8. Ensemble learning

The basic idea is that by having several learners or classifiers that each gets slightly different results on a dataset - some learning certain things well and some learning others – and putting them together. The results generated will be better than any one of them on its own provided that one succeeds well in joining them, otherwise could be even worse.

Fig. 8.1 shows the basic idea of *ensemble learning*, as these methods are called. Given a relatively simple binary classification problem and some learner that puts an ellipse around a subset of the data, combining the ellipses can provide a considerably more complex decision boundary.

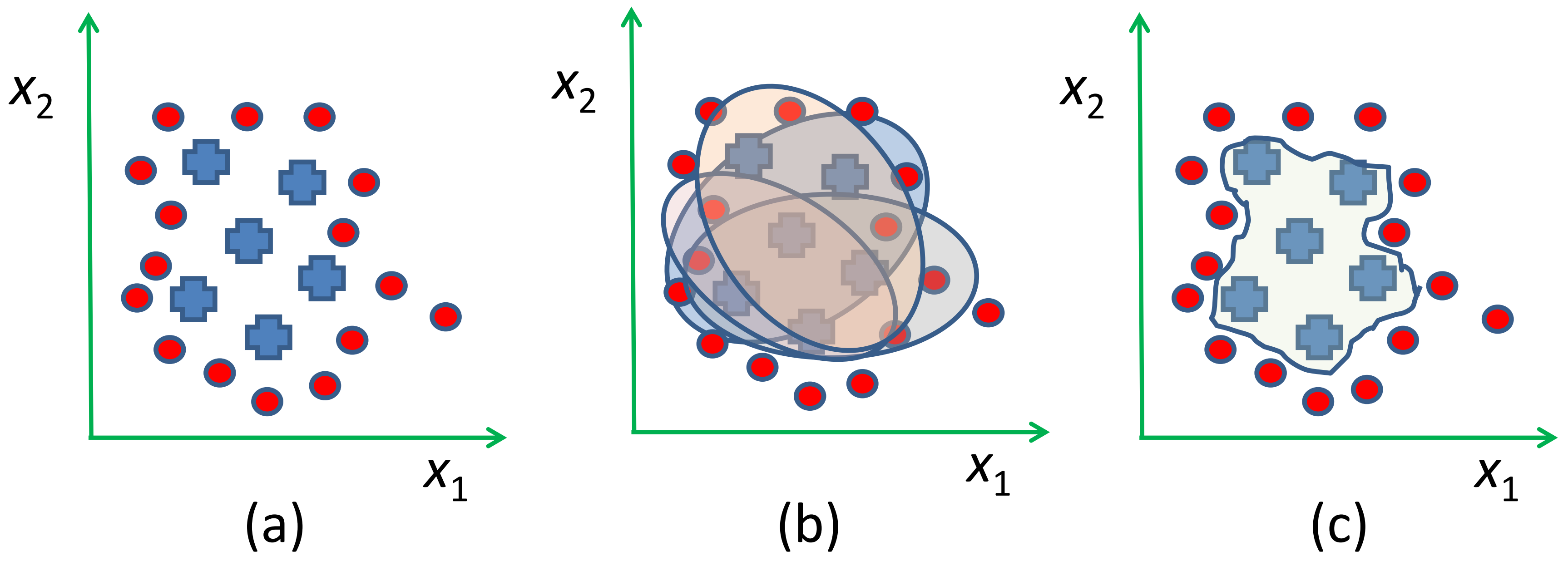

Fig. 8.1 (a) By combining several simple (b) classifiers (here hypothetically as ellipses), the decision boundary can be made much more complex, (c) enabling the difficult separation of two classes.

Interestingly, ensemble techniques do very well when there is very little data as well as when there is too much. This is a little bit like crossvalidation used when there was not enough data to go around, and trained several classifiers on different subsets of the data. The most built models are rejected. With an ensemble technique we keep them all, and combine their results in some way.

One simple way to combine the results is majority voting. This has the interesting property that for binary classification, the combined classifier will only receive the answer wrong if more than half of the classifiers were wrong. Hopefully, this is not going to occur frequently.

8.1 Boosting

At first sight the claim of the most popular ensemble technique, *boosting*, seems even amazing. By taking a set of poor learners, each performing only little better than chance, by putting them together it is possible to construct an *ensemble learner* that is able to perform arbitrarily well.

The principal algorithm of boosting is AdaBoost introduced in the 1990s by Freund and Schapire.

AdaBoost

The innovation is that AdaBoost (stands for *adaptive boosting*) is used to give weights to each datapoint according to how difficult previous classifiers have found to get it correct. Weights are given to the classifier as part of the input while training.

At each iteration a new classifier is trained on the training set, with the weights applied to the training set for each datapoint being modified at every iteration according to how successfully that datapoint has been classified in the past.

Initially, all weights are set to the same value, 1/*N*, where *N* is the number of datapoints or cases in the training set. At each iteration, the error *ε* is computed as the sum of the weights of the misclassified datapoints, and the weights for incorrect cases are updated by being multiplied by:

$$\alpha = \frac{1-\varepsilon}{\varepsilon}$$

Weights for correct datapoints are left alone, and then the whole set is normalized so that it sums up to 1, which is effectively a reduction in the importance of the correctly classified datapoints.

Training terminates after a set number of iterations. (Additionally, either after all datapoints are classified correctly, or one point contains more than half of the available weight can be used.)

Fig. 8.2 shows the influence of weighting incorrectly classified cases as training proceeds.

Function $h_t(\mathbf{x}_j)$ maps onto binary, actually *bipolar* -1 and +1 values (classes).

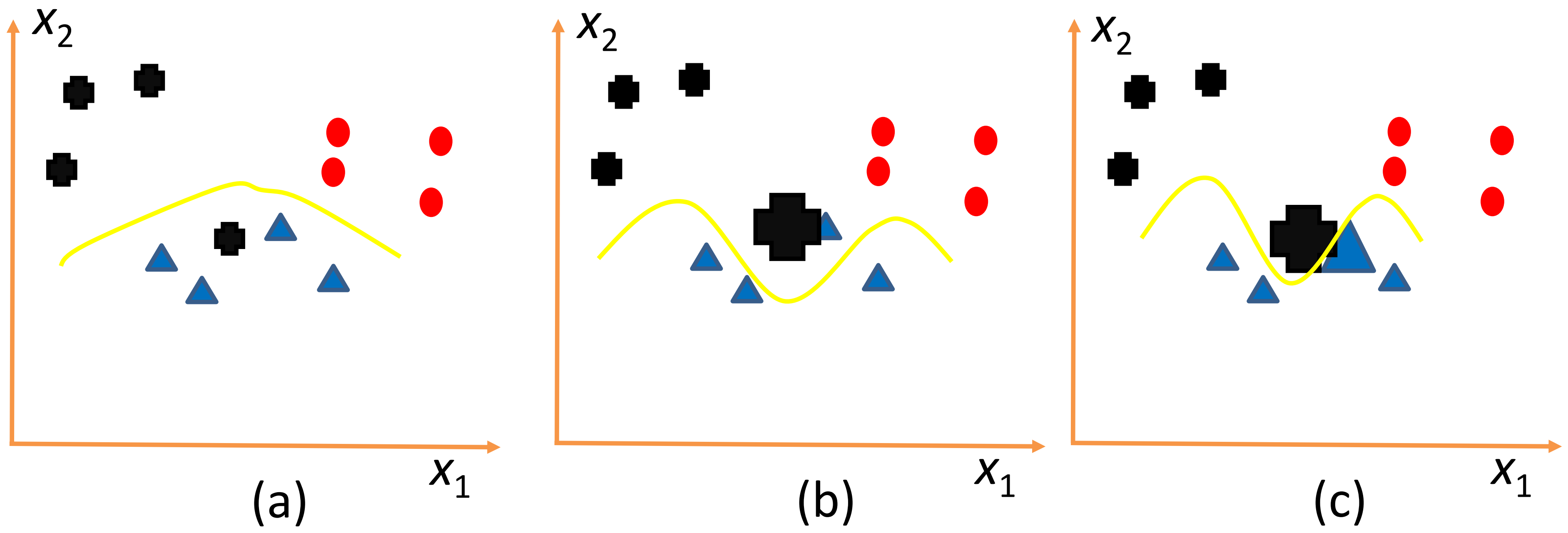

Fig. 8.2 (a) As datapoints are misclassified, (b), (c) so their weights increase in boosting (with the size of each datapoint being a measure of its importance), which makes the importance of those datapoints increase, making the classifiers pay more attention to them.

AdaBoost algorithm

(1) Given training set $L=\{(\mathbf{x}_1,y_1), (\mathbf{x}_2,y_2),\ldots, (\boldsymbol{x}_N,y_N)\}$, y_j in $\{-1,+1\}$ (y_j as known class label).

(2) Intialize all weights to $w_j^{(1)}=1/N$, where N is the number of cases.

(3) For $t=1,\ldots,T$:

(2.1) Train any ”weak” classifier on $\{L,w^{(t)}\}$, getting ”weak” *hypotheses* $h_t(\mathbf{x}_j)$ for cases $\mathbf{x}_j$, where

$h_t(\mathbf{x}_j) \to \{-1,+1\}$.

(2.2) Compute training error

$$\varepsilon_t = \sum_{j=1}^{N} w_j^{(t)} y_j h_t(\mathbf{x}_j)$$

(2.3) Set

$$\alpha_t = \frac{1}{2} \ln \frac{1-\varepsilon_t}{\varepsilon_t}$$

(2.4) Update weights using

$$w_j^{(t+1)} = w_j^{(t)} \exp\left(-y_j \alpha_t h_t\left(\mathbf{x}_j\right)\right) / Z_t,\ j = 1, \ldots, N$$

where Z_t is a normalization constant chosen so that new weights $w_j^{(t+1)}$ sum up to 1.

Note that this basic algorithm was designed for binary classification.

There were *N* labeled training cases with labels y_j from {-1,+1}. On each round *t*, weights were computed and a given weak classifier was applied to find a weak hypothesis h_t with low weighted error ε_t. The final, combined hypothesis gives a weighted combination of weak hypothesis, ”strong” classifier.

$$f(\mathbf{x}) = \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})$$

Here terms are like weighted voting for the series of generated classifiers. Finally, the classification ouput is from {-1,+1} as sign:

$$\operatorname{sign}(f(\mathbf{x}))$$

8.2 Bootstrap

We jump for a while to a testing method called *bootstrap*, because this is utilized in the next subsection.

The bootstrap is based on the statistical procedure of sampling data cases with *replacement*. Previously, whenever a sample of cases was taken randomly from the dataset to form a training set or test set (or validation set), it was drawn without replacement. In other words, any case, once selected, could not be selected again.

Now we also apply replacement so that it is possible to select a case more than once. This may, of course, yield such sets that some cases from the original set do not occur at all. Note that the resulting group of cases including more or less identical cases is not mathematically a set because of identical cases (duplicates). We could call it a collection.

The idea of bootstrap is to sample the dataset with replacement in order to create particularly a training set (set expressed ”freely”).

We view a particular variant of bootstrap called the 0.632 bootstrap. For this, a dataset of *n* cases is sampled *n* times, with replacement, to form another dataset of *n* cases. Since some elements or cases in this new dataset will almost certainly be repeated, there must be some cases in the original dataset that are missing the generated training set.

What is the chance that a particular case will not be selected for the training set? It has a 1/*n* probability of being picked each time and so a 1- 1/*n* probability of not being taken. By multiplying these probabilities together for a sufficient number of picking opportunities, *n*, we get the result as follows where *e* is the base of natural logarithms.

$$\left(1-\frac{1}{n}\right)^{n} \rightarrow e^{-1} \approx 0.368, n \rightarrow +\infty$$

This gives the chance of a particular case not being selected at all. Thus, for a reasonably large dataset, the test set will contain about 36.8% of all cases and the training set will comprise about 63.2% of them (this gives the name of the method). Some cases will be repeated in the training set, bringing it up to a total size of *n* cases, the same as in the original set. The test set will include 36.8% of original *n* cases.

The whole bootstrap procedure is repeated several times, with different replacement samples for the training set, and the results are averaged.

8.3 Bagging

The simplest way of combining classifiers is called *bagging*, which stands for *bootstrap aggregating*, the statistical description of the method.

The benefit of bagging is that we attain lots of classifiers or learners that perform slightly differently, which is exactly as desired for an ensemble classifier. 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.

8.4 Random forests

The idea of a random forest is largely that if one tree is good, then many trees, a forest, should be better, provided that 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 techniques that it uses is the one that we considered, bagging. To create a forest we can make the trees different by training them on slightly different data. Thus, we use bootstrap samples for every tree.

This does not yet suffice. We also add randomness by limiting the choices that the decision tree can make. At each node, a random subset of the variables is given to the tree, and it can only pick from that subset rather than from the whole set.

As well as increasing randomness in the training of each tree, it also speeds up the training, since there are fewer variables. Typically, the number of the selected variables is the square root of p, all variables.

The influence of these two forms of randomness is to reduce the variance without effecting the bias. In addition, there is no need to prune the trees.

The number of trees to be built is chosen, in its simplest way, as a predefined number.

Once the set of trees are trained, the output of the forest is the majority vote of the classifiers for classification. Ultimately, the algorithm is given.

(In Matlab random forests can be generated with ’TreeBagger’ and tested with ’predict’.)

The basic random forest training algorithm

For each of *M* trees:

- Create a new bootstrap sample of the training set.
- Use this bootstrap sample to train a decision tree.
- At each node of the decision tree, randomly select *m* variables, and compute the information gain (or Gini impurity) only on that set of variables, selecting the optimal one.
- Repeat until the tree is complete.