Master Big Data y Business Analytics

Machine Learning

Predicción de cancelaciones de reservas en establecimientos hoteleros

Alumno: Trosman, Denis

Profesores: Gómez, Daniel; Gutiérrez, Inmaculada.

Fecha de entrega extraordinaria: 21 de septiembre de 2023

Contenidos

Introducción

Los canales en línea para la reserva de hoteles han tenido un impacto significativo en las posibilidades de reserva y en el comportamiento de los clientes. En muchos casos, las reservas de hotel se cancelan debido a diversas razones, como cambios en los planes o conflictos en la programación. Esto a menudo se facilita al ofrecer la opción de cancelar de forma gratuita o a bajo costo, lo que es beneficioso para los huéspedes del hotel. Sin embargo, para los hoteles, esto puede ser un factor menos deseable, ya que posiblemente disminuye sus ingresos.

Por este motivo, el análisis y modelado de datos puede ser de gran ayuda para entender que razones y motivos se encuentran detrás de la decisión de los clientes para cancelar reservas de hoteles, otorgando información para prever estas mismas y ayudando a identificar potenciales clientes que cancelarán sus reservas.

Kaggle[1] ofrece un set de datos que nos da oportunidad de encontrar tendencias y probar modelos de clasificación binaria para este objetivo. En la Tabla 1 pueden observarse las descripciones de cada variable del set de datos.

Tabla 1. descripción de variables

Variable	Descripción
(ID reserva) Booking_id	identificador único de cada reserva
(Número de adultos) no_of_adults	número de adultos indicados en la reserva
(Número de niños) no_of_children	número de niños indicados en la reserva
(Número de noches de fin de semana) no_of_weekend_nights	número de noches de fin de semana que contiene la reserva en el hotel
(Número de noches de semana) no_of_week_nights	número de noches de semana que contiene la reserva en el hotel
(Tipo de plan de comida) type_of_meal_plan	tipo de plan alimenticio reservado (planes 1 a 3, sin plan, u otras opciones)
(Aparcamiento de vehiculo) required_car_parking_space	indicación binaria de necesidad de aparcamiento de vehículo (0=no, 1=sí)
(Tipo de habitacion reservada) type_of_meal_plan	tipo de habitación reservada (en códigos de INN hoteles, del 1 al 7)
(Tiempo de espera) lead_time	número de días entre reserva y fecha de llegada al hotel
(Año de llegada) arrival_year	año de llegada
(Mes de llegada) arrival_month	mes de llegada
(Fecha de llegada) arrival_date	fecha de llegada
(Segmento de mercado) market_segment_type	tipo de segmento de mercado (online, offline, corporativo, etc.)
(Huesped repetido) repeated_guest	indicador de huésped repetido (1) o no nuevo (0)
(N° de cancelaciones previas) no_of_previous_cancellations	número de veces que ha cancelado una reserva en el pasado

[1] https://www.kaggle.com/datasets/ahsan81/hotel-reservations-classification-dataset

(N° de no cancelaciones previas) no_of_previous_bookings_not_canceled	número de veces que no ha cancelado una reserva en el pasado
(Precio promedio por habitación) avg_price_per_room	precio diario promedio por habitación (precio variable día a día)
(N° de pedidos especiales) no_of_special_requests	número de pedidos especiales (pisos altos, vistas desde la habitación, etc)
(Estado de la reserva) booking_status	estado de la reserva.

Fuente: elaboración propia en base a Kaggle.

En este ejercicio, intentaremos predecir la variable **booking status (estado de la reserva)**, cual indica si una reserva fue cancelada o no. Originalmente la variable es categórica, con valores Canceled (cancelada) o Not_Canceled (no cancelada). Esta será transformada en binaria (0 y 1) para poder realizar predicciones correctamente.

De esta manera, la estructura del trabajo comenzará con una primera etapa de análisis exploratorio del conjunto de datos escogido, continuará con la construcción de modelos predictivos para estimar los estados de las reservas, y finalizará con la elección del mejor de estos modelos.

Análisis exploratorio de datos

La base de datos cuenta con 36.275 registros y 19 variables, de tipo numéricas y otras categóricas. Para este ejercicio se ha decidido realizar un muestreo aleatorio de 5.000 observaciones para poder ejecutar los modelos de alta carga computacional sin problemas, confirmando que la distribución sea la misma que en el conjunto de datos original.

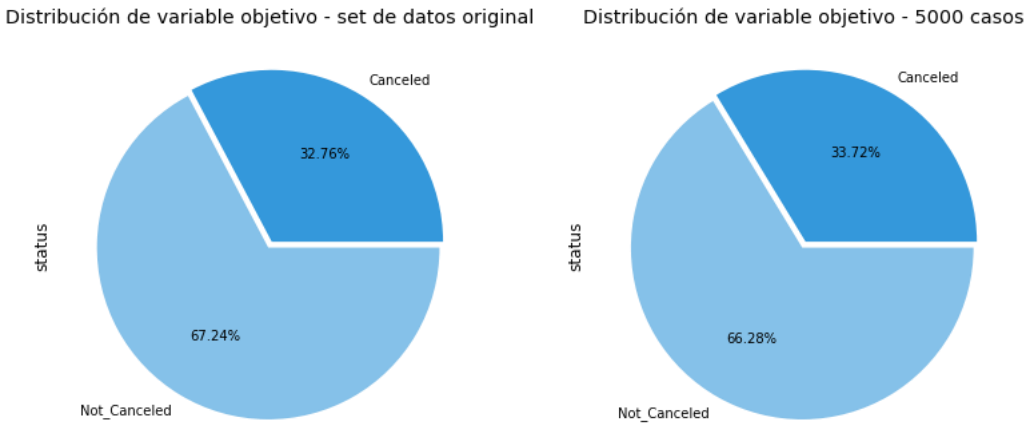
El set de datos no contiene datos nulos, como enseña la Figura 1:

Figura 1. Cantidad de nulos y clase de variables.

```
Data columns (total 19 columns):
 #   Column                                Non-Null Count  Dtype
---  ------                                --------------  -----
 0   Booking_ID                            36275 non-null  object
 1   no_of_adults                          36275 non-null  int64
 2   no_of_children                        36275 non-null  int64
 3   no_of_weekend_nights                  36275 non-null  int64
 4   no_of_week_nights                     36275 non-null  int64
 5   type_of_meal_plan                     36275 non-null  object
 6   required_car_parking_space            36275 non-null  int64
 7   room_type_reserved                    36275 non-null  object
 8   lead_time                             36275 non-null  int64
 9   arrival_year                          36275 non-null  int64
 10  arrival_month                         36275 non-null  int64
 11  arrival_date                          36275 non-null  int64
 12  market_segment_type                   36275 non-null  object
 13  repeated_guest                        36275 non-null  int64
 14  no_of_previous_cancellations          36275 non-null  int64
 15  no_of_previous_bookings_not_canceled  36275 non-null  int64
 16  avg_price_per_room                    36275 non-null  float64
 17  no_of_special_requests                36275 non-null  int64
 18  booking_status                        36275 non-null  object
dtypes: float64(1), int64(13), object(5)
memory usage: 5.3+ MB
```

La distribución de la variable objetivo en el set de datos original es de 67,2% de no cancelaciones (0s) y 32,7% de cancelaciones (1s). Luego de un muestreo aleatorio de cinco mil casos, la distribución cambia solo levemente a 66,3% y 33,7%, como enseña la Figura 2 debajo.

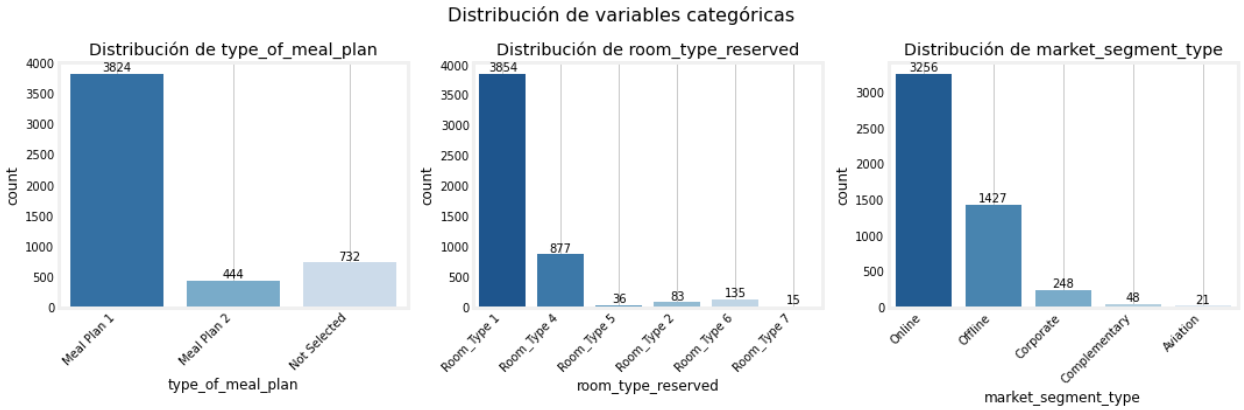
Figura 2. Distribución de variable objetivo original y muestreada.



Notamos que hay un leve desbalanceo, ya que la distribución de la variable objetivo no es exactamente 50-50. Sin embargo, se considera que el desbalance no es tal para necesitar técnicas de rebalanceo como *oversampling, undersampling o SMOTE.* Lo que sí hará falta es tener cuidado a la hora de elegir la métrica para comparar los modelos de predicción, lo cual será evaluado en los próximos apartados.

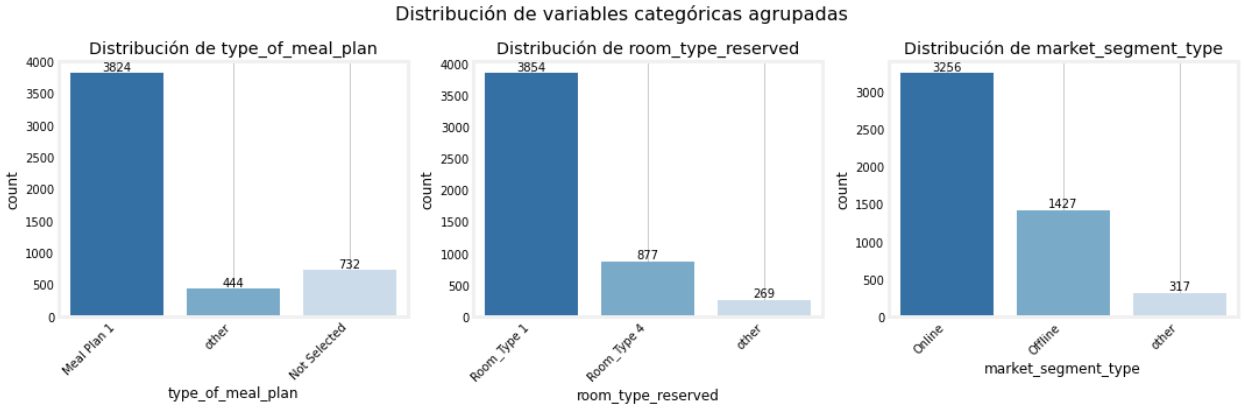
Como se enseñó en la Figura 1, la base de datos cuenta con diecinueve variables, de las cuales cinco son categóricas (*object)* y catorce numéricas (*int/float).* Como se mencionó previamente, una de las cinco variables categóricas es la binaria a predecir, *booking_status*, cual será convertida a binaria numérica. Otra de estas es el ID de la reserva, cual no contiene información relevante y será eliminada. Esto nos deja con tres variables categóricas, cuyas distribuciones se presentan en la Figura 3.

Figura 3. Distribución de las variables categóricas.



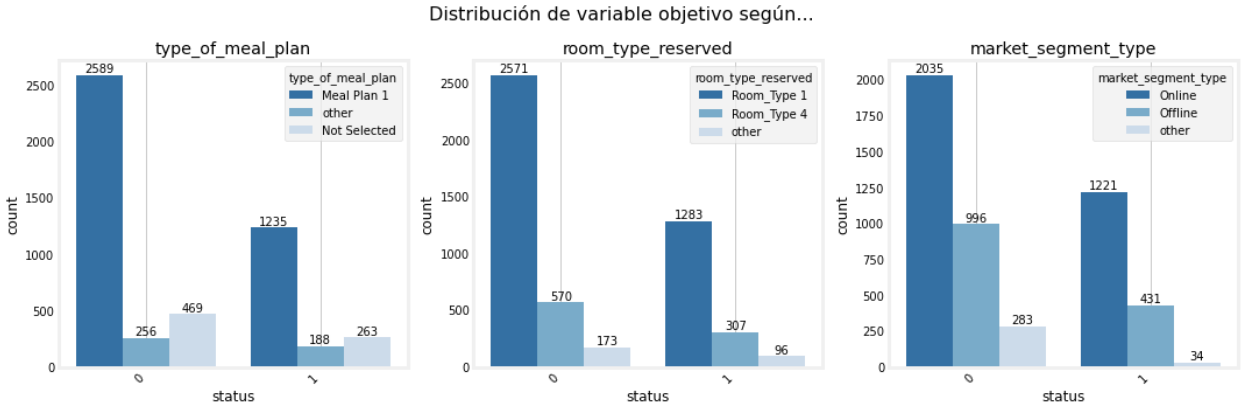
Observamos que hay grupos minoritarios, como por ejemplo tipos de habitaciones con poca frecuencia. Por este motivo, se ha procedido a agrupar estas categorías minoritarias, como muestra la Figura 4.

Figura 4. Distribución de variables categóricas agrupadas.



Lo que nos interesa, sin embargo, es la relación de estas con la variable objetivo a predecir. Para esto, podemos ver si hay diferencias entre la distribución de las variables categóricas según cancelaciones o no de las reservas. Como enseña la Figura 5, tanto las reservas canceladas como las no canceladas muestran una distribución parecida para las tres variables categóricas presentadas. Es decir, el plan de comida número 1 es el más común tanto para las reservas canceladas como para las no canceladas, por ejemplo.

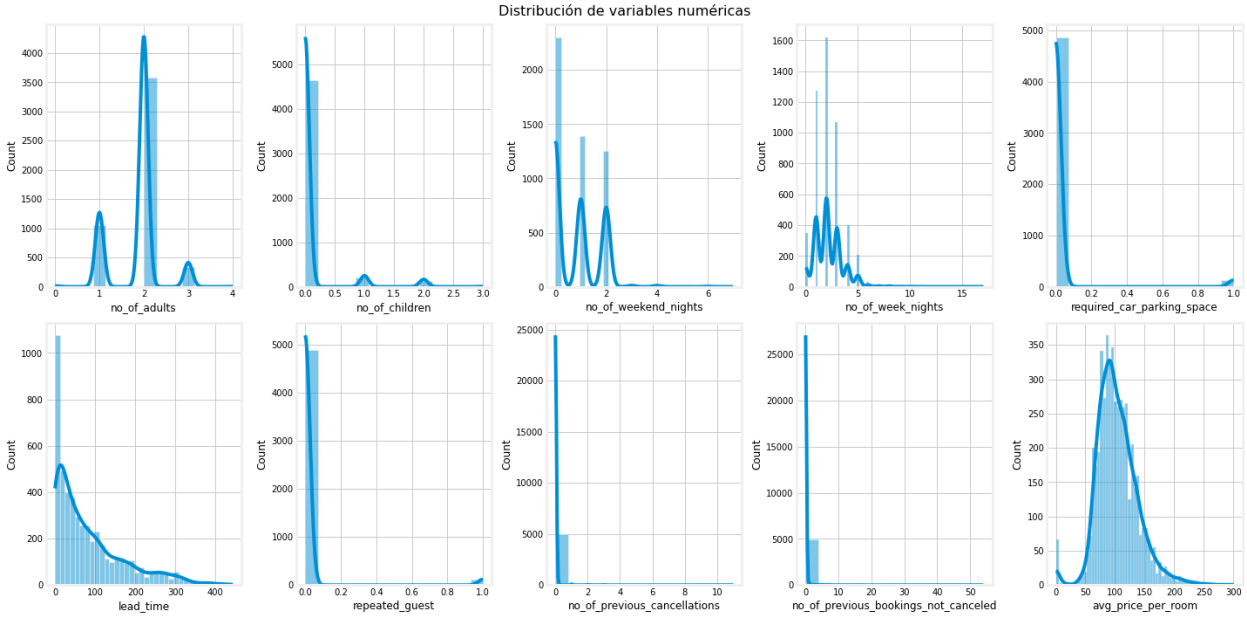
Figura 5. Distribución de variable objetivo según variables categóricas agrupadas.



Como último análisis de variables categóricas, se ha revisado la existencia de relación entre ellas mismas utilizando la V-Cramer. La relación más alta fue de 0.08, por lo que no hay relación a primera vista.

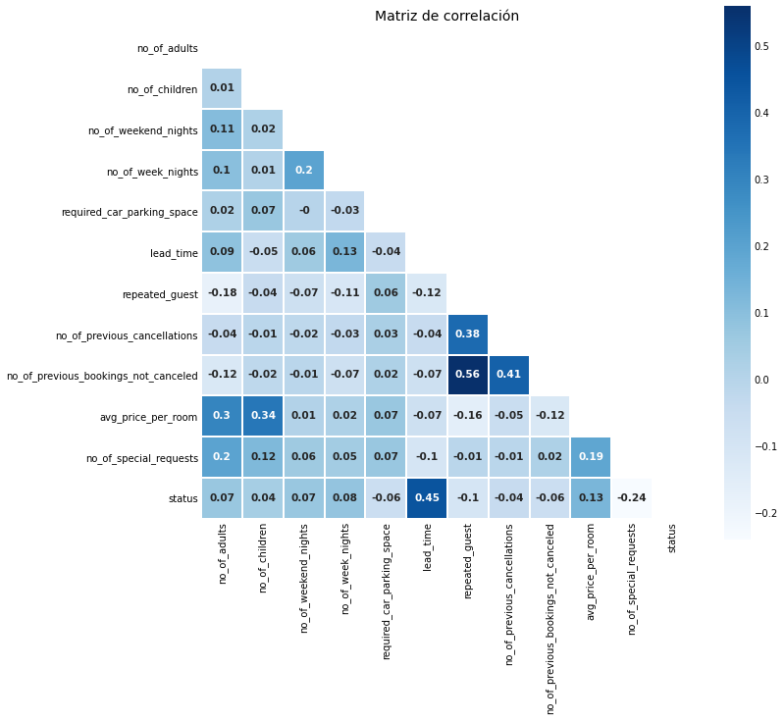
Un análisis parecido puede realizarse con las variables numéricas. Primero, veamos las distribuciones de cada una de ellas. La Figura 6 nos da información sobre esto. Vemos que la mayoría de las observaciones están concentrados en valores bajos para variables como *repeated_guests, no_of_previous_cancelations, no_of_previous_bookings_not_canceled, o required_car_parking_space*. Para otras como *lead_time* o *avg_price_per_room*, los valores están más repartidos.

Figura 6. Distribuciones de variables numéricas.



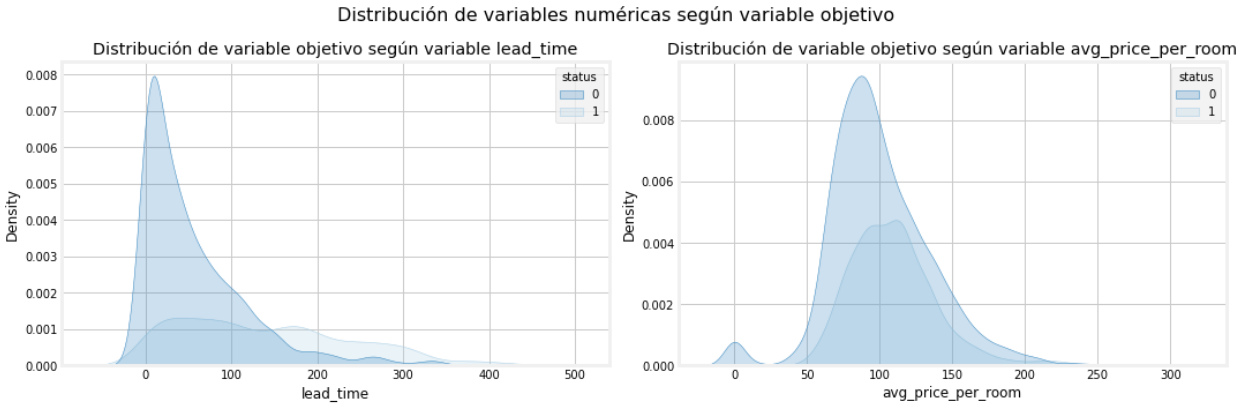
Podemos visualizar también su relación lineal con una matriz de correlación. Algunas relaciones que se destacan son las de la variable objetivo con *lead_time* (0.45), cual enseña la diferencia entre la fecha de reserva y llegada, o la de *repeated_guests* con *no_of_previous_booking_canceled* (0.56), lo cual tiene sentido dado que para poder haber cancelado previamente una reserva, hay que haber sido un huésped en algún momento.

Figura 7. Matriz de correlación.



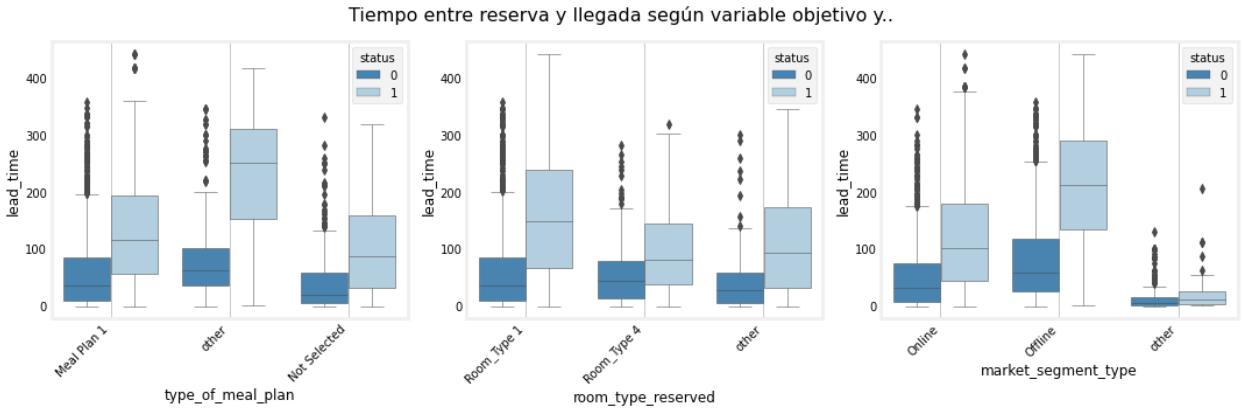
Así mismo, se observa la distribución entre algunas variables numéricas que tienen potencial importancia sobre la variable objetivo, cruzada por esta última. Como enseñan los gráficos debajo, las cancelaciones suelen ser en reservas creadas con mayor anticipación que las no cancelaciones, y el precio no es un factor determinante.

Figura 8. Distribución de variables numéricas según variable objetivo.



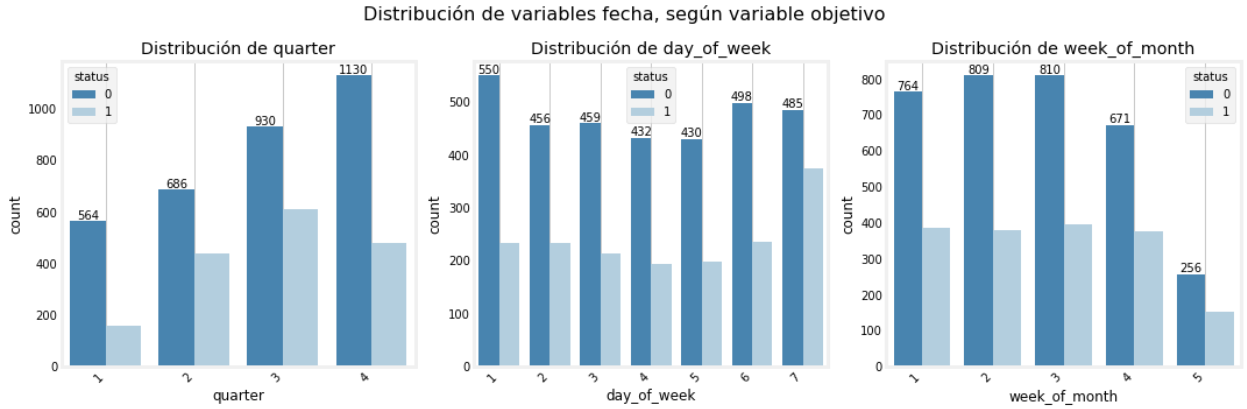
Indagando en la relación entre el tiempo entre llegada y fecha de reserva con la variable objetivo, podemos cruzarla por las variables categóricas para ver si hay alguna tendencia que no se vea a simple vista. La Figura 9 nos deja notar que, para todos los grupos de categorías, sin importar la variable, el tiempo entre reserva-llegada es mayor para los casos donde se ha cancelado una reserva. En algunos casos la diferencia es más clara, como por ejemplo en las reservas *offline*.

Figura 9. Lead time según variable objetivo y variables categóricas.



Como último de análisis exploratorio, podemos aprovechar la existencia de las variables año, mes y día, para poder generar una variable que enseñe la fecha de llegada y, a partir de ella, calcular otras variables como el trimestre, semana del mes, día del año, día de la semana, etc. Cabe notar que en este caso se han eliminado pocos casos en los que la fecha de llegada era 29 de febrero, por términos de facilidad de manejo de los datos.

Figura 10. Distribución de variables creadas a partir de la fecha, según variable objetivo.

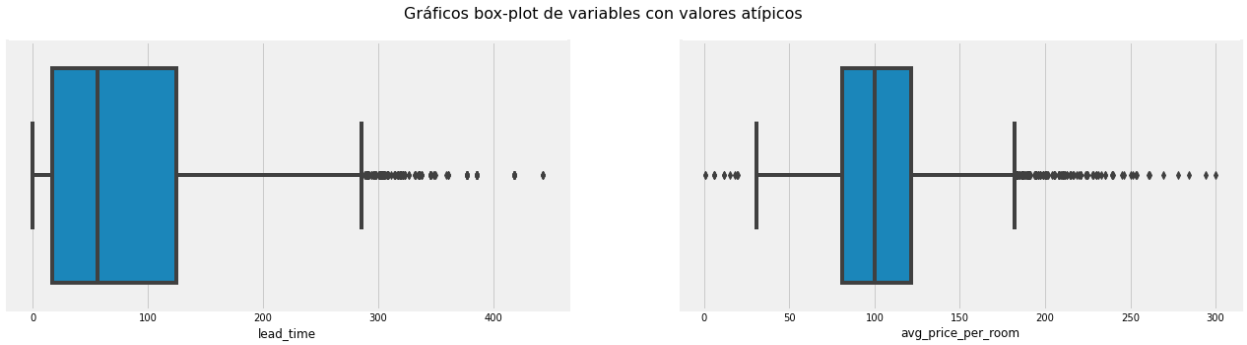


Los *quarters* 2 y 3 tienen mayor proporción de reservas canceladas que los 1 y 4, como también las reservas donde el día de llegada cae domingo.

Tratamiento de valores atípicos y preparación de datos para modelos

Como se ha comentado previamente, el conjunto de datos cuenta con variables numéricas como por ejemplo el tiempo entre fecha en que se realizó la reserva y la fecha de llegada al hotel, o el precio promedio por habitación, entre otras. Además de estas dos, se han seleccionado al número de estadías previas (canceladas o no) como variable a analizar, debido a sus altos valores máximos vistos a simple vista. Los valores mínimos y máximos del resto de las variables, entre otras binarias, fueron considerados lógicos y no entraron en un análisis de valores atípicos. Por lo tanto, el foco fue sobre *lead time y avg_price_per_room* (Ver Figura 11).

Figura 11. Box-plots de variables con valores atípicos.



Como primer filtro, se decidió filtrar aquellas reservas con precio nulo o negativo y las observaciones con *lead_time* igual a 0 (204 casos). Luego, utilizando la técnica de Tuckey, cual tiene en cuenta los rangos intercuartílicos de cada variable, filtramos valores que estén por debajo del valor de Q1-1.5*IQR o por encima de Q3+1.5*IQR (Ver Código 1), siendo IQR la diferencia entre el tercer y el primer cuartil.

Código 1. Detección de outliers a partir del método Tukey.

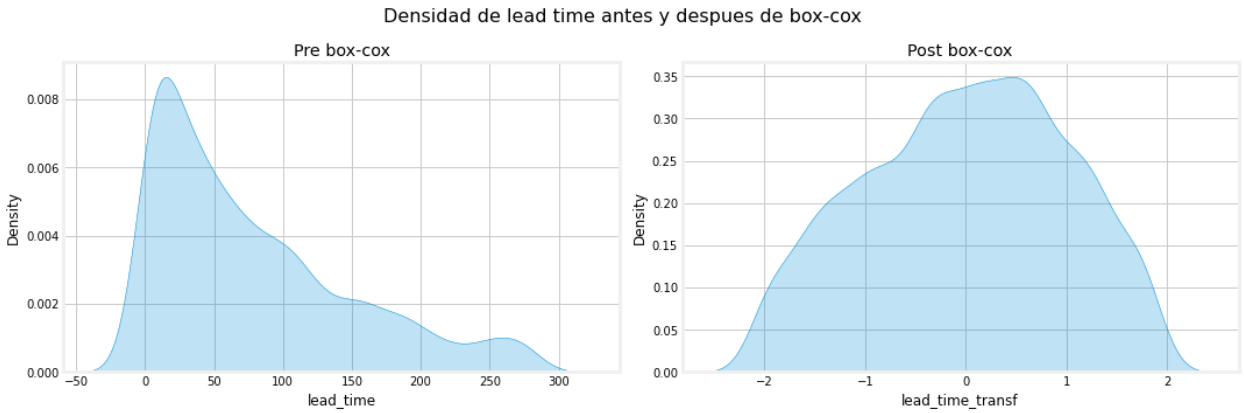
```
def tukey_outliers(df,column,extreme=False):
    q1, q3 = np.percentile(df[column],[25,75])
    iqr = q3 - q1
    constant = 1.5 if not extreme else 3
return df[~((df[column]>(q3+constant*iqr)) | (df[column]<(q1-
               constant*iqr)))]
```

Un 3,6% de casos del total se eliminarían como *outliers* de *lead_time,* y un 2,9% por el precio promedio de las habitaciones. Las variables pasarían a tener el aspecto enseñado en la Figura 12.

Figura 12. Box-plots de variables sin valores atípicos.

Más ajustes pueden realizarse a la base de datos. Por ejemplo, aunque la variable *avg_price_per_room* presente valores cerca de una densidad normal, esta fue escalada entre 0 y 1 utilizando la función **MinMaxScaler** de **scikit-learn**. Como también se puede notar, la variable *lead time* es asimétrica hacia la izquierda, por lo que podemos analizar que transformación **box-cox** puede servirnos para normalizarla lo mayor posible. Utilizando la herramienta **PowerTransformer** de **scikit-learn** podemos notar que la variable logra normalizarse en mayor medida. Ahora bien, cabe notar que esta transformación se realizó entrenando el transformador en el set de entrenamiento, y aplicandolo tanto a este mismo como al de testeo. Esto se hizo para evitar el filtrado de información entre ambos sets de datos.

Figura 13. Densidades antes y después de transformaciones box-cox.



El paso siguiente para realizar para preparar el conjunto de datos para modelados, es el de convertir las variables categóricas a numéricas. Recordar que contamos con variables como *type_of_meal_plan, room_type_reserved, market_segment_type*. Para esto utilizaremos la función de **Pandas** *get_dummies,* eliminando la primera categoría como columna.

Luego de eliminar casos atípicos en cada conjunto, emplear las transformaciones a *lead time, avg_price_per_room,* pasar a numéricas las variables categóricas y borrar columnas que ya quedaron redundantes, se obtuvieron los siguientes sets de datos enseñados en la Tabla 2.

Tabla 2. Conjuntos de entrenamiento y testeo.

Set de datos	Cantidad de observaciones	Cantidad de variables
Total	4.499	32
Entrenamiento	3.599	
Test	900	

La división fue generada utilizando **StratifiedShuffleSplit** separando un 20% para el set de testeo**,** cual tiene en cuenta la variable objetivo a la hora de realizar esta misma. De esta manera, nos aseguramos de que la distribución del indicador de la reserva se mantenga en ambos conjuntos de datos.

Selección de variables

Dado que contamos con 32 variables finales luego de efectuar transformaciones e ingeniería de variables, podemos evaluar cuales de estas son consideradas lo suficientemente importantes para ser utilizados a la hora de generar modelos predictivos.

La librería **scikit-learn** nos presenta muchas maneras de realizar esto. Una de ellas es con la función del módulo **feature_selection** llamada **SelectKBest.** Esta genera un ranking de puntajes de cada variable de entrada, utilizando como puntaje al método que se le pase. En el caso de regresiones de clasificación, se le debe pasar la función *f_classif*, que, por detrás, genera un puntaje a partir de un ANOVA. Luego, el usuario le indica el número de variables finales a obtener (parámetro *k*), y la función devuelve los nombres de estas (ver Código 2).

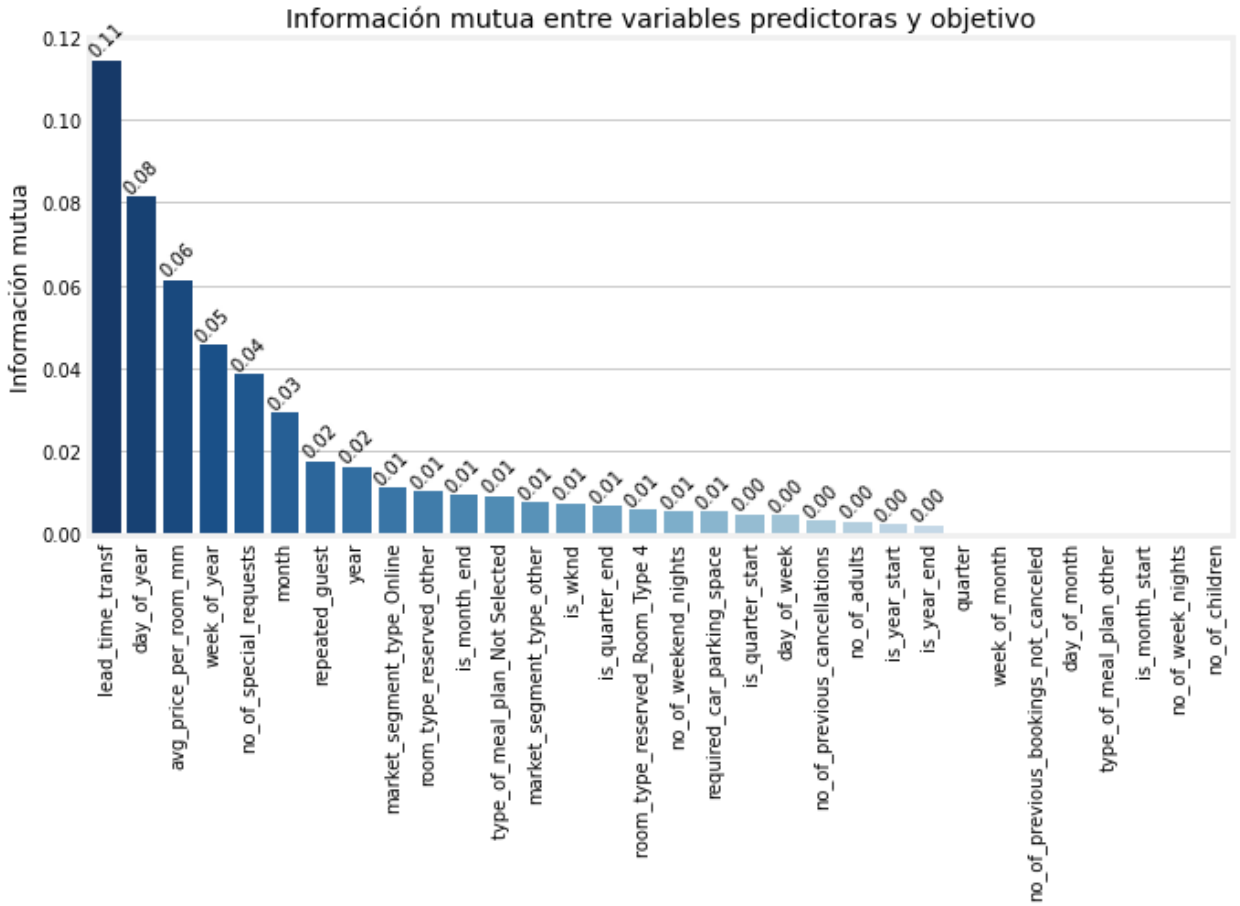
Código 2. SelectKBest para selección de variables.

```
from sklearn.feature_selection import SelectKBest, f_classif

kbest = SelectKBest(f_classif, k=15)
X_new = kbest.fit_transform(X, y)
variables_seleccionadas = [nombre for i, nombre
                           in enumerate(X.columns)
                           if kbest.get_support()[i]]
```

En este caso, se le ha indicado la selección de 15 variables, para lograr reducir a la mitad el número de variables de entrada a los modelos.

Este método de selección de variables puede ser convalidado con uno diferente, también ofrecido por la misma librería. A partir de la función **mutual_info_classif** podemos adentrarnos en el método de selección de variables de información mutua. Este es útil cuando se quiere evaluar la dependencia no lineal entre las características y la variable objetivo, ya que no asume una relación lineal como algunas otras métricas de puntuación. Cuanto mayor sea la información mutua entre una característica y la variable objetivo, más relevante se considerará la característica. Para calcular la información mutua entre una característica y la variable objetivo, se utilizan las distribuciones de probabilidad conjunta e individual de ambas variables.

Figura 14. Selección de variables con información mutua.



La Figura 14 nos enseña las variables explicativas con sus puntajes de información mutua, de las cuales se filtran aquellas con valores mayores a 0. Esto, combinándolo a las que entrega el método SelectKBest, nos entregaría un total de 12 variables. Estas son *'avg_price_per_room_mm', 'day_of_week', 'is_wknd', 'lead_time_transf', 'market_segment_type_Online', 'market_segment_type_other', 'no_of_special_requests', 'no_of_weekend_nights', 'repeated_guest', 'required_car_parking_space', 'type_of_meal_plan_Not_Selected',* e *'year'.*

Regresión logística

Antes que crear los modelos de predicción, hay que determinar la métrica a utilizar para evaluarlos. En nuestro caso, como se comentó previamente, contamos con un leve desbalanceo de la variable objetivo (dos tercios – un tercio). Por este motivo, no utilizaremos a la *accuracy* como medida de calidad, ya que estaríamos subestimando la importancia de predecir los casos positivos. En cambio, utilizaremos a la medida AUC (*Area Under Curve)*, el área bajo la curva ROC, cual tiene en cuenta la sensibilidad y especificidad de los modelos. Así mismo, prestaremos atención al *recall*, especialmente de la clase positiva.

La **regresión logística** es un modelo estadístico utilizado para predecir la probabilidad de ocurrencia de un evento binario (por ejemplo, sí/no) en función de uno o más predictores. Utiliza una función sigmoidea para transformar la combinación lineal de los predictores en una probabilidad entre 0 y 1. El modelo se ajusta a los datos utilizando el método de máxima verosimilitud y se utiliza para hacer predicciones sobre la probabilidad de ocurrencia del evento para nuevas observaciones.

Utilizaremos un método de validación cruzada de la librería ***scikit-learn***, *KFold*, con un parámetro de *shuffle = True.* Este indica que, tras cada separación, mezcla de forma aleatoria el set de entrenamiento. Sumando este método a la función *GridSearchCV* para validación de cruzada, podemos encontrar la combinación óptima de hiper-parámetros. Cabe notar que la validación cruzada fue creada a partir del set de entrenamiento, para evitar la presencia de *leakage.*

En este caso, los parámetros escogidos a optimizar son:

- **Penalty**: Es una penalización que se aplica en función de la cantidad de variables incluidas en el modelo. Esta penalización ayuda a evitar que el modelo se sobreajuste a los datos de entrenamiento y mejora su capacidad para generalizar a nuevos datos.
- **C**: Es la inversa de la fuerza de regularización y mide el nivel de ajuste del modelo. Un valor menor de C indica una regularización más fuerte, lo que significa que el modelo tendrá una tendencia a subestimar los coeficientes de los predictores. Por otro lado, un valor mayor de C indica una regularización más débil, lo que significa que el modelo tendrá una tendencia a sobreestimar los coeficientes.
- **Solver**: Es el algoritmo que se utiliza para optimizar los coeficientes del modelo.

La Tabla 3 enseña las opciones escogidas para cada parámetro, y en azul, aquellas opciones que optimizaron el AUC. El mejor modelo obtuvo un puntaje de validación de cruzada de 0.852. Como la gran mayoría de las combinaciones de parámetros entregaban métricas muy cercanas, se omite la presentación de todos los resultados. Sin embargo, si mostraremos los resultados según los hiper-parámetros para los modelos siguientes a analizar.

Tabla 3. Hiper-parámetros a escoger en el modelo de regresión lineal.

Penalty	L1	L2				
Solver	liblinear	newton-cg				
C	0.001	0.01	0.1	1	10	100

El Código 3 enseña el proceso que sigue la validación cruzada y la optimización de parámetros. El mejor modelo que sale de este código también nos permite también evaluar los resultados en el conjunto de testeo. Dicho análisis será realizado en la sección final, donde se escogerá el mejor modelo teniendo en cuenta tanto los resultados de validación cruzada como de testeo (nuevos datos).

Código 3. Creación del modelo de regresión logística.

```
params_lr = {
    "penalty": ["l1","l2"],
    "C": [0.001,0.01,0.1,1,10,100],
    "solver": ["newton-cg", "liblinear"]
}

kfold = KFold(n_splits=5,shuffle=True,random_state=seed)

grid_search_lr = GridSearchCV(LogisticRegression(random_state=seed),
                              param_grid=params_lr,
                               scoring="roc_auc",
                               cv= kfold,
                               verbose=-1)

grid_search_lr.fit(X_train, y_train)
```

Árbol de decisión

Los **árboles de decisión** son un modelo de aprendizaje supervisado utilizado para la clasificación y regresión. En este modelo, se construye un árbol a partir de los datos de entrenamiento, donde cada nodo representa una pregunta sobre una de las características de los datos. El árbol se divide en ramas basándose en la respuesta a cada pregunta, y cada hoja representa una clase o valor de regresión. El objetivo es dividir los datos en ramas de manera que se maximice la homogeneidad dentro de cada rama y se minimice la heterogeneidad entre las ramas.

Para encontrar el mejor modelo de árboles de decisión, realizaremos la optimización de los siguientes parámetros:

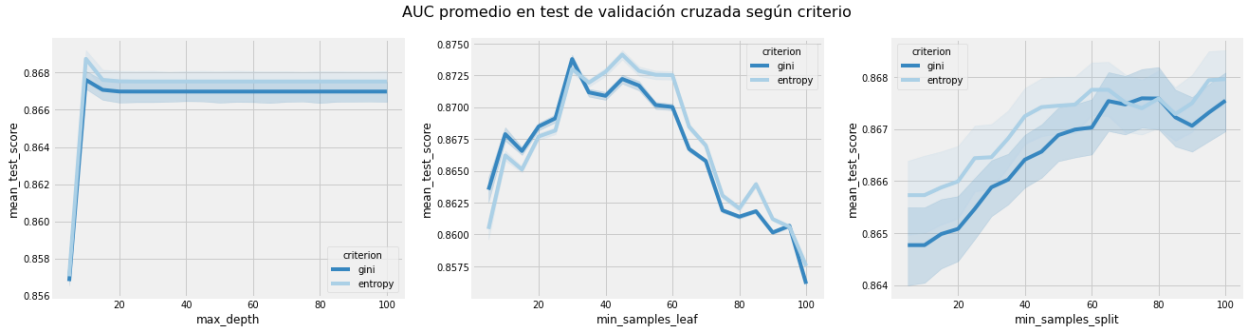
- **Max_depth**: máxima profundidad del árbol. Mientras más alto sea este valor, más sobreajustará el modelo, pues intentará llegar a una complejidad extrema.
- **Criterion**: método de partición (se escogen Gini y entropía)
- **Min_sample_leaf**: número mínimo de muestras por nodo
- **Min_sample_split**: número mínimo de muestras requeridas para realizar una división en un nodo interno

Se tuvieron en cuenta valores de 0 al 100, de cinco en cinco, para los parámetros *min_samples_split, min_samples_leaf* y *max_depth.*

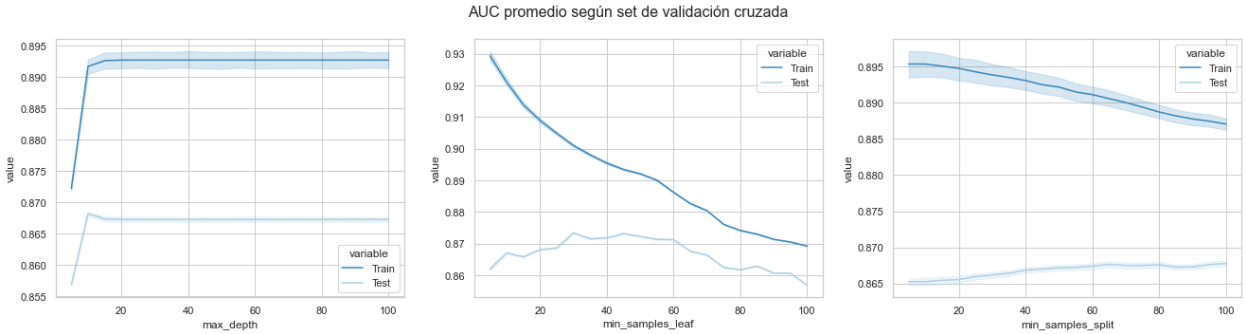
De esta manera, el mejor modelo encontrado en validación cruzada otorga un AUC de 0.876, más alto que el encontrado en la regresión logística. Cabe notar que tanto para los criterios de Gini de entropía, se encuentra un número óptimo de muestras por nodo de 10 unidades, y de 65 unidades para ser suficiente para un nodo divisivo. La máxima profundidad optimizada es de 10 (Ver Figura 15.a). Luego de este valor, se mantiene estable.

Figura 15.

a) AUC en test en validación cruzada según hiper-parametros – Decision Tree



b) AUC según set de validación cruzada – Decision Tree.



El mejor modelo encontrado cuenta con las características descriptas en el Código 4. Este será utilizado para el bagging en la próxima sección.

Codigo 4. Árbol de decisión.

```
decision_tree = DecisionTreeClassifier(max_depth=10,
                                       min_samples_leaf=10,
                                       min_samples_split=65,
                                       criterion='gini')
```

Bagging

Los dos ingredientes clave de **Bagging** son bootstrap y combinación. Por lo general, esto implica el uso de un solo algoritmo de aprendizaje automático, casi siempre un árbol de decisiones, el cual es entrenado

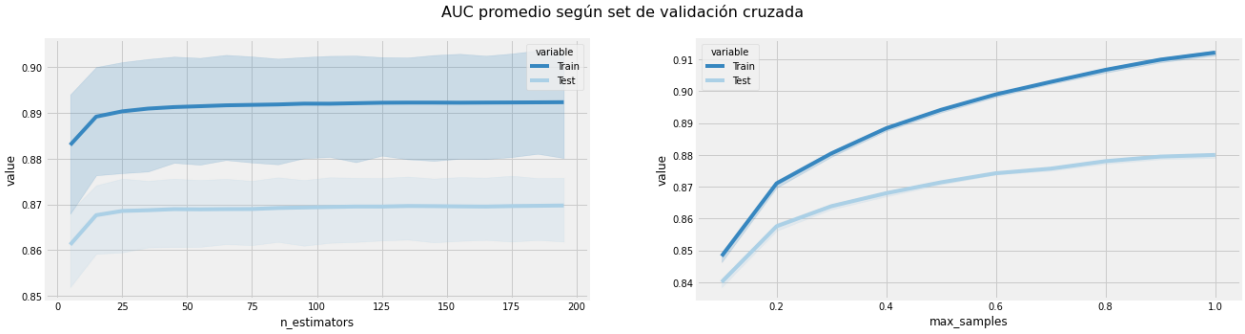
utilizando diferentes conjuntos de datos de entrenamiento. Así se generan N árboles y las predicciones que realizan cada uno de los miembros del conjunto se combinan usando alguna estadística simple: votaciones, promedios, promedios ponderados, etc. Cada modelo obtiene su propio conjunto de datos utilizando el método de bootstrap. Esto quiere decir que una fila dada puede estar presente en un conjunto de datos cero, una o múltiples veces.

Como modelo base para **Bagging** utilizaremos el DecisionTree previamente optimizado enseñado en el Código 4. Además, se optimizarán los nuevos parámetros del BaggingClassifier:

- **max_samples**: el porcentaje de muestras a extraer para entrenar el estimador base
- **n_estimators**: número de estimadores base a crear en el ensamblado

Se han escogido valores de 5 a 200 para el número de estimadores, de cinco en cinco, y de 0.1 a 1 para *max_samples*.

Figura 16. AUC según set de validación cruzada - Bagging.



Como enseña la Figura 16, mientras mayor es el porcentaje de la muestra utilizada para el entrenamiento, mayor es el resultado de AUC tanto para el set de entrenamiento como de testeo en la validación cruzada. El valor optimo encontrado por la validación cruzada es de 1, aunque cabe notar que luego de 0.2 la diferencia entre entrenamiento y testeo se agranda. Algo parecido sucede con el número de estimadores. Aunque luego de aproximadamente 20 estimadores el valor se mantiene casi constante, el valor de este hiper parámetro que optimiza el AUC es de 185 estimadores, llegando a 0.881.

Random Forest

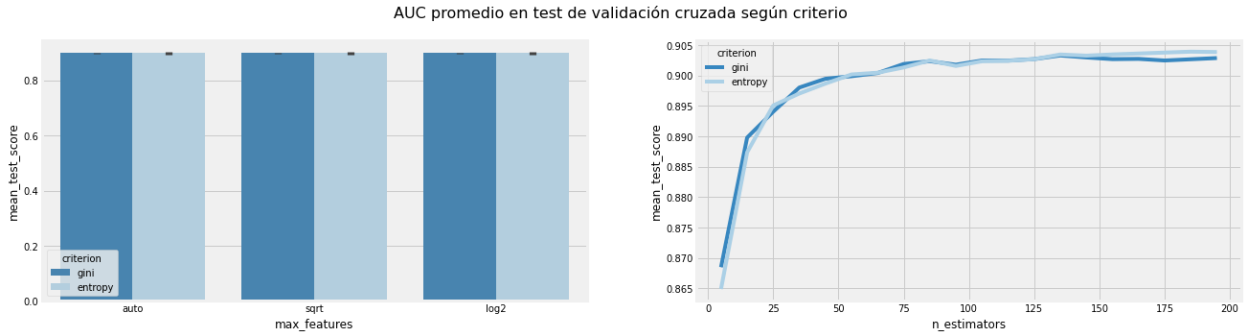
Por otro lado, **Random Forest** es una modificación del Bagging que consiste en incorporar aleatoriedad en las variables utilizadas para segmentar cada nodo del árbol. Para encontrar el mejor modelo de Random Forest, optimizaremos los parámetros *n_estimators, max_features* y *criterion*, con los valores observados en la grilla de parámetros del Código 5.

Código 5. Validación cruzada para Random Forest

```
param_grid = {
    "n_estimators": [x for x in range(5,201,10)],
    "max_features": ["auto","sqrt","log2"],
    "criterion":["gini","entropy"]
}

kfold = KFold(n_splits=5,shuffle=True,random_state=seed)

grid_search_rf = GridSearchCV(RandomForestClassifier(random_state = seed)
                              ,param_grid=param_grid
                              ,scoring="roc_auc"
                              ,cv=kfold
                              ,n_jobs=-1
                              ,verbose=0
                              ,return_train_score=True)

grid_search_rf.fit(X_train, y_train)
```

La metodología utilizada para escoger la máxima cantidad de variables en el modelo no parece ser crucial, al no mostrar diferencias según sus distintos tipos (al ser 12 variables, entregan el mismo número de variables finales) y según el método de partición. Sin embargo, el número de estimadores parece crecer a lo largo de su cantidad, y estabilizarse alrededor de 150 (ver Figura 17).

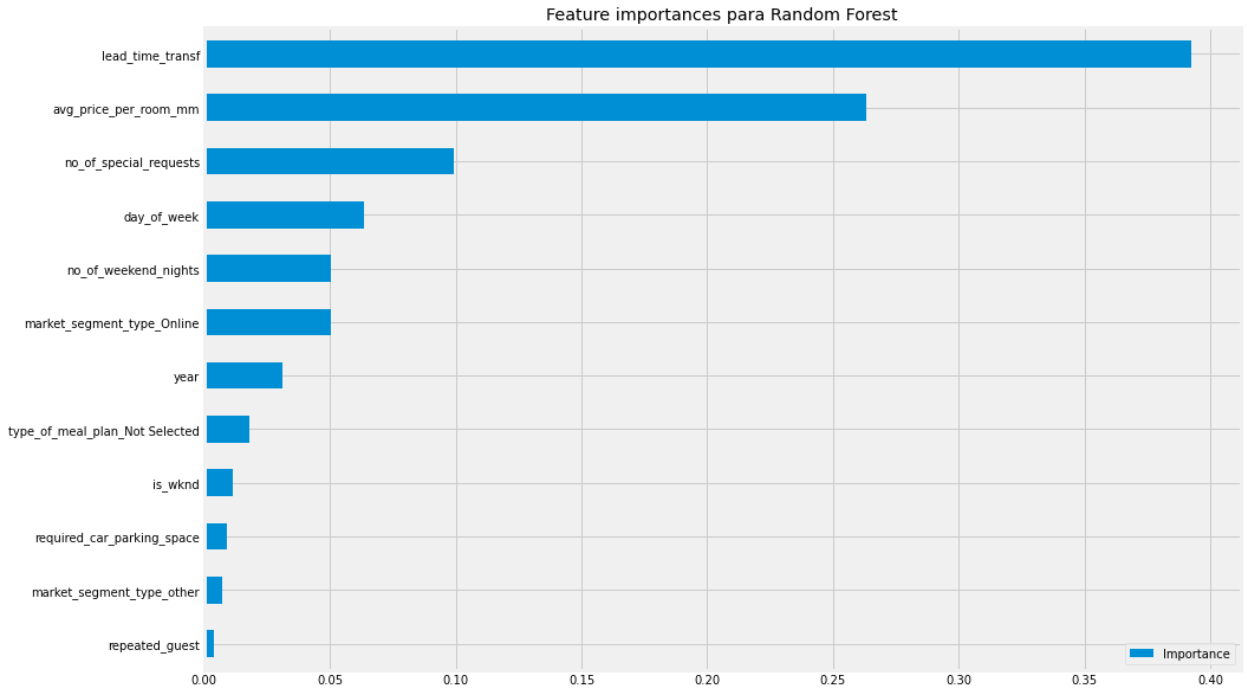
Figura 17. AUC promedio en test de validación cruzada según criterio – Random Forest



Con un criterio de entropía, 185 estimadores y seleccionado la máxima cantidad de *features* automáticamente, el modelo de Random Forest entrega un AUC de validación cruzada de 0.904, el más alto hasta ahora. Sin embargo, el modelo tiene un valor de AUC en entrenamiento de validación cruzada que puede estar indicando sobreajuste. Esto será evaluado en las conclusiones.

Además, nos permite visualizar las importancias de variables. Este es un método de prueba y error que consiste en evaluar cómo se degrada la precisión del modelo al permutar aleatoriamente los valores de una variable. Si la precisión del modelo disminuye al permutar los valores de una variable, se considera que esa variable es importante para el modelo. Véase como lead_time es la variable con más peso para el modelo, seguido por el precio promedio y número de pedidos especiales.

Figura 18. Importancia de variables – Random Forest



Gradient Boosting

Boosting es un algoritmo de aprendizaje automático que ayuda a mejorar las performances de los modelos. Como bagging, utiliza el sistema de votación para clasificaciones, combinando distintos modelos del mismo tipo. Mientras que en bagging cada modelo es construido por separado, en boosting estos modelos son preparados teniendo en cuenta la performance de sus pares.

Así como la técnica les aplica una ponderación a los modelos por su buena performance, en el mismo sentido los penaliza por su error de clasificación. El algoritmo **Gradient Boosting** consiste en repetir la construcción de árboles de clasificación, modificando ligeramente las predicciones iniciales cada vez, intentando ir minimizando los residuos en la dirección de decrecimiento, dada por el negativo del gradiente de la función de error. A diferencia de los anteriores, este modelo no calcula árboles y después los agrega, sino que, para cada árbol construido, ve que errores comete y lo mejora, otorgando siempre el mejor al final.

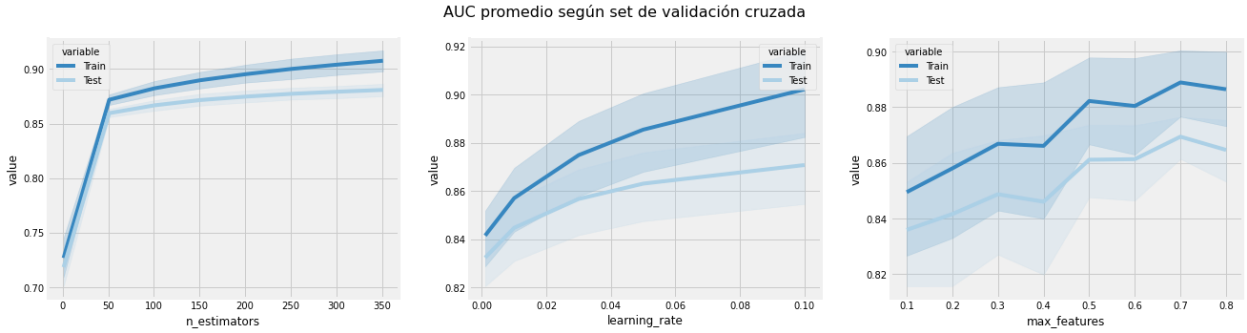
Optimizaremos los parámetros *n_estimators, learning_rate* y *max_features.* Optimizando el ratio de aprendizaje, podemos controlar cómo de rápido aprende el modelo. Y agregando un número máximo de variables para los modelos, podemos controlar el sobreajuste.

Código 6. Grilla de parámetros para Gradient Boosting.

```
param_grid = {
    "n_estimators": [x for x in range(1,401,50)],
    "learning_rate": [0.1, 0.05, 0.03, 0.01, 0.001],
    "max_features": list(np.arange(0.1,0.80,0.1))
}
```

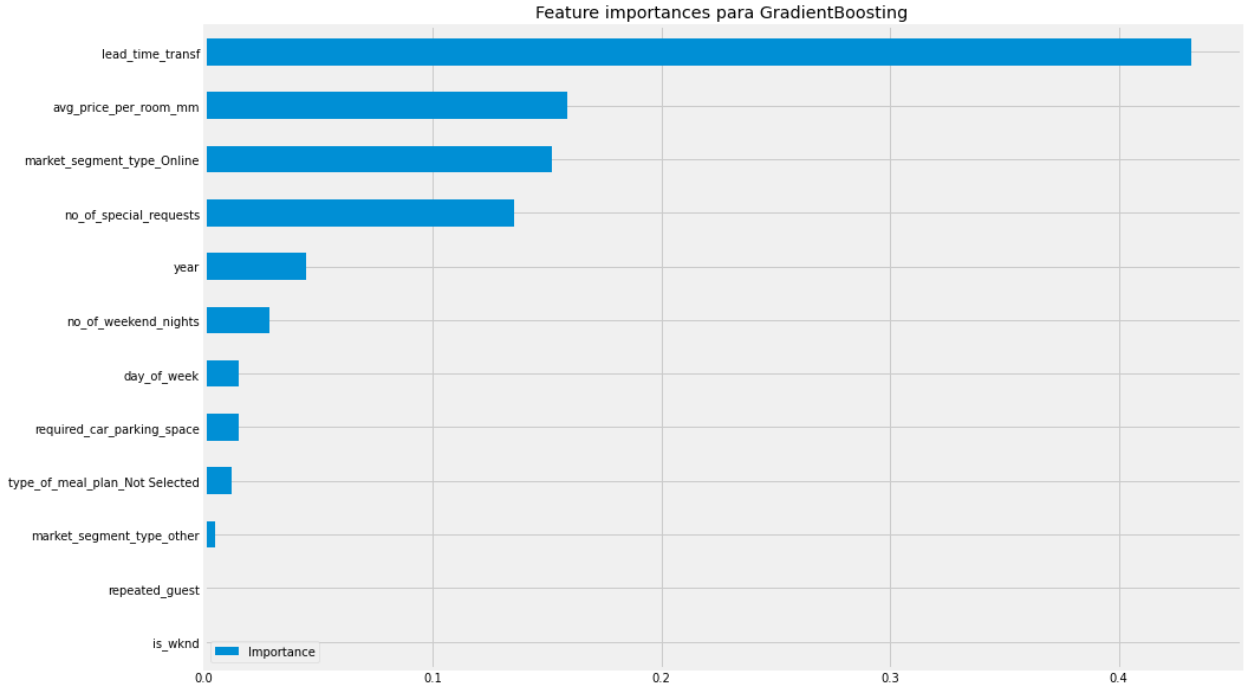
La performance del modelo de GradientBoosting entrega un AUC de 0.904, al igual que el Random Forest, utilizando una *learning rate* de 0.1, un *max_features* de 0.7 y 350 estimadores. Podemos notar el comportamiento del AUC según los distintos valores de estos parámetros.

Figura 19. AUC según set de validación cruzada – Gradient Boosting



Luego de una *learning rate* de 0.1, notamos como se comienza a desprender el AUC en entrenamiento versus testeo. Algo parecido sucede con el número de estimadores superando los 150. También notamos que la variable de segmento Online incrementa su importancia relativa al Random Forest.

Figura 20. Importancia de variables – Gradient Boosting



XGBoost

El modelo **XGBoost** (Extreme Gradient Boosting) es una evolución del anterior, utilizando el concepto de *regularización*. Este concepto sirve para evitar el sobreajuste dentro de la optimización interna del modelo en sí. Una de las técnicas más comunes para la regularización en modelos de aprendizaje automático son las penalizaciones L1 y L2. La penalización L1 (Alpha) disminuye la influencia de las variables menos importantes al forzar algunos coeficientes a ser igual a cero. En contraste, la penalización L2 (lambda), también llamada regularización de Ridge, reduce la influencia de las variables más destacadas al penalizar los valores grandes de los coeficientes.

Cuando se trata de XGBoost y árboles en general, estas técnicas de regularización afectarán el tamaño e importancia de las hojas finales en cada árbol. A continuación, evaluaremos el impacto de ambas formas de regularización. El Código 7 enseña el proceso de validación cruzada para el modelo, obteniendo un AUC de 0.902.

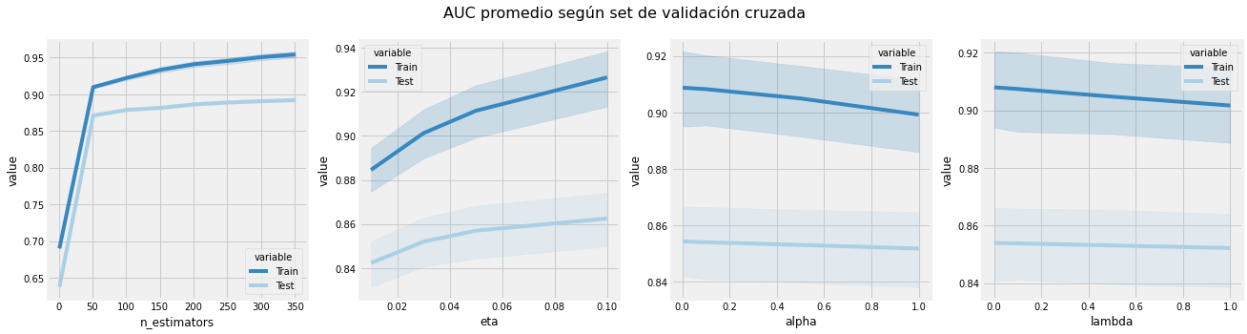
Código 7. Validación cruzada para XGBoost

```
param_grid = {
    "n_estimators": [x for x in range(1,401,50)],
    "alpha": [0.001, 0.01, 0.1,0.5, 1],
    "lambda": [0.001, 0.01, 0.1,0.5, 1],
    "eta": [0.1, 0.05, 0.03, 0.01]
}

kfold = KFold(n_splits=5,shuffle=True,random_state=seed)

grid_search_xgb = GridSearchCV(XGBClassifier(random_state = seed
                                             ,colsample_bytree = 0.3
                                             )
                               ,param_grid=param_grid
                               ,scoring="roc_auc"
                               ,cv=kfold
                               ,n_jobs=-1
                               ,verbose=0
                               ,return_train_score=True)

grid_search_xgb.fit(X_train, y_train)
```

Notar que se ha marcado un número máximo de *colsample_bytree* de 0.3 (4 variables), para evitar sobreajustes (esta actúa como el *max_features* en RandomForest). Al evaluar los resultados en términos de AUC para cada parámetro, observamos que nuevamente la *learning rate* (*eta* en XGB) es de 0.01 para encontrar el mejor resultado. La Figura 21 también nos enseña los resultados de Alpha y lambda, es decir, las regularizaciones. Ambas muestran una tendencia uniforme, con leve decrecimiento al crecer sus valores. Por último, el resultado del modelo mejora al crecer también el número de estimadores, como venía sucediendo también con los modelos anteriores.

A diferencia de los otros modelos, *lead_time* reduce su importancia, siendo la clave en XGB el segmento Online.

Figura 21. AUC según set de validación cruzada y parámetros - XGBoost



Redes neuronales

En resumen, el modelo de redes neuronales utiliza capas de neuronas artificiales para aprender patrones y relaciones en los datos. En un modelo de red neuronal, las entradas se envían a través de una o varias capas de neuronas, donde se aplican pesos y sesgos a las entradas y se produce una salida. Luego, la salida se pasa a través de una función de activación para producir la salida final. Durante el entrenamiento, el modelo ajusta los pesos y sesgos para minimizar la función de pérdida y mejorar el rendimiento del modelo en los datos de entrenamiento. A medida que se agregan más capas y neuronas a un modelo de red neuronal, se pueden capturar patrones y relaciones más complejos en los datos.

Sumado a las transformaciones generadas en la sección de EDA y preparación de datos, agregamos el escalado con MinMaxScaler() para las variables *no_of_weekend_nights* y *no_of_special_requests,*. Se ha notado que con estas transformaciones el modelo mejora su performance.

Utilizaremos el modelo **MLPClassifier** (Multi-layer Perceptron classifier) que nos entrega **scikit-learn** en su sub-modulo *neural networks.* Los parámetros de este modelo a optimizar son *hidden_layer_sizes*, *learning_rate_init* y *activation*. El primero nos dirá el efecto del número de neuronas en la capa oculta de la red. El segundo el ratio de aprendizaje, y la activación el tipo de transformación que se aplicará a los parámetros de las capas de la red. Con estos, generamos el modelo observado en el Código 8.

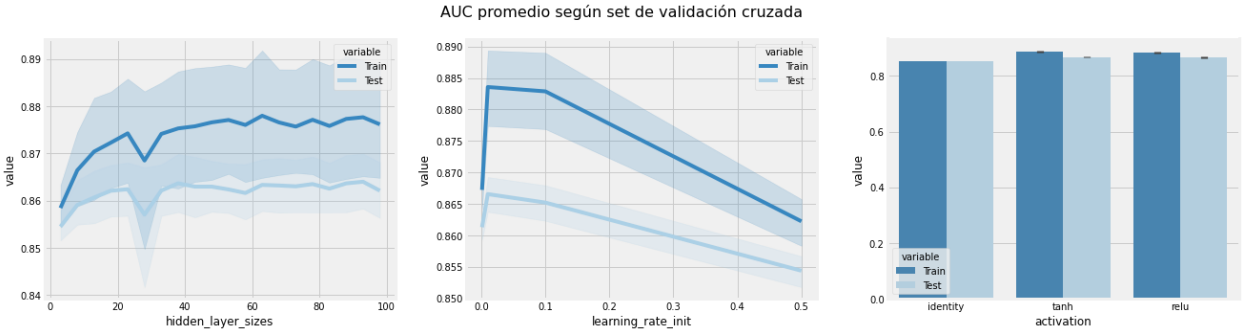
Código 8. Validación cruzada – Redes Neuronales

```
param_grid = {
    "hidden_layer_sizes": [(i,) for i in range(3,100,5)] ,
    "learning_rate_init": [0.001, 0.01, 0.1, 0.5],
    "activation": ["identity", "tanh", "relu"],
}


kfold = KFold(n_splits=5,shuffle=True,random_state=seed)

grid_search_nn = GridSearchCV(MLPClassifier(random_state = seed)
                              ,param_grid=param_grid
                              ,scoring="roc_auc"
                              ,cv=kfold
```

```
                                ,n_jobs=-1
                                ,verbose=0
                                ,return_train_score=True)

grid_search_nn.fit(X_train, y_train)
```

La validación cruzada otorga un AUC un poco más bajo que los que entregaron los modelos anteriores, de 0.881. El comportamiento de los hiper-parámetros nos muestra que el ratio de aprendizaje baja fuertemente luego de 0.1 (óptimo en 0.01), que las activaciones tanh y **relu** son levemente mejor que la identidad, y que el número de neuronas en la capa oculta cae en 30 y luego se mantiene constante hasta los 100 (óptimo en 78), como se puede ver en la Figura 22. Notar que el *solver* default que utiliza el clasificador es el de Adam. Este, como también otros parámetros, no fueron optimizados por términos de recursos computacional y objetivos del trabajo.

Figura 22. AUC según set de validación cruzada y parámetros – Redes Neuronales



Support Vector Machines

El modelo de máquinas de vector soporte utiliza métodos de entrenamiento supervisado para problemas de regresión o clasificación. Su funcionamiento se basa en el concepto de hiperplanos. Este busca encontrar un hiperplano óptimo que separe las muestras de diferentes clases en el espacio de características. El hiperplano óptimo es aquel que maximiza la distancia entre los vectores de soporte más cercanos de las diferentes clases. Estos son efectivos en casos de alta dimensionalidad. Sus principales conceptos son el de *maximal margin*, el encontrar un vector de parámetros que maximice el margen de separación de las clases, *soft margin*, una especie de permiso de fallar, y *kernel*, la transformación a aplicar a los datos no lineales para transformar los datos a un espacio de dimensión mayor donde sean separables linealmente. El modelo clasificará el nuevo dato según su ubicación en relación con el hiperplano encontrado durante el entrenamiento.

Para encontrar el mejor modelo SVM, iremos optimizando los parámetros para cada *kernel*. Como se comentó previamente, el *kernel* determina cómo se mapean los datos de entrada a un espacio de mayor dimensión para que se puedan separar las clases. Analizaremos los resultados para *kernels poly* (polinómico), lineal y *rbf* (radial).

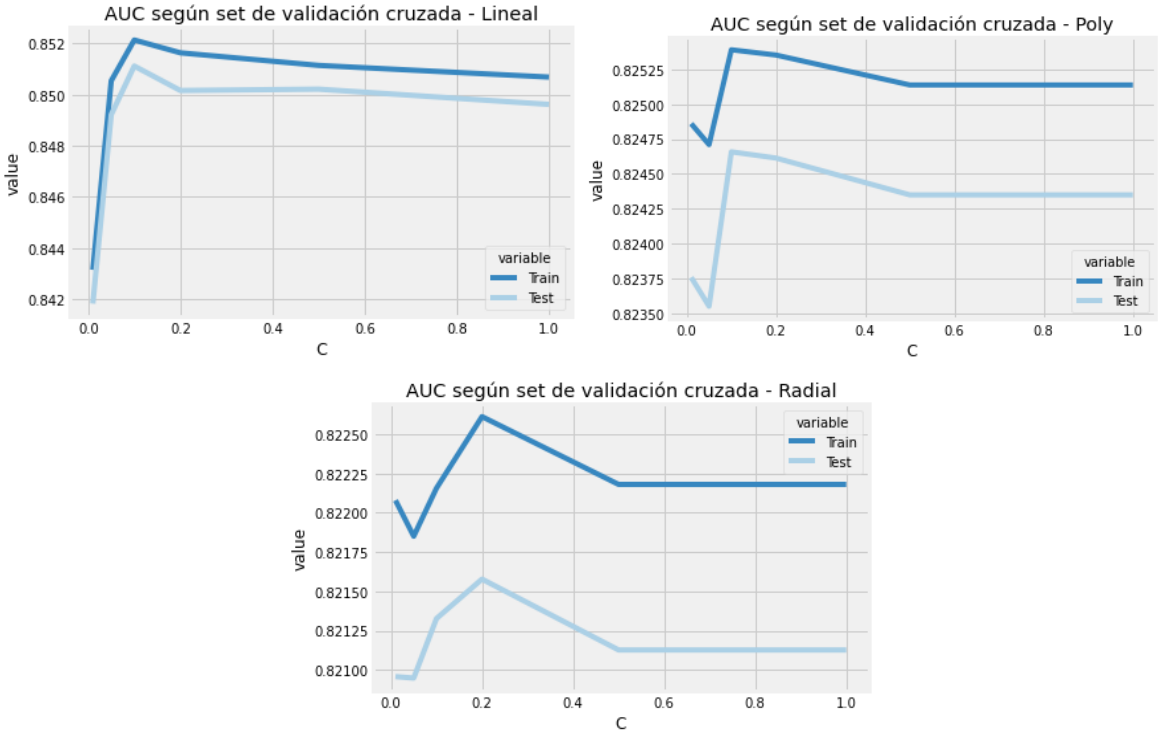
Comenzamos optimizando el valor del parámetro C para el *kernel* linear. Este parámetro controla la penalización por errores en la clasificación. Un valor alto de C significa que se penalizarán fuertemente los errores, lo que lleva a un modelo más complejo y ajustado a los datos de entrenamiento. Por otro lado, un valor bajo de C significa que se penalizarán menos los errores, lo que lleva a un modelo más simple y generalizado. Probamos valores de 0.01 a 1 (ver Figura 23). El valor óptimo para este parámetro resulta de 0.1, lo cual tiene sentido viendo la caída luego de este valor en el gráfico enseñado debajo. Esta combinación de *kernel* y C obtiene un AUC en validación cruzada de 0.851. Este análisis fue llevado a cabo con el Código 9.

Código 9. Optimización del parámetro C – SVM

```
param_grid = {
    "C": [0.01, 0.05, 0.1, 0.2, 0.5, 1]
}

kfold = KFold(n_splits=5, shuffle=True, random_state=seed)

grid_search_svm = GridSearchCV(SVC(random_state = seed
                                   , kernel='kernel'
                                   )
                               ,param_grid=param_grid
                               ,scoring="roc_auc"
                               ,cv=kfold
                               ,n_jobs=-1
                               ,verbose=0
                               ,return_train_score=True)

grid_search_svm.fit(X_train, y_train)
```

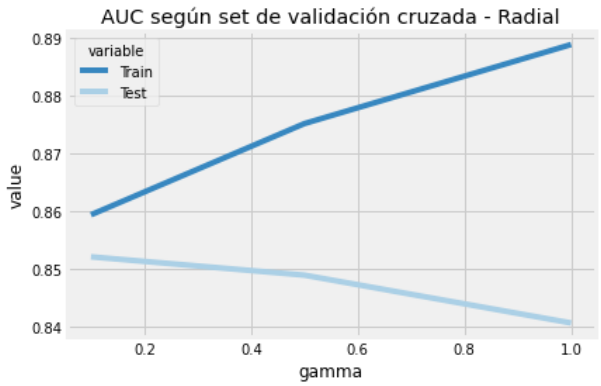
A la hora de optimizar un modelo de SVM con un *kernel poly (*transforma los datos en un espacio dimensional superior utilizando una función polinómica), el parámetro C también es de 0.1, aunque con un AUC menor (0.825). El *kernel* radial se utiliza comúnmente en SVM porque tiene la capacidad de mapear los datos a un espacio de características de alta dimensión, donde incluso datos no linealmente separables en el espacio original pueden volverse linealmente separables. Para este, el valor del coste aumenta a 0.2, con un AUC de 0.822 en test de validación cruzada promedio. Esto puede reflejarse en la distancia entre los puntajes de entramiento y testeo, ya que un mayor coste lleva a un mayor sobreajuste en muchos casos. Por otro lado, en el *kernel* lineal los resultados en entrenamiento y testeo son más cercanos que para el resto.

Figura 23. Optimización de parámetro C – SVM



Para el *kernel* radial introducimos también el parámetro gamma. Este parámetro controla la curvatura de la frontera de decisión. Un valor alto de gamma significa que el *kernel* tendrá un alcance más limitado, lo que puede llevar a un modelo más complejo y ajustado a los datos de entrenamiento. Por otro lado, un valor bajo de gamma significa que el *kernel* tendrá un alcance más amplio, lo que puede llevar a un modelo más simple y generalizado. Este parámetro solo fue optimizado para el *kernel* mencionado por límites de recursos computacionales. Los valores probados fueron de 0.1, 0.5 y 1, como se puede ver en la Figura 24. Incorporando el coste optimizado previamente, el AUC aumenta con un gamma de 0.1 a 0.852. El más alto de las iteraciones para todos los *kernels*, por lo que será utilizado para la comparación final de modelos. Sin embargo, notar como al aumentar este parámetro aumenta la diferencia entre los resultados en entrenamiento y testeo.

Figura 24. Optimización de parámetro gamma – SVM radial



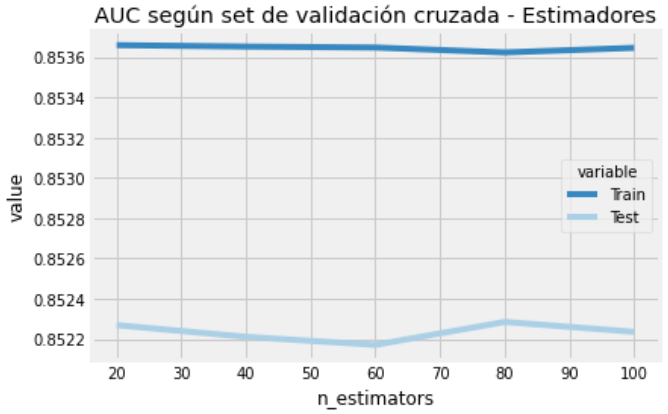
Bagging con regresión lineal

De la misma manera que realizamos el *bagging* utilizando un árbol de decisión, podemos hacerlo utilizando al modelo optimizado de regresión logística que se enseñó al principio de este ejercicio. Este usaba un método de penalidad L2, el *solver* newton-cg y un C de 10.

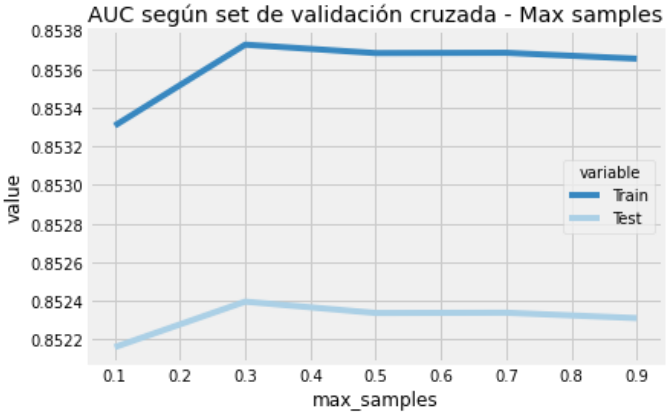
Los parámetros que optimizar entonces serán el porcentaje de muestras a extraer para entrenar el estimador base, *max samples,* y el número de estimadores. Para este último, utilizaremos valores más bajos que los probados con el resto de los modelos, simplemente por recursos computacionales. Como enseña la Figura 25a, se ha comenzado el análisis comprobando cual es el numero optimo de estimadores, yendo de 20 a 100.

Figura 25

a) Optimización del número de estimadores – Bagging con regresión logística



b) Optimización de max samples – Bagging con regresión logística



No parecen haber grandes diferencias en los resultados, pero notamos un pico en el testeo y una baja en el entrenamiento cuando se utilizan 80 estimadores. Si usamos este numero de estimadores para calcular el óptimo de muestras, vemos que probando de 0.1 a 0.9, luego de utilizar el 30% de la muestra los resultados se mantienen estables. El AUC que entrega esta combinación de hiper-parámetros optimizados

es de 0.852 en promedio para el set de testeo de validación cruzada. Esto es un poco menor que el obtenido con *bagging* con árboles de decisión.

Stacking

El método de stacking es una técnica de ensamblaje que consiste en combinar múltiples modelos de aprendizaje automático para obtener una mejor predicción. En este método, se utilizan varios modelos para hacer predicciones individuales, **modelos-base**, cuyas salidas se utilizan como entradas para un modelo de nivel superior, un **meta-modelo**, que produce la predicción final. Esto ayuda a reducir tanto la varianza como el sesgo y puede resultar en una mejora significativa en el acierto de las predicciones.

Para nuestro ejemplo, escogemos a una regresión logística como clasificador final, y utilizamos los previamente optimizados: Random Forest, redes neuronales y la misma regresión logística como modelos base (ver Código 10a).

Código 10.

a) Creación del modelo stacking

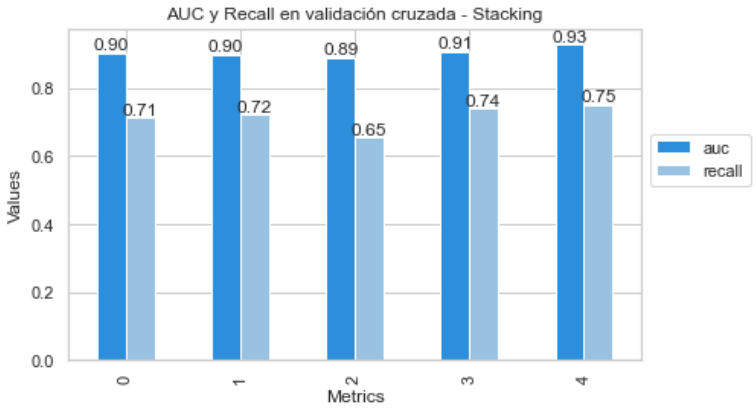
```
level0 = list()
level0.append(("reg_log", reg_log_model))
level0.append(("random_forest", rf_model))
level0.append(("nn", nn_model))

level1 = LogisticRegression(random_state=seed)

StackingClassifier(estimators=level0, final_estimator=level1)
```

b) Validación cruzada – Stacking

```
kfold = KFold(n_splits=5,shuffle=True,random_state=seed)

results = cross_validate(
    stacking,
    X_train,
    y_train,
    cv=kfold,
    scoring=["recall", "roc_auc"],
    return_train_score=False,
    n_jobs=-1
)

results_stacking = pd.DataFrame(results)
```

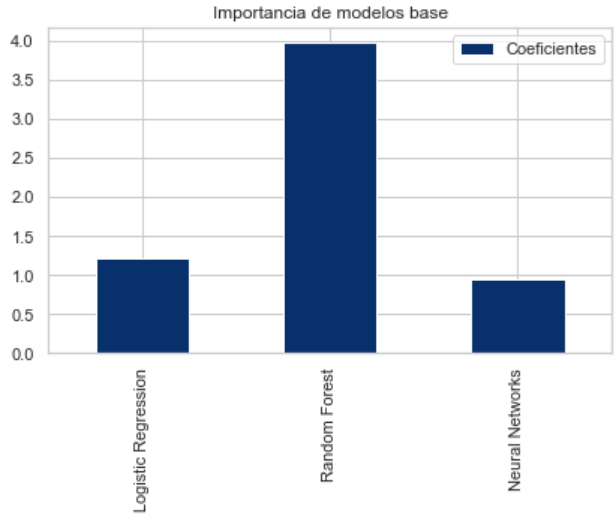
Los resultados del ensamble son positivos, obteniendo un AUC promedio de validación cruzada de 0.91, mostrando lo poderoso que puede ser un ensamble. La Figura 26 enseña los resultados tanto de AUC como de *recall* para los cincos sets de validación cruzada. Para obtener estos resultados, se ha utilizado el Código 10b, cual cuenta con la función *cross_validate*.

Figura 26. Resultados de validación cruzada - Stacking



Por último, podemos visualizar la importancia de cada estimador base, a partir de los coeficientes que entrega el clasificador stacking. Como se ve en la Figura 27, el modelo base Random Forest es el que más peso tiene de los tres.

Figura 27. Importancia de modelos base - Stacking

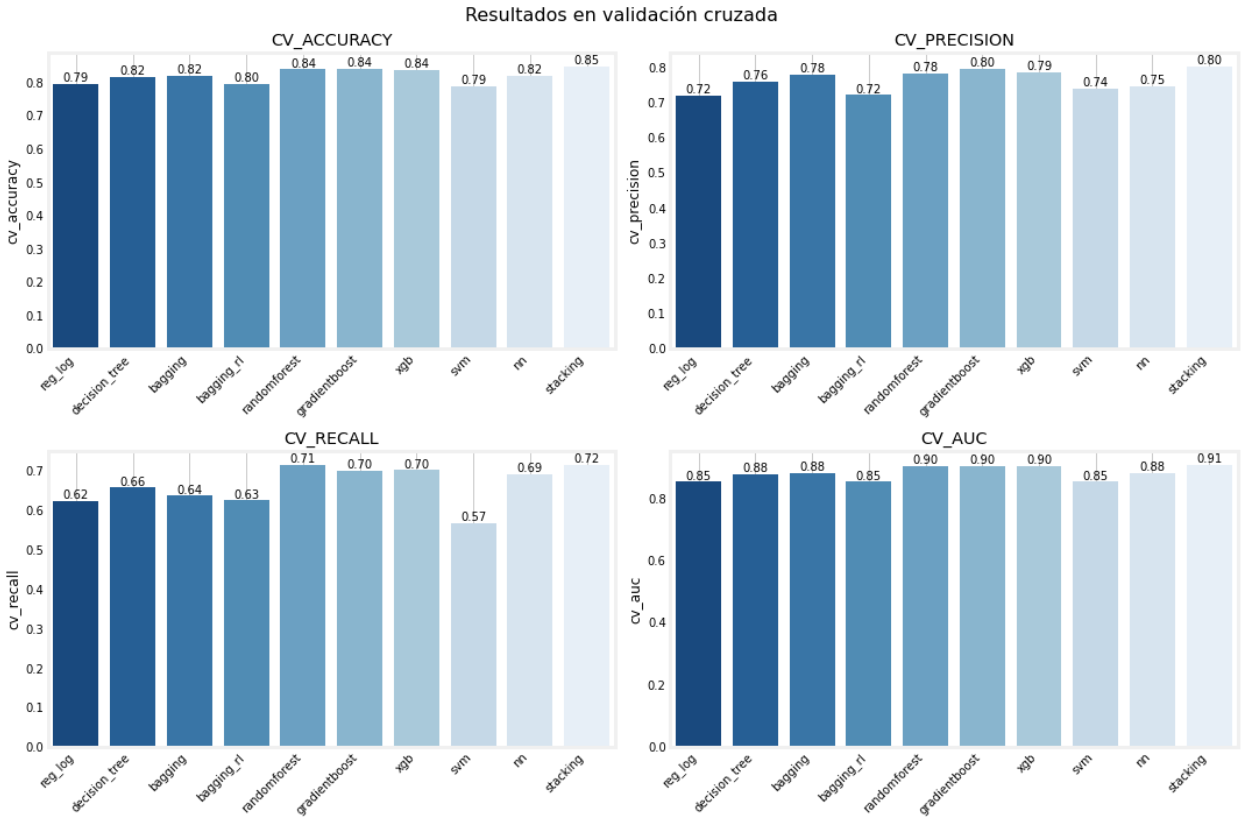


Selección del modelo ganador

Hemos generado modelos predictivos de diversos tipos de algoritmos. A la hora de evaluar cual es el mejor de ellos, hay que tener en cuenta tanto el puntaje (en este caso el área bajo la curva ROC y *recall*) como los costos computacionales, la explicabilidad del modelo, la complejidad y la interpretación.

Comencemos visualizando los resultados promedio en el test de validación cruzada para cada uno de los modelos (ver Figura 28). En cuanto a AUC, el mejor resultado fue el del modelo Stacking (0.91), seguido de XGB, Random Forest y Gradient Boosting, los tres en 0.90. Al observar el *recall* (cantidad de cancelaciones de reservas que hemos encontrado correctamente del total), el podio se repite, con valores apenas encima de 0.70. La *accuracy* de estos se encuentra alrededor de 0.85, y la precisión cerca de 0.80.

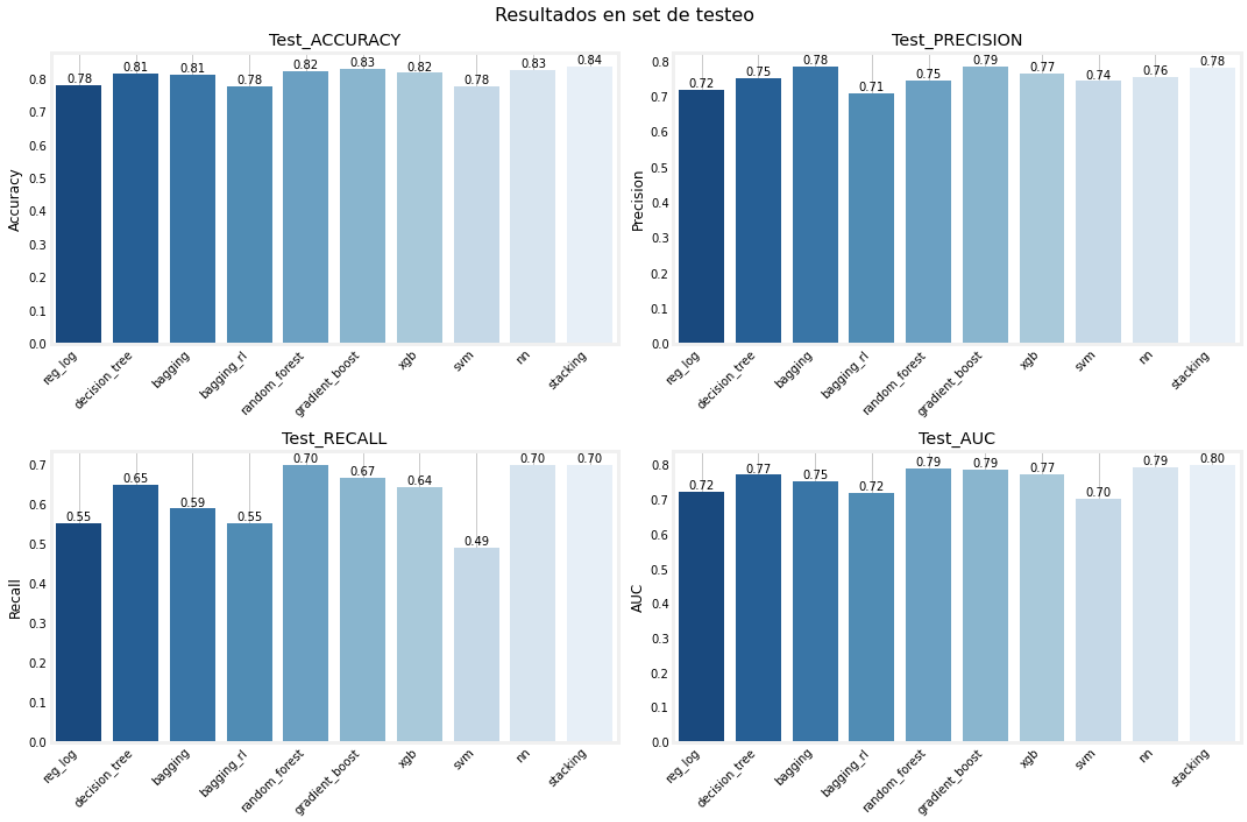
Figura 28. Resultados de modelos en validación cruzada



Los modelos más simples como la regresión logística o los árboles de decisión no tienen malos resultados en cuanto al AUC de validación cruzada, pero tienen menor *recall.* Por otro lado, las redes neuronales están en cercanas al podio en cuanto a las métricas comentadas previamente.

Ahora bien, cuando estos modelos se encuentran con datos nuevos, separados previamente en un set de testeo, todos reducen sus puntajes. El máximo AUC sigue siendo del modelo stacking, pero ahora de 0.80. Le siguen el modelo de Random Forest, Gradient Boosting y redes neuronales, con 0.79. En cuanto a *recall*, todos los mencionados salvo las redes entregan un valor razonable, de 0.70. El resto de los modelos se encuentran entre 0.60 y 0.65 (ver Figura 29). En cuanto a la precisión y la *accuracy*, el Stacking sigue siendo el ganador (0.84 y 0.78, respectivamente). El SVM no pudo encontrar un *recall* suficiente como para ser tenido en cuenta. Sin embargo, notar que este no fue optimizado a su máxima capacidad debido a recursos computacionales limitados, pero una vez hecho esto, podría llegar a encontrar resultados parecidos a los obtenidos por los otros modelos.

Figura 29. Resultados de modelos en set de testeo.



Como se comentó a lo largo del trabajo, cada uno de los modelos fue optimizado para encontrar el mejor resultado en términos de AUC. Sin embargo, se nota la presencia de muchos hiper-parámetros que no fueron optimizados, por lo que los puntajes podrían haber sido más altos todavía.

En base a los resultados comentados en esta sección, el modelo stacking parece obtener los mejores puntajes. Tanto en términos de AUC como en el resto de las métricas mencionadas, es el mas alto tanto en la validación cruzada como en el set de testeo reservado con datos nuevos. Aunque este modelo es un poco más costoso en términos de recursos computacionales, no fue una traba para ejecutar los códigos en esta prueba, por lo que sería el modelo seleccionado para seguir mejorando este análisis de cancelaciones de reservas. Si lo que se buscará fuera más explicabilidad de los resultados y se estuviera dispuesto a reducir la performance del modelo, podría optarse por el modelo Random Forest, que obtuvo resultados cercanos.

Documentación

La totalidad del código puede ser encontrado en el siguiente repositorio de GitHub:
https://github.com/DenisTros/Hotel_Reservations