Day 11

30 Days of Machine Learning

Error Correcting Output Codes

Prepared By

Ahmed Ali

Quaid-e-Awam University of Engineering Science & Technology Nawabshah

Error Correcting Output Codes

The Error-Correcting Output Codes method is a technique that allows a multi-class classification problem to be reframed as multiple binary classification problems, allowing the use of native binary classification models to be used directly.
Unlike one-vs-rest and one-vs-one methods that offer a similar solution by dividing a multi-class classification problem into a fixed number of binary classification problems, the error-correcting output codes technique allows each class to be encoded as an arbitrary number of binary classification problems. When an overdetermined representation is used, it allows the extra models to act as “error- correction” predictions that can result in better predictive performance.
In error-correcting output codes (ECOC), the main classification task is defined in terms of several subtasks that are implemented by the base-learners. The idea is that the original task of separating one class from all other classes may be a difficult problem. Instead, we want to define a set of simpler classification problems, each specializing in one aspect of the task, and combining these simpler classifiers, we get the final classifier.
Base-learners are binary classifiers having output $-1/+1$, and there is a code matrix W of K $\times$ L whose K rows are the binary codes of classes in terms of the L base-learners dj. For example, if the second row of W is $[-1,+1,+1,-1]$, this means that for us to say an instance belongs to C2, the instance should be on the negative side of d1 and d4, and on the positive side of d2 and d3. Similarly, the columns of the code matrix define the task of the base-learners. For example, if the third column is $[-1,+1,+1]^T$, we understand that the task of the third base-learner, d3, is to separate the instances of C1 from the instances of C2 and C3 combined. This is how we form the training set of the base-learners. For example, in this case, all instances labeled with C2 and C3 form $X+$ and instances labeled with C1 form $X-$, and d3 is trained so that $x^t \in X+$ give output $+1$ and $x^t \in X-$ give output -1.
The code matrix thus allows us to define a polychotomy (K > 2 classification problem) in terms of dichotomies (K = 2 classification problem), and it is a method that is applicable using any learning algorithm to implement the dichotomize base-learners—for example, linear or multilayer perceptron’s (with a single output), decision trees, or SVMs whose original definition is for two-class problems.
The typical one discriminant per class setting corresponds to the diagonal code matrix where L = K. For example,
for K = 4,

we have

$$W = \begin{bmatrix} +1 & -1 & -1 & -1 \\ -1 & +1 & -1 & -1 \\ -1 & -1 & +1 & -1 \\ -1 & -1 & -1 & +1 \end{bmatrix}$$

The problem here is that if there is an error with one of the base learners, there may be a misclassification because the class code words are so similar. So the approach in error-correcting codes is to have L > K and increase the Hamming distance between the code words. One possibility is pairwise separation of classes where there is a separate base learner to separate Ci from Cj, for i < j. In this case, L = K(K − 1)/2 and with K = 4, the code matrix is

$$\mathbf{W} = \begin{bmatrix} +1 & +1 & +1 & 0 & 0 & 0 \\ -1 & 0 & 0 & +1 & +1 & 0 \\ 0 & -1 & 0 & -1 & 0 & +1 \\ 0 & 0 & -1 & 0 & -1 & -1 \end{bmatrix}$$

where a 0 entry denotes “don’t care.” That is, d1 is trained to separate C1 from C2 and does not use the training instances belonging to the other classes. Similarly, we say that an instance belongs to C2 if d1 = −1 and d4 = d5 = +1, and we do not consider the values of d2, d3, and d6. The problem here is that L is O(K2), and for large K pairwise separation may not be feasible.

If we can have L high, we can just randomly generate the code matrix with −1/ + 1 and this will work fine, but if we want to keep L low, we need to optimize W. The approach is to set L beforehand and then find W such that the distances between rows, and at the same time the distances between columns, are as large as possible, in terms of Hamming distance. With K classes, there are 2(K−1) − 1 possible columns, namely, two-class problems. This is because K bits can be written in 2K different ways and complements (e.g., “0101” and “1010,” from our point of view, define the same discriminant) dividing the possible combinations by 2 and then subtracting 1 because a column of all 0s (or 1s) is useless. For example, when K = 4,

we have

$$\mathbf{W} = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & +1 & +1 & +1 & +1 \\ -1 & +1 & +1 & -1 & -1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 \end{bmatrix}$$

When K is large, for a given value of L, we look for L columns out of the 2(K−1)−1. We would like these columns of W to be as different as possible so that the tasks to be learned by the base-learners are as different from each other as possible. At the same time, we would like the rows of W to be as different as possible so that we can have maximum error correction in case one or more base learners fail.

ECOC can be written as a voting scheme where the entries of W, wij , are considered as vote weights:

$$y_i = \sum_{j-1}^{L} w_{ij} d_j$$

and then we choose the class with the highest yi. Taking a weighted sum and then choosing the maximum instead of checking for an exact match allows dj to no longer need to be binary but to take a value between −1 and +1, carrying soft certainties instead of hard decisions. Note that a value pj between 0 and 1, for example, a posterior probability, can be converted to a value dj between −1 and +1 simply as

$$d_j = 2p_j - 1$$

One problem with ECOC is that because the code matrix W is set a priori, there is no guarantee that the subtasks as defined by the columns of W will be simple.

Ahmed Ali

Bagging

Bootstrap aggregating, often abbreviated as bagging, involves having each model in the ensemble vote with equal weight. To promote model variance, bagging trains each model in the ensemble using a randomly drawn subset of the training set. As an example, the random forest algorithm combines random decision trees with bagging to achieve very high classification accuracy.

The simplest method of combining classifiers is known as bagging, which stands for bootstrap aggregating, the statistical description of the method. This is fine if you know what a bootstrap is, but useless if you don't. A bootstrap sample is a sample taken from the original dataset with replacement, so that we may get some data several times and others not at all. The bootstrap sample is the same size as the original, and lots and lots of these samples are taken B of them, where B is at least 50, and could even be in the thousands. The name bootstrap is more popular in computer science than anywhere else, since there is also a bootstrap loader, which is the first program to run when a computer is turned on. It comes from the nonsensical idea of 'picking yourself up by your bootstraps,' which means lifting yourself up by your shoelaces, and is meant to imply starting from nothing.

Bootstrap sampling seems like a very strange thing to do. We've taken a perfectly good dataset, mucked it up by sampling from it, which might be good if we had made a smaller dataset (since it would be faster), but we still ended up with a dataset the same size. Worse, we've done it lots of times. Surely this is just a way to burn up computer time without gaining anything. The benefit of it is that we will get lots of learners that perform slightly differently, which is exactly what we want for an ensemble method. Another benefit is that estimates of the accuracy of the classification function can be made without complicated analytic work, by throwing computer resources at the problem (technically, bagging is a variance reducing algorithm; the meaning of this will become clearer when we talk about bias and variance). Having taken a set of bootstrap samples, the bagging method simply requires that we fit a model to each dataset, and then combine them by taking the output to be the majority vote of all the classifiers. A NumPy implementation is shown next, and then we will look at a simple example.

```
# Compute bootstrap samples
    samplePoints = np.random.randint(0,nPoints,(nPoints,nSamples)) classifiers = [ ]
        for i in range(nSamples):
            sample = [ ]
            sampleTarget = [] for j in range(nPoints):
            sample.append(data[samplePoints[j,i]])
            sampleTarget.append(targets[samplePoints[j,i]])

# Train classifiers

classifiers.append(self.tree.make_tree(sample,sampleTarget,features))
```

The example consists of taking the party data that was used to demonstrate the decision tree, and restricting the trees to stumps, so that they can make a classification based on just one variable

Ahmed Ali

When we want to construct the decision tree to decide what to do in the evening, we start by listing everything that we've done for the past few days to get a suitable dataset (here, the last ten days):

Deadline?	Is there a party?	Lazy?	Activity
Urgent	Yes	Yes	Party
Urgent	No	Yes	Study
Near	Yes	Yes	Party
None	Yes	No	Party
None	No	Yes	Pub
None	Yes	No	Party
Near	No	No	Study
Near	No	Yes	TV
Near	Yes	Yes	Party
Urgent	No	No	Study

The output of a decision tree that uses the whole dataset for this is not surprising: it takes the two largest classes and separates them. However, using just stumps of trees and 20 samples, bagging can separate the data perfectly, as this output shows:

```
Tree Stump Prediction
['Party', 'Party', 'Party', 'Party', 'Pub', 'Party', 'Study', 'Study', ↩
'Party', 'Study']
Correct Classes
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party', ↩
'Study']
Bagged Results
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party', ↩
'Study']
```

RANDOM FORESTS

A random forest is an ensemble learning method where multiple decision trees are constructed and then they are merged to get a more accurate prediction.

If there is one method in machine learning that has grown in popularity over the last few years, then it is the idea of random forests. The concept has been around for longer than that, with several different people inventing variations, but the name that is most strongly attached to it is that of Bierman, who also described the CART algorithm in unit 2.

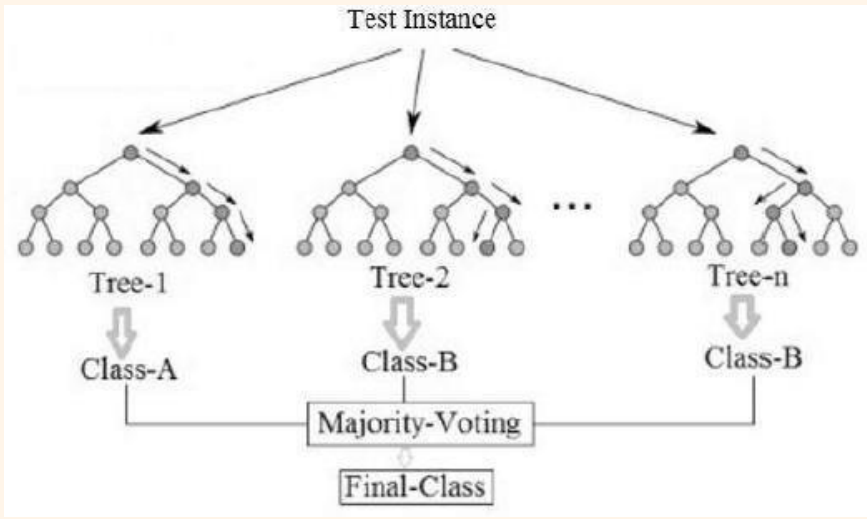

Figure 3: Example of random forest with majority voting.

© Ahmed Ali

The idea is largely that if one tree is good, then many trees (a forest) should be better, if there is enough variety between them. The most interesting thing about a random forest is the ways that it creates randomness from a standard dataset. The first of the methods that it uses is the one that we have just seen: bagging. If we wish to create a forest then we can make the trees different by training them on slightly different data, so we take bootstrap samples from the dataset for each tree. However, this isn't enough randomness yet. The other obvious place where it is possible to add randomness is to limit the choices that the decision tree can make. At each node, a random subset of the features is given to the tree, and it can only pick from that subset rather than from the whole set.

As well as increasing the randomness in the training of each tree, it also speeds up the training, since there are fewer features to search over at each stage. Of course, it does introduce a new parameter (how many features to consider), but the random forest does not seem to be very sensitive to this parameter; in practice, a subset size that is the square root of the number of features seems to be common. The effect of these two forms of randomness is to reduce the variance without effecting the bias. Another benefit of this is that there is no need to prune the trees. There is another parameter that we don't know how to choose yet, which is the number of trees to put into the forest. However, this is fairly easy to pick if we want optimal results: we can keep on building trees until the error stops decreasing.

Once the set of trees are trained, the output of the forest is the majority vote for classification, as with the other committee methods that we have seen, or the mean response for regression. And those are pretty much the main features needed for creating a random forest. The algorithm is given next before we see some results of using the random forest.

Algorithm

Here is an outline of the random forest algorithm.

1. The random forests algorithm generates many classification trees. Each tree is generated as follows:
    a) If the number of examples in the training set is N, take a sample of N examples at random - but with replacement, from the original data. This sample will be the training set for generating the tree.
    b) If there are M input variables, a number m is specified such that at each node, m variables are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the generation of the various trees in the forest.
    c) Each tree is grown to the largest extent possible.
2. To classify a new object from an input vector, put the input vector down each of the trees in the forest. Each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification

The implementation of this is very easy: we modify the decision to take an extra parameter, which is m, the number of features that should be used in the selection set at each stage. We will look at an example of using it shortly as a comparison to boosting.

Looking at the algorithm you might be able to see that it is a very unusual machine learning method because it is embarrassingly parallel: since the trees do not depend upon each other, you can both create and get decisions from different trees on different individual processors if you have them. This means that the random forest can run on as many processors as you have available with nearly linear speedup.

There is one more nice thing to mention about random forests, which is that with a little bit of programming effort they come with built-in test data: the bootstrap sample will miss out about 35% of the data on average, the so-called out-of-bootstrap examples. If we keep track of these datapoints then they can be used as novel samples for that tree, giving an estimated test error that we get without having to use any extra datapoints.
This avoids the need for cross-validation.
As a brief example of using the random forest, we start by demonstrating that the random forest gets the correct results on the Party example that has been used in both this and the previous chapters, based on 10 trees, each trained on 7 samples, and with just two levels allowed in each tree:

```
RF prediction
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party',
'Study']
```

As a rather more involved example, the car evaluation dataset in the UCI Repository contains 1,728 examples aiming to classify whether a car is a good purchase based on six attributes. The following results compare a single decision tree, bagging, and a random forest with 50 trees, each based on 100 samples, and with a maximum depth of five for each tree. The random forest is the most accurate of the three methods.

```
Tree
Number correctly predicted 777.0
Number of testpoints  864
Percentage Accuracy  89.9305555556

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 18.0
Percentage Accuracy 46.1538461538
-----
Bagger
Number correctly predicted 678.0
Number of testpoints  864
Percentage Accuracy  78.4722222222

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 0.0
Percentage Accuracy 0.0
-----
Forest
Number correctly predicted 793.0
Number of testpoints  864
Percentage Accuracy  91.7824074074

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 20.0
Percentage Accuracy 51.28205128
```

Strengths and weaknesses

Strengths

The following are some of the important strengths of random forests.

- It runs efficiently on large data bases.
- It can handle thousands of input variables without variable deletion.
- It gives estimates of what variables are important in the classification.
- It has an effective method for estimating missing data and maintains accuracy when a large proportion of the data are missing.
- Generated forests can be saved for future use on other data.
- Prototypes are computed that give information about the relation between the variables and the classification.
- The capabilities of the above can be extended to unlabeled data, leading to unsupervised clustering, data views and outlier detection.
- It offers an experimental method for detecting variable interactions.
- Random forest run times are quite fast, and they are able to deal with unbalanced and missing data.
- They can handle binary features, categorical features, numerical features without any need for scaling.

Weaknesses

- A weakness of random forest algorithms is that when used for regression they cannot predict beyond the range in the training data, and that they may over-fit data sets that are particularly noisy.
- The sizes of the models created by random forests may be very large. It may take hundreds of megabytes of memory and may be slow to evaluate.
- Random forest models are black boxes that are very hard to interpret.

Up Next:

Boosting: Train next learner on mistakes made by previous Learner.