Proyecto Final: Online News Popularity

Aprendizaje Automático

Miguel Lentisco Ballesteros Antonio Coín Castro

Curso 2019-20

Índice

1. Introducción

En esta práctica perseguimos ajustar el mejor modelo lineal en dos conjuntos de datos dados, para resolver un problema de clasificación y otro de regresión. En ambos casos seguiremos una estrategia común que nos llevará finalmente a elegir un modelo y estimar su error.

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 una clase de hipótesis (lineal) para resolver el problema.
4. Fijaremos algunos modelos concretos y seleccionaremos el mejor según algún criterio.
5. Estimaremos el error del modelo.

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 regresión.
- En `visualization.py` se recogen todas las funciones de visualización de gráficas, tanto comunes como propias de cada problema.

La ejecución de los dos programas principales 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` de la función principal en cada caso, eliminando todas las gráficas (valor 0) o mostrándolas todas (valor 2). 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

Citar a (Fernandes, Vinagre, & Cortez, 2015) y poner link a la BBDD. Mencionar un resumen muy general de los diferentes atributos que se recogen (tipo real, entero, etc). Mostrar tabla-resumen que viene en el artículo.

2.1. Resumen estadístico de los datos (?)

Estudiar la distribución de clases si hacemos clasificación binaria (número exacto, porcentaje).

3. Selección de la clase de funciones

4. Conjuntos de entrenamiento, validación y *test*

Debido a la gran cantidad de datos que tenemos, se ha hecho una partición (usando `train_test_split`) 20/50/30 % para los conjuntos preanálisis, training y test, de manera que no hay conjunto de validación.

El conjunto de preanálisis se utiliza 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. Después se entrena cada modelo por *K-fold cross validation* usando training y los entornos de las configuraciones óptimas del preanalisis, para obtener la mejor configuración de hiperparámetros por modelo (agrupando todos los modelos lineales, y boosting, respectivamente).

Finalmente cada mejor configuración de cada modelo es evaluada en el conjunto test.

5. Preprocesado de datos

El procesado general que suele ser el más esencial es el de normalizar/estandarizar los datos, de manera que todas las variables estén en el mismo rango (importante para clasificadores que usan distancias); si bien hay que tener en cuenta que esta transformación (y cualquier otra) debe hacerse primero únicamente en el conjunto donde se esté entrenando en el clasificador, y a la hora de evaluar hacer la misma transformación (mismos parámetros) en el conjunto donde se valide.

Por tanto para la fase de preprocesado usaremos los *pipelines* de `sklearn`{Documentación sobre `Pipeline` en `sklearn`.}, que nos permiten aplicar las mismas transformaciones en varias fases de forma cómoda. En ambos casos, consideramos cinco pasos en el preprocesado: tratamiento de valores perdidos, selección de variables, transformaciones polinómicas, umbral de varianza y estandarización.

5.1. Valores perdidos

No tenemos valores pérdidos ni valores que parezcan inconsistentes por lo que saltamos este paso.

5.2. Selección de características

5.3. Transformaciones polinómicas

5.4. Estandarización y umbral de varianza

5.5. Orden de las transformaciones

6. Métricas de error

Ya que las clases están casi 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 $1-E_{in}$ es lo que se conoce como el *accuracy* del modelo. Presentaremos los resultados utilizando esta última descripción ya que parece más ilustrativa.

También consideraremos el área bajo la curva ROC, que nos permite comparar lo bien que lo hace el clasificador frente a un clasificador aleatorio.

TODO: expandir esto?

7. Regularización

El uso de la regularización es esencial para limitar la complejidad del modelo y el *overfitting*, cosa que nos permitirá obtener una mejor capacidad de generalización. Consideramos las siguientes regularizaciones:

- **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.
- **Poda mínima del coste computacional**: se hace poda en las ramas de cada árbol de decisión según la función de coste computacional,
$$R_\lambda(T) = R(T) + \lambda|T|$$
siendo $R(T)$ el ratio mal clasificado de las hojas y $|T|$ el número de nodos hojas.

El valor de $\lambda > 0$ es un hiperparámetro del modelo, que controla la intensidad de la regularización (a mayor valor, más pequeños serán los pesos). Encontrar un valor adecuado es una tarea complicada, pues si es demasiado pequeño seguiremos teniendo sobreajuste, pero si es demasiado grande podríamos caer en el fenómeno opuesto: tener *underfitting* porque el modelo sea poco flexible y no consiga ajustar bien los datos de entrenamiento.

Hemos considerado L2 para los modelos lineales ya que queremos generalizarlos para bajar la varianza en pos de extraer un mayor valor de la métrica, a costa de no rebajar el tiempo de computación (reduciendo variables con L1), pero no nos importa puesto que el tiempo de entrenamiento es razonable y preferimos aumentar el acierto.

En el caso de los árboles de decisión (Random Forest) queríamos bajar la alta varianza que tienen los árboles, consiguiendo también una mejoría del tiempo de entrenamiento.

8. Técnicas y selección de modelos

Pasamos a realizar la selección de hiperparámetros para cada modelo. Hemos considerado usar la técnica de *K-fold cross validation* para escoger la mejor configuración de hiperparámetros de cada modelo. 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.

Para esto utilizamos 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. Para el caso de clasificación, estas divisiones serán en particular instancias de `StratifiedKFold`, que respeta la distribución de clases en las divisiones.

Para todos los modelos, hacemos un preanálisis para estimar un buen espacio de búsqueda de los hiperparámetros numéricos, de manera que no es completamente arbitrario, si no que intentamos restringir a un entorno (holgado, puesto que usaremos menos datos) de la configuración óptima de hiperparámetros que hemos encontrado en este preanálisis. Además, al usar un conjunto menor nos permite explorar un espacio mucho más grande que usando el dataset de entrenamiento entero.

Una vez hemos encontrado la mejor configuración para cada modelo (agrupando todos los modelos lineales, y boosting, respectivamente), se vuelve a entrenar sobre todo el conjunto de entrenamiento, para obtener un mejor rendimiento. Este es el comportamiento por defecto de la función `GridSearchCV`.

Comentamos ahora los modelos que pre-seleccionamos en el problema de clasificación. La métrica usada para decidir el mejor modelo será, de forma natural, el accuracy medio en los conjuntos de validación. Fijamos el número máximo de iteraciones en 1000 para todos los modelos que usen iteraciones.

8.1. Modelos lineales

8.1.1. Regresión Logística

En primer lugar consideramos un modelo de regresión logística, implementado en el objeto `LogisticRegression`, usando regularización L2. 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 preanálisis.

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

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.

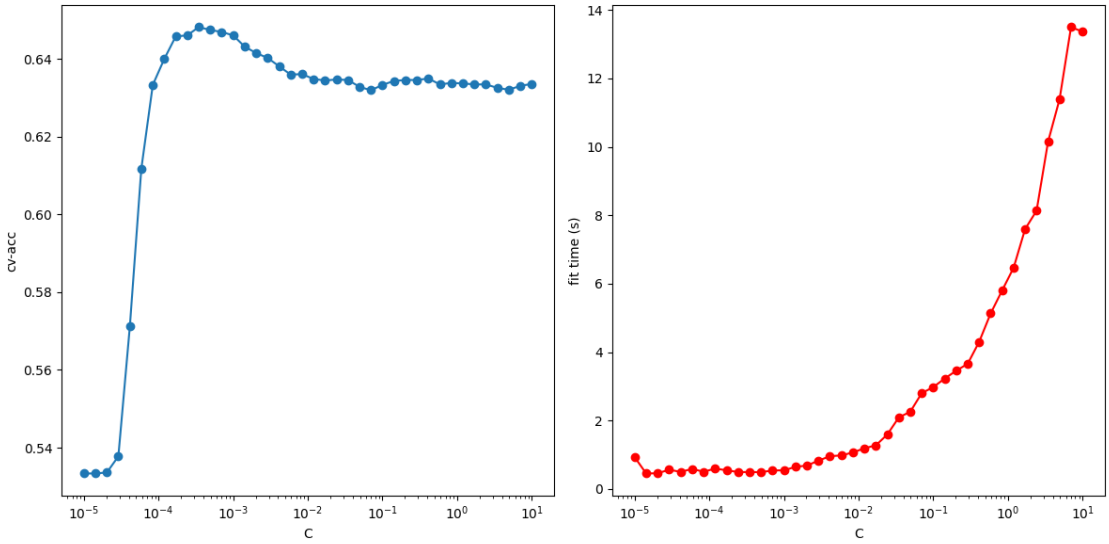

Figura 1: acc-cv/tiempo según C en LogisticRegression.

El resultado del preanálisis de 1 nos indica que desde el orden de 10^{-4} en adelante, los resultados son igual de buenos, alcanzando un máximo entre 10^{-4} y 10^{-3}. Por tanto restringimos C al espacio logarítmico $[-4, 0]$, quedando:

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

8.1.2. Regresión lineal

Consideramos también un modelo de regresión lineal. 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 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.

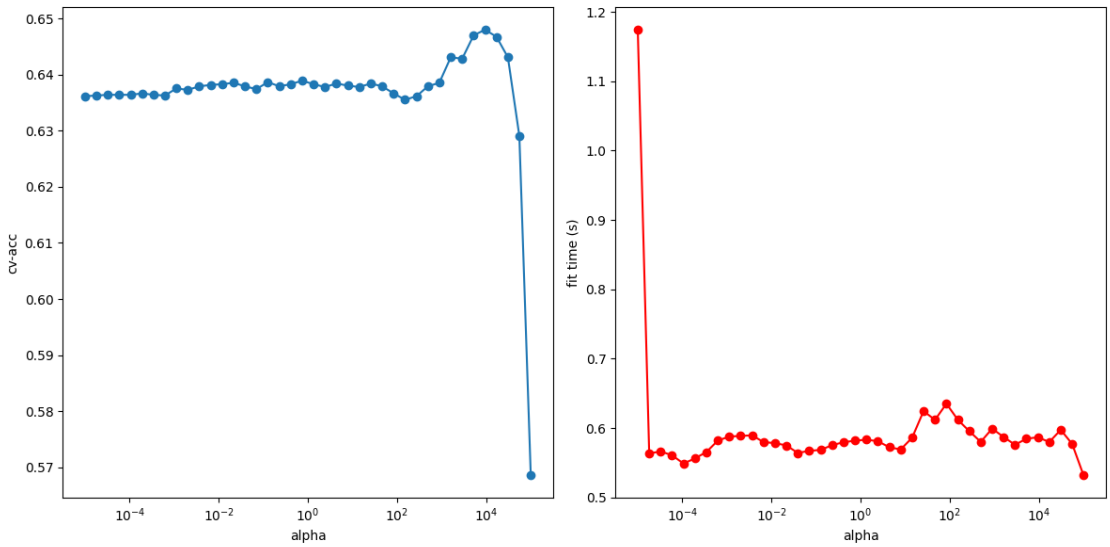

Figura 2: acc-cv/tiempo según alpha en RidgeClassifier.

El resultado del preanálisis de 2 nos arroja un máximo cerca de 10^4, por lo que restringimos el espacio de búsqueda al espacio logarítmico $[0, 5]$:

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

8.1.3. SVM Lineal

Finalmente en modelos lineales, consideramos las máquinas de soporte vectorial (SVM) lineales (sin usar kernel) utilizando el objeto `SGDClassifier` con regularización L2, y tomando alpha con 40 puntos en el espacio logarítmico $[-6, 2]$.

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

La técnica usada es SGD, de manera que minimizamos el error hinge junto con la regularización L2:

$$L_{svm}(w) = \frac{1}{N}\,\text{máx}(0, 1 - y_i f(x_i))$$

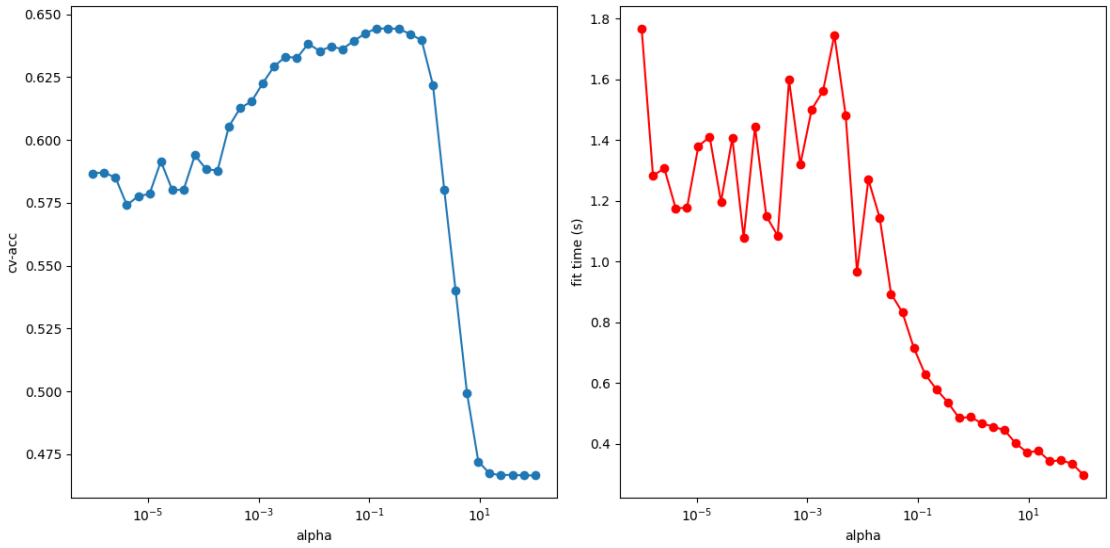

Figura 3: acc-cv/tiempo según alpha en SGDClassifier.

.

Los resultados del preanálisis en 3 nos indican un máximo cerca de 10^{-1} y 1, por lo que restringimos al espacio logarítmico $[-5, 1]$:

```
{"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)}
```

También hemos considerado como hiperparámetro adicional el tipo de tasa de aprendizaje, fijando la inicial (`eta0`) como 0.1: `optimal` (tasa escogida por heurística), `adaptive` (inicial, y decrementa cuando el error no decrementa) y `invscaling` (inicial, y decrementa dividiendo por la raiz del nº de iteraciones).

8.2. Modelos no lineales

8.2.1. Random Forest (RF)

Consideramos primero Random Forest mediante el objeto `RandomForest`, fijando el nº de características de cada arbol a $\sqrt{n_{caract}}$ (usando la regla a ojo) y el criterio gini para decidir las divisiones del árbol. Consideramos como hiperparámetros el nº de árboles `n_estimators` y la profundidad máxima de cada árbol `max_depth`, inicialmente 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]}
```

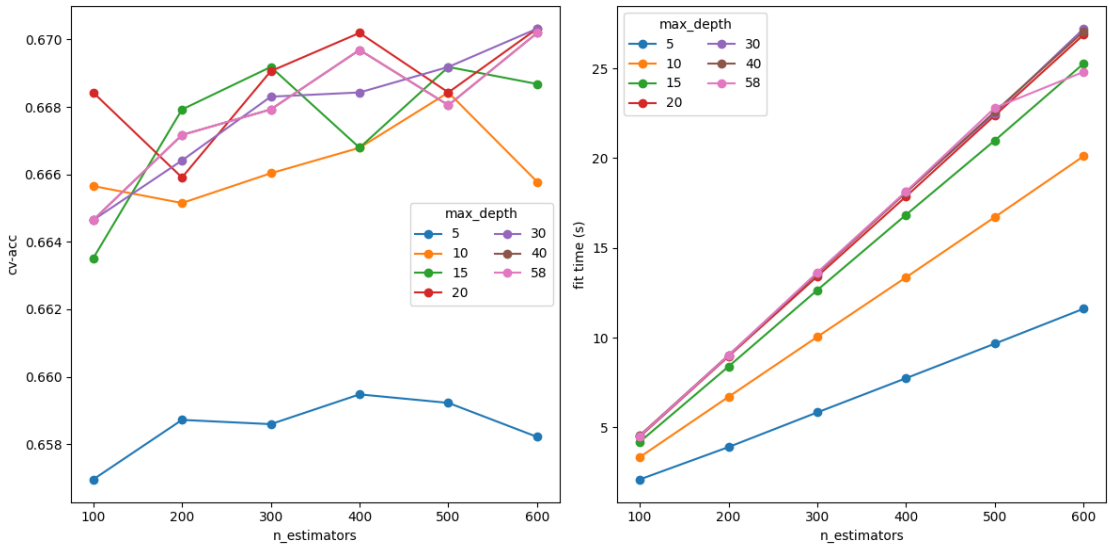

Figura 4: acc-cv/tiempo según hiperparámetros en RandomForest.

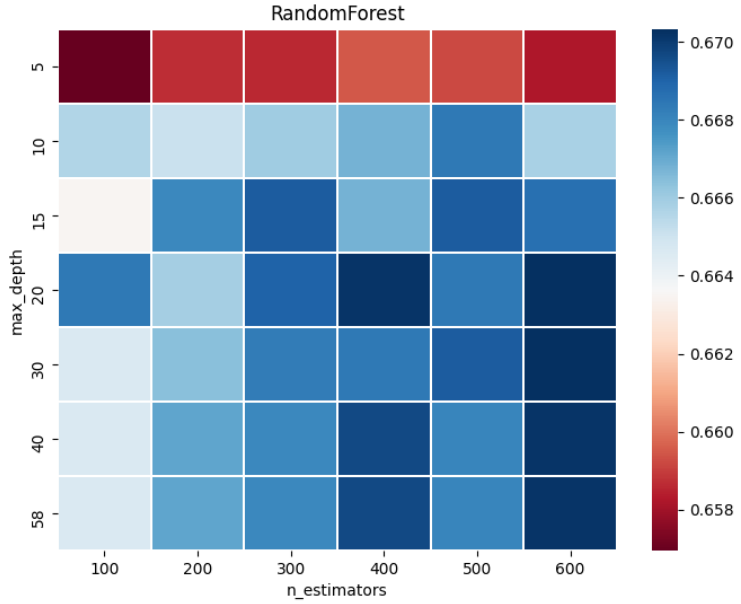

Figura 5: Mapa de calor según hiperparámetros en RandomForest.

La clase de funciones que buscamos para ajustar son la agrupación (ensemble) de funciones definidas por los hiperplanos paralelos a los ejes que particionan el espacio, donde se entrenan usando el algoritmo CART.

El preanálisis 4, 5 nos muestra que en general expectuando la profundidad máxima 5, el resto de configuraciones son casi igual de buenas (diferencias de centésimas), destacando profundidad máxima 20 con 400 árboles y profundidad máxima $>=$ 20 con 600 árboles.

Sabemos que al aumentar el nº de árboles conseguimos reducir el término de la varianza aunque a coste de incrementar el tiempo de computación; que en este caso no nos importa pagar

el precio. 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 este valor (menor varianza cuanto menor profundidad).

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]}
```

Hemos añadido el hiperparámetro `cc_alpha` que se usa para la poda de mínimo coste computacional, para experimentar con más regularización.

8.2.2. Boosting

AdaBoost

GradientBoosting

8.2.3. Multilayer Perceptron (MLP)

8.2.4. 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]}
```

TODO: Añadir clase de funciones

Los resultados preanálisis 6 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 el hiperparámetro `weights` que permite cambiar el peso de los vecinos: `uniform` todos importan igual, `distance` los vecinos importan en función de la distancia.

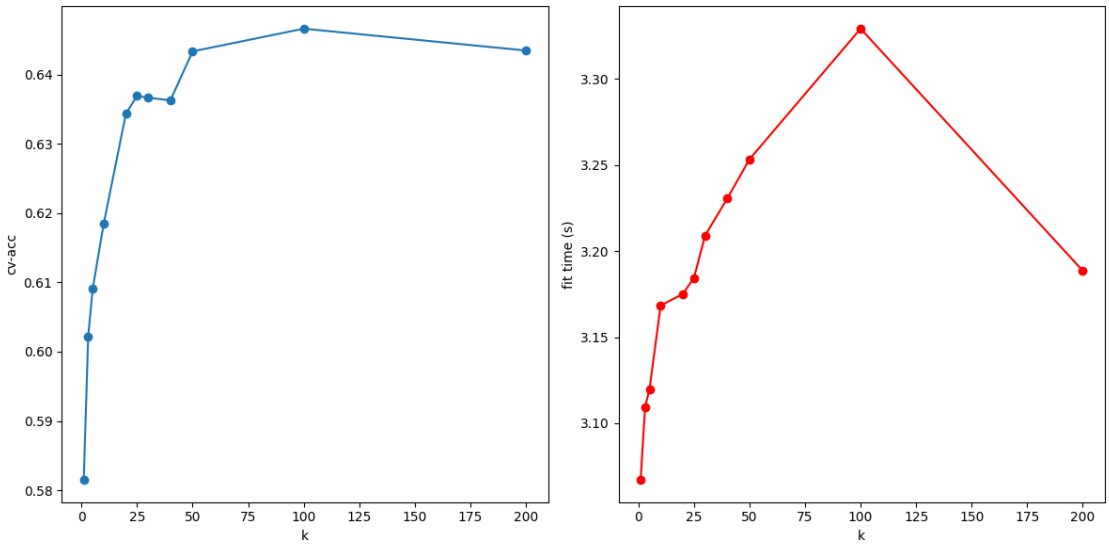

Figura 6: acc-cv/tiempo según k en KNN.

8.2.5. Funciones de Base Radial (RBF)

El clasificador está implementado siguiendo el algoritmo 1 en REFERENCIA_LIBRO por nostros en la clase `RBFNetworkClassifier`.

TODO: cambiar espacio de búsqueda?

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

$\mu = \mu_1, \ldots, \mu_k$ = 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);

```
{"clf": [RBFNetworkClassifier(random_state = SEED)],
        "clf__k": [10, 25, 50, 100, 150, 200, 250, 300]}
```

Los resultados del preanálisis 7

9. Análisis de resultados y estimación del error

Resultados de la mejor configuración de cada modelo en training/test en 1.

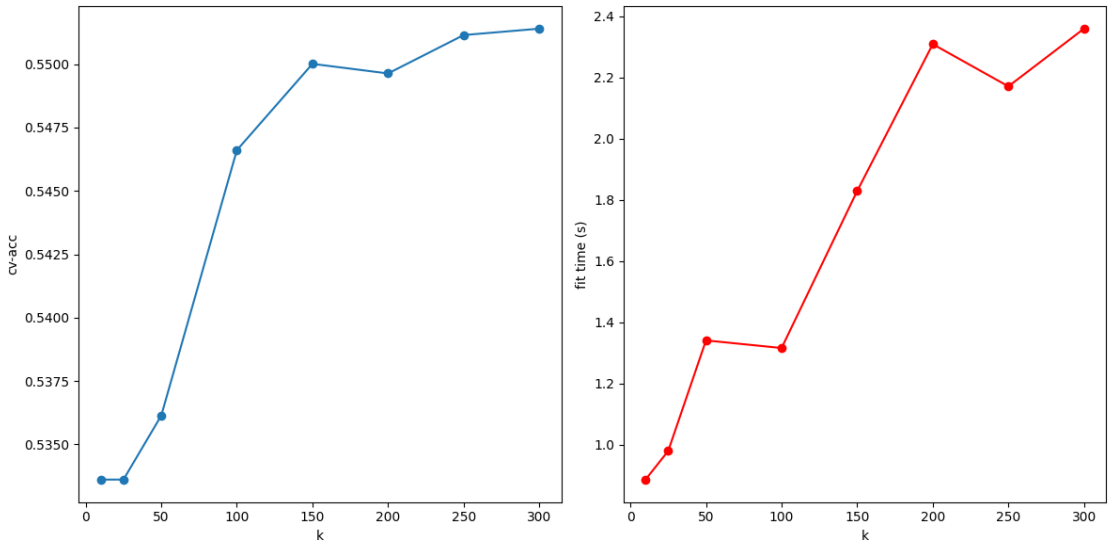

Figura 7: acc-cv/tiempo según k en RBF.

Modelo	acc_{in}	acc_{test}	AUC_{in}	AUC_{test}
Modelo Lineal	67.74	65.66	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	65.96	64.84	71.32	70.08
Aleatorio	50.43	50.58	50.08	49.78

Tabla 1: Resultados de cada modelo

10. Conclusiones y justificación

Anexo: Funcionamiento del código

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.