# SVM
# SUPPORT VECTOR MACHINE

EJEMPLO DE USO

***Maquinas de Soporte Vectorial***

## Las Máquinas de Soporte Vectorial:

MSV es uno de los algoritmos clásicos del Machine Learning y que puede ser usado en combinación con arquitecturas como las Redes Neuronales o las Convolucionales.

El principal uso de las Máquinas de Soporte Vectorial se da en la clasificación binaria, es decir para separar un set de datos en dos categorías o clases diferentes.

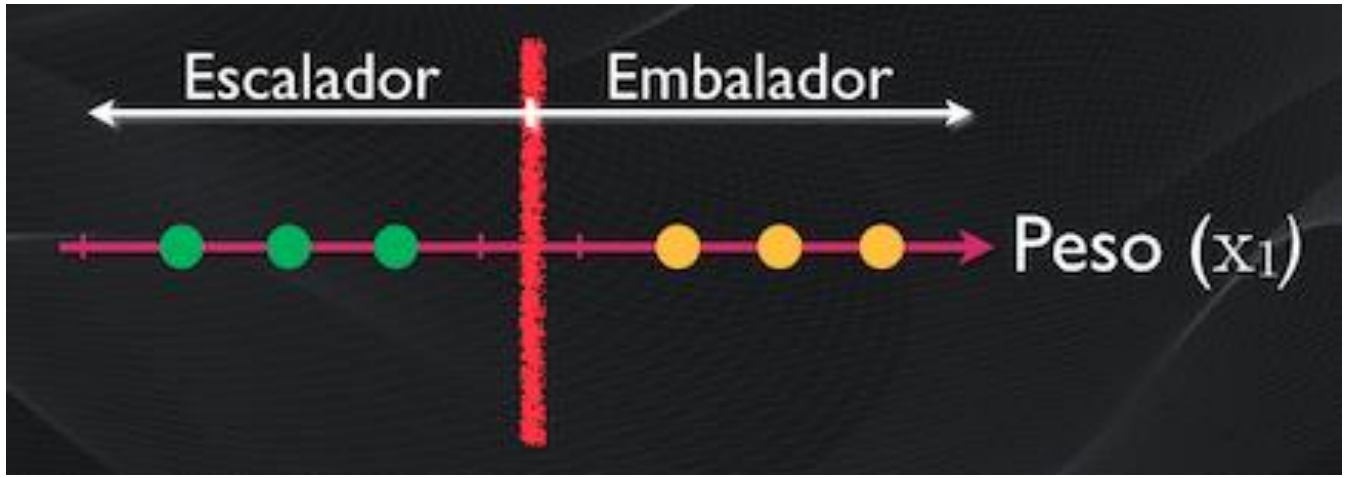
Las Máquinas de Vector Soporte se fundamentan en el Maximal Margin Classifier, que a su vez, se basa en el concepto de hiperplano. Comprender los fundamentos de las SVMs requiere de conocimientos sólidos en álgebra lineal. Una descripción detallada en el libro Support Vector Machines Succinctly by Alexandre Kowalczyk.

*EJEMPLO*

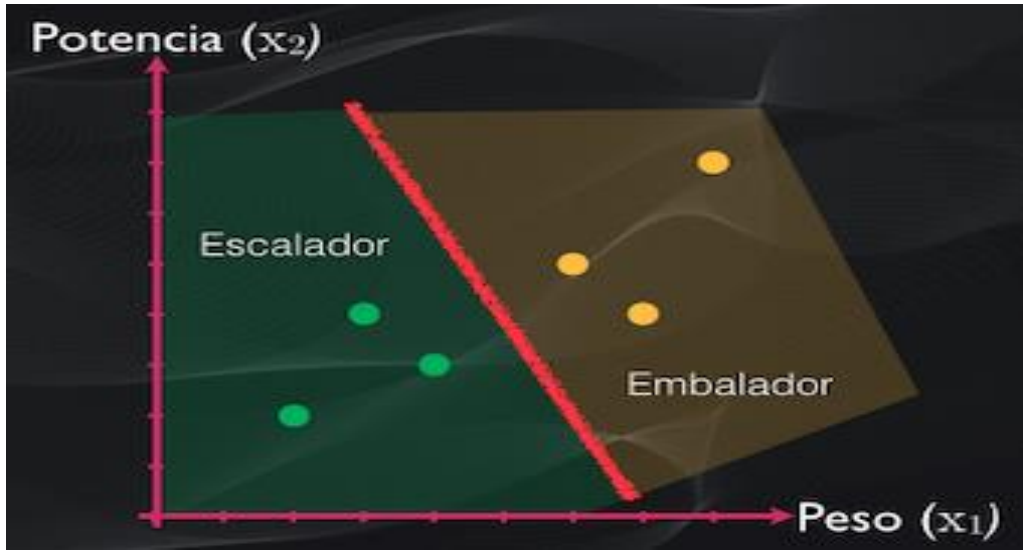
Supongamos que estamos conformando un equipo de ciclismo y debemos seleccionar a los ciclistas en una de dos categorías: escalador (especialista en la montaña), o embalador (especialista en las llegadas con terreno plano).

Para asignar la categoría a la que pertenece cada ciclista, es decir para clasificarlo, podemos usar varias *características*.

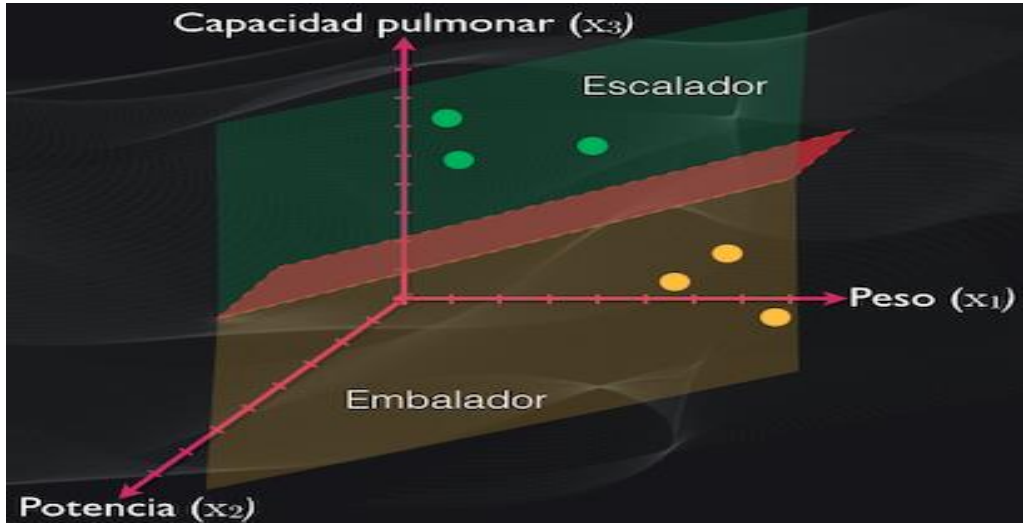
Consideremos una primera característica: el peso. Generalmente los ciclistas más livianos son mejores escaladores, y aquellos con más peso tienen más potencial para el embalaje. En este caso podemos definir un umbral y al momento de la clasificación simplemente definimos la categoría dependiendo si el peso del ciclista está a la izquierda o a la derecha de este umbral:



Pero para mejorar la precisión de la clasificación podemos agregar más características. Por ejemplo, si añadimos la potencia que desarrolla el ciclista en cada pedalazo.
Ahora tenemos dos variables: peso y potencia, esto podríamos representarlo en dos dimensiones y trazar una línea que divida a estos dos grupos.




Ahora incluyamos otra característica por ejemplo la capacidad pulmonar.
Ahora tenemos tres características: peso, potencia y capacidad pulmonar. Entonces tendremos una distribución de puntos en tres dimensiones, y en este caso tendremos un plano que separa una categoría de otra:

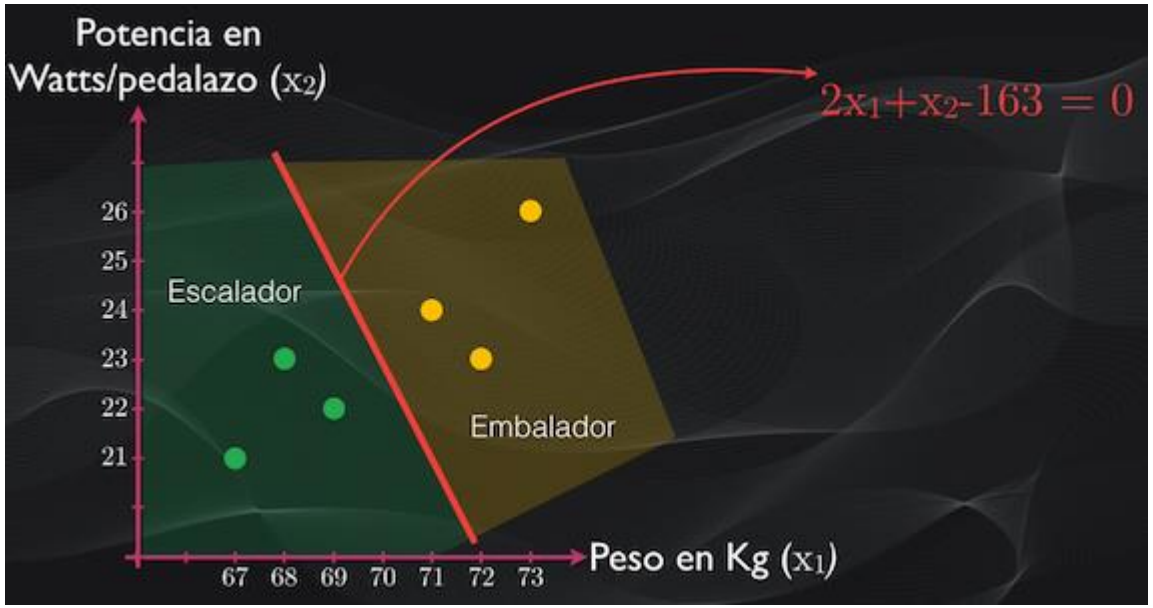



Y con cuatro o más dimensiones no resulta fácil dibujarlo, pero la idea es que en todos los casos idealmente tendremos una frontera de decisión que permite determinar la categoría a la que pertenece cada ciclista.

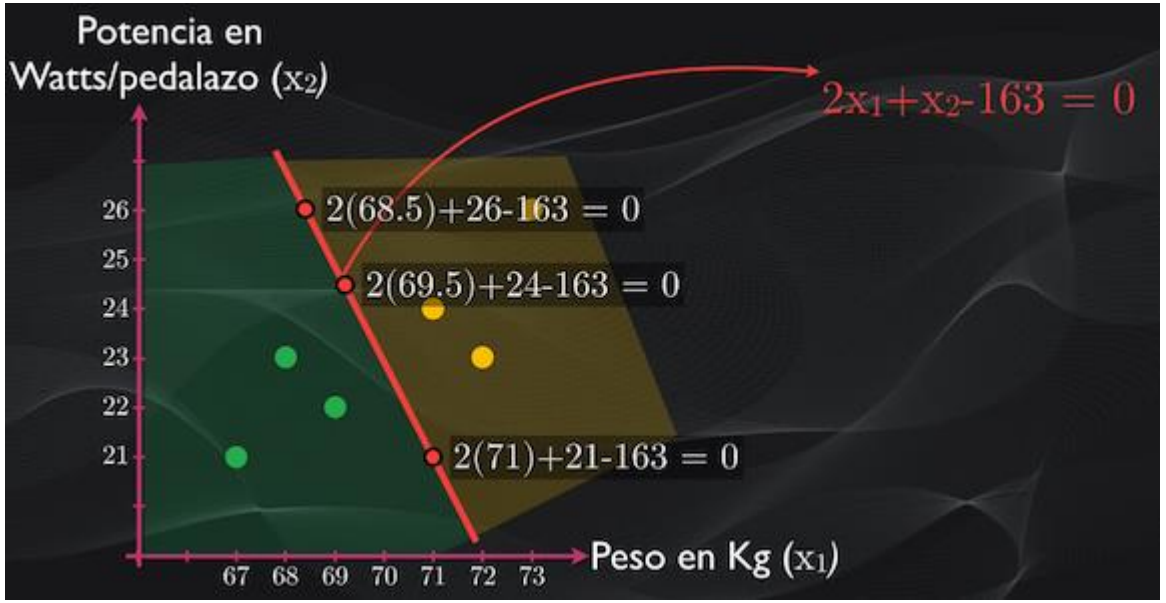
## El hiperplano y la clasificación

En adelante vamos a llamar a esta frontera el **hiperplano** y para la explicación nos enfocaremos en dos dimensiones, pues resulta más fácil de entender gráficamente. Pero estos mismos conceptos se aplican para una, tres o más dimensiones.

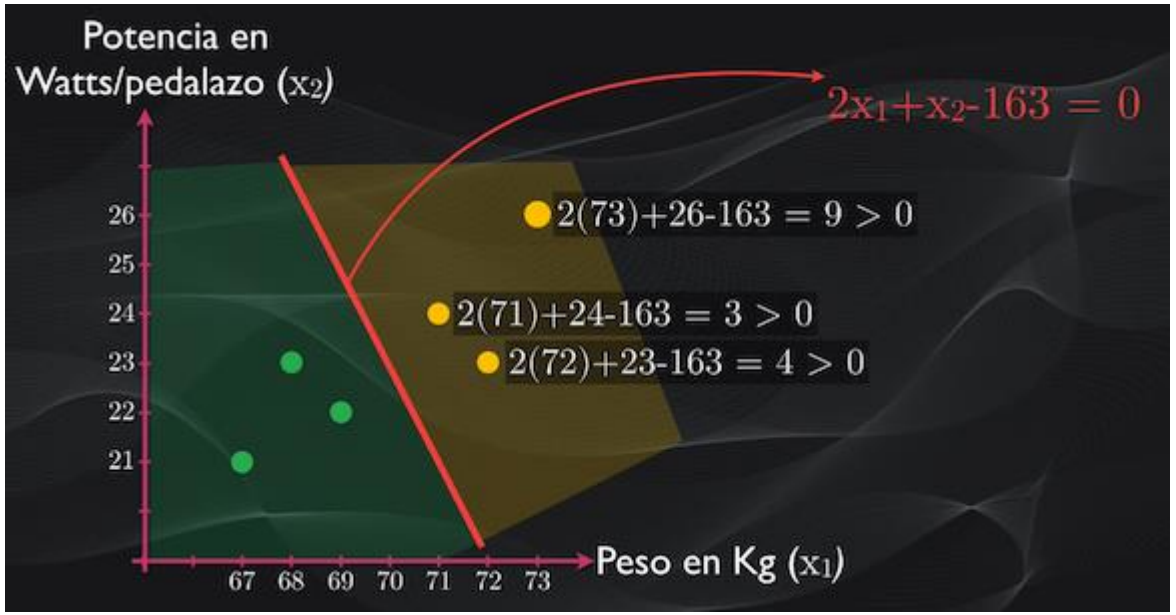
En dos dimensiones, nos enfocaremos en dos características: el peso y la potencia de cada pedalazo. Podemos ver que los dos grupos están separados por un hiperplano que en este caso es simplemente una línea recta:



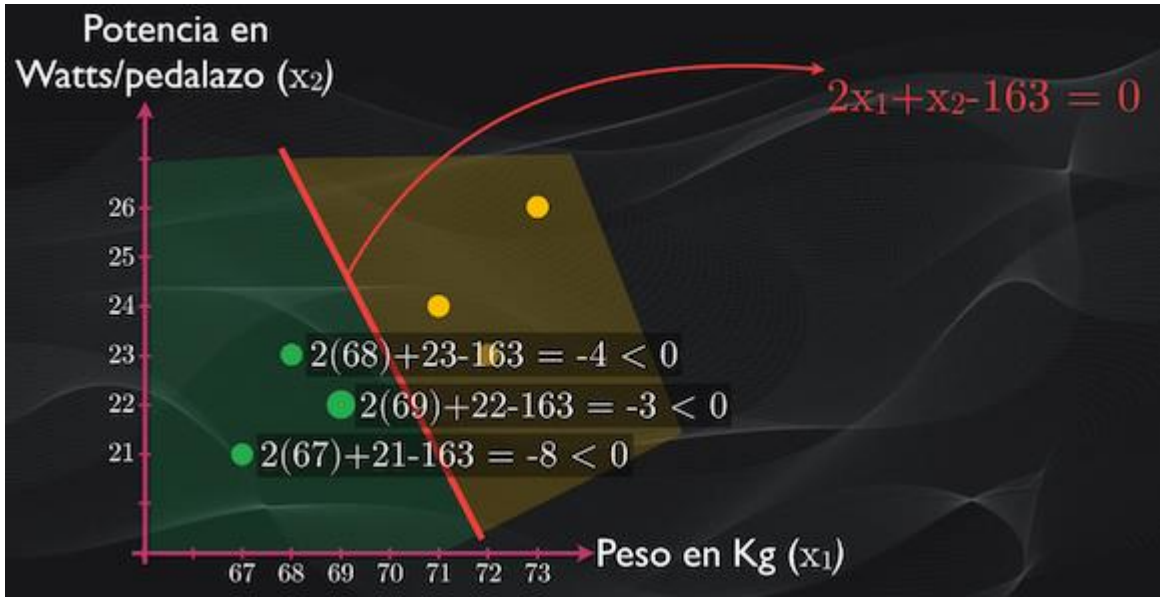
Si analizamos esta línea recta veremos que cualquier punto que pertenezca a ella tiene una característica importante: al reemplazarlo en la ecuación el resultado será exactamente igual a cero. Veamos por ejemplo varios de estos puntos:




Pero si tomamos por ejemplo los puntos en los cuales están ubicados los embaladores, veremos que al reemplazarlos en la ecuación del hiperplano todos son mayores que cero.



mientras que al hacer lo mismo para los puntos donde están los escaladores se obtienen valores menores que cero:




Así que partiendo de esto podemos definir un algoritmo para realizar la clasificación:

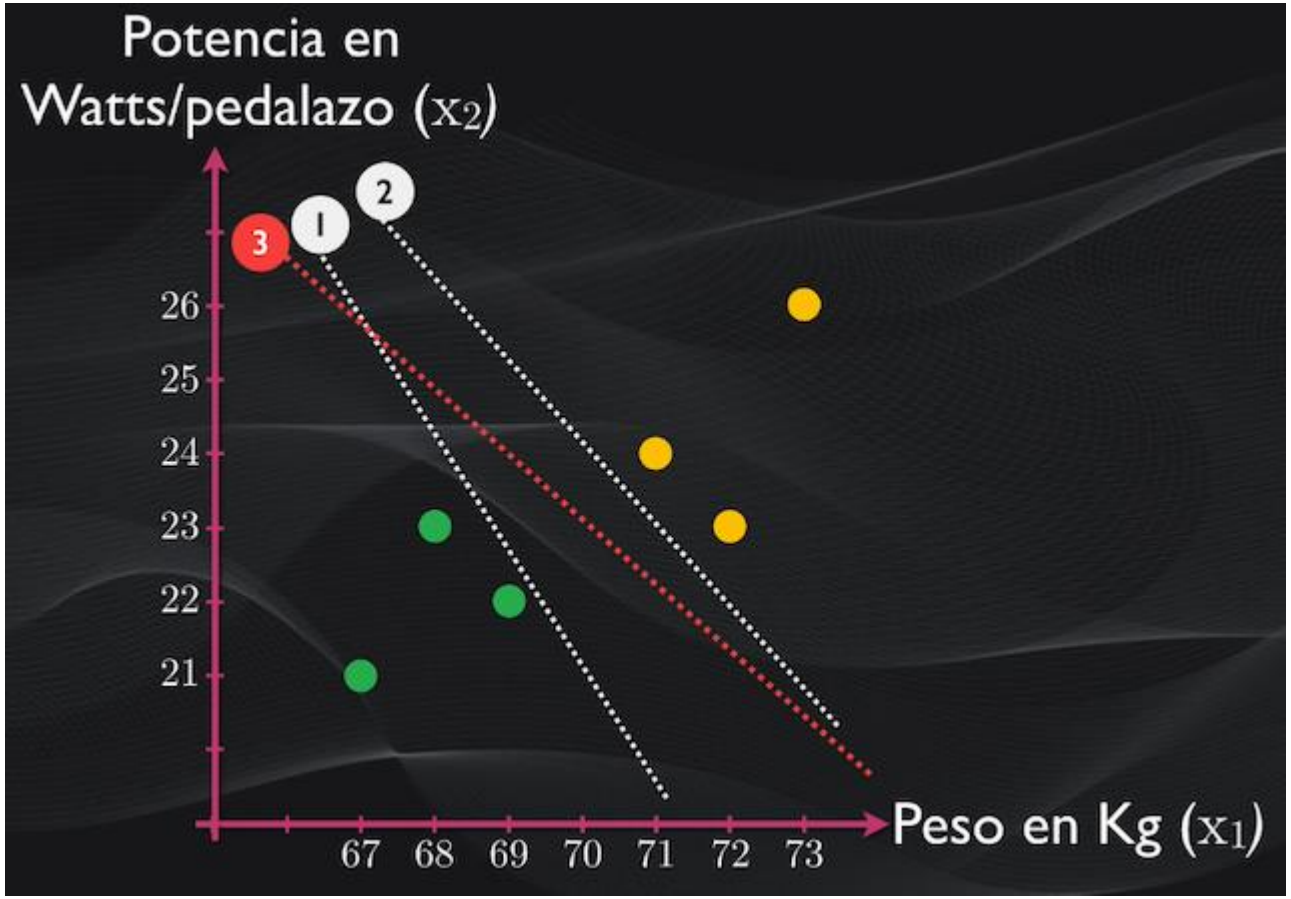
1. Obtener la ecuación del hiperplano, lo que equivale a encontrar sus coeficientes.
2. Por cada dato que deseo clasificar reemplazar sus coordenadas en la ecuación del hiperplano, y dependiendo del valor obtenido clasificar el dato en una u otra categoría.

Si miramos en detalle este sencillo algoritmo veremos que el problema de la clasificación se reduce a encontrar la ecuación de este hiperplano. Y es allí donde entra en juego el algoritmo de Máquinas de Soporte Vectorial.

## El mejor hiperplano y las Máquinas de Soporte Vectorial: clasificación “hard margin” (margen duro)

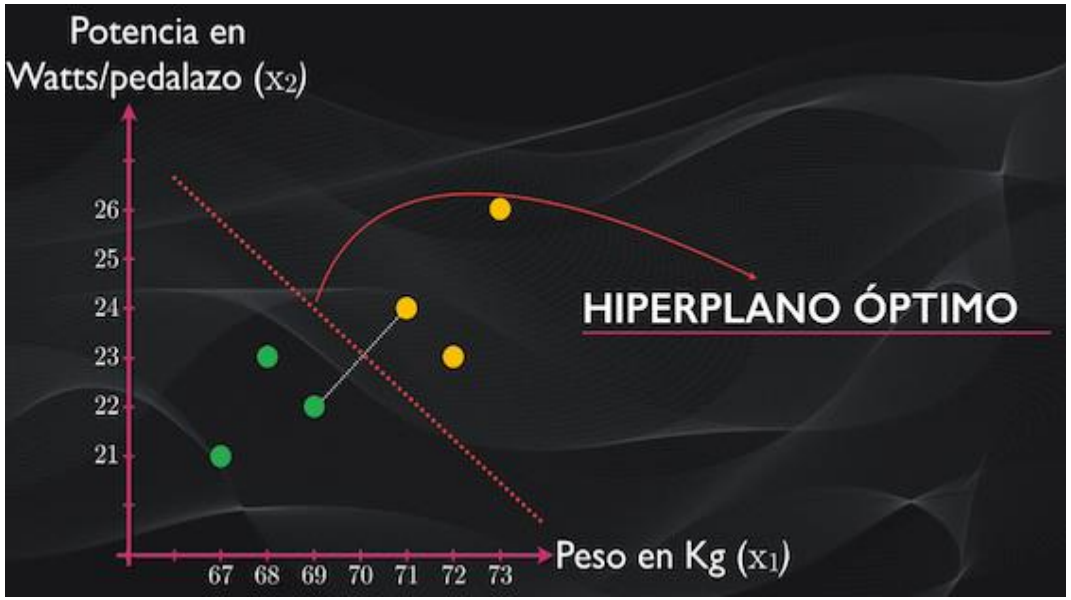
Para entender cómo funciona este algoritmo debemos comprender un concepto muy importante: el de el mejor hiperplano.

Al intentar separar nuestro set de datos podemos obtener diferentes líneas o hiperplanos, y todos ellos logran dividir correctamente el set en dos categorías. Pero, ¿cuál de ellas es mejor?



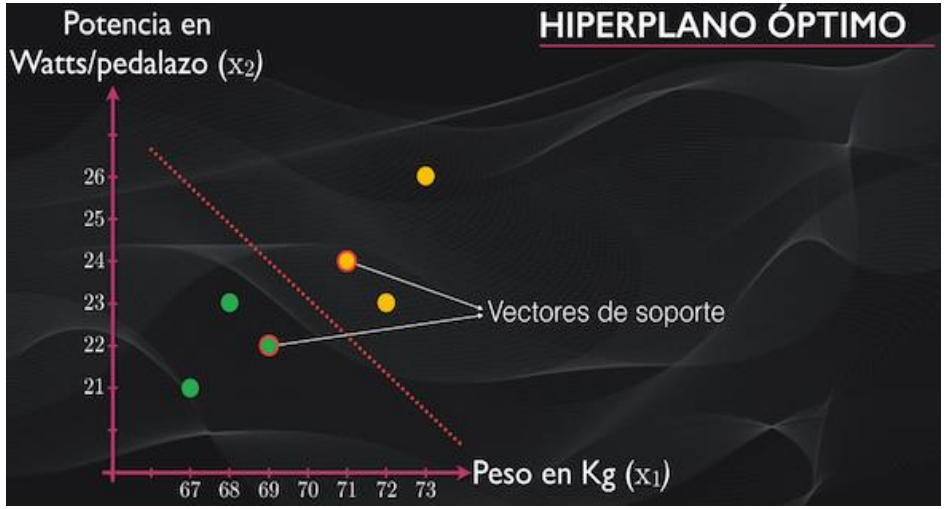

Vemos que las líneas 1 y 2 están demasiado cerca de una de las categorías, mientras que la línea tres está en un punto intermedio entre las dos agrupaciones. Esta línea es precisamente el hiperplano óptimo, pues es la que se encuentra más alejada de todas las observaciones, y esto hace que al momento de clasificar un nuevo dato no exista un sesgo hacia una categoría u otra.

El algoritmo de Máquinas de Soporte Vectorial permite precisamente obtener este hiperplano óptimo. Y aunque existen diferentes maneras de implementarlo computacionalmente, en esencia lo que en el fondo logra hacer este algoritmo es primero detectar los puntos más cercanos entre una clase y otra, luego encuentra la línea que los conecta y finalmente traza una frontera perpendicular que divide esta línea en dos. La línea que se obtiene es precisamente el hiperplano óptimo:

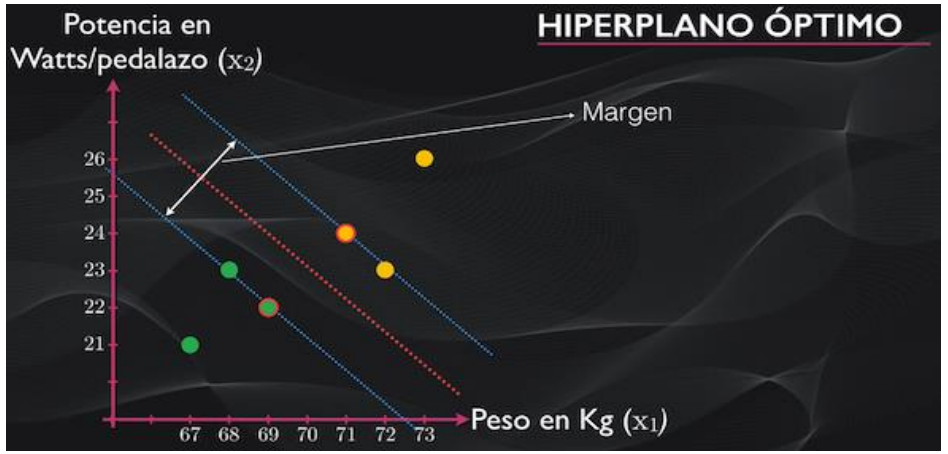


Y acá debemos resaltar tres definiciones importantes:

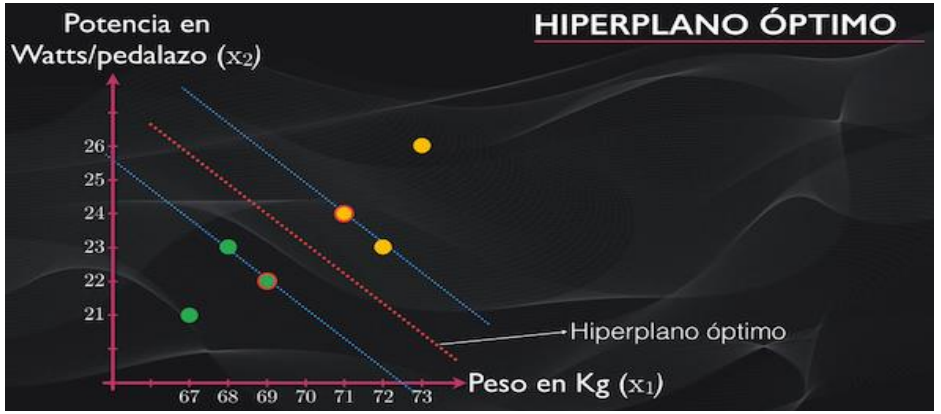
1. Los **vectores de soporte**, que son precisamente los puntos más cercanos entre una clase y otra, y son los que le dan el nombre al algoritmo:




2. El **margen**, que es la distancia entre el hiperplano y los vectores de soporte:



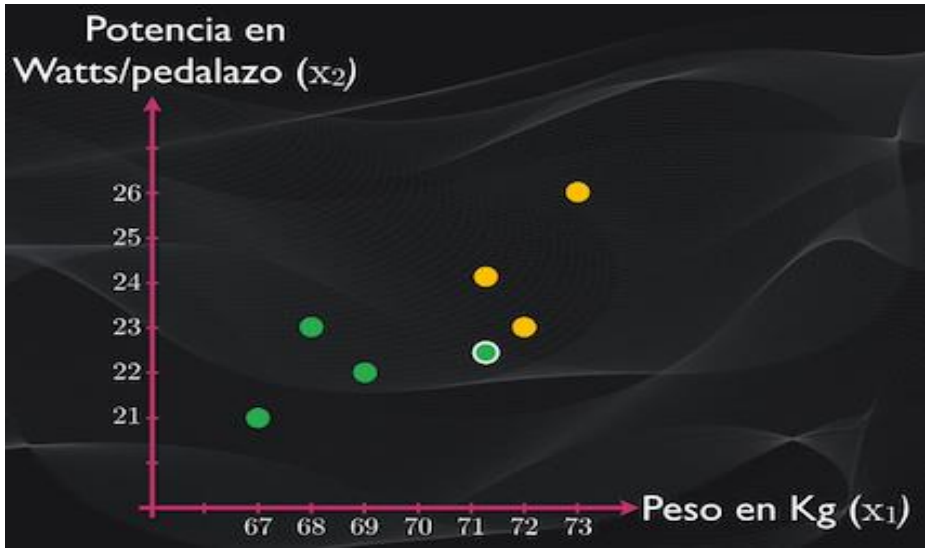

3. Y el mismo **hiperplano óptimo** que es la frontera de separación que consigue el mayor margen posible:

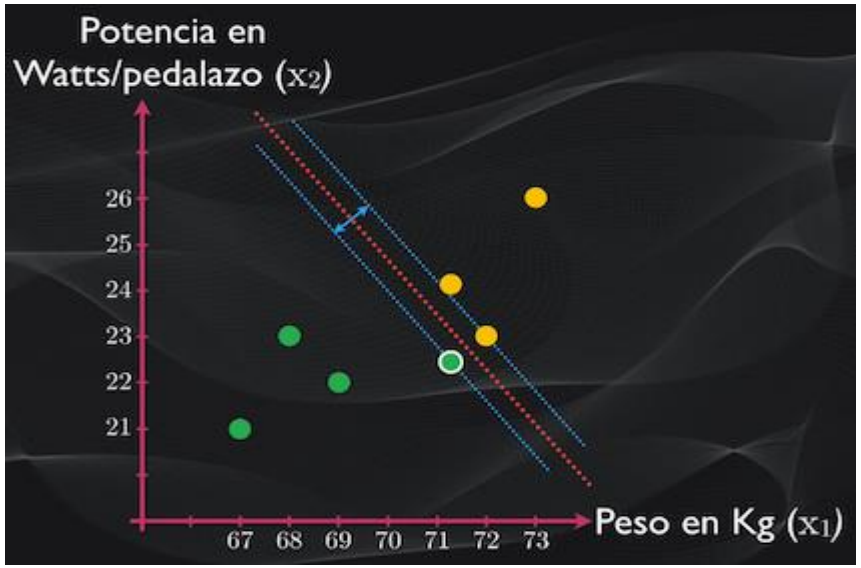


## Presencia de *outliers* y el parámetro C: clasificación “soft margin” (margen suave)

Pero, el algoritmo descrito funciona sólo para el caso ideal, en el que nuestros datos están perfectamente separados y podemos usar una línea recta o frontera de decisión óptima, es decir con un margen bastante amplio.

Pero, volviendo a nuestro ejemplo, qué pasaría si llega a nuestro equipo un corredor excepcional, que se destaca en la montaña pero que también es muy buen embalador. En este caso ese nuevo corredor se comporta como un *outlier* o valor atípico, pues no obedece al comportamiento esperado para un escalador:



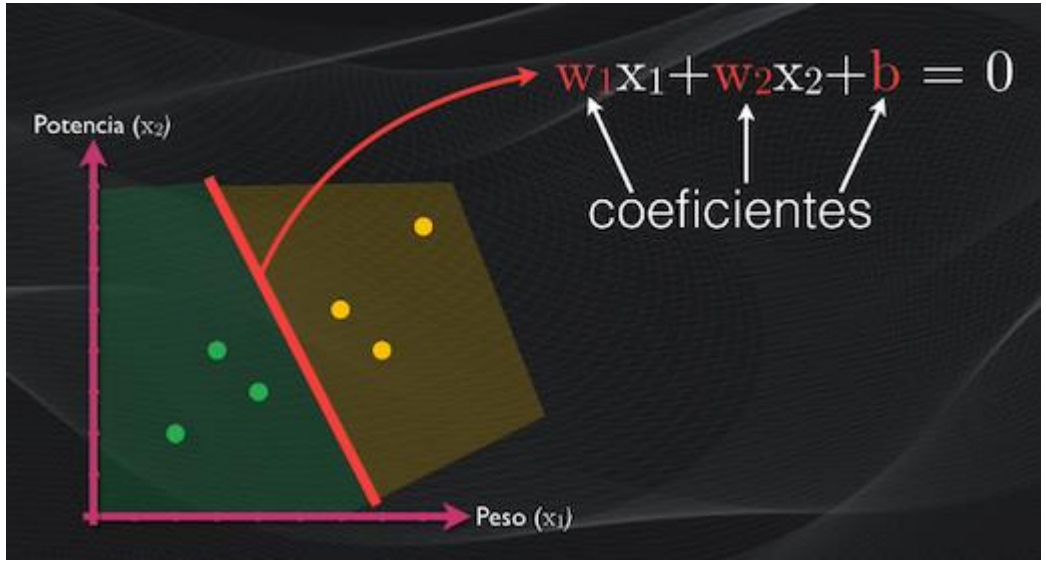




Si aplicamos el algoritmo explicado anteriormente veremos que se obtendrá un hiperplano óptimo pero que el margen será muy pequeño. Esto se debe a que los vectores de soporte están más cerca. Y esto puede llevar al overfitting: es decir que si introducimos un dato nuevo, que no haya sido nunca visto por el algoritmo, muy probablemente será clasificado incorrectamente porque el margen es muy reducido y no hay una adecuada separación entre las clases:

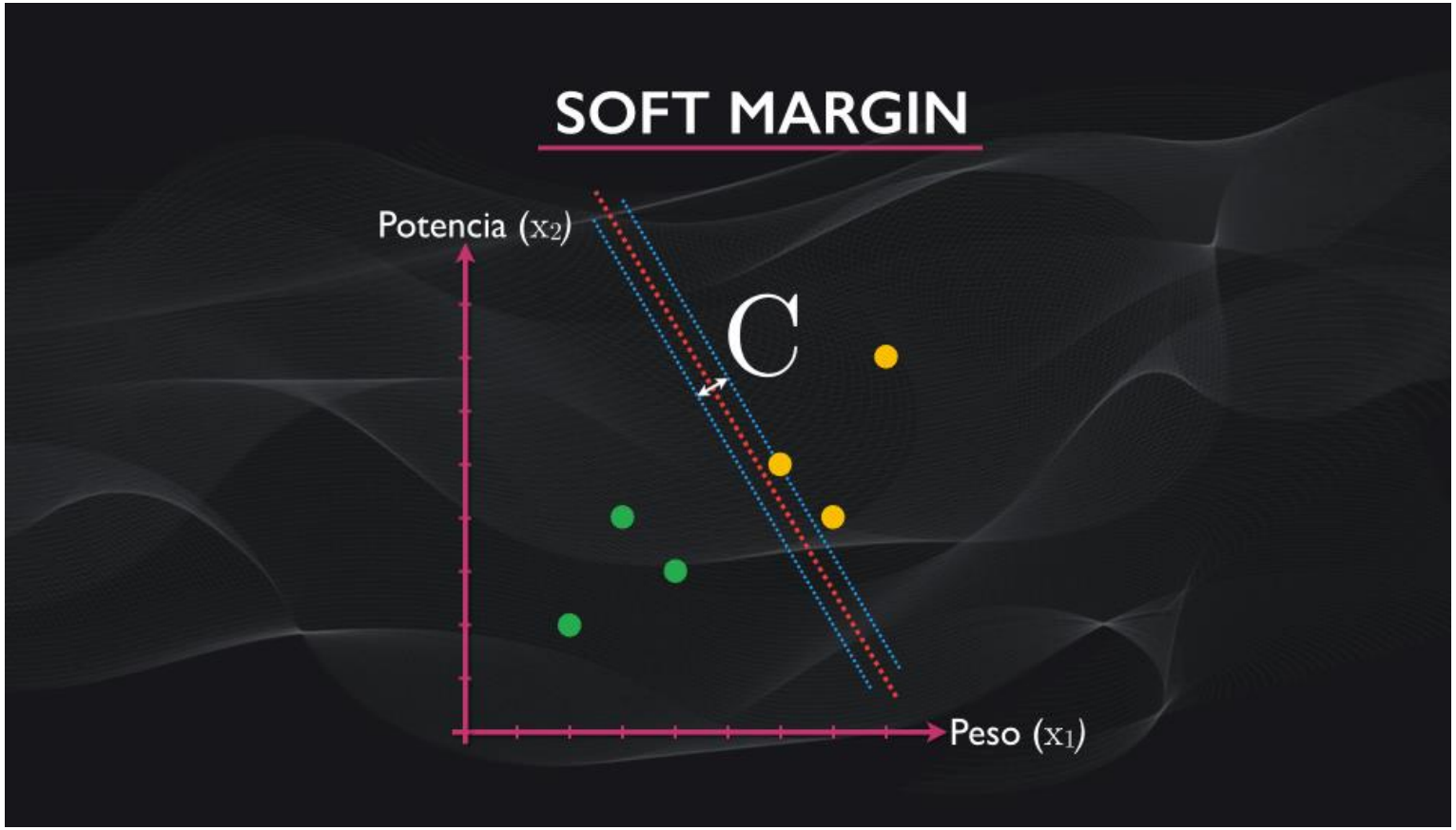
Así que el algoritmo que vimos en la sección anterior, conocido como *hard margin* (o margen duro) no resulta muy flexible y no es ideal usarlo si hay outliers como el de nuestro corredor excepcional.

La manera de resolver esto es ensanchando el margen, permitiendo que la mayoría de los datos estén clasificados correctamente, y aceptando la posibilidad de que existan unos cuantos errores al momento de la clasificación.

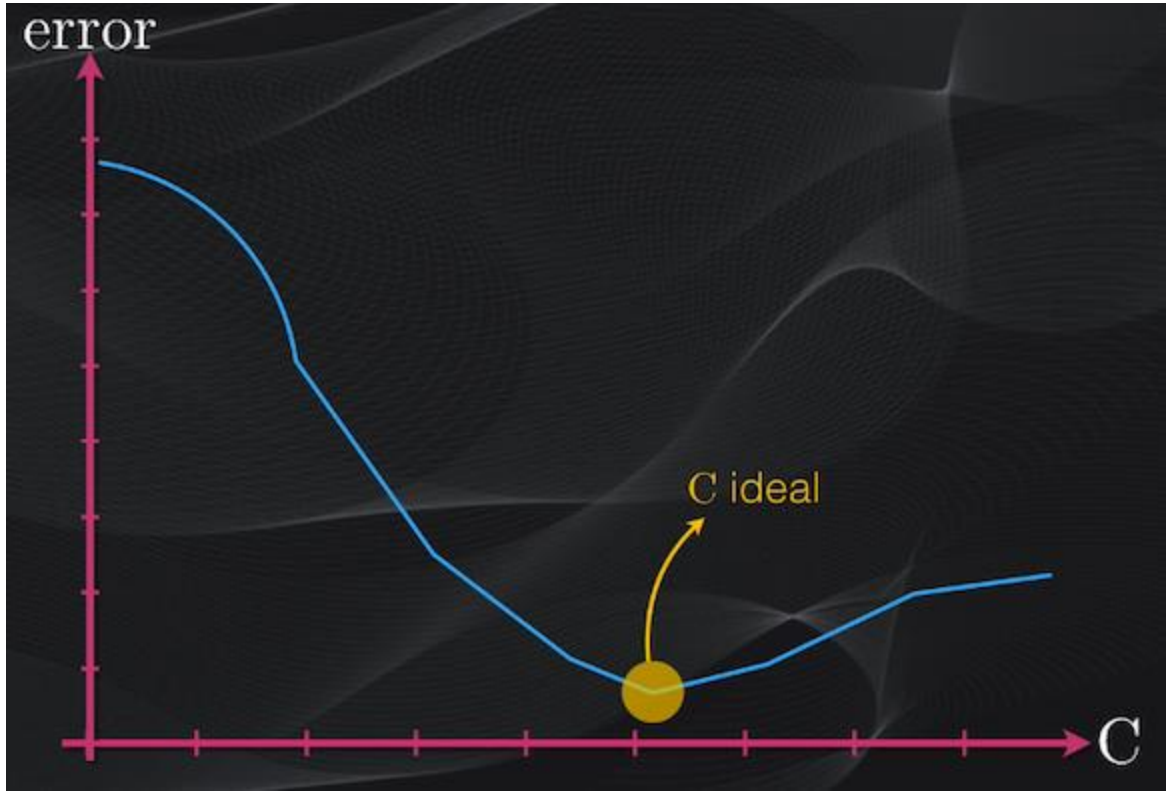
Esto se logra modificando el algoritmo para obtener el hiperplano óptimo: originalmente en el clasificador hard margin se buscaba maximizar el margen, y para esto se modificaban únicamente los coeficientes del hiperplano:



Ahora, para contrarrestar el efecto del outlier, se incluye un término adicional a esta función, lo que permite flexibilizar el margen. Este término depende de un parámetro *C*, un hiperparámetro, que yo como diseñador elijo durante el entrenamiento. La idea es que un *C* relativamente pequeño permite generar márgenes amplios, y a medida que su tamaño aumente el margen se va reduciendo poco a poco:



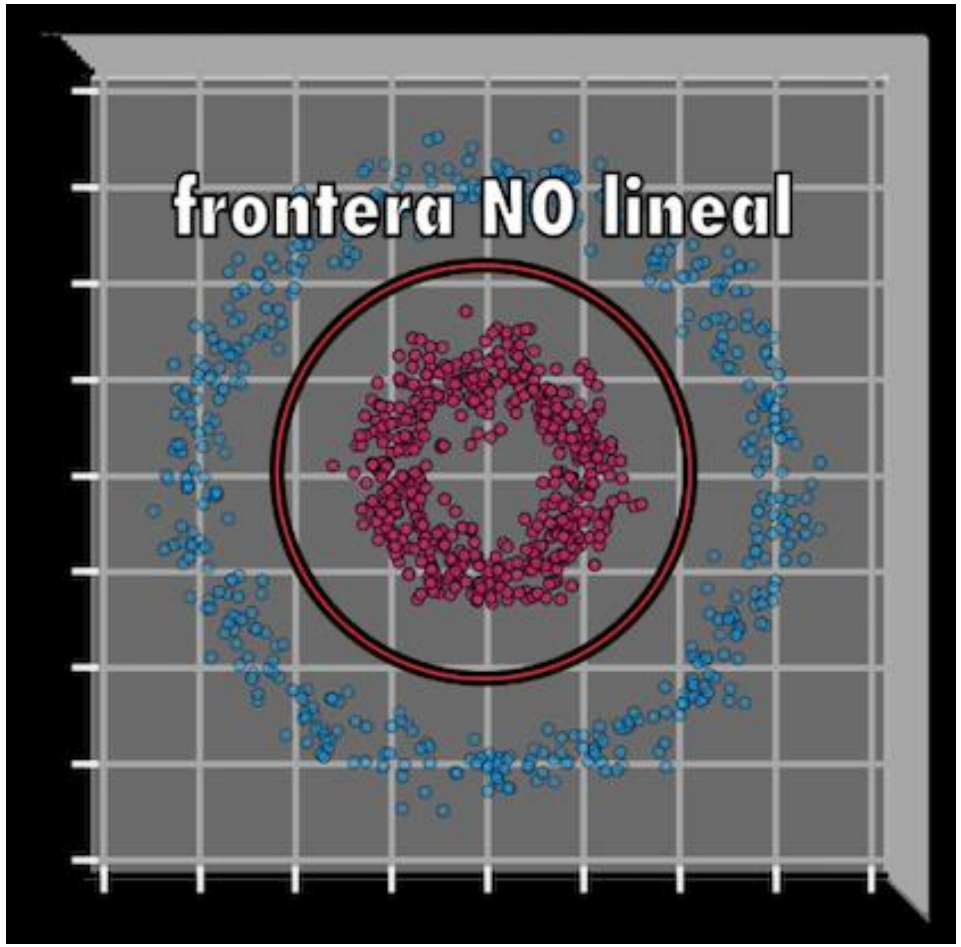
Este parámetro se escoge de manera empírica analizando el error que se obtiene en la clasificación vs. diferentes valores de C, y a este algoritmo de máquina de vectores de soporte se le conoce como “soft margin”:



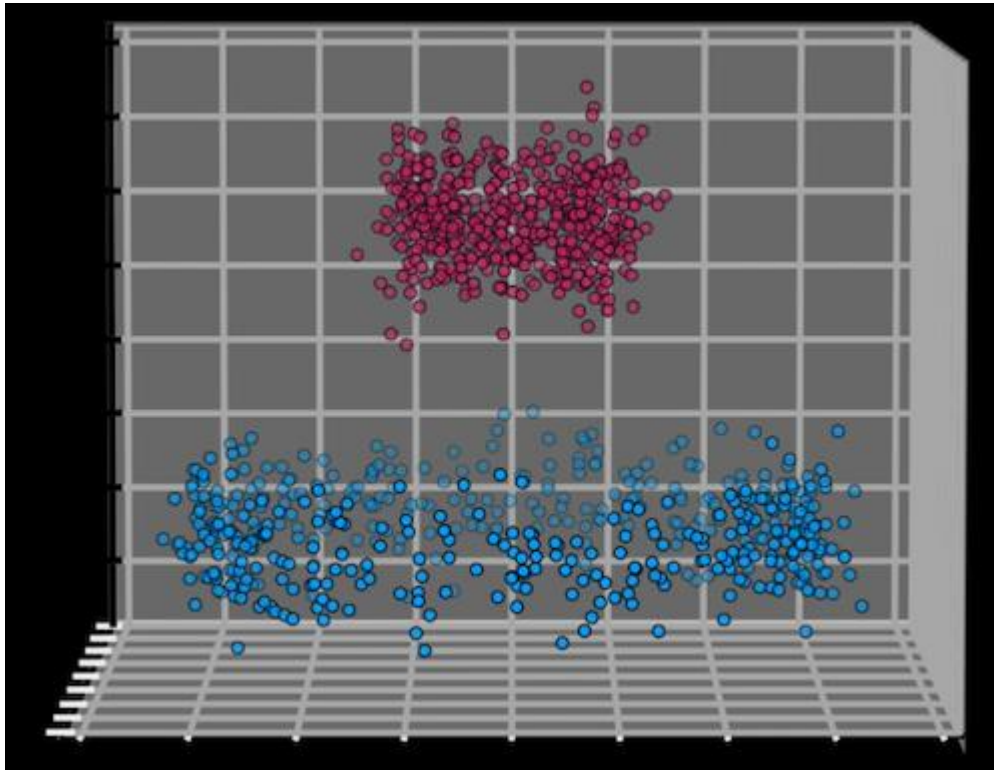
# Máquinas de Soporte Vectorial no lineales: el truco del Kernel

Bien, ya tenemos un clasificador mucho más versátil y sobre el cual tenemos algún tipo de control al momento de obtener el margen. Pero todavía estamos considerando una situación bastante ideal: la mayoría de los datos están lo suficientemente separados y basta trazar una simple línea recta para poder clasificar correctamente a la mayoría de ellos.

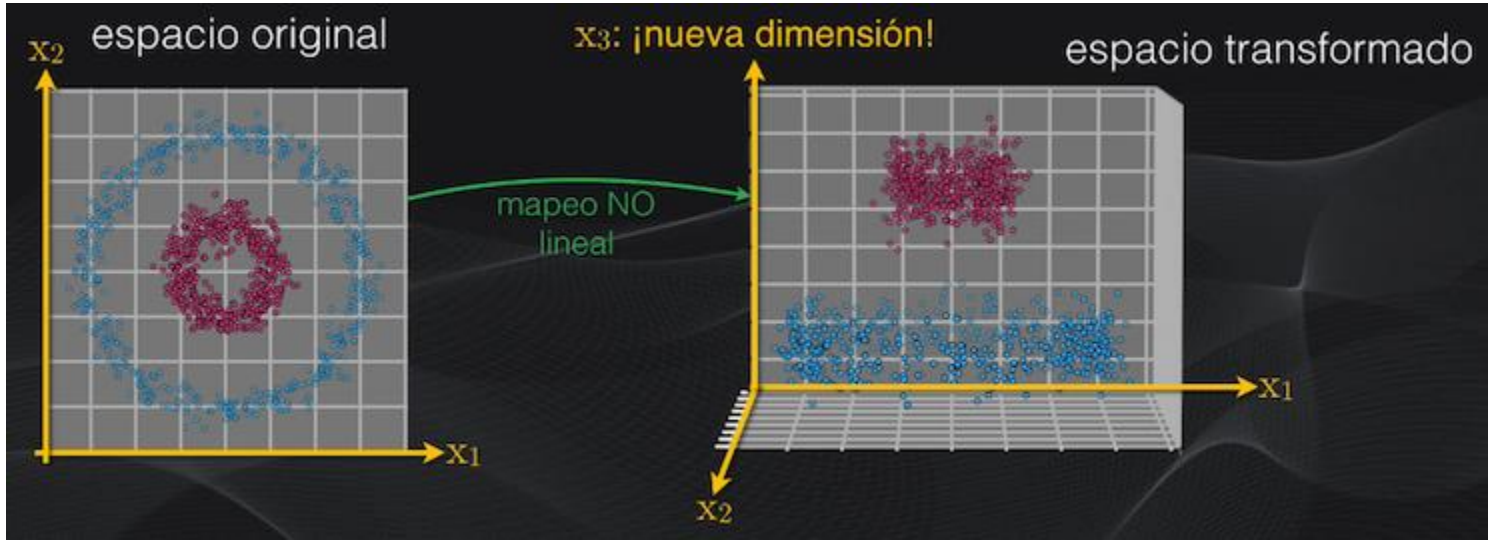
Pero en casos reales, la situación es más complicada. Las fronteras de división usualmente no son líneas rectas sino que tienen formas mucho más complejas. Y el problema es que, hasta donde hemos visto, las máquinas de soporte vectorial sólo permiten obtener hiperplanos o fronteras de decisión lineales. Entonces, ¿Qué se puede hacer en este caso?



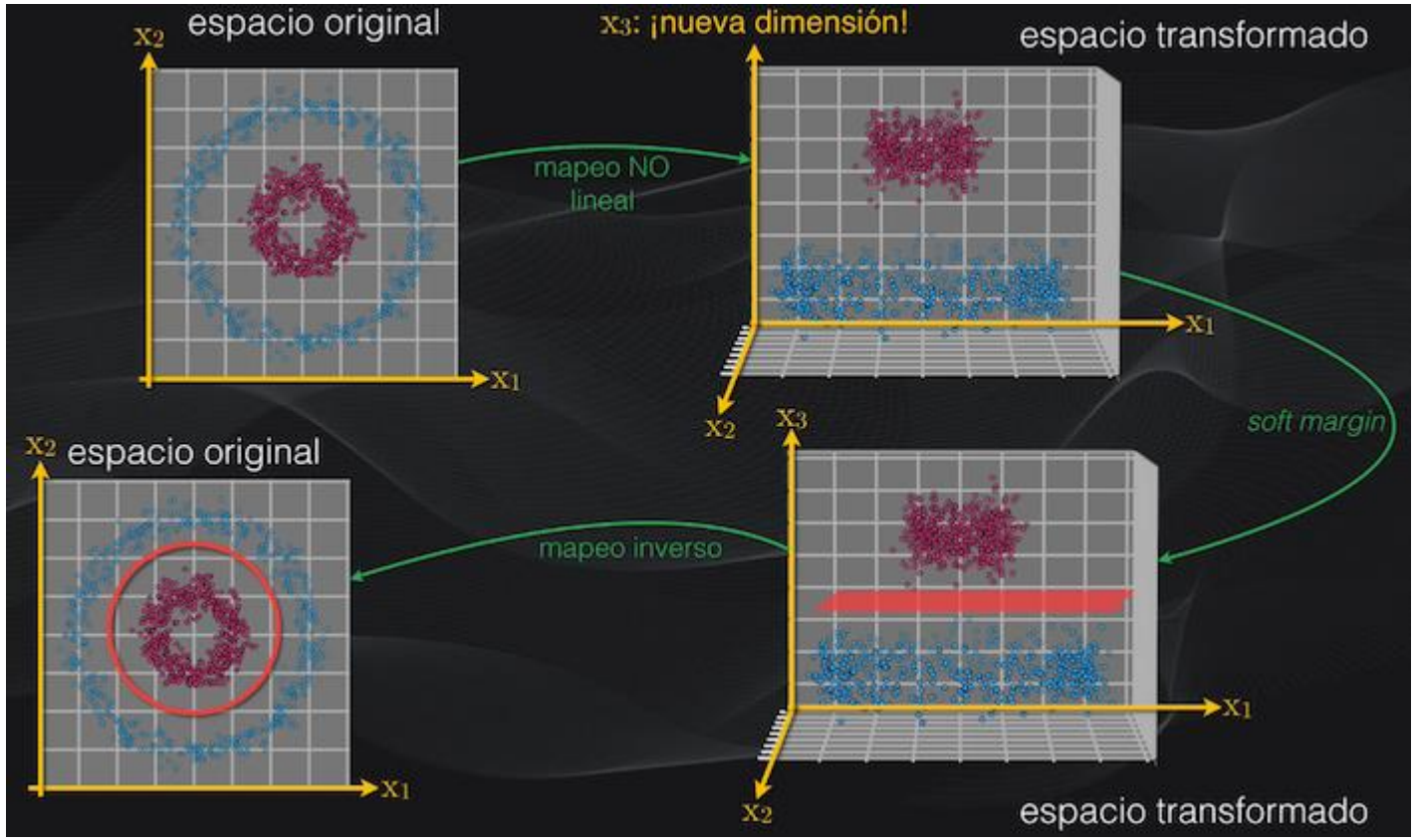
Una alternativa sería agregar más dimensiones a cada dato. Es decir, ¿qué pasaría si encontramos una forma de agregar una o más dimensiones adicionales para así lograr separar las dos categorías? En este caso podríamos usar el mismo algoritmo de máquinas de vectores de soporte para clasificar los datos en esas tres dimensiones:



¿Pero cómo logramos agregar más dimensiones a los datos para así lograr su clasificación? Pues el método usado en las máquinas de soporte vectorial se conoce como el **Truco del kernel**. Básicamente consiste en tomar el set de datos original, que no es separable, y mapearlo a un espacio de mayores dimensiones usando una función no lineal:



La idea es que con esta transformación el dataset ahora será separable linealmente, es decir que se puede usar una máquina de vectores de soporte tipo soft margin, como la que vimos anteriormente, para obtener el hiperplano óptimo. Una vez lo hayamos obtenido hacemos la transformación inversa, para volver al espacio original, y así llevar a cabo la clasificación:



En la práctica el truco del Kernel no implementa todos estos pasos, sino que realiza este mapeo y el cálculo del hiperplano de una manera simplificada, usando algo de álgebra lineal para evitar demasiadas operaciones y hacer más rápido el algoritmo.
Las transformaciones más usadas en este truco del kernel se logran usualmente a través de dos tipos de funciones:

1. Las polinomiales, que implican obtener combinaciones de los vectores de características usando potencias mayores que 1, o
2. Usando funciones gaussianas, con forma de campana, que se conocen como funciones de base radial.

En cualquiera de estos casos, y dependiendo del set de datos, lo que se logra es añadir más dimensiones a los datos originales para, en este espacio de más dimensiones, lograr la separación lineal de los datos.

Aunque las Máquinas de Soporte Vectorial fueron desarrolladas a comienzos de los años 90, aún son muy usadas en la actualidad pues permiten separar de manera óptima dos agrupaciones de datos, logrando crear una frontera de decisión que está a la misma distancia de ambas agrupaciones.

Esto hace que una Máquina de Vectores de Soporte tenga usualmente una precisión más alta que un perceptrón o neurona artificial. De hecho esto ha permitido que en la actualidad sea común encontrar sistemas de procesamiento de imágenes, como por ejemplo para el reconocimiento facial, que extraen características de las imágenes usando redes convolucionales y que luego alimentan estas características a una Máquina de Soporte Vectorial para realizar la clasificación final del rostro.

Obtenido de:

https://codificandobits.com/blog/maquinas-de-soporte-vectorial/

# 3 Feature Space



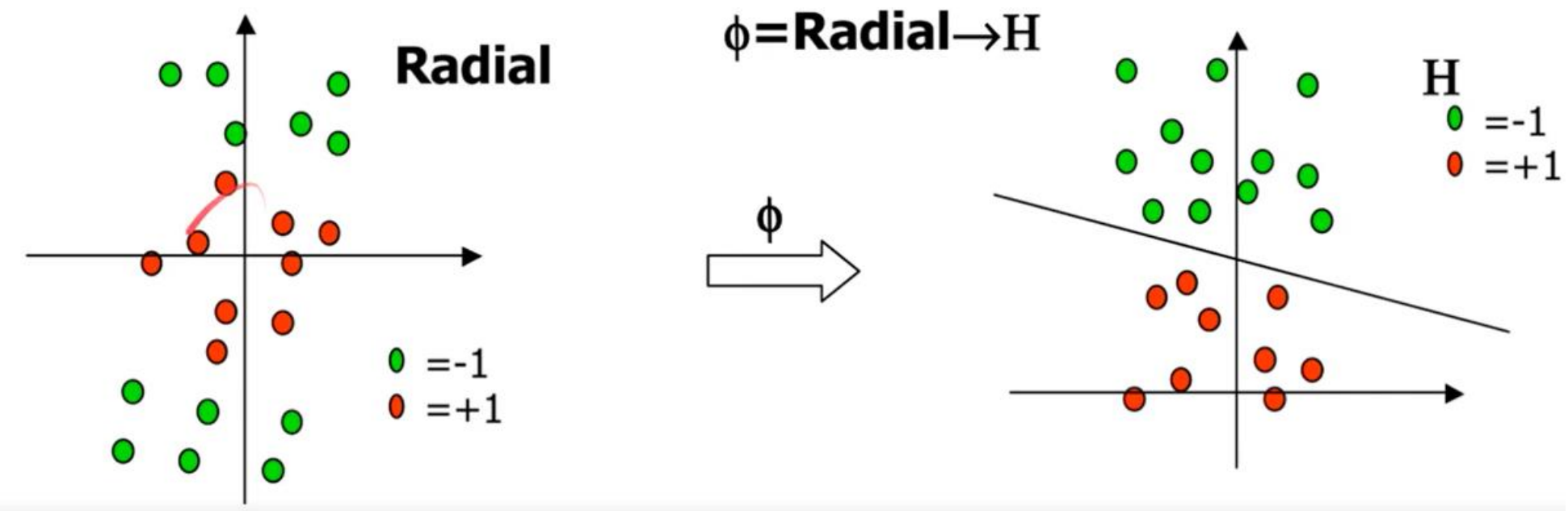
$$Kernel\ lineal : k(x, y) = x^T y + c$$

$$Kernel\ polinomial : k(x, y) = (\alpha x^T y + c)^p$$

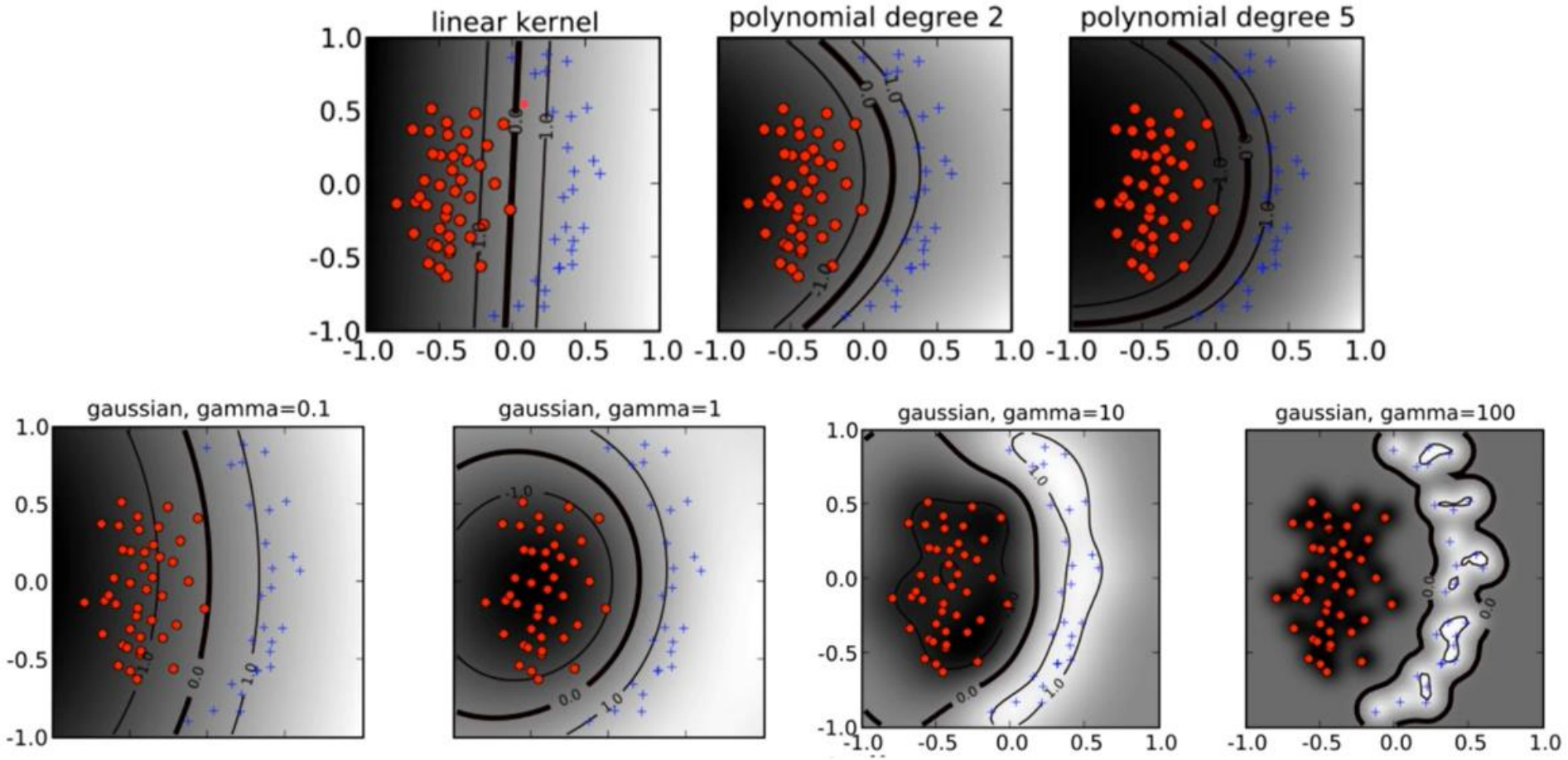
$$Kernel\ gaussiano(radial) : k(x, y) = exp(-\gamma||x - y^2||)$$

## ¿Para qué sirve el Estimador de Máxima Verosimilitud? (MLE)

Por ejemplo usando un kernel radial podemos transformar puntos que se muestran en una distribución “circular” de coordenadas cartesianas a polares y así convertirlos en linealmente separables



# Ejemplos de funciones de kernel



## Mas de Variables de holgura (Slack variables)

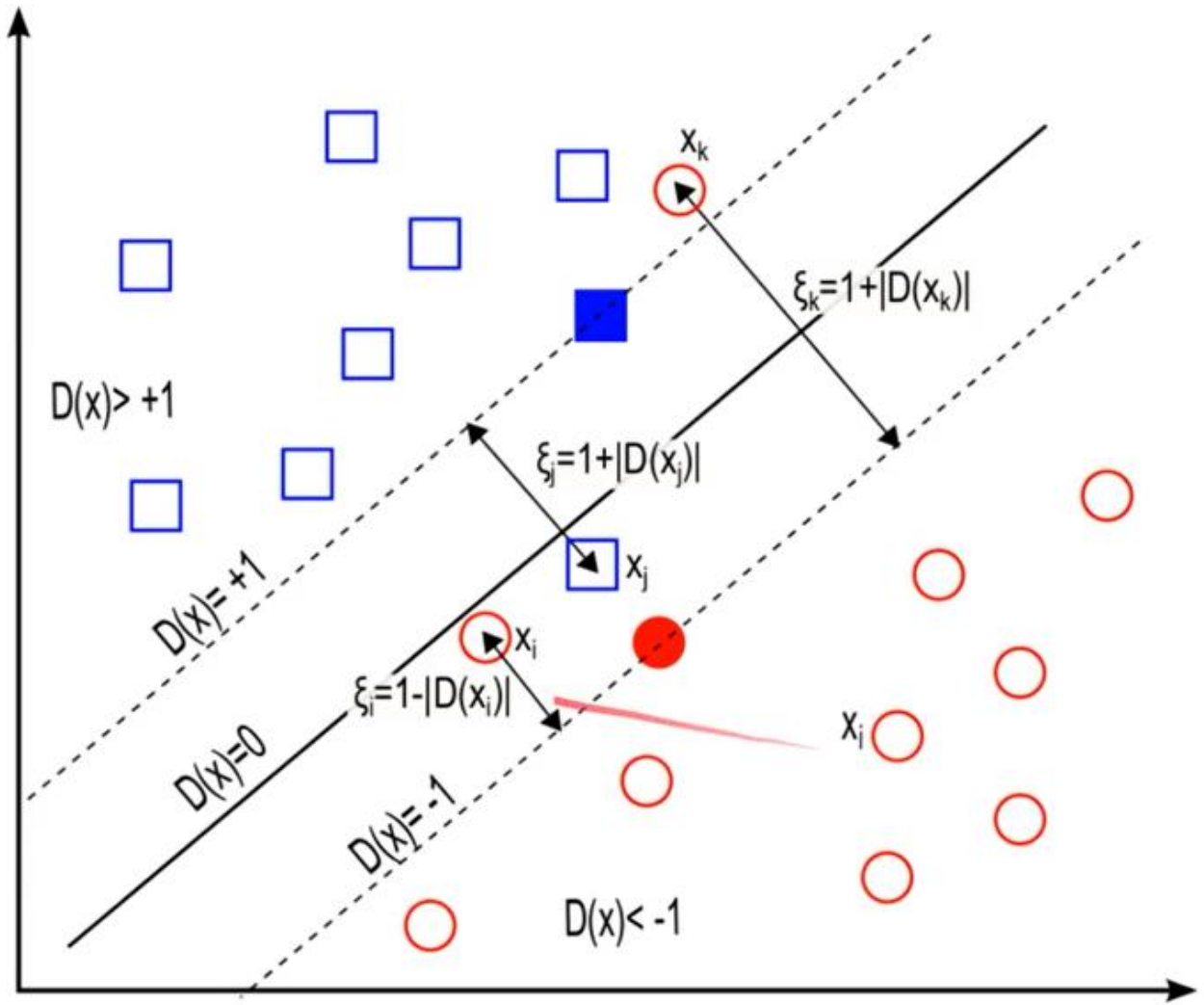
Hasta ahora hemos supuesto que las clases son linealmente separables en un hiperplano. Supongamos que no, o que por algún motivo preferimos crear márgenes más grandes a costa de producir errores de clasificación de aquellas observaciones más cercanas a la frontera de decisión, **las variables de holgura** $\xi_i$ nos permiten controlar el número de ejemplos no separables. Las variables de holgura miden la distancia de cada observación al borde del margen de la clase respectiva.
Añadir variables de holgura modifica el problema y convierte la función de optimizar en:

$$minimizar\ \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i$$

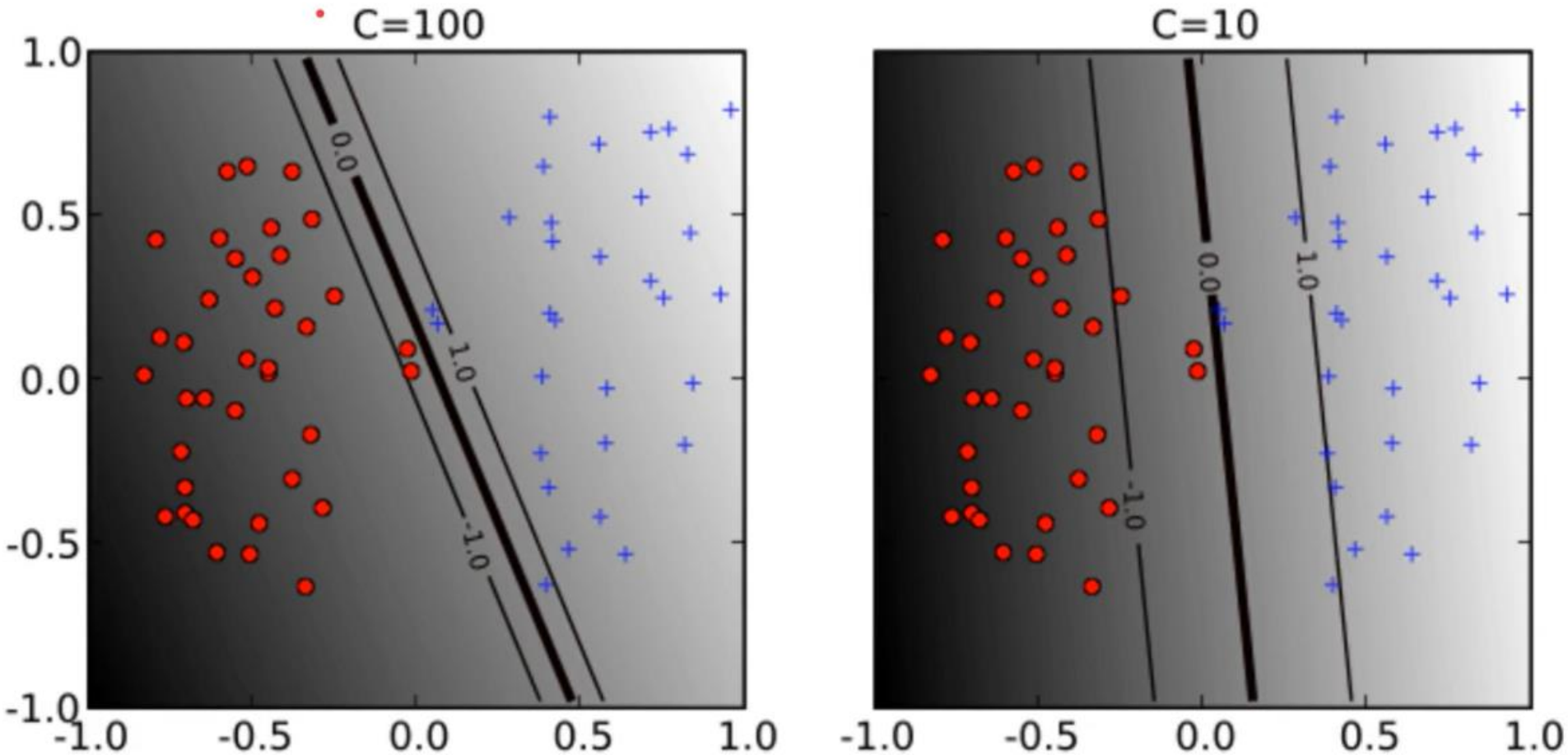
$$sujeto\ a\ y_i(w \cdot x + b) \geq 1 - \xi_i\ \forall x, i = 1, 2, ...n, \xi \geq 0$$

Donde C es un tipo de hiperparámetro del modelo que indica el compromiso entre el sobre ajuste del modelo y el número de observaciones no separables (una medida de cuantos ejemplos podemos permitirnos no clasificar correctamente)

# Variables de holgura (Slack variables)(cont)



## Variables de holgura (Slack variables)(cont)



## Hiperparámetros

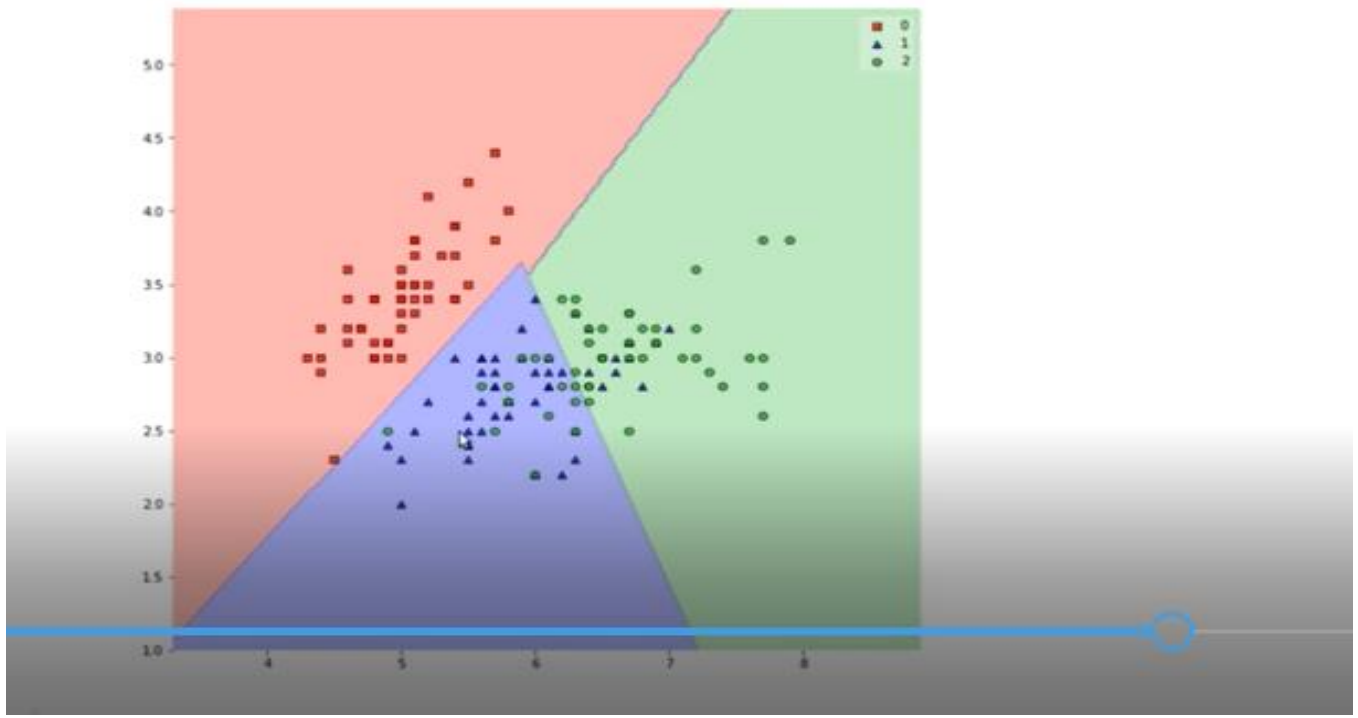
- **C** es el parámetro de Coste (que regula el impacto de las variables de holgura y ayuda a regularizar el modelo.
- **kernel** indica que kernel usar (rbf, radial basis function por defecto). Se puede usar cualquier kernel definido por nosotros, por defecto SVC conoce `rbf`, `poly` (polinomial), `linear` (lineal) o `sigmoid` (sigmoide).
- **class_weight**, nos permite pasar un diccionario de la forma `{clase:peso}` que permite asignar más peso a una clase que a otra. Para problemas con clases no balanceadas, podemos usar el parámetro 'balanced' para que se ajusten los pesos en función del número de casos de cada clase.
- **decision_function_shape** si usar una estrategia de uno contra uno (ovo) o uno contra todos (one versus rest, ovr) en casos de clasificación multiclase.
- **cache_size** es el tamaño (en megabytes) del caché del modelo (cuantos datos puede guardar en memoria y reutilizarlos sin tener que calcularlos). SVMs son computacionalmente complejos asi que si hay mas memoria disponible mejor incrementarl este valor(por ejemplo, a 1000mb o 2000mb)

Mas sobre Kernel

$$Kernel\ lineal : k(x, y) = x^T y + c$$

```
estimador_svm_lineal = SVC(kernel="linear")
estimador_svm_lineal.fit(X, y)

plot_decision_regions(X, y, clf=estimador_svm_lineal);
```



$$Kernel\ polinomial : k(x, y) = (\alpha x^T y + c)^p$$

El kernel polinomial tiene el hiperparámetro degree que indica el grado de expansión polinómica (esto es, el grado de las combinaciones de las variables que queremos tener en cuenta). Por defecto es 3.

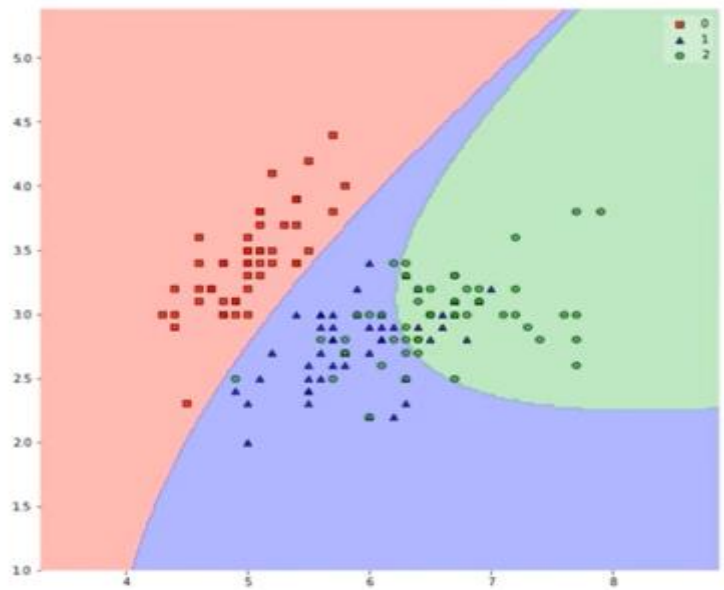
```
estimador_svm_polinomial = SVC(kernel="poly")
estimador_svm_polinomial.fit(X, y)


plot_decision_regions(X, y, clf=estimador_svm_polinomial);
```

Podemos ver como varia la frontera de decisión en función de los grados de expansión. Cuantos más grados más complejo podrá ser el hiperplano. Con grado 1 se convierte en un kernel lineal.

```
estimador_svm_polinomial = SVC(kernel="poly", degree=1).fit(X, y)
plot_decision_regions(X, y, clf=estimador_svm_polinomial);
```

```
estimador_svm_polinomial = SVC(kernel="poly", degree=2).fit(X, y)
plot_decision_regions(X, y, clf=estimador_svm_polinomial);
```




Podemos ver como varia la frontera de decisión en función de los grados de expansión. Cuantos más grados más complejo podrá ser el hiperplano. Con grado 1 se convierte en un kernel lineal.

$$Kernel\ gaussiano(radial) : k(x, y) = exp(-\gamma||x-y^2||)$$

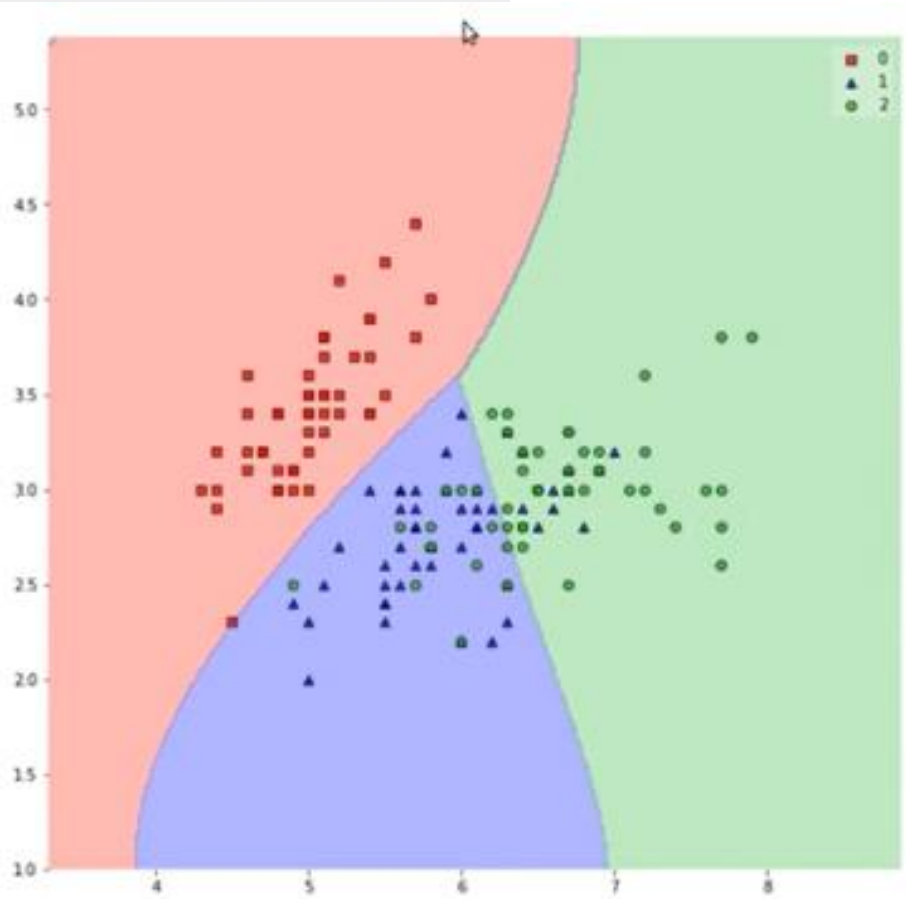
```
estimador_svm_rbf = SVC(kernel="rbf")
estimador_svm_rbf.fit(X, y)

plot_decision_regions(X, y, clf=estimador_svm_rbf);
```

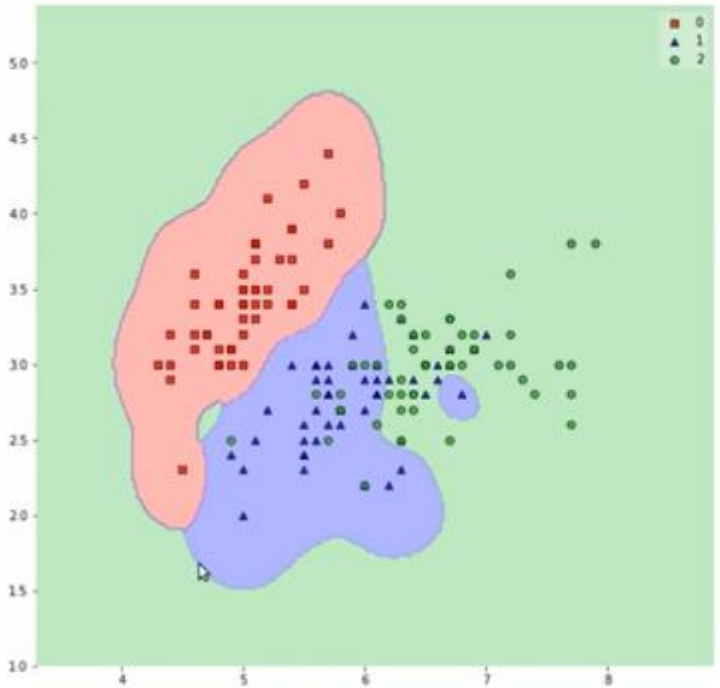
Podemos probar como varia la frontera de decisión en función de gamma:

```
estimador_svm_rbf = SVC(kernel="rbf", gamma=0.1).fit(X, y)
plot_decision_regions(X, y, clf=estimador_svm_rbf);
```

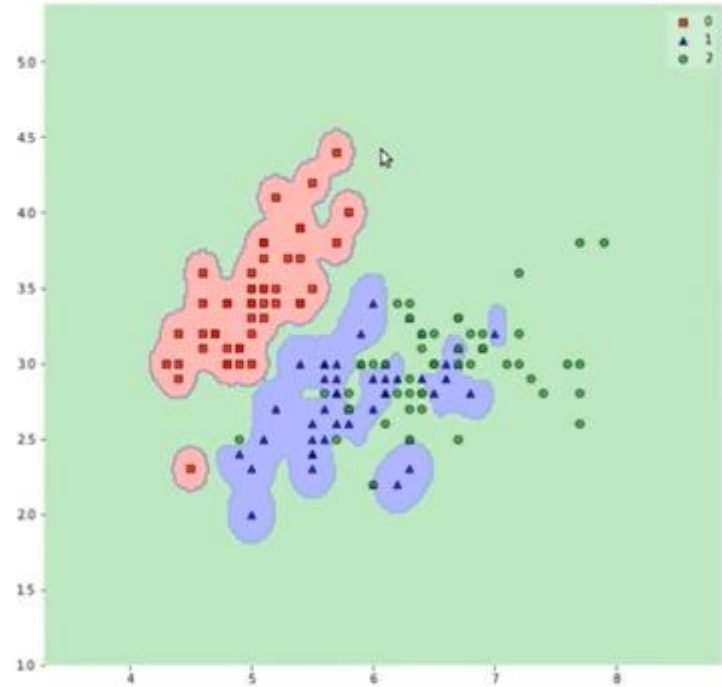
```
estimador_svm_rbf = SVC(kernel="rbf", gamma=10).fit(X, y)
plot_decision_regions(X, y, clf=estimador_svm_rbf);
```



```
estimador_svm_rbf = SVC(kernel="rbf", gamma=10).fit(X, y)
plot_decision_regions(X, y, clf=estimador_svm_rbf);
```




```
estimador_svm_rbf = SVC(kernel="rbf", gamma=100).fit(X, y)
plot_decision_regions(X, y, clf=estimador_svm_rbf);
```



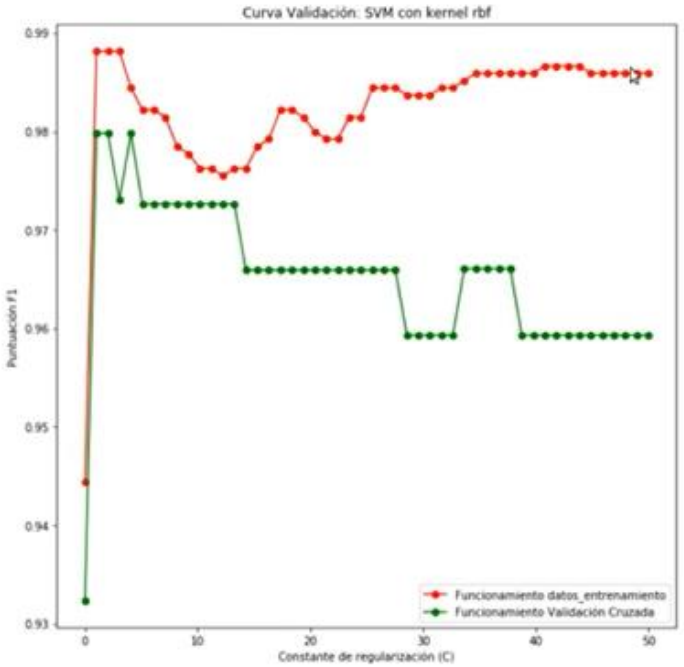
# Parámetro de coste C

El parámetro C nos da una medida de como queremos penalizar al modelo cuando clasifica un ejemplo de forma errónea y es la manera de regularizar los modelos SVM. Valores altos de C permiten controlar la complejidad del modelo (evitando el sobreajuste) a coste de no clasificar bien un porcentaje de los ejemplos en los datos de entrenamiento

```
from sklearn.model_selection import validation_curve
```

```
rango_c = np.linspace(0.01,50,50)
```

```
train_scores, test_scores = validation_curve(estimador_svm, iris_X, iris_y, param_name="C",
                                             param_range=rango_c,cv=10, scoring="f1_weighted")
```




Vemos que conforme aumenta C, el modelo sobreajusta más (ya que mejora su funcionamiento en los datos de entrenamiento pero empeora en los de test).

# Probabilidades ¶

Los modelos SVM no proporcionan probabilidades (por que no hacen inferencia estadística en ese sentido, sino que funcionan de un modo geométrico), por eso por defecto el modelo SVC no proporciona el metodo `predict_proba` que hemos visto en otros estimadores (regresión logística por ejemplo).

Sin embargo, la implementación de sklearn permite pasarle el parámetro `probability=True` que calcula de forma adicional las probabilidades usando escalado de Platt (básicamente, entrena una regresión logística en las distancias al hiperplano computadas por el SVM).

Éste método es computacionalmente complejo, y además tiene ciertos fallos teóricos (por ejemplo, puede haber casos en los que se prediga una clase en un problema de clasificación binaria y que su método `predict_proba` produzca una probabilidad menor que 0.5).

Para aquellos casos que se necesite una forma de puntuar nuevas observaciones pero que dicha puntuacion no tenga que ser una probabilidad es mejor usar directamente el output de la función de decisión con `decision_function`

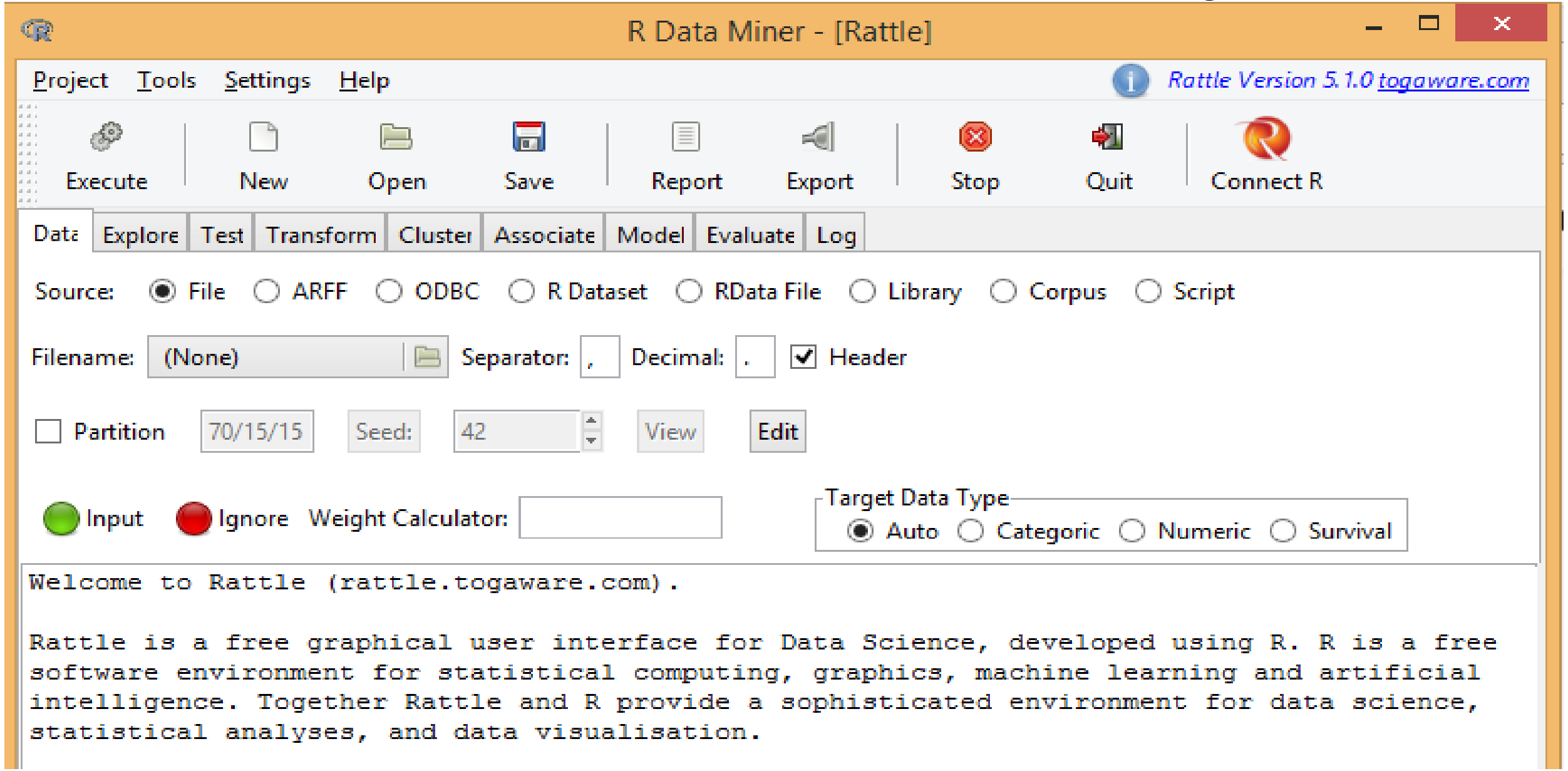

¿GATO O GATA?
kg
g
Sexo
Peso
Peso Corazón
F
2.2
5
F
2.4
8
M
3.1
10
M
2.3
7
M
3.7
11

En R

USANDO EL PAQUETE **Rattle** (R Analytical Tool To Learn Easily)

Instalar desde la consola de R

Interface grafica de Rattle



# Ejemplo 1: IRIS.CSV

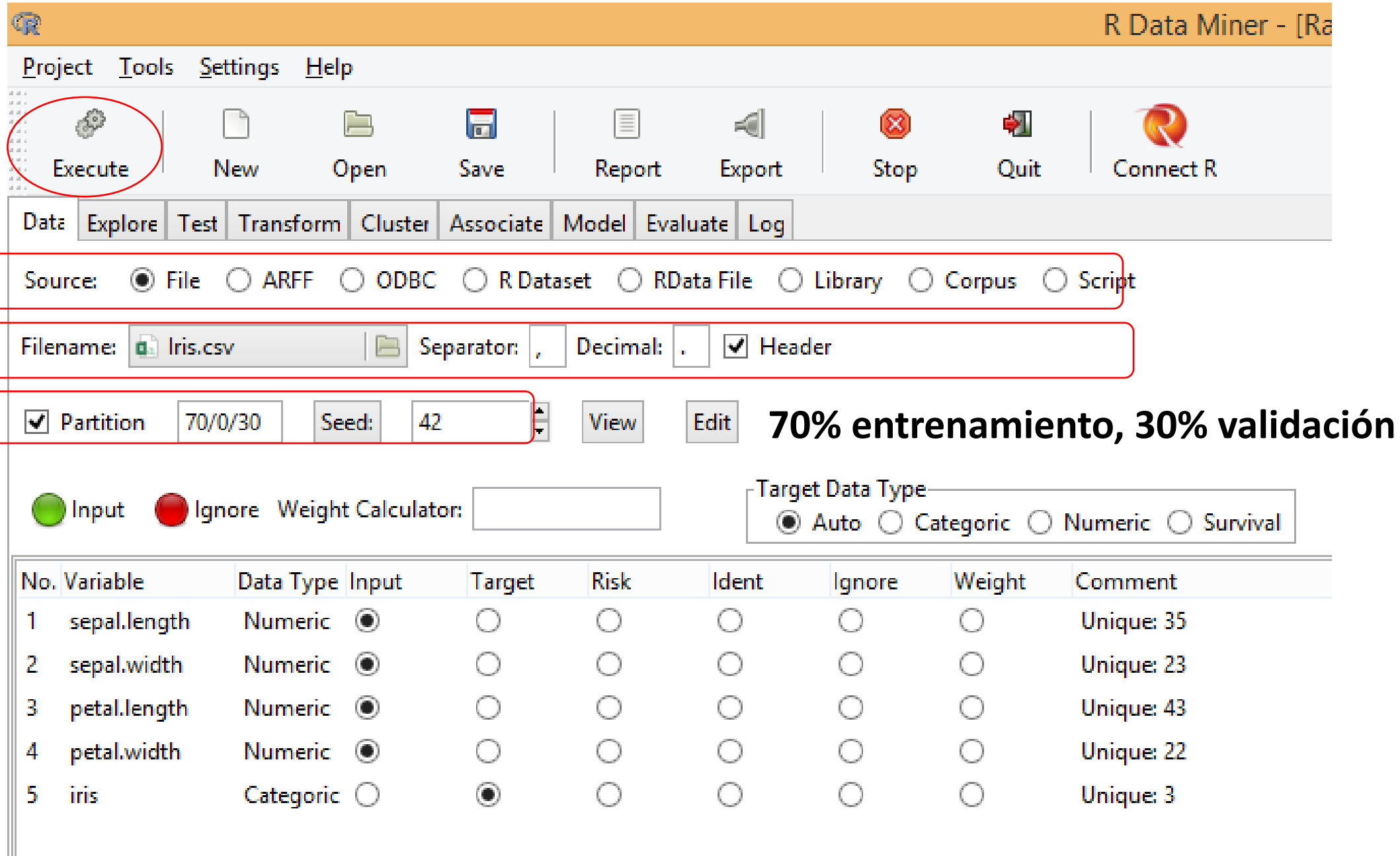
Ejemplo con la tabla de datos IRIS
IRIS Información de variables:
1.sepal largo en cm
2.sepal ancho en cm
3.petal largo en cm
4.petal ancho en cm
5.clase:

- Iris Setosa
- Iris Versicolor
- Iris Virginica

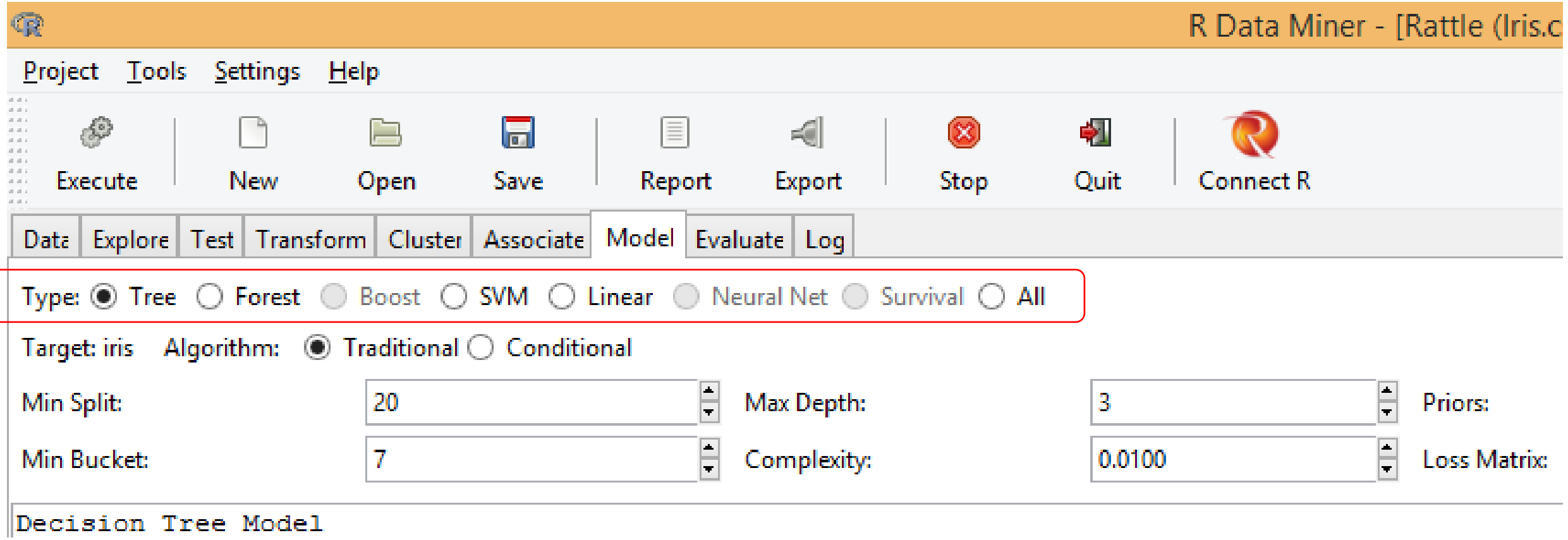

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | s.largo | s.ancho | p.largo | p.ancho | tipo |
| 2 | 5.1 | 3.5 | 1.4 | 0.2 | setosa |
| 3 | 4.9 | 3.0 | 1.4 | 0.2 | setosa |
| 4 | 4.7 | 3.2 | 1.3 | 0.2 | setosa |
| 5 | 4.6 | 3.1 | 1.5 | 0.2 | setosa |
| 6 | 5.0 | 3.6 | 1.4 | 0.2 | setosa |
| 7 | 5.4 | 3.9 | 1.7 | 0.4 | setosa |
| 8 | 4.6 | 3.4 | 1.4 | 0.3 | setosa |
| 9 | 5.0 | 3.4 | 1.5 | 0.2 | setosa |
| 10 | 4.4 | 2.9 | 1.4 | 0.2 | setosa |
| 11 | 4.9 | 3.1 | 1.5 | 0.1 | setosa |
| 12 | 5.4 | 3.7 | 1.5 | 0.2 | setosa |
| 13 | 4.8 | 3.4 | 1.6 | 0.2 | setosa |
| 14 | 4.8 | 3.0 | 1.4 | 0.1 | setosa |
| 15 | 4.3 | 3.0 | 1.1 | 0.1 | setosa |
| 16 | 5.8 | 4.0 | 1.2 | 0.2 | setosa |
| 17 | 5.7 | 4.4 | 1.5 | 0.4 | setosa |
| 18 | 5.4 | 3.9 | 1.3 | 0.4 | setosa |
| 19 | 5.1 | 3.5 | 1.4 | 0.3 | setosa |
| 20 | 5.7 | 3.8 | 1.7 | 0.3 | setosa |
| 21 | 5.1 | 3.8 | 1.5 | 0.3 | setosa |
| 22 | 5.4 | 3.4 | 1.7 | 0.2 | setosa |
| 23 | 5.1 | 3.7 | 1.5 | 0.4 | setosa |
| 24 | 4.6 | 3.6 | 1.0 | 0.2 | setosa |
| 25 | ... | ... | ... | ... | ... |

Variable a predecir “**tipo**”
Se quiere predecir la flor de que tipo es.

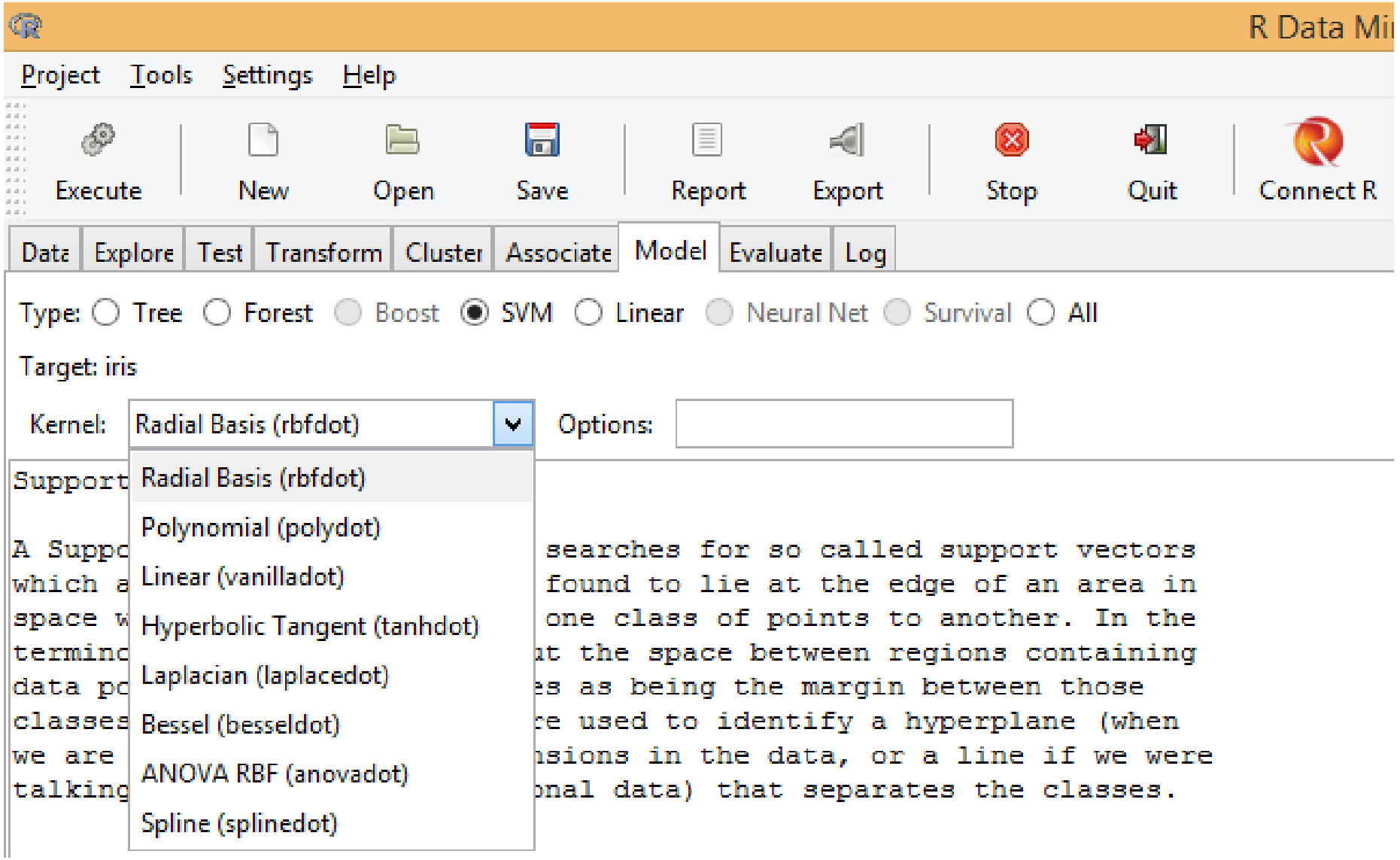
Cargando datos en rattle



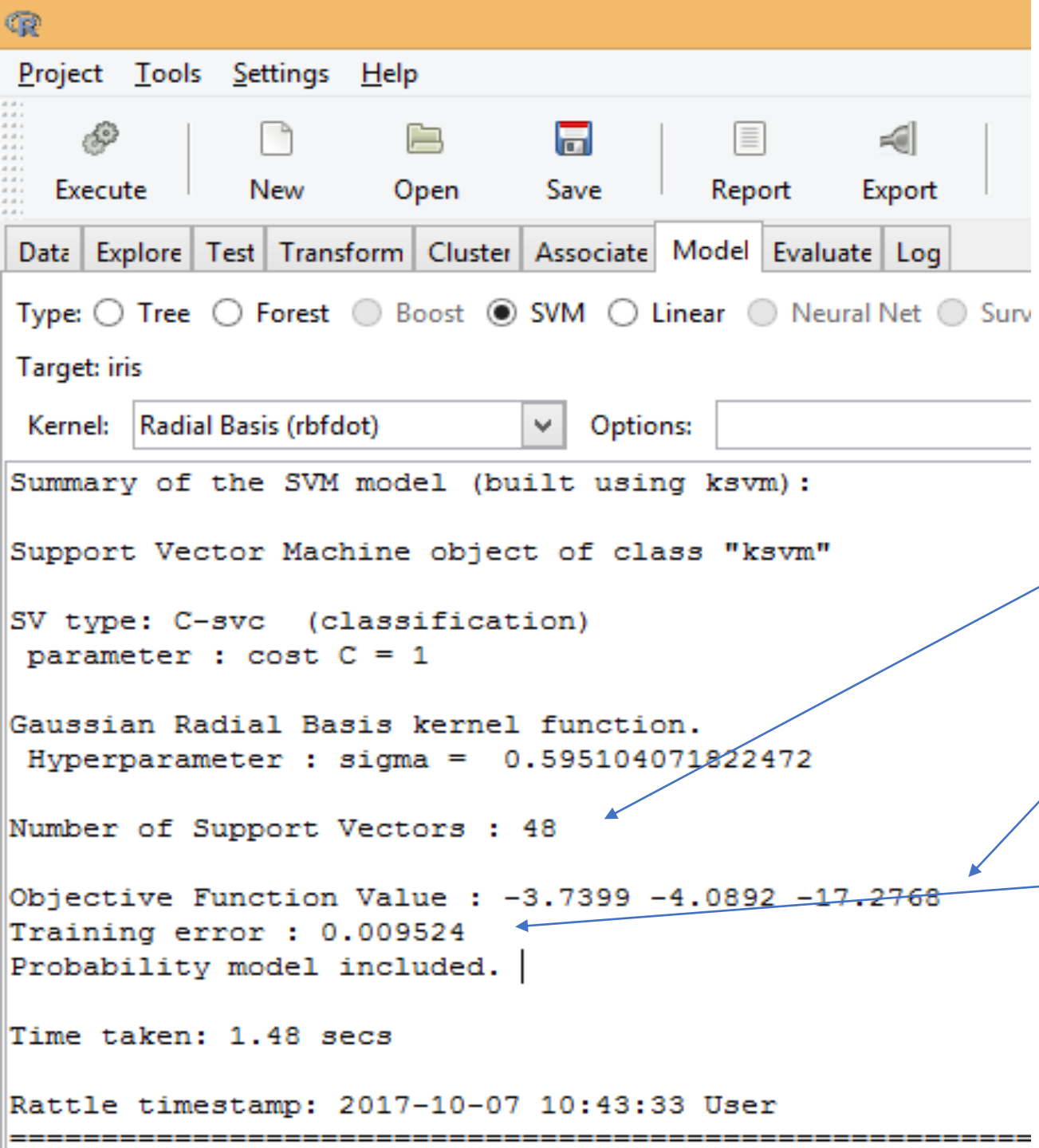
# PESTAÑA MODEL




- ARBOLES, BOSQUES ALEATORIOS, BOOSTING O METODOS DE POTENCIACION, SVM, REGRESION LINEAL LOGISTICA, REDES NEURONALES, SUPERVIVENCIA O TODOS.
- **EN NUESTRO CASOS SELECCIONE SVM.**




Puede probar con todos, por ahora por defecto
Seleccione ejecutar

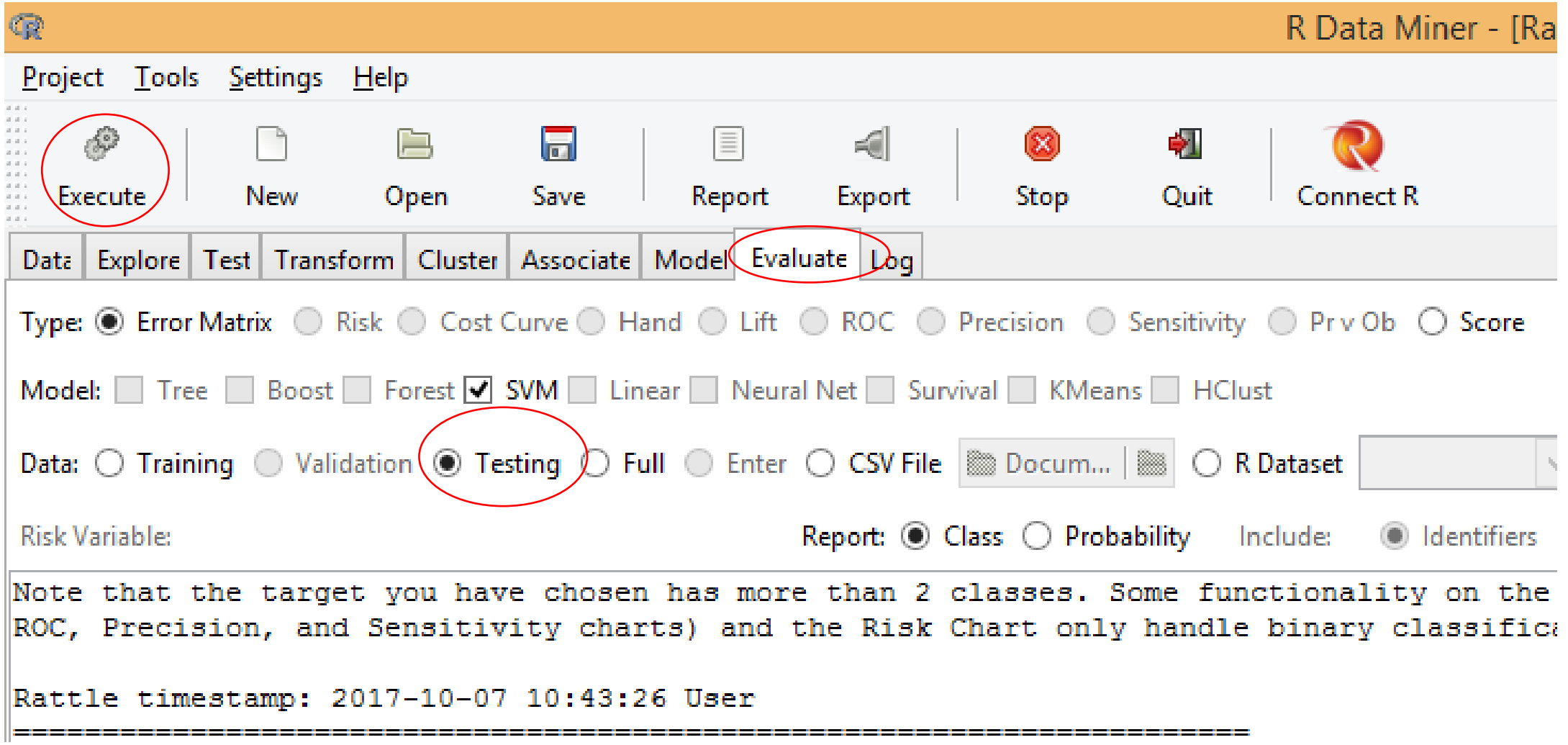



Uso 48 vectores de soporte

Coeficientes del hiperplano

Error en la tabla de entrenamiento, bajo

¿Cómo se puede evaluar que un modelo predictivo es bueno?




Pestaña evaluar

- Ver matriz de errores o de confusiones (ayuda a medir la calidad del modelo predictivo)

# ¿Cómo evaluar la calidad del Modelo Predictivo?

# Matriz de confusión (Matriz de Error)

- La ***Matriz de Confusión*** contiene información acerca de las predicciones realizadas por un ***Método o Sistema de Clasificación***, comparando para el conjunto de individuos en de la tabla de aprendizaje o de testing, la predicción dada versus la clase a la que estos realmente pertenecen.
- La siguiente tabla muestra la matriz de confusión para un clasificador de dos clases:

| | | Predicción | |
|---|---|---|---|
| | | Negativo | Positivo |
| Valor Real | Negativo | a | b |
| | Positivo | c | d |

**Se confecciona con la tabla de testing**

# Matriz de confusión

| | | Predicción | |
|---|---|---|---|
| | | Negativo | Positivo |
| Valor Real | Negativo | a | b |
| | Positivo | c | d |

- La Precisión ***P*** de un modelo de predicción es la proporción del número total de predicciones que son correctas respecto al total. Se determina utilizando la ecuación: ***P = (a+d)/(a+b+c+d)***

- ***Cuidado***, este índice es a veces engañoso y debe ser siempre analizado en la relación a la dimensión de las clases.

# Matriz de confusión para más de 2 clases

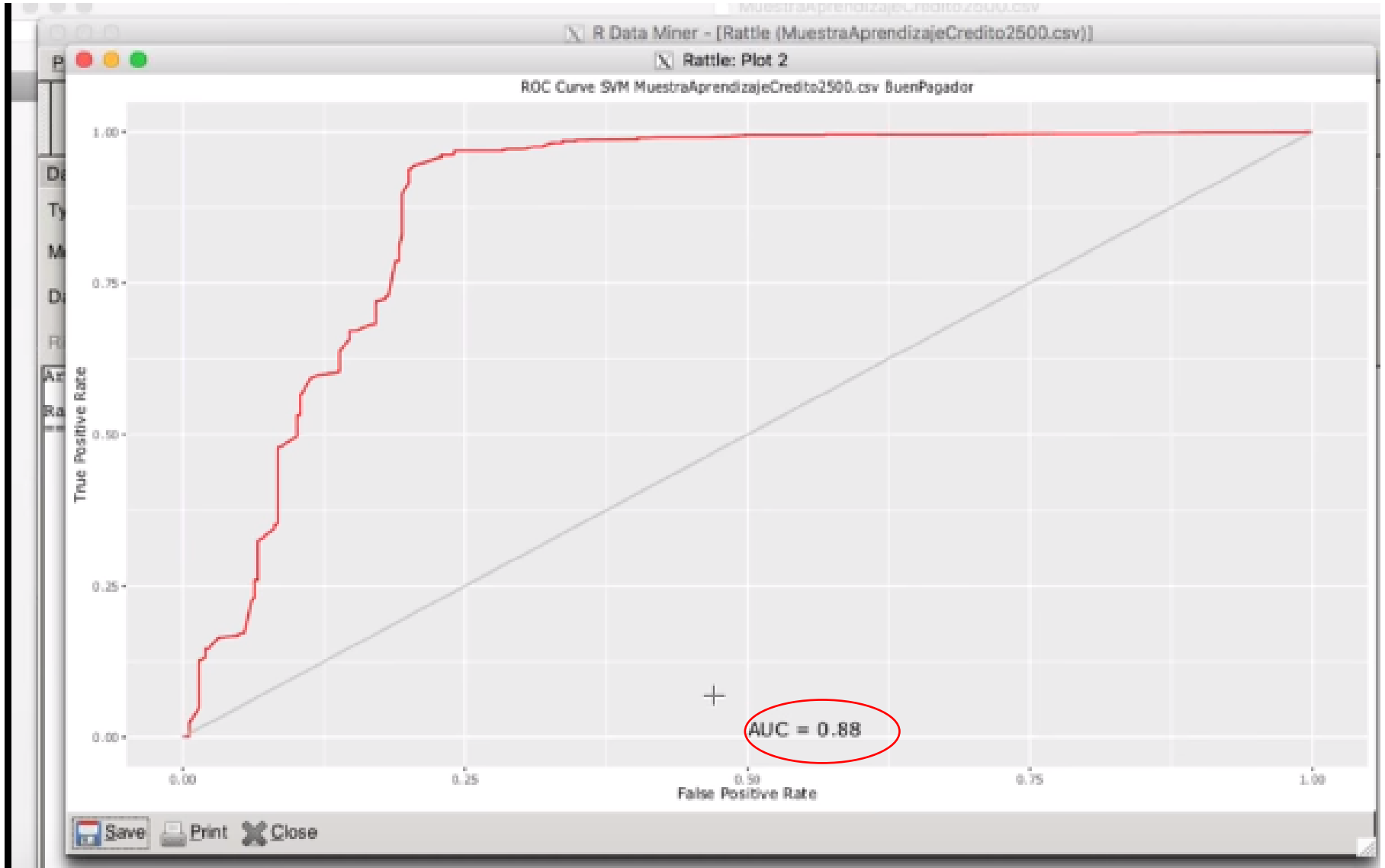
- La Matriz de Confusión puede calcularse en general para un problema con ***p*** clases.

- En la matriz ejemplo que aparece a continuación, de 8 alajuelenses reales, el sistema predijo que 3 eran heredianos y de 6 heredianos predijo que 1 era un limonense y 2 eran alajuelenses. A partir de la matriz se puede ver que el sistema tiene problemas distinguiendo entre alajuelenses y heredianos, pero que puede distinguir razonablemente bien entre limonenses y las otras provincias.

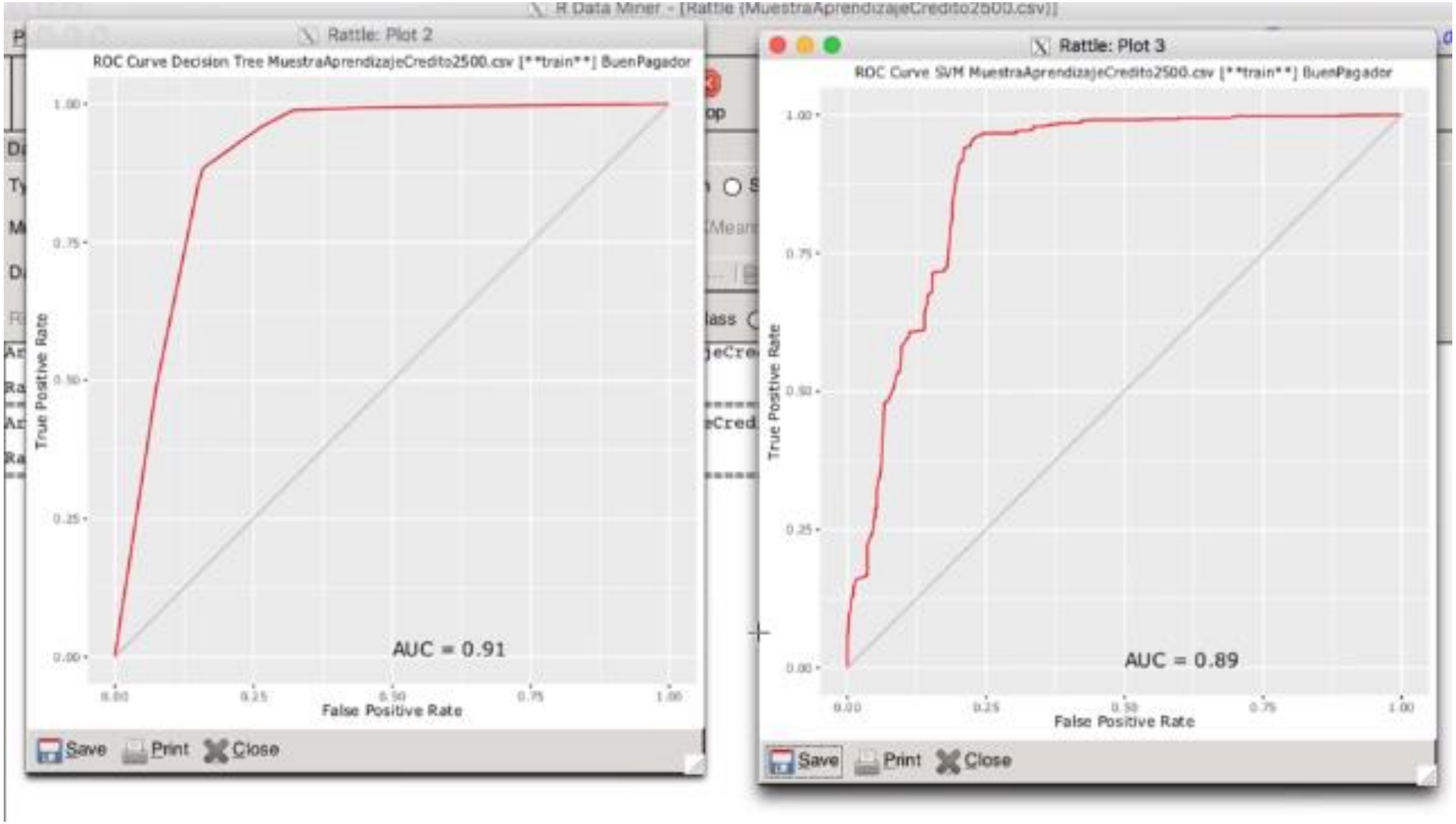
| | | Predicción | | |
|---|---|---|---|---|
| | | alajuelense | herediano | limonense |
| Valor Real | alajuelense | 5 | 3 | 0 |
| | herediano | 2 | 3 | 1 |
| | limonense | 0 | 2 | 11 |

Si es binaria la variable a predecir, se puede usar las curvas ROC

Seleccione curvas ROC y ejecute




88% de efectividad




Comparando árbol con SVM