Algorithmic Project: SMO and SGD algorithms - Group 7

Do Thanh Dat LE, Piseth KHENG

April 2024

1 Introduction

In the field of machine learning, Support Vector Machine (SVM) is a well-known model for classification and regression tasks due to its high efficiency and accuracy. Our project focuses on optimizing SVM's performance through two main algorithms: Sequential Minimal Optimization (SMO) and Stochastic Gradient Descent (SGD). The aim of our study is to improve SVM's training by effectively solving the quadratic programming problem. Additionally, we conducted an experiment to compare the time complexity of both algorithms and determine which one performs better. To test algorithm complexity, we simulated data with increasing n and d dimensions using R and Rcpp.

2 Overview of Support Vector Machines - SVM

The Support Vector Machines is a supervised learning technique invented by Vladimir Vapnik in 1979 [V.82] to solve problems of classification and regression. Its simplest is the linear form, an Support Vector Machines (SVM) is a hyperplane that separates a set of positive examples from a set of negative examples with maximum margin. In the linear case, the margin is defined by the distance of the hyperlane to the nearest of the positive and negative examples. The distance is illustrated in the **Figure 1**.

The output of the linear SVM is

$$y = w^T x + b$$

where w is the normal vector to the hyperplane and x is the input vector. The separating hyperplane is the plane $y = 0$. The nearest points lie on the plane $y = \pm 1$. The margin is thus

$$m = \frac{1}{\|w\|_2}$$

Then maximizing margin lead to the following primal optimization problem

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

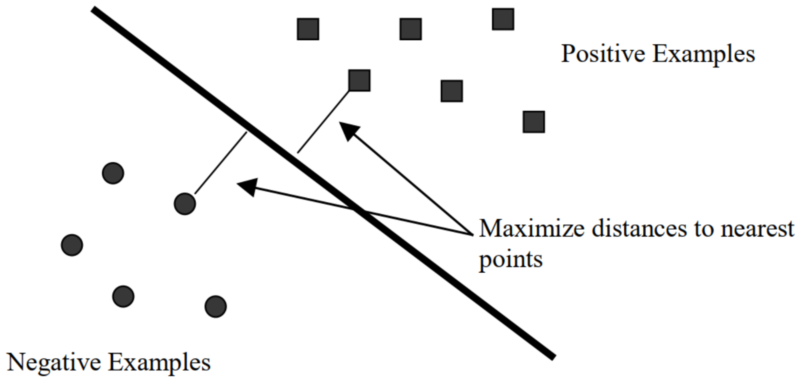


Figure 1: A linear SVM

subject to

$$y_i(w^T x_i + b) \geq 1, \forall i$$

where x_i is the i-th training example and y_i is the correct output of the SVM for the i-th training example. The value y_i is $+1$ for the positive examples and -1 for the negative examples.

However, not all data sets are linearly separable, there may be no hyperplane that splits the data well. Therefore, Cortes & Vapnik][CC95] suggested a modification to the original optimization, which penalizes the failure of an example to reach the correct margin. Then we have the following primal optimization problem

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} \xi_i$$

subject to

$$y_i(w^T x_i + b) \geq 1 - \xi_i, \forall i$$

$$\xi_i \geq 0$$

where ξ_i are variables which permit margin failure and C is a parameter trades off wide margin with a small number of margin failures. Using Lagrangian, this optimization problem could be converted into a following dual form with kernel function K, which is a quadratic programming (QP) problem

$$\max_{\alpha} W_\alpha = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

subject to

$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

$$0 \leq \alpha_i \leq C; \; i = 1, 2, ..., n$$

where K is a kernel function that measures the similarity or distance between the vectors x_i and x_j

The Karush-Kuhn-Tucker (KKT) conditions for this QP problem is

$$\alpha_i = 0 \Rightarrow y_i(w^T x_i + b) \geq 1$$

$$\alpha_i = C \Rightarrow y_i(w^T x_i + b) \leq 1$$

$$0 < \alpha_i < C \Rightarrow y_i(w^T x_i + b) = 1$$

3 Sequential minimal optimization - SMO

Since the dual optimization problem of SVM being presented in the previous section is a constrained optimization problem, applying gradient-based optimization methods could be challenging and require specialized techniques to handle the constraints effectively. Moreover, computing the gradient for each α_i could be computationally expensive, especially for large datasets where the number of α_i (equal to the number of training examples) is high. In addition, storing and manipulating the entire large gradient vector for all α values may cause memory problems.

Therefore, Sequential minimal optimization (SMO) is an algorithm proposed by John C. Platt [C.98] in 1998 for solving the QP problem in SVM quickly without any extra matrix storage and without using numerical QP optimization steps. It then became the fastest quadratic programming optimization algorithm, especially for linear SVM and sparse data performance. SMO decomposes the overall QP problem into QP sub-problems, which optimize a minimal subset of just a pair of α_i corresponding to 2 training examples at each iteration.

The convergence of the SMO algorithm is ensured by Osuna's Theorem [OE97], which suggested that the large QP problem could be broken down into a series of smaller QP sub-problems. As long as at least one example violating the Karush-Kuhn-Tucker (KKT) conditions is incorporated into the examples for the preceding sub-problem, each step will progressively reduce the overall objective function while maintaining a feasible point that adheres to all constraints. Consequently, a succession of QP sub-problems that consistently introduce at least one violator is ensured to converge.

Suppose that we have the set of α_i that satisfy the constraints, if we hold $\alpha_2, ..., \alpha_n$ fixed and reoptimize $W(\alpha)$ with respect to just the parameter α_1. We could not update the α_1 in this case since the Lagrange multipliers α_i must satisfy a linear constraint

$$\sum_{i=1}^{n} \alpha_i y_i = 0,$$

which means

$$\alpha_1 = -y_1 \sum_{i=2}^{n} \alpha_i y_i,$$

Hence, α_1 is determined by the other α_i, and if we hold $\alpha_2, ..., \alpha_n$ fixed, then we could not make any update to α_1 without violating this constraint. Therefore, if we want to update the α_i, we must update at least two of them simultaneously in order to keep satisfying the constraint, this motivate the SMO algorithm. At each step, SMO selects a pair of Lagrange multipliers to simultaneously optimize, determines the optimal values for these multipliers, and then adjusts the SVM accordingly to incorporate the updated optimal values. SMO consists of two main components: an analytical approach for computing the optimal values of two Lagrange multipliers, and a heuristic for selecting which pair of multipliers to optimize at each iteration.

3.1 Heuristics for choosing multipliers α to optimize

Choosing which α_i and α_j to optimize is important for the speed of the algorithm, especially for large data sets, since there are $n(n-1)$ possible choices for α_i and α_j, and some choices results in much less improvement than others. In this project, we iterate over all α_i, $i = 1, ..., n$ and choose the α_i which is not satisfied the KKT conditions within a numerical tolerance, we select α_j random from the remaining $n-1$ α then we try to jointly optimize α_i and α_j.

3.2 Optimizing Two Lagrange Multipliers

Without loss of generality we will assume that α_1, α_2 have been chosen for optimizing. In order not to violate the linear constraint $\sum_{i=1}^{n} \alpha_i y_i = 0$ the new values of the multipliers must be on a line

$$y_1\alpha_1^{(\text{old})} + y_2\alpha_2^{(\text{old})} = \text{constant} = y_1\alpha_1 + y_2\alpha_2$$

in (α_1, α_2) space, and in the box defined by $0 \leq \alpha_1, \alpha_2 \leq C$ as shown in the **Figure 2**

Without loss of generality, the algorithm first computes the second Lagrange multiplier α_2 and use it to obtain α_1. The box constraint $0 \leq \alpha_1, \alpha_2 \leq C$ with the linear equality constraint provide a more strict bound on α_2

$$L \leq \alpha_2^{(\text{new})} \leq H,$$

where L and H are defined as follows:

- If $y_1 \neq y_2$:

$$L = \max(0, \alpha_2^{(\text{old})} - \alpha_1^{(\text{old})}), H = \min(C, C + \alpha_2^{(\text{old})} - \alpha_1^{(\text{old})})$$

- If $y_1 = y_2$:

$$L = \max(0, \alpha_2^{(\text{old})} + \alpha_1^{(\text{old})} - C), H = \min(C, \alpha_2^{(\text{old})} + \alpha_1^{(\text{old})})$$

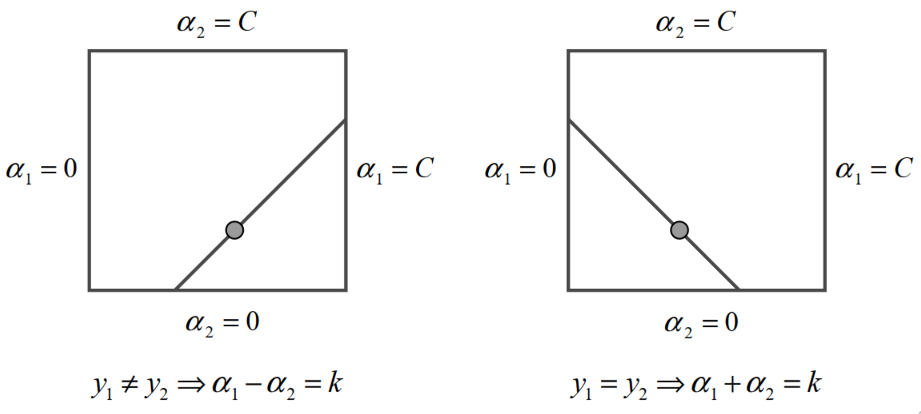


Figure 2: The two Lagrange multipliers must satisfy the equality and inequality constraints

The second derivative of the objective function along the diagonal line can be expressed as:

$$\kappa = -K(x_1, x_1) - K(x_2, x_2) + 2K(x_1, x_2) \leq 0$$

The $\kappa = 0$ if more than one training example has the same input vector x. Then we could compute the new unconstrained α_2 :

$$\alpha_2^{(\text{new})} = \alpha_2^{(\text{old})} + \frac{y_2 \left(E_2^{(\text{old})} - E_1^{(\text{old})} \right)}{\kappa},$$

where $E_i = f(x_i) - y_i = \left(\sum_{j=1}^{n} \alpha_i y_i K_{ij} + b \right) - y_i$ is the error on the i-th training example. As a next step, the $\alpha_2^{(\text{new})}$ is found by apply the bound of α_2:

$$\alpha_2^{(\text{new})} = \begin{cases} H & \text{if} \quad \alpha_2^{(\text{new})} \geq H; \\ \alpha_2^{(\text{new})} & \text{if} \quad L < \alpha_2^{(\text{new})} < H; \\ L & \text{if} \quad \alpha_2^{(\text{new})} \leq L \end{cases}$$

Then the value of $\alpha_1^{(\text{new})}$ is obtained from $\alpha_2^{(\text{new})}$ as

$$\alpha_1^{(\text{new})} = \alpha_1^{(\text{old})} + y_1 y_2 \left(\alpha_2^{(\text{old})} - \alpha_2^{(\text{new})} \right)$$

3.3 Computing the threshold b

After we optimize α_i and α_j, we compute the threshold b that the KKT conditions are satisfied for the i-th and j-th examples. This threshold b is recomputed after each step to ensure the KKT conditions are satisfied for both examples.

When the new α_1 is not at the bounds $(0 < \alpha_i < C)$, the following threshold b_1 is valid, it forces the SVM to output y_i when the input is x_i

$$b_1 = b - E_i - y_i(\alpha_i - \alpha_i^{(\text{old})})K(x_i, x_i) - y_j(\alpha_j - \alpha_j^{(\text{old})})K(x_i, x_j)$$

Similarly, the threshold b_2 is valid if $0 < \alpha_j < C$

$$b_2 = b - E_j - y_i(\alpha_i - \alpha_i^{(\text{old})})K(x_i, x_j) - y_j(\alpha_j - \alpha_j^{(\text{old})})K(x_j, x_j)$$

If $0 < \alpha_i, \alpha_j < C$ then both these thresholds are valid and they are equal. If both new α are at the bounds then all the thresholds between b_1 and b_2 satisfy the KKT conditions, so we let $b = \dfrac{b_1 + b_2}{2}$. Totally, we have

$$b = \begin{cases} b_1 & \text{if} \quad 0 < \alpha_i < C; \\ b_2 & \text{if} \quad 0 < \alpha_j < C; \\ \dfrac{b_1 + b_2}{2} & \text{otherwise} \end{cases}$$

3.4 Pseudocode for the SMO algorithm

Algorithm 1 Sequential minimal optimization - SMO

```
Input:
  $(x_1, y_1), \dots, (x_n, y_n)$: Training data
  $max\_iter$: Number of times to iterate over $\alpha$ without changing
  $C$: Regularization parameter
  $tol1$: numerical tolerance for choosing $\alpha_i$
  $tol2$: numerical tolerance for the distance between new and old $\alpha_j$
Output:
  $\alpha \in \mathbb{R}^n$: Lagrange multipliers for solution
  $b \in \mathbb{R}$: threshold for solution
Initialize $\alpha_i \leftarrow 0$, $\forall i$, $b \leftarrow 0$
Initialize $iter \leftarrow 0$
while $iter < max\_iter$ do
    $num\_changed\_alphas = 0$
    for $i \leftarrow 1$ to $n$ do
        Calculate $E_i = f(x_i) - y_i$
        if $(y_i E_i < -tol1 \;\&\; \alpha_i < C) \,||\, (y_i E_i > tol1 \;\&\; \alpha_i > 0)$ then
            Select $j \neq i$ randomly
            Calculate $E_j = f(x_j) - y_j$
            Save old $\alpha$: $\alpha_i^{(old)} = \alpha_i$, $\alpha_j^{(old)} = \alpha_j$
            Compute $L$ and $H$ for $\alpha_j$
            if L == H then
                Continue to the next $i$
            end if
            Compute $\kappa$
            if $\kappa \geq 0$ then
                Continue to the next $i$
            end if
            Compute the new value for $\alpha_j$
            if $|\alpha_j - \alpha_i^{(old)}| < tol2$ then
                Continue to the next $i$
            end if
            Compute new $\alpha_i$ from new $\alpha_j$
            Compute threshold $b_1$ and $b_2$, then compute $b$
            $num\_changed\_alphas \leftarrow num\_changed\_alphas + 1$
        end if
    end for
    if $num\_changed\_alphas == 0$ then
        $iter \leftarrow iter + 1$
    else
        $iter \leftarrow 0$
    end if
end while
return $\alpha, b$
```

3.5 Time complexity

To compute the time complexity of the SMO, we can first look at the $linear_kernel$ function, which involves a matrix multiplication of X and X^T, representing the input data. Since X is a matrix of $n \times d$ this operation has a complexity of $O(n^2 d)$.

Moreover, we assume that there are m iterations in the while loop. Additionally, the while loop contains a for loop that has n iterations. Inside the for loop, there is a sum function with a complexity of $O(n)$.

Therefore we can get the worst-case time complexity below:

$$T(n) = O(n^2 d) + O(m \times n \times n) = O(n^2 p)$$

Where $p = max(m, d)$

In the special case when the training data X has only one or few features and the value of m is too small, then we can get the best-case complexity such that:

$$T(n) = \Omega(n^2 m)$$

4 Stochastic gradient descent - SGD

Stochastic Gradient Descent (SGD) is an optimization algorithm that is widely used in machine learning and deep learning to minimize the loss function of the model. Unlike the older Gradient Descent method that looks at all the data at once, SGD randomly selects small parts of the data to make the process faster and more efficient, especially for large datasets. This method keeps changing the model's settings little by little to get the best results. Stochastic gradient descent's convergence has been studied through the theories of stochastic approximation and convex minimization.

As previously mentioned, the primal optimization problem of SVM can be converted into this form:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} \max(0, 1 - y_i(w^T x_i + b))$$

With this objective function, the SGD will optimize it by iteratively updating the model parameters w and b based on a subset of the training data at each step. The update equations for the weight vector w and bias b at iteration t are:

$$w_{t+1} = w_t - \eta_t \nabla_w J(w_t, b_t)$$

$$b_{t+1} = b_t - \eta_t \nabla_b J(w_t, b_t)$$

Where η_t is the learning rate at iteration t, $\nabla_w J$ and $\nabla_b J$ are the gradients of the objective function J with respect to w and b. The gradient for the weight and bias for the misclassified point when $y_i(w^T x_i + b) < 1$ are:

$$\nabla_w J = w - C \sum_{i=1}^{N} y_i x_i$$

$$\nabla_b J = -C \sum_{i=1}^{N} y_i$$

For the points that are correctly classified and satisfy $y_i(w^T x_i + b) \geq 1$, the gradient of the loss with respect to w is simply w, and the gradient with respect to b is 0.

Moreover, the update rules for w and b during each iteration of SGD depending on whether a training sample is correctly classified or not.

- If $y_i(w^T x_i + b) < 1$:

$$w_{t+1} = w_t - \eta_t (w_t - C y_i x_i)$$

$$b_{t+1} = b_t + \eta_t C y_i$$

- If $y_i(w^T x_i + b) \geq 1$:

$$w_{t+1} = w_t - \eta_t w_t$$

$$b_{t+1} = b_t$$

These updated rules are applied iteratively over random subsets of the training data until convergence, optimizing the SVM to find a decision boundary that maximizes the margin between different classes while penalizing misclassification.

However, the effectiveness of SGD for SVM depends on the characteristics of the data:

- When dealing with sparse and low-dimensional data, linearly separable data leads to quick convergence. Furthermore, smaller datasets and well-preprocessed features also contribute to faster training.

- For the dense, high-dimensional, and noisy data that isn't linearly separable can slow down convergence. Larger datasets and poorly scaled features can also result in longer training times.

4.1 Pseudocode for the SGD algorithm

Algorithm 2 Stochastic Gradient Descent for Support Vector Machine

```
Input:
    $X$: Training data features, matrix of shape $(n_{samples}, n_{features})$
    $y$: Training data labels, vector of length $n_{samples}$
    $epochs$: Number of passes over the training data
    $C$: Regularization parameter
    $learning\_rate$: Learning rate for updates
    $tolerance$: Threshold for stopping criteria
Output:
    $w$: Learned weight vector
    $b$: Learned bias term
$w \leftarrow$ vector of zeros of shape$(n_{features})$
$b \leftarrow 0$
$previous\_loss \leftarrow \infty$
for $epoch \leftarrow 1$ to $epochs$ do
    $total\_loss \leftarrow 0$
    for $i \leftarrow 1$ to $n_{samples}$ do
        $random\_index \leftarrow \textsc{RandomInteger}(1, n_{samples})$
        $x_i \leftarrow X[random\_index]$
        $y_i \leftarrow y[random\_index]$
        $margin \leftarrow y_i \cdot (\textsc{DotProduct}(w, x_i) + b)$
        if $margin < 1$ then
            $w\_gradient \leftarrow w - C \cdot y_i \cdot x_i$
            $b\_gradient \leftarrow -C \cdot y_i$
            $total\_loss \leftarrow total\_loss + (1 - margin)$
        else
            $w\_gradient \leftarrow w$
            $b\_gradient \leftarrow 0$
        end if
        $w \leftarrow w - learning\_rate \cdot w\_gradient$
        $b \leftarrow b - learning\_rate \cdot b\_gradient$
    end for
    $average\_loss \leftarrow total\_loss / n_{samples}$
    if $|previous\_loss - average\_loss| < tolerance$ then
        break
    end if
    $previous\_loss \leftarrow average\_loss$
    $learning\_rate \leftarrow learning\_rate \cdot DecayFactor$
end for
return $w, b$
```

4.2 Time complexity

In the SGD function, it contains two loops for the operation. The first one iterates over a fixed number of epochs until the convergence is reached, and its complexity is $O(e)$. Moreover, the second loop has `n` iterations with has the complexity of $O(n)$. Updating the weights and bias in the case of a misclassified sample or a sample within the margin also has a complexity of O(d) because it requires an element-wise operation over all features.

The worst-case time complexity of this function is given by:

$$T(n) = O(e \times n \times d) = O(end)$$

Furthermore, if we train our model on a small set of features and for only a few epochs, we can consider it a constant, the best-case scenario for this algorithm:

$$T(n) = \Omega(n)$$

References

[C.98] Platt J. C. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines". Technical Report MSR-TR-98-14*, 1998.

[CC95] Vapnik V. Cortes C. *Support Vector Networks. Machine Learning*, 1995.

[OE97] Girosi F. Osuna E., Freund R. *Improved Training Algorithm for Support Vector Machines. Proc. IEEE NNSP '97*, 1997.

[V.82] Vapnik V. *Estimation of Dependences Based on Empirical Data. Springer-Verlag*, 1982.

Algorithmique Projet: SMO and SGD algorithm in SVM classification - M2DS

Do Thanh Dat LE, Piseth KHENG

Evry University, Paris-Saclay University

April 10, 2024

Testing Algorithms

```r
library(SVMalgo)
library(microbenchmark)
```

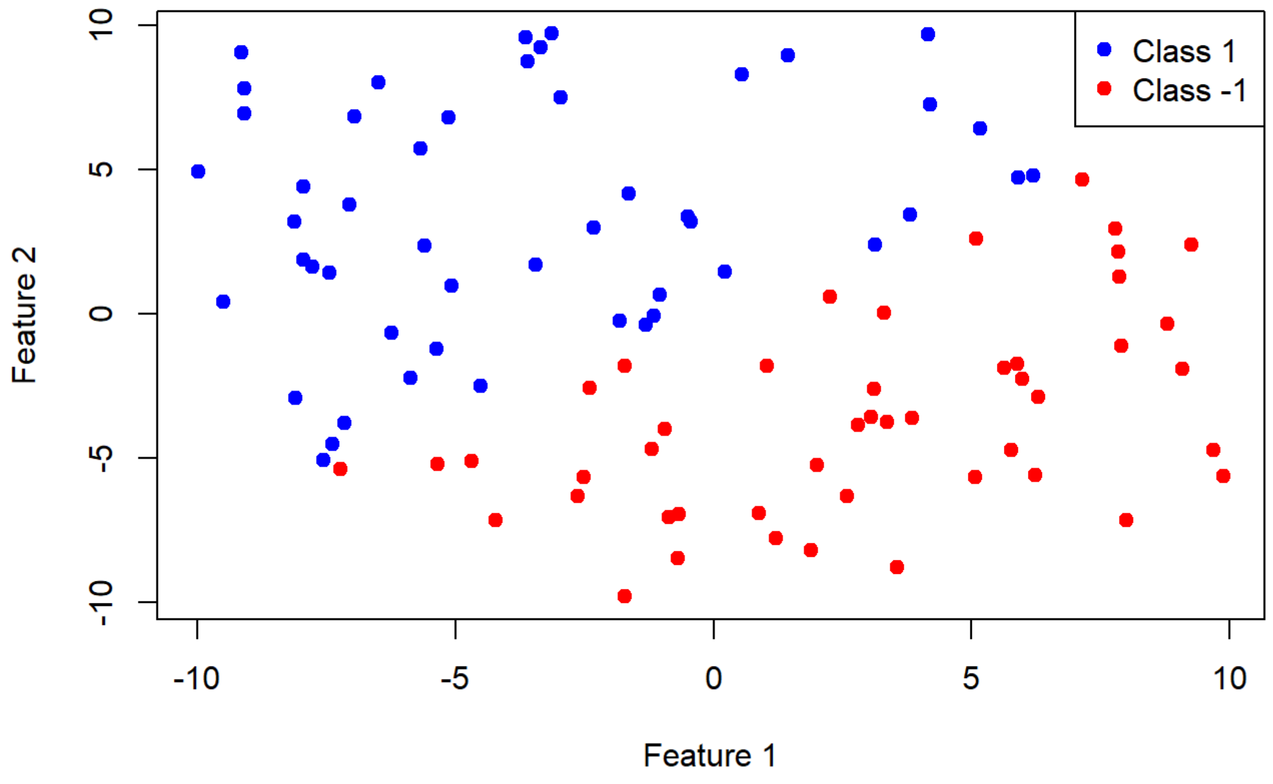
```
## Warning: package 'microbenchmark' was built under R version 4.3.3
```

```r
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.3.3
```

We simulate a linearly separable data set. The input data was random uniform m-dimensional vectors. A m-dimensional weight vector was generated randomly in [-10,10]. If the dot product of the weight with an input point is greater than 0, then a positive label 1 is assigned to the input point. If the dot product is less than 0, then a negative label -1 is assigned.

```
linearly_separable_data <- function(n_samples, n_features, noise = 0) {
  # Generate random coefficients for the linear equation
  coefficients <- runif(n_features, -10, 10)

  # Generate random data points
  X <- matrix(runif(n_samples * n_features, -10, 10), ncol = n_features)

  # Compute the target variable (labels) based on the linear equation
  y <- ifelse(X %*% coefficients > 0, 1, -1)

  # Add noise if specified
  if (noise > 0) {
    y <- y * rnorm(length(y), mean = 1, sd = noise)
  }

  # Return the data
  return(list(X = X, y = y))
}

# Example usage:
set.seed(123)  # for reproducibility
data <- linearly_separable_data(n_samples = 100, n_features = 2, noise = 0)

# Plot the data
plot(data$X, col = ifelse(data$y > 0, "blue", "red"), pch = 19, xlab = "Feature 1", ylab = "F
eature 2")
legend("topright", legend = c("Class 1", "Class -1"), col = c("blue", "red"), pch = 19)
```



We define a function `one.simu` to simplify the simulation study for time complexity.

```
one.simu <- function(n, num_features = 10, type = "sample", func = "svm_smo")
{
  set.seed(123)
  if(type == "sample"){data <- linearly_separable_data(n, num_features)}
  if(func == "svm_smo"){t <- system.time(svm_smo(data$X, data$y, C = 1, tol = 1e-3, max_passe
s = 1, kernel_fun = linear_kernel))[[1]]}
  if(func == "svm_sgd"){t <- system.time(svm_sgd(data$X, data$y, epochs = 100, C = 1, learnin
g_rate = 1e-2, tolerance = 1e-3))[[1]]}
  if(func == "svm_smo_Rcpp"){t <- system.time(svm_smo_Rcpp(data$X, data$y, C = 1, tol = 1e-3,
max_passes = 1))[[1]]}
  if(func == "svm_sgd_Rcpp"){t <- system.time(svm_sgd_Rcpp(data$X, data$y, epochs = 100, C =
1, learning_rate = 1e-2, tolerance = 1e-3))[[1]]}
  return(t)
}
```

First, we evaluate the running time of the algorithms on the simulated data with the size of 1000 observations and only 2 features (the simplest case).

```
n <- 1000
```

and we get:

```
one.simu(n, num_features = 2, func = "svm_smo")
```

```
## [1] 1.62
```

```
one.simu(n, num_features = 2, func = "svm_sgd")
```

```
## [1] 0.05
```

```
one.simu(n, num_features = 2, func = "svm_smo_Rcpp")
```

```
## [1] 0.65
```

```
one.simu(n, num_features = 2, func = "svm_sgd_Rcpp")
```

```
## [1] 0
```

The Rcpp is faster than R in both SMO and SGD.

Comparisons

we compare the running time with repeated executions (`nbSimus` times)

```
nbSimus <- 10
time1 <- 0; time2 <- 0; time3 <- 0; time4 <- 0

for(i in 1:nbSimus){time1 <- time1 + one.simu(n, num_features = 2, func = "svm_smo")}
for(i in 1:nbSimus){time2 <- time2 + one.simu(n, num_features = 2, func = "svm_sgd")}
for(i in 1:nbSimus){time3 <- time3 + one.simu(n, num_features = 2, func = "svm_smo_Rcpp")}
for(i in 1:nbSimus){time4 <- time4 + one.simu(n, num_features = 2, func = "svm_sgd_Rcpp")}
```

```
#gain R -> Rcpp
time1/time3
```

```
## [1] 2.008237
```

```
time2/time4
```

```
## [1] 9.333333
```

Rcpp is faster than R for our 2 algorithms.

```
#gain smo -> sgd
time1/time2
```

```
## [1] 43.53571
```

```
time3/time4
```

```
## [1] 202.3333
```

With the data length of `1000` and 2 features, SGD runs faster than SMO.

```
#max gain
time1/time4
```

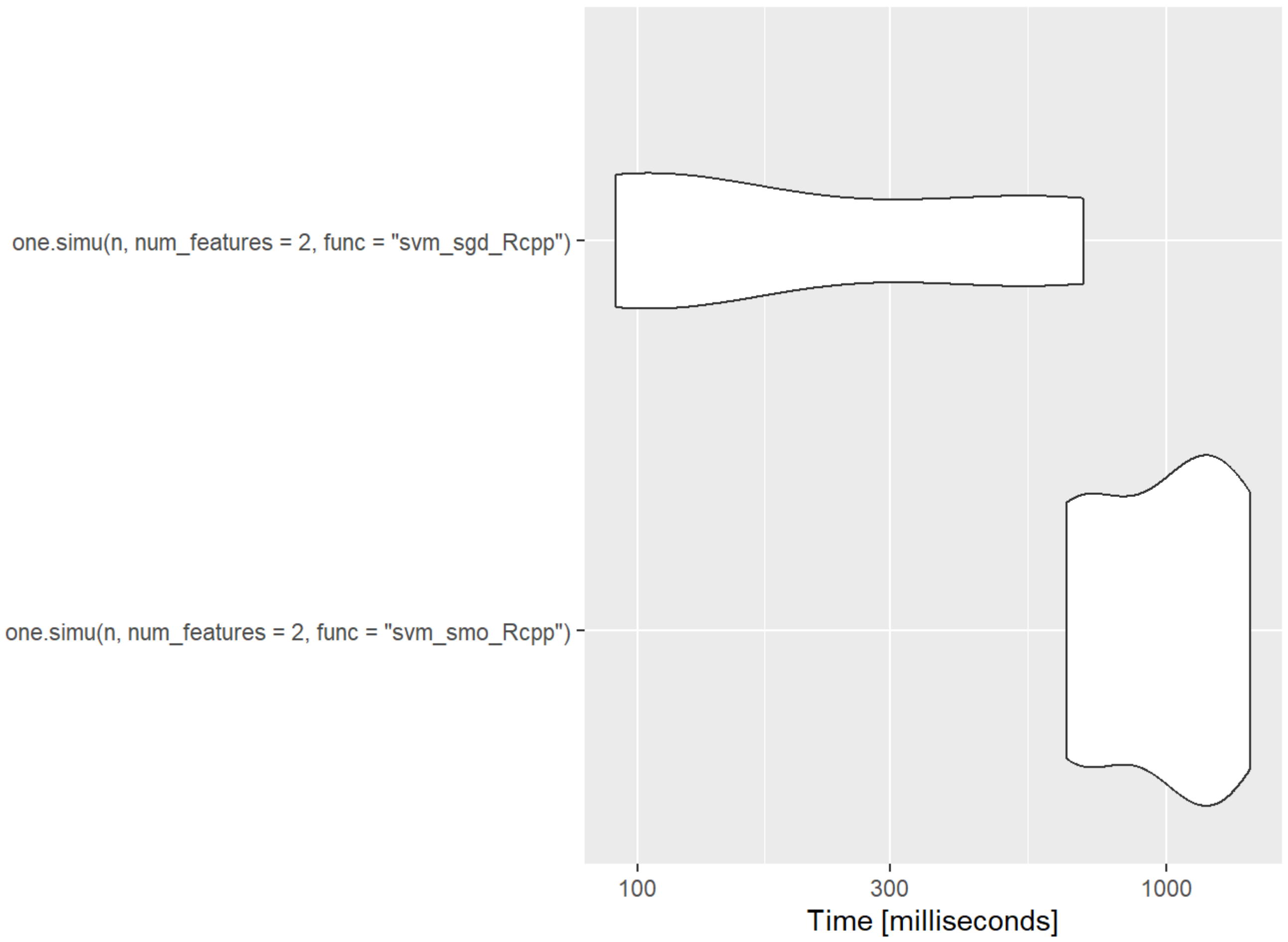
```
## [1] 406.3333
```

The gain between the slow SMO R algorithm and the faster SGD Rcpp algorithm is of order 200

Microblenchmark

We compare `svm_smo_Rcpp` with `svm_sgd_Rcpp` for data lengths `n = 1000` and with only 2 features.

```
n <- 1000
res <- microbenchmark(one.simu(n, num_features = 2, func = "svm_smo_Rcpp"), one.simu(n, num_f
eatures = 2, func = "svm_sgd_Rcpp"), times = 5)
autoplot(res)
```



```
res
```

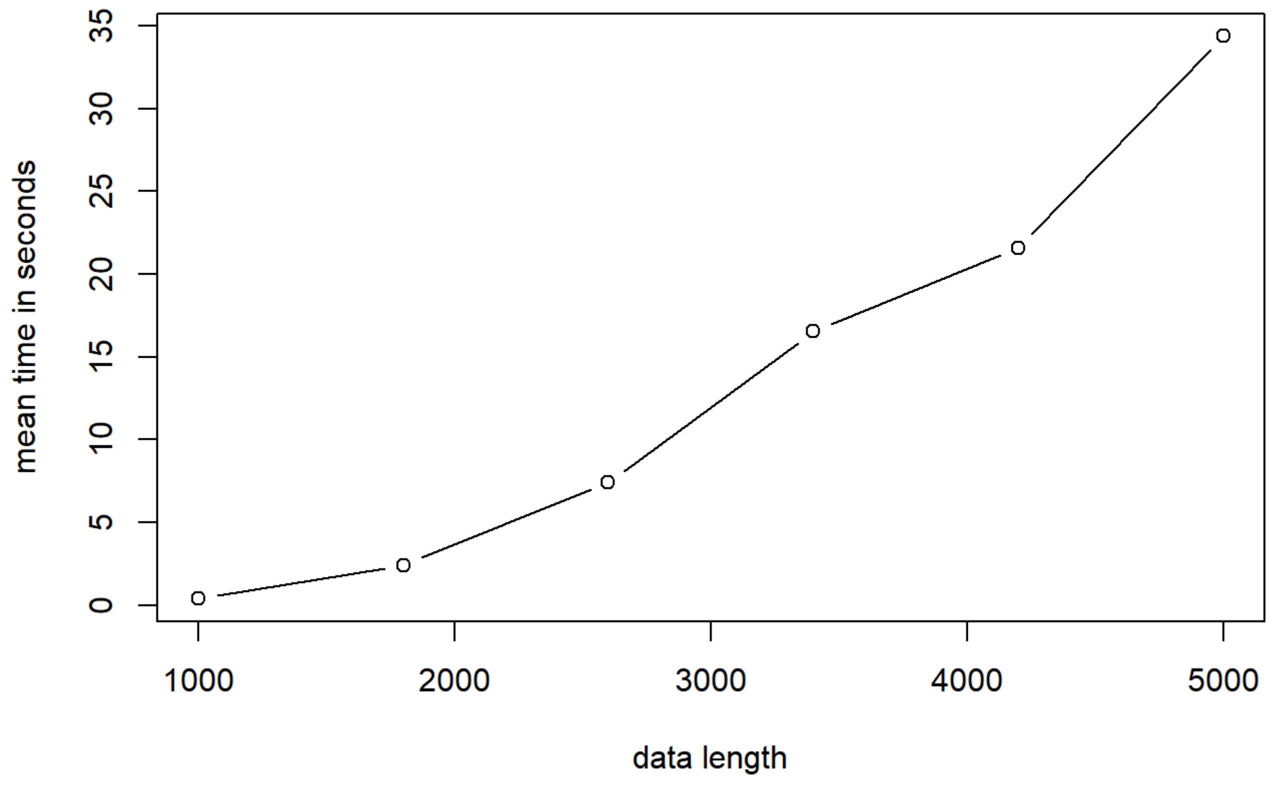
```
## Unit: milliseconds
##                                                  expr      min       lq
##  one.simu(n, num_features = 2, func = "svm_smo_Rcpp") 649.3084 707.5911
##  one.simu(n, num_features = 2, func = "svm_sgd_Rcpp")  90.9001 101.1078
##       mean    median        uq       max neval
##  1016.2229 1138.6147 1142.6336 1442.9668     5
##   300.6327  112.3601  502.4334  696.3619     5
```

At this data length `1000` and only 2 features, we have a robust difference in running time between SMO_Rcpp and SGD_Rcpp.

Time complexity

We run `nbRep = 5` times the `svm_smo_Rcpp` algorithm of each value of the `vector_n` vector of length `nbSimus = 11`. We show the plot of the mean running time with respect to data length.

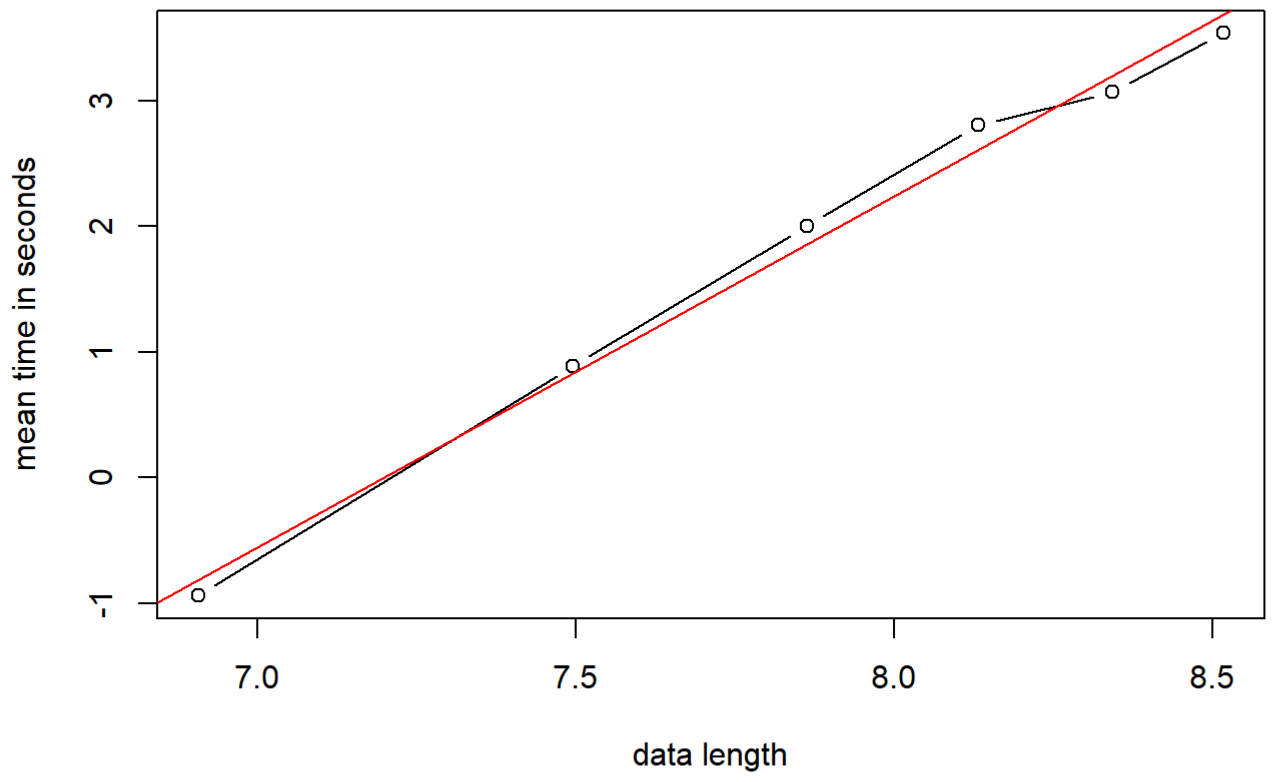
```
set.seed(123)
nbSimus <- 6
vector_n <- seq(from = 1000, to = 5000, length.out = nbSimus)
nbRep <- 3
res_smo <- data.frame(matrix(0, nbSimus, nbRep + 1))
colnames(res_smo) <- c("n", paste0("Rep",1:nbRep))

j <- 1
for(i in vector_n)
{
  res_smo[j,] <- c(i, replicate(nbRep, one.simu(i, num_features = 2, func = "svm_smo_Rcpp")))
  #print(j)
  j <- j + 1
}

res.smo <- rowMeans(res_smo[,-1])
plot(vector_n, res.smo, type = 'b', xlab = "data length", ylab = "mean time in seconds")
```



```
fit.smo <- lm(log(res.smo) ~ log(vector_n))
summary(fit.smo)
```

```
##
## Call:
## lm(formula = log(res.smo) ~ log(vector_n))
##
## Residuals:
##        1        2        3        4        5        6
## -0.12875  0.05574  0.14557  0.19965 -0.12573 -0.14648
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)   -20.1129     1.0173  -19.77 3.86e-05 ***
## log(vector_n)   2.7940     0.1289   21.68 2.68e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1718 on 4 degrees of freedom
## Multiple R-squared:  0.9916, Adjusted R-squared:  0.9895
## F-statistic: 470.2 on 1 and 4 DF,  p-value: 2.676e-05
```

```
plot(log(vector_n), log(res.smo), type = 'b', xlab = "data length", ylab = "mean time in seco
nds")
abline(fit.smo, col='red')
```

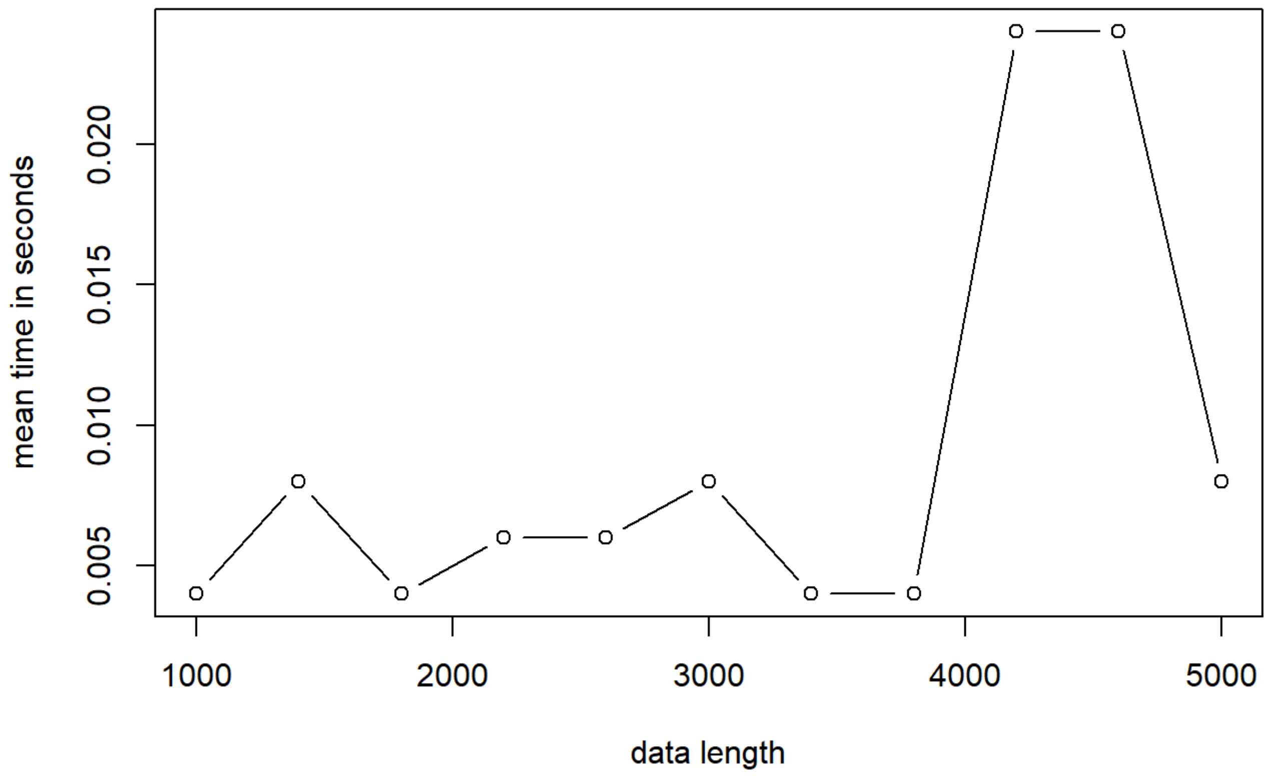


$$log(T(n)) = C_1 + C_2 * log(n)$$

we could see that the $log(T(n)) \approx log(n^{C_2})$ which is the similar to the best time complexity $\Omega(n^2)$ of the SMO algorithm. It is reasonable because we are trying the SMO algorithm on the linearly separate data sets with only 2 features and m = max_passes = 1 (which nearly the best case).

Same strategy but with the `svm_sgd_Rcpp` algorithm.

```
set.seed(123)
nbSimus <- 11
vector_n <- seq(from = 1000, to = 5000, length.out = nbSimus)
nbRep <- 5
res_sgd <- data.frame(matrix(0, nbSimus, nbRep + 1))
colnames(res_sgd) <- c("n", paste0("Rep",1:nbRep))

j <- 1
for(i in vector_n)
{
  res_sgd[j,] <- c(i, replicate(nbRep, one.simu(i, num_features = 2, func = "svm_sgd_Rcp
p"),))
  #print(j)
  j <- j + 1
}

res.sgd <- rowMeans(res_sgd[,-1])
plot(vector_n, res.sgd, type = 'b', xlab = "data length", ylab = "mean time in seconds")
```



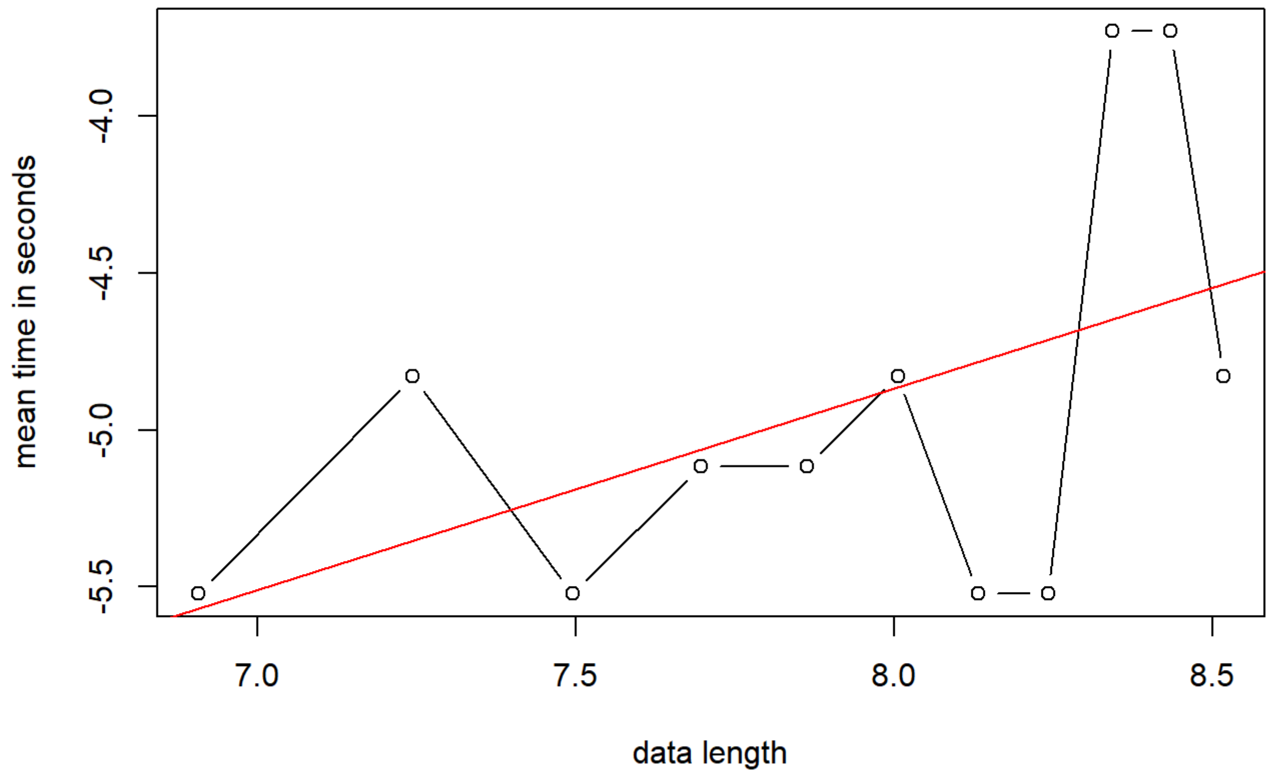
```
fit.sgd <- lm(log(res.sgd) ~ log(vector_n))
summary(fit.sgd)
```

```
##
## Call:
## lm(formula = log(res.sgd) ~ log(vector_n))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.80955 -0.31129 -0.05339  0.28569  0.91799
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)   -10.0009     2.9278  -3.416  0.00768 **
## log(vector_n)   0.6417     0.3700   1.734  0.11688
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.604 on 9 degrees of freedom
## Multiple R-squared:  0.2505, Adjusted R-squared:  0.1672
## F-statistic: 3.008 on 1 and 9 DF,  p-value: 0.1169
```

$$log(T(n)) = C_1 + C_2 log(n)$$

we could see that the $log(T(n)) \approx log(n^{C_2})$ which is the similar to the best time complexity $\Omega(n)$ of the SGD algorithm. It is reasonable because we are trying the SGD algorithm on the linearly separate data sets with only 2 features (which nearly the best case).

```
plot(log(vector_n), log(res.sgd), type = 'b', xlab = "data length", ylab = "mean time in seco
nds")
abline(fit.sgd, col='red')
```


mean time in seconds
data length