Statistical Computing 1 - Assessed Coursework 2

Conor Newton

1 Support Vector Machines & Subgradient Descent

The parameter $\mathbf{w}$ for a SVM can be found by solving the following unconstrained optimization problem

$$\min_{\mathbf{w}} \left[||\mathbf{w}||^2 + C \sum_{i=1}^{n} \underbrace{\max\{0, 1 - y_i \cdot f(\mathbf{x}_i; \mathbf{w})\}}_{\text{hinge loss}} \right]$$

where C is a parameter that determines how hard/soft the margin is and $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ is a dataset where $y = \{-1, +1\}$. The objective and hinge loss function are in fact both convex.

We can rewrite the objective function as a summation

$$\begin{aligned} \min_{\mathbf{w}} \left[\frac{1}{n} \sum_{i=1}^{n} \left[||\mathbf{w}||^2 + nC \max\{0, 1 - y_i \cdot f(\mathbf{x}_i; \mathbf{w})\} \right] \right] &= \min_{\mathbf{w}} \left[\frac{1}{n} \sum_{i=1}^{n} \left[\lambda \cdot ||\mathbf{w}||^2 + \max\{0, 1 - y_i \cdot f(\mathbf{x}_i; \mathbf{w})\} \right] \right] \\ &= \min_{\mathbf{w}} \left[\frac{1}{n} \sum_{i=1}^{n} Q_i(\mathbf{w}) \right] \end{aligned}$$

This is now in the correct form to apply stochastic gradient descent to solve this problem. This approach would be ideal since we could be dealing with a large dataset. However, since the hinge loss is not differentiable, we cannot find the gradient. Instead, we will consider the subgradient.

A vector $\mathbf{s}$ is a subgradient of a convex function g at $\mathbf{x}_0$ if

$$\frac{g(\mathbf{x}) - g(\mathbf{x}_0)}{\mathbf{x} - \mathbf{x}_0} \geq \mathbf{s} \text{ for all } \mathbf{x}$$

This can be thought of as a lower bound of all the tangents at $\mathbf{x}_0$ coming from all of the different possible directions.

A subgradient of the hinge loss function at $\mathbf{w}$ for a fixed data points $(\mathbf{x}_i, y_i)$ is

$$\begin{cases} -y_i \cdot \mathbf{x}_i & \text{if } 1 > y_i \cdot f(\mathbf{x}; \mathbf{w}) \\ \mathbf{0} & \text{if } 1 \leq y_i \cdot f(\mathbf{x}; \mathbf{w}) \end{cases}$$

A subgradient of $Q_i(\mathbf{w})$ is therefore

$$\begin{cases} 2\lambda\mathbf{w} - y_i \cdot \mathbf{x}_i & \text{if } 1 > y_i \cdot f(\mathbf{x}; \mathbf{w}) \\ 2\lambda\mathbf{w} & \text{if } 1 \leq y_i \cdot f(\mathbf{x}; \mathbf{w}) \end{cases}$$

Now using this subgradient in place of the gradient in a stochastic gradient descent algorithm we can approximate the minimum of (1). This is done by iterating the following for a random choice of i

$$\begin{aligned} \mathbf{w}_{t+1} &\leftarrow \mathbf{w}_t + \eta_t (y_i \cdot \mathbf{x}_i - \lambda \mathbf{w}_t) && \text{if } 1 > y_i \cdot f(\mathbf{x}; \mathbf{w}) \\ &\leftarrow \mathbf{w}_t - \eta_t \lambda \mathbf{w}_t && \text{otherwise} \end{aligned}$$

Here the learning rate is $\eta_t = \frac{2}{\lambda t}$.

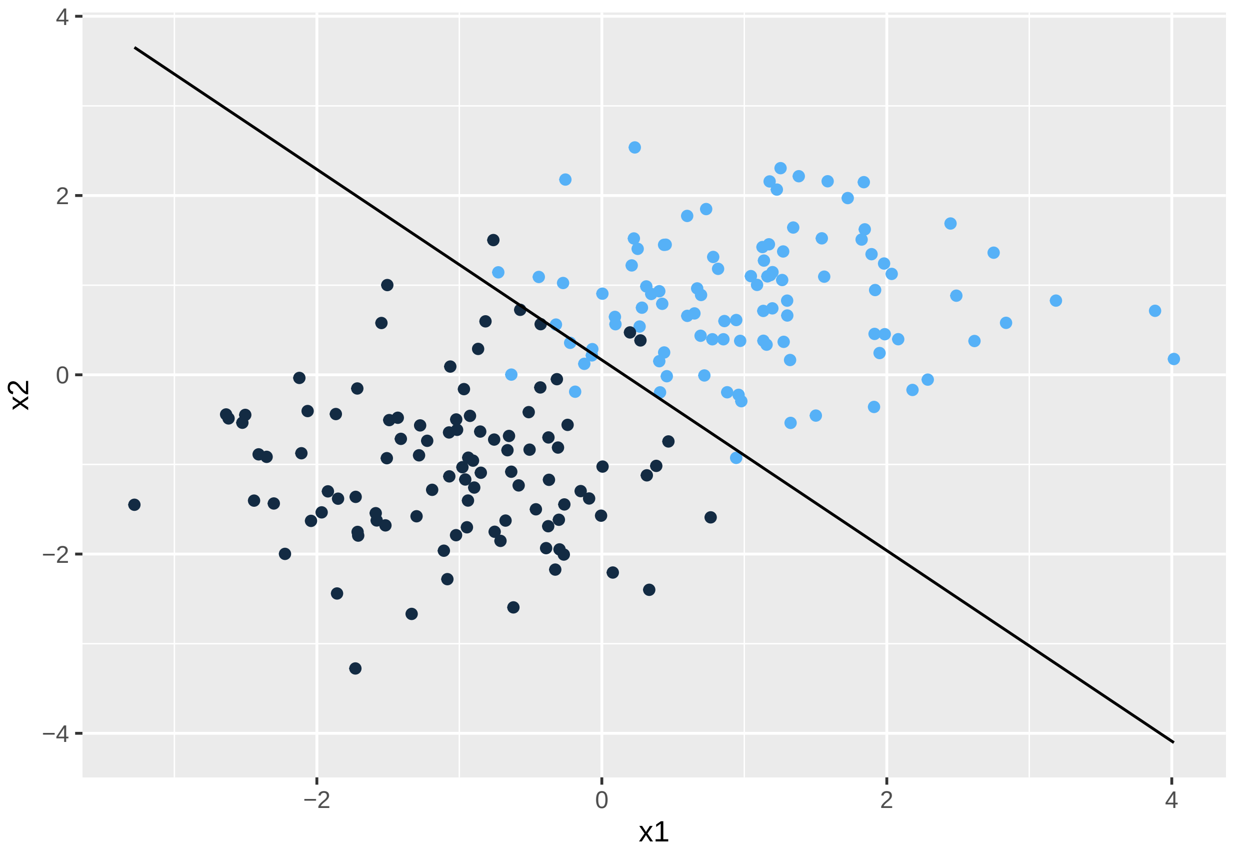
This type of algorithm is known as a subgradient descent. They have many provable convergence cases including this one!

2 SubGD for SVM's in R

The most basic implementation of the subgradient descent to find the parameters of an SVM is as follows

```r
data <- data.frame(x1 = c(rnorm(n = 100, mean = 1, sd = 0.8), rnorm(n = 100, mean = -1, sd = 0.8)),
                   x2 = c(rnorm(n = 100, mean = 1, sd = 0.8), rnorm(n = 100, mean = -1, sd = 0.8)),
                   y = c(rep(1, 100), rep(-1, 100)))

train_svm <- function(x, y, C = 1, iters=100) {
    l <- 2 / C # This is lambda from the algorithm
    w <- rep(0, ncol(x) + 1)

    # We repeat the SGD algorithm for a better predicition
    for (i in 1:iters) {

        # permuation of the rows of x
        perm <- sample(nrow(x))

        for (j in 1:(nrow(x))) {
            # Step size eta_t
            step <- 1 / (l * j)

            # Need to shuffle the data using the permuation
            yy <- y[perm[j]]
            xx <- x[perm[j], ]

            if (yy * (sum(w * c(1, xx)) < 1)) {
                w <- w + step * (yy * c(1, xx) - l * w)
            } else {
                w <- w - step * l * w
            }
        }
    }
    return(as.vector(w))
}

w <- train_svm(as.matrix(data[,-3]), as.vector(data[,3]))
```

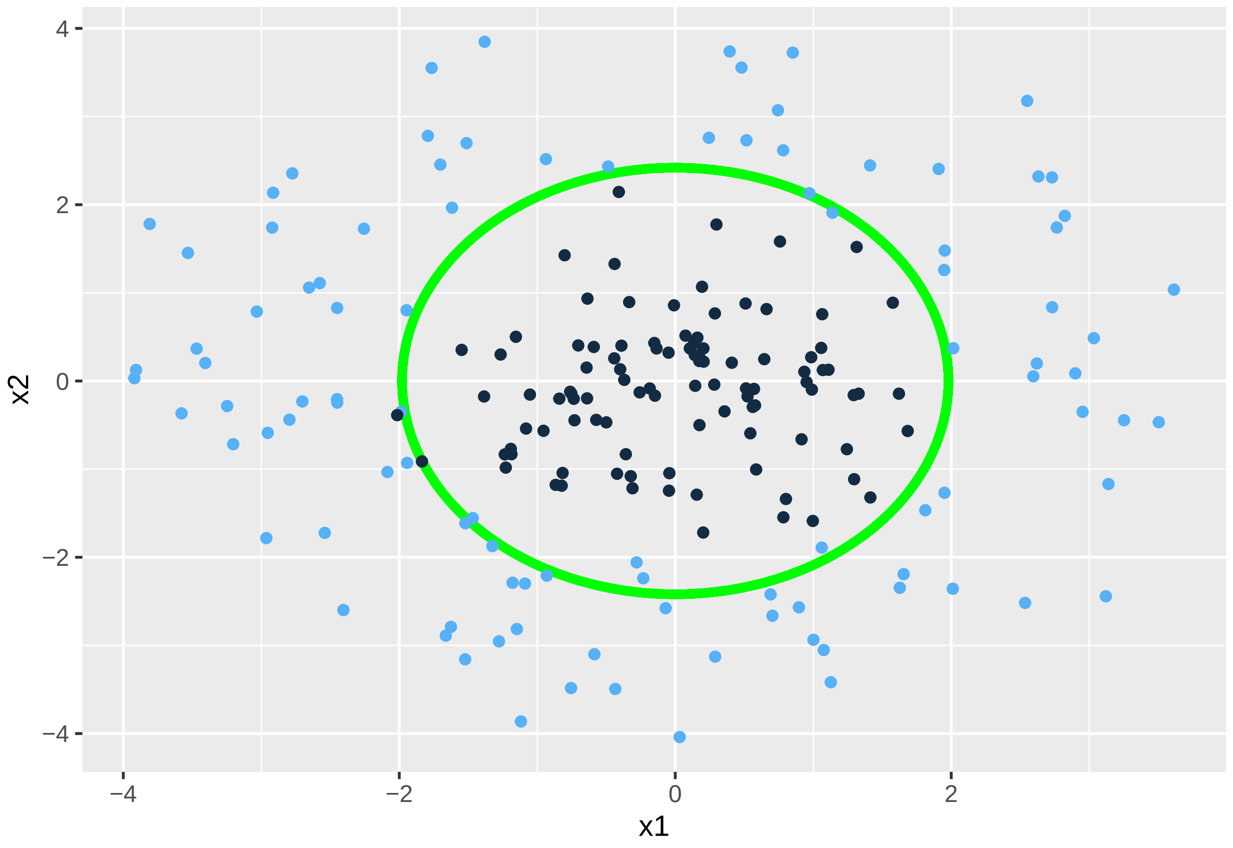
We can plot the boundary created by **w** against the data set



If our data is not linearly separable we can simply adapt our code to use a feature transform. We just need to wrap every instance of **x** with the chosen feature transform function.

```
# Generate circular data
r <- runif(n = 100, min = 0, max = 2 * pi)
data2 <- data.frame(x1 = c(3 * sin(r) + rnorm(n = 100, mean = 0, sd = 0.5), rnorm(n = 100, mean = 0, sd
                    x2 = c(3 * cos(r) + rnorm(n = 100, mean = 0, sd = 0.5), rnorm(n = 100, mean = 0, sd =
                    y = c(rep(1, 100), rep(-1, 100)))

# SGD algorithm with feature transform support
train_svm_ft <- function(x, y, ft, C = 100, iters=1000) {
    l <- 2 / C # This is lambda from the algorithm
    w <- rep(0, length(ft(x[1,])) + 1)

    # We repeat the SGD algorithm for a better prediction
    for (i in 1:iters) {

        # permuation of the rows of x
        perm <- sample(nrow(x))

        for (j in 1:(nrow(x))) {
            # Step size eta_t
            step <- 1 / (l * j)

            # Need to shuffle the data using the permutation
            yy <- y[perm[j]]
            xx <- ft(x[perm[j], ])
            if (yy * (sum(w * c(1, xx)) < 1)) {
                w <- w + step * (yy * c(1, xx) - l * w)
            } else {
                w <- w - step * l * w
            }
        }
    }
    return(as.vector(w))
}

ft <- function(x) c(x[1]^2, x[2]^2)
w <- train_svm_ft(as.matrix(data2[,-3]), as.vector(data2[,3]), ft)
```

Again, we plot the boundary against the data to see the result.



Some notes on implementation:

- Iterating through the x values random increases performance greatly!
- Circle feature transform requires fairly high margin variable (i.e. $C \geq 100$) to get the best results.

3 Multi-Class Classification

I will now show how to make use the SVM we have implemented to perform multi-class classification. For this example we will use the infamous iris dataset. This dataset probably wont highlight the power of this classifier but it at least shows that we can perform multiclass classification.

To perform the multi-class classification, we will create $n(n-1)/2$ SVM's (one for each pair of possible classes). Each will decide if an input belongs to one of two classes. We then make the classification by choosing the most popular decision amongst the classifiers.

We start by loading the iris dataset and randomly splitting it into testing and training data

```r
library(datasets)
data(iris)

smp_size <- floor(0.75 * nrow(iris))
train_ind <- sample(seq_len(nrow(iris)), size = smp_size)
train <- iris[train_ind, ]
test  <- iris[-train_ind, ]

classes <- unique(iris$Species)
```

We modify our `train_svm` function to take two parameters that correspond to the two classes we are training an SVM for. Then `train_svm` will skip all of data that does not correspond to these classes in training. This is much easier than making modifications to the dataset.

```r
train_svm_ft <- function(x, y, ft, C = 100, iters=10000, label1 = 1, label2 = -1) {
    l <- 2 / (nrow(x) * C)
    w <- rep(0, length(ft(x[1, ])) + 1)
    for (i in 1:iters) {

        # permutation of the rows of x
        perm <- sample(nrow(x))

        for (j in 1:(nrow(x))) {
            step <- 1 / (l * i)
            # We only want to consider the data for the two given classes
            # The two classes should be mapped to 1 and -1
            if (y[perm[j]] == label1) {
                yy <- 1
            } else if (y[perm[j]] == label2) {
                yy <- -1
            } else {
                next # Skip this iteration if y value is not needed
            }

            xx <- as.vector(ft(x[perm[j], ]))

            if (yy * (sum(w * c(1, xx))) < 1) {
                w <- w + step * (yy * c(1, xx) - l * w)
            } else {
                w <- w - step * l * w
            }
        }
    }
    return(as.vector(w))
}
```

The following is a helper function trains the SVM's for all pairs of classes in parallel and returns a matrix where the columns are the parameters for the different classifiers

```
library(parallel)

train_one_vs_one <- function(data, classes, ft) {
    #returns a matrix of all pairs of classes
    pairs <- combn(classes, 2)
    #Train the one vs one classifiers in parallel
    ws <- mclapply(seq_len(ncol(pairs)),
                   function(x) train_svm_ft(as.matrix(data[,-ncol(data)]),
                                            as.vector(data[,ncol(data)]),
                                            ft,
                                            label1 = pairs[1, x],
                                            label2 = pairs[2, x]),
                   mc.cores = 4)
    # Returns the data as a matrix where the columns are the parameters for different svms
    return(matrix(unlist(ws), ncol = ncol(pairs)))
}

ws <- train_one_vs_one(train, classes, identity)
```

We have successfully trained our SVM's! Now we want to use these SVM's to make a prediction, this can be done with the following function.

```
predict_one_vs_one <- function(ws, x, classes) {
    pairs <- combn(classes, 2)
    votes <- sapply(seq_len(ncol(pairs)), function(i) {
        if (sum(ws[, i] * c(1, x)) > 0) return(as.vector(pairs[1, i]))
        else return(as.vector(pairs[2, i]))
    })
    # Does some magic to return the mode
    return(as.vector(classes[which.max(tabulate(match(votes, classes)))]))
}

# Predict the class of the first element of the testing data
print(predict_one_vs_one(ws, as.numeric(test[1, -1]), classes))
```

```
## [1] "setosa"
```

To finish off, we can test our SVM classifier against the whole testing dataset with the following function

```
test_one_vs_one <- function(ws, x, y, classes) {
    res <- unlist(lapply(seq_len(nrow(x)), function(i) {
        if (as.vector(y[i]) == predict_one_vs_one(ws, as.numeric(x[i, ]), classes)) {
            return(1)
        }
        return(0)
    }))
    return(sum(res) / length(res))
}
print(test_one_vs_one(ws, test[, -ncol(test)], test$Species, classes))
```

```
## [1] 0.9736842
```

After many runs this has constantly achieved 90%+ accuracy!