UNIVERSIDAD DE GRANADA

Aprendizaje Automático

Proyecto Final

Statlog (Shuttle) Data Set

Curso: 2019-2020
Autores: Alba Casillas Rodríguez y Francisco Javier Bolívar Expósito

Índice

1. Descripción del problema y enfoque elegido

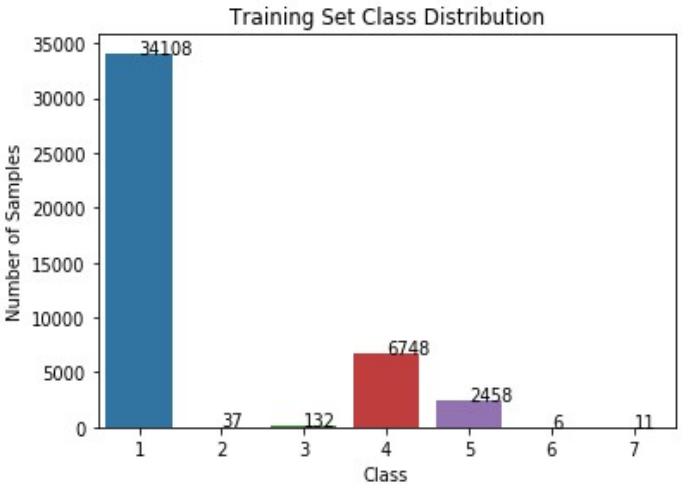
Trabajaremos con el conjunto de datos *Shuttle (Versión Statlog)*, creado por Jason Catlett, del departamento de informática de la Universidad de Sydney. Este conjunto de datos contiene 9 características y fue creado originalmente para extraer reglas comprensibles para determinar las condiciones bajo las que es preferible un aterrizaje automático al control manual de una nave espacial.

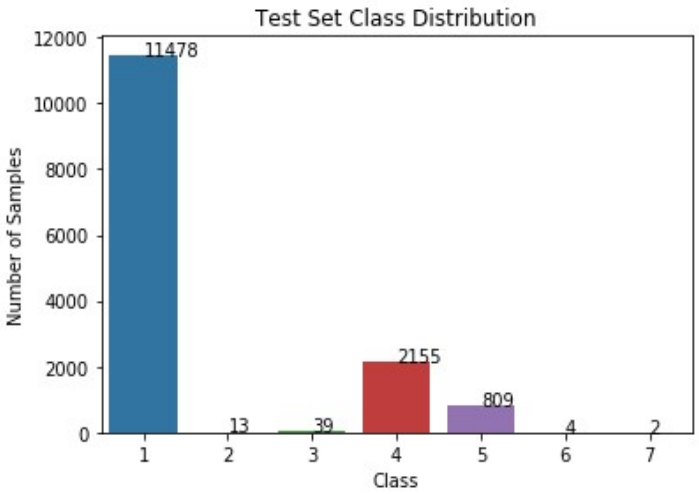
Por tanto, nuestro modelo tiene que predecir a partir de las características 'X' de entrada, las cuales son numéricas, una etiqueta 'Y'. La etiqueta puede hacer referencia a una de las 7 clases indicadas en la última columna del conjunto de datos, cuyo significado son:

1. *Rad Flow*
2. *Fpv Close*
3. *Fpv Open*
4. *High*
5. *Bypass*
6. *Bpv Close*
7. *Bpv Open*

Originalmente, nuestro conjunto de datos ya se encuentra dividido en training y test, disponiendo de 43500 datos de entrenamiento (75% de las muestras) y 14500 de test (25% de las muestras). Además, según la información proporcionada por la descripción del conjunto de datos, originalmente los datos estaban ordenados en el tiempo pero como esto podía ser relevante en la clasificación fueron barajados aleatoriamente, eliminando una parte del conjunto de datos original para la validación.

Como en la descripción se indica que aproximadamente el 80% de los datos pertenecen a la clase 1 ya sabemos de antemano que la distribución está desbalanceada, sin embargo, resulta interesante mostrar gráficamente la proporción de elementos pertenecientes a cada clase.





Co-

rroborando lo dicho anteriormente, la gran mayoría de los datos pertenecen a la clase 1. Siendo las clases 2, 3, 6 y 7 de las que tenemos menor número de muestras.

Podemos observar también que la proporción de las clases es similar en el training set y en el test set, permitiendo entrenar y obtener un error de test en conjuntos representativos del dataset original. Era importante asegurarnos de esto especialmente con el desbalance de clases existente, ya que con una partición aleatoria podrían incluso haberse quedado totalmente fuera de algún conjunto las clases con menor número de muestras.

El repositorio no dejaba constancia sobre si faltaban valores en los conjuntos de datos, por lo que comprobaremos si están todos los valores tanto en el conjunto de entrenamiento como de test mediante

isnull():

```
¿Existen valores perdidos?: False
```

Como no hay datos perdidos no será necesario eliminarlos o imputarlos.

En base al enfoque elegido para el análisis y la elección de modelos, podemos distinguir varias fases:

Comenzaremos con la lectura de los datos. Para ello haremos uso de la función ***loadtxt()*** , la cual recibirá como parámetros el fichero de texto a leer, y el delimitador que separa unos datos de otros, que en nuestro problema es un ‘ ‘ . Separaremos y almacenaremos en distintas estructuras las características de las etiquetas que, como ya se ha dicho anteriormente, se encuentran en la última columna del conjunto de datos.

Tras esto, realizaremos un análisis de los datos, donde discutiremos sobre qué técnicas y decisiones serán las más adecuadas para conocer mejor el problema y obtener unos resultados de calidad. Según este análisis elegiremos el subconjunto de variables, la codificación de los datos y la normalización a llevar a cabo, realizando así el preprocesamiento.

Después elegiremos la función de pérdida y los modelos a evaluar, comentando las técnicas de ajuste, y los hiperparámetros a probar que mejor se adapten al problema. Comentaremos además la necesidad de regularización en base a obtener unos buenos resultados.

Tras la elección, evaluaremos mediante validación cruzada los distintos modelos variando sus parámetros y poder realizar una comparación de los resultados obtenidos. Debatiremos sobre las métricas a utilizar, donde seleccionaremos una serie de valores que nos permitan comparar los modelos.

Una vez terminado el proceso de validación cruzada, elegiremos y ajustaremos los modelos en base a los mejores resultados obtenidos. Según los resultados obtenidos tanto en training como en la validación cruzada si detectamos un problema de alta varianza o alto bias revisaremos las decisiones realizadas en los anteriores pasos. Esto nos permitirá hacer una elección de nuestro mejor modelo y calcularemos el error obtenido para el conjunto de test; con lo que podremos concluir si nuestro trabajo ha dado buenos resultados.

2. Elección de modelos

2.1. Random Forest

Random Forest es un modelo basado en árboles que se sustenta en la técnica de *bagging*, aportando una mejora ya que construye árboles no correlacionados. Esta correlación sucede ya que cada árbol es generado eligiendo una muestra de datos mediante *bootsrap*, de manera que si hay un predictor (es decir, una variable que haga decrecer mucho la varianza) muy fuerte, será utilizado casi siempre en la primera partición de los datos de cada árbol. Por ello Random Forest selecciona, en tiempo de construcción, aleatoriamente una parte de las variables para la generación del árbol, de forma que elimina en gran parte la posible dependencia entre ellos.

Se ha elegido este modelo ya que trabaja de manera eficiente para problemas de clasificación multiclase. Además, este modelo en problemas de clasificación nos proporciona la probabilidad de pertenecer a una clase, siendo fácil de interpretar.

Otra de las razones es que, como bien se ha visto en clase, el sesgo del clasificador Random Forest es muy bajo por construcción, ya que se usan árboles sin podar; y una baja varianza, debido a que esta decrece al tener múltiples árboles no correlacionados.

2.2. Support Vector Machine

El segundo modelo es un SVM y estudiaremos en apartados posteriores qué tipo de *kernel* (*RBG-Gaussiano* o *Polinomial*) será el que mejor se adapte a nuestro problema.

Este modelo no depende de la dimensionalidad de las muestras y además, para problemas con datos linealmente separables garantiza el mejor estimador, es decir, una solución óptima, puesto que separa los datos maximizando el margen entre ellos. En el caso de que los datos no linealmente separables, se permiten algunas violaciones de este margen (puntos que no han sido clasificados bien), pudiendo controlarse el grado de tolerancia de errores. Es por ello que podemos esperar un resultado bueno en nuestro problema.

En este punto se ha decidido explicar el por qué de nuestra elección, por lo que se hablará de sus parámetros en apartados posteriores.

3. Codificación de los datos de entrada

Además de asegurarnos, como ya se ha dicho, de que no faltan valores, mostramos información relevante del conjunto de datos:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 43500 entries, 0 to 43499
Data columns (total 9 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   0       43500 non-null  int32
 1   1       43500 non-null  int32
 2   2       43500 non-null  int32
 3   3       43500 non-null  int32
 4   4       43500 non-null  int32
 5   5       43500 non-null  int32
 6   6       43500 non-null  int32
 7   7       43500 non-null  int32
 8   8       43500 non-null  int32
dtypes: int32(9)
memory usage: 1.5 MB
```

Como podemos ver, el tipo de los datos es *'int32'*, lo que nos indica que la representación de todas nuestras características son variables numéricas; por lo que consideramos que no necesitamos modificar la codificación del conjunto de datos.

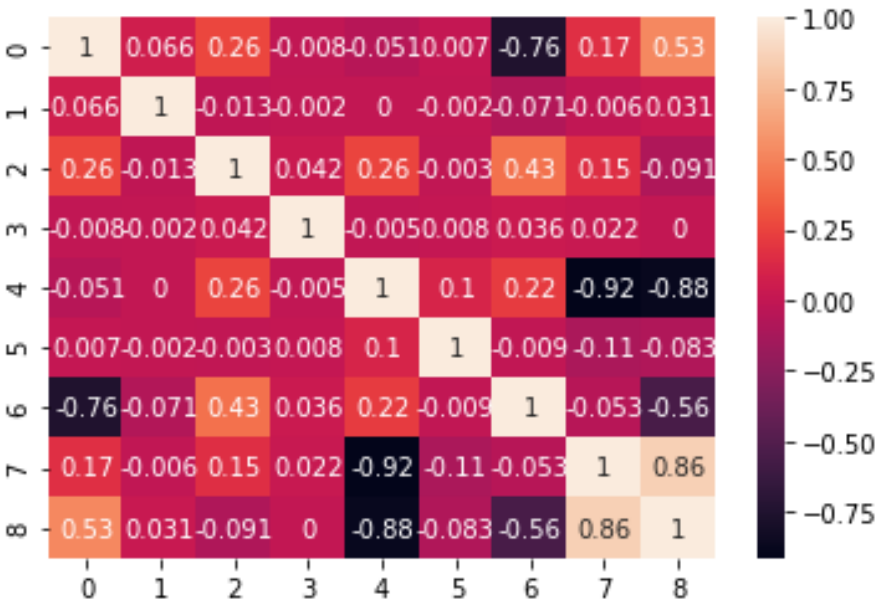
4. Valoración de variables y elección de un subconjunto

Se muestra a continuación distintas estadísticas sobre las variables de entrada.

	0	1	2	3	4	5	6	7	8
count	43500.000000	43500.000000	43500.000000	43500.000000	43500.000000	43500.000000	43500.000000	43500.000000	43500.000000
mean	48.249747	-0.205126	85.341563	0.262736	34.528782	1.298276	37.074552	50.899862	13.964598
std	12.252618	78.142770	8.908602	41.004131	21.703409	179.486760	13.135557	21.463250	25.648404
min	27.000000	-4821.000000	21.000000	-3939.000000	-188.000000	-13839.000000	-48.000000	-353.000000	-356.000000
25%	38.000000	0.000000	79.000000	0.000000	26.000000	-5.000000	31.000000	37.000000	0.000000
50%	45.000000	0.000000	83.000000	0.000000	42.000000	0.000000	39.000000	44.000000	2.000000
75%	55.000000	0.000000	89.000000	0.000000	46.000000	5.000000	42.000000	60.000000	14.000000
max	126.000000	5075.000000	149.000000	3830.000000	436.000000	13148.000000	105.000000	270.000000	266.000000

Esto nos sirve para estudiar si hay variables que aportan poco información analizando la desviación estándar según el rango de la variable. No parece haber un caso claro, ya que vemos que ninguna tiene una desviación estándar especialmente baja, teniendo en cuenta que solo disponemos de 8 características se ha decidido no eliminar ninguna por este motivo.

También resulta interesante saber si existe correlación lineal entre las características de entrada, ya que así podemos decidir si eliminar alguna característica redundante o no. Para ello, mostramos la matriz de correlación de los datos de entrenamiento:



Podemos observar que todos los atributos nos proporcionan información. Los valores en la matriz de correlación varían en un rango de [-1, 1], sin embargo, una alta correlación negativa demuestra una conexión entre dos características de igual manera que una alta correlación positiva. Por eso, destacamos que los atributos 4 y 7 tienen una correlación negativa de -0’92 y los atributos 7 y 8 una correlación positiva de 0.86.

Teniendo en cuenta el hecho de que no sabemos cuál es la importancia de cada atributo y que, además, nos encontramos en un problema que no tiene demasiadas dimensiones, consideramos que estos valores no son suficientes como para eliminar ninguna característica de entrada y perder la información que aporta esta.

Como conclusión vamos a seleccionar todas las características disponibles ya que no tenemos un gran número de ellas y perder la información que aportan podría introducir sesgo en el modelo. Si tras entrenar los modelos se detectara un problema de *overfitting* se podría revisar esta decisión.

5. Normalización de las variables

Al mostrar información general del conjunto:

	0	1	2	3	4	5	6	7	8
count	43500.000000	43500.000000	43500.000000	43500.000000	43500.000000	43500.000000	43500.000000	43500.000000	43500.000000
mean	48.249747	-0.205126	85.341563	0.262736	34.528782	1.298276	37.074552	50.899862	13.964598
std	12.252618	78.142770	8.908602	41.004131	21.703409	179.486760	13.135557	21.463250	25.648404
min	27.000000	-4821.000000	21.000000	-3939.000000	-188.000000	-13839.000000	-48.000000	-353.000000	-356.000000
25%	38.000000	0.000000	79.000000	0.000000	26.000000	-5.000000	31.000000	37.000000	0.000000
50%	45.000000	0.000000	83.000000	0.000000	42.000000	0.000000	39.000000	44.000000	2.000000
75%	55.000000	0.000000	89.000000	0.000000	46.000000	5.000000	42.000000	60.000000	14.000000
max	126.000000	5075.000000	149.000000	3830.000000	436.000000	13148.000000	105.000000	270.000000	266.000000

Si observamos las filas que muestran el valor mínimo y máximo de cada característica, no es difícil darse cuenta que hay una amplia diferencia de valores. Las características se encuentran en rangos muy diferentes, lo que puede sesgar nuestro modelo a hacer que algunas variables tengan más relevancia que otras. Por esta razón hemos decido que es necesario normalizar los datos.

Para normalizar los datos, hemos decidido estandarizarlos, es decir, reescalaremos las características de manera que la media será 0 y la desviación estándar 1, siguiendo la siguiente ecuación:

$$x_{\text{stand}} = \frac{x - \text{mean}(x)}{\text{standard deviation } (x)}$$

Para ello, hemos usado la función ***StandardScaler()*** de *scikit-learn*, modificando el parámetro "*copy=False"* para no guardar una copia del conjunto de datos. Tras aplicarlo, ajustaremos el modelo con la función ***fit_transform()***.

Como esta técnica consiste en escalar el valor de las características entre 0 y 1, resulta útil para algoritmos de optimización que ponderan con pesos las entradas (como la regresión).

Se ha elegido esta técnica ya que soluciona el problema de los rangos que tienen las variables, facilita la convergencia de algoritmos como el gradiente descendente y es más robusta a *'outliers'* que otras técnicas para la transformación de los datos.

6. Función de pérdida

Para nuestro modelo lineal, Regresión Logística, la función de pérdida utilizada es ***Categorical Cross-Entropy***, una variante de *Cross Entropy* para problemas multiclase, con la que minimizamos el error que se comete al intentar predecir que un elemento pertenece a una clase determinada.

Esta función de pérdida viene dada por la siguiente expresión:

$$E(\mathbf{w}_1, \mathbf{w}_{2,} \cdots, \mathbf{w}_K) = -\ln L(Y|\,\mathbf{w}_1, \mathbf{w}_{2,} \cdots, \mathbf{w}_K) = -\sum_{n=1}^{N}\sum_{k=1}^{K} y_{nk} \ln t_{nk}$$

donde,

$$t_{nk} = \sigma(\mathbf{w}_k^T \mathbf{x}_n),$$

En el caso de Random Forest, cómo no se intenta minimizar nada, no utiliza ninguna función de pérdida.

La función de pérdida utilizada para el SVM es ***Hinge-Loss***. Consultamos la fórmula en la documentación de *Learning From Data* sobre los SVM, proporcionada en los apuntes de la asignatura, donde:

$$E_{\text{SVM}}(b, \mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \max(1 - y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b), 0).$$

donde el término enésimo en $E_{SVM}(b,w)$ evalúa 0 si no hay violación en la muestra enésima, por lo que:

$$y_n(\mathbf{w}^{\mathrm{T}}\mathbf{x}_n + b) \geq 1;$$

Por otra parte, el término enésimo es la cantidad de violaciones correspondiente a dicho punto.

7. Selección del modelo lineal

Como modelo lineal, se ha elegido la **Regresión Logística Multinomial**, variación de la Regresión Logística para problemas multiclase; modelo que hace uso de la función logística, la ***función sigmoide***:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

La Regresión Logística Multinomial entrena un único modelo, al contrario de la Regresión Logística que utiliza el *criterio ORV (One vs Rest)*, donde se entrenan múltiples modelos. Para ello, se modela la probabilidad de que cada muestra pertenezca a una clase, teniendo en cuenta la dependencia entre estas posibilidades. Como la probabilidad de todas las clases debe sumar 100%, habrá "1 – la suma de las probabilidades" de probabilidad de que una muestra pertenezca a una clase.

El estudio de este modelo lo haremos en base a dos clases de funciones. Una de ellas será las combinaciones lineales de los valores observados. Analizando el error de validación cruzada y el error de entrenamiento, nos daremos cuenta de que el modelo no representa de manera adecuada los datos y que una combinación lineal no es suficiente para explicarlos claramente, por lo que utilizaremos una clase de funciones más compleja con combinaciones polinómicas de grado 2 (mediante ***PolynomialFeatures*** del modulo *sklearn.preprocessing*) con la intención de mejorar los resultados del modelo.

La función de un modelo lineal puede ser transformada a combinaciones no lineales:

$$Y = \theta_0 + \theta_1 x \quad \longrightarrow \quad Y = \theta_0 + \theta_1 x + \theta_2 x^2$$

En comparación con otros modelos lineales como un *SGDClassifier*, la Regresión Logística ya usa por defecto el *algoritmo de Gradiente Descendente* (aunque mas adelante trataremos con más profundidad cómo hemos abordado la métrica de ajuste); y aunque un *SGD* pueda dar unos resultados aceptables, nuestro modelo elegido es mejor clasificador, con una convergencia rápida y eficiente. Se podría haber optado por un modelo *Perceptron*, pero no tenemos certeza de si los datos son linealmente

separables; y en caso de no serlo, se necesitaría varios perceptrones para poder clasificar bien los datos, por lo que se ha decidido no utilizarlo.

Por tanto, como nos encontramos en un problema de clasificación, cuya salida es un valor categórico de entre varias clases y donde no se tiene una relación lineal entre las variables, se ha decidido elegir la Regresión Logística frente a otros modelos estudiados a lo largo de la asignatura por ser el que mejor puede adaptarse a nuestro problema.

8. Estimación de los parámetros, hiperparámetros y error de generalización.

A continuación se va a describir el proceso que se ha seguido para evaluar los modelos y por ende, poder elegir aquellos ajustes para cada modelo que nos proporcionen el mejor resultado.

Para elegir la mejor estimación de parámetros hemos utilizado ***GridSearchCV*** de *scikit-learn*. El *Grid Search* es un proceso de ajuste de hiperparámetros para determinar los valores óptimos de un modelo dado, con el objetivo de encontrar una combinación de valores que obtengan un buen rendimiento. Para ello, se le proporciona una lista de valores para elegir a cada hiperparámetro del modelo, de forma se que realizará un proceso de validación cruzada para determinar el conjunto de ajustes que proporciona los mejores niveles de precisión.

Cabe destacar que, al no indicar específicamente el valor del parámetro *"cv"*, el cuál indica la estrategia de división en la validación cruzada, por defecto se está estableciendo una validación cruzada de 5 particiones. Además, se ha modificado el parámetro *"n_jobs"* el cual le asignamos el valor '-1' para que se utilicen todas las CPUs en paralelo y así consumir menos tiempo de ejecución.

GridSearchCV nos devuelve el modelo que ha tenido la mejor combinación dado los parámetros explicados a continuación:

En nuestro caso, para la ***Regresión Logística*** (el modelo proporcionado por *sckit-learn* es ***LogisticRegression***) hemos decidido jugar con la regularización del modelo, de la cual hablaremos con más detalle en el apartado 9. Se ha probado este modelo tanto sin ningún tipo de regularización, como con las regularizaciones *Lasso* y *Ridge*, para una lista de valores de C = [0'001 – 1000 – 10^7] .

Además, es conveniente hablar también de otros elementos a tener en cuenta que han sido utilizados en el modelo:

Como técnica de ajuste, hemos utilizado el *algoritmo SAGA*, un método iterativo basado en gradientes, el cual nos proporciona una mayor efectividad y ratio de convergencia que el SGD. Proporcionamos el siguiente pseudocódigo de *SAGA* el cual podemos encontrar en el documento oficial sobre esta técnica de ajuste:

(Fuente: https://papers.nips.cc/paper/5258-saga-a-fast-incremental-gradient-method-with-support-for-non-strongly-convex-composite-objectives.pdf)

We start with some known initial vector $x^0 \in \mathbb{R}^d$ and known derivatives $f_i'(\phi_i^0) \in \mathbb{R}^d$ with $\phi_i^0 = x^0$ for each i. These derivatives are stored in a table data-structure of length n, or alternatively a $n \times d$ matrix. For many problems of interest, such as binary classification and least-squares, only a single floating point value instead of a full gradient vector needs to be stored (see Section 4). SAGA is inspired both from SAG [1] and SVRG [5] (as we will discuss in Section 3). SAGA uses a step size of γ and makes the following updates, starting with $k = 0$:

SAGA Algorithm: Given the value of x^k and of each $f_i'(\phi_i^k)$ at the end of iteration k, the updates for iteration $k+1$ is as follows:

1. Pick a j uniformly at random.
2. Take $\phi_j^{k+1} = x^k$, and store $f_j'(\phi_j^{k+1})$ in the table. All other entries in the table remain unchanged. The quantity ϕ_j^{k+1} is not explicitly stored.
3. Update x using $f_j'(\phi_j^{k+1})$, $f_j'(\phi_j^k)$ and the table average:

$$w^{k+1} = x^k - \gamma \left[f_j'(\phi_j^{k+1}) - f_j'(\phi_j^k) + \frac{1}{n} \sum_{i=1}^{n} f_i'(\phi_i^k) \right], \tag{1}$$

$$x^{k+1} = \text{prox}_\gamma^h \left(w^{k+1} \right). \tag{2}$$

The proximal operator we use above is defined as

$$\text{prox}_\gamma^h (y) := \underset{x \in \mathbb{R}^d}{\text{argmin}} \left\{ h(x) + \frac{1}{2\gamma} \|x - y\|^2 \right\}. \tag{3}$$

Hemos ampliado el número de iteraciones máximas, cuyo valor por defecto es 100 a 1000, un aumento de iteraciones aumenta la precisión con la que el modelo intenta ajustarse a los datos lo que genera un buen ajuste para los datos de entrenamiento, aunque puede conducir a *overfitting*. Sin embargo, tras probar con ambos ajustes, no sólo hemos comprobado que no se genera *overfitting*, sino que obtenemos un error de validación cruzada mejor, por lo que mantendremos esta decisión.

Por último, mantenemos con los valores por defecto los parámetros *dual* (=False , ya que el número de muestras es mayor que el de características) y *tol* (= 0'0001), que indica el criterio de parada.

Para ***Random Forest*** (cuyo modelo proporcionado por *scikit-learn* es ***RandomForestClassifier***) experimentamos con distintos números de árboles en busca del adecuado, para ello modificaremos el parámetro *"n_estimators"*. Se ha probado con los valores 10, 100, 250 y 500; ya que los valores por defecto varían de 10 a 100, sin embargo, en base a la *documentación de scikit-learn*, se indica que a media que se aumenta el número de árboles (e inevitablemente, el tiempo de ejecución), también mejora el resultado; los cuales dejarán de mejorar significativamente más allá de un valor crítico de árboles, por eso decidimos aumentar el número de 250 y 500.

Otro de los parámetros más importantes que se ha ido modificando ha sido *"max_features"* , que se corresponden con el número de predictores, es decir, la cantidad máxima de características a probar en un árbol individual. El aumento de esta característica generalmente mejora el rendimiento del modelo, ya que en cada nodo habrá un mayor número de opciones a considerar, aunque la velocidad del algoritmo y la diversidad de los árboles puede verse disminuida; por ello, se ha probado a modificar este parámetro con algunos valores discutidos en clase como:

- ***Auto (m= p)*** , no colocamos ninguna restricción en el árbol individual y usamos todas las características que tienen sentido en cada árbol (simplemente obtenemos *bagging*).

- ***sqrt (m =*** $\sqrt{p}$***)*** , tomaremos la raíz cuadrada del número total de características en cada árbol.

- ***log2 (m = log2(p))*** , se realizará el logaritmo del número total de características para cada árbol.

En cuanto al ***modelo SVM*** (cuyo modelo proporcionado por *scikit-learn* para problemas de clasificación es ***SVC***) probaremos con dos tipos de *Kernel*: el *polinomial* y el *RBF-Gaussiano*. Ambos tipos de núcleo permiten el aprendizaje en modelos no lineales, la diferencia se encuentra en que un *kernel polinomial* no solo observa las características de entrada para determinar su similitud, sino también combinaciones de estas; pero al fin y al cabo estamos tratando con un modelo paramétrico. Un SVM con *kernel RBF* no lo es, por lo que la complejidad de este crece con el tamaño del conjunto de datos; por lo que a priori podremos esperar un buen rendimiento y eficiencia ante este tipo de núcleo.

Las funciones utilizadas para estos núcleos son las siguientes:

$$k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j} + 1)^d$$

Polynomial kernel equation

$$k(\mathbf{x_i}, \mathbf{x_j}) = \exp(-\gamma \|\mathbf{x_i} - \mathbf{x_j}\|^2) \quad \gamma > 0$$

Gaussian radial basis function (RBF)

Para indicar qué tipo de núcleo se ha utilizado, hemos modificado el parámetro ***kernel*** (a *'rbf'* o *'poly'*).

También ajustaremos los parámetros ***C*** y ***gamma (γ)***. Sin embargo, es conveniente explicar el significado de ambos parámetros antes de indicar qué valores se han elegido:

El *hiperparámetro C* es un parámetro de regularización el cual está ligado a la importancia a la que se le asocia las violaciones producidas en el margen al separar las clases. Cuanto menos sea el valor de este parámetro, menos peso tendrán estas violaciones, consiguiendo un margen más grande.

Por otro lado, "*γ*" define qué tan grande es la influencia de un punto de la muestra. Por ello, un valor bajo de γ significará que los puntos tienen una influencia alta, por lo que se tendrán en cuenta algunos puntos a pesar de estar alejados de la muestra; afectando además a la regularización, debido a que un valor alto de γ tomará influencia de puntos muy cercanos, lo que podría conducir a *overffiting*.

Se ha decidido probar con distintos valores de C y γ de forma que:

C = [0'1, 10, 100, 1000]

γ = [0'1, 0'01, 0'001]

A pesar de que en las funciones de los kernel mostradas anteriormente se puede observar que el *kernel polinomial* no usa el valor de γ , lo hemos añadido en el ajuste puesto que en librerías como *sckit-learn* añaden este parámetro en su función.

(Fuente: https://stats.stackexchange.com/questions/375340/why-do-we-need-the-gamma-parameter-in-the-polynomial-kernel-of-svms)

The polynomial kernel is sometimes defined as just:

$$K(x, y) := (\langle x, y\rangle + c)^d$$

with two parameters: the degree d and constant coefficient c.

But others (e.g., libsvm, and sklearn which uses libsvm) include a scaling parameter γ:

$$K'(x, y) := (\gamma \langle x, y\rangle + r)^d$$

Cabe destacar que para el *kernel de tipo polinomial* se podría haber ajustado el parámetro *"degree"*, el cual indica el grado de la función polinómica. Se ha decidido simplemente probar con su valor por defecto (grado 3) , ya que de lo contrario el tiempo de ejecución aumentaba considerablemente y no se obtenía ningún resultado significativo.

Se ha utilizado ***Accuracy*** como métrica de error , la cual indica el porcentaje de muestras correctamente clasificadas.

CV para RL

	mean_fit_time	param_C	param_penalty	mean_test_score	std_test_score	rank_test_score
0	39.607437	NaN	none	0.964230	0.002092	1
1	5.912332	0.001	l1	0.917954	0.001329	15
2	5.113410	0.001	l2	0.919793	0.001995	14
3	16.876880	0.01	l1	0.955126	0.001572	12
4	11.314784	0.01	l2	0.950874	0.002679	13
5	39.940645	0.1	l1	0.962184	0.001903	10
6	28.585904	0.1	l2	0.961655	0.002111	11
7	42.415048	1	l1	0.963954	0.001978	8
8	22.729787	1	l2	0.963931	0.001953	9
9	33.903027	10	l1	0.964230	0.002092	1
10	30.368894	10	l2	0.964230	0.002092	1
11	45.699669	100	l1	0.964230	0.002092	1
12	25.062831	100	l2	0.964230	0.002092	1
13	36.621204	1000	l1	0.964230	0.002092	1
14	24.717740	1000	l2	0.964230	0.002092	1

Resultados de la CV para los parámetros de la RL

CV para RL con combinación no lineal

	mean_fit_time	param_C	param_penalty	mean_test_score	std_test_score	rank_test_score
0	83.377794	NaN	none	0.977241	0.001375	1
1	124.424411	0.001	l1	0.924805	0.001861	15
2	55.974928	0.001	l2	0.951011	0.002223	14
3	114.178337	0.01	l1	0.963448	0.001971	13
4	80.763234	0.01	l2	0.968989	0.001405	12
5	129.294917	0.1	l1	0.975747	0.001186	11
6	80.983414	0.1	l2	0.976391	0.001332	10
7	126.974350	1	l1	0.977126	0.001298	9
8	83.320545	1	l2	0.977149	0.001329	8
9	122.739895	10	l1	0.977241	0.001375	1
10	72.164826	10	l2	0.977241	0.001375	1
11	137.675629	100	l1	0.977241	0.001375	1
12	68.060810	100	l2	0.977241	0.001375	1
13	117.365560	1000	l1	0.977241	0.001375	1
14	65.126053	1000	l2	0.977241	0.001375	1

Resultados de la CV para los parámetros de la RL con combinación no lineal

CV para RF

	mean_fit_time	param_max_features	param_n_estimators	mean_test_score	mean_score_time	std_score_time
0	0.166555	auto	10	0.999724	0.013365	0.011777
1	1.581971	auto	100	0.999793	0.056448	0.001017
2	3.993165	auto	250	0.999793	0.147606	0.003153
3	7.886319	auto	500	0.999816	0.292019	0.002554
4	0.161368	sqrt	10	0.999701	0.007381	0.000488
5	1.574501	sqrt	100	0.999747	0.059640	0.000747
6	3.870842	sqrt	250	0.999816	0.146314	0.003365
7	7.803336	sqrt	500	0.999816	0.288459	0.008695
8	0.152193	log2	10	0.999770	0.007283	0.000603
9	1.612624	log2	100	0.999793	0.059042	0.001323
10	3.787912	log2	250	0.999793	0.139029	0.006869
11	6.652698	log2	500	0.999793	0.239770	0.004444

Resultados de la CV para los parámetros del RF

CV para SVM

	mean_fit_time	param_C	param_gamma	param_kernel	mean_test_score	std_test_score	rank_test_score
0	3.291206	0.1	0.1	rbf	0.994897	0.000330	8
1	8.295435	0.1	0.01	rbf	0.966184	0.001974	11
2	13.888298	0.1	0.001	rbf	0.917379	0.001307	18
3	0.923832	10	0.1	rbf	0.998391	0.000317	5
4	1.990482	10	0.01	rbf	0.997563	0.000359	6
5	5.121851	10	0.001	rbf	0.978253	0.001771	10
6	0.677396	100	0.1	rbf	0.998598	0.000276	3
7	0.911279	100	0.01	rbf	0.998598	0.000211	3
8	3.153205	100	0.001	rbf	0.994575	0.000784	9
9	0.611768	1000	0.1	rbf	0.998713	0.000303	2
10	0.735450	1000	0.01	rbf	0.998989	0.000256	1
11	1.618844	1000	0.001	rbf	0.997379	0.000407	7
12	13.106737	0.1	0.1	poly	0.922506	0.001545	16
13	25.686763	0.1	0.01	poly	0.844368	0.000442	21
14	22.284027	0.1	0.001	poly	0.784230	0.000169	24
15	20.818013	10	0.1	poly	0.949540	0.001755	14
16	12.971150	10	0.01	poly	0.902851	0.001357	19
17	22.527463	10	0.001	poly	0.791287	0.001103	23
18	43.883418	100	0.1	poly	0.955402	0.001758	13
19	11.363736	100	0.01	poly	0.922506	0.001545	16
20	18.326795	100	0.001	poly	0.844368	0.000442	21
21	134.123955	1000	0.1	poly	0.964483	0.001908	12
22	10.577104	1000	0.01	poly	0.935011	0.001418	15
23	12.863800	1000	0.001	poly	0.879747	0.001663	20

Resultados de la CV para los parámetros del SVM

Finalmente los hiperparámetros escogidos y el error de generalización (el cual será " 1-Accuracy") son:

- Regresión Logística: Sin regularización

- Regresión Logística Polinomial: Sin regularización

- Random Forest: max_features = 'auto', n_estimators = '500'

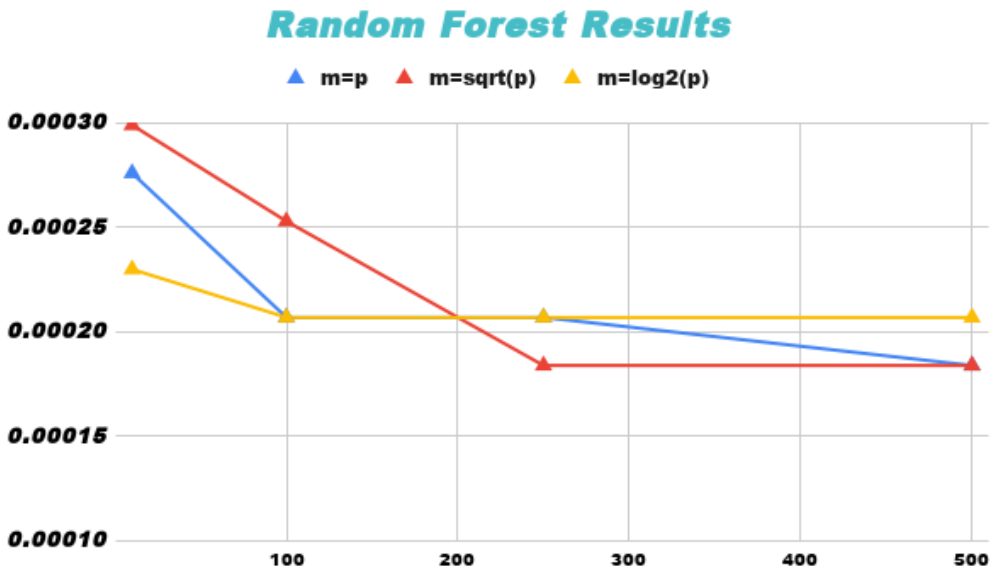
- SVM: Kernel: 'RBF', C = '1000', gamma = '0.01'

```
Resultados de selección de hiperparámetros por validación cruzada
LR Best hyperparameters:  {'penalty': 'none'}
LR CV-Accuracy : 0.9642298850574713
LRP Best hyperparameters:  {'penalty': 'none'}
LRP CV-Accuracy : 0.9772413793103448
RF Best hyperparameters :  {'max_features': 'auto', 'n_estimators': 500}
RF CV-Accuracy : 0.9998160919540231
SVM Best hyperparameters :  {'C': 1000, 'gamma': 0.01, 'kernel': 'rbf'}
SVM CV-Accuracy : 0.9989885057471266
```

Mostramos una tabla con los mejores modelos obtenidos:

Modelo	Accuracy
LR sin regularización	**0.964229**
LRP sin regularización	**0.977241**
RF max_features: auto , n_estimators: 500	**0.999816**
SVM kernel RBF, C: 1000, gamma: 0.01	**0.998988**

Comencemos hablando de Random Forest quien es, a priori, aquel que ha mostrado el mejor resultado. Para ello, ofrecemos también una gráfica (*generada con Google Docs*) sobre como ha variado el error de generalización en el proceso de validación cruzada con respecto al número de árboles:



Observamos que, aunque el mejor modelo elegido indicase que $m = p$, no es un factor demasiado relevante, ya que en todos los casos hemos obtenido valores muy similares hasta el punto de que con otras maneras de generar los subconjuntos de características (como *m=sqrt(p)*) obtenemos exactamente el mismo error de generalización. Lo que si podemos concluir es que un factor decisivo es el número de árboles, ya que a medida que se aumenta el valor de dicho parámetro, nuestro modelo es capaz de ajustarse mejor a los datos.

Para los modelos SVM hemos visto que usar un *kernel polinomial* no da buenos resultados. Con este tipo de núcleo, el mejor ha sido aquel que tiene un valor C: 1000 y λ: 0'1 , por lo que vemos que necesita darle mucho peso a las violaciones para poder hacer una clasificación medianamente buena, ya que su resultado (***accruacy: 0.964483***) ni siquiera puede acercarse al mejor resultado obtenido con el *kernel radial*. Es evidente que para este problema, la mejor opción es usar un modelo no paramétrico *RBF* , aunque también se necesitará un valor intermedio de λ y darle gran importancia a las violaciones.

Los peores resultados son los proporcionados por la Regresión Logística. Aunque aplicando una clase de funciones con combinaciones no lineales ha mejorado el resultado, no ha sido suficiente como para alcanzar a ninguno de los dos modelos.

Para analizar con más profundidad estos resultados, en el apartado 10 haremos una valoración de resultados tras entrenar los modelos con los hiperparámetros elegidos y con el conjunto de datos de entrenamiento.

9. Regularización

En base de la ***Regresión Logística***, tenemos un reducido número de características de las que no se han encontrado motivos para suponer que no aportan información suficiente y no son predictivas (no podemos descartar ninguna característica por tener una baja desviación estandar). Por este motivo la selección de características que proporciona *Lasso* no parece ser de interés para resolver este problema. *Ridge* que solo intenta reducir los coeficientes sin embargo podría ser interesante para reducir la varianza del modelo si esta fuera un problema.

Se puede estimar fácilmente usando validación cruzada cual es la mejor opción, por lo que se han probado ambas regularizaciones con distinta relevancia de la regularización. De este forma comprobamos que los mejores resultados se obtienen **sin aplicar regularización**.

Las funciones usadas para hacer las pruebas con regularización son las siguientes, donde se usa el parámetro C para controlar la relevancia de la regularización mediante la variable λ.

La relación entre C y λ es: **C = 1/** λ . Por tanto, cuanto menor sea el valor de C, más fuerte será la regularización.

Regularización Lasso, consistente en añadir a la función de coste como penalización el valor absoluto de la magnitud de los coeficientes.

$$min_w \sum_{i=1}^{n} |w_i| + C \cdot CostFunction \qquad min_w \|w\|_1 + C \cdot CostFunction$$

Forma Vectorizada

Regularización Ridge, consistente en añadir a la función de coste como penalización la magnitud al cuadrado de los coeficientes.

$$min_w \sum_{i=1}^{n} W_i^2 + C \cdot CostFunction \qquad min_w w^T w + C \cdot CostFunction$$

Forma Vectorizada

En ***Random Forest*** no podemos hablar específicamente de regularización, debido a que, como ya se ha explicado anteriormente, es un modelo que presenta una varianza baja, lo que no lo hace tan susceptible al sobreajuste.

La regularización del ***SVM*** viene dada por los parámetros C y γ , cuya influencia en los resultados y regularización ya ha sido explicada con anterioridad en la propia descripción del modelo.

10. Valoración de resultados

Con la estimación por validación cruzada obtenemos que aproximadamente: $\boldsymbol{E_{out}(g) <= E_{cv}(g)}$ (medida de error donde cuanto menor error, mejor modelo).

Podemos obtener el porcentaje de muestras mal clasificadas (*Miss-classification*) mediante “1 - Accuracy” y utilizar esta medida para calcular una estimación de $\boldsymbol{E_{out.}}$ de nuestros modelos ya entrenados. Por tanto, a partir de lo obtenido en el proceso de validación cruzada tendríamos:

Cota RL: $\boldsymbol{E_{out}(g) <= 0{,}035771}$

Cota RLP: $\boldsymbol{E_{out}(g) <= 0{,}022759}$

Cota RF: $\boldsymbol{E_{out}(g) <= 0{,}000184}$

Cota SVM: $\boldsymbol{E_{out}(g) <= 0{,}001012}$

```
Métricas de evaluación para los modelos entrenados para train y test
LR Train-Accuracy:  0.9651954022988506
LRP Train-Accuracy:  0.9781149425287357
RF Train-Accuracy:  1.0
SVM Train-Accuracy:  0.9997011494252873

LR Test-Accuracy:  0.9667586206896551
LRP Test-Accuracy:  0.979448275862069
RF Test-Accuracy:  0.9998620689655172
SVM Test-Accuracy:  0.9988275862068966
```

A pesar de que *accuracy* es una métrica intuitiva y fácil de interpretar, creemos conveniente, sobre todo por el hecho de que nuestras clases están desbalanceadas, realizar un análisis más exhaustivo para poder ver qué elementos de cada clase clasifica correctamente y en cuales falla (y ver en qué clase las clasifica). De esta manera nos aseguraremos de que nuestros modelos estén clasificando bien todas las clases y no solamente aquellas donde hay un mayor número de elementos ya que de ser así, podríamos eliminar alguna clase si se considera no apta para el aprendizaje.

Para ello, vamos a utilizar ***otras métricas de evaluación***, las cuales obtendremos a partir de la función ***classification_report*** del módulo *sklearn.metrics*:

- ***Precission***: cuantifica el número de predicciones de clase positivas que realmente pertenecen a una clase positiva.

$$Precision = \frac{TP}{TP + FP}$$

donde TP (*True Positives*) son los elementos de una clase clasificados correctamente y FP (*False Positives*) son los elementos que se han predicho que pertenecen a una clase distinta a la que realmente pertenecen.

- ***Recall***: cuantifica el número de predicciones de clase positivas hechas de todos los ejemplos positivos del conjunto de datos.

$$Recall = \frac{TP}{TP + FN}$$

donde TP son los elementos de una clase clasificados correctamente y FN (*Flase Negatives*) son los elementos de una clase que han sido clasificados como elementos de otra.

- ***F- Measure:*** es la media armónica entre *precission* y *recall*:

$$F-Measure=\frac{Precission*Recall}{Precission+Recall}$$

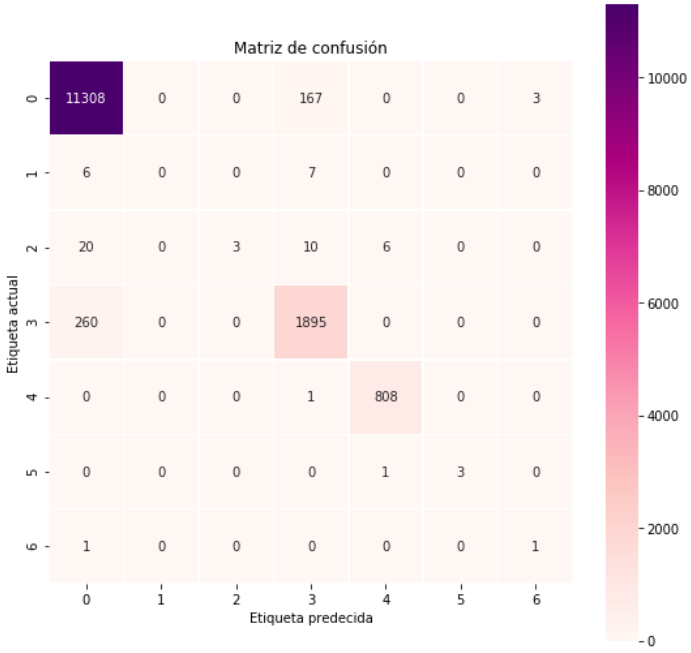
Por último, se muestra un parámetro ***support***, que indica el número de ocurrencias de cada clase en el conjunto de datos.

También proporcionaremos la ***matriz de confusión***, herramienta que permite la visualización del desempeño de un modelo de aprendizaje automático donde el eje horizontal (las columnas) representa los valores que ha predicho, mientras que el eje vertical se corresponde a los valores reales de las etiquetas. NOTA: Aunque en la matriz aparezcan que las etiquetas toman un valor entre [0,6], realmente los consideraremos entre [1,7].

<u>- Regresión Logística: Sin regularización</u>

Matriz de resultados LR:

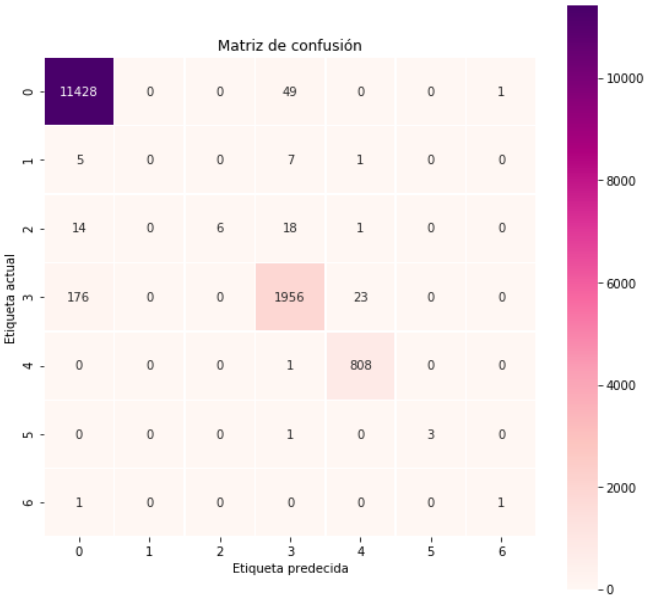
	precision	recall	f1-score	support
1	0.98	0.99	0.98	11478
2	0.00	0.00	0.00	13
3	1.00	0.08	0.14	39
4	0.91	0.88	0.89	2155
5	0.99	1.00	1.00	809
6	1.00	0.75	0.86	4
7	0.25	0.50	0.33	2
accuracy			0.97	14500
macro avg	0.73	0.60	0.60	14500
weighted avg	0.97	0.97	0.97	14500



- Regresión Logística Polinomial: Sin regularización

Matriz de resultados LRP:

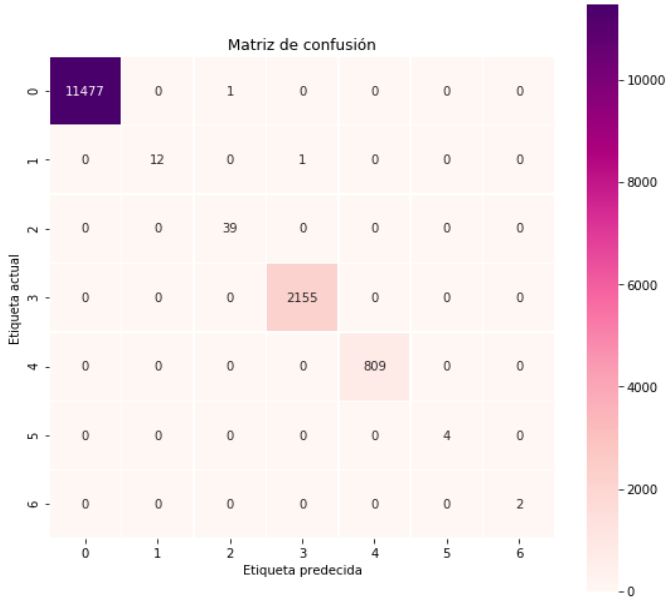
	precision	recall	f1-score	support
1	0.98	1.00	0.99	11478
2	0.00	0.00	0.00	13
3	1.00	0.15	0.27	39
4	0.96	0.91	0.93	2155
5	0.97	1.00	0.98	809
6	1.00	0.75	0.86	4
7	0.50	0.50	0.50	2
accuracy			0.98	14500
macro avg	0.77	0.62	0.65	14500
weighted avg	0.98	0.98	0.98	14500



Random Forest: max_features = 'auto', n_estimators = '500'

Matriz de resultados RF:

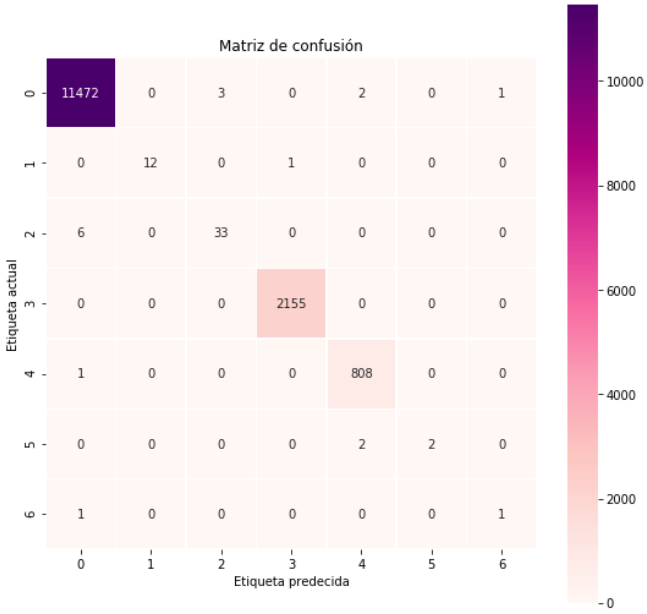
	precision	recall	f1-score	support
1	1.00	1.00	1.00	11478
2	1.00	0.92	0.96	13
3	0.97	1.00	0.99	39
4	1.00	1.00	1.00	2155
5	1.00	1.00	1.00	809
6	1.00	1.00	1.00	4
7	1.00	1.00	1.00	2
accuracy			1.00	14500
macro avg	1.00	0.99	0.99	14500
weighted avg	1.00	1.00	1.00	14500



- SVM: Kernel: 'RBF' , C = '1000' , gamma = '0.01'

Matriz de resultados SVM:

	precision	recall	f1-score	support
1	1.00	1.00	1.00	11478
2	1.00	0.92	0.96	13
3	0.92	0.85	0.88	39
4	1.00	1.00	1.00	2155
5	1.00	1.00	1.00	809
6	1.00	0.50	0.67	4
7	0.50	0.50	0.50	2
accuracy			1.00	14500
macro avg	0.92	0.82	0.86	14500
weighted avg	1.00	1.00	1.00	14500



En vista de los resultados obtenidos, ahora es más intuitivo ver por qué en ambos modelos de Regresión Logística no obtenemos buenos resultados: para clases con muy pocas ocurrencias como pueden ser la 2 o la 7, acierta en menos de la mitad de los casos. Observamos que, por ejemplo, para la clase 2 donde se obtiene una precisión del 0% y por tanto, la mayoría de los errores, se ha confundido 7 veces con la clase '3' y 5 o 6 veces con la clase '1'.

Se podrían haber eliminado estas clases, pero nos encontramos con que esto se arregla en mayor parte con los modelos de Random Forest y Support Vector Machine. Para el SVM, sigue siendo complicado no cometer un error en la clase 7, que acierta el 50% de las veces (como solo tiene dos ocurrencias, quiere decir que solo ha predicho bien en una de ellas). Aún así, no hay ninguna duda sobre que Random Forest es un buen predictor, acertando casi en el 100% de los casos.

11. Justificación de resultados

En el mundo real, no podemos calcular un *Eout* exacto, puesto que no conocemos la totalidad de las instancias posibles. A partir de la ***desigualdad de Hoeffding*** podemos obtener una cota probabilística de *Eou*t:

$$E_{out}(g) <= E_{test}(g) + \sqrt{(\frac{1}{2N} ln \frac{2}{\alpha})}$$

donde N es el tamaño de nuestro conjunto test (130500) y $\boldsymbol{\alpha}$ = 0.05 , es decir, una confianza del 95%.

Tenemos las siguientes cotas de Eout con el 95% de confianza:

Cota LR: $E_{out}(g) <= 0{,}03324137 + \sqrt{\frac{1}{2*130500} \cdot \ln \frac{2}{0.05}} = 0{,}03803$

Cota LRP: $E_{out}(g) <= 0{,}0205517 + \sqrt{\frac{1}{2*130500} \cdot \ln \frac{2}{0.05}} = 0{,}025342$

Cota RF: $E_{out}(g) <= 0{,}00013794 + \sqrt{\frac{1}{2*130500} \cdot \ln \frac{2}{0.05}} = 0{,}004929$

Cota SVM: $E_{out}(g) <= 0{,}0011724 + \sqrt{\frac{1}{2*130500} \cdot \ln \frac{2}{0.05}} = 0{,}00596$

El modelo lineal escogido (la regresión logística) es que el peor resultados nos aporta. No consigue explicar totalmente los datos del conjunto de entrenamiento y en la validación cruzada obtiene un error similar al Ein, mostrando un por lo tanto un problema de 'bias'. Esto observación se refuerza al ver que mejora al usar una combinación no lineal de las características. Con estos resultados en el modelo lineal (que según el problema de clasificación al que nos enfrentemos puede dar resultados suficientemente buenos) parece una buena idea el estudio de modelos no lineales que puedan ajustarse mejor al conjunto de entrenamiento.

A la vista de los resultados obtenidos, **el modelo que mejor se adapta a nuestro problema es Random Forest con *max_features = 'auto' , n_estimators = '500',* con menos de un 1% de error con un 95% de confianza. Podemos asegurar que nuestro modelo es bastante bueno.**

De haber obtenido un peor resultado, podríamos indicar que obteniendo una muestra más grande de datos daríamos una cota con un intervalo más reducido, puesto que la cantidad de los datos es un factor decisivo tanto a la hora de entrenar un modelo como para acotar el error en el aprendizaje automático. Sin embargo, hemos obtenido un resultado muy bueno como para considerar esto.

Cuando se obtuvo el error del conjunto de entrenamiento para este modelo, obtuvimos un *accuracy* de 1.0 , es decir, una clasificación perfecta; esto podría habernos llevado a suponer que estábamos tratando con un problema de *overffiting* tras haberse ajustado demasiado a los datos de entrenamiento. Sin embargo, el error que obteníamos al usar la validación cruzada nos indicaba una buena generalización fuera de la muestra de entrenamiento, al consultar el error obtenido con el conjunto de test vemos que efectivamente hemos obtenido un buen resultado. Esto significa que la muestra de entrenamiento representa muy bien al modelo y que ha sido bien entrenado aprendiendo el patrón que hay entre los datos de entrada y los resultados.

Como conclusión final, podría suponerse que para un conjunto de datos con una distribución similar, se obtendría un resultado similar pero que, por desgracia, como nuestro conjunto de datos está tan desbalanceado, ante una entrada de datos donde haya una mayor representación de las etiquetas menos representadas en el conjunto de entrenamiento podríamos tener unos resultados no tan buenos.

12. Bibliografía

https://sci2s.ugr.es/keel/dataset.php?cod=108

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

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

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html#sklearn.ensemble.RandomForestClassifier

http://numerentur.org/nucleo-kernel-de-las-svm/

https://www.researchgate.net/post/Diffference_between_SVM_Linear_polynmial_and_RBF_kernel

https://www.quora.com/Why-does-RBF-kernel-generally-outperforms-linear-or-polynomial-kernels/answer/Ferenc-Husz%C3%A1r?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

https://www.researchgate.net/post/Does_anyone_know_what_is_the_Gamma_parameter_about_RBF_kernel_function

https://www.quora.com/What-are-C-and-gamma-with-regards-to-a-support-vector-machine

https://data-flair.training/blogs/svm-kernel-functions/

https://en.wikipedia.org/wiki/Polynomial_kernel

https://stats.stackexchange.com/questions/375340/why-do-we-need-the-gamma-parameter-in-the-polynomial-kernel-of-svms

https://joshlawman.com/metrics-classification-report-breakdown-precision-recall-f1/

https://towardsdatascience.com/polynomial-regression-bbe8b9d97491

https://datascience.stackexchange.com/questions/6838/when-to-use-random-forest-over-svm-and-vice-versa

https://towardsdatascience.com/comparing-different-classification-machine-learning-models-for-an-imbalanced-dataset-fdae1af3677f

https://machinelearningmastery.com/bagging-and-random-forest-for-imbalanced-classification/

https://www.kdnuggets.com/2020/04/data-transformation-standardization-normalization.html

https://stats.stackexchange.com/questions/10289/whats-the-difference-between-normalization-and-standardization

https://gombru.github.io/2018/05/23/cross_entropy_loss/

https://arxiv.org/pdf/1808.02169.pdf

https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc

https://medium.com/datadriveninvestor/an-introduction-to-grid-search-ff57adcc0998

https://www.analyticsvidhya.com/blog/2015/06/tuning-random-forest-model/

https://papers.nips.cc/paper/5258-saga-a-fast-incremental-gradient-method-with-support-for-non-strongly-convex-composite-objectives.pdf

https://stats.stackexchange.com/questions/36165/does-the-optimal-number-of-trees-in-a-random-forest-depend-on-the-number-of-pred/36183

https://en.wikipedia.org/wiki/Polynomial_regression

https://machinelearningmastery.com/logistic-regression-for-machine-learning/