Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

The CART of Prolog: Implementierung von Entscheidungsbäumen mittels logischer Programmierung

Bachelorarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von
Heiko Kauschke

Beginn der Arbeit: 02. Juni 2022
Abgabe der Arbeit: 02. September 2022

Erstgutachter: Prof. Dr. Michael Leuschel
Zweitgutachter: Prof. Dr. Stefan Conrad

Selbstständigkeitserklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 02. September 2022

Heiko Kauschke

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Frage, wie eine Implementierung des CART-Algorithmus und ein darauf aufgebauter Random-Forest-Algorithmus, in der logischen Programmiersprache Prolog, aussehen könnte. Zusätzlich wird auch die Schnelligkeit und Genauigkeit dieser Implementierung mit den Implementierung aus der Python-Bibliothek scikit-learn und den R-Bibliotheken rpart und randomForest verglichen. Prolog ist eine beliebte Programmiersprache, die vorallem im Bereich der künstlichen Intelligenz gerne verwendet wird. Allerdings existiert nur wenig Unterstützung für die Sprache, um Algorithmen, aus dem Bereich des maschinellen Lernens, einem Teilgebiet des künstlichen Intelligenz, direkt anzuwenden. Es wird sich leider herausstellen, dass diese Implementierung noch nicht ausgereift genug ist, um im Rahmen der Schnelligkeit mit den Referenzimplementierung mithalten zu können.

Danksagung

Ich möchte mich herzlichst bei Jannik Dunkelau für die gute Betreuung bedanken. Zusätzlich danke ich allen, die meine Arbeit Probe gelesen haben und mir geholfen haben sie stets ein Stück besser zu machen.

Inhaltsverzeichnis

1 Einleitung

Das Schlagwort maschinelles Lernen hat durch die heutzutage zur Verfügung stehende Rechenleistung, immer mehr Aufmerksamkeit gefunden. Zudem ist durch das Internet die Sammlung von Daten sehr viel leichter geworden und es stehen ausreichend Daten zur Verfügung, die analysiert werden können. Um diese Daten zu verarbeiten werden gerne Klassifikationsalgorithmen benutzt. Diese finden nicht nur Verwendung im Bereich des Data-Mining [SK16], sondern auch in klinischen Studien [Lew00] und der psychologischen Forschung [SMT09]. In diesen Feldern wird für die Klassifikation ein sogenannter Entscheidungsbaum benutzt. Entscheidungsbäume sind in den eben genannten Gebieten beliebt, da sie sich leicht grafisch darstellen lassen und die Regeln, nach denen klassifiziert wird, ablesbar und leicht verständlich sind. Die zwei bekanntesten Algorithmen zur Erstellung von Entscheidungsbäumen sind schon lange im Umlauf. Der CART-Algorithmus [BFSO84] wurde von Breiman im Jahr 1984 und C4.5, von Quinlan, im Jahr 1993 [Qui93] vorgestellt. CART lässt sich sogar für sogenannte Regressionsprobleme verwenden, wodurch für eine Eingabe auch ein stetiger Wert vorhergesagt werden kann anstatt einer Klasse. Dieser Algorithmus ist folglich auch in verschiedenen Bibliotheken für maschinelles Lernen, in verschiedenen Programmiersprachen, implementiert. Allerdings nicht in Prolog.

Prolog [CR93] ist eine logische Programmiersprache und spielt eine große Rolle beim Bau von Expertensystemen [Mer12] und wird heutzutage gerne im Bereich der künstlichen Intelligenz [Sho14] und diversen anderen Feldern [LSW15, WS16] eingesetzt. Diese Programmiersprache besitzt keine Standard Bibliothek für Algorithmen des maschinellen Lernens, trotz seiner wichtigen Rolle in der künstlichen Intelligenz. Da der CART-Algorithmus rekursiv ist, macht es durchaus Sinn diesen Algorithmus, in Prolog umzusetzen, da diese Programmierspache hervorragend mit Rekursion umgehen kann. Zusätzlich würde eine Implementierung eines Entscheidungsbaums in Prolog zeigen, ob es sinnvoll ist Machine-Learning-Algorithmen in puren Prolog zu implementieren.

In dieser Arbeit soll untersucht werden, wie gut eine pure Prolog Implementierung des Entscheidungsbaum Algorithmus CART, im Vergleich zu den weit verbreiteten Referenzimplementierungen aus Python und R ist. Dafür wird zunächst das benötigte Wissen über die benutzten Algorithmen und Prolog eingeführt. Daraufhin wird die Implementierung und anschließend die Ergebnisse eines Performancevergleichs vorgestellt.

2 Grundlagen

In diesem Kapitel werden die benötigten Konzepte und Algorithmen erklärt, die für das Verständnis der Arbeit notwendig sind. In Abschnitt 2.1 wird auf die grundlegende Funktionsweise von Entscheidungsbaum-Algorithmen, in speziellem den CART-Algorithmus, eingegangen. Daraufhin wird in Abschnitt 2.2 das Vorgehen von Random-Forest-Algorithmen beschrieben. In Abschnitt 2.3 wird die Programmiersprache Prolog kurz eingeführt.

2.1 Entscheidungsbaum

Entscheidungsbaumalgorithmen sind sogenannte nicht-parametrische Methoden, das heißt, dass vor der Entstehung des Baums keine Parameter vorgegeben werden müssen und die Struktur des Baums von den Daten abhängt. Ein Entscheidungsbaum ist eine hierarchische Struktur die aus internen Entscheidungsknoten und terminalen Blättern besteht. Jeder Entscheidungsknoten beinhaltet eine einfache Funktion mit diskreten Ergebnissen, die die Zweige repräsentieren. Diese Funktionen werden so gewählt, dass sie den Eingaberaum in immer kleinere Regionen unterteilen, welche wiederum unterteilt werden, je weiter man den Pfad von der Wurzel nach unten folgt. Ein terminales Blatt entsteht dann, wenn die lokale Region im Eingaberaum nicht weiter unterteilt werden soll. Im Falle von Klassifikationsbäumen ist das meist der Fall, wenn die Region nur aus den gleichen diskreten Werten besteht, während es bei Regressionsbäumen sehr ähnliche stetige Werte sind. Ein beispielhafter Entscheidungsbaum und die dazugehörige Unterteilung des Eingaberaums ist in Abbildung 1 [Alp22] dargestellt. Bei gegebener Eingabe wird nun an jedem Knoten

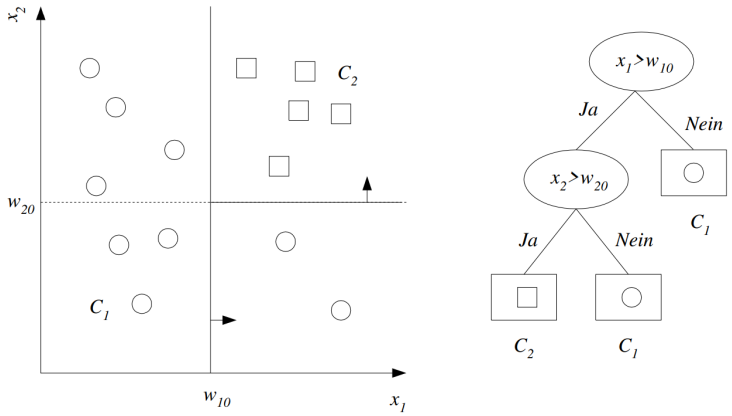

Abbildung 1: beispielhafter Entscheidungsbaum und Unterteilung. Ovale sind Entscheidungsknoten, Vierecke sind Blattknoten.

das Ergebnis der entsprechenden Funktion berechnet. In Abhängigkeit von dem Ergbnis wird daraufhin der Zweig ausgewählt, welcher entlang gegangen wird. Das wird so lange wiederhohlt, bis man an in einem Blattknoten angekommen ist. Der in diesem Blatt stehende Wert wird dann als Ausgabe benutzt.

In dieser Arbeit geht es um den CART Algorithmus, deswegen wird im Folgenden nur die typische Vorgehensweise [Cra89] für diesen Algorithmus vorgestellt. Diese ist in Algorithmus 1 exemplarisch dargestellt. Der Baum besteht ursprünglich nur aus einen Wurzelknoten und beinhaltet den gesamten Datensatz T. Als erstes wird die Menge aller möglichen binären Aufteilungen, von T, durchsucht, bis eine optimale gefunden wurde. Was eine optimale Aufteilung ist, wird dabei durch ein Maß der Unreinheit bestimmt, welches minimiert werden muss. Typischerweise benutzt der CART Algorithmus den Gini-Index für Klassifikation, abgebildet in Gleichung (1). Im Falle der Regression wird meist die Residuenquadratsumme

Algorithmus 1 Training eines Entscheidungsbaums

```
1: function CreateTree(Data)
2:     if Abbruchbedingung(D) = true then
3:         label ← Klasse(D)
4:         return label
5:     end if
6:     bestValue ← ∞
7:     bestSplit ← NIL
8:     splits ← erstelleSplitliste(D)
9:     for all s ∈ splits do
10:        value ← KombiniertGini(D, s)
11:        if value < bestValue then                ▷ Aktualisiere besten Wert
12:            bestValue ← value
13:            bestSplit ← s
14:        end if
15:    end for
16:    leftTree, rightTree ← aufteilen(D,bestSplit)
17:    leftRes ← CreateTree(leftTree)
18:    rightRes ← CreateTree(rightTree)
19:    return (s, leftRes, rightRes)
20: end function
```

benutzt, abgebildet in Gleichung (2).

$$Gini(T) = 1 - \sum_{i=1}^{n} p_i^2 \tag{1}$$

$$RSS(T) = \sum_{i=1}^{n} (\bar{y} - x_i)^2 \tag{2}$$

Hier ist p_i die relative Häufigkeit von der i-ten Klasse in T, während $\bar{y}$ das arithmetische Mittel der Zielvariablen in T ist und x_i die Zielvariable des i-ten Datenpunkts in T. Diese Maße werden benutzt um die Reinheit eines einzelnen Knoten zu messen. Um eine optimale Aufteilung zu finden, zum Beispiel, im Fall der Klassifikation mit Gini-Unreinheit, muss Gleichung (3) minimiert werden.

$$Gini(T)_{Split} = N_1/N * Gini(T_1) + N_2/N * Gini(T_2) \tag{3}$$

Hier ist der Datensatz T in zwei Teile, T_1 und T_2, aufgeteilt und N_1 und N_2 geben die Anzahl an Datenpunkten in der jeweiligen Teilmenge an. Eine Aufteilung durch stetige Attribute nimmt dabei eine Regel der Form, $A \leq r$ an, wobei r die Mitte von zwei beliebigen, unterschiedlichen Datenpunkten ist. Für ein kategorisches Attribut, welches die Werte D_1 bis D_n annehmen kann, nimmt die Aufteilung die Form $A \in D_T$ an, wobei D_T eine Teilmenge von $\{D_1,...,D_n\}$ ist. Nachdem eine optimale Aufteilung gefunden wurde, werden zwei Nachfolgerknoten, für den Wurzelknoten, erstellt. Der linke enthält die Datenpunkte aus dem Datensatz, die die Regeln erfüllen, während der rechte den restlichen Datensatz enthält.

Dieser Prozess nun wird rekursiv auf jedes Blatt angewendet, bis eine Abbruchbedingung erreicht ist. Eine Abbruchbedingung kann einsetzen, wenn der Datensatz in einem Blatt zu klein ist, nur noch eine Klasse in allen Datenpunkten vertreten ist oder sich die Datenpunkte nicht mehr unterscheiden lassen. Da ein Entscheidungsbaum in den meisten Fällen unter „overfitting“ leidet, wird dieser am Ende gestutzt. Das Pruning ist ein komplexer Vorgang, der nicht in meiner Implementierung angewandt wurde, deswegen wird hier nicht näher darauf eingegangen.

2.2 Random Forest

Nach Breiman ist ein Random Forest [Bre01] ein Klassifikator der aus einer Sammlung von baumartigen Klassifikatoren besteht, welche aus unabhängig identisch verteilten Zufallsvektoren erstellt wurden, und jeder Baum stimmt für die beliebteste Klasse der Eingabe. Random Forest gehört zu den sogenannten „Ensemble Methoden“. Der Gedanke hinter diesen Methoden ist es, das Problem der Instabilität von Entscheidungsbäumen zu lösen [SMT09], indem, bei Vorhersagen, der Durchschnitt von mehreren Bäumen genommen wird. Durch diese Technik soll auch das Problem gelöst werden, dass Entscheidungsbäume sehr empfindlich gegenüber Änderungen im Trainingsdatensatz reagieren.

Ein Random-Forest-Algorithmus besteht aus zwei Hauptbestandteilen: Bagging [Bre96] und Feature Randomness. Bagging („bootstrap aggregating“) ist ein Verfahren das 1996 von Breiman vorgestellt wurde um die Präzision von instabilen Verfahren, wie Entscheidungsbäumen, zu verbessern. Beim bagging werden aus einem N großen Datensatz, N zufällige Instanzen mit zurücklegen gezogen. Diese N Instanzen werden dann als Datensatz benutzt um einen Klassifikator zu trainieren. Dieser Vorgang wird mehrfach wiederholt, bis man genug Klassifikatoren trainiert hat. Um eine Vorhersage zu treffen, wird eine Vorhersage von allen Klassifikatoren getroffen. Anschließend wird für die finale Vorhersage ein Mehrheitsentscheid getroffen, wobei alle Klassifikatoren eine Stimme haben. Bei einem Random Forest sind diese Klassifikatioren Entscheidungsbäume.

Jedoch werden die Klassifikatoren nicht nur mit einem zufällig erzeugten Datensatz trainiert. Bei dem Training von Entscheidungsbäumen in einem Random Forest wird zudem Feature Randomness eingesetzt. Das heißt, dass bei der Konstruktion nicht alle Attribute der Instanzen im Datensatz genutzt werden. Stattdessen wird voher eine zufällige Teilmenge aller Attribute erstellt, die während des Trainings benutzt werden. Wie groß die Teilmenge ist, hängt davon ab, ob ein Klassifikationsbaum oder Regressionsbaum trainiert wird. In „The Elements of Statistical Learning: Data Mining, Inference, and Prediction“ wird vorgeschlagen, dass man im Falle der Klassifikation Gleichung (4) und der Regression Gleichung (5) benutzt. Hier ist m die Anzahl der Attribute, die eine Instanz besitzt.

$$M_{Klassifikation} = \lfloor \sqrt{m} \rfloor \tag{4}$$

$$M_{Regression} = \lfloor m/3 \rfloor \tag{5}$$

Somit wird in einem Random-Forest-Algorithmus in jedem Schritt ein Entscheidungsbaum,

mithilfe eines zufällig erstellten Datensatz, der auf einem Eingabe Datensatz basiert, und einer zufälligen Teilmenge von Attributen erstellt, bis die gewünschte Anzahl an Entscheidungsbäumen erreicht ist. Um eine Vorhersage mit einem Random Forest zu treffen, wird eine Vorhersage von jedem Baum getroffen. Im Falle der Klassifikation wird die Vorhersage nach einem Mehrheitsentscheid getroffen und im Falle der Regression wird das arithmetische Mittel aller Vorhersagen ermittelt.

2.3 Prolog

Prolog [CR93] ist eine Programmiersprache, die 1972 von dem französichen Informatiker Alain Colmerauer maßgeblich entwickelt wurde. Sie ermöglicht deklaratives Programmieren und gilt als die wichtigste logische Programmierspache. Das Einsatzfeld befindet sich, auch historisch, in den Expertensystemen [Mer12], aber auch beim Theorem Proving [ZUB20] und Model Checking [SH10], um ein paar Beispiele zu nennen.

Als logische Programmiersprache ist Prolog deklarativ und nicht imperativ, wie zum Beispiel die Pogrammiersprachen Java und C. In Prolog Programmen wird beschrieben was berechent wird und nicht wie. Die Grundlagen eines Prolog Programms sind Fakten und Regeln. Fakten sind Aussagen, die ohne Bedingung wahr sind und bestehen nur aus einem Kopf. Regeln bestehen aus einem Kopf und einem Rumpf. Im Rumpf können endlich viele Aussagen stehen, die logisch verknüpft sind. Die logischen Verknüpfungen in Prolog sind in Tabelle 1 abgebildet. Relevant für meine Implementierung sind Prädikate. Ein Prädikat p

Tabelle 1: logische Verknüpfungen in Prolog

Verknüpfung	Prolog
$\Leftarrow$	:-
$\wedge$	,
$\vee$	;
$\neg$	\+

hat die Arität n, wobei n die Anzahl der Datenwerte ist, die das Prädikat bekommt. Ein Prädikat, das als Fakt geschrieben ist, also nur als Kopf, ist immer wahr. Wenn es als Regel geschrieben ist, ist das Prädikat nur wahr, wenn der Rumpf wahr ist. Zudem dürfen logische Verknüpfungen, wie $\wedge$, $\vee$ und $\neg$ nur im Rumpf verwendet werden. Das sorgt dafür, dass, wenn man die Implikation zwischen Kopf und Rumpf auflöst, der Kopf das einzige positive Literal darstellt und die restlichen Terme, negative Literale. Die Datenwerte, die in ein Prädikat hinein gegeben werden können, können im Programmcode aus Variablen bestehen. Variablen fangen mit Großbuchstaben oder einem Unterstrich an. Wichtig zu beachten ist, dass der Datenwert einer Variable nicht verändert werden kann, sollte es jedoch zu Backtracking kommen, kann es sein, dass die Variable in einem anderen Teilbaum, mit einem anderen Datenwert unifiziert wird. Rekursion von Prädikaten ist erlaubt und wird

Quellcode 1: Prolog implementation of `is_list/1`

```
1: is_list(X) :-
2:         var(X), !,
3:         fail.
4: is_list([]).
5: is_list([_|T]) :-
6:         is_list(T).
```

bei der Ausführung von Programmcode unter gewissen Umständen von Prolog performance technisch optimiert. Für meine Implementierung werden außerdem Komplexe Werte, oder auch compound Terms genannt, gebraucht. Diese bestehen aus einem Funktor f mit einer Stelligkeit von n. Im Gegensatz zu Prädikaten stehen sie für neue Datenwerte und stellen einen Term dar. Ob f(a,b) ein Prädikat oder ein Term ist, hängt letzendlich von der Position in der Klausel ab.

Prolog besitzt die Eigenschaft der Homoikonizität. Somit sind Programme gleichzeitig auch Datenstrukturen. Jedoch bietet Prolog, durch eine spezielle Schreibweise, gewisse Vorteile für die Benutzung einer Liste als Datenstruktur. Eine Liste wird durch [] umschlossen und kann beliebig viele Elemente beinhalten, welche alle von unterschiedlichen Typen sein können. Bei der Verarbeitung von Listen, wird meist die eingebaute Head-Tail-Aufteilung benutzt. Dabei wird das erste Element der Liste mit der Head-Variable unifiziert, während die restliche Liste mit der Tail-Variable unifiziert wird. Quellcode 1 zeigt das in SWI-Prolog [WSTL12] eingebaute Prädikat `is_list/1` und demonstriert ein einfaches Prolog Programm, dass überprüfen soll, ob die Eingabe eine Liste ist oder nicht. Das Prädikat `is_list/1` wird hier zweimal als Regel und einmal als Fakt verwendet. Bei einem Programmaufruf wird zuerst die oberste Regel aufgerufen. Diese unifiziert die Eingabe mit der Variable X. Das sorgt dafür, dass diese Eingabe auch von `var/1` benutzt wird. Diese Prädikat wird wahr, wenn die Eingabe eine Variable ist und schlägt sonst fehl. Sollte das Programm an der Stelle fehlschlagen, wird als nächstes der Fakt überprüft, da Regeln Hornklauseln darstellen und bei etwas im Rumpf falsch ist, die Regel falsch ist. Falls die Eingabe tatsächlich eine Variable ist, ist `var/1` wahr und der sogenannte Cut (!) wird eingelesen. Dieser verhindert, dass Backtracking betrieben wird. In dem Fall würde durch das fail die Regel falsch sein und durch den Cut wird der Fakt und die andere Regel nicht mehr überprüft und das Programm gibt falsch aus. Falls die leere Liste die Eingabe ist, gibt das Programm in dem Moment wahr aus, wenn es den Fakt liest. Wenn das Programm mit einer Liste, die nicht leer ist, aufgerufen wird, wird auch die letzte Regel erreicht. Diese kann nur aufgerufen werden, wenn sich die Eingabe in die Head-Tail-Aufteilung zerlegen lässt. Hier wird der Kopf mit einer sogenannten anonymen Variable unifiziert und der Tail mit der Variable T. Als nächstes wird dann überprüft ob T eine Liste ist. Diese Überprüfung wird somit rekrusiv so lange ausgeführt, das Prädikat fehlschlägt oder der Fakt erreicht wird.

3 Implementierung

In diesem Kapitel wird zunächst der selbst angefertigte Quellcode für den CART Algorithmus ausführlich erklärt. Darufhin folgt der Code für den Random Forest Algorithmus. Zum Schluss wird noch auf den Code eingegangen, der benötigt wird um Vorhersagen mit den erzeugten Strukturen zu Treffen.

3.1 Entscheidungsbäume

Vorab muss erwähnt werden, dass die Art wie ich meinen Baum und den Datensatz repräsentiere, auf der Code-Skizze zu Entscheidungsbäumen aus dem Buch „Prolog programming for artificial intelligence“ [Bra01], basiert. Wie das aussieht ist in Tabelle 2 dargestellt.

Tabelle 2: Darstellung von Baum und Datensatz

Strukur	Darstellung
Baum	$tree(Attribut = Wert, LinkerTeilbaum, RechterTeilbaum)$
Datensatz	$[example(Klasse, [Att1 = Wert1, ...]), example(...), ...]$

Analog zu der Erklärung zum Algorithmus, wird die Implementierung von oben nach unten erklärt. Im ersten Prädikat, Quellcode 2, befindet sich der rekursive Aufruf der den Baum konstruiert. Der erste Fall gibt einen leeren Baum, beziehungsweise Liste, zurück, falls keine Daten übergeben wurden. Der zweite Fall überprüft, ob alle momentan genutzten Datenpunkte dieselbe Klasse haben, wie der erste Datenpunkt in der Liste. Dafür wird das Hilfsprädikat `check-classes/2` verwendet. Falls dies der Fall sein sollte wird ein Blatt erzeugt. Der dritte Fall erzeugt zwei Kinder im Entscheidungsbaum. Dafür wird zuerst, mithilfe von Quellcode 3, aus den vorhandenen Datenpunkten, ein Wert zu einem Attribut gefunden, der am besten zwischen den Klassen differenziert. Daraufhin wird mit Hilfe dieses Attribut-Wert-Paares und `split/4` der Datensatz entsprechend aufgeteilt. Der Wert von dem gefundenen Attribut-Wert-Paar ist im numerischen Fall eine Zahl und im kategorischen Fall eine Teilmenge der Werte, die das entsprechende Attribut annehmen kann. Dabei landen die Datenpunkte, bei numerischen Werten, die kleiner sind als der gegebene Wert, im linken ansonsten im rechten Teilbaum. Bei kategorischen Werten landen die Datenpunkte im linken Teilbaum, wenn deren Werte, teil der Teilmenge sind, ansonten im rechten. Die daraus entstehenden Teilmengen werden dann genutzt um Teilbäume zu konstruieren. Im Gegensatz zu den üblichen Implementierungen habe ich mich dazu entschieden den Baum bis zur höchsten Reinheit aufzubauen. Zum einen besteht laut Demidova und Usachev [DU20] die Gefahr, dass bei anderen Abbruchbedingungen die Klassifikations-Qualität verloren geht und zum anderen muss kein extra Quellcode für die Verwendung von Random Forest angefertigt werden. Jedoch gibt es eine weitere Abbruchbedingung die eingebaut werden musste. Falls der aktuelle Datensatz Datenpunkte enthält, die von verschiedenen Klassen

Quellcode 2: Prolog implementation of `induce_tree/3`

```
1:  %! induce_tree(+Daten:list, +Attribute:list, -Tree:term) is det
2:  %
3:  %  induziert den Entscheidungsbaum durch 3 verschiedene Faelle:
4:  %  1) Tree = null, wenn keine Daten existieren
5:  %  2) Tree = leaf(Class), wenn alle Beispiele zur selben Klasse gehoeren
6:  %             oder nicht mehr unterschieden werden koennen
7:  %  3) Tree = tree(Attribut = Wert, SubTree1, SubTree2), wenn es mehr als
8:  %             eine vorhandene Klasse gibt.
9:  induce_tree([], _, []) :- !.
10: induce_tree([example(Klasse, _)|T], _, leaf(Klasse)) :-
11:     check_classes(T, Klasse), !.
12: induce_tree(Daten, Attribute, tree(BestAttVal, SubTree1, SubTree2)) :-
13:     find_best_split(Daten, Attribute, BestAttVal),
14:     split(Daten, BestAttVal, Split1, Split2),
15:     induce_tree(Split1, Attribute, SubTree1),
16:     induce_tree(Split2, Attribute, SubTree2).
17: induce_tree(Daten, Attribute, leaf(Klasse)) :-
18:     find_best_split(Daten, Attribute, err = Klasse), !.
```

sind, sich aber nicht unterscheiden lassen, wird ein Blatt erzeugt. Bei Klassifikationsbäumen hat das Blatt die Klasse, die am meisten im Datensatz vertreten ist, bei Regressionsbäumen ist das Blatt das arithmetische Mittel der Zielvariablen im Datensatz.

Als nächstes geht es darum das beste Attribut-Wert-Paar zum aufteilen des Datensatzes zu finden. Dieser Schritt lässt sich in zwei Teilschritte unterteilen. Als erstes muss eine Liste erstellt werden, die die Attribut-Wert Paare beinhaltet an denen der Datensatz potentiell aufgeteilt werden kann. Als zweites muss das Paar ausgewählt werden, welches die zwei „reinsten“ Datensätze hervorbringt. Die Koordinierung dieser beiden Aufgaben wird von Quellcode 3 übernommen. In der Implementierung wird an dieser Stelle die eben erwähnte Abbruchbedingung abgefangen. Sollte `split_candidates/2` eine leere Kandidatenliste ausgeben, ist klar, dass der aktuelle Datensatz nicht weiter aufgeteilt werden kann. In dem Fall wird wie beschrieben vorgegangen.

Das erstellen der Liste von potentiellen Aufteilungen wird von `split_candidates/3` erledigt. Dieses Prädikat benutzt einen Akkumulator. Das heißt, dass es ein Prädikat mit seinen Variablen und einer zusätzlichen leeren Liste aufruft. Hier wird mithilfe von Quellcode 4 eine Liste von Kandidaten für ein einzelnes Attribut erstellt. Diese Liste wird an den Akkumulator angehangen. Das wird so lange wiederholt, bis alle Attribute abgearbeitet sind. Dann wird der Akkumulator mit der Ausgabevariablen unifiziert.

Um eine Liste der Kandidaten zu erstellen, geht Quellcode 4 wie folgt vor: Zunächst wird durch `check_attribute_number/2` überprüft, ob das Attribut numerisch oder kategorisch ist. Bei numerischen Werten wird `get_values_n/3` aufgerufen. Hier wird wieder ein Akkumulator benutzt in dem die Werte zu dem aktuellen Attribut eingefügt werden.

Quellcode 3: Prolog implementation of `find_best_split/3`

```
1:  %! find_best_split(+Daten:list, +Attribute:list, -BestAttVal:Att = Val)
2:  %  is det
3:  %
4:  % Findet den besten Split zur aktuellen Datenlage
5:  find_best_split(Daten, Attribute, BestAttVal) :-
6:      split_candidates(Daten, Attribute, Kandidaten),
7:      ([] \= Kandidaten ->
8:          best_split(Kandidaten, Daten, BestAttVal)
9:      ;   get_classes(Daten, Klassen),
10:         sort(Klassen, Klassenliste),
11:         get_max(Klassenliste, Klassen, Max),
12:         best_split(Max, err, BestAttVal)).
```

Quellcode 4: Prolog implementation of `split_candidates_attribute/3`

```
1:  %! split_candidates_attribute(+Daten:list, +Attribut:atom, -Kandidaten:list)
2:  %  is det
3:  %
4:  %  Erstellt ein Liste mit allen moeglichen Splits zu einem Attribut
5:  split_candidates_attribute(Daten, Attribut, Kandidaten) :-
6:      (check_attribute_number(Daten, Attribut) ->
7:      get_values_n(Daten, Attribut, Values)
8:      ; get_values_k(Daten, Attribut, Values)),
9:      create_candidates(Values, Attribut, Kandidaten).
```

Dabei wird darauf geachtet, dass kein Wert, zu einem Attribut, doppelt vorkommt. Bei kategorischen Werten wird `get_values_k/3` aufgerufen. Hier wird analog zum vorherigen Prädikat vorgegangen. Nachdem eine Liste mit unterschiedlichen Werten vorliegt, berechnet `create_candidates/3` daraus eine Liste mit möglichen Aufteilungen. Im numerischen Fall wird dabei die Mitte von zwei Werten aus der übergebenen Liste berechnet und als Attribut-Wert Paar in die Ergebnisliste aufgenommen. Im kategorischen Fall werden mit Hilfe von `my_powerset/2` mindestens $2^{m-1} - 1$ verschiedene Kombinationen von Werten aus der Kandidaten Liste erzeugt, wobei m die Anzahl an Werten ist, die das Attribut annehmen kann. Dabei ist jede Kombination in einer Liste verpackt und wird dann als Attribut-Liste-Paar in die Ergebnisliste aufgenommen. Dadurch beinhaltet die Liste alle möglichen Kandidaten zur Aufteilungen des aktuellen Datensatzes. Alternativ hätte man bei numerischen Werten das Minimum und Maximum nehmen können und in diesem Intervall eine feste Anzahl an Aufteilungen, in gleichmäßigen Abständen durchführen können [BFSO84]. Mola [MS97] hat auch schon einen schnelleren Aufteilungsalgorithmus vorgestellt, der auf dem Vorhersehbarkeits Index basiert.

Mit der Liste aller möglichen Aufteilungen wird durch Quellcode 5 die Aufteilung vom Datensatz gefunden, die das Maß der Unreinheit minimiert. Im ersten Fall wird die Ab-

Quellcode 5: Prolog implementation of `best_split/3`

```
1:  %! best_split(+Kandidaten:list, +Daten:list, -BestAttVal: Att = Val)
2:  %  is det
3:  %! Kandidaten = [Att = Val1, Att = Val2, ...]
4:  %
5:  %  Verwendet alle Kandidaten aus der Liste um den besten Split
6:  %  zu finden und auszugeben
7:  best_split(Klasse, err, err = Klasse).
8:  best_split([H|T], Daten, Res) :-
9:      impurity(Daten, H, Min),
10:     best_split(T, Daten, Min, H, Res).
11: best_split([], _, _, Res, Res) :- !.
12: best_split([H|T], Daten, Min, Cur, Res) :-
13:     impurity(Daten, H, CombUnreinheit),
14:     (CombUnreinheit < Min ->
15:         best_split(T, Daten, CombUnreinheit, H, Res)
16:     ;   best_split(T, Daten, Min, Cur, Res)).
```

Quellcode 6: Prolog implementation of `impurity/3`

```
1:  %! impurity(+Daten:list, +AttVal:Attribute = Value, -Unreinheit:double) is
    det
2:  %
3:  %  Gibt die kombinierte Unreinheit, von dem Split des Datensatzes, aus
4:  impurity(Daten, AttVal, CombUnreinheit) :-
5:      split_class(Daten, AttVal, Split1, Split2),
6:      combgini(Split1, Split2, CombUnreinheit).
```

bruchbedingung abgefangen und die Klasse für das Blatt nach oben gegeben. Ansonsten werden die Unreinheit der Aufteilung, die durch den ersten Kandidaten entsteht, und der Kandidat selbst, je als Variable gemerkt. Danach wird in jedem Schritt die Unreinheit eines neuen Kandidaten berechnet und mit der entsprechenden Variable verglichen. Wenn der Wert kleiner ist, wird mit diesem in den nächsten Schritten verglichen und der Kandidat wird auch gemerkt. Falls der neue Wert der Unreinheit größer ist, werden die Variablen nicht überschrieben und es geht mit dem nächsten Kandidaten weiter.

Entscheidend ist auch, welches Maß der Unreinheit verwendet wird. Dieses wird hinter dem Prädikat Quellcode 6 versteckt. In dieser Implementierung wird die „Gini-Unreinheit“ verwendet. Doch bevor gerechnet werden kann, bereitet `split-class/3` die Daten entsprechend auf. Dieses Prädikat erstellt aus dem übergebenen Datensatz, mithilfe eines Attribut-Wert Paars, zwei kleinere Datensätze. Die kleineren Datensätze bestehen dabei nur aus den Klassen der Datenpunkte, die in dem Datensatz zugeordnet wurden. Dafür werden wieder 2 Akkumulatoren verwendet. Im ersten Fall wird zunächst nach dem Attribut vom Attribut-Wert-Paar im Datensatz gesucht. Sobald dieses gefunden ist, gibt es einen Fall

Quellcode 7: Prolog implementation of `combgini/3`

```
1:  %! combgini(+Split1:list, +Split2:list, -CombUnreinheit:double) is det
2:  %
3:  %  Berechnet die gewichtete Summe von zwei Splits.
4:  %  Beachte Split ist eine Liste von Klassen!
5:  combgini(Split1, Split2, CombUnreinheit) :-
6:      gini(Split1, Unreinheit1),
7:      gini(Split2, Unreinheit2),
8:      length(Split1, LSplit1),
9:      length(Split2, LSplit2),
10:     LDaten is LSplit1 + LSplit2,
11:     Gewicht1 is LSplit1 / LDaten,
12:     Gewicht2 is LSplit2 / LDaten,
13:     GUnreinheit1 is Gewicht1 * Unreinheit1,
14:     GUnreinheit2 is Gewicht2 * Unreinheit2,
15:     CombUnreinheit is GUnreinheit1 + GUnreinheit2.
```

dafür, dass der Wert zu diesem Attribut kategorisch oder numerisch ist. Im numerischen Fall wird überprüft, ob der Wert des Datenpunktes kleiner ist, als der des übergebenen Werts. Falls dies der Fall ist, wird die Klasse des Datenpunktes an den linken Akkumulator angehangen, ansonsten am rechten. Analog dazu wird mit kategorischen Werten umgegangen, nur das hier überprüft wird, ob der Wert des Datenpunkts in der Liste der übergebenen Werte vorhanden ist.

Nach der Erstellung, der aus Klassen bestehenden Datensätze, berechnet Quellcode 7 die kombinierte Gini-Unreinheit der beiden Datensätze. Als erstes wird die Gini-Unreinheit der einzelnen Datensätze mit `gini/2` berechnet. Die Ergebnisse daraus werden dann mit dem Anteil an Datenpunkten, den der jeweilige Datensatz aus dem ursprünglichen Datensatz hatte, multipliziert. Diese Ergebnisse werden dann addiert und bilden die kombinierte Unreinheit. Dieses Verfahren entspricht Gleichung (3).

Der Algorithmus für die Regression funktioniert zu weiten Teilen wie der Klassifikationalgorithmus, nur dass hier, als Maß der Unreinheit, die Summe der quadrierten Residuen minimiert wird. Quellcode 8 ruft `mean/2` auf um das arithmetische Mittel, der Werte in den Datensätzen zu berechnen. Damit wird dann in `residual-split/3` die Summe der quadrierten Differenzen, der einzelnen Werte, zum arithmetischen Mitteln, ihres jeweiligen Datensatzes, berechnet. Dieses Verfahren entspricht Gleichung (2) Zum Schluss werden die beiden Summen addiert und ergeben das Maß der Unreinheit.

Um keinen Code programmieren zu müssen, der eine Wahl ermöglicht, ob ein Klassifikationsbaum oder Regressionsbaum erstellt werden soll, sind beide Verfahren in eigenen Dateien gespeichert, sodass man nur das entsprechende Modul benutzten muss, den dazugehörigen Baum zu erzeugen.

Quellcode 8: Prolog implementation of `residual_sum_sq/3`

```
1:  %! residual_sum_sq(+Split1:list, +Split2:list, -ResSumSq:double) is det
2:  %
3:  %  Berechnet die Summe der quadrierten Residuen von zwei Splits.
4:  %  Beachte Split ist eine Liste von Target Werten!
5:  residual_sum_sq(Split1, Split2, ResSumSq) :-
6:      mean(Split1, Mean1),
7:      mean(Split2, Mean2),
8:      residual_split(Split1, Mean1, SumSq1),
9:      residual_split(Split2, Mean2, SumSq2),
10:     ResSumSq is SumSq1 + SumSq2.
```

Quellcode 9: Prolog implementation of `induce_forest/3`

```
1:  %! induce_forest(+Daten:list, Attribute:list, N:int,-RandomForest:list) is det
2:  %
3:  %  erstellt aus den Daten einen Entscheidungswald;
4:  %  Die Anzahl der Baueme wird angegeben;
5:  %  die Anzahl pro Schritt benutzten Attribute orientiert sich
6:  %  an einer in der praxis bewaehrten Faustregel
7:  induce_forest(Daten, Attribute, N, RandomForest) :-
8:      bootstrappen(Daten, N, Samples),
9:      choose_attributes(Attribute, N, AttributSets),
10:     induce_trees(Samples, AttributSets, RandomForest).
```

3.2 Random Forest

Mein Random-Forest-Algorithmus besteht aus drei Schritten. Als erstes werden zufällig Datenpunkte aus dem Datensatz, mit zurücklegen gezogen. Dadurch werden Datenpunkte entsprechend der Anzahl im Datensatz gezogen. Als zweites wird eine Liste von zufälligen Teilmengen der Attribute erzeugt. Als drittes wird aus jeweils einem gezogenen Datensatz und einer Teilmenge von Attributen ein Entscheidungsbaum erzeugt. Sowohl die Anzahl der gezogenen Datensets und der Teilmengen von Attributen werden im voraus bestimmt und resultieren in einer Liste mit dieser Anzahl an Entscheidungsbäumen. Dieser Vorgang wird durch Quellcode 9 koordiniert.

Als nächstes werden die benötigten Prädikate im Detail erklärt.

Das zufällige erstellen von Datensätzen wird durch Quellcode 10 implementiert. Zunächst wird die Länge des Datensatzes bestimmt. Danach werden durch `get_sample/3`, entsprechend dieser Anzahl zufällig Datenpunkte, mit zurücklegen, gezogen. Das Ergebnis wird als Liste in eine Liste eingefügt. Dieser Vorgang wird so oft wiederholt, wie Entscheidungsbäume am Ende erzeugt werden sollen.

Quellcode 10: Prolog implementation of `bootstrappen/3`

```
1:  %! bootstrappen(+Daten:list, +Trees:int, -Samples:list) is det
2:  %
3:  %  erstellt eine Liste von Listen, wobei jede der N Listen
4:  %  ein bootstrap sample von dem Datensatz ist
5:  bootstrappen(Daten, N, Samples) :-
6:      length(Daten, Len),
7:      bootstrappen(Daten, Len, N, Samples).
8:  bootstrappen(_, _, 0, []) :- !.
9:  bootstrappen(Daten, Len, N, [Sample|Samples]) :-
10:     get_sample(Daten, Len, Sample),
11:     NewN is N - 1,
12:     bootstrappen(Daten, Len, NewN, Samples).
```

Quellcode 11: Prolog implementation of `choose_attributes/3`

```
1:  %! choose_attributes(+Attribute:list, +N:int, -AttributSets) is det
2:  %
3:  %  erstellt N Listen, welche aus zufaellig ausgewahelten Attributen bestehen.
4:  %  Die Anzahl der Ausgewaehlten Attributen ist,
5:  %  bei Klassifikation, die abgerundete Wurzel, der Anzahl, der Attribute
6:  choose_attributes(Attribute, N, AttributSets) :-
7:      length(Attribute, Int),
8:      TRes is sqrt(Int),
9:      Res is floor(TRes),
10:     choose_all_members(Attribute, Res, N, AttributSets).
```

Die Erstellung der Teilmengen der Attribute wird durch Quellcode 11 initiiert. Hier wird die Größe der Teilmengen bestimmt. Die Anzahl der benutzten Attribute wird, wie in Gleichungen (4) und (5) vorgestellt berechnet.

Das eigentliche Erstellen der Teilmengen ist in Quellcode 12 implementiert. Hier wird in jeden Schritt eine Permutation der Attribut-Liste erstellt. Anschließend werden so viele Attribute wie benötigt, vom Anfang der Liste beginnend herausgenommen und als Teilmenge in die Ergebnisliste aufgenommen. Dieser Vorgang wird wiederholt, bis die Anzahl der angefertigten Teilmengen, der vorgegebenen Anzahl entspricht.

Zum Schluss wird von `induce_trees/3` eine Liste von Entscheidungsbäumen erstellt. Dafür wird jeweils ein zufällig erstellter Datensatz und eine Teilmenge der Attribute genommen und Quellcode 2 aufgerufen.

Quellcode 12: Prolog implementation of `choose_members/4`

```
1:  %! choose_all_members(+Attribute:list, +Len:int, +N:int, -AttributSets:list) is det
2:  %
3:  %  erstell N, Len Lange Listen, welche, zufaellig ausgewaehlte, unterschiedliche
4:  %  Elemente aus Attribute beinhalten
5:  choose_all_members(_, _, 0, []) :- !.
6:  choose_all_members(Attribute, Len, N, [AttributSet|AttributSets]) :-
7:      random_permutation(Attribute, Permutation),
8:      take(Permutation, Len, AttributSet),
9:      NewN is N - 1,
10:     choose_all_members(Attribute, Len, NewN, AttributSets).
```

Quellcode 13: Prolog implementation of `check_sample/3`

```
1:  %! check_sample(+Tree:Entscheidungsbaum, +Example:list, -Res:Class) is det
2:  %
3:  %  Gibt die vom Entscheidungsbaum vorhergesehene Klasse zu einem Datenpunkt
4:  %  ([att1=val1,...]) aus
5:  check_sample(leaf(Class), _, Class).
6:  check_sample(tree(Att = Val1, SubTree1, SubTree2), Example, Result) :-
7:      get_value(Example, Att, Val2),
8:      (is_list(Val1) ->
9:          (member(Val2,Val1) ->
10:             check_sample(SubTree1,Example,Result)
11:         ;   check_sample(SubTree2,Example,Result))
12:     ;   (Val2 < Val1 ->
13:             check_sample(SubTree1,Example,Result)
14:         ;   check_sample(SubTree2,Example,Result))).
```

3.3 Vorhersagen

In diesem Unterkapitel wird der Quellcode erklärt, der benutzt wird um eine Vorhersage mit einem Entscheidungsbaum oder einem Random Forest zu treffen. Bei der Vorhersage die von Entscheidungsbäumen getroffen werden, überprüft Quellcode 13, ob der Wert mit dem getrennt wird numerisch oder kategorisch ist. Im numerischen Fall, wird überprüft, ob der Wert vom Datenpunkt, zum entsprechendem Attribut kleiner ist, als der im Knoten. Falls ja, geht es in dem linken Teilbaum weiter, ansonsten im rechten. Der kategorische Fall funktioniert analog, nur das überprüft wird ob, der Wert des Datenpunktes, ein Element in der Liste der Werte ist. Sobald man in einem Blatt angekommen ist, ist der Inhalt des Blatts die Vorhersage.

Um mehrere Datenpunkte auf einmal einzugeben wurde `check_all_sample/3` implementiert. Dieses Prädikat ruft für jeden Datenpunkt Quellcode 13 auf und verpackt alle Ergebnisse in einer Liste.

Ein Random Forest, trifft seine Vorhersage, durch einen Mehrheitsentscheid oder das arithmetische Mittel. In der Implementierung wird für einen Datenpunkt eine Vorhersage von jedem Entscheidungsbaum im Random Forest getroffen. Im Fall der Klassifikation wird die am meisten vorhergesagte Klasse ausgegeben, im Fall der Regression, wird das arithmetische Mittel aller Vorhersagen ausgegeben.

4 Experiment

In diesem Kapitel werden zunächst die nötigen Vorbereitungen für das Experiment in Abschnitt 4.1 vorgestellt. In dem Abschnitt 4.2 wird die Präzision und die Ausführungszeit, der in dieser Arbeit vorgestellten Implementierung in SWI-Prolog [WSTL12] und SICSTus-Prolog [CWA+88], mit Implementierungen aus den R Bibliotheken „rpart“ [TARR15] und „randomForest“ [RL18] und der Python Bibliothek „scikit-learn“ [PVG+11].

4.1 Vorbereitung

Zunächst wird die Art der Datenbeschaffung vorgestellt. Daraufhin werden die benutzten Metriken erklärt. Zum Schluss werden die Implementierungen aus den Referenzimplementationen exemplarisch vorgestellt.

4.1.1 Datenbeschaffung

Um die Ergebnisse der einzelnen Implementierungen miteinander vergleichen zu können wurden jeweils fünf Train-Test-Aufteilungen zu den einzelnen Datensätzen erstellt und in CSV-Dateien abgespeichert. Die Daten werden über scikit-learn heruntergeladen und anschließend mithilfe der Bibliothek „pandas“ [M+11] als CSV Datei abgespeichert. Für Train-Test-Aufteilungen bietet scikit-learn die Funktion `train_test_split` an. Diese zerlegt einen übergebenen Datensatz, in einem selbst bestimmbaren Verhältnis, in Trainings und Test Daten. Ich habe mich für 70% der Daten im Trainings- und 30% der Daten im Test-Datensatz entschieden. Um die Zerlegung zu replizieren, kann sogar ein Seed für den Zufallsgenerator übergeben werden. Quellcode 14 zeigt beispielhaft die Erstellung einer Train-Test-Aufteilung. Für alle fünf Aufteilungen eines Datensatzes wurden jeweils die Seeds 18, 19, 20, 21 und 22 verwendet.

4.1.2 Zeitmessung

Der Kern des Experiments soll es sein, die gemessene Trainingszeit der Implementierungen miteinander zu vergleichen. Dafür wird in diesem Unterkapitel die Zeitmessung vorgestellt.

Quellcode 14: Train-Test-Split and convert to csv

```
iris_feature_train1, iris_feature_test1, iris_target_train1, iris_target_test1 =
        train_test_split(iris['data'],iris['target'],test_size=0.3,random_state=18)

df = pd.DataFrame(data=iris_feature_train1, columns = iris['feature_names'])
df['class'] = iris_target_train1
df.to_csv('iris_train1.csv', sep = ',', index = False)

df = pd.DataFrame(data=iris_feature_test1, columns = iris['feature_names'])
df['class'] = iris_target_test1
df.to_csv('iris_test1.csv', sep = ',', index = False)
```

Quellcode 15: Prolog implementation of `create_tree/2`

```
create_tree(File, Tree, Time) :-
    csv_read_file(File, [H|T]),
    get_attribute_list(H, Attribute),
    fit_format(T, Attribute, Data),
    statistics(walltime, _),
    induce_tree(Data, Attribute, Tree),
    statistics(walltime, [_,Time]).
```

Die Zeitmessung ist in allen Implementierungen analog. Es wird eine vorgefertigte Methode für Zeitstempel vor und nach dem Aufruf der Methode, die den Algorithmus durchführt, benutzt. Daraufhin werden die Zeitstempel subtrahiert um die vergange Zeit zu erhalten oder die Methode für den Zeitstempel hat bereits eine Ausgabe für den Abstand zum letzten Aufruf. Letzteres ist der Fall in der Prolog Implementierung. Das eingebaute Prädikat `statistics/2` liefert in seiner zweiten Variable, als zweiten Wert die Zeit zum letzten Aufruf in Millisekunden. Das ist exemplarisch in Quellcode 15 implementiert.

In der R Implementierung wird mit der Bibliothek „tictoc“ [Izr14] gearbeitet. Hier wird der Algorithmus Aufruf von zwei Aufrufen eingeschlossen, wobei der zweite Aufruf die vergange Zeit auf der Konsole in Sekunden ausgibt und in ein Log schreibt. Das ist exemplarisch in Quellcode 16 dargestellt. Allerdings kann diese Implementierung nur bis auf 10 Millisekunden genau messen.

In der Python Implementierung wird mit der eingebauten „time“ Bibliothek gearbeitet. Hier wird vor und nach der Ausführung die Summe aus System und CPU-Zeit gemessen und anschließend voneinander subtrahiert. Das ist exemplarisch in Quellcode 17 implementiert. Hier ist die Präzision der Zeitmessung um die 16 Millisekunden begrenzt.

Quellcode 16: R implementation of `train.test.classification`

```
train.test.classification <- function(trainData, testData) {
  train <- read.csv(trainData)
  train$class <- as.factor(train$class)
  tic(trainData)
  fit <- rpart(class ~ ., train)
  toc(log = TRUE)
  test <- read.csv(testData)
  prediction <- predict(fit, newdata = test, type = "class")
  true <- prediction == test$class
  precision <- sum(true) / length(true)
  print(c("Precision:", precision))
  return(precision)
}
```

Quellcode 17: Python implementation of `train_test_classification`

```
def train_test_classification(feature_train,target_train,
                                    feature_test,target_test):
    t1 = time.process_time()
    clf = tree.DecisionTreeClassifier()
    clf = clf.fit(feature_train,target_train)
    t2 = time.process_time()
    trainTime = t2 - t1
    print('Train Time:       {time} sec.'.format(time=trainTime))
    preds = clf.predict(feature_test)
    prec = precision(preds, target_test)
    print('Precision:        {p}'.format(p = prec))
    return trainTime, prec
```

Quellcode 18: Prolog implementation of `precision/3`

```
1:  %! precision(+Klassen:list, +Pred:list, -Gen:double) is det
2:  %
3:  %  Berechnet den Anteil der Elemente,
4:  %  die in beiden Listen miteinander uebereinstimmen
5:  precision(Klassen, Pred, Gen) :-
6:      precision(Klassen, Pred, 0, 0, Gen).
7:  precision([], [], Len, Acc, Gen) :- !, Gen is Acc / Len.
8:  precision([KH|KT], [PH|PT], Len, Acc, Gen) :-
9:      NewLen is Len + 1,
10:     (KH == PH ->
11:         NewAcc is Acc + 1,
12:         precision(KT, PT, NewLen, NewAcc, Gen)
13:     ;   precision(KT, PT, NewLen, Acc, Gen)).
```

4.1.3 Metriken

Nun geht es darum die benutzten Metriken in der Implementierung vorzustellen. Als Performance Metrik für die Klassifikation, habe ich mich für die einfachste entschieden, die Genauigkeit, wie in Gleichung (6) dargestellt. Hier steht p_i für die i-te Vorhersage und nimmt den Wert 1 an, wenn richtig vorhergesagt wurde oder 0, wenn nicht. Im Fall der Regression habe ich mich für den „Mean-Squared-Error“ Gleichung (7), als Metrik für die Vorhersage, entschieden. Zusätzlich wird für die Zeitmessung der „Standard Error of Mean“ Gleichung (8) berechnt.

$$Precision = 1/n * \sum_{i=1}^{n} p_i \tag{6}$$

$$MeanSquaredError = 1/n * \sum_{i=1}^{n} (y_i - f(x_i))^2 \tag{7}$$

$$StandardErrorofMean = o/\sqrt{n} \tag{8}$$

$$o = \sqrt{(\sum_{i=1}^{n} (t_i - \mu)^2)/n - 1}\,, \tag{9}$$

Hierbei ist y_i der i-te tatsächliche Wert und $f(x_i)$ der i-te vorhergesagte Wert, t_i die i-te Zeitmessung und μ das arithmetische Mittel. Die Implementierung der Präzision ist für SWI- und SICSTus-Prolog in Quellcode 18, für R in Quellcode 16 (als Variable „precision“) und für Python in Quellcode 19 dargestellt.

Die Implementierung des Mean Squared Error ist für SWI- und SICSTus-Prolog in Quellcode 20, für R in Quellcode 21 (als Variable „mse“) und für Python in Quellcode 22 dargestellt.

Quellcode 19: Python implementation of `precision`

```
def precision(preds, test):
    l = len(preds)
    akk = 0
    for i in range(l):
        if(preds[i] == test[i]):
            akk += 1
    return akk / l
```

Quellcode 20: Prolog implementation of `mean_sq_error/3`

```
%! mean_sq_error(+Klassen:list, +Pred:list, -Gen:double) is det
%
%  berechnet den Mean-Squared-Error zwischen Klassen und Pred.
mean_sq_error(Klassen, Pred, Gen) :-
    mean_sq_error(Klassen, Pred, 0, 0, Gen).
mean_sq_error([], [], Len, Acc, Gen) :- !, Gen is Acc / Len.
mean_sq_error([HK|TK], [HP|TP], L, Acc, Gen) :-
    Diff is HK - HP,
    Sq is Diff**2,
    NewAcc is Acc + Sq,
    NewL is L + 1,
    mean_sq_error(TK, TP, NewL, NewAcc, Gen).
```

Quellcode 21: R implementation of `train.test.regression`

```
train.test.regression <- function(trainData, testData) {
  train <- read.csv(trainData)
  tic(trainData)
  fit <- rpart(Y ~ ., train)
  toc(log = TRUE)
  test <- read.csv(testData)
  prediction <- predict(fit, newdata = test, type = "vector")
  mse <- sum((prediction - test$Y)**2) / length(test$Y)
  print(c("Mean-Squared-Error:", mse))
  return(mse)
}
```

Quellcode 22: Python implementation of `mean_sq_err`

```
def mean_sq_err(preds, tests):
    l = len(preds)
    acc = 0
    for i in range(l):
        acc += (preds[i] - tests[i])**2
    return acc / l
```

Quellcode 23: Prolog implementation of `run_iris_tree/0`

```
1:  run_iris_tree :-
2:      train_test_data_tree('Datensets/iris_train1.csv',
3:                           'Datensets/iris_test1.csv', Time1, Gen1),
4:      format("----------------------------------------~n",[]),
5:      train_test_data_tree('Datensets/iris_train2.csv',
6:                           'Datensets/iris_test2.csv', Time2, Gen2),
7:      format("----------------------------------------~n",[]),
8:      train_test_data_tree('Datensets/iris_train3.csv',
9:                           'Datensets/iris_test3.csv', Time3, Gen3),
10:     format("----------------------------------------~n",[]),
11:     train_test_data_tree('Datensets/iris_train4.csv',
12:                          'Datensets/iris_test4.csv', Time4, Gen4),
13:     format("----------------------------------------~n",[]),
14:     train_test_data_tree('Datensets/iris_train5.csv',
15:                          'Datensets/iris_test5.csv', Time5, Gen5),
16:     mean([Time1, Time2, Time3, Time4, Time5],AvgT),
17:     mean([Gen1, Gen2, Gen3, Gen4, Gen5],AvgG),
18:     calc_metrics([Time1, Time2, Time3, Time4, Time5], AvgT, _, _, Err),
19:     format("iris:~n Average train time: ~w sec.~n
20:                     Standard Error of Mean(Time): ~w ~n
21:                     Average accuracy: ~w ~n", [AvgT,Err,AvgG]).
```

Um mögliche Ausreißer in der Performance auszugleichen werden zu jedem Datensatz fünf verschiedene Trainings-Test-Aufteilungen durchgeführt und ausgewertet, sodass für diese Ergebnisse das arithmetische Mittel berechnet werden kann. Zusätzlich wird für die Zeit der Standard Error of Mean Gleichung (8) berechnet. Dies ist exemplarisch für den Iris-Datensatz in Quellcodes 23 und 24, für Prolog, in Quellcode 25, für R und in Quellcodes 26 und 27 für Python, implementiert.

4.2 Performance

Alle der hier benutzten Datensätze wurden aus der Python Bibliothek scikit-learn übernommen. Die Erklärungen basieren auf dem jeweiligen „User Guide“ Eintrag auf der Website von scikit-learn.

4.2.1 Iris

Bei diesem Datensatz handelt es sich um eine Abwandlung des weit verbreiteten Iris Datensatzes [Fis36] von Fisher. Im Gegensatz zum Original sind in dem hier benutzten Datensatz keine fehlerhaften Datenpunkte. Der Datensatz besteht enthält 3 verschiedene Klassen (0 entspricht „Iris-Setosa“, 1 entspricht „Iris-Versicolor“, 2 entspricht „Iris-Verginica“),

Quellcode 24: Prolog implementation of `calc_metrics/5`

```
1:  %! calc_metrics(+Targets:list, +Mean:double, -Varianz:double,
2:  %                                          -Abw:double, -Err:double) is det
3:  %
4:  %  Berechnet die Sample Varianz, Sample Standardabweichung und
5:  %  Standard error of mean
6:  calc_metrics([], _, 0, 0, 0) :- !.
7:  calc_metrics(Targets, Mean, Varianz, Abw, Err) :-
8:      calc_metrics(Targets, Mean, -1, 0, Varianz, Abw, Err).
9:  calc_metrics([], _, Len, SumSq, Varianz, Abw, Err) :-
10:     !,
11:     Varianz is SumSq / Len,
12:     Abw is sqrt(Varianz),
13:     RealLen is Len + 1,
14:     RootLen is sqrt(RealLen),
15:     Err is Abw / RootLen.
16: calc_metrics([H|T], Mean, Len, Acc, Varianz, Abw, Err) :-
17:     Diff is H - Mean,
18:     Sq is Diff**2,
19:     NewAcc is Acc + Sq,
20:     NewLen is Len + 1,
21:     calc_metrics(T, Mean, NewLen, NewAcc, Varianz, Abw, Err).
```

wovon jede 50 Instanzen besitzt und sich auf ein Typ von Schwertlilien beziehen. Die Insatzen werden sowohl durch die Kelchblatt Länge und Breite, als auch die Blütenblatt Länge und Breite in Centimeter beschrieben.

In Tabelle 3 erkennt man, dass meine Implementierung, für diesen kleinen Datensatz, die selbe Präzision hat und noch relativ nah an den Referenz Implementierungen liegt. Es fällt allerdings schon auf, dass die Ausführung in Sicstus Prolog, im Vergleich zu SWI, deutlich länger braucht.

Tabelle 3: Entscheidungsbaum Ergebnisse Iris. Die Spalte für die Zeit beinhaltet die durchschnittliche Laufzeit und den Standard Error of Mean in Millisekunden. Die Spalte Präzision beinhaltet die durchschnittliche Präzision der Vorhersagen in Prozent

Implementierung	Zeit	Präzision
SWI	26.0 $\pm$6.6 ms	92.88
Sicstus	74.2 $\pm$1.8 ms	92.88
R	$<$ 10 $\pm<$ 10 ms	92.00
Python	3.1 $\pm$3.1 ms	92.88

In Tabelle 4 fällt dieser kleiner Abstand stärker ins Gewicht, da jeweils 100 Bäume erstellt werden. Inzwischen braucht meine Implementierung fast das hundertfache an Zeit im

Quellcode 25: R implementation of `run.iris`

```
run.iris <- function() {
  print("Training iris: Set1")
  p1 <- train.test.classification('iris_train1.csv', 'iris_test1.csv')
  print('------------------------------------------------------------')
  print("Training iris: Set2")
  p2 <- train.test.classification('iris_train2.csv', 'iris_test2.csv')
  print('------------------------------------------------------------')
  print("Training iris: Set3")
  p3 <- train.test.classification('iris_train3.csv', 'iris_test3.csv')
  print('------------------------------------------------------------')
  print("Training iris: Set4")
  p4 <- train.test.classification('iris_train4.csv', 'iris_test4.csv')
  print('------------------------------------------------------------')
  print("Training iris: Set5")
  p5 <- train.test.classification('iris_train5.csv', 'iris_test5.csv')
  print('------------------------------------------------------------')
  log.lst <- tic.log(format = FALSE)
  tic.clearlog()
  timings <- unlist(lapply(log.lst, function(x) x$toc - x$tic))
  mp <- mean(c(p1,p2,p3,p4,p5))
  variance <- var(timings)
  abw <- sqrt(variance)
  err <- abw / sqrt(length(timings))
  print(c('Average Train Time:', mean(timings)))
  print(c('Standard Error of Mean(Time:)', err))
  print(c('Average Precision:', mp))
  print('-------------------------------------------')
  print('-------------------------------------------')
}
```

Quellcode 26: Python implementation of `run_iris`

```
def run_iris():
    print("Training iris: Set1")
    t1, p1 = train_test_classification(iris_feature_train1, iris_target_train1,
                                       iris_feature_test1, iris_target_test1)
    print('------------------------------------------------------------')
    print("Training iris: Set2")
    t2, p2 = train_test_classification(iris_feature_train2, iris_target_train2,
                                       iris_feature_test2, iris_target_test2)
    print('------------------------------------------------------------')
    print("Training iris: Set3")
    t3, p3 = train_test_classification(iris_feature_train3, iris_target_train3,
                                       iris_feature_test3, iris_target_test3)
    print('------------------------------------------------------------')
    print("Training iris: Set4")
    t4, p4 = train_test_classification(iris_feature_train4, iris_target_train4,
                                       iris_feature_test4, iris_target_test4)
    print('------------------------------------------------------------')
    print("Training iris: Set5")
    t5, p5 = train_test_classification(iris_feature_train5, iris_target_train5,
                                       iris_feature_test5, iris_target_test5)
    print('------------------------------------------------------------')
    mt = mean([t1,t2,t3,t4,t5])
    mp = mean([p1,p2,p3,p4,p5])
    var, abw, err = metrics([t1,t2,t3,t4,t5])
    print('Average Train Time:            {t} sec.'.format(t = mt))
    print('Standard Error of Mean(Time):  {err}'.format(err = err))
    print('Average Precision:             {p}'.format(p = mp))
    print()
    print()
```

Quellcode 27: Python implementation of `metrics`

```
def metrics(targets):
    l = len(targets)
    m = mean(targets)
    sumsq = 0
    for i in range(l):
        sumsq += (targets[i] - m)**2
    variance = sumsq / (l - 1)
    abw = variance**(1/2)
    err = abw / (l**(1/2))
    return variance, abw, err
```

Vergleich zu R, ohne dabei eine höhere Präzision zu erreichen. Die Ausführung in Sicstus dauert sogar drei mal so Lange wie die in SWI.

Tabelle 4: Random Forest Ergebnisse Iris. Die Spalte für die Zeit beinhaltet die durchschnittliche Laufzeit und den Standard Error of Mean in Millisekunden. Die Spalte Präzision beinhaltet die durchschnittliche Präzision der Vorhersagen in Prozent

Implementierung	Zeit	Präzision
SWI	930.8 ±33.7 ms	92.88
Sicstus	3579.8 ±66.3 ms	93.33
R	12.0 ±3.7 ms	93.77
Python	115.6 ±3.8 ms	93.77

4.2.2 Digits

Dieser Datensatz ist eine Kopie des Testsets von dem UCI ML handgeschriebenen Ziffern Datensatz [DG17]. Er enthält die Bilder von handgeschriebenen Ziffern. Somit sind die 10 Zielvariablen die Ziffern von 0 bis 9. Es sind insgesammt 1797 Bilder vorhanden. Die Bilder wurden so aufbereitet, dass sie durch eine 8×8 Matrix beschrieben werden können, in der jeder Eintrag eine Zahl in dem Intervall von 0 bis 16 ist. Dadurch haben die Instanzen des Datensatzes 64 Attribute.

Tabelle 5 zeigt sehr deutlich, dass meine Implementierung für große Datensätze nicht geeignet ist. Während Python und R im Schnitt unter einer Sekunde bleiben, braucht SWI knapp eine Minute und Sicstus davon mehr als das doppelte. Überraschenderweise ist die Präzision der R Umsetzung weit abgeschlagen im Vergleich zu den anderen Programmiersprachen.

Tabelle 5: Entscheidungsbaum Ergebnisse Digits. Die Spalte für die Zeit beinhaltet die durchschnittliche Laufzeit und den Standard Error of Mean in Millisekunden. Die Spalte Präzision beinhaltet die durchschnittliche Präzision der Vorhersagen in Prozent

Implementierung	Zeit	Präzision
SWI	58003.0 ±2819.1 ms	83.96
Sicstus	159970.0 ±3516.1 ms	83.96
R	110.0 ±< 0.1 ms	75.22
Python	15.6 ±< 0.1 ms	83.59

Durch Tabelle 6 bestätigt sich der Trend. R und Python bleiben unter einer Sekunde, während meine Implementierung über 600 benötigt. Außerdem liegt auch die Präzision leicht unter den anderen Implementierungen, das könnte aber mit der zufälligen Auswahl der Attribute zusammenhängen. Da für Sicstus-Prolog nach über 90 Minuten kein Ergebnis

vorlag wurde das Experiment abgebrochen.

Tabelle 6: Random Forest Ergebnisse Digits. Die Spalte für die Zeit beinhaltet die durchschnittliche Laufzeit und den Standard Error of Mean in Millisekunden. Die Spalte Präzision beinhaltet die durchschnittliche Präzision der Vorhersagen in Prozent

Implementierung	Zeit	Präzision
SWI	615226.6 ±14648.1 ms	95.52
Sicstus	-	-
R	480.0 ±12.6 ms	97.37
Python	268.7 ±3.1 ms	97.22

4.2.3 Diabetes

Der Diabetes Datensatz [EHJT04] besteht aus den 10 Attributen: „Alter in Jahren“, „Geschlecht“, „Body Mass Index“, „durchschnittlicher Blutdruck“ und sechs verschiedenen Blut Serum Messungen. Dies sind die gemessenen Attribute von den 442 Diabetes Patienten, die diesen Datensatz representieren. Dazu kommt die Zielvariable die den Fortschritt der Krankheit nach einem Jahr darstellt. Alle Attribute sind numerisch.

In diesen Datensatz fällt auf, dass meine Implementierung, zumindest von dem Mean Square Error her, die Python Implementierung schlägt. Dadurch, dass der Datensatz wieder kleiner ist, ist die Laufzeit bedeutend kleiner, allerdings immernoch weit entfernt von R und Python, wie man in Tabelle 7 sehen kann.

Tabelle 7: Entscheidungsbaum Ergebnisse Diabetes. Die Spalte für die Zeit beinhaltet die durchschnittliche Laufzeit und den Standard Error of Mean in Millisekunden. Die Spalte MSE beinhaltet den durchschnittlichen Mean Squared Error der Vorhersagen

Implementierung	Zeit	MSE
SWI	1917.2 ±61.4 ms	6272.5489
Sicstus	6860.6 ±222.8 ms	6272.5489
R	4.0 ±4.0 ms	4275.9852
Python	3.1 ±3.1 ms	6541.9654

Tabelle 8 bestätigt, dass auch im Fall der Regression meine Implementierung nicht gut skaliert.

Tabelle 8: Random Forest Ergebnisse Diabetes. Die Spalte für die Zeit beinhaltet die durchschnittliche Laufzeit und den Standard Error of Mean in Millisekunden. Die Spalte MSE beinhaltet den durchschnittlichen Mean Squared Error der Vorhersagen

Implementierung	Zeit	MSE
SWI	46008.2 ±1958.4 ms	3800.8435
Sicstus	138488.6 ±1704.5 ms	3737.4738
R	52.0 ±7.3 ms	3484.6973
Python	190.6 ±3.1 ms	3531.5891

4.2.4 California Housing

Der California Housing Datensatz [PB97] stammt aus dem StatLib Repository. Die Zielvariable ist der Median Haus Wert für Districte in Californien, ausgedrückt in 100.000 Dollar. Der Datensatz ist abgeleitet aus dem 1990 U.S Zensus. Jede Instanz aus dem Datensatz wird durch die Attribute „Median Einkommen“, „Median Haus Alter“, „durchschnittlische Anzahl Räume“, „durchschnittliche Anzahl Schlafzimmer“, „Bevölkerung“, „durchschnittlische Anzahl Hausbewohner“, „Geographische Breite“ und „Geographische Länge“.

Im Fall von SWI, als auch Sicstus-Prolog konnte nach 20 Minuten kein Entscheidungsbaum oder Random Forest, auf Grundlage, der ersten Train-Test-Aufteilung erstellt werden, weswegen, dass Experiment abgebrochen wurde.

Tabelle 9: Entscheidungsbaum Ergebnisse California. Die Spalte für die Zeit beinhaltet die durchschnittliche Laufzeit und den Standard Error of Mean in Millisekunden. Die Spalte MSE beinhaltet den durchschnittlichen Mean Squared Error der Vorhersagen

Implementierung	Zeit	MSE
SWI	-	-
Sicstus	-	-
R	200.0 ±3.2 ms	0.6052
Python	134.4 ±3.8 ms	0.5261

5 Zusammenfassung

In dieser Arbeit werden zunächst die theoretischen Grundlagen eingeführt, um ein besseres Verständnis für den Hauptteil zu bekommen. Dafür wird erklärt was ein Entscheidungsbaum ist und, wie der CART-Algorithmus einen erstellen kann, was ein Random Forest ist und wie es konstruiert und ausgewertet wird und Grundlegende Informationen zu der

Tabelle 10: Random Forest Ergebnisse Calinfornia. Die Spalte für die Zeit beinhaltet die durchschnittliche Laufzeit und den Standard Error of Mean in Millisekunden. Die Spalte MSE beinhaltet den durchschnittlichen Mean Squared Error der Vorhersagen

Implementierung	Zeit	MSE
SWI	-	-
Sicstus	-	-
R	16162.0 ±91.7 ms	0.2365
Python	8209.4 ±208.0 ms	0.2481

Programmiersprache Prolog, welche für den Hauptteil meiner Arbeit benutzt wird. Im nächsten Kapitel wird dann meine Implementierung für den CART-Algorithmus und die Entscheidungen, die ich während der Programmierung getroffen habe, vorgestellt. Zum Schluss werden zunächst die Grundlagen für den Vergleich der Implementierungen eingeführt und erklärt, um letztendlich die Ergebnisse von eben diesem vorzustellen.

Das Experiment hat gezeigt, dass meine Implementierung funktioniert. Die Präzision oder der Mean Squared Error ist dabei stets mit einer Referenz Implementierung vergleichbar. Allerdings fällt auf, dass je größer der Datensatz ist, meine Implementierung deutlich mehr Zeit für die Durchführung des Algorithmus braucht. Dazu kommt, dass die Ausführungszeit mit Sicstus-Prolog bemerkbar länger ist, als die von SWI-Prolog. Da die Datenmengen, die heutzutage zur Verfügung stehen, immer größer werden, ist es fragwürdig, ob eine Implementierung die so schlecht mit der Größe des Datensatzes skaliert Anwendung finden kann.

Der große Geschwindigkeitsunterschied lässt sich vielseitig begründen. Zunächst einmal ist die hier vorgestellte Implementation, im Rahmen einer Bachelorarbeit, also ungefähr in drei Monaten, entstanden. Somit kann der Code nicht zu einem Ausmaß optimiert sein, wie er es bei den Referenzimplementierungen ist, welche von großen Communitys über einen längeren Zeitraum gepflegt wurden. Außerdem ist der Alogrithmus an sich nicht optimiert. Ich habe mich in meiner Implementierung an dem grundlegenden CART-Algorithmus orientiert, jedoch wurden, wie bereits in meiner Arbeit erwähnt, schnellere Verfahren zur Aufteilung des Datensatzes vorgestellt [MS97] oder aber auch ein generell optimierter CART-Algorithmus [Cra89]. Die verwendeten Programmiersprachen spielen natürlich auch eine Rolle. In dieser wurde motiviert, dass meine Implementierung in puren Prolog geschrieben ist. Allerdings benutzt scikit-learn, beziehungsweise numpy, C/C++ im Hintergrund, um die Laufzeit des Programms deutlich zu verbessern. Im Falle der R Implementierung hat die Programmiersprache den Vorteil, dass sie darauf ausgelegt ist große Datenmengen effizient zu verarbeiten.

An meiner Implementierung lässt sich also noch einiges machen. Man könnte damit anfangen ein schnelleres Splitting Verfahren zu implementieren oder auch erstmal kleinschrittig den vorhandenen Code optimieren. Außerdem bietet Prolog eine Schnittstelle zu C und C++.

Somit könnte man die rechenintensiven Teile nach C/C++ auslagern um Zeit zu sparen.

Anhang

A Alle Ergebnisse

Tabelle 11: Ergebnisse für die Klassifikation. In der Spalte Datensatz Nr. befindet sich der benutzte Datensatz und welche Train-Test-Aufteilung benutzt wurde (1 - 5). Die Spalte Zeit beinhaltet die benötigte Trainingszeit in Millisekunden. Die Spalte Präzision beinhaltet die Präzision der Vorhersagen für den Durchlauf in Prozent. Die Spalte Typ zeigt an, ob es sich bei dem benutzten Algorithmus um einen Entscheidungsbaum (DT) oder Random Forest (RF) handelt.

Implementierung	Datensatz Nr.	Zeit	Präzision	Typ
SWI	Iris 1	52 ms	97.77	DT
SWI	Iris 2	23 ms	93.33	DT
SWI	Iris 3	17 ms	88.88	DT
SWI	Iris 4	18 ms	93.33	DT
SWI	Iris 5	20 ms	91.11	DT
SWI	Iris 1	1052 ms	100.00	RF
SWI	Iris 2	946 ms	91.11	RF
SWI	Iris 3	896 ms	91.11	RF
SWI	Iris 4	853 ms	93.33	RF
SWI	Iris 5	907 ms	91.11	RF
SWI	Digits 1	62921 ms	85.18	DT
SWI	Digits 2	65391 ms	83.14	DT
SWI	Digits 3	56592 ms	82.59	DT
SWI	Digits 4	49487 ms	86.85	DT
SWI	Digits 5	55624 ms	82.04	DT
SWI	Digits 1	664739 ms	97.04	RF
SWI	Digits 2	607559 ms	94.26	RF
SWI	Digits 3	600152 ms	95.18	RF
SWI	Digits 4	577392 ms	95.18	RF
SWI	Digits 5	626291 ms	95.74	RF
SICSTus	Iris 1	79 ms	97.77	DT
SICSTus	Iris 2	76 ms	93.33	DT
SICSTus	Iris 3	71 ms	88.88	DT
SICSTus	Iris 4	69 ms	93.33	DT
SICSTus	Iris 5	76 ms	91.11	DT
SICSTus	Iris 1	3679 ms	100.00	RF
SICSTus	Iris 2	3395 ms	91.11	RF
SICSTus	Iris 3	3452 ms	88.88	RF

Weiter auf der nächste Seite

Tabelle 11: Fortsetzung der Klassifikation

Implementierung	Datensatz Nr.	Zeit	Präzision	Typ
SICSTus	Iris 4	3637 ms	91.11	RF
SICSTus	Iris 5	3736 ms	93.33	RF
SICSTus	Digits 1	162956 ms	85.18	DT
SICSTus	Digits 2	165763 ms	83.14	DT
SICSTus	Digits 3	162231 ms	82.59	DT
SICSTus	Digits 4	146121 ms	86.85	DT
SICSTus	Digits 5	162779 ms	82.04	DT
R	Iris 1	$<$ 10 ms	97.77	DT
R	Iris 2	$<$ 10 ms	93.33	DT
R	Iris 3	$<$ 10 ms	88.88	DT
R	Iris 4	$<$ 10 ms	86.66	DT
R	Iris 5	$<$ 10 ms	93.33	DT
R	Iris 1	10 ms	100.00	RF
R	Iris 2	20 ms	93.33	RF
R	Iris 3	$<$ 10 ms	88.88	RF
R	Iris 4	10 ms	93.33	RF
R	Iris 5	20 ms	93.33	RF
R	Digits 1	110 ms	76.29	DT
R	Digits 2	110 ms	71.11	DT
R	Digits 3	110 ms	77.40	DT
R	Digits 4	110 ms	76.11	DT
R	Digits 5	110 ms	75.18	DT
R	Digits 1	460 ms	97.77	RF
R	Digits 2	470 ms	96.30	RF
R	Digits 3	530 ms	97.59	RF
R	Digits 4	470 ms	97.41	RF
R	Digits 5	470 ms	97.77	RF
Python	Iris 1	$<$ 16 ms	97.77	DT
Python	Iris 2	$<$ 16 ms	93.33	DT
Python	Iris 3	16 ms	88.88	DT
Python	Iris 4	$<$ 16 ms	93.33	DT
Python	Iris 5	$<$ 16 ms	91.11	DT
Python	Iris 1	110 ms	100.00	RF
Python	Iris 2	125 ms	93.33	RF
Python	Iris 3	125 ms	88.88	RF
Python	Iris 4	110 ms	93.33	RF
Python	Iris 5	125 ms	93.33	RF

Weiter auf der nächste Seite

Tabelle 11: Fortsetzung der Klassifikation

Implementierung	Datensatz Nr.	Zeit	Präzision	Typ
Python	Digits 1	16 ms	85.18	DT
Python	Digits 2	16 ms	82.04	DT
Python	Digits 3	16 ms	82.41	DT
Python	Digits 4	16 ms	85.74	DT
Python	Digits 5	16 ms	82.59	DT
Python	Digits 1	266 ms	97.22	RF
Python	Digits 2	266 ms	95.92	RF
Python	Digits 3	281 ms	97.96	RF
Python	Digits 4	266 ms	97.41	RF
Python	Digits 5	266 ms	97.60	RF

Tabelle 12: Ergebnisse für die Regression. In der Spalte Datensatz Nr. befindet sich der benutzte Datensatz und welche Train-Test-Aufteilung benutzt wurde (1 - 5). Die Spalte Zeit beinhaltet die benötigte Trainingszeit in Millisekunden. Die Spalte MSE beinhaltet den Mean Squared Error der Vorhersagen für den Durchlauf. Die Spalte Typ zeigt an, ob es sich bei dem benutzten Algorithmus um einen Entscheidungsbaum (DT) oder Random Forest (RF) handelt.

Implementierung	Datensatz Nr.	Zeit	Mean Squared Error	Typ
SWI	Diabetes 1	2043 ms	6373.8647	DT
SWI	Diabetes 2	1925 ms	6026.2707	DT
SWI	Diabetes 3	1686 ms	6515.2481	DT
SWI	Diabetes 4	1991 ms	6697.6391	DT
SWI	Diabetes 5	1941 ms	5749.7218	DT
SWI	Diabetes 1	53311 ms	3950.7812	RF
SWI	Diabetes 2	42426 ms	3439.3818	RF
SWI	Diabetes 3	42716 ms	3654.0344	RF
SWI	Diabetes 4	45052 ms	3818.4196	RF
SWI	Diabetes 5	42798 ms	3419.3666	RF
SICSTus	Diabetes 1	6405 ms	6373.8647	DT
SICSTus	Diabetes 2	6817 ms	6026.2707	DT
SICSTus	Diabetes 3	6569 ms	6515.2481	DT
SICSTus	Diabetes 4	7695 ms	6697.6391	DT
SICSTus	Diabetes 5	6817 ms	5749.7218	DT
SICSTus	Diabetes 1	134504 ms	4032.9350	RF
SICSTus	Diabetes 2	138064 ms	3538.8876	RF
SICSTus	Diabetes 3	144524 ms	3687.1283	RF
SICSTus	Diabetes 4	136209 ms	3937.2247	RF

Weiter auf der nächste Seite

Tabelle 12: Fortsetzung der Regression

Implementierung	Datensatz Nr.	Zeit	MSE	Typ
SICSTus	Diabetes 5	139142 ms	3415.3460	RF
R	Diabetes 1	< 10 ms	4672.6583	DT
R	Diabetes 2	20 ms	4907.4094	DT
R	Diabetes 3	< 10 ms	3852.7208	DT
R	Diabetes 4	< 10 ms	4386.8610	DT
R	Diabetes 5	< 10 ms	3560.2763	DT
R	Diabetes 1	80 ms	3690.6854	RF
R	Diabetes 2	40 ms	3569.2840	RF
R	Diabetes 3	40 ms	3221.6204	RF
R	Diabetes 4	50 ms	3714.2661	RF
R	Diabetes 5	50 ms	3227.6305	RF
R	California 1	200 ms	0.6002	DT
R	California 2	200 ms	0.6170	DT
R	California 3	210 ms	0.6265	DT
R	California 4	190 ms	0.6117	DT
R	California 5	200 ms	0.5709	DT
R	California 1	16110 ms	0.2296	RF
R	California 2	15910 ms	0.2362	RF
R	California 3	16220 ms	0.2460	RF
R	California 4	16470 ms	0.2464	RF
R	California 5	16100 ms	0.2246	RF
Python	Diabetes 1	< 16 ms	6357.8120	DT
Python	Diabetes 2	< 16 ms	6979.8646	DT
Python	Diabetes 3	< 16 ms	6397.5338	DT
Python	Diabetes 4	< 16 ms	6815.6090	DT
Python	Diabetes 5	16 ms	6159.0075	DT
Python	Diabetes 1	187 ms	3492.9041	RF
Python	Diabetes 2	203 ms	3847.5434	RF
Python	Diabetes 3	187 ms	3401.4384	RF
Python	Diabetes 4	187 ms	3768.4654	RF
Python	Diabetes 5	187 ms	3147.5942	RF
Python	California 1	125 ms	0.5590	DT
Python	California 2	141 ms	0.5507	DT
Python	California 3	141 ms	0.5074	DT
Python	California 4	141 ms	0.5340	DT
Python	California 5	125 ms	0.4793	DT
Python	California 1	8000 ms	0.2408	RF

Weiter auf der nächste Seite

Tabelle 12: Fortsetzung der Regression

Implementierung	Datensatz Nr.	Zeit	MSE	Typ
Python	California 2	8000 ms	0.2484	RF
Python	California 3	8110 ms	0.2606	RF
Python	California 4	7906 ms	0.2560	RF
Python	California 5	9031 ms	0.2347	RF

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Quellcodeverzeichnis

Literatur

[Alp22] Alpaydin, Ethem: *Maschinelles Lernen*. Berlin, Boston : De Gruyter Oldenbourg, 2022. http://dx.doi.org/doi:10.1515/9783110740196. http://dx.doi.org/doi:10.1515/9783110740196. – ISBN 9783110740196

[BFSO84] Breiman, L. ; Friedman, J. ; Stone, C.J. ; Olshen, R.A.: *Classification and Regression Trees*. Taylor & Francis, 1984 https://books.google.de/books?id=JwQx-WOmSyQC. – ISBN 9780412048418

[Bra01] Bratko, Ivan: *Prolog programming for artificial intelligence*. Pearson education, 2001

[Bre96] Breiman, Leo: Bagging predictors. In: *Machine learning* 24 (1996), Nr. 2, S. 123–140

[Bre01] Breiman, Leo: Random Forests. In: *Machine Learning* 45 (2001), Oct, Nr. 1, 5-32. http://dx.doi.org/10.1023/A:1010933404324. – DOI 10.1023/A:1010933404324. – ISSN 1573–0565

[CR93] Colmerauer, Alain ; Roussel, Philippe: The birth of Prolog. In: *HOPL-II*, 1993

[Cra89] Crawford, Stuart L.: Extensions to the CART algorithm. In: *International Journal of Man-Machine Studies* 31 (1989), Nr. 2, 197-217. http://dx.doi.org/https://doi.org/10.1016/0020-7373(89)90027-8. – DOI https://doi.org/10.1016/0020–7373(89)90027–8. – ISSN 0020–7373

[CWA+88] Carlsson, Mats ; Widen, Johan ; Andersson, Johan ; Andersson, Stefan ; Boortz, Kent ; Nilsson, Hans ; Sjöland, Thomas: *SICStus Prolog user's manual*. Swedish Institute of Computer Science Kista, 1988

[DG17] Dua, Dheeru ; Graff, Casey: *UCI Machine Learning Repository*. http://archive.ics.uci.edu/ml. Version: 2017

[DU20] Demidova, L A. ; Usachev, P O.: Development and approbation of the improved CART algorithm version. In: *Journal of Physics: Conference Series* 1479 (2020), mar, Nr. 1, 012085. http://dx.doi.org/10.1088/1742-6596/1479/1/012085. – DOI 10.1088/1742–6596/1479/1/012085

[EHJT04] Efron, Bradley ; Hastie, Trevor ; Johnstone, Iain ; Tibshirani, Robert: Least angle regression. In: *The Annals of statistics* 32 (2004), Nr. 2, S. 407–499

[Fis36] Fisher, Ronald A.: The use of multiple measurements in taxonomic problems. In: *Annals of eugenics* 7 (1936), Nr. 2, S. 179–188

[Izr14] Izrailev, Sergei: *tictoc: Functions for timing R scripts, as well as implementations of Stack and List structures.. R package version 1.0*. 2014

[Lew00] Lewis, Roger J.: An introduction to classification and regression tree (CART) analysis. In: *Annual meeting of the society for academic emergency medicine in San Francisco, California* Bd. 14 Citeseer, 2000

[LSW15] Liu, Changwei ; Singhal, Anoop ; Wijesekera, Duminda: A Logic Based Network Forensics Model for Evidence Analysis, Advances in Digital Forensics XI, Orlando, FL, US, 2015-01-28 00:01:00 2015

[M+11] McKinney, Wes u. a.: pandas: a foundational Python library for data analysis and statistics. In: *Python for high performance and scientific computing* 14 (2011), Nr. 9, S. 1–9

[Mer12] Merritt, Dennis: *Building expert systems in Prolog.* Springer Science & Business Media, 2012

[MS97] MOLA, FRANCESCO ; SICILIANO, ROBERTA: A fast splitting procedure for classification trees. In: *Statistics and Computing* 7 (1997), Sep, Nr. 3, 209-216. http://dx.doi.org/10.1023/A:1018590219790. – DOI 10.1023/A:1018590219790. – ISSN 1573–1375

[PB97] Pace, R K. ; Barry, Ronald: Sparse spatial autoregressions. In: *Statistics & Probability Letters* 33 (1997), Nr. 3, S. 291–297

[PVG+11] Pedregosa, F. ; Varoquaux, G. ; Gramfort, A. ; Michel, V. ; Thirion, B. ; Grisel, O. ; Blondel, M. ; Prettenhofer, P. ; Weiss, R. ; Dubourg, V. ; Vanderplas, J. ; Passos, A. ; Cournapeau, D. ; Brucher, M. ; Perrot, M. ; Duchesnay, E.: Scikit-learn: Machine Learning in Python. In: *Journal of Machine Learning Research* 12 (2011), S. 2825–2830

[Qui93] Quinlan, J. R.: *C4.5: programs for machine learning.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1993 http://portal.acm.org/citation.cfm?id=152181. – ISBN 1–55860–238–0

[RL18] RColorBrewer, Suggests ; Liaw, Maintainer A.: Package 'randomforest'. In: *University of California, Berkeley: Berkeley, CA, USA* (2018)

[SH10] Sridhar, Meera ; Hamlen, Kevin W.: ActionScript in-lined reference monitoring in Prolog. In: *International Symposium on Practical Aspects of Declarative Languages* Springer, 2010, S. 149–151

[Sho14] Shoham, Yoav: *Artificial intelligence techniques in Prolog.* Morgan Kaufmann, 2014

[SK16] Sharma, Himani ; Kumar, Sunil: A survey on decision tree algorithms of classification in data mining. In: *International Journal of Science and Research (IJSR)* 5 (2016), Nr. 4, S. 2094–2097

[SMT09] Strobl, Carolin ; Malley, James ; Tutz, Gerhard: An introduction to recursive partitioning: rationale, application, and characteristics of classification and regression trees, bagging, and random forests. In: *Psychol Methods* 14 (2009), Dezember, Nr. 4, S. 323–348

[TARR15] Therneau, Terry ; Atkinson, Beth ; Ripley, Brian ; Ripley, Maintainer B.: Package 'rpart'. In: *Available online: cran. ma. ic. ac. uk/web/packages/rpart/rpart. pdf (accessed on 20 April 2016)* (2015)

[WS16] Wicaksono, Handy ; Sammut, Claude: Relational tool use learning by a robot in a real and simulated world. In: *Proceedings of ACRA*, 2016

[WSTL12] Wielemaker, Jan ; Schrijvers, Tom ; Triska, Markus ; Lager, Torbjörn: SWI-Prolog. In: *Theory and Practice of Logic Programming* 12 (2012), Nr. 1-2, S. 67–96. – ISSN 1471–0684

[ZUB20] Zombori, Zsolt ; Urban, Josef ; Brown, Chad E.: Prolog technology reinforcement learning prover. In: *International Joint Conference on Automated Reasoning* Springer, 2020, S. 489–507