite...

Fortsetzung: Einzelergebnisse der Evaluierung

Test-datensatz	Ansatz	Genauigkeit (bei der Klassifikation von Testdaten)	Relevanz (bei der Klassifikation von Testdaten)	Sensitivität (bei der Klassifikation von Testdaten)	Spezifität (bei der Klassifikation von Testdaten)	Trainings-dauer [sec]	Vorhersage-dauer [sec]	Arbeitspeicher-bedarf [MB]
Teil 0	E2E-GRU	0,7563	0,7754	0,7069	0,8038	210,73	0,10	936,16
Teil 1	E2E-GRU	0,7053	0,7210	0,6583	0,7512	209,78	0,11	969,86
Teil 2	E2E-GRU	0,7206	0,6898	0,6562	0,7706	198,13	0,13	955,60
Teil 3	E2E-GRU	0,7023	0,6900	0,6231	0,7679	213,35	0,12	912,61
Teil 4	E2E-GRU	0,7253	0,7096	0,6766	0,7664	198,35	1,36	930,36
Teil 5	E2E-GRU	0,6999	0,7190	0,6583	0,7418	225,85	0,74	924,20
Teil 6	E2E-GRU	0,7234	0,7554	0,6842	0,7649	202,88	0,13	932,50
Teil 7	E2E-GRU	0,7384	0,7264	0,6464	0,8102	183,59	0,11	911,35
Teil 8	E2E-GRU	0,6845	0,6997	0,6646	0,7051	196,29	0,11	916,17
Teil 9	E2E-GRU	0,7222	0,7443	0,6348	0,8016	196,79	0,11	920,46
Teil 0	AE-MLP	0,6918	0,6958	0,6578	0,7245	2.917,96	0,04	2.375,68
Teil 1	AE-MLP	0,6835	0,6923	0,6469	0,7192	2.449,11	0,04	2.222,08
Teil 2	AE-MLP	0,6585	0,6154	0,5845	0,7160	3.115,96	0,04	2.181,12
Teil 3	AE-MLP	0,6638	0,6364	0,6030	0,7143	3.353,01	0,04	2.242,56
Teil 4	AE-MLP	0,6858	0,6796	0,5932	0,7640	2.902,78	0,04	2.232,32
Teil 5	AE-MLP	0,6532	0,6758	0,5914	0,7152	2.250,80	0,09	2.222,08
Teil 6	AE-MLP	0,6830	0,6982	0,6769	0,6896	1.760,11	0,04	2.498,56
Teil 7	AE-MLP	0,6997	0,6598	0,6493	0,7390	2.405,75	0,04	2.211,84
Teil 8	AE-MLP	0,6509	0,6793	0,5936	0,7101	2.277,88	0,05	2.222,08
Teil 9	AE-MLP	0,6601	0,6762	0,5497	0,7606	2.367,39	0,05	2.181,12
Teil 0	AE-CNN	0,7495	0,7762	0,6858	0,8105	22.085,27	0,06	2.539,52
Teil 1	AE-CNN	0,7146	0,7207	0,6898	0,7389	19.331,36	0,06	2.549,76
Teil 2	AE-CNN	0,7080	0,6853	0,6146	0,7806	16.544,54	0,06	2.478,08
Teil 3	AE-CNN	0,7084	0,6879	0,6528	0,7545	18.317,98	0,07	2.498,56
Teil 4	AE-CNN	0,7214	0,7426	0,5989	0,8248	18.119,71	0,06	2.549,76
Teil 5	AE-CNN	0,7031	0,7073	0,6948	0,7114	14.068,50	0,15	2.549,76
Teil 6	AE-CNN	0,7070	0,7056	0,7393	0,6727	11.677,27	0,07	2.478,08
Teil 7	AE-CNN	0,7549	0,7397	0,6797	0,8136	17.133,42	0,07	2.529,28
Teil 8	AE-CNN	0,6882	0,6872	0,7100	0,6658	14.508,56	0,06	2.478,08
Teil 9	AE-CNN	0,7179	0,7232	0,6606	0,7700	18.921,29	0,06	2.488,32
Teil 0	AE-GRU	0,7296	0,7406	0,6886	0,7688	960,17	0,10	2.836,48
Teil 1	AE-GRU	0,7016	0,7211	0,6456	0,7562	1.140,34	0,13	2.836,48
Teil 2	AE-GRU	0,7036	0,6633	0,6547	0,7416	1.012,93	0,13	2.826,24
Teil 3	AE-GRU	0,6693	0,6558	0,5693	0,7522	2.113,63	0,11	2.908,16
Teil 4	AE-GRU	0,7175	0,6950	0,6822	0,7473	1.014,77	0,11	2.928,64
Teil 5	AE-GRU	0,6999	0,7087	0,6810	0,7190	1.643,92	0,69	2.949,12
Teil 6	AE-GRU	0,7089	0,7273	0,6952	0,7234	2.418,54	0,11	2.836,48
Teil 7	AE-GRU	0,7435	0,7248	0,6681	0,8023	2.441,71	0,11	2.928,64
Teil 8	AE-GRU	0,6920	0,7152	0,6548	0,7304	924,58	0,11	2.795,52
Teil 9	AE-GRU	0,7124	0,7072	0,6761	0,7453	1.020,26	0,11	2.918,40