**Dokumentation zu den Programmen in Java  
Data Mining mit Entscheidungsbäumen**

Inhaltsverzeichnis

[1. Einordnung des Themas 2](#_Toc74157386)

[2. Kurze Zusammenfassung der Dokumentation 2](#_Toc74157387)

[3. Bearbeitung der Aufgaben 3](#_Toc74157388)

[Aufgabe 2.1: Vorverarbeitung der Daten 3](#_Toc74157389)

[1.1 CSVReader 3](#_Toc74157390)

[1.2 BinningDiscretizer 3](#_Toc74157391)

[1.3 Categorizer 3](#_Toc74157392)

[1.4 Categoric 3](#_Toc74157393)

[1.5 Continuously 3](#_Toc74157394)

[Aufgabe 2.2: Entscheidungsbaum lernen mit ID3 4](#_Toc74157395)

[2.1 ID3Utils 4](#_Toc74157396)

[2.2 DecisionTreeNode 4](#_Toc74157397)

[2.3 EntropyUtils 4](#_Toc74157398)

[Aufgabe 2.3: Reduced Error Pruning 5](#_Toc74157399)

[3.1 ReducedErrorPruner 5](#_Toc74157400)

[3.2 CrossValidator 5](#_Toc74157401)

[Aufgabe 2.4: Ausgabe des Entscheidungsbaums 6](#_Toc74157402)

[4.1 XMLWriter 6](#_Toc74157403)

[Aufgabe 2.5: Rapidminer 7](#_Toc74157404)

[5 Kurze Reflexion der Problematiken und Umsetzung 8](#_Toc74157405)

[6 Design Entscheidungen 8](#_Toc74157406)

# Einordnung des Themas

Dieses Programm ist in der Lage ein eine CSV-Datei einzulesen und auf Grundlage der eigelesenen Daten einen Entscheidungsbaum zu generieren. Hierbei wird der ID3 Algorithmus aus der Vorlesung genutzt, der durch Rekursion den Baum rekursiv generiert. Anschließend kann man den Baum als XML-Datei ausgeben lassen oder den Baum zur weiteren Informationsverarbeitung nutzen… NOT FINAL

# Kurze Zusammenfassung der Dokumentation

In dieser Dokumentation wird die Funktionalität sowie das Vorgehen bei der Erstellung der verschiedenen Programme erläutert. Einige dieser Programme wurden mit Graphiken zur Veranschaulichung und mit Referenzen zu ihren Code Zeilen ergänzt.

XXX

# Bearbeitung der Aufgaben

## Aufgabe 2.1: Vorverarbeitung der Daten

### CSVReader

Die CSVReader Klasse wir zur einlesen von Daten aus einer CSV-Datei genutzt. Hierbei liest die Klasse über readCsvToArray und einen Buffered Reader Zeilenweise ein uns zerlegt jede Zeile in ihre einzelnen Attribute. Nach der Zerlegung werden die Daten in einer Liste zusammengefasst im Return zurückgegeben. Hierbei wird noch geguckt, ob wir den ersten Eintrag ausschließen sollen oder nicht, dies hängt davon ab, ob es sich hierbei um den Header handelt.

### BinningDiscretizer

Die BinningDiscretizer Klasse legt Intervall Bereiche für continuous Attribute fest und teilt die übergebenen Daten den entsprechenden Bereichen zu.

Die Methode discretize erhält hierfür die Anzahl der Intervalle, eine Liste von Daten die diskretisiert werden sollen sowie eine Attribut ID, welche angibt welches Attribut diskretisiert wird. Zuerst werden die vom Categorizer übergebenen Daten in Spaltenform umgewandelt (Zeile 28-30). Anschließend wird der minimale und maximale Wert des Attributes mit linearer Suche ermittelt (Zeile 37-40). Anschließend wird die Intervallgröße ausgerechnet indem das Minimum vom Maximum subtrahiert wird und durch die Anzahl der gewünschten Intervalle dividiert wird. Diese Intervallgröße wird in einem statischen Array für die spätere Verwendung gespeichert (Zeile 46-47). Als letzter Schritt wird durch alle übergebenen Daten iteriert und ausgerechnet in welchem Intervall der Eintrag liegt. Hierzu wird das Minimum vom hinterlegten Wert subtrahiert und durch die Anzahl der Intervalle dividiert. Das ausgerechnete Intervall wird dann im jeweiligen Objekt in der Variable category gespeichert (Zeile 50-56). Zurückgegeben wird eine Liste mit diskretisierten Daten.

### Categorizer

Die Categorizer Klasse dient dem Umwandeln der Eingabedaten in CSVAttributes (Categoric & Continuously) sowie dem Aufrufen des BinningDiscretizers für Continuously Attribute. Die Klasse umfasst die Hauptmethode categorize sowie die Hilfsfunktion isCategoric.

Die Methode categorize erhält eine Liste von String Arrays und einen Integer als Übergabeparameter. Die Liste umfasst die Daten, die kategorisiert werden sollen und der Integer gibt an wie viele bins beim Aufrufen des BinningDiscretizers verwendet werden sollen. Da die Listeneinträge in Zeilenform vorliegen werden diese zunächst in Spaltenform umgewandelt (Zeile 24-26). Anschließend wird für jede Spalte mit der Hilfsfunktion isCategoric getestet, ob ein Categoric oder Continuously Attribut vorliegt und in einem boolean Array abgespeichert (Zeile 29-30). Als nächstes wird spaltenweise über alle Daten iteriert und anhand dem boolean Array ein entsprechendes CSVAttribute Objekt angelegt, welches in einem CSVAttribute Array gespeichert wird (Zeile 33-35). Als vorletzter Schritt wird der BinningDiscretizer für alle Attribute aufgerufen die als continuous identifiziert wurden (Zeile 38-40). Die Methode gibt am Ende eine Liste von CSVAttribute Arrays zurück.

Die Hilfsmethode isCategoric erhält als Übergabeparameter eine Liste von Strings,welche die einzigartigen Vorkommen der Attribute repräsentieren. Es wird nun in einem try-catch Block versucht den String zu einem Double zu parsen (Zeile 55). Falls dies eine Exception wirft, wird im catch ein “true” zurückgegeben (Zeile 57), da es sich nicht um eine Zahl handelt. Falls der String erfolgreich geparst werden kann, wird noch überprüft ob nur zwei einzigartige Werte vorliegen. In diesem Fall ist das Attribut binomial und damit kategorisch weshalb ein “true” returned wird. In allen anderen Fällen wird ein “false” zurückgegeben, da es sich um ein continous Attribut handelt.

### Categoric

Die Klasse Categoric dient dem Darstellen von kategorischen Daten. Sie implementiert das Interface CSVAttribute und hat zwei Klassenvariablen sowie dazugehörige Getter & Setter Methoden.

Die Variable “value” speichert den ursprünglichen Wert aus der CSV Datei.

Die Variable “category” speichert die Ausprägung des Attributs dem dieses Object zugehört. Dies entspricht dem gleichen Wert wie “value” aber wird dennoch benötigt, um einen generischen Programmablauf zu garantieren.

### Continuously

Die Klasse Continously dient dem Darstellen von continous Daten. Sie implementiert das

Interface CSVAttribute und hat zwei Klassenvariablen sowie dazugehörige Getter & Setter Methoden sowie eine compareTo Methode.

Die Variable “value” speichert den ursprünglichen Wert aus der CSV Datei.

Die Variable “category” speichert das Intervall des Attributs dem dieses Object angehört. Dieser Wert wird nicht beim Erstellen des Objects belegt, sondern erst wenn das Attribut dem das Object angehört, diskretisiert wird.

Die compareTo Methode ermöglicht das Vergleichen von Objekten dieser Klasse. Es wird

dazu das “value” der eigenen sowie der Vergleichsinstanz in ein double geparst. (Zeile

39-40). Anschließend werden diese doubles miteinander verglichen und

* 1 zurückgegeben, falls die Vergleichsinstanz größer als die eigene ist
* 1 zurückgegeben, falls die Vergleichsinstanz kleiner als die eigene ist
* 0 zurückgegeben, falls die Vergleichsinstanz gleich der eigenen ist

## Aufgabe 2.2: Entscheidungsbaum lernen mit ID3

### ID3Utils

Der eigentliche Baum wird durch ID3Utils generieret, hierbei handelt es sich um einen Rekursiven Algorithmus, der nach dem Motto der Tiefensuche arbeitet. Dabei berechnet der Algorithmus am Anfang immer die Entropie der Momentanen Daten (CZ 37) und sucht sich dann die Daten mit dem höchsten Gain raus (CZ 41-46). Dabei müssen wir bei der Auswahl jedoch achten, dass wir nur Attribute auswählen können, die vorher noch nicht ausgewählt waren und die nicht das Label Attribut sind. Anschließend markieren wir, dass wir das ausgewählte Attribut mit dem höchsten Gain jetzt besucht haben (CZ 48) und gehen in den Rekursiven Aufruf, um den Rest des Baumes zu generieren (CZ 51-60). Hierbei gehen wir nur so lange in die Tiefe wie es Sinnvoll ist (CZ 52). Ist eine weitere Verzweigung nicht sinnvoll, dann geben wir den Momentanen Knoten als Blattknoten zurück und setzen den Wert des Knoten als Label Wert (CZ 61-63). Da Blätter nicht immer zu 100% eindeutig sein können, weil es z.B. positive und negative Einträge gibt, müssen wir die Mehrzahl als Label wählen (CZ 67-86).

Ob es sinnvoll ist, weitere Verzweigungen zu bilden, finden wir über MoreOptions und sowie den Gain Wert raus. Falls es keine Weiten Attribute gibt, dann haben wir die Maximale Tiefe des Baumes erreicht und können keine weiteren Verzeigungen Bilden. Falls der Gain 0 ist, dann wissen wir, dass wir keine weiteren nützlichen Informationen aus den Daten ziehen können, weitere Verzweigungen wären auch hier nicht sinnvoll.

Falls es sinnvoll ist weitere Verzweigungen zu bilden, dann gehen wir in die Schleife. Diese wird die Daten nach ihren Ausprägungen aufteilen und in einzelne Listen unterteilen (CZ 53 & 96-105). Für jede Unterteilung wird dann der Rekursive Aufruf, aufgerufen, um den Unterbaum rekursive zu erzeugen (CZ 54). Der Resultierende Unterbaum ist dann das Kind des momentanen Knotenpunktes. Dem momentanen Knoten müssen wir dann noch einen Verweis zu seinen Kindern geben (CZ 56) und den Kindern einen Verweis zu ihrem Elternknoten (CZ 57). Anschließend geben wir den Knoten dann im return an die obere Rekursion zurück (CZ 63).

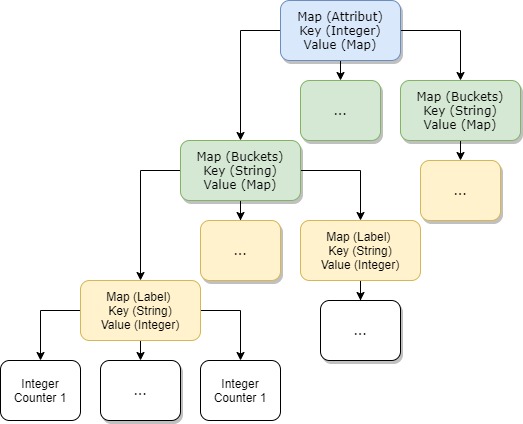
### DecisionTreeNode

Hierbei handelt es sich um eine einfache Klasse zur Repräsentation von Knoten Punkten im Baum. Jeder Knotenpunkt hat hierbei einen Verweis auf sein Überknoten (CZ 7) und seine Unterknoten (CZ 9) sowie auf den Index, welcher er in den Daten repräsentiert (CZ 8). Bei der Erstellung eines Knoten Punktes müssen wir sofort im Konstruktor sagen, welcher Index repräsentiert wird.

Über „getter“ und „setter“ Methoden könne Attribute erhalten / setzen.

### EntropyUtils

Die Klasse EntropyUtils wird genutzt, um die Entropie zu jedem Attribut zu berechnen. Hierbei wurde eine Baumstruktur genutzt, um die Verteilung der Attribute sowie deren Ausprägungen in Relation zu den Labels zu berechnen.

(Blaue) repräsentiert alle möglichen Attribute (CreditScore;Geography…), jedes Attribut verweist auf die jeweiligen Ausprägungen (Buckets).

(Grün) repräsentiert alle möglichen Ausprägungen (Buckers) eines Attributes (Gender 🡪 Female), jeder Bucket verweist auf die möglichen Labels.

(Gelb) repräsentiert alle möglichen Labels (Gender 🡪 Female 🡪 0) einer Ausprägung (Bucket), jedes Label verweist auf die Anzahl der Vorkommen.

Jetzt wo die Datensätze in einer geordneten Form vorliegen, können wir über die Map Iterieren und für jedes Attribut die Entropie brechen. Hierbei gehen wir nach der Formel aus der Vorlesung vor.

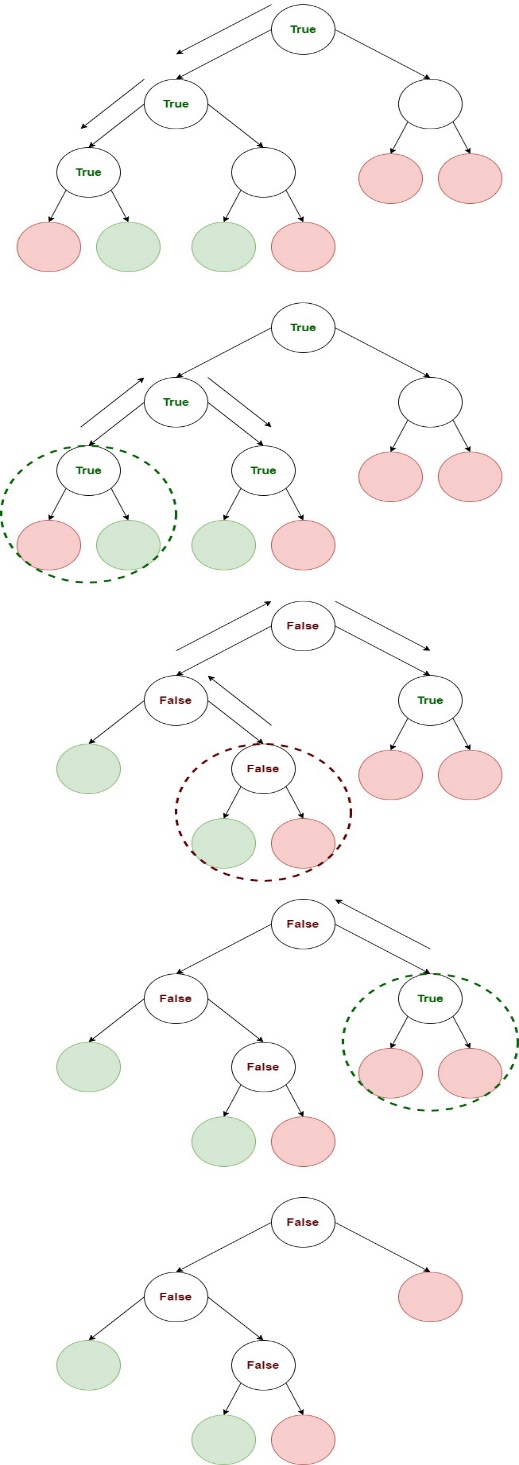
Zuerst müssen wir die Entropie für unsere Label berechnen, dafür schauen wir uns in der Map das Label Attribut an. Hier werden wir dann die gesamten Aufzählungen für alle möglichen Ausprägungen wiederfinden (Bsp. 0/1). Diese übergeben wir dann an HE(), welche uns einen Startwert zurückgibt. Auf Grundlage dessen werden die Gains für alle restlichen Attribut berechnet.

Nach dem Berechnen der Entropie für unser Label, können wir die Entropie für alle restlichen Attribute berechnet, dies erfolgt über einen Stream, welcher über alle Attribute und deren Ausprägungen streamt und diese an H() übergibt, diese Funktion wird uns die Rest Entropie für eine Ausprägung zurückgeben. Diese Rest Entropie müssen wir dann mit allen anderen Ausprägenden des momentanen Attributs summieren und von der Start Entropie abziehen. Der Resultierende Wert wird der Gain für das gewählte Attribut sein. Dieser Vorgang wird für jedes Attribut wiederholt. Schlussendlich fangen wir alle Gains in einer Liste auf und geben diese im return zurück

## Aufgabe 2.3: Reduced Error Pruning

### ReducedErrorPruner

Hierbei handelt es sich um ein "Brute Force" der nach dem Motto "einmal alles" vorgeht. Dabei geht er rekursiv durch den Baum und versucht dann von unten nach oben Knoten zu kombinieren. Der Algorithmus kann aber nicht wissen, nach welchen Kriterien er kombinieren soll, deshalb versucht er alle möglichen Kombinationen aus und entscheidet sich für die beste. Auf Grundlage der Tatsache, dass der Algorithmus sehr viele Sachen ausprobieren muss und dies viel Zeit kostet, wurden verschiedene Abbruchbedingungen eingebaut, die unnötiges Ausprobieren verhindern.

Grundsätzlich können wir immer nur Blattknoten kombinieren, deshalb können wir auch davon ausgehen, dass wenn der untere Knoten nicht gepruned werden kann, dann können wir auch nicht den oberen Knoten prunen. Nach diesem Prinzip Iterieren wir rekursiv durch den Baum, bis wir auf ein Blattknoten stößen (CZ 39-41 & 36). Wir versuchen diesen dann mit seinen Geschwisterblattknoten zu Kombinerien. Da wir nicht wissen, welche Kombination zu dem besten Ergebnis führt, müssen wir einmal alles ausprobieren. Dabei merken wir uns aber, welcher Versuch der Beste war (CZ . 53–64)

Falls wir alles versucht haben, aber keines der Kombinationen ein besseres Ergebnis erzeugt hat, dann wissen wir, dass wir den Zweig nicht prunen können. Falls wir aber eine Kombination gefunden haben, die besser oder gleich der alten ist, dann übernehmen wir die Kombination in unseren alten Baum (CZ 66-69). Dies versuchen wir nun bei allen Knoten, die noch zum Prunen offen stehen.

Um festzustellen, ob eine Kombination besser oder schlechter als die alte ist, müssen wir alle Möglichkeiten ausprobieren und die Genauigkeit berechnen, anhand dieser Genauigkeit können wir feststellen, ob sich durch die Veränderung der Baum verschlechtert oder verbessert (CZ 57-64).

Hierbei ist zu beachten, dass nur mit "True“ markierten Knoten auch führ das prunen offenstehen (CZ 38). Falls also der untere Knoten nicht gepruned werden kann, dann teilt dieser das auch seinem Überknoten mit, welcher das gleiche bei seinem Überknoten macht, etc (CZ 43).

Dieser Vorgang läuft so lange, bis wir alle Knoten gepruned oder als unprunebar markiert haben. Nach dem Prunen erhalten wir einen Baum in den Redundanzen und Extrema entfernt wurden und nur noch die wichtigsten Knotenpunkte enthält.

### CrossValidator

Der CrossValidator wird allgemein genutzt, um den besten Baum aus einer beliebigen Anzahl an Testdurchläufen zu erhalten. Hierbei benötigt die Methode performCrossValidation die Daten, mit denen wir arbeiten, den Algorithmus, nach dem der Baum gebaut wird und die Position des Label Indexes.

Sobald diese Daten vorliegen, wird der Algorithmus die Daten (je nach Anzahl der Folds) in gleichgroße Stücke zerlegen (CZ 32-36). Wir entnehmen dann jeweils iterativ immer eines dieser Stücke, um den Baum zu testen und der Rest der Daten wird zur Generierung des Baumes genutzt (CZ 38-39) .Nach jeder Iteration müssen wir noch testen, wie gut der Baum ist, dies erreichen wir, indem wir die Testdaten an den Baum übergeben (CZ 40) und schauen wieviel Prozent an der richtigen Position rauskommen (CZ 43). Anschließen vergleichen wir die Bäume miteinander uns merken uns den Besten (CZ 44-47). Am ende der Iterationen geben wir den besten Baum im Return zurück.

## Aufgabe 2.4: Ausgabe des Entscheidungsbaums

### XMLWriter

Die XMLWriter Klasse serialisiert einen Entscheidungsbaum als XML Datei. Sie umfasst eine Hauptmethode writeXML und eine Hilfsmethode addXmlChildren. Die writeXML Methode schreibt einen jdom Tree als XML Datei in den übergebenen Pfad. Die addXmlChildren Methode dient dem rekursiven Aufbau eines jdom Trees der dem übergebenen Entscheidungsbaum entspricht.

In der writeXML Methode wird zuerst die Formatierung der XML Datei festgelegt (Zeile 26-27). Anschließend wird das Dokument, der Wurzelknoten sowie der Knoten des ersten Attributs erstellt (Zeile 35-38). Anschließend wird die Hilfsmethode addXmlChildren aufgerufen die rekursiv den gesamten Jdom Tree erstellt (Zeile 38). Zuletzt wird dann die XML Datei in den übergebenen Pfad geschrieben.

Die addXmlChildren Methode prüft für alle Kinder der übergebenen DecisionTreeNode ob es sich um Leafs (null-Verweis) oder normale Knoten handelt (Zeile 55). Handelt es sich um einen normalen Knoten wird ein Element erstellt, welches den Namen If erhält und ein Attribut mit dem entsprechenden Auswahlkriterium (numerisches Intervall oder Name der kategorischen Ausprägung). Anschließend wird das Knoten Element mit dem Namen Node und einem Attribut, das die entsprechende Attribut ID angibt erstellt (Zeile 57-68). Anschließend erfolgt ein rekursiver Aufruf der addXmlChildren Methode für den eben benutzen Knoten (Zeile 69). Falls es sich bei einem Knoten jedoch um ein Blatt handelt wird der entsprechende Knoten mit einem Blattknoten ersetzt (Zeile 72-76).

## Aufgabe 2.5: Rapidminer

# Kurze Reflexion der Problematiken und Umsetzung

Auf Grundlage der Reorganisation hatten wir anfängliche Probleme mit der Organisation sowie der Synchronisation der Daten. Dieses Problem wurde dann schlussendlich von Herrn Moll, welcher die Nutzung von GitHub vorschlug, gelöst. Dies erlaubte es uns dann schnell und synchron and den Aufgaben im Team zu arbeiten.

Nach den anfänglichen Synchronisation Probleme, haben wir öfters Backtracking betreiben müssen, weil wir Fehler in Programmen hatten, die in späteren Iterationen komplett umgeschrieben werden mussten, diese Fehlerbehebung hat uns schlussendlich viel Zeit gekostet, uns aber auch die Möglichkeit gegeben uns näher mit verschieden Prozessen zu beschäftigen und diese zu verinnerlichen.

Es wurde versucht das Programm so effizient wie möglich zu haten, hierbei wurden Bibliotheken genutzt, die in Relation der Aufgaben am besten geeignet waren. Wir konnten somit effizient die Daten durch ein Buffer Reader einlesen. Der Umgang mit großen Daten wurde durch Stream abgewickelt, die es erlaubten sehr schnell über die Daten zu gehen.

# Design Entscheidungen