HW4

March 15, 2019

0.0.1 Homework 4

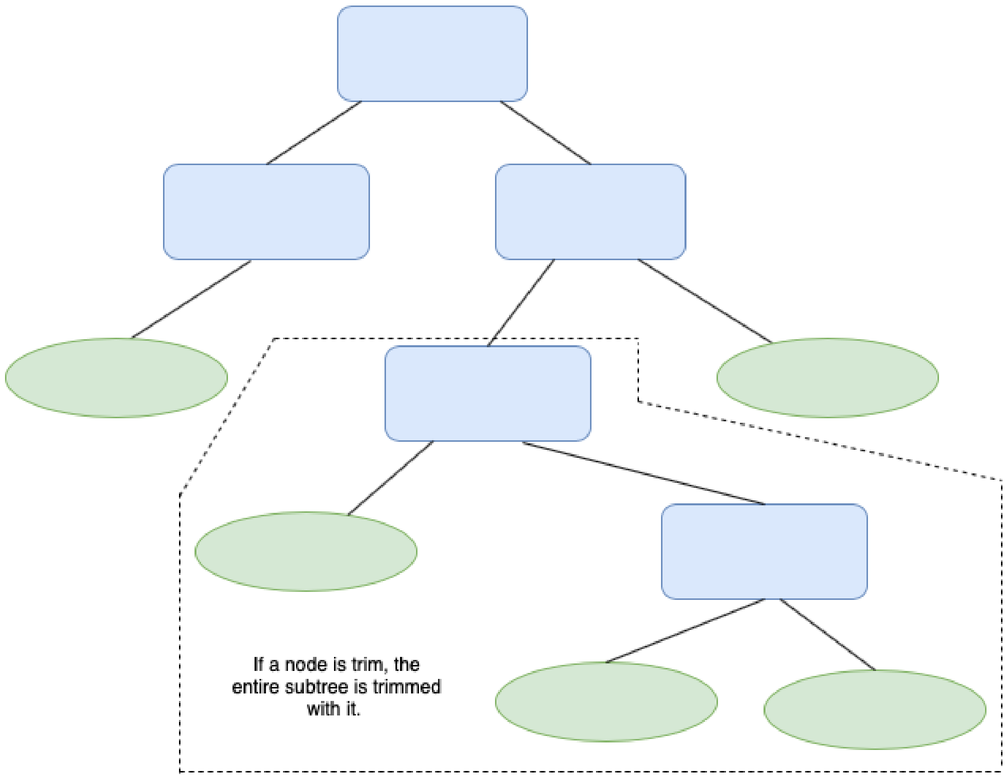
Erin Witmer CSC 440

[8.3] Given a decision tree, you have the option of (a) converting the decision tree to rules and then pruning the resulting rules, or (b) pruning the decision tree and then converting the pruned tree into rules. What advantage does (a) have over (b)? A decision tree can be converted to IF-THEN rules by tracing the path from the root node to each leaf in the tree. If the tree is pruned before the rules are created, as is suggested in case (b), then any path dependent on the trimmed node will also be trimmed. First creating the rules, and then trimming them as proposed in (a) allows for more precision when trimming. Leaves dependent on a specific preceding node may be trimmed, while other leaves dependent on that same preceding node may be kept. However, it is important to note the higher computational complexity associated with building a full tree, converting to rules and then trimming. Regardless, method (a) generally generates a more precise and accurate set of rules and is worth the computational cost.

[8.5] Given a 5-GB data set with 50 attributes (each containing 100 distinct values) and 512 MB of main memory in your laptop, outline an efficient method that constructs decision trees in such large data sets. Justify your answer by rough calculation of your main memory usage. The use of the RainForest algorithm would be most effective in this case. RainForest adapts to the amount of main memory available and applies to any decision tree induction algorithm. The first step is to construct an AVC-set for each attribute. If the total data set is 5MB, on average each attribute would be 100MB. The process would be to extract one attribute, calculate the AVC, and store that set in main memory. On average, if you assume you are storing the value/class count as an integer (4 bytes), there are 100 values per attribute, and assume there are 2 class outcomes, an individual AVC set would require approximate (100 x 2 x 4 bytes) = 800 bytes. So the data holding the aggregate information from which the decision tree can be constructed is only 40 KB. Since this is the aggregate data, every level of the tree will be some derivative of this data. Utilizing the basic algorithm for inducing a decision tree, you could loop through many times without running out of memory. If the tree got very deep and space was an issue, tree prunning could take place while the tree is being generated and only summary information could be retained at each level.

[8.7] The following table consists of training data from an employee database. The data have been generalized....

```
In [1]: data = {'t0': ['sales', 'senior', '3135', '46K50K', 30],
                't1': ['sales', 'junior', '2630', '26K30K', 40],
                't2': ['sales', 'junior', '3135', '31K35K', 40],
```




```
't3': ['systems', 'junior', '2125', '46K50K', 20],
't4': ['systems', 'senior', '3135', '66K70K', 5],
't5': ['systems', 'junior', '2630', '46K50K', 3],
't6': ['systems', 'senior', '4145', '66K70K', 3],
't7': ['marketing', 'senior', '3640', '46K50K', 10],
't8': ['marketing', 'junior', '3135', '41K45K', 4],
't9': ['secretary', 'senior', '4650', '36K40K', 4],
't10': ['secretary', 'junior', '2630', '26K30K', 6]}
```

(a.) How would you modify the basic decision tree algorithm to take into consideration the count of each generalized data tuple? The relative weights of each of the tuples in the entropy calculation needs to be adjusted to reflect the proportion of the count to total employees (sum of count).

(b.) Use your algorithm to construct a decision tree from the given data.

```
In [2]: import pandas as pd
        import math
        from functools import reduce

        def entropy(a,b):
            sum = a+b
            total = -(a/sum)*math.log2((a/sum))-(b/sum)*math.log2((b/sum))
```

```
        return total

    df = pd.DataFrame.from_dict(data, orient='index', columns=['dept', 'class', 'age', 'sal
    class_count = df.groupby('class').sum()
    print(class_count)
    head = entropy(113,52)
```

```
        count
class
junior    113
senior     52
```

```
In [3]: # department
        dept_count = df.groupby(['dept']).sum()
        print(dept_count)
```

```
           count
dept
marketing     14
sales        110
secretary     10
systems       31
```

```
In [4]: sales = 110/165
        systems = 31/165
        marketing = 14/165
        secretary = 10/165

        dept_outcome = df.groupby(['dept','class']).sum()
        print(dept_outcome)
```

```
                  count
dept      class
marketing junior      4
          senior     10
sales     junior     80
          senior     30
secretary junior      6
          senior      4
systems   junior     23
          senior      8
```

```
In [5]: dept_gain = head - (sales*entropy(80,30)
                            +systems*entropy(23,8)
                            +marketing*entropy(4,10)
                            +secretary*entropy(6,4))
        print(dept_gain)
```

```
0.048606785991983426
```

```
In [6]: # age
        age_count = df.groupby(['age']).sum()
        print(age_count)
```

```
      count
age
2125     20
2630     49
3135     79
3640     10
4145      3
4650      4
```

```
In [7]: early20=20/165
        late20=49/165
        early30=79/165
        late30=10/165
        early40=3/165
        late40=4/165

        age_outcome = df.groupby(['age','class']).sum()
        print(age_outcome)
```

```
             count
age  class
2125 junior     20
2630 junior     49
3135 junior     44
     senior     35
3640 senior     10
4145 senior      3
4650 senior      4
```

```
In [8]: age_gain = head - (early30*entropy(44,35))
        print(age_gain)
```

```
0.4247351209783661
```

```
In [9]: # salary
        salary_count = df.groupby(['salary']).sum()
        print(salary_count)
```

```
        count
salary
```

```
26K30K     46
31K35K     40
36K40K      4
41K45K      4
46K50K     63
66K70K      8
```

```
In [10]: high20=46/165
         low30=40/165
         high30=4/165
         low40=4/165
         high40=63/165
         high60=8/165

         salary_outcome = df.groupby(['salary','class']).sum()
         print(salary_outcome)
```

```
                count
salary  class
26K30K junior     46
31K35K junior     40
36K40K senior      4
41K45K junior      4
46K50K junior     23
        senior     40
66K70K senior      8
```

```
In [11]: salary_gain = head - (high40*entropy(23,40))
         print(salary_gain)
```

```
0.5375181264158646
```

```
In [12]: # The first level of the decision tree is by salary
         salary_age = df.groupby(['salary','age','class']).sum()
         print(salary_age)
```

```
                      count
salary  age   class
26K30K 2630 junior     46
31K35K 3135 junior     40
36K40K 4650 senior      4
41K45K 3135 junior      4
46K50K 2125 junior     20
        2630 junior      3
        3135 senior     30
        3640 senior     10
```

```
66K70K 3135 senior      5
        4145 senior      3
```

```
In [13]: salary_dept = df.groupby(['salary','dept','class']).sum()
         print(salary_dept)
```

```
                          count
salary  dept      class
26K30K sales     junior     40
        secretary junior      6
31K35K sales     junior     40
36K40K secretary senior      4
41K45K marketing junior      4
46K50K marketing senior     10
        sales     senior     30
        systems   junior     23
66K70K systems   senior      8
```

The first level of the decision tree is a split by salary, because the information gain is the highest based on that split. There is only one ambiguous category after the salary split (46k...50k), and a further split based on either age (21...25 -> junior, 26...30 -> junior, 31...35 -> senior, 36...40 -> senior) or department (marketing -> senior, sales -> senior, systems -> junior) will result in a pure split of the data.

(c.) Given a data tuple having the values "systems," "26...30," and "46–50K" for the attributes department, age, and salary, respectively, what would a naive Bayesian classification of the status for the tuple be?

```
In [14]: p_senior = 52/165
         p_junior = 113/165

         p_senior_systems = 8/52
         p_junior_systems = 23/113

         p_senior_26 = 1/52 # modifier
         p_junior_26 = 49/113

         p_senior_46 = 40/52
         p_junior_46 = 23/113

         senior = p_senior * p_senior_systems * p_senior_26 * p_senior_46
         junior = p_junior * p_junior_systems * p_junior_26 * p_junior_46

         print(senior)
         print(junior)
```

```
0.0007172314864622557
0.012302997078625555
```

The Bayesian classification would be junior.

[8.12] The data tuples of Figure 8.25 are sorted by decreasing probability value, as returned by a classifier. For each tuple, compute the values for the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). Compute the true positive rate (TPR) and false positive rate (FPR). Plot the ROC curve for the data.

In [15]:
```
roc = pd.read_excel('roc.xlsx', index_col=0, dtype={'TPR': float, 'FPR': float})
roc
```

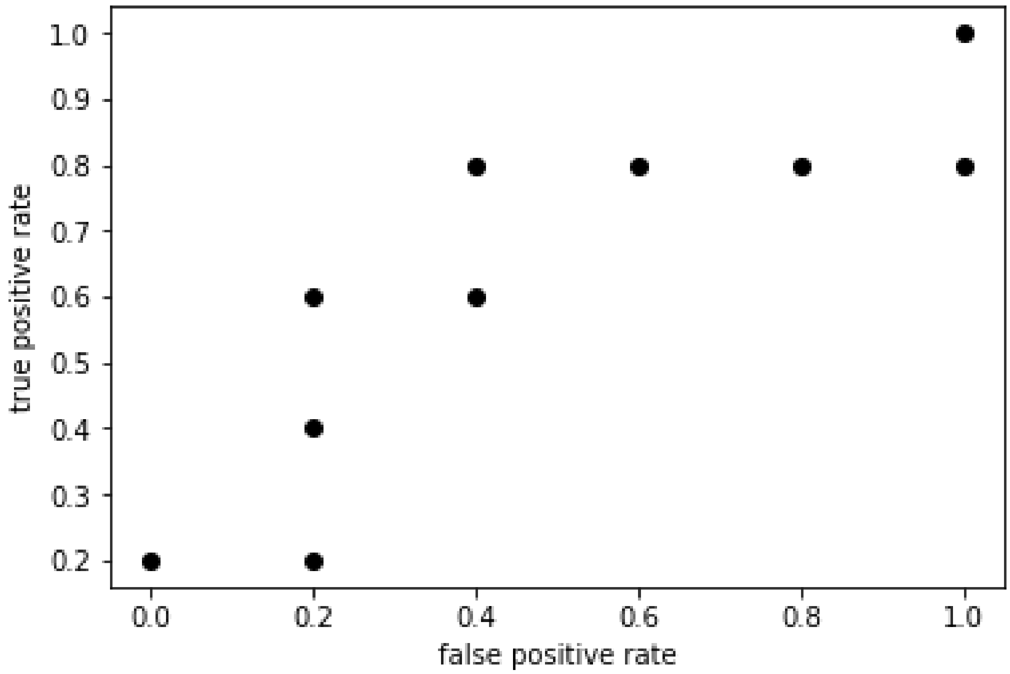
Out[15]:

Tuple	Class	Prob.	TP	FP	TN	FN	TPR	FPR
1	P	0.95	1	0	5	4	0.2	0.0
2	N	0.85	1	1	4	4	0.2	0.2
3	P	0.78	2	1	4	3	0.4	0.2
4	P	0.66	3	1	4	2	0.6	0.2
5	N	0.60	3	2	3	2	0.6	0.4
6	P	0.55	4	2	3	1	0.8	0.4
7	N	0.53	4	3	2	1	0.8	0.6
8	N	0.52	4	4	1	1	0.8	0.8
9	N	0.51	4	5	0	1	0.8	1.0
10	P	0.40	5	5	0	0	1.0	1.0

In [16]:
```
import matplotlib.pyplot as plt
tpr = roc.iloc[:,6].values
fpr = roc.iloc[:,7].values
```

In [17]:
```
plt.plot(fpr, tpr, 'o', color='black');
plt.xlabel('false positive rate')
plt.ylabel('true positive rate')
```

Out[17]: `Text(0,0.5,'true positive rate')`



[8.14] Suppose that we want to select between two prediction models, M1 and M2. We have performed 10 rounds of 10-fold cross-validation on each model, where the same data partitioning in round i is used for both M1 and M2. The error rates obtained for M1 are 30.5, 32.2, 20.7, 20.6, 31.0, 41.0, 27.7, 26.0, 21.5, 26.0. The error rates for M2 are 22.4, 14.5, 22.4, 19.6, 20.7, 20.4, 22.1, 19.4, 16.2, 35.0. Comment on whether one model is significantly better than the other considering a significance level of 1%.

In [18]:
```
import numpy as np
m1 = [30.5, 32.2, 20.7, 20.6, 31.0, 41.0, 27.7, 26.0, 21.5, 26.0]
m2 = [22.4, 14.5, 22.4, 19.6, 20.7, 20.4, 22.1, 19.4, 16.2, 35.0]
```

In [19]:
```
def var_diff(m1,m2):
    total = 0
    m1_avg = np.average(m1)
    m2_avg = np.average(m2)
    k = len(m1)
    d = m1_avg - m2_avg

    for i in range(k):
        add = (m1[i]-m2[i]-(m1_avg-m2_avg)) ** 2
        total += add

    total = total / k
    return d/np.sqrt(total / k)
```

```
        var_diff(m1,m2)
```

```
Out[19]: 2.4712371600876786
```

```
In [20]: from scipy.stats import t

         # two sided t-test with k-1 dof
         t.interval(0.99, df=9)
```

```
Out[20]: (-3.2498355440153697, 3.2498355440153697)
```

The p-value (2.47) is not greater than 3.25 or less than -3.25, therefore we cannot reject the null hypothesis that one model is significantly better than the other at a significance level of 1%.