Práctica Final

Ajuste del mejor modelo para clasificación de cardiotocografías

Celia Arias Martínez
Miguel Ángel Fernández Gutiérrez

Aprendizaje Automático
4º Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Índice

1. Introducción

El objetivo de la práctica final es seleccionar y ajustar el mejor predictor para la base de datos *Cardiotocography*, de UCI Machine Learning Repository.[1] Para ello, compararemos un modelo lineal como **regresión logística**, con otros no lineales como **perceptrón multicapa**, **SVM** (*Support Vector Machine*), y ***random forest***.

La estrategia que seguiremos es la siguiente:

1. Comprender el problema que pretendemos resolver.
2. Preprocesar los datos para poder trabajar adecuadamente con ellos.
3. Fijar un conjunto de modelos y usar validación para escoger el mejor de ellos.
4. Analizar los resultados y estimar el error del modelo.
5. Sacar conclusiones del desempeño de nuestro modelo.

Para la implementación haremos uso principalmente de las funciones de `scikit-learn`, que se especificarán en esta memoria.

2. Descripción del problema

Lo primero que tenemos que identificar es el problema que queremos resolver. En este caso, ayudados por la información proporcionada por el dataset de UCI,[1] podemos ver que tenemos datos correspondientes a cardiotocografías en fetos.

Una **cardiotocografía** es un método de evaluación fetal que registra simultáneamente un conjunto de características tales como la frecuencia cardiaca fetal, los movimientos fetales y las contracciones uterinas.

Según el *dataset* podemos resolver dos problemas: uno que consiste en asignar un código de patrón de clase, con números del uno al diez; y otro que consiste en asignar un código de estado al feto, para determinar si es normal, sospechoso o patológico. Hemos decidido resolver el segundo problema, ya que estamos más familiarizados con esa terminología, y por tanto vamos a poder entender mejor el problema y los datos que nos proporcionan.

Usaremos el *dataset* proporcionado, consistente en un conjunto de 2126 instancias que recogen información de 22 atributos. La descripción exhaustiva de estos atributos está disponible en el *dataset* original. Entre estos podemos encontrar los movimientos del feto por segundo, las contracciones uterinas por segundo o el porcentaje del tiempo con variabilidad normal, además del estado del feto, que es el valor que queremos predecir.

Estamos, por tanto, ante un problema de **clasificación multiclase** en el que, en concreto:

- El espacio de características $\mathcal{X}$ está conformado por las 21 características del feto, y por tanto es $\mathcal{X} = \mathbb{R}^{21}$.

[1] *UCI Machine Learning Repository: Cardiotocography Data Set*. UCI Machine Learning Repository. Recuperado 13 de junio de 2021, de http://archive.ics.uci.edu/ml/datasets/Cardiotocography

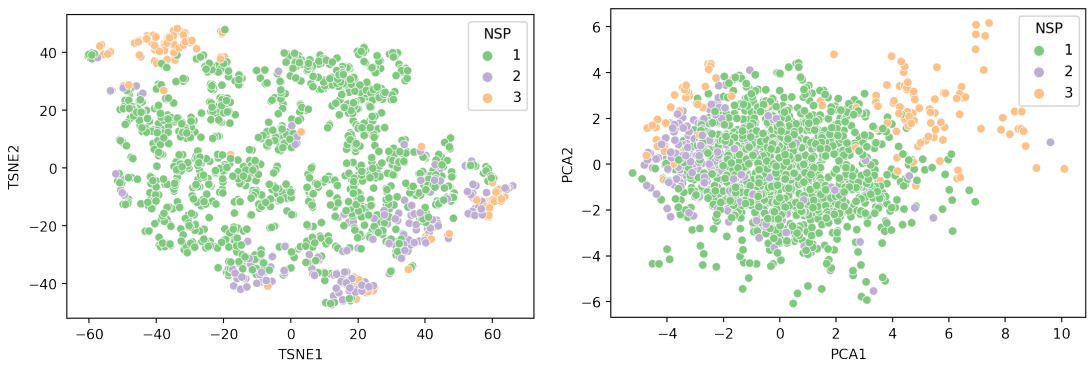
- El conjunto de etiquetas $\mathcal{Y} = \{1, 2, 3\}$ clasifica el estado de cada feto, correspondiendo la etiqueta 1 a la característica "normal" (N), la etiqueta 2 a la característica "sospechoso" (S) y la etiqueta 3 a la característica "patológico" (P).
- La función objetivo $f : \mathcal{X} \longrightarrow \mathcal{Y}$ (desconocida) es la que queremos aproximar, la cual a cada conjunto de atributos del cardiotocograma le asigna su estado correspondiente.

2.1. Visualización

Para poder tener una mejor comprensión del problema hemos decidido visualizar los datos de entrenamiento –no lo hemos hecho con los de *test* para no realizar *data snooping*– usando PCA y t-SNE.

- **PCA** (*Principal Component Analysis*) es una técnica que sirve para reducir el número de atributos, de forma que se queda con los atributos que puedan explicar el máximo de varianza.
- **t-SNE** (*t-Distributed Stochastic Neighbor Embedding*), por su parte, intenta reproducir la distribución que existe en el espacio original en otro espacio de dimensión menor, de forma que los puntos con características parecidas queden cerca en el modelo final y los que menos se parecen queden más alejados.

Pensamos que es importante visualizar los datos, pues así podemos ver cómo se reparten las clases y saber, en un principio, qué clases va a ser más fácil que confunda nuestro modelo, y en cuales la separación es más clara.



(a) t-SNE **(b)** PCA

Figura 1: Visualización de datos de *train*

Podemos ver que la mayoría de los datos corresponden a fetos normales. A continuación, vemos una clara franja de fetos patológicos y, por último, vemos cómo los fetos sospechosos se confunden muchas veces con las otras dos clases. La visualización que hemos obtenido tiene sentido con el problema que estamos resolviendo: es lógico que las clases correspondientes a los fetos sospechosos se confundan con los otros dos.

Aún así, podemos apreciar que la separación es destacable, por lo que podremos esperar que nuestro modelo tendrá unos resultados aceptables.

También hemos que tener en cuenta que, para poder visualizar los datos en 2 dimensiones, hemos tenido que reducir muchos atributos y hemos perdido información, por lo que es probable que realmente las clases estén más separadas.

3. Partición en *train* y *test*

La base de datos no traía conjuntos diferenciados de *train* y *test*, por lo que dividiremos el conjunto proporcionado en subconjuntos de *train* y *test* respecto a la proporción 20/80, acorde con lo recomendado en teoría.

Para ello hemos utilizado la función `train_test_split`, haciendo una mezcla previa de los datos, para evitar la presencia de sesgo porque estuviesen ordenados de alguna forma previamente.

Respecto a la selección de conjuntos de validación, dado que usaremos *K-fold cross validation* no será necesario especificarlos.

Para comprobar la distribución final de las etiquetas hemos visualizado un diagrama de barras, con el número de muestras correspondientes a cada etiqueta, pero solo con los datos de entrenamiento para no hacer *data snooping*.

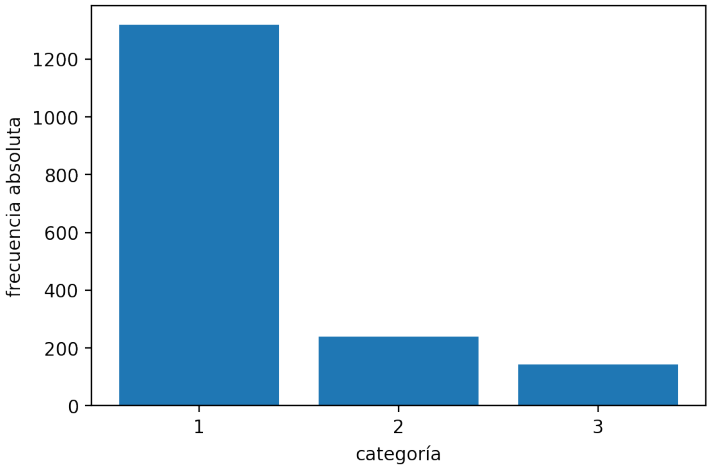


Figura 2: Distribución de las etiquetas en *train*

Vemos que la mayor parte de los datos pertenecen a la etiqueta 1, es decir, de fetos normales. Esto podría darnos problemas porque las etiquetas no están igualmente distribuidas – lo tendremos en cuenta a la hora de analizar los resultados. Creemos que la diferencia en las proporciones de una clase o de otra no se debe a que hayamos realizado la partición mal, pues en `train_test_split` explicitamos que se hiciera la división manteniendo las proporciones; sino que se debe más bien a las características del problema, ya que es más común que el feto esté sano a que sea patológico o sospechoso.

4. Preprocesado

A continuación, haremos un preprocesamiento de los datos, por pasos. Resumimos a continuación los pasos de preprocesado, y cómo actúan en nuestro modelo:

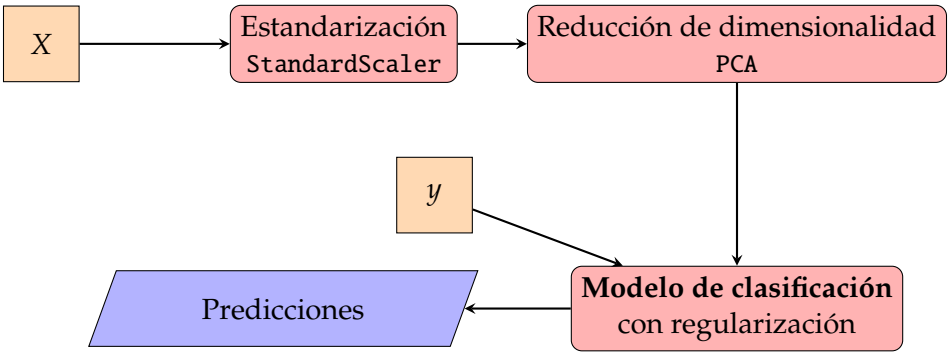


Figura 3: Diagrama de preprocesado para nuestro problema

Todos los pasos de preprocesado serán implementados mediante el uso de *pipelines* en `scikit-learn`. En concreto, el *pipeline* de preprocesado para este problema puede verse en la variable `preprocessing`.

Para poder evaluar la calidad de nuestro preprocesado, tendremos en cuenta la correlación de nuestros atributos. La **correlación** indica la fuerza y la dirección de una relación entre dos variables. Para medir el grado de correlación de las variables utilizaremos en este caso el coeficiente de correlación de Pearson (que nos indica dependencia directa si es cercano a 1, inversa si lo es a –1 y falta de dependencia si es próximo a 0).

Creemos que en cualquier problema de aprendizaje automático es deseable que las correlaciones entre los datos sean bajas, pues así no nos encontramos problemas de redundancia. Además, datos que tengan una correlación muy fuerte van a aportar la misma información al problema, por lo cual no nos interesa tenerlos.

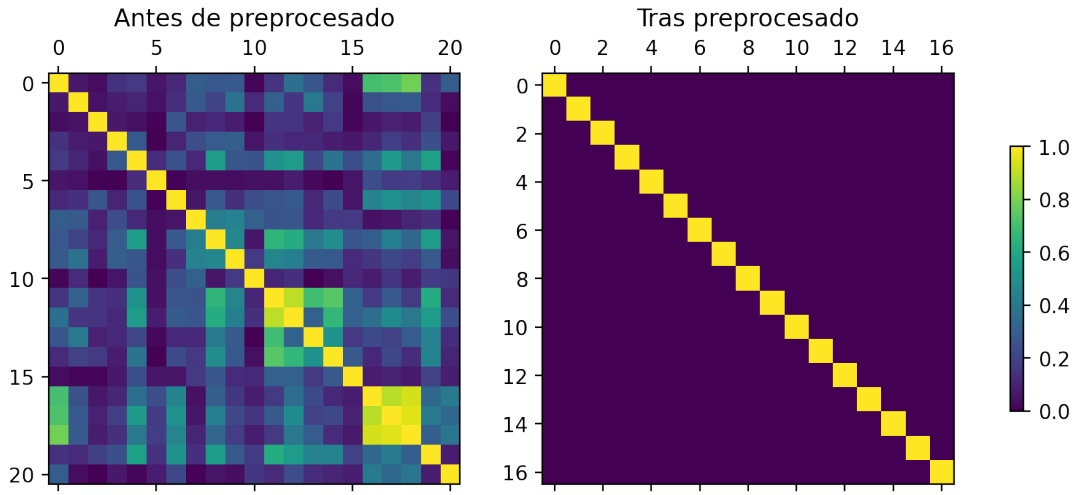


Figura 4: Matriz de correlación en *train* antes y después del preprocesado

Hemos visualizado la **matriz de correlación** antes y después del preprocesado, para tener una idea de si lo hemos llevado a cabo bien o no.

Podemos observar que antes del preprocesado existía poca correlación (algo más entre los atributos 16, 17 y 18 –correspondientes a la moda, media y mediana del pulso fetal, lo que puede ser debido a que dicha distribución sea simétrica–[2]), pero vemos claramente que después del preprocesado la poca correlación que había ha desaparecido.

Por tanto, podemos decir que el conjunto de atributos final es correcto, pues las variables son independientes y hemos conseguido realizar una selección que mantenga la mayor parte de la varianza.

Pasamos a especificar los pasos del preprocesado con más detalle.

4.1. Tratamiento de valores perdidos y *outliers*

En muchos conjuntos de datos es posible que algunos ejemplos no tengan algún que otro atributo, en cuyo caso sería necesario procesar los datos de una forma específica. En nuestro caso, vemos que no hay valores perdidos, por lo que no añadiremos una fase de preprocesado con el objetivo de tratar con éstos.

Estudiamos ahora la **variabilidad** de los datos. Esto lo hacemos pues entendemos que, una vez escalados los datos, puede darnos una idea de qué valores influyen más y cuales menos en la asignación de las etiquetas, pues un atributo que tenga varianza casi nula nos podrá aportar muy poca información, ya que tendrá prácticamente el mismo valor para todos los datos.

Para ello hemos dibujado un *boxplot* o diagrama de caja, que es un método para representar datos a partir de sus cuartiles. De esta forma podemos rápidamente visualizar los valores extremos -que pueden ser valores atípicos- y las diferencias entre las varianzas de los atributos.

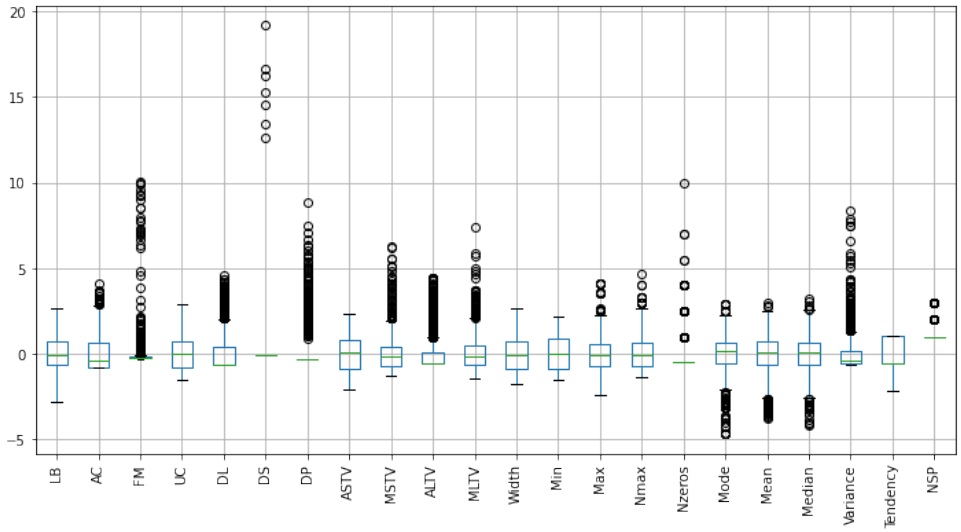


Figura 5: Boxplot de los datos de *train*, escalados

[2]Pildner von Steinburg, S., Boulesteix, A. L., Lederer, C., Grunow, S., Schiermeier, S., Hatzmann, W., Schneider, K. T. M., Daumer, M. (2013). What is the "normal" fetal heart rate? *PeerJ*, *1*, e82. https://doi.org/10.7717/peerj.82

Vemos que no hay ningún dato que tenga muy poca varianza, por lo que es probable que al aplicar PCA, como haremos a continuación, no eliminemos muchos atributos.

Por otra parte, respecto a la presencia de *outliers*, podríamos encargarnos de detectarlos y eventualmente eliminarlos, pero no lo hemos hecho dado que desconocemos si esos puntos pueden tener información valiosa para nuestro problema. Lo ideal sería consultar con un experto si es posible esta eliminación.

4.2. Estandardización

La estandardización es el reescalado de los datos para que tengan las propiedades de una distribución normal estándar (de media nula y desviación estándar unidad, $\mathcal{N}(0,1)$). Este paso es necesario porque muchos algoritmos requieren que las características estén normalizadas para un funcionamiento adecuado. De hecho, la estandardización es especialmente relevante en el caso de PCA. En PCA, nos interesan los componentes que maximizan la varianza. Si un componente varía menos que otro por sus escalas respectivas, es posible que PCA determine que la dirección de mayor varianza corresponde con los que más varíen, si no se escalan las varianzas.[3]

Para ello hemos utilizado la función `StandardScaler`: hemos escalado con los datos de entrenamiento y luego lo hemos aplicado a los de test. De esta forma conseguimos que todos los datos tengan media cero y varianza uno, pero sin hacer *data snooping*, ya que escalamos con los valores obtenidos de los datos de entrenamiento.

4.3. Reducción de dimensionalidad

Creemos que es importante reducir la dimensionalidad del problema, para de esta forma quedarnos con los atributos que realmente son determinantes. La reducción de dimensionalidad tiene algunas ventajas:

- Reducir el *overfitting*: datos menos redundantes conducen a menos oportunidad para que nuestro modelo tome decisiones basadas en el ruido.
- Reducir el tiempo de entrenamiento.
- Aumentar la precisión, ya que usamos datos menos "engañosos".

Para nuestro problema, utilizaremos el PCA (*Principal Component Analysis*), una técnica de selección no supervisada, que considera combinaciones lineales de los atributos (componentes principales) que permitan recoger la mayor varianza de los datos de acuerdo a diversas proyecciones. Realizamos PCA indicando que la varianza acumulada de las variables tras la aplicación del algoritmo debe ser un 99 % de la original. De este modo, pasamos a tener 17 atributos (de los 21 atributos que teníamos inicialmente).

La elección del porcentaje de varianza del PCA depende de cómo de agresiva queremos que sea la reducción: cuanto menor sea el valor, más agresiva será. Un porcentaje del 99 % es una

[3] *Importance of feature scaling.* Scikit-Learn 0.24.2 Documentation. Recuperado 13 de junio de 2021, de https://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html

reducción muy poco agresiva, ya que creemos que tenemos pocos atributos y además hemos visto que no están demasiado correlados, ni hay atributos con varianza nula.

5. Métrica de error a usar

Hemos decidido elegir la métrica *accuracy*, es decir, la fracción de predicciones que el modelo ha realizado correctamente. La hemos elegido frente a precisión, sensibilidad o especificidad porque entendemos que lo que queremos es resolver un problema para el que no disponemos de ninguna solución, por tanto lo deseable es que nuestro modelo acierte lo máximo posible.

Además, la precisión es la métrica que se utiliza en la mayoría de problemas de clasificación, lo que nos da un valor para comparar nuestros resultados con otros que se hayan podido obtener. El *accuracy* se calcula como sigue:

$$\text{Acc}(h) = \frac{1}{N}[\![h(x_n) = y_n]\!] \quad \forall h \in \mathcal{H}$$

6. Regularización

Como hemos comentado anteriormente, usaremos regularización para evitar el *overfitting* (de hecho usaremos modelos muy propensos a este fenómeno), mediante la limitación de la complejidad de nuestros modelos. Utilizaremos dos regularizadores:

- **Regularización L_2 (*ridge*).** Añadimos a la función de pérdida una penalización proporcional al cuadrado de los pesos:

  $$L_{reg}(w) = L(w) + \lambda \|w\|_2^2$$

  Esta regularización favorece que los coeficientes de los atributos sean bajos. Es especialmente útil cuando la mayor parte de los atributos son relevantes, algo muy interesante en nuestro caso, pues hemos preprocesado los datos precisamente para esto.

- **Regularización L_1 (*lasso*).** La penalización es, en este caso, lineal:

  $$L_{reg}(w) = L(w) + \lambda \sum_i w_i$$

  La regularización *lasso* favorece que algunos de los coeficientes valgan cero. Funciona mejor si los atributos no están correlados entre sí.

Como en el preprocesamiento de datos que hemos realizado hemos intentado eliminar la irrelevancia y la correlación hemos decidido probar con los dos tipos de regularización en la mayoría de los modelos.

7. Elección de los modelos a usar

Como comentamos al principio de la práctica, hemos decidido comparar **regresión logística** con otros modelos no lineales como **perceptrón multicapa**, **SVM** y ***random forest***.

Antes de proceder a especificar los modelos, debemos tener en cuenta que el espacio de características que usaremos en nuestros algoritmos no será $\mathcal{X}$ (un espacio con dimensión $d = 21$), sino $\mathcal{X}'$, el espacio de dimensionalidad reducida obtenido tras el preprocesamiento, con dimensión $d' \leq d = 21$ (tras aplicar el preprocesamiento, tenemos $d' = 17$).

Por otra parte, fijaremos una serie de *hiperparámetros* que ajustaremos mediante *cross-validation*, como veremos en el siguiente apartado. Además de estos hiperparámetros, especificaremos una serie de *parámetros* (fijos en cada *fold*). Un parámetro que mantendremos constante en todos los modelos será el máximo de iteraciones (`max_iter`), que fijaremos a 10 000 (excepto cuando explicitemos lo contrario).

7.1. Regresión logística

Como modelo lineal nos hemos decantado por la **regresión logística**, dado que es el mejor modelo lineal para clasificación multietiqueta de los introducidos en clase. Especificaremos a continuación los parámetros e hiperparámetros usados, pero haremos una introducción en primer lugar a la clase de funciones $\mathcal{H}$ que hemos elegido al decantarnos por este modelo.

Al estar en un problema de clasificación multiclase, usaremos la técnica *one-vs-all*. Tendremos por tanto un clasificador binario para cada $i \in \mathcal{Y} = \{1, 2, 3\}$ en $\mathcal{X}'$:

$$h_i(x) = w^T x, \quad w \in \mathbb{R}^{d'}$$

Este clasificador, salvo normalización con $\frac{1}{\|w\|}$, mide la distancia con signo de x al hiperplano definido por w. Si esta distancia es positiva, etiquetamos en favor de la clase i. Para predecir la clase a asignar a una instancia $x \in \mathcal{X}'$, le asignaremos la clase que tenga una clasificación positiva más fuerte, o la que esté más cerca de la frontera de clasificación, es decir, le asignamos la clase

$$\arg\max_{i \in \mathcal{Y}} h_i(x)$$

Obteniendo la clase de funciones para nuestro problema:

$$\mathcal{H} = \left\{ \arg\max_i w_{(i)}^T x : w_{(i)} \in \mathbb{R}^{d'}, i \in \mathcal{Y} \right\}$$

En definitiva, tendremos un modelo con un total de $|\mathcal{Y}| \cdot d' = 3d'$ parámetros (un clasificador binario por clase).

Como función de pérdida en regresión logística usaremos la pérdida logarítmica:

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

A la que sumaremos la pérdida asociada al regularizador. Debemos tener en cuenta que, aunque esta función de pérdida sea la que optimiza el algoritmo de regresión logística, la métrica de error que usamos es en todo momento el *accuracy*, y será la que usaremos para *cross-validation*.

Hemos usado la implementación de `sklearn.linear_model.SGDClassifier`[4] con los siguientes parámetros e hiperparámetros:

[4] *sklearn.linear_model.SGDClassifier*. Scikit-Learn 0.24.2 Documentation. Recuperado 13 de junio de 2021, de https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

- **Función de pérdida (`loss`):** fijada en `log`.

  Implementa regresión logística con gradiente descendiente estocástico.

- **Tasa de aprendizaje (`learning_rate`):** fijado en `optimal` (valor por defecto).

  Tasa de aprendizaje en SGD. `optimal` la fija en $\eta = 1/(\alpha * (t + t_0))$ (el t_0 es escogido mediante una heurística específica).

- **Término de regularización (`penalty`):** a escoger entre {`l1`, `l2`}.

  Término de regularización. La regularización `l1` corresponde con la regularización ridge (L_1) y la regularización `l2` corresponde con la regularización lasso (L_2), ambas explicadas en el apartado anterior.

- **Constante de regularización (`alpha`):** a elegir entre 10 valores espaciados logarítmicamente entre 10^{-5} y 10^{-1}.

  Constante que multiplica el efecto de la regularización. Si el valor es más alto la regularización es más intensa. Hemos decidido probar entre esos valores pues son parecidos a los de por defecto y, además, al no haber obtenido sobreajuste al hacer *cross-validation* no creemos que sea necesario aplicar una regularización más intensa.

7.2. Perceptrón multicapa

Ajustaremos un modelo de **perceptrón multicapa**. Usaremos este algoritmo porque tiene la complejidad suficiente para obtener un sesgo bajo, y porque es idóneo para la clasificación no binaria, pues tiene capacidad para resolver problemas no linealmente separables, al contrario que el algoritmo perceptrón.

Procederemos del mismo modo a especificar $\mathcal{H}$. Usaremos un MLP de tres capas: la capa de entrada, dos capas ocultas y una de salida. Como función de activación en cada neurona usaremos $\theta = \tanh$, teniendo por tanto

$$\mathcal{H} = \left\{ \underset{i \in \mathcal{Y}}{\arg\max} \left[W_{(3)}^T \theta \left(W_{(2)}^T \theta \left(W_{(1)}^T x \right) \right) \right]_i : W_{(j)} \in M_{n_{j-1} \times n_j}(\mathbb{R}), n_0 = d', n_3 = 3 \right\}$$

Para el ajuste de los pesos usaremos *backpropagation* con el error cuadrático entre predicciones y etiquetas:

$$L_{cuad}(y_n, h(x_n)) = (y_n - h(x_n))^2$$

Hemos usado la implementación de `sklearn.neural_network.MLPClassifier`,[5] usando los siguientes parámetros e hiperparámetros:

- **Tasa de aprendizaje (`learning_rate`)**: fijado en `adaptive`.

  Una tasa de aprendizaje `adaptive` es dinámica, es decir, tiene un valor inicial por defecto, que se mantendrá siempre que vayamos en la dirección adecuada. Sin embargo si alcanzamos un número determinado de iteraciones sin hacer que el error vaya a menos reduce su valor. Esto se hace para que el algoritmo avance de forma más lenta conforme se acerque a la solución, pero tiene la desventaja de que puede caer en mínimos locales.

[5] *sklearn.neural_network.MLPClassifier.* Scikit-Learn 0.24.2 Documentation. Recuperado 13 de junio de 2021, de https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

- **Función de activación (`activation`):** fijado en `tanh`.

  Función de activación para las etiquetas ocultas. Como hemos especificado anteriormente, $\theta = \tanh$. Lo hemos decidido así porque ha sido la función estudiada en clase y además tiene como ventaja que devuelve una probabilidad, y no un valor discreto 0 ó 1.

- **Método de resolución (`solver`):** fijado en `sgd`.

  Método de resolución que utiliza el algoritmo. Hemos usado SGD pues ha sido el método principal visto en clase, y el que hemos utilizado en las prácticas.

- **Tamaño de las capas ocultas (`hidden_layer_sizes`):** a escoger entre $\{40, 50, 60\}$.

  Representa el número de nodos en cada capa oculta de la red neuronal (según nuestra notación, el valor de n_1 y de n_2). Hemos puesto entre 40 y 60 pues, aunque en el guión de prácticas se recomienda entre 50 y 100 pensamos que nuestro problema tiene muy pocos atributos y pocas instancias, por lo que no será necesario tener tantos nodos, y además podemos caer en sobreajuste.

- **Término de regularización:** fijamos regularización L_2 (la que usa el algoritmo implementado por defecto).

- **Constante de regularización (`alpha`):** a escoger entre 5 valores espaciados logarítmicamente entre 10^{-3} y 10.

  Si el valor es más alto la regularización es más alta. Hemos decidido probar entre esos valores pues son parecidos a los que están por defecto, y además al hacer *cross-validation* no hemos obtenido sobreajuste, y por tanto no creemos que sea necesario aplicar una regularización más estricta.

- **Máximo de iteraciones (`max_iter`):** fijado en 2 000.

  Dada la complejidad en tiempo del algoritmo MLP, decidimos reducir el máximo de iteraciones. Dado que emplea SGD, tampoco es necesario un número excesivo de iteraciones para optimizar la solución (llegará a un óptimo local mucho antes).

7.3. Support Vector Machine

También ajustaremos un modelo de **Support Vector Machine**, que separa las clases mediante hiperplanos de forma que la distancia entre éste y los puntos de cada clase sea la máxima posible. Hemos escogido este modelo por varios factores, entre los que se encuentran los mencionados anteriormente para MLP: es un modelo ampliamente usado en clasificación no binaria, y que es capaz de obtener resultados con poco *overfitting* (notablemente inferior al caso de MLP). Pero, además, el SVM es mucho más eficiente en tiempo y memoria mediante el uso de *kernels*.

En este caso, usaremos el *kernel* RBF (*Radial Basis Funcion*), que se define del siguiente modo:

$$K_{RBF}(x_1, x_2) = \exp\left(-\gamma \|x_1 - x_2\|^2\right) \quad \forall x_1, x_2 \in \mathcal{X}'$$

El hiperparámetro γ se usa para configurar la sensitividad de las diferencias en los vectores de características: si hacemos que γ sea muy grande, entramos en *overfitting* (cuando $\gamma \to \infty$ la matriz del kernel se convierte en la matriz identidad, teniendo un ajuste perfecto).

En este modelo, usamos `sklearn.svm.SVC`.[6] De nuevo, especificamos los parámetros e hiperparámetros explorados:

- ***Kernel* (`kernel`):** fijado en `rbf`.
- **Término de regularización:** fijaremos regularización L_2 (es la diseñada para SVM).
- **Constante de regularización (C):** a escoger entre 10 valores espaciados logarítmicamente entre 10^0 y 10^4.

  Esta constante es inversamente proporcional a la fuerza de la regularización (es por ello por lo que los exponentes son positivos).
- **γ en *kernel* RBF (`gamma`):** a elegir entre `auto` y `scale`.

  Este γ es el que aparece en la fórmula de K_{RBF} que hemos explicitado anteriormente. Si se fija en `scale`, asignamos $\gamma = 1/(d' \cdot \text{varianza}(X_{test}))$. Si por otro lado lo fijamos en `auto`, tenemos $\gamma = 1/d'$. Recordemos que d' es el número de atributos del espacio preprocesado.

7.4. Random Forest

Random Forest es una combinación de árboles predictores de forma que cada árbol depende de los valores de un vector aleatorio probado independientemente y con la misma distribución para cada uno de ellos.

Para este modelo usaremos `sklearn.ensemble.RandomForestClassifier`[7] con los siguientes parámetros e hiperparámetros:

- **Número de estimadores** (`n_estimators`): a elegir entre 50, 100, 200, 400 y 600.

  Número de árboles que vamos a utilizar.
- **Profundidad máxima** (`max_depth`): a elegir entre 5, 10, 15 y 20.

  Profundidad máxima que pueden tener los árboles.
- **Función de calidad de división** (`criterion`): `gini`.

8. Análisis de resultados

8.1. Selección del mejor modelo

Para elegir el modelo que mejor se ajusta a nuestro problema hemos empleado ***K-fold cross-validation***. El *cross-validation* es una técnica utilizada para garantizar que los resultados que obtenemos al entrenar los datos son independientes de la partición previa que hayamos hecho. Para ello, dividimos el conjunto de entrenamiento en K conjuntos con la misma cantidad de

[6] *sklearn.svm.SVC.* Scikit-Learn 0.24.2 Documentation. Recuperado 13 de junio de 2021, de https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

[7] *sklearn.ensemble.RandomForestClassifier.* Scikit-Learn 0.24.2 Documentation. Recuperado 13 de junio de 2021, de https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

datos, y realizando K entrenamientos, de modo que en cada uno de ellos el modelo entrena los datos de los $K-1$ conjuntos y los evalúa en el conjunto sobrante. A continuación, calculamos la media de los errores a lo largo de todos los conjuntos de validación y escogemos el modelo de error menor. Este error, el error de cross-validation, E_{cv}, es un buen estimador de E_{out}, por lo que nos servirá para escoger el mejor modelo en términos de generalización y error. De esta forma conseguimos asegurar la independencia de la elección del mejor modelo y la partición, pero sin caer en el *data snooping*.

Cada vez que seleccionamos un modelo, también estamos seleccionando sus *hiperparámetros*. De este modo, al emplear la técnica de validación, obtendremos el mejor modelo con los mejores parámetros. Además, y con el objetivo de eliminar el *data snooping*, realizaremos todas las fases de preprocesado y selección de modelos de una vez.

Para hacer todo esto, usamos los *pipelines* (`sklearn.pipeline.`**`Pipeline`**[8]) y la función `sklearn.model_selection.`**`GridSearchCV`**[9] de `scikit-learn`, a la que le pasamos el conjunto de modelos e hiperparámetros que usamos (*search space*) y la métrica de error a usar (en nuestro caso, *accuray*), así como K (el número de *folds*, 5 en nuestro caso). `GridSearchCV` nos permite conocer qué modelo obtiene mejores resultados en *cross-validation*.

Para profundizar en cada uno de los modelos, hemos usado `GridSearchCV` en cada uno de ellos. Una vez terminados todos los *fits*, obtenemos los siguientes mejores hiperparámetros:

	Parámetro	Mejor valor
Regresión logística	`penalty`	L_2
	`alpha`	$1.6681 \cdot 10^{-3}$
SVM-RBF	`gamma`	`scale`
	`C`	166.8101
Perceptrón multicapa	`hidden_layer_sizes`	40
	`alpha`	10^{-2}
Random Forest	`max_depth`	15
	`n_estimators`	400

Después de tener el modelo elegido entrenamos con la muestra de entrenamiento completa porque, al tener más datos, el modelo que obtengamos será mejor. Por último aplicamos los resultados obtenidos al conjunto de *test*, y comprobamos la predicción de las etiquetas con las etiquetas reales. De este modo obtenemos los siguientes errores en cada modelo:

	Acc_{cv} (%)	Acc_{train} (%)	Acc_{test} (%)	Tiempo* (s)
Regresión logística	89.9412	90.1765	88.7324	4.6314
SVM-RBF	93.0588	98.8824	93.4272	2.7153
Perceptrón multicapa	90.0588	91.0588	90.1408	87.9395
Random Forest	90.8235	99.9412	91.3146	33.3037

* Tiempo de ejecución de todos los *fits* en *cross-validation*

Podemos ver que el modelo que mejores resultados ha obtenido ha sido **SVM**, y en un tiempo de ejecución excepcional. Sin embargo, el resto de modelos también han proporcionado resultados

[8] *sklearn.pipeline.Pipeline.* Scikit-Learn 0.24.2 Documentation. Recuperado 13 de junio de 2021, de https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

[9] *sklearn.model_selection.GridSearchCV.* Scikit-Learn 0.24.2 Documentation. Recuperado 13 de junio de 2021, de https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

muy buenos, incluso el lineal.

Profundicemos en el desempeño de nuestro mejor modelo. Para ello, veremos las **matrices de confusión** en *train* y en *test*. Estas matrices cuentan el número de datos de una cierta clase que el modelo ha clasificado en otra.

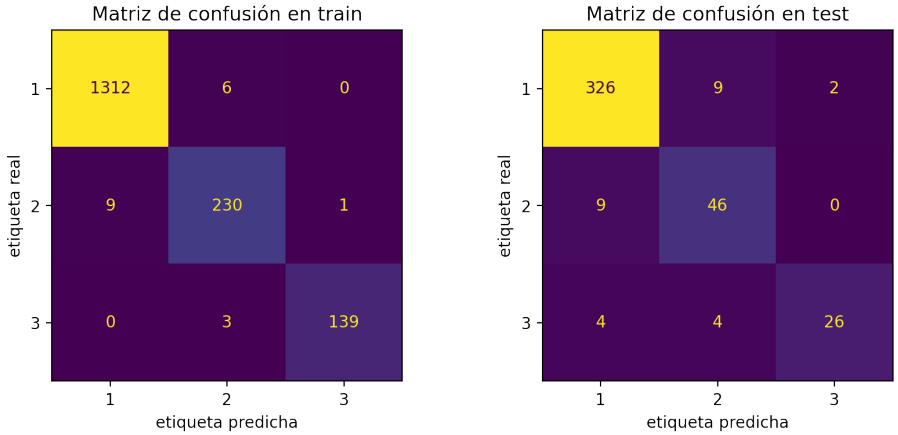


Figura 6: Matrices de confusión en *train* y en *test* para SVM

De la matriz de confusión podemos extraer ciertas observaciones interesantes:

- El modelo se confunde más clasificando fetos sospechosos que normales y patológicos. Esto es algo esperable, dado que la característica "sospechoso" está más mezclada entre las otras dos.
- El modelo tiene un cierto sesgo hacia clasificar los fetos como sanos. Esto puede ser una consecuencia de que, como vimos, disponemos de más datos de fetos sanos que de fetos patológicos o sospechosos.

8.2. Estimación de E_{out}

Sabemos que el error de *cross-validation* es buen estimador del error fuera de la muestra. Sin embargo, la mejor cota que podemos obtener en función de la desigualdad de Hoeffding la obtendremos a partir de E_{test} –tengamos en cuenta que no hemos usado el conjunto de *test* en ninguna fase de entrenamiento y selección–. Dicha cota queda como sigue:

$$E_{out}(g) \leq E_{test}(g) + \sqrt{\frac{1}{2N_{test}} \log \frac{2}{\delta}} \text{ con probabilidad } \geq 1 - \delta$$

Tomando, por ejemplo, $\delta = 0.05$ podemos garantizar al 95 % de confianza que

$$E_{out}(g) \leq 0.065728 + \sqrt{\frac{1}{2 \cdot 426} \log \frac{2}{0.05}} = 0.109091$$

En otras palabras, podemos decir que el modelo que hemos obtenido tiene un error del 10.9091 % (equivalentemente, un *accuracy* del 89.0909 %) al 95 % de confianza.

9. Conclusiones

Para evaluar el desempeño de nuestro modelo, analizaremos las curvas de aprendizaje, escalabilidad y rendimiento. Concretemos en primer lugar qué es cada una de ellas:

- La **curva de aprendizaje** presenta la evolución del *accuracy* conforme aumentamos el número de ejemplos.
- La **curva de escalabilidad** mide el tiempo de entrenamiento empleado conforme aumentamos el número de ejemplos.
- La **curva de rendimiento** muestra cuánto tiempo fue requerido para entrenar los modelos para cada tamaño de entrenamiento.

Representaremos y analizaremos estas curvas para el mejor modelo obtenido, SVM.

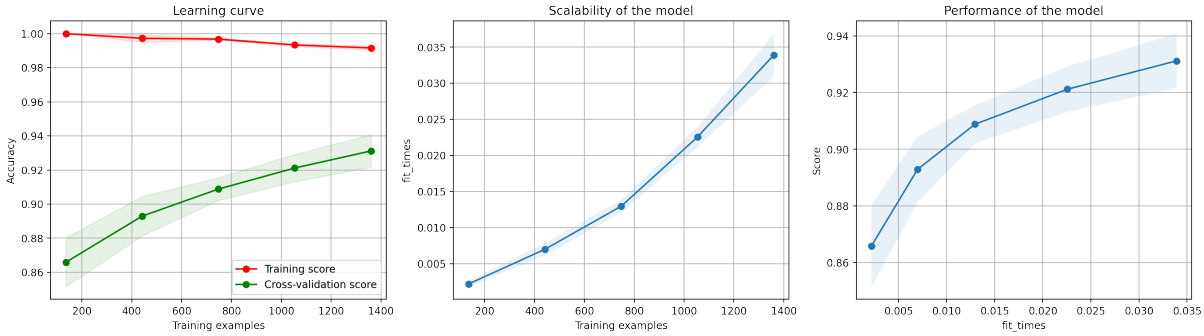


Figura 7: Curvas de aprendizaje, escalabilidad y rendimiento para el mejor modelo (SVM)

En primer lugar, podemos ver cómo la curva de aprendizaje tiene el comportamiento esperado: para pocos ejemplos, el modelo conseguirá ajustar los datos perfectamente (por eso el *training score* es del 100 %), mientras que el *accuracy* en *cross-validation* no es tan bueno (recordemos que E_{cv} es proxy de E_{out}), pues no tenemos ejemplos suficientes para obtener una buena generalización. Conforme aumentamos el número de ejemplos, la tendencia cambia: el *training score* decrementa, pues el modelo regularizado no podrá ajustar los datos perfectamente (evitando el *overfitting*); mientras que el *cross-validation score* aumenta, pues tenemos más ejemplos y de este modo podremos clasificar mejor nuevos datos. Observemos cómo la curva de *cross-validation score* tiene tendencia logarítmica, pero podemos intuir que la precisión podrá aumentar aún más con más ejemplos. Es decir, nuestro modelo tiene una buena generalización, y podría obtener un mejor desempeño con más datos.

En el caso de las otras dos curvas, también vemos un comportamiento esperable. En la primera, vemos cómo los tiempos de ejecución irán aumentando cada vez más al tener más ejemplos. Por otro lado, vemos cómo el rendimiento del modelo sigue una tendencia logarítmica: llegará un momento en el que no podremos obtener una mejora significativa de los resultados con incrementos de tamaños de entrenamiento.

Como resumen comentar que creemos que nuestro modelo a grandes rasgos predice bien la clasificación de los fetos, aunque podría obtener mejores resultados si dispusiéramos de una mayor cantidad de datos. Aún así creemos que los atributos que han sido seleccionados para hacer mediciones son los correctos, porque con pocos datos hemos conseguido resultados aceptables.

Por último, en relación a la comparación entre el modelo lineal (regresión logística) y los no lineales, podemos observar que los resultados obtenidos han sido muy parecidos – algo peores en regresión logística que en los no lineales, pero también, junto con MLP, el que menor sobreajuste ha obtenido. El modelo que elegiríamos finalmente sería SVM, ya que es el que mejor resultado nos ha dado en *cross validation*, y además el que menor tiempo de ejecución ha tenido.