420-LCW-MS Programming Techniques and Applications - Assignment 3

April 28, 2022

As always, remember the general requirements for assignments in this course!
Goals for this assignment:

- Experiment with a machine learning algorithm.
- Learn about different measures of the quality of a classifier.

Introduction

In basic machine learning exercises it is common to examine percentage of "correct" predictions made by a classifier. This is easy to compute, in that you can just count the number of times the classifier agrees with the "known" label, and divide this by the total number of items classified.

In practice, this simple measure isn't always enough. For example, in a simple two-class problem, one class may outnumber the other. Suppose our goal is to detect a disease that occurs in only 1% of the population, and our dataset reflects the distribution of measurements and classifications in the overall population. So only 1% of our data will have be labeled as positive for the disease, and the rest will be labeled negative. In this case, a dumb classifier that just outputs the negative prediction (no disease) in every case will be right 99% of the time. This kind of imbalance in the training data is one of the important potential sources of "bias" in our model, in the common sense of the word "bias".

To get a better idea of the performance of the algorithm in a classification problem with C classes, we want to break down our errors into C^2 different "bins". In this exercise we'll only worry about a 2-class problem, so we'll have four bins to fill. The bins record the number of times we observe each of the following situations:

1. True positives (TP) - the classifier predicted a correct positive label.
2. True negatives (TN) - the classifier predicted a correct negative label.
3. False positives (FP) - the classifier predicted an incorrect positive label.
4. False negatives (FN) - the classifier predicted an incorrect negative label.

It is common to arrange these counts in a C by C matrix called the *confusion matrix*:

		Correct	
		0	1
Predicted	0	TN	FN
	1	FP	TP

From these four values, you can compute a number of possible measures of the quality of your classifier. In this exercise you'll just compute a few of the most useful.

The first, the *sensitivity*, also known as both *recall* or *true positive rate*, is computed as:

$$TPR = \frac{TP}{TP + FN}$$

The second, the *specificity*, or *true negative rate* is computed as:

$$TNR = \frac{TN}{TN + FP}$$

The third, the *false positive rate*, is:

$$FPR = \frac{FP}{FP + TN} = 1 - TNR$$

Finally, the *false negative rate* is:

$$FNR = \frac{FN}{FN + TP} = 1 - TPR$$

There are other combinations that can be useful, for more information the Wikipedia page on the topic Confusion matrix is worth reading.

The dataset

For this exercise, you'll use a dataset called "Spambase" that has been used for several years as a benchmark for machine learning algorithms that detect spam email messages. The data was downloaded from the UCI ML repository and is described in the UCI repository page. The datasets consists of 4601 instances of 57 attributes each, with a class label that is either 0 for "good" and 1 for "spam". The attributes are all numeric and reflect various things like word or character counts in the message.

Obviously this is a case where it is quite likely that the number of spam and non-spam emails is unequal, so performing a detailed evaluation of the classifier is important. Arbitrarily, we think of the decision to classify a message as spam as the "positive" classification. Incorrectly classifying good emails as spam may be quite bothersome to users, so we typically want to minimize the false positive rate. However, minimizing the FPR will trade off with the true positive rate (and false negative rate) of the classifier - reducing the FPR will often reduce the TPR and increase the FNR as well, letting spam messages through the spam filter.

The included files

I have included many files you can use as the basis of your work:

- `bagging.py` - A simple implementation of the "bagging" approach to random forests. This implements the `bagging_trees` classifier.
- `classifier.py` - An *abstract class* that implements a generic interface for classification. This also includes some global functions which are useful in classifiers generally.
- `datasets.py` - Some code to read a few of the example labeled datasets.
- `decision_tree.py` - A very basic implementation of a decision tree classifier.
- `extra_trees.py` - An implementation of the "extremely randomized" trees classifier.
- `Geometry.py` - A few simple classes used for visualization and geometric objects.
- `kdtree.py` - An implementation of K-dimensional (K-D) trees for relatively fast nearest-neighbor computation.
- `knnclassifier.py` - Simple k-nearest neighbor classification using K-D trees.
- `perceptron.py` - Implementation of the classic "perceptron", a predecessor of true neural networks.
- `simple_nn.py` - A simple backpropagation neural network. Hopelessly primitive by today's standards, but demonstrating many of the same basic principles still in use.
- `spambase.py` - Code to read the `spambase.data` file.

Your tasks

1. Using the basic code provided in `spambase.py`, add code to split the dataset into training and testing folds using 5-fold *random sub-sampling* cross validation. This is not exactly the same procedure as used in the `classifier.py` function `evaluate`, but you can use that function as a model.

   To do random sub-sampling validation, reshuffle the dataset on every fold, and split it into two pieces, one containing 4/5 of the data (the training set) and the other containing 1/5 of the data (the test set).

   Since there are 4601 items in the dataset, for each fold you should have 920 items in each test set and 3681 items in the training set. Do *not* use these numbers directly in your code, compute them based on the length of the dataset.

2. For each fold, create a classifier. I recommend Extra trees, but feel free to try a different method[1]. Train the classifier on the training set, then compute its predictions for the test set. Remember that all of the classifiers used follow the framework defined in `classifier.py` - they have two public methods, `train()` and `predict()`. Also keep in mind that this is a relatively large dataset, so it will take a while to train and test 5 folds. You can experiment with a smaller number of folds just to get your code running.

[1] If you are super ambitious, try to use a "real" algorithm from scikit-learn, tensorflow, or similar.

3. Compute the overall confusion matrix by combining all results over all of your folds. Since you will have 5 folds with 1/5 of the data included in each test set, the four values in your confusion matrix should add up to 3680 (4 times 920).

4. Compute and print the TPR and FPR.

5. Experiment with different settings for your chosen classifier. For example, with extra trees you can try varying the number of trees `M`, the minimum split size `Nmin`, or the number of tests evaluated per node, `K`. Does changing these values have a big effect on your results? Vary these parameters systematically (pick at least 4 or 5 different values or sets of values) and produce a graph or a table.

6. Use a second classifier and compare results. For time reasons, you might want to try *k*-nearest-neighbor as your comparison. If you do use kNN, report how your results change if you vary *k*.

7. Create an (at most) 1-page document (text, PDF, or Word) that summarizes your results.

What to hand in

Hand in any Python code you create or modify, as well as a brief (at most one page) summary of your results. Combine it all into a ZIP file and upload to Omnivox.