HOMEWORK 2: DECISION TREES

10-301/10-601 Introduction to Machine Learning (Spring 2022)
http://www.cs.cmu.edu/~mgormley/courses/10601/
OUT: Wednesday, January 26th
DUE: Friday, February 4th
TAs: Zachary, Sami, Yuxin, Junhui, Rita

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. This assignment consists of several parts: In the Written component, you will work through some Information Theory basics in order to "learn" a Decision Tree on paper, and also work through some pseudocode that will help you algorithmically think through the programming assignment. Then in Programming component, you will implement Decision Tree learning, prediction, and evaluation. Using that implementation, you will answer the empirical questions found at the end of the Written component.

START HERE: Instructions

- **Collaboration Policy**: Please read the collaboration policy here: http://www.cs.cmu.edu/~mgormley/courses/10601/syllabus.html
- **Late Submission Policy:** See the late submission policy here: http://www.cs.cmu.edu/~mgormley/courses/10601/syllabus.html
- **Submitting your work:** You will use Gradescope to submit answers to all questions and code. Please follow instructions at the end of this PDF to correctly submit all your code to Gradescope.
  - **Written:** For written problems such as short answer, multiple choice, derivations, proofs, or plots, please use the provided template. Submissions can be handwritten onto the template, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. If your scanned submission misaligns the template, there will be a 5% penalty. Alternatively, submissions can be written in LaTeX. Each derivation/proof should be completed in the boxes provided. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader.
  - **Programming:** You will submit your code for programming questions on the homework to Gradescope (https://gradescope.com). After uploading your code, our grading scripts will autograde your assignment by running your program on a virtual machine (VM). When you are developing, check that the version number of the programming language environment (e.g. Python 3.9.6) and versions of permitted libraries (e.g. `numpy` 1.21.2 and `scipy` 1.7.1) match those used on Gradescope. You have 10 free Gradescope programming submissions. After 10 submissions, you will begin to lose points from your total programming score. We recommend debugging your implementation on your local machine (or the Linux servers) and making sure your code is running correctly first before submitting your code to Gradescope.

6 Programming: (70 points)

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 two tasks (predicting the party of a politician / predicting final grade for high school students) and report your results. You will write two programs: `inspection.py` (Section 6.2) and `decision_tree.py` (Section 6.3). The programs you write will be automatically graded using the Gradescope system. In this homework you have to choose Python as your programming language. Submitting code for more than one language may result in undefined behavior.

6.1 The Tasks and Datasets

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

Starter Code The handout will contain a preexisting `decision_tree.py` file that itself contains some starter code for the assignment. While we do not require that you use the starter code in your final submission, we *heavily* recommend building upon the structure layed out in the starter code.

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`.

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.[2]

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.

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

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

Write a program `inspection.py` 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 The autograder runs and evaluates the output from the files generated, using the following command:

```
$ python inspection.py <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 6.1), 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`. You would run the following command:

```
$ python 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
```

Our autograder 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! The Autograder will automatically incorporate the right tolerance for float comparisons.

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.

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

In `decision_tree.py`, 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.
- 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 different columns to have equal values for mutual information. In this case, you should split on the **first** column to break ties (e.g. if column 0 and column 4 have the same mutual information, use column 0).
- Do not hard-code any aspects of the datasets into your code. We may autograde 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.
- It is best to think of a Decision Tree as a collection of nodes, where nodes are either leaf nodes (where final decisions are made) or interior nodes (where we split on attributes). It is helpful to design a function to train a single node (i.e. a depth-0 tree), and then recursively call that function to create 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.

6.4 Command Line Arguments

The autograder runs and evaluates the output from the files generated, using the following command:

```
$ python decision_tree.py [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 6.1)
2. `<test input>`: path to the test input `.tsv` file (see Section 6.1)
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 6.5)
5. `<test out>`: path of output `.labels` file to which the predictions on the *test* data should be written (see Section 6.5)
6. `<metrics out>`: path of the output `.txt` file to which metrics such as train and test error should be written (see Section 6.6)

As an example, 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`.

```
$ python decision_tree.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.

```
$ python decision_tree.py politicians_train.tsv politicians_test.tsv \
        3 pol_3_train.labels pol_3_test.labels pol_3_metrics.txt
```

6.5 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 the autograder 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
...
```

6.6 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 Autograder will automatically incorporate the right tolerance for float comparisons. 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)

6.7 Output: Printing the Tree

Finally, you should write a function to pretty-print your learned decision tree. **Your function should print your tree only *after* you are done generating the fully-trained tree.** Each row should correspond to a node in the tree. They should be printed using a *Pre-Order depth-first-search* traversal (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. **This functionality of your program will not be autograded**.

```
$ python decision_tree.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.

```
$ python decision_tree.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).

```
$ python decision_tree.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.

At this point, you should be able to go back and answer questions 1-7 in the "Empirical Questions" section of this handout. Write your solutions in the template provided.

6.8 Submission Instructions

Programming Please ensure you have completed the following files for submission.

```
inspection.py
decision_tree.py
```

When submitting your solution, make sure to select and upload both files. Ensure the files have the exact same spelling and letter casing as above. You can either directly zip the two files (by selecting the two files and compressing them - do not compress the folder containing the files) or directly drag them to Gradescope for submission.

Written Questions Make sure you have completed all questions from Written component (including the collaboration policy questions) in the template provided. When you have done so, please submit your document in **pdf format** to the corresponding assignment slot on Gradescope.