h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

Fachbereiche Mathematik und
Naturwissenschaften und Informatik

Ensemble-basiertes Handling von Concept Drifts durch Messung der Modell-Unsicherheit

Abschlussarbeit zur Erlangung des
akademischen Grades

Master of Science (M.Sc.)

vorgelegt von

Paula Möller

Matrikelnummer: 767782

Referent	:	Prof. Dr. Arnim Malcherek
Korreferent	:	Prof. Dr. Antje Jahn-Eimermacher

Paula Möller: *Ensemble-basiertes Handling von Concept Drifts durch Messung der Modell-Unsicherheit*, © 19. Januar 2022

ERKLÄRUNG

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

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

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

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

Frankfurt am Main, 19. Januar 2022

Paula Möller

ABSTRACT

We live in a world that is in a constant state of change. This applies not at least to predictive models in machine learning or artificial intelligence. The change of our environment is described by concept drifts.
For detection of concept drifts, many methods need true labels. However, computing the deviation of the predictions from the true values can take some time, since labels in a stream are not always available immediately and sometimes not completely available. To avoid making predictions with a model that is outdated for the current time, unsupervised methods are often used. However, when these approaches rely only on monitoring the distribution of input features, certain types of concept drifts cannot be detected.
Therefore, in this method, the predictive uncertainty of the model is measured by an ensemble and used as an indicator of a concept drift. After a detection, an adjustment takes place by retraining the whole ensemble or only parts of the ensemble on data of the last context. Comparisons are made between the detection and adaptation of an ensemble of neural networks to a neural network using Monte Carlo Dropouts on a range of synthetic and real-world data sets addressing both regression and classification problems. It is shown that, in contrast to Monte Carlo Dropouts in a neural network, neural network ensembles not only output the more conservative estimate of predictive uncertainty as evidenced in numerous studies, but are also better suited in dealing with data in which concept drifts are present.

Keywords— Concept Drift, Detection, Handling, Model Uncertainty, Ensembles, Monte Carlo Dropouts, Streaming

ZUSAMMENFASSUNG

Wir leben in einer Welt, die sich in einem ständigen Wandel befindet. Dies betrifft nicht zuletzt prädiktive Modelle aus dem Machine Learning bzw. der Künstlichen Intelligenz. Die Veränderung unserer Umwelt wird durch Concept Drifts beschrieben.
Für die Detektion von Concept Drifts braucht es in vielen Methoden die wahren Labels. Die Abweichung der Vorhersagen zu den wahren Werten zu berechnen kann aber einige Zeit dauern, da Labels in einem Stream nicht immer sofort und erst recht nicht vollständig vorliegen. Um nicht mit einem für den aktuellen Zeitpunkt veralteten Modell Vorhersagen zu treffen, werden häufig unüberwachte Methoden verwendet. Wenn diese Ansätze nur auf der Überwachung der Verteilung der Input Features beruhen, können bestimmte Arten von Concept Drifts jedoch nicht detektiert werden.
Daher wird in dieser Methode die prädiktive Unsicherheit des Modells durch ein Ensemble gemessen und als Indikator für einen Concept Drift verwendet. Nach einer Detektion findet gleichzeitig noch eine Anpassung statt, indem das gesamte Ensemble oder nur Teile des Ensembles neu an die zuletzt aktuelle Situation angepasst werden. Es wurden Vergleiche zwischen der Detektion und Anpassung von einem Ensemble aus Neuronalen Netzen zu einem Neuronalen Netz unter Verwendung von Monte-Carlo Dropouts gezogen. Diese wurden auf einer Reihe von synthetischen und Real-World Datensätzen durchgeführt, die sowohl Regressions- als auch Klassifikations-Probleme adressieren. Es wurde gezeigt, dass Ensembles aus Neuronalen Netzen im Gegensatz zu Monte-Carlo Dropouts in einem Neuronalen Netz nicht nur wie in zahlreichen Studien belegt die konservativere Schätzung der prädiktiven Unsicherheit ausgeben, sondern auch im Umgang mit Daten besser geeignet sind, in denen Concept Drifts vorliegen.

Schlagworte— Concept Drift, Detektion, Handling, Model Uncertainty, Ensembles, Monte-Carlo Dropouts, Streaming

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

ABKÜRZUNGSVERZEICHNIS

ADWIN Adaptive Windowing

AUC Area Under Curve

BNN Bayessche Neuronale Netze

CUSUM Cumulative Sum

CDBD Confidence Distribution Batch Detection

DDM Drift Detection Method

EUDD Ensemble Uncertainty Drift Detection

KS-Test Kolmogorov-Smnirnov-Test

MAE Mean Average Error

MCC Matthew's Correlation Coefficient

NLL negative Log-Likelihood

OOD Out-of-Distribution

PCA Principal Component Analysis

PH Page-Hinkley

RMSE Root Mean Squared Error

ROC Receiver Operating Characteristic

SPRT Sequential Probability Ratio Test

UDD Uncertainty Drift Detection

Teil I

THESIS

1

EINLEITUNG

1.1 MOTIVATION

Machine Learning und Deep Learning sind längst keine reinen Forschungsthemen mehr. Während in 2019 zwar erst 6 Prozent aller befragten Unternehmen KI nutzten oder implementierten [65], waren es in 2021 bereits 13 Prozent, die KI nutzten [29]. Obwohl während dieses Zeitraums die Wirtschaft stark durch die Corona Pandemie beeinträchtigt worden ist, kam es trotzdem zu einem deutlichen prozentualen Anstieg. Es ist also zu erwarten, dass die Nutzung von Machine Learning und Deep Learning zukünftig noch deutlich weiter ansteigen wird.
Doch für den effektiven Einsatz von Künstlicher Intelligenz reicht es nicht aus, Modelle initial zu trainieren und anschließend die dort erzeugte Version produktiv zu nutzen.
Während der Verwendung solcher Modelle verändern sich deren Umweltbedingungen. Dies kann bspw. in Produktionen durch fehlerhafte Sensoren verursacht sein. Eine solche Veränderung wird verhältnismäßig einfach zu erkennen und zu beheben sein. Aber wie kann ein sich veränderndes Verhalten von Kunden erkannt werden, das bspw. aus einer wirtschaftlichen Rezession oder veränderten Kundenbedürfnissen entstanden ist? Wie kann ein Modell mit Situationen umgehen, die ihm gar nicht bekannt sind?
Nicht zuletzt ist die Corona Pandemie ein Beispiel dafür, wie schnell sich Umweltfaktoren verändern können, die nicht vorherzusehen waren und die demnach auch kein Modell in Form von Features (Einflussvariablen) enthalten konnte. Es ist also anzunehmen, dass kein Modell vollumfänglich für alle Veränderungen gewappnet sein kann. Dies ist allein durch begrenzte Speicherkapazitäten oder Anforderungen für Verarbeitung in Echtzeit schwer möglich.
Um Modelle immer wieder an sich verändernde Bedingungen anzupassen, werden diese üblicherweise in einem produktiven Umfeld re-

gelmäßig neu trainiert. Damit nimmt die Bedeutung an Technologien und Plattformen aus dem MLOps Bereich zu. Hier steht im Vordergrund, neben dem Deployment auch ein Monitoring der Modelle durchzuführen, um die Modellperformance zu überwachen und reproduzierbare Ergebnisse zu erhalten. Einer Befragung nach sehen 24 Prozent der AI Leader die Notwendigkeit, auch unabhängig der Modell-Performance alle $1 - 30$ Tage Modelle neu zu trainieren [47]. Doch das Modell nur in regelmäßigen Zeitabständen neu zu trainieren, birgt das Risiko, bei abnehmender Performance nicht rechtzeitig einzugreifen, sondern - hält man sich an die genannten Werte - im schlechtesten Falle erst 30 Tage später. Somit würde in diesen 30 Tagen die Vorhersagegüte des Modells verringert sein, was zu fehlerhaften Entscheidungen führt und im schlimmsten Fall hohe Kosten oder Sicherheitsrisiken verursacht. Erforderlich ist also eine rechtzeitige Erkennung und automatisierte Reaktion auf sich verändernde Bedingungen. Eine solche Veränderung der Umwelt wird auch als Concept Drift bezeichnet.

Da Concept Drifts nicht immer in den bereits im Modell befindlichen Features auftreten und selbst diese nicht immer auch bei Veränderungen einen direkten Einfluss auf die Zielgröße haben, wird häufig allein die Modellperformance basierend auf dem Vorhersagefehler überwacht und bei einer deutlichen Änderung eingegriffen. Wenn aber nicht sofort die Zielvariable (Label) für eine mögliche Kontrolle vorliegt, was in vielen Fällen zu erwarten ist, empfiehlt sich ein unüberwachter Ansatz. Zu diesem Zweck wird in der in dieser Arbeit vorgestellten Methode die prädiktive Modellunsicherheit (Model Uncertainty) gemessen. Basierend auf dem aktuellen Forschungsstand in der Quantifizierung von Uncertainty und dem Umgang mit Concept Drifts müssen theoriegeleitet Modelle und Algorithmen identifiziert werden, die beide Anforderungen erfüllen können. Es wurde sich an einem bereits existierenden Ansatz, der UDD [4] orientiert. Da in diesem Ansatz jedoch nicht der derzeitige State of the Art methodisch eingesetzt wurde, wird der Algorithmus angepasst.

1.2 ZIEL DER ARBEIT

Ziel dieser Arbeit ist die Entwicklung eines unüberwachten Ansatzes, der basierend auf der Uncertainty Concept Drifts erkennt und automatisiert auf solche reagiert. Auch hier wird, ähnlich zu den Grundsätzen von MLOps, das verwendete Modell neu angepasst. Unterschiedlich ist jedoch das reaktive Verhalten des Algorithmus im Gegensatz zu rein zeitabhängigen und regelmäßigen Anpassungen. Diese basieren nicht auf der Modell-Performance, da von dem Fall ausgegangen wird, dass die dafür notwendigen Zielgrößen gar nicht oder nicht vollständig vorliegen. Der Algorithmus soll zudem auf gestreamten Daten anwendbar sein.
Für die Umsetzung dessen wird ein Algorithmus entwickelt und in Python implementiert, der anhand von typischen Benchmark-Datensätzen in Bezug auf den Umgang mit Concept Drifts untersucht wird. Mit Hilfe von synthetischen und Real-World Datensätzen soll ein Stream simuliert werden. Innerhalb dieses Streams sollen Detektionen von Concept Drifts stattfinden und das Gesamtergebnis soll durch Anpassungen des Modells nach solchen Ereignissen verbessert werden.
Um eine Aussage über die Güte der Methode treffen zu können, sollen verschiedene Strategien als Vergleich dienen. Abschließend wird die weiterentwickelte Methode in Bezug auf die Ergebnisse der UDD beurteilt. Damit soll gezeigt werden, dass die Nutzung derzeitiger State of the Art Methoden aus der Quantifizierung von Uncertainty zuverlässigere Ergebnisse als die der UDD liefert.

1.3 AUFBAU DER ARBEIT

Zuerst werden in Kapitel 2 Grundlagen für den Umgang mit Concept Drifts in gestreamten Daten und den Methoden für die Quantifizierung von Uncertainty geliefert. In Kapitel 3 wird der zu entwickelnde Algorithmus konzeptuell beschrieben. Dessen Umsetzung sowie die Ergebnisse der Tests auf verschiedenen Datensätzen finden sich in Kapitel 4. Die Arbeit wird mit einer Diskussion der Ergebnisse und einem Ausblick für zukünftige Forschung in Kapitel 5 geschlossen.

2

GRUNDLAGEN UND VERWANDTE ARBEITEN

2.1 STREAMS VON DATEN

Algorithmen werden häufig auf statischen Daten entwickelt, d.h. einem Datensatz, der schon in seiner Gesamtheit vorliegt [33]. Neben der Tatsache, dass sich auch statische Datensätze weiterentwickeln, beruhen viele praktische Anwendungsfälle aber auf gestreamten Daten [58]. Dies verursacht Besonderheiten, aufgrund derer die klassischen Methoden aus dem Umgang mit statischen Daten nicht direkt übertragbar sind.

2.1.1 *Definition und Abgrenzung*

Ein Stream von Daten ist eine potenziell unbegrenzte und geordnete Sequenz von Daten, die im Laufe der Zeit eintrifft [33]. Über die Reihenfolge des Eintreffens besteht keine Kontrolle [19].
Charakteristisch für viele Streams ist ebenfalls, dass die Daten schnell (in Relation zu der Rechenkapazität des Systems) eintreffen, wobei die Zeit zwischen den eintreffenden Instanzen variabel ist [5]. Die Geschwindigkeit kann in manchen Anwendungsfällen sogar so hoch sein, dass jede Instanz aufgrund der enorm hohen Rate eintreffender Daten und begrenzter Speicher-Ressourcen nur einmal verarbeitet werden kann [45]. Verschärft werden kann diese Situation zusätzlich durch die Zeit, in der Ergebnisse vorliegen müssen [33]. Aufgrund begrenzter Ressourcen ist es üblich, Daten direkt nach der Verarbeitung (wie z.B. einer Aggregation) zu verwerfen. Nur wenn zwingend notwendig, werden die Daten gespeichert [56].
Das Vorliegen einer großen Menge einströmender Daten führt oftmals dazu, dass nicht allen Daten ihre wahren Labels zugewiesen werden können oder dies nur zeitverzögert geschehen kann [19]. Daraus ergeben sich unterschiedliche Ansätze für das maschinelle Lernen.

Auch der Datentyp dieser unbegrenzten Folge von Datenelementen kann variieren. So können eintreffende Daten von einfachem Typ (wie Tupel) sein oder auch komplexe Datenstrukturen (wie Graphen) aufweisen [33].
Sequenzen, folglich auch Streams von Daten, können eingeteilt werden in [73]:

MATERIALIZATION SEQUENCE: Die Daten sind höchst abhängig voneinander, d.h. der aktuell beobachtete Punkt hätte nicht ohne die vorigen existieren können (ist meistens in Zeitreihen-Daten der Fall). Damit ist eine einzige Beobachtung jedoch nicht wichtig.

OBSERVATIONAL SEQUENCE: Die Daten sind nicht voneinander abhängig, sondern werden von einem zugrunde liegenden Kontext bzw. Konzept generiert/beeinflusst. Damit ist deren Eintreffen (Reihenfolge) zufällig. Jede beobachtete Instanz ist eine Stichprobe aus einer (wahrscheinlich) noch größeren Menge an Instanzen. Um mit dieser Menge umzugehen ist es zielführend, das Label jeder einzelnen Instanz zu ermitteln. Damit ist jede Beobachtung wichtig.

TEMPORAL SEQUENCE: Die Reihenfolge und/oder Zeit, in der die Daten auftreten, ist zeitabhängig.

SPATIAL SEQUENCE: Das Eintreffen der Daten hängt nicht von der Zeit ab, aber folgt einer anderen logischen Komponente (z.B. die Analyse des Teer-Zustands einer großen Straße kann in mehrere Abschnitte aufgeteilt werden, deren zeitliches Eintreffen irrelevant ist).

LOGICAL SEQUENCE: Alles, was weder zeit- noch raumabhängig ist, aber einer bestimmten Logik folgt.

Derartige Muster zu erkennen bzw. zu verstehen kann entscheidend für den Erfolg entsprechender Modelle sein [73].
Zu erwarten ist gerade bei einer *Observational Sequence*, dass sich die Daten über die Zeit verändern, da diese durch zugrunde liegende Kontexte bestimmt werden. In der Regel treten diese Veränderungen in den Verteilungen der Daten auf. Es ist davon auszugehen, dass

Daten aus einer zugrunde liegenden stationären Verteilung entstanden sind. Diese Verteilung wird auch als Kontext bezeichnet und ist i. d. R. nicht beobachtbar. Sich verändernde Daten werden also durch verschiedene zugrunde liegende Kontexte verursacht. Daher kann ein Stream von Daten auch als eine Reihe aufeinander folgender Kontexte verstanden werden [21].

2.1.2 *Lernansätze in Streams*

Je nach Vorliegen des Labels unterscheidet sich die Vorgehensweise in dem maschinellen Lernen.
Angenommen, der Stream S bestände aus Instanzen z^t, denen ein Label zugewiesen ist $z^t = (X^t, y^t)$ für $t = 1, ...T$. Vorhersagen würden dementsprechend auf Basis von X^t getroffen werden, sobald die Input-Daten vorliegen. Treffen auch die Labels y^t der Features X^t nach gewisser Zeit durch den Stream ein, kann die Abweichung dieser Vorhersage $\hat{y}^t$ und dem tatsächlichen Wert ermittelt und genutzt werden, um das Modell anzupassen. Dieses Vorgehen wird auch als *vollständig überwachter Ansatz* bezeichnet [33].
Doch nicht immer können allen einströmenden Daten Labels zugewiesen werden. In einem *eingeschränkt überwachten Ansatz* treffen die Labels sehr spät oder nur teilweise ein. Ein eingeschränkt überwachter Ansatz kann also folgenden Szenarien entsprechen [33]:

DELAYED LABELING: Daten erhalten das Label erst sehr spät (z.B. bei Fraud Detection, bei denen das Label vielleicht nie bestätigt sein wird).

SEMI-ÜBERWACHTES LERNEN: Labels sind nur teilweise vorhanden oder werden durch Ansätze wie Active Learning explizit angefordert (z.B. bei Kreditbewilligungen, die sich erst nach mehreren Jahren als erfolgreich bzw. nicht erfolgreich erweisen).

Unüberwacht wäre ein Ansatz im Gegenteil dazu, wenn Modelle mit einer limitierten Anzahl an Instanzen, die Labels enthalten, gebildet werden und anschließend ohne Zugriff auf weitere Labels eingesetzt werden.

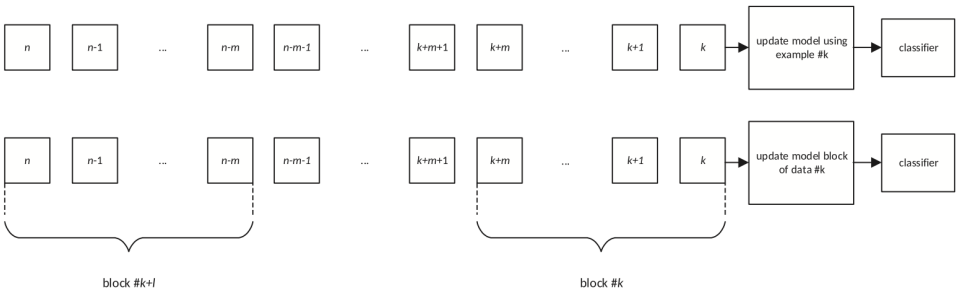

Abbildung 2.1: Varianten der Datenverarbeitung des Maschinellen Lernens in Streams [33]

Je nach Datenmenge und Geschwindigkeit der einströmenden Daten können die gesamten Daten oder nur eine Untermenge von Daten genutzt werden [3]. Aufgrund der üblicherweise hohen Datenmengen ist es jedoch i. d. R. nicht möglich, alle Daten zu speichern und für eine möglicherweise notwendige Anpassung des Modells zu verwenden.
Daher haben sich für das Lernen in gestreamten Daten zwei Vorgehensweisen entwickelt. Eine Möglichkeit wäre, wie in der obigen Variante in Abbildung 2.1, die Daten alle einzeln zu verarbeiten. Bei dieser Strategie wird von *Online Learning* (auch inkrementelles Lernen) gesprochen. In der anderen Herangehensweise, dem *Offline Learning*, wird der Algorithmus sequentiell auf Batches von Daten trainiert [34]. Wie in Abbildung 2.1 dargestellt, werden in der unteren Variante Daten zu Batches zusammengefasst. Ein Batch B ist eine bestimmte Menge an Instanzen. Anders ausgedrückt könnte damit der Stream S auch als Sequenz bestimmter Mengen von Daten verstanden werden $S = B_1 \cup B_2 \cup ... \cup B_n$ [33]. Damit würde eine Verarbeitung dieser Daten erst erfolgen, wenn die Gesamtmenge an Daten der jeweiligen Batch vollständig vorliegt [33].

2.1.3 *Anforderungen an Systeme für gestreamte Daten*

Systeme sollten unabhängig von der bisher gesehenen Datenmenge jede Instanz in der gleichen (kurzen) Zeit verarbeiten [34]. Zusätzlich ist mit einer unbegrenzten Länge des Streams zu rechnen. Damit können häufig keine klassischen batch-orientierten Verfahren angewendet

werden [20], da diese an mindestens einer der folgenden Einschränkungen scheitern [5]:

1. Aufgrund der unbegrenzten Datenmenge ist es unmöglich, alle Daten zu speichern. Nur ein kleiner Teil des Streams kann im Speicher gehalten werden.

2. Bei sehr schnell einströmenden Daten müssen diese in Echtzeit verarbeitet und anschließend wieder verworfen werden. Damit kann eine Instanz der Daten nur einmal verarbeitet werden.

3. Die zugrunde liegende Verteilung der Daten kann sich über Zeit ändern, weshalb ältere Daten zur Modellierung des neuen Kontextes irrelevant oder sogar schädlich sein könnten.

Während sich wegen der ersten beiden Anforderungen Methoden entwickelt haben, die den Speicherverbrauch und die Verarbeitungszeit optimieren, führt die dritte Größe dazu, dass Veränderungen in den Daten erkannt und Modelle angepasst werden müssen [54].
Eine weitere Herausforderung für das Maschinelle Lernen auf Streams ergibt sich dadurch, dass Algorithmen häufig eine unabhängige und identische Verteilung der Daten fordern. Neben den Veränderungen der Daten, die sich zwischen Zeitpunkt t und t' ergeben, konnte aber auch gezeigt werden, dass in einigen Real-World Datensätzen zwischen den Daten zeitliche Abhängigkeiten bestehen [8, 72]. So konnte bspw. in dem *Electricity* Datensatz, der häufig für die Untersuchung von Concept Drifts verwendet wird, eine Autokorrelation nachgewiesen werden [72]. Viele Algorithmen sollten demnach nicht ohne eine Berücksichtigung dieser Abhängigkeiten für den Einsatz in Streams verwendet werden.
Neben diesen technischen Anforderungen stehen Anwender in der Praxis weiteren Herausforderungen gegenüber. In sicherheitskritischen Anwendungsbereichen (z.B. Fraud Detection Systeme) kann es beispielsweise erforderlich sein, dass eine Entscheidung sofort vorliegen muss, während in anderen Bereichen wie bspw. der Kreditbewilligung die Entscheidung erst nach mehreren Tagen bis Wochen benötigt wird [73]. Des Weiteren ist es in manchen Anwendungsbereichen möglich, dass das sog. *Ground Truth Label*, also die tatsächliche Ausprägung der Zielvariablen, nicht durch klare Akzeptanzkriterien (wie

z.B. bankrott sein) definiert ist, sondern auf einer subjektiven Meinung (wie z.B. interessant/nicht interessant) basiert [73].
Auch die Kosten einer Fehlentscheidung oder eines Fehlalarms sind abzuwägen, weshalb es erforderlich ist, geeignete Kostenfunktionen und Metriken auszuwählen [73].

2.2 CONCEPT DRIFT

Wie bereits in Kapitel 2.1.3 deutlich geworden ist, wird in der Entwicklung von Algorithmen auf statischen Daten von einem zugrunde liegenden stationären Prozess ausgegangen.
In der Praxis werden Algorithmen jedoch häufig auf Streams angewendet, bei denen mit Nicht-Stationarität gerechnet werden muss. Folglich müssen klassische Algorithmen an das Problem der Nicht-Stationarität angepasst werden.

2.2.1 *Der Begriff der Stationarität*

In einem stationären Umfeld ist die (unbekannte) Verteilung der Daten stabil und verändert sich nicht [33].
Stammt eine Reihe von Daten aus einem Zustand, bei dem die Verteilung stationär war, werden diese auch als *Kontext* oder *Konzept* bezeichnet [21]. Die Daten eines nichtstationären Streams würden demnach aus mindestens zwei Kontexten entstanden sein.
Bei Veränderungen der Verteilung der Daten in nicht-stationären Streams spricht man häufig auch von einem Drift.

2.2.2 *Der Begriff des Concept Drift*

Das Phänomen, durch das sich die Verteilung der Daten des zuletzt aktuellen stationären Kontexts verändert, wird als Concept Drift bezeichnet [54]. Dieser kann durch eine Veränderung des zugrunde liegenden, nicht beobachtbaren (Hidden) Kontexts verursacht sein [64].
Nicht als Concept Drift bezeichnet werden einmalige und zufällige Abweichungen in den Daten, durch Störeinflüsse wie Rauschen (Noise) oder durch Ausreißer verursacht [54]. Die Unterscheidung zwischen einem Concept Drift und Noise zählt laut Tsymbal et al. [58] zu den größeren Schwierigkeiten in der Detektion. Um Concept Drifts also sukzessive voneinander unterscheiden zu können, wird nicht nur die zeitliche Entwicklung und deren Veränderungsgeschwindigkeit geprüft, sondern auch die sog. *Drift randomness*, in der zwischen Nicht-Stationarität und Rauschen unterschieden wird.

Concept Drifts können in vielen verschiedenen Varianten auftreten, nicht zuletzt abhängig von der zeitlichen Entwicklung oder der statistischen Problemstellung.

2.2.3 *Varianten anhand zeitlicher Entwicklungen*

Eine oft verwendete Einteilung der Varianten von Concept Drifts ist die in abrupte, graduelle oder inkrementelle Drifts [33, 54, 73]. Des Weiteren könnte es auch zu wiederkehrenden Kontexten kommen, zwischen denen Concept Drifts liegen.
Wie in Abbildung 2.2 dargestellt, führt ein abrupter Concept Drift zu einer sofortigen und plötzlichen Veränderung der Verteilung der Daten. Demnach würden nach dem Drift sofort Daten zu beobachten sein, die aus einem anderen Kontext als dem vorigen stammen [54].

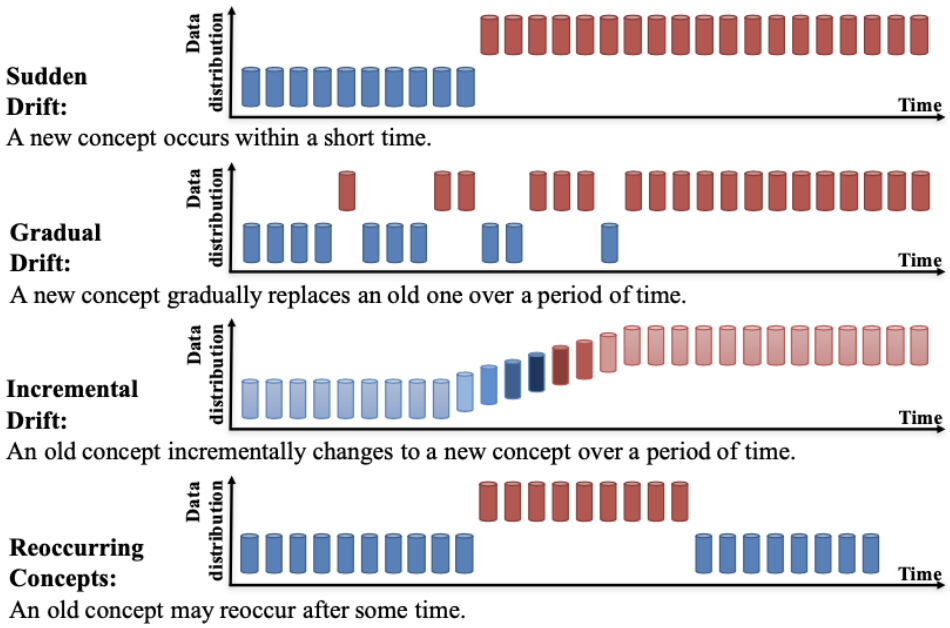

Abbildung 2.2: Varianten eines Concept Drifts [37]

Bei einem graduellen Concept Drift kommt es zu einem Wechsel zwischen dem initialen und dem finalen Kontext. Die Wahrscheinlichkeit steigt immer weiter an, Instanzen aus dem nachfolgenden Kontext zu beobachten, während die Wahrscheinlichkeit, Instanzen aus dem vorigen Kontext zu beobachten, immer weiter sinkt [54]. Beide Kontexte sind jedoch in sich stationär und zeigen auch in der Über-

gangsphase keine Veränderungen [54]. Weiter differenziert werden kann ein gradueller Concept Drift anhand seiner Geschwindigkeit. Dabei werden eine moderate oder langsame Veränderungsgeschwindigkeit unterschieden [55].
Anders verhält es sich bei einem inkrementellen Drift. Hier kommt es zu einer Folge unterschiedlicher Kontexte zwischen dem initialen und dem finalen Kontext [54].
In bestimmten Anwendungsszenarien tritt eine periodische bzw. zyklische Rückkehr zu einem zuvor aktuellen Kontext auf [13]. Dies könnte bei wiederkehrenden Phänomenen beeinflusst durch bspw. die Jahreszeiten oder auch den Stromverbrauch der Fall sein. Wiederkehrende Kontexte können auch irregulär sein (wie z.B. die Inflation) [58].
Ein Concept Drift kann zudem als unerwartet (z.B. Entwicklungen auf dem Finanzmarkt), teilweise erwartet (z.B. eine Finanzkrise kündigt sich durch externe Warnsysteme an) oder erwartet (z.B. durch Saisonalität) bezeichnet werden [73].

2.2.4 *Varianten anhand der statistischen Problemstellung*

Werden Concept Drifts aus statistischer Perspektive betrachtet, wird deutlich, an welcher Stelle der gemeinsamen Wahrscheinlichkeitsverteilung es zu einer Veränderung kam. Dabei kann sich an den zwei Arten von Problemen in der statistischen Modellierung [15] orientiert werden:

$X \rightarrow y$: Die Zielvariable bzw. das Label wird von den Input-Daten abgeleitet. Deren gemeinsame Wahrscheinlichkeitsverteilung wird geschrieben als $P(X, y) = P(y|X) * P(X)$. Dies wäre beispielhaft der Fall, wenn versucht wird, die Absatzmenge von Speiseeis anhand von Wochentag, Jahreszeit und/oder Wetterbedingungen vorherzusagen.

$y \rightarrow X$: Die Zielvariable bzw. das Label bestimmt die Ausprägungen der Features. Die gemeinsame Wahrscheinlichkeitsverteilung wird geschrieben als $P(X, y) = P(X|y) * P(y)$. Auch dieses Phänomen lässt sich in der Praxis wiederfinden, wie z.B. das

Auftreten charakteristischer Symptome bei einer Grippe wie z.B. Fieber, Gliederschmerzen und Anzeichen einer Erkältung.

In der Literatur wird häufig angenommen, dass ein Concept Drift durch eine Veränderung der Verteilung der Daten, also einem *Dataset Shift* verursacht wird. Dies greifen Moreno-Torres et al. [43] auf und liefern anhand der zwei Arten der statistischen Modellierung eine genauere Aufgliederung in folgende Phänomene: Covariate Shift, Prior Probability Shift oder Concept Shift.

Ein Covariate Shift betrifft nur die Verteilung der Features und kann damit nur in $X \rightarrow y$ Szenarien auftreten [43].

Definition 2.2.1 (Covariate Shift)**.** *Ein Covariate Shift tritt dann auf, wenn die Verteilung $P_t(y|X)$ der Verteilung zu dem nachfolgenden Zeitpunkt $P_{t+1}(y|X)$ entspricht, aber die Verteilung der Features sich zwischen den Zeitpunkten unterscheidet $P_t(X) \neq P_{t+1}(X)$.*

Nur in $y \rightarrow X$ Problemen kann es zu einem Einfluss durch den Prior Probability Shift kommen, da dieser ausschließlich zu einer Veränderung der Zielvariablen führt [43].

Definition 2.2.2 (Prior Probability Shift)**.** *Obwohl sich die Verteilung der Zielvariablen von Zeitpunkt t zu Zeitpunkt $t+1$ verändert $P_t(y) \neq P_{t+1}(y)$, kommt es nicht zu einer Veränderung der bedingten Wahrscheinlichkeitsverteilung der Features $P_t(X|y) = P_{t+1}(X|y)$.*

Trotzdessen muss nicht gelten, dass $P_t(y|X) = P_{t+1}(y|X)$ [54].

Nach Moreno-Torres et al. [43] ist ein Concept Shift als Folge einer Veränderung der Beziehung zwischen der Zielvariablen und den Features die am schwierigsten zu erkennende Situation. Hier muss es nicht zwingend zu einer Änderung der Verteilung der Input-Features kommen [33].

Definition 2.2.3 (Concept Shift)**.** *Von einem Concept Shift wird gesprochen, wenn eine der folgenden Szenarien zutrifft:*

1. *$P_t(y|X) \neq P_{t+1}(y|X)$ und $P_t(X) = P_{t+1}(X)$ in $X \rightarrow y$ Problemen*

2. $P_t(X|y) \neq P_{t+1}(X|y)$ *und* $P_t(y) = P_{t+1}(y)$ *in* $y \rightarrow X$ *Problemen*
3. $P_t(y|X) \neq P_{t+1}(y|X)$ *und* $P_t(X) \neq P_{t+1}(X)$ *in* $X \rightarrow y$ *Problemen*
4. $P_t(X|y) \neq P_{t+1}(X|y)$ *und* $P_t(y) \neq P_{t+1}(y)$ *in* $y \rightarrow X$ *Problemen*

In den Varianten 1 und 3 kommt es zu einer Veränderung in der bedingten Verteilung der Zielvariablen, wobei sich deren zugehörige Features in Variante 1 nicht verändern [54]. Der Unterschied zwischen den Szenarien liegt also darin, dass in Variante 1 die Ausprägungen der Features für eine Detektion eines solchen Concept Shifts nicht beachtet werden müssen.
Bei der zweiten Variante verändern sich die Ausprägungen der Features, wohingegen die Verteilung der Zielvariablen gleich bleibt. Dies gilt in der vierten Variante nicht.
Da alle Varianten des Concept Shift mit der bedingten Verteilung der Zielvariablen zusammenhängen, ist es mit unüberwachten Ansätzen, die nur auf den Verteilungen der Features beruhen, demnach nicht möglich, einen Concept Shift zu erkennen.

2.2.5 *Reale und virtuelle Concept Drifts*

Der Einfachheit halber wird ein Concept Drift in den meisten Fällen nur als Veränderung der gemeinsamen Wahrscheinlichkeitsverteilung $P^t(X, y) \neq P^{t+\Delta}(X, y)$ von den Features und der Zielvariablen verstanden [33], bei der sich die posteriore Verteilung durch eine geänderte a-priori Verteilung oder durch eine Änderung der bedingten Verteilung ergibt.
In allen Varianten, die eine Veränderung der Verteilung der Zielvariablen beinhalten, wird in diesem Zusammenhang auch von einem *realen* Drift gesprochen. Aus statistischer Perspektive entspricht ein realer Concept Drift dem Concept Shift und kann in jeder möglichen statistischen Problemstellung auftreten.
Um einen *virtuellen Drift* handelt es sich hingegen, wenn es nur in $P(X)$ zu einer Veränderung kommt (siehe Abbildung 2.3), aber die Verteilung von $P(y|X)$ nicht beeinträchtigt wird [54]. Damit liegt also ein Covariate Shift vor und kann sich nur in Szenarien äußern, wenn die Zielvariable von den Features abgeleitet wird ($X \rightarrow y$). Wie in der

Abbildung 2.3 zu sehen ist, verändert sich die Verteilung der Daten, aber nicht die Entscheidungsgrenze (*Decision Boundary*).

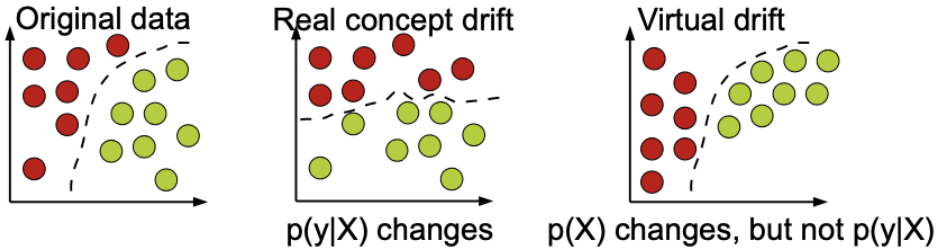

Abbildung 2.3: Unterscheidung von realem und virtuellem Drift [23]

Für den Umgang mit virtuellen Drifts wird zusätzliches Wissen bedeutsam, durch das bspw. ein Tuning der Parameter des Modells stattfinden könnte [13]. In dem Zusammenhang wird auch von *Supplement Learning* gesprochen, während bei einem realen Drift bisheriges Wissen in Form voriger Daten irrelevant wird (*Replacement Learning*) [13]. Beispielhaft kann hier der Umgang mit Spam-Mails angeführt werden: wie eine solche E-Mail aussieht, verändert sich mit der Zeit [58]. Es ist allerdings zweifelhaft, dass eine Änderung von $P(X)$ keinen Einfluss auf die Zielvariable bzw. Entscheidungsgrenze (*Decision Boundary*) hat [33]. Insofern zielen Studien selten auf eine Unterscheidung von realen und virtuellen Drifts ab.
Das zu einem virtuellen Drift gegenteilige Verhalten, wenn $P(y)$ sich verändert, aber nicht zu einer Veränderung von $P(y|X)$ führt, wird *Hidden Kontext* genannt. In einem Hidden Kontext sind also für die Vorhersage der Zielvariablen relevante Informationen nicht in den Features enthalten [27]. Würden diese nicht bekannten Größen vorliegen, könnte die Nicht-Stationarität natürlich aufgehoben werden [13]. Beispielhaft für einen Hidden Kontext werden häufig Kundenpräferenzen angeführt, da diese von einer Vielzahl von Möglichkeiten abhängen können, wie z.B. der Zeit, dem Tag der Woche, dem Vorhandensein von Alternativen, Inflationsraten uvm. [58].

2.2.6 *Algorithmen zur Detektion von Concept Drifts*

Um eine Abweichung detektieren zu können werden meist entweder die Input-Daten oder die Performance des Modells überwacht [33].

Ein Monitoring der Features entspricht einem unüberwachten Ansatz. Mit diesem werden nur Drifts detektiert, die sich in einer Veränderung der Verteilung der Features $P(X)$ ausdrücken [33].
Wenn Metriken zur Messung der Modell-Performance verwendet werden, führt dies häufig zu einem Trade-off zwischen wahren und falsch positiven (Fehlalarme) Ergebnissen [33]. Außerdem kann es zu einer Verzögerung der Detektion des Drifts kommen, wenn die wahren Labels verspätet eintreffen [33].
Methoden für die Detektion können kategorisiert werden als [33]:

1. Methoden basierend auf statistischer Prozesskontrolle
2. Methoden basierend auf sequentieller Analyse
3. Methoden, in denen Verteilungen der Daten anhand zweier disjunkter Fenster verglichen werden
4. Methoden, die auf Domänenwissen aufbauen

Nachfolgend wird auf die ersten drei Methoden eingegangen, da diese allgemeiner betrachtet werden können und nicht von dem Anwendungsfall abhängig sind.

2.2.6.1 *Drift Detektion durch Methoden statistischer Prozesskontrolle*

Eine der ersten und bekanntesten Methoden dieser Kategorie von Concept Drift Detektoren ist die Drift Detection Method (DDM) nach Gama et al. [21]. Dieser liegt eine Annahme aus der theoretischen Statistik zugrunde: bei einer stationären Verteilung der Daten muss die Fehlerrate des Modells bei zunehmender Datenmenge gleich bleiben oder sinken [38]. Ein Anstieg der Fehlerrate kann damit als Indiz für einen Concept Drift gewertet werden [3]. In der DDM werden die Wahrscheinlichkeit einer Fehlentscheidung p_i und die Standardabweichung $s_i = \sqrt{p_i(1-p_i)/i}$ ausgewertet [21]. Der Algorithmus hält p_{min} und s_{min} für die statistische Prozesskontrolle vor und aktualisiert die Werte. Über die doppelte bzw. dreifache Standardabweichung werden Schwellwerte definiert [21]. Bei Überschreitung des Warn-Levels ($p_i + s_i \geq p_{min} + 2 * s_{min}$) werden nachfolgend eintreffende Daten gesondert gespeichert und bei Überschreitung des Drift-Levels ($p_i + s_i \geq p_{min} + 3 * s_{min}$) für die Aktualisierung des Modells

verwendet [3]. Es wird angenommen, dass auf Basis dieses Vorgehens auch einige Stichproben aus dem nicht mehr aktuellen Kontext für die Aktualisierung verwendet werden. Folglich kann das Risiko einer Überanpassung an den neuen Kontext begrenzt werden. Wird das Drift-Level nicht überschritten, wird von einem Fehlalarm ausgegangen und die gespeicherten Daten werden nicht für eine Aktualisierung genutzt [21].

2.2.6.2 *Drift Detektion durch Methoden sequentieller Analyse*

Der Sequential Probability Ratio Test (SPRT) [62] berücksichtigt das Verständnis von Concept Drift als Veränderung der Wahrscheinlichkeiten des Auftretens der Daten aus verschiedenen Verteilungen. Sei X_1^w, $1 < w < n$ eine Untermenge an Daten aus einer unbekannten Verteilung P_0 und die folgende Untermenge X_w^n entstanden aus einer anderen unbekannten Verteilung P_1. Kommt es zu Zeitpunkt w zu einer deutlichen Änderung der Wahrscheinlichkeit, dass die gesehenen Daten aus der Verteilung P_1 stammen, sollte das Verhältnis der beiden Wahrscheinlichkeiten nicht kleiner als ein bestimmter Schwellwert sein. Berechnet wird dies wie folgt:

$$T_w^n = \log \frac{P\left(x_w ... x_n | P_1\right)}{P\left(x_w ... x_n | P_0\right)} = \sum_{i=w}^{n} \log \frac{P_1\left[x_i\right]}{P_0\left[x_i\right]} = T_w^{n-1} + \log \frac{P_1\left[x_n\right]}{P_0\left[x_n\right]} \tag{2.1}$$

Anhand eines durch den Nutzer definierten Schwellwertes wird entschieden, ob ein Concept Drift vorgelegen hat [23].
Auf Basis des SPRT hat Page [49] das Cumulative Sum (CUSUM) Verfahren entwickelt. Hier wird die kumulative Summe der Differenzen g_t eingehender Daten, wie bspw. der Erwartungswert des Vorhersagefehlers [33], mit einem Schwellwert λ verglichen. Wenn $g_t > \lambda$ wird ein Alarm ausgegeben [23]. Berechnet wird dies als $g_t = \max\left(0, g_{t-1} + (x_t - \delta)\right)$, wobei $g_0 = 0$ und x_t den aktuellen Daten entspricht [23]. Der Parameter δ gibt die tolerierte Höhe einer Veränderung an. Das Verfahren ist demnach abhängig von der Wahl der Parameter λ und δ. Der Test verbraucht keinen Speicherplatz.
Eine weitere Variante des CUSUM Verfahrens ist der Page-Hinkley (PH) Test [49]. Mit der Testvariable m_T wird das aktuelle Verhalten des Mo-

dells x_t zu Zeitpunkt t mit allen vorigen Parametern des gleichen Modells verglichen, wobei $\bar{x}_T = \frac{1}{T}\sum_{t=1}^{T} x_t$ [23]:

$$m_T = \sum_{t=1}^{T} (x_t - \bar{x}_T - \delta) \tag{2.2}$$

Der Parameter δ gibt auch hier die Höhe der tolerierten Veränderung an. Die Testvariable m_T wird mit dem Minimum bisheriger Ergebnisse $M_T = \min(m_t, t = 1, ...T)$ verglichen [23]. Wenn $PH_T = m_T - M_T$ größer als ein zuvor definierter Schwellwert λ ist, wird ein Alarm ausgegeben [23].
Die Methoden sequentieller Analyse eignen sich sowohl für Detektionen auf Basis der Zielvariablen sowie auch für unüberwachte Verfahren und weisen damit mehr Flexibilität als das zuvor vorgestellte Verfahren unter statistischer Prozesskontrolle auf.

2.2.6.3 *Drift Detektion durch Vergleich zweier Fenster*

In diesen Methoden werden Daten aus zwei (meist disjunkten) Fenstern anhand von statistischen Tests miteinander verglichen.
Diese beruhen zumeist auf einem fixierten Fenster, welches vergangene Informationen beinhaltet und einem gleitenden Fenster für neu eintreffende Informationen [23]. Zwischen zwei disjunkten Fenstern existiert keine Schnittmenge an Daten. Informationen über die Verteilung der Daten, Modellparameter oder Metriken beider Fenster werden durch statistische Tests verglichen [23].
Der bekannteste Algorithmus für den Vergleich der Daten mittels zweier disjunkter Fenster ist Adaptive Windowing (ADWIN) [6]. In anfänglichen Implementierungen gleitender Fenster musste die Fenstergröße zuvor festgelegt werden. Bei einer zu kleinen Größe (bzw. Datenmenge) wird somit zwar die aktuelle Verteilung der Daten besser repräsentiert, es kommt aber voraussichtlich ebenfalls zu einer Überanpassung an den dort vorliegenden Kontext. Bei einem größeren Fenster (mehr Daten) sorgt dies in einem stationären Umfeld für bessere Ergebnisse (wie z.B. eine höhere Accuracy (Korrektklassifizierungsrate)) [7], aber repräsentiert weniger gut den aktuellen Kontext. Dies wird auch als das *Stability-Plasticity Dilemma* [34] bezeichnet. Um diesem Trade-off zu entgehen, wird in ADWIN die Fenstergröße über

die zuletzt eingetroffenen Daten flexibel gehandhabt. Anhand der bestimmenden Metrik kann das Fenster bei einem stationären Zustand bis zu einer Maximalgröße anwachsen und wird bei der Detektion im Sinne einer signifikanten Änderung der Metrik verkleinert [7]. Dazu wurde in dem ursprünglichen Algorithmus ein großes Fenster W wiederholt mit verschiedenen Schnittpunkten in zwei disjunkte Fenster (W_1, W_2) geteilt, deren Mittelwerte μ_{w_1} und μ_{w_2} durch eine Statistik verglichen werden [7]. Bei einem signifikanten Unterschied, der durch den Parameter δ kontrolliert wird, war von einem Concept Drift auszugehen. Da dieses Vorgehen allerdings viel Speicherplatz verbrauchte und zeitintensiv war, weil alle Fenster ausreichender Größe miteinander verglichen werden mussten, wurde der Algorithmus als eine Variante des exponentiellen Histogramms weiterentwickelt [7]. Dabei werden, wie in Abbildung 2.4 dargestellt, bereits eingetroffene Daten aus dem Stream in Buckets organisiert, in denen die Daten komprimiert werden. In einem Bucket wird die Summe der eingetroffenen Daten und deren Varianz gehalten.

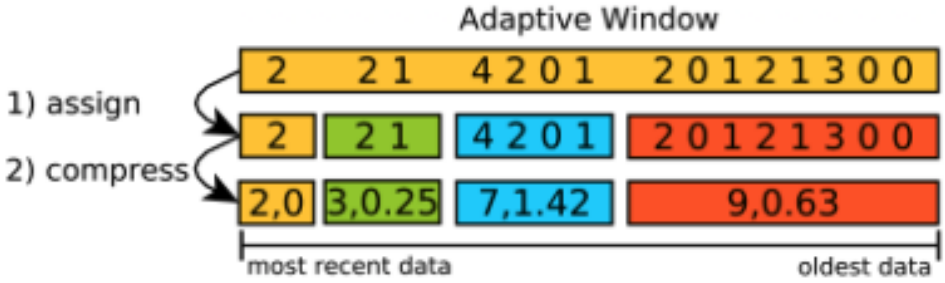

Abbildung 2.4: ADWIN als Variante des exponentiellen Histogramms [57].

Buckets umfassen eine exponentiell ansteigende Anzahl von Daten, wobei ältere Buckets eine deutlich größere Anzahl an Daten enthalten, während neu eingetroffene Daten in kleineren Buckets komprimiert werden. Wie schon in der ersten Variante des Algorithmus wird auch hier der Unterschied durch den Parameter δ kontrolliert.
Der Vergleich von zwei disjunkten Fenstern eignet sich ebenfalls für unüberwachte Verfahren, wenn Labels verspätet oder gar nicht eintreffen [70]. Diese Methoden überwachen i. d. R. die Verteilung der Input-Features [39], die bspw. durch einen Kolmogorov-Smnirnov-Test (KS-Test) verglichen werden könnten. Wenn es in $P(X)$ keine Veränderung gab, kann ein Concept Drift mit dieser Methode jedoch

nicht detektiert werden.
Vorteilhaft bei dem Einsatz von Methoden basierend auf zwei Fenstern ist, dass in diesen alle vorstellbaren Ausgaben des Modells aufgenommen werden können. Anstatt nur die Verteilung der Input-Features zu überwachen, kann auch die prädiktive Modell-Unsicherheit (Model Uncertainty, siehe Kapitel 2.3) als Vergleichsgröße gewählt werden. Indem die Uncertainty quantifiziert wird, wird die Entscheidung nicht mehr alleinig auf der Änderung von Verteilungen getroffen. Bereits die Confidence Distribution Batch Detection (CDBD) [36] lieferte eine Möglichkeit für den Vergleich der Sicherheit über die Modellausgaben in zwei disjunkten Fenstern. Auch die Uncertainty Drift Detection (UDD) [4] zeigte sich als vielversprechend. Dafür wurde in dem Algorithmus der UDD nach entsprechender Parametrisierung ein ADWIN Detektor (siehe Abbildung 2.5) genutzt, der als Eingabe die Model Uncertainty aus einem Neuronalen Netz unter Monte-Carlo Dropouts erhält.

Algorithm 1 Uncertainty Drift Detection

Input: Trained model M; Data stream $\mathcal{D}$; Training data $\mathcal{D}_{tr}$
Output: Prediction $\widehat{y}_t$ at time t
repeat
  Receive incoming instance x_t
  $\widehat{y}_t, U_t \leftarrow M.\text{predict}(x_t)$
  Add U_t to $ADWIN$
  if $ADWIN$ detects change **then**
    Acquire most recent labels y_{recent}
    $M.\text{train}(\mathcal{D}_{tr} \cup \mathcal{D}_{recent})$
  end if
until $\mathcal{D}$ ends

Abbildung 2.5: Algorithmus der UDD [4].

Im Vergleich zu einem KS-Test konnte die Anzahl von erneuten Trainings in der UDD bei gleich bleibender Modellgüte deutlich reduziert werden [4].
Die Methoden des Vergleichs zweier Fenster führen trotz der Einfachheit zu einem entscheidenden Nachteil: das eigentlich dynamische Problem wird als ein statisches betrachtet, mit dem nur lokale Charakteristika überwacht werden können [33].

2.2.7 *Concept Drift Handling*

Nach der Detektion wird in der Regel ein erneutes Training (Re-Training) oder ein Update des Modells (Refit) getriggert [33]. Bei diesen wird unterschieden, welche Strategie bezüglich des Zeitpunkts in der Modellanpassung verfolgt wird und wie diese in dem Lernansatz angewendet wird [23].

2.2.7.1 *Zeitpunkt der Modellanpassung*

Ein optimales Modell sollte jederzeit an die aktuellen Daten angepasst sein. In der passiven Adaption (auch: *Continuous Rebuild* oder *blinde Adaption* [36]) wird dies erreicht, indem zuvor ein Fenster fixierter Größe oder feste Zeitpunkte definiert werden, zu denen das Modell aktualisiert wird [23]. In der aktiven Adaption (auch: *Triggered Rebuild* oder *informierte Adaption* [36]) wird ein reaktives Verhalten des Modells gewünscht [23], sodass bestimmte Trigger (wie bpsw. ein Alarm aus dem Vorliegen eines Concept Drifts) eine Modellanpassung auslösen [7] (siehe Abbildung 2.6).

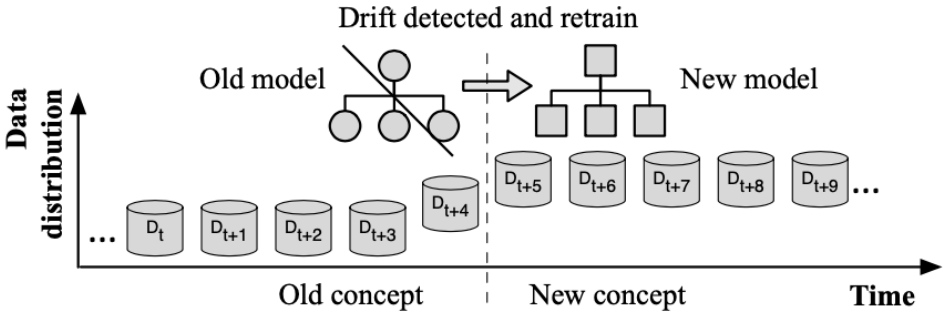

Abbildung 2.6: Schematische Darstellung eines Re-Training nach einem Trigger [37].

2.2.7.2 *Lernansatz*

Der Lernansatz bestimmt unter anderem, ob das Modell wie in der vorliegenden Abbildung 2.6 vollständig ersetzt wird und damit die vorige Version verworfen wird oder ob nur Teile des Modells angepasst werden [23]. Außerdem muss festgelegt werden, mit welchen

Daten Modelle aktualisiert bzw. angepasst werden.
Grundsätzlich ist immer mit einem Einfluss aus dem *Stability-Plasticity Dilemma* [34] zu rechnen. Es muss also entschieden werden, wie stark gewichtet der neue und der vergangene Kontext berücksichtigt werden. Daher sollte das Modell für das Lernen auch einen Mechanismus haben, der ihm erlaubt, bisheriges und neues Wissen zu vereinen, auch wenn dieses in einem Konflikt steht [13]. Das beinhaltet ebenfalls, irrelevant gewordenes Wissen verwerfen oder löschen zu können [13]. Zu diesem Zweck wurden Methoden entwickelt, in denen die Daten nicht nur durch Fenster fixierter oder flexibler Größe vorgehalten werden, sondern auch eine individuelle Gewichtung der Daten oder der Einsatz von Heuristiken möglich sind [7]. Diese erfordern jedoch die Vorgabe bestimmter Größen, über die der Anwender entscheiden muss und sind daher weniger in der Anwendung anzutreffen.
Bestenfalls sollte es zudem möglich sein, auf vergangenes Wissen erneut zurückzugreifen, wenn ein zyklischer Prozess vorliegt. Wissen sollte daher regelmäßig und inkrementell gespeichert werden, sodass das Modell zu jeder Zeit das beste Ergebnis ausgeben kann [13]. Um vorige Informationen nicht völlig zu verwerfen, kann dazu auch nur ein Refit des Modells durchgeführt werden. Dabei wird eine Anpassung bestehender Modellparameter durch Trainieren des Modells mit den zuvor angepassten Parametern vorgenommen. Rein methodisch könnten dafür auch einzelne Modelle verwendet werden. Diese können je nach Modelltyp (bspw. Neuronale Netze) gut inkrementell angepasst werden [73], wobei deren ältere Version verworfen werden muss. Ensembles wiederum behalten je nach Anwendung noch über eine gewisse Zeit das zurückliegende Wissen über andere Kontexte [73]. Des Weiteren empfiehlt sich der Einsatz von Ensembles, da diese dynamisch erweitert oder einzelne Bestandteile ersetzt werden können.
Natürlich sollte eine Erkennung und Anpassung an einen Concept Drift möglichst schnell erfolgen, wobei die Anpassung nicht durch Rauschen ausgelöst werden darf [58]. I. d. R. muss dafür nicht unterschieden werden, ob ein realer oder virtueller Drift vorliegt, da beide zu der Notwendigkeit eines erneuten Trainings führen [58].
Das Lernen auf neuen Daten erfolgt schließlich über Re-Training mit-

tels Batches von Daten (auch Offline-Learning) oder über eine inkrementelle Vorgehensweise (Online-Learning) [23] (siehe Kapitel 2.1.2). Re-Training erfordert immer einen gewissen Speicherverbrauch, insbesondere aber bei dem Offline-Learning. Stehen Schnelligkeit der Anpassung und die Nutzung von Speicher-Ressourcen im Vordergrund, kann stattdessen ein Online-Learning Ansatz gewählt werden. Damit kann ein Modell direkt nach einer fehlerhaften Entscheidung angepasst werden, wobei nur die aktuellen Daten für ein Update des Modells verwendet werden [23]. Dies führt zu dem Nachteil, dass ein verstärkter Einfluss des *Stability-Plasticity Dilemma* zu erwarten ist. Letztendlich ist eine Entscheidung über den Lernansatz also stark von dem Anwendungsfall, den verfügbaren Ressourcen sowie den Präferenzen der Anwender abhängig.

2.2.8 *Ensembles*

2.2.8.1 *Grundlagen*

Wie in Abbildung 2.7 anhand eines Ensembles für die Klassifikation dargestellt, setzen sich Ensembles aus einer Menge an Modellen zusammen. Deren Vorhersagen werden durch eine bestimmte Regel kombiniert, um eine Entscheidung über eine neu eintreffende Instanz zu erzielen [33]. Diese Kombination der Entscheidungen einzelner Modelle wird auch als *Majority Voting* bezeichnet, da eine (möglicherweise individuell gewichtete) Mehrheitsentscheidung getroffen wird.

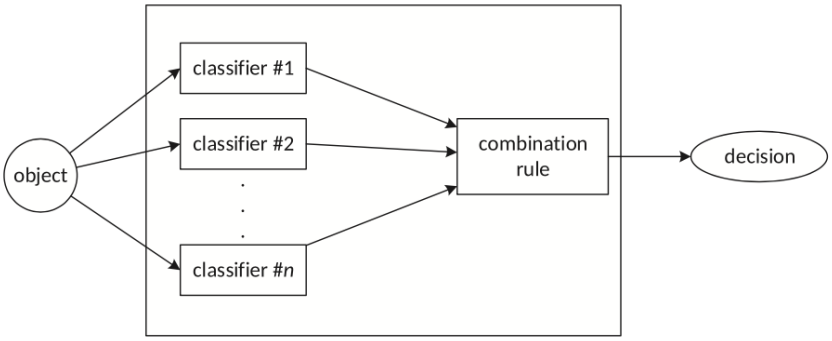

Abbildung 2.7: Schematische Darstellung eines Ensembles zur Klassifikation [33]

Wie auch das „No Free Lunch Theorem“ [66] besagt, sind die einzelnen Modelle des Ensembles allein für die Aufgabe des gesamten Modells alle gleich gut bzw. schlecht. In einem Ensemble hat aber jedes Modell seine eigene spezifische Domäne bzw. seinen eigenen Kompetenzbereich [33], da alle Modelle in einem anderen lokalen Minimum konvergiert sind. Oft wird daher im Zusammenhang mit einzelnen Modellen aus dem Ensemble auch von sog. *Weak Learnern* gesprochen. Der Mittelwert einer unbegrenzten Anzahl solcher Modelle in einem Ensemble führt nachweislich aber zu derselben Entscheidung wie der eines optimalen Bayes Klassifizierers [59]. Nur die Gesamtmenge aller Modelle des Ensembles kann die beste Entscheidung treffen.

Ensembles können nach zwei Strategien aufgebaut werden [30]:

- *Coverage optimization*: Bei einer festgelegten Entscheidungsfunktion werden komplementäre Modelle erzeugt, die die optimale Modellgüte erzielen können. Möglich wäre bspw., verschiedene Modelltypen zu verwenden oder die einzelnen Modelle mit unterschiedlichen Daten zu trainieren, um möglichst heterogene Modelle zu erhalten.

- *Decision optimization*: Für eine Menge vorgegebener Modelle wird versucht, eine optimale Entscheidungsfunktion zu finden. Ein möglicher Implementierungsansatz wäre hier eine dynamisch gehaltene und individuelle Gewichtung der einzelnen Modelle des Ensembles.

Ensembles sind in der Regel das Mittel der Wahl, wenn die Genauigkeit und nicht die Zeit im Vergleich zu anderen Modellen herangezogen wird [34]. Ein ideales Ensemble (in Bezug auf die Genauigkeit) besteht aus komplementären, möglichst heterogenen Komponenten. Deren Diversität wird als wichtige Einflussgröße verstanden [68].
Grundlegende Algorithmen sind das Bagging, Boosting und die Random Forests, die alle auf die Verarbeitung von gestreamten Daten unter Concept Drifts erweitert worden sind [37].

2.2.8.2 *Lernmethoden für Ensembles*

Ensembles müssen wie alle anderen Algorithmen in einem Stream auch nach dem Eintreffen neuer Instanzen angepasst werden, um langfristig ihre Vorhersagegüte zu erhalten bzw. zu optimieren. Erreicht wird dies bspw. über die Ergänzung des Ensembles (siehe Abbildung 2.8) um neu trainierte Modelle, was ggf. das Löschen veralteter Modelle erfordert. Alternativ kann ein Re-Training von allen oder nur Teilen der in dem Ensemble enthaltenen Modellen durchgeführt werden [33].

Dadurch, dass nicht das gesamte Modell, sondern nur Teile aktualisiert oder ersetzt werden, ist es Ensembles möglich, altes Wissen über gewisse Zeit beizubehalten und mit wiederkehrenden Kontexten umzugehen [37].
Die Möglichkeiten der Anpassung eines Ensembles orientieren sich

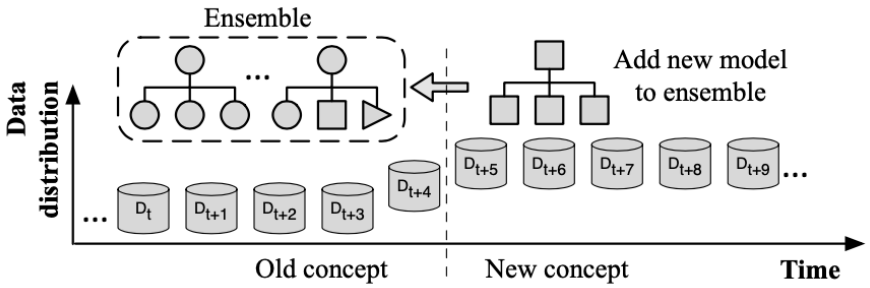

Abbildung 2.8: Aktualisierung eines Ensembles nach einem Concept Drift durch Hinzufügen eines neuen Modells [37].

an den Strategien des Aufbaus eines Modells im Sinne einer Coverage oder Decision Optimization. Diese können getrennt voneinander umgesetzt oder kombiniert werden [34]:

DYNAMISCHE ENTSCHEIDUNGSREGELN: Modelle werden im Voraus trainiert und die Regel für die Kombination der Entscheidungen wird dynamisch angepasst.

AKTUALISIERTE TRAININGSDATEN: Es werden Online-Updates der Modelle durch Re-Training auf Basis der aktuellsten Daten vorgenommen, auch eine Anpassung der Kombinationsregel ist möglich.

AKTUALISIERTE KOMPONENTEN DES ENSEMBLES: Modelle des Ensembles werden online aktualisiert oder durch eine Batch neu trainiert.

STRUKTURELLE VERÄNDERUNGEN DES ENSEMBLES: Modelle werden ersetzt (bspw. kann das schlechteste, auch: *Replace the Loser*, oder das älteste, auch: *Replace the Oldest*, Modell ersetzt werden).

HINZUFÜGEN NEUER FEATURES: Um das Ensemble nicht komplett neu trainieren zu müssen, werden weitere Modelle aufgenommen, die mit den neuen Features trainiert worden sind.

2.2.9 *Evaluation auf Streams von Daten*

Algorithmen werden anhand von zwei Kriterien evaluiert [67]: 1) Validierungsmethodik und 2) Metriken zur Evaluation.

2.2.9.1 *Validierungsmethodik*

Die Validierungsmethodik beschreibt, welche Daten für das Training und welche Daten für das Testen des Modells verwendet werden [67]. Hier wird zwischen Holdout Evaluation, Test-then-Train (auch genannt Prequential) und kontrollierter Permutation unterschieden [67].
In der Holdout Methode werden periodisch im Wechsel Daten aus einem prädefinierten Fenster für das Training und aus dem nachfolgenden Fenster zum Testen verwendet [25]. Damit muss ein Holdout-Set zu Zeitpunkt t auch den Kontext enthalten, der zu dem Zeitpunkt aktuell war [37]. Für die Anwendung dieser Methode muss der Zeitpunkt des Drifts bekannt sein, demnach ist sie am besten für synthetische Datensätze geeignet [37] und kann für einen Vergleich mit anderen Algorithmen verwendet werden.
Wie der Name der Test-then-Train Methode bereits vermuten lässt, wird in dieser jede Instanz der Daten zunächst für das Testen verwendet und anschließend für das Training [52]. Der Zeitpunkt des Drifts muss nicht bekannt sein. Umgesetzt werden kann die Test-then-Train Methode mit den gesamten Daten (auch genannt *Landmark Window*), einem gleitenden Fenster [37] oder durch einen Mechanismus, einem sog. Fading Factor, der Vergessen ermöglicht [25]. Durch einen Fading Factor können Daten basierend auf der Höhe des Vorhersagefehlers (Loss) ein individuelles Gewicht zugewiesen bekommen [22].
Durch Permutationen [71] oder Cross-Validation [25] würden parallelisiert mehrere Tests für die Evaluation durchgeführt werden.
In den meisten Systemen werden Test-then-Train Ansätze gewählt [22]. Da jedoch nicht immer sofort die wahren Labels vorhanden sind, kann eine Evaluation oft erst erfolgen, wenn das Modell bereits neu trainiert worden ist. Daher werden in einer Continuous Re-Evaluation [25] mehrere Vorhersagen des Modells generiert, bis das wahre Label eingetroffen ist. Nur die erste Schätzung des Modells für eine Eva-

luation zu verwenden wäre oft pessimistisch [25]. In der Continuous Re-Evaluation wird empfohlen, im Laufe des Streams erneute Vorhersage zu berechnen, da das Modell in der Zwischenzeit wahrscheinlich schon angepasst worden ist. Diese Vorhersagen werden ermittelt, wenn eine lange Zeit zwischen der ersten Vorhersage und dem Eintreffen des wahren Labels liegt oder wenn in der Zwischenzeit viele Labels anderer Instanzen der Daten eingetroffen sind.

2.2.9.2 *Metriken zur Evaluation*

Im Gegensatz zu einem statischen Umfeld darf in Streams nicht die punktweise oder gemittelte Vorhersagegüte eines Modells betrachtet werden. Gerade im Zusammenhang mit Concept Drifts ist das Verhalten in dem Verlauf des Streams entscheidend [33]. Zur Analyse dessen werden häufig Plots verwendet.
Eine Reihe von gängigen Metriken für die Evaluation eines Streams sind [33, 37]:

- Traditionelle Metriken: Accuracy, Mean Squared Error (MSE) bzw. Root Mean Squared Error (RMSE) und Recall
- G-Mean (geometrischer Mittelwert von Recall und Precision) für unbalancierte Daten
- Kappa-Statistik: $\kappa = \frac{p-p_{ran}}{1-p_{ran}}$ mit p als Accuracy und p_{ran} als Wahrscheinlichkeit, mit der ein zufälliges Modell eine richtige Entscheidung treffen würde (diese wurde für die Anwendung auf zeitabhängigen oder unbalancierten Daten erweitert)
- Area Under Curve (AUC): aus der Receiver Operating Characteristic (ROC) Kurve für ein Holdout-Set oder durch ein gleitendes Fenster

In gestreamten Daten werden weitere Größen für die Evaluation und den Vergleich von Algorithmen benötigt. Der Speicherverbrauch ist gerade in Bezug auf einzelne Arbeitsschritte besonders interessant zu untersuchen [37]. Im Zusammenhang damit kann auch die Zeit gemessen werden, die ein Algorithmus für die Anpassung an neue Daten braucht. Dies wird auch als *Recovery Analysis* [33] bezeichnet. Diese Zeit sollte idealerweise kleiner sein als die Zeit, in der eine

neue Batch an Daten eingetroffen ist [37], wenn Modelle offline angepasst werden. Auch die Zeit für eine Vorhersage bzw. Entscheidung aus dem Modell kann in bestimmten Fällen, insbesondere bei Anforderungen an die Verarbeitung in Echtzeit, entscheidend sein und sollte dann überprüft werden. Kommt es durch die Verarbeitung einer Instanz zu einer Verzögerung, entsteht gerade wenn das Modell anschließend aktualisiert werden muss eine Art Bottleneck-Effekt [33]. Dies gilt insbesondere für Algorithmen, die nicht gleichzeitig Vorhersagen treffen und aktualisiert werden können.

2.3 PRÄDIKTIVE UNCERTAINTY

Wie Baier et al. [4] bereits zeigen konnten, eignet sich die prädiktive Uncertainty unter anderem auch für eine Detektion von Concept Drifts. Die Uncertainty lässt sich aber nicht direkt an einer einfachen Schätzung des wahren Wertes durch ein Modell ableiten. Beispielhaft deutlich wird dies anhand eines Klassifikations-Problems. Dabei würden für k Klassen Schätzungen auf Basis des Outputs eines k-dimensionalen Vektors getroffen werden. Dieser wird auf den Wertebereich von $(0, 1]$ durch die Anwendung der Softmax-Funktion transformiert, wobei sich die Komponenten des Vektors zu 1 aufsummieren. Hohe Werte für eine Klasse, d. h. Stelle des Vektors, werden dabei häufig fälschlicherweise als hohe Sicherheit seitens des Modells interpretiert [18]. Da diese Werte gerade bei Neuronalen Netzen aber nur durch Linearkombinationen entstanden sind, kann das Modell trotz eines hohen Output-Wertes einer Klasse unsicher sein. Zudem kann allein basierend auf dem Softmax-Output nicht ermittelt werden, wie diese Unsicherheit zustande gekommen ist. Somit braucht es andere Methoden, um die Uncertainty eines Modells zu erklären und zu messen.
Nachfolgend werden daher eine Definition der Uncertainty geliefert, Methoden für die Quantifizierung eben dieser vorgestellt und auf Basis empirischer Vergleiche ermittelt, welches Verfahren sich am besten eignet.

2.3.1 *Definition und Arten von Uncertainty*

Unterschieden werden *aleatoric* und *epistemic* Uncertainty, auch häufig wie hier im folgenden als Data und Model Uncertainty bezeichnet [53]. In Abbildung 2.9 werden diese schematisch dargestellt.
Data Uncertainty beschreibt unkontrollierbare Störungen in den Daten, die bspw. durch nicht gut kalibrierte Sensoren entstehen können. Bleibt dieses Rauschen konstant bei verschiedenen Eingaben kann dies als *homoscedastic* Uncertainty bezeichnet werden, im Fall unterschiedlicher Intensität des Rauschens als *heteroscedastic* Uncertainty

[32]. Die homoscedastic Uncertainty ist unabhängig von den vorliegenden Datenmengen immer vorhanden.

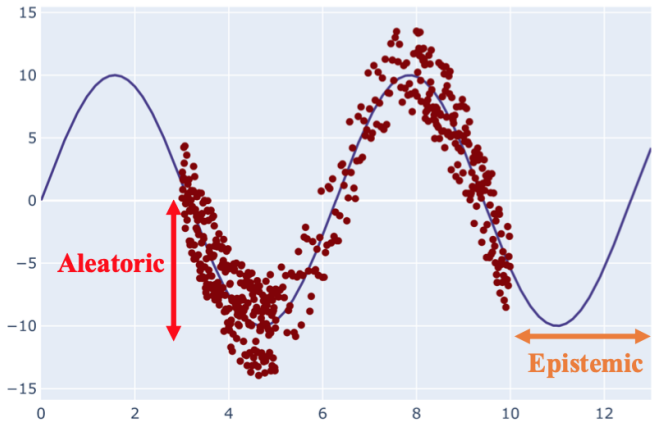

Abbildung 2.9: Schematische Darstellung der Uncertainty [1].

Die Model Uncertainty wird jedoch von den vorliegenden Datenmengen beeinflusst. Mit diesem Phänomen wird die Uncertainty bezogen auf das datengenerierende Modell beschrieben, die durch die Modell-Parameter wiedergegeben wird.
Aus dem Vorliegen von mehr Daten können damit auch durch das Modell bisher durch unzureichende Datenmengen schlecht angenäherte Repräsentationen besser gelernt werden. Die Model Uncertainty quantifiziert also auch Repräsentationen, die in den Trainingsdaten nicht enthalten sind (sog. Out-of-Distribution (OOD) Samples) [28].
Model Uncertainty kann zunächst als ein statistischer Wert verstanden werden [44]. Ausgehend davon kann die Model Uncertainty auch als Abweichung betrachtet werden. Für die Ermittlung von solchen Abweichungen braucht es also mehrere Vorhersagen. Angenommen, es lägen mehrere Vorhersagen für einen Datensatz vor, würde der Mittelwert die Schätzung des wahren Werts ergeben und die Höhe der Standardabweichung bzw. Varianz wäre hinweisgebend für die Model Uncertainty eben jener Vorhersagen. Mehrere Schätzungen gleicher bzw. ähnlicher Höhe führen zu einer geringen Varianz und können mit höherer Sicherheit als richtig angenommen werden, et vice

versa. Dies gilt insbesondere für Regressions-Probleme, bei denen die Varianz berechnet wird als [4]:

$$\hat{\sigma}^2 = \frac{1}{T} \sum_{i=1}^{T} \left(p_i \left(y | w_i, x \right) - \hat{p}(y|x) \right)^2 \tag{2.3}$$

Für Klassifikations-Probleme werden häufig entropie-basierte Maße verwendet, wie hier am Beispiel der Shannon-Entropie [4]:

$$H \left[\hat{p}(y|x) \right] = - \sum_{k=1}^{K} \hat{p}(y = k|x) - \log_2 \hat{p}(y = k|x) \tag{2.4}$$

2.3.2 *Relevanz der erfassten Art von Uncertainty*

Die meisten Ansätze, insbesondere solche die auf Sampling-Methoden beruhen, erfassen die Model Uncertainty. Von den insgesamt 15 betrachteten Studien (für eine Übersicht siehe Anhang A.1) wurde in nur drei die Data Uncertainty gemessen, während in 14 Studien die Model Uncertainty erfasst worden ist.
Durch die Vorhersage der Varianz der Eingabedaten kann ebenfalls die Data Uncertainty gemessen werden [32]. In der empirischen Studie von Kendall et al. [32] zeigte die Erfassung der Data Uncertainty eine stärkere Verbesserung als die der Model Uncertainty. Aus ihren Untersuchungen schlussfolgerten Kendall et al. [32]:

1. Die Data Uncertainty kann nicht durch Hinzunahme weiterer Daten besser erklärt werden
2. Die Data Uncertainty nimmt bei OOD-Samples nicht zu, wohingegen die Model Uncertainty ansteigt

Somit sei die Modellierung von Data Uncertainty insbesondere für große Datenmengen sinnvoll, bei der keine nennenswerte Model Uncertainty zu erwarten sei. Da die Erfassung von Model Uncertainty zumeist durch Sampling-Methoden realisiert wird, kann auf großen Datenmengen also eine erhebliche Steigerung der Performance erreicht werden, indem nur die Data Uncertainty gemessen wird. Dies macht solche Methoden auch für Echtzeit-Anwendungen praktikabel [32]. Zu beachten ist jedoch, dass Model und Data Uncertainty in vielen

Methoden separat betrachtet werden. Dies sollte nicht derart trennscharf gesehen werden, da sich beide Arten der Uncertainty gegenseitig bedingen oder verstärken können [53].
Die Messung von Model Uncertainty ist dahingegen besonders in sicherheitskritischen Umgebungen und kleineren Datenmengen wichtig.
In Bezug auf Concept Drifts können beide Arten der Uncertainty einen Einfluss zeigen. Eine erhöhte Data Uncertainty durch veränderte Verteilungen der Input-Daten muss jedoch nicht direkt mit einer Beeinflussung der Zielgrößen (virtueller Concept Drift) einhergehen. Die Model Uncertainty wiederum quantifiziert alle Unsicherheiten in der Vorhersage eben jener Zielgröße und ist daher für die Detektion von Concept Drifts entscheidender. Aufgrund dessen wird das Hauptaugenmerk im Folgenden auf Methoden für die Quantifizierung der Model Uncertainty gelegt.

2.3.3 *Methoden für die Quantifizierung von Uncertainty*

In wenig komplexen Situationen, in denen die Wahrscheinlichkeitsverteilung bekannt ist oder die Schätzung der a-priori-Verteilung sehr nahe der realen Verteilung ist, liefern Uncertainty- und Sensitivitätsanalysen ein zuverlässiges Maß für die prädiktive Uncertainty. Oft werden Sampling-Methoden angewendet, im speziellen Monte-Carlo Sampling für die zufällige Auswahl von Input-Parametern [44]. In der Sensitivitätsanalyse wiederum werden sukzessive Input-Parameter variiert und die Reaktion des Modells analysiert, um Rückschlüsse auf besonders sensitive Parameter für das Outcome zu ziehen [44].
Der Einsatz dieser Methoden wird jedoch häufig von der benötigten Anzahl an Iterationen im Training oder der Notwendigkeit mehrerer Modelldurchläufe während der Testzeit beeinträchtigt. Des Weiteren nimmt die Schwierigkeit, die tatsächliche Verteilung zu finden, durch komplexe Verteilungen der Daten mit bspw. nicht-linearen Zusammenhängen, zu. Mit immer komplexeren Bedingungen sind Modelle basierend auf klassischen statistischen Methoden also nicht mehr praktikabel.
Im Folgenden findet sich in Abbildung 2.10 eine Auswahl von Me-

thoden, die auf bereits durchgeführten empirischen Untersuchungen beruht und eher praxisrelevante Techniken umfasst. Ein großer Teilbereich der Forschung fokussiert in der Quantifizierung von Uncertainty den Bereich der Bayesschen Methoden. Ziel dieser ist immer das Finden der a-posteriori-Verteilung durch die Kombination einer a-priori-Verteilung mit Stichprobendaten. Als a-priori-Verteilung wird dabei i. d. R. die Normalverteilung angenommen. Besonders häufig von den Bayesschen Methoden werden Monte-Carlo Dropouts angewendet. Ein weiterer, großer Teilbereich von Forschung und Praxis richtet sich auf die Ensembles. Aufgrund von Limitationen in beiden Methoden hat sich noch ein dritter Bereich etabliert, in dem einfachere Modelle erweitert werden, um die Uncertainty quantifizieren zu können. Diese Erweiterungen haben bisher jedoch noch einen stärkeren Forschungsbezug und sind selten mit den anderen Bereichen direkt verglichen worden.

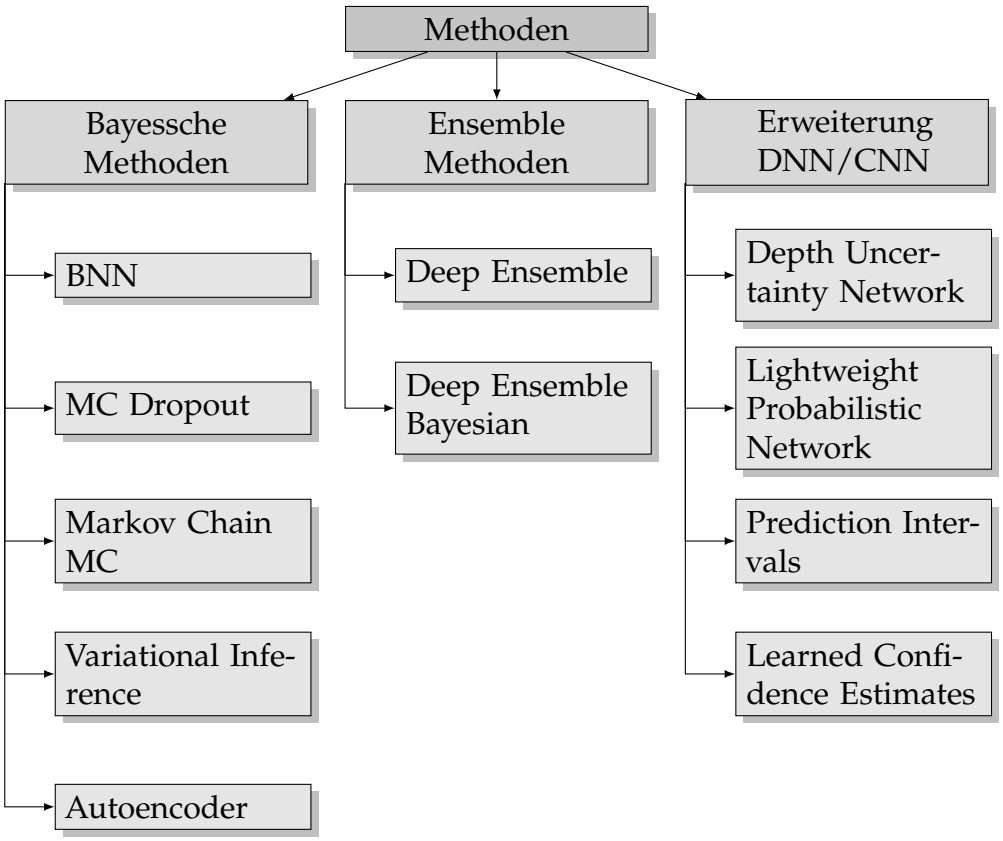

Abbildung 2.10: Übersicht relevanter Methoden

2.3.3.1 *Bayessche Neuronale Netze*

Um probabilistische Vorhersagen zu machen, repräsentieren Bayessche Neuronale Netze (BNN) die Gewichte durch parametrische Verteilungen [53]. Es wird angenommen, dass BNNs unbegrenzter Größe zu exakten Gauß-Prozessen konvergieren [46], was mit hohem Erfolgspotential bewertet wird [2]. Deren direkte Anwendung auf große Datenmengen ist jedoch nicht skalierbar.
Mit BNNs ließe sich die Uncertainty ausgedrückt durch die Parameter sehr gut beschreiben, jedoch führt deren Einsatz gleichzeitig zu einer drastischen Erhöhung zu trainierender Parameter und zu speziellen Optimierungs-Ansätzen [53]. Aufgrund des sehr hohen Berechnungsaufwands beeinflusst dies den Einsatz in der Praxis [18]. Dennoch sind sie sehr robust gegenüber Overfitting und können daher gut mit kleineren Datenmengen trainiert werden [17].
Aufgrund des Erfolgs der BNNs wird in der Forschung immer wieder versucht, eine besser skalierbare Implementierung zu finden. Gast et al. [24] haben bspw. aufgrund der zuvor beschriebenen Nachteile der BNNs vorgeschlagen, die Gewichte des Netzes deterministisch zu halten und nur über Input, Aktivierungen und Output eine Verteilung zu legen.

2.3.3.2 *Monte-Carlo Dropout*

KONZEPTUELLE BESCHREIBUNG Für die Quantifizierung von Uncertainty konnte Hamilton Monte-Carlo [46] in BNNs die bestmögliche Inferenz erreichen. Auch diese Methode hat allerdings aufgrund der eingeschränkten Skalierbarkeit [2] in der Praxis nur bedingte Relevanz. Stattdessen werden Approximationen wie Monte-Carlo Dropouts in Neuronalen Netzen verwendet ([17], [18], [26], [32], uvm.).
Besonders ist, dass randomisierte Dropouts zu Trainings- und Test-Zeit realisiert werden. Vorhersagen werden durch mehrfaches Forward Propagieren (Durchlauf der Daten) durch das Neuronale Netz generiert. Da Monte-Carlo Dropouts also auch während der Vorhersagen auf den Testdaten beibehalten werden, kann dieses Vorgehen als Sampling-Methode eingestuft werden [53].
Die Monte-Carlo Dropouts können mit einer bestimmten Wahrscheinlichkeit jeden Knoten in jedem Hidden Layer betreffen. Auch in der

Backpropagation werden die randomisierten Dropouts beibehalten [1]. Varianten von Dropouts werden in Abbildung 2.11 dargestellt.

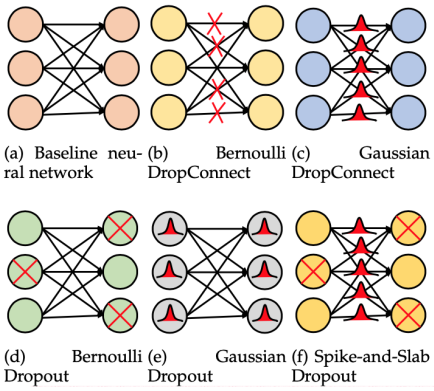

Abbildung 2.11: Varianten der Realisierung von Dropouts [1].

Die Dropouts können entweder auf die Einheit (Knoten) selbst oder auf deren Verbindungen angewendet werden [1]. Zudem unterscheiden sich Dropouts, indem diese entweder basierend auf einer Bernoulli-Verteilung zufällig Elemente ausblenden oder betroffenen Elementen basierend auf einer Gauß-Verteilung zufälliges Rauschen hinzufügen [41]. Üblicherweise werden Dropouts durch Bernoulli Verteilungen realisiert [41] und auf die Knoten eines Neuronalen Netzes angewendet.
Die Model Uncertainty wird gemessen, indem Mittelwert und Varianz der Vorhersagen aus einer bestimmten Anzahl an Forward Passes berechnet werden.
Rein konzeptuell liegt den Monte-Carlo Dropouts damit eine einfache Erklärung zugrunde. Es ist davon auszugehen, dass ein Neuronales Netz für häufig vorhandene Repräsentationen in den Daten Redundanzen entwickelt. Ist eine dieser Redundanzen durch Monte-Carlo Dropouts deaktiviert, kann also ein anderer Teil des Netzes übernehmen, weshalb die Abweichung in den Vorhersagen von häufig auftretenden Phänomenen als gering zu erwarten ist. Im Fall von seltenen Repräsentationen in den Daten kann das Netz weniger Redundanzen bilden. Sobald das Monte-Carlo Dropout eine der wenigen Redundanzen im Netz trifft, sollte es also zu einer hohen Abweichung kommen und damit eine erhöhte Model Uncertainty detektiert werden. Somit kann ein Netz unter Verwendung von Monte-Carlo Dropouts die Mo-

del Uncertainty für seltene Phänomene sowie in den Daten nicht vorhandene Phänomene (OOD Samples) quantifizieren.

EMPIRISCHE UNTERSUCHUNGEN Basierend auf Gal et al. [18] kann Monte-Carlo Dropout zu Trainings- und Test-Zeit als Bayessche Approximation des probabilistischen Gauß-Prozesses interpretiert werden. In den ersten empirischen Studien konnten Monte-Carlo Dropouts in einem Neuronalen Netz auch mit einer sehr kleinen Dropout-Rate alle anderen Modelle im Vergleich des RMSE und der Log-Likelihood in der Test-Phase übertreffen [18]. In späteren Vergleichen konnten diese Erfolge nicht gehalten werden. Besonders auffällig ist in diesen Ergebnissen, dass Neuronale Netze unter Verwendung von Monte-Carlo Dropouts überoptimistische Schätzungen zeigen ([28] , [35], [60], uvm.). Das bedeutet, dass diese Modelle gerade in Bereichen, in denen eine erhöhte Uncertainty gemessen werden müsste, eine zu hohe Sicherheit ausgeben.
Es bestehen daher weiterführende Ansätze, um eine genauere Schätzung der Uncertainty zu erreichen. Durch die Erweiterung eines Neuronales Netzes unter Monte-Carlo Dropouts um einen Autoencoder konnten bessere Schätzungen erreicht werden [69]. Auch Kendall et al. [32] lieferten ein Neuronales Netz, das für jeden Output (konkret: im Umfeld von Computer Vision für jedes Pixel) eine Varianz ausgibt und damit sogar die Data Uncertainty erfassen kann. Jedoch führt diese Methode zu der Notwendigkeit von Anpassungen in der Architektur des Netzes und zu einer Veränderung der Verlustfunktion, was nicht immer praktikabel ist [53].
Auch Wang et al. [63] nutzen Monte-Carlo Dropouts, augmentieren die Inputdaten aber zusätzlich zu Trainings- und Test-Zeit. Damit konnten weitere zu optimistische Einschätzungen der Uncertainty verringert werden [63].
Nachteilig für die Performance ist wie bei allen anderen auf Monte-Carlo Dropouts basierenden Methoden das mehrmalige Forward Propagieren durch das Netz.

2.3.3.3 *Ensemble Methoden*

KONZEPTUELLE BESCHREIBUNG Ensembles werden wie auch das Monte-Carlo Dropout den Sampling-Methoden zugeordnet ([50], [60], [35], [24] uvm.).
Ein typisches Ensemble besteht aus mehreren (M) Modellen, von denen jedes eine Schätzung $\hat{r}_i$ abgibt. Folglich kann für jeden Datensatz aus den Softmax-Ausgaben oder der Schätzung des metrisch skalierten Wertes das Ergebnis als Mittelwert dieser einzelnen Schätzungen berechnet werden [28]:

$$\hat{r} = \frac{1}{m} \sum_{i=1}^{m} \hat{r}_i \tag{2.5}$$

Die Model Uncertainty der Vorhersagen wird in metrisch skalierten Zielvariablen durch die Standardabweichung wiedergegeben [60]:

$$\hat{\sigma} = \sqrt{\frac{1}{m} \sum_{i=1}^{m} (\hat{r}_i - \hat{r})^2} \tag{2.6}$$

Ensembles benötigen für das Training deutlich weniger Daten, da jedes Modell in einem anderen lokalen Minimum konvergiert [28]. In den Bayesschen Methoden hängt die Güte der Quantifizierung von Model Uncertainty von dem Grad der Approximation und der richtigen Auswahl der a-priori-Verteilung ab [35]. Im Gegenteil dazu wird bei den Ensembles davon ausgegangen, dass sich das datengenerierende Modell nicht unter den Modellen des Ensembles finden lässt [1]. Der Erfolg von Ensemble Methoden lässt sich auf Basis der Varianzreduktion durch die Kombination mehrerer schwächerer Modelle erklären [1]. Durch den hohen Rechenaufwand im Training wird allerdings eine umfangreiche zugrunde liegende Architektur gefordert, weshalb Ensembles in der Anwendbarkeit begrenzt sind [2].

EMPIRISCHE UNTERSUCHUNGEN Auch wenn diese Modelle nicht dem Grundprinzip der Bayesschen Statistik folgen, zeigen sie in empirischen Studien eine bessere Performance ([31], [51], uvm.), da sie weniger überoptimistische Schätzungen treffen.
In weiteren empirischen Untersuchungen ([28] , [35], [60], uvm.) konn-

te neben einer verbesserten Performance auch gezeigt werden, dass Ensembles konservativere Schätzungen der Uncertainty gemessen anhand der Kalibrierung[1] liefern.
Für die Erfassung der Model Uncertainty in Regressions-Daten setzten Lakshminarayanan et al. [35] ingesamt fünf Modelle mit je zwei Hidden Layern in dem Ensemble ein, von denen jedes eine punktweise Schätzung des Mittelwertes und der Varianz ausgab (siehe Abbildung 2.12).

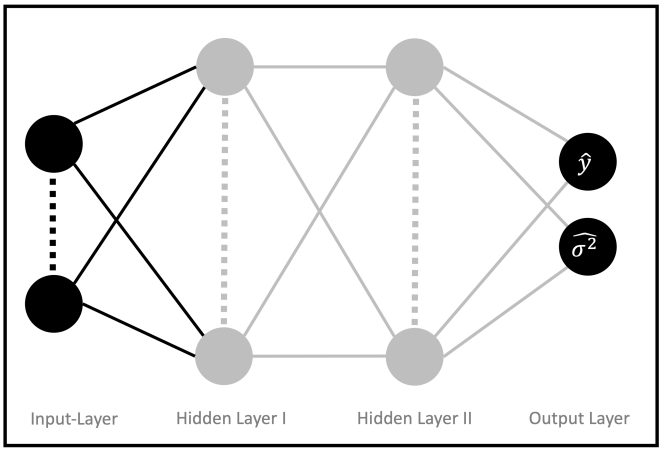

Abbildung 2.12: Schematische Darstellung eines einzelnen Neuronalen Netzes in dem Ensemble nach [35]. Quelle: eigene Darstellung.

Schon bei dieser relativ geringen Anzahl von fünf Modellen zeigten Ensembles eine Verringerung der negative Log-Likelihood (NLL) auf 10 Standard-Datensätzen im Regressions-Umfeld. Für Klassifikations-Daten ist laut [35] die Verwendung der Softmax-Ausgaben für die Ermittlung der Model Uncertainty ausreichend.
Ensembles treffen also im Vergleich konservativere Aussagen [35], ordnen mehr Ausgaben des Modells eine geringe Sicherheit zu [28] und performen bei unbekannten Klassen besser [35], insbesondere bei zunehmender Größe des Ensembles. Außerdem zeigen sie klare Vorteile in der Anwendung bereits trainierter Modelle für die Vorhersage, da sie nur einen Forward Pass benötigen und sich damit für eine Parallelisierung eignen [35].

[1] Bei dieser werden die probabilistischen Vorhersagen der Klasse k eines Modells mit dem tatsächlichen Anteil der Klasse k in allen als k klassifizierten Daten verglichen (d.h. bei einer Wahrscheinlichkeit von 0.8 in der Vorhersage von 10 Datenpunkten sollten 8 Objekte der vorhergesagten Klasse entsprechen).

2.3.3.4 *Methodenvergleich*

Ensemble Methoden und Monte-Carlo Dropouts stehen besonders häufig in einem direkten Vergleich in empirischen Studien, während BNNs wegen der beschriebenen Nachteile in den Hintergrund geraten.
Bezogen auf die Komplexität der Implementierung lassen sich Monte-Carlo Dropouts leicht auf bestehende Modelle anwenden und erhöhen die Komplexität nur geringfügig. Wird aber eine Verarbeitung in Echtzeit benötigt, sind Ensembles das Mittel der Wahl. Dennoch ist deren Anwendung durch große Datenmengen und unzureichende technische Skalierung manchmal nicht praktikabel.
Trotz des Erfolgs von Monte-Carlo Dropouts in BNNs [32] oder Neuronalen Netzen [18] konnte diese Methode in empirischen Studien auf Regressions-Daten durch Ensembles übertroffen werden.
Auch in Klassifikations-Problemen wurden häufig Monte-Carlo Dropouts in Neuronalen Netzen eingesetzt. Gal und Ghahramani ([18], [17]) verwendeten als erste Monte-Carlo Dropout in CNNs zur Verringerung von Overfitting sowie der Ermittlung von Model Uncertainty durch den Vergleich der Softmax-Outputs. Jedoch zeigten auch hier Ensemble-Methoden in nachfolgenden Studien bessere Ergebnisse.
Wird die Kalibrierung als Maß für den Vergleich herangezogen, erzielen Deep Ensembles die zuverlässigsten Ergebnisse für einzelne Datensätze hoher und sehr niedriger Uncertainty, was ebenfalls zu der besten Kalibrierung führt [28]. Nur bei Ausgaben in höherer Sicherheit kann die Kalibrierung von Monte-Carlo Dropouts der von Deep Ensembles gleichgestellt werden [28].
Als eines der besten Ensembles zeigte sich der Vorschlag nach Lakshminarayanan et al. [35], in dem jedes Regressions-Modell sowohl die Schätzung des wahren Wertes als auch der Varianz ausgab. Durch eine Augmentation der Trainingsdaten und eine Erhöhung der Anzahl der Modelle konnte der Erfolg der Ensembles noch weiter gesteigert werden.
Zusammenfassend performen Ensembles also i. d. R. besser als andere Methoden und sollten daher präferiert gegenüber Monte-Carlo Dropouts verwendet werden, wenn eine konservativere Schätzung gefordert ist.

3

METHODIK

Nicht immer liegen sofort die wahren Labels der Daten vor oder treffen manchmal gar nicht ein. In diesen Fällen können alle Algorithmen für die Detektion von Concept Drifts, die von den Labels abhängig sind, nicht angewendet werden. Davon sind nur die unüberwachten Algorithmen nicht betroffen. Die *Uncertainty Drift Detection* (UDD) [4] basiert auf der prädiktiven Model Uncertainty und ist daher nicht abhängig von dem Eintreffen der Labels aus dem Stream. Es konnte gezeigt werden, dass die UDD zuverlässig Concept Drifts erkennt, im Gegensatz zu anderen Methoden deutlich unempfindlicher gegenüber virtuellen Drifts ist und nur die tatsächlich aufgetretenen, realen Drifts detektiert.
Aufgrund des korrelativen Zusammenhangs der Uncertainty von Modellen mit dem Vorhersagefehler [32] und dem Erfolg der UDD basiert die hier vorgestellte Methode der Ensemble Uncertainty Drift Detection (EUDD) ebenfalls auf der prädiktiven Model Uncertainty. Gerade für Concept Drifts ist ein deutlich höherer Einfluss aus der Model Uncertainty im Vergleich zu der Data Uncertainty zu erwarten. In der UDD wurde ein Neuronales Netz unter Monte-Carlo Dropouts eingesetzt, da dies laut den Veröffentlichenden die besten Ergebnisse lieferte. Aus einem Vergleich der Methoden für die Quantifizierung der Model Uncertainty geht jedoch hervor, dass sich Ensemble Methoden in den meisten Fällen besser eignen. Es wird angenommen, dass die Ergebnisse der UDD durch die konservativeren Schätzungen der Model Uncertainty eines Ensembles übertroffen werden können.
Diese Methode wird auf Streams von Daten angewendet, die einer *Observational Sequence* (siehe Arten von Streams unter Kapitel 2.1.1) entsprechen. Es wird also davon ausgegangen, die Daten seien durch einen oder mehrere zugrundeliegende Kontexte bedingt aufgetreten.
Um zwei verschiedene Arten von Detektoren zu prüfen, wird die Uncertainty in einem ADWIN Detektor unter Vergleich zweier Fenster (siehe Kapitel 2.2.6.3) und einem PH Detektor als Methode der se-

quentiellen Analysen (siehe Kapitel 2.2.6.2) ausgewertet. Ensembles empfehlen sich nicht nur aufgrund des Erfolgs in der Quantifizierung von Uncertainty (siehe Kapitel 2.3.3.4), sondern ermöglichen zugleich eine hohe Flexibilität im Umgang mit Concept Drifts (siehe Kapitel 2.2.8.2). Analog zu [35] besteht auch das für diese Methode verwendete Ensemble zunächst aus fünf Modellen, da sich diese Anzahl an Modellen für die Quantifizierung der prädiktiven Uncertainty als ausreichend erwiesen hat. Für den Aufbau des Ensembles wird sich an die Strategie der *Coverage Optimization* gehalten, bei der bei einer vorgegebenen Entscheidungsfunktion möglichst unterschiedliche Modelle gebildet werden (siehe Kapitel 2.2.8.1).
Die Erkennung des Concept Drifts beruht nicht auf dem Vorhersagefehler, sondern nur auf den Features. Analog zu der Einteilung nach [43] (siehe Kapitel 2.2.3) sollten bis auf eine Variante alle Typen von Concept Drifts detektiert werden. Nur wenn es eine Veränderung der bedingten Wahrscheinlichkeit von Zielvariablen und Features gab ($P_t(y|X) \neq P_{t+1}(y|X)$), die sich jedoch nicht in der Verteilung der Features wiederspiegelt ($P_t(X) = P_{t+1}(X)$), kann keine Detektion stattfinden.
Die Modelle des Ensembles geben für jeden neu eintreffenden Datensatz eine Schätzung des wahren Wertes und der Uncertainty ab.
Die prädiktive Uncertainty wird in Klassifikations-Szenarien durch die Shannon-Entropie für jedes einzelne Modell aus den jeweiligen Schätzungen der kategorialen Ausprägungen ermittelt:

$$H\left[\hat{p}(y|X)\right] = -\sum_{k=1}^{K} \hat{p}(y=k|X) - \log_2 \hat{p}(y=k|X) \tag{3.1}$$

In Regressions-Problemen wird ein Ensemble nach [35] eingesetzt, in dem jedes Modell eine Schätzung des wahren Wertes und der Varianz ausgibt. Dazu werden die Modelle mit einer angepassten, auf der negativen Log-Likelihood basierenden Verlustfunktion trainiert.
Die Schätzung des wahren Wertes ergibt sich aus dem Mittelwert der Vorhersagen aller Modelle.
Auch aus den Ergebnissen der prädiktiven Uncertainty jedes einzelnen Modells aus dem Ensemble wird der Mittelwert gebildet und dieser in dem jeweiligen Detektor ausgewertet. Auf die Detektion ei-

nes Concept Drifts folgt ein *Replacement Learning* (siehe Kapitel 2.2.4), indem entweder ein Re-Training des gesamten Ensembles oder einzelner Modelle vorgenommen wird. Somit werden bestehende Modelle durch Modelle gleicher Architektur ersetzt, die mit einer Kombination der neu eingetroffenen Daten und der anfänglich genutzten Trainingsdaten angepasst werden. Damit wird eine aktive Strategie der Modellanpassung durch einen *Triggered Rebuild* als Reaktion auf einen Concept Drift verfolgt (siehe Strategien der Modellanpassung in Kapitel 2.2.7.1).
Es wird davon ausgegangen, dass die Anpassung einzelner Modelle gerade bei wiederkehrenden Mustern zu einer besseren Performance des Ensembles führt. Dadurch, dass nicht alle Modelle an die neuen Daten angepasst würden, werden vorige Informationen länger beibehalten. Sollte ein früherer Kontext erneut auftreten, ist es also möglich, dass ältere, nicht angepasste Modelle mit diesen Daten besser umgehen können. Liegen keine wiederkehrenden Kontexte vor, verspricht eine vollständige Anpassung des Ensembles eine bessere Performance des Modells auf nachfolgenden Daten. Verglichen werden also die Strategien der vollständigen Anpassung aller im Ensemble enthaltenen Modelle als strukturelle Veränderung des Ensembles (im folgenden *retrain-all* genannt) und die Anpassung des schlechtesten Modells (Replace the Loser, im folgenden *retrain-worst* genannt). Letzteres dient dazu, eine Strategie analog zu dem Prinzip der aktualisierten Trainingsdaten nachzuahmen (siehe Lernmethoden für Ensembles in Kapitel 2.2.8.2). Das schlechteste Modell wird je nach Datengrundlage entweder durch die höchste Entropie (siehe 3.1) oder die höchste Varianz der Ergebnisse der bisher eingetroffenen Daten identifiziert.
Für die Detektion von Concept Drifts werden beide Detektoren vor der Anwendung parametrisiert. Sowohl der ADWIN als auch der PH Algorithmus bestehen aus einem fixierten Fenster fester Größe von Daten, auf denen die Parametrisierung ausgeführt wird und einem flexiblen Fenster das neu eintreffende Daten enthält.
Der Algorithmus 1[1]der EUDD kann nach abgeschlossenem initialem Training des Ensembles und der Wahl des geeigneten Parameters δ für den ADWIN- oder PH-Detektor folgendermaßen dargestellt wer-

den:

Algorithm 1 Uncertainty Ensemble Drift Detection

Input: Trainiertes Ensemble $\mathcal{E}$, Stream $\mathcal{D}$, Trainings-Daten $\mathcal{D}_{tr}$, Parametrisierter Detektor

Output: Vorhersage $\hat{y}_t$ zu Zeitpunkt t

repeat
  Aufnahme eintreffender Instanzen X_t
  for each Model $\mathcal{M} \in \mathcal{E}$ **do**
    $\hat{y}_{Mt}, U_{Mt} \leftarrow$ M.predict(X_t)
  end for
  $\hat{y}_t \leftarrow \frac{1}{M} \sum_{t=1}^{M} \hat{y}_{Mt}$
  $\bar{U}_t \leftarrow \frac{1}{M} \sum_{t=1}^{M} \hat{U}_{Mt}$
  Gebe $\bar{U}_t$ in den Detektor:
  if Detektor einen Drift detektiert **then**
    Rufe die zuletzt aktuellen Labels y_{recent} ab
    if *retrain-all*==True **then**
      for each Model $\mathcal{M} \in \mathcal{E}$ **do**
        $\mathcal{M}$.train($\mathcal{D}_{tr} \cup \mathcal{D}_{recent}$)
      end for
    else if *retrain-worst*==True **then**
      Uncertainty $\leftarrow$ [U_t for $\mathcal{M}$ in $\mathcal{E}$]
      $MAX \leftarrow$ max(Uncertainty)
      $\mathcal{E}[MAX]$.train($\mathcal{D}_{tr} \cup \mathcal{D}_{recent}$)
    end if
    D ends

Hier wird das Re-Training gemeinsam mit den zuletzt aktuellen Daten und dem anfänglichen Trainingsdatensatz ausgeführt. Dies kann variiert werden.
Die prädiktive Model Uncertainty U entspricht in Klassifikations-Problemen der Entropie und in Regressions-Problemen der Varianz.

[1] Darstellung in Anlehnung an [4]

4

ERGEBNISSE

4.1 EXPERIMENTELLER AUFBAU

Alle nachfolgend beschriebenen Ergebnisse wurden auf einem MacOS Laptop mit einem Intel Core *i*5 2 GHz Prozessor, 16 GB 3733 MHz *LPDDR4X* RAM durchgeführt. Die unter Python3.7 verwendeten Module und zugehörige Versionen können dem Anhang A.2 entnommen werden.

4.2 DATENSÄTZE

4.2.1 *Synthetische Datensätze*

Um die EUDD Methode zu testen und deren Wirksamkeit zu überprüfen, werden für Regression als auch Klassifikation synthetische Datensätze eingesetzt. In synthetischen Datensätzen lassen sich kontrolliert Concept Drifts sowie auch virtuelle Drifts generieren. Insbesondere von Vorteil ist auch die Simulation verschiedener Arten von Drifts.
Für die Evaluation in Regressions-Problemen werden die **Friedman Regression** Datensätze [16] verwendet. In diesen sind 10 Features enthalten, die alle durch eine Gleichverteilung im Intervall $[0, 1]$ generiert worden sind.

Wie in Abbildung 4.1 erkennbar, werden Features so erzeugt, dass nur drei relevant für die Vorhersage der Zielvariablen sind, während die anderen den Daten ein Rauschen hinzufügen [4]. Im Fall des Datensatzes **Friedman Target (3)** enthalten drei dieser nicht in Zusammenhang mit der Zielvariablen stehenden Features einen virtuellen Drift. Unter ähnlichem Vorgehen werden drei weitere Datensätze erzeugt, die inkrementelle Drifts (**Friedman inkrementell**, **Fried-**

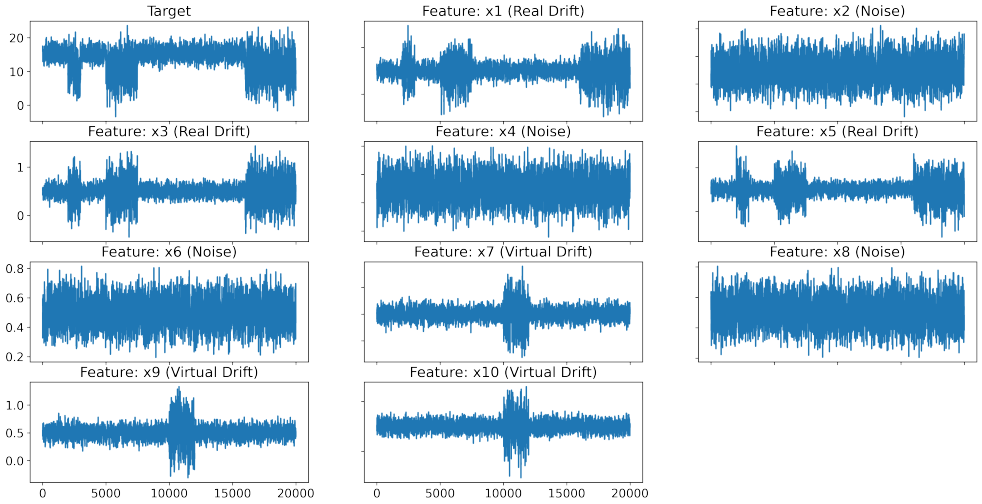

Abbildung 4.1: Friedman Target (3) Regressions-Datensatz

man No Return) oder graduelle Drifts (**Friedman gradual**) enthalten .

Auch für die Klassifikation wurden Daten von Baier et al. [4] verwendet, die angelehnt an Gama et al. [21] für die **Mixed** Datensätze sechs Features, zwei mit binären Ausprägungen und vier aus einer diskreten Verteilung, erzeugten. Auch hier wird den Daten durch Features ein Rauschen hinzugefügt und es kommt zu virtuellen Drifts (siehe Abbildung 4.2).

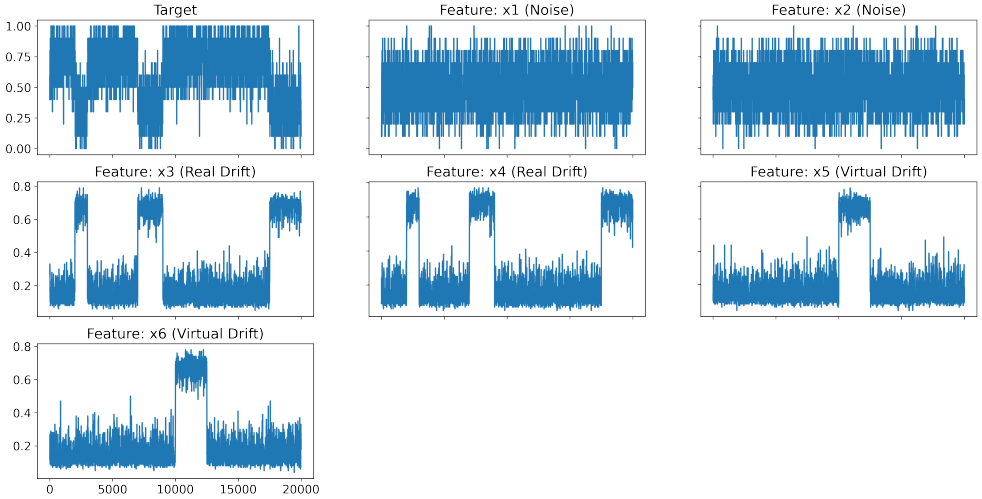

Abbildung 4.2: Mixed Target (3) Klassifikations-Datensatz

[o] Alle Datensätze stammen aus der Veröffentlichung nach Baier et al. [4]

Auch in den Mixed Daten können abrupte Drifts (**Mixed Abrupt**), inkrementelle Drifts (**Mixed inkrementell**) oder virtuelle Drifts, zusätzlich zu den drei realen Drifts (**Mixed Target (3)**), simuliert werden.
Sowohl für die Friedman als auch für die Mixed Datensätze werden wie beschrieben bestimmte Features sukzessiv variiert, um einen realen oder virtuellen Concept Drift zu erzeugen. Weitere Abbildungen der Verteilungen finden sich in Anhang A.3.1. Zusammenfassend wurden folgende synthetische Datensätze generiert und für die Validierung der Methode verwendet:

Tabelle 4.1: Synthetische Datensätze

	Anzahl Features		**Anzahl Drifts**	
Datensatz	relevant	irrelevant	real	virtuell
Friedman inkrementell	3	7	1	0
Friedman gradual	3	7	4	0
Friedman Target (3)	3	7	3	1
Friedman No Return	3	7	1	0
Mixed Abrupt	2	4	1	2
Mixed inkrementell	2	4	1	0
Mixed Target (3) [1]	2	4	3	1

4.2.2 *Real-World Datensätze*

Um eine Vergleichbarkeit zu der UDD gewährleisten zu können, werden für die EUDD die gleichen Real-World Datensätze für die Evaluation verwendet.
Die Daten, die in Klassifikations-Problemen angewendet werden, stammen aus dem Concept Drift Repository nach Souza et al. [54]. Von den insgesamt sieben Real-World Klassifikations-Datensätzen stammt mehr als die Hälfte aus einem kontrollierten Experiment. In diesem sind die Bewegungen sechs verschiedener Insektenarten

[1] Die Zahl in den Klammern steht für die drei realen Drifts

durch Sensoren erfasst worden und durch Temperaturschwankungen verschiedene Arten von Concept Drifts künstlich herbeigeführt worden. Aufgabe ist die Klassifikation der Insekten anhand der aufgezeichneten Bewegungsdaten. Durch eine plötzliche Temperaturänderung (**Insects Abrupt**) ist mit einem abrupten Drift zu rechnen, wohingegen bei einer allmählichen Änderung auch ein allmählicher Drift (**Insects Inc**) zu erkennen ist. Auch wiederkehrende Muster wurden berücksichtigt, wobei durch drei Zyklen inkrementeller Temperaturschwankungen in Kombination mit plötzlichen Veränderungen ein Datensatz (**Insects IncAbr**) entstanden ist und wiederkehrende Muster sich durch zyklische Temperaturveränderungen ebenfalls in einem Datensatz (**Insects IncReo**) finden. Alle Insekten-Datensätze liegen normiert und balanciert vor.
Zu den anderen Klassifikations-Datensätzen gehört der **KDDCUP99** Datensatz, in dem TCP Verbindungen aus einem Netzwerk erfasst worden sind und vorhergesagt werden soll, ob die Verbindung normal ist oder eine von 22 verschiedenen Varianten von Attacken vorliegt. Ebenfalls verwendet worden ist der **Gassensor** Datensatz, bei dem sechs verschiedene Arten von Gasen klassifiziert werden sollen und Sensor Drifts (virtuelle Drifts) sowie Concept Drifts vorhanden sind. Zuletzt für die Klassifikation anzuführen ist der **Electricity** Datensatz, in dem Verbrauchsdaten und Marktpreise des australischen New South Wales Elektrizitäts-Markts zu finden sind. Aufgabe ist vorherzusagen, ob der Preis im Vergleich zu den letzten 24 Stunden ansteigen oder absinken wird.
Real-World Datensätze für Concept Drifts in Regressions-Problemen lassen sich weniger häufig finden. Es werden zwei Datensätze für den Vergleich zu der UDD eingesetzt. In dem **Air Quality** Datensatz soll die Benzol-Konzentration vorhergesagt werden und Concept Drifts liegen durch saisonale Veränderungen vor. In dem **Bike Sharing** Datensatz werden stündliche Vorhersagen von Fahrrad-Leihen gefordert. Concept Drifts sind hier durch den Einfluss unterschiedlicher Wetterlagen, Wochentage oder Feiertage zu erwarten.
Neben den bisherigen Benchmark-Datensätzen werden die **New York Taxi** Daten als weiterer Anwendungsfall betrachtet. Aufgabe ist die Vorhersage der Anzahl von Fahrten pro Tag. Hier ist mit allmählichen Drifts durch saisonale Effekte und mit abrupten Drifts durch verschie-

dene Einflüsse zu rechnen. Letztere können durch Wetterlagen oder andere umweltbedingte Faktoren ausgelöst werden. Insbesondere ist aber auch durch die Corona-Pandemie eine deutliche Veränderung der Taxifahrten in betroffenen Monaten zu beobachten (siehe Abbildung 4.3). Ab dem 01. März 2020 befand sich die Stadt in einem Lockdown. Nach diesem abrupten Drift wird bei Auflösung der Maßnahmen eine langsame Veränderung in den Normalzustand erwartet, die einem graduellen oder inkrementellen Drift entspricht.

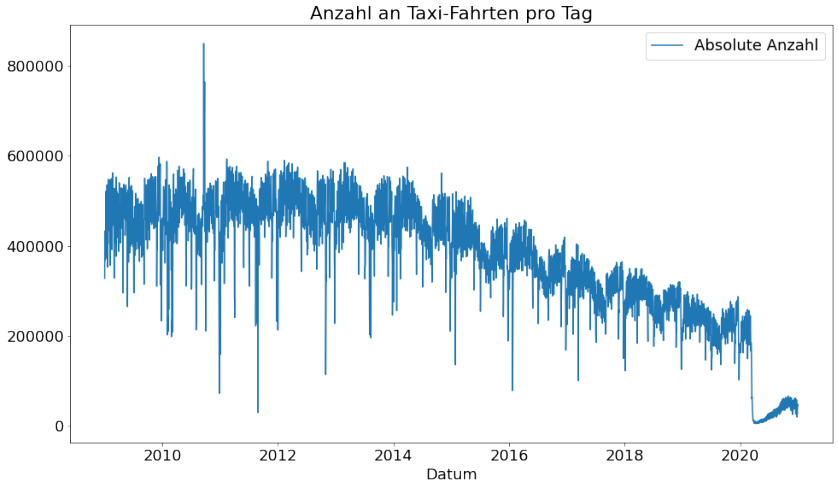

Abbildung 4.3: Absolute Anzahl Taxifahrten in New York pro Tag

Die New York Taxi Daten wurden mit Informationen zu Feiertagen[2] und Wetterdaten[3] angereichert.
Eine detaillierte Übersicht zu den verwendeten Real-World Datensätzen, zu deren Anzahl an Features sowie weiteren Beschreibungen lässt sich in Anhang A.2 finden.
Wenn Bereinigungen oder Feature Engineering für die Verwendung eines Datensatzes notwendig sind, werden diese direkt bei dem Einlesen über die dem Datensatz zugehörige *data_loader()* Funktion vorgenommen.
In der Vorverarbeitung der Daten wurde auch mittels einer Principal Component Analysis (PCA) eine Verringerung der Anzahl der Featu-

[2]Datensatz aus Kaggle: https://www.kaggle.com/donnetew/us-holiday-dates-2004-2021
[3]Download der Daten der Wetterstation *GHCND* : *USW*00094728 im Central Park von https://www.ncdc.noaa.gov/cdo-web/search

res getestet. Bei dieser wurden die Hauptkomponenten verwendet, die 95 Prozent der Varianz erklären konnten. Da diese Verringerung der Komplexität keinen nennenswerten Einfluss auf die Ergebnisse der verschiedenen Datensätze mit häufig ohnehin kleinerer Anzahl von Features zeigte, wurde im weiteren Verlauf keine PCA für die Vorbereitung der Daten angewendet.

4.3 ARCHITEKTUR DER MODELLE

Es werden Neuronale Netze eingesetzt, da diese die benötigte Flexibilität aufweisen, die insbesondere für die Anpassung der Regressions-Modelle wichtig ist, damit im Output-Layer die Varianz quantifiziert werden kann.
Wie in Abbildung 4.4 erkennbar, besteht jedes im Ensemble befindliche Modell aus einem Input-Layer, dessen Anzahl von Neuronen die Features des Datensatzes bestimmen, vier Hidden Layern und einem Output Layer.

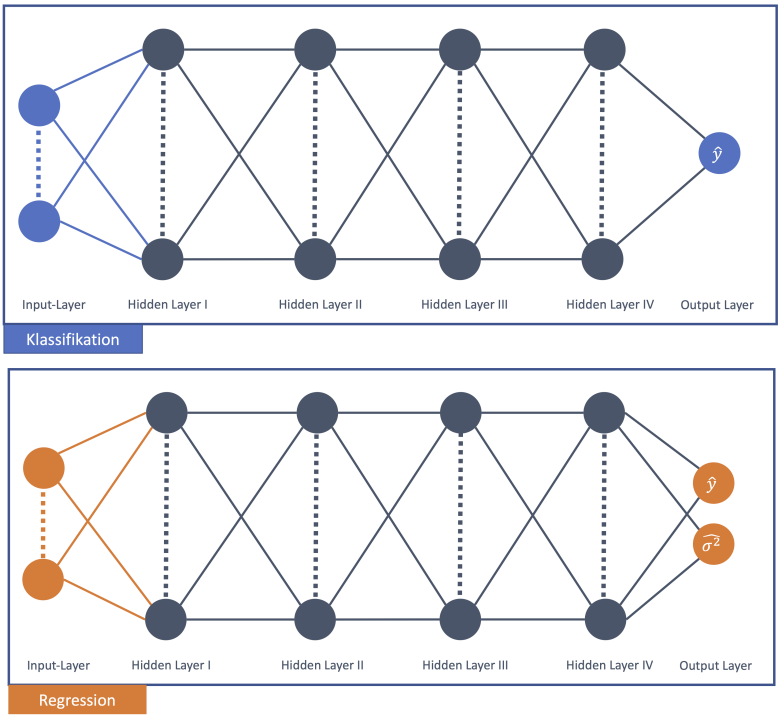

Abbildung 4.4: Schematische Darstellung der Neuronalen Netze im Ensemble je statistischer Problemstellung

Weitere Hidden Layer in den Modellen aufzunehmen konnte keine deutliche Verbesserung erreichen.
Der Output Layer für Regressions-Probleme erzielt Vorhersagen für den wahren Wert und die Varianz und besteht somit aus zwei Neuronen. In den Modellen für Klassifikations-Daten entspricht die Anzahl an Neuronen der Anzahl der Ausprägungen in der Zielvariablen.
Für jeden Datensatz wurde die Anzahl der Neuronen je Hidden Layer der im Ensemble befindlichen Modelle durch den Keras Tuner [48] optimiert (siehe Tabelle 4.2).

Tabelle 4.2: Anzahl der Neuronen der Hidden Layer des Modells je Datensatz

Datensatz	**Anzahl je Hidden Layer**	**Best (Worst)** [4]
Friedman inkrementell	56	0.595 (1.267)
Friedman gradual	64	0.357 (0.858)
Friedman Target (3)	56	0.292 (0.880)
Friedman No Return	64	6.562 (20.088)
Mixed Abrupt	64	0.778 (0.756)
Mixed inkrementell	44	0.982 (0.971)
Mixed Target (3)	40	0.998 (0.996)
Air Quality	24	0.940 (1.508)
Bike Sharing	12	23.503 (26.443)
New York Taxi	36	1273.92 (1251.64)
Insects Abrupt	56	0.654 (0.651)
Insects Inc	64	0.6035 (0.60)
Insects IncAbr	56	0.690 (0.694)
Insects IncReo	64	0.571 (0.545)
KDDCUP99	24	0.996 (0.990)
Gassensor	64	0.65 (0.41)
Electricity	64	0.65 (0.45)

[4] Metrik (Accuracy oder MAE) des besten und schlechtesten Modells

Berichtet werden auch die Ergebnisse des besten und schlechtesten Modells anhand von Metriken, um den Einfluss der Optimierung zu überprüfen. Zu beachten ist bei dem Vergleich der nachfolgenden Ergebnisse, dass der Keras Tuner im Gegensatz zu der EUDD immer den vollständigen Datensatz zur Verfügung stehen hatte. Festzustellen ist, dass der Unterschied des Mean Average Error (MAE) in Regressions-Daten in den meisten Fällen zwischen dem besten und dem schlechtesten Modell deutlich größer ist als die Unterschiede der Accuracy bei den Klassifikations-Daten. In den meisten Regressions-Modellen wird zudem nach der Optimierung eine hohe Anzahl an Neuronen nahe der maximalen Anzahl von 64 angegeben, während diese in vielen Klassifikations-Modellen deutlich niedriger ist.

4.4 ALGORITHMUS UND IMPLEMENTIERUNG

Die Implementierung ist angelehnt an das öffentlich verfügbare Repository von Baier et al. [4][5]. Der in der Implementierung geschriebene Code ist in einem GitLab Repository zu finden[6]. Die Haupt-Datei, in der die durchzuführenden Detektoren und Anpassungsstrategien sowie das Ensemble und dessen einzelne Modelle definiert werden, ist die Python-Datei **Performance_Evaluation.py** (siehe Anhang A.4.1). Nach dem Aufruf des jeweiligen Datensatzes finden die Bereinigung und eventuell das Feature Engineering statt.
Anpassungen des Ensembles und der Modelle können in Zeile $37 - 69$ vorgenommen werden. Sowohl für Regressions- als auch Klassifikations-Probleme wurden die Modelle durch das Modul *Keras* [9] realisiert. Ein Modell wird durch eine Funktion definiert, die der Klasse des Ensembles übergeben wird. Dazu wird die Funktion *classifier_model()* (siehe Anhang A.4.2) oder *regression_model()* (siehe Anhang A.4.3) aufgerufen, die wiederum eine Funktion enthält, mit der ein Modell aufgebaut wird (*create_model()*). Ohne diese Verschachtelung würde bei mehrfachen Aufrufen der einfachen Funktion immer das gleiche Modell aufgebaut und damit nicht die gewünschte, heterogene Struktur des Ensembles erreicht werden.
In der Klasse des Ensembles wird neben weiteren Konfigurationen angegeben, wie viele Modelle (*num_members*) das Ensemble enthalten soll.
Je nach Anzahl der Modelle im Ensemble wird für die Bildung des gesamten Ensembles die Funktion für ein einzelnes Modell entsprechend oft aufgerufen und die Modelle in einer Liste abgelegt. Dabei kann gleichzeitig die Anzahl der Hidden Layer angegeben werden. Aufgrund unterschiedlicher Konfigurationen des Compilers und dessen Verlustfunktion sowie der Vorhersagen wird zwischen Regression und Klassifikation differenziert. Daher ist in der Implementierung für beide Arten eine eigene Klasse des Ensembles zu finden. Die Ensemble-Klasse enthält mehrere Funktionen:

[5] Link zum Repository: https://github.com/anonymous-account-research/uncertainty-drift-detection
[6] Link zum Repository: https://gitlab.fbi.h-da.de/istppmoel/ensemble-uncertainty-drift-detection

FIT() Dient dem initialen Training aller im Ensemble befindlichen Modelle.

RETRAIN_ALL_MEMBERS() Re-Training aller im Ensemble befindlichen Modelle.

RETRAIN_WORST_MODEL() Re-Training des schlechtesten Modells. Das schlechteste Modell wird anhand der höchsten gemittelten Entropie oder Varianz aller getätigten Vorhersagen je Batch bestimmt.

PREDICT() In Regressions-Modellen wird eine Schätzung des wahren Wertes und der Varianz von denen in das Modell gegebenen Testdaten berechnet. An den ADWIN Detektor wird die Standardabweichung weitergegeben. In Klassifikations-Modellen wird die Schätzung des wahren Wertes, die Entropie und die Schätzung der Wahrscheinlichkeit weitergegeben.

PREDICT_FOR_EACH_MEMBER() Jedes Modell gibt eine Schätzung des tatsächlichen Wertes und der Uncertainty (Entropie oder Varianz) ab. Die Uncertainty der Testdaten wird je Modell gemittelt und ein Array in der Größe der Anzahl der Modelle weitergegeben.

Als Anzahl von Modellen pro Ensemble wurden analog zu Lakshminarayanan et al. [35] anfangs fünf gewählt.
Zu Beginn des Trainings der EUDD werden auf den ersten fünf Prozent des zur Verfügung stehenden Datensatzes die Modelle in 500 Epochen trainiert. Möglich ist zudem eine Standardisierung der Daten (über den Parameter *scale_data*) vor der weiteren Verarbeitung durch das Modell. Auf diese wurde jedoch in den Experimenten verzichtet, da einige Datensätze bereits normiert vorliegen. Das Regressions-Modell wird durch eine NLL-Verlustfunktion angepasst[7]. Diese ermöglicht das Training durch Schätzung des wahren Wertes und gleichzeitig die Anpassung eines Modells für die Vorhersagen, um den wahren Wert und die Varianz zu schätzen. In den Klassifikations-Modellen können gängige Verlustfunktionen verwendet werden. Hier wurde die Kreuzentropie aus Keras verwendet.

[7] Code angelehnt an: https://github.com/mvaldenegro/keras-uncertainty/blob/master/keras_uncertainty/models/DeepEnsembleRegressor.py

Wie in Abbildung 4.5 dargestellt, dienen die nachfolgenden 10 Prozent der Daten der Parametrisierung des ADWIN oder PH Detektors.

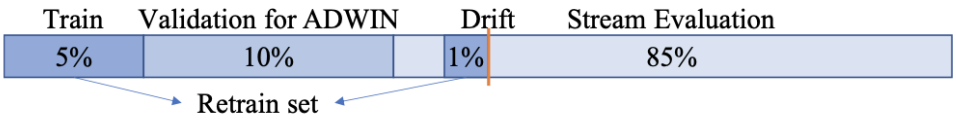

Abbildung 4.5: Partitionierung des Streams [4]

Hier gilt es, einen optimalen Wert für den Parameter δ zu finden, mit dem die Abweichungen in dem Detektor geprüft werden (Verwendung des Detektors analog zu der Beschreibung in Kapitel 2.2.6.3). Anhand einer vorgegebenen Anzahl zu erkennender Concept Drifts wird der Parameter δ festgelegt, der im weiteren Verlauf bei einem signifikanten Anstieg der prädiktiven Unsicherheit für die Erkennung eines Concept Drifts sorgt. Der ADWIN Detektor wird aus dem Modul **scikit-multiflow** [42] verwendet, in welchem ein Default-Wert von $\delta = 0.002$ zugrunde gelegt wird. In der Suche nach dem optimalen Wert wird der ADWIN-Detektor mit jeweils zwei Werten ausgeführt (Startwerte $\delta = 0.1$ und $\delta = 0.01$). Entspricht die Anzahl gefundener Drifts der Anzahl zu detektierender Drifts, wird der Algorithmus angehalten und derjenige Parameter gewählt, der zu einem korrekten Ergebnis geführt hat. Werden mehr Drifts als angegeben detektiert, werden die Werte verringert und erneut eine Detektion mit dem ADWIN Detektor durchgeführt. Der einem Grid-Search ähnelnde Algorithmus wird erst angehalten, wenn die Anzahl zu detektierender Drifts der Anzahl gefundener Drifts entspricht. Wenn jedoch bereits im ersten Durchlauf keine Drifts gefunden worden sind, wird der Parameter δ auf den Default-Wert gesetzt. Das gleiche Verfahren wird ebenfalls für den PH Detektor und dessen Parameter durchgeführt.
Diese Parametrisierung wurde für jeden Datensatz umgesetzt. Die Ergebnisse können nachstehender Tabelle 4.3 entnommen werden.

Tabelle 4.3: ADWIN-Parametrisierung je Datensatz

Datensatz	**Optimaler Parameter** δ
Friedman inkrementell	0.002
Friedman gradual	0.002
Friedman Target (3)	0.1
Friedman No Return	0.002
Mixed Abrupt	0.01
Mixed inkrementell	0.002
Mixed Target (3)	0.002
Air Quality	0.0001
Bike Sharing	0.01
New York Taxi	10^{-05}
Insects Abrupt	0.002
Insects Inc	0.002
Insects IncAbr	0.1
Insects IncReo	0.1
KDDCUP99	0.002
Gassensor	0.1
Electricity	10^{-16}

Auf den restlichen 85 Prozent der gestreamten Daten werden nach der Parametrisierung Vorhersagen getroffen. Hier kann eine Detektion von Concept Drifts stattfinden. Bei dem Auftreten eines Concept Drifts wird je nach Modellanpassungs-Strategie das schlechteste Modell oder das ganze Ensemble mit den zuletzt gestreamten Daten neu trainiert. Dafür wird zu Anfang die Datenmenge für ein Re-Training bestimmt, bei der 1 Prozent der restlichen Daten aus dem Stream verwendet werden sollen. Getestet wird auch der Unterschied

der Modellgüte, wenn für die Anpassung ebenfalls die anfänglichen Trainings-Daten hinzugenommen werden. Damit reduziert sich die Menge der Daten für das Re-Training nicht, sondern erhöht sich um die Anzahl der Daten aus dem Training. Alle für das Re-Training verwendeten Daten werden durch Bootstrapping (Resampling with Replacement) gewonnen. Dies unterstützt die Heterogenität der Modelle.
Das Programm endet, wenn alle Daten aus dem Datensatz durch das Modell gelaufen sind.
Verglichen wird die Strategie eines Triggered Rebuild in Form der EUDD unter Verwendung von zwei verschiedenen Detektoren (ADWIN, PH) mit passiven Ansätzen basierend auf zwei Strategien des Continuous Rebuild. Eine Baseline, in der keine Neuanpassung erfolgt, dient ebenfalls dem Vergleich. Damit werden folgende Strategien zusätzlich zu der EUDD durchgeführt:

1. *No_Retraining (No-Retr.)*: Keine Neuanpassung.
2. *Equal_Distribution (Eq. D.)*: Gleichmäßig verteilte Anpassung des gesamten Ensembles. Die Anzahl an Re-Trainings wird durch die Anzahl an durch den ADWIN Detektor gefundenen Concept Drifts bestimmt.
3. *Randomized (Rand.)*: Zufällig gewählte (d.h. nicht gleich verteilte) Anpassungen des ganzen Ensembles. Anzahl an Retrainings analog zu der *Eq. D.*

Als Metriken für den direkten Vergleich werden in den Regressions-Daten der RMSE und für Klassifikations-Daten der Matthew's Correlation Coefficient (MCC) herangezogen. Beide Metriken wurden gewählt, um eine Vergleichbarkeit zu der UDD zu ermöglichen. Da der MCC nicht so gängig wie andere Metriken in der Anwendung ist, wird zusätzlich zu den Ergebnissen der Klassifikations-Daten die AUC in Tabellen im Anhang geliefert.
Der RMSE ist mit Vorhersage $\hat{y}_t$ und dem wahren Wert y_t zu Zeitpunkt t mit Gesamtheit der Instanzen T definiert als:

$$RMSE = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(\hat{y}_t - y)^2} \tag{4.1}$$

Der MCC für die Klassifikation hingegen setzt die True Positives (TP), True Negatives (TN), False Positives (FP) und False Negatives (FN) in ein Verhältnis. Damit könnte der MCC (auch Phi-Koeffizient) auch von einer Confusion Matrix abgeleitet werden und definiert sich folgendermaßen[11]:

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4.2}$$

Mit der AUC wiederum wird der Flächeninhalt unter der ROC Kurve angegeben [40].

4.5 ANALYSEN

Zur Überprüfung der Wirksamkeit der EUDD wurden für die Probleme der Regression und Klassifikation synthetische Datensätze angewendet. Diese enthalten verschiedene Arten von Drifts und weitere Störeinflüsse aus Features, die den Daten Rauschen hinzufügen.
Nachfolgend wird die EUDD auch auf die genannten Real-World Datensätze angewendet.

4.5.1 *Synthetische Daten*

Aufgrund der im Gegensatz zu den Real-World Daten geringeren Komplexität der synthetischen Daten wurden die Experimente mit einem unveränderten Ensemble aus fünf Modellen durchgeführt. Sowohl für Regressions- als auch Klassifikations-Daten wurden die Modelle mit jeweils vier Hidden Layern aufgebaut. Bei Anpassungen durch Re-Trainings wurden bei allen synthetischen Datensätzen zusätzlich zu den zuletzt eingetroffenen Daten des Streams auch die Trainingsdaten verwendet.

4.5.1.1 *Regression*

Aufgrund dessen, dass die Zeitpunkte der Drifts bekannt sind, kann das Ergebnis gut visuell analysiert werden.
Nachfolgend dargestellt wird der Verlauf der Uncertainty des Ensembles über die Zeit (in Form des Index-Wertes). Reale Drifts werden durch rote Linien gekennzeichnet und virtuelle Drifts in der Farbe Orange. Die Detektionen sind der Abbildung als schwarze vertikale Linien hinzugefügt. Bestenfalls sollte auf jede rote Linie (realer Drift) sofort eine schwarze Kennzeichnung (Detektion) folgen. Auf orange Linien (virtuelle Drifts) sollte es keine Reaktion geben.
Abgebildet wird die Model Uncertainty der *retrain-all* (siehe Abbildung 4.6) als auch der *retrain-worst* (siehe Abbildung 4.7) Strategie. Wie in diesen Abbildungen erkennbar, werden in beiden Anpassungsstrategien der EUDD neben den realen Drifts auch die virtuellen Drifts detektiert.

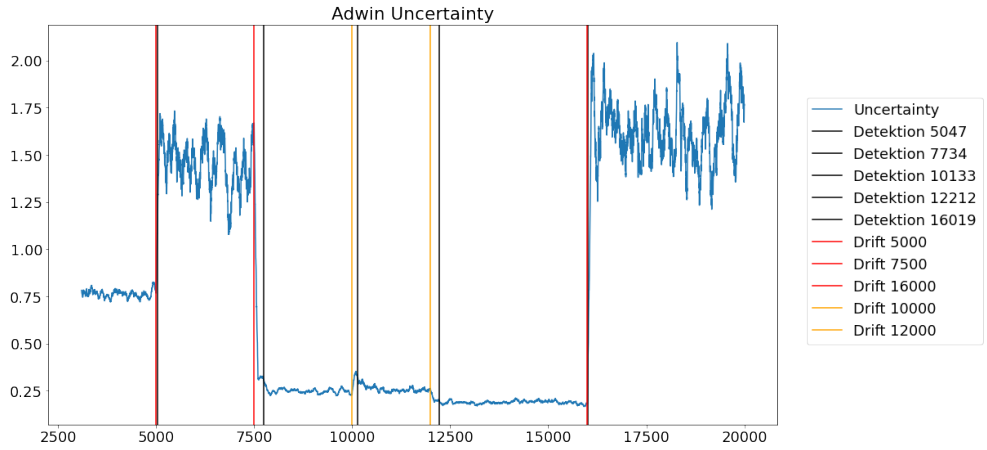

Abbildung 4.6: Model Uncertainty und Detektionen über Zeit bei *retrain-all* der Friedman Target (3) Daten.

Zusätzlich ist ein Unterschied in der Höhe der Model Uncertainty zu verzeichnen. Wird nur ein Modell nach einem Drift weiter angepasst, kommt es zu erhöhter Model Uncertainty im Gegensatz zu dem Ensemble, in dem alle Modelle angepasst werden.

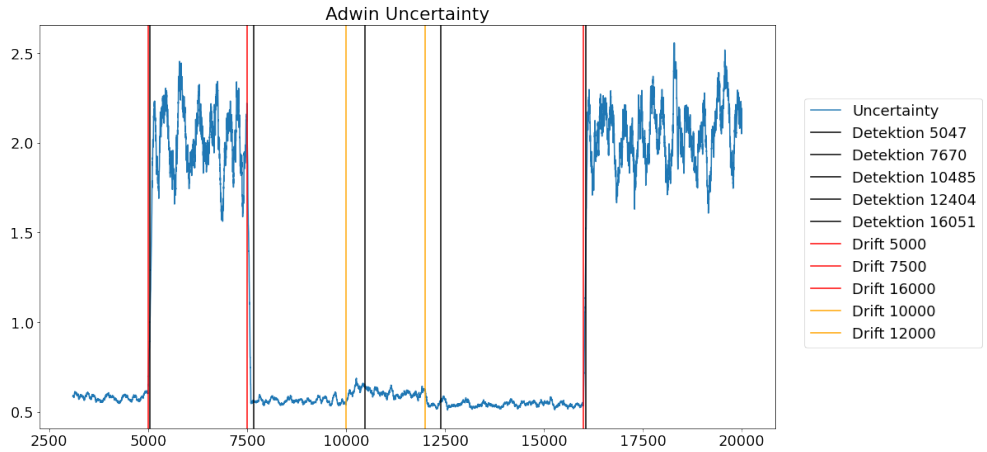

Abbildung 4.7: Model Uncertainty und Detektionen über Zeit bei *retrain-worst* der Friedman Target (3) Daten

Das Muster der Model Uncertainty zeigt sich auch in der Darstellung des RMSE (siehe Abbildung 4.8), visualisiert als gleitendes Fenster über die zuletzt eingetroffenen 200 Daten und deren Vorhersagen. Hier kommt es während der Zeit, in der ein realer Drift vorliegt, zu einem deutlich erhöhten RMSE. Obwohl das Ensemble auf den virtuellen Drift reagiert, führt eine Anpassung der Modelle nur zu einem minimalen Anstieg des RMSE.

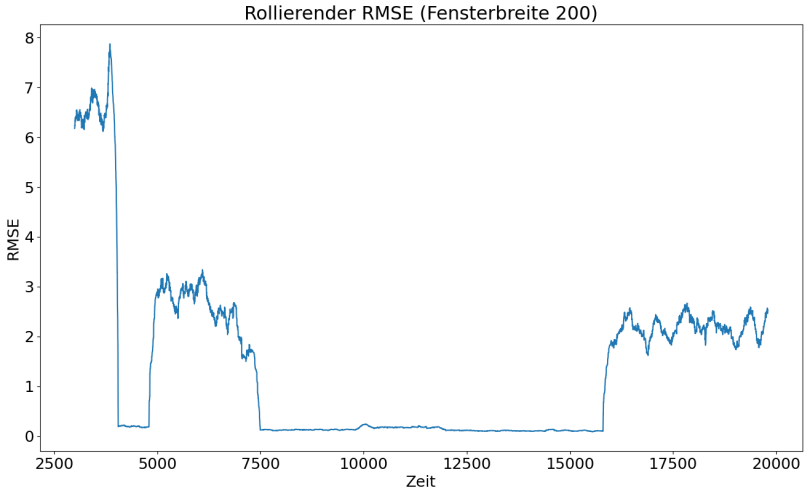

Abbildung 4.8: Rollierender RMSE der Friedman Target (3) Daten bei *retrain-all*

Der zu Anfang des Streams erhöhte RMSE lässt sich mit der geringeren Menge an Daten begründen, die zu dem Zeitpunkt für eine Modellanpassung vorgelegen haben und sollte demnach nicht überbewertet werden.
Für die Friedman-Datensätze wurden relevante Metriken ermittelt (siehe Tabelle 4.4). Es werden die Anzahl von fälschlicherweise oder nicht detektierten Drifts (Fehler) sowie die mittlere Zeit zur Detektion tatsächlicher Drifts (Zeit)[8] je Datensatz angegeben.
Bei den Friedman No Return Daten kam es schon vor dem inkrementellen Drift zu einer Detektion. Diese muss als Fehler gewertet werden. Die anderen, nachfolgenden Detektionen resultieren aus der sich stetig verändernden Verteilung und sind damit eher Folge des andauernden Drifts als ein Fehler.

[8] Wird anhand der Indexwerte der Detektion und des tatsächlichen Drifts ermittelt.

Tabelle 4.4: Ergebnisse der synthetischen Regressions-Datensätze mit M=5

Datensatz	Strategie	Fehler	Zeit
Friedman Target (3)	*retrain-all*	2	100
	retrain-worst	2	89.33
Friedman inkrementell	*retrain-all*	9	47
	retrain-worst	8	47
Friedman gradual	*retrain-all*	2	1843.5
	retrain-worst	3	5379
Friedman No Return	*retrain-all*	1 (9)	78
	retrain-worst	1 (10)	46

Auf drei der vier Datensätze ist die Reaktionsgeschwindigkeit der *retrain-worst* Strategie höher als die der Modellierung mit *retrain-all* Anpassung. Trotz der relativ langen Zeit bis zu einer Detektion reagiert das Ensemble im Fall der Friedman Target (3) Daten schneller als das Modell der UDD mit einer durchschnittlichen Zeit von 132.7 bis zu der Detektion.
Das scheinbar bessere Ergebnis der *retrain-worst* Strategie kann jedoch im Vergleich zu der *retrain-all* Strategie anhand des RMSE nicht gehalten werden. Nachstehende Tabelle 4.5 enthält dessen Ergebnisse sowie die Anzahl an Retrainings je Strategie in Klammern. Generell gilt hier: je kleiner der RMSE ausfällt, desto besser ist die Modellanpassung.

Wie zu erkennen, übertrifft die *retrain-all* Strategie die Ergebnisse der *retrain-worst* Strategie in den meisten Fällen. Bei zwei Datensätzen beträgt der RMSE im Vergleich mehr als das 20-fache.
Die Ergebnisse der EUDD mit dem ADWIN Detektor können jedoch durch die Nutzung eines PH Detektors noch übertroffen werden. Bei zwei Datensätzen scheint dieser jedoch eine sehr empfindliche Reaktion auf Unterschiede in der Uncertainty zu zeigen, was zu einer deutlich höheren Anzahl an Re-Trainings und damit Verlusten in der Performance führt. Graduelle Drifts verarbeiten beide Detektoren nicht besser als regelmäßiges Re-Training (Eq. D.).
Es lässt sich also feststellen, dass die EUDD in einigen Fällen zu sensibel auf geringe Veränderungen der Uncertainty anspricht und damit

Tabelle 4.5: RMSE (Anzahl Re-Trainings) der synthetischen Regressions-Datensätze mit M=5

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Friedman Target (3)	1.68(5)	2.69(5)	**1.23**(**2**)	2.93(0)	1.78(5)	2.22(5)
Friedman inkrementell	11.04(10)	226.19(9)	**6.59**(**52**)	256.69(0)	16.55(10)	77.85(10)
Friedman gradual	12.06(2)	9.86(2)	12.24(1)	8.62(0)	**8.55**(2)	9.57(2)
Friedman No Return	10.14(11)	289.59(12)	**6.72**(**28**)	335.89(0)	22.85(11)	29.24(11)

auf virtuelle Drifts reagiert. Dennoch wurden bis auf die graduellen Drifts (siehe Abbildung A.4 im Anhang) alle anderen realen Drifts erkannt. In den Regressions-Daten kann ein PH Detektor die Ergebnisse übertreffen, führt allerdings in der Hälfte der Datensätze zu einer deutlichen Erhöhung der Anzahl von Re-Trainings.

4.5.1.2 *Klassifikation*

In den **Mixed Abrupt** Daten wurden alle vier realen Drifts durch den Detektor gefunden (siehe Abbildung 4.9). Im Gegensatz zu den vorigen Ergebnissen findet in diesen Klassifikations-Daten bei den virtuellen Drifts keine Detektion statt.

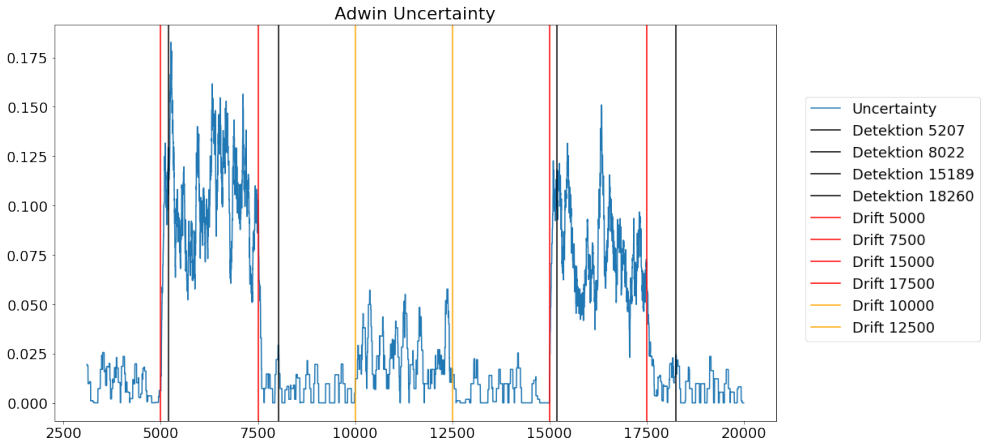

Abbildung 4.9: Model Uncertainty und Detektionen über Zeit bei *retrain-all*.

Je nach Anpassungsstrategie unterscheidet sich nur die Höhe der prädiktiven Model Uncertainty, jedoch nicht die Anzahl der Detektionen. Wird nur das schlechteste Modell angepasst, nimmt die Model Uncertainty deutlich höhere Werte an (siehe Abbildung 4.10).

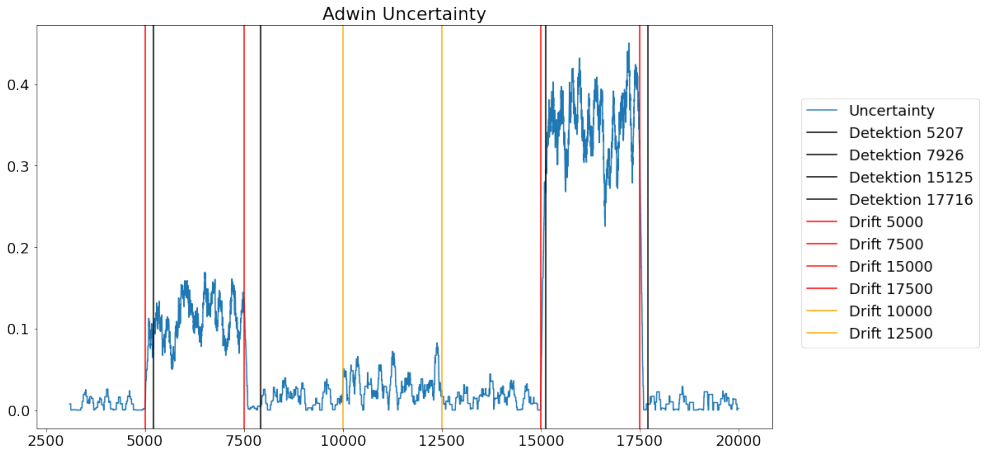

Abbildung 4.10: Model Uncertainty und Detektionen über Zeit bei *retrain-worst*

Auch in dieser Strategie hat der Detektor trotz der höheren Model Uncertainty nicht auf die virtuellen Drifts reagiert.

Wird die Accuracy nach jedem klassifizierten Datensatz über ein gleitendes Fenster der zuletzt gesehenen 200 Daten ermittelt, kann in den Bereichen, in denen ein Drift vorlag, eine Abnahme der Accuracy beobachtet werden (siehe Abbildung 4.11).

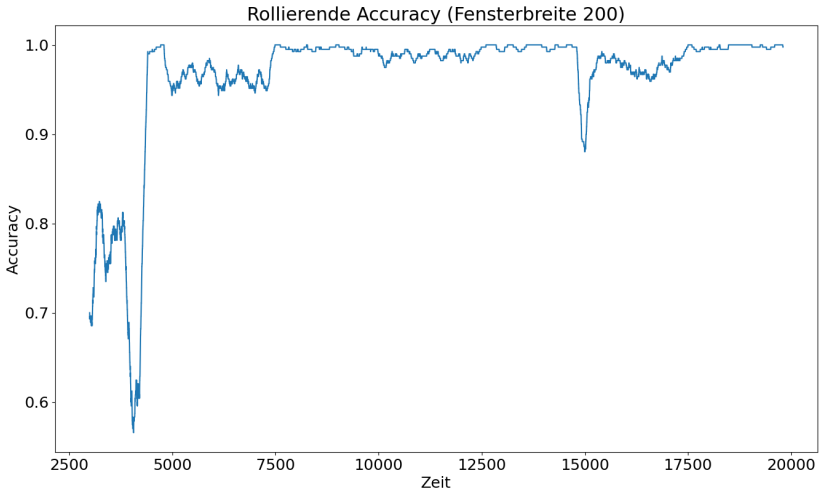

Abbildung 4.11: Rollierende Accuracy der Mixed Abrupt Daten bei *retrain-all*

An den Punkten, bei denen die Accuracy im Verhältnis besonders niedrig ist, wurde ein Drift detektiert und nachfolgend das Ensemble weiter angepasst. Nach der Anpassung ist dennoch weiterhin im Vergleich zu der Drift-freien Zeit immer noch eine geringere Accuracy zu beobachten. Erst nach dem Ende des Concept Drifts gleicht sich diese wieder den Werten vor dem Drift an. Auch hier ist anfangs wieder eine deutlich geringere Accuracy zu beobachten, die vermutlich aufgrund noch geringer Datenmengen für das initiale Training entstanden ist.
Auch für die Klassifikations-Daten wird die Anzahl von fälschlicherweise detektierten oder nicht detektierten Drifts (Fehler) sowie die mittlere Zeit zur Detektion tatsächlicher Drifts (Zeit) in nachfolgender Tabelle 4.6 je Datensatz dargestellt. In der Detektion der Drifts in den Mixed inkrementell Daten kam es als Folge auf die Entwicklung nach dem Eintreten des Drifts zu weiteren Detektionen in der *retrain-worst* Strategie. Diese sind daher in Klammern angegeben, da sie nicht als Fehler, sondern eher als Folge des andauernden Drifts zu interpretieren sind. Bei dem Datensatz Mixed Target (3) ist zu be-

achten, dass die fehlerhafte Detektion durch eine Detektion vor dem eigentlichen Drift entstanden ist.
Die Detektion eines inkrementellen Drifts auf den **Mixed inkrementell** Daten dauert deutlich länger als bei den abrupten Drifts. Reale, abrupte Drifts (**Mixed Target (3)**) werden im Gegensatz dazu zuverlässiger detekiert. Es kommt nicht zu zusätzlichen Detektionen. Eine Darstellung der Detektion auf den beiden anderen Datensätzen findet sich in Anhang A.5.1.2.

Tabelle 4.6: Ergebnisse der synthetischen Klassifikations-Datensätze M=5

Datensatz	Strategie	Fehler	Zeit
Mixed Abrupt	*retrain-all*	0	419.5
	retrain-worst	0	368.5
Mixed inkrementell	*retrain-all*	0	4047
	retrain-worst	(3)	3055
Mixed Target (3)	*retrain-all*	1	380
	retrain-worst	1	124

Die Strategie der *retrain-worst* Anpassung führt in dem direkten Vergleich zu schnelleren Detektionen der realen Drifts. Hier ist die Zeit bis zu der Detektion auf den Mixed Abrupt Daten im Vergleich zu der UDD mit 247.3 deutlich erhöht.
Bei inkrementellen Drifts kommt es zu einer erhöhten Anzahl an Fehlern.
Im Vergleich des MCC aller Strategien kann der Erfolg der *retrain-worst* Strategie mit dem ADWIN Detektor ähnlich wie schon in den Regressions-Daten nicht gehalten werden (siehe Tabelle 4.7).
Für den MCC gilt: je größer, desto besser die Modellanpassung. Auf zwei der drei Datensätze erreicht der ADWIN Detektor mit der *retrain-all* Strategie die höchsten Ergebnisse und auf den Daten mit inkrementellem Drift der PH Detektor mit vollständiger Anpassung des Ensembles nach einem Drift. Der Wertunterschied ist jedoch gering, weshalb der ADWIN Detektor unter *retrain-all* insgesamt im Vergleich zu den anderen Strategien die beste Performance zeigt.

Tabelle 4.7: MCC (Anzahl Re-Trainings) der synthetischen Klassifikations-Datensätze mit M=5

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Mixed Abrupt	**0.93**(**4**)	0.78(4)	0.77(0)	0.81(0)	0.84(4)	0.79(4)
Mixed inkre-mentell	0.93(1)	0.92(4)	**0.94**(**1**)	0.90(0)	0.92(1)	0.91(1)
Mixed Target (3)	**0.97**(**3**)	0.87(3)	0.87(0)	0.87(0)	0.93(3)	0.92(3)

Eine Übersicht der Ergebnisse verglichen anhand der AUC findet sich im Anhang unter A.3.
Wie schon bei den Regressions-Daten kann also auch hier die Strategie *retrain-all* die Anpassung des schlechtesten Modells unter *retrain-worst* übertreffen. Obwohl das Ensemble schneller reagiert, wenn nur ein Modell angepasst wird, ist dennoch die Vorhersagegüte mit *retrain-worst* insgesamt schlechter.
Weiterhin lassen die Ergebnisse dieser synthetischen Daten im Vergleich zu den Regressions-Daten vermuten, dass die EUDD besser für die Klassifikations-Daten geeignet ist. Neben dem so oft verwendeten ADWIN Detektor scheint sich auch ein PH Detektor je nach Anwendungsfall zu eignen. Dieser birgt allerdings das Risiko, zu viele Drifts (wie in der Regression) oder zu wenig Drifts (wie in diesem Beispiel) zu erkennen.
Um diese Aussagen zu stützen, wurden die verschiedenen Methoden nachfolgend auf Real-World Daten angewendet.

4.5.2 *Real-World Daten*

Aufgrund der höheren Komplexität der Real-World Datensätze im Vergleich zu den synthetischen Daten wurden verschiedene Konfigurationen des Ensembles getestet. Während es in Bezug auf die Höhe der relevanten Metriken bei den synthetischen Datensätzen keinen nennenswerten Unterschied ergab, konnten die Ergebnisse der Real-World Datensätze für die Klassifikation durch eine Verringerung der Anzahl der Hidden Layer auf insgesamt zwei deutlich profitieren.
Auch eine Erhöhung der Anzahl von Modellen des Ensembles wurde getestet. Damit konnte jedoch für die meisten Datensätze keine deutliche Verbesserung der Ergebnisse erzielt werden. Eine Erhöhung auf $M = 10$ Modelle ergab in wenigen Fällen eine deutliche Verbesserung. Da der Unterschied bei einer Erhöhung auf $M > 10$ noch weniger deutlich war, werden keine Ergebnistabellen für diese Konfiguration geliefert.
Im Folgenden werden nur die jeweils erfolgreichsten Strategien für Regressions- als auch Klassifikations-Daten gezeigt, weitere Ergebnis-Tabellen verschiedener Konfigurationen des Ensembles befinden sich im Anhang A.5.2.

4.5.2.1 *Regression*

Wie Vergleiche mit verschiedenen Konfigurationen (Anzahl Modelle, Anzahl Hidden Layer, Nutzung der Trainingsdaten für das Re-Training) des Ensembles ergeben haben, wurden die besten Ergebnisse über ein Ensemble von 10 Modellen mit jeweils vier Hidden Layern unter Anpassung der Modelle nur durch die zuletzt eingetroffenen Daten (refit_use_X_train=False), d.h. ohne die Trainings-Daten, erreicht (für einen Vergleich siehe Ergebnisse anderer Konfigurationen im Anhang A.5.2.1).

Für die Real-World Regressions-Daten zeigte der PH Detektor ein besseres Ergebnis der gesamten Modellanpassung bzw. Vorhersagen. Zu berücksichtigen ist allerdings für die Bike Sharing Daten, dass der Detektor deutlich mehr Drifts erkannt hat und demnach ein Performance-Verlust im Sinne einer verlängerten Laufzeit des Pro-

[9] Ensemble wurde mit M=10 statt fünf Modellen trainiert.

Tabelle 4.8: RMSE (Anzahl Re-Trainings) der Real-World Regressions-Datensätze

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Air Quality	13.92(30)	36.40(28)	**9.03**(**13**)	40.45(0)	15.62(30)	24.90(30)
Bike Sharing[9]	1.17(10)	0.85(13)	**0.49**(**46**)	0.93(0)	3.11(10)	1.34(10)

gramms aufgrund der hohen Anzahl von Re-Trainings eingetreten ist. Die Methode wurde ebenfalls auf die New York Taxi Daten angewendet. Aufgrund der Darstellbarkeit unter Berücksichtigung der ohnehin sehr hohen Werte des RMSE wurden die Ergebnisse der Strategien aufgerundet und enthalten demnach keine Nachkommastellen. Auch hier wurden die Modelle nur mit Hilfe der zuletzt durch den Stream eingetroffenen Daten angepasst (refit_use_X_train=False).

Tabelle 4.9: RMSE (Anzahl Re-Trainings) der New York Taxi Daten mit M=5

EUDD		PH	No-Retr.	Eq. D.	Rand.
retrain-all	retrain-worst				
39852(**10**)	58539(14)	74963(118)	42321(0)	39586(10)	81360(10)

Das beste Ergebnis wurde mit dem ADWIN Detektor unter der *retrain-all* Strategie erreicht. Der RMSE der Daten ist dennoch sehr hoch. Die vergleichsweise geringe Größe des Datensatzes mit nur 4383 Einträgen wird als Ursache für dieses Ergebnis vermutet. In dem Algorithmus wird immer ein bestimmter Prozentsatz für das initiale Training und die Neuanpassung verwendet. Aufgrund der geringen Datenmenge wird angenommen, dass der aktuelle Kontext unzureichend repräsentiert wird. Es kommt folglich zu einer Überanpassung an wenige Instanzen und damit einer schlechteren Vorhersagegüte über nachfolgende Daten. Zudem wurde dieser Datensatz für diese Arbeit zusammengestellt und enthält demnach

nur frei verfügbare Daten, deren Einsatz als Features möglicherweise nicht geeignet oder unzureichend ist. Daher sind die Ergebnisse für den New York Taxi Datensatz nicht überzubewerten, da dieser Datensatz nur einen möglichen Anwendungsfall simulieren soll und nicht für einen methodenübergreifenden Vergleich dienen kann.

Um einen abschließenden Vergleich zu der UDD aus [4] zu liefern, werden die besten Ergebnisse aus der EUDD unter Nutzung des ADWIN Detektors verglichen.

Tabelle 4.10: Vergleich von UDD und EUDD

Datensatz	**Algorithmus**	**RMSE**
Air Quality	*UDD*	**1.15**(**14**)
	EUDD	13.92(30)
Bike Sharing	*UDD*	129.93(5)
	EUDD	**0.85**(**13**)

Zu erkennen ist, dass für die Air Quality Daten mit der UDD ein besseres Ergebnis erzielt werden konnte. Trotz einer deutlich höheren Anzahl an Re-Trainings konnte unabhängig der Nutzung von verschiedenen Detektoren und Anpassungsstrategien kein RMSE nahe dem Ergebnis der UDD erreicht werden.
Im Gegensatz dazu konnte der RMSE der Bike Sharing Daten im Vergleich zu der UDD deutlich verbessert werden. Zu beachten ist hier ebenfalls eine höhere Anzahl an Re-Trainings. Würde an dieser Stelle das Ergebnis des PH Detektors gezeigt werden, wäre der Unterschied zu dem RMSE der UDD noch stärker.

Es ist also festzuhalten, dass keiner der beiden Algorithmen für Regressions-Daten deutlich besser funktioniert als der andere. Dieser Vergleich basiert jedoch nur auf zwei Datensätzen.

4.5.2.2 *Klassifikation*

Alle Ensembles der Real-World Klassifikations-Daten wurden mit $M = 5$ oder $M = 10$ Modellen trainiert, die jeweils zwei Hidden Layer aufwiesen. Weniger als die anfänglichen vier Hidden Layer zu verwenden führte im Gegensatz zu den Modellen auf den Regressions-Daten zu verbesserten Ergebnissen gemessen anhand von MCC und AUC (siehe Anhang A.5.3). Nur bei einem Datensatz wurden die anfänglichen Trainings-Daten erneut für eine Anpassung gemeinsam mit den zuletzt aktuellen Daten genutzt.
Bei fast allen Datensätzen führt die EUDD unter Nutzung eines ADWIN Detektors zu den höchsten Ergebnissen in Bezug auf den MCC. In zwei Fällen erreicht der PH Detektor und einmal die Anpassungsstrategie *retrain-worst* die gleiche Höhe in dem Gesamtergebnis. In den Gassensor Daten wird mit keinem der Detektoren ein Drift detektiert. Damit wird anhand dieser Daten kein Vergleich von den Detektoren oder den Anpassungsstrategien vorgenommen.

Tabelle 4.11: MCC (Anzahl Re-Trainings) der Real-World Klassifikations-Datensätze mit zwei Hidden Layern je Modell

Datensatz	M	EUDD		PH	No-Retr.	Eq. D.	Rand.
		retrain-all	retrain-worst				
Insects Abrupt[10]	5	**0.54**(**5**)	**0.54**(**2**)	**0.54**(**4**)	0.50(0)	0.52(5)	0.51(5)
Insects Inc	5	**0.48**(**3**)	0.29(4)	**0.48**(**4**)	0.11(0)	0.44(3)	0.26(3)
Insects IncAbr	10	**0.53**(**20**)	0.22(21)	0.30(0)	0.32(0)	0.46(18)	0.40(18)
Insects IncReo	10	**0.61**(**21**)	0.30(24)	0.47(15)	0.18(0)	0.53(21)	0.45(21)
KDDCUP99	5	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)
Gassensor	5	**0.66**(**44**)	0.53(53)	0.41(19)	0.36(0)	0.62(44)	0.65(44)
Electricity	10	**0.48**(**9**)	0.21(20)	0.24(2)	0.21(0)	0.36(9)	0.32(9)

[10] Ensemble wurde mit den zuletzt eingetroffenen Daten und den Trainingsdaten neu angepasst (refit_use_X_train=True).

Entgegen der anfänglichen Erwartung zeigte auf allen Datensätzen mit inkrementellen oder wiederkehrenden Drifts die *retrain-all* Strategie bessere Ergebnisse als die *retrain-worst* Strategie des ADWIN Detektors. Dies galt auch für die meisten Datensätze, wenn mehr Modelle als nur das schlechteste neu trainiert worden sind. Getestet wurde dies auch mit der Hälfte der Modelle, d.h. mit fünf von den zehn Komponenten des Ensembles (siehe Tabelle A.15 im Anhang für den Gesamtvergleich).

Tabelle 4.12: Vergleich von UDD und EUDD

Datensatz	**Algorithmus**	**MCC**
Insects Abrupt	*UDD*	0.52(9)
	EUDD	**0.54**(**5**)
Insects Inc	*UDD*	0.24(4)
	EUDD	**0.48**(**3**)
Insects IncAbr	*UDD*	0.52(22)
	EUDD	**0.53**(**20**)
Insects IncReo	*UDD*	0.21(10)
	EUDD	**0.61**(**21**)
KDDCUP99	*UDD*	0.96(20)
	EUDD	**0.99**(**0**)
Gassensor	*UDD*	0.48(39)
	EUDD	**0.66**(**44**)
Electricity	*UDD*	0.44(13)
	EUDD	**0.48**(**9**)

Abschließend werden die Ergebnisse der Real-World Klassifikations-Daten mit denen der UDD verglichen. Es ist zu erkennen, dass die EUDD die UDD in allen Datensätzen übertreffen konnte.

5

DISKUSSION

Aufbauend auf dem Konzept der UDD wurde ein Ensemble genutzt, um Schätzungen der Model Uncertainty zu erhalten und gleichzeitig mehr Flexibilität in den Strategien der Modellanpassung zu haben. Wie in zahlreichen Studien gezeigt worden ist, ergeben Ensembles ein zuverlässigeres und konservativeres Maß der Model Uncertainty.
Die Schätzung und Auswertung der Model Uncertainty über zwei verschiedene Detektoren (ADWIN und PH) in Streams von Datensätzen verschiedener Charakteristika konnte die Ergebnisse der UDD gemessen anhand von RMSE und MCC in allen bis auf einem der verwendeten Datensätze übertreffen.

Bei der Überprüfung der Funktionsweise des Algorithmus auf synthetischen Daten ist auffällig, dass der Detektor gerade bei Regressions-Daten auch auf virtuelle Drifts reagiert. Bei Klassifikations-Daten kam es zu weniger fehlerhaften Detektionen. Dennoch wurden die meisten realen Drifts zuverlässig, wenn auch zeitverzögert, detektiert. Ob die Detektion virtueller Drifts aber wirklich als fehlerhaft bewertet werden sollte ist strittig. Analog zu [58] müsse keine Unterscheidung vollzogen werden, da auch ein virtueller Drift eine Anpassung des Modells erfordere.
Im Vergleich zu dem ADWIN Detektor konnte der Algorithmus unter Verwendung eines PH Detektors in wenigen Fällen gleich gute oder bessere Ergebnisse erzeugen. Dennoch kommt es bei dem PH Detektor auch zu starken Unterschieden in der Anzahl der detektierten Drifts, sodass deutlich mehr Drifts detektiert werden und die Performance abnimmt, da sehr viele Re-Trainings stattfinden müssen. In wieder anderen Fällen hat der PH Detektor keine Concept Drifts detektiert. Aufgrund dieser ambivalenten Ergebnisse wird ein Einsatz dieses Detektors nur eingeschränkt empfohlen.
Es wurde gezeigt, dass die EUDD mit verschiedenen Arten von Drifts umgehen kann, wobei bei abrupten Drifts zuverlässigere Ergebnisse

erzielt worden sind. Die Wirksamkeit der EUDD konnte dementsprechend nachgewiesen werden.

Aufgrund der erhöhten Komplexität der Real-World Datensätze im Gegensatz zu den synthetischen Daten wurden verschiedene Konfigurationen des Ensembles und einzelner Modelle getestet. Für eine Anpassung aller Modelle bzw. des schlechtesten Modells des Ensembles wurden nur in einem von neun Datensätzen die anfänglichen Trainingsdaten verwendet, da dies in den Real-World Datensätzen zu besseren Ergebnissen führte. Obwohl erwartet worden ist, dass das Modell sich an den aktuellen Kontext überanpassen würde, konnte dies in keiner der Anpassungsstrategien eindeutig nachgewiesen werden.
Vor der Durchführung wurde zudem angenommen, dass das Ensemble bei Anpassung nur eines oder nur der Hälfte der Modelle gerade bei langsam auftretenden oder wiederkehrenden Drifts profitieren würde. Damit sollte noch über längere Zeit zurückliegendes Wissen erhalten bleiben, das bei einem vollständigen Re-Training des Ensembles verfallen würde. Gerade auf Real-World Datensätzen mit inkrementellen Drifts oder wiederkehrenden Kontexten konnte dies nicht gezeigt werden.
Im Vergleich der Anpassungsstrategien zeigt die *retrain-worst* Strategie insgesamt keine Verbesserung gegenüber der *retrain-all* Methode. Gemessen wurde die Vorhersagegüte eines jeden Modells anhand von Entropie bei Klassifikations-Daten und Varianz bei Regressions-Daten. Beide Anpassungsstrategien zeigen zudem in der Anwendung auf synthetischen Datensätzen eine relativ große Zeitspanne bis zu einer Detektion gemessen anhand der Indexwerte. Gerade in Situationen, in denen die Daten besonders langsam einströmen, hat dies eine lange Zwischenzeit zur Folge. Zukünftig sollte daran geforscht werden, wie die Zeit bis zu einer Detektion vor allem in der vielversprechenden *retrain-all* Strategie verkürzt werden kann, wohingegen auf Basis der Ergebnisse die *retrain-worst* Strategie nicht priorisiert untersucht werden sollte.

Aufbauend auf der Theorie wurde weiterhin erwartet, dass eine Erhöhung der Anzahl von Modellen des Ensembles und eine verstärkte

Heterogenität einzelner Modelle zu besseren Ergebnissen führen. Dies kann nur teilweise gehalten werden. Eine Tendenz zeigte sich bereits in der Optimierung der Neuronen in einzelnen Neuronalen Netzen. Die Modelle für Klassifikations-Daten zeigten in einigen Fällen bei geringerer Komplexität durch weniger Neuronen je Hidden Layer eine Verbesserung, was für Modelle auf Regressions-Daten nicht galt. In den Ergebnissen der Regressions-Daten konnte eine Erhöhung der Modelle im Ensemble die gewünschte Verbesserung erreichen. Jedoch führt die Verringerung der Hidden Layer, als Mittel um unterschiedlichere Modelle zu erhalten, zu dem gegenteiligen Effekt. Für die Klassifikations-Daten verhält es sich gegensätzlich. Hier kann eine Erhöhung auf 10 Modelle im Ensemble nur bei drei von sieben Datensätzen einen erhöhten MCC erzielen, wohingegen eine Verringerung der Hidden Layer auf insgesamt zwei bei allen Datensätzen erfolgreich ist.

In dem Vergleich mit Baseline-Strategien, wie keiner, zufälliger oder regelmäßiger Modellanpassung ist anzumerken, dass diese oftmals ähnliche Werte zu den Ergebnissen der EUDD zeigten. Diese Ergebnisse sind jedoch stark von dem Zufall abhängig. Hier ist entscheidend, ob die Zeitpunkte zufällig passend für ein Re-Training sind. Zu beobachten sind durch ein zufälliges oder regelmäßiges Re-Training stark schwankende Ergebnisse, wohingegen durch ein Triggered Rebuild nach einem Concept Drift sehr ähnliche Ergebnisse nach mehreren Durchläufen vorliegen. Demnach kann gezeigt werden, dass die Güte des Modells davon abhängt, dass das Modell zu dem richtigen Zeitpunkt angepasst wird.

Ziel der Arbeit war ein Vergleich mit der UDD. Dieser Algorithmus konnte nur in einem Fall ein deutlich besseres Ergebnis gemessen anhand des RMSE erzielen. Diesem Wert kann sich mit keiner der getesteten Konfigurationen des Ensembles angemessen angenähert werden. Die EUDD kann die UDD bei allen anderen der neun Datensätze übertreffen. Hier muss jedoch berücksichtigt werden, dass Ensembles naturgemäß bessere Ergebnisse als ein einzelnes Modell erzeugen, da hier mehrere Weak Learner kombiniert werden.

Es wird zudem vermutet, dass Ensembles in bestimmten Situationen - insbesondere in dieser Arbeit erkennbar anhand der synthetischen Datensätze - eine sehr sensible Reaktion auf Änderungen in der Höhe der Uncertainty zeigen. Wie bereits belegt treffen Uncertainty-Analysen unter Nutzung eines Neuronalen Netzes mit Monte-Carlo Dropouts überoptimistische Schätzungen eben jener Model Uncertainty. Vielleicht kann damit auch die Eigenschaft des Ensembles als konservativer Schätzer als Begründung für fehlerhafte Detektionen oder einen geringeren RMSE der Air Quality Daten dienen. Weiterhin ist ein Einfluss aus der verhältnismäßig geringen Größe des Datensatzes zu erwarten.

Da vielversprechende Ergebnisse auf Basis eines Ensembles erreicht worden sind, sollte sich in zukünftiger Forschung bei unüberwachten Ansätzen in der Detektion von Concept Drifts an die Uncertainty-Analyse durch eine Strategie wie die EUDD, basierend auf einem Ensemble, gehalten werden. Weiterhin sollten verschiedene Modelltypen in dem Ensemble verwendet werden, wie z.B. Long Short-Term Memory Netze oder gänzlich andere Modelltypen, die eine Schätzung der Varianz in Regressions-Problemen zulassen. Des Weiteren könnte den Modellen je nach Performance ein unterschiedliches Gewicht zugewiesen werden.
Um noch bessere Schätzungen der Uncertainty zu erreichen, könnte ebenfalls die Data Uncertainty gemessen werden. So ließen sich fehlerhafte Detektionen, ausgelöst durch ein erhöhtes Rauschen in den Daten, vermeiden. Dabei spielt also nur die *heteroscedatic Uncertainty* eine Rolle, aus der *homoscedatic Uncertainty* lassen sich keine Hinweise für eine Veränderung der Daten oder des Kontexts ableiten. Die Model und Data Uncertainty könnten nachfolgend wie bei [26] als geometrischer Mittelwert durch den Detektor ausgewertet werden, wenn ein Concept Drift nur bei hohen Werten beider Uncertainty-Arten detektiert werden soll.
Eine weitere Möglichkeit für die Konfiguration des Ensembles ergibt sich aus der Optimierung der *Decision Optimization*. Nachdem gezeigt worden ist, wie das Ensemble für verschiedene Datensätze konfiguriert werden sollte, könnte im Folgenden nach einer optimalen Entscheidungsfunktion gesucht werden.

Teil II

APPENDIX

A

ANHANG

A.1 STUDIEN ZU METHODEN FÜR DIE QUANTIFIZIERUNG VON UNCERTAINTY

Tabelle A.1: Gesamtübersicht empirischer Experimente für den Vergleich

Studie	**Jahr**	**Data Uncertainty**	**Model Uncertainty**
Antorán et al. [2]	2020		✓
Cobb et al. [10]	2019		✓
Gal und Ghahramani [17]	2016		✓
Gast et al. [24]	2018	✓	
Hall et al. [26]	2020		✓
Henne et al. [28]	2020		✓
Kendall et al. [32]	2017	✓	✓
Lakshminarayanan et al. [35]	2017		✓
Multaheb et al. [44]	2021		✓
Pearce et al. [50]	2020		✓
Segu et al. [53]	2019		✓
TV et al. [60]	2021		✓
Vergara et al. [61]	2019		✓
Wang et al. [63]	2019	✓	✓

A.2 VERWENDETE PAKETE

```
absl-py==0.13.0
altair==4.1.0
appnope==0.1.2
argcomplete==1.12.3
argon2-cffi==21.1.0
astor==0.8.1
astunparse==1.6.3
attrs==21.2.0
backcall==0.2.0
backports.zoneinfo==0.2.1
base58==2.1.1
bleach==4.1.0
blinker==1.4
cached-property==1.5.2
cachetools==4.2.2
certifi==2021.5.30
cffi==1.15.0
charset-normalizer==2.0.5
clang==5.0
click==7.1.2
cycler==0.10.0
debugpy==1.5.1
decorator==5.1.0
defusedxml==0.7.1
entrypoints==0.3
flatbuffers==1.12
gast==0.4.0
gitdb==4.0.9
GitPython==3.1.24
google-auth==1.35.0
google-auth-oauthlib==0.4.6
google-pasta==0.2.0
grpcio==1.40.0
h5py==3.1.0
idna==3.2
```

```
importlib-metadata==4.8.1
ipykernel==6.4.2
ipython==7.29.0
ipython-genutils==0.2.0
ipywidgets==7.6.5
jedi==0.18.0
Jinja2==3.0.2
joblib==1.0.1
jsonschema==4.1.2
jupyter-client==7.0.6
jupyter-core==4.9.1
jupyterlab-pygments==0.1.2
jupyterlab-widgets==1.0.2
keras==2.6.0
Keras-Preprocessing==1.1.2
kiwisolver==1.3.2
Markdown==3.3.4
MarkupSafe==2.0.1
matplotlib==3.4.3
matplotlib-inline==0.1.3
mistune==0.8.4
nbclient==0.5.4
nbconvert==6.2.0
nbformat==5.1.3
nest-asyncio==1.5.1
notebook==6.4.5
numpy==1.19.5
oauthlib==3.1.1
opt-einsum==3.3.0
packaging==21.2
pandas==1.3.3
pandocfilters==1.5.0
parso==0.8.2
pexpect==4.8.0
pickleshare==0.7.5
Pillow==8.3.2
prometheus-client==0.12.0
```

```
prompt-toolkit==3.0.21
protobuf==3.18.0
ptyprocess==0.7.0
pyarrow==6.0.0
pyasn1==0.4.8
pyasn1-modules==0.2.8
pycparser==2.20
pydeck==0.7.1
Pygments==2.10.0
pyparsing==2.4.7
pyrsistent==0.18.0
python-dateutil==2.8.2
pytz==2021.1
pytz-deprecation-shim==0.1.0.post0
PyYAML==5.4.1
pyzmq==22.3.0
requests==2.26.0
requests-oauthlib==1.3.0
rsa==4.7.2
scikit-learn==0.24.2
scikit-multiflow==0.5.3
scipy==1.7.1
Send2Trash==1.8.0
six==1.15.0
smmap==5.0.0
sortedcontainers==2.4.0
streamlit==1.1.0
tensorboard==2.6.0
tensorboard-data-server==0.6.1
tensorboard-plugin-wit==1.8.0
tensorflow==2.6.0
tensorflow-estimator==2.6.0
termcolor==1.1.0
terminado==0.12.1
testpath==0.5.0
threadpoolctl==2.2.0
toml==0.10.2
```

```
toolz==0.11.1
tornado==6.1
traitlets==5.1.1
typing-extensions==3.7.4.3
tzdata==2021.5
tzlocal==4.1
urllib3==1.26.6
validators==0.18.2
wcwidth==0.2.5
webencodings==0.5.1
Werkzeug==2.0.1
widgetsnbextension==3.5.2
wrapt==1.12.1
zipp==3.5.0
```

A.3 DATENSÄTZE

A.3.1 *Synthetische Daten*

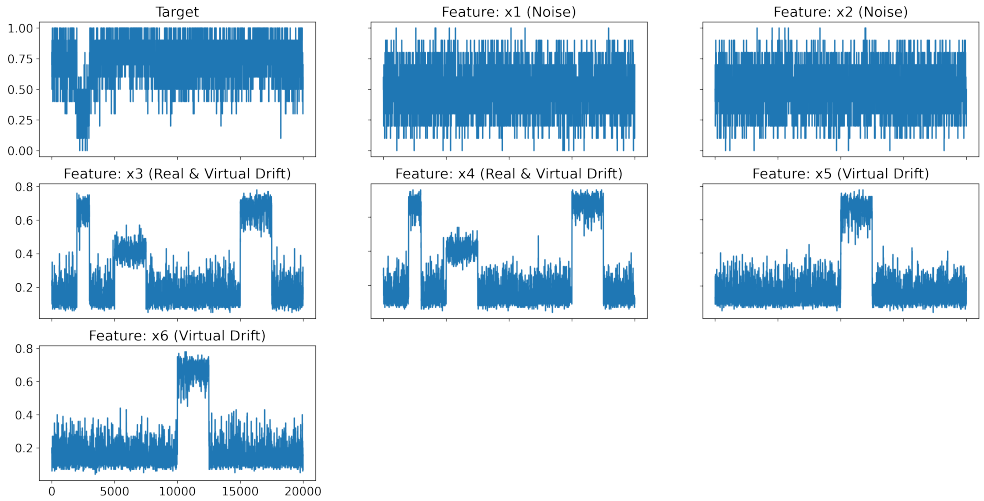

Abbildung A.1: Verteilung der Zufallsvariablen im Mixed Abrupt Datensatz

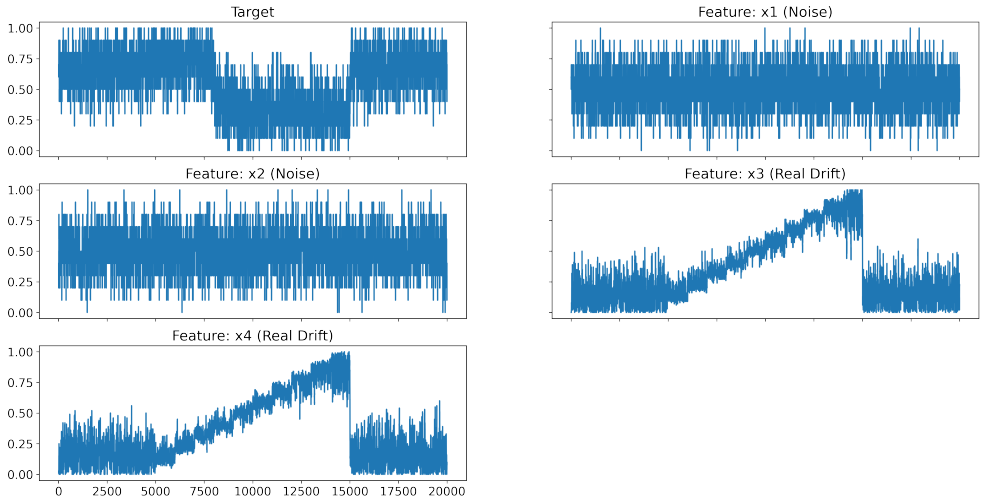

Abbildung A.2: Verteilung der Zufallsvariablen im Mixed inkrementell Datensatz

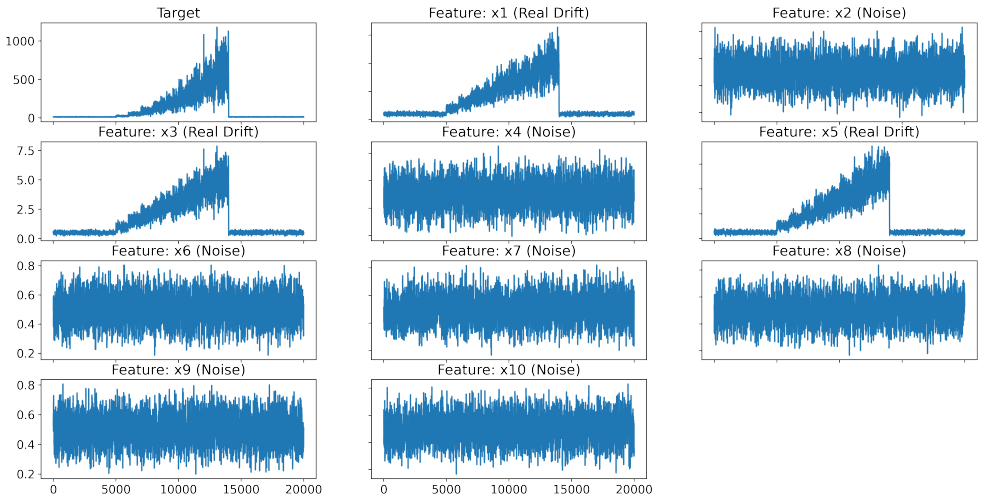

Abbildung A.3: Verteilung der Zufallsvariablen im Friedman inkrementell Datensatz

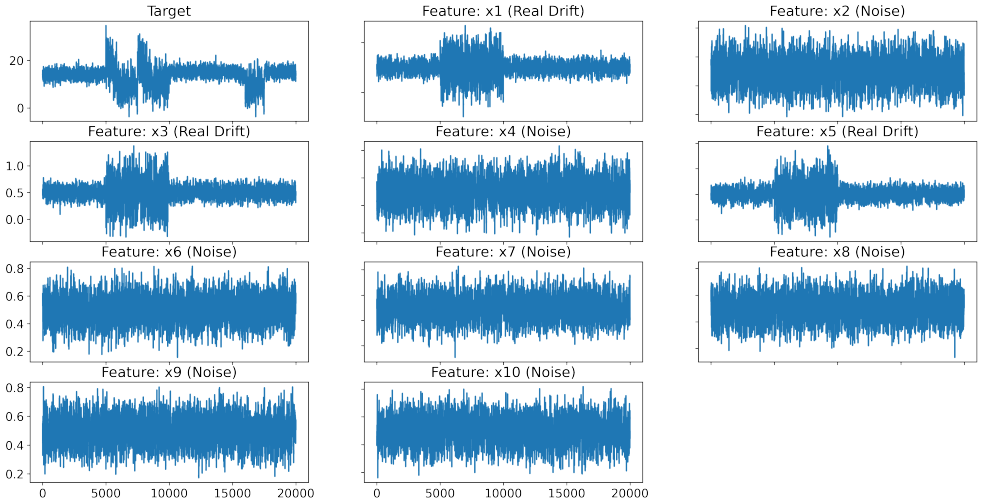

Abbildung A.4: Verteilung der Zufallsvariablen im Friedman gradual Datensatz

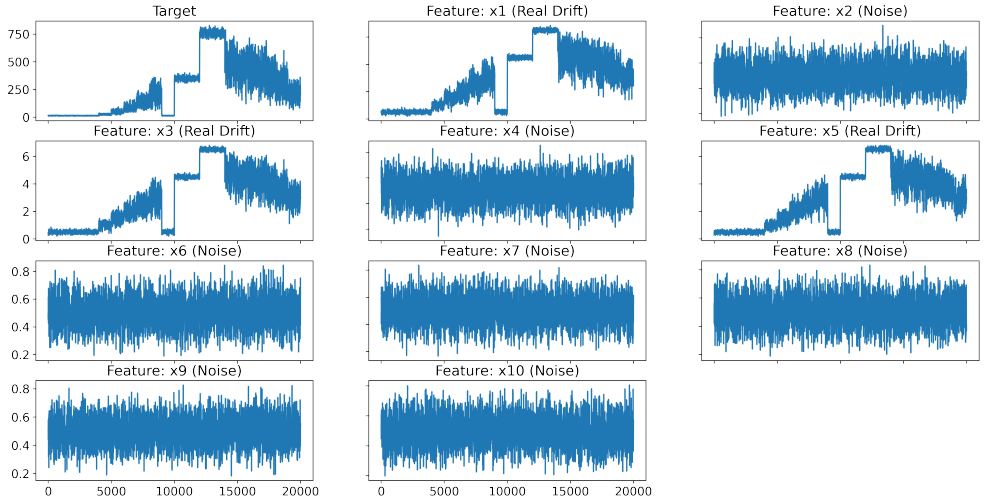

Abbildung A.5: Verteilung der Zufallsvariablen im Friedman No Return Datensatz

A.3.2 *Real-World Daten*

Tabelle A.2: Übersicht der Real-World Datensätze

Datensatz	Art	Anzahl Features	Anzahl Outputs	Größe
Air Quality [14]	Regression	16	1	9358
Bike Sharing [12]	Regression	14	1	17389
New York Taxi	Regression	15	1	4383
Insects Abrupt [54]	Klassifikation	33	6	52848
Insects Inc [54]	Klassifikation	33	6	452044
Insects IncAbr [54]	Klassifikation	33	6	79986
Insects IncReo [54]	Klassifikation	33	6	79986
KDDCUP99 [54]	Klassifikation	44	23	494021
Gassensor [54]	Klassifikation	6	128	13910
Electricity [54]	Klassifikation	2	8	45312

A.4 WESENTLICHER CODE AUS DEM REPOSITORY

A.4.1 *Datei Performance_Evaluation.py*

Listing A.1: Performance_Evaluation.py

```
from warnings import simplefilter

# ignore all future warnings
simplefilter(action='ignore', category=FutureWarning)
simplefilter(action='ignore', category=RuntimeWarning)
import warnings

warnings.filterwarnings("ignore", category=DeprecationWarning)

from tensorflow.python.framework.ops import
    disable_eager_execution
disable_eager_execution()

import pickle
from skmultiflow.data import DataStream

from classifier_models import CLF_Ensemble, classifier_model
from regression_models import Reg_Ensemble, regression_model
# from simple_regression_models import Reg_Ensemble,
    regression_model
from Detection_Strategies import Adwin_Uncertainty,
    No_Retraining, Equal_Distribution, Uninformed, \
     PageHinkley_Uncertainty
from utils import run_performance_experiment,
    plot_uncertainty_with_rolling_mean, plot_metrics_per_window
from dataloader import *
from dimensionality_reduction import
    pca_dimensionality_reduction

algorithm = 'DNN_Ensemble'  # 'PCA_prep_DNN_Ensemble'
data, dataset_name, features, stream_length = data_loader_mixed
    ('Datasets/Synthetic/mixed_1_abrupt_2_virtual_drift.csv')
print(dataset_name)
print('Dataset Shape: ', data.shape)
print(algorithm)

#
    ----------------------------------------------------------------------


#
```

```
# Make your adjustments here
#
#
    ------------------------------------------------------------------------


# comment out if data should not be reduced in its complexity by
    pca
# data, features = pca_dimensionality_reduction(data)

prediction_type = 'classification'

# number of nodes per hidden layer
num_nodes = 64
num_members = 5

refit_use_X_train = True
print('Use Training Data for Retraining: ', refit_use_X_train)

# configure model and ensemble
if prediction_type == 'regression':
    num_hidden_layer = 4
    # determine number of nodes per hidden layer
    sizes = [num_nodes] * num_hidden_layer

    targets = 1
    model = regression_model(features, sizes=sizes,
        num_hidden_layer=num_hidden_layer)
    model_init = Reg_Ensemble(model, num_members=num_members,
        epochs=500, debug=False, scale_data=False)

else:
    num_hidden_layer = 2
    # determine number of nodes per hidden layer
    sizes = [num_nodes] * num_hidden_layer

    targets = data.iloc[:, -1].nunique()
    print('{} classes'.format(targets))
    model = classifier_model(features, targets, sizes,
        num_hidden_layer=num_hidden_layer)
    model_init = CLF_Ensemble(model, num_members=num_members,
        targets=targets, epochs=500, debug=False, scale_data=
        False)

# stream dataframe and determine first 5 percent of instances
stream = DataStream(data)
five_percent = int(stream.n_remaining_samples() * 0.05)
```

```
# predefine result dictionaries
metrics_list = []
errors_dict = {}
drifts_dict = {}
raw_dict = {}
ra = 0

#
    ------------------------------------------------------------------------

#
# ADWIN Uncertainty & determine the number of retrainings for
    benchmarks
#
#
    ------------------------------------------------------------------------


adwin_uncertainty = Adwin_Uncertainty()
adwin_uncertainty.prediction_type = prediction_type

adwin_uncertainty.gridsearch_adwin_parameter(stream, model_init,
     1, targets, features, starting_value=-1)
# adwin_param = 0.002 #1e-4
# adwin_uncertainty.set_parameter(adwin_param)

metrics, detected_drifts, raw_results =
    run_performance_experiment(stream, dataset_name, model,
    adwin_uncertainty, prediction_type, targets, retrain=True,
    retrain_after=ra, retrain_use_Xtrain=refit_use_X_train,
    retrain_all=True, features=features, num_members=num_members)

key = adwin_uncertainty.name + '_retrain_all'
errors_dict[key] = raw_results['errors']
drifts_dict[key] = detected_drifts
raw_dict[key] = raw_results
metrics['Dataset'] = dataset_name
metrics['Alg_run'] = key
metrics['Detection'] = adwin_uncertainty.name
metrics_list.append(metrics)

df_metrics = pd.DataFrame(metrics_list)
results_dict = {'metrics': df_metrics, 'errors': errors_dict, '
    drifts': drifts_dict, 'raw': raw_dict}

# create directory for results if not exists
if not os.path.exists('results/{}'.format(dataset_name)):
    os.makedirs('results/{}'.format(dataset_name))
```

```
# Save data
filename = 'results/{}/results_'.format(dataset_name) +
    algorithm + '_' + dataset_name + '.pickle'
pickle.dump(results_dict, open(filename, 'wb'), protocol=4)

# create directory for results if not exists
if not os.path.exists('results/{}/img'.format(dataset_name)):
    os.makedirs('results/{}/img'.format(dataset_name))

# plot rolling mean of uncertainties and drift points
variance = results_dict['raw'][key]['variances']
uncertainties = results_dict['raw'][key]['uncertainties'].tolist
    ()
plot_uncertainty_with_rolling_mean(detected_drifts,
    uncertainties, variance, algorithm, dataset_name,
    five_percent)

# plot rmse or accuracy per each 20 datasets
plot_metrics_per_window(results_dict, data, prediction_type,
    dataset_name, five_percent)

number_retrainings = len(detected_drifts)

#
    ------------------------------------------------------------------------

#
# ADWIN Uncertainty with single model retraining
#
#
    ------------------------------------------------------------------------


metrics, detected_drifts, raw_results =
    run_performance_experiment(stream, dataset_name, model,
    adwin_uncertainty, prediction_type, targets, retrain=True,
    retrain_after=ra, retrain_use_Xtrain=refit_use_X_train,
    retrain_all=False, features=features, num_members=num_members
    )

key = adwin_uncertainty.name + '_retrain_worst'
errors_dict[key] = raw_results['errors']
drifts_dict[key] = detected_drifts
raw_dict[key] = raw_results
metrics['Dataset'] = dataset_name
metrics['Alg_run'] = key
metrics['Detection'] = adwin_uncertainty.name
```

```
metrics_list.append(metrics)

df_metrics = pd.DataFrame(metrics_list)
results_dict = {'metrics': df_metrics, 'errors': errors_dict, '
    drifts': drifts_dict, 'raw': raw_dict}

# Save data
filename = 'results/{}/results_'.format(dataset_name) +
    algorithm + '_' + dataset_name + '.pickle'
pickle.dump(results_dict, open(filename, 'wb'), protocol=4)

#
    ------------------------------------------------------------------------

#
# No-Retraining, Equal Distribution, Uninformed and ADWIN
    Uncertainty
#
#
    ------------------------------------------------------------------------


page_hinkley_uncertainty = PageHinkley_Uncertainty()
page_hinkley_uncertainty.prediction_type = prediction_type
page_hinkley_uncertainty.gridsearch_ph_parameter(stream,
    model_init, 1, targets, features, starting_value=-1)
# page_hinkley_param = 0.002 #1e-4
# page_hinkley_uncertainty.set_parameter(page_hinkley_param)

# set parameters for benchmark strategies
print('Number of Retrainings for benchmark strategies: ',
    number_retrainings)
no_retraining = No_Retraining()
equal_distribution = Equal_Distribution(number_retrainings)
uninformed = Uninformed(number_retrainings)

detection = {'page_hinkley': page_hinkley_uncertainty,
             'no_retraining': no_retraining,
             'equal_distribution': equal_distribution,
             'uninformed': uninformed}

for cd_strategy in detection.keys():
    strategy = detection[cd_strategy]
    print(cd_strategy)

    metrics, detected_drifts, raw_results =
        run_performance_experiment(stream, dataset_name, model,
        strategy, prediction_type, targets, retrain=True,
```

```
        retrain_after=ra, retrain_use_Xtrain=refit_use_X_train,
        retrain_all=True, features=features, num_members=
        num_members)

    key = cd_strategy
    errors_dict[key] = raw_results['errors']
    drifts_dict[key] = detected_drifts
    raw_dict[key] = raw_results
    metrics['Dataset'] = dataset_name
    metrics['Alg_run'] = key
    metrics['Detection'] = cd_strategy
    metrics_list.append(metrics)

    df_metrics = pd.DataFrame(metrics_list)
    results_dict = {'metrics': df_metrics, 'errors': errors_dict
        , 'drifts': drifts_dict, 'raw': raw_dict}

    # Save data
    filename = 'results/{}/results_'.format(dataset_name) +
        algorithm + '_' + dataset_name + '.pickle'
    pickle.dump(results_dict, open(filename, 'wb'), protocol=4)
```

A.4.2 *Datei classifier_models.py*

Listing A.2: classifier_models.py

```
import numpy as np
from scipy.stats import entropy
from sklearn.utils import resample
from sklearn.preprocessing import StandardScaler

from keras.layers import Input, Dense
from keras.models import Model
from tensorflow.keras.utils import to_categorical

from numpy.random import seed

seed(1)
import tensorflow as tf
tf.random.set_seed(2)


def classifier_model(features, targets, sizes, num_hidden_layer)
    :
    """
    Declare model structure of each model with two hidden layers
        containing 50 nodes
    """

    def create_model():
        inputs = Input(shape=(features,))
        x = Dense(sizes[0], activation='relu',
            kernel_initializer='random_normal', bias_initializer
            ='zeros')(
            inputs)  # , kernel_initializer='random_normal',
                bias_initializer='zeros'

        # build variable number of hidden layers
        for i in range(1, num_hidden_layer):
            x = Dense(sizes[i], kernel_initializer='
                random_normal', bias_initializer='zeros',
                activation="relu")(x)

        outputs = Dense(targets, activation='softmax')(x)

        model_clf = Model(inputs, outputs)

        if targets > 2:
```

```
            model_clf.compile(optimizer='adam', loss='
                categorical_crossentropy')
        else:
            model_clf.compile(optimizer='adam', loss='
                binary_crossentropy')

        return model_clf

    return create_model


class CLF_Ensemble():
    def __init__(self, create_model, targets=2, epochs=500,
        num_members=5, scale_data=False, debug=False):

        self.num_members = num_members
        self.ensemble = [None] * num_members

        for member in range(num_members):
            self.ensemble[member] = create_model()

        self.targets = targets

        self.scaler = StandardScaler()
        self.scale_data = scale_data
        self.epochs = epochs

        if debug:
            self.epochs = 3
            print('Epochs: ', 3)

    def fit(self, X_train, y_train):

        if self.scale_data:
            X_train = self.scaler.fit_transform(X_train)

        y_train_cat = to_categorical(y_train, num_classes=self.
            targets)

        for i in range(self.num_members):
            X_train_resampled, y_train_cat_resampled = resample(
                X_train, y_train_cat, replace=True, random_state
                =0)
            self.ensemble[i].fit(x=X_train_resampled, y=
                y_train_cat_resampled, epochs=self.epochs,
                verbose=0)

        print('>Trained the whole ensemble.')
```

```
    def retrain_all_members(self, X_retrain, y_retrain):
        """
79      Function to retrain the whole ensemble
        """
        if self.scale_data:
            X_retrain = self.scaler.fit_transform(X_retrain)

84      y_retrain_cat = to_categorical(y_retrain, num_classes=
            self.targets)

        for i in range(self.num_members):

            if self.targets > 2:
89              self.ensemble[i].compile(optimizer='adam', loss
                    ='categorical_crossentropy')
            else:
                self.ensemble[i].compile(optimizer='adam', loss
                    ='binary_crossentropy')

            X_retrain_resampled, y_retrain_cat_resampled =
                resample(X_retrain, y_retrain_cat, random_state
                =0, replace=True)
94          # train_on_batch(x=X_retrain, y=y_retrain_cat) # to
                retrain the model while using the previous
                parameters
            self.ensemble[i].fit(x=X_retrain_resampled, y=
                y_retrain_cat_resampled, epochs=self.epochs,
                verbose=0)

        print('>Retrained the whole ensemble.')

99  def retrain_worst_model(self, X_retrain, y_retrain, X_test):
        """
        Based on the entropy the model gets or the models with
            the worst performance get retrained. Entropy should
            be small.
        """

104     if self.scale_data:
            X_retrain = self.scaler.fit_transform(X_retrain)

        y_retrain_cat = to_categorical(y_retrain, num_classes=
            self.targets)

109     # determine number of models to retrain and choose
            models performing the worst
        num_models_to_retrain = 1  # int(self.num_members / 2)
```

```
        entropy_measures = np.hstack(self.
            predict_for_each_member(X_test))
        worst_ensemble_members = (-entropy_measures).argsort()[:
            num_models_to_retrain]

        for i in range(num_models_to_retrain):
            model_to_update = worst_ensemble_members[i]
            X_retrain_resampled, y_retrain_cat_resampled =
                resample(X_retrain, y_retrain_cat, random_state
                =0, replace=True)

            # compile model depending on the targets scaling
            if self.targets > 2:
                self.ensemble[model_to_update].compile(optimizer
                    ='adam', loss='categorical_crossentropy')
            else:
                self.ensemble[model_to_update].compile(optimizer
                    ='adam', loss='binary_crossentropy')

            # train_on_batch(x=X_retrain, y=y_retrain) # to
                retrain the model while using the previous
                parameters
            self.ensemble[model_to_update].fit(
                X_retrain_resampled, y_retrain_cat_resampled,
                epochs=self.epochs, verbose=0)

        # print information on retrained models
        print('>Retrained models: ', worst_ensemble_members)

    def predict(self, X_test):
        """
        Function to predict with all members of the ensemble
        """
        if self.scale_data:
            X_test = self.scaler.fit_transform(X_test)

        predictions = []
        proba_predictions = []

        for j in range(self.num_members):
            y_pred = self.ensemble[j].predict(X_test)
            predictions.append(y_pred)
            y_pred = self.ensemble[j].predict_on_batch(X_test)
            proba_predictions.append(y_pred)

        y_pred_mean = np.mean(predictions, axis=0)
        y_pred = np.argmax(y_pred_mean, axis=1)
        y_pred_proba = np.mean(proba_predictions, axis=0)
```

```
149         entro = entropy(y_pred_mean, axis=1, base=2)

            return y_pred, entro, y_pred_proba

        def predict_for_each_member(self, X_test):
154         """
            Compute each models entropy for Test Dataset
            """
            if self.scale_data:
                X_test = self.scaler.fit_transform(X_test)
159
            predictions = []
            mean_entropy_per_model = []

            for j in range(self.num_members):
164             y_pred = self.ensemble[j].predict(X_test)
                predictions.append(y_pred)

            entropy_per_model = entropy(predictions, axis=1, base=2)

169         for i in range(len(entropy_per_model)):
                mean_entropy_per_model.append(np.mean(
                    entropy_per_model[i], axis=0))

            return mean_entropy_per_model
```

A.4.3 *Datei regression_models.py*

Listing A.3: regression_models.py

```
import numpy as np

import keras.backend as K
from keras.layers import Input, Dense
from keras.models import Model

from numpy.random import seed

seed(1)
import tensorflow as tf

tf.random.set_seed(2)
from sklearn.preprocessing import StandardScaler
from sklearn.utils import resample

'''
Verlustfunktion und Regressions-Modell: https://github.com/
    mvaldenegro/keras-uncertainty/blob/master/keras_uncertainty/
    models/DeepEnsembleRegressor.py
'''


def deep_ensemble_regression_nll_loss(sigma_sq, epsilon=1e-6):
    """
        Regression loss for a Deep Ensemble, using the negative
            log-likelihood loss.
        This function returns a keras regression loss, given a
            symbolic tensor for the sigma square output of the
            model.
        The training model should return the mean, while the
            testing/prediction model should return the mean and
            variance.
    """

    def nll_loss(y_true, y_pred):
        return 0.5 * K.mean(K.log(sigma_sq + epsilon) + K.square
            (y_true - y_pred) / (sigma_sq + epsilon))

    return nll_loss


def regression_model(features, sizes, num_hidden_layer):
    def create_model():
```

```
        inp = Input(shape=(features,))
        x = Dense(sizes[0], kernel_initializer='random_normal',
            bias_initializer='zeros', activation="relu")(inp)

        # build variable number of hidden layers
        for i in range(1, num_hidden_layer):
            x = Dense(sizes[i], kernel_initializer='
                random_normal', bias_initializer='zeros',
                activation="relu")(x)

        mean = Dense(1, activation="linear")(x)
        var = Dense(1, activation="softplus")(x)

        train_model = Model(inp, mean)
        pred_model = Model(inp, [mean, var])

        train_model.compile(loss=
            deep_ensemble_regression_nll_loss(var), optimizer=tf.
            keras.optimizers.Adam(0.001))

        return train_model, pred_model

    return create_model


class Reg_Ensemble():
    """
        Implementation of a Deep Ensemble for regression.
        Uses two models, one for training and another for
            inference/testing. The user has to provide a model
            function that returns
        the train and test models, and use the provided
            deep_ensemble_nll_loss for training.
    """

    def __init__(self, create_model, num_members=5, epochs=500,
        debug=False,
                 scale_data=False):

        self.num_members = num_members
        self.train_estimators = [None] * num_members
        self.test_estimators = [None] * num_members
        self.epochs = epochs
        self.scale_data = scale_data

        for member in range(num_members):
            train_model, test_model = create_model()
            self.train_estimators[member] = train_model
```

```
            self.test_estimators[member] = test_model

        self.scaler = StandardScaler()

        if debug:
            self.epochs = 3

    def fit(self, X_train, y_train, batch_size=32):
        """
            Fits the Deep Ensemble, each estimator is fit
                independently on the same data.
        """

        if self.scale_data:
            X_train = self.scaler.fit_transform(X_train)

        for i in range(self.num_members):
            X_train_resampled, y_train_resampled = resample(
                X_train, y_train, random_state=0, replace=True)
            self.train_estimators[i].fit(X_train_resampled,
                y_train_resampled, epochs=self.epochs, batch_size
                =batch_size, verbose=0)

        print('>Trained the whole ensemble.')

    def retrain_all_members(self, X_retrain, y_retrain):
        """
        Function to retrain the whole ensemble
        """
        if self.scale_data:
            X_retrain = self.scaler.transform(X_retrain)

        for i in range(self.num_members):
            X_retrain_resampled, y_retrain_resampled = resample(
                X_retrain, y_retrain, random_state=0, replace=
                True)
            # train_on_batch(x=X_retrain, y=y_retrain) # to
                retrain the model while using the previous
                parameters
            self.train_estimators[i].fit(X_retrain_resampled,
                y_retrain_resampled, epochs=self.epochs, verbose
                =0)

        print('>Retrained the whole ensemble.')

    def retrain_worst_model(self, X_retrain, y_retrain, X_test):
        """
```

```
        Based on the variance of each model the one with the
            worst performance gets retrained. Variance should be
            small.
        """

        if self.scale_data:
            X_retrain = self.scaler.fit_transform(X_retrain)

        # determine number of models to retrain and choose
            models performing the worst
        num_models_to_retrain = 1  # int(self.num_members / 2)
        variances = np.hstack(self.predict_for_each_member(
            X_test))
        worst_ensemble_members = (-variances).argsort()[:
            num_models_to_retrain]

        for i in range(num_models_to_retrain):
            model_to_update = worst_ensemble_members[i]
            X_retrain_resampled, y_retrain_resampled = resample(
                X_retrain, y_retrain, random_state=0, replace=
                True)
            # train_on_batch(x=X_retrain, y=y_retrain) # to
                retrain the model while using the previous
                parameters
            self.train_estimators[model_to_update].fit(
                X_retrain_resampled, y_retrain_resampled, epochs=
                self.epochs, verbose=0)

        # print information on retrained models
        print('>Retrained models: ', worst_ensemble_members)

    def predict(self, X_test):
        """
            Makes a prediction. Predictions from each estimator
                are used to build a gaussian mixture and its mean
                 and standard deviation returned.
        """

        if self.scale_data == True:
            X_test = self.scaler.transform(X_test)

        means = []
        variances = []

        for i in range(len(self.test_estimators)):
            results = self.test_estimators[i].predict(X_test)
            mean = results[0]
            var = results[1]
```

```
            means.append(mean)
            variances.append(var)

        means = np.array(means)
        variances = np.array(variances)

        mixture_mean = np.mean(np.array(means), axis=0)
        mixture_var = np.mean(variances + np.square(means), axis
            =0) - np.square(mixture_mean)
        mixture_var[mixture_var < 0.0] = 0.0
        mixture_var = np.hstack(mixture_var)

        return mixture_mean, np.sqrt(mixture_var)

    def predict_for_each_member(self, X_test):
        """
        Compute each models variance for Test Dataset
        """

        if self.scale_data:
            X_test = self.scaler.transform(X_test)

        means = []
        variances = []

        for i in range(self.num_members):
            results = self.test_estimators[i].predict(X_test)
            mean = results[0]
            var = results[1]
            means.append(mean)
            variances.append(var)

        means = np.array(means)
        variances = np.array(variances)

        var_per_model = np.mean(variances + np.square(means),
            axis=1) - np.square(np.mean(means, axis=1))
        var_per_model[var_per_model < 0.0] = 0.0

        return np.sqrt(var_per_model)
```

A.4.4 *Datei Detection_Strategies.py*

Listing A.4: Detection_Strategies.py

```
import numpy as np
from skmultiflow.drift_detection import ADWIN, PageHinkley
import pandas as pd
from sklearn.metrics import log_loss, accuracy_score,
    mean_absolute_error, matthews_corrcoef, mean_squared_error, \
     roc_auc_score
from scipy.stats import spearmanr
import random


def smape(A, F):
    return 100 / len(A) * np.sum(2 * np.abs(F - A) / (np.abs(A)
        + np.abs(F)))


def compute_metrics(prediction_type, true_values, predictions,
    class_probabilities, targets, uncertainties,
                    detected_drifts, retraining_counter):
    metrics = {}

    if prediction_type == 'regression':
        metrics['type'] = 'regression'
        metrics['MAE'] = mean_absolute_error(true_values,
            predictions)
        metrics['MSE'] = mean_squared_error(true_values,
            predictions)
        metrics['RMSE'] = np.sqrt(mean_squared_error(true_values
            , predictions))
        metrics['SMAPE'] = smape(true_values, predictions)
        errors = pd.Series(np.square(np.absolute(true_values -
            predictions)))

    if prediction_type == 'classification':

        metrics['type'] = 'classification'
        metrics['Acc'] = accuracy_score(true_values, predictions
            )

        if (len(class_probabilities) != np.unique(true_values).
            shape[0]):
            metrics['Log-loss'] = log_loss(true_values,
                class_probabilities, labels=np.arange(targets))
        else:
```

```
            metrics['Log-loss'] = log_loss(true_values,
                class_probabilities)

        metrics['MCC'] = matthews_corrcoef(true_values,
            predictions)

        if targets > 2:
            metrics['AUC'] = roc_auc_score(true_values,
                class_probabilities, average='weighted',
                multi_class='ovo', labels=np.arange(targets))
        else:
            metrics['AUC'] = roc_auc_score(true_values,
                class_probabilities[:, 1])

        errors = pd.Series((~np.equal(true_values, predictions))
            .astype(int))

    metrics['r_spear'] = spearmanr(errors, uncertainties)[0]
    metrics['r_pears'] = np.corrcoef(errors, uncertainties)[0,
        1]

    metrics['detected_drift_numbers'] = len(detected_drifts)
    metrics['retraining_counter'] = retraining_counter

    return metrics, errors


def execute_stream_no_drift_detection(stream, model,
    retrain_points, retrain_use_Xtrain, X_train, y_train,
    retrain_size,
                                      prediction_type='
                                          regression',
                                          retrain_all=True):
    retrain_size = int((stream.n_remaining_samples() / 0.85) *
        0.01)
    print('Retrain size: ', retrain_size)

    predictions = []
    true_values = []
    uncertainties = []
    predictions_proba = []
    X_retrain = []

    overall_i = 0
    retraining_counter = 0

    while stream.n_remaining_samples() > 1:
        # Standard batch size for computation
```

```
        batch_size = 1000

        # Check if remaining stream has still enough instances
        if (stream.n_remaining_samples() < batch_size):
            batch_size = stream.n_remaining_samples()

        X_test, y_test = stream.next_sample(batch_size)

        if prediction_type == 'regression':
            y_pred, y_pred_uncertainty = model.predict(X_test)
            predictions.append(y_pred)
        if prediction_type == 'classification':
            y_pred, y_pred_uncertainty, y_pred_proba = model.
                predict(X_test)
            predictions.append(y_pred)
            predictions_proba.append(y_pred_proba)

        true_values.append(y_test)
        uncertainties.append(y_pred_uncertainty)
        X_retrain.append(X_test)

        for j in range(len(y_pred_uncertainty)):

            if (overall_i + j) in retrain_points:
                retraining_counter += 1

                # Remove training point
                _ = retrain_points.pop(0)

                # Remove excess training data
                _ = X_retrain.pop()
                _ = true_values.pop()
                _ = predictions.pop()
                _ = uncertainties.pop()
                if prediction_type == 'classification':
                    _ = predictions_proba.pop()

                # Add data up to drift instead
                X_retrain.append(X_test[:j, :])
                true_values.append(y_test[:j])
                if prediction_type == 'regression':
                    predictions.append(y_pred[:j])
                if prediction_type == 'classification':
                    predictions.append(y_pred[:j])
                    predictions_proba.append(y_pred_proba[:j])
                uncertainties.append(y_pred_uncertainty[:j])

                # Stack training data
```

```
                X_retrain_array = np.vstack(X_retrain)
                true_values_array = np.hstack(true_values)
                # Attention, need to adapt X_retrain
                if retrain_use_Xtrain:
                    print('retrain size:', np.concatenate([
                        X_train, X_retrain_array[-retrain_size:,
                        :]]).shape)
                    model.retrain_all_members(np.concatenate([
                        X_train, X_retrain_array[-retrain_size:,
                        :]]), np.concatenate([y_train,
                        true_values_array[-retrain_size:]]))
                else:
                    print('retrain size:', np.concatenate([
                        X_retrain_array[-retrain_size:, :]]).
                        shape)
                    model.retrain_all_members(np.concatenate([
                        X_retrain_array[-retrain_size:, :]]), np.
                        concatenate([true_values_array[-
                        retrain_size:]]))
                batch_size = j

                # Restart stream and set to correct position
                stream.restart()
                _, _ = stream.next_sample((3 * X_train.shape[0])
                     + (overall_i + j))
                break

            # Need to adapt overall_i
        overall_i += batch_size

        results = {}
        results['predictions'] = predictions
        results['true_values'] = true_values
        results['uncertainties'] = uncertainties
        results['class_probabilities'] = predictions_proba
        results['X_retrain'] = X_retrain
        results['retraining_counter'] = retraining_counter

    return results


class Adwin_Uncertainty():
    def __init__(self):
        self.name = 'adwin_uncertainty'
        self.delta = 0.002
        self.prediction_type = 'regression'

    def set_parameter(self, delta):
```

```
        self.delta = delta

    def gridsearch_adwin_parameter(self, stream, model,
        drifts_to_detect, targets, features=1, starting_value=-3)
        :

        stream.restart()

        five_percent = int(stream.n_remaining_samples() * 0.05)

        X_train, y_train = stream.next_sample(five_percent)

        # fit algorithm
        model.fit(X_train, y_train)

        condition = False
        m = starting_value  # -1
        n = starting_value  # -1

        X_test, y_test = stream.next_sample(2 * five_percent)

        if self.prediction_type == 'regression':
            y_pred, y_pred_uncertainty = model.predict(X_test)
        if self.prediction_type == 'classification':
            y_pred, y_pred_uncertainty, y_pred_proba = model.
                predict(X_test)

        counter = 0

        while condition == False:
            print('-------------------------')
            param_box_upper = 10 ** m
            param_box_lower = 10 ** (n - 1)

            # evaluate for param_box_upper
            drifts_upper = []
            adwin = ADWIN(param_box_upper)

            for i in range(len(y_pred_uncertainty)):
                adwin.add_element(y_pred_uncertainty[i])
                if adwin.detected_change():
                    drifts_upper.append(i)
                    adwin.reset()

            # evaluate for param_box_lower
            drifts_lower = []
            adwin = ADWIN(param_box_lower)
```

```
            for i in range(len(y_pred_uncertainty)):
                adwin.add_element(y_pred_uncertainty[i])
                if adwin.detected_change():
                    drifts_lower.append(i)
                    adwin.reset()

            print('Adwin Parameter: {} - Detected Drifts: {}'.
                format(param_box_upper, len(drifts_upper)))
            print('Adwin Parameter: {} - Detected Drifts: {}'.
                format(param_box_lower, len(drifts_lower)))

            counter += 1

            if len(drifts_upper) == drifts_to_detect:
                print('case1')
                parameter = param_box_upper
                condition = True

            elif len(drifts_lower) == drifts_to_detect:
                print('case1')
                parameter = param_box_lower
                condition = True

            elif ((param_box_upper < 1e-20) & (param_box_lower <
                 1e-21)) | counter > 30:
                print('case5')
                parameter = 0.002
                condition = True

            elif (len(drifts_upper) > drifts_to_detect) & (len(
                drifts_lower) > drifts_to_detect):
                print('case2')
                m -= 1
                n -= 1

            elif (len(drifts_upper) < drifts_to_detect) & (len(
                drifts_lower) < drifts_to_detect) & (
                    param_box_upper == 0.1):
                print('case3')

                parameter = 0.002
                condition = True

            elif (len(drifts_upper) > drifts_to_detect) & (len(
                drifts_lower) < drifts_to_detect):
                print('case4')

                m -= 0.05
```

```
241                 n += 0.05

            print('Final ADWIN parameter: ', parameter)
            self.delta = parameter

246     def run_stream(self, stream, model, X_train, y_train,
            retrain, prediction_type, retrain_after,
            retrain_use_Xtrain,
                       ks_features, targets, retrain_all=True):

            adwin = ADWIN(self.delta)

251         retrain_size = int((stream.n_remaining_samples() / 0.85)
                * 0.01)
            print('Retrain size: ', retrain_size)

            predictions = []
            predictions_proba = []
256         true_values = []
            uncertainties = []
            class_probabilities = []
            accepted_drifts = []
            detected_drifts = []
261         X_retrain = []
            variance = []
            window_width = []

            overall_i = 0
266         next_retrain = -1
            retraining_counter = 0

            while stream.n_remaining_samples() > 1:

271             # Standard batch size for computation
                batch_size = 1000

                # Check if remaining stream has still enough
                    instances
                if (stream.n_remaining_samples() < batch_size):
276                 batch_size = stream.n_remaining_samples()

                X_test, y_test = stream.next_sample(batch_size)

                if self.prediction_type == 'regression':
281                 y_pred, y_pred_uncertainty = model.predict(
                        X_test)
                    predictions.append(y_pred)
                if self.prediction_type == 'classification':
```

```
                y_pred, y_pred_uncertainty, y_pred_proba = model
                    .predict(X_test)
                predictions.append(y_pred)
                predictions_proba.append(y_pred_proba)

            true_values.append(y_test)
            uncertainties.append(y_pred_uncertainty)
            X_retrain.append(X_test)

            for j in range(len(y_pred_uncertainty)):

                adwin.add_element(y_pred_uncertainty[j])

                if adwin.detected_change():
                    if not accepted_drifts:
                        if retrain:
                            next_retrain = overall_i + j +
                                retrain_after
                    elif (overall_i + j) - accepted_drifts[-1] >
                         retrain_after:
                        if retrain:
                            next_retrain = overall_i + j +
                                retrain_after

                    if not accepted_drifts:
                        accepted_drifts.append(overall_i + j)
                    elif (overall_i + j) - accepted_drifts[-1] >
                         retrain_after:
                        accepted_drifts.append(overall_i + j)

                    detected_drifts.append(overall_i + j)

                    adwin.reset()

                window_width.append(adwin.width)
                if adwin.width != 0:
                    variance.append(adwin.variance)
                else:
                    variance.append(0)

                if (overall_i + j) == next_retrain:

                    retraining_counter += 1

                    # Remove excess training data
                    _ = X_retrain.pop()
                    _ = true_values.pop()
                    _ = predictions.pop()
```

```
                    _ = uncertainties.pop()
                    if prediction_type == 'classification':
                        _ = predictions_proba.pop()

                    # Add data up to drift instead
                    X_retrain.append(X_test[:j, :])
                    true_values.append(y_test[:j])
                    if prediction_type == 'regression':
                        predictions.append(y_pred[:j])
                    if prediction_type == 'classification':
                        predictions.append(y_pred[:j])
                        predictions_proba.append(y_pred_proba[:j
                            ])
                    uncertainties.append(y_pred_uncertainty[:j])

                    # Stack training data
                    X_retrain_array = np.vstack(X_retrain)
                    true_values_array = np.hstack(true_values)

                    # Attention, need to adapt X_retrain
                    if retrain_all:
                        if retrain_use_Xtrain:
                            print('retrain size:', np.
                                concatenate([X_train,
                                X_retrain_array[-retrain_size:,
                                :]]).shape)
                            model.retrain_all_members(np.
                                concatenate([X_train,
                                X_retrain_array[-retrain_size:,
                                :]]), np.concatenate([y_train,
                                true_values_array[-retrain_size
                                :]]))
                        else:
                            print('retrain size:', np.
                                concatenate([X_retrain_array[-
                                retrain_size:, :]]).shape)
                            model.retrain_all_members(np.
                                concatenate([X_retrain_array[-
                                retrain_size:, :]]), np.
                                concatenate([true_values_array[-
                                retrain_size:]]))
                    else:
                        if retrain_use_Xtrain:
                            print('retrain size:', np.
                                concatenate([X_train,
                                X_retrain_array[-retrain_size:,
                                :]]).shape)
```

```
                            model.retrain_worst_model(np.
                                concatenate([X_train,
                                X_retrain_array[-retrain_size:,
                                :]]), np.concatenate([y_train,
                                true_values_array[-retrain_size
                                :]]), X_test)
                        else:
                            print('retrain size:', np.
                                concatenate([X_retrain_array[-
                                retrain_size:, :]]).shape)
                            model.retrain_worst_model(np.
                                concatenate([X_retrain_array[-
                                retrain_size:, :]]), np.
                                concatenate([true_values_array[-
                                retrain_size:]]), X_test)

                    batch_size = j
                    next_retrain = -1

                    # Restart stream and set to correct position
                    stream.restart()
                    _, _ = stream.next_sample((3 * X_train.shape
                        [0]) + (overall_i + j))
                    break

            # Need to adapt overall_i
            overall_i += batch_size

        if prediction_type == 'regression':
            predictions = np.hstack(np.concatenate(predictions,
                axis=0))
        if prediction_type == 'classification':
            predictions = np.hstack(predictions)
            class_probabilities = np.vstack(predictions_proba)

        uncertainties = np.hstack(np.concatenate(uncertainties,
            axis=0))
        true_values = np.concatenate(true_values, axis=0)
        variance = np.asarray(variance)
        window_width = np.asarray(window_width)

        metrics, errors = compute_metrics(prediction_type,
            true_values, predictions, class_probabilities,
            targets, uncertainties, detected_drifts,
            retraining_counter)

        print('ADWIN Uncertainty: ', detected_drifts)

```

```
        raw_results = {'uncertainties': uncertainties, 'tv':
            true_values, 'tv_cat': None, 'preds': predictions, '
            probs': class_probabilities, 'errors': errors, '
            variances': variance, 'window_with': window_width}

        return metrics, detected_drifts, raw_results


class No_Retraining():
    def __init__(self):
        self.name = 'no_retraining'

    def run_stream(self, stream, model, X_train, y_train,
        retrain, prediction_type, retrain_after,
        retrain_use_Xtrain,
                   ks_features, targets, retrain_all=True):

        predictions = []
        true_values = []
        uncertainties = []
        class_probabilities = []
        predictions_proba = []

        stream.restart()

        batch_size = stream.n_remaining_samples()

        X_test, y_test = stream.next_sample(batch_size)

        if prediction_type == 'regression':
            y_pred, y_pred_uncertainty = model.predict(X_test)
            predictions.append(y_pred)
        if prediction_type == 'classification':
            y_pred, y_pred_uncertainty, y_pred_proba = model.
                predict(X_test)
            predictions.append(y_pred)
            predictions_proba.append(y_pred_proba)

        true_values.append(y_test)
        uncertainties.append(y_pred_uncertainty)

        if prediction_type == 'regression':
            predictions = np.hstack(predictions)
        if prediction_type == 'classification':
            predictions = np.hstack(predictions)
            class_probabilities = np.vstack(predictions_proba)

        true_values = np.hstack(true_values)
```

```
        predictions = np.hstack(predictions)
        uncertainties = np.hstack(np.concatenate(uncertainties,
            axis=0))

        detected_drifts = []
        retraining_counter = 0

        metrics, errors = compute_metrics(prediction_type,
            true_values, predictions, class_probabilities,
            targets, uncertainties, detected_drifts,
            retraining_counter)

        raw_results = {'uncertainties': uncertainties, 'tv':
            true_values, 'tv_cat': None, 'preds': predictions, '
            probs': class_probabilities, 'errors': errors}

        return metrics, detected_drifts, raw_results


class Uninformed():
    '''
    Generates randomly retrain points out of all possible
        training points
    '''

    def __init__(self, number_retrainings):
        self.name = 'uninformed'
        self.number_retrainings = number_retrainings

    def run_stream(self, stream, model, X_train, y_train,
        retrain, prediction_type, retrain_after,
        retrain_use_Xtrain, ks_features, targets, retrain_all=
        True):

        retrain_size = int((stream.n_remaining_samples() /
            0.850) * 0.01)

        # generate random retraining points
        k = self.number_retrainings
        n = stream.n_remaining_samples()
        d = retrain_size

        sample = random.sample(range(n - (k - 1) * (d - 1)), k)
        indices = sorted(range(len(sample)), key=lambda i:
            sample[i])
        return_indices = sorted(indices, key=lambda i: indices[i
            ])
```

```
        values = [s + (d - 1) * r for s, r in zip(sample,
            return_indices)]
        values.sort()

        print(self.name, ': ', values)

        results = execute_stream_no_drift_detection(stream,
            model, values, retrain_use_Xtrain, X_train, y_train,
            retrain_size, prediction_type=prediction_type,
            retrain_all=True)

        predictions = results['predictions']
        true_values = results['true_values']
        uncertainties = results['uncertainties']
        class_probabilities = results['class_probabilities']
        X_retrain = results['X_retrain']
        retraining_counter = results['retraining_counter']

        if prediction_type == 'regression':
            # predictions = np.hstack(predictions)
            predictions = np.hstack(np.concatenate(predictions,
                axis=0))
        if prediction_type == 'classification':
            predictions = np.hstack(predictions)
            class_probabilities = np.vstack(class_probabilities)

        true_values = np.hstack(true_values)
        uncertainties = np.hstack(np.concatenate(uncertainties,
            axis=0))

        detected_drifts = []

        metrics, errors = compute_metrics(prediction_type,
            true_values, predictions, class_probabilities,
            targets, uncertainties, detected_drifts,
            retraining_counter)

        raw_results = {'uncertainties': uncertainties, 'tv':
            true_values, 'tv_cat': None, 'preds': predictions, '
            probs': class_probabilities, 'errors': errors}

        return metrics, detected_drifts, raw_results


class Equal_Distribution():
    '''
    Generates equal distributed retrain points
    '''
```

```
    def __init__(self, number_retrainings):
        self.name = 'equal_distribution'
        self.number_retrainings = number_retrainings

    def run_stream(self, stream, model, X_train, y_train,
        retrain, prediction_type, retrain_after,
        retrain_use_Xtrain,
                   ks_features, targets, retrain_all=True):
        retrain_size = int((stream.n_remaining_samples() /
            0.850) * 0.01)

        # generate random retraining points
        k = self.number_retrainings
        n = stream.n_remaining_samples()

        # divide n by number of retrainings + 2
        space = int(np.round(n / (k + 1), 0))
        values = [i * space for i in range(1, k + 1)]  # set
            index values for retraining
        values.sort()
        print(self.name, ': ', values)

        # Execute stream logic
        results = execute_stream_no_drift_detection(stream,
            model, values, retrain_use_Xtrain, X_train, y_train,
            retrain_size, prediction_type=prediction_type,
            retrain_all=True)

        predictions = results['predictions']
        true_values = results['true_values']
        uncertainties = results['uncertainties']
        class_probabilities = results['class_probabilities']
        retraining_counter = results['retraining_counter']

        if prediction_type == 'regression':
            predictions = np.hstack(np.concatenate(predictions,
                axis=0))
        if prediction_type == 'classification':
            predictions = np.hstack(predictions)
            class_probabilities = np.vstack(class_probabilities)

        true_values = np.hstack(true_values)
        uncertainties = np.hstack(np.concatenate(uncertainties,
            axis=0))

        detected_drifts = []
```

```
        metrics, errors = compute_metrics(prediction_type,
            true_values, predictions, class_probabilities,
            targets, uncertainties, detected_drifts,
            retraining_counter)

        raw_results = {'uncertainties': uncertainties, 'tv':
            true_values, 'tv_cat': None, 'preds': predictions, '
            probs': class_probabilities, 'errors': errors}

        return metrics, detected_drifts, raw_results


class PageHinkley_Uncertainty():
    def __init__(self):
        self.name = 'page_hinkley_uncertainty'
        self.delta = 0.005
        self.prediction_type = 'regression'

    def set_parameter(self, delta):
        self.delta = delta

    def gridsearch_ph_parameter(self, stream, model,
        drifts_to_detect, targets, features=1, starting_value=-3)
        :

        stream.restart()

        five_percent = int(stream.n_remaining_samples() * 0.05)

        X_train, y_train = stream.next_sample(five_percent)

        # fit algorithm
        model.fit(X_train, y_train)

        condition = False
        m = starting_value  # -1
        n = starting_value  # -1

        X_test, y_test = stream.next_sample(2 * five_percent)

        if self.prediction_type == 'regression':
            y_pred, y_pred_uncertainty = model.predict(X_test)
        if self.prediction_type == 'classification':
            y_pred, y_pred_uncertainty, y_pred_proba = model.
                predict(X_test)

        while condition == False:
            print('-------------------------')
```

```
            param_box_upper = 10 ** m
            param_box_lower = 10 ** (n - 1)

            # evaluate for param_box_upper
            drifts_upper = []
            pagehinkley = PageHinkley(param_box_upper)

            for i in range(len(y_pred_uncertainty)):
                pagehinkley.add_element(y_pred_uncertainty[i])
                if pagehinkley.detected_change():
                    # print('Change has been detected in data: -
                        of index: ' + str(i))
                    drifts_upper.append(i)
                    pagehinkley.reset()

            # evaluate for param_box_lower
            drifts_lower = []
            pagehinkley = PageHinkley(param_box_lower)

            for i in range(len(y_pred_uncertainty)):
                pagehinkley.add_element(y_pred_uncertainty[i])
                if pagehinkley.detected_change():
                    # print('Change has been detected in data: -
                        of index: ' + str(i))
                    drifts_lower.append(i)
                    pagehinkley.reset()

            print('PH Parameter: {} - Detected Drifts: {}'.
                format(param_box_upper, len(drifts_upper)))
            print('PH Parameter: {} - Detected Drifts: {}'.
                format(param_box_lower, len(drifts_lower)))

            if len(drifts_upper) == drifts_to_detect:
                print('case1')
                parameter = param_box_upper
                condition = True

            elif len(drifts_lower) == drifts_to_detect:
                print('case1')
                parameter = param_box_lower
                condition = True

            elif (param_box_upper < 1e-10) & (param_box_lower <
                1e-10):
                print('case5')
                parameter = 0.005
                condition = True
```

```
            elif (len(drifts_upper) > drifts_to_detect) & (len(
                drifts_lower) > drifts_to_detect):
                print('case2')
                m -= 1
                n -= 1

            elif (len(drifts_upper) < drifts_to_detect) & (len(
                drifts_lower) < drifts_to_detect) & (
                    param_box_upper == 0.1):
                print('case3')
                parameter = 0.005
                condition = True

            elif (len(drifts_upper) > drifts_to_detect) & (len(
                drifts_lower) < drifts_to_detect):
                print('case4')
                m -= 0.05
                n += 0.05

        print('Final PH parameter: ', parameter)
        self.delta = parameter

    def run_stream(self, stream, model, X_train, y_train,
        retrain, prediction_type, retrain_after,
        retrain_use_Xtrain,
                   ks_features, targets, retrain_all=True):

        pagehinkley = PageHinkley(delta=self.delta)

        retrain_size = int((stream.n_remaining_samples() / 0.85)
             * 0.01)

        predictions = []
        predictions_proba = []
        true_values = []
        uncertainties = []
        class_probabilities = []
        accepted_drifts = []
        detected_drifts = []
        X_retrain = []
        variance = []
        window_width = []

        overall_i = 0
        next_retrain = -1
        retraining_counter = 0

        while stream.n_remaining_samples() > 1:
```

```
            # Standard batch size for computation
666         batch_size = 1000

            # Check if remaining stream has still enough
                instances
            if (stream.n_remaining_samples() < batch_size):
                batch_size = stream.n_remaining_samples()
671
            X_test, y_test = stream.next_sample(batch_size)

            if self.prediction_type == 'regression':
                y_pred, y_pred_uncertainty = model.predict(
                    X_test)
676             predictions.append(y_pred)
            if self.prediction_type == 'classification':
                y_pred, y_pred_uncertainty, y_pred_proba = model
                    .predict(X_test)
                predictions.append(y_pred)
                predictions_proba.append(y_pred_proba)
681
            true_values.append(y_test)
            uncertainties.append(y_pred_uncertainty)
            X_retrain.append(X_test)

686         for j in range(len(y_pred_uncertainty)):

                pagehinkley.add_element(y_pred_uncertainty[j])

                if pagehinkley.detected_change():
691                 if not accepted_drifts:
                        if retrain:
                            next_retrain = overall_i + j +
                                retrain_after
                    elif (overall_i + j) - accepted_drifts[-1] >
                         retrain_after:
                        if retrain:
696                         next_retrain = overall_i + j +
                                retrain_after

                    if not accepted_drifts:
                        accepted_drifts.append(overall_i + j)
                    elif (overall_i + j) - accepted_drifts[-1] >
                         retrain_after:
701                     accepted_drifts.append(overall_i + j)

                    detected_drifts.append(overall_i + j)
```

```
                    pagehinkley.reset()
706
                if (overall_i + j) == next_retrain:

                    retraining_counter += 1

711                 # Remove excess training data
                    _ = X_retrain.pop()
                    _ = true_values.pop()
                    _ = predictions.pop()
                    _ = uncertainties.pop()
716                 if prediction_type == 'classification':
                        _ = predictions_proba.pop()

                    # Add data up to drift instead
                    X_retrain.append(X_test[:j, :])
721                 true_values.append(y_test[:j])
                    if prediction_type == 'regression':
                        predictions.append(y_pred[:j])
                    if prediction_type == 'classification':
                        predictions.append(y_pred[:j])
726                     predictions_proba.append(y_pred_proba[:j
                            ])
                    uncertainties.append(y_pred_uncertainty[:j])

                    # Stack training data
                    X_retrain_array = np.vstack(X_retrain)
731                 true_values_array = np.hstack(true_values)

                    # Attention, need to adapt X_retrain
                    if retrain_all:
                        if retrain_use_Xtrain:
736                         print('retrain size:', np.
                                concatenate([X_train,
                                X_retrain_array[-retrain_size:,
                                :]]).shape)
                            model.retrain_all_members(np.
                                concatenate([X_train,
                                X_retrain_array[-retrain_size:,
                                :]]), np.concatenate([y_train,
                                true_values_array[-retrain_size
                                :]]))
                        else:
                            print('retrain size:', np.
                                concatenate([X_retrain_array[-
                                retrain_size:, :]]).shape)
                            model.retrain_all_members(np.
                                concatenate([X_retrain_array[-
```

```
                                retrain_size:, :]]), np.
                                concatenate([true_values_array[-
                                retrain_size:]]))
                    else:
                        if retrain_use_Xtrain:
                            print('retrain size:', np.
                                concatenate([X_train,
                                X_retrain_array[-retrain_size:,
                                :]]).shape)
                            model.retrain_worst_model(np.
                                concatenate([X_train,
                                X_retrain_array[-retrain_size:,
                                :]]), np.concatenate([y_train,
                                true_values_array[-retrain_size
                                :]]), X_test)
                        else:
                            print('retrain size:', np.
                                concatenate([X_retrain_array[-
                                retrain_size:, :]]).shape)
                            model.retrain_worst_model(np.
                                concatenate([X_retrain_array[-
                                retrain_size:, :]]), np.
                                concatenate([true_values_array[-
                                retrain_size:]]), X_test)

                    batch_size = j
                    next_retrain = -1

                    # Restart stream and set to correct position
                    stream.restart()
                    _, _ = stream.next_sample((3 * X_train.shape
                        [0]) + (overall_i + j))
                    break

            # Need to adapt overall_i
            overall_i += batch_size

        if prediction_type == 'regression':
            predictions = np.hstack(np.concatenate(predictions,
                axis=0))
        if prediction_type == 'classification':
            predictions = np.hstack(predictions)
            class_probabilities = np.vstack(predictions_proba)

        uncertainties = np.hstack(np.concatenate(uncertainties,
            axis=0))
        true_values = np.concatenate(true_values, axis=0)
        variance = np.asarray(variance)
```

```
        window_width = np.asarray(window_width)

771     metrics, errors = compute_metrics(prediction_type,
            true_values, predictions, class_probabilities,
            targets, uncertainties, detected_drifts,
            retraining_counter)

        print('Page Hinkley Uncertainty: ', detected_drifts)

        raw_results = {'uncertainties': uncertainties, 'tv':
            true_values, 'tv_cat': None, 'preds': predictions, '
            probs': class_probabilities, 'errors': errors, '
            variances': variance, 'window_with': window_width}
776
        return metrics, detected_drifts, raw_results
```

A.5 ERGEBNISSE

A.5.1 *Synthetische Datensätze*

A.5.1.1 *Regression*

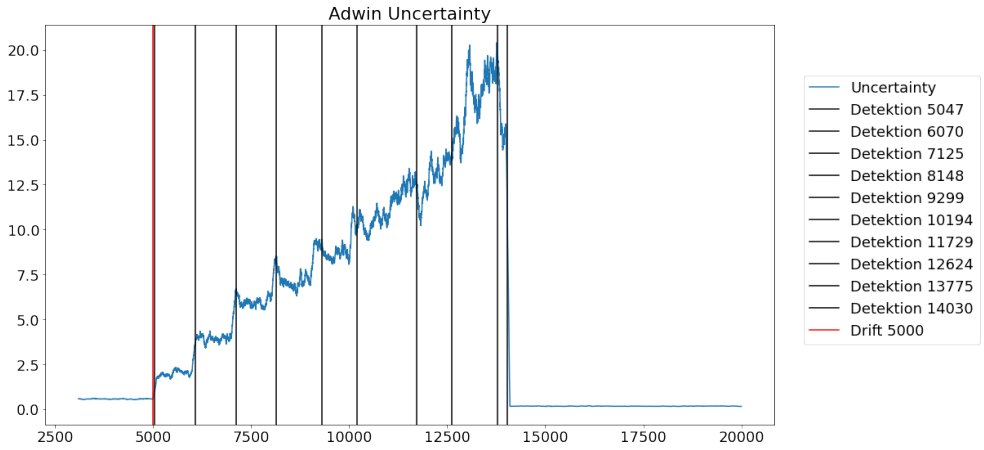

Abbildung A.6: Modellunsicherheit und Detektionen über Zeit bei *refit-all* der Friedman inkrementell Daten.

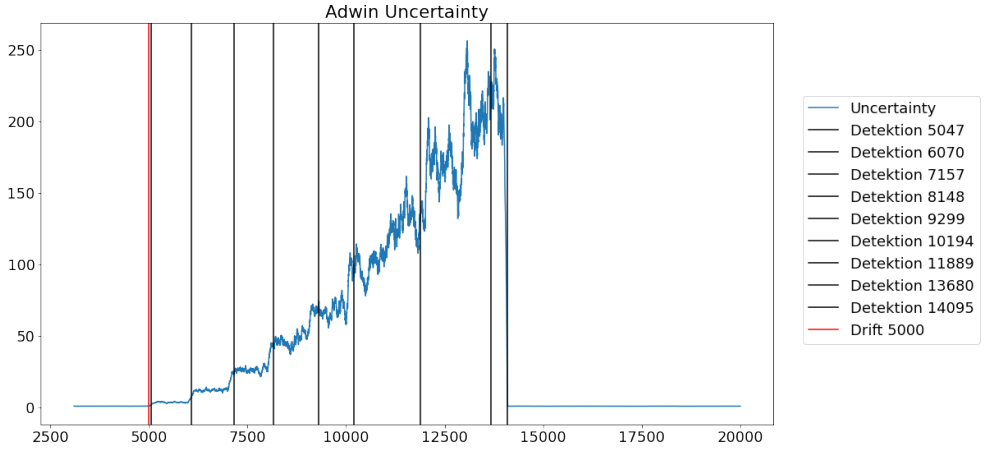

Abbildung A.7: Modellunsicherheit und Detektionen über Zeit bei *refit-worst* der Friedman inkrementell Daten

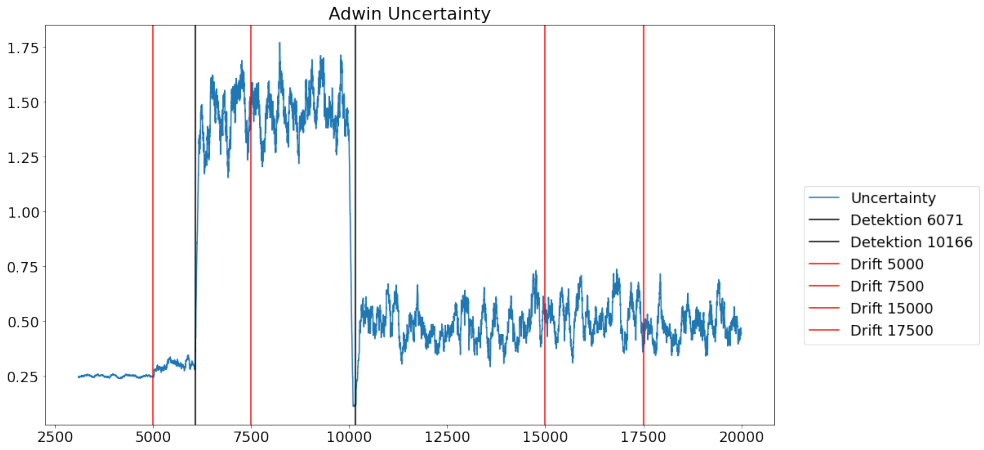

Abbildung A.8: Modellunsicherheit und Detektionen über Zeit bei *refit-all* der Friedman gradual Daten

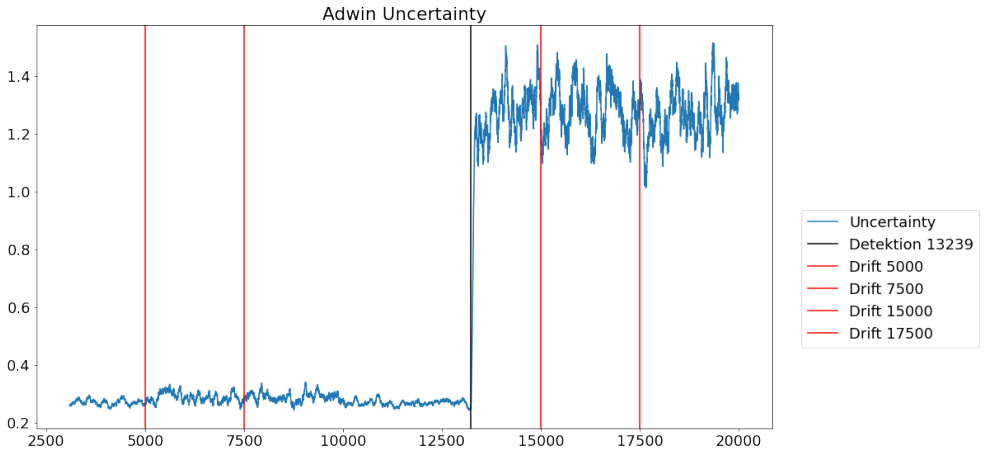

Abbildung A.9: Modellunsicherheit und Detektionen über Zeit bei *refit-worst* der Friedman gradual Daten.

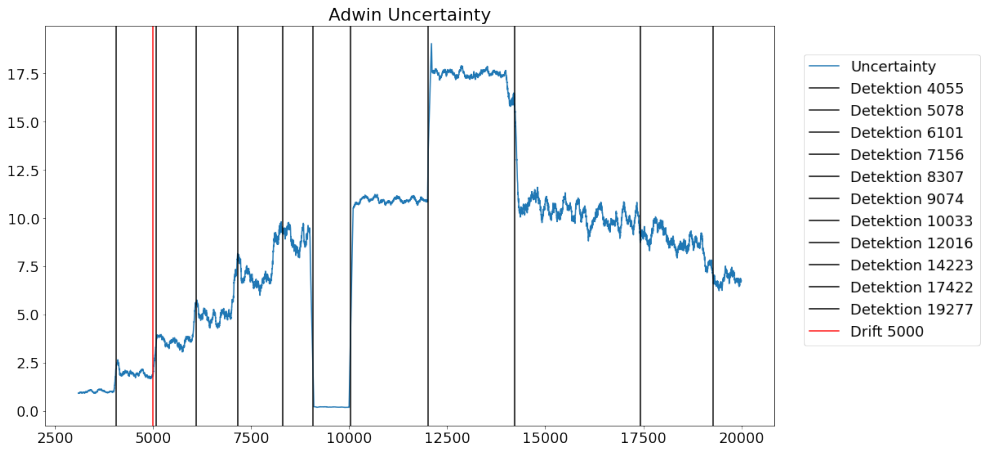

Abbildung A.10: Modellunsicherheit und Detektionen über Zeit bei *refit-all* der Friedman No Return Daten.

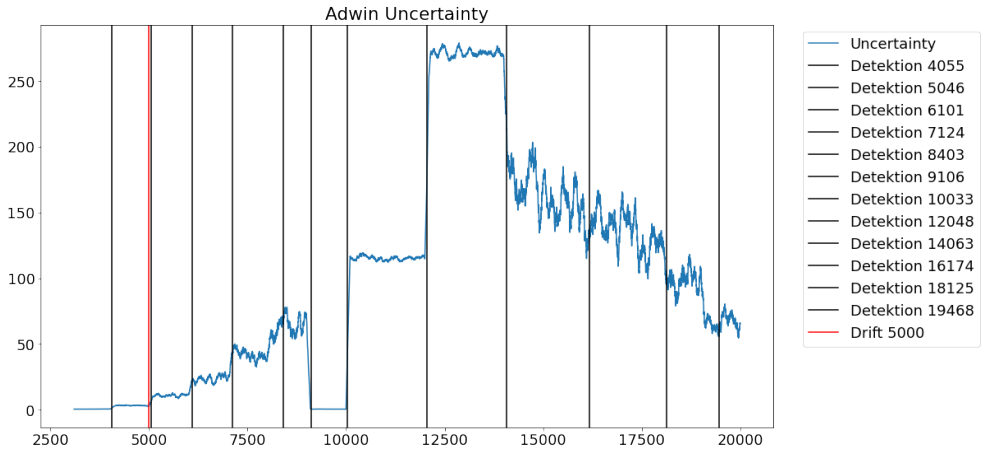

Abbildung A.11: Modellunsicherheit und Detektionen über Zeit bei *refit-worst* der Friedman No Return Daten

A.5.1.2 *Klassifikation*

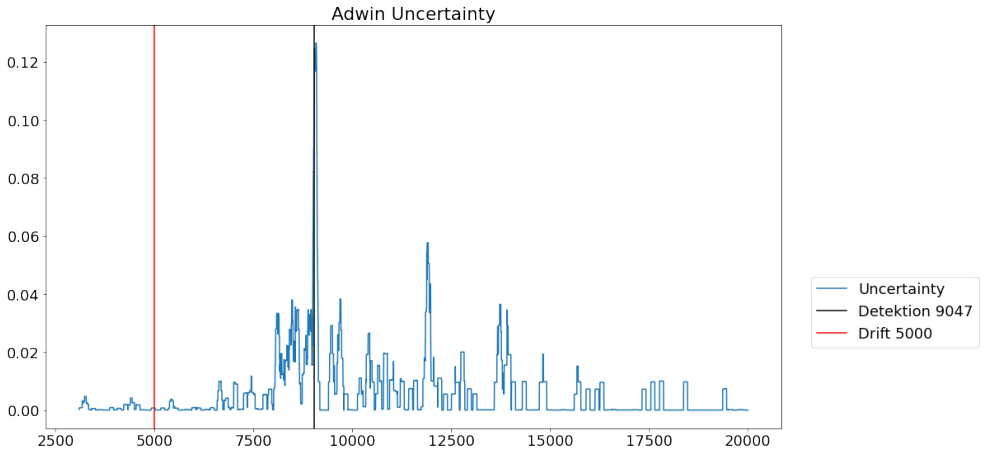

Abbildung A.12: Modellunsicherheit und Detektionen über Zeit bei *refit-all* der Mixed inkrementell Daten

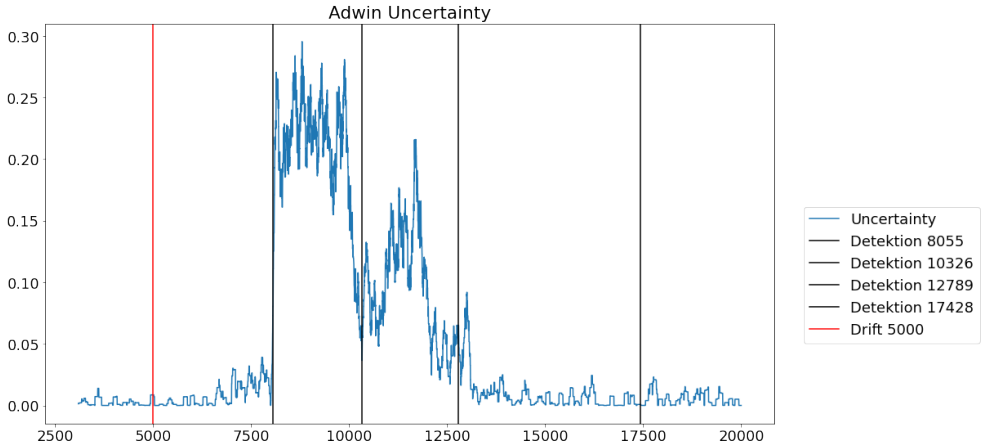

Abbildung A.13: Modellunsicherheit und Detektionen über Zeit bei *refit-worst* der Mixed inkrementell Daten

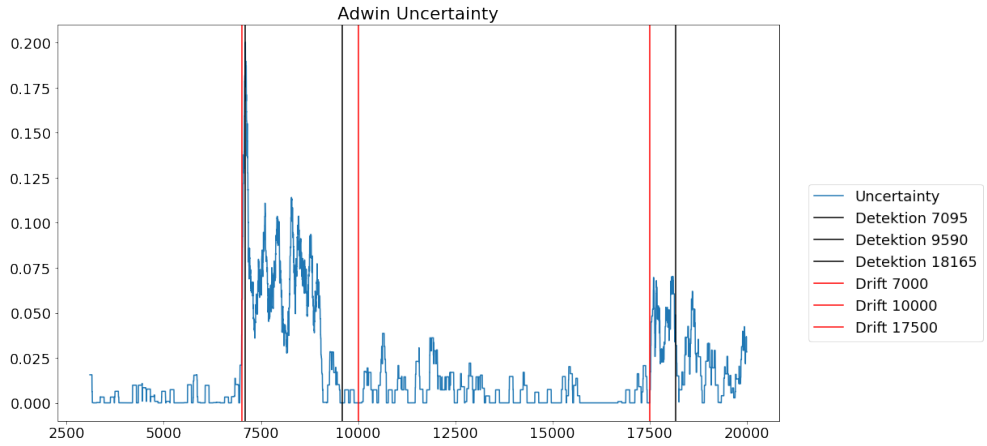

Abbildung A.14: Modellunsicherheit und Detektionen über Zeit bei *refit-all* der Mixed Target (3) Daten

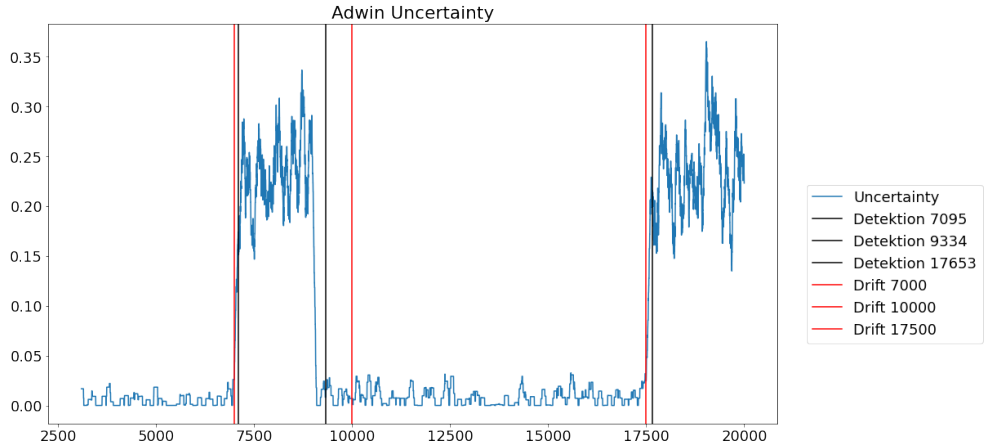

Abbildung A.15: Modellunsicherheit und Detektionen über Zeit bei *refit-worst* der Mixed Target (3) Daten

Tabelle A.3: AUC (Anzahl Re-Trainings) der synthetischen Klassifikations-Datensätze mit M=5 und refit_use_X_train=True

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Mixed Abrupt	0.92(6)	0.80(6)	**0.94**(**2**)	0.80(0)	0.83(6)	0.80(6)
Mixed inkrementell	0.95(4)	**0.97**(**2**)	0.95(2)	0.93(0)	0.87(4)	0.90(4)
Mixed Target (3)	**0.98**(**4**)	0.95(4)	0.87(0)	0.87(0)	0.95(4)	0.96(4)

A.5.2 *Real-World Datensätze*

A.5.2.1 *Regression*

Tabelle A.4: **RMSE** (Anzahl Re-Trainings) der Real-World Regressions-Datensätze mit M=5 und refit_use_X_train=True

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Air Quality	**28.26**(**24**)	37.66(26)	25.0(14)	39.98(0)	21.40(24)	23.44(24)
Bike Sharing	**7.32**(**13**)	14.57(15)	318.25(47	27.37(0)	8.91(13)	32.95(13)

Tabelle A.5: **RMSE** (Anzahl Re-Trainings) der Real-World Regressions-Datensätze mit M=5 und refit_use_X_train=False

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Air Quality	15.77(26)	36.50(21)	12.88(15)	40.45(0)	18.45(26)	**12.10**(**26**)
Bike Sharing	**4.08**(**15**)	14.05(14)	896.73(488	20.64(0)	4.35(15)	24.92(15)

Tabelle A.6: **RMSE** (Anzahl Re-Trainings) der Real-World Regressions-Datensätze M=5, nur 2 Hidden Layer und refit_use_X_train=True

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Air Quality	633.77(53)	364.82(37)	103.08(57)	**37.45**(**0**)	522.31(53)	412.95(46)
Bike Sharing	20.82(9)	11.35(13)	15.55(136)	3.83(0)	60.44(9)	**2.30**(**9**)

Tabelle A.7: **RMSE** (Anzahl Re-Trainings) der Real-World Regressions-Datensätze mit M=5, nur 2 Hidden Layer und refit_use_X_train=False

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Air Quality	87.60(48)	165.77(43)	89.42(63)	**36.98**(**0**)	48.24(48)	56.34(48)
Bike Sharing	3.59(10)	20.73(13)	88.52(412)	**2.73**(**0**)	4.17(10)	5.63(10)

Tabelle A.8: **RMSE** (Anzahl Re-Trainings) der Real-World Regressions-Datensätze mit M=10, 4 Hidden Layer und refit_use_X_train=False

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Air Quality	17.96(19)	39.94(22)	19.31(14)	39.91(0)	22.0(19)	11.10(19)
Bike Sharing	1.17(10)	0.85(13)	0.49(46)	0.93(0)	3.11(10)	1.34(10)

A.5.3 *Klassifikation*

Tabelle A.9: **MCC** (Anzahl Re-Trainings) der Real-World Klassifikations-Datensätze mit M=5 und refit_use_X_train=True

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Insects Abrupt	0.47(5)	0.45(4)	0.46(0)	**0.49**(**0**)	0.46(5)	0.38(5)
Insects Inc	**0.18**(**1**)	0.06(2)	0.04(0)	0.11(0)	0.14(1)	0.13(1)
Insects IncAbr	**0.47**(**21**)	0.39(17)	0.43(4)	0.32(0)	**0.47**(**21**)	0.41(21)
Insects IncReo	0.36(14)	0.26(16)	0.25(4)	0.20(0)	0.37(14)	**0.38**(**14**)
KDDCUP99	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)
Gassensor	0.30(20)	0.42(43)	**0.37**(**6**)	**0.37**(**0**)	0.36(20)	0.33(20)
Electricity	**0.39**(**5**)	0.32(14)	0.31(2)	0.22(0)	0.36(5)	0.24(5)

Tabelle A.10: **MCC** (Anzahl Re-Trainings) der Real-World Klassifikations-Datensätze mit M=5 und refit_use_X_train=False

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Insects Abrupt	0.36(3)	**0.53**(**4**)	0.49(7)	0.49(0)	0.31(3)	0.45(3)
Insects Inc	0.27(2)	0.18(3)	**0.48**(**4**)	0.12(0)	0.44(2)	0.28(2)
Insects IncAbr	0.40(21)	0.35(17)	0.30(0)	0.34(0)	**0.45**(**21**)	0.38(21)
Insects IncReo	0.55(19)	0.30(21)	0.52(14)	0.18(0)	**0.59**(**19**)	0.47(19)
KDDCUP99	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)
Gassensor	**0.51**(**30**)	0.34(43)	0.50(24)	0.34(0)	**0.51**(**20**)	0.32(20)
Electricity	**0.41**(**6**)	0.27(12)	0.16(2)	0.22(0)	0.33(6)	0.28(6)

Tabelle A.11: **MCC** (Anzahl Re-Trainings) der Real-World Klassifikations-Datensätze mit M=5, nur 2 Hidden Layer und refit_use_X_train=True

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Insects Abrupt	**0.54**(**5**)	**0.54**(**2**)	**0.54**(**4**)	0.50(0)	0.52(5)	0.51(5)
Insects Inc	0.24(2)	0.07(3)	**0.27**(**6**)	0.11(0)	0.24(2)	0.16(2)
Insects IncAbr	**0.47**(**13**)	0.41(16)	0.32(6)	0.31(0)	0.42(13)	0.33(13)
Insects IncReo	0.36(18)	0.27(20)	0.33(14)	0.18(0)	**0.40**(**18**)	0.30(18)
KDDCUP99	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)
Gassensor		0.39(39)	0.27(44)	0.36(6)	0.35(0)	**0.60**(**39**)
0.34(39)						
Electricity	**0.37**(**8**)	0.21(14)	0.35(9)	0.20(0)	0.41(8)	0.36(8)

Tabelle A.12: **AUC** (Anzahl Re-Trainings) der Real-World Klassifikations-Datensätze mit M=5, nur 2 Hidden Layer und refit_use_X_train=True

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Insects Abrupt	0.90(5)	**0.91**(**5**)	0.90(4)	0.89(0)	0.88(5)	0.88(5)
Insects Inc	0.76(2)	0.68(3)	**0.79**(**6**)	0.59(0)	0.76(2)	0.67(2)
Insects IncAbr	**0.87**(**13**)	0.84(16)	0.77(6)	0.70(0)	0.84(13)	0.79(13)
Insects IncReo	0.81(18)	0.76(20)	0.79(14)	0.63(0)	**0.84**(**18**)	0.77(18)
KDDCUP99	0.90(0)	0.92(0)	0.89(0)	0.88(0)	0.93(0)	0.98(0)
Gassensor	0.75(39)	0.70(44)	0.71(6)	0.70(0)	**0.84**(**39**)	0.78(39)
Electricity	0.75(8)	0.75(14)	0.71(9)	0.54(0)	**0.77**(**8**)	0.68(8)

Tabelle A.13: **MCC** (Anzahl Re-Trainings) der Real-World Klassifikations-Datensätze mit M=5, nur 2 Hidden Layer und refit_use_X_train=False

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Insects Abrupt	0.24(3)	0.54(5)	**0.57**(**7**)	0.49(0)	0.24(3)	0.29(3)
Insects Inc	**0.48**(**3**)	0.29(4)	**0.48**(**4**)	0.11(0)	0.44(3)	0.26(3)
Insects IncAbr	**0.50**(**18**)	0.31(18)	0.46(15)	0.31(0)	0.46(18)	0.41(18)
Insects IncReo	0.49(17)	0.23(22)	**0.56**(**15**)	0.18(0)	0.54(17)	0.48(17)
KDDCUP99	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)
Gassensor	**0.62**(**38**)	0.50(42)	0.22(4)	0.30(0)	0.58(38)	0.60(38)
Electricity	**0.42**(**9**)	0.20(39)	0.34(8)	0.22(0)	0.37(9)	0.39(9)

Tabelle A.14: **AUC** (Anzahl Re-Trainings) der Real-World Klassifikations-Datensätze mit M=5, nur 2 Hidden Layer und refit_use_X_train=False

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Insects Abrupt	0.73(3)	**0.91**(**5**)	**0.91**(**7**)	0.89(0)	0.72(3)	0.75(3)
Insects Inc	**0.88**(**3**)	0.79(4)	**0.88**(**4**)	0.59(0)	0.86(3)	0.69(3)
Insects IncAbr	**0.88**(**18**)	0.78(18)	0.84(15)	0.69(0)	0.85(18)	0.81(18)
Insects IncReo	0.82(17)	0.79(22)	**0.89**(**15**)	0.63(0)	0.88(17)	0.84(17)
KDDCUP99	0.90(0)	0.92(0)	0.91(0)	0.92(0)	0.89(0)	0.90(0)
Gassensor	0.84(38)	**0.87**(**42**)	0.60(4)	0.68(0)	0.85(38)	**0.87**(**38**)
Electricity	**0.77**(**9**)	0.76(18)	0.72(8)	0.73(0)	0.70(9)	0.71(9)

Tabelle A.15: **MCC** (Anzahl Re-Trainings) der Real-World Klassifikations-Datensätze mit M=10, nur 2 Hidden Layer, refit_use_X_train=False und num_models_to_retrain = 5

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Insects Abrupt	0.43(5)	0.43(7)	**0.50**(**3**)	0.49(0)	0.46(5)	0.41(5)
Insects Inc	0.38(2)	0.48(4)	**0.49**(**4**)	0.10(0)	0.42(2)	0.42(2)
Insects IncAbr	**0.53**(**20**)	0.22(21)	0.30(0)	0.32(0)	0.46(18)	0.40(18)
Insects IncReo	**0.61**(**21**)	0.30(24)	0.47(15)	0.18(0)	0.53(21)	0.45(21)
KDDCUP99	0.90(0)	0.92(0)	0.91(0)	0.92(0)	0.89(0)	0.90(0)
Gassensor	0.59(45)	0.62(48)	0.62(18)	0.38(0)	0.58(44)	**0.64**(**45**)
Electricity	**0.48**(**9**)	0.21(20)	0.24(2)	0.21(0)	0.36(9)	0.32(9)

Tabelle A.16: **AUC** (Anzahl Re-Trainings) der Real-World Klassifikations-Datensätze mit M=10, nur 2 Hidden Layer, refit_use_X_train=False und num_models_to_retrain = 5

Datensatz	EUDD		PH	No-Retr.	Eq. D.	Rand.
	retrain-all	retrain-worst				
Insects Abrupt	0.82(5)	0.88(7)	0.86(3)	**0.89**(**0**)	0.85(3)	0.84(3)
Insects Inc	0.82(2)	0.88(4)	**0.89**(**4**)	0.59(0)	0.84(2)	0.85(2)
Insects IncAbr	**0.89**(**20**)	0.87(21)	0.69(0)	0.70(0)	0.84(18)	0.80(18)
Insects IncReo	**0.91**(**21**)	0.87(24)	0.86(15)	0.63(0)	0.86(17)	0.84(17)
KDDCUP99	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)	0.99(0)
Gassensor	0.80(45)	**0.88**(**46**)	0.84(18)	0.70(0)	0.83(45)	0.86(45)
Electricity	**0.81**(**9**)	0.79(20)	0.69(2)	0.62(0)	0.76(9)	0.70(9)

LITERATUR

[1] M. Abdar u. a. *A Review of Uncertainty Quantification in Deep Learning: Techniques, Applications and Challenges*. 2021. arXiv: 2011.06225 [cs.LG].

[2] J. Antoran, J. Urquhart Allingham und J. M. Hernández-Lobato. *Depth Uncertainty in Neural Networks*. 2020. arXiv: 2006.08437 [stat.ML].

[3] M. Baena-García, J. Campo-Ávila, R. Fidalgo-Merino, A. Bifet, R. Gavald und R. Morales-Bueno. "Early Drift Detection Method". In: (Jan. 2006).

[4] L. Baier, T. Schlör, J. Schöffer und N. Kühl. *Detecting Concept Drift With Neural Network Model Uncertainty*. 2021. arXiv: 2107.01873 [cs.LG].

[5] A. Bifet. "Adaptive Learning and Mining for Data Streams and Frequent Patterns". In: 11.1 (2009). ISSN: 1931-0145. DOI: 10.1145/1656274.1656287. URL: https://doi.org/10.1145/1656274.1656287.

[6] A. Bifet. "Classifier Concept Drift Detection and the Illusion of Progress". In: Mai 2017, S. 715–725. ISBN: 978-3-319-59059-2. DOI: 10.1007/978-3-319-59060-8_64.

[7] A. Bifet und R. Gavaldà. "Learning from Time-Changing Data with Adaptive Windowing". In: Bd. 7. Apr. 2007. DOI: 10.1137/1.9781611972771.42.

[8] A. Bifet, J. Read, I. Zliobaite, B. Pfahringer und G. Holmes. "Pitfalls in Benchmarking Data Stream Classification and How to Avoid Them". English. In: *Proc. of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECMLPKDD'13)*. Bd. 8818. LNAI. Proceeding volume: 8818. International: Springer, 2013. ISBN: 978-3-642-40987-5. DOI: 10.1007/978-3-642-40988-2_30.

[9] François Chollet u. a. *Keras*. https://github.com/fchollet/keras. 2015.

[10] A. D. Cobb u. a. "An Ensemble of Bayesian Neural Networks for Exoplanetary Atmospheric Retrieval". In: *The Astronomical Journal* 158.1 (Juni 2019), S. 33. ISSN: 1538-3881. DOI: 10.3847/1538-3881/ab2390. URL: http://dx.doi.org/10.3847/1538-3881/ab2390.

[11] H. Cramér. *Mathematical Methods of Statistics (PMS-9)*. Princeton University Press, 2016. ISBN: 9781400883868. DOI: doi:10.1515/9781400883868. URL: https://doi.org/10.1515/9781400883868.

[12] S. De Vito, E. Massera, M. Piga, L. Martinotto und G. Di Francia. "On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario". In: *Sensors and Actuators B: Chemical* 129.2 (2008), S. 750–757. ISSN: 0925-4005. DOI: https://doi.org/10.1016/j.snb.2007.09.060. URL: https://www.sciencedirect.com/science/article/pii/S0925400507007691.

[13] R. Elwell und R. Polikar. "Incremental Learning of Concept Drift in Nonstationary Environments". In: *Neural Networks, IEEE Transactions on* 22 (Nov. 2011), S. 1517–1531. DOI: 10.1109/TNN.2011.2160459.

[14] H. Fanaee-T und J. Gama. "Event labeling combining ensemble detectors and background knowledge". In: *Progress in Artificial Intelligence* (2013), S. 1–15. ISSN: 2192-6352. DOI: 10.1007/s13748-013-0040-3.

[15] T. Fawcett und P. Flach. "A Response to Webb and Ting's On the Application of ROC Analysis to Predict Classification Performance Under Varying Class Distributions". In: *Machine Learning* 58 (Jan. 2005), S. 33–38. DOI: 10.1007/s10994-005-5256-4.

[16] Jerome H. Friedman. "Multivariate Adaptive Regression Splines". In: *The Annals of Statistics* 19.1 (1991), S. 1–67. ISSN: 00905364. URL: http://www.jstor.org/stable/2241837.

[17] Y. Gal und Z. Ghahramani. *Bayesian Convolutional Neural Networks with Bernoulli Approximate Variational Inference*. 2016. arXiv: 1506.02158 [stat.ML].

[18] Y. Gal und Z. Ghahramani. *Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning*. 2016. arXiv: 1506.02142 [stat.ML].

[19] J. Gama. *Knowledge Discovery from Data Streams*. 1st. Chapman und Hall/CRC, 2010. ISBN: 1439826110.

[20] J. Gama und M. Gaber. *Learning from data streams. Processing techniques in sensor networks*. Jan. 2007. DOI: 10.1007/3-540-73679-4.

[21] J. Gama, P. Medas, G. Castillo und P. Rodrigues. "Learning with Drift Detection". In: Bd. 8. Sep. 2004, S. 286–295. ISBN: 978-3-540-23237-7. DOI: 10.1007/978-3-540-28645-5_29.

[22] J. Gama, R. Sebastião und P. Rodrigues. "Issues in evaluation of stream learning algorithms". In: Jan. 2009, S. 329–338. DOI: 10.1145/1557019.1557060.

[23] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy und A. Bouchachia. "A survey on concept drift adaptation". In: *ACM Computing Surveys (CSUR)* 46 (2013), S. 1–37.

[24] J. Gast und S. Roth. "Lightweight Probabilistic Deep Networks". In: *CoRR* abs/1805.11327 (2018). arXiv: 1805.11327. URL: http://arxiv.org/abs/1805.11327.

[25] M. Grzenda, H. M. Gomes und A. Bifet. "Delayed labelling evaluation for data streams". In: *Data Mining and Knowledge Discovery* 34 (Sep. 2020). DOI: 10.1007/s10618-019-00654-y.

[26] D. Hall, F. Dayoub, J. Skinner, H. Zhang, D. Miller, P. Corke, G. Carneiro, A. Angelova und N. Sünderhauf. *Probabilistic Object Detection: Definition and Evaluation*. 2020. arXiv: 1811.10800 [cs.CV].

[27] M. Harries, C. Sammut, K. Horn und N. Wales. "Extracting Hidden Context". In: *Machine Learning* 32 (Mai 2003). DOI: 10.1023/A:1007420529897.

[28] M. Henne, A. Schwaiger, K. Roscher und G. Weiss. "Benchmarking Uncertainty Estimation Methods for Deep Learning With Safety-Related Metrics". In: *SafeAI@AAAI*. 2020.

[29] Martin Hensel. *KI-Nutzung in Unternehmen stagniert*. 2021. URL: https://www.bigdata-insider.de/ki-nutzung-in-unternehmen-stagniert-a-1071917/ (besucht am 10. 12. 2021).

[30] T. K. Ho. "Complexity of Classification Problems and Comparative Advantages of Combined Classifiers". In: *Multiple Classifier Systems*. Springer Berlin Heidelberg, 2000, S. 97–106. DOI: 10.1007/3-540-45014-9_9. URL: https://doi.org/10.1007/3-540-45014-9_9.

[31] S. B. K, N. Hochgeschwender, P. Plöger, F. Kirchner und M. Valdenegro-Toro. *Evaluating Uncertainty Estimation Methods on 3D Semantic Segmentation of Point Clouds*. 2020. arXiv: 2007.01787 [cs.CV].

[32] A. Kendall und Y. Gal. *What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?* 2017. arXiv: 1703.04977 [cs.CV].

[33] B. Krawczyk, L. Minku, J. Gama, J. Stefanowski und M. Woźniak. "Ensemble learning for data stream analysis: A survey". In: *Information Fusion* 37 (Sep. 2017), S. 132–156. DOI: 10.1016/j.inffus.2017.02.004.

[34] L. Kuncheva. "Classifier Ensembles for Changing Environments". In: Bd. 3077. Juni 2004, S. 1–15. ISBN: 978-3-540-22144-9. DOI: 10.1007/978-3-540-25966-4_1.

[35] B. Lakshminarayanan, A. Pritzel und C. Blundell. *Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles*. 2017. arXiv: 1612.01474 [stat.ML].

[36] P. Lindstrom, B. Mac Namee und S. Delany. "Drift Detection Using Uncertainty Distribution Divergence". In: Bd. 4. Dez. 2011, S. 604–608. DOI: 10.1109/ICDMW.2011.70.

[37] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama und G. Zhang. "Learning under Concept Drift: A Review". In: *IEEE Transactions on Knowledge and Data Engineering* (2018), S. 1–1. ISSN: 2326-3865. DOI: 10.1109/tkde.2018.2876857. URL: http://dx.doi.org/10.1109/TKDE.2018.2876857.

[38] M. A. Maloof und R. S. Michalski. In: *Machine Learning* 41.1 (2000), S. 27–52. DOI: 10.1023/a:1007661119649. URL: https://doi.org/10.1023/a:1007661119649.

[39] M. Markou und S. Singh. "Novelty detection: a review—part 1: statistical approaches". In: *Signal Processing* 83.12 (2003), S. 2481–2497. DOI: https://doi.org/10.1016/j.sigpro.2003.07.018. URL: https://www.sciencedirect.com/science/article/pii/S0165168403002020.

[40] D. K. McClish. "Analyzing a Portion of the ROC Curve". In: *Medical Decision Making* 9.3 (1989). PMID: 2668680, S. 190–195. DOI: 10.1177/0272989X8900900307.

[41] P. McClure und N. Kriegeskorte. "Representing inferential uncertainty in deep neural networks through sampling". In: *International Conference on Learning Representations*. Hrsg. von ICLR 2017-Conference Track Proceedings. 2016.

[42] J. Montiel, J. Read, A. Bifet und T. Abdessalem. "Scikit-Multiflow: A Multi-output Streaming Framework". In: *Journal of Machine Learning Research* 19.72 (2018), S. 1–5. URL: http://jmlr.org/papers/v19/18-251.html.

[43] J. Moreno-Torres, T. Raeder, R. Alaiz, N. Chawla und F. Herrera. "A unifying view on dataset shift in classification". In: *Pattern Recognition* 45 (Jan. 2012), S. 521–530. DOI: 10.1016/j.patcog.2011.06.019.

[44] S. A. Multaheb, B. Zimmering und O. Niggemann. "Expressing uncertainty in neural networks for production systems". In: *at - Automatisierungstechnik* 69.3 (2021), S. 221–230. DOI: doi:10.1515/auto-2020-0122. URL: https://doi.org/10.1515/auto-2020-0122.

[45] S. Muthukrishnan. "Data Streams: Algorithms and Applications". In: *Found. Trends Theor. Comput. Sci.* 1.2 (2005). DOI: 10.1561/0400000002. URL: https://doi.org/10.1561/0400000002.

[46] R. M. Neal. "Bayesian Learning for Neural Networks". PhD thesis. Diss. 1997.

[47] Neuo. *2021 MLOps Platforms Vendor Analysis Report*. 2021. URL: https://neu.ro/wp-content/themes/neuro/neuro/report-new.pdf (besucht am 10. 12. 2021).

[48] T. O'Malley, E. Bursztein, J. Long, F. Chollet, H. Jin, L. Invernizzi u. a. *KerasTuner*. https://github.com/keras-team/keras-tuner. 2019.

[49] E. S. Page. "Continuous Inspection Schemes". In: *Biometrika* 41.1/2 (1954), S. 100–115. ISSN: 00063444. URL: http://www.jstor.org/stable/2333009.

[50] T. Pearce, F. Leibfried, A. Brintrup, M. Zaki und A. Neely. *Uncertainty in Neural Networks: Approximately Bayesian Ensembling*. 2020. arXiv: 1810.05546 [stat.ML].

[51] R. Pop und P. Fulop. *Deep Ensemble Bayesian Active Learning : Addressing the Mode Collapse issue in Monte Carlo dropout via Ensembles*. 2018. arXiv: 1811.03897 [cs.LG].

[52] D. M. dos Reis, P. Flach, S. Matwin und G. Batista. "Fast Unsupervised Online Drift Detection Using Incremental Kolmogorov-Smirnov Test". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, S. 1545–1554. ISBN: 9781450342322. DOI: 10.1145/2939672.2939836. URL: https://doi.org/10.1145/2939672.2939836.

[53] M. Segù, A. Loquercio und D. Scaramuzza. "A General Framework for Uncertainty Estimation in Deep Learning". In: *CoRR* abs/1907.06890 (2019). arXiv: 1907.06890. URL: http://arxiv.org/abs/1907.06890.

[54] V. M. A. Souza, D. M. dos Reis, A. G. Maletzke und G. E. A. P. A. Batista. "Challenges in benchmarking stream learning algorithms with real-world data". In: *Data Mining and Knowledge Discovery* 34.6 (Juli 2020), S. 1805–1858. ISSN: 1573-756X. DOI: 10.1007/s10618-020-00698-5. URL: http://dx.doi.org/10.1007/s10618-020-00698-5.

[55] K. Stanley. "Learning Concept Drift with a Committee of Decision Trees". In: (Mai 2001).

[56] J. Stefanowski. "Adaptive Ensembles for Evolving Data Streams: Combining Block-Based and Online Solutions". In: Gewerbestrasse 11 CH-6330, Cham (ZG), CHE: Springer, 2015. ISBN: 139783319393148.

[57] J. Traub, P. M. Grulich, A. R. Cuellar, S. Breß, A. Katsifodimos, T. Rabl und V. Markl. "Scotty: Efficient Window Aggregation for out-of-order Stream Processing". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, S. 1300–1303.

[58] A. Tsymbal. "The Problem of Concept Drift: Definitions and Related Work". In: (Mai 2004).

[59] K. Tumer und J. Ghosh. "Analysis of decision boundaries in linearly combined neural classifiers". In: *Pattern Recognition* 29.2 (Feb. 1996), S. 341–348. DOI: 10.1016/0031-3203(95)00085-2. URL: https://doi.org/10.1016/0031-3203(95)00085-2.

[60] V. TV, Diksha, P. Malhotra, L. Vig und G. Shroff. *Data-driven Prognostics with Predictive Uncertainty Estimation using Ensemble of Deep Ordinal Regression Models*. 2021. arXiv: 1903.09795 [cs.LG].

[61] D. Vergara, S. Hernández, M. Valdenegro-Toro und F. Jorquera. *Improving Predictive Uncertainty Estimation using Dropout – Hamiltonian Monte Carlo*. 2019. arXiv: 1805.04756 [cs.LG].

[62] A. Wald. *Sequential analysis*. John Wiley, 1947.

[63] G. Wang, W. Li, M. Aertsen, J. Deprest, S. Ourselin und T. Vercauteren. "Aleatoric uncertainty estimation with test-time augmentation for medical image segmentation with convolutional neural networks". In: *Neurocomputing* 338 (Apr. 2019), S. 34–45.

ISSN: 0925-2312. DOI: 10.1016/j.neucom.2019.01.103. URL: http://dx.doi.org/10.1016/j.neucom.2019.01.103.

[64] Gerhard Widmer und Miroslav Kubat. "Learning in the presence of concept drift and hidden contexts". In: *Machine Learning* 23.1 (1996), S. 69–101. DOI: 10.1007/BF00116900. URL: https://doi.org/10.1007/BF00116900.

[65] PricewaterhouseCoopers GmbH Wirtschaftsprüfungsgesellschaft. *Künstliche Intelligenz in Unternehmen*. 2019. URL: https://www.pwc.de/de/digitale-transformation/kuenstliche-intelligenz/studie-kuenstliche-intelligenz-in-unternehmen.pdf (besucht am 10. 12. 2021).

[66] David H. Wolpert. "The Supervised Learning No-Free-Lunch Theorems". In: *Soft Computing and Industry: Recent Applications*. Hrsg. von R. Roy, M. Köppen, S. Ovaska, T. Furuhashi und F. Hoffmann. London: Springer London, 2002, S. 25–42. DOI: 10.1007/978-1-4471-0123-9{_}3. URL: https://doi.org/10.1007/978-1-4471-0123-9_3.

[67] S. Yu, X. Wang und J. C. Príncipe. "Request-and-Reverify: Hierarchical Hypothesis Testing for Concept Drift Detection with Expensive Labels". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence* (Juli 2018). DOI: 10.24963/ijcai.2018/421. URL: http://dx.doi.org/10.24963/ijcai.2018/421.

[68] G. Zenobi und P. Cunningham. "Using Diversity in Preparing Ensembles of Classifiers Based on Different Feature Subsets to Minimize Generalization Error". In: *ECML*. 2001.

[69] L. Zhu und N. Laptev. "Deep and Confident Prediction for Time Series at Uber". In: *2017 IEEE International Conference on Data Mining Workshops (ICDMW)* (Nov. 2017). DOI: 10.1109/icdmw.2017.19. URL: http://dx.doi.org/10.1109/ICDMW.2017.19.

[70] I. Žliobaitė. "Change with Delayed Labeling: When is it Detectable?" In: Jan. 2011, S. 843–850. DOI: 10.1109/ICDMW.2010.49.

[71] I. Žliobaitė. "Controlled permutations for testing adaptive learning models". In: *Knowledge and Information Systems* 39 (Juni 2013). DOI: 10.1007/s10115-013-0629-7.

[72] I. Žliobaitė, A. Bifet, J. Read, B. Pfahringer und G. Holmes. "Evaluation methods and decision theory for classification of streaming data with temporal dependence". In: *Machine Learning* 98.3 (2015), S. 455–482. DOI: 10.1007/s10994-014-5441-4.

[73] I. Žliobaitė, M. Pechenizkiy und J. Gama. "An Overview of Concept Drift Applications". In: 2016.