CS 536 : Boosting

16:198:536

Consider the problem of classification: we have data generated according to some distribution, and we want to generate a hypothesis to correctly classify this data as much as possible. In the usual way, we can define the true error of a hypothesis and the sample error of a hypothesis (on a data set S) as

$$\begin{aligned} \text{err}(f) &= \mathbb{P}\left(f(\underline{X}) \neq Y\right) \\ \text{err}_S(f) &= \frac{1}{|S|} \sum_{(\underline{x},y)\in S} \mathbb{I}\left\{f(\underline{x}) \neq y\right\}. \end{aligned} \tag{1}$$

The general problem is to take a set of data S and try to generate a hypothesis f such that f achieves minimal true error.

But consider the following problem: if instead of one hypothesis or model f, we had a collection, $f_1, \ldots, f_T$, how could we potentially pool their collective knowledge into one 'meta'-model? An easy way to do this would be via majority vote:

$$F(\underline{x}) = \text{majority}(f_1(\underline{x}), \ldots, f_T(\underline{x})). \tag{2}$$

If we are dealing with binary classification (± 1), this could equivalently be expressed and generalized as

$$F(\underline{x}) = \text{sign}\left(\sum_{t=1}^{T} f_t(\underline{x})\right). \tag{3}$$

We could generalize this in the following way,

$$F(\underline{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t f_t(\underline{x})\right), \tag{4}$$

where the α_t are constants indicating the relative importance or significance of the individual models.

Three questions, at this point:

- Why consider collections of models?
- Where do these models come from?
- What is the best way to combine models into a final prediction?

Why collections? Two related concepts here: one, every model is going to have some weaknesses, so by considering multiple models together, the weaknesses of any one can be made up for in strengths of the others - in this way, there is strength in diversity; two, economics - it can be very difficult and costly to train incredibly complex models, and very frequently simple models can be trained very easily. If we could then combine these simple models to build a highly accurate complex models, we might be able to achieve the same performance more economically.

Where do these models come from? One way to produce a collection of models would be to divide up your data and train a different model on each fraction of the data. This is frequently done with **bagging** - a representative subsample of the data is taken, and a model trained on that subsample, and this is repeated until sufficiently many models are generated. One nice thing about this approach is that it is highly parallelizable.

A second approach for generating these models is a more sequential approach, where each new model tries to make up for the weaknesses of the previous. In particular, if any model misclassifies some points, the next model can be trained to emphasize getting those misclassified points correct. This is connected to the idea of *residuals*. In

regression, the residuals of a model f are taken to be $y_i - f(\underline{x}_i)$; we could consider fitting a new model f' to these residuals, then using $f + f'$ to predict y. This is the main concept behind **boosting** - building sequences of models where each one tries to improve on the performance of the last.

What is the best way to combine these models into a final prediction? This cannot be answered so cleanly - a wide variety of boosting models exist each with its own advantages and disadvantages.

Note: Because boosting algorithms are 'meta'-algorithms, they aren't really paying attention to the features of the data so much as the properties of the classifiers themselves. Garbage classifiers (guessing totally at random) won't be able to be boosted usefully, but classifiers that actually do detect something about the data will. The analysis of boosting will thus focus on the properties and performances of the classifiers / sub-models, rather than the data. As a result of this, I will drop the vector notation on the Xs - they may be vectors, they may not, it just isn't relevant here.

AdaBoost: Weighted Majority Classifier

In this case, we are interested in building classifiers of the form

$$F(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t f_t(x)\right). \tag{5}$$

It is convenient to denote $Q_t(x) = \alpha_1 f_1(x) + \alpha_2 f_2(x) + \ldots \alpha_t f_t(x)$, so that $F(x) = \text{sign}(Q_T(x))$.

Define the 'exponential error' at time t to be

$$E_t = \sum_{(x,y)\in S} e^{-yQ_t(x)}. \tag{6}$$

Note, if Q_t has the correct sign to classify x, $yQ_t(x)$ will be positive ($e^{-yQ_t(x)} < 1$), and $yQ_t(x)$ will be negative if it has the wrong sign to classify x ($e^{-yQ_t(x)} > 1$). Correctly classifying all points in S will produce a smaller error than misclassifying some of the points, and we can real this exponential error back to the 'real' error of the classifier $\text{sign}(Q_t(x))$ by

$$\sum_{(x,y)\in S} \mathbb{I}\left\{\text{sign}(Q_t(x)) \neq y\right\} \leq E_t, \tag{7}$$

or

$$\text{err}_S(\text{sign}(Q_t)) \leq \frac{1}{|S|} E_t. \tag{8}$$

Consider the following, effectively working backwards: suppose you had derived t models $f_1, \ldots, f_t$ and their weights, $\alpha_1, \ldots, \alpha_t$, and you developed a new model f_{t+1} you wanted to include. How could you determine the appropriate weight α_{t+1}?

Consider trying to find α to minimize

$$E_{t+1} = \sum_{(x,y)\in S} e^{-yQ_{t+1}(x)} = \sum_{(x,y)\in S} e^{-yQ_t(x)} e^{-y\alpha f_{t+1}(x)}. \tag{9}$$

If the weak models are restricted to only output ± 1, then for any point f_{t+1} correctly classifies, we have $yf_{t+1}(x) = 1$, otherwise $yf_{t+1}(x) = -1$ for incorrectly classified points. We can partition the error then as

$$E_{t+1} = \left[\sum_{(x,y)\in S: f_{t+1}(x)=y} e^{-yQ_t(x)}\right] e^{-\alpha} + \left[\sum_{(x,y)\in S: f_{t+1}(x)\neq y} e^{-yQ_t(x)}\right] e^{\alpha}. \tag{10}$$

It's convenient to introduce the following, the 'good' weight of f_{t+1} (weight of correctly classified points) and the 'bad' weight - the weight of f_{t+1} for incorrectly classified points:

$$\begin{aligned} B_{t+1} &= \sum_{(x,y)\in S: f_{t+1}(x)\neq y} e^{-yQ_t(x)} \\ G_{t+1} &= \sum_{(x,y)\in S: f_{t+1}(x)=y} e^{-yQ_t(x)}, \end{aligned} \tag{11}$$

so that

$$E_{t+1} = G_{t+1}e^{-\alpha} + B_{t+1}e^{\alpha}. \tag{12}$$

To find the optimal α_{t+1}, we can consider optimizing the above with respect to α. This yields a solution of

$$\alpha_{t+1} = \frac{1}{2}\ln\left(\frac{G_{t+1}}{B_{t+1}}\right). \tag{13}$$

This choice leads to an optimal error at time $t+1$ of

$$E_{t+1} = 2\sqrt{G_{t+1}B_{t+1}}. \tag{14}$$

But how small is this error?

Note additionally the following, that $G_{t+1} + B_{t+1} = E_t$, that is G_{t+1} and B_{t+1} represent a partitioning of the error at time t without the $t+1$-th model. Hence we have

$$E_{t+1} = 2\sqrt{(E_t - B_{t+1})B_{t+1}} = 2E_t\sqrt{\left(1 - \frac{B_{t+1}}{E_t}\right)\left(\frac{B_{t+1}}{E_t}\right)}. \tag{15}$$

The above is maximized the closer B_{t+1}/E_t is to $1/2$ - that is, the error is maximized when f_{t+1} partitions the error at time t equally between the correctly and incorrectly classified points. In this case, we would get that $E_{t+1} = E_t$ and there is no gain. To minimize the error at time $t+1$, we would like to choose f_{t+1} to minimize B_{t+1}/E_t, or effectively to solve

$$f_{t+1} = \text{argmin}_f \frac{\sum_{(x,y)\in S} e^{-yQ_t(x)}\mathbb{I}\{f(x)\neq y\}}{\sum_{(x,y)\in S} e^{-yQ_t(x)}}. \tag{16}$$

In other words, f_{t+1} needs to *minimize the weighted error on the data set S, where the weight on point (x,y) is given by $e^{-yQ_t(x)}$*. (Note, if $t=0$, we take $Q_t(x) = 0$.) Given a weighting function w, the *weighted sample error* of f is given by

$$\text{err}_{w,S}(f) = \frac{\sum_{(\underline{x},y)\in S}\mathbb{I}\{f(\underline{x})\neq y\}w(\underline{x})}{\sum_{(\underline{x},y)\in S} w(\underline{x})}. \tag{17}$$

To summarize, using the notation that $w(\underline{x}) = e^{-yQ_t(\underline{x})}$, we have the following: given a new model f_{t+1}, we want to add it to the aggregate model with weight

$$\alpha_{t+1} = \frac{1}{2}\log\left(\frac{1-\text{err}_{w,S}(f_{t+1})}{\text{err}_{w,S}(f_{t+1})}\right). \tag{18}$$

No matter what the model f_{t+1} is, this is the optimal weight for including it in the ensemble classifier (to minimize the exponential error). And this will lead to an overall exponential error given by

$$E_{t+1} = 2E_t\sqrt{\text{err}_{w,S}(f_{t+1})(1-\text{err}_{w,S}(f_{t+1}))}. \tag{19}$$

This holds for any f_{t+1}, so to minimize E_{t+1} as much as possible, we want $\text{err}_{w,S}(f_{t+1})$ as small as possible. This gives us the classical AdaBoost algorithm for a weighted majority classifier, adapting the above into the language of weights and weighted error:

AdaBoost for Weighted Majority Classifiers: Given a data set S, execute the following steps:

- Initialize the weights with $w_1(x) = 1$ for all $(x, y) \in S$.
- At time t, find f_t to minimize $\text{err}_{w_t,S}(f)$.
- For f_t, define the weight for this classifier as

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \text{err}_{w_t,S}(f_t)}{\text{err}_{w_t,S}(f_t)} \right). \tag{20}$$

- Update the weights on the data:

$$w_{t+1}(x) = e^{-yQ_t(x)} = e^{-yQ_{t-1}(x)} e^{-y\alpha_t f_t(x)} = w_t(x) e^{-y\alpha_t f_t(x)}. \tag{21}$$

- Iterate this to generate models $f_1, \ldots, f_T$, and corresponding $\alpha_1, \ldots, \alpha_T$.
- Construct the final model:

$$F(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t f_t(x) \right). \tag{22}$$

How good is the final model? It's convenient to define

$$\epsilon_t = \text{err}_{w_t,S}(f_t). \tag{23}$$

We have then that since $B_t/E_{t-1} = \epsilon_t$,

$$\begin{aligned} E_T &= 2E_{T-1}\sqrt{(1-\epsilon_T)(\epsilon_T)}, \\ &= m \prod_{t=1}^{T} \sqrt{4(1-\epsilon_t)\epsilon_t}. \end{aligned} \tag{24}$$

Note that for each t, we should have that $\epsilon_t \leq 1/2$ - if this were not true, a better classifier could be recovered taking $-f_t$. Suppose that $\epsilon_t \leq 1/2 - \gamma_t$, where γ_t represents the 'slack'. The above gives us

$$\begin{aligned} E_T &= m \prod_{t=1}^{T} \sqrt{4(1/2+\gamma_t)(1/2-\gamma_t)} \\ &= m \prod_{t=1}^{T} \sqrt{1-4\gamma_t^2} \\ &\leq m \prod_{t=1}^{T} \sqrt{e^{-4\gamma_t^2}} \\ &\leq m e^{-2\sum_{t=1}^{T} \gamma_t^2}. \end{aligned} \tag{25}$$

This, combined with the previous result relating the exponential sample error to the usual sample error, gives us the following result:

Let $F(x) = \text{sign}(\alpha_1 f_1(x) + \alpha_2 f_2(x) + \ldots + \alpha_T f_T(x))$, generated in the above way. If we have that $\text{err}_{w_t,S}(f_t) \leq 1/2 - \gamma$ for each $t = 1, \ldots, T$, then

$$\text{err}_S(F) \leq \frac{1}{m} E_T \leq e^{-2T\gamma^2}. \tag{26}$$

Gradient Boosting

A common alternative to AdaBoost is GradientBoost, which goes as follows: Again, we want to take a given model F and try to improve it in a greedy sort of way, layering a new model on top of it to make up for its deficiencies. Given a model F and a data set, suppose the total loss or error is given by

$$E = \sum_{i=1}^{m} L(y^i, F(\underline{x}^i)), \tag{27}$$

where $L(y, f(x))$ is a generic loss function comparing the true value y to the predicted value $f(x)$. For regression problems this might be the squared error; for classification problems this might be the exponential error as above.

It's convenient to imagine all the predicted $F(\underline{x}^i)$ values summarized into a single m-component vector $\underline{F}$, so $E(\underline{F}) = \sum_i L(y^i, F_i)$. In that way, viewing the error as a function of these predictions, we are essentially asking the following question:

Given a vector $\underline{F}$, and a function $E(\underline{F})$, can we construct a new vector $\underline{F}'$ with a smaller value of E?

A natural approach to this would be a gradient descent approach, i.e., take

$$\underline{F}' = \underline{F} - \lambda \nabla_{\underline{F}} E(\underline{F}). \tag{28}$$

Note, the gradient here is an m-dimensional vector of derivatives *with respect to the predicted values.*

This kind of updating would potentially allow us to make new predictions for each $\underline{x}^i$ value that reduced the overall error, and eventually reach a minimum of error.

Two problems immediately present themselves, however:

- If we are repeatedly tweaking the predicted values to try to reduce the error, this is at high risk for overfitting. Much like in AdaBoost, while we will be able to achieve very low training error, we want to make sure that this will be able to generalize.
- Specifying how to change individual predictions of the model does not necessarily specify how to change the model itself. For instance, if gradient descent here tells how to modify the prediction for $F(3)$ and $F(4)$, it doesn't really say anything about how to modify the prediction for $F(3.5)$.

To address the second concern, we take the following approach: compute the gradient of the error with respect to the predictions, and then use these *as the training data for another model.* For any i, define

$$r^i = \left[\nabla_{\underline{F}} E(\underline{F})\right]_i = \frac{\partial}{\partial F_i} L(y^i, F_i). \tag{29}$$

Taking the data set $\{(\underline{x}^i, r^i)\}_{i=1,\ldots,m}$, fit a new model f to these 'pseudo-residuals', so that

$$\underline{f} = [f(\underline{x}^1), f(\underline{x}^2), \ldots, f(\underline{x}^m)] \approx \nabla_{\underline{F}} E(\underline{F}). \tag{30}$$

Using the function f to approximate the gradient, we can ask how far to move in that direction, how far to tweak the individual predictions of F to try to reduce the overall error. Define the one-dimensional optimization problem:

$$\lambda^* = \text{argmin}_\lambda \; E(\underline{F} + \lambda \underline{f}) = \text{argmin}_\lambda \; \sum_{i=1}^{m} L(y^i, F(\underline{x}^i) + \lambda f(\underline{x}^i)). \tag{31}$$

Having determined the optimal distance to move to modify the predictions of f, we can then construct a new model based on

$$F_{\text{new}}(\underline{x}) = F(\underline{x}) + \lambda^* f(\underline{x}). \tag{32}$$

Iterating this process gives the following gradient boosting algorithm:

The Gradient Boosting Algorithm:

- Train a model f_1 on the data to minimize $\sum_i L(y^i, f_1(\underline{x}^i))$. Take $\lambda_1 = 1$.
- At any time $t \geq 1$, with $F_t = \lambda_1 f_1 + \ldots + \lambda_t f_t$:
  - Define the pseudo-residuals for each $\underline{x}^i, y^i$

$$r_t^i = \frac{\partial}{\partial F_t(\underline{x}_i)} L(y^i, F_t(\underline{x}^i)). \tag{33}$$

  - Fit a model f_{t+1} to the data $\{(\underline{x}^i, r_t^i)\}_{i=1,\ldots,m}$.
  - Solve or estimate λ_{t+1} as the solution to

$$\min_{\lambda} \sum_{i=1}^{m} L(y^i, F_t(\underline{x}^i) + \lambda f_{t+1}(\underline{x}^i)). \tag{34}$$

  - Define the new model $F_{t+1} = F_t + \lambda_{t+1} f_{t+1}$.

Some notes on implementation: as with AdaBoost, you need to be careful that the supplemental models are adding something new. For instance if at each step you are only doing linear regression on the pseudo-residuals, then the ensemble model will be a sum of linear models and therefore linear - which means you are unlikely to get any improvement over the original best linear model. Frequently, gradient boosting is applied to decision tree ensembles (for instance CART trees for regression). Techniques like random subsampling the data and regularization on tree depth can additionally help prevent overfitting for each individual model and ensure diversity of your models.

Early Termination

The previous results indicate that boosting can be quite powerful in terms of turning weak classifiers into classifiers that have very lower sample error. But a big risk here is over fitting - that the boosted classifier fits very tightly to the training data, but does not generalize well. How can this be avoided?

The key point of control that we have here is T, the number of classifiers we build in the ensemble model. The above indicates that too many risks overfitting, but of course too few will not take advantage of the whole notion of boosting. The easiest way to determine good values of T is by *early termination*: setting aside a portion of the data for training, and a portion of the data for testing, we can look at how the error of the current ensemble model on the testing data changes as a function of t. Initially it should decrease, but past a certain point (with enough models in the ensemble) the boosted model will overfit to the training data and we will see a corresponding increase in the testing data. The optimal T should be taken at this minimum.