Aprendizaje Automático

Abulones

Miguel Ángel Castillo Moreno

Rodrigo Lagartera Peña

Índice

Abulones

Los abulones, también conocidos como orejas de mar o haliótidos son una familia de moluscos gasterópodos. Su carne es consumida especialmente en Asia Oriental, aunque recientemente también en Estados Unidos y México, por lo que hay problemas con su conservación.

La edad de las orejas de mar se determina cortando la cáscara a través del cono, tiñéndola, y contando el número de anillos desde un microscopio. Sumándole 1.5 al número de anillos podemos averiguar su edad. Sin embargo, esto es proceso aburrido y que consume mucho tiempo. Por ello, este proyecto tiene como objetivo encontrar una manera de averiguar su edad (o número de anillos) utilizando otros factores más fáciles de medir.

Entre los factores que podemos analizar están el sexo, su longitud, el diámetro, la altura y diversas formas de pesarlo.

Los ejemplos de entrenamiento

El dataset del que disponemos se compone de 4177 ejemplos, con 8 atributos cada uno. Estos son:

- **El sexo**: este puede ser macho, hembra o infante.

- **La longitud**: Obtenida en milímetros, mida la parte más larga del abulón.

- **El diámetro**: Se mide en milímetros, perpendicularmente a la longitud.

- **La altura**: En milímetros, incluyendo la carne en el caparazón.

- **El peso completo**: En gramos, se trata del peso del abulón completo.

- **El peso desvainado**: En gramos, teniendo en cuenta solo la carne.

- **El peso de las vísceras**: Es el peso de las vísceras tras el sangrado, en gramos.

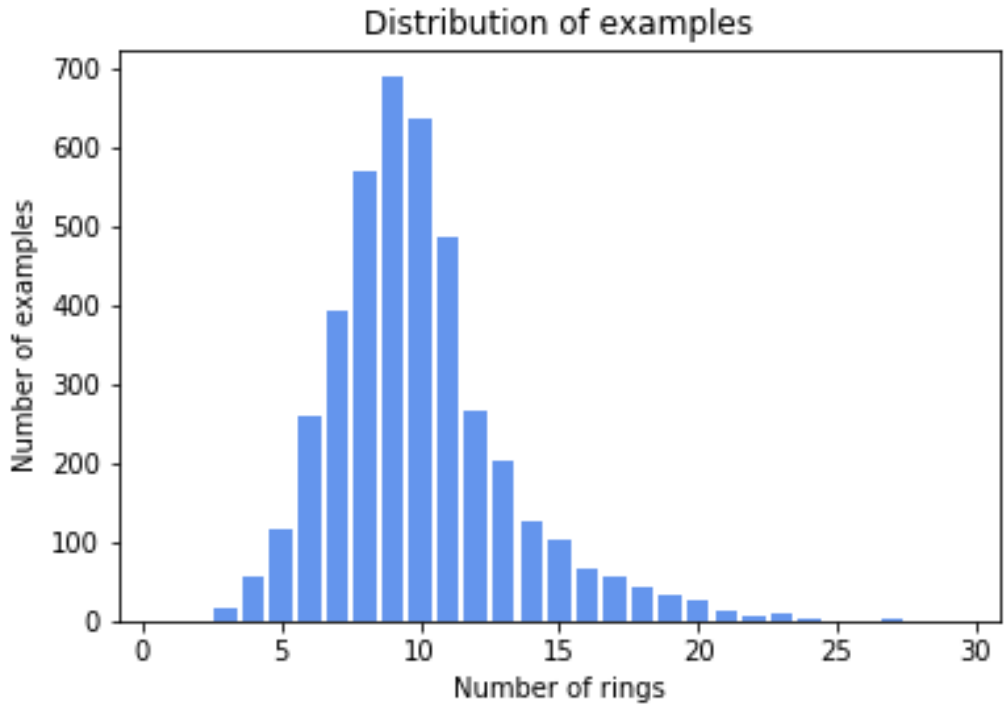
- **El peso de la concha**: En gramos, después de ser secada.

Como valor de salida se nos proporciona el número de anillos que tiene cada abulón. Trabajaremos con todos estos atributos.

Análisis de los ejemplos

Antes de comenzar a aplicar técnicas de aprendizaje automático, es conveniente que analicemos cómo son los ejemplos proporcionados.

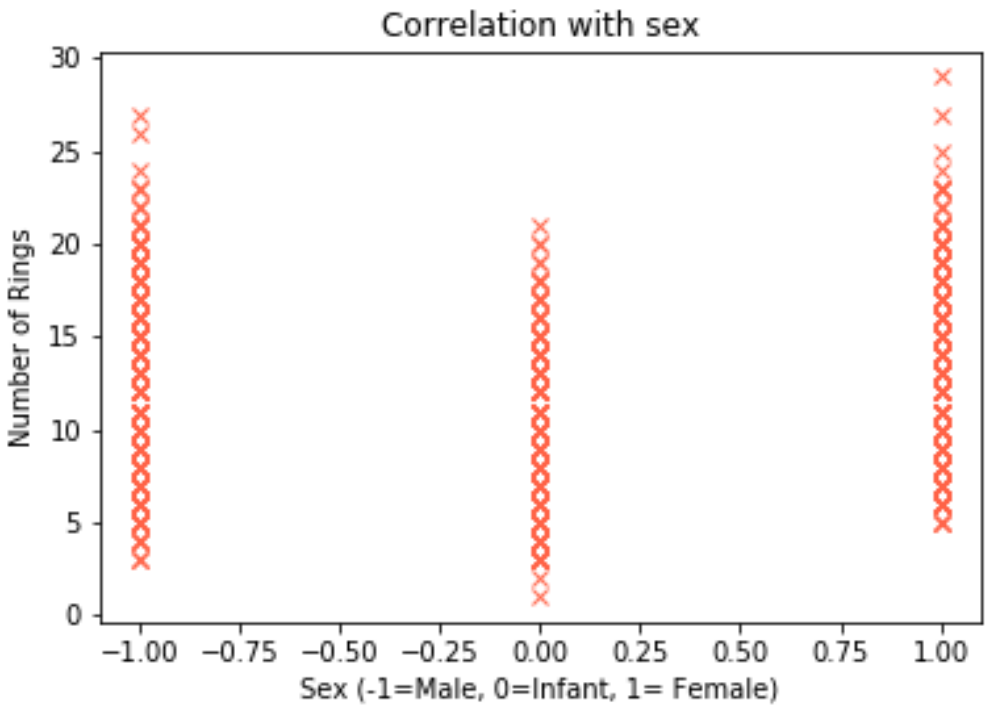
Para empezar, podemos observar cuantos valores distintos para el número de anillos hay, y cuantos ejemplos hay para cada valor. La información se proporciona en la siguiente gráfica.



Como se puede observar, el rango de anillos oscila entre 1 y 29. También cabe destacar (aunque no se aprecie en la gráfica), que hay ejemplos para cada número de anillos del 1 al 29 salvo para 28 anillos, y que solo hay un ejemplo de 1 abulón con 29.

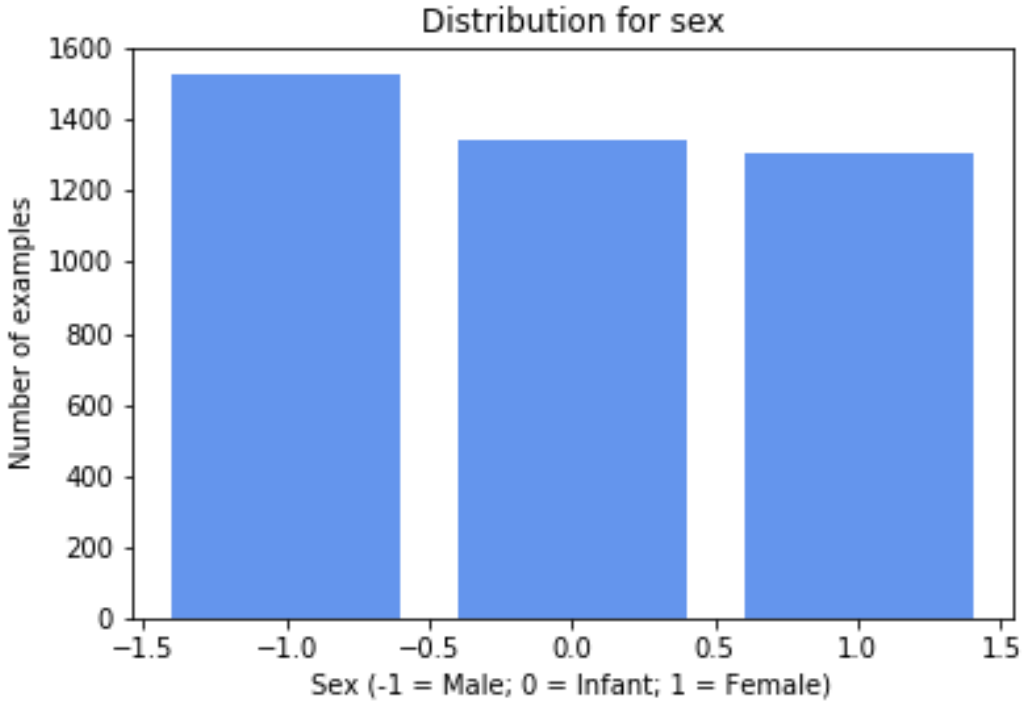
En esta gráfica también podemos ver que la media de número de anillos es 9 con una desviación típica aproximada de 4. Tener tantas clases distintas va a hacer difícil clasificar los ejemplos, especialmente los que se alejen mucho de la media de 9. Por lo tanto, pondremos medidas para que las predicciones puedan ser más precisas.

Ahora pasamos a ver la relación de cada atributo con la edad, empezando con el sexo:

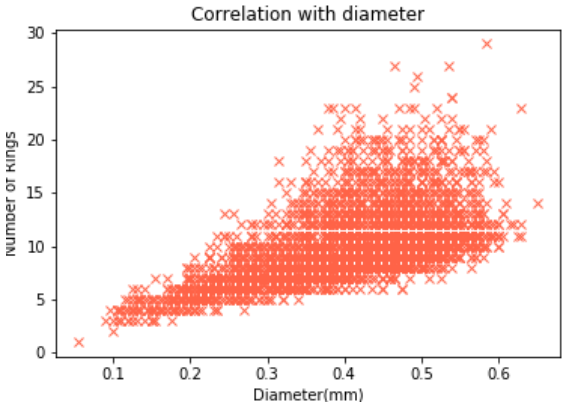


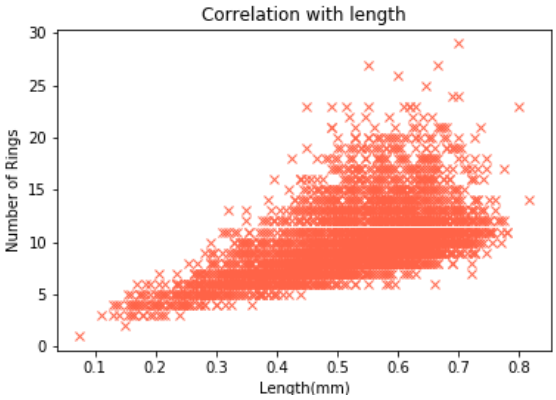
La elección de valores numéricos para el sexo se comentará más tarde. Podemos ver que en los infantes la edad, tanto mínima como máxima es menor respecto a los adultos, como cabría esperar. Comparando machos y hembras vemos que las hembras suelen tener mayor edad.

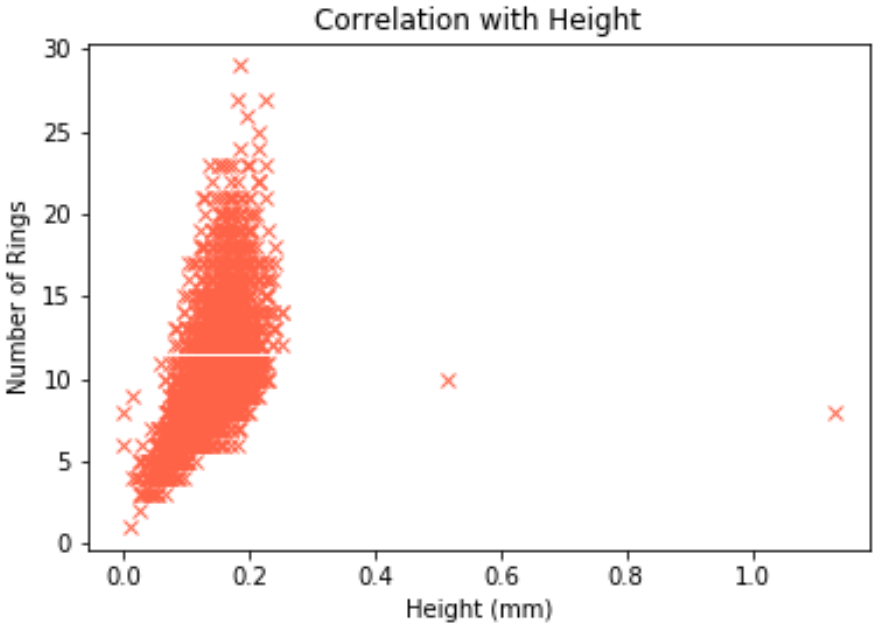
Se puede comprobar también en este otro gráfico que el número de ejemplos es similar para cada sexo, aunque hay ligeramente más ejemplos de abulones macho.



Pasamos a ver el diámetro, la longitud y la altura respectivamente:

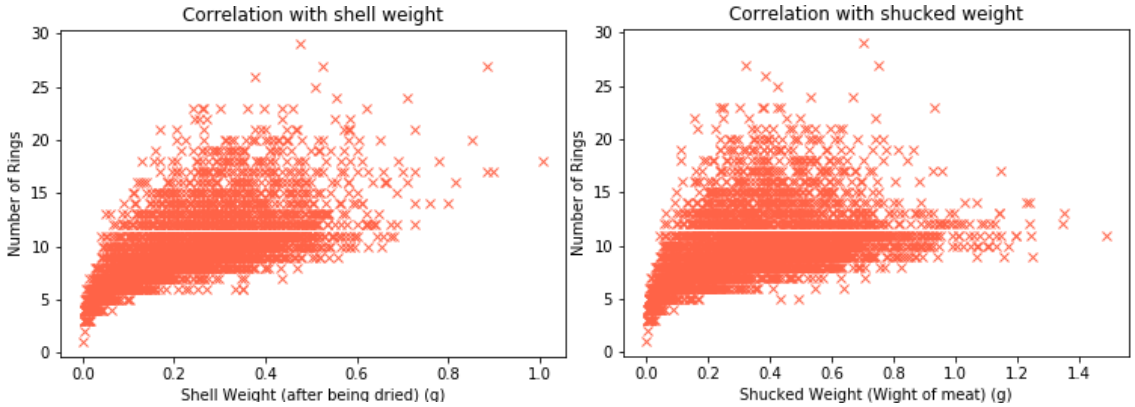


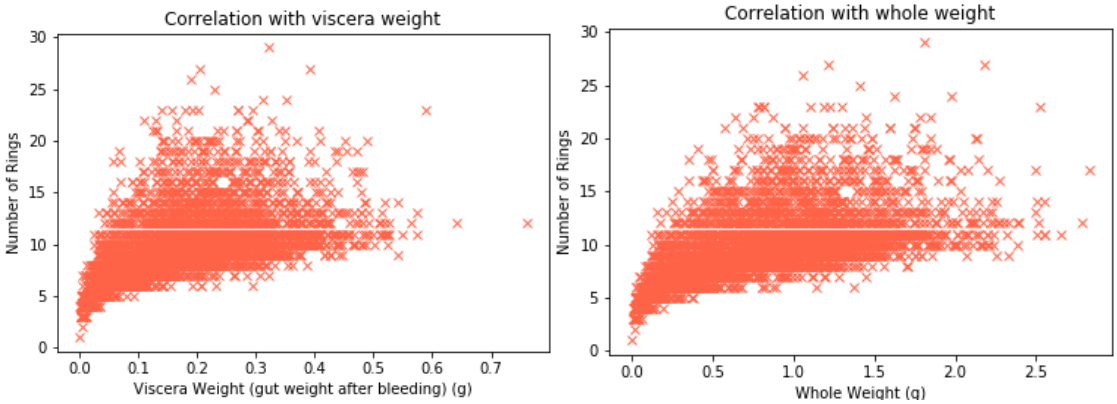




Los resultados son los esperables, a mayor valor, mayor tiende a ser su edad. Podemos resaltar el gráfico de la altura, en el que hay dos ejemplos de aproximadamente 7-9 anillos con una altura mucho mayor a la media, y que los gráficos del diámetro y la longitud son muy similares.

Ahora observaremos los pesos:





De nuevo obtenemos resultados comprensibles, a mayor peso mayor tiende a ser su edad. De los 4 pesos se obtienen gráficas muy similares, siendo esto posiblemente porque todos están correlacionados.

Aprendizaje automático

A lo largo de toda esta sección intentaremos resolver el problema mediante tres métodos distintos de aprendizaje automático: Support Vector Machines, redes neuronales y regresión logística.

Antes de introducirnos a cualquiera de estos métodos tenemos que ver de qué manera podemos cargar los datos. Aquí es donde nos encontramos con el primer problema: el sexo no es un valor numérico.

7 de los atributos vienen dados como valores reales, todos salvo el sexo. Esto es inconveniente, ya que nos es imposible utilizar la función para cargar los datos, y lo que es más importante, los métodos de aprendizaje automático no funcionarán con valores como “M”.

La solución es sencilla: asignamos a “I”, “F” y “M” un valor numérico cualquiera. Para esto creamos un nuevo archivo llamado “abalonesPreprocessed.txt” en el que copiemos el contenido del .txt original sustituyendo “F”, “I” y “M” por -1, 0 y 1 respectivamente. La elección de estos valores es completamente arbitraria y funcionaría igual de ser cualquier otro set de valores distintos entre sí.

Es el momento en el que tenemos que aclarar algo importante: Hay muchas formas diferentes de cargar los valores, junto a muchos parámetros que podemos manejar. Iremos a lo largo de los 3 métodos probando qué es más eficaz y qué no, cuando tengamos los recursos.

Otro factor importante es dividir los ejemplos que utilizaremos para entrenar y los que utilicemos para buscar parámetros o comprobar. De momento utilizaremos 3000 ejemplos de entrenamiento, 800 de validación y el resto para comprobaciones. Veremos por qué en la regresión lineal.

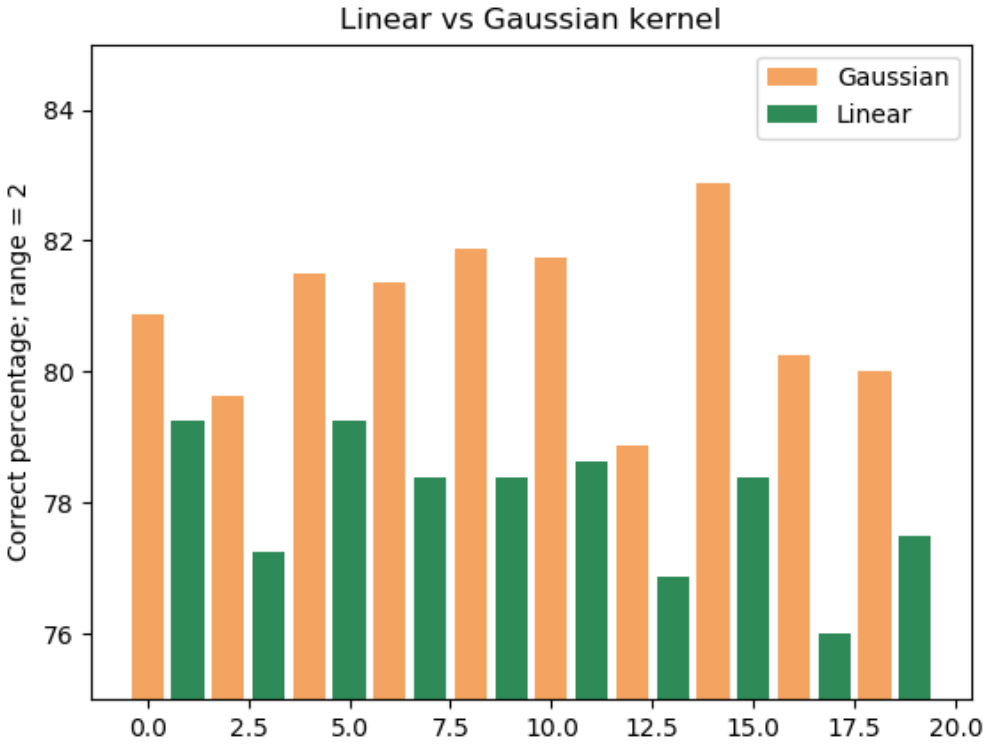
Support Vector Machine

Las máquinas de vectores de soporte son algoritmos de aprendizaje supervisados, al igual que el resto de métodos que veremos, y que utilizaremos para clasificar los ejemplos en sus respectivas clases.

Lo primero que tenemos que ver es qué kernel utilizaremos para clasificar. Hay varias posibilidades, y de todas ellas nos vamos a fijar en dos, el kernel lineal y el kernel gaussiano. Para seleccionarlo hay que tener en cuenta dos factores: el número de ejemplos de entrenamiento y el número de ejemplos disponibles.

El kernel lineal nos proporcionará mejores resultados cuando el número de atributos es mucho mayor que el número de ejemplos, mi entras que el gaussiano nos será útil si hay menos atributos que ejemplos. Si hay una cantidad intermedia de ejemplos de entrenamiento respecto a los atributos, utilizaremos el kernel gaussiano directamente, mientras que si hay una cantidad muy superior de ejemplos de entrenamiento, es conveniente que añadamos o creemos más atributos antes de utilizar la SVM.

Pasamos a ver la diferencia en los resultados una vez implementada la SVM de ambas formas, e iremos explicando con ellos la implementación y que significan los resultados obtenidos.



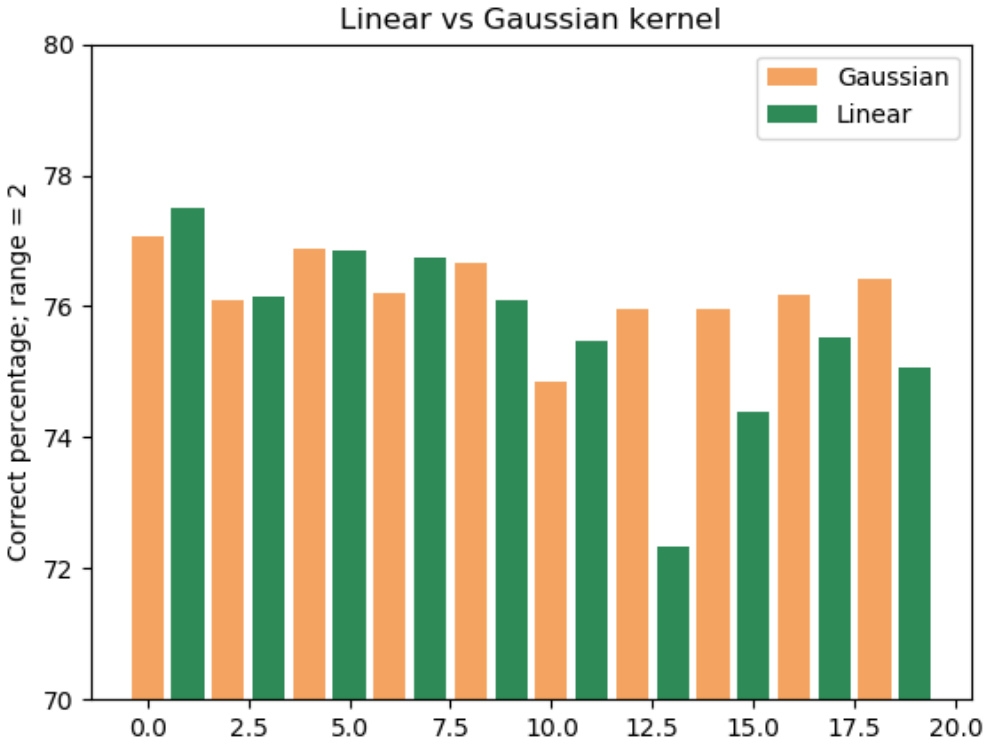
Antes de empezar hay que explicar que significa el eje y, "*Correct percentage; range = 2*". Esta es nuestra forma de medir que forma de aprendizaje ha realizado las mejores predicciones sobre los 800 ejemplos de validación.

Primero se entrena con los 3000 ejemplos de entrenamiento y se calcula con los de validación el porcentaje de ellos que se acercan al valor real. Y la expresión "se acercan" se utiliza porque para medir la efectividad hemos decidido que es mejor premiar a las predicciones que se acerquen al resultado real en vez de a las que acierten completamente. Esto se debe a que al haber tantas clases (casi 30), es muy difícil realizar predicciones exactas, por lo que será mejor que siempre se acerquen al valor real. "Range 2" expresa que se aceptarán por predicciones correctas aquellas que den como máximo 2 anillos más o menos que el valor real de número de anillos. Más tarde se medirá la eficacia de este método.

Una vez sabemos qué expresa el eje y, pasamos a mirar el eje x, en el que los valores no significan nada. En cambio, cada para de barras naranja-verde representa el mejor resultado obtenido para una iteración.

Una iteración consiste en buscar que valores de C y σ (esta última solo para el método linear) obtienen el mejor porcentaje de aciertos usando el método descrito anteriormente. Los valores que se han utilizado para probar son [0.01, 0.03, 0.07, 1, 3, 7, 10, 30], y con el kernel gaussiano se prueban todas las combinaciones de C y σ con estos valores.

Sabido todo esto y sin entrar en detalles todavía de qué son C y σ, podemos ver en el gráfico que para cada iteración el kernel gaussiano siempre ha encontrado una solución con mejor porcentaje de aciertos. Esto lo suponíamos, ya que el número de casos de entrenamiento (3000) es medianamente superior al número de atributos (8). Si bajamos mucho el número de ejemplos de entrenamiento…



Vemos que ahora, con 100 ejemplos de entrenamiento (y 3700 de validación) los resultados son distintos. Aunque el porcentaje de aciertos sigue siendo alto, en alguna ocasión el kernel lineal obtiene mejores resultados que el gaussiano. De reducir más el número de entrenamiento, o aumentar el de atributos, es probable que resultase conveniente utilizar el lineal.

Una vez sabemos que nos conviene utilizar el kernel gaussiano, vamos a adentrarnos en los 2 parámetros que utiliza este método y que hemos mencionado anteriormente. Pero antes hay que comprender dos conceptos: La varianza y el sesgo.

- La **varianza** se produce cuando los resultados se acercan demasiado a los ejemplos de entrenamiento. Esto causaría que no generalice lo suficiente y que los ejemplos distintos a los de entrenamiento no los clasifique bien.

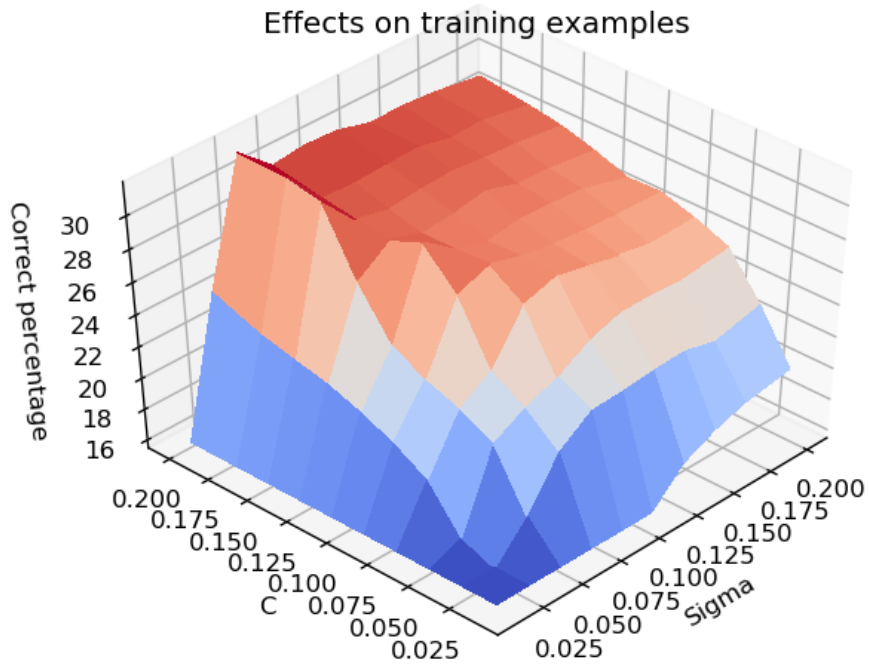
- El **sesgo** es lo contrario a la varianza, los resultados se alejan demasiado a los de entrenamiento por un exceso de simplificación. Por esto, todos los ejemplos los clasificará de manera incorrecta.

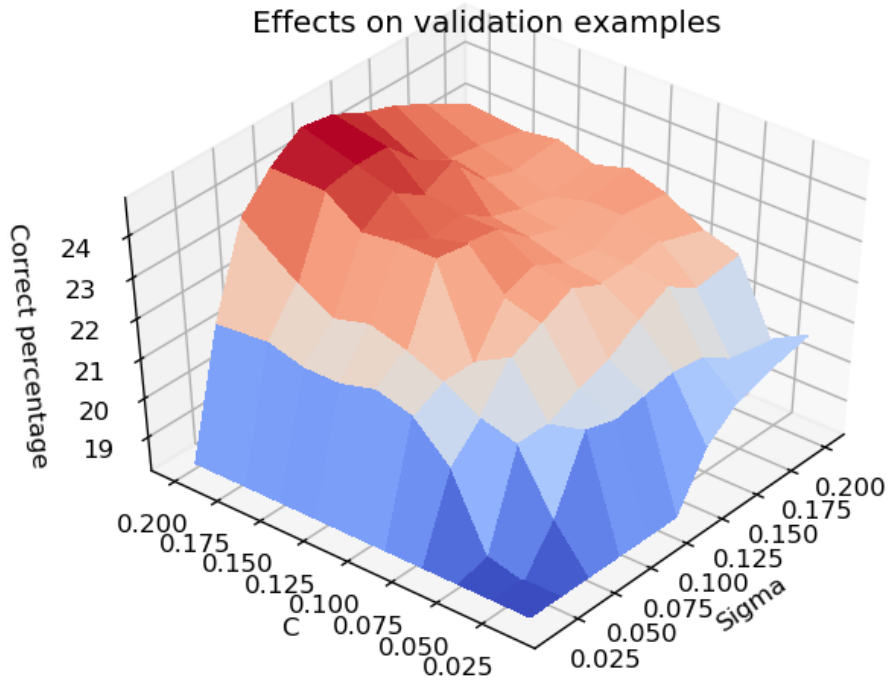
Sabiendo esto, vemos a ver que significa C y σ:

- **C** es **1/λ**. El factor de regularización lambda (λ) se verá en la explicación de los parámetros de la red neuronal, pero ahora hay que comprender que una C grande causará una gran varianza pero poco sesgo, mientras que una grande causará lo contrario.

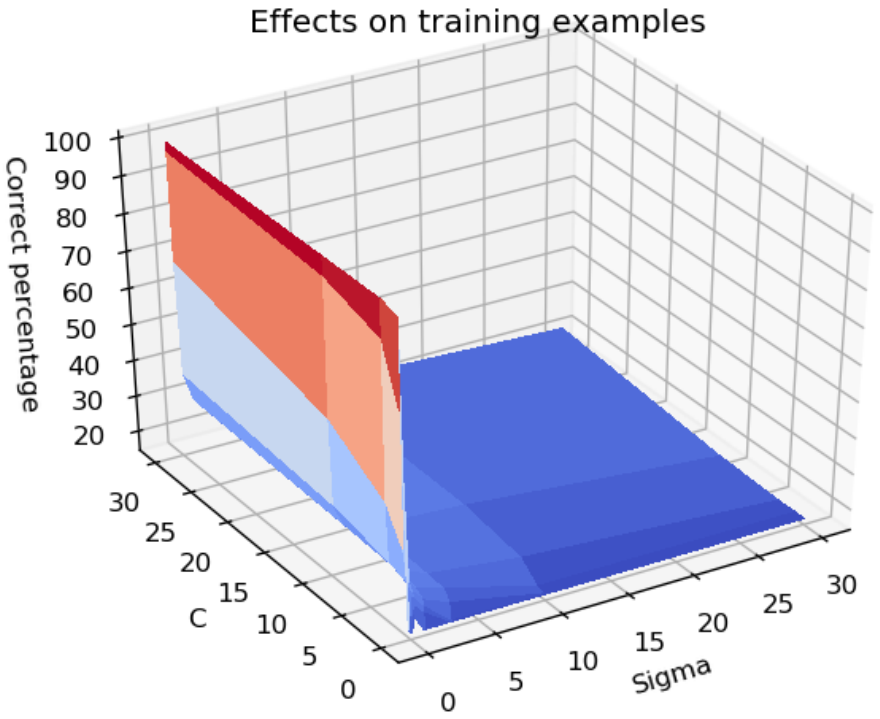
- **Sigma** (**σ**) es un valor que hace que las predicciones se vean de una manera más o más "suave", es decir, que no varíe tanto el resultado según nos alejamos de un valor. Un valor pequeño de σ hará que las variaciones sean menos suaves, provocando un menor sesgo y mayor varianza, mientras que uno grande hará lo contrario.

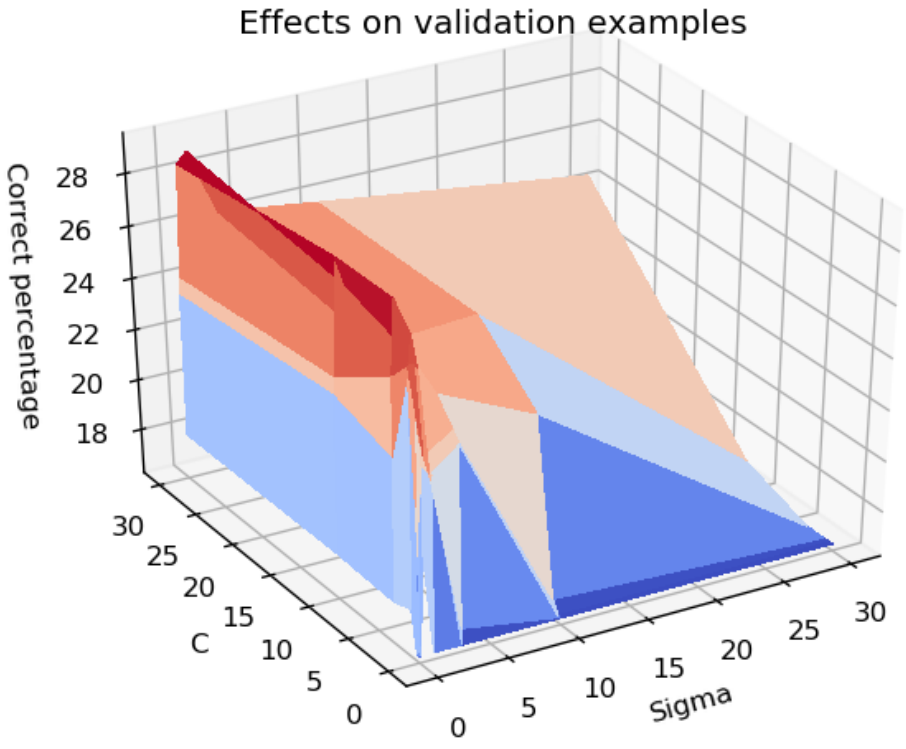
Vamos a coger 3000 ejemplos de entrenamiento y 800 de validación, y vamos a ver cómo afectan distintos valores de C y σ al entrenamiento. Usaremos para controlar la efectividad el método anterior, aunque esta vez solo daremos por correctos las predicciones que sean exactamente el valor real. Usaremos como valores [0.01, 0.03, 0.05, 0.07, 0.09, 0.11, 0.13, 0.15, 0.17, 0.20].





El primer gráfico muestra los efectos de cada combinación de C y σ para los ejemplos de entrenamiento, y el segundo su correspondencia con los de validación. Obviando que evidentemente el porcentaje de aciertos sobre los de entrenamiento es mayor, vemos que los 2 gráficos son similares. A ambos les benéfica un mayor valor de C y de σ, aunque hay un pico en un valor de σ pequeño. Pero para asegurarnos vamos a ver 2 gráficos más, con un rango de valores mayor para C y σ, que serán [0.01, 0.03, 0.07, 1, 3, 7, 10, 30].



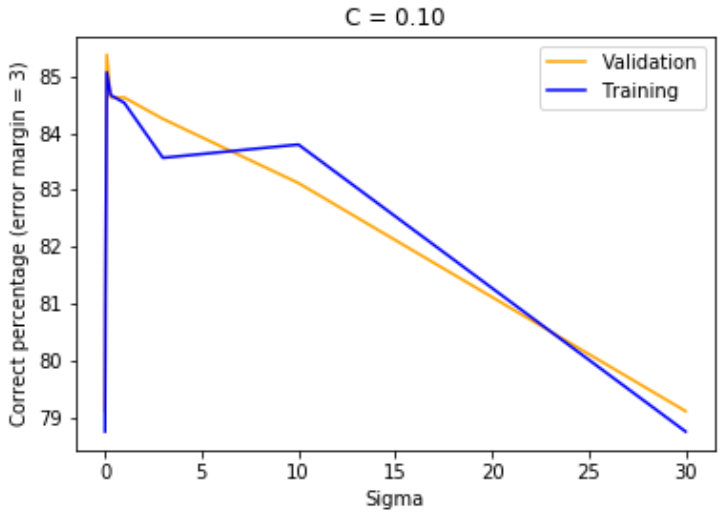


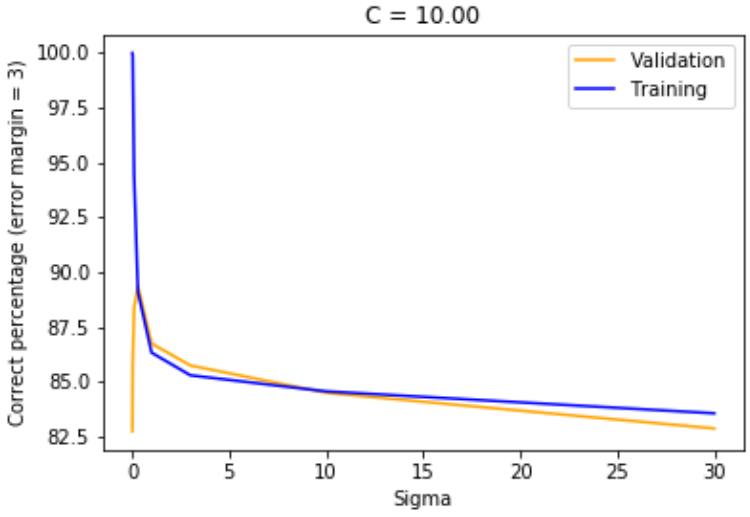
Se mostraba antes el gráfico con valores más pequeños porque este es bastante más confuso. Sin embargo, se aprecia en este a diferencia del anterior que los mejores resultados se obtienen con una σ relativamente pequeña, aunque coincida que la C debe ser grande. Esto tiene una implicación clara: una C grande y una σ pequeña ambas benefician a una mayor varianza, menor sesgo.

Esto significa que es beneficioso acercarse a los ejemplos de entrenamiento. Esto es probablemente debido a que los ejemplos de entrenamiento se parecen a los de validación. Como los ejemplos se han escogido aleatoriamente, es lógico que los ejemplos anómalos se hayan repartido entre entrenamiento y validación, preparando a la SVM para tratar ejemplos "raros".

Podemos visualizar de otra manera estos datos, realizando un corte por un valor determinado de C, para poderlo ver en un gráfico en 2D.

Hemos hecho esto, esta vez con un margen de error para el porcentaje de correctos de 3, con los mismos valores para C y σ [0.01, 0.03, 0.07, 1, 3, 7, 10, 30]. Visualizaremos por ejemplo C = 0.01, y C = 10.





Se observa que con C = 0.1 las rectas son bastante similares, beneficiando a los valores de σ menores, con lo que se llega a alcanzar aproximadamente un 85% de resultados considerados correctos. Con C = 10 se hace evidente que un valor alto de c causa un sobreajuste extremo a los valores de entrenamiento, llegando al 100% de ejemplos correctos para ellos. Aun así, un valor menor para σ sigue siendo beneficioso, llegando a casi un 90% de aciertos. Y como el valor máximo al que se ha llegado es con una C mayor, se ve fácilmente que lo mejor es una C grande y σ pequeño.

Vamos a mostrar algunos de los resultados que ofrece una SVM entrenada con un margen de error de 2, 3000 ejemplos de entrenamiento y 800 de validación. A continuación se muestran 10 ejemplos sobre unos ejemplos de prueba (distintos a los de entrenamiento y validación), extraído directamente de un print realizado por el programa.

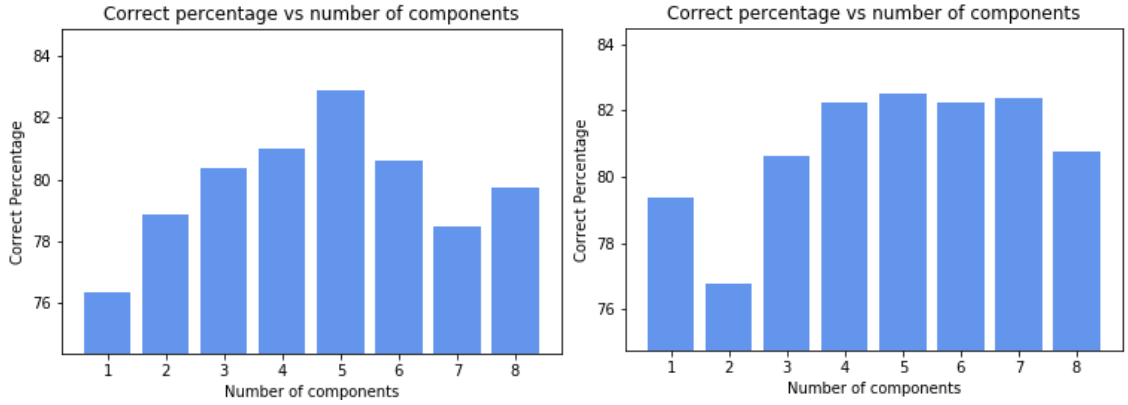
```
Valores de x:  [-1.     0.535  0.44   0.15   0.6765 0.256  0.139  0.26  ]
El valor predicho es: 12
El valor real es: 12
Valores de x:  [0.   0.395 0.29  0.095 0.319 0.138 0.08  0.082]
El valor predicho es: 7
El valor real es: 7
Valores de x:  [1.   0.385  0.305  0.105  0.3315 0.1365 0.0745 0.1  ]
El valor predicho es: 10
El valor real es: 7
Valores de x:  [0.    0.49  0.375  0.115  0.4615 0.204  0.0945 0.143 ]
El valor predicho es: 8
El valor real es: 8
Valores de x:  [0.    0.515  0.395  0.125  0.6635 0.32   0.14   0.17  ]
El valor predicho es: 8
El valor real es: 8
Valores de x:  [0.   0.475 0.375 0.11  0.456 0.182 0.099 0.16 ]
El valor predicho es: 8
El valor real es: 9
Valores de x:  [-1.     0.46   0.35   0.125  0.5165 0.1885 0.1145 0.185 ]
El valor predicho es: 10
El valor real es: 9
Valores de x:  [1.   0.675 0.54  0.21  1.593 0.686 0.318 0.45 ]
El valor predicho es: 11
El valor real es: 11
Valores de x:  [-1.     0.58   0.49   0.13   1.1335 0.586  0.2565 0.237 ]
El valor predicho es: 9
El valor real es: 9
Valores de x:  [0.   0.205 0.15  0.065 0.04  0.02  0.011 0.013]
El valor predicho es: 4
El valor real es: 4
```

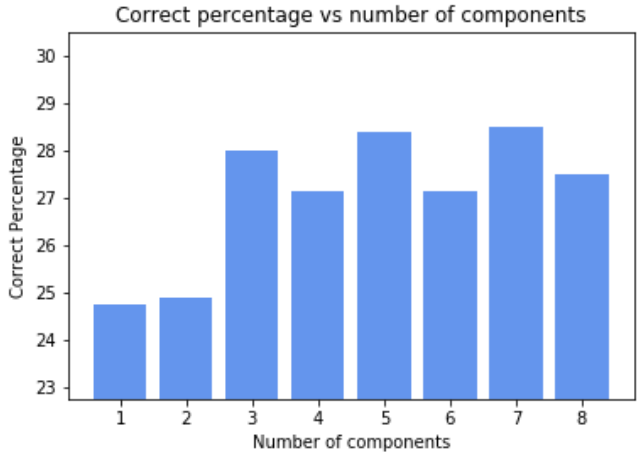
Principal Component Analysis

Procedamos a observar otro aspecto de la carga de ejemplos, que aunque no está directamente relacionado con las SVM, las utilizaremos para las comprobaciones. Vamos a comprobar qué número de componentes o atributos son mejores para realizar los entrenamientos. Hasta ahora hemos estado utilizando los 8 que nos proporcionan, pero podemos utilizar un método para reducir este número.

El **análisis de componente principal**, también conocido como *Prinicpal Component Analysis* (o PCA por sus siglas), es un método de preprocesado de datos que nos permite reducir el número de componentes, tratando de mantener las características originales. Esto se utiliza principalmente para poder representar gráficamente los datos, o para reducir el coste computacional del aprendizaje.

Vamos a ver las siguientes gráficas:





Para realizar estos gráficos, se ha utilizado el algoritmo PCA para reducir el número de componentes a tantos como indica el eje x, y luego se ha buscado el mejor porcentaje de aciertos para cada combinación de valores de C y σ [0.01, 0.03, 0.07, 1, 3, 7, 10, 30]. Sin embargo, el conjunto de ejemplos de entrenamiento y validación aleatorios para cada número de elementos es aleatorio y distinto, por lo que esto podría introducir variaciones en el diagrama.

Para contrarrestar las variaciones se han hecho 3 gráficos, los dos primeros permitiendo alejarse como máximo 2 al número de anillos predicho respecto a la realidad, y el tercero buscando el valor exacto (a esto se debe la gran diferencia entre el porcentaje de aciertos). El único patrón relativamente claro que siguen es que el porcentaje de aciertos para 1 o 2 componentes tiende a ser inferior. Esto es indicación de que se ha perdido información al reducir los componentes. El resto de número de componentes proporcionan resultados similares.

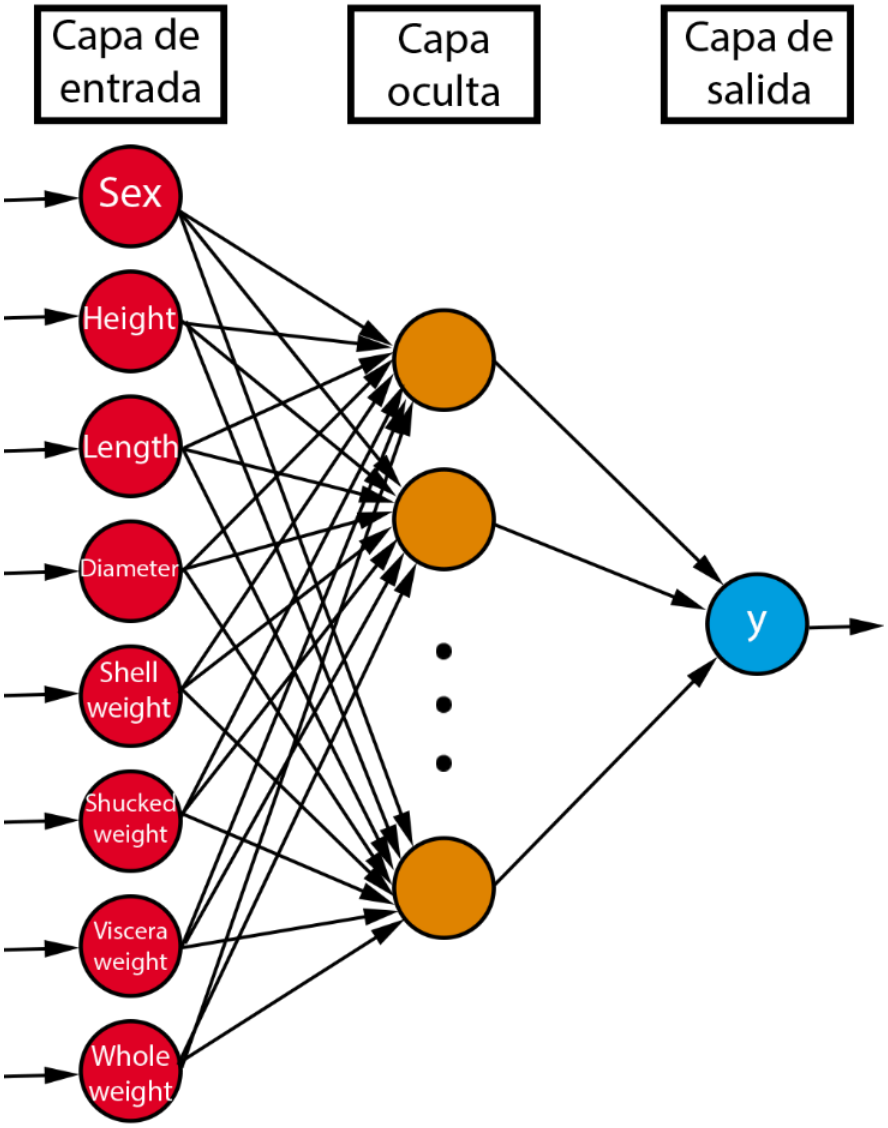
Entonces podemos concluir que mientras usemos 3 o más componentes los resultados serán parecidos. ¿Pero por qué 3?

Mi teoría es que hay 3 aspectos diferenciados en los atributos de los abulones: El sexo, la altura y el peso. Si bien hay 8 atributos, todos ellos se pueden clasificar dentro de estos 3 aspectos, y entre todos los que comparten la clase, por ejemplo, la altura, intuyo que guardarán una estrecha relación.

Red neuronal

Las redes neuronales son el método más conocido de aprendizaje automático, y uno de los más efectivos. Una red neuronal consta de 3 partes: La capa de entrada, que será nuestros ejemplos con todos sus atributos, las capas ocultas, en las que se realiza el entrenamiento, y la capa de salida, en la que se calcula el resultado final. Puede haber una o varias capas ocultas, pero en nuestro caso solo utilizaremos una.

En este esquema se muestra como sería en nuestro caso:



Cada una de las capas tiene un determinado número de nodos:

- La **capa de entrada** constará de los 8 atributos de nuestros ejemplos, mientras que la de salida de un único.

- La **capa de salida**, aunque se haya dibujado un único nodo para simplificar, constará de tantos nodos como número de clases haya (29). La red neuronal generará valores para todos estos nodos, y como resultado nos quedaremos con el número de nodo que haya obtenido un mayor valor, lo que significará que el valor que la red cree que es más probable que sea el resultado correcto.

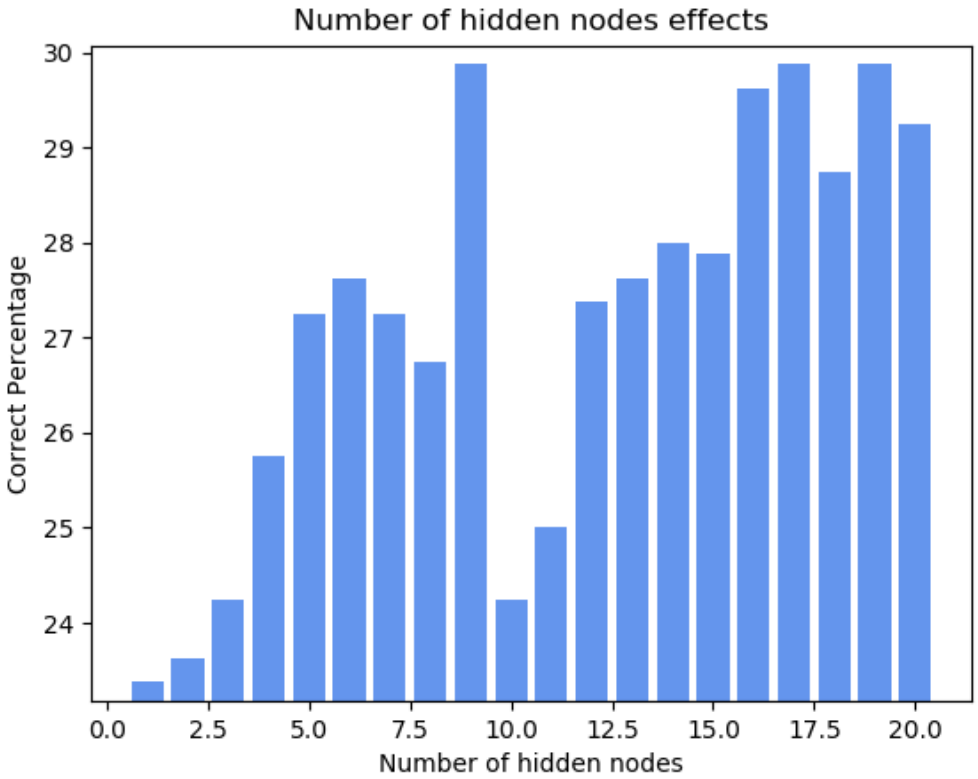
- La **capa oculta** consta de un número indeterminado de nodos, que podremos decidir nosotros. Un número pequeño produce resultados más lineares, mientras que uno grande producirá soluciones más complejas. Experimentaremos con este número.

Para la red neuronal tendremos que elegir dos parámetros:

- El parámetro de **regularización λ**: Este valor hace que la función a minimizar evite acercarse demasiado a los ejemplos de entrenamiento. Por tanto, un valor grande de lambda hará que disminuya la varianza y aumente el sesgo.

- El **número de nodos en la capa oculta:** Aumentar el número aumenta el coste computacional, pero como resultado genera funciones más complejas para seleccionar las salidas.

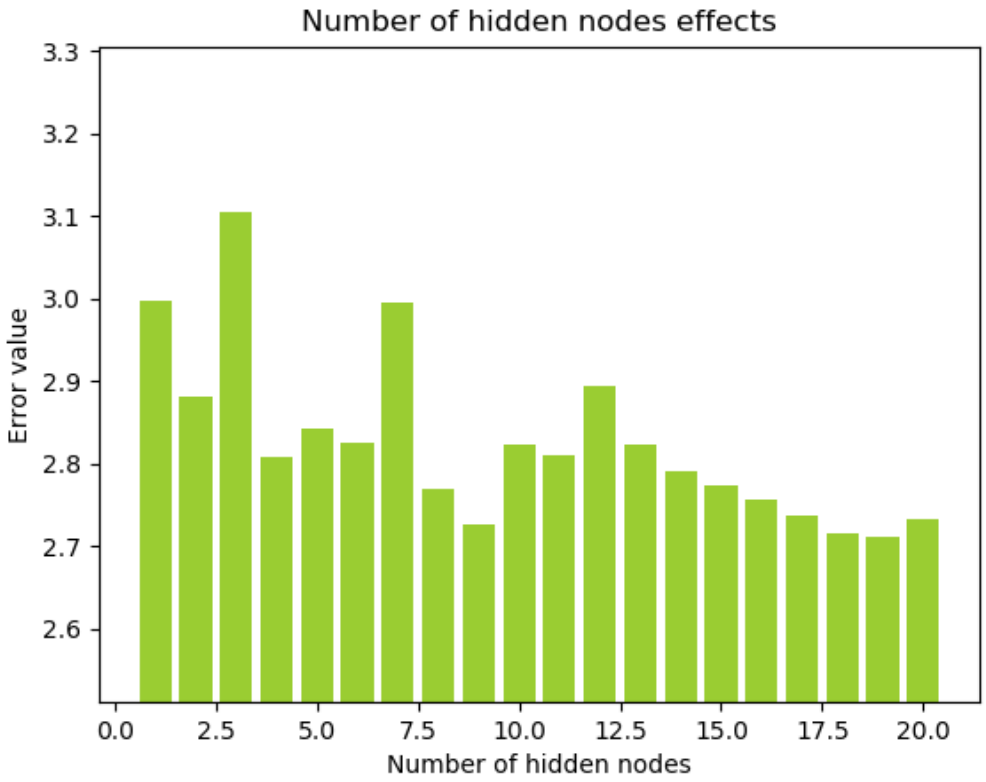
Empezaremos por ver cómo afecta el número de nodos al resultado:



El eje x representa el número de nodos en la capa oculta que se han usado en la red neuronal. El eje y representa el porcentaje de aciertos sobre los 800 ejemplos de validación utilizando el método que teníamos en las SVM, y en este caso solo damos por válidos las predicciones que acierten completamente con el número de anillos real.

Cabe destacar que para todos los números de nodos se ha probado con el mismo valor de regularización (0), se ha hecho con el mismo set de datos de entrenamiento y validación, y que la inicialización de los valores de theta (con los que empieza a trabajar en los entrenamientos) es necesariamente distinta para cada número, por lo que introduce cierta incertidumbre.

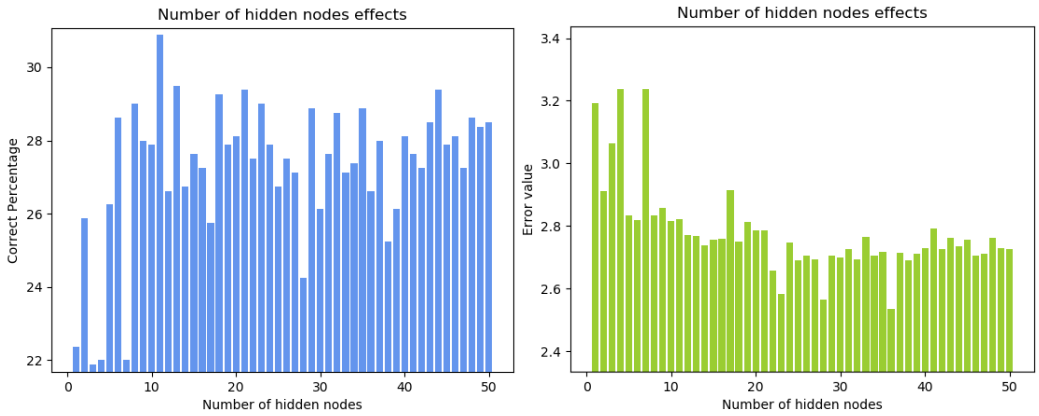
Como se puede observar, los peores resultados se obtienen para un número de nodos muy bajo. Eso significa probablemente que el problema requiere algo más que una solución completamente lineal. Pero no nos detenemos aquí, porque ahora vamos a ver otra posible métrica para la efectividad del sistema.



Esta gráfica se ha calculado a la vez que la anterior con los mismos datos. El eje y en este caso representa el valor final obtenido para la función de error que la red neuronal intenta minimizar. Es decir, un menor valor significa mejores resultados sobre los ejemplos de entrenamiento.

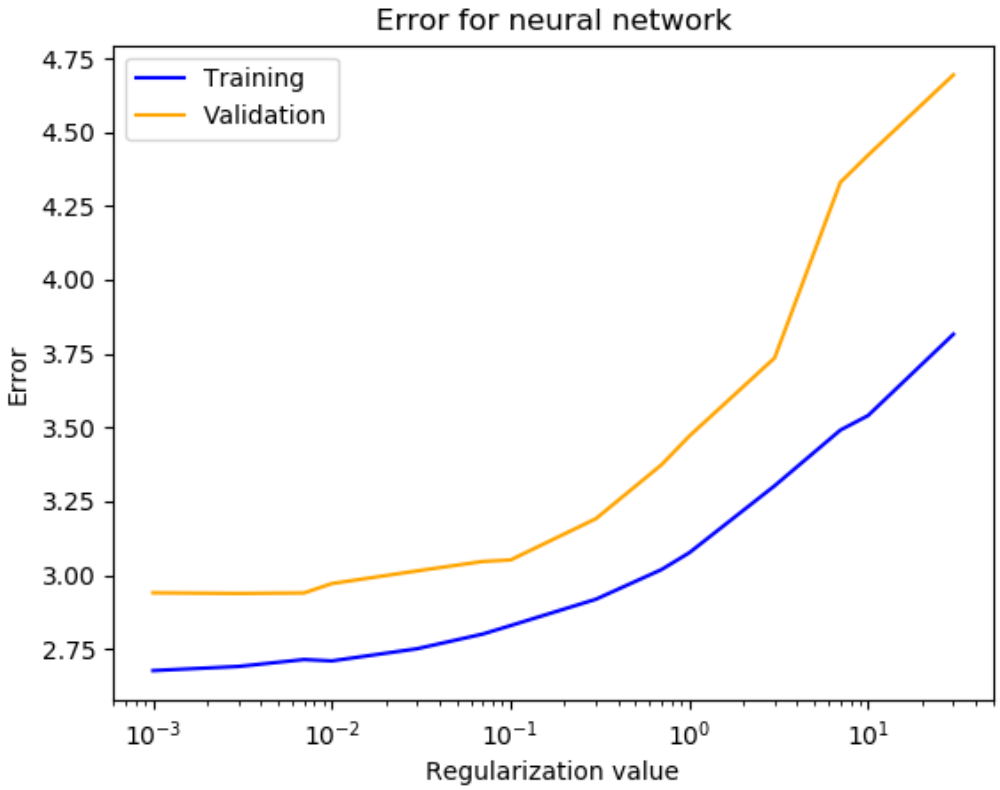
Si la miramos y comparamos los resultados de la anterior, se observa que generalmente cuando se obtenían mejores resultados en la gráfica anterior, en esta también se obtienen mejores resultados.

Aunque parezca que a mayor número de nodos mejores resultados, vamos a mirar otra gráfica con un mayor número de nodos ocultos para verificar si esto es cierto.



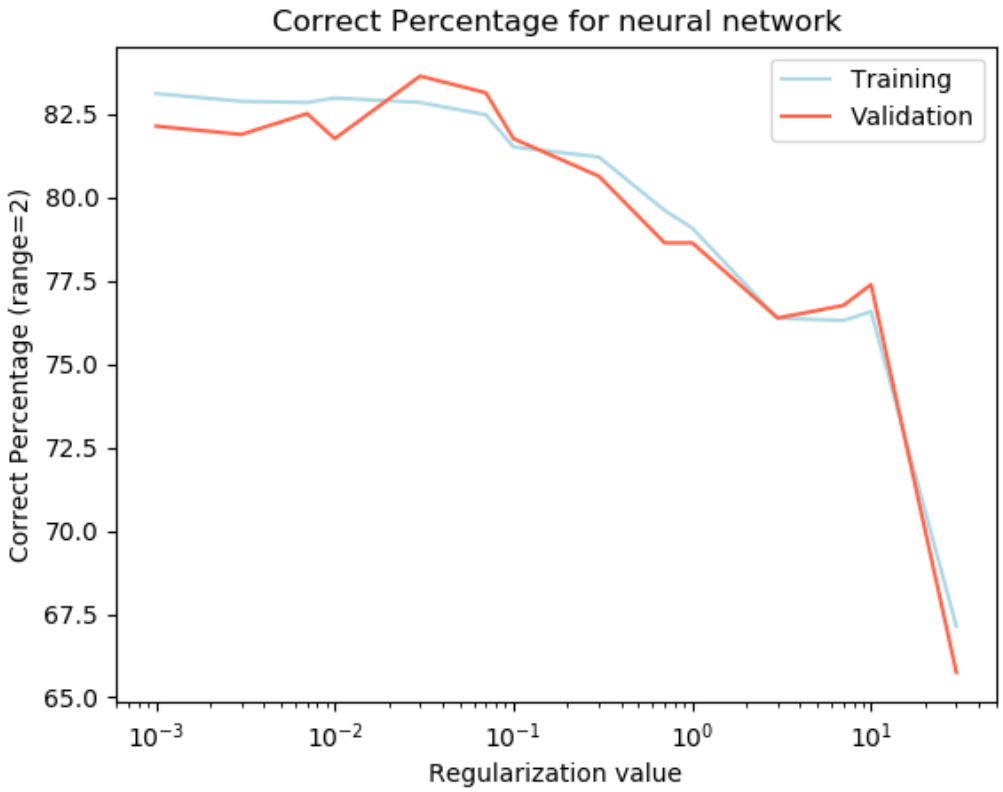
Tras un tiempo en ejecución obtenemos estas gráficas. Vemos que con un número muy pequeño de nodos los resultados son peores que con un número mayor, pero también vemos que los resultados a partir de cierto punto no varían demasiado, por lo que con utilizar un número de nodos de entre 10 y 20 será suficiente.

Pasemos entonces al otro parámetro, el factor de regularización.



El eje x, en escala logarítmica, representa el valor de λ. El y vuelve a representar el error calculado por la función. Como denota la gráfica ambos errores crecen con el valor λ, lo que implicaría que los ejemplos de entrenamiento son parecidos a los de validación, y que conviene que los resultados se ajusten a los ejemplos de entrenamiento. Obviamente el error para los ejemplos de validación es mayor, porque no se han utilizado para entrenar.

Se ha probado con los valores [0.001, 0.003, 0.007, 0.01, 0.03, 0.07, 0.1, 0.3, 0.7, 1, 3, 7, 10, 30] para λ. Vamos a verlo también con el porcentaje de correctos:



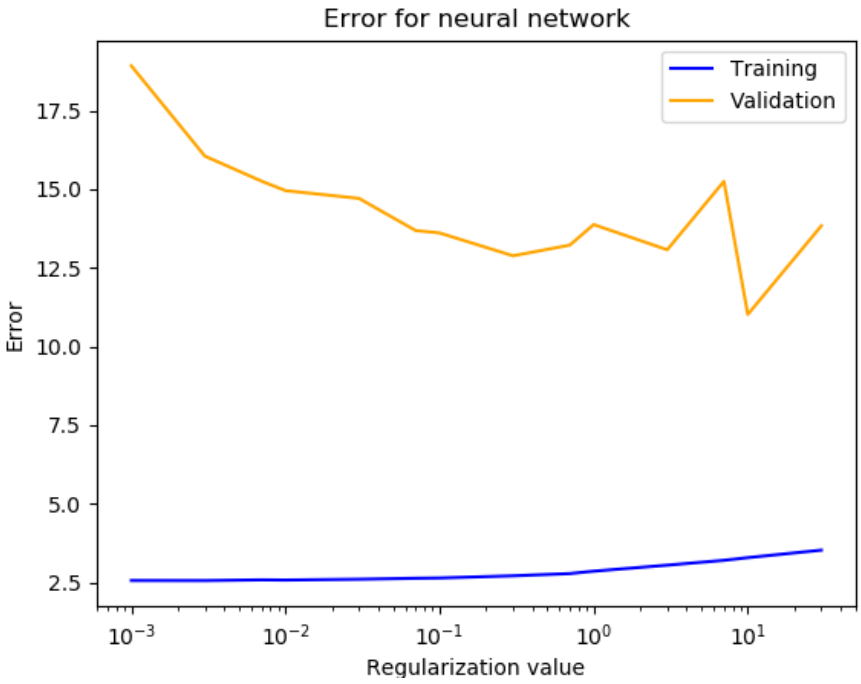
Esta gráfica se ha generado al mismo tiempo con la otra y con el mismo conjunto de datos, con el método de porcentaje de correctos en los de validación y un margen máximo de error de 2. Se observa que los mejores valores también son para una regularización pequeña. Pero lo que más destaca en esta gráfica es que obtiene mejores porcentajes de aciertos con los ejemplos de validación que con los de entrenamiento en algunos casos con λ pequeño. Esto es raro, ya que con un valor de regularización pequeño suele haber sobreajuste a los ejemplos de entrenamiento, pero al ser tan similares los entrenamientos de la validación, afecta positivamente.

Vamos a mostrar ahora unos cuantos ejemplos de predicciones hechas por una red neuronal de 15 nodos ocultos, factor de regularización de 0.01. Se hacen sobre ejemplos distintos a los de validación y entrenamiento. El primer valor de estas x es siempre 1 y se debe a un ajuste para poder realizar el entrenamiento en las redes neuronales.

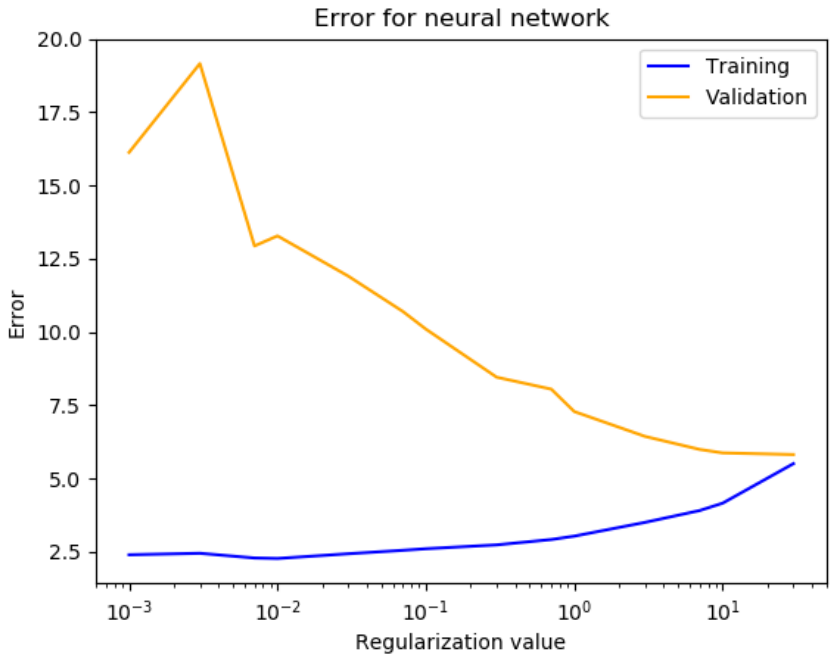
```
Valores de x:  [1.    0.    0.59  0.465  0.195  1.0885 0.3685 0.187  0.375 ]
El valor predicho es: 16
El valor real es: 17
Valores de x:  [1.    0.    0.475  0.35   0.12   0.4905 0.2035 0.13   0.135 ]
El valor predicho es: 8
El valor real es: 7
Valores de x:  [1.    0.    0.37   0.28   0.095  0.2655 0.122  0.052  0.08  ]
El valor predicho es: 7
El valor real es: 7
```

```
Valores de x:  [ 1.   -1.    0.545  0.41   0.145  0.873  0.3035 0.196  0.31  ]
El valor predicho es: 13
El valor real es: 18
Valores de x:  [ 1.   -1.   0.6   0.45  0.16  1.142 0.539 0.225 0.307]
El valor predicho es: 9
El valor real es: 10
Valores de x:  [ 1.   -1.    0.475  0.37   0.11   0.4895 0.2185 0.107  0.146 ]
El valor predicho es: 9
El valor real es: 8
Valores de x:  [1.   1.   0.505 0.425 0.14  0.85  0.275 0.1625 0.285 ]
El valor predicho es: 13
El valor real es: 19
Valores de x:  [1.   1.   0.615 0.5   0.165 1.327 0.6   0.3015 0.355 ]
El valor predicho es: 10
El valor real es: 10
Valores de x:  [1.   1.   0.515 0.405 0.12  0.646 0.2895 0.1405 0.177 ]
El valor predicho es: 9
El valor real es: 10
Valores de x:  [1.   1.   0.57  0.45  0.165 0.903 0.3305 0.1845 0.295 ]
El valor predicho es: 13
El valor real es: 14
```

Ahora vamos a realizar otras pruebas sobre los datos de entrenamiento, utilizando la red neuronal esta vez. En este caso, en vez de cargarlos aleatoriamente, vamos a entrenar con los ejemplos “normales”, y a validar con los “raros”. Para los propósitos de esta prueba vamos a entiende como normales los ejemplos con más de cuatro anillos y menos de 16, ya que la media es 9 y la desviación típica aproximadamente 4. El resto los usaremos para validar.



El error sobre los datos de validación es descomunal, ya que no se parecen en nada a los entrenamientos que se han realizado. Vamos a ver si hacemos lo contrario, entrenamiento con los “raros” y validación con los “normales”.



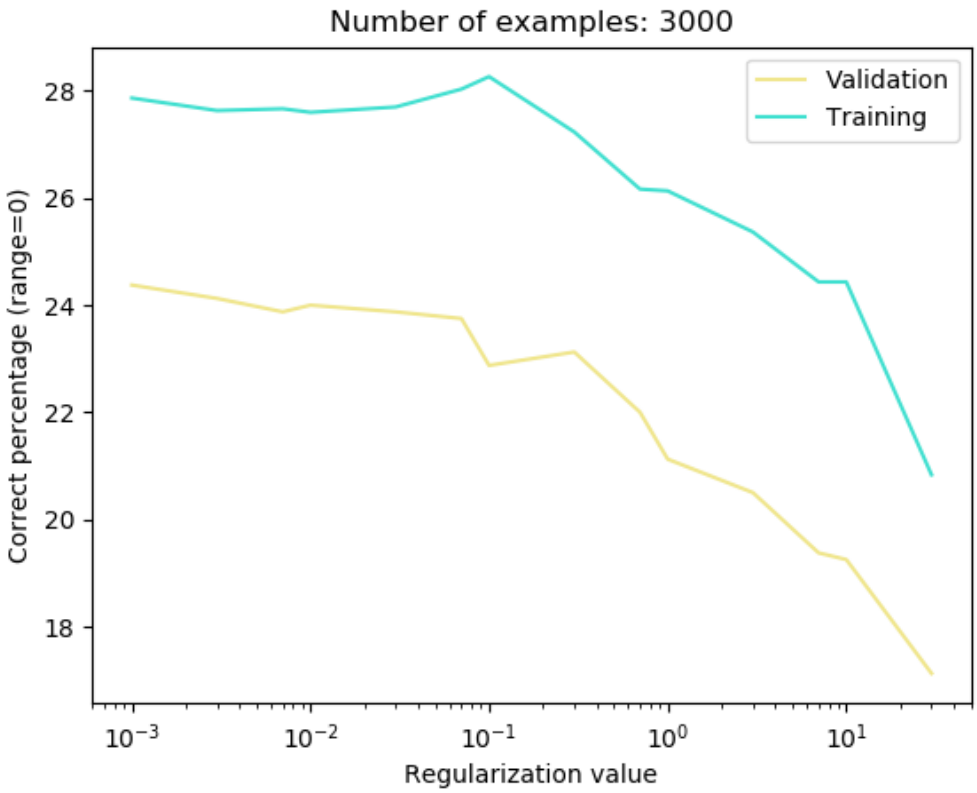
En este caso, sorprendentemente, el error de validación empieza a bajar mucho con un valor muy alto de regularización. Sin embargo, el error sigue siendo de aproximadamente 5, mucho más alto de lo que vimos con el entrenamiento de datos aleatorios.

Con estos experimentos podemos concluir que lo óptimo es entrenar con ejemplos aleatorios y variados, porque de hacerlo de otra manera conseguimos que no logre clasificar correctamente ejemplos fuera de los de entrenamiento.

Regresión logística

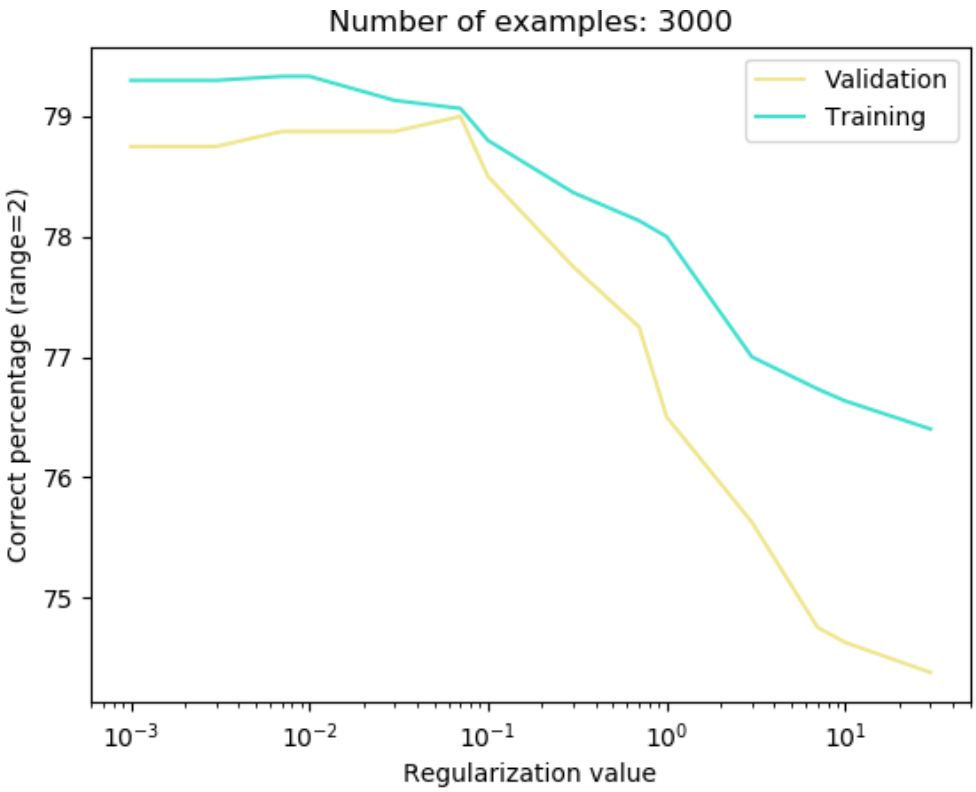
La regresión logística es otro de los métodos que podemos utilizar para el aprendizaje automático. Se basa a grandes rasgos en hacer iteraciones a una función que poco a poco minimizará el valor de otra función, la de coste.

Una vez la tenemos implementada, hay que elegir de nuevo el factor de regularización. Recordemos que un valor de regularización alto hace que las predicciones se alejen más de los ejemplos de entrenamiento. Vamos a ver esto en un gráfico:



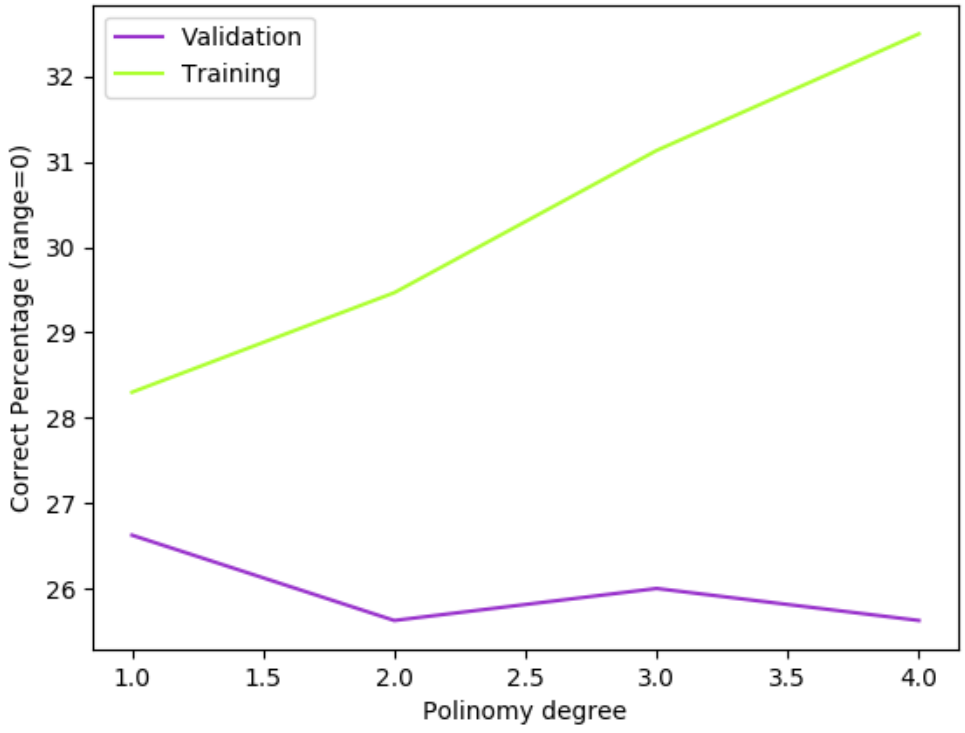
Hemos entrenado con 3000 ejemplos, la y representa como ya es habitual el porcentaje de predicciones correctas en la validación, obligándolas a ser exactas. El mejor porcentaje de aciertos para los ejemplos de validación se obtienen con un valor de regularización extremadamente bajo en este caso. También observamos que, como era de esperar, hay siempre más aciertos en los que se han utilizado para entrenar.

Los resultados obtenidos son muy parecidos a los que hemos logrado con otros métodos. Pero aun así vamos a ver otro gráfico, con un margen de error de 2 en vez de 0 para considerar una predicción correcta.



Nada de esta gráfica nos sorprende. Lo único que cambia es un porcentaje de aciertos mayor, debido al mayor margen de error.

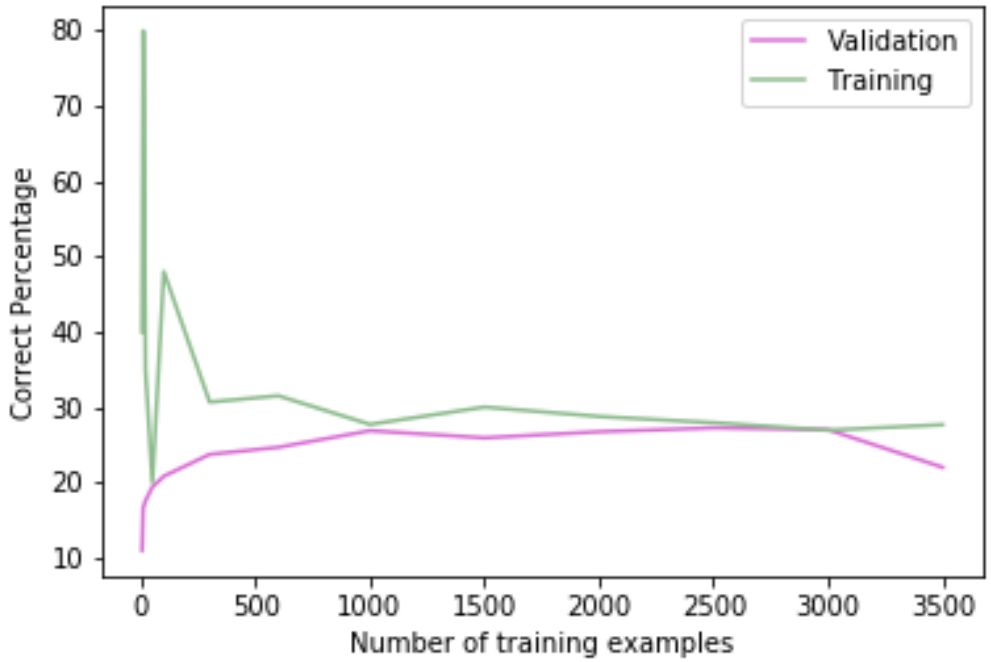
Ahora vamos a comprobar otro factor con los ejemplos de entrenamiento. Antes de emepezar a entrenar, vamos a coger todos los atributos y a multiplicar todos entre todos, para poder crear un polinomio de datos de entrada mucho mayor. Con esto conseguiremos un número de atributos de entrada mucho más grande. Tras hacer esto con diferentes grados de polinomio, obtenemos estos resultados:



El eje x representa el grado del polinomio usado y el y, el porcentaje de aciertos exactos. Solo se ha hecho hasta 4 porque es computacionalmente muy costoso elevar a números tan grande un polinomio de 8 elementos (los atributos de entrada). Para todos se han utilizado los mismos ejemplos de entrenamiento y validación, y el mismo factor de regularización de 0.01.

Observamos que los ejemplos de entrenamiento cada vez consiguen mejores resultados, mientras que los de validación, al contrario. Para los de entrenamiento es lógico que con más atributos sea capaz de generar resultados más complejos y por tanto ajustarse más al entrenamiento. Para los de validación no le afecta que sea más complejo, y lo que le beneficia es una función más lineal.

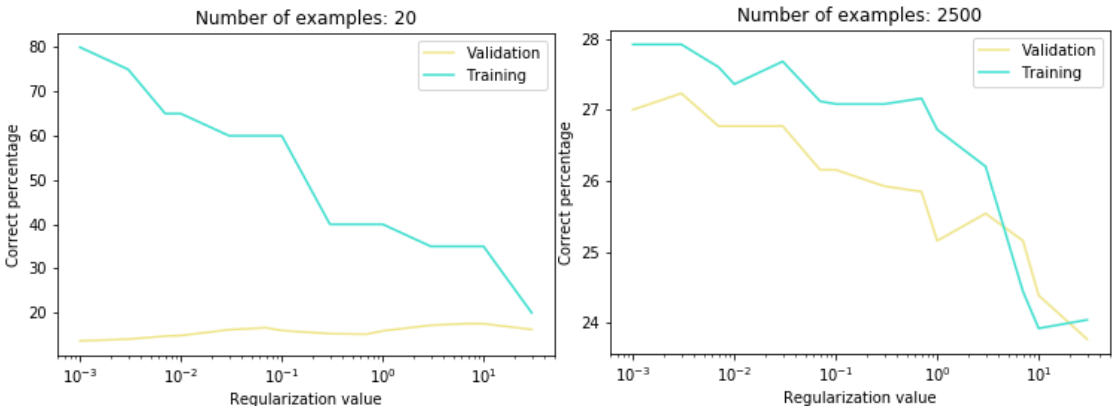
Una vez hemos comprendido que lo mejor es un valor de regularización bajo y un polinomio de grado uno, vamos a ver las diferencias entre coger más o menos ejemplos de entrenamiento. Haremos varias iteraciones, cogiendo N ejemplos de entrenamiento y 3800 – N ejemplos de validación, y veremos los resultados. N lo seleccionaremos entre los siguientes valores: [5, 10, 20, 50, 100, 300, 600, 1000, 1500, 2000, 2500, 3000, 3500].



Se ha juzgado con el porcentaje de aciertos en las predicciones de todos los ejemplos de validación, y se ha buscado el mejor valor de λ para cada nº de entrenamiento.

El porcentaje de aciertos para los ejemplos de entrenamiento es muy alto cuando hay pocos (llegando al 80%), pero muy bajo para los de validación. Se debe a que es muy fácil ajustarse a todos, pero no representa al total de la población. Según aumenta y a partir de los 1000 se empiezan a obtener los mejores resultados, entre ellos con 3000 ejemplos, que es la cantidad que hemos ido usando hasta el momento.

Vamos a ver más en detalle el caso de los 5 ejemplos y el de 2500, para ver los resultados con cada λ.



Con 20 hay muy pocos aciertos en validación y muchos de entrenamiento, mientras que el con 2500 el gráfico es muy similar a los que hemos ido obteniendo.

Vamos a ver unas cuantas predicciones, con 3000 ejemplos de entrenamiento, los 8 atributos originales, un margen de error para las predicciones de 2 y 800 ejemplos de validación. Se utiliza el valor de λ con mejor resultado.

```
Valores de x:  [1.    0.    0.575  0.455  0.155  0.8725 0.349  0.2095 0.285 ]
El valor predicho es: 9
El valor real es: 8
Valores de x:  [ 1.   -1.    0.605  0.47   0.165  1.2315 0.6025 0.262  0.2925]
El valor predicho es: 9
El valor real es: 11
Valores de x:  [1.    0.    0.295  0.215  0.075  0.116  0.037  0.0295 0.04 ]
El valor predicho es: 6
El valor real es: 8
Valores de x:  [ 1.   -1.    0.685  0.53   0.17   1.56   0.647  0.383  0.465]
El valor predicho es: 11
El valor real es: 11
Valores de x:  [1.    1.    0.57  0.44  0.19  1.018 0.447 0.207 0.265]
El valor predicho es: 9
El valor real es: 9
Valores de x:  [ 1.   -1.    0.49   0.355  0.155  0.981  0.465  0.2015 0.2505]
El valor predicho es: 10
El valor real es: 8
Valores de x:  [1.    0.    0.46   0.35   0.11   0.4675 0.2125 0.099  0.1375]
El valor predicho es: 8
El valor real es: 7
Valores de x:  [ 1.   -1.    0.675  0.515  0.15   1.312  0.556  0.2845 0.4115]
El valor predicho es: 10
El valor real es: 11
Valores de x:  [1.    0.    0.375  0.305  0.115  0.2715 0.092  0.074  0.09 ]
El valor predicho es: 7
El valor real es: 8
Valores de x:  [1.    1.    0.57   0.45   0.16   0.9715 0.3965 0.255  0.26 ]
El valor predicho es: 10
El valor real es: 12
```

Comparaciones

Ahora que tenemos los 3 algoritmos funcionando, vamos a comprobar con un simple gráfico de barras cual es la mejor opción.

Hemos utilizado 3000 ejemplos de entrenamiento y 800 de validación, los mismos para todos los algoritmos. Utilizamos los 8 atributos también.

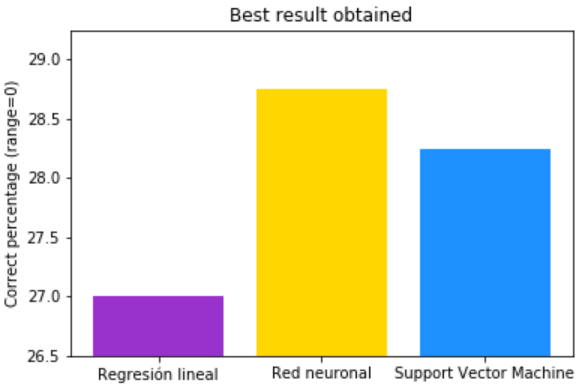
Para la SVM hemos utilizado el kernel gaussiano y pasado por todos los valores de C y σ en [0.01, 0.03, 0.07, 1, 3, 7, 10, 30], y hemos escogido el mejor valor obtenido.

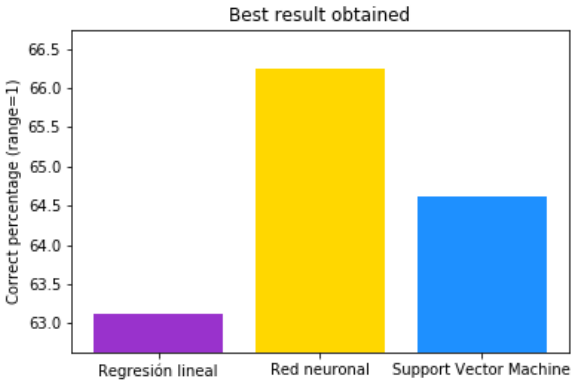
Para la red neuronal utilizamos 15 nodos en la capa oculta, y escogido entonces el mejor valor obtenido habiendo probado con λ = [0.001, 0.003, 0.007, 0.01, 0.03, 0.07, 0.1, 0.3, 0.7, 1, 3, 7, 10, 30].

Para la regresión logística hemos utilizado un polinomio de grado uno, y escogido el mejor valor obtenido habiendo probado con λ = [0.001, 0.003, 0.007, 0.01, 0.03, 0.07, 0.1, 0.3, 0.7, 1, 3, 7, 10, 30].

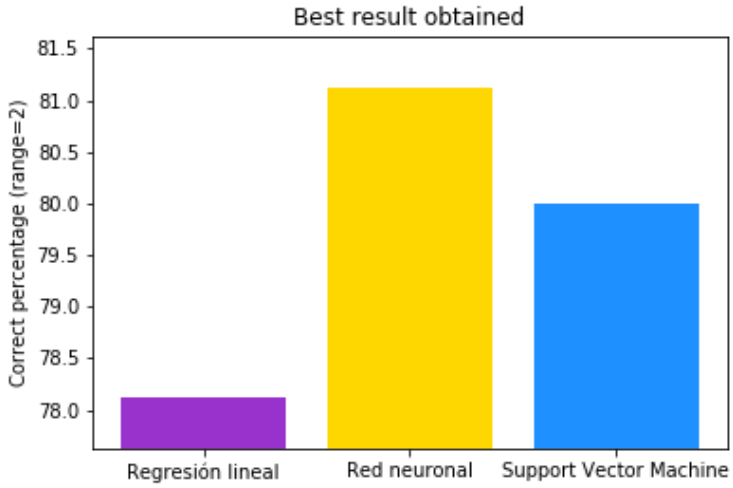
Hemos utilizado para mirar la corrección el porcentaje de aciertos en predicciones hechas en los ejemplos de validación. Tenemos 3 gráficos diferentes, permitiendo márgenes de error de 0, 1 y 2 respectivamente.

Observamos que

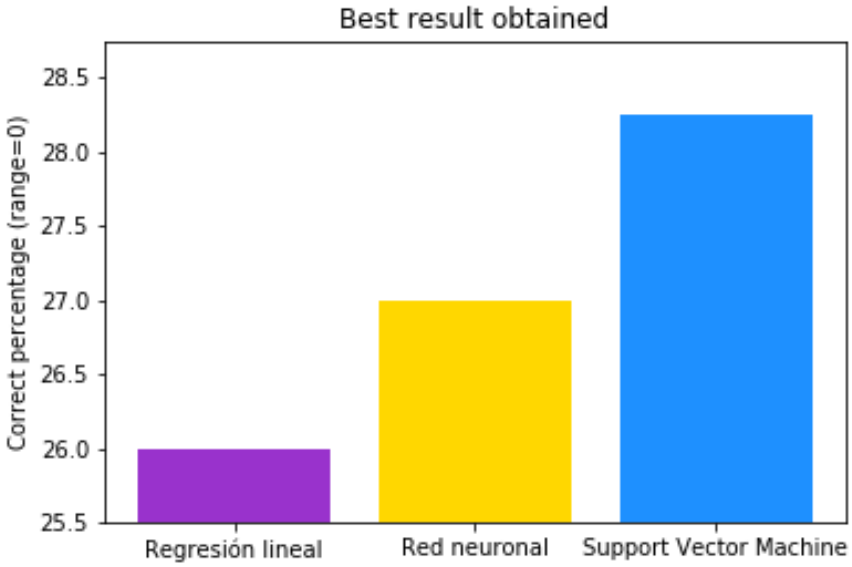




O



Observamos que el mejor resultado para cualquier margen de error lo obtiene la red neuronal, y el peor la regresión linear. Esto lo hemos repetido varias veces, con distinto set de entrenamiento y validación, y consistentemente se obtienen peores resultados con la regresión lineal. Sin embargo, aunque normalmente se obtienen ligeramente peores resultados con SVM que con redes neuronales, en alguna ocasión le supera, como en el caso a continuación:



Este caso es raro, por lo que podemos considerar que el método con el que obtenemos mejores resultados es la red neuronal, seguida de las SVM y, por último, la regresión lineal.

Comentarios

Hay algunas pruebas que no se han tratado en el documento por producir resultados poco relevantes. Entre ellas, están las funciones errorAmmountSVM() y errorAmmountNN(), que utilizan otra forma de medir el error en los ejemplos de entrenamiento.

Trataban de premiar los resultados que se acercaban, elevando al cuadrado la diferencia entre el valor real y el predicho. Sin embargo, los resultados obtenidos eran muy parecidos a correctPercentage.

También hay un parámetro en loadDataShuffled() que sirve para agrupar las y en clases más grandes, para facilitar las predicciones reduciendo el número de clases. El efecto que esto producía era el equivalente al utilizar correctPercentage(), por lo que no se ha considerado relevante.

Internamente también se ha probado a poner otros valores al sustituir M, F e I por números, pero los resultados no se veían variados en absoluto.

Hay que comentar también que el código puede haberse visto alterado ligeramente, y no se produzcan exactamente los gráficos utilizados. Pero todo el código se ha utilizado con ligeras variaciones para generar toda la información en este documento.

Código

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import random
import scipy.optimize as opt
from scipy.io import loadmat
from sklearn.svm import SVC
from sklearn import decomposition
from sklearn import datasets
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from sklearn.preprocessing import PolynomialFeatures

#Coge el archivo abalone.txt y sustituye F M e I
#por valores numéricos que pueda leer luego
#Almacena la sustitución en abaloneProcessed.txt
def preprocess():
    f = open("abalone.txt", 'r')
    o = open("abaloneProcessed.txt", 'w')
    c = f.read(1)
    while True:
        if not c:
            break
        if c == 'M':
            o.write('-1')
        elif c == 'F':
            o.write('1')
        elif c == 'I':
            o.write('0')
        else:
            o.write(c)
        c = f.read(1)
    f.close()
    o.close()
```

```
preprocess()

#Precondición
#0 < Numtraining < 3800; 0 < components < 9; 1 < groupsize < 30
#Carga los datos aleatoriamente
def loadDataShuffled(decompose=False, components=2, shrinky=False,
                     groupSize=2, numTraining = 3000):
    data = np.genfromtxt('abaloneProcessed.txt', delimiter=',')

    x = data[:,:-1]
    y = data[:,-1]

    #Si se quiere reducir los parámetros con PCA
    if decompose == True:
        pca = decomposition.PCA(n_components = components)
        pca.fit(x)
        x = pca.transform(x)

    #Si se quiere agrupar la y en grupos
    if shrinky == True:
        y = y // groupSize

    join = np.hstack((x, np.c_[y]))
    np.random.shuffle(join)

    #Se vuelven a separar tras barajearse
    xShuffled = join[:,:-1]
    yShuffled = join[:,-1]

    #numTraining decide cuantos ejemplos hay para entrenamiento,
    #y cuantos hay para validación (3800 - numTraining)
    xTraining = xShuffled[:numTraining]
    yTraining = yShuffled[:numTraining]
    xVal = xShuffled[numTraining:3800]
    yVal = yShuffled[numTraining:3800]
    xTest = xShuffled[3800:]
    yTest = yShuffled[3800:]

    return xTraining, yTraining, xVal, yVal, xTest, yTest

#Carga para entrenar las muestras distintas a lo normal,
#y para validar las normales.
def loadDataTrainOdd():
    data = np.genfromtxt('abaloneProcessed.txt', delimiter=',')

    x = data[:,:-1]
    y = data[:,-1]

    xTraining = np.empty((0, x.shape[1]))
    yTraining = np.array([])
    xVal = np.empty((0, x.shape[1]))
    yVal = np.array([])
    xTest = np.empty((0, x.shape[1]))
    yTest = np.array([])

    for i in range(x.shape[0]):
        #Valores extraidos a partir de la desviación
        #típcia y la media de y
        if y[i] < 5 or y[i] > 15:
            xTraining = np.vstack((xTraining, x[i]))
```

```
            yTraining = np.append(yTraining, y[i])
        elif random.randint(0,1) == 1:
            xVal = np.vstack((xVal, x[i]))
            yVal = np.append(yVal, y[i])
        else:
            xTest = np.vstack((xTest, x[i]))
            yTest = np.append(yTest, y[i])
    return xTraining, yTraining, xVal, yVal, xTest, yTest

#Carga para entrenar las muestras normales,
#y para validar las raras.
def loadDataTrainNormal():
    data = np.genfromtxt('abaloneProcessed.txt', delimiter=',')

    x = data[:,:-1]
    y = data[:,-1]

    print(np.mean(y), np.std(y))

    xTraining = np.empty((0, x.shape[1]))
    yTraining = np.array([])
    xVal = np.empty((0, x.shape[1]))
    yVal = np.array([])
    xTest = np.empty((0, x.shape[1]))
    yTest = np.array([])

    for i in range(x.shape[0]):

        if y[i] >= 5 and y[i] <= 15:
            xTraining = np.vstack((xTraining, x[i]))
            yTraining = np.append(yTraining, y[i])
        elif random.randint(0,1) == 1:
            xVal = np.vstack((xVal, x[i]))
            yVal = np.append(yVal, y[i])
        else:
            xTest = np.vstack((xTest, x[i]))
            yTest = np.append(yTest, y[i])
    return xTraining, yTraining, xVal, yVal, xTest, yTest

#Muestra en un gráfico la distribución del número de ejemplos
#respecto al número de anillos.
def showDistribution():
    data = np.genfromtxt('abaloneProcessed.txt', delimiter=',')
    y = data[:,-1]

    distinctY = np.unique(y)
    count = np.zeros_like(distinctY)
    k = 0
    for i in distinctY:
        count[k] = np.sum(y == i)
        k = k + 1
    print(count)
    plt.figure()
    plt.title("Distribution of examples")
    plt.ylabel("Number of examples")
    plt.xlabel("Number of rings")
    plt.bar(distinctY, count, color="cornflowerblue")
    plt.savefig("graph/distribution")

showDistribution()
```

```
#Muestra la correlación de un atributo con
#el número de anillos
def showCorrelation(attribute):
    data = np.genfromtxt('abaloneProcessed.txt', delimiter=',')
    y = data[:,-1]
    x = data[:,:-1]

    plt.figure()
    plt.plot(x[:,attribute], y, "x", c="tomato")
    plt.ylabel("Number of Rings")
    plt.title("Correlation with shell weight")
    plt.xlabel("Shell Weight (after being dried) (g)")
    plt.savefig("graph/ShellWeight")

showCorrelation(7)

#Asume que esta hecho con -1 Male, 0 Infant, 1 Female
#Muestra cuantos ejemplos hay para cada sexo
def showExamplesEachSex():
    data = np.genfromtxt('abaloneProcessed.txt', delimiter=',')
    y = data[:,-1]
    x = data[:,:-1]

    sex = np.array([-1, 0, 1])
    number = np.zeros_like(sex)
    number[0] = np.sum(x[:,0] == -1)
    number[1] = np.sum(x[:,0] == 0)
    number[2] = np.sum(x[:,0] == 1)
    plt.figure()
    plt.title("Distribution for sex")
    plt.ylabel("Number of examples")
    plt.xlabel("Sex (-1 = Male; 0 = Infant; 1 = Female)")
    plt.bar(sex, number, color="cornflowerblue")
    plt.savefig("graph/distributionSex")

showExamplesEachSex()
```

SVM

```
#Porcentaje de correctos para la SVM, teniendo en cuenta
#que consideramos correctos los que predigan +- rango
#del valor correcto
def correctPercentage(svm, x, y, rango):
    array = svm.predict(x)
    cmp = (np.abs(array - y) <= rango)
    return np.sum(cmp)/len(y)*100

#Cantidad de error, hace que los que se alejen demasiado
#lo aumenten exponencialmente más.
def errorAmmountSVM(svm, x, y):
    array = svm.predict(x)
    return np.sum(np.abs(array - y) ** 2)

#Pone ejemplos, predicciones hechas por la svm y
#el valor real.
def printExamplesSVM(svm, x, y):
    array = svm.predict(x)
    k = 0
    for i in array:
        print("Valores de x: ", x[k])
        print("El valor predicho es: %i" % i)
```

```
        print("El valor real es: %.0f" % y[k])
        k = k+1

#Implementación de una SVM, dibujando variós gráficos, incluida
#uno en 3d sobre la efectividad de C y sigma
def SVM(rango):
    xTraining,yTraining,xVal,yVal,xTest,yTest = loadDataShuffled(numTraining=
3000)
    bestvalue = -1
    nums = np.array([0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30])
    k = 0
    bestValuesMatrix = np.array([])
    bestValuesMatrixT = np.array([])

    #Prueba para todos los sigmas con todos los C
    for sigma in nums:
        plt.figure()
        arrayPer = np.array([])
        arrayPerTraining = np.array([])
        bestValuesArray = np.array([])

        plt.xlabel("C")
        plt.ylabel("Correct percentage (error margin = %i)" % rango)
        plt.title("Sigma = %.2f" % sigma)
        for c in nums:
            svm = SVC(kernel='rbf', C=c ,gamma=1 / (2 * sigma**2))
            svm.fit(xTraining,yTraining)
            per = correctPercentage(svm,xVal,yVal, rango)
            perEnt = correctPercentage(svm,xTraining,yTraining, rango)
            arrayPer = np.append(arrayPer, per)
            arrayPerTraining = np.append(arrayPerTraining, perEnt)
            if per >= bestvalue:
                bestc = c
                bestsigma = sigma
                bestvalue = per
                bestsvm = svm
            print("%.2f para C = %.2f y sigma = %.2f" % (per, c, sigma))
            print("#%.2f para C = %.2f y sigma = %.2f (entrenamiento)" % (per
Ent,
                                                                           c,
sigma))
            print("errorAmmount", errorAmmountSVM(svm, xVal, yVal))

        bestValuesMatrix = np.append(bestValuesMatrix, arrayPer)
        bestValuesMatrixT = np.append(bestValuesMatrixT, arrayPerTraining)

        #Dibuja la figura en 2D, comparando cada sigma con cada C
        plt.plot(nums, arrayPer, "-", color="orange" ,label="Validation")
        plt.plot(nums, arrayPerTraining, "-", color="blue" ,label="Training")
        plt.legend()
        plt.savefig("svm/sigma%i.png" % k)
        k = k + 1

    #Dibuja en 3D:
    #Los ejemplos de Validación
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    sigma2, c2 = np.meshgrid(nums, nums)
    bestValuesMatrix = np.reshape(bestValuesMatrix, c2.shape)
    ax.plot_surface(c2, sigma2,
```

```
                    bestValuesMatrix, cmap=cm.coolwarm,
                          linewidth=0, antialiased=False)

    ax.set_xlabel("Sigma")
    ax.set_ylabel("C")
    ax.set_zlabel("Correct percentage")
    ax.set_title("Effects on validation examples")

    #Los ejemplos de entrenamiento
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    bestValuesMatrixT = np.reshape(bestValuesMatrixT, c2.shape)
    ax.plot_surface(c2, sigma2,
                    bestValuesMatrixT, cmap=cm.coolwarm,
                          linewidth=0, antialiased=False)

    ax.set_xlabel("Sigma")
    ax.set_ylabel("C")
    ax.set_zlabel("Correct percentage")
    ax.set_title("Effects on training examples")

    print("Mejor porcentaje de acierto sobre los ejemplos de validación: %.2f
" % bestvalue)

    svm = SVC(kernel='rbf', C=30 ,gamma=1 / (2 * 0.01**2))
    svm.fit(xTraining,yTraining)



SVM(0)

#Otra implementación igual que la anterior, sin generar gráficos
#Incluye opción para pasar los ejemplos por parámetro(data),
#Probar con diferentes números de componentes(componentes y Trycomponentes de
be
#ser True), y para probar con distintos tamaños de grupo con groupsize
def SVM2(rango, data=None, TryComponents=False,componentes = 8, groupsize = 1
):
    if data is None:
        xTraining,yTraining,xVal,yVal,xTest,yTest = loadDataShuffled(TryCompo
nents,
                                                                     componen
tes,
                                                                     TryCompo
nents,
                                                                     groupsiz
e)
    else:
        xTraining,yTraining,xVal,yVal,xTest,yTest = data

    bestvalue = -1
    nums = np.array([0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30])

    for c in nums:
        for sigma in nums:
            svm = SVC(kernel='rbf', C=c ,gamma=1 / (2 * sigma**2), max_iter=-
1)
            svm.fit(xTraining,yTraining)
            per = correctPercentage(svm,xVal,yVal, rango)
            if per >= bestvalue:
```

```
                bestc = c
                bestsigma = sigma
                bestvalue = per
                bestsvm = svm

    #Imprime 10 ejemplos
    #printExamplesSVM(bestsvm, xTest[0:10], yTest[0:10])

    #print("Mejor porcentaje de acierto sobre los ejemplos de validación: %.2
f" % bestvalue)

    return bestvalue

#Busca el mejor resultado para el kernel linear y gaussiano y lo devuelve
def linearVSGaussian(rango, numT):
    xTraining, yTraining, xVal, yVal, xTest, yTest = loadDataShuffled(numTrai
ning=numT)
    bestvalue = -1
    nums = np.array([0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30])
    bestvalueLinear = -1
    for c in nums:
        for sigma in nums:
            svm = SVC(kernel='rbf', C=c ,gamma=1 / (2 * sigma**2), max_iter=-
1)
            svm.fit(xTraining,yTraining)
            per = correctPercentage(svm,xVal,yVal, rango)
            if per >= bestvalue:
                bestc = c
                bestsigma = sigma
                bestvalue = per
        svmLinear = SVC(kernel='linear', C=c)
        svmLinear.fit(xTraining, yTraining)
        perLinear = correctPercentage(svmLinear, xVal, yVal, rango)
        if perLinear >= bestvalueLinear:
            bestcLinear = c
            bestvalueLinear = perLinear


    #print("Mejor porcentaje de acierto sobre los ejemplos de validación: %.2
f" % bestvalue)
    #print("Mejor porcentaje de acierto sobre los ejemplos de validación (Lin
ear):
    #%.2f" % bestvalueLinear)

    return bestvalue, bestvalueLinear

#Ejecuta varias veces linearVSGaussian() para hacer gráficos, comparando
#Los resultados de utilizar cada kernel
def linearVSGaussianGraphic():
    arrayGaussian = np.array([])
    arrayLinear = np.array([])
    arrayI = np.array([])
    for i in range(10):
        resGaussian, resLinear = linearVSGaussian(2, 100)
        arrayGaussian = np.append(arrayGaussian, resGaussian)
        arrayLinear = np.append(arrayLinear, resLinear)
        arrayI = np.append(arrayI, i * 2)

    plt.figure()
    plt.title("Linear vs Gaussian kernel")
```

```
    plt.bar(arrayI, arrayGaussian, color = "sandybrown", label="Gaussian")
    plt.bar(arrayI + 1, arrayLinear, color = "seagreen", label="Linear")
    plt.ylabel("Correct percentage; range = 2")
    plt.legend()
    plt.xticks()
    plt.ylim((70,80))
    plt.savefig("svm/LinearGaussian2")

linearVSGaussianGraphic()

#Compara la efectividad para cada numero de componentes
#dibujando un gráfico sobre ello
def aciertosVSnComponentes():
    plt.figure()
    bestValues = np.array([])
    component = np.array([])
    for i in range(1, 9):
        bestValues = np.append(bestValues, SVM2(0, TryComponents=True, compon
entes = i))
        component = np.append(component, i)
    plt.title("Correct percentage vs number of components")
    plt.ylabel("Correct Percentage")
    plt.xlabel("Number of components")
    plt.ylim((np.min(bestValues) - 2, np.max(bestValues) + 2))
    plt.bar(component, bestValues, color="cornflowerblue")
    plt.savefig("svm/AciertosVSComponentes3.png")

aciertosVSnComponentes()

SVM2(3.)

SVM(2.)
```

Neural network

```
def sigmoide(x):
    return 1./(1. + np.exp(-1. * x))

#Genera una theta inicial con valores aleatorios
def pesosAleatorios(Lin, Lout):
    ini = 0.12
    theta = np.random.random((Lout,Lin+1))*(2*ini)-ini
    return theta

def h0(x,theta1,theta2):
    z2 = np.matmul(x, theta1.T)
    a2 = sigmoide(z2)
    a2 = np.hstack((np.ones((a2.shape[0],1)), a2))

    #Capa de salida
    z3 = np.matmul(theta2, a2.T)
    a3 = sigmoide(z3)

    return a3, a2

#Solo funciona para 1 capa oculta
def backdrop(params_rn, num_entradas, num_ocultas, num_etiquetas, x, y, mini,
reg = 0 ):
    #Desempaqueta los parámetros
    theta1 = np.reshape(params_rn[: num_ocultas * (num_entradas + 1) ],
                        (num_ocultas , (num_entradas + 1)))
```

```
    theta2 = np.reshape(params_rn[num_ocultas * (num_entradas + 1) :],
                        (num_etiquetas , (num_ocultas + 1)))
    xOnes = np.hstack((np.ones((x.shape[0],1)), x))

    #Genera la matriz de etiquetas.
    y2 = np.zeros((len(y), num_etiquetas))
    #Pone los 1 en la matriz de 0
    for i in range(len(y)):
        y2[i,int(y[i]) - mini] = 1

    #FORWARD PROPAGATION-----------------------------------------------------
    a3, a2 = h0(xOnes, theta1, theta2)
    a = - y2 * np.log(a3).T
    b = (1 - y2) * np.log(1 - a3).T
    c = np.sum(1. / (xOnes.shape[0]) * (a - b))
    coste = c + (reg/(2 * xOnes.shape[0]))*(np.sum(theta1**2) + np.sum(theta2
**2))

    #BACKPROPAGATION---------------------------------------------------------
    delta3 = a3 - y2.T
    delta2 = np.matmul(theta2.T, delta3) * (a2.T * (1. - a2.T))
    #Copia theta para evitar dañar la original para vectorizar que
    #la primera fila no se modifica al regularizar
    theta1c = np.copy(theta1)
    theta2c = np.copy(theta2)
    theta2c[:,0] = theta2c[:,0] * 0
    theta1c[:,0] = theta1c[:,0] * 0

    #Calcula las theta resultantes, incluyendo la regularización
    triangulo2 = (np.matmul(delta3, a2) / xOnes.shape[0])
    triangulo2 = triangulo2 + (reg/xOnes.shape[0]) * theta2c
    triangulo1 = (np.matmul(delta2[1:], xOnes) / xOnes.shape[0])
    triangulo1 = triangulo1 + (reg/xOnes.shape[0]) * theta1c

    return coste, np.concatenate((triangulo1.ravel(),triangulo2.ravel()))

def coste(xOnes, y2, theta1, theta2, reg):
    a3, a2 = h0(xOnes, theta1, theta2)
    a = - y2 * np.log(a3).T
    b = (1 - y2) * np.log(1 - a3).T
    c = np.sum(1. / (xOnes.shape[0]) * (a - b))
    coste = c + (reg/(2 * xOnes.shape[0]))*(np.sum(theta1**2) + np.sum(theta2
**2))
    return coste

#printExamplesSVM pero para Redes neuronales
def checkNeural(theta_opt1, theta_opt2, x, y, maxi, mini):
    xOnes = np.hstack((np.ones((x.shape[0],1)), x))
    for i in range(xOnes.shape[0]):
        salida = h0(xOnes[i,np.newaxis], theta_opt1, theta_opt2)
        salida = salida[0].ravel()
        print("Valores de x: ", xOnes[i])
        print("El valor predicho es: %i" % (salida.argmax() + mini % maxi))
        print("El valor real es: %i" % int(y[i]))

#correctPercentage pero para redes neuronales
def correctPercentageNN(theta_opt1, theta_opt2, x, y, maxi, mini, rango):
    xOnes = np.hstack((np.ones((x.shape[0],1)), x))
    array = h0(xOnes, theta_opt1, theta_opt2)[0]
```

```
    result = (np.argmax(array, axis=0) + mini) % maxi

    cmp = np.abs(y - (result)) <= rango
    return np.sum(cmp)/len(y)*100

#errorAmmountSVM pero para redes neuronales
def errorAmmountNN(theta_opt1, theta_opt2, x, y, maxi, mini):
    xOnes = np.hstack((np.ones((x.shape[0],1)), x))
    array = h0(xOnes, theta_opt1, theta_opt2)[0]

    result = (np.argmax(array, axis=0) + mini) % maxi
    return np.sum(np.abs(result - y) ** 2)

#Entrena una red neuronal para un valor discreto de regularización
def neuralNetwork(reg, num_ocultas, rango, data = None):

    if data is None:
        xTraining, yTraining, xVal, yVal, xTest, yTest = loadDataShuffled()
    else:
        xTraining, yTraining, xVal, yVal, xTest, yTest = data
    num_entradas = xTraining.shape[1]

    #Utiliza el mínimo y máximo para crear el número de etiquetas
    maxi = int(np.max([np.max(yTraining), np.max(yVal), np.max(yTest)]))
    mini = int(np.min([np.min(yTraining), np.min(yVal), np.min(yTest)]))
    num_etiquetas = maxi - mini + 1

    t1 = pesosAleatorios(num_entradas,num_ocultas)
    t2 = pesosAleatorios(num_ocultas, num_etiquetas)
    params_t = np.vstack((np.reshape(t1, (t1.shape[0]*t1.shape[1],1)),
                          np.reshape(t2, (t2.shape[0]*t2.shape[1],1))))

    #Entrena la red neuronal
    result = opt.minimize(fun=backdrop, x0=params_t,
                          args=(num_entradas, num_ocultas,
                                num_etiquetas, xTraining, yTraining, mini, re
g),
                          method='TNC', jac=True)

    #Desempaqueta la salida del entrenamiento
    salida1 = np.reshape(result.x[: num_ocultas * (num_entradas + 1) ],
                         (num_ocultas , (num_entradas + 1)))
    salida2 = np.reshape(result.x[num_ocultas * (num_entradas + 1) :],
                         (num_etiquetas , (num_ocultas + 1)))

    #Imprime 10 ejemplos de test
    #checkNeural(salida1, salida2, xTest[0:10], yTest[0:10], maxi, mini)

    return correctPercentageNN(salida1, salida2, xVal, yVal, maxi, mini,0), r
esult.fun



#Busca el mejor valor probando con distintas lambdas.
#Puedes decidir si se dibujan gráficas o no con plot
def neuralNetworkDecide(maxiter, num_ocultas, rango, data=None, plot=True):

    if data is None:
        xTraining, yTraining, xVal, yVal, xTest, yTest =  loadDataShuffled()
    else:
        xTraining, yTraining, xVal, yVal, xTest, yTest = data
```

```
    num_entradas = xTraining.shape[1]

    #Utiliza el mínimo y máximo para crear el número de etiquetas,
    #Se utiliza el minimo y máximo entre todos los ejemplos, aunque
    #no aparezcan en los de entranamineto
    maxi = int(np.max([np.max(yTraining), np.max(yVal), np.max(yTest)]))
    mini = int(np.min([np.min(yTraining), np.min(yVal), np.min(yTest)]))
    num_etiquetas = maxi - mini + 1


    xValOnes = np.hstack((np.ones((xVal.shape[0],1)), xVal))

    #Genera la matriz de etiquetas.
    yVal2 = np.zeros((len(yVal), num_etiquetas))
    #Pone los 1 en la matriz de 0
    for i in range(len(yVal)):
        yVal2[i,int(yVal[i]) - mini] = 1

    reg = np.array([0.001, 0.003, 0.007, 0.01, 0.03, 0.07,
                    0.1, 0.3, 0.7, 1, 3, 7, 10, 30])

    #Array para almacenar el resultado del coste para cada lambda
    errorTra = np.array([])
    errorVal = np.array([])
    correctPerTra = np.array([])
    correctPerVal = np.array([])
    #Entrena la red neuronal
    t1 = pesosAleatorios(num_entradas,num_ocultas)
    t2 = pesosAleatorios(num_ocultas, num_etiquetas)
    params_t = np.vstack((np.reshape(t1, (t1.shape[0]*t1.shape[1],1)),
                          np.reshape(t2, (t2.shape[0]*t2.shape[1],1))))
    for i in reg:

        result = opt.minimize(fun=backdrop, x0=params_t,
                              args=(num_entradas, num_ocultas,
                                    num_etiquetas, xTraining, yTraining, mini
, i),
                              method='TNC', jac=True, options={'maxiter': max
iter})

    #Desempaqueta la salida del entrenamiento
        salida1 = np.reshape(result.x[: num_ocultas * (num_entradas + 1) ],
                             (num_ocultas , (num_entradas + 1)))
        salida2 = np.reshape(result.x[num_ocultas * (num_entradas + 1) :],
                             (num_etiquetas , (num_ocultas + 1)))
        correctPerVal = np.append(correctPerVal,
                                  correctPercentageNN(salida1, salida2,
                                                      xVal, yVal,
                                                      maxi, mini, rango))
        correctPerTra = np.append(correctPerTra,
                                  correctPercentageNN(salida1, salida2,
                                                      xTraining, yTraining,
                                                      maxi, mini, rango))

        #print(correctPercentageNN(salida1, salida2, xVal, yVal, maxi, mini,
rango))
        cost = coste(xValOnes, yVal2, salida1, salida2, i)
        errorTra = np.append(errorTra, [result.fun])
        errorVal = np.append(errorVal, [cost])
```

```
            #print("%.5f para reg = %.3f" % (cost, i))
        if plot:
            #Gráfico con el error
            plt.figure()
            plt.plot(reg, errorTra, "-", color="blue", label="Training")
            plt.plot(reg, errorVal, "-", color ="orange", label="Validation")
            plt.title("Error for neural network")
            plt.xlabel("Regularization value")
            plt.legend()
            plt.ylabel("Error")
            plt.xscale("log")
            plt.savefig("nn/reg1")

            #Gráfico con el porcentaje de correctos
            plt.figure()
            plt.plot(reg, correctPerTra, "-", color="lightblue", label="Training"
)
            plt.plot(reg, correctPerVal, "-", color ="tomato", label="Validation"
)
            plt.title("Correct Percentage for neural network")
            plt.xlabel("Regularization value")
            plt.legend()
            plt.ylabel("Correct Percentage (range=2)")
            plt.xscale("log")
            plt.savefig("nn/reg2")
        #checkNeural(salida1, salida2, xVal[0:20], yVal[0:20], maxi, mini)
        return np.max(correctPerVal)#salida1, salida2, result.fun, maxi, mini

    #Compara la efectividad para cada numero de nodos en la capa oculta.
    #Muestra dos graficos con los mismos datos.
    def aciertosVSnOcultas():
        bestValues = np.array([])
        minError = np.array([])
        component = np.array([])
        datos = loadDataShuffled()

        for i in range(1,11):
            res = neuralNetwork(0., i, 0, data=datos)
            bestValues = np.append(bestValues, res[0])
            minError = np.append(minError, res[1])

            component = np.append(component, i)

        #Gráfica para los porcentajes
        plt.figure()
        plt.title("Number of hidden nodes effects")
        plt.ylabel("Correct Percentage")
        plt.xlabel("Number of hidden nodes")
        plt.ylim((np.min(bestValues) - 0.2, np.max(bestValues) + 0.2))
        plt.bar(component, bestValues, color="cornflowerblue")
        plt.savefig("nn/AciertosVSnOcultas10.png")

        #Gráfica para los errores
        plt.figure()
        plt.title("Number of hidden nodes effects")
        plt.ylabel("Error value")
        plt.xlabel("Number of hidden nodes")
        plt.ylim((np.min(minError) - 0.2, np.max(minError) + 0.2))
        plt.bar(component, minError, color="yellowgreen")
        plt.savefig("nn/AciertosVSnOcultas20.png")
```

```
aciertosVSnOcultas()

res = neuralNetworkDecide(1000, 15, 2)

neuralNetwork(0.01, 15, 0)
```

Regresión logistica multicase

```
#Función de coste
def J(theta, x, y, landa):
    theta = np.c_[theta]
    aux = sigmoide(np.matmul(x, theta))
    a = np.log(aux) * y
    b = np.log(1 - aux) * (1 - y)
    c = - 1 / y.shape[0] * (a + b)
    d = landa / (2 * y.shape[0]) * np.sum(theta**2)
    return np.sum(c) + d

#Gradiente
def descenso(theta, x, y, landa):
    theta = np.c_[theta]
    a = sigmoide(np.matmul(x,theta)) - y
    b = np.matmul(np.transpose(x),a)
    c = (1/y.shape[0] * b)
    d = landa/y.shape[0] * theta
    d[0,0] = 0
    return np.ravel(c + d)

def oneVsAll(x, y, num_etiquetas, reg, mini):
    #Crea theta
    theta = np.ravel(np.zeros((1,x.shape[1])))

    #Genera la matriz de etiquetas.
    y2 = np.zeros((len(y), num_etiquetas))
    #Pone los 1 en la matriz de 0
    for i in range(len(y)):
        y2[i,int(y[i]) - mini] = 1


    #Crea theta auxiliar para ir guardando los
    #resultados de cada optimizacion
    theta_opt = np.reshape(theta, (len(theta),1))

    #Ahora para cada etiqueta hacemos el descenso
    for i in range(num_etiquetas):
        result = opt.fmin_tnc(func=J , x0=theta, fprime=descenso,
                              args=(x, np.c_[y2[:,i]], reg))
        theta_opt = np.hstack((theta_opt, np.reshape(result[0],
                                                     (len(result[0]),1))))

    #Se devuelve todo menos la primera fila, que se usaba como auxiliar
    #para ir creando la matriz
    return theta_opt[:,1:]

def train(x,y,reg,num_etiquetas, mini):
    #Vector de X con unos para operar con theta
    xOnes = np.hstack((np.ones((x.shape[0],1)), x))

    #Devuelve el vector optimizado
    return oneVsAll(xOnes,y, num_etiquetas,reg, mini)
```

```
#printExamplesSVM o checkNeural para regresión logística
def doExamples(x, y, theta, mini, maxi):

    #Vector de x con unos para operar matricialmente con theta
    xOnes = np.hstack((np.ones((x.shape[0],1)), x))

    for i in range(xOnes.shape[0]):
        salida = sigmoide(np.matmul(xOnes[i], theta))
        print("Valores de x: ", xOnes[i])
        print("El valor predicho es: %i" % ((salida.argmax() + mini) % maxi))
        print("El valor real es: %i" % int(y[i]))

#correctPercentage o correctPercentageNN para regresión logística
def correctPercentageRL(theta, x, y, maxi, mini, rango):
    xOnes = np.hstack((np.ones((x.shape[0],1)), x))

    array = sigmoide(np.matmul(xOnes, theta))
    result = (np.argmax(array, axis=1) + mini) % maxi

    cmp = np.abs(y - (result)) <= rango
    return np.sum(cmp)/len(y)*100

#Entrena para un valor discreto de reg
def regresionLogistica(reg, rango, data=None):

    if data is None:
        xtrain, ytrain, xval, yval, xtest, ytest = loadDataShuffled()
    else:
        xtrain, ytrain, xval, yval, xtest, ytest = data

    #Entrena con los ejemplos dados y lambda
    maxi = int(np.max([np.max(ytrain), np.max(yval), np.max(ytest)]))
    mini = int(np.min([np.min(ytrain), np.min(yval), np.min(ytest)]))
    num_etiquetas = maxi - mini + 1
    theta = train(xtrain, ytrain, reg, num_etiquetas, mini)

    trainPer = correctPercentageRL(theta, xtrain, ytrain, maxi, mini, rango)
    return correctPercentageRL(theta, xval, yval, maxi, mini, rango), trainPe
r, theta


#Busca el mejor resultado probando con varios valores de lambda
def chooseLanda(numTrain, rango, data=None, plot=True):
    reg = np.array([0.001, 0.003, 0.007, 0.01, 0.03, 0.07,
                    0.1, 0.3, 0.7, 1, 3, 7, 10, 30])
    if data is None:
        data = loadDataShuffled(numTraining=numTrain)

    xtrain, ytrain, xval, yval, xtest, ytest = data
    maxi = int(np.max([np.max(ytrain), np.max(yval), np.max(ytest)]))
    mini = int(np.min([np.min(ytrain), np.min(yval), np.min(ytest)]))

    values = np.array([])
    valuesTrain = np.array([])
    bestPer = -1
    for i in reg:
        RL = regresionLogistica(i,rango,data)
        values = np.append(values, RL[0])
        valuesTrain = np.append(valuesTrain, RL[1])
        if RL[0] > bestPer:
            bestPer = RL[0]
```

```
            bestPerT = RL[1]
            bestTheta = RL[2]
            bestLan = i

    if plot:
        plt.figure()
        plt.xscale("log")
        plt.plot(reg, values, "-", c="khaki", label="Validation")
        plt.plot(reg, valuesTrain, "-", c="turquoise", label="Training")
        plt.legend()
        plt.xlabel("Regularization value")
        plt.ylabel("Correct percentage (range=%i)" % rango)
        plt.title("Number of examples: %i" % numTrain)
        print("Best lambda = %.3f" % bestLan)
        plt.savefig("rl/correctPercentage%i" % numTrain)

    #doExamples(xtest[0:10],ytest[0:10],bestTheta, mini, maxi)

    return bestPer, bestPerT


#Preprocesa los ejemplos para hacer polinomios de grado numpoly
#y hace la regresión discreta con los parámetros dados
def regresionLogisticaPoly(reg, numpoly,data,rango):
    xtrain, ytrain, xval, yval, xtest, ytest = data
    poly = PolynomialFeatures(numpoly) #Función polinomial
    xtrain = poly.fit_transform(xtrain)
    xval = poly.fit_transform(xval)
    xtest = poly.fit_transform(xtest)
    datos = xtrain, ytrain, xval, yval, xtest, ytest
    RL = regresionLogistica(reg,rango,datos)
    return RL[0], RL[1]

#Busca el mejor tamaño para el polinomio; dibuja la gráfica
def bestNumPoly(rango):
    percentages = np.array([])
    percentagesT = np.array([])
    k = np.array([])

    data = loadDataShuffled()

    #Solo 4, por el gran coste computacional.
    for i in range(1,5):
        bestValues = regresionLogisticaPoly(0.01,i,data,rango)
        percentages = np.append(percentages, bestValues[0])
        percentagesT = np.append(percentagesT, bestValues[1])
        k = np.append(k, i)

    plt.figure()
    plt.xlabel("Polinomy degree")
    plt.ylabel("Correct Percentage (range=%i)" % rango)
    plt.plot(k, percentages, "-", c="darkorchid", label="Validation")
    plt.plot(k, percentagesT, "-", c="greenyellow", label="Training")
    plt.legend()
    plt.savefig("rl/bestnumpoly")


bestNumPoly(0)
```

```
#Busca cual es el mejor número de ejemplos de entrenamiento,
#dibujando gráficos.
def bestNumTraining():
    numTraining = np.array([5,10, 20,50,100,300,
                            600, 1000, 1500, 2000, 2500,
                            3000, 3500])
    percentages = np.array([])
    percentagesT = np.array([])


    for i in numTraining:
        bestValues = chooseLanda(i, 0, plot=False)
        percentages = np.append(percentages, bestValues[0])
        percentagesT = np.append(percentagesT, bestValues[1])

    plt.figure()
    plt.xlabel("Number of training examples")
    plt.ylabel("Correct Percentage")
    plt.plot(numTraining, percentages, "-", c="orchid", label="Validation")
    plt.plot(numTraining, percentagesT, "-", c="darkseagreen", label="Trainin
g")
    plt.legend()
    plt.savefig("rl/NTrainingExamplesVSCorrectPercentage")

bestNumTraining()

chooseLanda(3000, 2)
```

Comparaciones entre los 3 métodos

```
def bestResults(rango, datos):
    #Regresión lineal
    bestPerRL = chooseLanda(3000, rango, data=datos, plot=False)[0]

    #Neural Network
    bestPerNN = neuralNetworkDecide(1000, 15, rango, data=datos, plot=False)

    #SVM
    bestPerSVM = SVM2(rango, data=datos)

    resultsArray = [bestPerRL, bestPerNN, bestPerSVM]

    plt.figure()
    plt.title("Best result obtained")
    plt.ylabel("Correct percentage (range=%i)" % rango)
    plt.ylim((np.min(resultsArray)-0.5, np.max(resultsArray)+0.5))
    plt.bar(["Regresión lineal", "Red neuronal", "Support Vector Machine"],
            [bestPerRL, bestPerNN, bestPerSVM],
            color=["darkorchid", "gold", "dodgerblue"])
    plt.savefig("graph/comparisonrange%i" % rango)

#Prueba con tres márgenes de errores y los mismos datos
#de entrenamiento
def bestResultsRangos():
    datos = loadDataShuffled()
    bestResults(0, datos)
    bestResults(1, datos)
    bestResults(2, datos)

bestResultsRangos()
```

Bibliografía

Apuntes de Aprendizaje Automático

https://es.wikipedia.org/wiki/Haliotis

https://matplotlib.org/

https://stackoverflow.com/