Lecture Notes on Boosting – Draft
Advanced Machine Learning

Prof: Yannet Interian

February 3, 2017

Required reading: The Elements of Statistical Learning (Ch 10.1, 10.2, 10.3, 10.4, 10.5).
Optional reading: Rest of chapter 10.

1 Introduction

Based on

- The elements of Statistical learning (chapter 10).
- Machine Learning, A probabilistic perspective (chapter 16).

This lecture is going to be mostly about boosting, since you already know about random forest and bagging.

The motivation for boosting was a procedure that combines the outputs of many "weak" classifiers to produce a powerful "committee." Boosting was discover as a way to prove a theorem. The theorem says that you can convert (boost into) a *weak learner* (learner that is right 50% + ϵ) into a *strong learner* (learner that is right 100% - ϵ).

Boosting is similar to bagging in the sense that you learn many classifiers and they vote but the difference is that each learner is going to use they training set with different weights.

Boosting is consider one of the best practical algorithms used in machine learning today. Boosting and DeepLearning where the top rated algorithms in Kaggle in 2016.

Here is a pseudocode for a boosting (adaboost) algorithm.

Algorithm 1 Adaboost

```
1: procedure ADABOOST
2:     Maintain a weight of each point in a training set {(x_i, y_i)}_{i=1}^N
3:     Initialize weights w_i = 1/N, i ∈ [1 ··· N]
4:     for iter = 1 to M do
5:         Apply base learner to weighted examples.
6:         Increase weights of misclassified examples.
7:     end for
8:     Combine models by weighted voting.
9: end procedure
```

- Base learner = weak learner, it is usually a decision tree
- M = number of rounds of boosting. This number should be 50, 100, 500; 1000 is probably too much.
- If your base learner doesn't take weights you can always sample with replacement using weights as probabilities.

2 Building a weighted decision stump

A decision stump is a machine learning model consisting of a one-level decision tree. That is, it is a decision tree with one internal node which is immediately connected to the terminal nodes. A decision stump makes a prediction based on the value of just a single input feature. Decision stumps or simple decision trees are often used as based classifiers for boosting.

Let's $x = (x^1, ..., x^D)$ for each feature x^j we can define a decision stump classifier for two classes is defined by

$$g_{j,\alpha}(x) = \begin{cases} 1, & \text{if } x^j > \alpha \\ -1, & \text{otherwise} \end{cases}$$

where $j \in \{1, \dots D\}$. This is for a continuous variable you can similarly define it for a categorical variable. How would you do that?

How to find the best decision stump for a weighted dataset? One possibility is to minimize the weighted 0-1 loss function or misclassification error. That is to find the j, α pair that satisfies:

$$j^*, \alpha* = arg \min_{j,\alpha} \frac{\sum_{i=1}^{N} w_i \mathbb{1}(y_i \neq g_{j,\alpha}(x_i))}{\sum_{i=1}^{N} w_i}$$

You are finding at the same time a feature and the threshold that will give the best training error. How would you compute this efficiently? Hint: which α should we consider?

Note the $\mathbb{1}_A(x)$ denotes the characteristic / indicator function with is 1 for $x \in A$ and 0 otherwise. In this example $\mathbb{1}(y \neq g_{j,\alpha}(x))$ is 0 for $y = g_{j,\alpha}(x)$ and otherwise is 1.

3 Boosting

3.1 AdaBoost for binary classification

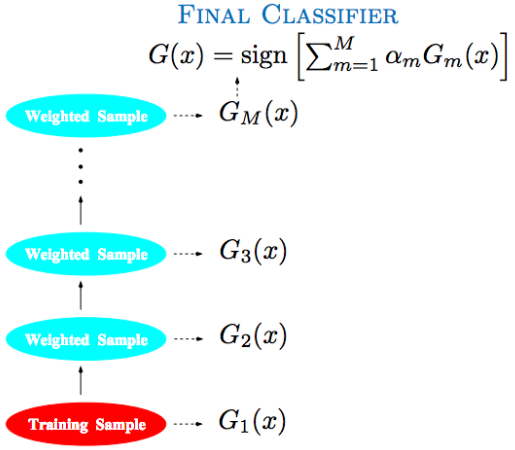
Consider a two-class problem, with the output variable coded as $Y \in \{-1, 1\}$. Given a vector of predictor variables X, a classifier $G(X)$ produces a prediction taking one of the two values $\{-1, 1\}$. The error rate on the training sample is

$$err = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}(y_i \neq G(x_i))$$

A weak classifier is one whose error rate is only slightly better than random guessing. The purpose of boosting is to sequentially apply the weak classification algorithm to repeatedly modified versions of the data, thereby producing a sequence of weak classifiers $G_m(x), m = 1, 2, \cdots, M$.

The predictions from all of them are then combined through a weighted majority vote to produce the final prediction:

Figure 1: Schematic of AdaBoost. Classifiers are trained on weighted versions of the dataset, and then combined to produce a final prediction



$$G(x) = sign\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$$

Here $\alpha_1, \ldots, \alpha_M$ are computed by the boosting algorithm, and weight the contribution of each respective $G_m(x)$. Their effect is to give higher influence to the more accurate classifiers in the sequence. Figure 1 shows a schematic of the AdaBoost procedure.

The data modifications at each boosting step consist of applying weights $w_1, w_2, \ldots, w_N$ to each of the training observations $(x_i, y_i), i = 1, 2, \ldots, N$. Initially all of the weights are set to $w_i = 1/N$, so that the first step simply trains the classifier on the data in the usual manner. For each successive iteration $m = 2, 3, ..., M$ the observation weights are individually modified and the classification algorithm is reapplied to the weighted observations. At step m, those observations that were misclassified by the classifier $G_{m-1}(x)$ induced at the previous step have their weights increased, whereas the weights are decreased for those that were classified correctly. Thus as iterations proceed, observations that are difficult to classify correctly receive ever-increasing influence. Each successive classifier is thereby forced to concentrate on those training observations that are missed by previous ones in the sequence.

Algorithm 2 Adaboost

procedure ADABOOT
  Initialize weights $w_i = \frac{1}{N}, i \in [1 \cdots N]$
  for $m = 1$ to M **do**
    Fit classifier $G_m(x)$ to training data using weights w_i
    Compute

$$err_m = \frac{\sum_{i=1}^{N} w_i \mathbb{1}(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$$

    Compute $\alpha_m = log((1 - err_m)/err_m)$
    Set $w_i = w_i \cdot exp[\alpha_m \cdot \mathbb{1}(y_i \neq G_m(x_i))], i = 1, \ldots, N$
  end for
  Output $G(x) = sign\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$
end procedure

3.2 Boosting fits an Additive Model

Looking at how boosting makes predictions

$$G(x) = sign\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$$

we see that Boosting is a way of fitting an additive expansion in a set of elementary "basis" functions.

More generally, basis expansions take the form:

$$f(x) = \sum_{m=1}^{M} \beta_m b(x, \gamma_m)$$

where β_m are the expansion coefficients and $b(x, \gamma)$ are simple functions characterized by a set of parameters γ.

Typically these models are fit by minimizing a loss function averaged over the training data.

$$\min_{\{\beta_m, \gamma_m\}_{m=1}^{M}} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b(x, \gamma_m)\right) \tag{1}$$

3.3 Forward Stagewise Additive Modeling

Forward stagewise modeling **approximates** the solution of (1) by sequentially adding new basis functions to the expansion like in the following algorithm.

Algorithm 3 Forward Stagewise Additive Modeling

1: **procedure** FORWARD STAGEWISE ADDITIVE MODELING
2: Initialize $f_0(x) = 0$
3: **for** $m = 1$ to M **do**
4: Compute

$$(\beta_m, \gamma_m) = \underset{\beta,\gamma}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i, \gamma))$$

5: Set $f_m(x) = f_{m-1}(x) + \beta_m b(x, \gamma_m)$
6: **end for**
7: **end procedure**

3.4 Exponential Loss and Adaboost

We now show that *Adaboost (Algorithm 1) is equivalent to forward stagewise additive modeling (Algorithm 3) using the exponential loss function*

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

For AdaBoost the basis functions are the individual classifiers $G_m(x) \in \{-1, 1\}$. Using the exponential loss function, one must solve

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

for the classifier G_m and corresponding coefficient β_m to be added at each step. This can be expressed as

$$(\beta_m, \gamma_m) = \underset{\beta,\gamma}{\operatorname{argmin}} \sum_{i=1}^{N} w_i^{(m)} exp[-y_i(\beta G(x_i))] \tag{2}$$

with $w_i^{(m)} = exp(-y_i(f_{m-1}(x_i)))$. Since each $w_i^{(m)}$ depends neither on β nor $G(x)$, it can be regarded as a weight that is applied to each observation. This weight depends on $f_{m_1}(x_i)$, and so the individual weight values change with each iteration m.

The solution to (2) can be obtained in two steps. First, let's show that for any value of $\beta > 0$, the solution to (2) for $G_m(x)$ is

$$G_m = \underset{G}{\operatorname{argmin}} \sum_{i=1}^{N} w_i^{(m)} \mathbb{1}(y_i \neq G(x_i)) \tag{3}$$

which is the classifier that minimizes the weighted error rate in predicting y. This can be found by applying the weak learner to a weighted version of the dataset, with weights $w_i^{(m)}$. This can be easily seen by expressing the criterion in (4) as

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

which in turn can be written as

$$(e^{\beta} - e^{-\beta}) \cdot \sum_{i=1}^{N} w_i^{(m)} \mathbb{1}(y_i \neq G(x_i) + e^{-\beta} \cdot \sum_{i=1}^{N} w_i^{(m)} \tag{4}$$

Plugging the solution G_m into (4) and solving for β one obtains

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

where err_m is the minimized weighted error rate

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

To see this, pugging G_m into (4) and derivate with respect to β.

The approximation is then updated.

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

which causes the weights for the next iteration to be

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

Using the fact that $-y_i G_m(x_i) = 2 \cdot \mathbb{1}(y_i \neq G_m(x_i)) - 1$, the previous equation becomes

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

Hence we conclude that **AdaBoost minimizes the exponential loss criterion via a forward stagewise additive modeling** approach.

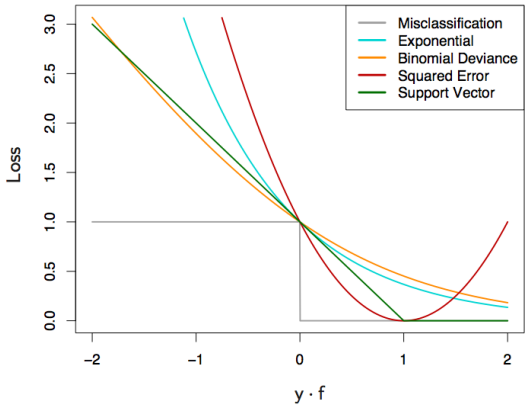
The trouble with the exponential loss is that it puts a lot of weight on misclassified examples. This makes the method very sensitive to outliers (mislabeled examples).

3.5 Loss Functions and Robustness

Loss functions for classification with +1/-1 response are usually a function of the "margin" $y \cdot f(x)$. The margin plays a role similar to the residual $y - f(x)$ in regression. The classification rule $G(x) = sign[f(x)]$ implies that observations with positive margin $y_i f(x_i) > 0$ are classified correctly whereas those with negative margin $y_i f(x_i) < 0$ are misclassified. The decision boundary is defined by $f(x) = 0$. The goal of the classification algorithm is to produce positive margins as frequently as possible. Any loss criterion used for classification should penalize negative margins more heavily than positive ones since positive margin observations are already correctly classified.

Figure (2) shows both the exponential and binomial deviance criteria as a function of the margin $y \cdot f(x)$. Also shown is misclassification loss $L(y, f(x)) = \mathbb{1}(y \cdot f(x) < 0)$, which gives unit penalty for negative margin values, and no penalty at all for positive ones. Both the exponential

Figure 2:



and deviance loss can be viewed as monotone continuous approximations to misclassification loss. They continuously penalize increasingly negative margin values more heavily than they reward increasingly positive ones. The difference between them is in degree. The penalty associated with binomial deviance increases linearly for large increasingly negative margin, whereas the exponential criterion increases the influence of such observations exponentially.

At any point in the training process the exponential criterion concentrates much more influence on observations with large negative margins. Binomial deviance concentrates relatively less influence on such observations, more evenly spreading the influence among all of the data. It is therefore far more robust in noisy settings and especially in situations where there is misspecification of the class labels in the training data. The performance of AdaBoost has been empirically observed to dramatically degrade in such situations.

Here are the equations for the exponential, $L(y, f(x)) = e^{-yf(x)}$, binomial deviance or logloss $L(y, f(x)) = \log(1+e^{-2yf(x)})$ and hinge loss $L(y, f(x)) = \max(0, 1 - yf(x))$.

For a similar discussion for regression see page 349 of ESLII.

4 Gradient Boosting

Rather than deriving new versions of boosting for every different loss function, it is possible to derive a generic version, known as gradient boosting by changing the exponential loss function with other loss functions such as the binomial deviance (logistic regression loss function).

4.1 Boosting Trees

Regression and classification trees partition the space of all joint predictor variables into disjoint regions $R_j, j \dots J$, as represented by the terminal node of the tree. For x in particular region R_j the value of $T(x)$ constant. A tree can be formally expressed as:

$$T(x, \gamma) = \sum_{j=1}^{J} \beta_j \mathbb{1}(x \in R_j)$$

with parameters $\gamma = \{R_j\}_1^J$. How do we find the parameters given some training data?

The boosted tree model is a sum of M trees

$$f_M(x) = \sum_{m=1}^{M} T(x, \gamma_m)$$

induced in a forward stagewise manner. At each step m, we first find the regions $\{R_{jm}\}_{m=1}^J$ by solving

$$\gamma_m = arg \min_{\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + T(x_i, \gamma))$$

Given the regions R_{jm} finding the optimal constants for each region can be done by

$$\beta_{jm} = arg \min_{\beta} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \beta))$$

4.2 Example: Boosting for Regression with Square Loss

To make the problem more clear, let's consider the case of the square loss for regression.

$$\begin{aligned}\gamma_m &= arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + T(x_i, \gamma)) \\ &= arg\min_{\gamma} \sum_{i=1}^{N} (y_i - f_{m-1}(x_i) - T(x_i, \gamma))^2 \\ &= arg\min_{\gamma} \sum_{i=1}^{N} (r_{im} - T(x_i, \gamma))^2\end{aligned}$$

where $r_{im} = y_i - f_{m-1}(x_i)$ is the residual for observation i at iteration m. This problem is a standard regression tree problem that can be solved using CART or any other regression tree method.

$$\begin{aligned}\beta_{jm} &= arg\min_{\beta} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \beta) \\ &= arg\min_{\beta} \sum_{x_i \in R_{jm}} (r_{im} + \beta)^2\end{aligned}$$

This last problem has an analytical solution where β is the average of r_{im} over R_{jm}.

4.3 Intuition: Gradient Descent in Functional Space

TODO: add the gradient descent in functional space. Summary from page 368 of ESL.

4.4 Gradient Boosting Algorithm

Algorithm 4 presents a generic gradient boosting algorithm for regression.

Algorithm 4 Gradient Tree Boosting Algorithm

procedure GRADIENT TREE BOOSTING
  Initialize $f_0(x) = arg\min_\beta \sum_{i=1}^{N} L(y_i, \beta)$
  for $m = 1$ to M **do**
    (a) For $i = 1, 2, \ldots, N$ compute

$$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x_i)=f_{m-1}(x_i)}$$

    (b) Fit a regression tree to the targets r_{im} giving terminal regions $R_{jm}, j = 1 \ldots J_m$
    (c) For $j = 1 \ldots J_m$ compute

$$\beta_{jm} = arg\min_{\beta} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \beta)$$

    (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \beta_{jm} \mathbb{1}(x \in R_{jm})$
  end for
  Output $f_M(x)$
end procedure

4.5 Regularization of boosting

M is the main tuning parameter of the method. Often we pick it by monitoring the performance on a separate validation set, and then stopping once performance starts to decrease.

In practice, better (test set) performance can be obtained by performing partial updates of the form

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

Here $0 < \nu \leq 1$ is a step-size parameter. In practice it is common to use a small value such as $\nu = 0.1$. This is called shrinkage. M and ν are related, smaller values of ν lead to larger values of M

Modern boosting packages such as gbm (`https://cran.r-project.org/web/packages/gbm/gbm.pdf`) in R have other regularization parameters. For example *interaction.depth* is the depth of the base learner tree and $bag.fraction$ is the the fraction of the training set observations randomly selected to train the next tree in the expansion. *interaction.depth* is often set for values between 1 and 8. $bag.fraction$ is often set to 0.5.

4.6 Xgboots

Xgboots is the latest gradient boosting library which is available in Python and R. The implementation of the model supports the features of the scikit-learn and R implementations, with new additions like regularization. Three main forms of gradient boosting are supported:

- Gradient Boosting algorithm also called gradient boosting machine including the learning rate.
- Stochastic Gradient Boosting with sub-sampling at the row, column and column per split levels.
- Regularized Gradient Boosting with both L1 and L2 regularization.

Read this article to learn more about it `http://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/`