Proyecto Final: Online News Popularity

Aprendizaje Automático

Miguel Lentisco Ballesteros Antonio Coín Castro

Curso 2019-20

Índice

1. Introducción

En este proyecto perseguimos ajustar el mejor modelo dentro de una clase de modelos, para resolver un problema de clasificación binaria. Para ello seguiremos los siguientes pasos:

1. Analizaremos la bases de datos y entenderemos el contexto del problema a resolver.
2. Preprocesaremos los datos de forma adecuada para trabajar con ellos.
3. Elegiremos unas clases de hipótesis para resolver el problema.
4. Fijaremos algunos modelos concretos dentro de cada clase y seleccionaremos el mejor de cada una según algún criterio.
5. Compararemos los mejores modelos de cada clase y seleccionaremos el mejor de todos.
6. Estimaremos el error del modelo final.

Trabajamos en su mayoría con las funciones del paquete `scikit-learn`, apoyándonos cuando sea necesario en otras librerías como `numpy`, `matplotlib` ó `pandas`. El código de la práctica se ha estructurado en dos *scripts* de `Python` debidamente comentados:

- En `fit.py` se resuelve el problema de clasificación.
- En `visualization.py` se recogen todas las funciones de visualización de gráficas.

La ejecución del programa principal está preparada para que se muestren solo algunas gráficas además del procedimiento de ajuste del modelo (aquellas que consumen menos tiempo). Este comportamiento puede cambiarse mediante el parámetro `SHOW`, eliminando todas las gráficas (valor `Show.NONE`) o mostrándolas todas (valor `Show.ALL`). Además, en las operaciones de cálculo intensivo que lo permitan se utiliza el parámetro `n_jobs = -1` para paralelizar el flujo de trabajo en tantas hebras como se pueda. Por pantalla se muestra información sobre el tiempo de ejecución de los ajustes de los modelos y del programa completo.

Para que los experimentos sean reproducibles se fija una semilla aleatoria al incio del programa. Todos los resultados y gráficas que se muestran a continuación se han obtenido con el valor 2020 para la semilla, y pueden reproducirse si se ejecuta el programa tal y como se proporciona.

2. Base de datos y descripción del problema

Para este problema trabajaremos con la base de datos *Online News Popularity*. Se trata de una recopilación de diferentes estadísticas sobre artículos publicados en el sitio web *mashable* por un período de dos años. El objetivo es predecir la popularidad de los artículos, cuantificada como el número de veces que se comparte cada artículo. Aunque este valor es continuo, tomamos un umbral a partir del cual consideramos que un artículo es popular. Los autores del *dataset* recomiendan en (Fernandes, Vinagre, & Cortez, 2015) fijar este valor a 1400, por lo que así lo hacemos nosotros.

Disponemos de un total de 58 atributos predictivos, todos de tipo numérico (entero o real), que se utilizan para codificar un artículo; por ejemplo, se miden el número de enlaces que contiene, el día de la semana en que se publicó ó el ratio de palabras positivas y negativas, entre otros. Como los atributos no se encuentran normalizados y son de distinto tipo, consideramos como espacio muestral el producto de cada uno de los espacios muestrales individuales, que variará según el atributo concreto: $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_{58}$.En general, los espacios muestrales serán

$[0, 1]$, $[-1, 0]$, $\mathbb{R}^+$, $\mathbb{N}$ ó $\{0, 1\}$. Como conjunto de etiquetas consideramos $\mathcal{Y} = \{1, -1\}$, que representará si un artículo es popular (se comparte más de 1400 veces) o no.

Como tenemos información sobre los valores que queremos predecir en los 39644 ejemplos disponibles, nos encontramos ante un problema de aprenizaje supervisado; en concreto, un problema de clasificación binaria. Queremos aprender o estimar una función $f : \mathcal{X} \to \mathcal{Y}$ que asigne a cada artículo codificado como se ha explicado anteriormente una etiqueta que indique si será popular o no.

2.1. Análisis exploratorio de los datos

Comenzamos analizando la distribución de clases en nuestros datos. Disponemos de un total de 39644 ejemplos, y si fijamos el umbral de popularidad en 1400, el reparto de clases queda configurado como se aprecia en la Tabla 1.

Tabla 1: Distribución de ejemplos por cada clase.

Rango de popularidad	Número de ejemplos	Porcentaje
<1400 (clase -1)	18490	46.6 %
≥ 1400 (clase 1)	21154	53.4 %

Como vemos se trata de un reparto más o menos balanceado de las clases, de forma que podemos emplear las técnicas de ajuste usuales sin preocuparnos por problemas de desbalanceo.

También podemos intentar visualizar el conjunto de datos en dos dimensiones, empleando para ello la técnica TSNE para visualizar datos de alta dimensión. Este algoritmo realiza una proyección en 2 dimensiones del conjunto de datos, minimizando una métrica estadística conocida como *divergencia de Kullback-Leibler*. En la Figura 1 se puede observar el resultado obtenido.

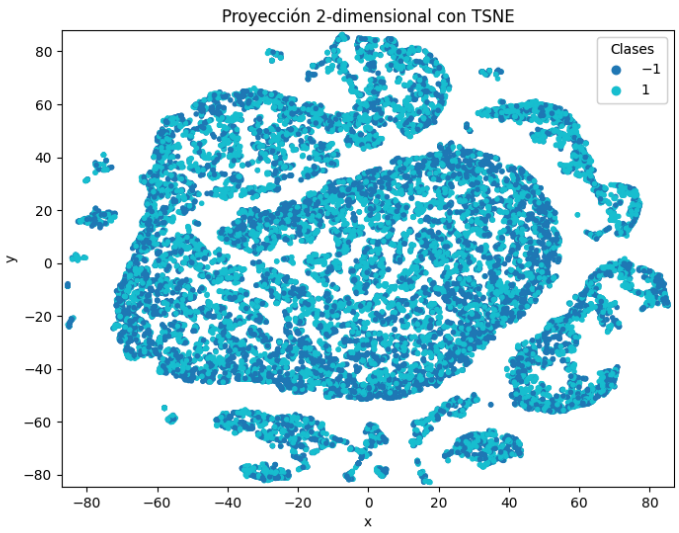

Figura 1: Proyección en 2 dimensiones con TSNE.

Vemos que no obtenemos demasiada información positiva. No se aprecian *clústers* diferenciados, lo que nos lleva a pensar que no vamos a obtener unos resultados excesivamente buenos

con los datos de los que disponemos. No nos sorprenderá si a priori no conseguimos una calidad muy alta, y tendremos que realizar ajustes finos de hiperparámetros y eventuales transformaciones del espacio de entrada para aumentar el rendimiento de los modelos.

2.2. Análisis de las variables

Mostramos a continuación un breve análisis de los predictores disponibles en la base de datos, recogido en la Tabla 2, en el que mostramos para cada atributo su nombre, los valores mínimo y máximo, la media y la desviación típica. Una explicación del significado de cada atributo puede consultarse en el archivo `OnlineNewsPopularity.names` que se proporciona junto a los datos.

Tabla 2: Resumen estadístico de las características.

Característica	Valor mínimo	Valor máximo	Media	STD
n_tokens_title	2.0000	23.0000	10.3987	2.1140
n_tokens_content	0.0000	8474.0000	546.5147	471.1016
n_unique_tokens	0.0000	701.0000	0.5482	3.5207
n_non_stop_words	0.0000	1042.0000	0.9965	5.2312
n_non_stop_unique_tokens	0.0000	650.0000	0.6892	3.2648
num_hrefs	0.0000	304.0000	10.8837	11.3319
num_self_hrefs	0.0000	116.0000	3.2936	3.8551
num_imgs	0.0000	128.0000	4.5441	8.3093
num_videos	0.0000	91.0000	1.2499	4.1078
average_token_length	0.0000	8.0415	4.5482	0.8444
num_keywords	1.0000	10.0000	7.2238	1.9091
data_channel_is_lifestyle	0.0000	1.0000	0.0529	0.2239
data_channel_is_entertainment	0.0000	1.0000	0.1780	0.3825
data_channel_is_bus	0.0000	1.0000	0.1579	0.3646
data_channel_is_socmed	0.0000	1.0000	0.0586	0.2349
data_channel_is_tech	0.0000	1.0000	0.1853	0.3885
data_channel_is_world	0.0000	1.0000	0.2126	0.4091
kw_min_min	-1.0000	377.0000	26.1068	69.6323
kw_max_min	0.0000	298400.0000	1153.9517	3857.9422
kw_avg_min	-1.0000	42827.8571	312.3670	620.7761
kw_min_max	0.0000	843300.0000	13612.3541	57985.2980
kw_max_max	0.0000	843300.0000	752324.0667	214499.4242
kw_avg_max	0.0000	843300.0000	259281.9381	135100.5433
kw_min_avg	-1.0000	3613.0398	1117.1466	1137.4426
kw_max_avg	0.0000	298400.0000	5657.2112	6098.7950
kw_avg_avg	0.0000	43567.6599	3135.8586	1318.1338
self_reference_min_shares	0.0000	843300.0000	3998.7554	19738.4216
self_reference_max_shares	0.0000	843300.0000	10329.2127	41027.0592
self_reference_avg_sharess	0.0000	843300.0000	6401.6976	24211.0269
weekday_is_monday	0.0000	1.0000	0.1680	0.3739
weekday_is_tuesday	0.0000	1.0000	0.1864	0.3894
weekday_is_wednesday	0.0000	1.0000	0.1875	0.3903

Característica	Valor mínimo	Valor máximo	Media	STD
weekday_is_thursday	0.0000	1.0000	0.1833	0.3869
weekday_is_friday	0.0000	1.0000	0.1438	0.3509
weekday_is_saturday	0.0000	1.0000	0.0619	0.2409
weekday_is_sunday	0.0000	1.0000	0.0690	0.2535
is_weekend	0.0000	1.0000	0.1309	0.3373
LDA_00	0.0000	0.9270	0.1846	0.2630
LDA_01	0.0000	0.9259	0.1413	0.2197
LDA_02	0.0000	0.9200	0.2163	0.2821
LDA_03	0.0000	0.9265	0.2238	0.2952
LDA_04	0.0000	0.9272	0.2340	0.2892
global_subjectivity	0.0000	1.0000	0.4434	0.1167
global_sentiment_polarity	-0.3937	0.7278	0.1193	0.0969
global_rate_positive_words	0.0000	0.1555	0.0396	0.0174
global_rate_negative_words	0.0000	0.1849	0.0166	0.0108
rate_positive_words	0.0000	1.0000	0.6822	0.1902
rate_negative_words	0.0000	1.0000	0.2879	0.1562
avg_positive_polarity	0.0000	1.0000	0.3538	0.1045
min_positive_polarity	0.0000	1.0000	0.0954	0.0713
max_positive_polarity	0.0000	1.0000	0.7567	0.2478
avg_negative_polarity	-1.0000	0.0000	-0.2595	0.1277
min_negative_polarity	-1.0000	0.0000	-0.5219	0.2903
max_negative_polarity	-1.0000	0.0000	-0.1075	0.0954
title_subjectivity	0.0000	1.0000	0.2824	0.3242
title_sentiment_polarity	-1.0000	1.0000	0.0714	0.2654
abs_title_subjectivity	0.0000	0.5000	0.3418	0.1888
abs_title_sentiment_polarity	0.0000	1.0000	0.1561	0.2263

Como ya comentábamos antes, cada variable tiene su propio rango de variación, por lo que será necesario posteriormente abordar este problema. En cuanto a la relevancia de las variables, hemos hecho un estudio basándonos en el criterio de importancia que otorga un modelo de Random Forest ajustado a los datos, obteniendo los resultados reflejados en la Figura 2.

A la luz de este análisis nos inclinamos por pensar que todas las variables son relevantes para la predicción en mayor o menor medida, excepto quizás la tercera. Sin embargo, el hecho de que sospechamos que vamos a necesitar toda la información disponible para alcanzar un buen poder de predicción, unido a que los autores se basaron en una serie de métodos y análisis para escoger estas características concretas y no otras, nos motivan a concluir que no tenemos motivos para dudar de la idoneidad de ninguna variable para la predicción.

3. Conjuntos de entrenamiento, validación y *test*

Como disponemos de una cantidad considerable de datos, se ha hecho una partición (usando la función `train_test_split`) de 20/50/30 % para los conjuntos validación, entrenamiento y *test*, respectivamente. Además, esta partición se realiza de forma estratificada para que se mantenga la distribución de clases, indicándolo con el parámetro `stratify`. Podemos observar en

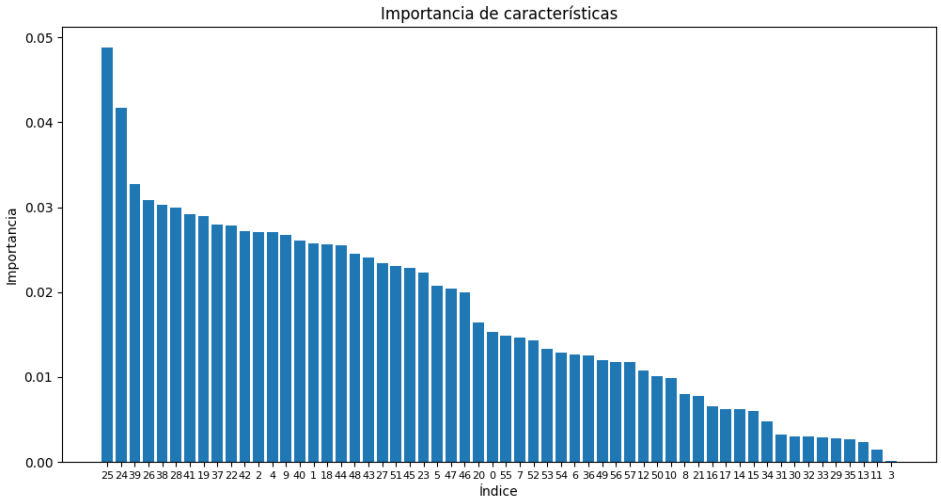

Figura 2: Importancia de características según criterio de Random Forest.

la Figura 3 cómo queda esta distribución en los conjuntos unidos de validación y entrenamiento, y en el conjunto de *test*.

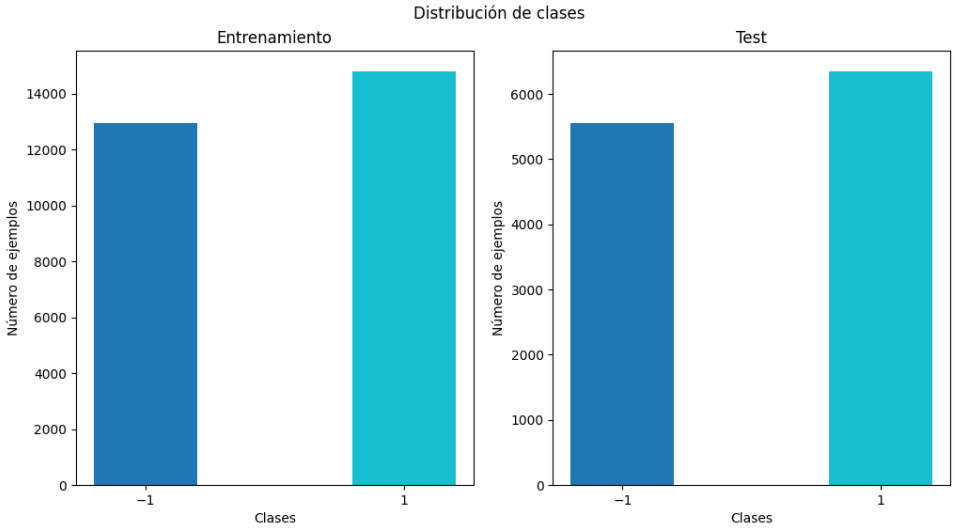

Figura 3: Distribución de clases en entrenamiento y *test*.

El conjunto de *test* se utilizará para evaluar y comparar los mejores modelos de cada clase, y no habrá sido usado en ninguna otra fase del ajuste. El papel que desempeñarán los conjuntos de entrenamiento y validación se describe con detalle en la sección Técnicas de selección de modelos.

4. Lectura y preprocesado de datos

Para cargar los datos nos ayudamos de la librería `pandas` y su función `read_csv`, en la que podemos especificar las columnas concretas que queremos leer, si queremos incluir la cabecera o no, y algunos otros detalles como el separador usado. De esta forma no es necesario modificar el archivo original `OnlineNewsPopularity.csv`.

```
df = read_csv(
    filename, sep = ', ',
    engine = 'python', header = 0,
    usecols = [i for i in range(2, 62)],
    index_col = False,
    dtype = np.float64)
```

Aprovechamos también para transformar la columna a predecir (la última) en una variable discreta que tome únicamente los valores 1 y -1:

```
df.iloc[:, -1] = df.iloc[:, -1].apply(
    lambda x: -1.0 if x < CLASS_THRESHOLD else 1.0)
```

En el archivo de información del *dataset* nos dicen que no hay valores perdidos (también podemos comprobarlo con una llamada a `isnull().values.any()`), por lo que no es necesario realizar tratamiento alguno en este sentido. Tampoco es necesario codificar los datos de entrada para hacerlos útiles a los algoritmos, pues ya vienen expresados en valores numéricos.

Más atención merece el hecho de que los datos no se encuentran en la misma escala, lo que puede provocar problemas con algunos algoritmos (por ejemplo aquellos basados en distancias), y en general es beneficioso siempre normalizar los datos. Esto lo realizamos con la transformación `StandardScaler`, que modifica cada columna restándole su media y dividiendo por la desviación típica, de forma que los datos quedan con media 0 y desviación típica 1.

Además, añadimos previamente una transformación `VarianceThreshold` para eliminar variables con varianza 0 (que son constantes y no nos aportan información relevante para predecir). Aunque en nuestro conjunto inicial no hay variables constantes, la división en varios subconjuntos podría provocar que aparecieran, por lo que lo tenemos en cuenta.

Hay que tener presente que estas transformaciones (y cualquier otra) deben hacerse primero únicamente en el conjunto de entrenamiento, y a la hora de evaluar se tienen que hacer exactamente las mismas transformaciones (con mismos parámetros) en el conjunto donde se valide, para evitar caer en el fenómeno de *data snooping*. Es por esto que haremos uso de los *pipelines* de `sklearn`, que nos permiten aplicar las mismas transformaciones en varias fases de forma cómoda, además de encadenar varias. En concreto, el *pipeline* de preprocesado general quedaría como

```
preproc = Pipeline([
    ("var", VarianceThreshold()),
    ("standardize", StandardScaler())])
```

5. Métricas de error

Ya que las clases están prácticamentes balanceadas, la métrica que usaremos será el *error de clasificación*, una medida simple pero efectiva del error cometido. Si denotamos los ejemplos de entrada a un clasificador h por (x_n, y_n), podemos expresar el error como la fracción total de elementos mal clasificados:

$$E_{class}(h) = \frac{1}{N}\sum_{i=1}^{N}[\![h(x_n) \neq y_n]\!].$$

Sabemos que esta medida de error se encuentra en el intervalo $[0, 1]$, siendo 0 lo mejor posible y 1 lo peor. Se trata de una medida de fácil interpretación; podemos expresarla en porcentaje o invertirla, de forma que $1 - E_{class}$ es lo que se conoce como el *accuracy* del modelo. Presentaremos los resultados utilizando esta última descripción ya que parece más ilustrativa.

Consideraremos además una métrica secundaria de error, también de uso muy extendido: el área bajo la curva ROC (AUC). Esta métrica que nos permite comparar el desempeño de los clasificadores en tanto que varía el umbral de clasificación (aquel valor a partir del cual se considera que un ejemplo pertenece a la clase positiva), proporcionando una medida del poder de discriminación que tiene el clasificador. Para computar esta métrica, se obtiene primero la curva ROC representando el ratio de verdaderos positivos frente al ratio de falsos positivos, para todos los posibles umbrales de clasificación, y finalmente se toma el área bajo esta curva. Esta métrica está también en $[0, 1]$, y será mejor cuanto más alta sea.

Todos los clasificadores empleados tienen internamente una función que asigna un valor numérico a cada punto para cada clase, ya sean probabilidades (`predict_proba`) u otros valores propios de cada clasificador (`decision_function`). Son estas funciones las que se emplean en el cálculo de la métrica AUC, materializado mediante una llamada a la función `roc_auc_score`.

Disponemos de una función `print_evaluation_metrics` que imprime el valor de estas dos métricas para un clasificador y unos conjuntos de datos datos.

6. Técnicas de regularización

El uso de la regularización es esencial para limitar la complejidad de los modelos y evitar el *overfitting*, cosa que nos permitirá obtener una mejor capacidad de generalización. Consideramos las siguientes regularizaciones, que aplicaremos según el modelo que estemos ajustando:

- **Regularización L2 (Ridge)**: se añade una penalización a la función de pérdida que es cuadrática en los pesos,
$$L_{reg}(w) = L(w) + \lambda||w||_2^2.$$
- **Profundidad máxima**: se restringe la profundidad máxima de cada árbol de decisión a un valor λ.
- **Poda de coste-complejidad minimal**: se hace poda en las ramas de cada árbol de decisión de forma que se minimice la función de coste
$$R_\lambda(T) = R(T) + \lambda|T|,$$
siendo $R(T)$ el ratio mal clasificado en las hojas y $|T|$ el número de nodos hoja.

- **Learning rate**: en los clasificadores boosting, reduce la contribución de cada clasificador débil en un factor λ.

El valor de $\lambda > 0$ es un hiperparámetro del modelo, que controla la intensidad de la regularización. Encontrar un valor adecuado es una tarea complicada, pues según varíe podríamos tener sobreajuste o caer en el fenómeno opuesto: tener *underfitting* porque el modelo sea poco flexible y no consiga ajustar bien los datos de entrenamiento.

En la sección Ajuste de modelos comentaremos para cada modelo concreto la regularización utilizada junto con los argumentos que justifican la decisión.

7. Técnicas de selección de modelos

Para elegir el mejor modelo dentro de una clase concreta hemos considerado usar la técnica de *K-fold cross validation*. Esta técnica se basa en dividir el conjunto de entrenamiento en K conjuntos de igual tamaño, y va iterando sobre ellos de forma que en cada paso entrena los modelos en los datos pertenecientes a $K - 1$ de los conjuntos, y los evalúa en el conjunto restante. Finalmente se realiza la media del error a lo largo de todos los mini-conjuntos de validación y se escoge el modelo con menor error. Este error se conoce como *error de cross-validation*, denotado E_{cv}, y sabemos que es un buen estimador del error fuera de la muestra, por lo que tiene sentido utilizarlo para estimar la calidad del ajuste y decidir entre varias configuraciones distintas. Hemos optado por esta técnica frente a otras, además de por su probada eficacia e idoneidad para problemas como el nuestro, porque de esta manera no tenemos que reservar un nuevo conjunto para ir validando cada configuración de hiperparámetros, aprovechando mejor los datos de los que disponemos.

En cada clase de modelos consideraremos uno o varios tipos de clasificadores (por ejemplo, en los clasificadores *boosting* podemos considerar `AdaBoost` y `GradientBoosting`), y para todos realizaremos una búsqueda en el espacio de hiperparámetros para encontrar los que mejor se ajustan a nuestro problema. Es aquí donde entra en juego el conjunto de validación que separamos, pues dividimos este proceso en dos partes:

1. En primer lugar se realiza un preanálisis en el conjunto de validación para hacer una estimación del espacio de búsqueda óptimo de los hiperparámetros de cada modelo, permitiéndonos hacer una búsqueda más grande y rápida que usando todo el *dataset* completo. En general restringimos este análisis a uno o dos parámetros en cada modelo, los que se consideran que tienen más relevancia y/o un espacio de búsqueda que en principio no podemos restringir demasiado.

2. Basándonos en los resultados obtenidos en el paso anterior, configuramos el espacio de búsqueda para los modelos en un entorno reducido de los que se consideran óptimos (no los fijamos porque al entrenar con más datos puede haber fluctuaciones que hagan que otros sean mejores). Aplicamos de nuevo la técnica de *grid search*, esta vez en el conjunto de entrenamiento, y fijamos el modelo que menor error de *cross-validation* tenga como el mejor de su clase.

Una vez obtenido el mejor modelo, **lo reentrenamos con todos los datos de entrenamiento y validación**, para aprovechar todos los datos de los que disponemos e intentar mejorar la

calidad del ajuste. Es importante notar que siguiendo este procedimiento estamos obteniendo a la vez el mejor modelo con los mejores parámetros.

Para todo este proceso es clave la función `GridSearchCV`, la cual puede recibir un *pipeline* como estimador y una lista de diccionarios que represente el espacio de hiperparámetros. Para evitar el fenómeno de *data snooping* que podría contaminar los conjuntos de validación, todo el cauce de preprocesado y selección de modelos se realiza de principio a fin: fijamos al principio las divisiones en K folds y las utilizamos en todo el proceso. Estas divisiones serán en particular instancias de `StratifiedKFold`, que respeta la distribución de clases en las divisiones. También consideramos la función `RandomizedSearchCV`, que funciona de forma similar pero permite que le pasemos los parámetros como distribuciones de las que va extrayendo muestras. De esta forma podemos considerar un espacio continuo de parámetros y especificar el número de muestras a extraer, eliminando la necesidad de discretizar manualmente el espacio si no tenemos información sobre cómo hacerlo.

En la sección Ajuste de modelos veremos cómo se especifica en el código el espacio de parámetros y el procedimiento de ajuste.

8. Ajuste de modelos

Pasamos ahora a describir las clases de modelos que se ajustan, detallando dentro de cada una el procedimiento de ajuste y la justificación de los pasos seguidos. La métrica usada para decidir el mejor modelo será, de forma natural, el *accuracy* medio en los conjuntos de validación. Fijamos el valor de K en 5 para la etapa de *cross-validation*, pues se trata de un valor no demasiado elevado que no dispara el tiempo de entrenamiento, pero lo suficiente como para conseguir unos resultados fiables.

De las clases de modelos propuestas, la única que no analizamos es la de SVM con distintos *kernels* no lineales, pues consideramos que debido a la gran cantidad de puntos que tenemos sería computacionalmente poco viable.

8.1. Modelos lineales

En primer lugar consideramos modelos lineales, que son simples pero muchas veces efectivos y suficientemente buenos para muchos problemas. Vamos a intentar aumentar un poco la complejidad de los modelos para intentar conseguir un mejor ajuste. Para esto, consideramos transformaciones polinómicas de las variables de entrada, concretamente polinomios de grado 2 (no pensamos que un grado mayor merezca la pena en términos de eficiencia, ya que tendríamos demasiadas variables). De esta forma el modelo final seguirá siendo lineal en los pesos, pero permite realizar una clasificación más potente en el espacio transformado y sacar a la luz relaciones entre las variables que resulten en una mejor predicción. Concretamente, si $\mathcal{X}$ es el espacio de entrada y $x = (x_1, \dots, x_d) \in \mathcal{X}$, consideramos la función

$$\Phi_2(x) = (1, x_1, \dots, x_d, x_1^2, \dots, x_d^2, x_1x_2, x_1x_3, \dots, x_1x_d, x_2x_3, \dots, x_{d-1}x_d),$$

de forma que la clase de funciones que ajustan nuestros modelos lineales es

$$\mathcal{H}_{lin} = \left\{\text{signo}(w^T\Phi_2(x)) \,:\, w \in \mathbb{R}^{\tilde{d}}\right\},$$

con

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

Como estamos aumentando significativamente la dimensionalidad del espacio de entrada, pensamos que merece la pena realizar previamente una selección de características para resumir la información disponible y evitar que se dispare el número de características. La estrategia utilizada es *Principal Component Analysis* o PCA, que transforma las variables considerando ciertas combinaciones lineales de las mismas, llamadas componentes principales, de forma que la primera recoge la mayor varianza según una cierta proyección de los datos, la segunda la segunda mayor, etc. Podemos especificar el porcentaje de varianza acumulada que deben explicar las nuevas variables, que en nuestro caso fijamos al 95 %. Con estos parámetros obtenemos una reducción del espacio original de 58 variables a 36, de forma que con ellas se consigue explicar el 95 % de la varianza original, lo que consideramos aceptable y suficiente para nuestro problema. De esta forma, el espacio original se sustituye por otro $\mathcal{X}'$ de dimensión $d' < d$ (en nuestro caso $d' = 36$), y es en este espacio donde aplicamos las transformaciones polinómicas comentadas anteriormente, obteniendo finalmente 702 variables efectivas para la predicción.

A continuación mostramos en la Figura 4 una ilustración de las matrices de correlaciones absolutas en entrenamiento, antes y después del preprocesado realizado. Se observa que aunque aumentamos el número de variables no se disparan las correlaciones, más allá de las evidentes por la forma de las transformaciones realizadas. Cabe destacar que, como comentamos antes, estas transformaciones se incorporan al *pipeline* para realizarlas también en la fase de predicción.

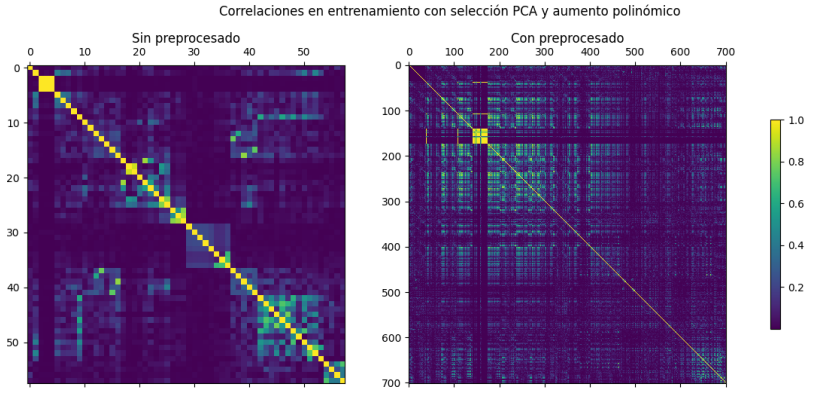

Figura 4: Matriz de correlación en entrenamiento antes y después del preprocesado.

Por otro lado, hemos considerado regularización L2 para los modelos lineales. Este tipo de regularización no conduce a modelos dispersos (al contrario que L1), y pensamos que esto es adecuado porque ya hemos realizado una reducción de dimensionalidad previa a la transformación polinómica, y creemos que la mayoría de las variables son relevantes para la predicción. Además, esta estrategia suele funcionar bien en muchos problemas, disminuyendo la varianza de los modelos y consiguiendo una mejor capacidad de generalización y un mayor valor de la métrica. Por último, otro punto a favor es que no introduce pérdida de derivabilidad, haciendo que su tratamiento computacional sea más flexible y eficiente.

Finalmente fijamos a 1000 el número de iteraciones para todos los modelos que utilicen métodos iterativos en el ajuste.

Regresión Logística

En primer lugar consideramos un modelo de regresión logística, implementado en el objeto `LogisticRegression`. Este modelo predice probabilidades y al final utiliza un umbral para decidir la clase de cada ejemplo. Se trata de un modelo que suele proporcionar muy buenos resultados, adecuado para problemas con variables numéricas con clases balanceadas como el nuestro.

En este caso, la técnica de optimización es la que viene por defecto, que se conoce como LBFGS. Se trata de un algoritmo iterativo similar al método de Newton para optimización, pero que utiliza una aproximación de la inversa de la matriz Hessiana. Se ha elegido porque tiene un tiempo de ejecución asumible y los resultados suelen ser buenos. La función a optimizar es la pérdida logarítmica:

$$L_{log}(w) = \frac{1}{N}\sum_{n=1}^{N} \log(1 + e^{-y_n w^T x_n}),$$

a la que se añade el término de penalización L2. Debemos tener en cuenta que aunque el algoritmo optimice esta función para proporcionar un resultado, la métrica de error que nosotros estamos usando es el accuracy y no el error logarítmico.

El parámetro de regularización, cuyo inverso es lo que en el código se alude como `C`, viene dado por el preanálisis, considerando 40 puntos en el espacio logarítmico $[-5, 1]$ para el mismo:

```
{"clf": [LogisticRegression(penalty = 'l2',
                            random_state = SEED,
                            max_iter = max_iter)],
 "clf__C": np.logspace(-5, 1, 40)}
```

Los resultados obtenidos en el preanálisis se pueden observar en la Figura 5, y nos indica que desde el orden de 10^{-4} en adelante, los resultados son más o menos igual de buenos, alcanzando un máximo entre 10^{-4} y 10^{-3}.

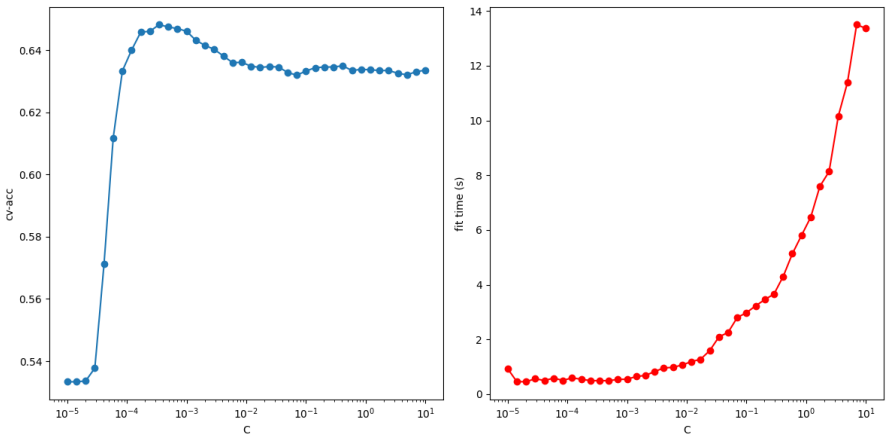

Figura 5: acc-cv/tiempo según C en LogisticRegression.

Por tanto restringimos `C` al espacio logarítmico $[-4, 0]$, reduciendo el número de puntos en el que lo dividimos para no aumentar en exceso el tiempo, quedando finalmente el espacio de búsqueda configurado como sigue:

```
{"clf": [LogisticRegression(penalty = 'l2',
                            random_state = SEED,
                            max_iter = 1000)],
 "clf__C": np.logspace(-4, 0, 9)}
```

Regresión lineal

Consideramos también un modelo de regresión lineal, el más simple dentro de su clase pero muchas veces efectivo. Utilizamos un objeto `RidgeClassifier`, que fija implícitamente la regularización L2. En este caso, la constante de regularización se llama `alpha`, considerando el espacio de búsqueda inicial como 40 puntos en el espacio logarítmico $[-5, 5]$.

```
{"clf": [RidgeClassifier(random_state = SEED,
                         max_iter = max_iter)],
 "clf__alpha": np.logspace(-5, 5, 40)}
```

En este caso se pretende minimizar el error cuadrático de regresión (añadiendo regularización L2):

$$L_{lin}(w) = \frac{1}{N}\sum_{n=1}^{N}(y_n - w^T x_n)^2.$$

Para ello se utiliza la técnica de la Pseudoinversa basada en la descomposición SVD de la matriz de datos, obteniendo una solución en forma cerrada y sin seguir un procedimiento iterativo. Se ha elegido esta técnica en lugar de SGD porque el tiempo de ejecución es más reducido y las soluciones obtenidas son suficientemente buenas.

El resultado del preanálisis mostrado en la Figura 6 nos arroja un máximo cerca de 10^4, por lo que parece razonable restringir el espacio de búsqueda al espacio logarítmico $[0, 5]$.

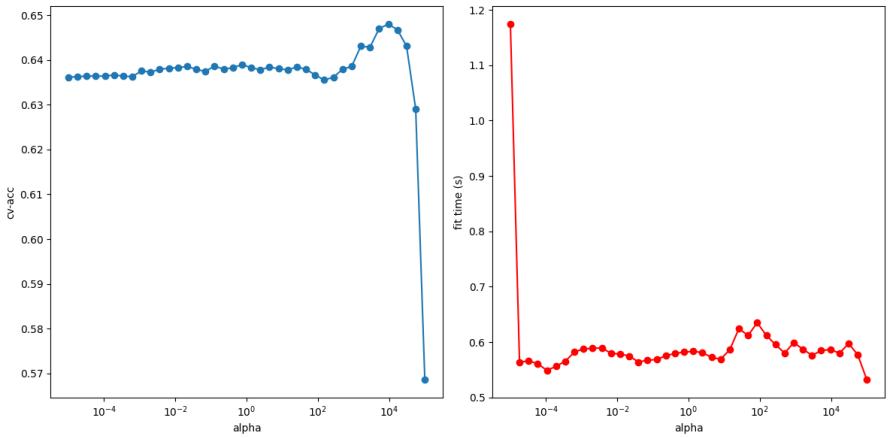

Figura 6: acc-cv/tiempo según alpha en RidgeClassifier.

```
{"clf": [RidgeClassifier(random_state = SEED,
                         max_iter = max_iter)],
 "clf__alpha": np.logspace(0, 5, 9)}
```

SVM Lineal

Finalmente en modelos lineales, consideramos las máquinas de soporte vectorial (SVM) lineales (sin usar *kernel*), utilizando el objeto `SGDClassifier` con la pérdida *hinge* y regularización L2:

$$L_{hinge}(w) = \frac{1}{N}\sum_{n=1}^{N} \text{máx}(0, 1 - y_n w^T x_n).$$

Considerando este modelo perseguimos aumentar el rendimiento, pues se intenta maximizar el margen del hiperplano hipótesis para que sea más robusto en la clasificación, y al usar la técnica SGD en el ajuste no incurrimos en tiempos demasiado grandes para entrenar. Estudiamos el parámetro de regularización `alpha` con 40 puntos en el espacio logarítmico $[-6, 2]$, obteniendo los resultados de la figura 7.

```
{"clf": [SGDClassifier(random_state = SEED,
                       penalty = 'l2',
                       max_iter = max_iter)],
 "clf__alpha": np.logspace(-6, 2, 40)}
```

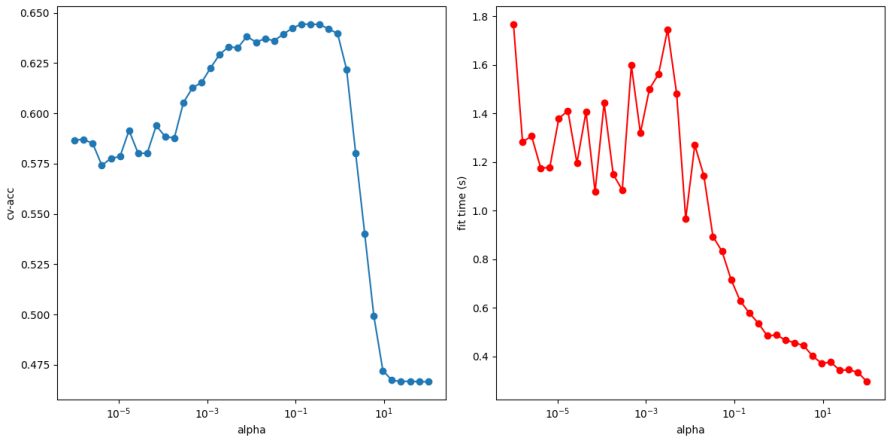

Figura 7: acc-cv/tiempo según alpha en SGDClassifier.

Los resultados del preanálisis nos indican un máximo cerca de 10^{-1} y 1, por lo que restringimos al espacio logarítmico $[-5, 1]$ (aportando cierta flexibilidad a la hora de escoger el espacio). También hemos considerado como hiperparámetro adicional el tipo de tasa de aprendizaje: `optimal` (tasa escogida por heurística), `adaptive` (decrementa cuando el error no decrementa) e `invscaling` (decrementa dividiendo por la raiz del número de iteraciones). De esta forma aportamos flexibilidad en el ajuste, intentando que al principio se acerque rápidamente a un óptimo y que después vaya disminuyendo la tasa de aprendizaje para asegurar una buena convergencia. La tasa de aprendizaje inicial, `eta0`, la fijamos a 0.1, un valor no demasiado pequeño para dar pie a ir disminuyendo progresivamente sin reducirse a valores despreciables.

```
{"clf": [SGDClassifier(random_state = SEED,
                       penalty = 'l2',
```

```
                        max_iter = max_iter,
                        eta0 = 0.1)],
 "clf__learning_rate": ['optimal', 'invscaling', 'adaptive'],
 "clf__alpha": np.logspace(-5, 1, 7)}
```

8.2. Random Forest

El primer modelo no lineal que consideramos es Random Forest (RF). Este modelo se construye como un *ensemble* árboles de decisión, utilizando la técnica de *bootstrap*. Mediante dicha técnica construimos N árboles a partir de una única muestra (realizando muestreo aleatorio con reemplazamiento), y para la construcción de cada árbol realizamos además una selección aleatoria de características. La salida del clasificador será un agregado de los resultados de cada árbol individual; en nuestro caso la clase mayoritaria.

Los árboles de decisión particionan el espacio de entrada por hiperplanos paralelos a los ejes, de forma que la clase de funciones que ajustan los árboles de clasificación es:

$$\left\{ \arg\max_k \frac{1}{N_m} \sum_{x_n \in R_m} [\![y_n = k]\!] \ : \{R_m \ : \ m = 1, \dots, M\} \text{ es una partición de } \mathcal{X} \right\},$$

donde N_m es el número de ejemplos que caen en la región R_m y $k \in \{-1, 1\}$ son las clases. Estos árboles consiguen un sesgo muy bajo al ir aumentando la profundidad, y al incorporar varios de ellos con la técnica de selección de características de Random Forest reducimos también la varianza, por lo que al final obtenemos un clasificador de muy buena calidad y que esperamos que obtenga buenos resultados. Una cosa que hay que tener en cuenta es que los árboles son muy propensos al *overfitting*, por lo que debemos aplicar técnicas de regularización para evitar este fenómeno y garantizar una buena generalización.

Consideramos el objeto `RandomForestClassifier`, fijando el número de características de cada árbol a $\sqrt{d}$ (usando la regla heurística vista en clase) y el criterio *gini* para decidir las divisiones del árbol (la otra opción sería el criterio de entropía, pero sabemos que en clasificación binaria son equivalentes). Consideramos como hiperparámetros más relevantes el número de árboles totales, `n_estimators`, y la profundidad máxima de cada árbol, `max_depth` (como regularización). Para el estudio inicial tenemos el siguiente espacio:

```
{"clf": [RandomForestClassifier(random_state = SEED)],
 "clf__max_depth": [5, 10, 15, 20, 30, 40, 58],
 "clf__n_estimators": [100, 200, 300, 400, 500, 600]}
```

Los resultado del preanálisis los podemos ver en la Figura 8 y la Figura 9, que nos muestran que en general exceptuando la profundidad máxima 5, el resto de configuraciones proporcionan buenos resultados, destacando profundidad máxima 20 con 400 árboles y profundidad máxima ≥ 20 con 600 árboles.

Sabemos que al aumentar el número de árboles conseguimos reducir el término de la varianza aunque a coste de incrementar el tiempo de computación; pero en este caso no nos importa pagar el precio y permitimos variar entre 400 y 600 estimadores. Además, como conforme aumenta la profundidad de los árboles conseguimos menos error pero más varianza y para la profunidad máxima 20 obtenemos de los mejores resultados, optamos por fijar este valor.

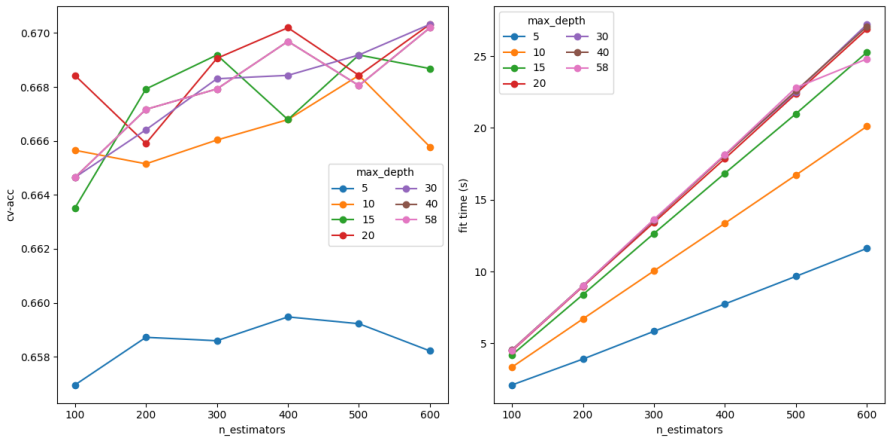

Figura 8: acc-cv/tiempo según n_estimators y max_depth en RandomForest.

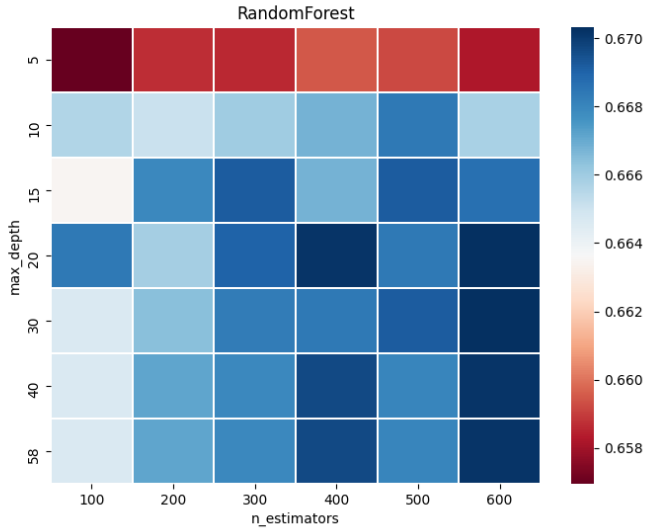

Figura 9: Mapa de calor para acc-cv según n_estimators y max_depth en RandomForest.

También añadimos una nueva posibilidad de regularización, la poda de coste-complejidad, para dar la posibilidad de disminuir aún más el sobreajuste. Los parámetros efectivos para el valor `ccp_alpha` se han encontrado mediante una llamada al método `cost_complexity_pruning_path` de la clase `DecisionTreeClassifier`. Finalmente la configuración de hiperparámetros quedaría:

```
{"clf": [RandomForestClassifier(random_state = SEED,
                                max_depth = 20)],
     "clf__n_estimators": [400, 600],
     "clf__ccp_alpha": [0.0, 1e-4, 5e-4, 1e-3, 5e-3, 1e-2]}
```

8.3. Boosting

La siguiente clase de modelos no lineales que consideramos vuelve a ser una clase de clasificadores *ensemble*, obtenidos como uniones de varios clasificadores. En este caso, la técnica de *boosting* se basa en la agregación secuencial de varios clasificadores *débiles* (árboles de decisión con profundidad muy reducida) en versiones sucesivas modificadas de los datos (aplicando distintos pesos a los mismos). Lo que hacemos en este caso añadiendo más estimadores es intentar reducir el sesgo del estimador combinado, pues la varianza será pequeña al ser clasificadores muy simples.

De forma esquemática, se utiliza un algoritmo *greedy* que en cada paso ajusta un clasificador simple y lo añade al agregado anterior de forma ponderada,

$$F_m(x) = F_{m-1}(x) + \alpha_m h_m(x),$$

donde el nuevo clasificador h_m y la constante α_m se ajustan para minimizar una cierta función de pérdida, cuya forma genérica es

$$\sum_{n=1}^{N} L(y_n, F_{m-1} + \alpha h(x_n))$$

Esta clase de modelo puede considerarse como un enfoque dual a las agregaciones que realizan los Random Forest, y también ha demostrado dar muy buenos resultados. Retiene todas las ventajas de utilizar árboles de decisión como clasificadores principales, pero propone un enfoque novedoso para intentar mejorar la calidad del ajuste, por lo que pensamos que merece la pena probarlo en nuestro problema.

AdaBoost

La función de pérdida que intenta minimizar este modelo es la pérdida exponencial:

$$L_{exp}(y_n, h(x_n)) = \exp(-y_n h(x_n)).$$

Como ya comentamos, se ajustan una serie de clasificadores débiles en sucesivas versiones modificadas de los datos. Los pesos de modificación se adaptan de forma que en iteraciones sucesivas se aumenta el peso de los puntos de entrenamiento cuya clase se va prediciendo incorrectamente, y se decrementa el peso de las predicciones correctas. Se trata de un modelo flexible cuya eficiencia puede ser probada matemáticamente.

Usamos AdaBoost con el objeto `AdaBoostClassifier`, usando como clasificador débil un árbol de decisión `DecisionTreeClassifier`. Los hiperparámetros que dejamos para buscar n el análisis inicial son `max_depth`, la profundidad máxima de los clasificadores, y `n_estimators` el número de clasificadores. Probamos con profundidades muy bajas y un tamaño alto de clasificadores:

```
{"clf": [AdaBoostClassifier(random_state = SEED,
                            base_estimator = DecisionTreeClassifier())],
 "clf__base_estimator__max_depth": [1, 2, 3, 4, 5],
 "clf__n_estimators": [100, 200, 300, 400, 500]}
```

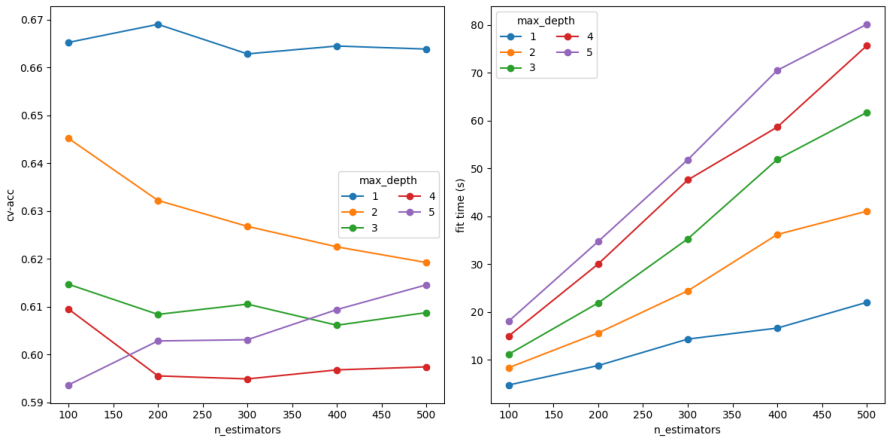

Figura 10: acc-cv/tiempo según n_estimators y max_depth en AdaBoost.

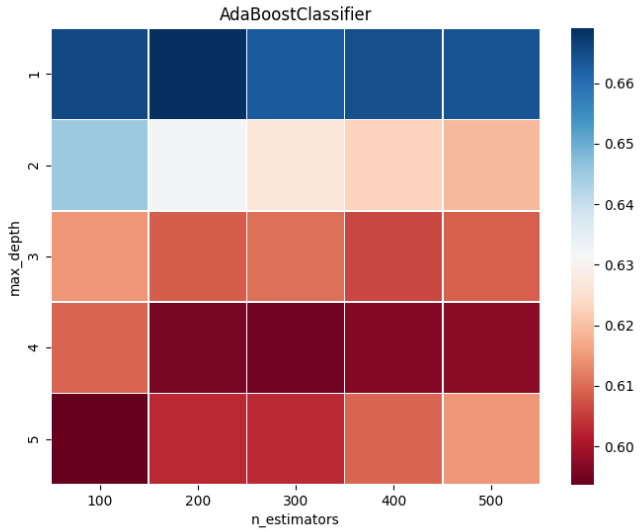

Figura 11: Mapa de calor para acc-cv según n_estimators y max_depth en AdaBoost.

Los resultados obtenidos pueden verse en la Figura 10 y la Figura 11, que nos dejan claro que la mejor elección son funciones *stump*: árboles con profundidad 1.

Por otra parte, el número óptimo de árboles resulta ser unos 200, luego para la búsqueda del mejor clasificador dejamos que este parámetro varíe en un entorno de ese valor. Además, añadimos el hiperparámetro del *learning rate* como forma de regularización del modelo, permitiendo que se pueda escalar a la mitad la contribución de cada clasificador débil.

```
{"clf": [AdaBoostClassifier(random_state = SEED)],
 "clf__n_estimators": [175, 200, 225],
 "clf__learning_rate": [0.5, 1.0]}
```

Gradient Boosting

Gradient Boosting surge como una generalización de los modelos de *boosting* que permite usar funciones de pérdida arbitrarias, siempre que sean derivables. Realiza una aproximación

del gradiente utilizando la fórmula de Taylor y obtiene una forma cerrada para la optimización.

Para el ajuste usamos el objeto `GradientBoostingClassifier` fijando la función de pérdida a `deviance`, que es un alias para la función de pérdida logística que ya comentamos anteriormente (la única disponible distinta de la exponencial):

$$L_{log}(y_n, h(x_n)) = \log(1 + \exp(-y_n h(x_n))).$$

Los hiperparámetros que variamos inicialmente son la profundidad de los árboles, `max_depth`, y el número de árboles, `n_estimators`:

```
{"clf": [GradientBoostingClassifier(random_state = SEED)],
 "clf__max_depth": [1, 2, 3, 4, 5],
 "clf__n_estimators": [100, 200, 300, 400, 500]}
```

En los resultados del preanálisis, vistos en la Figura 12 y la Figura 13, observamos que no es ninguna sorpresa que para los árboles más profundos se obtienen mejores resultados con menos árboles, y para los más simples se necesitan muchos más. En este caso la profundidad de interacción óptima no es necesariamente igual a 1.

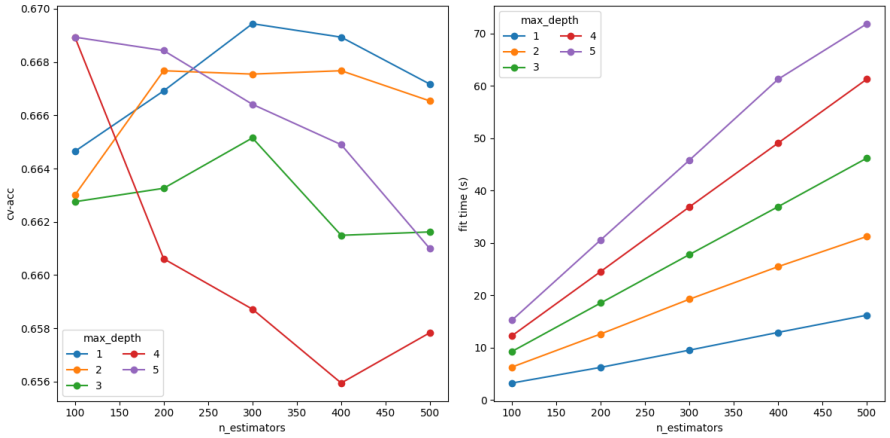

Figura 12: acc-cv/tiempo según n_estimators y max_depth en GradientBoosting.

En vista de lo obtenido, probamos una configuración con pocos árboles (100) para profundidad alta y otra con muchos (300) para profundidad baja:

```
{"clf": [GradientBoostingClassifier(random_state = SEED,
                                    n_estimators = 100)],
 "clf__learning_rate": [0.05, 0.1, 1.0],
 "clf__subsample": [1.0, 0.75],
 "clf__max_depth": [4, 5]},
{"clf": [GradientBoostingClassifier(random_state = SEED,
                                    n_estimators = 300)],
 "clf__learning_rate": [0.05, 0.1, 1.0],
 "clf__subsample": [1.0, 0.75],
 "clf__max_depth": [1, 2]}
```

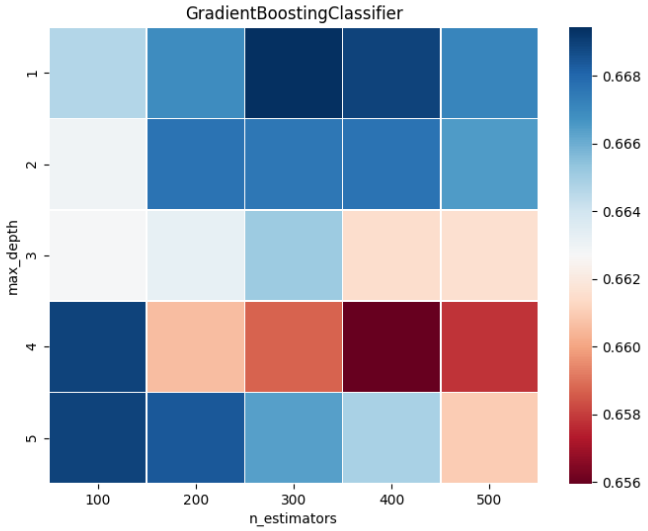

Figura 13: Mapa de calor para acc-cv según n_estimators y max_depth en GradientBoosting.

Hemos añadido los hiperparámetros `learning_rate`, la tasa de aprendizaje (misma interpretación que en AdaBoost, pero ahora con valores más pequeños porque usamos profundidades más grandes) y `subsample`, la proporción de datos usados en el entrenamiento de un clasificador, para intentar bajar la varianza y ampliar un poco más el espacio de búsqueda. Según (Friedman, 2002), una combinación del *subsampling* con un *learning rate* menor que 1 proporcionan una versión estocástica que puede aumentar la calidad del modelo.

8.4. Perceptrón multicapa

Los clasificadores basados en perceptrón multicapa ajustan funciones lineales por capas, introduciendo una activación no lineal al final de cada capa para diversificar la clase de funciones que ajustan. Nosotros utilizaremos clasificadores con dos capas ocultas, y una capa de salida con una neurona que nos proporciona la clase de los puntos que queramos predecir. Entonces, la clase de funciones ajustada es

$$\left\{\text{signo}\left(W_{(3)}^T\theta\left(W_{(2)}^T\theta\left(W_{(1)}^T x\right)\right)\right), W_{(i)} \in \mathscr{M}_{n_{i-1}\times n_i},\ n_0 = d, n_3 = 1\right\},$$

donde θ es la función de activación elegida. Para ajustar los pesos, se utiliza la técnica de *backpropagation* para propagar hacia atrás el error, que en este caso es el error cuadrático entre predicciones y etiquetas:

$$L(y_n, h(x_n)) = (y_n - h(x_n))^2$$

Utilizamos el objeto `MLPClassifier` con los siguientes parámetros fijos:

- `solver = sgd`. Elegimos como optimizador descenso estocástico de gradiente, una técnica fiable y con probada eficiente, que proporciona un tiempo de ejecución razonable.
- `learning_rate = 'adaptive'`. Especifica que la tasa de aprendizaje es *adaptable*, es decir, va disminuyendo conforme aumentan las iteraciones si el error no disminuye.

- `learning_rate_init = 0.01`. Es la tasa inicial de aprendizaje. La incrementamos ligeramente respecto al valor por defecto para permitir que vaya decreciendo desde un valor más elevado.
- `max_iter = 300`. Número máximo de iteraciones para SGD.
- `activation = relu`. Función de activación ReLU al final de cada capa, que viene dada por la expresión $ReLu(x) = \text{máx}\{0, x\}$. Es una de las más usadas en la actualidad y suele proporcionar mejores resultados que otras como `tanh`.
- `tol = 1e-3`. Tolerancia para la convergencia, que reducimos porque no nos interesa seguir optimizando el *accuracy* más allá de las milésimas.

Dejamos como hiperparámetro en el análisis inicial el número de neuronas en las capas ocultas, `hidden_layer_sizes`, tomando el mismo tamaño para ambas capas y haciendo una búsqueda aleatoria en el rango $[50, 100]$.

```
{"clf": [MLPClassifier(random_state = SEED,
                       learning_rate_init = 0.01,
                       solver = 'sgd',
                       max_iter = 300,
                       learning_rate = 'adaptive',
                       activation = 'relu',
                       tol = 1e-3)],
 "clf__hidden_layer_sizes": multi_randint(50, 101, 2)}
```

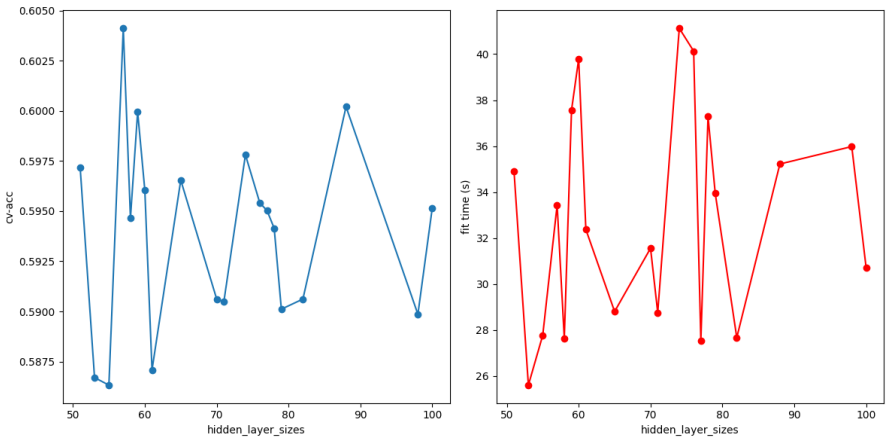

Figura 14: acc-cv/tiempo según hidden_layer_sizes en MLP.

El resultado del preanálisis en la Figura 14 nos deja dos tamaños con los mejores resultados en sus entornos, en 57 y 88, por tanto probamos con estas configuraciones. Además, añadimos un parámetro `alpha` que controla la regularización (de tipo L2), pues estos modelos son conocidos por su tendencia al *overfitting*. Permitimos que este parámetro varíe en el espacio logarítmico $[-2, 2]$, y elegimos 10 configuraciones de manera uniforme. También aumentamos el *learning rate* porque ahora trabajamos con más datos y preferimos empezar desde un valor más elevado.

```
{"clf": [MLPClassifier(random_state = SEED,
                       learning_rate_init = 0.1,
```

```
                        solver = 'sgd',
                        max_iter = 300,
                        learning_rate = 'adaptive',
                        activation = 'relu',
                        tol = 1e-3)],
 "clf__hidden_layer_sizes": [(57, 57), (88, 88)],
 "clf__alpha": loguniform(1e-2, 1e2)}
```

8.5. K-Nearest Neighbors

Después de considerar modelos lineales, modelos basados en *ensembles* y modelos sofisticados como el perceptrón multicapa, pasamos a un modelo no lineal bastante simple, pero que cambia el enfoque: se basa en medidas de similaridad para clasificar los puntos. La fase de entrenamiento de este clasificador consiste en memorizar los puntos de entrenamiento, y a la hora de predecir, para cada punto considera sus k vecinos más cercanos de entre los que tenía guardados, y elige como clase la clase mayoritaria entre ellos. De esta forma realiza una partición del espacio de entrada, cuya forma dependerá del valor de k.

Aplicamos el algoritmo mediante el objeto `KNeighborsClassifier` usando la métrica euclídea, y especificamos el espacio de búsqueda para el hiperparámetro k considerando la regla heurística que recomienda usar $k = \sqrt{N}$. Vamos desde 1 hasta 200 pasando por $k \approx 90$ (aproximadamente la raíz cuadrada del tamaño del *dataset* de preanálisis):

```
{"clf": [KNeighborsClassifier()],
 "clf__n_neighbors": [1, 3, 5, 10, 20, 25, 30, 40, 50, 100, 200]}
```

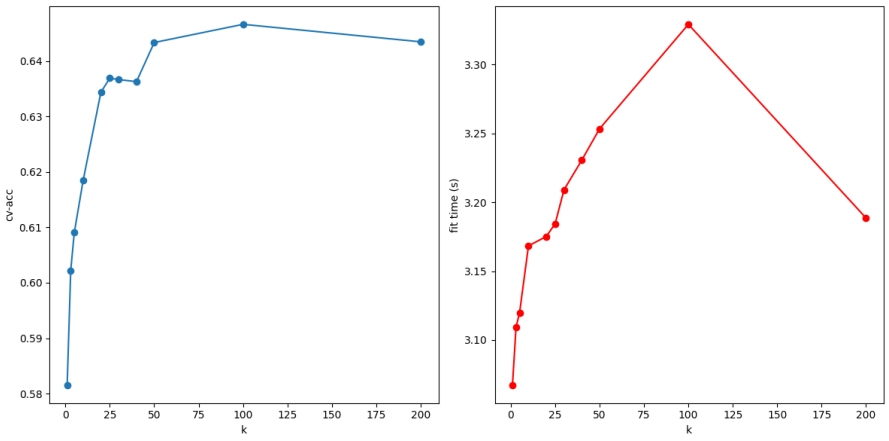

Figura 15: acc-cv/tiempo según k en KNN.

Los resultados de preanálisis en la Figura 15 nos confirman la regla experimental, ya que el óptimo está en torno a $k = 100$; por tanto para el conjunto de entrenamiento usamos un entorno de $k \approx 141$ (raíz cuadrada del tamaño de dicho conjunto), pero permitiendo cierta variabilidad hacia abajo para no obligar al modelo a escoger demasiados vecinos. Por tanto, el espacio de búsqueda queda como sigue:

```
{"clf": [KNeighborsClassifier()],
 "clf__n_neighbors": [80, 100, 120, 150],
 "clf__weights": ['uniform', 'distance']}
```

Hemos añadido para dar más variabilidad en la búsqueda el hiperparámetro `weights`, que permite ponderar la contribución de los k vecinos encontrados: con `uniform` todos importan por igual, mientras que con `distance` los vecinos importan de forma inversamente proporcional a su distancia al punto en cuestión.

8.6. Redes de funciones de Base Radial

Por último, consideramos un clasificador también basado en la similaridad, pero utilizando estrategias de *kernel* para mejorar el rendimiento. Se trata de las redes de funciones de base radial (RBF-Network), que en nuestro caso se basan en el núcleo gaussiano

$$\phi(z) = e^{\frac{-1}{2}z^2}.$$

La idea es fijar una serie de centros μ_j en los datos de entrada, y considerarlos como referencia para aplicar el núcleo utilizando la distancia de cada punto a los centros, reescalada por un parámetro r. Concretamente, la clase de funciones que ajustan estos clasificadores es

$$\left\{ w_0 + \sum_{j=1}^{k} w_j \phi\left(\frac{||x - \mu_j||}{r}\right) \,:\, w_0, w_j \in \mathbb{R},\ \mu_j \in \mathbb{R}^d,\ j = 1, \dots, k \right\}.$$

Podemos entenderlo como una especie de red *feed-forward*, que añade complejidad sobre los modelos lineales de forma similar a las transformaciones no lineales que consideramos para ellos, pero haciendo que estas transformaciones dependan de los datos de entrenamiento (a través de los centros μ_j). Como los centros son un parámetro del modelo y estos aparecen dentro de una función no lineal, este modelo no es lineal en sus parámetros.

Este clasificador no está implementado en `sklearn`, por lo que construimos una clase `RBFNetworkClassifier` siguiendo el algoritmo de referencia en (Abu-Mostafa, 2012), que mostramos en el Algoritmo 1. Hemos considerado el algoritmo de K-medias (`KMeans`) para encontrar los k centroides, y la sugerencia de fijar

$$r = \frac{R}{k^{1/d}},\ \text{con } R = \max_{i,j} ||x_i - x_j||.$$

Finalmente, el algoritmo lineal escogido en el espacio transformado Z ha sido Regresión Lineal + L2 (`RidgeClassifier`), ya que usamos el método de la psuedoinversa que es más sencillo y rápido, aunque podría haberse usado cualquier otro modelo lineal como clasificador.

Algorithm 1: Fit RBF-Network(X, y, k)

$\mu = \mu_1, \dots, \mu_k = \text{KMeans}(X, k)$;
$R = \max_{i,j} ||x_i - x_j||$;
$r = \dfrac{R}{k^{1/d}}$;
$Z = \text{KernelRBF}\left(\frac{d(X,\mu)}{r}\right)$;
Fit LinealRegresion(Z, y);

Una descripción más específica de la implementación se puede encontrar en el Anexo: Funcionamiento del código. Los hiperparámetros a considerar son tanto el número de centroides `k` como el parámetro de regularización en el modelo lineal `alpha` (muy necesario cuando `k` aumenta), cuya configuración de búsqueda inicial es:

```
{"clf": [RBFNetworkClassifier(random_state = SEED)],
 "clf__k": [5, 10, 25, 50, 100, 200, 300, 400],
 "clf__alpha": [0.0, 1e-10, 1e-5, 1e-3, 1e-1, 1.0, 10.0]}
```

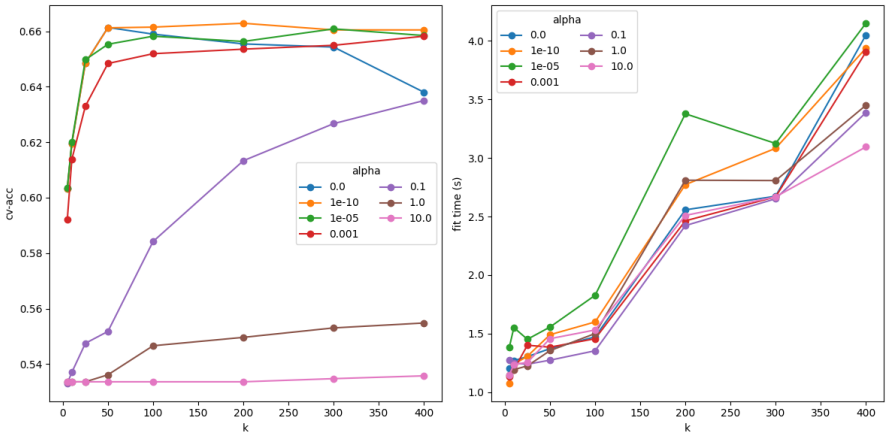

Figura 16: acc-cv/tiempo según k y alpha en RBF.

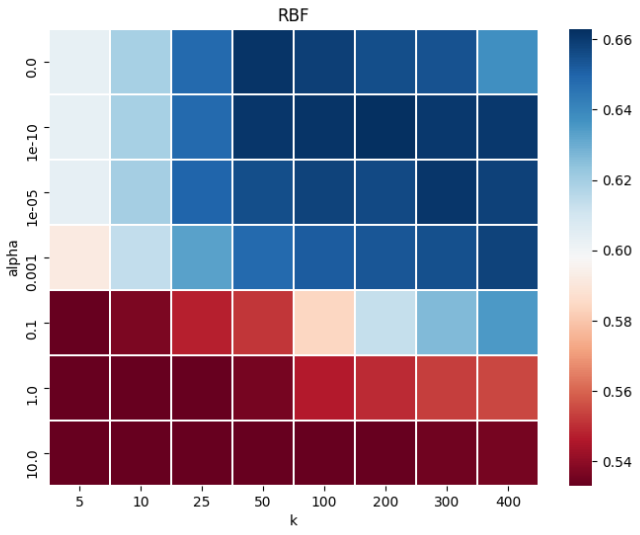

Figura 17: Mapa de calor para acc-cv según k y alpha en RBF.

Los resultados del preanálisis en la Figura 16 y la Figura 17 nos indican como los mejores valores se obtienen con $\texttt{k} \geq 50$ y $\texttt{alpha} \leq 10^{-5}$, donde además vemos que los resultados buenos tienden a estabilizarse. Por tanto el espacio de búsqueda final queda:

```
{"clf": [RBFNetworkClassifier(random_state = SEED)],
 "clf__k": [50, 100, 200, 300, 400],
 "clf__alpha": [0.0, 1e-10, 1e-5]}
```

9. Análisis de resultados

En esta sección mostramos los resultados de la técnica de selección que hemos realizado. En la Tabla 3 se pueden observar las mejores configuraciones para cada modelo, las que han obtenido menor error de *cross-validation* dentro de su clase.

Modelo	Hiperparámetros
Lineal	LogisticRegression, C = 0.1
RandomForest	max_depth = 20, n_estimators = 600, ccp_alpha = 0
Boosting	GradientBoosting, max_depth = 4, n_estimators = 100, subsample = 0.75
MLP	hidden_layer_sizes = (88, 88), alpha = 3.0
KNN	n_neighbors = 150, weights = 'distance'
RBF-Network	k = 300, alpha = 1e-10

Tabla 3: Mejores configuraciones obtenidas en CV.

Podemos estudiar las curvas de aprendizaje de cada modelo para ver si hay *underfitting*, *overfitting* o si más o menos se ha realizado un buen ajuste. Mostramos estas curvas junto con una medida del tiempo de entrenamiento en las Figuras 18 a 23.

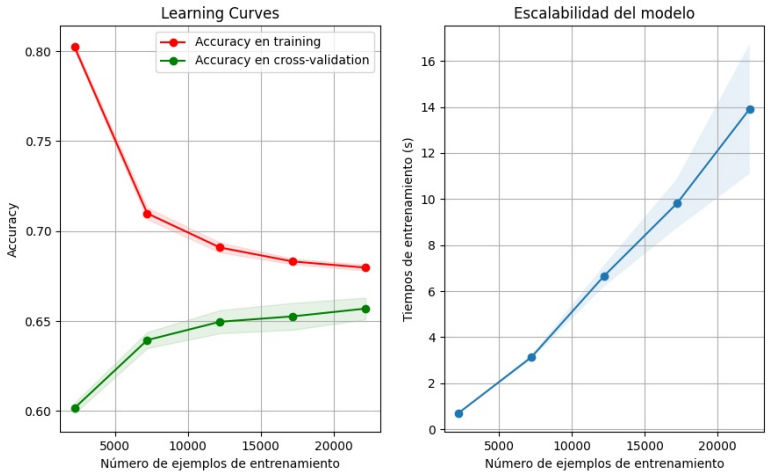

Figura 18: Curvas de aprendizaje para LogisticRegression.

Sabemos que los modelos de RandomForest presentan casi siempre un alto sobreajuste, a pesar de la intensa regularización aplicada. También observamos un fenómeno parecido en las curvas de KNN, pero esta vez la explicación es distinta: en la fase de entrenamiento habíamos memorizado los datos, por lo que siempre vamos a clasificar correctamente el 100 % de los puntos. En el resto de modelos observamos cómo las curvas de *accuracy* en entrenamiento y validación comienzan separadas pero se van acercando a medida que aumenta el número de ejemplos,

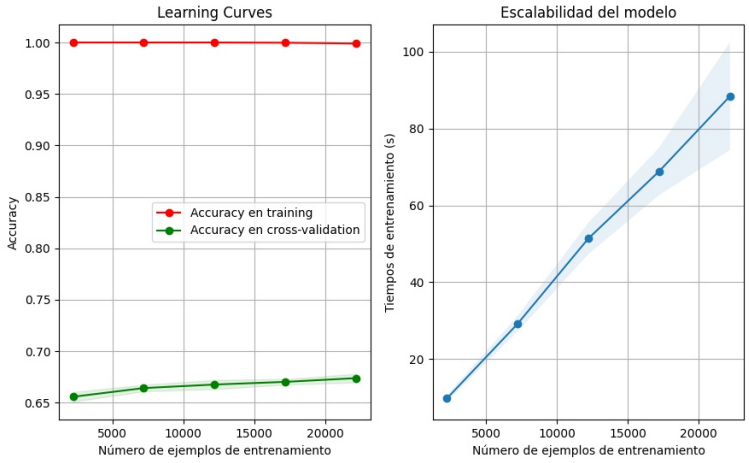

Figura 19: Curvas de aprendizaje para RandomForest.

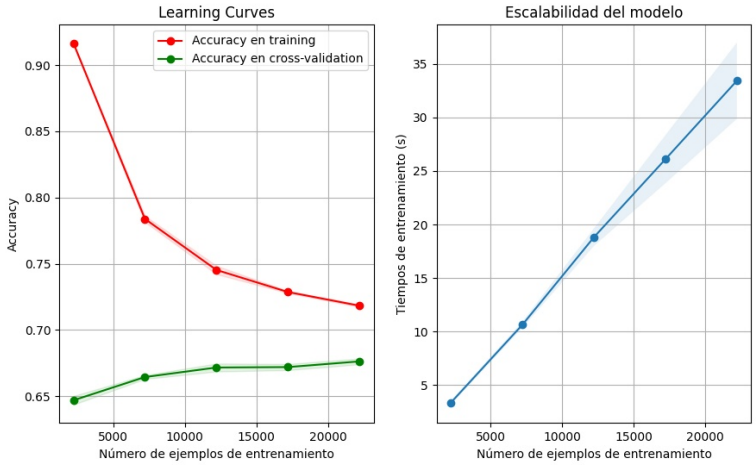

Figura 20: Curvas de aprendizaje para GradientBoostingClassifier.

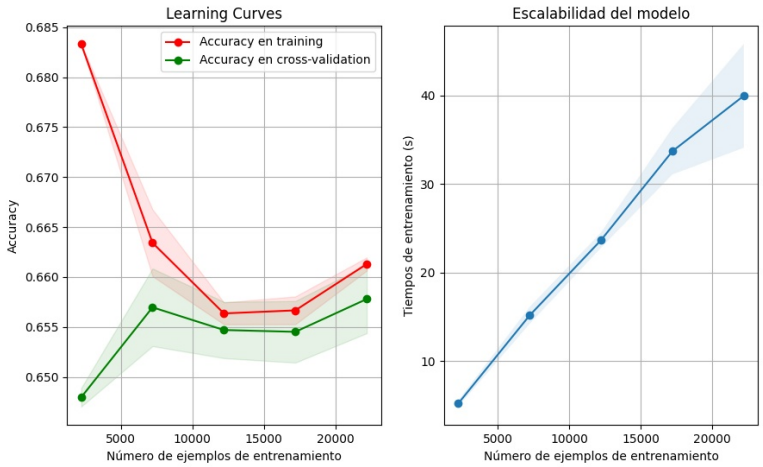

Figura 21: Curvas de aprendizaje para MLPClassifier.

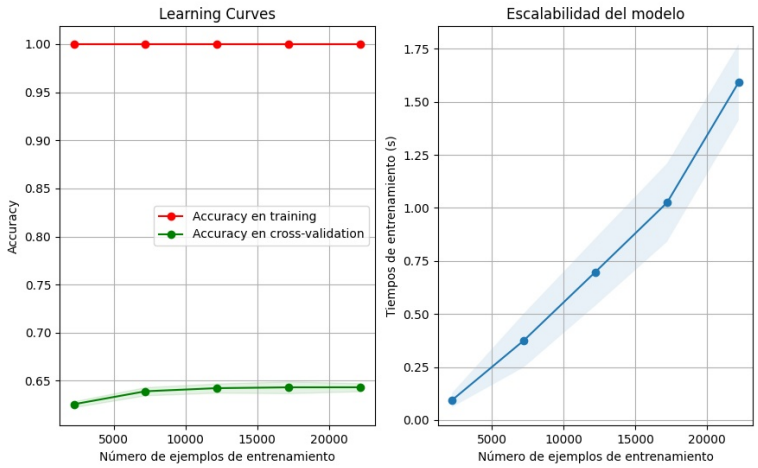

Figura 22: Curvas de aprendizaje para KNeighborsClassifier.

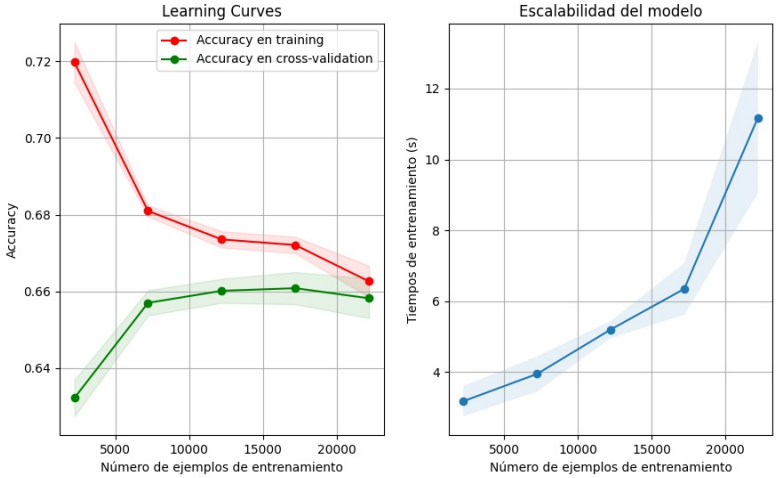

Figura 23: Curvas de aprendizaje para RBFNetworkClassifier.

hasta acabar prácticamente por juntarse. Esto nos indica que los modelos han aprendido bien de los datos, y salvo excepciones como el MLP, no se beneficiarían demasiado de más datos de entrenamiento.

Por su parte, los resultados para las métricas escogidas en entrenamiento y test se recogen en la Tabla 4. Incorporamos un clasificador aleatorio con el objeto `DummyClassifier` como clasificador base para comparar el resto de modelos. Se espera que este clasificador tenga un acierto de 50 % (2 clases) por lo que cualquier modelo que se precie debería obtener un acierto por encima de este.

Comenzamos por los modelos que han tenido un peor desempeño: KNN, RBF-Network y MLP. No nos sorprende que el modelo de KNN haya sido el peor, pues es un modelo que sigue una filosofía muy simple, que a veces resulta efectiva, pero parece que no se adecúa demasiado a nuestro problema. Los otros dos son modelos basados en redes, y parece que aunque hemos intentado conseguir el mejor ajuste dentro de cada clase, los resultados no son todo lo altos que se esperaría. Ambos siguen un esquema de caja negra, en la que se pierde interpretabilidad en pos de ganar precisión, pero parece que en este caso no compensa. Un detalle a tener en cuenta

Modelo	acc_{in}	acc_{test}	AUC_{in}	AUC_{test}
Regresión Logística	67.74	65.55	74.32	70.92
Random Forest	**99.80**	**66.71**	**99.99**	72.60
Gradient Boosting	71.21	66.44	78.61	**72.84**
MLP	66.04	64.83	71.77	70.25
KNN	—	63.95	—	68.80
RBF-Network	65.93	64.83	71.40	70.14
Aleatorio	50.43	50.58	50.08	49.78

Tabla 4: Resultados de los mejores modelos. Se destacan en negrita los mejores valores de cada columna.

además es que en RBF calculamos el *diámetro* de los datos, para lo que necesitamos almacenar temporalmente en memoria la matriz de distancias, cuyo tamaño puede ser bastante grande debido a la gran cantidad de ejemplos que tenemos.

Por su parte, el modelo lineal ha tenido un desempeño sorprendente, siendo de los más sencillos pero resultando competitivo con otros modelos mucho más complejos, como son los vencedores Gradient Boosting y Random Forest. Estos modelos sí que tienen una mayor facilidad de interpretación (podríamos imprimir los árboles de decisión usados internamente), e incluso nos proporcionan si quisiésemos una medida de la importancia de las características usadas en la predicción. Sin embargo, debemos tener en cuenta que al agregar muchos árboles juntos podemos perder ese poder interpretativo o verlo reducido en gran medida, aunque podríamos sacar reglas a partir de los modelos ajustados para recuperar las interpretaciones.

En cuanto al tiempo de entrenamiento, KNN, Regresión Logística y RBF-Network son los que menos tardan (en el primer caso es casi instantáneo, todo el tiempo se consume después en la predicción). Los otros tres modelos tienen un gran tiempo de entrenamiento, incluso habiendo restringido el número de capas ocultas en MLP, por lo que no es muy viable emplear un espacio de búsqueda demasiado extenso.

Un último detalle a comentar es que en los modelos lineales **hemos repetido el procedimiento sin hacer selección de variables**, obteniendo unos resultados peores (65.26 % de *accuracy* en *test*) y además elevando el número de variables a 1769. Por tanto, podemos concluir que el uso de la técnica de PCA está justificado, no solo para reducir el tiempo de ajuste, sino también para mejorar la calidad del mismo.

10. Conclusiones y estimación del error

El mejor modelo que nos indica la métrica *accuracy* (y con el que nos quedamos como modelo final) es `RandomForest`, con una tasa de acierto del 66.71 % en *test*, a parte de un *accuracy* de casi el 100 % en entrenamiento. Sin embargo, merece la pena comentar que `GradientBoostingClassifier` se acerca mucho con un acierto del 66.44 % e incluso bajo la métrica *AUC* es mejor (72.84 % frente al 72.60 % de RF).

De hecho todos los modelos han obtenido resultados en acc_{test} muy parecidos, con diferencias de no más de 3 unidades y con valores por encima del 50 % base (aleatorio) por al menos más de 10 puntos, por lo que en general los modelos han obtenido unas soluciones con un rendimiento muy parecido y que podemos decir que son “buenas”, en el sentido que son mejores

significativamente que usar el azar.

Analizando también la métrica secundaria, vemos que el orden de los valores de AUC_{test} van muy ligados con acc_{test} (no muy sorprendente debido al casi balanceo de las clases), dándonos mayor fiabilidad en los resultados y también nos permiten dar más información sobre la bondad de los modelos, ofreciendo otro punto de vista para compararlos en cuanto a su poder de discriminación.

Notemos además que el mejor modelo lineal `LogisticRegression` ha conseguido un rendimiento muy bueno, más que otros modelos mucho más complejos como `RBF` o `MLP`. Esto último nos indica que a pesar de la "simplicidad" de un modelo lineal se puede obtener resultados muy buenos, dependiendo del problema que estemos tratando. Esto refuerza la idea de que en primer lugar lo mejor es probar siempre con un modelo lineal, ya que puede darnos un resultado suficientemente bueno sin tener que usar otros algoritmos mucho más complejos.

Podemos estudiar la buena calidad del ajuste `RandomForest` obtenido mediante su matriz de confusión en Figura 24. En ella observamos cómo la tasa de falsos positivos es del 19 %, mientras que la de falsos negativos es algo menor, del 14 %. Si bien no son valores excesivamente bajos, dentro de lo que cabe acertamos las etiquetas en una proporción razonable de ejemplos.

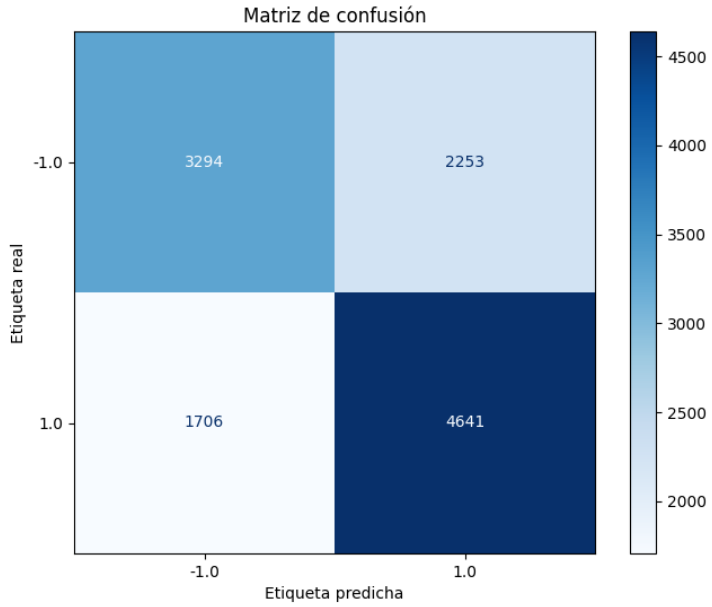

Figura 24: Matriz de confusión de RandomForest.

También mostramos la curva ROC de Random Forest en Figura 25, donde añadimos KNN para ejemplificar la comparación con el peor modelo, observando que lo supera para todos los posibles umbrales.

Por completitud podemos intentar establecer una cota para el error de generalización de nuestro modelo final. Como hemos usado el conjunto de *test* para discriminar y elegir el mejor modelo, podemos aplicar la *cota de Hoeffding* tomando como tamaño de la clase 6, el número de modelos finales comparados. Sabemos que la cota en general se expresa como

$$E_{out} \leq E_{test} + \sqrt{\frac{1}{2N_{test}} \log \frac{2|\mathscr{H}|}{\delta}}, \quad \text{con probabilidad } \geq 1 - \delta.$$

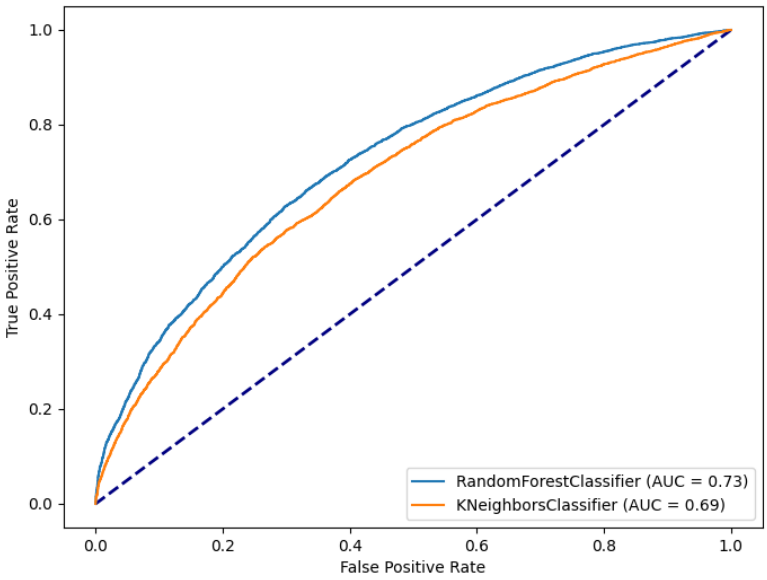

Figura 25: Curva ROC con RandomForest y KNN.

Aplicándolo con nuestros datos y un valor de $\delta = 0.05$, obtenemos que

$$E_{out} \leq 0.3329 + \sqrt{\frac{1}{2 \cdot 11893} \log\left(\frac{2 \cdot 6}{0.05}\right)} \leq 0.3481,$$

es decir, con una confianza del 95 % el error de generalización de nuestro modelo será más pequeño que 34.81 %, o lo que es lo mismo, el *accuracy* de generalización será superior al 65.2 %.

Si comparamos con los resultados obtenidos por los autores del *dataset* en (Fernandes et al., 2015) con diversas técnicas, vemos que conseguimos alcanzar el mejor resultado base conocido: con una precisión de dos cifras decimales, nuestro modelo ganador alcanza un *accuracy* de 0.67 y un valor de AUC de 0.73, el mismo que reportan en el artículo. Podemos concluir entonces que, en base a la técnica de selección seguida y a la luz de los resultados obtenidos, hemos conseguido el mejor modelo dentro de las clases fijadas, con un error de ajuste prácticamente nulo y una buena capacidad de generalización a pesar del sobreajuste; en general, estamos satisfechos con el modelo final en términos de rendimiento.

Anexo: Funcionamiento del código

Vamos a comentar brevemente cómo está estructurado el código. En primer lugar, hay una función `main` que controla la ejecución del programa. Se sirve de unas constantes globales fijadas al inicio del archivo para ver el camino a seguir. En primer lugar se cargan los datos con la función `read_data`, y se realizan las divisiones oportunas en la función `split_data`. Después se muestran, si están activadas las imágenes, una serie de estadísticas y visualizaciones de los datos. Posteriormente se pasa al procedimiento de ajuste y selección de modelos, que puede hacerse de dos formas:

1. Si queremos ejecutar todo el procedimiento de selección de modelos, activamos el valor `DO_MODEL_SELECTION = True`, lo que pasará el control a la función `fit_model_selection`, que se encarga de realizar este proceso. Si queremos ver más detalles aún podemos activar el parámetro `SHOW_ANALYSIS = True`, que realizará el proceso completo de preanálisis y nos mostrará las conclusiones.
2. Si queremos omitir el procedimiento de selección de modelos y simplemente pasar al entrenamiento de los mejores modelos, basta dejar `DO_MODEL_SELECTION = False` (su valor por defecto). Los parámetros con los que se entrenan los modelos son los que se encontraron en el paso 1.

Finalmente mediante la función `compare` se evalúan los modelos entrenados y se imprimen los resultados, terminando el programa y mostrando los tiempos de ejecución.

Cabe destacar que mediante la función `preprocess_pipeline` obtenemos el cauce de preprocesado apropiado para cada modelo, controlado mediante los tipos enumerados `Model` y `Selection`. Otro detalle de implementación es la clase `multi_randint`, que permite escoger un entero aleatorio simbolizado como una tupla de tamaño arbitrario, útil por ejemplo para establecer el espacio de parámetros de los modelos MLP (en concreto, el número de neuronas por capa).

Por último, recogemos aquí la implementación (omitiendo los comentarios) de la clase `RBFNetworkClassifier`, para ilustrar cómo podemos implementar un clasificador propio e integrarlo dentro de `sklearn`, de forma que se puede utilizar en *pipelines* y en el método de `GridSearchCV`, por ejemplo.

```
class RBFNetworkClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, k = 7, alpha = 1.0, batch_size = 100,
                 random_state = None):
        self.k = k
        self.alpha = alpha
        self.batch_size = batch_size
        self.random_state = random_state
        self.centers = None
        self.r = None

    def _choose_centers(self, X):
        init_size =
            3 * self.k if 3 * self.batch_size <= self.k  else None
        kmeans = MiniBatchKMeans(
```

```
            n_clusters = self.k,
            batch_size = self.batch_size,
            init_size = init_size,
            random_state = self.random_state)
        kmeans.fit(X)
        self.centers = kmeans.cluster_centers_

    def _choose_radius(self, X):
        R = np.max(euclidean_distances(X, X))
        self.r = R / (self.k ** (1 / self.n_features_in_))

    def _transform_rbf(self, X):
        return rbf_kernel(X, self.centers, 1 / (2 * self.r ** 2))

    def fit(self, X, y):
        self.model = RidgeClassifier(
            alpha = self.alpha,
            random_state = self.random_state)

        self.classes_ = unique_labels(y)
        self.n_features_in_ = X.shape[1]

        self._choose_centers(X)
        self._choose_radius(X)
        Z = self._transform_rbf(X)

        self.model.fit(Z, y)

        self.intercept_ = self.model.intercept_
        self.coef_ = self.model.coef_

        return self

    def score(self, X, y = None):
        Z = self._transform_rbf(X)
        return self.model.score(Z, y)

    def predict(self, X):
        Z = self._transform_rbf(X)
        return self.model.predict(Z)

    def decision_function(self, X):
        Z = self._transform_rbf(X)
        return self.model.decision_function(Z)
```

Bibliografía

Abu-Mostafa, M.-I., Y. S. (2012). *Learning from data: A short course*.

Fernandes, K., Vinagre, P., & Cortez, P. (2015). A proactive intelligent decision support system for predicting the popularity of online news. *Proceedings of the 17th portuguese conference on artificial intelligence*. Coimbra, Portugal.

Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statistics & Data Analysis*, *38*(4), 367–378.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Sklearn api reference. (2020). Retrieved from https://scikit-learn.org/stable/modules/classes.html