Doble Grado en
Ingeniería Informática y Matemáticas

Aprendizaje Automático

(E. Computación y Sistemas Inteligentes)

UNIVERSIDAD DE GRANADA

PROYECTO FINAL

Ajuste y selección del mejor predictor a la BBDD de la UCI Default of Credit Card Clients con el apoyo de la libreria Scikit-Learn.

Alberto Estepa Fernández
Email: albertoestep@correo.ugr.es

Carlos Santiago Sánchez Muñoz
Email: carlossamu7@correo.ugr.es

18 de junio de 2020

Índice

1. Definición del problema a resolver y enfoque elegido **2**

2. Elección de los modelos **7**

3. Codificación de los datos de entrada **9**

4. Valoración del interés de las variables para el problema y selección de un subconjunto **10**

5. Normalización de las variables **13**

6. Justificación de la función de pérdida usada **14**

7. Selección de las técnicas paramétricas **15**

8. Selección de parámetros e hiperparámetros **15**

9. Idoneidad de la función regularización **21**

10. Valoración de los resultados **22**

11. Justificar que se ha obtenido la mejor de las posibles soluciones con la técnica elegida y la muestra dada **25**

Referencias **27**

1. Definición del problema a resolver y enfoque elegido

Tenemos entre manos un conjunto de datos de crédito de un banco de Taiwán a fecha de octubre de 2005. El conjunto de datos recoge información de los clientes e indica si se les concede o no un crédito de forma predeterminada. La base de datos está disponible en [1].

El dataset consiste en 24 variables de las cuales una de ellas será la variable a predecir (`default payment next month`), una variable binaria que indica si se le concede el crédito (`1`) o si no se le concede el crédito (`0`). Así podemos afirmar que estamos ante un problema de clasificación binaria. Las otras 23 características serán las variables explicativas del problema que nos ayudaran a predecir la variable objetivo. Explicamos las 23 variables explicativas del conjunto de datos a continuación:

- `LIMIT_BAL`: Cantidad del crédito otorgado (en NT dollar, es decir, nuevo dólar taiwanés). Incluye tanto el crédito al consumidor individual como el crédito (complementario) de su familia.
- `SEX`: Género (`1` = masculino; `2` = femenino).
- `EDUCATION`: Educación (`1` = escuela de posgrado; `2` = universidad; `3` = escuela secundaria; `4` = otros).
- `MARRIAGE`: Estado civil (`1` = casado; `2` = soltero; `3` = otras).
- `AGE`: Edad (en años).
- `PAY_0`: el estado del reembolso en septiembre de 2005.
- `PAY_2`: el estado del reembolso en agosto de 2005.
- `PAY_3`: el estado del reembolso en julio de 2005.
- `PAY_4`: el estado del reembolso en junio de 2005.
- `PAY_5`: el estado del reembolso en mayo de 2005.
- `PAY_6`: el estado del reembolso en abril de 2005.
- `BILL_AMT1`: Cantidad de dinero (en NT dollar) en la cuenta en septiembre de 2005.
- `BILL_AMT2`: Cantidad de dinero (en NT dollar) en la cuenta en agosto de 2005.
- `BILL_AMT3`: Cantidad de dinero (en NT dollar) en la cuenta en julio de 2005.
- `BILL_AMT4`: Cantidad de dinero (en NT dollar) en la cuenta en junio de 2005.
- `BILL_AMT5`: Cantidad de dinero (en NT dollar) en la cuenta en mayo de 2005.

- `BILL_AMT6`: Cantidad de dinero (en NT dollar) en la cuenta en abril de 2005.
- `PAY_AMT1`: Cantidad de dinero (en NT dollar) pagado en septiembre de 2005.
- `PAY_AMT2`: Cantidad de dinero (en NT dollar) pagado en agosto de 2005.
- `PAY_AMT3`: Cantidad de dinero (en NT dollar) pagado en julio de 2005.
- `PAY_AMT4`: Cantidad de dinero (en NT dollar) pagado en junio de 2005.
- `PAY_AMT5`: Cantidad de dinero (en NT dollar) pagado en mayo de 2005.
- `PAY_AMT6`: Cantidad de dinero (en NT dollar) pagado en abril de 2005.

Como vemos tenemos información historial de pagos pendientes, es decir, tenemos los últimos registros de pagos mensuales (de abril a septiembre de 2005) (atributos `PAY_x`, donde `x` $\in \{0, 2, 3, 4, 5, 6\}$. La escala de medición para el estado de reembolso es:

- `-1`: pagado debidamente.
- `1`: retraso en el pago de un mes.
- `2`: retraso en el pago de dos meses.
- ...
- `8`: retraso en el pago de ocho meses.
- `9`: retraso en el pago de nueve meses o más.

También disponemos de información del dinero del que dispuso el cliente en cuenta en los últimos 6 meses (`BILL_AMTx`) y el dinero ya pagado (`PAY_AMTx`), donde `x` $\in \{1, 2, 3, 4, 5, 6\}$.

Definiendo matemática y computacionalmente nuestro problema, es obvio que estamos ante un problema de aprendizaje supervisado ya que, para un conjunto finito de instancias del problema, conocemos la etiqueta que corresponde a cada una de las instancias y a partir de dicho conjunto de instancias, correctamente clasificadas, podemos aprender una función de clasificación que nos indique la clase de cada instancia del dominio. Vemos así que $\mathcal{X}$ es el conjunto de 23 características que incluye el dataset que incluye la información sobre el cliente, es decir:

$$\begin{aligned}\mathcal{X} =& \texttt{LIMIT\_BAL} \times \{1, 2\} \times \{1, 2, 3, 4\} \times \{1, 2, 3\} \times \{18, \ldots, 100\} \times \\ & \times \{-1, \ldots, 9\}^6 \times \texttt{BILL\_AMTi} \times \texttt{PAY\_AMTi}\end{aligned}$$

donde `LIMIT_BAL` $\in [0, +\infty)$, `BILL_AMTi` $\in (-\infty, +\infty)^6$ y `PAY_AMTi` $\in [0, +\infty)^6$.

El conjunto de etiquetas $\mathcal{Y}$ incluye números enteros (`0` o `1`) que indican si se le concede (`1`) o si no se le concede (`0`) el crédito a cada cliente, es decir: $\mathcal{Y} = \{\texttt{0}, \texttt{1}\}$.

La función de objetivo es $f : \mathcal{X} \to \mathcal{Y}$ que para cada instancia $(x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$ verifica $f(x_n) = y_n$.

En la búsqueda de la función objetivo f es necesario fijar la clase de funciones o conjunto de hipótesis $\mathcal{H}$. Más adelante elegiremos $g \in \mathcal{H}$ de modo que $g \approx f$.

Cuando en un problema de clasificación hay una gran cantidad de variables es necesario tener cuidado para que no haya *overfitting* o sobreajuste. Por este motivo vamos a usar una clase de hipótesis lineal con la intención de que el modelo converja correctamente:

$$\mathcal{H}_1 = \left\{ w_0 + \sum_{i=1}^{N} w_i x_i, \quad w \in \mathbb{R}^{N+1} \right\}.$$

Para observar mejor la distribución de las instancias en atributos como `LIMIT_BAL`, `BILL_AMTi` y `PAY_AMTi` se van a construir unas gráficas de barras. Comenzamos con la de `LIMIT_BAL`:

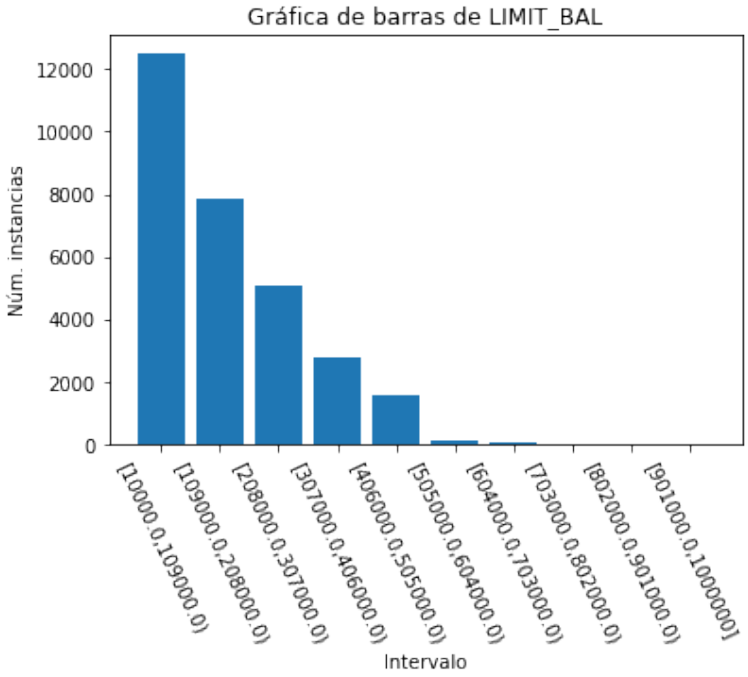
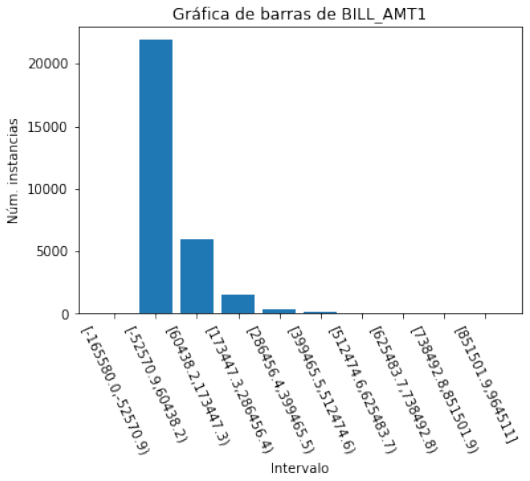


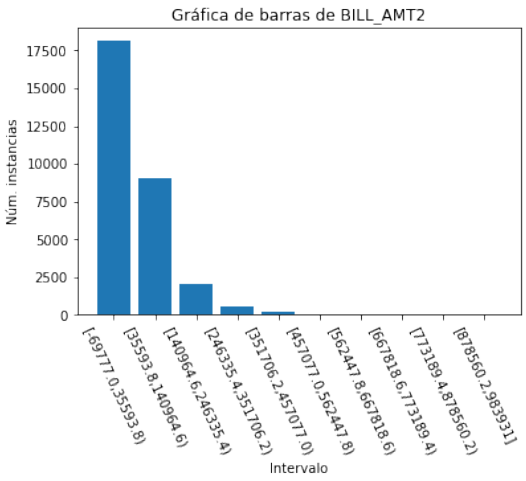
Imagen 1: Atributo `LIMIT_BALL`

Encontramos diez intervalos de la misma longitud repartidos entre el valor mínimo y máximo que alcanza el atributo tratado. Observamos que hay un mayor número de instancias en los primeros intervalos y conforme se va a aumentando la presencia de instancias con un valor en dicho atributo es menor.

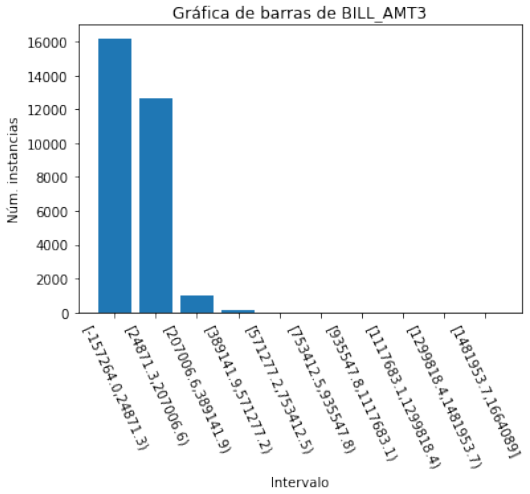
Otro aspecto importante a comentar es que hemos comprobado que disponemos de 30.000 instancias en nuestro conjunto de datos. Cada una de éstas instancias se



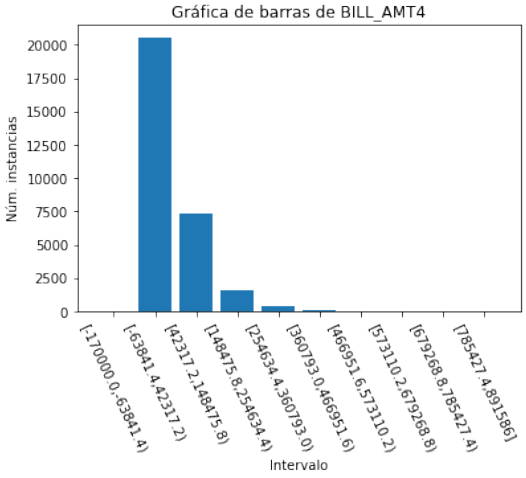
(a) Atributo `BILL_AMT1`



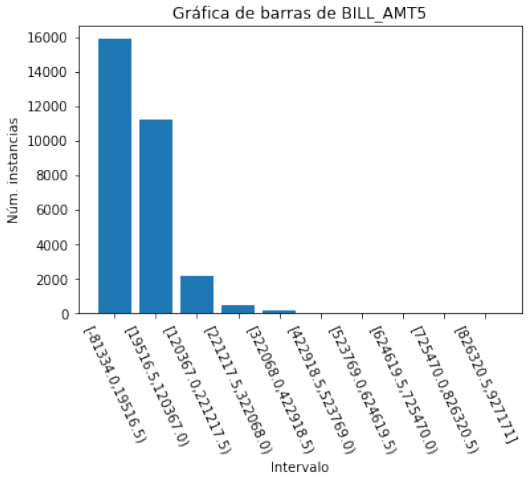
(b) Atributo `BILL_AMT2`



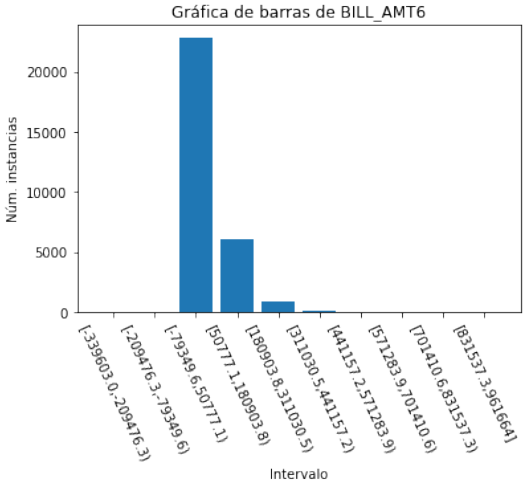
(c) Atributo `BILL_AMT3`



(d) Atributo `BILL_AMT4`

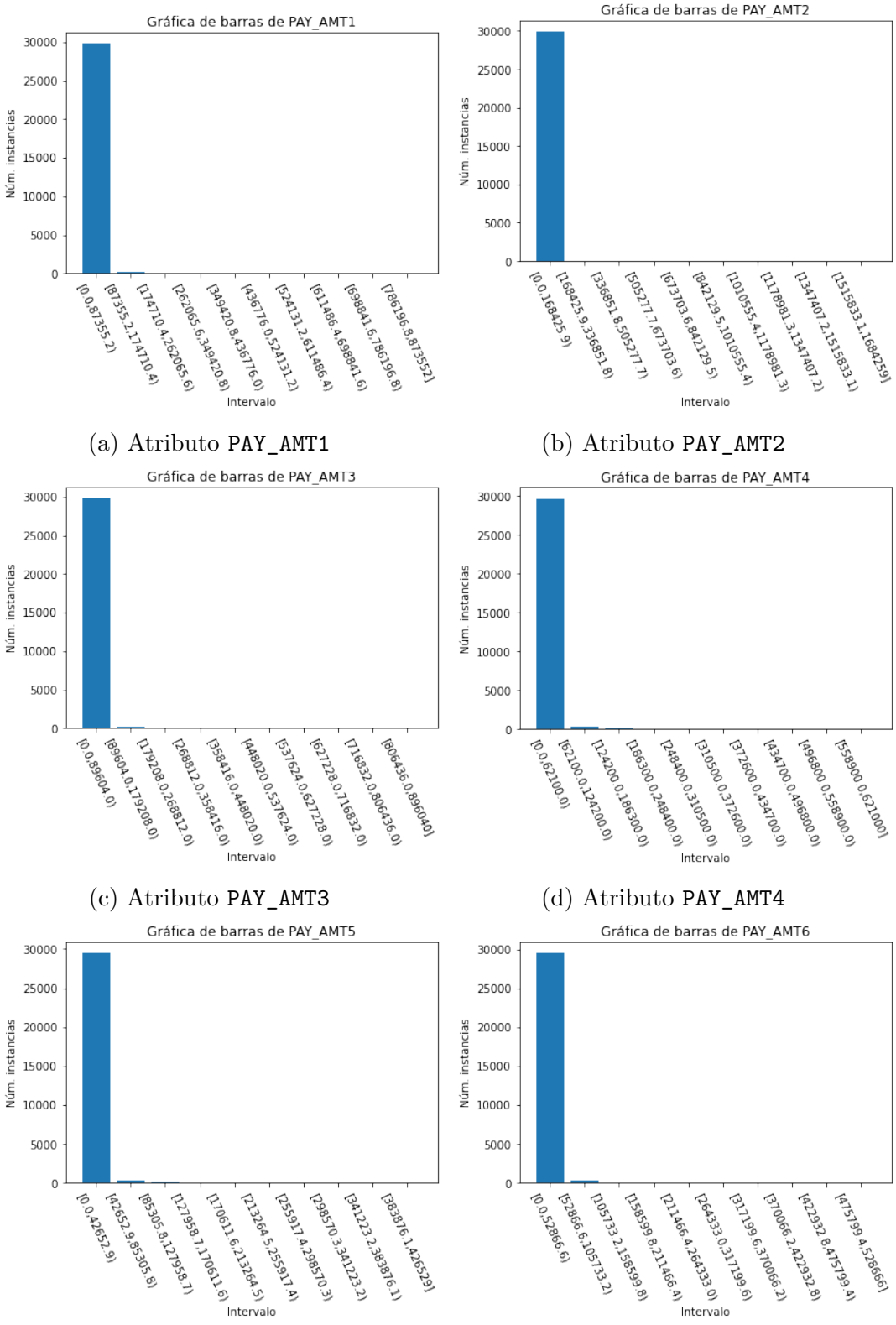


(e) Atributo `BILL_AMT5`



(f) Atributo `BILL_AMT6`

Imagen 2: Gráficas de barras de `BILL_AMTx`



(a) Atributo `PAY_AMT1`

(b) Atributo `PAY_AMT2`

(c) Atributo `PAY_AMT3`

(d) Atributo `PAY_AMT4`

(e) Atributo `PAY_AMT5`

(f) Atributo `PAY_AMT6`

Imagen 3: Gráficas de barras de `PAY_AMTx`

corresponde con un cliente del banco distinto. Analizando el número de instancias que identifican cada una de las dos clases podemos ver que disponemos de 23364 instancias (un 77.88 % del total) de la clase 0 (no se le concede el crédito) y 6636 instancias un 22.12 % del total) de la clase 1 (se le concede el crédito). Aunque esto pueda parecer que las clases están bastante desbalanceadas, hay que recordar la situación en la que estamos: el banco necesita ser restrictivo sobre a que cliente concede un crédito o no.

El banco concederá un crédito a aquellos clientes que puedan devolver con garantías el dinero y es importante, por el beneficio del banco, no conceder créditos a aquellos clientes que no tengan garantías de devolver el crédito. Por esto es normal que las clases mantengan dicha proporción en nuestro conjunto de datos. De esta forma, dado el gran número de instancias totales del conjunto de datos y que el número total de instancias de la clase más pequeña permite aprender de forma suficiente, consideramos que no debemos tratar de forma especial éste tema.

El enfoque elegido para este problema es clasificación. Aunque parezca una decisión obvia lo cierto es que también se puede plantear como un problema de regresión. Un ejemplo de este hecho es el paper [2] en donde se realiza una regresión $Y = AX + B$ en donde se pretende obtener una probabilidad de incumplimiento del pago.

El documento citado examina seis técnicas de clasificación en la minería de datos y compara el rendimiento de la clasificación y la precisión predictiva entre ellas. Las redes neuronales artificiales realizan la clasificación con mayor precisión que los otros cinco métodos. Se obtienen coeficientes de determinación R^2 muy cercanos 1.

2. Elección de los modelos

Nuestro problema es de clasificación binaria y hay varios modelos que se pueden usar. Se han elegido tres modelos, uno de ellos lineal. Veamos:

(A) Regresión Logística

El primer modelo elegido es el de Regresión Logística, que sabemos que es un modelo lineal.

De sobra conocemos que el modelo de regresión logística aplica una una función logística (denominada sigmoide) a los datos, es decir aplica $\sigma(\omega^T x)$, donde $\sigma(z) \in [0, 1]$ y

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

Así la estimación de los parámetros ω del modelo se efectúa por medio de del método de estimación por máxima verosimilitud.

Como tenemos un problema de clasificación binaria, podemos usar dicho método para comprobar si las clases son linealmente separables (en dicho caso obtendremos resultados bastante óptimos con dicho modelo). Sabemos que es más robusto al ruido que otros modelos de clasificación lineales. Además la regresión logística resulta bastante eficaz al enfrentarnos a un gran volumen de datos, como es nuestro caso. Por todo esto y ya que en la práctica anterior obtuvimos bastante buenos resultados con él, nos hemos decantado por dicho modelo frente a otros modelos lineales.

Usaremos por tanto la clase `LogisticRegression` de la librería Sklearn de Python.

(B) Máquina de Soporte de Vectores (SVM).

El siguiente método escogido es una Máquina de Soporte de Vectores (SVM) con un núcleo no lineal. Esto permitirá precisar una buena clasificación si nuestros datos no son linealmente separables.

Por un lado, los SVMs no dependen de la dimensionalidad de las muestras, lo cual para nuestro problema, al que aplicaremos un aumento de dimensionalidad puede ser útil.

A pesar que SVM era adecuado para conjuntos con un número bajo de instancias por su baja eficiencia $\mathcal{O}(N^3)$, sabemos que un SVM busca un hiperplano óptimo de separación entre las clases. Puede ser interesante comparar dicho modelo para nuestro problema.

Usaremos por tanto la clase `SVC` de la librería Sklearn de Python.

(C) Random Forest.

El último método escogido para la comparación es Random Forest. Se basa en tener varios árboles de decisión entrenados con una muestra escogida por *bootstrapping*. Cada uno de dichos árboles usará solo unos ciertos predictores para predecir el resultado. [7]

Dicho modelo no espera que las dependencias sean lineales o incluso que los atributos interactúen de forma lineal.

Además sabemos que se comportan de forma eficaz ante los espacios de alta dimensión (anticipándonos a los acontecimientos, vamos a aumentar la dimensionalidad del dataset en las secciones posteriores), así como con un gran número de ejemplos de entrenamiento, como es nuestro caso. Además en general, al no tener muchos hiperparámetros que ajustar, suele funcionar muy bien.

Usaremos por tanto la clase `RandomForestClassifier` de la librería Sklearn de Python.

3. Codificación de los datos de entrada

Los datos se han proporcionado en un fichero `.xls` que es el formato de archivo implementado por las versiones antiguas de Microsoft Excel. Usaremos la estructura `dataframe` proporcionada por Python para el tratamiento de los datos. Para su lectura hemos hecho uso de la función `read_excel` de la librería `Pandas` de Python. Con dicha función le indicamos también que no considere la primera fila del dichero, pues no proporciona información al conjunto de datos, así como que tome como columna índice la denominada `ID`. De esta forma no hemos necesitado modificar el fichero original proporcionado por la UCI.

```
import pandas as pd
df = pd.read_excel('datos/default of credit card clients.xls',
                   skiprows = 1, index_col = 'ID')
```

En la sección inicial (1) estudiamos los dominios de las variables de nuestro problema de acuerdo a la documentación proporcionada por la UCI, sin embargo nos hemos encontrado que no se corresponden con los datos encontrados en el dataset. Veamos los problemas en forma de 'outliers' encontrados, uno a uno:

- `EDUCATION`: Recordamos que la variable que indicaba los estudios del cliente se encontraba en un rango 1 y 4, siendo un valor natural, es decir, `EDUCATION` $= \{1, 2, 3, 4\}$. Sin embargo al estudiar el conjunto de datos nos hemos encontrado con que existen 280 instancias del valor 5, 51 instancias del valor 6 y 14 instancias del valor 0. Al ser tan pocas instancias y no resultan significativas en el conjunto de datos (aproximadamente el 1 % de las instancias) hemos considerado incluirlos en el valor 4, que recordemos que significaba 'Otros niveles de estudios' (de los contemplados por los valores 1, 2 y 3) ya que es lógico que estén ahí incluidas.

- `MARRIAGE`: Recordamos que dicha variable indicaba el estado civil del cliente y sus posibles valores eran 1, 2 o 3. Sin embargo, hemos encontrado 54 instancias del valor 0 (menos del 0.2 % de las instancias). De nuevo, hemos considerado incorporarlos al valor 3, que indica 'Otro estado civil', basándonos en el mismo razonamiento que en la variable `EDUCATION`.

- `PAY_x` (donde `x` $\in \{0, 2, 3, 4, 5, 6\}$). Este conjunto de atributos ha sido el más problemático, pues el dominio no se corresponde con el proporcionado en la documentación: se indicaba que los posibles valores eran $\{-1, 1, 2, \ldots, 9\}$ y nos hemos encontrado que los valores encontrados son $\{-2, -1, 0, 2, 3, 4, 5, 6, 7, 8\}$. Como no existe una relación lógica entre lo que indica la documentación y los valores encontrados hemos decidido no tratarlos de forma especial, pues eliminar dichas instancias problemáticas podría incluir el eliminar información clave del conjunto de datos a la hora de entrenar los diferentes algoritmos.

Por otro lado, como comentamos en la sección inicial (1), todas las variables son numéricas, por lo que no tenemos variables categóricas a tratar.

Por último, respecto a ésta sección, aunque podíamos tratarla en la sección 5, hemos decidido binarizar cada una de las variables discretas multivaluadas. La motivación de realizar dicha transformación es que sabemos que los algoritmos a tratar trabajan mucho mejor con variables binarias (`0` y `1`) que con variables cuyo dominio abarca un gran número de valores, además de que el costo computacional de dicha transformación para nuestro conjunto de datos no será excesivamente grande. Para dicha transformación hemos recurrido a la función `get_dummies` de la librería `Pandas`, indicándole las columnas que convertiremos en binarias (que como hemos comentado serán aquellas que contemplan valores discretos en un rango pequeño, para no sobrecargar después computacionalmente los algoritmos).

```
df = pd.get_dummies(df, columns=['SEX', 'EDUCATION', 'MARRIAGE', 'PAY_0',
                                 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6'])
```

Por último renombramos las variables resultantes (por ejemplo `SEX` se ha transformado en las variables `Hombre` y `Mujer`; y así con las demás transformaciones). Así damos por concluída la codificación de los datos de entrada para hacerlos útiles a los algoritmos, teniendo 87 variables explicativas en nuestro conjunto de datos.

4. Valoración del interés de las variables para el problema y selección de un subconjunto

En la sección 1 se estudió el significado de cada una de las variables y durante la sección 3 se mostró como hemos transformado cada una de dichas variables para el correcto funcionamiento de los siguiente algoritmos que usaremos. Para ésta sección usaremos las variables ya transformadas para estudiar la selección de variables de interés para nuestro problema.

El primer paso que daremos es eliminar aquellas variables multicolineadas. Recordemos que la multicolinealidad entre variables se produce cuando existen relaciones lineales entre dichas variables, lo que implica que se aporte información repetida al problema y puede provocar un sobrecoste computacional del modelo.

Nos basaremos en el coeficiente de Pearson para calcular las correlaciones entre variables. Recordemos que si este coeficiente es igual a 1 o -1 (o cercano a estos valores) significa que las variables sufren multicolinealidad (aproximada, si no es 1 o -1; o exacta, si alcanza dichos valores). Para ello usaremos la matriz de correlaciones (no se incluye en los ejes el nombre de todas las variables por la dificultad visual de dicha acción, pero la matriz, obviamente, esta calculada sobre todos los atributos):

Vemos que existe una gran correlación entre las variables `BILL_AMTx` donde `x` $\in \{1, 2, 3, 4, 5, 6\}$. Además se observa una gran correlación inversa entre las variables `Mujer` y `Hombre` o entre las variables `Casado` y `Soltero`, entre otras.

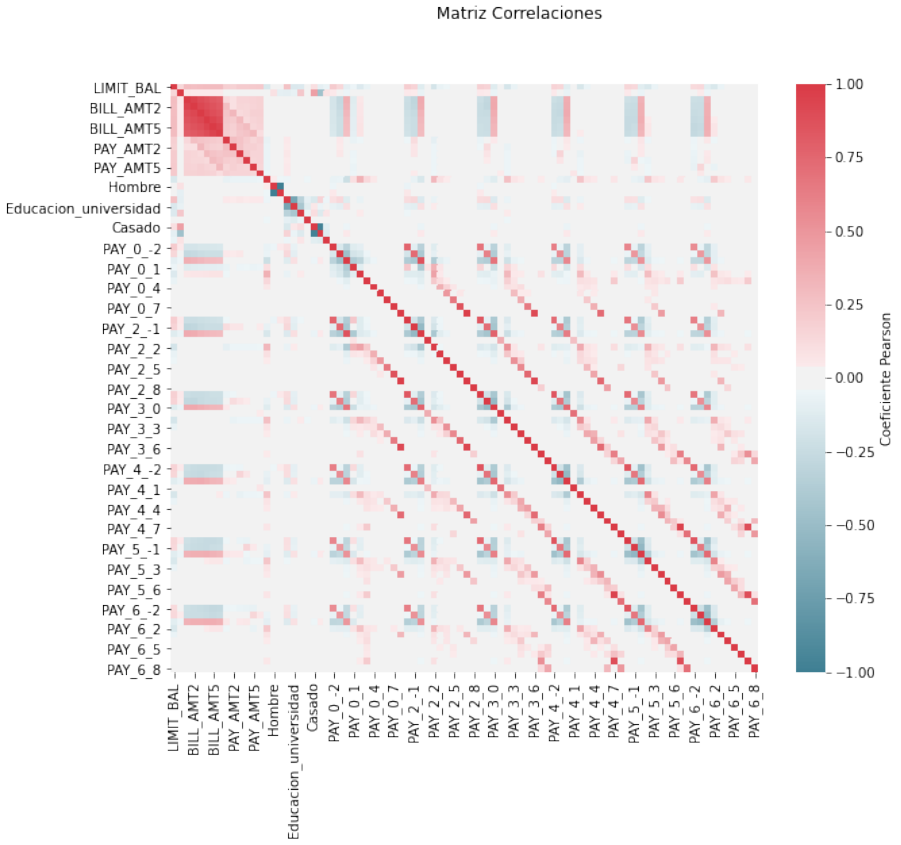


Imagen 4: Matriz de correlaciones

Vamos a estudiar aquellas variables con multicolinealidad exacta (que tienen un coeficiente de Pearson en valor absoluto de 1, pues mediante una combinación lineal podemos obtener la otra variable, por lo que no es necesario su uso). Para ésta acción hemos implementado dos funciones en Python: la primera que calcula las variables que superen un valor de correlación dado; y la segunda que imprima la matriz de correlación ampliada ordenada de mayor a menor correlación de Pearson, de cada una de las variables anteriores.

```
# Seleccionamos las columnas que son multicolineales
def columnas_correladas(dataframe, coeficiente = 1):
    # Matriz de correlación en valor absoluto
    corr_matrix = dataframe.corr(method = 'pearson').abs()

    # Seleccionamos la matriz triangular superior de la matriz de correlación anterior
    upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape),
                                      k=1).astype(np.bool))

    # Buscamos la columnas con una correlacion dada con alguna otra
    to_drop = [column for column in upper.columns
                      if any(upper[column] >= coeficiente)]
    return to_drop
```

```
# Matriz de correlación en valor absoluto ampliada por columnas
def matriz_correlacion_ampliada(dataframe, columnas, k = 3):
# k: Número de variables a mostrar.
    # Matriz de correlación en valor absoluto
    corr_matrix = dataframe.corr(method = 'pearson').abs()
    for i in columnas:
        cols = corr_matrix.nlargest(k, i)[i].index
        cm = np.abs(np.corrcoef(dataframe[cols].values.T))
        ax = plt.axes()
        sns.set(font_scale = 1.25)
        hm = sns.heatmap(cm, cbar = True, annot = True, square = True,
                         fmt = '.2f', annot_kws = {'size': 10},
                         yticklabels = cols.values,
                         xticklabels = cols.values,
                         cbar_kws={'label': 'Coeficiente Pearson \
                                             en valor absoluto'})
        ax.set_title('M. de corr. ampliada de {}'.format(i))
        plt.show()
```

De esta forma, si realizamos las siguientes ejecuciones:

```
to_drop = columnas_correladas(dataframe = df, coeficiente = 1)
matriz_correlacion_ampliada(dataframe = df, k = 3, columnas = to_drop)
```

Obtenemos la siguiente salida:

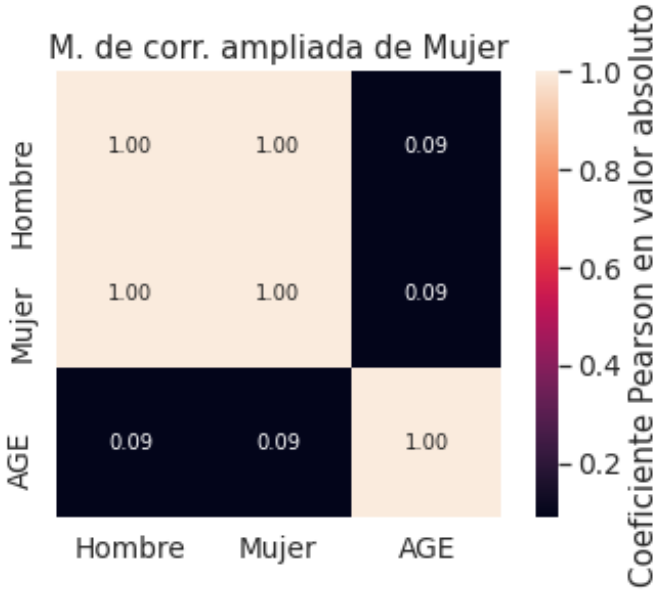


Imagen 5: Matriz de correlaciones en valor absoluto ampliada entre `Mujer` y `Hombre`

Vemos que las variables `Mujer` y `Hombre` están multicolineadas de forma exacta entre sí (coeficiente de Pearson 1 en valor absoluto). Por tanto, apoyándonos en lo anteriormente descrito, podemos eliminar una de las dos variables (eliminaremos la variable `Hombre`, por ejemplo).

Si relajamos el coeficiente de correlación de la función anterior (4) vemos que la siguiente pareja de variables multicolineadas es `Casado` y `Soltero` con un coefi-

ciente de Pearson en valor absoluto de 0.98. Consideramos que, como el conjunto de variables no es demasiado grande (86 variables explicativas al eliminar la variable `Hombre`) y como queremos explicar toda la varianza posible de nuestro conjunto de datos, preferimos mantener dichas variables en nuestro conjunto de datos para no perder nada de información que puede ser útil para los algoritmos que vamos a utilizar.

De esta forma, consideramos que las demás variables son de interés para nuestro problema y que no necesitamos seleccionar un subconjunto más pequeño de entre las 87 variables de estudio).

5. Normalización de las variables

Tal y como comentamos en la sección 3, hemos binarizado cada una de las variables discretas multivaluadas con menos de 10 valores en su dominio. El rango de las nuevas variables obtenidas solo consiste en $\{0, 1\}$, por lo que no es necesario normalizarlas para nuestro algoritmo.

Sin embargo existen variables que no hemos tratado todavía: `LIMIT_BAL`, `AGE`, `BILL_AMTx` y `PAY_AMTx` donde `x` $= \{1, 2, 3, 4, 5, 6\}$. Dichas variables poseen valores enteros en un rango bastante amplio (para más información consulte la sección 1 donde se concretan los diferentes dominios de las variables). Para evitar problemas entre las escalas de las diferentes variables vamos a normalizar los datos de dichas variables entre 0 y 1.

Es de sobra conocido que el objetivo de la normalización es cambiar los valores de las columnas numéricas del conjunto de datos para usar una escala común, sin distorsionar las diferencias en los intervalos de valores ni perder información. Además, la normalización también es necesaria para que algunos algoritmos modelen los datos correctamente. La normalización evita los problemas anteriormente expuestos mediante la creación de nuevos valores que mantienen la distribución general y las relaciones en los datos de origen [4].

Para ello hemos usado la clase de Sklearn `MinMaxScaler` [10] que transforma las características escalando cada una a un rango dado (usaremos el rango por defecto $[0, 1]$. La transformación está dada por:

$$X\_std = (X - X.min(axis = 0))/(X.max(axis = 0) - X.min(axis = 0))$$

$$X\_scaled = X\_std * (max - min) + min$$

donde $[min, max] = [0, 1]$ en nuestro caso.

De esta forma tenemos normalizadas todas las variables de nuestro conjunto de datos.

6. Justificación de la función de pérdida usada

La función de pérdida usada dependerá de cada modelo elegido:

- Para la regresión logística (A) usaremos la función de pérdida cross-entropy (o entropía cruzada) que sabemos que mide la diferencia entre probabilidad pronosticada cuando difiere de la etiqueta real, lo que matemáticamente en nuestro problema (donde las clases son 0 y 1) se expresa como [5]:

$$CE = -\sum_i (y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

  donde y_i es la etiqueta de la clase ($y_i \in \{0, 1\}$) de la instancia i y p_i la probabilidad predicha de que pertenezca a la clase y_i ($p_i = \sigma(w_0 + \sum_{j=1}^{N} w_j x_{ij})$).

- Para el modelo (B) sabemos que para un clasificador SVM lineal se tiene la función de pérdida Hinge, definida como [6]:

$$Hinge_{lineal} = \sum_i \text{máx}(0, 1 - y_i(w_0 + \sum_{j=1}^{N} w_j x_{ij})).$$

  Esta es la función de pérdida que nos permite maximizar el margen que dejamos al separar ambas clases.

  Sin embargo, como vamos a usar el kernel $Gaussian - RBF$ la función de pérdida es distinta a la del kernel lineal.

$$Hinge_{no-lineal} = \sum_i \text{máx}(0, 1 - y_i(w_0 + \sum_{j=1}^{N} w_j f_i(x_{ij})))$$

  donde notamos que hemos introducido la $f_i()$, que es el la función del kernel en la característica i, que en nuestro caso es $Gaussian - RBF$, como hemos comentado. [8]

- Para el clasificador Random Forest (C) usaremos la función de pérdida: 0-1 *loss* o *zero-one loss*.

  Aunque existen otras variantes como el error cuadrático esta es comúnmente usada. Básicamente para un dato $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ la función de pérdida es:

$$L(y_i, f(x_i)) = \begin{cases} 0 & si \quad y_i = f(x_i) \\ 1 & si \quad y_i \neq f(x_i) \end{cases}.$$

  Esta función muestra la pérdida en un punto pero si queremos ver la pérdida de todos los puntos:

$$\sum_i L(y_i, f(x_i)).$$

7. Selección de las técnicas paramétricas

En esta sección estudiaremos la técnica de ajuste de los datos escogida para el modelo lineal (paramétrico). Es decir estudiaremos la técnica de ajuste del método de regresión logística escogido.

Por un lado, y anticipándonos a los acontecimientos de la sección 9, aplicaremos una regularización ℓ_1 al modelo. Esto es importante, pues algunas técnicas de ajuste (el método del Gradiente Conjugado de Newton (`newton-cg` en Sklearn), el algoritmo L-BFGS (`lbfgs` en Sklearn) y el Gradiente Estocástico Promedio (*Stochastic Average Gradient* y `sag` en Sklearn, que es la variante del SGD para el modelo de regresión logística de Sklearn)) son incompatibles con dicha regularización ya que al introducir ℓ_1 a la función de coste, ésta ya no es derivable en 0.

Entre el *Stochastic Average Gradient Aggregation* (`saga` en Sklearn) y el descenso coordinado proporcionado por por la librería `liblinear`, hemos escogido la técnica `saga` por recomendación de la documentación de Sklearn para la clase `LogisticRegression` [5], ya que tanto el número de instancias como el número de atributos es bastante elevado en nuestro conjunto de datos y dicha técnica precisamente permite una convergencia más rápida en éstas circunstancias.

Anotamos aunque existen otras técnicas que las discutidas anteriormente, hemos escogido entre las 5 técnicas que nos permite elegir la librería de Sklearn, como era pedido por el guión de prácticas.

8. Selección de parámetros e hiperparámetros

Vamos a usar la validación cruzada para ajustar los hiperparámetros de los modelos. Sabemos que, al aplicar la validación cruzada, el conjunto de datos de entrenamiento se divide en grupos de igual tamaño. Una vez realizada la partición se procede a entrenar el modelo una vez por cada uno de los grupos. Utilizando todos los grupos menos el de la iteración para entrenar y este para validar los resultados. Una diferencia importante con la validación fuera de muestra es que en esta ocasión se entrena y valida con todos los datos, cambiando el conjunto utilizado para la validación en cada iteración. Así es posible identificar si los modelos son inestables o estables, es decir, si el resultado depende de los datos utilizados o no. En caso de que los resultados dependan de los datos utilizados, el modelo posiblemente estará siendo siendo sobreajustado, indicando que el modelo empleado dispone de demasiados grados de libertad.

Como hemos discutido en la sección 2, usaremos tres clasificadores: el primer modelo elegido (A) es `LogisticRegression` de la librería Sklearn [11]. Los parámetros del clasificador son:

- `penalty='l1'`: Regularización Lasso (ℓ_1). Explicado en la sección 9.

- `tol = 1e-4`: Indica la precisión de la solución. Puede considerarse un hiperparámetro, pero sabemos que cuanto más pequeño sea dicho valor (siempre positivo), mejores resultados obtendremos (a costa de un coste computacional mayor). Sin embargo, es una sucesión convergente y las mejoras a partir de dicho valor (10^{-4}) son insignificantes, por eso lo mantenemos fijado a ese valor.

- `fit_intercept = True`: Especifica si se debe agregar (en nuestro caso sí) una constante (también conocida como sesgo o intercepción) a la función de decisión.

- `class_weight = None`: Pesos asociados con cada clase. En nuestro caso, indicamos que todas las clases tienen un peso uno, pues no damos prioridad a ninguna de ellas.

- `random_state = None`: Sirve para controlar el generador de números aleatorios utilizado en la técnica `saga`. Nosotros usaremos la instancia de estado aleatorio global de numpy.random, por defecto.

- `solver = 'saga'`: Algoritmo a utilizar en el problema de optimización (el algoritmo es Stochastic Average Gradient Aggregation). Explicado en la sección 7.

- `max_iter = 1000`: Número máximo de iteraciones del algoritmo de optimización. Se ha probado que es el número correcto de iteraciones necesarias para que el algoritmo de optimización converja.

- `multi_class='auto'`: Para indicarle que el modelo se ajuste a clasificación binaria y no a clasificación multinomial.

- `warm_start=False`: Indica si, al ajustar los hiperparámetros del estimador repetidamente en el mismo conjunto de datos, reutilizamos aspectos del modelo aprendidos del valor del parámetro anterior, ahorrando tiempo. Sin embargo, para proporcionar la misma aleatoriedad al entrenamiento de todos los modelos, indicamos el valor `False` de éste parámetro de la clase.

- `C`: Realizaremos validación cruzada para estimar el mejor valor de éste hiperparámetro, que como hemos dicho en la sección 9 mide la influencia de la complejidad del modelo.

- Los demás valores parámetros no son compatibles con los usados y por esto solo es posible su valor por defecto, por eso no se han mencionado.

El segundo clasificador era el modelo (B) que es `SVC` de la librería Sklearn [12]. Los parámetros del clasificador son:

- `kernel = 'rbf'`. El tipo de núcleo que se utilizará. Usaremos el kernel *Radial Basis Function* dado por

$$k(x, \hat{x}) = -\gamma \exp \|x - \hat{x}\|^2$$

- `shrinking = True`. Este parámetro especifica que deseamos una heurística de reducción utilizada en su optimización de la SVM, que se utiliza en la optimización mínima secuencial.

- `probability = False`. Este parámetro indica si el modelo devuelve para cada predicción, el vector de probabilidades de pertenecer a cada clase de la variable de respuesta. No necesitamos ésto para nuestro entrenamiento.

- `tol = 1e-2`: Indica la precisión de la solución. Puede considerarse un hiperparámetro, pero sabemos que cuanto más pequeño sea dicho valor (siempre positivo), mejores resultados obtendremos (a costa de un coste computacional mayor). Sin embargo, es una sucesión convergente y las mejoras a partir de dicho valor (10^{-2}) son insignificantes, por eso lo mantenemos fijado a ese valor. Un mayor valor provoca un excesivo coste computacional.

- `cache_size = 200`: Especifica el tamaño de la memoria caché del núcleo (en MB). Dejamos el valor por defecto de Sklearn ya que no necesitamos mayor tamaño para nuestro entrenamiento.

- `class_weight = None`: Pesos asociados con cada clase. En nuestro caso, indicamos que todas las clases tienen un peso uno, pues no damos prioridad a ninguna de ellas.

- `max_iter = 5000`: Límite estricto en las iteraciones dentro del solucionador. Se ha comprobado que a partir de dichas iteraciones apenas se aprecia mejora.

- `decision_function_shape`: Al ser clasificación binaria, se ignora dicho parámetro.

- `break_ties`: Al ser clasificación binaria, se ignora dicho parámetro.

- `random_state`: Ignorado al indicar textttprobability = False.

- `gamma = 'scale'`: Es el coeficiente γ del kernel `rbf`. Usaremos el valor `scale` que indica que $\gamma = 1/(n\_features * X.var()) = 1/(87 * X.var())$ donde $X.var()$ es la varianza de la característica. Tras numerosas pruebas hemos comprobado que dicho valor es el más correcto para el clasificador.

- `C`: Realizaremos validación cruzada para estimar el mejor valor de éste hiperparámetro, que como hemos dicho en la sección 9 mide la influencia de la complejidad del modelo.

El último clasificador es el modelo (C) que es `RandomForestClasiffier` de la librería Sklearn [13]. Los parámetros del clasificador son:

- `n_estimators`: Indica el número de árboles en el bosque. Es un hiperparámetro del modelo.

- `criterion = 'gini'`. Este parámetro es el criterio que necesita el algoritmo para hacer divisiones estratégicas, ya que, la creación de subnodos en el árbol incrementa la homogeneidad de los subnodos resultantes, es decir, la pureza del nodo se incrementa respecto a la variable objetivo. Los criterios admitidos son 'gini' para la impureza de Gini y 'entropía' para la ganancia de información.

  El *índice Gini* [14] se utiliza para atributos con múltiples valores discretos o valores continuos (precisamente como los que tenemos en `LIMIT_BAL`, `BILL_AMTx` o `PAY_AMTx` donde `x` $= \{1, 2, 3, 4, 5, 6\}$). Esta función de coste mide el grado de impureza de los nodos, es decir, cuán desordenados o mezclados quedan los nodos una vez divididos. El objetivo es minimizar dicho índice.

  Por otra parte la *ganancia de información* [14], se utiliza para atributos categóricos o atributos discretos con pocos valores. Este criterio intenta estimar la información que aporta cada atributo basado en la *teoría de la información*. Al obtener la medida de entropía de cada atributo, podemos calcular la ganancia de información del árbol y deberemos maximizar esa ganancia.

  Debido a que las variables mas importantes para nuestro problema se corresponde con las que el índice Gini resulta más eficaz, hemos decidido decantarnos por dicho parámetro.

- `max_depth = None`: Indica la profundidad máxima del árbol. Con `None`, los nodos se expanden hasta que todas las hojas sean puras o hasta que todas las hojas contengan menos de `min_samples_split` muestras.

- `min_samples_split = 5`: Es el número mínimo de muestras necesarias para dividir un nodo interno. Usamos dicho valor en vez del dado por defecto para evitar que sobreajuste.

- `min_samples_leaf = 1`: El número mínimo de muestras necesarias para estar en un nodo hoja. Usamos el valor por defecto ya que no tenemos motivos para modificarlo.

- `min_weight_fraction_leaf = 0.0`: Especifica la fracción mínima ponderada de la suma total de pesos requerida para estar en un nodo hoja. Indicamos que todas muestras tienen el mismo peso.

- `max_features = 'auto'`: Especifica la cantidad de características a considerar cuando se busca la mejor división. En nuestro caso $max\_features = \sqrt{n\_features} = \sqrt{87}$, redondeado.

- `max_leaf_nodes = 2000`: Indica el límite de nodos hojas. Tras varias comprobaciones hemos comprobado que dicho valor para evitar que sobreajuste.

- `min_impurity_decrease = 0.0`: Un nodo se dividirá si esta división induce una disminución de la impureza mayor o igual a este valor.

- `min_impurity_split`: Atributo en desuso desde la versión 0.19.

- `bootstrap = True`: Las muestras de bootstrap se usan al construir árboles.

- `oob_score = False`: Indicamos que las muestras de *out of bag* (OOB) no se usen para estimar la precisión de la generalización.
- `n_jobs = None`: Indicamos que no ejecute trabajos en paralelo.
- `random_state = None`: Sirve para controlar el generador de números aleatorios utilizado en el *bootstrapping* de las muestras utilizadas al construir árboles. Nosotros usaremos la instancia de estado aleatorio global de `numpy.random`, por defecto.
- `warm_start = False`: Indica si, al ajustar los hiperparámetros del estimador repetidamente en el mismo conjunto de datos, reutilizamos aspectos del modelo aprendidos del valor del parámetro anterior, ahorrando tiempo. Sin embargo, para proporcionar la misma aleatoriedad al entrenamiento de todos los modelos, indicamos el valor `False` de éste parámetro de la clase.
- `class_weight = None`: Pesos asociados con cada clase. En nuestro caso, indicamos que todas las clases tienen un peso uno, pues no damos prioridad a ninguna de ellas.
- `ccp_alpha = 0.0001`: Es el parámetro de complejidad utilizado para poda mínima de complejidad de costo. Indicamos un valor pequeño, pero no cero, para evitar que sobreajuste.
- `max_samples = None`: Con `None` indicamos que use todas las instancias del conjunto de datos, ya que a priori no sabemos si escogiendo un subconjunto podrá empeorar nuestra predicción.

Para la estimación de los hiperparámetros se ha usado `GridSearchCV`. Dicha clase realiza una búsqueda exhaustiva sobre los parámetros especificados de un estimador. Utiliza validación cruzada en el procedimiento. Algunas de sus funcionalidades más básicas son ajustarse a los datos y predecir otros (funciones `fit` y `predict`).

Los hiperparámetros a estimar con `GridSearchCV` son `C` de SVM, `C` de regresión logística y `n_estimator` de Random forest.

El parámetro de regularización `C`, tanto de SVM como de la regresión logística, le indica a la optimización cuánto desea evita clasificar mal en cada ejemplo de entrenamiento.

Un valor alto elegirá en la optimización un hiperplano de margen más pequeño, por lo que la tasa de clasificación de errores de datos de entrenamiento será más baja. Por otro lado, si el valor es bajo, entonces el margen será grande, incluso si resultan datos de entrenamiento mal clasificados [15].

Para conseguir una optimización precisa de dichos hiperparámetros con un grado de precisión de dos cifras o decimales se ha seguido el siguiente procedimiento: en primer lugar una búsqueda inicial en un rango de ordenes de magnitud

$[10^{-2}, 10^{-1}, .., 10^{2}]$ y una vez encontrado el intervalo continuar con una búsqueda dicotómica. Este procedimiento debe converger en pocos pasos.

De manera análoga, pero cambiando el rango de valores posibles, hemos tratado de estimar el correcto valor del hiperparámetro `n_estimators` del clasificador Random Forest. Recordemos que dicho hiperámetro indica el número de árboles que genera nuestro clasificador. El rango inicial de valores a buscar será $[10, 30, 50, 70, 90, 110]$.

Para realizar dicha búsqueda dicotómica hemos implementado dos funciones: por un lado para estimar el intervalo donde se encuentra el valor correcto del hiperparámetro, `busqueda_inicial`; y por otro, una vez que tenemos el intervalo concreto, para realizar la búsqueda dicotómica para valores reales, denominado `busqueda_dicotomica` (aunque en este caso también hemos realizado una búsqueda dicotómica para valores enteros, `busqueda_dicotomica_entera`). Incluimos el código a continuación:

```
# Código de la búsqueda inicial de los parámetros a partir de un intervalo proporcionado grande

def busqueda_inicial(X_train, y_train, pipe, param_name, params, scor, crossval=5):
    # Cross-validation para elegir hiperparámetros
    grid = GridSearchCV(pipe, params, scoring=scor, cv=crossval)
    grid.fit(X_train, y_train)

    lis = np.array(grid.cv_results_['mean_test_score'])
    pos1 = np.where(lis==np.max(lis))[0][0]
    print(lis)
    print(pos1)

    if(pos1==len(lis)-1):
        izq = grid.cv_results_['params'][pos1-1][param_name]
        der = grid.cv_results_['params'][pos1][param_name]
    else:
        if(pos1==0):
            izq = grid.cv_results_['params'][0][param_name]
            der = grid.cv_results_['params'][1][param_name]
        else:
            if(grid.cv_results_['mean_test_score'][pos1-1] >
                    grid.cv_results_['mean_test_score'][pos1+1]):
                izq = grid.cv_results_['params'][pos1-1][param_name]
                der = grid.cv_results_['params'][pos1][param_name]
            else:
                izq = grid.cv_results_['params'][pos1][param_name]
                der = grid.cv_results_['params'][pos1+1][param_name]
    return izq, der, grid
```

El de la búsqueda dicotómica para valores reales es el siguiente. En él se ha parametrizado el parámetro a optimizar.

```
# Código búsqueda parámetros con grado de precisión de dos decimales
def busqueda_dicotomica(X_train, y_train, pipe, param_name, izq, der, scor,
                            crossval=5):
    while(abs(der-izq)>=0.01):
        mid = (izq+der)/2
        params = {param_name: [izq, mid, der]}
        izq, der, grid = busqueda_inicial(X_train, y_train, pipe,
                param_name, params, scor)
```

```
    return grid
```

Por último pero no menos importante, encontramos el código para el caso entero. Es muy similar al anterior. Un detalle importante es el cambio en la tolerancia, ahora es entera.

```
# Código búsqueda parámetros con grado de precisión de 1 entero
def busqueda_dicotomica_entera(X_train, y_train, pipe, param_name, izq, der, scor,
                crossval=5):
    while(abs(der-izq)>1):
        mid = int((izq+der)/2)
        izq = int(izq)
        der = int(der)
        params = {param_name: [izq, mid, der]}
        izq, der, grid = busqueda_inicial(X_train, y_train, pipe,
                param_name, params, scor)
    return grid
```

Los mejores clasificadores obtenidos son, después de estimar los hiperparámetros, los siguientes: `LogisticRegression` con `C = 1.03515625`, `RandomForestClassifier` con `n_estimators = 97` y `SVC` con `C = 0.71875`. El valor de `accuracy` obtenido es cercano al 0,82 para los datos de entrenamiento y un poco menor, pero no mucho, para los datos de test.

En los apartados 10 y 11 se examinará con detalle los resultados de dichos modelos y el por qué obtenemos dicho porcentaje de acierto.

9. Idoneidad de la función regularización

Como ya hemos comentado en la sección 7, usaremos la regularización ℓ_1 o también llamada regularización Lasso.

Como hemos comprobado, el dataset tiene una alta dimensionalidad una vez preprocesado, es decir, gran cantidad de características, y una gran cantidad de instancias, lo que puede provocar sobreajuste si el modelo es complejo. Para disminuir o regular dicha complejidad es necesario hacer uso de la regularización: un método que penaliza la complejidad del modelo, introduciendo un término en la función de coste tal que si antes de regularizar la función de coste era J, ahora resultará que es $J + \alpha C$, donde α indica cómo de importante es para nosotros que el modelo sea simple en relación a cómo de importante es su rendimiento y C es la medida de complejidad del modelo, y la escogeremos posteriormente mediante validación cruzada ya que será un hiperparámetro del modelo. Esto provoca que el modelo sea más simple y pueda generalizar mejor [3].

Como medida de complejidad del modelo, las más importantes son:

- Regularización Lasso (ℓ_1): Es la media del valor absoluto de los coeficientes

del modelo, es decir:

$$\frac{1}{N}\sum_{j=1}^{N}|\omega_j|$$

y será efectiva cuando algunas características sean irrelevantes. Al usar la regularización ℓ_1 favorecemos que algunos de los coeficientes acaben valiendo 0, es decir nos puede ayudar a hacer la selección de características. Lasso funciona mejor cuando los atributos no están muy correlados entre ellos.

- Regularización Ridge (ℓ_2): Es la media del cuadrado de los coeficientes del modelo, es decir:

$$\frac{1}{2N}\sum_{j=1}^{N}\omega_j^2$$

y será efectiva cuando algunas característica estén correladas entre ellas. Al usar la regularización ℓ_2 los coeficientes acaban siendo más pequeños lo que minimiza el efecto de la correlación entre los atributos de entrada y hace que el modelo generalice mejor. Ridge funciona mejor cuando la mayoría de los atributos son relevantes.

De esta forma y situándonos en nuestro problema, intuitivamente no todos los atributos son relevantes ya que seguramente dará igual si el cliente es `Hombre` o `Mujer` o si algún cliente ha pagado mas o menos en un mes pasado como indican las variables `PAY_AMTx` con `x` $\in \{1, 2, 3, 4, 5, 6\}$; además alguna de las variables creadas como resultado de la binarización 3 pueden no ser importantes para nuestra predicción. Creemos así que la regularización que mejor resultados nos puede proporcionar con respecto a nuestro problema es la regularización Lasso o ℓ_1.

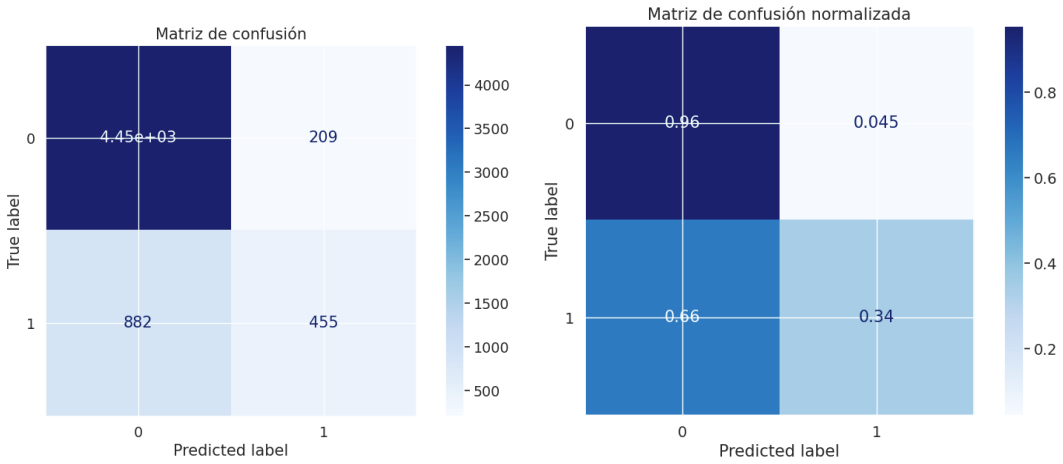
10. Valoración de los resultados

Los resultados obtenidos son, en general, notables. Comenzamos analizando los del Modelo (A). Se han obtenido los siguientes errores:

$$E_{in} = 0{,}177833333333333$$
$$E_{val} = 0{,}178770833333336$$
$$E_{test} = 0{,}181833333333333$$

Para analizar de forma correcta dichos errores, vamos a hacer uso de la matriz de confusión de nuestro clasificador. Hemos preferido incluir la matriz de confusión normalizada entre 0 y 1 para visualizar de forma correcta los valores sin tener en cuenta la escala del problema.

Como podemos comprobar por los errores obtenidos, tenemos más de un 82 % de acierto para nuestro conjunto de datos en la muestra de entrenamiento. De igual forma ocurre con la muestra reservada para el test, para la que obtenemos un 81.8 % de acierto. Esto quiere decir que nuestro clasificador no sobreajusta los datos ya que



(a) Sin normalizar (b) Normalizada

Imagen 6: Matriz de confusión para `LogisticRegression`

tanto el error de entrenamiento como el de test es bastante similar.

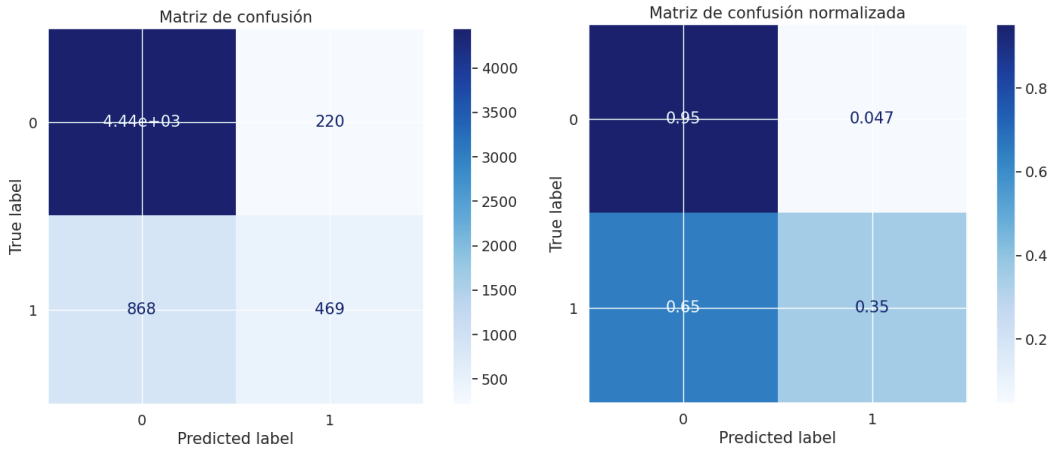
Como ya hemos comentado en la sección 1 nuestro problema consiste en decidir a qué cliente le concedemos un préstamo o no. Obviamente, preferimos que nuestro clasificador conceda solo el crédito a aquellas personas que tengan menos riesgo de que no puedan devolverlo en un futuro. En otras palabras, dentro del error, no nos importa tanto como se equivoca el clasificador sobre los clientes que no supone apenas riesgo a devolver el crédito, si no que debemos minimizar el porcentaje de clientes a los que el clasificador considera como personas sin riesgo y en realidad van a tener problemas en un futuro para devolver el crédito, ya que el banco, en este caso, perdería más dinero.

De esta forma, adentrándonos de lleno en el error de test, podemos analizar la matriz de confusión con profundidad: es cierto que nuestro clasificador es bastante restrictivo con los clientes, ya que no concede el crédito a todos los que debería, sin embargo, si realiza una correcta selección entre aquellos clientes que no deben de optar al crédito, equivocándose solo en un 4.5 % de ellos. Éste es el valor importante. Observando los errores generales obtenidos, es un número bastante notable, y como hemos comentado en la discusión anterior, esto es bastante bueno para los intereses del banco.

En el Modelo (B) los errores obtenidos son:

$$E_{in} = 0{,}17483333333333329$$
$$E_{val} = 0{,}17833333333333334$$
$$E_{test} = 0{,}18133333333333335$$

Las matrices de confusión (normalizada y sin normalizar) obtenidas son:



(a) Sin normalizar (b) Normalizada

Imagen 7: Matriz de confusión para `SVM`

De forma análoga que para el modelo anterior, el error del conjunto de test, que es el nos da información más veraz sobre lo que puede ocurrir fuera de la muestra (como error de generalización), el resultado es idéntico. Por lo que podemos decir que obtenemos unos resultados notables con dicho clasificador.

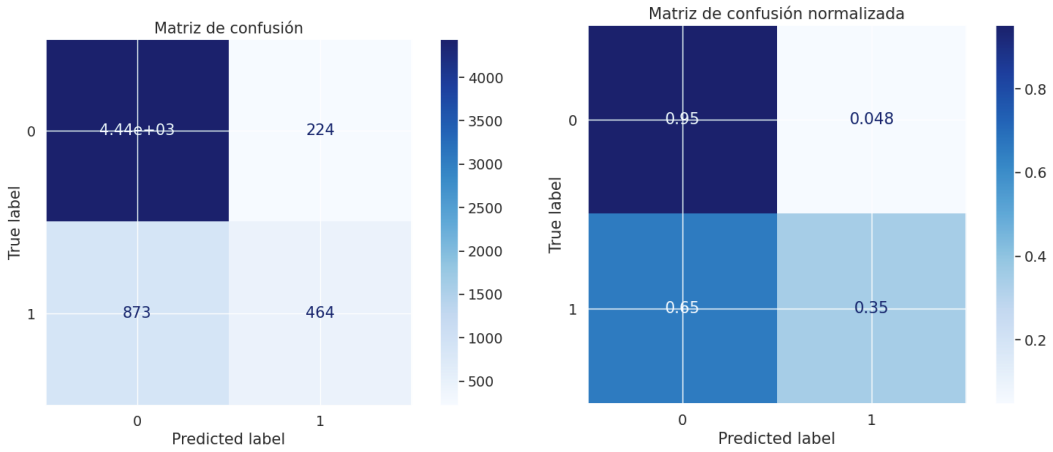
A la hora de decidir cuál de los elegir, hay una pequeña diferencia: este modelo concede el crédito a un 4.8 % de los clientes a los que no debería concedérselos, frente al 4.5 % del modelo A. Como hemos comentado antes, cuanto mayor sera dicho número, mayores probabilidades tiene el banco de perder dinero, que obviamente no nos interesa. Así, por esa mínima diferencia, consideramos que el mejor resultado de los dos lo proporciona el modelo A.

Por último en el Modelo (C) se ha obtenido:

$$E_{in} = 0{,}1302916666666667$$
$$E_{val} = 0{,}17937500000000006$$
$$E_{test} = 0{,}1828333333333333$$

Por la diferencia que hay del error E_{in} con el de E_{test} y E_{val} es muy probable que exista un poco *overfitting* al conjunto de entrenamiento. Para la obtención de este resultado los parámetros de `RandomForest` se han modificado reduciendo mucho el sobreajuste y como consecuencia se ha aumentado el error en el conjunto de entrenamiento pero se ha mejorado el de test. Esto es satisfactorio ya que queremos que se aprenda del modelo de los datos, de la información que representan y hay detrás de ellos y no de los datos de entrenamiento. Las matrices de confusión (normalizada y no normalizada) de este modelo son las siguientes:

Al igual que ocurrió en los modelos anteriores hay un porcentaje de alto de etiquetas que se predicen como `0` (no se concede el préstamo) pero cuya etiqueta real



(a) Sin normalizar (b) Normalizada

Imagen 8: Matriz de confusión para `RandomForest`

es `1` (se concede).

En vista de los errores lo primero es decir que todos ellos son muy bajos y los modelos son satisfactorios. Atendiendo a los diferentes errores calculadas claramente los mejores modelos son (A) y (B). El modelo (B) tiene el hándicap del *overfitting*. No obstante, hay que destacar la simplicidad de (A) así como su bajo tiempo de ejecución. Sin duda para esta base de datos elegiría ese modelo.

11. Justificar que se ha obtenido la mejor de las posibles soluciones con la técnica elegida y la muestra dada

Los resultados obtenidos son bastante satisfactorios, el `accuracy` es superior siempre al 80 %. El preprocesado realizado sobre la base de datos original ayuda a estos buenos resultados.

Una vez más en un problema de Aprendizaje Automático el interés no reside sólo en el modelo sino también en los datos, la calidad de ellos y en particular su manipulación. En este ejemplo hemos comprobado la necesidad del tratamiento de los mismos y la importancia de hacer esto adecuadamente y no sin motivos.

En conclusión el problema estudiado es un problema de clasificación muy interesante y de vital importancia para un banco. Cada ejemplo bien clasificado supone un gran beneficio para el banco y sin embargo un error en la clasificación va a suponer una pérdida.

Tras el estudio realizado uno de los aspectos fuertes para obtener una clasificación satisfactoria es el exhaustivo procesamiento que se ha hecho de los datos. Esto permite que diferentes modelos obtuvieran un `accuracy` muy alto. Los modelos estudiados han sido regresión logística, SVM y Random Forest. La mejor elección debido a la equivalencia de resultados es regresión logística por su simplicidad.

Referencias

[1] http://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients

[2] I-Cheng Yeh & Che-hui Lien, *The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients*. ScienceDirect, Expert Systems with Applications **36** (2009). https://doi.org/10.1016/j.eswa.2007.12.020

[3] https://iartificial.net/regularizacion-lasso-l1-ridge-l2-y-elasticnet/

[4] https://docs.microsoft.com/es-es/azure/machine-learning/algorithm-module-reference/normalize-data

[5] https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

[6] https://scikit-learn.org/stable/modules/model_evaluation.html#hinge-loss

[7] https://www.cienciadedatos.net/documentos/33_arboles_de_prediccion_bagging_random_forest_boosting

[8] https://towardsdatascience.com/optimization-loss-function-under-the-hood-part-iii-5d

[9] http://www.jmlr.org/papers/volume16/mokhtari15a/mokhtari15a.pdf

[10] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

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

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

[13] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[14] https://www.aprendemachinelearning.com/arbol-de-decision-en-python-clasificacion-y-p

[15] https://towardsdatascience.com/svm-and-kernel-svm-fed02bef1200