APRENDIZAJE AUTOMÁTICO: PROYECTO FINAL

RAFAEL HERRERA TROCA
RUBÉN RUPERTO DÍAZ

APRENDIZAJE AUTOMÁTICO Y BIG DATA
DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS

Proyecto final de la asignatura
Curso 2019/2020

7 de julio de 2020

Índice general

Capítulo 1

Introducción y análisis del conjunto de datos

Este proyecto consiste en la elección de un conjunto de datos para clasificarlo usando algunas de las técnicas de aprendizaje automático estudiadas durante la asignatura.

Hemos elegido el conjunto de datos *Mushroom Classification*[1] de la página *Kaggle*. Los datos se encuentran en un archivo *CSV*, que consta de 8124 filas, correspondientes a cada elemento, y 23 columnas, correspondientes a los atributos de los champiñones. El atributo a predecir es si el champiñón es venenoso o no.

Los valores que pueden tomar los atributos son categóricos, por lo que para que los datos puedan ser utilizados por un algoritmo de aprendizaje automático debemos transformar estos valores a números. Para ello, utilizamos una codificación *one hot*, que consiste en dividir cada uno de los atributos en tantas variables binarias como valores puede tomar, y poner todas estas nuevas variables a 0 excepto aquella que indica el valor original del atributo. En cuanto al atributo a predecir, puesto que sólo tiene dos valores posibles, los codificaremos dando el valor 1 si el champiñón es comestible y 0 si no lo es. En un principio se tienen 23 atributos, incluyendo el que se pretende predecir, pero tras la codificación *one hot*, se transforman en 118, incluyendo también el atributo a predecir

Tras la codificación, dividimos los 8124 ejemplos que tiene el conjunto de datos en tres grupos. El primero será el de los datos de entrenamiento y tendrá el 60 % de los ejemplos, que usaremos para entrenar los modelos. El segundo grupo será el validación y tendrá el 20 % de los ejemplos, que usaremos para determinar qué parámetros son los óptimos para el modelo y comprobar si se ha producido sesgo o varianza. El 20 % restante lo utilizaremos como datos de test, para calcular los resultados definitivos obtenidos por el modelo con los parámetros óptimos. Esta separación en

[1] «Mushroom Classification». [Online]. Disponible en: `https://kaggle.com/uciml/mushroom-classification`. [Accedido: 19-jun-2020]

tres grupos la hacemos aleatoriamente mediante la función `train_test_split` de la librería de *Python* `sklearn.model_selection`, sin embargo, fijamos la semilla para que el resultado sea igual en cualquier ejecución del código.

Ya que el objetivo del dataset es clasificar los ejemplos en función de un atributo binario, hemos decidido emplear regresión logística regularizada, redes neuronales y maquinas de soporte de vectores, que están especialmente indicadas para este tipo de problemas. Hemos descartado el uso de regresión lineal o polinomial, ya que esta está pensada para encontrar la recta o curva que mejor se ajusta a un conjunto de puntos, y no es adecuada para un problema de clasificación.

1.1. Análisis de los datos

Hemos realizado varios diagramas de barras en los que se representan los posibles valores de cada atributo de los champiñones y el número de ejemplos venenosos y comestibles que hay para cada valor. Esto nos ha permitido observar que hay ciertas variables que permiten distinguir con claridad si un champiñón va a ser venenoso o no. El ejemplo más representativo es la variable *odor*, cuya gráfica se muestra en la figura 1.1.

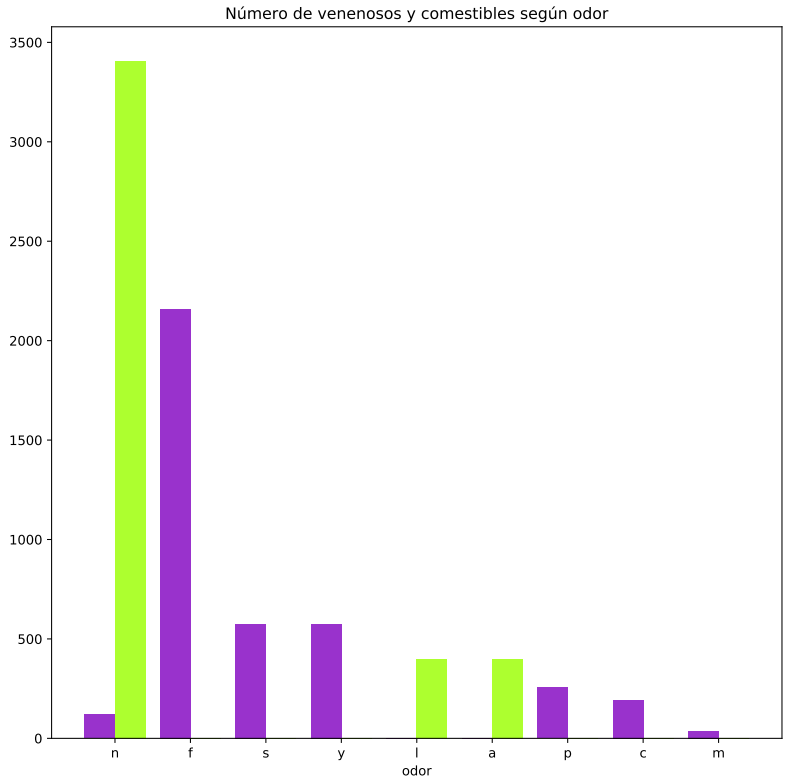

Figura 1.1: Número de champiñones venenosos (morado) y comestibles (verde) en función del olor del champiñón.

Observamos que sólo sabiendo el olor ya se puede clasificar prácticamente cualquier champiñón. Como consecuencia, los modelos que probaremos obtendrán tasas de acierto muy elevadas.

El resto de variables no son tan determinantes a la hora de clasificar, y presentan ejemplares tanto comestibles como venenosos en la mayoría de sus posibles valores. Por ello, probaremos a clasificar el conjunto de datos sin la variable *odor* y compararemos ambos resultados.

Para entender mejor cómo de relacionadas están las variables con las que vamos a trabajar, aplicamos un PCA sobre el conjunto de datos y estudiamos la varianza explicada. Comprobamos que con 31 variables se alcanza un 90 %, con 56, un 99 % y con 79, se explica el 100 % de la varianza. Esto nos lleva a la conclusión de que las variables están muy correlacionadas entre sí, ya que con sólo 31 de las 118 variables se recoge un 90 % de la información y, con 79, el 100 %.

Si aplicamos el PCA después de haber eliminado la variable *odor*, que como ya vimos en la figura 1.1 está muy relacionada con nuestra variable objetivo, comprobamos que con 29 variables se explica el 90 % de la varianza, con 53, el 99 % y que basta con 81 para tener el 100 % de la varianza explicada y, por tanto, de la información. Vemos que hacen falta más variables para conseguir el 100 % de la varianza, lo cual tiene sentido ya que al eliminar el olor hemos quitado una variable que proporcionaba mucha información. Además, hay que tener en cuenta que, como esta variable puede tomar 8 valores, ahora trabajamos con 8 variables menos, por lo que en realidad, antes hacían falta un 66,95 % de las variables para conseguir condensar toda la información y ahora es necesario un 73,64 %, lo cuál hace más patente la información que añadía esta variable antes.

Para terminar el análisis, hemos probado a aplicar una técnica de aprendizaje no supervisado, el algoritmo de *clustering K-Means*, ya que al ser no supervisado nos proporcionará información sobre la forma de los datos. Como sabemos que el resultado esperado es que haya dos *clusters*, no hemos probado con diferentes valores para K como se suele hacer y lo hemos aplicado directamente con $K = 2$. Lo hemos entrenado con los datos de entrenamiento, después hemos escogido la orientación de las etiquetas atendiendo al porcentaje de acierto sobre los datos de validación y, finalmente, hemos calculado el porcentaje de acierto sobre los datos de test.

Al aplicarlo sobre los datos originales (sin eliminar la variable *odor*), llega a obtener un porcentaje de acierto del 89,23 % sobre los datos de test, mientras que sobre los datos sin la variable *odor* se consigue un 72,8 %. Como *K-Means* sólo se basa en la distancia entre los atributos para clasificar los puntos y no todos los atributos guardan relación con la variable que queremos averiguar, el porcentaje no resulta tan elevado como el que obtendremos con otras técnicas. Sin embargo, tampoco es tan bajo como el que se obtendría si no hubiera relación alguna entre nuestra variable objetivo y las demás, lo cual corrobora que existe una fuerte correlación hará que las técnicas de aprendizaje supervisado resultasen extremadamente efectivas. Al quitar la variable *odor*, que como hemos visto en la figura 1.1 tiene una fuerte correlación

con la toxicidad, el porcentaje de acierto de *K-Means* desciende considerablemente, casi en un 20 %, pero sigue siendo suficientemente alto como para denotar la correlación entre el resto de variables y el objetivo, lo que explica que, como veremos, incluso después de haber quitado esta variable, las técnicas de aprendizaje supervisado que hemos aplicado siguen obteniendo un 100 % de acierto.

Capítulo 2

Regresión logística

2.1. Procedimiento

Para esta parte de la práctica usaremos las funciones que tenemos ya implementadas de la práctica 2, en la que también utilizábamos regresión logística. A continuación pasamos a explicar detalladamente estas funciones.

En primer lugar, definimos la función sigmoide que, dado un cierto valor, lo transforma para que esté entre 0 y 1. Esta función es además la que utilizaremos para que el valor devuelto por el modelo logístico sea interpretable: positivo (comestible) si es mayor que 0,5 y negativo (venenoso) si no lo es. La función sigmoide se define como sigue:

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

También hemos definido la función de coste para la regresión logística regularizada, en el método `P1coste`, utilizando la siguiente fórmula:

$$J(\theta) = -\frac{1}{N}((log(g(X\theta)))^T y + (log(1 - g(X\theta)))^T (1 - y)) + \frac{\lambda}{2N}\sum_{j=1}^{n}\theta_j^2$$

Esta función se presenta en su forma vectorizada, que es como la hemos implementado en el código, haciendo todas las operaciones de forma literal tal y como aparecen en la fórmula. El hecho de que esté vectorizada hace que podamos trabajar directamente con los *arrays* de la librería *Numpy* aprovechando todo su potencial, en lugar de ir haciendo los cálculos elemento a elemento. Esto supone un importante ahorro de tiempo a la hora de entrenar la regresión.

Para calcular el gradiente de la función de coste, también de forma vectorizada, utilizamos el método `P1gradiente`, que emplea la fórmula que se define a continuación:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{N}X^T(g(X\theta) - y) + \frac{\lambda}{N}\theta_j$$

Del mismo modo que sucedía con la función de coste, hemos implementado el gradiente también en su forma vectorizada, calculando todos sus elementos a la vez, de manera que tras aplicar la función se obtiene un array con todos ellos. Hay que tener especial cuidado porque el término final de la fórmula sólo se suma cuando $j \neq 0$. Para implementarlo, sumamos un array que tiene en cada posición el valor de θ_j correspondiente y, en la primera, un 0.

Una vez tenemos estas dos funciones, basta utilizar `fmin_tnc`, del paquete `Scipy.optimize`, para obtener los valores de θ que hacen mínimo el coste. Estos valores serán los que definan nuestro modelo, de forma que, dado un nuevo dato X, la predicción para éste viene dada por $g(X\theta)$. Probaremos a entrenar el modelo con distintos valores del parámetro de regularización λ para estudiar como va variando y a representar el error obtenido con cada uno. Para obtener dicho error utilizaremos la función de coste con $\lambda = 1$. Para escoger el mejor valor para el parámetro de regularización, compararemos el error obtenido con cada uno para el conjunto de datos de validación, distinto del de entrenamiento, quedándonos con el que menos error haya producido. Finalmente, comprobamos el porcentaje de acierto obtenido sobre un tercer conjunto de datos, con el fin de que este porcentaje sea completamente independiente del proceso de entrenamiento. Los resultados concretos se explican en la sección correspondiente.

Para el cálculo del porcentaje de error del que hablábamos, utilizaremos una última función definida por nosotros, `P1porc_ac`, que devuelve, dados los datos de entrada, el valor de θ y la salida esperada, el porcentaje de acierto que se consigue.

Pasamos ahora a explicar los resultados obtenidos con esta técnica.

2.2. Resultados

Hemos aplicado la técnica de regresión logística regularizada al dataset de champiñones y hemos analizado sus resultados. En primer lugar, entrenamos el modelo para distintos valores de λ en el conjunto $\{10^i \mid i \in [-10, 3], i \in \mathbb{Z}\}$ y evaluamos su tasa de acierto al clasificar tanto los propios datos de entrenamiento como los de validación.

El parámetro de regularización λ influye en la importancia que tiene el término de regularización de la función de coste $J(\theta)$ a la hora de minimizarla. Si el valor de λ es pequeño, el término de regularización no será muy relevante, y por tanto a la hora de minimizar $J(\theta)$ lo que tendrá más peso es reducir el error cometido al clasificar los datos de entrenamiento. Esto puede provocar que el modelo se sobreajuste a estos datos de entrenamiento, siendo capaz de clasificarlos con una tasa de acierto muy elevada, pero fallado a la hora de clasificar datos nuevos. Es lo que se conoce como varianza o *variance* en inglés. Por otro lado, para valores grandes de λ ocurre lo contrario. El término de regularización gana peso en la función de coste $J(\theta)$, lo que obliga a reducir el valor de las componentes de θ para compensarlo. Al hacer esto,

se limita la libertad del modelo para elegir θ y por tanto se restringe la posibilidad de que se ajuste demasiado a los datos de entrenamiento. Sin embargo, si se elige un λ demasiado grande, se puede restringir tanto el modelo que no sea capaz de aprender correctamente las relaciones entre las variables y el atributo a predecir, y por tanto falle al clasificar tanto los datos de entrenamiento como los datos nuevos. Esta situación se conoce como sesgo o *bias* en inglés.

La figura 2.1 muestra los resultados obtenidos al evaluar el modelo entrenado con los distintos valores de λ. La imagen de la izquierda muestra el porcentaje de casos predichos correctamente, siendo la línea roja la correspondiente a los datos de entrenamiento y la azul la de los datos de validación. La imagen de la derecha muestra el valor de la función $J(\theta)$ sin termino de regularización, que se corresponde con el error cometido durante la predicción.

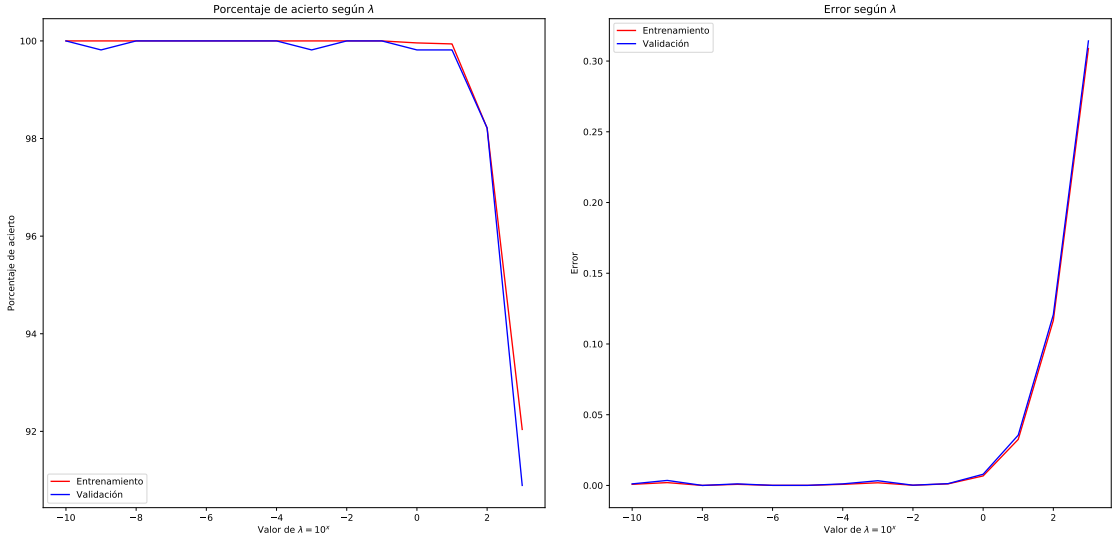

Figura 2.1: Resultados de la regresión logística en función del parámetro de regularización λ

Observamos que el modelo obtiene resultados excelentes a la hora de clasificar los datos para todos los valores de λ probados. Para valores de $\lambda \leq 100$, la tasa de acierto se mantiene en el 100 % o prácticamente en el 100 % tanto con los datos de entrenamiento como con los de validación. Esto muestra que no se produce sobreaprendizaje con valores pequeños de λ, puesto que de ser así percibiríamos un éxito al clasificar datos de entrenamiento mucho mayor que al clasificar los de validación. A partir de $\lambda = 100$, la tasa de aciertos baja tanto para los datos de entrenamiento como para los de validación. Para $\lambda = 100$ queda sobre el 99 % y para $\lambda = 1000$ cae hasta el 92 %. Esto sugiere que se empieza a producir algo de sesgo, y que valores de λ tan grandes entorpecen al modelo el aprendizaje de las relaciones entre la toxicidad del champiñón y sus características. En cuanto al error, lógicamente es más pequeño cuanto más alta es la tasa de acierto, mientras que crece ligeramente cuando la tasa de acierto baja, por lo que nos conduce a las mismas conclusiones.

El hecho de que se alcancen tasas de éxito tan elevadas al clasificar los datos de validación para casi todos los valores de λ probados confirma que es bastante sencillo clasificar los champiñones del dataset como comestibles o venenosos. En efecto, como vimos en el capítulo 1, la variable *odor* ya permite clasificar la gran mayoría de ejemplos como venenosos o comestibles.

Fijando el valor de λ a 10^{-8}, para el que se minimiza el error con los datos de validación, volvemos a evaluar el modelo con los datos de test, obteniendo una tasa de acierto del 100 %.

2.3. Resultados sin la variable *odor*

Tras eliminar la variable *odor* del dataset, esperábamos que el modelo no fuera capaz de legar al 100 % de acierto, pero sin embargo sí fue capaz de alcanzarlo. A continuación se muestran los resultados obtenidos.

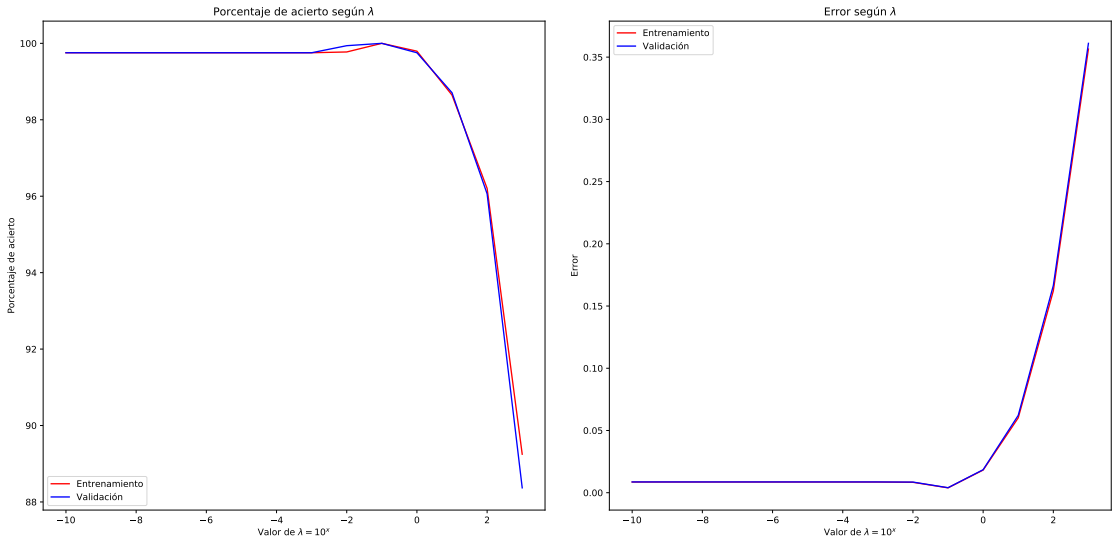

Figura 2.2: Resultados de la regresión logística tras eliminar la variable *odor* en función del parámetro λ

La figura 2.2 muestra resultados muy similares a los obtenidos en la sección anterior. En esta ocasión, los valores de $\lambda < 0{,}1$ no han llegado al 100 % de acierto, pero se han quedado muy cerca. Tampoco parece haber varianza con estos valores, ya que la tasa de aciertos con los datos de entrenamiento es exactamente igual que con los de validación. Con $\lambda = 0{,}1$ se alcanza la máxima tasa de acierto, que es del 100 % para entrenamiento y validación. A partir de ahí, se empieza a producir un ligero sesgo que hace que la tasa de acierto disminuya hasta un 89 % para datos de entrenamiento y un 88 % para los de validación cuando $\lambda = 1000$. Al clasificar los datos de test con el parámetro óptimo $\lambda = 0{,}1$ se vuelve a obtener un 100 % de acierto, por lo que quitar la variable *odor* no empeora los resultados.

Capítulo 3

Red neuronal

3.1. Procedimiento

Comenzamos por explicar las funciones más sencillas que hemos definido, mientras que las funciones más complejas para el cálculo del coste y su gradiente se desarrollarán en secciones posteriores.

En primer lugar, volvemos a utilizar la función `sigmoide` para transformar el resultado devuelto por una capa de la red en un valor en el intervalo $[0, 1]$, y que recordamos que viene definida según la fórmula:

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

La segunda función definida es la derivada de la sigmoide, `diffSigmoide`, cuya definición para una capa l de la red, siendo $a^{(l)}$ la entrada de dicha capa y $z^{(l)}$ la salida de la capa $l - 1$, viene dada por la fórmula:

$$g'(z^{(l)}) = a^{(l)}(1 - a^{(l)})$$

Si tenemos en cuenta que $a^{(l)} = [1 \sim g(z^{(l)})]$, esto es, calcular la sigmoide del resultado devuelto por la capa anterior y poner una columna de unos delante (normalmente referida como $a_0^{(l)}$). Podemos expresar la derivada de la sigmoide según esta otra fórmula:

$$g'(z^{(l)}) = [1 \sim g(z^{(l)})](1 - [1 \sim g(z^{(l)})])$$

No obstante, por ser la implementación mucho más sencilla de este modo, en el código se ha utilizado la primera fórmula.

También hemos definido las funciones `P2applyLayer` y `P2applyNet` que, como su nombre indica, aplican una capa de la red dada la entrada y la matriz de pesos correspondientes, o la red entera dadas la entrada y un array con las matrices de pe-

sos correspondientes a cada capa. Además, *P2applyNet* no sólo devuelve el resultado de aplicar la red, también devuelve la entrada utilizada en cada una de las capas, ya que necesitábamos este dato para calcular el gradiente de la función de coste.

Finalmente, también hemos definido las funciones `P2randomWeights`, que inicializa una matriz de pesos aleatorios para las dimensiones indicadas, y `P2porc_ac`, que calcula el porcentaje de acierto dadas la respuesta y las etiquetas correctas.

A continuación se explica con mayor grado de detalle el desarrollo de la función de coste y su gradiente.

3.1.1. Función de coste

Para la función de coste hemos aplicado la fórmula correspondiente manteniendo la generalidad todo lo posible. Dicha fórmula viene dada por:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2N}\sum_{l=1}^{L}\left[\sum_{i,j>0}(\Theta_{i,j}^{(l)})^2\right]$$

En esta fórmula, N indica el número de casos, K, el número de etiquetas y L, el número de matrices de pesos que tiene la red. Asimismo, y es la salida esperada de la red neuronal, x es la entrada, $h_\theta(x)$ es el resultado devuelto, λ es el parámetro de regularización y Θ es la matriz de pesos de una cierta capa. Además, el sumatorio que aparece al final del término de regularización expresa que se debe sumar el cuadrado de todos los términos de cada matriz de pesos teniendo en cuenta no sumar la primera columna (recordemos que el índice de las filas empieza en uno siempre).

En cualquier caso, para aprovechar la potencia del cálculo matricial que ofrece Python, hemos vectorizado la función de coste, evitando así tener que hacer el cálculo de la primera parte para cada uno de los casos de prueba. La fórmula que hemos utilizado es la siguiente:

$$J(\theta) = \frac{1}{N}\sum_{i,j}\left[y \odot \log(h_\theta(x)) - (1 \ominus y) \odot \log(1-h_\theta(x))\right]_{i,j} + \frac{\lambda}{2N}\sum_{l=1}^{L}\left[\sum_{i,j>0}(\Theta_{i,j}^{(l)})^2\right]$$

En esta fórmula, las letras vuelven a significar lo mismo que en la fórmula anterior, el término $(1 \ominus y)$ representa la matriz M dada por $M_{i,j} = 1 - y_{i,j}$, y el símbolo $\odot$ indica el producto elemento a elemento de dos matrices o producto de Hadamard.

3.1.2. Gradiente de la función de coste

Para el cálculo del gradiente implementamos la función `P2gradiente`, que lo calcula, nuevamente, de la forma más genérica posible para que se pueda utilizar para una red neuronal cualquiera sin tener que adaptarlo. Recibe como argumentos la salida esperada para la red neuronal, la entrada de cada capa, el resultado de aplicar la red, la lista de matrices de pesos y el parámetro de regularización, y devuelve la lista de matrices de gradiente.

Para obtener este resultado vamos calculando la matriz del gradiente correspondiente a cada matriz de pesos empezando por la última y retrocediendo hasta llegar a la primera. Hacemos estos cálculos de forma vectorizada, procesando todos los casos a la vez, si bien cada matriz se calcula por separado en un bucle. La primera matriz que calculamos, que se corresponde a la última capa de la red, se obtiene aplicando las fórmulas siguientes:

$$\delta^{(L+1)} = a^{(L+1)} - y$$
$$D^{(L)} = \frac{(\delta^{(L+1)})^T a^{(L+1)} + \lambda \Theta^{(L)}_{\bullet, j>0}}{N}$$

El resto de matrices se calculan iterativamente con las fórmulas recursivas correspondientes, que para $0 < i < L$, donde L es el número de matrices de pesos que tiene la red, vienen dadas por:

$$\delta^{(i+1)} = (\delta^{(i+2)} \Theta(i+1)) \odot g'(z^{(i+1)})$$
$$D^{(i)} = \frac{(\delta^{(i+1)}_{\bullet, j>0})^T a^{(i)} + \lambda \Theta^{(i)}_{\bullet, j>0}}{N}$$

En estas fórmulas, N es el número de casos con el que se entrena la red, y es la respuesta esperada, $a^{(l)}$ es la entrada de la capa l de la red, z es la salida de la capa $l-1$, $\Theta^{(l)}$ es la matriz de pesos que pasa de la capa l a la $l+1$ y λ es el parámetro de regularización.

Pasamos ahora a explicar los resultados obtenidos al aplicar la red neuronal sobre nuestro dataset.

3.2. Resultados

La red neuronal que hemos utilizado consta de tres capas. La capa de entrada tiene 117 neuronas, tantas como atributos de los datos sin contar el atributo a predecir, mientras que la capa de salida tiene una única neurona que determina si el champiñón es venenoso o no. Hemos entrenado esta red con distintos valores para el parámetro de regularización λ y el número de iteraciones. Para λ, hemos probado

las potencias de 10 entre 10^{-6} y 10^{3}, y para el límite de iteraciones los múltiplos de 10 entre 10 y 100.

$$\lambda \in \{10^{i} \mid i \in [-6, 3], i \in \mathbb{Z}\} \quad iter \in \{n \in [10, 100] \mid n \equiv 0 \,(\text{mód } 10)\}$$

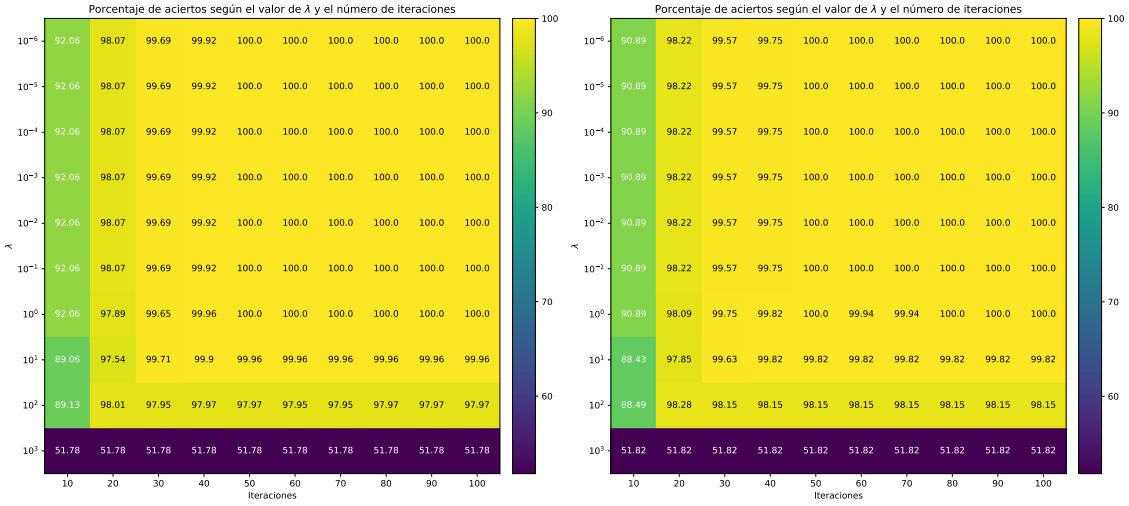

Figura 3.1: Resultados de la red neuronal con los datos de test (izquierda) y los de validación (derecha) en función del parámetro de regularización λ y el número de iteraciones

La figura 3.1 muestra que el porcentaje de aciertos de la red neuronal crece si permitimos un mayor número máximo de iteraciones. Esto se debe a que la función de minimización se va aproximando al mínimo progresivamente en cada iteración según le indica la dirección del gradiente. Si permitimos más iteraciones, nos aproximaremos más al mínimo de la función de coste y por tanto al mínimo error al predecir los datos de entrenamiento. Sin embargo, no es bueno minimizar excesivamente la función de coste puesto que, aunque permita acertar más con los datos de entrenamiento, puede perjudicar la predicción de datos nuevos si se produce sobreaprendizaje, haciendo que la red sólo prediga correctamente los datos de entrenamiento.

En nuestro caso, vemos que la red neuronal empieza a obtener buenos resultados con muy pocas iteraciones. Con tan solo 10 iteraciones la tasa de acierto de los datos de validación está por encima del 88 % para casi todos los valores de λ probados. A partir de 50 iteraciones ya se alcanza un 100 % de acierto para $\lambda \leq 1$. Observamos que, a pesar de aumentar el número de iteraciones, el modelo no sufre varianza (sobreaprendizaje), ya que no se aprecia una disminución de la tasa de acierto en los datos de validación.

Asimismo, se obtienen más aciertos para valores más pequeños de λ. Para entender esto, recordamos que en el término de regularización que se añade a la función de coste, λ aparece multiplicando al sumatorio de los valores que aparecen en las

matrices de pesos. Esto implica que, para minimizar la función de coste, mayores valores de λ fuerzan a elegir pesos más pequeños, mientras que λ más pequeños permiten mayor libertad al elegir los pesos. De este modo, con valores de λ pequeños la red tiene más flexibilidad para ajustarse a los datos de entrenamiento, y por tanto también existe la posibilidad de que aparezca varianza en el modelo si permitimos que la red se ajuste demasiado.

En este caso, no se llega a producir tal varianza, ya que la tasa de acierto para los datos de validación se mantiene en el 100 % a pesar de reducir el valor de λ. Sin embargo, sí que se produce sesgo con valores grandes de λ, especialmente en el caso de $\lambda = 1000$, lo que indica que con ese parámetro el modelo es incapaz de aprender las relaciones entre los datos, y como consecuencia ni siquiera se alcanza el 52 % de acierto. En realidad, no tiene sentido elegir un λ tan grande para este dataset, pero lo incluimos con fines académicos para mostrar un ejemplo de sesgo.

En general, la red neuronal obtiene tasas de acierto muy elevadas para la mayoría de elecciones de los parámetros, en muchos casos del 100 %. Además, no se llega a producir sobreajuste a los datos de entrenamiento. Todo ello nos confirma que la relación entre los atributos de los champiñones y su clasificación como venenosos o comestibles es muy clara, haciendo que el dataset sea muy sencillo de clasificar.

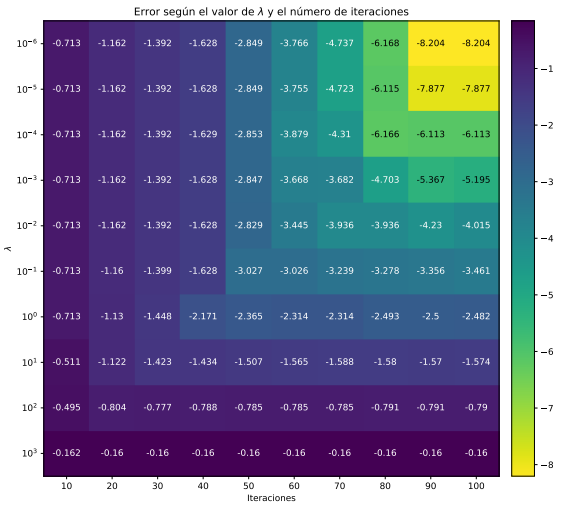

Figura 3.2: Error cometido por la red neuronal al clasificar los datos de validación en función del parámetro de regularización λ y el número de iteraciones. Se muestra el logaritmo en base 10 del error.

Tomamos $\lambda = 10^{-6}$ y 80 iteraciones, que son los parámetros que minimizan el error, dado por el valor de la función de coste $J(\theta)$ sin término de regularización (ver figura 3.2. Al clasificar los datos de test con estos parámetros, se obtiene una tasa de acierto del 100 %.

3.3. Resultados sin la variable *odor*

Los resultados obtenidos al entrenar la red neuronal sin la variable *odor* son prácticamente iguales que los obtenidos al tener la variable. Estos se pueden ver el la figura 3.3.

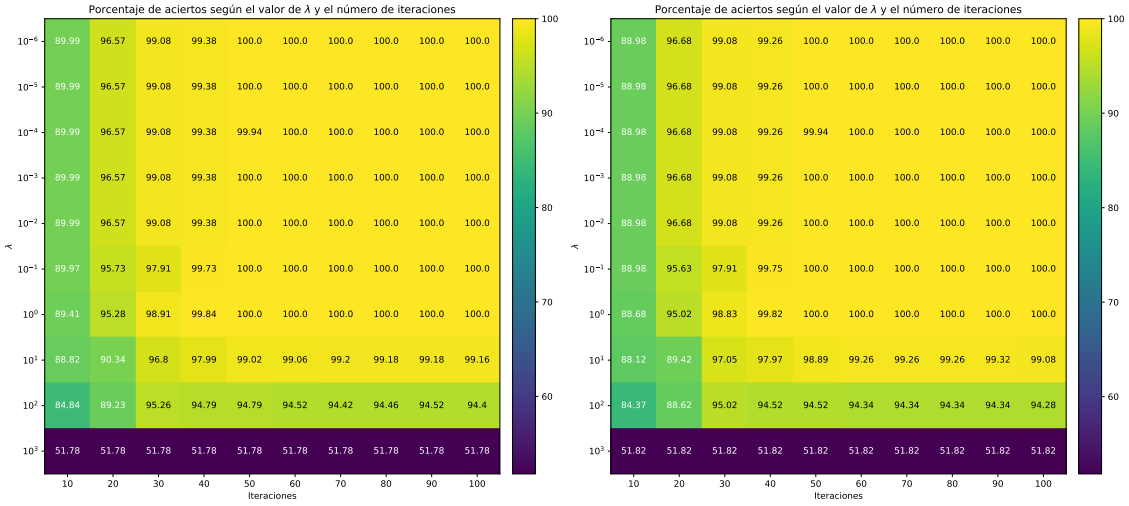

Figura 3.3: Resultados de la red neuronal con los datos de test (izquierda) y los de validación (derecha) en función del parámetro de regularización λ y el número de iteraciones

No se puede decir nada sobre la figura 3.3 que no se haya dicho en la sección anterior. A la vista de estos resultados, parece que aún quitando la variable *odor* siguen existiendo relaciones claras entre los atributos de los champiñones y su toxicidad.

En cuanto al error devuelto por la función de coste al clasificar los datos de validación (figura 3.4), son necesarias más iteraciones para minimizarlo, por lo que los parámetros que elegimos finalmente son $\lambda = 10^{-6}$ y 100 iteraciones, con los que se obtiene un 100 % de acierto con los datos de test.

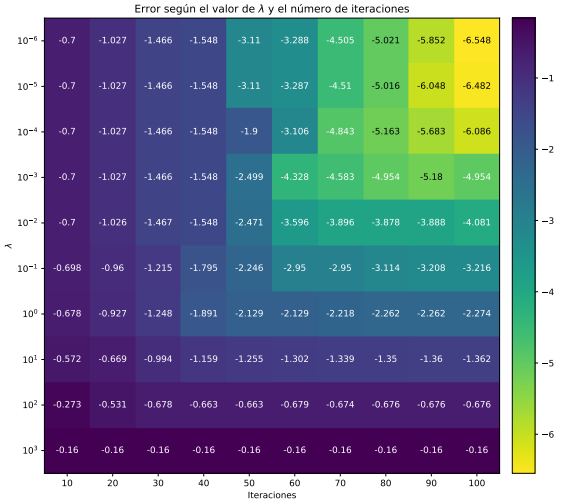

Figura 3.4: Error al clasificar los datos de validación sin la variable *odor*

Capítulo 4

Maquinas de vectores de soporte (SVM)

4.1. Procedimiento

En este caso utilizaremos funciones ya definidas en paquetes de *Python*, por lo que no crearemos nuestras propias funciones.

Para la SVM usaremos la clase `SVC` del paquete `sklearn.svm`. Resulta muy sencilla de utilizar, 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.

Hemos probado con dos funciones de *kernel* distintas. En primer lugar, utilizamos el *kernel lineal*, que solo depende del parámetro C. Tomaremos $C \in \{0{,}01,\ 0{,}03,\ 0{,}1,\ 0{,}3,\ 1,\ 3,\ 10,\ 30,\ 100,\ 300\}$, y veremos para qué valor se consigue un mayor porcentaje de acierto para los datos de validación. Después, comprobaremos el porcentaje de aciertos final obtenido con los datos de test. Los resultados se pueden comprobar en la sección correspondiente.

A continuación probamos con el *kernel gaussiano*. La clase `sklearn.svm` no tiene este *kernel* 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 $\gamma = \frac{1}{2\sigma^2}$. Igual que en el caso anterior, probaremos con distintos valores de C y σ, concretamente, $C, \sigma \in \{0{,}01,\ 0{,}03,\ 0{,}1,\ 0{,}3,\ 1,\ 3,\ 10,\ 30,\ 100,\ 300\}$. 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 los valores adecuados para C y σ, comprobamos el porcentaje de acierto sobre los datos de test, que aún no hemos utilizado y por tanto son más realistas. Nuevamente, los detalles sobre los resultados se encuentran en la siguiente sección.

4.2. Resultados

Primero probamos a clasificar el *dataset* empleando una SVM con *kernel* lineal. Los resultados obtenidos se muestran en la figura 4.1

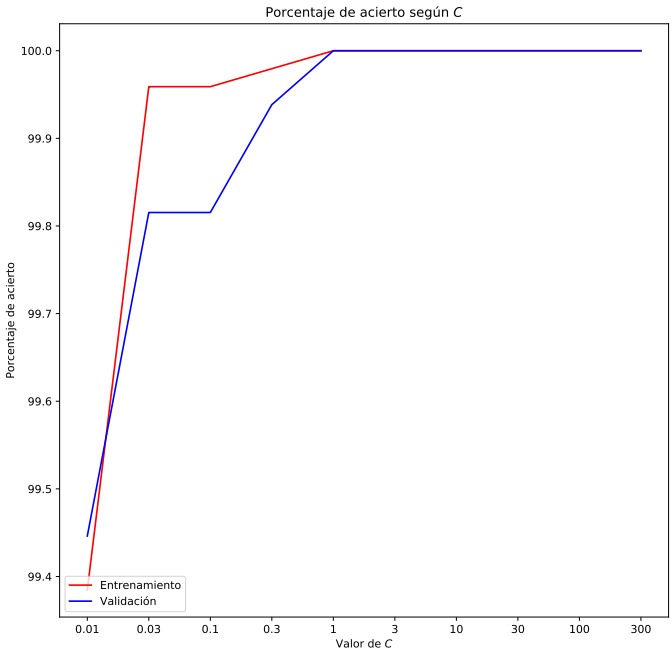

Figura 4.1: Resultados de la SVM con kernel lineal en función del parámetro C

Lo primero que se observa en la gráfica de la figura 4.1, es que la tasa de acierto con los datos de entrenamiento y validación es muy alta (por encima del 99 %) para todos los valores de C analizados. Para $C \geq 1$, se alcanza un 100 % de acierto tanto para los datos de entrenamiento como para los de validación.

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.

Por tanto, 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 sesgo y mayor varianza), 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 sesgo y menor varianza).

En la gráfica no se aprecia varianza aún incrementando mucho el valor de C, ya que la tasa de aciertos para los datos de validación se mantiene en el 100 %. Esto está en la línea de los resultados obtenidos con el resto de técnicas que hemos probado, y vuelve a mostrar que el dataset es muy fácil de clasificar. Para valores pequeños de C podemos detectar algo de sesgo, ya que la tasa de aciertos para datos entrenamiento y validación baja ligeramente, pero este sesgo es muy poco relevante, ya que a pesa de ello el éxito es superior al 99 %.

Finalmente, tomamos $C = 1$, que es el primer valor con el que se alcanza un 100 % de acierto con los datos de validación, y probamos a clasificar los datos de test, obteniendo de nuevo un 100 % de acierto.

A continuación, vamos a evaluar la capacidad de una SVM con *kernel gaussiano* para clasificar los champiñones del dataset. Ya que el *kernel* lineal ha funcionado muy bien, esperamos que este modelo también alcance tasas de éxito muy altas. Los resultados obtenidos se muestran en la figura 4.2. Probamos el modelo con distintos valores para los parámetros C, $\sigma \in \{0{,}01,\ 0{,}03,\ 0{,}1,\ 0{,}3,\ 1,\ 3,\ 10,\ 30,\ 100,\ 300\}$.

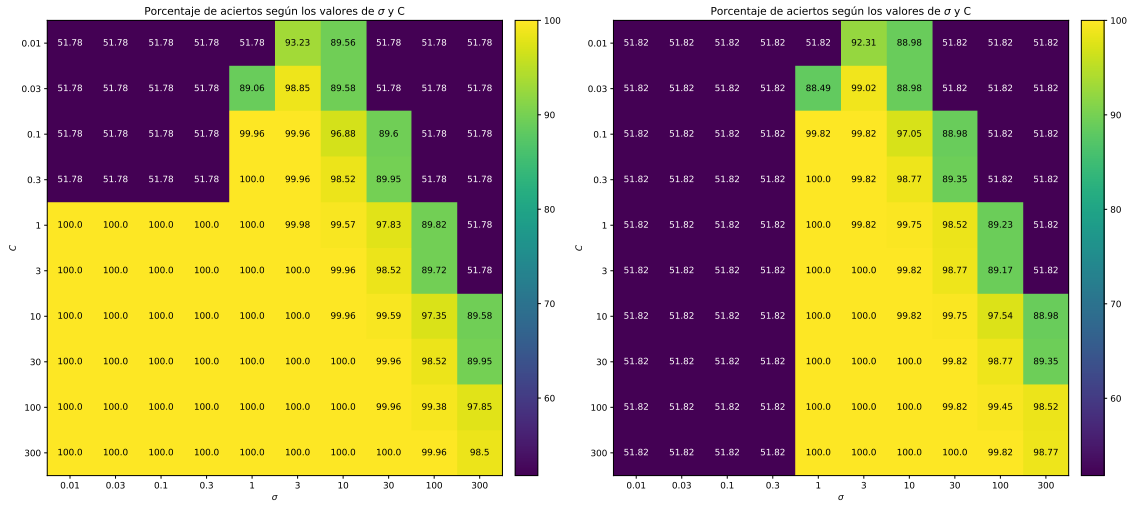

Figura 4.2: Resultados de la SVM con kernel *gaussiano* en función de σ y C al clasificar los datos de entrenamiento (izquierda) y de validación (derecha).

En el kernel *gaussiano* entra en escena el parámetro σ, además de la C. La influencia de 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.

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

Como consecuencia, con σ^2 grande se produce mayor sesgo y menor varianza, y con σ^2 pequeño se produce menor sesgo y mayor varianza.

En efecto, la figura 4.2 muestra que se produce varianza para valores de $\sigma < 1$, ya que se clasifican perfectamente los datos de entrenamiento, alcanzando un 100 % de acierto con $C \geq 1$, pero se falla mucho con los datos de validación, logrando tan solo un 51,82 % de aciertos. Es la primera vez que detectamos varianza o sobreaprendizaje de entre todas las técnicas que hemos probado. Por otro lado, cuando σ crece empieza a producirse sesgo, sobre todo a partir de $\sigma = 30$. Esto se ve reflejado en la figura 4.2, ya que tanto la tasa de acierto con los datos de validación como con los datos de entrenamiento comienzan a disminuir, especialmente si se combina con valores pequeños de C.

La figura muestra que para $\sigma = 1,\ 3,\ 10$ se consiguen las mayores tasas de acierto, siendo estos valores muy altos al combinarlos con la mayoría de valores de C.

En cuanto al valor de C, podemos extraer conclusiones parecidas a las del kernel lineal. Con C pequeños se produce mayor sesgo, haciendo que el modelo no sea capaz de clasificar correctamente ni los datos de entrenamiento ni los de validación. Esto se ve por ejemplo para $C = 0{,}01$, cuando σ es distinto de 3 o 10. Al incrementar C se reduce este sesgo, pero puede llegar a producirse varianza, es decir, que el modelo

se sobreajuste a los datos de entrenamiento y no clasifique bien otros daos, como los de validación. Este no es el caso, ya que aún para los valores de C más grandes que hemos probado la tasa de acierto es muy alta tanto para los datos de entrenamiento como los de validación.

Para la prueba final, elegimos los valores $\sigma = 1$ y $c = 0{,}3$, que son los primeros con los que el modelo ha alcanzado un 100 % de acierto con los datos de validación. Utilizamos el modelo con estos parámetros para clasificar los datos de test, alcanzando un 100 % de éxito también.

De nuevo, queda patente que el dataset es muy sencillo de clasificar y que las relaciones entre las características de los champiñones y su toxicidad está muy clara y es aprendida rápidamente por los modelos de aprendizaje automático.

4.3. Resultados sin la variable *odor*

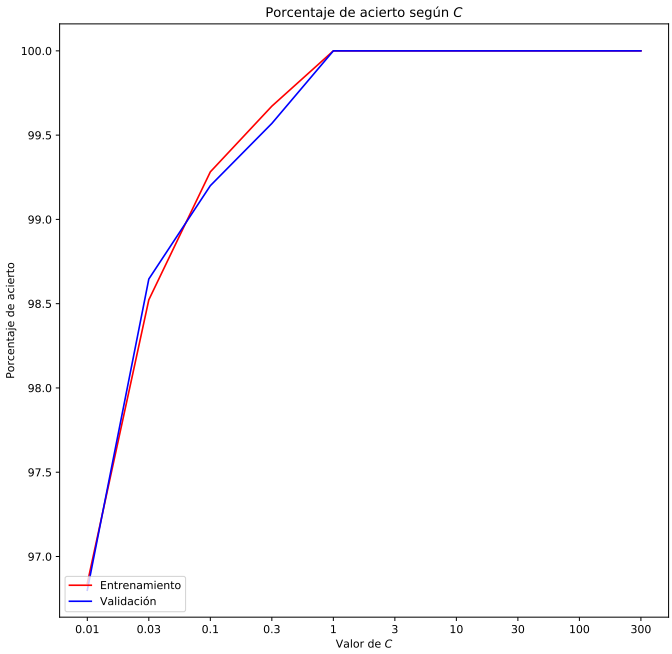

Figura 4.3: Resultados de la SVM con kernel lineal en función de σ y C al clasificar los datos de entrenamiento (izquierda) y de validación (derecha) sin la variable *odor*.

De nuevo observamos que no hay diferencias destacables tras entrenar la SVM con kernel lineal sin la variable *odor* respecto a los resultados obtenidos cuando sí se incluía esa variable. Esto confirma que el resto de variables aportan suficiente información para clasificar correctamente todos los ejemplos champiñones. Para el kernel lineal, el valor óptimo de C es 1, ofreciendo un 100 % de acierto sobre los datos de test.

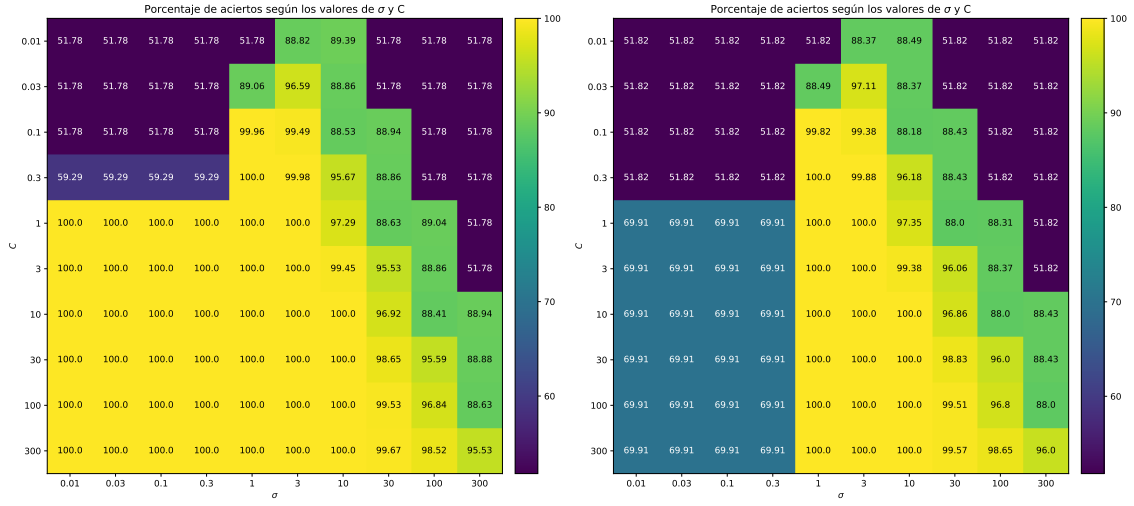

Figura 4.4: Resultados de la SVM con kernel *gaussiano* en función de σ y C al clasificar los datos de entrenamiento (izquierda) y de validación (derecha) sin la variable *odor*.

En cuanto al kernel *gaussiano*, se vuelven a repetir los resultados. La elección óptima de parámetros es $\sigma = 1$ y $c = 0{,}3$, exactamente igual que al incluir la variable *odor*, y el resultado al clasificar los datos de test es del 100 % de aciertos.

Capítulo 5

Conclusión

Tras la aplicación de diversas técnicas de aprendizaje automático para clasificar el conjunto de datos de champiñones utilizado, llegamos a la conclusión de que la relación entre las características de los champiñones y su toxicidad está muy clara, lo que hace que el dataset sea sencillo de clasificar mediante cualquiera de las técnicas de clasificación que hemos estudiado durante el curso. La variable más representativa para realizar la clasificación es el olor, pero aún quitándola las técnicas obtienen clasificar a la perfección todos los ejemplares.

Como todas las técnicas han alcanzado tasas de acierto del 100 %, es difícil determinar si unas son mejores que otras, salvo quizá por el tiempo que requieren para entrenarse. La regresión logística es la más rápida, seguida por la red neuronal y por las SVM, que son las más lentas, aunque ningún modelo tardó más de 10 minutos en terminar de entrenarse probando varios parámetros para evaluar cuál es el óptimo. En este sentido, la regresión logística es mejor por ser más rápida de entrenar y alcanzar un éxito del 100 %.

Al ser el dataset tan sencillo de clasificar, tuvimos la sensación de que los resultados no eran lo bastante interesantes, por lo que probamos a eliminar la variable *odor* que por sí sola permitía clasificar directamente la gran mayoría de ejemplos del dataset. Esperábamos que tras hacer esto los modelos tuvieran más dificultad para realizar la clasificación y por tanto que la tasa de aciertos bajara ligeramente, pero a pesar de haber retirado esta variable las técnicas probadas seguían siendo capaces de clasificar a la perfección todos los ejemplos. Además, seguía sin producirse sobreaprendizaje, mostrando que las características restantes de los champiñones siguen teniendo una relación muy clara con su toxicidad.

Por otro lado, la técnica de clustering *KMeans* no llegó a alcanzar un 100 % de acierto, aunque consiguió un porcentaje de acierto anda desdeñable, del 89,23 %. En principio, una mayor o menor distancia entre los puntos correspondientes a cada champiñón no tiene que depender de que sea venenoso o no, por lo que la técnica de

KMeans no debería obtener muy buenos resultados. Sin embargo, en este dataset, la diferencia entre los champiñones venenosos y los que no lo son está tan marcada que incluso el clustering obtiene una tasa de acierto bastante alta.

Capítulo 6

Código del proyecto

A continuación se ofrece el código escrito para realizar los experimentos explicados en esta memoria:

```
# -*- coding: utf-8 -*-
'''
Proyecto de la asignatura Aprendizaje Automático y Big Data

Rubén Ruperto Díaz y Rafael Herrera Troca
'''

import os
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from matplotlib import cm
from mpl_toolkits.axes_grid1 import make_axes_locatable
from sklearn.model_selection import train_test_split
from scipy.optimize import fmin_tnc, minimize
from sklearn.svm import SVC
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA

os.chdir("./resources")


#%% Funciones auxiliares


# Función sigmoide
def sigmoide(z):
```

```
    return 1 / (1 + np.exp(-z))

# Derivada de la función sigmoide
def diffSigmoide(a):
    return a * (1 - a)


## Funciones para regresión logística (práctica 2):

# Función de coste
def P1coste(theta, X, Y, reg=0):
    gXTheta = sigmoide(np.dot(X, theta))
    factor = np.dot(np.log(gXTheta).T, Y) + np.dot(np.log(1 - gXTheta).T,
                                                   1-Y)
    return -1 / len(Y) * factor + reg / (2 * len(Y)) * np.sum(theta**2)

# Gradiente de la función de coste
def P1gradiente(theta, X, Y, reg=0):
    gXTheta = sigmoide(np.dot(X, theta))
    thetaJ = np.concatenate(([0], theta[1:]))
    return 1 / len(Y) * np.dot(X.T, gXTheta-Y) + reg / len(Y) * thetaJ

# Función que devuelve el porcentaje de acierto de un resultado
# según el valor real
def P1porc_ac(X, Y, theta):
    gXTheta = sigmoide(np.dot(X, theta))
    resultados = [((gXTheta >= 0.5) & (Y == 1)) | ((gXTheta < 0.5)
                                                   & (Y == 0))]
    return np.count_nonzero(resultados) / len(Y) * 100


## Funciones para redes neuronales (práctica 4):

# Devuelve una matriz de pesos aleatorios con la dimensión dada
def P2randomWeights(l_in, l_out):
    eps = np.sqrt(6)/np.sqrt(l_in + l_out)
    rnd = np.random.random((l_out, l_in+1)) * (2*eps) - eps
    return rnd

# Dada la entrada 'X' y los pesos 'theta' de una capa de una red
# neuronal, aplica los pesos y devuelve la salida de la capa
def P2applyLayer(X, theta):
    thetaX = np.dot(X, theta.T)
    return sigmoide(thetaX)

# Dada la entrada 'X' y el array de matrices de pesos 'theta',
```

```python
# devuelve la entrada de cada capa y el resultado final devuelto
# por la red neuronal
def P2applyNet(X, theta):
    lay = X.copy()
    a = []
    for i in range(len(theta)):
        lay = np.hstack((np.array([np.ones(len(lay))]).T, lay))
        a.append(lay.copy())
        lay = P2applyLayer(lay, theta[i])

    return lay,a

# Calcula la función de coste de una red neuronal para la
# salida esperada 'y', el resultado de la red 'h_theta', el array
# de matrices de pesos 'theta' y el término de regularización 'reg'
def P2coste(y, h_theta, theta, reg):
    sumandos = -y * np.log(h_theta) - (1-y) * np.log(1-h_theta)
    regul = 0
    for i in range(len(theta)):
        regul += np.sum(theta[i][:,1:]**2)
    result = np.sum(sumandos) / len(y) + reg * regul / (2*len(y))
    return result

# Calcula el gradiente de la función de coste haciendo
# retropropagación dada la salida esperada 'y', la entrada
# de cada capa 'a', la salida de la red 'h_theta', el array de
# matrices de pesos 'theta' y el término de regularización 'reg'
def P2gradiente(y, a, h_theta, theta, reg):
    d = h_theta - y
    delta = [np.dot(d.T, a[-1]) / len(y)]

    for i in range(len(theta)-1,0,-1):
        d = np.dot(d, theta[i]) * diffSigmoide(a[i])
        d = d[:,1:]
        delta.insert(0, np.dot(d.T, a[i-1]) / len(y))

    for i in range(len(delta)):
        delta[i][:,1:] += reg * theta[i][:,1:] / len(y)

    return delta

# Calcula y devuelve el coste y el gradiente de una red neuronal
# dados todos los pesos en el array 'param_rn', las dimensiones
# de cada capa en 'capas', los datos de entrada 'X', la salida
# esperada 'y' y el término de regularización 'reg'
def P2backprop(params_rn, capas, X, Y, reg):
```

```
    # Convertimos el vector de todos los pesos en las distintas
    # matrices
    theta = [np.reshape(params_rn[:capas[1]*(capas[0]+1)],
                        (capas[1],capas[0]+1))]
    gastados = capas[1]*(capas[0]+1)
    for i in range(len(capas)-2):
        theta.append(np.reshape(params_rn[gastados:gastados+capas[i+2]*
                            (capas[i+1]+1)],(capas[i+2],capas[i+1]+1)))
        gastados += capas[i+2]*(capas[i+1]+1)

    # Aplicamos la red neuronal
    h_theta,a = P2applyNet(X, theta)

    cost = P2coste(Y, h_theta, theta, reg)
    grad = P2gradiente(Y, a, h_theta, theta, reg)

    g = np.array([])
    for i in range(len(grad)):
        g = np.concatenate((g, grad[i].ravel()))

    return cost, g

# Calcula el porcentaje de acierto dada la respuesta de la red y el
# resultado real
def P2porc_ac(res, Y):
    resultados = [((res >= 0.5) & (Y == 1)) | ((res < 0.5) & (Y == 0))]
    return np.count_nonzero(resultados) / len(Y) * 100


#%% Lectura y estudio de los datos


np.random.seed(27)

data = pd.read_csv('mushrooms.csv')

# Transformamos
Y = data['class'].replace({'p':0, 'e':1})
X = pd.get_dummies(data.drop('class', axis=1))

# Dividimos los datos en entrenamiento, validación y test
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.2,
                               random_state=0, shuffle=True, stratify=Y)
Xtrain, Xval, Ytrain, Yval = train_test_split(Xtrain, Ytrain,
       test_size=0.25, random_state=0, shuffle=True, stratify=Ytrain)
```

```
# Preparamos los datos
Xtrain2 = np.hstack((np.array([np.ones(len(Ytrain))]).T, Xtrain))
Xval2 = np.hstack((np.array([np.ones(len(Yval))]).T, Xval))
Xtest2 = np.hstack((np.array([np.ones(len(Ytest))]).T, Xtest))
Ytrain2 = np.array([Ytrain]).T
Yval2 = np.array([Yval]).T
Ytest2 = np.array([Ytest]).T

# Representamos un histograma para cada variable según la distribución de
# champiñones venenosos y comestibles para cada posible valor
for name in data.columns[1:]:
    plt.figure(figsize=(10,10))
    plt.title("Número de venenosos y comestibles según " + name)
    values = data[name].value_counts().axes[0].to_list()
    cuentaP = []
    cuentaE = []
    for v in values:
        cuentaP.append(len(data[(data[name]==v) & (data['class']=='p')]))
        cuentaE.append(len(data[(data[name]==v) & (data['class']=='e')]))
    plt.bar(np.arange(len(values)), cuentaP, 0.4, color='darkorchid')
    plt.bar(np.arange(len(values))+0.4, cuentaE, 0.4, color='greenyellow')
    plt.ylabel('Número de casos')
    plt.xlabel(name)
    plt.xticks(np.arange(len(values))+0.2, values)
    plt.savefig("var" + name + ".pdf", format='pdf')
    plt.show()


#%% Parte 1: Regresión logística


# Entrenamos la regresión con distintos valores para el término de
# regularización
theta0 = np.zeros(np.shape(Xtrain2)[1])
regValues = range(-10, 4)
thetas = []
errorTrain = []
acTrain = []
errorVal = []
acVal = []
for reg in regValues:
    theta = fmin_tnc(func=P1coste, x0=theta0, fprime=P1gradiente,
                                args=(Xtrain2, Ytrain, 10**reg))[0]
    thetas.append(theta)
    errorTrain.append(P1coste(theta, Xtrain2, Ytrain))
    acTrain.append(P1porc_ac(Xtrain2, Ytrain, theta))
```

```
    errorVal.append(P1coste(theta, Xval2, Yval))
    acVal.append(P1porc_ac(Xval2, Yval, theta))

# Comprobamos el error y el pocentaje de acierto según el término de
# regularización
opt = np.argmin(errorVal)
print('El valor óptimo del parámetro de regularización es',
                                                10**regValues[opt])

plt.figure(figsize=(10,10))
plt.plot(regValues, acTrain, 'r', label="Entrenamiento")
plt.plot(regValues, acVal, 'b', label="Validación")
plt.title(r"Porcentaje de acierto según $\lambda$")
plt.xlabel(r"Valor de $\lambda = 10^x$")
plt.ylabel("Porcentaje de acierto")
plt.legend(loc="lower left")
plt.savefig("aciertoLogistica.pdf", format='pdf')
plt.show()

plt.figure(figsize=(10,10))
plt.plot(regValues, errorTrain, 'r', label="Entrenamiento")
plt.plot(regValues, errorVal, 'b', label="Validación")
plt.title(r"Error según $\lambda$")
plt.xlabel(r"Valor de $\lambda = 10^x$")
plt.ylabel("Error")
plt.legend(loc="upper left")
plt.savefig("errorLogistica.pdf", format='pdf')
plt.show()

# Calculamos el porcentaje de acierto sobre los datos de test para el
# valor escogido del término de regularización
ac = P1porc_ac(Xtest2, Ytest, thetas[opt])
print('El porcentaje de acierto sobre los datos de test es', ac, '%')


#%% Parte 2: Redes neurales


# Creamos unas matrices inciales con pesos aleatorios
size2 = 25
theta01 = P2randomWeights(np.shape(Xtrain)[1], size2)
theta02 = P2randomWeights(size2, 1)
theta0 = np.concatenate((theta01.ravel(), theta02.ravel()))

regValues = range(-6, 4)
itera = range(10, 110, 10)
```

```
errorTrain = np.zeros((len(regValues), len(itera)))
acTrain = np.zeros((len(regValues), len(itera)))
errorVal = np.zeros((len(regValues), len(itera)))
acVal = np.zeros((len(regValues), len(itera)))

for i in range(len(regValues)):
    for j in range(len(itera)):
        theta = minimize(fun=P2backprop, x0=theta0,
                args=((np.shape(Xtrain)[1],size2,1), Xtrain, Ytrain2,
                10**regValues[i]), method='TNC', jac=True,
                options={'maxiter':itera[j]})['x']

        theta1 = np.reshape(theta[:size2*(np.shape(Xtrain)[1]+1)],
                            (size2,np.shape(Xtrain)[1]+1))
        theta2 = np.reshape(theta[size2*(np.shape(Xtrain)[1]+1):],
                            (1,size2+1))

        resTrain =  P2applyNet(Xtrain, (theta1, theta2))[0]
        acTrain[i][j] = P2porc_ac(resTrain, Ytrain2)
        resVal = P2applyNet(Xval, (theta1, theta2))[0]
        acVal[i][j] = P2porc_ac(resVal, Yval2)

        errorTrain[i][j] = P2coste(Ytrain2, resTrain, [theta1,theta2], 0)
        errorVal[i][j] = P2coste(Yval2, resVal, [theta1,theta2], 0)

# Comprobamos el error y el pocentaje de acierto según el término de
# regularización y el número de iteraciones
opt = np.argmin(errorVal)
optReg, optItera = 10**regValues[opt//len(itera)], itera[opt%len(itera)]
print('El valor óptimo del parámetro de regularización es', optReg)
print('El valor óptimo para el número de iteraciones es', optItera)

xLabels = [str(it) for it in itera]
yLabels = [r'$10^{' + str(r) + '}$' for r in regValues]

plt.figure(figsize=(10,10))
plt.title(r"Porcentaje de aciertos según el valor de $\lambda$ y" +
          " el número de iteraciones")
plt.ylabel(r'$\lambda$')
plt.xlabel('Iteraciones')
fig = plt.subplot()
im = fig.imshow(acVal, 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, round(acVal[j][i],2), ha="center",
               va="center", color=("k" if acVal[j][i] > 93 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("aciertoValNeuronal.pdf", format='pdf')
plt.show()

plt.figure(figsize=(10,10))
plt.title(r"Porcentaje de aciertos según el valor de $\lambda$ y " +
          "el número de iteraciones")
plt.ylabel(r'$\lambda$')
plt.xlabel('Iteraciones')
fig = plt.subplot()
im = fig.imshow(acTrain, 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, round(acTrain[j][i],2), ha="center",
               va="center", color=("k" if acTrain[j][i] > 93 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("aciertoTrainNeuronal.pdf", format='pdf')
plt.show()

plt.figure(figsize=(10,10))
plt.title(r"Error según el valor de $\lambda$ y el número de iteraciones")
plt.ylabel(r'$\lambda$')
plt.xlabel('Iteraciones')
fig = plt.subplot()
im = fig.imshow(np.log10(errorVal), cmap=cm.viridis_r)
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, round(np.log10(errorVal[j][i]),3),
               ha="center", va="center",
               color=("k" if np.log10(errorVal[j][i]) < -5 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("errorValNeuronal.pdf", format='pdf')
plt.show()

plt.figure(figsize=(10,10))
plt.title(r"Error según el valor de $\lambda$ y el número de iteraciones")
plt.ylabel(r'$\lambda$')
plt.xlabel('Iteraciones')
fig = plt.subplot()
im = fig.imshow(np.log10(errorTrain), cmap=cm.viridis_r)
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, round(np.log10(errorTrain[j][i]),3),
               ha="center", va="center",
               color=("k" if np.log10(errorTrain[j][i]) < -5 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("errorTrainNeuronal.pdf", format='pdf')
plt.show()

# Calculamos el porcentaje de acierto sobre los datos de test para el
# valor escogido del término de regularización y de iteraciones
theta = minimize(fun=P2backprop, x0=theta0,
        args=((np.shape(Xtrain)[1],size2,1), Xtrain, Ytrain2,
        optReg), method='TNC', jac=True, options={'maxiter':optItera})['x']

theta1 = np.reshape(theta[:size2*(np.shape(Xtrain)[1]+1)],
                                  (size2,np.shape(Xtrain)[1]+1))
theta2 = np.reshape(theta[size2*(np.shape(Xtrain)[1]+1):],(1,size2+1))

resTest =  P2applyNet(Xtest, (theta1, theta2))[0]
ac = P2porc_ac(resTest, Ytest2)
print('El porcentaje de acierto sobre los datos de test es', ac, '%')


#%% Parte 3: Máquinas de soporte vectorial


# Comenzamos usando kernel lineal y distintos valores de C
parValues = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100, 300]
acTrain = []
acVal = []
```

```
for C in parValues:
    svm = SVC(kernel='linear', C=C)
    svm.fit(Xtrain,Ytrain)

    acTrain.append(svm.score(Xtrain,Ytrain) * 100)
    acVal.append(svm.score(Xval,Yval) * 100)

# Comprobamos el porcentaje de acierto según el valor de C
opt = np.argmax(acVal)
print('El valor óptimo de C es', parValues[opt])

plt.figure(figsize=(10,10))
plt.plot(range(len(parValues)), acTrain, 'r', label="Entrenamiento")
plt.plot(range(len(parValues)), acVal, 'b', label="Validación")
plt.title(r"Porcentaje de acierto según $C$")
plt.xlabel(r"Valor de $C$")
plt.xticks(range(len(parValues)), parValues)
plt.ylabel("Porcentaje de acierto")
plt.legend(loc="lower left")
plt.savefig("aciertoSVM.pdf", format='pdf')
plt.show()

# Calculamos el porcentaje de acierto sobre los datos de test para el
# valor escogido de C
svm = SVC(kernel='linear', C=parValues[opt])
svm.fit(Xtrain,Ytrain)
ac = svm.score(Xtest,Ytest) * 100
print('El porcentaje de acierto sobre los datos de test es', ac, '%')

# Probamos ahora con kernel gaussiano utilizando distintos valores de C
# y de sigma
acTrain = np.zeros((len(regValues), len(regValues)))
acVal = np.zeros((len(regValues), len(regValues)))
for i in range(len(parValues)):
    for j in range(len(parValues)):
        svm = SVC(kernel='rbf', C=parValues[i],
                  gamma=1/(2*parValues[j]**2))
        svm.fit(Xtrain,Ytrain)

        acTrain[i][j] = svm.score(Xtrain,Ytrain) * 100
        acVal[i][j] = svm.score(Xval,Yval) * 100

# Comprobamos el pocentaje de acierto según los valores de C y sigma
opt = np.argmax(acVal)
optC, optSigma = parValues[opt//len(parValues)],
parValues[opt%len(parValues)]
```

```
print('El valor óptimo del parámetro C es', optC)
print('El valor óptimo para el parámetro sigma es', optSigma)

plt.figure(figsize=(10,10))
plt.title(r"Porcentaje de aciertos según los valores de $\sigma$ y C")
plt.ylabel('$C$')
plt.xlabel(r'$\sigma$')
fig = plt.subplot()
im = fig.imshow(acTrain, 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(parValues)):
    for j in range(len(parValues)):
        text = fig.text(i, j, round(acTrain[j][i],2), ha="center",
            va="center", color=("k" if acTrain[j][i] > 70 else "w"))
fig.set_xticks(np.arange(len(parValues)))
fig.set_yticks(np.arange(len(parValues)))
fig.set_xticklabels(parValues)
fig.set_yticklabels(parValues)
plt.savefig("aciertoTrainSVM.pdf", format='pdf')
plt.show()

plt.figure(figsize=(10,10))
plt.title(r"Porcentaje de aciertos según los valores de $\sigma$ y C")
plt.ylabel('$C$')
plt.xlabel(r'$\sigma$')
fig = plt.subplot()
im = fig.imshow(acVal, 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(parValues)):
    for j in range(len(parValues)):
        text = fig.text(i, j, round(acVal[j][i],2), ha="center",
            va="center", color=("k" if acVal[j][i] > 70 else "w"))
fig.set_xticks(np.arange(len(parValues)))
fig.set_yticks(np.arange(len(parValues)))
fig.set_xticklabels(parValues)
fig.set_yticklabels(parValues)
plt.savefig("aciertoValSVM.pdf", format='pdf')
plt.show()

# Calculamos el porcentaje de acierto sobre los datos de test para el
# valor escogido de C y sigma
svm = SVC(kernel='rbf', C=optC, gamma=1/(2*optSigma**2))
svm.fit(Xtrain,Ytrain)
ac = svm.score(Xtest,Ytest) * 100
```

```
print('El porcentaje de acierto sobre los datos de test es', ac, '%')

#%% Parte 4: K-Medias


# Entrenamos un K-Medias con 2 clusters
kmeans = KMeans(n_clusters=2, random_state=0).fit(Xtrain)
trainLabels = kmeans.labels_
valLabels = kmeans.predict(Xval)

# Comprobamos el pocentaje de acierto según la interpretación de las etiquetas
acTrainA = np.count_nonzero(trainLabels == Ytrain) / len(Ytrain) * 100
acTrainB = np.count_nonzero(trainLabels != Ytrain) / len(Ytrain) * 100
acValA = np.count_nonzero(valLabels == Yval) / len(Yval) * 100
acValB = np.count_nonzero(valLabels != Yval) / len(Yval) * 100

print('Interpretando las etiquetas de forma directa, el entrenamiento ' +
      'obtiene un porcentaje de acierto del ', acTrainA, '%')
print('Interpretando las etiquetas de forma inversa, el entrenamiento ' +
      'obtiene un porcentaje de acierto del ', acTrainB, '%')
print('Interpretando las etiquetas de forma directa, la validación ' +
      'obtiene un porcentaje de acierto del ', acValA, '%')
print('Interpretando las etiquetas de forma inversa, la validación ' +
      'obtiene un porcentaje de acierto del ', acValB, '%')

# Calculamos el porcentaje de acierto sobre los datos de test para la
# interpretación escogida de las etiquetas
testLabels = kmeans.predict(Xtest)
ac = (np.count_nonzero(testLabels == Ytest) if acValA > acValB
      else np.count_nonzero(testLabels != Ytest)) / len(Ytest) * 100

print('El porcentaje de acierto sobre los datos de test es', ac, '%')

#%% Parte 5: Reducción de dimensionalidad


# Aplicamos PCA y comprobamos la varianza explicada para cada componente
pca = PCA()
XtrainR = pca.fit_transform(Xtrain)
expVar = pca.explained_variance_ratio_
expVarAcum = [expVar[0]]
for i in range(1, len(expVar)):
    expVarAcum.append(expVar[i] + expVarAcum[-1])

print("La varianza explicada acumulada es:", np.array(expVarAcum) * 100)
```

```
#%% Alternativa: Eliminar la variable 'olor' correlacionada


print("A partir de este punto repetimos los experimentos eliminando la " +
      "variable 'odor', ya que está fuertemente correlacionada con la " +
      "variable objetivo")

# Preparamos un conjunto adicional de datos eliminando la variable 'odor'
odor_lab = []
for c in Xtrain.columns:
    if c[:4] == 'odor':
        odor_lab.append(c)
Wtrain = Xtrain.copy().drop(odor_lab, 1)
Wval = Xval.copy().drop(odor_lab, 1)
Wtest = Xtest.copy().drop(odor_lab, 1)
Wtrain2 = np.hstack((np.array([np.ones(len(Ytrain))]).T, Wtrain))
Wval2 = np.hstack((np.array([np.ones(len(Yval))]).T, Wval))
Wtest2 = np.hstack((np.array([np.ones(len(Ytest))]).T, Wtest))


#%% Parte 1b: Regresión logística


# Entrenamos la regresión con distintos valores para el término de
# regularización
theta0 = np.zeros(np.shape(Wtrain2)[1])
regValues = range(-10, 4)
thetas = []
errorTrain = []
acTrain = []
errorVal = []
acVal = []
for reg in regValues:
    theta = fmin_tnc(func=P1coste, x0=theta0, fprime=P1gradiente,
                     args=(Wtrain2, Ytrain, 10**reg))[0]
    thetas.append(theta)
    errorTrain.append(P1coste(theta, Wtrain2, Ytrain))
    acTrain.append(P1porc_ac(Wtrain2, Ytrain, theta))
    errorVal.append(P1coste(theta, Wval2, Yval))
    acVal.append(P1porc_ac(Wval2, Yval, theta))

# Comprobamos el error y el pocentaje de acierto según el término de
# regularización
opt = np.argmin(errorVal)
print('El valor óptimo del parámetro de regularización es',
```

```
      10**regValues[opt])

plt.figure(figsize=(10,10))
plt.plot(regValues, acTrain, 'r', label="Entrenamiento")
plt.plot(regValues, acVal, 'b', label="Validación")
plt.title(r"Porcentaje de acierto según $\lambda$")
plt.xlabel(r"Valor de $\lambda = 10^x$")
plt.ylabel("Porcentaje de acierto")
plt.legend(loc="lower left")
plt.savefig("aciertoLogisticaNO.pdf", format='pdf')
plt.show()

plt.figure(figsize=(10,10))
plt.plot(regValues, errorTrain, 'r', label="Entrenamiento")
plt.plot(regValues, errorVal, 'b', label="Validación")
plt.title(r"Error según $\lambda$")
plt.xlabel(r"Valor de $\lambda = 10^x$")
plt.ylabel("Error")
plt.legend(loc="upper left")
plt.savefig("errorLogisticaNO.pdf", format='pdf')
plt.show()

# Calculamos el porcentaje de acierto sobre los datos de test para el
# valor escogido del término de regularización
ac = P1porc_ac(Wtest2, Ytest, thetas[opt])
print('El porcentaje de acierto sobre los datos de test es', ac, '%')


#%% Parte 2b: Redes neuronales


# Creamos unas matrices inciales con pesos aleatorios
size2 = 25
theta01 = P2randomWeights(np.shape(Wtrain)[1], size2)
theta02 = P2randomWeights(size2, 1)
theta0 = np.concatenate((theta01.ravel(), theta02.ravel()))

regValues = range(-6, 4)
itera = range(10, 110, 10)

errorTrain = np.zeros((len(regValues), len(itera)))
acTrain = np.zeros((len(regValues), len(itera)))
errorVal = np.zeros((len(regValues), len(itera)))
acVal = np.zeros((len(regValues), len(itera)))

for i in range(len(regValues)):
```

```
    for j in range(len(itera)):
        theta = minimize(fun=P2backprop, x0=theta0,
                args=((np.shape(Wtrain)[1],size2,1), Wtrain, Ytrain2,
                10**regValues[i]), method='TNC', jac=True,
                options={'maxiter':itera[j]})['x']

        theta1 = np.reshape(theta[:size2*(np.shape(Wtrain)[1]+1)],
                            (size2,np.shape(Wtrain)[1]+1))
        theta2 = np.reshape(theta[size2*(np.shape(Wtrain)[1]+1):],
                            (1,size2+1))

        resTrain =  P2applyNet(Wtrain, (theta1, theta2))[0]
        acTrain[i][j] = P2porc_ac(resTrain, Ytrain2)
        resVal = P2applyNet(Wval, (theta1, theta2))[0]
        acVal[i][j] = P2porc_ac(resVal, Yval2)

        errorTrain[i][j] = P2coste(Ytrain2, resTrain, [theta1,theta2], 0)
        errorVal[i][j] = P2coste(Yval2, resVal, [theta1,theta2], 0)

# Comprobamos el error y el pocentaje de acierto según el término de
# regularización y el número de iteraciones
opt = np.argmin(errorVal)
optReg, optItera = 10**regValues[opt//len(itera)], itera[opt%len(itera)]
print('El valor óptimo del parámetro de regularización es', optReg)
print('El valor óptimo para el número de iteraciones es', optItera)

xLabels = [str(it) for it in itera]
yLabels = [r'$10^{' + str(r) + '}$' for r in regValues]

plt.figure(figsize=(10,10))
plt.title(r"Porcentaje de aciertos según el valor de $\lambda$ y el " +
          "número de iteraciones")
plt.ylabel(r'$\lambda$')
plt.xlabel('Iteraciones')
fig = plt.subplot()
im = fig.imshow(acVal, 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, round(acVal[j][i],2), ha="center",
               va="center", color=("k" if acVal[j][i] > 93 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("aciertoValNeuronalNO.pdf", format='pdf')
plt.show()

plt.figure(figsize=(10,10))
plt.title(r"Porcentaje de aciertos según el valor de $\lambda$ y el " +
          "número de iteraciones")
plt.ylabel(r'$\lambda$')
plt.xlabel('Iteraciones')
fig = plt.subplot()
im = fig.imshow(acTrain, 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, round(acTrain[j][i],2), ha="center",
            va="center", color=("k" if acTrain[j][i] > 93 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("aciertoTrainNeuronalNO.pdf", format='pdf')
plt.show()

plt.figure(figsize=(10,10))
plt.title(r"Error según el valor de $\lambda$ y el número de iteraciones")
plt.ylabel(r'$\lambda$')
plt.xlabel('Iteraciones')
fig = plt.subplot()
im = fig.imshow(np.log10(errorVal), cmap=cm.viridis_r)
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, round(np.log10(errorVal[j][i]),3),
            ha="center", va="center",
            color=("k" if np.log10(errorVal[j][i]) < -5 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("errorValNeuronalNO.pdf", format='pdf')
plt.show()

plt.figure(figsize=(10,10))
plt.title(r"Error según el valor de $\lambda$ y el número de iteraciones")
plt.ylabel(r'$\lambda$')
```

```
plt.xlabel('Iteraciones')
fig = plt.subplot()
im = fig.imshow(np.log10(errorTrain), cmap=cm.viridis_r)
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, round(np.log10(errorTrain[j][i]),3),
            ha="center", va="center",
            color=("k" if np.log10(errorTrain[j][i]) < -5 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("errorTrainNeuronalNO.pdf", format='pdf')
plt.show()

# Calculamos el porcentaje de acierto sobre los datos de test para el
# valor escogido del término de regularización y de iteraciones
theta = minimize(fun=P2backprop, x0=theta0,
        args=((np.shape(Wtrain)[1],size2,1), Wtrain, Ytrain2, optReg),
        method='TNC', jac=True, options={'maxiter':optItera})['x']

theta1 = np.reshape(theta[:size2*(np.shape(Wtrain)[1]+1)],
                    (size2,np.shape(Wtrain)[1]+1))
theta2 = np.reshape(theta[size2*(np.shape(Wtrain)[1]+1):],(1,size2+1))

resTest =  P2applyNet(Wtest, (theta1, theta2))[0]
ac = P2porc_ac(resTest, Ytest2)
print('El porcentaje de acierto sobre los datos de test es', ac, '%')


#%% Parte 3b: Máquinas de soporte vectorial


# Comenzamos usando kernel lineal y distintos valores de C
parValues = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30, 100, 300]
acTrain = []
acVal = []
for C in parValues:
    svm = SVC(kernel='linear', C=C)
    svm.fit(Wtrain,Ytrain)

    acTrain.append(svm.score(Wtrain,Ytrain) * 100)
    acVal.append(svm.score(Wval,Yval) * 100)
```

```
# Comprobamos el porcentaje de acierto según el valor de C
opt = np.argmax(acVal)
print('El valor óptimo de C es', parValues[opt])

plt.figure(figsize=(10,10))
plt.plot(range(len(parValues)), acTrain, 'r', label="Entrenamiento")
plt.plot(range(len(parValues)), acVal, 'b', label="Validación")
plt.title(r"Porcentaje de acierto según $C$")
plt.xlabel(r"Valor de $C$")
plt.xticks(range(len(parValues)), parValues)
plt.ylabel("Porcentaje de acierto")
plt.legend(loc="lower left")
plt.savefig("aciertoSVMNO.pdf", format='pdf')
plt.show()

# Calculamos el porcentaje de acierto sobre los datos de test para el
# valor escogido de C
svm = SVC(kernel='linear', C=parValues[opt])
svm.fit(Wtrain,Ytrain)
ac = svm.score(Wtest,Ytest) * 100
print('El porcentaje de acierto sobre los datos de test es', ac, '%')

# Probamos ahora con kernel gaussiano utilizando distintos valores de C
# y de sigma
acTrain = np.zeros((len(regValues), len(regValues)))
acVal = np.zeros((len(regValues), len(regValues)))
for i in range(len(parValues)):
    for j in range(len(parValues)):
        svm = SVC(kernel='rbf', C=parValues[i],
                  gamma=1/(2*parValues[j]**2))
        svm.fit(Wtrain,Ytrain)

        acTrain[i][j] = svm.score(Wtrain,Ytrain) * 100
        acVal[i][j] = svm.score(Wval,Yval) * 100

# Comprobamos el pocentaje de acierto según los valores de C y sigma
opt = np.argmax(acVal)
optC, optSigma = parValues[opt//len(parValues)],
parValues[opt%len(parValues)]
print('El valor óptimo del parámetro C es', optC)
print('El valor óptimo para el parámetro sigma es', optSigma)

plt.figure(figsize=(10,10))
plt.title(r"Porcentaje de aciertos según los valores de $\sigma$ y C")
plt.ylabel('$C$')
plt.xlabel(r'$\sigma$')
```

```
fig = plt.subplot()
im = fig.imshow(acTrain, 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(parValues)):
    for j in range(len(parValues)):
        text = fig.text(i, j, round(acTrain[j][i],2), ha="center",
            va="center", color=("k" if acTrain[j][i] > 70 else "w"))
fig.set_xticks(np.arange(len(parValues)))
fig.set_yticks(np.arange(len(parValues)))
fig.set_xticklabels(parValues)
fig.set_yticklabels(parValues)
plt.savefig("aciertoTrainSVMNO.pdf", format='pdf')
plt.show()

plt.figure(figsize=(10,10))
plt.title(r"Porcentaje de aciertos según los valores de $\sigma$ y C")
plt.ylabel('$C$')
plt.xlabel(r'$\sigma$')
fig = plt.subplot()
im = fig.imshow(acVal, 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(parValues)):
    for j in range(len(parValues)):
        text = fig.text(i, j, round(acVal[j][i],2), ha="center",
            va="center", color=("k" if acVal[j][i] > 70 else "w"))
fig.set_xticks(np.arange(len(parValues)))
fig.set_yticks(np.arange(len(parValues)))
fig.set_xticklabels(parValues)
fig.set_yticklabels(parValues)
plt.savefig("aciertoValSVMNO.pdf", format='pdf')
plt.show()

# Calculamos el porcentaje de acierto sobre los datos de test para el
# valor escogido de C y sigma
svm = SVC(kernel='rbf', C=optC, gamma=1/(2*optSigma**2))
svm.fit(Wtrain,Ytrain)
ac = svm.score(Wtest,Ytest) * 100
print('El porcentaje de acierto sobre los datos de test es', ac, '%')


#%% Parte 4b: K-Medias


# Entrenamos un K-Medias con 2 clusters
```

```
kmeans = KMeans(n_clusters=2, random_state=0).fit(Wtrain)
trainLabels = kmeans.labels_
valLabels = kmeans.predict(Wval)

# Comprobamos el pocentaje de acierto según la interpretación de las etiquetas
acTrainA = np.count_nonzero(trainLabels == Ytrain) / len(Ytrain) * 100
acTrainB = np.count_nonzero(trainLabels != Ytrain) / len(Ytrain) * 100
acValA = np.count_nonzero(valLabels == Yval) / len(Yval) * 100
acValB = np.count_nonzero(valLabels != Yval) / len(Yval) * 100

print('Interpretando las etiquetas de forma directa, el entrenamiento' +
      ' obtiene un porcentaje de acierto del ', acTrainA, '%')
print('Interpretando las etiquetas de forma inversa, el entrenamiento' +
      ' obtiene un porcentaje de acierto del ', acTrainB, '%')
print('Interpretando las etiquetas de forma directa, la validación ' +
      'obtiene un porcentaje de acierto del ', acValA, '%')
print('Interpretando las etiquetas de forma inversa, la validación ' +
      'obtiene un porcentaje de acierto del ', acValB, '%')

# Calculamos el porcentaje de acierto sobre los datos de test para la
# interpretación escogida de las etiquetas
testLabels = kmeans.predict(Wtest)
ac = (np.count_nonzero(testLabels == Ytest) if acValA > acValB
      else np.count_nonzero(testLabels != Ytest)) / len(Ytest) * 100

print('El porcentaje de acierto sobre los datos de test es', ac, '%')


#%% Parte 5b: Reducción de dimensionalidad


# Aplicamos PCA y comprobamos la varianza explicada para cada componente
pca = PCA()
WtrainR = pca.fit_transform(Wtrain)
expVar = pca.explained_variance_ratio_
expVarAcum = [expVar[0]]
for i in range(1, len(expVar)):
    expVarAcum.append(expVar[i] + expVarAcum[-1])

print("La varianza explicada acumulada es:", np.array(expVarAcum) * 100)
```