Lernverfahren autonomer Roboter - Übung 8

G10
Andre Osse
Waldemar Stockmann
Markus Strehling
Tobias Hahn

January 11, 2017

1 Decision Tree

1.1 Continous data

Bei andauernden Datenströmen kann man einen Entscheidungsbaum schlecht lernen, da für die Entscheidung, welches Attribut man für die nächste Entscheidung nimmt, statistische Daten über die Verteilung der Labels anfallen, welche nur mit annähernd kompletten Daten herausgefunden werden können.

1.2 Same path twice

In dem selben Pfad wird das gleiche Attribut nie zweimal getestet, da davon ausgegangen wird dass das Ergebnis dieser Abfrage immer eindeutig ist, d.h. die Ausprägungen des Attributs sind immer eindeutig und bleiben es auch. Dann macht es nicht Sinn, das gleiche Attribut nocheinmal abzufragen, da ja die Subbäume die Ausprägungen des Attributs unter sich aufteilen. Sinn machen würde dies dann, wenn das Attribut nicht eindeutig ist, also einmal den einen Wert und dann wieder einen anderen annehmen, oder gleich mehrere Ausprägungen aufeinmal annehmen kann. In diesem Fall macht zweimal nachfragen Sinn, da unterschiedliche Ergebnisse entstehen können, also sich der Knoten wieder in Subbäume aufspalten würde.

1.3 Information gain

Der Gain einer Partitionierung ist gegeben durch die Formel

$$Gain(X,T) = Info(T) - Info(X,T)$$

Um also zu zeigen dass der Gain von der Partitionierung 0 ist, muss gezeigt werden dass $Info(T) = Info(X,T)$. Dafür ist es hilfreich, sich deren Definitionen zu vergegenwärtigen. Die verwendeten Zeichen sind:

- n = Anzahl der negativen Beispiele
- p = Anzahl der positiven Beispiele
- $|T_i|$ = Anzahl der Elemente in Partition i
- —T— = Gesamtanzahl der Elemente
- n_i = Anzahl der negativen Beispiele in Partition i
- p_i = Anzahl der positiven Beispiele in Partition i

$$Info(T) = I(\frac{p}{p+n}, \frac{n}{p+n}) = -(\frac{p}{p+n} * log_2(\frac{p}{p+n}) + \frac{n}{p+n} * log_2(\frac{n}{p+n}))$$
$$Info(X,T) = \sum_i \frac{|T_i|}{|T|} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$$

Wir wissen, dass $\frac{p_i}{p_i+n_i}$ für alle Partitionen gleich ist. Nun müssen wir noch zeigen dass daraus auch folgt dass $\frac{n_i}{p_i+n_i}$ für alle Partitionen gleich ist, dass hätten wir gezeigt dass wir uns das gewichtete Mitteln sparen können, da das gewichtete Mitteln von immergleichen Zahlen immer die gleiche Zahl ergibt. Dies ist jedoch einfach zu zeigen, da $\frac{n_i}{p_i+n_i} = 1 - \frac{p_i}{p_i+n_i}$, d.h. wenn das eine für alle gleich ist ist das andere Verhältnis auch für beide gleich. Damit kann man die zweite Gleichung vereinfachen zu:

$$Info(X,T) = I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$$

Diese vereinfachte Form sieht nun schon fast so aus wie der Informationsgehalt von Info(T), nur dass statt der Gesamtanzahl von n und p jeweils die Anzahlen in einer der Partitionen stehen. Da wir aber gezeigt haben dass beide Verhältnisse in allen Partitionen gleich sind, so folgt daraus dass das Verhältnis auch in der Summe der Partitionen das gleiche ist. D.h. Info(T) = Info(X,T), und damit Gain = 0.

1.4 Decision tree learning

1.4.1 Code

../code/decisiontree.py

```
1  import numpy as np
2  import pandas as pd
3  from sklearn.model_selection import KFold
4  from collections import Counter
5  import matplotlib.pyplot as plt
6  import math
7
8
9  class DecisionTree():
10     """ Representing a decision tree class.
11     """
12     def __init__(self):
13         self.tree = None
14
15     def apply_k_fold_cv(self, X, y, classifier=None, n_folds=5, **kwargs):
16         """K fold cross validation.
17
18         Parameters
19         ----------
20         X : array-like, shape (n_samples, feature_dim)
21             The data for the cross validation
22
23         y : array-like, shape (n-samples, label_dim)
24             The labels of the data used in the cross validation
25
26         classifier : function
27             The function that is used for classification of the training data
28
29         n_splits : int, optional (default: 5)
30             The number of folds for the cross validation
31
32         kwargs :
33             Further parameters that get used e.g. by the classifier
34
35         Returns
36         -------
37         accuracies : array, shape (n_splits,)
38             Vector of classification accuracies for the n_splits folds.
39         """
40         assert X.shape[0] == y.shape[0]
41
42         if len(X.shape) < 2:
43             X = np.atleast_2d(X).T
44         if len(y.shape) < 2:
45             y = np.atleast_2d(y).T
46
47         cv = KFold(n_splits=n_folds, shuffle=True, random_state=42)
48         scores = []
49
50         for train_index, test_index in cv.split(X):
51             train_data = X[train_index, :]
52             train_label = y[train_index, :]
53             test_data = X[test_index, :]
54             test_label = y[test_index, :]
55
56             score = classifier(train_data, test_data,
57                                train_label, test_label, **kwargs)
58
```

```
59              scores.append(score)
60
61          return np.array(scores)
62
63      def dt_classifier(self, X_train, X_test, y_train, y_test,
64                        attr_names=None, max_depth=-1, **kwargs):
65          """ Decision tree implementation.
66
67          Parameters
68          ----------
69          X_train : array-like, shape (n_samples, feature_dim)
70              The data to train the classifier
71
72          X_test : array-like, shape (n-samples, label_dim)
73              The data to test the learned classifier
74
75          y_train : array-like, shape (n_samples, 1)
76              The labels for the training data
77
78          y_test : array-like, shape (n_samples, 1)
79              The labels for the test data
80
81          attr_names : list of tuple
82              A list of tuples, where the first element is the index and the
83              second the name of the attribute corresponding to the data
84
85          max_depth : int, optional (default: -1)
86              The maximal depth of the decision tree
87
88          kwargs :
89              Further parameters that get used e.g. by the classifier
90
91          Returns
92          -------
93          accuracy : double
94              Accuracy of the correct classified test data
95          """
96          y_train = y_train.ravel()
97          y_test = y_test.ravel()
98
99          assert len(attr_names) > 0
100
101         if max_depth == -1 or max_depth > len(attr_names):
102             max_depth = len(attr_names)
103
104         # Additionally work with a list of indices.
105         idx = np.arange(X_train.shape[0])
106
107         # First build the classifier
108         dtn = DecisionTreeNode()
109         self.tree = dtn.make_tree(idx, X_train, y_train, attr_names, max_depth)
110
111         ### YOUR IMPLEMENTATION GOES HERE ###
112         ### Use the learned classifier to evaluate the performance
113         ### on the test set. Return as written in the specification
114         true = 0
115         false = 0
116
117         for x,y in zip(X_test, y_test):
118             pred = dtn.classify(x)
119             if (pred == y):
120                 true += 1
121             else:
122                 false += 1
123
124         return true / (true+false)
125
126
127 class DecisionTreeNode():
128     """ Class to represent a tree node.
129         Leaves are empty trees that only have a label and no children
```

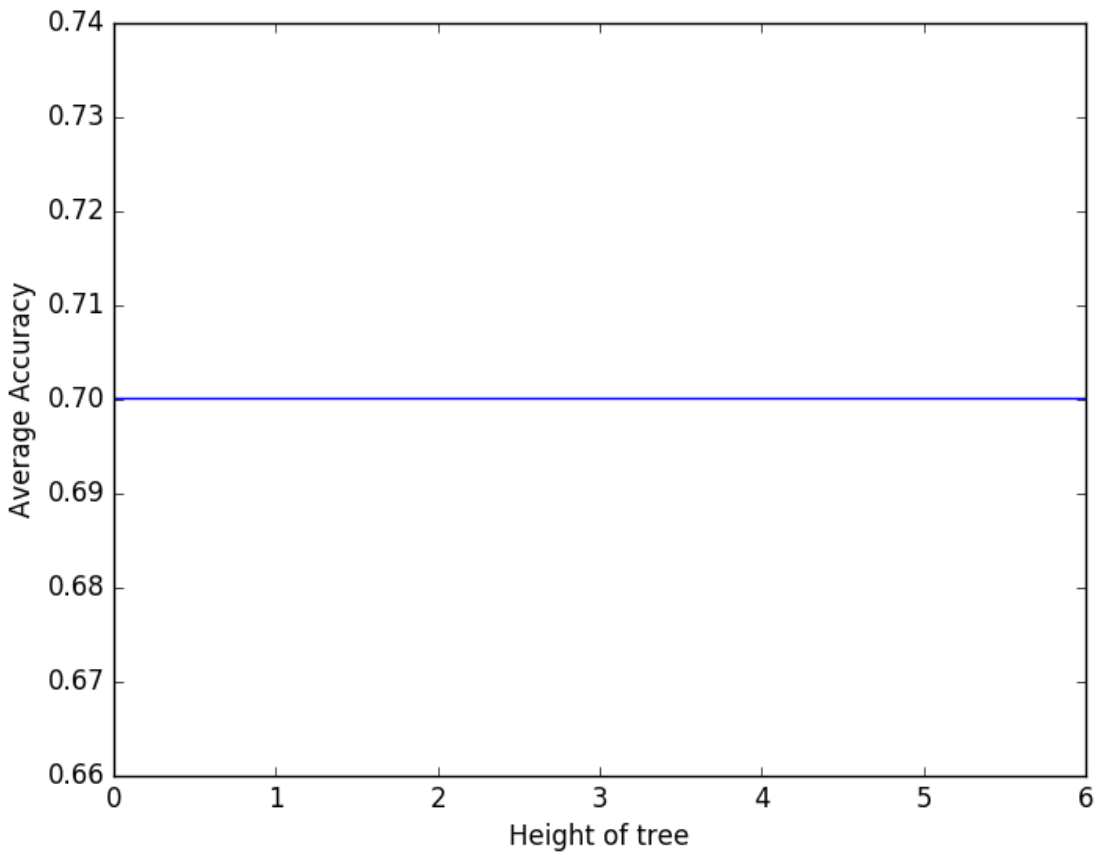
```
130     """
131     def __init__(self):
132         self.label = None
133         self.attribute = None
134         self.attribute_value = None
135         self.children = []  # List of DecisionTreeNodes
136 
137     def calc_impurity(self, class_labels):
138         """ Calculate the impurity for a subset of the data
139 
140         Parameters
141         ----------
142         class_labels : array-like, shape (n_labels,)
143             A list of class labels to calculate the impurity for.
144 
145         Returns
146         -------
147         impurity : double
148             Impurity for the set of given class labels
149         """
150 
151         ### YOUR IMPLEMENTATION GOES HERE ###
152         ### Decide for an impurity measure of your choice.
153 
154     def classify(self, data_vector):
155         """ Recursive method to assign a label to a given data point.
156 
157         Parameters
158         ----------
159         data_vector : array-like, shape (n_features,)
160             One data point to be classified
161 
162         Returns
163         -------
164         label :
165             The label for the classified data point
166         """
167 
168         ### YOUR IMPLEMENTATION GOES HERE ###
169         ### Use the tree structure to recursively go through
170         ### all decisions for the tree
171         ### If the attribute value was not in the training set
172         ### weight each possible branch uniform and return the
173         ### most common label.
174         if not self.children:
175             return self.label
176         exp = data_vector[self.attribute]
177 
178         if exp not in [c.attribute_value for c in self.children]:
179             return Counter([c.classify(data_vector) for c in self.children]).most_common(1)
                [0][0]
180 
181         for c in self.children:
182             if (c.attribute_value == exp):
183                 return c.classify(data_vector)
184 
185 
186 
187     def make_tree(self, idx_lst, data, label, attributes, max_depth=0,
188                   default=None):
189         """ Create the decision tree recursively for the given data.
190 
191         Parameters
192         ----------
193         idx_lst : array-like, shape (n_samples,)
194             A list of indices to represent a certain subset of the data
195 
196         data : array-like, shape (n_samples, n_features)
197             The data that is to used to train the decision tree
198 
199         label : array-like, shape (n_samples,)
```

```
200             The labels for all the training data.
201
202         attributes : list of tuple
203             A list of tuples, where the first element is the index and the
204             second the name of the attribute corresponding to the data
205
206         max_depth : int, optional (default: -1)
207             The maximal depth of the decision tree to be formed
208
209         default : label
210             The default label for the case no decision can be made.
211
212         Returns
213         -------
214         self : object
215             Reference to a tree node structure representing a (sub)tree.
216         """
217
218         ### YOUR IMPLEMENTATION GOES HERE ###
219         ### Decide on which attribute to split the data
220         ### referenced by the idx_list and create corresponding
221         ### subtrees. Build recursivly the whole decision tree
222         ### up to the desired maximum depth
223         if (len(attributes) == 0):
224             self.label = Counter([label[i] for i in idx_lst]).most_common(1)[0][0]
225             return self
226         if (max_depth == 0):
227             self.label = Counter([label[i] for i in idx_lst]).most_common(1)[0][0]
228             return self
229         if (max_depth > -1):
230             max_depth -= 1
231
232         info = self.info(*self.count_whole(idx_lst, label))
233         infos = [self.comp_gain(info, self.partition(idx_lst, data, label, a)) for a in
                attributes]
234         a_index = np.argmax(infos)
235
236         a = attributes[a_index]
237         new_attribs = attributes[:]
238         new_attribs.remove(a)
239         self.attribute = a[0]
240         exps = list(set([data[i][a[0]] for i in idx_lst]))
241         idx_lsts = {}
242         for exp in exps:
243             lst = []
244             for i in idx_lst:
245                 if (data[i][a[0]] == exp):
246                     lst.append(i)
247             idx_lsts[exp] = lst
248
249         self.children = [DecisionTreeNode() for exp in exps]
250         for i,x in enumerate(exps):
251             self.children[i].make_tree(idx_lsts[exp], data, label, new_attribs, max_depth,
                    default)
252         for i,x in enumerate(exps):
253             self.children[i].attribute_value = x
254         return self
255
256     def info(self, v, g, a, n):
257         w = v + g + a + n
258         v_v = 0
259         g_v = 0
260         a_v = 0
261         n_v = 0
262         if (v > 0):
263             v_v = v/w * math.log(v/w, 2)
264         if (g > 0):
265             g_v = g/w * math.log(g/w, 2)
266         if (a > 0):
267             a_v = a/w * math.log(a/w, 2)
268         if (n > 0):
```

```
269             n_v = n/w * math.log(n/w, 2)
270         return -(v_v + g_v + a_v + n_v)
271
272     def comp_gain(self, info, attrib):
273         whole = sum([sum(d.values()) for d in attrib[1:]])
274         infos = []
275         for exp in attrib[0]:
276             count = sum([d[exp] for d in attrib[1:]])
277             infos.append(count / whole * self.info(attrib[1][exp], attrib[2][exp], attrib
                    [3][exp], attrib[4][exp]))
278         return info - sum(infos)
279
280     def count_whole(self, idx_list, label):
281         v = 0
282         g = 0
283         a = 0
284         n = 0
285
286         for i in idx_list:
287             lbl = label[i]
288
289             if (lbl == "vgood"):
290                 v += 1
291             elif (lbl == "good"):
292                 g += 1
293             elif (lbl == "acc"):
294                 a += 1
295             else:
296                 n += 1
297
298         return (v, g, a, n)
299
300
301     def partition(self, idx_list, data, label, attribute):
302         exps = set([])
303         vgood = {}
304         good = {}
305         acc = {}
306         nacc = {}
307
308         for i in idx_list:
309             exp = data[i][attribute[0]]
310             if exp not in exps:
311                 exps.add(exp)
312                 vgood[exp] = 0
313                 good[exp] = 0
314                 acc[exp] = 0
315                 nacc[exp] = 0
316             lbl = label[i]
317             if (lbl == "vgood"):
318                 vgood[exp] += 1
319             elif (lbl == "good"):
320                 good[exp] += 1
321             elif (lbl == "acc"):
322                 acc[exp] += 1
323             else:
324                 nacc[exp] += 1
325
326         return (exps, vgood, good, acc, nacc)
327
328
329
330 def read_in_data(filename):
331
332     names = ["buying", "maint", "doors", "persons", "lug_boot",
333              "safety", "Class"]
334     data = pd.read_csv(filename, names=names)
335     data = data.dropna()
336
337     attribute_names = list(data.dtypes.index[:-1])
338
```

```
339     data = data.as_matrix()
340     labels = data[:, -1]
341     data = data[:, :-1]
342
343     return (data, labels, list(enumerate(attribute_names)))
344
345
346 if __name__ == '__main__':
347     (data, labels, attr_names) = read_in_data("../data/car.data")
348     dt = DecisionTree()
349
350     ### YOUR IMPLEMENTATION GOES HERE ###
351     av_accuracies = [np.mean(dt.apply_k_fold_cv(data, labels, dt.dt_classifier, 10,
            attr_names=attr_names, max_depth=d)) for d in range(1,8)]
352     print(av_accuracies)
353     plt.plot(av_accuracies)
354     plt.ylabel("Average Accuracy")
355     plt.xlabel("Height of tree")
356     plt.show()
357
358     tree = DecisionTreeNode()
359     tree.make_tree(np.arange(data.shape[0]), data, labels, attr_names, 2)
360     print("[{0}](None)".format(attr_names[tree.attribute][1]))
361     new_childs = []
362     for c in tree.children:
363         print("|[{0}]({1})".format(attr_names[c.attribute][1], c.attribute_value))
364         new_childs.extend(c.children)
365     for c in new_childs:
366         print("||={0}".format(c.label))
```

1.4.2 Plot



1.4.3 Visualisation

```
[safety](None)
|[buying](high)
|[buying](med)
|[buying](low)
||=unacc
||=unacc
||=unacc
||=unacc
||=unacc
||=unacc
||=unacc
||=unacc
||=unacc
||=unacc
||=unacc
```

2 Support Vector Machines

2.1 Unequal Classes

Wenn die Klassenverteilung sehr ungleich ist, dann tritt bei SVM folgendes Problem auf: Da wir den Abstand der Datenpunkte zur Hyperebene maximieren, kann es Sinn machen die Trennlinie möglichst weit von den Punkten der größeren Klasse wegzuschieben, auch wenn dabei die Trennschärfe verloren geht, da der Abstand der zu den Punkten der großen Menge gewonnen wird größer ist als der der verloren geht dadurch dass der Abstand zu den kleineren Punkten verloren geht. Damit würde SVM nicht mehr richtig klassifizieren, obwohl das Maximierungsziel erreicht wurde.

2.2 Workings

Das Maximisierungsproblem von SVM besteht darin den Abstand der Datenpunkte von der Hyperebene zu maximieren. Dafür wird eine entsprechende Fehlerfunktion gewählt, welche nicht einfach nur die richtige Klassifizerung des Datenpunkts angibt, sondern wieviel Abstand zwischen der Trennlinie und dem Datenpunkt besteht. Der wichtigste Tuningfaktor ist dabei C, also der Faktor der angibt wie wichtig es relativ ist, den Fehler zu minimieren, anstatt die Regularisierungskomponente zu maximieren. Ist dieser Faktor zu gering gewählt, so achtet der Algorithmus eher darauf die Gewichte der Linie minimal zu halten, anstatt richtig zu klassifizieren. Ist er zu groß gewählt, so maximiert der Algorithmus die Klassifizerung, wodurch Outlier die Hyperebene stark beeinflussen und den Abstand zu den Datenpunkten verringert.

2.3 Kerneltrick

Normalerweise schafft es SVM nur, Datenpunkte linear zu trennen. Dies kann jedoch ein Problem sein, wenn Datenpunkte gar nicht linear getrennt sind sondern z.B. polynomial. Hier kann man abhilfe schaffen, indem man die Datenpunkte in neue Datenpunkte mappt, mithilfe eines Kernels. Dieser gibt uns an wie ein Datenpunkt gemappt wird, z.B. x auf x^2. Nun wird dieser Abstand maximiert, wodurch sich Trennlinien ergeben können die nicht linear sind. Prinzipiell kann es immer angewandt werden wenn man einen Kernel hat und der Kernel gewisse Bedingungen erfüllt.

2.4 Decision Boundary

Die Decision Boundary verändert sich natürlich, wenn der Kernel gewechselt wird. Dies liegt daran dass nun die Fehlerfunktion nicht mehr auf die Datenpunkte, sondern auf die gemappten Datenpunkte angewandt wird, d.h. der Abstand zwischen zwei Punkten kann sich gewaltig verändern, wodurch sich auch die Decision Boundary verändert.