UNIVERSIDAD DE GRANADA

Práctica Final AA

AJUSTE DEL MEJOR MODELO

Yunhao Lin Pan
Victor Diaz Bustos

Universidad de Granada
España

15 de noviembre de 2021

Resumen

En este documento vamos a hacer una descripción y análisis de distintos modelos de aprendizaje automático aplicado sobre la base de datos *Boston Housing Data*, los cuales tratarán de predecir el valor medio de una vivienda dado un conjunto de características de esta. Los datos fueron recolectados en 1978 y están compuestos de 506 muestras con 14 diferentes valores tales como crimen per cápita, concentración de óxido nítrico, impuestos sobre la propiedad cada 10 mil dólares, etc, y por supuesto el precio medio de las viviendas(tantos miles de dólares).

Así, el objetivo será comparar las predicciones de los diferentes modelos, tras el correspondiente preprocesado de los datos y entrenamiento de estos modelos, explicando las diferencias entre estos y el motivo de los resultados finales.

Índice

1. Introducción

Esta práctica consiste en el ajuste y selección del mejor predictor para una base de datos mediante el uso de la librería Scikit-Learn (sklearn) de python, la cual contiene funciones de muy alto nivel que pueden llegar a ser muy útiles para el objetivo de esta práctica.
La base de datos que utilizaremos será housing.data obtenida del siguiente enlace `http://archive.ics.uci.edu/ml/machine-learning-databases/housing/` , la cual almacena diferentes datos sobre viviendas. El fichero con los datos obtenido de dicho enlace será ubicado en el directorio data/, el cuál se ubica en el mismo directorio que el archivo .py el cuál contiene el código.

Para probar el código escrito durante el desarrollo de esta práctica hay que, primero, descargar la base de datos del enlace previo, y ponerlo dentro de una carpeta data. El código se situará fuera de la carpeta data. En el árbol de directorio de a continuación se puede observar dónde debe estar cada archivo.

```
-- Directorio actual
|-- practicaFinal.py
|-- data
  |-- housing.data
  |-- housing.names(opcional, debido a que no es obligatorio estar presente, ya que
                    este documento tan solo explica el contenido de los datos)
```

2. Definición del problema y enfoque elegido

2.1. Entendiendo la base de datos

Para esta base de datos se nos plantean dos problemas, bien tratar de predecir el valor de óxido de nitrógeno (variable NOX) o bien el precio medio de una casa (MEDV). En nuestro caso trataremos de predecir el precio medio de una casa.

Dicha base de datos cuenta con 14 variables:

1. **CRIM**: tasa de delincuencia per cápita por ciudad.
2. **ZN**: proporción de terreno residencial dividido en zonas para lotes de más de 25,000 pies cuadrados.
3. **INDUS**: proporción de acres comerciales no minoristas por ciudad.
4. **CHAS**: Variable tonto de Charles River (= 1 si el tramo limita con el río; 0 en caso contrario).

5. **NOX**: concentración de óxidos de nitrógeno (partes por 10 millones).
6. **RM**: número medio de habitaciones por vivienda.
7. **AGE**: proporción de unidades ocupadas por sus propietarios construidas antes de 1940.
8. **DIS**: distancias ponderadas a cinco centros de empleo de Boston.
9. **RAD**: índice de accesibilidad a carreteras radiales.
10. **TAX**: tasa de impuesto a la propiedad de valor total por $10,000.
11. **PTRATIO**: ratio alumno-profesor por municipio.
12. **B**: $1000(Bk - 0{,}63)^2$ donde Bk es la proporción de negros por ciudad.
13. **LSTAT**: % menor estado de la población.
14. **MEDV**: Valor medio de las viviendas ocupadas por sus propietarios en $1000.

Para nuestro problema, utilizaremos la columna 14 del conjunto de datos, (MEDV), como etiqueta (Y), el cual trataremos de predecir, y el resto como datos (X):
f = modelo que para unas características de entrada nos da el precio de predecido de la vivienda
X = características tomadas por el creador de la BBDD y explicadas en 2.1.
Y = valor medio de la vivienda

2.2. Haciendo uso de la base de datos

Vamos a importar la base de datos a Python y lo primero que haremos va a ser una división aleatoria de los datos, reservando un 70 % de estos para el entrenamiento y el 30 % restante para la prueba. Para ello hacemos uso de la función train_test_split() perteneciente a sklearn.model_selection, la cual se encarga de hacer esta división.

Vamos a trabajar solamente con los datos de entrenamiento y en ningún momento se tocará los datos de prueba, exceptuando el caso donde le aplicamos la normalización usada en entrenamiento a este conjunto.

3. Codificación de los datos de entrada

En nuestra base de datos, no se va a proceder a realizar ninguna codificación de los datos, ya que vienen ya codificados. Por ejemplo la características CHAS que es la variable discreta que tenemos que podría requerir de codificación ya vienen codificadas con una codificación binaria.

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
count	354	354	354	354	354	354	354	354	354	354	354	354	354	354
mean	3.35	10.41	11.27	0.06	0.55	6.29	69.28	3.78	9.39	405.65	18.51	355.58	12.63	22.57
std	8.79	21.66	6.87	0.25	0.11	0.71	27.39	2.12	8.59	167.80	2.08	92.95	7.00	9.15
min	0.00	0.00	0.46	0.00	0.39	3.56	6.00	1.12	1.00	187.00	12.60	0.32	1.92	5.00
25 %	0.08	0.00	5.19	0.00	0.45	5.90	46.82	2.10	4.00	277.00	17.40	375.24	7.15	17.12
50 %	0.24	0.00	9.79	0.00	0.53	6.20	77.00	3.20	5.00	330.00	19.05	391.77	11.37	21.20
75 %	3.51	12.50	18.10	0.00	0.62	6.61	94.07	5.11	20.00	666.00	20.20	395.91	17.15	24.95
max	88.9	100.00	27.74	1.00	0.87	8.78	100.0	12.12	24.00	711.00	22.00	396.90	37.97	50.00

Cuadro 1: Estadística de los datos de entrenamiento

4. Valoración del interés de variables y selección de un subconjunto

En este tipo de problemas es muy común que se de un conjunto de datos con un número excesivamente grande de variables, lo que producirá que el modelo se ajuste demasiado provocando overfitting, por lo que en esos casos se suele hacer una valoración de dichas variables y comprobar cuáles son más relevantes y cuáles son descartables. También cabe la posibilidad de encontrarnos con un dataset con un número escaso de variables, dando lugar a un modelo mal ajustado que genere estimaciones sesgadas (underfitting), para lo cual se pueden hacer transformaciones para aumentar el número de variables del problema. En nuestro caso no vamos a hacer ni reducción de dimensionalidad ni aumento de ésta.

Estadística de los datos que tenemos en el conjunto de prueba En el cuadro 1 podemos observar que nuestros datos, aparte de otras métricas como la desviación típica o la media, tienen distintos rangos, e.g: TAX esta en el rango de 117-711 y CRIM está en el rango de 0-88.9. Esto es un indicativo de que tenemos que normalizar nuestros datos, cosa que explicaremos en el siguiente apartado. También observamos que nuestros datos no siguen una distribución de Gauss o también llamado normal, por lo tanto, normalizaremos en vez de estandarizar.

Tratamiento de outliers No se ha procesado los outliers por dos motivos. El primero, nuestros datos no siguen una distribución gaussiana como se puede observar en la figura 1. Esto no sería un problema si tuviéramos más datos ya que según el teorema del límite central, al tener más datos, ésta se aproxima a una distribución normal. Esto nos impide usar alguna de las técnicas más comunes de tratamiento de outliers como el Z-score o la detección de outliers haceindo uso de la distancia Mahalanobis. Y el segundo motivo es el tamaño de datos que tenemos, ya que al ser tan pequeña la base de datos que tenemos, y nuestro conjunto de entrenamiento es más pequeño aún al hacerle el split training-test, al eliminar outliers nos reduciría aún más los datos de entrenamiento.

Correlación entre las distintas características Observando el mapa de calor 2, se puede observar la existencia de algunas correlaciones entre las características. Por ejemplo, NOX y DIS tinen una correlación negativa bastante fuerte. En cuanto a los datos que queremos predecir, i.e. MEDV, se puede observar que hay un correlación negativa con LSTAT, es decir, a medida que esta aumenta, MEDV disminuye. También se observa una correlación positiva fuerte con la característica RM.

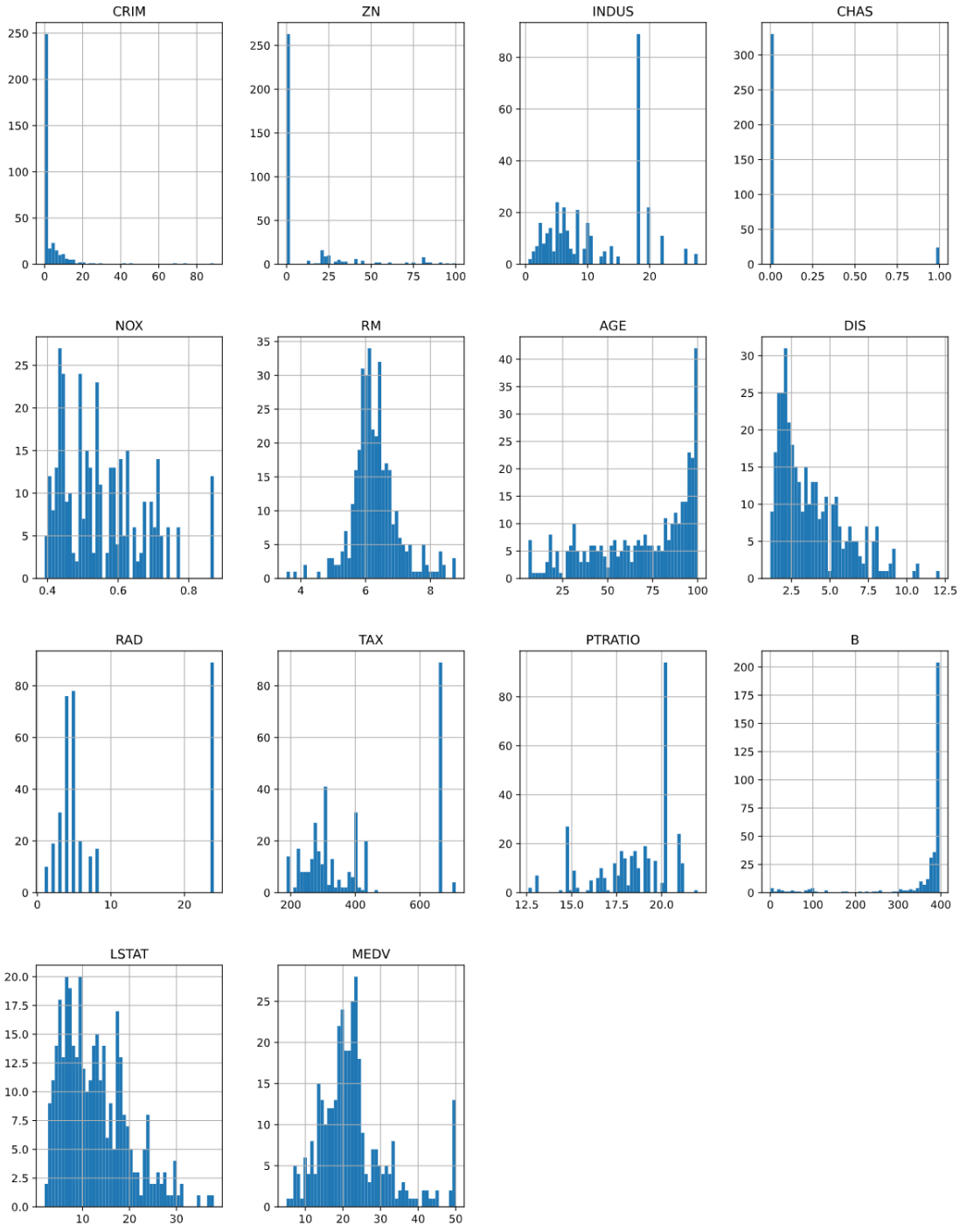

Figura 1: Distribución de los valores de las distintas características y la etiqueta

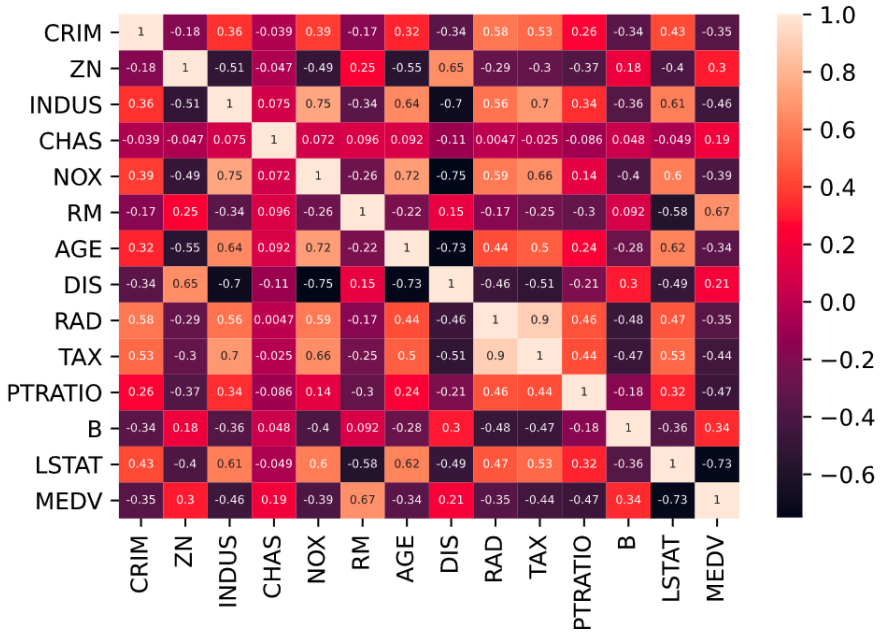

Figura 2: Mapa de calor que muestra las correlaciones de los datos

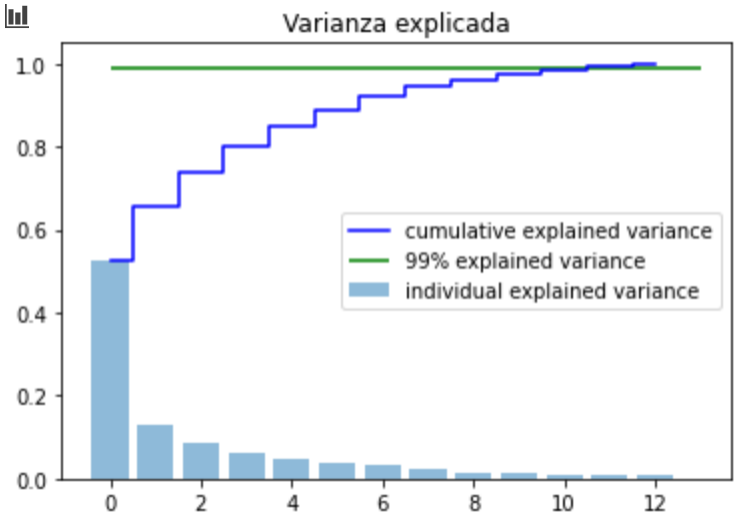

Figura 3: Explained variance ratio, con una explicación acumulada: 0.52592132, 0.6548698 , 0.7385515 , 0.80183015, 0.84995641, 0.88911852, 0.92287341, 0.94745396, 0.96187224, 0.97361202, 0.98369374, 0.99273313, 1. Al tener que usar casi todas las variables para lograr que explique la varianza al 99 % se ha optado por no reducir dimensionalidad

5. Normalización

Muchas veces los datos que se proporcionan para que los distintos algoritmos aprendan vienen definidos en distintos rangos de valores, e.g, el rango de los datos de una característica es [-1, 1] mientras que otra puede ser[-200, 150]. El objetivo de la normalización es transformar los datos (no las etiquetas) de forma que todos estén en la misma escala, en el rango [0,1], sin distorsionar las diferencias en el rango de los valores o perdiendo información. Para normalizar, me refiero a normalización y no a la otra técnica de estandarización que se resta el mínimo y se divide por el máximo menos el mínimo de ese feature:

$$X' = \frac{X - X_{min}}{X_{max} - Xmin}$$

Las ventajas de normalizar nuestros datos son varias, siendo una de ellas ayudar a la convergencia de nuestros modelos, ahorrándonos tiempo y recursos.

CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
3.67367	0.0	18.10	0.0	0.583	6.312	51.9	3.9917	24.0	666.0	20.2	388.62	10.58	21.2
0.09604	40.0	6.41	0.0	0.447	6.854	42.8	4.2673	4.0	254.0	17.6	396.90	2.98	32.0
3.53501	0.0	19.58	1.0	0.871	6.152	82.6	1.7455	5.0	403.0	14.7	88.01	15.02	15.6
0.06211	40.0	1.25	0.0	0.429	6.490	44.4	8.7921	1.0	335.0	19.7	396.90	5.98	22.9
0.14476	0.0	10.01	0.0	0.547	5.731	65.2	2.7592	6.0	432.0	17.8	391.50	13.61	19.3

Cuadro 2: Muestra de las primeras 5 muestras de los datos de entrenamiento, antes de aplicar la normalización

Normalización en nuestro caso Nosotros vamos a normalizar los datos, ya que nuestros datos no siguen una distribución gausiana y es más recomendable normalizar antes que estandarizar [9], haciendo uso de la función MinMax Scaler[8] que nos ofrece Scikit Learn.

CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0.041220	0.0	0.646628	0.0	0.398747	0.527112	0.488298	0.260264	1.0	0.914122	0.808511	0.979121	0.240222	21.2
0.001008	0.4	0.218109	0.0	0.114823	0.630964	0.391489	0.285326	0.130435	0.127863	0.531915	1.0	0.029404	32.0
0.039662	0.0	0.700880	1.0	1.0	0.496455	0.814894	0.056007	0.173913	0.412214	0.223404	0.221116	0.363384	15.6
0.000627	0.4	0.028959	0.0	0.077244	0.561219	0.408511	0.696787	0.0	0.282443	0.755319	1.0	0.112621	22.9
0.001556	0.0	0.350073	0.0	0.323591	0.415788	0.629787	0.148187	0.217391	0.467557	0.553191	0.986384	0.324272	19.3

Cuadro 3: Muestra de las primeras 5 muestras de los datos de entrenamiento, tras aplicar la normalización

6. Función de pérdida

Debido a que nuestro problema es del tipo regresión vamos a emplear el error cuadrático como la función de pérdida, el cual mide la distancia entre el valor predecido y el valor real de los datos al cuadrado, es decir, la diferencia entre el estimador y lo que se estima y se calcula como:

$$Error(x) = ((h_w(x) - f(x))^2$$

donde $h_w(x)$ es el valor predicho, y $f(x)$ es el valor real. Al tratarse de un error "cuadrático", como su propio nombre indica, se penalizan aún más aquellos errores que distan más del valor real de la

etiqueta, y menos aquellas predicciones que se acerquen a dicho valor real, lo que lo convierte en una buena medida de la función de pérdida en problemas de regresión, donde se trata de ajustar un modelo de forma que genere predicciones lo más cercanas posibles al valor real de las etiquetas.

7. Idoneidad de la regularización

7.1. Qué es la regularización y por qué es importante

La regularización consiste en añadir una penalización o sesgo a la función de pérdida forzando a que los coeficientes del modelo tiendan a cero con el objetivo de reducir la varianza final, i.e, que la diferencia entre nuestro modelo entrenado sobre el conjunto de prueba se comporte igualmente bien, evitando así que se produzca **overfitting**, definido como el efecto de sobreentrenar un modelo de manera que las predicciones con los datos fuera de la muestra de entrenamiento no sean buenas.

La regularización l_1 o **Lasso** consiste en la siguiente penalización: l_1 $penalty = \lambda \sum_{j=1}^{k} \mid w_j \mid$, donde λ determina el peso de la regularización, y k es el número total de pesos que tenemos.

Por el otro lado, la regularización l_2 o **Ridge** es muy similar con la diferencia de, en lugar de ser el valor absoluto, es el cuadrado de los pesos. l_2 $penalty = \lambda \sum_{j=1}^{k} w_j^2$, donde λ determina el peso de la regularización, y k es el número total de pesos que tenemos.

Viendo esto, podemos ver si $\lambda = 0$ es como si la regularización no afectase y tendríamos el modelo original sin modificar, y cuanto más se incrementa λ, más peso tiene y esto nos puede llevar a **underfitting**, es decir, a un modelo demasiado simple incapaz de generar buenas predicciones.

En la práctica, la diferencia clave entre ambas regularizaciones es que, Lasso es capaz de reducir las características menos importantes a cero, mientras Ridge lo reduce también pero nunca llega a cero. Es por esto, que Lasso pueda ser utilizado tambíen para reducir la dimensionalidad del problema.

7.2. Bias-Variance Tradeoff

En modelos sin penalización, podemos encontrarnos con que nuestro modelo funciona muy bien en nuestro entrenamiento, pero sobre el conjunto de prueba funcione muy diferente (overfitting), por lo tanto ese modelo tiene una alta varianza. Pero y ¿si le introducimos un poco de sesgo? Hay casos en los que aumentar el sesgo nos podría reducir la varianza, que es algo beneficioso. Incrementar demasiado el sesgo produce que nuestro modelo infra-ajuste, mientras que tener una alta varianza hace que nuestro modelo se sobre-ajuste, tal como se puede verse refleado en la gráfica 9. Por lo tanto el trueque consiste en encontrar el punto dulce donde ambas variables, el sesgo y la varianza, sean la mínima. Esto es lo que se conoce como Bias-Variance Tradeoff, que trata de encontrar un equilibrio en el modelo de manera que se minimice tanto el sesgo como la varianza.

Definimos varianza y sesgo como:

- **Varianza** (variance): Es la media de la variabilidad dado un punto entre nuestra predicción y su valor real, calculada como $var(x) = E_D[(g^D(x) - \overline{g}(x))^2]$, donde $g^D(x)$ es un valor aleatorio en función de D y $\overline{g}(x) = E_D[g^D(x)]$ es la predicción promedio en x(esperanza). Por lo tanto, la varianza se podría escribir también como $var(x) = E_D[(g^D(x)] - \overline{g}(x))^2$ Un valor demasiado alto de la varianza indica que pequeños cambios en los datos producirán grandes cambios en el resultado (overfitting), mientras que una varianza pequeña permitirá modificar los datos sin que se produzcan grandes cambios en la estimación (buen ajuste).

- **Sesgo** (bias): Es la diferencia entre el valor de una predicción y su valor real, calculado como $bias(x) = (\overline{g}(x) - f(x))^2$. Un sesgo muy elevado producirá un modelo muy simple que no se ha ajustado lo suficiente a los datos de entrenamiento, por lo que tendremos un error elevado en las muestras de entrenamiento, validación y test (underfitting).

Así, lo óptimo será tener un modelo que tenga poco sesgo y poca varianza.

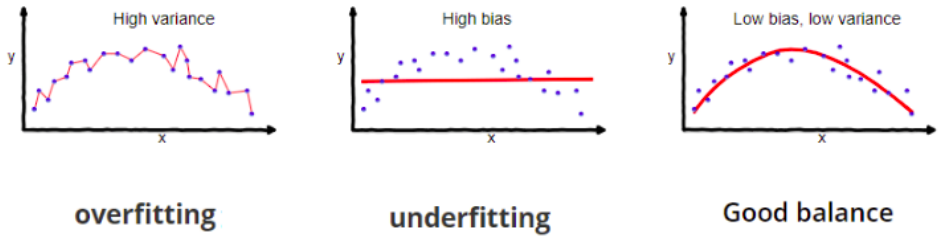

Figura 4: Influencia del sesgo y la varianza en el aprendizaje [3]

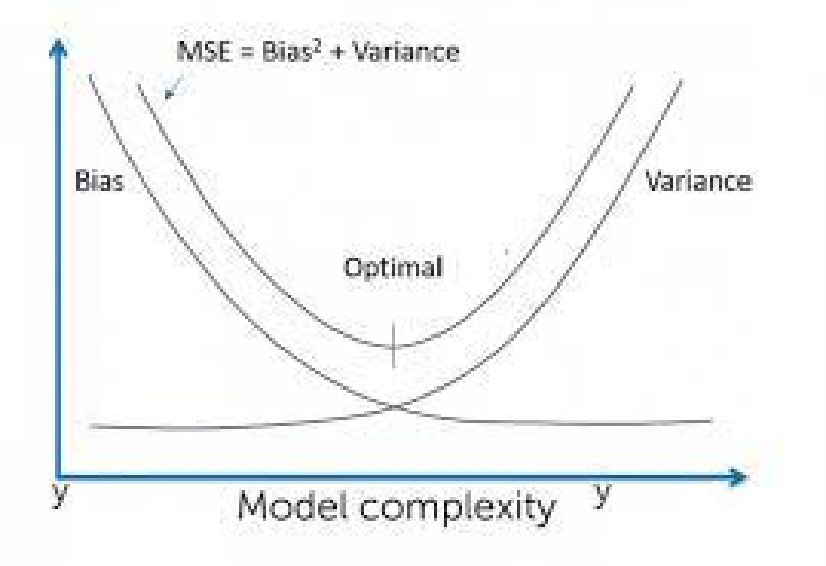

Figura 5: Relación sesgo, varianza y complexidad del modelo [3]

7.3. Aplicación a nuestro problema

En nuestra selección de la mejor hipótesis, haremos uso de la regularización Ridge o l_2 en el modelo de regresión lineal y en el del perceptrón multicapa, y no en random forest, ya que, como explicaremos posteriormente, la propia naturaleza del algoritmo ya solventa el problema del overfitting.

8. Algoritmos de aprendizaje utilizados

Para este problema vamos a comparar 3 modelos de regresión: **Perceptrón Multicapa** (MLP - Multi Layer Perceptron), **Random Forest** y **Regresión Lineal con penalización Ridge**, implementados con las funciones de sklearn neural_network.MLPRegressor(), ensemble.RandomForestRegressor() y linear_model.Ridge(), respectivamente.

8.1. Descripción de los algoritmos

8.1.1. Regresión lineal

La regresión lineal trata de modelizar la relación entre dos variables ajustando una ecuación lineal a los datos observados. La métrica del error empleado es el error cuadrático medio (MSE). Por lo tanto, nuestro objetivo será minimizar ese error: El error a minimizar se puede escribir como la suma de la diferencia entre la predicción y el valor real al cuadrado dividido entre el total de las muestras:

$$E_{in} = \frac{1}{N} \sum \left(w^T X_n - y_n \right)^2,$$

donde w^T es el vector de pesos que buscamos, X_n son los features y y_n son las etiquetas. Esto, si nos fijamos, es minimizar la función de pérdida, tal como se explica en el apartado 6, en todo el conjunto de datos.

Una vez minimizado el error podemos usar el vector de pesos calculado, i.e. w_T, para calcular predicciones dado unas características de entrada.

$$y_{pred} = w_0 * x_0 + w_1 * x_1 + ... + w_n x_n,$$

siendo n el total de características, y x_0 vale 1, por lo que normalmente no se pone.

8.1.2. Multi Layer Perceptron

El algoritmo Multi Layer Perceptron (MLP) consiste en combinar varios perceptrones simples, generando así lo que se conoce como una red neuronal y logrando representar funciones no lineales, por lo que, para poder entender bien este algoritmo, vamos a explicar en qué consiste el algoritmo Perceptrón (PLA).

El algoritmo **Perceptrón** es una red de una sola capa, la cual trata de encontrar un hiperplano capaz de dividir un conjunto de datos linealmente separables, lo que lo convierte en un buen modelo de clasificación binaria. Para ello, el algoritmo trata de encontrar función h capaz de predecir correctamente la etiqueta y_i para cada dato x_i, donde h(x_i) = sign(w*x), donde w*x es la ecuación del hiperplano definida por dos vectores (w y x), de los cuales el único desconocido es el vector de pesos w. Por lo que el objetivo de este algoritmo será encontrar el vector w óptimo para el hiperplano capaz de separar correctamente los datos.

Para ello, PLA comienza el proceso con un hiperplano cualquiera definido por el vector aleatorio w, clasifica los datos de acuerdo a dicho hiperplano y actualiza el vector de pesos w como w = w + $x_i * y_i$, y repite la clasificación y actualización hasta que el algoritmo logre clasificar los datos correctamente

Por su parte, el algoritmo **MLP** genera varios hiperplanos, dando lugar a un modelo apto tanto para clasificación multiclase como para regresión. Dicho algoritmo está compuesto por: una capa de entrada, donde cada una de las neuronas corresponde con cada una de las variables de entrada de la red; una de salida, la cual proporciona los resultados de la red; y n capas ocultas intermedias, las cuales realizan un procesamiento no lineal de los datos recibidos. De esta forma, tanto el número de neuronas

en la capa de entrada, como el número de neuronas en la capa de salida, vienen dados por el número de variables definidas en el problema. Aunque en problemas con un gran número de variables suele ser conveniente realizar un análisis previo de las variables de entrada para ver cuáles son más relevantes y cuales se pueden descartar porque no aporten información a la red. En nuestro caso esto no será necesario ya que contamos únicamente con 14 características en nuestro problema. Por su parte, tanto para el número de capas ocultas, como para el número de neuronas en estas capas, no existe un método que determine la cantidad óptima de estas, por lo que se suelen determinar por prueba y error. En nuestro caso, nuestra red tendrá únicamente 2 capas ocultas, tal y como se especifica en el guión, y el número de neuronas en estas se determinará tras una búsqueda exhaustiva de estas.

Cada capa está conectada con la capa siguiente, lo que se conoce como "feedforward", donde normalmente cada neurona de una capa está conectada con todas las neuronas de la capa siguiente, en este caso se dice que la red está totalmente conectada. Además, las conexiones entre las neuronas llevan asociado un **umbral**, que en este este algoritmo suele tratarse como una conexión más a la neurona, cuya entrada es constante e igual a 1.

De esta forma, se define una relación entre las variables de entrada y las de salida de la red, la cuál se obtiene propagando hacia adelante los valores de las variables de entrada. Para ello, cada neurona procesa la información recibida por sus entradas y produce una respuesta o **activación** que se propaga hacia las neuronas de la siguiente capa.

Para finalizar, vamos a ver como se calcula la activación de las neuronas. Sea un Perceptrón Multicapa con C capas (C-2 capas ocultas), $W^c = (w_{ij}^c)$ la matriz de pesos, donde w_{ij}^c representa el peso de la conexión de la neurona i de la capa c para c = 2,...,C. Denotaremos a_i^c la activación de la neurona i de la capa c. Estas activaciones se calculan de la siguiente forma:

- Activación de las neuronas de la capa de entrada (a_i^1). En esta capa, las neuronas se encargan de transmitir a la red los datos recibidos desde el exterior de la red. De esta forma, $a_i^1 = x_i$ para $i = 1, ..., n_1$, donde X = $(x_1, ..., x_{n_1})$ es el vector de entrada de la red y n_1 es el número de neuronas en la capa de entrada.

- Activación de las neuronas de la capa oculta n (a_i^c). En las capas ocultas, las neuronas procesan la información recibida aplicando la función de activación f a la suma de los productos de las activaciones que recibe por sus correspondientes pesos, es decir: $a_i^c = f(\sum_{j=1}^{n_{c-1}} w_{ji}^{c-1} a_j^{c-1} + u_i^c)$ para $i = 1, ..., n_c$ y $c = 2, ..., C-1$, donde a_j^{c-1} son las activaciones de las neuronas de la capa $c-1$ y n_c es el número de neuronas en la capa c.

- Activación de las neuronas de la capa de salida (a_i^C). En la capa de salida, la activación de las neuronas viene dada por la función de activación f aplicada a la suma de los productos de las entradas que recibe por sus correspondientes pesos, al igual que en las capas ocultas. Por tanto, $y_i = a_i^C = f(\sum_{j=1}^{n_{c-1}} w_{ji}^{c-1} a_j^{c-1} + u_i^c)$ para $i = 1, ..., n_C$, donde Y = $(y_1, ...y_{n_C})$ es el vector de salida de la red y n_C es el número de neuronas en la capa de salida.

Función de activación de la capas ocultas Nosotros haremos uso de Rectified Linear Unit o ReLU. Esta función de activación se ha vuelto bastante popular últimamente debido a que es computacionalmente más eficiente. Esta eficiencia se debe a que la neurona solamente se activa cuando la salida es menos de 0. Tal como se puede ver en la figura 6.

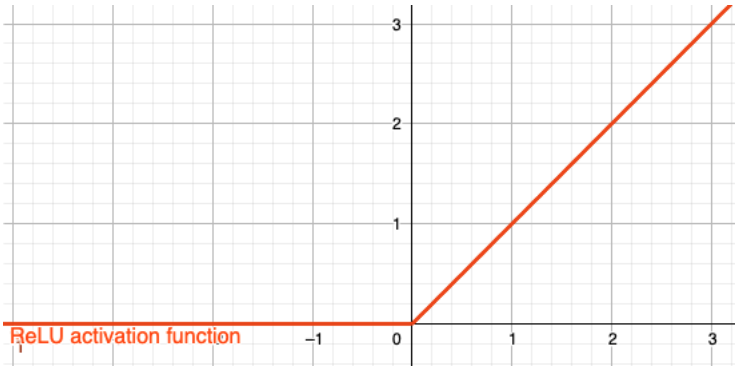

Figura 6: Función de activación ReLU

8.1.3. Random Forest

El algoritmo de Random Forest consiste en generar aleatoriamente múltiples árboles de decisión y los combina para obtener una predicción precisa y estable. Mientras más árboles se generen, más robusto será el bosque, es decir, se generará un modelo que se adaptará mejor a modificaciones en los datos de entrada. Además, añade aleatoriedad al expandir los árboles, pues, en lugar de buscar la característica más importante al dividir un nodo, busca la mejor característica entre un subconjunto aleatorio de características, dando como resultado una gran diversidad que generalmente resulta en un mejor modelo.

De esta manera, Random Forest evita que se sufra overfitting, frente a árboles de decisión muy profundos, que tienden a un sobreajuste, por lo que no aplicaremos regularización en este modelo. Este algoritmo crea subconjuntos aleatorios de las características, así como árboles de decisión más pequeños utilizando estos subconjuntos y los combina para obtener un resultado final, aunque un número demasiado elevado de estos árboles puede provocar que el algoritmo sea lento y no logre ajustarse lo suficiente (underfitting).

9. Idoneidad de los modelos seleccionados

9.1. Regresión Lineal

Este algoritmo es algo limitado para este problema, ya que solo logra un buen ajuste para conjuntos de datos que son linealmente separables. Por ello, como veremos posteriormente, es el que genera peores predicciones, aunque nos será útil para comparar modelos lineales contra no lineales. También tiene la ventaja de ser uno de los modelos que más fáciles de entender y que funciona bien para cualquier tamaño de datos.

9.2. Multi Layer Perceptron

Este modelo tiene la ventaja de ser muy flexible y adaptarse muy bien a la no linealidad de nuestro problema, ya que las neuronas aprenden de los datos de entrada y nos ofrecen un modelo predictivo bastante robusto.

9.3. Random Forest

La principal ventaja del algoritmo Random Forest es que es muy útil y fácil de usar, sus parámetros predeterminados a menudo producen una buena predicción, además de que es complicado dar con un hiper-parámetro que produzca un mal ajuste de este modelo. No obstante, este algoritmo ofrece mejores resultados para problemas de clasificación que de regresión, ya que en estos casos no puede predecir más allá del rango de valores del conjunto de entrenamiento. A pesar de esto, como veremos adelante, ofrece muy buenas predicciones para este problema de regresión.

10. Selección de mejor hipótesis

Una vez hemos decidido los modelos que vamos a usar, a continuación lo que tenemos que hacer es, del conjunto de hipótesis, escoger aquella que mejor se comporte. ¿Cómo vamos a hacer esta selección? Con **cross-validation**. Para hacer cross validation, lo primero que haremos será dividir nuestro conjunto de entranamiento en k bloques. Se escoge k-1 bloques y se entrena una hipótesis con esos datos, una vez entrenado, ajustamos ese modelo sobre el bloque que hemos dejado fuera, y calculamos su puntuación o error. Repetimos el proceso hasta que todos los bloques hayan sido el conjunto de validación.

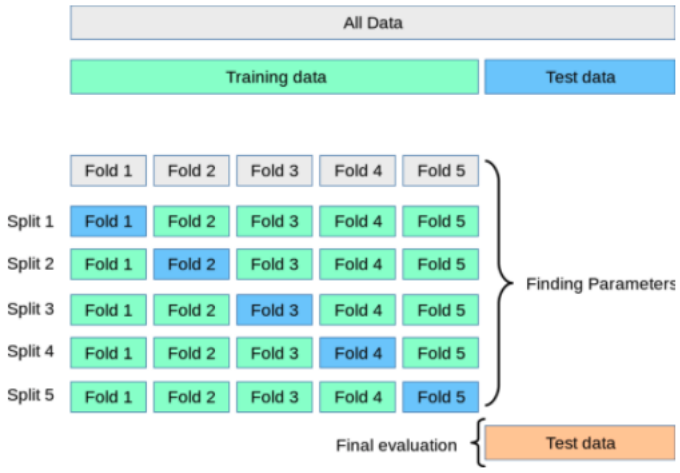

Figura 7: Gráfico que refleja cómo se divide las bloques en cross validation[12]

En nuestro caso hacemos uso de 5-Fold cross validation para escoger los mejores parámetros para los modelo que queremos usar, y haremos uso de RMSE como la métrica de evaluación. Se ha decidido por 5 bloques ya es menos intenso computacionalmente.

10.1. Búsqueda del mejor hiper-parámetro, y explicación de los parámetros disponibles en sklearn

10.1.1. Regresión Lineal

Los parámetros de regresión lineal con penalización l2, Ridge() se definen de la siguiente forma:

- **alpha : {float, ndarray of shape (n_targets,)}, default=1.0**: Es el parámetro de penalización (término de regularización), cuanto mayor sea alpha, mayor será la regularización, evitando así un overfitting (modelo demasiado ajustado, generando errores en predicciones fuera de los datos de entrenamiento) ya que reduce la varianza, aunque si alpha es demasiado alto se producirá un underfitting (modelo insuficientemente ajustado). Si se pasa una matriz, se supone que las penalizaciones son específicas de los objetivos.
- **fit_intercept : bool, default=True**: Indica si ajustar la intersección para este modelo. Si se establece en False, no se utilizará ninguna intersección en los cálculos.
- **normalize : bool, default=False**: Indica si se normalizarán los regresores X antes de la regresión. En caso de fit_intercept=False, se ignorará.
- **copy_X : bool, default=True**: Indica si X se copiará (True) o si puede sobreescribirse (False).
- **max_iter : int, default=None**: Es el número máximo de iteraciones. El solucionador itera hasta la convergencia (determinada por 'tol') o este número de iteraciones. Un valor demasiado pequeño haría que el modelo dejase de entrenar antes de alcanzar un buen ajuste, y un valor demasiado grande que el entrenamiento haga iteraciones innecesarias en caso de que esta sea la única condición de parada.
- **tol : float, default=1e-3**: Es la precisión de la solución. Un valor demasiado grande podría provocar un underfitting en el modelo, mientras que, uno demasiado pequeño, podría hacer que el tiempo de entrenamiento aumentase exponencialmente.
- **solver : {'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga'}, default='auto'**: Es el solucionador a utilizar, y puede ser: 'auto', elige el solucionador automáticamente según el tipo de datos; 'svd', usa una descomposición de valores singulares de X para calcular los coeficientes de cresta y es más estable para matrices singulares que 'cholesky'; 'cholesky', usa la función estándar scipy.linalg.solve para obtener una solución de forma cerrada; 'sparse_cg', usa el solucionador de gradiente conjugado que se encuentra en scipy.sparse.linalg.cg, como algoritmo iterativo, este solucionador es más apropiado que 'cholesky' para datos a gran escala, pues nos da laposibilidad de configurar tol y max_iter; 'lsqr', usa la rutina dedicada de mínimos cuadrados regularizados scipy.sparse.linalg.lsqr, es el más rápido y utiliza un procedimiento iterativo; 'sag', usa un descenso de gradiente medio estocástico; y 'saga' usa su versión mejorada e imparcial llamada SAGA, ambos métodos también utilizan un procedimiento iterativo y, a menudo, son más rápidos que otros solucionadores cuando tenemos un gran conjunto de datos con muchas variables.
- **random_state : int, RandomState instance, default=None**: Determina la generación de números aleatorios para mezclar datos cuando solver='sag' o 'saga';

Hemos implementado un grid search, o búsqueda exhaustiva, para seleccionar el mejor regulador, así como el mejor valor para los parámetros **alpha** ([9,8,7, 6, 5, 3, 2, 1, 0.1, 0.01]), si tenemos regularización para ver la intensidad de ésta; **max_iter** ([10000, 100000]) para ver si más iteraciones significa mejor resultado; **fit_intercept** ([True, False]) para ver si calcular w_0 o no calcular ésta influye en la predicción; **solver** (['svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga']), ya que dependiendo del algoritmo empleado para la implementación de la regresión logística pueden converger antes y/o encontrar mejores óptimos locales o el óptimo global y **tol** ([1e-3, 1e-4, 1e-5]). Para el caso de utilizar el regulador

'ridge'; **alpha** ([1, 0.1, 0.01, 0.05, 0.0010]), **max_iter** ([10000, 100000]), **fit_intercept** ([True, False]), **selection** (['random', 'cyclic']) y **tol** ([1e-3, 1e-4, 1e-5]) para el caso de **'lasso'**; y **fit_intercept** ([True, False]), **normalize ([False])**. Se ha usado el parámetro **n_jobs** = -1 para utilizar todos los núcleos del ordenador.

El resto de parámetros los mantenemos en su valor predeterminado ya en este problema no nos interesa profundizar tanto en el algoritmo.

Además, Ridge() cuenta con los siguientes atributos:

- **coef_ : ndarray of shape (n_features,) or (n_targets, n_features)**: Vector de pesos.
- **intercept_ : float or ndarray of shape (n_targets,)**: Término independiente en función de decisión. Si fit_intercept=False, valdrá 0.
- **n_iter_ : None or ndarray of shape (n_targets,)**: Número actual de iteraciones para cada objetivo. Si solver es distinto de 'sag' y 'lsqr', devolverá None.

10.1.2. Multi Layer Perceptron

Los parámetros de MLPRegressor() se definen de la siguiente forma:

- **hidden_layer_sizes : tuple, length = n_layers - 2, default=(100,)**: Representa el número de capas ocultas y el número de neuronas por capa, donde el i-ésimo elemento representa el número de neuronas en la i-ésima capa oculta.
- **activation : {'identity', 'logistic', 'tanh', 'relu'}, default='relu'**: Es la función de activación para la capa oculta. Puede tomar los siguientes valores: 'identity', activación sin operación, es útil para implementar un cuello de botella lineal, devuelve f(x) = x. 'logistic', es la función sigmoidea logística, devuelve f(x) = 1 / (1 + exp (-x)). 'tanh', es la función tangente hiperbólica, devuelve f (x) = tanh (x). 'relu', es la función de unidad lineal rectificada, devuelve f (x) = max (0, x). Nosotros haremos uso de la función de activación ReLU.
- **solver : {'lbfgs', 'sgd', 'adam'}**, default='adam': Es el tipo solucionador para la optimización del peso. Los tipos de solucionador son: 'lbfgs', es un optimizador de la familia de métodos quasi-Newton, nosotros emplearemos este solo. 'sgd', se refiere al descenso de gradiente estocástico. 'adam', se refiere a un optimizador estocástico basado en gradientes propuesto por Kingma, Diederik y Jimmy Ba.
- **alpha : float, default=0.0001**: Es el parámetro de penalización L2 (término de regularización), cuanto mayor sea alpha, mayor será la regularización, evitando así un overfitting (modelo demasiado ajustado, generando errores en predicciones fuera de los datos de entrenamiento) ya que reduce la varianza, aunque si alpha es demasiado alto se producirá un underfitting (modelo insuficientemente ajustado).
- **batch_size : int, default='auto'**: Indica el tamaño de los minibatches para optimizadores estocásticos. Si el solucionador es 'lbfgs', el clasificador no usará minibatch. Cuando se establece en 'auto',batch_size=min(200, n_samples)
- **learning_rate : {'constant', 'invscaling', 'adaptive'}, default='constant'**: Indica cómo varía la tasa de aprendizaje. Si es 'constant', es una tasa de aprendizaje constante dada por 'learning_rate_init'. Si es 'invscaling', disminuye gradualmente la tasa de aprendizaje learning_rate en cada paso de tiempo 't' usando un exponente de escala inversa de 'power_t'. tasa_de_aprendizaje_efectiva = tasa_de_aprendizaje_init / pow (t, potencia_t). Si es 'adaptive' mantiene la tasa de aprendizaje constante a 'learning_rate_init' siempre que la pérdida de entrenamiento siga disminuyendo. Cada vez que dos épocas consecutivas no reducen la pérdida de entrenamiento en al menos tol, o no aumentan la puntuación de validación al menos en tol si está activado 'Early_stopping', la tasa de aprendizaje actual se divide por 5. Solo se usa cuando solver = 'sgd'.

- **learning_rate_init : double, default=0.001**: Es la tasa de aprendizaje inicial utilizada. Controla el tamaño del paso al actualizar los pesos, cuanto mayor sea, mayor será el paso en dicha actualización. Un valor demasiado pequeño haría que el modelo convergiese demasiado despacio hacia un buen ajuste, y uno demasiado grande haría que el modelo diese pasos demasiado grandes en la actualización de los pesos sin llegar a encontrar un óptimo. Solo se usa cuando solver = 'sgd' o 'adam'.
- **power_t : double, default=0.5**: Es el exponente de la tasa de aprendizaje de escala inversa. Se utiliza para actualizar la tasa de aprendizaje efectiva cuando learning_rate se establece en 'invscaling' e indica como de rápido desciende la tasa de aprendizaje. Solo se usa cuando solver = 'sgd'.
- **max_iter : int, default = 200**: Número máximo de iteraciones. El solucionador itera hasta la convergencia (determinada por 'tol') o este número de iteraciones. Un valor demasiado pequeño haría que el modelo dejase de entrenar antes de alcanzar un buen ajuste, y un valor demasiado grande que el entrenamiento haga iteraciones innecesarias en caso de que esta sea la única condición de parada. Para los solucionadores estocásticos ('sgd', 'adam'), debemos tener en cuenta que esto determina el número de épocas (cuántas veces se utilizará cada punto de datos), no el número de pasos de gradiente.
- **shuffle : bool, default = True**: Indica si queremos o no mezclar muestras en cada iteración. Solo se usa cuando solver = 'sgd' o 'adam'.
- **random_state : int, RandomState instance**, default=None: Determina la generación de números aleatorios para las ponderaciones y la inicialización de sesgo.
- **tol : float, default = 1e-4**: Tolerancia para la optimización. Cuando la pérdida o la puntuación no mejoran al menos tol en n_iter_no_change iteraciones consecutivas, a menos que learning_rate se establezca en "adaptative", se considera que se ha alcanzado la convergencia y se detiene el entrenamiento. Un valor demasiado grande podría provocar un underfitting en el modelo, mientras que, uno demasiado pequeño, podría hacer que el tiempo de entrenamiento aumentase exponencialmente.
- **verbose : bool, default = False**: Indica si imprimir o no mensajes de progreso en stdout.
- **warm_start : bool, default = False**: Cuando se establece en True, reutiliza la solución de la llamada anterior para que encaje como inicialización; de lo contrario, simplemente borra la solución anterior.
- **early_stopping : bool, default = False**: Indica si se debe utilizar la parada anticipada para finalizar el entrenamiento cuando la puntuación de validación no mejora. Si se establece en verdadero, automáticamente reservará el 10 % de los datos de entrenamiento como validación y finalizará el entrenamiento cuando la puntuación de validación no mejore al menos tol durante n_iter_no_change épocas consecutivas. Solo es efectivo cuando solver = 'sgd' o 'adam'.
- **validation_fraction : float, default = 0.1**: La proporción de datos de entrenamiento que se deben reservar como conjunto de validación para la detención anticipada. Debe estar entre 0 y 1, donde 0 indica una proporción del 0 % y 1 del 100 %. Solo se usa si early_stopping es True, por lo que en nuestro caso no será necesario su estudio.
- **n_iter_no_change : int, default=10**: Número máximo de épocas para no cumplir con la mejora tol. Un valor demasiado pequeño podría provocar un underfitting en el modelo. Solo es efectivo cuando solver = 'sgd' o 'adam'.

- **max_fun : int, default = 15000**: Solo se usa cuando solver = 'lbfgs'. Número máximo de llamadas a funciones. El solucionador itera hasta la convergencia (determinada por 'tol'), el número de iteraciones alcanza max_iter, o este número de llamadas a funciones. La cantidad de llamadas a funciones será mayor o igual que la cantidad de iteraciones para MLPRegressor.

Para estimar el mejor valor de los parámetros, hemos implementado un grid search para tratar de encontrar el valor óptimo de los parámetros **hidden_layers_size** ([[14, 14], [14, 7], [7, 7], [4, 8], [8, 4], [7, 14]]), **alpha** ([15, 14, 12, 11, 10,9,8,7, 6, 5, 3, 2, 1]) y **learning_rate_init** ([0.001, 0.01, 0.1]).

El total de las neuronas en las capas ocultas se ha seguido un "thumb rule" por la que establece que de las neuronas estén entre el tamaño de la entrada, i.e. el total de características, y el tamaño de la salida, en nuestro caso 1, ref. [1]. Incrementar más el tamaño de las capas ocultas, para nuestros datos, no significaría más que incrementar los cálculos computacionales.

El **solver** lo hemos establecido a 'lbfgs' ya que nuestro conjunto de datos es bastante pequeño y este solucionador suele dar mejores resultados para estos casos, y **max_iter** lo hemos establecido en 200, ya que como se puede ver en la gráfica 8, dar más iteraciones que 200 no significa en una mayor reducción el error.

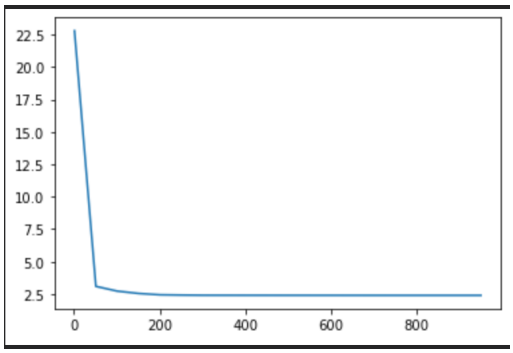

Figura 8: Evolución del error de evaluación, usando RMSE, con diferentes límites de iteraciones

El resto de parámetros los mantenemos en su valor predeterminado ya que, o bien no son utilizados para el solucionador 'lbfgs', o bien porque en este problema no nos interesa profundizar tanto en el algoritmo.

Además, MLPRegressor() cuenta con los siguientes atributos:

- **loss_ : float**: La pérdida actual calculada con la función de pérdida.
- **best_loss_ : float**: La pérdida mínima alcanzada por el solucionador durante el ajuste.
- **loss_curve_ : list of shape (n_iter_,)**: Valor de pérdida evaluado al final de cada paso de entrenamiento. El i-ésimo elemento de la lista representa la pérdida en la i-ésima iteración.
- **t_ : int**: El número de muestras de entrenamiento vistas por el solucionador durante el ajuste. Matemáticamente es igual a n_iters * X.shape[0], y es utilizado por el planificador de tasas de aprendizaje del optimizador.
- **coefs_ : list of shape (n_layers-1,)**: El i-ésimo elemento de la lista representa la matriz de ponderaciones correspondiente a la capa i.
- **intercepts_ : list of shape (n_layers-1,)**: El i-ésimo elemento de la lista representa el vector de sesgo correspondiente a la capa i+1.
- **n_iter_ : int**: El número de iteraciones que ha ejecutado el solucionador.
- **n_layers_ : int**: Número de capas.

- **n_outputs_ : int**: Número de salidas.
- **out_activation_ : str**: Nombre de la función de activación de salida.

10.1.3. Random Forest

Los parámetros de RandomForestRegressor() se definen de la siguiente forma:

- **n_estimators : int, default=100**: Indica el número de árboles que contendrá el bosque. Un mayor número aumenta el rendimiento y hace que las predicciones sean más estables, pero vuelve más lento el entrenamiento.
- **criterion : {"mse", "mae"}, default="mse"**: La función para medir la calidad de una división. Los criterios admitidos son "mse"para el error cuadrático medio, que es igual a la reducción de la varianza como criterio de selección de características, y "mae"para el error absoluto medio.
- **max_depth : int, default=None**: Se refiere a la profundidad de los árboles. Si no se especifica ningún valor, los nodos se expandirán hasta que todas las hojas sean puras (tengan una probabilidad de 1) o todas las hojas contengan menos de min_samples_split muestras. Generar árboles demasiado profundos provocaria un sobreajuste en el modelo, mientras que generarlos con muy poca profundidad incapacitaría al modelo de alcanzar un buen ajuste, provocando un underfitting.
- **min_samples_split : int or float, default=2**: Es el número mínimo de muestras necesarias para dividir un nodo interno. Si es int, se considera min_samples_split el número mínimo. Si es float, min_samples_split es una fracción y ceil(min_samples_split * n_samples) es el número mínimo de muestras para cada división. Su valor mínimo es 2, ya que no se podría hacer una división de menos de 2 observaciones. Un valor muy elevado podría provocar que los árboles no se expandan lo suficiente como para que el algoritmo llegue a converger a un buen óptimo, produciendo así underfitting.

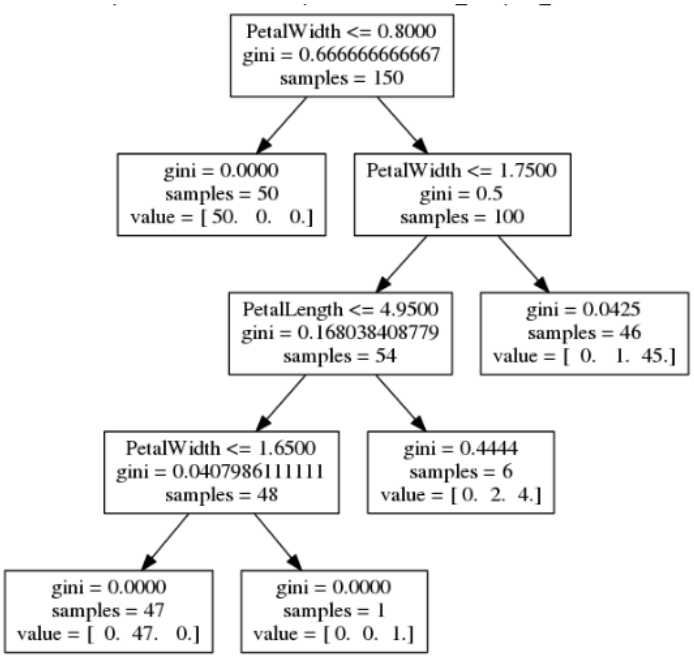

Figura 9: Ejemplo de árbol de decisión con min_samples_split=10. [13]

- **min_samples_leaf : int or float, default=1**: Es el número mínimo de muestras necesarias para que un nodo pueda ser considerado hoja. Solo se considerará un punto de división a cualquier profundidad si deja al menos min_samples_leaf muestras de entrenamiento en cada una de las ramas, garantizando así un número mínimo de muestras en las hojas. Esto puede tener el efecto de suavizar el modelo, especialmente en regresión. Si es int, se considera min_samples_leaf como el número mínimo de muestras necesarias. Si es float, entonces min_samples_leaf es una fracción y ceil(min_samples_split * n_samples)) es el número mínimo de muestras para cada nodo.

  Para entender mejor la diferencia entre estos dos últimos parámetros vamos a ver un ejemplo. Supongamos un nodo interno con 7 muestras, min_samples_split=5 y min_samples_leaf=3. Supongamos ahora que la división de este nodo interno genera dos nodos hijos, uno con 2 muestrs y otro con 5. El valor de min_samples_split (5) permitiría la división, pues el nodo interno tiene más muestras del mínimo requerido (7). Sin embargo, el valor de min_samples_lead (3) impediría esta división, pues se generaría una hoja con menos muestras del mínimo requerido (2). Por tanto, el valor mínimo que se puede asignar al parámetro min_samples_lead será 1, ya que en una división no se puede generar un nodo con menos de una muestra. Además, no tendría sentido que min_samples_lead fuese mayor que min_samples_split, ya que de la división de un nodo, nunca se podrá generar un nodo hijo con más muestras que el padre.

- **min_weight_fraction_leaf : float, default=0.0**: Es la fracción ponderada mínima de la suma total de pesos (de todas las muestras de entrada) que se requiere para estar en un nodo hoja. Las muestras tienen el mismo peso cuando no se proporciona ningún valor de sample_weight. De esta forma podemos dar mayor importancia a unas muestras que ha otras.

- **max_features : {"auto", "sqrt", "log2"}, int or float, default="auto"**: Define el número máximo de características que el algoritmo puede probar en un árbol individual. Si es int, entonces considera las max_features características en cada división. Si es float, entonces max_features es una fracción y se consideran round(max_features * n_features) en cada división. En caso de ser "auto", max_features=n_features. Si es "sqrt", max_features es igual a la raiz de n_features. Si es "log2", max_features=log_2(n_features). Si es None, max_features=n_features. En caso de introducir manualmente este parámetro (mediante un int o un float) debemos tener en cuenta que un valor demasiado pequeño podría hacer que el algoritmo no logre converger correctamente. Además, no tendría mucho sentido darle un valor mayor que n_features.

- **max_leaf_nodes : int, default=None**: El algoritmo crea árboles con un máximo de max_leaf_nodes nodos. Los mejores nodos se definen como una reducción relativa de la impureza. Si es None, entonces hay un número ilimitado de nodos hoja. Un valor muy pequeño aquí podría provocar que se generasen árboles con demasiada profundidad, dando poca diversidad a la solución y provocando un overfitting.

- **min_impurity_decrease : float, default=0.0**: Un nodo se dividirá si esta división induce una disminución de la impureza mayor o igual a este valor. La ecuación pondera de disminución de la impureza es: $\frac{N_t}{N*(impureza - \frac{N_{t_R}}{N_t*impureza_derecha} - \frac{N_{t_L}}{N_t*impureza_izquierda})}$, donde N es el número total de muestras, N_t es el número de muestras del nodo actual, N_{t_R} es el número de muestras del hijo derecho y N_{t_L} es el número de muestras del hijo izquierdo. Así, la impureza irá descendiendo hasta alcanzar el mínimo indicado, por lo que un valor muy elevado provocaría un resultado final con una elevada impureza.

- **min_impurity_split : float, default=None**: Umbral de parada temprana en el crecimiento de los árboles. Un nodo se dividirá si su impureza está por encima del umbral; de lo contrario, es una hoja. Un valor demasiado elevado daría lugar a árboles con una gran impureza.

- **bootstrap : bool, default=True**: Si está en falso se usa todos los datos para contruir los árboles de decisión y si está a verdadero se selecciona datos del conjunto de datos pero no todos ellos.

- **oob_score : bool, default=False**: Indica si utilizar muestras fuera del subconjunto de datos bootstrap. En caso de que bootstrap sea False, este también lo será, pues no habrá muestras disponibles para utilizar fuera del conjunto seleccionado.
- **n_jobs : int, default=None**: Indica el número de trabajos que se ejecutarán en paralelo (fit, predict, decision_path, apply). None indica que únicamente se ejecute un trabajo simultáneamente y -1 que se utilicen todos los procesadores posibles.
- **random_state : int, RandomState instance or None, default=None**: Especifica la semilla
- **verbose : int, default=0**: Util si queremos depurar el ajuste y la predicción
- **warm_start : bool, default=False**: Si está en verdadero, hacer uso de la llamada anterior para ajustar más datos.
- **ccp_alpha : non-negative float, default=0.0**: Es el parámetro de complejidad utilizado para la poda de costo mínimo-complejidad. Se elegirá el subárbol con la complejidad de costo menor que ccp_alpha. De forma predeterminada, no se realiza ninguna poda. A mayor valor de ccp_alpha, más probable será que se produzca una poda, pues habrá mas ramas con un costo en el rango [0,ccp_alpha].
- **max_samples : int or float, default=None**: Solo se aplica si bootstrap es True, e indica el número de muestras que contendrá el subconjunto de X. Si es None, se extraen X.shape[0] muestras (todas las muestras). Si es un int, se extraen max_samples muestras. Si es un float, deberá ser un valor entre 0 y 1, y se extraerán max_samples*X.shape[0] muestras. De este modo, extraer subconjuntos demasiado pequeños generaría árboles de decisión también demasiado pequeños, lo que provocaría una convergencia más lenta, pudiendo producirse un underfitting.

Para elegir el valor de **n_estimators** ([10,20,50,100,150,200,500]), para ver el valor óptimo de número de árboles, y **max_depth** ([5,7,10,15,20,30,50]), para ver la profundidad máxima óptima de los árboles, hemos implementado un grid search para estimar el mejor valor posible de estos parámetros.

En el caso de **criterion** lo hemos mantenido en "mse"para diferenciar más los errores grandes de los pequeños.

Por su parte, **max_features** lo mantenemos en "auto" para poder utilizar cualquier subconjunto de características en un árbol cualquiera.

El resto de parámetros los mantenemos en su valor predeterminado ya en este problema no nos interesa profundizar tanto en el algoritmo.

Además, RandomForestRegressor() cuenta con los siguientes atributos:

- **base_estimator_ : DecisionTreeRegressor**: Es la plantilla de estimador hijo utilizada para crear la colección de subestimadores ajustados.
- **estimators_ : list of DecisionTreeRegressor**: Colección de subestimadores ajustados.
- **feature_importances_ndarray of shape (n_features,)**: Indica la importancia de características basadas en la pureza.
- **n_features_ : int**: Número de características.
- **n_outputs_ : int**: Número de salidas.
- **oob_score_ : float**: Puntaje del conjunto de datos de entrenamiento obtenido usando una estimación fuera de bootstrap cuando oob_score sea True.
- **oob_prediction_ : ndarray of shape (n_samples,)**: Predicción calculada con estimación fuera bootstrap en el conjunto de entrenamiento cuando oob_score sea True.

** Para todos los grid search hemos utilizado la función GridSearchCV() de sklearn.model_selection, la cual realiza una búsqueda exhaustiva de los valores especificados para un estimador, implementando un método de ajuste y puntuación. Los valores del estimador utilizados para aplicar estos métodos se optimizan mediante una búsqueda de cuadrícula con validación cruzada sobre una cuadrícula de parámetros.*

Modelo	**Puntuación de CV(RMSE)**
Regresión lineal	4.961241
Regresión lineal + l1	4.961962
Regresión lineal + l2	4.948122
MLP	3.434288
Random forest	3.677844
Boosting	3.343135

: Resultado de cross validation de las mejores hipótesis de cada modelo

11. Valoración de resultados

Una vez terminada la búsqueda de hiperparámetros, vamos a proceder a seleccionar la hipótesis que mejro se ha comportado en cross validation de cada modelo y entrenaremos nuestras hipótesis con los datos que tenemos. A continuación procederemos a evaluar qué tal se comporta el modelo escogido y entrenado con nuestro conjunto de entrenamiento y nuestro conjunto de prueba.

Vamos a usar 2 métricas para medir el error de evaluación, RMSE y R^2.

11.1. Root Mean Square Error(RMSE)

El error cuadrático medio (RMSE) es la media de las raices de las distancias al cuadrado, es decir, de la función de pérdida utilizada, $RMSE = \sqrt{\frac{\sum_{i=0}^{N}(w^T x_i - y_i)^2}{N}}$, la cual indica cuán cerca están los valores predichos por el modelo de los valores reales, donde a menor valor de RMSE, mejor será el ajuste. Para ello haremos uso de la función mean_squared_error() de sklearn.metrics. Ésta métrica tiene la ventaja de que es un poco más fácil de entender que MSE.

Resultados obtenidos dentro de la muestra
Regresión Lineal: 4.812900828845337
MLP: 2.6220435698466313
Random Forest: 1.4187544574763153

Resultados obtenidos fuera de la muestra
Regresión Lineal: 4.814378295422549
MLP: 3.574959721894952
Random Forest: 3.3025939633685906

11.2. R squared(R^2)

Esta métrica indica la aptitud del modelo, se calcula como $R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \overline{y})^2}$, donde y_i es el valor real de la etiqueta i, $\hat{y}_i$ es el valor predecido de dicha etiqueta, y $\overline{y}_i$ la media de las etiquetas. Se entiende que el numerador es la suma de los residuos al cuadrado y el denominador es la sumar total de cuadrados. Esto nos dará un valor entre 0 y 1 que, a medida que se aproxima a 0, indica que el modelo no realiza una buena predicción, y a medida que se aproxima a 1, indica que la predicción del modelo es perfecta.

Resultados obtenidos dentro de la muestra
Regresión Lineal: 0.7229471317006773
MLP: 0.9177703138524694
Random Forest: 0.9759251659190851

Resultados obtenidos fuera de la muestra
Regresión Lineal: 0.7314090493456338
MLP: 0.8519006141432562
Random Forest: 0.8736074962856174

11.3. Entendiendo la métrica de evaluación

Se observa que los ajustes obtenidos dentro de la muestra son un poco mejores que los obtenidos fuera de la muestra. Esto es un comportamiento natural ya que el error en entrenamiento suele ser menor que el error en prueba. En regresión lineal, el modelo que hemos ajustado se comporta casi exactamente igual en ambos conjuntos. En MLP podemos ver que se comporta muy bien en training, ofreciendo mejores resultados que regresión lineal, pero el modelo obtenido se comporta peor en test, pero aún así mejor que regresión lineal. Con Random Forest observo que ofrece resultados muy, muy buenos, casi perfectos, superando a MLP, en training pero cuando este modelo es aplicado sobre el conjunto de prueba no es mucho mejor que MLP. Por lo tanto tenemos un ligero overfitting con este modelo.

12. Argumentación de la mejor solución posible

Como hemos podido comprobar en el apartado anterior, y como era de esperar, el algoritmo de Regresión Lineal es el que peor resultados da, muy limitado por no tratarse de un problema con datos linealmente separables. No obstante, podemos apreciar que el algoritmo no empeora las predicciones fuera de la muestra de entrenamiento respecto a las predicciones dentro de esta, por lo que podemos intuir que el algoritmo no genera overfitting y que, de tratarse de un problema con datos linealmente separables, podría ser un modelo a tener muy en cuenta.

Por su parte, los algoritmos MLP y Random Forest ofrecen mejores resultados, ya que estos no son modelos lineales, y logran un mejor ajuste que un modelo lineal para este problema. En la comparativa entre estos dos algoritmos, vemos que el que sale vencedor es el algoritmo de Random Forest, al tener un menor RMSE y un mayor R^2 tanto con los datos de entrenamiento como con los de prueba. Esto

se debe a que el algoritmo MLP, como red neuronal que es, requiere de muchos mas datos para lograr un ajuste óptimo, cantidad de datos con la que no cuenta nuestro dataset. Por ello, concluimos que el mejor modelo de predicción de los tres comparados, dado este conjunto de datos, es el algoritmo de Random Forest.

Independientemente de los modelos que hemos usado, el mejor modelo será aquel que predizca correctamente el precio de la vivienda tanto dentro y fuera de la muestra, y además, deberá ser capaz de dar buenas predicciones con nuevos datos que se obtengan y que no haya visto antes sin tener que ser reentrenado con los nuevos datos.

13. Bonus

Como parte adicional a la práctica, vamos a aplicar el algoritmo de Boosting al problema anterior, y compararlo así con los demás modelos.

13.1. Boosting

Boosting es una técnica de aprendizaje secuencial, la cual funciona entrenando un modelo con todo el conjunto de entrenamiento y los modelos posteriores se construyen ajustando los valores del error residual del modelo inicial, pseudo residual, dando mayor peso a aquellas muestras que el modelo anterior estimó pobremente. Una vez creada la secuencia de los modelos, las predicciones hechas por los modelos son ponderadas por sus puntuaciones de precisión y los resultados se combinan para crear una estimación final. Este modelo hace uso de un learning rate, entre 0 y 1. De acuerdo con Jerome Friedman, la experiencia empírica demuestra que dar pequeños pasos, i.e menor learning rate, resulta en mejores predicciones en los datos de prueba, o lo que es lo mismo, una menor varianza.

Para ello utilizaremos la función GradientBoostingRegressor() de sklearn.ensemble, el cual combina árboles de decisión para generar una predicción final, de forma que el algoritmo va añadiendo árboles aplicando un procedimiento de gradiente descendente que minimizará la función de pérdida para determinar los parámetros que tendrá cada árbol, de tal forma que la combinación de estos árboles minimiza la pérdida del modelo y mejora la predicción. En dicha función se definen los siguientes parámetros:

- **loss : 'ls', 'lad', 'huber', 'quantile', default='ls'**: Función de pérdida a optimizar. 'ls' se refiere a la regresión de mínimos cuadrados. 'lad' (desviación mínima absoluta) es una función de pérdida muy robusta basada únicamente en la información de orden de las variables de entrada. 'huber' es una combinación de los dos. 'quantile' permite la regresión de cuantiles.
- **learning_rate : float, default=0.1**: La tasa de aprendizaje reduce la contribución de cada árbol en learning_rate. Existe una compensación entre learning_rate y n_estimators, la cual veremos posteriormente.
- **n_estimators : int, default=100**: Es el número de etapas de boosting a realizar, es decir, el número de árboles de decisión a crear. El aumento de gradiente es bastante robusto al sobreajuste, por lo que un gran número generalmente da como resultado un mejor rendimiento, aunque, al igual que en Random Forest, un número demasiado elevado relentizaría el proceso de aprendizaje, pudiendo producir underfitting.
- **subsample : float, default=1.0**: La fracción de muestras que se utilizará para ajustar a los learners de base individuales. Si es menor que 1.0, esto da como resultado un aumento de gradiente estocásticom lo cual conduce a una reducción de la varianza y un aumento del sesgo.. subsample interactúa con el parámetro n_estimators, pues cuanto menor sea la muestra, mayor será el número de árboles necesarios.
- **criterion : 'friedman_mse', 'mse', 'mae', default='friedman_mse'**: Es muy similar a Random Forest, con la única diferencia de que añade una opción más: 'friedman_mse', la cual representa el error cuadrático medio con puntuación de mejora de Friedman.
- **min_samples_split : int or float, default=2**: Igual que en Random Forest.
- **min_samples_leaf : int or float, default=1**: Igual que en Random Forest.
- **min_weight_fraction_leaf : float, default=0.0**: Igual que en Random Forest.

- **max_depth : int, default=3**: Igual que en Random Forest.
- **min_impurity_decrease : float, default=0.0**: Igual que en Random Forest.
- **min_impurity_split : float, default=None**: Igual que en Random Forest.
- **init : estimator or 'zero', default=None**: Estimador utilizado para calcular las predicciones iniciales. Si es 'zero', las predicciones sin procesar iniciales se establecen en cero
- **random_state : int, RandomState instance or None, default=None**: Igual que en Random Forest.
- **max_features'auto', 'sqrt', 'log2', int or float, default=None**: Igual que en Random Forest.
- **alpha : float, default=0.9**: Es l alfa-cuantil de la función de pérdida de huber y la función de pérdida de cuantiles, por lo que solo se utilizará cuando loss sea 'huber' o 'quantile'.
- **verbose : int, default=0**: Igual que en Random Forest.
- **max_leaf_nodes : int, default=None**: Igual que en Random Forest.
- **warm_start : bool, default=False**: Igual que en Random Forest.
- **validation_fraction : float, default=0.1**: La proporción de datos de entrenamiento que se deben reservar como conjunto de validación para la detención anticipada. Debe estar entre 0 y 1, donde 0 indica una proporción del 0 % y 1 del 100 %. Solo se usa si n_iter_no_change se establece en un número entero, por lo que en nuestro caso no será necesario su estudio.
- **n_iter_no_change : int, default=None**: Se utiliza para decidir si la parada anticipada se utilizará para finalizar el entrenamiento cuando la puntuación de validación no mejora. De forma predeterminada, está configurado en None para deshabilitar la parada anticipada. Si se establece en un número, reservará un subconjunto de datos de tamaño validation_fraction de los datos de entrenamiento para validación y finalizará el entrenamiento cuando la puntuación de validación no mejore en todos los n_iter_no_change anteriores de iteraciones.
- **tol : float, default=1e-4**: Tolerancia a la parada anticipada. Cuando la pérdida no mejora en al menos un tol para las n_iter_no_change iteraciones, el entrenamiento se detiene, por lo que solo se usa si n_iter_no_change se establece en un número entero.
- **ccp_alpha : non-negative float, default=0.0**: Igual que en Random Forest.

Los parámetros más relevantes son n_estimators y learning_rate, que están altamente relacionados. Cuanto menor sea el valor que asignemos a learning_rate, el modelo requerirá un mayor número de árboles (n_estimators) para alcanzar un buen ajuste. Por ello, para seleccionar el mejor valor de dichos parámetros, vamos a aplicar una búsqueda exhaustiva de **n_estimators**([50,100,150,200]) y **learning_rate**([0.07,0.85,0.1,0.2,0.3,0.5]).

El resto de parámetros los vamos a mantener en su valor predeterminado pues no queremos profundizar demasiado en este algoritmo.

Como podemos ver, este algoritmo comparte muchos parámetros con Random Forest. Esto se debe a que ambos son métodos de ensamblado, los cuales consisten en combinar un conjunto de diversos modelos, en este caso árboles de decisión (aquí es donde reside la mayor coincidencia en los parámetros), para dar lugar a un modelo con mayor estabilidad y mayor poder predictivo.

Además, GradientBoostingRegressor() cuenta con los siguientes atributos:

- **feature_importances_ : ndarray of shape (n_features,)**: Indica la importancia de características basadas en la pureza.

- **oob_improvement_ : ndarray of shape (n_estimators,)**: Indica la mejora en la pérdida (desviación) en las muestras de tamaño subsample*n_sample, por lo que solo está disponible si subsample ¡1.0, en relación con la iteración anterior. oob_improvement_[0] es la mejora en la pérdida de la primera etapa sobre el estimador init.
- **train_score_ : ndarray of shape (n_estimators,)**: La puntuación train_score_[i] es la desviación del modelo en la iteración i de la submuestra. Si subsample==1, esta es la desviación de los datos de entrenamiento.
- **loss_ : LossFunction**: Devuelve la función de pérdida utilizada.
- **init_ : estimator**: Estimador que proporciona las predicciones iniciales.
- **estimators_ : ndarray of DecisionTreeRegressor of shape (n_estimators, 1)**: Colección de subestimadores ajustados.
- **n_classes_ : int**: Número de clases, en caso de regresión, devuelve 1.
- **n_estimators_ : int**: El número de estimadores seleccionados por parada anticipada en caso de que se haya especificado n_iter_no_change, si no, se establece a n_estimators.
- **n_features_ : int**: Número de características.
- **max_features_ : int**: Valor inferido de max_features.

13.2. Resultados

Resultados obtenidos dentro de la muestra
RMSE: 0.9472574275860565 R^2: 0.989267896227737

Resultados obtenidos fuera de la muestra
RMSE: 2.977685502931762 R^2: 0.8972531291139219

13.3. Análisis de resultados

Como podemos ver, los resultados obtenidos son muy similares a los obtenidos con Random Forest, llegando incluso a mejorarlos ligeramente. Esto se debe a que ambos disminuyen la varianza de sus estimación combinando varias estimaciones de un mismo modelo: árboles de decisión. Sin embargo, Boosting genera predicciones ligeramente mejores debido a que el modelo de árboles de decisión obtiene un rendimiento muy bajo y Boosting optimiza las ventajas y disminuye las limitaciones de este modelo. Mientras que Random Forest aplica mayor diversidad al modelo, tal y como explicamos anteriormente, por lo que que daría mejores resultados si los árboles de decisión se ajustasen en exceso. Aunque es cierto que en este caso los árboles de decisión pueden generar overfitting, quizá no sea lo suficiente como para que Random Forest mejore los resultados de Boosting, lo cual permite a Boosting generar mejores predicciones

Referencias

[1] https://web.archive.org/web/20140721050413/http://www.heatonresearch.com/node/707

[2] Documentación acerca de los parámetros y atributos del Ridge

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html

[3] Documentación acerca de Bias-Variance Tradeoff

https://themachinelearners.com/tradeoff-bias-variance/

https://www.analyticsvidhya.com/blog/2020/12/a-measure-of-bias-and-variance-an-experiment/

[4] Documentación acerca del algoritmo perceptron

https://www.cienciadedatos.net/documentos/50_algoritmo_perceptron#:~:text=El%20algoritmo%20Perceptron%20fue%20publicado,puede%20utilizarse%20para%20clasificaciones%20binarias.

[5] Documentación acerca de los parámetros y atributos del MLP_Regressor

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html

[6] Documentación acerca de los parámetros y atributos del RandomForest_Regressor.

https://www.youtube.com/watch?v=TqsD5q6aqt0

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

https://medium.datadriveninvestor.com/decision-tree-adventures-2-explanation-of-decision-tree-clas

[7] Documentación acerca de GridSearchCV.

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[8] MinMax Scaler

https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

[9] Normalización vs estandarización https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/

[10] Perceptrón multicapa

http://bibing.us.es/proyectos/abreproy/12166/fichero/Volumen+1+-+Memoria+descriptiva+del+proyecto%252F3+-+Perceptron+multicapa.pdf

[11] Métricas de error

https://aprendeia.com/evaluando-el-error-en-los-modelos-de-regresion/

[12] Cross Validation

https://scikit-learn.org/stable/modules/cross_validation.html

[13] min_samples_split

https://discuss.analyticsvidhya.com/t/what-does-min-samples-split-means-in-decision-tree/6233/3

[14] min_samples_leaf

https://stackoverflow.com/questions/46480457/difference-between-min-samples-split-and-min-samples-

[15] Boosting

https://relopezbriega.github.io/blog/2017/06/10/boosting-en-machine-learning-con-python/