CSEP546 - HW1 Answers

Name: Abhijith Jagannath
Student ID: 1776317

Autumn 2017
10 / 22 / 2017

1.1. Code description:

- Language used: python3 , (arrff reader used is liac-arff)
- Usage :
  - $ python3 main.py <path_training_set.arff> <path_test_set.arff> <confidence_level>
  - confidence_level is optional, when left out, it will run for all confidences -> 0 50 80 95 99
- Main.py :

  __main__ :

  accepts params from command line, decides on defaults and passes them to main()

  main()

  Reads and loads both training and test files
  Target_attribute decided as the last one attribute_indecies[-1]
  It also measures each sub functions and timings
  We call the create_dtree from here and use the returned tree to call the classification functions.

- DTree.py:

  dtree_init() -> we do bunch of initializations here. Store all the common values such as data, target_attribute, multithreading pools, valid values of attributes, etc..

  dtree_close() -> clears what we initiated.

  dtree_create() -> a wrapper function, it collects everything from main, calls init to store common values and then calls grow_dtree to grow the tree.

  entropy() -> gives the entropy of given counts of value (We basically pass a counted hash here)

  process_attr() -> processes one attribute at a time, it stores all the valid counts in a counter.

  We decide what to do with unknowns here. Basically unknown counts are added to the counts of most common value. We will also store most common value and least common value of that attribute to a hash.(also the row numbers of data where each value of attribute is present in data for that node, it will be useful when calling this function again and process only that rows.)

  process_data() -> Here will process the data by making use of process_attr. We calculate, all the necessary things here. Mainly gain ratio and decide on what to do next. If we found a best_attribute, we pass all the information we got about it from the process_attr.

  split_test() -> this does the chi2 test.

  grow_dtree() -> This is our main recursive function. It calls the process_data function and gathers all the information it wants, calls split_test and decides to weather to grow further or not.

- classifier.py:

  This basically has all the classifier and helper functions. This also calculates the paths taken by examples.

1.2. Classification results on **Training** Set:

CONFIDENCE %	ACCURACY %	PRECISION %	RECALL %
0	93.052	90.746	71.718
50	85.207	98.963	24.465
80	81.327	98.285	4.406
95	80.597	82.857	0.743
99	80.595	81.690	0.743

1.3 Classification results on **Test** Set:

CONFIDENCE %	ACCURACY %	PRECISION %	RECALL %
0	67.428	32.671	29.45
50	74.252	45.258	10.998
80	75.004	54.919	3.814
95	74.876	78.947	0.238
99	74.876	78.947	0.238

1.4 Analyzing the Constructed Tree:

CONFIDENCE %	Num Nodes	Num Leaves	First few attributes (sneak peak)
0	14349	23939	[13, 8, 207, 2]
50	1118	2362	[13, 8, 207, 2]
80	152	280	[13, 8, 207, 2]
95	9	21	[13, 8, 167, 166]
99	8	20	[13, 8, 167, 166]

1.5 Choosing the best tree:

In my view the one which is doing good on the test data and reasonably well on train data is the best one. Hence it is confidence level 80.

- Its accuracies are close for train(81%) and test(75%) data. Meaning it is somewhat treating the datas in a similar way.
- It is not overfitting the train data
- It has reasonable number of nodes / leaves (~150)
    - For 0 and 50, node/ leaves are too high (> 1000)
    - For 95 and 99 node/ leaves are very low (< 10)

1.6 Tree interpretation:

Highest accuracy tree gives this:

[(13, 'Friend/Co-worker'), (8, 'spouse'), (207, "\\'(-inf-0.5]\\"), (2, 'Male')]

13 -> HowDidYouFindUs

8 -> WhoMakesPurchasesForYou

207 -> Num Wax Product Views

2 -> Gender

Which means , If the user came to website due to **reference given by Friend/Co-worker** , user's **spouse does the purchases** for him/her, **number of wax product views is very low** and also **user is a Male.** He is likely to stay.

I get the same interpretation for majority of both negatively and positively labeled cases in the train set. The sentences by themselves makes sense. But if you look at them combined, they do not make sense!!

1.7 It was very hard to improve on 75% accuracy. Few things I tried:

- Use least common values for values found in test and not in train.
    - Improved to 75.032 from 75.004
- Use a pre calculated split information of each attribute for the whole set VS calculate at it at each split point. (This is a separate script DTree_v2.py which should be included in main and comment DTree.py). Thanks Alon for confirming that split_information is for whole dataset!
    - Improved to 75.14 from 75.032 (
- While calculating the information gain, it matters if we subtract the sum entropies by total entropy. However not subtracting didn't help!

2.1.1. There will be 2^d / 2 rules learned.

Each path of the tree can be directly converted into a rule. However, we can remove all negative classification paths, because we can only be interested in rules that will positively classify new examples and default to a negative classification if no rule applies. And we can use the reverse logic of this where negative classification is of interest (default to positive). So this means we can nicely prune and for 2^d paths from the node, we can have half of them as rules.

2.1.2. Each rule will have d+1 preconditions.

Each path will be of depth d, plus a root node at the beginning of each path as a precondition to all of them.

2.1.3. (2^d / 2) x (d + 1) distinct choices.

Number of rules x precondition for each one.

2.1.4. I think rule based system will most likely to overfit.

On surface it looks like both systems are equally prone to overfitting. However, my vote goes to rule based system as it will (even with stopping mechanisms) have replications of training values within the preconditions of several rules. So, it will try to fit tightly to the training data, which can be small compared to real world and also I think it will have trouble dealing with all the unknown values in there.

2.2.1. $2n^2 + 4n$

P(a,b) != P(b,a), so position matters!
When n variables are reused → n x n
When new variable is introduced at each position → n + n
Total → $n^2 + 2n$
Including negations → 2 x ($n^2 + 2n$)

2.2.2. Let's assume that this rule do not exist. It will have consequences at both training and test time.

When training, for instance, the new literal with only new variables are considered, there is a chance that the of the new variables are related to the existing ones or not! If they are not related, then it can be that this new literal which is considered is already in the clause with a different name and variable positions! So we may just have many many duplicates. Which makes the algorithm inefficient.

During testing, if the new literal is unrelated to the variables, then it will cause the program to go search the whole world (our database) for it slowing down significantly!. If it finally finds a result, it will always be true or always be false regardless of our input query !