Statistical Machine Learning (W4400)
Fall 2015
https://courseworks.columbia.edu

John P. Cunningham
jpc2181@columbia.edu
Xiaoou Li
xiaoou@stat.columbia.edu
Chris Mulligan
clm2186@columbia.edu

Homework 3

Due: Tuesday 27 October 2015

Students are encouraged to work together, but homework write-ups must be done individually and must be entirely the author's own work. Homework is due at the **beginning** of the class for which it is due. **Late homework will not be accepted under any circumstances.** To receive full credit, students must thoroughly explain how they arrived at their solutions and include the following information on their homeworks: name, UNI, homework number (e.g., HW03), and class (STAT W4400). All homework must be turned in online through Courseworks in PDF format, have a .pdf extension, and be less than 4MB. If programming is part of the assignment, the code must be turned in in one or more .R files. Homeworks not adhering to these requirements will receive no credit.

1. **Boosting** (70 points)

The objective of this problem is to implement the AdaBoost algorithm. We will test the algorithm on handwritten digits from the USPS data set.

AdaBoost: Assume we are given a training sample $(\mathbf{x}^{(i)}, y_i), i = 1, ..., n$, where $\mathbf{x}^{(i)}$ are data values in $\mathbb{R}^d$ and $y_i \in \{-1, +1\}$ are class labels. Along with the training data, we provide the algorithm with a training routine for some classifier c (the "weak learner"). Here is the AdaBoost algorithm for the two-class problem:

```
1. Initialize weights: $w_i = \frac{1}{n}$
2. for $b = 1, ..., B$
    (a) Train a weak learner $c_b$ on the weighted training data.
    (b) Compute error: $\epsilon_b := \frac{\sum_{i=1}^n w_i \mathbb{I}\{y_i \neq c_b(\mathbf{x}^{(i)})\}}{\sum_{i=1}^n w_i}$
    (c) Compute voting weights: $\alpha_b = \log\left(\frac{1-\epsilon_b}{\epsilon_b}\right)$
    (d) Recompute weights: $w_i = w_i \exp\left(\alpha_b \mathbb{I}\{y_i \neq c_b(\mathbf{x}^{(i)})\}\right)$
3. Return classifier $\hat{c}_B(\mathbf{x}^{(i)}) = \text{sgn}\left(\sum_{b=1}^B \alpha_b c_b(\mathbf{x}^{(i)})\right)$
```

Decision stumps: Recall that a stump classifier c is defined by

$$c(\mathbf{x}|j, \theta, m) := \begin{cases} +m & x_j > \theta \\ -m & \text{otherwise.} \end{cases} \tag{1}$$

Since the stump ignores all entries of $\mathbf{x}$ except x_j, it is equivalent to a linear classifier defined by an affine hyperplane. The plane is orthogonal to the jth axis, with which it intersects at $x_j = \theta$. The orientation of the hyperplane is determined by $m \in \{-1, +1\}$. We will employ stumps as weak learners in our boosting algorithm. To train stumps on weighted data, use the learning rule

$$(j^*, \theta^*) := \arg\min_{j,\theta} \frac{\sum_{i=1}^n w_i \mathbb{I}\{y_i \neq c(\mathbf{x}^{(i)}|j, \theta, m)\}}{\sum_{i=1}^n w_i} . \tag{2}$$

In the implementation of your training routine, first determine an optimal parameter θ_j^* for each dimension $j = 1, ..., d$, and then select the j^* for which the cost term in (2) is minimal.

Homework problems:

1. (30 points) Implement the AdaBoost algorithm in R. The algorithm requires two auxiliary functions, to train and evaluate the weak learner. We also need a third function which implements the resulting boosting classifier. We will use decision stumps as weak learners, but a good implementation of the boosting algorithm should permit you to easily plug in arbitrary weak learners. To make sure that is possible, please use function calls of the following form:
   - `pars <- train(X, w, y)` for the weak learner training routine, where `X` is a matrix the columns of which are the training vectors $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$, and `w` and `y` are vectors containing the weights and class labels. The output `pars` is a list which contains the parameters specifying the resulting classifier. (In our case, `pars` will be the triplet (j, θ, m) which specifies the decision stump).
   - `label <- classify(X, pars)` for the classification routine, which evaluates the weak learner on `X` using the parametrization `pars`.
   - A function `c_hat <- agg_class(X, alpha, allPars)` which evaluates the boosting classifier ("aggregated classifier") on `X`. The argument `alpha` denotes the vector of voting weights and `allPars` contains the parameters of all weak learners.
2. (15 points) Implement the functions `train` and `classify` for decision stumps.
3. (20 points) Run your algorithm on the USPS data (the digit data we used in Homework 2) and evaluate your results using cross validation.

   More precisely: Your AdaBoost algorithm returns a classifier that is a combination of B weak learners. Since it is an incremental algorithm, we can evaluate the AdaBoost at every iteration b by considering the sum up to the b-th weak learner. At each iteration, perform 5-fold cross validation to estimate the training and test error of the current classifier (that is, the errors measured on the cross validation training and test sets, respectively).
4. (5 points) Plot the training error and the test error as a function of b.

Submission. Please make sure your solution contains the following:

- Your implementation for `train`, `classify` and `agg_class`.
- Your implementation of `AdaBoost`.
- Plots of your results (training error and cross-validated test error).

Solution:

Question 1. The implementation of `adaBoost` is straightforward from the exercise sheet. Note that we use a list `allPars` to store the triplet of parameters (j^*, θ^*, m^*) for the base classifiers.

```
## function <adaBoost>

adaBoost <- function(X,y,B) {
    n <- dim(X)[1]
    w <- rep(1/n,times=n)
    alpha <- rep(0,times=B)
    allPars <- rep(list(list()),B)

    # boost base classifiers
    for(b in 1:B) {

        # step a) train base classifier
        allPars[[b]] <- train(X,w,y)

        # step b) compute error
        missClass <- (y != classify(X,allPars[[b]]))
        e <- (w %*% missClass/sum(w))[1]

        # step c) compute voting weight
        alpha[b] <- log((1-e)/e)

        # step d) recompute weights
        w <- w*exp(alpha[b]*missClass)

    }

    return(list(allPars=allPars, alpha=alpha))
}
```

agg_class computes the boosted classifier response on all samples X:

```
## function <agg_class> for AdaBoost implementation

agg_class <- function(X,alpha,allPars) {
    n <- dim(X)[1]
    B <- length(alpha)
    Labels <- matrix(0,nrow=n,ncol=B)

    # determine labeling for each base classifier
    for(b in 1:B) {
        Labels[,b] <- classify(X,allPars[[b]])
    }

    # weight classifier response with respective alpha coefficient
    Labels <- Labels %*% alpha

    c_hat <- sign(Labels)

    return(c_hat)
}
```

Question 2. To train stumps efficiently, we make the following two observations:

1. It is sufficient to test for $x_j^{(i)}, i = 1, \dots, n$ as the value of the optimum thresholding point θ_{j^*}. We do not have to consider the whole domain of x_j.
2. θ_j^* is the global extremum of a cumulative sum. Firstly, sort the samples in ascending order along dimension j. Then compute $w_{cum} := \sum_{\{i:\, x_j^{(i)} \leq \theta\}} w_i y_i$, the sum of weight on the left side of the threshold, while progressively shifting the threshold to the larger elements. Finally, θ_j^* is the global extremum of w_{cum}, and the sign determines the orientation of the inequality. Consider the following example where we have chosen $w_i = 1\ \forall i$ for simplicity:

sample:	A	B	C	D	E	F
$x_j^{(i)}$:	1	2	2	2	3	4
y_i:	-	-	+	+	+	-
w_{cum}:	-1	-2	-1	0	1	0

   The extremal value of w_{cum} is -2, suggesting that the optimum split point lies between samples B and C for a total of one error (sample F would be mislabelled '+'). However, $x_j^{(i)} = 2$ occurs three times, so this is not a valid thresholding point. Masking out entries B and C, we conclude that the optimum split point lies either between samples A and B ($x_j > 1$) or E and F ($x_j \leq 3$), for a total of two errors in both cases.

The R implementation uses the built-in cumsum function to efficiently compute the cumulative sum, avoiding an explicit loop over the samples:

```
## function <train> for AdaBoost implementation

train <- function(X,w,y) {
    n <- dim(X)[1]
    p <- dim(X)[2]

    mode <- rep(0,times=p)
    theta <- rep(0,times=p)
    loss <- rep(0,times=p)

    # find optimal theta for every dimension j
    for(j in 1:p) {

        # sort datapoints along dimension
        indx <- order(X[,j])
        x_j <- X[indx,j]

        # using a cumulative sum, count the weight when progressively
        # shifting the threshold to the right
        w_cum <- cumsum(w[indx] * y[indx])

        # handle multiple occurrences of same x_j value: threshold
        # point must not lie between elements of same value
        w_cum[duplicated(x_j)==1] <- NA

        # find the optimum threshold and classify accordingly
        m <- max(abs(w_cum), na.rm=TRUE)
        maxIndx <- min(which(abs(w_cum)==m))
        mode[j] <- (w_cum[maxIndx] < 0)*2 - 1
        theta[j] <- x_j[maxIndx]
        c <- ((x_j > theta[j])*2 - 1)*mode[j]
        loss[j] <- w %*% (c != y)
    }

    # determine optimum dimension, threshold and comparison mode
    m <- min(loss)
    j_star <- min(which(loss==m))

    pars <- list(j=j_star, theta=theta[j_star], mode=mode[j_star])
    return(pars)
}
```

Samples are classified in parallel:

```
## function <classify> for AdaBoost implementation

classify <- function(X,pars) {
    label <- (2*(X[,pars$j] > pars$theta) - 1)*pars$mode

    return(label)
}
```

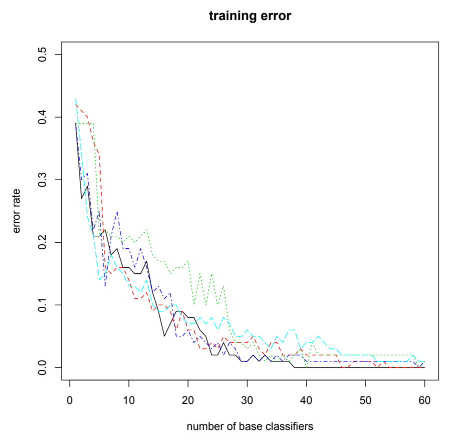
Question 3. (**Note:** There are various plausible ways in which cross validation can be used here; below is one possible solution that I suggested in an email to students. If anybody used a different method, that is perfectly fine as long as it makes sense.)

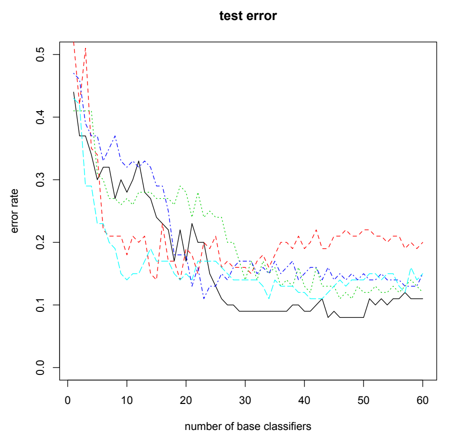
1. Split the input data into five equally sized folds. The five folds define 5 training data sets (each consisting of four folds, with one fold removed respectively).
2. Train a boosting classifier on *each* of the five training sets; at each iteration, compute:
   - The training error (misclassification rate on the four folds used for training).
   - The prediction error erstimate (misclassification error on the remaining fold).
3. For each iteration b, average the five training error estimates and average the five test error estimates.
4. Plot the averaged error estimates as a function of b.

Another way is to randomly split the data into training and testing halves five different times, as is done in the R script that runs all of our functions and plots the error:

```
## file to run AdaBoost

rm(list=ls())
source("adaBoost.R")
source("agg_class.R")
source("train.R")
source("classify.R")

set.seed(10)

B_max <- 60
nCV <- 5

# load datasets
X <- read.table("uspsdata.txt")
y <- read.table("uspscl.txt")[,1]
n <- dim(X)[1]

testErrorRate <- matrix(0,nrow=B_max,ncol=nCV)
trainErrorRate <- matrix(0,nrow=B_max,ncol=nCV)
for(i in 1:nCV) {

    # randomly split data in training and test half
    p <- sample.int(n)
    trainIndx <- p[1:round(n/2)]
    testIndx <- p[-(1:round(n/2))]

    ada <- adaBoost(X[trainIndx,], y[trainIndx], B_max)
    allPars <- ada$allPars
    alpha <- ada$alpha

    # determine error rate, depending on the number of base classifier
    for(B in 1:B_max) {
        c_hat_test <- agg_class(X[testIndx,], alpha[1:B], allPars[1:B])
        testErrorRate[B,i] <- mean(y[testIndx] != c_hat_test)
        c_hat_train <- agg_class(X[trainIndx,], alpha[1:B], allPars[1:
        trainErrorRate[B,i] <- mean(y[trainIndx] != c_hat_train)
    }
}

# plot results
matplot(trainErrorRate,type="l",lty=1:nCV,main="training␣error",
    xlab="number␣of␣base␣classifiers",ylab="error␣rate",ylim=c(0,0.5))
quartz()
matplot(testErrorRate,type="l",lty=1:nCV,main="test␣error",
    xlab="number␣of␣base␣classifiers",ylab="error␣rate",ylim=c(0,0.5))
```

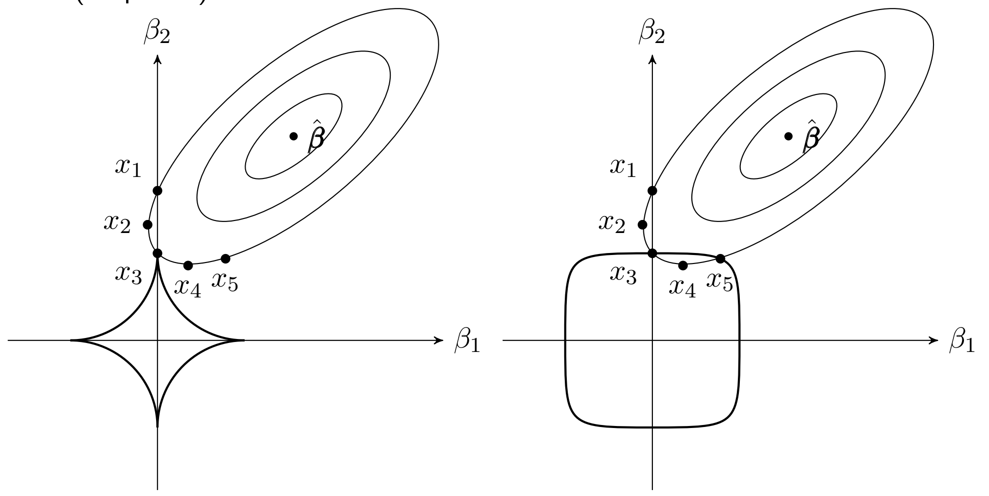
Question 4. The error rate of the boosted decision stumps is plotted in the following two graphs. Each line style corresponds to a random split of the data in training and testing halves.





As expected, the training error vanishes for sufficiently many base classifiers. Judging from three cross validation runs, going beyond $b = 40$ is pointless because all training samples have already been classified correctly. For zero training error we expect performance degradation on test data because of overfitting. However, judging from the right plot, AdaBoost is robust in that regard. The curves essentially stay flat for $b > 30$, i.e. increasing the number of model parameters beyond the optimum $b^*_{CV} \approx 30$ has no adverse effects.

2. ℓ_q **regression** (30 points)

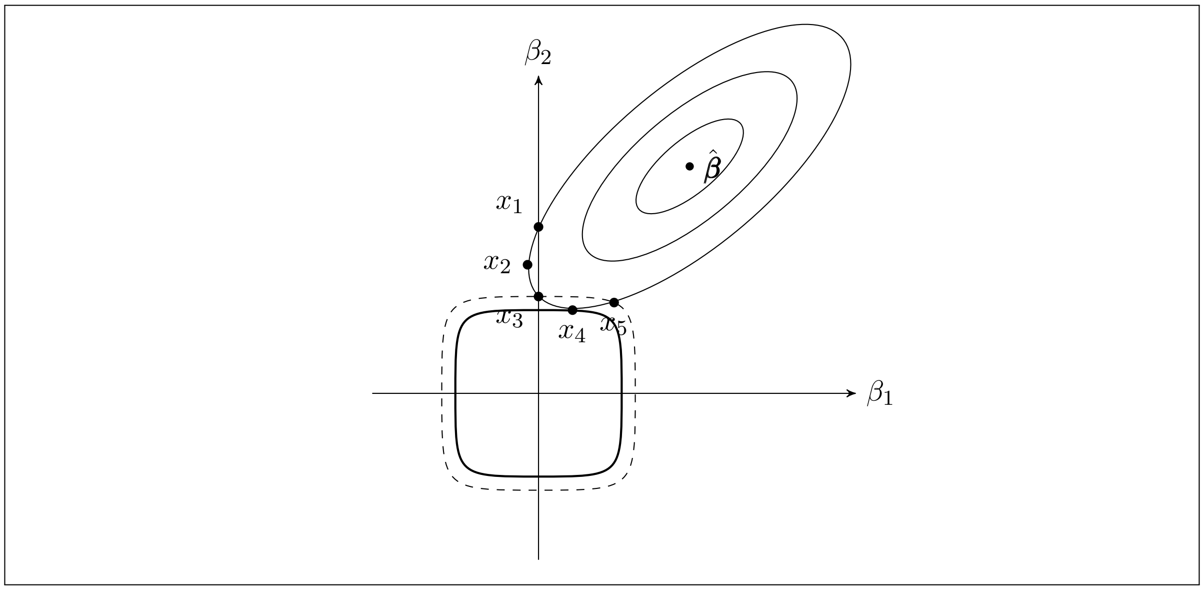


The figures show the cost function components of the ℓ_q-regression problems with $q = 0.5$ (left) and $q = 4$ (right).

1. (15 points) Does one/none/both of the cost functions encourage sparse estimates? If so, which one? Explain your answer.
2. (15 points) Which of the points $x_1, \ldots, x_5$ would achieve the smallest cost under the ℓ_q-constrained least squares cost function? For each of the two cases, name the respective point and give a brief explanation for your answer.

Solution:

1. $q = 0.5$ encourages sparse solutions, wheras $q = 4$ does not.
   Explanation: If $q = 0.5$, then for any ellipse iso-line of the square-loss which interesects an axis, the minimal $\ell_{0.5}$-distance to the origin (and hence to the smallest penalty term) is achieved by a point on the axis. Hence, the entry for the respective other axis is zero. In contrast, $q = 4$ encourages entries β_i of roughly even size.
2. $q = 0.5$: The cost-optimal point is x_3, since it is located on the β_2-axis as explained above.
   $q = 4$: The cost-optimal point is x_4, since we can shrink the penalty iso-line in the figure further until it intersects the ellipse only at x_4.


β_2
$\hat{\beta}$
x_1
x_2
x_3
x_4
x_5
β_1