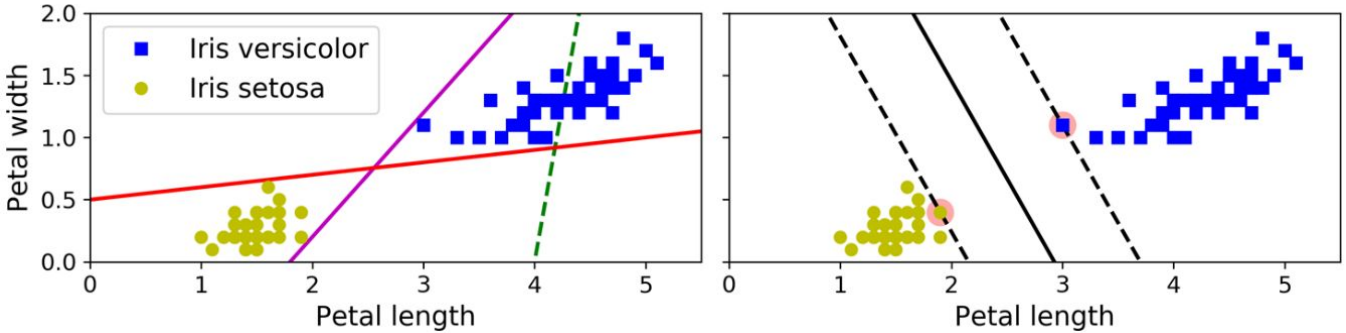
Support Vector Machines (SVM)

UCC UNIVERSIDAD CATÓLICA DE CÓRDOBA JESUITAS

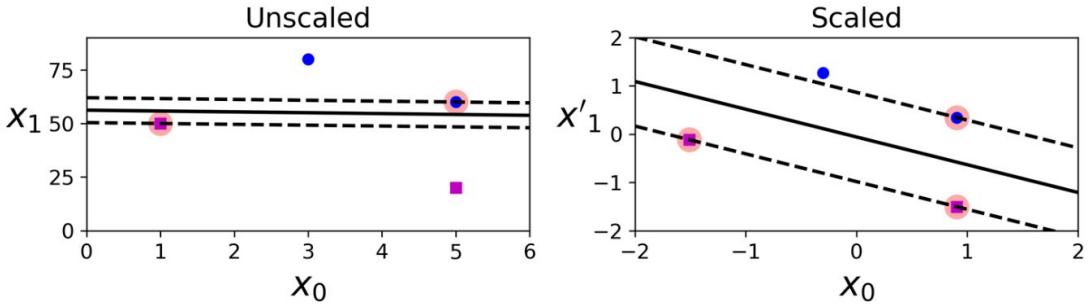
Dr. Francisco Arduh

2023

Clasificación con SVM Lineal

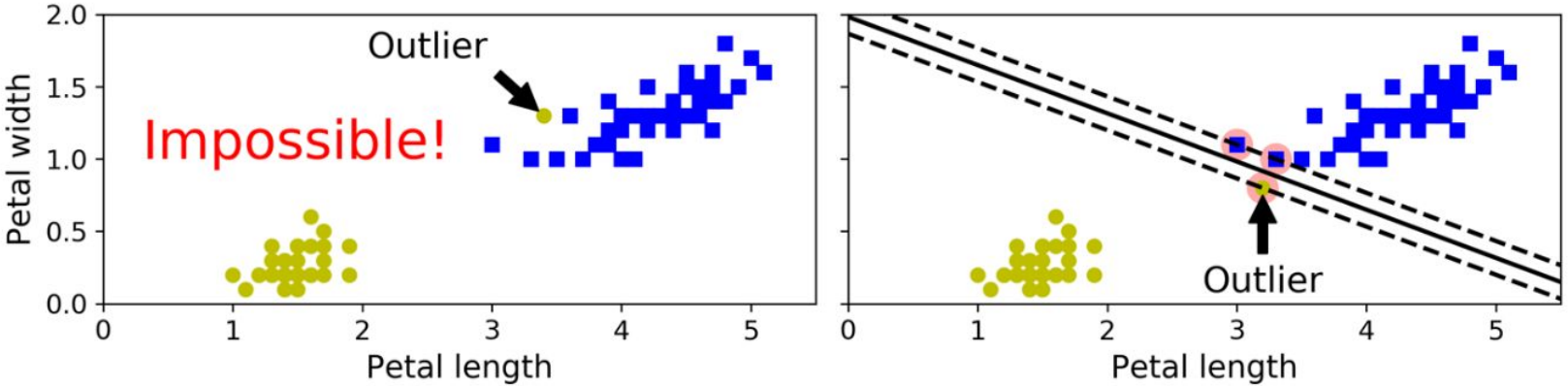


SVM es afectado por la escala:



Clasificación con SVM Lineal: SoftMargin

¿Qué pasaría si hay un dato inusual (outlier)?



Para evitar esto problemas se flexibiliza el modelo permitiendo qué haya una separación con un número muy bajo de violaciones del margen (SoftMargin)

Clasificación con SVM Lineal: Implementación

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris virginica
svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1, loss="hinge")),
])
svm_clf.fit(X, y)
```

Si quisiera hacer una predicción:

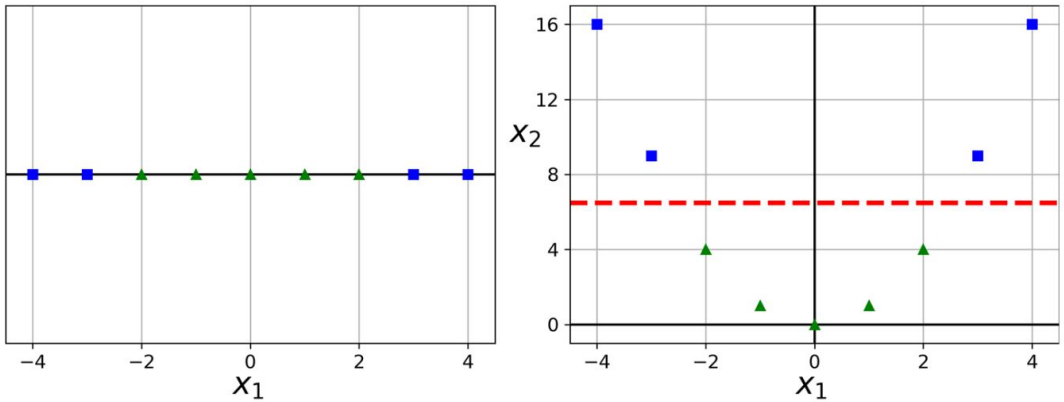
```
>>> svm_clf.predict([[5.5, 1.7]])
array([1.])
```

Otras formas utilizar el mismo clasificador es:

SVC(kernel="linear", C=1) o **SGDClassifier(loss="hinge", alpha=1/(m*C))**

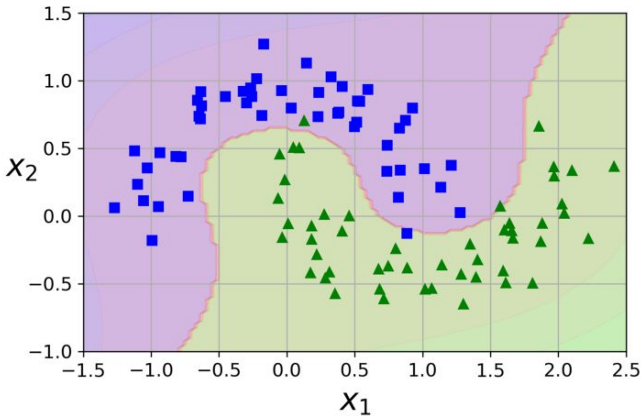
Clasificación No Lineal con SVM

Al igual que cuando se utilizaba regresión lineal para dataset no lineal, se puede realizar clasificación de datos que son no linealmente separables.



Clasificación No Lineal con SVM: Implementación

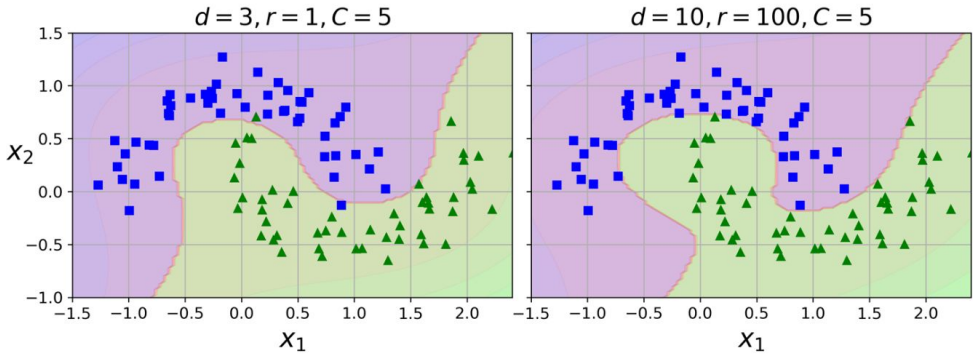
```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

X, y = make_moons(n_samples=100, noise=0.15)
polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge"))
    ])

polynomial_svm_clf.fit(X, y)
```



Kernel Polinomial

En vez de utilizar **PolynomialFeatures,** SVM permite utilizar el "kernel trick" que permite obtener el mismo resultado, pero sin añadir características extra a nuestro dataset.

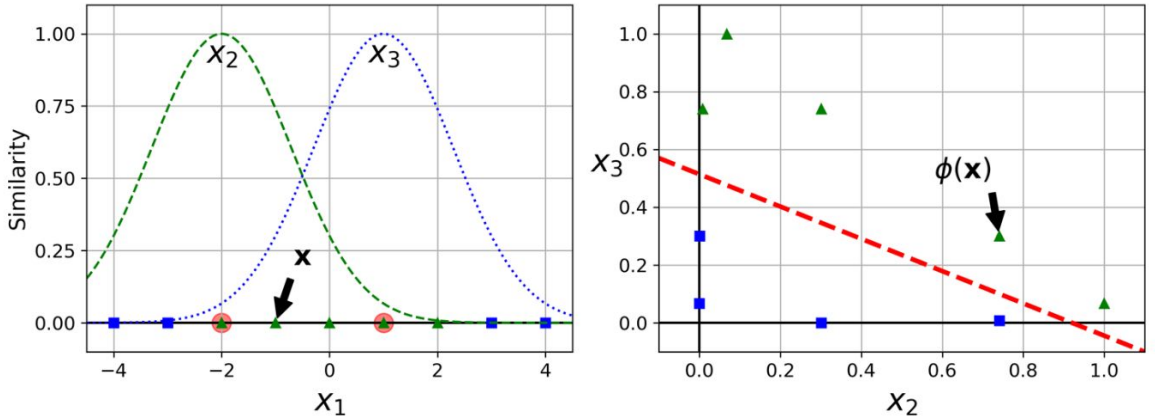
```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```



Características de similitud

Otra técnica para abordar problemas no lineales es añadir características usando una función de similitud, que mide qué tanto una característica se parece a un punto de referencia.
Una función para medir similitud es Gaussian Radial Basis Function (RBF)

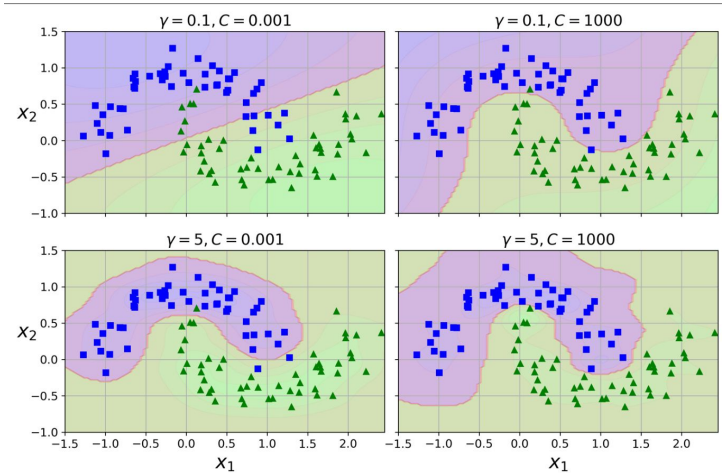
$$\phi_\gamma(\mathbf{x}, \ell) = \exp\left(-\gamma \| \mathbf{x} - \ell \|^2\right)$$



Kernel RBF Gaussiano

Añadir características de similitud puede ser muy costoso computacionalmente, pero en SVM se puede utilizar otra vez el “kernel trick”.

```
rbf_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ])
rbf_kernel_svm_clf.fit(X, y)
```



¿Qué kernel utilizar?

Depende de la información qué se tenga de antemano, pero en caso no tenerla se puede empezar:

1. Kernel lineal, es más barato computacionalmente.
2. Kernel RBF, funciona bastante bien en el resto de los casos.
3. Si se puede, experimentar con otros kernels.
4. Utilizar validación cruzada.
5. Realizar una optimización de hiperparámetros.

Kernels disponibles

Linear: $K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^\top \mathbf{b}$

Polynomial: $K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^\top \mathbf{b} + r)^d$

Gaussian RBF: $K(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma \| \mathbf{a} - \mathbf{b} \|^2\right)$

Sigmoid: $K(\mathbf{a}, \mathbf{b}) = \tanh\left(\gamma \mathbf{a}^\top \mathbf{b} + r\right)$

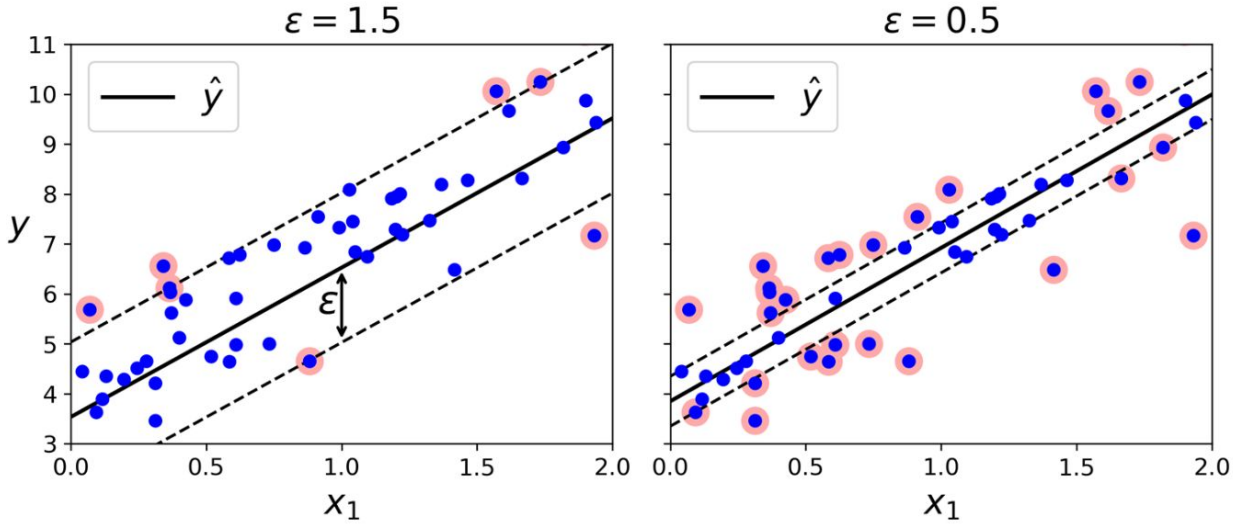
Complejidad computacional

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
`LinearSVC`	$O(m \times n)$	No	Yes	No
`SGDClassifier`	$O(m \times n)$	Yes	Yes	No
`SVC`	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

Regresión con SVM

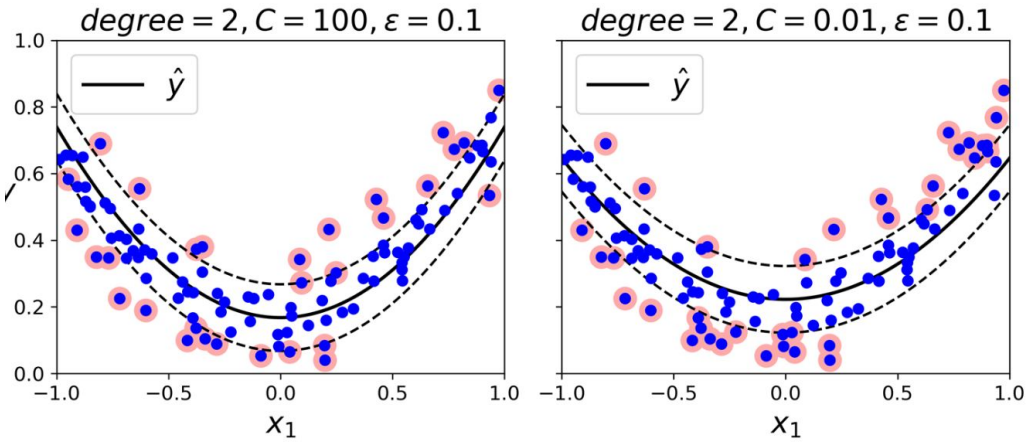
Se modifica el algoritmo para que en vez de buscar la línea de separación con menor violaciones de margen, se busque qué la mayor cantidad de instancias caigan en la línea minimizando las violaciones de margen (fuera del mismo)

```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```



Regresión con SVM “kernelizado”

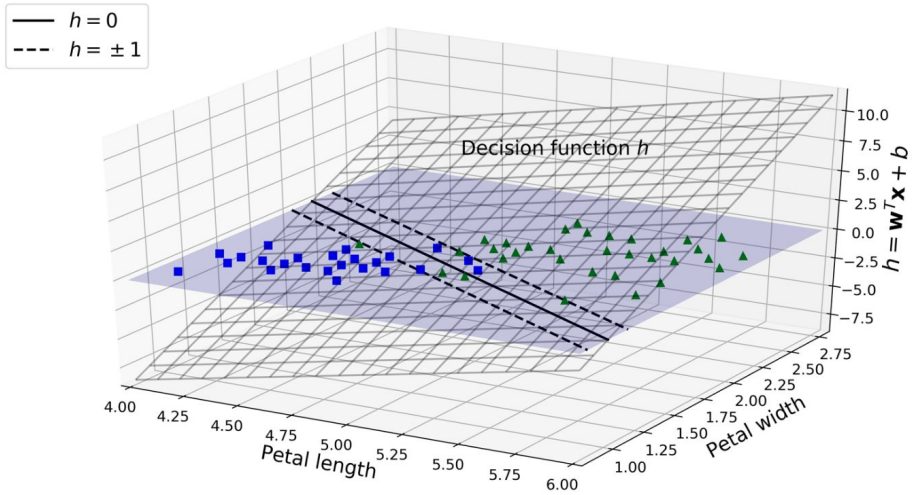
```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```



Función de decisión

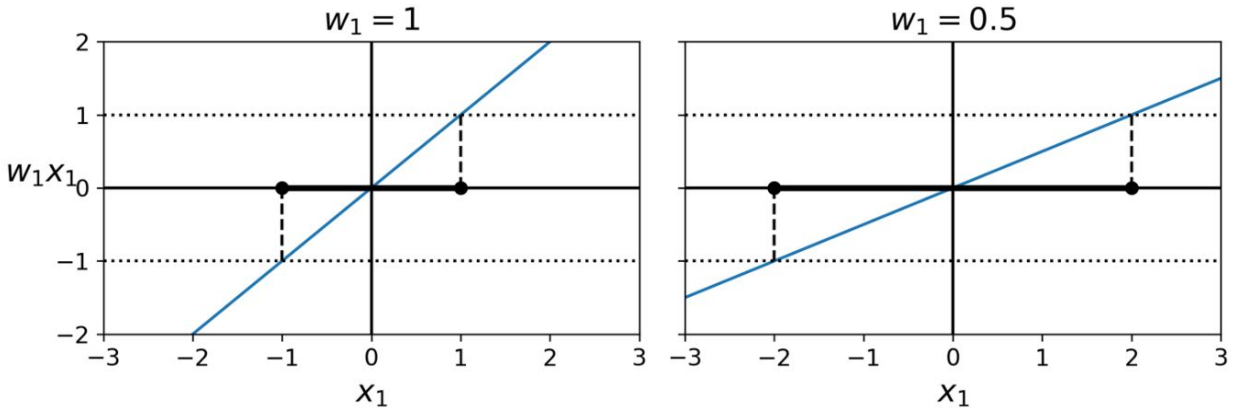
Predicción del clasificador SVM lineal:

$$\hat{y} = \begin{cases} 0 \text{ if } \mathbf{w}^T\mathbf{x} + b < 0, \\ 1 \text{ if } \mathbf{w}^T\mathbf{x} + b \geq 0 \end{cases}$$



Entrenando el modelo

Si tomo la norma del vector de peso **||w||** (pendiente) y lo divido en dos el margen se duplica.



En nuestra función objetivo queremos minimizar **||w||** para hacer los márgenes lo más grandes posibles.

Entrenando el modelo

En el caso de hard margin, no queremos que ninguna instancia tenga un valor inferior a 1.

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^\intercal\mathbf{w}$$

$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^\intercal\mathbf{x}^{(i)} + b\right) \geq 1 \quad \text{for } i = 1, 2, \cdots, m$$

, con $t = \begin{cases} 1 & \text{si } y = 1 \\ -1 & \text{si } y = 0 \end{cases}$

En el caso del soft margin se define una función $\boldsymbol{\zeta} \geq 0$, qué mide el grado de violación de margen.

$$\underset{\mathbf{w}, b, \zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^\intercal\mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)}$$

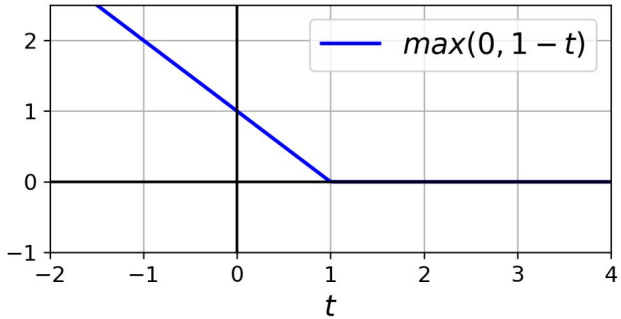
$$\text{subject to} \quad t^{(i)}\left(\mathbf{w}^\intercal\mathbf{x}^{(i)} + b\right) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

Estas ecuaciones se resuelven utilizando programación cuadrática “QP” (problemas de optimización cuadráticos sujetos a condiciones)

https://towardsdatascience.com/support-vector-machines-soft-margin-formulation-and-kernel-trick-4c9729dc8efe

Online SVM

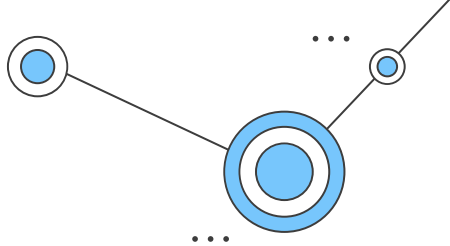
Implementado en scikit-learn utilizando SGDClassifier, con función objetivo Hinge Loss:



Este resultado es obtenido de aplicar QP a

$$J(\mathbf{w}, b) = \frac{1}{2}\mathbf{w}^\top\mathbf{w} \quad + \quad C\sum_{i=1}^{m} max\left(0, 1 - t^{(i)}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b\right)\right)$$

Este algoritmo converge más lento que los métodos basados puramente en QP



¿Dudas?