Introducción a las máquinas de vectores de soporte (SVMs). Teoría, implementación y complejidad.

Eduardo Rivero Rodríguez
Universidad Complutense de Madrid, España

21 de marzo de 2019

1. Introducción

En este documento se explicará el concepto de Máquina de Vectores de Soporte (SVM por sus siglas en inglés), así como se describirá una posible implementación para una SVM de kernel lineal. Además, se pondrán de manifiesto algunas de las técnicas utilizadas para optimizar su rendimiento.

1.1. ¿Qué es una SVM?

Las SVM son una serie de algoritmos de aprendizaje automático supervisado utilizados principalmente para resolver problemas de clasificación, regresión e incluso otros como la selección de variables en una muestra dada. A pesar de sus múltiples usos, en este documento nos centraremos en las SVMs lineales y en su aplicación a la hora de resolver el problema de clasificación, su comportamiento en el mismo, sus ventajas y limitaciones y las técnicas algorítmicas que se utilizan para su optimización.

1.2. ¿Cómo funciona una SVM lineal?

En el caso de la clasificación (binaria), el algoritmo de las SVMs lineales se basa en, dada una serie de datos n-dimensionales, hallar un hiperplano (o

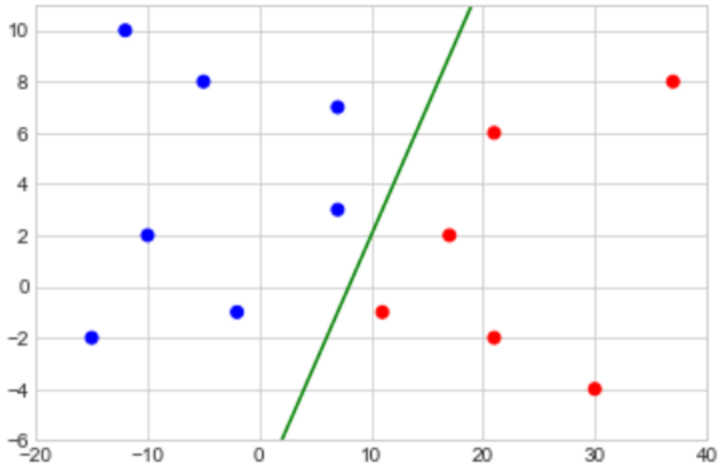

Figura 1: El hiperplano óptimo (verde) y las dos clases de puntos

un conjunto de hiperplanos) que separen las distintas clases de datos maximizando una cierta distancia (margen) a los puntos de las mismas.

1.2.1. Hiperplanos, márgenes y el problema de optimización

Consideremos la ecuación de un hiperplano en $\mathbb{R}^n$

$$w_1x_1 + w_2x_2 + ... + w_nx_n + b = 0$$

Con el producto escalar usual, obtenemos que esta ecuación es equivalente a

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

Donde $\mathbf{w} = (w_1, w_2, ..., w_n)$ y $\mathbf{x} = (x_1, x_2, ..., x_n)$. Con esta notación, llamaremos a $\mathbf{w}$ *vector de costes* de la SVM.

Dado un hiperplano, el margen a considerar es lo que llamamos *margen geométrico* que, entre un punto $\mathbf{x}$ y un hiperplano determinado por la ecuación $\mathbf{w} \cdot \mathbf{x} + b = 0$ viene dado por

$$\gamma = y\left(\frac{\mathbf{w}}{||\mathbf{w}||} \cdot \mathbf{x} + \frac{b}{||\mathbf{w}||}\right)$$

Para hallar el margen entre un conjunto finito $\mathcal{D} = \{\mathbf{x_1}, ..., \mathbf{x_m}\}$ y un hiperplano es entonces

$$M = \min_{i=1...m} \gamma_i$$

Es decir, la distancia mínima de cualquiera de los puntos del conjunto al hiperplano dado. Cabe mencionar que el *margen geométrico* es invariante respecto al módulo de $\mathbf{w}$, es decir, respecto a la escala del vector $\mathbf{w}$. Esta definición viene motivada por el uso "intuitivo" del *margen funcional*, que definimos como

$$f = y \times \beta = y(\mathbf{w} \cdot \mathbf{x} + b)$$

donde la distancia de la nube de puntos al plano sería

$$F = \min_{i=1...m} f_i$$

y cuyo gran problema es la no invarianza por la escala de $\mathbf{w}$.

Tras estas definiciones, tenemos que el problema a resolver para crear nuestra SVM es hallar el hiperplano con máximo *margen geométrico* entre todos los hiperplanos posibles.

$$\max_{\mathbf{w},b} \quad M$$
$$sujeto\ a \quad f_i \geq 1,\ i = 1...m$$

Puesto que podemos reescalar $\mathbf{w}$ y b a nuestro antojo sin que el valor de M varíe, lo haremos de forma que $F = 1$. Y entonces, como $M = \frac{F}{||\mathbf{w}||}$ podemos transformar el problema anterior en

$$\min_{\mathbf{w},b} \quad ||\mathbf{w}||$$
$$sujeto\ a \quad f_i = y_i(\mathbf{w} \cdot \mathbf{x_i}) + b_i \geq 1,\ i = 1...m$$

Que es un problema de optimización cuadrática convexa más simple que el inicial. Por conveniencia, utilizaremos el siguiente problema (equivalente al anterior) para continuar el desarrollo teórico

$$\min_{\mathbf{w},b} \quad \frac{1}{2}||\mathbf{w}||$$
$$sujeto\ a \quad f_i = y_i(\mathbf{w} \cdot \mathbf{x_i}) + b_i \geq 1,\ i = 1...m$$

1.2.2. Multiplicadores de Lagrange y el problema dual de Wolfe

El método que utilizaremos para resolver el problema de optimización es el de los multiplicadores de Lagrange. Para esto, introducimos el lagrangiano

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = f(\mathbf{w}) - \sum_{i=1}^{m} \alpha_i g_i(\mathbf{w}, b)$$

Siendo $\boldsymbol{\alpha}$ el vector de multiplicadores de Lagrange asociados a las funciones de las restricciones y $f(\mathbf{w}) = \frac{1}{2}||\mathbf{w}||$. Una forma de solucionar el problema es resolver analíticamente $\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = 0$. Pero esto es complejo y dificulta aumentar el número de datos de entrada. Para simplificar, tomamos el siguiente problema de optimización que utiliza el Lagrangiano

$$\min_{\mathbf{w},b} \max_{\alpha} \mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha})$$
$$sujeto\ a \quad \alpha_i \geq 0,\ i = 1...m$$

Consideraremos este como el problema lagrangiano primal. Si consideramos una solución del mismo, debe verificar $\nabla_{\mathbf{w}} \mathcal{L} = \mathbf{0}$ y además $\frac{\partial \mathcal{L}}{\partial b} = 0$ (pues dicho punto será un extremo de la función $\mathcal{L}_{\boldsymbol{\alpha}}(\mathbf{w}, b) = \mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha})$ donde $\boldsymbol{\alpha}$ es el vector $\boldsymbol{\alpha}$ que maximiza el Lagrangiano).

Entonces, aplicando estas restricciones sobre $\mathcal{L}$ en el problema anterior, obtenemos

$$\max_{\boldsymbol{\alpha}} W(\boldsymbol{\alpha}, b) = \max_{\boldsymbol{\alpha}} \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x_i} \cdot \mathbf{x_j}$$
$$sujeto\ a \quad \alpha_i \geq 0,\ i = 1...m$$
$$\sum_{i=1}^{m} \alpha_i y_i = 0$$

Un problema cuya función objetivo depende únicamente de los multiplicadores y que se denomina *problema dual de Wolfe*.

.

La descripción matemática de las SVMs viene adaptada de las referencias [4, 5, 2]

2. Implementando un clasificador binario mediante una SVM lineal

Consideremos el siguiente problema:

> Sea un conjunto $A = \{a_1, a_2, ..., a_m\} \subset \mathbb{R}^n$. Sea también $\mathbf{y} \in \{-1, 1\}^m$ un vector indicando la categoría de cada punto x_i. Entonces, dado un nuevo punto $x \in \mathbb{R}^n$, queremos averiguar la categoría a la que pertenece.

Para resolver el problema dual de Wolfe asociado al enunciado anterior, utilizaremos Python y un módulo de programación cuadrática llamado CVXOPT.

```
import numpy as np
from cvxopt import matrix
import cvxopt.solvers as solvers
```

Código 1: Imports de dependencias

2.1. Implementación de la SVM

A continuación crearemos una SVM con un modelo similar al utilizado en el paquete *sklearn* de Python. *Sci-kit learn* o *sklearn* es un módulo de aprendizaje automático de software libre para el lenguaje de programación Python. En particular, tiene una implementación de un clasificador mediante SVMs con una interfaz muy similar a la que implementaremos a continuación

1. *fit*: Recibe los datos de entrenamiento, sus categorías correspondientes y entrena el modelo.

2. *predict*: Devuelve las predicciones en base a los datos de entrada.

El código de esta sección está basado en el de una página web publicada en GitHub [1] así como en el de uno de los libros utilizados para este trabajo [4], libremente adaptado para ilustrar mejor el funcionamiento de las SVMs.

3. *test*: En base a los datos de prueba y sus categorías, calcula la precisión del modelo.

Además, se utilizarán las funciones auxiliares *compute_w* y *compute_b* que calcularán el vector de pesos $\mathbf{w}$ y el término independiente b respectivamente. Por conveniencia y legibilidad, el código de la clase vendrá dividido en las distintas partes de esta sección.

2.1.1. La clase *BinaryLinearSVM*

Declaramos la clase, inicializando los coeficientes y el término independiente (el modelo inicial) a valores arbitrarios. Obsérvese que, en Python, no es imperativo realizar esta inicialización.

```
"""Clase representando el funcionamiento de una SVM lineal"""
class BinaryLinearSVM:

    def __init__(self):
        self.weights = []
        self.b = 0
        self.accuracy = -1 #Desconocida
```

Código 2: Declaración de clase e inicialización

2.1.2. Funciones auxiliares

Para comenzar, daremos la función que calcula los coeficientes del hiperplano a partir de los multiplicadores de Lagrange de los puntos de la muestra, siguiendo la fórmula

$$\nabla_{\mathbf{w}}\mathcal{L} = 0 \iff \mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i \mathbf{x_i}$$

```
"""Computamos los pesos del hiperplano"""
def compute_w(multipliers,X,y):
    return sum(multipliers[i] * y[i] * X[i] for i in
     ↪  range(len(y)))
```

Código 3: Función para calcular los coeficientes

Una vez obtenemos los coeficientes del hiperplano, el cálculo del término independiente se realiza como

$$b = \frac{1}{n}\sum_{i=0}^{n} y_i - \mathbf{w} \cdot \mathbf{x_i}$$

pues tomando la media obtenemos una solución numéricamente más estable que simplemente realizando el cálculo con una componente arbitraria.

```
"""Computamos el término independiente"""
def compute_b(w,X,y):
    return sum(y[i] - np.dot(w,X[i]) for i in
     ↪  range(len(X)))/len(X)
```

Código 4: Cálculo del término independiente

2.1.3. Realizando predicciones

Para realizar una predicción, basta detectar la posición relativa del punto dado respecto del hiperplano de nuestro modelo. Con nuestra codificación, tendríamos lo siguiente

$$f(x) = \begin{cases} 1 & si\ \mathbf{w} \cdot \mathbf{x} + b > 0 \\ -1 & en\ otro\ caso \end{cases}$$

```
    def predict(self, x):
        #Computamos el resultado de la operación wx+b y
        ↪  determinamos la posición
        #relativa del punto respecto al hiperplano
        value = np.dot(self.weights,x)+self.b
        if value > 0:
            return 1
        else:
            return -1
```

Código 5: Función *predict*

2.1.4. Entrenando la SVM

Para entrenar la SVM es necesario hallar la solución del problema dual de Wolfe, para lo que utilizaremos el paquete CVXOPT mencionado anteriormente. La función que resuelve problemas de programación cuadrática solo admite problemas de optimización formulados de la forma [3]

$$\text{mín}\, \frac{1}{2}\mathbf{x}^T P\mathbf{x} + q^T\mathbf{x}$$
$$\textit{sujeto a}\ G\mathbf{x} \preceq \mathbf{h},\ A\mathbf{x} = b$$

donde $\preceq$ es una desigualdad vectorial por componentes. Por tanto, debemos transformar el problema dual de Wolfe con nuestra formulación anterior a la que acabamos de mencionar. Observando que $\text{máx}\, f = -\,\text{mín}\, -f$ tenemos que, sabiendo que

$$\mathbf{x} = \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix}$$

y fijándonos en las condiciones de ambos problemas, se tienen las siguientes equivalencias

$$G\mathbf{x} \preceq \mathbf{h} \iff \forall\ i,\ \alpha_i \geq 0$$

$$A\mathbf{x} = b \iff \sum_{i=1}^{m} \alpha_i y_i = 0$$

$$\frac{1}{2}\mathbf{x}^T P \mathbf{x} = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x_i} \cdot \mathbf{x_j}$$

$$q^T \mathbf{x} = -\sum_{i=1}^{m} \alpha_i$$

de donde podemos deducir que las matrices P, q, G, h y A son

$$P = (p_{ij})_{i,j=0}^{m} = (y_i y_j \mathbf{x_i} \cdot \mathbf{x_j})_{i,j=0}^{m}$$

$$q = \begin{pmatrix} -1 \\ \vdots \\ -1 \end{pmatrix}$$

$$G = \begin{pmatrix} -1 & 0 & \cdots & 0 \\ 0 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & -1 \end{pmatrix}$$

$$h = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$

$$A = \begin{pmatrix} y_1 & \cdots & y_m \end{pmatrix}$$

La función *fit* ha sido troceada por legibilidad

```
    def fit(self, train_data, train_labels):
        m = train_data.shape[0]
        K = np.array([np.dot(train_data[i], train_data[j])
                      for i in range(m)
                      for j in range(m)]).reshape((m,m))

        aux = np.outer(train_labels,train_labels)*K
        P = matrix(aux, aux.shape, 'd')
        q = matrix(-1 * np.ones(m))
        #Identificamos los vectores de soporte (aquellos con
        ↪  multiplicador positivo)
        positive_multiplier = multipliers>1e-7
        sv_multipliers = multipliers[positive_multiplier]
```

Código 6: Cálculo de la forma cuadrática P y de q

```
        A = matrix(train_labels, (1,m), 'd')
        b = matrix(0.0)
        G = matrix(np.diag(-1*np.ones(m)))
        h = matrix(np.zeros(m))
```

Código 7: Cálculo de las restricciones de igualdad y desigualdad

```
        # Desactiva la salida de texto por defecto
        solvers.options['show_progress'] = False
        # Resuelve el problema de programación cuadrática con
        ↪  los datos deseados
        solution = solvers.qp(P,q,G,h,A,b)
        # Guarda las soluciones en un vector unidimensional
        multipliers = np.ravel(solution['x'])
```

Código 8: Resolución del problema y cálculo de los multiplicadores de Lagrange

```
        support_vectors = train_data[positive_multiplier]
        support_vector_labels =
        ↪  train_labels[positive_multiplier]
        self.weights = compute_w(multipliers, train_data,
        ↪  train_labels)
        self.b = compute_b(self.weights,support_vectors,
        ↪  support_vector_labels)
```

Código 9: Identificación de los vectores de soporte y cálculo de $\mathbf{w}$ y b

2.1.5. Probando la eficacia del modelo

Para saber cómo de eficiente es el modelo, consideraremos la métrica *precisión*, que definimos como, dado un conjunto de datos de tamaño n, la clase a la que está asociada cada punto y las predicciones del modelo

$$precision = \frac{numero\ de\ aciertos}{n}$$

siendo una predicción un acierto si coincide con la clase real.

```
    def test(self, test_data, test_labels):
        # Probamos las predicciones y devolvemos la precisión
        results = [self.predict(test_data[i]) for i in
        ↪  range(len(test_data))]
        count = sum(1 for i in range(len(test_data)) if
        ↪  test_labels[i]==results[i])
        self.accuracy = 100*(count/len(test_data))
```

Código 10: Función *test* y precisión de la SVM

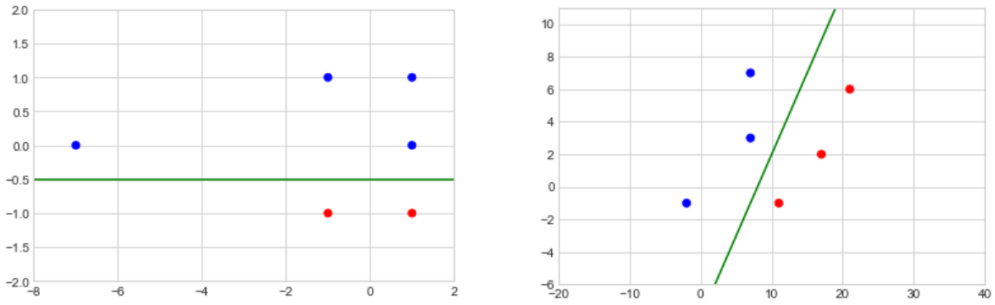

Figura 2: Algunos ejemplos de cálculo del hiperplano óptimo

Se pueden ver la clase completa y ejemplos de ejecución en GitHub

3. Comportamiento de las SVMs

Por otro lado, la complejidad en tiempo de la predicción está en $\mathcal{O}(d)$ siendo d la dimensión del espacio de características (*feature space*).

Inicialmente, parecería que la complejidad de entrenamiento en tiempo de la SVM es exponencial por ser esta equivalente a la resolución de un problema de optimización cuadrática, siendo estos generalmente NP-duros [7]. Sin embargo, la observación de que el problema

$$\min_{\mathbf{w},b} \quad \frac{1}{2}||\mathbf{w}||$$
$$sujeto\ a \quad f_i = y_i(\mathbf{w} \cdot x_i) + b_i \geq 1,\ i = 1...m$$

es un problema de optimización convexa es clave para ver que la complejidad de entrenamiento es en realidad polinómica [11]. El valor concreto de esta complejidad depende del algoritmo y varía en la literatura [11, 12, 2, 10], aunque típicamente tiene complejidad computacional por iteración en torno a $\mathcal{O}(n^2)$ y número de iteraciones en torno a $\mathcal{O}(n)$. Esto deja una complejidad total en $\mathcal{O}(n^3)$.

En cuanto a la complejidad en memoria, típicamente suele estar en $\mathcal{O}(n^2)$.

3.1. Técnicas algorítmicas para la optimización del funcionamiento de las SVMs

Puesto que la complejidad anterior es alta (aún siendo mucho mejor que la complejidad exponencial inicial) resulta difícil abordar problemas reales, dado el inmenso tamaño de los conjuntos de datos asociados a los mismos. Por ello, se han creado diversas técnicas que se utilizan con el objetivo de acelerar el entrenamiento de las SVMs y, en especial, de permitir que trabajen con grupos de datos de gran tamaño.

3.1.1. Optimización Mínima Secuencial (SMO)

Entre los algoritmos más importantes para la optimización del entrenamiento de las SVMs está la Optimización Mínima Secuencial, propuesto por

Nótese que la complejidad en tiempo anterior está calculada para el caso peor. Lo cierto es que, en general, puesto que los datos suelen ser dispersos, el rendimiento es mucho mejor.

John C. Platt [8]. Este algoritmo surge para evitar altos costes en memoria y mantener una complejidad computacional razonable en el entrenamiento. Platt propone comenzar reformulando el problema de optimización usando el Lagrangiano con el que hemos trabajado anteriormente.

$$\min_{\boldsymbol{\alpha}} \Psi(\boldsymbol{\alpha}) = \min_{\boldsymbol{\alpha}} \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \mathbf{x_i} \cdot \mathbf{x_j} - \sum_{i=1}^{m} \alpha_i$$

$$Sujeto\ a\ 0 \preceq \boldsymbol{\alpha} \preceq C\ \forall i,\ \sum_{i=0}^{m} y_i \alpha_i = 0$$

donde C es un parámetro de "flexibilidad" que sirve para ajustar la rigidez del modelo generado. Las condiciones necesarias y suficientes para la optimalidad de la solución son las llamadas condiciones de Karush-Kuhn-Tucker (KKT) que, en este problema en particular, son

$$\alpha_i = 0 \Longleftrightarrow y_i u_i \geq 1$$
$$0 < \alpha_i < C \Longleftrightarrow y_i u_i = 1$$
$$\alpha_i = C \Longleftrightarrow y_i u_i \leq 1$$

donde u_i es la salida de la SVM para la muestra i-ésima del conjunto de datos de entrenamiento.

Gracias al teorema probado por Osuna et al. [6] podemos partir el problema de programación cuadrática hasta reducirlo a un conjunto de problemas con solamente dos multiplicadores. Estos últimos problemas se pueden resolver analíticamente y, entonces, iterando el proceso hasta que se verifiquen las condiciones de KKT, obtenemos la solución óptima. A continuación la descripción, a grandes rasgos, del algoritmo.

Algoritmo 1: Optimización Mínima Secuencial (simplificada)

1. Hallar α_1 multiplicador de Lagrange que viole las condiciones de KKT
2. Hallar α_2 otro multiplicador
3. Resolver el problema con el par (α_1, α_2)
4. Iterar los 3 pasos anteriores hasta obtener la convergencia

Esta forma de entrenar la SVM tiene 2 grandes ventajas: primero, resuelve el problema de forma muy rápida (pues evita invocar a costosas librerías de

programación cuadrática) y, segundo, no necesita una matriz auxiliar, por lo que la complejidad en espacio es mucho menor.

El núcleo del algoritmo gira en torno a tres partes

1. Un método analítico para resolver el problema con dos multiplicadores.
2. Una heurística para escoger qué multiplicadores optimizar.
3. Un método para computar el término independiente b y actualizar el vector de pesos $\mathbf{w}$.

Analicemos a continuación los 3 principios del algoritmo. Sean α_1 y α_2 dos multiplicadores de Lagrange para el problema dual de Wolfe. Observemos que la restricción lineal que obliga a que la suma de los multiplicadores α_i por las etiquetas y_i se anule conlleva que $y_1\alpha_1^{nuevo}+y_2\alpha_2^{nuevo} = \gamma = \alpha_1^{antiguo}+s\alpha_2^{antiguo}$ donde $s = y_1y_2$ para mantener la suma total.

Las restricciones de este problema nos garantizan [8] que la solución al problema con dos multiplicadores (α_1, α_2) estará en el cuadrado $[0, C]\times[0, C]$. Además, la restricción lineal que hemos mencionado en el párrafo anterior obliga a que los multiplicadores candidatos estén en la diagonal de extremos

$$L = \text{máx}(0, \alpha_2 - \alpha_1), \quad H = \text{mín}(C, C + \alpha_2 - \alpha_1) \quad \text{si } y_1 \neq y_2$$

$$L = \text{máx}(0, \alpha_2 + \alpha_1 - C), \quad H = \text{mín}(C, \alpha_2 + \alpha_1) \quad \text{en caso contrario}$$

y sobre dicha diagonal, la diferencial de segundo orden de la función objetivo es

$$\eta = \mathbf{x_1} \cdot \mathbf{x_1} + \mathbf{x_2} \cdot \mathbf{x_2} - 2\mathbf{x_1} \cdot \mathbf{x_2}$$

Si llamamos al error en la muestra i $E_i = u_i - y_i$, siendo u_i la salida de la SVM sobre dicha muestra, entonces podemos calcular el valor de los nuevos multiplicadores.

$$\alpha_2^{no\ acotado} = \alpha_2 + \frac{y_2(E_1 - E_2)}{\eta}$$

de donde, aplicando las restricciones, obtenemos

$$\alpha_2^{nuevo} = \begin{cases} H & \alpha^{no\ acotado} > H \\ \alpha^{no\ acotado} & L \leq \alpha^{no\ acotado} \leq H \\ L & \text{en caso contrario} \end{cases}$$

$$\alpha_1^{nuevo} = \alpha_1 + s(\alpha_2 - \alpha_2^{nuevo})$$

Es importante mencionar también que el algoritmo del SMO relaja las condiciones de KKT de forma que se cumplan con un margen de error ε pequeño. Esto permite más flexibilidad y una convergencia ligeramente más rápida sin afectar significativamente a la precisión de las predicciones.

A continuación se describirán las 2 heurísticas utilizadas para la elección de los multiplicadores.

1. **Primer multiplicador:** Se iteran todos los multiplicadores hasta que cumplan las condiciones de KKT relajadas. A continuación, se itera sobre todos los multiplicadores que no son 0 ni C. Este proceso se repite hasta tener la convergencia del algoritmo.

2. **Segundo multiplicador:** El segundo multiplicador se escoge maximizando la variación en el paso de optimización, aproximando dicha variación utilizando $|E_1 - E_2|$ para evitar el coste de calcular los productos escalares.

Nótese que la SVM llevará una caché de errores en la que almacenará los errores precomputados para cada muestra.

Para computar el término independiente, tenemos que el algoritmo tomará $b^{nuevo} = \frac{b_1+b_2}{2}$ siendo

$$b_1 = E_1 + y_1(\alpha_1^{nuevo} - \alpha_1)\mathbf{x_1} \cdot \mathbf{x_1} + y_2(\alpha_2^{nuevo} - \alpha_2)\mathbf{x_1} \cdot \mathbf{x_2} + b$$

$$b_2 = E_2 + y_1(\alpha_1^{nuevo} - \alpha_1)\mathbf{x_1} \cdot \mathbf{x_2} + y_2(\alpha_2^{nuevo} - \alpha_2)\mathbf{x_2} \cdot \mathbf{x_2} + b$$

Por último, la actualización del vector de pesos $\mathbf{w}$ se hace como sigue

$$\mathbf{w}^{nuevo} = \mathbf{w} + y_1(\alpha_1^{nuevo} - \alpha_1)\mathbf{x_1} + y_2(\alpha_2^{nuevo} - \alpha_2)\mathbf{x_2}$$

Para el funcionamiento de los algoritmos descritos a continuación, es imprescindible que se tenga acceso a una lista de elementos de la muestra $muestra[1...n_{muestras}]$, una lista de multiplicadores $mult[1...n_{muestras}]$, la lista de tags $tag[1...n_{muestras}]$, un ε prefijado *eps* para la relajación de las condiciones y un valor de tolerancia *tol* que indica el margen.

Algoritmo 2: Optimización Mínima Secuencial (detallada)

```
1  cambiado ← false
2  examinarTodos ← true
3  mientras cambiado ∨ examinarTodos hacer
4  |  si examinarTodo entonces
5  |  |  para i ← 0 hasta n_muestras hacer
6  |  |  |  cambiado ← cambiado ∨ examinar(i)
7  |  |  fin para
8  |  en otro caso
9  |  |  para i ← 0 hasta n_muestras hacer
10 |  |  |  si mult[i] ≠ 0 ∨ mult[i] ≠ C entonces
11 |  |  |  |  cambiado ← cambiado ∨ examinar(i)
12 |  |  |  fin si
13 |  |  fin para
14 |  fin si
15 |  si examinarTodos entonces
16 |  |  examinarTodos ← 0
17 |  en otro caso
18 |  |  cambiado
19 |  fin si
20 |  examinarTodos ← 1
21 fin mientras
```

A continuación se describe el algoritmo para examinar una muestra, que recibe el índice $i1$ y devuelve *true* si se ha producido algún cambio en los multiplicadores y $false$ en caso contrario.

Algoritmo 3: Examinar una muestra

```
1  y2 ← tag[i2]
2  alpha2 ← mult[i2]
3  E2 ← predict(muestra[i2]) − y2
4  r2 ← E2 · y2
5  si (r2 < −tol ∧ alpha2 < C) ∨ (r2 > tol ∧ alpha2 > 0) entonces
6  |  si n_{mul≠0} > 1 ∧ n_{mul≠C} > 1 entonces
7  |  |  i1 ← resultado de la heurística
8  |  |  si paso(i1, i2) entonces
9  |  |  |  devolver true
10 |  |  fin si
11 |  fin si
12 |  para i ← 0 hasta n_{muestras} hacer
13 |  |  si mult[i] ≠ 0 ∧ mult[i] ≠ C entonces
14 |  |  |  i1 ← i
15 |  |  |  si paso(i1, i2) entonces
16 |  |  |  |  devolver true
17 |  |  |  fin si
18 |  |  fin si
19 |  fin para
20 |  para i ← 0 hasta n_{muestras} hacer
21 |  |  i1 ← i
22 |  |  si paso(i1, i2) entonces
23 |  |  |  devolver true
24 |  |  fin si
25 |  fin para
26 fin si
27 devolver false
```

Por último, se describe el algoritmo de resolución del problema con dos multiplicadores *paso*, que recibe los índices $i1$ e $i2$ de los multiplicadores en cuestión y devuelve *true* si se han producido cambios en los mutliplicadores y $false$ en caso contrario.

Algoritmo 4: Resolución del problema con dos multiplicadores (paso) - Parte 1

```
si i1 = i2 entonces
    devolver false
fin si
alpha1 ← mult[i1]
alpha2 ← mult[i2]
y1 ← tag[i1]
y2 ← tag[i2]
E1 ← predict(muestra[i1]) − y1
s ← y1 · y2
Computar L y H
si L = H entonces
    devolver false
fin si
eta ← ⟨muestra[i1], muestra[i1]⟩ + ⟨muestra[i2], muestra[i2]⟩ −
  2⟨muestra[i1], muestra[i2]⟩
si eta > 0 entonces
    a2 ← alpha2 + y2 · (E1 − E2)/eta
    si a2 < L entonces
        a2 ← L
    en otro caso
        a2 ← H
    fin si
en otro caso
    Lobj ← Ψ(alpha1, L)
    Hobj ← Ψ(alpha1, H)
    si Lobj < Hobj − eps entonces
        a2 ← L
    en otro caso
        si Lobj > Hobj + eps entonces
            a2 ← H
        en otro caso
            a2 ← alpha2
        fin si
    fin si
fin si
```

Algoritmo 5: Resolución del problema con dos multiplicadores (paso) - Parte 2

si $|a2 - alpha2| < eps \cdot (a2 + alpha2 + eps)$ **entonces**
  devolver *false*
fin si
$a1 = alpha1 + s \cdot (alpha2 - a2)$
Actualizar b para reflejar el cambio
Actualizar w para reflejar el cambio
Actualizar la caché de errores
$mult[i1] \leftarrow a1$
$mult[i2] \leftarrow a2$

3.1.2. Otras técnicas

Además del SMO, existen otras técnicas que se utilizan para acelerar el proceso de entrenamiento de la SVM. Tenemos por ejemplo el *chunking* [9], que busca subdividir el conjunto de datos inicial en distintas clases (*chunks*) que, a continuación, se usan independientemente para entrenar la SVM. A partir de aquí, se pueden aplicar muchas formas distintas de obtener un modelo final: tomar el chunk con mayor precisión, escoger un chunk aleatorio, tomar las medias de los pesos y los términos independientes...

Algoritmo 6: Ejemplo de Algoritmo de Chunking estático

mientras *queden muestras sin procesar* **hacer**
  Tomar aleatoriamente hasta N muestras no procesadas del conjunto de entrenamiento
  Entrenar el modelo de la SVM con dichas muestras
  Almacenar los pesos $\mathbf{w}$ y el término independiente b del modelo entrenado
fin mientras
Tomar como $\mathbf{w}$ y b del modelo definitivo la media aritmética de los almacenados

Muchos otros métodos para mejorar la eficiencia y la precisión de las SVMs dedican su esfuerzo a tratar el conjunto de datos para minimizarlo en la medida de lo posible, de modo que el conjunto final sea representativo de las relaciones entre los datos del conjunto inicial (*feature selection*, *outlier removal*...).

Algoritmo 7: Ejemplo de Algoritmo de Feature Selection

1 Cuantificar la influencia de los atributos sobre la clase a la que pertenece una muestra mediante el test estadístico χ^2
2 Descartar todos los atributos cuya influencia no esté en el top 25 % de las calculadas

Algoritmo 8: Ejemplo de Algoritmo de Outlier Removal

1 Calcular los cuartiles de los atributos numéricos del conjunto de datos
2 Descartar aquellas muestras cuyos datos no estén entre el primer y el tercer cuartil

Los cuartiles de una muestra se definen como los 3 valores que dividen a un conjunto de datos ordenado en 4 conjuntos de igual probabilidad.

Referencias

[1] *An SVM in just a few Lines of Python Code*. https://maviccprp.github.io/a-support-vector-machine-in-just-a-few-lines-of-python-code/. Accessed: 24-10-2018.

[2] Stephen Boyd y Lieven Vandenberghe. *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004. ISBN: 0521833787.

[3] *CVXOPT User Guide*. http://cvxopt.org/userguide/.

[4] Alexandre Kowalczyk. *Support Vector Machines Succintly*. https://www.syncfusion.com/ebooks/support_vector_machines_succinctly. Oct. de 2017.

[5] Yuh-jye Lee, Yi-ren Yeh y Hsing-kuo Pao. *An Introduction to Support Vector Machines*. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.488.5833&rep=rep1&type=pdf.

[6] E Osuna, Robert Freund y Federico Girosi. «An improved training algorithm of Support Vector Machines». En: oct. de 1997, págs. 276-285. ISBN: 0-7803-4256-9. DOI: 10.1109/NNSP.1997.622408.

[7] P.M. Pardalos y S.A. Vavasis. «Quadratic Programming with One Negative Eigenvalue is NP-Hard». En: *J Glob Optim 1: 15.* (1991). URL: https://doi.org/10.1007/BF00120662.

[8] John Platt. «Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines». En: (abr. de 1998), pág. 21. URL: https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/.

[9] Yuji Matsumoto Taku Kudo. «Chunking with Support Vector Machines.» En: *Proceedings of NAACL, 200, 192-199* (2001).

[10] Ivor W. Tsang, James T. Kwok y Pak-Ming Cheung. «Core Vector Machines: Fast SVM Training on Very Large Data Sets». En: *J. Mach. Learn. Res.* 6 (dic. de 2005), págs. 363-392. ISSN: 1532-4435.

[11] Paul Tseng, Institute of Technology. Laboratory for Information y Massachusetts Decision Systems. «A simple polynomial-time algorithm for convex quadratic programming». En: *LIDS Technical Reports* (ene. de 1988). URL: https://www.researchgate.net/publication/37594447_A_simple_polynomial-time_algorithm_for_convex_quadratic_programming.

[12] Yinyu Ye y Edison Tse. «An extension of Karmarkar's projective algorithm for convex quadratic programming». En: *Mathematical Programming* 44.1 (mayo de 1989), págs. 157-179. ISSN: 1436-4646. DOI: 10.1007/BF01587086. URL: https://doi.org/10.1007/BF01587086.