**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

# 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

### Categorizer

### Categoric

### Continuously

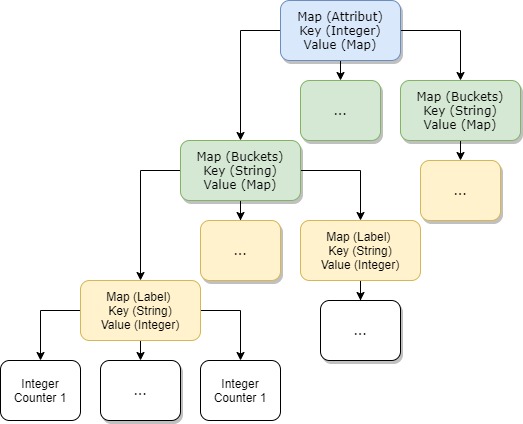
## Aufgabe 2.2: Entscheidungsbaum lernen mit ID3

### ID3Utils

### DecisionTreeNode

### 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.

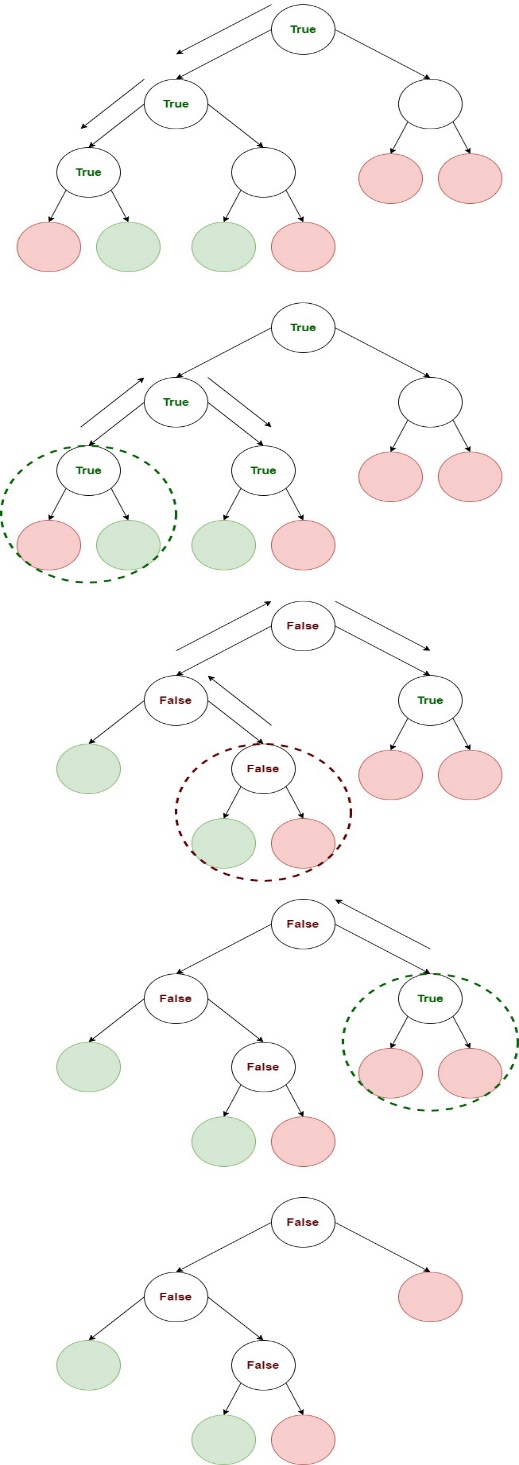
Zuerst müssen wir die Entropie für unsere Label berechnen, dafür schauen wir uns in der Map das Label Attribut an. Hier werden wir dann die gesamten Aufzählungen für alle möglichen Ausprägungen wiederfinden (Bsp. 0/1). Diese übergeben wir dann an HE(), welche uns einen Startwert zurückgibt. Auf Grundlage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

## Aufgabe 2.4: Ausgabe des Entscheidungsbaums

### XMLWriter

## 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