Práctica 4: Máquinas de vectores soporte

Convocatoria de enero (curso académico 2020/2021)

Asignatura: Introducción a los modelos computacionales
4º Grado Ingeniería Informática (Universidad de Córdoba)

24 de noviembre de 2020

Resumen

Esta práctica sirve para familiarizar al alumno con las máquinas de vectores soporte (*Support Vector Machines*, SVMs). De esta forma, utilizaremos las SVMs para varios problemas de clasificación, con objeto de que el alumno entienda mejor su funcionamiento y, sobre todo, el efecto de sus parámetros. Para ello, emplearemos la librería `libsvm`, que es una de las implementaciones más eficientes disponibles hoy en día, además de ser gratuita. La entrega se hará utilizando la tarea en Moodle habilitada al efecto. Se deberá subir un único fichero comprimido incluyendo los entregables que se indican en este guión. El día tope para la entrega es el **8 de diciembre de 2020**. En caso de que dos alumnos entreguen prácticas copiadas, no se puntuarán ninguna de las dos.

1. Introducción

En esta práctica, vamos a realizar distintos ejercicios para entender mejor el funcionamiento de las máquinas de vectores soporte (*Support Vector Machines*, SVMs) en problemas de clasificación. Las SVMs se han convertido, desde hace algunos años, en el estado del arte en problemas de reconocimiento de patrones y es importante conocer su funcionamiento y como responden a sus distintos parámetros. La práctica se realizará utilizando la librería `libsvm`, que es una de las implementaciones más populares y eficientes que se disponen actualmente. En todos los casos, consideraremos la versión "C-SVC", incluida en `libsvm`.

La sección 2 describe una serie de experimentos a realizar utilizando bases de datos sintéticas, que nos ayudarán a entender mejor el funcionamiento de las SVMs y de sus parámetros. La sección 3 describe otros experimentos a realizar sobre bases de datos reales. Finalmente, la sección 4 especifica los ficheros a entregar para esta práctica.

Se deben contestar todas las preguntas que vengan en recuadros de este estilo.

2. Bases de datos sintéticas

En esta primera parte de la práctica, utilizaremos una serie de experimentos sobre bases de datos sintéticas, que nos ayudarán a entender cómo funcionan las SVMs y qué efectos producen sobre los resultados los dos parámetros que hay que especificar para entrenarlas.

2.1. Instalación de libsvm

Para esta práctica vamos a utilizar Python. `scikit-learn` ya trae un clasificador que aplica el algoritmo SVC, utilizando la implementación de `libsvm`. En concreto, el clasificador es `sklearn.svm.SVC`.

2.2. Representación 2D de las SVMs

Para realizar la representación gráfica en dos dimensiones desde Python, vamos a utilizar el *script* `libsvm.py`.

Pregunta [1]: Abre este *script* y explica su contenido. Podrás ver que se utiliza el primer *dataset* de ejemplo y se realiza la representación gráfica del SVM. Comenta que tipo de *kernel* se está utilizando y cuáles son los parámetros de entrenamiento. Explica todos los elementos de la SVM que aparecen en la imagen, junto con los colores.

2.3. Primer dataset de ejemplo

Ejecuta el *script* mencionado anteriormente pero comenta las líneas correspondientes a la representación gráfica de la SVM, para ver solo los puntos del *dataset* (representación similar a la de Figura 1, `dataset1.csv`).

Pregunta [2]: Intuitivamente, ¿qué hiperplano crees que incurrirá en un menor error de *test* en la tarea de separar las dos clases de puntos?.

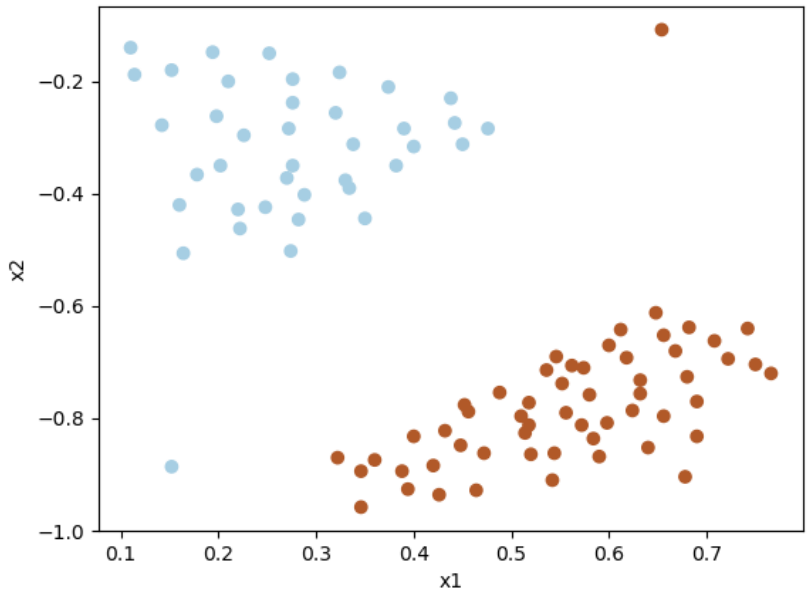

Figura 1: Dataset sintético o tipo *toy* #1.

En esta parte del ejercicio, vamos a utilizar una SVM lineal, es decir, una SVM donde la separación de las dos clases será una línea recta. La forma de especificar los parámetros del algoritmo SVC en Python puede consultarse en `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html`.

Vamos a probar a utilizar distintos valores para el parámetro C. De manera informal, podemos decir que el parámetro C es un parámetro que define cuánto vamos a penalizar el hecho de que se cometan errores en la clasificación. Un valor de C muy grande resultará en una SVM que cometerá el mínimo número posible de errores, mientras que un valor pequeño de C favorecerá el que se obtenga un clasificador con el máximo margen, aunque se cometan errores. Por supuesto, esto está relacionado con el posible sobre-entrenamiento (aprendizaje de patrones ruidosos de

entrenamiento que puedan despistar al modelo).

Pregunta [3]: Modifica el *script* probando varios valores de C, en concreto, $C \in \{10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}, 10^{3}, 10^{4}\}$. Observa qué sucede, explica porqué y escoge el valor más adecuado.

2.4. Segundo dataset de ejemplo

En este caso, trabajaremos con el *dataset* de la Figura 2 (`dataset2.csv`).

Pregunta [4]: Prueba a lanzar una SVM lineal con los valores para C que se utilizaron en la pregunta anterior. ¿Consigues algún resultado satisfactorio en el sentido de que no haya errores en el conjunto de entrenamiento?. ¿Por qué?.

Para conseguir hiperplanos de separación no lineales, las SVMs utilizan el llamado *truco del kernel* que, explicado de manera informal, construye un modelo lineal en un espacio de muchas dimensiones y lo proyecta de nuevo al espacio original, resultando en un modelo no lineal. Esto se consigue mediante el uso de funciones de *kernel* aplicadas sobre cada uno de los patrones de entrenamiento. El *kernel* más común es el *kernel* RBF, también llamado Gaussiano. Para poder aplicarlo, el usuario tiene que especificar un parámetro adicional (γ, gamma, en `libsvm` $\gamma = 1/(2 * radio^2)$). Un radio alto tiende a soluciones más suaves, con menor sobre-entrenamiento, mientras que un radio de *kernel* pequeño tiende a producir mayor sobre-entrenamiento.

Pregunta [5]: Propón una configuración de SVM no lineal (utilizando el *kernel* tipo RBF o Gaussiano) que resuelva el problema. Prueba a utilizar valores en el rango $C, \gamma \in \{10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}, 10^{3}, 10^{4}\}$. El resultado debería ser similar al de la Figura 3. ¿Qué valores has considerado para C y para γ?. Además, incluye un ejemplo de una configuración de parámetros que produzca sobre-entrenamiento y otra que produzca infra-entrenamiento.

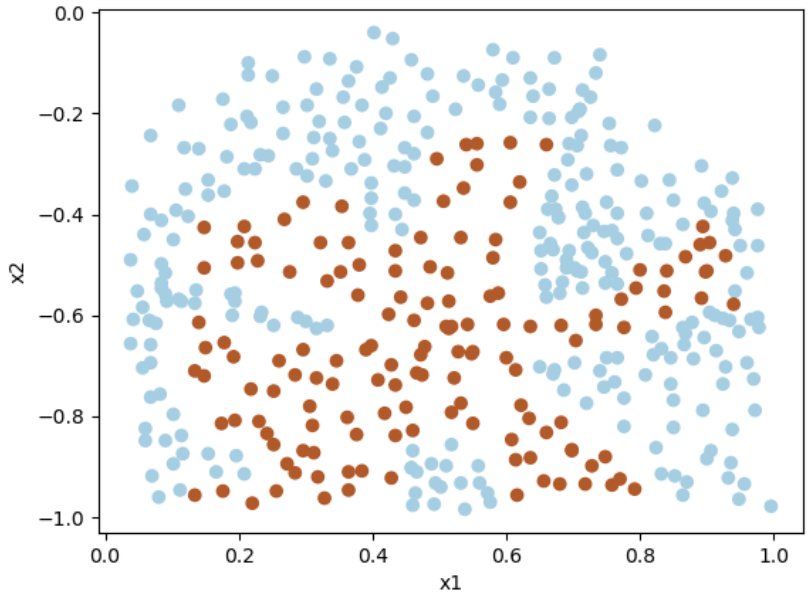

Figura 2: Dataset sintético o tipo *toy* #2.

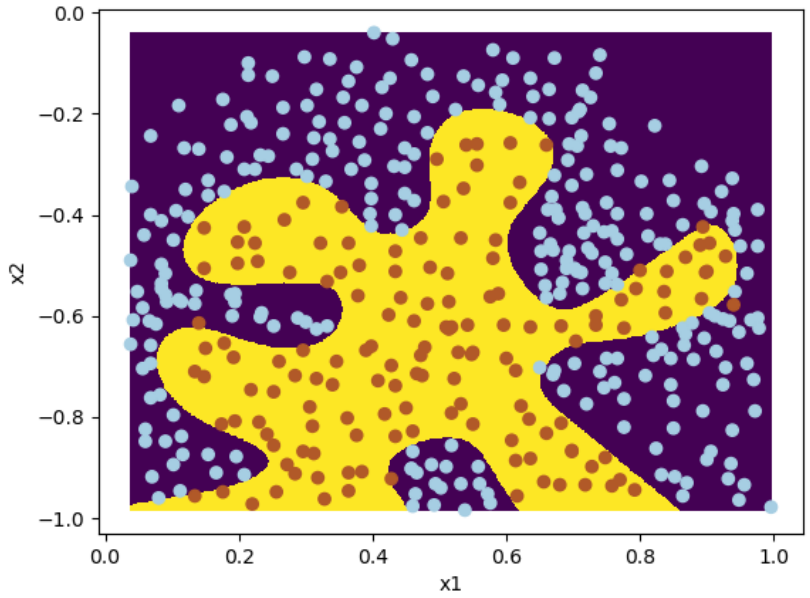

Figura 3: Dataset sintético o tipo *toy* #2 bien clasificado.

2.5. Tercer dataset de ejemplo

Por último, vamos a trabajar con el *dataset* de la Figura 4 (`dataset3.csv`).

Pregunta [6]: En este caso, ¿es el *dataset* linealmente separable?. A primera vista, ¿detectas puntos que presumiblemente sean *outliers*?, ¿por qué?.

Pregunta [7]: Lanza una SVM para clasificar los datos, con objeto de obtener un resultado lo más parecido al de la Figura 5. Ajusta los parámetros en el rango $C, \gamma \in \{10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}, 10^{3}, 10^{4}\}$. Establece el valor de los parámetros óptimos. Además, incluye un ejemplo de una configuración de parámetros que produzca sobre-entrenamiento y otra que produzca infra-entrenamiento.

2.6. Interfaz de consola

Aunque estamos utilizando la interfaz de Python, vamos a especificar algunas características de la interfaz de consola de `libsvm`[1], para conocer mejor el proceso de entrenamiento de una SVM. La interfaz de consola de `libsvm` está compuesta de tres ficheros ejecutables o programas, que pueden ser invocados desde la consola[2]:

1. `svm-scale`: estandariza el fichero de una base de datos, para que sea posible procesarlo con el clasificador SVM. Ten en cuenta que, utilizando un *kernel* tipo RBF, todas las variables de entrada deben estar en la misma escala, sino algunas variables recibirán más importancia que otras a la hora del cálculo de distancias.

2. `svm-train`: entrena un modelo SVM para un fichero de base de datos. El resultado del entrenamiento se guarda en un fichero con extensión `.model`.

[1] Disponible en `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`

[2] Consulte el fichero `README` de la raíz de `libsvm` para comprobar cómo funcionan y qué argumentos reciben cada uno de estos programas.

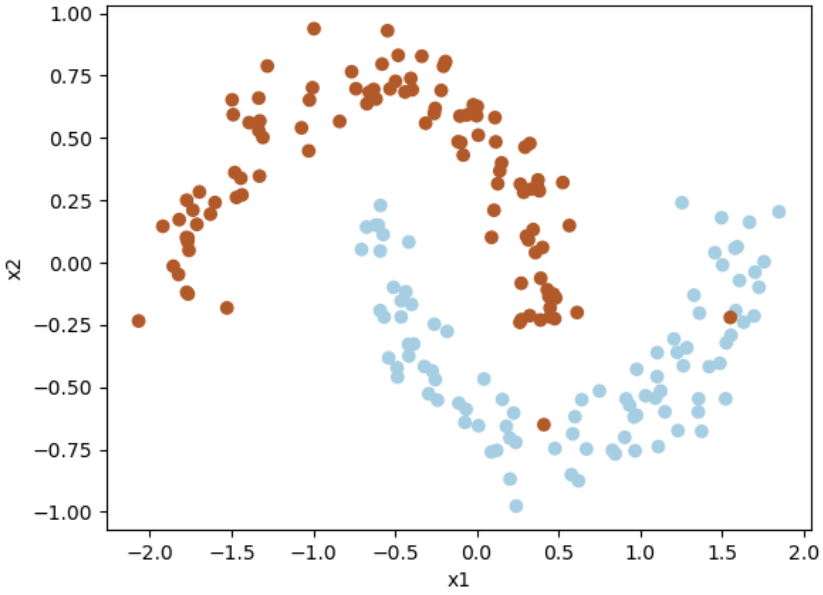

Figura 4: Dataset sintético o tipo *toy* #3.

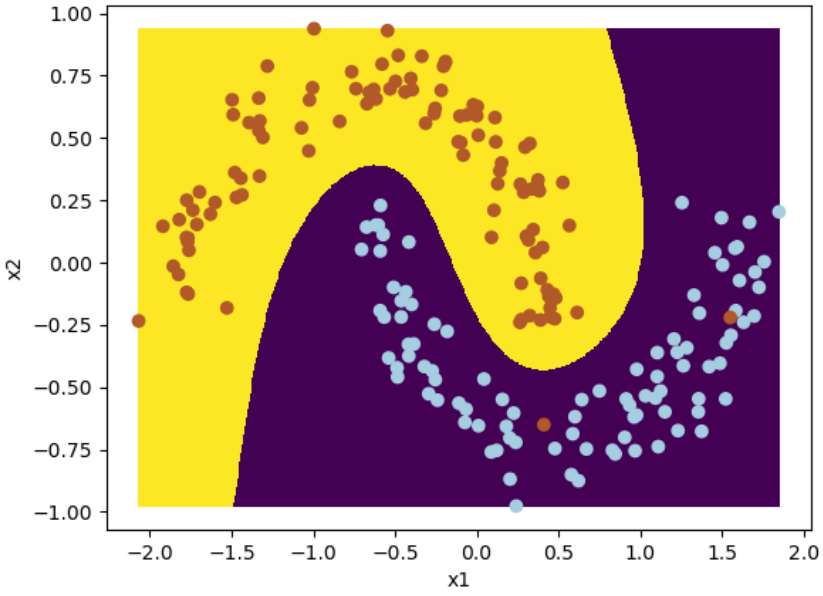

Figura 5: Dataset sintético o tipo *toy* #3 bien clasificado.

3. `svm-predict`: utiliza un modelo ya entrenado para ver como generaliza en una nueva base de datos (que debe tener la misma estructura). Es decir, realiza la fase de *test* del clasificador, comprobando cómo funciona para datos que no se han utilizado durante el entrenamiento.

Supongamos que tenemos dos ficheros de datos `train.libsvm` y `test.libsvm`. El proceso de entrenamiento estándar de una SVM implicaría, en primer lugar, entrenar la SVM utilizando `svm-train` y el fichero `train.libsvm`, generando un modelo SVM. En segundo lugar, utilizaríamos el programa `svm-predict` para ver que tal generaliza el modelo SVM sobre los datos `test.libsvm`.

Si utilizamos la clase `sklearn.svm.SVC` de `scikit-learn`, la estandarización tendremos que aplicarla nosotros, haciendo uso del objeto StandardScaler[3]. Aquí tienes un pequeño ejemplo de como se pueden estandarizar los datos:

```
from sklearn import preprocessing
scaler = preprocessing.StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Por otro lado, la clase `sklearn.model_selection.StratifiedShuffleSplit` realiza una o varias particiones de una base de datos de forma "estratificada", es decir, manteniendo la proporción de patrones de cada clase en la base de datos original[4]. Alternativamente, podéis utilizar el método `sklearn.model_selection.train_test_split`[5], que es algo más sencillo.

Pregunta [8]: Vamos a reproducir este proceso en Python. Divide el *dataset* sintético `dataset3.csv` en dos subconjuntos aleatorios de forma **estratificada**, con un 75 % de patrones en *train* y un 25 % de patrones en *test*. Realiza el proceso de entrenamiento completo (estandarización, entrenamiento y predicción), volviendo a optimizar los valores de C y γ que obtuviste en la última pregunta. Comprueba el porcentaje de buena clasificación que se obtiene para el conjunto de *test*. Repite el proceso más de una vez para comprobar que los resultados dependen mucho de la semilla utilizada para hacer la partición.

Este proceso tiene un pequeño inconveniente y es que es necesario especificar un valor para el parámetro C y un valor para el parámetro γ (ancho del *kernel*, para el caso en que queramos un clasificador no lineal) y, como ya viste en la primera parte de la práctica, estos parámetros son muy sensibles. Para evitar esto, existe una forma de ajustar automáticamente los parámetros, siguiendo el siguiente proceso de *validación cruzada anidada tipo K-fold*:

1. Realizamos una partición tipo K-fold (donde K es un parámetro) de los datos de entrenamiento. Es decir, dividimos los datos de entrenamiento en K subconjuntos disjuntos.

2. Para cada combinación de parámetros, realizamos K entrenamientos. Para cada entrenamiento k de los K totales:

   a) Utilizamos el subconjunto k como conjunto de *test* y el resto de subconjuntos como conjunto de entrenamiento.

   b) Acumulamos el error de *test* en una variable.

   Al terminar, calculamos el error medio cometido durante los K entrenamientos. El proceso de validación cruzada se repite completo para todas las combinaciones posibles de parámetros C y γ. Por ejemplo, si vamos a darle 9 valores a C y 9 valores a γ y vamos a realizar una validación cruzada de tipo 5-fold, esto implica que realizaremos $9 \times 9 \times 5 = 405$ entrenamientos. Nótese que hasta el momento no se han utilizado los datos de test.

[3] `http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html`

[4] `http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedShuffleSplit.html`

[5] `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html`

3. Escogemos la combinación de parámetros que resultó en el menor error medio en el paso anterior y tomamos esos valores de parámetros como los valores óptimos.

4. Utilizando esos parámetros, repetimos el proceso de entrenamiento pero ahora considerando el conjunto completo de entrenamiento (sin hacer subconjuntos).

5. Con el modelo obtenido en el paso anterior, comprobamos cuál es el error en test (fase real de *test*).

En `scikit-learn`, la búsqueda de parámetros la podemos realizar haciendo uso del objeto `sklearn.grid_search.GridSearchCV`[6]. Aquí tienes un pequeño ejemplo:

```
from sklearn.grid_search import GridSearchCV
Cs = np.logspace(-3, 3, num=7, base=10)
Gs = np.logspace(-3, 3, num=7, base=10)
optimo = GridSearchCV(estimator=svm_model, param_grid=dict(C=Cs,gamma=Gs),
                      n_jobs=-1,cv=5)
optimo.fit(X_train,y_train)
print optimo.score(X_test,y_test)
```

Pregunta [9]: Amplía el código anterior para realizar el entrenamiento de la pregunta 8 sin necesidad de especificar los valores de C y γ. Compara los valores óptimos obtenidos para ambos parámetros con los que obtuviste a mano. Extiende el rango de valores a explorar, si es que lo consideras necesario.

Pregunta [10]: ¿Qué inconvenientes observas en ajustar el valor de los parámetros "a mano", viendo el porcentaje de buena clasificación en el conjunto de test (lo que se hizo en la pregunta 8)?.

Pregunta [11]: Para estar seguros de qué has entendido como se realiza la búsqueda de parámetros, implementa de forma manual (sin usar GridSearchCV) la *validación cruzada anidada tipo K-fold* expuesta en esta sección. Te puede ser útil el uso de listas por compresión y la clase `StratifiedKFold`. Compara los resultados con los que obtienes usando `GridSearchCV`.

3. Bases de datos reales

Una vez te has familiarizado con el clasificador SVM y con su versiones lineal y no lineal, en esta parte de la práctica vamos a abordar distintos problemas reales, con objeto de ver su aplicabilidad en bases de datos más complejas.

3.1. Base de datos *noMNIST*

Esta base de datos, originariamente, está compuesta por 200,000 patrones de entrenamiento y 10,000 patrones de *test*, y un total de 10 clases. No obstante, para la práctica que nos ocupa, se ha reducido considerablemente el tamaño de la base de datos para realizar las pruebas en menor tiempo. Por lo tanto la base de datos que se utilizará está compuesta por 900 patrones de entrenamiento y 300 patrones de *test*. Está formada por un conjunto de letras (de la a a la f) escritas con difererentes tipografías o simbologías. Están ajustadas a una rejilla cuadrada de 28×28 píxeles. Las imágenes están en escala de grises en el intervalo $[-1{,}0; +1{,}0]$.[7]. Cada uno de los píxeles forman parte de las variables de entrada (con un total de $28 \times 28 = 784$ variables de entrada) y las clases se corresponden con la letra escrita (a, b, c, d, e y f, con un total de 6

[6] `http://scikit-learn.org/stable/modules/generated/sklearn.grid_search.GridSearchCV.html`

[7] Para más información, consultar `http://yaroslavvb.blogspot.com.es/2011/09/notmnist-dataset.html`

clases). La figura 6 representa un subconjunto de los 180 patrones del conjunto de entrenamiento, mientras que la figura 7 representa un subconjunto de 180 letras del conjunto de *test*. Además, todas las letras, ordenadas dentro de cada conjunto, está colgadas en la plataforma Moodle en los ficheros `train_img_nomnist.tar.gz` y `test_img_nomnist.tar.gz`.

Figura 6: Subconjunto de letras del conjunto de entrenamiento.

Figura 7: subconjunto de letras del conjunto de *test*.

Pregunta [12]: Utiliza el *script* que desarrollaste en la pregunta 9 para entrenar esta base de datos. Observe el valor de CCR obtenido para el conjunto de generalización y compáralo con el obtenido en prácticas anteriores. El proceso puede demorarse bastante. Al finalizar, toma nota de los valores óptimos obtenidos para los parámetros.

Pregunta [13]: Localiza dónde se especifica el valor de K para la validación cruzada interna y el rango de valores que se han utilizado para los parámetros C y γ. ¿Cómo podrías reducir el tiempo computacional necesario para realizar el experimento?. Prueba a establecer $K = 3$, $K = 5$ y $K = 10$ y compara, utilizando una tabla, los tiempos computacionales obtenidos y los resultados de CCR en test.

3.2. Base de datos clasificación de *spam*

Uno de los campos donde se utiliza el aprendizaje automático con un mayor éxito es la detección automática de *spam* en servidores de correo. Como dato significativo, ten en cuenta que el correo que recibes cada día en tu cuenta de la Universidad de Córdoba (UCO) es solo un $5\,\%$ de los correos reales que te llegan, de manera que el $95\,\%$ restante es detectado y descartado automáticamente por filtros anti-*spam*. La mayoría de estos filtros utilizan internamente técnicas de clasificación binaria (correo tipo *spam*, $y = 1$, o no *spam*, $y = 0$) y cada vez está ganando mayor aceptación el uso de SVM. En esta sección, vamos a explicar los pasos que se llevan a cabo para preprocesar el texto de un correo y convertirlo en un vector de características que podamos utilizar como entrada en cualquier clasificador. Posteriormente, se te pedirá que utilices SVM para clasificar una base de datos real.

Imaginemos un correo cuyo cuerpo sea como el de la Figura 8. Podemos observar cómo este correo contiene una URL, una dirección de correo electrónico (al final), números y una cantidad

```
> Anyone knows how much it costs to host a web portal ?
>
Well, it depends on how many visitors youre expecting. This can be
anywhere from less than 10 bucks a month to a couple of $100. You
should checkout http://www.rackspace.com/ or perhaps Amazon EC2 if
youre running something big..
To unsubscribe yourself from this mailing list, send an email to:
groupname-unsubscribe@egroups.com
```

Figura 8: Ejemplo de correo

en dólares. Muchos e-mails tendrán este tipo de cosas (direcciones de correo, URLs) pero, seguramente, el valor concreto (la URL concreta o el correo concreto) serán diferentes en cada e-mail. Normalmente, para la tarea concreta de clasificación (es *spam* o no), te interesa el hecho de que el correo contenga una dirección URL o un e-mail o un número, pero no su valor concreto. Por tanto, el proceso que vamos a seguir "normaliza" todas las URLs para que se traten del mismo modo, al igual que todos los números, todas las direcciones de correo, etc. Para ello, cada vez que aparezca en nuestro correo un dirección `http`, la reemplazaremos por la palabra reservada `httpaddr` que indica que había una dirección. Esto está demostrado que mejora el porcentaje de buena clasificación en detección de *spam*, porque casi siempre los *spammers* cambian de forma aleatoria las direcciones URL (y así es más difícil detectarlos).

Se te va a proporcionar una base de datos de correos que es parte del repositorio *SpamAssassin Public Corpus*[8]. Cada correo de esta base de datos ha recibido el siguiente preprocesamiento:

1. Todo el correo ha sido escrito en letras minúsculas (es decir, "`InDICAtE`" se va a tratar igual que "`Indicate`" o que "`indicate`"). A menudo, los *spammers* usan esta técnica para evitar ser detectados.

2. Eliminar todo código HTML: todos los correos serán procesados como texto plano, de manera que se elimina cualquier etiqueta de formato de tipo HTML.

3. Normalizar las URL: todas las URL se reemplazan por `httpaddr`.

4. Normalizar las direcciones de e-mail: todos las direcciones de e-mail se reemplazan por `emailaddr`.

5. Normalizar los números: todos los números se reemplazan por `number`.

6. Normalizar los dólares: todos los signos de dolar ($, asociados con frecuencia a correos de *spam*) se reemplazan por `dollar`.

7. *Stemming* de palabras: el *stem* de una palabra es su raíz. Para la detección de *spam* es buena idea utilizar las raíces de las palabras, en lugar de las palabras completas. Por ejemplo, las palabras "`discount`", "`discounts`", "`discounted`" y "`discounting`" se reemplazan todas por "`discount`". Hay programas (*stemmers*) que realizan este proceso de manera automática, utilizando para ello diccionarios específicos.

8. Quitar todo lo que no son palabras, por ejemplo, signos de puntuación o símbolos poco frecuentes. Además, todos los espacios, tabuladores o saltos de líneas que vayan consecutivos son sustituidos por un solo carácter de espacio.

El resultado de todos estos pasos sobre el correo anterior puede verse en la Figura 9. Con esta representación del correo es mucho más fácil extraer características que transformen el correo en un vector de números.

[8] http://spamassassin.apache.org/publiccorpus/

```
anyon know how much it cost to host a web portal well it depend on how mani visitor your
     expect thi can be anywher from less than number buck a month to a coupl of
    dollarnumb you should checkout httpaddr or perhap amazon ecnumb if your run someth
    big to unsubscrib yourself from thi mail list send an email to emailaddr
```

Figura 9: Ejemplo de correo ya preprocesado

Tras este paso, debemos elegir cuáles de las palabras que tenemos como resultado vamos a utilizar para clasificar *spam*. A esto se le llama lista de vocabulario. En el caso que nos ocupa, se han seleccionado 1899 palabras, que son las que aparecen con más frecuencia. Estas palabras están en el fichero `vocab.txt` y se muestran también en la Figura 10. Estas palabras son aquellas que aparecían al menos 100 veces en el total de e-mails de la base de datos.

```
1 aa
2 ab
3 abil
...
916 know
...
1898 zero
1899 zip
```

Figura 10: Lista de vocabulario

Utilizando esta lista de vocabulario, el e-mail de la Figura 9 queda reducido a un conjunto de índices que representan cada una de las palabras (suponiendo que la palabra esté en el vocabulario, sino, simplemente se ignora por ser una palabra muy rara). El resultado de esa transformación puede verse en la Figura 11.

```
86 916 794 1077 883 370 1699 790 1822 1831 883 431 1171 794 1002 1893 1364 592 1676 238
    162 89 688 945 1663 1120 1062 1699 375 1162 479 1893 1510 799 1182 1237 810 1895
    1440 1547 181 1699 1758 1896 688 1676 992 961 1477 71 530 1699 531
```

Figura 11: Ejemplo de correo preprocesado y del que se han extraído los índices de las palabras.

Desde el punto de vista de la detección de *spam*, nos interesa si la palabra del vocabulario aparece, pero no en qué orden. Por tanto, el último paso transforma el vector de índices anterior en un vector de valores binarios. Para nuestro caso tendremos un vector $\mathbf{x} \in \mathbb{R}^{1899}$ y la posición i valdrá 1 ($x_i = 1$) si la palabra del i-ésima del vocabulario aparece en el e-mail ($x_i = 0$ si no aparece). Este vector será el vector que se use para clasificar si el correo es o no *spam*. Usando esta representación, da igual que las palabras aparezcan más de una vez.

Dispones de dos ficheros de datos donde todo este proceso ya se ha realizado. El fichero de entrenamiento contiene 4000 correos, mientras que el de *test* tiene 1000 correos. Ambos utilizan la lista de vocabulario de 1899 palabras contenida en `vocab.txt`. Por tanto, cada patrón tiene 1899 valores binarios.

Pregunta [14]: Debes entrenar un modelo **lineal** de SVM con valores $C = 10^{-2}$, $C = 10^{-1}$, $C = 10^{0}$ y $C = 10^{1}$. Para ello utiliza un *script* similar al que usaste para la pregunta 9. Compara los resultados y establece la mejor configuración.

Pregunta [15]: Para la mejor configuración, construye la matriz de confusión y establece cuáles son los correos en los que la SVM se equivoca. Consulta las variables de entrada para los correos que no se clasifican correctamente y razona el motivo. Ten en cuenta que para cada patrón, cuando $\mathbf{x}_i$ es igual a 1 quiere decir que la palabra i-ésima del vocabulario aparece, al menos una vez, en el correo.

Pregunta [16]: Compara los resultados obtenidos con los resultados utilizando una red RBF. Para ello, haz uso del programa desarrollado en la práctica anterior. Utiliza solo una semilla (la que mejor resultado obtenga).

Pregunta [17]: Entrena una SVM no lineal y compara los resultados.

4. Entregables

Los ficheros a entregar serán los siguientes:

- Memoria de la práctica en un fichero `pdf` que conteste a todas las cuestiones que se han ido planteando en este guión. Incluye las tablas o imágenes que consideres necesarias para demostrar que se han realizado los ejercicios que se piden. La memoria de la práctica deberá incluir, al menos, el siguiente contenido:
  - Portada con el número de práctica, título de la práctica, asignatura, titulación, escuela, universidad, curso académico, nombre, DNI y correo electrónico del alumno.
  - Índice del contenido de la memoria con numeración de las páginas.
  - Respuesta a todas las cuestiones y ejercicios que se plantean en este guión.
  - Referencias bibliográficas u otro tipo de material distinto del proporcionado en la asignatura que se haya consultado para realizar la práctica (en caso de haberlo hecho).
- Un fichero comprimido con todos los *scripts*, ficheros de datos y ficheros de salida que hayas necesitado para contestar a las cuestiones.

Opcionalmente, podéis presentar tanto la memoria como el código de forma conjunta en un cuaderno de Jupyter.