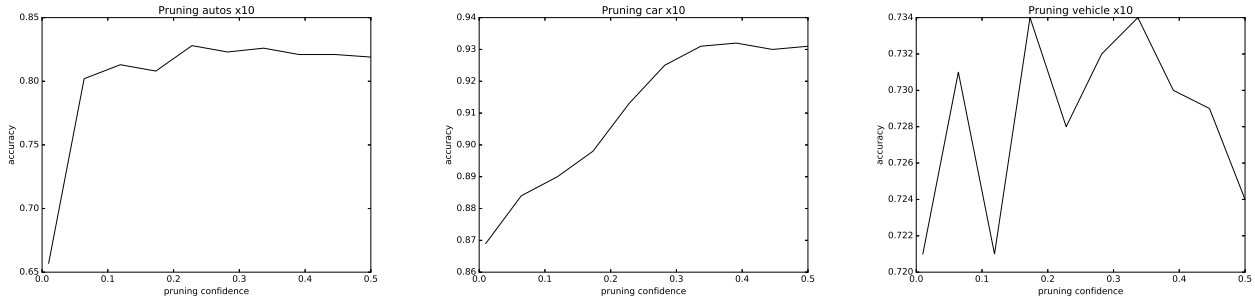
Machine Learning
Blatt 4

Markus Vieth, David Klopp, Christian Stricker

2. Juni 2016

Nr.1

b)



autos dataset car dataset vehicle dataset

Abbildung 1: Werte wurden über 10 Durchläufe gemittelt

c)

```
1  import com.sun.org.apache.xpath.internal.functions.WrongNumberArgsException;
2  import org.omg.CORBA.DynAnyPackage.InvalidValue;
3  import weka.classifiers.Classifier;
4  import weka.classifiers.Evaluation;
5  import weka.core.*;

7  import java.io.Serializable;
8  import java.util.ArrayList;
9  import java.util.Random;

11 /**
12  * Created by David on 24.05.16.
13  */

15 // extend RandomizableSingleClassifierEnhancer to use the method setClassifier

17 public class CVParameterSelection extends Classifier implements Serializable,
18         Cloneable {


21     //------------------------------------------------------------------------
22     //----------------------Internal CVParameter Class------------------------
23     //------------------------------------------------------------------------

25     // internal class that represents one parameter
26     protected class CVParameter implements Serializable{

28         protected char paramChar;
29         protected double lowerBound;
30         protected double upperBound;
31         protected double stepValue;
32         // current value to test
33         protected double value;


36         /**
```

```
37          *
38          * @param paramChar
39          * @param lowerBound
40          * @param upperBound
41          * @param steps
42          * @throws Exception
43          */
44         public CVParameter(char paramChar, double lowerBound, double upperBound, double steps)
                throws Exception {
45             this.paramChar = paramChar;
46             this.lowerBound = lowerBound;
47             this.upperBound = upperBound;
48             this.stepValue = steps;

50             // check if values are valid
51             if (this.lowerBound > this.upperBound) {
52                 throw new InvalidValue("Lower bound must be lesser than or equal to upper bound");
53             }
54         }

56         /**
57          * Input string given by the user
58          * @return name of parameter
59          */
60         protected String getParameterString() {
61             String p = String.valueOf(this.paramChar);
62             String u = String.valueOf(this.upperBound);
63             String l = String.valueOf(this.lowerBound);
64             String s = String.valueOf(this.stepValue);

66             return (p + " " + l + " " + u + " " + s);
67         }
68     }



72     //------------------------------------------------------------------------
73     //-------------------------Attributes-------------------------------------
74     //------------------------------------------------------------------------


77     private final int DEFAULT_NUM_OF_FOLDS = 10;

79     // available options
80     protected String[] classifierOptions = null;
81     // parameters which are set
82     private ArrayList<CVParameter> params = new ArrayList<>();
83     // use 10 number of folds as default
84     private int numFolds = DEFAULT_NUM_OF_FOLDS;
85     // classifier for selection
86     private Classifier classifier;
87     // lowest error rate
88     private double lowestError = Double.MAX_VALUE;
89     // best options for this classifier
90     private String[] bestOptions = null;
91     // initial classifierOptions
92     private String[] initClassifierOptions = null;


95     //------------------------------------------------------------------------
96     //-------------------------Setter-----------------------------------------
```

```
97      //-----------------------------------------------------------------------

99      /**
100      * @param classifier
101      */
102     public void setClassifier(Classifier classifier) {
103         this.classifier = classifier;
104         this.initClassifierOptions = this.getClassifier().getOptions();
105     }


108     /**
109      * add a single parameter
110      * @param param name of the parameter
111      * @return true on success otherwise false
112      */
113     public void addCVParameter(String param) throws Exception {
114         // we expect a string in format: 'N lower_bound upper_bound steps'

116         // split parameter in its arguments
117         // we need exactly 4 values
118         char paramChar;
119         double lowerBound, upperBound, steps;
120         String[] args = param.split(" ");

122         // check input
123         if (args.length == 4) {
124             // get parameter character
125             String paramString = args[0];
126             if (paramString.length() != 1) {
127                 throw new IllegalArgumentException("Invalid parameter char.");
128             } else {
129                 paramChar = paramString.charAt(0);
130             }

132             // save lower bound, upper bound and steps
133             try {
134                 lowerBound = Double.parseDouble(args[1]);
135                 upperBound = Double.parseDouble(args[2]);
136                 steps = Double.parseDouble(args[3]);
137             } catch (NumberFormatException e) {
138                 throw new Exception("Invalid values for parameter " + paramString);
139             }

141             // add CVParameter
142             CVParameter tmp = new CVParameter(paramChar, lowerBound, upperBound, steps);
143             this.params.add(tmp);


146         } else {
147             throw new WrongNumberArgsException("At least 4 values are required.");
148         }
149     }


152     /**
153      *
154      * @param params parameters to add
155      */
156     public void setCVParameters(String[] params) throws Exception {
157         for (String param : params) {
```

```
158             this.addCVParameter(param);
159         }
160     }

162     /**
163      * @param numFolds number of folds for crossvalidation
164      */
165     public void setNumberOfFolds(int numFolds) {
166         if (numFolds < 0) {
167             throw new IllegalArgumentException("Number of folds must be positiv");
168         } else {
169             this.numFolds = numFolds;
170         }
171     }



175     //-------------------------------------------------------------------------
176     //-----------------------------Getter--------------------------------------
177     //-------------------------------------------------------------------------


180     /**
181      *
182      * @return classifier
183      */
184     public Classifier getClassifier() {
185         return this.classifier;
186     }

188     /**
189      *
190      * @param index of parameter
191      * @return name of parameter
192      */
193     public String getCVParameter(int index) throws IndexOutOfBoundsException{
194         if (index > 0 && index < this.params.size()) {
195             return this.params.get(index).getParameterString();
196         } else {
197             throw new IndexOutOfBoundsException("Parameter index out of bounds");
198         }
199     }

201     /**
202      *
203      * @return names of all parameters
204      */
205     public String[] getCVParameters() {
206         String[] paramNames = new String[this.params.size()];
207         for (int i = 0; i < paramNames.length; i++) {
208             paramNames[i] = this.getCVParameter(i);
209         }
210         return paramNames;
211     }

213     /**
214      *
215      * @return number of folds
216      */
217     public int getNumberOfFolds() {
218         return this.numFolds;
```

```
219     }


222     public String[] getBestClassifierOptions() {
223         // return bestOptionsArray by deleting empty entries
224         int size = 0;
225         for (String s : this.bestOptions) {
226             if (s.equals("") == false) {
227                 size++;
228             }
229         }

231         // add values
232         String[] cleanBestOptions = new String[size];
233         int i = 0;
234         for (String s : this.bestOptions) {
235             if (s.equals("") == false) {
236                 cleanBestOptions[i] = s;
237                 i++;
238             }
239         }

241         return cleanBestOptions;
242     }


245     //------------------------------------------------------------------------
246     //----------------------------classifier Operations-------------------------
247     //------------------------------------------------------------------------


250     /**
251      * Combine cross validation options of this class with the classifier options
252      */
253     private String[] getCombinedOptions() throws Exception {
254         // remove all options from our classifier which are already set by this class
255         // wekas Util.getOption method does this for us
256         // Note: this changes the classifierOptions array
257         String[] classifierOptions = this.getClassifier().getOptions();
258         for (CVParameter param : this.params) {
259             Utils.getOption(param.paramChar, classifierOptions);
260         }

262         // create new array
263         int size = classifierOptions.length + 2*this.params.size();
264         String[] options = new String[size];

266         // add crossvalidation options of this class
267         int i = 0;
268         for (CVParameter param : this.params) {
269             options[i++] = "-"+String.valueOf(param.paramChar);
270             options[i++] = String.valueOf(param.value);
271         }

273         // add classifier options
274         for (String opt : classifierOptions) {
275             if (opt.equals("") == false) {
276                 options[i++] = opt;
277             }
278         }
```

```
280         // fill the rest
281         while (i < size) {
282             options[i++] = "";
283         }
284
285         return options;
286     }
287
288     /**
289      *
290      * @param paramIdx
291      * @param dataset
292      * @throws Exception
293      */
294     private void calculateBestValuesForOptions(int paramIdx, Instances dataset) throws Exception {
295         if (paramIdx < this.params.size()) {
296             // repeat for each parameter
297             CVParameter param = this.params.get(paramIdx);
298
299             // calculate our increment value
300             double u, l, s;
301             u = param.upperBound;
302             l = param.lowerBound;
303             s = param.stepValue;
304
305             double inc = (u-l)/(s-1);
306             for (param.value = l; param.value ≤u; param.value+=inc) {
307                 // calculate param.value => best options
308                 calculateBestValuesForOptions(paramIdx+1, dataset);
309             }
310         } else {
311             Evaluation eval = new Evaluation(dataset);
312
313             // get combined options
314             String[] options = this.getCombinedOptions();
315             // set the option for our classifier
316             // note this deletes our option array
317             this.classifier.setOptions(options);
318
319             int nFolds = this.getNumberOfFolds();
320             for (int i = 0; i<nFolds; i++) {
321                 // get a trainset and a testset
322                 // randomize the data the same way each time
323                 // i-te fold
324                 Instances train = dataset.trainCV(nFolds, i, new Random(1));
325                 // is not important to randomize the same way with our test set
326                 Instances test = dataset.testCV(nFolds, i);
327                 // build our classifier
328                 this.getClassifier().buildClassifier(train);
329                 // reset Prior probability of the evaluation
330                 eval.setPriors(train);
331                 // evaluate our model for each fold
332                 eval.evaluateModel(this.getClassifier(), test);
333             }
334
335             // get our error rate and save the options if the error rate is
336             // better than the last one
337             double errorRate = eval.errorRate();
338             /*System.out.print(errorRate);
339             System.out.print(" < ");
340             System.out.println(this.lowestError);*/
```

```
341             if (errorRate < this.lowestError) {
342                 this.lowestError = errorRate;
343                 this.bestOptions = this.getCombinedOptions();
344             }
345         }
346     }



351     //--------------------------------------------------------------------------
352     //---------------------------------Overrides--------------------------------
353     //--------------------------------------------------------------------------



357     /**
358      * Build classifier with bestOptions
359      * @param instances
360      * @throws Exception
361      */
362     @Override
363     public void buildClassifier(Instances instances) throws Exception {
364         Instances trainData = new Instances(instances);

366         // shuffle our trainData
367         trainData.randomize(new Random());

369         // if the user has not set any options then just build the classifier
370         if (this.params.isEmpty()) {
371             this.classifier.buildClassifier(trainData);
372             // set default params as best params
373             this.bestOptions = this.initClassifierOptions;
374         } else {
375             // calculate bestOptions
376             this.calculateBestValuesForOptions(0, trainData);

378             // set bestOptions for our classifier and build it
379             // make a copy of our options, because training the classifier deletes the entries
380             String[] opt = new String[this.bestOptions.length];
381             System.arraycopy(this.bestOptions, 0, opt, 0, opt.length);

383             this.getClassifier().setOptions(opt);
384             this.getClassifier().buildClassifier(trainData);
385         }

387     }



390     /**
391      *
392      * @return
393      */
394     @Override
395     public Capabilities getCapabilities() {
396         Capabilities cap = super.getCapabilities();
397         cap.setMinimumNumberInstances(this.getNumberOfFolds());
398         return cap;
399     }
```

```
402     /**
403      *
404      * @param instance
405      * @return
406      * @throws Exception
407      */
408     @Override
409     public double[] distributionForInstance(Instance instance) throws Exception {
410         return this.getClassifier().distributionForInstance(instance);
411     }
412 }
```

d)

```
1  import weka.core.*;
2  import java.util.Random;
3  import weka.classifiers.Classifier;
4  import weka.classifiers.trees.J48;
5  import weka.core.Capabilities;

7  /**
8   * Created by David on 24.05.16.
9   */
10 public class OptimalJ48 extends Classifier {
11     protected CVParameterSelection selection;
12     private String[] OPTIONS = {"C 0.1 0.5 10"};

14     public OptimalJ48() throws Exception {
15         J48 temp = new J48();
16         this.selection = new CVParameterSelection();
17         selection.setCVParameters(OPTIONS);
18         //selection.setNumberOfFolds(15);
19         selection.setClassifier(temp);
20     }

22     /**
23      * Generates a classifier. Must initialize all fields of the classifier that
24      * are not being set via options (ie. multiple calls of buildClassifier must
25      * always lead to the same result). Must not change the dataset in any way.
26      *
27      * @param data
28      *        set of instances serving as training data
29      *
30      * @throws Exception
31      *        if the classifier has not been generated successfully
32      */
33     @Override
34     public void buildClassifier(Instances data) throws Exception {
35         this.selection.buildClassifier(data);
36     }

38     /**
39      * Classifies the given test instance. The instance has to belong to a
40      * dataset when it's being classified. Note that a classifier MUST implement
41      * either this or distributionForInstance().
42      *
43      * @param instance
44      *        the instance to be classified
45      *
46      * @return the predicted most likely class for the instance or
47      * Utils.missingValue() if no prediction is made
```

```
48       *
49       * @throws Exception
50       *         if an error occurred during the prediction
51       */
52      @Override
53      public double classifyInstance(Instance instance) throws Exception {
54          return this.selection.classifyInstance(instance);
55      }

57      /**
58       * Predicts the class memberships for a given instance. If an instance is
59       * unclassified, the returned array elements must be all zero. If the class
60       * is numeric, the array must consist of only one element, which contains
61       * the predicted value. Note that a classifier MUST implement either this or
62       * classifyInstance().
63       *
64       * @param instance
65       *         the instance to be classified
66       *
67       * @return an array containing the estimated membership probabilities of the
68       * test instance in each class or the numeric prediction
69       *
70       * @throws Exception
71       *         if distribution could not be computed successfully
72       */

74      @Override
75      public double[] distributionForInstance(Instance instance) throws Exception {
76          return this.selection.distributionForInstance(instance);
77      }

79      /**
80       * Returns the Capabilities of this classifier. Maximally permissive
81       * capabilities are allowed by default. Derived classifiers should override
82       * this method and first disable all capabilities and then enable just those
83       * capabilities that make sense for the scheme.
84       *
85       * @return the capabilities of this object
86       *
87       * @see Capabilities
88       */
89      @Override
90      public Capabilities getCapabilities() {
91          return this.selection.getCapabilities();
92      }


95      // e)
```

e)

```
1       /**
2        * User pieces=3 for trainset, validationset and testset
3        * @param pieces number of pieces to split the data
4        * @return Array containing number of pieces Instances
5        */
6       public Instances[] splitDataset(int pieces, Instances data) {
7           // size of dataset
8           int size = data.numInstances();
9           int piece_size = sizeПieces;
```

```
        // randomize the data
        data.randomize(new Random());

        // save trainings, validation and test set
        Instances[] datasets = new Instances[pieces];

        // split the data
        for (int i = 0; i<pieces; i++) {
            int lowerBound = (int)(Math.ceil(piece_size*i));
            int upperBound = (int)(Math.ceil(piece_size*(i+1)));
            datasets[i] = new Instances(data, lowerBound, upperBound);
        }

        return datasets;
    }
}
```

f)

```
import weka.classifiers.Evaluation;
import weka.classifiers.trees.J48;
import weka.core.Instances;
import weka.core.converters.ConverterUtils;

import java.io.IOException;
import java.text.NumberFormat;
import java.util.*;
import java.util.stream.DoubleStream;

/**
 * Created by markus on 24.05.16.
 */
public class Test {
    public static void main(String... args) throws Exception {

        Instances car = load("res/car.arff");
        Instances autos = load("res/autos.arff");
        Instances vehicle = load("res/vehicle.arff");

        testPruning(car, 10, 0.01f, 0.5f, 10, 10, "Pruning car x10");
        testPruning(autos, 10, 0.01f, 0.5f, 10, 10, "Pruning autos x10");
        testPruning(vehicle, 10, 0.01f, 0.5f, 10, 10, "Pruning vehicle x10");

    }

    /**
     * tests pruning settings of J48 and saves graph as eps
     * @param data dataset
     * @param numFolds number of folds
     * @param lowerBorder start value for pruning
     * @param upperBorder end value
     * @param steps number of steps
     * @param repeats number of repeats fpr average
     * @param title title for graph
     * @throws Exception
     */
    public static void testPruning(Instances data, int numFolds, float
            lowerBorder, float upperBorder, int steps, int repeats,
            String title) throws Exception {
        double[][] accuracy = new double[steps][repeats];
        double[] x = new double[steps];
```

```
17          float dx = (upperBorder - lowerBorder)/(steps - 1);

19          for (int j = 0; j < repeats; j++) {
20              for (int i = 0; i < steps; i++) {
21                  float c = lowerBorder + i * dx;
22                  J48 classifier = new J48();
23                  classifier.setConfidenceFactor(c);
24                  Evaluation eval = new Evaluation(data);
25                  eval.crossValidateModel(classifier, data, numFolds,
26                          new Random());
27                  accuracy[i][j] = eval.pctCorrect() / 100.0;
28                  x[i] = c;
29              }
30          }

32          double[] result = new double[steps];
33          for (int i = 0; i < steps; i++) {
34              double tmp = DoubleStream.of(accuracy[i]).parallel().sum();
35              result[i] = tmp / repeats;
36          }

38          plotWithPython(title, "pruning confidence", "accuracy", x,
39                  result);

41      }
```

stdout

Acc aus 10 fold CV über 10 Werte gemittelt. parameter aus CVParameterSelection auf kompletten Datensätzen.

name	accuracy	parameter
autos	0.8165853658536584	-C, 0.14444444444444446, -M, 2
car	0.9314236111111109	-C, 0.32222222222222224, -M, 2
vehicle	0.7290780141843971	-C, 0.1, -M, 2

g)

```
1   import weka.classifiers.Evaluation;
2   import weka.classifiers.trees.J48;
3   import weka.core.Instances;
4   import weka.core.converters.ConverterUtils;

6   import java.io.IOException;
7   import java.text.NumberFormat;
8   import java.util.*;
9   import java.util.stream.DoubleStream;

11  /**
12   * Created by markus on 24.05.16.
13   */
14  public class Test {
15      public static void main(String... args) throws Exception {

17          Instances car = load("res/car.arff");
18          Instances autos = load("res/autos.arff");
19          Instances vehicle = load("res/vehicle.arff");

1           String[] carOpt = getOptions(car);
2           String[] autosOpt = getOptions(autos);
```

```
3         String[] vehicleOpt = getOptions(vehicle);

1     }

1     /**
2      * returns pruning for a given dataset in CVParameterSelection
3      * @param data dataset
4      * @return Options String array
5      * @throws Exception
6      */
7     public static String[] getOptions(Instances data) throws Exception {
8         OptimalJ48 oj48 = new OptimalJ48();
9         oj48.buildClassifier(data);
10        return oj48.selection.getBestClassifierOptions();
11    }
```

stdout

Acc aus 10 fold CV über 10 Werte gemittelt. parameter aus CVParameterSelection auf kompletten Datensätzen.

name	accuracy	parameter
autos	0.8165853658536584	-C, 0.14444444444444446, -M, 2
car	0.9314236111111109	-C, 0.32222222222222224, -M, 2
vehicle	0.7290780141843971	-C, 0.1, -M, 2

Gewählte Parameter entsprechen ca. dem lokalen Maximum des Graphen bzw. dem letzten Wert mit Verbesserung.
Anmerkung: Kompletter Source-Code in Moodle

h)

Wie in Aufgabenteil b) zu sehen war, steigt mit der pruning confidence meist auch die accuarcy. Dieses Wachstum stagniert jedoch ab einem bestimmten Punkt. CVParameterSelection wählt nun aus einer vorher festgelegten Menge von Optionen die Werte aus einem angegebenen Intervall, welche die niedrigste error rate aufweisen. Diese wird mithilfe einer n fold CV ermittelt, wobei n eine vorher belegte Variable ist. Da CVParameterSelection die Option nur dann updatet, wenn eine bessere gefunden wurde und die Optionenswerte in steigender Reihenfolge versucht werden, wird der kleinste beste Wert gewählt. Dadurch bleibt das Model (zumindest beim pruning) so einfach wie möglich.