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      *
119      */
120     public void train(List<Integer> trainset) {
121         // create root Node
122         this.root = new Node(trainset, null);
123         this.train_recursive(this.root);
124     }

126     /**
127      * Internal method
128      * @param n
129      */
130     public void train_recursive(Node n) {

132         // exit function
133         if (this.isSingleNode(n)) {
134             return;
135         }
136         //select attribute with biggest informationGain
137         n.setAttribute(this.selectAttribute(n));

139         // create edges for each value of the attribute
140         NominalAttribute attr = (NominalAttribute)n.attribute;
141         for (int i = 0; i < attr.getNumberOfValues(); i++) {
142             Value v = attr.getValue(i);
143             Edge edge = new Edge(v, n);
144             edge.end = new Node(new ArrayList<>(), n);
145         }

147         for (Integer idx : n.indices) {
148             Instance i = this.data[idx];
149             Value v = i.getValue(n.attribute);

151             // add index to right edge
152             for (Edge edge : n.edges) {
153                 if (edge.value.equals(v)) {
154                     edge.end.indices.add(idx);
155                     break;
156                 }
157             }
158         }
159         // create tree
160         for (Edge e : n.edges) {
161             this.train_recursive(e.end);
162         }
163     }

165     //-------------------------------------------------------------------------
166     //-------------------------classify tree-----------------------------------
167     //-------------------------------------------------------------------------

169     /**
170      * classifies given data set on decision Tree
171      * @param data
172      * @return
173      */
174     public double classify(List<Integer> data) {
175         //TODO test if tree is build
176         int correctlyClassified = 0;
177         // repeat for each instance
178         for (Integer i : data) {
```

```
179             Instance instance = this.data[i];

181             // iterate over tre
182             Node currentNode = this.root;
183             while (!currentNode.isSingleNode) {
184                 Attribute attr = currentNode.attribute;
185                 Value value = instance.getValue(attr);

187                 // find right edge
188                 for (Edge edge : currentNode.edges) {
189                     if (edge.value.equals(value)) {
190                         currentNode = edge.end;
191                         break;
192                     }
193                 }
194             }
195             // check if class attr is correct
196             if ( currentNode.getValue().equals(instance.getValue(classAttribute)) ) {
197                 correctlyClassified ++;
198             }
199         }
200         return (double)correctlyClassified/(double)data.size();
201     }
```

⋮

```
258     //------------------------------------------------------------------------
259     //----------------------------Helper--------------------------------------
260     //------------------------------------------------------------------------

262     /**
263      * chooses the attribute with the best information gain
264      * @param node
265      * @return
266      */
267     public Attribute selectAttribute(Node node) {
268         Attribute select = null;
269         double maxGain = Double.NEGATIVE_INFINITY;

271         // looks at all relevant attributes
272         for (Attribute attribute : node.notVisited) {

274             // does not look at classAttribute
275             /*if (attribute.equals(classAttribute)) {
276                 continue;
277             }*/
278             double gain = this.informationGain(attribute, node.indices);
279             if (gain > maxGain) {
280                 select = attribute;
281                 maxGain = gain;
282             }
283         }
284         return select;
285     }

287     /**
288      * Tests if node has only one class left and sets node.isSingleNode
289      * @param node
290      * @return
291      */
292     private boolean isSingleNode(Node node) {
```

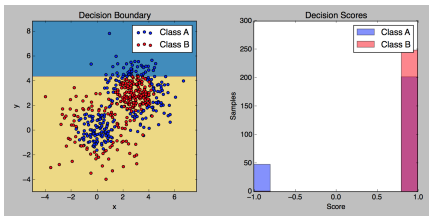
```
294         // if node is empty we set his value to the most common
295         // value of his parent
296         if (node.indices.size() == 0) {
297             node.value = mostCommonValue(node.parent, this.classAttribute);
298             node.isSingleNode = true;
299             return true;
300         }
301         // should not be possible anymore
302         // if last node we set value to most common value
303         /*if ( node.notVisited.size() == 1
304                 && node.notVisited.contains(classAttribute)) {
305             node.value = mostCommonValue(node, this.classAttribute);
306             node.isSingleNode = true;
307             return true;
308         }*/
309         /*// should not be possible, else like the one over this*/
310         // if last node we set value to most common value
311         if ( node.notVisited.size() == 0 ) {
312             node.value = mostCommonValue(node, this.classAttribute);
313             node.isSingleNode = true;
314             return true;
315         }

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

319         // if one instance has an other value as the rest return false
320         for (int i = 1; i < node.indices.size(); i++) {
321             Instance instance = data[node.indices.get(i)];
322             if (! instance.getValue(classAttribute).equals(value)) {
323                 return false;
324             }
325         }

327         // else set as single node and return true
328         node.isSingleNode = true;
329         node.value = value;
330         return true;
331     }

333     /**
334      * returns most common value from node in attribute
335      * @param node
336      * @param attribute
337      * @return
338      */
339     private Value mostCommonValue(Node node, Attribute attribute) {
340         // because attribute dose not know his values
341         Map<Value, Integer> numValues = new HashMap<Value, Integer>();
342         // temp
343         Value max = null;
344         int maxInt = -1;

346         for (Integer i : node.indices) {
347             Instance instance = data[i];
348             Value value = instance.getValue(attribute);
349             Integer integer = 1;
350             if(numValues.containsKey(value)) {
351                 integer = numValues.get(value);
352                 integer++;
353                 numValues.replace(value, integer);
```

```
354             } else {
355                 numValues.put(value, integer);
356             }
357             if (integer > maxInt) {
358                 max = value;
359             }
360         }
361         return max;
362     }
```

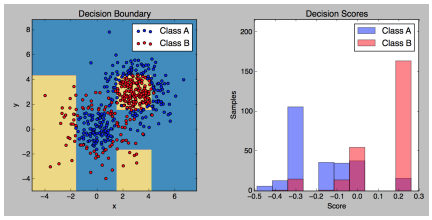
⋮

```
474     //------------------------------------------------------------------------
475     //-------------------------Train and Testset------------------------------
476     //------------------------------------------------------------------------

478     /**
479      * @return 2/3 trainset
480      */
481     public List<Integer> getTrainset() {
482         return getTrainset(2.0/3);
483     }

485     /**
486      * splits data set in train set
487      * @param split size og train set as percentage
488      * @return trainset
489      */
490     public List<Integer> getTrainset(double split) {
491         // get List with all indices
492         List<Integer> indices = new ArrayList<>();
493         for (int i = 0; i < this.data.length; i++) {
494             indices.add(i);
495         }

497         //TODO check split, throw Exception
498         // get random indices
499         int size = (int)Math.ceil(indices.size() * split);
500         while (indices.size() ≥size) {
501             Random random = new Random();
502             Integer randomIdx = random.nextInt(indices.size());
503             indices.remove(randomIdx);
504         }
505         return indices;
506     }


509     /**
510      * returns inverse data set
511      * @param originalSet
512      * @return
513      */
514     public List<Integer> getInverseSet(List<Integer> originalSet) {
515         List<Integer> inverseSet = new ArrayList<>();
516         for (int i=0; i<this.data.length; i++) {
517             if (!originalSet.contains(i))
518                 inverseSet.add(i);
519         }
520         return inverseSet;
521     }
```

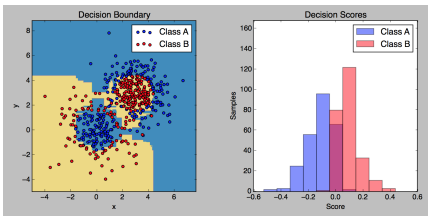
```
523     //-----------------------------------------------------------------------
524     //-------------------------test------------------------------------------
525     //-----------------------------------------------------------------------

527     /**
528      * prints some test data
529      */
530     private void testPrint() {
531         List<Integer> indices = new ArrayList<>();
532         for (int i = 0; i < data.length; i++) {
533             indices.add(i);
534         }
535         List<Integer> trainset = this.getTrainset();
536         this.train(trainset);
537         System.out.println(this.classify(this.getInverseSet(trainset)));
538     }

540     private void printTree() {
541         System.out.println("Tree");
542         root.print("");
543     }

545     /**
546      * a test
547      * @param args none
548      */
549     public static void main(String[] args) {
550         DecisionTree dt = new DecisionTree("res/car.arff");
551         dt.testPrint();
552         dt.printTree();
553     }
554 }
```



(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:

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