Seminararbeit über

EXTREMELY FAST DECISION TREES

1. April 2019

Pauline Speckmann
Technische Universität Dortmund
Fakultät Informatik
pauline.speckmann@udo.edu

1 Einleitung

Data-Mining[1] stellt in unserem digitalen Zeitalter ein allgegenwärtiges Problem dar. Viele Organisationen haben enorm große Datenbanken, die täglich ohne wirkliches Limit um weitere Millionen Einträge wachsen. Betrachtet man allein die kontinuierliche Expansion des Internets, kann man erwarten, dass diese Datenmengen von einer Ausnahme zur Regel werden. Entsprechend werden neue Herausforderungen für die Datenanalyse aufgeworfen, die ihre eigenen Probleme mit sich bringen [7].
Die ankommenden Daten werden zum Beispiel zu langsam verarbeitet, sodass es ungenutzte Daten gibt, deren Anzahl stetig steigt. Die triviale Lösung des Zwischenspeicherns für spätere Nutzung birgt dabei die Risiken des Datenverlusts und derer Korruption [7].

Algorithmen zum Zweck der schnellen Verarbeitung basieren meist auf Entscheidungsbaum[2]-Lernverfahren, da sie zu den effektivsten Klassifikations-Methoden gehören. Diese Arten von Lernern erstellen Modelle der Form eines Entscheidungsbaumes, wobei prinzipiell jeder Knoten des Baumes einen Test enthält, die Kanten die möglichen Testergebnisse darstellen und die Blätter eine zugehörige Vorhersage [7].
Der Schlüssel zur Effektivität der Entscheidungsbäume liegt dabei in dem Kriterium, anhand dessen die Knoten aufgeteilt werden (engl. splitting criteria measure[3]), d.h. welcher Knoten die Wurzel und welcher das Kind darstellt [8].
Das heutige Standardverfahren ist dabei die Entscheidungsbaum-Lernmethode des Hoeffding Trees. Der daraus entwickelte Very Fast Desicion Tree kann jedoch im Aspekt der Fehlerrate durch geringfügig erhöhten Berechnungsaufwand signifikant verbessert werden [11].
Relevant ist eine höhere Genauigkeit für z.B. den medizinischen Bereich, in dem Software anhand von Daten Diagnosen erstellen soll, oder auch die Börse, um die Aktienkurse besser prognostizieren zu können.

2 Weitere Ansätze

Concept-adapting Very Fast Decision Tree

Der Concept-adapting Very Fast Decision Tree (CVFDT) wurde explizit für drifting Szenarios[4] entwickelt. Er nutzt dazu ein sich bewegendes Fenster über den Eingabedaten, welches die notierten Werte an Knoten vermindert, falls diese auf Daten basieren, die aus dem Fenster heraus fallen (d.h. nicht mehr aktuell sind) [11].
Als Basis dient hier der Hoeffding Tree, wobei das Split-Kriterium[3] verändert wurde. Der CVFDT nutzt den Misclassification Error[5] anstatt der Hoeffding Bounds[6]. Die Missklassifikations-Fehlerrate wird für alle Attribute der Splits berechnet und ein existierender Split wird ersetzt, falls sein Attribut nicht länger das mit der besten Rate ist, also ein Set von alternativen Teilbäumen mit anderen Splits eine höhere Genauigkeit verbucht [8].

[1] Automatische Auswertung großer Datenmengen
[2] Darstellungsform mehrstufiger Entscheidungen
[3] Kriterium, anhand dessen die Knoten aufgeteilt werden
[4] Szenario, in dem die eingegebenen Daten ihr Muster verändern
[5] Split-Kretierium, welches die Fehlvorhersagenrate der Attribute berechnet
[6] Split-Kretierium; Wird zu einem späteren Zeitpunkt genauer betrachtet

Die Idee der Split-Reevaluation hat der Concept-adapting Very Fast Decision Tree mit dem Extremly Fast Desicion Tree (EFDT) gemeinsam. Im Gegensatz zum CVFDT wurde der EFDT jedoch für stationary Szenarien[7] entwickelt. Weiterhin baut der CVFDT parallel alternative Teilbäume, um seine eigenen gegebenfalls durch diese Alternativen zu ersetzen, während der EFDT nichts dergleichen tut [11].

Hoeffding Adaptive Tree

Die Methode des Hoeffding Adaptive Tree (HAT) baut ähnlich wie der Concept-adapting Very Fast Decision Tree einen Baum auf, der wiederum alternative Teilbäume aufbaut. Im Gegensatz zum CVFDT jedoch erst, falls ein Teilbaum für neue Beispiele schlechtere Voraussagen trifft. Ein alternativer Teilbaum wird dann eingewechselt, wenn er eine bessere Genauigkeit als das Original verbucht [11].
Als Split-Kriterium verwendet HAT ebenfalls eine Fehler-Kalkulation an jedem Knoten, namentlich den Adaptive Windowing Algorithm (ADWIN) [5]. Dabei vergleicht ADWIN die Vorhersagefehler bei neueren Datensequenzen mit der Fehlerrate einer längeren vergangenen Datensequenz. Bei einem erheblich erhöhtem Wert werden die alternative Teilbäume aufgebaut und entsprechend eingewechselt [11].
HAT basiert dabei auf seinen eigenen Vorhersageergebnissen, um auf Drifts[8] zu reagieren. Im Gegensatz dazu verlässt sich EFDT nicht auf Vorhersagen und zielt auch nicht darauf ab, Splits als Antwort auf Drifts zu ersetzen [11].

Hoeffding Tree → Very Fast Decision Tree

Das Verfahren des Hoeffding Trees, dessen Implementierung auch die Baseline für den Extremely Fast Decision Tree darstellen wird, nutzt als Split-Kriterium den Hoeffding Bound. Allgemein gesagt wird für jeden möglichen Split geprüft, ob der Unterschied der gemittelten Informationszuwächse zweier Top-Attribute mit hoher Wahrscheinlichkeit einen positiven Wert ungleich Null hat. Für diesen Fall wird der Split mit dem höheren Informationszuwachs gewählt [11].

Der Hoeffding Bound ist die Antwort auf die Frage, wie viele Beispiele an jedem Knoten benötigt werden, um eine Auswahl zu treffen. Dabei sagt er aus, dass mit einer Wahrscheinlichkeit von $1-\delta$ das wahre Mittel von r mindestens $\bar{r}-\epsilon$ für folgende Belegungen gilt [7]:

$r =$ zufällige reelle Zahl
$R =$ Spannweite[9] von r
$n =$ Anzahl der unabhängigen Beobachtungen von r
$\bar{r} =$ Mittelwert von r

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} \tag{1}$$

Darauf aufbauend wird nun ein heuristisches Maß[10] $G(X_i)$ zur Auswahl der Attribute genutzt. Für X_a mit dem höchsten und X_b mit dem zweithöchsten beobachteten $\bar{G}$ nach n Beispielen ist $\Delta\bar{G} = \bar{G}(X_a) - \bar{G}(X_b) \geq 0$ die Differenz der beiden beobachteten heuristischen Werte. Der Hoeffding Bound garantiert also für ein gewünschtes δ, dass X_a mit einer Wahrscheinlichkeit von $1-\delta$ die richtige Wahl ist, wenn n Beispiele an dem Knoten X_a gesehen wurden und $\Delta\bar{G} > \epsilon$ gilt.

[7] Szenario, in dem alle Eingabedaten das gleiche Muster haben
[8] Wechsel des Musters innerhalb der Eingabedaten
[9] Engl. range, z.B. ist für eine Wahrscheinlichkeit die Spannweite gleich eins
[10] Maß, welches trotz begrenztem Wissen und Ressourcen akzeptabel ist

Folglich muss ein Knoten die Beispiele nur so lange sammeln, bis ϵ kleiner als $\Delta\bar{G}$ wird. Insgesamt benötigt der Hoeffding Tree dadurch lediglich eine konstante Zeit pro Beispiel, während sein Ergebnisbaum fast identisch zu dem eines normalen Batch Learners[11] mit genügend Beispielen ist [7].

Der Very Fast Decision Tree ist das Lernsystem, welches direkt auf dem Hoeffding Tree basiert. Der Nachteil des Verfahrens ist dabei nicht nur die Hinauszögerung des Splits, bis die aktuell beste Option feststeht, sondern insbesondere, dass diese Entscheidung endgültig ist. Der Hoeffding Bound stellt zwar sicher, dass X_a die bessere Wahl als X_b ist, kontrolliert aber nicht das Risiko, dass ein weiteres Attribut X_c besser als X_a sein könnte. Dabei erhöht sich diese Chance mit der Gesamtanzahl der Attribute [11].

3 Hoeffding Anytime Tree

Ziele

Der Hoeffding Anytime[12] Tree verbessert den Hoeffding Tree indem er schneller lernt und für stationary Szenarien eine Konvergenz zum Batch Entscheidungsbaum[13] garantiert. Seine Implementierung, der Extremly Fast Desicion Tree, erzielt dabei in den meisten Instanzen eine signifikant höhere Genauigkeit als die Baseline, der Very Fast Desicion Tree [11].

Idee

Der Hoeffding Anytime Tree (HATT) verbessert die Kritikpunkte des Hoeffding Tree (HT), indem er einen Split durchführt, sobald er nützlich erscheint. Zu einem späteren Zeitpunkt wird diese Entscheidung noch einmal begutachtet und falls ein besserer Split verfügbar ist, wird der aktuelle durch die bessere Option ersetzt [11].

HATT verändert HT nur in der Nutzung der Hoeffding Bounds, indem er testet, ob der Split des aktuell besten Attributs keinem Split bzw. dem aktuellen vorzuziehen ist, anstatt darauf zu warten, dass das Top-Attribut das Zweitbeste übertrifft.
Einen neuen Split nimmt HATT vor, wenn der Informationszuwachs mithilfe des besten Attributs mit der erforderten Sicherheit positiv und ungleich Null ist. Ein alter Split wird ersetzt, wenn die Differenz der Informationszuwächse des aktuellen Top-Attributs und dem aktuellen Split-Attribut größer als Null ist [11].

Der Algorithmus

Der Algorithmus des Hoeffding Tree wurde in zwei Algorithmen aufgeteilt, wobei der eigentliche Split ausgelagert, und ein dritter für die Split-Reevaluation hinzugefügt wurde.
HATT erhält als Eingabe wie auch HT eine Sequenz von Beispielen S, eine Menge bestehend aus m Attributen $X = \{X_1, \ldots, X_m\}$, einen Wert für die akzeptable Wahrscheinlichkeit das falsche Split-Attribut zu wählen δ und eine Split-Evaluations-Funktion $G(.)$.
Im Gegensatz zu HT gibt HATT seinen konstruierten Entscheidungsbaum nicht explizit zurück, da durch seine Eigenschaft eines Anytime Algorithmus der Baum ohnehin immer bereit steht.

[11] Lernmethode, bei der ein komplettes Datenset auf einmal gelernt wird
[12] Ready-To-Use-Modell steht jerderzeit nach dem Betrachten der ersten paar Beispiele bereit
[13] Entscheidungsbaum, der mithilfe eines Batch Learners entsteht

Allgemein betrachtet baut der Hoeffding Anytime Tree seinen Entscheidungsbaum auf, indem er mit einem einzelnen Blatt, der Wurzel, startet und von da aus die Beispiele in die Blätter des HATT einfügt, woraufhin der Split an jedem Knoten auf dem Pfad von der Wurzel zum besagten Blatt erneut evaluiert wird.
Dabei wird zunächst der Informationszuwachs für die häufigste Klasse der Beispiele berechnet, und für alle Klassen und jeden Wert aller Attribute ein Zähler der Häufigkeit an der Wurzel auf Null gesetzt. Anschließend beginnt die erste Schleife, welche für alle gegebenen Beispiele diese zuerst in den Baum einsortiert und danach für alle Knoten auf dem Pfad von der Wurzel zu dem eingefügten Beispiel eine weitere Schleife aufruft. In dieser wird für jedes Attribut an einem Knoten zunächst der Häufigkeitszähler n_{ijk} erhöht und dann, falls der aktuelle Knoten das Blatt ist, an welchem das letzte Beispiel eingefügt wurde, die Split-Methode, oder sonst die Split-Reevaluations-Methode aufgerufen.

Hoeffding Anytime Tree $(S, X, \delta, G(.))$:

1: Let HATT be a tree with a single leaf, the root
2: Let $X_1 = X \cup X_\varnothing$
3: Let $G_1(X_\varnothing)$ be the G obtained by predicting the most frequent class in S
4: **for all** classes y_k **do**
5: **for all** values x_{ij} of each attribute $X_i \in X$ **do**
6: Set counter $n_{ijk}(\text{root}) = 0$
7: **for all** examples $(\vec{x}, y)$ in S **do**
8: Sort $(\vec{x}, y)$ into a leaf l using HATT
9: **for all** nodes in path (root ...l) **do**
10: **for all** x_{ij} in $\vec{x}$ such that $X_i \in X_{\text{node}}$ **do**
11: Increment $n_{ijk}(\text{node})$
12: **if** node = l **then**
13: AttemptToSplit(l)
14: **else**
15: ReEvaluateBestSplit(node)

Die in Zeile 9 eingesetzte for-Schleife bewirkt, im Unterschied zu HT, also nach jedem Einfügen eines neuen Beispiels die Reevaluation aller Knoten auf dem Pfad von der Wurzel zum eingefügten Knoten.
Der if-else-Block in den Zeilen 12-15 wird dabei für jeden Knoten auf dem Pfad aufgerufen und prüft, ob dieser das Blatt ist. Falls nein, so wird der Split am aktuellen Knoten reevaluiert, und falls die Bedingung erfüllt ist, wird der aktuell beste Split für das neu eingefügte Beispiel gesucht.

Im original Hoeffding Tree Algorithmus fehlen entsprechend die Zeilen 9 und 12-15, wobei letzterer Block aus dem Code der hier ausgelagerten „AttemptToSplit (leafNode l)“ Methode besteht.
In seiner letzten Zeile gibt der Hoeffding Tree den Entscheidungsbaum zurück.

In der AttemptToSplit Methode, welche mit dem Knoten des zuletzt eingefügten Beispiels aufgerufen wird, wird eben dieser mit der am häufigsten vorkommenden Klasse in ihm gelabelt. Falls alle Beispiele in diesem Knoten die gleiche Klasse haben passiert nichts weiter und die Methode endet. Falls die Bedingung jedoch nicht erfüllt ist, werden die Informationszuwächse aller Attribute des Knotens, mit Ausnahme des leeren Attributs, mit Hilfe der Häufigkeitszähler berechnet. Anschließend wird der Hoeffding Bound (1) für das Attribut mit dem höchsten Informationszuwachs X_a und das leere Attribut $X_\varnothing$ berechnet.
Falls die Differenz dieser beiden Informationszuwächse kleiner-gleich als des berechneten Hoeffding Bounds oder X_a gleich $X_\varnothing$ ist, so endet die Methode. Falls die Differenz jedoch größer als der Hoeffding Bound ist und die beiden Attribute ungleich sind, wird der aktuelle Knoten, in den das letzte Beispiel eingefügt wurde, durch einen inneren Knoten ersetzt, welcher anhand von X_a splittet. Daraufhin wird für alle Zweige des Splits ein neues Blatt angefügt und alle

Attribute außer X_a diesem zugeschrieben. Ebenso wird der Informationszuwachs der am häufigsten vorkommenden Klasse in diesen Attributen berechnet.
Für alle Klassen und alle Attributwerte des neuen Blattes, mit Ausnahme des leeren Attributs, wird der Häufigkeitszähler an diesem auf Null gesetzt und die Methode endet.

AttemptToSplit (leafNode l):

1: Label l with the majority class at l
2: **if** all examples at l are not of the same class **then**
3: Compute $\vec{G}_1(X_i)$ for each attribute $X_l - \{X_\varnothing\}$ using the counts $n_{ijk}(l)$
4: Let X_a be the attribute with the highest $\vec{G}_1$
5: Let $X_b = X_\varnothing$
6: Compute ε using equation 1
7: **if** $\vec{G}_l(X_a) - \vec{G}_l(X_b) > \varepsilon$ and $X_a \neq X_\varnothing$ **then**
8: Replace l by an internal node that splits on X_a
9: **for all** branches of the split **do**
10: Add a new leaf l_m and let $X_m = X - X_a$
11: Let $\vec{G}_m(X_\varnothing)$ be the G obtained by predicting the most frequent class at l_m
12: **for all** classes y_k and all values x_{ij} of each attribute $X_i \in X_m - \{X_\varnothing\}$ **do**
13: Let $n_{ijk}(l_m) = 0$

Die Methode wird also durch den if-else-Block der Hoeffding Anytime Tree Hauptmethode für alle Blätter des Entscheidungsbaumes aufgerufen. Dies entspricht auch der Implementierung des Hoeffding Trees, da dieser ausschließlich Blätter behandelt.
Der entscheidende Unterschied liegt in Zeile 5, in der Attributwahl für X_b. Der Hoeffding Tree verwendet das Attribut mit dem zweit höchsten Wert für $\bar{G}$, während der Hoeffding Anytime Tree das leere Attribut verwendet und damit testet, ob das Attribut X_a besser als kein Split ist.

Für HATT bietet diese Änderung insbesondere für Fälle, in denen der Informationszuwachs gleichmäßig verteilt ist, einen Lernvorteil, da HT eventuell abhängig von der Anzahl der Zufallsvariablen ist, die bei einem Gleichstand den Ausschlag zur Attributwahl geben [11].

Die Reevaluations-Methode für innere Knoten des Baums, namentlich ReEvaluateBestSplit (internalNode int), berechnet zunächst die Informationszuwächse aller Attribute, mit Ausnahme des leeren Attributs, des übergebenen Knotens mit Hilfe der Häufigkeitszähler. Auch hier wird für das Attribut mit dem höchsten Informationszuwachs X_a der Hoeffding Bound (1) berechnet, jedoch wird als zweites Attribut nicht das leere, sondern das aktuelle Split Attribut X_{current} verwendet.
Falls die Differenz der beiden Informationszuwächse kleiner-gleich dem berechneten Hoeffding Bound ist, endet die Methode. Ansonsten wird geprüft, ob die beiden verwendeten Attribute X_a und X_{current} gleich sind. Für diesen Fall wird der aktuelle Knoten durch ein Blatt ersetzt und der gesamte Teilbaum wird gelöscht. Andernfalls wird der aktuelle Knoten durch einen anderen Knoten, welcher anhand von X_a splittet, ersetzt und eine Schleife für alle Zweige des Splits aufgerufen.
In dieser wird nun genau wie am Ende der AttemptToSplit Methode der Informationszuwachs der am häufigsten vorkommenden Klasse in diesem Knoten berechnet, sowie für alle Klassen und alle Attributwerte, mit Ausnahme des leeren Attributs, der entsprechende Häufigkeitszähler auf Null gesetzt.

ReEvaluateBestSplit (internalNode int):

1: Compute $\vec{G}_{\text{int}}(X_i)$ for each attribute $X_{\text{int}} - \{X_\varnothing\}$ using the counts n_{ijk}(int)
2: Let X_a be the attribute with the highest $\vec{G}_{\text{int}}$
3: Let X_{current} be the *current* split attribute
4: Compute ε using equation 1
5: **if** $\vec{G}_l(X_a) - \vec{G}_l(X_{\text{current}}) > \varepsilon$ **then**
6: **if** $X_a = X_\varnothing$ **then**
7: Replace internal node int with a leaf (kills subtree)
8: **if** $X_a \neq X_{\text{current}}$ **then**
9: Replace int with an internal node that splits on X_a
10: **for all** branches of the split **do**
11: Add a new leaf l_m and let $X_m = X - X_a$
12: Let $\vec{G}_m(X_\varnothing)$ be the G obtained predicting the most frequent class at l_m
13: **for all** classes y_k and all values x_{ij} of each attribute $X_i \in X_m - \{X_\varnothing\}$ **do**
14: Let $n_{ijk}(l_m) = 0$

Da diese Methode das neue Konzept der Split Reevaluation verwirklicht und nicht im Hoeffding Tree Algorithmus vorkommt, ist sie trivialer Weise ein einziger Unterschied zum Original. ReEvaluateBestSplit hat jedoch viele Gemeinsamkeiten mit der AttemptToSplit Methode.

Erstens, die Berechnung des Hoeffding Bounds für das Attribut mit dem größten Informationszuwachs und einem weiteren Attribut.
Zweitens, die Abfrage der Differenz der beiden Informationszuwächse und dem Ergebnis des Hoeffding Bounds, sowie der Gleichheit der beiden verwendeten Attribute.
Drittens, die Berechnung des Informationszuwachses der am häufigsten vorkommenden Klasse des Knotens und die Setzung der Häufigkeitszähler am Ende der Methode.

Komplexität

Die Speicherkomplexität besteht aus dem benötigten Speicher und der Worst-Case-Komplexität. Um für jeden Knoten die Statistiken zu speichern, benötigen sowohl HT als auch HATT $\mathcal{O}(dvc)$ Speicher, mit d Attributen, v Werten pro Attribut und c Klassen. Da sich die Anzahl der Knoten geometrisch erhöht, ergibt sich die maximale Anzahl von $(1 - v^d)/(1 - v)$ Knoten mit der entsprechenden Worst-Case-Laufzeit von $\mathcal{O}(v^{d-1}dvc)$.
Die Worst-Case-Komplexität für HT in Abhängigkeit der aktuellen Anzahl an Blättern l ergibt sich als $\mathcal{O}(ldvc)$, sodass die Komplexität für HATT mit n als Knotenanzahl vergleichbar als $\mathcal{O}(ndvc)$ geschrieben werden kann. Durch $l \in \mathcal{O}(n)$ ist die Speicherkomplexität für HT und HATT äquivalent [11].

Die Zeitkomplexität ergibt sich aus zwei primären Operationen, welche mit dem Lernen für den Hoeffding Tree assoziiert werden. Zum einen das Aufbauen eines Trainingsbeispiels durch Erhöhung der Blatt Statistiken und zum anderen die Evaluation potentieller Splits an dem Blatt, in welches das Beispiel eingefügt wurde. Der Hoeffding Anytime Tree verfügt über die gleichen beiden Primäroperationen, wobei dieser auch die Statistiken der inneren Knoten erhöht und ebenso die potentiellen Splits für diese auf dem Pfad zum relevanten Blatt evaluiert.
Für alle Blätter des HT und alle Knoten des HATT müssen nie mehr Evaluationen als für ihre d Attribute betrachtet werden. An jedem Attribut müssen v Informationszuwächse berechnet werden und für jeden Informationszuwachs werden $\mathcal{O}(c)$ arithmetische Operationen benötigt, sodass jede Split-Reevaluation an jedem Knoten $\mathcal{O}(dvc)$ arithmetische Operationen erfordert. Um ein Beispiel zu integrieren müssen an jedem Knoten, welchen das Beispiel durchläuft, dvc

Updates gemacht werden und damit $\mathcal{O}(dvc)$ korrelierende arithmetische Operationen.
Die Kosten zum Updaten der Knoten Statistiken für HATT beläuft sich auf $\mathcal{O}(hdvc)$, mit h als maximale Tiefe des Baumes, da bis zu h Knoten von dem Beispiel durchlaufen werden, während HT durch die ausschließliche Betrachtung der Blätter nur ein Set der Statistiken aktualisiert muss und dafür $\mathcal{O}(dvc)$ benötigt. Im schlimmsten Fall belaufen sich die Kosten der Split Evaluationen an jedem Zeitschritt auf $\mathcal{O}(dvc)$ für HT und $\mathcal{O}(hdvc)$ für HATT, da ein Blatt beziehungsweise ein Pfad evaluiert werden muss [11].

4 Experimente

Daten

Zwischen dem Extremly Fast Decision Tree und dem Very Fast Decision Tree wurden insgesamt 26 Vergleiche gezogen, in dem für beide Modelle die Fehlerrate beim, und benötigte Zeit zum Verarbeiten verschiedener Datensätze gemessen wurde.
Verwendet wurden alle UCI [10] Datensätze mit über 200.000 Instanzen, welche ein offensichtliches Klassifikationsziel besitzen, keine fehlenden Werte beinhalten und kein text mining[14] benötigen. Um diese begrenzte Sammlung an großen Datensets auszugleichen wurde zudem noch der WISDM [9] Datensatz verwendet.
Insgesamt wurden zwölf Benchmark[15] Datensätze mit einer Mischung von numerischen und nominalen Attributen zwischen ein paar und hunderten Dimensionen für die Tests verwendet. Weiterhin gibt es zwei Vergleiche der Entwicklung der prequential[16] Fehler, deren Daten mit dem MOA RandomTreeGenerator mit fünf Klassen, fünf nominalen Attributen, fünf Werten pro Attribut und dem zehnfachen Mittel generiert wurden [11].

Da sowohl VFDT als auch EFDT für stationary Szenarios gegen den Batch Entscheidungsbaum konvergieren, aber die geordneten UCI Datensets diese Szenarien nicht wiedergeben, wurden die Daten zehn Mal mit der Unix *shuf* Utility[1] und der Hilfe von reproduzierbaren Zufallsbytes gemischt, wobei diese zehn Ergebnisse anschließend gemittelt wurden [11].
Alle Tests sind reproduzierbar. Eine Anleitung, die Quellcodes für EFDT und auch VFDT, sowie ein Python Skript zum Ausführen der Experimente wird bereitgestellt unter:

`https://github.com/chaitanya-m/kdd2018`

Performanz

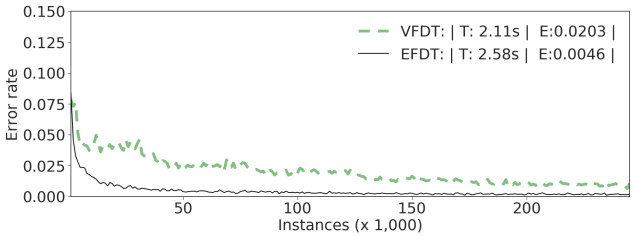
Der Extremely Fast Decision Tree erreicht eine erheblich höhere prequential Genauigkeit im Gegensatz zum Very Fast Decision Tree für die meisten Datenmengen, unabhängig davon, ob diese gemischt sind oder nicht. Die Differenzen in der Performanz für diese (un-)gemischten Datensätze heben insbesondere die Menge der Ordnung gut hervor, die in ungemischten Datenmengen vorhanden ist [11].

Der ungemischte Skin (Engl. für „Haut“) Datensatz beinhaltet RGB Werte und eine Zielvariable, die andeutet, ob die Eingabe zur Haut korrespondiert oder nicht. Alle positiven Beispiele befinden sich am Anfang, gefolgt von allen negativen Beispielen am Ende.
In dem Beispiel 1b lässt sich sehr gut beobachten, wie sowohl VFDT als auch EFDT einmal

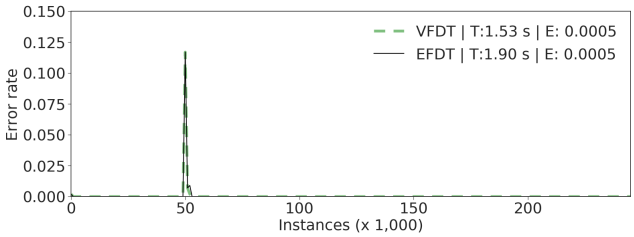
[14] Bündel von Analyse-Algorithmen zur Entdeckung von Bedeutungsstrukturen aus Textdaten.
[15] Genormte Mess- und Bewertungsverfahren
[16] Mischung der Worte „voraussagend“ (engl. predictive) und „sequenziell“ (engl. sequential)

umlernen müssen, sobald sich das Muster der Eingaben verändert.

Der sogenannte Netz Effekt beschreibt das Ersetzen eines extrem simplen Konzepts mit einem anderen innerhalb eines Lerners. Bei gemischten Datenmengen ist es nötig, ein komplexeres Entscheidungskonzept zu lernen, was die Performanz beider Lerner beeinflusst [11].



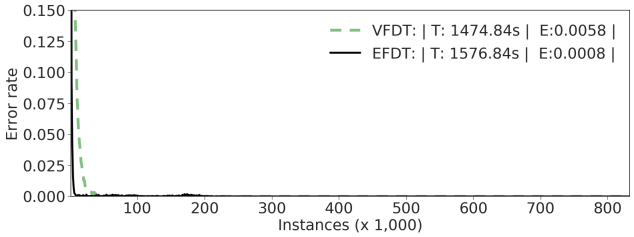
(a) Zehn Mal gemischt und gemittelt.



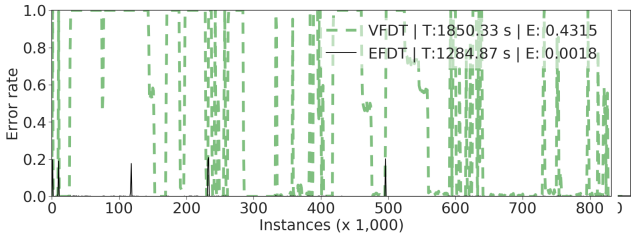
(b) Ungemischt.

Abbildung 1: „Skin“ Datenmenge [4, 10]

Ein weiterer Effekt kann bei einem mehrdimensionalen Datensatz, hier beispielsweise die Font (Engl. für „Schriftart“) Datenmenge, beobachtet werden. Ziel ist das Vorhersagen der Schriftart von 153 Möglichen in einem 19×19 Graustufenbild, wobei jeder Pixel die Möglichkeit hat, 255 verschiedene Intensitätswerte anzunehmen.
Wenn die Instanzen alphabetisch anhand ihres Schriftartennamens sortiert sind, muss der VFDT immer, wenn eine neue Schriftart an der Reihe ist, an jedem Blatt ein neues Konzept lernen, während die Baumkomplexität steigt. Im Gegensatz dazu kann der EFDT sein Modell zurechtrücken, indem er effektiv veraltete Splits verwirft, um eine Genauigkeit von ca. 99.8% zu erzielen. Diese Eigenschaft macht den EFDT zu einen potentiell mächtigen Lerner für Methoden, die für drifting Szenarien entworfen wurden.



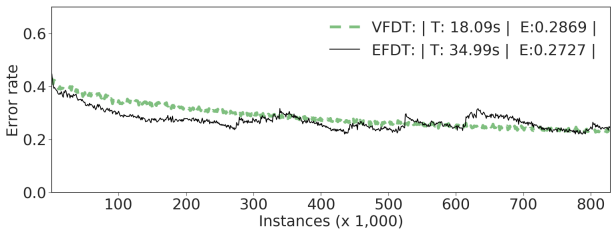
(a) Zehn Mal gemischt und gemittelt.



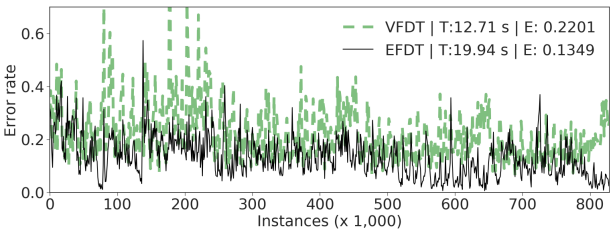
(b) Ungemischt.

Abbildung 2: „Font“ Datenmenge [10]

Die Ergebnisse der Poker und Forest-Covertype Datenmengen reflektieren beide Arten der Effekte: Der EFDT leistet erheblich bessere Resultate auf geordneten Daten und die Performanz beider Lerner verschlechtert sich im Vergleich zu ersterem für gemischte Daten.
Jede zusätzliche Ebene eines Entscheidungsbaumes fragmentiert dabei den Eingabespeicher, was den Baumwuchs exponentiell verlangsamt. Eine Verzögerung des Splits in einem Level führt zu einer Verzögerung des Starts zum Informationen sammeln für die Splits auf dem nächsten Level [11].

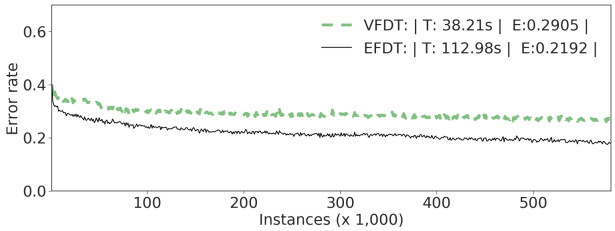


(a) Zehn Mal gemischt und gemittelt.

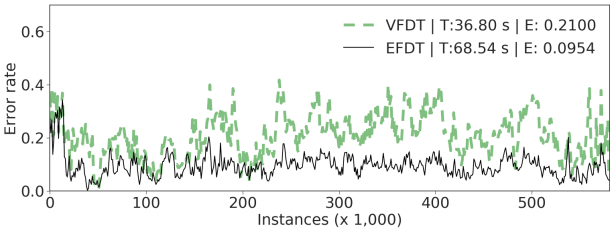


(b) Ungemischt.

Abbildung 3: Poker Datenmenge [10]



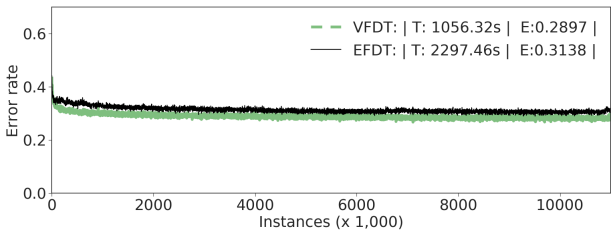
(a) Zehn Mal gemischt und gemittelt.



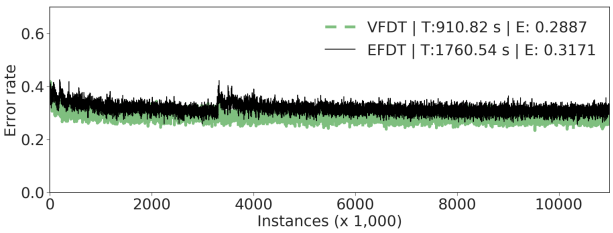
(b) Ungemischt.

Abbildung 4: „Forest covertype“ Datenmenge [6, 10]

Generell kann dadurch erwartet werden, dass HATT einen Vorteil gegenüber HT in Situationen hat, in denen letzterer die Splits auf jeder Ebene deutlich verzögert, z.B. wenn der Informationszuwachs zwischen den Top-Attributen zu niedrig ist, sodass eine bedeutende Anzahl an Beispielen benötigt wird um den Hoeffding Bound zu überwinden.
Auf der anderen Seite bedeutet dies, falls der Informationszuwachs sowohl zwischen den Top-Attributen, als auch der Zuwachs selbst niedrig ist, wählt HATT möglicherweise einen Split, welcher viele neue Beispiele zum Korrigieren benötigt [11].
Ein Beispiel für den Worst-Case des HATT bieten synthetische Daten aus Physik Simulationen:

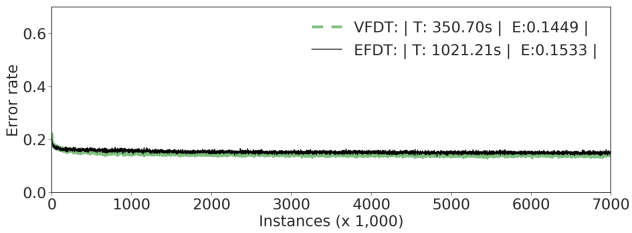


(a) Zehn Mal gemischt und gemittelt.

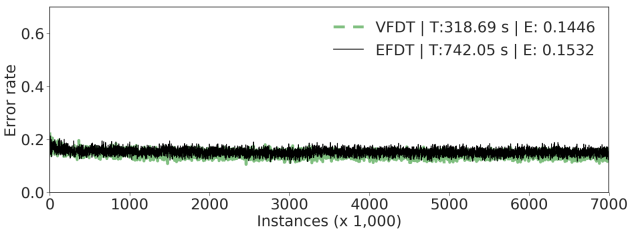


(b) Ungemischt.

Abbildung 5: „Higgs“ Datenmenge [3, 10]



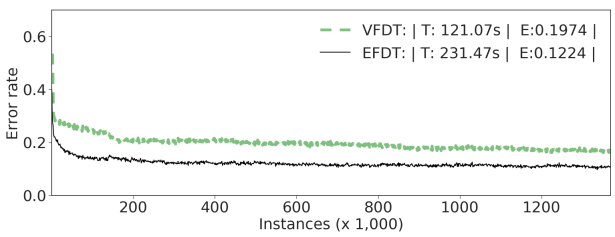
(a) Zehn Mal gemischt und gemittelt.



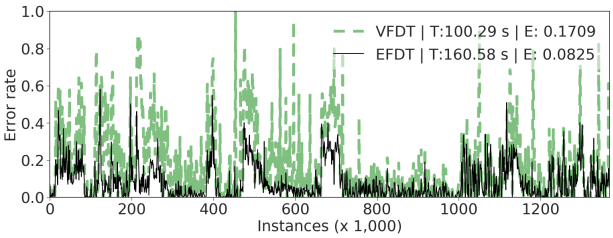
(b) Ungemischt.

Abbildung 6: „Hepmass“ Datenmenge [2, 10]

Einen weiteres interessantes Ergebnis bietet der Vergleich auf dem WISDM Datensatz, da in diesem Fall für die ungemischte Datenmenge klar zu sehen ist, wie die beiden Lerner in gleichem Maße auf neue Muster reagieren.



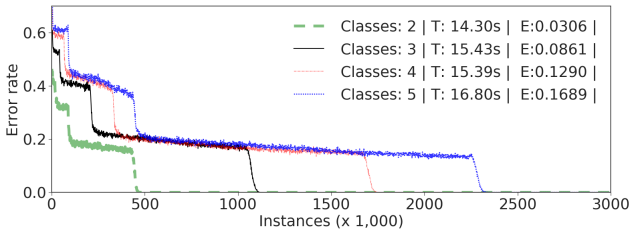
(a) Zehn Mal gemischt und gemittelt.



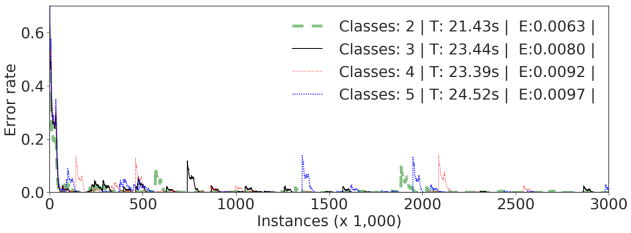
(b) Ungemischt.

Abbildung 7: „WISDM“ Datenmenge [9, 10]

Einen guten Gesamtvergleich zeigen folgende Grafiken, in denen die Entwicklung der prequential Fehlerrate für jeden Lerner mit Hilfe vier verschiedener Komplexitätsklassen gezeigt wird. Diese Daten wurden mit dem MOA RandomTreeGenerator mit fünf Klassen, fünf nominalen Attributen, fünf Werten pro Attribut und dem zehnfachen Mittel generiert. Der Langzeit Vergleich nutzt dabei eine Million Beispiele [11].

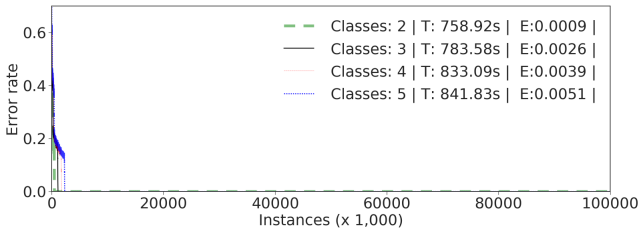


(a) VFDT.

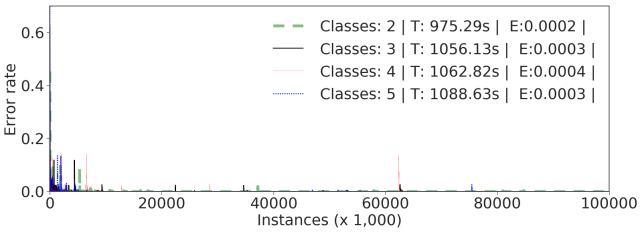


(b) EFDT.

Abbildung 8: Direkter Vergleich von VFDT und EFDT [11]



(a) VFDT.



(b) EFDT.

Abbildung 9: Langzeit Vergleich von VFDT und EFDT [11]

5 Eigene Wertung

Durch kleine Änderungen am Algorithmus des Hoeffding Trees konnte mit dem Hoeffding Anytime Tree eine signifikant bessere Genauigkeit erzielt werden. In den 18/24 Tests auf den UCI Datensätzen schneidet HATT im Aspekt der Fehlerrate nur in einem Fall genau so ab wie HT, ansonsten besser. Die anderen sechs Testfälle waren explizit darauf ausgelegt, das Worst-Case-Szenario des HATT zu testen, und selbst hier war HT nur minimal besser.
Die vom EFDT benötigte Zeit steigt dabei jedoch in 11/24 Fällen auf ca. das Doppelte, in zwei weiteren auf über doppelt so viel, und in einem Fall benötigt EFDT sogar fast drei Mal so viel Zeit wie der VFDT (Abbildung 4a). Der Name „Extremely Fast Decision Tree“ ist also denkbar unpassend.

Obwohl HATT nicht darauf abgezielt hat, scheint er eine Toleranz gegenüber drifting Szenarien zu haben, was ihn auch zu einem interessanten Konkurrenten für Methoden wie z.B. den Concept-adapting Very Fast Decision Tree und den Hoeffding Adaptive Tree macht.
Durch das Dasein des EFDT als Anytime Algorithmus hat er außerdem deutliche Vorteile, falls ein Baum nicht zu ende gelernt hat.

Insgesamt stellt der Extremely Fast Decision Tree eine solide Alternative zum Very Fast Decision Tree da. Die höhere Genauigkeit des EFDT ist nicht von der Hand zu weisen und daher für Anwendungen, die viele Daten schnell, aber nicht sehr schnell, jedoch sehr akkurat verarbeiten müssen, nur zu empfehlen.

Literatur

[1] *GNU Coreutils: Random sources*. https://www.gnu.org/software/coreutils/manual/html_node/Random-sources.html. Version: 2018

[2] Baldi, P. ; Cranmer, K. ; Faucett, T. ; Sadowski, P. ; Whiteson, D. : Parameterized neural networks for high-energy physics. In: *The European Physical Journal C* 76 (2016), Nr. 5, S. 235

[3] Baldi, P. ; Sadowski, P. ; Whiteson, D. : Searching for exotic particles in high-energy physics with deep learning. In: *Nature communications* 5 (2014), S. 4308

[4] Bhatt, R. ; Abhinav, D. : *Skin Segmentation Dataset: UCI Machine Learning Repository*. 2012

[5] Bifet, A. ; Gavalda, R. : Learning from time-changing data with adaptive windowing. In: *Proceedings of the 2007 SIAM international conference on data mining* SIAM, 2007, S. 443–448

[6] Blackard, J. A. ; Dean, D. J.: Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. In: *Computers and electronics in agriculture* 24 (1999), Nr. 3, S. 131–151

[7] Domingos, P. ; Hulten, G. : Mining high-speed data streams. In: *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '00* (2000). http://dx.doi.org/10.1145/347090.347107. – DOI 10.1145/347090.347107. ISBN 1581132336

[8] Hulten, G. ; Spencer, L. ; Domingos, P. : Mining time-changing data streams. In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining* ACM, 2001, S. 97–106

[9] Kwapisz, J. R. ; Weiss, G. M. ; Moore, S. A.: Activity recognition using cell phone accelerometers. In: *ACM SigKDD Explorations Newsletter* 12 (2011), Nr. 2, S. 74–82

[10] Lichman, M. : *UCI Machine Learning Repository*. http://archive.ics.uci.edu/ml. Version: 2013

[11] Manapragada, C. ; Webb, G. I. ; Salehi, M. : Extremely Fast Decision Tree. In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '18* (2018). http://dx.doi.org/10.1145/3219819.3220005. – DOI 10.1145/3219819.3220005. ISBN 9781450355520

[12] Reiss, A. ; Stricker, D. : Introducing a new benchmarked dataset for activity monitoring. In: *2012 16th International Symposium on Wearable Computers* IEEE, 2012, S. 108–109

Begriffsdefinitionen

1) **Data-Mining** bezeichnet die automatische Auswertung großer Datenmengen mit dem Zweck Muster und Trends zu erkennen.
2) Ein **Entscheidungsbaum** ist ein Graph, der durch Verzweigungen den Verlauf mehrstufiger Entscheidungen und ihrer Konsequenzen darstellt.
3) Das **Split Kriterium** (engl. splitting criteria measure) ist ein Maß, anhand dessen entschieden wird, wie ein Knoten aufzuteilen ist.
4) **Drifting Szenarien** beschreiben eine Situation, in der die Eingabe-Datenmenge ihr Muster verändert.
5) Der **Misclassification Error** ist eine mathematische Methode zur Auswahl des optimalen Attributs für einen Split, den man wie folgt repräsentieren kann:

$$G(S) = 1 - \frac{\max_{k\in\{1,\dots,k\}}\{n^k(S)\}}{n(S)}$$

   Mit $G(S)$ als Missklassifikations Fehler von S, k als Anzahl der Klassen, $n(S)$ als Kardinalität von S und $n^k(S)$ als Anzahl der Elemente der k-ten Klasse des Set S.
6) **Hoeffding Bounds** werden auf S. 3 erläutert.
7) **Stationary Szenarien** beschreiben eine Situation, in der die Eingabe Datenmenge ihr Muster stets beibehält.
8) Ein **Drift** beschreibt den Wechsel des Musters innerhalb der Eingabedaten.
9) Die **Spannweite** (engl. range) ist z.B. für eine Wahrscheinlichkeit gleich eins und für einen Informationszuwachs $\log c$, mit c als Anzahl der Klassen.
10) Ein**heuristisches Maß** basiert auf einem heuristischen Verfahren, welches mit begrenztem Wissen und begrenzten Ressourcen akzeptable Lösungen erzielt.
11) Ein **Batch Learner** ist eine Methode des maschinellen Lernens, bei der das gesamte Datenset direkt zur Verfügung steht und gelernt wird.
12) Ein **Anytime Algorithmus** stellt eine valide Lösung (ein Ready To Use Modell) bereit, selbst wenn er vor dem Beenden unterbrochen wird.
13) Ein **Batch Entscheidungsbaum** entsteht als Ergebnis eines Batch Learners, welcher alle alle Daten direkt zur Verfügung hat.
14) **Text-Mining** beschreibt Algorithmen zur Analyse von Textdaten, um Strukturen und Muster darin zu erkennen.
15) **Benchmark Datensätze** sind Datenmengen, deren Daten durch genormte Mess- und Bewertungsverfahren gesammelt wurden.
16) Das Wort „**prequential**“ hat noch keine reguläre deutsche Übersetzung. Es setzt sich aus den englischen Worten „predicitve“ und „sequential“ zusammen.