

**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE  
TELECOMUNICACIÓN**

**GRADO EN INGENIERÍA BIOMÉDICA**

# **PREDICCIÓN DEL SUEÑO CON *WEARABLE***

PROYECTO PROCESADO MASIVO DE DATOS

AUTORES: CARMEN PLAZA SECO, ROBERTO HOLGADO CUADRADO Y NOEMI GONZÁLEZ LOIS

TUTORES: SERGIO MUÑOZ Y JOSÉ LUIS ROJO

## RESUMEN

Hoy en día los dispositivos *wearables* se están convirtiendo en un producto tecnológico indispensable para muchos usuarios. Estos *gadgets* pueden medir una gran diversidad de parámetros como la actividad física habitual, la frecuencia cardiaca y la duración e incluso calidad del sueño.

El objetivo principal del proyecto es estimar el sueño y sus distintas etapas a partir de parámetros fisiológicos básicos como la frecuencia cardiaca o la movilidad del sujeto. Para ello, se plantean los siguientes objetivos específicos: procesado de la base de datos y extracción de las características, diseño de distintos modelos de clasificación y aplicación de herramientas de Big Data. Como fuente de información, se han utilizado datos de PhysioNet de 31 usuarios que llevaron un dispositivo *wearable Apple Watch* y sobre los que se realizó un estudio polisomnográfico (PSG).

Tras visualizar las distintas variables de cada uno de los sujetos, se ha realizado un preprocesamiento de la base de datos. Este preprocesado consiste en, además de la limpieza de valores inconsistentes, unificar y estructurar el conjunto total de datos en una única base de datos. Una vez unificada la base de datos, se ha aplicado la técnica de ‘ventana deslizante’ para la extracción de características como la media, valor máximo y valor mínimo. Estos estadísticos, se han utilizado como datos de entrada de distintos clasificadores para abordar los problemas de clasificación binaria y multiclase que se plantean en el proyecto. Por último, es de interés destacar el uso de estrategias de balanceo de clases sobre los datos de entrada con el propósito de mejorar el rendimiento de algunos clasificadores utilizados.

Respecto a los esquemas de clasificación, los algoritmos considerados son máquinas de vectores soporte lineal y no lineal, árboles de clasificación y perceptrón multicapa. Las tasas de acierto obtenidas en los diferentes modelos reflejan ciertas dificultades para abordar el problema de clasificación con el conjunto de datos considerado. A pesar de ello, tanto para el problema binario como para el multiclase, los mejores resultados se han obtenido con el algoritmo de máquinas de vectores soporte lineal.

Debido al gran volumen de datos y el gran número de operaciones que se manejan en el proyecto, se ha decidido realizar distintos métodos de computación paralela para mejorar el tiempo y la capacidad computacional haciendo uso de distintos núcleos.

Como líneas futuras de investigación, se propone obtener características a partir de la variable *steps* no utilizada en este proyecto, aplicar más técnicas de computación paralela que permitan utilizar un mayor número de muestras para entrenar los clasificadores, y utilizar métodos de clasificación no supervisada con los datos disponibles anteriores al estudio polisomnográfico.

# ÍNDICE

## Resumen

- 1. Introducción**
  - 1.1. Antecedentes**
  - 1.2. Objetivos**
- 2. Metodología y visualización**
  - 2.1. Obtención de los datos**
  - 2.2. Visualización**
  - 2.3. Preprocesamiento**
  - 2.4. Extracción de características**
  - 2.5. Subconjuntos de Train y Test**
  - 2.6. Modelos de clasificación**
    - 2.6.1. Máquinas de vectores soporte
    - 2.6.2. Árboles de clasificación
    - 2.6.3. Perceptrón multicapa
    - 2.6.4. Métodos de evaluación
  - 2.7. Paralelizar**
    - 2.7.1. Paquete *multiprocessing*
    - 2.7.2. Paquete *numba*
    - 2.7.3. Cython
    - 2.7.4. Google Colaboratory
- 3. Resultados**
  - 3.1. Problema clasificación binaria con datos balanceados**
    - 3.1.1. SVM lineal
    - 3.1.2. SVM no lineal
    - 3.1.3. Árboles de decisión
    - 3.1.4. Perceptrón multicapa

### **3.2. Problema clasificación multiclase con datos balanceados**

- 3.2.1. SVM lineal
- 3.2.2. SVM no lineal
- 3.2.3. Árboles de decisión
- 3.2.4. Perceptrón multicapa

### **3.3. Problema clasificación binaria con datos sin balancear**

- 3.3.1. SVM lineal
- 3.3.2. SVM no lineal
- 3.3.3. Árboles de decisión
- 3.3.4. Perceptrón multicapa

## **4. Conclusiones y líneas futuras**

## **Bibliografía**

## **Anexos**

# Capítulo 1

En este primer capítulo se describen de forma general las líneas principales de este proyecto. En primer lugar, se realiza una breve introducción clínica. A continuación, se describen los objetivos y conclusiones de un estudio previo de referencia. Por último, se describen los objetivos del Proyecto Coordinado Masivo de Datos que se ha llevado a cabo.

## Introducción

### 1.1. CLÍNICA DEL SUEÑO

El sueño no es un estado uniforme, sino que existe un comportamiento cíclico de los distintos patrones del sueño. Las fases del sueño se dividen de manera acordada en dos etapas: una fase REM de movimientos oculares rápidos y una fase NoREM que consta de 4 fases, como se puede observar en la Tabla 1.

Los **estudios polisomnográficos** se realizan en unidades del sueño y consisten en la recogida de diferentes señales fisiológicas de un sujeto durante un periodo de unas 8h, normalmente por la noche. A partir de estos indicadores, un experto clínico identifica cada una de las etapas del sueño. De esta forma, se consigue elaborar una representación gráfica de la sucesión de las etapas del sueño a lo largo de la noche, conocido como **hipnograma**. La distribución normal de las fases de sueño en un adulto joven varían considerablemente según la edad y las necesidades individuales, aunque de forma general suele ser:

- 5% de fase I
- 25% de fase II
- 45% de fases III y IV
- 25% de sueño REM

FASES PREVIAS AL SUEÑO	FASES DEL SUEÑO				
	NO REM				
	FASE I	FASE II	FASE III	FASE IV	

Activo	Relajado con ojos cerrados	Somnoliento	Sueño ligero	Profundo	Muy profundo	REM
--------	----------------------------	-------------	--------------	----------	--------------	-----

**Tabla 1.** Esquema de las diferentes fases del sueño y previas.

## 1.2. ANTECEDENTES

El desarrollo de este proyecto está basado en un trabajo previo que queda recogido en el paper de la referencia [1]. El código desarrollado en ese trabajo previo se encuentra en el repositorio de Github de la referencia [2]. Los objetivos que se llevaron a cabo en este estudio de referencia son:

- **Recopilar datos de aceleración y frecuencia cardíaca en crudo**, sin procesar. Se han extraído del acelerómetro y del PPG (PhotoPlethysmoGram) del *wearable* Apple Watch.
- **Utilizar métodos modernos de clasificación** para distinguir el sueño del despertar. Así, se pudieron determinar las etapas del sueño en comparación con los datos del estudio polisomnográfico (PSG).
- **Evaluar la incorporación de un término "proxy de reloj"** en el rendimiento de los clasificadores. Ese término representa la tendencia circadiana a dormir durante la noche.
- **Generalizar los algoritmos** probando los modelos entrenados para otro conjunto de datos recolectado de forma independiente, un Estudio Multiétnico de Aterosclerosis (MESA).

En este trabajo previo se probaron distintos métodos de clasificación: regresión logística, KNN (K-Nearest Neighbors), random forest y MLP ( Multilayer Perceptron). El mejor rendimiento se logró utilizando este último, las **redes neuronales**, aunque las diferencias entre los clasificadores fueron generalmente pequeñas. Para la clasificación binaria de sueño-vigilia, este método clasificó correctamente en el 90% de los casos. Además, la precisión para diferenciar la vigilia, el sueño NREM y el sueño REM fue aproximadamente del 72%.

## 1.3. OBJETIVOS

Hoy en día, el uso de los dispositivos portátiles, conocidos como *wearables*, es cada vez más común en la población. Generalmente, estos dispositivos están enfocados a controlar el bienestar de los sujetos que los utilizan monitorizando sus parámetros fisiológicos básicos. Entre

las aplicaciones que ofrecen estos dispositivos se encuentra la monitorización del sueño a partir de parámetros fisiológicos. Por ello, el objetivo principal de este proyecto es la **estimación del sueño y sus distintas etapas**, a partir de parámetros fisiológicos básicos como la frecuencia cardíaca o la movilidad del sujeto. Esto, junto con el gran avance de la tecnología en nuestros días, ha llevado a la estimación del sueño mediante técnicas de inteligencia artificial.

Para conseguir el objetivo general anterior, los objetivos específicos de este Proyecto Coordinado Masivo de Datos se desglosan de la siguiente manera:

- **Construir y comparar distintos modelos de clasificación** tanto binarios (sueño ligero o sueño profundo) como multiclase (distintas fases del sueño) que permitan establecer una relación con el estado del sueño para predecir el comportamiento del sueño en otros sujetos.
- **Aplicar técnicas y herramientas de Big Data** para abordar y solucionar este problema de datos masivo.

Durante todo el desarrollo del código del proyecto se ha trabajado con el lenguaje de programación *Python* y con el entorno de documentos de *Jupyter Notebook* y *Google Colaboratory*.

## Capítulo 2

En este capítulo se detallan todos los pasos seguidos durante el desarrollo del proyecto, documentando tanto los resultados positivos como los negativos. De esta manera, se recogen también los problemas que han ido surgiendo en las diferentes etapas y las soluciones que se han aplicado a dichos obstáculos.

## Metodología y visualización

### 2.1. OBTENCIÓN DE LOS DATOS

La base de datos se ha extraído del repositorio de datos de investigación médica **PhysioNet**. En la parte inferior de la página web de la referencia [4], en el apartado '*Download the ZIP file (550.1 MB)*', se pueden descargar todos los datos utilizados en el proyecto. Una vez descargados los datos, para poder ejecutar el código desarrollado en este proyecto es necesario clonar en local el repositorio de *GitHub* de la referencia [3]. Para clonar el repositorio se debe escribir en la terminal el comando '*git clone <url repositorio>*'. El repositorio se clonará en la ubicación del equipo en la que se ejecute el comando.

Tras realizar los pasos anteriores, se cuenta con cuatro carpetas de datos que contienen 31 ficheros de texto cada una de ellas. Las carpetas creadas son las siguientes:

1. **Heart rate:** frecuencia cardíaca obtenida mediante fotopletismografía (PPG) a través del dispositivo *wearable*, con una frecuencia de muestreo aleatoria.
2. **Steps:** número de pasos que da el sujeto. Obtenido a partir del dispositivo *wearable* con una frecuencia de muestreo también aleatoria.
3. **Motion:** aceleración triaxial medida a través del acelerómetro MEMS (*Microelectromechanical Systems*) del dispositivo *wearable*.



4. **Labels:** etiquetas del sueño fijadas por un experto durante 8h aproximadamente de estudio polisomnográfico (PSG). Hay una muestra cada 30s y los posibles valores son: muy despierto=-1, adormilado=0, sueño profundo (nREM)=1,2,3,4 y sueño ligero (REM)=5.

Una vez se han descargado los datos necesarios para el desarrollo del trabajo, se procede a su respectiva visualización.

## 2.2. VISUALIZACIÓN

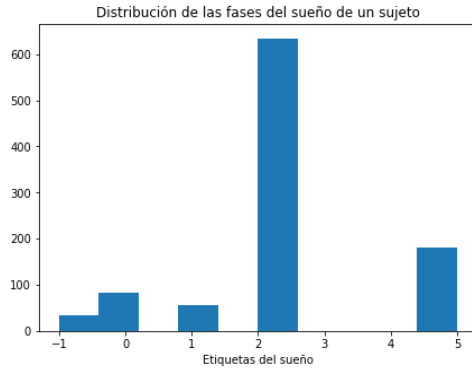
La etapa de visualización consiste en realizar un **análisis exploratorio de los datos** (EDA, del inglés *Exploratory Data Analysis*) y analizar los posibles valores inconsistentes. La visualización de los datos se puede observar ejecutando el archivo 'Visualización.ipynb' del repositorio de GitHub [3]. En la etapa de visualización se ha seguido un procedimiento similar para la mayoría de las variables:

- Visualización de las dimensiones de los datos → `.shape()`
- Extracción de parámetros estadísticos: media, máximo, mínimo, cuartiles, desviación estándar → `.describe()`
- Conteo de los valores Nulos y NaN → `.isnull().sum()`, `.isna().sum()`
- Histograma (sólo en algunas de las variables) → `plt.hist()`
- Representación gráfica en función del tiempo → `plt.plot()`

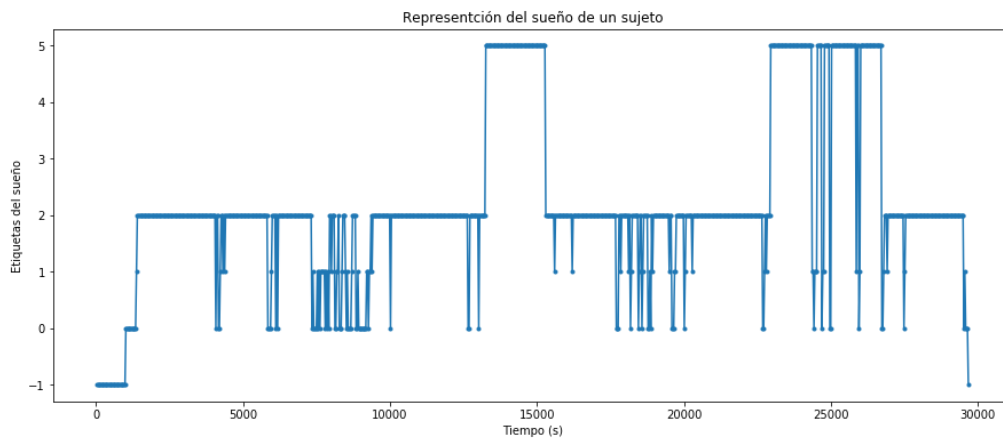
A continuación se muestra la información obtenida de la visualización de cada una de las cuatro variables en el **Sujeto 3**.

### LABELS

Para el sujeto 3, la variable labels tiene unas dimensiones de 989 x 2. La primera columna son las muestras de tiempo (cada 30s) y la segunda es la etiqueta del sueño fijada por el experto. Se reúne información únicamente de la noche en la que se realiza el estudio polisomnográfico ( $t > 0s$ ) y el número de valores Nulos o NaN es 0. De la Figura 1 se puede deducir que la fase del sueño número 4 será probablemente mal clasificada debido a que en este sujeto no hay ninguna muestra y en los demás hay muy pocas. Por este motivo, se aplicará preprocesamiento a dicha variable en etapas posteriores. Por otra parte, en la Figura 2 se puede observar la distribución de las etapas del sueño a lo largo de las 8 h (28.800 s) de grabación del estudio PSG.



**Figura 1.** Histograma de la variable labels del sujeto 3.

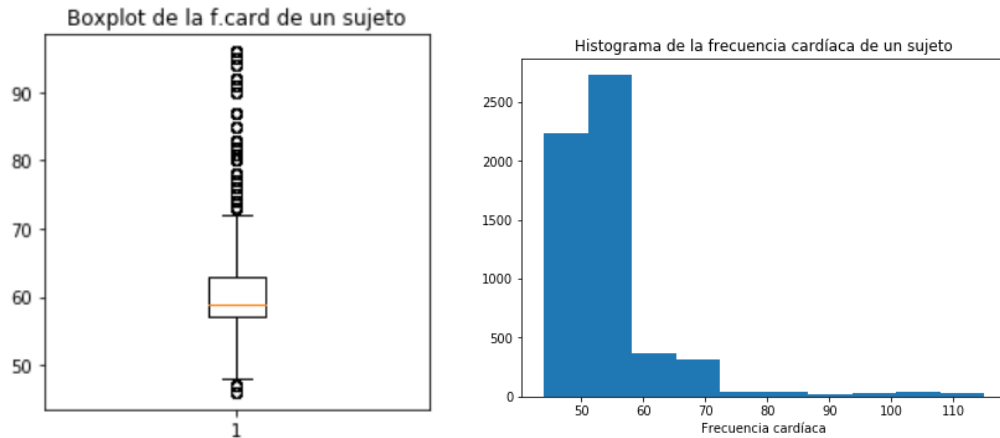


**Figura 2.** Representación a lo largo del tiempo de la variable labels del sujeto 3.

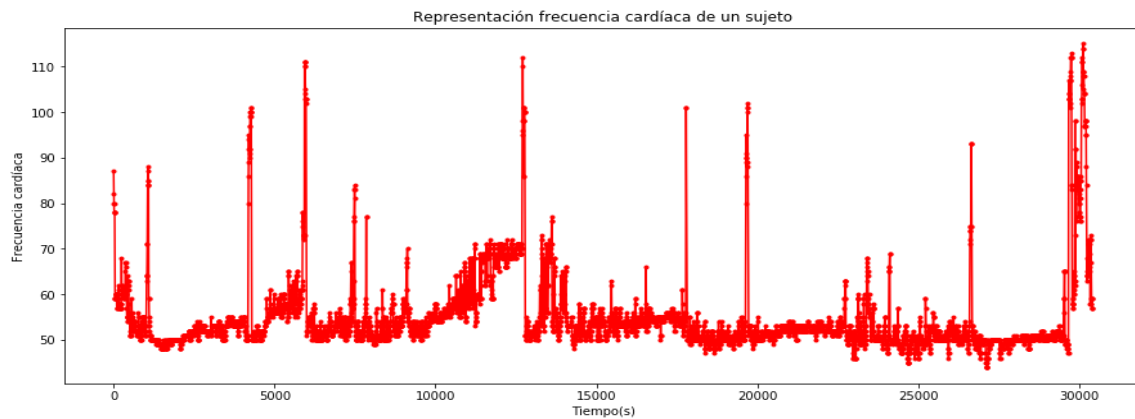
## HEART RATE

En cuanto a la variable heart rate, se ha decidido trabajar sólo con la información de frecuencia cardíaca asociada al estudio PSG, es decir, a partir de  $t=0s$ , para relacionar las distintas muestras de frecuencia cardíaca con las etiquetas del sueño respectivas.

La variable heart rate del sujeto 3 tiene unas dimensiones de  $5857 \times 2$ . La primera columna es de tiempo y la segunda es el valor de frecuencia cardíaca para cada  $t$ . El número de valores Nulos o NaN es 0. Además del histograma, se ha decidido representar también el *boxplot* (Figura 3) para localizar valores atípicos. A pesar de haber algunos valores atípicos, alejados de la distribución general de los datos, estos no son considerados erróneos ya que están entre 40 y 100, el rango fisiológico normal de la variable de frecuencia cardíaca [5]. En la Figura 4 podemos apreciar la distribución de la frecuencia cardíaca del sujeto a lo largo de la grabación del estudio PSG. Al final de la grabación hay anomalías de la señal relacionadas probablemente con la manipulación de la pulsera al terminar el estudio.



**Figura 3.** Diagrama de cajas y bigotes e histograma de la variable heart rate del sujeto 3.

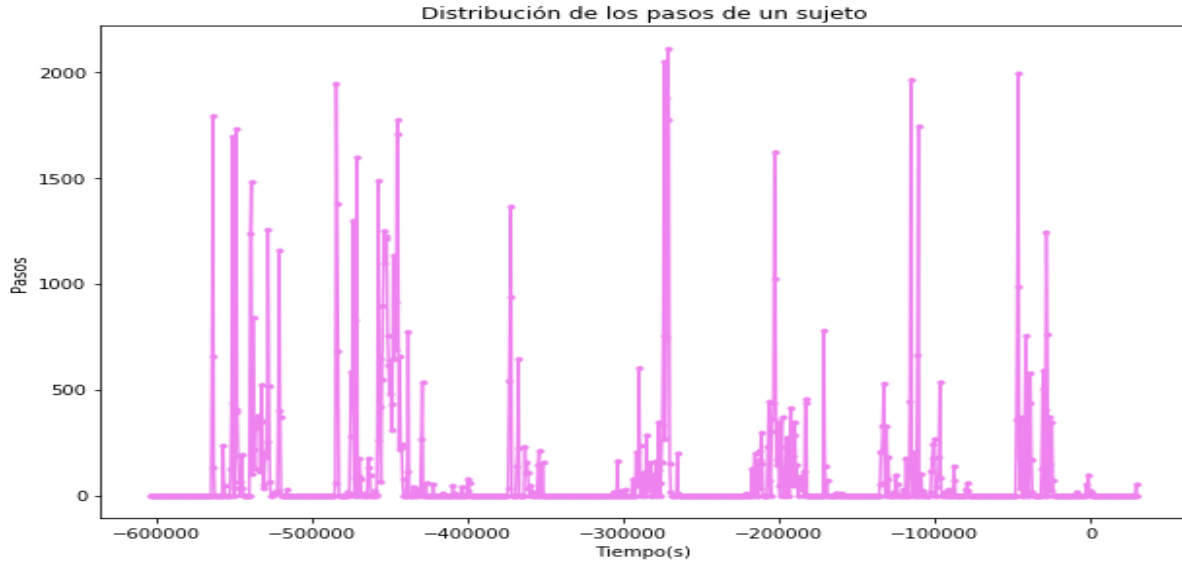


**Figura 4.** Representación a lo largo del tiempo de la variable heart rate del sujeto 3.

## STEPS

En este caso, sólo hay valores distintos de 0 para  $t < 0s$ , debido a que durante la PSG el sujeto permanece acostado toda la noche. La variable steps del sujeto 3 tiene unas dimensiones de  $1057 \times 2$ . La primera columna es de tiempo y la segunda es el número de pasos dados. El número de valores Nulos o NaN es 0.

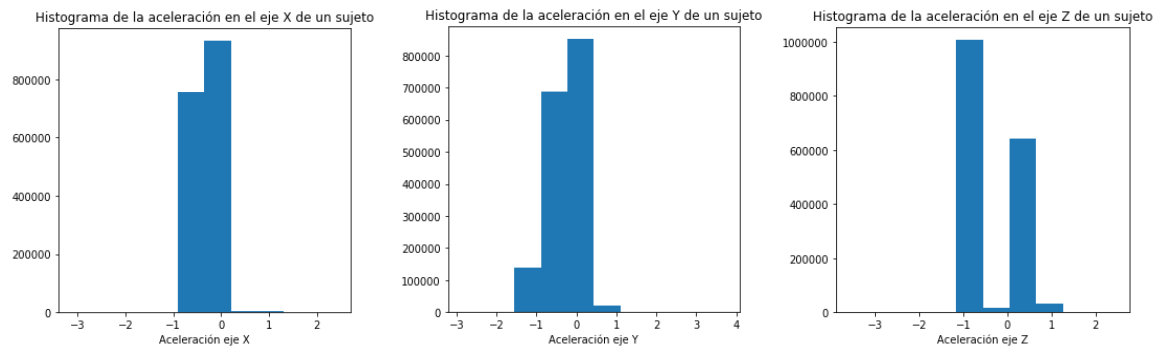
En la Figura 5 se pueden observar 'grupos' de pasos, esto es, periodos de tiempo en los que el número de pasos es elevado. Los grupos de pasos se corresponden con el periodo de mayor actividad diaria del sujeto; y por el contrario, otros intervalos en los que la actividad es nula podría coincidir con los periodos de descanso y sueño del sujeto. Así, estas tendencias están relacionadas con el día y la noche.



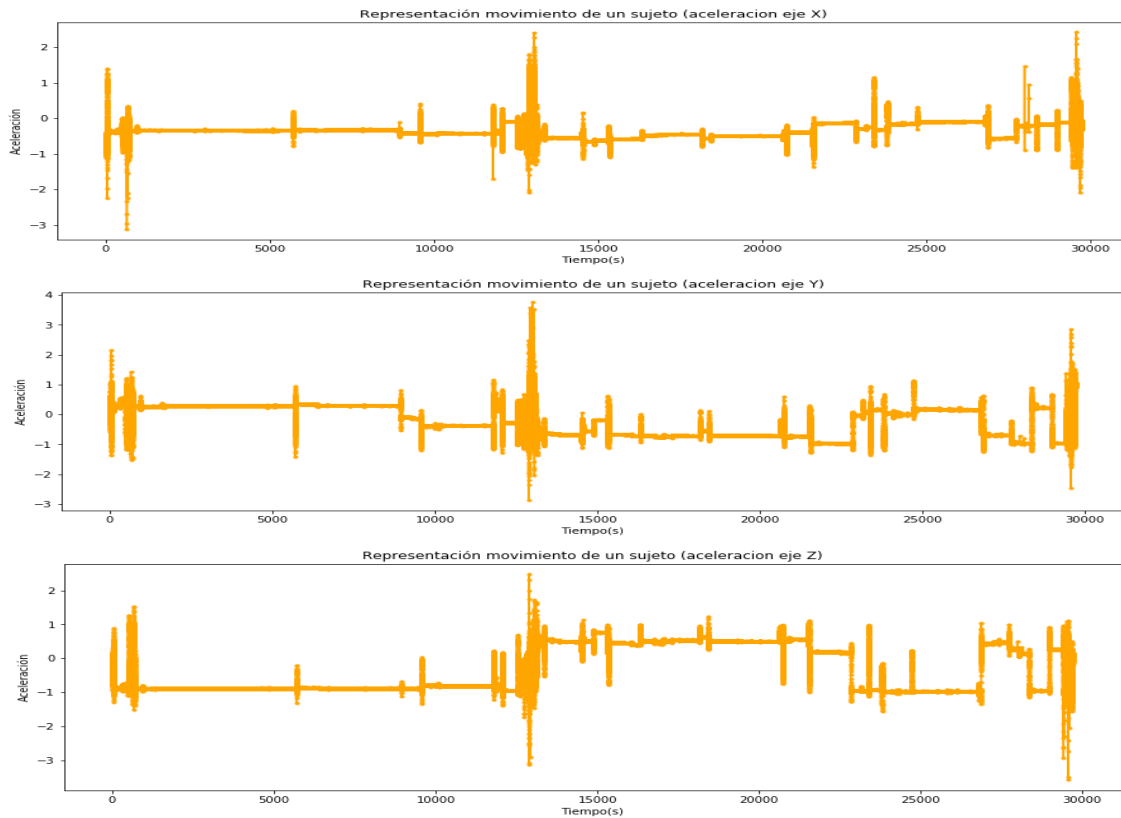
**Figura 5.** Representación a lo largo del tiempo de la variable steps del sujeto 3.

## MOTION

En este caso, se ha decidido trabajar sólo con la información de motion con  $t > 0s$  para relacionarla con las distintas fases del sueño etiquetadas durante el estudio PSG. Para el primer sujeto, el sujeto 0, la variable motion tiene unas dimensiones de 1.697.809x4. La primera columna es de tiempo y las 3 restantes son la aceleración en los ejes X, Y y Z respectivamente. El número de valores Nulos o NaN es, de nuevo, 0.



**Figura 6.** Histogramas de la variable motion en los 3 ejes de la aceleración del sujeto 3.



**Figura 7.** Representación a lo largo del tiempo de la variable motion aceleración del sujeto 3.

Como conclusiones de este apartado, tras representar y estudiar los datos de todos los sujetos podemos observar que los valores negativos en el eje de tiempo de las distintas variables indican, como se presuponía, que esas muestras se han tomado antes de realizar la polisomnografía ( $t=0s$ ). Es importante también destacar que cada sujeto, en cada variable, tiene un **número de muestras distinto** al resto de sujetos del estudio. La cantidad de datos recogidos en cada variable para cada sujeto varía en función del tiempo que cada uno de ellos utilizó el dispositivo Apple Watch y también de la frecuencia de muestreo a la que fue recogida cada señal.

Otra de las conclusiones relevantes que se ha extraído es el **desbalanceo de clases** en la variable labels. Por este motivo, la base de datos se someterá a un preprocesamiento previo a su utilización en los clasificadores.

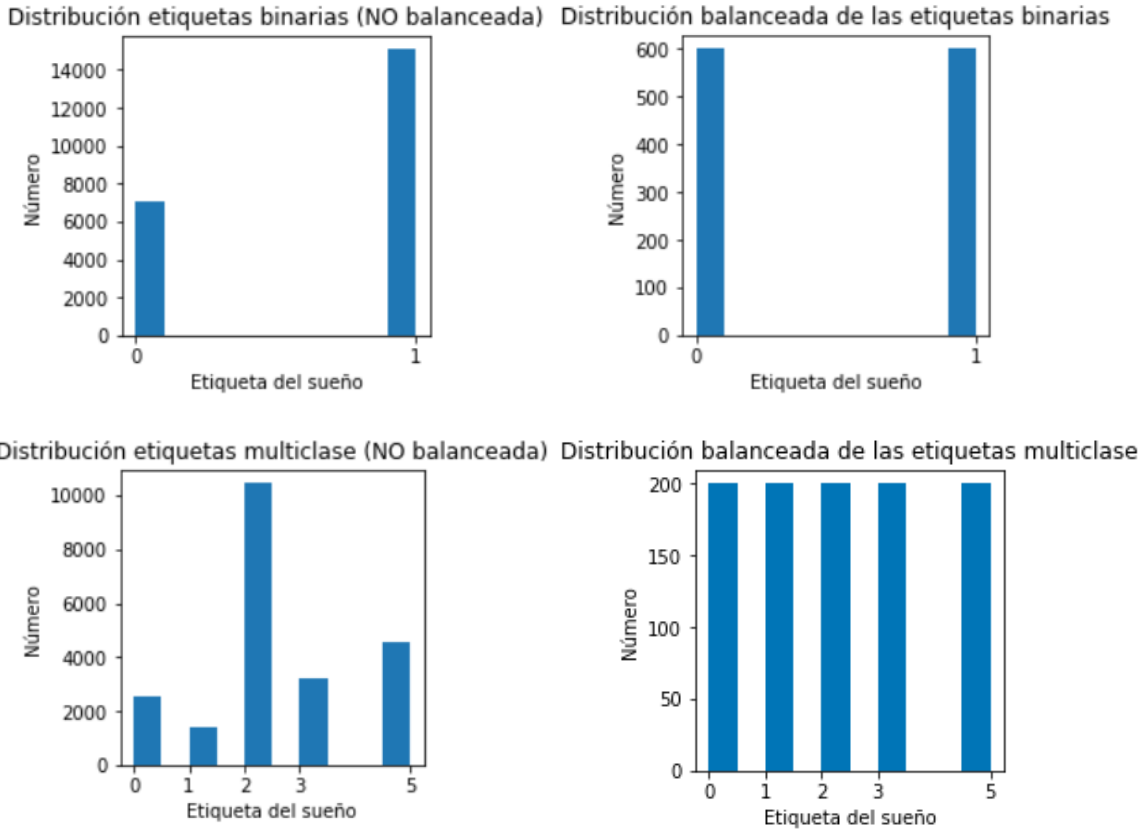
Además, la variable steps no va a ser de utilidad en esta aproximación debido a que se ha decidido utilizar únicamente las muestras de  $t > 0s$ . De esta forma, se busca relacionar las variables fisiológicas con las etiqueta del sueño del estudio polisomnográfico para trabajar problemas de clasificación supervisados.

## 2.3. PREPROCESAMIENTO

Como ya se ha comentado en la Sección 2.2 de Visualización, existen algunas inconsistencias que suponen problemas para trabajar con los distintos algoritmos con los que se pretenden extraer conclusiones. Por este motivo, es de real importancia una etapa de preprocesamiento para obtener unos datos más estructurados.

No todos los sujetos tienen la información recogida durante el mismo intervalo de tiempo. Además, las 4 variables con las que se cuenta han sido tomadas con frecuencias de muestreo diferentes. Para solucionar esto, se decide trabajar con la información de  $t > 0s$ . De esta forma, se utiliza la información directamente relacionada con la información extraída de la PSG (*labels*), y por tanto, se va a trabajar sobre un problema de aprendizaje supervisado. En la etapa de 'Extracción de características' (Sección 2.4) se soluciona el problema de la **dimensionalidad de cada variable**. Además, después de esta etapa aparecen algunos valores NaN, por lo que se decide eliminar aquellas muestras que contienen este tipo de datos para evitar problemas en la etapa de 'Clasificación'. Para ello, se utiliza la función '*clean\_dataset*' y el *dataframe* resultante se almacena en un archivo .csv con el nombre 'Dataframe' para su posterior utilización.

Para solucionar el desbalanceo de clases se han realizado dos modificaciones. La primera consiste en la **agregación de las clases minoritarias** -1 y 4 a sus clases más cercanas, de forma que se tienen cinco clases y no siete: 0 (0 y -1), 1, 2, 3 (3 y 4) y 5. La segunda modificación consiste en obtener el número de muestras por cada clase (*.value\_counts()*) y calcular el mínimo de esos valores (*min()*), que resultó ser 1411. En un principio, el número de muestras de cada clase que se emplea para construir el subconjunto de Train es el correspondiente al valor mínimo que se ha calculado. Tras realizar diferentes pruebas, se observó que ese número era demasiado grande para trabajar con los clasificadores por lo que se ha reducido a 600 para el caso binario y 200 para el caso multiclase. De esta manera, se consigue tener el mismo número de muestras para cada etiqueta, como se muestra en la Figura 8. Por último, una vez que ya se tienen los subconjuntos de Train y Test se realiza la normalización de los datos. De esta forma, se podrán evaluar los diferentes algoritmos con los conjuntos de Train y Test normalizados y valorar si esto supone una ventaja o no. Como se verá, la normalización realizada es la estandarización en la que se hace que los datos tomen media 0 y varianza o desviación típica 1.



**Figura 8.** Histogramas del efecto del balanceo de etiquetas binarias y multiclase.

## 2.4. EXTRACCIÓN DE CARACTERÍSTICAS

Debido a que la frecuencia de muestreo de las señales es distinta, se ha decidido aplicar el método de la ‘ventana deslizante’ para realizar la extracción de características y estructurar y compatibilizar los datos de todos los sujetos. Con este método, cuyo código está desarrollado en el *Notebook* ‘*Dataframe*’ del repositorio [3], se extraen los descriptores estadísticos de mayor interés de cada variable y se consiguen organizar los datos en intervalos de tiempo específicos.

La función ‘*get\_window*’ tiene como parámetros de entrada la señal fisiológica (‘*signal*’), el tamaño de la ventana en segundos (‘*w\_size*’) y la longitud de la señal de las etiquetas del sueño (‘*n\_sueño*’). Esta función recorre el eje de tiempo de la señal deseada y va asignando índices (True o False) según se cumpla o no una condición. Dicha condición depende directamente del tamaño de ventana que se seleccione. Si el valor del eje de tiempo cumple la condición, se asigna True. De lo contrario, se asigna False.

En esta ocasión, se selecciona un tamaño de ventana de 30 segundos para que la información de cada característica, recogida por medio del dispositivo *Apple Watch*, coincida con las etiquetas establecidas por la PSG, las cuales están equiespaciadas 30s.

Las características extraídas se basan en calcular descriptores estadísticos de cada una de las ventanas de 30s de las variables de todos los sujetos, con pequeñas modificaciones. Los estadísticos empleados son la media, el valor máximo y el valor mínimo para cada una de las variables. En el caso de la variable movimiento, estos estadísticos se calculan para cada uno de los ejes de la aceleración (x,y,z). Además, en esta variable, se ha calculado la raíz cuadrada de la suma de cada una de las componentes de la aceleración al cuadrado. A partir de la raíz cuadrada se ha calculado también la media (ec.  $\overline{a} = \sqrt{\overline{x}^2 + \overline{y}^2 + \overline{z}^2}$ ).

Con todas las características se ha creado un *dataframe* que incluye las ventanas de todos los sujetos y se ha identificado a cada uno de ellos con un valor numérico. De esta forma, se añade una columna con los identificadores de cada sujeto (ID) para posteriores tareas como conseguir que los conjuntos de entrenamiento y evaluación no contengan muestras de un mismo sujeto. En la Tabla 2 se muestran todas las características extraídas con la ventana deslizante a partir de las distintas variables originales. Estas características serán utilizadas como datos de entrada para los distintos modelos de clasificación.

Nombre	Información
Feat1 (Min_hr)	Frecuencias cardíacas mínimas
Feat2 (Max_hr)	Frecuencias cardíacas máximas
Feat3 (Mean_hr)	Frecuencias cardíacas medias
Feat4 (Mean_acc_sqrt)	Aceleraciones medias
Feat5 (Min_acc_x)	Aceleraciones eje x mínimas
Feat6 (Max_acc_x)	Aceleraciones eje x máximas
Feat7 (Mean_acc_x)	Aceleraciones eje x medias
Feat8 (Min_acc_y)	Aceleraciones eje y mínimas
Feat9 (Max_acc_y)	Aceleraciones eje y máximas
Feat10 (Mean_acc_y)	Aceleraciones eje y medias
Feat11 (Min_acc_z)	Aceleraciones eje z mínimas
Feat12 (Max_acc_z)	Aceleraciones eje z máximas
Feat13 (Mean_acc_z)	Aceleraciones eje z medias

**Tabla 2.** Características extraídas.

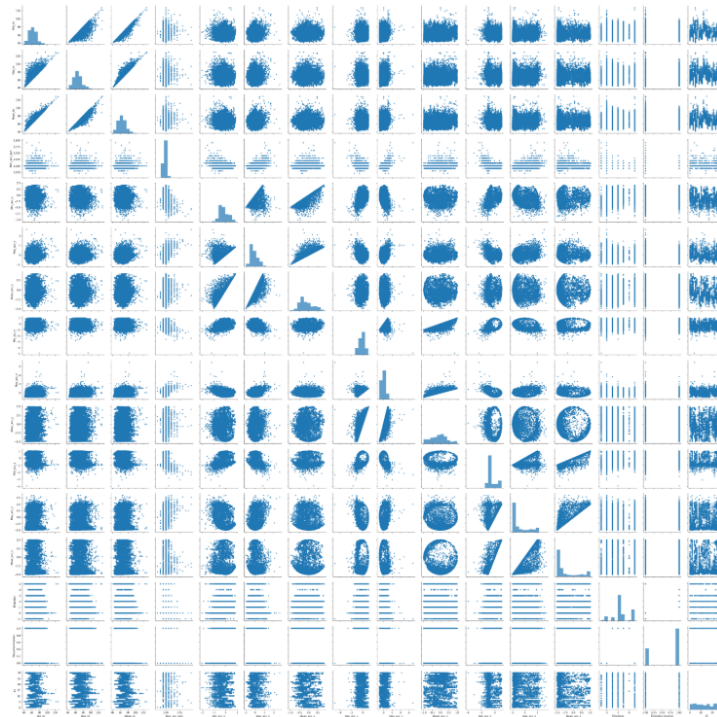
Además, la información extraída del estudio PSG se ha modificado para obtener dos columnas diferenciadas:

- Etiquetas **binarias**: con dos posibles valores: 0 para sueño ligero y 1 para sueño profundo. En la clase 0 se han agregado las clases (-1,0,5) y en la clase 1 las restantes (1,2,3,4).
- Etiquetas **multiclase**: con valores cinco posibles valores. En la etiqueta 0 se han agregado las clases (-1,0) y en la etiqueta 3 se han agregado las clases (3,4). Las demás etiquetas (1,2,5) permanecen con sus clases originales sin modificar ni agregar otras.

A continuación, se realizó un diagrama de dispersión y un mapa de calor del conjunto de datos para observar las relaciones lineales entre todas las características. En la Figuras 9 y 10 se puede



observar que ciertas variables (`mean_hr` y `min_hr`, por ejemplo) presentan una elevada correlación. Sin embargo, se considera no prescindir de ninguna de las características debido al reducido número de variables extraídas y la robustez de los algoritmos de clasificación seleccionados en el proyecto frente a la redundancia de variables.



**Figura 9.** Diagrama de dispersión del conjunto de datos

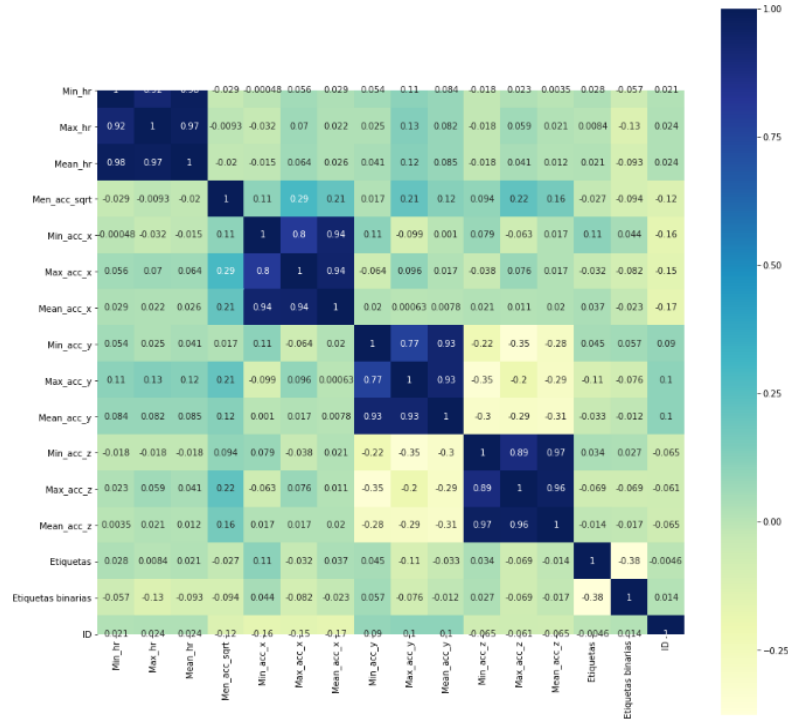


Figura 10. Mapa de calor del conjunto de datos

## 2.5. SUBCONJUNTOS DE TRAIN Y TEST

La creación de los subconjuntos de Train y Test se puede observar en el archivo ‘TrainTest.ipynb’ del repositorio de *GitHub* [3].

Como se explicaba en la Sección 2.3 de Preprocesamiento, se ha realizado el balanceo de clases de la matriz original de características extraídas ‘Dataframe’. Este proceso se ha realizado en dos vertientes: balanceo de las clases binarias (‘df\_bin\_balanced’) y balanceo de las clases multiclase (‘df\_multi\_balanced’). De este modo tenemos un *dataframe* sin balancear, uno con las clases binarias balanceadas y otro con las clases multiclase balanceadas.

Seguidamente se ha destinado el 78% de los sujetos al subconjunto de Train para estimar los parámetros de los modelos de clasificación. El subconjunto de datos de Test se ha realizado con un 22% de los sujetos y se empleará para comprobar el comportamiento del modelo estimado. Es realmente importante que las muestras de cada sujeto aparezcan tan sólo en uno de los dos subconjuntos, por lo que se realiza un estricto procedimiento de muestreo. Como primer paso, se elaboran 2 listas con números aleatorios del 0 al 30 en una proporción 3:1, de manera que no haya valores iguales en ambas listas. De esta manera, se eligen de manera aleatoria los sujetos que formarán parte de cada subconjunto. Con cada una de las listas se crean los subconjuntos de Train y Test de manera que los datos no se mezclen, es decir que no aparezca el mismo sujeto en los dos subconjuntos a la vez.

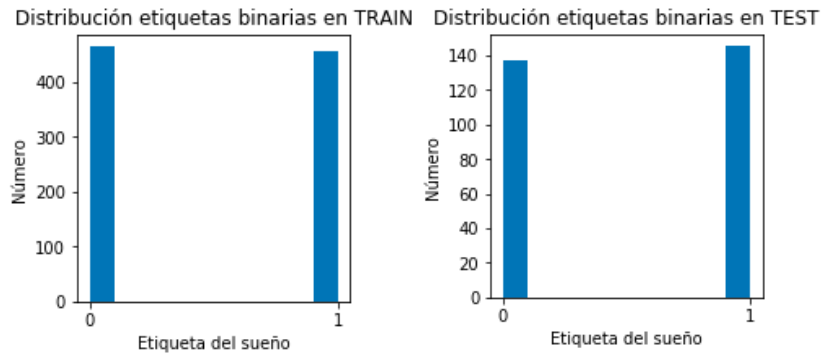
En lo referente al *dataframe* original sin balancear, en una primera aproximación se crearon dichos subconjuntos con los datos de tan sólo 30 ventanas deslizantes, aproximadamente un 3.5% del total de datos disponible. Este paso se introdujo debido a cuestiones de tiempo de ejecución del código. Finalmente se consideró subir ese umbral hasta 100 para poder tener unos conjuntos de mayores dimensiones y así mejorar las prestaciones de los clasificadores, teniendo todavía un tiempo de ejecución razonable. Los subconjuntos de Train y Test extraídos se han guardado en archivos .csv con el nombre de 'Train' y 'Test' y su versión normalizada con los nombres 'Train\_transformed' y 'Test\_transformed'. De esta forma, se podrá comprobar el efecto del balanceo y la normalización en la calidad de los clasificadores.

Al introducir el balanceo de las clases se reducen bastante las dimensiones de la matriz original, hasta un 12.73% para el *dataframe* con el balanceo de las etiquetas binarias y un 3.83% para el del balanceo de las etiquetas multiclase. Por ese motivo, se ha considerado trabajar con 600 ventanas para cada clase en los modelos binarios y 200 de cada clase para los modelos multiclase para agilizar la búsqueda de parámetros y la ejecución de los modelos. Además, se ha aplicado normalización a los respectivos subconjuntos para observar posibles cambios en los clasificadores. En perspectiva, se han extraído los subconjuntos mostrados en la tabla 3.

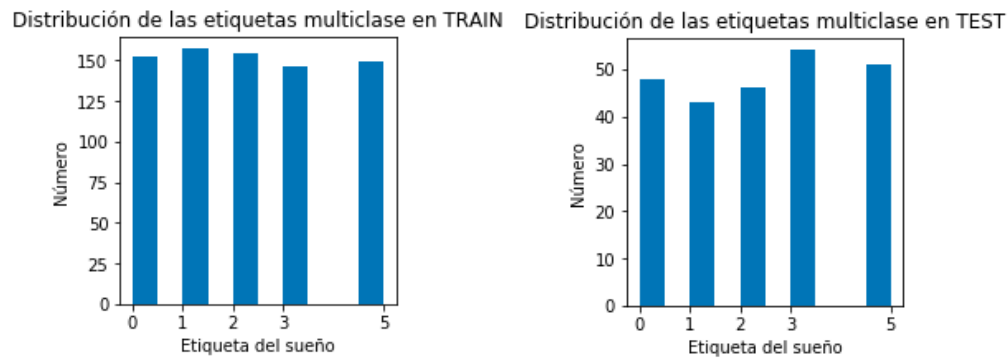
Dataframe sin balancear	Dataframe balanceado para etiquetas binarias	Dataframe balanceado para etiquetas multiclase
Train	Train_binary	Train_multiclass
Test	Test_binary	Test_multiclass
Train_transformed	Train_binary_transformed	Train_multiclass_transformed
Test_transformed	Test_binary_transformed	Test_multiclass_transformed

**Tabla 3.** Subconjuntos creados.

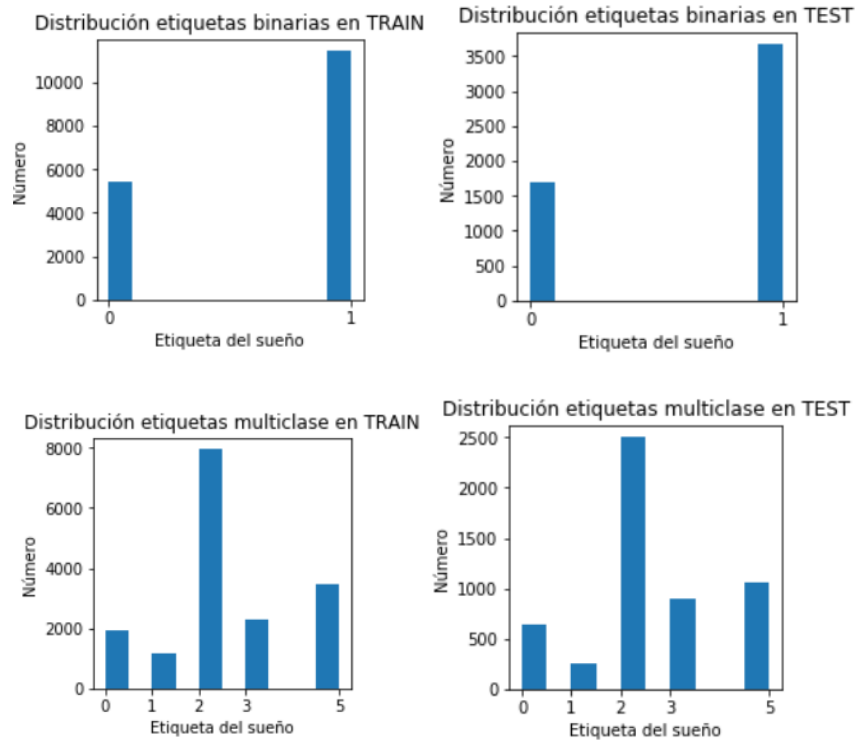
Todos los subconjuntos son sometidos a un proceso de visualización tanto gráfica como numérica para comprobar su calidad. A continuación se muestran algunos ejemplos de visualización gráfica de los conjuntos de Train y Test para etiquetas binarias, multiclase y los extraídos a partir del *dataframe* sin balancear. Como se puede apreciar en las Figuras 11 y 12, la distribución de las clases tras realizar el submuestreo de 3:1 no es exacta debido a que no todos los sujetos tienen exactamente el mismo número de muestras de cada clase. Aún así, se puede observar que el desbalanceo no es excesivo, por lo que no influirá negativamente en los resultados de clasificación.



**Figura 11.** Distribución de los subconjuntos balanceados de etiquetas binarias.



**Figura 12.** Distribución de los subconjuntos balanceados de etiquetas multiclase.



**Figura 13.** Distribución de clases para los subconjuntos sin balancear.

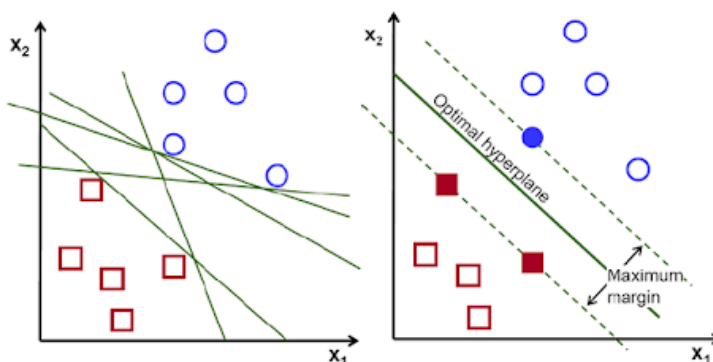
## 2.6. MODELOS DE CLASIFICACIÓN

En esta sección se definen de forma teórica los tres algoritmos de Inteligencia artificial utilizados para la estimación de las etiquetas del sueño: máquina de vectores soporte, perceptrón multicapa y árboles de decisión. En el *Capítulo 3* se detallan los resultados obtenidos tras aplicar cada uno de estos métodos de clasificación.

### 2.6.1. Máquinas de vectores soporte

Las máquinas de vectores soporte (SVM, del inglés *Support Vector Machines*), son un conjunto de algoritmos de aprendizaje supervisado relacionados con problemas de clasificación o regresión.

En el caso de que los datos sean linealmente separables, la frontera de separación es un hiperplano y el error de clasificación se minimiza bajo la restricción de maximizar el margen entre el hiperplano de separación y cualquier muestra del conjunto de entrenamiento. De esta manera, se busca aumentar la generalización. De entre todo el conjunto de datos, en este algoritmo de clasificación sólo son importantes para construir el clasificador algunas muestras, las conocidas como Vectores Soporte (SV, del inglés *Support Vector*). El SVM lineal se trata de un problema de optimización con restricciones, que se puede resolver aplicando el método de los multiplicadores de Lagrange. De esta manera, las muestras con un multiplicador de lagrange mayor que 0 serán los vectores soporte, los casos que definen el hiperplano.



**Figura 14.** Hiperplano óptimo de separación en el algoritmo SVM lineal. Tomado de [8]

Si el conjunto de datos no se puede separar razonablemente bien utilizando un decisor lineal, los datos se proyectan en un espacio de mayor dimensión. Esta transformación es no lineal y se puede realizar utilizando funciones núcleo o funciones *kernel*. De esta manera, se facilita la separación de las muestras y se diseña el decisor lineal en el nuevo espacio de características. Las funciones *kernel* pueden ser polinómicas o gaussianas, principalmente. Por último, cabe mencionar que este algoritmo es muy efectivo para problemas con muchas dimensiones.

En Python se puede implementar este algoritmo con el módulo *sklearn.svm* de la biblioteca *Scikit-learn*. Concretamente, la función *sklearn.svm.SVC* soporta clasificación binaria o multiclase tanto

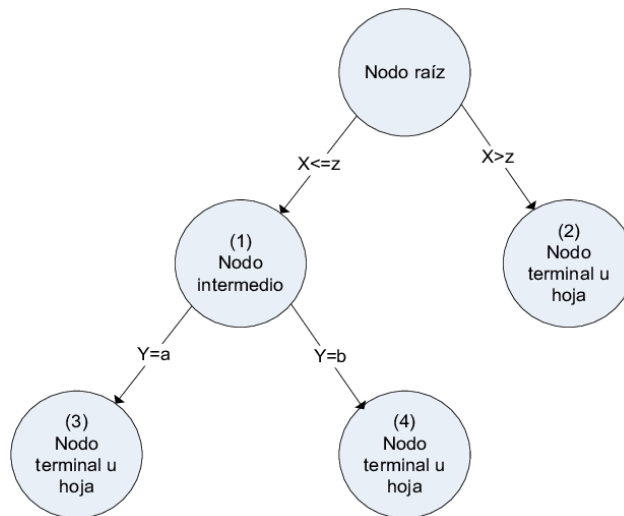
lineal como no lineal. A continuación se describen brevemente los parámetros a especificar, aunque se puede consultar la referencia [7] para obtener más información.

- **C** es el parámetro de regularización. Este parámetro es estrictamente positivo y se determina explorando varios valores y aplicando técnicas de validación cruzada. Se emplea una regularización L2, agregando una penalización igual al cuadrado de la magnitud de los coeficientes. Si C tiene un valor alto puede resultar en un modelo demasiado complejo, con poca suavidad en la frontera de decisión y con el riesgo de sobreajuste. Por el contrario, si el parámetro C tiene un valor demasiado bajo puede resultar en un modelo demasiado sencillo, con demasiada suavidad en la frontera de decisión. El valor por defecto es 1.
- El **kernel** determina si el modelo de clasificación es lineal (*'linear'*) o no lineal (*'poly'*, *'rbf'*...). El valor por defecto es *'rbf'*.
- **Gamma** es el coeficiente que se debe indicar al usar un kernel gaussiano (*kernel='rbf'*). Este parámetro refleja la anchura del kernel. El parámetro gamma puede entenderse como la inversa del radio del kernel gaussiano. Así, un menor gamma implica una mayor suavidad en la frontera de decisión, y por el contrario, un valor alto de gamma indica una menor suavidad en la frontera de decisión y la tendencia al sobreajuste. Los posibles valores de este parámetro son *'scale'*, *'auto'* y un número en formato *float* o *int*. El valor por defecto es *'scale'*.

### 2.6.2. Árboles de clasificación

Los árboles de clasificación o decisión son métodos de aprendizaje estadístico supervisado no paramétricos y no lineales. El objetivo de este algoritmo es crear un modelo en forma de árbol para predecir un valor de la variable dependiente (salida) mediante simples reglas de decisión aprendidas a partir de los casos de entrenamiento.

Los elementos principales del árbol son el nodo raíz, los nodos intermedios y los nodos terminales. La elección del nodo raíz se realiza utilizando una medida de pureza, cuyo valor máximo tendrá la muestra seleccionada como nodo raíz. Además, esta medida se calcula también para cada una de las variables al realizar una partición en un nodo del árbol. La variable seleccionada será siempre la que tenga el valor más alto de la medida de pureza.



**Figura 15.** Ejemplo de grafo que representa un árbol de clasificación. Tomado de [11]

Para evitar que los árboles sean demasiado profundos y complejos, y por tanto, haya sobreajuste, se deben de realizar estrategias de ‘poda’ o *pruning*. Hay varias formas de hacerlo y las más comunes son limitar el mínimo número de muestras para dividir un nodo, el mínimo número de muestras para un nodo terminal o la máxima profundidad del árbol.

En Python, se puede implementar este algoritmo con el módulo *sklearn.tree* de la biblioteca *Scikit-learn*. Concretamente, con la función *sklearn.tree.DecisionTreeClassifier*.

A continuación se describen brevemente los parámetros a especificar, aunque se puede consultar la referencia [9] para obtener más información.

- **criterion** es el parámetro relacionado con la medida de la pureza, que se utiliza para realizar las particiones. Los valores que puede tomar son ‘*gini*’ o ‘*entropy*’. El primero de ellos, el índice de Gini, es una generalización de la varianza binomial. El segundo valor hace referencia a la ganancia de entropía, que se calcula como la diferencia entre la entropía a priori y la entropía del atributo X. De esta forma se puede evaluar cuánta información proporciona este atributo. El valor por defecto es ‘*gini*’.
- **max\_deph** se refiere a la máxima profundidad del árbol.
- **min\_samples\_split** es el mínimo número de muestras requerido para dividir un nodo intermedio. El valor por defecto es 2.
- **min\_samples\_leaf** es el mínimo número de muestras requerido para un nodo terminal. El valor por defecto es 1.

Una de las desventajas de este método de clasificación es que es un método que sobreajusta con facilidad, por lo que es necesario incluir y revisar frecuentemente las estrategias de poda. Además, es un algoritmo poco robusto, ya que pequeños cambios en los datos de entrenamiento

pueden ocasionar grandes cambios en los árboles contruidos y, consecuentemente, en sus predicciones.

### 2.6.3. Perceptrón multicapa

El perceptrón multicapa (MLP, del inglés *Multilayer Perceptron*) es una red neuronal artificial formada por múltiples capas. Este algoritmo tiene la capacidad principal de resolver problemas no linealmente separables.

El MLP tiene una estructura característica en capas de tres tipos: capas de entrada, capas ocultas y capas de salida. La propagación hacia atrás (del inglés *backpropagation*) es el algoritmo que se utiliza para entrenar este tipo de modelos. Utilizando este método de retropropagación de errores se pueden determinar los parámetros asociados a las neuronas de las capas ocultas. El MLP es un método utilizado comúnmente para resolver problemas de asociación de patrones.

Este algoritmo se puede implementar en Python con el módulo `sklearn.neural_network` de la biblioteca *Scikit-learn*. Concretamente, con la función `'sklearn.neural_network.MLPClassifier'`. A continuación se describen brevemente los principales parámetros a especificar, aunque se puede consultar la referencia [10] para obtener más información.

- **hidden\_layer\_sizes** representa el número de neuronas de la capa oculta. El valor por defecto es 100.
- **activation** es la función de activación de la capa oculta. Puede tomar los valores: `'logistic'`, `'tanh'`, `'relu'`... Por defecto utiliza la función `'relu'`.
- **solver** es el método de optimización de los pesos. Admite los valores `'lbfgs'`, `'sgd'` y `'adam'`. Por defecto utiliza el método `'adam'`.
- **learning\_rate** es la tasa de aprendizaje en la actualización de los pesos. Puede tomar valores de `'constant'`, `'invscaling'`, `'adaptive'`. Por defecto utiliza `'constant'`.
- **max\_iter** es el número máximo de iteraciones. Toma valores enteros y por defecto es 200.
- **shuffle** indica si las muestras se mezclan en cada iteración o no. Puede tomar valores de `'True'` o `'False'`. Por defecto utiliza el valor `'True'`.
- **Early\_stopping** es la detección temprana para finalizar el entrenamiento cuando la puntuación de validación no mejora. El entrenamiento finalizará cuando el valor de validación no mejore durante las interacciones de `n_iter_no_change`. Puede tomar valores de `'True'` o `'False'`. Por defecto utiliza el valor `'False'`.
- **n\_iter\_no\_change** es el número máximo de iteraciones permitidas sin que se mejoren las prestaciones. Toma valores enteros y como valor por defecto tiene 10.
- **validation\_fraction** es la proporción de datos de entrenamiento que se reservan como conjunto de validación. Solo se utiliza cuando la opción de early stopping es `'True'`. Puede tomar valores entre 0 y 1. Por defecto utiliza 0.1.



- **verbose** muestra el entrenamiento. Puede tomar valores de 'True' o 'False'. Por defecto utiliza el valor 'False'.

#### 2.6.4. Métodos de evaluación

La evaluación del modelo es una de las fases principales en todo el proceso de análisis de datos. Consiste en evaluar la calidad de un modelo ya entrenado mediante varias métricas que aportan información sobre su rendimiento. La medida más simple es la proporción de aciertos o *accuracy* (exactitud). La finalidad de esta fase es comparar distintos modelos para elegir el mejor y además predecir cómo se comportará el modelo una vez usado en una situación real. Es muy importante tener claro que este proceso de evaluación ha de hacerse sobre un conjunto de datos distinto al empleado para entrenar al modelo.

En *scikit-learn* está implementado '**GridSearchCV**', un método que busca el modelo correspondiente a la mejor combinación de parámetros para un clasificador dado, evaluando con validación cruzada sobre el conjunto de Train. Los valores candidatos para los parámetros se aportan en forma de diccionarios.

Para evaluar cada uno de los algoritmos de clasificación se emplean diferentes métodos. Por un lado se ha obtenido la **matriz de confusión**. Es una de las medidas de prestación más útiles para saber si nuestro modelo está aprendiendo bien. En esta matriz, las columnas representan clases predichas mientras que las filas representan clases reales. Este tipo de métrica permite ver fácilmente si el clasificador está confundiendo clases o las está clasificando correctamente. Así, en la diagonal están los casos clasificados correctamente y fuera de ella veremos los fallos de nuestro clasificador.

		Predicción	
		Positivos	Negativos
Observación	Positivos	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (VN)

**Figura 16.** Esquema de los datos que se representan en la matriz de confusión

A partir de los valores expresados en la matriz de confusión (VP, FN, FP, VN) se calculan otras medidas de prestación como la tasa de acierto y la sensibilidad.

$$Tasa\ de\ acierto = \frac{VP + VN}{VP + VN + FP + FN}$$

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

**Figura 17.** Ecuaciones para la obtención de la tasa de acierto y la sensibilidad.

Otra de las medidas utilizadas para la evaluación de los distintos modelos utilizados en este proyecto, ha sido el ***classification report***. Para hacer uso del mismo, se puede utilizar el paquete *scikit-learn* de Python, en concreto, con la función '*sklearn.metrics.classification\_report*'. Esta función proporciona un informe con las principales métricas de clasificación [12]. Las prestaciones proporcionadas son *precision*, *recall* (sensibilidad) y *F1 score* (representa la *accuracy*) para cada clase. Estas prestaciones incluyen *macro average* y *weighted average*, las cuales representan medidas promedio de cada una de las prestaciones anteriores.

## **2.7. PARALELIZAR**

La computación paralela es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente. Gracias a este tipo de computación, la mayoría de los grandes problemas se pueden dividir en problemas más pequeños que se resuelven simultáneamente (en paralelo). Las estructuras de programación en las que se centra la paralelización son los bucles, ya que, en general, la mayor parte del tiempo de ejecución de un programa tiene lugar en el interior de algún bucle. Un proceso de paralelización intenta dividir un bucle de forma que sus iteraciones puedan ser ejecutadas en núcleos separados de forma concurrente.

Debido al gran volumen de datos con los que se trabaja y el gran número de operaciones que se realizan en este proyecto, se ha decidido realizar computación paralela para agilizar algunos procesos, como el bucle *for* que se utiliza para la obtención de la matriz de características o la función de búsqueda de los parámetros óptimos de los distintos clasificadores. Se ha comprobado, para estas tareas, que la computación paralela distribuye de manera horizontal las operaciones, ya que se realizan en un periodo de tiempo notablemente menor.

Antes de seleccionar el método más óptimo para paralelizar las distintas partes del proyecto, se realizó un estudio de distintos procesadores y compiladores existentes en nuestros días para llevar a cabo la computación paralelizada.

### **2.7.1. Paquete *multiprocessing***

En Python, el uso de una sola CPU es causado por el bloqueo global del intérprete (GIL, del inglés *Global Interpreter Lock*), que permite que solo un procesador lleve el intérprete de Python en un momento dado. El GIL se implementó para manejar un problema de administración de memoria, pero como resultado, Python se limita al uso de un único procesador. Omitir el GIL al ejecutar el código Python permite que el código se ejecute más rápido porque ahora podemos aprovechar el multiprocesamiento. El paquete *multiprocessing* incorporado de Python permite enviar el código a múltiples procesadores para su ejecución simultánea.

Dentro de *multiprocessing* existen dos formas de paralelizar: *Pool* y *Process*. Ambos ejecutan tareas en paralelo, pero su forma de ejecutarlas es diferente [13]. Por un lado, ***Pool*** distribuye las tareas a los núcleos disponibles mediante una programación FIFO (del inglés, *First In First Out*) basada en que el primero que entra será el primero que salga. Su arquitectura de funcionamiento es similar a la de *map reduce*, en la que se asigna la entrada a los diferentes procesadores y se recopila la salida de todos los procesadores. Después de toda la ejecución del código, devuelve el resultado en forma de lista o matriz. Los procesos en ejecución se almacenan en la memoria y los que no se ejecutan se almacenan fuera de la memoria. Por otro lado, ***Process*** coloca todos los procesos en la memoria y programa su ejecución utilizando la programación FIFO.

Elegir el enfoque apropiado depende de la tarea en cuestión. *Pool* permite realizar múltiples tareas por proceso y es uno de los métodos más utilizados para problemas en los que se tiene que realizar un gran número de tareas. Si el número de tareas es pequeño y solo se necesita realizar cada tarea una vez, es razonable usar *Process*, ya que procesa por separado para cada tarea, en lugar de configurar un *Pool*.

Además, la estructura de paralelización *Pool*, permite utilizar tres funciones dependiendo de la tarea a paralelizar: *.map()*, *.apply()* y *.starmap()*. La diferencia entre ellas es el tipo de operaciones que pueden procesar y los argumentos que pueden tomar. En el caso de la función *.map()*, que permite como argumentos listas o tuplas, se pueden paralelizar operaciones sencillas pero de una manera muy rápida. La función *.apply()* permite como argumento más tipos de estructuras y además, paralelizar procesos (como los bucles) y operaciones más complejas. Por último, *.starmap()* tiene el mismo funcionamiento que *.map()* pero permite como argumento estructuras más complejas que las listas o las tuplas [13].

### 2.7.2. Paquete numba

El paquete de Numba ofrece una gran variedad de opciones para paralelizar código en Python para CPU y GPU, a menudo con cambios menores en el código [14]. Si el código a paralelizar está orientado numéricamente, es decir, hace muchos cálculos, se utilizan mucho funciones del paquete NumPy o tiene muchos bucles, Numba suele ser una buena opción. El compilador JIT (*@jit*), *just in time*, es el compilador de Numba para funciones y código en Python.

Una vez conocidas las fortalezas de Numba, es importante conocer también las limitaciones de su uso, ya que Numba no entiende las funciones ni la programación del paquete de Pandas de Python. Si el código a paralelizar hace uso de este paquete, Numba simplemente ejecutaría este código a través del intérprete, lo que supone un mayor gasto de los recursos de Numba y por tanto, el código no se va a beneficiar de sus ventajas de paralelización.

### 2.7.3. Cython

Cython es una extensión de Python que permite aumentar la velocidad de ejecución de procesos programados en lenguaje Python. La forma más fácil de aumentar la velocidad de una función escrita en Python es volver a escribir esta en C y compilarla. Sin embargo, la programación en C es ligeramente diferente a la de Python. Por ello existe Cython, una librería que, con pequeñas modificaciones, permite traducir código en Python a C y compilarlo. Cython, además, sí entiende el paquete Pandas [15].

#### 2.7.4. Google Colaboratory

Google Colab es un servicio de Google gratuito en la nube. Permite desarrollar, en lenguaje de programación Python, aplicaciones de *Deep learning* utilizando las bibliotecas más populares de programación como Keras, TensorFlow, PyTorch y OpenCV. La característica más importante que distingue a Colab de otros servicios gratuitos en la nube es que Colab proporciona el uso de una GPU de forma totalmente gratuita.

Para la realización del proyecto se ha configurado un *Pool* para la obtención de la matriz de características, ya que para ello, es necesario realizar varias tareas un gran número de veces y la cantidad de datos con la que se trabaja es de gran tamaño. La función utilizada para la paralelización ha sido *.apply()* y el número de núcleos empleados para realizarla han sido dos. El tiempo empleado para la obtención de la matriz de características utilizando esta herramienta de paralelización, se ha reducido notablemente. El tiempo empleado para su obtención sin paralelizar era entorno a una hora (entre 50 y 60 minutos) y al configurar *Pool.apply()* este tiempo se ha reducido hasta los 30 minutos (ver anexo '*Tiempos*').

El uso del compilador de Numba se descartó, ya que para muchos de los procesos de este proyecto se ha utilizado el paquete de programación Pandas. Además, tras hacer varias pruebas con la extensión de Cython, se descartó también su aplicación ya que su optimización era más lenta que con otros procesos de paralelización, y no funcionaba para algunas de las secciones del código.

Por último, para aumentar la velocidad de procesamiento de los distintos clasificadores construidos en este proyecto, se ha utilizado la GPU de Google Colab. Al ejecutar los clasificadores en esta plataforma, se ha conseguido optimizar la búsqueda de los parámetros libres y por tanto su ejecución (ver anexo '*Clasificadores en Google Colab*'). Además, como última herramienta para agilizar los procesos, en la búsqueda de parámetros libres (realizada con *GridSearchCV*), los cálculos se han ejecutado en paralelo con la configuración de uno de sus parámetros, *n\_jobs = -1*.

## Capítulo 3

En este capítulo se ofrece una breve descripción del desarrollo seguido durante la etapa de diseño y evaluación de los distintos clasificadores y se muestran los resultados obtenidos. De esta forma, se detallan también los problemas que han ido surgiendo en las diferentes etapas y los métodos que se han planteado como solución.

## Resultados

Como ya se ha ido comentado, en este proyecto se abordan dos problemas de aprendizaje: un problema de clasificación binaria, con el objetivo de predecir si un sujeto se encuentra en una fase de vigilia-sueño ligero o en una fase de sueño profundo; y un problema multiclase que trata de predecir en qué etapa de sueño específica se encuentra el sujeto. Ambos problemas se abordan y evalúan independientemente uno del otro.

Como consideraciones previas, es necesario destacar que se han entrenado y evaluado distintos modelos de aprendizaje supervisado, pero el clasificador más desarrollado ha sido SVM debido a su robustez frente a variables correlacionadas linealmente. Los valores obtenidos para las prestaciones evaluadas en los distintos clasificadores no han sido tan satisfactorios como se deseaba. Por este motivo, se ha buscado mejorar los resultados realizando una normalización de tipo estandarización sobre los datos de entrada. La estandarización consiste en transformar los datos en un conjunto con media cero y varianza o desviación estándar uno. A pesar de realizar esta transformación, no se aprecia una mejoría en los resultados.

Por otro lado, se ha considerado diferente número de muestras en los subconjuntos de entrenamiento y testeo con el objetivo de mejorar la capacidad de aprendizaje y predicción. A pesar de ello, las prestaciones de ambos clasificadores tampoco mejoraron de forma notable. De hecho, se observa que el clasificador multiclase no es capaz de identificar algunas de las clases que conforman el problema. Para solucionar este último problema, se propone como solución aplicar estrategias de balanceo de clases, además de tratar de aumentar el número de muestras de entrenamiento.

Finalmente, se decide emplear la estrategia de balanceo explicada en la *Sección 2.3* sobre los datos de entrada. Al aplicar balanceo, se decide el número de muestras de cada clase con las que se va a entrenar cada uno de los clasificadores. Para el problema binario se ha decidido utilizar 600 muestras de cada clase, por lo que se utilizarán un total de 1200 muestras (5.41% del total de los datos), que se corresponden con 1200 ventanas entre todos los sujetos. Estas muestras, como ya se ha comentado, se dividirán en dos subconjuntos de Train y Test con una proporción de 78% y 22% respectivamente. En el caso del problema multiclase, se ha decidido utilizar 200 muestras de cada clase. En este caso, el total de muestras es de 1000 (4.51% del total de los datos), por lo que se utilizan 1000 ventanas entre todos los sujetos y se construyen los

subconjuntos de Train y Test con la misma proporción que para el caso binario. Estas restricciones de muestras por cada clase se han realizado debido a la capacidad de computación. Otra de las medidas empleadas para extraer conclusiones de los resultados de los clasificadores ha sido comparar las prestaciones sobre el conjunto de Train y de Test. De esta forma, se puede observar si los modelos sobreajustan sobre el conjunto de datos de entrenamiento y si en vez de aprender de ellos están memorizando. Una diferencia muy elevada entre las prestaciones de ambos subconjuntos será un indicador de que el modelo está sobreajustando. Por el contrario, si las prestaciones en ambos subconjuntos son similares, será indicador de que el modelo generaliza bien para cualquier subconjunto de datos.

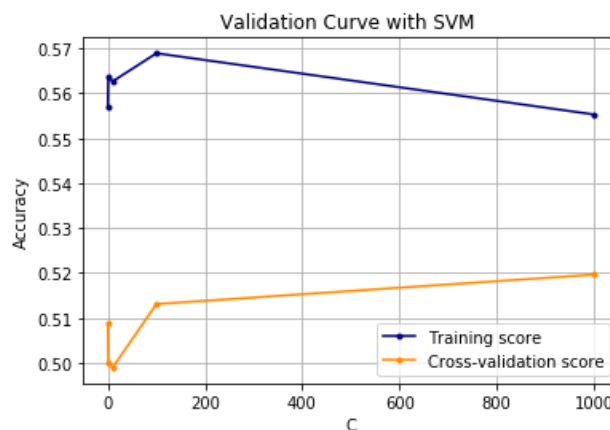
Como medida de evaluación visual, se han incluido gráficas en cada uno de los clasificadores para observar las etiquetas del sueño reales del subconjunto de Test frente a las etiquetas predichas por los clasificadores de este subconjunto.

Los algoritmos que se han utilizado para ambos problemas (binario y multiclase) son SVM lineal, SVM no lineal (kernel gaussiano), árboles de decisión y perceptrón multicapa.

### 3.1 PROBLEMA CLASIFICACIÓN BINARIA CON DATOS BALANCEADOS

#### 3.1.1 SVM lineal

En este clasificador se ha utilizado un kernel lineal y se han realizado distintas pruebas para estimar el mejor parámetro libre 'C' en un rango de escala logarítmica (0.1, 1,10,100). El parámetro óptimo obtenido ha sido C=100. En este caso, al ser el parámetro óptimo el límite del rango de búsqueda establecido, se ha ampliado este rango hasta 1000 por si el parámetro óptimo cambiaba pero sigue siendo C=100. En la Figura 18 se puede observar gráficamente la búsqueda del parámetro C realizada por el método de *GridSearch*.



**Figura 18.** Búsqueda del parámetro C para SVM lineal binario.

Al evaluar las prestaciones del modelo, se obtiene un valor de *accuracy* de 0.5. Además, en la Figura 19 se puede observar el classification report de este modelo con todas las prestaciones obtenidas y la matriz de confusión:

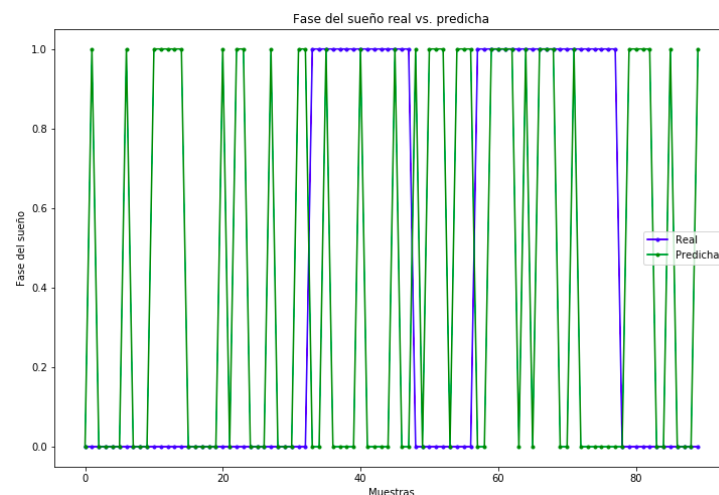
===== Classification Report =====							
	precision	recall	f1-score	support	Resultado del clasificador	0.0	1.0
0.0	0.49	0.52	0.50	137	Clase predicha		
1.0	0.51	0.48	0.50	145	0.0	71	66
accuracy			0.50	282	1.0	75	70
macro avg	0.50	0.50	0.50	282			
weighted avg	0.50	0.50	0.50	282			

**Figura 19.** Classification report y matriz de confusión de SVM lineal binario.

Se han comparado los resultados del clasificador SVM lineal sin la búsqueda de parámetros y con la búsqueda de parámetros. En el primer caso, se ha obtenido un valor de *accuracy* de 0.4574, por lo que se observa que la búsqueda de parámetros mejora ligeramente el rendimiento del clasificador. Además, podemos observar que el modelo no sobreajusta, ya que al evaluar las prestaciones en el subconjunto de Train y compararlas con las obtenidas en el subconjunto de Test, vemos que son similares. Por ello, a pesar de que las prestaciones son bajas, vemos que el modelo no memoriza sino que aprende.

- Accuracy en Train = 0.573
- Accuracy en Test = 0.5

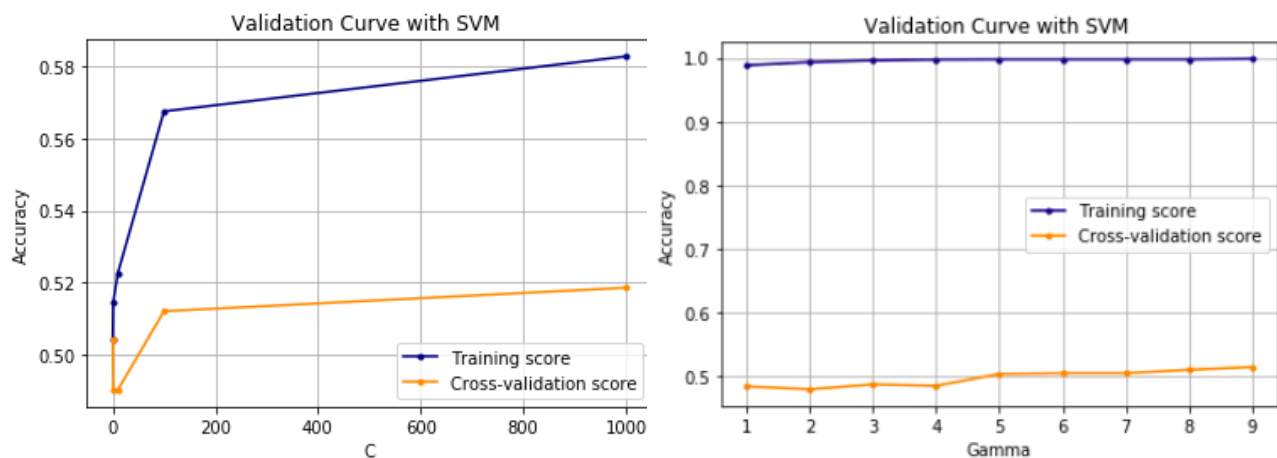
Como medida visual, podemos observar en la Figura 20 las 90 primeras etiquetas del sueño reales frente a las predichas por este clasificador.



**Figura 20.** Etiquetas del sueño reales frente a las predichas por SVM lineal binario.

### 3.1.2 SVM no lineal

En este clasificador se ha utilizado un kernel gaussiano y se han realizado distintas pruebas para estimar los mejores parámetros libres para el modelo. Los parámetros libres a estimar para este clasificador son *gamma* y *C*, con rangos de búsqueda de 1 a 10 para el parámetro *gamma* y de escala logarítmica para *C* (al igual que en el modelo lineal). Debido a la capacidad de computación, se han tenido que estimar por separado los parámetros, es decir, se ha fijado uno de ellos y se han barrido los posibles valores del otro. Este mismo proceso se ha realizado con ambos parámetros. De esta forma, se ha conseguido encontrar los mejores valores para cada uno de los parámetros. Los parámetros óptimos obtenidos para cada uno de los parámetros han sido  $\gamma=9$  y  $C=10$ . En el caso del parámetro *gamma*, al estar en el límite del rango de búsqueda, se ha ampliado el rango hasta 15, pero tras realizar de nuevo la búsqueda, el parámetro óptimo seguía siendo 9. En la Figura 21 se puede observar gráficamente la búsqueda de los parámetros realizada por el *GridSearch*.



**Figura 21.** Búsqueda del parámetro *C* y *gamma* para SVM no lineal binario.

Al evaluar las prestaciones del modelo, se obtiene un valor de *accuracy* 0.4752. Además, en la Figura 22 se puede observar el classification report de este modelo con todas las prestaciones obtenidas y la matriz de confusión:

===== Classification Report =====							
	precision	recall	f1-score	support	Resultado del clasificador	0.0	1.0
0.0	0.48	0.88	0.62	137	Clase predicha		
1.0	0.45	0.09	0.15	145	0.0	121	16
accuracy			0.48	282	1.0	132	13
macro avg	0.46	0.49	0.38	282			
weighted avg	0.46	0.48	0.38	282			



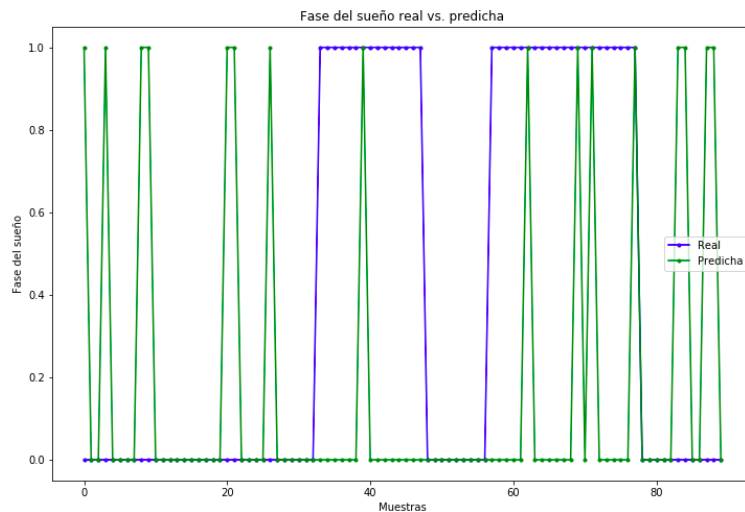
**Figura 22.** *Classification report* y matriz de confusión de SVM no lineal binario.

Al comparar los resultados del clasificador sin la búsqueda de parámetros y con la búsqueda de parámetros, se ha obtenido un valor de accuracy de 0.4574 sin la búsqueda. Estos valores de *accuracy* indican que la búsqueda de parámetros mejora el modelo en un 3.88%.

Al contrario que en el modelo SVM lineal, al evaluar el clasificador no lineal sobre el conjunto de Train y comparar los resultados con los de Test, se observa que este modelo sobreajusta mucho. Por tanto, el modelo SVM no lineal binario no está aprendiendo, sino que está memorizando.

- Accuracy en Train = 0.9978
- Accuracy en Test = 0.4752

Como medida visual, podemos observar en la Figura 23 las 90 primeras etiquetas del sueño reales del subconjunto de Test frente a las etiquetas predichas por este clasificador.



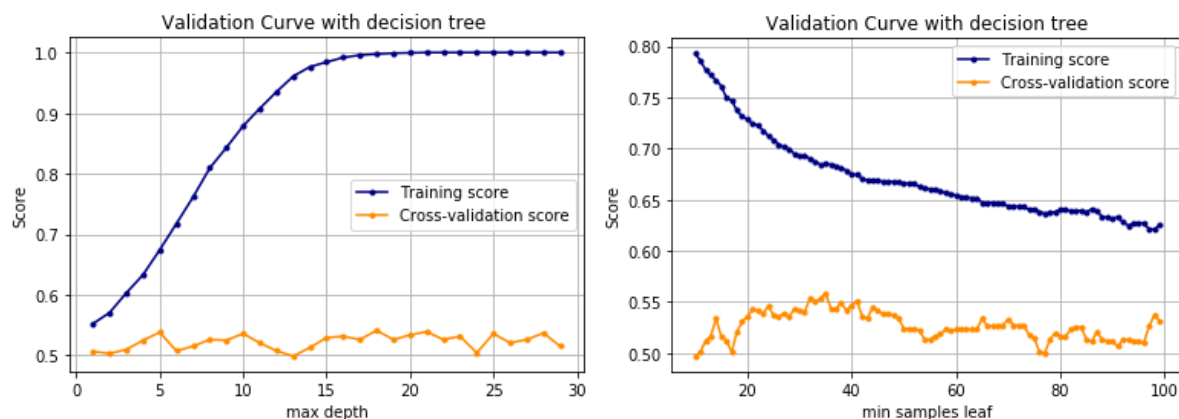
**Figura 23.** Etiquetas del sueño reales frente a las predichas por SVM no lineal binario.

### 3.1.3 Árboles de decisión

Los parámetros libres para los modelos basados en árboles de decisión que se han escogido son máxima profundidad del árbol y mínimo número de muestras en cada nodo, que constituyen una parte del total de parámetros libres configurables en los árboles de decisión. Para ambos parámetros se realiza un barrido de los posibles valores escogidos para elegir aquel que maximice la exactitud (accuracy):

- **Máxima profundidad del árbol:** realizando un barrido de 1 a 30
- **Mínimo número de muestras por nodo:** realizando un barrido de 10 a 100.

Ambos parámetros se han buscado a la par y el conjunto de valores que maximizan la tasa de acierto del clasificador son: máxima profundidad= 7 y mínimo número de muestras por hoja=35. Estos valores no se encontraban en los extremos del rango de búsqueda, por lo que no ha sido necesario realizar una nueva búsqueda. La Figura 24 muestra gráficamente la búsqueda de los parámetros realizada por el GridSearch y el nivel de *overfitting* del modelo. A continuación se representa la curva de validación en la búsqueda individual de cada uno de estos parámetros.



**Figura 24.** Búsqueda del parámetro máxima profundidad y mínimo número de muestras por nodo para árboles de decisión con salida binaria.

A pesar de que este algoritmo tiende al sobreajuste, se aprecia que se consigue evitarlo notablemente gracias a la búsqueda de parámetros libres. Esto lo podemos observar en la menor diferencia en el valor de acierto sobre el conjunto de entrenamiento y evaluación:

- Accuracy en Train =0.6797
- Accuracy en Test = 0.4645

En cuanto a la evaluación del modelo, podemos observar los siguientes resultados:

```

=====Classification Report=====
              precision    recall  f1-score   support

     0.0         0.43      0.31      0.36         137
     1.0         0.48      0.61      0.54         145

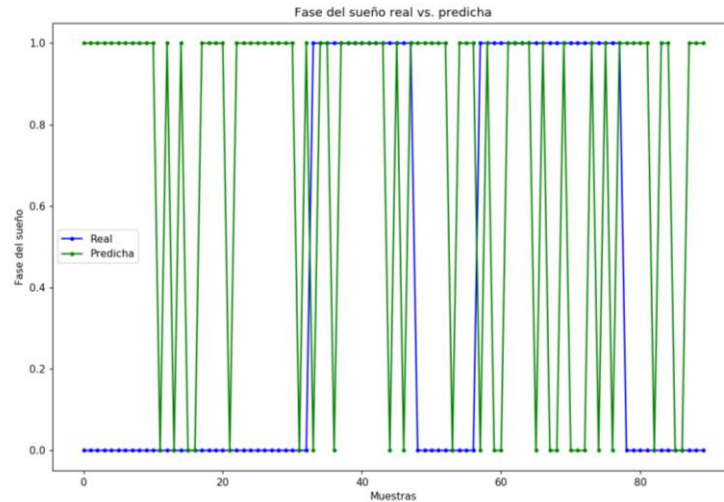
 accuracy                   0.46         282
 macro avg              0.46      0.46      0.45         282
 weighted avg           0.46      0.46      0.45         282

```

Resultado del clasificador	0.0	1.0
Clase predicha		
0.0	42	95
1.0	56	89

**Figura 25.** Prestaciones de árboles de clasificación.

A continuación se muestran representadas las 90 primeras etiquetas del sueño reales del subconjunto de Test frente a las predicciones por este clasificador.



**Figura 26.** Etiquetas del sueño reales frente a las predichas por árboles de decisión binario.

### 3.1.4 Perceptrón multicapa

Para el diseño del clasificador MLP se ha realizado una búsqueda del número de neuronas más adecuado de la capa oculta. El rango de búsqueda ha sido de entre 4 y 20 neuronas, dando saltos numéricos de 3 en 3. Además, para el diseño de este clasificador se ha realizado un ajuste de los siguientes parámetros:

- **hidden\_layer\_sizes:** se estudia el valor óptimo en el rango (4,20).
- **activation:** se utiliza la función por defecto *'relu'*.
- **solver:** se utiliza el método de descenso por gradiente *'sgd'*.
- **learning\_rate** se utiliza una tasa de aprendizaje constante *'constant'*.
- **max\_iter:** se utiliza el valor por defecto 200.
- **shuffle:** se utiliza valor *'True'* para aleatorizar las muestras.
- **early\_stopping:** se utiliza el valor *'True'*.
- **n\_iter\_no\_change:** se utiliza valor 5 ya que tras varias iteraciones se observa que no son necesarias más.
- **validation\_fraction:** se utiliza el valor por defecto 0.1.
- **verbose:** se utiliza el valor *'True'* para ver el entrenamiento y las prestaciones en las distintas pruebas.

La búsqueda del valor óptimo para el número de neuronas de la capa oculta se ha elegido según los valores de pérdida y de *accuracy* obtenidos al entrenar el modelo con las distintas opciones. En todas ellas el valor de *accuracy* en el subconjunto de Train es de 0.5. En la Figura 27 se puede observar la pérdida obtenida en cada una de ellas.

	Loss
4	0.701216
7	0.752764
10	0.992343
13	0.730532
16	0.858328
19	0.966003

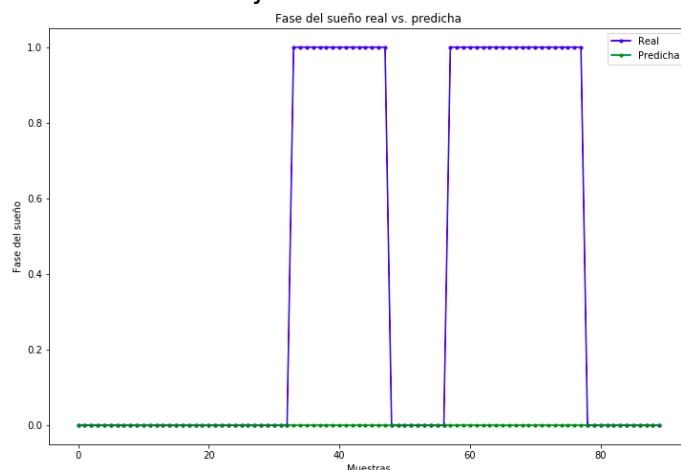
**Figura 27.** Pérdida del modelo MLP binario para el distinto número de neuronas.

Como todos los modelos tenían un valor de *accuracy* de 0.5, se ha elegido el modelo que menor pérdida proporciona. En este caso, se ha elegido 4 como valor óptimo de número de neuronas. Las prestaciones obtenidas sobre el conjunto de Test en este modelo se pueden observar en la Figura 28 y la predicción de clases en la Figura 29.

	Accuracy	Recall	F1-score
4	0.236017	0.485816	0.317693

**Figura 28.** Prestaciones del clasificador MLP binario.

Comparando los resultados de *accuracy* sobre los subconjuntos de Train y Test, podemos observar que el modelo tiende al sobreajuste.

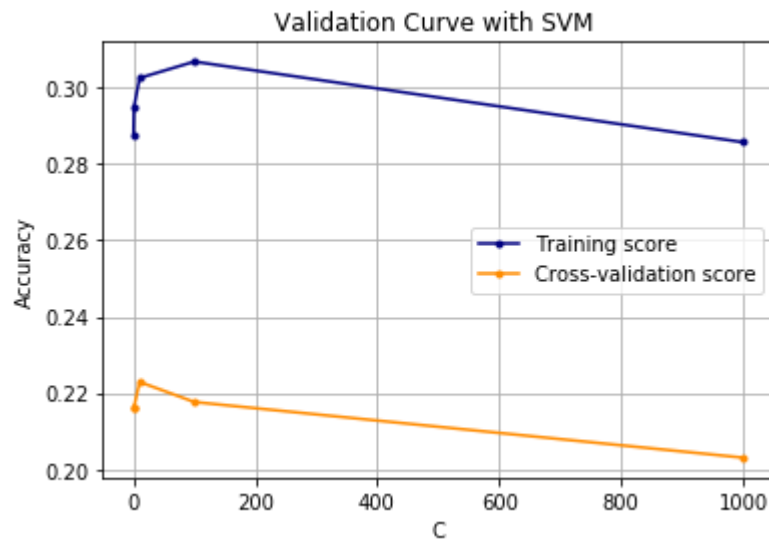


**Figura 29.** Etiquetas del sueño reales frente a las predichas por MLP.

## 3.2 PROBLEMA CLASIFICACIÓN MULTICLASE CON DATOS BALANCEADOS

### 3.2.1 SVM lineal

Al igual que en el clasificador binario, en este clasificador se ha utilizado un kernel lineal y se han realizado distintas pruebas para estimar el mejor parámetro libre 'C' en un rango de escala logarítmica (0.1, 1,10,100,1000). El parámetro óptimo obtenido ha sido C=10. En la Figura 30 se puede observar gráficamente la búsqueda del parámetro C realizada por el GridSearch.



**Figura 30.** Búsqueda del parámetro C para SVM lineal multiclase.

Al evaluar las prestaciones del modelo, se obtiene un valor de *accuracy* de 0.1859. Además, en la Figura 31 se puede observar el *classification report* de este modelo con todas las prestaciones obtenidas y la matriz de confusión:

===== Classification Report =====										
	precision	recall	f1-score	support	Resultado del clasificador	0.0	1.0	2.0	3.0	5.0
0.0	0.22	0.12	0.16	48	Clase predicha					
1.0	0.15	0.42	0.22	43	0.0	6	21	4	2	15
2.0	0.41	0.15	0.22	46	1.0	7	18	3	1	14
3.0	0.33	0.09	0.14	54	2.0	5	19	7	4	11
5.0	0.15	0.18	0.16	51	3.0	1	33	2	5	13
accuracy			0.19	242	5.0	8	30	1	3	9
macro avg	0.25	0.19	0.18	242						
weighted avg	0.25	0.19	0.18	242						

**Figura 31.** *Classification report* y matriz de confusión de SVM lineal multiclase.

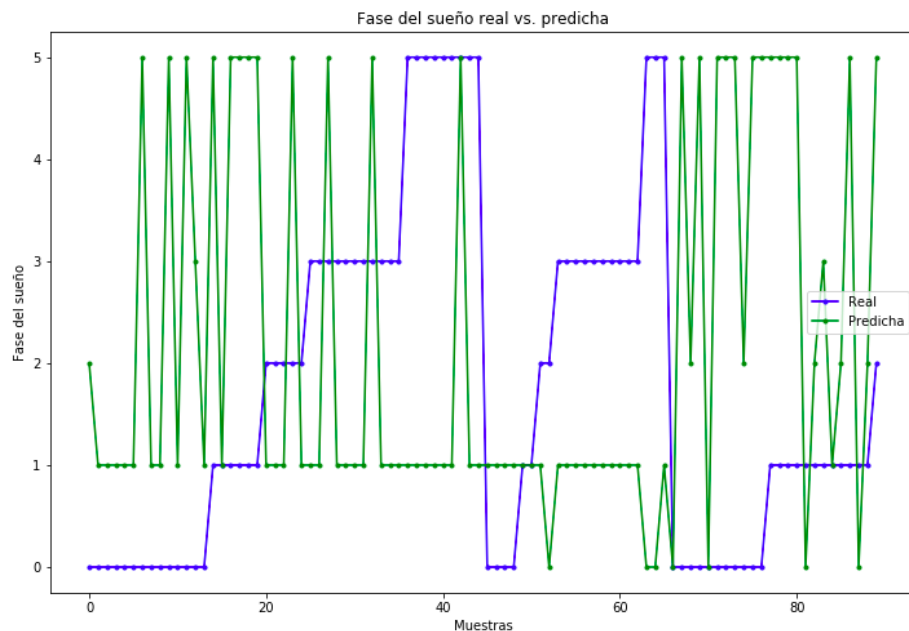
Además, se han comparado los resultados del clasificador SVM lineal sin la búsqueda de parámetros y con la búsqueda de parámetros. En el primer caso, se obtiene el mismo valor de

accuracy de 0.19. Se esperaría que la búsqueda de parámetros favoreciera el rendimiento del clasificador como ya se ha visto en otros clasificadores, pero en este caso no es así.

Podemos observar que el modelo puede sobreajustar un poco, ya que al evaluar las prestaciones en el subconjunto de Train y compararlas con las obtenidas en el subconjunto de Test, vemos que a pesar de su similitud su diferencia es significativa. Al ser las prestaciones obtenidas tan bajas, no se puede afirmar con certeza que haya sobreajuste en el modelo. Por ello, es necesario probar con distintos subconjuntos de datos y distinto número de muestras para obtener conclusiones más contundentes.

- Accuracy en Train = 0.2823
- Accuracy en Test = 0.1859

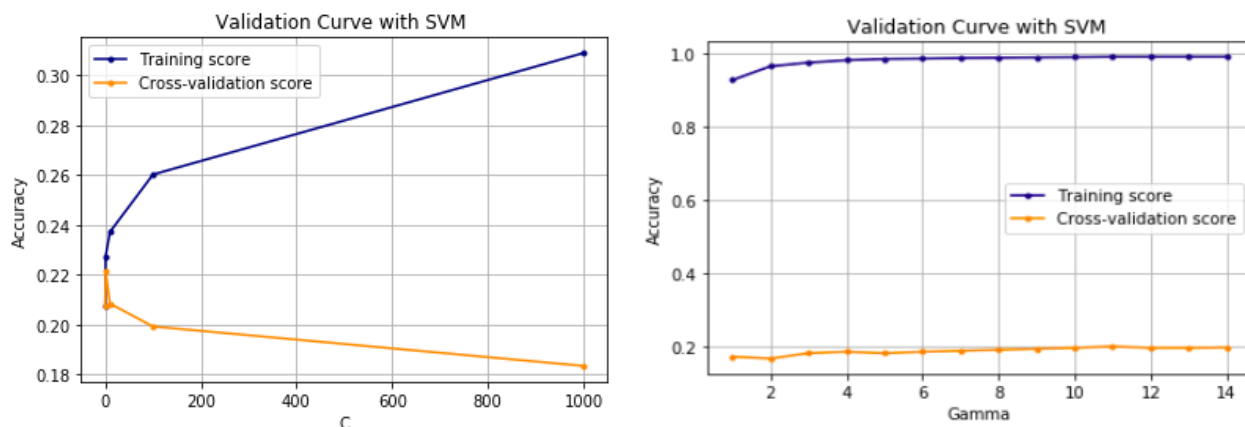
Como medida visual, podemos observar en la Figura 32 las 90 primeras etiquetas del sueño reales del subconjunto de Test frente a las etiquetas predichas por este clasificador.



**Figura 32.** Etiquetas del sueño reales frente a las predichas por SVM lineal multiclase.

### 3.2.2 SVM no lineal

Al igual que en el clasificador SVM no lineal binario, en este clasificador se ha utilizado un kernel gaussiano y se han realizado distintas pruebas para estimar los mejores parámetros libres para el modelo con los mismos rangos de búsqueda. Como ya se ha visto, los parámetros libres a estimar son gamma y C para este clasificador. El proceso seguido es el mismo que el realizado en el clasificador binario. En este caso los mejores valores para cada uno de los parámetros han sido gamma=11 y C=1. En la Figura 32 se puede observar gráficamente la búsqueda de los parámetros realizada por el GridSearch.



**Figura 32.** Búsqueda del parámetro C y gamma para SVM no lineal multiclase.

Al evaluar las prestaciones del modelo, se obtiene un valor de *accuracy* de 0.1736. Además, en la Figura 34 se puede observar el classification report de este modelo con todas las prestaciones obtenidas y la matriz de confusión:

===== Classification Report =====										
	precision	recall	f1-score	support	Resultado del clasificador	0.0	1.0	2.0	3.0	5.0
0.0	0.20	0.02	0.04	48	Clase predicha					
1.0	0.18	0.93	0.30	43	0.0	6	21	4	2	15
2.0	0.00	0.00	0.00	46	1.0	7	18	3	1	14
3.0	0.00	0.00	0.00	54	2.0	5	19	7	4	11
5.0	0.17	0.02	0.04	51	3.0	1	33	2	5	13
accuracy			0.17	242	5.0	8	30	1	3	9
macro avg	0.11	0.19	0.08	242						
weighted avg	0.11	0.17	0.07	242						

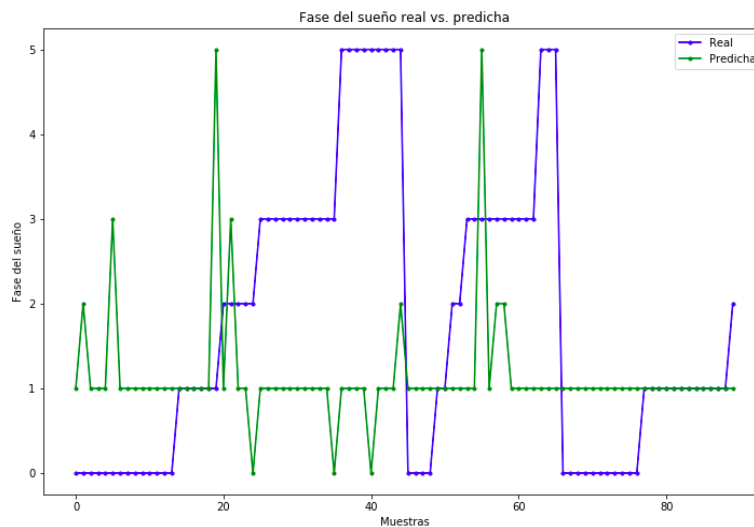
**Figura 34.** Classification report y matriz de confusión de SVM no lineal multiclase.

En este caso, al comparar los resultados del clasificador SVM lineal sin la búsqueda de parámetros y con la búsqueda de parámetros se ha obtenido una mejora en el rendimiento del 27.27% al

realizar la búsqueda de parámetros, ya que sin la búsqueda el valor de accuracy es del 0.1363. Al igual que con el clasificador no lineal binario, al comprobar si el modelo sobreajustaba o no, se ha observado que el modelo sobreajusta en gran medida. Los prestaciones obtenidas han sido:

- Accuracy en Train = 0.9894
- Accuracy en Test = 0.1736

Como medida visual, podemos observar en la Figura 35 las 90 primeras etiquetas del sueño reales del subconjunto de Test frente a las etiquetas predichas por este clasificador.



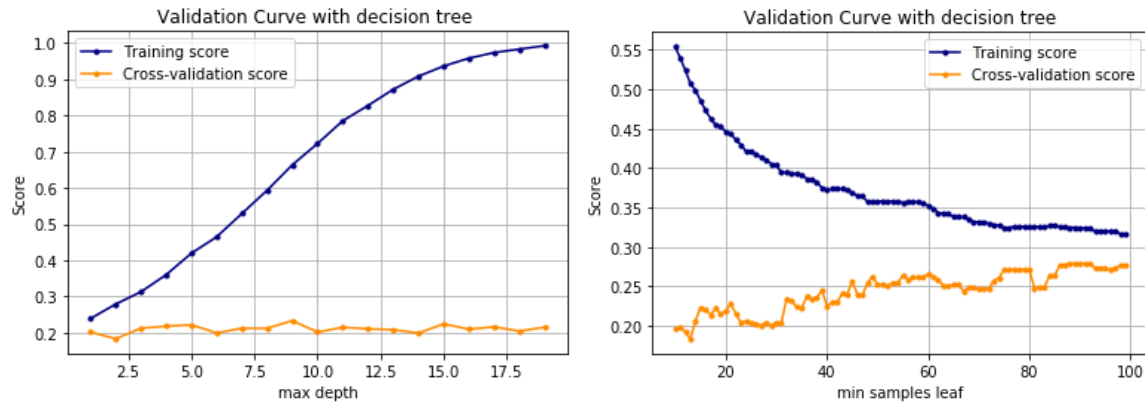
**Figura 35.** Etiquetas del sueño reales frente a las predichas por SVM no lineal multiclase.

### 3.2.3 Árboles de decisión

Este clasificador realiza la misma búsqueda de parámetros libres que la realizada en árboles de decisión con salida binaria. El conjunto de parámetros libres que maximizan la tasa de acierto del clasificador son: máxima profundidad= 4 y mínimo número de muestras por hoja=88. Ambos valores no se encontraban en los extremos del rango de búsqueda, por lo que no ha sido necesario realizar una nueva búsqueda.

La Figura 36 muestra gráficamente la búsqueda individual de cada uno de los parámetros seleccionados.





**Figura 36.** Búsqueda de los parámetros máxima profundidad y mínimo de muestras por hoja para árboles de clasificación multiclase.

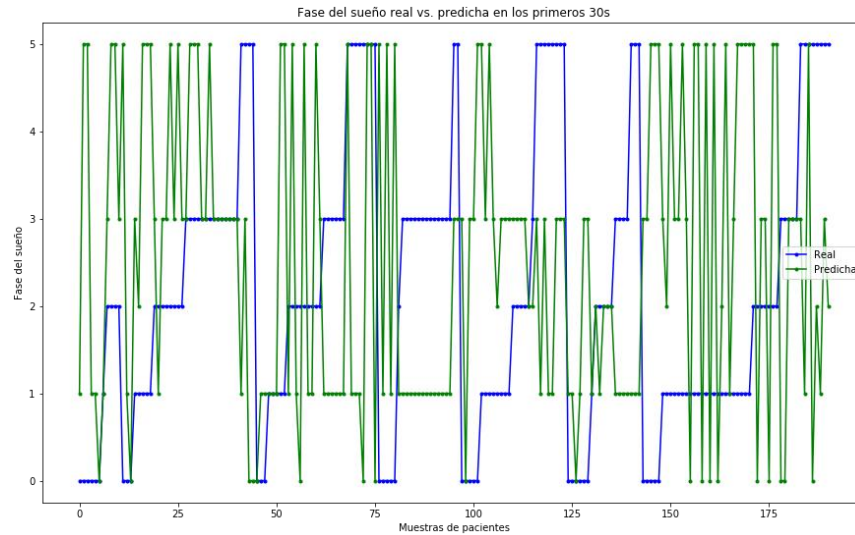
Respecto al sobreajuste del modelo, no se aprecia una enorme diferencia entre la accuracy del conjunto Train y del conjunto Test, pero sí mejorable. Estos valores son:

- Accuracy en Train = 0.3232
- Accuracy en Test = 0.1736

En cuanto a la evaluación del modelo, los resultados pueden apreciarse en la siguiente figura:

====Classification Report====										
	precision	recall	f1-score	support	Resultado del clasificador	0.0	1.0	2.0	3.0	5.0
0.0	0.16	0.19	0.17	48	Clase predicha					
1.0	0.17	0.33	0.23	43	0.0	5	11	0	7	10
2.0	0.00	0.00	0.00	46	1.0	4	6	4	11	18
3.0	0.15	0.09	0.11	54	2.0	3	7	5	14	10
5.0	0.20	0.27	0.23	51	3.0	2	23	1	13	4
accuracy			0.17	242	5.0	5	12	2	10	4
macro avg	0.14	0.18	0.15	242						
weighted avg	0.14	0.17	0.15	242						

**Figura 37.** Classification report y matriz de confusión de árboles de clasificación multiclase.



**Figura 38.** Etiquetas del sueño reales frente a las predichas por árboles de decisión multiclase.

### 3.2.4 Perceptrón multicapa

El diseño de este clasificador es igual que el diseño del clasificador binario. Se ha realizado una búsqueda del número de neuronas de la capa oculta más adecuado con el mismo rango que en el clasificador binario y el mismo ajuste de los parámetros.

La búsqueda del valor óptimo para el número de neuronas de la capa oculta se ha elegido según los valores de pérdida y de *accuracy* obtenidos al entrenar el modelo con las distintas opciones. En todas ellas el valor de *accuracy* sobre el subconjunto de Train es entorno a 0.2. En la Figura 38 se puede observar la pérdida obtenida en cada una de ellas.

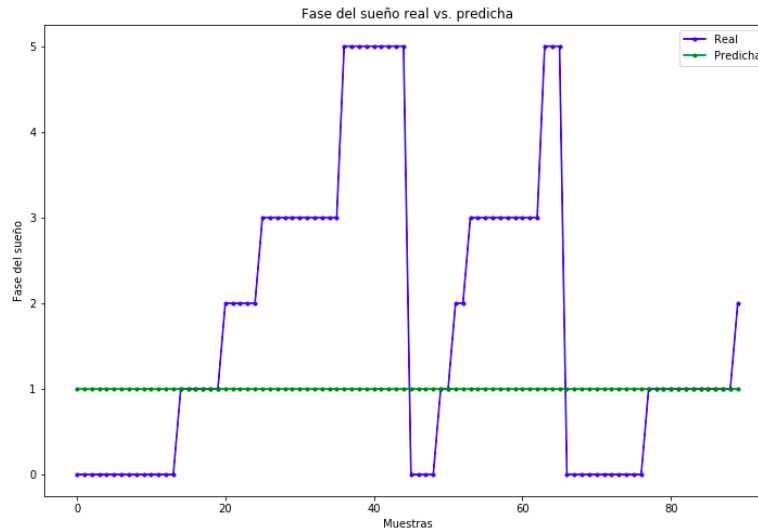
	Loss
<b>4</b>	2.05921
<b>7</b>	1.88633
<b>10</b>	1.71382
<b>13</b>	1.92514
<b>16</b>	46.2994
<b>19</b>	3.33353

**Figura 39.** Pérdida del modelo MLP multiclase para el distinto número de neuronas.

Se ha elegido el modelo que mayor *accuracy* proporciona y una de las menores pérdidas. En este caso, se ha elegido 7 como valor óptimo de número de neuronas ya que es el de mayor *accuracy*. Las prestaciones obtenidas en este modelo sobre el subconjunto de Test se pueden observar en la Figura 40 y la predicción de clases en la Figura 41.

	Accuracy	Recall	F1-score
4	0.0315723	0.177686	0.0536175

**Figura 40.** Prestaciones del clasificador MLP multiclase.



**Figura 41.** Etiquetas del sueño reales frente a las predichas por MLP..

A pesar de decidir utilizar los clasificadores con datos balanceados para que se reconozcan todas las clases, se ha observado que las prestaciones en los clasificadores binarios mejoran notablemente con los datos sin balancear. Por ello, se ha decidido evaluar también estos clasificadores con datos desbalanceados. Para esta ejecución se ha realizado la división de los subconjunto de Train y Test con las mismas proporciones utilizadas hasta ahora (78%-22%) y se han utilizado 100 ventanas por sujeto.

### 3.3 PROBLEMA CLASIFICACIÓN BINARIA CON DATOS SIN BALANCEAR

Para cada algoritmo, se ha realizado la mismo búsqueda de parámetros libres descrita en los anteriores apartados. A continuación se muestran los resultados (classification report, matriz de confusión y representación de las primeras 90 predicciones frente a su valor real) de los diferentes algoritmos con datos desbalanceados.

#### 3.3.1 SVM lineal

Con este clasificador se obtienen los mejores resultados de predicción. Se observa en la Figura 42, una tasa de acierto de 0.68, es decir, clasificaría correctamente casi el 70% de los casos. Además, como se puede ver en la matriz de confusión, se clasifican correctamente un mayor número de muestras de la clase 1 que de la clase 0.

===== Classification Report =====							
	precision	recall	f1-score	support	Resultado del clasificador	0.0	1.0
0.0	0.41	0.05	0.09	222	Clase predicha		
1.0	0.69	0.97	0.80	478	0.0	11	211
accuracy			0.68	700	1.0	16	462
macro avg	0.55	0.51	0.45	700			
weighted avg	0.60	0.68	0.58	700			

Figura 42. Classification report y matriz de confusión de SVM lineal binario con datos desbalanceados.

Efectivamente, en la Figura 43 también podemos observar que el clasificador predice mejor la fase de sueño profundo (etiqueta 1) que de sueño ligero (etiqueta 0).

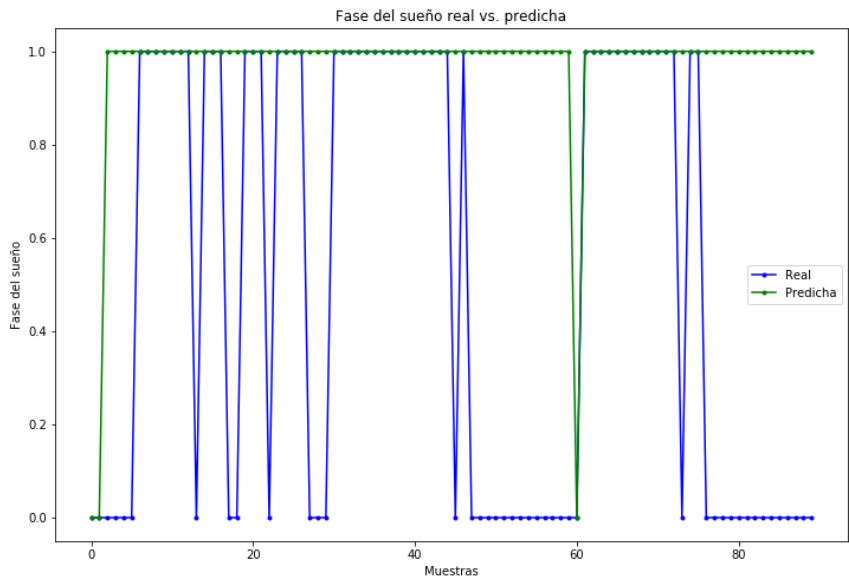
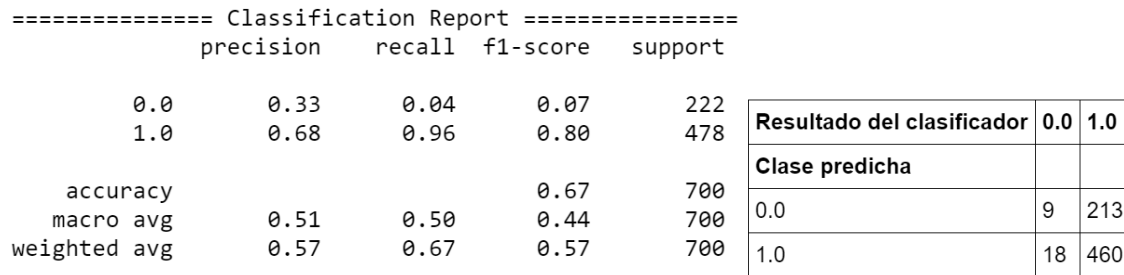


Figura 43. Etiquetas del sueño reales frente a las predichas por SVM lineal con datos desbalanceados.

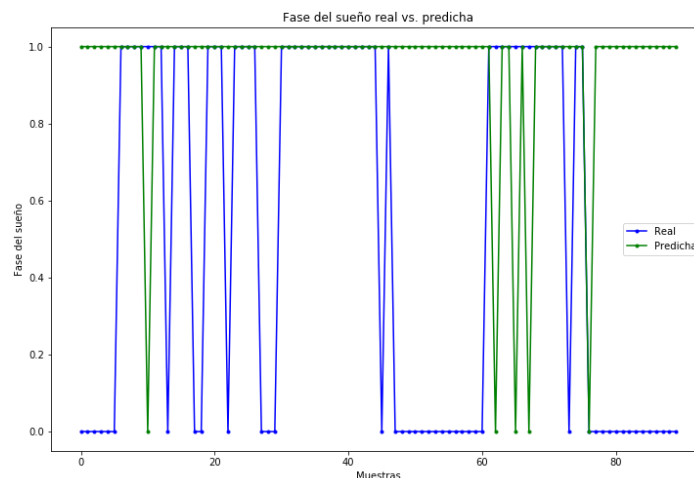
3.3.2 SVM no lineal

Este modelo presenta unos resultados similares respecto al anterior. Se puede observar en la figura 44 una accuracy de 0.67 y unas mejores prestaciones obtenidas en la clasificación del sueño profundo (etiqueta 1). Como se puede observar en la matriz de confusión se clasifican correctamente 460 etiquetas de la clase 1 frente a 9 etiquetas de la clase 0.



**Figura 44.** *Classification report* y matriz de confusión de SVM no lineal binario con datos desbalanceados.

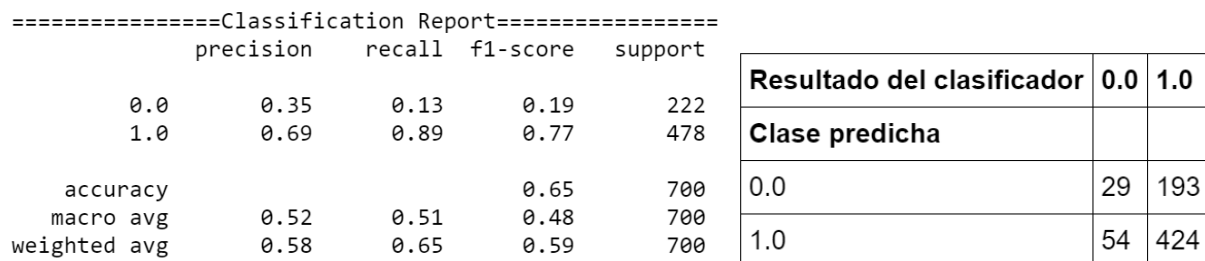
En la Figura 45 también se puede observar que el clasificador tiende a predecir la clase 1.



**Figura 45.** Etiquetas del sueño reales frente a las predichas por SVM no lineal binario con datos desbalanceados.

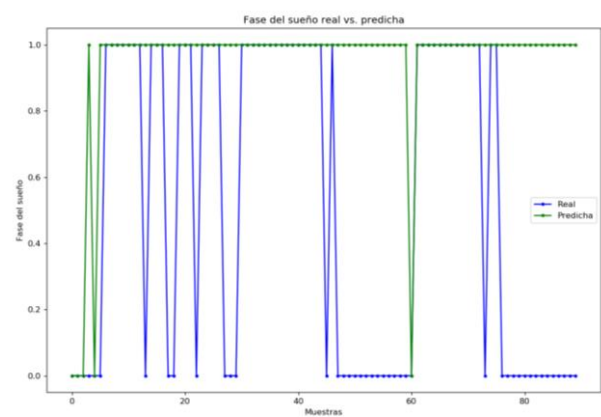
### 3.3.3 Árboles de decisión

Según se observa en el *classification report* representado en la Figura 46, la tasa de acierto obtenida utilizando árboles de decisión es de 0,65. De esta forma, se observa una pequeña mejoría en las prestaciones de la clasificación de sueño ligero (etiqueta 0), pero tan marcadas como son las obtenidas en la predicción de sueño profundo (etiqueta 1).



**Figura 46.** *Classification report* y matriz de confusión de árbol decisión binario con datos desbalanceados.

Observando las 90 primeras predicciones del modelo de clasificación, podemos observar que el clasificador tiende a clasificar sueño profundo (etiqueta 1) a pesar de tratarse de sueño ligero(etiqueta 0)



**Figura 47.** Etiquetas del sueño reales frente a las predichas por árboles decisión con datos desbalanceados.

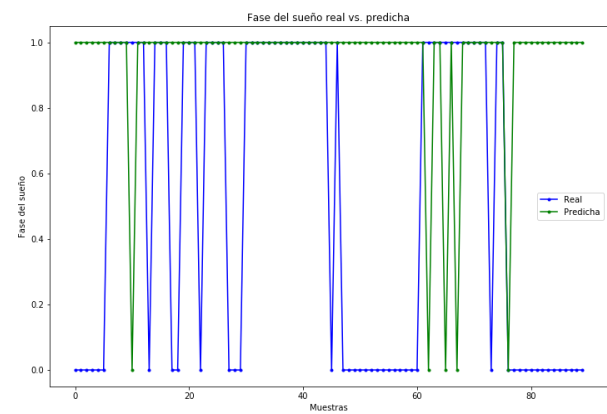
### 3.3.4 Perceptrón multicapa

Para el modelo perceptrón multicapa, como se expone en la Figura 48, la tasa de acierto es de 0.47, la sensibilidad es de 0.68 y el F1-score de 0.55 aproximadamente.

	Accuracy	Recall	F1-score
4	0.466294	0.682857	0.554169

**Figura 48.** Prestaciones del clasificador MLP binario.

Al igual que el resto de clasificadores mencionados previamente, este modelo tiende a clasificar erróneamente la fase de sueño ligero (etiqueta 0); esto se puede observar en las 90 primeras predicciones del clasificador representadas en la Figura 49.



**Figura 49.** Etiquetas del sueño reales frente a las predichas por MLP con datos desbalanceados.

En la *Tabla 4* se recogen las **prestaciones obtenidas** de los distintos modelos utilizados para el problema binario y en la *Tabla 5* los obtenidos para el problema multiclase.

CLASIFICADORES	ACCURACY	SOBREAJUSTE
<b>Problema binario datos balanceados</b>		
SVM lineal	0.5	No
SVM no lineal	0.4752	Si
Árboles	0.4645	Ligera tendencia
MLP	0.236	Ligera tendencia
<b>Problema binario datos desbalanceados</b>		
SVM lineal	0.6757	No
SVM no lineal	0.67	Si
Árboles	0.6471	Ligera tendencia
MLP	0.4662	Ligera tendencia

Accuracy Train: 0.9978  
Accuracy Test: 0.4752

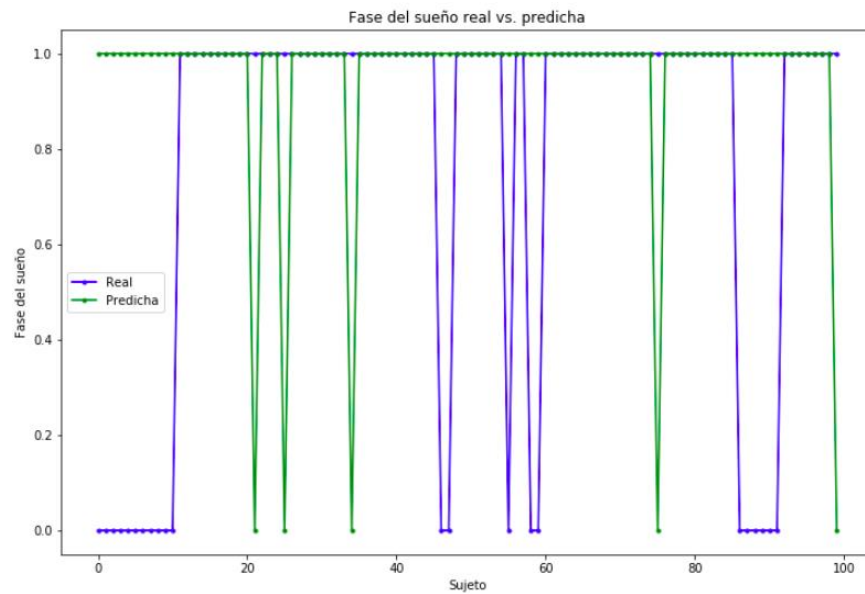
**Tabla 4.** Comparación de las prestaciones de los distintos modelos de clasificación binaria.

CLASIFICADORES	ACCURACY	SOBREAJUSTE
<b>Problema multiclase datos balanceados</b>		
SVM lineal	0.1859	No
SVM no lineal	0.1736	Si
Árboles	0.1736	Ligera tendencia
MLP	0.03	Ligera tendencia
<b>Problema multiclase datos desbalanceados</b>		
SVM lineal	0.4952	No
SVM no lineal	0.4371	Si
Árboles	0.4371	No
MLP	0.1936	Ligera tendencia

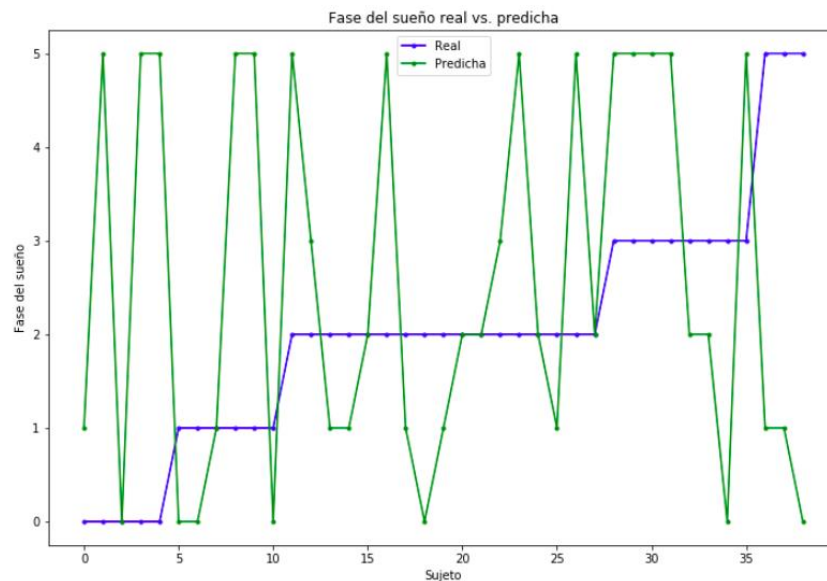
No se reconocen todas las clases

**Tabla 5.** Comparación de las prestaciones de los distintos modelos de clasificación multiclase.

Para el mejor modelo de cada problema, se ha realizado una **predicción del sueño de un paciente** aleatorio del conjunto de Test. En la *Figura 50* se puede observar la predicción de un paciente con SVM lineal para el problema binario y en la *Figura 51* la predicción de un paciente con el mismo método para el problema multiclase.



**Figura 50.** Predicción de un paciente completo con SVM lineal para el problema binario.



**Figura 51.** Predicción de un paciente completo con SVM lineal para el problema multiclase.



## Capítulo 4

En este último capítulo se analizan los resultados obtenidos durante la realización del proyecto y se proponen algunas líneas de investigación que lo continúen.

### Conclusiones y líneas futuras

A lo largo del proyecto se ha realizado un estudio de la predicción del sueño, utilizando para ello técnicas de aprendizaje automático supervisado. Concretamente, el contexto del estudio se ha centrado en la predicción del sueño a partir de variables extraídas mediante un dispositivo *wearable*. Para ello, se ha hecho uso de una base de datos extraída de *Physionet* que recoge información sobre 31 sujetos a los que se les realizó un estudio polisomnográfico y que habían llevado dicho *wearable* durante un periodo de tiempo de entre 7 y 14 días.

En el desarrollo del proyecto se han aplicado distintas herramientas de computación paralela que han permitido optimizar el tiempo de ejecución de distintos procesos. Se ha construido un *Pool* del paquete *multiprocessing* para optimizar el tiempo de la extracción de características de la base de datos y de la creación de una nueva base de datos con estas características. Por otro lado, se ha utilizado la plataforma Google Colab para optimizar la ejecución de los distintos modelos de clasificación.

Los resultados obtenidos de los clasificadores indican que el conjunto de datos utilizado para entrenarlos es insuficiente para resolver de manera fiable el problema de predicción planteado. La cantidad de muestras utilizadas dificulta el diseño de un algoritmo que pueda clasificar correctamente el sueño y sus etapas. A pesar de haber aplicado técnicas de paralelización, sigue presente el problema de la baja capacidad computacional de los algoritmos de aprendizaje. Esta dificultad imposibilita utilizar un mayor número de muestras en el entrenamiento y, por tanto, acentúa el problema de clasificar erróneamente las diferentes etapas del sueño.

A pesar de las bajas prestaciones de los modelos, el mejor clasificador para resolver el problema binario (clasificar vigilia y sueño ligero frente al sueño profundo) se obtiene con el modelo SVM lineal utilizando datos desbalanceados. Para este modelo se consigue clasificar correctamente casi el 70% de los casos (*accuracy* 0.67), aunque se aprecia una ligera tendencia a clasificar el estado de sueño profundo (etiqueta 1). Por otro lado, para el problema multiclase (clasificar diferentes etapas del sueño), las mejores prestaciones se obtienen con SVM lineal utilizando datos balanceados. En este caso, se consigue que el clasificador identifique todas las etapas del sueño, aunque solo clasifica bien el 20% de los casos (*accuracy* 0.19), aunque al igual que en el caso binario hay una ligera tendencia a clasificar las fases del sueño 1 y 5. Ambos modelos tienen

mejores prestaciones que el resto de los modelos utilizados y una capacidad de aprendizaje que no tiende al sobreajuste.

Por otro lado, al observar los resultados de otros clasificadores, podemos extraer algunas conclusiones relacionadas con los modelos SVM no lineal y MLP. Los modelos SVM no lineales tienden, en gran medida, al sobreajuste de los datos del subconjunto de entrenamiento. En el caso de los clasificadores de MLP observamos que el modelo tiene tendencia a predecir una única clase, independientemente de trabajar sobre el problema binario o multiclase.

En conclusión, los resultados del análisis predictivo se encuentran todavía lejos del diseño de un clasificador funcional de utilidad clínica. Esto se puede deber a diferentes motivos: (1) insuficiente número de características que aporten información relevante para la predicción del sueño y sus distintas etapas, (2) reducido tamaño del conjunto de datos de entrenamiento y evaluación debido a la baja capacidad computacional; (3) incorrecto agrupamiento de clases del sueño.

Como futuras líneas de investigación para continuar el desarrollo de este proyecto, se proponen:

- Extraer y añadir una variable relacionada con el patrón circadiano del usuario. Esta característica se podría crear con la información disponible anterior al estudio polisomnográfico utilizando, por ejemplo, la información de la variable *steps* (conteo de pasos). De esa forma, se podría simular el ritmo circadiano del usuario.
- Aplicar más herramientas de computación paralela que permitan optimizar el tiempo de ejecución de los clasificadores. De esta forma, se podría aumentar la cantidad de muestras utilizadas para entrenar los modelos y mejorar así sus rendimientos.
- Estudio y diseño de nuevos algoritmos de predicción. Una de las posibilidades que se proponen es utilizar métodos de clasificación no supervisada con los datos anteriores al estudio polisomnográfico.

# BIBLIOGRAFÍA

1. Olivia Walch, Yitong Huang, Daniel Forger, Cathy Goldstein, Sleep stage prediction with raw acceleration and photoplethysmography heart rate data derived from a consumer wearable device, *Sleep*, Volume 42, Issue 12, December 2019, zsz180. Disponible en: <https://doi.org/10.1093/sleep/zsz180>
2. Repositorio GitHub. Proyecto '*Sleep stage prediction with raw acceleration and photoplethysmography heart rate data derived from a consumer wearable device*', December 2019. Disponible en: [https://github.com/ojwalch/sleep\\_classifiers](https://github.com/ojwalch/sleep_classifiers)
3. Repositorio GitHub. Proyecto '*Predicción del sueño con wearable*', 2020. Disponible en: <https://github.com/Noemiglois/SleepAnalysisWearable>
4. Walch, O. (2019). Motion and heart rate from a wrist-worn wearable and labeled sleep from polysomnography (version 1.0.0). PhysioNet. Disponible en: <https://doi.org/10.13026/hmhs-py35>.
5. Miriam R Waldeck y Michael I Lambert (2000). *Frecuencia Cardíaca durante el Sueño: Implicaciones para el Monitoreo del Nivel de Entrenamiento*. PubliCE. Última vez accedido: 24-04-202. Disponible en: <https://g-se.com/frecuencia-cardiaca-durante-el-sueno-implicaciones-para-el-monitoreo-del-nivel-de-entrenamiento-571-sa-457cfb271613db>
6. 'Las máquinas de vectores de soporte para identificación en línea'. Última vez accedido: 17-04-2020. Disponible en: <https://ctrl.cinvestav.mx/~yuw/pdf/MaTesJAR.pdf>
7. Clasificador SVM, Scikit-learn. Última vez accedido: 22-04-2020. Disponible en: <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>
8. Bignú, Agustín. Clasificación de dispositivo médico utilizando diferentes técnicas de Machine Learning. Última vez accedido: 22-04-2020. Disponible en: <http://stening.blog/clasificacion-de-dispositivo-medico-utilizando-diferentes-tecnicas-de-machine-learning/>
9. Clasificador árboles de decisión, Scikit-learn. Última vez accedido: 23-04-2020. Disponible en: <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>
10. Clasificador MLP, Scikit-learn. Última vez accedido: 23-04-2020. Disponible en: [https://scikit-learn.org/stable/modules/generated/sklearn.neural\\_network.MLPClassifier.html](https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html)

[learn.org/stable/modules/generated/sklearn.neural\\_network.MLPClassifier.html#sklearn.neural\\_network.MLPClassifier](https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network.MLPClassifier)

11. Aplicación de árboles de clasificación a la detección precoz de abandono en los estudios universitarios de administración y dirección de empresas - Scientific Figure on ResearchGate. Última vez accedido: 23-04-2020. Disponible en: [https://www.researchgate.net/figure/Figura-1-Ejemplo-de-grafo-que-representa-un-arbol-de-clasificacion\\_fig1\\_324509645](https://www.researchgate.net/figure/Figura-1-Ejemplo-de-grafo-que-representa-un-arbol-de-clasificacion_fig1_324509645)
12. *Classification report*, Scikit-learn . Última vez accedido: 26-04-2020. Disponible en: [https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification\\_report.html](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html)
13. '16.6. multiprocessing - Process-based "threading" interface'. (n.d.). Última vez accedido: 17-04-2020. Disponible en: <https://docs.python.org/2/library/multiprocessing.html>
14. 'A High Performance Python Compiler'. (n.d.). Última vez accedido: 17-04-2020. Disponible en: <http://numba.pydata.org/>
15. 'C-Extensions for Python'. (n.d.). Última vez accedido: 17-04-2020. Disponible en: <https://cython.org/>


# ANEXOS

## ANEXO 'TIEMPOS'

En la siguiente tabla se muestran los tiempos obtenidos en una ejecución al paralelizar.

PROCESOS	TIEMPO
<b>Extracción de características</b>	
Sin Pool	56.32 minutos
Con Pool	31.10 minutos
<b>Búsqueda de C en SVM lineal</b>	
Sin GPU Google Colab	6.18 minutos
Con GPU Google Colab	4.46 minutos

## ANEXO 'CLASIFICADORES EN GOOGLE COLABORATORY'

 Clasificador SVM lineal binario.ipynb ☆

Archivo Editar Ver Insertar Entorno de ejecución Herramientas Ayuda [Se han guardado todos los cambios](#)

+ Código + Texto

<>

PREDICIÓN DEL SUEÑO CON SVM LINEAL BINARIO

Noemi González, Roberto Holgado y Carmen Plaza seco

Índice de contenidos

En este jupyter notebook se lleva a cabo la construcción de un clasificador SVM lineal de salida binaria para predecir el estado de vigilia y sueño de distintos sujetos.

1. Clasificador SVM lineal Binario
  - 1.1 Preparación datos de entrada y salida
  - 1.2 Creación del modelo y selección parámetros libres
  - 1.3 Búsqueda de parámetros con validación cruzada
  - 1.4 Evaluación del modelo
  - 1.5 Visualización de resultados

```
[1] import os
import numpy as np
import pandas as pd
from random import sample
import statistics as stats
from sklearn.svm import SVC
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import validation_curve
from sklearn.model_selection import RandomizedSearchCV
```

▼ 1. SVM LINEAL BINARIO

En este clasificador, se ha utilizado el método SVM lineal para clasificar la vigilia o sueño ligero del sueño profundo. Es decir, se ha construido un clasificador SVM con salida binaria: 0 y 1.

## ▼ 1.1 Preparación datos de entrada y salida

```
[2] from google.colab import drive
     drive.mount('/content/drive', force_remount=True)
```

↳ Mounted at /content/drive

```
[3] filename1="/content/drive/My Drive/4ºCARRERA/SEGUNDO CUATRI/PROCESADO MASIVO/TRABAJO/CODIGO/Train_binary.csv"
     filename2="/content/drive/My Drive/4ºCARRERA/SEGUNDO CUATRI/PROCESADO MASIVO/TRABAJO/CODIGO/Test_binary.csv"
```

```
[4] Train_bin= pd.read_csv(filename1)
     Test_bin= pd.read_csv(filename2)
```

```
[5] X_train_bin=Train_bin.drop(['Etiquetas multiclase', 'Etiquetas binarias','ID'], axis = 1)
     y_train_bin=Train_bin['Etiquetas binarias']

     X_test_bin=Test_bin.drop(['Etiquetas multiclase', 'Etiquetas binarias','ID'], axis = 1)
     y_test_bin=Test_bin['Etiquetas binarias']
```

## ▼ 1.2 Creación del modelo y selección parámetros libres

```
[6] clf = SVC(kernel='linear', decision_function_shape='ovr' )
     modelo_base=clf.fit(X_train_bin,y_train_bin)
```

```
[7] clf
```

↳ SVC(C=1.0, break\_ties=False, cache\_size=200, class\_weight=None, coef0=0.0, decision\_function\_shape='ovr', degree=3, gamma='scale', kernel='linear', max\_iter=-1, probability=False, random\_state=None, shrinking=True, tol=0.001, verbose=False)

### 1.3 Búsqueda de parámetros con validación cruzada

A continuación, se realiza la búsqueda del parámetro C. Se hace un 'barrido' de los posibles valores del parámetro para elegir aquel que maximice la exactitud (accuracy).

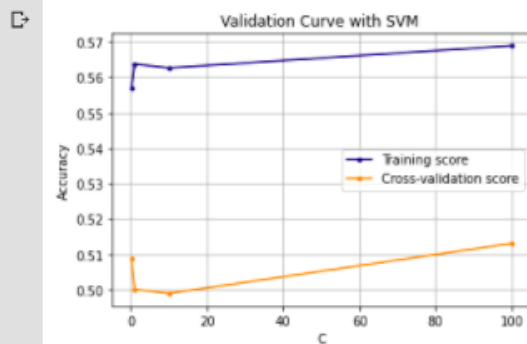
Tras varias iteraciones se ha visto que el mejor parámetro es C=100 para el clasificador SVM lineal binario.

```
[8] Cs=[0.1,1,10,100]
    param_grid = {'C': Cs}
    svc_grid = GridSearchCV(clf, param_grid, scoring='accuracy',cv=5,n_jobs=-1)
    svc_grid.fit(X_train_bin, y_train_bin)
    best_param=svc_grid.best_params_

[9] # Clasificador con los mejores parámetros
    svc_clf=svc_grid.best_estimator_

train_scores, test_scores = validation_curve(clf, X_train_bin, y_train_bin, param_name="C", param_range=Cs,cv=5)
train_scores_mean = np.mean(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
plt.title("Validation Curve with SVM ")
plt.xlabel('C')
plt.ylabel("Accuracy")

plt.plot(Cs, train_scores_mean, label="Training score",color="navy",marker='.')
plt.plot(Cs, test_scores_mean, label="Cross-validation score",color="darkorange",marker='.')
plt.grid()
plt.legend(loc="best")
plt.show()
```



```
[11] print("El valor más óptimo para el parámetro a estimar es:",best_param)

El valor más óptimo para el parámetro a estimar es: {'C': 100}

[12] print("Tasa de acierto del modelo base:",modelo_base.score(X_test_bin,y_test_bin))
    print("Tasa de acierto del modelo con la búsqueda de parámetros:",svc_clf.score(X_test_bin,y_test_bin))
    print('Mejora del {:.2f}%'.format( 100 * (svc_clf.score(X_test_bin,y_test_bin) - modelo_base.score(X_test_bin,y_test_b

Tasa de acierto del modelo base: 0.4574468085106383
Tasa de acierto del modelo con la búsqueda de parámetros: 0.5
Mejora del 9.30%.
```

## 1.4 Evaluación del modelo

```
[13] predict = svc_clf.predict(X_test_bin)

acc_test= svc_clf.score(X_test_bin, y_test_bin).round(4)
acc_train= svc_clf.score(X_train_bin, y_train_bin).round(4)
print('La exactitud para el modelo de SVM en el conjunto de TEST es:',acc_test,'%\n')
print('La exactitud para el modelo de SVM en el conjunto de TRAIN es:',acc_train,'%\n')

print("=====Classification Report=====")
print(classification_report(y_test_bin, predict))
```

```
La exactitud para el modelo de SVM en el conjunto de TEST es: 0.5 %
La exactitud para el modelo de SVM en el conjunto de TRAIN es: 0.573 %
```

```
=====Classification Report=====
      precision    recall  f1-score   support

    0.0         0.49      0.52      0.50        137
    1.0         0.51      0.48      0.50        145

 accuracy          0.50
 macro avg         0.50      0.50      0.50        282
 weighted avg      0.50      0.50      0.50        282
```

```
[14] matriz_confusion=pd.crosstab (y_test_bin,predict, rownames=['Clase predicha'], colnames=['Resultado del clasificador'])
matriz_confusion
```

```
Resultado del clasificador  0.0  1.0
Clase predicha
0.0                71   66
1.0                75   70
```

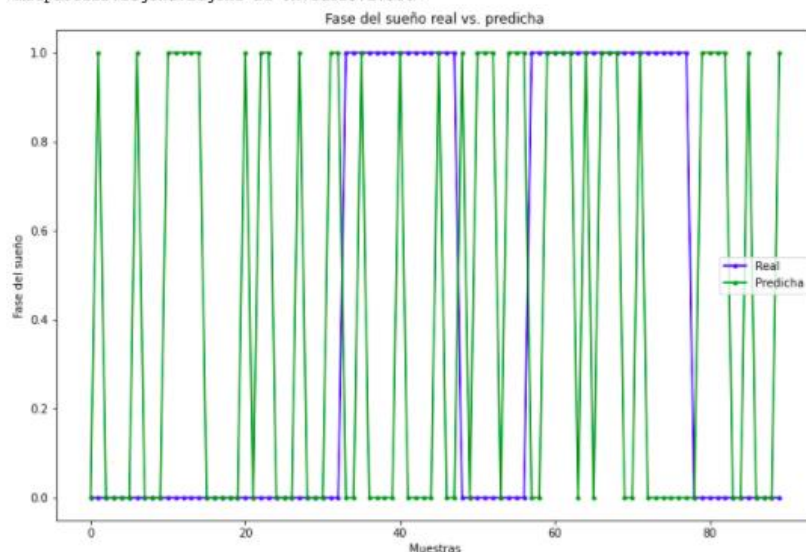
## 1.5 Visualización de resultados

A continuación se muestra un ejemplo de las fases del sueño predichas frente a las reales. Para facilitar esta visualización, se muestran la mitad de la muestras del subconjunto de test.

```
[15] plt.figure(figsize=(12,8))
plt.plot(y_test_bin[0:90],color='blue', marker='.',label='Real')
plt.plot(predict[0:90],color='green', marker='.',label='Predicha')

plt.title('Fase del sueño real vs. predicha')
plt.xlabel('Muestras')
plt.ylabel('Fase del sueño')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fd8257a7588>
```






## Residuos

```
[16] residuos= y_test_bin - predict
plt.figure(figsize=(12,8))
plt.plot(residuos[0:90],color='purple', marker='.',label='Residuos')

plt.title('RESIDUOS')
plt.xlabel('Muestras')
plt.ylabel('Fase del sueño')
plt.legend()
```

 <matplotlib.legend.Legend at 0x7fd82578bc88>

