Week 7 - Computation tricks

```
## Loading required package: foreach
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

Gradient Decent for Linear Regression

Let us consider the following regression model:

$$Y = X\beta + \epsilon, \epsilon \sim N(\mathbf{0}, \sigma^2 I)$$

, today we will implement the stochastic gradient decent algorithm and introduce some ways in which we can speed up the computation.

```
get_data <- function(){
n_samples <- 10000
features <- 50
X <- matrix(rnorm(n = n_samples*features),nrow=n_samples)
epsilon <- rnorm(n=n_samples,sd =0.1)
beta <- rnorm(n=features)
y <- X %*% beta + epsilon
return(cbind(X, y))
}
```

Our objective is:

$$\min_{\beta} \|X\beta - Y\|_2^2$$

The gradient with respect to β is:

$$\nabla_\beta \|X\beta - Y\|_2^2 = 2(X\beta - Y) \cdot X = \sum_{i=1}^{n} (X_{i.}\beta - Y_i)X_{.i}$$

Our goal today is to implement Gradient Decent algorithm, with the following update rule:

$$\begin{aligned} \beta^{(t+1)} &= \beta^{(t)} - \gamma \nabla_\beta \|X\beta - Y\|_2^2 \\ &= \beta^{(t)} - \gamma \cdot \sum_{i=1}^{n} (X_{i.}\beta^{(t)} - Y_i)X_{.i} \end{aligned}$$

```r
get_beta_sgd <- function(X, y,get_gradient, batch_size=32, n_iter=1000, tol=0.01, gamma = 0.00
1){
  features <- dim(X)[2]
  beta <- matrix(runif(features), nrow = 1)
  for (i in 1:n_iter){
    sub_sample <- sample(length(y), batch_size)
    grad <- get_gradient(X[sub_sample, ], y[sub_sample], beta)
    if (sum(grad^2) <= tol){
      return(beta)
    }

    beta  <- beta - gamma * grad
  }

  return(beta)

}
```

We start with naive implementation

```r
get_gradient_naive <- function(X, y, beta){
  grad <- beta * 0
  n_data_point <- length(y)
  for (i in 1:n_data_point){
    diff <- (X[i,] %*% beta -y[i])[1]
    grad <- grad + diff * X[i,]
  }
  return(grad)

}
```

Now let’s go one step further, using matrix operations

```r
get_gradient <- function(X, y, beta){
  diff <- X %*% matrix(beta, ncol=1) - y
  grad <-t(diff) %*% X
  return(grad)

}
```

```r
bm <- benchmark("naive sgd" = {
                  data <- get_data()
                  X <- data[,1:50]
                  y <- data[,51]
                  b <- get_beta_sgd(X, y, get_gradient_naive)
                },
                "sgd" = {
                  data <- get_data()
                  X <- data[,1:50]
                  y <- data[,51]
                  b <- get_beta_sgd(X, y, get_gradient)
                },
                replications = 5,
                columns = c("test",  "elapsed","relative"))

kable(bm)
```

test	elapsed	relative

test	elapsed	relative
naive sgd	1.976	2.827
sgd	0.699	1.000

Now suppose we want to do some stability analysis, fitting the model on variations of the dataset. We start the naive approach - looping

```
stability_beta <- function(){
    data <- get_data()
    X <- data[,1:50]
    y <- data[,51]
    beta <- get_beta_sgd(X, y, get_gradient)
  return(beta)

}
```

A better approach would be to parallelize this calculation, that is fit multiple model at the same time

```
stability_loop <- function(n){
  for (i in 1:n){
    beta <- stability_beta()
  }
}
stability_parallel <- function(n){
no_cores <- detectCores() - 1
cl <- makeCluster(no_cores, type="FORK")
registerDoParallel(cl)
result <- foreach(i=1:n) %dopar% stability_beta()

}
```

```
bm <- benchmark("paralell" = {
  stability_parallel(10)
},
                "loop" = {
                  stability_loop(10)
                },
                replications = 5,
                columns = c("test",  "elapsed","relative"))

kable(bm)
```

	test	elapsed	relative
2	loop	6.381	2.182
1	paralell	2.925	1.000

Rcpp

Rcpp enables improving performance by rewriting key functions in C++. We start with a simple example:

```
cppFunction('int add(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}')

add(1, 2, 3)
```

```
## [1] 6
```

Vector input, scalar output

```
sumR <- function(x) {
  total <- 0
  for (i in seq_along(x)) {
    total <- total + x[i]
  }
  total
}

cppFunction('double sumC(NumericVector x) {
  int n = x.size();
  double total = 0;
  for(int i = 0; i < n; ++i) {
    total += x[i];
  }
  return total;
}')
```

A few syntax comments:

- `double sumC(NumericVector x)` - double is the output type, `sumC(NumericVector x)` is the function.
- `int n = x.size()` - crate an integer equals to the length of x vector
- `double total = 0` - create a double type named total, equals to zero.

Let's compare the running times:

```
vector = rnorm(10000)
bm <- benchmark("Rcpp" = {
  sumC(vector)
},
                "R" = {
                  sumR(vector)
                },
                replications = 5,
                columns = c("test",  "elapsed","relative"))

kable(bm)
```

	test	elapsed	relative
2	R	0.002	2
1	Rcpp	0.001	1

Now matrices, let's implement row sums:

```
cppFunction('NumericVector rowSumsC(NumericMatrix x) {
  int nrow = x.nrow(), ncol = x.ncol();
  NumericVector out(nrow);

  for (int i = 0; i < nrow; i++) {
    double total = 0;
    for (int j = 0; j < ncol; j++) {
      total += x(i, j);
    }
    out[i] = total;
  }
  return out;
}')
```

A few syntax comments:

- `NumericVector out(nrow)` - create a new numeric vector of length n with a constructor: NumericVector out(n). Another useful way of making a vector is to copy an existing one: `NumericVector zs = clone(ys)`

- In C++, you subset a matrix with (), not []

```
rows <- 100
cols <- 5000

bm <- benchmark("Rcpp" = {
  x <- matrix(sample(rows*cols), rows)
  rowSumsC(x)
},
                "R" = {
                  x <- matrix(sample(rows*cols), rows)
                  rowSums(x)
                },
                replications = 5,
                columns = c("test",  "elapsed","relative"))

kable(bm)
```

	test	elapsed	relative
2	R	0.211	1.000
1	Rcpp	0.394	1.867