

# 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             } else {
34                 notVisited = parent.notVisited;
35             }
36         }

38         public void addEdge(Edge edge) {
39             this.edges.add(edge);
40         }

42         public Value getValue() {
43             if (isSingleNode)
44                 return value;
45             return null;
46         }

48         public void print(String prefix) {
49             if (isSingleNode) {
50                 System.out.println(prefix + " " + value);
51             } else {
52                 System.out.println(prefix + " " + attribute);
53             }
54             for(Edge edge : edges) {
55                 edge.end.print(prefix+'-');
56             }

```

```

57     }

59     public void setAttribute(Attribute attribute) {
60         this.attribute = attribute;
61         this.notVisited.remove(attribute);
62     }
63 }

66 /**
67  * Edge class
68  */
69 private class Edge {
70     Value value = null;
71     Node start;
72     Node end;

74     /**
75      * Constructor
76      * @param value
77      */
78     public Edge(Value value, Node start) {
79         this.value = value;
80         this.start = start;
81         this.start.addEdge(this);
82     }
83 }

88 private Node root = null;

90 private Instance[] data;
91 private Dataset dataset;
92 private Attribute classAttribute;

94 /**
95  * default constructor
96  */
97 public DecisionTree() {

99 }

104 //-----
105 //-----Train tree-----
106 //-----

108 /**
109  *
110  * @param node
111  * @return
112  */
113 public Attribute selectAttribute(Node node) {
114     Attribute select = null;
115     double maxGain = Double.NEGATIVE_INFINITY;
116     for (Attribute attribute : node.notVisited) {
117         if (attribute.equals(classAttribute)) {

```

```

118         continue;
119     }
120     double gain = this.informationGain(attribute, node.indices);
121     if (gain > maxGain) {
122         select = attribute;
123         maxGain = gain;
124     }
125 }
126 return select;
127 }

130 /**
131  *
132  */
133 public void train(List<Integer> trainset) {
134     // create root Node
135     this.root = new Node(trainset, null);
136     this.train_recursive(this.root);
137 }

139 /**
140  * Internal method
141  * @param n
142  */
143 public void train_recursive(Node n) {
144     // todo remove attributes to prevet duplicates

146     // exit function
147     if (this.isSingleNode(n)) {
148         return;
149     }
150     //select attribute with biggest informationGain
151     n.setAttribute(this.selectAttribute(n));

153     // create edges for each value of the attribute
154     NominalAttribute attr = (NominalAttribute)n.attribute;
155     for (int i = 0; i < attr.getNumberOfValues(); i++) {
156         Value v = attr.getValue(i);
157         Edge edge = new Edge(v, n);
158         edge.end = new Node(new ArrayList<>(), n);
159     }

163     for (Integer idx : n.indices) {
164         Instance i = this.data[idx];
165         Value v = i.getValue(n.attribute);

169         // add index to right edge
170         for (Edge edge : n.edges) {
171             if (edge.value.equals(v)) {
172                 edge.end.indices.add(idx);
173                 break;
174             }
175         }

177     }

```

```

179     // create tree
180     for (Edge e : n.edges) {
181         this.train_recursive(e.end);
182     }
183 }

185 /**
186  *
187  * @param node
188  * @return
189  */
190 private boolean isSingleNode(Node node) {

192     if (node.indices.size() == 0 || node.notVisited.size() == 0 || (node
193         .notVisited.size() == 1 && node.notVisited.contains(classAttribute))) {
194         node.value = mostCommonValue(node.parent, this.classAttribute);
195         node.isSingleNode = true;
196         return true;
197     }

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

201     for (int i = 1; i < node.indices.size(); i++) {
202         Instance instance = data[node.indices.get(i)];
203         if (! instance.getValue(classAttribute).equals(value)) {
204             return false;
205         }
206     }

208     node.isSingleNode = true;
209     node.value = value;
210     return true;
211 }

214 //-----
215 //-----classify tree-----
216 //-----

219 public double classify(List<Integer> data) {
220     int correctlyClassified = 0;
221     // repeat for each instance
222     for (Integer i : data) {
223         Instance instance = this.data[i];

225         // iterate over tree
226         Node currentNode = this.root;
227         while (!currentNode.isSingleNode) {
228             Attribute attr = currentNode.attribute;
229             Value value = instance.getValue(attr);

231             // find right edge
232             for (Edge edge : currentNode.edges) {
233                 if (edge.value.equals(value)) {
234                     currentNode = edge.end;
235                     break;
236                 }
237             }
238         }

```

```

240         // check if class attr is correct
241         if ( currentNode.getValue().equals(instance.getValue(classAttribute)) ) {
242             correctlyClassified ++;
243         }
244     }
245
246     return (double)correctlyClassified/(double)data.size();
247 }
248
249
250
251
252
253 //-----
254 //-----Constructor-----
255 //-----
256
257 private Value mostCommonValue(Node node, Attribute attribute) {
258     Map<Value, Integer> numValues = new HashMap<Value, Integer>();
259     Value max = null;
260     int maxInt = -1;
261
262     for (Integer i : node.indices) {
263         Instance instance = data[i];
264         Value value = instance.getValue(attribute);
265         Integer integer = 1;
266         if(numValues.containsKey(value)) {
267             integer = numValues.get(value);
268             integer++;
269             numValues.replace(value, integer);
270         } else {
271             numValues.put(value, integer);
272         }
273
274         if (integer > maxInt) {
275             max = value;
276         }
277     }
278
279     return max;
280 }
281
282 /**
283  * loads arff
284  * @param path path to arff
285  */
286 public DecisionTree(String path) {
287     try {
288         this.loadArff(path);
289     } catch (Exception e) {
290         e.printStackTrace();
291     }
292 }
293
294 /**
295  * loads arff
296  * @param file arff file
297  */
298 public DecisionTree(File file) {
299     try {
300         this.loadArff(file);

```

```

301     } catch (Exception e) {
302         e.printStackTrace();
303     }
304 }

306 /**
307  * loads arff
308  * @param path path to arff
309  * @throws Exception see kramerlabs dataset
310  */
311 public void loadArff(String path) throws Exception {
312     File file = new File(path);
313     this.loadArff(file);
314 }

316 /**
317  * loads arff
318  * @param file arff file
319  * @throws Exception see kramerlabs dataset
320  */
321 public void loadArff(File file) throws Exception {
322     this.dataset = new Dataset();
323     dataset.load(file);
324     this.data = new Instance[dataset.getNumberOfInstances()];

326     for (int i = 0; i < data.length; i++) {
327         this.data[i] = dataset.getInstance(i);
328     }
329     this.classAttribute = this.dataset.getAttributes().get(this.dataset
330         .getClassIndex());
331 }

334 // Implement a method informationGain that takes two arguments: attribute A
335 // and a list of indices i 1 , i 2 , i m .
336 /**
337  * calculates information gain for given attribute and instances
338  * @param attribute given attribute
339  * @param indices given indices of instances
340  * @return information gain
341  */
342 public double informationGain(Attribute attribute, List<Integer> indices) {
343     Attribute classAttr = classAttribute;

345     //TODO throw Exception
346     // Check if nominal
347     if (!attribute.isNominal()) {
348         System.err.println(attribute.getName() + "is not nominal");
349         return Double.NaN;
350     } else if (!classAttr.isNominal()) {
351         System.err.println(classAttr.getName() + "is not nominal");
352         return Double.NaN;
353     }

355     NominalAttribute attr = (NominalAttribute) attribute;
356     // Init gain
357     double gain = calculateEntropy((NominalAttribute)classAttr,
358         indices);
359     // sum over all values in attribute
360     for (int v = 0; v < attr.getNumberOfValues(); v++) {
361         List<Integer> subIndices = new ArrayList<>();

```



```

362         // Alternative we could copy data in a list and remove already
363         // picked instances to improve runtime
364         // creates subset with indices of instances with value v
365         for (int i : indices) {
366             Instance instance = data[i];
367             NominalValue value = (NominalValue)instance.getValue(attr);
368             if (attr.getValue(v).equals(value)) {
369                 subIndices.add(i);
370             }
371         }

373         // calculates entropy
374         double entropy = 0.0;
375         if (subIndices.size() != 0) {
376             entropy = calculateEntropy((NominalAttribute)classAttr,
377                                     subIndices);
378         }

380         // see formula
381         gain -= entropy * ((double)subIndices.size())
382                /((double)indices.size());
383     }

385     return gain;
386 }

389 /**
390  * calculates entropy
391  * @param classAttr given class attribute
392  * @param indices indices of instances
393  * @return entropy
394  */
395 private double calculateEntropy(NominalAttribute classAttr, List<Integer>
396                               indices
397 ) {
398     int[] values = new int[classAttr.getNumberOfValues()];
399     // calculates number of instances with value v in class attribute
400     for (int v = 0; v < classAttr.getNumberOfValues(); v++) {
401         values[v] = 0;
402         for(int i : indices) {
403             Instance instance = data[i];
404             NominalValue value = (NominalValue)instance.getValue(classAttr);
405             NominalValue classValue = classAttr.getValue(v);
406             if (classValue.equals(value)) {
407                 values[v]++;
408             }
409         }
410     }
411     return calculateEntropy(values);
412 }

414 /**
415  * calculates entropy
416  * @param values given values
417  * @return entropy
418  */
419 private double calculateEntropy(int[] values) {
420     double sum = 0;
421     for (int i : values) {
422         sum += i;

```

```

423     }
424     double entropy = 0.0;
425     for (int value : values) {
426         double p = value/sum;
427         entropy -= p * log2(p);
428     }

430     return entropy;
431 }

434 /**
435  * calculates log to base 2
436  * @param a given parameter
437  * @return log2(a) or 0 if a == 0
438  */
439 private double log2(double a) {
440     if ( Double.compare(0.0, Math.abs(a)) == 0 )
441         return 0;
442     return Math.log(a) / Math.log(2);
443 }

447 //-----
448 //-----Train and Testset-----
449 //-----

452 /**
453  * @return 2/3 trainset
454  */
455 public List<Integer> getTrainset() {
456     // get List with all indices
457     List<Integer> indices = new ArrayList<>();
458     for (int i = 0; i < this.data.length; i++) {
459         indices.add(i);
460     }

462     // get random indices
463     int size = indices.size() * 2/3;
464     while (indices.size() >= size) {
465         Random random = new Random();
466         Integer randomIdx = random.nextInt(indices.size());
467         indices.remove(randomIdx);
468     }

471     return indices;
472 }

475 public List<Integer> getInverseSet(List<Integer> originalSet) {
476     List<Integer> inverseSet = new ArrayList<>();
477     for (int i=0; i<this.data.length; i++) {
478         if (!originalSet.contains(i))
479             inverseSet.add(i);
480     }
481     return inverseSet;
482 }

```

```

488 //-----
489 //-----test-----
490 //-----

494 /**
495  * prints some test data
496  */
497 private void testPrint() {
498     List<Integer> indices = new ArrayList<>();
499     for (int i = 0; i < data.length; i++) {
500         indices.add(i);
501     }

503     /*for (Attribute attr : this.dataset.getAttributes()) {
504         System.out.print("Attribute " + attr.getName());
505         System.out.print(" has an InformationGain of " + informationGain
506             (attr, indices));
507         System.out.println();
508     }*/

510     List<Integer> trainset = this.getTrainset();
511     this.train(trainset);
512     System.out.println(this.classify(this.getInverseSet(trainset)));
513 }

515 private void printTree() {
516     root.print("");
517 }

519 /**
520  * a test
521  * @param args none
522  */
523 public static void main(String[] args) {
524     DecisionTree dt = new DecisionTree("res/car.arff");
525     dt.testPrint();
526     dt.printTree();
527 }
528 }

```

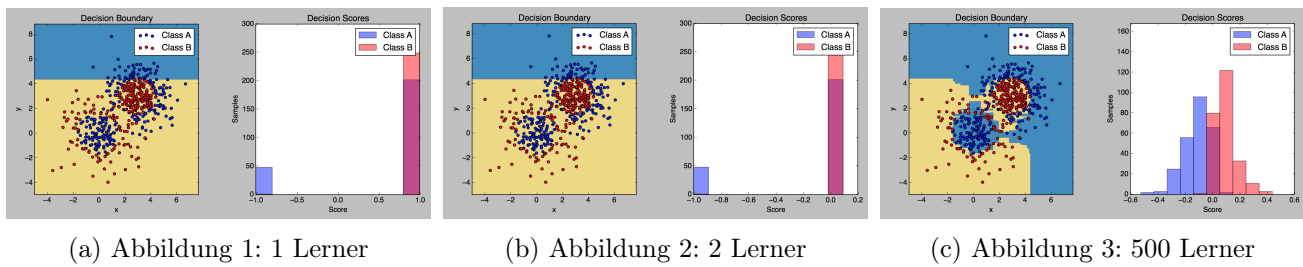


Abbildung 1: Plots bei verschieden vielen Lernern

## Nr.2

### Anmerkung

### Aufgabe

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 Borders 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 2 Lernern, wie in Abbildung 2 können mehr Instanzen richtig klassifiziert werden. Mit hunderten, wie in Abbildung 3, werden die Decision Borders komplexer und die Klassifizierung noch genauer. Im Histogramm wird deutlich, dass die „Sicherheit“ (Score) der Aussagen mit zunehmender Anzahl an Lernern, abnimmt.

## 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 dieses 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 somit beträgt

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

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

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

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

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