# Machine Learning
# Blatt 2

Markus Vieth, David Klopp, Christian Stricker

5. Mai 2016

# Nr.1

## Code

```
1  import org.kramerlab.teaching.ml.datasets.*;

3  import java.io.File;
4  import java.util.*;

6  //TODO exceptions
7  /**
8   * Created by David Klopp, Christian Stricker, Markus Vieth on 21.04.2016.
9   */
10 public class DecisionTree {

12     /**
13      * Inner Node class
14      */
15     private class Node {
16         Node parent;
17         Attribute attribute = null;
18         List<Edge> edges = new ArrayList<>();
19         List<Integer> indices;
20         List<Attribute> notVisited;
21         boolean isSingleNode = false;
22         Value value;

24         /**
25          * Constructor
26          * @param indices
27          */
28         public Node(List<Integer> indices, Node parent) {
29             this.indices = indices;
30             this.parent = parent;
31             if (parent == null) {
32                 notVisited = dataset.getAttributes();
33                 notVisited.remove(classAttribute);
34             } else {
35                 notVisited = parent.notVisited;
36             }
37         }

39         /**
40          * adds edge
41          * @param edge
42          */
43         public void addEdge(Edge edge) {
44             this.edges.add(edge);
45         }

47         /**
48          * get value if single node else null
49          * @return
50          */
51         public Value getValue() {
52             if (isSingleNode)
53                 return value;
54             return null;
55         }
```

```
57          /**
58           * prints tree recursive (still needs some work)
59           * @param prefix
60           */
61          public void print(String prefix) {
62              if (isSingleNode) {
63                  System.out.println(prefix + " " + value);
64              } else {
65                  System.out.println(prefix + " " + attribute);
66              }
67              for(Edge edge : edges) {
68                  edge.end.print(prefix+'-');
69              }
70          }

72          /**
73           * sets attribute and removes from not visited
74           * @param attribute
75           */
76          public void setAttribute(Attribute attribute) {
77              this.attribute = attribute;
78              this.notVisited.remove(attribute);
79          }
80      }

82      /**
83       * Edge class
84       */
85      private class Edge {
86          Value value = null;
87          Node start;
88          Node end;

90          /**
91           * Constructor
92           * @param value
93           */
94          public Edge(Value value, Node start) {
95              this.value = value;
96              this.start = start;
97              this.start.addEdge(this);
98          }
99      }

101     private Node root = null;

103     private Instance[] data;
104     private Dataset dataset;
105     private Attribute classAttribute;

107     /**
108      * default constructor
109      */
110     public DecisionTree() {
111     }

113     //--------------------------------------------------------------------------
114     //-------------------------Train tree---------------------------------------
115     //--------------------------------------------------------------------------

117     /**
```

```
118      * chooses the attribute with the best information gain
119      * @param node
120      * @return
121      */
122     public Attribute selectAttribute(Node node) {
123         Attribute select = null;
124         double maxGain = Double.NEGATIVE_INFINITY;
125
126         // looks at all relevant attributes
127         for (Attribute attribute : node.notVisited) {
128
129             // does not look at classAttribute
130             /*if (attribute.equals(classAttribute)) {
131                 continue;
132             }*/
133             double gain = this.informationGain(attribute, node.indices);
134             if (gain > maxGain) {
135                 select = attribute;
136                 maxGain = gain;
137             }
138         }
139         return select;
140     }
141
142     /**
143      *
144      */
145     public void train(List<Integer> trainset) {
146         // create root Node
147         this.root = new Node(trainset, null);
148         this.train_recursive(this.root);
149     }
150
151     /**
152      * Internal method
153      * @param n
154      */
155     public void train_recursive(Node n) {
156
157         // exit function
158         if (this.isSingleNode(n)) {
159             return;
160         }
161         //select attribute with biggest informationGain
162         n.setAttribute(this.selectAttribute(n));
163
164         // create edges for each value of the attribute
165         NominalAttribute attr = (NominalAttribute)n.attribute;
166         for (int i = 0; i < attr.getNumberOfValues(); i++) {
167             Value v = attr.getValue(i);
168             Edge edge = new Edge(v, n);
169             edge.end = new Node(new ArrayList<>(), n);
170         }
171
172
173
174         for (Integer idx : n.indices) {
175             Instance i = this.data[idx];
176             Value v = i.getValue(n.attribute);
177
178             // add index to right edge
```

```
179             for (Edge edge : n.edges) {
180                 if (edge.value.equals(v)) {
181                     edge.end.indices.add(idx);
182                     break;
183                 }
184             }
185         }
186         // create tree
187         for (Edge e : n.edges) {
188             this.train_recursive(e.end);
189         }
190     }

192     /**
193      * Tests if node has only one class left and sets node.isSingleNode
194      * @param node
195      * @return
196      */
197     private boolean isSingleNode(Node node) {

199         // if node is empty we set his value to the most common
200         // value of his parent
201         if (node.indices.size() == 0) {
202             node.value = mostCommonValue(node.parent, this.classAttribute);
203             node.isSingleNode = true;
204             return true;
205         }

207         // should not be possible anymore
208         // if last node we set value to most common value
209         /*if ( node.notVisited.size() == 1
210                 && node.notVisited.contains(classAttribute)) {
211             node.value = mostCommonValue(node, this.classAttribute);
212             node.isSingleNode = true;
213             return true;
214         }*/

216         /*// should not be possible, else like the one over this*/
217         // if last node we set value to most common value
218         if ( node.notVisited.size() == 0 ) {
219             node.value = mostCommonValue(node, this.classAttribute);
220             node.isSingleNode = true;
221             return true;
222         }

224         Value value = data[node.indices.get(0)].getValue(classAttribute);

226         // if one instance has an other value as the rest return false
227         for (int i = 1; i < node.indices.size(); i++) {
228             Instance instance = data[node.indices.get(i)];
229             if (! instance.getValue(classAttribute).equals(value)) {
230                 return false;
231             }
232         }

234         // else set as single node and return true
235         node.isSingleNode = true;
236         node.value = value;
237         return true;
238     }
```

```
241     //-------------------------------------------------------------------------
242     //-------------------------classify tree-----------------------------------
243     //-------------------------------------------------------------------------


246     /**
247      * classifies given data set on decision Tree
248      * @param data
249      * @return
250      */
251     public double classify(List<Integer> data) {
252         //TODO test if tree is build
253         int correctlyClassified = 0;
254         // repeat for each instance
255         for (Integer i : data) {
256             Instance instance = this.data[i];

258             // iterate over tre
259             Node currentNode = this.root;
260             while (!currentNode.isSingleNode) {
261                 Attribute attr = currentNode.attribute;
262                 Value value = instance.getValue(attr);

264                 // find right edge
265                 for (Edge edge : currentNode.edges) {
266                     if (edge.value.equals(value)) {
267                         currentNode = edge.end;
268                         break;
269                     }
270                 }
271             }

273             // check if class attr is correct
274             if ( currentNode.getValue().equals(instance.getValue(classAttribute)) ) {
275                 correctlyClassified ++;
276             }

278         }

280         return (double)correctlyClassified/(double)data.size();
281     }




286     //-------------------------------------------------------------------------
287     //-------------------------Constructor-------------------------------------
288     //-------------------------------------------------------------------------

290     /**
291      * returns most common value from node in attribute
292      * @param node
293      * @param attribute
294      * @return
295      */
296     private Value mostCommonValue(Node node, Attribute attribute) {
297         // because attribute dose not know his values
298         Map<Value, Integer> numValues = new HashMap<Value, Integer>();
299         // temp
300         Value max = null;
```

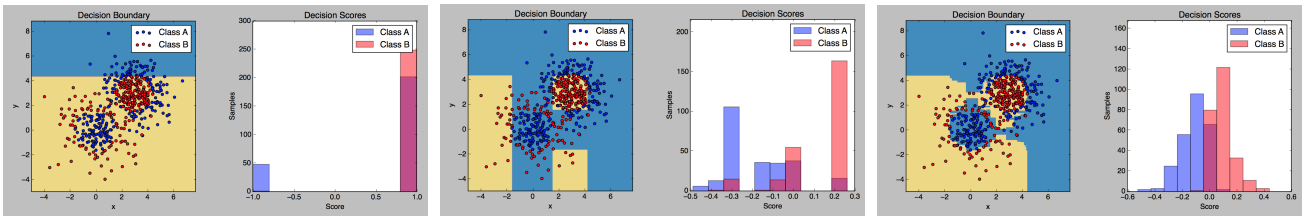
```
301         int maxInt = -1;

303         for (Integer i : node.indices) {
304             Instance instance = data[i];
305             Value value = instance.getValue(attribute);
306             Integer integer = 1;
307             if(numValues.containsKey(value)) {
308                 integer = numValues.get(value);
309                 integer++;
310                 numValues.replace(value, integer);
311             } else {
312                 numValues.put(value, integer);
313             }

315             if (integer > maxInt) {
316                 max = value;
317             }
318         }

320         return max;
321     }

323     /**
324      * loads arff
325      * @param path path to arff
326      */
327     public DecisionTree(String path) {
328         try {
329             this.loadArff(path);
330         } catch (Exception e) {
331             e.printStackTrace();
332         }
333     }

335     /**
336      * loads arff
337      * @param file arff file
338      */
339     public DecisionTree(File file) {
340         try {
341             this.loadArff(file);
342         } catch (Exception e) {
343             e.printStackTrace();
344         }
345     }

347     /**
348      * loads arff
349      * @param path path to arff
350      * @throws Exception see kramerlabs dataset
351      */
352     public void loadArff(String path) throws Exception {
353         File file = new File(path);
354         this.loadArff(file);
355     }

357     /**
358      * loads arff
359      * @param file arff file
360      * @throws Exception see kramerlabs dataset
361      */
```

```
362     public void loadArff(File file) throws Exception {
363         this.dataset = new Dataset();
364         dataset.load(file);
365         this.data = new Instance[dataset.getNumberOfInstances()];
366
367         for (int i = 0; i < data.length; i++) {
368             this.data[i] = dataset.getInstance(i);
369         }
370         this.classAttribute = this.dataset.getAttributes().get(this.dataset
371                 .getClassIndex());
372     }
373
374
375     // Implement a method informationGain that takes two arguments: attribute A
376     // and a list of indices i 1 , i 2 , i m .
377     /**
378      * calculates information gain for given attribute and instances
379      * @param attribute given attribute
380      * @param indices given indices of instances
381      * @return information gain
382      */
383     public double informationGain(Attribute attribute, List<Integer> indices) {
384         Attribute classAttr = classAttribute;
385
386         //TODO throw Exception
387         // Check if nominal
388         if (!attribute.isNominal()) {
389             System.err.println(attribute.getName() + "is not nominal");
390             return Double.NaN;
391         } else if (!classAttr.isNominal()) {
392             System.err.println(classAttr.getName() + "is not nominal");
393             return Double.NaN;
394         }
395
396         NominalAttribute attr = (NominalAttribute) attribute;
397         // Init gain
398         double gain = calculateEntropy((NominalAttribute)classAttr,
399                 indices);
400         // sum over all values in attribute
401         for (int v = 0; v < attr.getNumberOfValues(); v++) {
402             List<Integer> subIndices = new ArrayList<>();
403             // Alternative we could copy data in a list and remove already
404             // picked instances to improve runtime
405             // creates subset with indices of instances with value v
406             for (int i : indices) {
407                 Instance instance = data[i];
408                 NominalValue value = (NominalValue)instance.getValue(attr);
409                 if (attr.getValue(v).equals(value)) {
410                     subIndices.add(i);
411                 }
412             }
413
414             // calculates entropy
415             double entropy = 0.0;
416             if (subIndices.size() != 0) {
417                 entropy = calculateEntropy((NominalAttribute)classAttr,
418                         subIndices);
419             }
420
421             // see formula
422             gain -= entropy * ((double)subIndices.size())
```

```
423                 /((double)indices.size());
424         }

426         return gain;
427     }


430     /**
431      * calculates entropy
432      * @param classAttr given class attribute
433      * @param indices indices of instances
434      * @return entropy
435      */
436     private double calculateEntropy(NominalAttribute classAttr, List<Integer>
437             indices
438     ) {
439         int[] values = new int[classAttr.getNumberOfValues()];
440         // calculates number of instances with value v in class attribute
441         for (int v = 0; v < classAttr.getNumberOfValues(); v++) {
442             values[v] = 0;
443             NominalValue classValue = classAttr.getValue(v);

445             // counts number of value v in trainset
446             for(int i : indices) {
447                 Instance instance = data[i];
448                 NominalValue value = (NominalValue)instance.getValue(classAttr);
449                 if (classValue.equals(value)) {
450                     values[v]++;
451                 }
452             }
453         }
454         return calculateEntropy(values);
455     }

457     /**
458      * calculates entropy
459      * @param values given values
460      * @return entropy
461      */
462     private double calculateEntropy(int[] values) {
463         double sum = 0;
464         for (int i : values) {
465             sum += i;
466         }
467         double entropy = 0.0;
468         for (int value : values) {
469             double p = value/sum;
470             entropy -= p * log2(p);
471         }

473         return entropy;
474     }


477     /**
478      * calculates log to base 2
479      * @param a given parameter
480      * @return log2(a) or 0 if a == 0
481      */
482     private double log2(double a) {
483         if ( Double.compare(0.0, Math.abs(a)) == 0 )
```

```
484             return 0;
485         return Math.log(a) / Math.log(2);
486     }



490     //-------------------------------------------------------------------------
491     //-------------------------Train and Testset-------------------------------
492     //-------------------------------------------------------------------------


495     /**
496      * @return 2/3 trainset
497      */
498     public List<Integer> getTrainset() {
499         return getTrainset(2.0/3);
500     }

502     /**
503      * splits data set in train set
504      * @param split size og train set as percentage
505      * @return trainset
506      */
507     public List<Integer> getTrainset(double split) {
508         // get List with all indices
509         List<Integer> indices = new ArrayList<>();
510         for (int i = 0; i < this.data.length; i++) {
511             indices.add(i);
512         }

514         //TODO check split, throw Exception
515         // get random indices
516         int size = (int)Math.ceil(indices.size() * split);
517         while (indices.size() ≥size) {
518             Random random = new Random();
519             Integer randomIdx = random.nextInt(indices.size());
520             indices.remove(randomIdx);
521         }


524         return indices;
525     }


528     /**
529      * returns inverse data set
530      * @param originalSet
531      * @return
532      */
533     public List<Integer> getInverseSet(List<Integer> originalSet) {
534         List<Integer> inverseSet = new ArrayList<>();
535         for (int i=0; i<this.data.length; i++) {
536             if (!originalSet.contains(i))
537                 inverseSet.add(i);
538         }
539         return inverseSet;
540     }
```

```
546     //-------------------------------------------------------------------------
547     //-------------------------test--------------------------------------------
548     //-------------------------------------------------------------------------



552     /**
553      * prints some test data
554      */
555     private void testPrint() {
556         List<Integer> indices = new ArrayList<>();
557         for (int i = 0; i < data.length; i++) {
558             indices.add(i);
559         }

561         /*for (Attribute attr : this.dataset.getAttributes()) {
562             System.out.print("Attribute " + attr.getName());
563             System.out.print(" has an InformationGain of " + informationGain
564                     (attr, indices));
565             System.out.println();
566         }*/

568         List<Integer> trainset = this.getTrainset();
569         this.train(trainset);
570         System.out.println(this.classify(this.getInverseSet(trainset)));
571     }

573     private void printTree() {
574         System.out.println("Tree");
575         root.print("");
576     }

578     /**
579      * a test
580      * @param args none
581      */
582     public static void main(String[] args) {
583         DecisionTree dt = new DecisionTree("res/car.arff");
584         dt.testPrint();
585         dt.printTree();
586     }
587 }
```




(a) Abbildung 1: 1 Lerner (b) Abbildung 2: 10 Lerner (c) Abbildung 3: 500 Lerner

Plots bei verschieden vielen Lernern

## Nr.2

AdaBoost nutzt mehrere weak learner (in diesem Beispiel Bäume der Tiefe 1), um zusammen einen strong learner zu bilden. In Abbildung 1 ist zu sehen, dass die Decision Boundary bei einem learner aus einer geraden Linie besteht, welche die Instanzen in 2 Gruppen aufteilt. Im Histogramm ist zu sehen, dass dies zwar dazu führt, dass mehr als die Hälfte der Instanzen richtig klassifiziert werden, aber auch, dass die Fehlerquote sehr hoch ist. Bei 10 Lernern, wie in Abbildung 2 können mehr Instanzen richtig klassifiziert werden, weil der strong learner durch die vielen weak lerner den Merkmalsraum in mehr „Bereiche“ einteilt, welche im Decision Boundary Diagramm gut zu sehen sind. Diese Bereiche entstehen, weil die hier genutzten weak learner den Merkmalsraum in je 2 Hälften an unterschiedlichen Stellen unterteilen. Im strong learner werden nun, bildlich gesprochen, die weak learner übereinander gelegt. Dabei kommt es zu Überscheidungen von unterschiedlich „eingefärbten“ Bereichen. Diese bekommen im strong learner jene Klasse zugeteilt, welche in dem betrachteten Bereich mit dem größten Gewicht in den schwachen Lernern vorkommt. Mit ein ausreichend großen Zahl an schwachen Lernern, z.B. 500 wie in Abbildung 3, werden die Decision Boundary komplexer und die Klassifizierung noch genauer. Im Histogramm wird deutlich, dass die „Sicherheit“ der Aussagen, der Score, mit steigender Anzahl an schwachen Lernern abnimmt, jedoch der strong learner weniger Fehler macht, zum Vergleich, der „strong learner“ aus einem weak learner klassifizierte etwas über 200 von 500 Instanzen falsch (laut Histogramm aus Abbildung 1), während der strong learner aus Abbildung 3 nur noch etwa 70 Instanzen falsch zuordnet.

## Nr.3

0.632 Bootstrap meint die Bootstrap-Evaluierung mit einem Testset der Größe n, wobei n die Größe des genutzten Datensatzes ist. Für das Bootstrapverfahren werden zufällig gleichverteilt Instanzen aus dem Datensatz dem Trainingssatz hinzugefügt, bis dieser voll ist. Dabei wird in jeder Iteration der komplette Datensatz betrachtet, inklusive der bereits gewählten Instanzen. Die nicht gewählten Instanzen bilden den Testsatz. Die Wahrscheinlichkeit, dass eine Instanz in einer Iteration nicht gewählt wird beträgt

$$1 - \frac{1}{n}$$

Die prozentuale Größe des Testsatzes beträgt somit

$$(1 - \frac{1}{n})^m$$

wobei $n$ die Größe des Datensatzes und $m$ die Größe des Trainingssatzes ist. Für $n = m$ gilt:

$$1 - (1 - \frac{1}{n})^n \approx 1 - \frac{1}{e} \approx 0,632$$

Für einen Testsatz der Größe $2n$ folgt somit:

$$1 - (1 - \frac{1}{n})^2 n = 1 - \left((1 - \frac{1}{n})^n\right)^2 \approx 1 - \frac{1}{e^2} \approx 0,865$$