CS 5785 – Applied Machine Learning – Lec. 18

Prof. Nathan Kallus, Cornell Tech
Scribe: TBD

Nov. 2, 2017 (Under construction)

1 Boosting

1.1 Synthetic Example (Continued)

Picking up where we left off last lecture, if we apply Adaboost to this problem, we see in Fig. **??** we get down to around 6% error after 400 iterations. Recall that a single stump had a test error just slightly better than chance. Through a boosted combination of these very simple classifiers, we achieve a dramatic reduction in error rate.

Let's now take a closer look at how to arrive at the Adaboost algorithm.

1.2 Loss Function

Adaboost uses an exponential loss function:

$$L\left(y, f(x)\right) = \exp\left(-yf(x)\right)$$

where f is the raw prediction.

We use thresholding, i.e., compute sign(f), to get the class prediction from f.

Fig. **??** shows a plot of the exponential loss as a function of y times f. Recall that when $y = 1$, we want f to be positive, and when $y = -1$, we want f to be negative. Essentially, for a low loss value, we want the sign of the predicted value to match the ground truth. This implies that $yf > 0$ when the prediction is correct and we expect to incur relatively little loss. Hence, the ground truth y is either +1 or -1.

Originally, the function has a soft floating point in the training phase and in order to get the hard classification we apply the sigmoid function, which gives us +/-1. Example: Suppose a training vector is -1, then we want the predictive value to be negative. We get the loss which is less than 1.

The raw misclassification error (plotted in gray) is simply a step function, with zero loss when the classification is correct and unit loss otherwise - it tells how much loss you incur. The exponential is a convex upper bound (for loss) on step function. The plot shows other choices of loss functions and exponential is the one that leads to the AdaBoost algorithm.

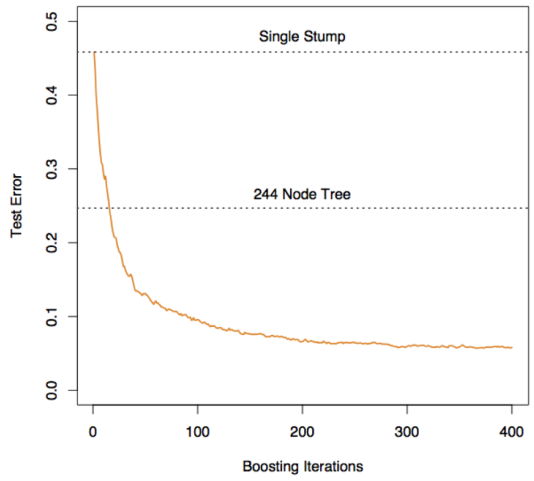


FIGURE 10.2. *Simulated data (10.2): test error rate for boosting with stumps, as a function of the number of iterations. Also shown are the test error rate for a single stump, and a 244-node classification tree.*

Figure 1: Fig 10.2

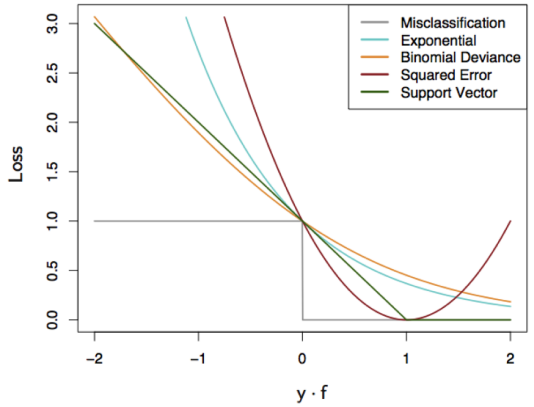


FIGURE 10.4. *Loss functions for two-class classification. The response is* $y = \pm 1$*; the prediction is* f*, with class prediction* $\text{sign}(f)$*. The losses are misclassification:* $I(\text{sign}(f) \neq y)$*; exponential:* $\exp(-yf)$*; binomial deviance:* $\log(1+\exp(-2yf))$*; squared error:* $(y-f)^2$*; and support vector:* $(1-yf)_+$ *(see Section 12.3). Each function has been scaled so that it passes through the point* $(0, 1)$*.*

Figure 2: Fig 10.4

1.3 Solving for the classifiers and weights

We cast our problem as one of finding the classifier G_m with coefficient β_m to be added at each step. (AdaBoost is a strictly additive approach.) Assume we have the classifier from the previous iteration of boosting and we'd like to tack on a new classifier (in our case, a decision stump) with some weight.

$$(\beta_m, G_m) = \underset{\beta, G}{\operatorname{argmin}} \sum_{i=1}^{N} \exp[-y_i(f_{m-1}(x_i) + \beta G(x_i))]$$

We want to find the weight and classifier to obtain some improvement in each step. We can write this as follows:

$$(\beta_m, G_m) = \underset{\beta, G}{\operatorname{argmin}} \sum_{i=1}^{N} w_i^{(m)} \exp(-\beta y_i G(x_i)) \tag{1}$$

with $w_i^{(m)} = \exp(-y_i(f_{m-1}(x_i))$. Note that $w_i^{(m)}$ depends neither on β nor $G(x)$, so we can think of it as a per-observation weight.

At each step, it ensures an improvement by weighting the mis-classifications with a higher weight so the next iteration addresses this. The weights change on each iteration since it depends on the previous iteration.

The solution to Equation (1) can be obtained in two steps. First, for any $\beta > 0$ the solution for $G_m(x)$ is the classifer that minimizes the weighted error rate in predicting y.

$$G_m = \underset{G}{\operatorname{argmin}} \sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i)) \tag{2}$$

To see this break up Equation (1) into the sum of two parts (the correctly predicted data points, contributing less to the cost, and the incorrect data points, contributing more):

$$e^{-\beta} \sum_{y_i = G(x_i)} w_i^{(m)} + e^{\beta} \sum_{y_i \neq G(x_i)} w_i^{(m)}$$

which we can rewrite G_m as:

$$(e^{\beta} - e^{-\beta}) \sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i)) + e^{-\beta} \sum_{i=1}^{N} w_i^{(m)}$$

Note: on any given iteration one will get some correct and some incorrect results. When $G(x_i)$ gets a threshold, then matching is conducted - it incurs a cost; positive when you get it correctly, negative when you get it wrong. In the cases when one gets it correct one gets a sum over all the weights.

Plug this G_m into Equation (1) and solve for β we get:

$$\beta_m = \frac{1}{2} \log \frac{1 - err_m}{err_m}$$

β_m is the weight on the weak learner that we're adding onto the new classifier (it is possible for β_m to be zero), and the err_m is the minimized weighted error

rate:

$$err_m = \frac{\sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i))}{\sum_{i=1}^{N} w_i^{(m)}}$$

If $err_m = 0.5$ (it's performing at chance) then the weight is 0 so the stump doesn't get added, which is what we want.

Before we do any iterations of boosting, the only notion of error was the sum over an indicator of any errors taking place. Once we have this, the error depends on these weights. We then use that to update the classifier.

We update our classifier as follows.

$$f_m(x) = f_{m-1}(x) + \beta_m G_m(x)$$

with weights for the next iteration defined as

$$w_i^{(m+1)} = w_i^{(m)} e^{-\beta_m y_i G_m(x_i)}$$

We use the fact that

$$-y_i G_m(x_i) = 2 \cdot I(y_i \neq G_m(x_i)) - 1$$

where it becomes 1 if an error occurs and 0 if its correct. Then we get

$$w_i^{(m+1)} = w_i^{(m)} \cdot e^{\alpha_m I(y_i \neq G_m(x_i))} e^{-\beta_m}$$

where $\alpha_m = 2\beta_m$ is the quantity in line 2c of the Adaboost algorithm, which is shown in Figure 3 from Lecture 17.

We can drop the factor e^{β_m} since it won't affect our maximization problem so we get line 2d shown in Figure 3 from Lecture 17:

$$\text{set } w_i \leftarrow w_i \ \exp[\alpha_m I(y_i \neq G_m(x_i))], \ i = 1, 2, ..., N$$

You take your weights from the last iteration, which emphasize the misclassified ones, and then try your best to solve for the parameters for the next stump. Initially, you have many weak classifiers, but at the end of AdaBoost you produce a strong classifier that's the weighted combination of these weak classifiers. In Fig. **??**, you can see how performance improves for our motivating example.

Fig. **??**, takes us through some of the iterations of AdaBoost, step by step. For a stump, you pick a variable (choice of 2 here) and threshold by sweeping. Now you have weights for each observation, bigger ones for the ones misclassified, so you do this again. The final strong classifier is a weighted combination of all these regions.

Fig. **??**, you see the same thing basically with the result of a final strong classifier. You get a weighted combination and just threshold half-way.

2 Viola-Jones

Recall that boosting is a meta-algorithm that sits on top of a weak classifier. For face detection, as seen Fig. **??**, advocated the use of a stump defined with a Haar-like wavelet.

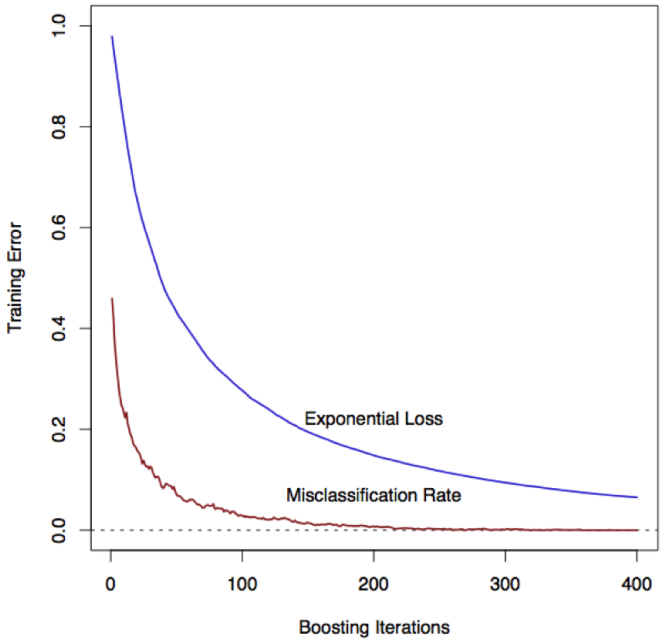
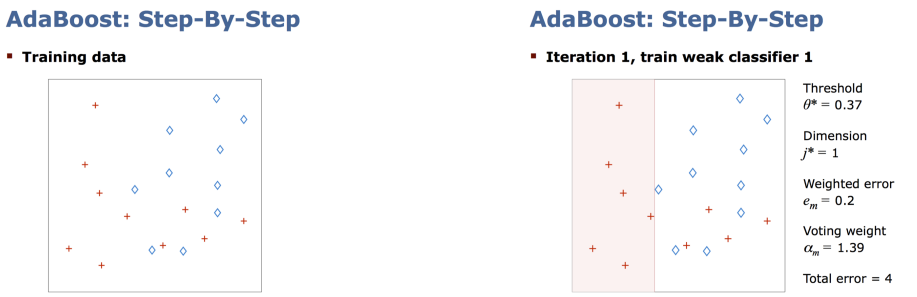


FIGURE 10.3. *Simulated data, boosting with stumps: misclassification error rate on the training set, and average exponential loss:* $(1/N)\sum_{i=1}^{N}\exp(-y_i f(x_i))$. *After about* 250 *iterations, the misclassification error is zero, while the exponential loss continues to decrease.*

Figure 3: Fig 10.3



(a) first (b) second

(c) third (d) fourth

(e) fifth (f) sixth

Figure 4: Animation of Adaboost

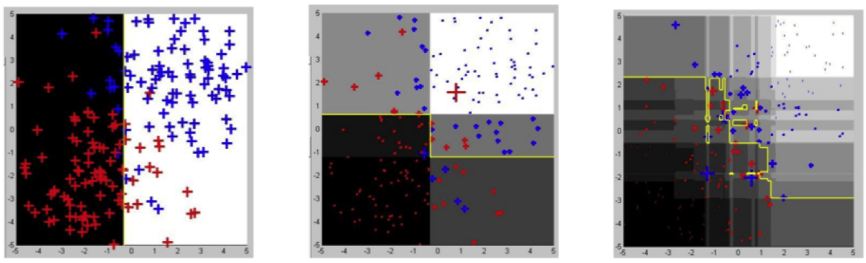


Figure 16.10 Example of adaboost using a decision stump as a weak learner. The degree of blackness represents the confidence in the red class. The degree of whiteness represents the confidence in the blue class. The size of the datapoints represents their weight. Decision boundary is in yellow. (a) After 1 round. (b) After 3 rounds. (c) After 120 rounds. Figure generated by `boostingDemo`, written by Richard Stapenhurst.

Figure 5: Fig 16.10

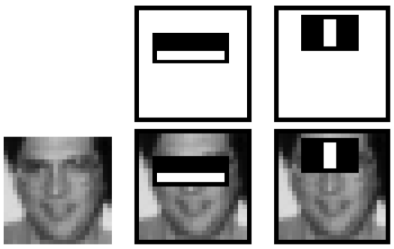


Figure 5: The first and second features selected by AdaBoost. The two features are shown in the top row and then overlayed on a typical training face in the bottom row. The first feature measures the difference in intensity between the region of the eyes and a region across the upper cheeks. The feature capitalizes on the observation that the eye region is often darker than the cheeks. The second feature compares the intensities in the eye regions to the intensity across the bridge of the nose.

Figure 6: Fig 5

Figure 8: Example of frontal upright face images used for training.

Figure 7: Training Images

A Haar-like wavelet involves picking an overall box size for the size of a face, say 96×96 with a fixed scale, that you then slide around the image ("cascaded implementation"). The Haar-like wavelet are a set of boxes that are $+1, -1$ and everywhere else is 0. These are the features that operationalize the stump, which is created using the inner product with the Haar-like wavelet and a cropped section of the image.

When you encounter an image of a face, a Haar-like wavelet finds a product between all the pixel values inside the window. It will compute the distance.

Viola-Jones method is super fast. Especially when compared to Neural Networks for the face bounding problem in 1999, around when Viola-Jones was introduced. Fig. **??** shows the training set of images and Fig. **??** shows the performance on different test images. Some of the things they used to make this efficient were:

1. Cascade: early rejection - created multilevel cascade with many boosted classifiers.

Figure 10: Output of our face detector on a number of test images from the MIT+CMU test set.

Figure 8: Test Images

TABLE 16.1. *Part of a* 15*-bit error-correcting coding matrix* **C** *for the* 10*-class digit classification problem. Each column defines a two-class classification problem.*

Digit	C_1	C_2	C_3	C_4	C_5	C_6	$\cdots$	C_{15}
0	1	1	0	0	0	0	$\cdots$	1
1	0	0	1	1	1	1	$\cdots$	0
2	1	0	0	1	0	0	$\cdots$	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
8	1	1	0	1	0	1	$\cdots$	1
9	0	1	1	1	0	0	$\cdots$	0

Figure 9: Table 16.1

2. Integral image trick: summed area table that is precomputed so the value is the sum of all pixel values inside a rectangle.

3. Multiscale search: also key to that faces can be huge selfies.

Boosting is still king in face detection. Random forests does well with multi-class classification.

Ada boost is a strictly additive classification method. When we are done boosting - then we threshold it at zero. Ada boost doesn't overfit. In most methods one needs to be careful how long one trains the model. For most methods when the training error is close to zero - the validation error can easily go up. This does not happen in boosting. Boosting is so pupular since it doesn't overfit. Increasing the iterations does no harm.

Fig. **??** shows an early example of a learning ensemble using boosting technology. Boosting can't handle multi-class. A clever response was ECOC (error correcting output code). We assign each digit a codeword and train the classifier to produce 0 or 1 for each digit. Boosting produces bits in a code word. Note that the rows have more bits than is neccessary and the idea is that the redundant "error-correcting" bits allow for some inaccuracies and can improve performance. In our case, for 10 digits, you have a length 15 code word. Then you calculate the Hamming distance to the closest codeword. This means some of the bits will get classified incorrectly but the error-correcting codes will tolerate this the best it can since the 8 classes code words are designed to be as far apart as possible. Actually this turns out not any better than randomly assigned code words but will not be covered and left as an advanced topic. Neural networks on the other hand requires complex arrays.

Finally, in order to use the above approaches - you needed lots of training data.