SVM classifier in AMPL

Alex Ferrando de las Morenas — Elías Abad Rocamora

5 de junio de 2020

Índice

1. Introducción **3**

2. Implemetación **5**
2.1. AMPL 5
2.1.1. Problema primal 5
2.1.2. Problema Dual 5
2.2. Python 6

3. Resultados **7**
3.1. gensvmdat 7
3.2. Swiss roll 9
3.3. Base de datos Skin 9

4. Conclusiones **11**

1. Introducción

En este proyecto se pretende implementar una *Support Vector Machine* para profundizar en el estudio y formulación de los problemas de optimización relacionados con este algoritmo de aprendizaje supervisado. La *SVM*, dado un conjunto de datos (representados como puntos en el espacio) para el entrenamiento, cada uno de los cuales está etiquetado con la clase de pertenencia entre dos posibles, construye una separación lineal del espacio que intenta maximizar la distancia entre las dos categorías. Esta separación constituye un modelo para la predicción categórica de nuevos datos diferentes a los de entrenamiento.

Para afrontar el estudio planteado, hemos creado un programa en Python que implementa la creación o lectura de una base de datos con sus respectivas variables explicativas y una variable respuesta binaria y encuentra la división espacial lineal que maximiza la separación entre las categorías. El problema de encontrar el hiperplano en el espacio que consigue este objetivo se plantea como un problema de optimización con restricciones, pues querremos maximizar la separación entre las dos categorías de puntos manteniendo que todos los puntos del conjunto de entrenamiento estén bien clasificados. Obviamente, si los grupos de puntos no son perfectamente separables linealmente, el problema será infactible. Para poder encontrar igualmente un plano de separación óptimo que, aunque no clasifique todos los datos de entrenamiento correctamente, nos permita encontrar una solución al problema de optimización planteado creamos variable auxiliares *slacks* que relajen las restricciones. Para que estas variables no tiendan a infinito las incluimos en la función a minimizar. Planteamos el problema de optimización:

$$\begin{aligned} \underset{(\omega,\gamma,s)}{min} \quad & \frac{1}{2}\omega^t\omega + \nu\sum_{i=1}^{m} s_i \\ s.a \quad & y_i(\omega^t x_i + \gamma) + s_i \geq 1 \\ & s_i \geq 0 \end{aligned}$$

Para encontrar la solución y aplicando que el problema es convexo, podemos resolver el problema dual. La convexidad del problema implica una condición de suficiencia para que la solución del primal (planteado arriba) y del dual sea la misma. Esto nos permite enfocar el problema desde otro punto de vista y encontrar soluciones más rápidamente. Si formulamos el problema vectorialmente obtenemos:

$$\begin{aligned} \underset{\lambda}{max} \quad & \lambda^t e - \frac{1}{2}\lambda^t Y K Y \lambda \\ s.a \quad & \lambda^t y = 0 \\ & 0 \leq \lambda \leq \nu e \end{aligned}$$

K es el kernel de nuestros datos. Si $K = AA^t$, el problema planteado será equivalente al primal pero si nos encontramos con un conjunto de datos linealmente no separable,

podemos realizar una transformación de los datos a otro espacio en el que sí sea posible separar linealmente los datos. En nuestro caso, utilizaremos un RBF o Kernel gaussiano. Dados dos puntos x e y, el Kernel correspondiente a estos dos puntos se define como:

$$K(x, y) = e^{-\frac{\|x-y\|^2}{2\sigma^2}}$$

En nuestro caso hemos utilizado $\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} Var(A[:, i])$, siendo $A[:, i]$ la i-ésima componente de los puntos de nuestra base de datos.

Para evaluar el rendimiento de nuestro clasificador, trabajaremos con 3 bases de datos diferentes:

1. La generada con el algoritmo proporcionado por el profesor, *gensvmdat*, que devuelve aleatoriamente puntos linealmente separables en 4 dimensiones clasificados en dos clases diferentes.

2. La generada con el método *make_swiss_roll()* de la librería sklearn de Python. Son puntos tridimensionales distribuidos de manera que forman una espiral que no es linealmente separable. Los puntos están divididos equitativamente entre dos clases y estas dos clases están definidas por la división de la espiral en dos partes iguales cuándo se desenrolla. Para conseguir la generación de puntos no perfecta, lo cual provocaría un error de clasificación 0, se le ha añadido un cierto ruido a los puntos.

3. Base de datos que contiene la cantidad de rojo, azul y verde (RGB) de diferentes imágenes y una variable respuesta que las clasifica según si son imágenes de piel o no[1]. Los datos tienen 3 dimensiones, donde cada dimensión es la intensidad de cada color en base RGB.

   Si hacemos un plot en 3 dimensiones, podemos observar que son datos linealmente separables:

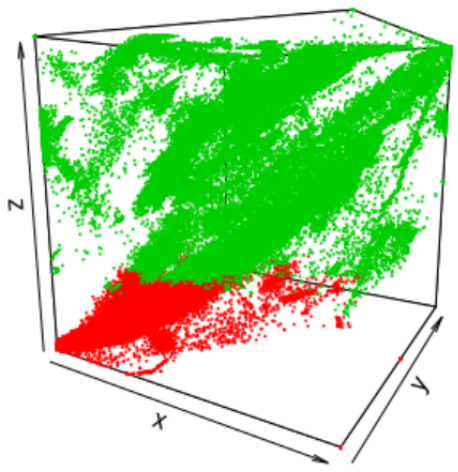

Figura 1: 3D plot de la base de datos

2. Implemetación

2.1. AMPL

La implementación de los códigos de este proyecto se ha llevado a cabo en dos lenguajes diferentes: AMPL y Python. AMPL se ha utilizado únicamente para definir los problemas a optimizar de forma abstracta y mediante Python, nos hemos encargado de gestionar la interfaz de lanzamiento del programa, la generación de datos y conversión a formato de entrada de AMPL, la obtención de los datos de salida de AMPL y el cálculo de resultados indirectos o estadísticas como la precisión de la clasificación.

A continuación se muestran los códigos en AMPL de las dos formulaciones del problema enunciadas en la introducción:

2.1.1. Problema primal

```
# Parameters
param n >= 1, integer;
param m >= 1, integer;
param nu >= 0;

param A {1..m, 1..n};
param y {i in 1..m};


# Variables
var gamma;
var w {1..n};
var s {1..m} >= 0;

#optimizacin
minimize SVM_primal: 1/2*sum{i in 1..n}(w[i]*w[i]) + nu*sum{j in 1..m}(s[j]);
subject to restric {j in 1..m}:
   y[j] * (sum{i in 1..n} (w[i]*A[j,i]) + gamma) + s[j] >= 1;
```

2.1.2. Problema Dual

```
# Parameters
param n >= 1, integer;
param m >= 1, integer;
param nu >= 0;

param K {1..m, 1..m};
param y {i in 1..m};

# Variables
var gamma;
var lambda {1..m} >= 0, <= nu;

#optimizacin
maximize SVM_dual: sum{i in 1..m}(lambda[i])
                -1/2*sum{j in 1..m, k in 1..m}(
                    lambda[j]*y[j]*lambda[k]*y[k]*K[j,k]
                );
subject to Dual_restric: sum{i in 1..m}(lambda[i]*y[i]) = 0;
```

Ambas formulaciones necesitan un formato de entrada específico para los parámetros de la optimización. Dado que los tipos de entrada son muy diferentes y están generados o almacenados de diferente forma, necesitábamos una manera automática y fiable de hacer esta conversión, así que hemos utilizado Python para crear un fichero auxiliar que sirva de entrada a AMPL. Este es transparente al usuario y solo se guarda el fichero que contiene los datos limpios, tal cual cómo se han generado. Si se quiere guardar también el archivo *.dat* que se le pasa a AMPL, en el README se explica cómo hacerlo.

Otro problema que nos surge es la obtención de la clasificación de los puntos a partir del resultado de la optimización. Sabemos que obtenido el hiperplano definido por ω, podemos asignar un punto a una clase o a otra con el siguiente criterio:

$$Class(x_i) = \begin{cases} \omega^t \phi(x_i) + \gamma > 0 \rightarrow x_i \in C_1 \\ \omega^t \phi(x_i) + \gamma \leq 0 \rightarrow x_i \in C_{-1} \end{cases}$$

Además, como con el problema dual no tenemos ni ω ni γ, necesitamos realizar aún más cálculos para obtener esta clasificación. Siendo *sv* el índice de un *support vector*, es decir, un punto que pertenece al plano de separación:

$$Class(x_i) = \begin{cases} \frac{1}{y_{sv}} + \sum_{j=1}^{m} \lambda_j y_j (K(x_i, x_j) - K(x_{sv}, x_j)) > 0 \rightarrow x_i \in C_1 \\ \frac{1}{y_{sv}} + \sum_{j=1}^{m} \lambda_j y_j (K(x_i, x_j) - K(x_{sv}, x_j)) \leq 0 \rightarrow x_i \in C_{-1} \end{cases}$$

Este nuevo criterio de clasificación se deriva a partir del anterior sustituyendo las variables primales por las duales. El nuevo punto x_i puede ser un punto con el que hemos entrenado el modelo o no, pero los puntos con índice j, deben ser obligatoriamente los puntos con los cuales se ha modelado y por lo tanto se han calculado las variables duales (λ).

2.2. Python

Como hemos visto, se han planteado ambas formulaciones en AMPL para encontrar con estos la solución óptima al problema de optimización. Sin embargo, para poder facilitar al máximo la ejecución de los códigos al usuario final, se ha construido un código en Python que, dados unos parámetros de entrada, hace las llamada pertinentes a AMPL y resuelve y trata los resultados obtenidos de forma transparente al usuario.

Para la resolución del problema, el código Python contempla los tres enfoques diferentes presentados en la introducción. Este permite escoger, a través de un parámetro de entrada, cómo se desea resolver el problema, ya sea con la formulación primal, la dual o aplicando un RBF (kernel gaussiano) en el cual obligatoriamente deberemos resolver usando el problema dual.

Una vez leídos los parámetros de entrada se escriben en un archivo *ampl_data.dat* para poder pasárselos a AMPL. Hemos tratado previamente las matrices guardas en el fichero para evitar errores de optimización en AMPL. Debido a errores de cálculo, AMPL computa valores propios de la matriz negativos muy pequeños, que en realidad deberían ser 0. Sumando una matriz con $\epsilon = 10^{-6}$ en toda la diagonal se solventa este problema asegurándonos así que la matriz sea siempre semidefinida positiva y *cplex* sea capaz de optimizar la función objetivo, ya que sólo es capaz de resolver problemas de optimización convexa.

Una vez obtenidos los resultados devueltos por AMPL estos se tratan para escribirlos en un archivo *.txt* de una forma comprensible y omitiendo aquellos cuyo conocimiento sea de escasa utilidad como por ejemplo el conjunto de *slacks*. Para todos los casos se obtiene la γ y para los casos donde se conoce ϕ, es decir, cuando no usamos el kernel gaussiano, se calculan y guardan los pesos ω. Para el problema primal, estos se obtienen directamente del resultado de AMPL mientras que para el caso de la formulación dual se deben obtener a partir del resultado de las λ que devuelve AMPL conociendo que la transformación ϕ es la identidad.

$$\omega = \sum_{i=1}^{m} \lambda_i y_i \phi(x_i)$$

Como hemos dicho, en los tres casos, el programa devuelve la γ, obtenida a partir del resultado AMPL con el primal y calculada a partir de algún punto de vector de soporte y_i que busca el programa Python.

$$\gamma = \frac{1}{y_i} - \phi(x_i)^t \omega$$

Este procedimiento se hace independientemente del método utilizado para la generación de datos. El usuario puede indicar a través de un parámetro de entrada cómo desea obtener los datos y cuantos datos desea en total. Con esta información el programa crea un conjunto de *training* con la *seed* indicada en la entrada y la función correspondiente ya sea generándolo con *gensvmdat*, *make_swiss_roll()* o leyendo de la base de datos *skin_noskin.txt*. Con este último método, la base de datos se reordena aleatoriamente para evitar obtener siempre los mismos resultados. Paralelamente, se genera un conjunto de *test* de la misma medida pero generado con una *seed* aleatoria por lo que cada vez que se ejecuta la aplicación, se genera un *dataset* diferente pero siempre asegurándose de no repetir los datos que tenemos en el *training*.

Finalmente, se computa la predicción sobre los conjuntos de *training* i *test* y se guardan los porcentajes de acierto en el fichero de texto de resultados.

3. Resultados

Para analizar la precisión de clasificación de la *SVM* utilizaremos, a parte del error de *training*, el porcentaje de acierto sobre el conjunto de test generado y así poder evaluar el rendimiento del algoritmo en datos diferentes a los de entrenamiento.

Dado que tenemos 3 bases de datos diferentes y varias formas de tratar los datos y encontrar el óptimo, haremos en análisis de rendimiento de los procedimientos usados para cada método de generación de datos por separado. En todas las ejecuciones generaremos 50 observaciones con la *seed* 34.

3.1. gensvmdat

Si generamos los datos con la función *gensvmdat* estaremos generando, como hemos explicado en la introducción, datos linealmente separables. Esta aplicación genera puntos en 4 dimensiones separables linealmente y clasifica aleatoriamente algunos mal. Se basa en la creación aleatoria de números entre el 0 y el 1 y si la suma de los cuatro valores

es inferior a dos los clasifica en un grupo y si es más grande o igual que dos en otro. Así pues, los resultados que esperaríamos tener es un conjunto de pesos ω, todos ellos, igual a 1 y un γ igual a -2 (podrían estar multiplicados por un escalar).

Observamos primero los resultados si no aplicamos el Kernel gaussiano. Como era de esperar por ser este un problema convexo, la solución del problema dual y el primal son idénticas para todas la ejecuciones indiferentemente de la *seed* usada o la ν. Al ser la nube de puntos linealmente separable observamos porcentajes de acierto elevados, cerca del 90 % (con pequeñas desviaciones) tanto para los datos de entrenamiento como para los de test con pocas variaciones, siempre que elijamos una ν adecuada. Este parámetro influye significativamente en el resultado del problema de optimización.

$\nu =$	0	0.1	0.5	1	5
ω_1	0.000	0.412	1.192	1.759	2.736
ω_2	0.000	0.588	1.784	2.397	3.465
ω_3	0.000	0.382	1.262	1.421	2.944
ω_4	0.000	0.412	1.312	1.995	2.839
γ	0.000	-1.300	-2.707	-3.857	-5.840
Acc. Tr	52.0 %	52.0 %	92.0 %	88.0 %	94.0 %
Acc. Te	48.0 %	52.0 %	86.0 %	96.0 %	90 %
f	0.00	4.38	16.04	26.60	88.0

Observamos que a medida que aumentamos la ν, los pesos ω y el intercept γ se magnifican aunque siempre acercándose a la recta $x_1 + x_2 + x_3 + x_4 - 2 = 0$ (el hecho de multiplicar por un escalar a ambos lados de la ecuación no modifica la recta). Esto sucede porque al aumentar la variable, estamos aumentando a su vez la penalización sobre los *slacks* y la minimización de estos se convierte cada vez más importante sobre la minimización de los pesos ω. Por otra parte, si ν tiende a 0 la minimización de los pesos es lo que predomina por lo que estos van tendiendo a 0 a medida que la variable reguladora decrece. Para el valor de $\nu = 0$, no se castiga el crecimiento de los *slacks* por lo que los pesos acaban valiendo 0. Esto provoca que no estemos calculando el plano separador correctamente y obtengamos, para valores de ν cercanos a 0, porcentajes de acierto muy bajos sobre el conjunto de test. Sin embargo, al aumentar la variable reguladora, como era de esperar, no se produce este comportamiento. Por ello podemos concluir que es preferible la elección de un valor de ν alto.

Por otro lado si usamos RBF y repetimos el experimento observamos:

$\nu =$	0	0.1	0.5	1	5
γ	-1.000	-0.693	-0.231	-0.088	0.033
Acc. Tr	52.0 %	52.0 %	90.0 %	90.0 %	100 %
Acc. Te	44.0 %	48.0 %	80.0 %	84.0 %	82.0 %
f	0.00	4.27	16.69	20.70	37.46

Vemos que el valor de γ para ν igual a 0 vale -1 y aumenta progresivamente con loa variable regularizadora. Además como en el caso anterior, la función objetivo aumenta a medida que lo hace la ν aunque si antes lo hacía indefinidamente ahora parece que el el crecimiento decrece significativamente a medida que aumenta ν. Por otro lado las *accuracies* sobre el conjunto de entrenamiento crecen con ν hasta estabilizarse en un 100 % mientras que en el conjunto de test se detienen en un aproximado de 80 %. Por tanto, aunque con los porcentajes de acierto pareciera ser que el método RBF es mejor para la

clasificación de puntos separables linealmente, con el conjunto de test comprobamos que no es cierto y que seguramente estemos produciendo un *overfitting* para la cantidad de puntos que tenemos que es bastante pequeña.

3.2. Swiss roll

Como hemos explicado anteriormente, *sklearn* genera un conjunto de puntos en 3 dimensiones que forman una espiral. Por lo tanto, si quisiéramos obtener una superficie de separación para estos puntos, esta sería extremadamente complicada y desde luego no lineal. Es por esto que esperamos que el kernel gaussiano nos sea útil ya que transformará el espacio de manera que podrá ser linealmente separable.

Si primero probamos a optimizar sin utilizar RFB, esperaremos obtener las mismas soluciones tanto con el problema primal cómo con el dual y ambas con un bajo porcentaje de acierto ya que los puntos no son linealmente separables.

$\nu =$	0	0.1	0.5	1	5
ω_1	0.000	0.099	0.104	0.104	0.104
ω_2	0.000	0.007	0.011	0.011	0.011
ω_3	0.000	-0.055	-0.053	-0.053	-0.053
γ	0.000	-0.256	-0.318	-0.318	-0.318
Acc. Tr	50.0 %	60.0 %	54.0 %	54.0 %	54.0 %
Acc. Te	54.0 %	64.0 %	44.0 %	42.0 %	54 %
f	0.00	3.84	19.181	38.35	191.75

Como podemos observar, a medida que aumenta ν, no conseguimos mejorar la separación entre grupos, parece ser que incluso empeoramos la clasificación a partir de $\nu = 0{,}5$. De este punto en adelante, todos los resultados de la optimización son iguales excepto el valor de la función objetivo en el óptimo. Este valor crece linealmente ya que los *slacks* obtenidos son los mismos, pero el parámetro de penaización (ν) de estos es mayor.

Veamos qué ocurre cuando utilizamos RBF:

$\nu =$	0	0.1	0.5	1	5
γ	-1.000	-0.656	0.488	0.600	0.787
Acc. Tr	50.0 %	50.0 %	96.0 %	98.0 %	100.0 %
Acc. Te	46.0 %	54.0 %	88.0 %	90.0 %	92.0 %
f	0.00	4.481	14.02	19.61	34.32

La mejora obtenida utilizando este Kernel es asombrosa. Conseguimos separar perfectamente los 2 grupos cuando $\nu = 5$ (precisión del 100 %) en el conjunto de entrenamiento y para $\nu \geq 0{,}5$ una precisión de aproximadamente 90 % en el conjunto de test. En cuanto a la evolución de los resultados con ν, podemos decir que se cumple el patrón visto en el dataset anterior, tanto γ como el valor de la función objetivo crecen con este parámetro.

3.3. Base de datos Skin

En el primer apartado hemos visto que nuestra SVM funciona bastante bien a la hora de clasificar datos linealmente separables. Pero ¿funcionará igual de bien con datos menos

'artificiales'? y sobre todo ¿Qué utilidad podemos sacar de esto? Para responder a estas preguntas probaremos a utilizar nuestra SVM en una base de datos real, *Skin* [1] que hemos presentado en la introducción. En la realización de este análisis, para poder obtener mejores conclusiones, utilizaremos una cantidad de puntos mayor, 250. Como previsiblemente, el resultado será más preciso con más puntos, usaremos un rango de valores de ν más amplio y con valores más pequeños ya que el peso de los *slacks* será mayor.

$\nu =$	0	0.000001	0.00001	0.0001	0.001	0.01
ω_1	0.000	0.001	0.009	0.015	0.020	0.021
ω_2	0.000	-8.23e-05	-8.25e-4	-0.005	-0.013	-0.015
ω_3	0.000	-0.001	-0.012	-0.018	-0.019	-0.019
γ	0.000	1.022	1.221	2.002	2.650	2.932
Acc. Tr	16.0 %	84.0 %	93.6 %	93.6 %	92.8 %	92.0 %
Acc. Te	20.4 %	75.6 %	91.6 %	94.0 %	91.6 %	93.6 %
f	0.00	7.89e-05	0.0007	0.005	0.048	0.474

Los resultados son nuevamente iguales para los problemas primal y dual aunque la tendencia de los resultados es algo diferente a los casos anteriores. Ahora a medida que aumenta ν, llega un momento que la precisión decrece y no conviene aumentar más la penalización de los slacks. Cabe destacar que las prestaciones del *SV classifier* son bastante buenas para los datos con hasta un 94 % de precisión.

Veamos cómo afecta la utilización de la RBF en este dataset:

$\nu =$	0	0.000001	0.00001	0.0001	0.001	0.01
γ	-1.000	-1.000	-1.000	1.000	0.997	0.974
Acc. Tr	16.0 %	16.0 %	16.0 %	84.0 %	84.0 %	84.0 %
Acc. Te	19.6 %	21.2 %	17.2 %	79.2 %	82.8 %	83.6 %
f	0.000	8.0e-5	8.0e-4	0.008	0.079	0.773

En este caso sucede algo que no pasaba en los casos anteriores. La evolución de los resultados para los mismos valores de ν que en la optimización primal y dual no es la misma. Parece ser que necesitaríamos valores más grandes de ν para llegar a valores altos de precisión.

Si continuamos con la progresión de ν:

$\nu =$	0.1	1	10	100
γ	0.763	1.079	2.268	3.308
Acc. Tr	97.2 %	98.4 %	99.2 %	99.6 %
Acc. Te	94.8 %	97.6 %	98.6 %	99.6 %
f	5.365	20.138	71.977	270.843

Efectivamente, observamos que para este caso se necesita un valor de ν mayor para obtener los mejores resultados en cuanto a precisión de clasificación. Para la base de datos *Skin*, observamos que los porcentajes de acierto sobre los datos al aplicar la RBF son mayores alcanzando casi el 100 % de *accuracy* tanto para los datos de test cómo de training.

4. Conclusiones

La primera conclusión a la que llegamos es que el parámetro ν es bastante sensible y que necesitamos algún criterio para elegir su valor antes de la optimización. Un criterio bastante lógico sería utilizar *Cross-validation* en el que haríamos varias ejecuciones con diferentes ν y escogeríamos la óptima en base al error medio de test.

Por otro lado, hemos observado que en términos de precisión de *training* al aplicar el Kernel gaussiano se obtienen mejores resultados, sin embargo esto provoca que perdamos la interpretabilidad del modelo que nos ofrece el conocer los coeficientes del hiperplano de separación así como un menor control sobre el mismo modelo.

Así pues, con esta práctica hemos conseguido profundizar en la comprensión de la formulación del problema SVM y la interpretación de sus resultados. Hemos podido observar como la *Support Vector Machine* nos ofrece un gran rendimiento en cuanto a tiempo de entrenamiento y a precisión de clasificación. Es por ello, que a raíz de esta práctica asumimos la SVM como un método útil para afrontar problemas de clasificación a parte de los vistos en otras asignaturas.

Referencias

[1] Rajen Bhatt and Abhinav Dhall. *Skin Segmentation Data Set*. UCI Machine Learning Repository, 2009.