Tema 2: Apredizaje Supervisado

Clasificación

Perceptrón

Prof. Wladimir Rodriguez

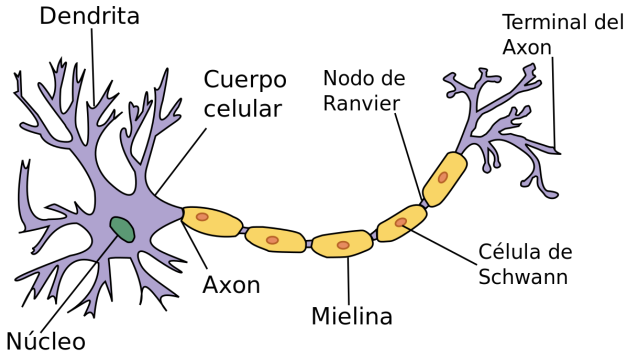
wladimir@ula.ve

Departamento de Computación

Algoritmos de Clasificación: El Perceptrón

Neuronas Artificiales

Las neuronas biológicas son células nerviosas interconectadas en el cerebro que están involucradas en el procesamiento y transmisión de señales químicas y eléctricas. Las dendritas y el cuerpo celular de la neurona reciben señales de entrada excitatorias e inhibitorias de las neuronas vecinas; el cuerpo celular las combina e integra y emite señales de salida. El axón transporta esas señales a los terminales axónicos, que se encargan de distribuir información a un nuevo conjunto de neuronas



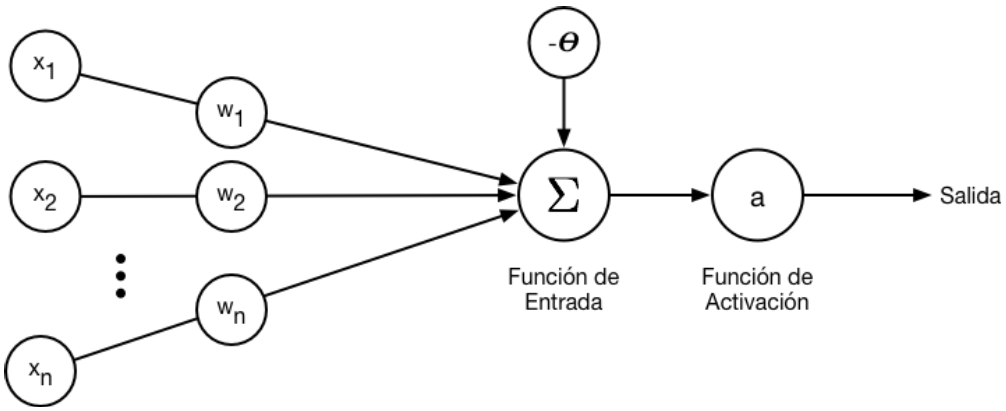
Neurona de McCulloch y Pitts

El modelo de neuronas de McCulloch y Pitts, publicado en 1943, fue el primer intento de formalizar matemáticamente el comportamiento de una neurona y de estudiar sus implicaciones en su capacidad de computar y procesar la información.

McCullock y Pitts describieron una célula nerviosa como una simple puerta lógica con salidas binarias; Múltiples señales llegan a las dendritas, se integran entonces en el cuerpo celular y, si la señal acumulada excede un cierto umbral, se genera una señal de salida que será transmitida por el axón.

La neurona de McCulloch-Pitts es una unidad de cálculo que intenta modelar el comportamiento de una neurona "natural", similares a las que constituyen del cerebro humano. Ella es la unidad esencial con la cual se construye una red neuronal artificial.

El resultado del cálculo en una neurona consiste en realizar una suma ponderada de las entradas, seguida de la aplicación de una función no lineal, como se ilustra en la siguiente figura



Esto se expresa matemáticamente como:

$$salida = a(z)$$

siendo:

- $z = w_1 x_1 + \cdots + w_n x_n - \boldsymbol{\Theta}$ es la suma ponderada de las entradas.
- x_i es el valor de la i-ésima entrada.
- w_i es el peso de la conexión entre la i-ésima entrada y la neurona.
- $\boldsymbol{\Theta}$ es el valor umbral
- $salida$ es la salida de la neurona.
- a es la función no lineal conocida como función de activación.

La función de activación que se usa es:

$$a = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$$

El Perceptrón de Rosenblatt

En 1957 Frank Rosenblatt publicó el primer concepto de la regla de aprendizaje del perceptrón basado en el modelo de neuronas MCP (F. Rosenblatt, *The Perceptron, a Perceiving and Recognizing Automaton*, Cornell Aeronautical Laboratory). Con su regla del perceptron, Rosenblatt propuso un algoritmo que aprendería automáticamente los coeficientes de peso óptimos que luego se multiplican con los atributos de entrada para tomar la decisión de si una neurona se dispara o no. En el contexto del aprendizaje supervisado y la clasificación, tal algoritmo podría utilizarse para predecir si una muestra pertenecía a una clase u otra.

Formalmente se puede plantear este problema como una tarea de clasificación binaria donde se hace referencia a dos clases, la clase 1 (clase positiva) y la clase -1 (clase negativa). Se puede entonces definir una función de activación $\phi(z)$ la cual toma una combinación lineal de un vector de entrada ***x*** y un correspondiente vector de pesos ***w***, donde z es la llamada entrada de red $(z = w_1x_1 + \ldots + w_nx_n)$:

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}, \; x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

Si la activación de una observación particular $x^{(i)}$ es mayor que un umbral definido $\boldsymbol{\theta}$, se predice clase 1 de lo contrario se predice clase -1, en el Perceptron, la función de activación $\phi(\cdot)$ es una función escalón unitario, que a veces también se denomina función escalón de Heaviside:

$$\phi(z) = \begin{cases} 1 & z \geq \boldsymbol{\theta} \\ -1 & z < \boldsymbol{\theta} \end{cases}$$

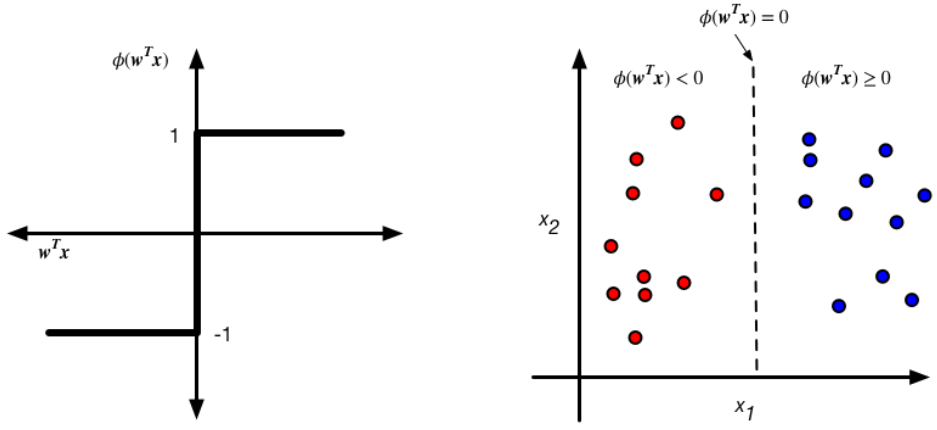
Para simplificar, podemos traer el umbral $\boldsymbol{\theta}$ al lado izquierdo de la ecuación y definir un peso-cero como $w_0 = -\boldsymbol{\theta}$ y $x_0 = 1$, asi se puede escribir z en una forma más compacta $z = w_0x_0 + w_1x_1 + \ldots + w_nx_n$ y

$$\phi(z) = \begin{cases} 1 & z \geq 0 \\ -1 & z < 0 \end{cases}$$

El calculo de la suma de productos de los valores ***x*** y ***w*** se puede abreviar usando el producto escalar:

$$z = w_0x_0 + w_1x_1 + \ldots + w_nx_n = \sum_{j=0}^{n} x_jw_j = w^Tx$$

La siguiente figura ilustra cómo la entrada $z = w^Tx$ es convertida a una salida binaria (-1 o 1) por la función de activación del perceptrón y cómo se puede utilizar para discriminar entre dos clases linealmente separables:



Algoritmo del Perceptrón de Rosenblatt

La regla inicial de perceptrón de Rosenblatt es bastante simple y puede ser resumida por los siguientes pasos:

1. Inicialice los pesos a *0* o a números aleatorios pequeños.
2. Para cada muestra de entrenamiento $x^{(i)}$ realice los siguientes pasos:
    A. Calcule el valor de salida $\hat{y}$.
    B. Actualizar los pesos y el sesgo.

Regla de actualización de los pesos y el sesgo

$$w_j = w_j + \Delta w_j$$

Los valores de actualización se calculan de la siguiente manera:

$$\Delta w_i = \eta(y^{(i)} - \hat{\mathrm{y}}^{(i)})x_j^{(i)}$$

Donde η es la tasa de aprendizaje (una constante entre 0.0 y 1.0), $y^{(i)}$ es la etiqueta de la clase verdadera de la i-ésima muestra de entrenamiento, $\hat{\mathrm{y}}^{(i)}$ es la etiqueta de la clase predicha.

Si la predicción es correcta, los pesos no cambian:

$$\Delta w_j = \eta(-1 - -1)x_j^{(i)} = 0$$

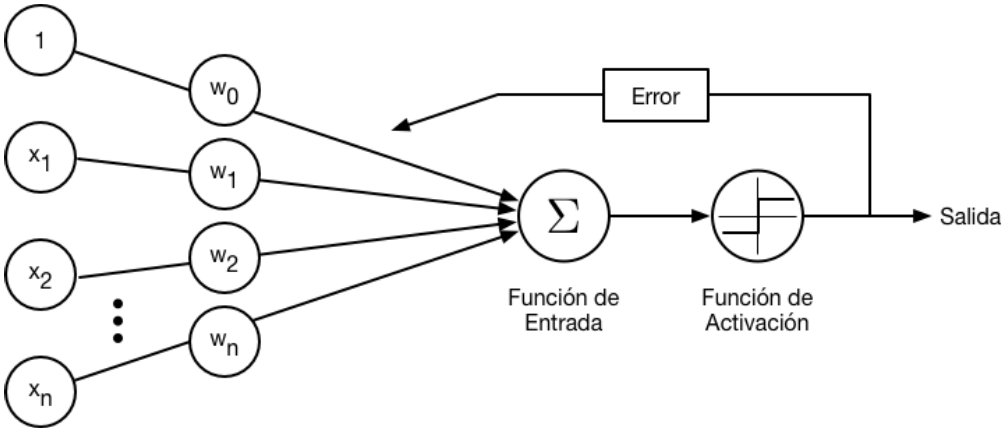
$$\Delta w_j = \eta(1 - 1)x_j^{(i)} = 0$$

Si la predcción es incorrecta, los pesos se ajustan hacia la clase positiva o negativa:

$$\Delta w_j = \eta(1 - -1)x_j^{(i)} = \eta(2)x_j^{(i)}$$

$$\Delta w_j = \eta(-1 - 1)x_j^{(i)} = \eta(-2)x_j^{(i)}$$

La siguiente figura ilustra el concepto general del Perceptrón:



Convergencia del algoritmo

Convergencia garantizada si

- Las dos clases son linealmente separables
- La tasa de aprendizaje es lo suficientemente pequeña

Si las clases no pueden ser separadas:

- Establezca un número máximo de pasadas sobre el conjunto de datos de entrenamiento (épocas)
- Establecer un umbral para el número de errores de clasificación a tolerar
- De lo contrario, nunca dejará de actualizar los pesos (no convergerá)

Implementación del algoritmo de aprendizaje del Perceptrón en Python

In [1]:

```python
import numpy as np


class Perceptron(object):
    ''' Clasificador Perceptron

    Parámetros
    ----------
    eta : float
      Taza de aprendizaje (entre 0.0 y 1.0)
    num_iteraciones : int
      Pasadas sobre el conjunto de datos de entrenamiento
    random_state : int
      Semilla para el generador de números aleatorios, para
      la inicialización de los pesos

    Atributos
    ---------
    w_ : arreglo de 1D
      Pesos después del entrenamiento

    '''

    def __init__(self, eta=0.01, num_iteraciones=40, random_state=8):
        self.eta = eta
        self.num_iteraciones = num_iteraciones
        self.random_state = random_state

    def fit(self, X, y):
        ''' Ajustar la data de entrenamiento

        Parámetros
        ----------
        X : arreglo, forma = [n_ejemplos, n_atributos]
          Vectores de entrenamineto, donde n_ejemplos es el números
          de ejemplos y n_atributos es el números de atributos
        y : arreglo, forma = [n_ejemplos]
          Valores objetivo

        Retorna
        -------
        self : objeto

        '''
        gen_a = np.random.RandomState(self.random_state)
        self.w_ = gen_a.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.errores_ = []

        for _ in range(self.num_iteraciones):
            errores = 0
            for xi, objetivo in zip(X, y):
                actualizar = self.eta * (objetivo - self.predict(xi))
                self.w_[1:] += actualizar * xi
                self.w_[0] += actualizar
                errores += int(actualizar != 0.0)
            self.errores_.append(errores)
        return self

    def entrada(self, X):
        ## Calcular la entrada
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        ## Devolver la etiqueta de la clase
```

```
        return np.where(self.entrada(X) >= 0.0, 1, -1)
```

Entrenar un Perceptrón con el conjunto de datos Iris

Leer el conjunto de datos Iris

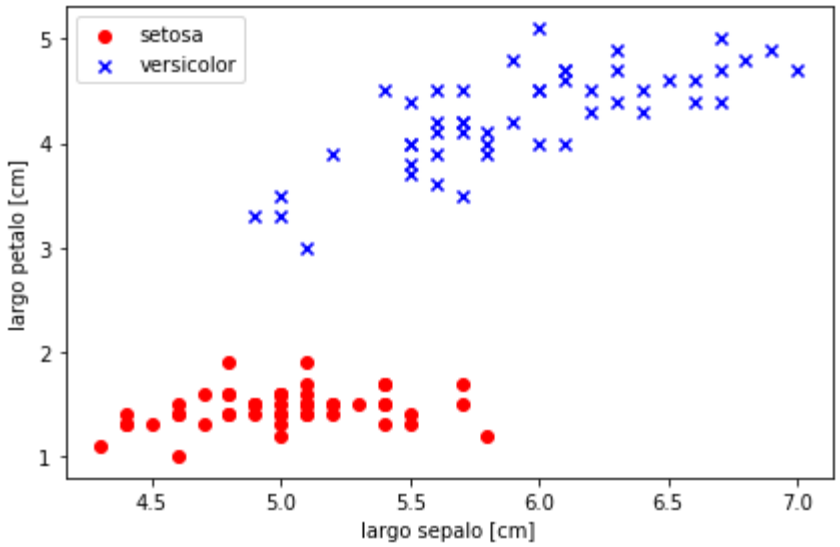
In [2]:
```
import pandas as pd

df = pd.read_csv('../datos/iris.csv', header=None)
df.tail()
```

Out[2]:

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

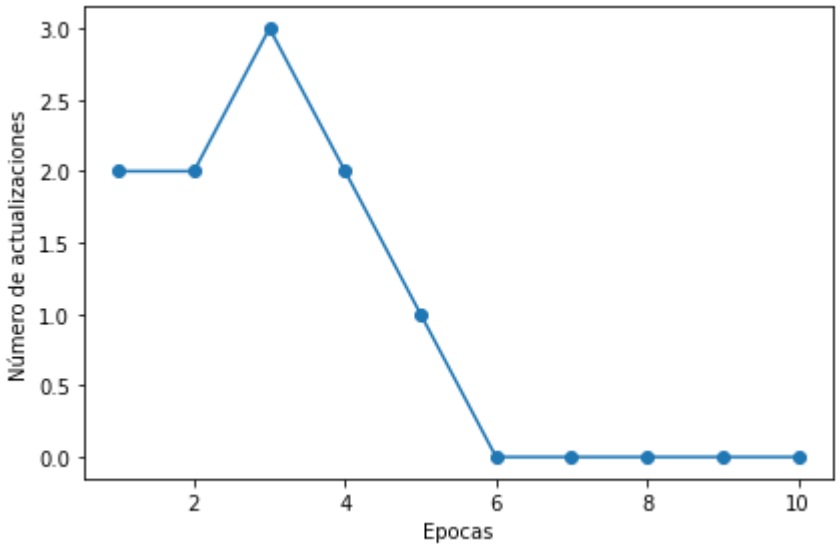
Graficar los datos de Iris

In [3]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

# seleccionar setosa y versicolor
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)

# extraer el largo del sepalo y el largo del petalo
X = df.iloc[0:100, [0, 2]].values

# graficar la data
plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='x', label='versicolor')

plt.xlabel('largo sepalo [cm]')
plt.ylabel('largo petalo [cm]')
plt.legend(loc='upper left')

plt.tight_layout()
plt.show()
```



Entrenar el Perceptrón

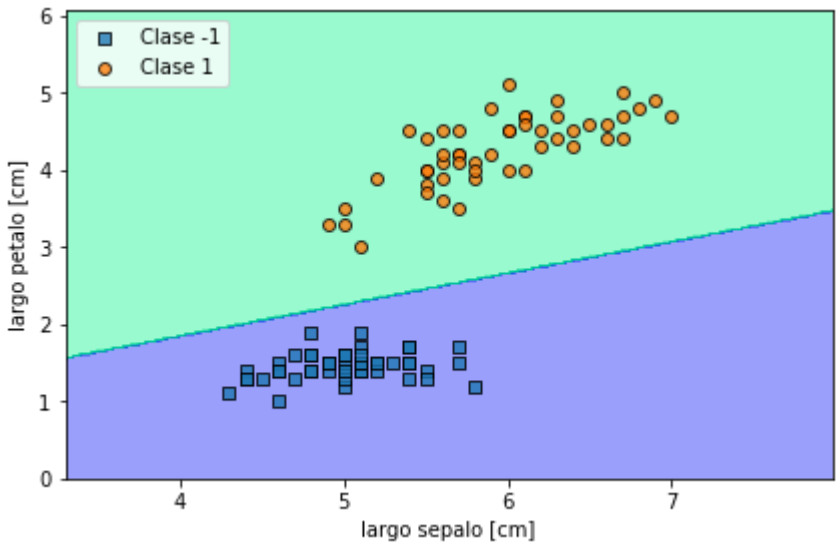
In [4]:
```
ppn = Perceptron(eta=0.1, num_iteraciones=10)

ppn.fit(X, y)

plt.plot(range(1, len(ppn.errores_) + 1), ppn.errores_, marker='o')
plt.xlabel('Epocas')
plt.ylabel('Número de actualizaciones')

plt.tight_layout()
plt.show()
```



Una función para graficar las regiones de decisión

In [5]:
```
from matplotlib.colors import ListedColormap


def graficar_regiones_decision(X, y, clasificador, resolucion=0.02):

    # fijar los marcadores y el mapa de colores
    marcadores = ('s', 'o', 'x', '^', 'v')
    colores = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    mapa_colores = ListedColormap(colores[:len(np.unique(y))])

    # plot the decision surface
    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, resolucion),
                           np.arange(x2_min, x2_max, resolucion))
    Z = clasificador.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap='winter')
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # graficar los ejemplos de clases
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, #c=colores[idx],
                    cmap='winter',
                    edgecolor='black',
                    marker=marcadores[idx],
                    label=f'Clase {cl}')
```

In [6]:
```
graficar_regiones_decision(X, y, clasificador=ppn)
plt.xlabel('largo sepalo [cm]')
plt.ylabel('largo petalo [cm]')
plt.legend(loc='upper left')

plt.tight_layout()
plt.show()
```

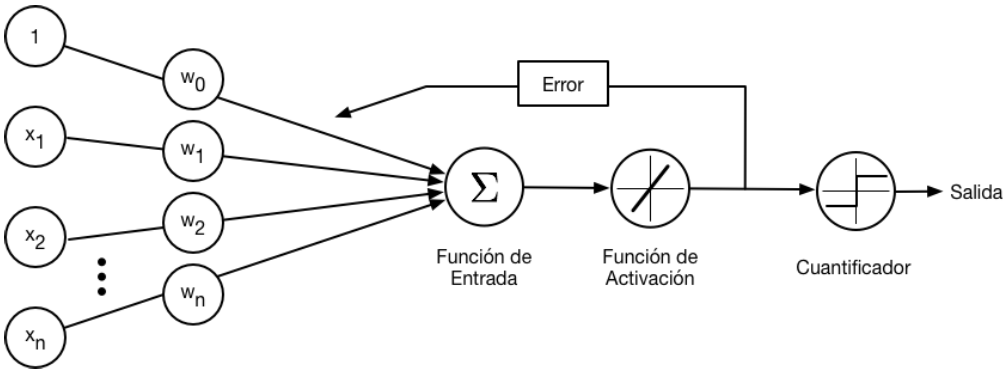


El Adaline (Adaptive linear neurons) y la convergencia del aprendizaje

El Adaline (ADAptive LInear NEuron) fue publicado por Bernard Widrow y su estudiante de doctorado Tedd Hoff, unos años después del algoritmo perceptrón de Frank Rosenblatt, y puede ser considerado como una mejora en este último (B. Widrow y otros, *An Adaptive "Adaline" neuron using chemical "memistors"*, Reporte técnico número 1553-2. Stanford Electron. Labs, Stanford, CA, Octubre 1960). El algoritmo de Adaline es particularmente interesante porque ilustra el concepto clave de definir y minimizar las funciones de coste, lo que sentará las bases para el entendimiento de algoritmos de aprendizaje automáticos más avanzados para la clasificación, como la regresión logística y máquinas de soporte vectoriales (SVM).

La diferencia clave entre la regla de Adaline (también conocida como la regla de Widrow-Hoff) y la del perceptron de Rosenblatt es que los pesos se actualizan sobre la base de una función de activación lineal en lugar de una función de escalón unitario como en el perceptron. En Adaline, esta función de activación lineal $\phi(z)$ es simplemente la función de identidad de la entrada de manera que $\phi(w^T x) = w^T x$.

Mientras que la función de activación lineal se utiliza para aprender los pesos, un cuantificador, que es similar a la función de escalón unitario que hemos visto antes, puede ser usado para predecir las etiquetas de clase, como se ilustra en la siguiente figura:



Funciones de Coste

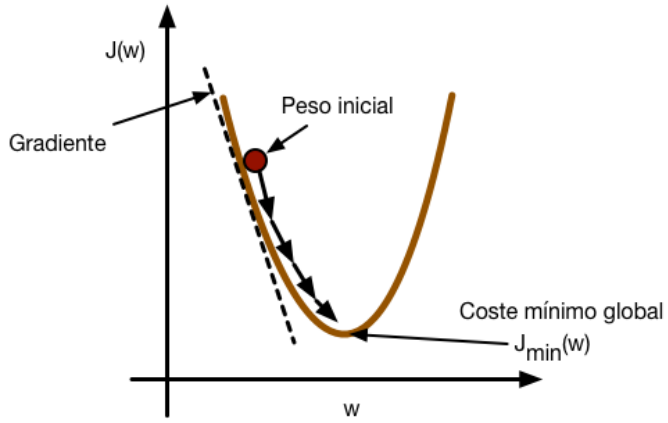
Uno de los ingredientes clave de los algoritmos supervisados de aprendizaje automático es definir una función objetiva que se debe optimizar durante el proceso de aprendizaje. Esta función objetivo es a menudo una función de coste que queremos minimizar. En el caso del Adaline, podemos determinar la función de coste J para aprender los pesos como el error cuadrático medio (MSE) entre el resultado calculado y la etiqueta de clase verdadera.

$$J(w) = \frac{1}{2n} \sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

La ventaja principal de esta función de activación lineal continua es que la función de coste se vuelve diferenciable. Otra propiedad importante de esta función de coste es que es convexa; Por lo tanto, podemos usar un algoritmo de optimización simple, pero potente, llamado descenso de gradiente para encontrar los pesos que minimizan nuestra función de coste para clasificar las muestras en el conjunto de datos.

Descenso del Gradiente

Como se ilustra en la siguiente figura, se puede describir el principio detrás del descenso del gradiente como bajar una colina hasta alcanzar un mínimo de coste local o global. En cada iteración, se toma un paso en sentido contrario del gradiente donde el tamaño del paso se determina por el valor de la tasa de aprendizaje, así como la pendiente del gradiente:



Usando descenso del gradiente, se pueden actualizar los pesos tomando un paso en sentido contrario al gradiente $\nabla J(w)$ de la finción de coste $J(w)$

$$w = w + \Delta w$$

Donde, el cambio de los pesos Δw es definido como gradiente negativo multiplicado por la tasa de aprendizaje η:

$$\Delta w = -\eta \, \Delta \, J(w)$$

Para calcular el gradiente de la función de coste, se necesita calcular la derivada parcial de la función de coste con respecto a cada peso w_j,

$$\frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

de esta forma se puede escribir la actualización del peso w_j como:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

Como todos los pesos se actualizan simultaneamente, la regla de aprendizaje del Adaline se convierte en:

$$w = w + \Delta w$$

Aunque la regla de aprendizaje del Adalina se parece a la del Perceptrón, la $\phi(z^{(i)})$ con $z^{(i)} = w^T x^{(i)}$ es un número real y no una etiqueta de clase entera. Además, la actualización de los pesos se calcula tomando en cuenta todos las muestras del conjunto de entrenamiento, en vez de, actualizar los pesos incrementalmente despues de cada muestra. Por lo que a este enfoque se le denomina descenso del gradiente por lotes.

Derivación de la derivada parcial de la suma al cuadrado de los errores

La derivada parcial de SCE con respecto a w_j se puede obtener de la siguente forma:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

$$= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

$$= \frac{1}{2} \sum_i 2(y^{(i)} - \phi(z^{(i)})) \frac{\partial}{\partial w_j} (y^{(i)} - \phi(z^{(i)}))$$

$$= \sum_i (y^{(i)} - \phi(z^{(i)})) \frac{\partial}{\partial w_j} (y^{(i)} - \sum_i (w_j^{(i)} x_j^{(i)}))$$

$$= \sum_i (y^{(i)} - \phi(z^{(i)}))(-x_j^{(i)})$$

$$= -\sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

Implementación del Adaline (Adaptive linear neuron) en Python

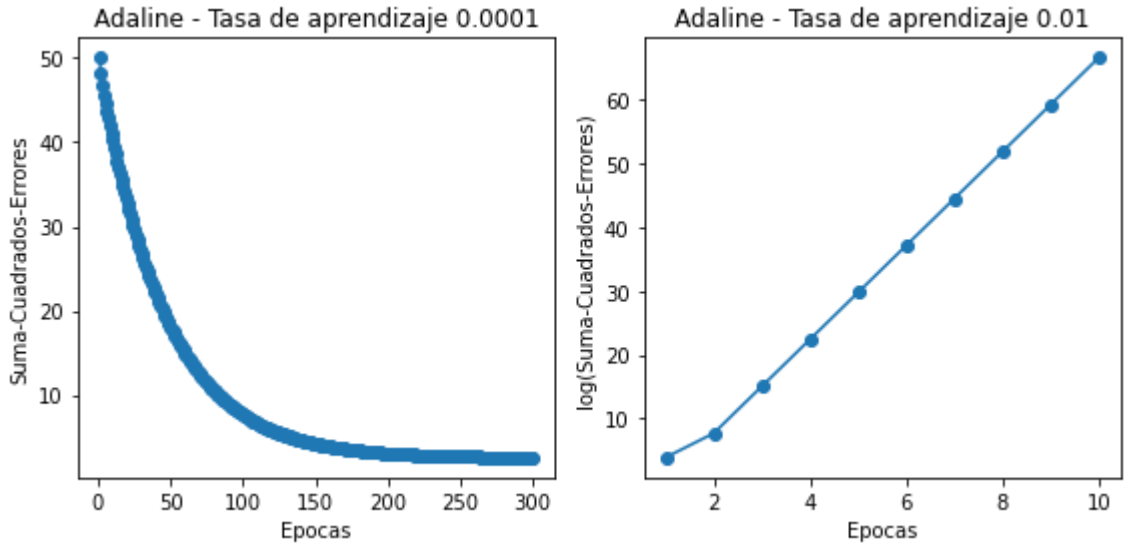
In [7]:

```
class AdalineGD(object):
    ''' Clasificador ADAptive LInear NEuron

    Parámetros
    ----------
    eta : float
      Taza de aprendizaje (entre 0.0 y 1.0)
    num_iteraciones : int
      Pasadas sobre el conjunto de datos de entrenamiento
    random_state : int
      Semilla para el generador de números aleatorios, para
      la inicialización de los pesos

    Atributos
    ---------
    w_ : arreglo de 1D
      Pesos después del entrenamiento

    '''

    def __init__(self, eta=0.01, num_iteraciones=50):
        self.eta = eta
        self.num_iteraciones = num_iteraciones

    def fit(self, X, y):
        ''' Ajustar la data de entrenamiento

        Parámetros
        ----------
        X : arreglo, forma = [n_ejemplos, n_atributos]
          Vectores de entrenamineto, donde n_ejemplos es el números
          de ejemplos y n_atributos es el números de atributos
        y : arreglo, forma = [n_ejemplos]
          Valores objetivo

        Retorna
        -------
        self : objeto

        '''
        self.w_ = np.zeros(1 + X.shape[1])
        self.coste_ = []

        for i in range(self.num_iteraciones):
            entrada = self.entrada(X)
            # En este caso el método "activacion" no hace nada
            # en el código ya que es la función de identidad. Se
            # podría escribir `salida = self.entrada(X)` directamente.
            # El proposito de la activation es conceptual, por lo que,
            # en el caso de regresión logistica, se puede cambiar por
            # una función sigmoid para implementar un clasificador de regresión logistica.
            salida = self.activacion(X)
            errores = (y - salida)
            self.w_[1:] += self.eta * X.T.dot(errores)
            self.w_[0] += self.eta * errores.sum()
            coste = (errores**2).sum() / 2.0
            self.coste_.append(coste)
        return self

    def entrada(self, X):
        #Calcular la entrada
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activacion(self, X):
```

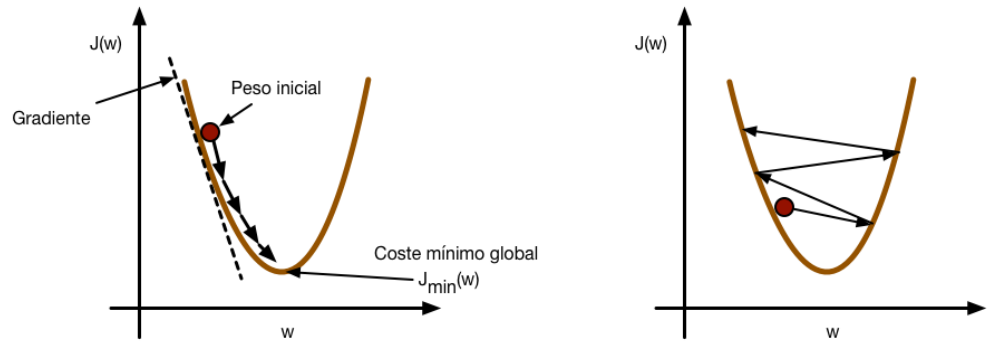
```
        #Calcular la activacion lineal
        return self.entrada(X)

    def predict(self, X):
        # Retornar la etiqueta de la clase
        return np.where(self.activacion(X) >= 0.0, 1, -1)
```

Influencia de la tasa de aprendizaje en la convergencia

In [8]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))

ada1 = AdalineGD(num_iteraciones=300, eta=0.0001).fit(X, y)
ax[0].plot(range(1, len(ada1.coste_) + 1), ada1.coste_, marker='o')
ax[0].set_xlabel('Epocas')
ax[0].set_ylabel('Suma-Cuadrados-Errores')
ax[0].set_title('Adaline - Tasa de aprendizaje 0.0001')

ada2 = AdalineGD(num_iteraciones=10, eta=0.01).fit(X, y)
ax[1].plot(range(1, len(ada2.coste_) + 1), np.log(ada2.coste_), marker='o')
ax[1].set_xlabel('Epocas')
ax[1].set_ylabel('log(Suma-Cuadrados-Errores)')
ax[1].set_title('Adaline - Tasa de aprendizaje 0.01')

plt.tight_layout()
plt.show()
```



Si la tasa de aprendizaje es demasiado alta, el algoritmo de descenso del gradiente puede que no converja.
Si la tasa de aprendizaje es demasiado baja, el algoritmo de descenso del gradiente puede que tome muchas epocas para convergir.



Normalización de los datos

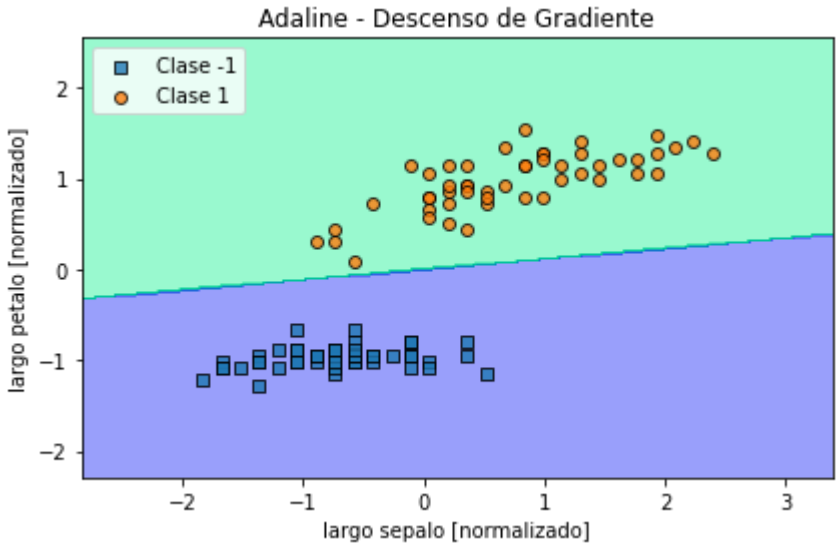
Para optimizar el rendimiento de muchos algoritmos de aprendizaje de maquina es necesario realizar un escalamiento de los atributos. Descenso del gradiente es uno de los algoritmos que se benefician del escalamiento de los atributos. Uno de los metodos de escalamiento es la normalización, que le da a los datos la propiedad de una distribución normal. Donde la media de cada atributo se centra en 0 y la desviación estándar es 1. Usando para cada atributo *j*, la siguiente formula:

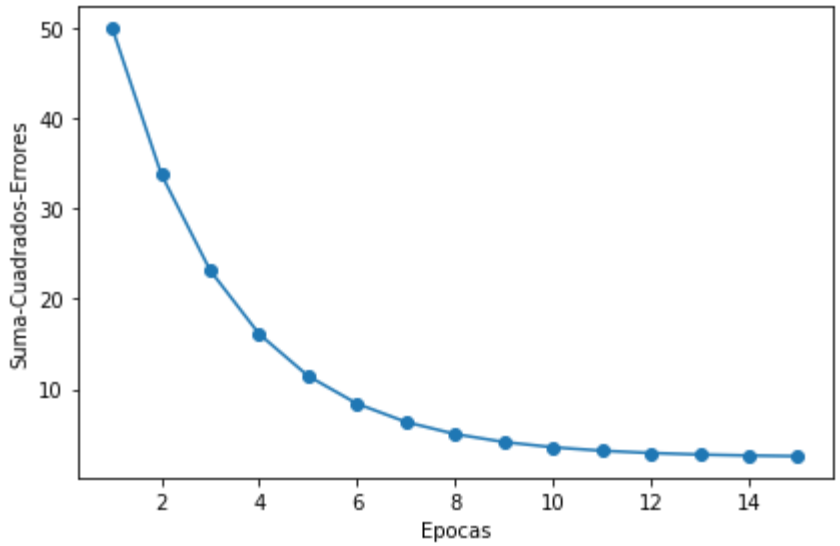
$$x_j^{'} = \frac{x_j - \mu_j}{\sigma_j}$$

In [9]:
```
# normalizar los atributos
X_std = np.copy(X)
X_std[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()
X_std[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()
```

Entrenar de nuevo el Adaline con una tasa de aprendizaje η = 0.01:

In [10]:
```
ada = AdalineGD(num_iteraciones=15, eta=0.01)
ada.fit(X_std, y)

graficar_regiones_decision(X_std, y, clasificador=ada)
plt.title('Adaline - Descenso de Gradiente')
plt.xlabel('largo sepalo [normalizado]')
plt.ylabel('largo petalo [normalizado]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()

plt.plot(range(1, len(ada.coste_) + 1), ada.coste_, marker='o')
plt.xlabel('Epocas')
plt.ylabel('Suma-Cuadrados-Errores')

plt.tight_layout()
plt.show()
```





Descenso del gradiente estocástico para el aprendizaje automático a gran escala

En este método de minimización de una función de coste dando un paso en la dirección opuesta al gradiente, es necesario usar para su calculo todo el conjunto de entrenamiento. Es por esta razón que este enfoque a veces también se conoce como descenso del gradiente por lotes. En el caso que se tenga un conjunto de datos muy grande con millones de datos, lo que no es infrecuente en muchas aplicaciones de aprendizaje automático. La ejecución del descenso del gradiente por lote puede ser computacionalmente bastante costoso en tales escenarios, ya que se necesita reevaluar todo el conjunto de datos de entrenamiento cada vez que damos un paso hacia el mínimo global.

Una alternativa muy popular al descenso del gradiente por lote es el descenso del gradiente estocástico, tambien llamdo descenso del gradiente iterativo o en-linea. En vez de actualizar los pesos basado en la suma de los errores acumulados sobre todas las muestras $x^{(i)}:

$$\Delta w_j = \eta \sum_i (y^{(i)} - \phi(z^{(i)}))x_j^{(i)},$$

Se actualizan los pesos incrementalmente para cada ejemplo de entrenamiento:

$$\eta(y^{(i)} - \phi(z^{(i)}))x_j^{(i)}$$

Aunque el descenso del gradiente estocástico puede considerarse como una aproximación del descenso del gradiente, alcanza típicamente la convergencia mucho más rápidamente debido a las actualizaciones más frecuentes de los pesos. Puesto que cada gradiente se calcula sobre la base de un solo ejemplo de entrenamiento, la superficie de error es más ruidosa que en el descenso del gradiente, lo que también puede tener la ventaja de que el descenso del gradiente estocástico puede escapar de los mínimos locales superficiales más fácilmente. Para obtener resultados precisos a través del descenso del gradiente estocástico, es importante presentarlos con datos en un orden aleatorio, por lo que se revuelve el conjunto de entrenamiento para cada época para prevenir ciclos.

Implementación del descenso de gradiente estocástico

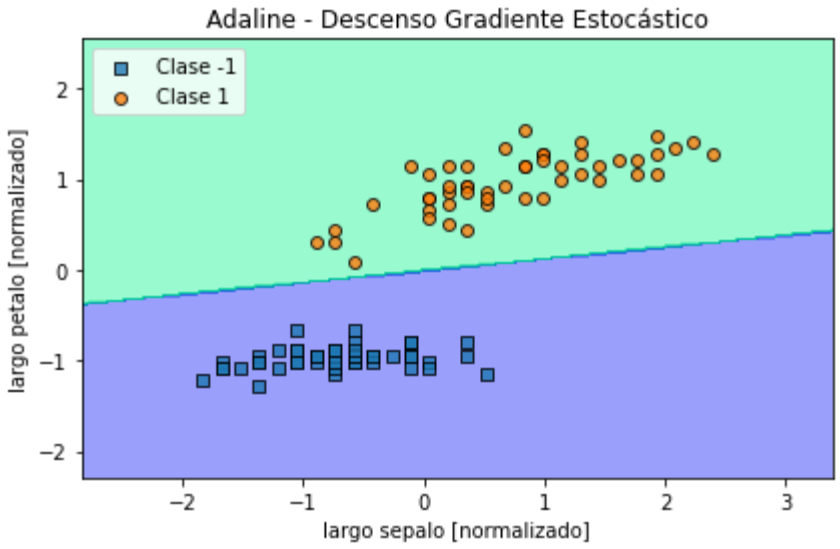
In [11]:

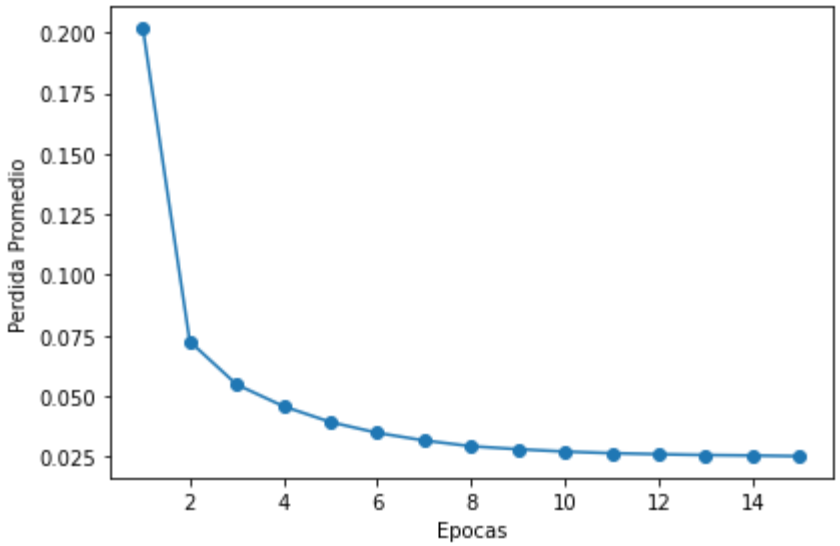
```
from numpy.random import seed

class AdalineDGE(object):
    ''' Clasificador ADAptive LInear NEuron

    Parámetros
    ----------
    eta : float
      Taza de aprendizaje (entre 0.0 y 1.0)
    num_iteraciones : int
      Pasadas sobre el conjunto de datos de entrenamiento
    barajar : bool (por defecto: True)
        Barajar la data de entrenamiento por cada epoca si es True
        para prevenir ciclos
    random_state : int
      Semilla para el generador de números aleatorios, para
      la inicialización de los pesos

    Atributos
    ---------
    w_ : arreglo de 1D
      Pesos después del entrenamiento

    '''

    def __init__(self, eta=0.01, num_iteraciones=10, barajar=True, semilla=None):
        self.eta = eta
        self.num_iteraciones = num_iteraciones
        self.w_initializado = False
        self.barajar = barajar
        if semilla:
            seed(semilla)

    def fit(self, X, y):
        ''' Ajustar la data de entrenamiento

        Parámetros
        ----------
        X : arreglo, forma = [n_ejemplos, n_atributos]
          Vectores de entrenamineto, donde n_ejemplos es el números
          de ejemplos y n_atributos es el números de atributos
        y : arreglo, forma = [n_ejemplos]
          Valores objetivo

        Retorna
        -------
        self : objeto

        '''
        self._inicializar_pesos(X.shape[1])
        self.coste_ = []
        for i in range(self.num_iteraciones):
            if self.barajar:
                X, y = self._barajar(X, y)
            coste = []
            for xi, objetivo in zip(X, y):
                coste.append(self._actualizar_pesos(xi, objetivo))
            coste_promedio = sum(coste) / len(y)
            self.coste_.append(coste_promedio)
        return self

    def ajuste_parcial(self, X, y):
        # Ajustar los datos de entrenamiento sin reinicializar los pesos
        if not self.w_initializado:
```

```
            self._inicializar_pesos(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, objtivo in zip(X, y):
                self._actualizar_pesos(xi, objetivo)
        else:
            self._actualizar_pesos(X, y)
        return self

    def _barajar(self, X, y):
        # Barajar los datos de entrenamiento
        r = np.random.permutation(len(y))
        return X[r], y[r]

    def _inicializar_pesos(self, m):
        # Inicializar los pesos a valores aleatorios pequeños
        self.w_ = np.zeros(1 + m)
        self.w_initializado = True

    def _actualizar_pesos(self, xi, objetivo):
        # Aplicar la regla de aprendizaje Adaline para actualizar los pesos
        salida = self.entrada(xi)
        error = (objetivo - salida)
        self.w_[1:] += self.eta * xi.dot(error)
        self.w_[0] += self.eta * error
        coste = 0.5 * error**2
        return coste

    def entrada(self, X):
        # Calcular la entrada
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activacion(self, X):
        # Calcular la activación lineal
        return self.entrada(X)

    def predict(self, X):
        # Retornar la etiqueta de la clase
        return np.where(self.activacion(X) >= 0.0, 1, -1)
```

In [12]:

```
ada = AdalineDGE(num_iteraciones=15, eta=0.01, semilla=1)
ada.fit(X_std, y)

graficar_regiones_decision(X_std, y, clasificador=ada)
plt.title('Adaline - Descenso Gradiente Estocástico')
plt.xlabel('largo sepalo [normalizado]')
plt.ylabel('largo petalo [normalizado]')
plt.legend(loc='upper left')

plt.tight_layout()
plt.show()

plt.plot(range(1, len(ada.coste_) + 1), ada.coste_, marker='o')
plt.xlabel('Epocas')
plt.ylabel('Perdida Promedio')

plt.tight_layout()
plt.show()
```





Como podemos ver, el costo promedio disminuye bastante rápido, y el límite de decisión final después de 15 épocas parece similar al descenso de gradiente por lotes con Adaline. Si queremos actualizar nuestro modelo, por ejemplo, en un escenario de aprendizaje en línea con flujo de datos, podríamos simplemente llamar al método partial_fit en muestras individuales, por ejemplo, `ada.partial_fit(X_std[0, :], y[0])`

Entrenando un Perceptrón usando scikit-learn

Usando dos de los atributos del conjunto de datos Iris entrenar un Perceptron.

In [13]:
```
# Importar el conjunto de datos Iris
from sklearn import datasets
import numpy as np
iris = datasets.load_iris()
# Seleccionar los atributos largo del petalo y ancho del petalo
X = iris.data[0:100, [0,2]]
y = iris.target[0:100]
```

Crear el conjunto de entrenamiento y el conjunto de prueba (70% entrenamiento y 30% prueba):

In [14]:
```
from sklearn.model_selection import train_test_split
X_entrenamiento, X_prueba, y_entrenamiento, y_prueba = train_test_split(X, y, test_size=0.3, ran
```

Normalizar los datos para tener un mejor rendimiento del algoritmo

In [15]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
sc.fit(X_entrenamiento)
X_entrenamiento_normalizado = sc.transform(X_entrenamiento)
X_prueba_normalizado = sc.transform(X_prueba)
```

Entrenar un Perceptrón usando los datos de entrenamiento normalizados

In [16]:
```
from sklearn.linear_model import Perceptron
perceptron = Perceptron(max_iter=40, eta0=0.1, random_state=0)
perceptron.fit(X_entrenamiento_normalizado, y_entrenamiento)
```

Out[16]:
```
Perceptron(eta0=0.1, max_iter=40)
```

Aplicar el modelo del Perceptrón entrenado para predecir las clases del conjunto de prueba:

In [17]:
```
y_prediccion  = perceptron.predict(X_prueba_normalizado)
print('Ejemplos clasificados erroneamente: %d' % (y_prediccion != y_prueba).sum())
```

```
Ejemplos clasificados erroneamente: 0
```

Usar la métrica de rendimiento `accuracy_score` de sklearn para calcular la precisión del Perceptrón:

In [18]:
```
from sklearn.metrics import accuracy_score
print('Precisión: %.2f' % accuracy_score(y_prueba, y_prediccion))
```

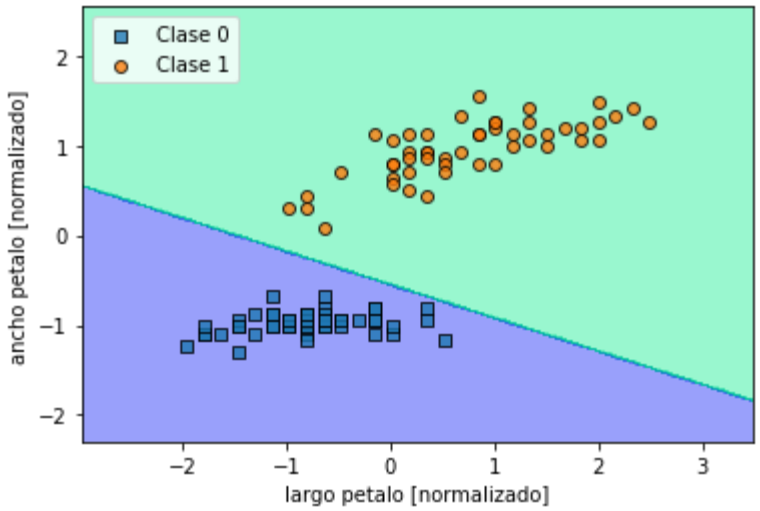
```
Precisión: 1.00
```

Usando una versión modificada de la función `graficar_regiones_decision` visualizar las regiones de decisión:

Con esta ligera modificación de la función `graficar_regiones_decision_2`, ahora se puede especificar los índices de los ejemplos que queremos marcar en las gráficas resultantes.

In [19]:
```
X_combinado = np.vstack((X_entrenamiento_normalizado, X_prueba_normalizado))
y_combinado = np.hstack((y_entrenamiento, y_prueba))
graficar_regiones_decision(X=X_combinado,
                           y=y_combinado,
                           clasificador=perceptron)
plt.xlabel('largo petalo [normalizado]')
plt.ylabel('ancho petalo [normalizado]')
plt.legend(loc='upper left')
plt.show
```

Out[19]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```


Clase 0
Clase 1
ancho petalo [normalizado]
largo petalo [normalizado]
2
1
0
−1
−2
−2
−1
0
1
2
3

In [ ]: