Soft margin SVM

Autor: Jan Retkowski

Importowanie bibliotek

```
library(datasets)
library(tidyr)
library(dplyr)
library(quadprog)
library(Matrix)

data("iris")
set.seed(0)
```

Implementacja SVM

Trenowanie maszyny wektorów nośnych sprowadza się do maksymalizacji funkcji wyrażającej szerokość marginesu wyrażonego wzorem arg max $\frac{2}{||w||}$, gdzie w to wektor prostopadły do hiperpłaszczyzny dzielądzej zbiory oraz $y_i(x_i \bullet w + b) \geq 1 - \xi_i$ i $\xi_i \geq 0$

Następnie można zastowować przekształcenia w celu ułatwienia dalszych obliczeń:

$$\arg\max \frac{2}{||w||} \Rightarrow \arg\min ||w|| \Rightarrow \arg\min \frac{1}{2}||w||^2$$

Dodatkowo by dozwolić na wykonywanie pomyłek, należy uwzględnić parametr ξ reprezentujący wielkość owej pomyłki. arg min $\frac{1}{2}||w||^2 + C\sum_{i=0}^{n}\xi_i$, gdzie C to hiperparametr (czułość na pomyłki)

By rozwiązać problem optymalizacji z ograniczeniami należy zapisać powyższe wyrażenie z zastosowaniem mnożników Lagrange'a:

$$L = \frac{1}{2}||w||^2 + C\sum_{i=0}^{n}\xi_i - \sum_{i=0}^{n}\alpha_i[y_i(x_i \bullet w + b) - 1 + \xi_i)] - \sum_{i=0}^{n}\lambda_i\xi_i$$

gdzie $\alpha_i, \lambda_i \geq 0$

Rozwiązaniem jest punkt w którym pochodne cząstkowe lagrangianu przyjmują wartość 0

$$\frac{\partial L}{\partial w} = w - \sum_{i=0}^{n}\alpha_i x_i y_i = 0 \Rightarrow w = \sum_{i=0}^{n}\alpha_i x_i y_i$$

$$\frac{\partial L}{\partial b} = -\sum_{i=0}^{n}\alpha_i y_i = 0 \Rightarrow \sum_{i=0}^{n}\alpha_i y_i = 0$$

$$\frac{\partial L}{\partial \xi} = \sum_{i=0}^{n} C - \sum_{i=0}^{n} \alpha_i - \sum_{i=0}^{n} \lambda_i = 0 => C - \alpha_i - \lambda_i = 0 \Rightarrow \lambda_i = C - \alpha_i$$

Jako że $\alpha_i, \lambda_i \geq 0$, to $0 \leq C - \alpha_i \ \wedge 0 \leq \alpha_i \Rightarrow 0 \leq \alpha_i \leq C$

Po podstawieniu zmiennych postać lagrangianu wygląda następująco:

$$L = \frac{1}{2} \sum_{i=0}^{n} \alpha_i - \sum_{i=0}^{n} \sum_{j=0}^{n} \alpha_i \alpha_j y_i y_j x_i \bullet x_j$$

Ostatnim krokiem wyprowadzenia jest zastosowanie funkcji $K(x_i, w) = \langle\ \phi(x_i), \phi(w) \rangle$, gdzie $\langle\ \phi(x_i), \phi(w) \rangle$, to produkt wewnętrzny wektrów po przekształceniu $\phi(x)$

$$L = \frac{1}{2} \sum_{i=0}^{n} \alpha_i - \sum_{i=0}^{n} \sum_{j=0}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j)$$

$$w = \sum_{i=0}^{n} \alpha_i y_i \phi(x_i)$$

$$b = \mathbb{E}\left[\ y_k - w \bullet \phi(x_k)\ \right] = \mathbb{E}\left[\ y_k - \sum_{i=0}^{n} \alpha_i y_i \langle\ \phi(x_i), \phi(x_k)\ \rangle\ \right] = \mathbb{E}\left[\ y_k - \sum_{i=0}^{n} \alpha_i y_i K(x_i, x_k)\ \right]$$

Taki lagrangian należy następnie zmaksymalizować stosując sie do wcześniej wyznaczonych ograniczeń. Wynik będzie ograniczeniem dolnym wcześniejszego problemu minimalizacji. Wektory x_i dla których wartość mnożnika lagrange'a $\alpha_i \neq 0$ to poszuiwane wektory nośne.

Do sprawnego rozwiązania tego problemu zastosowany został algorym programowania kwadratowego z biblioteki `quadprog`. Przed użyciem funkcji `solve.QP` problem musi zostać zapisany w postaci $\min\ \frac{1}{2}\alpha^T D\alpha - d^T\alpha,\ \ A^T\alpha \geq 0$, gdzie D jest macierzą dodatnio określoną.

```
SVM.fit <- function(X, y, kernel = kernel.linear, C = .Machine$integer.max, d = 0, s = 1) {
    n.samples <- nrow(X)
    n.features <- ncol(X)
    SVM.model <- list()

    if (identical(kernel, kernel.linear))
        kernel <- function(x1, x2) kernel.linear(x1, x2, d)

    if (identical(kernel, kernel.RBF))
        kernel <- function(x1, x2) kernel.RBF(x1, x2, s)

    Kmat <- matrix(rep(0, n.samples*n.samples), nrow=n.samples)
    for (i in 1:n.samples){
        for (j in 1:n.samples){
            Kmat[i,j] <- kernel(X[i,], X[j,])
        }
    }
    Dmat <- outer(y, y) * Kmat
    Dmat <- as.matrix(nearPD(Dmat)$mat) # D must be positive definite matrix
    dvec <- rep(1, n.samples)
```

```
    Amat <- rbind(y, diag(n.samples), -diag(n.samples))
    bvec <- c(0, rep(0, n.samples), rep(-C, n.samples))

    result <- solve.QP(Dmat, dvec, t(Amat), bvec = bvec, meq = 1)
    lagrange_multipliers <- result$solution # Lagrange multipliers

    SVM.model$bias <- mean(y - apply(X, 1, function(x) sum(lagrange_multipliers * y
                                                           * apply(X, 1, function(a) kernel(a, x)))))
    SVM.model$kernel <- kernel
    SVM.model$data_y <- y
    SVM.model$data_X <- X
    SVM.model$lagrange <- lagrange_multipliers

    return(SVM.model)
}

SVM.predict <- function(SVM.model, x) {
    return(sum(SVM.model$lagrange * SVM.model$data_y
               * apply(SVM.model$data_X, 1, function(a) SVM.model$kernel(a, x)))
           + SVM.model$bias)

}
```

Definicja kerneli

```
kernel.linear <- function(x, y, d) x %*% y + d

kernel.RBF <- function(x, y, s) exp(-s * (x - y) %*% (x - y))
```

Funkcja mierząca procent poprawnych przewydywań

```
measure_fitness <- function(SVM.model, X, y) {
    res <- sum(apply(X, 1, function(x) sign(SVM.predict(SVM.model, x))) == y) / length(y)

    return(res)
}
```

Funkcja dzieląca dane na zbiory testowy, treningowy i walidacyjny

```
test_validation_train.split <- function(X, y, train_frac, test_frac, validate_frac = NULL) {

    train_size <- floor(train_frac * length(y))
    validate_size <- floor(validate_frac * length(y))
    test_size <- floor(test_frac * length(y))

    train_ind <- sample(length(y), size = train_size)

    train_X <- X[train_ind, ]
    train_y <- y[train_ind]

    non_train_X <- X[-train_ind, ]
    non_train_y <- y[-train_ind]

    if (is.null(validate_frac)) {
```

```
        return(list(train_X = train_X, train_y = train_y,
                test_X = non_train_X, test_y = non_train_y))
    }

    validate_ind <- sample(length(non_train_y), size = validate_size)

    validate_X <- non_train_X[validate_ind, ]
    validate_y <- non_train_y[validate_ind]

    test_X <- non_train_X[-validate_ind, ]
    test_y <- non_train_y[-validate_ind]


    return(list(train_X = train_X, train_y = train_y,
                validate_X = validate_X, validate_y = validate_y,
                test_X = test_X, test_y = test_y))
}
```

Testowanie algorytmu

Ponieważ SVM, to klasyfikator binarny, to należy podzielić dane na zbiory zawierające tylko dwie wartości w kolumnie wynikowej. Dodatkowo należy zakodować te dane zamieniając jeden z wyników na 1, a drugi na -1. By dodatkowo usprawnić algorytm można dokonać normalizacji danych danych.

Setosa vs Virginica

```
setosa_vs_virginica <- iris %>%
    filter(Species == "setosa" | Species == "virginica") %>%
    mutate(y = if_else(Species == "setosa", 1, -1))

y <- setosa_vs_virginica$y
X <- setosa_vs_virginica %>%
    select(-c("Species", "y")) %>%
    mutate(Sepal.Length = (Sepal.Length - mean(Sepal.Length)) / sd(Sepal.Length),
           Sepal.Width = (Sepal.Width - mean(Sepal.Width)) / sd(Sepal.Width),
           Petal.Length = (Petal.Length - mean(Petal.Length)) / sd(Petal.Length),
           Petal.Width = (Petal.Width - mean(Petal.Width)) / sd(Petal.Width)) %>%
    as.matrix()
```

```
train_frac <- 0.7
test_frac <- 0.3

data_split <- test_validation_train.split(X, y, train_frac, test_frac)
```

```
model_svm_1 <- SVM.fit(X = data_split$train_X,
                       y = data_split$train_y,
                       kernel = kernel.RBF,
                       C = 100,
                       s = 1)
```

```
measure_fitness(model_svm_1, data_split$test_X, data_split$test_y)
```

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

Versicolor vs Virginica

```
versicolor_vs_virginica <- iris %>%
    filter(Species == "versicolor" | Species == "virginica") %>%
    mutate(y = if_else(Species == "versicolor", 1, -1))

y <- versicolor_vs_virginica$y
X <- versicolor_vs_virginica %>%
    select(-c("Species", "y")) %>%
    mutate(Sepal.Length = (Sepal.Length - mean(Sepal.Length)) / sd(Sepal.Length),
           Sepal.Width = (Sepal.Width - mean(Sepal.Width)) / sd(Sepal.Width),
           Petal.Length = (Petal.Length - mean(Petal.Length)) / sd(Petal.Length),
           Petal.Width = (Petal.Width - mean(Petal.Width)) / sd(Petal.Width)) %>%
    as.matrix()
```

```
train_frac <- 0.7
test_frac <- 0.3

data_split <- test_validation_train.split(X, y, train_frac, test_frac)
```

```
model_svm_2 <- SVM.fit(X = data_split$train_X,
                       y = data_split$train_y,
                       kernel = kernel.RBF,
                       C = 100,
                       s = 1)
```

```
measure_fitness(model_svm_2, data_split$test_X, data_split$test_y)
```

```
## [1] 0.9333333
```

Versicolor vs Setosa

```
versicolor_vs_setosa <- iris %>%
    filter(Species == "versicolor" | Species == "setosa") %>%
    mutate(y = if_else(Species == "versicolor", 1, -1))

y <- versicolor_vs_setosa$y
X <- versicolor_vs_setosa %>%
    select(-c("Species", "y")) %>%
    mutate(Sepal.Length = (Sepal.Length - mean(Sepal.Length)) / sd(Sepal.Length),
           Sepal.Width = (Sepal.Width - mean(Sepal.Width)) / sd(Sepal.Width),
           Petal.Length = (Petal.Length - mean(Petal.Length)) / sd(Petal.Length),
           Petal.Width = (Petal.Width - mean(Petal.Width)) / sd(Petal.Width)) %>%
    as.matrix()
```

```
train_frac <- 0.7
test_frac <- 0.3

data_split <- test_validation_train.split(X, y, train_frac, test_frac)
```

```
model_svm_3 <- SVM.fit(X = data_split$train_X,
                       y = data_split$train_y,
                       kernel = kernel.linear)
```

```
measure_fitness(model_svm_3, data_split$test_X, data_split$test_y)
```

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

Jak widać na powyższych przykładach algorytm osiąga bardzo dobrą dokładność.