Universidad Complutense de Madrid

Facultad de Informática

Doble Grado en Ingeniería Informática y Matemáticas

Memoria del proyecto final

Aprendizaje Automático y Big Data

Autores:

Juan Carlos Villanueva Quirós
Jorge Villarrubia Elvira

Grupo 14

4 de julio de 2021

Índice

1. Análisis y preparación de los datos

Este proyecto consiste en el tratamiento, mediante diversas técnicas de Aprendizaje Automático, de un dataset de la plataforma *Kaggle*, utilizando para ello el código desarrollado en las prácticas de la asignatura.

El dataset escogido para el desarrollo del proyecto puede encontrarse en https://www.kaggle.com/teertha/personal-loan-modeling y consiste en 5.000 ejemplos de personas que aceptaron o no un préstamo bancario junto a 12 variables características tales como su edad, ingresos o experiencia laboral. La finalidad del trabajo es desarrollar modelos que pudieran predecir si una persona aceptará o no un préstamo en base a las 12 características mencionadas, utilizando el dataset para su entrenamiento, validación y testing.

Para comenzar, comprobamos que no había ningún valor nulo en el dataset, aunque esto ya se anunciaba en la descripción de *Kaggle*.

Seguidamente visualizamos la distribuciones de los datos que, como también indicaba la descripción del dataset, estaban ya todos en formato numérico.

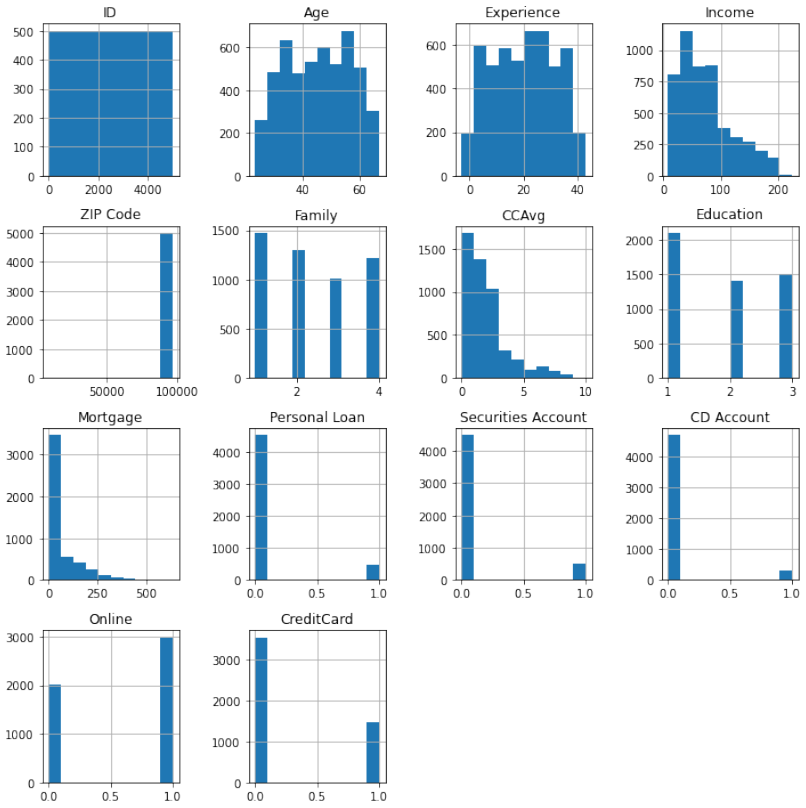

Figura 1.1: Distribuciones de las distintas variables del dataset.

La columna *ID* no se corresponde con una variable auténtica pues simplemente indica el número de fila de cada ejemplo y por eso fue eliminada.

La variable *ZIP Code* debía tener algún tipo de dato desviado pues su distribución ser representaba como una única barra. Al estudiarla vimos que había un código postal con valor 9307 cuando todos los demás estaban entre 90000 y 10000. Buscando por internet comprobamos que efectivamente era

un error, dado que ese valor no es el código postal de ningún condado de California como lo eran todos los demás (ni siquiera era un código postal de EEUU). Supusimos que faltaba alguna cifra y valoramos eliminar el ejemplo o modificarlo asignándole un valor razonable. Podíamos darle el valor medio del resto de ejemplos, pero como tan solo sucedía para este caso decidimos simplemente eliminarlo.

Tras eliminarlo, la distribución de la variable *ZIP Code* resultó ser la siguiente.

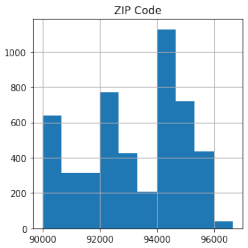

Figura 1.2: Distribución de *ZIP Code* tras eliminar el valor erróneo.

Otro aspecto que nos llamó la atención es que hubiese algunos valores negativos en la variable *Experience*. En otras soluciones al problema, que pueden verse en *Kaggle*, se dan cuenta de ello y asumen que se trata de un error de signo, tomando el valor absoluto de estos datos para solucionarlo. Como esta variable resultó estar profundamente correlacionada con *Age*, lo que supuso que la eliminaríamos, estos errores de signo fueron irrelevantes y no hubo que tratarlos.

Visualizamos los coeficientes de correlación entre todas la variables (en particular con la clase objetivo *Personal Loan*), para intuir cuales podían ser más importantes y eliminar variables redundantes.

Figura 1.3: Coeficientes de correlación entre variables.

Observamos una enorme correlación directa entre las variables *Age* y *Experience* (coeficiente 0.99). Esto nos indica que ambas aportan prácticamente la misma información y, por tanto, es posible y deseable eliminar una de ellas. Como el único motivo para elegir una u otra era el tema de los valores negativos en *Experience*, se decidió eliminar esa variable dejando la columna *Age*.

Además, surgió la tentación de eliminar las múltiples variables que tenían un coeficiente de correlación ridículamente pequeño con la clase objetivo *Personal Loan* (inferior a 0.05 en valor absoluto). Sin embargo, que no haya una correlación en este sentido significa que no son buenas para predecir por sí solas el objetivo, pero podrían ser útiles de manera combinada. En este sentido, no hay motivos suficientes para quitarlas.

Otro asunto que tratamos en este momento fue la discretización de valores. Parecía interesante discretizar la variable *Age*, pues para el tema de la edad quizás sea mejor generalizar. Seguramente no sea tan importante que una persona tenga 50 o 52 años, sino el grupo de edad al que pertenezca (joven, mediana edad o persona mayor). Observando la distribución de la variable y sus datos estadísticos (la edad más baja era 23 y la más alta 67) se decidió calcular los terciles: el primer tercil era de 39 años y el segundo tercil era de 52 años. Estos terciles se usaron como frontera para que, al separar en los 3 grupos, quedase la misma cantidad de ejemplos en cada uno.

A cada grupo se le dio el valor correspondiente a su edad mínima, quedando de la siguiente forma:

1. **Grupo joven:** personas hasta 39 años (todas quedan con valor 23 en la variable).
2. **Grupo de mediana edad:** personas de entre 39 y 52 años (todas quedan con valor 39 en la variable).
3. **Grupo de personas mayores:** personas a partir de 52 años (todas quedan con valor 52 en la variable).

Por otra parte, un aspecto absolutamente clave de cara a la partición de los datos y para la evaluación de nuestros modelos es que el dataset estaba profundamente desbalanceado en cuanto a cantidad de casos de la clase objetivo. En la distribución de la variable *Personal Loan* de la Figura 1.1 puede apreciarse este hecho. Más concretamente se obtuvo el dato de que el 90.4 % de los ejemplos eran de la clase 0 (personas que no aceptaron el crédito) y apenas el 9.6 % restante eran de la clase 1 (personas que aceptaron el crédito).

Figura 1.4: Proporción de casos en la variable objetivo *Personal Loan*.

Esto supone que hay que tener mucho cuidado para no generar modelos que parezcan funcionar bien debido a una clara tendencia a responder la clase mayoritaria. Estos modelos serían inútiles y

podrían alcanzar más de un 90 % de *accuracy* simplemente respondiendo siempre la clase 0. Para evitarlo, evaluamos nuestros modelos fijándonos en la *precisión* (proporción de predicciones de clase 1 que son acertadas) y el *recall* (proporción de casos reales de la clase 1 que son identificados). Como esto supone atender a dos métricas distintas, de manera frecuente echamos mano de la *f1* como media armónica de la *precisión* y el *recall* (es la medida en que nos fijamos especialmente para la elección de los hiperparámetros, aunque siempre visualizamos simultáneamente la *precisión* y el *recall*).

También por este motivo hicimos que la partición del dataset fuese "estratificada" según *Personal Loan* (respetando las proporciones para la clase objetivo de la Figura 1.4 en cada conjunto), tratando así de no tener un desbalanceo aún mayor al entrenar, validar o testear. Se decidió dividir nuestros 4.999 ejemplos en 3 conjuntos (entrenamiento, validación test) para tener un conjunto final de test, independiente del entrenamiento y de la elección de los hiperparámetros sobre el que poder corroborar los resultados de los modelos. La partición realizada fue: 60 % de los ejemplos para el conjunto de entrenamiento, 20 % de los ejemplos para el conjunto de validación y el 20 % restante para el conjunto de test.

Como paso final en la preparación de los datos se estandarizaron las variables[1]. Esto no es estrictamente necesario para los modelos que nosotros hemos utilizado (la regresión logística o las redes neuronales ajustarán sus pesos correctamente en función de la escala que tengan los datos), pero en la convergencia de los modelos suele ser más rápida cuando las variable toman valores pequeños (próximos a 0). En la Figura 1.1 con las distribuciones, puede comprobarse cómo las distintas variables se mueven en escalas muy diferentes, tomando algunas de ellas valores bastante grandes (por ejemplo *ZIP Code* o *Income*).

Un aspecto fundamental al escalar cada variable es hacerlo por separado para cada uno de los conjuntos de la partición. Por ejemplo, si los datos de entrenamiento fuesen tenidos en cuenta a la hora de calcular la media y la desviación típica para normalizar el valor de una variable correspondiente a un dato de validación o test, esta dato estaría contaminado por información de entrenamiento. Puesto que queremos mantener la independencia de los conjuntos la estandarización se hace del siguiente modo:

$$X_j^{train} \leftarrow \frac{X_j^{train} - \mu_j^{train}}{\sigma_j^{train}} \qquad X_j^{eval} \leftarrow \frac{X_j^{eval} - \mu_j^{eval}}{\sigma_j^{eval}} \qquad X_j^{test} \leftarrow \frac{X_j^{test} - \mu_j^{test}}{\sigma_j^{test}}$$

siendo j el índice que denota la variable.

[1] En clase la llamamos *normalization* pero puede ser confuso con el reescalado que consiste en dividir entre la norma.

2. Modelos de regresión logística

Después de limpiar y preprocesar los datos, ya estamos en condiciones para aplicar modelos de predicción sobre ellos. Comenzaremos por la regresión logística. En particular, aplicaremos dos tipos de regresión logística:

- Hipótesis lineal: Tomamos como atributos las categorías de entrada, lo que nos dará una frontera de decisión lineal (un hiperplano).
- Hipótesis no lineal: Tomamos como atributos una combinación polinómica de las categorías de entrada, con lo que conseguiremos una frontera de decisión no lineal.

Para implementar y aplicar la regresión, hemos extraído y adaptado muchas funciones de la práctica de regresión logística (Práctica 2). En particular, hemos cogido la función `sigmoide`, `costeRegu` y `gradienteRegu`, que usamos para el entrenamiento con regularización de los clasificadores.

También, hemos reusado la función `testLamda` de la práctica 2 y la hemos adaptado a `testParam` para que funcionase con cualquier parámetro y con cualquier modelo de predicción. La función `testLamda` variaba el parámetro de regularización λ, entrenando un modelo de regresión distinto para cada valor del parámetro, lo que nos servía para decidir qué λ escoger prediciendo sobre le conjunto de validación.

Con el propósito de generalizar y aprovechar esta función lo máximo posible, en `testParam` hemos pasado como argumentos el rango del parámetro y el nombre del parámetro que queremos variar. Además, también pasamos el nombre del modelo para el cual variar el parámetro. De esta manera, hemos generalizado la función de forma que podemos aprovecharla para todos los modelos de predicción que vamos a usar más adelante. Como vemos, afortunadamente hemos podido aprovechar mucho código de las prácticas anteriores.

Por último, hemos implementado también las funciones:

- `metricas`: dada las predicciones y los valores reales, calcula las métricas que usamos (*accuracy*, *precision*, *recall* y f_1).
- `muestraMetricas`: dada las predicciones y los valores reales, llama a metricas y las muestra por pantalla.
- `plot_confusion_matrix`: dada las predicciones y los valores reales, muestra por pantalla la matriz de confusión. Esta función la hemos extraído de https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html.

Después de implementar todas las funciones que necesitábamos, nos dispusimos a entrenar modelos de regresión logística con hipótesis lineal y no lineal. Para entrenar el modelo de regresión logística, al igual que en la práctica 2, calculamos los pesos óptimos mediante la función `opt.fmin_tnc`.

2.1. Hipótesis lineal

Para el modelo de regresión logística con hipótesis lineal, antes que nada, añadimos una columna de unos a los datos de entrenamiento, de validación y de test.

Una vez hecho esto, usamos la función `testParam`, variando el parámetro de regularización de 0 a 10, con un paso de 0.5. Con ello, tratamos de buscar el λ que mejor se comportase sobre los datos de evaluación.

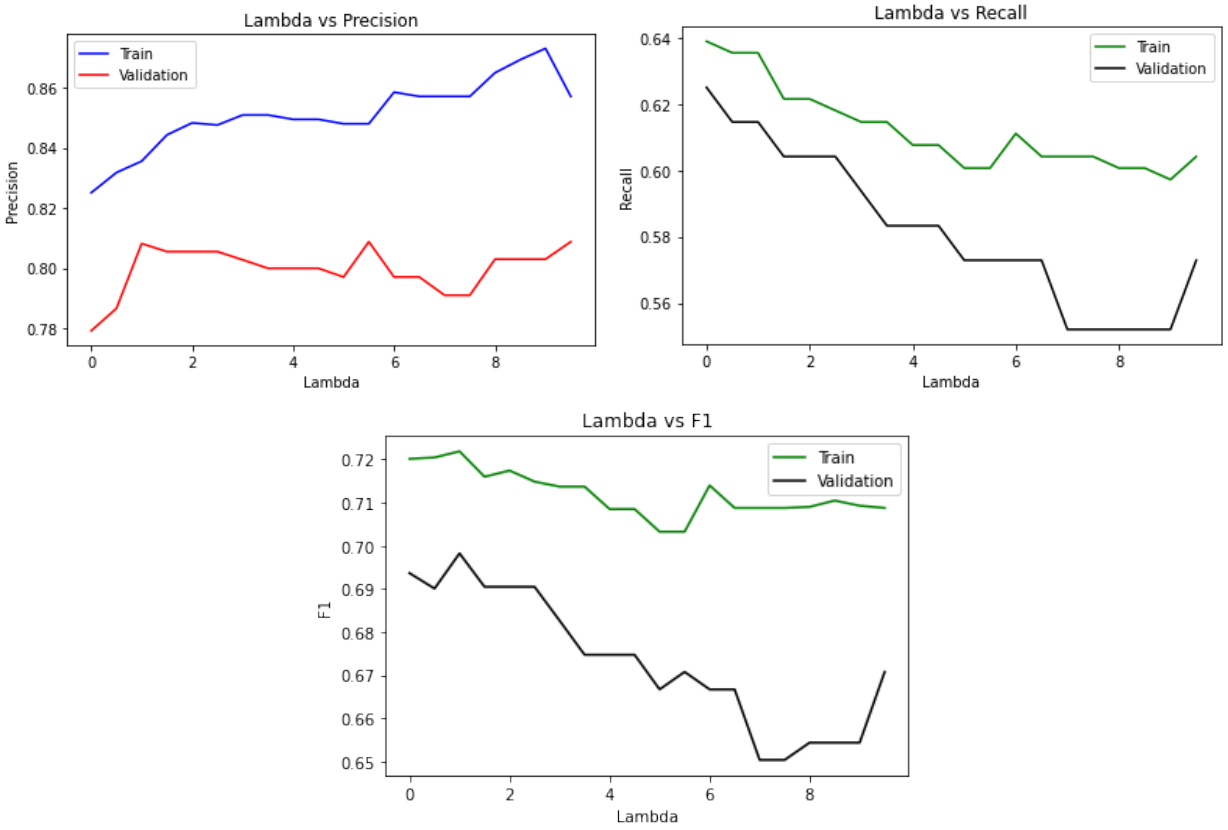

Figura 2.1: Resultados para la regresión logística con hipótesis lineal variando el parámetro de regularización λ.

Como podemos ver en las gráficas resultantes (Figura 2.1), para los datos de evaluación alcanzamos el valor máximo de $f_1 = 0.69$ con el parámetro de regularización $\lambda = 1.0$. Observando un poco más en detalle las gráficas, vemos que los resultados no son muy alentadores, pues hemos obtenido en evaluación un *recall* por debajo de 0.63, que resulta bastante bajo.

Por tanto, escogimos $\lambda = 1.0$ y entrenamos un nuevo modelo para evaluarlo con los datos de test. Incluimos en la Tabla 1 las métricas obtenidas para los datos de test y en la Figura 2.2 la matriz de confusión asociada.

Modelo	Conjunto	Accuracy	Precisión	Recall	F1
Regresión Logística con hipótesis lineal para $\lambda = 1$	Conjunto de test	0.953	0.855	0.614	0.715

Tabla 1: Métricas en test para la regresión logística con hipótesis lineal.

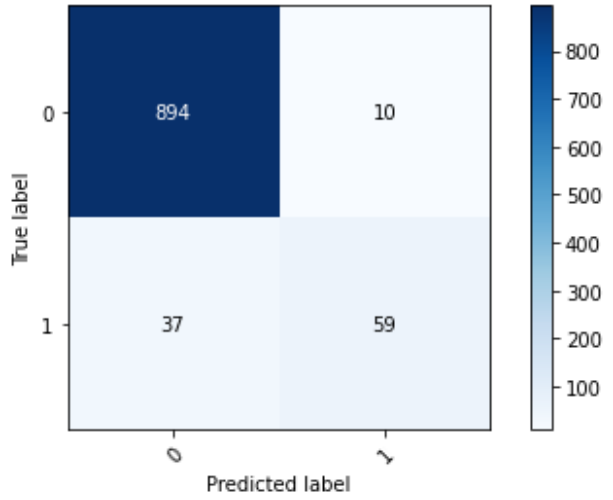

Figura 2.2: Matriz de confusión para la regresión logística con hipótesis lineal y $\lambda = 1.0$.

Hemos conseguido un $f_1 = 0.715$ para test. Analizando detenidamente la matriz de confusión observamos un exceso de falsos negativos provocado por una tendencia excesiva a predecir la clase cero (más de lo que debería). Como esta clase tiene muchos más ejemplos que la otra (90.4 % vs 9.6 %) se consigue un buen resultado de *accuracy*, pero el *recall* resulta bastante bajo, es decir, nos dejamos muchos positivos sin identificar. Este es el quid de la cuestión en nuestro problema.

Podemos concluir que una hipótesis lineal no es suficiente para nuestro problema. Aspiramos a obtener resultados mucho más prometedores en las métricas con una hipótesis más compleja que permita adaptarse mejor a los datos.

2.2. Hipótesis no lineales

Para la regresión logística con una hipótesis no lineal, tal y como vimos en clase, vamos a usar combinaciones polinómicas de los atributos de entrada. Para ello, usamos el código de la Práctica 2 que nos permite hacer esto mediante `PolynomialFeature`. Esta clase nos permite generar una matriz de características que consiste en todas las combinaciones polinómicas de las características de grado menor o igual al que recibe por parámetro. Después de probar para varios valores, decidimos quedarnos con grado 3, haciendo un total de 364 atributos. Con grado 2 vimos que los modelos se quedaban un poco cortos y para grados mayores a 3, los más de 1000 atributos que había que manejar, hacían bastante lento el entrenamiento sin que hubiese mejoras excesivas en el resultado.

Realizamos los pertinentes cambios a nuestros datos de entrada mediante la función `fit_transform` y, de nuevo, usamos la función `testParam` para variar el parámetro de regularización y escoger el que mejor se comporte con respecto a los datos de evaluación.

La Figura 2.3 nos muestra los resultados obtenidos. En concreto, vemos que con $\lambda = 9.5$ obtenemos el mejor valor de f_1 en evaluación ($f_1 = 0.85$). Así, entrenamos un nuevo modelo con $\lambda = 9.5$ para ver cómo se comporta con los datos de test.

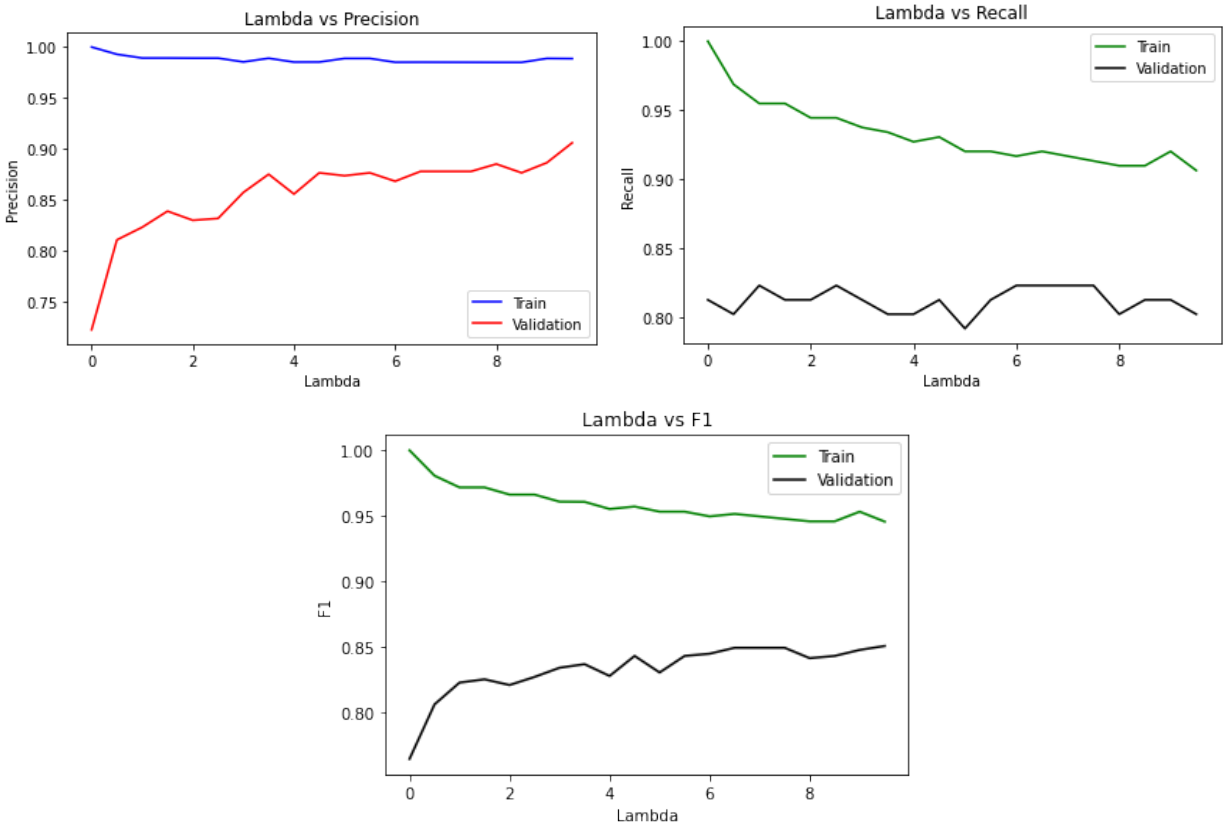

Figura 2.3: Resultados para la regresión logística con hipótesis no lineal variando el parámetro de regularización λ.

Modelo	Conjunto	Accuracy	Precisión	Recall	F1
Regresión Logística con hipótesis no lineal para $\lambda = 9.5$	Conjunto de test	0.979	0.962	0.812	0.881

Tabla 2: Métricas en test para la regresión logística con hipótesis no lineal.

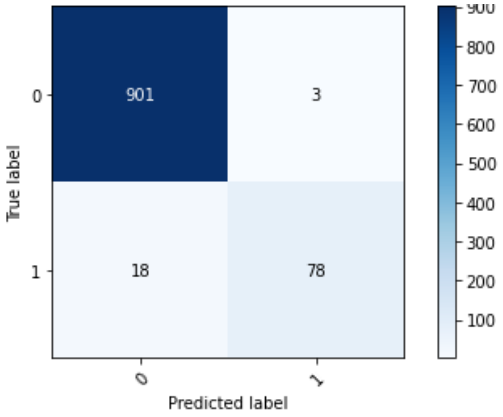

Figura 2.4: Matriz de confusión para la regresión logística con hipótesis no lineal y $\lambda = 9.5$.

Como vemos en la Tabla 2 y en la matriz de confusión (Figura 2.4), los resultados son bastante mejores que con la hipótesis lineal. Hemos obtenido una $f_1 = 0.881$ para test, y un *recall* que ya toma valores considerablemente buenos (por encima de 0.8). Podríamos haber utilizado directamente el *recall* para escoger el parámetro (en la Figura 2.3 se aprecia cómo llega a alcanzar alguna décima más), pero sería a costa de aumentar los falsos positivos reduciendo la precisión, que tampoco nos interesa. Con la f_1 tratamos de llevar controladas ambas cosas.

3. Redes neuronales

Para utilizar redes neuronales sobre nuestros datos probaríamos distintas topologías de red con distinta cantidad de neuronas y capas neuronas para distintos valores de los hiperparámetros (término de regularización λ y valor de iteraciones máximo *max_iter*).

Reutilizamos todo código desarrollado para la Práctica 4 pero, de cara a utilizar más de una capa oculta, generalizamos la función `backprop` para que funcionase también con varias capas neuronales apoyándonos en los apuntes del curso de Andrew Ng's[2].

También creamos alguna función extra respecto al código de las prácticas como `entrenaRed`. Esta función simplemente es una abstracción encargada de generar las matrices de pesos iniciales aleatorias con los las dimensiones adecuadas según los argumentos que recibe (recibe el número de capas de entrada y de etiquetas posibles), para llamar a la función `minimize` adecuadamente. Llevar esto a una función evita tener que repetir mucho código.

Adicionalmente se desarrolló la función `testLamdaItersNeuronas` para visualizar gráficamente los resultado de *precisión*, *recall* y f_1 para una red neuronal entrenada con distintos valores de λ, *max_iter* y cantidad de neuronas, movidos en 3 rangos recibidos por parámetro. Este modelo fue el único para el que no se reutilizó la función `testParam` ya que, al ser tres los parámetros a mover en lugar de uno, requería una distinción de casos excesiva que enfangaba bastante el código. Se prefirió tener una función exclusiva para ello aunque tuviese bastante en común con la otra.

Por último agregamos a todas nuestras funciones la posibilidad de fijar un *random_state* (que solo influye para la generación aleatoria de los pesos iniciales) d forma que nuestras ejecuciones pudiesen ser deterministas (si no los resultados podrían cambiar de una vez para otra que no era lo más deseable para nuestras pruebas).

3.1. Redes de una capa oculta

Como alterar el valor de 4 parámetros en busca de un buen modelo iba a ser excesivo, empezamos trabajando con topologías que tuviesen una sola capa oculta.

Lo primero que hicimos fue desarrollar un modelo que sobreaprendiese del conjunto de entrenamiento. Para ello, desusamos la regularización ($\lambda = 0$) y pusimos un número muy grande de iteraciones máximas ($max\_iter = 300$). El número de neuronas de la capa oculta lo fijamos a 25.

En la Tabla 3 pueden apreciarse los resultados perfectos de esta red sobre el conjunto de entrenamiento que demuestran su sobreaprendizaje y los malos resultados sobre el conjunto de evaluación que demuestran el problema que este sobre aprendizaje supone (la regresión logística no lineal obtuvo mejores resultados que este modelo).

Prueba	Conjunto	Accuracy	Precision	Recall	f_1
Red con 1 capa oculta de 25 neuronas	Entrenamiento	1.0	1.0	1.0	1.0
$\lambda = 0$ y $max\_iter = 300$	Validación	0.964	0.788	0.854	0.820

Tabla 3: Métricas en para entrenamiento y validación de una red neuronal que sobreaprende.

[2]Queremos mencionar que en el pseudocódigo que dan en ese curso hay un pequeño error de dimensiones que, tras buscar por internet, vimos que había sufrido más gente. Se trata de que no eliminan la primera componente de las $\delta^{(l)}$ (que es la derivada de una constante y no tiene sentido) al hacer la propagación hacia atrás. En estos enlaces está descrito el problema y la solución: https://stats.stackexchange.com/questions/278031/confusion-about-backpropagation-matrix-dimensions y https://stats.stackexchange.com/questions/321523/backpropagation-matrix-multiply-error-andrew-ng-machine-learning.

Consideramos los parámetros $\lambda \in \{0, 0.5, 1, 1.5, 2\}$, *max_iter* $\in \{30, 50, 70, 90, 110, 130, 150, 170, 190, 210, 230\}$ y *hidden_size* $\in \{25, 65, 105\}$ en todas sus posibles combinaciones, representando mediante la función `testLamdaItersNeuronas` las correspondientes gráficas (una gráfica por cada valor de λ y *hidden_size*, utilizando en ella el eje x para *max_iter* y el eje y para la puntuación de la métrica sobre el conjunto de evaluación).

Observamos que el parámetro de regularización $\lambda = 2$ ya empeoraba los resultados para el conjunto de validación incluso para valores grandes de *max_iter*. Por lo demás todos los resultados fueron similares, aunque si observamos cierta mejora al aumentar el número de neuronas de 25 a 65 (y no tanta al aumentar a 105). Escogimos el modelo cuyos parámetros se comportaron mejor con la métrica f_1 para validación. En la Figura 3.1 pueden verse las curvas de aprendizaje de la red neuronal de una capa con 65 neuronas y $\lambda = 1$, donde *max_iter* $= 230$ alcanzó una puntuación $f_1 = 0.903$.

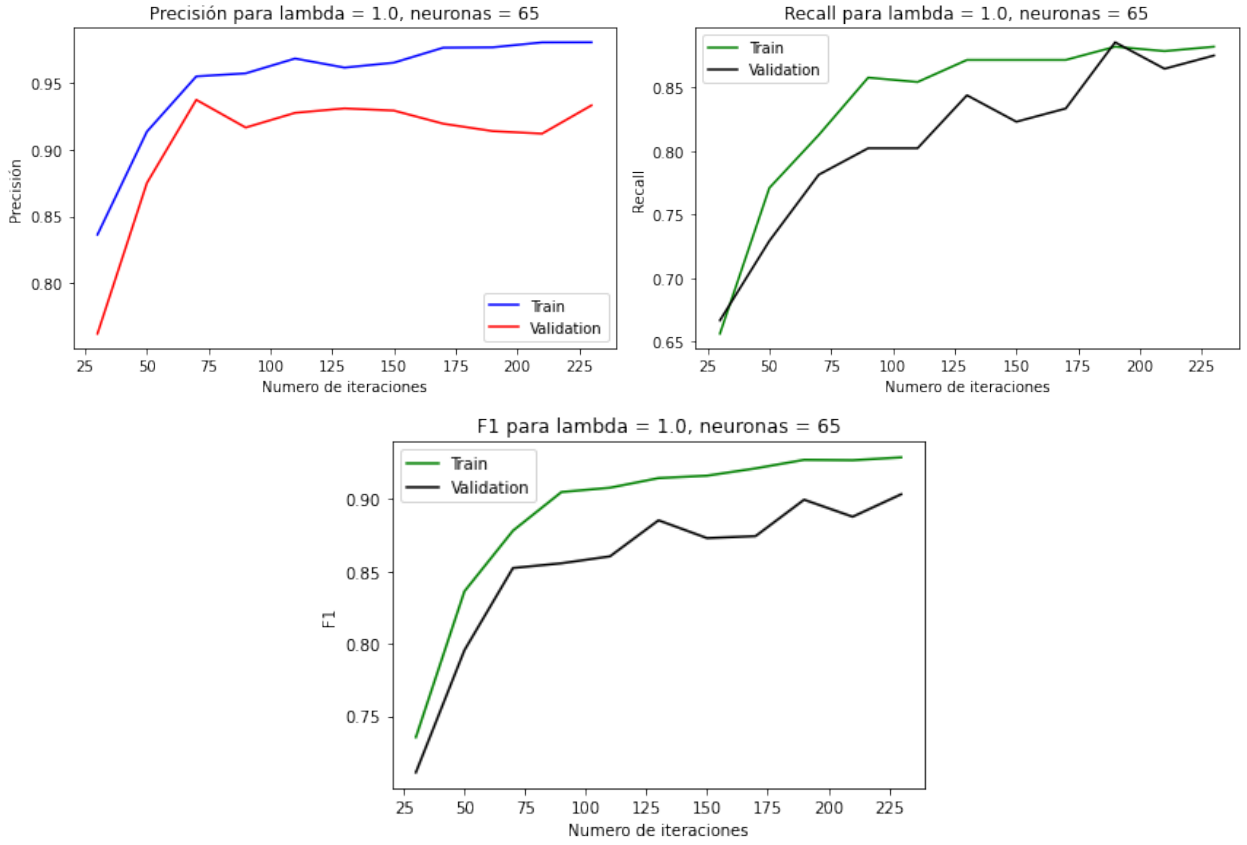

Figura 3.1: Resultado de las distintas métricas para la red neuronal con *hidden_size* $= 65$ y $\lambda = 1$ en función del número de iteraciones máximas.

Como percibimos que las puntuaciones estaban justo aumentando para el conjunto de validación al llegar a *max_iter* $= 230$, hicimos la gráfica para este caso llegando un poco más allá (hasta *max_iter* $= 300$), pero no el resultado no mejoró el resultado (seguía aumentando para ligeramente para entrenamiento pero no para validación).

Por lo tanto, escogimos esa red neuronal de un capa cuyas puntuaciones sobre los distintos conjuntos se pueden ver en la Tabla 4 y cuya matriz de confusión (sobre el conjunto de test) se encuentra en la Figura 3.2.

Prueba	Conjunto	Accuracy	Precision	Recall	f_1
Red con 1 capa oculta de 65 neuronas $\lambda = 1$ y $max\_iter = 230$	Entrenamiento	0.986	0.980	0.88	0.92
	Validación	0.982	0.933	0.875	0.903
	Test	0.981	0.963	0.833	0.893

Tabla 4: Resultados de las métricas para los 3 conjuntos con la red neuronal de una capa oculta escogida.

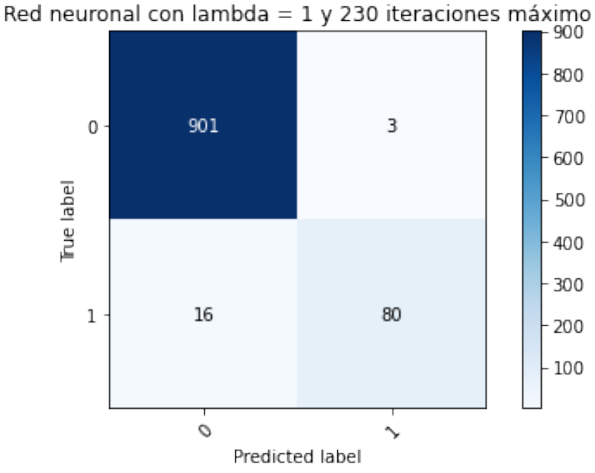

Figura 3.2: Matriz de confusión para la red neuronal escogida sobre el conjunto de test.

Observamos que los resultados, a grosso modo, se mantienen para el conjunto de test, donde aumenta la precisión ligeramente (un 0.03) y baja un poco el recall (un 0.04). En general, todos nuestros modelos han tenido más alta la precisión que el recall. Esto es algo que se consiguió aliviar al final con la técnica SMOT para introducir ejemplos sintéticos de la clase minoritaria.

3.2. Redes multicapa oculta

Para los hiperparámetros = 1 y $max\_iter = 230$ probamos con varias topologías de red de 2 y 3 capas neuronales y no apreció mejora alguna en los resultados para validación respecto a utilizar una sola capa. Estas redes tenían resultados ligeramente peores y tardaban algo más en entrenarse. En la Tabla 5 pueden verse las puntuaciones sobre el conjunto de validación para estas topologías multicapa y también para la red con una sola capa oculta de antes.

Prueba	Configuración de capas	Conjunto	Accuracy	Precision	Recall	f_1
Red con $\lambda = 1$ y max_iter = 230	(65)	Validación	0.982	0.933	0.875	0.903
	(85,85)	Validación	0.976	0.900	0.843	0.870
	(65,65)	Validación	0.979	0.903	0.875	0.888
	(45,45)	Validación	0.980	0.913	0.875	0.893
	(85,85,85)	Validación	0.976	0.928	0.812	0.866
	(65,65,65)	Validación	0.971	0.838	0.864	0.851
	(45,45,45)	Validación	0.976	0.867	0.885	0.876

Tabla 5: Resultados de las métricas para distintas topologías de red con distinto número de capas neuronales.

4. Support vector machines

Los siguiente modelo que aplicamos sobre nuestros datos fueron máquinas vectoriales de soporte. Al igual que hicimos en la Práctica 6, utilizamos las funciones que hay la librería `sklearn` para ello, pero lo acompañamos del estudio para elegir los hiperparámetros C y σ (en el caso del kernel gaussiano) mediante la función `testParam`.

Aunque ya intuíamos que nuestros datos no eran linealmente separables con un hiperplano debido a los malos resultados de la regresión logística de hipótesis lineal, probamos las SVM con kernel linear para comprobarlo (estas buscan la frontera de decisión que deje mejor margen pero no se diferencian en la frontera que proponen). En principio, la opción adecuada para nosotros era utilizar el kernel gaussiano.

4.1. Kernel lineal

En el caso del kernel lineal, primero tratamos de desarrollar un modelo que sobreaprendiese fijando un valor muy grande para el hiperparámetro C. Pero ni por esas se conseguían resultados demasiado buenos sobre el propio conjunto de entrenamiento, lo que demuestra que los datos no se pueden separar bien con un hiperplano.

Con $C = 100$ obtuvimos los resultados sobre entrenamiento y validación que pueden verse en la Tabla 6. Para los resultados de validación representamos su matriz de confusión (Figura 4.1).

Prueba	Conjunto	Accuracy	Precision	Recall	f_1
SVM linear con $C = 100$	Entrenamiento	0.954	0.896	0.600	0.719
	Validación	0.947	0.841	0.552	0.666

Tabla 6: SVM linear buscando que sobreaprenda ($C = 100$).

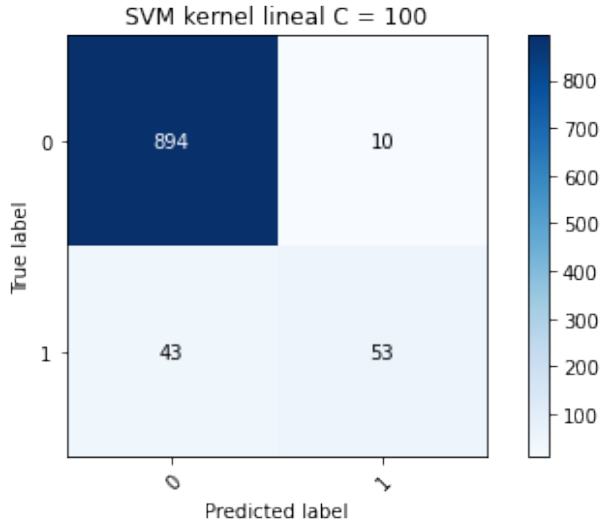

Figura 4.1: Matriz de confusión para la SVM linear con $C = 100$.

La puntuaciones de la máquina son malas incluso sobre el conjunto de entrenamiento al que tratamos de sobreajustar. El *recall* para entrenamiento es de apenas 0.6 y la $f_1 = 0.719$. Para el conjunto de validación es aún peor con $recall = 0.552$ y $f_1 = 0.666$. Viendo que la elección de la clase 1 está

por debajo de su proporción real en los datos parece que son bastantes los casos en que ejemplos de esta clase caen en el semiespacio definido por el hiperplano donde se predice clase 0. Si pudiésemos visualizarlo (no podemos porque son 11 dimensiones) seguramente veríamos que la clases están muy "mezcladas" y no es suficiente con una frontera lineal para separarlas bien.

A pesar de ello, probamos con los valores del hiperparámetro $C \in \{0.01, 0.05, 0.1, 0.5, 1, 1.5, 2.0, 2.5\}$. Encontramos que, a poco que C fuese suficientemente grande como para aprender algo del conjunto de entrenamiento, se producía la situación anterior (Figura 4.2). Definitivamente no era una buena idea este tipo de kernel.

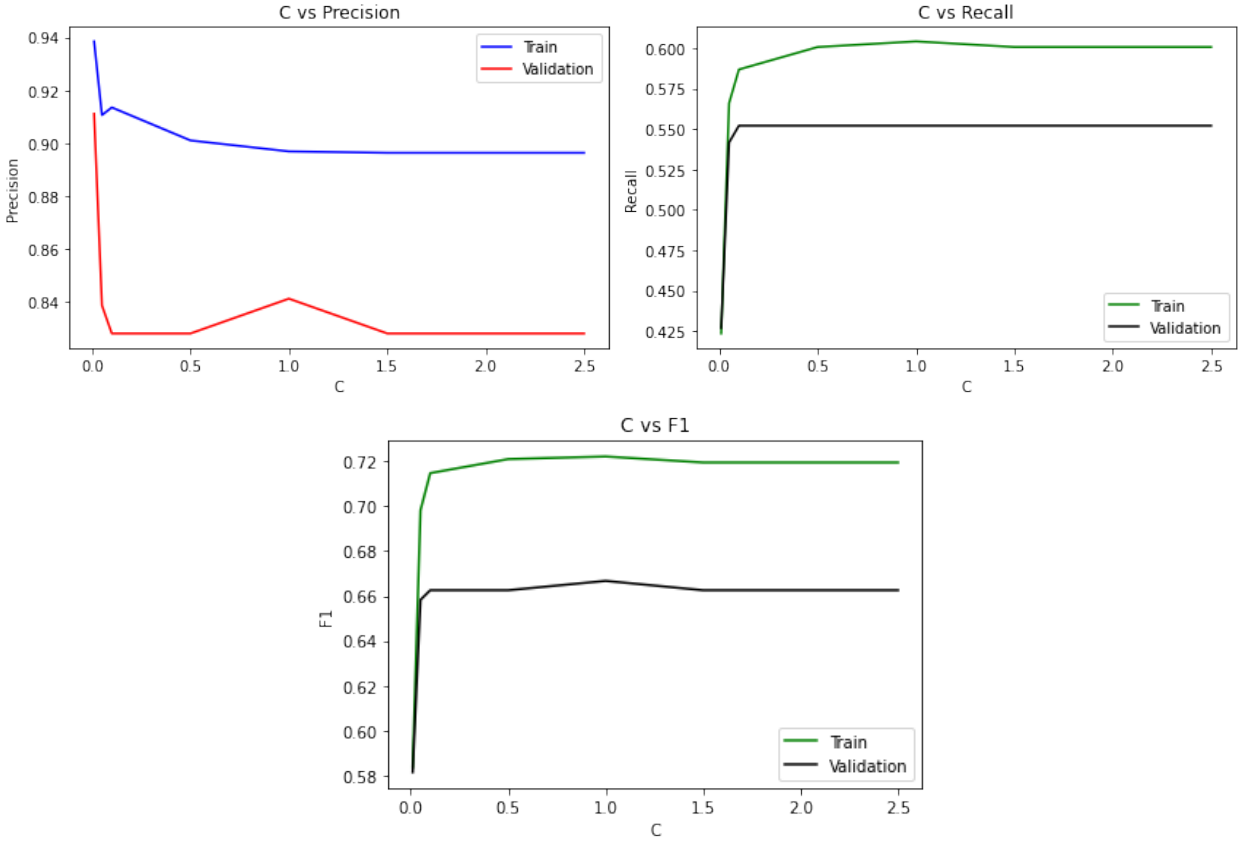

Figura 4.2: Métricas de puntuación para la SVM linear con distintos valores del parámetro C.

4.2. Kernel gaussiano

Comenzamos con el kernel gaussiano desarrollando también un modelo que sobreprendiese de los datos de entrenamiento. Para ello, fijamos un valor muy alto para C y un valor muy bajo para σ ($C = 100$ y $\sigma = 0.5$). En la Tabla 7 podemos ver sus resultados cobre entrenamiento y validación. El resultado perfecto en entrenamiento muestra el evidente sobre aprendizaje, que redunda en unos resultados desastrosos para validación.

Prueba	Conjunto	Accuracy	Precision	Recall	f_1
SVM gaussiana con $C = 100$ y $\sigma = 0.5$	Entrenamiento	1.0	1.0	1.0	1.0
	Validación	0.907	0.714	0.052	0.097

Tabla 7: SVM gaussiana que sobreaprenda ($C = 100$ y $\sigma = 0.5$).

Para tratar de ajustar los parámetros y regular el aprendizaje consideramos $\sigma \in \{0.5, 1, 1.5, ..., 10\}$ y $C \in \{0.5, 1, 1.5, ..., 24.5$ en todas sus posibles combinaciones, visualizando los resultados de las métricas sobre los conjuntos de entrenamiento y validación. Observamos mejores resultados cuando σ no era ni muy pequeño ni muy grande (en torno a 5) y algo similar para el para´metro C. El mejor resultado con validación se produjo para $sigma = 5$ y $C = 16.5$. Véanse las gráficas de puntuaciones para $\sigma = 5$ en función de C de la Figura 4.3 y los resultados de el modelo escogido de la tabla Tabla 8 también para el conjunto de test.

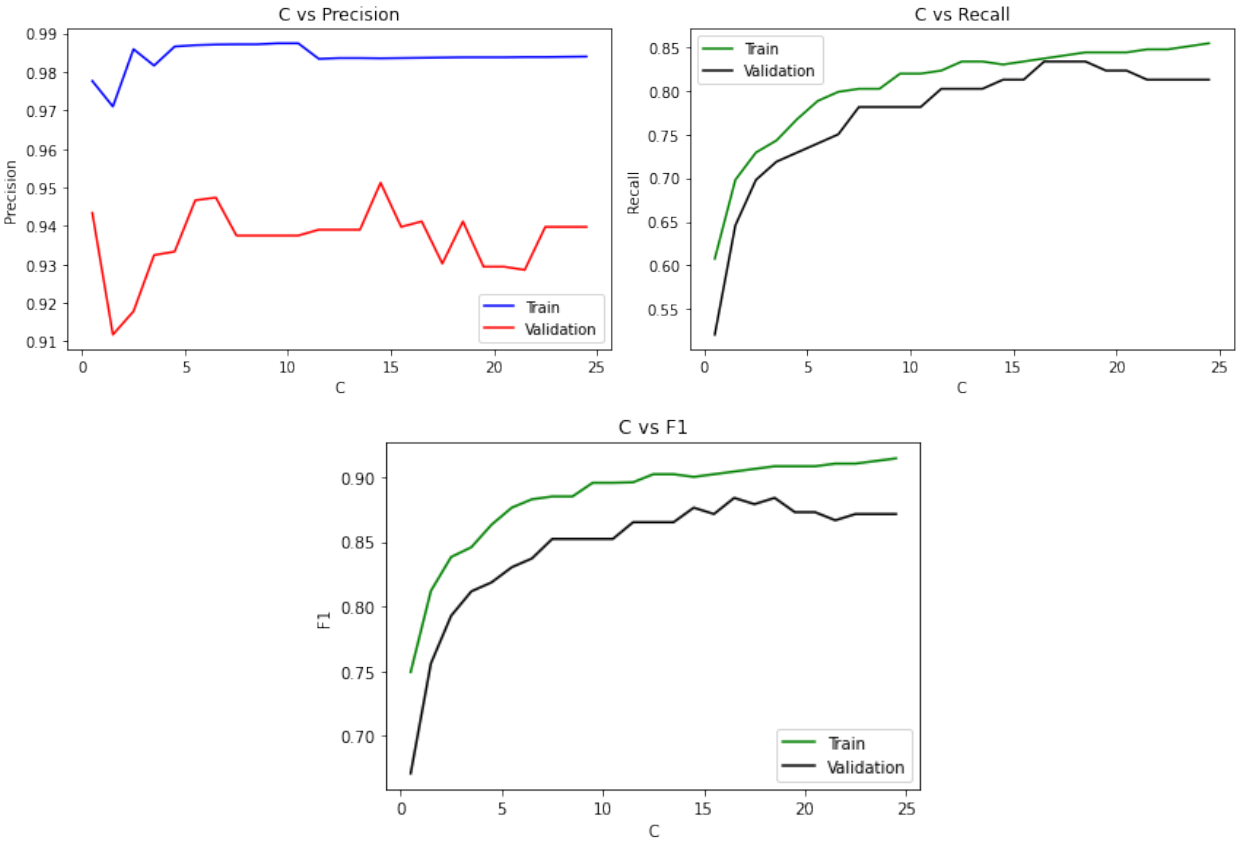

Figura 4.3: Métricas de puntuación para la SVM gaussiana con = 5en función del parámetro C.

Prueba	Conjunto	Accuracy	Precision	Recall	f_1
SVM gaussiana con $C = 16.5$ y $\sigma = 5$	Entrenamiento	0.982	0.983	0.836	0.904
	Validación	0.979	0.941	0.833	0.883
	Test	0.979	0.951	0.822	0.882

Tabla 8: SVM gaussiana elegida ($C = 16.5$ y $\sigma = 5$).

Observamos que el modelo tiene más o menos los mismos resultados sobre el conjunto de test que para el de validación, que no ha sido usado ni para entrenar ni para elegir parámetros, lo cual es satisfactorio. De nuevo apreciamos (al igual que en todos nuestros modelos) mejor *precisión* que el *recall*.

5. Modelos adicionales

Además de los modelos estudiados en la asignatura, vamos a aplicar un par de modelos adicionales para ver como se comportan con nuestro problema. En particular, vamos a usar árboles de decisión y random forest.

5.1. Árboles de decisión

Los árboles de decisión son un modelo de predicción donde cada nodo pregunta por el valor de una variable. Durante el entrenamiento, se construye un árbol eligiendo sucesivamente el atributo y la pregunta que mejor reducen el valor de cierta métrica, en nuestro caso la entropía (desorden de los ejemplos según sus clases). Básicamente se eligen las variables y preguntas más relevantes para solucionar el problema haciéndolas primero (parte del árbol cercana a la raíz). Así, el árbol construido durante el entrenamiento sirve para clasificar ejemplos nuevos, descendiendo desde la raíz respondiendo las preguntas para el ejemplo. Con ello, cuando se alcanza un nodo hoja, se elige como predicción la clase mayoritaria en este nodo.

Para entrenar este modelo, recurrimos a la clase `DecisionTreeClassifier` de `sklearn.tree`, que nos permite construir un árbol de decisión fácilmente mediante la función `fit`. En este caso, el parámetro que nos permite controlar el *overfitting* es la profundidad del árbol. Si dejásemos que el árbol fuese todo lo profundo que se quisiese podría clasificar correctamente todos los ejemplos de entrenamiento pero generalizaría peor de cara a nuevos datos de entrada. Limitando la profundidad máxima que puede alcanzar el árbol ayudamos a controlar el *overfitting*. De esta forma, al igual que con los modelos anteriores, haremos uso de la función `testParam` para entrenar varios árboles de decisión con profundidades desde 2 hasta 10.

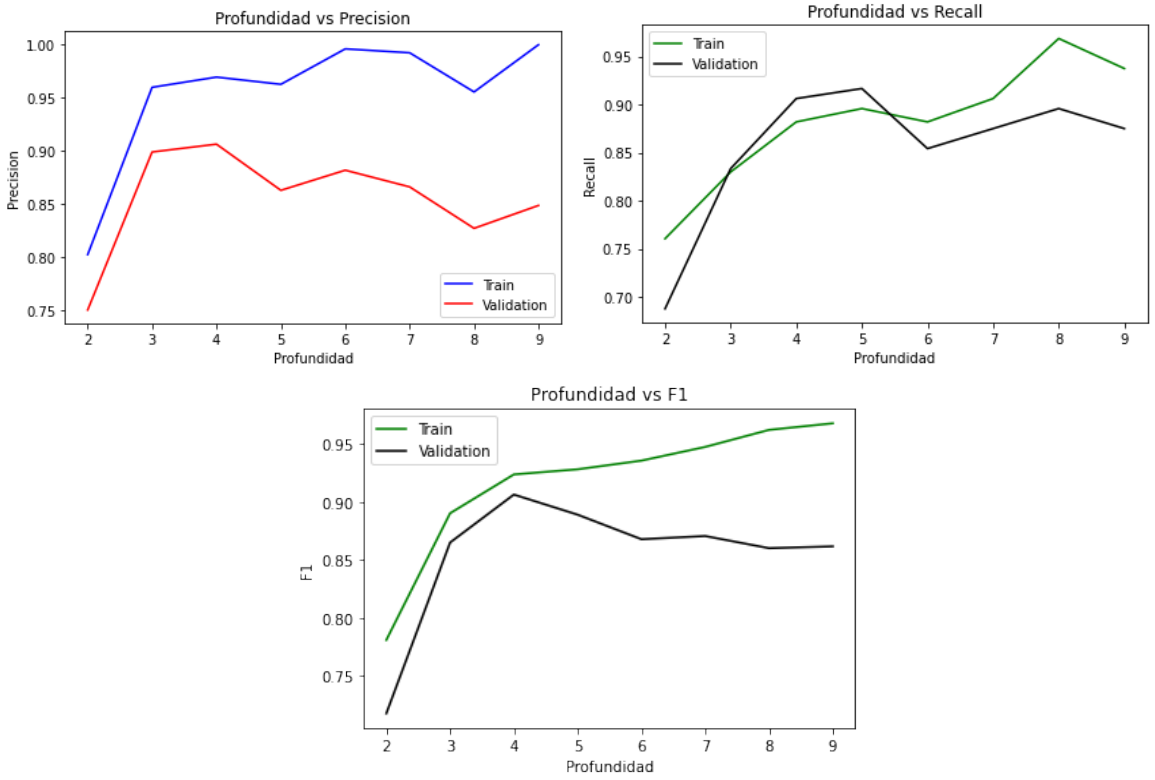

Figura 5.1: Resultados para árboles de decisión variando el parámetro de profundidad.

Como vemos en las gráficas de la Figura 5.1, para la profundidad 4 alcanzamos el f1 máximo en evaluación: 0.906. Notemos que, como hemos mencionado anteriormente, a medida que permitimos mayor profundidad, obtenemos casi el 100 % de aciertos en entrenamiento, pero en evaluación empeora considerablemente. Finalmente, nos quedamos con profundidad 4 y entrenamos un nuevo árbol de decisión para ver como se desempeña con los datos de test.

En la Tabla 9 se nos muestra las métricas obtenidas para test con profundidad 4. Hemos conseguido una gran precisión y un recall bastante aceptable. La f_1 consigue un valor de 0.917, que es bastante satisfactorio. Este modelo, siendo más simple e interpretable, resulta para este dataset igual de buenos que las redes neuronales.

Prueba	Conjunto	Accuracy	Precision	Recall	f_1
Árbol de decisión $profundidad = 4$	Entrenamiento	0.985	0.969	0.881	0.923
	Validación	0.982	0.906	0.906	0.906
	Test	0.985	0.976	0.864	0.917

Tabla 9: Métricas para el árbol de decisión de profundidad máxima 4.

Podemos ver también su matriz de confusión sobre el conjunto de validación en la Figura 5.2 donde, de nuevo, podríamos echamos en falta unos cuantas elecciones más de la clase 1 para llegar a los 96 casos que debería haber. Sin embargo, los apenas 2 falsos positivos están bastante bien en relación a las 85 veces que se ha respondido 1.

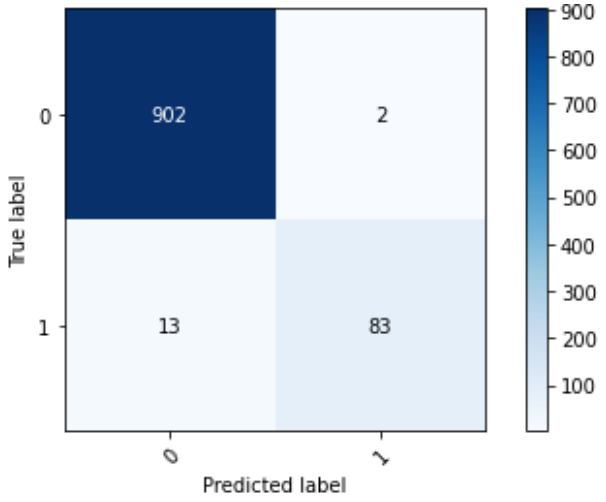

Figura 5.2: Matriz de confusión para árbol de decisión con profundidad 4.

Un punto muy a favor de los árboles de decisión es que podemos interpretarlo fácilmente. La función `plot_tree` de la librería `sklearn.tree` nos permite representar el árbol construido durante el entrenamiento con el fin de interpretarlo y poder explicar el porqué de una predicción u otra.

En la Figura 5.3 pueden verse los dos primeros niveles del árbol que, entre otras cosas, nos permiten identificar las categorías con más importancia para la predicción. En este caso, vemos que las categorías que mejor discriminan entre clases son: *Income* (el salario), *CCAvg* (gasto medio al mes con tarjeta de crédito) y *Education* (nivel de estudios).

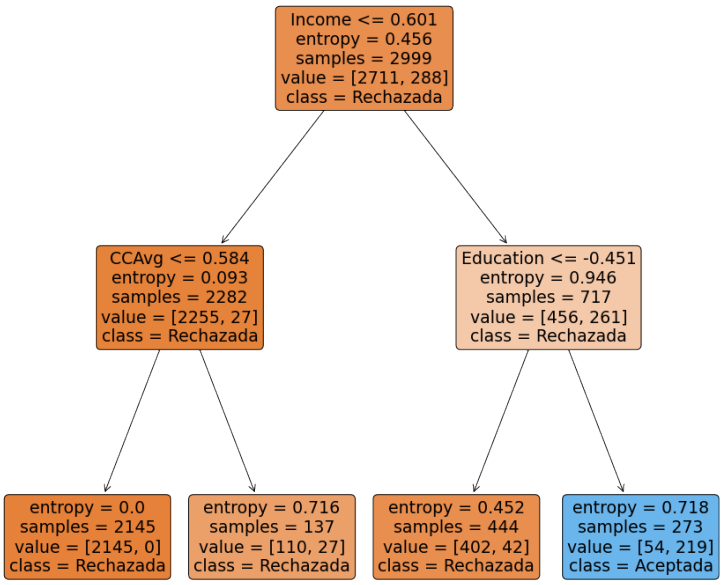

Figura 5.3: Primeros dos niveles del árbol de decisión con profundidad 4.

5.2. Random forest

Pasamos a trabajar con nuestro último modelo adicional, los *random forest*.

Este modelo consiste en varios árboles combinados con *bagging*, lo que significa que cada árbol recibe una parte de los ejemplos de entrenamiento y todos ellos aprenden por separado con los ejemplos recibidos de manera independiente (pudiendo ejecutarse en paralelo). De cara a la predicción, las decisiones de los distintos árboles se combinan de algún modo para formar la respuesta final. En un problema de clasificación, esta combinación de las respuestas de cada árbol suele consistir en elegir la respuesta más votada (la moda de las respuestas).

Para entrenar este modelo usamos la clase `RandomForestClassifier` de la librería `sklearn.ensemble`. De nuevo, probamos con varias cantidades de árboles (parámetro `n_estimators`) controlando el *overfitting* con la profundidad máxima de cada árbol (`max_depth`). Más concretamente, consideramos `n_estimators` $\in \{2, 10, 25, 50, 75, 100, 500\}$ y `max_depth` $\in \{2, 3, ..., 15\}$ en todas sus posibles combinaciones.

Observamos una ligera mejoría al aumentar el número de árboles para la predicción, aunque cada vez era más costoso el entrenamiento. Nos quedamos con 50 árboles porque sus resultados apenas mejoraban en alguna centésima con más árboles que tardaban mucho más en entrenarse. En la Figura 5.4 podemos ver las curvas para *precisión*, *recall* y f_1 sobre el conjunto de validación, para 50 árboles, en función de la profundidad máxima `max_depth`.

Para `max_depth` $= 12$ alcanzamos $f_1 = 0.907$ máxima en evaluación. Nos resulta un poco sospechoso que los resultados de este modelo sobre entrenamiento sean perfectos (por miedo a *overfitting*), pero sobre validación los resultados son muy buenos (que es lo que importa). Los resultados de este modelo para los 3 conjuntos (también sobre test) pueden verse en la Tabla 10, mientras que su matriz de confusión sobre el conjunto de test está en la Figura 5.5.

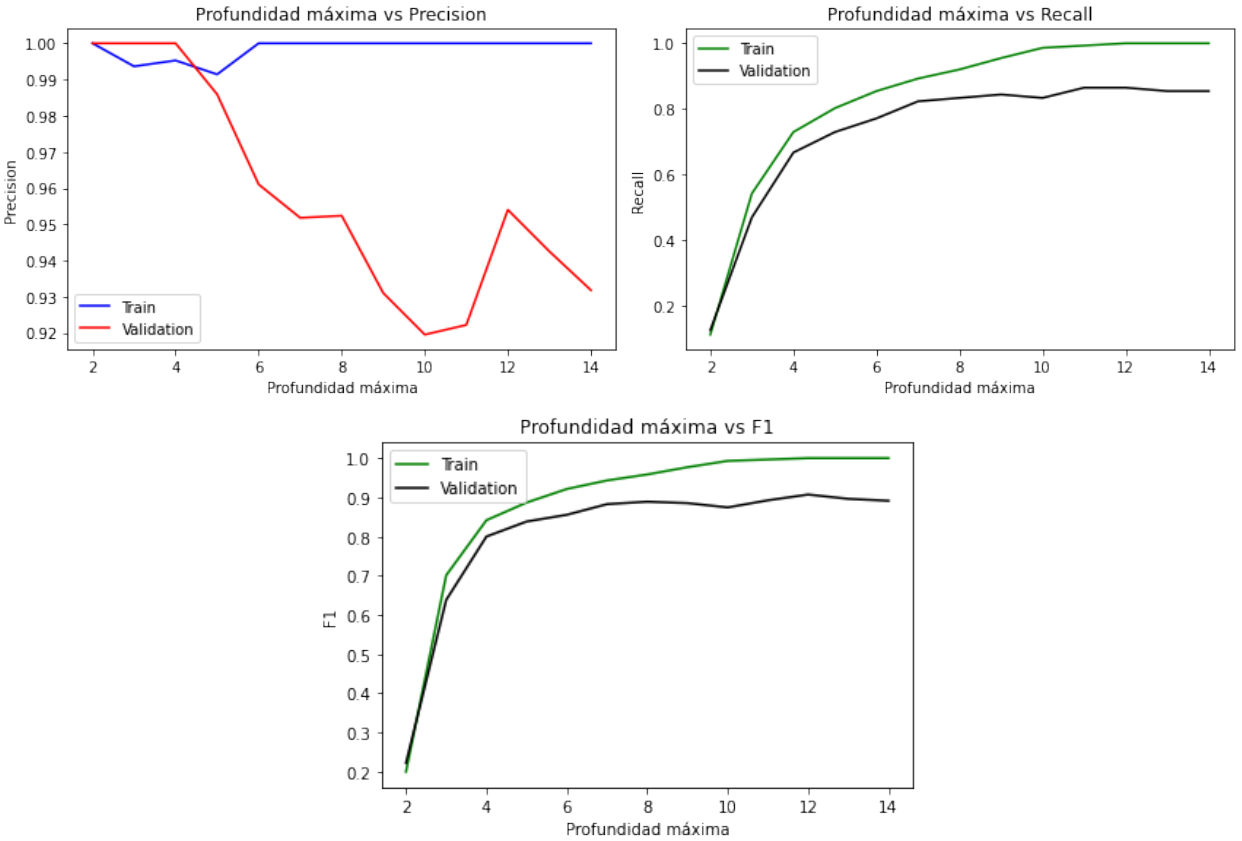

Figura 5.4: Resultados para random forest con `n_estimators` = 50 según `max_depth`.

Prueba	Conjunto	Accuracy	Precision	Recall	f_1
Random forest `n_estimators` = 50 y `max_depth` = 12	Entrenamiento	1.0	1.0	1.0	1.0
	Validación	0.983	0.954	0.864	0.907
	Test	0.985	1.0	0.843	0.915

Tabla 10: Métricas para random forest con 50 árboles y profundidad máxima sobre los 3 conjuntos.

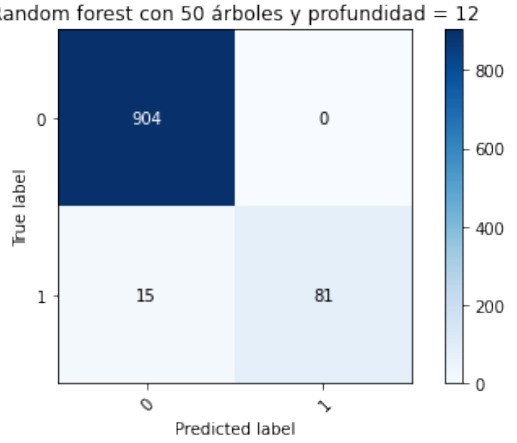

Figura 5.5: Matriz de confusión para random forest sobre el conjunto de test.

6. Sobremuestreo sintético de la clase minoritaria (SMOTE)

Como guinda final al proyecto decidimos probar SMOTE (Synthetic Minority Oversampling Technique), una técnica que conocíamos para tratar con datasets cuya clase objetivo está muy desbalanceada.

Esta técnica consiste en añadir ejemplos sintéticos de la clase minoritaria para el conjunto de entrenamiento, de forma que los modelos aprendan más esta clase y tiendan a tenerla más en cuenta. La manera de sintetizar ejemplos nuevos consiste en considerar el segmento que une a dos ejemplos ya existentes (considerándolos como puntos en el espacio) y tomando un valor intermedio de ese segmento que será el ejemplo sintético nuevo.

Dependiendo del problema puede ser interesante sesgar los modelos sobrerepresentando la clase minoritaria para darle más importancia en el entrenamiento. Por ejemplo, en los problemas de clasificación de cáncer, el cáncer maligno suele ser bastante minoritario, y un *recall* bajo puede ser muy peligroso, aunque la *precisión* sea alta, porque significa que hay bastantes cánceres de este tipo que no se están identificando. ¿Es preferible diagnosticar bien a todas las personas que tienen cáncer maligno aunque alguna que no lo tiene se pueda ir con un susto a casa pensando que sí, o es preferible ser más conservador al diagnosticar a riesgo de que haya cánceres malignos sin detectar correctamente? A nuestro juicio es preferible lo primero. En ese tipo de problemas la técnica SMOTE puede ser muy útil para aumentar el *recall*.

En nuestro problema el *recall* no parece tan determinante. Nuestros modelos parecen ligeramente más propensos al conservadurismo diciendo que una persona no aceptará el crédito algo menos de lo que deberían, pero esto no nos parece ni mejor ni peor que una tendencia opuesta a predecir de más la clase minoritaria. Sin embargo, no de más explorar otras opciones que pudieran alterar los valores de *precisión* y *recall*.

Aplicamos SMOTE de manera moderada aumentando la cantidad de ejemplos de entrenamiento de la clase 1 de 288, que había hasta el momento, a 500 ejemplos (112 ejemplos sintéticos). Volviendo a calcular las métricas para los modelos de cada tipo que mejor había funcionado (con los hiperparámetros que ya habíamos elegido) obtuvimos los resultados de la Tabla 11.

Prueba	SMOTE	Accuracy	Precision	Recall	f_1
Regresión logística con hipótesis no lineal	No	0.979	0.962	0.812	0.881
	Sí	0.974	0.843	0.895	0.868
Red neuronal con $\lambda = 1$ y max_iter $= 230$	No	0.982	0.975	0.833	0.898
	Sí	0.984	0.900	0.937	0.918
SVM con $\sigma = 5$ y $C = 16.5$	No	0.979	0.951	0.822	0.882
	Sí	0.980	0.895	0.895	0.895
Árbol de decisión de profundidad máxima 4	No	0.985	0.976	0.864	0.917
	Sí	0.851	0.387	0.947	0.549

Random Forest de 50 árboles conprofundidad máxima 12	No	0.985	1.0	0.843	0.915
	Sí	0.876	0.432	0.937	0.592

Tabla 11: Resultados de los modelos con y sin SMOTE sobre el conjunto de test.

Apreciamos cómo la técnica incrementa para la regresión logística, la red neuronal y la SVM el *recall*, disminuyendo la *precisión* más o menos en la misma proporción. La puntuación de *recall* pasa en todo los casos a superar a la puntuación *precisión*. Los resultados de f_1 se mantienen en estos casos más o menos igual. Para lo que el SMOTE funciona terriblemente mal es con los árboles de decisión y el random forest. A pesar de aumentar el *recall*, la *precisión* se desploma (0.387 y 0.432 respectivamente), lo que nos hace pensar en una predicción excesiva de la clase 1 debida al sobremuestreo artificial.

7. Conclusiones

Durante el análisis de datos, el estudio de los modelos para diferentes parámetros y la evaluación exhaustiva de los mismos, hemos ido extrayendo varias conclusiones que recopilamos a continuación:

- El dataset tiene un ejemplo con un código postal erróneo que optamos por eliminar.
- La variable *Experience* toma varios valores negativos que no tienen sentido. Hay motivos para pensar que se les ha colado un signo '-' al introducirlos y que la solución es tomar el valor absoluto de esa columna.
- Las variables *Age* y *Experience* están correlacionadas a la perfección y resultan redundantes. Se puede eliminar una de ellas.
- Es recomendable discretizar el valor de algunas variables como la edad (variable *Age*).
- La clase objetivo a predecir está profundamente desbalanceada en los datos. El 90.4 % de los ejemplos son de la clase 0 (personas que no aceptaron el crédito bancario) y apenas el 9.6 % restante son de la clase 1 (personas que sí aceptaron el crédito).
- El estudio de un dataset tan desbalanceado debe hacerse atendiendo a la *precisión* y al *recall*. Para ello, salvo que se quiera controlar especialmente una de esas dos medidas, es muy interesante usar la f_1. Las puntuaciones de *accuracy* en un dataset tan desbalanceado son enormemente buenas hasta en modelos que funcionan de manera absurda.
- El reescalado de los datos es útil, incluso para modelos que no lo necesitan, pues reduce considerablemente el tiempo de entrenamiento.
- Una hipótesis lineal es insuficiente para tratar adecuadamente los datos mediante una regresión logística. Es precisa una hipótesis con un grado superior para obtener buenos resultados. Con grado 3, donde la cantidad de atributos es 364, se alcanzan valores de f_1 cercanos a 0.9 manteniendo el *recall* por encima de 0.8.
- Las redes neuronales no mejoran demasiado el resultado que se puede obtener con una regresión logística. Eligiendo adecuadamente los hiperparámetros de aprendizaje y la topología de red se alcanza $f_1 \simeq 0.9$ con el *recall* algo por encima de 0.8 (tanto en validación como en test).
- Las redes neuronales con una sola capa oculta funcionan igual de bien (o incluso algo mejor) que las que tienen varias capas. Esto, junto a los malos con una hipótesis lineal para la regresión, nos hace pensar que para separar los datos hace falta cierta no linealidad, pero no demasiada.
- Las SVM son una mala opción con un kernel lineal. Deducimos que, como se vio también con la regresión logística, un hiperplano no puede separar bien los datos.
- Las SVM con un kernel gaussiano son una opción igual de buena que las redes neuronales o la regresión logística de hipótesis no lineal. Ajustando adecuadamente los hiperparámetros C y σ obtiene resultados muy similares a esos dos modelos.
- Los árboles de decisión y modelos random forest se comportan un poco mejor que todos los modelos anteriores consiguiendo rebasar ligeramente la barrera del 0.9 para f_1.
- Todos los modelos obtienen mayor puntuación de *precisión* que de *recall* debido a cierta tendencia a responder menos la clase minoritaria de lo que deberían por su proporción de ejemplos en el dataset (deberían responderla unas 96 de cada 1.000 veces).

- La técnica SMOT, que permite incrementar la cantidad de ejemplos de la clase minoritaria, es útil para cambiar las tornas haciendo que suba el *recall* a costa de perder *precisión*.
- Los resultados de nuestros modelos han sido más o menos igual de buenos sobre el conjunto de test que sobre el conjunto de validación. Creemos que los modelos desarrollados generalizan bien y serían útiles para predecir este problema en casos reales.

Anexo: código del proyecto

Para el desarrollo del proyecto hemos utilizado *Jupyter Notebook*, y para ver el código junto a su salida recomendamos acceder directamente al *notebook* desarrollado desde el siguiente enlace:

https://github.com/Jorgitou98/Aprendizaje-Automatico-y-Big-Data/blob/main/Proyectofinal/Proyecto_Final_Personal_Loan.ipynb

En cualquier caso, adjuntamos a continuación el código desarrollado durante el proyecto.

```
#!/usr/bin/env python
# coding: utf-8

# # Proyecto Final
#
# ### Jorge Villarrubia Elvira y Juan Carlos Villanueva Quirs

# In[1]:


#Todas las librerias externas usadas
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import scipy.optimize as opt

from sklearn.model_selection import train_test_split
from scipy.optimize import minimize
from sklearn.metrics import confusion_matrix
from sklearn.utils.multiclass import unique_labels
from sklearn.preprocessing import PolynomialFeatures
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import plot_tree
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler


# Primero, cargamos el dataset.

# In[2]:


#Leemos el data set
loan_rawdf = pd.read_csv('Bank_Personal_Loan_Modelling.csv')
#Vemos los primeros para comprobar si se han cargado correctamente
loan_rawdf.head()
```

```
# Ahora, procedemos a realizar el preprocesado de datos. Para ello, vamos a:
# - Comprobar si hay valores nulos y arreglarlos si es necesario.
# - Deshacernos de variables poco tiles (las que esten muy correlacionadas entre
    s).
# - Comprobar que los valores de las categoras sean plausibles y modificarlos si
    es necesario.
# - Discretizar un poco ms en rangos algunas variables continuas no eliminadas
    como los ingresos (no tiene sentido otras como el nivel educativo que apenas
    toma 4 valores distintos).
# - Hacer la particin de los datos en conjunto de entrenamiento, validacin y test
    .
# - Normalizar los datos (aunque algunos de nuestros modelos no lo necesiten
    tenemos rangos muy distintos como los ingresos y el nivel educativo y los
    modelos suelen funcionar mejor reescalando).

# Afortunadamente no tenemos valores nulos.

# In[3]:


#Comprobamos si tenemos valores nulos
loan_rawdf.isnull().sum()


# Ahora, visualicemos el data set y veamos las principales caractersticas. Llama
    la atencin es desbalanceo de la clase Personal Loan a predecir que tendremos
    que tener en cuenta en adelante y el histograma de ZIP Code.

# In[4]:


#Visualizamos los datos
loan_rawdf.hist(figsize=(10,10))
plt.tight_layout()


# Hay un ejemplo cuyo ZIP code est equivocado. Lo eliminamos.

# In[5]:


np.sum(loan_rawdf['ZIP Code']< 90000)
loan_rawdf[loan_rawdf['ZIP Code']< 90000]['ZIP Code']


# In[6]:
```

```
loan_rawdf = loan_rawdf.drop(index=loan_rawdf[loan_rawdf['ZIP Code']< 90000]['ZIP
    Code'].index)


# In[7]:


loan_rawdf['ZIP Code'].hist(figsize=(4,4))
plt.title('ZIP Code')
plt.tight_layout()


# Nos fijamos en la correlacin entre las variables. Observamos un un 0.99 de
   correlacin entre Experience y Age que hace que podamos eliminar una de las dos
   . Eliminamos Experience. Eliminamos tambin ID que simplemente indica el nmero
   de fila e cada ejemplo y no tiene ningn sentido.

# In[8]:


plt.figure(figsize=(13, 8))
sns.heatmap(loan_rawdf.corr(), annot=True, vmin=-1, vmax=1)


# In[9]:


loan_rawdf = loan_rawdf.drop(columns=['Experience'])
loan_rawdf = loan_rawdf.drop(columns=['ID'])


# En la visualizacin de los datos podemos observar que algunos valores de la
   columna _experiencia_ son negativos. Esto no tiene mucho sentido pues dicha
   categora representa los aos de experiencia profesional. Como decidimos
   eliminarlas no hay falta preocuparse de ello.

# In[10]:


#Imprimimos la informacion del data set
loan_rawdf.info()

#Imprimimos tambien las estadisticas principales
loan_rawdf.describe()


# In[11]:
```

```
print("Primer tercil de 'Age': ", np.percentile(loan_rawdf.Age, 33.33))
print("Segundo tercil de 'Age': ", np.percentile(loan_rawdf.Age, 66.66))


# Discretizamos la variable Age donde quizs podra interesarnos una informacin ms
    general (jven, mediana edad y mayor) en lugar de los aos concretos. La persona
     ms jven tiene 8 aos y la ms mayor 67

# In[12]:


age_values = loan_rawdf.Age
bins = [np.min(age_values) ,39, 52, np.max(age_values)]
loan_rawdf['Age'] = pd.cut(age_values, bins, labels=bins[:-1], include_lowest =
    True)
loan_rawdf


# Visualizamos a cantidad de casos de cada clase

# In[13]:


plt.pie(data=loan_rawdf,x=loan_rawdf["Personal Loan"].value_counts(),autopct='
    %1.1f%%')
plt.legend(['Personal Loan = 0','Personal Loan = 1'], loc="lower left")


# ### Separacin de los datos en entrenamiento, validacin y test.
# Hacemos un particin 60% para test, 20% para validacin y 20% para test. Primero
     partimos 80%-20%, y luego el de 80% lo partimos 75%-25% (el 75% del 80% es
     el 60% del original y el 25% del 80% es el 20%). Para no desbalancear an ms
     el dataset en nuetros conjuntos hacemos estas particiones estratificadas.
    Comprobamos que el porcentaje de casos de cada clase es el mismo en los 3
    conjuntos e igual al del dataset.

# In[14]:


X = loan_rawdf.loc[:, loan_rawdf.columns != 'Personal Loan'].to_numpy()
y = loan_rawdf['Personal Loan'].to_numpy()


# In[15]:


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=333, stratify = y)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size
```

```
    =0.25, random_state=333, stratify = y_train)


# In[16]:


print('Porcentaje de la clase 0 en entrenamiento: ', np.sum(y_train == 0) / len(
    y_train)*100, '%')
print('Porcentaje de la clase 1 en entrenamiento: ', np.sum(y_train == 1) / len(
    y_train)*100, '%')
print('Porcentaje de la clase 0 en validacin: ', np.sum(y_val == 0) / len(y_val)
    *100, '%')
print('Porcentaje de la clase 1 en validacin: ', np.sum(y_val == 1) / len(y_val)
    *100, '%')
print('Porcentaje de la clase 0 en test: ', np.sum(y_test == 0) / len(y_test)
    *100, '%')
print('Porcentaje de la clase 1 en test: ', np.sum(y_test == 1) / len(y_test)
    *100, '%')


# ### Normalizacin de los datos
# Para el mejor funcionamiento de algunos modelos normalizamos los datos para que
    tengan una escala similar y pequea. Es importante normalizar por separado los
    conjuntos de train, eval y test para no contaminar los conjunto de prueba con
    informacin de entrenamiento.

# In[17]:


def normalize_data(X):
    """
    Args:
        X: Datos de entrada

    Dados unos datos de entrada, los normaliza haciendo (datos - media) /
    desviacion_estandar.
    Ademas, devuelve las medias y desviaciones estandar obtenidas
    """
    media = np.mean(X, axis = 0)
    desviacion = np.std(X, axis = 0)
    print(media)
    print(desviacion)
    return (X - media) / desviacion, media, desviacion


# In[18]:


X_train_norm, _, _ = normalize_data(X_train)
```

```
X_val_norm, _ ,_ = normalize_data(X_val)
X_test_norm, _ ,_ = normalize_data(X_test)


# ## Regresin Logstica

# In[19]:


#Funciones relacionadas con las metricas

def metricas(predicted, correct):
    accuracy = np.sum(predicted == correct) / len(correct)
    if np.sum(predicted == 1) == 0:
        precision = 0
    else:
        precision = np.sum(np.logical_and(predicted == 1, predicted == correct))/
   np.sum(predicted == 1)
    if np.sum(correct == 1) == 0:
        recall = 0
    else:
        recall = np.sum(np.logical_and(predicted == 1, predicted == correct))/np.
   sum(correct == 1)
    if precision + recall == 0:
        f1 = 0
    else:
        f1 = 2*precision*recall/(precision + recall)
    return accuracy, precision, recall, f1

def muestraMetricas(predicted, correct):
    accuracy, precision, recall, f1 = metricas(predicted, correct)
    print('Accuracy obtenido: {}'.format(accuracy))
    print('Precisin obtenida: {}'.format(precision))
    print('Recall obtenido: {}'.format(recall))
    print('F1 obtenido: {}'.format(f1))

#Funcion que dibuja la matriz de confusion.
#Extraida de  https://scikit-learn.org/stable/auto_examples/model_selection/
   plot_confusion_matrix.html
def plot_confusion_matrix(y_true, y_pred,normalize=False,title=None,cmap=plt.cm.
   Blues):
    if not title:
        if normalize:
            title = 'Normalized confusion matrix'
        else:
            title = 'Confusion matrix, without normalization'

    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)
```

```
    # Only use the labels that appear in the data
    classes = unique_labels(y_true, y_pred)
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    fig, ax = plt.subplots()
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    # We want to show all ticks...
    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           # ... and label them with the respective list entries
           xticklabels=classes, yticklabels=classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    # Loop over data dimensions and create text annotations.
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")
    fig.tight_layout()
    return ax


# In[20]:


# Funcion sigmoide, coste y gradiente que usaremos para el entrenamiento de los
    clasificadores de regresion logistica
def sigmoide(z):
    return 1 / (1 + np.exp(-z))

def costeRegu(theta, X, Y, lamda):
    m = np.shape(X)[0]
    return (np.matmul(np.log(sigmoide(np.matmul(X, theta))).T, Y)    + np.matmul(
```

```
    np.log(1 - sigmoide(np.matmul(X, theta)) + 1e-6).T, (1-Y)))/-m    + lamda* np.
    sum(theta**2)/(2*m)

def gradienteRegu(theta, X, Y, lamda):
    m = np.shape(X)[0]
    aux = lamda/m * theta
    aux[0] = 0
    return np.matmul(X.T, sigmoide(np.matmul(X,theta)) - Y)/m + aux


# In[21]:


#Anadimos columna de unos a los datos de entrenamiento, evaluacion y test
X_train_norm_unos = np.hstack([np.ones([X_train_norm.shape[0], 1]), X_train_norm
    ])
X_val_norm_unos = np.hstack([np.ones([X_val_norm.shape[0], 1]), X_val_norm])
X_test_norm_unos = np.hstack([np.ones([X_test_norm.shape[0], 1]), X_test_norm])


# Funcin de test para eleccin de parametros que utilizaremos varias veces.

# In[22]:


def testParam(X_train, y_train, X_val, y_val, rango_parametro, nombre_modelo,
    nombre_parametro = 'Parametro', random_state = 0, gamma = None, n_estimators
    =100) :
    precision_train_list = []
    precision_val_list = []
    recall_train_list = []
    recall_val_list = []
    f1_train_list = []
    f1_val_list = []
    for parametro in rango_parametro :
        #Entrenamos el modelo con los datos de entrenamiento
        if nombre_modelo == 'regresion logistica' :
            modelo = opt.fmin_tnc(func = costeRegu, x0 = np.zeros(X_train.shape
    [1]), fprime = gradienteRegu, args=(X_train, y_train, parametro))
            #En modelo[0] estan guardados los thetas (pesos) optimos
            predicted_train = sigmoide(np.matmul(X_train, modelo[0])) >= 0.5
            predicted_val = sigmoide(np.matmul(X_val, modelo[0])) >= 0.5
        elif nombre_modelo == 'arbol decision' :
            modelo = DecisionTreeClassifier(criterion = 'entropy', max_depth =
    parametro, min_samples_split = 2, random_state = random_state)
            modelo = modelo.fit(X_train, y_train)
            predicted_train = modelo.predict(X_train)
            predicted_val = modelo.predict(X_val)
        elif nombre_modelo == 'random forest' :
```

```
            modelo = RandomForestClassifier(n_estimators =n_estimators, max_depth
=parametro, random_state=random_state)
            modelo.fit(X_train, y_train)
            predicted_train = modelo.predict(X_train)
            predicted_val = modelo.predict(X_val)
        elif nombre_modelo == 'SVM linear':
            modelo = SVC(kernel = 'linear', C = parametro, random_state =
random_state)
            modelo.fit(X_train, y_train)
            predicted_train = modelo.predict(X_train)
            predicted_val = modelo.predict(X_val)
        elif nombre_modelo == 'SVM gaussian':
            modelo = svm = SVC(kernel = 'rbf', C = parametro, gamma = gamma,
random_state = random_state)
            modelo.fit(X_train, y_train)
            predicted_train = modelo.predict(X_train)
            predicted_val = modelo.predict(X_val)
        else :
            return 'Nombre de modelo no encontrado'

        #Metricas con datos de entrenamiento
        _, precision_train, recall_train, f1_train = metricas(predicted_train,
y_train)
        precision_train_list.append(precision_train)
        recall_train_list.append(recall_train)
        f1_train_list.append(f1_train)

        #Metricas con datos evaluacion
        _, precision_val, recall_val, f1_val = metricas(predicted_val, y_val)
        precision_val_list.append(precision_val)
        recall_val_list.append(recall_val)
        f1_val_list.append(f1_val)

    fig, (ax1,ax2) = plt.subplots(1,2)
    fig.set_size_inches(12,4)
    ax1.plot(rango_parametro, precision_train_list, label = 'Train', color = '
blue')
    ax1.plot(rango_parametro, precision_val_list, label = 'Validation', color = '
red')
    ax1.set_xlabel(nombre_parametro)
    ax1.set_ylabel('Precision')
    ax1.set_title(nombre_parametro + ' vs Precision')
    ax1.legend()

    ax2.plot(rango_parametro, recall_train_list, label = 'Train', color = 'g')
    ax2.plot(rango_parametro, recall_val_list, label = 'Validation', color = '
black')
    ax2.set_xlabel(nombre_parametro)
    ax2.set_ylabel('Recall')
```

```
    ax2.set_title(nombre_parametro + ' vs Recall')
    ax2.legend()
    fig.tight_layout()
    plt.show()

    fig, ax = plt.subplots()
    ax.plot(rango_parametro, f1_train_list, label = 'Train', color = 'g')
    ax.plot(rango_parametro, f1_val_list, label = 'Validation', color = 'black')
    plt.xlabel(nombre_parametro)
    plt.ylabel('F1')
    ax.set_title(nombre_parametro + ' vs F1')
    plt.legend()
    plt.show()

    indice_maximo = f1_val_list.index(max(f1_val_list))
    print('Para el valor de ' + nombre_parametro + ' {} alcanzamos el f1 maximo
    en evaluacion: {}'.format(rango_parametro[indice_maximo], max(f1_val_list)))


# Entrenamos modelos para varios valores de lambda (Explicar ms a fondo aqu)

# In[23]:


testParam(X_train_norm_unos, y_train, X_val_norm_unos, y_val, rango_parametro =
    np.arange(0, 10, 0.5),
          nombre_modelo = 'regresion logistica', nombre_parametro = 'Lambda')


# Escogemos el valor de lambda para el que se maximiza F1 en evaluacin: 1.0

# In[24]:


lamda = 1.0
theta = np.zeros(X_train_norm_unos.shape[1])
result = opt.fmin_tnc(func = costeRegu, x0 = theta, fprime = gradienteRegu, args
    =(X_train_norm_unos, y_train, lamda))
theta_opt = result[0]

#Metricas con datos entrenamiento
predicted_train = sigmoide(np.matmul(X_train_norm_unos, theta_opt)) >= 0.5
print('Resultados para train')
muestraMetricas(predicted_train, y_train)

predicted_val = sigmoide(np.matmul(X_val_norm_unos, theta_opt)) >= 0.5
print('\nResultados para eval')
muestraMetricas(predicted_val, y_val)
```

```
predicted_test = sigmoide(np.matmul(X_test_norm_unos, theta_opt)) >= 0.5
print('\nResultados para test')
muestraMetricas(predicted_test, y_test)

print('\nMatriz de confusion')
plot_confusion_matrix(y_true = y_test, y_pred = predicted_test, normalize=False,
    title='Regresion logistica con lambda = {}'.format(lamda), cmap=plt.cm.Blues)


# Ahora probamos con polinomios

# In[25]:


poly = PolynomialFeatures(3)
X_train_norm_poly = poly.fit_transform(X_train_norm)
X_val_norm_poly = poly.fit_transform(X_val_norm)
X_test_norm_poly = poly.fit_transform(X_test_norm)


# In[26]:


testParam(X_train_norm_poly, y_train, X_val_norm_poly, y_val, rango_parametro =
    np.arange(0, 10, 0.5),
          nombre_modelo = 'regresion logistica', nombre_parametro = 'Lambda')


# Escogemos lambda = 9.5

# In[27]:


lamda = 9.5
theta = np.zeros(X_train_norm_poly.shape[1])
result = opt.fmin_tnc(func = costeRegu, x0 = theta, fprime = gradienteRegu, args
    =(X_train_norm_poly, y_train, lamda))
theta_opt = result[0]

#Metricas con datos entrenamiento
predicted_train = sigmoide(np.matmul(X_train_norm_poly, theta_opt)) >= 0.5
print('Resultados para train')
muestraMetricas(predicted_train, y_train)

predicted_val = sigmoide(np.matmul(X_val_norm_poly, theta_opt)) >= 0.5
print('\nResultados para eval')
muestraMetricas(predicted_val, y_val)

predicted_test = sigmoide(np.matmul(X_test_norm_poly, theta_opt)) >= 0.5
```

```
print('\nResultados para test')
muestraMetricas(predicted_test, y_test)

print('\nMatriz de confusion')
plot_confusion_matrix(y_true = y_test, y_pred = predicted_test, normalize=False,
    title='Regresion logistica con terminos polinomicos y lambda = {}'.format(
    lamda), cmap=plt.cm.Blues)


# ## Redes neuronales
#
# Funciones para el entrenamiento de redes neuronales

# In[28]:


def forwardPropagation(X, layers):
    #La variable 'a' corresponde a la entrada de cada capa, que al principio
    seran los datos de entrada X.
    #Ojo: trasponemos para tratar las dimensiones correctamente
    m = X.shape[0]
    a = X
    A = []
    #Propagacin hacia delante genrica que sirve para ms de 2 matrices
    for i in range(len(layers)):
        #Aadimos unos a la entrada de cada capa (los terminos independientes)
        a = np.hstack([np.ones([m, 1]), a])
        A.append(a)

        #Cogemos los pesos correspondientes a la capa que toca
        thetas = layers[i]
        #Hacemos la multiplicacion
        z = np.matmul(a, thetas.T)

        #Y por ultimo la funcion de activacion
        a = sigmoide(z)

    A.append(a)
    #Al salir del bucle, tendremos la salida de la red neuronal
    #Trasponemos para devolver la misma forma que los datos de entrada
    return A, a

def sigmoide(z):
    return 1 / (1 + np.exp(-z))

def costeNeuronalRegu(X,Y, layers, regu):
    _, h = forwardPropagation(X, layers)
    m = X.shape[0]
    reguTerm = 0
```

```
    # Para tantas matrices como nos pasen.
    # As es genrico, en lugar de hacerlo para 2 matrices como en los apuntes
    for layer in layers:
        #Quitamos la ltima columna de 1's
        reguTerm += np.sum(layer[:,1:]**2)
    reguTerm *= regu/(2*m)
    return np.sum(Y * np.log(h+1e-8) + (1 - Y) * np.log(1-h +1e-8))*-1/m +
   reguTerm

def backprop(params_rn, num_entradas, ocultas_dims, num_etiquetas, X, y, reg):
    m = X.shape[0]
    dims = (num_entradas,) + ocultas_dims + (num_etiquetas,)
    L = len(dims)
    layers = []
    ini = 0
    fin = 0
    for i in range(L-1):
        fin = fin + dims[i+1] * (dims[i] + 1)
        layers.append(np.reshape(params_rn[ini:fin], (dims[i+1], (dims[i] + 1))))
        ini = fin
    A, _ = forwardPropagation(X, layers)
    Deltas = []
    for i in range(L-1):
        Deltas.append(np.zeros(layers[i].shape))
    for k in range(m):
        d = []
        d.append(A[L-1][k] - y[k])
        d.append(np.dot(layers[L-2].T, d[-1]) * (A[L-2][k] * (1 - A[L-2][k])))
        for l in range(L-3, 0, -1):
            d.append(np.dot(layers[l].T, d[-1][1:]) * (A[l][k] * (1 - A[l][k])))
        d.append(None)
        d = d[::-1]
        for l in range(L-2):
            Deltas[l] = Deltas[l] + np.dot(d[l+1][1:, np.newaxis], A[l][k][np.
   newaxis, :])
        Deltas[L-2] = Deltas[L-2] + np.dot(d[L-1][:, np.newaxis], A[L-2][k][np.
   newaxis, :])
    coste = costeNeuronalRegu(X, y, layers, reg)
    ThetasCero = []
    for i in range(L-1):
        ThetasCero.append(np.c_[np.zeros((layers[i].shape[0],1)), layers[i
   ][:,1:]])
    D = []
    for i in range(L-1):
        D.append(Deltas[i]/m + reg*ThetasCero[i]/m)
    grad = np.ravel(D[0])
    for i in range(1,L-1):
        grad = np.r_[grad, np.ravel(D[i])]
    return coste, grad
```

```
def initializeRandom(size1, size2, INIT_EPSILON, random_state = None):
    np.random.seed(random_state)
    return np.random.uniform(low = -INIT_EPSILON, high = INIT_EPSILON, size = (
    size2, size1+1) )

def y_onehot(y, num_labels):
    m = y.shape[0]
    y_onehot = np.zeros((m, num_labels))
    for i in range(m):
        y_onehot[i][y[i]] = 1
    return y_onehot

def y_predicted(X, input_size, hidden_sizes, num_labels, thetaOpt):
    thetas = []
    dims = (input_size,) + hidden_sizes + (num_labels,)
    L = len(dims)
    ini = 0
    fin = 0
    for i in range(L-1):
        fin = fin + dims[i+1] * (dims[i] + 1)
        thetas.append(np.reshape(thetaOpt[ini:fin], (dims[i+1], (dims[i] + 1))))
        ini = fin
    # Obtenemos la probabilidad para todos los ejemplos de entrenamiento, de que
    pertenezca a cada clase
    _, probability = forwardPropagation(X, thetas)
    # Nos quedamos con el indice que contiene la mayor probabilidad, ajustando
    correctamente los indices
    index = np.argmax(probability, axis=1)
    return index

def entrenaRed(X, y, input_size, hidden_sizes, num_labels, reg, INIT_EPSILON =
    0.12, max_iter = None, random_state = None):
    Thetas_rand = []
    layersDims = (input_size,) + hidden_sizes + (num_labels,)
    L = len(layersDims)
    for i in range(L-1):
        Thetas_rand.append(initializeRandom(layersDims[i], layersDims[i+1],
    INIT_EPSILON, random_state))
    params = np.ravel(Thetas_rand[0])
    for i in range(1,L-1):
        params = np.r_[params, np.ravel(Thetas_rand[i])]
    return minimize(backprop, x0=params, args=(input_size, hidden_sizes,
    num_labels, X, y, reg), method='TNC', jac=True,options={'maxiter': max_iter})


# Funciones con las mtricas para evaluar los distintos modelos

# ### Modelo que sobreajusta
```

```
# Parece estar claramente sobreaprendiendo de los datos pues sus resultados sobre
    el conjunto de entrenamiento son muchisimo mejores (casi perfectos). La
    utilizacin de un valor no nulo para la regularizacin y la reduccin del nmero
    mximo de iteraciones pueden ser herramientas para controlar este overfitting y
    mejorar el comportamiento del modelo sobre datos desconocidos.

# In[29]:


input_size = X_train_norm.shape[1]
hidden_size = (25,)
num_labels = 2
reg = 0
max_iter = 300
fmin = entrenaRed(X_train_norm, y_onehot(y_train, num_labels), input_size,
    hidden_size, num_labels, reg, max_iter = max_iter, random_state = 100)


# In[30]:


predicted_train = y_predicted(X_train_norm, input_size, hidden_size, num_labels,
    fmin.x)
print('Resultados para train')
muestraMetricas(predicted_train, y_train)
predicted_val = y_predicted(X_val_norm, input_size, hidden_size, num_labels, fmin
    .x)
print('\nResultados para eval')
muestraMetricas(predicted_val, y_val)


# Vamos a representar la matriz de confusin del modelo sobre el conjunto de
    entrenamiento.
#
# Observamos que sus fallos provienen fundamentalmente de predecir la clase 1 (14
    fallos vs 77 aciertos para la clase 1 y 19 fallos vs 890 aciertos para la
    clase 0). Es decir, parece haber un exceso de falsos positivos (probablemente
    porque la infrarepresentacin de esta clase en el dataset ha hecho que la
    aprenda peor).

# In[31]:


plot_confusion_matrix(y_true = y_val, y_pred = predicted_val, normalize=False,
    title='Red neuronal sin regularizar y 300 iteraciones mximo',cmap=plt.cm.Blues
    )


# ### Regularizacin y control del sobreaprendizaje
```

```
# Para dicha red vamos a variar el valor del parametro de aprendizaje $\lambda$ y
    el nmero mximo de iteraciones tratando de encontrar un modelo que no
   sobreaprenda y funcione mejor sobre el conjunto de validacin para precisin y
   recall.

# In[32]:


def testLamdaItersNeuronas(X_train, y_train, X_val, y_val, rangoLambda,
   rangoIters, rangoNeuronas, input_size, num_labels, random_state = None):
    for hidden_size in rangoNeuronas:
        for lamda in rangoLambda:
            precision_lamda_train = []
            precision_lamda_val = []
            recall_lamda_train = []
            recall_lamda_val = []
            f1_lamda_train = []
            f1_lamda_val = []
            for max_iter in rangoIters:
                fmin = entrenaRed(X_train_norm, y_onehot(y_train, num_labels),
   input_size, (hidden_size,), num_labels, lamda, max_iter = max_iter,
   random_state = random_state)

                predicted_train = y_predicted(X_train_norm, input_size, (
   hidden_size,), num_labels, fmin.x)
                _, precision_train, recall_train, f1_train = metricas(
   predicted_train, y_train)
                precision_lamda_train.append(precision_train)
                recall_lamda_train.append(recall_train)
                f1_lamda_train.append(f1_train)

                predicted_val = y_predicted(X_val_norm, input_size, (hidden_size
   ,), num_labels, fmin.x)
                _, precision_val, recall_val, f1_val = metricas(predicted_val,
   y_val)
                precision_lamda_val.append(precision_val)
                recall_lamda_val.append(recall_val)
                f1_lamda_val.append(f1_val)

            fig, (ax1,ax2) = plt.subplots(1,2)
            fig.set_size_inches(12,4)
            ax1.plot(rangoIters, precision_lamda_train, label = 'Train', color =
   'blue')
            ax1.plot(rangoIters, precision_lamda_val, label = 'Validation', color
    = 'red')
            ax1.set_xlabel('Numero de iteraciones')
            ax1.set_ylabel('Precisin')
            ax1.set_title('Precisin para lambda = {}, neuronas = {}'.format(lamda
   , hidden_size))
```

```
            ax1.legend()

            ax2.plot(rangoIters, recall_lamda_train, label = 'Train', color = 'g'
    )
            ax2.plot(rangoIters, recall_lamda_val, label = 'Validation', color =
    'black')
            ax2.set_xlabel('Numero de iteraciones')
            ax2.set_ylabel('Recall')
            ax2.set_title('Recall para lambda = {}, neuronas = {}'.format(lamda,
    hidden_size))
            ax2.legend()
            fig.tight_layout()
            plt.show()

            fig, ax = plt.subplots()
            ax.plot(rangoIters, f1_lamda_train, label = 'Train', color = 'g')
            ax.plot(rangoIters, f1_lamda_val, label = 'Validation', color = '
    black')
            plt.xlabel('Numero de iteraciones')
            plt.ylabel('F1')
            ax.set_title('F1 para lambda = {}, neuronas = {}'.format(lamda,
    hidden_size))
            plt.legend()
            plt.show()
            indice_maximo = f1_lamda_val.index(max(f1_lamda_val))
            print('Para el max_iter = {} tenemos mxima f1 en evaluacin con lambda
     = {} y {} neuronas: {}'.format(rangoIters[indice_maximo], lamda, hidden_size,
     max(f1_lamda_val)))


# In[33]:


# Para cada lambda en el primera rango representamos la precisin recall y f1 con
    las iteraciones del segundo rango
testLamdaItersNeuronas(X_train = X_train_norm, y_train = y_train, X_val = X_val,
    y_val = y_val, rangoLambda = np.arange(0,2.5,0.5), rangoIters = np.arange
    (30,250,20), rangoNeuronas = range(25,110,40), input_size = input_size,
    num_labels= num_labels, random_state = 100)


# In[34]:


testLamdaItersNeuronas(X_train = X_train_norm, y_train = y_train, X_val =
    X_val_norm, y_val = y_val, rangoLambda = np.arange(1,1.1,0.5), rangoIters = np
    .arange(30,310,20), rangoNeuronas = range(65,66,40), input_size = input_size,
    num_labels= num_labels, random_state = 100)
```

```
# ### Test nmero de neuronas

# ### Red neuronal escogida (65 neuronas, $\lambda = 1$ y max_iter = 230) y
    prueba sobre conjunto de test

# La red neuronal escogida tiene los siguientes resultados sobre el conjunto de
    entrenamiento y validacin.

# In[35]:


lamda = 1
max_iter = 230
hidden_size = (65,)
fmin = entrenaRed(X_train_norm, y_onehot(y_train, num_labels), input_size,
    hidden_size, num_labels, reg = lamda, max_iter = max_iter, random_state = 100)
predicted_train = y_predicted(X_train_norm, input_size, hidden_size, num_labels,
    fmin.x)
print('Resultados para train')
muestraMetricas(predicted_train, y_train)
predicted_val = y_predicted(X_val_norm, input_size, hidden_size, num_labels, fmin
    .x)
print('\nResultados para val')
muestraMetricas(predicted_val, y_val)


# Sobre el conjunto de test que no ha influido ni en su entrenamiento ni en la
    eleccin de los hiperparmetros obtiene resultados muy similares.

# In[36]:


predicted_test = y_predicted(X_test_norm, input_size, hidden_size, num_labels,
    fmin.x)
print('\nResultados para test')
muestraMetricas(predicted_test, y_test)


# Observamos en la matriz de confusin cierta tendencia a responder la clase
    minoritaria, menos de lo que debera.

# In[37]:


plot_confusion_matrix(y_true = y_test, y_pred = predicted_test, normalize=False,
    title='Red neuronal con lambda = 1 y 230 iteraciones mximo',cmap=plt.cm.Blues)
```

```
# ### Pruebas de la red neuornal con varias capas
# Modificamos algunas de las funciones que teniamos para trabajar con una capa
    para que puedan funcionar con una tupla que indique el nmero de neuoronas en
    cada capa oculta.

# Prueba con ms capas de 25 neuronas.

# In[38]:


topologias = [(65,), (85,85), (65,65), (45,45), (85,85,85), (65,65,65),
    (45,45,45)]
input_size = X_train_norm.shape[1]
reg = 1
max_iter = 230
num_labels = 2
for topo in topologias:
    fmin = entrenaRed(X_train_norm, y_onehot(y_train, num_labels), input_size,
    topo, num_labels, reg, max_iter = max_iter, random_state = 100)
    print('Configuracin {}'.format(topo))
    predicted_train = y_predicted(X_train_norm, input_size, topo, num_labels,
    fmin.x)
    print('Resultados para train lambda = {}, max_iter = {}'.format(reg, max_iter
    ))
    muestraMetricas(predicted_train, y_train)
    predicted_val = y_predicted(X_val_norm, input_size, topo, num_labels, fmin.x)
    print('\nResultados para val lambda = {}, max_iter = {}'.format(reg, max_iter
    ))
    muestraMetricas(predicted_val, y_val)
    print('\n\n')


# ### Suported vector machines

# #### Kernel lineal
# Primero, un modelo que sobreaprenda (parametro C = 100). Con un modelo lineal
    no consigue sobreaprender ms.

# In[39]:


svm = SVC(kernel = 'linear', C=100)
svm.fit(X_train_norm, y_train)

predicted_train = svm.predict(X_train_norm)
print('Resultados para train')
muestraMetricas(predicted_train, y_train)
predicted_val = svm.predict(X_val_norm)
print('\nResultados para eval')
```

```
muestraMetricas(predicted_val, y_val)


# In[40]:


plot_confusion_matrix(y_true = y_val, y_pred = predicted_val, normalize=False,
    title='SVM kernel lineal C = 100',cmap=plt.cm.Blues)


# In[41]:


Cs = [0.01, 0.05, 0.1, 0.5, 1, 1.5, 2.0, 2.5]
testParam(X_train_norm, y_train, X_val_norm, y_val, rango_parametro = Cs,
          nombre_modelo = 'SVM linear', nombre_parametro = 'C')


# #### Kernel gaussiano
# Modelo que sobreaprende

# In[42]:


C= 100
sigma = 0.5
svm = SVC(kernel = 'rbf', C=C, gamma = 1/(2* sigma**2))
svm.fit(X_train_norm, y_train)

predicted_train = svm.predict(X_train_norm)
print('Resultados para train')
muestraMetricas(predicted_train, y_train)
predicted_val = svm.predict(X_val_norm)
print('\nResultados para eval')
muestraMetricas(predicted_val, y_val)


# Apenas dice la clase minoritaria.

# In[43]:


plot_confusion_matrix(y_true = y_val, y_pred = predicted_val, normalize=False,
    title='SVM kernel gaussiano que sobreaprende',cmap=plt.cm.Blues)


# Con sigma = 4 y C= 24.5 se obtiene un 0.89 de f1 y el recall es alto. Parece
    que la curvas siguen creciendo. Representemos un poco ms alla para este valor.
```

```
# In[44]:


sigmas = np.arange(0.5,10.5,0.5)
Cs = np.arange(0.5,25,1)
for sigma in sigmas:
    print('Sigma = {}'.format(sigma))
    testParam(X_train_norm, y_train, X_val_norm, y_val, rango_parametro = Cs,
          nombre_modelo = 'SVM gaussian', nombre_parametro = 'C', random_state =
    100, gamma = 1/(2* sigma**2))
    print()


# Nos quedamos con sigma = 5 y C = 16.5, donde obtenemos resultados igual de
    buenos.

# In[45]:


C= 16.5
sigma = 5
svm = SVC(kernel = 'rbf', C=C, gamma = 1/(2* sigma**2), random_state = 100)
svm.fit(X_train_norm, y_train)

predicted_train = svm.predict(X_train_norm)
print('Resultados para train')
muestraMetricas(predicted_train, y_train)
print()
predicted_val = svm.predict(X_val_norm)
print('Resultados para validacin')
muestraMetricas(predicted_val, y_val)
print()
predicted_test = svm.predict(X_test_norm)
print('Resultados para test')
muestraMetricas(predicted_test, y_test)


# ### rboles de decisin (EXTRA)

# In[24]:


testParam(X_train_norm, y_train, X_val_norm, y_val, rango_parametro = np.arange
    (2, 10, 1), nombre_modelo = 'arbol decision', nombre_parametro = 'Profundidad'
    )


# In[46]:
```

```
profundidad = 4
tree = DecisionTreeClassifier(criterion = 'entropy', max_depth = profundidad,
    min_samples_split = 2, random_state = 0)
tree = tree.fit(X_train_norm, y_train)

#Metricas con datos entrenamiento
predicted_train = tree.predict(X_train_norm)
print('Resultados para train')
muestraMetricas(predicted_train, y_train)

predicted_val = tree.predict(X_val_norm)
print('\nResultados para eval')
muestraMetricas(predicted_val, y_val)

predicted_test = tree.predict(X_test_norm)
print('\nResultados para test')
muestraMetricas(predicted_test, y_test)

print('\nMatriz de confusion')
plot_confusion_matrix(y_true = y_test, y_pred = predicted_test, normalize=False,
    title='Arbol decision con profundidad = {}'.format(profundidad), cmap=plt.cm.
    Blues)

plt.figure(figsize=(15,15))
plot_tree(tree, filled=True, feature_names = loan_rawdf.columns.values,
          class_names=['Rechazada', 'Aceptada'], rounded=True)
plt.show()


# ## Random forest (extra)

# In[47]:


n_estimators_range = [2,10,25,50,75,100,500]
for n_estimators in n_estimators_range:
  print('\nNmero de rboles:', n_estimators)
  testParam(X_train_norm, y_train, X_val_norm, y_val, rango_parametro = np.arange
    (2, 15, 1),
          nombre_modelo = 'random forest', nombre_parametro = 'Profundidad mxima'
    , n_estimators = n_estimators)


# In[48]:


profundidad = 12
n_estimators = 50
```

```
modelo = RandomForestClassifier(n_estimators = n_estimators, max_depth =
    profundidad, random_state = 0)
modelo = modelo.fit(X_train_norm, y_train)

#Metricas con datos entrenamiento
predicted_train = modelo.predict(X_train_norm)
print('Resultados para train')
muestraMetricas(predicted_train, y_train)

predicted_val = modelo.predict(X_val_norm)
print('\nResultados para eval')
muestraMetricas(predicted_val, y_val)

predicted_test = modelo.predict(X_test_norm)
print('\nResultados para test')
muestraMetricas(predicted_test, y_test)

print('\nMatriz de confusion')
plot_confusion_matrix(y_true = y_test, y_pred = predicted_test, normalize=False,
    title='Random forest con {} rboles y profundidad = {}'.format(n_estimators,
    profundidad), cmap=plt.cm.Blues)


# ### Pruebas con Synthetic Minority Oversampling Technique (SMOT)
# Conseguimos mejorar el recall

# In[49]:


sm = SMOTE(random_state=333, sampling_strategy =  {0: 2712, 1:500})
X_train_res, y_train_res = sm.fit_resample(X_train, y_train.ravel())

print('Nmero de ejemplos de entrenamiento: ', len(y_train_res))
print('Porcentaje de la clase 0 en entrenamiento: ', np.sum(y_train_res == 0) /
    len(y_train_res)*100, '%')
print('Porcentaje de la clase 1 en entrenamiento: ', np.sum(y_train_res == 1) /
    len(y_train_res)*100, '%')


# In[50]:


X_train_norm_res, _ ,_ = normalize_data(X_train_res)
X_val_norm, _ ,_ = normalize_data(X_val)
X_test_norm, _ ,_ = normalize_data(X_test)


# #### Regresin logstica SMOT
```

```
# In[51]:


poly = PolynomialFeatures(3)
X_train_norm_poly = poly.fit_transform(X_train_norm_res)
X_val_norm_poly = poly.fit_transform(X_val_norm)
X_test_norm_poly = poly.fit_transform(X_test_norm)


# In[52]:


testParam(X_train_norm_poly, y_train_res, X_val_norm_poly, y_val, rango_parametro
    = np.arange(0, 10, 0.5),
          nombre_modelo = 'regresion logistica', nombre_parametro = 'Lambda')


# In[53]:


lamda = 9.5
theta = np.zeros(X_train_norm_poly.shape[1])
result = opt.fmin_tnc(func = costeRegu, x0 = theta, fprime = gradienteRegu, args
   =(X_train_norm_poly, y_train_res, lamda))
theta_opt = result[0]

#Metricas con datos entrenamiento
predicted_train = sigmoide(np.matmul(X_train_norm_poly, theta_opt)) >= 0.5
print('Resultados para train')
muestraMetricas(predicted_train, y_train_res)

predicted_val = sigmoide(np.matmul(X_val_norm_poly, theta_opt)) >= 0.5
print('\nResultados para eval')
muestraMetricas(predicted_val, y_val)

predicted_test = sigmoide(np.matmul(X_test_norm_poly, theta_opt)) >= 0.5
print('\nResultados para test')
muestraMetricas(predicted_test, y_test)

print('\nMatriz de confusion')
plot_confusion_matrix(y_true = y_test, y_pred = predicted_test, normalize=False,
   title='Regresion logistica con terminos polinomicos y lambda = {}'.format(
   lamda), cmap=plt.cm.Blues)


# #### Red neuronal SMOT

# In[54]:
```

```
input_size = X_train_norm.shape[1]
lamda = 1
max_iter = 230
num_labels = 2
hidden_size = (65,)
fmin = entrenaRed(X_train_norm_res, y_onehot(y_train_res, num_labels), input_size
    , hidden_size, num_labels, reg = lamda, max_iter = max_iter, random_state =
    100)
predicted_train = y_predicted(X_train_norm_res, input_size, hidden_size,
    num_labels, fmin.x)
print('Resultados para train')
muestraMetricas(predicted_train, y_train_res)
predicted_val = y_predicted(X_val_norm, input_size, hidden_size, num_labels, fmin
    .x)
print('\nResultados para eval')
muestraMetricas(predicted_val, y_val)
predicted_test = y_predicted(X_test_norm, input_size, hidden_size, num_labels,
    fmin.x)
print('\nResultados para test')
muestraMetricas(predicted_test, y_test)


# #### SVM SMOT

# In[55]:


C= 16.5
sigma = 5
svm = SVC(kernel = 'rbf', C=C, gamma = 1/(2* sigma**2), random_state = 100)
svm.fit(X_train_norm_res, y_train_res)

predicted_train = svm.predict(X_train_norm_res)
print('Resultados para train')
muestraMetricas(predicted_train, y_train_res)

predicted_val = svm.predict(X_val_norm)
print('Resultados para validacin')
muestraMetricas(predicted_val, y_val)

predicted_test = svm.predict(X_test_norm)
print('Resultados para test')
muestraMetricas(predicted_test, y_test)


# #### rbol de decision SMOT

# In[56]:
```

```
profundidad = 4
tree = DecisionTreeClassifier(criterion = 'entropy', max_depth = profundidad,
    min_samples_split = 2, random_state = 0)
tree = tree.fit(X_train_norm_res, y_train_res)

#Metricas con datos entrenamiento
predicted_train = tree.predict(X_train_norm_res)
print('Resultados para train')
muestraMetricas(predicted_train, y_train_res)

predicted_val = tree.predict(X_val_norm)
print('\nResultados para eval')
muestraMetricas(predicted_val, y_val)

predicted_test = tree.predict(X_test_norm)
print('\nResultados para test')
muestraMetricas(predicted_test, y_test)

print('\nMatriz de confusion')
plot_confusion_matrix(y_true = y_test, y_pred = predicted_test, normalize=False,
    title='Arbol decision con profundidad = 4', cmap=plt.cm.Blues)


# In[57]:


profundidad = 12
n_estimators = 50
modelo = RandomForestClassifier(n_estimators = n_estimators, max_depth =
    profundidad, random_state = 0)
modelo = modelo.fit(X_train_norm_res, y_train_res)

#Metricas con datos entrenamiento
predicted_train = modelo.predict(X_train_norm_res)
print('Resultados para train')
muestraMetricas(predicted_train, y_train_res)

predicted_val = modelo.predict(X_val_norm)
print('\nResultados para eval')
muestraMetricas(predicted_val, y_val)

predicted_test = modelo.predict(X_test_norm)
print('\nResultados para test')
muestraMetricas(predicted_test, y_test)

print('\nMatriz de confusion')
plot_confusion_matrix(y_true = y_test, y_pred = predicted_test, normalize=False,
```

```
    title='Random forest con {} rboles y profundidad = {}'.format(n_estimators,
    profundidad), cmap=plt.cm.Blues)
```