Verbesserung des Patching Algorithmus durch Adaptive Windowing Techniken

Improving Patching by Applying Adaptive Windowing Techniques
Bachelor-Thesis von Raphael Burbach aus Hanau
Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dipl.-Inform. Sebastian Kauschke

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

Verbesserung des Patching Algorithmus durch Adaptive Windowing Techniken
Improving Patching by Applying Adaptive Windowing Techniques

Vorgelegte Bachelor-Thesis von Raphael Burbach aus Hanau

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dipl.-Inform. Sebastian Kauschke

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 28.2.2018

_____

(Raphael Burbach)

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Verbesserung des Patching-Algorithmus, um den Klassifizierer an auftretende Veränderungen durch einen Concept Drift in den Daten schneller anpassen zu können. Ein Concept Drift ist eine signifikante Veränderung in der Verteilung oder Beschaffenheit der ankommenden Daten. Gewöhnliche Klassifizierer verlieren unter dem Einfluss eines Concept Drifts in den Daten an Performance. Daher muss bei Datenströmen, die einen Concept Drift enthalten bei diesen Klassifizierern ein Modell neu gelernt werden.

Bei Klassifizierungsproblemen mit einem kontinuierlichen Strom an Daten kann es zu einem Concept Drift kommen. Der Patching-Algorithmus löst dieses Problem, in dem das zuvor gelernte Modell bei Auftreten eines Concept Drifts nicht verworfen wird. Nach einem Concept Drift werden Regionen im Instance Space erkannt, in denen sich die Performance des Basisklassifizierers verschlechtert hat. Auf diesen Fehlerregionen wird ein neuer Klassifizierer trainiert. Der neue Klassifizierer wird mit dem Basisklassifizierer kombiniert, in dem der Basisklassifizierer nur die Instanzen klassifiziert, die nicht in den Fehlerregionen liegen. Alle anderen Instanzen werden durch den neu gelernten Klassifizierer klassifiziert, der Patch genannt wird. Dadurch können die Informationen aus dem ursprünglich gelernten Klassifizierer weiter genutzt werden. Zur Erkennung der Concept Drifts wird der ADWIN-Algorithmus verwendet.

Man kommt zu dem Ergebnis, dass die Klassifizierungsgenauigkeit des Patching-Algorithmus durch die oben vorgestellten Erweiterungen erhöht werden kann. Parallel dazu kann zudem häufig die Laufzeit reduziert werden.

Inhaltsverzeichnis

1 Einleitung

In der heutigen Zeit gibt es eine stetig wachsende Menge an Daten. Der Umfang der zu verarbeitenden Daten schließt in den meisten Fällen eine nicht maschinelle Verarbeitung aus. Maschinelles Lernen befasst sich mit der Verarbeitung von Daten. Das Hauptziel des maschinellen Lernens ist die Extraktion von Wissen. Eine häufige Anwendung des maschinellen Lernens sind Klassifizierungsprobleme. In der Realität kann sich das Konzept, dem die Daten folgen, ändern. Ein Beispiel dafür sind Änderungen in der Messmethodik zum Erfassen der Daten. Dies kann beispielsweise technische Ursachen wie der Austausch von Detektoren oder Veränderung in der Messelektronik haben. Oft hat die Erfassung von Daten auch eine menschliche Komponente, wie zum Beispiel Abstimmungen oder Bewertungen von Produkten. Personen ändern ihre Meinungen und Ansichten und damit ihr Konzept, indem sie Trends und dem Zeitgeist folgen. Solche Änderungen werden *Concept Drift* genannt.

Ein Concept Drift kann beim Klassifizieren von Datenströmen problematisch sein, wenn dieser nicht erkannt und behandelt wird. Oftmals existiert bereits ein Modell, welches in der Vergangenheit erstellt wurde. Findet nun ein Concept Drift in den ankommenden Daten statt, wird das Klassifizierungsergebnis des vorhandenen Klassifizierers signifikant schlechter. Der *Patching*-Algorithmus versucht, schon vorhandene Modelle wiederzuverwenden. Dabei wird nach einem Concept Drift nicht das gesamte Modell erneuert, sondern nur Bereiche des Instance Space, in denen die Klassifikationsgenauigkeit abnimmt. Ein solches Update wird *Patch* genannt.

Bisher hat der Patching-Algorithmus einige Limitierungen im Umgang mit einem Concept Drift. Der Patching-Algorithmus hat neben dem schon vorhandenen Klassifzierer, genannt Basisklassifizierer, einen *Instance Store*. Im Instance Store werden die aktuellsten Instanzen gespeichert. Die Instanzen aus dem Instance Store werden verwendet, um die Patches zu erstellen. Die maximale Größe des Instance Stores ist beim gewöhnlichen Patching-Algorithmus konstant. Die feste Größe des Instance Stores macht es allerdings schwierig, schnell auf die neuen Konzepte zu reagieren. Der Instance Store folgt dabei dem *sliding window* Prinzip. Wählt man die Fenstergröße groß, sind nach einem Concept Drift noch viele Instanzen, welche dem alten Konzept folgen, im Instance Store vorhanden. Diese Instanzen verschlechtern das Klassifizierungsergebnis. Wird die Größe des Instance Stores klein gewählt, kann schnell auf Concept Drifts reagiert werden. Allerdings wird eine geringere Klassifizierungsgenauigkeit erreicht.

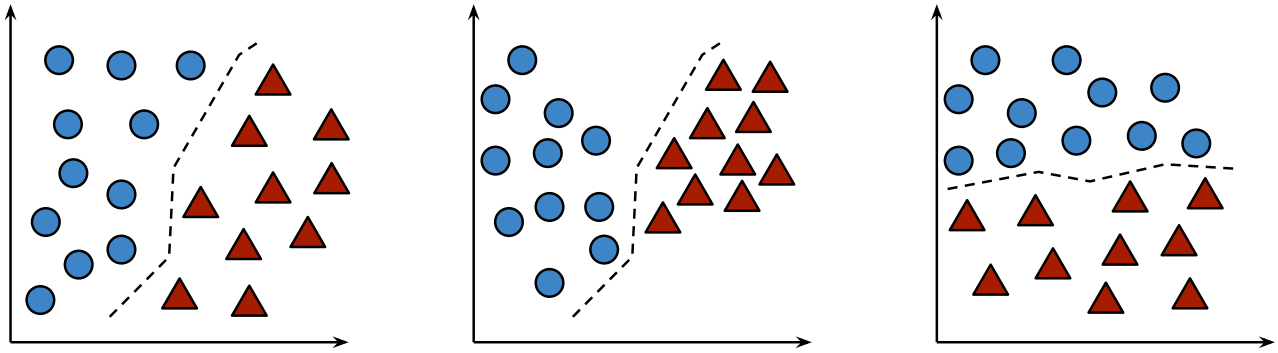
Da die verschiedenen Fenstergrößen ihre jeweiligen Vor- und Nachteile haben, ist die Motivation dieser Arbeit, die Fenstergröße dynamisch für die jeweiligen Situationen anpassen zu können. Ziel dieser Arbeit ist es, dass der Patching-Algorithmus durch adaptive windowing Techniken verbessert wird. Dadurch, dass die Fenstergröße des Lernfensters während der Ausführung dynamisch verändert werden kann, kann der Algorithmus schneller auf Veränderungen in den Daten reagieren. Somit kann bei der Erkennung eines Concept Drifts die Fenstergröße verringert werden, um Instanzen, welche dem alten Konzept folgen, aus dem Instance Store zu löschen. Die Erweiterungen des Patching-Algorithmus benutzen zur Erkennung eines Concept Drifts den ADWIN-Algorithmus. In der Evaluation werden die verschiedenen Implementierungen, die alle auf dem ursprünglichen Patching-Algorithmus aufbauen, untereinander und mit dem Patching-Algorithmus selbst verglichen.

2 Grundlagen

In diesem Kapitel werden die für diese Arbeit notwendigen Grundlagen vorgestellt. Dabei werden die Begriffe *Concept Drift* und *Transfer Learning* erklärt, sowie der Patching-Algorithmus erläutert. Anschließend wird kurz der ADWIN-Algorithmus vorgestellt, der für die Erweiterungen des Patching-Algorithmus genutzt wird. Schließlich werden noch die Erweiterungen des Patching-Algorithmus und die daraus resultierenden Unterschiede zum ursprünglichen Algorithmus erklärt.

2.1 Concept Drift

Unter einem *Concept Drift* versteht man eine nicht vorhersehbare signifikante Veränderung der Eigenschaften eines Datenstroms. Ein Concept wird dabei durch die Zuordnung der Instanzen zu den Klassen beschrieben. Formal definiert man die zu klassifizierende Klasse y und eine Menge von Attributbelegungen X. Bei einem *Concept Drift* kann sich die A-Priori-Wahrscheinlichkeit $\Pr(y)$ ändern. Weiterhin kann sich auch die bedingte Wahrscheinlichkeit $\Pr(X|y)$ für alle Klassen y ändern. Daher kann sich auch die A-Posteriori-Wahrscheinlichkeit der Klassen $\Pr(y|X)$ ändern, die sich über den Satz von Bayes aus $\Pr(y|X) = \frac{\Pr(y)\Pr(X|y)}{\Pr(X)}$ berechnen lässt. Man unterscheidet zwischen virtuellen und realen Concept Drifts. Bei einem virtuellen Concept Drift ändern sich die Wahrscheinlichkeiten der zugrunde liegenden Attribute. Es ändert sich also $\Pr(X)$, während $\Pr(y|X)$ gleich bleibt. Bei einem realen Concept Drift ändert sich die bedingte Wahrscheinlichkeit der Klassen bei vorausgesetzten Attributen. Es ändert sich also die Wahrscheinlichkeit $\Pr(y|X)$. Zusätzlich kann sich auch noch $\Pr(X)$ ändern[7]. Abbildung 1 zeigt den Vergleich zwischen einem virtuellen und einem realen Concept Drift. Dabei ist in Abbildung 1(b) der virtuelle Concept Drift zu erkennen, wobei sich nur $\Pr(X)$ ändert. Dadurch, dass $\Pr(y|X)$ gleich bleibt, wird die *class boundary* nicht verändert. Erst bei einem realen Concept Drift, wie er in Abbildung 1(c) dargestellt ist, ändert sich auch die *class boundary*. Der reale *Concept Drift* wird auch Class Drift genannt. Oftmals tritt ein virtueller Concept Drift allerdings nicht alleine auf, sondern es findet zusätzlich ein realer Concept Drift statt. Weiterhin unterscheidet man zwischen verschiedenen Unterkategorien von *Concept Drifts*.



(a) Original Daten

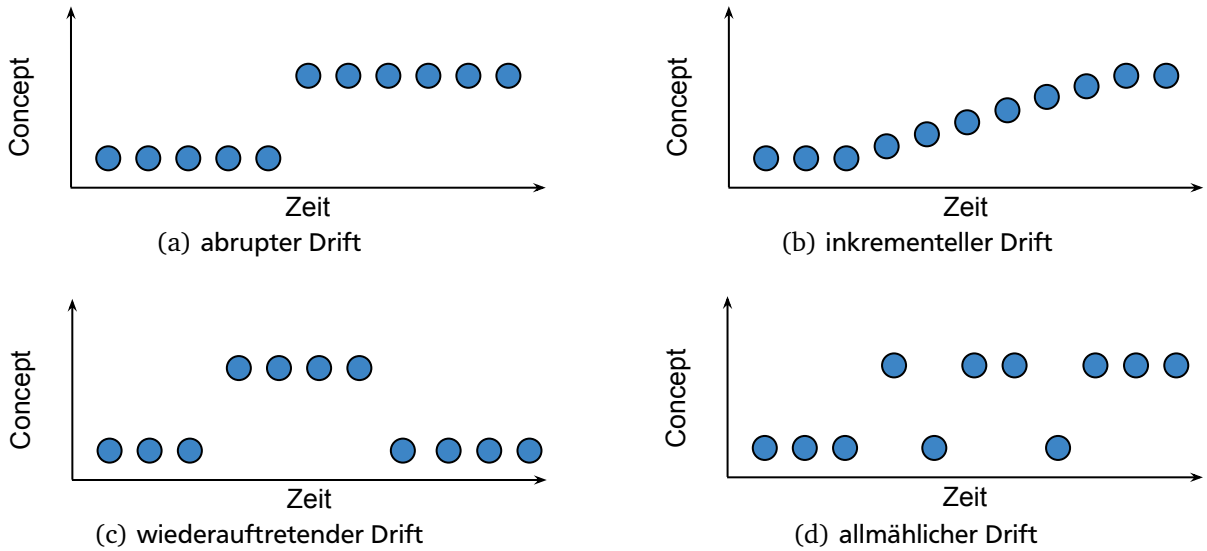
(b) virtueller Concept Drift $\Pr(X)$ ändert sich, $\Pr(y|X)$ bleibt gleich

(c) realer Concept Drift $\Pr(y|X)$ ändert sich

Abbildung 1: Concept Drift Typen: die Kreise und Dreiecke stellen die Instanzen dar, die Unterscheidung der Form und Farbe bildet die beiden Klassen ab, die schwarze Linie zeigt die *class boundary* (Quelle:[7], Seite 5)

Unter die Kategorie *Drift Subject* fallen beispielsweise die Typen *Class Drift* und *Novel Class appearance*. Bei einem Class Drift ändert sich, wie bereits oben erwähnt, die A-Posteriori-Wahrscheinlichkeit $\Pr(y|X)$ über die Zeit. Der Typ Novel Class appearance ist ein Spezialfall des Concept Drifts. Dabei kommen neue Klassen hinzu, die vorher nicht existent waren. $\Pr(y)$ ändert sich somit über die Zeit von $\Pr(y) = 0$, als die Klasse y noch nicht vorhanden war, zu $\Pr(y) > 0$, sobald die Klasse y erscheint.

Weiterhin unterscheidet man die *Drift Transitions* eines *Concept Drifts*. Die Kategorie *Drift Transition* beinhaltet unter anderem die Typen *Abrupt Drift*, *Incremental Drift*, *Reoccurring Drift* und *Gradual Drift*. Abbildung 2 zeigt die verschiedenen Drift Transitions.



(a) abrupter Drift (b) inkrementeller Drift

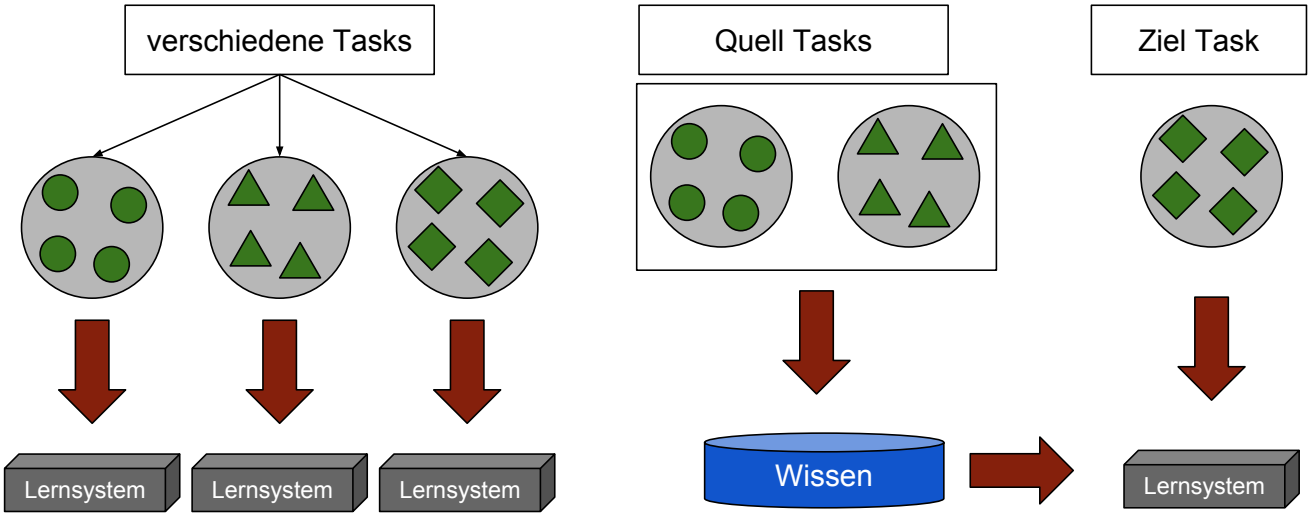
(c) wiederauftretender Drift (d) allmählicher Drift

Abbildung 2: Concept Transition Typen: die Kreise stellen die Instanzen dar, die minimalen und maximalen Werte auf der y-Achse bilden die alten bzw. neuen Concepts ab (Quelle:[7], Seite 6)

Ein abrupter Drift tritt auf, wenn sich das Concept der Daten instantan ändert. Dies ist beispielsweise im Datensatz $MNIST_{merge}$ (siehe Kapitel 5.1.1) der Fall. Die Klassen 4,5,7 der handgeschriebenen Ziffern ändern sich ab dem Change Punkt abrupt zur Klasse 9. Bei einem inkrementellen Drift werden mehrere zwischenliegende Concepts durchlaufen, bevor das finale Concept erreicht wird. Beim wiederauftretenden Drift ändert sich das aktuelle Concept zeitlich nur temporär, sodass nach einiger Zeit die Instanzen mit ihren Klassen wieder dem vorhergehenden Concept folgen. Beim allmählichen Drift pendeln die Instanzen zwischen dem alten und dem neuen Concept. Nach einigen Instanzen ändert sich das Concept jedoch hin zu dem neuen.[19]

2.2 Transfer Learning

Traditionelles maschinelles Lernen funktioniert, solange sich die Trainingsdaten und Testdaten nicht signifikant unterscheiden. Dazu zählt, dass die Wertebereiche der Attribute in den Trainings- und Testdaten gleich sind. Weiterhin muss auch die Verteilung der Attribute in den beiden Datensätze gleich sein. Ist dies nicht der Fall, muss ein Modell oftmals neu erstellt werden, was nur mit erheblichem Aufwand oder in bestimmten Situation gar nicht möglich ist. Um diesem Problem entgegenzuwirken, wird versucht, das bereits gewonnene Wissen der Task Domäne auf die neue Domäne anzuwenden. Dieses Anwenden von gesammeltem Wissen aus unterschiedlichen aber ähnlichen Problemdomänen wird als Transfer Learning bezeichnet. Es wird also das Wissen aus einer Domäne in eine neue Domäne transferiert. Der Machine Learning Algorithmus bekommt als Eingabe zusätzlich Informationen darüber, wie ein ähnliches Problem gelöst wurde[18]. In Abbildung 3 ist der Unterschied zwischen traditionellem maschinellen Lernen und Transfer Learning illustriert.



(a) traditionelles maschinelles Lernen (b) Transfer Learning

Abbildung 3: Unterschiede zwischen traditionellem maschinellen Lernen und Transfer Learning (Quelle:[12], Seite 2)

Beim traditionellen maschinellen Lernen wird für jede Aufgabe ein Lernsystem auf einem entsprechenden Trainingsdatensatz trainiert. Dies ist in Abbildung 3(a) dargestellt. Beim Transfer Learning wird das gelernte Wissen aus den Quell Tasks extrahiert und zusätzlich mit den Trainingsdaten als Eingabe für das Lernsystem verwendet (Abbildung 3(b)). Oftmals beinhaltet die vorhandene Wissensdatenbank deutlich mehr Instanzen, als für den Ziel Task Trainingsdaten zur Verfügung stehen. Somit kommt Transfer Learning auch dann zum Einsatz, wenn für einen Ziel Task nicht ausreichend viele Trainingsdaten zur Verfügung stehen, um ein Modell zu trainieren. Im Gegensatz zum Multi-Task Learning liegt beim Transfer Learning der Schwerpunkt auf dem Lernen des Ziel Tasks. Beim Multi-Task Learning liegt der Fokus darauf, alle Tasks gleichzeitig zu lernen[12]. Ein Beispieldatensatz für die Domäne Transfer Learning ist der *20 Newsgroups*-Datensatz (siehe Kapitel 5.1.2). Hierbei wird aus einer Unterkategorie die übergeordnete Kategorie bestimmt. Für die selbe übergeordnete Kategorie beinhalten die Trainings- und Testdatensätze allerdings unterschiedliche Unterkategorien. Somit muss der Algorithmus das Wissen der einen Unterkategorie zum Klassifizieren der übergeordneten Kategorie auf die andere Domäne, also die andere Unterkategorie, transferieren.

2.3 Patching-Algorithmus

Der *Patching*-Algorithmus ist ein Batch-Learner, der ein bereits vorhandenes gelerntes Modell an neue Daten anpassen kann. Die Grundidee des Algorithmus basiert darauf, dass ein Basisklassifizierer existiert, der auf einem bestehenden Datensatz trainiert wurde. Das so gelernte Modell kann allerdings aus den unterschiedlichsten Gründen nicht verändert werden. Zum einen kann es sein, dass das notwendige Wissen fehlt, wie das Modell in der Vergangenheit gelernt wurde, zum anderen, dass keine geeigneten Trainingsdaten zur Verfügung stehen, um das vorhandene Modell zu verändern. Weiterhin kann das Modell auch in Hardware abgebildet sein, und somit ist eine Veränderung nicht möglich. Um dennoch auf einen Concept Drift in den Daten reagieren zu können, erkennt der

Patching-Algorithmus die Regionen im Instance Space, die der unveränderliche Basisklassifizierer falsch klassifiziert hat. Dazu wird auf dem Instance Store ein Fehlerregionenklassifizierer trainiert, der die Fehlerregionen erkennen kann. Auf diesen Fehlerregionen werden nun neue so genannte Patchklassifizierer trainiert, die die Instanzen dieser Regionen besser als der Basisklassifizierer klassifizieren können. Das neu gelernte Modell wird als *Patch* bezeichnet. Die Klassifikation der Daten findet nun durch die Kombination des Basisklassifizierers und den neu gelernten Modellen statt. Dabei werden Instanzen, die in den Fehlerregionen liegen, von dem jeweiligen Patch klassifiziert. Instanzen, die nicht in den Fehlerregionen liegen, werden von dem Basisklassifizierer klassifiziert. Existieren mehrere Fehlerregionen, wird auch für jede Fehlerregion ein Patch gelernt. Der Fehlerregionenklassifizierer und somit auch der Patchklassifizierer werden mit jedem eintreffenden Batch neu erstellt.[9]

Formal existiert der Basisklassifizierer C_0 und der Instance Space D, der die Instanzen x mit den Labels $l(x)$ enthält. $l(x)$ gibt dabei die tatsächliche Klasse der Instanz x an. Die Wahrscheinlichkeit, dass der Basisklassifizierer C_0 die Instanzen aus D korrekt klassifiziert ($\Pr(C_0(x) = l(x))$), ist relativ hoch.

Für den Patching-Algorithmus müssen drei Klassifizierer als Eingabeparameter ausgewählt werden:

- der Basisklassifizierer (C_0 genannt)
- der Fehlerregionenklassifizierer (S genannt)
- der Patchklassifizierer (F_j genannt)

2.3.1 Fehlerregionen lernen

Erhält der Algorithmus einen neuen Batch D_i, wird ein neuer Klassifizierer auf den Instanzen des Batches trainiert, die in den Fehlerregionen von D_i liegen. Dabei wird eine neue Labelfunktion $L_{i,k}$ erstellt, die wie folgt definiert ist:

$$L_{i,k} = \begin{cases} 0, & \text{falls } C_0 \text{ diese Instanz falsch klassifiziert hat} \\ 1, & \text{falls } C_0 \text{ diese Instanz korrekt klassifiziert hat} \end{cases}, \; k \in 1, ..., |D_i|$$

Somit wird ein neuer Datensatz E_i erstellt, der die Instanzen aus D_i enthält. Die Labelfunktion von E_i ist allerdings die oben definierte Funktion $L_{i,k}$, sodass nur noch ein binäres Klassifizierungsproblem existiert. Auf diesem Datensatz E_i wird anschließend ein neuer Klassifizierer S trainiert. Dieser Klassifizierer S enthält implizit die Fehlerregionen R_j von E_i. R_j enthält also gerade genau die Instanzen, die der Basisklassifizierer C_0 auf dem Datensatz D_i potenziell falsch klassifiziert. Der Index j gibt dabei die Fehlerregion an, falls der Klassifizierer S mehrere Fehlerregionen zurückliefert.

2.3.2 Patches lernen

Die Patches werden nun auf den Instanzen gelernt, die durch den Fehlerregionenklassifizierer erkannt wurden. Die Trainingsdaten der Patchklassifizierer bestehen also jeweils aus den Instanzen, die der Basisklassifizierer falsch klassifiziert hat. Formal beschrieben, wird für jede Fehlerregion R_j ein neuer Klassifizierer F_j auf den Instanzen von D_i trainiert, für die gilt $S(I_i) = 1$. F_j ist also der Patch für die Fehlerregion R_j und klassifiziert diese nun besser. Die Instanzen von D_i wurden vorher dem Instance Store hinzugefügt, sodass das Trainieren des Fehlerregionen- und des Patchklassifizierers auf dem Instance Store stattfindet.

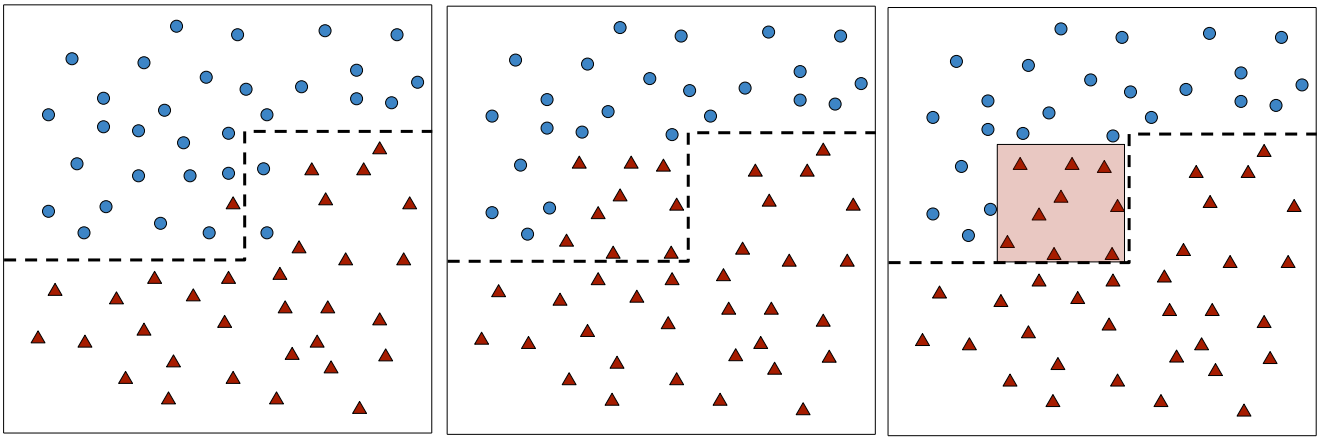
2.3.3 Klassifizierung

Der Algorithmus 1 klassifiziert beispielhaft die Instanz x. Wenn die Instanz in einer Fehlerregion R_j liegt, wird das Ergebnis der Klassifizierung des entsprechenden Patches F_j zurückgeliefert. Liegt die Instanz in keiner der Fehlerregionen, wird das Ergebnis der Klassifizierung des Basisklassifizierers zurückgeliefert.

Algorithm 1 Klassifizierung

```
1: for each R_j do
2:     if R_j(x) = 1 then
3:         return F_j(x)
4: return C_0(x)
```

Abbildung 4 zeigt den Ablauf des Klassifizierungsvorgangs mit dem Patching-Algorithmus. In Abbildung 4(a) wird angenommen, dass bereits ein Basisklassifizierer existiert und dieser auf den vorhandenen Trainingsdaten trainiert wurde. Dabei stellen die blauen Punkte und roten Dreiecke Instanzen dar. Das gesamte Schaubild soll den Instance Store abbilden. Eine Instanz besteht in diesem vereinfachten Beispiel aus zwei Attributen, daher wird die Abbildung als ein zweidimensionaler Raum dargestellt. Die schwarze gestrichelte Linie zeigt die *decision boundary* des Basisklassifizierers an. Da die Klassifizierungsgenauigkeit relativ hoch ist, sind nur wenige Instanzen falsch klassifiziert. Es liegen also jeweils wenige Instanzen jenseits der *decision boundary* des Basisklassifizierers. In Abbildung 4(b) hat sich die Verteilung der Attribute im Instance Store geändert, beispielsweise durch einen Concept Drift. Es werden jetzt deutlich mehr Instanzen falsch klassifiziert, da die *decision boundary* des Basisklassifizierers nicht angepasst werden kann, weil dieser nicht veränderbar ist. Um nun doch auf den Concept Drift reagieren zu können, wird mithilfe des Fehlerregionenklassifizierers die Fehlerregion erkannt. Diese ist in Abbildung 4(c) durch das rote Rechteck dargestellt. Auf den Instanzen, die sich in der Fehlerregion befinden, wird nun der Patchklassifizierer trainiert.[9]



(a) Basisklassifizierer (b) Verteilung der Attribute ändert sich (c) Fehelerregion erkennen und Patch lernen

Abbildung 4: Klassifizierung mit dem Patching-Algorithmus

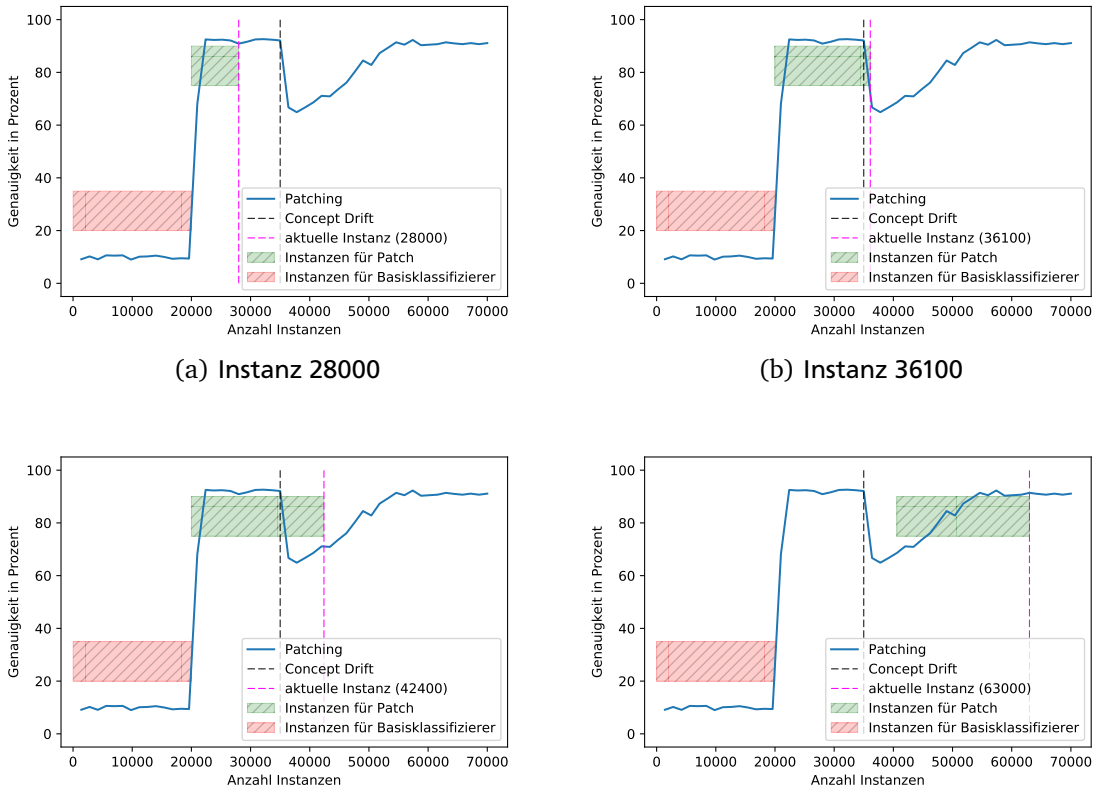
2.3.4 Batches

Die korrekten Labels der Instanzen $l(x)$ sind bei dem Datenstrom nicht sofort verfügbar und somit kann das Klassifizierungsergebnis nicht sofort evaluiert werden. Daher wird angenommen, dass die korrekten Labels $l(x)$ zu einem späteren Zeitpunkt verfügbar sind. Die Instanzen des Datenstroms werden zu Batches zusammengefasst. Um die für das Trainieren des Patchklassifizierers notwendigen Instanzen zu speichern, besitzt der *Patching*-Algorithmus einen so genannten Instance Store. Der Instance Store ist eine Liste, die das First-In-First-Out Verfahren verfolgt. Die Größe des Instance Stores k, also die Anzahl der Batches, die gespeichert werden sollen, kann durch einen Eingabeparameter angegeben werden. Für die Ausführung des Algorithmus ist k allerdings konstant.

Wird für k ein großer Wert gewählt, ist das Klassifizierungsergebnis nach einem *Concept Drift* sehr schlecht. Die vielen gespeicherten Instanzen aus dem Instance Store verschlechtern die Klassifizierung, da sich der Algorithmus nicht so schnell anpassen kann. Der Klassifizierer nutzt zum Klassifizieren bei einem großen k direkt nach dem *Concept Drift* noch viele Instanzen, die vor dem Concept Drift korrekt klassifiziert wurden. Diese Instanzen folgen allerdings dem alten Concept und sind für die Klassifizierung nach dem Concept Drift womöglich nicht mehr brauchbar. Abbildung 5 illustriert einen Lernvorgang des Patching-Algorithmus und zeigt dabei den Zweck des Instance Stores. In diesem Beispiel wurde für k ein großer Wert gewählt und somit werden viele Instanzen im Instance Store gespeichert. Der Instance Store ist in den Abbildungen 5(a)-(d) jeweils durch das grüne schraffierte Rechteck dargestellt. Auf den ersten 20000 Instanzen wurde der Basisklassifizierer trainiert. Bei diesem Trainingsvorgang stehen keine Evaluationsdaten zum Einzeichnen bereit, sodass hier die Baseline dargestellt wird. Die korrekte Klassifizierungsgenauigkeit des Patching-Algorithmus ist somit erst nach Instanz 20000 abzulesen. Das rote schraffierte Rechteck stellt die Instanzen dar, die dem Basisklassifizierer zum Trainieren zur Verfügung stehen. Dieses Fenster ist fix und unveränderlich, genau wie der Basisklassifizierer an sich. In Abbildung 5(a) ist die Klassifizierungsgenauigkeit hoch. Der Klassifizierungsfortschritt befindet sich hier an der Instanz 28000. Die gerade klassifizierte Instanz

befindet sich also immer am rechten Rand des grünen Rechtecks. Da der Instance Store seine maximale Größe noch nicht erreicht hat, werden mit jedem ankommenden Batch weitere Instanzen im Instance Store gespeichert. Abbildung 5(b) zeigt die Klassifizierung direkt nach dem Concept Drift, der bei Instanz 35000 stattfindet. Auch hier ist die maximale Größe des Instance Stores noch nicht erreicht, und das Fenster zum Klassifizieren wächst weiter in die Breite. Die Klassifizierungsgenauigkeit ist in dieser Abbildung deutlich kleiner als vor dem Concept Drift. Der Grund hierfür ist, dass der Instance Store zu diesem Zeitpunkt 16100 Instanzen beinhaltet, davon allerdings 15000 die dem alten Concept folgen und nur 1100 des neuen Concepts. Somit verringern die Instanzen des vorherigen Concepts die Klassifizierungsgenauigkeit, da diese weiterhin zum Trainieren des Klassifizierers genutzt werden. In Abbildung 5(c) an der Instanz 42400 hat der Instance Store gerade seine maximale Größe von 22400 gespeicherten Instanzen erreicht. Somit wächst das grüne Rechteck zwischen Abbildung 5(c) und Abbildung 5(d) nicht mehr weiter in die Breite. Erst wenn der Instance Store seine maximale Größe erreicht hat, erkennt man den Effekt des sliding window wie es sich über die Instanzen bewegt. Weiterhin erkennt man in Abbildung 5(c), dass die Genauigkeit des Klassifizierers aus demselben Grund wie in Abbildung 5(b) gering ist. Erst wenn die Anzahl der Instanzen die Größe des Instance Stores überschritten hat, wie in Abbildung 5(d) zu sehen ist, kann die Genauigkeit wieder signifikant besser werden. Die Genauigkeit des Klassifizierers nimmt nach dem Concept Drift mit der Anzahl der Instanzen des neuen Concepts im Instance Stores zu. Beim Klassifizieren der Instanz 63000 erkennt man, dass im Instance Store nur noch Instanzen vorhanden sind, die dem neuen Concept folgen. Dies ist ersichtlich, da das grüne Rechteck die Linie, die den Concept Drift anzeigt, nicht mehr schneidet. Ab diesem Zeitpunkt hat die Genauigkeit ihr Maximum erreicht und erhöht sich nicht mehr signifikant.

Tritt kein *Concept Drift* auf, hilft eine höhere Anzahl gespeicherter Batches potenziell dabei, dass sich die Genauigkeit erhöht.



(a) Instanz 28000

(b) Instanz 36100

(c) Instanz 42400

(d) Instanz 63000

Abbildung 5: Einfluss der Größe des Instance Stores auf die Klassifizierungsgenauigkeit

Wird k klein gewählt, kann der Algorithmus zwar auf einen *Concept Drift* reagieren und sich schnell anpassen. Dabei erzielt er aber potenziell eine geringere Genauigkeit wie der Klassifizierer, der mehr Batches abgespeichert hat.

2.4 ADWIN-Algorithmus

Der *ADWIN*-Algorithmus (*AD*aptive *WIN*dowing Algorithmus)[2] ist ein Algorithmus, der *Concept Drifts* in einem Datenstrom erkennen kann. Dabei ist er bezüglich der Speichernutzung und der Rechenzeit besonders effizient. Die Grundidee des Algorithmus basiert dabei darauf, dass ein *sliding window* über die Daten gelegt wird. Dieser Ansatz wird auch bei anderen Algorithmen verfolgt. Der Unterschied ist allerdings, dass die Größe dieses *sliding windows* nicht fest ist, sondern online anhand der ankommenden Daten berechnet wird. Wird ein *Concept Drift* erkannt, kann also schnell darauf reagiert werden, indem der Algorithmus die Größe des Fensters verkleinert. Das sliding window ist dabei durch eine Liste abgebildet. Diese Liste verfolgt das First-In-First-Out Prinzip.

Der *ADWIN*-Algorithmus erhält als Eingabe einen potenziell unendlichen Datenstrom mit Werten $x_t \in [0,1]$ und einen Schwellenwert $\delta \in (0,1)$.

2.4.1 Algorithmus

Der *ADWIN*-Algorithmus speichert die Eingaben x_i in einem *sliding window* W. Dabei gibt n die Länge des *sliding window* an. Weiterhin existiert $\hat{\mu}_W$, das den bekannten Durchschnitt über alle Elemente in W und μ_W, das den unbekannten Durchschnittswert von μ_t für $t \in W$ enthält.

Der Algorithmus benötigt weiterhin den Wert ϵ_{cut}. Dieser setzt sich wie folgt aus den Werten m und δ' zusammen:

$$m = \frac{2}{1/n_0 + 1/n_1} \tag{1}$$

In der Formel (1) stellt m den harmonischen Mittelwert der Größe der Fenster W_0 und W_1, also von n_0 und n_1 dar. In Formel (2) wird das gegebene δ durch die Bonferroni-Methode korrigiert, da δ durch mehrere Hypothesentests verfälscht würde. Daher wird ϵ_{cut} mit dem korrigierten δ' berechnet.

$$\delta' = \frac{\delta}{n} \tag{2}$$

und daraus folgt der Wert ϵ_{cut}

$$\epsilon_{cut} = \sqrt{\frac{2}{m} \cdot \sigma^2{}_W \cdot \ln \frac{2}{\delta'}} + \frac{2}{3m} \cdot \ln \frac{2}{\delta'} \tag{3}$$

Der Schwellenwert ϵ_{cut} wird für mehrere Zerlegungen von W in $W_0 \cdot W_1$ berechnet. $W_0 \cdot W_1$ steht hier für die Konkatenation der beiden Fenster W_0 und W_1 und soll die Aufteilung von W abbilden. Diese Zerlegungen werden durch Histogramme der enthaltenen Einsen in W berechnet. Die Werte n_0 und n_1 sind dabei die Längen von W_0 und W_1 und n ist die Länge von W. Es gilt also $n = n_0 + n_1$. Benötigt werden noch $\hat{\mu}_{W_0}$ und $\hat{\mu}_{W_1}$, welche jeweils die Mittelwerte von W_0 und W_1 angeben. Der Wert von $\sigma^2{}_W$ ist die Varianz aller Werte in W.

Algorithm 2 Adaptive Windowing Algorithm[2]

```
1: Initialize W as an empty list of buckets
2: Initialize WIDTH, VARIANCE and TOTAL
3: for each t > 0 do
4:     setInput(x_t, W)
5: return μ̂_w as TOTAL/WIDTH and ChangeAlarm

1: procedure setInput(item e, List W)
2:     insertElement(e, W)
3:     repeat deleteElement(W)
4:     until |μ̂w_0 − μ̂w_1| < ε_cut holds
5:     for every split of W into W = W_0 · W_1
```

```
1: procedure INSERTELEMENT(item e, List W)
2:     create a new bucket b with content e and capacity 1
3:     W ← W ∪ {b} (add e to the head of W)
4:     update WIDTH, VARIANCE and TOTAL
5:     COMPRESSBUCKETS(W)

1: procedure DELETEELEMENT(List W)
2:     remove a bucket from tail of List W
3:     update WIDTH, VARIANCE and TOTAL
4:     ChangeAlarm ← true

1: procedure COMPRESSBUCKETS(List W)
2:     traverse the list of buckets in increasing order
3:     do there are more than M buckets of the same capacity
4:         do merge buckets
```

In Algorithmus 2 wird der ADWIN-Algorithmus veranschaulicht. Der Algorithmus verkleinert also die Liste W, wenn die Differenz der Mittelwerte zweier Zerlegungen W_0 und W_1 von W größer als der zugehörige Schwellenwert ϵ_{cut} ist. Dabei werden jeweils die ältesten Elemente aus W entfernt.

3 Algorithmen zur Klassifizierung

Die folgenden Algorithmen basieren auf dem *Patching*-Algorithmus, wie er in Kapitel 2.3 beschrieben wurde. Die Implementierungen erweitern den Patching-Algorithmus um adaptive windowing Techniken, damit die Genauigkeit des Klassifizierers verbessert wird. Somit kann die Größe des Instance Stores während der Ausführung dynamisch verändert werden, um so bestmöglich auf die Veränderung in den Daten reagieren zu können. Die beiden folgenden Algorithmen erkennen allerdings nicht den *Concept Drift* im Datensatz. Vielmehr erkennen die Algorithmen die Veränderung in den Datensätzen dadurch, dass die Genauigkeit der Klassifizierer signifikant schlechter wird.

3.1 AdaptivePatchingADWIN

Der *AdaptivePatchingADWIN*-Algorithmus kann die Veränderungen in der Genauigkeit der Klassifizierungen erkennen. Da die Genauigkeit bei einem Concept Drift signifikant abnimmt, wird somit indirekt der Concept Drift im Datenstrom erkannt. Zur Erkennung des Concept Drifts wird eine Implementierung des *ADWIN*-Algorithmus (Kapitel 2.4) aus *WEKA*[8] genutzt. Nach jedem Batch, den der *Patching*-Algorithmus klassifiziert hat, wird für jede Instanz geprüft, ob der Patching-Algorithmus die Instanz korrekt klassifiziert hat. Ist dies der Fall, wird eine '0' an das Ende der Liste des ADWIN-Algorithmus eingefügt. Hat der Patching-Algorithmus eine Instanz falsch klassifiziert, wird eine '1' an das Ende der Liste des ADWIN-Algorithmus eingefügt. Wird eine Veränderung in den 'Nullen' und 'Einsen' durch den ADWIN-Algorithmus erkannt, werden automatisch so viele alte Elemente, also Elemente vom Anfang der Liste, entfernt, bis keine signifikante Veränderung mehr in den Elementen erkannt wird. Die Größe des Instance Stores wird auf die Größe der Liste des ADWIN-Algorithmus gesetzt. Dadurch sind nach einem Concept Drift weniger beziehungsweise keine Trainingsdaten des vorhergehenden Concepts im Instance Store vorhanden, sodass die Genauigkeit des Klassifizierers wieder schnell ansteigen kann. Der Fehlerregionenklassifizierer und der Patchklassifizierer werden somit nur noch auf Instanzen trainiert, die dem aktuellen Concept folgen.

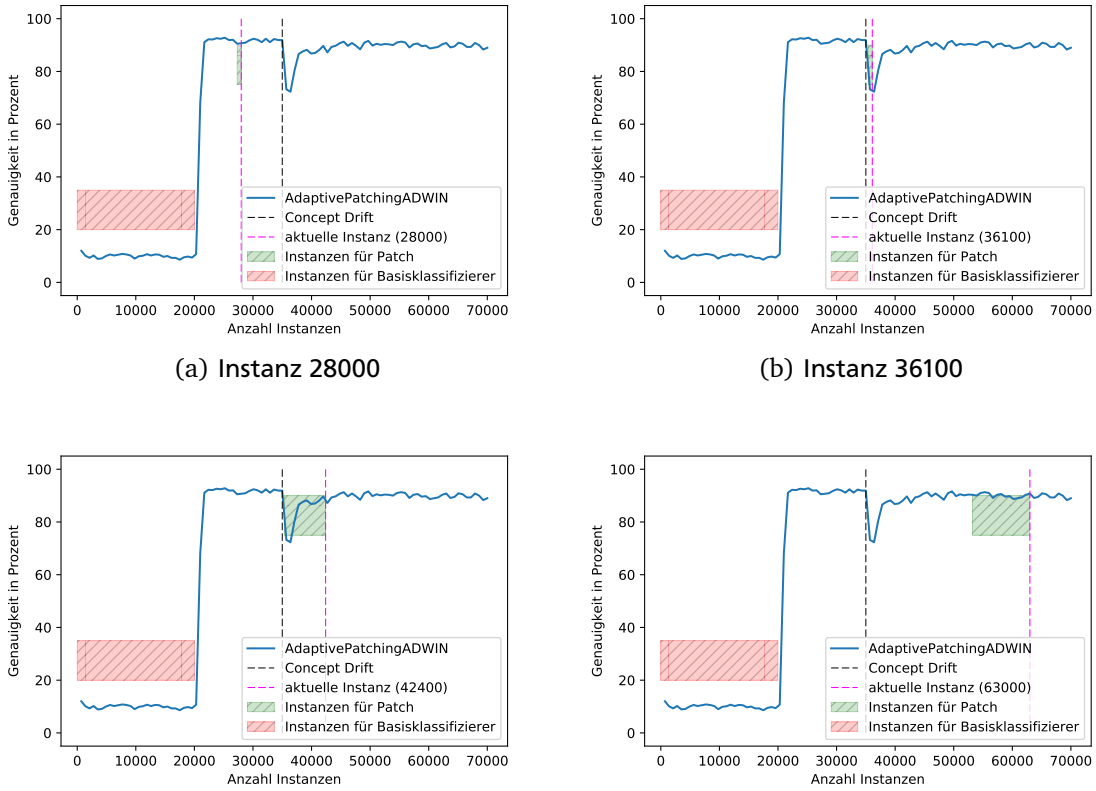
Der AdaptivePatchingADWIN-Algorithmus kann in zwei unterschiedlichen Konfigurationen ausgeführt werden. Standardmäßig wächst das tatsächliche Fenster zum Lernen nur dann, wenn es eine signifikante Änderung in der Varianz der Anzahl der korrekt klassifizierten Instanzen gibt. Somit ist die Anpassung der Größe des Instance Stores restriktiver, und der Instance Store ist nur maximal so groß wie die Liste des ADWIN-Algorithmus. Wird keine Änderung in den Daten festgestellt, wächst die Liste des ADWIN-Algorithmus an. Die Größe des Instance Stores wächst jedoch nur, wenn die Differenz der Varianzen der aktuell korrekt klassifizierten Instanzen und der Anzahl der korrekt klassifizierten Instanzen des letzten Batches einen zuvor definierten Wert überschreitet. Somit ist gewährleistet, dass der Instance Store nicht bei jedem neuen Batch anwächst, sondern nur bei signifikanten Änderungen. Dadurch wird erheblich Rechenzeit beim Lernen eingespart, da den Klassifizierern eine geringere Anzahl an Trainingsdaten zur Verfügung stehen. Da sich die Genauigkeit der Klassifizierung nicht signifikant geändert hat, müssen auch keine neuen Instanzen zum Lernen gespeichert werden. Ändert sich die Genauigkeit des Klassifizierers allerdings merklich, ohne dass der ADWIN-Algorithmus eine Veränderung erkennt, wird der Instance Store durch das Varianzkriterium trotzdem vergrößert.

Formal ist das Varianzkriterium wie folgt definiert. Die Varianz der letzten Klassifizierungsgenauigkeiten beim Eintreffen des vorherigen Batches D_{i-1} wird als σ_{i-1} bezeichnet. Die Varianz der letzten Klassifizierungsgenauigkeiten beim Eintreffen des aktuellen Batches D_i wird als σ_i definiert. Die Variable i stellt den Index des aktuellen Batches dar. Der Algorithmus 3 entscheidet anhand des Eingabeparameters $\epsilon_{variance}$, ob die Größe des Instance Stores vergrößert wird. Wird ein Concept Drift erkannt, wird die Liste der Klassifizierungsgenauigkeiten geleert.

Algorithm 3 Varianzkriterium

```
1:  initialisiere Liste Accuracy
2:  ε_variance setzen
3:  for each t > 0 do
4:      if change erkannt then
5:          Accuracy.clear
6:      berechne Varianz σ_{i-1} von Accuracy
7:      Accuracy_i := erreichte Genauigkeit auf dem aktuellen Batch
8:      Accuracy := Accuracy ∪ Accuracy_i
9:      berechne Varianz σ_i von Accuracy
10:     if i > 1 and |σ_{i-1} − σ_i| > ε_variance then
11:         InstanceStore.size := ADWIN.size
```

Wie bereits in Abbildung 5 anhand des Patching-Algorithmus gezeigt, soll nun auch in Abbildung 6 der Instance Store mit der ADWIN-Erweiterung und einer dynamischen Größe erklärt werden. Auf den ersten 20000 Instanzen wird, wie beim Patching-Algorithmus auch, der Basisklassifizierer trainiert. Beim Trainingsvorgang stehen keine Evaluationsdaten zur Verfügung, sodass als Genauigkeit lediglich die Baseline eingezeichnet werden kann. Die Instanzen, die dem Basisklassifizierer zur Verfügung stehen, sind durch das rote schraffierte Rechteck dargestellt. Ab Instanz 20000 wird die korrekte Klassifizierungsgenauigkeit angezeigt. In Abbildung 6(a) ist der aktuelle Klassifizierungsfortschritt bei Instanz 28000. Auch hier wurde, wie beim Patching-Algorithmus in Abbildung 5 auch, für k ein großer Wert gewählt. Dieser dient beim AdaptivePatchingADWIN-Algorithmus jedoch nur als obere Schranke für die Größe des Instance Stores. Im Unterschied zum Patching-Algorithmus werden erst bei Erkennen eines Concept Drift dem Instance Store Instanzen hinzugefügt. Daher ist die Größe des Instance Stores in Abbildung 6(a) sehr gering und das grüne Rechteck hat demnach eine kleine Breite. Die Instanz, die gerade evaluiert wird, befindet sich immer am rechten Rand des grünen Rechtecks. Die Breite des Rechtecks nach links deckt also nur zeitlich frühere Instanzen ab. Im Gegensatz zum Patching-Algorithmus kann man beim AdaptivePatchingADWIN-Algorithmus die sliding window Eigenschaften des Instance Stores schon zu Beginn erkennen. Im Instance Store befinden sich immer die aktuellsten Instanzen. Die Größe des Instance Stores legt fest, wie viele Instanzen gespeichert werden. Wenn die Größe des Instance Stores konstant ist, werden nach dem First-In-First-Out Prinzip die ältesten Instanzen verworfen um aktuelle Instanzen speichern zu können. In Abbildung 6(c) erkennt man, dass der Instance Store erst nach Auftreten des Concept Drifts bei Instanz 35000 anfängt, mehrere Instanzen zu speichern. Somit kann sich die Klassifizierungsgenauigkeit schneller erholen als beim Patching-Algorithmus. Auch in Abbildung 6(d) erkennt man, dass trotz des groß gewählten Wertes für k der Instance Store kleiner ist als beim Patching-Algorithmus. Hierbei hemmt das Varianzkriterium, wie bereits oben erwähnt, das Wachstum des Instance Stores, sodass nur dann die Größe des Instance Stores erhöht wird, wenn eine höhere Anzahl an Instanzen signifikante Verbesserungen in der Klassifizierungsgenauigkeit erzielen.



(a) Instanz 28000

(b) Instanz 36100

(c) Instanz 42400

(d) Instanz 63000

Abbildung 6: Einfluss der Größe des Instance Stores auf die Klassifizierungsgenauigkeit

In Tabelle 1 wurden die konkreten Werte aus den Abbildungen 5 und 6 dargestellt. Die Batch Größe wurde in diesem Beispiel auf 700 Instanzen gesetzt. Für die Größe des Instance Stores wurde $k = 32$ gewählt. Dies entspricht einer maximalen Anzahl von 22400 Instanzen, die im Instance Store gespeichert werden. Im AdaptivePatchingADWIN-Algorithmus ist diese Zahl jedoch, wie bereits erwähnt, nur als obere Schranke zu sehen.

Man erkennt hier, dass der Instance Store beim Patching-Algorithmus bereits ab der Instanz 20000, also nach der Beendigung des Trainierens des Basisklassifizierers, anfängt, Instanzen zu speichern und somit wächst. Bereits bei Instanz 35000, also an dem Punkt, an dem der Concept Drift stattfindet, beinhaltet er schon 15400 Instanzen. Der Instance Store des AdaptivePatchingADWIN-Algorithmus hat nur seine initiale Größe von einem Batch. Zu beachten ist bei dieser Tabelle, dass sich die Skala der Instanzen nach dem Concept Drift ändert, um detailliertere Werte zu veranschaulichen. Zu Beginn und am Ende beträgt ein Schritt in der Tabelle zehn Batches. Im gelb markierten Bereich, direkt nach dem Concept Drift, beträgt ein Schritt nur noch zwei Batches. Instanz 35000 ist durch **CP** als Change Punkt markiert. Bei Instanz 42000 hat der Patching-Algorithmus bereits die maximale Größe des Instance Stores von 32 Batches erreicht. Somit werden neue Batches eingefügt und dafür alte Batches entfernt, um immer eine konstante Anzahl an k Batches im Instance Store zu behalten. Der AdaptivePatchinADWIN-Algorithmus erkennt ab Instanz 49000, dass es keine signifikante Verbesserung mehr in den Daten gibt, und somit auch keine höhere Anzahl von Instanzen im Instance Store gespeichert werden muss. Somit bleibt die Größe des Instance Stores hier bei 14 Batches fest, und es werden ebenfalls bei Eintreffen neuer Batches die ältesten aus dem Instance Store verworfen, um die neuen Batches zu speichern. Die Größe des Instance Stores bleibt also auch hier konstant.

Instanzen	**Größe Instance Store *Patching***	**Größe Instance Store *AdaptivePatchingADWIN***
21000	1400	700
28000	8400	700
35000 (CP)	15400	700
36400	16800	1400
37800	18200	2800
39200	19600	4200
40600	21000	5600
42000	22400	7000
49000	22400	9800
56000	22400	9800
63000	22400	9800
70000	22400	9800

Tabelle 1: Vergleich Instance Store Größe

In Abbildung 7 ist die Größe des Instance Stores jeweils mit und ohne Concept Drift Erkennung durch ADWIN illustriert. Der in dieser Abbildung verwendete Datensatz beinhaltet drei Change Punkte. Den ersten bei Instanz 35000, den zweiten bei Instanz 45000 und schließlich eine Veränderung an Instanz 55000. Man erkennt hier sehr gut, wie der Instance Store mithilfe des ADWIN-Algorithmus auf die Veränderungen reagiert und somit Instanzen, die dem vorherigen Konzept folgen, aus dem Instance Store entfernt.

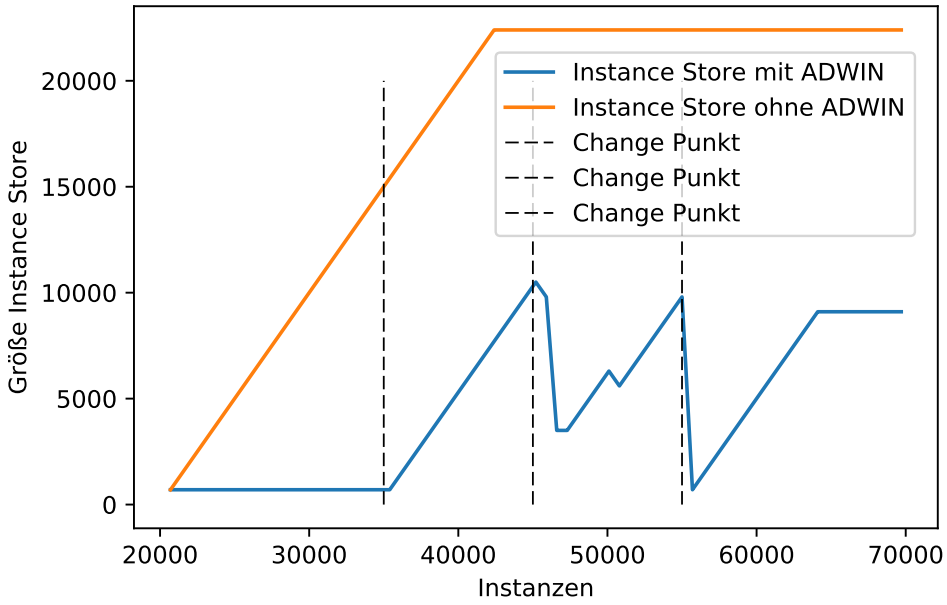


Abbildung 7: Vergleich Instance Store Größe

Im zweiten Modus wächst der Instance Store identisch zu der Liste des *ADWIN*-Algorithmus. Dabei ist er in der Größe nach oben nicht beschränkt. Dies führt zu besseren Ergebnissen in der Genauigkeit der Klassifizierung. Die vielen Instanzen im Instance Store gehen allerdings sehr zu Lasten der Leistung (vorwiegend Rechenzeit auf dem Prozessor). In der Evaluation wird der Algorithmus daher *AdaptivePatchingADWINNoLimit* genannt.

3.2 AdaptivePatchingTwoADWINs

Ein weiterer Ansatz war, dass man zwei Instanzen des *ADWIN*-Algorithmus parallel Veränderungen erkennen lässt. Eine Instanz des *ADWIN*-Algorithmus wird dabei durch den Parameter δ sehr empfindlich eingestellt. Somit werden auch kleine Änderungen in den Daten erkannt. Erkennt der empfindliche *ADWIN*-Algorithmus keine Veränderung, wächst die Größe des Instance Stores an. Sobald jedoch eine Änderung erkannt wird, wird die Größe des Instance Stores verringert. Die Größe des Instance Stores wird hierbei jedoch nur schrittweise angepasst im Gegensatz zu einer echten Concept Drift Erkennung, bei der der Instance Store abrupt verkleinert wird. Die zweite Instanz des ADWIN-Algorithmus ist über den Parameter δ nicht so empfindlich eingestellt, sodass sie nur tatsächliche Concept Drifts in den Daten erkennt. Diese Instanz arbeitet identisch zu der Concept Drift Erkennung im AdaptivePatchingADWIN-Algorithmus. Wird mit dieser Instanz ein Concept Drift erkannt, verhält sich der AdaptivePatchingTwoADWINs-Algorithmus wie der AdaptivePatchingADWIN-Algorithmus. Die Liste des ADWIN-Algorithmus verkürzt sich automatisch und die Länge dieser ADWIN-Liste wird für die Größe des Instance Stores übernommen. Dem AdaptivePatchingTwoADWINs-Algorithmus stehen durch die zusätzliche Anpassung der Instance Store Größe weniger Instanzen zum Trainieren des Fehlerregionen- und des Patchklassifizierers zur Verfügung. Oftmals erkennt die empfindlich eingestellte ADWIN-Instanz in jedem Batch eine Veränderung des Concepts. Dadurch verringert sich die Größe des Instance Stores rapide und der Instance Store beinhaltet nur noch einen Batch. Somit ist die Klassifizierungsgenauigkeit nicht so hoch wie bei den vorangegangenen Algorithmen, die Laufzeit ist jedoch besser. Durch die Änderung des Eingabeparameters des empfindlichen ADWIN kann jedoch die Klassifizierungsgenauigkeit im tradeoff gegen die Geschwindigkeit verändert werden und somit lassen sich auch bessere Klassifizierungsgenauigkeiten erzielen.

4 Implementierung im MOA Framework

In diesem Abschnitt wird das Framework *MOA*[3] kurz erläutert. Weiterhin werden die bereits existierenden und neu implementierten Funktionen vorgestellt.

4.1 MOA Framework

MOA (Massive Online Analysis) ist ein Framework für Data Stream Mining. In unserem Aufbau wird zusätzlich zu *MOA* auch *WEKA* eingesetzt. Dabei ist *WEKA* in *MOA* integriert, sodass in *MOA* lediglich die machine learning Algorithmen aus *WEKA* zur Verfügung stehen. *MOA* beinhaltet eine Vielzahl von Algorithmen, die sich zum Data Stream Mining eignen. Durch die Anbindung an *WEKA* stehen zusätzlich die Algorithmen zum Maschinellen Lernen zur Verfügung. Die Implementierung und Benutzung neuer Algorithmen in *MOA* gestaltet sich sehr einfach, da sich die neu erstellten Dateien problemlos in die GUI von *MOA* eingliedern. *MOA* stellt zusätzlich einige Funktionen zur Verfügung, mit denen die Leistung der Klassifizierer evaluiert werden kann.

5 Evaluation

Dieses Kapitel befasst sich mit den Experimenten, die mit den verschiedenen Anpassungen am *Patching*-Algorithmus[9] durchgeführt wurden. Um die Güte der Klassifizierer miteinander vergleichen zu können, wurden verschiedene Daten zur Leistungsmessung genutzt. Dazu zählen zum einen der prozentuale Anteil, wie viele Instanzen der Klassifizierer korrekt klassifiziert hat, zum anderen die Auslastung des RAM und die Rechenzeit auf dem Prozessor.

Im konkreten Anwendungsfall des Patching-Algorithmus, existiert bereits ein vorhandener Klassifizierer, der zur Verfügung steht. Die Aufgabe des Algorithmus besteht dann nur im Lernen der Patches, um die Instanzen auch nach einem Concept Drift noch mit einer hohen Genauigkeit klassifizieren zu können. Es muss also insbesondere kein Base-Learner im Algorithmus ausgewählt werden.

5.1 Datensätze

Nachfolgend werden die Datensätze erläutert, die für die Experimente genutzt wurden. Dabei wird kurz auf die originalen Datensätze eingegangen und anschließend erklärt, wie diese Datensätze für die Experimente verändert wurden. Durch die Veränderung der Datensätze wurde künstlich ein Concept Drift eingefügt. Dies war nötig, da die ursprünglichen Datensätze keinen Concept Drift beinhalteten.

5.1.1 MNIST

Der *MNIST* Datensatz[1] besteht aus Bildbeispielen handgeschriebener Ziffern von null bis neun. Die Attribute einer Instanz beschreiben die Pixel dieser Bilder und ihre Werte entsprechen Graustufenwerten im Wertebereich von 0 bis 255. Die generierten Datensätze für die Klassifizierung liegen in der Domäne des Concept Drifts und des Transfer Learnings. Jeder Datensatz beinhaltet 70000 beziehungsweise 140000 Instanzen. Die Datensätze setzen sich wie folgt zusammen:

Im Datensatz $MNIST_{merge}$ sind die ersten 35000 Instanzen unverändert. Der Klassifizierer lernt alle zehn Klassen. Danach ändern sich die Labels der Klassen ’4’, ’5’ und ’7’ zu dem Label ’9’. Also werden Instanzen, die vorher als ’4’, ’5’ oder ’7’ gelernt wurden, potenziell falsch klassifiziert, da zum einen nur noch die Klassen ’0’, ’1’, ’2’, ’3’, ’6’, ’8’ und ’9’ zur Verfügung stehen, zum anderen jedoch diese Instanzen alle als ’9’ klassifiziert werden sollen.

Der Datensatz $MNIST_{split}$ verhält sich gegenteilig zu dem Datensatz $MNIST_{merge}$. In den ersten 35000 Instanzen existieren zunächst nur die Klassen ’0’, ’1’, ’2’, ’3’, ’6’, ’8’ und ’9’. Der Klassifizierer lernt somit nur diese sieben Klassen. Danach ändern sich die Daten, sodass auch die Klassen ’4’, ’5’ und ’7’ existieren. Vorher wurden diese Klassen alle als Klasse ’9’ klassifiziert. Der Klassifizierer muss also die Klassen ’4’, ’5’ und ’7’ neu lernen.

Im Datensatz $MNIST_{appear}$ existieren in den ersten 35000 Instanzen nur die Klassen ’0’, ’1’, ’2’, ’4’, ’6’ und ’8’. In der zweiten Hälfte des Datensatzes kommen dann die anderen Klassen (’3’, ’5’, ’7’ und ’9’) hinzu. Der Klassifizierer lernt initial also nur sechs Klassen.

Der vierte Datensatz $MNIST_{step}$ ist in den ersten 35000 Instanzen unverändert. Anschließend werden im Abstand von 10000 Instanzen die Klassen ’4’, ’5’ und ’6’ verändert. Die Veränderung ist dabei, dass die Pixel einer Instanz sowohl horizontal als auch vertikal gespiegelt werden.

Im fünften Datensatz $MNIST_{flipped}$ sind die ersten 70000 Instanzen unverändert. Dieser Datensatz besteht aus 140000 Instanzen. Die folgenden Instanzen sind wie im vorherigen Datensatz horizontal und vertikal gespiegelt. Der Unterschied liegt jedoch darin, dass sich diese Veränderung sofort auf alle Klassen auswirkt. Dadurch liegt dieser Datensatz in der Domäne des Transfer Learnings.

[1] http://yann.lecun.com/exdb/mnist/

5.1.2 20 Newsgroups

Der zweite Datensatz befasst sich mit der Domäne Transfer Learning. Es handelt sich hierbei um den Datensatz *20 Newsgroups*[2]. Er enthält Newsgroups-Einträge, die in hierarchische Kategorien eingeordnet sind. Dabei gibt es übergeordnete Kategorien und Unterkategorien dieser. Die Klasse ist die übergeordnete Kategorie. Die Unterkategorie entscheidet, ob die Instanz den Trainings- oder den Testdaten zugehörig ist. Somit wird auf anderen Unterkategorien gelernt, als in den Testdaten evaluiert wird. Ein Datensatz beinhaltet zwischen 4300 und 4700 Instanzen.

5.1.3 SEA

Bei dem dritten Datensatz handelt es sich um den *SEA*-Datensatz[16]. Er besteht aus Instanzen mit jeweils drei reellen Werte zwischen 0 und 10 als Attribute. Aus diesem Datensatz wurden zwei Datensätze zur Evaluation generiert. Im ersten Datensatz wurden die ersten beiden Attribute miteinander addiert. Anhand eines Schwellenwerts θ wurden die Instanzen zwei Klassen zugeordnet. Im zweiten Datensatz wurden alle drei Attribute miteinander multipliziert und ebenfalls mithilfe eines Schwellenwerts θ zwei Klassen zugewiesen. Der abrupte Concept Drift wurde in beiden Datensätzen dadurch erreicht, dass sich der Schwellenwert ändert. Die so generierten Datensätze haben 500000 Instanzen.

5.1.4 Rotating Hyperplane

Der letzte Datensatz ist der $RotatingHyperplane$-Datensatz[5] von W. Fan. Der Datensatz beinhaltet Instanzen mit 10 Attributen. Diese stellen Koordinaten im 10-dimensionalen Raum dar. In diesem 10-dimensionalen Raum wird eine Hyperebene definiert. Um zu testen, ob ein Punkt in der Hyperebene liegt, kann das Skalarprodukt von dem Punktvektor $\vec{x}$ und einem für die Hyperebene stehenden Gewichtsvektor $\vec{w}$ gebildet werden. Das Skalarprodukt wird dann mit einem Wert w_0 verglichen: $\sum_{i=1}^{d} w_i x_i = w_0$. Die Instanzen werden in zwei Klassen unterteilt. Wenn die Instanz x die Formel $\sum_{i=1}^{d} w_i x_i \geq w_0$ erfüllt, liegt sie in der Klasse 'positiv', andernfalls in der Klasse 'negativ'. Der Concept Drift wird durch die Veränderung des Vektors $\vec{w}$ herbeigeführt. Durch die Veränderung von $\vec{w}$ ändert die Hyperebene ihre Ausrichtung beziehungsweise ihre Position. Die generierten Datensätze beinhalten 500000 Instanzen.

[2] http://people.cs.umass.edu/~mccallum/data/20_newsgroups.tar.gz

5.2 Vergleich der Klassifizierer

Nachfolgend werden verschiedene Ansätze zum Umgang mit Concept Drifts verglichen. Dabei werden folgende Algorithmen betrachtet:

- **Patching-Algorithmus**: Patching-Algorithmus ohne Erweiterung. Instance Store Größe ist limitiert.
- **AdaptivePatchingADWIN-Algorithmus**: Patching-Algorithmus mit ADWIN. Instance Store Größe variabel durch Varianzkriterium, nach oben beschränkt.
- **AdaptivePatchingADWINNoLimit-Algorithmus**: Patching-Algorithmus mit ADWIN. Instance Store Größe unbeschränkt.
- **AdaptivePatchingTwoADWINs-Algorithmus**: Patching-Algorithmus mit ADWIN. Größe des Instance Stores wird durch zweite ADWIN-Instanz geregelt.

Es werden die oben genannten Ansätze anhand von Genauigkeit und Laufzeit miteinander verglichen. Dabei wird auf die Datensätze $MNIST_{merge}$, $MNIST_{split}$, SEA_{linear}, $RotatingHyperplane$ und $20Newsgroups_{REC\,vs\,TALK}$ eingegangen.

In den Experimenten stehen den oben genannten Klassifizierern jeweils als Basis-, Fehlerregionen- und Patchklassifizierer ein *RandomForest mit 100 Bäumen* zur Verfügung.

5.2.1 $MNIST_{merge}$

In Abbildung 8 wird zunächst die Genauigkeit der Implementierungen auf dem Datensatz $MNIST_{merge}$ miteinander verglichen. Der Concept Drift ist mit der schwarzen gestrichelten Linie markiert (Instanz 35000). Die Patching-Implementierungen benutzen die ersten 20000 Instanzen zum Trainieren des Basisklassifizierers und gehen danach zum Batch-Lernen über. Man erkennt, dass die normale Implementierung des *Patching*-Algorithmus (blau dargestellt) im Gegensatz zu den Patching-Implementierungen mit ADWIN-Erweiterung nach Auftreten des Concept Drifts längere Zeit geringere Genauigkeit hat. Dies ist damit zu erklären, dass der normale Patching-Algorithmus den Instance Store nicht von veralteten Instanzen bereinigt. Die Patching-Implementierung ADWIN-No-Limit benutzt alle gesehenen Instanzen zum Erstellen des Patches, sofern kein Concept Drift erkannt wurde. Demnach ist diese Implementierung als obere Schranke der möglichen Genauigkeit zu sehen. Die Implementierungen AdaptivePatchingADWIN und AdaptivePatchingTwoADWINs sind als Tradeoff zwischen Genauigkeit und Laufzeit zu verstehen. Die geringere Genauigkeit des AdaptivePatchingTwoADWINs ist dadurch begründet, dass durch die zweite ADWIN-Implementierung die Größe des Instance Stores zu klein gehalten wird und somit dem Fehlerregionenklassifizierer und dem Patchklassifizierer weniger Trainingsdaten zur Verfügung stehen. Dies hat zur Folge, dass der Fehlerregionenklassifizierer und der Patchklassifizierer nicht ausreichend trainiert werden können und somit Fehler bei der Klassifikation macht, die sich dann negativ auf die gesamte Genauigkeit der Klassifizierung des Algorithmus auswirken. Durch die geringe Größe der Trainingsdaten ist jedoch die Laufzeit besser, als bei den anderen Patching-Implementierungen.

In Tabelle 2 sind die benötigten Laufzeiten der jeweiligen Klassifizierer dargestellt. In der ersten Spalte sind die Klassifizierer aufgelistet. Die zweite Spalte gibt die Laufzeit in Sekunden bis zum Concept Drift an. In der letzten Spalte wird die Gesamtlaufzeit des Klassifizierungsvorgangs dargestellt. Der AdaptivePatchingADWIN-Algorithmus kann in der Zeit bis zum Concept Drift Laufzeit einsparen, da der Instance Store zu Beginn nicht wächst und dem Fehlerregionenklassifizierer und den Patchklassifizierern somit nur wenige Daten zum Lernen zur Verfügung stehen. Das ist vor dem Concept Drift allerdings nicht schlimm, da hier der Großteil der Instanzen durch den Basisklassifizierer klassifiziert wird. Somit ist auch in Abbildung 8 bis zum Concept Drift kein Performanceverlust erkennbar. Der AdaptivePatchingADWINNoLimit-Algorithmus hat zwar die höchste Klassifizierungsgenauigkeit, benötigt aber auch die dreifache Laufzeit im Gegensatz zum Patching-Algorithmus. Der AdaptivePatchingTwoADWINs-Algorithmus kann für die kurze Laufzeit gerade gegen Ende eine hohe Klassifizierungsgenauigkeit erzielen.

Klassifizierer	Laufzeit bis CP [s]	Gesamtlaufzeit [s]
Patching	189	673
AdaptivePatchingADWIN	80	524
AdaptivePatchingADWINNoLimit	258	1853
AdaptivePatchingTwoADWINs	109	281

Tabelle 2: Laufzeit $MNIST_{merge}$

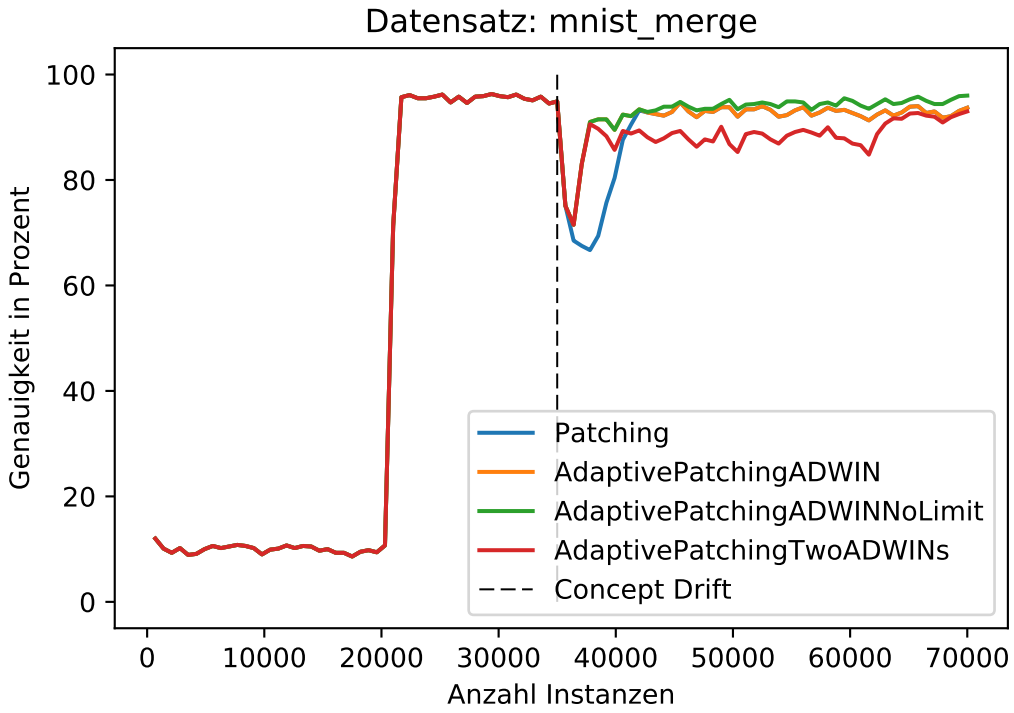


Abbildung 8: Genauigkeit $MNIST_{merge}$

In Tabelle 3 ist die mittlere Klassifizierungsgenauigkeit dargestellt. Diese berechnet sich aus dem Mittelwert aller Klassifizierungsergebnisse ab Instanz 20000. Hier kann man erkennen, dass der AdaptivePatchingADWINNoLimit-Algorithmus die höchste durchschnittliche Klassifizierungsgenauigkeit hat. Dies hat allerdings, wie bereits oben erwähnt und in Tabelle 2 zu sehen, zur Folge, dass die Laufzeit am höchsten ist. Auch der AdaptivePatchingADWIN-Algorithmus kann eine bessere Genauigkeit als der Patching-Algorithmus erzielen. Dies liegt vor allem an der geringen Zeit, die benötigt wird, die Klassifizierungsgenauigkeit nach dem Concept Drift wieder zu erhöhen.

Klassifizierer	**Klassifizierungsgenauigkeit [%]**
Patching	91,47
AdaptivePatchingADWIN	92,85
AdaptivePatchingADWINNoLimit	93,71
AdaptivePatchingTwoADWINs	90,37

Tabelle 3: mittlere Klassifizierungsgenauigkeit $MNIST_{merge}$

5.2.2 $MNIST_{split}$

Abbildung 9 zeigt die Klassifizierungsgenauigkeit für den Datensatz $MNIST_{split}$. Auch hier wird auf den ersten 20000 Instanzen der Basisiklassifizierer trainiert, und es stehen keine Evaluationsdaten zur Verfügung, weshalb die Baseline dargestellt ist. Ebenfalls ist auch in dieser Abbildung der Concept Drift bei Instanz 35000 durch eine schwarze Linie markiert. Ähnlich wie beim vorhergehenden Datensatz $MNIST_{merge}$ kann man auch hier nach dem Concept Drift erkennen, dass die Algorithmen, die den ADWIN-Algorithmus zur Concept Drift Erkennung benutzen, schneller zu einer höheren Klassifizierungsgenauigkeit zurückkehren können. Nachdem sich auch die Klassifizierungsgenauigkeit des Patching-Algorithmus erholt hat, ist seine Kurve jedoch identisch zu der des AdaptivePatchingADWIN-Algorithmus. Somit ist gegen Ende keine Leistungssteigerung des AdaptivePatchingADWIN-Klassifizieres gegenüber dem Patching-Algorithmus zu verzeichnen. Lediglich der AdaptivePatchingADWINNoLimit-Algorithmus kann ab dem Concept Drift dauerhaft eine bessere Performance erzielen als der Patching-Algorithmus. Das schlägt sich allerdings auch in der Laufzeit des Algorithmus nieder. Dazu wurden in der Tabelle 4 die Laufzeiten zu den Algorithmen abgetragen. Der AdaptivPatchingADWIN-Algorithmus

benötigt bis zum Concept Drift die geringste Rechenzeit. Auch in der Gesamtlaufzeit ist er schneller als der Patching-Algorithmus. Der AdaptivePatchingADWINNoLimit-Algorithmus war, wie zu erwarten, mit der dreifachen Laufzeit zum Patching-Algorithmus am langsamsten.

Klassifizierer	Laufzeit bis CP [s]	Gesamtlaufzeit [s]
Patching	171	669
AdaptivePatchingADWIN	71	547
AdaptivePatchingADWINNoLimit	244	1891
AdaptivePatchingTwoADWINs	118	402

Tabelle 4: Laufzeit $MNIST_{split}$

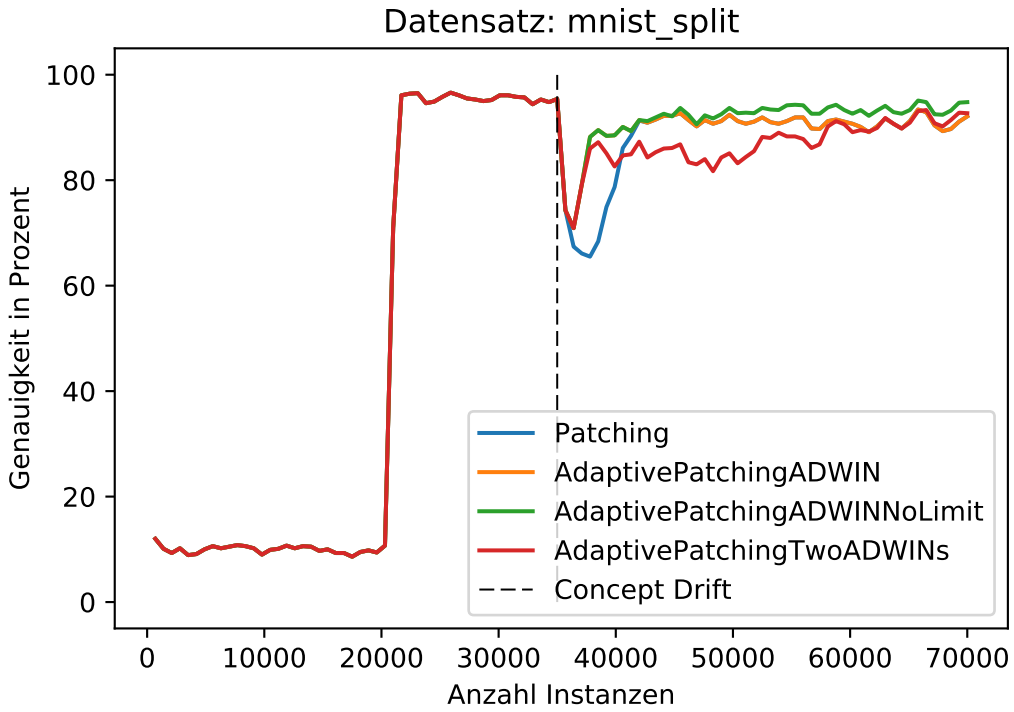


Abbildung 9: Genauigkeit $MNIST_{split}$

In Tabelle 5 sind die Klassifizierungsgenauigkeiten für den Datensatz $MNIST_{split}$ dargestellt. Wie auch im vorherigen Datensatz $MNIST_{merge}$ hat der AdaptivePatchingADWINNoLimit-Algorithmus die höchste Klassifizierungsgenauigkeit. Für diese Werte wurden ebenfalls die Klassifizierungsgenauigkeiten ab Instanz 20000 gemittelt. Auch der AdaptivePatchingADWIN-Algorithmus hat eine höhere Klassifizierungsgenauigkeit gegenüber dem Patching-Algorithmus. Dies ist durch die schnellere Erholung der Klassifizierungsgenauigkeit bei den Patching-Algorithmen mit Concept Drift Erkennung im Gegensatz zum originalen Patching-Algorithmus zu erklären.

Klassifizierer	Klassifizierungsgenauigkeit [%]
Patching	90,24
AdaptivePatchingADWIN	91,51
AdaptivePatchingADWINNoLimit	92,65
AdaptivePatchingTwoADWINs	89,38

Tabelle 5: mittlere Klassifizierungsgenauigkeit $MNIST_{split}$

5.2.3 SEA_{linear}

Im Datensatz SEA_{linear} tauchen zwei Change Punkte auf. Der Datensatz hat eine Größe von 500000 Instanzen und die Change Punkte treten bei den Instanzen 150000 und 250000 auf. Auf den ersten 120000 Instanzen wird der Basisklassifizierer trainiert, sodass hier keine Evaluationsdaten zur Verfügung stehen und deshalb die Baseline dargestellt wird. Auch bei diesem Datensatz ist deutlich erkennbar, dass die Patching-Varianten, die die ADWIN-Implementierung enthalten, schneller zur vorherigen höheren Klassifikationsgenauigkeit zurückkehren. Die Laufzeiten in Tabelle 6 zeigen wieder, dass der AdaptivePatchingADWIN-Algorithmus zu Beginn Laufzeit einsparen kann. Der Patching-Algorithmus benötigt bis zum ersten Change Punkt die höchste Rechenzeit, da die anderen Algorithmen erkennen, dass das Verwenden weiterer Batches zu keiner Verbesserung in der Genauigkeit führt. Der AdaptivePatchingADWINNoLimit-Algorithmus hat die längste Gesamtlaufzeit, da die Instance Store Größe nicht nach oben limitiert ist. Der AdaptivePatchingADWIN-Algorithmus hat die geringste Gesamtlaufzeit. Er ist schneller als der Patching-Algorithmus, da durch das Varianzkriterium und die Concept Drift Erkennung der Instance Store klein gehalten wird. Die Größe des Instance Stores ist proportional zur Laufzeit die benötigt wird, um einen Patch zu lernen. Auch bis zum zweiten Change Punkt benötigt der AdaptivePatchingADWIN-Algorithmus eine geringere Laufzeit, da durch das Varianzkriterium die Größe des Instance Stores weiter gering gehalten wird. Diese Zeitersparnis schlägt sich auch auf die Gesamtzeit nieder. Der Instance Store des Patching-Algorithmus wächst bereits ab Instanz 120000 und benötigt somit bis zum ersten Change Punkt eine längere Laufzeit, als die Implementierungen mit ADWIN als Concept Drift Erkennung. Bis zum zweiten Change Punkt benötigt der Patching-Algorithmus eine geringere Laufzeit, als der AdaptivePatchingADWINNoLimit-Algorithmus, da dessen Instance Store in der Größe nach oben nicht beschränkt ist. Dies wird auch in der Gesamtlaufzeit deutlich.

Klassifizierer	Laufzeit bis CP 1 [s]	Laufzeit bis CP 2 [s]	Gesamtlaufzeit [s]
Patching	70	391	888
AdaptivePatchingADWIN	33	71	507
AdaptivePatchingADWINNoLimit	45	417	2788
AdaptivePatchingTwoADWINs	35	412	2745

Tabelle 6: Laufzeit SEA_{linear}

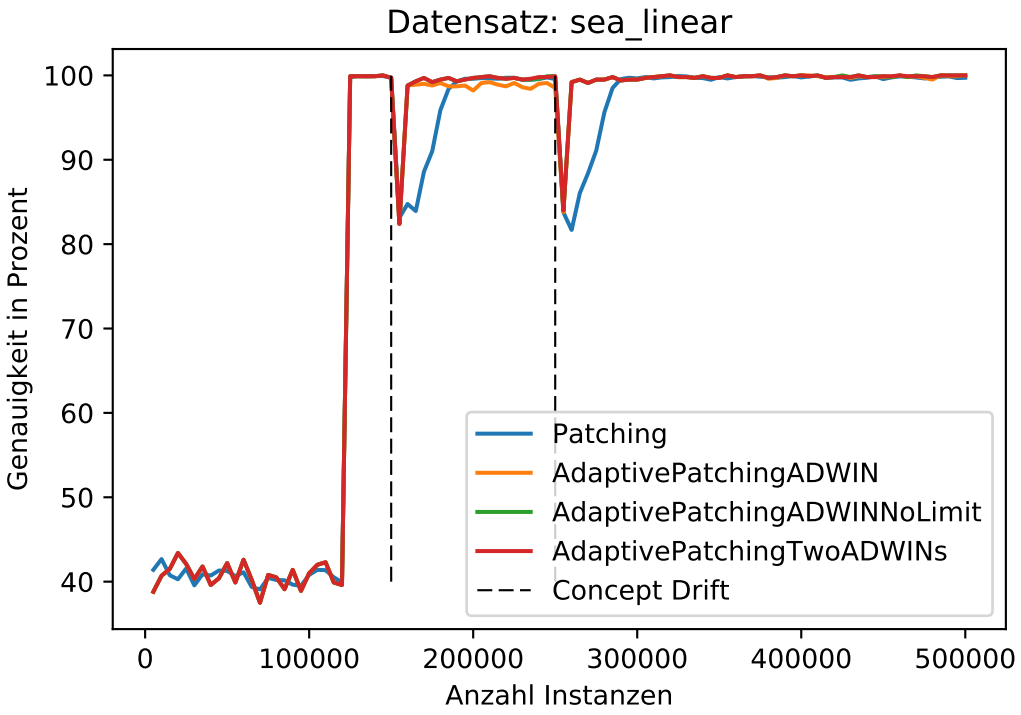


Abbildung 10: Genauigkeit SEA_{linear}

In Tabelle 7 werden die Klassifizierungsgenauigkeiten dargestellt. Die Daten sind jeweils über alle Klassifizierungsergebnisse ab Instanz 120000 gemittelt. Die Klassifizierungsgenauigkeit der AdaptivePatchingADWINNoLimit- und AdaptivePatchingTwoADWINs-Algorithmen ist bei diesem Datensatz identisch und bei nahezu 100%. Die Klassifizierungsgenauigkeit des AdaptivePatchingADWIN-Algorithmus ist trotz der geringsten Laufzeit besser, als die des Patching-Algorithmus. Der Grund ist auch hierfür wieder vorwiegend, dass die Genauigkeit nach dem Concept Drift deutlich schneller zunehmen kann als dies beim Patching-Algorithmus der Fall ist.

Klassifizierer	**Klassifizierungsgenauigkeit [%]**
Patching	97,83
AdaptivePatchingADWIN	99,05
AdaptivePatchingADWINNoLimit	99,31
AdaptivePatchingTwoADWINs	99,31

Tabelle 7: mittlere Klassifizierungsgenauigkeit SEA_{linear}

5.2.4 *RotatingHyperplane*

Im nächstes Datensatz, den wir in der Evaluation betrachten wollen, geht es um die *RotatingHyperplane*. Hier kann man keinen einzelnen Concept Drift definieren, da sich die Hyperebene dreht und sich die Daten somit schrittweise verändern. Daher ist in Abbildung 11 kein Concept Drift eingezeichnet. Wie beim vorherigen Datensatz auch, wird auf den ersten 120000 Instanzen der Basisklassifizierer trainiert. Somit stehen, wie bereits oben erwähnt, keine Evaluationsdaten zur Verfügung und es wird die Baseline dargestellt. Die Patching-Algorithmen, die den ADWIN-Algorithmus implementiert haben, haben bei diesem Datensatz immer eine höhere, beziehungsweise mindestens eine genauso hohe Klassifizierungsgenauigkeit wie der Patching-Algorithmus. Wie Tabelle 8 zeigt, benötigt der Patching-Algorithmus zusätzlich dazu die längste Rechenzeit. Der AdaptivePatchingADWIN- und der AdaptivePatchingADWINNoLimit-Algorithmus benötigen in etwa dieselbe Rechenzeit, und die Kurven der Klassifizierungsgenauigkeit liegen auch in Abbildung 11 übereinander. Der AdaptivePatchingTwoADWINs-Algorithmus hat die kürzeste Laufzeit, gleichzeitig aber bei diesem Datensatz die höchste durchschnittliche Klassifizierungsgenauigkeit, wie in Tabelle 9 zu sehen ist. Die geringe Laufzeit kommt dadurch zustande, dass die Größe des Instance Stores durch die vielen Veränderungen im Datensatz klein gehalten wird. Die empfindlichere ADWIN-Implementierung sorgt dabei zusätzlich dafür, dass die Größe nicht weiter wächst. Diese sehr geringe Größe des Instance Stores hilft ebenfalls, die Klassifizierungsgenauigkeit hoch zu halten.

Klassifizierer	**Gesamtlaufzeit [s]**
Patching	3735
AdaptivePatchingADWIN	2363
AdaptivePatchingADWINNoLimit	2410
AdaptivePatchingTwoADWINs	1201

Tabelle 8: Laufzeit *RotatingHyperplane*

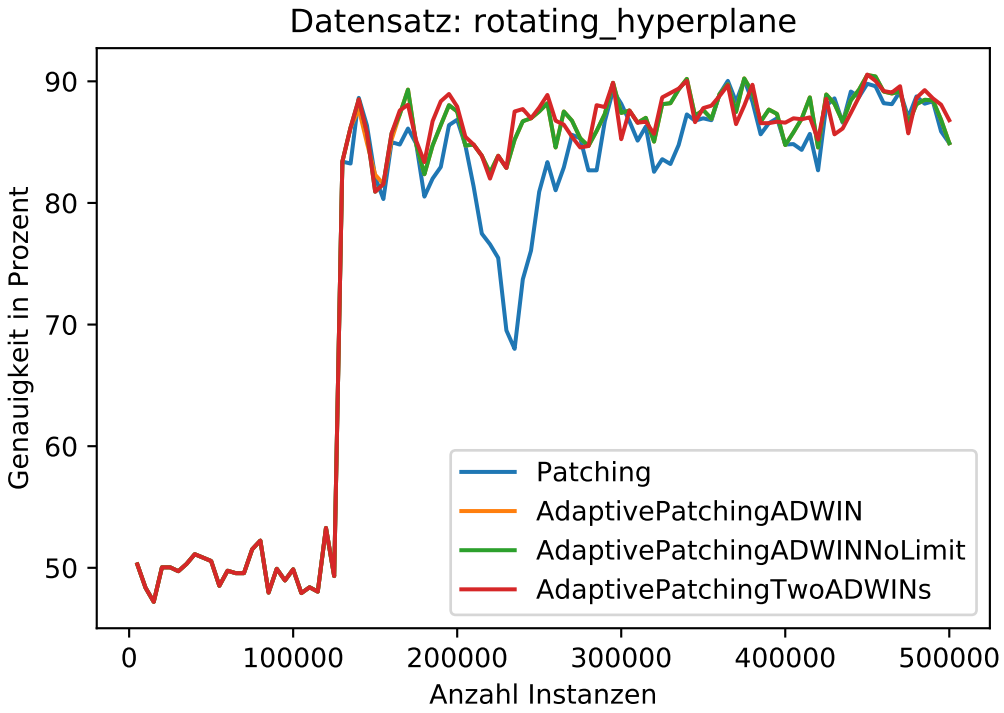


Abbildung 11: Genauigkeit $RotatingHyperplane$

Wie bereits oben erwähnt, wird in Tabelle 9 die Klassifizierungsgenauigkeit für den Datensatz $RotatingHyperplane$ dargestellt. Die Algorithmen, die die Veränderungen mithilfe des ADWIN-Algorithmus erkennen, haben eine ähnliche und höhere Klassifizierungsgenauigkeit als der Patching-Algorithmus. Die Klassifizierungsgenauigkeit des Patching-Algorithmus leidet auch hier wieder unter der festen Größe des Instance Stores.

Klassifizierer	**Klassifizierungsgenauigkeit [%]**
Patching	84,68
AdaptivePatchingADWIN	86,85
AdaptivePatchingADWINNoLimit	86,85
AdaptivePatchingTwoADWINs	86,96

Tabelle 9: mittlere Klassifizierungsgenauigkeit $RotatingHyperplane$

5.2.5 $20Newsgroups_{REC\,vs\,TALK}$

Zuletzt wird der Datensatz $20Newsgroups_{REC\,vs\,TALK}$ betrachtet. Dieser Datensatz fällt nicht wie die anderen Datensätze in die Domäne des Concept Drifts, sondern in die Domäne *Transfer Learning*. Abbildung 12 illustriert die Klassifizierungsgenauigkeit der Algorithmen auf diesem Datensatz. Die ersten 1000 Instanzen werden zum Erstellen des Basisklassifizierers genutzt und es stehen, wie bei den vorherigen Datensätzen auch, keine Evaluationsdaten zur Verfügung. Ab Instanz 1000 werden die korrekten Klassifizierungsgenauigkeiten angezeigt. Der AdaptivePatchingADWIN-Algorithmus hat auch bei diesem Datensatz die geringste Laufzeit, wie sich aus Tabelle 10 ablesen lässt. Die Patching-Varianten mit Concept Drift Erkennung liegen in der Klassifizierungsgenauigkeit nah beieinander. Dies ist sowohl in Abbildung 12 also auch in Tabelle 11 zu erkennen. Wie zu erwarten, hat auch bei diesem Datensatz der AdaptivePatchingADWINNoLimit-Algorithmus die höchste Laufzeit. Beim Transfer Learning wird in diesem Datensatz der Basisklassifizier auf anderen Unterkategorien trainiert als später zur Evaluation genutzt werden. Der Klassifizierer soll die Top-Kategorie, hier REC oder TALK anhand der Unterkategorien vorhersagen. Ab Instanz 2766 werden die anderen Unterkategorien zur Klassifizierung genutzt. Auch beim Transfer Learning ist es vorteilhafter, die Größe des Instance Stores dynamisch und nach einem Change Punkt klein zu halten, um schneller auf die Veränderung reagieren zu können.

Klassifizierer	Gesamtlaufzeit [s]
Patching	83
AdaptivePatchingADWIN	64
AdaptivePatchingADWINNoLimit	202
AdaptivePatchingTwoADWINs	95

Tabelle 10: Laufzeit $20Newsgroups_{RECvsTALK}$

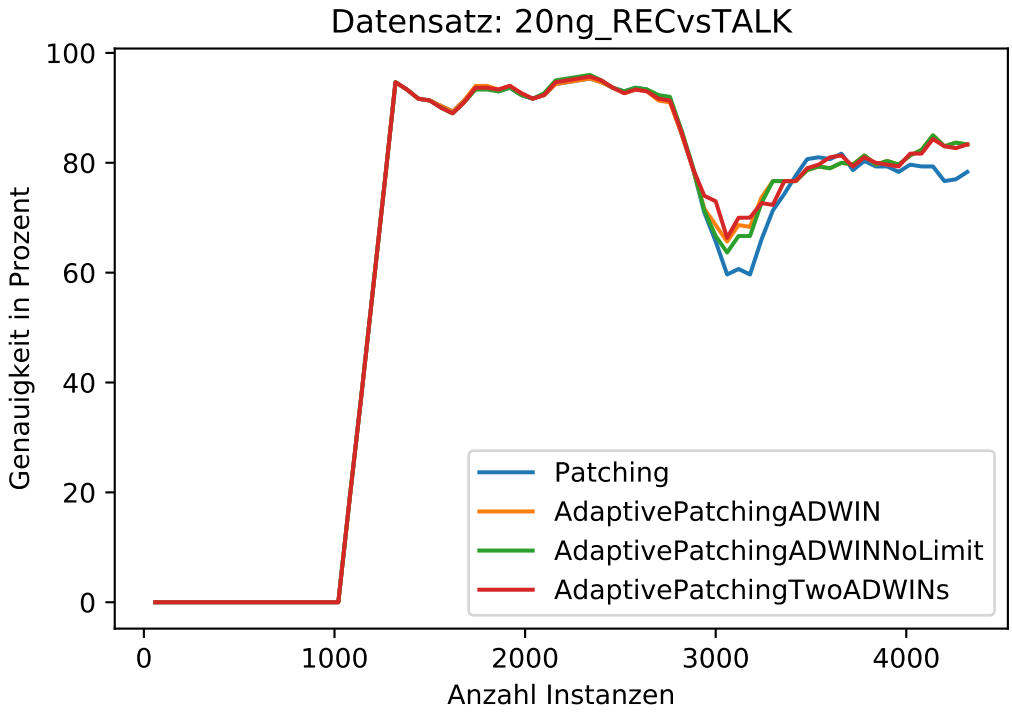


Abbildung 12: Genauigkeit $20Newsgroups_{RECvsTALK}$

Klassifizierer	Klassifizierungsgenauigkeit [%]
Patching	83,99
AdaptivePatchingADWIN	85,30
AdaptivePatchingADWINNoLimit	85,30
AdaptivePatchingTwoADWINs	85,21

Tabelle 11: mittlere Klassifizierungsgenauigkeit $20Newsgroups_{RECvsTALK}$

5.2.6 Nutzung des Basisklassifizierers

Nun wird betrachtet, in welchem Ausmaß der Basisklassifizierer nach einem Concept Drift weiterverwendet wird. Für die beiden Datensätze $MNIST_{merge}$ und SEA_{linear} ist die Nutzung des Basisklassifizierers dargestellt. Als Algorithmus wurde hier nur der AdaptivePatchingADWIN-Algorithmus betrachtet.

Abbildung 13 zeigt den Datensatz mit dem Concept Drift bei Instanz 35000, durch die schwarze Linie markiert. Bis zu dem Concept Drift werden über 95% der Instanzen durch den Basisklassifizierer klassifiziert. Dieses Ergebnis war zu erwarten, da die Instanzen dem Concept folgen, auf dem der Basisklassifizierer trainiert wurde. Ab dem Concept Drift bei Instanz 35000 werden nur etwa 70% durch den Basisklassifizierer klassifiziert. Die Anzahl der Instanzen, die vom Basisiklassifizierer klassifiziert werden, fluktuiert um etwa 69%. Sieht man sich die Änderung im Datensatz $MNIST_{merge}$ an, kann man auch erkennen, wie sich dieser Wert zusammensetzt. Ab Instanz 35000 werden die Klassen '4', '5' und '7' zu '9' geändert. Demnach werden 30% der Instanzen, jene mit den ursprünglichen Klassen '4', '5' und '7', vom Basisklassifizierer falsch klassifiziert. Diese Klassifizierungen werden vom Patchklassifizierer übernommen.

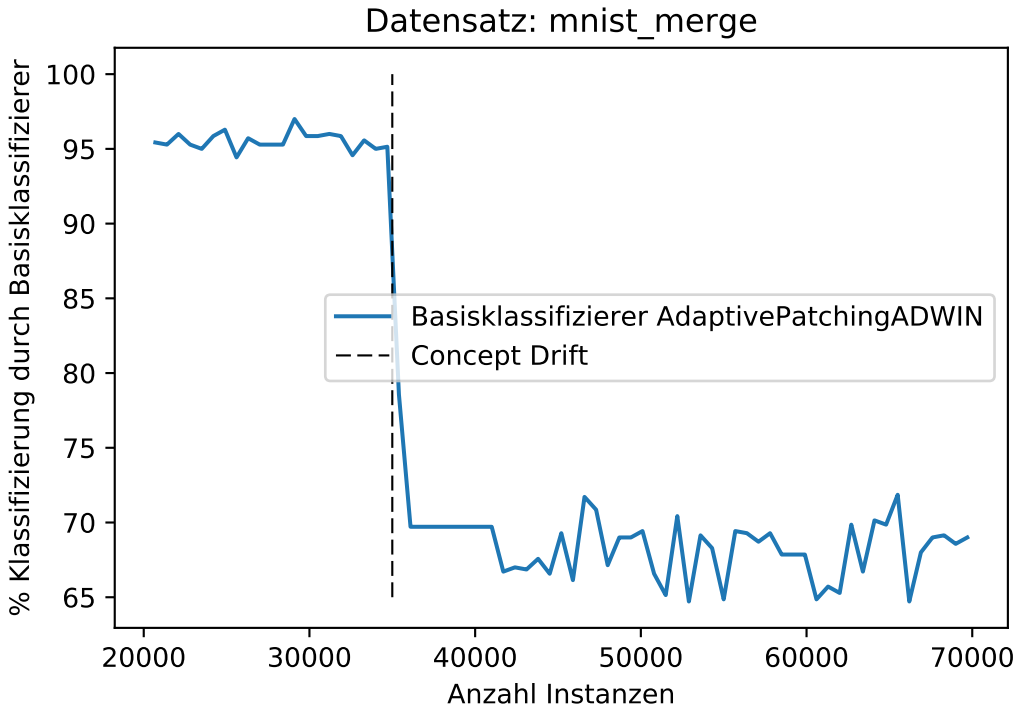


Abbildung 13: Klassifizierung durch den Basisklassifizierer $MNIST_{merge}$ (AdaptivePatchingADWIN)

In Abbildung 14 wird die Anzahl der Instanzen, die durch den Basisklassifizierer klassifiziert, werden dargestellt. Zu Beginn werden nahezu 100% der Instanzen vom Basisklassifizierer klassifiziert, welcher auf dem Concept dieser Instanzen trainiert wurde. Am ersten Change Punkt, bei Instanz 150000, wird der Schwellenwert θ verändert, sodass sich die Klassen einiger Instanzen ändern. Für 100000 Instanzen bleibt θ konstant und auch die Anzahl der Instanzen, die vom Basisklassifizierer klassifiziert werden, bleibt in etwa konstant. Beim zweiten Change Punkt bei Instanz 250000 ändert sich der Schwellenwert θ erneut. θ entfernt sich noch einmal mehr von seinem ursprünglichen Wert, sodass der Basisklassifizierer noch weniger Instanzen klassifiziert. Auch hier pendelt sich die Anzahl der Instanzen nach geringer Zeit auf einen Wert ein.

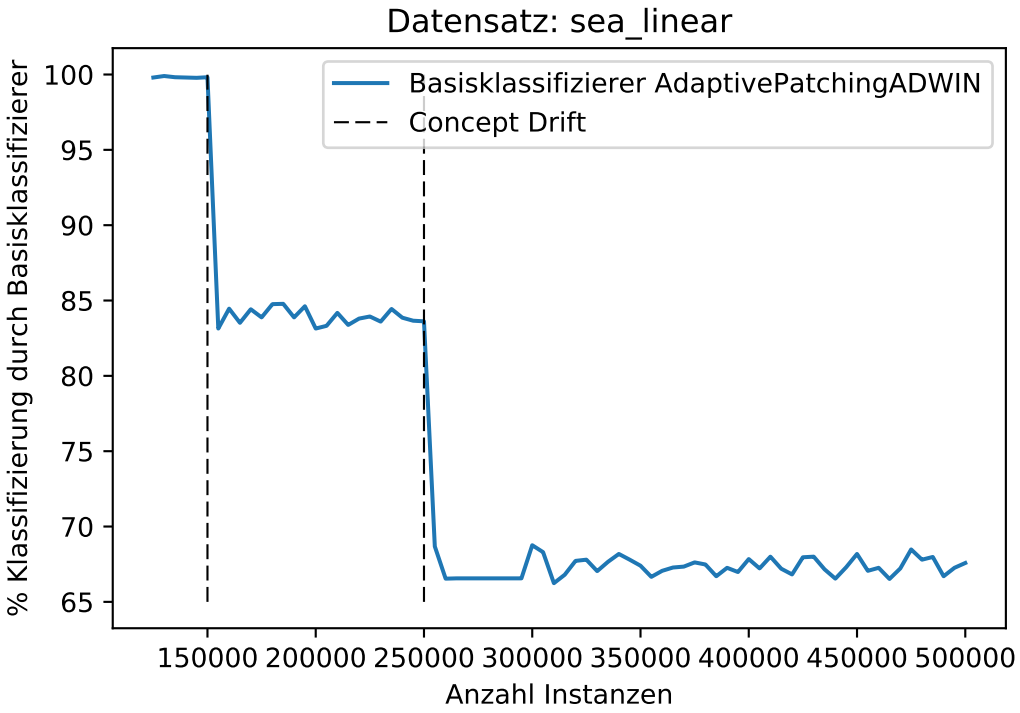


Abbildung 14: Klassifizierung durch den Basisklassifizierer SEA_{linear} (AdaptivePatchingADWIN)

6 Verwandte Arbeiten

In dieser Arbeit wurde der ADWIN-Algorithmus zur Detektion von Concept Drifts verwendet. Neben ADWIN gibt es noch viele weitere Algorithmen, um Concept Drifts zu erkennen. Dabei gibt es verschiedene Ansätze. Der ADWIN-Algorithmus benutzt die Information, dass sich die Genauigkeit des Klassifizierers durch Concept Drifts im Datenstrom verschlechtert. Alternativ könnten Paired Learners, vorgeschlagen von Stephen Bach et al.[1], verwendet werden. Paired Learners vergleichen die Genauigkeit zwischen zwei Klassifizierern. Der erste Klassifizierer wird auf den gesamten Daten trainiert, wohingegen der zweite Klassifizierer nur auf einem Fenster, bestehend aus den letzten Trainingsdaten, trainiert wird. Aus der Differenz der Genauigkeit der beiden Klassifizierer wird abgeleitet, ob ein Concept Drift auftritt. Um die Signifikanz der Differenzen in der Genauigkeit zu bestimmen, können auch mathematisch komplexere Methoden verwendet werden. In Statistical Test of Equal Proportions (STEPD) von Nishida et al.[10] wird zum Beispiel der χ^2-Test zur Detektion von Concept Drifts benutzt. Sobhani et al. benutzt den Nearest-Neighbour-Algorithmus[15], um jede Instanz des aktuellen Batches mit einer Instanz aus dem vorherigen Batch zu assoziieren. Aus den Distanzen wird abgeleitet, ob ein Concept Drift vorhanden ist.

Einer der ersten Ansätze zum Umgang mit Concept Drifts ist STAGGER[13]. Der STAGGER-Algorithmus weist jedem Attribut Gewichte zu. Je nach Vorkommen und Klassifikation werden diese Gewichte angepasst. Die Gewichte versuchen, das zugrundeliegende Concept zu erfassen. Zur Klassifizierung werden nur Attribute ausgewählt, welche günstige Gewichte haben. STAGGER sucht anhand der ausgewählten Attribute nach Regeln um die Trainingsdaten zu beschreiben. Der Ansatz von STAGGER ist vom Patching-Ansatz weit entfernt.

Ein weiterer Algorithmus zum Umgang mit Concept Drifts ist FLORA[20]. Der Ansatz von FLORA benutzt, ähnlich wie unser Ansatz, Windowing-Techniken. Es wird nur ein Fenster der Trainingsdaten für das Modell verwendet. Die Weiterentwicklung von FLORA, FLORA2[20] erlaubt sogar, dass die Fenstergröße variabel ist. FLORA2 benutzt also auch, wie der Patching-Algorithmus, Adaptive-Windowing-Techniken.

Der Patching-Algorithmus ist eng mit Ensemble Lernern verwandt. Der Basisklassifizierer zusammen mit den Patchklassifizierern kann als Ensemble aufgefasst werden. Man kann viele Gemeinsamkeiten zwischen dem Patching-Algorithmus und Ensemble Methoden feststellen. Der Patching-Algorithmus basiert darauf, dass ein Fehlerregionenklassifizierer trainiert wird, welcher die Fehlerregionen des Basisklassifizierers lernt. Dieser Ansatz ist ebenfalls in den Ensemble Techniken *grading*[14] und *arbitrating*[11] zu finden. Eine weitere verwandte Methode ist *stacking*[17]. Dabei werden Ensemble Lerner genutzt, um einen neuen Datensatz zu generieren, welcher sich aus den Vorhersagen der einzelnen Klassifizierer zusammen setzt. Auf diesem Datensatz wird nun ein Meta-Klassifizierer trainiert, der nun die endgültige Vorhersage der Klasse trifft. Beim *boosting*[6] wird mehrmals über den Trainingsdatensatz iteriert. Bei jeder Iteration wird ein Klassifizierer gelernt. Nach jeder Iteration werden einzelne Instanzen anhand des Klassifizierungsergebnisses neu gewichtet. Nach einer festgelegten Anzahl an Iterationen stimmen die Ensemble Klassifizierer durch *majority vote* über die endgültige Vorhersage ab. Im Gegensatz dazu werden beim *bagging*[4] Instanzen nicht gewichtet, sondern die Wahrscheinlichkeit verändert, dass eine Instanz in einer Iteration in den Trainingsdaten auftritt.

7 Fazit und Ausblick

Es wurde gezeigt, dass die vorgestellten Methoden einen Performancezuwachs bringen. Die Erweiterung durch den ADWIN-Algorithmus führt zu einer schnelleren Erholung der Klassifizierungsgenauigkeit nach Concept Drifts und Einsparungen in der Laufzeit der Algorithmen. Dies wird erreicht, indem Instanzen, welche dem vorherigen Concept folgen, aus dem Instance Store entfernt werden. Die zweite vorgestellte Erweiterung befasst sich mit der dynamischen Regulierung der Instance Store Größe. Dazu wurden zwei verschiedene Ansätze vorgestellt. Einmal wurde ein Varianzkriterium über die Klassifizierungsgenauigkeit benutzt. Dabei wurde entschieden, ob das Benutzen weiterer Batches zu einer signifikanten Verbesserung der Klassifizierungsgenauigkeit führt. Falls dies nicht der Fall ist, wird der Instance Store nicht weiter vergrößert. Als zweiter Ansatz wurde eine weitere Instanz des ADWIN-Algorithmus zur dynamischen Regulierung der Instance Store Größe genutzt. Dabei wird der Instance Store nur dann vergrößert, wenn die empfindlich eingestellte ADWIN-Instanz keine Veränderung in den Daten erkennt. In den durchgeführten Experimenten hat sich herausgestellt, dass in den meisten Fällen das Varianzkriterium zu besseren Ergebnissen führt.

Eine weitere Anpassung des Patching-Algorithmus könnte neben dem Varianzkriterium beispielsweise ein anderes Maß zur dynamischen Regulierung der Instance Store Größe sein. Ebenfalls lässt sich eventuell auch eine robustere Methode neben dem Ansatz mit zwei ADWIN-Instanzen finden. Zur Erkennung der Concept Drifts gibt es neben dem ADWIN-Algorithmus noch weitere Verfahren, die unterschiedlich arbeiten.

Literaturverzeichnis

[1] S. H. Bach und M. A. Maloof. *Paired Learners for Concept Drift*. In: *2008 Eighth IEEE International Conference on Data Mining*. Dez. 2008, S. 23–32. DOI: 10.1109/ICDM.2008.119.

[2] Albert Bifet und Ricard Gavaldà. *Learning from Time-Changing Data with Adaptive Windowing*. In: *Proceedings of the Seventh SIAM International Conference on Data Mining, April 26-28, 2007, Minneapolis, Minnesota, USA*. SIAM, 2007, S. 443–448. ISBN: 978-0-89871-630-6. DOI: 10.1137/1.9781611972771.42.

[3] Albert Bifet u. a. *MOA: Massive Online Analysis*. In: *J. Mach. Learn. Res.* 11 (Aug. 2010), S. 1601–1604. ISSN: 1532-4435.

[4] Leo Breiman. *Bagging predictors*. In: *Machine Learning* 24.2 (Aug. 1996), S. 123–140. ISSN: 1573-0565. DOI: 10.1007/BF00058655.

[5] Wei Fan. *Systematic data selection to mine concept-drifting data streams*. In: *KDD*. 2004.

[6] Yoav Freund und Robert Schapire. *A short introduction to boosting*. In: *Journal-Japanese Society For Artificial Intelligence* 14.771-780 (1999), S. 1612.

[7] João Gama u. a. *A survey on concept drift adaptation*. In: *ACM computing surveys (CSUR)* 46.4 (2014), S. 44.

[8] Mark Hall u. a. *The WEKA Data Mining Software: An Update*. In: *SIGKDD Explor. Newsl.* 11.1 (Nov. 2009), S. 10–18. ISSN: 1931-0145. DOI: 10.1145/1656274.1656278.

[9] S. Kauschke und J Fürnkranz. *Batchwise Patching of Classifiers*. In: *Proceeding 32nd AAAI Conference on Artificial Intelligence – AAAI '18*. 2018.

[10] Kyosuke Nishida und Koichiro Yamauchi. *Detecting Concept Drift Using Statistical Testing*. In: *Discovery Science*. Hrsg. von Vincent Corruble, Masayuki Takeda und Einoshin Suzuki. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, S. 264–269. ISBN: 978-3-540-75488-6.

[11] Julio Ortega, Moshe Koppel und Shlomo Argamon. *Arbitrating Among Competing Classifiers Using Learned Referees*. In: *Knowledge and Information Systems* 3.4 (Nov. 2001), S. 470–490. ISSN: 0219-1377. DOI: 10.1007/PL00011679.

[12] Sinno Jialin Pan und Qiang Yang. *A survey on transfer learning*. In: *IEEE Transactions on knowledge and data engineering* 22.10 (2010), S. 1345–1359.

[13] Jeffrey C. Schlimmer und Richard H. Granger. *Incremental learning from noisy data*. In: *Machine Learning* 1.3 (Sep. 1986), S. 317–354. ISSN: 1573-0565. DOI: 10.1007/BF00116895.

[14] Alexander K. Seewald und Johannes Fürnkranz. *An Evaluation of Grading Classifiers*. In: *Advances in Intelligent Data Analysis*. Hrsg. von Frank Hoffmann u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 115–124. ISBN: 978-3-540-44816-7.

[15] Parinaz Sobhani und Hamid Beigy. *New Drift Detection Method for Data Streams*. In: *Proceedings of the Second International Conference on Adaptive and Intelligent Systems*. ICAIS'11. Klagenfurt, Austria: Springer-Verlag, 2011, S. 88–97. ISBN: 978-3-642-23856-7.

[16] W. Nick Street und YongSeog Kim. *A Streaming Ensemble Algorithm (SEA) for Large-scale Classification*. In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '01. San Francisco, California: ACM, 2001, S. 377–382. ISBN: 1-58113-391-X. DOI: 10.1145/502512.502568.

[17] Kai Ming Ting und Ian H. Witten. *Issues in Stacked Generalization*. In: *CoRR* abs/1105.5466 (2011). arXiv: 1105.5466.

[18] Lisa Torrey und Jude Shavlik. *Transfer learning*. In: *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques* 1 (2009), S. 242.

[19] Geoffrey I. Webb u. a. *Characterizing Concept Drift*. In: *CoRR* abs/1511.03816 (2015). arXiv: 1511.03816.

[20] Gerhard Widmer und Miroslav Kubat. *Learning in the presence of concept drift and hidden contexts*. In: *Machine Learning* 23.1 (Apr. 1996), S. 69–101. ISSN: 1573-0565. DOI: 10.1007/BF00116900.