Práctica 6
Support Vector Machine (SVM)

Rafael Herrera Troca
Rubén Ruperto Díaz

Aprendizaje Automático y Big Data
Doble Grado en Ingeniería Informática y Matemáticas

4 de septiembre de 2020

1. Introducción

En esta práctica se pretende aprender a utilizar la SVM como técnica de aprendizaje supervisado mediante la clase *sklearn.svm.svc*. En la primera parte se prueba cómo varían los resultados al utilizar distintos valores para el *kernel* y para los parámetros C y σ. En la segunda parte, se aplican los conocimientos adquiridos en la primera parte para entrenar una SVM que clasifique correos electrónicos en correo deseado o no deseado.

Hemos programado el código en el entorno *Spyder* con Python 3.7.

2. Procedimiento

2.1. Parte 1: Ejemplos de aplicación de SVM

El procedimiento para todos los apartados de esta primera parte es muy sencillo. Consiste en leer los datos proporcionados, crear la SVM con los parámetros pedidos y representar el resultado.

Para la lectura de datos usamos *scipy.io.loadmat*, ya que los datos vienen dados en ficheros con extensión *.mat*. En los tres apartados los datos suministrados constan de dos variables y pertenecen a dos categorías distintas.

En cuanto al entrenamiento de la SVM, usaremos la clase *SVC* del paquete *sklearn.svm*. Resulta muy sencilla de utilizar, al menos al nivel al que lo hemos hecho nosotros, ya que para construir la SVM basta con usar el constructor de la clase con los parámetros deseados para el *kernel* y para C. Una vez se tiene construida, se puede entrenar con los datos con la función *fit*, y usarla, una vez entrenada, para clasificar nuevos datos mediante la función *predict*. Además, con la función *score* se puede obtener el porcentaje medio de aciertos sobre un conjunto de datos dado, lo cual evita tener que calcularlo aparte y nos ha sido muy útil para estudiar los resultados y para elegir la mejor configuración de los parámetros. Hay que comentar también que esta clase no tiene el *kernel gaussiano* entre los posibles valores para este parámetro. Sin embargo, este problema se puede solventar usando el *kernel* de función de base radial (*rbf*) y dándole como parámetro $\gamma = \frac{1}{2\sigma^2}$.

Finalmente, para representar los resultados usamos las funciones de la librería *matplotlib.pyplot*. A la hora de representar los datos, utilizamos *scatter* teniendo en cuenta la categoría a la que pertenece cada uno para pintarlo de un color u otro. La representación de la frontera de decisión, sin embargo, es algo más compleja, ya que la clase que hemos usado para construir la SVM no produce la frontera de forma explícita. Para solucionarlo, hemos usado una malla de 100×100 puntos para cubrir el espacio y hemos usado la SVM para predecir a qué categoría debería pertenecer cada punto. Una vez hecho esto, utilizando la función *contour* hemos representado

la curva de nivel donde la clasificación cambia de una categoría a otra, que es, por definición, la frontera de decisión.

2.2. Parte 2: Detección de *spam*

Para esta segunda parte, utilizaremos como datos una colección de 3301 correos electrónicos repartidos en 500 de correo basura (*spam*) y 2801 de correo deseado (*ham*). Además, estos 2801 correos de *ham* se dividen a su vez en 2551 de *easy ham*, que supuestamente son más fáciles de distinguir del *spam*, y 250 de *hard ham*, más difícil de distinguir. Como ya se intuye, querremos entrenar una SVM capaz de clasificar los correos en *spam* o *ham*.

El primer paso para construir esta SVM es procesar los datos, ya que no se le pueden dar los correos directamente. Este procesamiento consta de varias partes. Para comenzar, leemos cada correo usando el paquete *codecs* para interpretar la codificación *utf-8* correctamente, y lo transformamos a una lista con las palabras que contiene dicho correo. Hacemos esta transformación por medio de la función *email2TokenList* proporcionada en el fichero *process_email.py*. Una vez tenemos la lista de palabras, debemos transformarla a un dato de tipo numérico que pueda ser interpretado por la SVM. Para ello tomaremos un diccionario ordenado alfabéticamente con todas las palabras que aparecen al menos 100 veces en el cuerpo de los correos proporcionados. Para construir este diccionario utilizamos la función *getVocabDict* proporcionada en el fichero *get_vocab_dict.py*. Una vez lo tenemos, transformamos cada correo en un vector de longitud 1899, que es el tamaño del diccionario, que tiene 1 en las posiciones correspondientes a palabras que aparecen en él y 0 en las restantes. Este será el dato con el que trabaje la SVM.

Una vez tenemos todos los correos procesados, pasamos a dividir cada uno de los conjuntos proporcionados en tres partes: una para entrenamiento, una para validación y otra para test. La proporción que hemos escogido para cada una de estas tres partes ha sido un 60 % para entrenamiento, un 20 % para validación y un 20 % para test. Con los conjuntos divididos en cada parte, solo queda unirlas para formar la matriz de datos que usaremos en la parte correspondiente del proceso. Para construir el vector de respuestas, basta crear un vector con 0 en las posiciones cuyo dato corresponde a un correo de *ham* y 1 en las correspondientes a *spam*.

Cuando tenemos los datos listos procedemos a entrenar la SVM. Utilizamos un *kernel gaussiano* con distintos valores de C y σ, concretamente, $C, \sigma \in \{0{,}01,\ 0{,}03,\ 0{,}1,\ 0{,}3,\ 1,\ 3,\ 10,\ 30\}$. Para cada configuración de los parámetros calculamos el porcentaje de aciertos para el conjunto de entrenamiento y el de validación y los almacenamos. Después, nos quedamos con los parámetros que dan el mayor porcentaje de acierto para los datos de validación. Estos parámetros son los mejores, en principio, para los datos con los que lo hemos entrenado.

Una vez escogidos estos parámetros, comprobamos el porcentaje de acierto sobre

los datos de test, que aún no hemos utilizado y por tanto son más realistas. También realizamos algunos experimentos adicionales, como comprobar qué porcentaje de aciertos tiene sobre los datos de test correspondientes a *spam*, *hard ham* y *easy ham*.

3. Resultados

3.1. Parte 1: Ejemplos de aplicación de SVM

Comenzamos estudiando cómo afecta la elección del parámetro C al resultado de la SVM. Para ello, cargamos los datos del archivo *ex6data1.mat*, que contienen un conjunto de datos pertenecientes a dos categorías linealmente separables. Los datos están formados por dos variables y un tercer atributo binario a predecir.

Instanciamos una SVM con *kernel* lineal y la entrenamos con los datos, primero con $C = 1$ y luego con $C = 100$. Observamos que para el valor $C = 1$, la frontera de decisión se sobreajusta menos a los ejemplos de entrenamiento, dividiéndolos en dos grupos separados por un margen mayor, aunque un punto extremo queda mal clasificado. En cuanto al valor $C = 100$, la frontera de decisión resultante se sobreajusta a los datos de entrenamiento, forzando a clasificar correctamente el punto anómalo y resultando en una división que, a pesar de clasificar correctamente todos los casos de entrenamiento, resulta menos natural a la vista de la distribución de los datos.

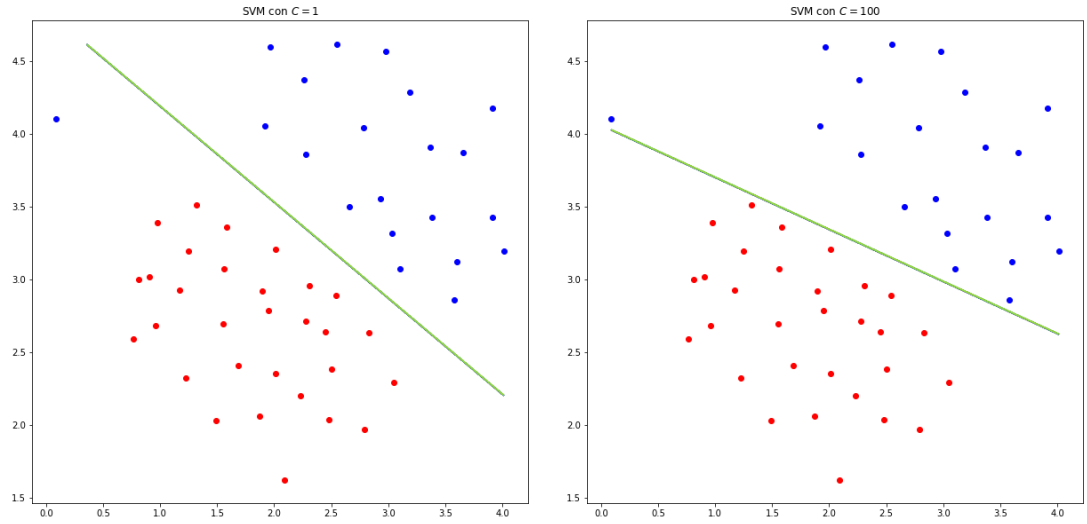

Figura 1: Resultados de la SVM con kernel lineal y $C \in \{1, 100\}$, clasificando el primer conjunto de datos

La función de coste que pretende minimizar la SVM es:

$$\min_{\theta} C \sum_{i=1}^{m} \left[y^{(i)} \, \text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)}) \, \text{cost}_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

donde m es el número de casos, n el número de variables y $\text{cost}_k(\theta^T x^{(i)})$ una función que devuelve 0 si el punto $x^{(i)}$ se predice perteneciente a la clase $k \in \{0, 1\}$ y un valor proporcional a la distancia a la frontera si no. En el fondo, el término $\left[y^{(i)}\,\text{cost}_1(\theta^T x^{(i)}) + (1 - y^{(i)})\,\text{cost}_0(\theta^T x^{(i)})\right]$ devolverá 0 si el punto se está clasificando correctamente y un valor mayor que 0 si no. Entonces, aumentar el valor de C incrementa el peso de los errores cometidos al clasificar durante el entrenamiento, y si C es muy grande, un sólo error en el entrenamiento aumentará mucho la función de coste. Esto se traduce en que al aumentar C se incrementa la tendencia del modelo a sobreajustarse a los datos de entrenamiento.

En efecto, el parámetro C es equivalente al inverso del λ de regularización utilizado en regresión: Valores grandes de C dan al modelo mayor libertad, pudiendo llegar a sobreajustarse a los datos de entrenamiento (menor *bias* y mayor *variance*), mientras que valores pequeños de C restringen la capacidad del modelo de adaptarse a los datos de entrenamiento, pudiendo llegar a no ser capaces de aprender las relaciones entre los atributos (mayor *bias* y menor *variance*).

A continuación, tratamos de clasificar los datos del segundo conjunto, incluido en el archivo *ex6data2.mat*. Estos datos también constan de dos atributos y un tercero binario a predecir, pero en esta ocasión no son linealmente separables, por lo que empleamos un *kernel gaussiano* para clasificarlos. Instanciamos la SVM con el *kernel* y los parámetros $C = 1$ y $\sigma = 0{,}01$ y mostramos el resultado en una gráfica. En esta ocasión, la frontera de decisión es una curva complicada, pero que consigue clasificar correctamente el 98,96 % de los datos de entrenamiento, gracias a que forman agrupaciones diferenciadas.

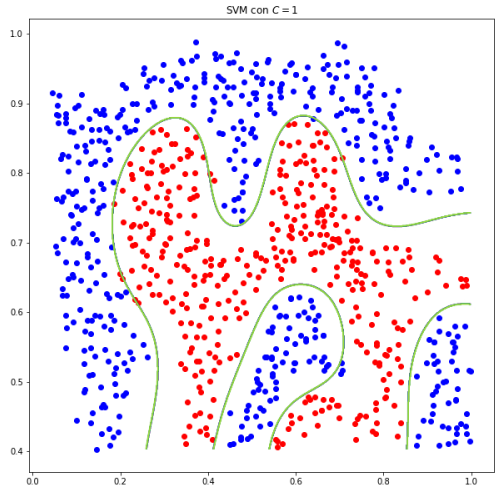

Figura 2: SVM con *kernel gaussiano* sobre el segundo conjunto de datos

Finalmente, creamos una última SVM con *kernel gaussiano* para clasificar el tercer conjunto de datos, que extraemos del archivo *ex6data3.mat*. En esta ocasión entrenamos la máquina con diferentes configuraciones de C y σ, dándoles valores pertenecientes al conjunto {0,01, 0,03, 0,1, 0,3, 1, 3, 10, 30}, y comprobamos que los mejores resultados se obtienen para $C = 1$ y $\sigma = 0{,}1$, con una tasa de acierto del 96,5 % para los datos de validación.

La influencia del parámetro σ consiste en que valores grandes de σ^2 provocan que los atributos f_i, que sustituyen a los x_i en la función de coste con *kernel*, varíen de forma más suave, mientras que valores pequeños de σ^2 hacen que los f_i varíen más abruptamente.

$$f_i = \exp\left(\frac{||x - l^{(i)}||^2}{2\sigma^2}\right) \quad l^{(i)} = x^{(i)}$$

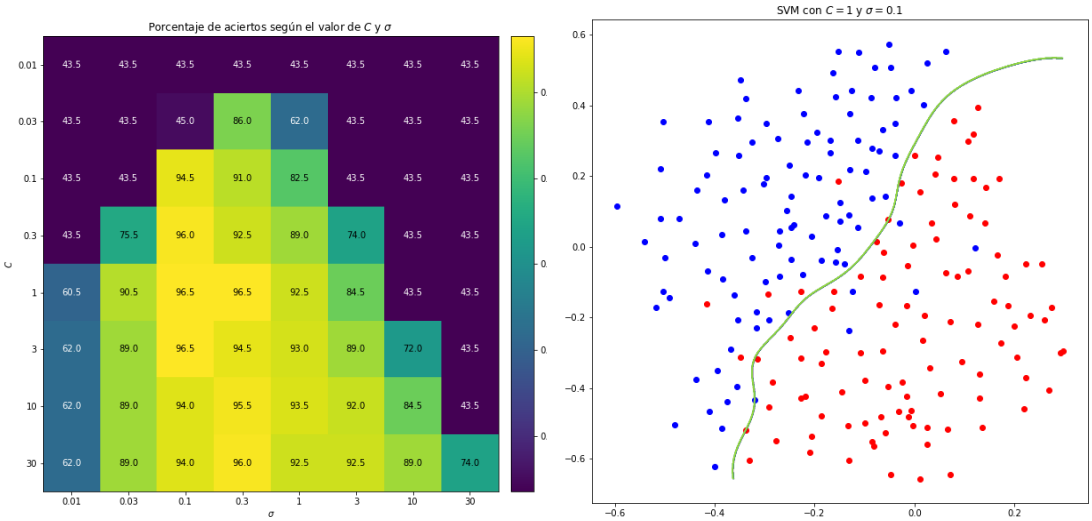

Figura 3: Porcentaje de acierto de la SVM con los datos de validación del tercer conjunto de datos para distintos valores de C y σ y frontera de decisión óptima obtenida para $C = 1$ y $\sigma = 0{,}1$ con los puntos de los ejemplos de entrenamiento

3.2. Parte 2: Detección de *spam*

Para entrenar la SVM que clasifica correos como *spam* o *ham*, leemos los datos y los separamos en ejemplos de entrenamiento, de validación y de test. Entrenamos la SVM con *kernel gaussiano* y distintos valores de C y σ con los ejemplos de entrenamiento y comprobamos su tasa de acierto con éstos y los de validación. Nos quedamos con la configuración que obtiene los resultados óptimos y la reevaluamos con los datos de test.

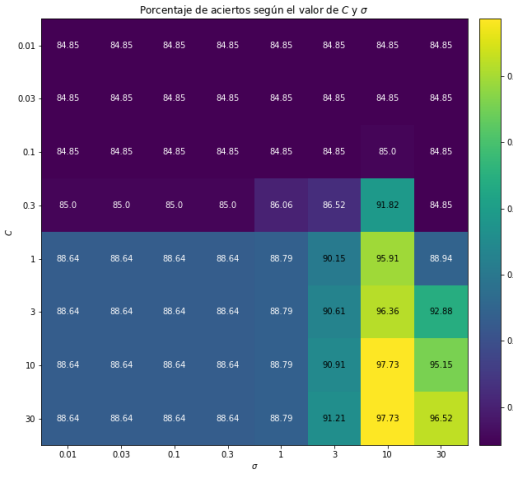

Figura 4: Éxito de la SVM al clasificar los ejemplos de validación en función del valor de C y σ

Los resultados obtenidos indican que la mejor configuración es la que tiene $C = 10$ y $\sigma = 10$, que clasifica correctamente el 99,75 % de los datos de entrenamiento y el 97,73 % de los de validación. Además, hemos desglosado el porcentaje de aciertos sobre los datos de test en los distintos tipos de correo analizados, siendo este un 94 % para los correos *spam*, un 100 % para los *easy ham* y un 92 % para los *hard ham*. Notamos que la SVM confunde más el *hard ham* que el *easy ham*, clasificándolo más veces como *spam*, sin embargo, al ser la proporción de *easy ham* mucho mayor que la de *hard ham*, la tasa total de aciertos para el conjunto de test es muy alta, del 98,49 %. En cualquier caso, el resultado obtenido por la SVM es bastante bueno incluso para el *hard ham*.

Por último, debemos indicar que los resultados pueden variar ligeramente entre distintas ejecuciones, debido a que la división de los datos en ejemplos de entrenamiento, validación y test se realiza de forma aleatoria, empleando la función *numpy.random.shuffle*.

4. Conclusiones

Esta práctica ha sido muy útil para entender mejor cómo funcionan las SVM y la influencia que tienen sus parámetros en el resultado obtenido. Asimismo, nos ha servido como primera toma de contacto con el paquete *sklearn*, comprobando que resulta muy sencillo y cómodo de utilizar. También ha resultado especialmente interesante aplicar la práctica sobre un ejemplo más realista que las matrices de números que hemos venido usando hasta ahora, como ha sido el clasificador de correos electrónicos.

5. Anexo: Código

A continuación se ofrece el código escrito para resolver los ejercicios propuestos en la práctica.

```
'''
Práctica 6

Rubén Ruperto Díaz y Rafael Herrera Troca
'''

import os
import codecs
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.axes_grid1 import make_axes_locatable
from sklearn.svm import SVC
from scipy.io import loadmat

os.chdir("./resources")

from process_email import email2TokenList
from get_vocab_dict import getVocabDict


#%% Parte 1.1

# Leemos los datos
data = loadmat('ex6data1.mat')
y = data['y'].ravel()
X = data['X']

# Entrenamos la SVM con kernel lineal y C=1 y
```

```
# representamos el resultado
svm = SVC(kernel='linear', C=1)
svm.fit(X,y)

meshX1 = np.linspace(np.min(X[:,0]), np.max(X[:,0]), 1000)
meshX2 = np.linspace(np.min(X[:,1]), np.max(X[:,1]), 1000)
meshX1, meshX2 = np.meshgrid(meshX1, meshX2)
meshY = svm.predict(np.array([meshX1.ravel(), meshX2.ravel()]).T).
    reshape(meshX1.shape)

plt.figure(figsize=(10,10))
plt.scatter(X[y==0,0], X[y==0,1], c='r', marker='o')
plt.scatter(X[y==1,0], X[y==1,1], c='b', marker='o')
plt.contour(meshX1, meshX2, meshY)
plt.title("SVM con $C=1$")
plt.savefig('P1.1C1.png')
plt.show()

# Entrenamos la SVM con C=100 y representamos el resultado
svm = SVC(kernel='linear', C=100)
svm.fit(X,y)

meshY = svm.predict(np.array([meshX1.ravel(), meshX2.ravel()]).T).
    reshape(meshX1.shape)

plt.figure(figsize=(10,10))
plt.scatter(X[y==0,0], X[y==0,1], c='r', marker='o')
plt.scatter(X[y==1,0], X[y==1,1], c='b', marker='o')
plt.contour(meshX1, meshX2, meshY)
plt.title("SVM con $C=100$")
plt.savefig('P1.1C100.png')
plt.show()


#%% Parte 1.2

# Leemos los datos
data = loadmat('ex6data2.mat')
y = data['y'].ravel()
X = data['X']

# Entrenamos la SVM con kernel gaussiano, C=1 y sigma=0.1 y
# representamos el resutlado
svm = SVC(kernel='rbf', C=1, gamma=1/(2*0.1**2))
svm.fit(X,y)

meshX1 = np.linspace(np.min(X[:,0]), np.max(X[:,0]), 1000)
meshX2 = np.linspace(np.min(X[:,1]), np.max(X[:,1]), 1000)
meshX1, meshX2 = np.meshgrid(meshX1, meshX2)
meshY = svm.predict(np.array([meshX1.ravel(), meshX2.ravel()]).T).
    reshape(meshX1.shape)

plt.figure(figsize=(10,10))
plt.scatter(X[y==0,0], X[y==0,1], c='r', marker='o')
plt.scatter(X[y==1,0], X[y==1,1], c='b', marker='o')
```

```
plt.contour(meshX1, meshX2, meshY)
plt.title("SVM con $C=1$")
plt.savefig('P1.2.png')
plt.show()


#%% Parte 1.3

# Leemos los datos
data = loadmat('ex6data3.mat')
y = data['y'].ravel()
yval = data['yval'].ravel()
X = data['X']
Xval = data['Xval']

# Entrenamos la SVM con kernel gaussiano y distintos valores para
# C y sigma, almacenando el porcentaje de acierto para cada uno
values = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]
acc = np.empty((len(values),len(values)))
for i in range(len(values)):
    for j in range(len(values)):
        svm = SVC(kernel='rbf', C=values[i], gamma=1/(2*values[j]**2))
        svm.fit(X,y)
        acc[i][j] = svm.score(Xval, yval)

# Representamos la matriz de porcentaje de aciertos según el valor
# de C y sigma
xLabels = [str(v) for v in values]
yLabels = [str(v) for v in values]
plt.figure(figsize=(10,10))
plt.title(r"Porcentaje de aciertos según el valor de $C$ y $\sigma$")
plt.ylabel('$C$')
plt.xlabel(r'$\sigma$')
fig = plt.subplot()
im = fig.imshow(acc, cmap=cm.viridis)
cax = make_axes_locatable(fig).append_axes("right", size="5%", pad
    =0.2)
plt.colorbar(im, cax=cax)
for i in range(len(xLabels)):
    for j in range(len(yLabels)):
        text = fig.text(i, j, acc[j][i]*100, ha="center", va="center",
    color=("k" if acc[j][i]>0.7 else "w"))
fig.set_xticks(np.arange(len(xLabels)))
fig.set_yticks(np.arange(len(yLabels)))
fig.set_xticklabels(xLabels)
fig.set_yticklabels(yLabels)
plt.savefig("P1.3Cuadros.png")
plt.show()

# Representamos el resultado de la SVM con mayor precisión
m = np.argmax(acc)
C = values[m//len(values)]
sigma = values[m%len(values)]
svm = SVC(kernel='rbf', C=C, gamma=1/(2*sigma**2))
svm.fit(X,y)
```

```
135
136 meshX1 = np.linspace(np.min(X[:,0]), np.max(X[:,0]), 1000)
137 meshX2 = np.linspace(np.min(X[:,1]), np.max(X[:,1]), 1000)
138 meshX1, meshX2 = np.meshgrid(meshX1, meshX2)
139 meshY = svm.predict(np.array([meshX1.ravel(), meshX2.ravel()]).T).
    reshape(meshX1.shape)
140
141 plt.figure(figsize=(10,10))
142 plt.scatter(X[y==0,0], X[y==0,1], c='r', marker='o')
143 plt.scatter(X[y==1,0], X[y==1,1], c='b', marker='o')
144 plt.contour(meshX1, meshX2, meshY)
145 plt.title(r"SVM con $C="+str(C)+"$ y $\sigma="+str(sigma)+"$")
146 plt.savefig('P1.3.png')
147 plt.show()
148
149
150 #%% Parte 2
151
152 # Cargamos el diccionario de palabras
153 dic = getVocabDict()
154
155 # Leemos y procesamos los datos correspondientes a spam
156 spam = np.zeros((len(os.listdir('spam')), len(dic)))
157 for i, filename in enumerate(os.listdir('spam')):
158     email_contents = codecs.open('spam/'+filename, 'r', encoding='utf
    -8', errors='ignore').read()
159     email_tokens = email2TokenList(email_contents)
160     for token in email_tokens:
161         if token in dic.keys():
162             spam[i,dic[token]-1] = 1
163
164 # Leemos y procesamos los datos correspondientes al easy ham
165 easy = np.zeros((len(os.listdir('easy_ham')), len(dic)))
166 for i, filename in enumerate(os.listdir('easy_ham')):
167     email_contents = codecs.open('easy_ham/'+filename, 'r', encoding='
    utf-8', errors='ignore').read()
168     email_tokens = email2TokenList(email_contents)
169     for token in email_tokens:
170         if token in dic.keys():
171             easy[i,dic[token]-1] = 1
172
173 # Leemos y procesamos los datos correspondientes al hard ham
174 hard = np.zeros((len(os.listdir('hard_ham')), len(dic)))
175 for i, filename in enumerate(os.listdir('hard_ham')):
176     email_contents = codecs.open('hard_ham/'+filename, 'r', encoding='
    utf-8', errors='ignore').read()
177     email_tokens = email2TokenList(email_contents)
178     for token in email_tokens:
179         if token in dic.keys():
180             hard[i,dic[token]-1] = 1
181
182 # Dividimos los conjuntos para entrenamiento, validación y test
183 spamIndx = np.arange(len(spam))
184 easyIndx = np.arange(len(easy))
185 hardIndx = np.arange(len(hard))
```

```
np.random.shuffle(spamIndx)
np.random.shuffle(easyIndx)
np.random.shuffle(hardIndx)

div1 = 0.6
div2 = 0.2

spamTrain = spam[spamIndx[:int(div1*len(spamIndx))]]
spamVal = spam[spamIndx[int(div1*len(spamIndx)):int((div1+div2)*len(spamIndx))]]
spamTest = spam[spamIndx[int((div1+div2)*len(spamIndx)):]]
easyTrain = easy[easyIndx[:int(div1*len(easyIndx))]]
easyVal = easy[easyIndx[int(div1*len(easyIndx)):int((div1+div2)*len(easyIndx))]]
easyTest = easy[easyIndx[int((div1+div2)*len(easyIndx)):]]
hardTrain = hard[hardIndx[:int(div1*len(hardIndx))]]
hardVal = hard[hardIndx[int(div1*len(hardIndx)):int((div1+div2)*len(hardIndx))]]
hardTest = hard[hardIndx[int((div1+div2)*len(hardIndx)):]]
X = np.vstack((spamTrain, easyTrain, hardTrain))
Xval = np.vstack((spamVal, easyVal, hardVal))
Xtest = np.vstack((spamTest, easyTest, hardTest))
y = np.concatenate((np.ones(len(spamTrain)), np.zeros(len(easyTrain)+len(hardTrain))))
yval = np.concatenate((np.ones(len(spamVal)), np.zeros(len(easyVal)+len(hardVal))))
ytest = np.concatenate((np.ones(len(spamTest)), np.zeros(len(easyTest)+len(hardTest))))

# Entrenamos una SVM con kernel gaussiano y distintos valores
# para C y sigma, y almacenamos el porcentaje de acierto
# con los datos de entrenamiento y los de validación
values = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]
accTrain = np.empty((len(values),len(values)))
accVal = np.empty((len(values), len(values)))
for i in range(len(values)):
    for j in range(len(values)):
        svm = SVC(kernel='rbf', C=values[i], gamma=1/(2*values[j]**2))
        svm.fit(X,y)
        accTrain[i][j] = svm.score(X, y)
        accVal[i][j] = svm.score(Xval, yval)

# Representamos la matriz de porcentaje de aciertos para los datos
# de validación según el valor de C y sigma
xLabels = [str(v) for v in values]
yLabels = [str(v) for v in values]
plt.figure(figsize=(10,10))
plt.title(r"Porcentaje de aciertos según el valor de $C$ y $\sigma$")
plt.ylabel('$C$')
plt.xlabel(r'$\sigma$')
fig = plt.subplot()
im = fig.imshow(accVal, cmap=cm.viridis)
cax = make_axes_locatable(fig).append_axes("right", size="5%", pad=0.2)
plt.colorbar(im, cax=cax)
```

```
234 for i in range(len(xLabels)):
235     for j in range(len(yLabels)):
236         text = fig.text(i, j, round(accVal[j][i]*100, 2), ha="center",
    va="center", color=("k" if accVal[j][i]>0.9 else "w"))
237 fig.set_xticks(np.arange(len(xLabels)))
238 fig.set_yticks(np.arange(len(yLabels)))
239 fig.set_xticklabels(xLabels)
240 fig.set_yticklabels(yLabels)
241 plt.savefig("P2Cuadros.png")
242 plt.show()
243
244 # Buscamos el mayor porcentaje de aciertos para los datos
245 # de validación
246 m = np.argmax(accVal)
247 C = values[m//len(values)]
248 sigma = values[m%len(values)]
249 print("La mejor SVM se construye con C =", C, "y sigma =", sigma)
250 print("Consigue un " + str(accTrain.ravel()[m] * 100) + "% de aciertos
    para los datos de entrenamiento.")
251 print("Consigue un " + str(accVal.ravel()[m] * 100) + "%  de aciertos
    para los datos de validación.")
252
253 # Entrenamos nuevamente la SVM para estos valores
254 svm = SVC(kernel='rbf', C=C, gamma=1/(2*sigma**2))
255 svm.fit(X,y)
256
257 # Comprobamos el porcentaje de acierto para los datos de test
258 accTest = svm.score(Xtest, ytest)
259 print("Consigue un " + str(accTest * 100) + "% de aciertos para los
    datos de test.")
260
261 # Comprobamos qué porcentaje de acierto tiene para el easy ham
262 # y para el hard ham por separado
263 accEasy = svm.score(easyTest, np.zeros(len(easyTest)))
264 accHard = svm.score(hardTest, np.zeros(len(hardTest)))
265 accSpam = svm.score(spamTest, np.ones(len(spamTest)))
266 print("Consigue un " + str(accEasy * 100) + "% de aciertos para el
    easy ham.")
267 print("Consigue un " + str(accHard * 100) + "% de aciertos para el
    hard ham.")
268 print("Consigue un " + str(accSpam * 100) + "% de aciertos para el
    spam.")
```