UNIVERSIDAD DE CÓRDOBA

GRADO EN INGENIERÍA INFORMÁTICA
ESCUELA POLITÉCNICA SUPERIOR

Trabajo fin de Grado:

Creación de patrones sintéticos para conjuntos de datos desbalanceados mediante la metaheurística voraz iterativa.

Manual de código

***Autor**:*

Francisco Javier Maestre García

Directores

D. Pedro Antonio Gutiérrez Peña

D. Carlos García Martínez

Enero, 2017

UNIVERSIDAD DE CÓRDOBA

GRADO EN INGENIERÍA INFORMÁTICA
ESCUELA POLITÉCNICA SUPERIOR

Trabajo fin de Grado:

Creación de patrones sintéticos para conjuntos de datos desbalanceados mediante la metaheurística voraz iterativa.

Manual de código

***Autor**:*

Francisco Javier Maestre García

Directores

D. Pedro Antonio Gutiérrez Peña

D. Carlos García Martínez

Enero, 2017

Datos del proyecto

Título: Creación de patrones sintéticos para conjuntos de datos desbalanceados mediante la metaheurística voraz iterativa.

Autor: *Francisco Javier Maestre García*
Email: *I12magaf@uco.es*
***DNI**: 31012172-F*
***Especialidad**: Grado en ingeniería informática*
***Directores del proyecto**:*

- D. Pedro Antonio Gutiérrez Peña
- D. Carlos García Martínez

Firma del autor y directores

El autor:

Fdo.: Francisco Javier Maestre García

Los directores:

Fdo.: Pedro Antonio Gutiérrez Peña

Fdo.: Carlos García Martínez

AGRADECIMIENTOS

Índice

Índice de Tablas

Índice de Figuras

Capítulo 1:
Introducción

Este es el manual de código del trabajo fin de grado ***Creación de patrones sintéticos para conjuntos de datos desbalanceados mediante la metaheurística voraz iterativa***, donde se incluye una descripción detallada de todos los elementos que componen el sistema desarrollado en el mismo. El objetivo de este documento es servir de manual de código para futuras modificaciones o mejoras que puedan desarrollarse sobre la aplicación *software*. En este manual encontraremos el código de la aplicación ordenado por módulos.

Por último es importante recordar que este manual es tan solo una referencia del código de la aplicación y que por tanto, para comprender completamente el funcionamiento de la misma es imprescindible la consulta del *Manual Técnico del trabajo*.

El objetivo de este proyecto ha sido desarrollar un algoritmo de sobremuestreo optimizado mediante una metaheurística como es *Iterated Greedy*, que sea capaz de clasificar de forma considerablemente satisfactoria bases de datos desbalanceadas.

Se ha utilizado el lenguaje de programación *Python* para este propósito. *Python* es uno de los lenguajes de programación dinámica más conocidos actualmente. Se encuentra entre los más populares junto a Perl, Tcl, PHP o Ruby. Este lenguaje, a menudo, suele ser considerado como lenguaje de *scripting* Un lenguaje interpretado, o lenguaje de *scripting*, es un lenguaje de programación que está diseñado para ser ejecutado por medio de un intérprete, en contraste con los lenguajes compilados, a pesar de que realmente es un lenguaje de propósito general.

Científicos de todo el mundo diseñan y programan cada día aplicaciones con un alto nivel de complejidad computacional para ser ejecutados en supercomputadores, y lo hacen en *Python* debido a su sencillez.

En este manual explicaremos cada uno de estos módulos y expondremos tanto la organización como la estructura de los mismos, poniendo énfasis en las características más relevantes del diseño que se ha seguido, así como su funcionalidad y cometido dentro del conjunto del proyecto.

Finalmente, se hará una exposición detallada de cada una de las funciones *Python* que componen los módulos desarrollados, explicándolas según el siguiente guion:

- **Nombre de la función.**

- **Descripción General de la función**: Análisis del cometido de la función dentro de su módulo, describiendo cada una de las funcionalidades del mismo que debe cubrir su desarrollo.

- **Variables de entrada de la función**: Descripción de cada una de las entradas de cada clase.

- **Variables de salida de la función**: Descripción de cada una de las salidas de cada clase.

- **Código de la función.**

Capítulo 2:
Estructura general del sistema

A partir de la descripción funcional del sistema, podemos extraer que nuestro trabajo estará constituido principalmente por tres funcionalidades claramente diferenciadas.

De esta forma, nuestro trabajo presenta tres módulos independientes.

- **Gestión de datos de entrada:** Este módulo podrá leer una base de datos indicada por el usuario y deberá prepararla para que el sistema pueda ejecutarse correctamente. También podrá analizar la base de datos en busca de patrones duplicados ya que estos pueden perjudicar al algoritmo, especialmente en las clases con una cantidad de patrones reducida.

- **Algoritmo de over-sampling:** En este módulo se engloban todas las operaciones que operan y modifican el conjunto introducido por el módulo anterior, como por ejemplo un método para la eliminación de ruido, el algoritmo de *over-sampling* y la metaheurística que lo optimiza. Una vez se ha generado un conjunto de sintéticos que equilibre la distribución de los datos, los conjuntos de datos oportunos son pasados al siguiente módulo.

- **Tratamiento de resultados**: Este módulo recoge las salidas generadas en el módulo anterior, las evalúa, y finalmente gestiona los resultados, ya sea mostrándolos por pantalla o bien almacenándolos en un archivo.

En la Figura 1: Diagrama de módulos del sistema, se muestra más detalladamente como ha sido diseñado el sistema y cuáles son las funciones que se incluyen en cada módulo.

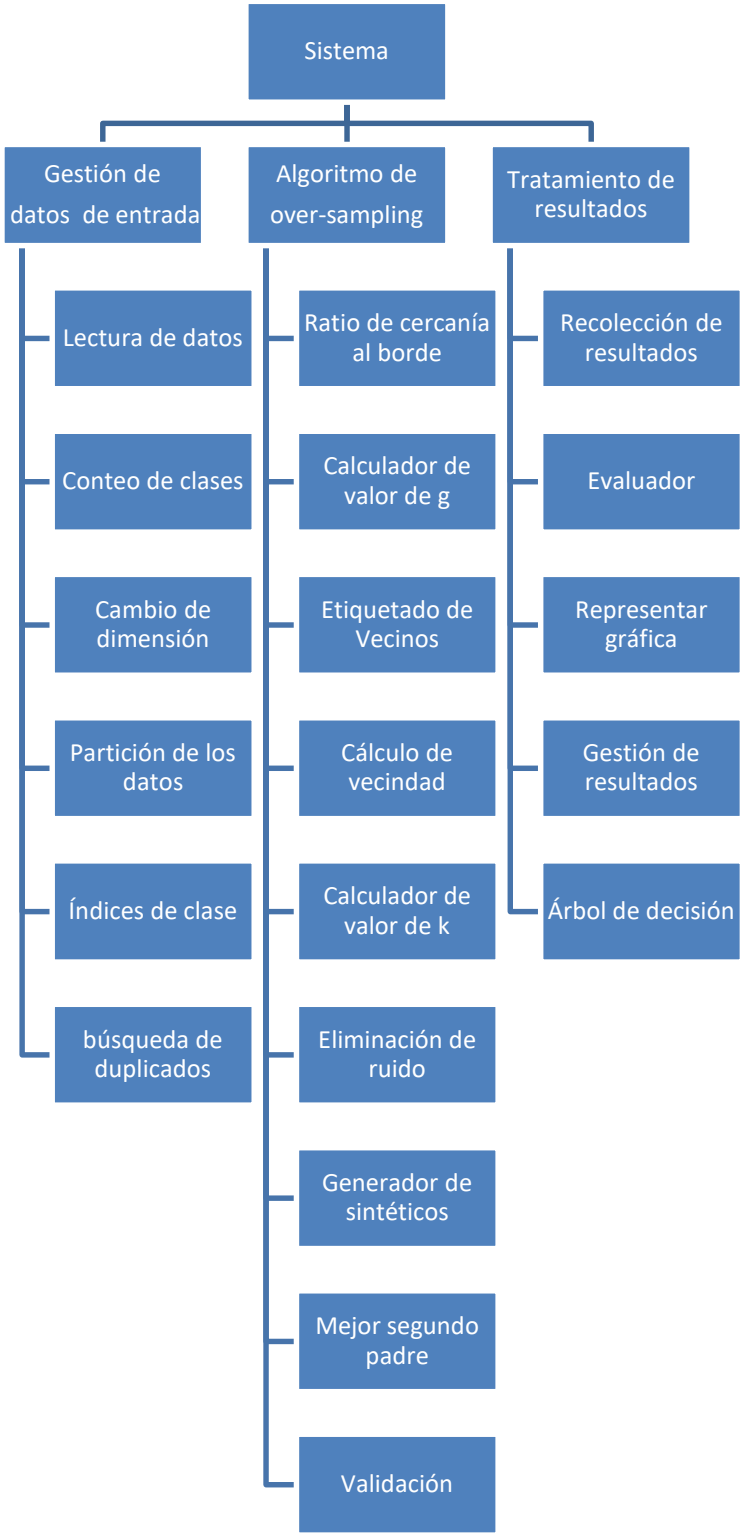

Figura 1: Diagrama de módulos del sistema

Capítulo 3:
Desarrollo y codificación

Una vez expuesta la estructura general de nuestro trabajo, en este capítulo explicaremos en detalle la implementación de cada una de las funciones que forman los módulos y que se han desarrollado para cumplir con los objetivos establecidos en nuestro trabajo.

Para ello, comenzaremos exponiendo un pequeño resumen inicial acerca de las características más importantes, tanto de la codificación desarrollada, como del proceso de documentación del trabajo. Finalmente expondremos uno a uno los componentes de cada uno de los dos módulos desarrollados en este trabajo.

3.1. Características de la implementación

Se ha utilizado el lenguaje de programación *Python* para este propósito. *Python* es uno de los lenguajes de programación dinámica más conocidos actualmente.

3.1.1. Notación de la implementación

En relación a la estandarización en la programación, se ha procurado seguir la guía de estilo PEP8 de *Python*.

En cuanto al código en sí, se seguirán las convenciones recomendadas de nombramiento tanto general como específico. Algunas de ellas son:

- Todos los nombres deben ser lo más auto explicativos posible.
- Los nombres de las variables y de las funciones deben estar en minúscula, aun cuando el nombre es la unión de varias palabras. En ese caso la primera palabra debe empezar por minúscula y cada una de las posteriores precedidas por un guion bajo, siendo el resto de letras minúsculas.
- Las abreviaciones y acrónimos no deben ser escritos en mayúscula cuando sean utilizados como nombres.

Como este Trabajo se ha inspirado en algoritmos anteriores, se han respetado algunos nombres de variables que eran recurrentes en la mayoría de ellos, por tanto se han considerado como excepciones y no han sido adaptados a las normas descritas arriba.

3.1.2. Estructuración de la documentación

Para documentar de forma detallada el código desarrollado en nuestro trabajo, se seguirá una estructura dividida en los diferentes módulos que conforman el sistema, seguido de una descripción general del cometido de cada función incluida en dichos módulos, para a continuación exponer mediante tablas el conjunto de variables de entrada y salida que componen a las mismas.

3.2. Módulo de gestión de datos de entrada.

Este módulo es el encargado, no solo de leer los datos de entrada, sino también de realizar todo tratamiento necesario al conjunto para que el siguiente módulo pueda ejecutarse correctamente. Además, debe identificar e intentar corregir algunas de las incoherencias más importantes que no deben aparecer en ningún conjunto de datos. Este módulo cumple con las siguientes funcionalidades.

3.2.1. Lectura de datos

Esta función debe ser capaz de leer correctamente la base de datos proporcionada por el usuario, identificar claramente las entradas y las salidas y separarlas en dos estructuras diferentes para que sea más fácil trabajar con ellas a lo largo de la ejecución del algoritmo.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
ruta	Ruta de la base de datos incluyendo el nombre del archivo.	--
rmv_dup	Si es True, Se activará la función que elimina los duplicados.	
Salidas		
Nombre	*Descripción*	
inputs	Matriz con las variables de entrada.	
outputs	Matriz con las variables de salida.	

Tabla 1: Función referente a la lectura de datos

Código:

```
def lectura_datos(ruta, rmv_dup=False):

    try:
        data_set = pd.read_csv(ruta, header=None)
    except IOError:
        print "no existe la BBDD introducida\n"
        sys.exit("Parando ejecución")

    if rmv_dup:
        data_set = buscar_duplicados(data_set)
    # Normaliza las variables de entrada
    # Estandarización con media=0 y varianza=1
    inputs  =  preprocessing.scale(np.array(data_set.values[:,  0:-1],
dtype='float'))
    outputs = data_set.values[:, -1]

    return inputs, outputs
```

Código 1: Referente a la lectura de datos

3.2.2. Conteo de clases

Este método identificaría el número de clases que existen en el *dasaset*, y el número de veces que aparece cada una de ellas, de esta forma se puede analizar el grado de desbalanceo que existe y proceder en consecuencia.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
outputs	Matriz con las variables de salida.	--
Salidas		
Nombre	*Descripción*	
labels	Matriz con las clases y su tamaño (ordenada ascendentemente por columna 1) • Columna 0: Etiqueta de clase. • Columna 1: Tamaño de clase.	

Tabla 2: Función referente al conteo e identificación de las clases

Código:

```
def conteo_clases(outputs):

    labels = itemfreq(outputs)
    labels = labels[labels[:, 1].argsort()]

    return labels
```

Código 2: Referente al conteo e identificación de las clases

3.2.3. Cambio de dimensión

El sistema está diseñado para trabajar con problemas de solo dos clases. No obstante, es posible realizar una transformación de un problema de más clases en uno de tan solo dos de ellas, de forma que el sistema pueda ejecutarse correctamente y conseguir resultados satisfactorios. El sistema analiza el grado de presencia de cada clase en la totalidad del conjunto original, y de menor a mayor, todas aquellas clases con una representación acumulada menor que un determinado grado introducido por el usuario, serán incluidas dentro de la definición de clase minoritaria. Las clases restantes formaran la clase mayoritaria. Para configurar este parámetro correctamente es muy importante que el usuario conozca la distribución de las clases dentro del conjunto original. Sin embargo, si no se desea introducir ningún grado de corte, el sistema considerará como clase minoritaria solo la clase más minoritaria de todas.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
outputs	Matriz con las variables de salida.	--
imbalanced_degree (opcional)	Grado de desbalanceo para determinar las clases minoritarias. Si no se introduce solo se considerará una clase como minoritaria y todas las demás mayoritarias.	None
Salidas		
Nombre	*Descripción*	
outputs	Variables de salida (con solo dos clases).	

Tabla 3: Función referente al cambio de dimensión

Código:

```
    def bi_clase(outputs, imbalanced_degree=None):

        clases = conteo_clases(outputs)
        if imbalanced_degree is not None:
            imbalance = clases[:, 1] / float(clases[:, 1].sum())  # imbalance
ratio
            imbalance = np.cumsum(imbalance)
            etiquetas = clases[:, 0][imbalance < imbalanced_degree]
            indices = np.where(np.in1d(outputs, etiquetas))[0]
            mask = np.zeros(outputs.shape[0], dtype=bool)
            mask[indices] = True
            if not etiquetas.size:
                print 'Para la BBDD utilizada, el valor de imbalanced_degree
debe ser almenos superior a', imbalance[0]
                print 'Introduce un valor superior a este o "None", para que
el sistema pueda ejecutarse'
                sys.exit("Valor erroneo para imbalanced_degree")
        else:
            mask = (outputs == clases[0][0])

        return mask.astype(int)
```

Código 3: Referente al cambio de dimensión

3.2.4. Partición de los datos

El sistema debe segmentar tanto las variables de entrada como las de salida en 3 subconjuntos que llamaremos *Train*, *Test* y *Validation*, o en dos si no desea en conjunto de validación. La partición será estratificada, por lo que la distribución de las clases deberá seguir el mismo grado en cada uno de los conjuntos así como en el conjunto original. En esta partición no deben crearse incoherencias, y la etiqueta asociada a cada grupo de entradas debe mantenerse igual en todo momento.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
inputs	Matriz con las variables de entrada.	--
outputs	Matriz con las variables de salida.	--
test_size	Porción estratificada de la base de datos que se utilizará en el test.	0.5
train_size	Porción estratificada de la base de datos que se utilizará en el entrenamiento.	0.5
val_size (opcional)	Porción estratificada de la base de datos que se utilizará en validación.	0
Salidas		
Nombre	*Descripción*	
train_inputs	Matriz con las variables de entrada de entrenamiento.	
train_outputs	Matriz con las variables de salida de entrenamiento.	
test_inputs	Matriz con las variables de entrada de test.	
test_outputs	Matriz con las variables de salida de test.	
val_inputs (opcional)	Matriz con las variables de entrada de validación.	
val_outputs (opcional)	Matriz con las variables de salida de validación.	

Tabla 4: Función referente a la partición de los datos

Código:

```
def particion_datos(inputs, outputs, test_size=0.5, train_size=0.5,
val_size=0):

    # La suma de los tres conjuntos debe ser 1
    if round(test_size + train_size + val_size, 3) != 1:
        sys.exit("En particion_datos: Los tamaños relativos deben sumar
estrictamente 1.0")

    # El tamaño relativo de los conjuntos debe estar entre 0 y 1
    if not (0 <= test_size <= 1 and 0 <= train_size <= 1 and 0 <= val_size
<= 1):
        sys.exit("En particion_datos: Los tamaños relativos deben estar
todos entre 0.0 y 1.0")

    train_inputs = train_outputs = test_inputs = test_outputs = val_inputs =
val_outputs = None

    train_inputs, test_inputs, train_outputs, test_outputs =
train_test_split(inputs, outputs, test_size=test_size,

stratify=outputs)

    if val_size > 0:
        train_inputs, val_inputs, train_outputs, val_outputs =
train_test_split(train_inputs, train_outputs,

test_size=(val_size / (1 - test_size)),

stratify=train_outputs)

        return train_inputs, train_outputs, test_inputs, test_outputs,
val_inputs, val_outputs
    return train_inputs, train_outputs, test_inputs, test_outputs
```

Código 4: Referente a la partición de los datos

3.2.5. Índices de clase

La aplicación cuenta con un método para identificar de forma sencilla los índices de los parones de cada clase

Entradas		
Nombre	*Descripción*	*Valor por defecto*
label	Etiqueta de la clase a identificar.	--
outputs	Matriz con las variables de salida.	--
Salidas		
Nombre	*Descripción*	
index	Máscara binaria que representa los índices de la clase identificada.	

Tabla 5: Función referente a la identificación de índices de una clase

Código:

```
def class_index(outputs, label):
    index = outputs == label
    return index
```

Código 5: referente a la identificación de índices de una clase

3.2.6. Búsqueda de duplicados

Esta función busca los posibles duplicados que existan en el conjunto original y los eliminará de forma automática.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
data_set	Conjunto de datos.	--
Salidas		
Nombre	*Descripción*	
data_set	Conjunto de datos (sin duplicados).	

Tabla 6: Función referente a la identificación de duplicados

Código:

```
def buscar_duplicados(data_set):
    data_set.drop_duplicates(keep='first', inplace=True)
    return data_set
```

Código 6: Referente a la identificación de duplicados

3.3. Módulo Algoritmo de over-sampling

Este es el modulo más importante del sistema ya que es el que incluye el algoritmo de over-sampling diseñado y la metaheurística que lo optimiza. En él se recogen los datos preparados en el módulo anterior, se generan los patrones sintéticos utilizando todos los métodos necesarios, se unen los sintéticos al conjunto de *Train* para que quede balanceado, y se le pasan los conjuntos de *Train* y *Test* al siguiente modulo para que sean evaluados. Las funcionalidades que se incluyen en este módulo se definen en las siguientes tablas.

3.3.1. Ratio de cercanía al borde

Esta función calculará el ratio de cercanía al borde (vecinos clase mayoritaria / *k* vecinos) y normalizará dicho ratio con respecto a suma total del ratio de cada patrón. Este ratio normalizado simbolizará la importancia de cada patrón, y será utilizado posteriormente para asignarle un número de sintéticos a generar.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
inputs_set1	Conjunto de entrada al que se le calcularán los vecinos con respecto a *inputs_set2.*	--
inputs_set2	Conjunto de entrada de donde se calcularán los vecinos de *inputs_set1*.	--
outputs_set2	Matriz con las variables de salida de *inputs_set2*.	--
minority_label	Etiqueta de la clase minoritaria.	--
k(opcional)	Número de vecinos a tener en cuenta (si no se proporciona se calculará automáticamente).	None
return_rlist(opcional)	Si es True, también devuelve *rlist* sin normalizar.	False
Salidas		
Nombre	*Descripción*	
normalizedrlist	Vector normalizado con los ratios de cercanía al borde.	
rlist(opcional)	Vector con los ratios de cercanía al borde sin normalizar.	

Tabla 7: Función referente al cálculo de cercanía al borde

Código:

```
def ratio_borde(inputs_set1, inputs_set2, outputs_set2, minority_label,
k=None, return_rlist=False):

    neighbors, k = neighbors_calculator(inputs_set1, inputs_set2,
outputs_set2, minority_label, k, self_neighbour=False)

    # todo aquello que no sea clase minoritaria se considerará clase
mayoritaria
    majorclass_neighbors = np.sum(neighbors != minority_label, axis=1,
dtype='float')

    rlist = majorclass_neighbors / k
```

```
    if rlist.sum():
        normalizedrlist = rlist / rlist.sum()
    else:
        normalizedrlist = rlist

    if return_rlist:
        return normalizedrlist, k, rlist
    return normalizedrlist, k
```

Código 7: Referente al cálculo de cercanía al borde

3.3.2. Calculador de valor de *g*

Esta función calcula y asigna a cada patrón cuantos sintéticos debe generar (*g*), para ello utilizará el ratio de cercanía al borde normalizado y multiplicará cada valor por G (número total de sintéticos a generar). Posteriormente redistribuye aquellas asignaciones que posean una parte decimal.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
normalizedrlist	Vector normalizado con los ratios de cercanía al borde.	--
G	Número de sintéticos a generar.	--
Salidas		
Nombre	*Descripción*	
g	Vector con los sintéticos que debe generar cada patrón de la clase minoritaria.	

Tabla 8: Función referente al cálculo de g

Código:

```
def g_calculator(normalizedrlist, G):

    g_float = normalizedrlist * G

    # Primero asigna a cada patrón su parte entera correspondiente
    g_trunc = np.trunc(g_float)
    restantes = G - int(sum(g_trunc))

    if restantes:
        # Utiliza el resto para asignar los restantes hasta G
        resto = g_float - g_trunc
        probabilities = resto / restantes

        # Los que más resto tengas tienen más posibilidades de ser elegidos
para generar un sintético adicional
        elegidos = np.random.choice(range(resto.shape[0]), restantes,
p=probabilities, replace=False)
        g_trunc[elegidos] += 1

    return g_trunc.astype(int)
```

Código 8: referente al cálculo de g

3.3.3. Etiquetado de Vecinos

Esta función buscará la etiqueta de clase de los *k* vecinos más cercanos para cada patrón de un conjunto dado (Conjunto A). Si se facilita otro conjunto diferente en la llamada de la función (conjunto B), podrá calcularse los vecinos para cada patrón del conjunto A, con respecto a dicho conjunto B. Si no se establece un valor para k, se utilizará un método para calcular automáticamente un valor.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
inputs_set1	Conjunto de entrada al que se le calcularán los vecinos con respecto a *inputs_set2*.	--
inputs_set2	Conjunto de entrada de donde se calcularán los vecinos de *inputs_set*1.	--
outputs_set2	Matriz con las variables de salida de *inputs_set2*.	--
minority_label	Etiqueta de la clase minoritaria.	--
k(opcional)	Número de vecinos a tener en cuenta (si no se proporciona se calculará automáticamente).	None
minority_label	Etiqueta de la clase minoritaria.	True
selfNeighbour	Si es True, se considerará que un patrón es vecino de sí mismo.	False
Salidas		
Nombre	*Descripción*	
elegidos	Matriz con la etiqueta de clase de los patrones vecinos. • Filas: Una fila por cada individuo de *inputs_set1.* • Columnas: Una columna por cada vecino calculado (*k*).	

Tabla 9: Función referente al etiquetado de la vecindad

Código:

```
def neighbors_calculator(inputs_set1, inputs_set2, outputs_set2,
minority_label, k=None, self_neighbour=False):

    dist = scp.spatial.distance.cdist(inputs_set1, inputs_set2, 'euclidean')
    nearest_index = np.argsort(dist, axis=1)

    if k is None:
        k = k_calculator(outputs_set2[nearest_index], minority_label)

    elegidos = nearest_index[:, 1 * (not self_neighbour):k + 1 * (not
self_neighbour)]
    return outputs_set2[elegidos], k
```

Código 9: Referente al etiquetado de la vecindad

3.3.4. Cálculo de vecindad

Esta función calcula los índices y si es necesario las distancias de los patrones más cercanos para cada patrón de un conjunto dado (Conjunto A). Si se facilita otro conjunto diferente en la llamada de la función (conjunto B), podrá calcularse los vecinos para cada patrón del conjunto A, con respecto a dicho conjunto B.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
inputs_set1	Conjunto de entrada al que se le calcularán los vecinos con respecto a *inputs_set2*.	--
inputs_set2	Conjunto de entrada de donde se calcularán los vecinos de *inputs_Set1*.	--
k	Número de vecinos a tener en cuenta.	5
return_distances	Booleano que determina si se devuelve las distancias o no.	False
selfNeighbour	Si es True, se considerará que un patrón es vecino de sí mismo.	False
Salidas		
Nombre	*Descripción*	
Elegidos (opcional)	Matriz con la etiqueta de clase de los patrones vecinos. • Filas: Una fila por cada individuo *de inputs_set1*. • Columnas: Una columna por cada vecino calculado (*k*).	
dist	Matriz con las distancias de los patrones vecinos. • Filas: Una fila por cada patrón de *inputs_set1.* • Columna: Una columna por cada vecino calculado.	

Tabla 10: Función referente al supervisor de vecindad

Código:

```
def neighbors_index(inputs_set1, inputs_set2, k=5, return_distances=False,
self_neighbour=False):

    dist = scp.spatial.distance.cdist(inputs_set1, inputs_set2, 'euclidean')

    elegidos = np.argsort(dist, axis=1)[:, 1 * (not self_neighbour):k + 1 *
(not self_neighbour)]

    if return_distances:
        dist = np.sort(dist)[:, 1 * (not self_neighbour):(k + 1 * (not
self_neighbour))]
        return dist, elegidos
    return elegidos
```

Código 10: Referente al supervisor de vecindad

3.3.5. Calculador de valor de *k*

Esta función calculará un valor automáticamente para *k* (número de vecinos a tener en cuenta), de modo que todos los patrones de la clase minoritaria puedan tener al menos un vecino de la clase mayoritaria. De este modo, siempre podrá calcularse un ratio de cercanía al borde sin importar la densidad de patrones en la distribución, y el sistema tendrá la capacidad de adaptarse a cualquier base de datos sin necesidad de que el usuario introduzca un valor adecuado en la configuración.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
neighbors	Matriz con la etiqueta de clase de los patrones vecinos. • Filas: Una fila por cada patrón. • Columna: Una columna por cada vecino calculado.	--
minority_label	Etiqueta de la clase minoritaria.	--
Salidas		
Nombre	*Descripción*	
k	Número de vecinos a tener en cuenta.	

Tabla 11: Función referente al cálculo de k

Código:

```
def k_calculator(neighbors, minority_label):

    # Se buscan los índices de los vecinos que son de la clase mayoritaria
    indices = np.where(neighbors != minority_label)

    # Se busca el primer vecino de la clase mayoritaria de cada patrón de
la clase mayoritaria
    index = np.unique(indices[0], return_index=True)[1]

    # El maximo se guarda en k
    k = indices[1][index].max() + 1
    if k < 5:
        k=5
    return k
```

Código 11: Referente al cálculo de k

3.3.6. Eliminación de ruido

Si se decide activarla, se eliminará todo aquél patrón de la clase minoritaria que tenga todos sus *k* vecinos pertenecientes a la clase mayoritaria, ya que serán considerados como ruido. Esto se realiza con el objetivo de perfeccionar más el borde y que quede mejor definido.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
inputs	Matriz con las variables de entrada.	--
outputs	Matriz con las variables de salida.	--
index	Índices de la clase minoritaria.	--
rlist	Vector con los ratios de cercanía al borde sin normalizar.	--
Salidas		
Nombre	*Descripción*	
inputs	Matriz con las variables de entrada sin ruido.	
outputs	Matriz con las variables de salida sin ruido.	
index	Índices de la clase minoritaria sin ruido.	
normalizedrlist	Vector con los ratios de cercanía al borde normalizado tras eliminar el ruido.	

Tabla 12: Función referente a la eliminación del ruido

Código:

```
def noise_remover(inputs, outputs, index, rlist):

    noise_index = np.copy(index)
    noise = rlist == 1

    if sum(np.logical_not(noise)) > 15:
        # índices de patrones de la clase minoritaria para los que todos sus
vecinos son de la clase mayoritaria (Ruido)
        noise_index[index] = noise

        rlist = rlist[~noise]
        normalizedrlist = rlist / rlist.sum()

        return inputs[~noise_index], outputs[~noise_index],
index[~noise_index], normalizedrlist
    normalizedrlist = rlist / rlist.sum()
    return inputs, outputs, index, normalizedrlist
```

Código 12: Referente a la eliminación del ruido

3.3.7. Generador de sintéticos (con *Iterated Greedy)*

Esta función encierra el núcleo más importante del sistema. En ella se generarán los sintéticos y se evolucionarán mediante la metaheurística voraz iterativa.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
inputs	Matriz con las variables de entrada.	--
outputs	Matriz con las variables de salida.	--
index	Índices de la clase minoritaria.	--
minority_label	Etiqueta de la clase minoritaria.	--
normalizedrlist	Vector con los ratios de vecindad normalizado.	--
K	Número de vecinos a tener en cuenta.	--
n	Factor de ponderación entre cercanía al borde o cercanía al primer padre.	--
G	Número de sintéticos a generar.	--
val_inputs	Matriz con las variables de entrada del conjunto de validación.	--
val_outputs	Matriz con las variables de salida del conjunto de validación.	None
val_auc	Si es True, la validación será guiada en función de AUC, si es false en función de G-Mean.	False
return_clf (opcional)	Si es True, devuelve el árbol entrenado durante la validación (default False).	True
graphics	Si es True, sacará una gráfica en cada ejecución.	False
Salidas		
Nombre	*Descripción*	
newdata_inputs	Matriz con las variables de entrada + sintéticos.	
newdata_outputs	Matriz con las variables de salida + sintéticos.	
mejor_clf (opcional)	El árbol ya entrenado de la mejor validación alcanzada.	

Tabla 13: Función referente a la generación de sintéticos

Código:

```
def generate_synthetics_ig(inputs, outputs, index, minority_label,
normalizedrlist, k, n, G, val_inputs, val_outputs,
                           test_inputs, test_outputs, val_auc=False,
return_clf=True, graphics=False):

    minority_inputs = inputs[index]
    distancias, indices = neighbors_index(minority_inputs, minority_inputs,
k, return_distances=True,
                                          self_neighbour=False)


    #inicializaciones
    synthetics_inputs = np.empty((0, inputs.shape[1]))
```

```
    synthetics_outputs = (np.ones(G, dtype='int') * minority_label)
    newdata_outputs = np.array(np.append(outputs, synthetics_outputs))
    destroyed = int(round(G * 0.15))
    neighborhood_outputs = np.append(outputs, synthetics_outputs[: G -
destroyed], axis=0)
    contador = 0
    mejor_validacion = -1
    mejor_clf = None
    val_array = 0
    test_array = 0

    while contador < 100 and normalizedrlist.sum():

        g = g_calculator(normalizedrlist, G - synthetics_inputs.shape[0])

        # Construcción
        for i in np.nonzero(g)[0]:  # un ciclo por cada individuo
            friends = better_choice(distancias[i], indices[i],
normalizedrlist[indices[i]], g[i], n=n)
            synthetics = minority_inputs[i] + (minority_inputs[friends] -
minority_inputs[i]) * np.random.rand(1)
            synthetics_inputs = np.append(synthetics_inputs, synthetics,
axis=0)

        newdata_inputs = np.array(np.concatenate((inputs,
synthetics_inputs), axis=0))

        validacion, clf = validation(newdata_inputs, newdata_outputs,
val_inputs, val_outputs, val_auc=val_auc,
                                     return_clf=True)


        if validacion > mejor_validacion:
            mejor_validacion = validacion
            mejor_synthetics = np.copy(synthetics_inputs)
            mejor_clf = clf
            mejor_newinputs = np.copy(newdata_inputs)
            contador = 0

        else:
            validacion, clf = validation(mejor_newinputs, newdata_outputs,
val_inputs, val_outputs, val_auc=val_auc,
                                         clf=mejor_clf, return_clf=True)
            contador += 1
        testeo = validation(mejor_newinputs, newdata_outputs, test_inputs,
test_outputs, clf=mejor_clf, val_auc=val_auc)

        val_array = np.hstack((val_array, validacion))
        test_array = np.hstack((test_array, testeo))

        # Destrucción
        removed = np.random.choice(np.arange(G), destroyed, replace=False)
        synthetics_inputs = np.delete(mejor_synthetics, removed, 0)

        neighborhood_inputs = np.append(inputs, synthetics_inputs, axis=0)

        normalizedrlist, k = ratio_borde(minority_inputs,
neighborhood_inputs, neighborhood_outputs, minority_label=True)

    if graphics:
        paint(val_array[1:], test_array[1:])

    if return_clf:
        return mejor_newinputs, newdata_outputs, mejor_clf
    return mejor_newinputs, newdata_outputs
```

Código 13: Referente a la generación de sintéticos

3.3.8. Generador de sintéticos (sin *Iterated Greedy*)

Esta función es una versión del generador de sintéticos el apartado 3.3.7, pero sin aplicarle una optimización mediante *Iterated Greedy.*

Entradas		
Nombre	*Descripción*	*Valor por defecto*
inputs	Matriz con las variables de entrada.	--
outputs	Matriz con las variables de salida.	--
minority_label	Etiqueta de la clase minoritaria.	--
normalizedrlist	Vector con los ratios de vecindad normalizado.	--
K	Número de vecinos a tener en cuenta.	--
n	Factor de ponderación entre cercanía al borde o cercanía al primer padre.	--
G	Número de sintéticos a generar.	--
Salidas		
Nombre	*Descripción*	
newdata_inputs	Matriz con las variables de entrada + sintéticos.	
newdata_outputs	Matriz con las variables de salida + sintéticos.	

Tabla 14: Función referente a la generación de sintéticos

Código:

```
def generate_synthetics(inputs, outputs, minority_label, k, n, G):


    minority_index = class_index(outputs, minority_label)  # índices de la
clase minoritaria
    minority_inputs = inputs[minority_index]
    normalizedrlist, k = ratio_borde(minority_inputs, inputs, outputs,
minority_label=minority_label, k=k,
                                     return_rlist=False)  # ratio de
cercanía normalizado

    distancias, indices = neighbors_index(minority_inputs, minority_inputs,
k, return_distances=True,
                                          self_neighbour=False)

    # inicializaciones
    synthetics_inputs = np.empty((0, inputs.shape[1]))
    synthetics_outputs = (np.ones(G, dtype='int') * minority_label)
    newdata_outputs = np.array(np.append(outputs, synthetics_outputs))

    g = g_calculator(normalizedrlist, G)

    for i in np.nonzero(g)[0]:  # un ciclo por cada individuo
        friends = better_choice(distancias[i], indices[i],
normalizedrlist[indices[i]], g[i], n=n)
        synthetics = minority_inputs[i] + (minority_inputs[friends] -
minority_inputs[i]) * np.random.rand(1)
        synthetics_inputs = np.append(synthetics_inputs, synthetics, axis=0)
```

```
    newdata_inputs = np.array(np.concatenate((inputs, synthetics_inputs),
axis=0))

    return newdata_inputs, newdata_outputs
```

Código 14: Referente a la generación de sintéticos

3.3.9. Mejor segundo padre

Este método se encargará de elegir los mejores vecinos de cada patrón con el que generar los sintéticos que le corresponda, en lugar de utilizar un método puramente aleatorio.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
distancias	Distancias de los *k* vecinos.	--
indices	Índices de los vecinos (para devolver aquellos más cercanos).	--
normalizedrlist	Vector normalizado con los ratios de cercanía al borde.	--
n	Factor de ponderación entre cercanía al borde o cercanía al primer padre.	0.5
g	Número de segundos padres a elegir.	---
Salidas		
Nombre	*Descripción*	
elegidos	Índices de los segundos padres elegidos.	

Tabla 15: Función referente a la elección del mejor vecino

Código:

```
def better_choice(distancias, indices, normalizedrlist, g, n=0.5):

    factor1 = normalizedrlist / normalizedrlist.sum()
    vector2 = 1 / distancias
    factor2 = vector2 / vector2.sum()

    importancia = (n * factor1) + ((1 - n) * factor2)

    probabilities = importancia / importancia.sum()
    elegidos = np.random.choice(indices, g, p=probabilities)

    return elegidos
```

Código 15: Referente a la elección del mejor vecino

3.3.10. Validación

Se encarga de realizar el proceso de validación para guiar la evolución de la metaheurística. La validación puede guiarse en función del AUC o del G-mean.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
train_inputs	Matriz con las variables de entrada de entrenamiento.	--
train_outputs	Matriz con las variables de salida de entrenamiento.	--
test_inputs	Matriz con las variables de entrada de test.	--
test_outputs	Matriz con las variables de salida de test.	--
clf	Árbol ya entrenado (Si no se proporciona se creará y entrenará un árbol en la ejecución de la función).	None
val_auc	Si es True, la validación será guiada en función de AUC, si es false en función de G-Mean.	False
return_clf	Si es True, también devuelve el árbol entrenado.	False
Salidas		
Nombre	*Descripción*	
normalizedrlist	Vector normalizado con los ratios de cercanía al borde.	
rlist(opcional)	Vector con los ratios de cercanía al borde sin normalizar.	

Tabla 16: Función referente al proceso de validación

Código:

```
def validation(train_inputs, train_outputs, test_inputs, test_outputs,
clf=None, val_auc=False, return_clf=False):

    if clf is None:
        clf = decision_tree(train_inputs, train_outputs)

    predicted__test = clf.predict(test_inputs)

    if val_auc:
        score = roc_auc_score(test_outputs, predicted__test)
    else:
        confusion_mtx = confusion_matrix(test_outputs, predicted__test)
        tp = confusion_mtx[0][0]
        fp = confusion_mtx[1][0]
        fn = confusion_mtx[0][1]
        tn = confusion_mtx[1][1]

        score = np.mean(np.sqrt((float(tp) / float(tp + fn)) * (float(tn) /
float(tn + fp))))

    if return_clf:
        return score, clf
    return score
```

Código 16: Referente al proceso de validación

3.3.11. Pseudocódigo

Para facilitar la comprensión del algoritmo se ha incluido un pseudocódigo.

Algoritmo de over-sampling

Entrada*:*

Train inputs (Variables de entrada del conjunto de entrenamiento)
Train outputs (Etiqueta a predecir entrada del conjunto de entrenamiento)
Validation inputs (Variables de entrada del conjunto de validación)
Validation outputs (Etiqueta a predecir entrada del conjunto de validación)

Salida:

NewData inputs (Entradas del conjunto de sintéticos + *Train*)
NewData outputs (Etiquetas del conjunto de sintéticos + *Train*)
Árbol entrenado (Opcional) (Entrenado durante la validación)

1. Repetir
2. Para Cada i perteneciente a m ($i \in m$)
3. | Calcular su ratio del cercanía al borde (número de vecinos mayoritarios / k)
4. | Normalizar su ratio del cercanía al borde ($\hat{r}_i$)
5. Fin para
6. Mientras contador **sea menor que** 100
7. **Para** Cada i perteneciente a m ($i \in m$)
8. | Asignarle un numero de sintéticos a generar (g_i) en función de $\hat{r}_i$
9. | Redistribuir la asignación de sintéticos (g_i) para eliminar los decimales ($\bar{g}_i$)
10. Fin para
11. **Para** cada i con $\bar{g}_i$ diferente de 0
12. | Seleccionar $\bar{g}_i$ vecinos mediante un modelo probabilístico basado en ε_j ($j \in k$)
13. | Generar un sintético S_{ij} por cada $\bar{g}_i$ utilizando la interpolación de SMOTE.
14. Fin para
15. Unir sintéticos S y *Validation*
16. Realizar validación con la unión de S y *Validation*
17. **Si** validación **es mejor que** la mejor validación encontrada antes
18. | Almacenar la mejor validación
19. | Almacenar árbol entrenado en la validación
20. Sino
21. | Contador se incrementa en 1
22. Fin si
23. Destruir un 15% de S (S_b, sabiendo que $S_b \subseteq S$)
24. Unir *Train* y S_b
25. Recalcular $\hat{r}_i$ para todo m
26. Fin

3.4. Tratamiento de resultados

Este módulo es el encargado tanto de calcular los resultados de la clasificación final, como de ordenarlos, mostrarlos y almacenarlos. Las funcionalidades que cumple este módulo se definen a continuación.

3.4.1. Recolección de resultados

Esta función almacena y recopila en un conjunto de datos, todos los resultados obtenidos en las sucesivas iteraciones del algoritmo.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
train_inputs	Matriz con las variables de entrada de entrenamiento.	--
train_outputs	Matriz con las variables de salida de entrenamiento.	--
test_inputs	Matriz con las variables de entrada de test.	--
test_outputs	Matriz con las variables de salida de test.	--
clf	Árbol ya entrenado (Si no se proporciona se creará y entrenará un árbol en la ejecución de la función).	None
datos	Matriz con las medidas de precisión de ejecuciones anteriores.	--
Salidas		
Nombre	*Descripción*	
normalizedrlist	Vector normalizado con los ratios de cercanía al borde.	
datos	Matriz con las medidas de precisión de cada ejecución. • Filas: Una fila por cada medida de precisión. • Columnas: Una columna por cada ejecución.	

Tabla 17: Función referente a la recolección de resultados

Código:

```
def tester(train_inputs, train_outputs, test_inputs, test_outputs, datos,
clf=None):

    oa, precision, recall, f1_measure, g_mean, auc = evaluator(train_inputs,
train_outputs, test_inputs, test_outputs,
                                                              clf)
    one_loop_dates = np.hstack((oa, precision, recall, f1_measure, g_mean,
auc))

    return np.c_[datos, one_loop_dates]
```

Código 17: Referente a la recolección de resultados

3.4.2. Evaluador

Es la función encargada de calcular algunos medidores de precisión de la clasificación.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
train_inputs	Matriz con las variables de entrada de entrenamiento.	--
train_outputs	Matriz con las variables de salida de entrenamiento.	--
test_inputs	Matriz con las variables de entrada de test.	--
test_outputs	Matriz con las variables de salida de test.	--
clf	Árbol ya entrenado (Si no se proporciona se creará y entrenará un árbol en la ejecución de la función).	None
Salidas		
Nombre	*Descripción*	
Oa	Medida de precisión.	
Precision	Medida de precisión.	
Recall	Medida de precisión.	
F1_measure	Medida de precisión.	
G_mean	Medida de precisión.	
AUC	Medida de precisión.	

Tabla 18: Función referente al cálculo de evaluadores

Código:

```
def evaluator(train_inputs, train_outputs, test_inputs, test_outputs,
clf=None):

    if clf is None:
        clf = decision_tree(train_inputs, train_outputs)

    predicted__test = clf.predict(test_inputs)

    oa = clf.score(test_inputs, test_outputs)
    precision, recall, f1_measure, support = score(test_outputs,
predicted__test, beta=1)
    g_mean = np.sqrt(recall[0] * recall[1])

    # fpr, tpr, thresholds = roc_curve(test_outputs, predicted__test)
    # auc2 = metrics.auc(fpr, tpr)
    auc = roc_auc_score(test_outputs, predicted__test)

    return oa, precision, recall, f1_measure, g_mean, auc
```

Código 18: Referente al cálculo de evaluadores

3.4.3. Gestión de resultados

Imprime en pantalla o sobre un fichero la media de las medidas de precisión alcanzadas en cada ejecución.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
datos	Matriz con las variables de entrada de entrenamiento.	--
aux_dates1	Matriz con las medidas de precisión de ejecuciones anteriores sobre un conjunto auxiliar.	--
aux_dates2	Matriz con las medidas de precisión de ejecuciones anteriores sobre un conjunto auxiliar.	--
aux_name1	Nombre de la columna donde aparecerán las medidas del conjunto *aux_dates1*.	'Aux1_column'
aux_name2	Nombre de la columna donde aparecerán las medidas del conjunto *aux_dates2*.	'Aux2_column'
to_screen	Si es True, imprime las medidas por pantalla.	True
to_csv	Si es True, creara un fichero con las medidas calculadas.	True
file_name	Nombre del fichero que será creado con las medidas. Solo tendrá sentido si *to_csv*=True.	'Results'

Tabla 19: Función referente a la visualización de los resultados

Código:

```
def imprimir_resultado(datos, second_dates=None, aux_dates=None,
second_name=['Second_column'], aux_name=['Aux_column
'],
                       to_screen=True, to_csv=True, file_name='Results'):

    datos = datos.reshape(-1, 1)
    cabecera = ['ANEIGSYN']
    if second_dates is not None:
        second_dates = second_dates.reshape(-1, 1)
        cabecera += second_name
        datos = np.c_[datos, second_dates]

    if aux_dates is not None:
        aux_dates = aux_dates.reshape(-1, 1)
        cabecera += aux_name
        datos = np.c_[datos, aux_dates]

    medidas = np.array(
        ['OA', 'Precision Class1', 'Precision Class0', 'Recall Class1',
'Recall Class0', 'F1_measure Class1',
         'F1_measure Class0', 'G_mean', 'AUC'])
    df = pd.DataFrame(datos, columns=cabecera, index=medidas)

    if to_screen:
        print df
    if to_csv:
        file_name = ''.join(file_name)
        file_name = file_name.replace(".csv", "")
        file_name += '.csv'
        df.to_csv(file_name, header=True, index=True, sep=',')
```

Código 19: Referente a la visualización de los resultados

3.4.4. Representar gráfica

Es una función que puede representar una gráfica de varios evaluadores diferentes para el mismo conjunto, o imprimir el mismo evaluador para dos conjuntos diferentes, como por ejemplo Validation y Test, para ver si se produce sobre-entrenamiento.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
val_array	Vector con los mejores resultados conseguidos en el conjunto de validación.	--
test_array	Valores de test conseguidos utilizando el mismo árbol y el mismo conjunto de entrenamiento que en validación, pero en esta ocasión utilizando el conjunto de test.	--
to_screen	Si es True, se mostrará una gráfica por pantalla.	True
to_csv	Si es True, se almacenará un archivo con los vectores de la gráfica.	False
file_name	Nombre para poder almacenar el archivo.	'Graphics'

Tabla 20: Función referente a la creación de gráficas

Código:

```
def paint(val_array, test_array, to_screen=True, to_csv=False,
file_name='grafica'):

    if to_screen:
        generaciones = test_array.shape[0]
        first_value = np.ones(generaciones) * test_array[1]
        t = np.arange(0, generaciones, 1)
        title = 'val= Green, test = Red \ntest with first synthetics = blue,
'

        pj = plt.plot(t, val_array, 'g-', t, test_array, 'r-', first_value,
'b--')


        plt.title(title)
        plt.show(pj)

    if to_csv:
        val_array = val_array.reshape(-1, 1)
        cabecera = ['val']
        test_array = test_array.reshape(-1, 1)
        cabecera += ['test']
        val_array = np.c_[val_array, test_array]

        df = pd.DataFrame(val_array, columns=cabecera)
        file_name = ''.join(file_name)
        file_name = file_name.replace(".csv", "")
        file_name += '.csv'
        df.to_csv(file_name, header=True, index=True, sep=',')
```

Código 20: referente a la creación de gráficas

3.4.5. Árbol de decisión

Es la función que contiene el árbol de decisión utilizado como clasificador. De este modo, tendremos centralizado el árbol que se utiliza en varios puntos como la validación o el test final, y solo requerirá de una configuración.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
inputs	Matriz con las variables de entrada de entrenamiento.	--
outputs	Matriz con las variables de salida de entrenamiento.	--
Salidas		
Nombre	*Descripción*	
clf	Árbol generado y entrenado mediante *inputs* y *outputs.*	

Tabla 21: Función referente al árbol de decisión

Código:

```
def decision_tree(inputs, outputs):

    clf = DecisionTreeClassifier(class_weight=None)
    clf.fit(inputs, outputs)

    return clf
```

Código 21: Referente al árbol de decisión

3.5. Funciones especiales

En este apartado se definirán algunas funciones que no pertenecen a ninguno de los módulos explicados anteriormente, pero que son vitales para el funcionamiento del sistema y para un correcto flujo de datos.

3.5.1. Configuración

Función destinada a recoger los parámetros de configuración más importantes, y a acercar su modificación al usuario para que resulte más fácil realizar un ajuste óptimo.

Salidas		
Nombre	*Descripción*	*Valor por defecto*
test_size	Tamaño porcentual del conjunto de test.	0.5
train_size	Tamaño porcentual del conjunto de entrenamiento.	0.3
val_size	Tamaño porcentual del conjunto de validación.	1 - (test_size + train_size)
runs	Matriz con las variables de salida de test.	100
beta	Especifica que grado de desbalanceo será cubierto con la generación de sintéticos. (β = 1 significa totalmente balanceado).	1
k	Número de vecinos a tener en cuenta.	None
beta	Árbol ya entrenado (Si no se proporciona se creará y entrenará un árbol en la ejecución de la función).	1
n	Factor de ponderación entre cercanía al borde o cercanía al primer padre.	0.7
val_auc	Si es True, la validación será guiada en función de AUC, si es false en función de G-Mean.	False
imbalanced_degree	Variable que se utilizará para encontrar las clases minoritarias en tiempo de ejecución.	None
pintar_graficas	Si es True, sacará una gráfica en cada ejecución.	False
rmv_dup	Si es True, Se activará la función que elimina los duplicados.	False
rmv_noise	Si es True, Se activará la función que elimina el ruido.	False

Tabla 22: Función referente a la configuración del algoritmo

Código:

```
def configuration():

    test_size = 0.5  # Tamaño porcentual del conjunto de test
    train_size = 0.3  # Tamaño porcentual del conjunto de entrenamiento
    val_size = 1 - (test_size + train_size)  # Tamaño porcentual del
conjunto de validación
    runs = 100
    beta = 1  # βϵ[0, 1] specify the balance level after generation of the
synthetic data (β = 1 means a fully balanced)
    k = None  # Number of neighbors to calculate the border
    n = 0.7
    val_auc = False  # Si es True, la valiadación será guiada en funcion de
AUC, si es false en funcion de G-Mean
    imbalanced_degree = None  # Variable que se utilizará para encontrar las
clases minoritarias en tiempo de ejecución
    pintar_graficas = True
    rmv_dup = False
    rmv_noise = False

    return test_size, train_size, val_size, runs, beta, k, n, val_auc,
imbalanced_degree, pintar_graficas, rmv_dup, \
           rmv_noise
```

Código 22: Referente a la configuración del algoritmo

3.5.2. Principal

Función destinada a controlar el flujo de datos por el sistema. Es la encargada de recoger las salidas de algunas funciones y realizar la llamada de otras cuando sea necesario.

Entradas		
Nombre	*Descripción*	*Valor por defecto*
fichero	Ruta de la base de datos incluyendo el nombre del archivo.	--
semillas	Vector de semillas para controlar la aleatoriedad del sistema.	--
test_size	Tamaño porcentual del conjunto de test.	0.5
train_size	Tamaño porcentual del conjunto de entrenamiento.	0.3
val_size	Tamaño porcentual del conjunto de validación.	1 - (test_size + train_size)
runs	Matriz con las variables de salida de test.	100
beta	Especifica que grado de desbalanceo será cubierto con la generación de sintéticos. (β = 1 significa totalmente balanceado).	1
k	Número de vecinos a tener en cuenta.	None
beta	Árbol ya entrenado (Si no se proporciona se creará y entrenará un árbol en la ejecución de la función).	None
n	Factor de ponderación entre cercanía al borde o cercanía al primer padre.	0.7
val_auc	Si es True, la validación será guiada en función de AUC, si es false en función de G-Mean.	False
imbalanced_degree	Variable que se utilizará para encontrar las clases minoritarias en tiempo de ejecución.	None
graphics	Si es True, sacará una gráfica en cada ejecución.	False
rmv_dup	Si es True, Se activará la función que elimina los duplicados.	False
rmv_noise	Si es True, Se activará la función que elimina el ruido.	False

Tabla 23: Función referente a la función principal

Código:

```
def principal(fichero, semillas, test_size, train_size, val_size, beta=1,
k_original=5, n=0.5, val_auc=False,
              imbalanced_degree=None, graphics=False, rmv_dup=False,
rmv_noise=False):

    # inicialización de los datos
    runs = semillas.shape[0]
    inputs, outputs = lectura_datos(fichero, rmv_dup)
    outputs = bi_clase(outputs, imbalanced_degree)

    # Estructura de datos donde se almacenará las medidas de precision
    datos = np.empty((9, 0))
    for loop in xrange(0, runs, 1):
        k = k_original
        semilla = semillas[loop]
        print 'loop:', loop, '    seed:', semilla
        np.random.seed(semilla)
```

```
        train_inputs, train_outputs, test_inputs, test_outputs, val_inputs,
val_outputs = particion_datos(inputs, outputs, test_size, train_size,
val_size)

         # indices de la clase minoritaria
        minority_index = class_index(train_outputs, True)
        # column 0 (class label) column 1 (class size)
        clases = conteo_clases(train_outputs)
        # Etiqueta de la clase minoritaria
        minority_label = clases[0][0]
         # entradas de la clase minoritaria
        minority_inputs = train_inputs[minority_index]

        # Se elimina ruido
        if rmv_noise and (clases[0][1] / float(clases[1][1])) > 0.1:
            # ratio de cercania normalizado
            normalizedrlist, k, rlist = ratio_borde(minority_inputs,
train_inputs, train_outputs, True, k, return_rlist=True)
            train_inputs, train_outputs, minority_index, normalizedrlist =
noise_remover(train_inputs, train_outputs, minority_index, rlist)

            # column 0 (class label) column 1 (class size)
            clases = conteo_clases(train_outputs)

        else:  # No se elimina ruido
            # ratio de cercania normalizado
            normalizedrlist, k = ratio_borde(minority_inputs, train_inputs,
train_outputs, minority_label=True, k=k, return_rlist=False)



        """variables"""
        # number of synthetic data examples to generate for the minority
class
        G = np.ceil(((clases[1:, 1].sum() - clases[0][1]) * beta) -
1).astype(int)


        newdata_inputs, newdata_outputs, mejor_clf =
generate_synthetics_ig(train_inputs, train_outputs, minority_index,
minority_label, normalizedrlist, k, n, G, val_inputs, val_outputs,
test_inputs, test_outputs, val_auc, return_clf=True, graphics=graphics)

        #
////////////////////////////////////////////////////////////////////////////
////////////////////////////////
        # Calculo de precisiones
        #
////////////////////////////////////////////////////////////////////////////
////////////////////////////////

        datos = tester(newdata_inputs, newdata_outputs, test_inputs,
test_outputs, datos, clf=mejor_clf)

    imprimir_resultado(datos.mean(axis=1), to_screen=True, to_csv=True)

    return True
```

Código 23: Referente a la función principal

3.5.3. Main

La función *main* es un ingrediente que debe encontrarse en todos los programas escritos en *Python*. En nuestro caso se encarga de recoger la configuración establecida por el usuario y ceder el control a la función principal (3.5.2 ***Principal***).

Código:

```
if __name__ == "__main__":

    ejecucion = "Para ejecutar: ./ 'nombre fichero' 'Ruta de una BBDD en
.CSV' 'Semilla para aleatoriedad'"
    try:
        fichero = sys.argv[1]
    except IndexError:
        print "Falta la ruta de una BBDD\n", ejecucion
        sys.exit("Parando ejecución")

    try:
        semilla = int(sys.argv[2])
    except IndexError:
        semilla = np.random.randint(low=10000000, high=99999999)
        print 'semilla aleatoria:', semilla

    test_size, train_size, val_size, runs, beta, k, n, val_auc,
imbalanced_degree, rmv_dup, rmv_noise = configuration()

    if k < 2 and k != None:
        print "k vale", k, "pero deberia tener un valor igual o superior a
2"
        sys.exit("Valor erroneo para k")

    if runs < 2:
        print "runs vale", runs, "pero deberia tener un valor igual o
superior a 2"
        sys.exit("Valor erroneo para runs")

    if not (0.1 <= beta <= 1):
        print "beta vale", beta, "pero deberia tener un valor entre 0 y 1"
        sys.exit("Valor erroneo para beta")

    if not (isinstance(val_auc, (int, bool))):
        print "val_auc vale", val_auc, "pero deberia tener un valor igual a
True o False"
        sys.exit("Valor erroneo para val_auc")

    if not (0.1 < imbalanced_degree <= 0.5) and imbalanced_degree is not
None:
        print "imbalanced_degree vale", imbalanced_degree, "pero deberia
tener un valor entre 0 y 0.5"
        sys.exit("Valor erroneo para imbalanced_degree")

    np.random.seed(semilla)
    semillas = np.random.choice(np.asarray(list(map("".join,
permutations('12345678'))), dtype=int), runs)

    principal(fichero, semillas, test_size, train_size, val_size, beta, k,
n, val_auc=val_auc,
              imbalanced_degree=imbalanced_degree, rmv_dup=rmv_dup,
rmv_noise=rmv_noise)
```

Código 24: Referente a la función principal

3.6. Librerías necesarias

En el Código 25, se muestra cuáles son las librerías que se deben incluir para que el sistema pueda funcionar correctamente.

```
import sys
import numpy as np
import pandas as pd
from scipy.stats import itemfreq
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, roc_auc_score,
precision_recall_fscore_support as score
from sklearn.tree import DecisionTreeClassifier
from sklearn import preprocessing
import matplotlib.pyplot as plt
import scipy as scp
from itertools import permutations
```

Código 25: librerías incluidas

Capítulo 4:
Ejecución de la aplicación

Para ejecutar la aplicación abriremos una terminal y nos desplazaremos al directorio donde hemos almacenado el script. Una vez allí ejecutaremos el siguiente comando:

./ANEIGSYN.py RutaBaseDeDatos Semilla

Donde "Ruta" es la ruta completo del directorio donde se encuentran la base de datos que quieres cargar como por ejemplo: **basesDatos/csv/pimadiabetes.csv**

Y donde semilla es un número entero cualquiera que servirá como semilla aleatoria para controlar la aleatoriedad de las ejecuciones del algoritmo. Se recomienda utilizar una permutación de los números del 1 al 8, por ejemplo: **25631487.**

Ejemplo de comando de ejecución completo:

./ANEIGSYN.py basesDatos/csv/vehicle.csv 65231478