Measuring Performance of a Classification Algorithm

Jugal Kalita

March, 2010

1 Learning Method

Let us assume we are using a machine learning approach for classification. In particular, let us assume that we are using a decision tree learning method to learn to classify unseen data into categories.

The number of categories can be just two, say *yes* or *no*. That is, the data can be classified as belonging to a certain class and not belonging to the same class. An example is classifying mortgage applicants to two categories: *approve for loan* being one category, and *don't approve for loan* being the other category.

Of course, the number of categories can be more than two. Applicants for mortgage loans can be classified into several categories: *preferred or low risk, moderate risk, high risk, don't approve or very high risk*.

2 Data

For decision tree learning, the data must be in tabular form. Each example of data must be labeled as belonging to one single class. The attributes or columns of the table may be categorial or nominal, i.e., an attribute has a small number of values it can take such as *very high, high, medium, low, very low*. Of course, this means, the attributes may have Boolean values. In addition, some other attributes may take numeric values, either continuous or integer. If an attribute is numeric in nature, some people convert it first to categorial form, by dividing the range of values the attribute takes to a small number of intervals and assigning names to those intervals. Others use the numeric values directly.

The considerations above also apply to the result attribute. If the result attribute is numeric, we call the decision tree more appropriately a regression tree.

2.1 Training and Test Sets

For a decision tree learning algorithm to learn, it first has to be shown a number of labeled examples from which it can acquire the necessary knowledge to classify unseen examples.

Suppose we have N labeled examples in the data. It is a good idea to have these examples show the same distribution as the data on which the program will be tested. However, usually it is difficult

to obtain data that truly conforms in its distribution to that of the data on which the program may be tested. This is because the program may be used in circumstances its designers did not expect.

We divide the labeled data we have into two subsets. One of these subsets is the *training set*, and the other is the *test set*.

Usually, from the data available D, a random number of elements are picked to form the training set R. The rest of the data S is used to test the trained program. Thus,

$$D = R \cup S$$

and R and S are disjoint. In other words,

$$S = D - R$$

and

$$R = D - S.$$

Different people choose the relative sizes of R and S differently. But, a usual choice is

$$\mid R \mid = k \times \mid S \mid .$$

In other words, the size of the training set is k times the size of the test set. Usually k is set to a number anywhere from 2 to 9, say. In other words, the size of the training set is 3 to 10 times the size of the test set.

2.1.1 Dealing with Small Amount of Available Data: Cross-validation

The amount of data available to train and test the learning program may be small. In such a situation, the division of the data into training and test sets actually may be of help in the following manner. Suppose, we have only 24 elements in the total data set. We can perform a *3-fold cross-validation*. We divide the data into training set and test set such that

$$\mid R \mid = 2 \times \mid S \mid .$$

In other words, we train the program on 16 randomly chosen elements of the data set and test it on the 8 remaining elements of the test set. The sizes of the two sets, especially, the test set is tiny. One way to ameliorate the situation a little is to repeat the selection of the training and test sets r number of times, each time randomly choosing the training set and the test set elements in the fashion described above. For example, if $r = 10$, we repeat the selection of the training and test sets 10 times, each time training with 16 randomly chosen elements and testing on the 8 elements left. In other words, we run the training program 8 times and run the tests each time also. In this way, we have tested the program on a larger number of elements, in some sense. Critics of this approach would say that the data items may have dependencies among them, and hence this way of cross-validating does not provide for much better results. However, choosing the elements

randomly, hopefully takes away some of the regularities from among the chosen examples. In many situations, this is the best one can do.

In k-fold cross-validation, we divide the labeled data randomly into k subsets. We train on randomly chosen $k-1$ subsets and test on the remaining subset. We repeat this process several times. We report averaged results over all repetitions.

Cross-validating by dividing the data into training and test sets, and running the training and test programs several times by repeating the experiments is still a good idea, even if we have a large amount of data. In such a case, usually we report the average performance and/or the best performance over the repeated experiments.

2.1.2 What if the Labeled Classes Are Unbalanced?

It is possible that in the labeled examples, the number of examples belonging to one class is much much higher than the number of examples belonging to the other class. Let us assume that we are creating a binary classifier. In such a case, for our decision tree to train correctly, we should try to balance out the numbers for the two classes as much as possible. This may mean that we may *oversample* or *undersample* the elements of a class so that the numbers of examples for the two classes shown during the training of the classifier are more or less equal. Oversampling means that we may choose an element from the class with fewer examples more than once for training. Undersampling means that we may not choose all elements from the class that has too many labeled examples. There are more complex ways to take care of the unbalanced class problem, but the approach outlined above works reasonably well.

2.1.3 What Happens When We Repeat Building the Decision Tree?

While building a decision tree, if we follow the cross-validation idea and repeat the experiment several times, every time a new decision tree will be produced. Thus, we will have several decision trees produced and we need to decide which one to keep. We can keep the decision tree that provides the best performance over the test data. A better way is to keep some of the top performers, prune them as discussed in many texts and papers, and choose the tree that gives the best performance with the smallest number of nodes.

3 Measuring Performance

When we build a learning program, there are several things to measure to describe how a program learns and performs. The usual things to measure are:

- predictive accuracy of the trained classifier,
- speed of learning or training,

- space required by the learning program, and
- speed of the trained classifier when classifying new examples.

We should provide values for each of these measures in our results.

We discuss measuring predictive accuracy below.

3.1 Measuring Predictive Accuracy of a Trained Classifier

How well does a learned classifier program perform? One of the main criteria to look at is its predictive accuracy. In general, we want to choose a learned program that has the best predictive accuracy. Usually, predictive accuracy is measured in terms of two numbers that can be calcuated: *precision* and *recall*. Sometimes, another measure called the *F-measure* or F_1*-measure* is also calculated to summarize the values of precision and recall.

3.1.1 Calculating Precision and Recall for a Class

Let us assume we are performing two-way classification. That is, we need to classify elements of the test set into either belonging to a class or not belonging to a class.

Let us use the following symbols for our discussion.

- $\mid S_{++} \mid$: The number of positive examples in the test set that are classified as positive by the trained program. S_{++} is the set of *true positives*, i.e., positives classified as positive.
- $\mid S_{+-} \mid$: The number of positive examples in the test set that are classified as negative by the trained program. S_{+-} is the set of *false negatives*, i.e., positives classified as negative.
- $\mid S_{--} \mid$: The number of negative examples in the test set that are classified as negative by the trained program. S_{--} is the set of *true negatives*, i.e., negatives classified as negative.
- $\mid S_{-+} \mid$: The number of negative examples in the test set that are classified as positive by the trained program. S_{-+} is the set of *false positives*, i.e., negatives classified as positive.

For precision and recall, usually one considers the positive examples only. The definitions are given below for a 2-class case.

$$precision_+ = \frac{\mid S_{++} \mid}{\mid S_{++} \mid + \mid S_{-+} \mid}$$

$$recall_+ = \frac{\mid S_{++} \mid}{\mid S_{++} \mid + \mid S_{+-} \mid}$$

$precision_+$ gives a measure of precision for the *yes* class. Precision is measured by dividing the number of test examples that are correctly classified as positive by the classifier by the number of

test examples that are classified as positive (either correctly or wrongly) by the classifier. In other words, precision is the proportion of examples that are classified correctly as truly belonging a certain class. For example, suppose our trained classifier classifies 4 training examples as belonging to the *yes* class. Suppose 3 of these are correct and 1 is wrong. That is, 3 *yes* examples are classified as *yes* by the classifier; in addition, 1 *no* test example is classified as *yes* by the classifier. In this situation, the precision for the *yes* class is $\frac{3}{4}$ or 75%.

Recall measures the proportion of training examples belonging to a certain class that are classified correctly as belonging to the class. The formula given above computes recall for the positive class. Assume we have a total of 10 examples of the *yes* class in all of the training set. Of these, our trained classifier classifies 8 as belonging to the *yes* class. Then, our recall for the *yes* class is $\frac{8}{10} = .8$ of 80%.

Of course, one can also calculate precision and recall considering the negative examples as well.

If we have c number of classes in total, we compute these two metrics, precision and recall for each class. In other words, we compute $2c$ values.

3.1.2 Calculating F-measure for a Class

When there are two measures for a class, it is sometimes difficult to compare results of learning experiments, even for a single class. So, a combined measure for a class is used by many researchers. The measure normally used is called the *F-measure*. It is defined as follows.

$$\frac{1}{F} = \frac{1}{2}\left(\frac{1}{precision} + \frac{1}{recall}\right)$$

or

$$F = 2 \times \frac{precision \times recall}{precision + recall}$$

Thus, the F-measure for a class is the harmonic mean of the recall and precision for the class. When the *F-measure* is reported, we have one value to evaluate how our classifier learned to classify, considering one single class only.

We compute the F-measure value each class if we have c classes in total. If we compute precision, recall and F-measure for each of c classes, we have $3c$ computed values.

3.1.3 Macro-averaging over All Classes

If we have several classes, we can compute the three metrics: precision, recall and F-measure for each class separately. But, how do we report the overall results considering how the trained classifier performs over all the classes?

In macro-averging, the values of *precision*, *recall* and *F-measure* are calculated separately for each class. Macro-averaged precision is simply the average of the precisions for each class. If we have c

classes, we compute the sum of the *precision* values for all classes and divide the sum by c to obtain *macro-averaged precision*. We compute *macro-averaged recall* and *macro-averaged F-measure*.

In macro-averaging, we treat each class as equal, even if one class has many more examples than another.

3.1.4 Micro-averaging over All Test Examples

In micro-averaging, we obtain the precision, recall and F-measure values for each individual class as usual. However, when we average to obtain the overall result, we perform a weighted average computation. In other words, we weigh the value for a class by the proportion of total number of examples the class contains.

If we have two classes, + and −, the micro-averaged precision can be obtained as

$$precision_{micro} = p_+ \times precision_+ + p_- \times precision_-$$

where p_+ and p_- are the proportions of elements in the test set that are in the + and − classes, respectively. Obviously, $p_+ = 1 - p_-$. Here, $precision_{micro}$ gives us an idea how the trained classifier performs in regards to precision over the two classes classes.

For example, if we have c classes, we compute *micro-averaged precision* by performing a weighted sum of the *precision* values for the c classes. We compute *macro-averaged recall* and *macro-averaged F-measure* in a similar way.

Usually, if the various classes are not balanced, the smaller sized classes contribute well to the final micro-averaged measure. Thus, it is preferred if this is the case.

If the classes are more or less balanced, either one of micro and macro averaging works.

3.1.5 Final Results for Cross-validation

For cross-validation, we repeat the classification experiment several times, as discussed earlier. If we repeat the experiment r times, we obtain the average precision, recall and F-measure values over all the repeated runs. These are the values that are used to evaluate a classification algorithm. Oftern just a single metric, the average F-measure over all runs is used to evaluate a classification algorithm. The higher the average F-measure over all runs of an experiment, the better the trained classifier.