HOMEWORK
DECISION TREES *

100074302 INTRODUCTION TO MACHINE LEARNING (SPRING 2024)

OUT: March 26, 2024
DUE: April 25, 2024

Summary It's time to build your first end-to-end learning system! In this assignment, you will build a Decision Tree classifier and apply it to several binary classification problems. You will implement Decision Tree learning, prediction, and evaluation.

START HERE: Instructions

- **Submitting your work:** You will upload your codes in a zip file to the Lexue platform.
- **Materials:** The data that you will need in order to complete this assignment is posted along with the instruction.

1 Programming:

Your goal in this assignment is to implement a binary classifier, entirely from scratch–specifically a Decision Tree learner. In addition, we will ask you to run some end-to-end experiments on three tasks (predicting the party of a politician / predicting final grade for high school students / predicting whether a mushroom is poisonous) and report your results. You will write two programs: `inspection.{py|java|cpp}` (Section 1.2) and `decisionTree.{py|java|cpp}` (Section 1.3). You may write your programs in **Python**, **Java**, or **C++**. However, you should use the same language for all parts below. In general, **Python** is recommended.

1.1 The Tasks and Datasets

Materials Download the zip file from the course website. The zip file will have a folder that contains all the data that you will need in order to complete this assignment.

Datasets The handout contains four datasets. Each one contains attributes and labels and is already split into training and testing data. The first line of each `.tsv` file contains the name of each attribute, and *the class is always the last column.*

1. **politician:** The first task is to predict whether a US politician is a member of the Democrat or Republican party, based on their past voting history. Attributes (aka. features) are short descriptions of bills that were voted on, such as *Aid_to_nicaraguan_contras* or *Duty_free_exports*. Values are given as *'y'* for yes votes and *'n'* for no votes. The training data is in `politicians_train.tsv`, and the test data in `politicians_test.tsv`.
2. **education:** The second task is to predict the final *grade* (A, not A) for high school students. The attributes (covariates, predictors) are student grades on 5 multiple choice assignments *M1* through *M5*, 4 programming assignments *P1* through *P4*, and the final exam *F*. The training data is in `education_train.tsv`, and the test data in `education_test.tsv`.

*Compiled on Wednesday 27th March, 2024 at 23:54

3. **small:** We also include `small_train.tsv` and `small_test.tsv`—a small, purely for demonstration version of the politicians dataset, with *only* attributes *Anti_satellite_test_ban* and *Export_south_africa*. For this small dataset, the handout tar file also contains the predictions from a reference implementation of a Decision Tree with max-depth 3 (see `small_3_train.labels`, `small_3_test.labels`, `small_3_metrics.txt`). You can check your own output against these to see if your implementation is correct.[1]

4. **mushrooms:** Finally, we have a large dataset of mushroom samples, `mushrooms_train.tsv` and `mushrooms_test.tsv`, for you to test your algorithm against. Each sample has discrete attributes which were split into boolean attributes. For example, *cap-size* could be *bell*, *conical*, *convex*, *flat*, *knobbed*, or *sunken*. This was split into boolean attributes *cap-size_bell*, *cap-size_conical*, *cap-size_convex*, etc. Your goal is to differentiate the poisonous versus edible mushrooms.

Note: For simplicity, all attributes are discretized into just two categories (i.e. each node will have at most two descendents). This applies to all the datasets in the handout, as well as the additional datasets on which we will evaluate your Decision Tree.

[1] Yes, you read that correctly: we are giving you the correct answers.

1.2 Program #1: Inspecting the Data [5pts]

Write a program `inspection.{py|java|cpp}` to calculate the label entropy at the root (i.e. the entropy of the labels before any splits) and the error rate (the percent of incorrectly classified instances) of classifying using a majority vote (picking the label with the most examples). You do not need to look at the values of any of the attributes to do these calculations, knowing the labels of each example is sufficient. **Entropy should be calculated in bits using log base 2.**

Command Line Arguments We will run and evaluate the output from the files generated, using the following command:

For Python:	`$ python3 inspection.py <input> <output>`
For Java:	`$ javac inspection.java; java inspect <input> <output>`
For C++:	`$ g++ inspection.cpp; ./a.out <input> <output>`

Your program should accept two command line arguments: an input file and an output file. It should read the `.tsv` input file (of the format described in Section 1.1), compute the quantities above, and write them to the output file so that it contains:

```
entropy: <entropy value>
error: <error value>
```

Example For example, suppose you wanted to inspect the file `small_train.tsv` and write out the results to `small_inspect.txt`. For Python, you would run the command below:

```
$ python3 inspection.py small_train.tsv small_inspect.txt
```

Afterwards, your output file `small_inspect.txt` should contain the following:

```
entropy: 0.996316519559
error: 0.464285714286
```

We will run your program on several input datasets to check that it correctly computes entropy and error, and will take minor differences due to rounding into account. You do not need to round your reported numbers!

For your own records, run your program on each of the datasets provided in the handout—this error rate for a *majority vote* classifier is a baseline over which we would (ideally) like to improve.

1.3 Program #2: Decision Tree Learner [65pts]

In decisionTree.{py | java | cpp}, implement a Decision Tree learner. This file should learn a decision tree with a specified maximum depth, print the decision tree in a specified format, predict the labels of the training and testing examples, and calculate training and testing errors.

Your implementation must satisfy the following requirements:

- Use mutual information to determine which attribute to split on.
- Be sure you're correctly weighting your calculation of mutual information. For a split on attribute X, $I(Y;X) = H(Y) - H(Y|X) = H(Y) - P(X=0)H(Y|X=0) - P(X=1)H(Y|X=1)$.
- As a stopping rule, only split on an attribute if the mutual information is > 0.
- You should split with replacement. That is, when you split on a column, you should retain this column in child datasets.
- Do not grow the tree beyond a max-depth specified on the command line. For example, for a maximum depth of 3, split a node only if the mutual information is > 0 and the current level of the node is < 3.
- Use a majority vote of the labels at each leaf to make classification decisions. If the vote is tied, choose the label that comes ***last*** in the lexicographical order (i.e. Republican should be chosen before Democrat)
- It is possible for attributes to have equal values for mutual information. In this case, you should split on the first attribute to break ties.
- Do not hard-code any aspects of the datasets into your code. We may check your programs on hidden datasets that include different attributes and output labels.

Careful planning will help you to correctly and concisely implement your Decision Tree learner. Here are a few *hints* to get you started:

- Write helper functions to calculate entropy and mutual information.
- Write a function to train a stump (tree with only one level). Then call that function recursively to create the sub-trees.
- In the recursion, keep track of the depth of the current tree so you can stop growing the tree beyond the max-depth.
- Implement a function that takes a learned decision tree and data as inputs, and generates predicted labels. You can write a separate function to calculate the error of the predicted labels with respect to the given (ground-truth) labels.
- Be sure to correctly handle the case where the specified maximum depth is greater than the total number of attributes.
- Be sure to handle the case where max-depth is zero (i.e. a majority vote classifier).
- Look under the FAQ's on Piazza for more useful clarifications about the assignment.

1.4 Command Line Arguments

We will run and evaluate the output from the files generated, using the following command:

For Python:	`$ python3 decisionTree.py [args...]`
For Java:	`$ javac decisionTree.java; java decisionTree [args...]`
For C++:	`$ g++ -g decisionTree.cpp -o decisionTree; ./decisionTree [args...]`

Where above `[args...]` is a placeholder for six command-line arguments: `<train input> <test input> <max depth> <train out> <test out> <metrics out>`. These arguments are described in detail below:

1. `<train input>`: path to the training input `.tsv` file (see Section 1.1)
2. `<test input>`: path to the test input `.tsv` file (see Section 1.1)
3. `<max depth>`: maximum depth to which the tree should be built
4. `<train out>`: path of output `.labels` file to which the predictions on the *training* data should be written (see Section 1.5)
5. `<test out>`: path of output `.labels` file to which the predictions on the *test* data should be written (see Section 1.5)
6. `<metrics out>`: path of the output `.txt` file to which metrics such as train and test error should be written (see Section 1.6)

As an example, if you implemented your program in Python, the following command line would run your program on the politicians dataset and learn a tree with max-depth of two. The train predictions would be written to `pol_2_train.labels`, the test predictions to `pol_2_test.labels`, and the metrics to `pol_2_metrics.txt`.

```
$ python3 decisionTree.py politicians_train.tsv politicians_test.tsv \
        2 pol_2_train.labels pol_2_test.labels pol_2_metrics.txt
```

The following example would run the same learning setup except with max-depth three, and conveniently writing to analogously named output files, so you can can compare the two runs.

```
$ python3 decisionTree.py politicians_train.tsv politicians_test.tsv \
        3 pol_3_train.labels pol_3_test.labels pol_3_metrics.txt
```

1.5 Output: Labels Files

Your program should write two output `.labels` files containing the predictions of your model on training data (`<train out>`) and test data (`<test out>`). Each should contain the predicted labels for each example printed on a new line. Use '\n' to create a new line.

Your labels should exactly match those of a reference decision tree implementation—this will be checked by running your program and evaluating your output file against the reference solution.

Note: You should output your predicted labels using the same string identifiers as the original training data: e.g., for the politicians dataset you should output democrat/republican and for the education dataset you should output A/notA. The first few lines of an example output file is given below for the politician dataset:

```
democrat
democrat
democrat
```

```
republican
democrat
...
```

1.6 Output: Metrics File

Generate another file where you should report the training error and testing error. This file should be written to the path specified by the command line argument `<metrics out>`. Your reported numbers should be within 0.01 of the reference solution. You do not need to round your reported numbers! The file should be formatted as follows:

```
error(train): 0.0714
error(test): 0.1429
```

The values above correspond to the results from training a tree of depth 3 on `small_train.tsv` and testing on `small_test.tsv`. (There is one space between the colon and value)

1.7 Output: Printing the Tree

Finally, you should write a function to pretty-print your learned decision tree. (You may find it more convenient to print the tree *as* you are learning it.) Each row should correspond to a node in the tree. They should be printed in a *depth-first-search* order (but you may print left-to-right or right-to-left, i.e. your answer do not need to have exactly the same order as the reference below). Print the attribute of the node's parent and the attribute value corresponding to the node. Also include the sufficient statistics (i.e. count of positive / negative examples) for the data passed to that node. The row for the root should include *only* those sufficient statistics. A node at depth d, should be prefixed by d copies of the string '| '.

Below, we have provided the recommended format for printing the tree (example for python). You can print it directly to standard out rather than to a file.

```
$ python3 decisionTree.py small_train.tsv small_test.tsv 2 \
small_2_train.labels small_2_test.labels small_2_metrics.txt

[15 democrat/13 republican]
| Anti_satellite_test_ban = y: [13 democrat/1 republican]
| | Export_south_africa = y: [13 democrat/0 republican]
| | Export_south_africa = n: [0 democrat/1 republican]
| Anti_satellite_test_ban = n: [2 democrat/12 republican]
| | Export_south_africa = y: [2 democrat/7 republican]
| | Export_south_africa = n: [0 democrat/5 republican]
```

However, you should be careful that the tree might not be full. For example, after swapping the train/test files in the example above, you could end up with a tree like the following.

```
$ python3 decisionTree.py small_test.tsv small_train.tsv 2 \
swap_2_train.labels swap_2_test.labels swap_2_metrics.txt

[13 democrat/15 republican]
| Anti_satellite_test_ban = y: [9 democrat/0 republican]
| Anti_satellite_test_ban = n: [4 democrat/15 republican]
| | Export_south_africa = y: [4 democrat/10 republican]
| | Export_south_africa = n: [0 democrat/5 republican]
```

The following pretty-print shows the education dataset with max-depth 3. Use this example to check your code before submitting your pretty-print of the politics dataset (asked in question 14 of the Empirical questions).

```
$ python3 decisionTree.py education_train.tsv education_test.tsv 3 \
edu_3_train.labels edu_3_test.labels edu_3_metrics.txt

[135 A/65 notA]
| F = A: [119 A/23 notA]
| | M4 = A: [56 A/2 notA]
| | | P1 = A: [41 A/0 notA]
| | | P1 = notA: [15 A/2 notA]
| | M4 = notA: [63 A/21 notA]
| | | M2 = A: [37 A/3 notA]
| | | M2 = notA: [26 A/18 notA]
| F = notA: [16 A/42 notA]
| | M2 = A: [13 A/15 notA]
| | | M4 = A: [6 A/1 notA]
| | | M4 = notA: [7 A/14 notA]
| | M2 = notA: [3 A/27 notA]
| | | M4 = A: [3 A/5 notA]
| | | M4 = notA: [0 A/22 notA]
```

The numbers in brackets give the number of positive and negative labels from the training data in that part of the tree.

1.8 Submission Instructions

Please ensure you have completed the following files for submission.

```
inspection.{py|java|cpp}
decisionTree.{py|java|cpp}
```

Note: Please make sure the programming language that you use is consistent within this assignment (e.g. don't use C++ for inspect and Python for decisionTree).