CS760 Spring 2019 Homework 4

Due April 9 at 11:59pm

General instructions:

- Homeworks are to be done individually.
- For any written problems:
  - We encourage you to typeset your homework in LaTeX. Scanned handwritten submissions will be accepted, but will lose points if they're illegible.
  - Your name and email must be written somewhere near the top of your submission (e.g., in the space provided below).
  - **Show all your work**, including derivations.
- For any programming problems:
  - All programming for CS760, Spring 2019 will be done in Python 3.6. See the `housekeeping/python-setup.pdf` document on Canvas for more information.
  - Follow all directions precisely as given in the problem statement.
- You will typically submit a zipped directory to Canvas, containing all of your work. The assignment may give additional instructions regarding submission format.

Goals of this assignment: to become familiar with two widely-used ensemble methods: bagging and AdaBoost. You will do this by:

- Implementing both bagging and AdaBoost, and using them to create ensembles of decision tree classifiers.
- Performing some simple evaluations for your ensemble classifiers:
  - generating confusion matrices;
  - plotting the performance of your ensembles, as a function of ensemble size.

Programming Problems

Things to Know Before You Start

Dataset Format. You can assume that all datasets will be provided in JSON files, structured like this example:

```
{
    'metadata': {
                    'features': [ ['feature1', 'numeric'],
                                  ['feature2', ['cat', 'dog', 'fish'],
                                  ...
                                  ['class', ['+', '-']
                                ]

                },

    'data':    [[ 3.14, 'dog', ... , '+' ],
                [ <instance 2> ],
                ...
                [ <instance N> ]]
}
```

That is, the file contains *metadata* and *data*. The metadata tells you the names of the features, and their types.

Real and integer-valued features are identified by the 'numeric' token. Categorical features are identified by a list of the values they may take.

The data is an array of feature vectors. The order of features in the metadata matches their order in the feature vectors.

JSON files are easy to work with in Python. You will find the `json` package (and specifically the `json.load` function) useful.

Decision Tree Implementation. We have provided you with a decision tree implementation in the `DecisionTree.py` file. You will construct your ensembles using instances of this `DecisionTree` class. You do **not** need to modify it during this assignment.

The interface is straightforward:

- `DecisionTree()` constructs an empty, untrained decision tree.
- `fit(X, y, metadata, instance_weights=None)` trains the decision tree on training set `X`, `y`. The `instance_weights` parameter allows you to assign differing weights to the training examples (this is important for AdaBoost).

- `predict(X, prob=False)` uses the decision tree to predict classes for X. Setting `prob=True` causes it to return class probabilities, rather than a single class label (this is important for bagging).

Example usage of the `DecisionTree` class:

```
# Import modules
import DecisionTree as dt
import json
import numpy as np

# Get training data
train = json.load(open('a_training_set.json', 'r'))
train_meta = train['metadata']['features']
train_data = np.array(train['data'])
train_X = train_data[:,:-1]
train_y = train_data[:,-1]

# Build and train a decision tree:
mytree = dt.DecisionTree()
mytree.fit(train_X, train_y, train_meta, max_depth=5)

# look at the structure of the trained tree:
print(mytree)

# Get test data
test = json.load(open('a_test_set.json', 'r'))
test_data = np.array(test['data'])
test_X = test_data[:,:-1]
test_y = test_data[:,-1]

# Predict the test labels:
predicted_y = mytree.predict(test_X, prob=True)
```

If you have questions about usage, **please** read the comments and docstrings in `DecisionTree.py` before posting to Piazza.

Part 1: Bootstrap Aggregation (Bagging) (20 pts)

Implement a bagged decision tree classifier.

For this part, keep the following in mind:

- In the JSON files that we provide, the class attribute is named `'class'` and it is the last attribute listed in the feature section.
- Numeric features need not be standardized for decision tree classification.

Command Line Signature. Your program should be called `bagged_trees`, and must be callable from a bash terminal as follows:

```
$ ./bagged_trees <#trees> <max depth> <train-set-file> <test-set-file>
```

That is,

- you should have an executable script called `bagged_trees`;
- the 1st argument specifies the number of trees in the ensemble;
- the 2nd argument specifies the maximum depth of the trees;
- the 3rd argument is the path to a training set file;
- and the 4th argument is the path to a test set file.

You *must* have this call signature—otherwise, the autograder will not be able to analyze your implementation correctly.

Output Format. Suppose you run `bagged_trees` with T trees; on a dataset with M features and K class values. Suppose also that the training set contains N_{train} examples, and the test set contains N_{test} examples.

Then your program should print the following information to stdout:

- A $N_{train} \times T$ table containing the indices of your bootstrapped samples. In other words: the tth tree in your ensemble is trained on some random resampling of the original training set; the indices of those random training instances should be printed in the tth column of the table.

  The table should be delimited by **commas**, with no spaces.

- An empty line—no spaces or tabs. Just a newline (`\n`).
- A $N_{test} \times (T + 2)$ table of *predictions*. The ith row of this table contains the T individual trees' predictions for test example i; followed by their *combined* prediction; followed by the true class label.

  The table should be delimited by **commas**, with no spaces.

- An empty line—no spaces or tabs. Just a newline (`\n`).
- The test set accuracy of the classifier. Your floating point output must match ours within a tolerance of 1e-9.

For example: the output for `bagged_trees` with 3 trees might look like this...

```
123,456,789
1,42,999
...
666,7,11

cat,cat,dog,cat,cat
cat,dog,dog,dog,dog
...
cat,dog,dog,cat,cat

0.789012345678901234
```

See the reference output for real examples.

Additional Implementation Details. You must follow these requirements—otherwise, your output may not match ours.

1. To ensure your program is deterministic, you must call `numpy.random.seed(0)` at the beginning of your program. For more details about this function, you can read the online documentation here.

   Here is an example usage:

```
import numpy as np
import sys

if __name__ == "__main__":      # the entrance of your program
    np.random.seed(0)
    main(sys.argv)              # your code starts here
```

2. To ensure that we all get the same output, use numpy's `random.choice(...)` function to generate your bootstrapped training sets. See the online documentation for more information about usage.

   Recall that in bagging, the bootstrapped training sets are resampled *with* replacement, and are the same size as the original training set.

3. In order to correctly combine the individual trees' predictions, do the following:

   - Compute the individual trees' *probabilistic* predictions.
   - Find the mode (argmax) of the average of the probabilistic predictions.

   This is different from simply taking the argmax of the individual trees' modal predictions. It accounts for the individual trees' uncertainty.

Part 2: Adaptive Boosting (AdaBoost) (50 pts)

Use Multi-class AdaBoost to generate an ensemble of decision trees.

Command Line Signature. Your program should be called `boosted_trees`, and must be callable from a bash terminal as follows:

```
$ ./boosted_trees <max trees> <max depth> <train-set-file> <test-set-file>
```

That is,

- you ought to have an executable script called `boosted_trees`;
- the 1st argument specifies the maximum number of trees in the ensemble.
  (Note that AdaBoost may produce fewer trees if it breaks out of its loop.)
- the 2nd argument specifies the maximum depth of the trees;
- the 3rd argument is the path to a training set file;
- and the 4th argument is the path to a test set file.

Multi-class AdaBoost. The AdaBoost algorithm described in the lecture notes (slide 10 of `ensembles.pdf`) is specifically for *binary* classification.

We will implement a simple generalization of AdaBoost that works for arbitrary numbers of classes.

For detailed pseudocode, see "Algorithm 2" on page 4 of this paper:
Hastie et al., 2006. *Multi-class AdaBoost.*

The only real differences between AdaBoost in the lecture notes and Multi-class AdaBoost are:

- the calculation of β (or, in the notation of Hastie's paper, α).
- instead of breaking out of the loop if $\epsilon \geq 0.5$, we break out of the loop if $\epsilon \geq 1 - \frac{1}{K}$, where K is the number of classes.

Notice that this reduces to two-class AdaBoost when $K = 2$.

Output Format. The output format for `boosted_trees` will be similar to that of `bagged_trees`, but with some changes.

As before: suppose you run `boosted_trees` with T trees; on a dataset with M features and K class values. Suppose also that the training set contains N_{train} examples, and the test set contains N_{test} examples.

Then your program should print the following information to stdout:

- A $N_{train} \times T$ table containing *training instance weights*. In other words: the tth tree in your ensemble is trained on a weighted training set; the weights of those training instances should be printed in the tth column of the table, in the same order as their corresponding training instances.

  The table should be delimited by **commas**, with no spaces.

- An empty line—no spaces or tabs. Just a newline (`\n`).
- A single row containing the *tree weights*—the lecture notes call them β, the Hastie paper calls them α.

  They must be printed in the order that their trees are generated, and they must be delimited by **commas**.

- An empty line—no spaces or tabs. Just a newline (`\n`).
- A $N_{test} \times (T + 2)$ table of *predictions*. The ith row of this table contains the T individual trees' predictions for test example i; followed by their *combined* prediction; followed by their true class label.

  The table should be delimited by **commas**, with no spaces.

  This is the same format as in `bagged_trees`.

- An empty line—no spaces or tabs. Just a newline (`\n`).
- The test set accuracy of the classifier.

For example: the output for `boosted_trees` with 3 trees might look like this...

```
0.001000000000,0.001234567890,0.00246913601
0.001000000000,0.001234567890,0.00098765432
...
0.001000000000,0.000987654321,0.00197530901

1.234567890123,1.456789012345,0.90123445678

cat,cat,dog,cat,cat
cat,dog,dog,dog,dog
...
dog,cat,dog,dog,cat

0.789012345678901234
```

See the reference output for real examples.

As usual: all floating point output must match reference within a tolerance of 1e-9.

Additional Implementation Details.

- Notice that the `DecisionTree` implementation allows you to provide a vector of weights for the training set instances. Please use this in your AdaBoost implementation.
- Since our decision trees can use weights in a deterministic fashion, your boosted tree learner should be fully deterministic. You shouldn't need to use a random number generator anywhere in your AdaBoost implementation.

- Notice that the AdaBoost algorithm (both binary and multi-class) combines the individual trees' *modal* (i.e, argmax) predictions to produce an ensemble prediction. There are ways to combine the individual trees' *probabilistic* predictions[1]—however they're a little bit more complicated and you are not required to implement any of them in this assignment.

Take some time to play with this—it's quite remarkable how boosting can combine very weak classifiers (slightly better than random) to make a strong one.

Part 3: Evaluation (30 pts)

Confusion Matrices (15 pts). Write a script that generates the confusion matrix for an ensemble on a given dataset. It should be called `confusion_matrix`, and have the following command line signature:

```
$ ./confusion_matrix <bag|boost> <# trees> <max tree depth> <train-set-file> <test-set-file>
```

The first argument is a string: either `bag` or `boost`. It indicates whether we're making a confusion matrix for a bagged-trees or boosted-trees classifier.

The other arguments are identical to those of the `bagged_trees` and `boosted_trees` scripts.

The script should train the indicated classifier with the given parameters, on the given training set. Then it should generate the confusion matrix from the test set, and print it to standard output in the following format:

- rows correspond to *predicted class*;
- columns correspond to *actual class*;
- entries are integers;
- the table is delimited by **commas**;
- the ordering of rows and columns should correspond to the list of class values in the metadata.

For example, the output for a 3-class problem with 100 test instances might look like this:

```
20,4,5
12,23,7
8,2,19
```

Visualizing Ensemble Performance (15 pts). Plot the test set accuracy of (1) bagged trees and (2) boosted trees as a function of ensemble size (i.e., number of trees). Requirements:

- You must produce **two separate PDF files:** (1) `bagged_tree_plot.pdf` for bagged trees, and (2) `boosted_tree_plot.pdf` for boosted trees.
- You may use any of the provided datasets to do this.
- For each plot, plot 2 or 3 lines on the same axes, corresponding to different settings of the max-tree-depth parameter.
- In order to plot some interesting behaviors, recall the following:
  1. Bagging is a useful way to combine classifiers with *high variance*. (What do "high-variance" decision trees look like?)
  2. AdaBoost is an excellent way to combine *weak* classifiers. (What does a "weak" decision tree look like?)

You may find the `digits` dataset especially useful for experimentation.

[1] For example, see Algorithm 4 in the Hastie paper.

Additional Notes

Submission instructions

Organize your submission in a directory with the following structure:

```
YOUR_NETID/
    # your scripts
    bagged_trees
    boosted_trees
    confusion_matrix

    # plots and discussion
    bagged_tree_plot.pdf
    boosted_tree_plot.pdf

    # your source files
    <your various *.py files>
    DecisionTree.py
```

Zip your directory (`YOUR_NETID.zip`) and submit it to Canvas.

The autograder will unzip your directory, `chmod u+x` your scripts, and run them on several datasets. Their results will be compared against our own reference implementation.

Resources

Executable scripts

We recommend writing your scripts in bash, and having them call your python code.

For example, your script `bagged_trees` might look like this (given your source code named `bagged_trees.py`):

```
#! /bin/bash

python bagged_trees.py $1 $2 $3 $4
```

If this doesn't make sense to you, try reading this tutorial:
`http://matt.might.net/articles/bash-by-example/`

Datasets

We've provided four datasets for you to experiment with. Two are new, and two will be familiar to you:

- `mushrooms*.json`. This is derived from the famous Mushroom dataset. The task is to classify whether mushrooms are poisonous or edible. We removed some features and downsampled the data, since classification was too easy with the original dataset.

  See the UCI repository for more info:
  `https://archive.ics.uci.edu/ml/datasets/Mushroom`

- `heart*.json`. This is the Heart Disease Data Set. The task is to determine the presence of heart disease in a patient.

  See the UCI repository for more info:
  `https://archive.ics.uci.edu/ml/datasets/heart+Disease`

- `winequality*.json`. This is the Wine Quality data set. The task is to predict the (integer-valued) quality of a wine. We treat it as a classification task in this assignment.

  See the UCI repository for more info:
  `https://archive.ics.uci.edu/ml/datasets/Wine+Quality`

- `digits*.json` This is the Digits dataset. The task is to determine the handwritten digits, from pixel intensities.

  See the UCI repository for more info:
  `https://archive.ics.uci.edu/ml/datasets/optical+recognition+of+handwritten+digits`

We will provide reference output for these datasets - you will be able to check your own output against them.

During grading, your code will be tested on these datasets as well as others.