Memoria Proyecto AA

Mario Quiñones Pérez y Guillermo García Patiño Lenza

Curso 2020-2021

Índice

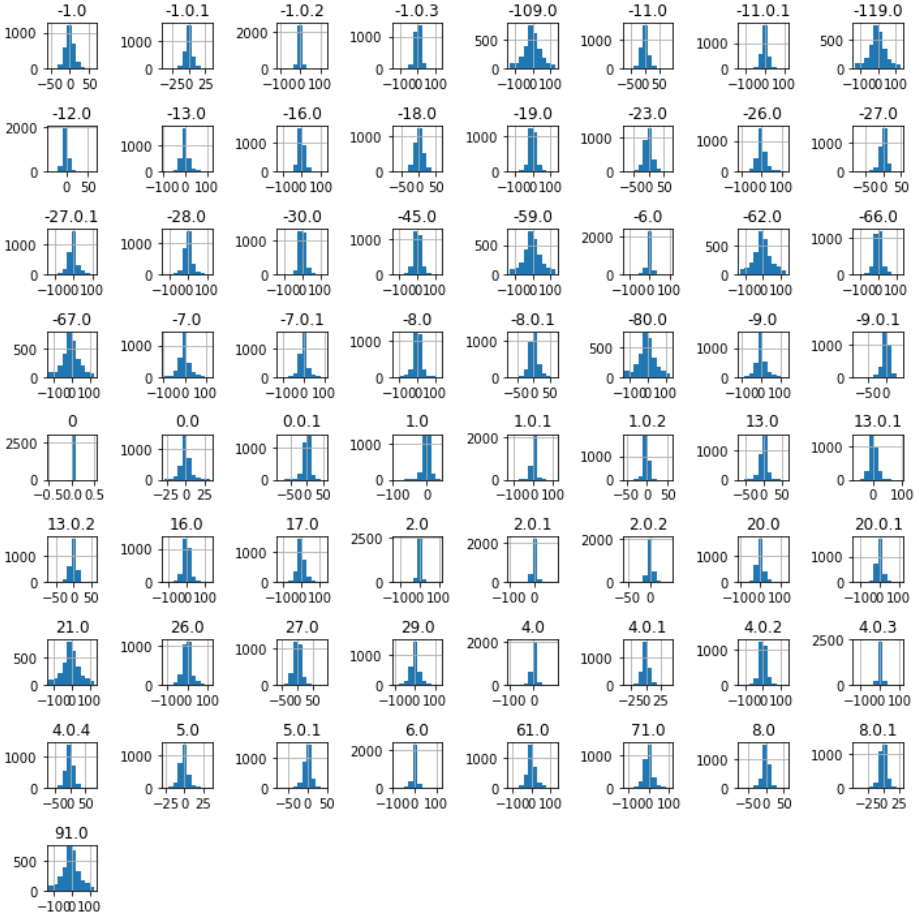
1. Datos

El dataset que se va a emplear en este proyecto ha sido obtenido desde Kaggle (https://www.kaggle.com/kyr7plus/emg-4?select=0.csv). Está formado por datos recopilados por 8 sensores diferentes conectados al brazo de una persona mientras realiza 4 gestos diferentes. Para cada movimiento cada sensor almacena datos en 8 momentos diferentes del movimiento.

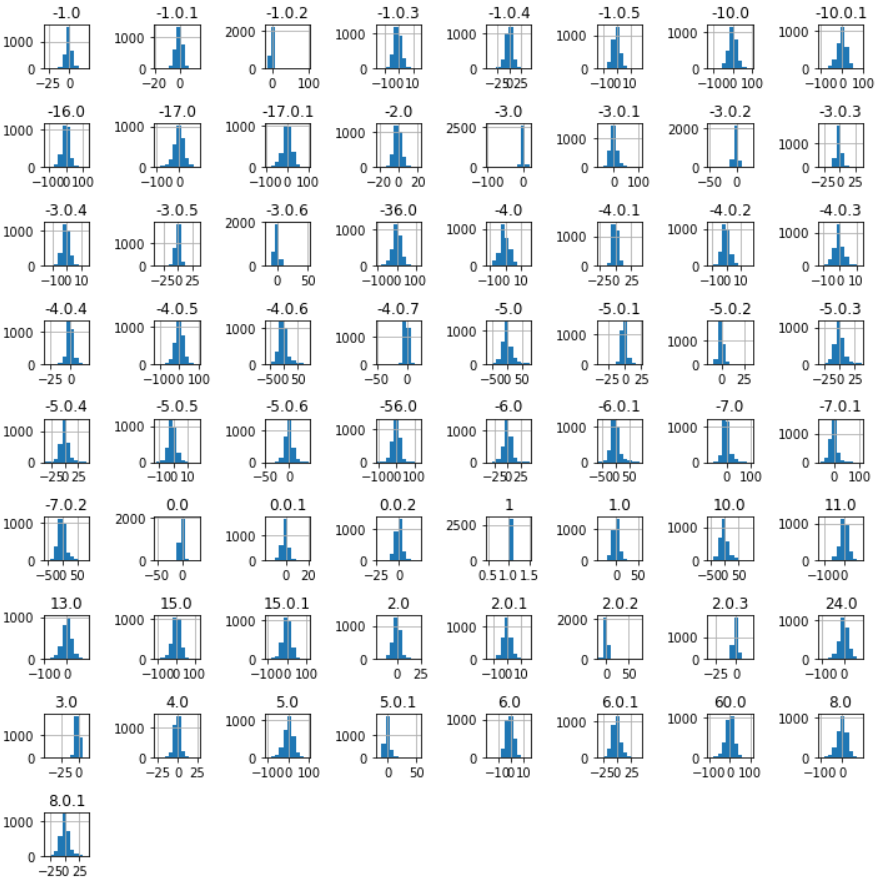
De esta manera, un ejemplo de entrenamiento de este conjunto de datos consiste en un vector de 65 filas, de las cuales 64 son los datos tomados por los 8 sensores en 8 momentos diferentes y 1 es la columna que indica a cuál de los 4 movimientos diferentes corresponde el ejemplo.

Puesto que cada ejemplo de entrenamiento tiene 64 atributos, realizar una representación gráfica de los ejemplos resulta muy complicado, por lo que para tener una visión previa de los datos se ha optado por elaborar un histograma de las variables separando a los ejemplos por la etiqueta a la que corresponden. A continuación se muestran dichos histogramas.

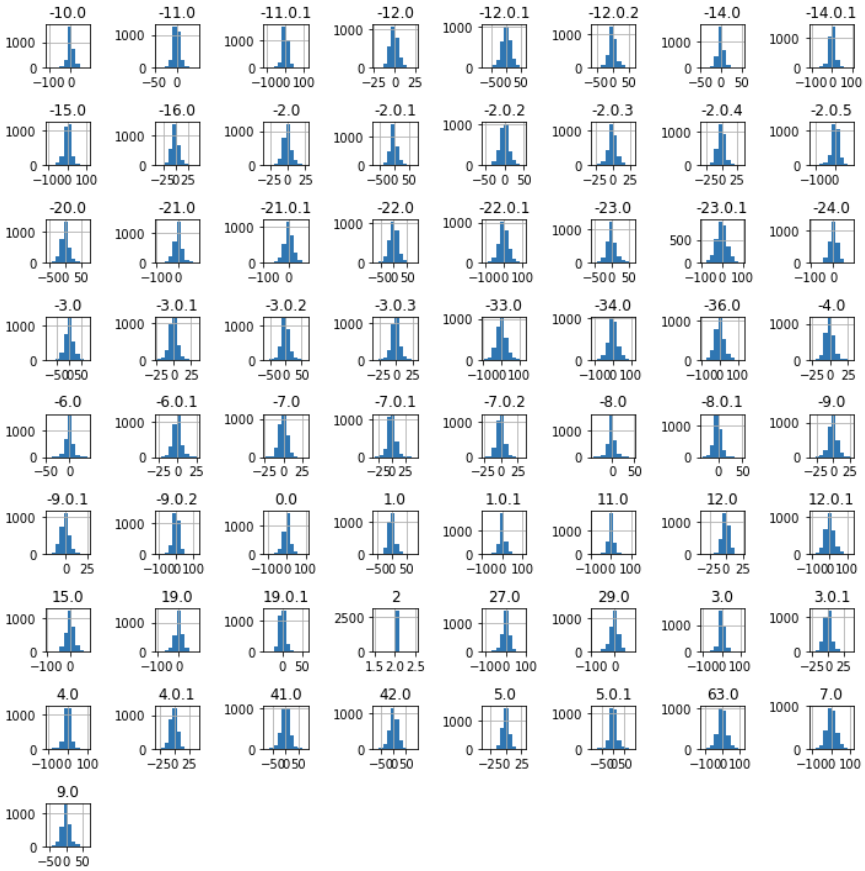
Movimiento 0



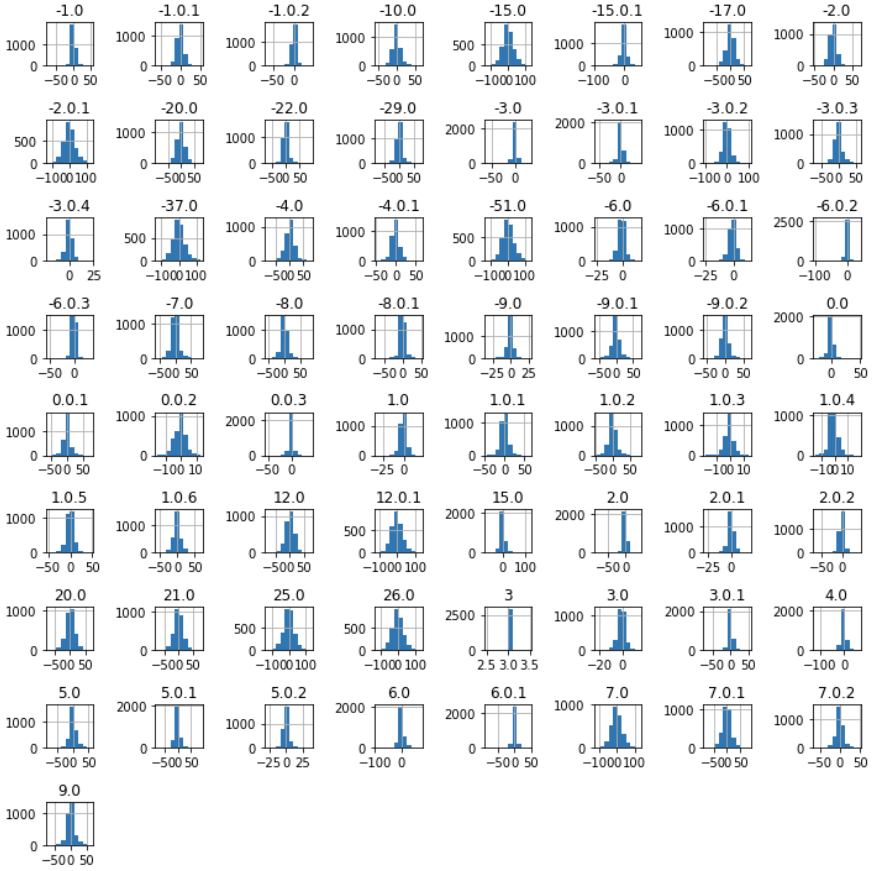
Movimiento 1



Movimiento 2



Movimiento 3



Observando las distribuciones de los atributos en las diferentes clases, no se puede concluir mucho. Lo que sí se detecta es que la mayoría de los atributos siguen una distribución normal en todas las clases, así que no debería de haber ningún problema al normalizarlos antes de procesarlos.

Ya que estas clases están balanceadas en el número de elementos por cada una de las mismas (aproximadamente 3000 por cada clase) utilizaremos la precisión como medida de aciertos de estos datos.

Por otro lado, durante todo el proyecto emplearemos una división del dataset que reparte el 60% de los ejemplos para entrenar, el 20% para realizar pruebas con hiperparámetros (validación) y otro 20% para probar y obtener métricas de los sistemas que obtengamos.

El código para cargar los datos, normalizarlos, y dividir el dataset se encuentra en el fichero '*loader.py*', más concretamente en las funciones normalizar, '*dividirDataSet*' , '*cargarDatos'* y '*carga_Numpy*'. El archivo *'fun_basicas.py'* contiene dos funciones básicas que serán usados por los métodos de aprendizaje, *'sigmoide'* y *'one_hot'.*

2. Técnicas de aprendizaje automático

En este apartado comentaremos los tres tipos de técnicas que hemos usado para este problema y los resultados obtenidos en cada una de ellas.

El problema con el que tratamos es de clasificación multi-clase habiendo cuatro tipos distintos, por lo tanto usaremos los siguientes métodos de clasificación:

- Regresión logística multi-clase
- Redes neuronales
- Máquinas de vector de soporte (SVM)

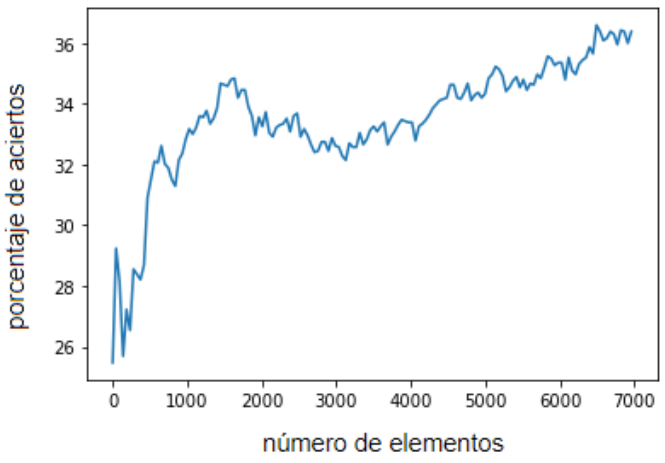
Todos estos modelos de clasificación han sido estudiados y usados a lo largo del curso y por tanto usaremos código que desarrollamos en las prácticas y que añadiremos junto a esta memoria.

a. Regresión logística multi-clase

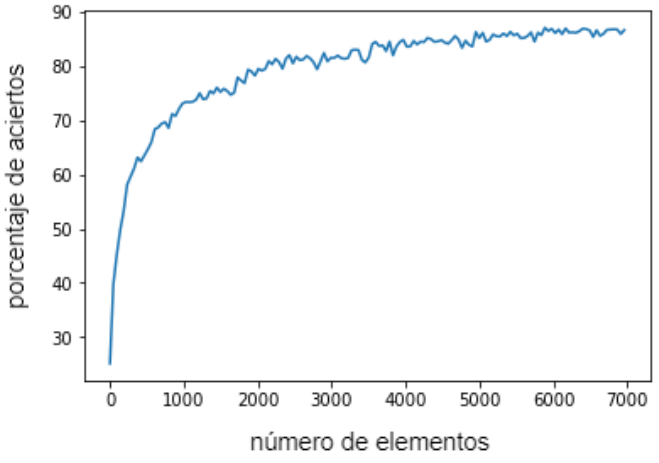
Los métodos de regresión son muy simples, ya que tienen muy pocos hiperparámetros que tener en cuenta, el parámetro de regularización y el número de términos polinómicos a añadir.

En primer lugar, hemos decidido obtener el número óptimo de términos polinómicos. Puesto que nuestros ejemplos de entrenamiento tienen 64 atributos, no hemos podido incluir términos polinómicos mayores a los de grado 2 puesto que producían desbordamiento en el entrenamiento.

Polinomio de grado 1



Polinomio de grado 2

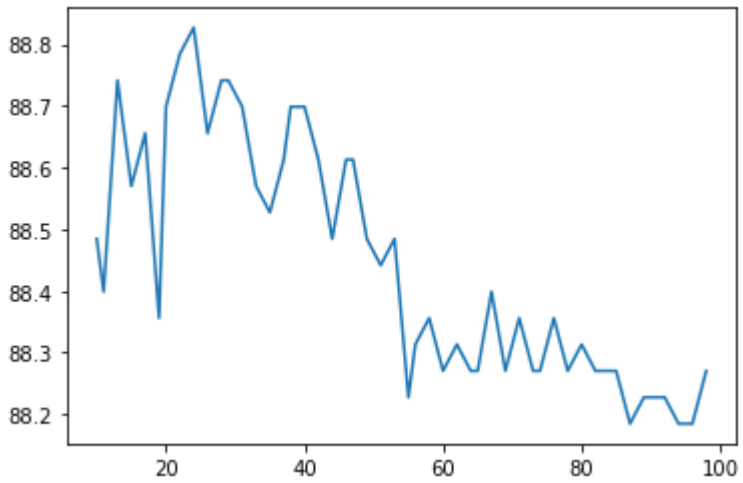


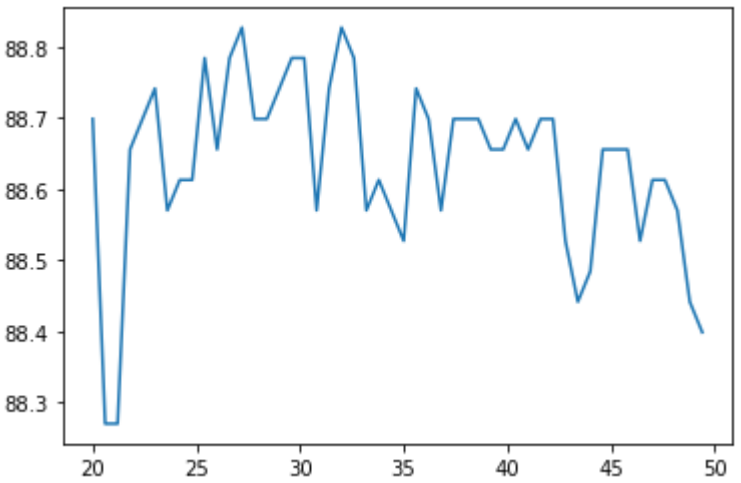
La primera gráfica representa el porcentaje de aciertos al entrenar el modelo con un término polinómico y probarlo sobre el conjunto de validación. El eje X representa el número de elementos del conjunto de entrenamiento usados para entrenar el clasificador y el eje Y el porcentaje de aciertos en el conjunto de validación. En la misma vemos que no se supera el 40% en ninguno de los subconjunto de datos y por tanto clasifica muy mal los elementos.

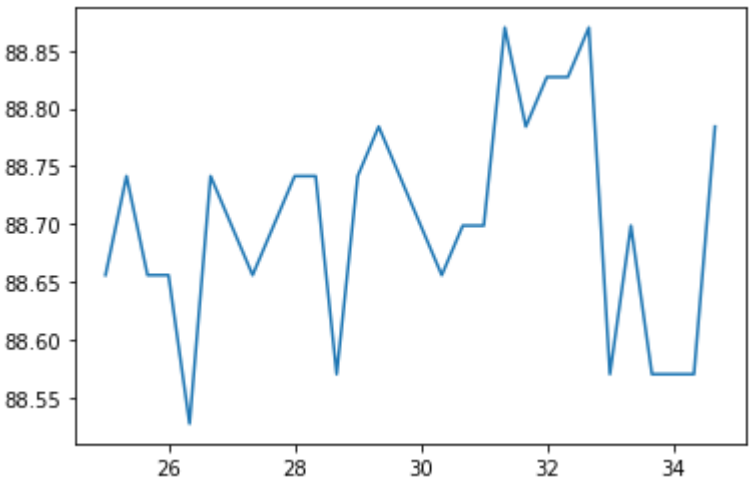
La segunda gráfica representa el porcentaje de aciertos sobre el conjunto de validación tras haber entrenado el modelo con datos a los que se les han añadido términos polinómicos hasta grado 2. Se observa que supera con amplia diferencia el porcentaje de aciertos anterior, ya que llega a alcanzar más de un 80 % de aciertos.

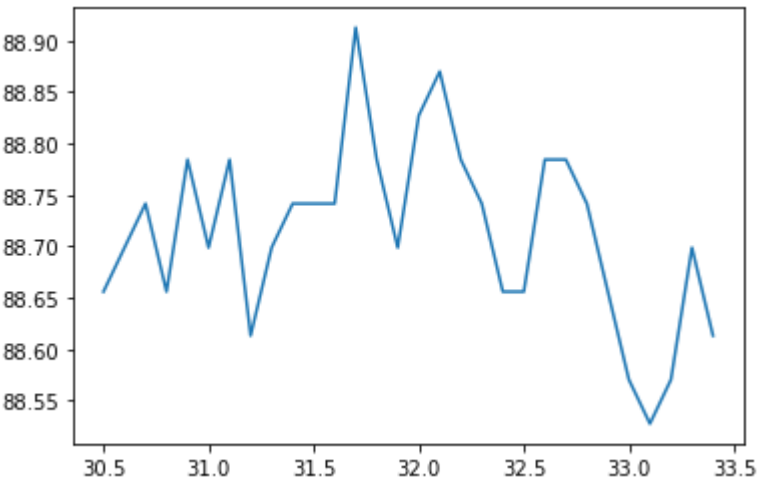
Sabiendo ahora que el término polinómico a usar es el de grado dos ya que proporciona unos mejores resultados, ahora debemos encontrar el parámetro de regularización que mejore el porcentaje de aciertos en el conjunto de validación

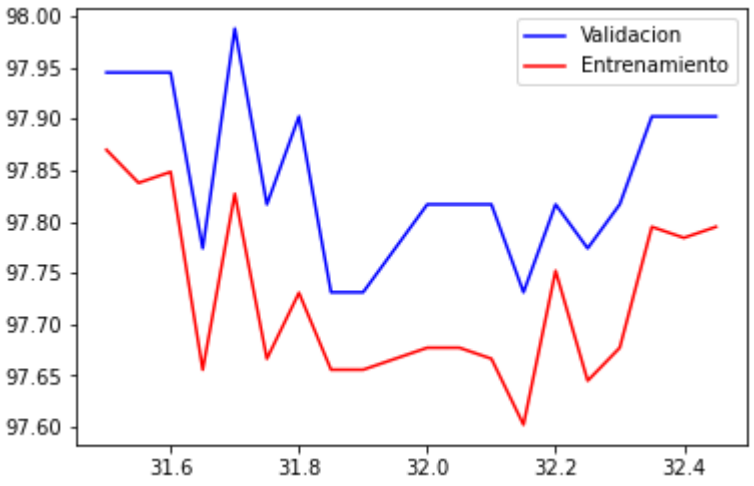
Para encontrar el parámetro de regularización hemos empezado probando con un rango amplio de valores, que progresivamente hemos ido acotando hasta encontrar uno que diera un porcentaje de aciertos suficientemente alto. A continuación se muestran los resultados obtenidos para los diferentes lambdas probados (siendo el eje X los valores de Lambda).

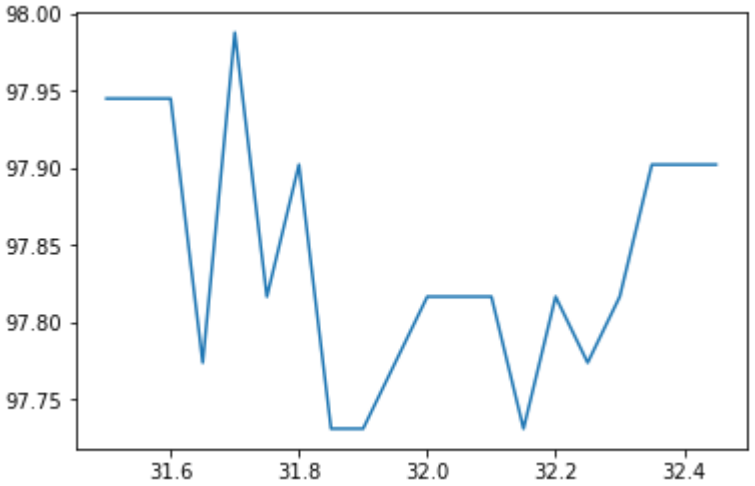






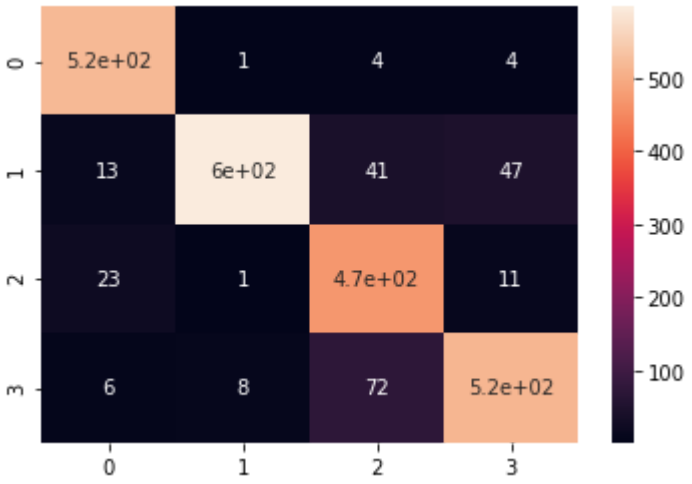






Al observar estas gráficas, hemos concluido que el parámetro de regularización óptimo es 32,35. Ahora que ya hemos determinado los hiperparámetros del sistema, vamos a obtener medidas de precisión y recall sobre cada clase evaluando el clasificador con estos hiperparámetros sobre el conjunto de prueba.

Para ayudarnos a observar el comportamiento del clasificador, hemos obtenido la matriz de confusión de los resultados que produce. En el eje X se indican las clases que ha predicho el clasificador y en el eje Y las clases reales a las que pertenecen los ejemplos.



En esta matriz de confusión se observa que el clasificador acierta en la mayoría de los casos (90,11%), por lo que podemos determinar que los hiperparámetros elegidos son los correctos. Esto se ve reforzado al calcular las métricas de precisión y recall para cada clase.

- Clase 0 :
    - Precisión = 0.92
    - Recall = 0.98

- Clase 1 :
    - Precisión = 0.98
    - Recall = 0.85

- Clase 2 :
    - Precisión = 0.8
    - Recall = 0.93

- Clase 3 :
    - Precisión = 0.89
    - Recall = 0.85

El código encargado de realizar la evaluación de una configuración de los hiperparámetros de la regresión se encuentra en el fichero '*regresión.py*', más concretamente en la función '*evaluar_validacion*' , que devuelve la clase asignada por el clasificador a los elementos del conjunto de validación y la precisión que obtiene con esa configuración.

Además, en el notebook '*pruebas_regresion_Lambda*' se ha declarado una nueva función llamada '*evalua_polinomicos*' que recibe como parámetro una lista de números que se emplearán como coeficientes de regularización, y obtiene una gráfica que representa el porcentaje de aciertos en validación frente al coeficiente de regularización empleado.

b. Redes neuronales

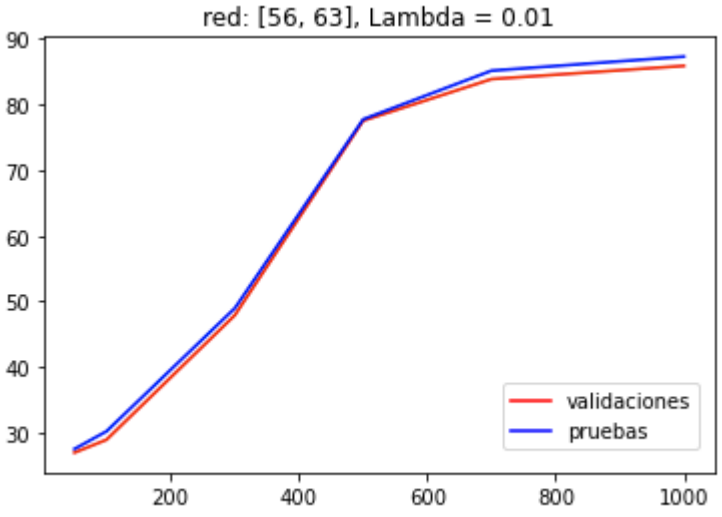
Las redes neuronales son útiles para una gran variedad de problemas por eso las utilizaremos para este en concreto. Uno de los problemas que conllevan estas redes son la gran cantidad de hiperparámetros que tienen (número de capas ocultas, elementos por capa, parámetro de regularización), lo que hace que a veces sea muy difícil encontrar una configuración óptima para los datos.

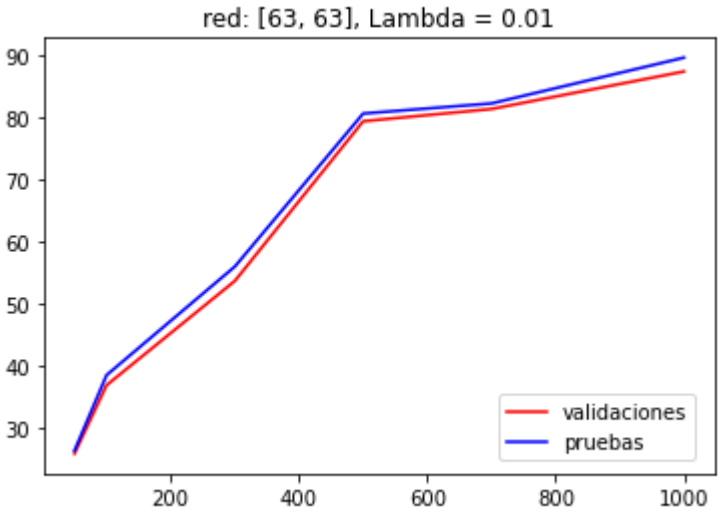
Para estas redes creamos una clase '*red_neuronal*' en el archivo '*creador_redes.py*' en la que incluimos todas las funciones de creación, entrenamiento, validación y prueba para redes con un número de capas ocultas y un número de nodos por capa oculta cualesquiera.

Tras comprobar con la función '*checkNN*' del archivo '*checkNNGradients'* (cambiada ligeramente para que aceptase todo tipo de redes), que para cualquier número de capas ocultas se hacía bien el backpropagation, era el momento de buscar la red óptima para el conjunto de datos.

Todas las pruebas siguientes se realizaron con la función *'pruebas'* de '*creador_redes.py*' un función aparte de la clase *'red_neuronal'* para crear y probar todo tipo de redes con distintos valores de Lambda y número de iteraciones.

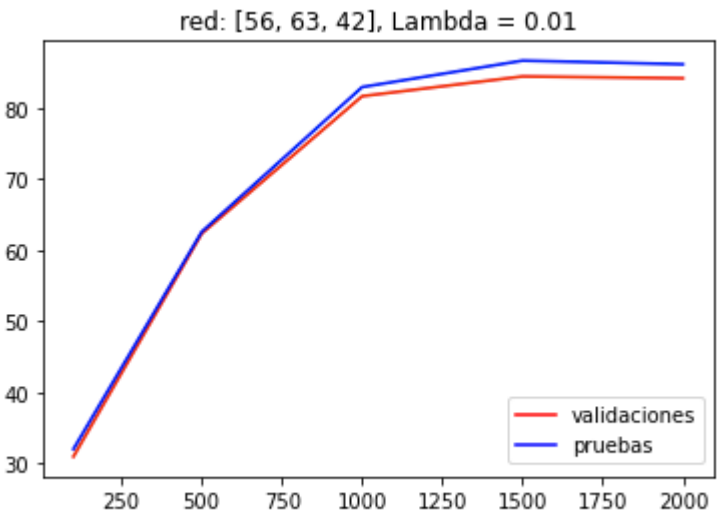
Para encontrar los valores óptimos de las capas ocultas, utilizamos una escala entre 4 (número de nodos en la capa de salida) y 65 (número de nodos en la capa de entrada) de múltiplos de 7 (7,14,28,...). Primero buscamos en todas las combinaciones de dos capas ocultas con todos estos valores como número de nodo por capas.

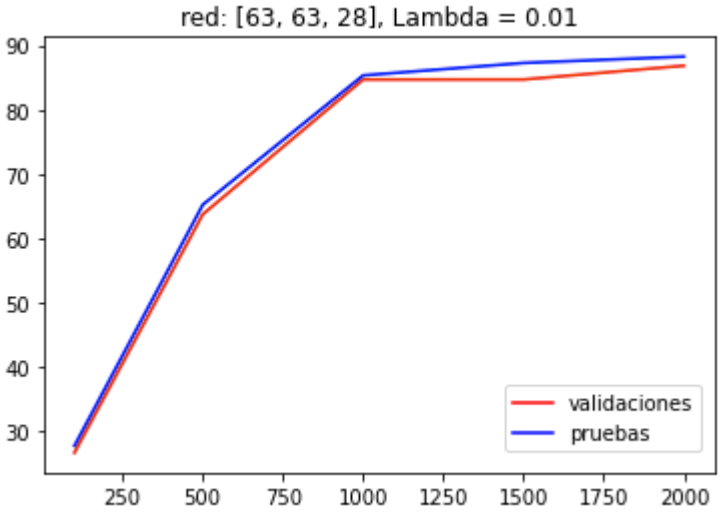




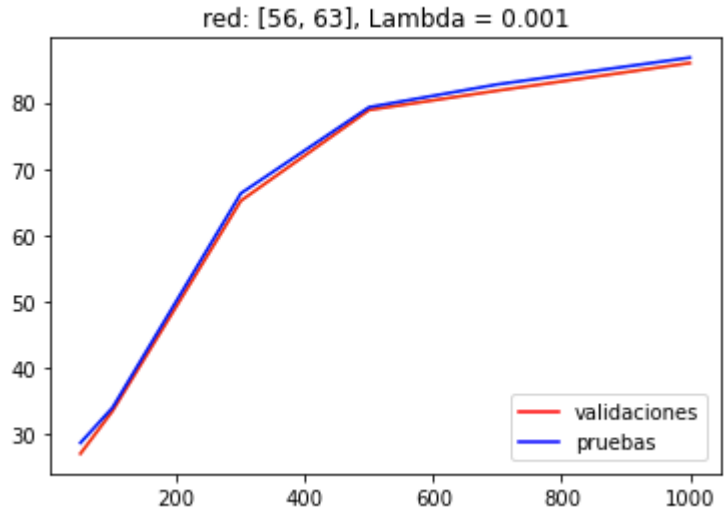
Tras esto podemos ver que las mejores redes de dos capas ocultas fueron dos capas ocultas de 63 nodos ambas y dos capas la primera de 56 nodos y la segunda de 63, en la imagen anterior podemos ver que las dos líneas que representan la precisión con un número de epochs del eje X. La línea roja representa aciertos en el conjunto de validación y la azul en el de prueba.

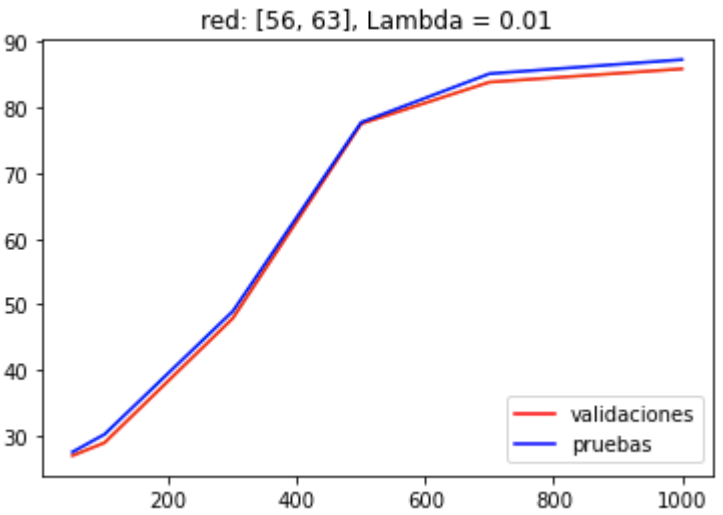
Buscamos todas las combinaciones de tres capas ocultas que empezasen por los elementos de que hemos comprobado anteriormente que funcionaban para dos capas ocultas.

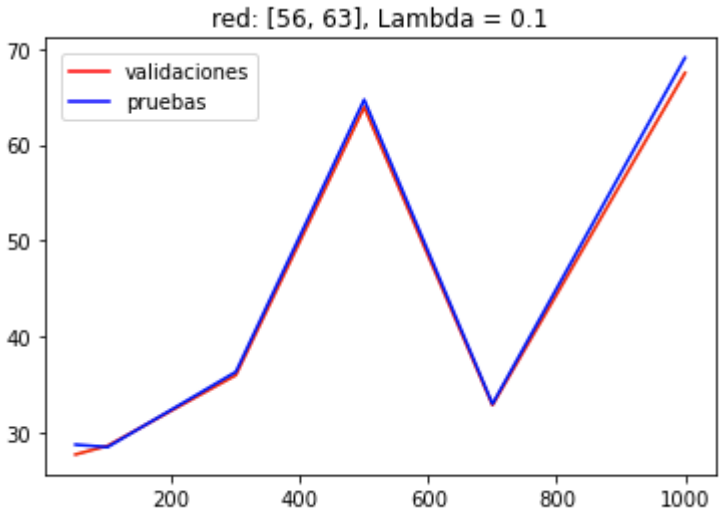




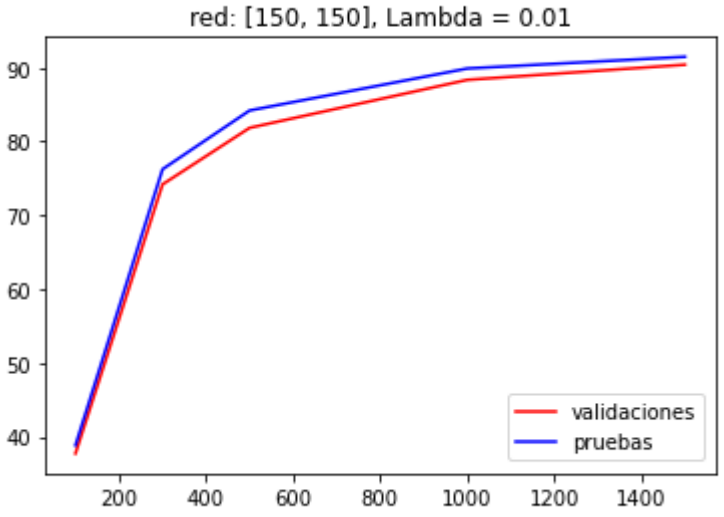
Para calcular el lambda óptimo utilizamos la segunda de las mejores combinaciones de dos capas ocultas (56 la primera y 63 la segunda). Cojemos los dos mejores lambda (0.001 y 0.01) y volvemos a comprobar con el resto de redes de dos capas y llegamos a la conclusión de que el valor 0.01 era el óptimo para el parámetro Lambda.



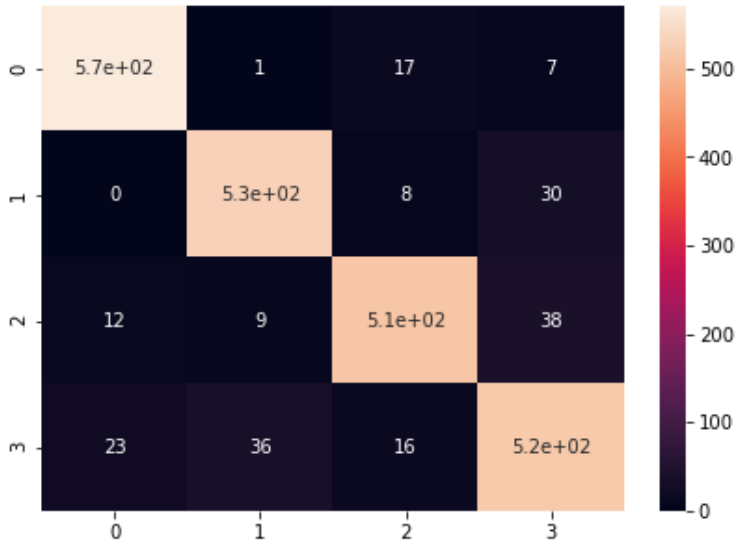




Para valores muy altos de epochs (mas de 1500) no se mejoraba los resultados en las redes de tres capas ocultas los resultados obtenidos en las de dos capas. Aumentando el número de nodos de ambas capas ocultas hasta 150 viendo que no mejoraba significativamente a partir de las de 63 nodos (orden de las milesimas), decidimos que la mejor red sería de 150 nodos en dos capas ocultas lo que llevaba a un acierto de 91% (Aumentando el número de nodos por capa empeoraba los resultados). Tras esto se aumentó el número de capas ocultas, lo que llevó a un empeoramiento ligero y por tanto la red final sería la anteriormente comentada.



Tras encontrar la mejor red se calculó su matriz de confusión, creada con la función *'matriz_confusion'* de *'creador_redes.py'*, con 2000 epochs (punto de convergencia).



Al final se obtiene un alto porcentaje de aciertos (91,57%) y unas métricas de precisión y recall bastante buenas que se exponen a continuación.

- Las clases que más se confunden son la 2 con la 3 porque hay 38 de clase 3 que se clasifican como clase 2 , la 3 con la 1 al haber 36 de clase 3 que se clasifican como clase 1 y la 1 con la 3, pues hay 30 de clase 1 que se clasifican como 3.

- Cálculo de precisión y recall por clase :

    - Clase 0 :
        - Precisión = 0.94
        - Recall = 0.96

    - Clase 1 :
        - Precisión = 0.92
        - Recall = 0.93

    - Clase 2 :
        - Precisión = 0.93
        - Recall = 0.9

    - Clase 3 :
        - Precisión = 0.87
        - Recall = 0.87

Las redes neuronales nos dieron los mejores resultados entre todos los métodos de aprendizaje supervisado usados y por tanto sería la mejor para el aprendizaje de este dataset. Aunque teniendo en cuenta la cantidad de pruebas que se necesitan realizar y el tiempo que estas conllevan en comparación con el resto de métodos, una mejora de un 1% no parece muy útil, pero en el mundo del Big Data esa mejora es siempre bienvenida.

c. Máquinas de vector de soporte (SVM)

En cuanto a las SVM, hemos utilizado, como en la práctica 6 de la asignatura, la clase SVC de scikit learn que permite utilizar una máquina de vector de soporte para problemas de clasificación, y a la que se le tiene que informar que tipo de kernel utilizar (lineal, polinómico, etc.), el parámetro de regularización C y gamma una valor que se utilizará para el cálculo de algunos de los kernels y que viene dado por la función gamma = 1/(2*sigma**2).

Para calcular los valores óptimos de los hiperparámetros utilizamos el conjunto de validación como en los otros modelos de aprendizaje. Empezamos calculando cuál era el tipo de kernel óptimo para este dataset utilizando la función '*evalua_Kernels'* del archivo '*supportvm.py*' que compara todos los kernel que se pueden utilizar en este problema y devuelve el porcentaje de aciertos de cada uno además de decir aquel que tiene un menor fallo. Gracias a esta función vimos que el kernel 'rfb' era el que daba mejores resultados.

```
In [48]: svm.evalua_Kernels(Ex2, Ey, Vx2, Vy, Px2, Py, 60, 10)
         rbf acertados :  90.41095890410958
         poly acertados :  44.56335616438356
         sigmoid acertados :  24.914383561643834
         [90.4109589  44.56335616 24.91438356]
         rbf 0.9118150684931506
Out[48]: ('rbf', 0.9118150684931506)
```

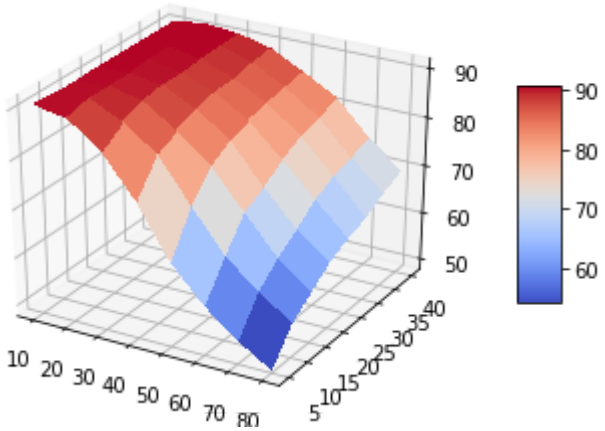
Para calcular los mejores valores para C y sigma utilizamos la función '*evalua_Lambdas'* que utiliza un vector de elementos que se le pase y una serie de datos para entrenar con los distintos valores y el conjunto de entrenamiento la SVC y calcular el porcentaje de aciertos en el conjunto de validación. Tras calcular que combinación de C y sigma fue la mejor en la validación, se calculará el porcentaje de aciertos con el conjunto de prueba con esa configuración y se devolverá.

Para encontrar estos valores óptimos empezamos con unos valores entre el 0.01 y el 30 y vimos que los menores valores eran los que tenían un mejor resultado.

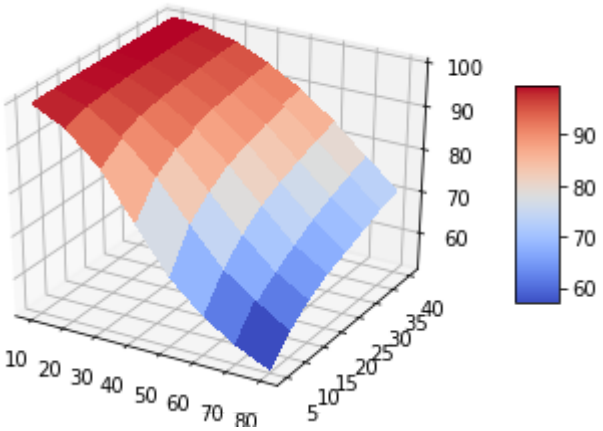
Tras realizar unas cuantas pruebas, determinamos que valores para '*C*' entre 10 y 100 podrían dar buenos resultados, mientras que para '*Sigma'* los mejores valores estaban próximos al 10 o el 15.

Teniendo esto en cuenta, hemos graficado el error en prueba y validación con respecto a los parámetros '*C*' y '*Sigma*' en los rangos mencionados.

Validación



Entrenamiento



Gracias a estas dos gráficas, hemos podido determinar que, para evitar el sobreaprendizaje, es conveniente usar *C* = 30 y *Sigma* = 10, puesto que para valores menores de *C* y mayores de *Sigma*, el error empieza a ser más pequeño y no varía mucho.

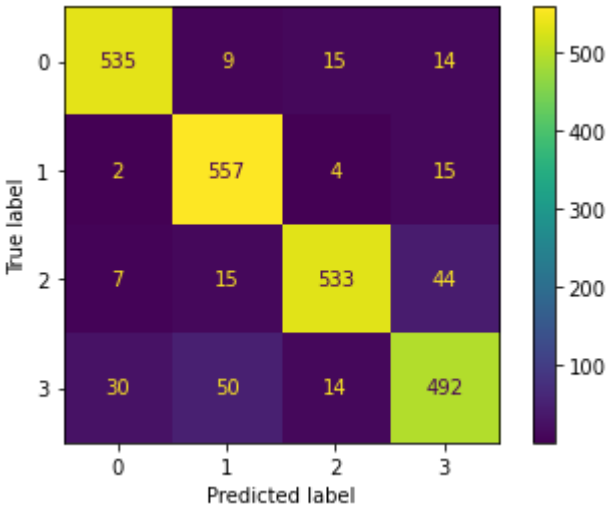
Sabiendo cuales son los valores óptimos y el tipo de kernel que mejor funciona para estos datos, utilizaremos la función '*evaluar_probar*' para calcular el número de aciertos en los datos de entrenamiento y los de prueba. En el primero, como era de esperar, tiene un porcentaje de aciertos de un 98%, mientras que en el segundo ,aunque tiene un mayor porcentaje de fallos, la precisión total supera un 90% lo que da buenos resultados.

```
In [39]: aciertosE2, aciertosP2 = svm.evaluar_probar(Ex2, Ey, Px2, Py, 30, 10)

In [40]: print(aciertosE2)
         print(aciertosP2)

         98.0730802169569
         90.625
```

Finalmente, para entender mejor el comportamiento del clasificador hemos creado la matriz de confusión y a la vista de ésta, hemos sacado las siguientes conclusiones.



- Las clases que más se confunden son la 3 con la 2 porque hay 44 de clase 2 que se clasifican como clase 3 , la 3 con la 1 al haber 50 de clase 3 que se clasifican como clase 1 y la 3 con la 0, pues hay 30 de clase 3 que se clasifican como 0.

- Cálculo de precisión y recall por clase :
    - Clase 0 :
        - Precisión = 0.93
        - Recall = 0.93
    - Clase 1 :
        - Precisión = 0.88
        - Recall = 0.96
    - Clase 2 :
        - Precisión = 0.94
        - Recall = 0.89
    - Clase 3 :
        - Precisión = 0.87
        - Recall = 0.83

A la vista de todo esto, podemos concluir que el clasificador es bastante preciso llegando hasta un 90% de aciertos en el conjunto de prueba.