Machine Learning Fall 2017 Homework 1

Homework must be submitted electronically on Canvas. Make sure to explain you reasoning or show your derivations. Except for answers that are especially straightforward, you will lose points for unjustified answers, even if they are correct.

General Instructions

You are allowed to work with at most one other student on the homework. With your partner, you will submit only one copy, and you will share the grade that your submission receives. You should set up your partnership on Canvas as a two-person group.

Submit your homework electronically on Canvas. We recommend using LaTeX, especially for the written problems. But you are welcome to use anything as long as it is neat and readable.

Include a README file that describes all other included files. Indicate which files you modified. You are welcome to create additional functions, files, or scripts, and you are also welcome to modify the included interfaces for existing functions if you prefer a different organization.

Since we may work on some of the homework in class, clearly indicate which parts of your submitted homework are work done in class and which are your own work.

Relatedly, cite all outside sources of information and ideas.

Written Problems

1. (5 points, Based on Murphy 2.2) Suppose a crime has been committed. Blood is found at the scene for which there is no innocent explanation. It is of a type that is present in 1% of the population. A suspect who has this rare blood type has been charged with the crime.

   The prosecutor claims: "There is a 1% chance that the defendant would have the crime blood type if he were innocent. Thus there is a 99% chance that he is guilty." This is known as the **prosecutor's fallacy**. What is wrong with this argument?

2. A Bernoulli distribution has the following likelihood function for a data set $\mathcal{D}$:

$$p(\mathcal{D}|\theta) = \theta^{N_1}(1-\theta)^{N_0}, \tag{1}$$

   where N_1 is the number of instances in data set $\mathcal{D}$ that have value 1 and N_0 is the number in $\mathcal{D}$ that have value 0. The maximum likelihood estimate is

$$\hat{\theta} = \frac{N_1}{N_1 + N_0}. \tag{2}$$

   (a) (5 points) Derive the maximum likelihood estimate above by solving for the maximum of the likelihood. I.e., show the mathematics that get from Equation (1) to Equation (2).

   (b) (5 points) Suppose we now want to maximize a posterior likelihood

$$p(\theta|\mathcal{D}) = \frac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})}, \tag{3}$$

   where we use the Bernoulli likelihood and a (slight variant[1] of a) symmetric Beta prior over the Bernoulli parameter

$$p(\theta) \propto \theta^{\alpha}(1-\theta)^{\alpha}. \tag{4}$$

   Derive the maximum posterior mean estimate.

[1] For convenience, we are using the exponent of α instead of the standard $\alpha - 1$.

Programming Assignment

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, whose origin is somewhat fuzzy, 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.

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, compute

$$\begin{aligned} G(Y, X_j) &= H(Y) - H(Y|X_j) \\ &= -\sum_y \Pr(Y = y) \log \Pr(Y = y) + \\ &\quad \sum_{x_j} \Pr(X_j = x_j) \sum_y \Pr(Y = y | X_j = x_j) \log \Pr(Y = y | X_j = x_j). \end{aligned} \tag{5}$$

   Your function should return the vector

$$[G(Y, X_1), \ldots, G(Y, X_d)]^\top. \tag{6}$$

   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 find the maximum likelihood parameters for the probability distribution

$$\Pr(y_i = c | \boldsymbol{x}_i) = \frac{\Pr(y_i = c) \prod_{w \in W} \Pr(x_{iw} | y_i = c)}{\Pr(x_i)}.$$

   Make sure to use log-space representation for these probabilities, since they will become very small, and notice that you can accomplish the goal of naive Bayes learning without explicitly computing the prior probability $\Pr(x_i)$. In other words, you can predict the most likely class label without explicitly computing that quantity.

   Implement additive smoothing (`https://en.wikipedia.org/wiki/Additive_smoothing`) for your naive Bayes learner. One natural way to do this is to let the input parameter `params` simply be the prior count for each word. For a parameter α, this would mean your maximum likelihood estimates for any Bernoulli variable X would be

$$\Pr(X) = \frac{(\#\text{ examples where } X) + \alpha}{(\text{Total } \#\text{ of examples}) + 2\alpha}.$$

   Notice that if $\alpha = 0$, you get the standard maximum likelihood estimate.

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 $\leftarrow \arg\max_c \sum_{(\boldsymbol{x},y)\in\mathcal{D}} I(y = c)$
        return node
    $w \leftarrow \arg\max_{w'} G(Y, X_w)$                              ▷ See Equation (5)
    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['alpha'] = 1.0
   score = cross_validate(naive_bayes_train, naive_bayes_predict, train_data,
               train_labels, 10, params)
   ```

   will compute the 10-fold cross-validation accuracy of naive Bayes using regularization parameter $\alpha = 1.0$.

   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 do 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`.