# Lab 08 - Boosting

[Sicily Xie]

2022-11-16

## Review

- **Boosting** is another method to "train" a classifier. The concept is to reweigh observations after each model fitting based on the prediction accuracy for that observation as well as the overall strength of that fitting. This fit-and-reweigh procedure is iterated and the results from the fits are aggregated to produce the final estimation/classification rule.
- The procedure for **Adaboost** (adaptive boosting) for binary response $Y_i \in \{-1, 1\},\ i = 1, \ldots, n$ can be understood as follows:

  1. Set $j = 1$. Build a classifier $T_j(\boldsymbol{x})$ using **equal weight** $w_i = 1/n$ for all observations. In this course we have always been fitting models this way.
  2. After obtaining the fit, identify two quantities:
     1. A measure of the performance of the classifier $T_j$ in (1), i.e. $\alpha_j$ in the lecture slides. Large values of $\alpha_j$ indicate a strong classifier with low misclassification rate.
     2. For each observation, the indicator of whether it has been misclassified by $T_j$, i.e. $I\left(y_i \neq T_j(\boldsymbol{x}_i)\right)$.

     Now reweigh each observation: New $w_i$ = Old $w_i \times \exp\left\{\alpha_j I\left(y_i \neq T_j(\boldsymbol{x}_i)\right)\right\}$. The idea here is that the weights of correctly classified observations remain unchanged, while **the weights for misclassified observations change according to the strength of the classifier** $T_j$ — if $T_j$ has low (high) misclassification rate, the misclassified observations will be adjusted to have much larger (smaller) weights.
  3. $j$ is incremented and the above steps are run again, replacing the old weights by the new weights. Note how this new classifier will adapt to the data — since more weight is given to observations missed by a previous strong classifier, **this new classifier will put more emphasis on these hard-to-classify points** in an attempt to improve the overall fit.
  4. With the above steps being run $K$ times, the final classifier is given by the sign of $\sum_{j=1}^{K} \alpha_j T_j(\boldsymbol{x})$, which takes into account the relative strengths of the classifiers. Strong classifiers obviously have heavier weights.

  This is again a general procedure that is applicable to many classifiers, e.g. logistic classifier, LDA, trees etc.
- From the form of the final classifier it is easy to see that this algorithm fits an **additive model** in a **forward fashion**, i.e. upon fitting the $k$th classifier, *every* classifier before and including the $k$th is used to build the final classifier. It can also be shown that Adaboost attempts to minimize the **exponential loss**, the function of which is

$$L\left(y, G(\boldsymbol{x})\right) = \exp\{-yG(\boldsymbol{x})\}$$

  where $y \in \{-1, 1\}$ is the observed response and $G(\boldsymbol{x})$ is the classifier.

- The exponential loss function penalizes bad mistakes much more heavily than the 0-1 loss function does. Suppose $y = 1$ while two classifiers $G_1$ and $G_2$ give $-1$ and $-5$ respectively. In each case the 0-1 loss assigns a value of 1 (mismatch in sign), but the exponential loss is $e^1 \approx 2.7$ for $G_1$ and $e^5 \approx 148$ for $G_2$! Also note that the exponential loss is non-zero for correct classifications, unlike the 0-1 loss function.
- The population solution for $G(\boldsymbol{x})$ that minimizes the exponential loss can be shown to be

$$\frac{1}{2}\log\left(\frac{\mathbb{P}(Y=1|\boldsymbol{X}=\boldsymbol{x})}{\mathbb{P}(Y=-1|\boldsymbol{X}=\boldsymbol{x})}\right)$$

  as a function of $\boldsymbol{x}$. Since we are using an additive model to estimate

$$eq : form$$

  , the flexibility of the classifiers we use and the underlying mechanism of data generation will determine our ability to fit the model well.

# Questions

In this lab we explore the power of the boosting algorithm and also try to find out when this method could fail.

## Boosting with a "good" data set

Use the file `boston` data in this part. This data set is derived from the Boston Housing data. The class label is `crime` and the two predictors are `lon` and `lat`. We will try to classify neighbourhoods with high crime rate using the spatial coordinates. You will need the `{adabag}` package for the functions below.

```
library(adabag)
library(ggplot2)
bos <- readRDS("boston.RDS")
bad_boost <- readRDS("bad_boost.RDS")
```

We set the classifiers $T_j$ to be **stumps** — a stump is a tree with only one split. To specify this, run the following line:

```
stump <- rpart.control(cp = -1, maxdepth = 1, minsplit = 0, xval = 0)
```

**Q1** Using the function `boosting()`, perform boosting on this data set with 150 trees (iterations). What is the misclassification error rate? Display the confusion matrix.

**Important notes — read before you fit!**

1. In this part set `boo=FALSE` so that the original sample is used in all trees.
2. Make use of the `control` parameter to incorporate the `stump` settings you specified above.
3. Run the line `set.seed(123)` right before running the boosting algorithm in R.

```
set.seed(123)
model <- boosting(class~lon+lat, data=bos, boo=FALSE, mfinal=150,control=stump)
pre <- predict(model,newdata = bos)
confus <- pre$confusion
err <- pre$error
print(confus)
```

```
##                Observed Class
## Predicted Class High Low
##            High  169   8
##            Low    43 286
```

```
print(err)
```

```
## [1] 0.1007905
```

The misclassification rate is 0.1007905.

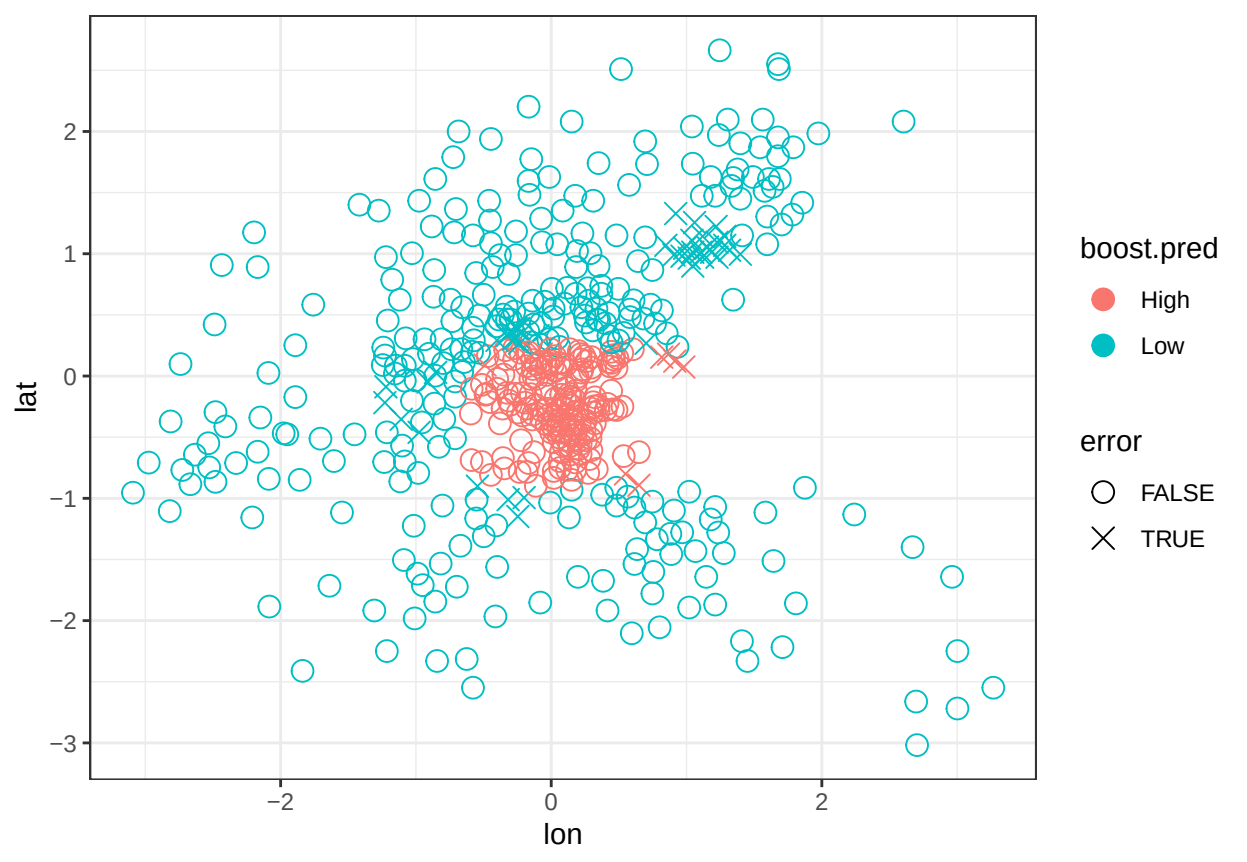
**Q2** Identify and plot the misclassified observations for the boosting algorithm up to the $i$th tree, where $i = 1, 10, 25, 50, 100$ and 150. You may use the provided `plot_boost()` function to generate your plots. **In the output, include the plot for $i = 150$ only.**

**Hint**: The `predict()` function in this package requires the vector of responses (as a factor) to be present. Which parameter of `predict` controls the number of iterations to use?

```
#'  Plot the results of a fitted boosting object
#'
#'  This function plots the class and classification error of a fitted boosting
#'  object to the boston housing data
#'  @param boost fitted boosting object
#'  @param dat data used to fit the boosting object
#'  @param ntree the number of trees of the boosting object to be used in the prediction
plot_boost <- function(boost, dat, ntree = length(boost$tree)) {
  pr <- predict(boost, newdata = dat, newmfinal = ntree)
  dat$boost.pred <- pr$class
  dat$error <- dat$boost.pred != dat$class
  ggplot(data = dat) +
    geom_point(aes(x = lon, y = lat, col = boost.pred, shape = error), size=3.5) +
    scale_shape_manual(values = c(1, 4)) +
    theme_bw()
}
```

```
# some code
for (i in c(1, 10, 25,50,100,150)) {
pred <- predict(model,newdata = bos,newmfinal=i)
print(paste0('tree:',i,',error:',pred$error))
}
```

```
# some code
plot_boost(boost=model,dat=bos,ntree=150)
```




### Boosting with a "bad" data set

**Q3** Repeat **Q1** and **Q2** using the `bad_boost` data. What is the misclassification error rate? Does increasing the number of iterations help? **Be sure to set the seed again.**

```
set.seed(123)
model2 <- boosting(class ~ lon + lat, data = bad_boost, boos = FALSE, mfinal = 150, control = stump)
predict(model2, bad_boost)$confusion
```

```
##                Observed Class
## Predicted Class   a   b
##               a  17   4
##               b  29 100
```

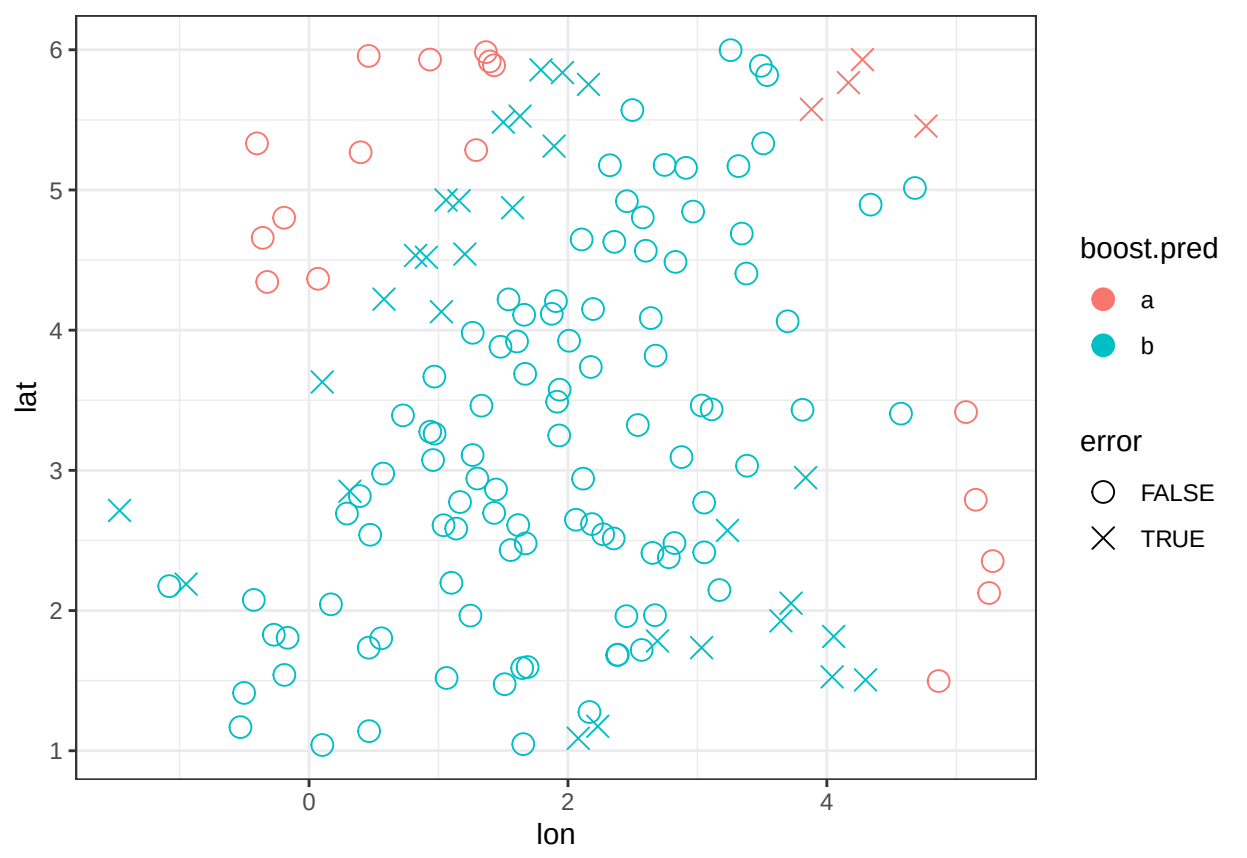
```
predict(model2, bad_boost)$error
```

```
## [1] 0.22
```

The misclassification error rate is 0.22. No, increasing the number of iterations does not help.

```
# some code
for (i in c(1, 10, 25,50,100,150)) {
pred <- predict(model2,newdata = bad_boost,newmfinal=i)
print(paste0('tree:',i,',error:',pred$error))
}
```

```
# some code
plot_boost(boost=model2,dat=bad_boost,ntree=150)
```



**Q4** Now allow the trees to grow a bit deeper by setting `maxdepth=2` in the `rpart.control` command while keeping other parameters fixed at those in **Q1**. Let the algorithm iterate for 100 times. What do you observe?

```
set.seed(123)
stump.new <- rpart.control(cp = -1, maxdepth = 2, minsplit = 0, xval = 0)
model3 <- boosting(class~lon+lat,data=bad_boost,mfinal=100,boo=FALSE,control=stump.new)
predict(model3, bad_boost)$confusion
```

```
##                Observed Class
## Predicted Class   a   b
##               a  46   0
##               b   0 104
```

```
predict(model3, bad_boost)$error
```

```
## [1] 0
```

The misclassification error rate now is 0. The misclassification error decreases as the number of iterations increases, and converges to zero when the number of iterations get close to 80.

## What's the difference?

The response variables in the data set `bad_boost` were in fact generated through the following relationship:

$$\frac{1}{2}\log\left(\frac{\mathbb{P}\left(Y=a|\boldsymbol{X}=\boldsymbol{x}\right)}{1-\mathbb{P}\left(Y=a|\boldsymbol{X}=\boldsymbol{x}\right)}\right)=\left[(x_2-2)_+-(x_1+1)_+\right](1-x_1+x_2)$$

where $(z)_+=\max(z,0)$. Recall that we attempt to estimate

$$\frac{1}{2}\log\left(\frac{\mathbb{P}\left(Y=a|\boldsymbol{X}=\boldsymbol{x}\right)}{1-\mathbb{P}\left(Y=a|\boldsymbol{X}=\boldsymbol{x}\right)}\right)$$

using a **linear combination** of the individual classifiers $T_j(\boldsymbol{x})$ in boosting.

**Q5** $We observed that boosting stumps yields good results on the first data set but not the second. Why is it so?

Since boosting performs well when combing several weak learners into strong learners. If there aren't enough interactions (second data set), boosting would not perform well.

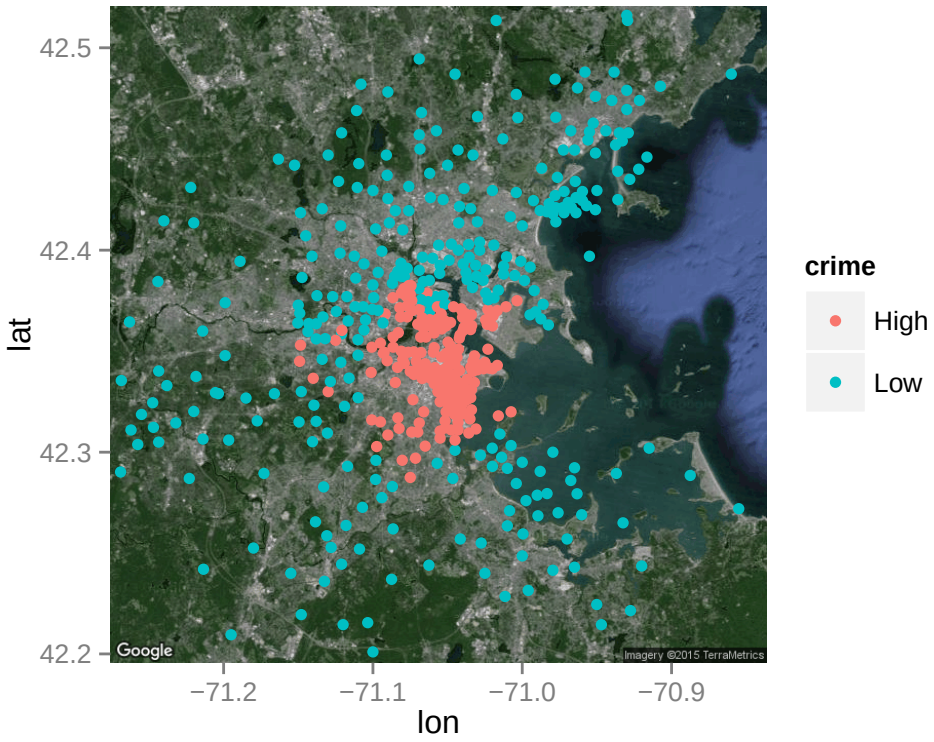



Figure 1: Boston Housing Data - Crime Rate