Opportunities@MeLi
Code Exercise

Data Scientist – Cristian Cardozo

1. **Introducción y objetivo:**

En MercadoLibre se requiere un algoritmo que prediga la condición de un ítem, determinando si es usado o nuevo. El objetivo es claro: clasificar cada ítem como 1 si es usado o 0 si es nuevo.

2. **Entregables:**

- Notebook:
  - exploratory_analysis.ipynb
  - model_01.ipynb
  - model_02.ipynb
- Scripts Python:
  - Preprocessing.py
  - Transformation.py
  - model_processing.py
  - new_or_used.py

3. **Descripción de los datos:**

Para comenzar, los datos provienen de una API en formato JSON, por lo que es necesario transformarlos en una estructura legible y utilizable por algoritmos de machine learning. Para ello, utilizaremos las funciones definidas en el script preprocessing.py.

Leer un archivo JSON es relativamente sencillo, sin embargo, el reto principal radica en manejar las estructuras anidadas, es decir, cuando una clave contiene otras características en su interior. Para resolver este problema, implementaremos una función recursiva que explore cada fila, identifique las estructuras anidadas y las expanda en columnas separadas.

```python
def explot_columns_nested(df):
    """
    expande los los campos anidados de un .json
    """
    cols_nested = get_columns_nested(df)

    if not cols_nested:
        return df

    for col in cols_nested:
        df_norm = df[col].apply(normalize_key)
        df_norm = pd.json_normalize(df_norm)
        df_norm = df_norm.add_prefix(f"{col}_")
        df = pd.concat([df, df_norm], axis=1)

    df.drop(columns=cols_nested, inplace=True)

    return explot_columns_nested(df)
```

Una vez desplegada la información, procedemos a su análisis. Para facilitar la extracción de insights, utilizamos una librería que genera automáticamente un EDA. Esto me permitió identificar rápidamente las transformaciones necesarias y realizar una primera selección de variables basándonos en la completitud de los datos

```
profile = ProfileReport(dataframe_raw)
```

YData Profiling Report — Overview | Variables | Interactions | Correlations | Missing values | Sample

Overview | Alerts 118 | Reproduction

Dataset statistics

Number of variables	79
Number of observations	100000
Missing cells	2624908
Missing cells (%)	33.2%
Duplicate rows	0
Duplicate rows (%)	0.0%
Total size in memory	57.6 MiB
Average record size in memory	604.0 B

Variable types

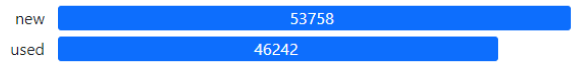
Text	20
Unsupported	11
Categorical	27
Numeric	15
DateTime	2
Boolean	4

condition

Categorical

High correlation

Distinct	2
Distinct (%)	< 0.1%
Missing	0
Missing (%)	0.0%
Memory size	781.4 KiB



4. **Transformación de datos:**

A partir de esta información se concluyó, varias transformaciones necesarias.

Por ejemplo:

- Conversión de moneda: Dado que la variable "base_price" puede incluir montos en dólares y pesos, se debe convertir todos los valores a una única moneda (por ejemplo, pesos argentinos) usando la tasa de conversión vigente en la fecha correspondiente (considerando si se usa el dólar blue o el oficial).

- Extracción de componentes: La variable "last_updated" contiene tanto fecha como hora. Se descompondrá en variables numéricas separadas: año, mes, día, día de la semana, semana ISO, hora, minuto y segundo.

Estas transformaciones están definidas en transformation.py y se emplearán posteriormente en nuestro mini pipeline.

Dentro de estas transformaciones se incluyen:

- Conversión de categóricas a booleanas:
  Cuando el porcentaje de nulos es alto, se transforma la variable en un indicador booleano para evitar pérdida de información.

- Extracción de latitud y longitud:
  Se realiza una consulta web a MercadoLibre API para obtener la latitud y longitud de las ciudades, reduciendo la alta cardinalidad.

- Eliminación de variables con más del 95% de nulos:
  Se descartan aquellas variables que aportan muy poca información.

- Conversión de dimensiones de fotos:
  El tamaño de las fotos se divide en dos variables numéricas (alto y ancho), pasando de una representación categórica a una numérica.

- Reducción de categorías dominantes:
  Para variables categóricas en las que una sola categoría representa más del 95% de los casos, se transforma la variable en un indicador binario que señale si el registro pertenece o no a dicha categoría dominante.

- Eliminación de variables con alta cardinalidad:
  Se eliminan variables con un alto porcentaje de valores únicos, como identificadores o URLs, que pueden complicar el modelo.

- Transformación de variables ordinales:
  Variables como listing_type_id se convierten a valores numéricos para reflejar su orden inherente.

Cabe destacar que, si bien aún falta el paso extra de selección de variables, utilizaremos modelos de ensamble. Estos modelos, aunque no son completamente inmunes a la multicolinealidad, son más flexibles y tolerantes a ella.

Cabe aclarar que también disponemos de funciones adicionales para el procesamiento de los datos. En el script model_processing.py se realiza lo siguiente: las variables numéricas se estandarizan, las variables categóricas se convierten en dummies y las variables booleanas se transforman en 1 y 0. La idea es que todas las variables queden en un formato numérico.

5. **Modelos:**

Como se adelantó, utilizaremos dos modelos: Extreme Gradient Boosting y Light Gradient Boosting. Debido a que muchas de nuestras variables tienen un alto porcentaje de valores nulos (más del 80%), estos métodos nos permiten evitar transformaciones o imputaciones adicionales, ya que los modelos de ensamble son robustos frente a la multicolinealidad y a datos incompletos. Sin embargo, es importante tener en cuenta que, con conjuntos de datos pequeños, estos modelos pueden sobreajustarse, por lo que se debe prestar especial atención a la selección de hiperparámetros adecuados.

La generación de estos modelos se detalla en los notebooks model_01.ipynb y model_02.ipynb, donde se implementa XGBoost y LightGBM, respectivamente.

Ahora bien, se utilizó GridSearchCV para encontrar los hiperparámetros adecuados, optimizando por acurracy, métrica que solicita el ejercicio:

```
param_grid = {
            'max_depth': [3, 5, 7],
            'learning_rate': [0.01, 0.1, 0.2],
            'n_estimators': [43, 87, 150]
        }
```

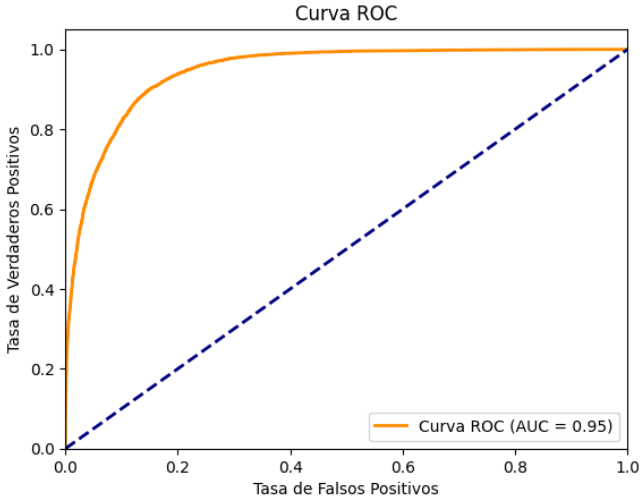
6. **Desempeño de los Modelos:**

Comencemos con el modelo de **Light Gradient Boosting**. En términos de **accuracy**, nuestra métrica principal, hemos cumplido e incluso superado el mínimo requerido, alcanzando un valor de 0.8733, lo que demuestra un desempeño robusto. Aunque el lift es una métrica que personalmente valoro, en este caso no resulta tan útil debido al balance de clases. Por ello, optamos por el F1 score, una métrica que equilibra recall y precisión. Esto nos permite evaluar no solo la tasa de aciertos, sino también la capacidad del modelo para identificar correctamente cada clase, ofreciendo una visión más completa y comprensible del rendimiento global.

En este caso el modelo tuvo un F1 score de 0.8675

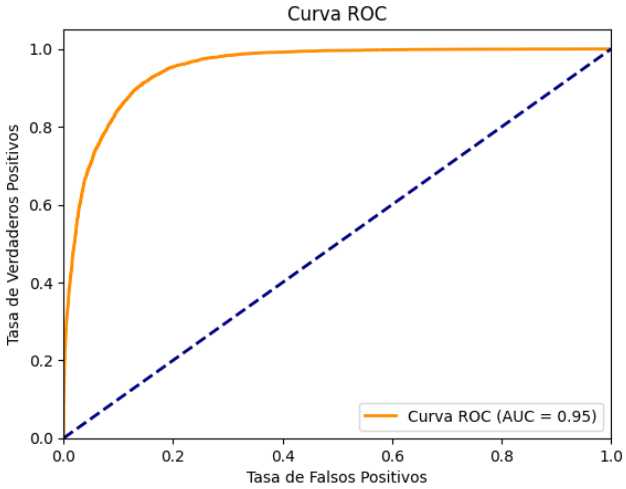
Otra métrica que me gusta es la **curva ROC**, ya que nos permite visualizar la relación entre la tasa de verdaderos positivos y la tasa de falsos positivos. El área bajo la curva (**AUC**) es especialmente útil, ya que proporciona una medida global del rendimiento, donde un valor cercano a 1 indica un excelente desempeño y un valor de 0.5 sugiere que el modelo actúa como un clasificador aleatorio.

Y lo mejor de todo es que la representación gráfica es sumamente intuitiva, entre más convexa la curva mejor:

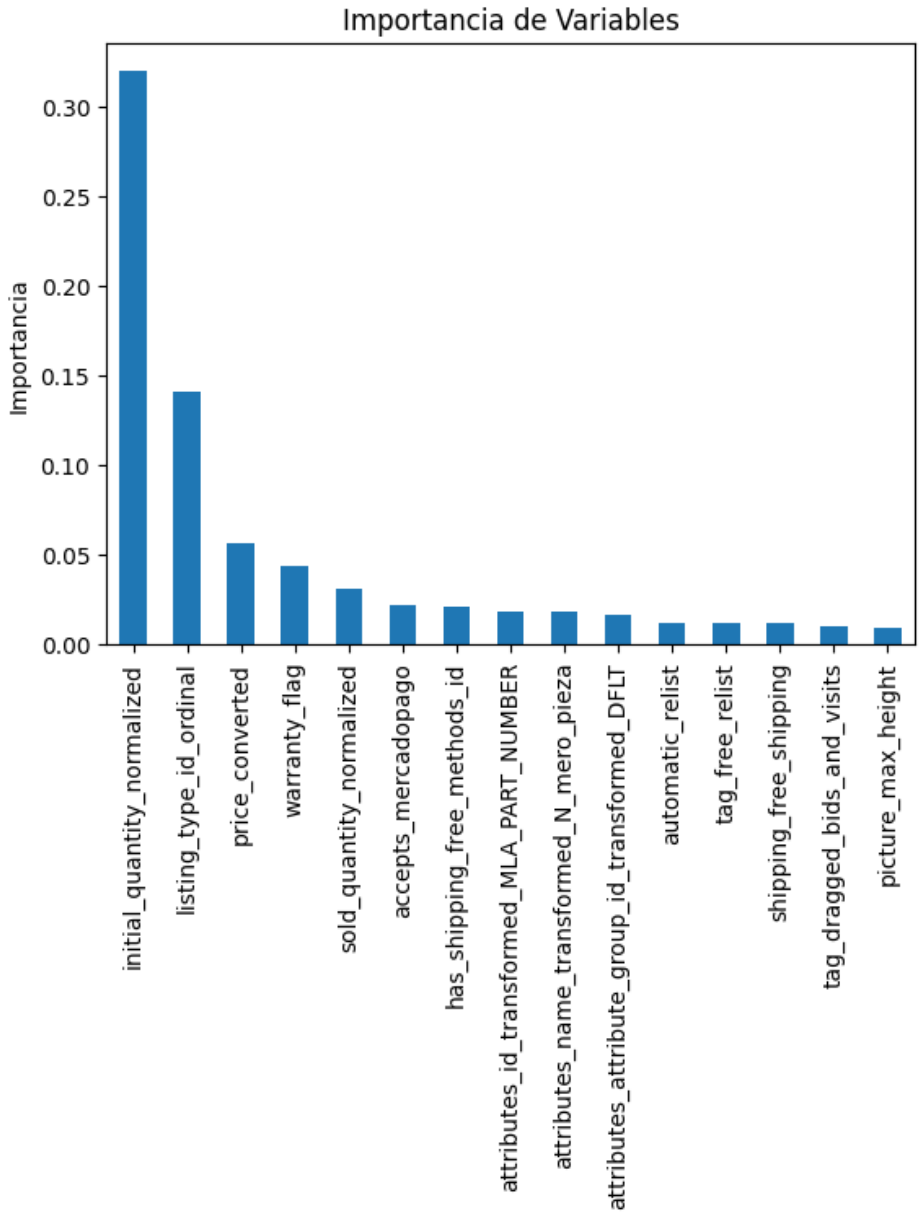


Por lo que mi propuesta como métrica secundaria sería el AUC.

Por otro lado, el modelo Extreme Gradient Boosting no se quedó atrás, de hecho, superó las métricas obtenidas por el Light Gradient Boosting, alcanzando un accuracy de 0.881, un F1-score de 0.875 y una destacada curva ROC.:



Por otro lado, ya yendo más hacia las variables, tenemos que las transformaciones que hicimos tuvieron resultado.



Despliegue:

En este caso, crearemos un código sencillo utilizando FastAPI para consumir una API cuyo principal input será una base de datos y que devolverá las predicciones correspondientes a dicha base. Este código lo guardaremos en el archivo api_model_serving.py.