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 %
$\geq$ 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.

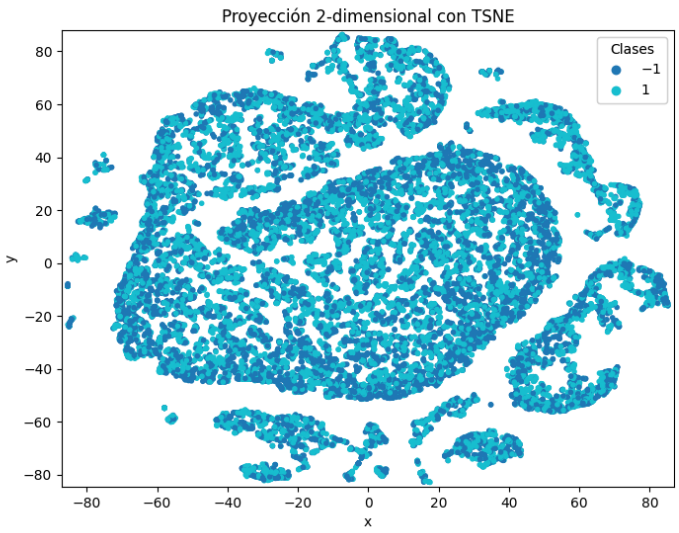

Figura 1: Proyección en 2 dimensiones con TSNE.

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

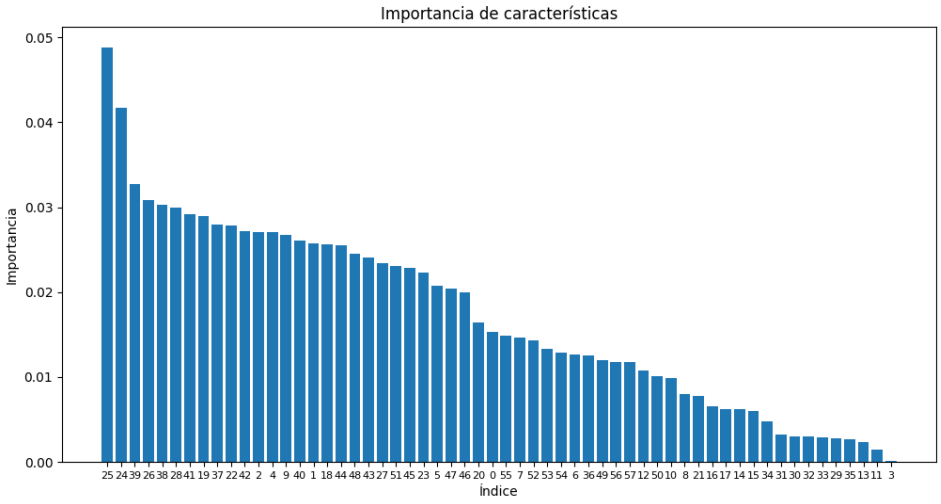

Figura 2: Importancia de características según criterio de Random Forest.

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

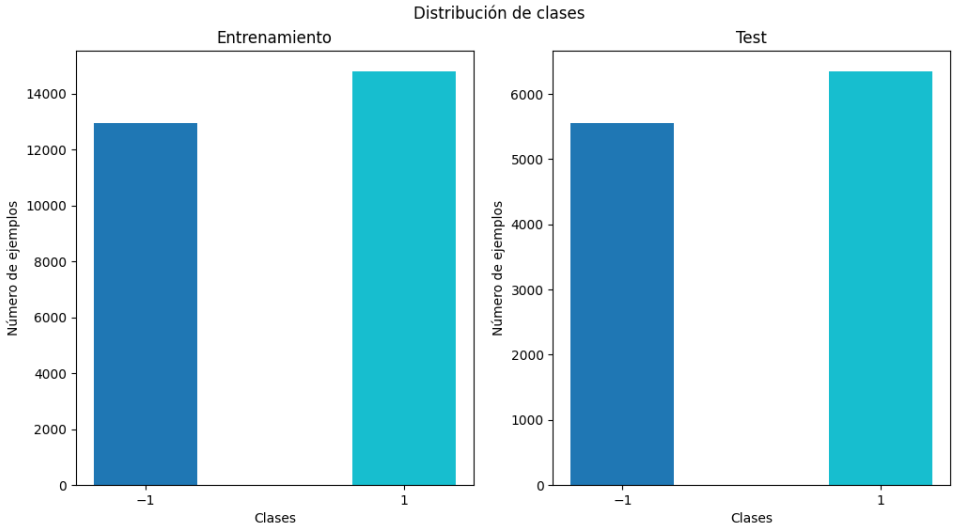

Figura 3: Distribución de clases en entrenamiento y *test*.

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.

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.

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

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

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.

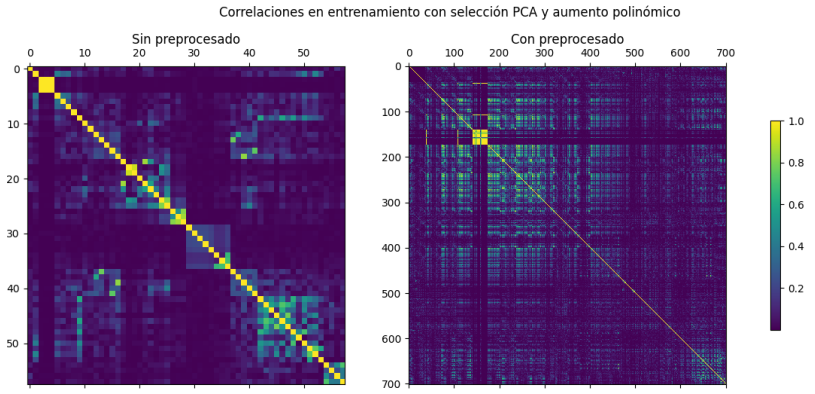

Figura 4: Matriz de correlación en entrenamiento antes y después del preprocesado.

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}.

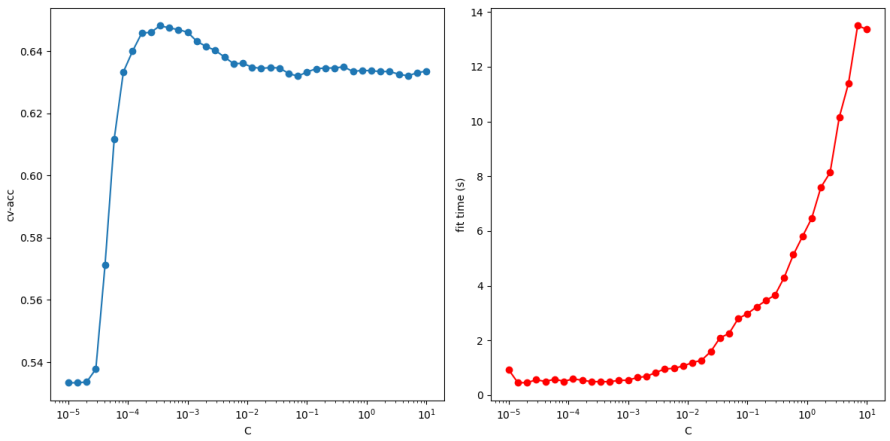

Figura 5: acc-cv/tiempo según C en LogisticRegression.

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 $[0, 5]$.

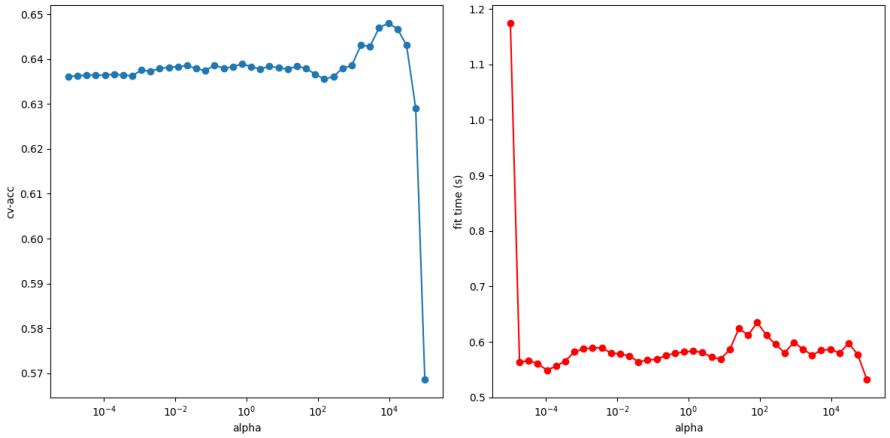

Figura 6: acc-cv/tiempo según alpha en RidgeClassifier.

```
{"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)}
```

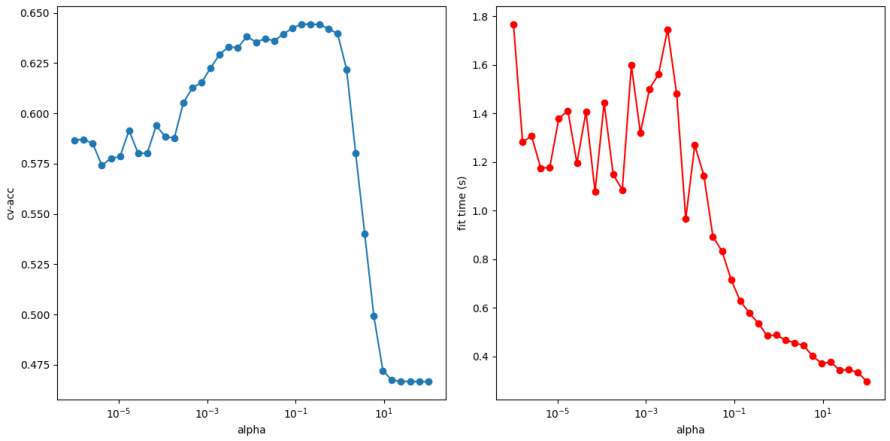

Figura 7: acc-cv/tiempo según alpha en SGDClassifier.

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.

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:

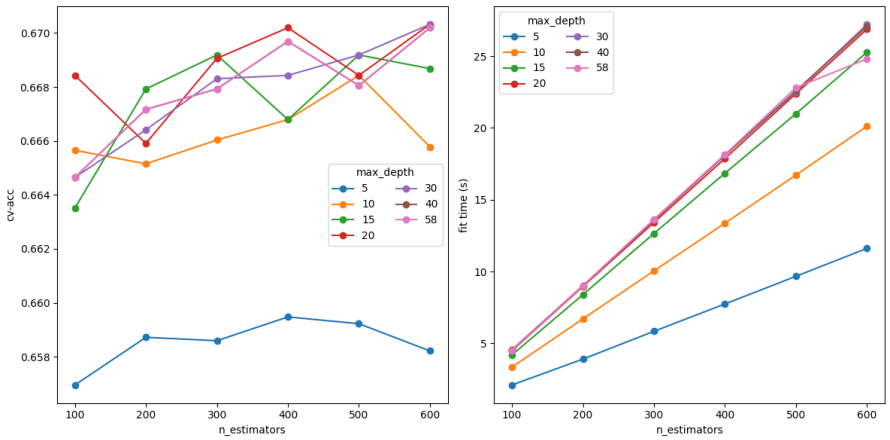

Figura 8: acc-cv/tiempo según n_estimators y max_depth en RandomForest.

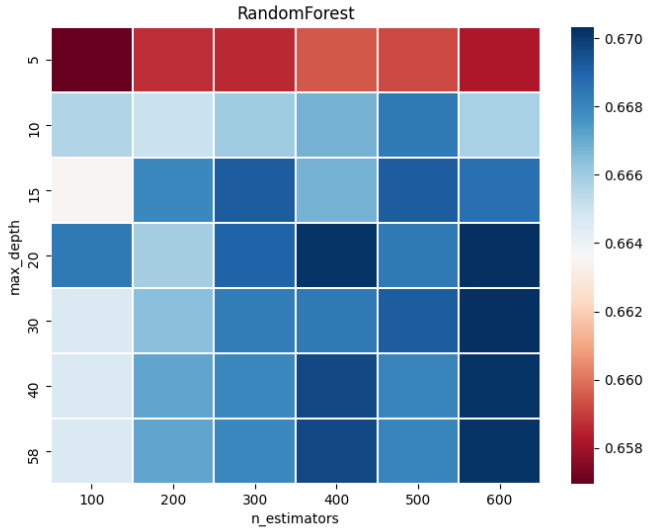

Figura 9: Mapa de calor según n_estimators y max_depth en RandomForest.

```
{"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

AdaBoost

Usamos AdaBoost con el objeto `AdaBoostClassifier`, fijando la tasa de aprendizaje por defecto a 1 y usando como clasificador *flojo* un árbol de decisión `DecisionTreeClassifier`. Los hiperparámetros que dejamos para buscar son: `max_depth` la profundidad máxima de los clasificadores, y `n_estimators` el nº de los 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]}
```

La base de AdaBoost es formar un buen clasificador entrenando muchos clasificadores *flojos* (*boosting*), por ejemplo arboles de decisión con una regla, repetídamente con muchas modificaciones de los datos (aplicando distintos pesos a los datos); de manera que la predicción de las etiquetas se hace con el voto mayoritario de todos los clasificadores. Además la función de error que intenta minimizar es la función exponencial:

$$L(y, f(x)) = \exp(-yf(x))$$

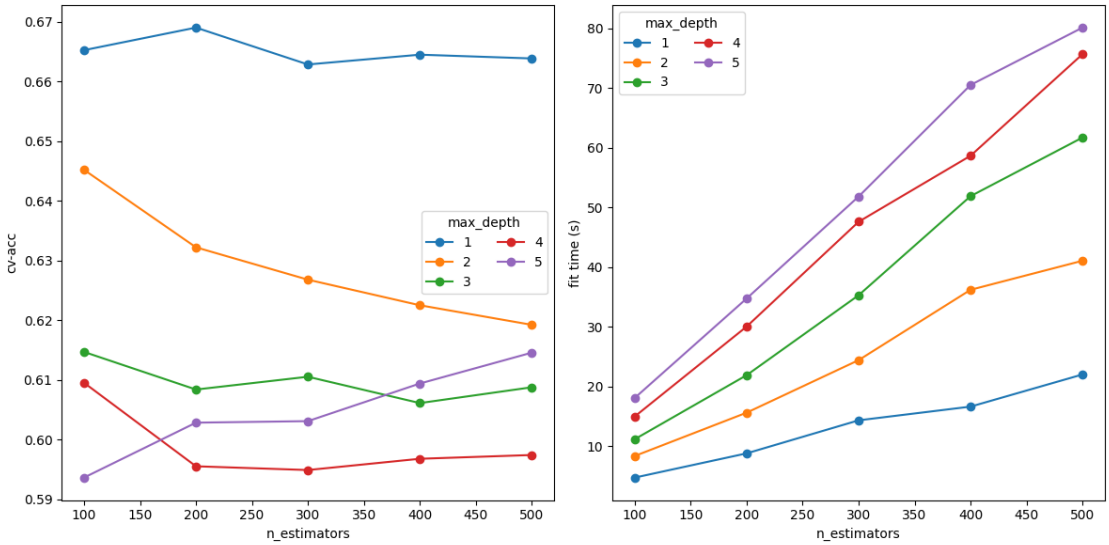

Figura 10: acc-cv/tiempo según n_estimators y max_depth en AdaBoost.

Los resultados del preanálisis 10 y 11 nos dejan claro que los mejores resultados se benefician de árboles lo más simple posibles (una regla) y dentro de este, el mejor valor se alcanza con 200 árboles. Por tanto dejamos dejamos 1 como profunidad máxima y variamos el nº de árboles entorno a 200:

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

También probamos añadiendo `learning_rate` como hiperparámetro para probar con otra tasa más pequeña.

Gradient Boosting

También usamos GradientTreeBoosting con el objeto `GradientBoostingClassifier` fijando la función de perdida a `'deviance'` (regresión logística), la tasa de aprendizaje a 0.1 y usando para entrenar todos los datos. Los hiperparámetros que variamos son la profundidad del los árboles `max_depth` y el nº de árboles `n_estimators`:

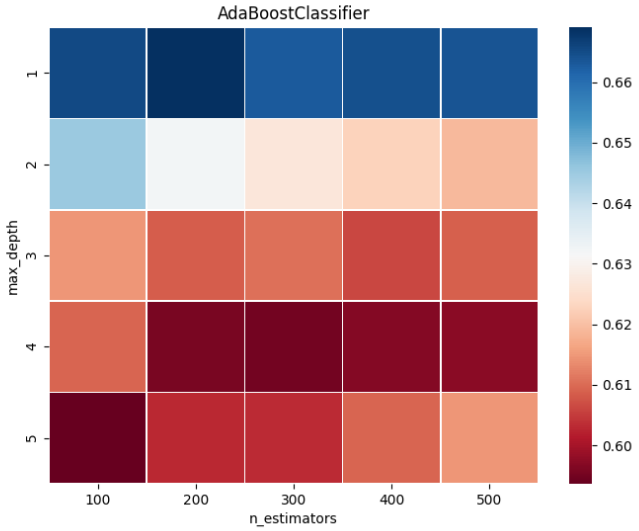

Figura 11: Mapa de calor para acc-cv según n_estimators y max_depth en AdaBoost.

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

GradientTreeBoosting surje como una generalización del método de boosting usando árboles como clasificadores, que permite utilizar distintas funciones de error para el entrenamiento. En cualquier caso la idea básica es la misma que AdaBoost, en concreto cambiamos la función de ajuste a la de regresión logística:

$$L(y, f(x)) = \log(1 + \exp(yf(x)))$$

En los resultados del preanálisis 12 y 13 vemos 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 cualquier caso 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]}
```

Hemos añadido los hiperparámetros `learning_rate`, la tasa de aprendizaje 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.

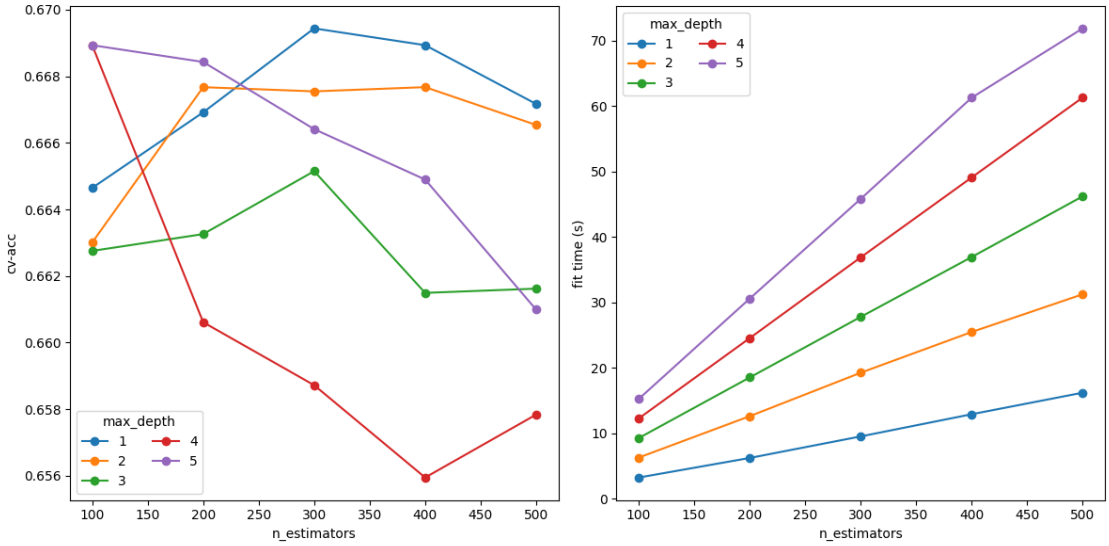

Figura 12: acc-cv/tiempo según n_estimators y max_depth en GradientBoosting.

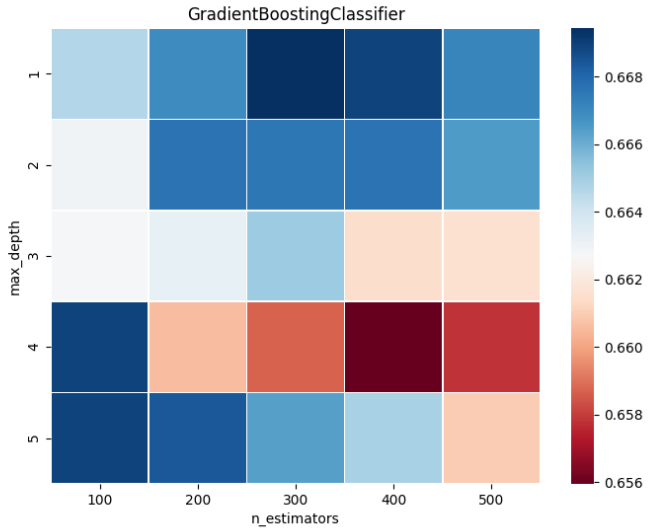

Figura 13: Mapa de calor según n_estimators y max_depth en GradientBoosting.

8.4. Perceptrón multicapa (MLP)

Perceptrón multicapa con 3 capas (2 ocultas y la de salida) con el objeto `MLPClassifier` con los siguientes parámetros fijados: `learning_rate_init = 0.01` (tasa de aprendizaje inicial), `solver = sgd` (ajuste con SGD), `max_iter = 300` (nº máximo de iteraciones), `learning_rate = 'adaptive'` (decrementa la tasa de aprendizaje cuando no baja el error), `activation = relu` (función de activación ReLU) y `tol = 1e-3` (tolerancia para la convergencia).

Dejamos como hiperparámetro el nº de neuronas en las capas ocultas `hidden_layer_sizes`, tomando el mismo tamaño para ambas capas y haciendo una búsqueda aleatoria en el rango $[50, 101]$:

```
{"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)}
```

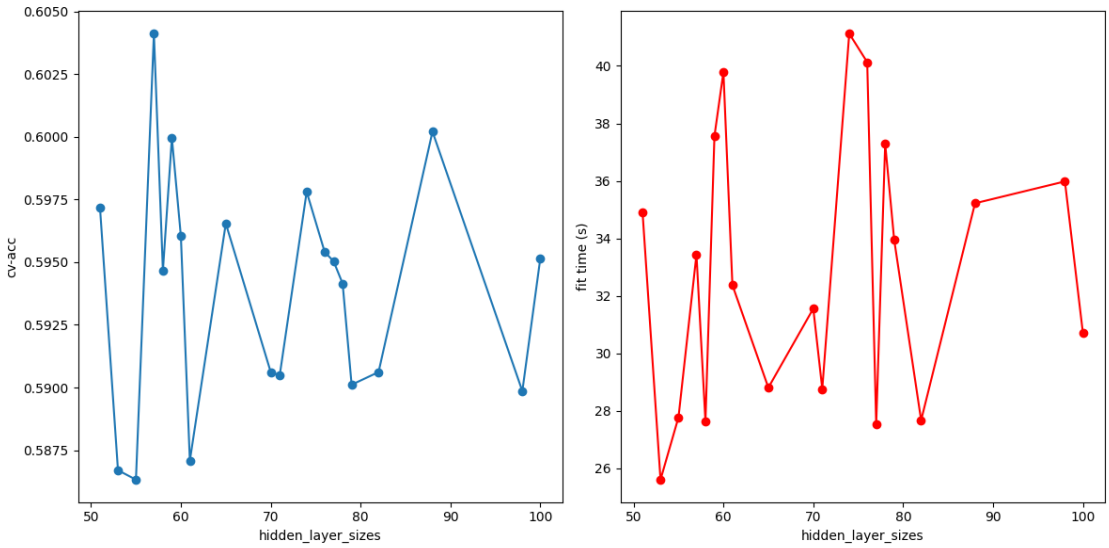

Figura 14: acc-cv/tiempo según hidden_layer_sizes en MLP.

El resultado de 14 nos deja dos tamaños con los mejores resultados en sus entornos, en 57 y 88, por tanto probamos con estas configuraciones:

```
{"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)}
```

Además añadimos el hiperparámetro `alpha` relativo a la regularización L2 para que el modelo generalice mejor.

8.5. K-Nearest Neighbors (KNN)

Algoritmo de los k vecinos más cercanos mediante el objeto `KNeighborsClassifier`, usando la métrica euclidea, y el espacio de búsqueda para el hiperparámetro `k`, considerando

la regla a ojo que recomienda usar $k = \sqrt{N}$, vamos desde 1 hasta 200 pasando por $k \approx 90$ (tamaño dataset preanálisis):

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

K-nn considera los k vecinos más cercanos al punto que se quiera etiquetar y se obtiene la etiqueta en función de las etiquetas de estos vecinos (moda, por ejemplo), por lo que las funciones que ajustamos son las que particionan el espacio de cualquier manera.

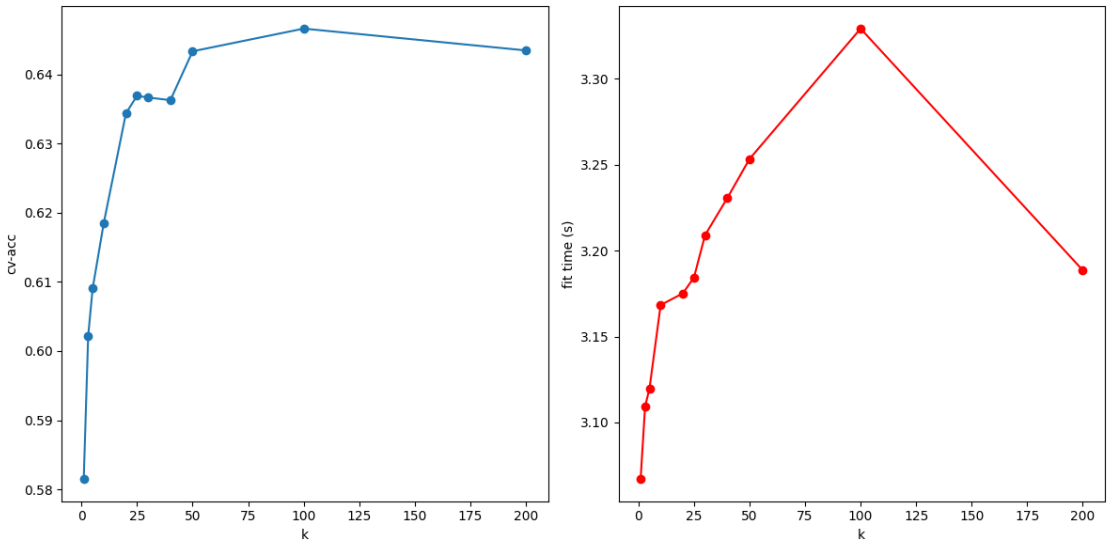

Figura 15: acc-cv/tiempo según k en KNN.

Los resultados preanálisis 15 nos confirman la regla experimental, ya que el óptimo está en 100; por tanto para el conjunto training usamos un entorno de $k \approx 141$, quedando el espacio de búsqueda como lo siguiente:

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

Además añadimos para dar más variabilidad en la búsqueda, el hiperparámetro `weights` que permite cambiar el peso de los k-vecinos encontrados: `uniform` todos importan por igual, `distance` los vecinos importan inversamente proporcional a la distancia.

8.6. Redes de funciones de Base Radial (RBF)

El clasificador está implementado en la clase `RBFNetworkClassifier` siguiendo el algoritmo 1 en REFERENCIA_LIBRO. Hemos considerado el algoritmo K-medias (`KMeans`) para encontrar los k centroides, la sugerencia de fijar $r = \frac{R}{k^{1/d}}$ y finalmente el algoritmo lineal considerado con 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 clasificador.

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

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

Los hiperparámetros a considerar son tanto el nº 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]}
```

Una implementacion más específica se puede encontrar en Anexo: Funcionamiento del código.

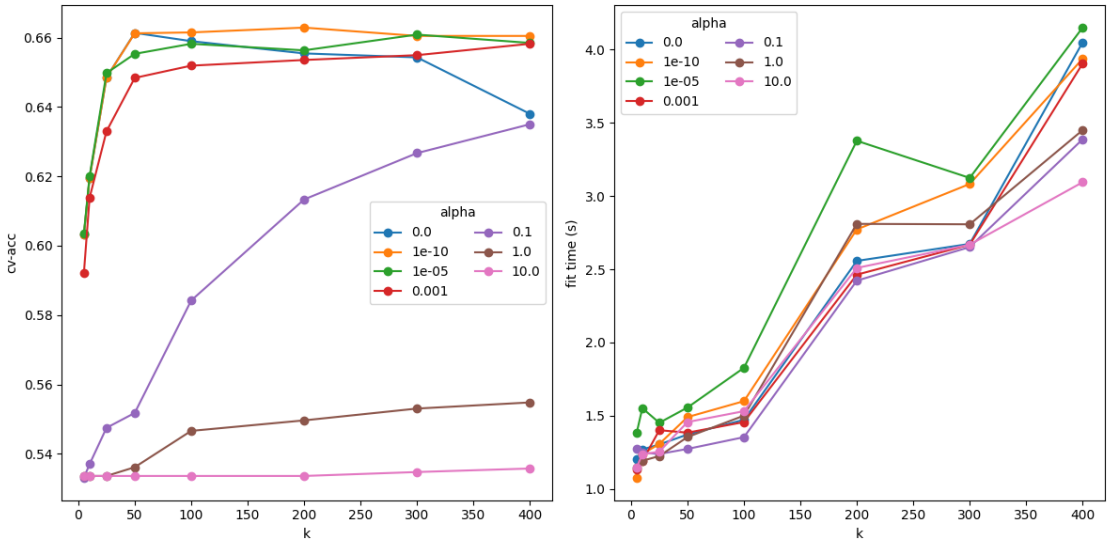

Figura 16: acc-cv/tiempo según k y alpha en RBF.

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

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

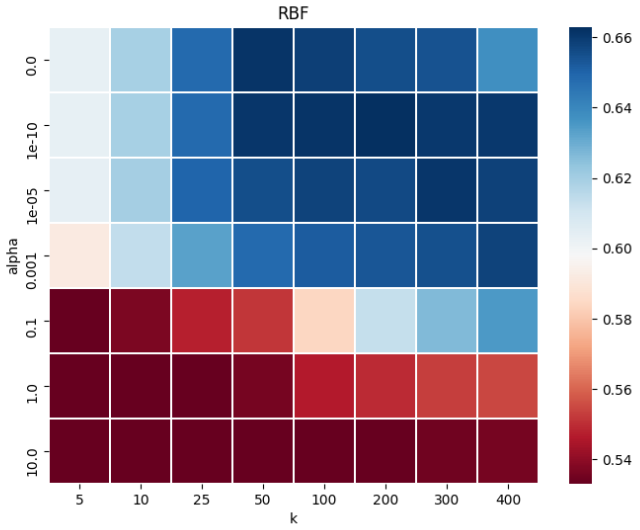

Figura 17: Mapa de calor según k y alpha en RBF.

Aleatorio

Incorporamos el 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 debería obtener un acierto por encima de este para considerarlo bueno.

9. Análisis de resultados

Las mejores configuraciones de cada modelo estan en la tabla 3, cuyos resultados de las métricas acc y AUC en training/test se recojen en la tabla 4.

Modelo	Hiperparámetros
Lineal	LogisticRegression, C = 0.1
RandomForest	max_depth = 20, n_estimators = 600, ccp_alpha = 0
Boosting	GradientBoostingClassifier, 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
Aleatorio	—

Tabla 3: Mejores configuraciones

Veamos las curvas de aprendizaje de cada modelo para ver si hay underfitting, overfitting o hay un ajuste más o menos bueno:

- `LogisticRegression`: 18
- `RandomForest`: 19

Modelo	acc_{in}	acc_{test}	AUC_{in}	AUC_{test}
Lineal	67.74	65.55	74.32	70.92
RandomForest	**99.80**	**66.71**	**99.99**	72.60
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 cada modelo

- `GradientBoostingClassifier`: 20
- `MLP`: 21
- `KNN`: 22
- `RBF-Network`: 23

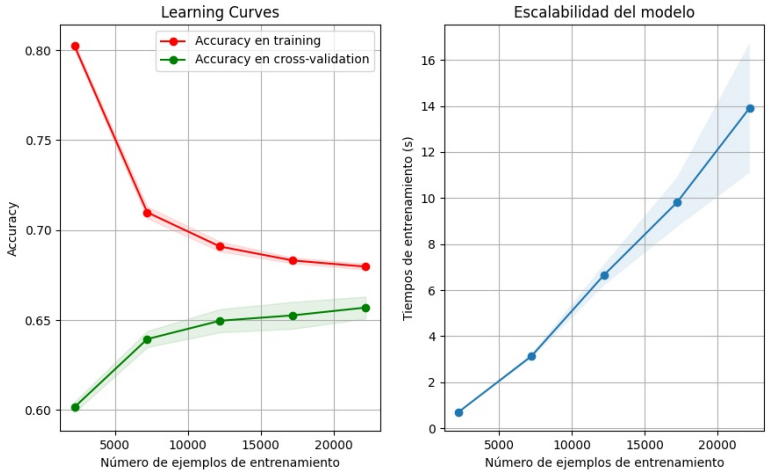

Figura 18: Curvas de aprendizaje para LogisticRegression.

En general, exceptuando KNN y RandomForest que sabemos que tienden mucho al sobreajuste siempre (KNN ya que tiene el propio punto en el training, y RandomForest por el bajísimo sesgo por construcción de árboles), los modelos empiezan con un gran sobreajuste que va disminuyendo conforme el nº de datos aumenta (las curvas se van acercando) y que acaban por casi juntarse con todos los datos.

Esto nos indica que los modelos consiguen un buen ajuste con poco overfitting, sacando casi todo el partido de los datos que tenemos. Observamos también que las curvas de validación se estabilizan, indicándonos que probablemente con muchos más datos no vamos a obtener ventajas notables por lo que si queremos aumentar mucho más la métrica deberemos considerar, por ejemplo, recolectar más características.

- Ventajas/inconvenientes modelos

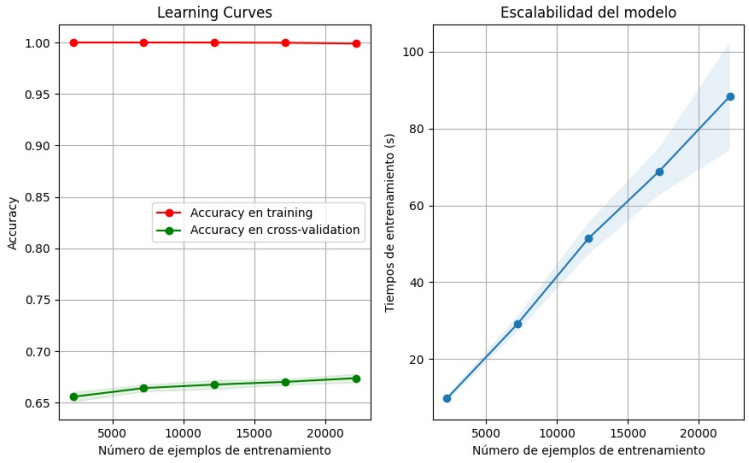

Figura 19: Curvas de aprendizaje para RandomForest.

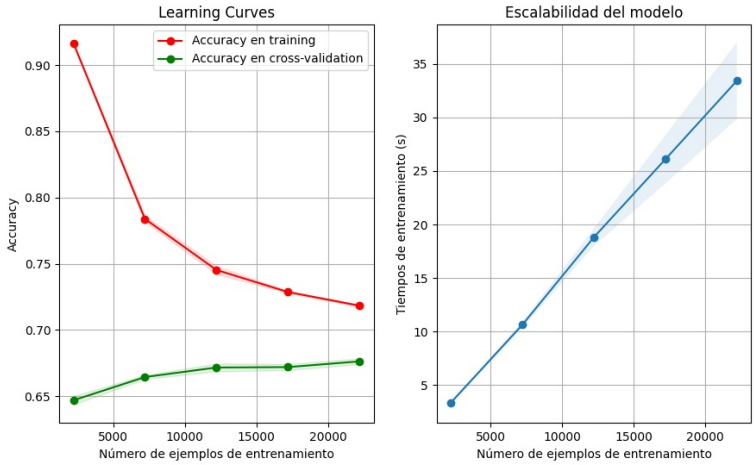

Figura 20: Curvas de aprendizaje para GradientBoostingClassifier.

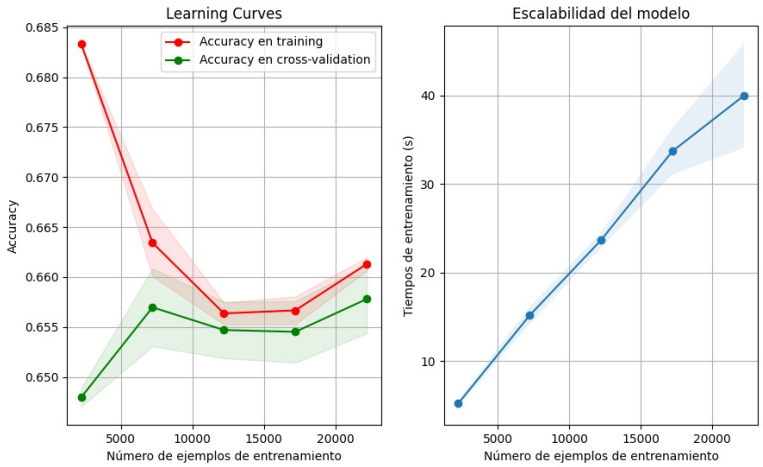

Figura 21: Curvas de aprendizaje para MLPClassifier.

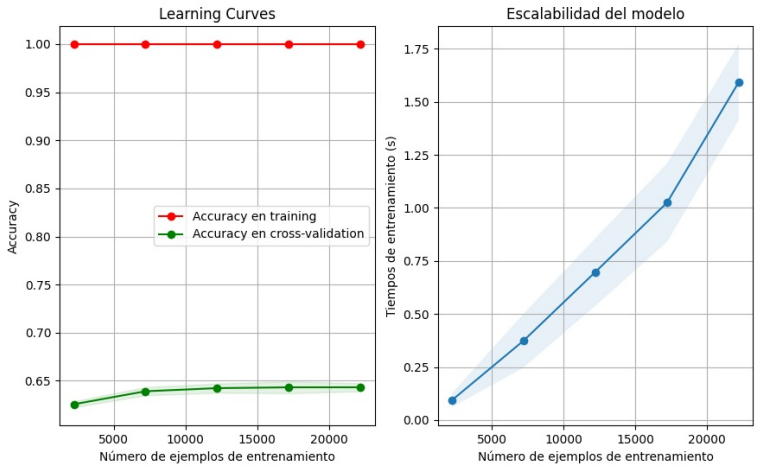

Figura 22: Curvas de aprendizaje para KNeighborsClassifier.

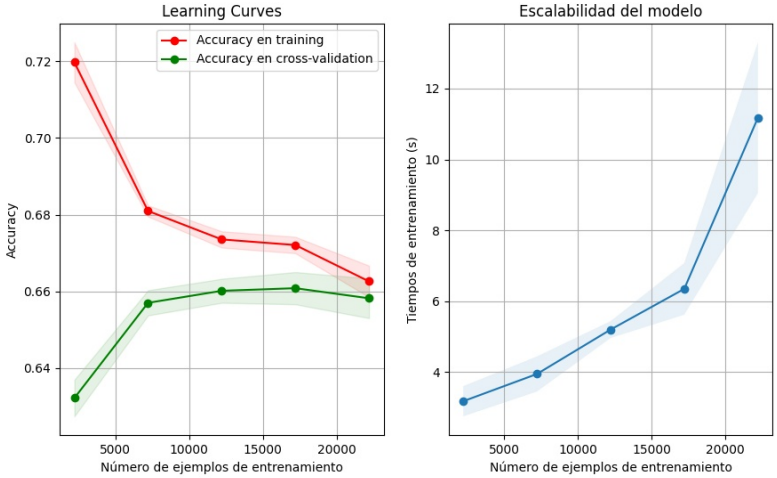

Figura 23: Curvas de aprendizaje para RBFNetworkClassifier.

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 %. 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.

Notemos además que el mejor modelo lineal `LogisticRegression` ha conseguido un rendimiento muy bueno, más que otros modelos mucho más complejos como `KNN` o `MLP`. Esto último nos indica que a pesar de la "simplicidad" de un modelo lineal se puede obtener resultados iguales o mejores, que dependerá del problema que estemos tratando pero sin lugar a dudas

no hay que descartar intentar ajustar un modelo lineal ya que puede darnos un buen resultado sin tener que usar otros algoritmos mucho más complejos.

Analizando también la metrica secundaria, vemos que el orden de los valores de AUC_{test} van muy ligados con acc_{test} (esperable 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, permitiendo otro punto de vista para compararlos.

Finalmente veamos la buena calidad de `RandomForest` mediante su matriz de confusión en Figura 24 y la curva ROC en Figura 25.

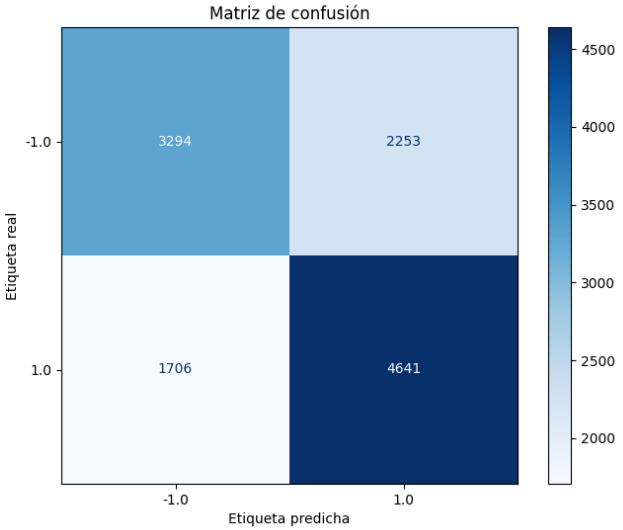

Figura 24: Matriz de confusión de RandomForest.

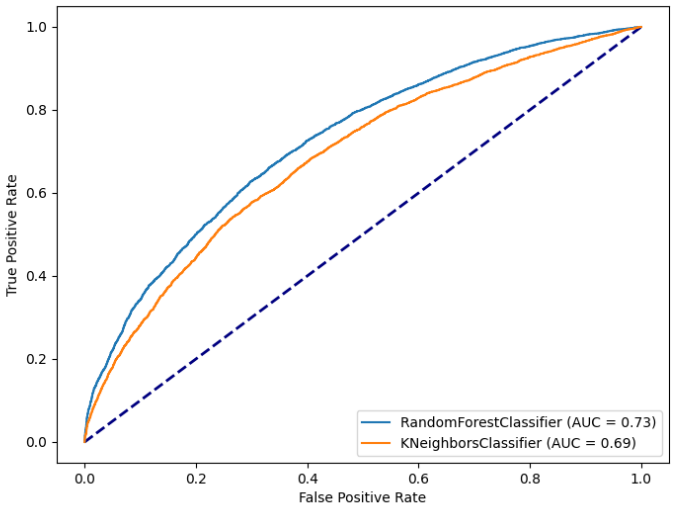

Figura 25: Curva ROC con RandomForest.

- Repaso por encima de todos.

(Fernandes et al., 2015)

Anexo: Funcionamiento del código

```
class RBFNetworkClassifier(BaseEstimator, ClassifierMixin):
    """Implementación de un clasificador de red de funciones (gaussianas)
          de base radial.
       Internamente utiliza un clasificador lineal RidgeClassifier para ajustar
       los pesos del modelo final."""

    def __init__(self, k = 7, alpha = 1.0, batch_size = 100,
                 random_state = None):
        """Construye un clasificador con los parámetros necesarios:
             - k: número de centros a elegir.
             - alpha: valor de la constante regularización.
             - batch_size: tamaño del batch para el clustering no supervisado.
             - random_state: semilla aleatoria."""

        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):
        """Usando k-means escoge los k centros de los datos."""

        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):
        """Escoge el radio para la transformación radial."""

        # "Diámetro" de los datos
        R = np.max(euclidean_distances(X, X))

        self.r = R / (self.k ** (1 / self.n_features_in_))

    def _transform_rbf(self, X):
        """Transforma los datos usando el kernel RBF."""
```

```
        return rbf_kernel(X, self.centers, 1 / (2 * self.r ** 2))

    def fit(self, X, y):
        """Entrena el modelo."""

        # Establecemos el modelo lineal subyacente
        self.model = RidgeClassifier(
            alpha = self.alpha,
            random_state = self.random_state)

        # Guardamos las clases y las características vistas
        #durante el entrenamiento
        self.classes_ = unique_labels(y)
        self.n_features_in_ = X.shape[1]

        # Obtenemos los k centros usando k-means
        self._choose_centers(X)

        # Elegimos el radio para el kernel RBF
        self._choose_radius(X)

        # Transformamos los datos usando kernel RBF respecto de los centros
        Z = self._transform_rbf(X)

        # Entrenamos el modelo lineal resultante
        self.model.fit(Z, y)

        # Guardamos los coeficientes obtenidos
        self.intercept_ = self.model.intercept_
        self.coef_ = self.model.coef_

        return self

    def score(self, X, y = None):
        # Transformamos datos con kernel RBF
        Z = self._transform_rbf(X)

        # Score del modelo lineal
        return self.model.score(Z, y)
```

Bibliografía

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.