Aprendizaje Automático

Proyecto Final - Communities and Crime

Alejandro Cruz Lemos
Guillermo García Arredondo

Índice

1. Descripción del problema a resolver

Se busca predecir el número total de crímenes violentos por cada 100 000 habitantes en comunidades de Estados Unidos, haciendo uso de 1994 datos con 127 atributos relacionados con la comunidad (porcentaje de población urbana, mediana del salario familiar...) y con las fuerzas de seguridad actuando en esa comunidad (número de policías per cápita, porcentaje de oficiales asignados a lucha contra drogas...).

Contaremos entonces con un espacio de entrada R^{127} para estimar una única variable continua (espacio de salida R), por lo que afrontaremos este problema como un problema de regresión.

a. Información sobre el dataset

Nuestro dataset combina datos socioeconómicos del censo de los Estados Unidos del año 1990, de encuestas LEMAS (Law Enforcement Management and Administrative Statistics) realizadas en Estados Unidos también en el año 1990, y del programa UCR (Uniform Crime Reporting) publicado por el FBI en el año 1995.

El número de crímenes violentos per cápita fue calculado usando la población y la suma de crímenes considerados como *crímenes violentos* en Estados Unidos: asesinato, violación, atraco y agresión. Se dio cierta controversia respecto al conteo de violaciones, lo que resultó en valores perdidos y cálculos incorrectos del número de crímenes violentos per cápita para algunas comunidades. No tendremos que tratarlas ya que fueron omitidas en el dataset, pero podría suponer estimaciones erróneas en la población al haber perdido varias comunidades del centro-oeste del país (esto es, si fuéramos a probar nuestro modelo en la población).

Todos los datos numéricos fueron previamente normalizados al rango $[0, 1]$ manteniendo su distribución y asimetría estadística (grado de simetría de la distribución de probabilidad de una variable aleatoria). Sin embargo, no se preservaron las relaciones

entre variables, pues no se consideraban relevantes (ya que, por ejemplo, no tendría sentido comparar el número de personas caucásicas con el número de personas afroamericanas por comunidad). Además, los valores de atributos tres veces la desviación típica por encima de la media fueron transformados a 1, mientras que los valores tres veces la desviación típica por debajo de la media fueron transformados a 0.

Por tanto, no será necesario normalizar ni tratar *outliers* (datos atípicos, posiblemente resultantes de medidas erróneas, numéricamente distantes del resto de datos). Se explicará en el apartado *2.e.* el porqué.

De los 127 atributos, 5 son informativos pero no predictivos. Es decir, no contribuyen en la predicción. Esos atributos son:
- "state": estado en USA (numérico).
- "county": código numérico del condado/municipio (además tiene muchos datos perdidos).
- "community": código numérico de la comunidad (también tiene muchos datos perdidos).
- "communityname": nombre de la comunidad.
- "fold": número de folds para validación cruzada no aleatoria. Dato de depuración.

No se tendrán en cuenta para resolver el problema.

Por lo que se nos informa en la descripción del dataset[1], los 122 atributos restantes fueron escogidos garantizando que tuviesen algún tipo de relación significativa con el crimen, además de con la variable a predecir (número de crímenes violentos por cada cien mil habitantes). Por tanto, los 122 atributos deberían ser relevantes para la predicción de nuestros modelos.

Sin embargo, existe una limitación respecto a los atributos asociados con las encuestas LEMAS, y es que solo se realizaron a departamentos de policía con al menos cien oficiales. Como la mayoría de las comunidades son pequeñas[2], esto dio lugar a un número considerable de datos perdidos.

[1] «However, clearly unrelated attributes were not included; attributes were picked if there was any plausible connection to crime (N=122), plus the attribute to be predicted (Per Capita Violent Crimes)». UCI - Communities and Crime Data Set.

[2] «(hence for example the population attribute has a mean value of 0.06 because most communities are small)». UCI - Communities and Crime Data Set.

2. Preprocesado de datos

Antes de plantear los modelos será necesario preparar los datos para hacer un uso óptimo y útil de ellos. Discutiremos por tanto la separación en *training* y *test*, necesidad de codificar las variables, el interés de los atributos y selección de estos, y la necesidad de normalizar, entre otras cuestiones.

a. Particionado del dataset

Lo primero y más importante será dividir los datos en *training* y *test*. De esta manera, podremos excluir el conjunto de *test* resultante hasta que hayamos seleccionado hipótesis y así evitar espiar esos datos, pues representan la población desconocida sobre la que se evaluaría el modelo en un caso real y no debemos saber nada de ellos.

Se dividirá la muestra en un 80% *training* y 20% *test*. Como la muestra es de un tamaño considerable (1994), podremos permitirnos dejar solo 20% de prueba (398) sin temor a tener datos insuficientes para que la evaluación sea fiable, y así realizar un entrenamiento más informado con el 80% que hemos dejado para el *training* (1596).

b. Codificación de los datos de entrada

La necesidad de codificar los datos de entrada surge con la presencia de variables categóricas, variables que solo pueden tomar un número finito y limitado de valores en base a alguna característica cualitativa (como podría ser el sexo, el color de pelo, etc).

En nuestro caso, tras haber descartado los atributos no predictivos solo nos queda una variable no categórica: “LemasGangUnitDeploy”. Esta variable indica si la comunidad tiene unidades de seguridad contra bandas callejeras, y puede tomar los valores 0 si es que no, 1 si es que sí, y 0.5 si es a media jornada. Es una variable numérica pero ordinal, por tanto categórica. Los valores que puede tomar son números arbitrarios que no representan ningún valor numérico real en el mundo, por lo que sería preciso tratarla con técnicas

como one-hot encoding (transformar características categóricas en vectores numéricos de $K = 3$ elementos) para que sean útiles para los modelos.

Sin embargo, no será necesario ya que es un atributo asociado a la encuesta LEMAS, y la tendremos que eliminar por tener más de un 20% de datos perdidos. En el siguiente apartado se explicará con más detalle el tratamiento de los datos perdidos.

c. Tratamiento de los datos perdidos

Debido a la limitación de la encuesta LEMAS mencionada en el apartado *1.a.*, sabemos con seguridad que tenemos datos perdidos en la muestra. Sin embargo, no tiene por qué ser solo los atributos asociados al LEMAS los que les faltan datos.

El propio dataset nos proporciona estadísticas sobre el número de datos perdidos (además de valores mínimos, máximos, media, desviación típica, etc), pero no las consultaremos porque nos aportaría información sobre la muestra completa y supondría espiar el conjunto de *test*. Para ejemplificar cómo claramente sería un caso de *data snooping*, supongamos que tenemos un 10% de datos perdidos en *training* y otro 10% en *test* para una variable. Si solo tuviésemos en cuenta el conjunto de entrenamiento, nuestra decisión consistiría en sustituir esos datos perdidos por la media de la variable más un valor aleatorio dentro de un intervalo ($[-1.5\sigma, 1.5\sigma]$ donde σ es la desviación típica de dicha variable), mientras que si tuviéramos en cuenta la muestra completa nuestra decisión sería descartar esa variable al superar el 20% de datos perdidos.

Por tanto, lo que haremos será comprobar el número de valores perdidos *exclusivamente* en el conjunto de entrenamiento.

Si ese número de datos perdidos es inferior o igual al 10% (respecto al tamaño de *training*), sustituiremos dichos valores por la media de la variable más un valor aleatorio en el intervalo $[-1.5\sigma, 1.5\sigma]$, donde σ es la desviación típica de la variable en *training*. Si luego nos encontramos con valores perdidos para la misma variable en el *test*, sustituiremos también esos datos por valores aleatorios en el mismo intervalo, usando la desviación típica de *training*.

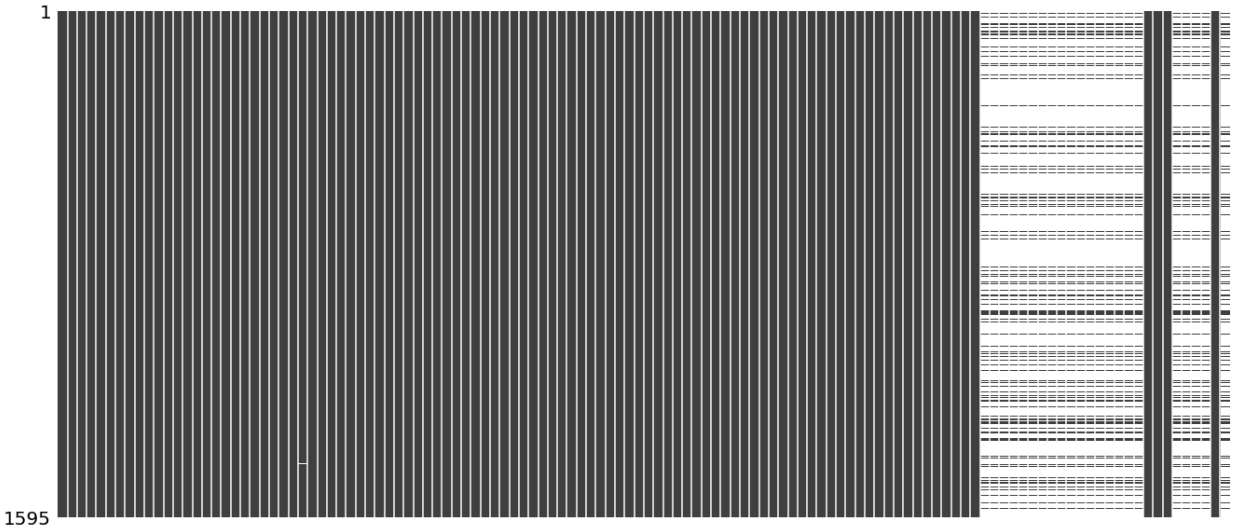
Si ese número de datos perdidos es superior al 10% y su eliminación no afecta al tamaño del conjunto de datos, o si es directamente superior al 20%, descartaremos la variable para ambos el *training* y *test*.

En ambos casos tomaremos decisiones en base al conjunto de entrenamiento y sin consultar el conjunto de prueba, por lo que evitaremos el espionaje de datos.

Este es el número de datos perdidos en *training* (solo se incluyen las variables que tienen al menos un dato perdido).

- OtherPerCap. 1 dato.
- LemasSwornFT. 1337 datos.
- LemasSwFTPerPop. 1337 datos.
- LemasSwFTFieldOps. 1337 datos.
- LemasSwFTFieldPerPop. 1337 datos.
- LemasTotalReq. 1337 datos.
- LemasTotReqPerPop. 1337 datos.
- PolicReqPerOffic. 1337 datos.
- PolicPerPop. 1337 datos.
- RacialMatchCommPol. 1337 datos.
- PctPolicWhite. 1337 datos.
- PctPolicBlack. 1337 datos.
- PctPolicHisp. 1337 datos.
- PctPolicAsian. 1337 datos.
- PctPolicMinor. 1337 datos.
- OfficAssgnDrugUnits. 1337 datos.
- NumKindsDrugsSeiz. 1337 datos.
- PolicAveOTWorked. 1337 datos.
- PolicCars. 1337 datos.
- PolicOperBudg. 1337 datos.
- LemasPctPolicOnPatr. 1337 datos.
- LemasGangUnitDeploy. 1337 datos.
- PolicBudgPerPop. 1337 datos.

Gráficamente (donde las líneas blancas horizontales en cada columna son datos perdidos):



Excepto por la variable “OtherPerCap”, el resto de atributos tienen más del 20% de datos perdidos. Bastante más de hecho, un 83%. Por tanto, sustituiremos el valor perdido de OtherPerCap por la media de la variable más un valor aleatorio en $[-1.5\sigma, 1.5\sigma]$ (σ desviación típica de la variable en *training*) y eliminaremos el resto de atributos.

d. Normalización

La normalización en un dataset se vuelve necesaria cuando las variables tienen rangos numéricos considerablemente distintos. Por ejemplo, si tuviéramos una variable edad, cuyos valores fueran desde 0 hasta 100, y una variable salario, cuyos valores fueran desde 0 hasta 20 000, tendríamos una diferencia de escala que podría afectar al rendimiento de nuestros modelos.

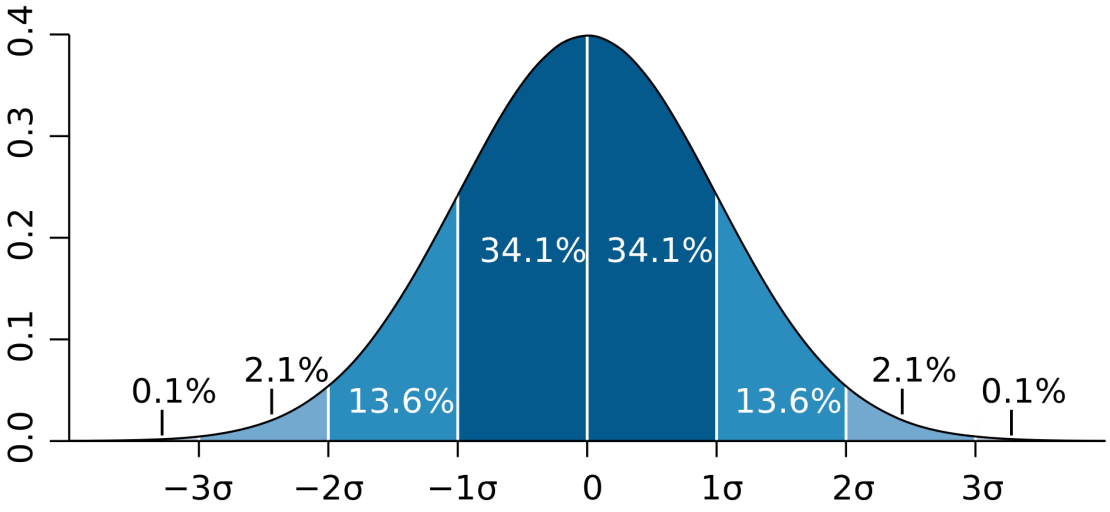
Esto se debe a que variables con mayor escala suponen coeficientes de también gran escala y por tanto una mayor influencia en el modelo que el resto de variables, además de afectar al proceso de regularización ya que ambos Lasso y Ridge actúan sobre la magnitud de los coeficientes del modelo.

No obstante, no será necesario aplicar normalización en nuestro caso ya que los valores del dataset (como se mencionó en el apartado *1.a.*) ya están normalizados al rango $[0, 1]$.

e. Tratamiento de datos atípicos (outliers)

Los *outliers* son valores atípicos numéricamente distantes del resto de la muestra, usualmente resultantes de errores en la recogida de datos (denominado *ruido estocástico*). Nos interesan eliminarlos para asegurarnos que no tengan influencia en el modelo, porque son datos por definición incorrectos (o al menos no representativos de la población) y podrían causar sobreajuste.

Sin embargo, ¿cómo identificamos estos outliers? Una opción sería, suponiendo la distribución de la muestra como Gaussiana o similar, considerar como datos atípicos aquellos que estén n veces la desviación típica por encima (o por debajo) de la media.



Siendo esta gráfica la distribución Gaussiana de un conjunto de datos cualesquiera, podemos observar como conforme nos alejamos n veces de la desviación típica los valores son cada vez más atípicos (valga la redundancia). Nuestro objetivo sería entonces escoger un n (que en la literatura suele ser 3, pues como podemos observar en la gráfica corresponde a menos de 1% de la muestra) y tratar aquellos que estén n veces la desviación típica por encima (o por debajo) de la media.

No obstante, esto no será posible ya que, como se explicó en el apartado *1.a.*, los valores atípicos ya venían tratados en el dataset. Concretamente, se transformaron a 1 todos aquellos valores 3 veces la desviación típica por encima de la media y se transformaron a 0 todos aquellos valores 3 veces la desviación típica por *debajo* de la media. Por tanto, no será necesario tratar *outliers.*

f. Valoración de interés y selección de las variables medidas

Ya hemos descartado las variables no predictivas (pues, por definición, no aportaban información relevante para las predicciones) y eliminado aquellas cuyos datos perdidos superaban con creces el umbral de lo aceptable. Para poder valorar y seleccionar las variables restantes necesitaríamos un conocimiento en criminología que nosotros, como estudiantes de ingeniería informática, no poseemos. No tenemos al alcance el contacto de un experto en la materia, y tampoco es un conocimiento que podamos adquirir en dos tardes de estudio.

No obstante, tenemos la información aportada por el propio dataset. Como se explicó en el apartado *1.a.*, los 122 atributos predictivos fueron escogidos garantizando que tuviesen una relación significativa con el crimen y con la variable a predecir (número de crímenes violentos por cada cien mil habitantes). Sabiendo que, en principio, todos los atributos aportan información relevante a nuestro modelo, la criba realizada en los apartados anteriores debería ser suficiente selección para dejar únicamente variables de interés en nuestros modelos.

3. Regularización

a. Interés de la regularización

La necesidad de usar regularización surge con la posibilidad de sobreajuste. Cuando nos encontramos con una muestra con ruido estocástico (casi siempre), existe el riesgo de que acabemos ajustando nuestros modelos a esos datos incorrectos con tal de reducir el error en la muestra, dando lugar a un modelo excesivamente complejo que bien funciona excelentemente para esa muestra pero que luego es desastroso a la hora de predecir en la población ($E_{out} > E_{in}$).

Para evitarlo, las técnicas de la regularización típicas actúan sobre la magnitud de los coeficientes con tal de reducir la sensibilidad a fluctuaciones entre los datos de entrenamiento (denominado *varianza*). Es decir, reduce la influencia de esas diferencias entre los valores de los datos simplificando el modelo a uno más general. Sin embargo, esta simplificación puede causar que nuestro modelo no encuentre las relaciones relevantes entre las características y la variable de salida, dando lugar a una falta de ajuste (alto *sesgo*).

Nuestra intención será por tanto alcanzar un equilibrio entre el *sesgo* y la *varianza*, generalizando el modelo para evitar sobreajuste (esto es, reduciendo la varianza), pero manteniendo un sesgo lo suficientemente bajo como para que sea capaz de representar la complejidad real de la función a estimar.

b. Selección de regularización

Las opciones de regularización que vamos a valorar para este proyecto son las dos dadas en clase: *Lasso* (también conocido como L1) y *Ridge* (L2).

Ambos actúan sobre los coeficientes del modelo con tal de reducir la varianza, pero los efectos resultantes de su aplicación son distintos.

Lasso fuerza a que los coeficientes de las variables tiendan a cero (pudiendo llegar a *ser* 0), de esta manera realizando una selección de predictores al excluir los menos relevantes.

Ridge, en cambio, reduce de forma proporcional el valor de los coeficientes del modelo pero sin que estos lleguen a cero, dando más peso a unas variables sobre otras pero sin llegar a excluir ninguna.

Volviendo a lo explicado en el apartado *1.a.*, sabemos que los 122 atributos predictivos son relevantes en mayor o menor medida a la hora de estimar la variable de salida, pues fueron escogidos para el dataset con ese objetivo. Los únicos atributos que podrían ser menos significativos eran los no predictivos o los que presentaban demasiados datos perdidos, y estos ya han sido eliminados.

Por este motivo, Lasso no sería una buena opción ya que, en principio, todas las variables restantes son relevantes y no nos interesa excluir ninguna. Por tanto, escogeremos Ridge para que reduzca proporcionalmente aquellas variables que estén más o menos correlacionadas pero sin descartar ninguna de ellas.

c. Funcionamiento de Ridge (L2)

Lo que hace Ridge es penalizar la suma de los coeficientes elevados al cuadrado ($||\beta||_2^2 = \sum_{j=1}^{p} \beta_j^2$) , reduciendo de forma proporcional aquellos subconjuntos de variables altamente correlacionadas pero sin que estos lleguen a cero. El grado de penalización viene dado por el hiperparámetro λ (*alpha* en scikit-learn). Conforme aumenta λ, mayor es la penalización y menor el valor de los predictores.

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

Empleando el valor adecuado de λ, Ridge es capaz de reducir la varianza en el modelo a costa de tan solo un leve aumento en sesgo.

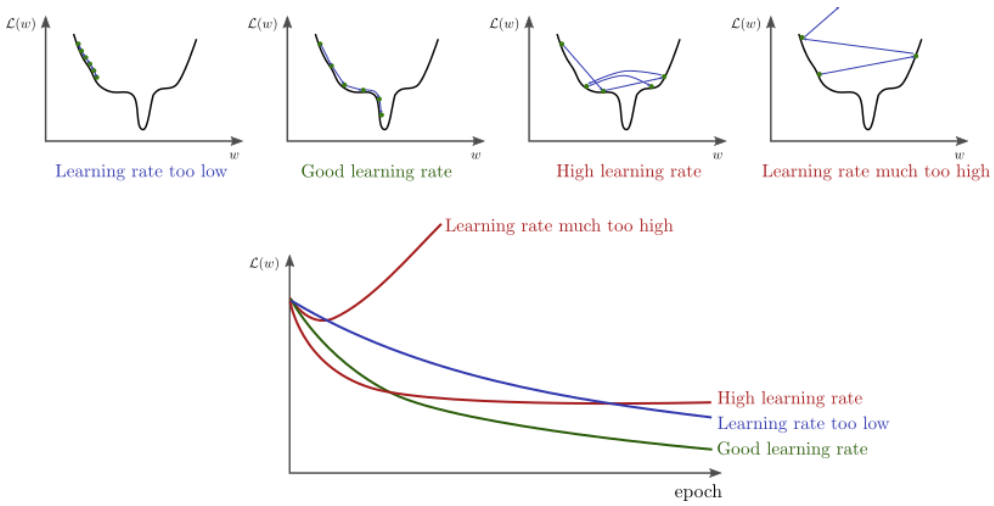
4. Algoritmos de aprendizaje

En este apartado vamos a tratar los distintos algoritmos de aprendizaje usados en los modelos lineal / no lineales. Para ello, comenzaremos con una breve explicación acerca del algoritmo, seguido de las ventajas e inconvenientes del mismo.

a. Gradiente descendente

Es el algoritmo de aprendizaje usado en modelos lineales con el que vamos a comparar el resto de algoritmos para modelos no lineales. A grandes rasgos *gradiente descendente* es un modelo matemático que usa el gradiente del error cuadrático medio para ir ajustando nuestro vector de pesos w y así crear una línea de regresión que minimice el error a obtener en nuestro problema. En cada época de nuestro algoritmo, el vector de pesos se actualiza de la siguiente forma $w_j := w_j - \eta \frac{\partial Ein(w)}{\partial w_j}$, siendo E_{in} la función de pérdida (función a minimizar, normalmente *error cuadrático medio*), $\partial E_{in}/\partial w$ la derivada de la función de pérdida respecto de w y η la tasa de aprendizaje.

La tasa de aprendizaje determinará el tamaño del paso a dar en el camino hacia el mínimo de la función de pérdida. Hay que saber que si es demasiado reducida el algoritmo podría no llegar a converger, mientras que si es demasiado grande podría saltarse el mínimo y acabar en puntos donde la función de pérdida incrementa. Hay que encontrar un punto intermedio o bien optar por una tasa de aprendizaje dinámica que se adecue a la pendiente conforme avanza el número de épocas.



Finalmente el algoritmo termina de iterar si bien se ha alcanzado el número máximo de épocas *k* o si hemos alcanzado un valor de la función de pérdida menor que el valor mínimo ε.

Entre las ventajas que caracterizan al gradiente *descendente* están que es fácil de implementar (por su simpleza) y eficiente a la hora de manejar vectores de características grandes (pues no realiza transformaciones costosas como inversiones a las matrices de datos), además de menos propenso a sufrir sobreajuste al ser un modelo de regresión lineal.

No obstante, no es un buen algoritmo para problemas complejos pues no es capaz de capturar relaciones no lineales sin antes hacer alguna transformación al vector de características.

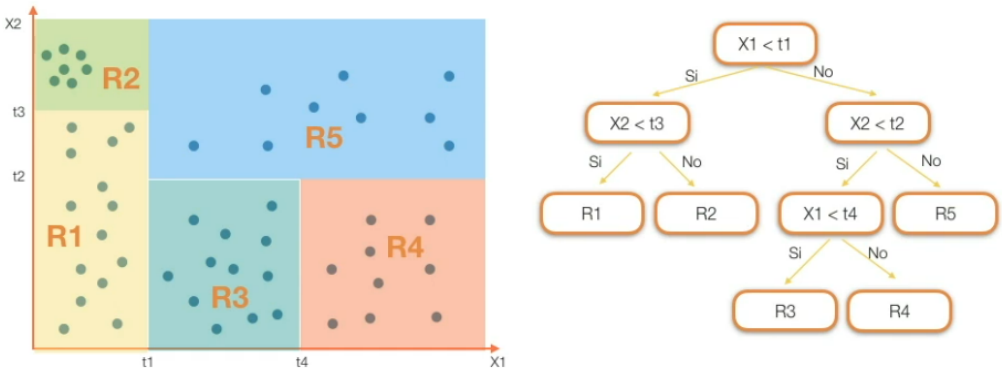
Lo hemos elegido para el modelo lineal porque creemos que puede ser útil para dar un primer vistazo al tipo de relación entre las características y la variable de salida.

b. Random Forest

Random Forest es un método de aprendizaje usado en ambos problemas de clasificación y regresión en las que opera mediante la construcción de una multitud de árboles de decisión.

En regresión, los árboles de decisión se encargan de dividir el espacio de características mediante hiperplanos paralelos al eje, dando como resultado un conjunto de regiones distintas que albergan los valores que puede tomar la variable a predecir dependiendo del camino que siga el árbol de decisión (ver figura más abajo). Para evaluar un dato, se profundiza en el árbol de decisión dependiendo de los valores de los predictores del dato, hasta llegar a un nodo hoja que corresponde con una de las regiones comentadas. De los valores de la variable de salida que se encuentran en esta región se realiza la media para finalmente obtener una predicción.

Por último (para no extender demasiado esta explicación), comentar que la construcción de los árboles (división de cada nodo del árbol a partir del nodo raíz) se realiza siguiendo el criterio de *información* y *entropía* .



La complejidad de dichos árboles de decisión viene dada por la cantidad de nodos que emplea. De forma matemática, hay un compromiso entre el número de nodos del árbol (n), que influye linealmente en el error de generalización, y el número de datos en la muestra (N). Tiene sentido pensar que un árbol con más nodos es más complejo que uno con menos nodos.

$$Eout(h) \leq Ein(h) + \sqrt{\frac{(n+1)\, log2(d+3) + log(\frac{2}{\delta})}{2N}}$$

Podemos asumir por tanto que Random Forest se trata de un algoritmo que deriva en modelos complejos, en el sentido de que utiliza un conjunto de árboles de decisión para tomar las predicciones. Esto nos garantiza que nuestro *sesgo* (lo lejos que se encuentra nuestro valor estimado respecto al valor real de la población) será relativamente pequeño, debido a que como sabemos el sesgo presenta una relación inversa a la complejidad de nuestro modelo.

Si nos queremos asegurar de que el modelo no sobreajuste además de reducir el sesgo debemos procurar que la varianza (diferencia entre distintas muestras) también se reduzca. El problema es que a medida que aumentamos la complejidad de los modelos, aumenta la probabilidad de sobreajuste (la varianza aumenta). Random Forest es capaz de lidiar con este problema debido primero a que es un método de "ensemble" o de conjunto. Es decir, utiliza múltiples algoritmos de aprendizaje (árbol de decisión en este caso) para

obtener un mayor rendimiento predictivo del que se podría obtener con cualquiera de los algoritmos que constituye por sí solo y segundo porque aporta una mejora al método usual de *bagging* construyendo árboles no correlados (más independientes entre sí).

Para entender correctamente el punto anterior cabe explicar lo que son *bootstrapping* y *bagging*. El primero se basa en, teniendo presente nuestro conjunto de datos, crear nuevos conjuntos del mismo tamaño pero escogiendo aleatoriamente los datos que van a pertenecer a ese conjunto (de manera que incluso se puedan repetir), para posteriormente crear un árbol de decisión por cada conjunto generado. El problema aquí es que en un árbol de decisión un mínimo cambio genera una variabilidad enorme, por lo que la varianza también sería grande. Para resolverlo usamos *bagging*, que se basa en la idea de que con un número suficientemente grande de árboles se puede reducir la varianza para que la suma de los errores asociados a sesgo-varianza sea muy pequeño. A grosso modo, consiste en realizar la media de las estimaciones obtenidas para cada uno de nuestros árboles de decisión creados después de *bootstrapping*.

Existe un problema con *bagging* y es que aun teniendo un número grande de árboles no se consigue disminuir la varianza tanto como se desea. Esto es debido a que puede ocurrir que varios de los árboles tengan nodos a una altura concreta que usan la misma variable, dando como resultado árboles dependientes entre sí (si queremos reducir la varianza los datos promediados tienen que ser lo más independientes posible). Lo que hace *random forest* para solucionarlo es que, a la hora de construir un árbol, por cada muestra generada en bootstrapping, selecciona de forma aleatoria un porcentaje del total de variables que tenemos. Tras numerosos experimentos y consenso, se consideró que el número de variables a seleccionar para cada árbol que obtiene mejores resultados es el equivalente a la raíz cuadrada del total de predictores que tenemos.

De esta forma decidimos usar random forest porque es un algoritmo ampliamente usado que debido al pequeño sesgo que trae de fábrica gracias a los árboles de decisión y a la varianza sensible ante el promedio comentado, consigue una gran disminución del error y una poca probabilidad de sobreajuste.

Entre las desventajas hay que tener en cuenta que con conjuntos de datos particularmente ruidosos (no nuestro caso) puede llegar a sobreajustarse, y que es muy

difícil interpretar la gran cantidad de árboles creados, en caso de que quisiéramos comprender y explicar a algún cliente su comportamiento.

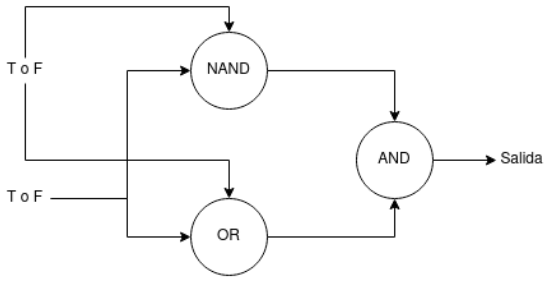
c. Perceptrón Multicapa

Es una red neuronal artificial que pretende solventar las limitaciones del Perceptrón simple incorporando capas ocultas al modelo para así poder representar funciones no lineales. La idea principal es, mediante un Perceptrón simple, representar las funciones lógicas AND y OR, ya que estas funciones se pueden implementar definiendo el signo de un hiperplano, de esta forma:

$$OR(x_1, x_2) = sign(x_1 + x_2 + 1.5)$$
$$AND(x_1, x_2) = sign(x_1 + x_2 - 1.5)$$

Si nos fijamos en el caso de OR, cuando ambas variables son negativas, el resultado será -1 -1 + 1.5 = -0.5; si uno es positivo y otro negativo, el signo siempre será positivo gracias al 1.5; Si ambos son positivos el signo es claramente positivo. En el caso de AND se podría aplicar el mismo procedimiento anterior para ver que se cumple.

Teniendo esto en cuenta, el típico ejemplo para saber cómo funciona Perceptrón Multicapa es intentar implementar con varios perceptrones la función lógica XOR, la cual no es linealmente separable. Si tenemos una tabla como la que se muestra a la derecha, nuestros datos de entrada son de la forma T o F (verdadero o falso) y los círculos del diagrama de la izquierda son perceptrones simples, vemos cómo podemos implementar con varias capas de perceptrones una función que no es linealmente separable. Esto es básicamente lo que hace Perceptrón Multicapa, donde la primera capa se denomina de entrada (los datos de entrada), las capas intermedias se denominan capas ocultas y la capa final corresponde con las salidas de toda la red.



XOR	True	False
True	False	True
False	True	False

En conclusión, el perceptrón multicapa consiste en un conjunto de nodos organizados por capas, de modo que una neurona (un conjunto de entradas, un conjunto de pesos y una función de activación) puede recibir entradas solo de aquellas situadas en la capa inmediatamente inferior. Si nos centramos en problemas de regresión, cada nodo del perceptrón multicapa calcula una combinación lineal de las entradas que llegan a él, le añade un sesgo y, finalmente, le aplica una función de activación, llamada también de transferencia, que por regla general traslada cualquier entrada real a un rango acotado, dando lugar así a la salida del nodo, que puede ser una de las entradas de un nuevo nodo.

Entre las ventajas a destacar, el perceptrón multicapa tiene la capacidad de llevar a cabo el proceso de entrenamiento y aprendizaje a partir de conjuntos reducidos de datos obteniendo buenos resultados, aunque sus resultados mejoran con un mayor conjunto de datos. Además, debido a que las RNAs (Red Neuronal Artificial) interpretan los datos únicamente como valores numéricos, son capaces de resolver numerosos problemas en diferentes campos de conocimiento.

Entre las desventajas principales de este algoritmos está el hecho de que conforme más perceptrones añadimos para su funcionamiento, más probabilidad de sobreajuste hay. En general, es necesario ajustar adecuadamente hiperparámetros como el número de capas ocultas, funciones de activación, número de iteraciones, solver (optimizador) y número de neuronas por capa. También son muy sensibles a la escala de datos aunque como nuestros datos ya están normalizados no deberíamos de tener problema.

d. Función de pérdida

Las opciones que hemos valorado para la función de pérdida han sido MSE (Mean Squared Error) y MAE (Mean Absolute Error).

MSE se calcula como el promedio de las diferencias cuadráticas entre los valores predichos y reales. Al ser cuadrático, penaliza sobre todo errores grandes.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \widehat{Y}_i)^2$$

Por otro lado, MAE se calcula como el promedio del valor absoluto de las diferencias entre los valores predichos y reales. Penaliza los errores proporcionalmente a esas diferencias.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|Y_i - \widehat{Y}_i|$$

MAE es interesante desde un punto de vista interpretativo ya que devuelve la diferencia exacta promedio entre las predicciones y valores reales. Es más intuitivo que MSE pues en este último se pierden las unidades y escala del valor a predecir al hacer el cuadrado. Sin embargo, no es la métrica de evaluación lo que estamos valorando sino la función de pérdida.

Optaremos por tanto por MSE ya que nos interesa penalizar errores cuantiosos con tal de que el modelo se ajuste adecuadamente al problema.

5. Selección de modelos

Una vez realizado el preprocesado de datos, escogido tipo de regularización y haber explicado los algoritmos que vamos a utilizar, procederemos a seleccionar los modelos que formarán nuestras hipótesis finales.

a. Selección de parámetros

Para todos parámetros e hiperparámetros de los algoritmos que no podamos escoger siguiendo un razonamiento concreto (como podría ser la tasa de aprendizaje, o el λ de regularización) los seleccionaremos utilizando una técnica conocida como *GridSearch*.

Dado unos subconjuntos de valores para un subconjunto de parámetros concreto, *GridSearch* prueba todas las combinaciones posibles y devuelve aquella con el mejor error de validación medio (esto es, utilizando validación cruzada, que se explicará en el apartado *5.f.*). Haremos uso de esta técnica para cada modelo y mostraremos los resultados en sus respectivos apartados.

b. Gradiente descendente estocástico con regularización Ridge y parámetros

Empezaremos con un modelo lineal, ya que aun siendo de los más simples a nuestro alcance será mucho más interesante en caso de que nos dé buenos resultados, pues significaría que hay una relación lineal entre las características y la variable de salida.

Gradiente descendente estocástico será el algoritmo que escogeremos para este modelo, porque es el más potente y eficiente entre los disponibles para regresión lineal al ser capaz de lidiar con un gran número de características (cosa que, por ejemplo, la pseudoinversa no puede por el alto coste computacional de invertir matrices) y converger hacia óptimos más que aceptables.

Lo implementaremos con la función de scikit-learn *SGDRegressor*. Es estocástico porque baraja el conjunto de entrenamiento en cada época. No obstante, cabe mencionar que este *SGDRegressor* no maneja minibatches. Los valores asignados a los parámetros de la función son:

- loss , este parámetro determina la función de pérdida a usar en el algoritmo y como ya se comentó en el apartado 4.d vamos a usar MSE (Mean Squared Error) que en este parámetro equivale a 'squared_loss'.
- penalty, nos determina la regularización a aplicar y como ya se comentó en el apartado 3.b usaremos el valor por defecto 'l2' (Ridge).
- alpha, que hace referencia al hiperparámetro λ de regularización, el cual determina la dureza con la que se va a penalizar los coeficientes. Resultados obtenidos mediante *GridSearch.*
- tol , dejamos el valor por defecto (1e-3) ya que pensamos que es un valor de función de pérdida lo suficientemente bajo, de hecho en la mayoría de casos probablemente se agote el número máximo de iteraciones.
- shuffle, vamos a usar el valor por defecto 'True' para que los datos de entrenamiento sean barajados después de cada época fomentando mayor variabilidad.
- learning_rate, el tipo de tasa de aprendizaje. Como se explicó en el apartado *4.a.*, nos interesa escoger una tasa de aprendizaje dinámica que pueda adaptarse a la pendiente conforme se aproxime al mínimo. Por tanto, escogeremos 'adaptive' el cual disminuye η cada vez que no se logra disminuir el error tras un número dado de épocas.
- eta0, equivale al valor inicial que toma la tasa de aprendizaje. Usando *GridSearch* hemos obtenido que '0.01' es el mejor valor.
- early_stopping, sirve para finalizar el entrenamiento cuando la función de pérdida no disminuye después de un número determinado de épocas (dado por el parámetro n_iter_no_change). Nosotros le hemos asignado el valor False porque no queremos que se restrinja ninguna posibilidad de mejora.

A continuación se muestra la tabla de los valores obtenidos en *GridSearch*. Donde se compara el MAE para distintos valores de los parámetros eta0 y alpha.

alpha \ eta0	**0.01**	**0.1**
0.0001	0.0967	0.0982
0.001	0.0966	0.0982

c. Random Forest con parámetros

Random Forest destaca por su flexibilidad y simpleza a la hora de ajustar y entrenar, y porque a pesar de esa simpleza es capaz de obtener resultados tan buenos como otros más complejos como los de redes neuronales. Es por este motivo que es uno de los algoritmos más populares y ampliamente utilizados, y la razón por la que lo hemos escogido para este primer modelo no lineal.

Sin regularización, pues no lo consideramos necesario porque el algoritmo ya reduce la varianza con la técnica de *bootstrapping* y la selección aleatoria de atributos para cada árbol.

Usamos la función *RandomForestRegressor* que proporciona scikit-learn para realizar la evaluación de este algoritmo. Los valores asignados a los parámetros de la función son:

- n_estimators, equivale al número de árboles que vamos a emplear. Evaluando con *GridSearch* obtenemos un resultado de 50 árboles.
- criterion, determina la función de pérdida a usar en el algoritmo y como ya se comentó en el apartado 4.d vamos a usar MSE (Mean Squared Error) que en este parámetro equivale a 'mse'.
- max_depth, es la profundidad máxima a la que puede llegar el árbol de decisión. Lo hemos dejado con el valor por defecto 'None' porque queremos que los nodos se expandan hasta que todas las hojas sean puras o hasta que todas las hojas contengan menos de min_samples_split (otro parámetro) muestras.
- min_samples_split, es la cantidad mínima de muestras que debe tener un nodo para que pase a dividirse. Tiene el valor por defecto 2.
- min_samples_leaf, es la cantidad mínima de muestras que debe tener un nodo para ser una hoja. Después de hacer *GridSearch* hemos concluido con que 5 es el valor que mejores resultados obtiene.
- max_features, es la cantidad de atributos que vamos a usar para crear cada uno de los árboles creados con *bootstrapping.* Usamos el valor el valor que se indicaba en teoría correspondiente a la raíz cuadrada del total de atributos que tenemos, en este parámetro se escribe como 'sqrt'.

- max_leaf_nodes, sirve para indicar el número máximo de hojas que queremos que tenga nuestro árbol, es este caso lo dejamos por defecto ('None') para que pueda tener todas las hojas necesarias.
- bootstrap, lo dejamos por defecto a True que nos realiza el método de bootstrapping tal y como se comentó en el apartado *4.b*.
- obb_score, también le asignamos el valor True para que trate con las muestras *out-of-bag* (muestras que debido a la selección aleatoria de *bootstrapping* no se hayan podido seleccionar en ningún árbol si las hubiera).
- max_samples, indica cómo de grande van a ser el conjunto de muestras seleccionadas en *bootstrapping* . Lo dejamos con el valor por defecto que escoge un número de muestras igual al total de datos de entrenamiento.

n_estimators \ min_samples_leaf	**5**	**10**
10	0.096	0.0962
50	0.0932	0.0946

d. MLP con regularización Ridge y parámetros

Consideramos MLP como algoritmo para el segundo modelo no lineal por el buen rendimiento con conjuntos reducidos de datos mencionado en el apartado *1.c.* . Tras la criba de características realizada en el preprocesado debido a los numerosos datos perdidos, nos falta una cantidad considerable de información que podría afectar al rendimiento de nuestros modelos. Es por esto que hemos escogido MLP con las esperanzas de que pueda lidiar con esa pérdida de datos.

Usamos la función *MLPRegressor* que proporciona scikit learn para realizar la evaluación del algoritmo. Los valores asignados a los parámetros de la función son:

- activation, indica la función de activación de las capas ocultas (define la salida de cada nodo dada una entrada o conjunto de entradas). Nosotros hemos usado la 'tanh' equivalente a la tangente hiperbólica que como vimos en teoría es capaz de obtener una buena aproximación para minimizar el error en la muestra.

- solver, el método de optimización. Por defecto hay una versión optimizada de SGD llamada 'adam' pero que está pensada para datasets grandes de más de miles de muestras de entrenamiento, como no es nuestro caso vamos a emplear 'sgd'.
- alpha, que hace referencia al hiperparámetro λ de regularización, el cual determina la dureza con la que se va a penalizar los coeficientes. Resultados obtenidos mediante *GridSearch.*
- learning_rate, el tipo de tasa de aprendizaje. Como se explicó en el apartado *4.a.*, nos interesa escoger una tasa de aprendizaje dinámica que pueda adaptarse a la pendiente conforme se aproxime al mínimo. Por tanto, escogeremos 'adaptive' el cual disminuye η cada vez que no se logra disminuir el error tras un número dado de épocas.
- learning_rate_init, equivale al valor inicial que toma la tasa de aprendizaje. Usando *GridSearch* hemos obtenido que '0.01' es el mejor valor.
- shuffle, vamos a usar el valor por defecto 'True' para que los datos de entrenamiento sean barajados después de cada época fomentando mayor variabilidad.
- tol, dejamos el valor por defecto (1e-4) ya que pensamos que es un valor de función de pérdida lo suficientemente bajo, de hecho en la mayoría de casos probablemente se agote el número máximo de iteraciones.
- early_stopping, sirve para finalizar el entrenamiento cuando la función de pérdida no disminuye después de un número determinado de épocas (dado por el parámetro n_iter_no_change). Nosotros le hemos asignado el valor False porque no queremos que se restrinja ninguna posibilidad de mejora.

alpha \ learning_rate_init	**0.001**	**0.01**
0.0001	0.1195	0.1016
0.001	0.1194	0.1016

e. Métrica de evaluación

Nuestra opción más interesante es el error absoluto medio (MAE) pues, como se explicó en el apartado *4.d.*, es fácilmente interpretable ya que nos devuelve la diferencia promedio exacta entre la predicción y el valor real, manteniendo las unidades y escala del problema.

Devuelve bondades menos arbitrarias como podrían ser las de $R^2 score$ (el cual devuelve una puntuación de, como máximo, 1, y que en muchas ocasiones devuelve errores cercanos a ese máximo cuando el modelo es incorrecto) o MSE, en el cual perdemos las unidades y escala de la variable de salida al hacer el cuadrado.

El mejor contendiente sería RMSE (Root Mean Squared Error), el cual es la raíz cuadrada de MSE y da errores muy similares al MAE. Sin embargo, no describe el error promedio como tal y tiene varias implicaciones que lo hacen más difícil de interpretar.

Por tanto, nos quedaremos con el error absoluto medio (MAE) pues es el más intuitivo y justificable de las opciones que hemos valorado.

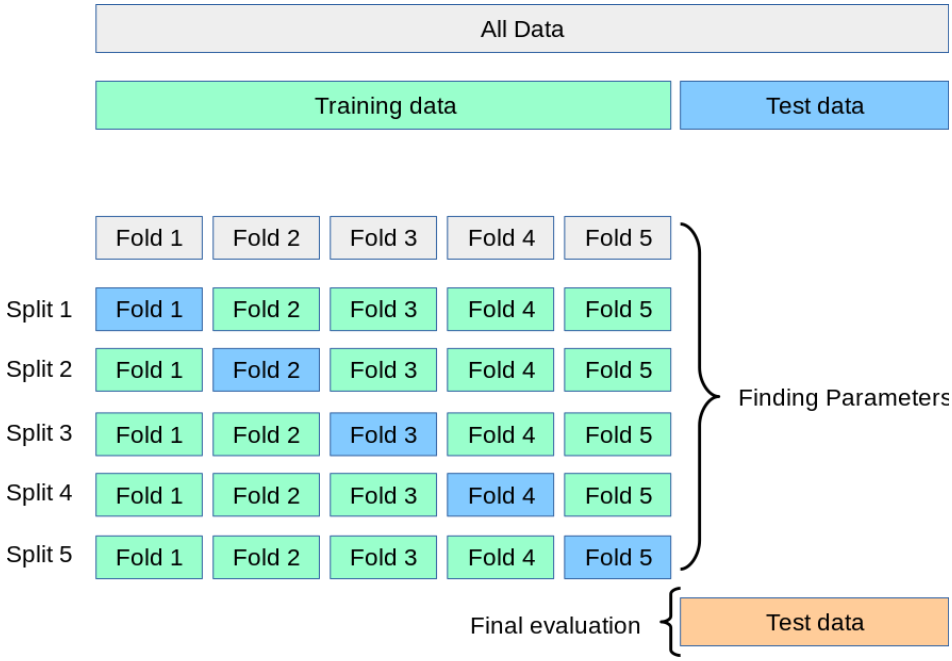
f. Selección del mejor modelo

Para ello se hará uso de la técnica conocida como "validación cruzada". Consiste en dividir el conjunto de entrenamiento en k partes (denominadas *folds*), y obtener el error medio de evaluar el modelo con cada una de esas k partes tras entrenarlo con las otras cuatro. De esta manera obtendremos un error más constante que no será tan dependiente del particionado (como lo sería si simplemente dividiésemos *training* en otros dos subconjuntos de entrenamiento y validación).

Escogeremos $k = 5$ pues es el que se utiliza comúnmente en la literatura y para asegurarnos de que cada *fold* de validación es suficientemente grande.

Tras escoger el modelo con el mejor error de validación medio, probaremos su rendimiento con el conjunto de *test* que habíamos excluido hasta ahora.

Gráficamente:



Estos son los resultados de validación:

1. Random Forest con parámetros. $E_{val} = 0.09344528149138068$
2. SGD con regularización Ridge y parámetros. $E_{val} = 0.09653170268714575$

3. MLP con regularización Ridge y parámetros. $E_{val} = 0.1015765922298119$

Para empezar, es de especial importancia comentar los resultados del modelo lineal, y es que son sorprendentes. Teniendo en cuenta que la variable de salida está normalizada, podemos interpretarla como un porcentaje para juzgar los resultados. Un 9% de error para un dato estadístico como es el número de crímenes violentos por cada cien mil habitantes es más que aceptable, ya que no importa tanto que sea exacto cifra a cifra sino que sea cercano a rasgos generales para poder extraer conclusiones. Además, traza la posibilidad de que haya una relación lineal entre las características y esa variable de salida.

Respecto al modelo de MLP, si tenemos en cuenta que una red neuronal artificial puede llegar a ser un regresor lineal (si eliminamos todas las capas ocultas y todas las funciones de activación, sigue siendo fundamentalmente una red neuronal), entonces no es del todo coherente que hayamos obtenido un peor resultado, aunque sea por poco, que el modelo lineal. Pensamos que se puede deber al ajuste de hiperparámetros, que como mencionamos en el apartado *4.c.* era necesario para que el algoritmo funcionase correctamente y no sobreajustase.

Por otro lado, podría deberse a esa flexibilidad que se comentó en el apartado *5.c*. que nuestro modelo de Random Forest haya conseguido adaptarse a esa relación lineal y superar al modelo de SGD. Los árboles de decisión tienen un gran componente adaptativo por usar el criterio de información y entropía, el cual intenta escoger atributos lo menos impuros y más predecibles posible (en un árbol un nodo se considera “puro” si, en el 100% de los casos, los nodos caen en una categoría específica del campo objetivo), tal que, independientemente del conjunto de datos, los árboles de decisión se construyan con la mayor predictibilidad posible.

g. Resultados para datos de test y comparación con training

- Random Forest con parámetros. $E_{in} = 0.06083858028957692$
- Random Forest con parámetros. $E_{test} = 0.09012677525281448$

El error obtenido en el entrenamiento tras ajustar el modelo al *training* completo es tan solo ligeramente inferior al error en el *test* (3% concretamente, una diferencia inapreciable para el tipo de variable de salida que estamos intentado predecir), por lo que podemos suponer que no se ha dado sobreajuste.

Sabiendo que el E_{test} es una estimación no sesgada del error en la población (pues es un subconjunto que no hemos consultado ni probado hasta la evaluación de la hipótesis), podremos asumir que ese error en la población E_{out} será por lo menos similar. Por tanto, consideraremos el modelo un éxito, pues (como se explicó en el apartado anterior) 9-10% de error es aceptable cuando se trata de datos estadísticos cuantiosos como lo es el número de crímenes violentos por cada cien mil habitantes.

6. Bibliografía

- http://archive.ics.uci.edu/ml/datasets/communities+and+crime
- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
- https://scikit-learn.org/stable/modules/ensemble.html#forest
- https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html
- https://scikit-learn.org/stable/modules/sgd.html
- https://www.cienciadedatos.net/documentos/py14-ridge-lasso-elastic-net-python.html
- https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff
- https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html
- https://scikit-learn.org/stable/modules/tree.html#tree
- https://data.library.virginia.edu/is-r-squared-useless/
- https://scikit-learn.org/stable/modules/neural_networks_supervised.html
- https://es.wikipedia.org/wiki/Random_forest
- https://builtin.com/data-science/random-forest-algorithm
- https://dialnet.unirioja.es/servlet/articulo?codigo=7025156
- https://www.researchgate.net/post/Can_anyone_explain_how_a_neural_network_model_is_better_than_a_regression_model