CS 457/557: Assignment 2

Due: October 27, 2022, by 11:00 PM (Central)

Overview

In this assignment, you will write a program that builds decision trees and evaluates their performance.

Academic Integrity Policy

All work for this assignment must be your own work and it must be completed **independently**. Using code from other students and/or online sources (e.g., Github) constitutes academic misconduct, as does sharing your code with others either directly or indirectly (e.g., by posting it online). Academic misconduct is a violation of the UWL Student Honor Code and is unacceptable. Plagiarism or cheating in any form may result in a zero on this assignment, a **negative score** on this assignment, failure of the course, and/or additional sanctions. Refer to the course syllabus for additional details on academic misconduct.

You should be able to complete the assignment using only the course notes and textbook along with relevant programming language documentation (e.g., the Java API specification). Use of additional resources is discouraged but not prohibited, provided that this is limited to high-level queries and not assignment-specific concepts. As a concrete example, searching for "how to use a HashMap in Java" is fine, but searching for "decision tree learning in Java" **is not**.

Deliverables

You should submit a single compressed archive (either `.zip` or `.tgz` format) containing the following to Canvas:

1. The complete source code for your program. I prefer that you use Java in your implementation, but if you would like to use another language, check with me before you get started. Additional source code requirements are listed below:
   - **Your name must be included in a header comment at the top of each source code file.**
   - Your code should follow proper software engineering principles for the chosen language, including meaningful comments and appropriate code style.
   - Your code must not make use of any non-standard or third-party libraries.
2. A `README` text file that provides instructions for how your program can be compiled (as needed) and run from the command line. **If your program is incomplete, then your `README` should document what parts of the program are and are not working.**

Program Requirements

The goal of this assignment is to construct **learning curves** for the decision tree classifier on a given data set. A learning curve plots a classifier's accuracy scores on both a training set and a validation set as a function of the size of the training set. As the classifier is given more data for training, it should generally perform better on the out-of-sample validation data. Your program should generate the raw data that is needed to plot a learning curve but it does not need to actually create a plot. To generate this data, your program will need to implement the decision tree learning algorithm **from scratch** (so you are **not allowed** to make use of any existing libraries for decision trees).

The general outline of the program is given below:

Algorithm 1 Program Flow

```
Process command-line arguments
Load full data set from file
for each training group size do
    for each trial do
        Create a training set by taking a random subset of appropriate size from the full data
        Fit a decision tree on the training set
        Compute the accuracy of the fitted tree on both the training and validation sets (the
        validation set is all data not used for training)
    Report summary statistics across trials
```

The outer loop is governed by three key program parameters: b, i, and l. Your program should generate training groups of size $b, b+i, b+2i, b+3i, \ldots, b+ki$, where k is the largest integer such that $b+ki \leq \min(l, n-1)$, with n being the size of the full data set. The inner loop is governed by the program parameter t, with controls the number of trials. Further details are given below.

Program Execution

The program should be runnable from the command line, and it should be able to process command-line arguments to update the program parameters as needed. The supported options include the following:

- `-f <FILENAME>`: Reads data from the file named `<FILENAME>` (specified as a `String`)
- `-b <INTEGER>`: Specify the base training group size; default is `10`
- `-i <INTEGER>`: Specify the increment in training group size; default is `10`
- `-l <INTEGER>`: Specify the limit for training group size; default is `100`
- `-t <INTEGER>`: Sets the number of trials to perform at each training group size; default is `20`
- `-d <INTEGER>`: Specifies the maximum depth limit to be used when building the decision tree; if unspecified, then no limit is used
  (*Note:* The root node is at depth 0, with child nodes at depth 1, and so on. This means that `-d 1` allows for splitting the root but not any of its children.)

Extra credit: Any program that includes support for the following option will earn a small amount of extra credit (approximately 3–5% of the total assignment grade):

- `-v <INTEGER>`: Specifies a verbosity level, indicating how much output the program should produce; default is `1` (see the **Output** section for details)

CS 557 students must also include support for the following options:

- `-s <INTEGER>`: Specifies the maximum number of splits to be used when building the decision tree; if unspecified, then no limit is used
- `-p`: Toggles printing of the decision tree built in the last trial for the last training group size

CS 457 students **may choose** to implement the `-p` option for a small amount of extra credit (approximately 3–5% of the total assignment grade). More details for these last two options can be found on page 8.

The `-f <FILENAME>` option is required; all others are optional. Filenames may include path information, so do not assume that the files are located in the same directory as your source code. You can assume that your program will only be run with valid arguments (so you do not need to include error checking, though it may be helpful for your own testing). Your program must be able to handle command-line arguments in **any** order (e.g., do **not** assume that the first argument will be `-f`). Several example runs of the program are shown near the end of this document.

File Format

The data for your program will be specified in a file containing header information about attributes (all of which are **qualitative**, or discrete-valued) and output classes, followed by lines corresponding to individual examples (data points).

Each line in the file will either be a comment, which is indicated by a `#` character at the start of the line, a blank line, a header line, or a record line. All header lines appear prior to any record lines. The first header line contains a single number indicating the number of attributes in the data set, which we will denote with p.

Each of the next p header lines gives an attribute name (which may consist of multiple words) followed by a colon (`:`) followed by a list of potential values for that attribute. Each value is represented by a single alphanumeric character, optionally followed by `=` with a longer name given by a sequence of one or more consecutive alphanumeric characters. Values are separated by spaces.

```
# An attribute with values {0,1,2,3,4} and descriptive value names
education level: 0=None 1=HighSchool 2=Bachelors 3=Masters 4=Doctorate

# Another attribute with values {b,c,x,f,k,s} lacking descriptive value names
cap shape: b c x f k s
```

The last header line begins with a colon (`:`) followed by a list of output values (i.e., classes). These output values have the same format as attribute values. Two examples are shown below:

```
# Output Classes (spam prediction)
: 0=Ham 1=Spam
```

```
# Output Classes (generic multi-class classification)
: 1 2 3
```

The remaining non-blank and non-comment lines in the file are record lines. Each record line contains the attribute (feature) value(s) and output (target) value of one data point, with values separated by spaces.

The file contents for a minimal data set are shown below:

```
# Spam prediction
3
external origin: y n
time of day: m=morning a=afternoon e=evening n=night
number of typos: 0 1 2 3 m=many
: 0=Ham 1=Spam

y n 3 1
y m 2 1
n a m 0
y e 0 1
```

Several example input files are provided in the `a02-data.zip` archive on Canvas.

Training Set Formation

For any specified training set size m, each trial for that size should use a training set constructed by selecting m records from the full data set at random (uniformly and without replacement, meaning no record should be selected twice in one trial). The remaining (unselected) records will form the validation set for that trial.

Fitting the Training Data

For any selected training set, your program should build a decision tree following the pseudocode outlined in class (see Lecture 05-2, page 14). Some additional modifications will be necessary to support the depth limit (`-d`) option. Additionally, pay attention to efficiency considerations (both in terms of time and space) in your own implementation. The IMPORTANCE() function should rate each available feature using the information gain heuristic described in class (see Lecture 05-2 for details).

Making Predictions

After fitting (building) the tree, your program should use the tree to make predictions for the training and validation sets and record the accuracy on each set. Note that the training set accuracy should be 100% in most if not all cases because of the tendency of decision trees to overfit the training data unless otherwise restricted (e.g., by using a depth limit).

Output

The level of output produced by the program is controlled by the `-v` flag (for *verbosity*). The levels range from 1 to 4, with 4 containing the most output; the output produced is additive, meaning that the output for level i should also be produced for any level j with $j \geq i$. Output level 1 is the default and is **required**, while output levels 2–4 are optional for **extra credit** (though strongly recommended for debugging purposes). The desired output for each level is discussed in more detail below.

Output Level 1 (default)

Level 1 output prints the average training set accuracy and average validation set accuracy across all trials, for each training group size that is considered by the program. Accuracy should be reported in terms of the proportion of records for which the prediction is correct (to 6 decimal places, as seen in the example output). The example below shows default behavior of the program with only the `-f` flag specified:

```
shell$ java Driver -f mushroom_data.txt
----------------------------------
* Using training group of size 10
  * Average accuracy across 20 trials:
    Training and validation accuracy:      1.000000      0.901864

----------------------------------
* Using training group of size 20
  * Average accuracy across 20 trials:
    Training and validation accuracy:      1.000000      0.945306

----------------------------------
* Using training group of size 30
  * Average accuracy across 20 trials:
    Training and validation accuracy:      1.000000      0.960955

... OUTPUT OMITTED HERE FOR BREVITY ...

----------------------------------
* Using training group of size 100
  * Average accuracy across 20 trials:
    Training and validation accuracy:      1.000000      0.984993
```

The example below uses some of the optional command-line arguments to modify the training group sizes that are considered:

```
shell$ java Driver -f mushroom_data.txt -b 50 -i 100 -l 300 -t 10
----------------------------------
* Using training group of size 50
  * Average accuracy across 10 trials:
    Training and validation accuracy:      1.000000      0.969896

----------------------------------
* Using training group of size 150
  * Average accuracy across 10 trials:
    Training and validation accuracy:      1.000000      0.989443

----------------------------------
* Using training group of size 250
  * Average accuracy across 10 trials:
    Training and validation accuracy:      1.000000      0.994216
```

Output Level 2 (extra credit)

At output level 2, the program also prints accuracy information within each trial. Two examples are shown below:

```
shell$ java Driver -f titanic-data.txt -b 100 -l 100 -t 3 -v 2
----------------------------------
* Using training group of size 100
  * Trial 1:
    Training and validation accuracy:      0.960000      0.739570

  * Trial 2:
    Training and validation accuracy:      0.970000      0.742099

  * Trial 3:
    Training and validation accuracy:      0.950000      0.753477

  * Average accuracy across 3 trials:
    Training and validation accuracy:      0.960000      0.745048
```

Using the Titanic data with a depth limit of 3:

```
shell$ java Driver -f titanic-data.txt -b 200 -l 200 -t 3 -v 2 -d 3
----------------------------------
* Using training group of size 200
  * Trial 1:
    Training and validation accuracy:      0.855000      0.772793

  * Trial 2:
    Training and validation accuracy:      0.845000      0.795948

  * Trial 3:
    Training and validation accuracy:      0.900000      0.751085

  * Average accuracy across 3 trials:
    Training and validation accuracy:      0.866667      0.773275
```

Output Level 3 (extra credit)

Output level 3 adds information about the decision tree learning process and the nodes being examined. Each node that is examined should have its status reported as being either: ineligible for further splitting due to being a pure node (i.e., consisting of one class), having no attributes left for splitting, or being at the max depth; or eligible for splitting. An example is shown below:

```
shell$ java Driver -f titanic-data.txt -b 100 -l 100 -t 1 -d 2 -v 3
----------------------------------
* Using training group of size 100
  * Trial 1:
    * Beginning decision tree learning
      Examining node 0 (depth=0): node is splittable
      Examining node 1 (depth=1): node is splittable
      Examining node 2 (depth=2): node is pure
      Examining node 3 (depth=2): node is pure
      Examining node 4 (depth=2): node is at max depth
      Examining node 5 (depth=1): node is splittable
      Examining node 6 (depth=2): node is at max depth
      Examining node 7 (depth=2): node is pure
      Examining node 8 (depth=2): node is at max depth
      Examining node 9 (depth=2): node is pure
      Examining node 10 (depth=2): node is pure
      Examining node 11 (depth=2): node is at max depth
      Examining node 12 (depth=2): node is at max depth
      Examining node 13 (depth=2): node is at max depth
    * Learned tree has 14 nodes.
    Training and validation accuracy:      0.820000      0.782554

  * Average accuracy across 1 trials:
    Training and validation accuracy:      0.820000      0.782554
```

Output Level 4 (extra credit)

Output level 4 includes the information gain scores for all candidate attributes that are considered for splitting at a splittable node. An example is shown below:

```
shell$ java Driver -f titanic-data.txt -b 100 -l 100 -t 1 -d 2 -v 4
----------------------------------
* Using training group of size 100
  * Trial 1:
    * Beginning decision tree learning
      Examining node 0 (depth=0): node is splittable
        Gain=0.1448 with split on [class]
        Gain=0.0454 with split on [age category]
        Gain=0.1941 with split on [sex]
        Gain=0.0815 with split on [embarked]
        Gain=0.1436 with split on [fare category]
        Gain=0.0322 with split on [siblings and spouses]
        Gain=0.0892 with split on [parents and children]
      Examining node 1 (depth=1): node is splittable
        Gain=0.3789 with split on [class]
        Gain=0.0998 with split on [age category]
        Gain=0.2462 with split on [embarked]
        Gain=0.2726 with split on [fare category]
        Gain=0.0493 with split on [siblings and spouses]
        Gain=0.0671 with split on [parents and children]
      Examining node 2 (depth=2): node is pure
      Examining node 3 (depth=2): node is pure
      Examining node 4 (depth=2): node is at max depth
      Examining node 5 (depth=1): node is splittable
        Gain=0.1517 with split on [class]
        Gain=0.0587 with split on [age category]
        Gain=0.0447 with split on [embarked]
        Gain=0.1637 with split on [fare category]
        Gain=0.0495 with split on [siblings and spouses]
        Gain=0.0957 with split on [parents and children]
      Examining node 6 (depth=2): node is at max depth
      Examining node 7 (depth=2): node is pure
      Examining node 8 (depth=2): node is at max depth
      Examining node 9 (depth=2): node is pure
      Examining node 10 (depth=2): node is pure
      Examining node 11 (depth=2): node is at max depth
      Examining node 12 (depth=2): node is at max depth
      Examining node 13 (depth=2): node is at max depth
    * Learned tree has 14 nodes.
    Training and validation accuracy:      0.820000      0.782554

  * Average accuracy across 1 trials:
    Training and validation accuracy:      0.820000      0.782554
```

Additional Work for CS 557 Students

As noted earlier, CS 557 students must include support for two additional command-line arguments: `-s <INTEGER>` limits the number of splits that can be used when building the decision tree, and `-p` toggles printing of the final generated tree (i.e., the tree that was built in the last trial with the largest training set size) after all other output is displayed.

Limiting the Number of Splits

To support the `-s` option, you will need to modify the tree search process. The pseudocode from Lecture 05-2 (page 14) uses recursive splitting which performs a **depth-first search** (DFS) traversal of the nodes in the tree. This means that all nodes in the "left" subtree of the root node will be split (or marked as leaves) before any nodes in the "right" subtree are considered (we are assuming a binary split at the root for simplicity). However, if we are only allowed a fixed number of splits, then DFS traversal can lead to us "wasting" some of our splits in the left subtree when such splits might be better spent in the right subtree.

To address this, the tree construction process will need to utilize a **best-first search** (BFS) process. During tree construction, BFS maintains a list of splittable nodes called the **open nodes list**, or **frontier**, which is sorted by priority in which these nodes should be split. At each iteration, BFS will remove the first (highest-priority) node from this list and split it. This split will generate child nodes; BFS will examine each of these child nodes to determine if it is splittable and if so, what the best split at that child is. However, rather than split a splittable child immediately, BFS will store the splittable child on the **open nodes list** for splitting later on. By keeping the nodes on the open nodes list sorted according to the information gain scores that result from applying the best possible splits at those nodes, BFS ensures that we always do good splits before poor ones.

The pseudocode for decision tree learning from Lecture 05-2 (page 14) can be restructured to allow for a BFS construction process. The biggest change is to separate the SPLIT() procedure into two procedures: one for *examining* a node to see if it can be split, and if so, what attribute should be used, and another for actually doing the splitting and generating children (which are *examined* upon generation but not immediately split themselves).

Implementing the `-s` option will lead to slightly different output at verbosity levels 3 and 4 compared to what was shown earlier (particularly in terms of the order in which nodes are split). This is fine though (with some additional work, you can allow for DFS-like construction if the `-s` option is not specified, but this is not necessary). **Stop by office hours** if you have more questions on how to implement the `-s` option!

Printing the Tree

When the `-p` flag is provided, your program should print a text representation of the final tree that got built (this output should be disabled by default).

The output of the tree should contain information about the features used for splitting at each internal node (referenced by feature name) along with the value of the feature used on each branch (referenced by the single-letter code). Leaves of the tree should be labeled based on the tree's predicted value, and the branching structure of the tree should be evident. Some examples of what this might look like are shown below, though other formats that meet the above requirements are also permitted.

```
shell$ java Driver -f titanic-data.txt -b 100 -l 100 -t 1 -v 3 -s 1 -p
----------------------------------
```

```
* Using training group of size 100
  * Trial 1:
    * Beginning decision tree learning
      Examining node 0 (depth=0): node is splittable
      Examining node 1 (depth=1): node is splittable
      Examining node 2 (depth=1): node is splittable
    * Learned tree has 3 nodes.
    Training and validation accuracy:      0.780000      0.787611

  * Average accuracy across 1 trials:
    Training and validation accuracy:      0.780000      0.787611

----------------------------------
* Final decision tree:
Node: Split on [sex]
  Branch [sex]=[f]
    Leaf: Predict [Survived]
  Branch [sex]=[m]
    Leaf: Predict [Died]
```

```
shell$ java Driver -f titanic-data.txt -b 100 -l 100 -t 1 -v 3 -s 3 -p
----------------------------------
* Using training group of size 100
  * Trial 1:
    * Beginning decision tree learning
      Examining node 0 (depth=0): node is splittable
      Examining node 1 (depth=1): node is splittable
      Examining node 2 (depth=1): node is splittable
      Examining node 3 (depth=2): node is pure
      Examining node 4 (depth=2): node is pure
      Examining node 5 (depth=2): node is splittable
      Examining node 6 (depth=3): node is splittable
      Examining node 7 (depth=3): node is pure
      Examining node 8 (depth=3): node is pure
      Examining node 9 (depth=3): node is pure
      Examining node 10 (depth=3): node is pure
    * Learned tree has 11 nodes.
    Training and validation accuracy:      0.840000      0.771176

  * Average accuracy across 1 trials:
    Training and validation accuracy:      0.840000      0.771176

----------------------------------
* Final decision tree:
Node: Split on [sex]
  Branch [sex]=[f]
    Node: Split on [class]
      Branch [class]=[1]
        Leaf: Predict [Survived]
      Branch [class]=[2]
        Leaf: Predict [Survived]
      Branch [class]=[3]
        Node: Split on [fare category]
          Branch [fare category]=[under10]
          Leaf: Predict [Died]
        Branch [fare category]=[10to20]
          Leaf: Predict [Died]
        Branch [fare category]=[20to30]
          Leaf: Predict [Survived]
        Branch [fare category]=[40to50]
          Leaf: Predict [Died]
        Branch [fare category]=[50to60]
          Leaf: Predict [Died]
Branch [sex]=[m]
  Leaf: Predict [Died]
```