1.

(1) Implement of train, classify and agg_class function.

Train a decision stamp using the weighted train data.
First, define a function called learning_rule to calculate the error rate. Then use optimize function to get the best split. However, the optimize function only do one-dimension optimization, we have three dimensions: the location of the axis j, the split t_j in axis j, and the m, thus we use two loops in j and m, getting the optimized t_j for each j and m.

```
# The train function
train <- function(X, w, y){
    if (!is.vector(y)){
        y <- as.vector(y$V1)
    }
    n <- dim(X)[1]   # the number of observations
    d <- dim(X)[2]   # the number of features
    theta <- c()
    error_rate <- c()
    m_list <- c()
    # compute the error rate for a weak learner
    learning_rule <- function(theta_j, x_j, weight, m_stump){
        predict_y <- c()
        # predict the classification based on the split point
        for (i in c(1: n)){
            if (x_j[i] > theta_j){
                predict_y[i] <- m_stump
            } else {
                predict_y[i] <- -m_stump
            }
        }
        error_rate <- 0
        for (i in c(1: n)){
            if (y[i] != predict_y[i]){
                error_rate <- error_rate + weight[i]
            }
        }

        error_rate <- error_rate / (sum(weight))
        return(error_rate)
    }

    # compute the optimal parameter theta_j and m for each x
    for (j in c(1: d)){
        test_parameter <- c()
        test_error_rate <- c()
```

```
        k <- 1
        # compute the arg min of cost function
        for (m in c(-1, 1)){
            optimization <- optimize(learning_rule, interval =
c(min(X[,j]), max(X[,j])), x_j = X[, j], weight = w, m_stump =
m)
            test_parameter[k] <- optimization$minimum
            test_error_rate[k] <- optimization$objective
            k <- k + 1
        }
        if(test_error_rate[1] < test_error_rate[2]){
            theta[j] <- test_parameter[1]
            error_rate[j] <- test_error_rate[1]
            m_list[j] <- c(-1)
        } else {
            theta[j] <- test_parameter[2]
            error_rate[j] <- test_error_rate[2]
            m_list[j] <- c(1)
        }
    }
    location_j <- which.min(error_rate)
    result_list = list(j = location_j, theta =
theta[location_j], mode = m_list[location_j])
    return(result_list)
}
```

Use the definition of the decision stamp to do classification.

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

Aggregate the weak learners.

```
# agg_class function
agg_class <- function(X, alpha, allPars) {
    n <- dim(X)[1]
    B <- length(alpha)
    labels <- matrix(0, nrow = n, ncol = B)
    for (i in c(1: B)){
        labels[,i] <- classify(X, allPars[[i]])
    }
    sum_label <- labels %*% alpha
    classifier <- as.vector(sign(sum_label))
    return(classifier)
}
```

Implement AdaBoost using train, classify and agg_class function.

```
# The adaBoost function
adaBoost <- function(X, y, B){
    if (!is.vector(y)){
        y <- as.vector(y$V1)
    }
    n <- dim(X)[1]
    d <- dim(X)[2]
    weight_list <- rep(1/n, d)
    alpha <- c()
    allPars <- list()

    # train routin
    for (i in c(1: B)){
        allPars[[i]] <- train(X, weight_list, y)
        predict_y <- classify(X, allPars[[i]])
        error_rate <- sum((predict_y != y) * weight_list) /
(sum(weight_list))
        # compute alpha and new weights
        alpha[i] <- log((1 - error_rate) / error_rate)
        weight_list <- weight_list * exp(alpha[i] *
(predict_y != y))
    }

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

As we need to perform k-fold cross-validation in USPS data, thus I defined a cross_validation function

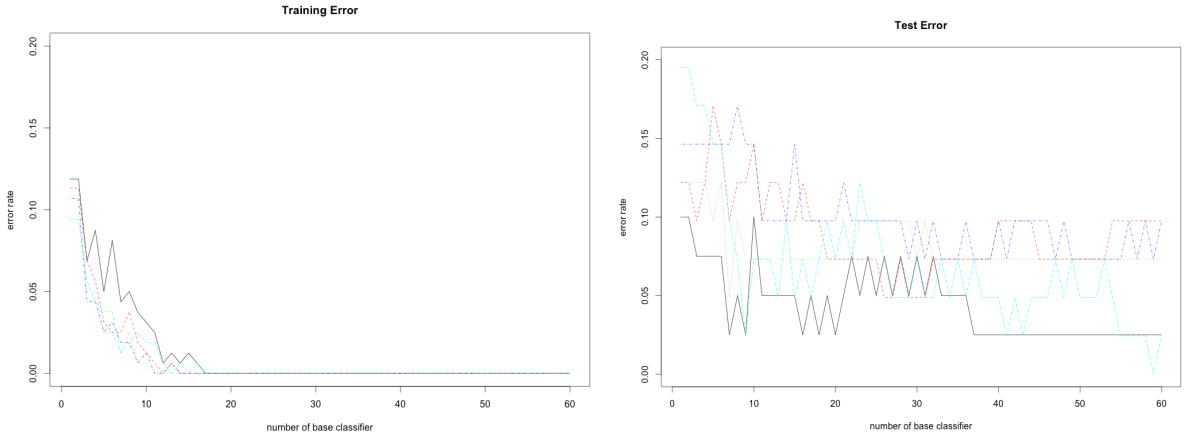
```
# cross validation function
cross_validation <- function(X, y, B_max, k_fold){
    n <- dim(X)[1]
    train_error_rate <- matrix(0, nrow = B_max, ncol = k_fold)
    test_error_rate <- matrix(0, nrow = B_max, ncol = k_fold)
    # split data into k group
    num_of_group <- round(n/k_fold)
    k_fold_groups <- list()
    for(k in 1:k_fold){
        ini_point <- (k - 1) * num_of_group
        stop_point <- k * num_of_group
        if(stop_point < n){
            k_fold_groups[[k]] <- c(ini_point: stop_point)
        } else {
            k_fold_groups[[k]] <- c(ini_point : n)
        }
```

```
    }

    # for each iteration in cross validation, choose train data
and test data
    for(k in 1:k_fold){
        train.x <- X[-k_fold_groups[[k]],]
        train.y <- y[-k_fold_groups[[k]],]
        test.x <- X[k_fold_groups[[k]],]
        test.y <- y[k_fold_groups[[k]],]
        ada <- adaBoost(train.x, train.y, B_max)
        allPars <- ada$allPars
        alpha <- ada$alpha

        # compute test error rate and train error rate for each
b
        for (b in 1: B_max){
            test_predict_y <- agg_class(test.x, alpha[1:b],
allPars = allPars[1:b])
            test_error_rate[b, k] <- mean(test.y !=
test_predict_y)
            train_predict_y <- agg_class(train.x, alpha[1:b],
allPars = allPars[1:b])
            train_error_rate[b, k] <- mean(train.y !=
train_predict_y)
        }
    }
    return(list(train_error_rate = train_error_rate,
test_error_rate = test_error_rate))
}
```

The Plot of train error and test error:



As the weak learner increase, train error vanishes quickly. When b = 18, the train error is almost zero. However, when b = 18, although the train error is minimal, increasing the number of weak learner would still decrease the test error. According to the test error graph, I would choose b > 30 and b < 40, as when b > 40, increasing b does not have effect in reducing test error.

2.

(1)

The left one (q = 0.5) encourages sparse solutions, the right one (q = 4) does not encourage sparse solutions.

For q = 0.5, the intersections of the ellipse iso-line of the square-loss and the edge of the penalty are on axis, which means the entry for the respective other axis is zero. In contrast, when q = 4, the entries are even size, not zero.

(2)

q = 0.5: x_3 would achieve the smallest cost, since it is intersecting the edge of the penalty and locate at the beta-2 axis.

q = 4: x_4 would achieve the smallest cost, since it has the same square-loss with other x's, while it has the smallest penalty.