Monitorización de la actividad del hogar con Aprendizaje Automático

David Cabornero, José Manuel Chacón, Mario García, Alejandro Santorum

Escuela Politécnica Superior - Universidad Autónoma de Madrid

Abstract. Se proponen varios métodos para la detección de estímulos (plátano y vino) mediante varios sensores fisicoquímicos. El objetivo es predecir con la mayor precisión posible dichos estímulos con datos en tiempo real. El artículo resume el proceso iterativo llevado a cabo por los autores durante el proyecto. A lo largo de las distintas fases del mismo, se han empleado diferentes modelos de Aprendizaje Automático y varias técnicas de tratamiento de datos, llegando a conseguir un rendimiento entorno al 85-90%.

1 Introducción

El problema a analizar es el siguiente: una nariz electrónica toma muestras del ambiente cada poco tiempo mediante sus sensores fisicoquímicos [1]. Mientras está en funcionamiento puede ser sometida a estímulos de vino o de plátano. El objetivo es distinguir eficazmente cuándo hay estímulo y qué estímulo es, y la respuesta se ha de que dar a tiempo real en un plazo razonable.

2 Análisis exploratorio de los datos

Los datos consisten en 100 series temporales de longitud variable etiquetadas como alguna de las 3 clases: vino, plátano y *background*. Cada serie temporal es una secuencia de mediciones de 10 cantidades fisicoquímicas: 8 de óxido metálico (R1-R8), una de temperatura (*Temp.*) y una de humedad (*Humidity*).

En las series de *background* todas las mediciones son de *background*, pero las de vino o plátano tienen la siguiente forma: mediciones de *background* preestímulo; seguido de mediciones del estímulo de la clase que corresponde; y por último mediciones de *background* postestímulo. El inicio y la duración del estímulo en cada serie está en el fichero de metadatos y se tiene que hacer la reclasificación de cada medición individual.

La distribución en clases de las 100 series es de 36 vinos, 33 plátanos y 31 *background*. Como se ha dicho, las series son de distinta longitud y las de estímulo también tienen mediciones de *background*, por tanto es mejor fijarse en la distribución de mediciones individuales, que es de $11,57\%$ vino, $8,44\%$ plátano y $79,99\%$ *background*.

	R1	R2	R3	R4	R5	R6	R7	R8	Temp.	Hum.
vi	11.27	7.12	6.52	8.08	13.64	14.65	3.67	4.13	27.09	57.10
pl	12.13	9.09	8.82	10.16	19.18	15.06	5.28	6.42	27.07	57.75
bg	12.32	9.21	9.31	10.42	14.95	16.36	5.65	6.12	27.33	57.62

Table 1: Tabla de medias de las mediciones de cada clase. Se observa que el vino se diferencia pero el plátano es muy similar al *background*

2.1 Problemas

Las mayores dificultades que nos encontramos al analizar el problema son

- la distribución desbalanceada de clases, que tiene el background sobrerepresentado;
- y la dificultad intrínseca para detectar el plátano.

Que los datos contenga un $\approx 80\%$ de *background* hace que, de manera natural, todo clasificador entrenado en ellos esté sesgado a predecir *background*. Esto hace que la precisión no sea la métrica más adecuada para medir el rendimiento de un clasificador. Por ello, además de la precisión usaremos el *F1-Score*, que es la media ponderada de la precisión sobre cada clase.

Si al exceso de *background* le añadimos que el plátano es intrínsecamente difícil de detectar con los sensores de la nariz electrónica del problema, tenemos que clasificar correctamente los plátanos es un reto.

2.2 Modificaciones

Hemos realizado unas modificaciones a los datos. La primera es que la serie 95 no tenía datos y la hemos eliminado. La segunda es más importante, y es que hemos ajustado los inicios y la duración de los estímulos de algunas series. Esto lo hicimos porque inspeccionando las series nos dimos cuenta de que algunas tenían el estímulo claramente atrasado o adelantado, y esto produce ruido, aunque poco, durante el entrenamiento.

En la figura siguiente vemos la gráfica de algunos sensores de dos series, una de vino y otra de plátano. En la de plátano no se aprecia el estímulo, es indetectable mirando las gráficas de los sensores; en la de vino vemos como el inicio del estímulo está ligeramente adelantado.

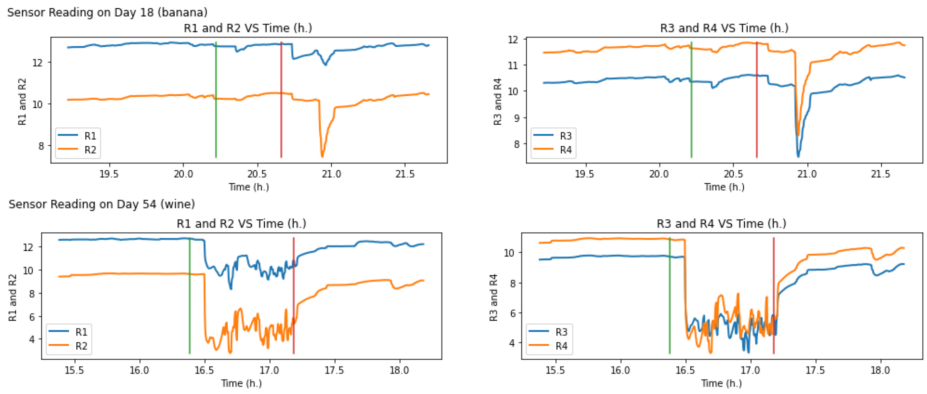


Fig. 1: Ejemplo de R1-R4 para series de plátano (arriba) y vino (abajo)

3 Descripción de los *datasets* y sus atributos

En primer lugar, y después de ver que algunos estímulos estaban mal marcados, hemos subsanado las desviaciones de comienzo y final de los mismos en los metadatos. Los atributos finales después de esta modificación siguen siendo los mismos que el *dataset* original.

Por otro lado, el sistema objetivo de este trabajo es crear un detector a tiempo real de ciertos gases, el cual puede utilizar las anteriores mediciones en su beneficio para determinar la clase de la lectura actual. Para preparar al sistema para esto, se ha generado un *dataset* donde cada lectura de cada serie contiene los 10 atributos habituales (sensores), 10 atributos que resumen la media de las últimas 120 lecturas de cada sensor, y 10 atributos formados por las desviaciones típicas de las últimas 120 mediciones de cada sensor. Se ha utilizado el valor 120 ya que en promedio hay una muestra por segundo, por lo que 120 son ventanas de 2 minutos. Adicionalmente, los primeros datos de cada serie han sido eliminados al no disponer de la información suficiente para calcular los últimos 20 atributos, pero siendo el tamaño de ventana igual a 120 provoca una pérdida de muestras menor del 1%.

En resumen y para ergonomizar la mención a los conjuntos de datos:

- `RAW-DB`: Conjunto de datos original.
- `CL-DB` (*Clean dataset*): Conjunto de datos original, pero con los valores de comienzo y final de estímulo ajustados correctamente.
- `MW-DB` (*Moving window dataset*): Conjunto de datos calculado a partir de `CL-DB`, donde cada muestra de una serie contiene las 10 lecturas de sensores, y las medias y desviaciones típicas de las 120 lecturas previas en el tiempo.
- `CL-DB-Sm` o `MW-DB-Sm`: Conjuntos de datos formados a partir de los dos anteriores donde se ha utilizado la técnica `SMOTE` (Synthetic Minority Oversampling Technique) para generar nuevos ejemplos artificiales de las clases minoritarias (vino y plátano) a partir de los datos del *trainset*, con el objetivo de reducir el peso de la clase *background* y aumentar el *F1-Score*.
- `SEQ-DB` (*Sequences dataset*): está compuesto de secuencias de 120 mediciones seguidas de cada serie. Debido al tamaño limitado de la memoria, se toman las secuencias con una separación (*step*) de 10 entre medidas iniciales. Cada dato tiene forma de matriz 120×10.

Es importante subrayar que los *datasets* anteriores pueden haber sido estandarizados según sea beneficioso para el modelo utilizado, es decir, se ha restado la media y dividido por la desviación típica de cada atributo. Para calcular las medias y las desviaciones típicas se ha utilizado únicamente la partición de *train*, y (lógicamente) ambas particiones (*train* y *test*) son normalizadas.

4 Modelos de clasificación

Para el análisis de los modelos de clasificación se ha utilizado validación simple con (mínimo) 10 repeticiones. El *dataset* usado se dividía en *trainset* (entorno al 75% del total de los ejemplos) y *testset* (el resto) en cada repetición y al

final se calculaba el rendimiento medio (ya sea precisión o *F1-Score*) junto con su desviación típica. Comentar que en cada repetición no solo se realizaba una partición aleatoria nueva del *dataset*, sino que se creaba una instancia nueva del clasificador en uso, con el objetivo de obtener el rendimiento medio generalizado. Finalmente, es importante notar que muestras de una misma serie (mismo `ID`) acaban en la misma partición.

4.1 Redes Neuronales

Empleamos las usadas en clase mediante la implementación de *sklearn MLPClassifier*. Posteriormente hemos realizado *Bagging* con la clase *BaggingClassifier*.

En primer lugar, los hiperparámetros analizados han sido el número de capas ocultas, las neuronas por capa y el parámetro regularizador. Además, se ha aumentado el número máximo de iteraciones a 2000, se han normalizado los datos para garantizar convergencia y se ha activado el *early stopping* y el *shuffle* (es decir, tras cada iteración los ejemplos se barajan). Se ha comprobado inicialmente que la constante de aprendizaje C no es muy relevante mientras se mantenga en niveles cercanos al de por defecto. Sin embargo, sí se ha permitido que ésta sea adaptativa, es decir, cuando no mejora el error de entrenamiento se reduce C.

El análisis completo se puede consultar en (A.1). Tras obtener la mejor red neuronal en cada *dataset*, se ha vuelto a entrenar pero ahora realizando *Bagging* con 20 estimadores. En todos los casos se ha mejorado de manera notable la precisión del algoritmo, así como aplicando la técnica `SMOTE`.

4.2 Kernels

Usaremos el clasificador *SGD* combinado con estos métodos. Las funciones kernel envían los datos a un espacio de dimensión mayor donde efectuar la separación para posteriormente proyectar dicha frontera en el espacio de atributos original.

En la web de *sklearn* hay 4 métodos distintos. Para elegir cual nos conviene más hemos ejecutado varias veces cada algoritmo sobre nuestra base de datos para comprobar qué método nos da mejor *accuracy*. Este método es el *skewed chi2 kernel*. Esta función admite dos parámetros (*skew*) y (*ncomp*) por tanto nos hemos propuesto a hacer el análisis de hiperparámetros correspondiente. La combinación elegida por tener mejor desempeño ha sido ($skew = 0.001, ncomp = 200$). Podemos ver la tabla completa en el Anexo (A.2).

Una vez obtenidos los hiperparámetros ejecutamos el algoritmo sobre `CL-DB` con validación simple de 10 repeticiones. Posteriormente hacemos *ensembles* de 500 de nuestro clasificador y volvemos a probar el rendimiento sobre `CL-DB` con 10 repeticiones. Cabe destacar que el uso de *ensembles* mejora un poco el rendimiento medio.

Tras realizar la prueba con `CL-DB`, nos proponemos usar nuestro clasificador en `MW-DB`, ya que esta DB tiene más información estadística sobre la serie temporal. De nuevo, realizamos una validación simple de 10 reps y posteriormente realizamos *Bagging* con 500 clasificadores y validación en 10 reps. Usar `MW-DB` mejora

notablemente la eficacia del algoritmo, discutiremos los resultados obtenidos en los siguientes apartados.

Finalmente hemos realizado `SMOTE`, explicaremos esta técnica y las implicaciones que tiene más adelante.

4.3 Conjuntos de árboles

Se usan los siguientes *ensembles* de árboles: *Random Forest* y *Boosting* de la librería Scikit-Learn [2]. En ambos casos, el primer paso es ajustar los hiperparámetros necesarios. Para el modelo *Random Forest* variamos el número total de estimadores, criterios de división (*gini* o entropía) y la profundidad máxima de cada árbol. La mejor configuración se obtiene con 500 estimadores, criterio por entropía y profundidad máxima 7. Para el modelo de *Boosting* se ha usado el clasificador *AdaBoost*, que para cada estimador aumenta el peso de los ejemplos mal clasificados por el anterior. Los hiperparámetros ajustados son el número de estimadores M y la constante de aprendizaje C, siendo $M = 300$ y $C = 0.5$ la mejor configuración. El informe completo de validación de hiperparámetros de ambos modelos se puede encontrar en (A.3).

Los mejores hiperparámetros se han utilizado para analizar el rendimiento medio de ambos modelos en los conjuntos de datos `CLEAN-DB` y `MW-DB`. Notar que los datos de `MW-DB` se han estandarizado, ya que existe diferencia entre las magnitudes de varios atributos.

Finalmente se ha calculado la matriz de confusión correspondiente y se han graficado varias series del *testset* para analizar tanto el rendimiento de clasificación por clase, así como la efectividad de la técnica `SMOTE`.

4.4 Redes neuronales recurrentes

Es natural usar redes recurrentes para este tipo de problemas, donde después de procesar una secuencia de observaciones se tiene que dar una predicción.

Se usa el conjunto de datos `SEQ-DB`, probamos distintas profundidades (una red o dos concatenadas) y número de unidades, y también se prueba a añadir una capa de ruido gaussiano aditivo por regularización y robustez. La red recurrente que se ha usado finalmente es LSTM, aunque también se hicieron pruebas con GRU. Más información en el apéndice (A.4).

En cuanto al proceso, es distinto al de los modelos de la parte principal del trabajo por la naturaleza de este: se separa conjunto de test y entrenamiento; sobre el conjunto de entrenamiento se hace validación cruzada con $k = 5$ para conjunto de hiperparámetros, haciendo *early stopping* con la precisión de validación; con los 5 modelos del mejor conjunto de hiperparámetros se hace un *ensemble* y se prueba en el conjunto de test para obtener el resultado final.

4.5 Redes neuronales convolucionales

Como en RNN, se entrena sobre el conjunto de datos `SEQ-DB`. Probamos con modelos convolucionales en 1 y 2 dimensiones. En el primero caso, un dato

120×10 se considera como 120 de longitud y 10 canales; en el segundo caso un *reshape* es necesario para convertir los datos 120×10 en $120 \times 10 \times 1$, es decir, 120 de altura, 10 de anchura y un solo canal.

La arquitectura de las CNN es de tres capas de Convolución/MaxPooling y una densa al final para hacer la clasificación, y en ambos casos se consideran distintos tamaños de filtros y de *kernels*. Para más información sobre la arquitectura y los hiperparámetros, consultar el apéndice (A.5). El proceso que se ha seguido es el mismo que para RNN.

5 Resumen de resultados

Tabla general con todo tal y como hemos visto en la pizarra online.

MODELO	`CL-DB`	`CL-DB-Sm`	`MW-DB`	`MW-DB-Sm`
NN	$84.6\% \pm 3\%$	- - -	$84.9\% \pm 2\%$	$82.2\% \pm 3\%$
Ens. NN	$84.4\% \pm 6\%$	- - -	$85.7\% \pm 3\%$	$86.2\% \pm 3\%$
Kernels	$79.1\% \pm 2\%$	- - -	$82.8\% \pm 3\%$	$82.6\% \pm 3\%$
Ens. Kernels	$80.8\% \pm 3\%$	- - -	$85.6\% \pm 3\%$	$84.8\% \pm 4\%$
Rand. Forest	$84.1\% \pm 5\%$	$83.1\% \pm 5\%$	$87.8\% \pm 4\%$	$85.6\% \pm 3\%$
AdaBoost	$83.3\% \pm 1\%$	$83.7\% \pm 1\%$	$86.3\% \pm 1\%$	$87.6\% \pm 2\%$

Table 2: Precisión media (*mean accuracy*) de los mejores clasificadores

MODELO	`CL-DB`	`CL-DB-Sm`	`MW-DB`	`MW-DB-Sm`
NN	$83.5\% \pm 3\%$	- - -	$84.2\% \pm 2\%$	$83.0\% \pm 3\%$
Ens. NN	$83.0\% \pm 7\%$	- - -	$84.2\% \pm 3\%$	$86.3\% \pm 3\%$
Kernels	$73.4\% \pm 3\%$	- - -	$79.5\% \pm 4\%$	$82.2\% \pm 3\%$
Ens. Kernels	$74.9\% \pm 2\%$	- - -	$82.7\% \pm 3\%$	$84.9\% \pm 3\%$
Rand. Forest	$80.3\% \pm 5\%$	$81.6\% \pm 5\%$	$84.3\% \pm 3\%$	$85.1\% \pm 4\%$
AdaBoost	$79.7\% \pm 2\%$	$81.4\% \pm 2\%$	$83.6\% \pm 1\%$	$84.3\% \pm 2\%$

Table 3: F1-Score medio de los mejores clasificadores

MODELO	Val. acc.	Val. F1-Sre.	Test acc.	Test F1-Sre.
RNN	$83.1\% \pm 4\%$	$78.6\% \pm 6\%$	81.7%	77.0%
CNN-1D	$81.8\% \pm 5\%$	$76.7\% \pm 7\%$	84.0%	79.4%
CNN-2D	$81.9\% \pm 3\%$	$78.4\% \pm 3\%$	83.0%	79.3%

Table 4: Precisión y F1-Score de validación y test para RNN, CNN-1D y CNN-2D sobre `SEQ-DB`

6 Discusión de los resultados

Tras analizar y limpiar los datos originales, se procedió a ejecutar los tres modelos principales del trabajo: redes neuronales, métodos Kernel y conjuntos de árboles.

Todos ellos han conseguido, tras la validación de hiperparámetros correspondiente, un rendimiento similar. Las redes neuronales y *Random Forest* alcanzan en un primer momento un 84% de precisión y aproximadamente un 82% de *F1-Score*. Por otro lado, parece que los métodos Kernel no funcionan tan bien con este *dataset*, y que necesitan más información estadística sobre la serie temporal. Obtienen una precisión algo inferior al 80% y un F1-score de 74%.

Los resultados son prometedores, pero teniendo en cuenta que el *background* conforma el 80% de los datos, nos hemos visto en la tesitura de intentar mejorar el rendimiento en las clases minoritarias. De ahí surge el *dataset* por ventanas móviles de 2 minutos, con el cual esperamos que los modelos puedan inferir mejor los ejemplos de los estímulos, ya que los nuevos atributos introducen información estadística de los atributos originales.

Con `MW-DB` mejoramos notablemente el rendimiento de todos los modelos, subiendo medio punto porcentual el rendimiento de las *NN* y 3 puntos los conjuntos de árboles. Los métodos Kernel son los que se ven más beneficiados con este cambio, que mejoran el rendimiento en un 5%. En cuanto al F1-score, todos los modelos han subido de forma proporcional al rendimiento, lo cual nos indica que el uso del *dataset* por ventanas mejora también la detección de clases minoritarias.

No obstante, podemos analizar las matrices de confusión de los modelos (2) y ver que el rendimiento en la clase *banana* es aún deficiente (imagen izquierda). Con sed de mejora, ejecutamos una técnica de *data augmentation* conocida como `SMOTE`: Synthetic Minority Oversampling Technique, con el objetivo de sintetizar nuevos ejemplos de las clases minoritarias.

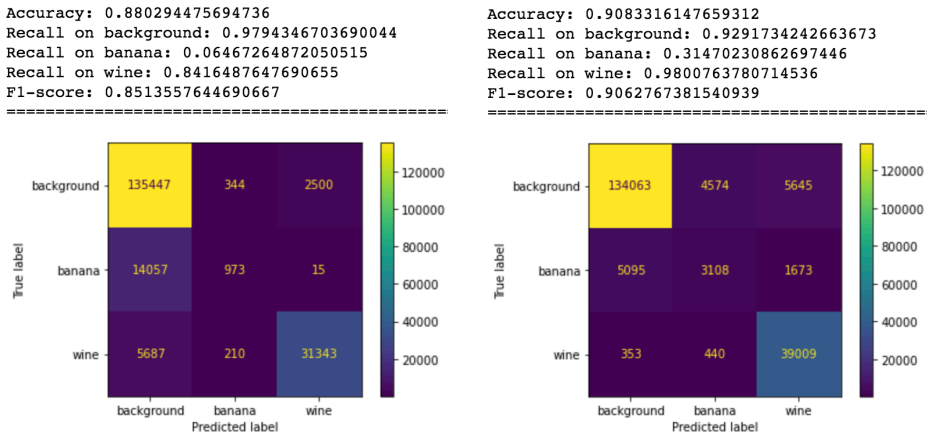


Fig. 2: Matriz de confusión del clasificador *Random Forest* `MW-DB` sin usar `SMOTE` (izquierda) y usando `SMOTE` (derecha)

Tras crear nuevos ejemplos a partir de los datos del *trainset*, entrenamos nuevos clasificadores de los modelos usados. Como resultado se obtiene una ligera disminución de la precisión pero una mejora considerable del *F1-Score*, que recordemos es una métrica más adecuada para tratar con datos desbalanceados.

Si observamos las columnas de la imagen de la derecha veremos que las clases no se confunden todas con todas uniformemente, sino que el error que se da con más frecuencia es confundir alguno de los estímulos con *background*, mientras que confundir los estímulos entre si es más raro. Notemos que tras el uso de `SMOTE` el rendimiento sobre el estímulo plátano ha aumentado claramente; antes se optaba por predecir clase *background* cuando era plátano (14057 casos) y ahora hemos conseguido una gran mejora, visible en la segunda fila de la matriz de confusión.

Aún con sed de mejora tras todo el trabajo realizado previamente, generamos *ensembles* de las mejores redes neuronales y los mejores métodos Kernel usando la técnica *Bagging*. Adicionalmente se ha intentado mejorar el rendimiento de los conjuntos de árboles usando el modelo *Boosting*, que varía los pesos de los ejemplos mal clasificados, algo interesante para mejorar el rendimiento de las *bananas*. El rendimiento de los anteriores en ambos *datasets* (`CL-DB` y `MW-DB`) junto con `SMOTE` es el mejor obtenido, entorno a un 88% de precisión y un 86-87% de *F1-Score*.

Para terminar la discusión, comentamos los resultados de los modelos de RNN y CNN, que se han hecho en paralelo a la parte principal del trabajo. Los resultados no son malos, pero son superados por modelos más clásicos, en particular si nos fijamos en el *F1-Score*. Puede haber varias razones para esto, pero lo más posible es que se deba a que estos modelos permiten más flexibilidad de arquitecturas y hayamos tomado decisiones subóptimas por nuestra inexperiencia.

7 Conclusiones

El *dataset* original tenía los inicios y finales de estímulo mal calculados. Esto implicaba una precisión mucho menor a la obtenida en cualquier resultado de los comentados anteriormente. Por ello hemos decidido trabajar directamente con el *dataset* limpio `CL-DB`.

El aumento del *F1-score* usando `MW-DB` es lógico, ya que con el análisis estadístico por ventanas tenemos mayor información sobre la tendencia de la serie temporal. Podremos por tanto identificar los patrones que siguen las clases que representan un estímulo real, es decir, las clases de vino y plátano.

En concreto, los mejores clasificadores se han obtenido con el conjunto de ventanas móviles y `SMOTE`. La mayor precisión se consigue con *Random Forest* (87.8%) y el mejor *F1-score* con *ensembles* de Redes Neuronales (86.3%).

El mejor rendimiento de cada clasificador ha sido muy parecido en la mayoría de los casos anteriores. Existe un error intrínseco a los datos que no podemos reducir, pero es posible que aún se pueda mejorar. En concreto, proponemos aumentar la cantidad de ejemplos ofrecidos o el número y la calidad de los sensores utilizados. Adicionalmente, se puede intentar utilizar otro conjunto de atributos calculados a partir de los ya existentes. Finalmente, también se pueden intentar mejorar los modelos más complejos utilizados: redes neuronales recurrentes y convolucionales.

El código y material adicional se puede encontrar en el repositorio [3].

References

[1] Ramon Huerta, Thiago Mosqueiro, Jordi Fonollosa, Nikolai F Rulkov, and Irene Rodriguez-Lujan. Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring. *Chemometrics and Intelligent Laboratory Systems*, 157:169–176, Oct 2016.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[3] Alejandro Santorum, José Manuel Chacón, David Cabornero, and Mario García. Gas sensors home activity monitoring repository. `https://github.com/AlejandroSantorum/Gas_sensors_home_activity_monitoring`.

[4] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[5] Jason Brownlee. Smote for imbalanced classification with python. `https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/`.

A Detalles de validación y ajuste de hiperparámetros

A.1 Hiperparámetros en redes neuronales

En esta tabla se muestra el análisis de hiperparámetros con la base de datos `CLEAN-DB`. Se han realizado más análisis de hiperparámetros con `MW-DB` y `MW-DB-Sm`, que se pueden consultar en el repositorio. Los hiperparámetros que se han variado en cualquier caso son el **parámetro regularizador** (*L2 penalty*) *alpha*, el **número de capas ocultas** y el **número de neuronas por capa** oculta. En concreto, los valores probados de *alpha* son {0.01,0.001}, el número de capas ocultas son {2,3,4} y el número de neuronas son {2,4,8}.

Todos los resultados se han probado con **validación simple** de 10 ejecuciones y en el notebook se incluye, además de la *accuracy* y el *F1-score* medio, el error en entrenamiento y el *recall* medio en la clase *banana*. Es en este último dato donde se puede comprobar que `SMOTE` mejora muchísimo la clasificación de las clases que no son *background*, mientras que observando el error de entrenamiento se observa que hay *overfitting* cuando se aumentan demasiado el número de neuronas.

Hiperparámetros	**Precisión media**	**F1-Score medio**
Alpha: 0.01, Neurons: (2,2)	$82.84\% \pm 3\%$	$78.66\% \pm 5\%$
Alpha: 0.01, Neurons: (4,4)	$83.44\% \pm 2\%$	$82.32\% \pm 3\%$
Alpha: 0.01, Neurons: (8,8)	$79.28\% \pm 5\%$	$79.34\% \pm 4\%$
Alpha: 0.01, Neurons: (2,2,2)	$83.35\% \pm 4\%$	$80.05\% \pm 7\%$
Alpha: 0.01, Neurons: (4,4,4)	$84.59\% \pm 3\%$	$83.46\% \pm 3\%$
Alpha: 0.01, Neurons: (8,8,8)	$81.04\% \pm 4\%$	$80.46\% \pm 4\%$
Alpha: 0.01, Neurons: (2,2,2,2)	$82.29\% \pm 5\%$	$77.50\% \pm 9\%$
Alpha: 0.01, Neurons: (4,4,4,4)	$81.32\% \pm 4\%$	$78.17\% \pm 7\%$
Alpha: 0.01, Neurons: (8,8,8,8)	$80.06\% \pm 3\%$	$79.98\% \pm 4\%$
Alpha: 0.001, Neurons: (2,2)	$83.08\% \pm 1\%$	$80.47\% \pm 3\%$
Alpha: 0.001, Neurons: (4,4)	$83.03\% \pm 3\%$	$81.33\% \pm 4\%$
Alpha: 0.001, Neurons: (8,8)	$82.64\% \pm 6\%$	$82.21\% \pm 6\%$
Alpha: 0.001, Neurons: (2,2,2)	$85.56\% \pm 4\%$	$81.93\% \pm 6\%$
Alpha: 0.001, Neurons: (4,4,4)	$83.71\% \pm 3\%$	$82.32\% \pm 5\%$
Alpha: 0.001, Neurons: (8,8,8)	$78.41\% \pm 4\%$	$78.93\% \pm 4\%$
Alpha: 0.001, Neurons: (2,2,2,2)	$80.42\% \pm 3\%$	$76.21\% \pm 4\%$
Alpha: 0.001, Neurons: (4,4,4,4)	$82.07\% \pm 2\%$	$80.47\% \pm 5\%$
Alpha: 0.001, Neurons: (8,8,8,8)	$76.78\% \pm 3\%$	$77.20\% \pm 2\%$

Table 5: Ajuste de hiperparámetros del modelo *Neural Networks*

Finalmente, se escogió un $alpha = 0.01$ con 3 capas ocultas de 4 neuronas cada una. En concreto, parece que los modelos simples (pocas capas ocultas y neuronas por capas) se comportan mejor. No solo se ha escogido en este modelo, sino que el análisis de hiperparámetros con ventanas y con `SMOTE` llevan a la misma conclusión.

A.2 Hiperparámetros en métodos *kernel*

En la siguiente tabla tenemos el análisis de hiperparámetros del método *kernel skewed chi 2*. Este método admite 2 parámetros, *skew* y número de componentes. Probamos con $skew = [.0001, .001, .01, .05, .1], ncomp = [20, 40, 80, 100, 200, 500]$ y hacemos validación simple de cada combinación con un **mínimo de 10 repeticiones**.

Hiperparámetros	**Precisión media**	**F1-Score medio**
Skew: 0.0001, Ncomp: 20	$82.83\% \pm 3\%$	$78.24\% \pm 4\%$
Skew: 0.0001, Ncomp: 40	$82.67\% \pm 3\%$	$77.74\% \pm 3\%$
Skew: 0.0001, Ncomp: 80	$82.02\% \pm 4\%$	$76.94\% \pm 5\%$
Skew: 0.0001, Ncomp: 100	$83.64\% \pm 2\%$	$79.14\% \pm 3\%$
Skew: 0.0001, Ncomp: 200	$82.55\% \pm 2\%$	$77.47\% \pm 2\%$
Skew: 0.0001, Ncomp: 500	$79.28\% \pm 1\%$	$73.94\% \pm 2\%$
Skew: 0.001, Ncomp: 20	$79.76\% \pm 1\%$	$73.94\% \pm 2\%$
Skew: 0.001, Ncomp: 40	$82.05\% \pm 3\%$	$76.60\% \pm 3\%$
Skew: 0.001, Ncomp: 80	$82.19\% \pm 1\%$	$76.29\% \pm 1\%$
Skew: 0.001, Ncomp: 100	$81.84\% \pm 4\%$	$76.32\% \pm 5\%$
Skew: 0.001, Ncomp: 200	$85.72\% \pm 2\%$	$79.82\% \pm 2\%$
Skew: 0.001, Ncomp: 500	$83.19\% \pm 3\%$	$78.71\% \pm 4\%$
Skew: 0.01, Ncomp: 20	$82.15\% \pm 3\%$	$78.71\% \pm 4\%$
Skew: 0.01, Ncomp: 40	$81.41\% \pm 2\%$	$76.21\% \pm 2\%$
Skew: 0.01, Ncomp: 80	$78.71\% \pm 2\%$	$76.21\% \pm 2\%$
Skew: 0.01, Ncomp: 100	$80.45\% \pm 4\%$	$75.25\% \pm 5\%$
Skew: 0.01, Ncomp: 200	$79.95\% \pm 2\%$	$75.28\% \pm 3\%$
Skew: 0.01, Ncomp: 500	$82.66\% \pm 2\%$	$78.05\% \pm 3\%$
Skew: 0.05, Ncomp: 20	$81.49\% \pm 1\%$	$76.35\% \pm 1\%$
Skew: 0.05, Ncomp: 40	$79.76\% \pm 3\%$	$73.67\% \pm 4\%$
Skew: 0.05, Ncomp: 80	$83.88\% \pm 3\%$	$79.04\% \pm 4\%$
Skew: 0.05, Ncomp: 100	$81.40\% \pm 4\%$	$76.25\% \pm 6\%$
Skew: 0.05, Ncomp: 200	$83.30\% \pm 3\%$	$78.39\% \pm 4\%$
Skew: 0.05, Ncomp: 500	$84.24\% \pm 3\%$	$80.10\% \pm 5\%$
Skew: 0.1, Ncomp: 20	$80.10\% \pm 2\%$	$74.03\% \pm 3\%$
Skew: 0.1, Ncomp: 40	$82.87\% \pm 3\%$	$77.82\% \pm 3\%$
Skew: 0.1, Ncomp: 80	$84.72\% \pm 2\%$	$79.81\% \pm 3\%$
Skew: 0.1, Ncomp: 100	$83.05\% \pm 3\%$	$79.81\% \pm 3\%$
Skew: 0.1, Ncomp: 200	$78.94\% \pm 3\%$	$73.43\% \pm 1\%$
Skew: 0.1, Ncomp: 500	$83.15\% \pm 1\%$	$77.79\% \pm 1\%$

Table 6: Ajuste de hiperparámetros del modelo *Kernel Method*

En general, parece que funciona bien para *Skew* bajo y un alto número de componentes.

Los **hiperparámetros que mayor** *accuracy* han dado han sido la combi-

nación de $Skew = 0.001, Ncomp = 200$, por tanto estos **serán los usados en el modelo**.

A.3 Hiperparámetros en conjuntos de árboles

Se expone a continuación el ajuste de hiperparámetros del modelo *Random Forest*, para el cual se ha supervisado su rendimiento comparando distintos valores del **número de estimadores (100, 300 o 500)**, los dos **criterios** para medir la calidad de una división **(gini o entropía)** y la **profundidad máxima de cada estimador (3, 7 u 11)**.
Recordar que la selección de hiperparámetros se ha realizado ejecutando un **mínimo de 10 veces validación simple** sobre el *dataset*, generando nuevas particiones *train-test* y un nuevo modelo *RandomForest* en cada repetición. Los valores de la siguiente tabla son los valores medios de todas esas repeticiones para cada configuración de hiperparámetros.

Hiperparámetros	**Precisión media**	**F1-Score medio**
N-estim: 100, Crit: gini, Max prof: 3	$79.46\% \pm 2\%$	$73.14\% \pm 3\%$
N-estim: 100, Crit: gini, Max prof: 7	$82.52\% \pm 3\%$	$79.18\% \pm 1\%$
N-estim: 100, Crit: gini, Max prof: 11	$77.70\% \pm 1\%$	$73.29\% \pm 2\%$
N-estim: 100, Crit: entropy, Max prof: 3	$84.67\% \pm 2\%$	$79.74\% \pm 3\%$
N-estim: 100, Crit: entropy, Max prof: 7	$86.27\% \pm 2\%$	$83.93\% \pm 3\%$
N-estim: 100, Crit: entropy, Max prof: 11	$83.57\% \pm 2\%$	$79.62\% \pm 3\%$
N-estim: 300, Crit: gini, Max prof: 3	$83.03\% \pm 4\%$	$77.90\% \pm 5\%$
N-estim: 300, Crit: gini, Max prof: 7	$85.25\% \pm 4\%$	$81.36\% \pm 5\%$
N-estim: 300, Crit: gini, Max prof: 11	$82.35\% \pm 3\%$	$80.41\% \pm 3\%$
N-estim: 300, Crit: entropy, Max prof: 3	$83.45\% \pm 3\%$	$78.49\% \pm 3\%$
N-estim: 300, Crit: entropy, Max prof: 7	$81.94\% \pm 2\%$	$77.72\% \pm 3\%$
N-estim: 300, Crit: entropy, Max prof: 11	$79.84\% \pm 4\%$	$75.79\% \pm 5\%$
N-estim: 500, Crit: gini, Max prof: 3	$84.88\% \pm 2\%$	$80.96\% \pm 2\%$
N-estim: 500, Crit: gini, Max prof: 7	$80.39\% \pm 3\%$	$75.29\% \pm 3\%$
N-estim: 500, Crit: gini, Max prof: 11	$83.53\% \pm 2\%$	$78.91\% \pm 2\%$
N-estim: 500, Crit: entropy, Max prof: 3	$85.61\% \pm 2\%$	$81.44\% \pm 3\%$
N-estim: 500, Crit: entropy, Max prof: 7	$85.75\% \pm 2\%$	$79.24\% \pm 4\%$
N-estim: 500, Crit: entropy, Max prof: 11	$77.91\% \pm 3\%$	$72.83\% \pm 5\%$

Table 7: Ajuste de hiperparámetros del modelo *Random Forest*

Primero, es fácil ver que el **mejor criterio de división** es el criterio usando la función de **entropía**. Por otro lado, podemos dudar qué valor de profundidad máxima escoger, pero nos vamos a decantar por **máxima profundidad=7** ya que una profundidad reducida regulariza el modelo al ser los árboles más simples, por lo que evitan el *overfitting*. No escogemos profundidad máxima igual a 3 ya que el **valor 7 otorga mejores resultados** medios. Finalmente, no se aprecia una mejora descomunal entre 300 estimadores y 500, pero sí que se percibe una

mayor estabilidad (desviación típica menor) si se utilizan 500 estimadores. Esto junto con el hecho de que aumentar el número de estimadores no perjudica al rendimiento medio del modelo *Random Forest* (**teorema de Condorcet**) nos permite asegurar que la mejor configuración de hiperparámetros obtenida es:

- Criterio de división: entropía ("entropy")
- Profundidad máxima de cada árbol: 7
- Número de estimadores (árboles del bosque): 500

Pasamos ahora a centrarnos en la **selección de mejores hiperparámetros** para el modelo de ***Boosting AdaBoost***. Se ha ajustado el **número de estimadores** del modelo **(100 o 300)** y la **constante de aprendizaje (0.1, 0.5, 1)**. En este caso no se ha considerado un número mayor de estimadores debido a que es un algoritmo de clasificación enormemente costoso de entrenar. Los resultados del rendimiento medio tras 10 repeticiones son los de la siguiente tabla.

Hiperparámetros	**Precisión media**	**F1-Score medio**
N-estim: 100, Const. aprendizaje: 0.1	$76.60\% \pm 5\%$	$71.82\% \pm 5\%$
N-estim: 100, Const. aprendizaje: 0.5	$79.08\% \pm 3\%$	$75.47\% \pm 3\%$
N-estim: 100, Const. aprendizaje: 1	$79.44\% \pm 2\%$	$79.46\% \pm 2\%$
N-estim: 300, Const. aprendizaje: 0.1	$82.48\% \pm 6\%$	$78.50\% \pm 6\%$
N-estim: 300, Const. aprendizaje: 0.5	$79.74\% \pm 5\%$	$75.93\% \pm 6\%$
N-estim: 300, Const. aprendizaje: 1	$82.02\% \pm 5\%$	$79.30\% \pm 6\%$

Table 8: Ajuste de hiperparámetros del modelo *AdaBoost*

Esta claro que la mejor configuración de hiperparámetros es usando una **constante de aprendizaje igual a 1** y **300 estimadores**.

A.4 Hiperparámetros en redes recurrentes

Si se mira la función que genera los modelos de RNN, se observa que tiene tres argumentos: `n units`, `seq to seq`, `lstm gru` y `noise`:

- `n units` cambia el número de unidades de la capa recurrente LSTM o GRU
- `seq to seq`, en caso estar activo crea el modelo con dos capas recurrentes, donde la primera está configurada en modo `return sequences`; en caso de no estarlo sólo se pone una capa
- `lstm gru` decide la capa recurrente, LSTM o GRU.
- `noise` añade una capa de ruido gaussiano aditivo al final de las capas recurrentes

A continuación se muestra una tabla con los resultados de la validación para los distintos hiperparámetros. Los distintos hiperparámetros que se han probado son todas las combinaciones de:

- número de unidades de la capa recurrente $\in \{8, 16\}$
- una capa recurrente o dos concatenadas
- añadir o no una capa de ruido gaussiano después de las recurrentes
- siempre se usan LSTM

Hiperparámetros	Val. accuracy	Val. F1-Score
('LSTM', 8, False, False)	$82.0\% \pm 4\%$	$76.5\% \pm 7\%$
('LSTM', 8, False, True)	$82.2\% \pm 4\%$	$77.4\% \pm 6\%$
('LSTM', 8, True, False)	$83.1\% \pm 4\%$	$78.6\% \pm 6\%$
('LSTM', 8, True, True)	$81.3\% \pm 5\%$	$76.1\% \pm 8\%$
('LSTM', 16, False, False)	$83.0\% \pm 5\%$	$78.3\% \pm 6\%$
('LSTM', 16, False, True)	$81.5\% \pm 5\%$	$75.9\% \pm 9\%$
('LSTM', 16, True, False)	$82.4\% \pm 5\%$	$78.0\% \pm 6\%$
('LSTM', 16, True, True)	$83.0\% \pm 4\%$	$78.3\% \pm 6\%$

Table 9: Ajuste de hiperparámetros del modelo *RNN*. En la columna de hiperparámetros se indica (LSTM/GRU, num. unidades, dos capas, ruido)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_68 (InputLayer)        [(None, 120, 10)]         0
_________________________________________________________________
lstm_94 (LSTM)               (None, 120, 16)           1728
_________________________________________________________________
lstm_95 (LSTM)               (None, 16)                2112
_________________________________________________________________
gaussian_noise_31 (GaussianN (None, 16)                0
_________________________________________________________________
dense_67 (Dense)             (None, 3)                 51
=================================================================
Total params: 3,891
Trainable params: 3,891
Non-trainable params: 0
```

Fig. 3: Ejemplo de resumen de modelo RNN

A.5 Hiperparámetros en redes convolucionales

Si se mira la función que genera los modelos de CNN, se observa que tiene tres argumentos: `kernel size`, `pool size`, `filters`:

- `kernel size` decide el tamaño de los kernels, en el caso de una dimensión es un número entero; en el de dos una dupla de enteros
- `pool size` decide el tamaño del pool, en el caso de una dimensión es un número entero; en el de dos una dupla de enteros
- `filters` indica el número de filtros de las capas convolucionales, es un número entero

La arquitectura que usamos es de tres capas Convolución/MaxPooling, y finalmente un Flatten y una capa densa para realizar la clasificación. Todas las

capas convolucionales tienen los mismos parámetros. Una forma de sofisticar la arquitectura puede ser permitir distintos parámetros para cada capa.

A continuación se muestra una tabla con los resultados de la validación para los distintos hiperparámetros. Los distintos hiperparámetros que se han probado son todas las combinaciones de:

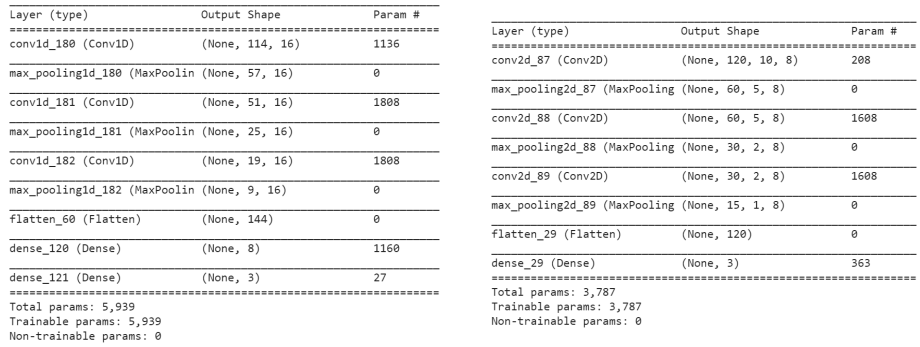
- tamaño del kernel $\in \{3, 5, 7\}$ para el caso unidimensional; $\{3, 5, 7\}$ en el caso bidimensional
- tamaño del pool se mantiene constante a 2 y $(2, 2)$
- el número de filtros $\in \{2, 4, 8, 16\}$ para el caso unidimensional; $\{2, 4, 8\}$ en el caso bidimensional

Hiperparámetros	**Val. accuracy**	**Val. F1-Score**
(3, 2, 2)	$78.1\% \pm 7\%$	$69.3\% \pm 10\%$
(3, 2, 4)	$80.3\% \pm 6\%$	$73.6\% \pm 9\%$
(3, 2, 8)	$80.7\% \pm 5\%$	$75.8\% \pm 7\%$
(3, 2, 16)	$80.4\% \pm 7\%$	$76.2\% \pm 10\%$
(5, 2, 2)	$77.4\% \pm 6\%$	$68.8\% \pm 9\%$
(5, 2, 4)	$81.8\% \pm 5\%$	$76.7\% \pm 7\%$
(5, 2, 8)	$81.7\% \pm 5\%$	$76.3\% \pm 6\%$
(5, 2, 16)	$81.6\% \pm 4\%$	$77.0\% \pm 7\%$
(7, 2, 2)	$78.2\% \pm 5\%$	$69.5\% \pm 6\%$
(7, 2, 4)	$80.1\% \pm 6\%$	$73.5\% \pm 9\%$
(7, 2, 8)	$80.2\% \pm 5\%$	$73.4\% \pm 8\%$
(7, 2, 16)	$81.3\% \pm 6\%$	$75.8\% \pm 9\%$

Table 10: Ajuste de hiperparámetros del modelo *CNN* unidimensional. En la columna de hiperparámetros se indica (tamaño kernel, tamaño pool, num. filtros)

Hiperparámetros	**Val. accuracy**	**Val. F1-Score**
((3, 3), (2, 2), 2)	$79.5\% \pm 5\%$	$72.9\% \pm 8\%$
((3, 3), (2, 2), 4)	$80.3\% \pm 4\%$	$77.4\% \pm 6\%$
((3, 3), (2, 2), 8)	$81.1\% \pm 3\%$	$77.2\% \pm 6\%$
((5, 5), (2, 2), 2)	$77.3\% \pm 2\%$	$70.7\% \pm 4\%$
((5, 5), (2, 2), 4)	$80.6\% \pm 4\%$	$75.6\% \pm 7\%$
((5, 5), (2, 2), 8)	$81.9\% \pm 3\%$	$78.4\% \pm 3\%$

Table 11: Ajuste de hiperparámetros del modelo *CNN* bidimensional. En la columna de hiperparámetros se indica (tamaño kernel, tamaño pool, num. filtros)



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_180 (Conv1D)          (None, 114, 16)           1136
_________________________________________________________________
max_pooling1d_180 (MaxPoolin (None, 57, 16)            0
_________________________________________________________________
conv1d_181 (Conv1D)          (None, 51, 16)            1808
_________________________________________________________________
max_pooling1d_181 (MaxPoolin (None, 25, 16)            0
_________________________________________________________________
conv1d_182 (Conv1D)          (None, 19, 16)            1808
_________________________________________________________________
max_pooling1d_182 (MaxPoolin (None, 9, 16)             0
_________________________________________________________________
flatten_60 (Flatten)         (None, 144)               0
_________________________________________________________________
dense_120 (Dense)            (None, 8)                 1160
_________________________________________________________________
dense_121 (Dense)            (None, 3)                 27
=================================================================
Total params: 5,939
Trainable params: 5,939
Non-trainable params: 0
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_87 (Conv2D)           (None, 120, 10, 8)        208
_________________________________________________________________
max_pooling2d_87 (MaxPooling (None, 60, 5, 8)          0
_________________________________________________________________
conv2d_88 (Conv2D)           (None, 60, 5, 8)          1608
_________________________________________________________________
max_pooling2d_88 (MaxPooling (None, 30, 2, 8)          0
_________________________________________________________________
conv2d_89 (Conv2D)           (None, 30, 2, 8)          1608
_________________________________________________________________
max_pooling2d_89 (MaxPooling (None, 15, 1, 8)          0
_________________________________________________________________
flatten_29 (Flatten)         (None, 120)               0
_________________________________________________________________
dense_29 (Dense)             (None, 3)                 363
=================================================================
Total params: 3,787
Trainable params: 3,787
Non-trainable params: 0
```

Fig. 4: Ejemplo de resúmenes de los modelos de CNN unidimensionales y bidimensionales