Problema de regresión:
Airfoil-Self-Noise

INDICE

Definición del problema y la base de datos

Mediante la información obtenida de https://archive.ics.uci.edu/ml/datasets/airfoil+self-noise podemos ver que contamos con un fichero airfoil_self_noise.txt que contiene información de 6 atributos extraídos de una serie de pruebas aerodinámicas y acústicas de secciones de palas aerodinámicas bidimensionales y tridimensionales realizadas en un túnel de viento anecoico. En total disponemos de 1503 instancias.

Observando el fichero, los datos se distribuyen en 6 columnas y 1503 filas. Cada columna hace referencia a las diferentes entradas del problema: frecuencia (en hercios), ángulo de ataque (en grados), longitud de la cuerda (en metros), velocidad de flujo libre (en m/s) y el espesor de desplazamiento del lado de succión (en metros). La última columna hace referencia a la única salida que tenemos, el nivel de presión sonora escalada (en decibelios). Estamos ante un problema de regresión multiclase.

La función que queremos aprender f: $X \rightarrow Y$ es desconocida, con $X = R^6$ $Y = [0, \infty)$, y será la que asigne el nivel de presión sonora para cada prueba que se realiza con sus diferentes atributos.

Para poder profundizar en nuestros datos, procederemos a visualizarlos.

Con la función pairplot procedente de Seaborn somos capaces de trazar una gráfica en la que se relacionan los atributos por pares de nuestro conjunto de datos. Las gráficas que se encuentran en la diagonal son una distribución univariante para mostrar la distribución marginal de los datos en cada columna. A parte de observar los datos, podremos comprobar que estos han sido cargados correctamente.

Además, vamos a seleccionar las variables más predictoras para observarlas y tratar de extraer alguna conclusión sobre ellas. Para realizarlo empleamos la clase DecisionTreeRegressor procedente de Scikit-Learn que permite hacerlo con gran facilidad.

Como conocemos el atributo que queremos predecir, también vamos a observar por medio de un histograma como se distribuye a lo largo del conjunto de datos.

Por último y para tratar de confirmar nuestras intuiciones, usaremos T-SNE para mostrar el conjunto de datos.

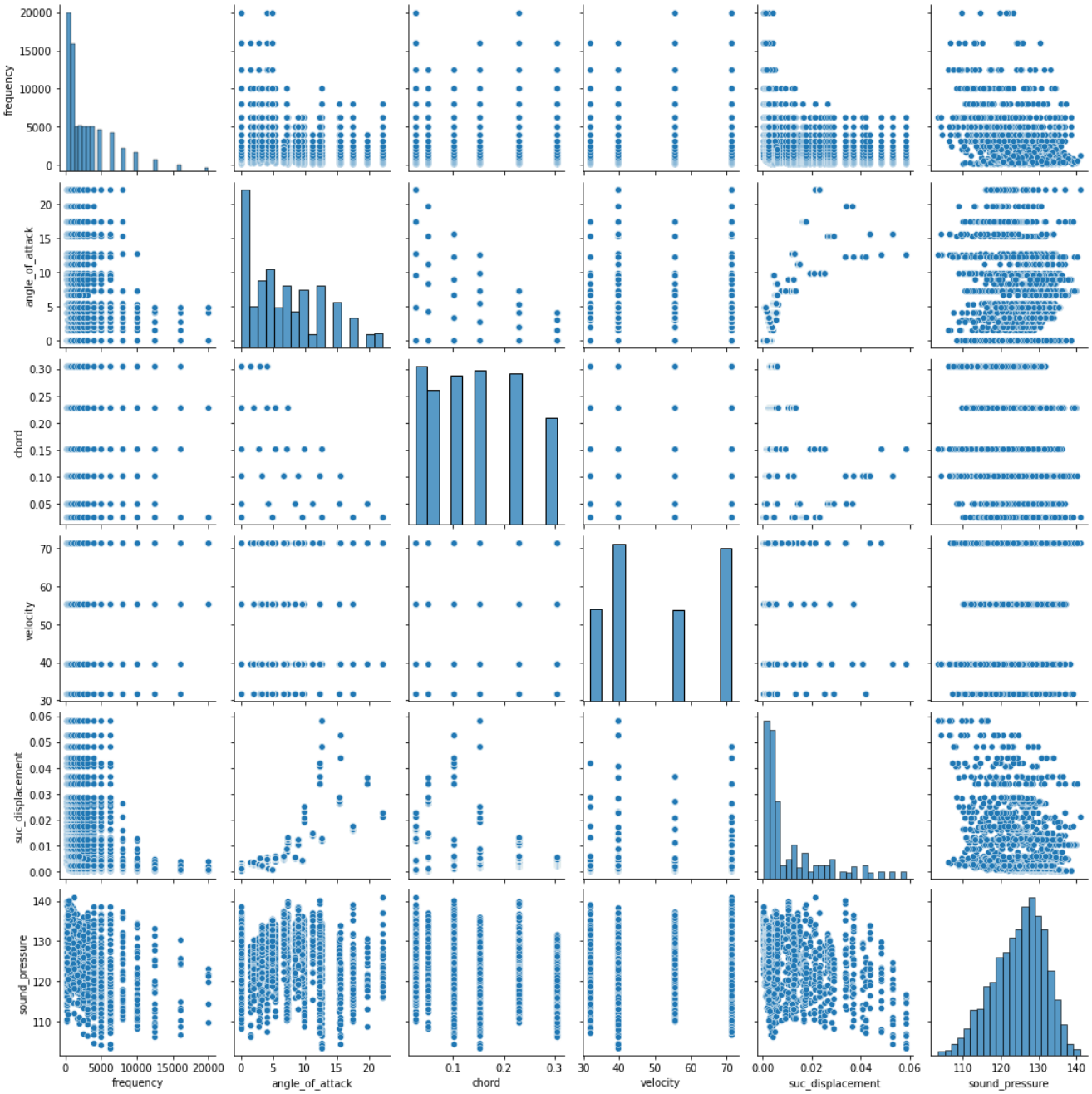


Figura 1. Gráfica con la relación de los atributos a pares y la distribución de los mismos en la diagonal

Vamos a ver las variables con mayor importancia de nuestro conjunto de datos:

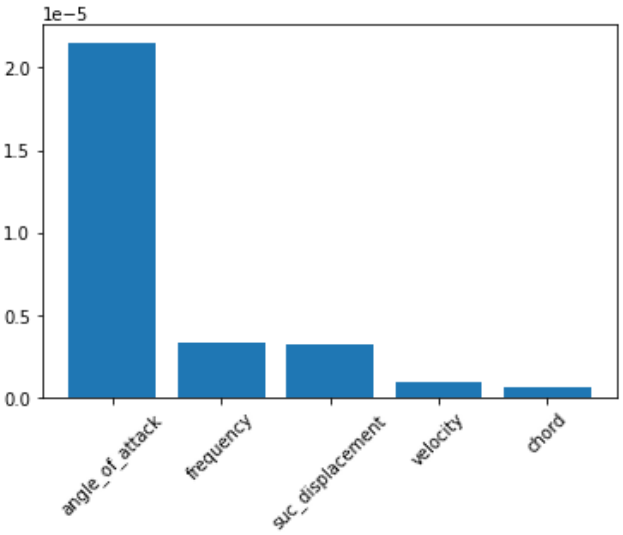


Figura 2. Gráfica con los atributos ordenados por importancia en el conjunto de datos

La variable con mayor poder predictivo es el ángulo de ataque (Angle of attack, seguida de la frecuencia (Frequency) y el espesor de desplazamiento del lado de succión (Suction side displacement thickness).

Vamos a comprobar si representando estos atributos junto con la variable a predecir (Scaled sound pressure level) somos capaces de extraer alguna conclusión sobre nuestros datos:

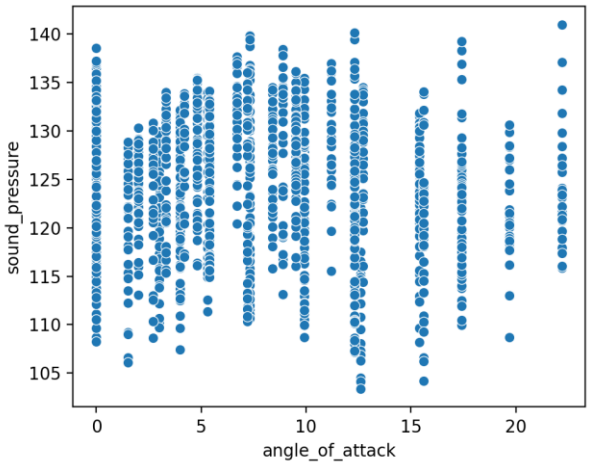


Figura 3. Gráfica del atributo más predictivo con respecto a la variable a predecir

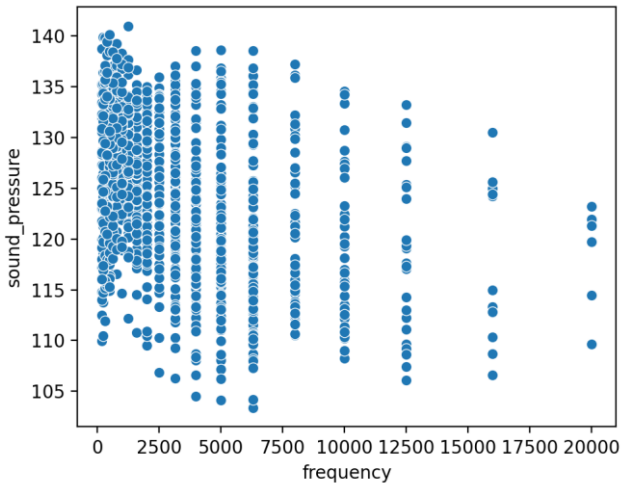


Figura 4. Gráfica del segundo atributo más predictivo con respecto a la variable a predecir

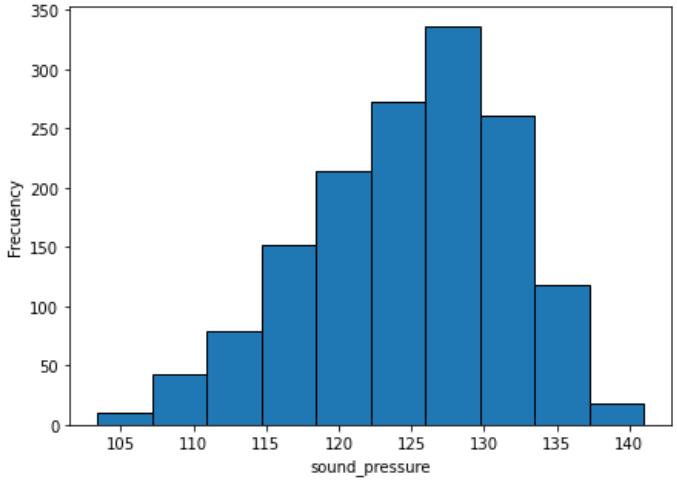


Figura 5. Gráfica de la distribución de la variable a predecir

Podemos ver que es difícil intuir alguna idea de regresión que sea capaz de relacionar nuestro objetivo en el problema. Además, en el histograma observamos que la presión sonora se distribuye aproximadamente de forma normal.

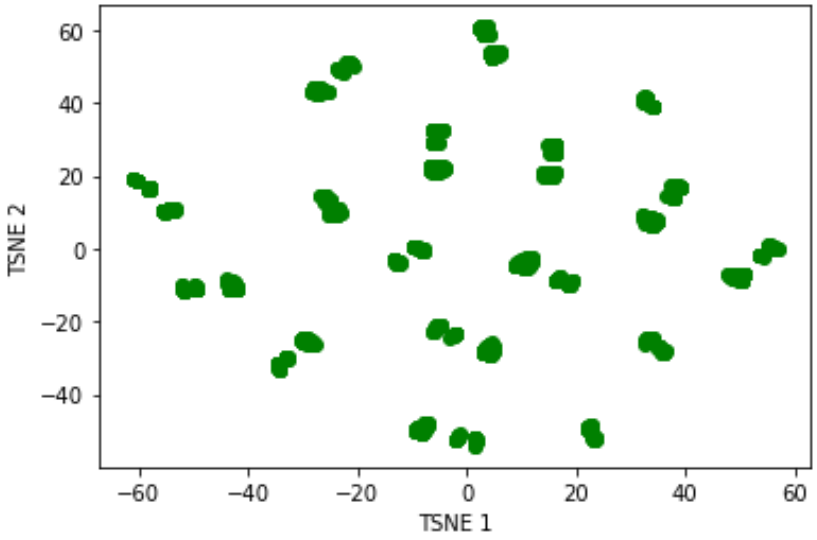


Figura 6. Gráfica de T-SNE para visualizar el conjunto de datos

Por medio de T-SNE (T-Distributed Stochastic Neighbor Embedding) podemos ver la distribución de los clusters de nuestro data set y sacar como conclusión que necesitaremos modelos no lineales para poder obtener las mejores predicciones para este problema.

Realmente estamos ante una base de datos relativamente sencilla. Esto lo sabemos gracias a las diferentes funciones que nos ofrece la biblioteca Pandas. La primera de ellas, head, nos permite observar las primeras filas de datos (por defecto, las 5 primeras). Esto nos será útil para conocer rápidamente si nuestros datos son del tipo que deseamos.

Column	Non-Null Count	Dtype
Frequency	1503 non-null	float64
Angle of attack	1503 non-null	float64
Chord length	1503 non-null	float64
Free stream velocity	1503 non-null	float64
Suction side displacement thickness	1503 non-null	float64
Scaled sound pressure level	1503 non-null	float64

Tabla con el formato de datos del conjunto

Vemos que no tenemos datos nulos (siempre contamos con 1503 datos) y que además son floats de 64 bits.

Con la función describe() podemos ver unas estadísticas descriptivas de nuestro dataset:

El recuento de datos, count, que como ya sabíamos es de 1503 ejemplares. La media de cada atributo, la desviación típica, los valores mínimos y máximos así como los percentiles 25, 50 y 75.

	Frequency	Angle of attack	...	Suction side displacement thickness	Scaled sound pressure level
count	1503.000000	1503.000000	...	1503.000000	1503.000000
mean	2886.380572	6.782302	...	0.011140	124.835943
std	3152.573137	5.918128	...	0.013150	6.898657
min	200.000000	0.000000	...	0.000401	103.380000
25%	800.000000	2.000000	...	0.002535	120.191000
50%	1600.000000	5.400000	...	0.004957	125.721000
75%	4000.000000	9.900000	...	0.015576	129.995500
max	20000.000000	22.200000	...	0.058411	140.987000

Tabla con información descriptiva del conjunto de datos

Una vez conocemos mejor los datos, vamos a ver por medio de la matriz de correlación la disposición de los mismos:

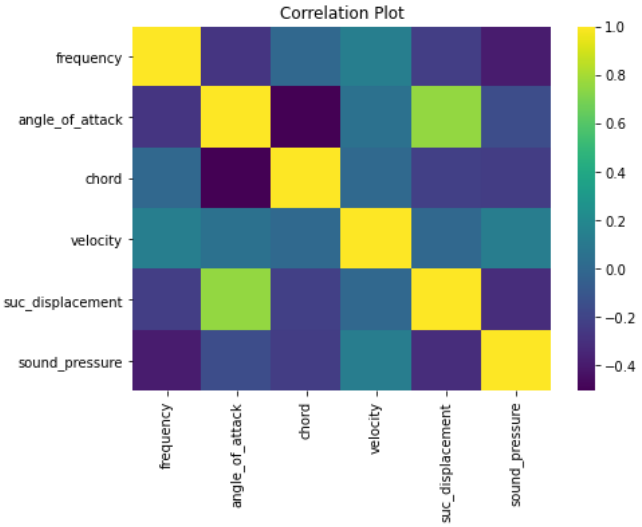


Figura 7. Gráfica de la matriz de correlación del conjunto de datos inicial

Como vemos, el ángulo de ataque y el espesor de desplazamiento del lado de succión se encuentran correlados. Es por ello por lo que vamos a aplicar un preprocesado a los datos para poder trabajar con ellos. Su explicación se encuentra en el siguiente apartado del informe.

Selección de H

Con la observación de datos realizada, es fácil comprobar que en un principio nuestro conjunto de datos no es linealmente separable. Esto nos hace pensar que nuestro modelo debería considerar transformaciones no lineales en vez de lineales para aproximar de la mejor manera posible los datos ofrecidos.

Realizar transformaciones polinómicas nos lleva a un aumento de la complejidad de H y, esto nos puede hacer que caigamos en overfitting. Es por ello por lo que se usarán técnicas de regularización para evitarlo. Entre ellas estará Lasso, la propuesta del profesor para los problemas de regresión.

Nuestro espacio se verá transformado y este será al que se le apliquen las transformaciones mencionadas. Como haremos una reducción de dimensionalidad, partiremos de un espacio X' con dimensión D' donde D' ≤ D. Tras las transformaciones polinómicas, nuestro espacio quedará como:

$\Phi_2 = X' \rightarrow Z$

$(x_1, \dots, x'_d) = \ x \rightarrow \ \Phi_2(x) =$
$(1, x_1, \dots, x_{d'}, x_1^2, \dots, x_{d'}^2, x1, x2, x1x3, \dots, x1x_{d'}, x2x3, \dots, x2x_{d'}, \dots, x_{d'-1}x_{d'})$

Nuestro espacio transformado $Z = \ \Phi_2\ (X')$, con dimensión:

$$d'' = 1 + 2d' + \frac{d'(d'-1)}{2}$$

Nuestra dimensión d' < 6 y la clase de funciones quedará definida de la siguiente forma:

$$H = \left\{ w_0 + \sum_{i=1}^{d'} w_i x_i + \sum_{i=1}^{d'} w'_i x_i^2 + \sum_{i=1}^{d'} \sum_{j=1}^{d'} w_{ij} x_i x_j : \ w_i, w'_i, w_{ij}\ \varepsilon\ \mathbb{R} \right\} = \{ \mathrm{w}^{\mathrm{T}} \Phi_2(x) : w \varepsilon \mathbb{R}^{d''} \}$$

Partición del conjunto de datos

Para realizar una partición de los datos, vamos a emplear las herramientas que nos ofrece Scikit-Learn. En concreto, hacemos uso de train_test_split la cual, mezclando los datos para evitar sesgos en la ordenación, nos separa el 80% de los datos en el conjunto train y el 20% restante en el conjunto test.

Mediante K-Fold Cross Validation vamos a seleccionar nuestro conjunto de validación. Lo más común es usar una k=5 o k=10 (en nuestro caso usaremos k=5). Usar CV evita que nuestro error sea demasiado optimista porque no es una medición, sino varias, por lo que no podemos especificarlo.

Preprocesado de los datos

Vamos a realizar todas las manipulaciones sobre los datos iniciales que nos permitan fijar el conjunto de vectores de características que se usarán en el entrenamiento. A esto le llamamos preprocesado de datos y se va llevar a cabo mediante el uso de la clase Pipeline de Scikit-Learn la cual nos permite automatizar este proceso aplicando las transformaciones y un estimador a nuestro conjunto de datos.

La reducción de dimensionalidad se va llevar a cabo usando PCA (Principal Component Analysis). Este es un método estadístico que simplifica la complejidad de espacios muestrales con muchas dimensiones a la vez que conserven la información. Esto nos llevará a reducir el overfitting así como reducir el tiempo de training.

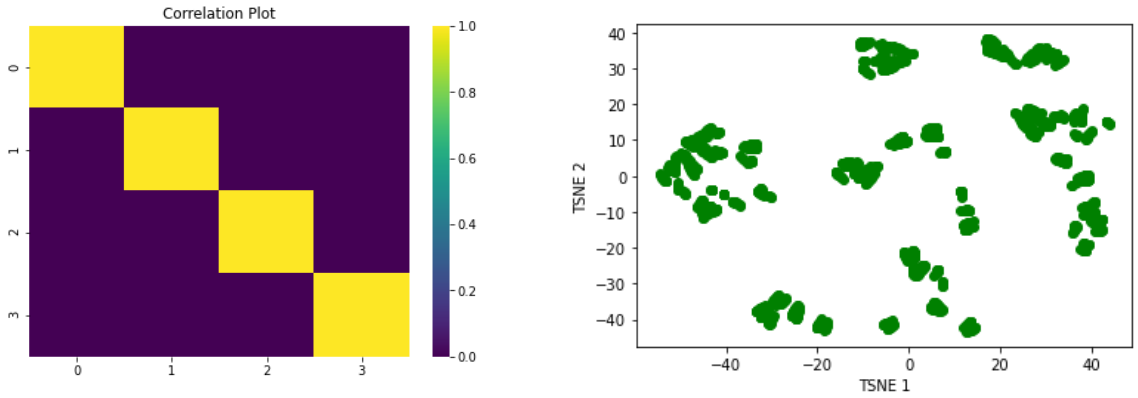
El argumento pasado a PCA, en nuestro caso 0.9, indica que la varianza acumulada de las variables tiene que ser un 90% de las iniciales. A menor valor, mayor será la reducción que se aplique y es importante hacer un uso responsable del mismo.

Para escalar los datos empleamos la clase MinMaxScaler de Scikit-Learn la cual escala todas las características entre 0 y 1, donde las características de valor mínimo será 0 y la característica de mayor será 1, escalando los valores comprendidos entre el valor mínimo y valor máximo.

Como hemos comentado en el apartado de selección de H, vamos a aplicar una transformación no lineal sobre los datos, en concreto, una transformación polinomial cuadrática. Esto consiste en elevar las características existentes a un exponente.

El grado del polinomio se usa para controlar el número de características agregadas, es decir, aplicar grado 3 significa añadir dos nuevas variables por cada variable de entrada. Normalmente se usa un grado pequeño, 2 o 3, siendo 2 el grado que utilizaremos para esta ocasión. Para realizar esta transformación vamos a hacer uso de la clase PolynomialFeatures de Scikit-Learn.

Podemos observar con la gráfica de la matriz de correlación la calidad de nuestro preprocesado. La correlación nos va a indicar la fuerza y la dirección de la relación entre dos variables. Es interesante eliminar aquellos datos con correlación fuerte para evitar problemas de redundancia:



Figuras 8 y 9. Gráficas de la matriz de correlación y T-SNE del conjunto de datos x_train preprocesado

Como vemos, con el preprocesado (x_train) hemos conseguido eliminar la correlación que existía entre las diferentes variables, sin embargo, vamos a contar con un conjunto de datos más reducido que nos debería asegurar unos mejores resultados. Hemos preferido sacrificar algunos datos con el fin de tener mejores resultados y mejores costes computacionales.

Métricas de error empleadas para este problema

Para este problema de regresión vamos a emplear las siguientes métricas de error: el error medio absoluto (MAE, mean absolute error), el error cuadrático medio (MSE, mean squared error), el coeficiente de determinación R^2 y la raíz del error cuadrático medio (RMSE, root mean squared error).

El coeficiente de determinación determina la calidad del modelo para replicar los resultados, y la proporción de variación de los resultados que puede explicarse por el modelo.

El error cuadrático medio es la función de pérdida cuando utilizamos regresión lineal. Este mide la cantidad de error que hay entre dos conjuntos de datos. En otras palabras, compara un valor predicho y un valor observado o conocido. Cuanto más pequeño sea este valor, más cercanos serán los valores.

El error medio absoluto es una medida de la diferencia entre dos variables continuas. Sirve para cuantificar la precisión de una técnica de predicción comparando por ejemplo los valores predichos frente a los observados.

El error cuadrático medio no es del todo intuitivo porque nos da el error medio al cuadrado y eso nos puede hacer pensar que podemos tener un error muy grande cuando no es así. Es por ello por lo que usaremos esta métrica para no tener una idea equivocada del error obtenido y dar una estimación en términos intuitivos de la calidad de la predicción.

Estas métricas serán explicadas con un poco de mayor detalle en el análisis de los datos.

Regularización empleada

La regularización consiste en añadir una penalización a la función de coste, esto produce modelos más simples que generalizan mejor. Como ya hemos comentado, los modelos complejos tienden a sobreajustar, es decir, encuentran una solución muy buena para los datos del tranining pero luego es muy mala para los datos de test, cosa que no queremos. Es por ello por lo que emplearemos dos de las técnicas más utilizadas para regularización en Machine Learning: la regularización Lasso (L1) y la regularización Ridge (L2).

La primera de ellas, mide la complejidad H como la media del valor absoluto de los coeficientes del modelo:

$$H = \frac{1}{N}\sum_{j=1}^{N} |w_i|$$

El desarrollo de Lasso para el error cuadrático medio queda de tal forma:

$$H = \frac{1}{M}\sum_{i=1}^{M} (real_i - estimado_i)^2 + \alpha \frac{1}{N}\sum_{j=1}^{N} |w_i|$$

L2 a diferencia de L1 mide la complejidad como la media del cuadrado de los coeficientes del modelo en vez de la media del valor absoluto de los coeficientes del modelo:

$$H = \frac{1}{2N}\sum_{j=1}^{N} w_i^2$$

El desarrollo de Ridge para el error cuadrático medio queda de tal forma:

$$H = \frac{1}{M}\sum_{i=1}^{M} (real_i - estimado_i)^2 + \alpha \frac{1}{2N}\sum_{j=1}^{N} w_i^2$$

Con Lasso podemos obtener un modelo que generalice mejor. Nos ayuda con la selección de atributos de entrada, pues como se favorece que los coeficientes acaben valiendo 0, nos permite conocer qué atributos son relevantes. Su funcionamiento es mucho mejor cuando los atributos no están correlados entre sí.

Ridge sirve para disminuir los coeficientes y por tanto minimiza la correlación entre los atributos de entrada. Esto nos permite que nuestro modelo generalice mejor. A diferencia de Lasso, L2 tiene un mejor funcionamiento cuando los atributos son relevantes en su mayoría.

Modelos seleccionados para este problema

Tal y como se nos explicó en las clases prácticas, para seleccionar nuestros modelos e hiperparámetros vamos a hacer uso de K-fold cross validation. Esta técnica es utilizada para evaluar los resultados de un análisis estadístico y garantizar que son independientes de la partición entre los datos training y test. Su funcionamiento consiste en repetir y calcular la media aritmética obtenida de las medidas de evaluación sobre diferentes particiones (k particiones, lo más común es 5 o 10).

Vamos a usar esta técnica para validar los modelos generados ya que nuestro objetivo principal es la predicción y estimar la precisión de los modelos. Es una forma de predecir el ajuste de un modelo a un "hipotético" conjunto test cuando desconocemos el conjunto de datos test explícito.

La clase GridSearchCV de Scikit-Learn nos permite hacer uso de esta técnica. Una vez obtenida la mejor hipótesis, con esa hipótesis reentrenamos con el conjunto de entrenamiento, validamos con el conjunto de test y ya tenemos nuestro E_{out}.

****(Tanto estos como todos los parámetros usados se explicarán en su correspondiente apartado en la memoria de 'parámetros de funciones' CON MUCHO MÁS DETALLE)****

Como modelos emplearemos:

LinearRegression, es decir, la clase correspondiente de Scikit-Learn para la regresión lineal, ya que su implementación es trivial y podemos configurar ciertos parámetros para obtener resultados más concretos. En esta ocasión usaremos los parámetros por defecto.

SGDRegressor(max_iter=2500), es decir, la clase correspondiente de Scikit-Learn para el gradiente descendente estocástico ya que aparte de su simplicidad en la implementación, nos ofrece una amplia posibilidad en los parámetros con los que podemos jugar y obtener unos resultados más profundos y detallados.

Para la tasa de aprendizaje vamos a seleccionar tanto 'optimal' como 'adaptive. Para el primero de ellos significa que η = 1.0 / (α * (t+ t_0)) y para el segundo de ellos (adaptive), que inicia $\eta = \eta_0$ y va dividiéndolo por 5 si alcanzamos un cierto número de iteraciones sin disminuir el error (en caso de activar early_stopping en cuyo caso el número de iteraciones vendría dado por n_iter_no_change) o en otro caso a las dos épocas sin cambios.

Como regularización para SGD usaremos tanto Lasso (L1) como Ridge (L2) los cuales hemos comentado anteriormente.

El resto de parámetros son los parámetros por defecto de SGDRegressor.

Además, usaremos los modelos lineales de **Lasso y Ridge** también procedentes de Scikit-Learn con los parámetros por defecto (ver parámetros explicados al final de la memoria).

Ridge, un modelo lineal que aplica regularización penalizando la suma de los coeficientes elevados al cuadrado ($||\beta||_2^2 = \sum_{j=1}^{p} \beta_j^2$). Esta es la penalización l2 que usaremos más adelante en otros modelos y consigue reducir proporcionalmente el valor de los coeficientes del modelo

evitando que lleguen a 0. El nivel al que se le aplica la penalización viene determinado por λ. Este modelo resuelve un modelo de regresión donde la función de pérdida es la función lineal de mínimos cuadrados y la regularización está dada por la norma l2.

$$\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2 + \lambda\sum_{j=1}^{p}\beta_j^2 = \text{suma residuos cuadrados} + \lambda\sum_{j=1}^{p}\beta_j^2$$

Lasso, que es similar al modelo de Ridge, pero con la diferencia de que la penalización que aplica es la de la suma del valor absoluto de los coeficientes de regresión ($||\ \beta\ ||_1 = \sum_{j=1}^{p}|\beta|_j$). Esta penalización es l1 y también será aplicada a otros modelos como regularización para hacer que los coeficientes de lo predictores tiendan a cero. Como en Ridge, λ determina el grado en el que se aplica la penalización y un valor de 0 significa una aplicación nula.

$$\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2 + \lambda\sum_{j=1}^{p}|\beta_j| = \text{suma residuos cuadrados} + \lambda\sum_{j=1}^{p}|\beta_j|$$

AdaBoostRegressor, ya que como modelos recomendados en la práctica se nos ofrece boosting que, además de poder ser aplicado a varios estimadores base como perceptrón o SVM, se propone que el algoritmo base al que se le aplique sea árboles, el cual es el parámetro por defecto (base_estimator = DecisionTreeRegressor). Dentro del campo de los árboles con Boosting destacan AdaBoost, Gradient Boosting y Stochastic Gradient Boosting, que ya es un motivo de selección junto a la cantidad de hiperparámetros con los que cuenta, lo cual nos va a permitir buscar combinaciones que obtengan resultados muy buenos. Una desventaja de este modelo puede ser su tendencia al sobreajuste, pero se emplea un término de regularización conocido como learning rate que controla el ritmo al que aprende el modelo. Otras ventajas tenidas en cuenta para la elección del modelo dentro de este problema es su velocidad y vertalidad.

GradientBoostingRegressor, que es otro modelo que hace uso de Boosting sobre árboles que construye un modelo aditivo de una manera avanzada en cuanto a la etapa; permite la optimización de funciones de pérdida diferenciables arbitrarias. La selección de este modelo se debe a que es una generalización de AdaBoost y premite utilizar cualquier función de coste, lo cual nos dará versatilidad en la búsqueda de hiperparámetros óptimos.

HistGradientBoostingRegressor, es decir, otro modelo que utiliza el boosting explicado anteriormente. Es similar al anterior, pero con una nueva implementación experimental que le hace tener algunos hiperparámetros diferentes y mayor rapidez. Debido a esto podremos comprobar si la regularización es necesaria en este tipo de modelos que tienden al sobreajuste.

SVR, una versión de SVM adaptada a regresión. Esta es una de las técnicas no lineales propuestas para resolver el problema que transforma los datos en una característica de espacio dimensional más alto cuando el problema no es lineal a través de la función Kernel para hacer posible la separación. La elección de este modelo ha estado conducida porque SVR reconoce la presencia de no linealidad en los datos y proporciona un modelo de predicción competente, haciéndolo un modelo prometedor.

ElasticNet, es un término medio entre Ridge y Lasso. El término de regularización es una combinación simple de los términos de regularización de Ridge y Lasso, y puede controlar la proporción de mezcla r. Cuando r = 0, Elastic Net es equivalente a Ridge Regression, y cuando r = 1, es equivalente a Lasso Regression. Este modelo será utilizado para comprobar si es mejor un modelo de regularización u otro. Generalmente se prefiere Elastic Net a Lasso, ya que Lasso puede comportarse de forma errática cuando el número de funciones es mayor que el número de instancias de entrenamiento o cuando varias funciones están fuertemente correladas.

BayessianRidge, es un modelo que utilizaremos ya que son muy útiles cuando el conjunto de datos es pequeño o los datos están mal distribuidos (nos encontramos en el primer caso). El resultado de BayessianRidge se obtiene a partir de una distribución de probabilidad, en comparación con las técnicas de regresión regulares donde el resultado se obtiene simplemente a partir de un valor único de cada atributo.

PassiveAggressiveRegressor, es uno de los pocos 'algoritmos de aprendizaje en línea', en los que los datos de entrada vienen en orden secuencial y el modelo de aprendizaje automático se actualiza paso a paso, a diferencia del aprendizaje por lotes, donde se usa todo el conjunto de datos de entrenamiento a la vez. Se denominan pasivo-agresivos porque la parte pasiva, en caso de tener una predicción correcta, conserva el modelo y no realiza ningún cambio. Y la parte agresiva, en caso de tener una predicción incorrecta, realice cambios en el modelo. es decir, algún cambio en el modelo puede corregirlo. Nos ha parecido interesante incluirlo como modelo adicional ya que no es muy conocido y al llamarnos la atención su funcionamiento, queríamos ver cómo actuaba en nuestro problema.

KNeighboursRegressor, es un algoritmo basado en instancia de tipo supervisado de Machine Learning. Es un método sencillo que se usar para predecir valores en regresión. La K significa la cantidad de 'grupos' (clusters) que tenemos en cuenta en las cercanías para predecir. Además es un algoritmo que requiere mucha memoria y recursos de procesamiento (CPU) por lo que funciona mejor en datasets pequeños y sin una cantidad enorme de features, como es nuestro caso.

RandomForestRegressor, es un modelo formado por un conjunto de árboles de decisión individuales, cada uno entrenado con una muestra ligeramente distinta de los datos de entrenamiento generada mediante bootstrapping. La predicción de una nueva observación se obtiene agregando las predicciones de todos los árboles individuales que forman el modelo. Los árboles pueden manejar tanto predictores numéricos como categóricos sin tener que crear variables dummy o one-hot encoding, no se ven muy influenciados por outliers, no requieren

de una estandarización, son muy útiles para identificar de forma rápida y eficiente los predictores más importantes y tienen buena escalabilidad. Se escogerá como modelo proporcionado para realizar este proyecto por parte del profesorado.

MLPRegressor (Multilayer Perceptron), es un tipo de red neuronal artificial. Es un modelo que puede funcionar con regresión de valores objetivo únicos o múltiples. Un factor a tener en cuenta es el de escalar los datos previamente antes de usarlos para entrenar el modelo. Se escogerá también como modelo proporcionado para realizar ese proyecto por parte del profesorado.

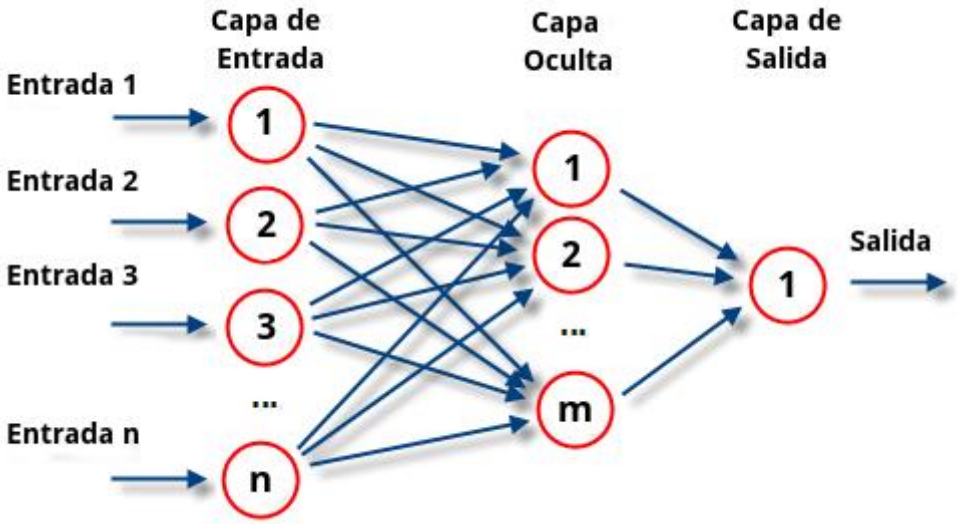


Imagen de la estructura de un peceptrón multicapa

Análisis de los resultados y conclusión final

Tras completar todo el proceso, es el momento de analizar y comprobar los resultados obtenidos:

MODELO	RMSE	MSE	MAE	R^2
SGD	4.8649	23.6671	3.7537	0.4428
GradientBoosting	2.0221	4.0891	1.5060	0.9037
Ridge	4.8639	23.6574	3.7564	0.4430
Lasso	4.8645	23.6637	3.7535	0.4429
ElasticNet	4.8637	23.6551	3.7575	0.4431
SVR	4.2023	17.6593	3.1689	0.5842
AdaBoost	4.1093	16.8864	3.3315	0.6024
HistGradientBoosting	2.3615	5.5766	1.7660	0.8687
LinearRegression	4.8647	23.6649	3.7534	0.4428
BayesianRidge	4.8642	23.6606	3.7550	0.4429
PassiveAggressive	5.7477	33.0364	4.2227	0.2222
KNeighbours	3.4383	11.8216	2.6777	0.7217
MLPRegressor	2.0903	4.3695	1.5462	0.8971

MSE es una de las métricas más simples y comunes de regresión.

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Donde y_i es el resultado real esperado e $\hat{y}$ es la predicción del modelo.

Cuanto mayor sea este valor, peor es el modelo. Nunca es negativo, ya que estamos cuadrando los errores de predicción individuales antes de sumarlos, pero sería cero para un modelo perfecto.

Esta métrica no es de las más certeras. Si hacemos una predicción muy mala, la cuadratura empeorará aún más el error y puede sesgar la métrica para sobreestimar la maldad del modelo. Este es un comportamiento particularmente problemático si tenemos datos ruidosos. Un modelo muy bueno puede tener un MSE alto en esa situación. Por lo que es difícil juzgar qué tan bien modelo está realizando.

Con MAE no ocurre esto:

$$\mathrm{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

Esta métrica es que penaliza errores enormes que no tan mal como lo hace MSE**.** Por lo tanto, no es tan sensible a los valores atípicos como el error cuadrático medio.

Para el coeficiente de determinación R^2:

$$R^2 = 1 - \frac{\text{MSE(model)}}{\text{MSE(baseline)}}$$

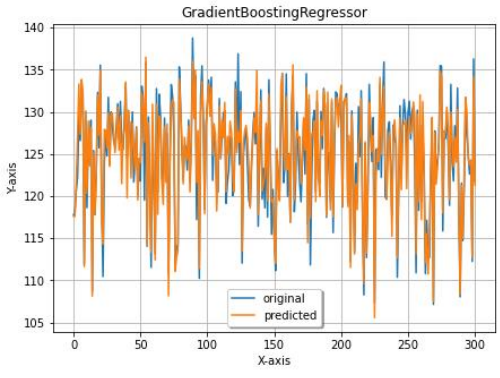
Este coeficiente es otra medida que podemos usar para evaluar un modelo y cuanto mayor sea su valor (con rango [-∞, 1]), mejor será nuestro modelo. Un error cercano a 1, en nuestro caso 0.9 aproximadamente para el mejor modelo, indica un modelo con un error cercano a cero, y un valor cercano a 0 indica un modelo cercano a la línea de base (baseline).

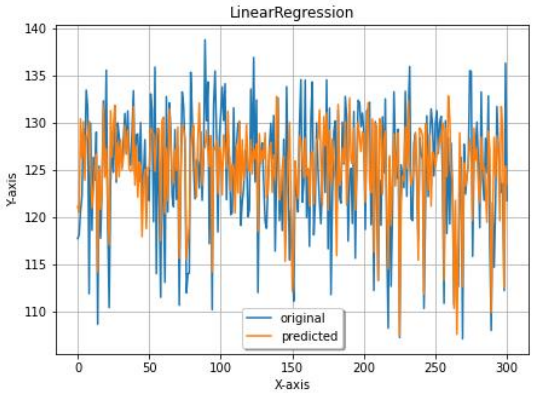
La raíz del error cuadrático medio (RMSE = $\sqrt{MSE}$) no deja de ser, como ya comentamos previamente, una métrica utilizada para reforzar nuestras conclusiones, de forma que si MSE(A) > MSE(B) ⇔ RMSE(A) > RMSE(B).

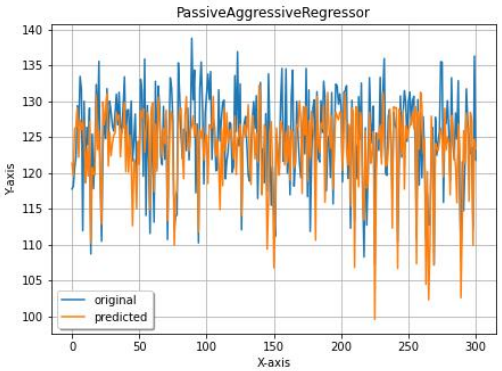
A vista de los resultados, tanto GradientBoosting como RandomForest seguidos de HistGradientBoosting y KNeighbors son los mejores modelos para nuestro problema, son los que mejor predicen. Los modelos lineales son los que ofrecen los peores resultados.

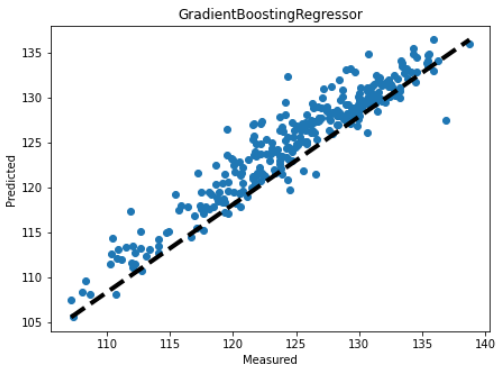
Para simplificar esta parte debido al gran número de modelos estudiados, vamos a comentar las gráficas que representan tanto la predicción frente a los valores de test, así como sus distribuciones de error para el mejor y los peores modelos (a vista de los datos) para ver las diferencias entre ellas:

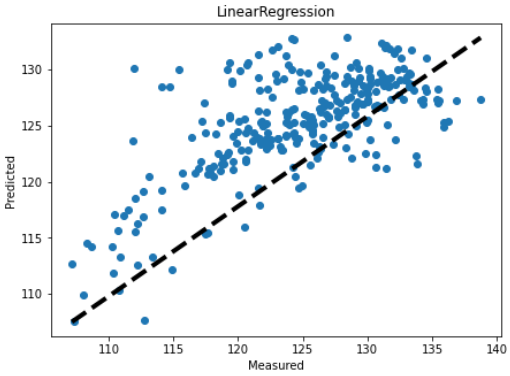
GradientBoosting vs LinearRegression vs PassiveAggresiveRegressor:

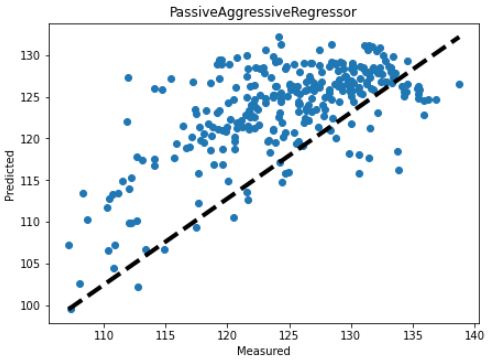


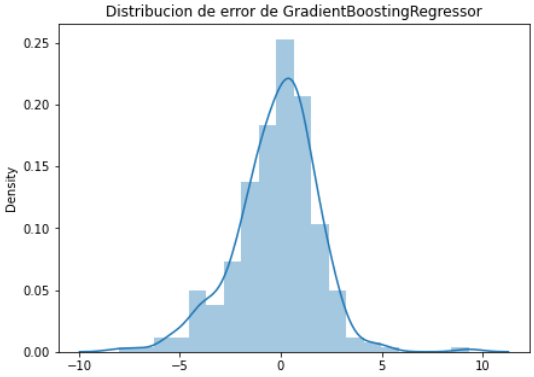


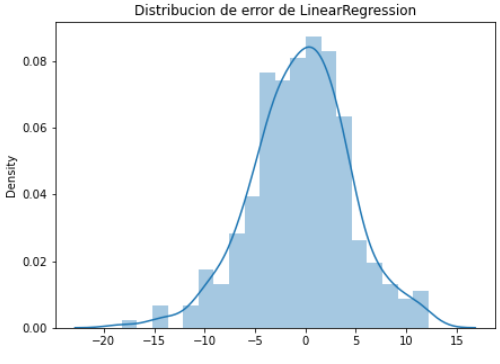


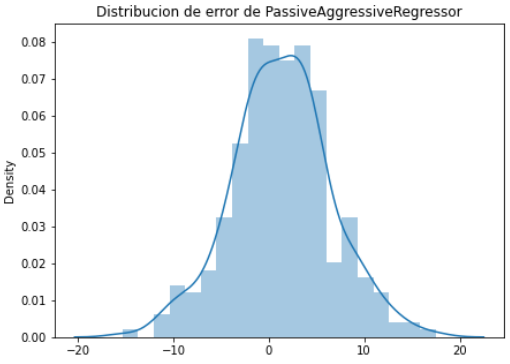








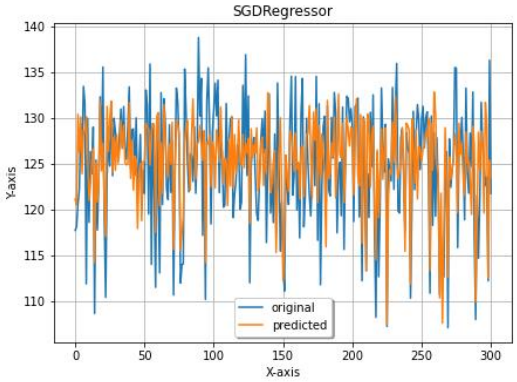


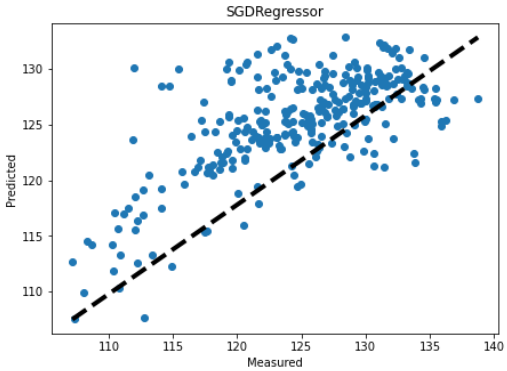


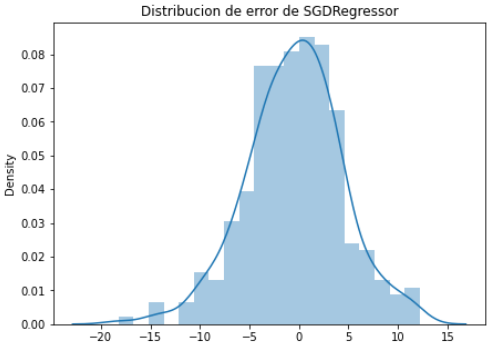
Gráficamente es fácil observar que la predicción realizada por GradientBoosting es mucho más precisa que la que realiza LinearRegression o PassiveAggressive. Como esperábamos, los modelos lineales van a ofrecer peores resultados que los modelos no lineales. Observando las gráficas de las distribuciones de los errores, las alturas de las barras siguen de cerca la forma de la línea de distribución ajustada, entonces podemos decir que los datos se ajustan adecuadamente a la distribución.

Vamos a observar el resto de gráficas correspondientes a cada modelo y poder dar una conclusión final, así como conocer el mejor modelo para nuestro problema mediante el uso de GridSearchCV:

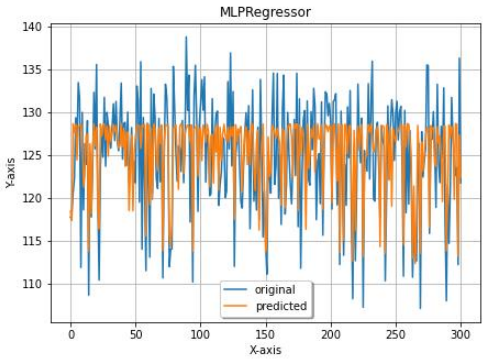
SGD:

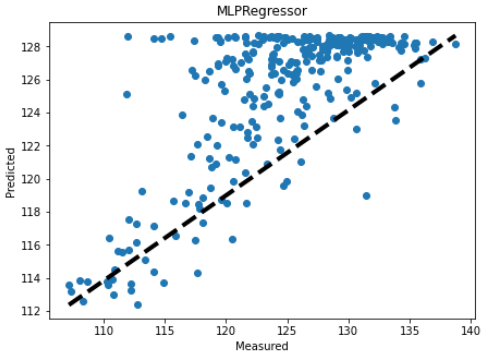


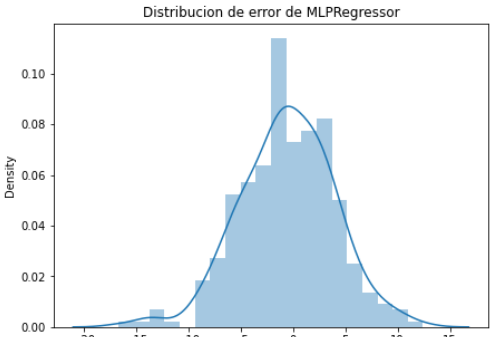




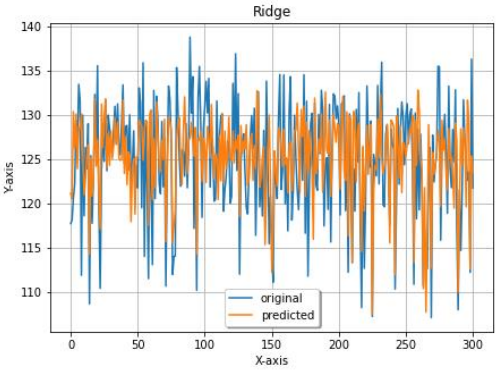
MULTILAYER PERCEPTRON (2H + 1s):

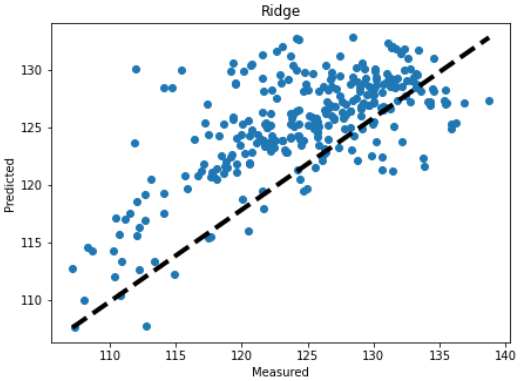


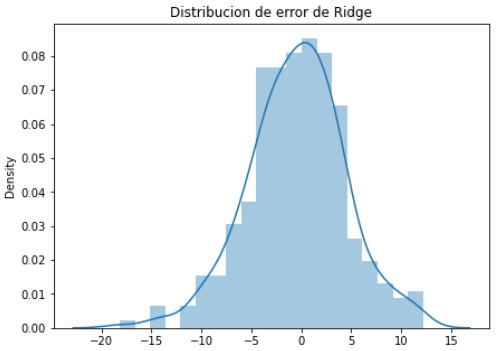




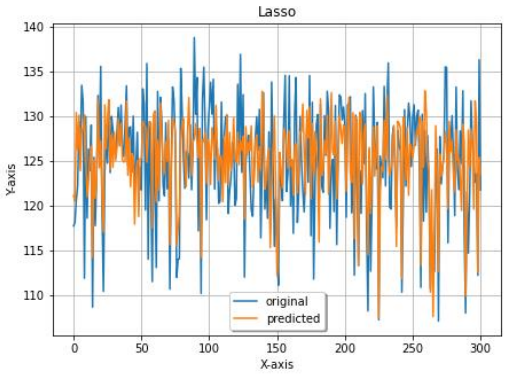
RIDGE:

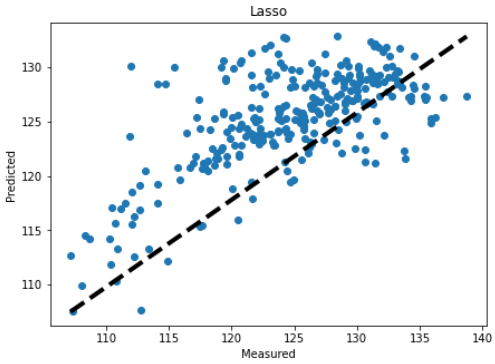


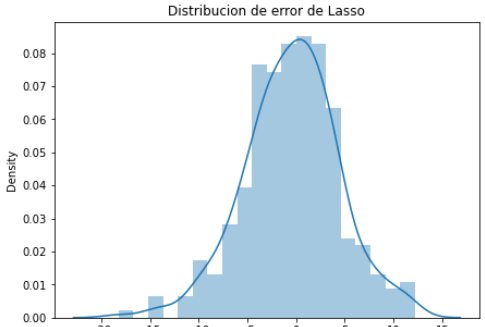




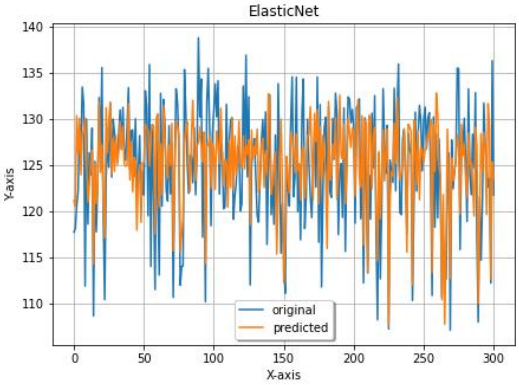
LASO:

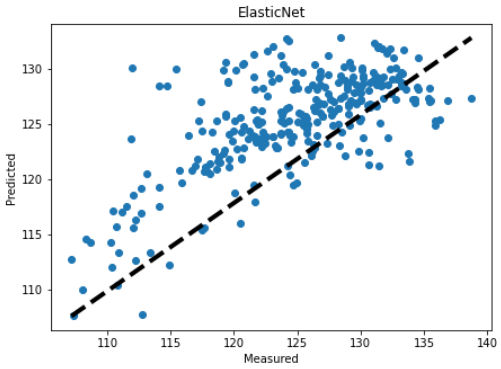


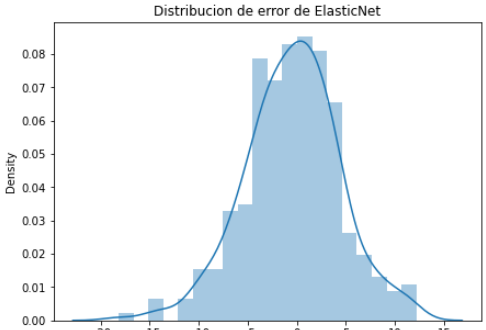




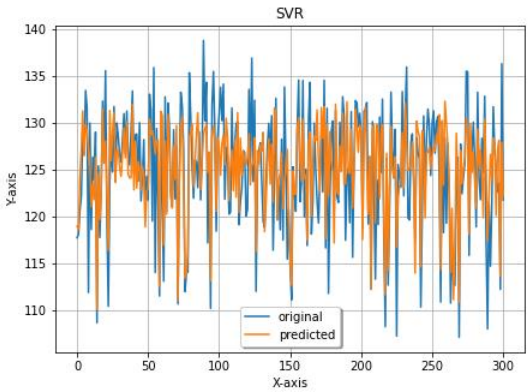
ELASTIC NET:

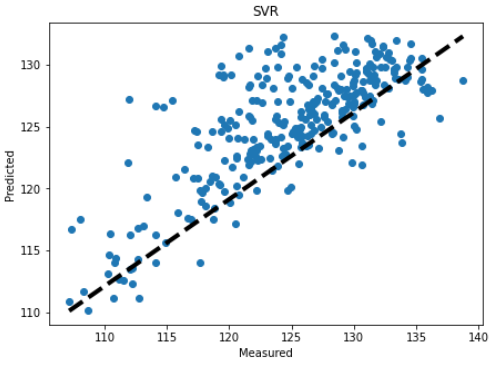


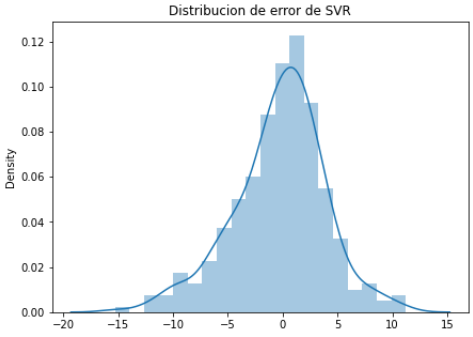




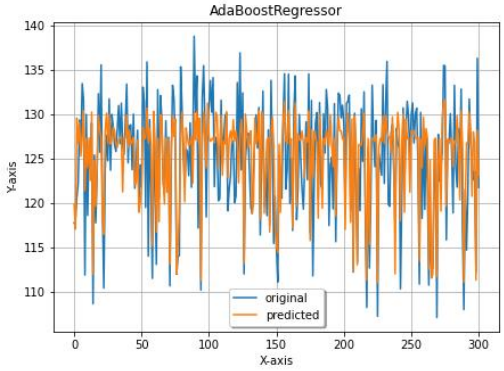
SVR:

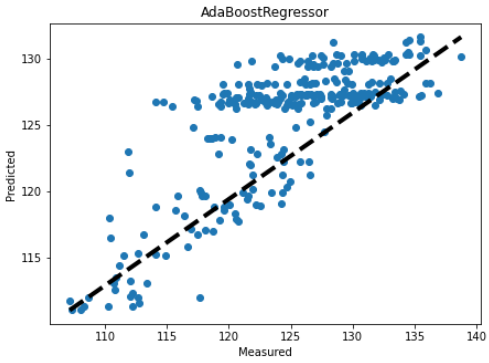


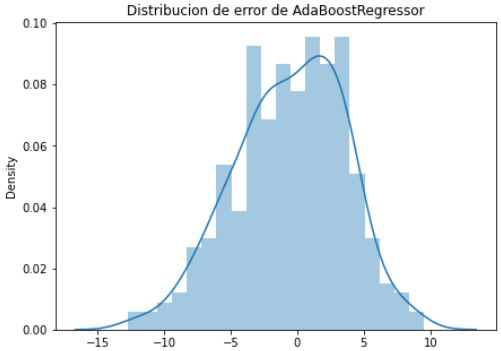




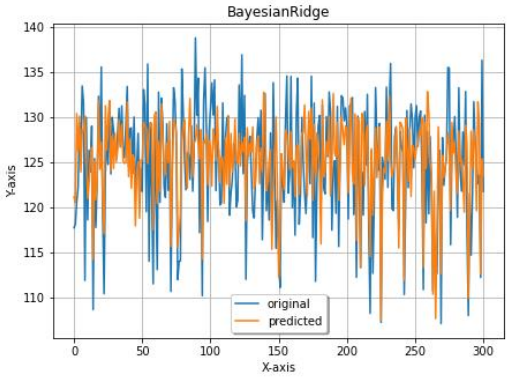
ADABOOST:

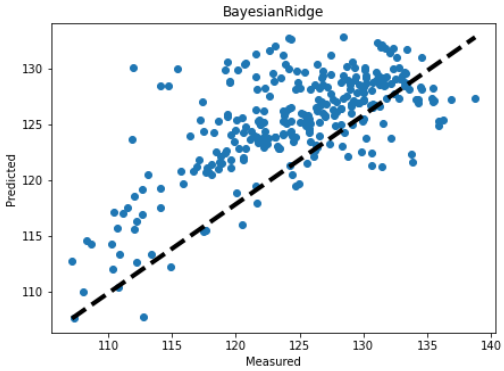


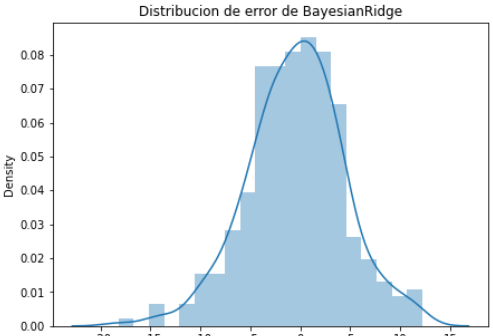




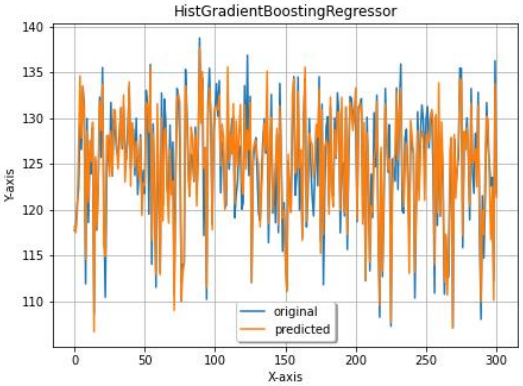
BAYESIAN RIDGE:

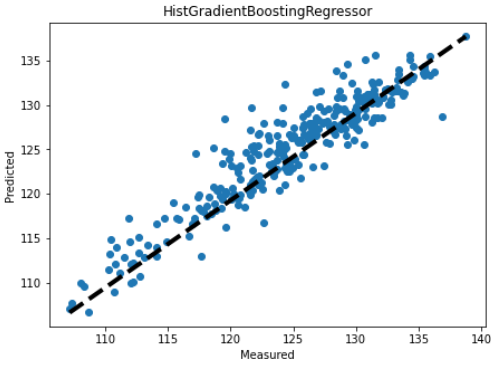


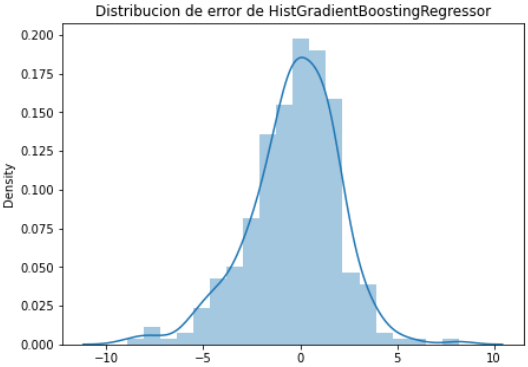




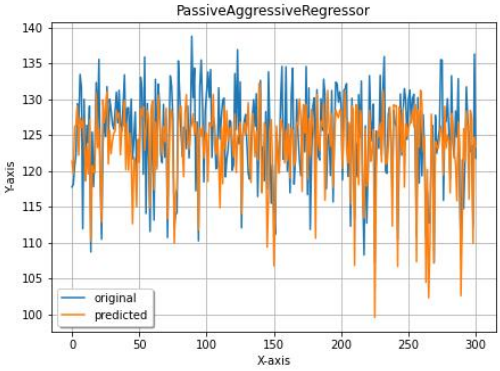
HIST GRADIENT BOOSTING:

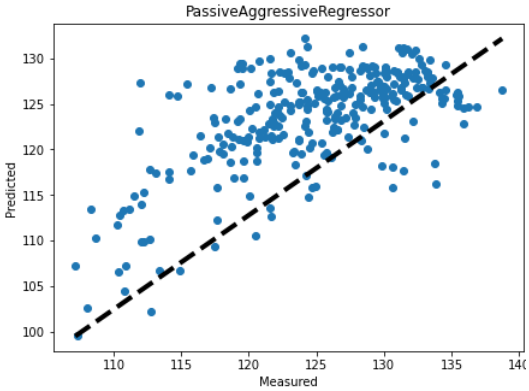


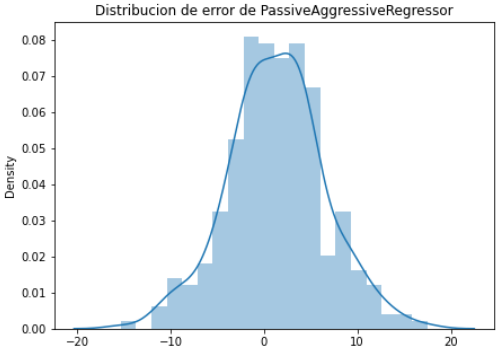




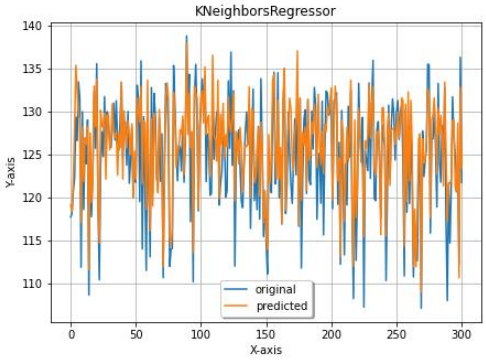
PASSIVE AGGRESSIVE REGRESSOR:

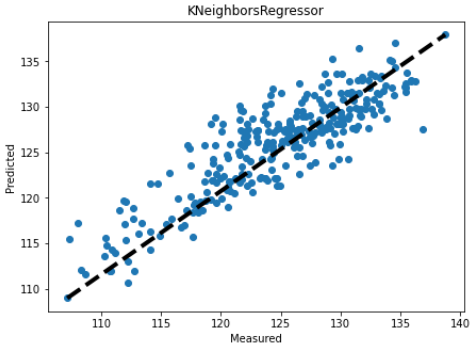


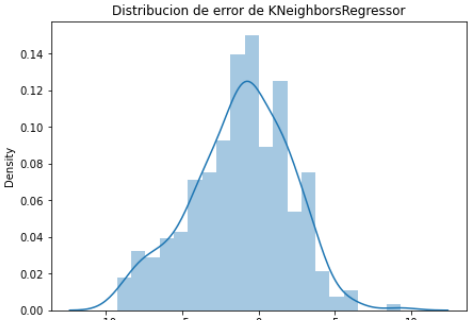




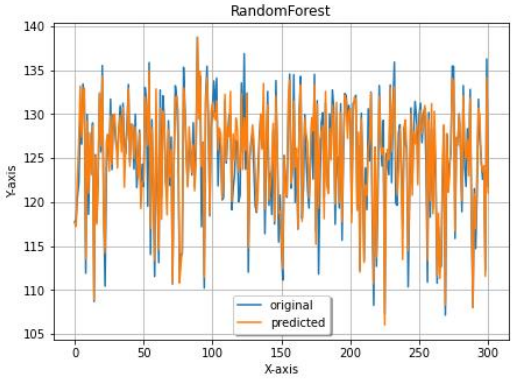
K-NEIGHBORS REGRESSOR:

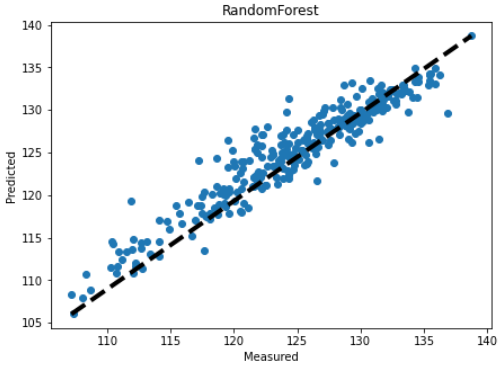


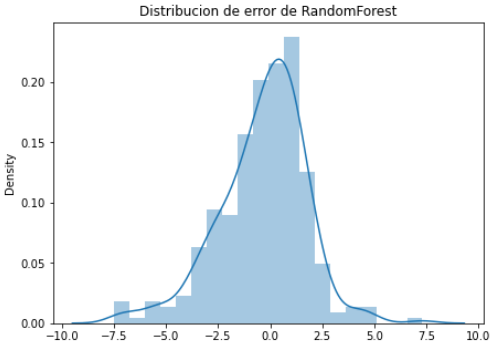




RANDOM FOREST:







Como conclusión obtenida tras ver estas gráficas, podemos decir que los modelos lineales han sido los que nos han proporcionado los peores resultados (Lasso, Ridge, Linear Regression, SGD, Bayesian Ridge, PassiveAggresive). Esto tiene cierta coincidencia con nuestra idea inicial. Tras la visión inicial de los datos y tras realizar un preprocesado que aumentaba la complejidad de la clase de funciones, era de esperar que los modelos lineales sean aquellos que den los peores resultados de predicción sobre nuestro problema. En general, observando las gráficas de las distribuciones de los errores, podemos decir que nuestros modelos han ajustado los datos adecuadamente a la distribución.

Para obtener el mejor modelo, tal y como ya comentamos, se he empleado el método de k-fold cross validation. La clase GridSearchCV de Scikit-Learn nos permite hacer uso de esta técnica. Una vez obtenida la mejor hipótesis, con esa hipótesis reentrenamos con el conjunto de entrenamiento, validamos con el conjunto de test y ya tenemos nuestro E_{out}. Le indicaremos todos los modelos y los parámetros a probar y evaluará el rendimiento en función de los segundos mediante validación cruzada maximizando (en caso de utilizarlo) una score concreta. En nuestra ocasión utilizaremos como scoring 'neg_mean_squared_error' ya que como buscamos minimizar el error, lo equivalente es maximizar el valor negativo de MSE y luego aplicarle el valor absoluto al resultado obtenido.

Resultado de best_params_ y best_score_ procedente de GridSearchCV:

```
{'model': GradientBoostingRegressor(max_depth=5, max_features='auto', n_estimators=500),
'model__learning_rate': 0.1, 'model__max_depth': 5, 'model__n_estimators': 500}
```

MEJOR ERROR OBTENIDO EN GridSearchCV: 5.258971950363181 (MSE)

Tal y como observábamos con los datos estadísticos, la validación cruzada nos ha confirmado que el mejor modelo para nuestro problema de entre todos los proporcionados ha sido Gradient Boosting Regressor con esa combinación de parámetros concreta ya que no solamente nos vale seleccionar un modelo bueno, sino que tiene que ir acompañado de los hiperparámetros que lo hagan el mejor.

Con el fin de no influenciar el resultado con el conocimiento obtenido sobre los datos tras preprocesarlos, el proceso de selección y de tratamiento de datos se realizará secuencialmente en el mismo paso.

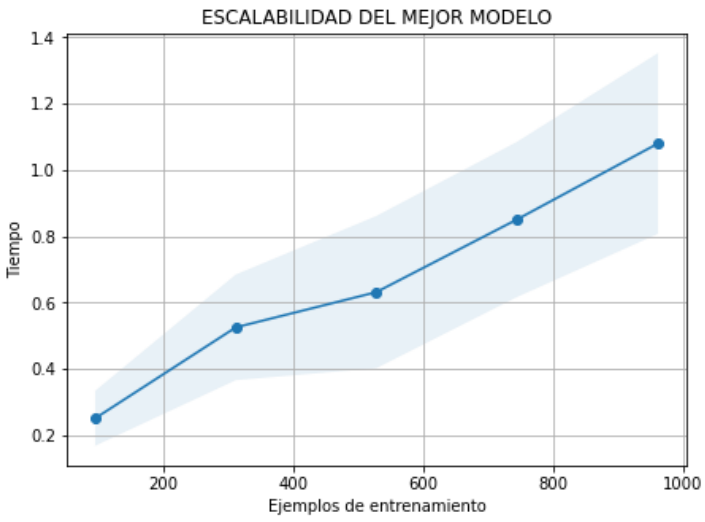
Es cierto que era imposible determinar el modelo y los hiperparámetros correctos previo a la experimentación, pero sí que contábamos con información que nos podía hacer esperar un resultado u otro. Para empezar, la observación de los datos, aunque no debe ser el motivo de adaptación de los modelos para evitar el data snooping, siempre puede aportarnos información sobre qué tan difícil lo pueden tener algunos modelos frente a otros. En este caso, la complejidad del conjunto nos podía hacer pensar que los modelos lineales iban a tener un

desempeño menos correcto, por lo que haber seleccionado como hipótesis Lasso, Ridge, ElasticNet, BayesianRidge, PassiveAgressiveRegressor, SGDRegressor o LinearRegression iba a ser seña de un posible error en el tratamiento de los datos y el desarrollo del experimento.

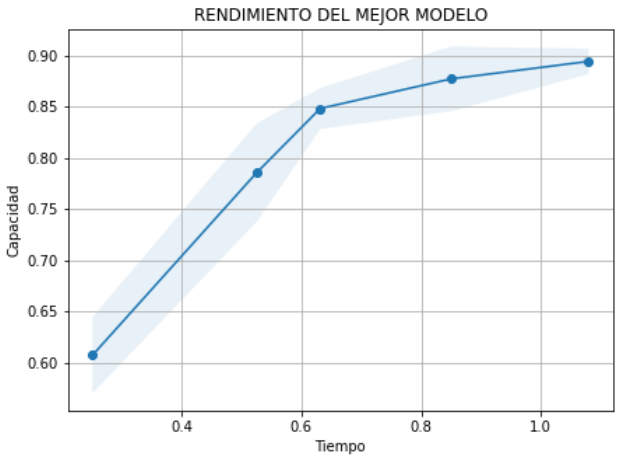
Esto dejaba como favoritos los modelos no lineales, entre los que son reconocidos popularmente como buenos modelos las redes neuronales y los que aplican Boosting. Igual con más tiempo de ejecución HistGradientBoostingRegressor podría haber obtenido mejores resultados, pero la opción de probarlo con regularización y sin ella hacía que se duplicasen las combinaciones de sus hiperparámetros y el número de iteraciones (equivalente al número de estimadores en el resto de modelos de Boosting) se prefijó a un punto intermedio, lo cual consideramos que ha condicionado mucho sus resultados. Por otra parte, AdaBoostRegressor no permite la opción de modificar la profundidad que alcanzan los árboles en su búsqueda de resultados, por lo que Gradient Boosting volvía a contar con un espacio de soluciones más amplio frente a este modelo, por lo que no sorprende que haya sido el mejor modelo.

Curvas de aprendizaje

Para el uso de gráficas sobre las curvas de aprendizaje será necesario el uso de la función de scikit learn sklearn.model_selection.learning_curve que determina el entrenamiento con cross validation y sus puntuaciones haciendo uso de conjuntos de entrenamiento de diferentes tamaños. Internamente divide el conjunto de datos k veces en datos de prueba y entrenamiento y calculará una puntuación por cada caso, haciendo un promedio de todas ellas al terminar. Esta herramienta es útil para obtener información como el beneficio que obtenemos al aumentar la cantidad de datos de entrenamiento o si el estimador se ve afectado por un error de varianza o de sesgo.

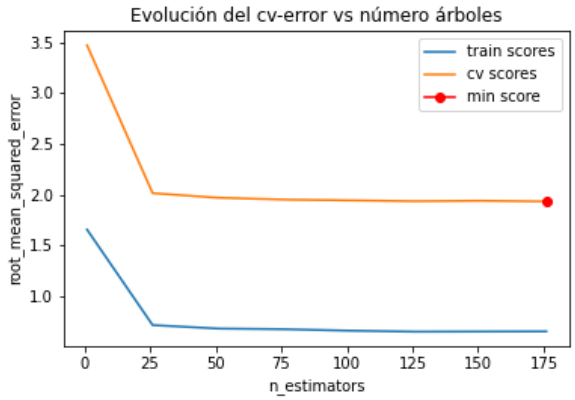


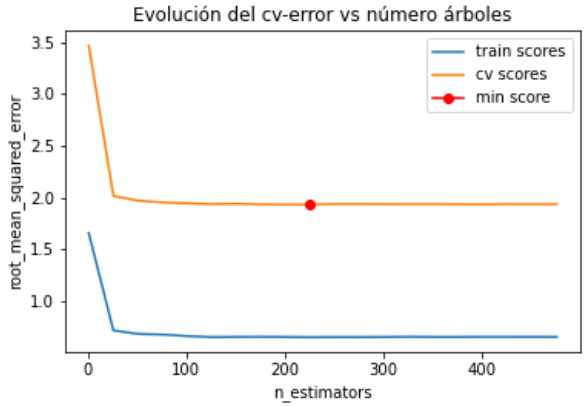
En esta gráfica lo que tenemos en cuenta es la escalabilidad del mejor modelo que hayamos obtenido que, en nuestro caso, es GradientBoostingRegressor. Medimos la escalabilidad como la relación entre el tamaño de los ejemplos de entrenamiento y el tiempo que tarda en realizarlos. Como vemos en la gráfica, el resultado es el esperado, con un aumento progresivo en el tiempo de ajuste conforme el tamaño incrementa. Un aumento excesivamente radical, como el exponencial, por ejemplo, significaría que no podemos aplicar el modelo a problemas con un conjunto de datos considerable porque no sería rentable en términos de tiempo.

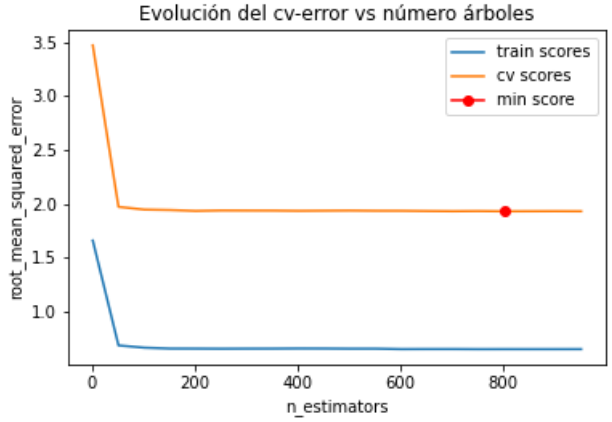


Por otra parte, utilizamos otra gráfica obtenida de learning_curve que mida el rendimiento del modelo. Este se ve reflejado en la relación de capacidad y tiempo. Como es lógico, el comportamiento del mejor modelo tiene un comportamiento logarítmico. Esto es debido a que cuanto mayor tiempo de ejecución, mejor capacidad predictiva alcanzará, pero llegará un punto en el que la capacidad se estabilice y se acerque lo suficiente al máximo como para que el aumento de tiempo no genere mejora.

Obtención de estimadores para Random Forest







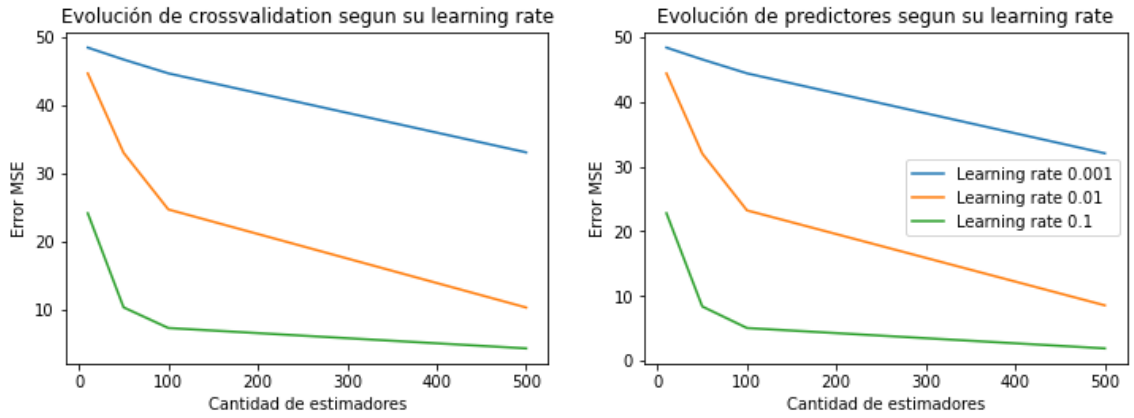
El experimento realizado para obtener el número de árboles para RandomForest adecuado consiste en la comparación de resultados obtenidos utilizando diferentes cantidades. Un primer ejemplo ha utilizado valores en el rango [1, 200] con diferencia de 25 estimadores cada vez (1, 26, 101, …, 176) y el mejor resultado se ha obtenido con el valor 176, que coincide con el más alto probado.

Este resultado nos condujo a un segundo experimento para probar si siempre va a ser mejor opción coger el máximo número de estimadores posibles. Para este intento simplemente se aumentó la cantidad de árboles que podía alcanzar, probando valores en el rango [1, 500] dando saltos de 25 igualmente. En este caso, y contradiciendo la hipótesis que pensábamos, el

mejor valor se obtuvo con 226 árboles aun habiendo probado con valores mayores como 251, 276, etc.

A pesar de contar con una naturaleza estocástica que podía hacer variar el número idóneo de estimadores, los dos resultados nos podrían conducir a pensar que el valor de árboles acertado oscilará alrededor de 200, por lo que la última comprobación consistirá en aumentar aún más la cota superior del intervalo y permitiéndole llegar hasta 1000 estimadores.

Por temas de eficiencia, los saltos se aumentaron a 50 unidades entre una prueba y otra, siendo los datos experimentados 1, 51, 101, …, 951. Para nuestra sorpresa, el valor correcto fue bastante mayor de lo esperado, pero confirmaba que no siempre el número más alto de estimadores es el que mejores resultados dan, por lo que no tenemos que abarcar en nuestro modelo definitivo los hiperparámetros que formen experimentos lentos con el único fin de tener muchísimos árboles. Esta falta de necesidad por perseguir altas cantidades de árboles nos condujo a la elección de [20, 100, 200, 300, 400, 500, 600, 1000] como el intervalo definitivo de árboles por el que nos moveremos al probar RandomForest, entre otros hiperparámetros.



Con estas gráficas queremos demostrar cómo es el comportamiento del mejor modelo seleccionado según el valor del hiperparámetro learning rate y este será comparado con la evolución del error de los predictores. Para la elección del modelo se ha necesitado de la ejecución del programa varias veces con el fin de obtener con cierta certeza cual iba a ser el mejor modelo aplicable a esta base de datos.

El modelo que empíricamente ha resultado ser el mejor es GradientBoostingRegressor y este se someterá a 3 experimentos reflejados en la gráfica de la izquierda. Se medirá el error que obtiene con un learning_rate de 0.001 aplicado a un número diferente de estimadores ([10, 50, 100, 500]). Esta corresponde a la línea azul que decrece más lentamente. Tras esto, se aplicará el mismo experimento de iterar por el número de estimadores calculando el error, pero con diferentes valores de learning_rate, siendo el naranja 0.01 y el verde 0.1

Seguidamente, este mismo experimento se realizará sin ser aplicado a ningún modelo, sino obteniendo el error de entrenamiento con el uso de predictores. A pesar de que parecen dos gráficas iguales, los puntos se encuentran a alturas ligeramente diferentes y una prueba de ello son los valores obtenidos:

```
Valores de predictores:
  [22.793689091227215, 8.415384724963106, 5.087619962447733, 1.944344117881575]
Valores del modelo:
  [24.145580187672838, 10.361006711500222, 7.336856318920388, 4.348599571545731]
```

Igualmente, aun viendo las diferencias, la similitud de ambas gráficas deja en claro que el modelo seleccionado es buena solución en cuanto al error que obtiene.

En ambos casos vemos que cuanto más crece learning rate, más inclinada es la pendiente descendente y, por tanto, más rápido aprende, pero antes tiende al sobreajuste. Es por esto por lo que se busca el equilibrio entre tiempo y coste computacional (a menos valor de learning rate menos sobreajuste, pero más lentitud en el aprendizaje).

Parámetros utilizados en el problema:

Hay parámetros por defecto que no van a ser comentados ya que la gran mayoría de ellos son útiles cuando se utiliza una configuración concreta de algunos parámetros. Vamos a comentar principalmente por qué utilizamos las funciones por defecto (en caso de hacerlo) y por qué realizamos las modificaciones pertinentes.

*class sklearn.preprocessing.**MinMaxScaler**(feature_range=0, 1, *, copy=True, clip=False)*

MinMaxScaler(): se usa por defecto. Queremos que el rango de los valores se encuentre entre 0 y 1 tal y como hemos comentado en el preprocesado de los datos. El parámetro copy se pone a False cuando se intenta evitar una copia y en su lugar se hace un escalado inplace. Igualmente es un parámetro que no garantiza que funcione siempre inplace y puede que devuelva una copia a pesar de tenerlo en False. No cambiamos estos valores ya que son los que queremos utilizar para nuestro conjunto de datos. La transformación se realiza de la siguiente forma:

X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))

X_scaled = X_std * (max - min) + min

Donde min y max corresponden con el rango de valores feature_range.

class sklearn.decomposition.**PCA**(*n_components=None*, *, *copy=True*, *whiten=False*, *svd_solver ='auto'*, *tol=0.0*, *iterated_power='auto'*, *random_state=None*)

PCA(0.9): lo único que nos interesa modificar de esta función es el número de componentes a mantener. Si esto no se modifica, se mantendrían todos los componentes (por defecto) y lo que nos interesa es realizar una reducción de la dimensionalidad. Si el parámetro svd_solver se encuentra en 'auto' significa que el solver se selecciona por una política predefinida basada en X.shape y n_components. El parámetro de la tolerancia no es necesario modificarlo ya que solo es útil cuando se escoge svd_solver = 'arpack'. Lo mismo ocurre con el parámetor iterated_power, que solo es útil cuando se escoge svd_solver = 'randomized'.

class sklearn.preprocessing.**PolynomialFeatures**(*degree=2*, ***, *interaction_only=False*, *include_bias=True*, *order='C'*)

PolynomialFeatures(): indicamos que queremos una transformación cuadrática de nuestros datos (ya lo hacemos por defecto).

sklearn.model_selection.**train_test_split**(**arrays*, *test_size=None*, *train_size=None*, *random_state=None*, *shuffle=True*, *stratify=None*)

train_test_split(x, y, test_size=0.2): indicamos los arrays que queremos separar, en nuestra ocasión separamos x e y en train y test y además indicamos que el 20% se quede en test por lo que el 80% restante quedará en train. Como vemos, por defecto shuffle=True por lo que esta función no solo separa los datos, sino que también los mezcla aleatoriamente para evitar desbalanceos.

class sklearn.linear_model.**Lasso**(*alpha=1.0*, ***, *fit_intercept=True*, *normalize=False*, *precompute=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Lasso(max_iter = 2500, tol=5, alpha = {1.00000000e-05, 2.33572147e-05, …, 4.28133240e+01, 1.00000000e+02}): Con la aplicación de este modelo, Ridge y ElasticNet, la obtención del mejor significa encontrar el valor óptimo de regularización lambda. Esta lambda se representa con el parámetro Alpha en los tres modelos, por lo que el principal objetivo es encontrar el valor con el que se obtienen mejores resultados. Esto nos lleva a que el primer parámetro que modificaremos será Alpha que, por defecto, vale α=0.1. Los alphas entre los que experimentaremos son 20 valores logarítmicamente separados entre 10^{-5} y 10^{2} con la ayuda de la función logspace(-5, 2, 20).

La documentación nos pide utilizar StandardScaler antes de ajustar en un estimador con el parámetro normalize=False en caso de querer estandarizar. Dos cosas que modificaremos será el número de iteraciones que, por defecto, es algo bajo y pasarán a ser 2500 frente a los 1000 que partían siendo y el valor de tolerancia de la función. Este es el parámetro 'tol' y representa la tolerancia para la optimización, por lo que el modelo funcionaría hasta obtener un valor mayor o menor que este umbral (dependiendo de la función objetivo que estemos maximizando o minimizando). El problema encontrado con este parámetro que, a priori, iba a quedar por defecto es que el valor más bajo que alcanzamos, comprobado a raíz de la experimentación y la observación de los resultados, oscila sobre 4, por lo que jamás alcanzamos los 0.0001 establecidos por defecto. Esto provocaba la salida de unos warnings que dificultaban la observación de la salida por pantalla, por lo que 'tol' se ha establecido a 5 para que se detenga antes de cumplir el máximo de iteraciones.

class sklearn.linear_model.**Ridge**(*alpha=1.0*, ***, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *solver='auto'*, *random_state=None*)

Ridge(max_iter = 2500, tol=5, alpha = {1.00000000e-05, 2.33572147e-05, …, 4.28133240e+01, 1.00000000e+02}): como ocurre con Lasso, lo único que vamos a cambiar con respecto de los parámetros por defecto es el valor de Alpha con valores en el mismo rango (logspace(-5, 2, 20)), el valor de la tolerancia para la optimización y la cantidad de iteraciones del modelo para que haga 2500.

*class sklearn.linear_model.**ElasticNet**(alpha=1.0, *, l1_ratio=0.5, fit_intercept=True, normalize=False, precompute=False, max_iter=1000, copy_X=True, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic')*

ElasticNet(max_iter = 2500, tol=5, alpha = {1.00000000e-05, 2.33572147e-05, …, 4.28133240e+01, 1.00000000e+02}, l1_ratio = {0, 0.2, 0.4, 0.6, 0.8, 1}): En cuanto a los 3 primeros parámetros modificados en el modelo, la explicación es similar a Lasso y Ridge, ya que buscamos obtener el valor de Alpha para el que el resultado es el mejor, las iteraciones de partida pueden ser pocas y necesitamos una tolerancia adaptada a que el programa no colapse en avisos sobre el número de iteraciones. El nuevo parámetro modificado es l1_ratio que representa el peso asignado a cada tipo de regularización de las que aplica combinadas, que son Lasso y Ridge. Cuando este parámetro vale 1 se aplica un Lasso puro y cuando vale 0 se aplica únicamente Ridge. Sabiendo esto, podemos intuir cómo funciona la modificación de este parámetro. Si se encuentra estrictamente entre 0 y 1, el peso de la regularización aplicada será proporcional a su valor. Este parámetro se modifica incrementando progresivamente desde el mínimo (0 / Ridge) hasta el máximo (1 / Lasso) con la intención de ver qué regularización tiene un mejor efecto en este modelo. De haber salido todo bien, el valor de l1_ratio saldrá más próximo a aquel modelo que independientemente dio mejores resultados en el conjunto de datos.

*class sklearn.ensemble.**AdaBoostRegressor**(base_estimator=None, *, n_estimators=50, learning_rate=1.0, loss='linear', random_state=None)*

AdaBoostRegressor(n_estimators = {10, 50, 100, 500}, learning_rate ={0.001, 0.01, 0.1}): Cuando boosting es aplicado a árboles, como es nuestro caso tras la especificación del enunciado, el número de estimadores (n_estimators) es referido al número de árboles que va a usar el modelo. El motivo por el que vamos a modificar este parámetro es que el número de árboles es un hiperparámetro crítico en el riesgo de sobreajuste conforme este aumenta. Los valores que probaremos son [10, 50, 100, 500]. A partir de estos valores, el tiempo de ejecución puede ser demasiado elevado y cuando alcanzamos una cantidad de estimadores concreto, el error se estabiliza y no conseguimos mejora con el aumento del número de árboles, por lo que encontrar un equilibrio es el objetivo de no seguir aumentando los estimadores. El otro parámetro más importante junto a este es el learning_rate porque controla cómo de rápido aprende el modelo y, de nuevo, con ello, el riesgo de llegar al overfitting. Son dos parámetros interdependientes, cuanto menor es el learning_rate, más árboles se necesitan para alcanzar buenos resultados, pero menor es el riesgo de sobreajuste. Realizaremos la comprobación dando valores de 0.001, 0.01 y 0.1 que son valores estándar para este parámetro, pero con pequeñas variaciones. En adaboost dejamos por defecto el resto de valores, aunque podríamos cambiar los, que es la función de pérdida que se usará cuando se actualicen los pesos después de cada iteración. Por temas de complejidad, lo dejaremos en lineal.

*class sklearn.ensemble.**GradientBoostingRegressor**(*, loss='ls', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, min_impurity_split=None, init=None, random_state=None, max_features=None, alpha=0.9, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0)*

***GradientBoostingRegressor*(n_estimators = {10, 50, 100, 500}, learning_rate ={0.001, 0.01, 0.1}, max_depth = {1, 3, 5}):** Para este modelo, las consideraciones acerca del número de árboles y de los valores de aprendizaje son las mismas que para AdaBoostRegressor. Vamos a modificar también max_depth con distintos valores. Este parámetro se refiere a la profundidad que puede alcanzar cada árbol. Como es de suponer, debemos medir bien los valores pues, a pesar de que cuanta más profundidad, mejores resultados parecería obtener, puede tender al sobreajuste y a un alto coste computacional. Debido a esto, los valores suelen ser bajos para que cada árbol solamente aprenda una pequeña parte de la relación entre predictores y variable respuesta. Estos valores serán [1, 3, 5]. Por defecto son 3, así que probaremos una profundidad mayor y una menor con el objetivo de ver cuál es mejor.

*class sklearn.ensemble.**HistGradientBoostingRegressor**(loss='least_squares', *, learning_rate=0.1, max_iter=100, max_leaf_nodes=31, max_depth=None, min_samples_leaf=20, l2_regularization=0.0, max_bins=255, categorical_features=None, monotonic_cst=None, warm_start=False, early_stopping='auto', scoring='loss', validation_fraction=0.1, n_iter_no_change=10, tol=1e-07, verbose=0, random_state=None)*

***HistGradientBoostingRegressor* (max_iter = 250, learning_rate ={0.001, 0.01, 0.1}, max_depth = {1, 3, 5}, l2_regularization = {0,1}):** Para este modelo, las consideraciones acerca del número de árboles, de los valores de aprendizaje y de la profundidad son las mismas que para GradientBoostingRegressor con el objetivo de que el ritmo de aprendizaje sea el mismo para todos los modelos que hagan uso de boosting. Por lo pronto, la función de error usada en el cálculo del peso de los puntos seguirá sin ser relevante para nosotros y mantendremos la que viene por defecto (least_squares). Los niveles de profundidad a los que se le someterá serán los mismo que GradientBoosting ya que, aunque se podría aumentar debido a su rapidez, vamos a probar las regularizaciones implicando un aumento de tiempo considerable como para incrementar además la profundidad de los árboles. Se va a perjudicar el modelo lo suficiente como para que no sea adecuado y por ello la profundidad de los árboles se mantendrá en el mismo rango que el resto. Además, este será el modelo de Boosting indicado para probar todas esas combinaciones con y sin regularización porque los modelos de Boosting tienden a sobreajuste y podríamos comprobar si esta regularización es necesaria. Lo único que se limitará será el número de iteraciones, que es equivalente al número de estimadores de los dos modelos anteriores. Para este modelo el valor será 250, un punto intermedio de los 500 estimadores que se llegan a probar en los otros modelos. Se hace para reducir la cantidad de combinaciones que hay entre sus parámetros y que el coste no se vaya de las manos.

*class sklearn.svm.**SVR**(*, kernel='rbf', degree=3, gamma='scale', coef0=0.0, tol=0.001, C=1.0, epsilon=0.1, shrinking=True, cache_size=200, verbose=False, max_iter=- 1)*

***SVR* (kernel = {poly , rbf}, gamma = {scale, auto}, C = {0.1, 0.2, 0.4, 0.7, 0.9, 1}):** El primer parámetro que vamos a tocar es “kernel” que especifica el tipo de kernel que se utilizará en el algoritmo, ya que se nos recomienda el uso de dos, RBF-Gaussiano o polinomial. Como no

tenemos motivo empírico que nos haga decantarnos por uno u otro antes de la ejecución, vamos a poner que pruebe ambos y sea GridSearch el que delibere qué kernel obtiene mejor resultados. Acompañado del tipo de kernel tenemos el parámetro "gamma" que es el coeficiente del núcleo que debe ser mayor que 0. La interpretación de este parámetro es que cuanto mayor sea, menor será el vector de soporte y viceversa, lo cual tiene un impacto relevante en la velocidad de entrenamiento y predicción. Los valores de gammas son scale (1 / (n_features * X.var())) y auto (1 / n_features) y vamos a probar ambos para ambos tipos de kernel. Por otra parte, se probarán también diferentes valores de C, siendo este parámetro el que representa la fuerza de la regularización, es decir, tolerancia del error. Cuanto mayor sea la C, mayor será el sobreajuste porque no se permite error y cuanto menor sea mayor error permitirá. Los valores para el coeficiente de penalización estarán entre 0 y 1, siendo 1 la menor regularización posible y conforme esta decrementa, aumenta la regularización.

class sklearn.model_selection.**GridSearchCV**(*estimator*, *param_grid*, ***, *scoring=None*, *n_jobs= None*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score=nan*, *return_train _score=False*)

GridSearchCV(pipeline, modelos, scoring='neg_mean_squared_error', cv=5, n_jobs=-1, verbose=4): en esta función incluimos como estimador el Pipeline que hemos creado de preprocesado de los datos. Como lista de parámetros incluimos tanto los modelos lineales como los no lineales que hemos seleccionado para esta práctica con sus respectivos parámetros, como scoring escogemos neg_mean_squared_error, es decir, la estrategia que cross validation usará para evaluar el modelo en el conjunto de datos de test. Este valor es negativo ya que CV trata de maximizar la scoring, y en nuestro caso queremos que el error sea el mínimo. El número de divisiones (k-folds) será por defecto cv=None ya que corresponde a un k-fold=5. El parámetro n_jobs sirve para indicar el número de cores que queremos asignar al procesador para esta tarea. Si ponemos -1 indicamos que vamos a usar todos los núcleos disponibles de nuestro ordenador para la ejecución, mientras que, por defecto, solo se utilizaría 1 y ralentizaría mucho más la ejecución. El parámetro verbose controla la verbosidad, es decir, la cantidad de texto que se nos muestra en la ejecución de cross validation:

- verbose >1 : the computation time for each fold and parameter candidate is displayed;
- verbose >2 : the score is also displayed;
- verbose >3 : the fold and candidate parameter indexes are also displayed together with the starting time of the computation

Como vemos en la documentación, para tener bastante información sobre la ejecución, el valor de este parámetro debe ser > 3, es por ello por lo que se escoge el valor 4.

El parámetro return_train_score que a priori puede parecer interesante tiene por defecto el valor False y es algo que mantenemos. Esto ocurre porque computar los scores en el training set puede ser muy costoso computacionalmente y no es algo estrictamente requerido para seleccionar los mejores parámetros con mejor rendimiento.

class sklearn.linear_model.**SGDRegressor**(*loss='squared_loss', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, random_state=None, learning_rate='invscaling', eta0=0.01, power_t=0.25, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, warm_start=False, average=False*)

SGDRegressor(learning_rate=[‘invscaling’, ‘optimal’], penalty=[l1, l2], "loss": ['squared_loss', 'epsilon_insensitive']): el uso del parámetro de learning_rate ha sido explicados en la página correspondiente a la elección de modelos para el problema.

Igualmente, comentar que pasamos a probar tanto con una penalización Lasso L1 como Ridge L2, una función de error ‘squared_loss’ que hace referencia al ajuste ordinario por mínimos cuadrados y ‘epsilon_insensitive’ que ignora los errores menores que épsilon (0.1 por defecto) y es lineal una vez pasado esto. Esta es la función de pérdida utilizada en SVR. Además, aumentamos el número máximo de iteraciones de 1000 a 2500 para que el modelo trate de ajustar mejor aún.

class sklearn.linear_model.**BayesianRidge**(**, n_iter=300, tol=0.001, alpha_1=1e-06, alpha_2=1e-06, lambda_1=1e-06, lambda_2=1e-06, alpha_init=None, lambda_init=None, compute_score=False, fit_intercept=True, normalize=False, copy_X=True, verbose=False*)

BayesianRidge(n_iter=2000): aumentamos el número de iteraciones para el modelo con respecto al número por defecto. El resto de parámetros (tolerancia, alfas, lambdas, alfa inicial, lambda inicial) serán suficientes para nuestro problema tal y como vienen de forma predeterminada. El parámetro alfa_init (precisión del ruido) será por defecto 1/Var(y) mientras que lambda_init (precisión de los pesos) será de 1.

sklearn.linear_model.**PassiveAggressiveRegressor**(**, C=1.0, fit_intercept=True, max_iter=1000, tol=0.001, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, shuffle=True, verbose=0, loss='epsilon_insensitive', epsilon=0.1, random_state=None, warm_start=False, average=False*)

PassiveAggressiveRegressor(max_iter=2000): para este modelo aumentaremos el número de iteraciones con respecto al máximo por defecto (1000). El parámetro C corresponde con el tamaño máximo de paso (regularización), que por defecto es 1. El criterio de parada, tol, predeterminado es 1e-3. Las iteraciones se detendrán cuando (loss > previous_loss – tol). El valor de épsilon será 0.1, es decir, si la diferencia entre la predicción actual y la etiqueta correcta está por debajo de este umbral, el modelo no se actualiza. Como función de pérdida se usará ‘epsilon_insensitive’ que ignora los errores menores que épsilon (0.1 por defecto) y es lineal una vez pasado esto tal y como vimos en SGDRegressor.

class sklearn.neighbors**.KNeighborsRegressor**(*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs*)

KNeighborsRegressor(n_jobs=-1): el número de vecinos que se usará será el valor por defecto de 5 vecinos. La función de pesos usada en la predicción será ‘uniform’. Esto quiere decir que todos los puntos en cada vecindario estarán ponderados de manera uniforme. Además, el

algoritmo usado para computar este modelo será automático, es decir, intentará decidir cuál es el más apropiado (entre BallTree, KDTree o fuerza bruta) en función de los valores pasados al método fit del mismo. El tamaño de las hojas será de 30 por defecto, pues, el valor óptimo para ello dependerá de la naturaleza del problema. La métrica de distancia usada en este modelo será por defecto minkowski, la cual equivale a una distancia euclídea (l2). Si bien es cierto que se podría intentar obtener los mejores parámetros probando diferentes combinaciones con GridSearchCV, esto ha sido omitido y se han preferido los valores por defecto con tal de agilizar el proceso computacional debido a la cantidad elevada de modelos que probamos sobre este proyecto.

class sklearn.ensemble.**RandomForestRegressor**(*n_estimators=100*, ***, *criterion='mse'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features='auto'*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *min_impurity_split=None*, *bootstrap=True*, *oob_score=False*, *n_jobs=None*, *random_state=None*, *verbose=0*, *warm_start=False*, *ccp_alpha=0.0*, *max_samples=None*)

RandomForestRegressor(n_jobs=-1, n_estimators={20, 100, 200, 300, 400, 500, 600, 1000}): experimentaremos para obtener el número de árboles adecuado. Para ello, se usará como hiperparámetro el número de estimadores (árboles) para este modelo siendo posible tanto 20 como 100, 200, 300, 400, 500, 600 o hasta 1000 árboles. El parámetro n_jobs=-1 indica que usaremos todos los cores procedentes de nuestro ordenador para la ejecución del código en concreto. La función para medir la calidad de una división será “mse” por defecto (el error cuadrático medio) ya que esta será también la métrica de error empleada en GridSearchCV.

Dejamos la profundidad máxima del árbol por defecto ya que en ese caso los nodos se expandirán mientras todas las hojas se hayan completado o todas las hojas contengan menos que samples que min_samples_split (por defecto, 2). También por defecto indicamos que tendremos un número ilimitado de nodos hoja (max_leaf_nodes=None).

El número de características a abordar por nuestro árbol de decisión será n_features. Esto viene dado por el valor por defecto max_features=’auto’.

Un nodo se dividirá si esta división induce una disminución de la impureza mayor o igual a este valor (0 por defecto). La ecuación ponderada de la disminución de impurezas que se sigue es la siguiente:

N_t / N * (impurity - N_t_R / N_t * right_impurity

- N_t_L / N_t * left_impurity)

donde N es el número total de muestras, N_t es el número de muestras en el nodo actual, N_t_L es el número de muestras en el hijo izquierdo y N_t_R es el número de muestras en el hijo derecho.

N, N_t, N_t_R y N_t_L todos se refieren a la suma ponderada, si sample_weight se pasa.

class sklearn.neural_network.**MLPRegressor**(*hidden_layer_sizes=100*, *activation='relu'*, ***, *solver='adam'*, *alpha=0.0001*, *batch_size='auto'*, *learning_rate='constant'*, *learning_rate_init=0.001*, *power_t=0.5*, *max_iter=200*, *shuffle=True*, *random_state=None*, *tol=0.0001*, *verbose=False*, *warm_start=False*, *momentum=0.9*, *nesterovs_momentum=True*, *early_stopping=False*, *validation_fraction=0.1*, *beta_1=0.9*, *beta_2=0.999*, *epsilon=1e-08*, *n_iter_no_change=10*, *max_fun=15000*)

MLPRegressor(max_iter=2000, hidden_layer_sizes: (50, 80), activation: ('tanh', 'relu', 'logistic'), solver: ('sgd', adam', 'lbfgs'), alpha: (0.0001, 0.05), learning_rate: ('constant', 'adaptive')): por defecto contamos con una única capa oculta de 100 neuronas. En nuestra ocasión se nos piden 3 capas y un número de unidades por capa en el rango 50-100, siendo 2 capas ocultas y 1 capa de salida. Consideramos el número de neuronas por capa como un hiperparámetro así que indicamos dos capas ocultas con 50 y 80 neuronas respectivamente.

La función de activación para las capas ocultas será 'tanh', la función hiperbólica f(x) = tanh(x); 'relu', la función de unidad lineal rectificada f(x) = max (0,x); 'logistic', la función sigmoidal logística f(x) = 1 / (1 + exp(-x)). El solver para la optimización de los pesos será tanto adam como sgd y lbfgs. El primero de ellos hace referencia a un SGD optimizado propuesto por Kingma, Diedrick y Jimmy Ba y funciona muy bien para grandes conjuntos de datos. El solver lbfgs es un optimizador de la familia de los métodos quasi-Newton y para datasets pequeños es capaz de converger más rápido y obtener mejores resultados. Por otro lado, el solver sgd es añadido para ver cómo actúa con respecto a adam.

Alfa tendrá tanto el valor 0.0001 (que corresponde con el valor por defecto) y como añadido el valor 0.05. Este parámetro hace referencia a L2 penalty (en términos de regularización).

Por último, el learning rate será tanto constant como adaptive. El primero de ellos significa que, durante las actualizaciones de los pesos, el learning rate se mantiene constante con el valor de learning_rate_init (0.001 por defecto). Sin embargo, el learning rate adaptive tiene el mismo funcionamiento que para el modelo de SGD que ya comentábamos que inicia $\eta = \eta_0$ y va dividiéndolo por 5 si alcanzamos un cierto número de iteraciones sin disminuir el error (en caso de activar early_stopping en cuyo caso el número de iteraciones vendría dado por n_iter_no_change) o en otro caso a las dos épocas sin cambios.

Bibliografía

Gradient Boosting:

Regresión cuantílica: Gradient Boosting Quantile Regression (cienciadedatos.net)

Gradient Boosting con python (cienciadedatos.net)

Ridge, Lasso, ElasticNet:

Rigde Regression | dlegorreta (wordpress.com)

Regularización Ridge, Lasso y Elastic Net con Python y Scikitlearn (cienciadedatos.net)

Perea_Luque_JuanRafael_TFM.pdf (uned.es)

Regresión Ridge | Interactive Chaos

Random Forest, Gradient Boosting:

Machine Learning con Python y Scikitlearn (cienciadedatos.net)

Todas las funciones, así como las clases y sus documentaciones:

https://scikit-learn.org/ → **Página oficial**

Validación cruzada:

https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada#cite_note-Devijver82-1

Pipelines:

https://www.analyticslane.com/2019/02/04/automatizacion-del-procesado-de-datos-en-scikit-learn-con-pipeline/#:~:text=En%20Scikit%2Dlearn%2C%20la%20automatizaci%C3%B3n,uso%20de%20tuber%C3%ADas%20(pipelines).&text=Permitiendo%20crear%20flujos%20de%20trabajos,si%20fuesen%20un%20estimador%20m%C3%A1s

Metrics and scoring:

https://www.geeksforgeeks.org/python-mean-squared-error/

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

Feature importances:

https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html

https://machinelearningmastery.com/calculate-feature-importance-with-python/#:~:text=Feature%20importance%20refers%20to%20a,feature%20when%20making%20a%20prediction

GitHub Codes References:

https://www.kaggle.com/djbacad/airfoil-self-noise-prediction-ann-vs-elasticnet

https://www.kaggle.com/nachiket10/nasa-airfoil-self-noise-prediction-using-xgboost

https://www.kaggle.com/venkatkrishnan/predicting-nasa-airfoil-self-noise

https://github.com/m-shilpa/Airfoil-Self-Noise-Capstone1-ML/blob/master/Airfoil%20Self-Noise%20prediction%20using%20linear%20regression.ipynb

https://github.com/claudinei-daitx/airfoil-self-noise-prediction/blob/master/README.md

Passive Aggressive:

https://www.bonaccorso.eu/2017/10/06/ml-algorithms-addendum-passive-aggressive-algorithms/

https://www.geeksforgeeks.org/passive-aggressive-classifiers/

https://jmlr.csail.mit.edu/papers/volume7/crammer06a/crammer06a.pdf

KNeighbor Regressor:

https://www.aprendemachinelearning.com/clasificar-con-k-nearest-neighbor-ejemplo-en-python/

https://scikit-learn.org/stable/auto_examples/neighbors/plot_regression.html

https://medium.com/analytics-vidhya/k-neighbors-regression-analysis-in-python-61532d56d8e4

Decision Trees:

https://www.geeksforgeeks.org/python-decision-tree-regression-using-sklearn/

https://mljar.com/blog/visualize-decision-tree/

Random Forest:

https://www.cienciadedatos.net/documentos/py08_random_forest_python.html

https://www.geeksforgeeks.org/random-forest-regression-in-python/

MLPRegressor:

https://towardsdatascience.com/deep-neural-multilayer-perceptron-mlp-with-scikit-learn-2698e77155e

https://www.python-machinelearning.com/multi-layer-perceptron-regressor/

https://www.dezyre.com/recipes/use-mlp-classifier-and-regressor-in-python

https://stackoverflow.com/questions/62131266/mlpregressor-working-but-results-dont-make-any-sense

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

BayesianRidge Regression:

https://scikit-learn.org/stable/auto_examples/linear_model/plot_bayesian_ridge.html#sphx-glr-auto-examples-linear-model-plot-bayesian-ridge-py

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

https://www.geeksforgeeks.org/implementation-of-bayesian-regression/

Linear Regression:

https://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html

https://www.activestate.com/resources/quick-reads/how-to-run-linear-regressions-in-python-scikit-learn/

Models and some references:

https://www.kaggle.com/shreayan98c/boston-house-price-prediction