E.T.S. de Ingeniería Industrial, Informática
y de Telecomunicación

IMPLEMENTACIÓN Y ESTUDIO DEL RENDIMIENTO DEL CLASIFICADOR DERS-BOOST PARA PROBLEMAS CON CONJUNTOS DE DATOS NO BALANCEADOS

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autora: Cristina Ojer Baztán

Director: Josean Sanz Delgado

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

RESUMEN

El objetivo de este trabajo es implementar, estudiar y comparar frente a otros métodos de aprendizaje automático ya conocidos, el rendimiento de DERS-Boost (Differential Evolutionary Resampling Boosting). DERS-Boost es una nueva propuesta frente a problemas de clasificación con conjuntos de datos no balanceados, que integra técnicas de re-muestreo guiadas por algoritmos de computación evolutiva diferencial. Esta memoria describe las bases teóricas y el funcionamiento del nuevo algoritmo de aprendizaje, y se exponen los experimentos llevados a cabo, teniendo en cuenta una distinta variedad de datasets, y comparando los resultados del rendimiento en cuanto a precisión y tiempo de ejecución entre DERS-Boost y una selección de métodos ya conocidos para afrontar los problemas de clasificación no balanceados.

Palabras clave: clasificación, problemas no balanceados, re-muestreo, ensemble learning, boosting, evolución diferencial, clustering.

Índice

1. INTRODUCCIÓN

En el marco del aprendizaje automático o *Machine Learning*, la construcción de modelos predictivos de clasificación es una herramienta muy útil y recurrente para infinidad de escenarios de aplicación, en los que se desea predecir ciertos comportamientos o clasificar una serie de entradas compuestas de distintas variables.

Uno de los procedimientos para dotar de conocimiento a un sistema es a través de algoritmos de aprendizaje supervisado, en donde se lleva a cabo la explotación de registros de datos ya conocidos para guiar el entrenamiento del clasificador. Concretamente, cuando hablamos de clasificación predictiva, disponemos de datos para los cuales ya conocemos su clase o *target* de salida, y es en base a esta información que el sistema es capaz de obtener dicho conocimiento para categorizar nuevas entradas de datos.

La construcción de un buen modelo de clasificación implica, tras su entrenamiento, evaluar el sistema para nuevas entradas supervisando el resultado obtenido respecto a la verdadera clase de la que se tenía constancia en el conjunto de entrenamiento original. Sobre esta información, se calcula el rendimiento del sistema, con lo que podremos afirmar si hemos obtenido un clasificador eficaz y preciso.

Las características del conjunto de entrenamiento utilizado para aprender el clasificador son determinantes a la hora de guiar el aprendizaje correctamente. Éste puede verse perturbado o alterado si existen ciertas anomalías respecto a la estructura de los datos de entrenamiento. En este trabajo en concreto, hablamos de cuando el conjunto de instancias no presenta una distribución de clases homogénea o, dicho de otra manera, dicha distribución de las clases no está balanceada y una queda representada en menor medida que la otra. Se dice que la distribución de las clases de un conjunto de datos no es homogénea, si el número de ejemplos difiere mucho de una clase a otra. En tal caso, los resultados del aprendizaje pueden llegar a ser incompletos y confusos debido a la falta de información de la clase minoritaria, que se ve solapada por la cantidad de ejemplos de la clase mayoritaria. En aplicaciones reales, puede llegar a ser muy usual dar con escenarios en los que la probabilidad de que una instancia pertenezca a una u otra clase sea muy distinta.

Un ejemplo claro de esto puede ser, en el ámbito médico, la detección o diagnosis de una enfermedad rara. Podría construirse, a partir de datos y variables derivadas de historiales médicos de un conjunto de pacientes, un modelo de clasificación capaz de determinar dicho pronóstico. En la base de datos utilizada para entrenar el modelo se encuentran tanto pacientes enfermos como

pacientes que no padecen dicha enfermedad. Al tratarse de una enfermedad rara, tal y como puede esperarse, disponemos de mucha más información y casos sobre personas sanas que enfermas. En este contexto, si entrenamos el modelo de clasificación sin tener en cuenta estas características de los datos que tenemos a disposición, lo más probable es que obtengamos un sistema robusto y eficaz a la hora de detectar que una persona NO padece la enfermedad rara. Sin embargo, el modelo no será tan bueno y cometerá más fallos al diagnosticar los casos positivos hacia dicha patología, cuando esto es, realmente, en lo que más interesado se suele estar en detectar.

Tal y como ocurre en el ejemplo recién presentado, en un conjunto de datos no balanceado, generalmente, la clase positiva es aquella en la que más interesado se está en predecir correctamente, y a la vez suele ser la minoritaria y por tanto de la que menos información disponemos. Todo esto frente a la clase negativa, que suele ser más numerosa y de la cual podemos extraer más conocimiento.
Definimos el ratio de desbalanceo o desequilibrio (*imbalance ratio*, IR) como el ratio entre el número de ejemplos de la clase mayoritaria frente al número de ejemplos de la clase minoritaria (Fórmula 1).

$$IR = \frac{nº\ de\ instancias\ de\ la\ clase\ mayoritaria}{nº\ de\ instancias\ de\ la\ clase\ minoritaria}$$

Fórmula 1: Imbalanced ratio

Con esta medida se puede cuantificar y comparar frente a otras, cuán homogénea es la distribución de los datos de los que se dispone.

Dada esta característica de no homogeneidad en la distribución de las clases en el conjunto de datos con el que dotar de conocimiento al sistema, nos encontramos ante un problema de clasificación no balanceado, y será indispensable afrontar una serie de técnicas y pautas de pre-procesamiento de los datos para abordar el aprendizaje y que éste sea eficaz y lo más preciso posible para cada una de las clases a predecir.

En esta línea, surge una nueva propuesta de la mano de investigadores de las universidades Beijing Technology and Business University y School of Economics and Management (Beihang University) para diseñar un algoritmo de entrenamiento que construya un clasificador, cuya innovación y

característica principal residirá en la **integración de técnicas de re-muestreo guiadas por algoritmos de evolución diferencial**. Este nuevo modelo adopta el nombre de **DERS-Boost.**

El objetivo de este trabajo es estudiar y comprobar si en efecto esta nueva propuesta es capaz de obtener un buen rendimiento y dar una solución eficaz a los problemas con conjuntos de datos no balanceados. Motivado también por la complejidad y la combinación de diversas técnicas planteadas en el método, se estudiará la razón de ser de cada una de ellas de manera independiente para constatar así la verdadera aportación de cada una al método en conjunto.
Finalmente, se compararán los resultados obtenidos con aquellos logrados con otros modelos ya conocidos en la bibliografía y en el estado del arte de cara a solucionar problemas con conjuntos de datos no balanceados.

2. PRELIMINARES

2.1 Problemas de clasificación no balanceados

Un problema de clasificación mediante aprendizaje supervisado tiene como objetivo construir un modelo capaz de categorizar correctamente una entrada compuesta de ciertas variables. Esto se logra dotando al sistema de clasificación de conocimiento previo, a partir de un conjunto de datos o entradas estructuradas para las cuales ya se conoce su clase real o de salida. Con un clasificador entrenado, es primordial contrastar los resultados y la eficacia de este en base a datos con los que no haya sido entrenado dicho modelo para evitar cualquier tipo de sobre-aprendizaje y asegurarnos de que el clasificador entrenado es capaz de generalizar para nuevas entradas. Existen para ello diversas metodología o técnicas. Todas ellas giran en torno a realizar un test de clasificación para datos con los que el modelo no haya sido entrenado y de los que se conozca su clase real, para poder calcular así el rendimiento o precisión en base a una métrica de evaluación. Las distintas métricas se explican más adelante en el apartado 2.1.1.

En este trabajo, la técnica seleccionada y que se llevará a cabo para la evaluación de modelos recibe el nombre ***'k-fold cross validation'***. Es una técnica de validación cruzada que divide el conjunto de datos original en *k* subconjuntos. El número de particiones que se tendrán en cuenta será **k=5**, un valor que tiende a ser muy común para este parámetro en el uso de este método de evaluación. Los subconjuntos generados a partir del dataset original deben tener el mismo número de ejemplos, o al menos similar. Dicha partición se realizará aleatoriamente y entre los subconjuntos generados debe existir una intersección vacía, es decir, cada instancia del dataset original se asignará a un sólo subconjunto. Es importante que la distribución de las clases en los subconjuntos para la validación cruzada sea similar a la del conjunto de datos original, por lo que la partición, además de aleatoria se creará usando una técnica de *estratificación*, que es un tipo de técnica que persigue este último objetivo.

Una vez obtenidos los *k* subconjuntos del dataset original, se procede a realizar *k* entrenamientos del modelo a aprender. Cada entrenamiento se llevará a cabo utilizando *k-1* subconjuntos de los generados y el restante se dejará para realizar el test de ese modelo. Tras realizar todos los test en cada uno de los *k* modelos, el rendimiento del clasificador vendrá dado por la media de los resultados obtenidos en cada uno de los test.

En la siguiente figura (Figura 1) se muestra el procedimiento *5-fold cross validation* gráficamente:

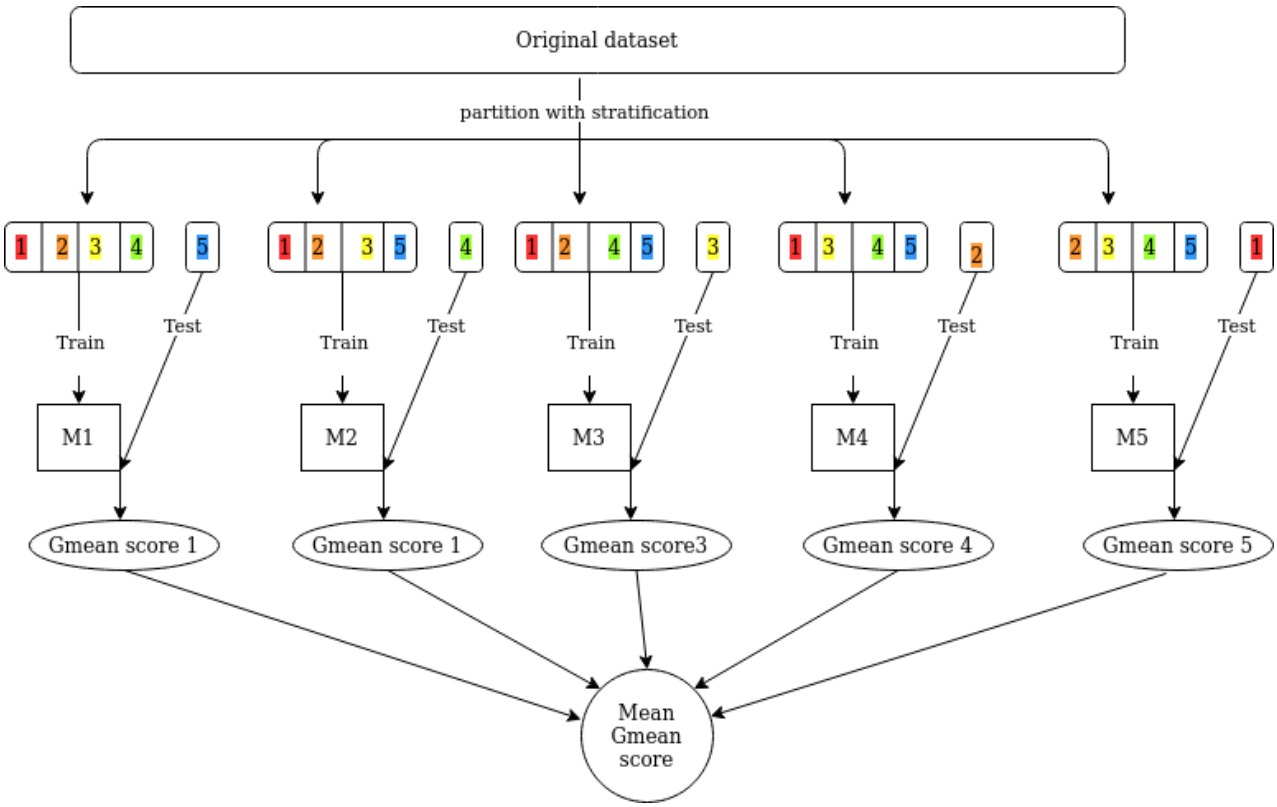

Figura 1: Flujo de trabajo para la técnica 5-fold cross validation. En este caso se evalúa el rendimiento del modelo en base a la media geométrica, para la cual se calcula la media de entre las obtenidas en los cinco entrenamientos+test que se han llevado a cabo.

Mediante este procedimiento obtenemos un rendimiento para el modelo de clasificación aprendido que garantiza que los resultados obtenidos son independientes a los datos con los que se ha entrenado dicho modelo, es decir, que refleja la capacidad de generalización del clasificador de cara a nuevos datos.

En el caso de problemas de clasificación con conjuntos de datos no balanceados, donde existe un importante sesgo en la distribución de cada una de las clases que componen el problema, deben aplicarse métricas de evaluación que tengan en cuenta esta característica y contemplen dicho desbalanceo a la hora de evaluar el rendimiento del modelo.

Asimismo, existen técnicas de pre-procesamiento de datos basadas generalmente en el re-muestreo tanto de ejemplos de la clase mayoritaria como de la minoritaria, cuyo fin es transformar el conjunto de entrenamiento original de manera que la representación de ambas clases quede igualada.

2.1.1 Métricas de evaluación para problemas no balanceados

La medida estándar y una de las más comunes de evaluación de modelos de clasificación es la tasa de acierto o *accuracy* (Fórmula 2). Esta métrica asume una distribución homogénea de las clases en el conjunto de entrenamiento.

$$acc = \frac{n^o\ de\ ejemplos\ clasificados\ correctamente}{n^o\ total\ de\ ejemplos\ clasificados} * 100$$

Fórmula 2: Accuracy

Tras entrenar un modelo de clasificación a partir de un conjunto de datos no balanceado y evaluar su rendimiento utilizando esta medida, a pesar y en el caso de haber obtenido un buen resultado, este ratio de precisión puede y seguramente no refleje la calidad del modelo, ya que los aciertos respecto a la clase negativa son muchos, al ser la más numerosa y disponer de muchos ejemplos de esta clase. Esto hace compensar y posiblemente ocultar la precisión respecto a la clase positiva, que generalmente es la que más interesados estamos en acertar, y en la que posiblemente se estén cometiendo más fallos.

Ante este problema, lo más común es recurrir a otro tipo de medidas más complementarias a la hora de evaluar el rendimiento del sistema de clasificación. Se tratan de métricas que contemplan y muestran más información acerca de la distribución de las clases en el problema.

La **matriz de confusión** (Tabla 1), descompone los resultados de la evaluación de un sistema clasificador para un problema de dos clases. Se obtienen cuatro tipos de resultados:

	Predicción positiva	**Predicción negativa**
Clase real positiva	Verdadero positivo (VP)	Falso negativo (FN)
Clase real negativa	Falso positivo (FP)	Verdadero negativo (VN)

Tabla 1: Matriz de confusión

Verdadero positivo (True positive): instancia de la clase positiva clasificada como positiva.
Falso positivo (False positive): instancia de la clase negativa clasificada como positiva.
Verdadero negativo (True negative): instancia de la clase negativa clasificada como negativa.
Falso negativo (False negative): instancia de la clase positiva clasificada como negativa.

A partir de esta descomposición de la información se definen los siguientes indicadores que evalúan el rendimiento del clasificador respecto a cada clase:

- **Recall** o **True Positve Rate**: es el ratio de los ejemplos positivos clasificados correctamente (Fórmula 3).

$$TPR = Recall = \frac{VP}{VP + FN}$$

Fórmula 3: True Positive Rate o Recall

- **Especificidad** o **True Negative Rate**: ratio de ejemplos de la clase negativa clasificados correctamente (Fórmula 4).

$$TNR = Especificidad = \frac{VN}{VN + FP}$$

Fórmula 4: True Negative Rate o Especificidad

- **Precisión:** esta métrica, al igual que el Recall o TPR, pone el foco en la clase positiva o minoritaria, ya que tal y como se ha comentado suele ser la clase más importante a la hora de predecir correctamente (Fórmula 5). En este caso la *precisión* hace referencia al ratio de ejemplos clasificados en la clase positiva y que son realmente de dicha clase.

$$Precisión = \frac{VP}{VP + FP}$$

Fórmula 5: Precisión

Del mismo modo, puede evaluarse el comportamiento negativo o erróneo del clasificador:
- **False positive rate**: ratio de ejemplos positivos clasificados erróneamente (Fórmula 6).

$$FPR = \frac{FP}{FP + VN}$$

Fórmula 6: False Positive Rate

- **False negative rate**: ratio de ejemplos negativos clasificados erróneamente (Fórmula 7).

$$FNR = \frac{FN}{FN + VP}$$

Fórmula 7: False negative rate

Según el contexto y las características externas del problema que se esté afrontando, será más o menos interesante utilizar una u otras métricas a la hora de evaluar el clasificador. En general, se suele recurrir a aquellas medidas que ponen el foco el la clase positiva, tales como el *Recall* y/o la *Precisión*, al ser esta en la mayoría de los casos la clase más importante a predecir correctamente y en la que menos margen de error se tiende a permitir.

En este sentido, aparece un nuevo indicador que reúne la información de las dos métricas previamente mencionadas en la evaluación del rendimiento hacia la clase positiva:

- **F-SCORE**: esta métrica contempla el balance entre la *Precisión* y el *Recall*, donde el valor de β influirá en la ponderación que se desea otorgar a cada una de estas medidas (Fórmula 8).

$$F_\beta = (1 + \beta^2) \cdot \frac{Precision \cdot Recall}{\beta^2 \cdot Precision + Recall}$$

Fórmula 8: F-score

Comúnmente, se usa $\beta=1$ para dotar del mismo peso tanto a la *Precisión* como al *Recall* (Fórmula 9).

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Fórmula 9: F1-Score

Si $\beta<1$, se le estará dando mayor importancia a la *Precisión*; mientras que si $\beta>1$, se tendrá más en consideración el *Recall*.

No obstante, en la mayoría de casos y aplicaciones reales es necesario obtener un buen rendimiento para ambas clases que componen el problema de clasificación. Por ello, se recurre a medidas que establecen un balance entre los indicadores mencionados al principio que hacen referencia a cada una de las clases y así evaluar el rendimiento del modelo para las dos clases en conjunto. Se

conocen como métricas balanceadas. Una de ellas es el caso de la **media geométrica**. Esta medida comprende el balance entre el Recall, que pone el foco en los aciertos de la clase positiva, y el TNR, que lo hace para los de la negativa (Fórmula 10).

$$Gmean = \sqrt{TNR \cdot Recall}$$

Fórmula 10: Media geométrica entre el Recall y el TNR

2.1.2 Técnicas y soluciones frente a problemas de clasificación con conjuntos de datos no balanceados

Los conjuntos de datos que presentan un alto índice de IR, comprenden al mismo tiempo una serie de características intrínsecas o propias y adicionales que derivan en un aprendizaje 'engañoso' y que por lo tanto, es necesario afrontar.

Por un lado, el hecho de disponer de una cantidad reducida de instancias para la clase minoritaria, hace que las reglas de clasificación y el aprendizaje del modelo queden altamente especializados en estas instancias de entrenamiento, con lo que el clasificador no terminaría de generalizar bien para nuevos ejemplos de esa clase. Además, los ejemplos de la clase minoritaria pueden verse altamente solapados por los de la clase mayoritaria, y en ocasiones quedar relegados a ser simples ejemplos ruidosos de la clase negativa y no como relevantes de la clase positiva.

Por otro lado, es también frecuente disponer de un conjunto de datos no balanceado en el cual la clase minoritaria se presente en forma de pequeños clusters disjuntos, lo que hace aún más dificultoso la tarea de considerar y diferenciar una de estas instancias como ejemplo de la clase minoritaria o ejemplo ruidoso.

Para hacer frente a estas dificultades, es común en la bibliografía sobre problemas no balanceados recurrir a **técnicas de re-muestreo** que traten de transformar el conjunto de datos original para obtener una muestra de ejemplos más adecuada con la que entrenar el clasificador. A continuación se presentan las técnicas más utilizadas, categorizadas según el tipo de re-muestreo que lleven a cabo.

Técnicas de undersampling

Reducen el conjunto de entrenamiento obviando ejemplos de la clase mayoritaria.

- **RUS (Random Undersampling) [16]:** método no heurístico que elimina aleatoriamente ejemplos de la clase mayoritaria con el fin de balancear el conjunto de entrenamiento. Se seleccionan tantos ejemplos a eliminar hasta conseguir igualar el número de ejemplos de la clase minoritaria.

La mayor ventaja de esta técnica reside en su simpleza y sencillez de cara a la implementación. Entre las desventajas, encontramos que se eliminan con la misma probabilidad tanto ejemplos seguros y fronterizos, necesarios para el correcto aprendizaje del clasificador, como instancias ruidosas y ejemplos redundantes, que son aquellos de los que realmente deseamos prescindir para el entrenamiento. Con una selección aleatoria puede perderse información relevante y potencialmente útil y confundir en algunos casos el aprendizaje.

No obstante, es un algoritmo que como muchos otros con base aleatoria, ha demostrado ser capaz obtener buenos resultados para un gran rango de aplicaciones y es utilizado muy frecuentemente debido a también a su rapidez en tiempo de ejecución.

- **TOMEK-LINKS [9]:** el objetivo de este método es realizar una limpieza de aquellos ejemplos de la clase mayoritaria que invadan zonas o clusters de la clase minoritaria y, por lo tanto, sean considerados como ruido.

Dado un ejemplo e_i de una clase y otro ejemplo e_j de otra clase, el par (e_i, e_j) es considerado Tomek-Link (TL) si no existe ningún otro ejemplos entre ellos.

Figura 2: Tanto 'a' como 'b' son ejemplos de la misma clase (verde). 'c' pertenece a la clase 'azul'. El par (a, c) no es TL ya que existe un ejemplo b entre ellos. Por el contrario, el par (b, c) sí que es TL.

En tal caso, deducimos que o bien los ejemplos forman parte de la frontera de decisión, y por lo tanto debemos mantenerlos para entrenar el clasificador, o bien uno de ellos es ruido y debe eliminarse.

El algoritmo analiza cada par de ejemplos de clases distintas y en caso de que se trate de un TL, el ejemplo de la clase mayoritaria será eliminado.
Con esta técnica, los ejemplos que resultan eliminados suelen ser muy pocos y el sub-muestreo no genera grandes cambios en el conjunto de entrenamiento. Por esta razón, Tomek-Links no se suele utilizar como técnica exclusiva de sub-muestreo, sino que acostumbra más a llevarse a cabo como método complementario para técnicas de re-muestreo híbridas en las que realiza una ligera limpieza de datos al final del proceso.

- ENN (Edited Nearest Neighbour) [10]: este método elimina ejemplos tanto de la clase mayoritaria como de la clase minoritaria y es generalmente usado, al igual que Tomek-Links, cuando deseamos realizar una limpieza de datos no demasiado grande para todo el conjunto de entrenamiento.
Para cada ejemplo e_i se aplica el algoritmo 3NN, es decir, se clasifica e_i en base a sus la mayoría de votos de entre sus 3 vecinos más cercanos. Si e_i se falla en está clasificación es eliminado.
En general, esta técnica suele eliminar más ejemplos que Tomek-Links.

Técnicas de oversampling

Amplían el conjunto de entrenamiento añadiendo ejemplos de la clase minoritaria.

- ROS (Random Oversampling) [16]: crea copias aleatorias de ejemplos de la clase minoritaria hasta conseguir balancear el conjunto de entrenamiento. Simple, aunque puede llevar a un incremento del sobre-aprendizaje.

- SMOTE (Synthetic Minority Oversampling Technique) [6]: el objetivo de SMOTE es generar ejemplos de la clase minoritaria a través de la interpolación de instancias de esta misma clase que se encuentren cerca en el espacio dimensional del problema. Estos nuevos ejemplos se conocerán como ejemplos ‘sintéticos’. Con una mayor cantidad de instancias de la clase minoritaria, no sólo se consigue balancear el conjunto de datos original, sino que también evita el sobre-aprendizaje del clasificador respecto a esa clase, al ser muy pequeño el número de ejemplos del que se disponía originalmente.
SMOTE (Algoritmo 1) puede implementarse para generar la cantidad de instancias sintéticas que en cualquier caso se deseen, aunque la implementación más común resulta en la creación del número de ejemplos que permitan balancear el conjunto de datos y que igualen a los de la clase mayoritaria.

Algoritmo SMOTE

Entrada:

Conjunto de entrenamiento $S = \{(x_i, y_i)\}$, $i = \{1, 2, \ldots, N\}$, $y_i \in \{-1,1\}$, $x_i=(x_{i1}, \ldots, x_{im})$
k: número de vecinos más cercano a calcular

$S_{syn} = \{\}$ Inicializar el conjunto de ejemplos sintéticos como un conjunto vacío

For $x_i \in S$ donde $y_i=1$ hacer Para cada ejemplo de la clase minoritaria
  $n_1, \ldots, n_k$ = nearest_neighbors(x_i,k) Calcular los k vecinos más cercanos de x_i
  n_syn = floor(IR(S))-1 Número de ejemplos a crear
  For t=1 to n_syn do
    n_a = random_select($n_1, \ldots, n_k$) Seleccionar aleatoriamente uno de los k vecinos más cercanos de x_i
    for j=1 to m do Para cada atributo de x_i
      $dist_j = n_{aj} - x_{ij}$ Calcular la distancia entre x_{ij} y su vecino
      $x\_syn_j = x_{ij} + rand([0,1])*dist_j$ Crear valor para el atributo del ejemplo sintético

    $S_{syn} = S_{syn} \cup \{x\_syn\}$ Añadir el nuevo ejemplo al conjunto de ejemplos sintéticos

Salida: S_{syn}

Algoritmo 1: SMOTE (Synthetic Minority Oversampling Technique)

Técnicas híbridas

Combinan ambas técnicas de *undersampling* y *oversampling* para obtener el nuevo conjunto de entrenamiento.

- SMOTE + TOMEK-LINKS [11]: con SMOTE se mejoran los clusters de la clase minoritaria y se consigue balancear el conjunto de datos. Tomek-Links se utiliza para una posterior limpieza de datos: elimina los ejempos de la clase mayoritaria que invadan las zonas de la clase minoritaria y prescinde de las instancias de la clase minoritaria que hayan podido ser generadas y que ahora invadan zonas de la clase mayoritaria.

- SMOTE + ENN [12]: otra posible combinación. En este caso, el algoritmo ENN suele mejorar la limpieza de datos ya que tiende a eliminar más ejemplos que con Tomek-Links.

Ensemble learning para problemas no balanceados

Las técnicas de re-muestreo descritas anteriormente pueden ser finalmente integradas dentro de algoritmos de *ensembles* de clasificadores, más conocidos como *ensemble learning*, para potenciar

el aprendizaje de un clasificador robusto. El aprendizaje combinado de modelos de predicción es un recurso que ha demostrado obtener buenos resultados en cuanto a mejorar el rendimiento de estos sistemas frente a técnicas donde sólo se entrena un único modelo de predicción.

- La base: Ada-Boost

Dentro del marco de *ensemble learning*, los algoritmos de *boosting* son un tipo de algoritmo iterativo en los que la idea es construir un modelo de clasificación fuerte a partir de la combinación de distintos clasificadores débiles o base. En cada iteración se entrenará un clasificador débil y se le otorgará un peso en función al error cometido para los datos de entrenamiento. El clasificador robusto final tomará su decisión combinando las salidas de cada uno de los clasificadores base y teniendo en cuenta, al mismo tiempo, el peso de cada uno de éstos. De esta manera, se consigue dotar de una mayor diversidad de conocimiento al clasificador fuerte a partir de los distintos fallos que puedan cometer los múltiples clasificadores base.

Ada-Boost (Y. Freund, R. Schapire, 1995) [13] forma parte de los algoritmos de la familia *boosting*. Consiste en entrenar cada uno de los clasificadores base a partir de un mismo conjunto de entrenamiento, pero en cada iteración se van actualizando los pesos según los ejemplos que se han fallado con el clasificador anteriormente entrenado, haciendo más hincapié en aquellas instancias clasificadas erróneamente (Algoritmo 2).

Algoritmo: Ada-Boost

Entrada:

Conjunto de entrenamiento S = $\{(x_i, y_i)\}$, $i = \{1, 2, \ldots, N\}$

T: número de iteraciones

I: clasificador base

$D_1(i) = 1/N \quad \forall i \in \{1, \ldots, N\}$ Asignar pesos de manera uniforme

for i = 1 .. T do

$h_t = I(D_t, S)$ Entrenar clasificador débil con los pesos

$E_t = \sum_{i, y_i \neq h_t(x_i)} D_t(i)$ Calcular el error como la suma de los pesos de los ejemplos mal clasificados

$\alpha_t = \frac{1}{2} \ln\left(\frac{1-E_t}{E_t}\right)$ Calcular el peso del clasificador débil

$D_{t+1}(i) = D_t(i) \cdot e^{(-\alpha_t \cdot h_t(x_i) \cdot y_i)} \quad \forall i \in \{1, \ldots, N\}$ Actualizar pesos

$D_{t+1}(i) = \frac{D_{t+1}(i)}{\sum_{i=1}^{N} D_{t+1}(i)}$ Normalizar los pesos

endfor

$H_{final} = sign(f(x)) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$ Clasificador final compuesto por múltiples clasificadores débiles

Salida: H_{final}

Algoritmo 2: Ada-Boost

A partir de esta estructura algorítmica, se integran dentro del bucle y previo al entrenamiento del clasificador base, las técnicas de re-muestreo que pretenden mejorar el aprendizaje para problemas de clasificación con conjuntos de datos no balanceados. A continuación se presentan algunos de los métodos que ya se encuentran recogidos en la bibliografía.

- RUSBoost (Random Undersampling Boosting)

RUS-Boost (Seiffert, Khoshgoftaar, Van Hulse, & Napolitano, 2010) [8], tal y como su nombre indica, integra dentro de cada iteración o *boosting* un sub-muestreo aleatorio de las instancias de la clase mayoritaria, aplicando para ello el algoritmo RUS, con el objetivo de balancear la distribución de las clases en el conjunto de datos de entrenamiento (Algoritmo 3).

Algoritmo: RUS-Boost

Entrada:

Conjunto de entrenamiento S = $\{(x_i, y_i)\}$, $i = \{1, 2, \ldots, N\}$

T: número de iteraciones

I: clasificador base

$D_1(i) = 1/N$ $\forall i \in \{1, \ldots, N\}$ Asignar pesos de manera uniforme

for i = 1 .. T do

S' = RUS(S) **Random undersampling de las instancias de la clase mayoritaria**

$D'_t = \{ d_i \in D_t : (x_i, y_i) \in S' \}$ **Distribución de pesos para los ejemplos seleccionados**

$h_t = I(D'_t, S')$ Entrenar clasificador débil con los pesos

$E_t = \sum_{i, y_i \neq h_t(x_i)} D_t(i)$ Calcular el error como la suma de los pesos de los ejemplos mal clasificados (del conjunto de datos original)

$\alpha_t = \frac{1}{2} \ln\left(\frac{1-E_t}{E_t}\right)$ Calcular el peso del clasificador débil

$D_{t+1}(i) = D_t(i) \cdot e^{(-\alpha_t \cdot h_t(x_i) \cdot y_i)}$ $\forall i \in \{1, \ldots, N\}$ Actualizar pesos

$D_{t+1}(i) = \frac{D_{t+1}(i)}{\sum_{i=1}^{N} D_{t+1}(i)}$ Normalizar los pesos

endfor

$H_{final} = sign(f(x)) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$ Clasificador final compuesto por múltiples clasificadores débiles

---

Salida: H_{final}

Algoritmo 3: RUS-Boost

- SMOTE-Boost

SMOTE-Boost (Chawla, Lazarevic, Hall, & Bowyer, 2003) [7] integra el algoritmo SMOTE para que en en este caso el re-muestreo se haga sobre los ejemplos de la clase minoritaria, de cara a entrenar un nuevo clasificador base (Algoritmo 4).

Algoritmo: SMOTE-Boost

Entrada:

Conjunto de entrenamiento $S = \{(x_i, y_i)\}$, $i = \{1, 2, \ldots, N\}$

T: número de iteraciones

I: clasificador base

$D_1(i) = 1/N \quad \forall i \in \{1, \ldots, N\}$ — Asignar pesos de manera uniforme

for i = 1 .. T do

S_{syn} = SMOTE(S) — **Generar conjunto de ejemplos sintéticos**

$S' = S \cup S_{syn}$

$D'_t = \{ d_i \in D_t : (x_i, y_i) \in S' \}$ — **Distribución de pesos para los ejemplos seleccionados**

$h_t = I(D'_t, S')$ — Entrenar clasificador débil con los pesos

$E_t = \sum_{i, y_i \neq h_t(x_i)} D_t(i)$ — Calcular el error como la suma de los pesos de los ejemplos mal clasificados (del conjunto de datos original)

$\alpha_t = \frac{1}{2} \ln\left(\frac{1-E_t}{E_t}\right)$ — Calcular el peso del clasificador débil

$D_{t+1}(i) = D_t(i) \cdot e^{(-\alpha_t \cdot h_t(x_i) \cdot y_i)} \quad \forall i \in \{1, \ldots, N\}$ — Actualizar pesos

$D_{t+1}(i) = \frac{D_{t+1}(i)}{\sum_{i=1}^{N} D_{t+1}(i)}$ — Normalizar los pesos

endfor

$H_{final} = sign(f(x)) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$ — Clasificador final compuesto por múltiples clasificadores débiles

---

Salida: H_{final}

Algoritmo 4: SMOTE-Boost

De manera análoga, se pueden diseñar otros algoritmos integrando en cada uno de ellos una técnica de re-muestreo diferente. En este trabajo se implementaron para la fase de experimentación y comparación los algoritmos **RUS-Boost**, **SMOTE-Boost**, **SMOTE-Tomek-Boost** y **SMOTE-ENN-Boost.** Estos dos últimos, siguen exactamente la misma estructura que los anteriores. Llevan a cabo el algoritmo SMOTE para remuestrear los ejemplos de la clase minoritaria y después incluyen, en cada caso, la técnica de sub-muestreo correspondiente (Tomek-Links en el caso de SMOTE-Tomek-Boost y ENN en el caso de SMOTE-ENN-Boost) antes de entrenar el clasificador base en cada boosting.

2.2 Introducción a la computación evolutiva y diferencial

DERS-Boost, el método para afrontar problemas de clasificación no balanceados que se describe en el siguiente apartado, y cuyo rendimiento es el objetivo a estudiar de esta memoria, integra técnicas de re-muestreo guiadas por computación evolutiva diferencial (R. Storn, K. Price, 1997) [14]. Es por esto que a continuación se hace una introducción a la teoría de esta materia.

La computación evolutiva tiene como objetivo dar solución a problemas complejos de optimización por medio de un algoritmo de búsqueda adaptativa y heurística, tratando de reproducir el comportamiento de la selección natural y la genética. Los algoritmos evolutivos hacen uso de la información acumulada y registrada durante la búsqueda para guiar la misma hacia la solución más óptima. Además incluyen un factor de aleatoriedad, también presente en los procesos de evolución de la naturaleza, que otorga diversidad y es capaz de guiar la búsqueda por otras direcciones en espacios de soluciones muy grandes, evitando así que la búsqueda quede estancada en entornos locales de soluciones sub-óptimas.

Para el correcto diseño de un algoritmo evolutivo, el problema de optimización a resolver se debe definir y formalizar matemáticamente en términos de una o varias funciones. Esta formalización es lo que se conoce como **función objetivo** y es en lo que se está interesado/a en maximizar o minimizar.
Cada posible solución dentro del espacio de búsqueda puede ser evaluada a través de la función objetivo y calcular así su *ajuste*, también denominado *adaptación* o *fitness*, que determina cómo de buena es dicha solución para el problema considerado (en problemas de maximización buscamos soluciones con un *fitness* elevado y en problemas de minimización, por el contrario, un *fitness* bajo).
Los algoritmos genéticos comienzan inicializando un conjunto de tamaño preestablecido de soluciones posibles dentro del espacio de búsqueda, a lo que nos referiremos como ‘población’ inicial. Dentro de una población, cada solución candidata se conoce como cromosoma. Esta inicialización suele ser generalmente aleatoria.
Los cromosomas no son más que una representación o codificación de las posibles soluciones al problema en forma de genes (elementos del cromosoma). Lo más común es encontrarnos con vectores n-dimensionales que den una representación del tipo de solución a la que se desee llegar.

A partir de la población inicial, se aplican de manera iterativa los operadores de **cruzamiento** y **mutación** y **reemplazo** para hacer evolucionar dicha población hacia generaciones con cromosomas más óptimos en relación al *fitness* definido.
Los operadores de cruce, tal y como su nombre indica, resultan en un nuevo individuo o 'hijo' a partir de dos cromosomas progenitores seleccionados de la población actual.
Los operadores de mutación se encargan de alterar ciertos genes de un individuo e introducir así cierto factor de aleatoriedad en la búsqueda.
Por último, los métodos de reemplazo de supervivientes presentan alternativas a la hora de sustituir a los padres o individuos de la población actual y seleccionar de entre los hijos recientemente creados, para la generación futura en el proceso de evolución.

Se debe establecer también una condición de parada que limite el proceso iterativo del algoritmo evolutivo. Esta condición se diseña en función de las características del problema a resolver. Entre otras opciones, puede referirse a, o bien se ha dado con la solución exacta en uno de los cromosomas (se ha alcanzado el fitness óptimo), o bien se ha dado con una solución aproximada suficientemente buena en función de alguna tolerancia o error permitido, o bien se ha superado un número de iteraciones predefinido y se toma como solución el cromosoma con mejor fitness hallado hasta el momento.
La figura 3 muestra el esquema general que puede seguir un algoritmo evolutivo.

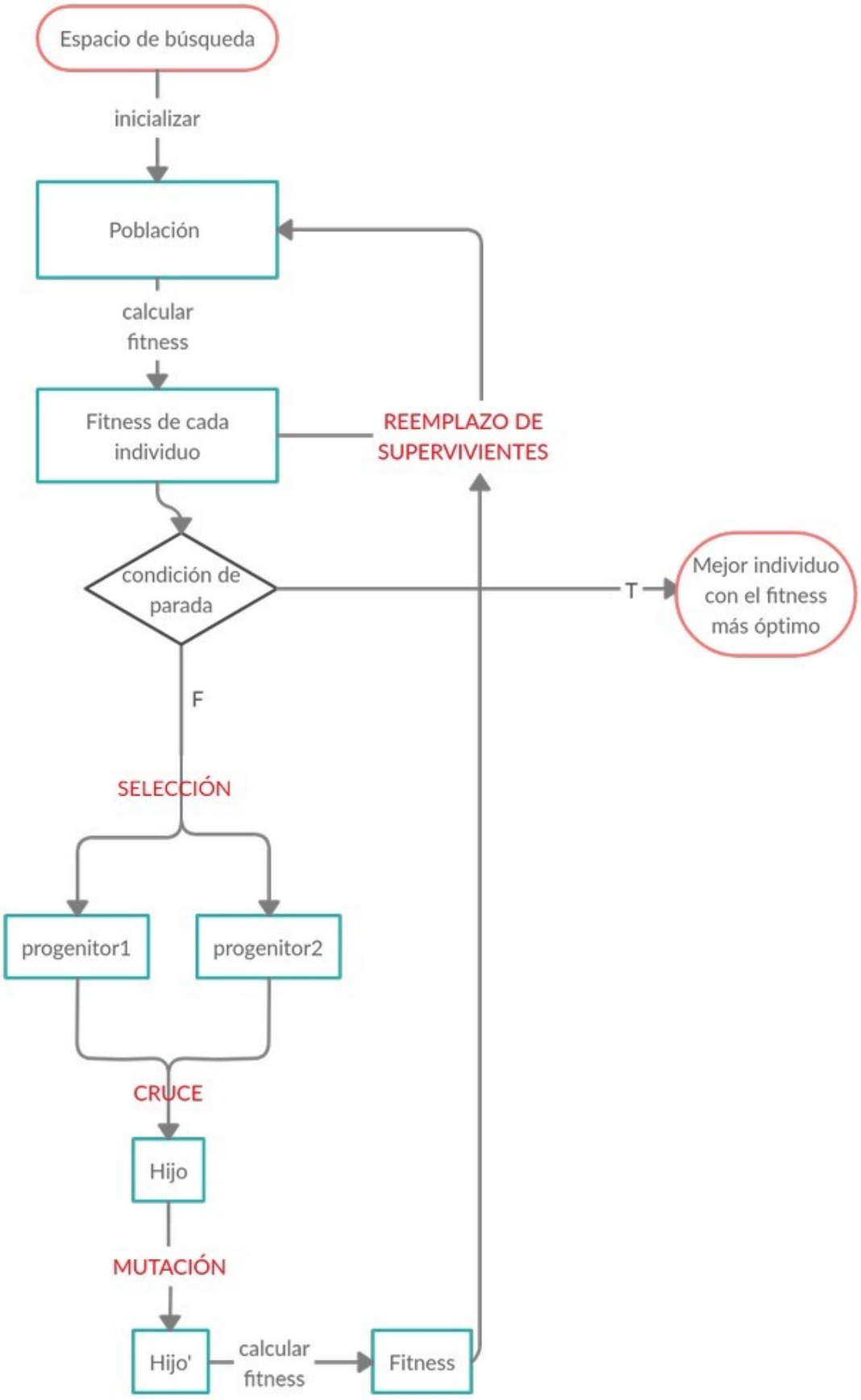

Figura 3: Flujo de trabajo de un algoritmo genético

Evolución diferencial

Los modelos evolutivos diferenciales introducen una nueva variante en el esquema de evolución previamente descrito. La característica principal radica en la **mutación previa al cruce** de dos cromosomas padre, **basada en la diferencia entre** cromosomas seleccionados aleatoriamente de la población actual. A través de la mutación diferencial, se explora el espacio cercano a los vectores pertenecientes a la población actual, se consideran nuevos vectores (vectores mutantes) dentro de ese espacio, que se utilizarán para realizar los cruces con los vectores padre, pertenecientes a la población actual, y obtener así los llamados vectores de prueba, que serán los candidatos a sustituir a los vectores padre con el fin de mejorar la población.

El esquema secuencial que sigue un algoritmo evolutivo diferencial es el siguiente:

1. **Inicialización de la primera generación**

2. **Hasta (condición de parada) hacer:**
   Para cada cromosoma (vector objetivo, V_o) de la población actual
   i. **Vector mutante (V_d) ← mutacion_diferencial(V_o)**
   ii. **Vector hijo o de prueba (V_p) ← cruce(V_o, V_d)**
   iii. **Población actualizada ← reemplazo_supervivientes(V_o,V_p)**

- Mutación diferencial: da como resultado el vector donante o mutante V_d (Fórmula 11), que será empleado para después realizar el cruce con un vector padre, también denominado vector objetivo V_o, de la población actual. El vector donante se calcula de la siguiente manera: se obtienen aleatoriamente tres cromosomas de la población actual X_1, X_2 y X_3. A partir de la diferencia escalada por un factor F de dos de éstos, y sumada al tercer vector, se consigue un nuevo vector, cercano en el espacio a los tres previamente aludidos.

$$V_d = X_1 + F(X_2 - X_3)$$

Fórmula 11: Cálculo del vector donante o mutante

F suele tomar valores entre 0,4 y 1.

Cruce: se obtiene un vector hijo o vector de prueba V_P a partir del vector objetivo V_o y el vector donante V_d calculado en el paso anterior. Se pueden llevar a cabo distintos métodos y/o operadores

de cruce. Uno de los más comunes y simples es el cruce binomial o uniforme (Fórmula 12), el cual elige aleatoriamente si cada valor de V_p, es decir cada gen del vector hijo, se toma de V_o o de V_d.

$$v_{p,j} = \begin{cases} v_{d,j} & \text{si } (rand_j[0,1] <= C_r) \text{ o } (j = j_{rand}) \\ v_{o,j} & \text{en otro caso} \end{cases}$$

Fórmula 12: Cruce binomial o uniforme

C_r es el ratio de cruce predeterminado, que toma valores entre 0 y 1 para establecer la probabilidad de que un gen del cromosoma padre sea mutado o no a la hora de generar el vector de prueba. Por otro lado, j_{rand} constituye un índice o gen aleatorio en {1, …, D} que asegura que el vector de prueba toma al menos un valor del vector donante.

Reemplazo de supervivientes: de manera también muy común y simple, si el fitness del vector de prueba V_P generado se ajusta mejor que el fitness del vector objetivo V_o, entonces este último será sustituido por su descendiente en la población para la siguiente generación. No obstante, pueden aplicarse nuevamente otras estrategias de reemplazo de supervivientes.

2.3 Herramientas para el desarrollo

A continuación se exponen las herramientas en calidad de lenguaje de programación y librerías que has sido utilizadas para la implementación y llevar a cabo las pruebas de ejecución en este trabajo.

El lenguaje de programación escogido ha sido **Pyhton 3** (https://www.python.org/doc/) por tener a disposición un sólido conjunto de de librerías orientadas a *Machine Learning*, creación de modelos predictivos, etc.

La librería ***scikit-learn*** de Python es muy común y recurrida a la hora de programar modelos de predicción, y es el recurso que predomina a lo largo de las líneas de código que se han desarrollado. Por ejemplo, existe una clase que implementa el clasificador Ada-Boost: *AdaBoostClassifier*()

Otra librería específica resulta ser ***imbalanced-learn***, que como su nombre indica, es muy propicia para afrontar el desarrollo de modelos y código de cara a lidiar con problemas de conjuntos no balanceados. Es esta librería la que nos proporciona los métodos ya implementados para llevar a cabo las distintas técnicas de re-muestreo: *RUSBoostClassifier(), SMOTE(), SMOTETomek(), SMOTEENN(),* etc.

Para llevar a cabo la implementación de los algoritmos evolutivos diferenciales que guian las técnicas de re-muestreo en DERSBoost, se ha hecho uso de la librería **DEAP (Distributed Evolutionary Algorithms in Python)**. Este paquete presenta un conjunto de estructuras de datos y herramientas muy interesantes que permiten definir, estructurar y personalizar los aspectos relativos a la programación de cualquier algoritmo evolutivo en Python. La persona que programa define las propias características de los tipos de datos que el problema a resolver requiere, es decir, customiza el tipo de sus cromosomas. Del mismo modo, tiene la oportunidad de diseñar y especificar todos los procesos que componen los métodos evolutivos, desde cómo se realiza la inicialización de la población, hasta los operadores de mutación, cruce, selección, reemplazo, etc. Para estos operadores, existen ya métodos implementados (aquellos más conocidos y que suelen aparecer en la bibliografía), sin embargo, quien diseña el algoritmo es capaz de definir sus propios operadores de cualquier tipo para cada una de las fases de la computación evolutiva.

3. DERS-BOOST: DIFFERENTIAL EVOLUTIONARY GUIDED RESAMPLING

DERS-Boost es un algoritmo propuesto y originalmente recogido en el borrador “Differential evolution-guided resampling combined with Ada-Boost ensemble learning for imbalanced data classification” y redactado por P. Gong, J. Gao, L. Wang y F. Zhang. Este no es un documento oficialmente publicado. El objetivo principal de este trabajo es analizar y comparar los resultados de esta innovadora técnica frente al resto de métodos presentados previamente.
Durante su estudio y observación se detectaron varias erratas, así como cierta insuficiencia en la descripción del método y poca determinación y profundización en algunos aspectos para su total comprensión. Es por esto que se toma la libertad de re-interpretar ciertos apartados y adecuar algunos aspectos a criterio personal, con el fin de esclarecer al máximo posible la técnica llevada a cabo y mantener su propósito principal.

El método propuesto para lidiar con problemas de clasificación no balanceados está compuesto por dos bloques principales (Figura 4): el primero, un sub-muestreo de las instancias de la clase mayoritaria guiado por un algoritmo evolutivo diferencial; el segundo, un aprendizaje combinado, *ensemble learning*, que integra un sobre-muestreo de las instancias de la clase minoritaria, guiado nuevamente por un algoritmo evolutivo diferencial. Todo esto con el fin de obtener conjuntos de datos más representativos y balanceados que favorezcan el entrenamiento de cada uno de los clasificadores base que componen el aprendizaje.

Figura 4: Flujo de trabajo dividido en dos bloques principales del algoritmo DERS-Boost.

3.1 - Bloque 1: sub-muestreo basado en un proceso de clustering, guiado por un algoritmo evolutivo diferencial.

La primera parte del método tiene como objetivo realizar una reducción del conjunto de instancias de la clase mayoritaria con el fin de obtener un subconjunto de datos más balanceado. A rasgos generales, lo ideal sería deshacerse de los ejemplos redundantes de la clase mayoritaria, que son aquellos que se encuentran más alejados de la frontera de decisión. Estas instancias no suelen presentar dudas al modelo a la hora de clasificar y ofrecen poca información adicional a la hora del aprendizaje. Eliminando estos ejemplos obtenemos un conjunto de datos más balanceado y reducimos el tiempo de entrenamiento.

El objetivo es, pues, identificar aquellos ejemplos redundantes en el conjunto de datos de la clase mayoritaria. Para ello, se llevarán a cabo H procesos de clustering (aprendizaje no supervisado) guiados por un algoritmo evolutivo diferencial, con los que se obtendrán parecidas pero distintas particiones del conjunto de datos original, y que, dado un nuevo ejemplo, lo categorizarán a una u otra clase del problema en cuestión. Este proceso se muestra de manera gráfica en la figura 5.

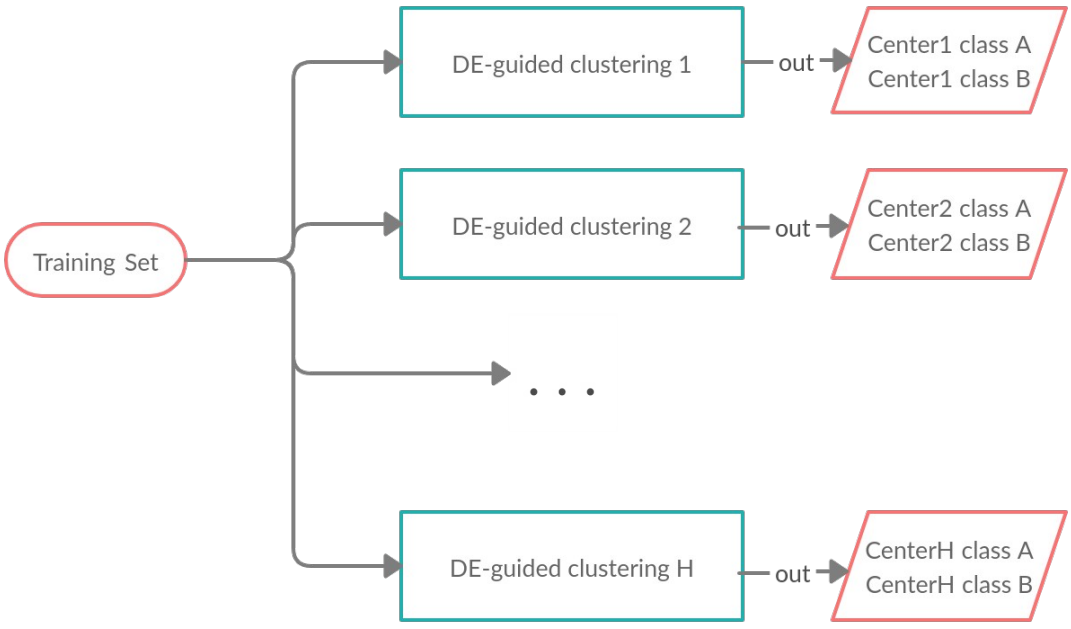

Figura 5: Se llevan a cabo H procesos de clustering a partir del conjunto de datos de entrenamiento inicial. Cada clustering devolverá como resultado dos particiones de dicho conjunto de datos. Este resultado vendrá dado por los centros representativos obtenidos de cada clase.

La utilización de un algoritmo evolutivo diferencial como guía del aprendizaje de los clusters aportará variedad en los resultados debido a la combinación de heurística y aleatoriedad que caracteriza las búsquedas dentro del marco de la computación evolutiva.

La meta de este procedimiento es calcular la ***estabilidad de clustering*** de cada uno de los ejemplos de la clase mayoritaria, para después considerarlos como fronterizos o redundantes. Nos interesa obtener distintas particiones del conjunto de datos para cerciorarnos de que un mismo ejemplo de la clase mayortaria tiende a ser clasificado siempre correctamente incluso en los diferentes clusterings computados anteriormente. Se dice entonces que dicho ejemplo es 'estable' respecto a los clusters inferidos del conjunto de datos inicial.
Definimos la estabilidad de clustering C_i, de un ejemplo s_i como la suma de los aciertos al clasificarlo en cada uno de los clusterings llevados a cabo previamente, dividida por el número total de procesos de clustering (Fórmula 13).

$$C_i = \frac{1}{H} \sum_{t=1}^{H} f(h_t(s_i), y_i)$$

Fórmula 13: Cálculo de la estabilidad de clustering de un ejemplo s_i

La función $h_t(s)$ clasifica el ejemplo s para el clustering t, mientras que y corresponde a la clase real del ejemplo en el conjunto de datos original. La función $f(x,y)$ adopta el valor 1 cuando ambos parámetros x e y coinciden, en otro caso adopta el valor 0. H es el número total de procesos de clustering llevados a cabo. La figura 6 muestra de manera esquemática todo el proceso para llegar a calcula la estabilidad de clustering de los ejemplos de la clase mayoritaria.

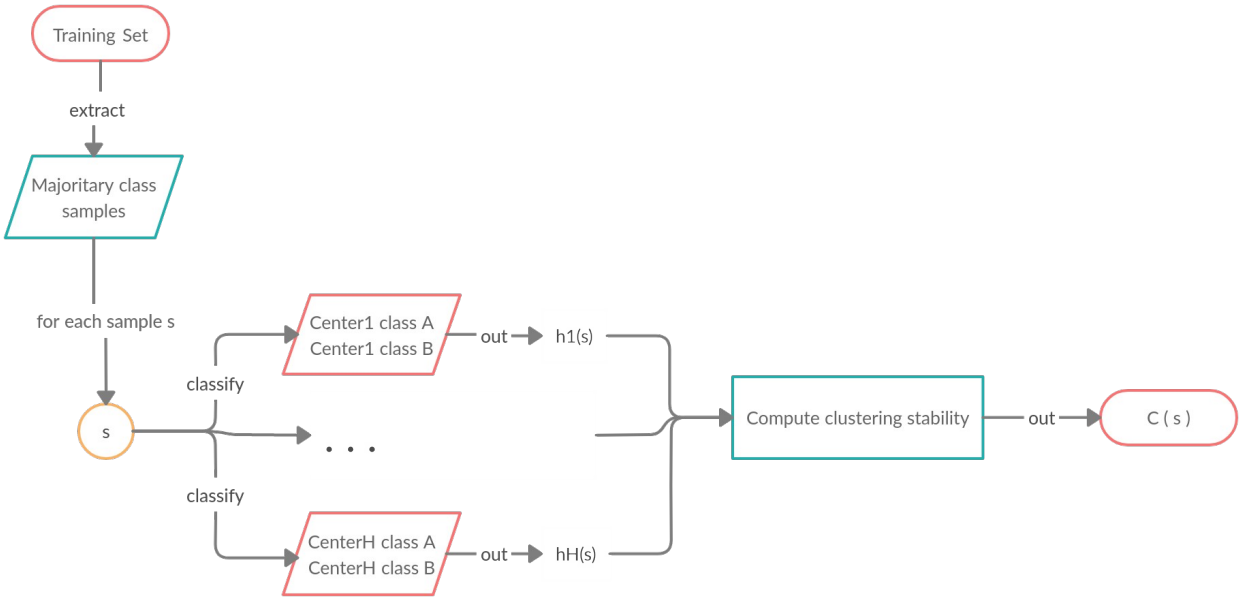

Figura 6: Flujo de trabajo para el cálculo de la estabilidad de clustering de los ejemplos de la clase mayoritaria. Se extraen dichos ejemplos y cada uno de ellos se clasifica en cada uno de los clusterings computados previamente. Después, se calcula su estabilidad de cluster en base a los resultados.

Se establece un umbral de 0,8 para categorizar los ejemplos de la clase mayoritaria. Aquellas instancias con un índice de estabilidad de clustering superior al umbral serán consideradas como instancias redundantes, mientras que aquellas con un índice inferior o igual se considerarán como ejemplos fronterizos con la clase minoritaria.

Las instancias fronterizas formaran todas parte del conjunto de datos con el que se entrenará el clasificador, mientras que se realizará un re-muestreo aleatorio de las instancias consideradas redundantes para seleccionar únicamente un subconjunto de estas.

El flujo de trabajo para este primer bloque, donde el objetivo es la obtención de un nuevo conjunto de datos de entrenamiento sub-muestreando los ejemplos redundantes de la clase mayoritaria a partir del conjunto original, viene representado de manera esquemática en la figura 7.

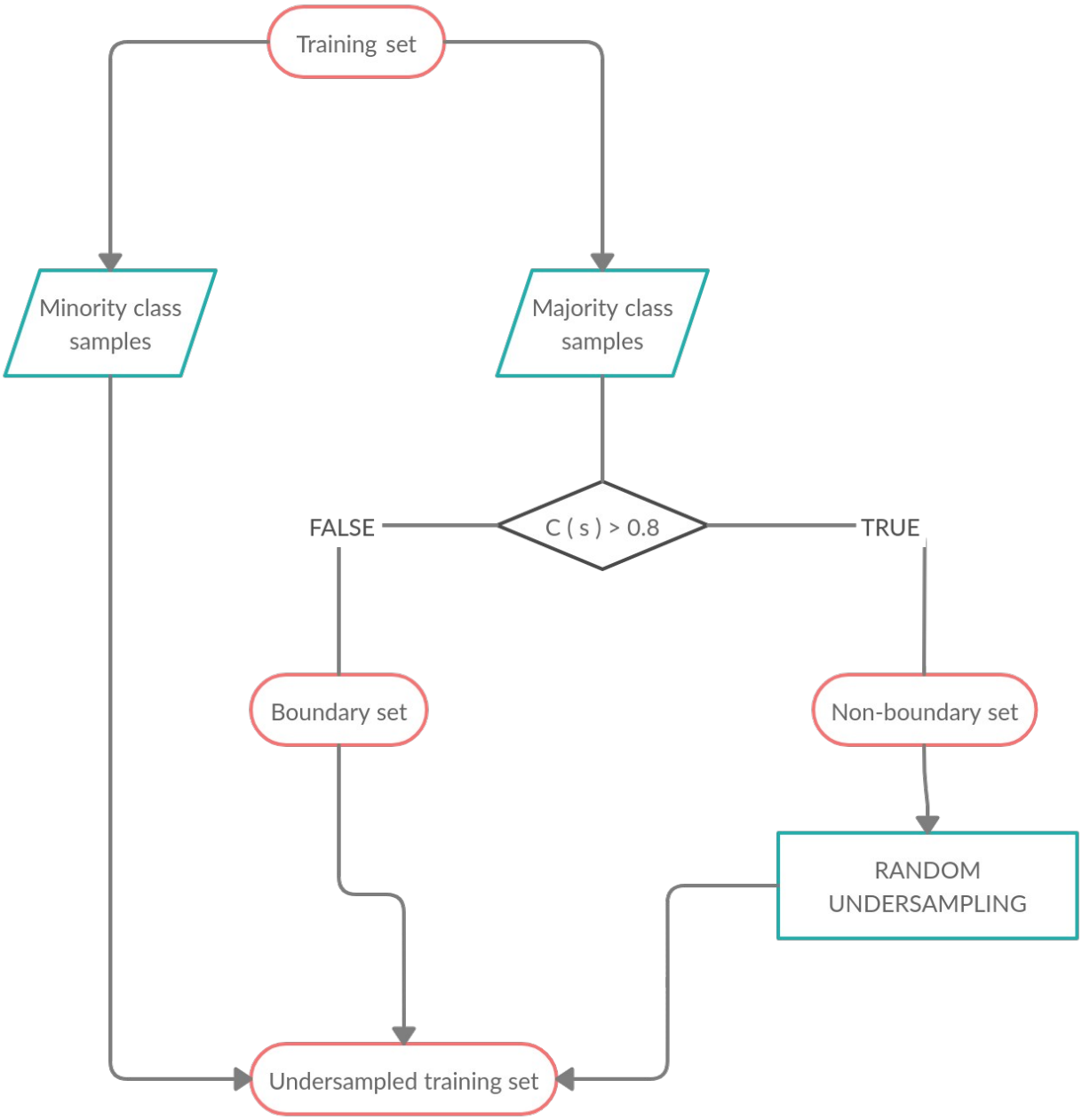

Figura 7: Flujo de trabajo para el sub-muestreo del conjunto de datos inicial. Los ejemplos de la clase minoritaria se mantienen intactos. Los ejemplos de la clase mayoritaria considerados fronterizos también son seleccionados. Se realiza un re-muestreo aleatorio (RUS) con aquellas instancias de la clase mayoritaria consideradas redundantes.

Clustering guiado por un algoritmo evolutivo diferencial

De todo este primer bloque, falta describir los detalles y características particulares que adopta el algoritmo evolutivo diferencial utilizado para llevar a cabo el proceso de aprendizaje no supervisado de clustering, que da como resultado dos particiones del conjunto de datos de entrenamiento.

Para realizar el clustering, el objetivo del algoritmo será encontrar los dos centros relativos a las dos clases del problema que mejor representen cada una de esas clases. Con ese fin, un cromosoma, que codifica una posible solución del problema, viene representado por un vector de tamaño 2F, siendo F el número de atributos/características que forman parte de las instancias del conjunto de datos. Los primeros F genes del cromosoma hacen referencia al centro de la clase mayoritaria y los F siguientes al centro de la clase minoritaria. La representación gráfica de esta codificación se muestre en la figura 8.

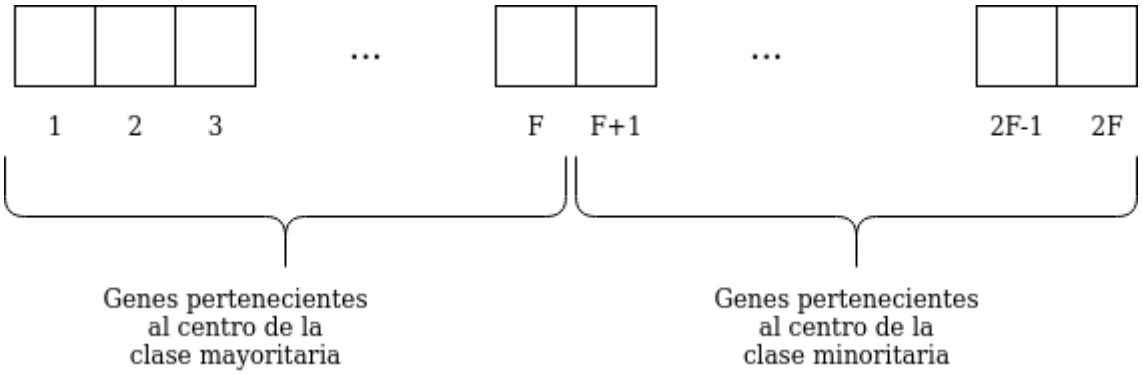

Figura 8: Representación de un cromosoma para el método evolutivo implementado como guía del proceso de clustering

Para dar con los centros que mejor representen las clases del problema, se busca minimizar la distancia dentro de los ejemplos de un mismo cluster. Para ello, se toma como función de fitness la misma función objetivo del algoritmo k-means [15], que busca minimizar la suma de los cuadrados de la diferencia de cada instancia del conjunto de entrenamiento con el centro de su clase (Fórmula 14).

$$fitness(v) = \sum_{i=1}^{2} \sum_{x_j \in S_i} \| x_j - v_{ij} \|^2$$

Fórmula 14: Función de fitness o ajuste

siendo v = (v_{11}, v_{12} , …, v_{1F}, $v_{21,}$ …, $v_{2,2F}$) el cromosoma que codifica los dos centros candidatos.

La primera generación de cromosomas será creada escogiendo, para cada individuo en la población inicial, un ejemplo aleatorio de cada clase.

Por último, como condición de parada del algoritmo se establecerán un número máximo de iteraciones.

3.2 - Bloque 2: sobre-muestreo guiado por un algoritmo evolutivo diferencial, integrado en un método de ensemble con Ada-Boost

En este segundo bloque se procede a entrenar el modelo de clasificación del problema en cuestión. Se plantea entrenar un clasificador fuerte mediante el procedimiento Ada-Boost. Sin embargo, el método DERS-Boost propone un re-muestreo de los ejemplos de las clase minoritaria previo al entrenamiento de cada clasificador débil en cada *boosting*. Este re-muestreo se llevará a cabo mediante el algoritmo SMOTE seguido de un sub-muestreo de los ejemplos sintéticos generados guiado por un algoritmo evolutivo diferencial:

La propuesta original de DERS-Boost plantea generar con SMOTE tantos ejemplos sintéticos de la clase minoritaria como número de instancias haya de esta misma en el conjunto original de datos de entrenamiento. Tras este paso, se dispondría de $N = n^- + 2n^+$ ejemplos en el conjunto de entrenamiento, siendo n^- el número de ejemplos para la clase mayoritaria (sub-muestreado previamente) y n^+ el número de ejemplos para la clase minoritaria. Analizando este planteamiento, se percató de que el hecho de generar tantos ejemplos sintéticos como número de instancias de la clase minoritaria no resuelve el problema de desbalanceo (teneiendo en cuenta un dataset de, por ejemplo, 400 entradas, si bien 370 se corresponden con la clase mayoritaria y 30 con la minoritaria, al duplicar el número de instancias para la segunda a 60, sigue existiendo un fuerte desequilibrio en la distribución de las clases). Por esto, se decide obviar esta restricción sugerida originalmente en la propuesta de DERS-Boost, y en lugar de generar dicho número de sintéticos, se utilizará SMOTE de tal manera que produzca tantas instancias de la clase minoritaria como para balancear e igualar el número de instancias de la clase mayoritaria**.**

A continuación, DERS-Boost propone seleccionar el mejor subconjunto de instancias sintéticas generadas para entrenar el clasificador débil de la correspondiente iteración de Ada-Boost. El objetivo es deshacerse de las posibles instancias sintéticas que hayan sido generadas pero que no resulten realmente beneficiosas para el entrenamiento del clasificador base. Este sub-muestreo de los ejemplos sintéticos vendrá guiado por un algoritmo evolutivo diferencial que se describe a continuación.

Algoritmo evolutivo diferencial para seleccionar el mejor subconjunto de instancias sintéticas con las que entrenar un clasificador débil

En el caso de este proceso evolutivo, los cromosomas vienen representados por un vector del tamaño correspondiente al número de instancias sintéticas de la clase minoritaria generadas en el paso anterior por SMOTE (ns^+). Se trata de un vector binario que recoge únicamente los valores 0 y 1. Un 1 en la posición i-ésima del cromosoma significa que la instancia sintética i-ésima es incluida dentro del conjunto de entrenamiento del clasificador débil. Por el contrario, un 0 indica que la instancia i-ésima no se incluye en el subconjunto de entrenamiento.

El objetivo del algoritmo es encontrar el sub-conjunto de ejemplos óptimo que maximice el rendimiento del clasificador. Por ello, la función de fitness (Fórmula 15) considera un balance entre el rendimiento esperado del clasificador con el subconjunto de entrenamiento seleccionado y la proporción de ejemplos de la clase minoritaria frente a los de la mayoritaria, penalizada por un factor *p* recomendado a tomar el valor de 0,2.

$$fitness(v) = G_{mean} - \left|1 - \frac{ns^+ + \sum_{i=1}^{ns^+} v_i}{n^-} \cdot p\right|$$

Fórmula 15: Función de fitness o ajuste propuesta en DERS-Boost.

En nuestro caso, optaremos por un fitness que optimice el rendimiento del clasificador en términos del f1-score en lugar de la media geométrica tal y como se propone originalmente en la fórmula 15, debido a que el análisis que se llevará a cabo en la sección 5 de experimentación se realizará en

base a esta primera métrica (f1-score). La función de fitness quedaría definida a través de la fórmula 16.

$$fitness(v) = F_1 - |1 - \frac{ns^+ + \sum_{i=1}^{ns^+} v_i}{n^-} \cdot p|$$

Fórmula 16: Función de fitness o ajuste utilizada en la implementación

Se inicializa una población de cromosomas aleatorios para los que se calcula su fitness. Hay que tener presente que cada cómputo del fitness requiere entrenar un clasificador, con lo que aparentemente el método tiende a ser muy costoso en tiempo de ejecución.

Además, los autores proponen introducir un parámetro de auto-adaptación λ (Fórmula 17), que se encarga de modificar el parámetro de ratio de mutación F (que escalaba la diferencia entre dos vectores de la población a la hora de calcular el vector donante) en cada generación (Fórmula 18).

$$\lambda = \exp\left(1 - \frac{G_{max}}{G_{max} + 1 - G}\right)$$

Fórmula 17: Función de adaptación del parámetro λ en cada generación.

$$F = F_0 \cdot 2^{\lambda}$$

Fórmula 18: Adaptación del ratio de mutación

El objetivo de los parámetros de auto-adaptación es el de variar (adaptar) el valor de ciertos parámetros clave en el algoritmo evolutivo, como en este caso es F, conforme este avanza. Esto se hace con la premisa de que, cuantas más generaciones se hayan creado en la evolución, más cerca estaremos de la solución óptima y la búsqueda debe realizarse en un área más limitada.

Una vez obtenido el vector donante, se debe mapear a un espacio binario a través de una función sigmoidea (Fórmula 19) .

$$sig(v) = \frac{1}{1 + \exp(-v_j)}$$

Fórmula 19: Función sigmoidal de mapeo del vector donante a un espacio binario (0, 1)

Nuevamente, como condición de parada se establecen un número máximo de iteraciones.

Finalmente, este algoritmo evolutivo nos devolverá la mejor selección de instancias sintéticas, representadas en forma de 1 y 0 en el vector cromosoma obtenido como solución, para entrenar un clasificador débil.

Integrando el re-muestreo guiado por el método evolutivo diferencial de las instancias de la clase minoritaria dentro del proceso Ada-Boost, queda diseñado el algoritmo DERSBoost (Algoritmo 5):

Algoritmo: DERS-Boost

Entrada:

Conjunto de entrenamiento $S = \{(x_i, y_i)\}$, $i = \{1, 2, \ldots, N\}$

T: número de iteraciones
I: clasificador base
H: número de procesos de clustering a llevar a cabo para el sub-muestreo

Centers(h) = DE_CLUSTERING(S), $\forall h \in \{1, \ldots, H'\}$ — **Realizar H procesos de clustering y obtener en cada caso los centros**

C(i) = — **Calcular estabilidad de clustering**

S_{US} = UNDERSAMPLE(S,C) — **Sub-muestreo de los ejemplos no fronterizos**

$D_1(i) = 1/n \quad \forall i \in \{1, \ldots, n\} \quad n = n^- + ns^+$ — Asignar pesos de manera uniforme

for i = 1 .. T do

S^+ = *SMOTE(S_{US})* — **Generar instancias sintéticas de la clase minoritaria**

$S^{+'}$ = *DE_resampling(S^+)* — **Seleccionar el mejor subconjunto de instancias sintéticas con las que entrenar el clasificador débil**

$D\_syn_t(i) = 1/n \quad \forall i \in \{1, \ldots, ns^{+'}\}$ — **Asignar pesos para las instancias sintéticas seleccionadas**

$D^+_t = D_t \cup D^+_t$

$h_t = I(D^+_t, S^{+'})$ — Entrenar clasificador débil con los pesos

$E_t = \sum_{i, y_i \neq h_t(x_i)} D_t(i)$ — Calcular el error como la suma de los pesos de los ejemplos mal clasificados. Considerar sólo los pesos de los ejemplos del conjunto de entrenamiento original.

$\alpha_t = \frac{1}{2} \ln\left(\frac{1-E_t}{E_t}\right)$ — Calcular el peso del clasificador débil

$D_{t+1}(i) = D_t(i) \cdot e^{(-\alpha_t \cdot h_t(x_i) \cdot y_i)} \quad \forall i \in \{1, \ldots, N\}$ — Actualizar pesos

$D_{t+1}(i) = \frac{D_{t+1}(i)}{\sum_{i=1}^{N} D_{t+1}(i)}$ — Normalizar los pesos

endfor

$H_{final} = sign(f(x)) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$ — Clasificador final compuesto por múltiples clasificadores débiles

Salida: H_{final}

Algoritmo 5: DERS-Boost

4. IMPLEMENTACIÓN

En este apartado se describen los aspectos más relevantes del código implementado.

4.1 Evaluación de modelos

Tal y como se expuso en el apartado 2.1 el método escogido para realizar la evaluación de modelos y obtener resultados que reflejen la capacidad de generalización de los clasificadores es el denominado *5-fold-cross-validation.* También se mencionaba que el particionamiento del conjunto de datos original que se lleva a cabo en este método debía hacerse mediante la técnica de estratificación, que mantiene en cada partición generada la distribución de las clases del dataset original.

La librería de *scikit-learn* para Python, provee de una función que nos permitirá realizar la validación cruzada con estratificación. Se trata del método sklearn.model_selection.StratifiedKFold, que nos permite generar los subconjuntos de entrenamiento y test manteniendo prácticamente el mismo IR que el conjunto de datos original.

Adicionalmente, de cara a obtener los mismo resultados durante la experimentación, se establece la misma semilla al comienzo de cada entrenamiento para fijar la aleatoriedad. Además, para reforzar el análisis estadístico, se llevan a cabo tres ejecuciones para distintas semillas con el fin de extraer la media del rendimiento entre todas y consolidar los resultados. El código fuente 1 muestra el esquema de dicha evaluación.

```
gmean = []
f1 = []
elapsed_times = []

for _,seed_value in enumerate([12,123,1234]):
    np.random.seed(seed_value)

    # Llamada al constructor del clasificador
    dtc = DecisionTreeClassifier()

    kf = StratifiedKFold(n_splits=kfold)

    start_time = time()
    for train_index, test_index in kf.split(X,y):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        clf = train(X_train, y_train, method_name, dtc, T, under_prop,\
              over_prop)

        partial_gmean, partial_f1 = evaluate_clf(clf, X_test, y_test)
        gmean.append(partial_gmean)
        f1.append(partial_f1)

    elapsed_times.append(time() - start_time)
rend_gmean = stats.mean(gmean)
rend_f1 = stats.mean(f1)

print("Rendimiento del clasificador {}: Gmean = {},\
  F1_score = {}".format(method_name,rend_gmean,rend_f1))
```

Código_fuente 1: Extracto de código del bucle que implementa el 5-fold-cross-validation.
En cada iteración del bucle interno se realiza una nueva partición de train y test: (X_train, y_train) y (X_test, y_test). A continuación se entrena un clasificador con el conjunto (X_train, y_train) mediante la función 'train'. Después el clasificador obtenido es evaluado a través del conjunto(X_test, y_test) y se registra el rendimiento obtenido en esa iteración.
El bucle externo realiza la validación cruzada para la tres distintas semillas. Cuando éste finaliza se calcula la media de los rendimientos obtenidos en cada una de las iteraciones en total (5x3=15).

4.2 Configuración general

Se establecen una serie de parámetros y aspectos comunes a todos los métodos implementados, para llevar a cabo una comparativa lo más coherente posible y en las mismas condiciones para todos los modelos.

a) **Clasificador base:** se elige un **árbol de decisión C4.5** como clasificador base para entrenar todos y cada un de modelos que se implementan y estudian en el trabajo.
La clase sklearn.tree.DecisionTreeClassifier de la librería *scikit-learn*, instancia este tipo de clasificadores.

b) **Número de clasificadores base para boosting: T=10** (es una número de clasificadores base muy común y utilizada para entrenar clasificadores fuertes con ensemble learning)

c) **Métricas de rendimiento:** el estudio del rendimiento se basará principalmente en los resultados obtenidos a través de la métrica F1-SCORE, implementada a través de la librería scikit-learn con el método sklearn.metrics.f1_score. Esta medida pone el foco en los aciertos respecto a la clase minoritaria, que como se ha explicado en el apartado 2.1.1, suele ser la más importante a la hora de clasificar correctamente.

   Por otro lado, en el apéndice B de esta memoria se presentarán los resultados del rendimiento respecto a la media geométrica entre el Recall y el TNR, implementada a través de la librería imbalanced-learn con el método imblearn.metrics.geometric_mean_score. En este caso, esta medida contempla el balance entre los aciertos para ambas clases.

d) **k_neighbors = 5.** Es la cantidad de vecinos que se tienen en cuenta a la hora de realizar la interpolación para crear ejemplos sintéticos con SMOTE. Siempre que SMOTE sea invocado se le pasará este valor.

e) **Parámetros en DERS-Boost:**

   · **H = 4.** Número de clusterings que se llevarán a cabo para calcular después la 'estabilidad de clustering' de cada ejemplo de la clase mayoritaria del conjunto de entrenamiento y realizar el sub-muestreo correspondiente.

   · **alpha = 0,8.** Umbral para la estabilidad de clustering que determina si un ejemplo es fronterizo o redundante.

   · Parámetros propios de los algoritmos evolutivos diferenciales:

      · **NGEN = 50**. Número máximo de iteraciones para la evolución.

      · **POP_SIZE = 10**. Tamaño de población.

      · **CR = 0,6**. Ratio de cruce para determinar la probabilidad de que un gen del cromosoma hijo sea heredado del padre o del vector donante.

      · **F = 0,5.** Ratio de mutación para la construcción del vector donante.

4.3 Implementación de los clasificadores

Ada-Boost

La implementación de Ada-Boost (código_fuente 2) se realiza a través de la librería *scikit-learn*, la cual provee la clase sklearn.ensemble.AdaBoostClassifier a la que tendremos que facilitar una serie de parámetros de configuración para crear el modelo deseado.

```
from sklearn.ensemble import AdaBoostClassifier
clf = AdaBoostClassifier(base_estimator = dtc, n_estimators=T)
clf.fit(X, y)
```

Código_fuente 2: Instanciando un clasificador Ada-Boost. 'dtc' es un árbol de clasificación C4.5 que ha tenido que ser instanciado previamente y T es el número de clasificadores base a entrenar.

RUS-Boost

La librería *imbalanced-learn* de Python (https://imbalanced-learn.org/stable/index.html), específica para manejar problemas y desarrollar código para aprendizaje automático con conjuntos de datos no balanceados, ofrece una clase para instanciar este tipo de clasificador: imblearn.ensemble.RUSBoostClassifier. La llamada a generar un objeto de dicha clase se muestra en el código_fuente 3.

```
clf = RUSBoostClassifier(base_estimator=dtc, n_estimators=T,\
      sampling_strategy='majority')
```

Código_fuente 3: Llamada al método que instancia el clasificador RUS-Boost

SMOTE-Boost, SMOTE-Tomek-Boost y SMOTE-ENN-Boost

Una implementación del algoritmo SMOTE-Boost se recogió de un repositorio en la plataforma *github* https://github.com/dialnd/imbalanced-algorithms/blob/master/smote.py y fue utilizada como base para hacer ciertas modificaciones e implementar los tres algoritmos de boosting integrando cada uno su técnica particular de re-muestreo.

El código fuente presenta dos clases: *SMOTEBoost* (código_fuente 5), que hace referencia al clasificador y hereda de la clase *AdaBoostClassifier* para poder implementar el entrenamiento con boosting; y *SMOTE* (código_fuente 4), que es una implementación propia, customizada, del algoritmo SMOTE. La clase *SMOTEBoost* modifica la función *fit* de la clase *AdaBoostClassifier* de *scikit-learn* introduciendo el re-muestreo previo al entrenamiento del clasificador implementándolo con la segunda clase mencionada *SMOTE*. En el apéndice A de esta memoria se puede encontrar la referencia al código fuente de ambas clases.

Para implementar los tres algoritmos (*SMOTE-Boost*, *SMOTE-Tomek-Boost* y *SMOTE-ENN-Boost*) se utiliza la clase *SMOTEBoost* como base y se modifica para que, en cada caso, se lleve a cabo la técnica de re-muestreo escogida.

En el caso de *SMOTE-Boost*, se utilizará la clase imblearn.over_sampling.SMOTE de la librería *imbalanced-learn* (en lugar de la clase *SMOTE* propuesta en el repositorio del que se ha obtenido el código) para aplicar el oversampling. Tanto para *SMOTE-Tomek-Boost* como para *SMOTE-ENN-Boost* se hará uso de la librería *imblearn.combine* que implementa ambos métodos: imblearn.combine.SMOTETomek e imblearn.combine.SMOTEENN.

Se crea así la clase ***OversampleBoost*** (código_fuente 6) que dará la opción de instanciar y entrenar cualquiera de estos tres algoritmos en función del nombre que se le pase como argumento al parámetro *oversampling_algorithm* al crear un objeto de la clase. El código para esta clase puede encontrarse en el apéndice A de esta memoria.

OversampleBoost también se encarga de editar la función *fit* de entrenamiento para llevar a cabo el re-muestreo correspondiente. Dentro de cada boosting, previo al entrenamiento del clasificador base, se hace una llamada a la función *fit_resample* del algoritmo de oversampling que se ha instanciado en la definición de la clase *OversampleBoost*. La referencia a este código fuente también se encuentra en el apéndice A de la memoria.

DERS-Boost

Para programar este clasificador se creará la clase ***DERSBoost*** (código_fuente 7), que al igual que la clase *OversampleBoost*, extiende a *AdaBoostClassifier* modificando su función *fit*. En este caso las modificaciones son las siguientes:

Por un lado, se realiza el sub-muestreo de ejemplos de la clase mayoritaria previo a iniciar el bucle de entrenamientos. La clase ***DECLUndersampling*** (código_fuente 8) es la encargada de implementar el algoritmo de sub-muestreo. Debemos indicarle el número de procesos de clustering que se desea entrenar (*H*), el umbral que delimitará los ejemplos frontera de los no fronterizos (*alpha*), así como los parámetros que configuran el algoritmo evolutivo diferencial que guiará los clusterings (*CR*, *F*, *POP_SIZE* y *NGEN*).

Con estos parámetros, se instancia un objeto de la clase ***DEClustering*** (código_fuente 9), que es la encargada de llevar a cabo el algoritmo evolutivo diferencial para inferir los clusters del conjunto de datos original. El objeto de la clase *DECLUndersampling* realiza esta llamada H veces, para después pasar a calcular la estabilidad de clustering de cada ejemplo de la clase mayoritaria, categorizarlos como redundantes o fronterizos, y devolver el conjunto de datos sub-muestreado.

Todo este procedimiento se lleva a cabo a través de la función *undersample* de la clase *DECLUndersampling*.

Por otro lado, dentro de cada boosting, se llevará a cabo el algoritmo SMOTE con la clase proporcionada por la librería *imbalanced-learn* (*imblearn.oversampling.SMOTE*), y seguido, antes de realizar el entrenamiento del clasificador débil se realizará una limpieza guiada por un algoritmo evolutivo diferencial para seleccionar el mejor subconjunto de ejemplos sintéticos creados y con los que entrenar el modelo.
En este caso, el algoritmo evolutivo diferencial es implementado a través de la clase ***DESMOTE*** ***(***código_fuente 10).
A través de la función *fit*, se lleva a cabo el método evolutivo, que guardará en la variable *best_ind* el cromosoma con mejor resultado. En este caso, es un vector compuesto de los dígitos 1 y 0 que hace referencia a los ejemplos sintéticos seleccionados y no seleccionados para entrenar el modelo.
Así, se van añadiendo en *DERSBoost* a la variable *selected_syn* cada uno de los ejemplos sintéticos seleccionados con el la estrategia DE del conjunto total de sintéticos generados con SMOTE.
El último paso sería unificar el conjunto de datos original sub-muestreado con los sintéticos seleccionados (*selected_syn*), para posteriormente entrenar el clasificador débil.

En el apéndice de esta memoria se puede encontrar el código fuente de todas las clases que contribuyen a la implementación del clasificador DERS-Boost: *DERS-Boost*, *DECLUndersampling*, *DECLustering* y *DESMOTE*.

5. EXPERIMENTACIÓN Y ANÁLISIS DE RESULTADOS

En este apartado se llevaron a cabo una serie de experimentos de comparación de todos los modelos previamente descritos, con el principal objetivo de analizar los resultados y contrastar el funcionamiento de DERS-Boost frente al resto.

5.1 Datasets

Las pruebas se realizaron entrenando los diversos modelos sobre un conjunto de 14 datasets referentes a distintos campos de aplicación. Para sacar conclusiones sobre el rendimiento de los clasificadores, una vez entrenado cada modelo con cada dataset y evaluado según la métrica f1-score, se calcula la media del resultado de esta medida para cada clasificador y así se obtiene una información genérica y estadística del rendimiento de cada modelo a la hora de afrontar problemas no balanceados.

En este análisis nos centraremos en estudiar el rendimiento de los clasificadores de cara a predecir correctamente y con mayor precisión la clase positiva o minoritaria, ya que es esta la más difícil de clasificar en los problemas con conjuntos de datos no balanceados. De ahí que la métrica de evaluación escogida sea la f1-score. No obstante, en el apéndice B de esta memoria se exponen los resultados de estos mismos experimentos para la media geométrica entre el Recall y el TNR, poniendo el foco en la precisión del clasificador a la hora de lograr un balance en la correcta predicción de ambas clases.

A continuación se presenta la lista de los 14 datasets utilizados para este trabajo (Tabla 2). Todos ellos muestran un claro desbalanceo en la distribución de las clases, que viene traducido por un alto IR (*imbalanced ratio*). Los datasets se despliegan de menor a mayor IR en la lista y en el conjunto de tablas de resultados que se mostrarán a continuación en el apartado de experimentos.

Los datasets han sido recogidos de dos fuentes:

1) Cinco de ellos (‘*ecoli*’, ‘*spectrometer*’, ‘*libras_move*’, ‘*arrythmia*’ y ‘*oil*’) son obtenidos a través de la librería *imbalanced-learn*, que ofrece el método imblearn.datasets.fetch_datasets para obtener dichos conjuntos de datos.
2) El resto son extraídos del repositorio KEEL-dataset (Knowledge Extraction Evolutionary Learning). Éste es un repositorio que aporta diversos conjuntos de datos para emplearlos en el estudio y experimentación de distintos algoritmos de aprendizaje automático. En nuestro

caso, extraeremos un conjunto de datasets caracterizados por una distribución no homogénea de las clases (https://sci2s.ugr.es/keel/imbalanced.php).

ID	Nombre	N.º instancias	N.º atributos	N.º instancias clase mayoritaria	N.º instancias clase minoritaria	IR
eco	ecoli	336	7	301	35	8.60
spec	spectrometer	531	93	486	45	10.80
led	led7digit-0-2-4-5-6-7-8-9_vs_1	443	7	406	37	10.97
glass	glass2	214	9	197	17	11.59
clev	cleveland-0_vs_4	177	13	160	13	12.62
lib	libras_move	360	90	336	24	14.00
ye	yeast-1_vs_7	459	7	429	30	14.30
page	page-blocks-1-3_vs_4	472	10	444	28	15.86
der	dermatology-6	358	34	338	20	16.90
arr	arrhythmia	452	278	427	25	17.08
oil	oil	937	49	896	41	21.85
shut	shuttle-6_vs_2-3	230	9	220	10	22.00
pok	poker-9_vs_7	244	10	236	8	29.50
win	winequality-red-3_vs_5	691	11	681	10	68.10

Tabla 2: Listado de datasets utilizados para entrenar los modelos en el apartado de experimentación.

5.2 Resultados de la comparación de modelos

A continuación se exponen los resultados obtenidos tras el entrenamiento y evaluación mediante la técnica *5-fold-cross-validation* de cada clasificador para cada dataset.

Se establece la misma semilla al comienzo de cada entrenamiento para fijar la aleatoriedad. Además, para reforzar el análisis estadístico, los resultados que aquí se muestran constituyen la media entre las ejecuciones para tres semillas diferentes, tal y como se ha explicado en el apartado 3 de implementación.

La tabla 3 muestra los resultados del rendimiento obtenido para DERS-Boost y una selección de clasificadores pertenecientes a la bibliografía y al estado del arte para problemas con conjuntos de datos no balanceados. Estos son Ada-Boost, RUS-Boost, SMOTE-Boost, SMOTE-Tomek-Boost y SMOTE-ENN-Boost, previamente descritos en el apartado 2.1.2. Los resultados hacen referencia al rendimiento obtenido para la métrica f1-score. Se añade una última fila en la que se calcula la media para un clasificador de entre todos los datasets con los que se ha entrenado el modelo.

ID	ADA-BOOST	RUS-BOOST	SMOTE-BOOST	SMOTE-TOMEK-BOOST	SMOTE-ENN-BOOST	DERS-BOOST
eco	57.97	55.27	52.99	53.54	68.36	50.39
spec	76.63	68.13	65.44	65.89	66.60	64.79
led	68.51	67.82	60.57	60.57	64.90	60.84
glass	20.03	31.25	32.43	30.39	30.66	36.84
clev	39.25	46.07	74.14	75.92	65.52	67.30
lib	18.66	58.78	63.66	63.66	59.41	55.58
ye	29.65	29.68	28.77	30.10	29.71	30.37
page	68.74	82.15	77.24	77.24	84.15	78.30
der	95.85	84.28	95.55	95.55	86.66	90.09
arr	60.57	67.84	63.63	63.63	69.86	62.20
oil	29.64	21.93	20.53	17.24	16.30	20.33
shut	100.00	97.77	100.00	100.00	100.00	100.00
pok	13.33	19.07	25.33	25.33	26.44	10.66
win	8.25	9.61	6.55	3.88	2.00	0.00
MEDIA	**49.08**	**52.83**	**54.77**	**54.50**	**55.04**	**51.98**

Tabla 3: Rendimiento sobre la métrica F1-score obtenido para cada uno de los datasets con los clasificadores Ada-Boost, RUS-Boost, SMOTE-Boost, SMOTE-Tomek-Boost, SMOTEENN-Boost y DERS-Boost.

Tal y como puede observarse, el clasificador Ada-Boost (49.08%), al no implementar ninguna técnica de re-muestreo, se queda por debajo del resto de modelos respecto al rendimiento. El clasificador **SMOTE-ENN-Boost (55.04%)** resulta ser el más eficaz a la hora de clasificar la clase positiva. Le siguen **SMOTE-Boost (54.77%)** y **SMOTE-Tomek-Boost (54.5%)**. En cuanto a los resultados logrados con **DERS-Boost (51.98%)**, a pesar de la mejora frente a Ada-Boost a causa de implementar re-muestreo, el modelo no consigue alcanzar la precisión lograda con el resto de los clasificadores que aplican también este tipo de técnicas.

Reforzaremos la fiabilidad de estos resultados llevando a cabo el **test de wilcoxon** para los distintos pares de clasificadores. Este test lleva a cabo un análisis estadístico cuyo objetivo es validar la hipótesis de que, para un par de muestras de datos, estos siguen la misma distribución y no existen diferencias en su rango medio. En nuestro caso, esta hipótesis se traduce en comprobar que el rendimiento obtenido para cada uno de los datasets y entre dos clasificadores distintos, sigue el mismo comportamiento, y por lo tanto, ambos clasificadores son iguales de competentes.

El test de wilcoxon para las medias de entre dos clasificadores, tiene como resultado el ***p-valor***, un indicador cuya interpretación viene a ser la siguiente: si dicho parámetro es menor que un valor fijado $\alpha \in (0,1)$ (generalmente α=0,05), podemos afirmar con un grado de fiabilidad de $1-\alpha$ que la hipótesis anterior de que ambos clasificadores son equivalentes en cuanto al rendimiento, quedaría descartada.

En este caso, se comparan todos los pares de clasificadores mostrados en la tabla 3 y los resultados para el test de wilcoxon quedan expuestos en la tabla 4.

	RUS-Boost	**SMOTE-Boost**	**SMOTE-Tomek-Boost**	**SMOTE-ENN-Boost**	**DERS-Boost**
Ada-Boost	0,5416	0,4215	0,5067	0,1519	0,8613
RUS-Boost		1,0	0,9515	0,3575	0,3258
SMOTE-Boost			0,4989	0,9721	**0,0747**
SMOTE-Tomek-Boost				0,8067	**0.1000**
SMOTE-ENN-Boost					**0.1519**

Tabla 4: Resultados del p-valor del test de Wilcoxon para las medias del f1-score, para distintos pares de clasificadores.

Respecto a los resultados del test que comparan DERS-Boost frente al resto de clasificadores, puede afirmarse con un grado de confianza mayor la diferencia en el rendimiento de éste modelo frente a SMOTE-Boost (p-valor = 0,0747), SMOTE-Tomek-Boost (p-valor = 0,1) y SMOTE-ENN-Boost (p-valor = 0,1519), siendo estos los que mejor precisión obtienen.

En cuanto al resto de resultados para el p-valor del test de wilcoxon, no revelan tal grado de confianza ya que se obtiene un p-valor más elevado para la comparación de cada par de clasificadores. Tal vez con un análisis sobre un conjunto de datasets más amplio ayudaría a esclarecer y acentuar estas diferencias. Sin embargo, el alto coste en tiempo de ejecución del clasificador DERS-Boost limita tanto el número de datasets como el tamaño asequible de estos a la hora de llevar a cabo la experimentación, lo cual ya nos coloca ante una de las mayores desventajas de este nuevo modelo.

En la tabla 5 se muestra el tiempo medio de ejecución (en segundos) que incluye los procesos de entrenamiento y evaluación de cada clasificador:

	Ada-Boost	**RUS-Boost**	**SMOTE-Boost**	**SMOTE-Tomek-Boost**	**SMOTE-ENN-Boost**	**DERS-Boost**
Tiempo	0.0531	0.1776	0.0967	0.2034	0.1871	368.20

Tabla 5: Tiempo medio de ejecución de entrenamiento y evaluación con 5fold-cross-validation entre todos los datasets para los clasificadores Ada-Boost, RUS-Boost, SMOTE-Boost, SMOTE-Tomek-Boost, SMOTEENN-Boost y DERS-Boost.

El tiempo de ejecución para Ada-Boost, RUS-Boost, SMOTE-Boost, SMOTE-Tomek-Boost y SMOTE-ENN-Boost es asumible. Entre éstos, aquellos métodos que emplean técnicas de re-muestreo y que presentan un mejor rendimiento en cuanto a la precisión del clasificador, a pesar de ser algo más lentos que Ada-Boost, no resultan demasiado costosos en tiempo de ejecución.

No ocurre lo mismo con DERS-Boost, donde resulta evidente la complejidad del método frente al resto de clasificadores. La lentitud en este caso viene marcada por el empleo de los dos algoritmos evolutivos: el primero, es utilizado para generar un clustering a partir del conjunto de datos de entrenamiento original. Además, este proceso se repite H=4 veces multiplicando el coste en tiempo. En el segundo algoritmo evolutivo, aquel que guía la selección de las mejores instancias sintéticas, hay que tener en cuenta que la evaluación del fitness de cada cromosoma implica aprender un nuevo árbol de decisión. Es decir, si deseamos entrenar un clasificador fuerte a través de un ensemble de T clasificadores base, empleando en cada boosting un algoritmo evolutivo para seleccionar las

mejores instancias caracterizado por un tamaño de población P y un número máximo de generaciones G_{max}, el total de modelos a aprender, en este caso árboles de decisión, es de $T*P*G_{max}$, frente a los T entrenamientos que implican los métodos anteriores.

Con ésto, resulta evidente que DERS-Boost presenta una gran desventaja frente al resto de clasificadores testeados. Si además tenemos en cuenta que no es capaz de mejorar el rendimiento, es un método que quedaría descartado para afrontar problemas de clasificación no balanceados.

5.3 Análisis fragmentado de las distintas estrategias de re-muestreo

Visto que el algoritmo DERS-Boost no aporta una mejora relevante del rendimiento frente al resto de métodos conocidos y que no resulta eficaz debido a su largo tiempo de ejecución, en este apartado se llevan a cabo diversos experimentos que tratan de estudiar el efecto de las distintas estrategias que implementa DERS-Boost por separado, y concluir si puede haber una manera o combinación de construir un clasificador más eficiente que integre alguna de las técnicas propuestas en este nuevo modelo.

5.3.1 Comparación de distintas técnicas de undersampling para entrenar un clasificador base

Para empezar, deseamos comprobar si la técnica de sub-muestreo de los ejemplos de la clase mayoritaria propuesta en DERS-Boost es verdaderamente útil y aporta una mejora a la hora de entrenar un clasificador base. Nos centramos en el entrenamiento de un único árbol de decisión y no en una combinación de ensemble para agilizar el proceso ya que aquellas estrategias que integran técnicas evolutivas son muy costosas en tiempo de ejecución.

DERS-Boost propone descartar parte de los ejemplos de la clase mayoritaria que no sean considerados como ejemplos cercanos a la frontera de decisión del problema (ejemplos no fronterizos o redundantes). Para categorizar de tal manera las instancias, se calcula su *estabilidad de clustering*, donde las instancias con alta estabilidad serán las catalogadas como redundantes. Para este cómputo, se llevan a cabo varios procesos de clustering guiados por un algoritmo evolutivo diferencial y a partir del conjunto original de entrenamiento. Después, se clasifica cada ejemplo de la clase mayoritaria para cada uno de los clusterings inferidos y se calcula así su *estabilidad de clustering*, según si se ha clasificado el ejemplo como parte de la clase mayoritaria más o menos veces.

A simple vista parece una técnica guiada por una heurística bastante interesante pero que debemos comprobar si realmente presenta mejores resultados que otro tipo de técnicas de *undersampling* más simples y que ya han demostrado ser funcionales.

Para empezar, y motivado por el alto coste en tiempo de ejecución que implica la evolución diferencial que guía los clusterings, nos preguntamos si la estrategia evolutiva aporta alguna mejora frente al algoritmo 'K-means'. 'K-means' es uno de los algoritmos más comunes a la hora de realizar particiones de conjuntos de datos para obtener los centros más representativos de cada clase mediante aprendizaje no supervisado. En cierto modo, la evolución diferencial sigue una heurística muy parecida a la utilizada en 'K-means', ya que la función de *fitness* consiste en minimizar la distancia entre los ejemplos clasificados para una clase con el centro de dicha clase, mismo objetivo que persigue el algoritmo de aprendizaje no supervisado.
Para llevar a cabo esta comparación, se realiza un experimento que consiste en entrenar dos clasificadores base (dejando de lado el método de boosting utilizado hasta ahora, para ahorrar tiempo de ejecución) sub-muestreando previamente la clase mayoritaria con cada uno de los métodos. El primero, al que denominaremos *DE_under+C4.5* llevará a cabo la técnica de *undersampling* propuesta en DERS-Boost y descrita en el apartado 3.1, previo al entrenamiento del árbol de decisión. El segundo, lo designaremos como *KMEANS_Under+C4.5*, y realizará un sub-muestreo idéntico en cuanto a dejar fuera los ejemplos redundantes de la clase negativa, pero en este caso, los H procesos de clustering se llevarán a cabo mediante el algoritmo K-Means.

Tras el entrenamiento y evaluación siguiendo la misma metodología que en la primera experimentación, obtenemos la tabla 6 con los resultados al comparar estas dos variantes del primer bloque de DERS-Boost:

ID	DE_Under+C4.5 (%)	KMEANS_Under+C4.5 (%)
eco	57.21	59.82
spec	72.00	76.46
led	69.57	69.63
glass	30.25	21.01
clev	46.00	43.17
lib	34.34	18.22
ye	33.41	28.98
page	74.60	68.74
der	97.77	92.67
arr	59.98	55.97
oil	24.96	29.32
shut	100.00	100.00
pok	11.11	13.33
win	5.23	7.14
MEDIA	**51.15**	**48.89**

Tabla 6: Rendimiento para la métrica F1-score obtenido para cada uno de los datasets con los clasificadores DE_Under+C4.5 y KMEANS_Under+C4.5.

El clasificador **DE_Under+C4.5 (51.15%)** obtiene mejor rendimiento. Teniendo en cuenta que la única diferencia entre ambos modelos es la estrategia utilizada a la hora de inferir los clusters que luego determinarán los ejemplos redundantes a eliminar del conjunto de datos original, podemos llegar a la conclusión de que la técnica DE guía mejor la clusterización que el propio algoritmo K-means.

A continuación se llevó a cabo el test de *wilcoxon* entre los dos conjuntos de medias para cada clasificador. Para comprobar la hipótesis de que ambos clasificadores siguen el mismo comportamiento ejecutaríamos la siguiente línea de código:

```
wilcoxon(f1_medias_DEunder,f1_medias_KMEANSunder)
>>> WilcoxonResult(statistic=27.0, pvalue=0.1960509634811064)
```

Con un resultado del p-valor igual a 0.1960, quedaría descartada dicha hipótesis con una fiabilidad del 80% en favor de que ambos clasificadores no son igual de competentes.

No obstante, no podemos dejar de tener en cuenta el tiempo medio de ejecución que implica cada algoritmo y que en este caso es:

	DE_Under	**KMEANS_Under**
Tiempo	310.77	6.84

Tabla 7: Tiempo medio de ejecución de entrenamiento y evaluación con 5fold-cross-validation entre todos los datasets para los clasificadores DE_Under+C4.5 y KMEANS_Under+C4.5.

En este caso debería plantearse si es verdaderamente útil aplicar la evolución diferencial frente al algoritmo K-means, ya que el tiempo de ejecución de *DE_Under* multiplica nada menos que 45 veces al de *KMEANS_Under*. Debe reflexionarse si la mejora del rendimiento es suficientemente buena como para invertir tanto tiempo en aprender el clasificador.

Una vez estudiado que la evolución diferencial aporta ventajas dentro la estrategia de undersampling propuesta en DERS-Boost, parece intersante comparar si esta tećnica, basada en la estabilidad de clustering para catalogar los ejemplos como fronterizos o redundantes, es realmente más efectiva frente a otros métodos ya conocidos como RUS, Tomek-Links o ENN.

Para llevar a cabo esta comparación se construyeron 3 modelos que, del mismo modo que el experimento anterior, integran cada una de las técnicas de sub-muestreo recientemente mencionadas para después pasar a entrenar un único clasificador base (árbol de decisión C4.5). Los datos respectivos a los modelos *DE_Under* y *KMEANS_Under* siguen siendo los mismos que en la tabla anterior. Los resultados de esta comparativa los mostramos en la tabla 8.

ID	DE_Under+C4.5	KMEANS_Under + C4.5	RUS+C4.5	Tomek+C4.5	ENN+C4.5
eco	57.21	59.82	64.15	52.54	65.67
spec	72.00	76.46	46.83	77.26	82.67
led	69.57	69.63	55.44	70.20	64.25
glass	30.25	21.01	24.34	15.92	20.59
clev	46.00	43.17	32.55	45.37	32.89
lib	34.34	18.22	27.88	20.79	20.54
ye	33.41	28.98	19.83	31.27	24.33
page	74.60	68.74	76.48	66.52	61.15
der	97.77	92.67	84.56	89.77	80.88
arr	59.98	55.97	52.00	52.03	64.21
oil	24.96	29.32	23.41	31.16	25.73
shut	100.00	100.00	100.00	100.00	100.00
pok	11.11	13.33	11.90	13.33	23.33
win	5.23	7.14	3.78	5.47	5.05
MEDIA	**51.15**	**48.89**	**44.51**	**47.97**	**47.95**

Tabla 8: Rendimiento para la métrica f1-score obtenido para cada uno de los datasets con los clasificadores DE_Under+C4.5, KMEANS_Under+C4.5, RUS+C4.5, Tomek+C4.5 y ENN+C4.5.

Los datos observados revelan que el modelo **DE_Under+C4.5 (51.15%)**, aquel que aplica la técnica de re-muestreo propuesta en DERS-Boost y guiada por la evolución diferencial, es el mejor en este caso para clasificar la clase positiva. Además, el clasificador KMEANS_Under+C4.5 (48.89%), que lleva a cabo la misma estrategia propuesta en DERS-Boost de sub-muestrear los ejemplos de la clase mayoritaria más redundantes, pero realizando la clusterización previa con otro algoritmo, obtiene el segundo mejor resultado. Esto quiere decir que la lógica llevada a cabo en la técnica de undersampling de DERS-Boost sí puede llegar a ser más útil que los métodos de sub-muestreo ya conocidos, de cara a entrenar un único clasificador base.

Sin embargo, no podemos obviar el alto coste en tiempo de ejecución que aleja al clasificador DE_Under+C4.5 de ser realmente eficiente frente al resto (Tabla 9).

	DE_Under	KMEANS_Under	RUS	Tomek	ENN
Tiempo	310.77	6.84	0.0257	0.1067	0.1017

Tabla 9: Tiempo medio de ejecución de entrenamiento y evaluación con 5fold-cross-validation entre todos los datasets para cada uno de los clasificadores que aplican una técnica de sub-muestreo de los ejemplos de la clase mayoritaria.

5.3.2 Comparación de distintas técnicas de *oversampling* para entrenar un clasificador base

En este segundo apartado se quiere estudiar el impacto de las técnicas de sobre-muestreo de los ejemplos de la clase minoritaria.

Recordamos que DERS-Boost lleva a cabo en cada boosting un sobre-muestreo de estos ejemplos con el algoritmo SMOTE, para después seleccionar, mediante un algoritmo evolutivo diferencial, el subconjunto óptimo de ejemplos sintéticos a añadir para entrenar el clasificador base de la iteración correspondiente. En las siguientes líneas nos referiremos a esta técnica de re-muestreo como *SMOTE+DE*.

La pregunta a la que queremos responder en esta ocasión es si esta estrategia propuesta en DERS-Boost resulta en un conjunto de entrenamiento que ayude a mejorar el rendimiento del clasificador base que se esté entrenando, frente a las ya conocidas *SMOTE, ROS (Random Over Sampling), SMOTE+Tomek o SMOTE+ENN.*

De manera análoga a cómo se hizo en el apartado anterior para el estudio de las técnicas de *undersampling*, a continuación se llevará cabo la construcción de varios clasificadores base aplicando cada sobre-muestreo previo con las técincas mencionadas en le párrafo posterior, y aprendiendo finalmente un árbol de decisión C4.5. Los resultados se muestran en la tabla 10.

ID	SMOTE+DE	SMOTE	ROS	SMOTE+Tomek	SMOTE+ENN
eco	53.47	53.45	53.42	55.67	68.92
spec	62.65	67.31	62.32	66.73	67.64
led	68.61	69.34	63.92	69.34	67.29
glass	30.79	34.32	33.51	27.67	28.50
clev	68.15	71.03	29.49	70.96	69.07
lib	57.41	60.42	46.79	60.42	58.63
ye	34.06	33.32	34.28	31.64	30.55
page	84.23	77.80	74.70	77.80	83.55
der	89.14	92.97	88.84	92.97	82.81
arr	87.35	63.46	61.31	63.46	67.38
oil	14.00	18.46	21.60	18.12	17.75
shut	100.00	100.00	100.00	100.00	100.00
pok	10.44	18.66	26.00	18.66	27.11
win	1.90	6.66	0.00	7.61	1.21
MEDIA	**52.30**	**54.80**	**49.73**	**54.36**	**55.03**

Tabla 10: Rendimiento para la métrica f1-score obtenido para cada uno de los datasets construyendo un clasificador base C4.5 y aplicando cada una de las técnicas: SMOTE+DE+C4.5, ROS+C4.5, SMOTE+C4.5, SMOTE+Tomek+C4.5 y *SMOTE+ENN+C4.5.*

El mejor rendimiento es obtenido con el clasificador **SMOTE+ENN+C4.5 (55,03%).** El modelo SMOTE+DE+C4.5 (52,30%), que integra la técnica evolutiva diferencial propuesta en DERS-Boost para realizar una limpieza de los ejemplos sintéticos creados previamente con SMOTE, no es capaz de superar el rendimiento del resto de clasificadores, a excepción de ROS+C4.5 (49,73%), para el cual los resultados son los más bajos de la tabla. En este sentido, podemos concluir que la limpieza de los ejemplos guiada por DE que se propone en DERS-Boost no ofrece ninguna ventaja respecto al resto de técnicas conocidas para entrenar un clasificador base. El uso de esta estrategia se queda incluso por debajo del método que tan sólo aplica un re-muestreo con SMOTE (sin ninguna limpieza posterior de los ejemplos sintéticos generados).

5.3.3 El mejor modelo

Se ha comprobado de manera independiente el efecto del uso de las estrategias evolutivas a la hora de entrenar un único árbol de decisión. Por un lado, el sub-muestreo mediante la técnica propuesta en DERS-Boost presentaba cierta mejora en el rendimiento frente a otro tipo de métodos de undersampling (RUS, Tomek y ENN). Por el contrario, se ha podido observar que el uso de DE para realizar una limpieza de los sintéticos generados con SMOTE no aporta mayor conocimiento al clasificador.

Motivado por los resultados anteriores, surge la idea de crear un modelo híbrido basado en DERS-Boost, pero que en este caso integre la técnica de re-muestreo SMOTE-ENN dentro del boosting, ya que es esta con la que mejores resultados se han obtenido analizando las estrategias de oversampling por separado. Los resultados para esta nueva propuesta híbrida, **DE_Under + SMOTE-ENN-Boost**, junto con los del clasificador que mejor rendimiento ha sido capaz de obtener (SMOTE-ENN-Boost) y junto a los de DERS-Boost, el modelo objeto de estudio de este trabajo, se exponen en la tabla 11.

ID	DE_Under + SMOTE-ENN-Boost	SMOTE-ENN-BOOST	DERS-BOOST
eco	62.37	68.36	50.39
spec	66.20	66.60	64.79
led	65.66	64.90	60.84
glass	29.45	30.66	36.84
clev	54.10	65.52	67.30
lib	62.09	59.41	55.58
ye	25.07	29.71	30.37
page	71.28	84.15	78.30
der	92.14	86.66	90.09
arr	68.65	69.86	62.20
oil	20.97	16.30	20.33
shut	100.00	100.00	100.00
pok	25.33	26.44	10.66
win	0.00	2.00	0.00
MEDIA	**53.09**	**55.04**	**51.98**

Tabla 11: Rendimiento sobre la métrica F1-score obtenido para cada uno de los datasets con los clasificadores DE_Under + SMOTE-ENN-Boost, SMOTEENN-Boost y DERS-Boost.

La tabla 12 muestra el tiempo medio en segundos para entrenar cada clasificador:

	DE_Under + SMOTE-ENN-Boost	SMOTE-ENN-Boost	DERS-Boost
Tiempo	311.35	0.1871	368.20

Tabla 12: Tiempo medio de ejecución de entrenamiento y evaluación con 5fold-cross-validation entre todos los datasets para los clasificadores DE_Under + SMOTE-ENN-Boost, SMOTEENN-Boost y DERS-Boost.

Observando los resultados del rendimiento en la tabla 11, en este caso, a pesar de la combinación de las dos estrategias que mejores resultados nos habían aportado en los anteriores apartados, el modelo SMOTE-ENN-Boost es el que mejor clasifica la clase positiva y además vuelve a ser el más rápido con diferencia, al mismo tiempo que el más simple.

6. CONCLUSIÓN Y LÍNEAS FUTURAS

El clasificador DERS-Boost es una propuesta que integra técnicas de re-muestreo guiadas por algoritmos de computación evolutiva diferencial con el objetivo de aportar una mejora en el rendimiento a la hora de afrontar problemas de clasificación no balanceados.

Las estrategias planteadas siguen una heurística y razonamiento interesante las cuales nos motivan a la realización de este estudio. El hecho de incorporar la computación evolutiva a la hora de guiar ciertas búsquedas, como son en este caso, por un lado, la clusterización del conjunto de datos original, y por otro, la búsqueda del mejor subconjunto de ejemplos sintéticos de la clase minoritaria con la que entrenar un árbol de decisión, genera curiosidad a la hora de comprobar si en efecto el uso de DE es capaz de guiar las búsquedas hacia mejores soluciones, tal y como se ha demostrado que son capaces este clase de técnicas a la hora de resolver otro tipo de problemas de optimización, y que estos resultados sirvan al mismo tiempo para guiar el aprendizaje del clasificador y obtener un mejor rendimiento frente a la predicción de clase positiva en problemas con conjuntos de datos no balanceados.

Sin embargo, a la hora de llevar DERS-Boost a su estudio y experimentación, surgen las primeras desventajas frente a otros métodos ya conocidos que tratan de poner solución al mismo problema y que han demostrado obtener cierta mejora en los resultados de rendimiento.

En primer lugar, resulta obvio estimar que el tiempo de ejecución del algoritmo que guía el aprendizaje de DERS-Boost superará con creces al de cualquier otro método ya existente. Las técnicas de computación evolutiva implican de por sí un ciclo de iteraciones considerable. Si además estos algoritmos se integran para entrenar un clasificador fuerte con un ensemble, o se llevan a cabo varias repeticiones de búsqueda a través de DE, como es el caso de realizar varios procesos de clustering al principio de DERS-Boost, nos encontramos ante un algoritmo de aprendizaje notablemente lento y costoso.

Esta característica limita el rango de datasets con los que realizar los experimentos de comparación con otros modelos, ya que para varios conjuntos de datos disponibles en donde el número de instancias es elevado, resulta inviable entrenar y evaluar el modelo siguiendo la metodología de validación propuesta. En este aspecto, DERS-Boost presenta un aparente gran inconveniente frente a otros clasificadores que tratan de resolver el problema de predicción con conjuntos de datos no balanceados.

Por otro lado, una vez llevados a cabo los experimentos que consisten en el entrenamiento y evaluación de distintos clasificadores (entre ellos DERS-Boost) para un conjunto de 14 datasets que presentan un alto IR, los resultados sobre la métrica f1-score, que ponen el foco en los aciertos obtenidos para la clase minoritaria, no revelan ninguna mejora del modelo DERS-Boost frente al resto. La insuficiente mejora en el rendimiento y la lentitud del aprendizaje nos llevan a descartar DERS-Boost como un potencial modelo de clasificación para afrontar problemas con conjuntos de datos no balanceados.

No obstante, se quiso estudiar más en profundidad el efecto de las distintas estrategias propuestas en DERS-Boost de manera independiente.
En el apartado 4.3.2 se constató que el uso de la evolución diferencial no a portaba ninguna mejora a la hora de guiar la búsqueda para el mejor subconjunto de sintéticos con los que entrenar un clasificador base. Por el contrario, en el apartado 4.3.1 sí se puedo observar un mejor rendimiento en el modelo que utilizaba DE para guiar la clusterización con la que después realizar el undersampling de los ejemplos redundantes de la clase mayoritaria, frente al modelo que utilizaba el algoritmo K-means para ese primer objetivo de crear los clusters.
En este sentido, aparece un nuevo marco de estudio en el que analizar con mayor detalle el efecto de las técnicas de DE en problemas de clustering de aprendizaje no supervisado. Debido a su coste en tiempo de ejecución, quizá no sea lo más óptimo integrar estos métodos en un completo algoritmo de aprendizaje de un clasificador que ya combine otro tipo de técnicas, como es el caso de DERS-Boost. Una posible nueva línea de estudio sería centrar y enfocar el análisis del efecto de las técnicas de DE en métodos de aprendizaje no supervisado, tal y como se ha podido intuir en este trabajo en el apartado 4.3.1 que podría obtener mejores resultados que algunos métodos ya conocidos.

En conclusión, DERS-Boost no presenta ninguna mejora en el rendimiento frente a modelos ya conocidos como SMOTE-Boost, RUS-Boost o SMOTE-ENN-Boost para afrontar problemas de clasificación no balanceados e igualmente, su mayor desventaja reside en el alto coste en tiempo de ejecución que implica el entrenamiento del clasificador, por lo que no viene a aportar ningún avance en cuanto a clasificadores que integren técnicas para hacer frente al problema de la distribución de las clases en el conjunto de datos.

BIBLIOGRAFÍA

[1] P. Gong, J. Gao, L. Wang, F. Zhang, School of Economics and Management, Beihang University, Beijing, China. **Differential evolution-guided resampling combined with Ada-Boost ensemble learning for imblanced data classification**. *ESWA-D-19-01163* manuscript draft.

[2] V. López, A. Fernandez, S. Garcia, V. Palade and F. Herrera, **An Insight into Classification with Imbalanced Data: Empirical Results and Current Trends on Using Data Intrinsic Characteristics.** Information Sciences 250 (2013) 113-141 doi: 10.1016/j.ins.2013.07.007

[3] A. Fernández, Soft Computing and Intelligent Information Systems, a University of Granada research group. **Classification with Imbalanced Datasets**. https://sci2s.ugr.es/imbalanced

[4] Sanz-Delgado, J. (2018). Sistemas inteligentes para la extracción de información.

[5] Fernández-Fernández, F.J. (2019) Computación. Introducción a los algoritmos genéticos.

[6] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, **SMOTE: Synthetic minority over-sampling technique** *(2002) Journal Of Artificial Intelligence Research, 16: pp 321-357.* doi: 10.1016/S0031-3203(96)00142-2

[7] Chawla, N. V., Lazarevic, A., Hall, L. O., & Bowyer, K. W. (2003, September). **SMOTEBoost: Improving prediction of the minority class in boosting**. In European conference on principles of data mining and knowledge discovery (pp. 107-119). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-39804-2_12

[8] C. Seiffert, T.M. Khoshgoftaar, J. Van Hulse, A. Napolitano, **RUSBoost: A Hybrid Approach to Alleviating Class Imbalance** *(2010) IEEE Transactions On Systems Man And Cybernetics Part A-Systems And Humans, 40:1 pp: 185-197.* doi: 10.1109/TSMCA.2009.2029559

[9] I. Tomek, “Two modifications of CNN,” IEEE Transactions on Systems, Man, and Cybernetics, vol. 6, pp. 769-772, 1976.

[10] D. Wilson, “Asymptotic Properties of Nearest Neighbor Rules Using Edited Data,” IEEE Transactions on Systems, Man, and Cybernetrics, vol. 2(3), pp. 408-421, 1972.

[11] G. E. A. P. A. Batista, A. L. C. Bazzan, M. C. Monard, “Balancing training data for automated annotation of keywords: A case study,” In Proceedings of the 2nd Brazilian Workshop on Bioinformatics, pp. 10-18, 2003.

[12] G. E. A. P. A. Batista, R. C. Prati, M. C. Monard, “A study of the behavior of several methods for balancing machine learning training data,” ACM Sigkdd Explorations Newsletter, vol. 6(1), pp. 20-29, 2004.

[13] Y. Freund, R. Schapire, “A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting”, 1995.

[14] Storn, R.; Price, K. (1997). **"Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces"**. *Journal of Global Optimization.* ***11*** *(4): 341–359.* doi:10.1023/A:1008202821328. S2CID 5297867.

[15] MacQueen, J. B. (1967). *Some Methods for classification and Analysis of Multivariate Observations*. Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability **1**. University of California Press. pp. 281-297. MR 0214227. Zbl 0214.46201.

[16] G. Batista, R. Prati, and M. Monard, “A study of the behavior ofseveral methods for balancing machine learning training data,”SIGKDDExplorations, vol. 6, no. 1, pp. 20–29, 2004.

APÉNDICE A: código fuente

Este apéndice recoge el código fuente y varias de las clases más relevantes a la hora de implementar los distintos clasificadores que se entrenar en el apartado de experimentación.

En primer lugar se presenta la implementación del algoritmo SMOTE-Boost (código_fuente 5) que se recogió de un repositorio en la plataforma *github* https://github.com/dialnd/imbalanced-algorithms/blob/master/smote.py y que fue utilizada como base para hacer ciertas modificaciones e implementar el resto de modelos que forman parte de este trabajo, integrando cada uno en cada caso su técnica particular de re-muestreo.

```
class SMOTE(object):

    def __init__(self, k_neighbors=5, random_state=None):
        self.k = k_neighbors
        self.random_state = random_state

    def sample(self, n_samples):

        np.random.seed(seed=self.random_state)

        S = np.zeros(shape=(n_samples, self.n_features))
        # Calculate synthetic samples.
        for i in range(n_samples):
            j = np.random.randint(0, self.X.shape[0])

            # Find the NN for each sample.
            # Exclude the sample itself.
            nn = self.neigh.kneighbors(self.X[j].reshape(1, -1),
                                       return_distance=False)[:, 1:]
            nn_index = np.random.choice(nn[0])

            dif = self.X[nn_index] - self.X[j]
            gap = np.random.random()

            S[i, :] = self.X[j, :] + gap * dif[:]

        return S

    def fit(self, X):
        self.X = X
        self.n_minority_samples, self.n_features = self.X.shape

        # Learn nearest neighbors.
        self.neigh = NearestNeighbors(n_neighbors=self.k + 1)
        self.neigh.fit(self.X)

        return self
```

Código_fuente 4: Implementación del algoritmo SMOTE en https://github.com/dialnd/imbalanced-algorithms/blob/master/smote.py. En este trabajo no se utiliza, se opta por la clase que ofrece la librería imbalanced-learn para llevar a cabo el algoritmo.

```
class SMOTEBoost(AdaBoostClassifier):

    def __init__(self,
                 n_samples=100,
                 k_neighbors=5,
                 base_estimator=None,
                 n_estimators=50,
                 learning_rate=1.,
                 algorithm='SAMME.R',
                 random_state=None):

        self.n_samples = n_samples
        self.algorithm = algorithm
        self.smote = SMOTE(k_neighbors=k_neighbors,
                           random_state=random_state)

        super(SMOTEBoost, self).__init__(
            base_estimator=base_estimator,
            n_estimators=n_estimators,
            learning_rate=learning_rate,
            random_state=random_state)

    def fit(self, X, y, sample_weight=None, minority_target=None):

        # Check that algorithm is supported.
        if self.algorithm not in ('SAMME', 'SAMME.R'):
            raise ValueError("algorithm %s is not supported" %
                             self.algorithm)

        # Check parameters.
        if self.learning_rate <= 0:
            raise ValueError("learning_rate must be greater than zero")

        if (self.base_estimator is None or
                isinstance(self.base_estimator, (BaseDecisionTree,
                                                 BaseForest))):
            DTYPE = np.float64  # from fast_dict.pxd
            dtype = DTYPE
            accept_sparse = 'csc'
        else:
            dtype = None
            accept_sparse = ['csr', 'csc']

        X, y = check_X_y(X, y, accept_sparse=accept_sparse, dtype=dtype,
                         y_numeric=is_regressor(self))

        if sample_weight is None:
            # Initialize weights to 1 / n_samples.
            sample_weight = np.empty(X.shape[0], dtype=np.float64)
            sample_weight[:] = 1. / X.shape[0]
        else:
            sample_weight = check_array(sample_weight, ensure_2d=False)
            # Normalize existing weights.
            sample_weight = sample_weight /
                            sample_weight.sum(dtype=np.float64)
            # Check that the sample weights sum is positive.
            if sample_weight.sum() <= 0:
                raise ValueError(
                    "Attempting to fit with a non-positive "
                    "weighted number of samples.")
      ...
```

```
        if minority_target is None:
            # Determine the minority class label.
            stats_c_ = Counter(y)
            maj_c_ = max(stats_c_, key=stats_c_.get)
            min_c_ = min(stats_c_, key=stats_c_.get)
            self.minority_target = min_c_
        else:
            self.minority_target = minority_target

        # Check parameters.
        self._validate_estimator()

        # Clear any previous fit results.
        self.estimators_ = []
        self.estimator_weights_ = np.zeros(self.n_estimators, dtype=np.float64)
        self.estimator_errors_ = np.ones(self.n_estimators, dtype=np.float64)

        random_state = check_random_state(self.random_state)

        for iboost in range(self.n_estimators):
            # SMOTE step.
            X_min = X[np.where(y == self.minority_target)]
            self.smote.fit(X_min)
            X_syn = self.smote.sample(self.n_samples)
            y_syn = np.full(X_syn.shape[0], fill_value=self.minority_target,
                            dtype=np.int64)

            # Normalize synthetic sample weights based on current training set.
            sample_weight_syn = np.empty(X_syn.shape[0], dtype=np.float64)
            sample_weight_syn[:] = 1. / X.shape[0]

            # Combine the original and synthetic samples.
            X = np.vstack((X, X_syn))
            y = np.append(y, y_syn)

            # Combine the weights.
            sample_weight = \
                np.append(sample_weight, sample_weight_syn).reshape(-1, 1)
            sample_weight = \
                np.squeeze(normalize(sample_weight, axis=0, norm='l1'))

            # X, y, sample_weight = shuffle(X, y, sample_weight,
            #                               random_state=random_state)

            # Boosting step.
            sample_weight, estimator_weight, estimator_error = self._boost(
                iboost,
                X, y,
                sample_weight,
                random_state)

            # Early termination.
            if sample_weight is None:
                break

            self.estimator_weights_[iboost] = estimator_weight
            self.estimator_errors_[iboost] = estimator_error

    ...
```

```
            # Stop if error is zero.
            if estimator_error == 0:
                Break
            sample_weight_sum = np.sum(sample_weight)

            # Stop if the sum of sample weights has become non-positive.
            if sample_weight_sum <= 0:
                break

            if iboost < self.n_estimators - 1:
                # Normalize.
                sample_weight /= sample_weight_sum

        return self
```

Código_fuente 5: Implementación de la clase SMOTEBoost en https://github.com/dialnd/imbalanced-algorithms/blob/master/smote.py, que modifica la función fit de AdaBoostClassifier para introducir un re-muestreo con SMOTE antes de entrenar un clasificador débil. Queda destacado en negrita dicho fragmento de código.

Los siguientes cuadros de código presentan las clases donde, a traveś de una serie de ligeras modificaciones sobre la clase *SMOTEBoost*, se implementan cada uno de los clasificadores que han sido entrenados en este trabajo y que integran otras técnicas de re-muestreo:

```
class OversampleBoost(AdaBoostClassifier):

      def __init__(self, oversampling_algorithm, over_ratio='auto',
            undersampling_algorithm=None, under_ratio='auto', k_neighbors=5,
            base_estimator=None, n_estimators=10,learning_rate=1.,
            algorithm='SAMME.R', random_state=None):

            self.smote_kneighbors = k_neighbors
            self.algorithm = algorithm

            # instanciar el algoritmo de re-muestreo indicado
            if oversampling_algorithm=='SMOTE':
                  self.smote = SMOTE(k_neighbors =
                        self.smote_kneighbors,sampling_strategy=over_ratio)
            elif oversampling_algorithm=='SMOTE-TOMEK':
                  self.smote = SMOTETomek(sampling_strategy=over_ratio)
            elif oversampling_algorithm=='SMOTE-ENN':
                  self.smote = SMOTEENN(sampling_strategy=over_ratio)

            super(OversampleBoost, self).__init__(
                  base_estimator=base_estimator,
                  n_estimators=n_estimators,
                  learning_rate=learning_rate,
                  random_state=random_state)

      def fit(self, X, y, sample_weight=None, minority_target=None):

            # Check that algorithm is supported.
            if self.algorithm not in ('SAMME', 'SAMME.R'):
                  raise ValueError("algorithm %s is not supported" %
                        self.algorithm)

            # Check parameters.
            if self.learning_rate <= 0:
                  raise ValueError("learning_rate must be greater than zero")

            if self.base_estimator is None:
                  DTYPE = np.float64  # from fast_dict.pxd
                  dtype = DTYPE
                  accept_sparse = 'csc'
            else:
                  dtype = None
                  accept_sparse = ['csr', 'csc']

            X, y = check_X_y(X, y, accept_sparse=accept_sparse, dtype=dtype,
                              y_numeric=is_regressor(self))

            if sample_weight is None:
                  # Initialize weights to 1 / n_samples.
                  sample_weight = np.empty(X.shape[0], dtype=np.float64)
                  sample_weight[:] = 1. / X.shape[0]

            ...
```

```
            else:
                  # Normalize existing weights.
                  sample_weight = sample_weight /
                        sample_weight.sum(dtype=np.float64)

                  # Check that the sample weights sum is positive.
                  if sample_weight.sum() <= 0:
                        raise ValueError(
                              "Attempting to fit with a non-positive "
                              "weighted number of samples.")

            if minority_target is None:
                  # Determine the minority class label.
                  stats_c_ = Counter(y)
                  # maj_c_ = max(stats_c_, key=stats_c_.get)
                  min_c_ = min(stats_c_, key=stats_c_.get)
                  self.minority_target = min_c_
            else:
                  self.minority_target = minority_target

            # Check parameters.
            self._validate_estimator()

            # Clear any previous fit results.
            self.estimators_ = []
            self.estimator_weights_ = np.zeros(self.n_estimators,
                  dtype=np.float64)
            self.estimator_errors_ = np.ones(self.n_estimators,
                  dtype=np.float64)

            random_state = check_random_state(self.random_state)

            for iboost in range(self.n_estimators):

                  # oversampling step.
                  X_res, y_res = self.smote.fit_resample(X, y)
                  #number of generated synthetic samples
                  n_syn = X_res.shape[0]-X.shape[0]
                  #weights for the synthetics
                  sample_weight_syn = np.empty(n_syn, dtype=np.float64)
                  sample_weight_syn[:] = 1. / X.shape[0]

                  # Combine the weights.
                  sample_weight = np.append(sample_weight,
                        sample_weight_syn).reshape(-1, 1)
                  # Combine the weights.
                  sample_weight = np.squeeze(normalize(sample_weight, axis=0,
                        norm='l1'))

                  # Boosting step.
                  sample_weight, estimator_weight, estimator_error =
                        self._boost(iboost, X_res, y_res, sample_weight,
                        random_state)

                  ...
```

```
                # Early termination.
                if sample_weight is None:
                        break

                self.estimator_weights_[iboost] = estimator_weight
                self.estimator_errors_[iboost] = estimator_error

                # Stop if error is zero.
                If estimator_error == 0:
                        break

                sample_weight = sample_weight[:X.shape[0]]
                sample_weight_sum = np.sum(sample_weight)

                # Stop if the sum of sample weights has become non-positive.
                if sample_weight_sum <= 0:
                        break

                if iboost < self.n_estimators - 1:
                        # Normalize.
                        sample_weight /= sample_weight_sum

        return self
```

Código_fuente 6: Implementación de la clase OversampleBoost que permite instanciar los clasificadores SMOTEBoost, SMOTE-Tomek-Boost y SMOTE-ENN-Boost según el nombre que se le indique como argumento en la llamada al constructor de la clase.

```
class DERSBoost(AdaBoostClassifier):

    def __init__(self, k_neighbors=5, over_ratio='auto', base_estimator=None,
      n_estimators=50,learning_rate=1., algorithm='SAMME.R', random_state=None,
      H=4, CR=0.6, F=0.5, POP_SIZE=10,NGEN=100, us_alpha=0.8,nbp_rus_size=0.5):

        self.algorithm = algorithm

        self.declu = DECLUndersampling(H=H, alpha=us_alpha,
            nbp_rus_size=nbp_rus_size, CR=CR, F=F,POP_SIZE=POP_SIZE, NGEN=NGEN)
        self.smote = SMOTE(k_neighbors=k_neighbors,
            sampling_strategy=over_ratio)
        self.desmote = DESMOTE(CR,F,POP_SIZE,NGEN)

        super(DERSBoost, self).__init__(
            base_estimator=base_estimator,
            n_estimators=n_estimators,
            learning_rate=learning_rate,
            random_state=random_state)

    def fit(self, X, y, sample_weight=None, minority_target=None):
        # Check that algorithm is supported.
        if self.algorithm not in ('SAMME', 'SAMME.R'):
            raise ValueError("algorithm %s is not supported" % self.algorithm)

        # Check parameters.
        if self.learning_rate <= 0:
            raise ValueError("learning_rate must be greater than zero")

        if self.base_estimator is None:
            DTYPE = np.float64  # from fast_dict.pxd
            dtype = DTYPE
            accept_sparse = 'csc'
        else:
            dtype = None
            accept_sparse = ['csr', 'csc']

        X, y = check_X_y(X, y, accept_sparse=accept_sparse, dtype=dtype,
                         y_numeric=is_regressor(self))

        #DE-guided undersampling step
        X, y = self.declu.undersample(X,y)

        if sample_weight is None:
            # Initialize weights to 1 / n_samples.
            sample_weight = np.empty(X.shape[0], dtype=np.float64)
            sample_weight[:] = 1. / X.shape[0]
        else:
            # Normalize existing weights.
            sample_weight = sample_weight / sample_weight.sum(dtype=np.float64)

            # Check that the sample weights sum is positive.
            if sample_weight.sum() <= 0:
                raise ValueError(
                    "Attempting to fit with a non-positive "
                    "weighted number of samples.")

      ...
```

```
        if minority_target is None:
            # Determine the minority class label.
            stats_c_ = Counter(y)
            maj_c_ = max(stats_c_, key=stats_c_.get)
            min_c_ = min(stats_c_, key=stats_c_.get)
            self.minority_target = min_c_
            self.majority_target = maj_c_
        else:
            self.minority_target = minority_target

        # Check parameters.
        self._validate_estimator()

        # Clear any previous fit results.
        self.estimators_ = []
        self.estimator_weights_ = np.zeros(self.n_estimators, dtype=np.float64)
        self.estimator_errors_ = np.ones(self.n_estimators, dtype=np.float64)

        random_state = check_random_state(self.random_state)

        for iboost in range(self.n_estimators):
            X_min = X[np.where(y == self.minority_target)]
            n_min = X_min.shape[0]

            #fit and resample with smote
            X_res, y_res = self.smote.fit_resample(X, y)

            #select synthetic samples
            X_syn = X_res[y_res==self.minority_target][n_min:]

            #DEguided selection of best synthetics
            self.desmote.fit(X,y,self.majority_target,\
                  self.minority_target,X_syn)
            selected_syn = []
            for i,value in enumerate(self.desmote.best_ind):
                  if self.desmote.best_ind[i]>0:
                        selected_syn.append(X_syn[i])
            selected_syn = np.array(selected_syn)
            y_syn = np.full(selected_syn.shape[0],
                  fill_value=self.minority_target, dtype=np.int64)

            # Normalize synthetic sample weights based on current training set.
            sample_weight_syn = np.empty(selected_syn.shape[0],
                  dtype=np.float64)
            sample_weight_syn[:] = 1. / X.shape[0]

            # Combine the original and synthetic samples.
            print(" SE HAN INCLUIDO {} EJEMPLOS
                  SINTÉTICOS".format(selected_syn.shape[0]))
            X_train = np.vstack((X, selected_syn))
            y_train = np.append(y, y_syn)

            # Combine the weights.
            sample_weight = np.append(sample_weight,
                  sample_weight_syn).reshape(-1, 1)
            sample_weight = np.squeeze(normalize(sample_weight, axis=0,
                  norm='l1'))

    ...
```

```
            # Boosting step.
            sample_weight, estimator_weight, estimator_error =
                  self._boost(iboost, X_train, y_train,\
                  sample_weight, random_state)

            # Early termination.
            if sample_weight is None:
                  print("sample_weight is None")
                  break

            self.estimator_weights_[iboost] = estimator_weight
            self.estimator_errors_[iboost] = estimator_error

            # Stop if error is zero.
            if estimator_error == 0:
                  #print("estimator_error")
                  break

            sample_weight = sample_weight[:X.shape[0]]
            sample_weight_sum = np.sum(sample_weight)

            # Stop if the sum of sample weights has become non-positive.
            if sample_weight_sum <= 0:
                  print("sample_weight_sum <= 0")
                  break

            if iboost < self.n_estimators - 1:
                  # Normalize.
                  sample_weight /= sample_weight_sum
      return self
```

Código_fuente 7: Implementación del clasificador DERS-Boost

```
class DECLUndersampling(object):
    def __init__(self, H, alpha, nbp_rus_size, CR, F, POP_SIZE, NGEN):
        self.H = H
        self.alpha = alpha
        self.nbp_rus_size = nbp_rus_size

        self.decl = DEClustering(CR, F, POP_SIZE, NGEN)


    def clustering_centers(self,X,y,maj_class,min_class):
        clustering_centers = []
        for i in range(int(self.H)):
            self.decl.fit(X,y,maj_class,min_class)
            centers = self.decl.best_ind
            clustering_centers.append(centers)
        return clustering_centers

    def cluster_stabilities(self,majority_samples,maj_class,min_class,\
      clustering_centers):
        classes = [maj_class,min_class]

        cluster_stabilities = []
        for sample in majority_samples:
            S = 0
            for clustering in clustering_centers:
                c = classes[classify(sample,clustering)]
                if c==maj_class:
                    S += 1
            cluster_stabilities.append(S/self.H)
        return cluster_stabilities

    def undersample(self,X,y):
        #compute H clustering processes
        cl_c = self.clustering_centers(X,y,-1,1)

        #compute cluster stability for each majority sample
        majority_samples = X[y==-1]
        cl_st = self.cluster_stabilities(majority_samples,-1,1,cl_c)

        #compute boundary and non-boundary samples
        boundary_points = majority_samples[np.array(cl_st)<=self.alpha]
        non_boundary_points = majority_samples[np.array(cl_st)>self.alpha]

        #undersample non-boundary samples
        nbp_us = rus(non_boundary_points,self.nbp_rus_size)

        #build undersampled training set
        X_US, y_US = unify_training_set(X,y,-1,1,boundary_points,nbp_us)

        return X_US, y_US
```

Código_fuente 8: Implementación de la clase DECLUndersampling, que lleva a cabo el algoritmo de sub-muestreo de los ejemplos redundantes de la clase mayoritaria propuesto en DERS-Boost.

```
class DEClustering(object):

    def __init__(self, CR,F,POP_SIZE,NGEN):

        self.CR = CR
        self.F = F
        self.POP_SIZE = POP_SIZE
        self.NGEN = NGEN

    def fit(self, X_train, y_train,maj_class,min_class):
        creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
        creator.create("Individual", array.array, typecode='d',
            fitness=creator.FitnessMin)
        self.toolbox = base.Toolbox()
        self.toolbox.register("select", tools.selRandom, k=3)
        self.toolbox.register("population",load_individuals, X_train, y_train,
            maj_class, min_class, creator.Individual)
        self.toolbox.register("evaluate", evaluate, X_train)

        NDIM = X_train.shape[1]

        pop = self.toolbox.population(n=self.POP_SIZE);
        hof = tools.HallOfFame(1)
        stats = tools.Statistics(lambda ind: ind.fitness.values)
        stats.register("avg", np.mean)
        stats.register("std", np.std)
        stats.register("min", np.min)
        stats.register("max", np.max)

        logbook = tools.Logbook()
        logbook.header = "gen", "evals", "std", "min", "avg", "max"

        # Evaluate the individuals
        fitnesses = self.toolbox.map(self.toolbox.evaluate, pop)
        for ind, fit in zip(pop, fitnesses):
            ind.fitness.values = fit

        record = stats.compile(pop)
        logbook.record(gen=0, evals=len(pop), **record)

        for g in range(1, self.NGEN):
            for k, agent in enumerate(pop):
                a,b,c = self.toolbox.select(pop)
                y = self.toolbox.clone(agent)
                index = random.randrange(NDIM)
                for i, value in enumerate(agent):
                    if i == index or random.random() < self.CR:
                        y[i] = a[i] + self.F*(b[i]-c[i])
                y.fitness.values = self.toolbox.evaluate(y)
                if y.fitness > agent.fitness:
                    pop[k] = y

            hof.update(pop)
            record = stats.compile(pop)
            logbook.record(gen=g, evals=len(pop), **record)
        self.best_ind = hof[0]
        self.best_fitness = hof[0].fitness.values[0]
        return self
```

Código_fuente 9: Clase DEClustering, que implementa el algoritmo evolutivo diferencial que guía los clusters

```
class DESMOTE(object):
    def __init__(self, CR, F ,POP_SIZE, NGEN, p=0.2):
        self.CR = CR
        self.F = F
        self.POP_SIZE = POP_SIZE
        self.NGEN = NGEN
        self.p = p

    def fit(self, X, y, maj_class, min_class, syn):

        creator.create("FitnessMax", base.Fitness, weights=(1.0,))
        creator.create("Individual", array.array, typecode='d',
            fitness=creator.FitnessMax, clf=None)
        self.toolbox = base.Toolbox()
        self.toolbox.register("attr_int", random.randint, 0, 1)

        self.toolbox.register("select", tools.selRandom, k=3)
        NDIM = syn.shape[0]

        self.toolbox.register("individual", tools.initRepeat,
            creator.Individual, self.toolbox.attr_int, NDIM)
        self.toolbox.register("population", tools.initRepeat, list,
            self.toolbox.individual)
        self.toolbox.register("evaluate", compute_fitness, self.p, X, y,
            maj_class, min_class, syn)

        pop = self.toolbox.population(n=self.POP_SIZE);
        hof = tools.HallOfFame(1)
        stats = tools.Statistics(lambda ind: ind.fitness.values)
        stats.register("avg", np.mean)
        stats.register("std", np.std)
        stats.register("min", np.min)
        stats.register("max", np.max)

        logbook = tools.Logbook()
        logbook.header = "gen", "evals", "std", "min", "avg", "max"

        # Evaluate the individuals
        fitnesses = self.toolbox.map(self.toolbox.evaluate, pop)
        for ind, fit in zip(pop, fitnesses):
            ind.fitness.values = fit

        record = stats.compile(pop)
        logbook.record(gen=0, evals=len(pop), **record)

      ...
```

```
        for g in range(1, self.NGEN):
            for k, agent in enumerate(pop):
                a,b,c = self.toolbox.select(pop)
                #we adopt a self-adaptative operator
                l = math.exp(1-(self.NGEN/(self.NGEN+1-g)))
                F_g = self.F*(2**l)
                d = self.toolbox.clone(agent) #donor vector
                sig_d = self.toolbox.clone(agent)
                y = self.toolbox.clone(agent)
                index = random.randrange(NDIM)
                for i, value in enumerate(agent):
                    d[i] = a[i] + F_g*(b[i]-c[i]) #donor vector
                    #the mutated donor is mapped to binary space by a sigmoid
                        #function with displacement
                    sig_d[i] = round(1/(1+math.exp(-(d[i]))))
                    if i == index or random.random() < self.CR:
                        y[i] = sig_d[i]
                y.fitness.values = self.toolbox.evaluate(y)
                if y.fitness > agent.fitness:
                    pop[k] = y
                #print(pop[k].fitness)
            hof.update(pop)
            record = stats.compile(pop)
            logbook.record(gen=g, evals=len(pop), **record)
        self.best_ind = hof[0]
        self.best_fitness = hof[0].fitness.values[0]
        return self
```

Código_fuente 10: Clase DESMOTE, que implementa el DE que guía la selección del mejor subconjunto de sintéticos con el que entrenar un clasifiacdor base.

APÉNDICE B

Resultados para ambas métricas de rendimiento:
media geométrica entre el Recall y el TNR y f1-score

En este apéndice se muestran los resultados del rendimiento para cada clasificador entrenado con cada dataset para las métricas de la media geométrica y el f1-score. El orden en el que se disponen las tablas viene a ser el mismo que el de la experimentación llevada a cabo en el apartado 5.

ID	ADA-BOOST		RUS-BOOST		SMOTE-BOOST		SMOTE-TOMEK-BOOST		SMOTE-ENN-BOOST		DERS-BOOST	
Métrica	GM	F1	GM	F1	GM	F1	GM	F1	GM	F1	GM	F1
eco	73.30	57.97	71.50	55.27	68.15	52.99	68.70	53.54	84.93	68.36	66.49	50.39
spec	85.05	76.63	85.53	68.13	84.60	65.44	84.67	65.89	83.51	66.60	83.96	64.79
led	83.38	68.51	82.71	67.82	81.49	60.57	81.49	60.57	81.63	64.90	81.96	60.84
glass	30.32	20.03	61.08	31.25	54.69	32.43	52.91	30.39	64.67	30.66	57.60	36.84
clev	51.33	39.25	65.01	46.07	84.57	74.14	84.79	75.92	83.51	65.52	78.43	67.30
lib	39.55	18.66	78.58	58.78	82.94	63.66	82.94	63.66	78.15	59.41	76.67	55.58
ye	53.45	29.65	62.26	29.68	56.77	28.77	57.59	30.10	69.07	29.71	57.99	30.37
page	80.51	68.74	93.93	82.15	87.33	77.24	87.33	77.24	94.05	84.15	88.65	78.30
der	98.87	95.85	92.25	84.28	97.89	95.55	97.89	95.55	93.67	86.66	95.14	90.09
arr	75.20	60.57	89.16	67.84	79.93	63.63	79.93	63.63	88.26	69.86	76.51	62.20
oil	64.91	29.64	57.06	21.93	51.69	20.53	48.50	17.24	55.08	16.30	56.26	20.33
shut	100.00	100.00	98.04	97.77	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
pok	14.14	13.33	51.08	19.07	38.29	25.33	38.29	25.33	42.78	26.44	20.28	10.66
win	23.09	8.25	38.99	9.61	13.96	6.55	9.28	3.88	9.05	2.00	0.00	0.00
MEDIA	**62.36**	**49.08**	**73.37**	**52.83**	**70.16**	**54.77**	**69.59**	**54.50**	**73.45**	**55.04**	**67.14**	**51.98**

Tabla 13: *Resultados de rendimiento (media geométrica y f1-score) obtenidos para cada uno de los datasets con los clasificadores Ada-Boost, RUS-Boost, SMOTE-Boost, SMOTE-Tomek-Boost, SMOTEENN-Boost y DERS-Boos.*

ID	DE_Under+C4.5		KMEANS_Under+C4.5	
Métrica	**GM**	**F1**	**GM**	**F1**
eco	72.43	57.21	74.76	59.82
spec	83.65	72.00	84.56	76.46
led	84.59	69.57	84.70	69.63
glass	47.68	30.25	32.15	21.01
clev	64.03	46.00	54.90	43.17
lib	55.65	34.34	36.85	18.22
ye	56.84	33.41	52.99	28.98
page	87.64	74.60	80.51	68.74
der	99.85	97.77	96.03	92.67
arr	72.81	59.98	70.77	55.97
oil	57.96	24.96	63.22	29.32
shut	100.00	100.00	100.00	100.00
pok	14.04	11.11	14.14	13.33
win	13.89	5.23	18.54	7.14
MEDIA	**65.07**	**51.15**	**61.72**	**48.89**

Tabla 14: *Resultados de rendimiento (media geométrica y f1-score) obtenidos para cada uno de los datasets con los clasificadores DE_Under y KMEANS_Under.*

ID	DE_Under+C4.5		KMEANS_Under + C4.5		RUS+C4.5		Tomek+C4.5		ENN+C4.5	
Métrica	GM	F1	GM	F1	GM	F1	GM	F1	GM	F1
eco	72.43	57.21	74.76	59.82	83.11	64.15	66.58	52.54	84.03	65.67
spec	83.65	72.00	84.56	76.46	82.07	46.83	84.47	77.26	90.18	82.67
led	84.59	69.57	84.70	69.63	77.82	55.44	85.02	70.20	82.62	64.25
glass	47.68	30.25	32.15	21.01	65.93	24.34	26.81	15.92	32.45	20.59
clev	64.03	46.00	54.90	43.17	70.02	32.55	58.98	45.37	49.80	32.89
lib	55.65	34.34	36.85	18.22	68.31	27.88	39.09	20.79	39.01	20.54
ye	56.84	33.41	52.99	28.98	64.46	19.83	55.34	31.27	55.28	24.33
page	87.64	74.60	80.51	68.74	93.79	76.48	78.78	66.52	76.80	61.15
der	99.85	97.77	96.03	92.67	97.77	84.56	94.98	89.77	90.75	80.88
arr	72.81	59.98	70.77	55.97	89.42	52.00	67.44	52.03	81.45	64.21
oil	57.96	24.96	63.22	29.32	67.62	23.41	63.76	31.16	58.45	25.73
shut	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
pok	14.04	11.11	14.14	13.33	56.57	11.90	14.14	13.33	28.13	23.33
win	13.89	5.23	18.54	7.14	40.12	3.78	13.91	5.47	13.87	5.05
MEDIA	**65.07**	**51.15**	**61.72**	**48.89**	**75.50**	**44.51**	**60.66**	**47.97**	**63.06**	**47.95**

Tabla 15: *Resultados de rendimiento (media geométrica y f1-score) obtenidos para cada uno de los datasets con los clasificadores DE_Under (DECL+4.5), KMEANS_Under (KMEANSCL+C4.5), RUS+C4.5, Tomek+C4.5 y ENN+C4.5.*

ID	SMOTE+DE		SMOTE		ROS		SMOTE+Tomek		SMOTE+ENN	
Métrica	GM	F1	GM	F1	GM	F1	GM	F1	GM	F1
eco	70.50	53.47	68.78	53.45	66.56	53.42	70.67	55.67	84.90	68.92
spec	81.02	62.65	84.20	67.31	77.78	62.32	83.84	66.73	84.73	67.64
led	84.31	68.61	84.97	69.34	82.47	63.92	84.97	69.34	82.89	67.29
glass	51.67	30.79	57.16	34.32	49.02	33.51	50.28	27.67	62.18	28.50
clev	78.67	68.15	81.08	71.03	41.77	29.49	81.11	70.96	86.98	69.07
lib	79.83	57.41	81.10	60.42	64.53	46.79	81.10	60.42	80.66	58.63
ye	63.49	34.06	63.35	33.32	55.96	34.28	60.70	31.64	69.85	30.55
page	93.38	84.23	87.94	77.80	85.49	74.70	87.94	77.80	93.79	83.55
der	94.25	89.14	97.01	92.97	93.27	88.84	97.01	92.97	91.72	82.81
arr	75.30	87.35	76.84	63.46	75.99	61.31	76.84	63.46	87.45	67.38
oil	40.43	14.00	50.52	18.46	42.87	21.60	51.30	18.12	54.47	17.75
shut	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
pok	17.68	10.44	28.96	18.66	32.74	26	28.96	18.66	38.31	27.11
win	4.64	1.90	13.98	6.66	0.00	0.00	18.50	7.61	4.57	1.21
MEDIA	**66.80**	**52.30**	**69.71**	**54.80**	**62.03**	**49.73**	**69.51**	**54.36**	**73.04**	**55.03**

Tabla 16: *Resultados de rendimiento (media geométrica y f1-score) obtenidos para cada uno de los datasets construyendo un clasificador base C4.5 y aplicando cada una de las técnicas: SMOTE+DE, ROS, SMOTE, SMOTE+Tomek* y *SMOTE+ENN.*

ID	DE_Under + SMOTE-ENN-Boost		SMOTE-ENN-BOOST		DERS-BOOST	
Métrica	**GM**	**F1**	**GM**	**F1**	**GM**	**F1**
eco	83.36	62.37	84.93	68.36	66.49	50.39
spec	85.13	66.20	83.51	66.60	83.96	64.79
led	80.67	65.66	81.63	64.90	81.96	60.84
glass	63.01	29.45	64.67	30.66	57.60	36.84
clev	71.57	54.10	83.51	65.52	78.43	67.30
lib	82.59	62.09	78.15	59.41	76.67	55.58
ye	61.08	25.07	69.07	29.71	57.99	30.37
page	86.73	71.28	94.05	84.15	88.65	78.30
der	96.82	92.14	93.67	86.66	95.14	90.09
arr	87.64	68.65	88.26	69.86	76.51	62.20
oil	59.64	20.97	55.08	16.30	56.26	20.33
shut	100.00	100.00	100.00	100.00	100.00	100.00
pok	35.58	25.33	42.78	26.44	20.28	10.66
win	0.00	0.00	9.05	2.00	0.00	0.00
MEDIA	**70.99**	**53.09**	**73.45**	**55.04**	**67.14**	**51.98**

Tabla 17: Rendimiento sobre la métrica F1-score obtenido para cada uno de los datasets con los clasificadores DE_Under + SMOTE-ENN-Boost, SMOTEENN-Boost y DERS-Boost.