CS 4824 / ECE 4424 HW 1 Programming Portion Due: Feb 20, 2020

General Instructions: Upload only those Python files that you have modified for this assignment, which should include `naive_bayes.py`, `decision_tree.py`, and `crossval.py`. You are welcome to create additional functions, files, or scripts, and you are also welcome to modify the included interfaces for existing functions in the given files if you prefer a different organization.

For this homework, you will build two text categorization classifiers: one using naive Bayes and the other using decision trees. You will write general code for cross-validation that will apply to either of your classifiers.

Data and starter code: In the HW1 archive, you should find the 20newsgroups data set (also available from the original source `http://qwone.com/~jason/20Newsgroups/`). This data set consists of newsgroup posts from an earlier era of the Internet. The posts are in different categories, and this data set has become a standard benchmark for text classification methods.

The data is represented in a bag-of-words format, where each post is represented by what words are present in it, without any consideration of the order of the words.

We have also provided a unit test class in `tests.py`, which contains unit tests for each type of learning model. These unit tests may be easier to use for debugging in an IDE like PyCharm than the iPython notebook. A successful implementation should pass all unit tests and run through the entire iPython notebook without issues. You can run the unit tests from a *nix command line with the command

```
python -m unittest -v tests
```

or you can use an IDE's unit test interface. These tests are not foolproof, so it's possible for code that does not meet the requirements for full credit to pass the tests (and, though it would be surprising, it may be possible for full credit code to fail the tests).

Before starting all the tasks, examine the entire codebase. Follow the code from the iPython notebook to see which methods it calls. Make sure you understand what all of the code does.

Your required tasks follow.

1. (0 points) Examine the iPython notebook `test_predictors.ipynb`. This notebook uses the learning algorithms and predictors you will implement in the first part of the assignment. Read through the data-loading code and the experiment code to make sure you understand how each piece works.

2. (0 points) Examine the function `calculate_information_gain` in `decision_tree.py`. The function takes in training data and training labels and computes the information gain for each feature. That is, for each feature dimension, X_i, it computes the gain in splitting based on X_i, $G(X_i)$. Your function should return the vector
$$[G(X_1), \ldots, G(X_d)]^\top. \tag{1}$$
You will use this function to do feature selection and as a subroutine for decision tree learning. Note how the function avoids loops over the dataset and only loops over the number of classes. Follow this style to avoid slow Python loops; use `numpy` array operations whenever possible.

3. (5 points) Finish the functions `naive_bayes_train` and `naive_bayes_predict` in `naive_bayes.py`. The training algorithm should return the probability tables for each class conditional probability term and prior probability term. During testing, since there are repeated multiplications of probabilities that can become very small, make sure to use log-space representation for these probabilities, so that instead of taking product, you can take sum over log-probabilities. Also note that you can infer the likely class label without explicitly computing th evidence term, $P(x_i)$.

   Implement Laplace estimate to avoid class conditional probabilities from vanishing to 0. For a binary variable X, this amounts to
$$\Pr(X = True|Y = y) = \frac{(\#\text{ examples of class y where } X = True) + 1}{(\text{Total } \#\text{ of examples of class y}) + 2}.$$

Algorithm 1 Recursive procedure to grow a classification tree

function FITTREE($\mathcal{D}$, depth)
  if not worth splitting (because $\mathcal{D}$ is all one class or max depth is reached) **then**
    node.prediction $\leftarrow$ majority class of $\mathcal{D}$
    return node
  $w \leftarrow \arg\max_w G(X_w)$
  node.test $\leftarrow w$
  node.left $\leftarrow$ FITTREE($\mathcal{D}_L$, depth+1) ▷ where $\mathcal{D}_L := \{(\boldsymbol{x}, y) \in \mathcal{D} | x_w = 0\}$
  node.right $\leftarrow$ FITTREE($\mathcal{D}_R$, depth+1) ▷ where $\mathcal{D}_R := \{(\boldsymbol{x}, y) \in \mathcal{D} | x_w = 1\}$
  return node

4. (5 points) Finish the functions `recursive_tree_train` and `decision_tree_predict` in `decision_tree.py`. Note that `recursive_tree_train` is a helper function used by `decision_tree_train`, which is already completed for you. You'll have to design a way to represent the decision tree in the `model` object. Your training algorithm should take a parameter that is the maximum depth of the decision tree, and the learning algorithm should then greedily grow a tree of that depth. Use the information-gain measure to determine the branches (hint: you're welcome to use the `calculate_information_gain` function). Algorithm 1 is abstract pseudocode describing one way to implement decision tree training. You are welcome to deviate from this somewhat; there are many ways to correctly implement such procedures.

   The pseudocode suggests building a tree data structure that stores in each node either (1) a prediction or (2) a word to split on and child nodes. The pseudocode also includes the formula for the entropy criterion for selecting which word to split on.

   The prediction function should have an analogous recursion, where it receives a data example and a node. If the node has children, the function should determine which child to recursively predict with. If it has no children, it should return the prediction stored at the node.

5. (5 points) Finish the function `cross_validate` in `crossval.py`, which takes a training algorithm, a prediction algorithm, a data set, labels, parameters, and the number of folds as input and performs cross-fold validation using that many folds. For example, calling

```
params['max_depth'] = 16
score = cross_validate(decision_tree_train, decision_tree_predict, train_data,
           train_labels, 10, params)
```

   will compute the 10-fold cross-validation accuracy of decision tree using max depth $= 16$.

   The cross-validation should split the input data set into `folds` subsets. Then iteratively hold out each subset: train a model using all data *except* the subset and evaluate the accuracy on the held-out subset. The function should return the average accuracy over all `folds` splits.

   Some code to manage the indexing of the splits is included. You are welcome to change it if you prefer a different way of organizing the indexing.

   Once you complete this last step, you should be able to run the notebook `cv_predictors.ipynb`, which should use cross validation to compare decision trees to naive Bayes on the 20-newsgroups task. Naive Bayes should be much more accurate than decision trees, but the cross-validation should find a decision tree depth that performs a bit better than the depth hard coded into `test_predictors.ipynb`.