MODELO PREDICTIVO: CREDIT SCORE CLASSIFICATION

Trabajo de Fin de Máster

11 DE SEPTIEMBRE DE 2023

LLUIS GONZAGA FUSTER

ÍNDICE

1. Resumen 2
2. Introducción 2
3. Análisis exploratorio 3
    a. Análisis estadístico 3
    b. Análisis gráfico 5
4. Feature Engineering 8
5. Selección de variables 11
6. Evaluación de modelos 14
7. Elección del modelo 17
8. Tuneado 20
9. Conclusiones 21
10. Anexo 22

1. Resumen

En el presente trabajo se aborda la construcción de un modelo de predicción de puntuación del crédito, más conocido como 'Credit score classification'. Este proceso se desarrolla desde las etapas más básicas y primarias hasta la puesta a punto del modelo en un contexto empresarial. La variable a predecir es de tipo categórica y se llama 'credit_score'. Puede tomar tres valores diferentes, concretamente 'Good', 'Standard' y 'Poor' (refiriéndose a la calidad del cliente para recibir un préstamo), por lo tanto, se trata de una variable objetivo multivariante, lo cual aporta una serie de limitaciones a la hora de elegir distintos algoritmos de machine learning.

Por otro lado, cabe destacar que el dataset, procedente de *kaggle*, no ha resultado nada fácil de tratar, puesto que contiene valores nulos, nulos mal codificados, errores tipográficos, datos mal casteados, columnas sin poder predictivo, así como variables de tipo numérico y categórico. Por otro lado, el dataset se ha prestado a la creación de nuevas columnas a partir de las inicialmente presentes mediante procesos de 'feature engineering" que me han resultado laboriosos, pero realmente interesantes, dada la vital importancia de esta etapa del modelo para el correcto funcionamiento de los algoritmos y la obtención de resultados competitivos.

En lo referente a modelos predictivos, se ha probado con modelos de diversas características como *Logistic Regression*, *Decission Tree Classifier, Cat Boost Classifier, Random Forest Classifier, XGBoost Classifier* y *MLP classifier*, entre otros. Se ha elegido el mejor de los modelos revisando métricas como *accuracy, recall* y *f1-score* y mediante el estudio de la matriz de confusión. Por último, una vez elegido el modelo con más potencial, se ha realizado un tuneo para extraer el máximo potencial posible al algoritmo sin caer en el sobreajuste.

2. Introducción

El dataset seleccionado para el presente trabajo procede del siguiente enlace:

Credit score classification | Kaggle

Cabe destacar que se ha descargado con el máximo de columnas y filas posibles y que además, contiene dos datasets diferenciados para los datos de entrenamiento y de prueba (train and test).

La variable objetivo se llama 'Credit_Score' y define la calidad del cliente como posible beneficiario del crédito. Este modelo puede ayudar a los bancos a tomar decisiones acerca de aceptar o denegar una petición de crédito. Contiene tres posibles valores: 'Poor', 'Standard' y 'Good'.

En cuanto al resto de variables no profundizaré demasiado puesto que alargaríamos mucho esta parte del trabajo. El corrector puede indagar un poco más en el enlace o preguntarme a mí lo que considere oportuno. No obstante, si me gustaría señalar que en el dataset existen variables de tipo numéricas (int and float), de tipo object (posteriormente convertidas a 'category' por motivos de eficiencia y memoria), así como fechas y variables booleanas mal codificadas.

3. Análisis exploratorio

3.1. Análisis estadístico

Lo primero que hago es comprobar el tipo de variable y los estadísticos básicos de las variables usando dos funciones personalizadas que muestran todo lo que quiero saber de una vez (estas funciones están en el anexo, b):

```
numerical_dqr(train_data)
```

✓ 0.3s Python

	Data Type	Count	Unique Values	Missing Values	Missing %	Mean Value	Standard Deviation	Minimum Value	1st Quartile	Median	3rd Quartile	Maximum Value
Monthly_Inhand_Salary	float64	84998	13235	15002	15.0	4194.2	3183.7	303.6	1625.6	3093.7	5957.4	15204.6
Num_Bank_Accounts	int64	100000	943	0	0.0	17.1	117.4	-1.0	3.0	6.0	7.0	1798.0
Num_Credit_Card	int64	100000	1179	0	0.0	22.5	129.1	0.0	4.0	5.0	7.0	1499.0
Interest_Rate	int64	100000	1750	0	0.0	72.5	466.4	1.0	8.0	13.0	20.0	5797.0
Delay_from_due_date	int64	100000	73	0	0.0	21.1	14.9	-5.0	10.0	18.0	28.0	67.0
Num_Credit_Inquiries	float64	98035	1223	1965	2.0	27.8	193.2	0.0	3.0	6.0	9.0	2597.0
Credit_Utilization_Ratio	float64	100000	100000	0	0.0	32.3	5.1	20.0	28.1	32.3	36.5	50.0
Total_EMI_per_month	float64	100000	14950	0	0.0	1403.1	8306.0	0.0	30.3	69.2	161.2	82331.0

De este primer análisis se obtienen una serie de errores que trataré de subsanar posteriormente:

- Hay missings evidentes en dos columnas: 'Monthly_Inhand_Salary' y 'Num_Credit_inquiries'.
- Hay valores que parecen incorrectos para ciertas columnas. Por ejemplo, 'Num_Bank_Accounts' y 'Delay_from_due_data' toman los valores -1.0 y -5.0, respectivamente. Son valores incorrectos. No pueden ser negativos en ningún caso. Por tanto, además de convertirlos a números enteros en lugar de reales, asumiremos que todos los números negativos son nulos.
- En columnas como 'Num_Credit_Inquiries' hay mucha diferencia entre la mediana y el valor máximo, lo cual hace sospechar sobre la existencia de outliers o incluso una distribución de los datos poco favorable que quizás podamos solucionar pasando dicha columna a escala logarítmica o mediante algún otro tipo de transformación.
- También en la columna 'Num_Credit_Inquiries' existen valores 0, lo cual no tendría sentido. En este TFM se está asignando a cada 'solicitante de crédito' una puntuación según el riesgo de impago. Si un usuario no ha solicitado ninguno entonces no debería estar presente en el dataset.

```
categorical_dqr(train_data)
```
✓ 1.4s

	Data Type	Records	Unique Values	Missing Values	Missing %	Mode	Mode freq.	Mode %
Customer_ID	object	100000	12500	0	0.0	CUS_0x1000	8	0.0
Month	object	100000	8	0	0.0	April	12500	12.5
Name	object	90015	10139	9985	10.0	Langep	44	0.0
Age	object	100000	1788	0	0.0	38	2833	2.8
SSN	object	100000	12501	0	0.0	#F%$D@*&8	5572	5.6
Occupation	object	100000	16	0	0.0	_____	7062	7.1
Annual_Income	object	100000	18940	0	0.0	17273.83	16	0.0
Num_of_Loan	object	100000	434	0	0.0	3	14386	14.4
Type_of_Loan	object	88592	6260	11408	11.4	Not Specified	1408	1.6
Num_of_Delayed_Payment	object	92998	749	7002	7.0	19	5327	5.7
Changed_Credit_Limit	object	100000	4384	0	0.0	_	2091	2.1
Credit_Mix	object	100000	4	0	0.0	Standard	36479	36.5
Outstanding_Debt	object	100000	13178	0	0.0	1360.45	24	0.0
Credit_History_Age	object	90970	404	9030	9.0	15 Years and 11 Months	446	0.5
Payment_of_Min_Amount	object	100000	3	0	0.0	Yes	52326	52.3
Amount_invested_monthly	object	95521	91049	4479	4.5	__10000__	4305	4.5
Payment_Behaviour	object	100000	7	0	0.0	Low_spent_Small_value_payments	25513	25.5
Monthly_Balance	object	98800	98792	1200	1.2	__-333333333333333333333333__	9	0.0
Credit_Score	object	100000	3	0	0.0	Standard	53174	53.2

De este análisis se obtienen las siguientes conclusiones:

- Hay columnas categóricas que deberían ser numéricas con total seguridad: 'Annual_Income', 'Num_Banck_Accounts', 'Num_of_Loan', 'Num_of_Delayed_Payment', 'Amount_invested_monthly', 'Amount_invested_monthly', 'Monthly_Balance', 'Age', 'Delay_from_due_date', 'Num_Bank_Accounts', 'Num_Credit_Card', 'Interest_Rate', 'Num_of_Loan'. Luego casteamos.
- A parte de los nulos que tiene cada variable, se encuentran ciertos valores como '__-333333333333333333333333_' de 'Monthly_Balance' o '_______' de 'Occupation' que no tienen ningún significado, por lo tanto, son valores nulos mal tipificados.
- No hay ninguna columna que tenga un número de nulos extremadamente elevado. No borraremos ninguna por ese motivo, en prinicpio.
- Las columnas 'Credit_History_Age' y 'Payment_Behaviour' tienen un contenido con poco valor predictivo si no las transformamos en el proceso de 'Feature Engineering'. Luego veremos cómo.
- Payment_of_Min_Amount es una columna con valores 'yes', 'no' y 'NM'. Es decir, se trata de una variable booleana (1 y 0) con 'NM' como representando los valores nulos.
- En general, este dataset contiene valores valores numéricos acompañados de guiones, por ejemplo, '38321.39_'. Se asumirá que los guiones y caracteres extraños representan al número cero, puesto que están a la derecha de la coma y no aportan valor adicional al número.
- Por último, pero no menos importante, existen columnas como 'Age', que representa la edad o 'Num_of_Loan,' que representa el número de préstamos solicitados, que solo pueden tomar valores positivos. Por tanto, se convertirán en nulos todos los valores negativos. Esto ocurre en numerosas variables.

Una vez hecho el primer acercamiento a las variables de estudio y a sus estadísticos más básicos, se corrigen los errores expuestos arriba. Para ello se han usado Python y las

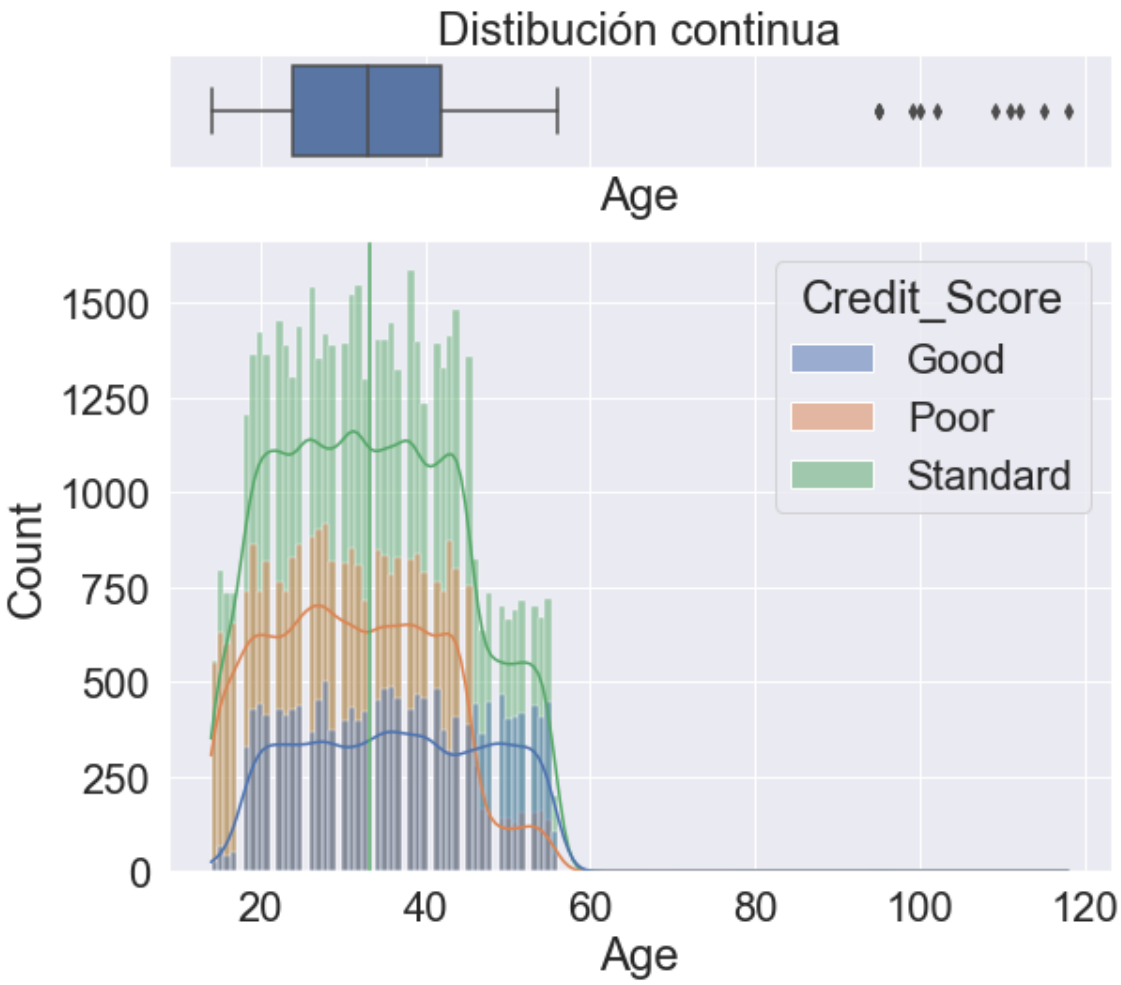
herramientas más convencionales de la librería pandas. Por no extender demasiado el texto principal, se omiten dichas líneas de código. El corrector podrá revisar el código empleado con este fin en el anexo (d).

3.2. Análisis gráfico

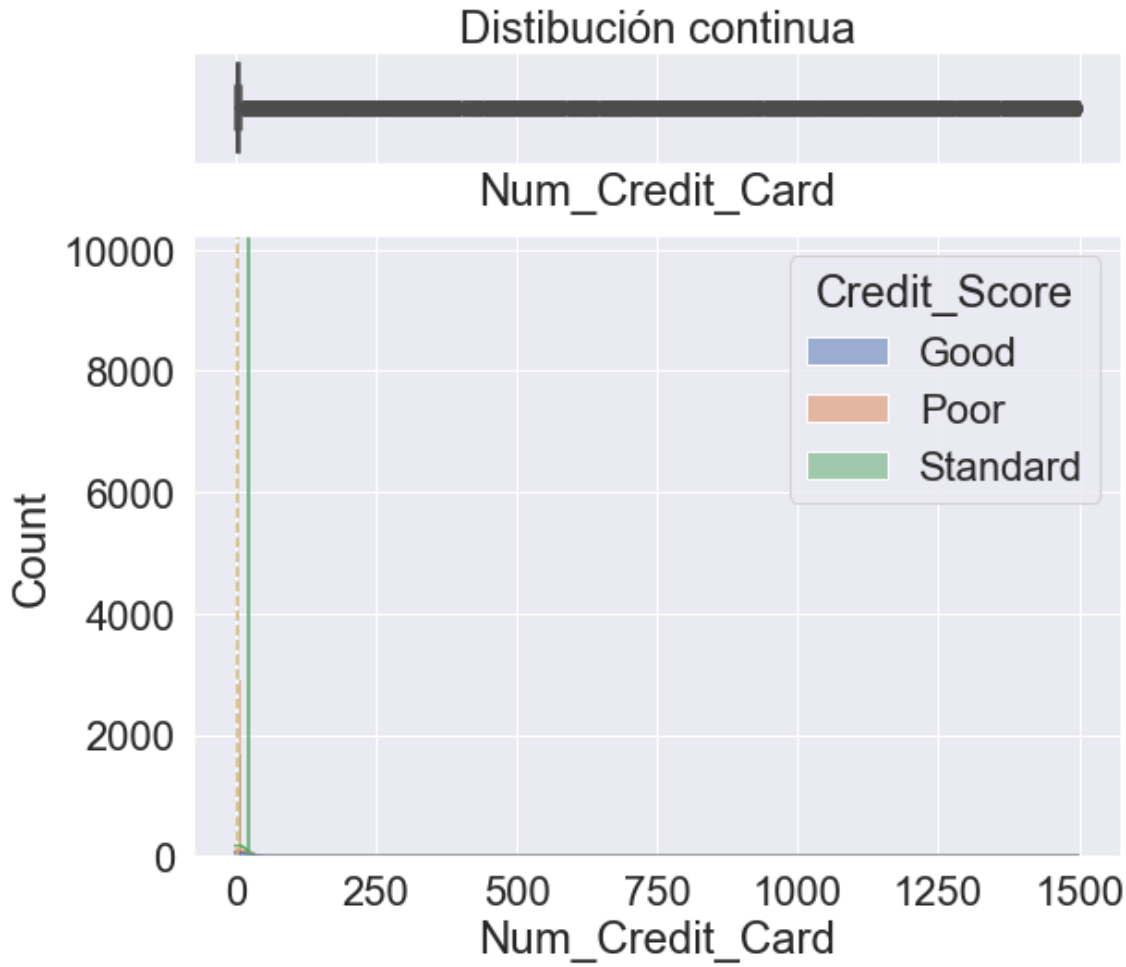
Para comprender el estado inicial de los datos es muy importante visualizar una serie de gráficos que nos muestren la distribución general de cada variable con respecto a la variable objetivo y podamos comprender si existe alguna posible relación entre las mismas.

Recordemos que en este punto del trabajo tenemos un dataset en que se han corregido valores mal escritos, se han casteado las variables al tipo correcto, se han establecido como nulos valores mal tipificados y se han borrado columnas que carecían de poder predictivo. Por tanto, nuestros datos todavía necesitan una revisión de los nulos (ya se verá el mejor método de imputación), así como un estudio de los outliers (en caso de que existan).

Para la visualización gráfica se ha usado una función propia (anexo, c) que, aplicada por columnas a nuestro dataframe, devuelve un gráfico de barras para variables de tipo categórico y un histograma para variables numéricas. Todo ello segregado por la variable objetivo. En las variables de tipo numérico se representan, además, la media y la mediana como línea verde y línea amarilla discontinua, respectivamente. Veamos algunos ejemplos relevantes:

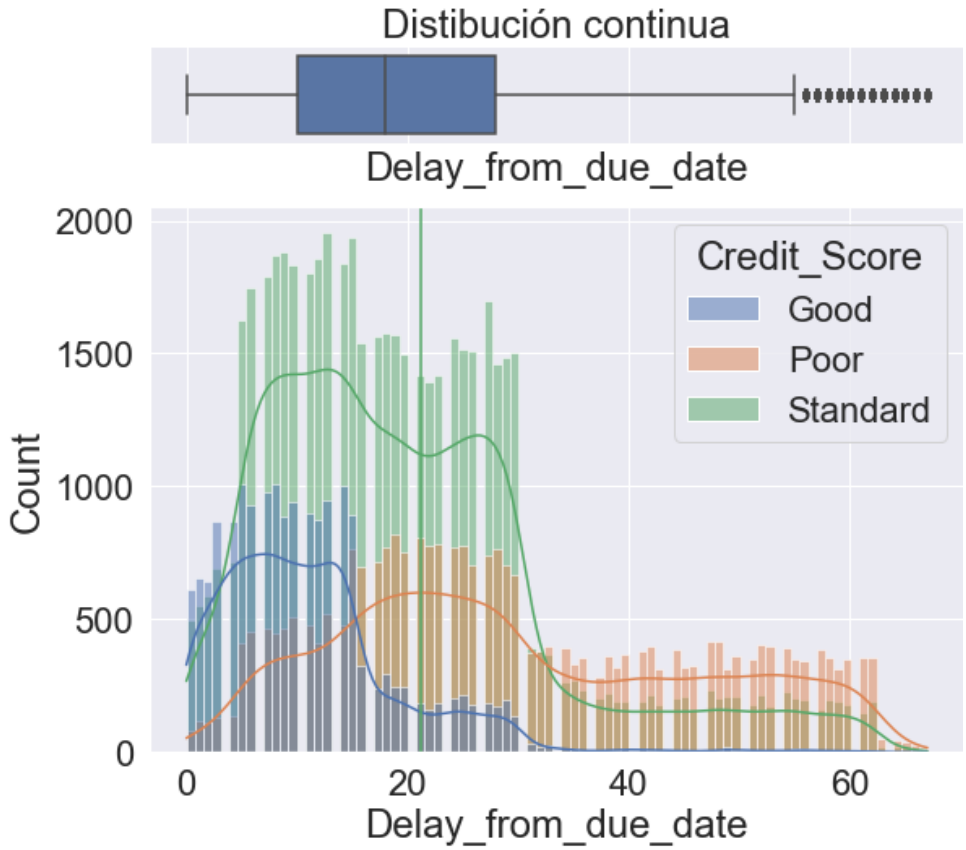


La variable 'Age', de tipo numérico está bastante bien distribuida y puede albergar algo de información, especialmente en la relación edad avanzada – 'poor'. Se observan algunos posibles outliers. Valores de 100 referidos a una edad para pedir un crédito son altamente improbables, pero no podemos eliminarlos puesto que no son 100% imposibles y de hacerlo estaríamos introduciendo un sesgo. Estudiaremos los outliers mediante otra función propia y probablemente serán eliminados o modificados.



Esta variable numérica, 'Num_Credit_Card', es realmente interesante. Se encuentra muy comprimida a la izquierda del gráfico. No podemos considerar, usando la misma lógica que en el apartado anterior, que una persona o compañía no pueda poseer 1000 o 2000 tarjetas de crédito. Por lo tanto, no podemos simplemente eliminar dichas observaciones, sino que estudiaremos los outliers y probaremos algún tipo de transformación logarítmica o el método 'power transform' de sklearn para intentar mejorar la distribución de esta variable. Este tipo de transformaciones serán aconsejables también para el resto de las variables que tengan este comportamiento. En concreto:

['Num_Credit_Card', 'Num_Bank_Accounts', 'Interest_Rate', 'Changed_Credit_Limit', 'Total_EMI_per_month', 'Num_of_Loan']



Esta variable resulta bastante interesante y podría tener cierto valor predictivo (luego lo comprobaremos), puesto que la categoría 'Poor' se encuentra claramente más concentrada en lado derecha de la gráfica que el resto.

También es muy importante observar la variable objetivo, 'Credit_Score'. En este caso, la distribución está relativamente bien compensada y equilibrada. Por lo que no parece una necesidad el uso undersampling, oversamplig o cualquier método de compensación del número de observaciones de la variable objetivo. En principio, aceptamos la distribución existente. No obstante, se probará la técnica de oversampling y se comprobará si la precisión de los modelos aumenta con ella o no.

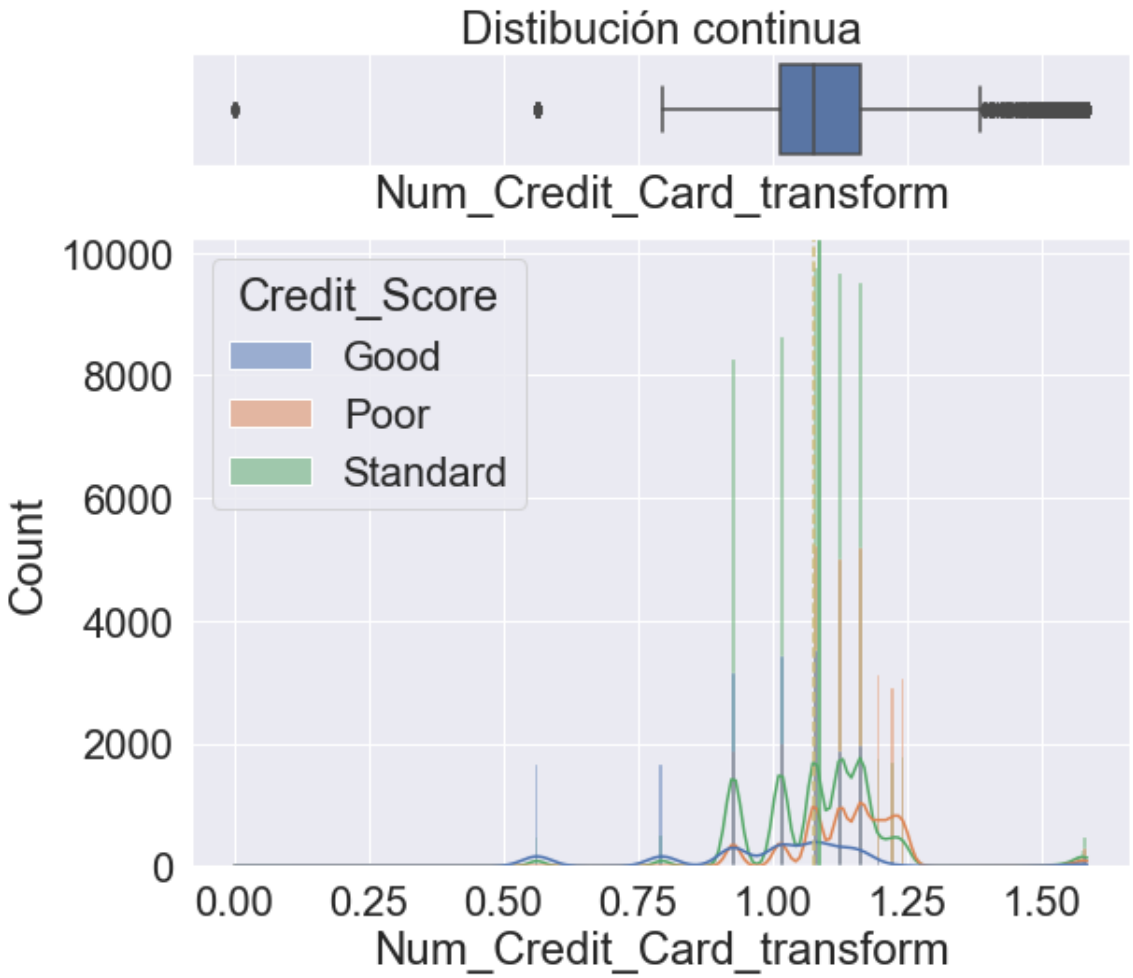
4 Feature Engineering

A lo largo de los apartados anteriores se han comentado multitud de posibles transformaciones e incluso de creación de nuevas columnas que vamos a construir y cuyo comportamiento vamos a comprobar en los distintos modelos predictivos.

A continuación, seguiremos el paso a paso realizado:

- Columna 'Credit_history_Age': Contiene datos con esta estructura 22 *Years and 1 Months.* Se han construido tres columnas a partir de esta. Una columna con los años, una con los meses y otra con el total de tiempo en días.
- Columna 'Payment_Behaviour': Contienen datos con esta estructura *High_spent_Small_value_payments.* Se han aislado las palabras que ocupan la posición de 'High' y de 'Small' porque son las únicas que podrían contener información relevante. Se han guardado esas dos palabras en dos columnas diferentes.

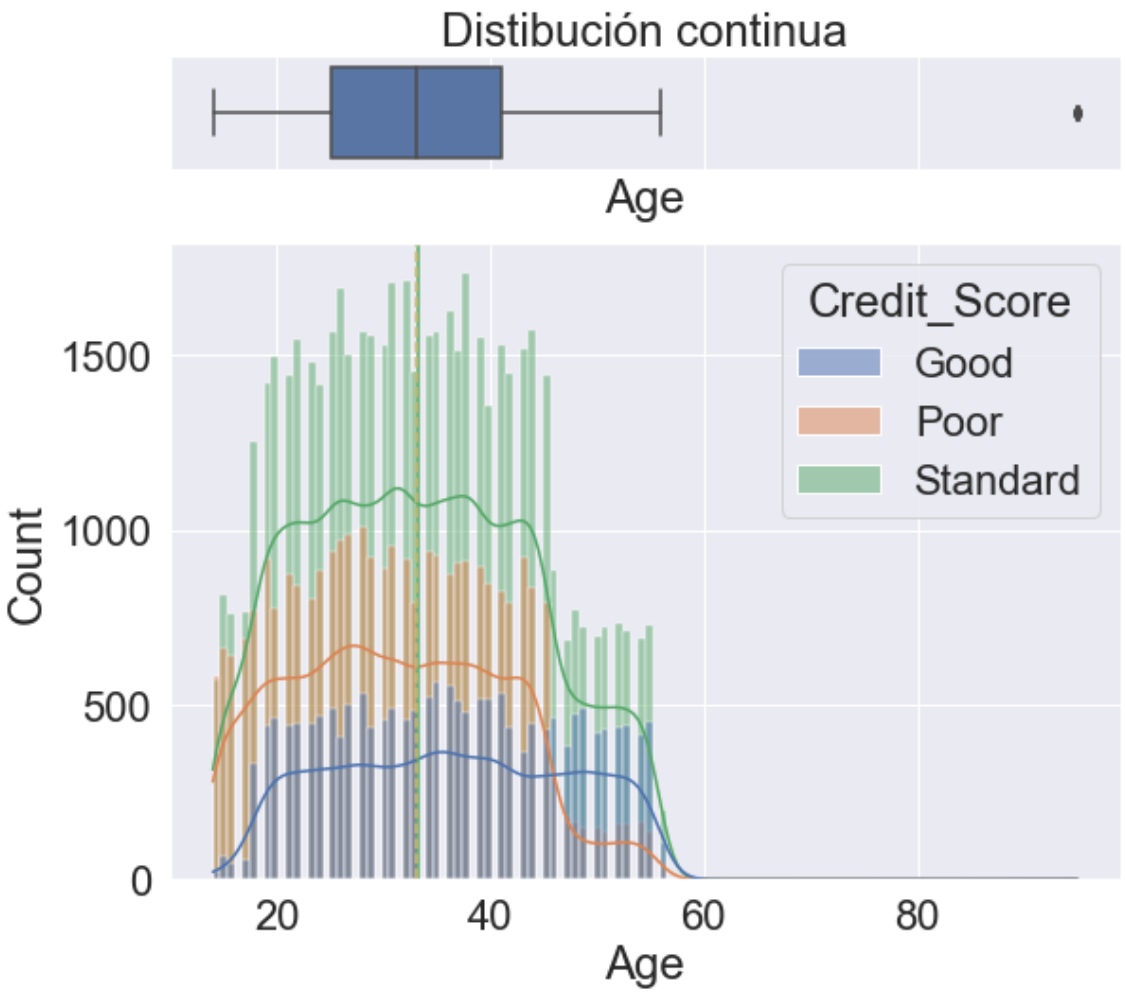
Se aplica el objeto de scikit-learn 'PowerTransformer' con el método de 'yeo-johnson'. Recordemos que este método aplica una serie de transformaciones a las variables numéricas con el fin de normalizarlas y hacerlas más simétricas. Estas características resultan de vital importancia puesto que normalmente las variables bien normalizadas y simétricas son mejor procesadas por el modelo y devuelven mejores resultados. Remarquemos que esto es sólo válido para variables de tipo numérico. En el presente trabajo se han añadido las columnas modificadas al dataframe principal y se comprueba mediante gráficos si ha mejorado el 'aspecto' de las variables modificadas respecto de las originales. En caso afirmativo, borraremos las antiguas.



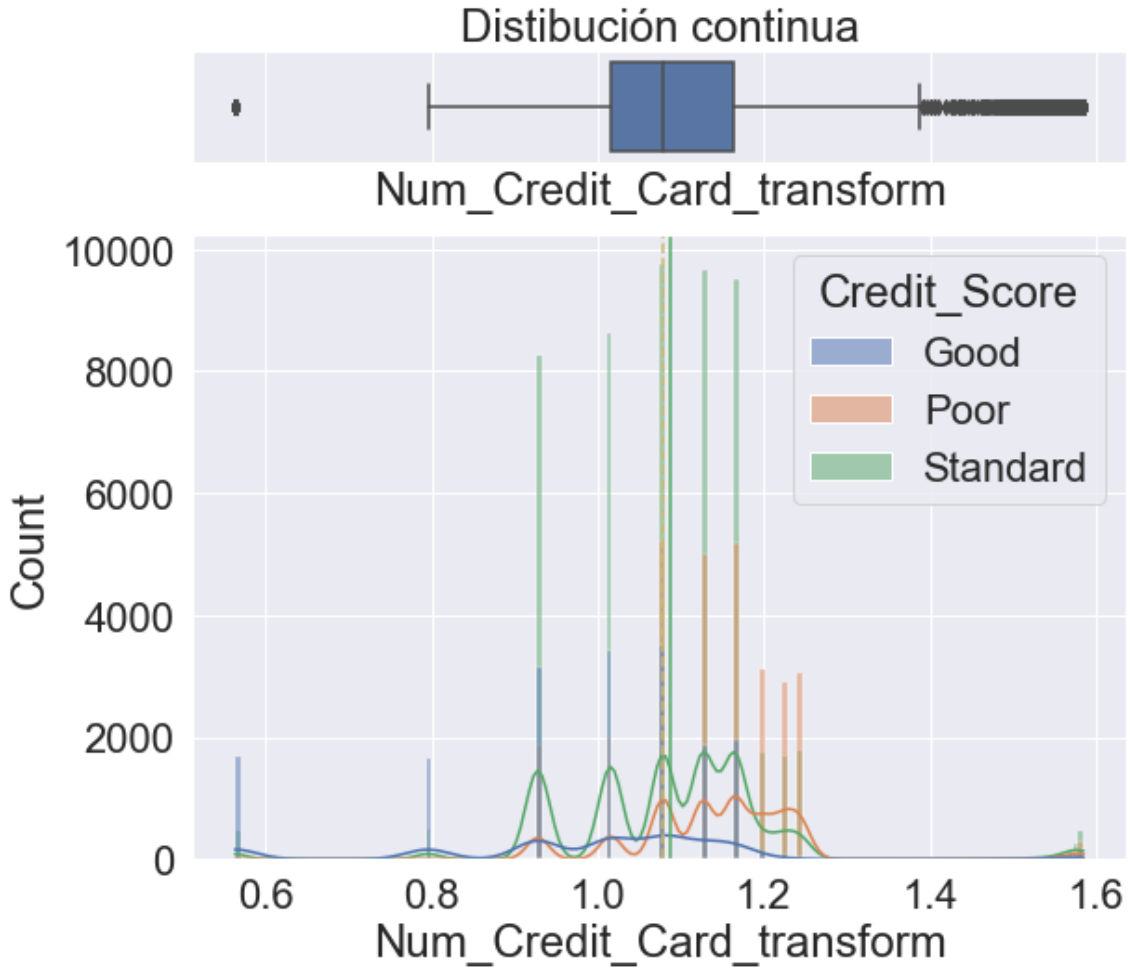
En este caso concreto, tras las transformaciones aplicadas a la columna 'Num_Bank_Accounts' se ha obtenido una columna más centrada y uniformemente distribuida. No obstante, sigue estando bastante dispersa. Vamos a estudiar el comportamiento de esta nueva columna con los distintos modelos y comprobaremos cuál de las dos funciona mejor.

- Gestión de Outliers: Se ha utilizado una función propia que usa la técnica de winsorización ajustando los valores atípicos para que estén dentro de los límites definidos por los cuantiles calculados. Esto ayuda a reducir la influencia de los valores atípicos en el análisis de datos y a obtener resultados más robustos. En cualquier caso, la función usada para manipular outliers se adjunta en el anexo (b).

Fijémonos en el cambio que ha habido en las gráficas tras modificar los outliers. A simple vista se observa que algunos valores que se ubicaban en los extremos han sido 'centrados'. Todo ha sido realizado de una manera muy sensible y superficial puesto que no soy partidario de gestionar los outliers con demasiada intensidad puesto que estaríamos cambiando datos de entrada que existen en nuestro problema. Sin embargo, creo que este punto es correcto.



Realicemos un rápido repaso también a la columna 'Num_Credit_Card_transform' y cómo se ha estrechado un poco el gráfico. Se han 'redistribuido' algunos valores extremos que estaban 'muy a la izquierda' del primer cuartil.

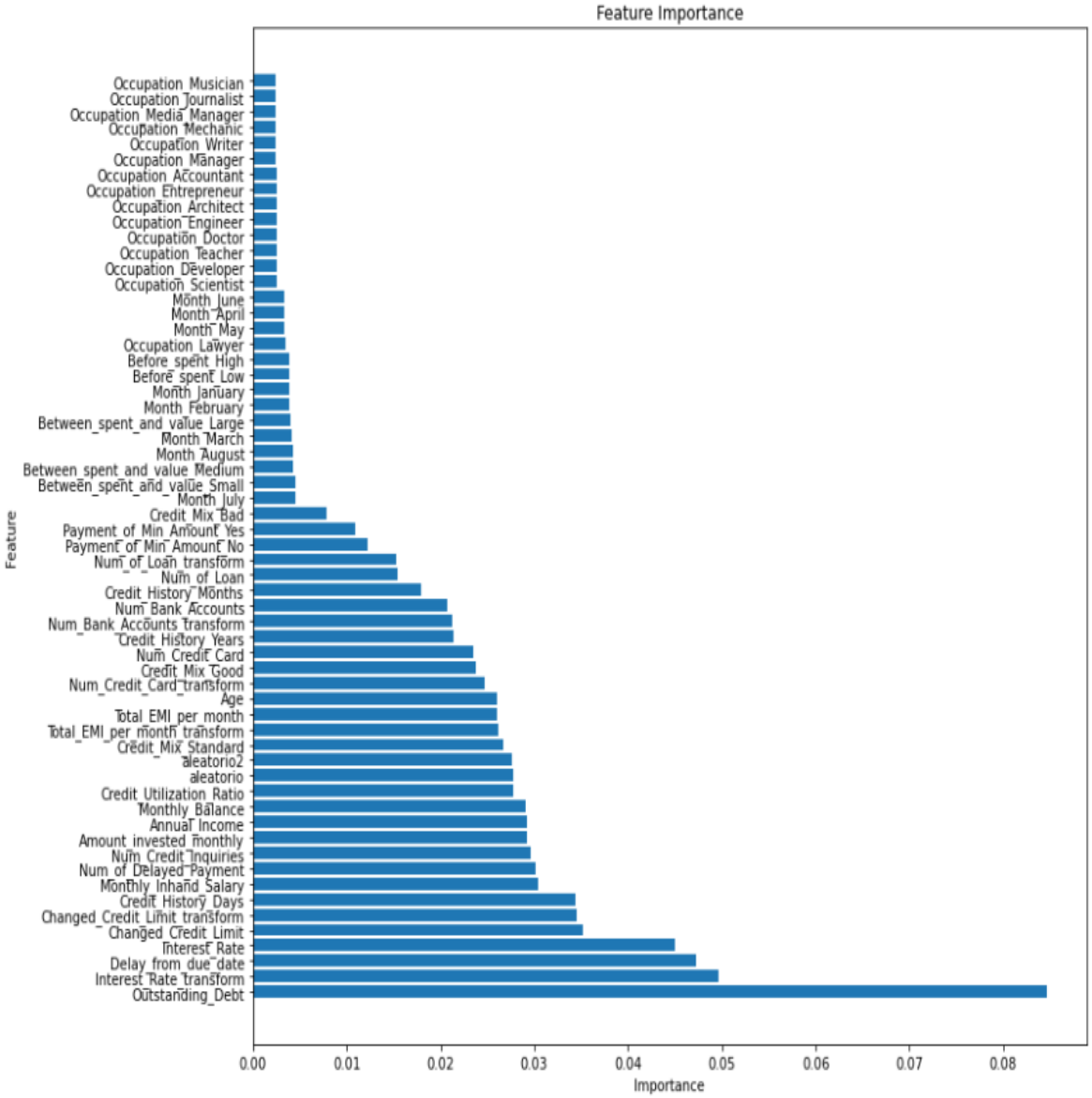


a) Respecto al tratamiento de variables, sólo quedaría usar un *One Hot Encoder* (o similar) para convertir las columnas categóricas en numéricas.
b) En este caso he aplicado también un *StandardScaler*.

5 Selección de variables

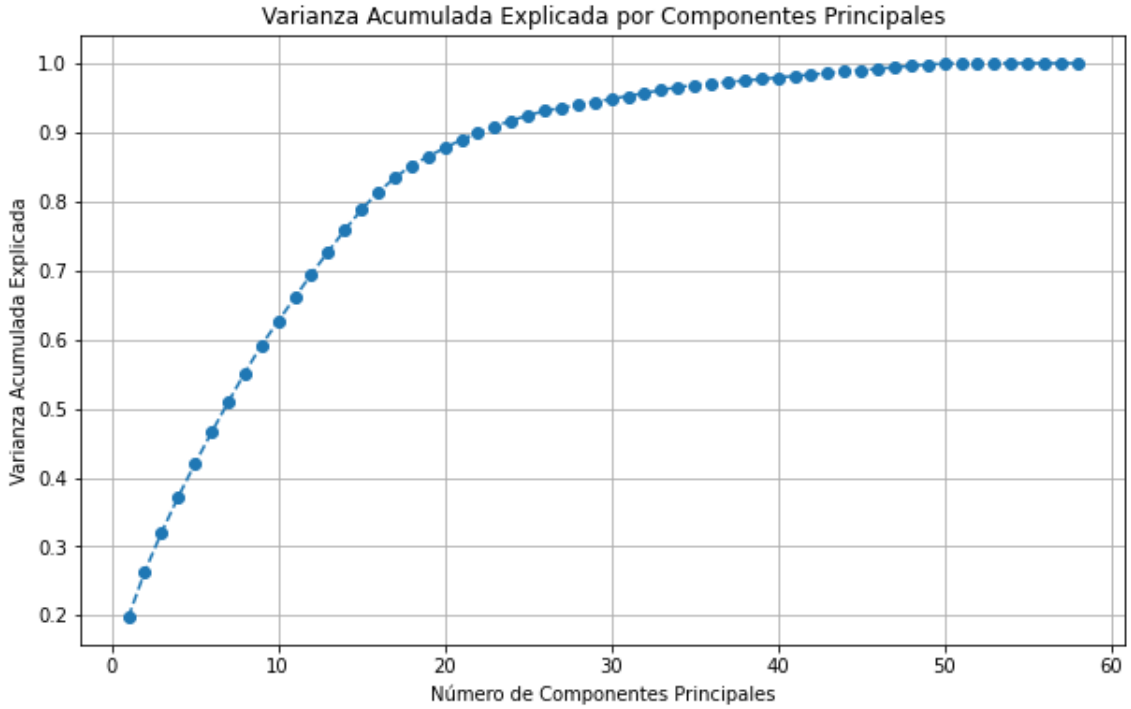
En primer lugar, utilizamos un *Random Forest* con el atributo *feature_importances* (ver código en anexo, e) para visualizar las variables que tienen más importancia en la predicción en comparación con una variable aleatoria que se ha creado como referencia. Toda variable que tenga menos importancia que una aleatoria podrá ser eliminada sin problemas.

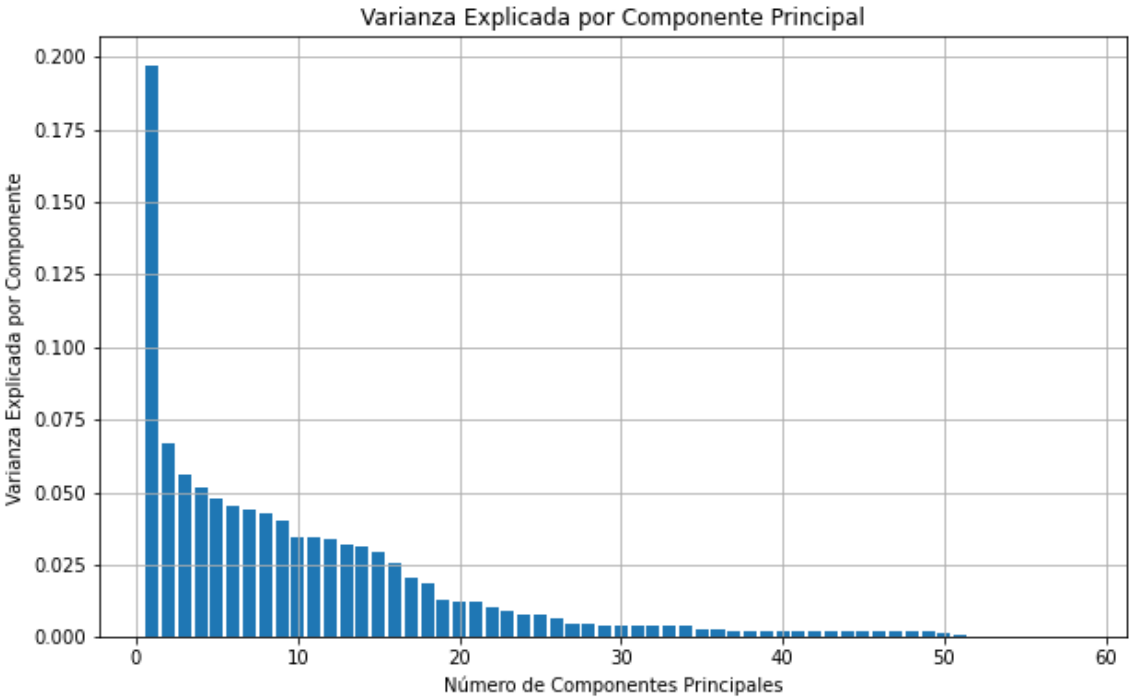
Veamos:



Resulta bastante revelador que la columna 'Outstanding_Debt' es la variable más importante para el *RF*. Por otro lado, hay muchas que están por encima de 'aleatorio' y 'aleatorio2', por lo que podrán ser eliminadas sin problemas.

Por otro lado, se ha probado un análisis *PCA* para distinto número de componentes. De esta manera se pretende obtener el número exacto en que la precisión es buena sin complicar demasiado el modelo.





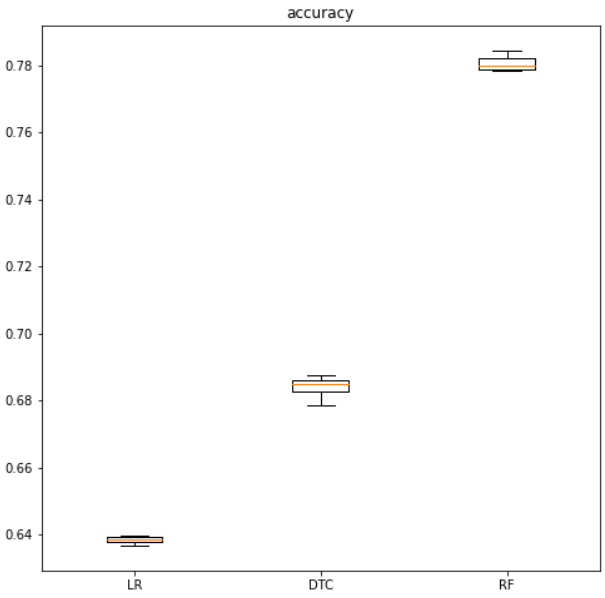
En este caso, considero el 20 como un número de componentes adecuado para este problema. Son “pocos” datos, por lo que no ralentizaría demasiado la ejecución de los códigos.

6 Evaluación de modelos

Los apartados anteriores han sido verdaderamente necesarios para todo lo que viene ahora. En este punto del informe, tenemos un dataframe libre de valores incorrectos, sin nulos y sin outliers. En realidad, ahora el informe diverge en varios caminos. Vamos a evaluar una selección de modelos en varios dataframes distintos. Me explico, vamos a aplicar los modelos típicos y más sencillos al dataframe con todas las variables, también al dataframe con el subcojunto de columnas seleccionadas previamente, también al dataframe tras usar la técnica de oversampling t, por último, también al dataframe obtenido mediante PCA.

Utilizaremos esta técnica sólo con unos pocos modelos básicos y rápidos de ejecutar (LogisticRegression, DecisionTreeClassifier y RandomForestClassifier), para elegir el mejor dataframe posible de las cuatro opciones expuestas y, una vez seleccionada la mejor opción, aplicaremos mayor variedad de modelos y tunearemos el más competitivo.

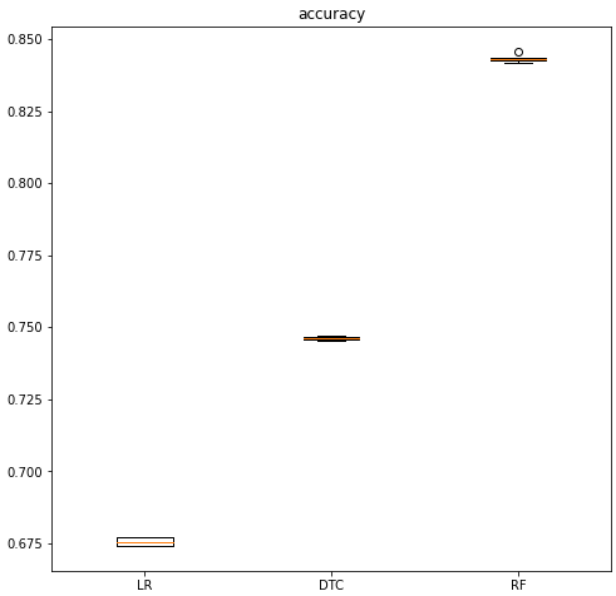
a) Dataframe completo, KNN imputer, sin oversampling




```
LR: 0.6384
[[ 9035  1670  5708]
 [  515 14697  7358]
 [ 8278 12631 40108]]
 ***
DTC: 0.684
[[10627  1054  6190]
 [ 1098 19312  8412]
 [ 6103  8632 38572]]
 ***
RF: 0.7807
[[12499   745  4211]
 [  112 22834  6216]
 [ 5217  5419 42747]]
 ***
```

Dejando a un lado que RF parece el modelo más competitivo en términos de accuracy, la matriz de confusión también afirma lo mismo. La mayoría de valores etán bien predichos.

b) Dataframe completo, KNN imputer, con oversampling:

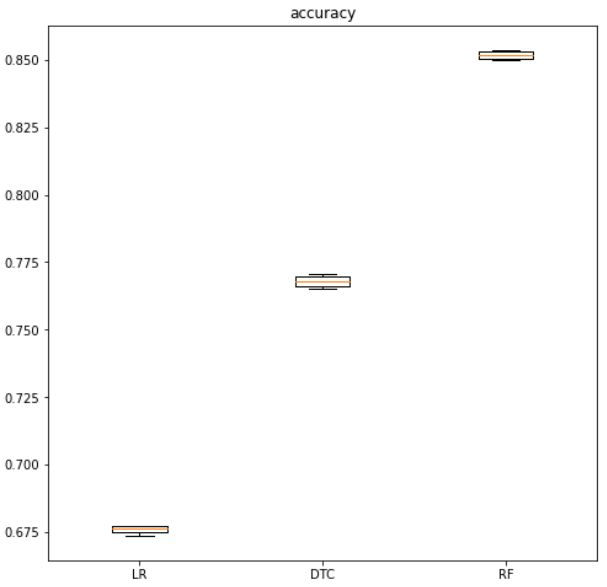



```
LR: 0.6755557041224726
[[40584  6980 10138]
 [ 2091 35717 11571]
 [10499 10477 31465]]
 ***
DTC: 0.7462105479385425
[[42441  3917  7314]
 [ 3889 39839  9203]
 [ 6844  9418 36657]]
 ***
RF: 0.8433068668594486
[[49115  3806  6295]
 [  592 45164  6729]
 [ 3467  4204 40150]]
 ***
```

El resultado de este test me ha resultado realmente sorprendente. El simple hecho de aplicar la técnica de oversampling aumenta en bastante puntos la accuracy de todos los modelos. Así mismo aumenta la predicción de las variables que antes eran minoritarias (fila 1 y fila 2). Este hecho vuelve a reitar la grandísima importancia de tener unos datos de calidad y bien trabajados. Esto es mucho más importante que el tuneado fino del modelo.

En definitiva, de ahora en adelante trabajaremos con este dataframe por el buen resultado que obtenemos del oversampling.

c) Dataframe con columnas seleccionadas, KNN imputer, con oversampling:



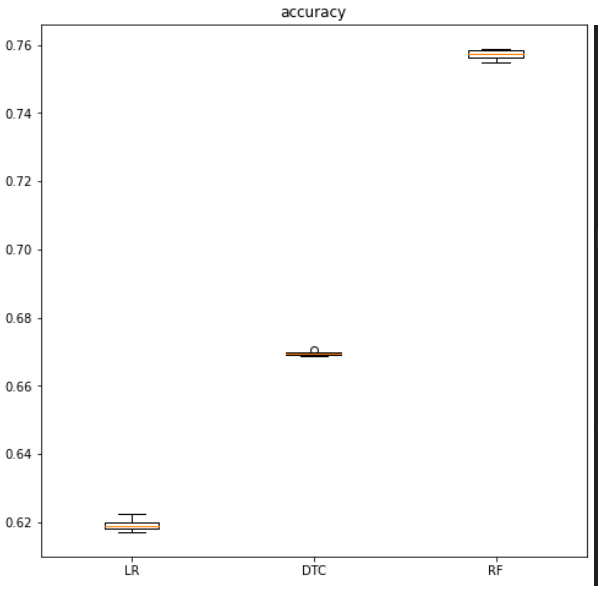

```
LR: 0.6758754110492254
[[40366  7131  9824]
 [ 1774 35470 11369]
 [11034 10573 31981]]
 ***
DTC: 0.7679253984905495
[[43285  3295  6791]
 [ 3305 41348  8449]
 [ 6584  8531 37934]]
 ***
RF: 0.8516944190310205
[[49035  2971  5461]
 [  599 46059  6861]
 [ 3540  4144 40852]]
 ***
```

Reduciendo el número de columnas hemos mejorado la accuracy y hemos mejorado el número de aciertos en la matriz de confusión en cada una de las tres categorías. Por lo tanto, a

partir de ahora sólo trabajaremos con este subconjunto del dataframe formado por las siguientes columnas:

['Outstanding_Debt', 'Delay_from_due_date', 'Interest_Rate_transform', 'Interest_Rate', 'Credit_Mix_Good', 'Num_Credit_Inquiries', 'Num_of_Delayed_Payment', 'Credit_History_Days', 'Num_Bank_Accounts_transform', 'Changed_Credit_Limit_transform', 'Credit_Mix_Standard', 'Payment_of_Min_Amount_No', 'Changed_Credit_Limit', 'Num_Credit_Card_transform', 'Num_Bank_Accounts', 'Num_Credit_Card', 'Monthly_Inhand_Salary', 'Credit_History_Years', 'Annual_Income', 'Monthly_Balance', 'Age', 'Amount_invested_monthly', 'Total_EMI_per_month', 'Payment_of_Min_Amount_Yes', 'Credit_Utilization_Ratio']

d) Dataframe con 20 componentes de *PCA*:



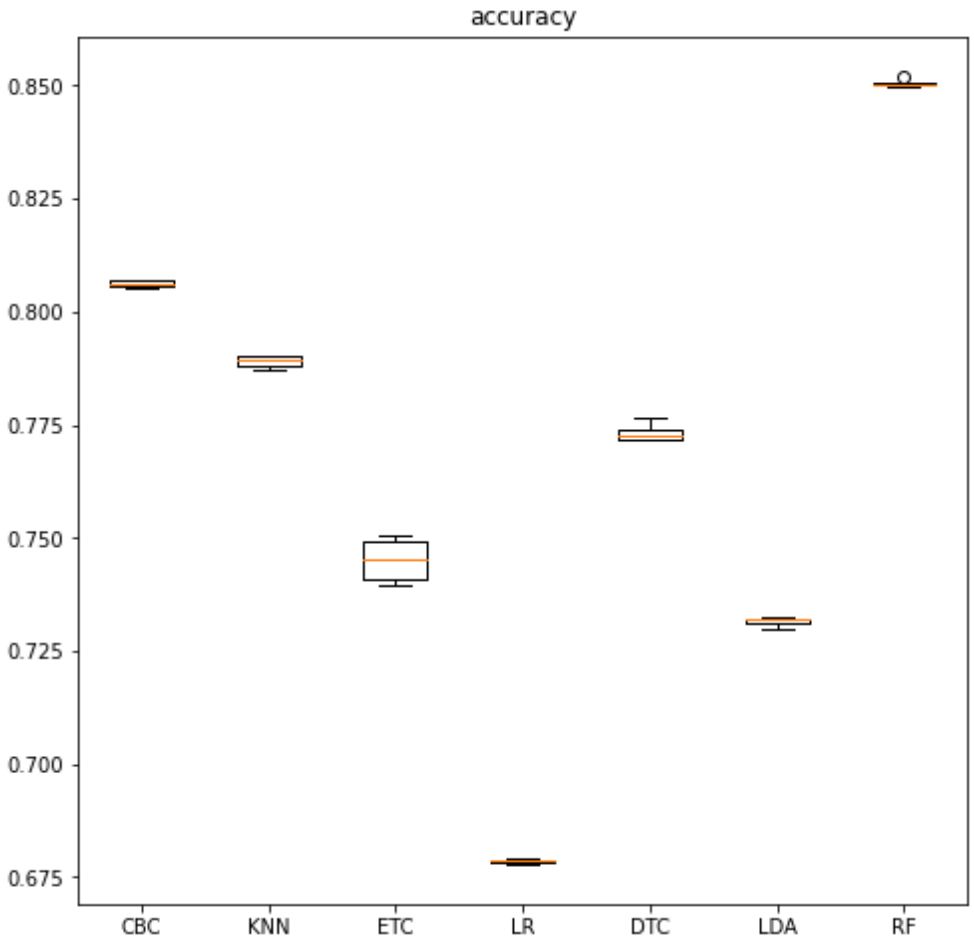

```
LR: 0.61928
[[ 7852  1636  4749]
 [  485 12412  6761]
 [ 9491 14950 41664]]
 ***
DTC: 0.66957
[[10307  1225  6250]
 [ 1231 18699  8894]
 [ 6290  9074 38030]]
 ***
RF: 0.75718
[[11967   823  4302]
 [  113 21021  6136]
 [ 5748  7154 42736]]
 ***
```

Se observa que los resultados son bastante similares a los de la opción a. En ese caso descartamos el uso de *PCA* puesto que no mejoramos la *accuracy* ni mejoramos la matriz de confusión, sin embargo, estamos añadiendo complejidad y falta de interpretabilidad al modelo.

Conclusión: a lo largo de estas líneas hemos expuesto ideas y extraído conclusiones de las pruebas realizadas y queda bastante claro que la opción c es la mejor. El dataframe con el que vamos a seguir el presente informe es el de la opción c: un dataframe al que, además de las transformaciones básicas expuestas en los apartados 2 y 3, hemos seleccionado un subconjunto de todas las columnas, se ha aplicado la técnica de *KNN* para la imputación de los missings categóricos, el método *OneHotEncoder* para convertir las variables categóricas en numéricas y, por último, la técnica de *oversampling*, la cual ha impulsado notablemente la calidad del modelo.

7) Elección del modelo

Una vez justificada la elección del dataframe con el que vamos a continuar el informe, se evalúan y comparan una lista más amplia de modelos, que incluyen *Extra Tree Classifier*, *XGB Classifier* y *Cat Boost Classifier*, entre otros. Este último no es un modelo tan popular, pero lo he probado en otros proyectos y suele devolver buenos resultados. Veamos a continuación una comparativa gráfica de la *accuracy* así como de la matriz de confusión de cada uno, con el fin de observar cuál es la categoría que mejor predice cada modelo y tomar decisiones en función de la categoría que más nos importe predecir correctamente y la que más nos interese evitar el error:



Nota: se ha añadido aparte el algoritmo *XGBClassifier*. El motivo es que necesita que la variable objetivo esté codificada con números, en lugar de con los caracteres 'Good', 'Standard' y 'Poor'. Por ese motivo se ha aplicado un *label encoder* a la variable objetivo y se ha probado después que los demás.

```
CBC: 0.8087392829329521
[[46569  4982  6205]
 [ 1146 42372  7312]
 [ 5459  5820 39657]]
 ***
KNN: 0.7890134227876076
[[48862  3282 11138]
 [  894 45484 10517]
 [ 3418  4408 31519]]
 ***
ETC: 0.7451260639192974
[[42083  3299  7671]
 [ 3445 40552  9274]
 [ 7646  9323 36229]]
 ***
LR: 0.6785082911971541
[[40480  7061  9740]
 [ 1760 35701 11378]
 [10934 10412 32056]]
 ***
DTC: 0.7732475671982222
[[43393  3308  6517]
 [ 3402 41440  8188]
 [ 6379  8426 38469]]
 ***
LDA: 0.7314602468484666
[[42539  5032  9068]
 [ 2086 40466 10563]
 [ 8549  7676 33543]]
 ***
RF: 0.850622480169628
[[48857  3006  5349]
 [  628 45894  6824]
 [ 3689  4274 41001]]
 ***
xgb: 0.8022843245453117
[[46363  5103  6161]
 [ 1247 41707  7101]
 [ 5564  6364 39912]]
 ***
```

¡La gráfica de arriba y las matrices de confusión de la izquierda son realmente interesantes! Los modelos basados en árboles resultan interesantes (hablando de precisión), destacando por encima de todos el *Random Forest* y el *XGB Classifier*. Profundicemos un poco en su funcionamiento y su lógica:

El *Random Forest* se basa en la idea de construir un conjunto de árboles de decisión, donde cada árbol se entrena de manera independiente en una submuestra aleatoria de los datos y con un conjunto aleatorio de características. Luego, las predicciones de cada árbol se combinan para tomar una decisión final. Este enfoque reduce el sobreajuste y aumenta la robustez del modelo, lo que lo hace eficaz y fácil de usar. Por otro lado, el *XGBoostClassifier* utiliza un enfoque de *Gradient Boosting* para crear un conjunto de árboles de decisión secuenciales, donde cada nuevo árbol se enfoca en corregir los errores del modelo anterior. Esto permite que el modelo se ajuste de manera adaptativa a los datos, mejorando continuamente su rendimiento. Además, *XGBoost* incorpora técnicas de regularización y muestreo aleatorio para prevenir el sobreajuste y mejorar la generalización.

En definitiva, estos dos algoritmos suelen ser efectivos y tienen herramientas para prevenir el sobreajuste.

Tras una primera revisión, el *Random Forest* parece el más competitivo por su alta precisión. No obstante, me gustaría mostrar otras métricas muy útiles que también nos pueden ayudar a elegir mejor el modelo más allá de la precisión, como *recall* y *f1-score*. Para comprender las decisiones que vamos a tomar, definamos *recall* como un parámetro que indica la proporción de casos positivos reales que el modelo logró identificar correctamente y f1-score como la media armónica de ambas métricas y proporciona una medida equilibrada del rendimiento del modelo en la clasificación de instancias positivas. Es una métrica que combina precisión y *recall* en un solo valor.

Fijándonos en nuestro problema en particular, no hay una explicación clara y concisa de la utilidad real del dataframe, por tanto, no podemos perseguir un objetivo con total seguridad. Sin embargo, creo interesante minimizar el número de clientes calificados como malos que en realidad son buenos o estándar, así como el número de clientes malos mal calificados. Por lo tanto, queremos un *recall* alto para la categoría 'Poor'.

```
KNN: 0.7899725548853878
              precision    recall  f1-score   support

        Good       0.77      0.92      0.84     53174
        Poor       0.80      0.86      0.83     53174
    Standard       0.80      0.59      0.68     53174

    accuracy                           0.79    159522
   macro avg       0.79      0.79      0.78    159522
weighted avg       0.79      0.79      0.78    159522

 ***
LR: 0.6791351872434837
              precision    recall  f1-score   support

        Good       0.71      0.76      0.73     53174
        Poor       0.73      0.67      0.70     53174
    Standard       0.60      0.60      0.60     53174

    accuracy                           0.68    159522
   macro avg       0.68      0.68      0.68    159522
weighted avg       0.68      0.68      0.68    159522

 ***
RF: 0.851067564953832
              precision    recall  f1-score   support

        Good       0.85      0.92      0.88     53174
        Poor       0.86      0.86      0.86     53174
    Standard       0.84      0.77      0.80     53174

    accuracy                           0.85    159522
   macro avg       0.85      0.85      0.85    159522
weighted avg       0.85      0.85      0.85    159522

 ***
```

Estas métricas que no observamos ala izquierda albergan mucha información relevante. Empecemos hablando del modelo KNN. Pese a tener una *accuracy* mejorable predice muy bien (alto *recall*) las categorías 'Good' y 'Poor'. Esta última es claramente la más importante de predecir para el banco que busque mejorar la calidad de los clientes a los que presta su dinero. Para nuestro propósito, podríamos elegir indistintamente el *KNN* y el *RF*, puesto que predicen igual de bien (mismo *recall*) las categorías que más no interesa predecir. Entonces, seleccionamos finalmente el Random Forest porque además de tener alto *recall* para las categorías 'Poor' y 'Good' tiene un valor alto de *accuracy*: 0,8506.

8) Tuneado

El tuneo de un algoritmo es importante para la generación de un modelo predictivo competitivo y de calidad. Realmente, lo interpreto como la punta del iceberg de un largo proceso. Es una etapa en la que el modelo puede ganar algunas décimas de precisión y mejorar, por tanto, la predicción. No obstante, si ajustamos con demasiada profundidad los hiperparámetros corremos el riesgo de sobreajuste y perder, en esencia, capacidad predictiva para datasets diferentes al de entrenamiento. Por ello, se ha realizado un tuneo de hiperparámetros mediante el uso del *RandomizedSearchCV*, más rápido que el *GridSearchCV* que he usado en otras ocasiones. Si el corrector desea ver el código usado para el tuneo lo puede visualizar en el anexo, h.

Los mejores hiperparámetros tras el tuneo del modelo son los siguientes:

RandomForestClassifier(n_estimators=200, max_depth=None, min_samples_split=3, min_samples_leaf=5, max_features='auto', 'bootstrap=True, random_state=42)

Se obtienen las siguientes métricas:

RF: 0.8581241297245911

	precision	recall	f1-score	support
Good	0.85	0.92	0.88	53174
Poor	0.86	0.85	0.85	53174
Standard	0.85	0.81	0.81	53174
accuracy			0.85	159522
macro avg	0.85	0.85	0.85	159522
weighted avg	0.85	0.85	0.85	159522

***

El modelo ha cambiado un poco respecto al modelo por defecto de *Random Forest*. No hemos mejorado el *recall* de *Poor* y *Good* y el tiempo de cómputo ha sido bastante elevado. En mi opinión, la relación *esfuerzo / recompensa* no merece la pena. En cualquier caso, podríamos tomar cualquiera de los dos modelos y ambos serían competentes, de calidad y robustos.

Antes de dar por finalizado este apartado, me gustaría destacar que se ha prestado especial atención al uso correcto de métodos correctos de segmentación de los datos, 'datos de entrenamiento' y 'datos test'. Para todas las pruebas de modelos realizadas en el presente informe se ha utilizado el algoritmo de validación cruzada repetida, el cual considero bastante equilibrado en cuanto a seguridad y rapidez en comparación a otros métodos.

9) Conclusiones

A lo largo del informe se han tomado multitud de decisiones de forma razonada, lógica y argumentada, acerca de valores nulos, mal codificados o erróneos, columnas con distribuciones extrañas alejadas de la distribución normal, *outliers* y descompensación en la variable objetivo, entre otras circunstancias. Se han soslayado dificultades usando técnicas avanzadas de Machine Learning como *oversampling*, validación cruzada repetida o PCA y se han analizado estadísticos importantes y métricas como la precisión (*accuracy*), *recall* y *f1-score* útiles para la elección correcta del modelo según las necesidades del problema.

Tras todo este proceso, se ha elegido el algoritmo *Random Forest* con los hiperparámetros mostrados en el apartado anterior y extraídos gracias a un tuneo exhaustivo a partir del método *RandomizedSearchCV* de la librería *scikit learn* (se ha usado esta función porque exige menos poder de cómputo que el *GridSearchCV* y este código ha sido ejecutado en mi ordenador personal).

Considero que he sabido dar respuesta al problema principal de este informe. Se ha creado un modelo predictivo capaz de predecir de manera competitiva la 'calidad' de un cliente como receptor de un crédito bancario con una precisión de más del 85%. Sin embargo, tengo claras algunas mejoras que podría haber implemento en el informe si hubiera podido dedicarle más tiempo y que seguro desarrollaré en el futuro, por ejemplo, testear distintos tipos de ensamblado, investigar por *GitHub* y *Stackoverflow* algún algoritmo distinto, probar *undersampling* y, especialmente, crear una red neuronal y probarla en el dataset. Por último, también podríamos haber enfocado el modelo a la productivización del mismo. Es decir, aplicar ciertos métodos bajo una infraestructura de *pipelines* que pudiera recibir nuevos datos y devolviera las predicciones. Desde luego, será un reto enfrentarse a ello en el futuro próximo.

10) Anexo

En el presente anexo se adjuntan algunos pedazos de código y gráficas a los que se hace referencia a lo largo del informe.

a) Función que calcula y muestra los estadísticos básicos para variables numéricas y categóricas:

```
def numerical_dqr(df):

    # Select numerical columns
    numerical = df.select_dtypes(include = ['int', 'Int64',
'float']).columns.tolist()

    # Data type
    data_types = pd.DataFrame(df[numerical].dtypes, columns = ['Data
Type'])

    # Missing data
    missing_data = pd.DataFrame(df[numerical].isnull().sum(), columns =
['Missing Values'])

    # Unique values
    unique_values = pd.DataFrame(columns = ['Unique Values'])
    for row in list(df[numerical].columns.values):
        unique_values.loc[row] = [df[numerical][row].nunique()]

    # Number of records
    count_values = pd.DataFrame(columns = ['Count'])
    for row in list(df[numerical].columns.values):
        count_values.loc[row] = [df[numerical][row].count()]

    # Maximum value
    maximum_values = pd.DataFrame(columns = ['Maximum Value'])
    for row in list(df[numerical].columns.values):
        maximum_values.loc[row] = [df[numerical][row].max()]

    # Minimum value
    minimum_values = pd.DataFrame(columns = ['Minimum Value'])
    for row in list(df[numerical].columns.values):
        minimum_values.loc[row] = [df[numerical][row].min()]

    # Mean value
    mean_values = pd.DataFrame(columns = ['Mean Value'])
    for row in list(df[numerical].columns.values):
        mean_values.loc[row] = [df[numerical][row].mean()]
```

```
    # Standard Deviation
    std = pd.DataFrame(columns=['Standard Deviation'])
    for row in list(df[numerical].columns.values):
        std.loc[row] = [np.std(df[numerical][row])]

    # First quartile
    quartile_1 = pd.DataFrame(columns = ['1st Quartile'])
    for row in list(df[numerical].columns.values):
        quartile_1.loc[row] = [df[numerical][row].quantile(0.25)]

    # Median
    median = pd.DataFrame(columns = ['Median'])
    for row in list(df[numerical].columns.values):
        median.loc[row] = [df[numerical][row].quantile(0.5)]

    # Third quartile
    quartile_3 = pd.DataFrame(columns = ['3rd Quartile'])
    for row in list(df[numerical].columns.values):
        quartile_3.loc[row] = [df[numerical][row].quantile(0.75)]

    # Join columns
    dq_report_num =
data_types.join(count_values).join(missing_data).join(unique_values).join
(minimum_values)\
        .join(maximum_values).join(mean_values).join(std).join(quartile_1
).join(median).join(quartile_3)

    # Percentage missing
    dq_report_num['Missing %'] = (dq_report_num['Missing Values'] /
len(df[numerical]) * 100)

    # Change order of columns
    dq_report_num = dq_report_num[['Data Type', 'Count', 'Unique Values',
'Missing Values', 'Missing %', 'Mean Value',
                                   'Standard Deviation', 'Minimum Value',
'1st Quartile', 'Median', '3rd Quartile',
                                   'Maximum Value']]

    # Round
    dq_report_num[['Missing %', 'Mean Value', 'Standard Deviation',
'Minimum Value', '1st Quartile', 'Median', '3rd Quartile', 'Maximum
Value']] =  dq_report_num[['Missing %', 'Mean Value', 'Standard
Deviation', 'Minimum Value', '1st Quartile', 'Median', '3rd Quartile',
'Maximum Value']].round(1)

    # Return report
    return(dq_report_num)

def categorical_dqr(df):
```

```
  #select categorical columns
  categorical = df.select_dtypes(include = ['object',
'category']).columns.tolist()

  #datatype
  data_types = pd.DataFrame(
    df[categorical].dtypes,
    columns=['Data Type'])

  #count
  count_values = pd.DataFrame(
      columns=['Records'])
  for row in list(df[categorical].columns.values):
    count_values.loc[row] = [df[categorical][row].count()]

   #missing data
  missing_data = pd.DataFrame(
    df[categorical].isnull().sum(),
    columns=['Missing Values'])

  #unique values
  unique_values = pd.DataFrame(
      columns=['Unique Values'])
  for row in list(df[categorical].columns.values):
    unique_values.loc[row] = [df[categorical][row].nunique()]

  #mode
  mode_values = pd.DataFrame(
      columns=['Mode'])
  for row in list(df[categorical].columns.values):
    mode_values.loc[row] = [df[categorical][row].mode()[0]]
    mode = mode_values.loc[row]

  #mode frequency
  listModeFreq = []
  for row in categorical:
    mode = df[row].mode().iat[0]
    ModeFreq = df[row].value_counts()[mode]
    #print(x, mode, df[x].value_counts()[mode])
    listModeFreq .append(ModeFreq )
  listModeFreq  = np.array(listModeFreq )


   #create data quality report
  dq_report_cat =
data_types.join(missing_data).join(count_values).join(unique_values).join
(mode_values)
  dq_report_cat['Mode freq.'] = listModeFreq
```

```
  dq_report_cat['Mode %'] = (dq_report_cat['Mode freq.'] /
dq_report_cat['Records']*100) .astype('float')
  dq_report_cat['Missing %'] = (dq_report_cat['Missing Values'] /
len(df[categorical]) *100)

  #change order of columns
  dq_report_cat = dq_report_cat[['Data Type', 'Records', 'Unique Values',
'Missing Values', 'Missing %', 'Mode', 'Mode freq.', 'Mode %']]
  dq_report_cat[['Missing %', 'Mode %']] =   dq_report_cat[['Missing %' ,
'Mode %']].round(1  )

  #return report
  return(dq_report_cat)
```

b) Función que gestiona los outliers de nuestro dataframe:

```
## Función manual de winsor con clip+quantile
def winsorize_with_pandas(s, limits):
    """
    s : pd.Series
        Series to winsorize
    limits : tuple of float
        Tuple of the percentages to cut on each side of the array,
        with respect to the number of unmasked data, as floats between 0.
and 1
    """
    return s.clip(lower=s.quantile(limits[0], interpolation='lower'),
                  upper=s.quantile(1-limits[1], interpolation='higher'))

## Función para gestionar outliers
def gestiona_outliers(col, clas = 'check'):

     print(col.name)
     # Condición de asimetría y aplicación de criterio 1 según el caso
     if abs(col.skew()) < 1:
        criterio1 = abs((col-col.mean())/col.std())>3
     else:
        criterio1 = abs((col-col.median())/col.mad())>8

     # Calcular primer cuartil
     q1 = col.quantile(0.25)
     # Calcular tercer cuartil
     q3 = col.quantile(0.75)
     # Calculo de IQR
     IQR=q3-q1
     # Calcular criterio 2 (general para cualquier asimetría)
     criterio2 = (col<(q1 - 3*IQR))|(col>(q3 + 3*IQR))
```

```
    lower =
col[criterio1&criterio2&(col<q1)].count()/col.dropna().count()
    upper =
col[criterio1&criterio2&(col>q3)].count()/col.dropna().count()
    # Salida según el tipo deseado
    if clas == 'check':
           return(lower*100,upper*100,(lower+upper)*100)
    elif clas == 'winsor':
           return(winsorize_with_pandas(col,(lower,upper)))
    elif clas == 'miss':
           print('\n MissingAntes: ' + str(col.isna().sum()))
           col.loc[criterio1&criterio2] = np.nan
           print('MissingDespues: ' + str(col.isna().sum()) +'\n')
           return(col)
```

c) Función que realiza el plot de una manera muy cómoda, con distintas características en función de si se trata de una variable categórica o numérica y que se aplica por columnas mediante una función *apply:*

```
def histogram_boxplot(data, col, hue = None, xlabel = None, title = None,
font_scale=2, figsize=(9, 8), bins = None):
    """ Boxplot and histogram combined
    data: 1-d data array
    xlabel: xlabel
    title: title
    font_scale: the scale of the font (default 2)
    figsize: size of fig (default (9,8))
    bins: number of bins (default None / auto)

    example use: histogram_boxplot(np.random.rand(100), bins = 20,
title="Fancy plot")
    """
    # Definir tamaño letra
    sns.set(font_scale = font_scale)
    # Crear ventana para los subgráficos
    f2, (ax_box2, ax_hist2) = plt.subplots(2, sharex = True, gridspec_kw
= {"height_ratios": (.15, .85)}, figsize = figsize)
    # Crear boxplot
    sns.boxplot(data = data, x = col, hue = hue, ax = ax_box2)
    # Crear histograma
    sns.histplot(data = data, x = col, ax = ax_hist2, bins = bins, kde =
True, hue = hue) if bins else sns.histplot(data = data, x = col, ax =
ax_hist2, kde = True, hue = hue)
    # Pintar una línea con la media
    ax_hist2.axvline(np.mean(data[col]),color='g',linestyle='-')
    # Pintar una línea con la mediana
    ax_hist2.axvline(np.median(data[col]),color='y',linestyle='--')
```

```
    # Asignar título y nombre de eje si tal
    if xlabel: ax_hist2.set(xlabel = xlabel)
    if title: ax_box2.set(title = title, xlabel = xlabel)
    #if title: ax_box2.set(title=title.encode('utf-8').decode('latin1'),
xlabel="")
    # Mostrar gráfico
    plt.show()

## Función para gráfico de barras de variables categóricas (básica)
def cat_plot(col):
     if col.dtypes == 'category':
        #fig = px.bar(col.value_counts())
        #pio.show(fig)
        fig = sns.countplot(x = col)
        plt.show()

## Función para gráfico de barras de variables categóricas. Muestra
porcentajes en las etiquetas
def box_porcentajes (col):
    if col.dtypes == 'category':

        total = len(col)
        porcentajes = col.value_counts(normalize=True) * 100

        # Crear el gráfico de barras
        fig, ax = plt.subplots(figsize=(8, 6))  # Crear figura y ejes
        sns.countplot(x=col, ax=ax)  # Gráfico de barras

        # Agregar etiquetas de texto con los porcentajes
        for p in ax.patches:
            height = p.get_height()
            ax.annotate(f"{height/total*100:.1f}%", (p.get_x() +
p.get_width() / 2, height), ha='center', va='baseline')

        plt.show()

## Función general plot para aplicar al archivo por columnas
def plot(df, hue = None):
    """ Ejemplo de uso:
    plot(data, hue = 'target')
    data: contiene el dataframe completo, incluída la variable objetivo o
target
    hue: Si quieremos segregar el boxplot en funcion de la variable
objetivo, hay que introducir el nombre de la columna de la variable
objetivo, en este caso, target.
    """
    tipos_prohibidos = ['category', 'object', 'datetime64[ns]']
    for col in df.columns:
        if df[col].dtype not in tipos_prohibidos:
```

```
            print('Cont: {}'.format(df[col].name))
            histogram_boxplot(df, col = col, hue = hue, xlabel =
df[col].name, title = 'Distibución continua')
        elif df[col].dtype == 'category':
            print('Cat: {}'.format(df[col].name))
            #cat_plot(col)
            box_porcentajes(df[col])
        elif df[col].dtype == 'datetime64[ns]':
            print('Fecha')
```

d) Código usado para corregir columnas con errores tipográficos:

```
train_data = train_data.drop_duplicates()
train_data['Monthly_Balance'] =
train_data['Monthly_Balance'].replace('__-333333333333333333333333333__',
None)
train_data['Occupation'] = train_data['Occupation'].replace('_______',
None)
train_data['Amount_invested_monthly'] =
train_data['Amount_invested_monthly'].replace('__10000__', None)
train_data['Changed_Credit_Limit'] =
train_data['Changed_Credit_Limit'].replace('_', None)
train_data.loc[train_data['Num_Bank_Accounts'] < 0, 'Num_Bank_Accounts']
= None
train_data.loc[train_data['Delay_from_due_date'] < 0,
'Delay_from_due_date'] = None
train_data['Payment_Behaviour'] =
train_data['Payment_Behaviour'].replace('!@9#%8', None)
train_data['Payment_of_Min_Amount'] =
train_data['Payment_of_Min_Amount'].replace('NM', None)
train_data['Credit_Mix'] = train_data['Credit_Mix'].replace('_', None)
train_data['Occupation'] = train_data['Occupation'].replace({np.nan:
None})
train_data['Payment_of_Min_Amount'] =
train_data['Payment_of_Min_Amount'].replace({np.nan: None})
valores_no_numericos =
train_data['Num_of_Delayed_Payment'].str.contains(r'[^0-9.-]',
regex=True)

# Los _ están al final del número, a la derecha de la coma. Por tanto, lo
convertiré en cero
train_data['Annual_Income'] =
train_data['Annual_Income'].str.replace(r'[^0-9.-]', '0')
train_data['Num_of_Loan'] = train_data['Num_of_Loan'].str.replace(r'[^0-
9.-]', '0')
train_data['Age'] = train_data['Age'].str.replace(r'[^0-9.-]', '0')
```

```
train_data['Num_of_Delayed_Payment'] =
train_data['Num_of_Delayed_Payment'].str.replace(r'[^0-9.-]', '0')
train_data['Outstanding_Debt'] =
train_data['Outstanding_Debt'].str.replace(r'[^0-9.-]', '0')
train_data['Changed_Credit_Limit'] =
train_data['Changed_Credit_Limit'].str.replace(r'[^0-9.-]', '0')

# Amount_invested_monthly and Monthly_Balance should be numerical columns
train_data[['Annual_Income', 'Monthly_Balance',
'Amount_invested_monthly', 'Outstanding_Debt']] =
train_data[['Annual_Income', 'Monthly_Balance',
'Amount_invested_monthly', 'Outstanding_Debt']].astype(float)

# Num_of_Loan columns has only integer values but is of object type. We
will cast to integer
train_data[['Num_of_Loan', 'Age']] = train_data[['Num_of_Loan',
'Age']].astype(int)
train_data['Num_of_Delayed_Payment'] =
pd.to_numeric(train_data['Num_of_Delayed_Payment'], errors='coerce')
train_data['Changed_Credit_Limit'] =
pd.to_numeric(train_data['Changed_Credit_Limit'], errors='coerce')

#'Changed_Credit_Limit'
train_data['Age'] = train_data['Age'].apply(lambda x: None if (x < 0 or x
> 120) else x)
train_data['Num_of_Loan'] = train_data['Num_of_Loan'].apply(lambda x:
None if (x < 0) else x)
train_data['Num_of_Delayed_Payment'] =
train_data['Num_of_Delayed_Payment'].apply(lambda x: None if (x < 0) else
x)
train_data['Changed_Credit_Limit'] =
train_data['Changed_Credit_Limit'].apply(lambda x: None if (x < 0) else
x)

# Encontramos columnas numéricas con menos de 10 valores
columnas_object = train_data.select_dtypes(include =
'object').columns.tolist()
def convert_to_category(df, cols):
    """
    Convierte las columnas especificadas en la lista 'cols' a tipo
'category' en el DataFrame 'df'.
    """
    for col in cols:
        df[col] = df[col].astype('category')
    return df

# Llamar a la función para convertir las columnas en 'train_data' a tipo
'category'
train_data = convert_to_category(train_data, columnas_object)
```

```
train_data = train_data.drop(['Customer_ID', 'Name', 'SSN'], axis=1)
```

e) Código usado para obtener de forma ordenada un gráfico con la importancia de cada variable en el modelo predictivo.

```
rf = RandomForestClassifier(n_estimators=100, random_state=42)
X['aleatorio'] = np.random.uniform(0, 1, len(X))
X['aleatorio2'] = np.random.uniform(0, 1, len(X))

# Entrenar el modelo
rf.fit(X, y)

# Obtener la importancia de las características del modelo Random Forest
feature_importances = rf.feature_importances_

# Crear un DataFrame que incluya el nombre de la característica y su
importancia
feature_importance_df = pd.DataFrame({'Feature': X.columns, 'Importance':
feature_importances})

# Ordenar el DataFrame por importancia en orden descendente
feature_importance_df =
feature_importance_df.sort_values(by='Importance', ascending=False)

# Crear un gráfico de barras para visualizar la importancia de
características
plt.figure(figsize=(12, 12))
plt.barh(feature_importance_df['Feature'],
feature_importance_df['Importance'])
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importance')
plt.show()

X.drop(['aleatorio', 'aleatorio2'], axis = 1, inplace = True)
```

f) Código para estudiar el comportamiento del PCA y el número de variables óptimo en caso de que sea necesario:

```
# Ajustar el modelo PCA
pca = PCA()
pca.fit(X)

# Obtener la varianza explicada por cada componente
varianza_explicada = pca.explained_variance_ratio_
```

```
# Calcular la varianza acumulada explicada
varianza_acumulada = np.cumsum(varianza_explicada)

# Crear un gráfico de la varianza explicada
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(varianza_explicada) + 1), varianza_acumulada,
marker='o', linestyle='--')
plt.xlabel('Número de Componentes Principales')
plt.ylabel('Varianza Acumulada Explicada')
plt.title('Varianza Acumulada Explicada por Componentes Principales')
plt.grid()
plt.show()

pca = PCA()
pca.fit(X)

# Obtener la varianza explicada por cada componente
varianza_explicada = pca.explained_variance_ratio_

# Crear un gráfico de la varianza explicada por cada componente
plt.figure(figsize=(10, 6))
plt.bar(range(1, len(varianza_explicada) + 1), varianza_explicada)
plt.xlabel('Número de Componentes Principales')
plt.ylabel('Varianza Explicada por Componente')
plt.title('Varianza Explicada por Componente Principal')
plt.grid()
plt.show()
```

g) Función que evalúa los algorítmicos de una lista y devuelve la precisión, matriz de confusión y un gráfico comparativo entre todos los modelos:

```
def model_evaluation(models, X = X, y = y, scoring = 'accuracy',
confussion = False):

    names = []
    cv_results = []

    for name, model in models:
        kfold = KFold(n_splits = 4, shuffle = True, random_state = 42)
        results = cross_val_score(model, X, y, cv = kfold, scoring =
scoring)

        cv_results.append(results)
        names.append(name)
        print(name + ': ' + str(results.mean()))
```

```
        if confussion == True:
            y_pred = cross_val_predict(model, X, y, cv = kfold)
            print(confusion_matrix(y_pred, y))
            print(' *** ')

    plt.figure(figsize=(8,8))
    plt.title(scoring)
    plt.boxplot(cv_results)
    plt.xticks(range(1, len(names) + 1), names)
    plt.show()
```

h) Código utilizado para el tuneo:

```
# Elegimos este modelo
# Tuneamos el Random Forest

from sklearn.model_selection import RandomizedSearchCV

# Definir los hiperparámetros que deseas ajustar
param_grid = {
    'n_estimators': [10, 50, 100, 200, 300, 1000, 5000, 10000],
    'max_depth': [None, 10, 20, 30, 40, 50, 100],
    'min_samples_split': [2, 5, 10, 30],
    'min_samples_leaf': [1, 2, 4, 10],
    'max_features': ['auto', 'sqrt', 'log2'],
    'bootstrap': [True, False]
}

# Crear un RandomForestClassifier
rf = RandomForestClassifier(random_state=42)

# Crear un objeto RandomizedSearchCV
random_search = RandomizedSearchCV(estimator=rf,
param_distributions=param_grid, n_iter=100,
                                   scoring='accuracy', cv=4, verbose=2,
random_state=42, n_jobs=-1)

# Realizar la búsqueda de hiperparámetros en tus datos
random_search.fit(X, y)

# Mostrar los mejores hiperparámetros encontrados
print("Mejores hiperparámetros:")
print(random_search.best_params_)

# Mostrar la mejor puntuación
print("Mejor puntuación:", random_search.best_score_)
```