PRÁCTICA 3:

Programación

Aprendizaje Automático

Ángel Cabeza Martín

1. Problema de Clasificación

1. Comprender el problema a resolver. Identificar los elementos X, Y y f del problema y describirlos en detalle.

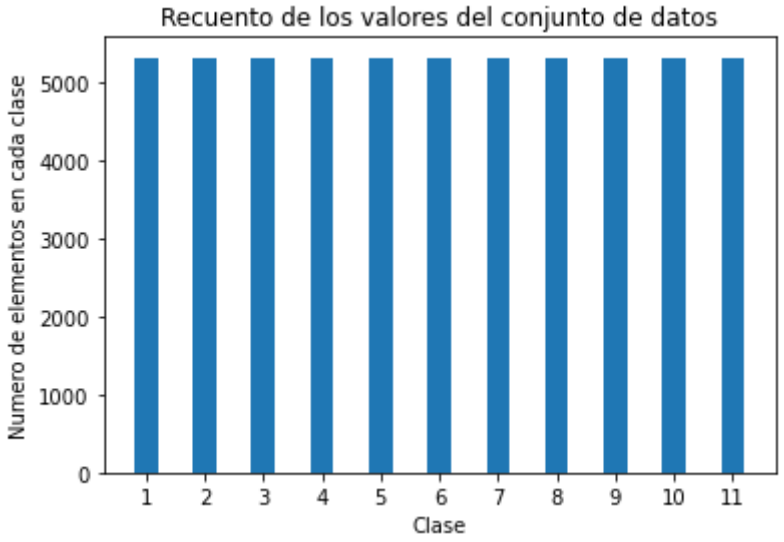
El problema a resolver es el de “diagnóstico de unidad sin sensor”.

Disponemos de un dataset con 58509 muestras que se han extraído de las señales de conducción de corriente eléctrica de un motor. Este motor tiene componentes intactos (que funcionan bien) y defectuosos. Esto da como resultado 11 clases diferentes con diferentes condiciones. Cada condición se ha medido varias veces mediante 12 condiciones de funcionamiento diferentes, es decir, mediante diferentes velocidades, momentos de carga y fuerzas de carga. Las señales de corriente se miden con una sonda de corriente y un osciloscopio en dos fases.

Las clases a la que puede pertenecer una muestra son aquellas con el código en el rango de enteros [1,11].

Por lo tanto nuestro modelo tiene que “predecir” a partir de un input de 64 reales la clase ‘Y’ que indica el estado de salud del motor.

Una información a tener en cuenta sobre nuestro dataset es la distribución de clases, la distribución es la siguiente:



como vemos no hay desbalance de clases por lo que estamos en una situación ideal y no tenemos que lidiar con clases desbalanceadas.

2. Selección de la clase/s de funciones a usar. Justificar cuáles y por qué.

Con el objetivo de no perder tiempo en realizar una optimización prematura es buena idea obtener un modelo funcional simple lo más rápido posible, y una vez que tenemos este modelo buscar los mejores cambios que podemos hacer para evitar los problemas que surjan. Por esta razón en principio usaré combinaciones lineales de los valores observados.

Si después de analizar el error veo que hay un problema de “bias” (sesgo) o de alta varianza tomaré decisiones que vea apropiadas para el tipo que problema que aparezca (si aparece).

Por lo tanto la función de predicción lineal que usaré será:

$$f(x) = w^T * x + b$$

siendo “x” el vector de característica de nuestra muestra de entrenamiento y “w” y “b” los parámetros que nuestro modelo deberá ajustar (la b podría eliminarse añadiendo al vector de características x un 1 al principio).

3. Identificar las hipótesis finales que usará.

En este apartado voy a comentar los pasos que voy a dar para resolver el problema al que nos enfrentamos, no voy a entrar en detalles técnicos, sólo voy a decir el procedimiento que voy a realizar para intentar lograr un buen aprendizaje del dataset. Los aspectos técnicos los comentaré en apartados posteriores que se focalizan más en una parte concreta del proceso.

Al realizar el preprocesamiento de datos me he dado cuenta de que hay muchas variables altamente correladas por lo que utilizar regularización lasso nos va a ayudar mucho de cara a nuestro modelos. Por lo tanto voy a usar esta regularización para todas las hipótesis que realice

Después de todo esto, hay que pensar en los modelos que vamos a usar para el aprendizaje, en primera instancia he decidido utilizar el Perceptrón, he decidido usar este modelo porque si los datos son linealmente separables, nos va a dar una función predictora muy buena. Este algoritmo utiliza el signo de la clase de funciones

$$sign(w^T x_n) \neq y_n$$

y si el signo que le asignamos con el predictor actual difiere del signo que le da la función verdadera, ajustamos los pesos para que ese punto tenga el signo correcto y esta operación la repetimos hasta que no haya mejora, es decir, clasifiquemos todos los puntos de manera correcta o lleguemos a un máximo número de iteraciones. Una de las desventajas de este algoritmo es que no tiene memoria, es decir, si para predecir bien el punto actual tiene que dejar de predecir bien otros puntos lo va a hacer porque no se acuerda de los puntos anteriores además de que es muy sensible al ruido, si el dataset contiene ruido, perceptrón no va a poder converger y va a devolver una solución cualquiera (la recta predictora que tenga en su última iteración). Pese a estas desventajas he decidido usar el perceptrón porque como he dicho antes si los datos son linealmente separables nos va a dar una buena función predictora y porque lo hemos usado durante el curso y me gustaría ver como funciona en un problema "real" pese a que no promete un buen resultado.

También he decidido usar Regresión Logística Multinomial que es un método de clasificación que generaliza la regresión logística para problemas multiclase (porque permite más de dos categorías), modelo que hace uso de la función sigmoide:

$$\sigma(x) = \frac{1}{1+e^{-}x} = \frac{e^x}{e^x+1}$$

La regresión Logística Multinomial, al contrario de la Regresión Logística utiliza el criterio ORV (One vs Rest), donde se entrenan múltiples modelos. Para ello, se modela la probabilidad de que cada muestra pertenezca a una clase, teniendo en cuenta la dependencia entre estas posibilidades. Como la probabilidad de todas las clases suma el 100%, la probabilidad de pertenecer a una clase será de 1 - suma de probabilidades de pertenecer a otras clases.

Por lo tanto la forma de generalizar regresión logística para el caso binario a el caso multiclase es muy fácil. La idea es la siguiente:

$$L(x_1, x_2 \ldots x_N) = \prod_{i=1}^{N} \sigma(w^T x_i)^{[[y==1]]} \; (1 - \sigma(w^T x_i))^{[[y==0]]}$$

Esta ecuación hace el producto de todas las circunstancias en la que la clase vale 1 y todas en las que la clase vale 0 ya que uno de los dos términos para cualquier caso siempre es la unidad porque si no pertenece a la clase 1 por ejemplo, tiene que pertenecer a alguna del resto de las clases (criterio One vs Rest).

Y por último, he decidido usar una herramienta más potente para afrontar este problema, los SVM (support vector machine). Este algoritmo lo que intenta es buscar un hiperplano entre todos los posibles que separan, que cumplan que la distancia mínima entre los puntos de las distintas clases sea lo más grande posible. Esta distancia que tratamos de maximizar es lo que llamamos margen o pasillo. Vamos a ver un ejemplo para dos clases; dado el vector de característica x y el vector de etiquetas y (1 y -1), nuestro objetivo es encontrar una w y una b que en la predicción dada por:

$signo(w^t \phi(x) + b)$

sea correcta para la mayoría de muestras. Así que lo que estamos intentando resolver es el siguiente problema, denominado problema primal:

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$
$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$
$$\zeta_i \geq 0, i = 1, \ldots, n$$

Estamos intentando maximizar el pasillo (minimizando ||w||² = w^tw), mientras que aplicamos una penalización cuando una muestra está mal clasificada o dentro del pasillo. Idealmente el valor de $y_i(w^t \phi(x_i) + b >= 1$ para todas las muestras, lo que indica una predicción perfecta. Pero los problemas no son siempre separables mediante un hiperplano, así que lo que hacemos es ser menos optimistas y permitir que algunos datos estén a una distancia ζ_i de dónde deberían estar, es decir, permitimos que algunos puntos estén en el pasillo. A este problema le llamamos el problema dual y tiene la siguiente formulación

$$\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha$$
$$\text{subject to } y^T \alpha = 0$$
$$0 \leq \alpha_i \leq C, i = 1, \ldots, n$$

donde C es una constante que controla la fuerza de la penalización ζ_i, e es un vector con todo 1 y Q es una matriz semidefinida nxn, Qij es equivalente a $y_i y_j K(x_i x_j)$ $donde\ K(x_i x_j) = \phi(x_i)^T \phi(x_j)$ es el kernel del que hablaremos más adelante. Los términos α_i son los llamados coeficientes duales.

Una vez el problema de optimización está solucionado, la salida de la función de decisión para una muestra x dada es la siguiente:

$$\sum_{i \in SV} y_i \alpha_i K(x_i, x) + b,$$

Una función kernel es una función, tal que si evalúo esas funciones en los datos originales (no en los extendidos) puede llegar a calcular el producto escalar en el espacio extendido, dependiendo de la función núcleo que fijemos así sería la función ϕ que estemos usando aunque nosotros no tendríamos que calcularla. Como tenemos que utilizar en esta práctica modelos lineales ¡, vamos a utilizar un kernel lineal para que las características sigan siendo lineales.

$$K(x, x^{'}) = (x, x')$$

Para aplicar este algoritmo a un problema multiclase como el nuestro vamos a usar una estrategia denominada “One vs All” o “Uno contra Todos”, esta estrategia consiste en entrenar un predictor para cada clase. Para cada predictor, la clase está entrenada contra el resto de clases, es decir, tenemos un problema de dos clases la clase que queremos predecir y el resto. Además de que es muy eficiente computacionalmente hablando, otra ventaja de esta representación es su interpretabilidad. Como cada clase está representada por un clasificador, es posible ganar algo de conocimiento sobre la clase inspeccionando su correspondiente clasificador.

4. Si la base de datos define conjuntos de training y test, únalos en un solo conjunto y genere sus propios conjuntos de training y test. Justifique el mecanismo de partición.

En este problema no se nos proporcionan un conjunto de training y otro de test predefinido por lo que tenemos que generarlo nosotros. Como hemos visto en apartados anteriores, la distribución de clases está balanceada así que no tenemos que preocuparnos por eso, solo tenemos que mantener este balance y asegurarnos que hay datos suficientes en cada conjunto.

Para realizar la partición he decidido hacerlo de manera aleatoria utilizando el método train_test_split proporcionado por scikit learn con una proporción del 68% de las muestras para training y un 32% para test. Lo que nos dejaría el siguiente volúmen de datos:

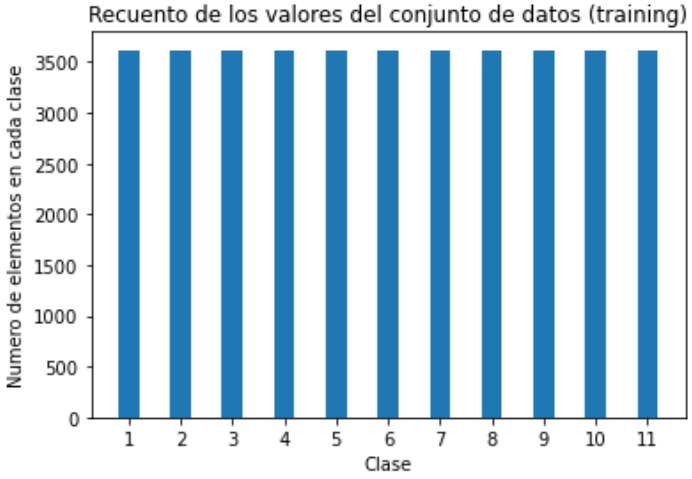
- Training: 39786 datos en total, 3616 para cada clase.
- Test: 18722 datos en total, 1702 para cada clase.

Lo que me parece una cantidad razonable de datos para ambos conjuntos con los que poder hacer una buena evaluación de nuestro ajuste y obtener una buena función predictora.

La función train_test_split la he usado con los siguientes parámetros:

1. Los arrays de características y sus etiquetas.
2. Train_size = 0.68 para indicar que el conjunto de training tenga el 68% de datos que tiene el dataset.
3. Test_size = 0.32 para indicar que el conjunto de test tenga el 32% de los datos que tiene el dataset.
4. stratify = yes para dividir x en grupos homogéneos según y y así mantener el balanceo de clases.

El conjunto de training y de test se nos queda con la siguiente distribución:



Y como vemos hemos mantenido las clases equilibradas tanto en training como test.

5. Justifique todos los detalles del preprocesado de los datos: codificación, normalización,proyección, etc.

Lo primero que voy a hacer es comprobar que no haya valores perdidos en el conjunto de datos, para ello he usado la función isnull() de la librería pandas. Este método recibe un array con los datos del problema y devuelve un bool con los valores true si hay valores perdidos en el dataset y false si no. En nuestro caso, no hay valores perdidos.

Una vez comprobado esto, el siguiente paso es comprobar si hay columnas con el mismo valor siempre. Nos interesa buscar esto porque las columnas que tienen un único valor son inútiles para modelar. Para hacer esto he usado la función unique de numpy que devuelve una lista con los valores diferentes que tiene un array. Una vez hecho este análisis obtenemos que ninguna columna tiene tan solo un valor por lo que no podemos reducir el tamaño del dataset. También he comprobado que no hay filas con los mismos valores, para esto he usado la función duplicated de pandas que te devuelve un array con las filas que están duplicadas, para nuestro dataset tampoco hay filas duplicadas por lo que tampoco podemos eliminar nada.

Lo siguiente que he hecho en esta fase es mirar si hay correlación entre alguna de las características de nuestro dataset. Para ello he utilizado el coeficiente de correlación de Pearson que es una medida de dependencia lineal entre dos variables cuantitativas. Esta medida es independiente de la escala de medida de las variables, por lo que podemos usarla sin haber normalizado los datos (aún no he pensado si nuestro dataset necesita de normalización o no). Esta medida se calcula de la siguiente manera:

$$\rho_{X,Y} = \frac{\sigma_{XY}}{\sigma_X \sigma_Y} = \frac{Cov(X,Y)}{\sqrt{Var(X)Var(Y)}}$$

donde:

- σ_{XY} es la covarianza de (X,Y).
- σ_X es la desviación estándar de la variable X.
- σ_Y es la desviación estándar de la variable Y.

El valor del índice de correlación varía en el intervalo [-1,1], si el valor es cercano a 1 esto indica que existe una fuerte correlación positiva entre esas dos características, es decir, cuando una de ellas aumenta la otra también lo hace en

proporción constante. Y si el valor es cercano a -1 existe una fuerte correlación negativa entre esas dos características, cuando una de ellas crece la otra disminuye en proporción constante.

Para calcular esta medida he usado la función corrcoef que nos devuelve el coeficiente de correlación de Pearson dada una matriz. Los parámetros que recibe esta función son: el array con las variables y observaciones, en nuestro caso nuestro conjunto de entrenamiento, “y” que es un conjunto adicional de variables y observaciones, nosotros no le hemos pasado nada, “rowvar” si este parámetro está a true entonces cada fila representa una variable y cada columna una observación y si está a false es justo lo contrario, nosotros la hemos puesto a false porque es la representación que se ajusta a nuestro problema y por último dtype para indicar el tipo de dato en nuestro caso float64.

Una vez hecho esto hemos buscado las características que tengan un coeficiente > 0.9 o < 0.9 y he obtenido estos resultados:

```
Mostrando características con un coeficiente de correlación de Pearson > 0.9 o < -0.9
6 con 7 correlación: 0.9999986031528334
6 con 8 correlación: 0.9999746916192164
7 con 6 correlación: 0.9999986031528333
7 con 8 correlación: 0.999978518647415
8 con 6 correlación: 0.9999746916192164
8 con 7 correlación: 0.999978518647415
9 con 10 correlación: 0.9999995489971337
9 con 11 correlación: 0.999992875045916
10 con 9 correlación: 0.9999995489971336
10 con 11 correlación: 0.9999939964170695
11 con 9 correlación: 0.9999928750459159
11 con 10 correlación: 0.9999939964170694
12 con 13 correlación: 0.9083605355512249
13 con 12 correlación: 0.9083605355512249
15 con 16 correlación: 0.9214845613290619
16 con 15 correlación: 0.921484561329062
18 con 19 correlación: 0.9999999820333837
18 con 20 correlación: 0.9999994672253323
18 con 21 correlación: 0.9999476877029647
18 con 22 correlación: 0.9999478565687457
18 con 23 correlación: 0.999948688492358
19 con 18 correlación: 0.9999999820333837
19 con 20 correlación: 0.9999995400193776
19 con 21 correlación: 0.9999474836051832
19 con 22 correlación: 0.9999476593557957
19 con 23 correlación: 0.9999485396689294
20 con 18 correlación: 0.9999994672253322
20 con 19 correlación: 0.9999995400193776
20 con 21 correlación: 0.9999456247973104
20 con 22 correlación: 0.999945845093673
20 con 23 correlación: 0.9999472196719082
21 con 18 correlación: 0.9999476877029647
21 con 19 correlación: 0.9999474836051832
21 con 20 correlación: 0.9999456247973104
21 con 22 correlación: 0.9999999845660288
21 con 23 correlación: 0.9999995042199078
22 con 18 correlación: 0.9999478565687457
22 con 19 correlación: 0.9999476593557957
22 con 20 correlación: 0.9999458450936731
22 con 21 correlación: 0.9999999845660288
22 con 23 correlación: 0.9999995666764403
23 con 18 correlación: 0.999948688492358
23 con 19 correlación: 0.9999485396689295
```

```
23 con 20 correlación: 0.9999472196719081
23 con 21 correlación: 0.9999995042199077
23 con 22 correlación: 0.9999995666764403
30 con 31 correlación: 0.9999240713215684
30 con 32 correlación: 0.9991176411053947
31 con 30 correlación: 0.9999240713215684
31 con 32 correlación: 0.9992635385691515
32 con 30 correlación: 0.9991176411053947
32 con 31 correlación: 0.9992635385691515
33 con 34 correlación: 0.9999699870416379
33 con 35 correlación: 0.9995795376084712
34 con 33 correlación: 0.9999699870416379
34 con 35 correlación: 0.9996211915393615
35 con 33 correlación: 0.9995795376084712
35 con 34 correlación: 0.9996211915393615
42 con 43 correlación: 0.9996629039185082
42 con 44 correlación: 0.9963882665194123
43 con 42 correlación: 0.9996629039185082
43 con 44 correlación: 0.9967016894622197
44 con 42 correlación: 0.9963882665194121
44 con 43 correlación: 0.9967016894622197
45 con 46 correlación: 0.9996267337123045
45 con 47 correlación: 0.9964078433883774
46 con 45 correlación: 0.9996267337123045
46 con 47 correlación: 0.9964977731645992
47 con 45 correlación: 0.9964078433883773
47 con 46 correlación: 0.9964977731645993
```

Al hacer este análisis no me esperaba que salieran tantas características con una fuerte relación entre ellas. En principio, mi criterio iba a ser ver las características que tienen correlación entre ellas y eliminar una de manera aleatoria (por ejemplo eliminar siempre la característica con el índice más alto), sin embargo al salir tantas correlaciones he decidido no tocar el dataset y que la regularización se encargue de darle más o menos importancia a estas características porque si quitaba tantas características quizás estaba eliminando información necesaria para el modelo.

Por último, he decidido normalizar los datos porque al calcular los valores mínimos y máximos de cada característica es fácil de apreciar que se encuentran en rangos muy diferentes lo que puede sesgar nuestro modelo a hacer que algunas variables tengan más relevancia que otras. Por esta razón he decidido que es necesario hacer esta normalización de datos.

Para normalizar los datos he decidido estandarizarlos, es decir, reescalar las características de manera que la media será 0 y la desviación estándar 1, siguiendo esta ecuación:

$$x_{stand} = \frac{x - mean(x)}{standard\ deviation(x)}$$

Para hacer esto he usado la función StandardScaler de scikit-learn.

He decidido usar esta técnica porque soluciona el problema de los rangos de las variables, además que facilita la convergencia de algunos algoritmos lineales y es más robusta a 'outliers' o valores atípicos que otras técnicas.

6. Justifique la métrica de error a usar. Discutir su idoneidad para el problema.

La métrica de error que voy a usar será la "Accuracy" que mide el ratio de predicciones correctas frente al total de predicciones hechas:

$$Accuracy = \frac{Nº\ de\ predicciones\ correctas}{Nº\ total\ de\ predicciones}$$

He elegido esta métrica porque funciona bien para problemas de clases balanceadas. Por ejemplo, si tuviéramos un problema con dos clases (A y B) y la clase A tuviera el 98% de los datos del conjunto de entrenamiento y la clase B solo el 2% sería muy fácil llegar al 98% de accuracy simplemente prediciendo cualquier nueva entrada como si fuese de la clase A, para este tipo de problemas sería mejor usar una métrica como "Precision" o "Recall" que permiten fijarnos en el rendimiento para una clase en concreto. Sin embargo si las clases están balanceadas sí nos da una buena guía de si nuestro modelo predice bien o no.

7. Justifique todos los parámetros y el tipo de regularización usada en el ajuste de los modelos seleccionados. Justificar la idoneidad de la regularización elegida.

Tenemos un gran número de características, por lo que utilizar una regularización L1 o Lasso parece una buena idea para reducir el coeficiente de alguna de estas a 0, además hemos visto que tenemos muchas características correladas así que debería reducir el coeficiente de muchas características a 0 por lo que esta parece la regularización que mejores resultados nos va a dar, ya que, además de ayudarnos con el problema del overfitting también nos va a ayudar para saber de qué características aprender algo y de cuáles no.

La regularización Lasso consiste en añadir a la función de coste como penalización el valor absoluto de los coeficientes, por lo tanto la función que vamos a usar para la regularización es la siguiente, donde C es un parámetro que nos sirve para controlar la relevancia de la regularización.

$$min_w \sum_{i=1}^{n} |w_i| + C * FunciónCoste$$

Para el modelo de regresión logística, este algoritmo recibe los siguientes parámetros:

- **penalty**: puede tomar los valores 'l1','l2','elasticnet' y 'none', con este parámetro le indicamos la regularización que queramos aplicar siendo l1 regularización Lasso, l2 regularización Ridge y elasticnet, esta última lo que hace es combinar las regularizaciones Lasso y Ridge, mientras que la regularización Ridge lo que hace es penalizar a los coeficientes que toman valores grandes. Aquí nosotros hemos elegido usar l1 por los motivos comentados anteriormente.
- **dual:** sirve para indicarle si queremos la formulación primal o dual, este parámetro lo pondremos a false porque solo está implementado para la regularización Ridge.
- **tol:** tolerancia para el criterio de parada, este parámetro lo dejaremos con su valor por defecto (1e-4) porque me ha parecido un valor razonable como criterio de parada.
- **C:** este parámetro como hemos dicho, indica la revelancia de la regularización, para este parámetro probaremos con varios y gracias a la validación cruzada veremos cuál se comporta mejor.
- **fit_intercept:** este parámetro cuando está a false obliga a la función predictora que obtenemos pase por el 0,0, como no queremos eso la dejamos en su valor por defecto que es true y así dejamos que el algoritmo aprenda solo sin introducir ninguna restricción extra (que además puede empeorar el predictor).
- **intercept_scaling:** este parámetro solo sirve cuando fit_intercept está a true así que lo dejamos en su valor por defecto que es 1.
- **class_weight:** este parámetro solo sirve cuando las clases están desbalanceadas y en nuestro problema no lo están así que lo dejamos None
- **random_state:** parámetro que sirve para mezclar los datos, he fijado la semilla a 1 porque es la que hemos ido usando en todas las prácticas.
- **solver:** Este parámetro sirve para indicar la técnica de ajuste que queremos usar, nosotros vamos a usar saga porque es el único que sirve con la regularización Lasso y en problemas multiclase.
- **max_iter:** indica el número máximo de iteraciones, para este parámetro me ayudaré del grid para fijarlo y probaré varios valores

porque un número mayor de iteraciones nos puede ayudar a generar un mejor predictor o puede hacer que caigamos en overfitting.

- **multi_class:** este parámetro sirve para indicar si estas en un problema binario o multiclase, nosotros lo dejaremos en auto que es el default para que en el código sea más cómodo pero el problema es multiclase así que lo reconocerá como multiclase.
- **verbose:** solo sirve para otras técnicas de ajuste diferentes a la que hemos escogido así que lo dejamos en 0.
- **warm start:** esto sirve para reusar la llamada anterior de este algoritmo como inicio, nosotros no lo vamos a usar así que lo ponemos a false.
- **n_jobs**: sirve para indicar el número de cores de la CPU a usar, en principio nuestro algoritmo va a acabar en un tiempo razonable así que no voy a usar este parámetro (tendrá valor none).
- **l1_ratio**: solo se usa con la regularización elasticnet así que lo dejaremos a none.

Como técnica de ajuste he decidido usar el algoritmo SAGA, un método iterativo basado en gradientes, el cual nos proporciona una mayor efectividad y ratio de convergencia para muestras de datos grandes. El siguiente pseudocódigo de este algoritmo es el que podemos encontrar en el documento oficial de esta técnica:

https://papers.nips.cc/paper/2014/file/ede7e2b6d13a41ddf9f4bdef84fdc737-Paper.pdf

We start with some known initial vector $x^0 \in \mathbb{R}^d$ and known derivatives $f_i'(\phi_i^0) \in \mathbb{R}^d$ with $\phi_i^0 = x^0$ for each i. These derivatives are stored in a table data-structure of length n, or alternatively a $n \times d$ matrix. For many problems of interest, such as binary classification and least-squares, only a single floating point value instead of a full gradient vector needs to be stored (see Section 4). SAGA is inspired both from SAG [1] and SVRG [5] (as we will discuss in Section 3). SAGA uses a step size of γ and makes the following updates, starting with $k = 0$:

SAGA Algorithm: Given the value of x^k and of each $f_i'(\phi_i^k)$ at the end of iteration k, the updates for iteration $k+1$ is as follows:

1. Pick a j uniformly at random.
2. Take $\phi_j^{k+1} = x^k$, and store $f_j'(\phi_j^{k+1})$ in the table. All other entries in the table remain unchanged. The quantity ϕ_j^{k+1} is not explicitly stored.
3. Update x using $f_j'(\phi_j^{k+1})$, $f_j'(\phi_j^k)$ and the table average:

$$w^{k+1} = x^k - \gamma \left[f_j'(\phi_j^{k+1}) - f_j'(\phi_j^k) + \frac{1}{n} \sum_{i=1}^{n} f_i'(\phi_i^k) \right], \tag{1}$$

$$x^{k+1} = \text{prox}_\gamma^h \left(w^{k+1} \right). \tag{2}$$

The proximal operator we use above is defined as

$$\text{prox}_\gamma^h (y) := \underset{x \in \mathbb{R}^d}{\text{argmin}} \left\{ h(x) + \frac{1}{2\gamma} \|x - y\|^2 \right\}. \tag{3}$$

Al final los hiperparámetros que he probado ha sido solo el valor de C y no con el valor de max_iter porque el tiempo que gastaba realizando estas comprobaciones era muchísimo (más de 15 minutos) y creo que es mejor probar menos parámetros. He elegido probar con C en vez de con max_iter porque creo que C es un parámetro que nos interesa más optimizar al tener tantas variables correladas ya que si lo optimizamos bien nos va a dar mejores resultados que si optimizamos el parámetro max iter. Para hacer el grid de parámetros he usado la función GridSearchCV de scikit learn que recibe como parámetros: el predictor que quieres utilizar (Logistic Regression en este caso), los distintos parámetros que quieres probar, la métrica que quieres usar (Accuracy en este caso) y la estrategia para hacer cross validation que quieres (si no especificas nada se hace 5-fold cross validation, he explicado qué es validación cruzada en el apartado siguiente porque lo he visto más apropiado).

Para el valor de C he probado los valores {0.001,0.01,1,10,100,1000} y he obtenido los siguientes resultados:

```
Cross Validation para Regresión Logística
    mean_fit_time param_C  mean_test_score  std_test_score  rank_test_score
0       15.684163   0.001         0.537577        0.018305                7
1       17.159934    0.01         0.765822        0.018074                6
2       18.974319     0.1         0.772232        0.019961                1
3       20.158403       1         0.771000        0.020027                2
4       19.737434      10         0.770900        0.020119                3
5       19.667192     100         0.770774        0.020106                4
6       19.843667    1000         0.770774        0.020106                4
```

Donde como vemos los mejores resultados los obtenemos con C = 0.01 y con un accuracy de 0.7721562174016325.

```
Parametros seleccionados para RL usando validacion cruzada
{'C': 0.1}

Cross Validation para LR:  0.7722316227051389
```

Para el perceptrón, contábamos con los siguientes parámetros:

- **penalty:** Regularización que vamos a usar, nosotros usaremos la regularización l1 por lo comentado anteriormente.
- **alpha**: Es una constante que multiplica el término de regularización, es decir, es nuestro parámetro C que teníamos en regresión logística. Para ver qué valor es el mejor haremos un grid con distintos valores de alpha.
- **l1_ratio:** este parámetro solo sirve para la regularización elasticnet así que lo dejaremos a default.

- **fit_intercept:** este parámetro cuando está a false obliga a la función predictora que obtenemos pase por el 0,0, como no queremos eso la dejamos en su valor por defecto que es true y así dejamos que el algoritmo aprenda solo sin introducir ninguna restricción extra (que además puede empeorar el predictor).
- **max_iter:** El máximo número de iteraciones sobre el conjunto de entrenamiento. Lo vamos a dejar a 1000 (default) porque me parece un número suficiente de iteraciones, si no tarda mucho el algoritmo probaremos a hacer un grid con este parámetro.
- **tol:** Este es el criterio de parada, el algoritmo parará si la pérdida_actual > pérdida_anterior - tol. Para elegir este parámetro vamos a probar con varios valores y a través de validación cruzada verémos cual es el mejor.
- **shuffle:** Este parámetro indica si los datos deberían estar mezclados después de cada iteración, lo vamos a dejar a true que es su valor por defecto para que no siempre tenga el mismo comportamiento el algoritmo.
- **verbose:** este parámetro indica el nivel de “verbosity”, es decir, cuánto queremos que el algoritmo se comunique con nosotros para saber qué está haciendo en cada momento. Esto a nosotros no nos interesa así que lo dejaremos a 0.
- **eta0:** es el learning rate, es decir, una constante que llevará la “fuerza” de la actualización de pesos. Para elegir un valor óptimo de este parámetro también probaremos varios valores y con validación cruzada veremos cuál es el mejor.
- **n_jobs:** sirve para indicar el número de cores de la CPU a usar, en principio nuestro algoritmo va a acabar en un tiempo razonable así que no voy a usar este parámetro (tendrá valor none).
- **random_state:** es la semilla con la que vamos a hacer la mezcla de datos en cada iteración. Yo la voy a fijar a 1 porque es la semilla que hemos ido utilizando durante todas las prácticas.
- **early_stopping:** Este parámetro sirve para indicar si queremos usar early_stopping, es decir, parar de ejecutar cuando empeoramos el valor de validación de la iteración anterior. No voy a usar este parámetro porque esto implica tener que reducir el training test para tener un conjunto de datos que usar en validación, lo que va a empeorar un poco nuestro predictor.
- **validation_fraction y n_iter_no_change:** son dos atributos que sirven para configurar el early stopping y como no vamos a usar early_stopping no los vamos a necesitar.
- **class_weight:** este parámetro solo sirve cuando las clases están desbalanceadas y en nuestro problema no lo están así que lo dejamos None.

- **warm_start:** esto sirve para reusar la llamada anterior de este algoritmo como inicio, nosotros no lo vamos a usar así que lo ponemos a false.

Para este algoritmo hemos obtenido los siguientes resultados:

Cross Validation para Perceptron

	mean_fit_time	param_tol	param_eta0	mean_test_score	std_test_score	rank_test_score
0	0.759461	0.01	0.001	0.552557	0.022959	1
1	0.708868	0.1	0.001	0.552557	0.022959	1
2	0.700692	1	0.001	0.552557	0.022959	1
3	0.766336	10	0.001	0.552557	0.022959	1
4	0.638569	100	0.001	0.552557	0.022959	1
5	1.175860	0.01	0.1	0.540994	0.018282	20
6	0.780116	0.1	0.1	0.549465	0.028157	16
7	0.684442	1	0.1	0.552557	0.022959	1
8	0.745917	10	0.1	0.552557	0.022959	1
9	0.723301	100	0.1	0.552557	0.022959	1
10	1.785049	0.01	1	0.538907	0.031540	23
11	1.216939	0.1	1	0.540994	0.018282	20
12	0.772659	1	1	0.549465	0.028157	16
13	0.651842	10	1	0.552557	0.022959	1
14	0.756701	100	1	0.552557	0.022959	1
15	1.821543	0.01	5	0.550821	0.031709	14
16	1.592192	0.1	5	0.541596	0.025354	19
17	1.009658	1	5	0.530537	0.044259	25
18	0.681659	10	5	0.552557	0.022959	1
19	0.708345	100	5	0.552557	0.022959	1
20	1.844845	0.01	10	0.549589	0.030368	15
21	1.823491	0.1	10	0.538907	0.031540	23
22	1.150876	1	10	0.540994	0.018282	20
23	0.695503	10	10	0.549465	0.028157	16
24	0.746798	100	10	0.552557	0.022959	1

Como vemos este algoritmo tiene muchos parámetros óptimos, esto se debe a que este algoritmo no es capaz de separar los datos porque no son linealmente separables por lo que el algoritmo para en cuanto llega al número máximo de iteraciones y por eso todos hay tantos óptimos. Con este algoritmo podemos afirmar que los datos no son linealmente separables (algo que ya intuíamos por la descripción del problema). En este algoritmo hemos obtenido una Accuracy del 0.55

```
Parametros seleccionados para Perceptron usando validacion cruzada
{'eta0': 0.001, 'tol': 0.01}

Cross Validation para Perceptron:  0.5525566026970236
```

Para el SVM Lineal voy a usar los siguientes parámetros:

- **C:** este parámetro como hemos dicho, indica la revelancia de la regularización, para este parámetro probaremos con varios y gracias a la validación cruzada veremos cuál se comporta mejor
- **kernel**: con este parámetro indicamos la función kernel que vamos a usar. Como hemos dicho antes vamos a usar la lineal para que nuestro modelo sea lineal.
- **degree**: este parámetro solo tiene sentido cuando usamos un kernel polinomial por lo que lo vamos a dejar en default ya que el algoritmo no lo va a usar
- **gamma**: este parámetro no se usa con el kernel lineal así que lo dejamos a default porque no se va a usar
- **coef0**: este parámetro no se usa con el kernel lineal así que lo dejamos a default porque no se va a usar
- **shrinking**: con este parámetro decidimos si usar la heurística de shrinking o no. Esta heurística es útil cuando el número de iteraciones es grande ya que reduce el tiempo de entrenamiento, sin embargo si si resolvemos el problema de optimización de una manera flexible (por ejemplo, usando una tolerancia de detención grande), usar shrinking puede ocasionar tiempos más grandes. Nosotros esperamos obtener nuestro predictor en un gran número de iteraciones ya que nuestro problema no es fácil de resolver como hemos visto en los otros algoritmos así que lo vamos a dejar a True.
- **probability**: con este método indicamos si queremos habilitar las estimaciones de probabilidad. Este parámetro debe estar habilitado si vamos a usar la función fit y como es la función que estamos usando para obtener un ajuste y calcular el accuracy teenmos que ponerlo a true.
- **tol**: es el parámetro de tolerancia para el criterio de parada. Lo vamos a dejar a 1e-3 porque me parece un valor adecuado como criterio de parada.
- **cache_size**: este parámetro sirve para estimar el tamaño de la cache, lo vamos a dejar como está porque es un parámetro que tiene que ver con el hardware que no es lo que nos ocupa en esta asignatura.
- **class_weight:** este parámetro solo sirve cuando las clases están desbalanceadas y en nuestro problema no lo están así que lo dejamos None.
- **verbose**: este parámetro indica el nivel de “verbosity”, es decir, cuánto queremos que el algoritmo se comunique con nosotros para saber qué está haciendo en cada momento. Esto a nosotros no nos interesa así que lo dejaremos a 0.
- **max_iter:** Limitación fuerte en cuántas iteraciones podemos hacer. Como nosotros queremos que use todas las iteraciones que necesite lo mantenemos a -1.

- **decision_function_shape**: este parámetro sirve para indicar la estrategia. Como hemos dicho nosotros vamos a usar la estrategia one vs rest así que le indicamos que queremos usar esa estrategia
- **break_ties:** este parámetro sirve para indicar el método que usar cuando ocurran empates. Si se pone a true los empates se decidirán según los valores de confianza de la función de decisión y si está a false la clase que gana el empate es la primera. Nosotros lo vamos a dejar a false ya que ponerlo a true gasta mucho tiempo de computación.
- **random_state**: sirve para indicar la semilla. Indicamos la semilla 1 porque con el parámetro probability se introduce un factor aleatorio.

Para este modelo no he realizado grid de parámetros (solo se puede ajustar el parámetro C) porque el tiempo computacional de hacer un grid (he probado hasta grid muy pequeños de 3 valores de C) era altísimo. Así que he decidido utilizar el default (1.0) porque según scikit learn suele ser un valor bueno para C y porque es un valor neutro no penaliza la regularización ni tampoco lo contrario. Este modelo ha conseguido encontrar un buen predictor para este problema, consiguiendo una accuracy de 0.93, es decir, es capaz de clasificar el 93% de los datos de manera correcta.

```
Cross Validation para SVMLineal:  0.9278138845250246
```

8. Selección de la mejor hipótesis para el problema. Discuta el enfoque seguido y el criterio de selección usado. ¿Cúal es su error Eout?

Para estimar las mejores hipótesis he usado validación cruzada. Esta técnica se usa para estimar la precisión de un modelo en un hipotético conjunto de datos de prueba y consiste en coger un dato para hacer la validación y repetir este proceso N veces con N datos diferentes y luego realizar la media del error obtenido con estos datos. El único problema de esta técnica es que hay que repetir el experimento N veces y N puede ser un número muy grande así que el tiempo de cómputo puede dispararse.

$$E_{cv} = \frac{1}{N}\Sigma E_{val}(g_i^-)$$

Para solucionar esto se utiliza la técnica “5-Fold cross-validation”, que lo que hace es dividir los datos de entrenamiento en 5 subconjuntos, y calcular el error del

modelo usando un subconjunto como conjunto de prueba y el resto como conjunto de entrenamiento y este proceso lo repite para cada conjunto en 5 iteraciones. De esta manera en vez de hacer N iteraciones solo tenemos que hacer 5 y ahorramos mucho tiempo de cómputo.

Esta forma de evaluar cómo de buena es cada hipótesis es conveniente porque sin usar validación cruzada, sólo tenemos información sobre cómo funciona nuestro modelo con nuestros datos en la muestra. Idealmente, nos gustaría ver cómo funciona el modelo cuando tenemos nuevos datos en términos de precisión de sus predicciones y con este enfoque podemos hacer una buena estimación de cómo funciona. Hemos obtenido los siguientes resultados de CV:

Modelo	CV-Accuracy
Regresión Logística Multinomial	0.7722316227051389
Perceptrón	0.5525566026970236
SVM Lineal	0.9278138845250246

Con esta estimación obtenemos obtenemos una cota de Eout de esta forma para una medida de error en la que cuanto menor error mejor modelo:

$$E_{out}(g) \leq E_{CV}(g)$$

Como hemos evaluado los modelos con una métrica que cuanto mayor mejor es el modelo vamos a convertir esta métrica en una métrica de porcentaje de error en la clasificación (“Miss-classification”) haciendo 1 - Accuracy y con esto podemos hacer una estimación de su Eout calculando E_{CV} en el conjunto de test. Así que nuestra mejor hipótesis será la que tiene menor Ecv y en nuestro caso es SVM Lineal. Por lo que obtenemos que:

$$E_{Out}(g) \leq 0.07218611547497544$$

```
Cota Eout usando CV para SVM Lineal 0.07218611547497544
```

Y a partir de la desigualdad de Hoeffding podemos obtener una cota probabilística de Eout.

$$E_{out}(g) \leq E_{test}(g) + \sqrt{\frac{1}{2N} log \frac{2}{\delta}}$$

δ= 0.05

La desigualdad de Hoeffding proporciona una cota superior a la probabilidad de que la suma de variables aleatorias se desvíe una cierta cantidad de su valor esperado o, lo que es lo mismo:

$$\mathbb{P}(\mathcal{D}: |\mu - \nu| > \epsilon) \leq 2e^{-2\epsilon^2 N} \quad \text{for any } \epsilon > 0$$

$$\mathbb{P}(\mathcal{D}: |\mu - \nu| \leq \epsilon) \geq 1 - 2e^{-2\epsilon^2 N} \quad \text{for any } \epsilon > 0$$

Ahora vamos a considerar h como una función y f(x) como la función verdadera, ahora μ va a ser la probabilidad de que nuestro predictor h falle Pr([[f(x) != h(x)]]), y para un conjunto de entrenamiento D de tamaño N, ν = Fraction([[f(x) != h(x)]]) on D. Por lo que μ y νson el error fuera de la muestra de h y el error dentro de la muestra de h respectivamente. Por lo que la desigualdad de Hoeffding puede ser reescrita de este modo:

$$P(\mathcal{D}: |\mathrm{E}_{out}(h) - \mathrm{E}_{in}(h)| > \epsilon) \leq 2e^{-2\epsilon^2 N} \text{ for any } \epsilon > 0$$

Ahora si consideramos $\delta = 2e^{-2\epsilon^2 N}$ entonces tenemos que:

$$P(\mathcal{D}: |\mathrm{E}_{out}(h) - \mathrm{E}_{in}(h)| > \epsilon) \leq \delta \Leftrightarrow P(\mathcal{D}: |\mathrm{E}_{out}(h) - \mathrm{E}_{in}(h)| < \epsilon) \geq 1 - \delta$$

o lo que es equivalente:

$$\mathrm{E}_{out}(h) \leq \mathrm{E}_{in}(h) + \epsilon \text{ , with probability at least } 1 - \delta \text{ on } \mathcal{D}$$

Y si escribimos ϵ como una función de N y δ entonces tenemos:

$$\mathrm{E}_{out}(h) \leq \mathrm{E}_{in}(h) + \sqrt{\frac{1}{2N} \log \frac{2}{\delta}} \text{ with probability at least } 1 - \delta \text{ on } \mathcal{D}$$

Y de ahí sale la fórmula que hemos vamos a usar para acotar el error Eout.

Por lo que tenemos la siguiente cota de Eout con un 95% de confianza utilizando Etest.

$$E_{out}(g) \leq 0.1290835744987889$$

Y la siguiente cota de Eout con un 95% de confianza utilizando Ein.

$$E_{out}(g) \leq 0.07590340181024427$$

Las cotas basadas en el test set nos dan un resultado menos optimista y con menor sesgo al estar basada en prueba con datos fuera del conjunto de entrenamiento.

Si el error se hubiera obtenido de una muestra más grande daríamos una cota con un intervalo más reducido, esto muestra la importancia de la cantidad de los datos en el aprendizaje automático, tanto para realizar un buen entrenamiento como para acotar el error.

Por lo tanto el error de nuestro modelo con un 95% de confianza es del 12% por lo que es un modelo bueno.

9. Suponga ahora que Ud. que desea afinar la mejor hipótesis encontrada en el punto anterior usando todos los datos para entrenar su modelo final. Calcule la nueva hipótesis y el error Eout de la misma?. Justifique las decisiones y los criterios que aplique.

En este apartado vamos a coger nuestra hipótesis del SVM lineal con C=1.0 y lo vamos a entrenar usando todos los datos, es decir, sin partir nuestro conjunto de datos en train y test, haciendo esto deberíamos tener un mejor predictor porque hemos usado más datos para entrenamiento. Y una vez lo tenemos vamos a calcular la cota de Eout utilizando la desigualdad de Hoeffding y el error dentro de la muestra (Ein) para obtener una aproximación de Eout.

La cota de Eout utilizando la desigualdad de Hoeffding y Ein es la siguiente:

$$E_{out}(g) \leq 0.07391181187236268$$

Y como vemos al entrenar con más datos, el modelo es capaz de aprender más y por tanto la cota de Eout se ha visto reducida, sin embargo lo malo de utilizar todos los datos para entrenamiento es que no podemos saber con certeza cómo se comporta el modelo ante la llegada de datos nunca vistos, pero como antes se ha

conseguido un modelo que generalizaba bien, vamos a suponer que este modelo también generaliza bien y que por tanto estamos ante un buen modelo.

```
Cota de Eout usando Ein y desigualdad de Hoeffding:  0.07391181187236268
```

Podemos extraer otra cota menos optimista utilizando validación cruzada como hemos explicado en el apartado anterior, así podemos saber cómo va a funcionar el modelo con datos que no haya visto nunca. La cota que obtenemos es:

$$E_{out} \leq 0.16896850221079274$$

2. Problema de Regresión

1. Comprender el problema a resolver. Identificar los elementos X,Y y f del problema y describirlos en detalle

El problema a resolver es la predicción de la temperatura crítica de un superconductor.

Disponemos de un dataset con 21263 muestras, cada muestra tiene 81 características de datos asociados al superconductor. Cada característica está extraída midiendo las siguientes características

	Variable
8	Atomic Mass
	First Ionization Energy
	Atomic Radius
	Density
	Electron Affinity
	Fusion Heat
	Thermal Conductivity
	Valence

y de cada una de ellas tomando las siguientes medidas:

	Feature & Description
10	Mean
	Weighted mean
	Geometric mean
	Weighted geometric mean
	Entropy
	Weighted entropy
	Range
	Weighted range
	Standard deviation
	Weighted standard deviation

Lo que nos da 10 medidas x 8 propiedades de cada superconductor + 1 característica del número total de elementos y la temperatura crítica (variable a predecir) 82 características.

Por tanto nuestro modelo tiene que “predecir” (nuestra función “f” será esa) a partir de un input X de características de un superconductor, el elemento ‘Y’ que indica la temperatura crítica de un superconductor (en Kelvins).

2. Selección de la clase/s de funciones a usar. Justificar cuáles y por qué

Al igual que en el problema de clasificación, con el objetivo de no perder tiempo en optimización prematura se intentará obtener un modelo funcional simple lo más rápido posible, y una vez que tenemos este modelo buscar los mejores cambios que podemos hacer para evitar problemas (si surgen). Por esta razón en principio usaremos combinaciones lineales de los valores observados. Tras analizar el error de validación cruzada y el error de entrenamiento del modelo, dependiendo de si encontramos un problema de ‘bias” (para el que aumentar la complejidad de la función podría ser una solución) o de alta varianza/overfitting se tomarán decisiones acordes al problema para mitigarlo.

Por lo tanto nuestra función de predicción lineal será:

$$f(x) = w^T * x + b$$

Siendo x el vector de características de nuestra muestra de entrenamiento y “w” y “b” los parámetros que el modelo deberá ajustar.

3. Identificar las hipótesis finales que usará.

Para resolver este problema lo primero que haré será hacer el preprocesamiento de datos, ver si hay datos perdidos, filas que se repiten, si es necesario un cambio en la codificación y ver si las características están altamente correladas o no. Si lo están en este caso sí voy a intentar quitar características para no caer en grandes tiempos de ejecución (en el paper del dataset se comenta que las características están altamente correladas así que todo parece indicar que voy a utilizar esta técnica). Una vez hecho esto estudiaré la necesidad de regularización y en su caso cuál es la mejor regularización a aplicar en el problema.

Una vez tenemos esto, tenemos que decidir qué técnica de ajuste de pesos usar. La primera que he elegido es SGD (Stochastic Gradient Descent) he escogido este algoritmo por su eficiencia, aunque tiene una desventaja y es que tiene un gran número de parámetros a escoger que influyen en el algoritmo.

El SGD usado sigue el siguiente pseudocódigo:

- Escoge un vector inicial de pesos w y una learning rate η_0 dado como parámetro
- Repite hasta que llegue al máximo número de iteraciones o hasta que la mejora en una época sea menor que tol (otro parámetro dado).
    - Permuta aleatoriamente el orden de la muestra de entrenamiento (la mezcla).
    - Para cada i = 0 hasta n
        - $w = w - \eta \nabla Q_i(w)$
- $\eta = \eta_0 / t^{0.25}$

Siendo $\nabla Q_i(w)$ una estimación del gradiente de la función a minimizar para todo el dataset pero calculada con una sola muestra, reduciendo así la carga computacional, t es el número de actualizaciones de los pesos realizadas que se usa para ir actualizando el learning rate, disminuyendo su valor conforme aumentamos las iteraciones para que el algoritmo sea capaz de mejorar su convergencia.

La segunda técnica de ajuste que he escogido ha sido la pseudoinversa, este algoritmo se basa en que si nosotros tenemos unos datos de entrada X y unos datos de salida Y, asociados a X, debe existir una función que, aplicada a X, transforme X a Y. El ajuste de pesos de este algoritmo se realiza siguiendo esta fórmula:

$$\mathbf{w} = \mathrm{X}^{\dagger}\mathbf{y}.$$

siendo:

$$\mathrm{X}^{\dagger} = (\mathrm{X}^{\top}\mathrm{X})^{-1}\mathrm{X}^{\top}.$$

Por lo que finalmente obtenemos que:

$$w = (X^T X)^{-1} X^T y$$

Como última técnica de ajuste, voy a usar una técnica que no hemos dado en clase llamada Least Angle Regression (LARS). He decidido usar esta técnica porque funciona bien en datasets con dimensiones grandes como es nuestro caso. Este algoritmo encuentra la característica que está más relacionada al objetivo y mueve la línea de regresión en esa dirección. Sin embargo, puede que haya más de una característica que tenga la misma correlación, en estos casos la línea de regresión se mueve en una dirección equiangular a cada una de las características que estén empatadas, es decir, se mueve en una dirección con el mismo ángulo para todas las características.

4. Genere sus conjuntos de training y test. Justifique el mecanismo de partición.

En este problema no se nos proporcionan un conjunto de training y otro de test predefinido por lo que tenemos que generarlo nosotros.

Para realizar la partición he decidido hacerlo de manera aleatoria utilizando el método train_test_split proporcionado por scikit learn con una proporción del 75% de las muestras para training y un 35% para test. Lo que nos dejaría el siguiente volúmen de datos:

- Training: 15947 instancias.
- Test: 5316 instancias.

Lo que me parece una cantidad razonable de datos para ambos conjuntos con los que poder hacer una buena evaluación de nuestro ajuste y obtener una buena función predictora.

La función train_test_split la he usado con los siguientes parámetros:

5. Los arrays de características y sus etiquetas.
6. Train_size = 0.75 para indicar que el conjunto de training tenga el 70% de datos que tiene el dataset.
7. Test_size = 0.25 para indicar que el conjunto de test tenga el 30% de los datos que tiene el dataset.
8. stratify = no ya que no nos hace falta mantener ninguna clase balanceada porque no estamos en un problema de clasificación

He decidido usar el 75% de los datos para training y el 25% de los datos para test porque es una división que se hace con frecuencia en muchos problemas de aprendizaje automático y funciona bien, además que no deja ningún conjunto con pocos datos por lo que podremos entrenar y evaluar nuestro modelo de manera adecuada.

5. Justifique todos los detalles del preprocesado de los datos: codificación, normalización,proyección, etc. Es decir, todas las manipulaciones sobre los datos iniciales hasta fijar el conjunto de vectores de características que se usarán en el entrenamiento.

Lo primero que voy a hacer es comprobar que no haya valores perdidos en el conjunto de datos, para ello he usado la función isnull() de la librería pandas. Este método recibe un array con los datos del problema y devuelve un bool con los valores true si hay valores perdidos en el dataset y false si no. En nuestro caso, no hay valores perdidos.

Una vez comprobado esto, el siguiente paso es comprobar si hay columnas con el mismo valor siempre. Nos interesa buscar esto porque las columnas que tienen un único valor son inútiles para modelar. Para hacer esto he usado la función unique de numpy que devuelve una lista con los valores diferentes que tiene un array. Una vez hecho este análisis obtenemos que ninguna columna tiene tan solo un valor por lo que no podemos reducir el tamaño del dataset.

Lo siguiente que he hecho en esta fase es mirar si hay correlación entre alguna de las características de nuestro dataset. Para ello he utilizado el coeficiente de correlación de Pearson que es una medida de dependencia lineal entre dos variables cuantitativas. Esta medida es independiente de la escala de medida de las variables, por lo que podemos usarla sin haber normalizado los datos (aún no he pensado si nuestro dataset necesita de normalización o no). Esta medida se calcula de la siguiente manera:

$$\rho_{X,Y} = \frac{\sigma_{XY}}{\sigma_X \sigma_Y} = \frac{Cov(X,Y)}{\sqrt{Var(X)Var(Y)}}$$

donde:

- σ_{XY} es la covarianza de (X,Y).
- σ_X es la desviación estándar de la variable X.

- σ_Y es la desviación estándar de la variable Y.

El valor del índice de correlación varía en el intervalo [-1,1], si el valor es cercano a 1 esto indica que existe una fuerte correlación positiva entre esas dos características, es decir, cuando una de ellas aumenta la otra también lo hace en proporción constante. Y si el valor es cercano a -1 existe una fuerte correlación negativa entre esas dos características, cuando una de ellas crece la otra disminuye en proporción constante.

Para calcular esta medida he usado la función corrcoef que nos devuelve el coeficiente de correlación de Pearson dada una matriz. Los parámetros que recibe esta función son: el array con las variables y observaciones, en nuestro caso nuestro conjunto de entrenamiento, y que es un conjunto adicional de variables y observaciones, nosotros no le hemos pasado nada, rowvar si este parámetro esta a true entonces cada fila representa una variable y cada columna una observación y si está a false es justo lo contrario, nosotros la hemos puesto a false porque es la representación que se ajusta a nuestro problema y por último dtype para indicar el tipo de dato en nuestro caso float64.

Una vez hecho esto hemos usado esa función de numpy y hemos buscado las características que tengan un coeficiente > 0.9 o < 0.9 y he obtenido estos resultados (he puesto solo una muestra de las características correladas, hay muchas más pero no las enseño todas porque ocuparía mucho):

```
Mostrando características con un coeficiente de correlación de Pearson > 0.9 o < -0.9
0 con 2 correlación: 0.9399693178899241
1 con 3 correlación: 0.9637141051403196
2 con 0 correlación: 0.9399693178899241
3 con 1 correlación: 0.9637141051403196
4 con 14 correlación: 0.9643258429128511
4 con 24 correlación: 0.9719217669920448
4 con 34 correlación: 0.932409644531057
4 con 54 correlación: 0.9271281769726211
4 con 74 correlación: 0.963107757226784
5 con 25 correlación: 0.9611803875060202
5 con 75 correlación: 0.9173030169735283
6 con 8 correlación: 0.9609316882446282
6 con 9 correlación: 0.9189727131656074
8 con 6 correlación: 0.960931688244628
8 con 9 correlación: 0.9204986722700432
9 con 6 correlación: 0.9189727131656074
9 con 8 correlación: 0.9204986722700432
10 con 12 correlación: 0.9688702223324346
11 con 13 correlación: 0.992364175027966
11 con 23 correlación: -0.9144544210586475
12 con 10 correlación: 0.9688702223324346
```

Esta vez al haber tantas características correladas voy a aplicar un algoritmo denominado Análisis de componentes principales (PCA) es una técnica utilizada para describir un conjunto de datos en términos de nuevas variables (características) no correlacionadas. Los componentes se ordenan por la cantidad de varianza original que describen, por lo que la técnica es útil para reducir la dimensionalidad de un conjunto de datos.

El proceso entero de obtener los componentes principales de un dataset puede ser simplificado en 5 pasos:

- Calcular la media para cada dimensión de todo el dataset
- Calcular la matriz de covarianza de todo el dataset
- Calculas los eigenvector y los correspondientes eigenvalues
- Ordenar los eigenvector en orden descendente de eigenvalues y elige k eigenvectors con los mayores eigenvalues para formar una matriz de dimensión d x k
- Coger esta matriz de d x k eigenvector para transformar las muestras en el nuevo subespacio.

Un eigenvector es un vector cuya dirección se mantiene invariable cuando se le aplica una transformación lineal.

Siendo A una matriz cuadrada (la matriz de covarianza), v un vector y λun escalar que satisface que Av = λv, entonces λ es lo que llamamos eigenvalue asociado al eigenvector v de A.

Los eigenvalues siguen la siguiente ecuación característica:

$$det(A - \lambda I) = 0$$

donde I es la matriz identidad y λes lo único desconocido de la ecuación.

Una vez decidido que voy a usar PCA, voy a estandarizar los datos antes de aplicar PCA ya que en esta técnica nos interesa escoger los componentes que maximicen la varianza. Estandarizar los datos consiste en reescalar las características de manera que la media será 0 y la desviación estándar 1, siguiendo esta ecuación:

$$x_{stand} = \frac{x - mean(x)}{standard\ deviation(x)}$$

Para hacer esto he usado la función StandardScaler de scikit-learn y tras esto he ajustado el modelo con la función fit_transform(). Para aplicar PCA he utilizado la función pca de scikit learn con el parámetro n_components = 0.95 para que

seleccione el número de componentes de modo que la cantidad de varianza tenga que ser mayor que el 95%.

Tras realizar todos los pasos me he quedado con un conjunto de entrenamiento que tiene 17 de características de las 80 que teníamos al principio.

6. Justifique la métrica de error a usar. Discutir su idoneidad para el problema.

Como métrica de error para medir la calidad del modelo voy a usar principalmente MAE (Mean Absolute Error) que calcula el valor medio de los errores absolutos. Estamos ante una métrica que cuanto más pequeña sea mejor resultado obtenemos.

$$MAE = \frac{1}{N}\sum_{i=1}^{N} |y_i - ypredicted_i|$$

Ambas métricas son adecuadas para problemas de regresión y tienen la ventaja de ser fácilmente interpretables ya que nos permiten evaluar el modelo de forma simple con solo un número. Además MAE nos da una media del error absoluto que es una métrica fácilmente interpretable y adecuada ya que un error el doble de grave nos basta con penalizarlo el doble.

7.Justifique todos los parámetros y el tipo de regularización usada en el ajuste de los modelos seleccionados. Justificar la idoneidad de la regularización elegida.

Al principio, al haber tantas variables correladas la regularización Lasso fue la primera que se me pasó por la cabeza ya que esta pone a 0 los coeficientes de algunas variables lo que nos puede ayudar a “eliminar” algunas variables. Sin embargo, al utilizar PCA nos hemos quedado con las variables no correlacionadas por lo que mi interés por Lasso ha bajado y puede que nos interese más una regularización Ridge que lo que busca es reducir la complejidad del modelo manteniendo todas las variables en el modelo. Si tuviera que quedarme con una

elegiría Ridge porque no creo que tras utilizar PCA sea necesario dejar algún coeficiente a 0, sin embargo como es la primera vez que utilizo PCA voy a probar con ambas regularizaciones de manera experimental y voy a ver cuál es la mejor. En principio, creo que la regularización Ridge va a dar mejores resultados por lo comentado anteriormente.

Regularización Lasso consiste en añadir a la función de coste como penalización el valor absoluto de la magnitud de los coeficientes:

$$min_w \alpha * \sum_{i=1}^{N} |w_i| + FuncionCoste$$

Regularización Ridge consiste en añadir a la función de coste como penalización la magnitud al cuadrado de los coeficientes:

$$min_w \alpha * \sum_{i=1}^{N} w_i^2 + \text{FuncionCoste}$$

Para ver que parámetros (que regularización es mejor, que learning_rate es mejor etc..) son mejores voy a utilizar Cross-Validation y voy a hacer un grid de parámetros.

Para hacer el grid de parámetros he usado la función GridSearchCV de scikit learn que recibe como parámetros: el predictor que quieres utilizar (Logistic Regression en este caso), los distintos parámetros que quieres probar, la métrica que quieres usar (MAE en este caso) y la estrategia para hacer cross validation que quieres (si no especificas nada se hace 5-fold cross validation).

Cross Validation se usa para estimar la precisión de un modelo en un hipotético conjunto de datos de prueba y consiste en coger un dato para hacer la validación y repetir este proceso N veces con N datos diferentes y luego realizar la media del error obtenido con estos datos. El único problema de esta técnica es que hay que repetir el experimento N veces y N puede ser un número muy grande así que el tiempo de cómputo puede dispararse.

$$E_{cv} = \frac{1}{N} \Sigma E_{val}(g_i^-)$$

Para solucionar esto se utiliza la técnica “5-Fold cross-validation”, que lo que hace es dividir los datos de entrenamiento en 5 subconjuntos, y calcular el error del modelo usando un subconjunto como conjunto de prueba y el resto como conjunto de entrenamiento y este proceso lo repite para cada conjunto en 5 iteraciones. De

esta manera en vez de hacer N iteraciones solo tenemos que hacer 5 y ahorramos mucho tiempo de cómputo.

Para SGD voy a comentar los parámetros que había disponibles y aquellos que he usado y que no he usado:

- **loss**: Con este parámetro indicamos la función de pérdida a usar. Nosotros vamos a usar la función 'squared_loss' (es la que viene por defecto) porque es la que hemos usado durante todo el curso.
- **penalty**: Con este parámetro indicamos la regularización que queremos usar. Como hemos dicho vamos a probar con la regularización Lasso (l1) y la Ridge (l2) y veremos cuál funciona mejor.
- **alpha**: es una constante que multiplica el término de regularización, con ella podemos controlar la "fuerza" de la regularización que aplicamos. Para esta variable tambien probaremos varios valores y veremos cuál es lo mejor
- **l1_ratio**: este parámetro solo sirve para la regularización elasticnet por lo que no lo vamos a usar.
- **fit_intercept**: este parámetro cuando está a false obliga a la función predictora que obtenemos pase por el 0,0, como no queremos eso la dejamos en su valor por defecto que es true y así dejamos que el algoritmo aprenda solo sin introducir ninguna restricción extra (que además puede empeorar el predictor).
- **max_iter**: Con este parámetro indicamos el número de iteraciones que realizamos sobre los datos. Lo he dejado a 1000 porque me parece un valor suficiente para obtener un buen resultado.
- **tol**: es el parámetro de tolerancia para el criterio de parada. Lo vamos a dejar a 1e-3 porque me parece un valor adecuado como criterio de parada.
- **shuffle**: Con este parámetro indicamos si queremos mezclar los datos después de cada iteración. Lo vamos a poner a True (es el valor por defecto) para que en cada iteración el algoritmo se comporte de manera diferente.
- **verbose**: este parámetro indica el nivel de "verbosity", es decir, cuánto queremos que el algoritmo se comunique con nosotros para saber qué está haciendo en cada momento. Esto a nosotros no nos interesa así que lo dejaremos a 0.
- **epsilon**: este parámetro solo se usa para algunas funciones de pérdida en concreto. Con nuestra función de pérdida no se usa por lo que no le hacemos caso
- **random_state**: es la semilla, la voy a fijar a 400 sin tener ningún criterio, es una semilla, mientras la fijemos para todos los algoritmos la misma da igual su valor.

- **learning_rate**: con este parámetro indicamos como queremos que vaya cambiando el learning rate. Nosotros hemos elegido 'invscaling' que actualiza el learning rate como hemos comentado en el apartado de hipótesis, siguiendo esta fórmula: $\eta = \eta_0 / t^{0.25}$ porque es lo recomendado por scikit learn y porque me ha parecido una buena manera de ir regulando el learning_rate.
- **eta0**: este parámetro sirve para indicar el valor del learning rate inicial, como vimos en la práctica 1 este parámetro puede ser muy influyente en el ajuste por lo que vamos a probar con varios valores y ver cuál da mejores resultados.
- **power_t**: es el exponente para utilizar el escalado inverso de learning rate. Lo he dejado a 0.25 porque es el recomendado por scikit learn
- **early_stopping**: Este parámetro sirve para indicar si queremos usar early_stopping, es decir, parar de ejecutar cuando empeoramos el valor de validación de la iteración anterior. No voy a usar este parámetro porque esto implica tener que reducir el training test para tener un conjunto de datos que usar en validación, lo que va a empeorar un poco nuestro predictor.
- **validation_fraction y n_iter_no_change:** estos parámetros solo sirve si activamos el early_stopping así que no les vamos a hacer caso.
- **warm_start**: este parámetro sirve para usar la solución anterior de la llamada a fit como inicialización para este algoritmo. Como no queremos hacer eso lo dejamos a False.
- **average**: cuando está a true calcula la media de pesos de todas las actualizaciones y guarda el resultado. Como no nos interesa hacer esto lo he dejado a False.

Con lo cual vamos a dejar todos los parámetros por defecto y vamos a probar valores para alpha, eta0 y regularización. Como el algoritmo tarda muy poco en ejecutarse he decidido probar muchos parámetros y ver cuál funciona mejor. Los parámetros con los que he probado son:

- regularización l1 o l2
- alpha: desde 1e-20 hasta 10000 con saltos de un cero en un cero (1, 10, 100, 1000…)
- eta0: desde 1e-4 hasta 1 con saltos de cero en cero (0.01, 0.1, 1…)

Y los mejores parámetros que obtenemos son:

```
Resultados de selección de hiperparámetros por CV:
SGD mejores parámetros:  {'alpha': 0.1, 'eta0': 0.0001, 'penalty': 'l2'}
SGD CV-MAE:  17.106953260524772
```

Como vemos finalmente ha sido mejor utilizar regresión Ridge como habíamos predicho. Sin embargo, el óptimo se alcanza con distintos parámetros de alpha y regularización por lo que podemos decir que la regularización no es muy importante.

Para el caso de la pseudoinversa tenemos los siguientes parámetros:

- **fit_intercept**: este parámetro cuando está a false obliga a la función predictora que obtenemos pase por el 0,0, como no queremos eso la dejamos en su valor por defecto que es true y así dejamos que el algoritmo aprenda solo sin introducir ninguna restricción extra (que además puede empeorar el predictor).
- **normalize**: este parámetro si está a true x ser normalizará antes de la regresión restando la media y dividiendo por la norma l2. Como no queremos que se normalicen (ya hemos estandarizado los datos) lo vamos a dejar a false.
- **copy_x**: si está a true x será copiado y si no puede que se sobreescriba x, como no queremos que x se sobreescriba vamos a dejar este parámetro a true.
- **n_jobs**: este parámetro sirve para indicar el número de procesadores que queremos usar para calcular el modelo, lo vamos a dejar a default porque este parámetro no interesa para nada en esta práctica.
- **positive**: si este parámetro está a true fuerza a todos los coeficiente a ser positivos, nosotros no queremos forzar esta situación así que lo vamos a dejar a false.

Como vemos en este algoritmo no necesitamos probar ningún parámetro por lo que simplemente he lanzado el algoritmo y he obtenido los siguientes resultados:

```
Pseudoinversa CV-MAE:  17.12908375085755
```

Para el caso de Lars estos son los parámetros que hay disponible y uno a uno iré diciendo si lo he usado y que valor ha tomado:

- **fit_intercept**: ste parámetro cuando está a false obliga a la función predictora que obtenemos pase por el 0,0, como no queremos eso la dejamos en su valor por defecto que es true y así dejamos que el algoritmo aprenda solo sin introducir ninguna restricción extra (que además puede empeorar el predictor).
- **verbose**: este parámetro indica el nivel de “verbosity”, es decir, cuánto queremos que el algoritmo se comunique con nosotros para saber qué

está haciendo en cada momento. Esto a nosotros no nos interesa así que lo dejaremos a 0.

- **normalize**: este parámetro si está a true x ser normalizará antes de la regresión restando la media y dividiendo por la norma l2. Como no queremos que se normalicen (ya hemos estandarizado los datos) lo vamos a dejar a false.
- **precompute**: este parámetro sirve para agilizar los cálculos. Lo he dejado a 'auto' porque usa la matriz de Gram para realizar esta mejora de tiempo y como no sé muy bien qué es una matriz de Gram he decidido que scikit learn diga cuándo se usa y cuando no.
- **n_nonzero_coefs**: el número de coeficientes que toman un valor distinto de 0, he dejado el valor por defecto (500) ya que es el que recomienda scikit learn.
- **eps**: este parámetro sirve para definir la precisión de la máquina a la hora de realizar cálculos intermedios, lo he dejado en su valro por defecto porque era float y nuestro dataset solo tiene variables en coma flotante por lo que la precisión también tiene que estar en coma flotante.
- **copy_x**: si está a true x será copiado y si no puede que se sobreescriba x, como no queremos que x se sobreescriba vamos a dejar este parámetro a true.
- **fit_path**: sirve para guardar el camino en una variable lo he dejado a true que es su valor por defecto por si me servía la información para algo pero en realidad podría haberlo puesto a false, no influye en nada al algoritmo.
- **jitter**: parámetro que sirve para hacer un límite superior de un parámetro de ruido uniforme que se agregará a los valores de y. Lo he dejado a none porque no queremos agregar nada a las y que predecimos.
- **random_state**: este parámetro fija la semilla y solo se usa si está jitter activo (con un valor diferente a none) por lo que no le hacemos caso.

Para este algoritmo tampoco hay ningún parámetro con el que tengas que probar diferentes combinaciones por lo que simplemente lanzamos el algoritmo. Tampoco tiene ningún parámetro para fijar la regularización que hacer (había un modelo que implementa lasso + este modelo pero he decidido no usarlo porque ya hemos visto que la regularización en sgd no era un factor muy relevante).

```
LARS CV-MAE 17.12908375085755
```

8.Selección de la mejor hipótesis para el problema. Discuta el enfoque seguido y el criterio de selección usado. ¿Cúal es su error Eout?

Para estimar las mejores hipótesis he usado validación cruzada. Esta técnica se usa para estimar la precisión de un modelo en un hipotético conjunto de datos de prueba y consiste en coger un dato para hacer la validación y repetir este proceso N veces con N datos diferentes y luego realizar la media del error obtenido con estos datos. El único problema de esta técnica es que hay que repetir el experimento N veces y N puede ser un número muy grande así que el tiempo de cómputo puede dispararse.

$$E_{cv} = \frac{1}{N}\Sigma E_{val}(g_i^-)$$

Para solucionar esto se utiliza la técnica “5-Fold cross-validation”, que lo que hace es dividir los datos de entrenamiento en 5 subconjuntos, y calcular el error del modelo usando un subconjunto como conjunto de prueba y el resto como conjunto de entrenamiento y este proceso lo repite para cada conjunto en 5 iteraciones. De esta manera en vez de hacer N iteraciones solo tenemos que hacer 5 y ahorramos mucho tiempo de cómputo.

Esta forma de evaluar cómo de buena es cada hipótesis es conveniente porque sin usar validación cruzada, sólo tenemos información sobre cómo funciona nuestro modelo con nuestros datos en la muestra. Idealmente, nos gustaría ver cómo funciona el modelo cuando tenemos nuevos datos en términos de precisión de sus predicciones y con este enfoque podemos hacer una buena estimación de cómo funciona. Hemos obtenido los siguientes resultados de CV:

Modelo	CV-MAE
SGD	17.106953260524772
Regresión LIneal	17.12908375085755
LARS	17.12908375085755

Como vemos SGD nos da mejores resultados que la regresión lineal y LARS (ambas dan una solución muy parecida) ya que la métrica de error que hemos usado (Mean absolute error) indica cuánto varía la solución que damos con la solución verdadera, por tanto, cuanto más pequeño sea el resultado mejor es el modelo.

Con esta estimación obtenemos obtenemos una cota de Eout de esta forma para una medida de error en la que cuanto menor error mejor modelo:

$$E_{out}(g) \leq E_{CV}(g)$$

Así que podemos obtener una primera cota de Eout sería esta:

$$E_{out}(g) \leq 17.106953260524772$$

Y a partir de la desigualdad de Hoeffding podemos obtener una cota probabilística de Eout.

$$E_{out}(g) \leq E_{test}(g) + \sqrt{\frac{1}{2N} log \frac{2}{\delta}}$$

δ= 0.05

La desigualdad de Hoeffding proporciona una cota superior a la probabilidad de que la suma de variables aleatorias se desvíe una cierta cantidad de su valor esperado o, lo que es lo mismo:

$$\mathbb{P}(\mathcal{D}: |\mu - \nu| > \epsilon) \leq 2e^{-2\epsilon^2 N} \quad \text{for any } \epsilon > 0$$

$$\mathbb{P}(\mathcal{D}: |\mu - \nu| \leq \epsilon) \geq 1 - 2e^{-2\epsilon^2 N} \quad \text{for any } \epsilon > 0$$

Ahora vamos a considerar h como una función y f(x) como la función verdadera, ahora μ va a ser la probabilidad de que nuestro predictor h falle Pr([[f(x) != h(x)]]), y para un conjunto de entrenamiento D de tamaño N, ν = Fraction([[f(x) != h(x)]]) on D. Por lo que μ y νson el error fuera de la muestra de h y el error dentro de la muestra de h respectivamente. Por lo que la desigualdad de Hoeffding puede ser reescrita de este modo:

$$P(\mathcal{D}: |\mathrm{E}_{out}(h) - \mathrm{E}_{in}(h)| > \epsilon) \leq 2e^{-2\epsilon^2 N} \text{ for any } \epsilon > 0$$

Ahora si consideramos $\delta = 2e^{-2\epsilon^2 N}$ entonces tenemos que:

$$P(\mathcal{D}: |\mathrm{E}_{out}(h) - \mathrm{E}_{in}(h)| > \epsilon) \leq \delta \Leftrightarrow P(\mathcal{D}: |\mathrm{E}_{out}(h) - \mathrm{E}_{in}(h)| < \epsilon) \geq 1 - \delta$$

o lo que es equivalente:

$$\mathrm{E}_{out}(h) \leq \mathrm{E}_{in}(h) + \epsilon \text{ , with probability at least } 1 - \delta \text{ on } \mathcal{D}$$

Y si escribimos ϵ como una función de N y δ entonces tenemos:

$$\mathrm{E}_{out}(h) \leq \mathrm{E}_{in}(h) + \sqrt{\frac{1}{2N}\log\frac{2}{\delta}} \text{ with probability at least } 1 - \delta \text{ on } \mathcal{D}$$

Y de ahí sale la fórmula que hemos vamos a usar para acotar el error Eout.

Por lo que tenemos la siguiente cota de Eout con un 95% de confianza utilizando Etest.

$$E_{out}(g) \leq 17.25342127201309$$

Y la siguiente cota de Eout con un 95% de confianza utilizando Ein.

$$E_{out}(g) \leq 17.103782862538367$$

Las cotas basadas en el test set nos suelen dar un resultado menos optimista, y con menor sesgo al estar basada en prueba con datos fuera del conjunto de entrenamiento.

Si el error se hubiera obtenido de una muestra más grande daríamos una cota con un intervalo más reducido, esto muestra la importancia de la cantidad de los datos en el aprendizaje automático, tanto para realizar un buen entrenamiento como para acotar el error.

Por lo tanto la predicción de nuestra mejor hipótesis se aleja aproximadamente 17.3K (grados Kelvin) de la original con un 95% de confianza lo que es bastante así que nuestro predictor no es muy bueno. Quizás este predictor se podría mejorar aumentando la clase de funciones (añadiendo por ejemplo los conjuntos de los polinomios de orden dos o más) pero tendríamos que tener cuidado en no tener problemas de overfitting. Como esta práctica no trata de hacer un ajuste buenísimo de los datos, he decidido no realizar este estudio pero comento esta posibilidad ya que probablemente mejore el predictor.

9. Suponga ahora que Ud. que desea afinar la mejor hipótesis encontrada en el punto anterior usando todos los datos para entrenar su modelo final. Calcule la nueva hipótesis y el error Eout de la misma?. Justifique las decisiones y los criterios que aplique.

Para la realización de este apartado he entrenado nuestro modelo del Gradiente Descendente Estocástico con los parámetros:

alpha=0.1, eta0=0.0001, penalty='l2'

con todos los datos del modelo, para ello les he aplicado las mismas transformaciones al dataset completo que le apliqué al dataset de entrenamiento, es decir, una estandarización y un pca con los mismos parámetros que cuando se lo apliqué al conjunto de entrenamiento. Una vez hecho esto entrenamos al modelo con nuestros datos transformados y obtenemos los siguientes resultados:

Ein = 17.12207441044398

Y una cota de Eout (haciendo uso de la desigualdad de Hoeffding comentada anteriormente) de:

$$E_{out}(g) \leq 17.131388061535223$$

Y como era de esperar, cuando usamos más datos obtenemos un mejor modelo ya que este ha podido aprender más y por tanto obtener un mejor modelo y una mejor cota de Eout (usando solamente Ein), esta cota es una cota optimista, por lo que si tuviéramos otro conjunto de datos distintos a los nuestros (Etest) podríamos fijar una cota menos optimista.

Podemos extraer otra cota menos optimista utilizando validación cruzada como hemos explicado en el apartado anterior, así podemos saber cómo va a funcionar el modelo con datos que no haya visto nunca. La cota que obtenemos es:

$$E_{out} \leq 18.244311862598288$$