Práctica 3

Celia Arias Martínez

En esta práctica vamos a realizar el ajuste y selección del mejor predictor lineal para un conjunto de datos dados. Vamos a tener dos problemas: uno de regresión y otro de clasificación, por lo que haremos dos secciones, y desarrollaremos dentro de cada sección los pasos que llevaremos a cabo, todos ellos encaminados a seleccionar el mejor modelo y la mejor estimación de error E_{out}

Regresión

Para este problema utilizamos la base de datos *Superconductivty Data Data Set* encontrada aqui.

Comprensión del problema a resolver. Identificación de los elementos X, Yyf.

Lo primero que tenemos que identificar es el problema que queremos resolver. En este caso, ayudados por la información proporcionada por el dataset, podemos ver que lo que tenemos que resolver es la asignación de una temperatura crítica a un superconductor, dadas unas características como el número de elementos, la masa atómica, el radio atómico, el punto de fusión, la entropía, etcétera. En concreto tenemos datos sobre 21263 superconductores, y de cada uno de ellos tenemos 81 características.

Por tanto los elementos son:

X : matriz con las características de los superconductores

Y: temperatura crítica del superconductor

f: función que asocia a cada superconductor su temperatura crítica.

Selección de modelos.

Lo primero que tenemos que fijar al resolver un problema de aprendizaje automático es la clase de funciones a usar. No he realizado transformaciones de

los datos, pues, tras analizar las variables que actúan y representarlas mediante t-sne no he encontrado ninguna relación entre ellas que me indicara que iba a conseguir mejores resultados ni con transformaciones lineales ni con no lineales.

De modelos he utilizado un modelo de regresión lineal que implementa gradiente descendiente estocástico. En total he probado con 16 modelos, cada uno de ellos con unos parámetros diferentes.

Los parámetros utilizados han sido:

- Función de pérdida: *squared_loss* y *epsilon_insensitive*

  Estas funciones de pérdida definen hacia dónde avanza el algoritmo. *squared_loss* es la utilizada por defecto y es el ajuste por mínimos cuadrados ordinario. *epsilon_insensitive* ignora los errores que son menores que *epsilon* (por defecto 0.1) y es la función de pérdida utilizada en SVR (Support Vector Regression, una variante de SVM). SVR funciona igual que SVM estableciendo un margen de tolerancia en el que se permiten los errores. He utilizado estas dos funciones de pérdida ya que me ha parecido interesante comprobar el método de regresión lineal estudiado en teoría con el equivalente de SVM que también lo hemos estudiado en teoría.

- Learning rate: *optimal* y *adaptive*

  Learning rate es la tasa de aprendizaje, es decir, a qué velocidad avanza gradiente descendiente estocástico. *optimal* tiene la siguiente fórmula: $\eta = 1/(\alpha * (t + t_0))$. En *adaptive* $\eta = \eta_0$ siempre que vayamos en la dirección adecuada, y si alcanzamos un número determinado de iteraciones sin hacer que el error vaya a menos se divide η por 5. He seleccionado estos dos tipos de learning rate para probar dos modelos de tasa de aprendizaje: uno en el que learning rate está fijo y otro en el que se va adaptando a la situación en la que se encuentre. La ventaja de este último es que se comportará mejor cuando esté cerca de la solución, pues podrá avanzar en pasos más pequeños, pero también tiene más probabilidades de caer en mínimos locales.

- eta_0 : 0.01

  Valor inicial de learning rate para *adaptive*. Valor por defecto.

- Máximo de iteraciones: 10000

  El valor por defecto es 1000, pero viendo que siempre se alcanzaba el máximo de iteraciones sin obtener un resultado final he decidido aumentarlo algo más, ya que he comprobado que con ese valor el tiempo de ejecución no es demasiado alto y los valores obtenidos en *cross-validation* eran mejores.

Para la regularización también he empleado dos parámetros, que probaré con los modelos antes mencionados para ver cuál se adapta mejor a nuestro problema.

- Tipo de regularización: l1 y l2

l2, también llamada regularización Ridge, es la regularización por defecto, y favorece que los coeficientes de los atributos sean bajos. Su fórmula es $sum(w^2) < C$. Es efectiva cuando la mayoría de atributos son relevantes, lo cual puedo pensar que es nuestro caso, dado el preprocesamiento de datos que realizamos. l1, también llamada regularización Lasso, favorece que alguno de los coeficientes valgan cero. La fórmula es: $sum(abs(w)) < C$. l2 funciona mejor si la mayoría de los atributos son relevantes y l1 si no están correlados entre ellos, y tienen efectos contrapuestos: l2 minimiza el efecto de la correlación y l1 la irrelevancia de atributos. Como en el preprocesamiento de datos que he realizado he intentado eliminar la irrelevancia y la correlación he decidido probar con los dos modelos de regularización.

- Constante de regularización: 0.0001 y 0.001

  Constante que multiplica el efecto de la regularización. Si el valor es más alto la regularización es más alta. 0.0001 es el valor que tiene por defecto y 0.001 he decidido probarlo para ver qué resultado tenía potenciar la regularización de nuestros modelos. No he probado con valores más altos pues al hacer *cross-validation* no he obtenido sobreajuste, y por tanto no creo que sea necesaria aplicar una regularización más estricta.

Partición en test y entrenamiento.

He dividido el conjunto de datos proporcionado en test y entrenamiento, según la proporción 20 %/80 % vista en teoría. Para ello he utilizado la función *train_test_split*, haciendo una mezcla previa de los datos, para evitar sesgo por que estuviesen ordenados de alguna forma previamente.

La base de datos no traía conjuntos diferenciados de train y test, por lo que no he podido hacer la comprobación de si los resultados eran parecidos utilizando esta división o la proporcionada.

Preprocesado de datos.

Lo primero que hacemos es **normalizar** los datos. Creo que esto es necesario porque así podemos tener una idea de la verdadera importancia de una variable o de otra, así como las diferencias entre las varianzas de cada una de ellas.

Para ello he utilizado la función *StandardScaler*, he escalado con los datos de entrenamiento y he aplicado los valores obtenidos a los de test, con el objetivo de no hacer *data snooping*.

Después he decidido **visualizar** los datos con *t-sne*, para ver si me podía aportar más información sobre el problema. He comentado el código pues el tiempo de ejecución era demasiado alto pero adjunto las imágenes obtenidas. Los parámetros

utilizados son los de defecto de t-sne, es decir, *perplexity* = 30, *early_exaggeration* = 12, *learning_rate* = 200.

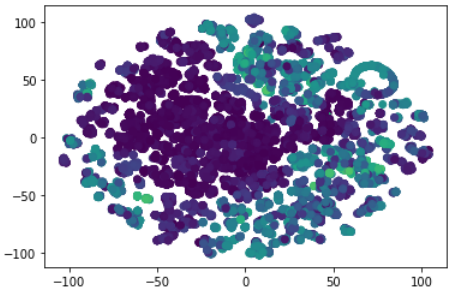


Figura 1: t-SNE parámetros por defecto

Como puede observarse la visualización por t-sne no ha añadido mucha información al problema que tenemos que tratar.

Lo siguiente que hacemos es ver si hay algún **valor perdido** en el conjunto de datos que nos han proporcionado. Esto lo hacemos primero convirtiendo nuestra matriz en un data frame de pandas, y después llamando a la función:

```
np.all(df_train.notnull())
```

Nos devuelve *True*, por tanto significa que no hay valores perdidos.

Lo siguiente que vamos a estudiar es la **correlación** entre los atributos.

Para ello utilizamos la función *corr()* del data frame de pandas, y utilizamos la función *heatmap* de la biblioteca *seaborn*. Dado el gran número de características que tenemos he decidido solo colorear las relaciones fila-columna que tengan coeficiente de *Pearson* mayor que 0.95. La gráfica obtenida es:

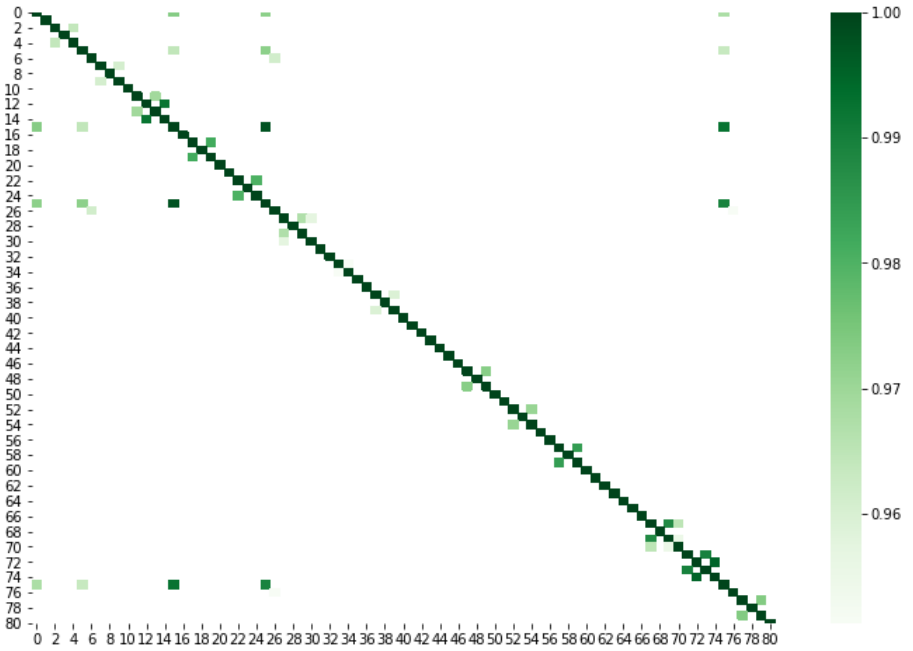


Figura 2: Matriz de correlación

Podemos observar que, en general, no existe una correlación generalizada entre las características. Sí que se da más entre características que están cercanas entre ellas, esto puede deberse a que algunas de las características son diferentes medias de las variables (media aritmétrica, geométrica, etc) y están situadas consecutivamente.

He decidido eliminar las características que tengan más de 0.9 de coeficiente de Pearson entre ellas. Para ello hago uso de una función que muestra por pantalla las variables que tengan más de dicho coeficiente en valor absoluto, y sin mostrar los pares redundantes. Los resultados obtenidos han sido:

Carac. 1	Carac. 2	Coef Pearson	Carac.1	Carac.2	Coef Pearson
15	25	0.998	11	13	0.969
72	74	0.995	0	75	0.968
72	74	0.995	27	29	0.967
15	75	0.993	67	70	0.965
12	14	0.992	5	15	0.965
71	73	0.99	2	4	0.964
25	75	0.99	5	75	0.964
67	69	0.988	6	126	0.962
57	59	0.984	7	9	0.961
17	19	0.982	37	39	0.96

Carac. 1	Carac. 2	Coef Pearson	Carac.1	Carac.2	Coef Pearson
22	24	0.980	27	30	0.957
77	79	0.974	69	70	0.955
0	15	0.973	33	34	0.951
47	49	0.973	26	76	0.951
0	25	0.972	52	54	0.971
5	25	0.972			

Como podemos ver los coeficientes de relación de Pearson son muy cercanos a 1 en valor absoluto. Creo que por este motivo puede ser beneficioso reducir variables ya que lo que nos puede aportar una característica va a ser casi igual a lo que nos aporte otra que tenga un coeficiente de correlación tan cercano a uno.

Por tanto elimino las características correspondientes a las columnas:

0,2,5,6,7,11,12,15,17,22,26,25,27,33,37,47,52,57,67,69,71,72 y 77.

Hemos eliminado 23 atributos y nos quedan 58, un número que me parece demasiado grande todavía.

Vamos a estudiar ahora la **variabilidad** en los datos.

Para ello he dibujado un *boxplot* o diagrama de caja, que representa los cuartiles de las variables y nos muestra a simple vista los valores atípicos y la dispersión de los valores de las características. Esto lo he hecho con la función *boxplot* del data frame de pandas. La gráfica es:

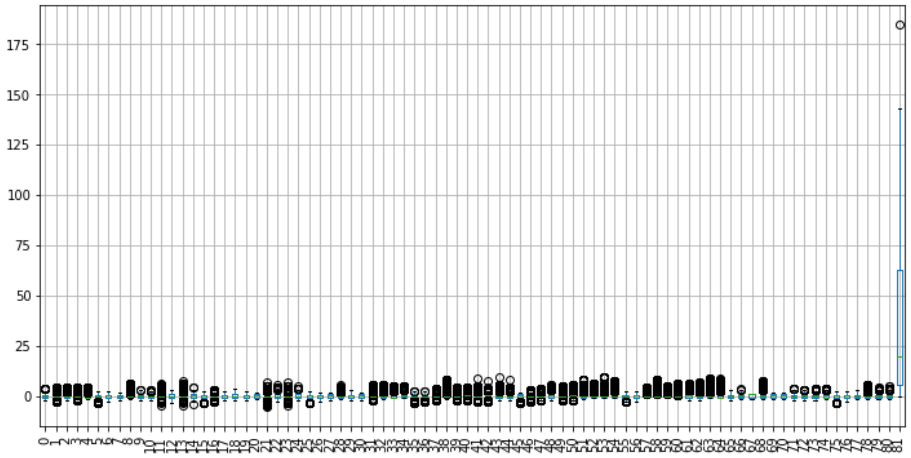


Figura 3: Diagrama de caja

Podemos ver que la columna 81, correspondiente a las etiquetas, tiene un valor atípico por encima de 175, lo que podría ser un error. Sin embargo, ya que

no conocemos las características propias de los superconductores, no podemos decir si ese dato es válido o no, para ello tendríamos que consultarlo con los expertos que hayan tomado la muestra de los datos. Las demás características no presentan valores atípicos, pero hay algunas que nos llaman la atención por su poca -o nula- variabilidad. Estas son las columnas 20,69,70. He decidido eliminar estas características ya que pienso que pueden contribuir muy poco a la varianza global, y por tanto nos van a aportar poca información respecto al problema que queremos resolver. La columna 69 la habíamos eliminado antes, así que eliminamos ahora la 20 y la 70.

No he utilizado PCA porque he considerado que el preprocesado de datos realizado ha sido suficiente para quitar las variables redundantes o que no aportaban información.

Tras el preprocesado de datos que acabo de explicar he vuelto a dibujar en 2D los datos para ver si podía obtener información nueva, pero al igual que anteriormente no ha dado resultado.

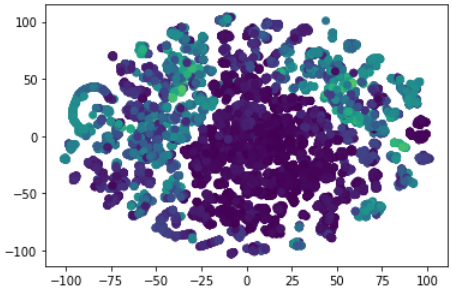


Figura 4: t-SNE tras reducir variables

Métrica de error a usar.

Para la selección del modelo final en *cross-validation* he elegido la métrica de R^2, es decir, el coeficiente de determinación, que es el cuadrado del coeficiente de correlación de Pearson, y determina la calidad del modelo para replicar los resultados así como la proporción de variación de los resultados que puede explicarse por el modelo.Toma valores entre 0 y 1, y el modelo será mejor cuanto más se acerque a 1.

Para evaluar el modelo final también he utilizado, además del coeficiente de determinación ya mencionado, el error cuadrático medio, que es un estimador que mide el promedio de los errores al cuadrado, es decir, la diferencia entre el estimador y lo que se estima. Lo he utilizado porque es un criterio de evaluación muy usado en problemas de regresión.

Selección de la mejor hipótesis.

Para seleccionar el mejor modelo para el problema he utilizado *cross-validation*.

Cross-validation es una técnica utilizada para evaluar los resultados de un determinado modelo y garantizar que son independientes de los datos de test. Los pasos son: se divide el conjunto de entrenamiento en k subconjuntos (lo recomendado es 5 ó 10, he elegido 5 porque los tiempos de ejecución eran menores), se realizan k iteraciones, en cada una de ellas se deja un subconjunto para evaluar y se entrena con los restantes. Por último se calcula la media del error en todas las iteraciones y nos quedamos con el modelo que tenga mejor media. De esta forma conseguimos que la elección del modelo sea más independiente de la partición elegida, sin hacer *data snooping*, es decir, sin utilizar los datos de test

En mi caso he recorrido todos los modelos y he utilizado la función *cross_val_score*, que devuelve un array con los errores obtenidos en un modelo determinado. He calculado la media de los errores de cada modelo y me he quedado con el modelo que mejor media tuviese.

Después de tener el modelo ya elegido he entrenado toda la muestra de train con dicho modelo, ya que los resultados que podemos obtener serán mejores cuanto más grande sea el conjunto de entrenamiento, por tanto en principio deberían ser mejores que los obtenidos en *cross-validation*. Por último he hecho predicción de los valores de las etiquetas en test y entrenamiento y he calculado los errores.

Los resultados que he obtenido han sido:

- Modelo seleccionado:

  Regresión lineal con función de pérdida *squared_loss*, learning rate *adaptive*, regularización *l1* y coeficiente de regularización 0.0001.

  Vemos que concuerda con lo que en principio habíamos pensado que tenía sentido, ya que la función de pérdida *squared_loss* tiene menos tolerancia a los errores que *SVR*. Además el coeficiente de regularización es pequeño, como habíamos supuesto que sería ya que nuestro modelo tiene poca tendencia al sobreajuste.

- Coeficiente de determinación:

  Coeficiente de determinación en train: 0.709

  Coeficiente de determinación en test: 0.713

- Error cuadrático medio:

  Error cuadrático medio en train: 342.675

  Error cuadrático medio en test: 330.337

Podemos observar que los errores en train y test son muy similares, ligeramente superior el de entrenamiento. Esto nos dice que no hay sobreajuste en el modelo, es decir, no intenta explicar el error estocástico, haciendo que E_{in} se vaya a cero mientras que E_{out} crece mucho.

Otras consideraciones

He realizado dos experimentos más: uno orientado a saber si hemos reducido de forma correcta los atributos que nos proporcionaban, y otro a saber si los errores que hemos obtenido podemos pensar que son lo suficientemente buenos o no.

Para el primero he vuelto vuelto a entrenar el modelo, esta vez con la matriz de datos original -con los datos normalizados- y he calculado los errores obtenidos. Los resultados han sido:

- Coeficiente de determinación en train: 0.7355
- Coeficiente de determinación en test: 0.7356

Podemos comprobar que hemos obtenido resultados muy parecidos, pero algo mejores que los obtenidos con la muestra tras el preprocesado de datos. Creo que esto es normal, ya que al tener más datos es normal que obtengamos mejores resultados, pero al ser una diferencia tan pequeña me lleva a pensar que la reducción de variables ha sido correcta. En un caso real utilizaría la primera opción si el número de variables es demasiado grande, y si eso puede afectar seriamente al tiempo de ejecución. Si no tenemos prisa por obtener los resultados y el tiempo de ejecución es razonable quizás utilizaría la segunda opción, ya que los resultados son algo mejores.

Para el segundo experimento he utilizado un modelo *naive*, que asigna valores aleatorios entre el mínimo y el máximo de las etiquetas. El resultado ha sido:

- Error cuadrático medio: 7289.733

De esta forma podemos comprobar que, aunque los resultados obtenidos con mi modelo no son increíblemente buenos, si es un avance positivo respecto a lo que tendríamos si utilizáramos un estimador aleatorio, es decir, sin usar técnicas de aprendizaje automático.

Clasificación

Para este problema utilizamos la base de datos *Superconductivty Data Data Set* encontrada en https://archive.ics.uci.edu/ml/datasets/Dataset+for+Sensorless+Drive+Diagnosis.

Comprensión del problema a resolver. Identificación de los elementos X, Yyf.

El problema que queremos resolver es dado un motor, asignarlo a una de las once clases que tenemos. Cada motor tiene unas características como DUDA. En concreto tenemos datos sobre 58509 motores, y de cada uno de ellos tenemos 49 características.

Para comprender mejor el problema vamos a visualizar los datos. Para ello aplicaremos el algoritmo t-SNE. En este caso no hace falta aplicar antes PCA ya que tenemos un número de atributos inferior a 50 (tenemos 49). t-SNE intenta reproducir la distribución que existe en el espacio original en otro espacio de dimensión menor, en este caso de dimensión 2 para que podamos visualizarlo. Al contrario que PCA, que simplemente maximiza la varianza, t-SNE hace que puntos con características parecidas queden cerca en el modelo final, y los que menos se parecen queden alejados.

Adjunto el gráfico obtenido con t-SNE, se puede ejecutar en el código pero recomiendo no hacerlo, ya que tarda mucho. He intentado cambiar algunos parámetros para que vaya más rápido pero no ha funcionado ninguno, como explico más abajo.

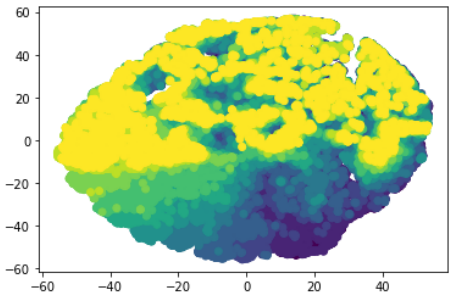


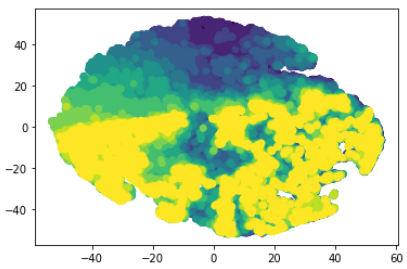
Figura 5: t-SNE parámetros por defecto

t-SNE admite algunos parámetros tales como:

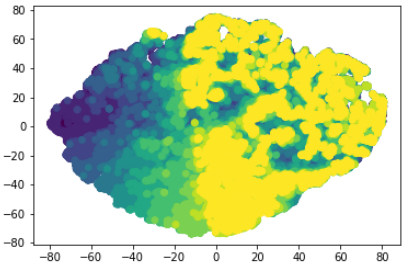
- *perplexity* : valor entre 5 y 50 (mayor cuanto mayor sea el dataset). Por defecto vale 30.
- *early_exaggeration*: Este parámetro controla la distancia entre bloques semejantes en el espacio final. La elección de este valor no es crítico. Por defecto vale 12.

- *learning_rate*: Habitualmente en el rango (10-1000). Si es muy elevado, los datos transformados estarán formados por una bola de puntos equidistantes unos de otros. Si es muy bajo, los puntos se mostrarán comprimidos en una densa nube con algunos outliers. Poe defecto vale 200.
- *n_iter*: Número máximo de iteraciones para la optimización. Debería ser, por lo menos, 250. Por defecto vale 1000.
- *metric*: métrica para la medición de las distancias.
- *method*: algoritmo a usar para el cálculo del gradiente.

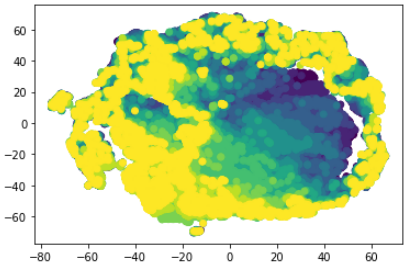
Vamos a cambiar algunos valores de los parámetros, para ver cómo influye en el resultado final y cuales de ellos se ajustan mejor a nuestro modelo.



(a) perplexity = 45



(b) learning rate = 400



(c) early-exaggeration=30

No he incluido estas gráficas en el código porque tardan mucho tiempo en ejecutar, pero las menciono aquí a modo de comentario.

Vemos que no tenemos diferencias muy significativas al variar los parámetros, al menos no tenemos diferencias que nos añadan más información de la que disponemos. En cuanto al tiempo podemos ver que el método tarda mucho tiempo en ejecutarse, pero tampoco he podido disminuir ese tiempo al cambiar los parámetros.

Por último he comprobado que las clases estén proporcionadas, es decir, que haya un número parecido de elementos de cada clase en la muestra. Los resultados han sido:

[0.0909 0.0909 0.0909 0.0909 0.0909 0.0909 0.0909 0.0909 0.0909 0.0909 0.0909]

Es decir, hay exactamente el mismo número de elementos en cada clase, así que la muestra es válida.

En primer lugar pensé visualizar los datos para poder comprender mejor el problema a tratar, y ver s. Para ello realizaremos dos pasos: reducir el conjunto de datos a un número de dimensiones razonable con la técnica PCA y visualizar el conjunto de datos con la técnica t-SNE.

Selección de la clase de funciones a usar.

Primero vamos a ajustar un modelo lineal, ya que si obtenemos buenos resultados con él no hace falta probar con otros modelos más complejos.

Probaremos al principio sin hacer ninguna transformación de los valores observados, y si vemos que. . . .

MODELOS: PLA, PLA_POCKET, REGRESION LINEAL, REGRESION LOGISTICA

Identificación de las hipótesis finales que vamos a usar.

Partición en test y entrenamiento.

Dividimos el conjunto de datos en test y entrenamiento. Para ello utilizamos la función *train_test_split*, introduciendo la aleatoriedad y reservando un 20 % de los datos para test. He reservado una proporción bastante grande de los datos porque tenemos una muestra muy grande, así que los resultados que obtendremos con el 80 % de los datos serán presumiblemente buenos.

Comprobamos, igual que hemos hecho antes, que los datos están bien balanceados:

Entrenamiento: [0.0907 0.0904 0.0913 0.0912 0.0909 0.0911 0.0901 0.0917 0.0920 0.0907 0.09]

[0.09177918 0.09306102 0.08921552 0.08955734 0.09109554 0.09032644 0.09425739 0.08793369 0.08648094 0.09160827 0.09468467] ## Preprocesado de datos.

Justificación de la métrica de error a usar.

Justificación de los parámetros y del tipo de regularización usada.

Selección de la mejor hipótesis del problema.

Modelo final con todos los datos.