Aplicación de Machine Learning a Predicción de Demanda

Tercer Informe de Introducción al Aprendizaje Automático y Aprendizaje Supervisado

12 DE SEPTIEMBRE DE 2021

Integrantes:

Gutiérrez Montecino, Denise

Nieva, Sofía

Rodríguez, Alfredo Manuel

Contenido

1 INTRODUCCIÓN

En el siguiente informe se detallan los resultados obtenidos del aprendizaje automático supervisado de la base de datos suministrada como parte de la Mentoría ***Predicciones de Demanda de Producto*** de la Diplomatura de Ciencia de Datos de la Facultad de Matemáticas, Astronomía y Física (FaMAF) de la Universidad Nacional de Córdoba, Argentina.

El conjunto de datos de partida contiene productos vendidos de los últimos 5 años de una compañía de venta de alimentos congelados en distintos países de la región. El objetivo final de la mentoría es poder predecir la demanda de los productos elaborados en los centros de elaboración mes a mes en los diferentes países y zonas en donde opera esta compañía.

El presente entregable contiene documentación de la introducción a aprendizaje automático y aprendizaje supervisado, utilizando como base el dataset obtenido del entregable de exploración y curación filtrado por la localidad de Córdoba, Córdoba. El objetivo de este informe es experimentar aplicando distintos modelos de aprendizaje supervisado y optimizarlos para obtener mejores predicciones.

2 RESPUESTAS DEL PRÁCTICO

En esta sección se desarrollarán las consignas dadas para el informe con sus respectivas observaciones y comentarios de los resultados obtenidos así como también de las decisiones que se fueron realizando, si fuese el caso. A continuación se describe cada una de ellas.

1. Obtención dataset

En esta sección se obtiene el dataset que se utilizará para desarrollar las consignas del presente informe. Para ello, se utilizó el dataset resultante de la exploración y curación desarrollado para el entregable anterior.

En primer lugar, se filtró por la provincia y localidad de Córdoba para realizar las primeras iteraciones de entrenamiento del modelo. Las variables consideradas fueron cantidad_pedida (con sus respectivos 3 lags), cantidad_pedida_mean (resultante de los cálculos de los 3 lags), año (2018 y 2019), mes, categoría_depurada, sku, unidadkg, totalkg, id_proveedor, marca_depurada, presentacion_depurada y ubicación (ya filtrado por Córdoba):

2. Multicolinealidad Exacta

La multicolinealidad es la correlación alta entre más de dos variables explicativas lo cual deriva en la imposibilidad de estimar los parámetros de forma precisa. Para detectar la multicolinealidad se utilizó el método descrito en https://www.kaggle.com/remilpm/how-to-remove-multicollinearity.

A partir de la aplicación de este método se encontraron varias variables que presentan colinealidad. A continuación se describe cada una de ellas junto con las decisiones que se tomaron:

1. Año = 2018: es un resultado de esperarse dado que existen dos años en la base de datos y por lo tanto al aplicar el one-hot encoding resulta que [columna año 2019] = 1 - [columna año 2018]. Se decidió por lo tanto eliminar la columna del dataset.
2. Mes = 10: No es un resultado que uno esperaría, por lo tanto no se eliminó.
3. Categorías, marcas, presentaciones depuradas y id_proveedor: Se eliminaron las que el método señaló como altamente correlacionadas, ya que probablemente la información que aportaban era redundante con la de la columna sku.
4. sku: se eliminaron los dos que indicaba el método, 1022 y otros .

Además, de las columnas que quedaban se analizó cuales tienen mayor correlación simple.

Filtrando solo las que tenían índice de correlación > 0.7, se obtuvo la siguiente matriz:

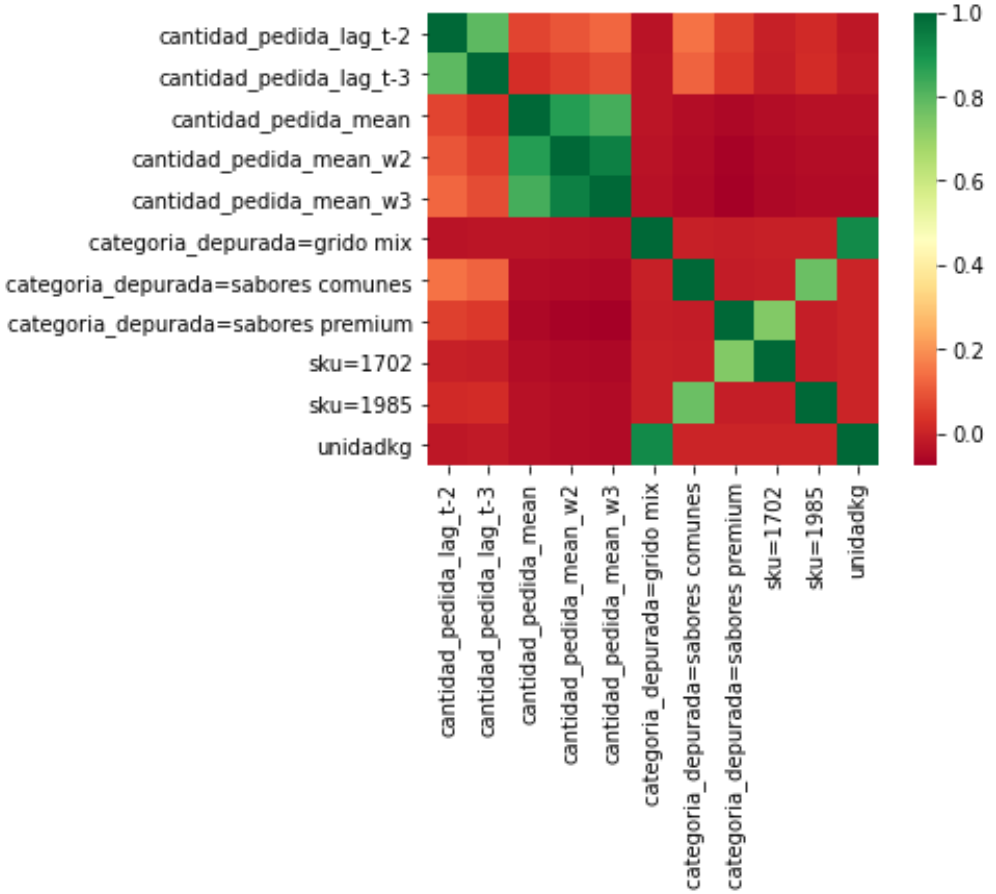

Figura 1 – Matriz de correlación

Se pudo observar que los lags y las medias móviles estaban muy relacionados entre sí, lo cual es de esperarse, por lo que no se eliminaron. Pero sí se eliminó la categoría grido mix que estaba muy correlacionada con unidadkg y las categorías sabores comunes y sabores premium que estaban muy correlacionadas con los sku 1985 y 1702 respectivamente.

3. Comprobación de Series de Tiempo Estacionaria y Tendencia

En este inciso se comprueba con qué tipo de serie de tiempo se está trabajando. El objetivo es determinar si la serie de tiempo es estacionaria o no, que será determinante para el desarrollo del resto de los incisos.

Para la corroboración se utilizaron dos test estadísticos, a continuación se describe brevemente cada uno de ellos:

- Dickey Fuller Aumentado (ADF): es un test estadístico para la verificación de estacionalidad de una serie. En otras palabras, es una prueba de raíz unitaria para una muestra de serie de tiempo. Su hipótesis nula es que la serie es de raíz unitaria o es no estacionaria. Mientras más negativo sea, más fuerte es el rechazo de la hipótesis nula.
- Kwiatkowski–Phillips–Schmidt–Shin (KPSS): es un test estadístico para la verificación de estacionalidad de una serie alrededor de una tendencia determinística. La hipótesis nula es que la serie es estacionaria.

Se aplicaron ambos test sobre la variable cantidad_pedida como resultado se obtuvo que para la prueba de ADF se rechaza la hipótesis nula con un nivel de significancia de 0.05, es decir, que existe suficiente evidencia estadística para determinar que la serie es estacionaria. Mientras que para el KPPS no se rechaza la hipótesis nula, es decir, que la serie es estacionaria entorno a una tendencia determinística para el mismo nivel de significancia. Por lo tanto, dado que ambos test concluyen que la serie es estacionaria se puede decir que corresponde a una serie estacionaria estricta.

Para modelar mejor la serie, se agregó además una nueva feature: la diferencia entre la variable dependiente y su valor (shift) anterior, ya que esto enriquecerá el dataset lo suficiente como para eliminar la tendencia.

4. Normalización y Estandarización de Atributos

Para algoritmos basados en árboles, como XGBoost o Random Forest, no es necesario escalar la data, pero para K neighbors sí es conveniente. Por lo tanto, se aplicó el MinMaxScaler y se utilizó la data escalada en los modelos donde esto era necesario.

5. División en Train/Test

Para la división del dataset, se utilizó un 20% de los datos disponibles como conjunto de test y el restante 80% para entrenamiento. Teniendo en cuenta que estamos utilizando series temporales la división de train y test debe respetar la línea de tiempo. Esto es, el 80% de datos más antiguos serán para train y el 20% más recientes serán utilizados para test. Además, dentro del 80% de train, para poder utilizar grid search con cross validation hay que tener en cuenta que se está trabajando con series de tiempo y por lo tanto no es posible utilizar K-Fold Cross Validation dado que esta forma de validación cruzada no respeta el orden de las observaciones. Por eso se utilizó la función TimeSeriesSplit() con n_splits=10, que es una variación de k-fold que devuelve los primeros pliegues como conjunto de entrenamiento y los últimos como conjunto de test, respetando así el orden temporal.

6. Grid Search de los Modelos Seleccionados

Para experimentar con los distintos modelos se procedió de la siguiente forma:

- Primero se investigó, para cada modelo, cuáles eran los hiperparámetros más importantes y su posible rango de valor y con esta información se procedió a armar grillas de parámetros.
- Luego se llamó a la función GridSearch(), pasándole como argumentos el modelo, la grilla correspondiente a ese modelo, la métrica respecto a la cual optimizar los hiperparámetros, en este caso el r2-score, y los folds obtenidos de TimeSeriesSplit().
- Se fitteo el GridSearch a la data, utilizando la data escalada cuando correspondía.
- Finalmente se evaluaron los resultados y se guardó el modelo con los hipérparametros que maximizan nuestra métrica.

A continuación mostramos los gráficos de la performance aplicando cross validation en el conjunto de entrenamiento de cada modelo obtenido comparado con el mismo modelo pero utilizando los parámetros que vienen por default, es decir, que no fueron optimizados para este problema en particular.

A. Parámetros del Random Forest optimizado:

{'criterion': 'mse', 'max_depth': 11, 'max_features': 'auto', 'min_samples_leaf': 2, 'n_estimators': 50}

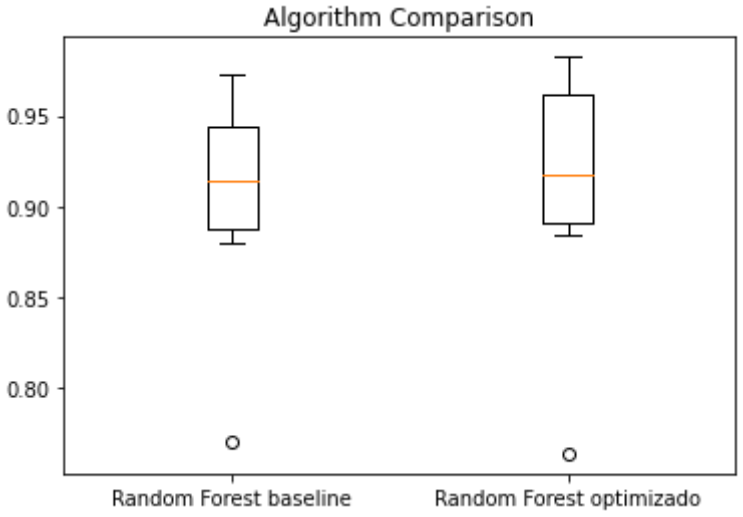

Figura 2 – Comparación de modelos Random Forest con distintos parámetros

B. Parámetros del XGBoost optimizado:

{'colsample_bytree': 0.8, 'gamma': 0, 'learning_rate': 0.2, 'max_depth': 1, 'n_estimators': 1000, 'reg_alpha': 0.01, 'reg_lambda': 1, 'subsample': 0.6}

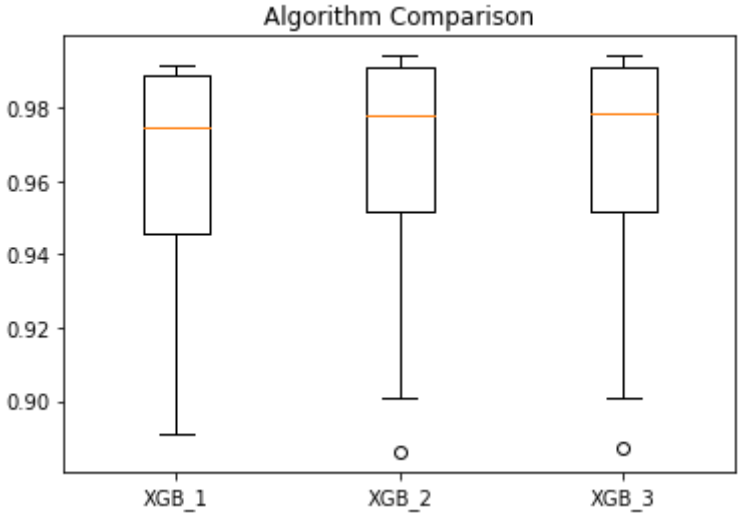

Figura 3 – Comparación de modelos XGBoost con distintos parámetros

En el caso del XGBoost, como es un modelo más pesado y tarda mucho en correr se realizaron distintos grid searches:

- Primero un RandomizedSearchCV() con 100 iteraciones para encontrar un rango más acotado para los hiperparámetros y así disminuir la cantidad de posibilidades de una búsqueda exhaustiva. XGB_1 es el modelo que se obtuvo como mejor modelo de esta primera búsqueda.

- Luego se hizo un GridSearch() para algunos de los parametros ('max_depth', 'n_estimators', 'learning_rate', 'colsample_bytree', 'subsample') y se obtuvo como mejor modelo XGB_2, que, como se puede observar, supera en performance a XGB_1.
- Por último, se realizó otro GridSearch(), esta vez incluyendo también los parametros de regularización ('gamma', 'reg_alpha' y 'reg_lambda') y se obtuvo como mejor modelo XGB_3 que mejora muy levemente la performance de XGB_2. Por lo tanto, se eligió a XGB_3 como representante del XGBoost.

El XGBoost además, al estar basado en árboles de decisión, permite calcular la feature importance, la cual nos indica cuales son las variables que más información aportan para predecir la variable objetivo. La siguiente figura muestra los resultados que se obtuvieron.

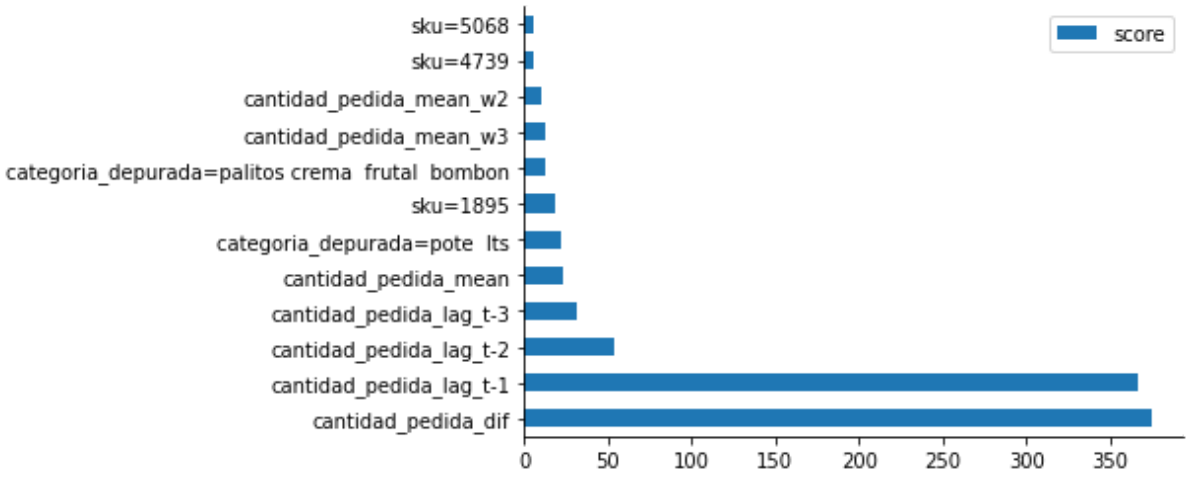

Figura 4 – Feature Importance, basado en el modelo XGB_3

C. Parámetros del K Neighbor Regressor optimizado:

{'algorithm': 'auto', 'metric': 'minkowski', 'n_neighbors': 5, 'p': 2, 'weights': 'distance'}

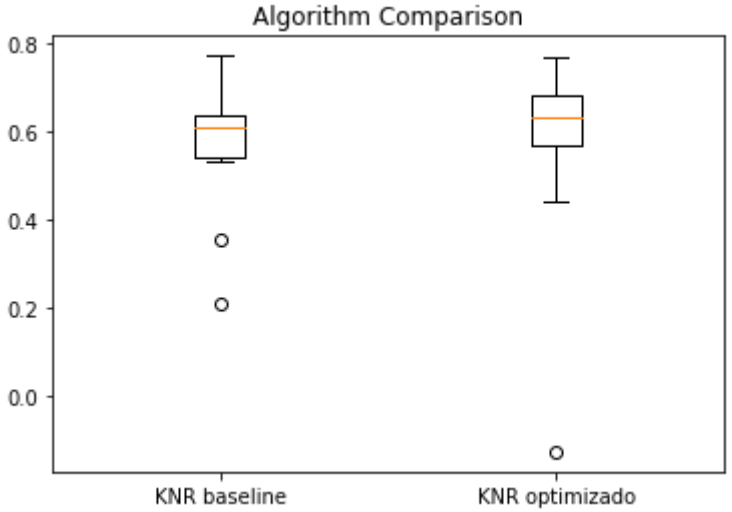

Figura 5 – Comparación de modelos KNeighbors con distintos parámetros

D. Parámetros del LGBM optimizado:

{'learning_rate': 0.1, 'max_depth': 6, 'n_estimators': 200, 'num_leaves': 31}

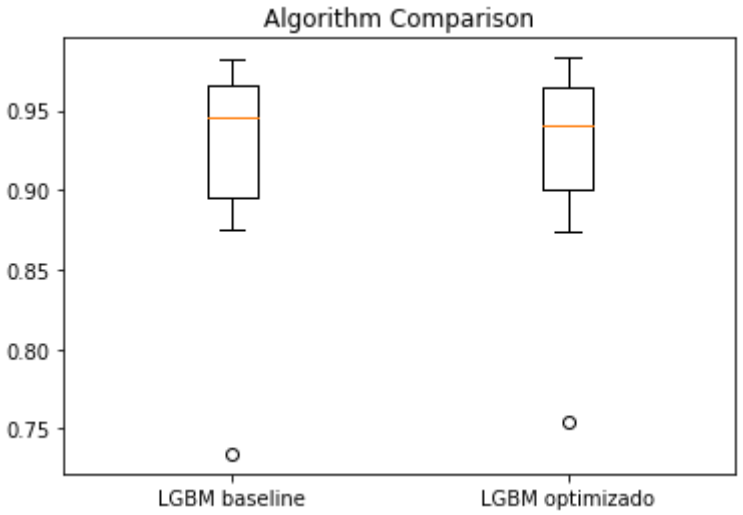

Figura 6 – Comparación de modelos LightGBM con distintos parámetros

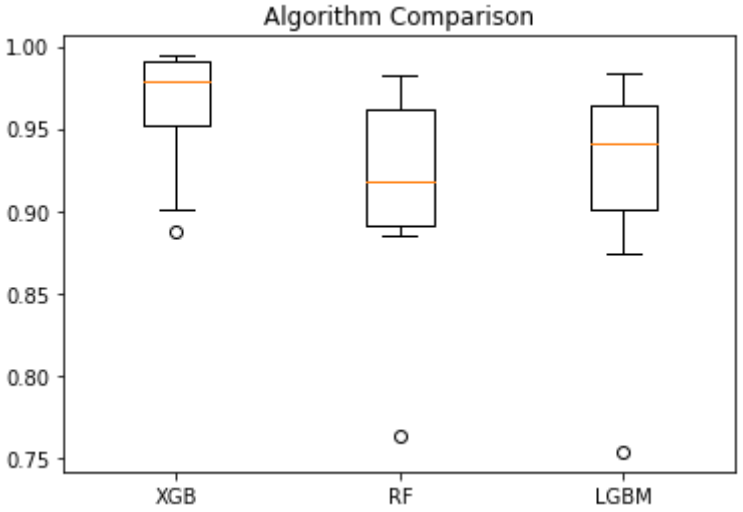

Figura 7 – Comparación de los mejores modelos

modelo	rmse	mape	r2
XGB	11.547643	0.201504	0.987865
RF	22.110115	0.266866	0.955514
LGBM	24.496418	0.258762	0.945393

Tabla 1 - Comparación de las distintas métricas para los mejores modelos

Por último se probó de armar distintos votings utilizando el VotingRegressor, para ver si se podían mejorar aún más las métricas. No se incluyó el KNeighborsRegressor en el voting ya que su performance era mucho menor que la del resto de los modelos.

Se realizó un GridSearch() para cada voting para encontrar la mejor combinación de pesos para los distintos modelos que participan de la votación y se lo comparó con un voting con pesos uniformes de los mismos modelos pero utilizando los parámetros por defecto (voting baseline). Los resultados fueron los siguientes:

- Parametros VotingRegressor([('XGB', xgb3), ('RFR', rf1)]): {'weights': array([0.5, 0.1])}

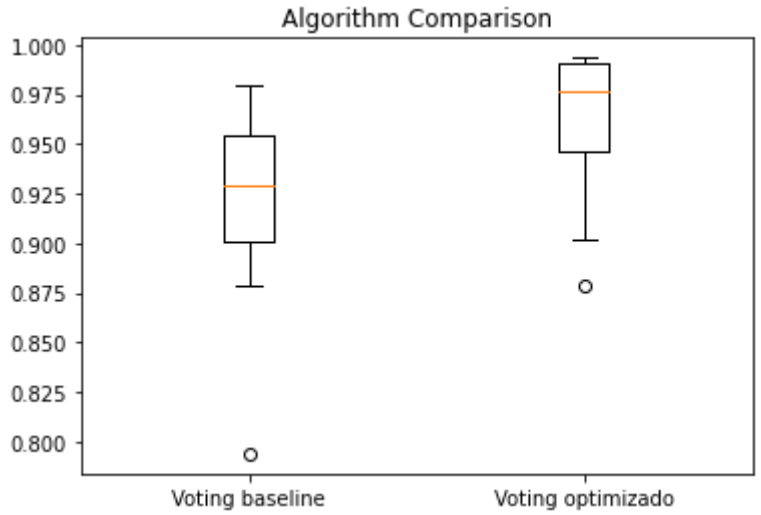

Figura 8 – Comparación de distintos votings con Random Forest y XGBoost

- Parametros VotingRegressor([('XGB', xgb3), ('RFR', rf1), ('LGBM', lgbm1)]): {'weights': array([0.8, 0.4, 0.1])}

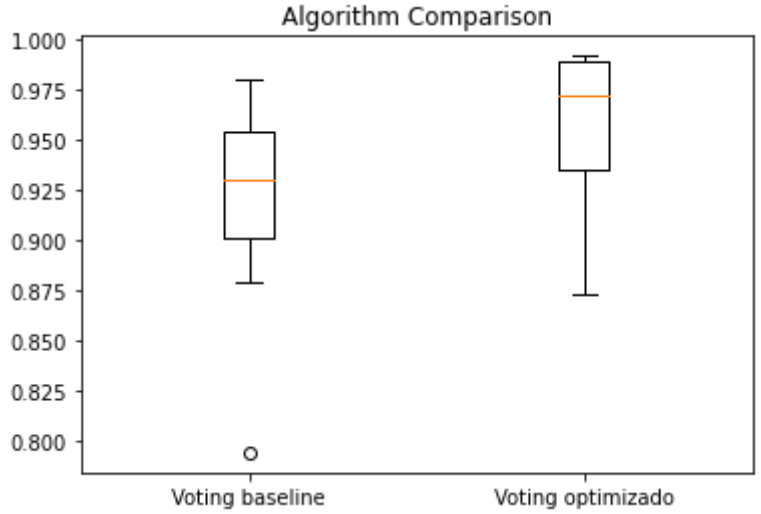

Figura 9 – Comparación de distintos votings con Random Forest, XGBoost y LGBM

Por último comparamos los tres votings juntos

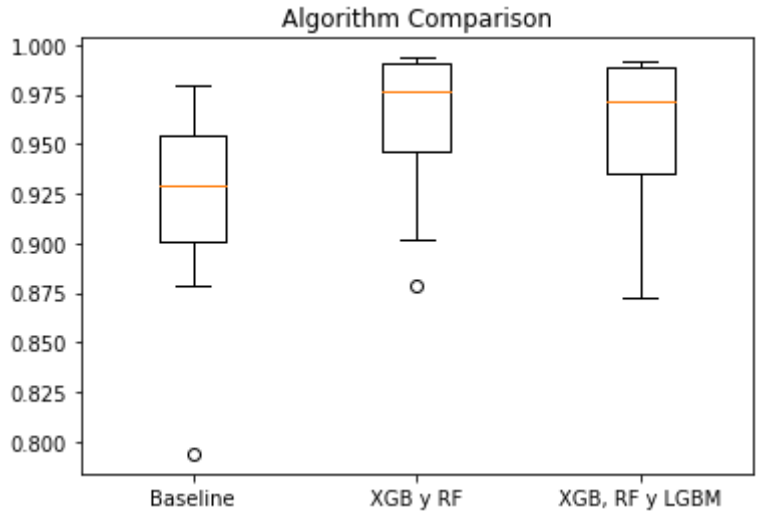

Figura 10 – Comparación de los distintos votings

modelo	rmse	mape	r2
Baseline	21.219175	0.259929	0.959027
XGB y RF	12.191981	0.204646	0.986473
XGB, RF y LGBM	14.005493	0.208036	0.982150

Tabla 1 - Comparación de las distintas métricas para los distintos votings

3 PRINCIPALES CONCLUSIONES

En general, la mayoría de los modelos tuvieron una muy buena performance, con un r2-score de más de 0.9. Esto se debe a todas las features con las que se enriqueció el dataset, en especial los lags, las medias móviles y las diferencias, como se pudo ver cuando se analizó la feature importance.

De los distintos votings, con el que se obtuvo mejores resultados fue el que combinaba XGBRegressor con parámetros:

- random_state=0
- subsample=0.6
- n_estimators=1000
- max_depth=1
- learning_rate=0.2
- colsample_bytree=0.8
- gamma=0
- reg_alpha=0.01
- reg_lambda=1

y RandomForestRegressor con parámetros:

- random_state=0
- criterion='mse'
- max_depth=11
- max_features='auto'
- min_samples_leaf=2
- n_estimators=50

y pesos: 0.5 para XGBRegressor y 0.1 para RandomForestRegressor.

Sin embargo, se puede observar que el modelo XGBRegressor sólo, con los parámetros antes mencionados, tiene una mejor performance que la del voting. Por lo tanto, fue finalmente el modelo elegido.