Clase5 IMA539

Alejandro Ferreira Vergara

March 31, 2023

1 Regresión Logística y SVM

1.1 ¿Qué aprenderemos hoy?

- **Regresión Logística**
- **Clasificación Multiclase**
- **Predicciones**
- **Regularización**
- **Suport Vector Machine (SVM) kernel lineal**

2 Regresión Logística como modelo probabilístico

- **Algoritmo de clasificación**
- **Modelo lineal**
- **Puede extenderse a problemas de clasificación multiclase**

Evento positivo: Es un evento (o suceso) que nos interesa predecir. Por ejemplo, la probabilidad de que un paciente tenga una determinada enfermedad; podemos pensar en el **evento positivo como la etiqueta de clase** $y = 1$.

Tasa de probabilidades (Odds ratio): Las probabilidades a favor de un evento particular (por ejemplo, el evento positivo). La podemos escribir como $\frac{p}{1-p}$, donde p representa la probabilidad del evento positivo.

Ahora, definamos la función *logit* como **logaritmo natural de la tasa de probabilidades (log-odds)**:

$$logit(p) = ln\left(\frac{p}{1-p}\right) = log\left(\frac{p}{1-p}\right) \quad (convencin)$$

- **¿Cuál es el dominio y el recorrido de *logit*?**

Ahora, definamos:

$$logit(p(y = 1|x)) = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \sum_{i=0}^{m} w_ix_i = w^tx$$

- $p(y = 1|x)$ es la **probabilidad condicional de que una determinada muestra** x **pertenezca a la clase 1** (dadas sus características x_i).

Para modelar probabilidades, tomaremos la inversa de la función *logit*, denominada **función sigmoidea logística**, o simplemente **función Sigmoide**:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

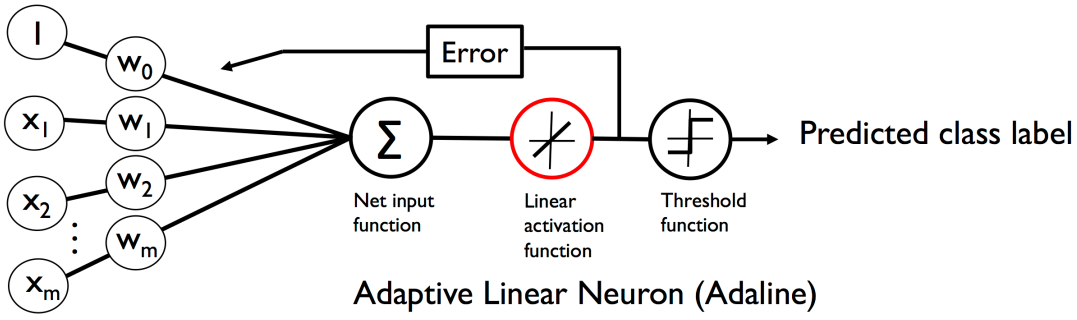
$$z = w^t x = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m$$

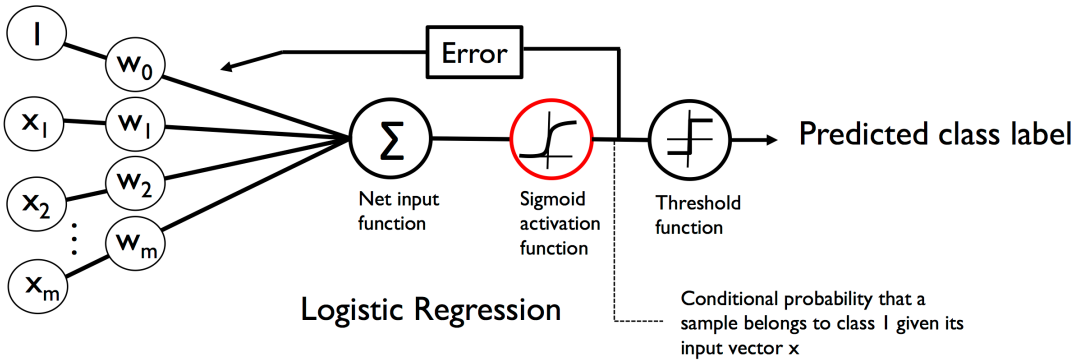
[ ]:
```
import matplotlib.pyplot as plt
import numpy as np

def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

z = np.arange(-7, 7, 0.1)
phi_z = sigmoid(z)

plt.plot(z, phi_z)
plt.axvline(0.0, color='k')
plt.ylim(-0.1, 1.1)
plt.xlabel('z')
plt.ylabel('$\phi (z)$')

plt.yticks([0.0, 0.5, 1.0])
ax = plt.gca()
ax.yaxis.grid(True)

plt.tight_layout()
#plt.savefig('images/03_02.png', dpi=300)
plt.show()
```

2.1 Modelo predictivo

[1]:
```
from IPython.display import Image

Image(filename=r'clase5/5_1.png', width=600)
```

[1]:





$$\hat{y} = \begin{cases} 1 & \text{si } \phi(z) \geq 0.5 \\ 0 & \text{en caso contrario} \end{cases}$$

$$\hat{y} = \begin{cases} 1 & \text{si } z \geq 0.0 \\ 0 & \text{en caso contrario} \end{cases}$$

2.2 Aprendizaje de los pesos

Definamos la **Función de coste logístico:**

$$L(w) = P(y|x;w) = \prod_{i=1}^{n} P\left(y^{(i)}|x^{(i)};w\right) = \prod_{i=1}^{n} \left(\phi\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1-\phi\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

$$l(w) = \log L(w) = \sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1-y^{(i)}\right) \log\left(1-\phi\left(z^{(i)}\right)\right)\right]$$

Aplicar *log*() permite reducir el potencial de "desbordamiento numérico", que puede ocurrir si las probabilidades son muy pequeñas. En segundo lugar, convertimos el producto de los factores en una suma de factores, lo que facilita la obtención de la derivada de esta función.

Ahora podemos utilizar Descenso de Gradiente (Ascenso de Gradiente) para maximizar esta función de verosimilitud. Alternativamente, reescribamos la verosimilitud como una función de coste J que puede ser minimizada.

$$J(w) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right] = -\sum_{i=1}^{n} \left[y^{(i)} \log\left(\phi\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \phi\left(z^{(i)}\right)\right) \right]$$

Cálculo del Gradiente de la función de coste (o de pérdida) $J(w)$, pensando sólo en una muestra de entrenamiento:

$$\frac{\partial}{\partial w_j} J(w) = \left(y \frac{1}{\phi(z)} - (1 - y) \frac{1}{1 - \phi(z)} \right) \frac{\partial \phi(z)}{\partial w_j}$$

Cálculo de la derivada parcial de la función sigmoide (punto clave en la formulación):

$$\frac{\partial \phi(z)}{\partial z} = \frac{\partial}{\partial z} \frac{1}{1 + e^{-z}} = \frac{1}{\left(1 + e^{-z}\right)^2} e^{-z} = \frac{1}{1 + e^{-z}} \left(1 - \frac{1}{1 + e^{-z}} \right) = \phi(z) \left(1 - \phi(z)\right)$$

Ahora, sustituyendo:

$$\frac{\partial}{\partial w_j} J(w) = \left(y \frac{1}{\phi(z)} - (1 - y) \frac{1}{1 - \phi(z)} \right) \frac{\partial \phi(z)}{\partial w_j}$$

$$= \left(y \frac{1}{\phi(z)} - (1 - y) \frac{1}{1 - \phi(z)} \right) \phi(z) \left(1 - \phi(z)\right) \frac{\partial}{\partial w_j} z$$

$$= \left(y \left(1 - \phi(z)\right) - (1 - y) \phi(z) \right) x_j$$

$$= \left(y - \phi(z) \right) x_j$$

Dado que la maximización de la log-verosimilitud es igual a la minimización de la función de coste J que definimos anteriormente y dado que actualizamos todos los pesos simultáneamente observando todas las muestras de entrenamiento, podemos escribir la regla de actualización del descenso del gradiente como:

$$w := w + \Delta w, \ \Delta w = -\eta \nabla J(w)$$

$$w_j := w_j + \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi\left(z^{(i)}\right) \right) x_j^{(i)}$$

2.3 Implementación Regresión Logística con Python

[ ]:
```
class LogisticRegressionGD(object):
    """Clasificador regresión logística mediante descenso de gradiente.

    Parámetros
    ------------
    eta : float
      Learning rate (entre 0.0 y 1.0)
    n_iter : int
      Cantidad de épocas de entrenamiento.
    random_state : int
      Semilla para generar pesos aleatorios.


    Atributos
    -----------
    w_ : 1d-array
      Vector de pesos al término del entrenamiento.
    cost_ : list
      Valor de la función de costo logístico en cada época.

    """
    def __init__(self, eta=0.05, n_iter=100, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Entrenamiento.

        Parametros
        ----------
        X : {array-like}, shape = [n_samples, n_features]
          Vector de entrenamiento, donde n_samples es el número de muestras y
          n_features es el número de características.
        y : array-like, shape = [n_samples]
          Valor de salida (etiquetas).

        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.cost_ = []
```

```
        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()

            cost = -y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output)))
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        """Calcular entrana neta, z"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, z):
        """Calcular activación sigmoidea logística"""
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

    def predict(self, X):
        """Etiqueta de clase después del paso unitario"""
        return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
        # esto es equivalente a:
        # return np.where(self.net_input(X) >= 0.0, 1, 0)
```

[ ]:
```
from sklearn import datasets

iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target

print('Etiquetas de Clase:', np.unique(y))
```

[ ]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=1, stratify=y)

print('Cantidad de etiquetas en y:', np.bincount(y))
print('Cantidad de etiquetas en y_train:', np.bincount(y_train))
print('Cantidad de etiquetas en y_test:', np.bincount(y_test))
```

[ ]:
```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
sc.fit(X_train)
```

```
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
```

[ ]:
```
%matplotlib inline
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt


def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],y=X[y == cl, 1],alpha=0.8,c=colors[idx],
                    marker=markers[idx],label=cl,edgecolor='black')

    if test_idx:
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0],X_test[:, 1],edgecolor='black',
                    alpha=1.0,linewidth=1,marker='o',s=100, label='test set')
```

[ ]:
```
X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]
y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]

lrgd = LogisticRegressionGD(eta=0.05, n_iter=1000, random_state=1)
lrgd.fit(X_train_01_subset,y_train_01_subset)

plot_decision_regions(X=X_train_01_subset, y=y_train_01_subset,classifier=lrgd)

plt.xlabel('largo pétalo [estandarizado]')
plt.ylabel('ancho pétalo [estandarizado]')
plt.legend(loc='upper left')

plt.tight_layout()
#plt.savefig('images/03_05.png', dpi=300)
```

```
plt.show()
```

2.4 Regresión Logística con Scikit-learn

[ ]:
```
from sklearn.linear_model import LogisticRegression

X_combined_std = np.vstack((X_train_std, X_test_std))
y_combined = np.hstack((y_train, y_test))

lr = LogisticRegression(C=100.0, random_state=1)
lr.fit(X_train_std, y_train)

plot_decision_regions(X_combined_std, y_combined,classifier=lr,␣
 ↪test_idx=range(105, 150))

plt.xlabel('largo pétalo [estandarizado]')
plt.ylabel('ancho pétalo [estandarizado]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_06.png', dpi=300)
plt.show()
```

[ ]:
```
print(lr.predict_proba(X_test_std[:3, :]))
print('-------------------------------------------')

print(lr.predict_proba(X_test_std[:3, :]).sum(axis=1))
print('-------------------------------------------')

print(lr.predict_proba(X_test_std[:3, :]).argmax(axis=1))
print('-------------------------------------------')

print(lr.predict(X_test_std[:3, :]))
```

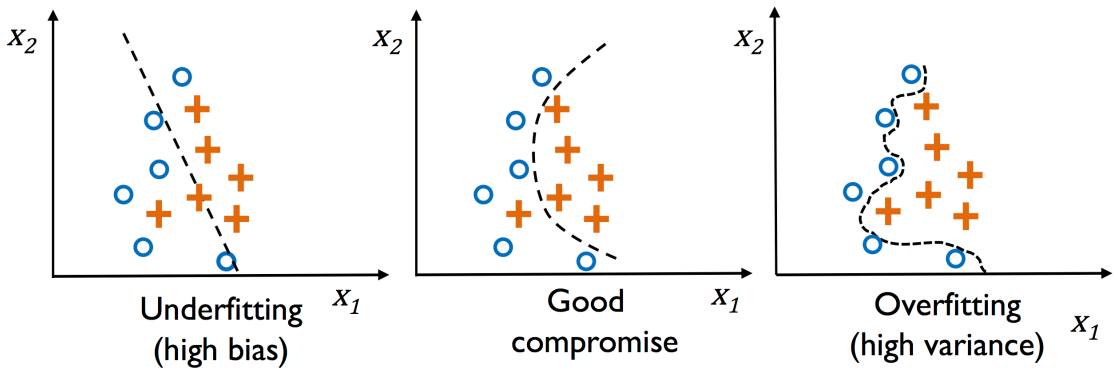
3 Cómo abordar el sobreajuste mediante la regularización

El **Overfitting** o sobreajuste es un problema común en aprendizaje automático. Un modelo con overfitting funciona bien con los datos de entrenamiento, pero no generaliza bien (la predicción) en los datos no vistos (datos de testeo). Si un modelo sufre de sobreajuste, también decimos que el modelo tiene una **alta varianza**, que puede ser causada por tener demasiados parámetros que conducen a un modelo que es demasiado complejo dado los datos subyacentes.

Del mismo modo, un modelo también podría sufrir de **Underfitting (alto sesgo)**, lo que significa que el modelo no es lo suficientemente complejo como para capturar bien el patrón en los datos de entrenamiento y, por lo tanto, también sufre de bajo rendimiento en los datos no vistos.

[2]:
```
Image(filename=r'clase5/5_2.png', width=700)
```

[2]:



Una forma de encontrar un buen equilibrio entre sesgo y varianza es **ajustar la complejidad del modelo mediante la regularización**. La regularización es un método muy útil para manejar la colinealidad (alta correlación entre características), filtrar el ruido de los datos y, finalmente, evitar el sobreajuste.

El concepto de regularización consiste en introducir información adicional (sesgo) para penalizar los valores extremos de los parámetros (pesos). La forma más común de regularización es la llamada **regularización L2** (a veces también llamada contracción L2 o decaimiento del peso).

$$\frac{\lambda}{2}\|w\|^2 = \frac{\lambda}{2}\sum_{i=1}^{m} w_j^2$$

- **NOTA: Re-escalar las características también ayuda a regularizar un modelo.**
- **Regularizar la función de costo:**

$$J(w) = \sum_{i=1}^{n}\left[-y^{(i)}\log\left(\phi\left(z^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-\phi\left(z^{(i)}\right)\right)\right] + \frac{\lambda}{2}\|w\|^2$$

A través del parámetro de regularización λ, podemos controlar lo bien que nos ajustamos a los datos de entrenamiento manteniendo los pesos en valores pequeños. Al aumentar el valor de λ, aumentamos la fuerza de regularización.

[ ]:
```
weights, params = [], []
for c in np.arange(-5, 5):
    lr = LogisticRegression(C=10.**c, random_state=1)
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[1])
    params.append(10.**c)

weights = np.array(weights)
plt.plot(params, weights[:, 0],
         label='petal length')
plt.plot(params, weights[:, 1], linestyle='--',
```

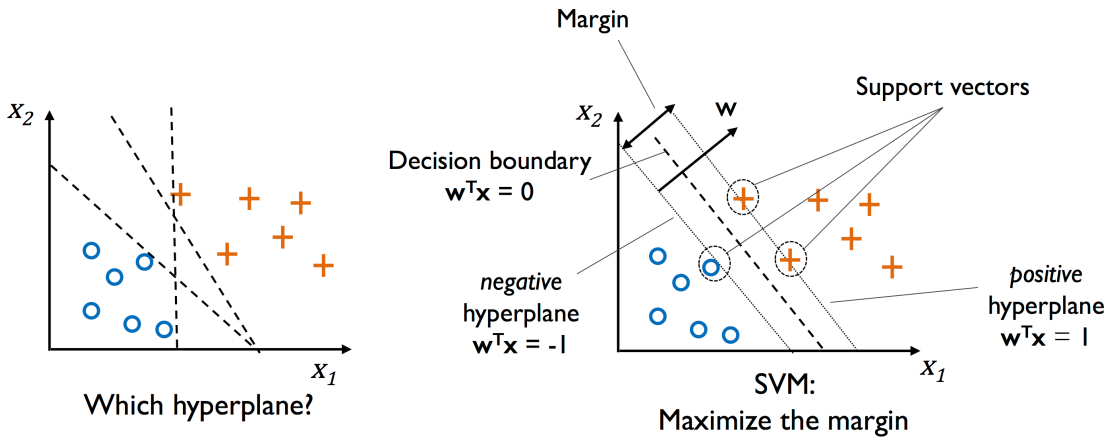
```
         label='petal width')
plt.ylabel('weight coefficient')
plt.xlabel('C')
plt.legend(loc='upper left')
plt.xscale('log')
#plt.savefig('images/03_08.png', dpi=300)
plt.show()
```

4 Support Vector Machine

Con SVM nuestro objetivo de optimización es maximizar el margen. El margen lo podemos definir como la **distancia** entre el hiperplano de separación (frontera de decisión) y las muestras de entrenamiento más cercanas a este hiperplano, que son los llamados **Vectores de Soporte**.

[3]:
```
Image(filename=r'clase5/5_3.png', width=700)
```

[3]:



Para tener una idea de la maximización de los márgenes, veamos con más detalle los **hiperplanos positivos y negativos** que son paralelos a la frontera de decisión:

$$w_0 + w^T x_{pos} = 1 \qquad (1)$$

$$w_0 + w^T x_{neg} = -1 \qquad (2)$$

Restando (1) y (2):

$$w^T \left(x_{pos} - x_{neg}\right) = 2$$

Podemos normalizar dividiendo por la norma del vector w, $\|w\| = \sqrt{\sum_{j=1}^{m} w_j^2}$:

$$\frac{w^T\left(x_{pos}-x_{neg}\right)}{\|w\|}=\frac{2}{\|w\|}$$

El lado izquierdo de la ecuación anterior puede interpretarse como la distancia entre el hiperplano positivo y el negativo, y es el llamado margen que queremos maximizar.

La **función objetivo de SVM se convierte en la maximización de** $\frac{2}{\|w\|}$ **bajo la restricción de que las muestras sean clasificadas correctamente**:

$$w_0+w^Tx^{(i)}\geq 1 \quad \text{si } y^{(i)}=1$$

$$w_0+w^Tx^{(i)}\leq -1 \quad \text{si } y^{(i)}=-1$$

$$\text{Para} \quad i=1....N$$

Estas dos ecuaciones dicen básicamente que **todas las muestras negativas deben caer en un lado del hiperplano negativo, mientras que todas las muestras positivas deben caer detrás del hiperplano positivo**:

$$y^{(i)}\left(w_0+w^Tx^{(i)}\right)\geq 1 \quad \forall i$$

En la práctica, sin embargo, es más fácil minimizar $\frac{1}{2}\|w\|^2$**.**

Inclusión variable de holgura (clasificación de margen suave):

$$w_0+w^Tx^{(i)}\geq 1-\xi^{(i)} \quad \text{si } y^{(i)}=1$$

$$w_0+w^Tx^{(i)}\leq -1+\xi^{(i)} \quad \text{si } y^{(i)}=-1$$

$$\text{Para} i=1....N$$

Formulación final:

$$min \quad \frac{1}{2}\|w\|^2+C\sum_i \xi^{(i)}$$

$$s.a.$$

$$w_0+w^Tx^{(i)}\geq 1-\xi^{(i)} \quad \text{si } y^{(i)}=1$$

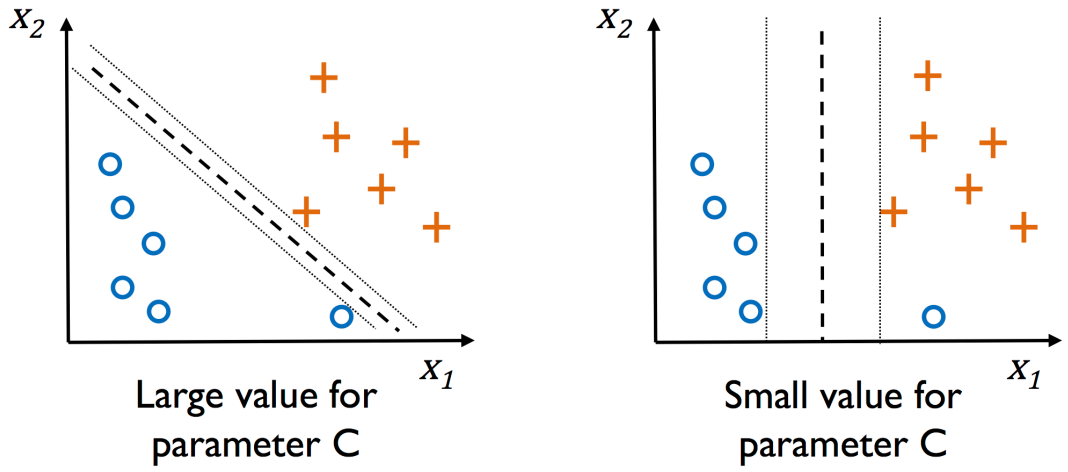
$$w_0 + w^T x^{(i)} \leq -1 + \xi^{(i)} \quad \text{si } y^{(i)} = -1$$

$$\text{Para} i = 1....N$$

A través del parámetro C, podemos controlar la penalización por clasificación errónea. Los valores grandes de C corresponden a grandes penalizaciones por el error, mientras que seremos menos estrictos con los errores de clasificación si elegimos valores más pequeños para C.

[4]:
```
Image(filename=r'clase5/5_4.png', width=600)
```

[4]:



[ ]:
```
from sklearn.svm import SVC

svm = SVC(kernel='linear', C=100., random_state=1)
svm.fit(X_train_std, y_train)

plot_decision_regions(X_combined_std,y_combined,classifier=svm,test_idx=range(105,␣
↪150))

plt.xlabel('largo pétalo [estandarizado]')
plt.ylabel('ancho pétalo [estandarizado]')
plt.legend(loc='upper left')
plt.tight_layout()
#plt.savefig('images/03_11.png', dpi=300)
plt.show()
```

[ ]:
```

```