CHAPTER

9 Ensemble methods

Ensemble methods are learning algorithms that create a collection, or "ensemble" of other learners and then combine the outputs of this collection to make the overall prediction or decision. Ensembles can be made up of many instances of the same type of learner (*homogeneous* ensembles) or contain many different types of learner (*heterogeneous* ensembles). There are many methods for constructing ensembles of learners, and different types of ensembles can be used to either increase the flexibility of a model, or to mitigate the effects of overfitting.

9.1 Committee models and stacking

The simplest type of ensembles simply collect a number of different predictors, and then combine them through an averaging or voting process. For example, we may have a number of different learners $\{f_1(x), \ldots, f_B(x)\}$ that we have fit to our data set. In previous chapters, we have selected amongst various models by comparing their empirical loss scores. However, if each model makes its mistakes on a different subset of examples, it is possible for an aggregated predictor to do better than any one of the individual models. We can aggregate our predictions through, for example, and weighted average (regression) or vote (classification):

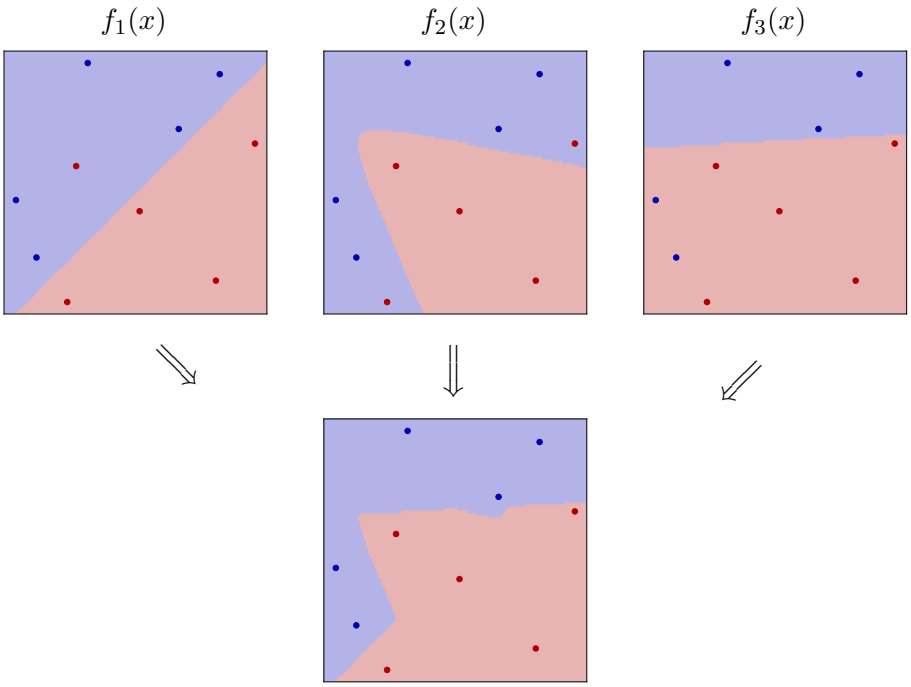
$$f(x) = \sum_b \alpha_b f_b(x\,;\, D_b) \qquad \text{(Regression)}$$

$$f(x) = \arg\max_y \sum_b \alpha_b \mathbb{1}[f_b(x\,;\, D_b) = y] \qquad \text{(Classification)}$$

Committees. If $\alpha_b = \frac{1}{B}$ for all b, this is a simple average (or vote), and the resulting classifier is called a *committee* **?**. Alternatively, the coefficients α_b of each individual model can be used to up-weight models that are more accurate, and downweight models that are less accurate. For classifiers that express not only the decision $f_b(x)$, but also a confidence in each possible prediction ("soft" decisions), we can also opt to average the confidence values instead of the decision indicators $\mathbb{1}[f_b(x) = y]$. This allows models that are more confident about a particular data point to have more influence on the average, compared to models that are uncertain in their prediction.

Example 9-1 : Committee classifier

Consider a binary classification problem, with $y^{(i)} \in \{-1, +1\}$. We train three models $\{f_1, f_2, f_3\}$, each of which produces a decision boundary with one to two errors on the training data. However, each classifier makes a different set of mistakes, so that the committee prediction is correct on all the data:



$$f(x) = \text{sign}\left[\tfrac{1}{3}f_1(x) + \tfrac{1}{3}f_2(x) + \tfrac{1}{3}f_3(x)\right]$$

Because the class predictions are binary, $f_b(x) \in \{-1, +1\}$, we can compute the committee prediction by simply testing whether the average of the $\{f_b\}$ is positive or negative.

Stacking. Suppose that we wish to find good values for the coefficients α_b of each classifier. Looking at the form of the vote, we can see that it matches the form of a perceptron, in which the features are given by the individual model outputs $f_b(x)$. So, we could identify values for the α_b by simply treating our individual model outputs as features, and then fitting a perceptron.

We call this approach *stacking*, because we are creating another classifier on top of our existing collection. If both the individual classifiers and the stacked classifier are perceptrons, this operation looks a bit like a multi-layer perceptron (neural network), except that the individual classifiers are all trained separately first, and then held fixed while the stacked classifier is trained, rather than being optimized jointly as would a neural network.

A major advantage of this piecewise training process is that we can stack individual classifiers that are not (or not all) perceptrons – nearest neighbor models, decision trees, etc. Moreover, we are not limited to using perceptrons for the stacking model, to combine the individual outputs; since the individual models are fixed, their outputs become features which can be input to any type of model.

One important point is that we should train the stacked classifier using data that have been held out from the data used to train the individual classifiers. If we use the same training data for both stages, it is easy to see we can have a problem – any model that overfits sufficiently on its training data will then appear to predict with high accuracy, and will be relied on heavily in the stacked model. Using hold-out

or validation data to train the stacked model ensures that heavily overfit models will not be given outsize weight. In practice, we can often use most or all of our usual validation data for this purpose, since the stacked model is usually not very complex (for example, a linear model) and so is unlikely to overfit, which would necessitate evaluating on its own validation data.

Can also use local weighting functions, $\alpha_b(x)$? Mixtures of experts models?

9.2 Bagging

Bagging, or "bootstrap aggregation" is a very simple but powerful type of ensemble method. The basic procedure behind bagging is to create a collection of B learners, each of which was trained on only a subset of the original m training data points. We use *bootstrap sampling* Efron and Tibshirani [1993], which draws a set of m_b (where $m_b \leq m$) data points from our data set with replacement. Each learner draws a subset of the data and trains itself on only this subset. To make a prediction, we evaluate each learner on the new test data point, and simply take the majority prediction.

The code for bagging, given in Algorithm 9.1, is extremely simple. Let the "base" learner be some standard type of learner (often a decision tree, but it can be anything). Then, if m and m_b represent the full training data size and the bootstrap size respectively, we simply draw m_b data indices uniformly, and build the bootstrap data set D_b by adding the data point corresponding to each index. Since we are sampling with replacement, some points will be drawn multiple times (and thus we will have multiple observations of the same data point in D_b), while other points will be left out. We then fit learner b using D_b, and repeat the process until we have trained B different learners. Finally, our overall predictor is constructed from the unweighted average of the B learners, as in a committee model.

The purpose of bootstrap sampling is to assess the sensitivity of a function (here, fitting our learner) to small differences in the data samples. Recall from Section **??** that the frequentist concept of the variance of a learner measured the variation in our predictions as a function of the randomness inherent in generating or collecting the data. This variance is what bootstrapping is designed to estimate – we imagine that the "true" distribution is defined by D, and evaluate how much variation we would experience, and how much our learner would change, as we simulate drawing new datasets.

Instead of merely estimating this variance, bagging uses the bootstrap samples to reduce variance by averaging. If our base learner $f_b(\cdot)$ has low bias, but high variance (i.e., it is prone to overfitting), the average of several learners should also have low bias, but the variance of the average will be reduced by a factor of B, reducing the degree of overfitting.

Example 9-2 : Bagged decision trees

Suppose that we perform bootstrap sampling on the Iris dataset, followed by fitting a "full" decision tree (until all data in D_b are correctly classified). We can look at the first few such learners:

Algorithm 9.1 Building a bagged ensemble of learners.

for each ensemble member $b \in \{1, \ldots, B\}$, **do**

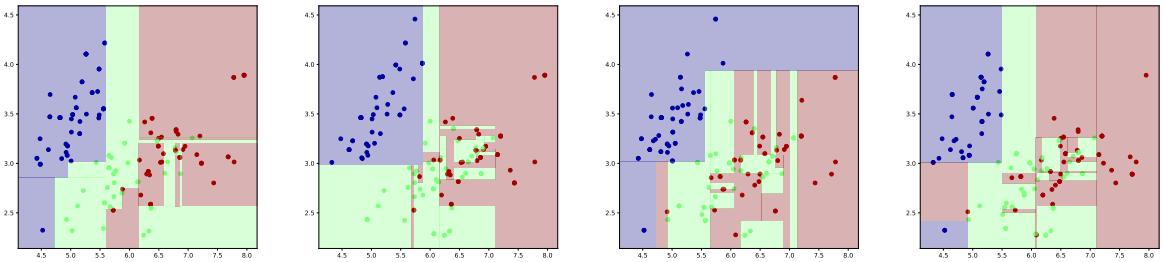
Sample a dataset D_b of size m_b from D, independently with replacement:

$$Z_b = \{i \sim \text{Discrete}(\tfrac{1}{m}, \ldots, \tfrac{1}{m})\} \qquad D_b = \{(x^{(i)}, y^{(i)}) \ : \ i \in Z_b\}$$

Train learner $f_b(x\,;\, D_b)$

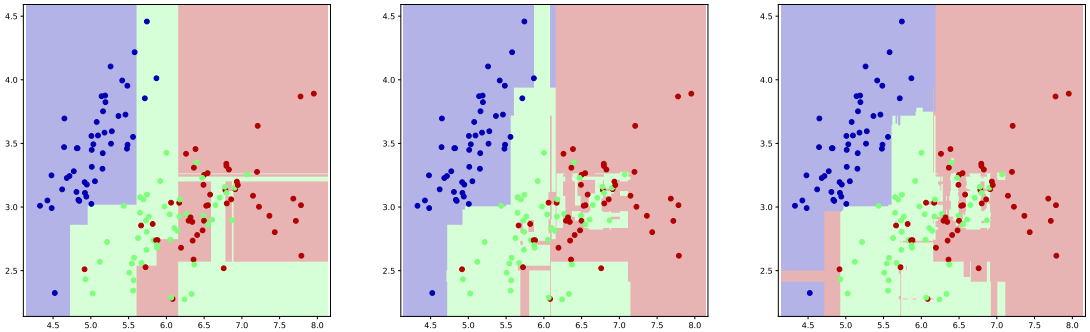
end for

Predict using an average or majority vote on the learners $\{f_b(x)\}$.



$b = 1$ $b = 2$ $b = 3$ $b = 4$

Here we see that sampling produces small differences in the training data – for example, the blue point at the far bottom left appears in three of the four samples, and the nearby red point appears in two of four. Even these minor changes lead to significantly different decision functions in this part of the space, indicating the learner's high variance. Computing the average (unweighted vote) over B such trees (and showing the full data set), we find:



$B = 1$ $B = 5$ $B = 25$

i.e., the decision function becomes "simpler" in the sense that it better matches the our intuition of the general regions of each class (compare to, for example, the k nearest neighbor classifiers in Figures 2.2 and 2.4). This suggests it may be more likely to generalize well to new data as well.

Selecting $m_b = m$ is typical, since this corresponds to estimating the amount of variation we expect to see in D itself, and ensures that any behavior of f_b or its training that depends on data size (for example regularization settings, or stopping criteria in decision trees) will be matched to how the learner would behave with the original data set. However, changing m_b can make our learners more or less random, if desired. If $m_b \gg m$, each D_b will be almost identical to D and exhibit little variation. If $m_b \ll m$, many data will be left out of each D_b and we will see more variation in the individual learners.

Note, must be nonlinear? If $f_b(x)$ is a linear function of X (linear regression), then so is f, and we will not see any improvement.

Random forests

Random forests are a variant of bagging that makes use of decision trees as the base classifier, but adds a small modification. Specifically, the greedy learning process used for constructing decision trees often interferes with our ability to create a good bagged ensemble. The purpose of bagging is to sample a variety of approximately equally good models that *could have* been fit to the data, i.e., we want to capture the diversity of possible explanations for our training data. However, if the number of data m is large,

the first, greedy split of each bootstrap sample D_b will often be the same, leading to substantially similar decision trees.

One solution to this is to enforce diversity in the decision tree construction. At each stage of the decision tree, we limit our search over features to a randomly chosen subset of size $n' < n$. Thus, at the root, any given tree may be unable to select the "best" feature, and so many trees will start in different ways. This does not significantly affect the quality of each final tree, since the subset is chosen randomly at each node – so, if some particular feature j is very important, and is unavailable at a particular node, the tree will split on a less important feature but can likely split on j at the next level of the decision tree. In this way we encourage finding very different trees (capturing the many possible explanations for the data) without increasing the bias of our models. The change is a trivial one-line alteration in Algorithm 8.2 (looping over n' random features, instead of all features). Typical practical choices for n' include $n' \approx \sqrt{n}$ [**?**] or $n' \approx n/2$ [**?**].

9.3 Boosting

Boosting is another, very popular style of ensemble. Boosted ensembles of learners are trained in a sequential way, with each new learner focusing on improving performance on the data that are not yet well predicted. This defines an incremental learning procedure, with a learner being added at each step in order to improve the prediction quality of the ensemble. The precise details of this process are typically what distinguishes different boosting algorithms.

Gradient Boosting

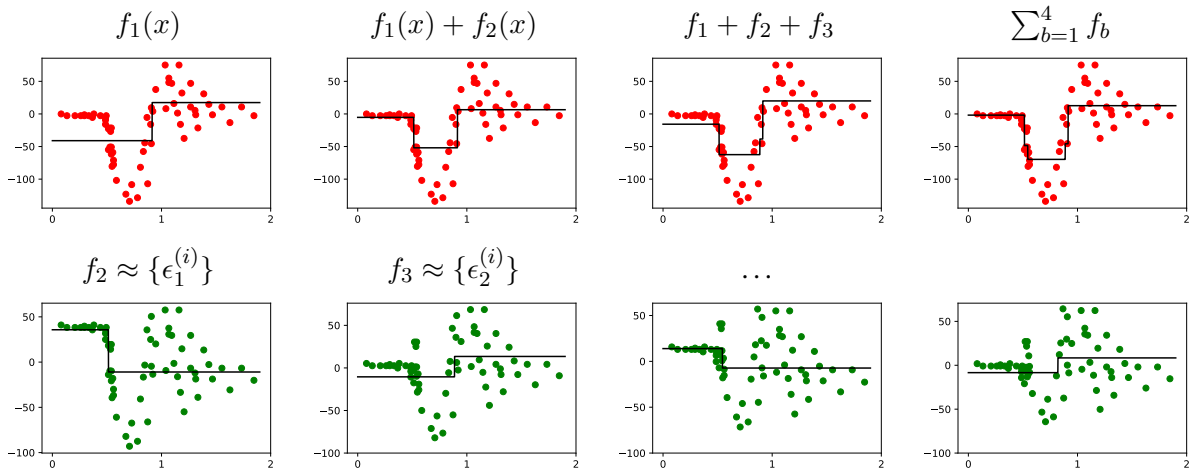
Gradient boosting methods work by building a sequence of base learners with continuous-valued outputs (i.e., regressors), although as we shall discuss they can be trained to perform classification as well. Let us first consider a simple case of regression to minimize a mean-squared error loss, and then generalize to other losses and tasks.

Suppose that we are given a collection of data $D_1 = \{(x^{(i)}, y^{(i)}\}$ for training, but we are only able to use a very simple learner, for example, a decision stump, to fit the data. We begin by training a model $f_1(x)$, which will end up underfitting the data; we can characterize its quality in terms of its error residuals, $\epsilon_1^{(i)} = y^{(i)} - f_1(x^{(i)})$.

Now, we learn the next model, f_2, to fix the errors made by f_1. We can do so by fitting $f_2(x)$ to the data $D_2 = \{(\epsilon_1^{(i)}, x^{(i)}\}$. Then, to predict y, we simply predict with f_1, and then estimate the error in f_1 with f_2, giving the overall predictor $f(x) = f_1(x) + f_2(x)$. Continuing, this predictor makes errors $\epsilon_2^{(i)} = y^{(i)} - f_1(x^{(i)}) - f_2(x^{(i)})$, and we can train f_3 to estimate and correct for these errors, and so on. This process is illustrated in an example.

Example 9-3 : Boosted decision stumps for MSE regression

Consider the motorcycle data from Silverman [1985]. We can fit $f_1(x)$ to the original data (red scatter plot), then evaluate the error residuals ϵ_1 at each point (green scatter plot) and fit the next function, f_2 to those points. This gives a new predictor $f_1 + f_2$, with new error residuals ϵ_2, with which we fit f_3, and so on:



Now let us see how to extend this idea to a more general loss function J. Suppose that we have trained $f_1(x)$; we can view its output $\hat{\underline{y}}$ on the data as a vector, $\hat{\underline{y}} = [f_1(x^{(1)}), \ldots, f_1(x^{(m)})]$, and compare it to the target vector $\underline{y}$ using loss $J(\underline{y}, \hat{\underline{y}})$.

If we could update our predictions $\hat{\underline{y}}$, how should we update them to improve J? One obvious solution is to update $\hat{\underline{y}}$ to move in the direction of the gradient of J, i.e., define

$$\epsilon_1^{(i)} = [\nabla J]_i = \frac{\partial J}{\partial \hat{\underline{y}}_i}.$$

Then, we view fitting f_2 to ϵ_1 as *projecting* this gradient onto the family of functions that f_2 can represent, and the new ensemble prediction $f(x) = f_1(x) + \alpha_2 f_2(x)$ as a (projected) gradient step in the vector space of $\hat{\underline{y}}$, with step size α_2. Evaluating the gradient of J at these new locations allows us to train f_3, and so on. After B steps, we are left with an overall predictor,

$$f(x) = \sum_{b=1}^{B} \alpha_b \, f_b(x),$$

whose output matches our (projected) gradient descent on $\hat{\underline{y}}$.

If our loss $J(\cdot)$ is the mean squared error, then the derivative

$$\epsilon_1^{(i)} = \frac{\partial J}{\partial \hat{\underline{y}}_i} = 2\big(y^{(i)} - f_1(x^{(i)})\big),$$

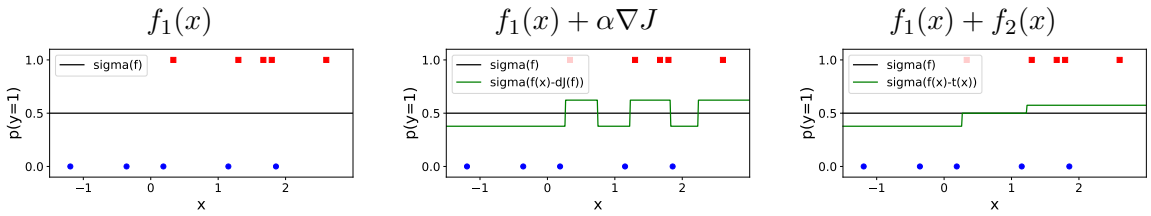
is simply the error residual, and step size $\alpha_b = 0.5$ gives the procedure discussed previously.

More generally, choosing large steps α_b will often give fast initial improvement in $f(x)$, but using smaller steps will allow the path to more closely approximate the gradient of J and also typically reduce the impact of the projection steps. Thus using a smaller step size usually leads to a better final predictor $f(x)$, but at the cost of increasing the number of boosted learners B that we train and store. In some settings this can lead to an undesirably slow prediction time, since each prediction also involves evaluating all B learners.

Using a general loss function J also allows us to apply the gradient boosting technique to classification. In particular, we can choose J to be a smooth surrogate classification loss, for example the logistic negative log-likelihood J_{NLL}, or logistic MSE J_{LMSE}, and use our regressors $f(x)$ to represent the smooth function of x inside the loss.

Example 9-4 : Gradient Boosting with Logistic NLL

Consider a simple classification problem involving two classes, $y \in \{0, 1\}$ and one feature x. We use $\sigma(f(x^{(i)}))$ to approximate the probability that $y^{(i)} = 1$, and train f using gradient boosting[1]. We can start with a trivial initial function, $f_1(x) = 0.5$, then evaluate the derivative at each data point; data with $y^{(i)} = 1$ will want $f(x^{(i)})$ to increase, while data with $y^{(i)} = 0$ will want $f(x^{(i)})$ to decrease. We visualize ∇J by interpolating between the data points using a full-depth decision tree, but train $f_2(x)$ by projecting ∇J onto a decision tree of maximum depth 2:



Notice that, due to the complexity restriction on f_2, we are unable to make all the data points more probable under our model (the rightmost blue point becomes less probable, for example), but that overall the loss J_{NLL} has improved. We can then proceed to continue to learn boosted trees:

Fill in figure, remaining iterations

AdaBoost

Another very popular boosting algorithm is AdaBoost, which was also the boosting algorithm that first popularized the framework. Adaboost is easiest to describe for binary classification on classes $y \in \{+1, -1\}$, and works by training a sequence of classifiers on *weighted* data, updating the weights to emphasize certain points and encourage the next classifier to predict them correctly.

More explicitly, let us associate a collection of weights $w^{(i)}$ with our data $D = \{(x^{(i)}, y^{(i)})\}$. For any decomposable loss J, we can define a weighted extension of J as

$$J_w = \sum_i w^{(i)} J^{(i)}\big(y^{(i)}, f(x^{(i)})\big).$$

When all weights $w^{(i)} = \frac{1}{m}$, J_w is simply the usual average loss J; however if some point i has $w^{(i)}$ large, its error $J^{(i)}$ will be more important to minimize than other data. Often it is straightforward to extend other, non-decomposable losses (such as AUC) to weighted data collections as well.

Typically, it is no harder to train a weighted loss than an unweighed one. For gradient-based optimization procedures on decomposable losses, the gradient of the weighted loss is simply a weighted average of data gradients. Some learners may require a model-specific adaptation; for example, the decision tree training algorithm is modified to use weighted averages in the computation of the information gain. Even in settings where weighted training is not possible, it can still be approximated by using a weighted bootstrap procedure to generate an unweighted data set that can stand in for the weighted set [**?**]. Simply draw each data point with probability $w^{(i)}$, with replacement, to build the unweighted data set – data points with high weight will be repeated more often in the bootstrap sample, and thus have more impact on the unweighted loss. However, this sampling step introduces extra variance into the learning process, which may be undesirable.

[1] If $f(x)$ were linear, this model would be equivalent to logistic regression.

Now, for AdaBoost, we take J to be the classification error rate, so that $J^{(i)} = \mathbb{1}[y^{(i)} = f(x^{(i)})]$. We initialize $w^{(i)} = \frac{1}{m}$, and train a classifier $f_1(x)$. We then choose our coefficient α_1 in such a way that it is larger if f_1 is more accurate, and smaller if it is not. We also update the weights $w^{(i)}$ to increase the weight of any data that we got wrong, and decrease the weights of data that we got right. Then we train $f_2(x)$ and repeat the process.

Specifically, we compute the coefficient for each learner b based on its weighted error rate,

$$\alpha_b = .5 \log \Big[\frac{1 - J_w}{J_w}\Big].$$

Note that, if J_w is small, α_b will be large; if $J_w \approx 0.5$ (indicating that f_b is no better than random guessing on the weighted data), $\alpha_b \approx 0$. We also use α_b to update the weights as well:

$$w^{(i)} \leftarrow w^{(i)} \exp\big[-\alpha_b\, y^{(i)}\, f_b(x^{(i)}\big]$$

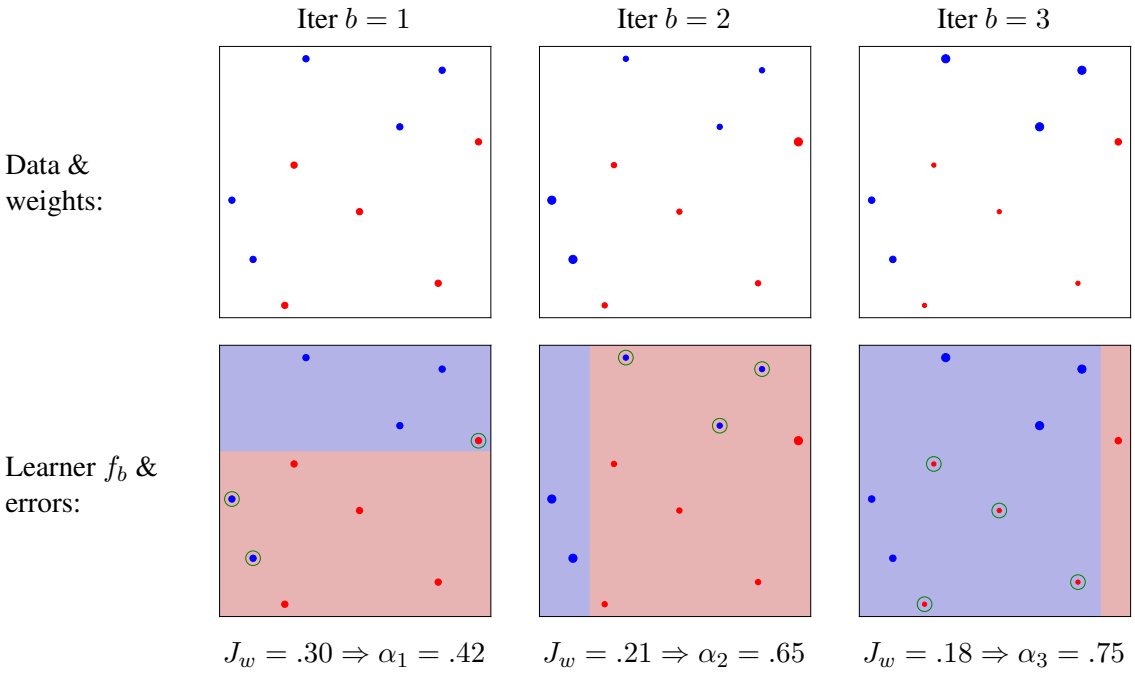
and re-normalize the weights to sum to one. Note that, since $f(x)$ and $y \in \{+1, -1\}$, the quantity $y\, f(x)$ is either +1 or -1 – it equals +1 if the prediction agrees with the true class, and -1 if it does not. This up-weights data on which we have predicted incorrectly, and down-weights correctly predicted data.

The final classifier is given by summing the weighted individual learners and applying a threshold:

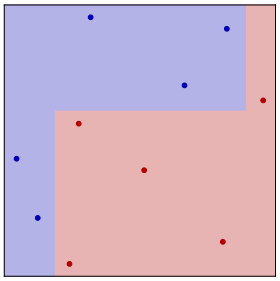
$$r(x) = \sum_b \alpha_b f_b(x) \qquad\qquad f(x) = \text{sign}\,\big[r(x)\big]$$

Example 9-5 : Adaboost example

Consider a small two-dimensional data set, on which we will learn an AdaBoosted ensemble of three decision stumps. Each column shows the training of one learner: at top, the data points, sized according to their current weight; at bottom, the learned decision stump function and its errors (green circles). We compute the weighted error J_w, which determines the coefficient α_b and the new data weights for the next iteration.



Initially, all the data are equal weight; at the next iteration, the three points we mis-predicted have increased weight, while the others have smaller weight. This means that we learn a different decision stump, which gets the high-weight data correct, but makes errors on some lower-weight data. Computing J_w and α_2, we re-weight the data again, and find a third decision stump, which now makes more errors, but only on low-weight points. The combined ensemble prediction is given by the linear combination of the three members; in this particular example, any two learners can outvote the other, giving the decision boundary at right. Notice that the final decision boundary is now more complex than any single member.



$$f(x) = \text{sign}\Big[\sum_b \alpha_b f_b(x)\Big]$$

AdaBoost and the exponential loss

Often, boosting algorithms can be shown to correspond to sequentially optimizing some particular surrogate loss function. In particular, AdaBoost corresponds to minimizing the exponential loss function,

$$J_{\text{exp}} = \sum_i \exp\big[-y^{(i)}\, r(x^{(i)})\big]. \tag{9.1}$$

To see this, we can proceed by induction. Let $r(x) = f_0(x) = 0$, so that $J_{\text{exp}} = m$, and define the weights on each data point i by

$$w^{(i)} = J^{(i)}_{\text{exp}} \,/\, J_{\text{exp}} = \frac{1}{m},$$

since all data points i have the same exponential loss, $J^{(i)}_{\text{exp}} = \exp(0) = 1$.

Now, suppose that we have trained ensemble members $\{f_0, \ldots, f_{B-1}\}$, with sum of learners $r(x) = \sum_{b=0}^{B-1} \alpha_b\, f_b(x)$, and have weights $w^{(i)} \propto J^{(i)}_{\text{exp}}(r)$. Consider training the next ensemble member, f_B, and selecting its coefficient α_B. The exponential loss of our new ensemble is,

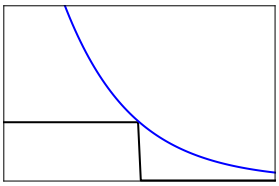
$$\begin{aligned}
J_{\text{exp}} &= \sum_i \exp\Big[-y^{(i)}\big(r(x^{(i)}) + \alpha_B\, f_B(x^{(i)})\big)\Big] \\
&= \sum_i \exp\big[-y^{(i)}\, r(x^{(i)})\big] \cdot \exp\big[-y^{(i)}\, \alpha_B\, f_B(x^{(i)})\big] \\
&\propto \sum_i w^{(i)} \cdot \exp\big[-y^{(i)}\, \alpha_B\, f_B(x^{(i)})\big] \\
&= \sum_{i:y^{(i)}=f_B(x^{(i)})} w^{(i)}\, \exp[-\alpha_B] \;+ \sum_{i:y^{(i)}\neq f_B(x^{(i)})} w^{(i)}\, \exp[\alpha_B] \\
&= \big(1 - J_w(f_B)\big)\, \exp[-\alpha_B] + J_w(f_B)\, \exp[\alpha_B]
\end{aligned}$$

since the $w^{(i)}$ sum to one. So, for any $\alpha_B > 0$, optimizing over f_B is equivalent to minimizing its weighted error rate, where the weights are set to $w^{(i)} \propto J^{(i)}_{\text{exp}}(r)$. If we train f_B and fix its value, optimizing over α_B gives

$$\begin{aligned}
0 = \frac{\partial J_{\text{exp}}}{\partial \alpha_B} &= -\big(1 - J_w(f_B)\big)\, \exp[-\alpha_B] + J_w(f_b)\, \exp[\alpha_B] \\
\Rightarrow \qquad \alpha_B &= \frac{1}{2} \log\big[1 - J_w(f_B)\big] - \log\big[J_w(f_B)\big]
\end{aligned}$$

Then, updating $r(x) \leftarrow r(x) + \alpha_B f_B(x)$ and $w^{(i)} \leftarrow w^{(i)} \exp[-y^{(i)}\, \alpha_B\, f_B(x^{(i)})]$ completes the induction.

Like other surrogate losses, the exponential loss (9.1) provides a smooth and convex upper bound on the 0/1 loss J_{01}. Intuitively, if y and $r(x)$ share the same sign (the prediction is correct), the loss will be small, and will grow smaller the more confident $r(x)$ is of the correct sign. At the decision boundary, $r(x) = 0$, the cost is 1, and the cost continues to increase as $r(x)$ becomes more confident in the incorrect sign.

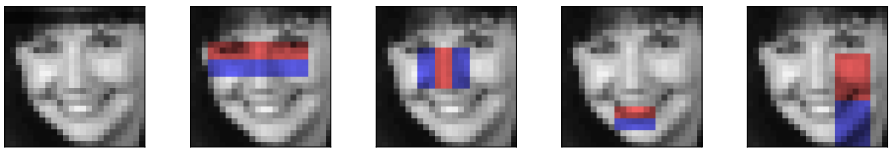


Exponential loss, J_{exp}

Example 9-6 : Face detection

An early success of boosting algorithms was the Viola-Jones model for face detection [Viola and Jones, 2001]. We first frame the problem of detecting a face in an image as a supervised learning task, of identifying whether a particular grayscale image patch of some fixed size (say, 24×24) contains a face. We then build a database of many image patches that are of faces, and many patches that are non-faces, and train a classifier. Suppose that we want to use some variant of decision trees for this problem.

Unfortunately, the raw pixel values themselves are not particularly informative; to detect a face, we need to look at many pixel values together. We will thus need to transform the pixels into some more useful representation; for example, the responses of various filters to the image. The Viola-Jones model uses *Haar wavelet* filter responses, which detect a set of horizontal and vertical patterns, parameterized by the patterns' size and location in the image. We can visualize what a few of these filters look like on a given image $x^{(i)}$:



$x^{(i)}$ $\Phi_{18280}(\cdot)$ $\Phi_{126740}(\cdot)$ $\Phi_{9816}(\cdot)$ $\Phi_{36834}(\cdot)$

Each filter response $\Phi_j(x)$ is computed by adding the intensities of all the pixels under the red region, and subtracting the intensity of pixels under the blue region; the remaining pixels are ignored.

For a 24×24 image patch, there are $n = 162,336$ possible Haar wavelet features Φ_j, making overfitting a serious concern for this representation. Viola-Jones works by using AdaBoost on decision stumps, so that each learner uses only a single feature at a time. We then check all the features, find the most informative – for example, Φ_{18280}, which may have a strong (negative) response on faces due to darker pixels around a person's eyes, and lighter pixels on their cheekbones – and add it to the ensemble, reweight the data, and repeat.