

# Machine Learning and Computational Statistics

## Homework 6: Ensemble Methods

Zhuoru Lin

**Due: TBD in week after test, at 10pm (Submit via Gradescope)**

**Instructions:** Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g. L<sup>A</sup>T<sub>E</sub>X, L<sup>A</sup>T<sub>E</sub>X, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the [minted](#) package convenient for including source code in your L<sup>A</sup>T<sub>E</sub>X document. If you are using L<sup>A</sup>T<sub>E</sub>X, then the [listings](#) package tends to work better.

### 1 Gradient Boosting Machines

Recall the general gradient boosting algorithm<sup>1</sup>, for a given loss function  $\ell$  and a hypothesis space  $\mathcal{F}$  of regression functions (i.e. functions mapping from the input space to  $\mathbf{R}$ ):

1. Initialize  $f_0(x) = 0$ .

2. For  $m = 1$  to  $M$ :

(a) Compute:

$$\mathbf{g}_m = \left( \left. \frac{\partial}{\partial f(x_j)} \sum_{i=1}^n \ell(y_i, f(x_i)) \right|_{f(x_i)=f_{m-1}(x_i), i=1, \dots, n} \right)_{j=1}^n$$

(b) Fit regression model to  $-\mathbf{g}_m$ :

$$h_m = \arg \min_{h \in \mathcal{F}} \sum_{i=1}^n ((-\mathbf{g}_m)_i - h(x_i))^2.$$

(c) Choose fixed step size  $\nu_m = \nu \in (0, 1]$ , or take

$$\nu_m = \arg \min_{\nu > 0} \sum_{i=1}^n \ell(y_i, f_{m-1}(x_i) + \nu h_m(x_i)).$$

(d) Take the step:

$$f_m(x) = f_{m-1}(x) + \nu_m h_m(x)$$

---

<sup>1</sup>Besides the lecture slides, you can find an accessible discussion of this approach in <http://www.saedsayad.com/docs/gbm2.pdf>, in one of the original references <http://statweb.stanford.edu/~jhf/ftp/trebst.pdf>, and in this review paper <http://web.stanford.edu/~hastie/Papers/buehlmann.pdf>.

3. Return  $f_M$ .

In this problem we'll derive two special cases of the general gradient boosting framework:  $L_2$ -Boosting and BinomialBoost.

1. Consider the regression framework, where  $\mathcal{Y} = \mathbf{R}$ . Suppose our loss function is given by

$$\ell(\hat{y}, y) = \frac{1}{2} (\hat{y} - y)^2,$$

and at the beginning of the  $m$ 'th round of gradient boosting, we have the function  $f_{m-1}(x)$ . Show that the  $h_m$  chosen as the next basis function is given by

$$h_m = \arg \min_{h \in \mathcal{F}} \sum_{i=1}^n [(y_i - f_{m-1}(x_i)) - h(x_i)]^2.$$

In other words, at each stage we find the weak prediction function  $h_m \in \mathcal{F}$  that is the best fit to the residuals from the previous stage. [Hint: Once you understand what's going on, this is a pretty easy problem.]

---

**Answer:**

The negative gradient direction of  $\ell$  can be calculated by:

$$-\frac{\partial \ell(\hat{y}, y)}{\partial \hat{y}} = -(\hat{y} - y) = y - \hat{y} = y - f(x)$$

$h_m$  is chosen to minimize the square error with  $-\frac{\partial \ell(\hat{y}, y)}{\partial \hat{y}}$ , therefore:

$$h_m = \arg \min_{h \in \mathcal{F}} \sum_{i=1}^n [(y_i - f_{m-1}(x_i)) - h(x_i)]^2.$$

2. Now let's consider the classification framework, where  $\mathcal{Y} = \{-1, 1\}$ . In lecture, we noted that AdaBoost corresponds to forward stagewise additive modeling with the exponential loss, and that the exponential loss is not very robust to outliers (i.e. outliers can have a large effect on the final prediction function). Instead, let's consider the logistic loss

$$\ell(m) = \ln(1 + e^{-m}),$$

where  $m = yf(x)$  is the margin. Similar to what we did in the  $L_2$ -Boosting question, write an expression for  $h_m$  as an argmin over  $\mathcal{F}$ .

---

**Answer:** The negative gradient direction is calculated by:

$$-\frac{\partial \ell(y, f(x))}{\partial f(x)} = \frac{ye^{-yf(x)}}{1 + e^{-yf(x)}}$$

$h_m$  is chosen by:

$$\arg \min_{h \in \mathcal{F}} \sum_{i=1}^n \left[ \frac{y_i e^{-y_i f(x_i)}}{1 + e^{-y_i f(x_i)}} - h(x_i) \right]^2$$

## 2 From Margins to Conditional Probabilities<sup>2</sup>

Let's consider the classification setting, in which  $(x_1, y_1), \dots, (x_n, y_n) \in \mathcal{X} \times \{-1, 1\}$  are sampled i.i.d. from some unknown distribution. For a prediction function  $f : \mathcal{X} \rightarrow \mathbf{R}$ , we define the **margin** on an example  $(x, y)$  to be  $m = yf(x)$ . Since our class predictions are given by  $\text{sign}(f(x))$ , we see that a prediction is correct iff  $m(x) > 0$ . We have said we can interpret the magnitude of the margin  $|m(x)|$  as a measure of confidence. However, it is not clear what the “units” of the margin are, so it is hard to interpret the magnitudes beyond saying one prediction is more or less confident than another. In this problem, we investigate how we can translate the margin into a conditional probability, which is much easier to interpret. In other words, we are looking for a mapping  $m(x) \mapsto p(y = 1 | x)$ .

In this problem we will consider margin-based losses. A loss function is a **margin-based loss** if it can be written in terms of the margin  $m = yf(x)$ . We are interested in how we can go from an empirical risk minimizer of a margin loss,  $\hat{f} = \arg \min_{f \in \mathcal{F}} \sum_{i=1}^n \ell(y_i f(x_i))$ , to a conditional probability estimator  $\hat{\pi}(x) \approx p(y = 1 | x)$ . Our approach will be to try to find a way to use the Bayes prediction function<sup>3</sup>  $f^* = \arg \min_f \mathbb{E}_{x,y} [\ell(yf(x))]$  to get the true conditional probability  $p(y = 1 | x)$ , and then apply the same mapping to the empirical risk minimizer. While there is plenty that can go wrong with this “plug-in” approach (primarily, the empirical risk minimizer from a hypothesis space  $\mathcal{F}$  may be a poor estimate for the Bayes prediction function), it is at least well-motivated, and it can work well in practice. And **please note** that we can do better than just hoping for success: if you have enough validation data, you can directly assess how well “calibrated” the predicted probabilities are. This blog post has some discussion of calibration plots: <https://jmetzen.github.io/2015-04-14/calibration.html>.

It turns out it is straightforward to find the Bayes prediction function  $f^*$  for margin losses, at least in terms of the data-generating distribution: For any given  $x \in \mathcal{X}$ , we'll find the best possible prediction  $\hat{y}$ . This will be the  $\hat{y}$  that minimizes

$$\mathbb{E}_y [\ell(y\hat{y}) | x].$$

If we can calculate this  $\hat{y}$  for all  $x \in \mathcal{X}$ , then we will have determined  $f^*(x)$ . We will simply take

$$f^*(x) = \arg \min_{\hat{y}} \mathbb{E}_y [\ell(y\hat{y}) | x].$$

Below we'll calculate  $f^*$  for several loss functions. It will be convenient to let  $\pi(x) = \mathbb{P}(y = 1 | x)$  in the work below.

1. Write  $\mathbb{E}_y [\ell(yf(x)) | x]$  in terms of  $\pi(x)$  and  $\ell(f(x))$ . [Hint: Use the fact that  $y \in \{-1, 1\}$ .]

---

**Answer:**

$$\pi(x)\ell(f(x)) + (1 - \pi(x))\ell(-f(x))$$

---

<sup>2</sup>This problem is based on Section 7.5.3 of Schapire and Freund's book *Boosting: Foundations and Algorithms*.

<sup>3</sup>In this context, the Bayes prediction function is often referred to as the “population minimizer.” In our case, “population” refers to the fact that we are minimizing with respect to the true distribution, rather than a sample. The term “population” arises from the context where we are using a sample to approximate some statistic of an entire population (e.g. a population of people or trees).

2. Show that the Bayes prediction function  $f^*(x)$  for the exponential loss function  $\ell(y, f(x)) = e^{-yf(x)}$  is given by

$$f^*(x) = \frac{1}{2} \ln \left( \frac{\pi(x)}{1 - \pi(x)} \right)$$

and, given the Bayes prediction function  $f^*$ , we can recover the conditional probabilities by

$$\pi(x) = \frac{1}{1 + e^{-2f^*(x)}}.$$

[Hint: Differentiate the expression in the previous problem with respect to  $f(x)$ . To make things a little less confusing, and also to write less, you may find it useful to change variables a bit: Fix an  $x \in \mathcal{X}$ . Then write  $p = \pi(x)$  and  $\hat{y} = f(x)$ . After substituting these into the expression you had for the previous problem, you'll want to find  $\hat{y}$  that minimizes the expression. Use differential calculus. Once you've done it for a single  $x$ , it's easy to write the solution as a function of  $x$ .]

---

**Answer:**

Let  $R$  be the Bayes risk:

$$\begin{aligned} R &= \mathbb{E}_y [\ell(yf(x)) \mid x] = \pi(x)e^{-f(x)} + (1 - \pi(x))e^{f(x)} \\ \frac{\partial R}{\partial f(x)} &= -\pi(x)e^{-f(x)} + (1 + \pi(x))e^{f(x)} \end{aligned} \quad (1)$$

The Bayes risk minimizer should have  $\frac{\partial R}{\partial f(x)} = 0$ .

$$\begin{aligned} \frac{\partial R}{\partial f(x)} = 0 &\iff (1 + \pi(x))e^{f^*(x)} = \pi(x)e^{-f^*(x)} \\ \frac{e^{f^*(x)}}{e^{-f^*(x)}} &= \frac{\pi(x)}{1 + \pi(x)} \\ f^*(x) &= \frac{1}{2} \ln \left( \frac{\pi(x)}{1 + \pi(x)} \right) \end{aligned} \quad (2)$$

Rearrange (1) we get:

$$\pi(x) = \frac{1}{1 + e^{-2f^*(x)}}.$$

3. Show that the Bayes prediction function  $f^*(x)$  for the logistic loss function  $\ell(y, f(x)) = \ln(1 + e^{-yf(x)})$  is given by

$$f^*(x) = \ln\left(\frac{\pi(x)}{1 - \pi(x)}\right)$$

and the conditional probabilities are given by

$$\pi(x) = \frac{1}{1 + e^{-f^*(x)}}.$$

Again, we may assume that  $\pi(x) \in (0, 1)$ .

---

**Answer:**

Let  $R$  be the Bayes risk:

$$R = \mathbb{E}_y [\ell(yf(x)) \mid x] = \pi(x) \ln(1 + e^{-f(x)}) + (1 - \pi(x)) \ln(1 + e^{f(x)})$$

$$\frac{\partial R}{\partial f(x)} = 0 \implies (1 - \pi(x)) \frac{e^{f^*(x)}}{1 + e^{f^*(x)}} = \pi(x) \frac{e^{-f^*(x)}}{1 + e^{-f^*(x)}} \quad (3)$$

$$e^{f^*(x)} = \frac{\pi(x)}{1 - \pi(x)} \quad (4)$$

$$f^*(x) = \ln \frac{\pi(x)}{1 - \pi(x)} \quad (5)$$

Rearrange we get:

$$\pi(x) = \frac{1}{1 + e^{-2f^*(x)}}.$$

4. [Optional] Show that the Bayes prediction function  $f^*(x)$  for the hinge loss function  $\ell(y, f(x)) = \max(0, 1 - yf(x))$  is given by

$$f^*(x) = \text{sign} \left( \pi(x) - \frac{1}{2} \right).$$

Note that it is impossible to recover  $\pi(x)$  from  $f^*(x)$  in this scenario. However, in practice we work with an empirical risk minimizer, from which we may still be able to recover a reasonable estimate for  $\pi(x)$ . An early approach to this problem is known as “Platt scaling”: [https://en.wikipedia.org/wiki/Platt\\_scaling](https://en.wikipedia.org/wiki/Platt_scaling).

### 3 AdaBoost Actually Works [Optional]

#### Introduction

Given training set  $D = \{(x_1, y_1), \dots, (x_n, y_n)\}$ , where  $y_i$ 's are either  $+1$  or  $-1$ , suppose we have a weak learner  $G_t$  at time  $t$  and we will perform  $T$  rounds of AdaBoost. Initialize observation weights uniformly by setting  $W^1 = (w_1^1, \dots, w_n^1)$  with  $w_i^1 = 1/n$  for  $i = 1, 2, \dots, n$ . For  $t = 1, 2, \dots, n$ :

1. Fit the weak learner  $G_t$  at time  $t$  to training set  $D$  with weighting  $W^t$ .
2. Compute the weighted misclassification error:  $\text{err}_t = \sum_D w_i^t 1(G_t(x_i) \neq y_i) / \sum_i w_i^t$
3. Compute the contribution coefficient for the weak learner:  $\alpha_t = \frac{1}{2} \log(\frac{1}{\text{err}_t} - 1)$
4. Update the weights:  $w_i^{t+1} = w_i^t \exp(-\alpha_t y_i G_t(x_i))$

After  $T$  steps, the cumulative contributions of weak learners is  $G(x) = \text{sign}(\sum_{t=1}^T \alpha_t G_t(x))$  as the final output. We will prove that with a reasonable weak learner the error of the output decreases exponentially fast with the number of iterations.

#### Exponential bound on the training loss

More precisely, we will show that the training error  $L(G, D) = \frac{1}{n} \sum_{i=1}^n 1_{\{G(x_i) \neq y_i\}} \leq \exp(-2\gamma^2 T)$  where the error of the weak learner is less than  $1/2 - \gamma$  for some  $\gamma > 0$ . To start, let's denote two cumulative variables: the output at time  $t$  as  $f_t = \sum_{s \leq t} \alpha_s G_s$  and  $Z_t = \frac{1}{n} \sum_{i=1}^n \exp(-y_i f_t(x_i))$ .

1. For any function  $g$ , show that  $1_{\{g(x) \neq y\}} < \exp(-yg(x))$ .

---

#### Answer:

For any function  $g(x)$  we have:

When  $y = g(x)$ ,  $yg(x) = 1$  then:

$$1_{(g(x) \neq y)} \leq e^{-1} = e^{-y^2} = e^{-yg(x)}$$

When  $y \neq g(x)$ ,  $yg(x) = -1$  then:

$$1_{(g(x) \neq y)} \leq e = e^{-yg(x)}$$



2. Use this to show  $L(G, D) < Z_T$

---

**Answer:**

If  $G(x) = y$ , then we must have  $y f_T(x) > 0$ . Therefore:

$$1_{(G(x) \neq y)} = 0 < \exp(-y f_T(x))$$

Similarly if  $G(x) \neq y$ , then we must have  $y f_T(x) < 0$ . Then:

$$1_{(G(x) \neq y)} = 1 < \exp(0) < \exp(-y f_T(x))$$

The statements above hold for all  $G$  and  $f_T$ . Therefore we must have:

$$\sum_{i=1}^n 1_{(G(x_i) \neq y_i)} < \sum_{i=1}^n \exp(-y_i f_T(x_i))$$

3. Show that  $w_i^{t+1} = \exp(-y_i f_t(x_i))$

---

**Answer:**

We show this by strong induction on  $t$ .

Base Case:  $f_0 = \alpha_0 G_0(x_i)$ . Then  $w_i^0 = \exp(-y_i f_0(x_i))$

Induction step: if  $w_i^T = \exp(-y_i f_{T-1}(x_i))$  is true for  $T = 1, 2, 3, \dots, t$ . Then

$$\begin{aligned} w_i^{t+1} &= w_i^t \exp(-\alpha_t y_i G_t(x_i)) \\ &= w_i^0 \exp(-y_i \sum_{T=0}^t \alpha_T G_t(x_i)) \\ &= w_i^0 \exp(-y_i f_t(x_i)) \end{aligned}$$

4. Use part 3 to show  $\frac{Z_{t+1}}{Z_t} = 2\sqrt{\text{err}_{t+1}(1 - \text{err}_{t+1})}$  (Hint: use the definition of weight updates and separate the sum on where  $G_t$  is equal to 1 and  $-1$ .)

---

**Answer:**

By part 3, we can write  $Z_t = \sum_{i=0}^n w_{t+1}$ . Also by the update rule of  $w_t$ , we have  $w_{t+1} = w_t \exp(-\alpha_t y_i G_{t+1}(x_i))$

$$\begin{aligned} \frac{Z_{t+1}}{Z_t} &= \frac{\sum_{i=1}^n w_i^{t+2}}{\sum_{i=1}^n w_i^{t+1}} = \frac{\sum_{i=1}^n w_i^{t+1} \exp(-\alpha_t + 1 y_i G_{t+1}(x_i))}{\sum_{i=1}^n w_i^{t+1}} \\ &= \frac{\exp(-\alpha_{t+1}) \sum_{i=1}^n w_i^{t+1} + (\exp(\alpha_{t+1}) - \exp(-\alpha_{t+1})) \sum_{i=1}^n w_i^{t+1} 1_{G_{t+1}(x_i) \neq y_i}}{\sum_{i=1}^n w_i^{t+1}} \end{aligned}$$

Notice that  $\text{err}_{t+1} = \frac{\sum_{i=1}^n w_i^{t+1} 1_{G_{t+1} \neq y_i}}{\sum_{i=1}^n w_i^{t+1}}$ , and  $\alpha_{t+1} = \frac{1}{2} \ln\left(\frac{1 - \text{err}_{t+1}}{\text{err}_{t+1}}\right)$  therefore:

$$\begin{aligned} \frac{Z_{t+1}}{Z_t} &= e^{-\alpha_{t+1}} + (e^{\alpha_{t+1}} - e^{-\alpha_{t+1}}) \text{err}_{t+1} \\ &= 2\sqrt{(1 - \text{err}_{t+1}) \text{err}_{t+1}} \end{aligned}$$

5. Show that the function  $g(a) = a(1 - a)$  is monotonically increasing on  $[0, 1/2]$ . Show that  $1 - a \leq \exp(-a)$ . And use the assumption on the weak learner to show that  $\frac{Z_{t+1}}{Z_t} \leq \exp(-2\gamma^2)$

---

**Answer:**

$\frac{d(g(a))}{da} = 1 - 2a > 0$  for  $a \in [0, 1/2]$ . Thus  $g(a)$  is monotonically increasing on  $[0, 1/2]$ .

By expanding  $\exp(-a)$  using Maclaurin's series we have:  $\exp(-a) = 1 - a + \frac{a^2}{2} \dots > 1 - a$ .

By Part 4 and definition of  $err_{t+1}$ :

$$\frac{Z_{t+1}}{Z_t} = 2\sqrt{(1 - \frac{1}{2} + \gamma)(\frac{1}{2} - \gamma)} = \sqrt{1 - 4\gamma^2} = \sqrt{(1 - 2\gamma)(1 + 2\gamma)}$$

Since  $\frac{1}{2} > \gamma > 0$ :

$$\frac{Z_{t+1}}{Z_t} < 1 - 2\gamma < 1 - 2\gamma^2 < \exp(-2\gamma^2)$$

6. Conclude the proof!

By part 5, we already show that  $\frac{Z_{t+1}}{Z_t} \leq \exp(-2\gamma^2)$  for all  $t$ . Then:

$$L(G, D) < Z_T = \prod_{t=1}^T \frac{Z_t}{Z_{t-1}} \leq \exp(-2\gamma^2 T)$$

## 4 AdaBoost is FSAM With Exponential Loss [Optional]

The AdaBoost score function  $G(x) = \sum_{t=1}^T \beta_t G_t(x)$  is a linear combination (actually a conic combination) of functions. (The prediction function is, of course, the sign of the score function.) Forward stagewise additive modeling (FSAM) is another approach to fitting a function of this form.

In FSAM, we have a base hypothesis space  $\mathcal{H}$  of real-valued functions  $h : \mathcal{X} \rightarrow \mathbf{R}$  and a loss function  $\ell(y, \hat{y})$ . In FSAM, we attempt to find a linear combination of  $h$ 's in  $\mathcal{H}$  that minimize the empirical risk. The procedure initializes  $f_0(x) = 0$ , and then repeats the following steps for  $t = 1, \dots, T$ :

1.  $(\beta_t, h_t) = \operatorname{argmin}_{\beta \in \mathbf{R}, h \in \mathcal{H}} \sum_{i=1}^n \ell(y_i, f_{t-1}(x_i) + \beta h(x_i))$
2.  $f_t(x) = f_{t-1}(x) + \beta_t h_t(x)$

### Exponential loss and AdaBoost

Consider a generic input space  $\mathcal{X}$ , the classification outcome space  $\mathcal{Y} = \{-1, 1\}$ , the exponential loss function  $\ell(y, f(x)) = \exp(-yf(x))$ , and an arbitrary base hypothesis space  $\mathcal{H}$  consisting of  $\{-1, 1\}$ -valued functions. We will show that FSAM in this setting is equivalent to a version of AdaBoost (Algorithm 1) described below. To get this equivalence, we either need to assume that FSAM chooses nonnegative step sizes, i.e.  $\beta_t \geq 0$ , or we need to assume that  $\mathcal{H}$  is symmetric, in the sense that if  $h \in \mathcal{H}$ , then  $-h \in \mathcal{H}$  as well.

1. Write the first step of FSAM using the exponential loss function. In particular, show that the FSAM optimization problem can be written as a minimization of a weighted exponential loss of the step  $\beta h$ :

$$(\beta_t, h_t) = \operatorname{argmin}_{\beta, h \in \mathcal{H}} \left( \frac{1}{\sum_{i=1}^n w_i^t} \right) \sum_{i=1}^n w_i^t \exp(-y_i \beta h(x_i)),$$

where  $w_i^t = \exp(-y_i f_{t-1}(x_i))$ . (Note that for any  $t$ , if we rescale each of  $w_1^t, \dots, w_n^t$  by the same constant factor, there is no effect on the arg min. Thus the first factor  $(\sum_{i=1}^n w_i^t)^{-1}$  can be dropped. However, we keep it so we can refer to the expression as a **weighted mean**.)

2. Define the weighted 0/1 error of  $h$  at round  $t$  to be

$$\text{err}_t(h) = \left( \frac{1}{\sum_{i=1}^n w_i^t} \right) \sum_{i=1}^n w_i^t 1(y_i \neq h(x_i)).$$

(It's the weights that are specific to round  $t$ .) Show that the weighted exponential loss at round  $t$  can be written in terms of the weighted 0/1 error. Specifically, show that

$$\left( \frac{1}{\sum_{i=1}^n w_i^t} \right) \sum_{i=1}^n w_i^t \exp(-\beta y_i h(x_i)) = e^{-\beta} + (e^{\beta} - e^{-\beta}) \text{err}_t(h).$$

[Hint: Use indicators  $1(h(x_i) \neq y_i)$  and  $1(h(x_i) = y_i)$  to split the summand on the LHS into pieces. Each piece simplifies, since  $y_i, h(x_i) \in \{-1, 1\}$ . Then note that  $1(h(x_i) = y_i) = 1 - 1(h(x_i) \neq y_i)$ .]

3. We now would like to show that for any fixed “step size”  $\beta$ , the optimal “step direction”  $h$ , for which  $\beta h$  minimizes the weighted exponential loss, can be found by minimizing the weighted 0/1 error of  $h$ . But more precisely, show that if  $\beta \geq 0$  then

$$\operatorname{argmin}_{h \in \mathcal{H}} \left( \frac{1}{\sum_{i=1}^n w_i^t} \right) \sum_{i=1}^n w_i^t \exp(-\beta y_i h(x_i)) = \operatorname{argmin}_{h \in \mathcal{H}} \operatorname{err}_t(h).$$

Also show that if  $\beta < 0$  then

$$\operatorname{argmin}_{h \in \mathcal{H}} \left( \frac{1}{\sum_{i=1}^n w_i^t} \right) \sum_{i=1}^n w_i^t \exp(-\beta y_i h(x_i)) = \operatorname{argmin}_{h \in \mathcal{H}} \operatorname{err}_t(-h).$$



4. Show that if  $\mathcal{H}$  is symmetric, in the sense that  $h \in \mathcal{H}$  implies  $-h \in \mathcal{H}$ , then there is always an optimal FSAM step  $(\beta_t, h_t)$  with  $\beta_t \geq 0$ . Thus if we assume that either  $\mathcal{H}$  is symmetric or FSAM chooses nonnegative step sizes, then we can conclude that

$$h_t = \operatorname{argmin}_{h \in \mathcal{H}} \operatorname{err}_t(h)$$

is a solution to  $h_t$  in the minimization problem in the first part, and thus is the FSAM step direction in round  $t$ .

5. Now that we've found  $h_t$ , show that the corresponding optimal step size is given by  $\beta_t = \frac{1}{2} \log \left( \frac{1 - \text{err}_t}{\text{err}_t} \right)$ , where we let  $\text{err}_t = \text{err}_t(h_t)$  as a shorthand. [Hint: You'll need to use some differential calculus. Show that what you've found is a minimum by showing that the function you're differentiating is convex.]

6. Show that

$$w_i^{t+1} = \begin{cases} e^{-\beta_t} w_i^t & \text{if } y_i = h_t(x_i) \\ e^{-\beta_t} w_i^t e^{2\beta_t} & \text{otherwise,} \end{cases}$$

This is the weight update equation from AdaBoost. [Hint: First show that  $w_i^{t+1} = w_i^t \exp(-\beta_t y_i h_t(x_i))$ . Then write  $y_i h_t(x_i)$  in terms of the indicator function  $y_i \neq h_t(x_i)$ .]

7. Let's introduce a specific instance of AdaBoost we'll call "Exact AdaBoost", given in Algorithm 1. The only difference between Exact AdaBoost and AdaBoost is that in Exact AdaBoost,

---

**Algorithm 1:** Exact AdaBoost

---

```

input: Training set  $\mathcal{D} = ((x_1, y_1), \dots, (x_n, y_n)) \in \mathcal{X} \times \{-1, 1\}$ 
 $w_i^1 = 1$  for  $i = 1, \dots, n$  #Initialize weights
for  $t = 1, \dots, T$ :
     $h_t = \arg \min_{h \in \mathcal{H}} \sum_{i=1}^n w_i^t 1(y_i \neq h(x_i))$ 
     $\text{err}_t = \text{err}_t(h_t) = \left( \frac{1}{\sum_{i=1}^n w_i^t} \right) \sum_{i=1}^n w_i^t 1(y_i \neq h(x_i))$ 
     $\alpha_t = \ln \left( \frac{1 - \text{err}_t}{\text{err}_t} \right)$ 
     $w_i^{t+1} = \begin{cases} w_i^t & \text{if } y_i = h_t(x_i) \\ w_i^t e^{\alpha_t} & \text{otherwise,} \end{cases} \quad \text{for } i = 1, \dots, n$ 
return  $f = \sum_{t=1}^T \alpha_t h_t$  #Returns the score function.
(Predictions are  $x \mapsto \text{sign}(f(x))$ ) .
```

---

we require that the base classifier return the best possible  $h \in \mathcal{H}$ , while in AdaBoost we only vaguely stated that the "base learner fits the weighted training data", but there was no requirement that the result be the best possible. Indeed, since a typical base classifier is decision trees, and it's computationally prohibitive to find the best possible tree, Exact AdaBoost is not usually an implementable algorithm. Show that the score functions returned by Exact Adaboost and by FSAM (in our setting) differ only by a constant factor, and of course the hard classifications will be exactly the same.

8. Suppose our ultimate goal is to find the score function returned by FSAM after  $T$  rounds in the context described above. Suppose we only have access to an implementation of Exact AdaBoost described in Algorithm 1, and it returns the score function  $f(x)$ . What would be the score function returned by FSAM?

## 5 [Optional] Decision Tree Implementation

In this problem we'll implement decision trees for both classification and regression. The strategy will be to implement a generic class, called `Decision_Tree`, which we'll supply with the loss function we want to use to make node splitting decisions, as well as the estimator we'll use to come up with the prediction associated with each leaf node. For classification, this prediction could be a vector of probabilities, but for simplicity we'll just consider hard classifications here. We'll work with the classification and regression data sets from Homework #4.

1. [Optional] Complete the class `Decision_Tree`, given in the skeleton code. The intended implementation is as follows: Each object of type `Decision_Tree` represents a single node of the tree. The depth of that node is represented by the variable `self.depth`, with the root node having depth 0. The main job of the fit function is to decide, given the data provided, how to split the node or whether it should remain a leaf node. If the node will split, then the splitting feature and splitting value are recorded, and the left and right subtrees are fit on the relevant portions of the data. Thus tree-building is a recursive procedure. We should have as many `Decision_Tree` objects as there are nodes in the tree. We will not implement pruning here. Some additional details are given in the skeleton code.

---

**Answer:**

```
class Decision_Tree(BaseEstimator):

    def __init__(self, split_loss_function, leaf_value_estimator,
                  depth=0, min_sample=5, max_depth=10):
        """
        Initialize the decision tree classifier

        :param split_loss_function: method for splitting node
        :param leaf_value_estimator: method for estimating leaf value
        :param depth: depth indicator, default value is 0, representing root node
        :param min_sample: an internal node can be splitted only if it contains poi
        :param max_depth: restriction of tree depth.
        """
        self.split_loss_function = split_loss_function
        self.leaf_value_estimator = leaf_value_estimator
        self.depth = depth
        self.min_sample = min_sample
        self.max_depth = max_depth
        #####Simon's Edit#####
        self.is_leaf = None
        self.split_id = None
        self.split_value = None
        self.left = None
        self.right = None
        self.value = None
```

```

self.loss = None
#####
def fit(self, X, y=None):
    """
    This should fit the tree classifier by setting the values self.is_leaf,
    self.split_id (the index of the feature we want to split on, if we're splitting
    self.split_value (the corresponding value of that feature where the split is
    and self.value, which is the prediction value if the tree is a leaf node.
    splitting the node, we should also init self.left and self.right to be Decision
    objects corresponding to the left and right subtrees. These subtrees should
    contain the data that fall to the left and right, respectively, of self.split_value.
    This is a recursive tree building procedure.

    :param X: a numpy array of training data, shape = (n, m)
    :param y: a numpy array of labels, shape = (n, 1)

    :return self
    """
    #If depth is max depth turn into leaf
    if self.depth == self.max_depth:
        self.is_leaf = True
        self.value = self.leaf_value_estimator(y)
        return self
    #If reach minimum sample size turn into leaf
    if len(y) <= self.min_sample:
        self.is_leaf = True
        self.value = self.leaf_value_estimator(y)
        return self
    #If not is_leaf, i.e. in the node, we should create left and right subtree
    #But First we need to decide the self.split_id and self.split_value that minimize
    #Compare with constant prediction of all X
    loss_before = self.split_loss_function(y)
    best_split_value = None
    best_split_id = None
    best_loss = self.split_loss_function(y)
    best_left_X = None
    best_right_X = None
    best_left_y = None
    best_right_y = None
    best_pos = None
    #Concatenate y into X for sorting together
    X = np.concatenate([X, y], 1)
    for i in range(X.shape[1]-1):
        #Note: The last column of X is y now
        X = np.array(sorted(X, key=lambda x: x[i]))
        for split_pos in range(len(X)-1):

```

```

        #:split_pos+1 will include the split_pos data in left_X
        left_X = X[:split_pos+1,:-1]
        right_X = X[split_pos+1:,:-1]
        #you need left_y to be in (n,1) i.e (-1,1) dimension
        left_y = X[:split_pos+1,-1].reshape(-1,1)
        right_y = X[split_pos+1:,-1].reshape(-1,1)
        left_loss = len(left_y)*self.split_loss_function(left_y)/len(y)
        right_loss = len(right_y)*self.split_loss_function(right_y)/len(y)
        #If any choice of splitting feature and splitting position results
        #record following information and discard the old one
        if ((left_loss+right_loss)<best_loss):
            best_split_value = X[split_pos,i]
            best_pos = split_pos
            best_split_id = i
            best_loss = left_loss+right_loss
            best_left_X = left_X
            best_right_X = right_X
            best_left_y = left_y
            best_right_y = right_y
        #Condition when you have a split position that results in better loss
        if best_split_id!=None:
            self.left = Decision_Tree(self.split_loss_function,self.leaf_value_estimator,
                                     self.min_sample,self.max_depth)
            self.right = Decision_Tree(self.split_loss_function,self.leaf_value_estimator,
                                      self.min_sample,self.max_depth)

#         #####Debugging Printint#####
#         print('split feature: x[%s]'%(best_split_id))
#         #print('split pos: %s'%(best_pos))
#         print('split value:%s'%(best_split_value))
#         print('(left size,right size): (%s,%s)'%(len(best_left_X),len(best_right_X)))
#         print('Loss before splited: %.6f'%(loss_before))
#         print('Loss after splited: %.6f'%(best_loss))
#         print()
#         #####
        self.left.fit(best_left_X,best_left_y)
        self.right.fit(best_right_X,best_right_y)
        self.split_id = best_split_id
        self.split_value = best_split_value
        self.loss = best_loss
    else:
        self.is_leaf = True
        self.value = self.leaf_value_estimator(y)

    #print(self.split_id, self.split_value)
    return self

```



```

def predict_instance(self, instance):
    """
    Predict label by decision tree

    :param instance: a numpy array with new data, shape (1, m)

    :return whatever is returned by leaf_value_estimator for leaf containing in
    """
    if self.is_leaf:
        return self.value
    if instance[self.split_id] <= self.split_value:
        #print('split at x[%s]<=%s'%(self.split_id, self.split_value))
        return self.left.predict_instance(instance)
    else:
        return self.right.predict_instance(instance)

```

2. [Optional] Complete either the `compute_entropy` or `compute_gini` functions. Run the code provided that builds trees for the two-dimensional classification data. Include the results. For debugging, you may want to compare results with `sklearn`'s decision tree. For visualization, you'll need to install `graphviz`.

---

**Answer:**

**The following are codes for entropy and gini index loss and the result of decision tree implementation on classification data**

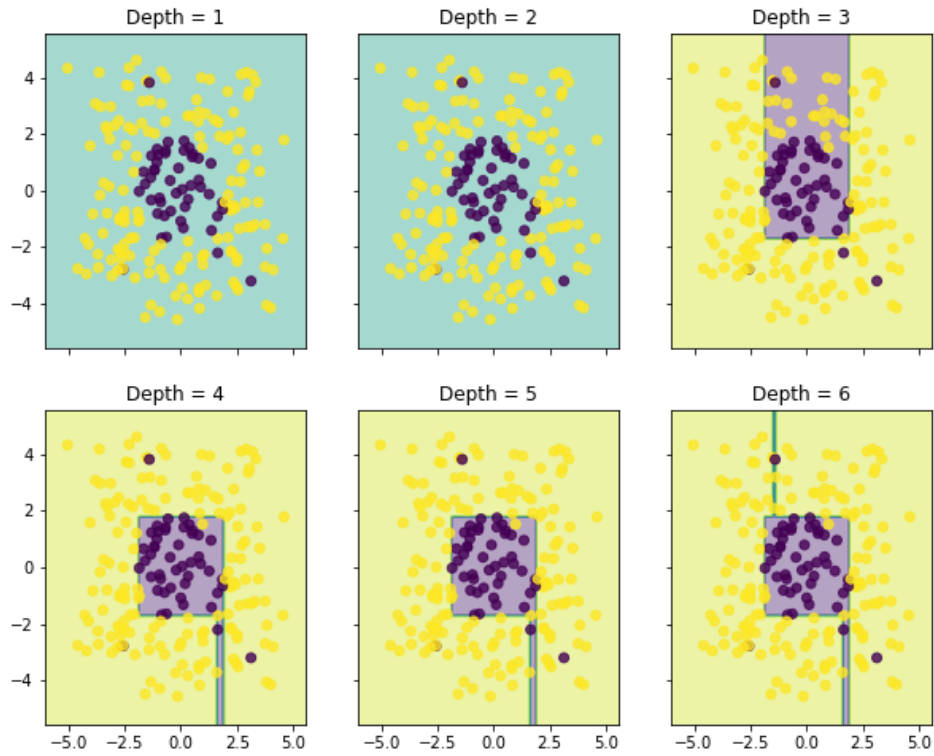
```
def compute_entropy(label_array):
    """
    Calculate the entropy of given label list

    :param label_array: a numpy array of labels shape = (n, 1)
    :return entropy: entropy value
    """
    n_classes = np.unique(label_array)
    entropy = 0
    for label in n_classes:
        p = np.sum(label_array==label)/float(len(label_array))
        entropy += -p*np.log(p)
    return entropy

def compute_gini(label_array):
    """
    Calculate the gini index of label list

    :param label_array: a numpy array of labels shape = (n, 1)
    :return gini: gini index value
    """
    n_classes = np.unique(label_array)
    gini = 0
    for label in n_classes:
        p = np.sum(label_array==label)/len(label_array)
        gini += p*(1-p)
    return gini
```

Figure 1: Decition Tree with entropy loss implemented



3. [Optional] Complete the function `mean_absolute_deviation_around_median` (MAE). Use the code provided to fit the `Regression_Tree` to the `krr` dataset using both the MAE loss and median predictions. Include the plots for the 6 fits.

---

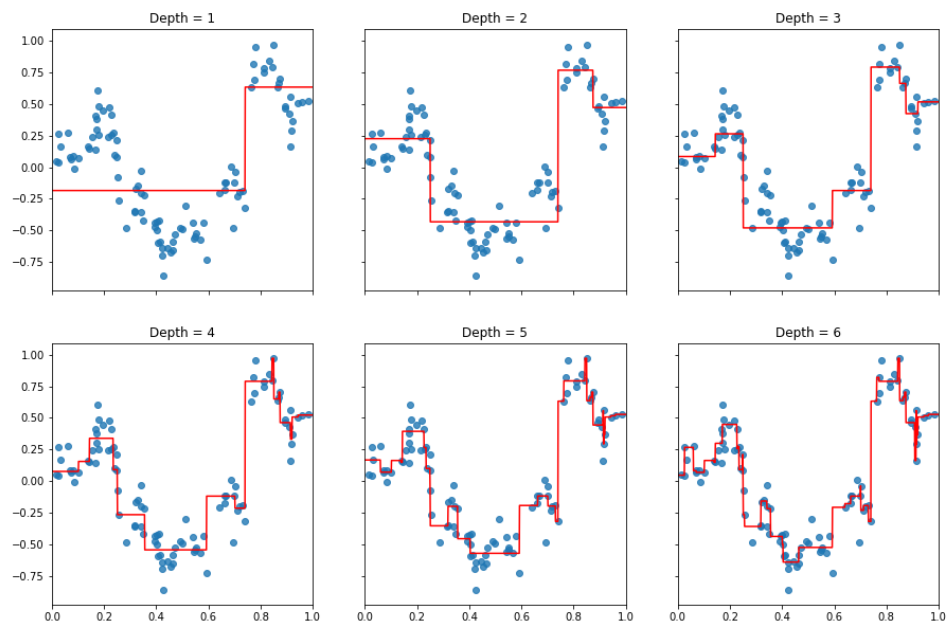
**Answer:**

**The following are codes for MAE loss and the results of decision tree implemented on regression data**

```
# Regression Tree Specific Code
def mean_absolute_deviation_around_median(y):
    """
    Calculate the mean absolute deviation around the median of a given target list

    :param y: a numpy array of targets shape = (n, 1)
    :return mae
    """
    # Your code goes here
    median = np.median(y)
    mae = np.mean(np.abs(y - median))
    return mae
```

Figure 2: Decision Tree regression with MAE loss implemented



## 6 Gradient Boosting Implementation

This method goes by many names, including gradient boosting machines (GBM), generalized boosting models (GBM), AnyBoost, and gradient boosted regression trees (GBRT), among others. Although one of the nice aspects of gradient boosting is that it can be applied to any problem with a subdifferentiable loss function, here we'll keep things simple and consider the standard regression setting with square loss.

1. Complete the `gradient_boosting` class. As the base regression algorithm, you should use the regression tree from the previous problem, if you completed it. Otherwise, you may use `sklearn`'s regression tree. You should use the square loss for the tree splitting rule and the mean function for the leaf prediction rule. Run the code provided to build gradient boosting models on the classification and regression data sets, and include the plots generated. Note that we are using square loss to fit the classification data, as well as the regression data.

---

Answer:

Result:

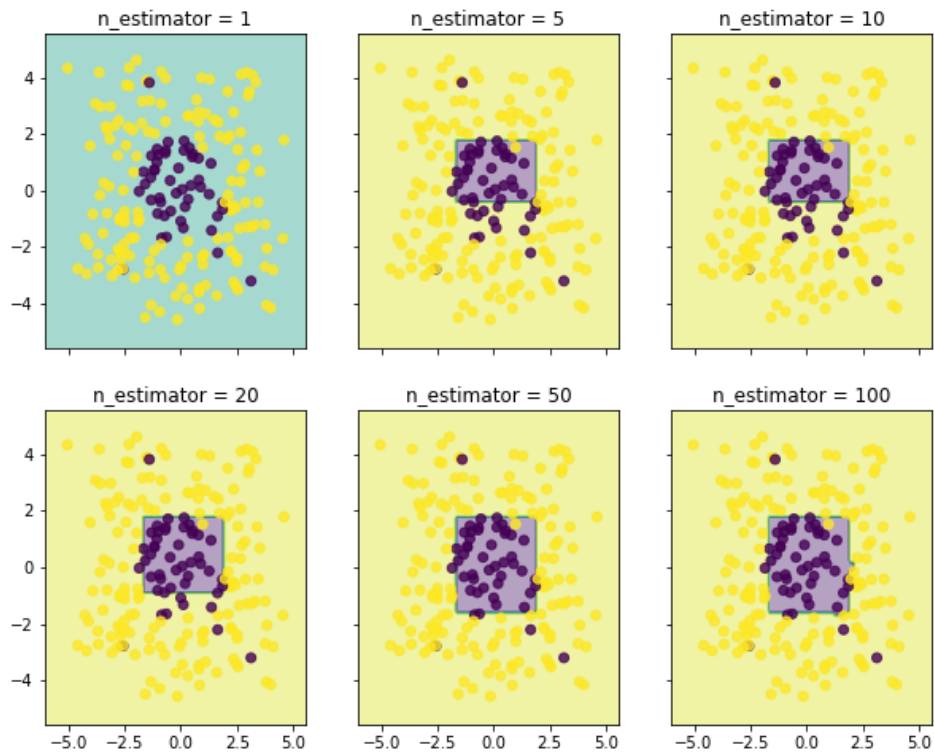


Figure 3: Gradient Boosting Tree with l2-loss

Following is my result and code for GBM:

```
class gradient_boosting():  
    """  
    Gradient Boosting regressor class  
    :method fit: fitting model  
    """  
    def __init__(self, n_estimator, pseudo_residual_func,\br/>                  learning_rate=0.1, min_sample=5, max_depth=3):  
        """
```

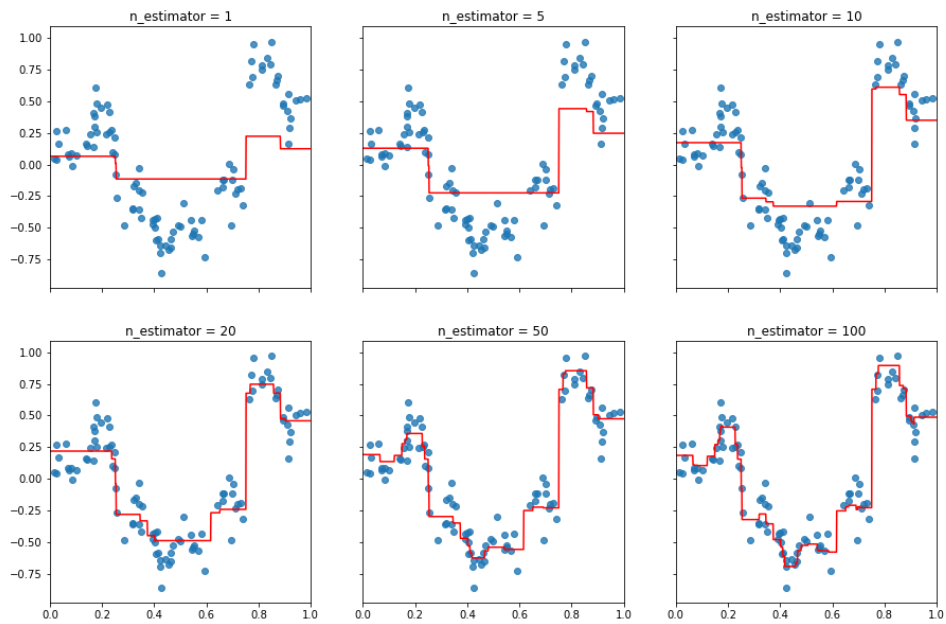


Figure 4: Gradient Boosting Regression

*Initialize gradient boosting class*

```
:param n_estimator: number of estimators (i.e. number of rounds of gradient bo
:pseudo_residual_func: function used for computing pseudo-residual
:param learning_rate: step size of gradient descent
'''

self.n_estimator = n_estimator
self.pseudo_residual_func = pseudo_residual_func
self.learning_rate = learning_rate
self.min_sample = min_sample
self.max_depth = max_depth
#####
#Note: Should be optimized so that the model does not waste memory
#list for saving hm for each step and the
self.base_models = []
self.f0 = None
#####
```



```

def fit(self, train_data, train_target):
    """
    Fit gradient boosting model
    """
    #Fit base model f0
    #Note: My f0 may looks like others h1 since I did not initialized f0=0
    self.f0 = DecisionTreeRegressor(max_depth=self.max_depth,\
                                    min_samples_leaf=self.min_sample)
    self.f0.fit(train_data,train_target)
    for step in range(self.n_estimator):
        step_prediction = self.learning_rate*self.f0.predict(train_data)
        #If there is not model in list directly calculate residuals with f0
        if len(self.base_models)==0:
            residuals = self.pseudo_residual_func(train_target.reshape(-1),\
                                                  step_prediction)

        else:
            for i in range(len(self.base_models)):
                #Calculate prediction using weight sum of predictions of hm_s
                step_prediction += \
                    self.learning_rate*self.base_models[i].predict(train_data)
            #Calculate residuals with fm(x)
            #Notice: sometimes y_train is of (-1,1) shape we need to reshape to (-1)
            residuals = self.pseudo_residual_func(train_target.reshape(-1)\
                                                  ,step_prediction)

        #Fit hm to residuals, which is some negative gradient direction
        hm = DecisionTreeRegressor(max_depth=self.max_depth,\
                                   min_samples_leaf=self.min_sample)
        hm.fit(train_data,residuals)
        #Update hm list
        self.base_models.append(hm)

def predict(self, test_data):
    """
    Predict value
    """
    step_prediction = 0.2*self.f0.predict(test_data)
    for i in range(len(self.base_models)):
        #Get fm(x) by summation of base predictions
        step_prediction += self.learning_rate*\
            self.base_models[i].predict(test_data)
    return step_prediction

```

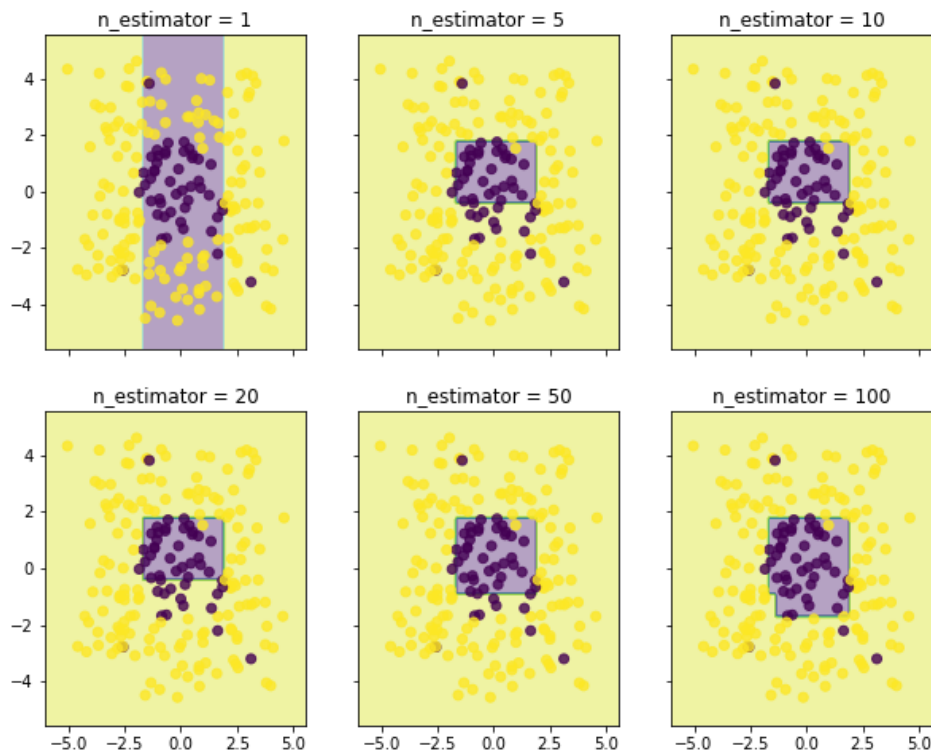
2. [Optional] Repeat the previous runs on the classification data set, but use a different classification loss, such as logistic loss or hinge loss. Include the new code and plots of your results. Note that you should still use the same regression tree settings for the base regression algorithm.

---

**Answer:**

Result:

Figure 5: Gradient Boosting Tree with logistic lost



```
def pseudo_residual_logistic(train_target, train_predict):  
    y = train_target  
    fx = train_predict  
    m = y*fx  
    return (y*np.exp(-m)) / (1+np.exp(-m))
```