
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

Máster en Big Data: Tecnología y Analítica Avanzada

# MACHINE LEARNING II: ENSEMBLE

Jorge Ayuso Martínez

Francisco Javier Gisbert Gil

Carlota Monedero Herranz

Diego Sanz-Gadea Sánchez

José Manuel Vega Gradit

Madrid
Marzo 2023

# Índice

# 1. Ajuste de los modelos

## 1.1. Introducción

Para este apartado se usa el dataset `AJU_DATOS_DEM_C3.csv` que se proporciona. Este dataset contiene 2879 observaciones y 30 variables explicativas, de las cuales tres de ellas no pueden ser usadas. Concretamente, estas variables son la **fecha**, el **año** y el **día**.

Adicionalmente, podemos representar las series temporales tanto de los coeficientes de reparto como de las temperaturas máximas y mínimas. Estas gráficas nos permiten tener un mayor entendimiento acerca de la evolución de dichas variables explicativas con el tiempo:

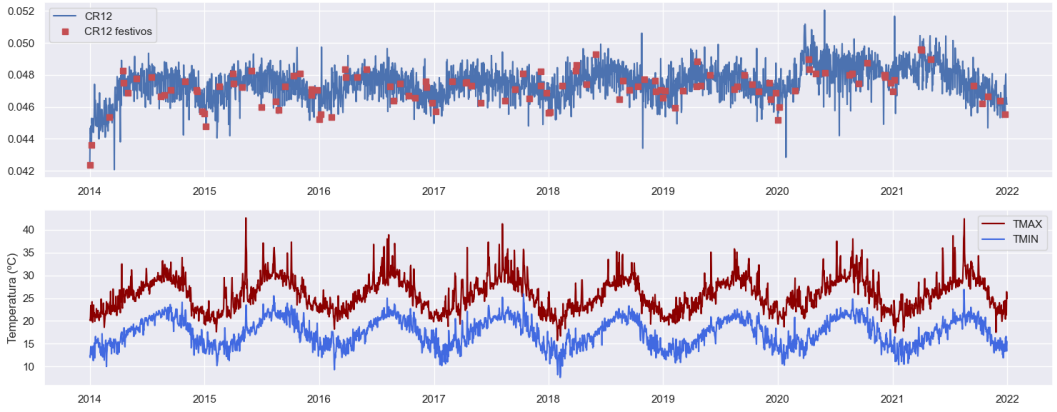



Figura 1: Coeficientes de reparto y temperaturas máxima y mínima a lo largo del tiempo

## 1.2. Análisis exploratorio de datos

En primer lugar, se muestra una matriz de correlación que nos permite entender mejor que variables están correlacionadas linealmente entre sí:

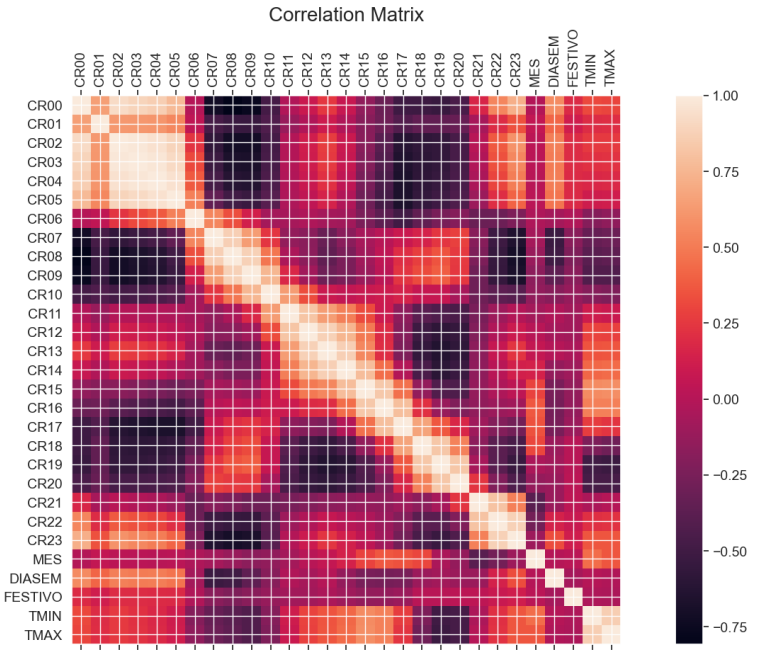



Figura 2: Matriz de correlación

Se pueden apreciar correlaciones muy importantes entre algunos pares de variables. Sin especificar en detalle todas estas correlaciones (dada la existencia de un gran número de ellas), se puede apreciar que las correlaciones existentes concuerdan con el perfil de demanda dado por los periodos horarios de Valle, Llano y Punta. Podemos apreciar que algunas de las correlaciones más relevantes son:

- Correlación muy positiva entre los coeficientes de reparto desde CR00 a CR05. Correspondientes a las horas Valle.
- Correlación muy negativa entre los coeficientes de reparto CR00...CR05 y los coeficientes de reparto CR07...CR10. Entre el Valle y la Primera Punta.
- Correlación muy negativa entre los coeficientes de reparto CR00...CR05 y los coeficientes de reparto CR16...CR20. Entre el Valle y la Segunda Punta.
- Correlación muy negativa entre los coeficientes de reparto CR12...CR15 y los coeficientes de reparto CR18...CR20. Entre el Llano y la Segunda Punta.
- En general parece haber una correlación positiva entre coeficientes de reparto consecutivos, como CR21 y CR22, lo cual tiene cierto sentido desde un punto de vista lógico. Obviamente esta correlación es más clara en algunos casos que en otros.
- Correlación bastante positiva entre $T_{min}$ y $T_{max}$. También se observa que las temperaturas están correlacionadas con las horas de más y menos sol.

Adicionalmente, es fundamental tener en cuenta que con este gráfico solamente somos capaces de detectar correlaciones lineales entre los datos. Esto significa que pueden existir otras correlaciones que no sigan una relación estrictamente lineal, y que por tanto habrá que estudiar más en detalle, si existen, en siguientes secciones.

Por otro lado, en la siguiente gráfica se muestra un gráfico interactivo (ver notebook para poder hacer uso de dicha interactividad) que permite entender más en detalle como evoluciona la demanda en función de la hora. Se muestra en detalle el perfil de demanda dado por lor periodos horarios de Valle, Llano y Punta:

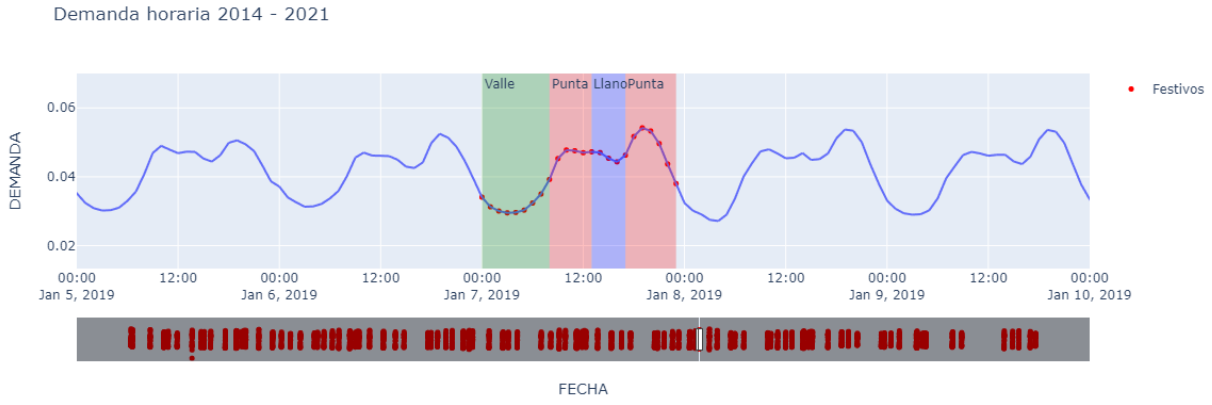



Figura 3: Demanda horaria desde 2014 hasta 2021

### 1.2.1. Variables cuantitativas discretas

En esta sección se procede a estudiar si existe alguna relación entre la variable de salida y las variables cuantitativas discretas de nuestro dataset, que son:

- Día de la semana.
- Mes.

Respecto al **día de la semana**, el gráfico obtenido es el siguiente:

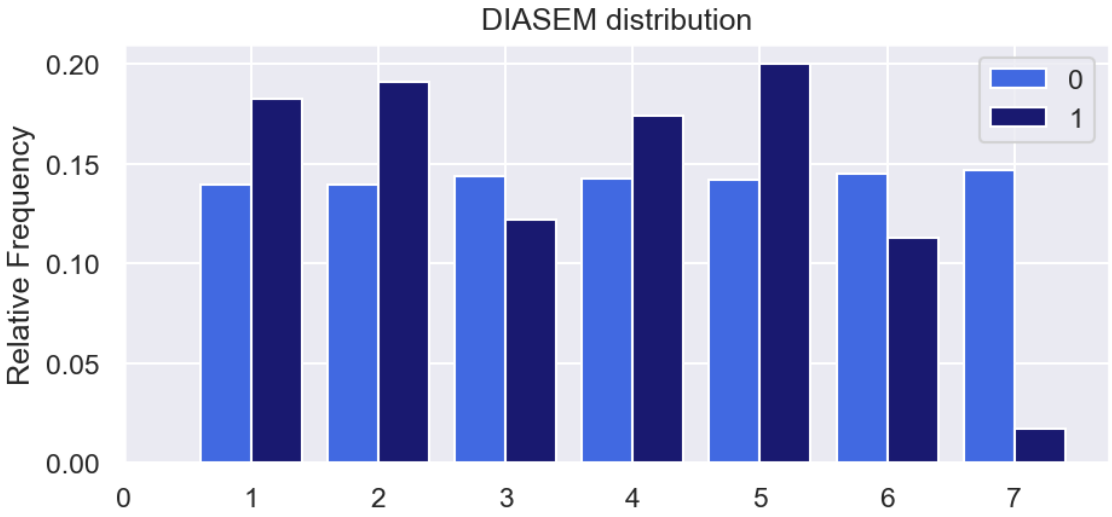



Figura 4: Relación entre días festivos y no festivos para cada día de la semana

Mientras que la gráfica correspondiente a la variable **mes** se muestra a continuación:

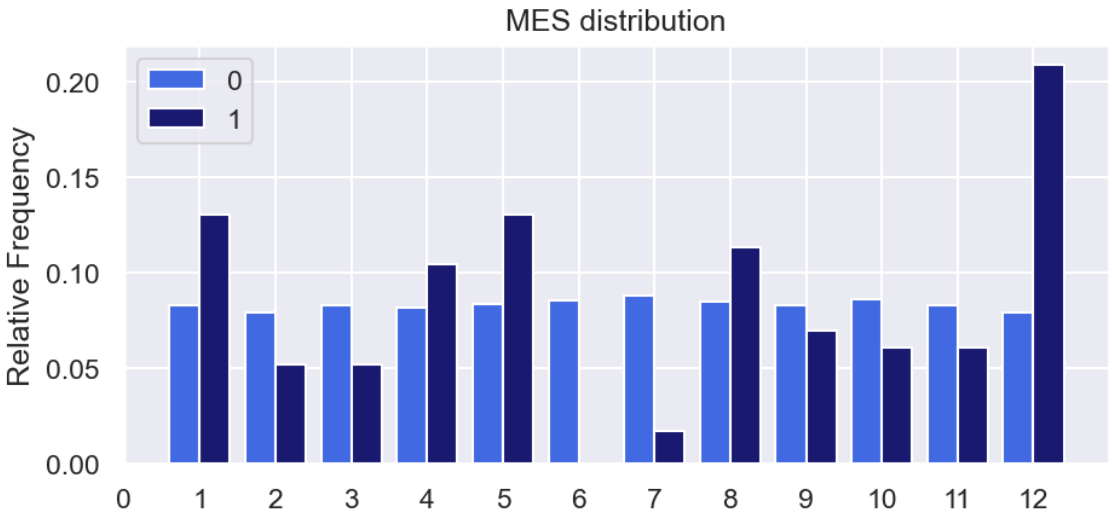



Figura 5: Relación entre días festivos y no festivos para cada mes

Se observa que tanto el mes como el día de la semana parecen variables relevantes para predecir si un día es festivo o no. Concretamente, se aprecia que:

**Respecto al día de la semana:**

Podríamos pensar que la relación existente entre el día de la semana y la variable de salida se debe a que tenemos una muestra finita de datos, dado que cada año un día en concreto (por

ejemplo el día de Navidad) se encuentra en el siguiente día de la semana respecto al año anterior (es decir, un año cae en lunes, al siguiente en martes, al siguiente en miércoles, etc). Sin embargo, si analizamos la cantidad de datos de los que disponemos, vemos que tenemos datos de prácticamente 8 años, por lo que esa diferencia entre días festivo por día de la semana no se debe a que tenemos datos de pocos años, sino a que realmente existe dicha relación. Por ejemplo, podemos ver que generalmente los festivos no caen en domingo, por que se suelen aplazar a otro día de la semana (típicamente al viernes anterior o al lunes siguiente).

**Respecto al mes:**

Vemos que claramente hay meses que tienen más días festivos que otros, siendo Diciembre el mes con más días festivos del año, y Junio el que menos festivos tiene.

### 1.2.2. Variables cuantitativas continuas

En este apartado se procede a estudiar si existe alguna relación entre la variable de salida y las variables cuantitativas continuas del dataset. Concretamente, se muestra la distribución de la variable explicativa en cuestión, segmentando los datos por los valores de la variable de salida. Esto nos permite entender qué variables pueden tener una mayor importancia a la hora de segmentar las observaciones correspondientes a la clase positiva y las observaciones correspondientes a la clase negativa. Adicionalmente, se muestra un boxplot de cada una de las variables de forma que se puede observar mejor la diferencia en media de ambas distribuciones o la presencia de outliers, entre otros aspectos.

Pese a que no se aprecia tan claramente la distribución correspondiente a los días festivos debido a que hay muchos más datos correspondientes a días no festivos, gracias a los boxplots, así como a las medias de las distribuciones (líneas discontinuas verticales), podemos apreciar que no para todas las variables la media de ambas distribuciones son igual de parecidas. En consecuencia, podemos apreciar que algunas variables parecen ser más útiles de cara a discriminar días festivos de días no festivos, como por ejemplo CR00, CR07 y CR08:

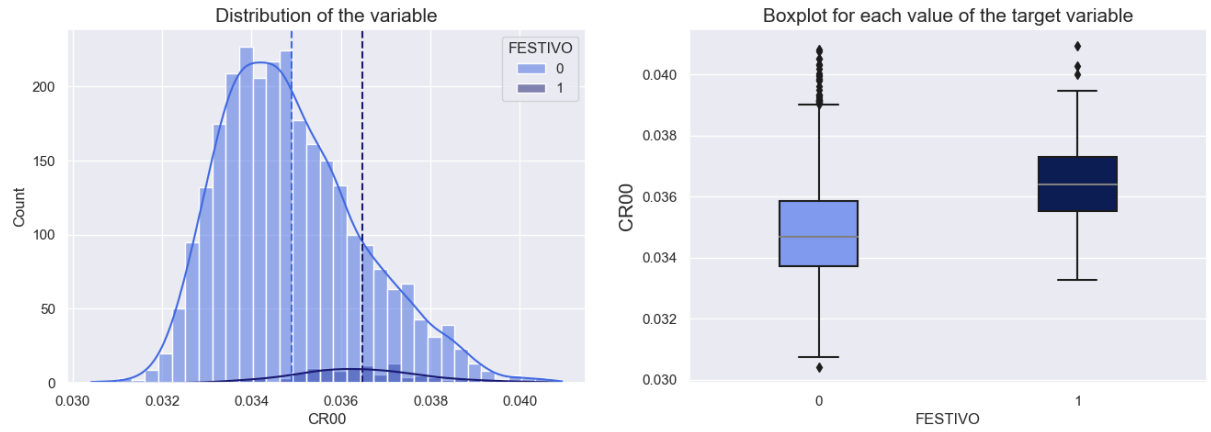



Figura 6: Distribución del coeficiente de reparto CR00

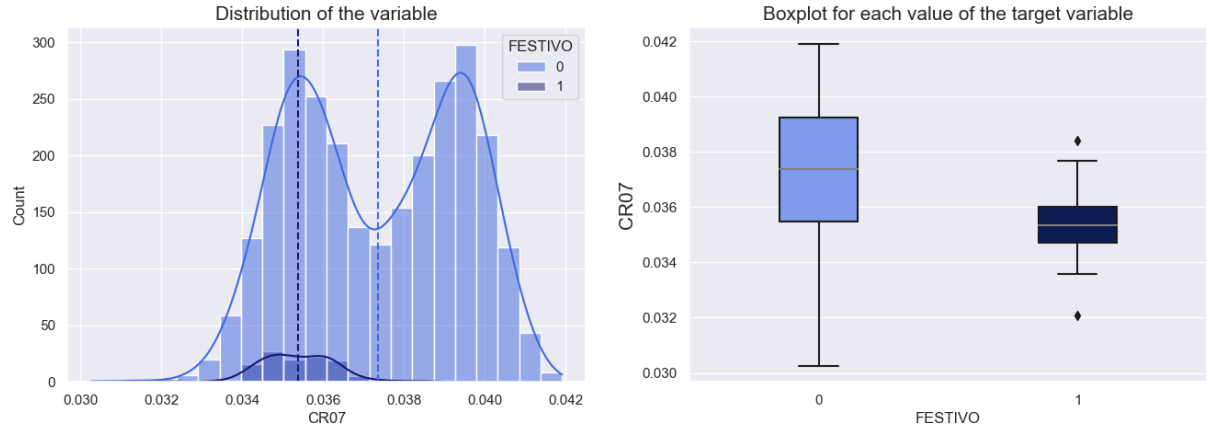



Figura 7: Distribución del coeficiente de reparto CR07

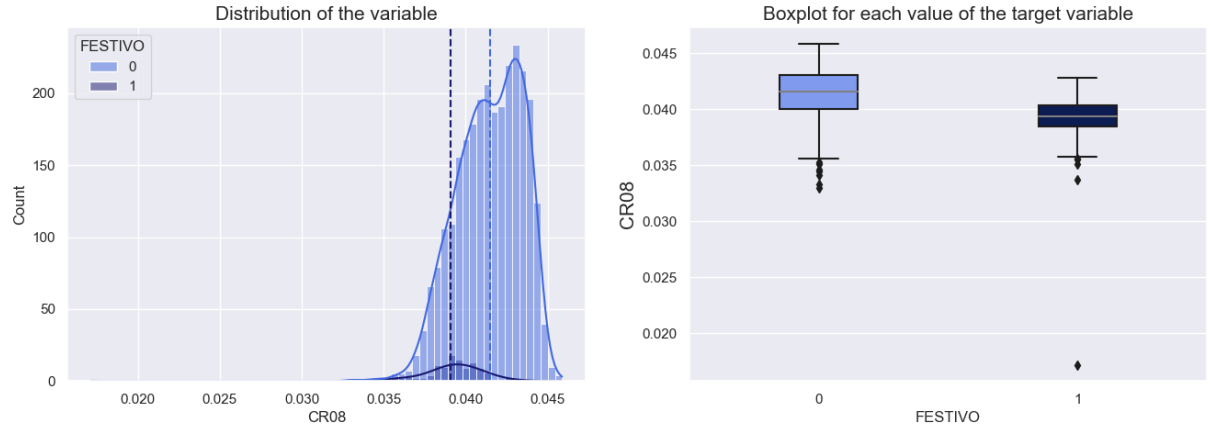



Figura 8: Distribución del coeficiente de reparto CR08

Asimismo, la distribución de algunas variables presenta un mayor nivel de asimetría, como por ejemplo CR01, CR02 y CR09:

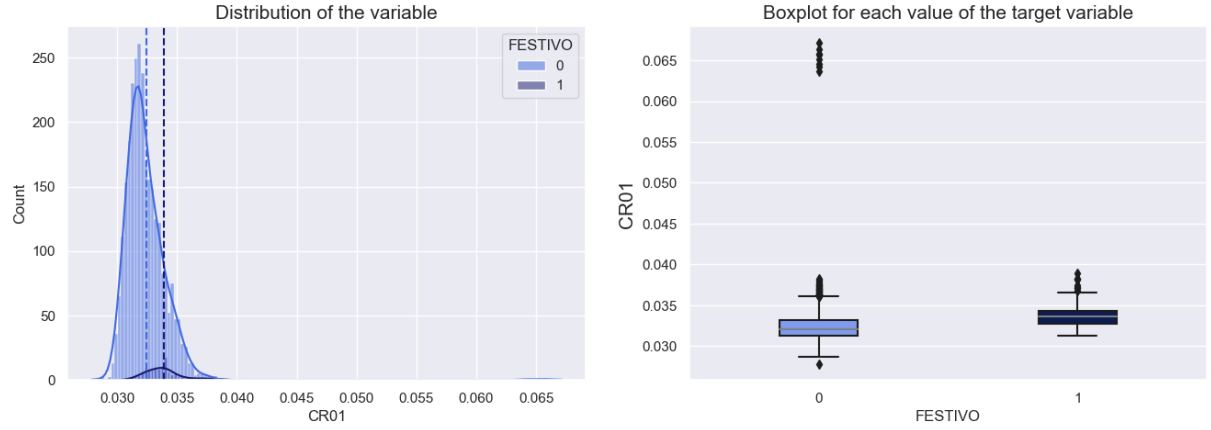



Figura 9: Distribución del coeficiente de reparto CR01

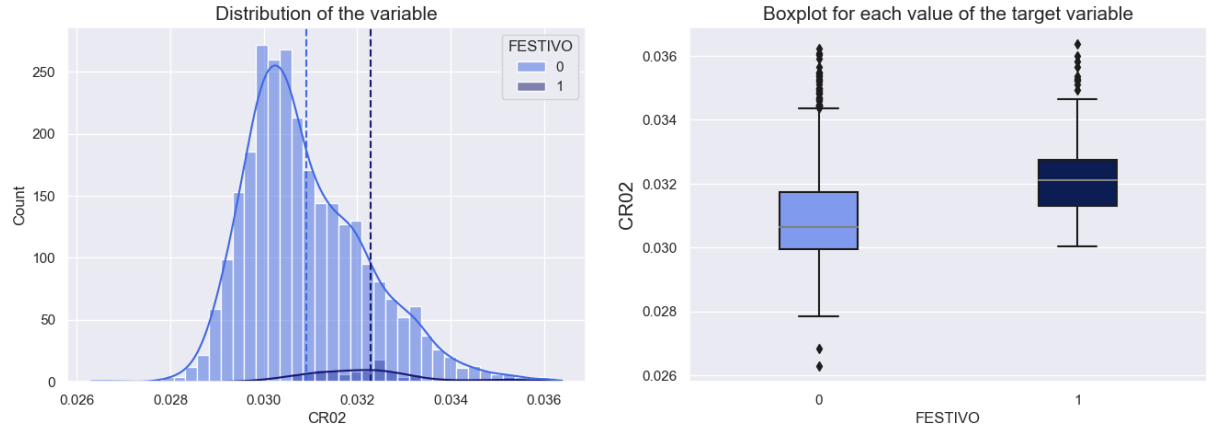



Figura 10: Distribución del coeficiente de reparto CR02

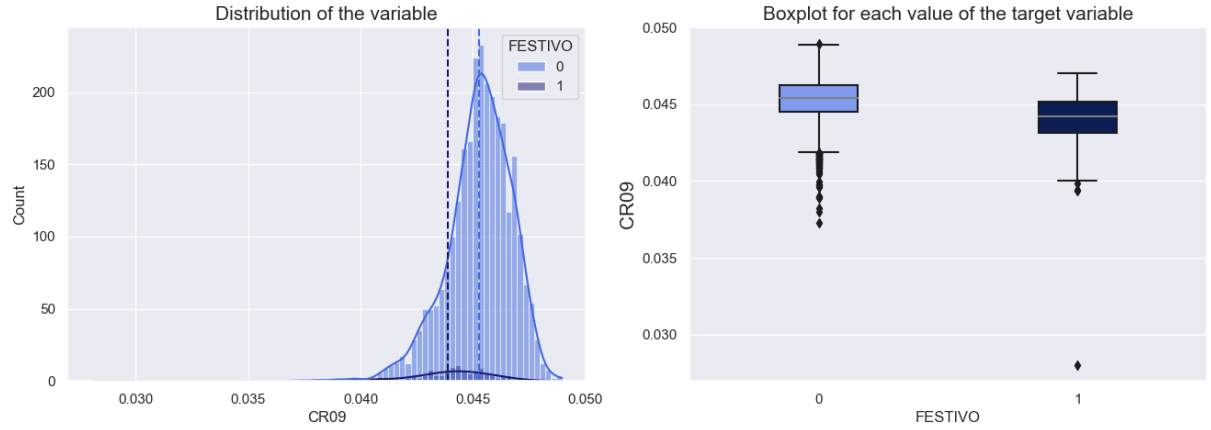



Figura 11: Distribución del coeficiente de reparto CR09

Es bastante probable que la asimetría que se observa a las 2 de la mañana (podemos comprobar en los boxplots que se debe a unos pocos datos), se deba a fechas muy concretas donde la gente estaba despierta hasta más tarde de lo habitual, como por ejemplo puede ser el día de Nochevieja. Si obtenemos dichos datos en detalle:

| | | | | | |
|---|---|---|---|---|---|
| 2014-01-01 | 2015-01-01 | 2016-01-01 | 2018-01-01 | 2019-01-01 | 2020-01-01 |
| 2020-04-10 | 2020-05-03 | 2020-05-10 | 2020-05-17 | 2020-05-24 | 2020-05-30 |
| 2020-05-31 | 2020-06-06 | 2020-06-07 | 2020-06-14 | 2020-06-20 | 2020-06-25 |
| 2020-06-28 | 2020-07-05 | 2020-07-19 | 2020-08-09 | 2020-09-27 | 2021-01-01 |

Tabla 1: Submuestra de *outliers* para CR02

Vemos que muchas de esas fechas fueron efectivamente los días de Nochevieja. El resto de fechas corresponden a la época del confinamiento durante el Covid-19, que suponen una situación anómala en cuanto a consumo energético se refiere.

Por otro lado, otras distribuciones parecen alejarse de una distribución normal. De hecho, parecen seguir una distribución bimodal, como son los casos de CR07 y CR19. Respecto a este último punto, es muy probable que esta distribución bimodal se deba a las diferentes épocas del año. Por

ejemplo, en invierno a las 7 de la mañana, es muy probable que se produzca un aumento del consumo de energía debido a que en muchas viviendas y oficinas se empiece a usar la calefacción. Sin embargo, en verano, a esa hora no suele hacer generalmente tanto calor como para que sea necesario hacer uso del aire acondicionado, dando lugar a un consumo menor de energía, y en consecuencia, un menor coeficiente de reparto.

Por ejemplo, si analizamos concretamente el coeficiente de reparto CR07, vemos que para valores mayores a 0,037 (es decir, pasado el primer pico de la distribución), la mayoría de valores corresponden a meses donde hace más frío. Si hacemos la misma comprobación pero a la inversa, se puede comprobar que generalmente un menor coeficiente de reparto corresponde a los meses más cálidos del año, confirmando nuestra hipótesis inicial.

| Mes | CR07 $< 0.037$ | CR07 $> 0.037$ |
|---|---|---|
| 1 | 96 | 149 |
| 2 | 73 | 151 |
| 3 | 81 | 155 |
| 4 | 101 | 137 |
| 5 | 105 | 141 |
| 6 | 127 | 109 |
| 7 | 219 | 27 |
| 8 | 226 | 21 |
| 9 | 119 | 119 |
| 10 | 78 | 167 |
| 11 | 75 | 161 |
| 12 | 113 | 129 |

Tabla 2: Número de observaciones correspondientes a cada mes según el valor de CR07

### 1.2.3. Outliers

Vamos también a ver el porcentaje de outliers que tiene cada variable haciendo uso de diversos métodos de calculo univariable de outliers a través de la clase `OutlierManager` que proporciona el porcentaje de outliers y el índice en el que se encuentran a través de métodos como:

- Rangos intercuartílicos.
- Desviación estándar: aplica a aquellas distribuciones normales de los datos.

En este caso, vamos a centramos en el IQR. Se observa que para ninguna variable explicativa se supera el porcentaje crítico del 5 % que típicamente se toma como umbral para determinar si el porcentaje de outliers es excesivo. Se observa además que en algunos casos este porcentaje es mayor que en otros, como por ejemplo de la variable explicativa CR10, donde el porcentaje es prácticamente del 4 %.

Por otro lado, se pueden estudiar los outliers con un enfoque multivariable por registro. Esto se puede hacer a través del algoritmo `Isolation Forest` que `sklearn` proporcina. Este algoritmo se basa en la construcción de un conjunto de árboles de decisión aleatorios donde objetivo del algoritmo es identificar observaciones anómalas (*outliers*) en un conjunto de datos que puedan diferir significativamente del resto de las observaciones. El algoritmo funciona dividiendo repetidamente el conjunto de datos en subconjuntos aleatorios mediante la selección de variables aleatorias y valores de corte aleatorios en cada iteración. Las observaciones anómalas tienden a requerir menos divisiones para ser aisladas, lo que significa que se encuentran en hojas más pequeñas de los árboles de

decisión y, por lo tanto, tienen una profundidad menor. Este tipo de algoritmos no cuentan con una métrica estándar como pueden ser los rangos intercuartílicos o $n$ número de desviaciones estándar. La definición de outliers sobre un espacio multivariable es más relativa, por lo que el algoritmo obtiene los registros más extremos según un porcentaje de registros a extraer. Se ha observado que existen 115 festivos, por lo que ajustando el parámetro de *outliers* a considerar 115 registros equivalen al 3,99 % de los registros totales, ello con la intención de observar si dichos *outliers* corresponden directamente con festivos o domingos. Se utiliza de nuevo la clase `OutliersManager`, que además realizará una gráfica sobre las tres componentes principales los *outliers* identificados según la contaminación porcentaje de *outliers* esperados. Dependiendo de la cantidad de varianza representada, la visualización de los *outliers* será más o menos fiable. Esto no quiere decir que los *outliers* sean menos fiables, sino que la representación sobre las tres dimensiones puede que no sea tan real.

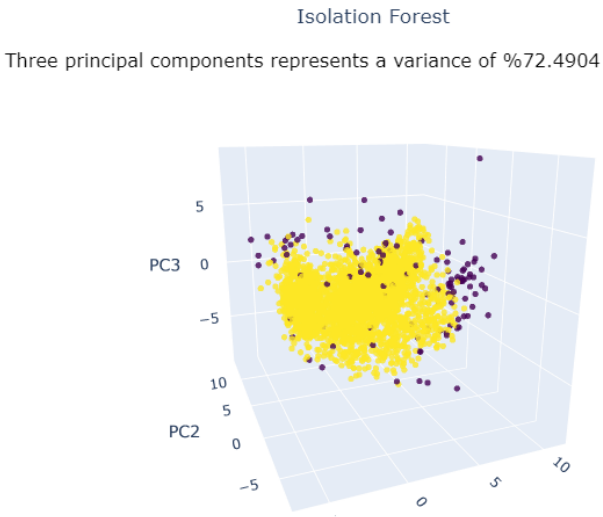



Figura 12: Outliers detectados con un algoritmo de Isolation Forest

El porcentaje de varianza representanda es del 72 % por lo que la gráfica es meramente orientativa y no representa la realidad visualizada. El número de outliers detectado dado los 115 registros más aislados del conjunto de datos coinciden con 15 de los 115 festivos, lo que indica que gran parte de los outliers no tienen por que ser festivos. Desde un punto de vista lógico, es bastante razonable pensar que estos outliers correspondan a observaciones asociadas a domingos. Por ello, a continuación se muestra un barplot de los 115 registros más aislados no coincidentes con los festivos:

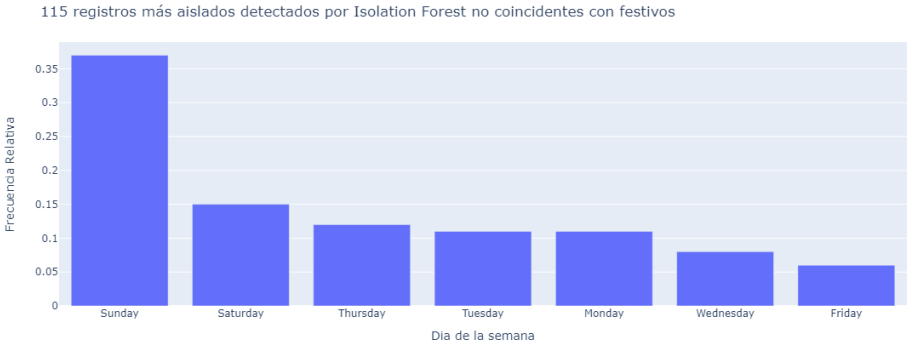



Figura 13: Relación de outliers detectados con cada día de la semana

Se observa que la mayoría de los 115 registros más aislados son domingos y sábados. Con esto se puede concluir que hay más outliers correspondientes a domingos y sábados que a los festivos etiquetados en el modelo -etiqueta_festivo =! domingo- correspondientes a outliers.

### 1.3. División de datos en entrenamiento y test

Es importante en los problemas de clasificación asegurarse de que el porcentaje de datos que pertenece a cada clase es el mismo en ambos datasets (train y test), de forma que que ambos datasets son representativos de la población de salida. Para ello, se ha utilizado la clase `StratifiedShuffleSplit` de `scikit-learn`. De esta forma, la frecuencia relativa de la variable de salida es la misma en ambos datasets. Tras dividir los datos en train y test comprobamos que efectivamente la frecuencia relativa de las dos clases de la variable de salida es prácticamente idéntico en ambos casos.

| Dataset | % Clase negativa | % Clase positiva |
|---|---|---|
| Train | 96.0052 | 3.9948 |
| Test | 96.0069 | 3.9931 |

Tabla 3: Distribución de ambas clases en los datasets de entrenamiento y test

### 1.4. Desbalanceo de clases

Como continuación de los resultados obtenidos en el apartado anterior, donde se comprobó que la distribución de ambas clases es idéntica a efectos prácticos en los datasets de entrenamiento y test, se aprecia además que las clases están totalmente desbalanceadas, lo cual se debe a que hay muchos más días festivos que no festivos. Si analizamos visualmente el balanceo de clases existente, se observa que en este caso hay una enorma diferencia de observaciones entre ambas clases, lo cual era de esperar dada la naturaleza del problema en cuestión:

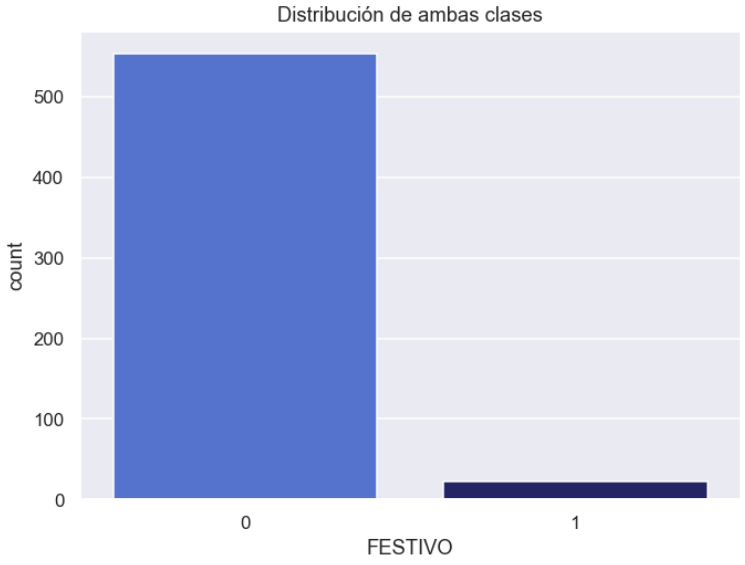



Figura 14: Desbalanceo de clases existente en el dataset

El desbalanceo de clases genera que los modelos sufran de *overfitting*. Esto es muy común en problemas como por ejemplo la detección de fraude, donde se disponen de mucho más datos de

la clase negativa que de la clase positiva. Por ello, para obtener un modelo que produzca buenos resultados, tenemos que aplicar técnicas de *upsampling*, como es el caso de **SMOTE**, las cuales nos permite aumentar el tamaño de observaciones correspondientes a la clase positiva. Más adelante se explicará en detalle la técnica usada para corregir el desbalanceo de clases, así como el código asociado a ello. Entrando más en detalle, existen dos posibilidades para solucionar el problema del desbalanceo de clases:

- Reducir los datos de la clase más grande, también conocido como ***downsampling***. Esta opción es ideal cuando se tiene una gran cantidad de datos, ya que después de eliminar los datos de la clase más grande se sigue teniendo un número significativo de instancias en cada clase. Sin embargo, esto no suele suceder en muchos casos.
- Crear datos sintéticos para la clase más pequeña, conocido como ***upsampling***. Con esta técnica, la clase más pequeña pasará a tener el mismo número de instancias que la clase más numerosa.

En este caso, usar *downsampling* no es la mejor opción debido a que no se tiene una cantidad excesiva de datos, por lo que utilizaremos la técnica de ***upsampling***, específicamente la técnica **SMOTE** (como se tienen variables categóricas, se hará uso de una variante de esta técnica que permite trabajar con variables categóricas). Esta técnica funciona de la siguiente manera:

1. Se toma una muestra aleatoria de la clase más pequeña.
2. Se buscan los $k$ vecinos más cercanos a esa muestra.
3. Se elige al azar una instancia de esos $k$ vecinos más cercanos.
4. Se obtiene un vector entre la muestra aleatoria y el vecino seleccionado anteriormente.
5. Se multiplica ese vector por un número aleatorio entre 0 y 1.
6. Se obtiene una nueva muestra sintética.

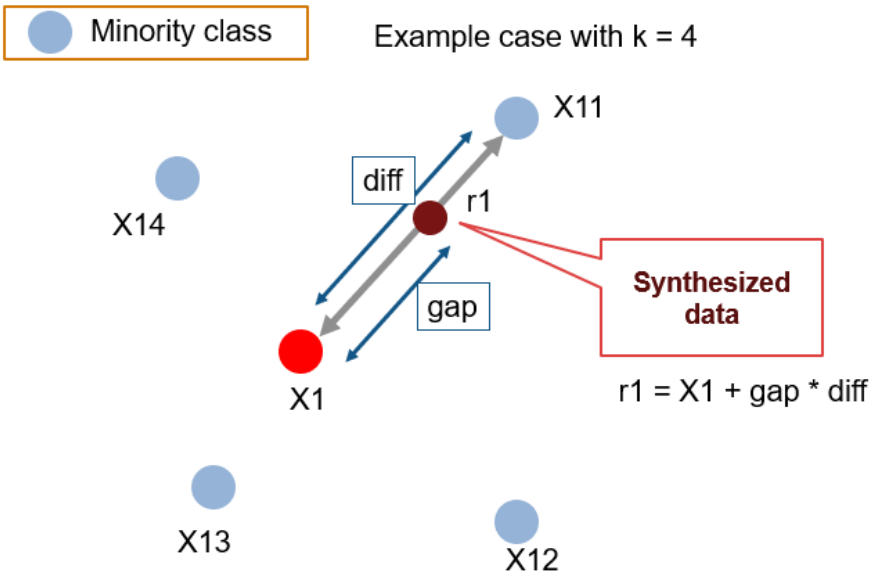



Figura 15: SMOTE

Este proceso se repite $n$ veces hasta que la clase más pequeña tenga el mismo número de instancias que la clase más grande. Un problema consecuente de aplicar **SMOTE** es que debido a que generamos un mayor número de muestras correspondientes a la clase positiva, generalmente se obtiene como resultado un aumento del *Recall* a costa de una disminución en el *Precision* ya que, si recordamos las fórmulas de ambas:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{1}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2}$$

Al introducir nuevas observaciones correspondientes a la clase positiva se produce un aumento del número de verdaderos positivos TP (y como consecuencia una disminución del *Recall*), pero también un aumento del número de falsos positivos FP (y como consecuencia una disminución del *Precision*).

En algunos casos, combinar ambas técnicas, *downsampling* y *upsampling*, puede dar mejores resultados, especialmente si utilizar solo *upsampling* resulta en una cantidad mayor de datos sintéticos que reales en la clase minoritaria, lo que podría generar un gran sesgo en el modelo.

En la siguiente gráfica se puede observar que ambas clases se encuentran balanceadas una vez esta técnica ha sido aplicada.

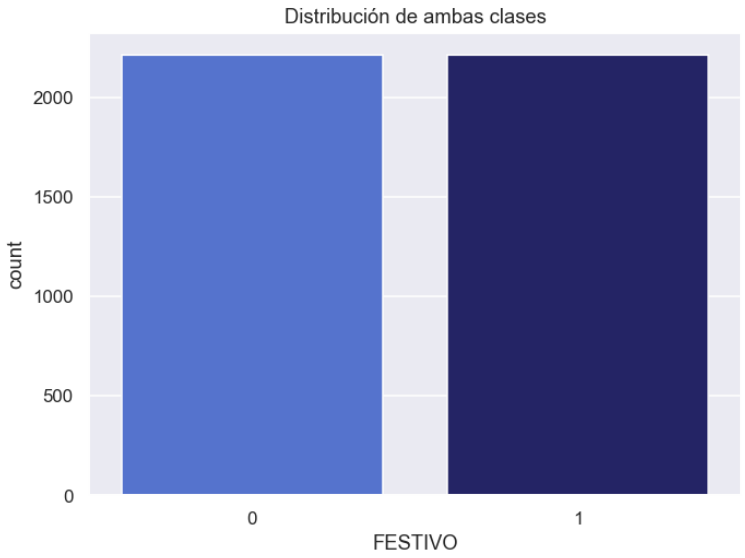



Figura 16: Clases balanceadas tras aplicar SMOTE

## 1.5. Pipeline de transformación de datos

Una vez hemos balanceado las clases en el dataset de entrenamiento, debemos de preparar los mismos para ser proporcionados a los modelos que entrenaremos en las siguientes secciones. Esto implica:

- Aplicar `Standard Scaler` sobre las variables numéricas. De esta forma, estas variables siguen una distribución normal de media 0 y desviación típica 1.

- Aplicar `One Hot Encoder` a las variables categóricas, de forma que se obtiene una columna por cada valor que la variable explicativa puede tomar.

Una vez el *Pipeline* de datos ha sido entrenado a los datos de entrenamiento y ha transformado los mismos, debemos aplicar dicho Pipeline también al dataset de test.

## 1.6. Árbol de decisión simple

***Nota***: *se ha probado diferentes combinaciones de hiperparámetros hasta obtener un árbol que sea robusto, de forma que pueda ser utilizado como punto de referencia a la hora de compararlo con otros modelos más complejos. No se muestra toda la experimentación realizada en esta sección ya que no es el objetivo de la práctica.*

Se ha aplicado la técnica de *Cross Validation* en el árbol para encontrar la mejor combinación de hiperparámetros. Adicionalmente, se ha seleccionado una profundidad pequeña (5 niveles de profundidad concretamente) para que el árbol no sufra de *overfitting*, pues se trata de un problema bastante común en este tipo de modelos. Una vez entrenado el árbol, la mejor combinación de hiperparámetros obtenida es:

| Hiperparámetro | Valor |
|---|---|
| `Min samples split` | 75 |
| `Min samples leaf` | 25 |
| `Min impurity decrease` | 0 |

Tabla 4: Mejor combinación de hiperparámetros para el árbol de decisión

A continuación se muestra la estructura del árbol de decisión, de forma que se pueda entender mejor el proceso de toma de decisiones del modelo:

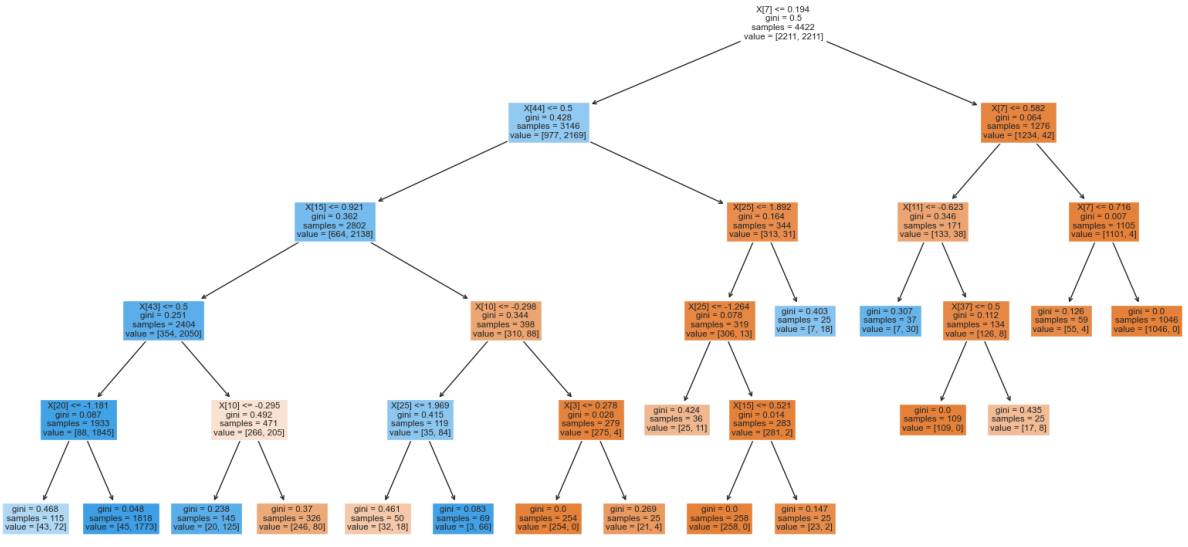



Figura 17: Estructura del mejor árbol de decisión obtenido

Se observa que el primer corte que realiza el modelo es para la séptima variable, que corresponde

al coeficiente de reparto CR07, que como podemos comprobar en la siguiente gráfica, es la variable más importante de todas para el árbol. Tras ello, para los dos siguiente cortes utiliza las variables:

- DIASEM7, es decir, si se trata de un domingo, que resulta ser otra de las variable más importantes para el modelo.
- De nuevo la séptima variable, es decir, CR07.

Adicionalmente se muestra cuales son las variables más importantes para el árbol:

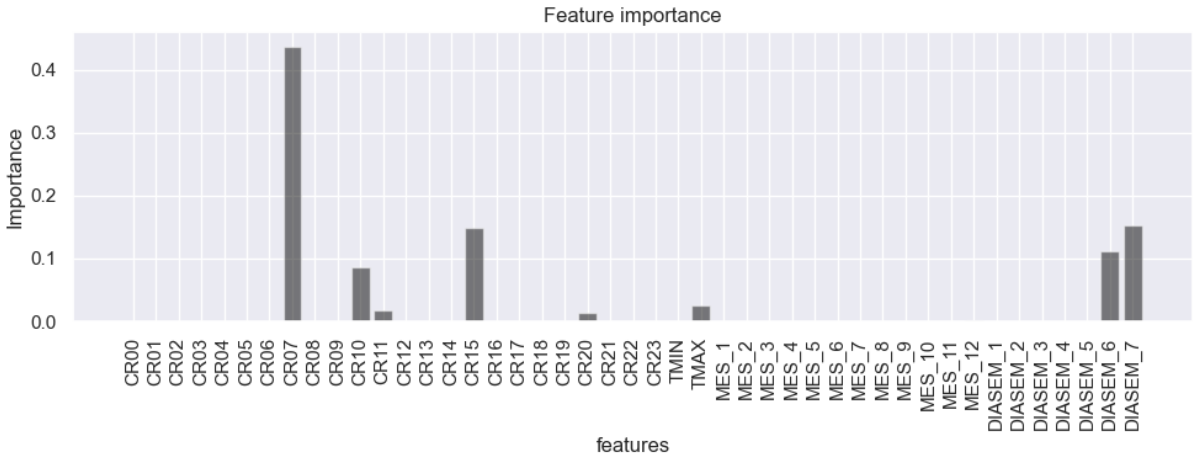



Figura 18: Importancia de las variables para el mejor árbol obtenido

Se observa que las variables más importantes para el modelo son:

| CR07 | DIASEM7 | CR15 | DIASEM6 |
|---|---|---|---|
| CR10 | $T_{max}$ | CR11 | CR20 |

Tabla 5: Variables más importantes para el árbol de decisión

Se podía esperar por ejemplo que CR07 fuera una de las variables más importantes, ya que cuando es festivo la gente suele despertarse en promedio más tarde, lo que también conlleva un consumo energético menor en las primeras horas de la mañana.

A continuación se muestra las métricas obtenidas por el árbol de decisión en los datasets de entrenamiento y test (se ha usado un *threshold* de 0.5):

| Métrica | Entrenamiento | Test |
|---|---|---|
| Accuracy | 0.9430 | 0.9497 |
| Balanced Accuracy | 0.9430 | 0.9321 |
| Precision | 0.9434 | 0.4375 |
| Recall | 0.9426 | 0.9130 |
| $F_1$ | 0.9430 | 0.5915 |
| ROC_AUC_SCORE | 0.9430 | 0.9321 |

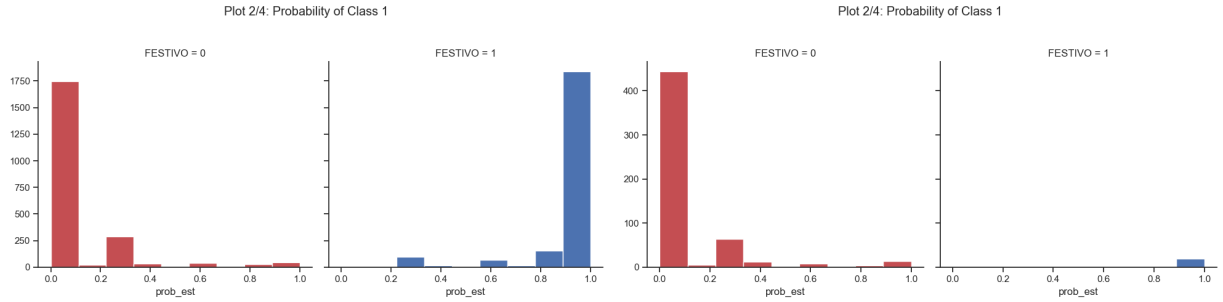
Tabla 6: Métricas del árbol de decisión en entrenamiento y test

Si observamos únicamente los resultados obtenidos en el dataset de entrenamiento, se observa como el modelo, pese a tratarse de un modelo sencillo, se ha aprendido en principio bastante bien

los datos de entrenamiento (vemos que el *Balanced Accuracy* es igual al *Accuracy* dado que hemos balanceado las clases). Adicionalmente, vemos que el *Recall* es prácticamente igual al *Precision* (y en consecuencia dando lugar a un $F_1$ muy similar a ambos). Adicionalmente, cabe destacar que el ROC AUC score obtenido es bastante elevado.

Si comparamos los resultados obtenidos en test con los obtenidos en el entrenamiento, se comprueba que efectivamente el modelo obtiene resultados significativamente peores sobre el dataset de test, especialmente en la métrica *Precision*. Vemos que el *Accuracy* ha incluso mejorado ligeramente en test, mientras que el *Balanced Accuracy* ha disminuido muy poco. Además, vemos una caída especialmente brusca en el *Precision*, como se ya se ha mencionado, pasando de 0,94 a 0,44. Esto, como se ha explicado anteriormente, es debido a la técnica de *upsampling* empleada, que tiene como consecuencia un aumento del *Recall* (es decir, una disminución de la tasa de falsos negativos) a costa de perjudicar al *Precision* (es decir, un aumento de la tasa de falsos positivos), siendo este efecto mayor cuando las clases están más desbalanceadas.

Si comparamos como de bien el modelo separa las observaciones de ambas clases en los datasets de entrenamiento y test:




(a) (b)

Figura 19: Capacidad del modelo para separar observaciones de cada clase en train (a) y test (b)

Comprobamos de forma visual que los nuevos resultados son peores en el dataset de test. En la figura 19(b) se observa que el modelo da una probabilidad de muy alta de pertenecer a la clase positiva a observaciones que pertenecen a la clase negativa. En consecuencia, ni modificando el *threshold* seríamos capaces de aumentar el *Precision* de forma significativa.

En consecuencia, podemos afirmar que el árbol de decisión sencillo, pese a servirnos como referencia para los nuevos modelos que se entrenen, no parece ser un modelo suficientemente bueno como para ser utilizado en producción dado que sufre claramente de *overfitting*, lo que da lugar a una pérdida de performance muy significativa cuando es expuesto a datos totalmente nuevos, especialmente en cuanto a falsos positivos se refiere.

## 1.7. Bagging

A la hora de realizar un Bagged Tree debemos de tener en cuenta no sólo los hiperparámetros del modelo final (como el número de estimadores empleados), sino también los hiperparámetros del estimador base, es decir, de estos modelos simples o *weak learners* que combinamos. De nuevo, aplicamos la técnica de *Cross Validation* o validación cruzada para encontrar la mejor combinación de hiperparámetros, obteniéndose:

| Hiperparámetro | Valor |
|---|---|
| `Max depth` | 11 |
| `Min samples split` | 5 |
| `Min samples leaf` | 5 |
| `Num estimators` | 150 |

Tabla 7: Mejor combinación de hiperparámetros para el Bagged Tree

Como podemos ver, la validación cruzada ha obtenido como resultado un número aceptable de estimadores (150), todos ellos bastante profundos (profundidad máxima de 11 nodos, lo que implica $2^{11}$ hojas), así como un número mínimo de muestras por split de 5 y un número mínimo de muestras por hoja de 5. Podemos apreciar que los valores de estos 2 últimos hiperparámetros es muy reducido, sobre todo considerando el tamaño de nuestro dataset inicial. Esto probablemente resulte en que los estimadores individuales que conformarán nuestro modelo tengan alto nivel de varianza, que es precisamente lo que se busca en el caso de los modelos de Bagging. Al combinar modelos con alta varianza, los cuales tienen puntos fuertes y débiles distintos al haberse sobreajustado a los datos a los cuales se entrenaron, obtenemos un modelo combinado con mayor capacidad de generalización.

Con el fin de encontrar las variables más significativas para el bagging classifier podemos utilizar `permutation_importance`. El método se basa en cuantificar cómo variaciones aleatorias en una variable impactan en la *performance* del modelo, manteniendo el resto de variables sin alterar. Las variables que más afecten al modelo al ser modificadas obtienen una mayor puntuación, lo cual se traduce en una mayor importancia para el modelo.

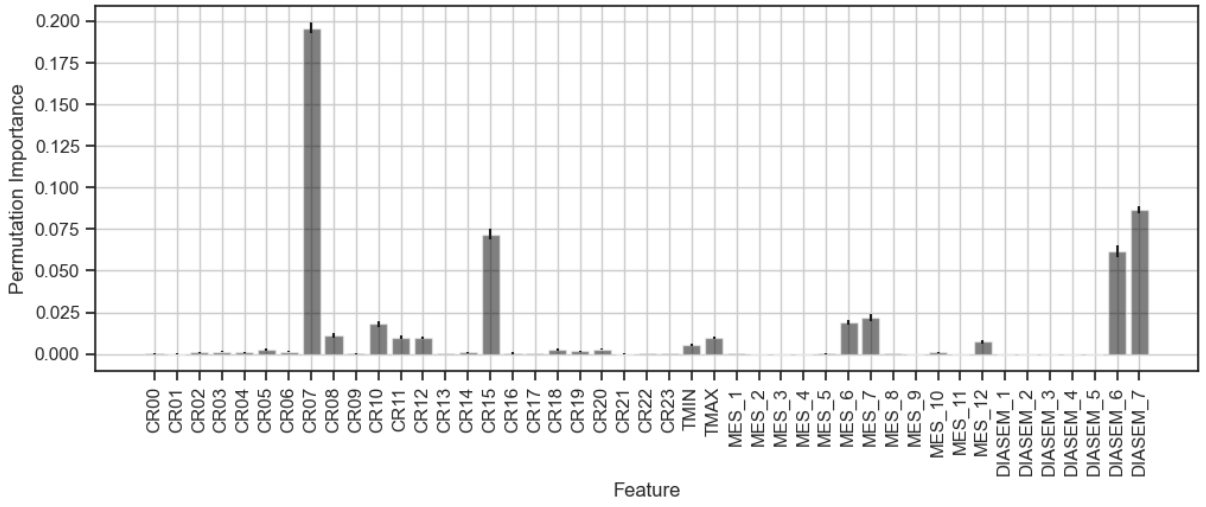



Figura 20: Importancia de las variables para el mejor modelo de Bagging obtenido

Se observa que las variables más importantes para el modelo son:

| | | | |
|---|---|---|---|
| CR07 | DIASEM7 | CR15 | DIASEM6 |
| CR10 | MES6 | MES7 | $T_{max}$ |

Tabla 8: Variables más importantes para el modelo de Bagging

La importancia de las variables es bastante similar a la del árbol de decisión simple.

Las matrices de confusión y métricas para el modelo de Bagging son:

$$\text{Train: } \begin{pmatrix} 2196 & 15 \\ 14 & 2197 \end{pmatrix}$$

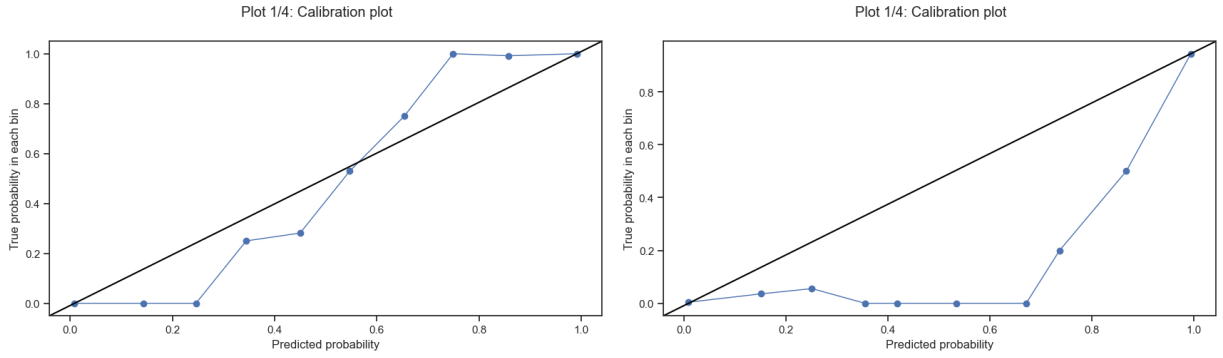
$$\text{Test: } \begin{pmatrix} 540 & 13 \\ 4 & 19 \end{pmatrix}$$

| Métrica | Entrenamiento | Test |
|---|---|---|
| Accuracy | 0.9934 | 0.9705 |
| Balanced Accuracy | 0.9934 | 0.9013 |
| Precision | 0.9932 | 0.5938 |
| Recall | 0.9937 | 0.8261 |
| $F_1$ | 0.9934 | 0.6909 |
| ROC_AUC_SCORE | 0.9934 | 0.9013 |

Tabla 9: Métricas del modelo de Bagging en entrenamiento y test

Podemos observar que hemos conseguido una mejora significativa en los resultados con respecto a lo que habíamos observado en el modelo más simple. Vemos que, si bien en el conjunto de entrenamiento estamos alcanzando unos resultados casi perfectos (con un accuracy del 99 %), la mejora más sustancial se encuentra en el desempeño en el conjunto de test, sobre todo atendiendo a métricas clave como la precisión del modelo.

Ahora, si comparamos las gráficas de calibración para ambos conjuntos de datos:




(a) (b)

Figura 21: Capacidad del modelo para separar observaciones de cada clase en train (a) y test (b)

Centrándonos en las gráfica correspondientes al de datos de entrenamiento, vemos que el modelo de Bagged Tree presenta un calibration plot algo más descalibrado que el árbol de decisión simple, así como claros indicios de *overfitting* cuando observamos el gráfico AUC. En el conjunto de test, seguimos observando un claro sesgo hacia las clases positivas, que se traduce en un incremento del ratio de falsos positivos, como habíamos podido apreciar en las matrices de confusión señaladas más arriba.

## 1.8. Random Forest

Como comentábamos en la sección anterior, los modelos de bagging combinan múltiples modelos de aprendizaje con alta varianza y bajo sesgo para mejorar la generalización a nuevos datos del modelo final y reducir su varianza. Sin embargo, es importante tener cuidado de no generar un modelo final que sufra de sobreajuste, ya que esto puede llevar a un peor rendimiento en datos nuevos.

En nuestra elección de hiperparámetros seguimos esta filosofía, tratando de crear modelos de aprendizaje débiles con alta varianza para evitar que el modelo final haga un ajuste excesivo a los datos de entrenamiento y así lograr una mejor generalización. Para ello, aumentamos la profundidad de los árboles y el número de árboles y características utilizados en cada árbol, buscando un equilibrio adecuado entre la complejidad y la capacidad de generalización del modelo final.

Adicionalmente, utilizaremos una aproximación algo diferente a la optimización de hiperparámetros. Dado que la búsqueda de la complejidad óptima mediante *GridSearch* es computacionalmente intensiva, queremos explorar la opción de emplear algoritmos basado en una búsqueda guiada, que permitan una exploración eficiente del espacio de soluciones. Esto es especialmente crítico cuando queremos encontrar combinaciones de varios hiperparámetros, donde podemos vernos significativamente limitados si queremos realizar una búsqueda por *GridSearch* (debido al problema combinatorio que supone). Con modelos simples como Decision Tree esto no suele ser un problema, pero en modelo más complejos como los que utilizaremos a continuación es necesario implementar métodos más avanzados si queremos hacer un tuning detallado de los hiperparámetros.

La opción que hemos escogido es la optimización bayesiana. Esta técnica se basa en la construcción de un modelo probabilístico sobre la función a optimimizar. Dicho modelo define una distribución *a posteriori* de la función objetivo a través de un muestreo aleatorio de puntos en el espacio de búsqueda definido. Dichos puntos son evaluados sobre la función objetivo, lo cual permite actualizar el modelo probabilístico y usarlo para predecir cómo se comportará la función objetivo en puntos inexplorados del espacio. De esta forma, podemos dirigir la búsqueda a las zonas donde es más probable que se encuentren los máximos de la función, lo que permite reducir el número de veces que se evalúe la función objetivo.

El código para esta optimización bayesiana se encuentra en la clase `ModelOptimizer`. Cabe destacar que dentro de la clase definimos una `black_box_function`, que será la función a optimizar, y también definimos la manera en la cual se realiza la optimización. En este caso, emplearemos como métrica el *cross_val_score*, y la métrica de scoring empleada dentro de la función de validación cruzada será el *accuracy*, ya que al haber balanceado las clases haciendo uso de SMOTE esta es equivalente al *balanced-accuracy* pese al desbalanceo inicial entre las clases positiva y negativa. Los valores de variables finales obtenidos son:

| Hiperparámetro | Valor |
|---|---|
| `Max depth` | 10 |
| `Min samples split` | 15 |
| `Min samples leaf` | 5 |
| `Num estimators` | 154 |
| `Max Features` | 6 |

Tabla 10: Mejor combinación de hiperparámetros para el Random Forest

Si lo comparamos con el modelo de Bagging ajustado anteriormente podemos observar que no hay un gran cambio con respecto a los hiperparámetros escogidos. Sin embargo, es importante destacar la introducción de un nuevo hiperparámetro: `max_features`, que hace referencia al número

máximo de variables que se le presentan a cada uno de los árboles simples a la hora de hacer cada corte. La introducción de este hiperparámetro es una de las mejoras del modelo de Random Forest frente al algoritmo base de Bagging, y busca reducir significativamente la correlación entre los árboles y, con ello, reducir el error asociado a la varianza y mejor el rendimiento del modelo.

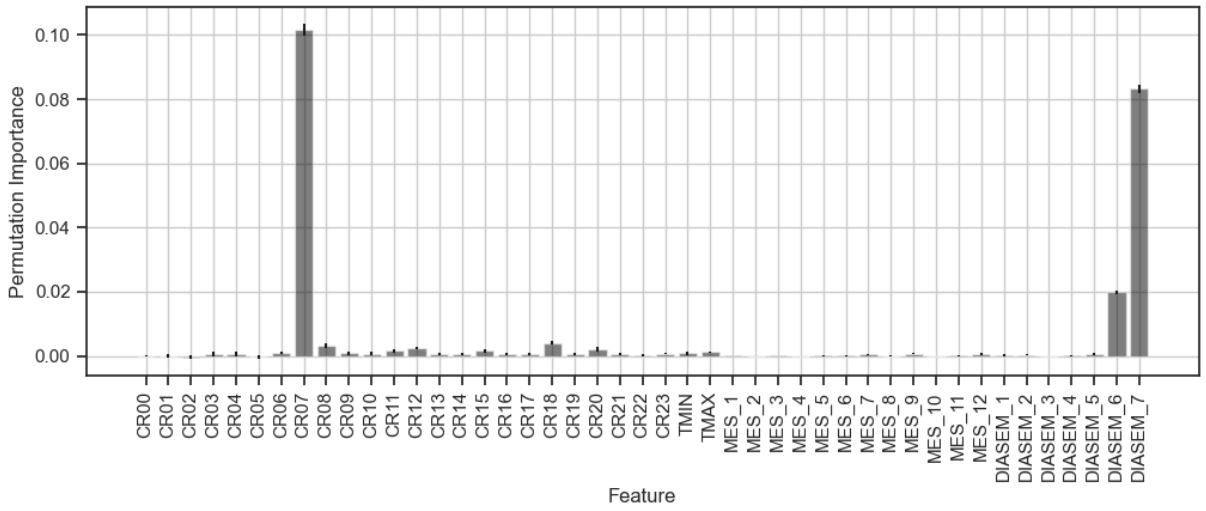



Figura 22: Importancia de las variables para el mejor modelo de Random Forest obtenido

Podemos ver como en este caso el perfil de importancia es significativamente diferente lo que habíamos observado para los árboles anteriores. Las variables principales son únicamente CR07, DIASEM7, DIASEM6, siendo el resto de variables de una importancia comparativamente menor.

Las matrices de confusión y métricas para el modelo de Random Forest son:

$$\text{Train: } \begin{pmatrix} 2183 & 28 \\ 7 & 2204 \end{pmatrix}$$

$$\text{Test: } \begin{pmatrix} 540 & 10 \\ 3 & 20 \end{pmatrix}$$

| Métrica | Entrenamiento | Test |
|---|---|---|
| Accuracy | 0.992 | 0.9774 |
| Balanced Accuracy | 0.992 | 0.9257 |
| Precision | 0.9874 | 0.6667 |
| Recall | 0.9968 | 0.8695 |
| $F_1$ | 0.9921 | 0.7547 |
| ROC_AUC_SCORE | 0.992 | 0.9257 |

Tabla 11: Métricas del Random Forest en entrenamiento y test

A priori, el modelo de Random Forest puede parecer tan sólo marginalmente mejor que el modelo de Bagging anterior, sobre todo si atendemos únicamente a las métricas de Accuracy. Sin embargo, podemos ver que sí hay una mejora significativa en los niveles de precisión y $F_1$ de este

estimador con respecto a modelos anteriores, lo que nos indica que estamos consiguiendo corregir parte del sesgo del modelo hacia las clases positivas. Del mismo modo, la estrategia de reducir el nivel de correlación entre los árboles contribuye a reducir el error asociado a la varianza, lo que se demuestra en una diferencia menos pronunciada del rendimiento entre los conjuntos de train y test, así como un área bajo la curva ROC excelente. lo que sugiere un poder discriminativo sobresaliente. No obstante, a pesar de las mejoras observadas, seguimos teniendo algo de sesgo hacia las clases positivas (nuestra principal fuente de error siguen siendo los falsos positivos).

## 1.9. Gradient Boosting

Con el fin de ampliar el rango de modelos empleados para resolver el problema proponemos la implementación de dos técnicas diferentes basadas en boosting:

- En primer lugar, emplearemos el paquete lightgbm, con una implementación ligera del algoritmo de gradient boosting que presenta ventajas en cuanto a la optimización del uso de memoria y velocidad de computación
- En segundo lugar, emplearemos adicionalmente el paquete XGboost, que a pesar de ser algo más lento se encuentra entre los paquetes más populares para implementar la técnica de Gradient Boosting.

De manera similar a lo realizado para Random Forest, emplearemos el método de optimización bayesiana para el ajuste de hiperparámetros. Además, incluimos en la parte de LGBM una comparación entre la optimización bayesiana y el empleo de `GridSearch`, con el fin de comprobar si ambos métodos llegan a soluciones similares.

### 1.9.1. LightGBM

#### LightGBM con Optimización Bayesiana

En este caso, los hiperparámetros óptimos escogidos para el modelo son:

| Hiperparámetro | Valor |
|---|---|
| `Max depth` | 3 |
| `Num estimators` | 130 |
| `Learning Rate` | 0.1 |

Tabla 12: Mejor combinación de hiperparámetros para el LightGBM

Es importante notar que en este caso hemos cambiado de forma radical nuestra aproximación al problema, puesto que en este caso estamos tratando de diseñar un comité de árboles poco expandidos que se vayan incorporando secuencialmente para reducir progresivamente los errores del modelo global. Es por ello que, a pesar de que el número de estimadores óptimos pueda ser similar, la profundidad máxima de los mismos es mucho más reducida. En este caso, no se ha procedido a utilizar los parámetros de `min_samples_split` ni `min_samples_leaf` debido a que tratamos de tener árboles muy poco expandidos, por lo que la limitación establecida por `max_depth` ya es suficiente para conseguir nuestro objetivo.

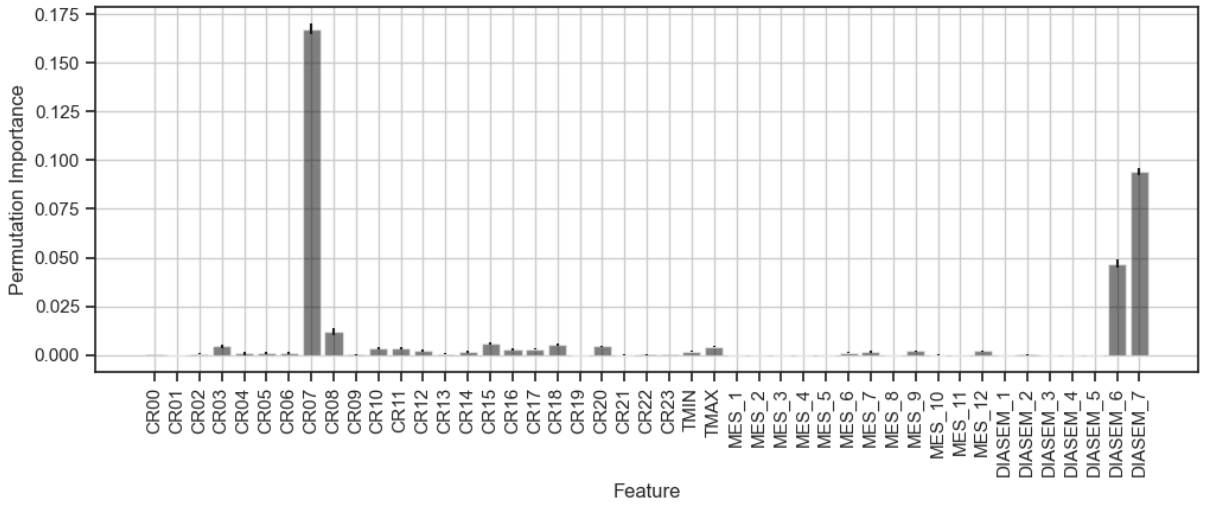


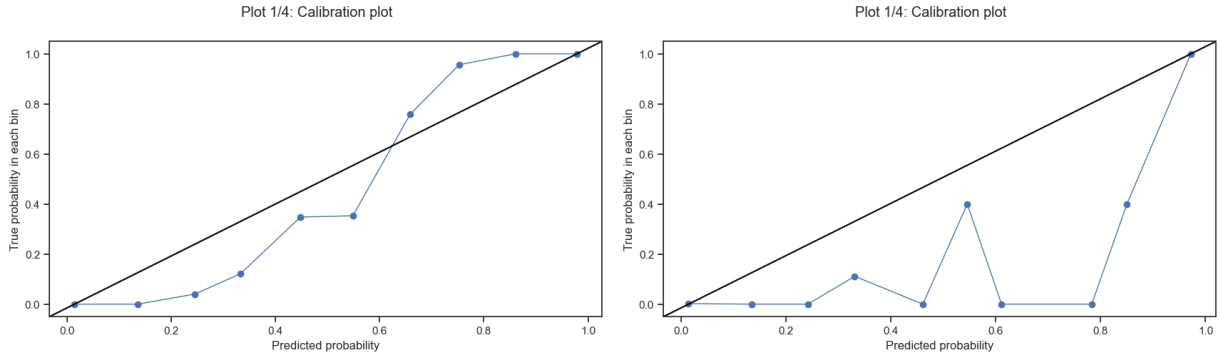

Figura 23: Importancia de las variables para el mejor modelo de LightGBM

Podemos comprobar que la importancia de las variables empleadas es similar a lo que observamos con el algoritmo de Random Forest anterior, siendo las variables CR07, DIASEM7 y DIASEM6 las más importantes. Las matrices de confusión y métricas para el modelo son:

| Métrica | Entrenamiento | Test |
|---|---|---|
| Accuracy | 0.9923 | 0.9826 |
| Balanced Accuracy | 0.9923 | 0.9492 |
| Precision | 0.9909 | 0.7241 |
| Recall | 0.9936 | 0.9130 |
| $F_1$ | 0.9923 | 0.8077 |
| ROC_AUC_SCORE | 0.9923 | 0.9493 |

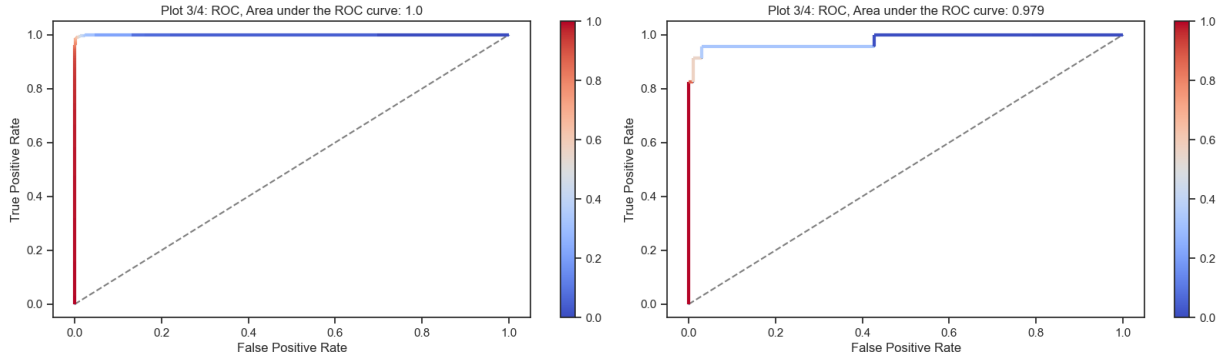
Tabla 13: Métricas de LightGBM en entrenamiento y test

En este caso podemos observar que el desempeño del modelo es superior a todos los modelos evaluados anteriormente, consiguiendo reducir la diferencia observada entre el dataset de train y el dataset de test con respecto a lo observado para el modelo de Random Forest. De igual manera, hay un aumento significativo en la precisión y el score $F_1$ para el dataset de test, que nos indica una vez más que estamos consiguiendo contrarrestar el sesgo hacia la clase positiva (aunque todavía de manera subóptima).




(a) (b)

Figura 24: Curva de calibración en train (a) y test (b)




(a) (b)

Figura 25: Área bajo la curva en train (a) y test (b)

Con respecto a las representaciones gráficas del desempeño, podemos ver que nuestras conclusiones anteriores son respaldadas por los gráficos. Observamos que el perfil de AUC del dataset de test es mucho más equilibrado en el caso de LGBM con respecto a modelos anteriores, a pesar de que el plot de calibración sigue mostrándonos cierto sesgo hacia las clases positivas.

### LightGBM con Grid Search

Por mero interés científico, hemos decidido hacer una comprobación de si el modelo de optimización bayesiana presenta unos resultados similares a los obtenidos por medio de una búsqueda intensiva por medio de `GridSearch`. En este caso, hemos escogido hacer este experimento con el modelo de LGBM y no con el modelo de Random Forest por dos motivos principales:

- LGBM presenta una performance mejor que Random Forest, por lo que es interesante evaluar cómo el modelo de mejor performance podría variar en función del método de optimización empleado
- En el caso de LGBM hemos realizado una búsqueda mucho más guiada del óptimo de los hiperparámetros, con unos rangos mucho más acotados y un número menor de variables. Es por ello que, para reducir en lo posible el nivel de consumo de recursos computacionales,

preferimos compara una búsqueda intensiva sobre un problema combinatorio menor que en el caso del modelo de Random Forest.

Podemos ver que los parámetros obtenidos son bastante similares con respecto a aquellos observados en el caso de la optimización bayesiana:

| Hiperparámetro | Valor Bayesian | Valor Grid-Search |
|---|---|---|
| Max Depth | 3 | 2 |
| Learning Rate | 0.1 | 0.095 |
| N_Estimators | 130 | 145 |

Tabla 14: Hiperparámetros de LightGBM en GridSearch vs Bayesian Optimization

Es interesante notar que en el caso del modelo optimizado por GridSearch se ha optado por utilizar un número mayor de árboles de menor profundidad, en vez de emplear menos árboles algo más profundos. Ahora, vamos a comparar el rendimiento de ambos modelos:

| Métrica | Train Bayesian | Test Bayesian | Train Grid | Test Grid |
|---|---|---|---|---|
| Accuracy | 0.9923 | 0.9826 | 0.9815 | 0.9757 |
| Balanced Accuracy | 0.9923 | 0.9492 | 0.9815 | 0.9457 |
| Precision | 0.9909 | 0.7241 | 0.9767 | 0.6363 |
| Recall | 0.9936 | 0.9130 | 0.9864 | 0.913 |
| $F_1$ | 0.9923 | 0.8077 | 0.9815 | 0.75 |
| ROC_AUC_SCORE | 0.9923 | 0.9493 | 0.9814 | 0.9457 |

Tabla 15: Métricas de LightGBM en entrenamiento y test

Podemos observar que la performance alcanzada por ambos métodos de optimización es ciertamente similar en términos generales. Sin embargo, en métricas clave como la precisión o el $F_1$-score se observa que el modelo optimizado mediante el método menos intensivo presenta unos mejores scores. Hipotetizamos que esto se debe a la posibilidad de evaluar muchos más puntos dentro del espacio y dirigir la búsqueda de manera más eficiente, lo que permite encontrar los máximos no sólo rápida sino también óptimamente.

Este punto nos lleva a discutir la diferente aproximación de GridSearch y Bayesian Optimization a la hora de resolver el problema. La hipótesis que barajamos para explicar este fenómeno es que la optimización bayesiana es capaz de evaluar mejor el espacio de posibles soluciones, puesto que cada evaluación está guiada tomando información del resultado de evaluaciones anteriores. Esto permite afinar mucho más la combinación de hiperparámetros escogida para el árbol. Por otra parte, los potenciales valores entre los que buscar no están definidos de manera rígida, sino que son valores continuos dentro de un rango definido (de ahí la necesidad de transformar los valores discretos, como el número de estimadores, previos a la evaluación en el modelo). Esto es posible gracias a que la búsqueda es guiada, lo que nos permite reducir el número de evaluaciones necesarias en la función objetivo (típicamente costosa), proporcionándonos mucha más flexibilidad en la búsqueda de hiperparámetros óptimos.

En el caso de GridSearch, cada vez que queramos probar una combinación de parámetros tenemos que realizar una evaluación de la función objetivo. Además, no podemos integrar información de evaluaciones pasadas para guiar nuestra búsqueda, por lo que tenemos que evaluar cada uno de los puntos que deseemos. Es por ello que, en la práctica, donde no tenemos una idea a priori del valor óptimo de la función a optimizar, el coste computacional del proceso hace que tengamos que

reducir el número de potenciales valores que podemos probar para cada uno de los hiperparámetros, así como el número de hiperparámetros a testar (para evitar problemas combinatorios). Esto se traduce en un número de evaluaciones mucho más grande, pero a la vez mucho menos óptimo, puesto que no podemos evaluar sino un subconjunto del espacio de posibles soluciones.

En conclusión, las razones por las que la optimización bayesiana acaba superando a la búsqueda intensiva pueden resumirse en:

1. Búsqueda guiada, informada por los resultados de evaluaciones pasadas
2. Mayor flexibilidad a la hora de escoger hiperparámetros dentro de un rango determinado
3. Mayor flexibilidad a la hora de combinar dichos hiperparámetros.

### 1.9.2. XGBoost

A continuación procedemos a entrenar el segundo modelo basado en descenso de gradiente: XGBoost. A diferencia de los modelos de boosting tradicionales, XGBoost incluye técnicas de regularización para evitar el sobreajuste y mejorar la capacidad de generalización del modelo. La regularización implica la adición de una penalización a la función de pérdida del modelo durante el entrenamiento, con el objetivo de limitar el tamaño de los parámetros del modelo. Esto ayuda a evitar el sobreajuste, ya que limita la complejidad del modelo y reduce la posibilidad de que se ajuste demasiado a los datos de entrenamiento. Esperamos que la inclusión de estas mejoras repercuta en un incremento de la performance del modelo en el conjunto de test.

Podemos observar que en este caso el número de hiperparámetros que hemos ajustado en el modelo es bastante superior a lo observado en el caso de LGBM. Los valores de hiperparámetros más importantes a destacar incluyen:

| Hiperparámetro | Valor |
|---|---|
| `Max depth` | 3 |
| `Num estimators` | 102 |
| `Subsample` | 0.1 |
| `Learning Rate` | 0.1 |
| `Gamma` | 0 |
| `Reg Alpha` | 0.1 |
| `Reg Lambda` | 0 |
| `Colsample ByTree` | 1 |

Tabla 16: Mejor combinación de hiperparámetros para el modelo de XGBoost

Podemos observar que la profundidad máxima es igual que la escogida para el modelo de Gradient Boosting anterior, mientras que el número de estimadores es mucho menor. Del mismo modo, hemos introducido parámetros de regularización nuevos:

- `subsample`: establece la fracción de muestras que se utilizarán para ajustar cada árbol. Si se establece en un valor menor que 1, el modelo será más rápido pero también menos preciso. En este caso, el valor óptimo es de 0.8, lo cual es bastante próximo al valor de 1, lo que puede contribuir a reducir el sobreaprendizaje del modelo.
- `colsample_bytree`: indica la fracción de características que se utilizarán para ajustar cada árbol. Si se establece en un valor menor que 1, el modelo será más rápido pero también menos preciso. En este caso, puesto que el valor es de 1, el parámetro no aplica.

- `gamma`: indica la cantidad mínima de reducción requerida para dividir un nodo. Un valor más alto significa que el modelo será más conservador al dividir nodos. En este caso podemos observar que el valor de 0 implica que no se ha aplicado esta reducción.

También hemos incluido dos hiperparámetros de regularización (`reg_alpha` y `reg_lambda`), que permiten hacer el modelo más conservador mediante regularización de los pesos. En este caso, sólo se ha escogido regularización L1 (`reg_alpha`), puesto que `reg_lambda` tiene un valor de 0.

Con respecto al perfil de importancia de variables, no se aprecian cambios con respecto a lo observado en el modelo LGBM de optimización bayesiana:

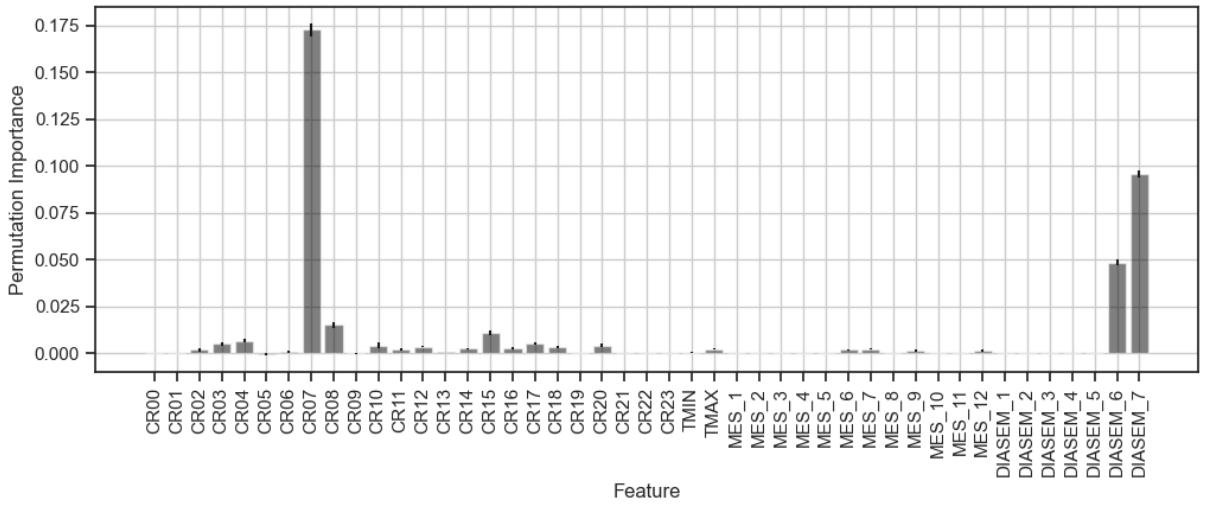


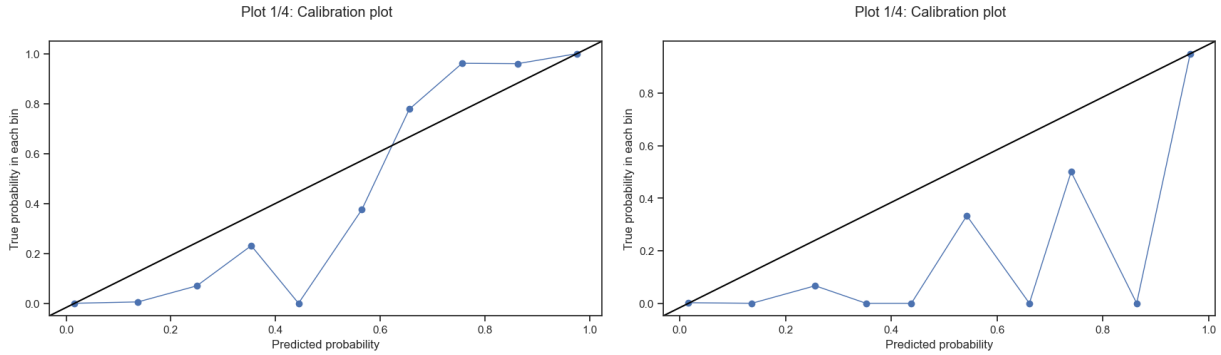

Figura 26: Importancia de las variables para el mejor modelo de XGBoost

Con respecto al rendimiento del modelo, podemos observar que el modelo de XGBoost presenta una performance mejor que el modelo de Random Forest y tan sólo ligeramente inferior al modelo LGBM, sobre todo en lo que respecta al conjunto de test:

| Métrica | Entrenamiento | Test |
|---|---|---|
| Accuracy | 0.9909 | 0.9809 |
| Balanced Accuracy | 0.9909 | 0.9483 |
| Precision | 0.9891 | 0.7 |
| Recall | 0.9927 | 0.9130 |
| $F_1$ | 0.99 | 0.79 |
| ROC_AUC_SCORE | 0.9909 | 0.9483 |

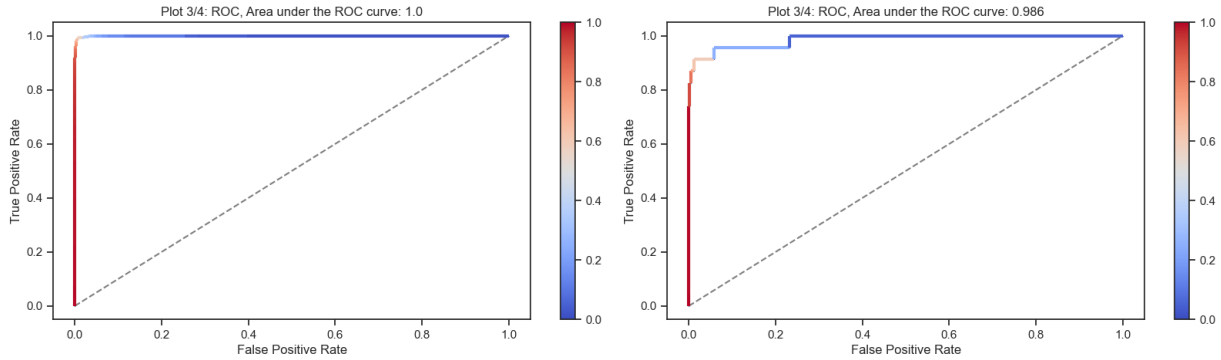
Tabla 17: Métricas de XGBoost en entrenamiento y test

Finalmente, los perfiles gráficos revelan un comportamiento del modelo similar a lo observado en el caso de LGBM, aunque con una performance algo más inestable en el caso del dataset de test.




(a) (b)

Figura 27: Curva de calibración en train (a) y test (b)




(a) (b)

Figura 28: Área bajo la curva en train (a) y test (b)

## 1.10. Stacking

Con el fin de combinar las capacidades de los modelos y conseguir reducir el error de generalización, podemos realizar un modelo de stacking. En primer lugar, es necesario definir los modelos a utilizar dentro del stacking. En este caso, escogeremos los mejores candidatos dentro de los modelos entrenados: Random Forest y los dos modelos de Gradient Boosting. Por el momento, no añadiremos pesos a los modelos. Dado que ya hemos hecho un proceso de entrenamiento de los modelos y únicamente queremos encontrar la forma óptima de combinarlos, emplearemos el parámetro `cv = prefit` al instanciar la clase `StackingClassifier`. Las matrices de confusión y métricas son:

$$\text{Train: } \begin{pmatrix} 2194 & 17 \\ 16 & 2195 \end{pmatrix}$$

$$\text{Test: } \begin{pmatrix} 547 & 6 \\ 2 & 21 \end{pmatrix}$$

| Métrica | Entrenamiento | Test |
|---|---|---|
| Accuracy | 0.9925 | 0.9861 |
| Balanced Accuracy | 0.9925 | 0.9511 |
| Precision | 0.9923 | 0.7777 |
| Recall | 0.9927 | 0.9130 |
| $F_1$ | 0.9925 | 0.84 |
| ROC_AUC_SCORE | 0.992 | 0.951 |

Tabla 18: Métricas del modelo de Stacking en entrenamiento y test

Podemos observar como el modelo de Stacking supone una ligera mejora con respecto al mejor modelo hasta ahora (LGBM). Esto se debe a que mediante este modelo estamos consiguiendo combinar los votos de varios modelos de arquitectura diferente para poder obtener unos resultados más generalizables al conjunto de test.

## 1.11. Árbol de decisión con PCA

El objetivo en esta sección es crear un número de variables reducido que contengan la información recogida en las variables CR00 a CR23, y crear un árbol sencillo basado en estas variables. Dado que carecemos de un conocimiento experto en el área para poder realizar una combinación manual de los coeficientes de manera óptima, hemos decidido optar por el uso de una técnica de reducción de dimensionalidad como es PCA. Esto nos permitirá obtener nuevas variables que sean combinación lineal de las variables de entrada (en este caso, los coeficientes), combinados en la dirección de mayor varianza. Si bien esto no implica que la combinación sea óptima, puede ser una primera aproximación a obtener variables que resuman la información de los coeficientes, contribuyendo a tener un menor espacio de variables de entrada que nos permita evitar la maldición de la dimensionalidad. A continuación, mostramos la varianza de las variables:

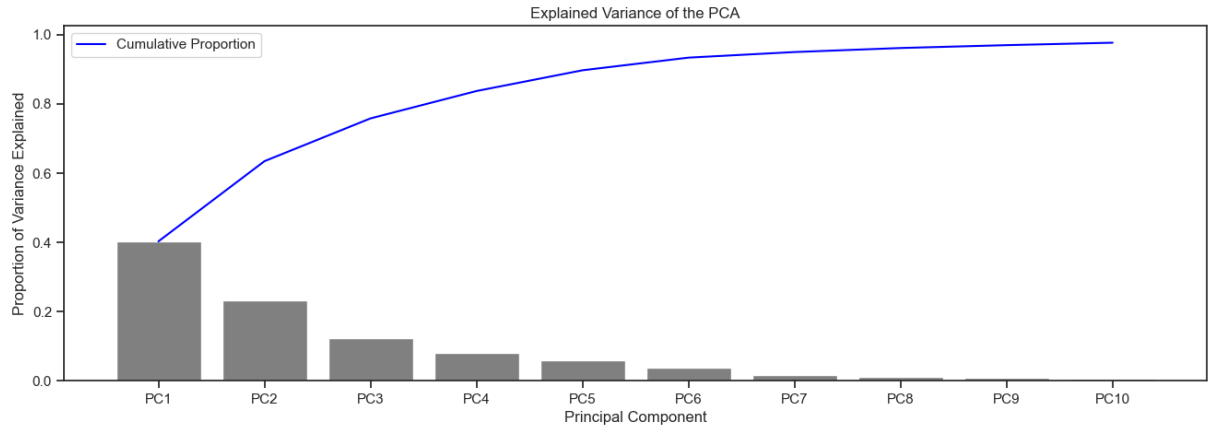



Figura 29: Varianza de las variables

Podemos observar cómo el uso de los primeros 6 componentes principales nos puede permitir resumir el 90 % de la varianza de los datos, requiriendo de tan solo 4 de ellos para explicar el 80 % de la misma.

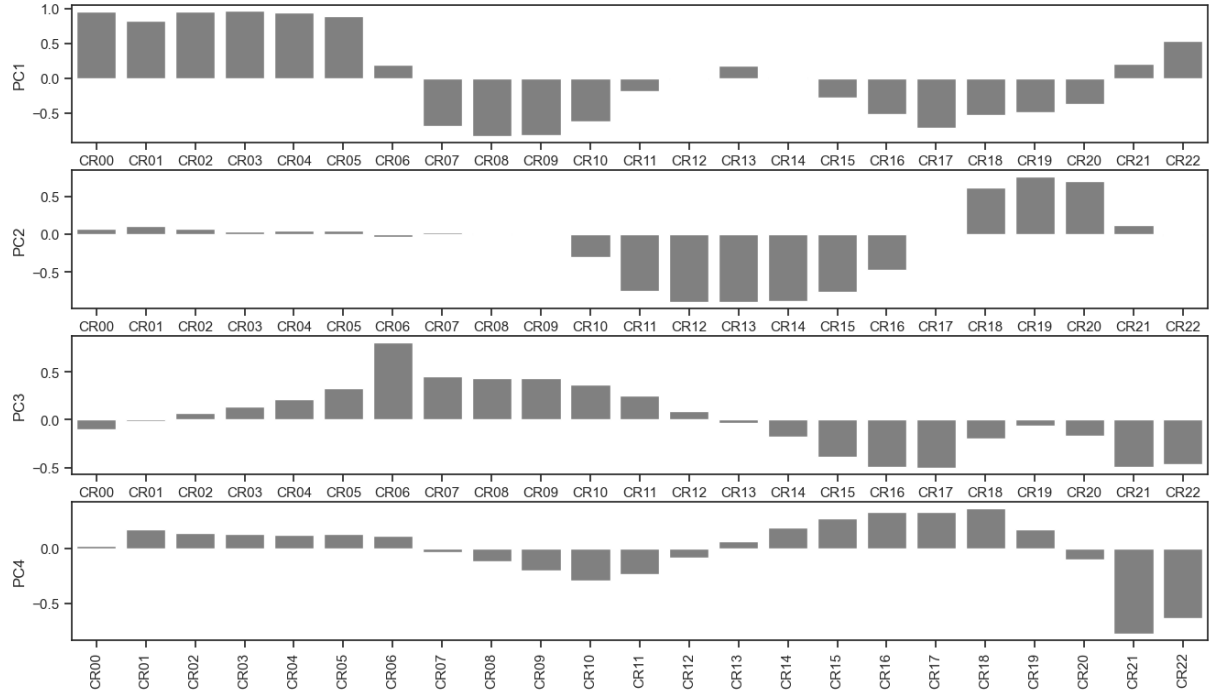



Figura 30: Loadings de las variables

Centrándonos en las primeros 4 componentes, podemos observar que el primero representa las horas pico de las horas Valle, puesto que presenta valores muy positivos para aquellos coeficientes de demanda baja (horas Valle correspondientes a la noche tardía y la madrugada), mientras que presenta valores negativos para horas de demanda alta (Primera y Segunda Punta).

El segundo coeficiente también presenta un perfil interesante, que permite permite discriminar la zona Valle del mediodía del resto de horas del día (algo que el primer componente no podía hacer de manera eficiente). El perfil del tercer componente parece centrarse en discriminar entre la mañana y la tarde. Finalmente, el cuarto componente tiene un perfil menos interpretable, que presenta un pico significativo de valor negativo para los coeficientes CR21 y CR22. Con estas nuevas variables entrenamos un árbol de decisión simple, el cual muestra el siguiente rendimiento:

| Métrica | Entrenamiento | Test |
|---|---|---|
| Accuracy | 0.8744 | 0.7864 |
| Balanced Accuracy | 0.8744 | 0.7846 |
| Precision | 0.8413 | 0.1323 |
| Recall | 0.9231 | 0.7826 |
| $F_1$ | 0.8803 | 0.2264 |
| ROC_AUC_SCORE | 0.8745 | 0.7846 |

Tabla 19: Métricas del modelo de árbol de decisión con PCA en entrenamiento y test

En general, se puede observar que el modelo está teniendo dificultades para generalizar a nuevos datos, como se puede ver por la diferencia en la *Precision* y el *Accuracy* entre los conjuntos de entrenamiento y prueba.

En el conjunto de entrenamiento, el modelo parece tener un desempeño aceptable, con un accuracy del 0.87 y un F1-score de 0.88. Sin embargo, la matriz de confusión sugiere que el modelo está

clasificando incorrectamente una cantidad significativa de muestras. Por otro lado, en el conjunto de prueba, el modelo tiene un recall bastante alto (0.92) lo que significa que es capaz de identificar la gran mayoría de las muestras positivas, pero su precisión es extremadamente baja en el connjunto de training (0.13). Esto sugiere que el modelo está clasificando muchos casos como positivas cuando en realidad no lo son, lo que se refleja en la matriz de confusión y en el bajo F1-score de 0.22.

En resumen, si bien se obtienen resultados aceptables a la hora de localizar positivos, esto es a costa de una gran cantidad de falsos positivos. Además, el modelo parece estar sufriendo de *overfitting*, ya que se desempeña bien en el conjunto de entrenamiento pero no generaliza adecuadamente a nuevos datos. Por tanto, no supone una mejora significativa frente a los modelos anteriores.

### 1.11.1. Comparativa de los modelos para los conjuntos de entrenamiento y test

| Model | Accuracy Train | Accuracy Test | Balanced accuracy Train | Balanced accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1-Score Train | F1-Score Test | ROC-AUC Train | ROC-AUC Test |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Decision Tree | 0.943012 | 0.949653 | 0.943012 | 0.932109 | 0.943413 | 0.437500 | 0.942560 | 0.913043 | 0.942986 | 0.591549 | 0.943012 | 0.932109 |
| Bagging Tree | 0.993442 | 0.970486 | 0.993442 | 0.901289 | 0.993219 | 0.593750 | 0.993668 | 0.826087 | 0.993443 | 0.690909 | 0.993442 | 0.901289 |
| Random Forest | 0.992085 | 0.977431 | 0.992085 | 0.925741 | 0.987455 | 0.666667 | 0.996834 | 0.869565 | 0.992122 | 0.754717 | 0.992085 | 0.925741 |
| LGBM Bayesian | 0.992311 | 0.982639 | 0.992311 | 0.949288 | 0.990979 | 0.724138 | 0.993668 | 0.913043 | 0.992322 | 0.807692 | 0.992311 | 0.949288 |
| LGBM GridSearch | 0.981456 | 0.975694 | 0.981456 | 0.945672 | 0.976713 | 0.636364 | 0.986431 | 0.913043 | 0.981548 | 0.750000 | 0.981456 | 0.945672 |
| XGBoost | 0.990954 | 0.980903 | 0.990954 | 0.948384 | 0.989184 | 0.700000 | 0.992763 | 0.913043 | 0.990971 | 0.792453 | 0.990954 | 0.948384 |
| Stacked | 0.992537 | 0.986111 | 0.992537 | 0.951097 | 0.992315 | 0.777778 | 0.992763 | 0.913043 | 0.992539 | 0.840000 | 0.992537 | 0.951097 |
| Decision Tree PCA | 0.874491 | 0.786458 | 0.874491 | 0.784614 | 0.841303 | 0.132353 | 0.923112 | 0.782609 | 0.880311 | 0.226415 | 0.874491 | 0.784614 |

Figura 31: Tabla comparativa de los modelos

En general, se puede observar que salvando los dos modelos de Árboles de Decisión simple (el primer modelo y el modelo de árbol utilizando variables derivadas de PCA), todos los modelos presentan tanto una *Precision* como un *Recall* razonables en ambos conjuntos de datos, destancando especialmente entre ellos los modelos de Gradient Boosting y el modelo de Stacking. Asimismo, se puede observar una diferencia notable en términos de rendimiento entre los conjuntos de datos de entrenamiento y test para todos los modelos, especialmente en términos de $F_1$-Score, *Recall* y *Precision*.

Con respecto a los modelos con mejor rendimiento, podemos observar que el modelo de Stacking (obtenido combinando Random Forest, XGBoost y LGBM) tiene la *Precision* más alta en ambos conjuntos de datos, así como la mayor $F_1$-Score y *Accuracy* en el conjunto de datos de test. Los modelos de Stacking, XGBoost, LightGBM y RandomForest tienen una Precision elevada, contrastando con la baja Precision de los Árboles de Decisión Simple y el modelo de Bagging.

A pesar de que no hemos conseguido reducir completamente el problema de los falsos positivos, podemos ver cómo el entrenamiento secuencial de los modelos de creciente complejidad nos ha ido ayudando a mejorar las métricas de Precisión y $F_1$-score, llegando a unos niveles de más del 77 % y 84 % para el mejor de nuestros modelos.

A continuación, mostramos las curvas de calibración de todos los modelos. Hemos dibujado la curva de calibración ideal, con la cual se comparan las curvas de calibración de todos los modelos:

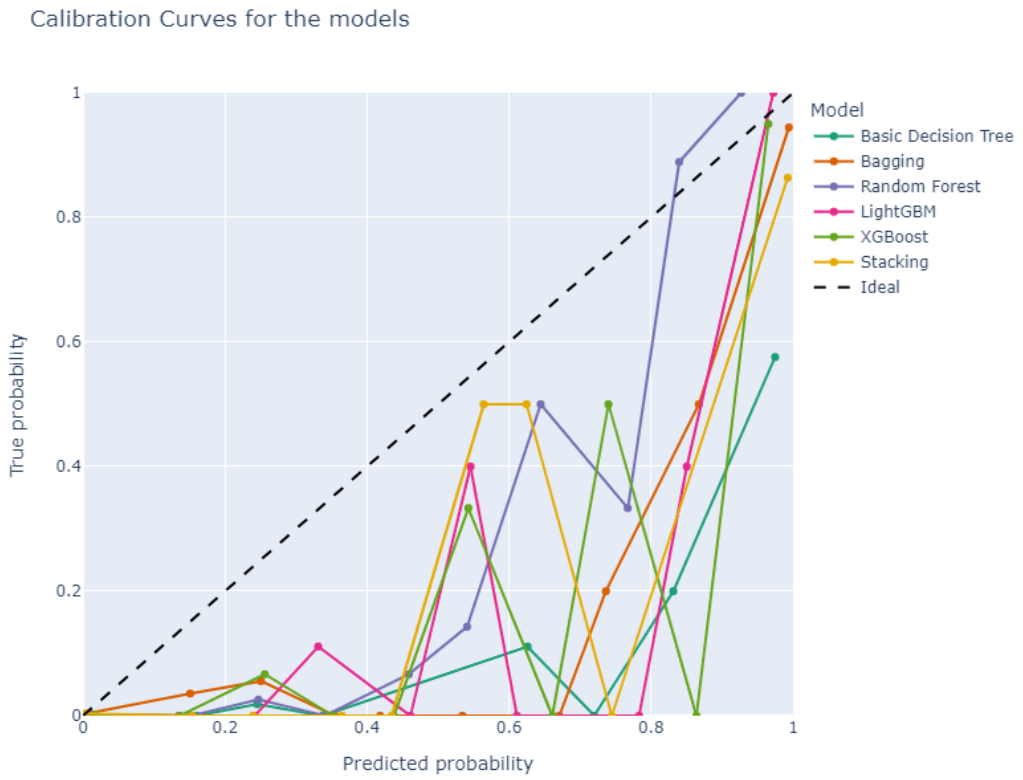



Figura 32: Comparación curvas de calibración

Esta gráfica es interactiva y, en el `Jupyter Notebook`, podemos mostrar o esconder cada modelo clickando sobre la leyenda. Podemos ver cómo ninguno de los modelos consigue adaptarse de manera ideal a la curva, puesto que todos ellos tienden a sobreestimar la probabilidad de que la clase sea positiva. Sin embargo, entre ellos cabe destacar el modelo de Random Forest como el que mejor se aproxima a la curva ideal.

En esta última gráfica tenemos las predicciones (probabilidades) para cada modelo en los conjuntos de entrenamiento y validación. Hemos dividido el intervalo [0, 1] en 10 intervalos iguales y para cada uno de estos puntos mostramos el porcentaje de predicciones para cada clase. Así, en cada subgráfica se superponen las probabilidades para las clases positivas y negativas:

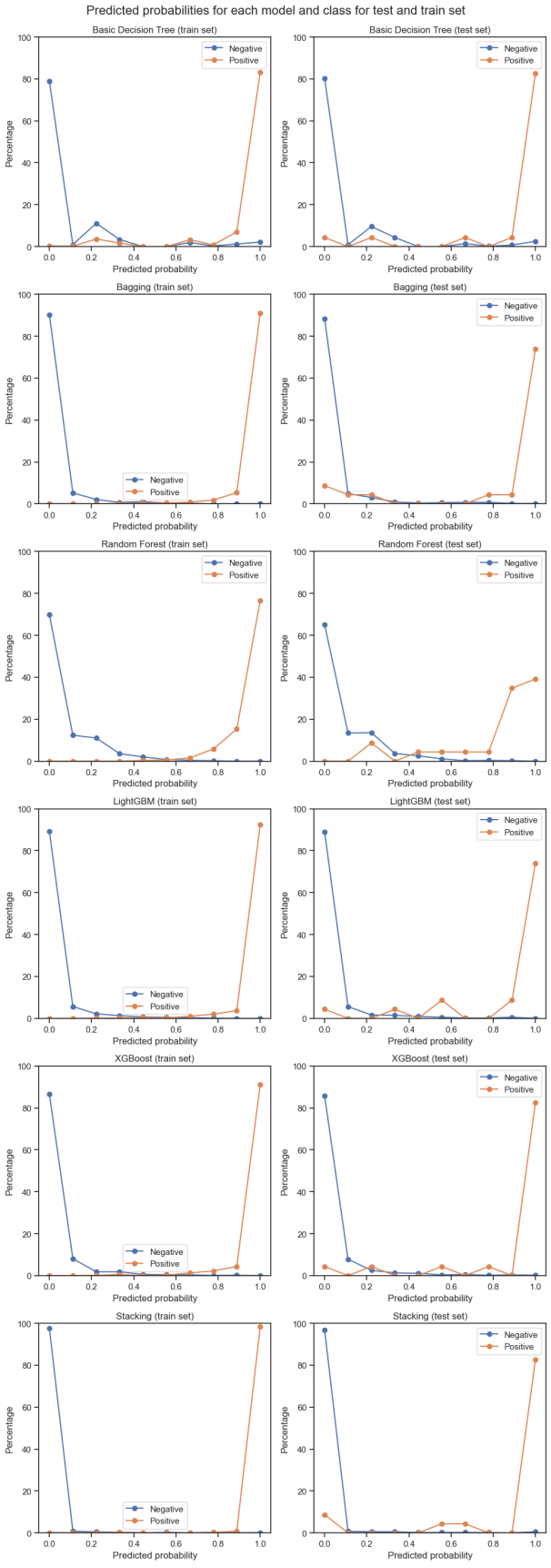



Figura 33: Comparación de probabilidades para cada clase

En el conjunto de entrenamiento, los modelos de Boosting y Stacking presentan prácticamente un 100 % de probabilidad en el 0 para la clase negativa y en el 1 para la clase negativa. Sin embargo, en el conjunto de test los 3 modelos tienen en torno a un 20 % de probabilidad en el 1 para la clase positiva. Esto nos indica que hay del orden de 4 o 5 falsos negativos en todos estos modelos (ya que en el conjunto de entrenamiento tenemos 21 positivos en total), lo cual va acorde a lo observado en sus respectivas matrices de confusión. Por otro lado, si bien el modelo de Random Forest presenta unos resultados ligeramente mejores que el resto de modelo (expecto el modelo de Stacking) según las métricas, aquí podemos ver que el modelo no es todo lo discriminante que querríamos - las probabilidades no se encuentran concentradas en torno al 0 y el 1 sino repartidas.

# 2. Validación final de los modelos

## 2.1. Comparación de los errores obtenidos en validación

En esta sección de hace uso del dataset `VAL_DATOS_DEM_C3.csv`. Hemos realizado las predicciones de todos los modelos para el año 2022, y vamos a realizar la comparativa de los errores obtenidos en validacion con los observados previamente en ajuste/test mediante la siguiente tabla:

| Model | Accuracy Train | Accuracy Test | Accuracy Validation | Balanced accuracy Train | Balanced accuracy Test | Balanced accuracy Validation | Precision Train | Precision Test | Precision Validation | Recall Train | Recall Test | Recall Validation | F1-Score Train | F1-Score Test | F1-Score Validation | ROC-AUC Train | ROC-AUC Test | ROC-AUC Validation |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Decision Tree | 0.943012 | 0.949653 | 0.901961 | 0.943012 | 0.932109 | 0.885088 | 0.943413 | 0.437500 | 0.282609 | 0.942560 | 0.913043 | 0.866667 | 0.942986 | 0.591549 | 0.426230 | 0.943012 | 0.932109 | 0.885088 |
| Bagging Tree | 0.993442 | 0.970486 | 0.960784 | 0.993442 | 0.901289 | 0.852047 | 0.993219 | 0.593750 | 0.523810 | 0.993668 | 0.826087 | 0.733333 | 0.993443 | 0.690909 | 0.611111 | 0.993442 | 0.901289 | 0.852047 |
| Random Forest | 0.992085 | 0.977431 | 0.969188 | 0.992085 | 0.925741 | 0.824561 | 0.987455 | 0.666667 | 0.625000 | 0.996834 | 0.869565 | 0.666667 | 0.992122 | 0.754717 | 0.645161 | 0.992085 | 0.925741 | 0.824561 |
| LightGBM | 0.992311 | 0.982639 | 0.991597 | 0.992311 | 0.949288 | 0.963743 | 0.990979 | 0.724138 | 0.875000 | 0.993668 | 0.913043 | 0.933333 | 0.992322 | 0.807692 | 0.903226 | 0.992311 | 0.949288 | 0.963743 |
| XGBoost | 0.990954 | 0.980903 | 0.985994 | 0.990954 | 0.948384 | 0.960819 | 0.989184 | 0.700000 | 0.777778 | 0.992763 | 0.913043 | 0.933333 | 0.990971 | 0.792453 | 0.848485 | 0.990954 | 0.948384 | 0.960819 |
| Stacking | 0.992537 | 0.986111 | 0.983193 | 0.992537 | 0.951097 | 0.927485 | 0.992315 | 0.777778 | 0.764706 | 0.992763 | 0.913043 | 0.866667 | 0.992539 | 0.840000 | 0.812500 | 0.992537 | 0.951097 | 0.927485 |
| Decision Tree PCA | 0.874491 | 0.786458 | 0.862745 | 0.874491 | 0.784614 | 0.641520 | 0.841303 | 0.132353 | 0.130435 | 0.923112 | 0.782609 | 0.400000 | 0.880311 | 0.226415 | 0.196721 | 0.874491 | 0.784614 | 0.641520 |

Figura 34: Comparación de métricas para los 3 conjuntos de datos

En esta tabla resumen podemos observar la comparación de las métricas obtenidas en validación con respecto a los conjuntos de train y test para cada uno de los modelos evaluados. En general y como es de esperar, hay una caída en la performance de los modelos con respecto a lo observado en train y test. Sin embargo, esta caída no es excesivamente significativa con respecto a lo evaluado en el conjunto de test, lo que nos indica que los esfuerzos para aumentar la capacidad de generalización de nuestros modelos han dado sus frutos.

Aquellos que presentan una disminución de performance más significativa son los modelos de árbol de decisión, que también habíamos identificado como aquellos que sufrían de mayores problemas de sobreaprendizaje a los datos. Por el contrario, modelos más complejos como LGBM, XGBoost y Stacking presentan un nivel de robustez significativo, con métricas comparables tanto en testing como en validation. De hecho, los modelos de LGBM y XGBoost presentan una performance mejor aún en el conjunto de validación que en el conjunto de test, lo que nos demuestra que el conjunto de test no siempre es un buen proxy de cómo se va a comportar el modelo en el escenario real de producción y resalta la importancia de reservar un conjunto de datos relativamente grande para realizar el análisis de validación externa de los modelos.

## 2.2. Perfil de los días peor clasificados

Hemos realizado una función que obtiene, por cada modelo, los días que no se han clasificado correctamente en el conjunto de datos de validación. También hemos dibujado los días para los cuales se cometen errores:

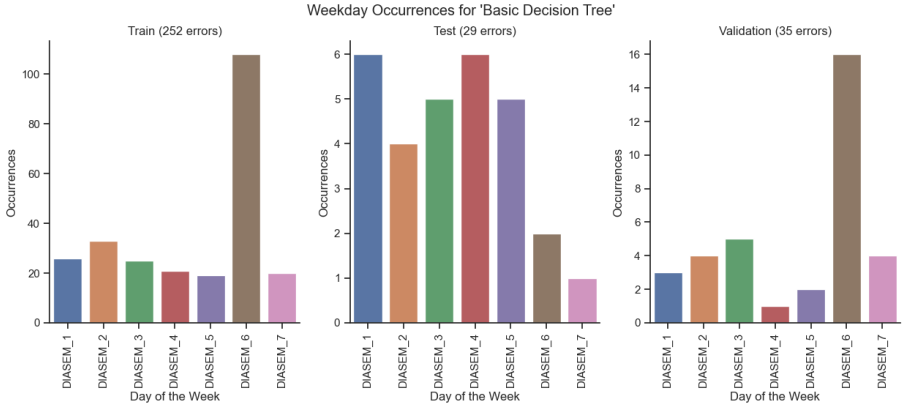



Figura 35: Principales errores para el Árbol de Decisión

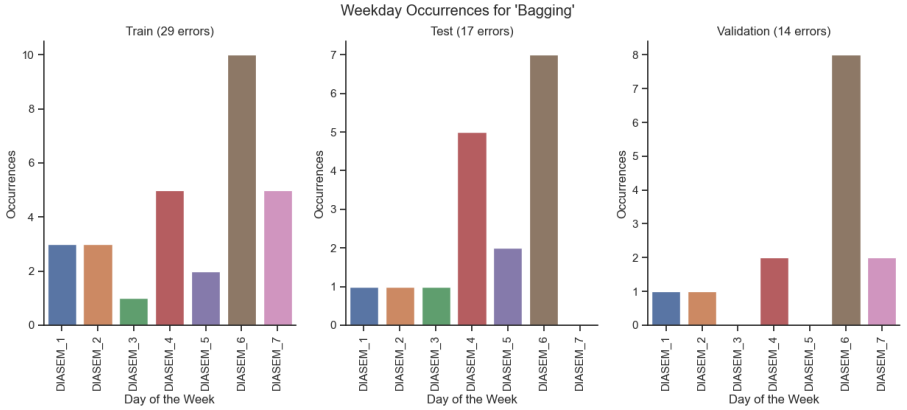



Figura 36: Principales errores para el modelo de Bagging

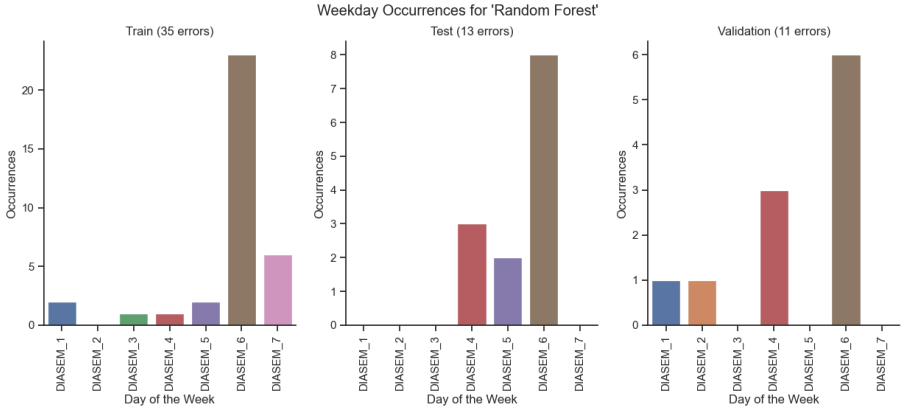



Figura 37: Principales errores para el Random Forest

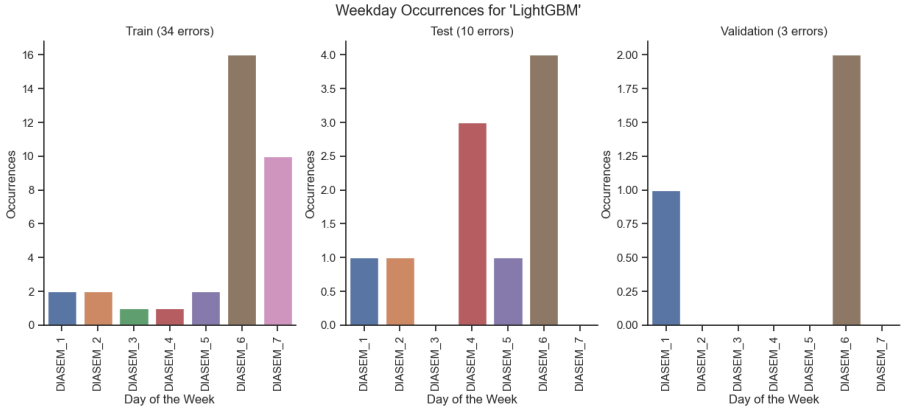



Figura 38: Principales errores para LightGBM

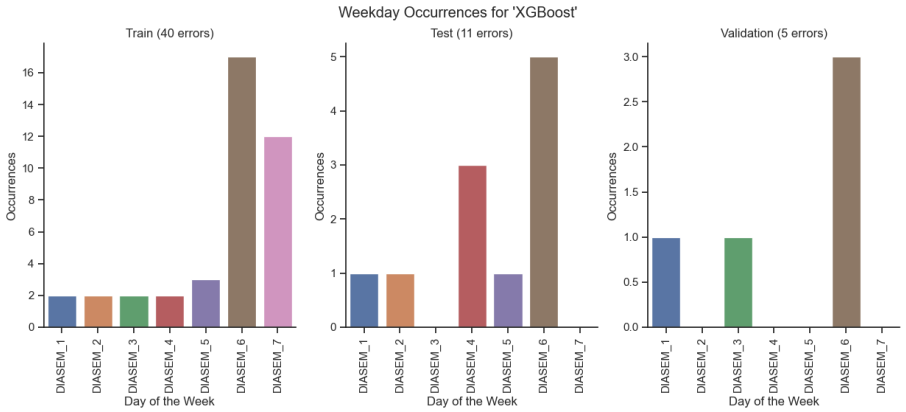



Figura 39: Principales errores para XGBoost

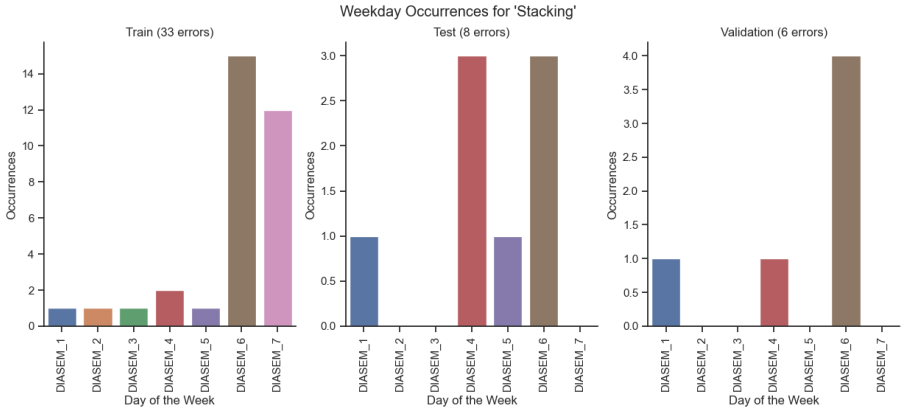



Figura 40: Principales errores para Stacking

Como podemos observar en las gráficas anteriores, los días más frecuentes en los que se cometen errores parecen ser sábados y domingos. Esto parece tener sentido con la realidad, puesto que esperamos que el perfil de esos días sea bastante parecido a lo que corresponde a un día festivo. Contrastándolo con lo observado en el análisis de importancia de los modelos, podemos ver que para todos ellos la variable que indica si el día es sábado o domingo es muy relevante para poder

decidir si clasificarlo como festivo o no. Con respecto al mes del año, la mayor parte de los errores se concentran en validación en el mes de agosto, un mes bastante atípico en el perfil de demanda del calendario, puesto que es cuando la mayor parte de personas tienen vacaciones de verano. Por ello, es posible que un perfil de demanda en un día de diario de agosto no sea muy diferente al observado en un festivo, haciendo que la diferenciación entre ambas clases sea complicada hasta para los modelos más complejos.

## 2.3. Perfil de los días peor clasificados

En esta sección, hemos realizado una función que localiza los $n$ días de la semana con mayor número de errores. Con esto, hemos podido comparar el perfil de demanda de los días de la semana más difíciles de clasificar con respecto al perfil de demanda típico en uno de esos días:

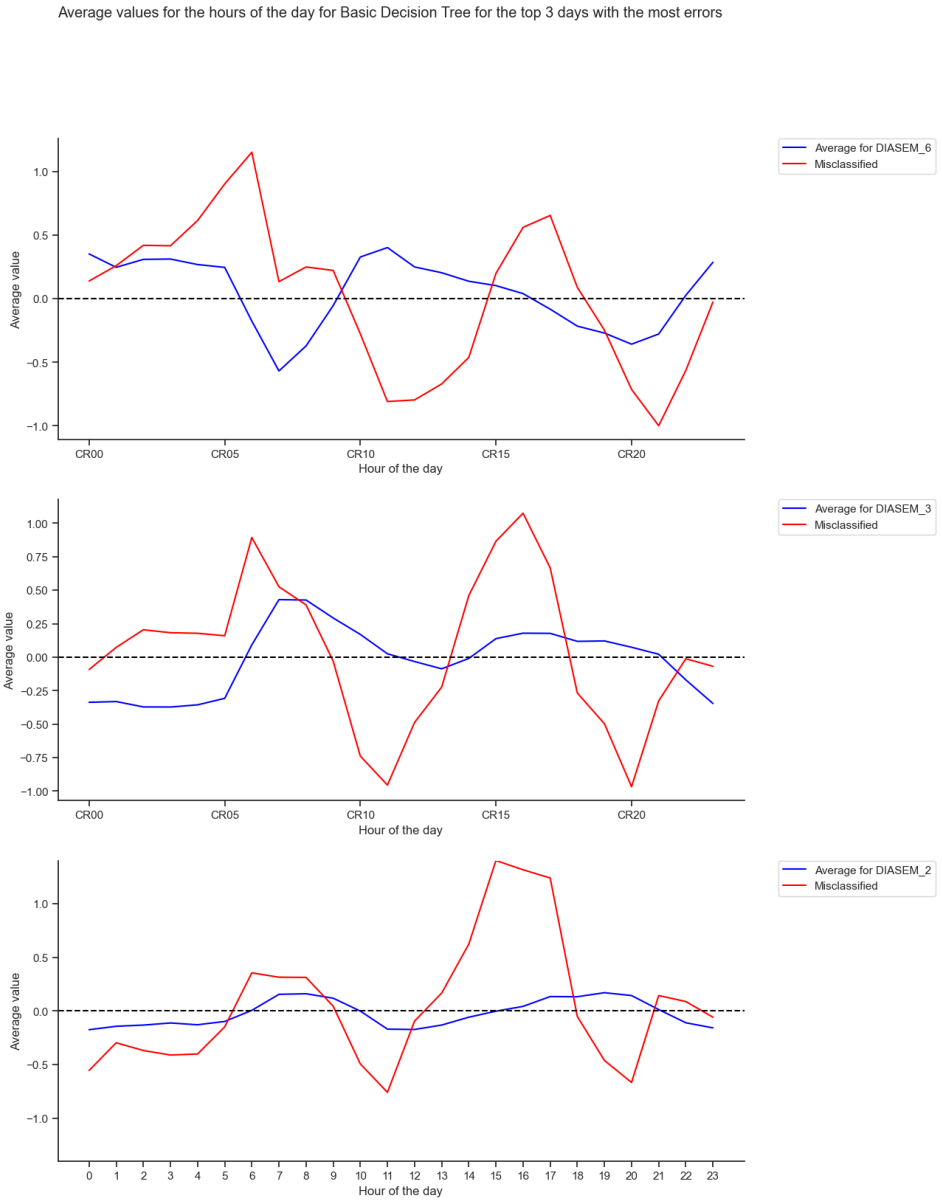



Figura 41: Comparación de los 3 días con más fallos para el Árbol de Decisión Simple

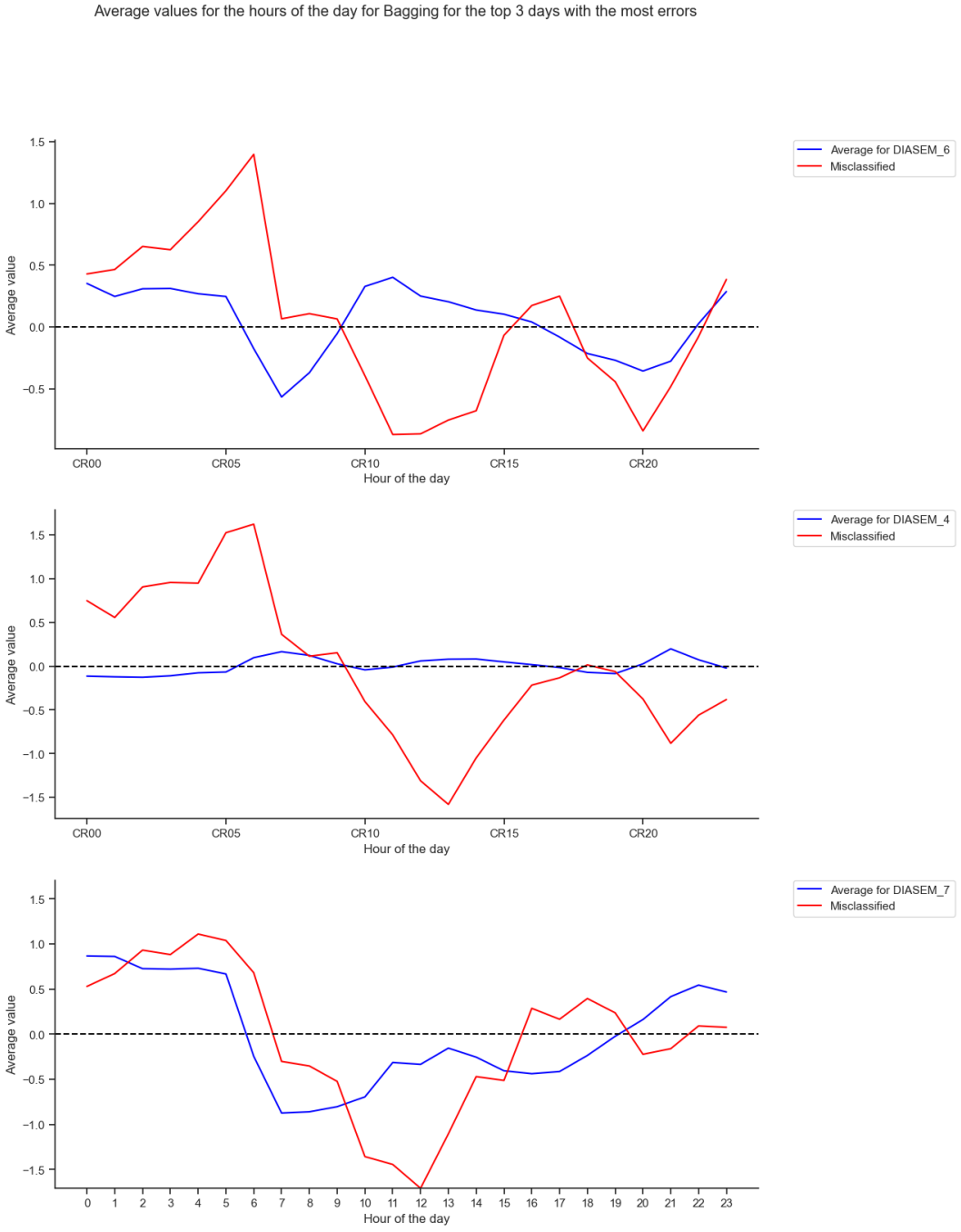



Figura 42: Comparación de los 3 días con más fallos para el modelo de Bagging

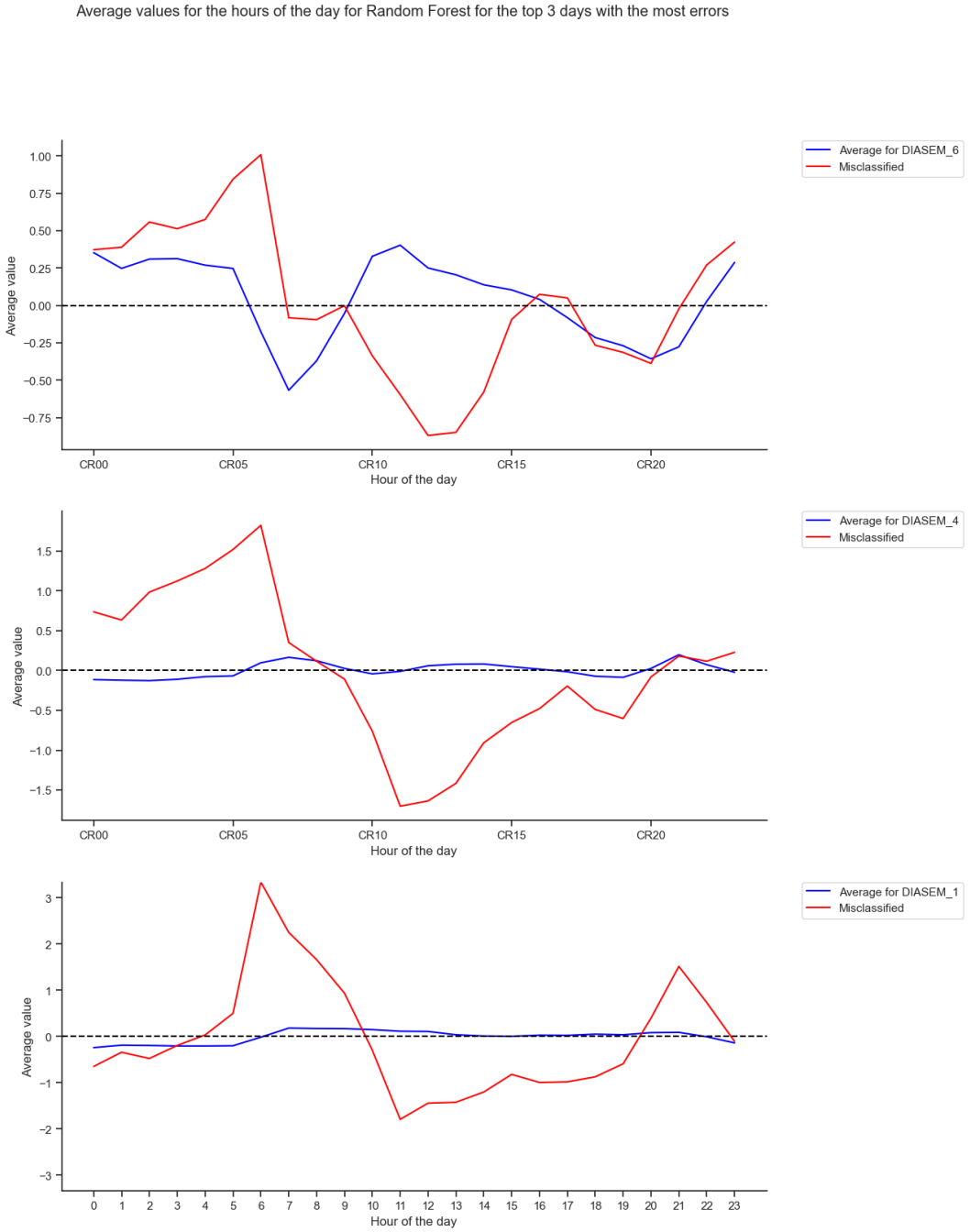



Figura 43: Comparación de los 3 días con más fallos para el modelo de Random Forest

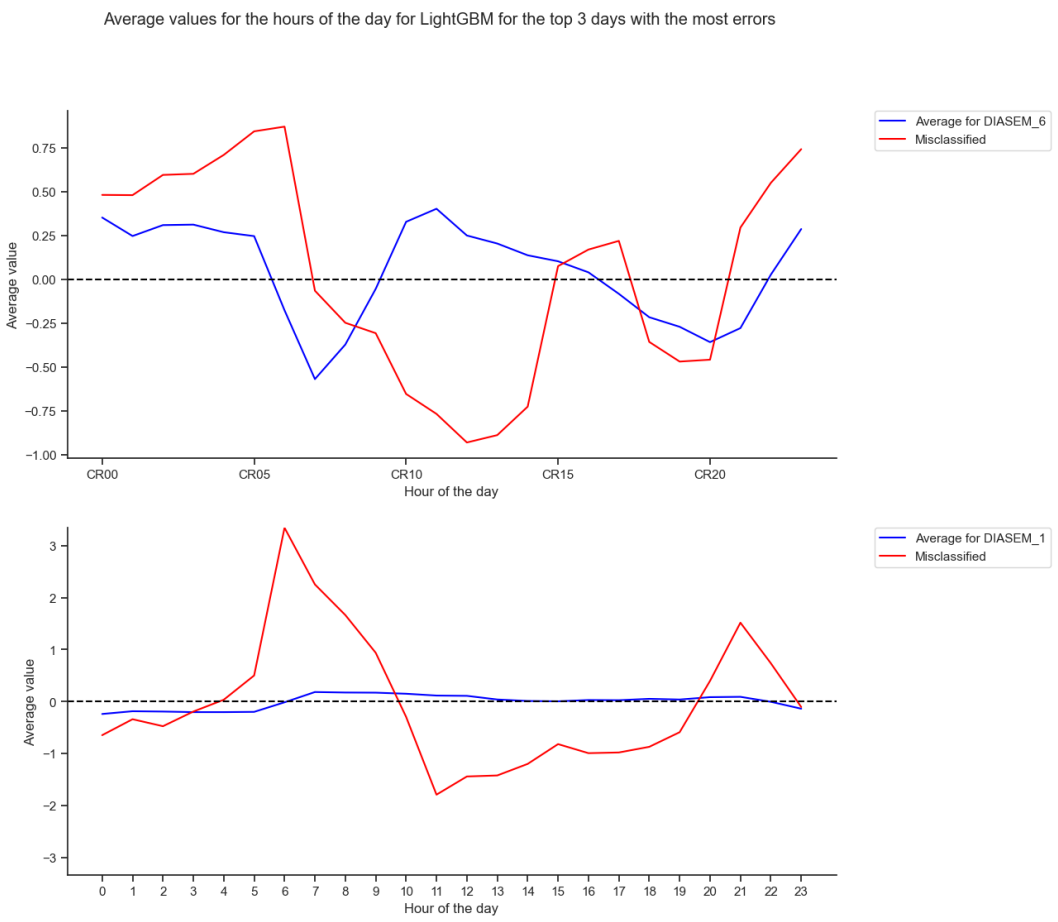



Figura 44: Comparación de los 3 días con más fallos para el modelo de LightGBM

***Nota***: *Como el modelo de LightGBM únicamente presenta fallos para 2 tipos de días (lunes y sábados), solo se muestran 2 días en la gráfica.*

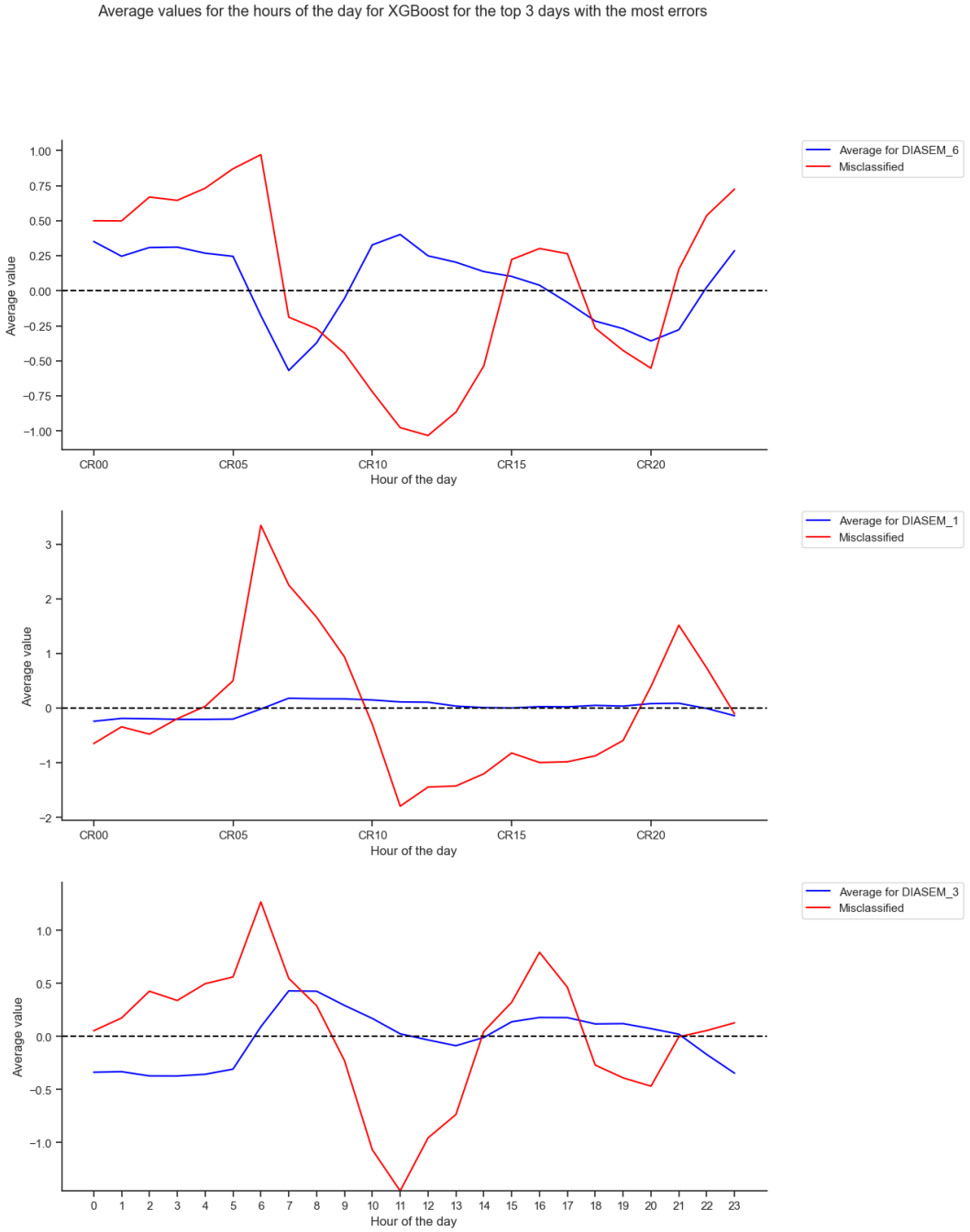



Figura 45: Comparación de los 3 días con más fallos para el modelo de XGBoost

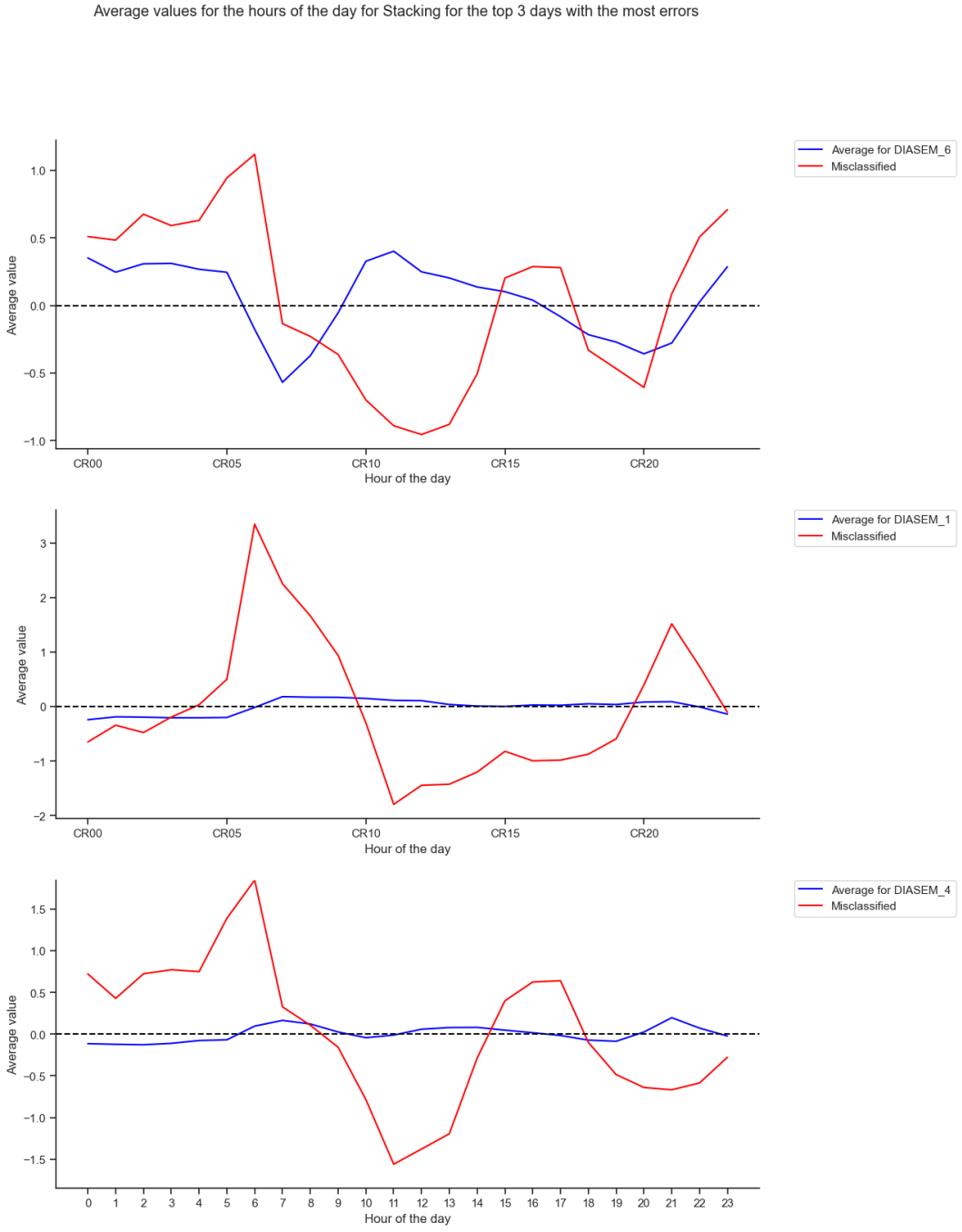



Figura 46: Comparación de los 3 días con más fallos para el modelo de Stacking

En todos los gráficos podemos observar que el perfil del día mal clasificado difiere sustancialmente del perfil medio que consideramos como típico para ese día. Esto nos lleva a pensar que se trata de días atípicos, en los que el modelo tiene grandes dificultades para realizar la clasificación.