

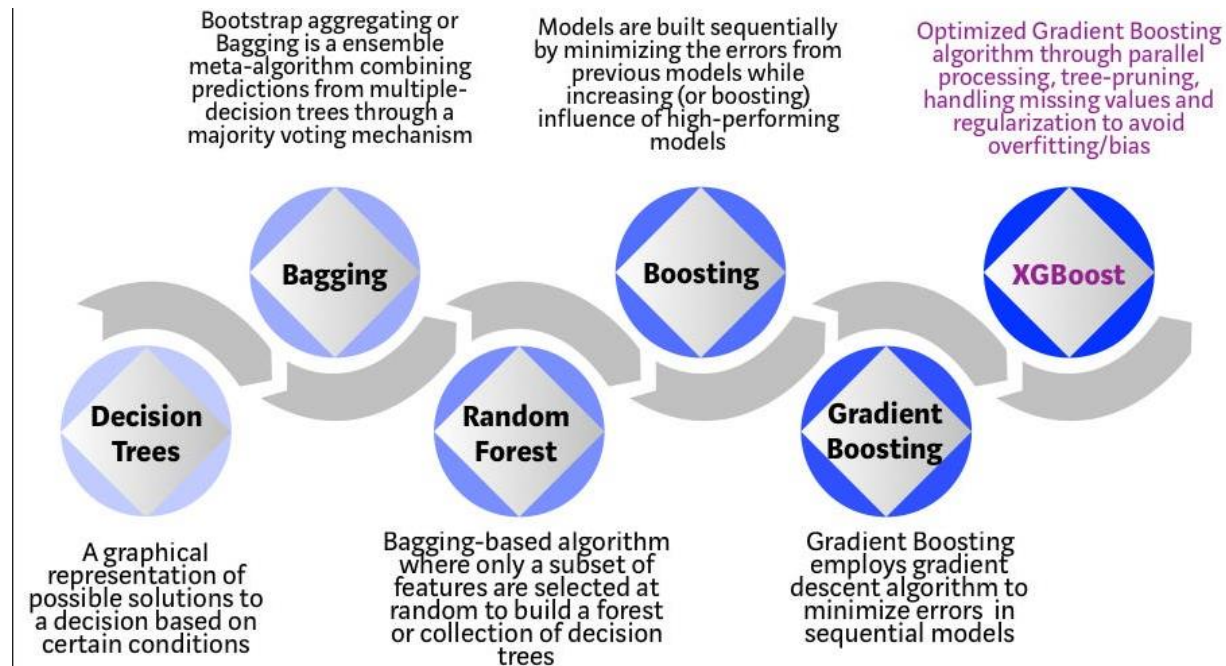
# Actividad 3 Detalle el Siguiente notebook XGBOOST

Carlos Mauricio Arteaga Bermúdez

MACHINE LEARNING I M7MLRU2

2 de noviembre de 2024

# Introducción al Algoritmo XGBoost



La imagen hace referencia a diferentes algoritmos de aprendizaje derivados de los árboles de decisión y como de izquierda a derecha los algoritmos van desde el mas simple al más robusto XGBoost.

- Decision Trees: Es la base de todos los otros modelos.
- Bagging: combina las predicciones de múltiples árboles de decisión de forma paralela, utilizando un mecanismo de votación para generar una predicción final.

Un ejemplo de bagging es el Random Forest, que selecciona aleatoriamente un subconjunto de características para construir varios árboles. Cada árbol trabaja de forma independiente, y el resultado final se obtiene al promediar o votar entre las predicciones de todos los árboles.

- Boosting : El boosting entrena los árboles de manera secuencial, donde cada árbol intenta corregir los errores del árbol anterior.
- Gradient Boosting: Utiliza un método de descenso de gradiente para optimizar cada árbol en la secuencia y reducir los errores, se centra en minimizar los errores de los modelos anteriores al construir nuevos modelos en la secuencia.
- XGBoost: Este algoritmo es una versión optimizada de Gradient Boosting, esta diseñado para ser extremadamente eficiente y rápido, este algoritmo se caracteriza

por un procesamiento en paralelo, poda de árboles, manejo de valores faltantes y regularización para evitar el sobreajuste o sesgo.

## Introducción al Algoritmo XGBoost

XGBoost, Es una librería de machine learning de alto rendimiento basada en el trabajo de Jerome H. Friedman derivada del artículo Greedy function approximation: A gradient boosting machine.

- Es de código abierto
- El objetivo es implementar árboles de decisión optimizados por gradiente.
- Originalmente escrito en C++

## Características Principales de XGBoost

Las razones para utilizar este algoritmo es su precisión y eficiencia frente a los recursos puede realizar cálculos en paralelo y se utiliza para grandes volúmenes de datos.

- Incluye funcionalidades para realizar validación cruzada.
- Características:
  - Sparsity (Esparcidad): Permite entrenar modelos con datos incompletos o datos faltantes sin que afecte mucho el rendimiento.
  - Customization (Personalización): Acepta funciones personalizadas, flexibilidad para las necesidades del usuario.
  - DMatrix: Utiliza una estructura de datos optimizada

## Bagging vs Boosting

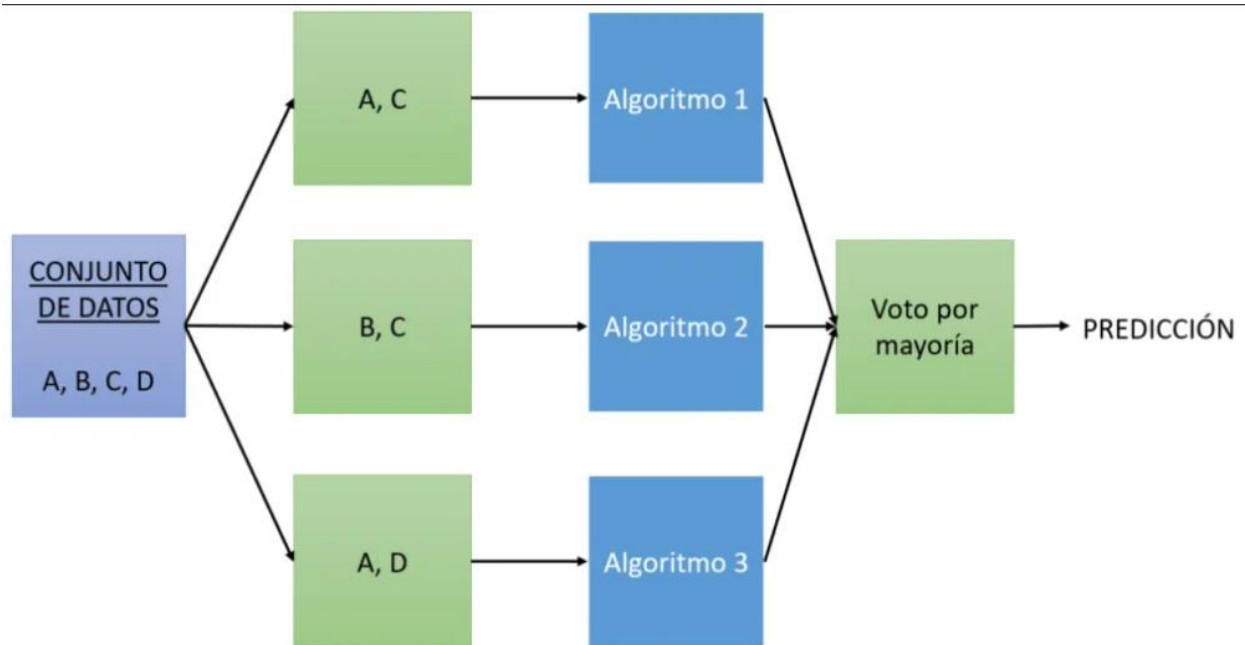
Métodos de Ensemble:

Los métodos de ensemble combinan varios árboles de decisión para obtener una predicción más robusta y precisa en comparación con usar un solo árbol de decisión.

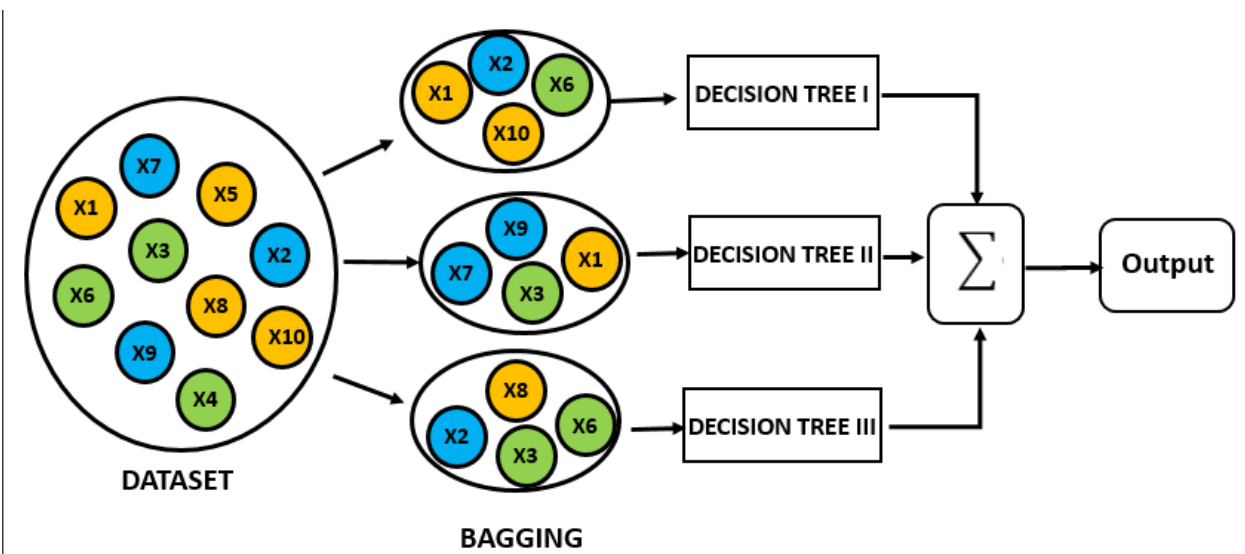
El principio detrás del ensemble es que la combinación de múltiples "árboles débiles" puede crear un "árbol fuerte" con un mejor rendimiento general.

## Bagging

Es un método de ensemble en el que varios árboles de decisión se entrenan de manera independiente. Luego, sus predicciones se combinan (usualmente mediante votación o promedio) para obtener un resultado final. Ejemplos de bagging incluyen el algoritmo Random Forest.



Proceso de ensamble basado en votación, varios algoritmos trabajan sobre varios subconjuntos de datos, cada algoritmo se entrena en su respectiva combinación y las combinaciones de estos algoritmos se combinan a través de una votación por mayoría para obtener una predicción final.



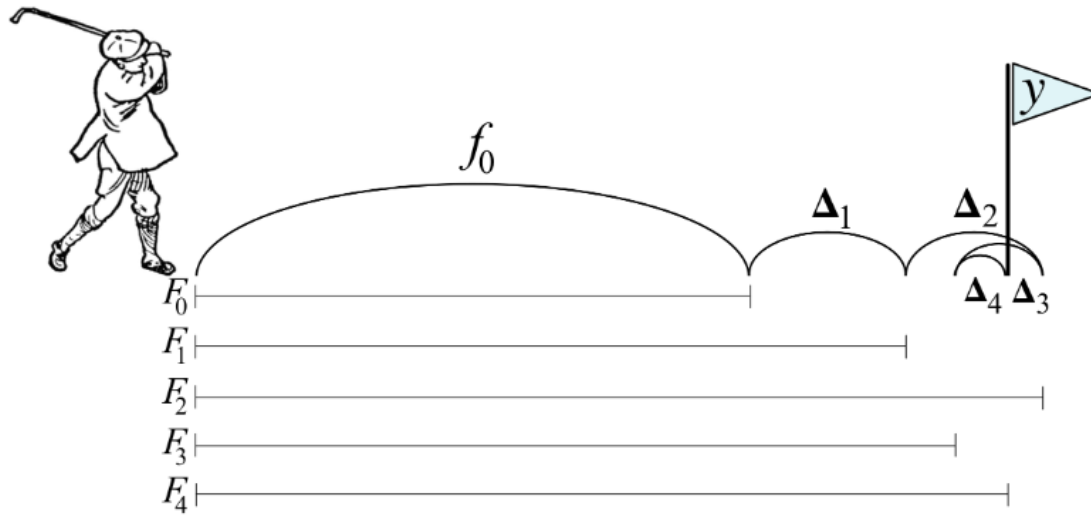
Al igual que la primera imagen, representa un proceso de entrenar diferentes algoritmos de árbol de decisión, al final los resultados se combinan mediante un promedio o una votación.

Este enfoque ayuda a reducir la varianza del modelo y hace que el ensemble sea menos sensible al sobreajuste en comparación con un solo árbol.

## Boosting

- Es una técnica de manejo del algoritmo donde cada iteración pretende mejorar la eficiencia del que le precedió, el modelo se entrena de manera secuencial.
- Los primeros modelos tienden a ser simples y no muy precisos, pero con cada paso, los modelos ajustan sus predicciones enfocándose en los errores previos.
- Si una observación es clasificada incorrectamente, su peso se incrementa, lo que significa que los modelos posteriores le prestarán más atención. De esta forma, el boosting se enfoca más en los casos difíciles de clasificar.
- En cada paso, el objetivo del boosting es reducir el error total. Cada modelo intenta resolver el error neto del modelo anterior, mejorando la precisión global del ensemble.
- boosting es un ajuste de árboles de decisión consecutivos, donde cada uno intenta corregir los errores del anterior. Esto contrasta con el bagging, donde los modelos se entrenan de forma independiente.

La idea del *boosting*, extraída de [aquí](http://etc.usf.edu/clipart/), podría verse como un golfista que inicialmente golpea una pelota de golf hacia el hoyo que se encuentra en la posición  $y$ , pero que solo llega hasta  $f_0$ . Luego, el golfista golpea la pelota repetidamente de forma más suave, moviéndola hacia el hoyo poco a poco y después de reevaluar la dirección y la distancia al hoyo en cada golpeo. El siguiente diagrama ilustra 5 golpes que llegan al hoyo,  $y$ , incluidos dos golpes que sobrepasan el hoyo. (Clipart de golfista de <http://etc.usf.edu/clipart/>)



Ejemplo del golfista para el boosting

De forma matemática podríamos decir que una *máquina de aumento de gradiente* (GBM) es un algoritmo de modelado aditivo que construye gradualmente un modelo compuesto agregando iterativamente  $M$  submodelos débiles basados en el rendimiento compuesto de las iteraciones anteriores:

La analogía del golfista ayuda a entender como el proceso iterativo va acumulando ajustes hasta que el modelo alcanza un nivel de precisión aceptable, en lugar de hacer un solo ajuste. Cada golpe representa una corrección en la predicción del modelo.

## CLASIFICACIÓN CON XGBOOST

El algoritmo se va a desarrollar sobre un DataSet (Wholesale customers data.csv) que describe un conjunto de datos sobre clientes mayoristas de algún sector de la industria alimentaria.

- El dataset tiene 440 entradas y 8 columns.
- ['Channel', 'Region', 'Fresh', 'Milk', 'Grocery', 'Frozen', 'Detergents\_Paper', 'Delicassen'].
- El Dataset no tiene valores nulos

```
1 X = df.drop('Channel', axis=1) ###DATASET PARA PREDECIR
2
3 y = df['Channel'] ### VARIABLE OBJETIVO
```

Separamos el conjunto de datos entre variable dependiente (Objetivo) y variables independientes.

```
[ ] 1 ### PREPROCESAMIENTO DE DATOS.
    2
    3 y.head() # VERIFICAMOS VARIABLE OBJETIVO
    4
    5 y[y == 2] = 0
    6
    7 y[y == 1] = 1
    8
    9 # Transformamos a binario, lo que sea 2 se vuelve 0 y lo que sea 1 se queda como 1
```

Se hace un mapeo de la variable objetivo para convertirla en binaria. (Problema de clasificación binaria)

```
[ ] 1 # import XGBoost
    2 import xgboost as xgb
    3
    4
    5 # define data_dmatrix
    6 data_dmatrix = xgb.DMatrix(data=X, label=y)
    7
    8
    9 # split X and y into training and testing sets
    10
    11 from sklearn.model_selection import train_test_split
    12
    13 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 0)
```

1. Importamos la librería necesaria para cargar el algoritmo xgb y el modulo train\_test\_split.

2. Definimos las variables de entrenamiento y testeo y dividimos las variables (dependiente e independientes) en 30% para el test y 70% para el entrenamiento.

```
1 # import XGBClassifier
2 from xgboost import XGBClassifier
3
4
5 # declare parameters
6 params = {
7     'objective': 'binary:logistic',
8     'max_depth': 4,
9     'alpha': 10,
10    'learning_rate': 1.0,
11    'n_estimators': 100
12 }
13
14
15 # instantiate the classifier
16 xgb_clf = XGBClassifier(**params)
17
18
19 # fit the classifier to the training data
20 xgb_clf.fit(X_train, y_train)
21
```

3. Importamos el modelo XGBClassifier y creamos un diccionario con los parámetros con los que vamos a trabajar los parámetros.
  - a. Objective: 'binary:logistic', indica que se trata de un problema de clasificación binaria.
  - b. max\_depth: 4, indica la máxima profundidad del árbol a trabajar. Valores mas altos permite que el modelo capture patrones mas complejos pero también aumenta el riesgo de sobreajuste.
  - c. Alpha: 10, es un método de regularización (Regularización L1) este parámetro ayuda a reducir las características menos importantes . el valor de 10 indica una penalización moderada que ayuda a la simplificación del modelo y esto ayuda a reducir el sobreajuste.
  - d. learning\_rate: 1.0, Esta es la tasa de aprendizaje controla el peso de los nuevos arboles en el modelo un valor de 1 indica que el modelo va a utilizar completamente los nuevos arboles sin reducir el impacto. Las tasas de aprendizaje mas bajas van a requerir de mas iteraciones del modelo para llegar a buenos resultados, pero esto logra generar modelos mas estables.



- e. `n_estimators`: 100, Este parámetro representa la cantidad de arboles que va a generar el modelo, este parámetro afecta la complejidad del modelo ya que mas arboles pueden mejorar el rendimiento pero por otro lado pueden también aumentar el riesgo de overfit.
- 4. Configuramos el modelo y le pasamos como un diccionario los parámetros que seteamos en la variable `params`.
- 5. Generamos el entrenamiento del modelo con el `.fit` y obtenemos el modelo para generar las predicciones.

```
XGBClassifier(alpha=10, base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=1.0, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=4, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=100, n_jobs=None,
              num_parallel_tree=None, ...)
```

```
[ ] 1 # make predictions on test data
    2
    3 y_pred = xgb_clf.predict(X_test)
```

- 6. Guardamos en la variable `y_pred` los resultados del modelo en la variable `X_test`

```
[ ] 1 # compute and print accuracy score
    2
    3 from sklearn.metrics import accuracy_score
    4
    5 print('XGBoost model accuracy score: {0:0.4f}'.format(accuracy_score(y_test, y_pred)))
```

→ XGBoost model accuracy score: 0.8788

- 7. Evaluamos el modelo teniendo como referencia la métrica precisión del modelo `accuracy`
- 8. Importamos el módulo (`accuracy_score`).
- 9. Aplicamos la métrica a las variables (`y_test`, `y_pred`) y encontramos que para este conjunto de datos el modelo explica el 87.8 % de la variable objetivo en este caso la variable 'Channel'

## Validación Cruzada k-fold utilizando XGBoost

La validación cruzada es una técnica para evaluar el rendimiento de un modelo, su objetivo es estimar como se desempeña el modelo con datos nuevos que no ha visto para reducir el riesgo de sobreajuste.

- Siempre se debe realizar validación cruzada k-fold para un modelo de XGBOOST.
- Así aseguramos que el conjunto de datos de entrenamiento se utilice también para la validación.
- Para utilizar la validación cruzada se utiliza el método `cv()`

Parámetros para la validación cruzada:

```
[ ] 1 from xgboost import cv
    2
    3 params = {"objective": "binary:logistic", "colsample_bytree": 0.3, "learning_rate": 0.1,
    4           "max_depth": 5, "alpha": 10}
    5
    6 xgb_cv = cv(dtrain=data_dmatrix, params=params, nfold=3,
    7            num_boost_round=50, early_stopping_rounds=10, metrics="auc", as_pandas=True, seed=123)
```

1. Importamos el modulo `cv` de la librería `xgboost`
2. Generamos nuevamente un diccionario de parámetros para el modelo donde
  - a. `objective: 'binary:logistic'`, indica problema de clasificación binaria
  - b. `colsample_bytree: 0.3`, este parámetro controla la cantidad de características utilizadas para entrenar cada árbol. El valor 0.3 indica que solo se utilizan el 30% de las características y esto ayuda a reducir el sobreajuste y a que el modelo sea mas robusto.
  - c. `learning_rate: 0.1`, tasa de aprendizaje. Tasas mas pequeñas implica mas iteraciones.
  - d. `max_depth: 5`, profundidad máxima del árbol. Arboles muy profundos generan sobreajuste
  - e. `alpha: 10`, parámetro de regularización.
3. Generamos parámetros para la función `cv` (Validación cruzada)
  - a. `dtrain: data_dmatrix`, es el conjunto de datos que se utilizan para entrenar y evaluar el modelo durante la validación cruzada, `data_dmatrix` es un objeto de tipo `DMatrix`. Esto es una estructura de datos optimizada para XGBoost.
  - b. `params: params`, diccionario de parámetros previamente seteados.
  - c. `nfold: 3`, cantidad de pliegues o subconjuntos en el que vamos a dividir el dataset, en cada iteración para este caso 2 divisiones se van a utilizar para el entrenamiento y 1 división para la prueba.

- d. `num_boost_round`: 50, número máximo de iteraciones o boosting, el modelo intentará generar hasta 50 árboles para mejorar el modelo.
- e. `early_stopping_rounds`: 10, Permite detener el entrenamiento de manera anticipada si no hay mejora en el rendimiento en las últimas 10 rondas. Si el AUC (área bajo la curva) deja de mejorar durante 10 rondas consecutivas, el proceso de entrenamiento se detendrá, ahorrando tiempo y evitando el sobreajuste.
- f. `metrics`: "auc", esta es la métrica utilizada para evaluar el rendimiento en cada fold. AUC (AREA UNDER THE CURVE), esta es una métrica muy común para modelos de clasificación binaria que se enfoca en medir la capacidad de un modelo de distinguir entre dos clases.
- g. `as_pandas`: True, Indica que el resultado de la validación cruzada debe devolverse como un DataFrame de pandas, lo cual facilita la visualización y el análisis de los resultados.
- h. `seed`: 123, Valor semilla que garantiza que el resultado sea reproducible. Los resultados son consistentes y cada vez que se ejecuta el código el resultado es el mismo.

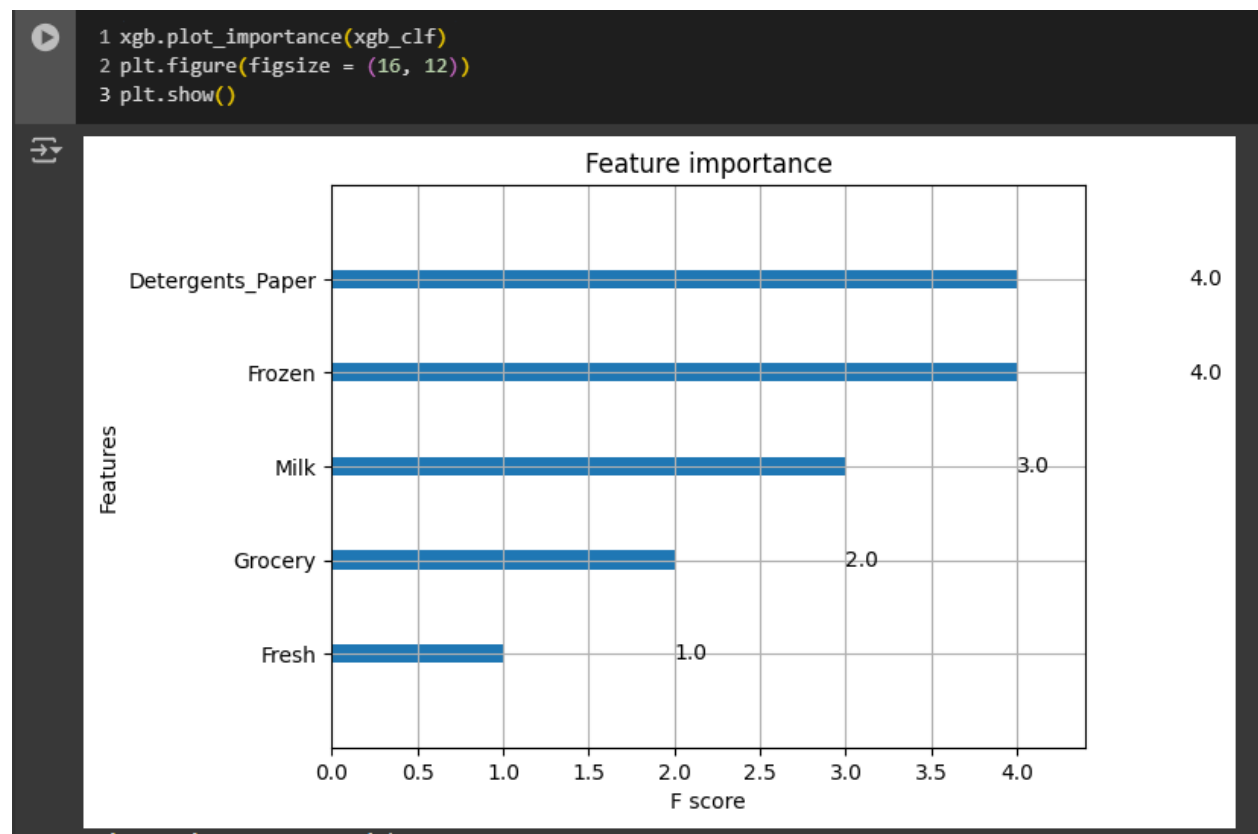
	train-auc-mean	train-auc-std	test-auc-mean	test-auc-std
0	0.668359	0.024161	0.608504	0.032623
1	0.668915	0.023947	0.608712	0.032715
2	0.929184	0.014786	0.891158	0.033113
3	0.933945	0.012181	0.890108	0.030048
4	0.958340	0.004118	0.934210	0.011683
5	0.959050	0.002962	0.930304	0.010824
6	0.961286	0.002454	0.939295	0.009608
7	0.967068	0.000861	0.942824	0.011598
8	0.966917	0.001675	0.942459	0.011385
9	0.968456	0.003024	0.944999	0.012773
10	0.968342	0.003647	0.946087	0.012948
11	0.969199	0.004456	0.946217	0.013501

## Importancia de Características con XGBoost

El modelo XGBoost tiene una forma para examinar que importancia tiene cada característica dentro del conjunto de datos original, nos ayuda a entender cuales variables tienen mas peso y contribuyen en la predicción.

La importancia de una característica se evalúa contando cuántas veces aparece en los árboles del modelo (es decir, cuántas veces se divide un nodo usando esa característica). Las características que aparecen más veces suelen ser más relevantes, ya que contribuyen más al rendimiento del modelo.

En el grafico las características se agregan por orden de importancia, esto nos puede ayudar para hacer un correcto uso de características, ver cuales son importantes para el modelo, cuales podríamos quitar para simplificar el mismo y evitar el sobreajuste.



# Segundo Ejercicio

Project Title:  Consumer Electronics Sales |  
EDA + Model 

Descripción de las etiquetas.

- **ProductID:** Identificador único para cada producto.
- **ProductCategory:** Categoría del producto de electrónica de consumo (por ejemplo, Smartphones, Laptops).
- **ProductBrand:** Marca del producto (por ejemplo, Apple, Samsung).
- **ProductPrice:** Precio del producto (\$).
- **CustomerAge:** Edad del cliente.
- **CustomerGender:** Género del cliente (0 - Masculino, 1 - Femenino).
- **PurchaseFrequency:** Número promedio de compras por año.
- **CustomerSatisfaction:** Calificación de satisfacción del cliente (1 - 5).
- **PurchaseIntent (Variable Objetivo):** Intención de compra basada en la edad del cliente, género y satisfacción.

El modelo de ML se desarrollará sobre el conjunto de datos (consumer\_electronics\_sales\_data.csv), este conjunto de datos describe el comportamiento de productos tecnológicos dentro de un ambiente de mercado con diferentes marcas, tipo de producto y por supuesto precios lo que sugiere que puede ser un conjunto de datos destinado a regresión para predicción de precios.

Revisando el código y la descripción de las variables vemos que el modelo girara entorno a la predicción de intención de compra 'PurchaseIntent' basada en la edad del cliente, genero y satisfacción.

El modelo comienza importando las librerías necesarias y cargando el dataset en Google colab. Se setean algunos parámetros de pandas para poder visualizar mas cómodo los resultados.

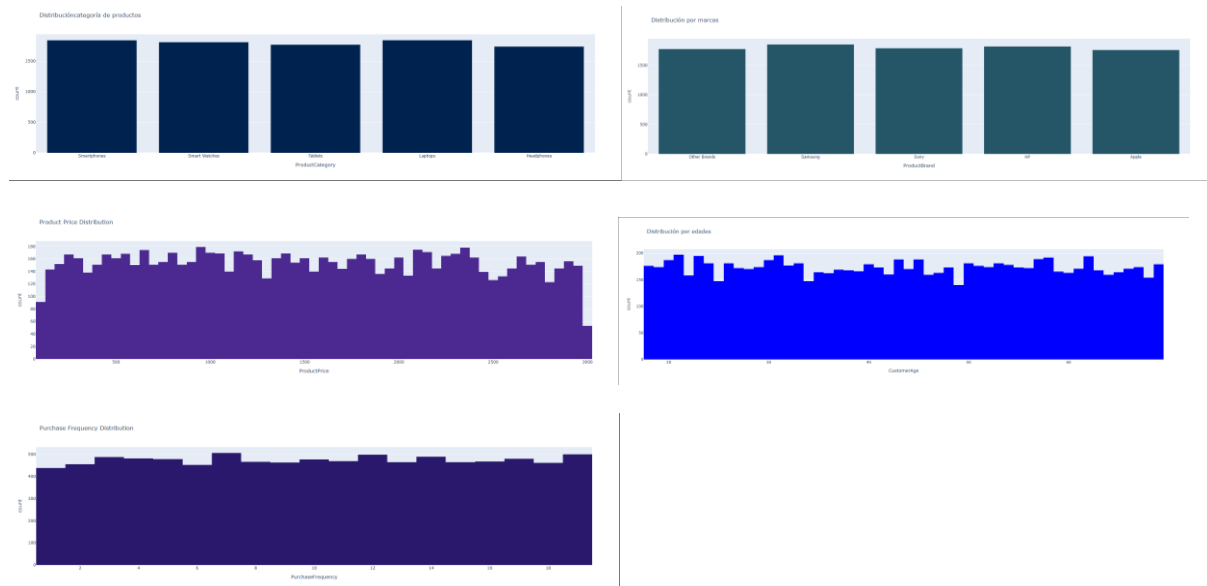
```
1 # importing libraries for data processing and visualization
2 import pandas as pd
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import seaborn as sns
6 import plotly.express as px
7
8
9 # importing libraries for machine learning
10 from sklearn.model_selection import train_test_split
11 from sklearn.linear_model import LinearRegression
12 from sklearn.ensemble import RandomForestClassifier
13 from sklearn.tree import DecisionTreeClassifier
14 from sklearn.ensemble import GradientBoostingClassifier
15 from catboost import CatBoostClassifier
16 from xgboost import XGBClassifier
17
18 from sklearn.metrics import mean_squared_error, r2_score
19 from sklearn.preprocessing import StandardScaler
20 from sklearn.preprocessing import LabelEncoder
21
22
23 # ignore warnings
24 import warnings
25 warnings.filterwarnings('ignore')
26
27 # full display of columns and rows
28 pd.set_option('display.max_columns', None)
29 pd.set_option('display.max_rows', None)
30 pd.set_option('display.width', 1000)
```

```
[21] 1 import numpy as np # linear algebra
      2 import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
      3 import matplotlib.pyplot as plt # for plotting facilities
      4
      5 #from google.colab import drive
      6 #drive.mount('/content/drive')
      7
      8 path = '/content/drive/MyDrive/Colab Notebooks/Archivos_clase/'
      9
     10 filename = 'consumer_electronics_sales_data.csv'
     11
     12
     13 df = pd.read_csv(f'{path}{filename}')
     14
```

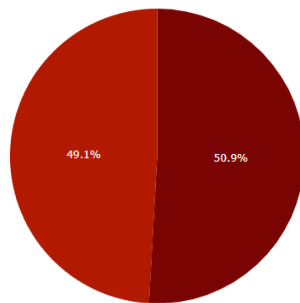
- Visualizamos el dataset cargado para entender su dimensionalidad y la unidad básica de descripción categórica y/o numérica para cada etiqueta.

	ProductID	ProductCategory	ProductBrand	ProductPrice	CustomerAge	CustomerGender	PurchaseFrequency	CustomerSatisfaction	PurchaseIntent
0	5874	Smartphones	Other Brands	312.949668	18	0	2	1	0
1	5875	Smart Watches	Samsung	980.389404	35	1	7	2	1
2	5876	Tablets	Samsung	2606.718293	63	0	1	5	1
3	5877	Smartphones	Samsung	870.395450	63	1	10	3	1
4	5878	Tablets	Sony	1798.955875	57	0	17	3	0
5	5879	Smartphones	Samsung	373.148325	37	1	8	1	1
6	5880	Smartphones	Samsung	2330.036775	26	1	5	5	1
7	5881	Smartphones	HP	780.101494	35	0	12	5	1
8	5882	Laptops	Other Brands	2264.561583	19	1	3	4	1
9	5883	Laptops	HP	1001.624006	66	1	8	4	1

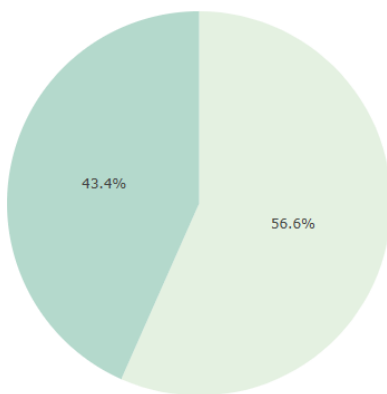
- El código muestra un proceso de análisis exploratorio previo al preprocesamiento de variables, entendimiento de las distribuciones y comportamiento de cada variable para verificar la importancia de la variable objetivo.
- Dentro del análisis exploratorio se pueden ver distintos gráficos de distribución que permiten entender el comportamiento de las variables categóricas. Categoría de producto, marcas, precio de producto, distribución por edades.



- Distribución por género en la intención de compra



- Distribución intento de compra.



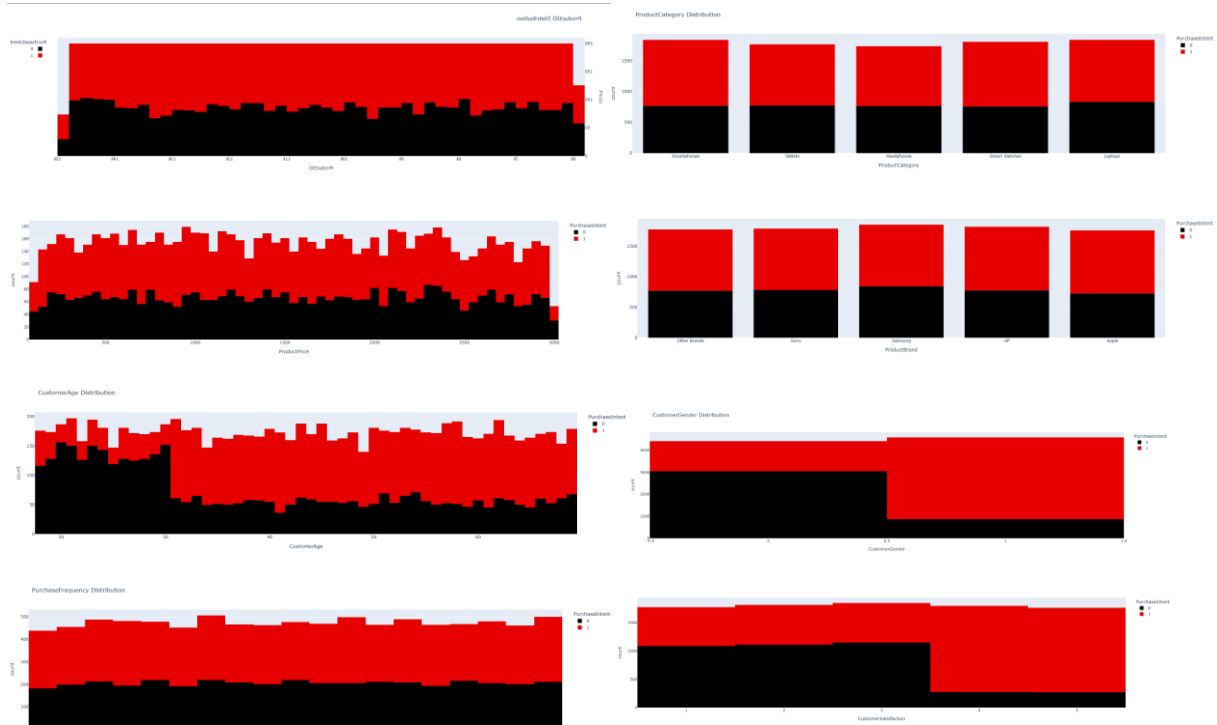


- Posterior a esto se mira el comportamiento de las variables respecto a la categoría intento de compra.

```
1 for col in df.columns:
2     plt = px.histogram(df, x=col, color='PurchaseIntent', title=f"{col} Distribution", color_discrete_sequence=px.colors.sequential.Blackbody)
3     plt.show()
```

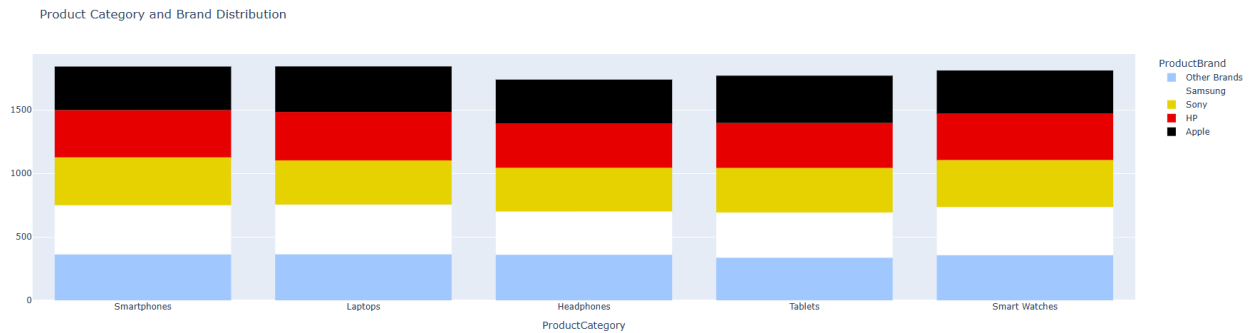
Este código recorre todas las etiquetas y pinta las intenciones de de compra por color.

Imágenes de referencia de la exploración.

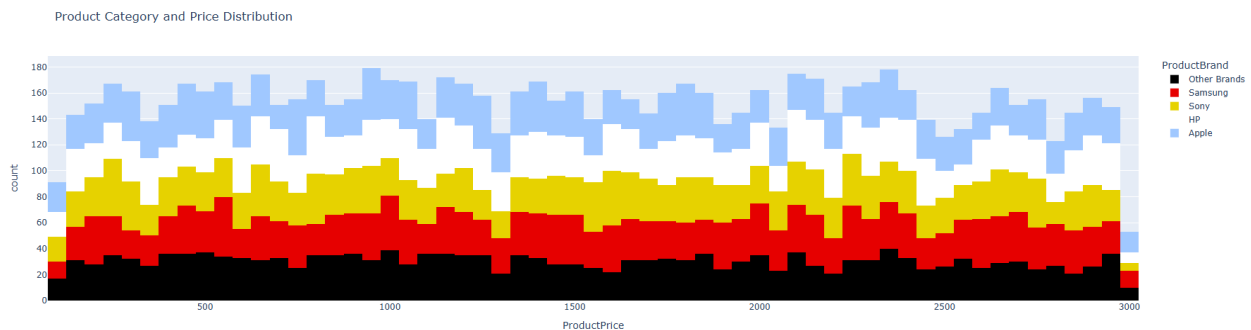


- Ahora observamos el comportamiento de las variables referente a la marca (Brand)

```
1 plt = px.histogram(df, x = 'ProductCategory', color='ProductBrand', title="Product Category and Brand Distribution", color_discrete_sequence=px.colors.sequential.Blackbody_r)
2 plt.show() #Relación de producto con la marca
```



```
1 plt = px.histogram(df, x = 'ProductPrice', color='ProductBrand', title="Product Category and Price Distribution", color_discrete_sequence=px.colors.sequential.Blackbody_r)
2 plt.show()
```



- Codificamos la data categórica a numérica no binaria como un proceso de estandarizaion de datos con el método LabelEncoder()

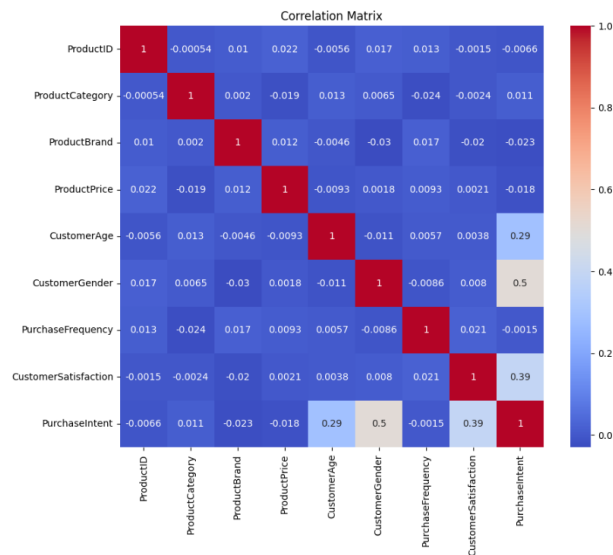
```
1 le = LabelEncoder()
2 df['ProductCategory'] = le.fit_transform(df['ProductCategory'])
3 df['ProductBrand'] = le.fit_transform(df['ProductBrand'])
```

Y verificamos que efectivamente nuestro df tiene codificación numérica.

	ProductID	ProductCategory	ProductBrand	ProductPrice	CustomerAge	CustomerGender	PurchaseFrequency	CustomerSatisfaction	PurchaseIntent
0	5874	3	2	312.949668	18	0	2	1	0
1	5875	2	3	980.389404	35	1	7	2	1
2	5876	4	3	2606.718293	63	0	1	5	1
3	5877	3	3	870.395450	63	1	10	3	1
4	5878	4	4	1798.955875	57	0	17	3	0
5	5879	3	3	373.148325	37	1	8	1	1

- Se genera una matriz de correlación.

```
1 import matplotlib.pyplot as plt
2 # correlation matrix and heatmap
3 corr = df.corr()
4 plt.figure(figsize=(10, 8))
5 sns.heatmap(corr, annot=True, cmap='coolwarm')
6 plt.title('Correlation Matrix')
7 plt.show()
```



Llama la atención como solo las variables edad y genero son las que podrían tener una incidencia mayor para la redición por lo tanto todas las variables entrarían en el modelo inicial sin hacer ingeniería de variable.

## Generación del modelo ML

```
[41] 1 X = df.drop(['ProductID', 'PurchaseIntent'], axis=1) # Data de entrenamiento (variables independientes)
      2 y = df['PurchaseIntent'] #Variable objetivo o variable a predecir

[42] 1 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42) # Splitear
```

1. Dividimos el dataset entre variable dependiente y variables independientes.
2. Generamos las variables de entrenamiento y test.

```
1 scaler = StandardScaler()
2 X_train = scaler.fit_transform(X_train)
3 X_test = scaler.transform(X_test)
```

3. Se aplica normalización de escala para todo el dataset de entrenamiento.

```
1 # Models
2
3 models = {
4     'Linear Regression': LinearRegression(),
5     'Random Forest': RandomForestClassifier(random_state=42),
6     'Decision Tree': DecisionTreeClassifier(random_state=42),
7     'Gradient Boosting': GradientBoostingClassifier(random_state=42),
8     'CatBoost Classifier': CatBoostClassifier(random_state=42),
9     'XGBoost': XGBClassifier(random_state=42)
10 }
```

4. Se genera un código donde el dataset de entrenamiento y variable objetivo de entrenamiento entran a un bucle de iteración donde se corren distintos modelos (Regresión lineal, random fores, árbol de decisión, gradient boosting, catboost classifier, XGBoost).

```
11
12 # Training the model
13
14 for name, model in models.items():
15     model.fit(X_train, y_train)
16     print(f"{name} trained.")
17
18 print("-----")
```

5. Generamos el entrenamiento para todos los modelos seleccionados

```

20 # Evaluate the models
21 model_scores = {}
22
23 for name, model in models.items():
24     y_pred = model.predict(X_test)
25     mse = mean_squared_error(y_test, y_pred)
26     r2 = r2_score(y_test, y_pred)
27     model_scores[name] = {
28         'MSE': mse,
29         'R2': r2
30     }
31     print(f"{name} MSE: {mse:.2f}, R2: {r2:.2f}")
32
33 # Find the best model
34 best_model_name = max(model_scores, key=lambda x: model_scores[x]['R2'])
35 best_model_score = model_scores[best_model_name]['R2']
36 print(f"Best model: {best_model_name} with R2: {best_model_score:.2f}")

```

6. Se general las predicciones y se definen las métricas de evaluacion, en este caso MSE y R2.

```

-----
Linear Regression MSE: 0.13, R2: 0.48
Random Forest MSE: 0.05, R2: 0.81
Decision Tree MSE: 0.10, R2: 0.60
Gradient Boosting MSE: 0.05, R2: 0.81
CatBoost Classifier MSE: 0.05, R2: 0.81
XGBoost MSE: 0.05, R2: 0.80
Best model: Random Forest with R2: 0.81

```

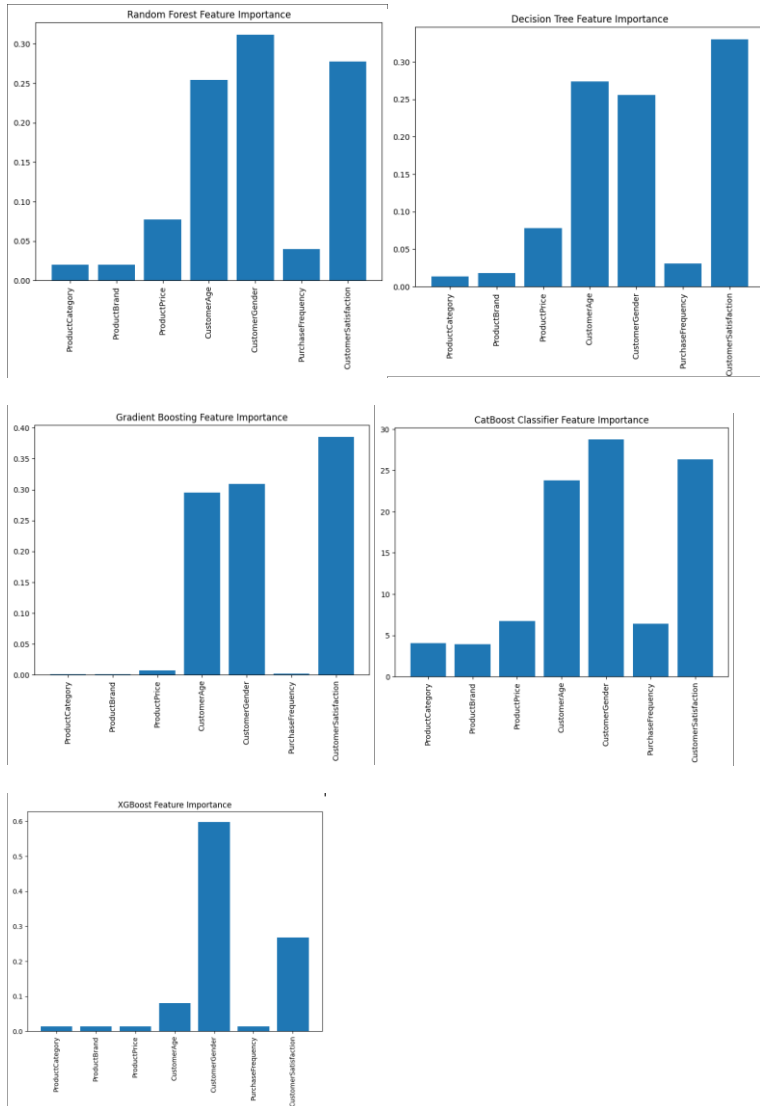
7. Se determina que la mejor métrica para el conjunto de datos evaluado en la lista de modelos fue Random Fores, con un R2 del 81% un valor bastante aceptable.

```

[ ] 1 import matplotlib.pyplot as plt
    2
    3 for name, model in models.items():
    4     if hasattr(model, 'feature_importances_'):
    5         feature_importances = model.feature_importances_
    6         plt.figure(figsize=(8, 6))
    7         plt.bar(range(len(feature_importances)), feature_importances)
    8         plt.xticks(range(len(feature_importances)), X.columns, rotation=90)
    9         plt.title(f"{name} Feature Importance")
   10         plt.show()

```

8. Se genera el feature importance. Donde se ve que para los 5 modelos las variables que explican la varianza del modelo son 'CustomerAge', 'CustomerGender' y 'CustomerSatisfaction'.



# REGRESION CON XGBOOST

Para este ultimo dataset se va a desarrollar un modelo de regresión que apunto a predecir el precio de los diamantes según las características listadas en el conjunto de datos.

En este modelo se van a utilizar algunas estrategias para tunig, con la finalidad de explorar las configuraciones de los hiperparametros basados en GBM (Gradient boosting machine).

- Los hiperparametros básicos que se pueden modificar inicialmente en el modelo son el aumento de numero de arboles y la tasa de aprendizaje. Aumentar el número de árboles puede mejorar el rendimiento del modelo, pero si se hace sin ajustar la tasa de aprendizaje, el modelo puede sobre ajustarse. La parada temprana ayuda a detener el entrenamiento si el rendimiento en los datos de validación deja de mejorar, evitando el sobreajuste.
- hiperparámetros de regularización. Estos incluyen parámetros como alpha (regulación L1) y lambda (regulación L2), que ayudan a reducir la complejidad del modelo y prevenir el sobreajuste.
- Reevaluación de la tasa de aprendizaje: Cuando se encuentran configuraciones de hiperparámetros diferentes a las predeterminadas, es importante reajustar la tasa de aprendizaje para asegurarse de que el modelo esté convergiendo correctamente. La tasa de aprendizaje determina el impacto de cada árbol en el modelo final, por lo que ajustarla puede ayudar a lograr una convergencia más estable y precisa.

El conjunto de datos describe el precio y otros atributos de casi 54.000 diamantes. Hay 10 atributos descritos incluyendo la variable objetivo precio.

## Descripción de las características.

### Descripción de las Características:

- **price:** Precio en dólares estadounidenses (326 — —18,823). Esta es la columna objetivo que contiene etiquetas para las características.

### Los 4 Cs de los Diamantes:

- **carat (0.2--5.01):** El quilate es el peso físico del diamante medido en quilates métricos. Un quilate equivale a 1/5 de gramo y se subdivide en 100 puntos. El peso en quilates es la calificación más objetiva de los 4Cs.
- **cut (Fair, Good, Very Good, Premium, Ideal):** Al determinar la calidad del corte, el evaluador del diamante considera la habilidad del cortador en la talla del diamante. Cuanto más preciso sea el corte del diamante, más cautivador será para la vista.
- **color (de J (peor) a D (mejor)):** El color de los diamantes de calidad gema se presenta en muchos tonos. En el rango de incoloro a amarillo claro o marrón claro. Los diamantes incoloros son los más raros. Otros colores naturales (azul, rojo, rosa, por ejemplo) se conocen como "fancy" y su clasificación de color es diferente a la de los diamantes incoloros blancos.
- **clarity (I1 (peor), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (mejor)):** Los diamantes pueden tener características internas conocidas como inclusiones o características externas conocidas como manchas. Los diamantes sin inclusiones o manchas son raros; sin embargo, la mayoría de las características solo se pueden ver con magnificación.

### Dimensiones:

- **x:** Longitud en mm (0–10.74)
- **y:** Ancho en mm (0–58.9)
- **z:** Profundidad en mm (0–31.8)

```
1 df.head()
```

	Unnamed: 0	carat	cut	color	clarity	depth	table	price	x	y	z
0	1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
1	2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
2	3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
3	4	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
4	5	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75

El shape del Dataframe es de 11 columnas y un total de 53940 instancias.

- La primera columna del set de datos es un índice que no aporta nada a la lectura de los datos por esta razón se elimina del dataframe.



```

1 #La primera columna es sólo un índice, la eliminamos
2 data = df.drop(["Unnamed: 0"], axis=1)
3 data.describe()

```

	carat	depth	table	price	x	y	z
count	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000
mean	0.797940	61.749405	57.457184	3932.799722	5.731157	5.734526	3.538734
std	0.474011	1.432621	2.234491	3989.439738	1.121761	1.142135	0.705699
min	0.200000	43.000000	43.000000	326.000000	0.000000	0.000000	0.000000
25%	0.400000	61.000000	56.000000	950.000000	4.710000	4.720000	2.910000
50%	0.700000	61.800000	57.000000	2401.000000	5.700000	5.710000	3.530000
75%	1.040000	62.500000	59.000000	5324.250000	6.540000	6.540000	4.040000
max	5.010000	79.000000	95.000000	18823.000000	10.740000	58.900000	31.800000

- Se eliminan los diamantes que tengan valores de '0' en alguna de sus dimensiones.

```

] 1 #Eliminamos los diamantes que tengan valor de 0 en algunas de sus dimensiones
2 data = data.drop(data[data["x"]==0].index)
3 data = data.drop(data[data["y"]==0].index)
4 data = data.drop(data[data["z"]==0].index)
5 data.shape

```

- Se realiza un proceso de codificación de las variables categóricas.

```

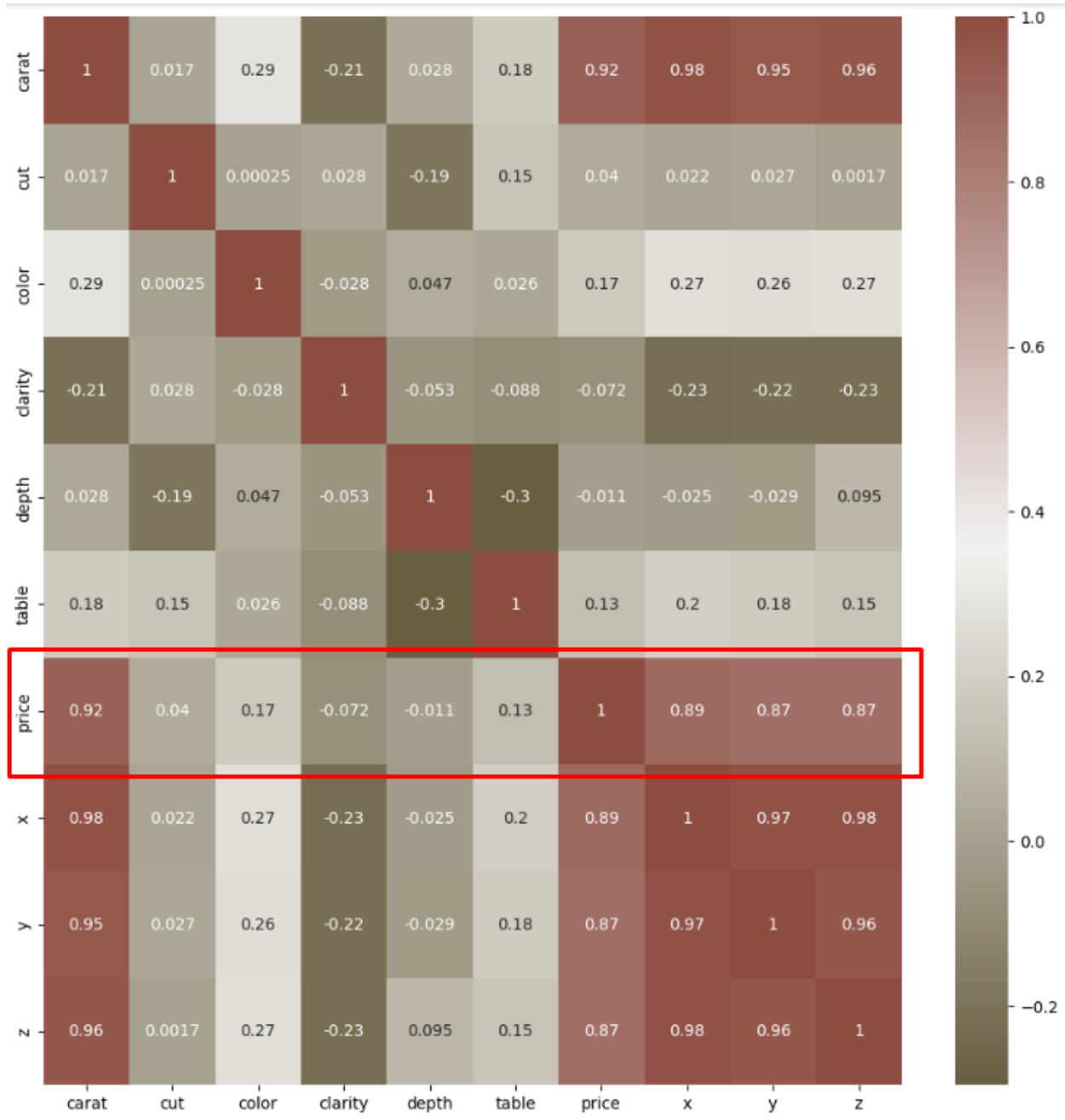
1 # Creamos una copia del dataset original para evitar modificar los datos origina
2 label_data = data.copy()
3
4 # Aplicamos el codificador a cada columna de object_cols, recordemos que object
5 label_encoder = LabelEncoder()
6 for col in object_cols:
7     label_data[col] = label_encoder.fit_transform(label_data[col])
8 label_data.head()
9

```

Iteramos a lo largo del dataframe para codificar a valores numéricos las variables con valores categóricos.

- Generamos una matriz de correlación para determinar las variables que se relacionan con nuestra variable objetivo 'price'.

```
1 #correlation matrix
2 cmap = sns.diverging_palette(70,20,s=50, l=40, n=6,as_cmap=True)
3 corrmat= label_data.corr()
4 f, ax = plt.subplots(figsize=(12,12))
5 sns.heatmap(corrmat,cmap=cmap,annot=True, )
```



En el nivel de relación de nuestra variable objetivo, las variables que mas correlación tienen es carat que es el valor en kilates del diamante, y las variables x, y, z que contemplan las medidas del diamante y que por obvias razones son el factor critico para la forma y el valor de pureza de la estructura del mineral.

## Construcción del modelo

1. Configuración de características y variable objetivo.
2. Construir una tubería (pipeline) de escalado estándar y un modelo para cinco diferentes regresores.
3. Ajustar todos los modelos con los datos de entrenamiento.
4. Obtener la media de la validación cruzada en el conjunto de entrenamiento para todos los modelos en términos de error cuadrático medio negativo.
5. Elegir el modelo con la mejor puntuación de validación cruzada.
6. Ajustar el mejor modelo en el conjunto de entrenamiento y obtener los resultados.
7. Codificación de variables categóricas.

### 1. Construcción del Modelo

```
1 # Assigning the featur as X and trarget as y
2 X= label_data.drop(["price"],axis =1)
3 y= label_data["price"]
4 X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.25, random_state=7)
5
```

- Dividimos el dataset entre variables independientes y variable dependiente.
- Definimos las variables de entrenamiento y test, y dividimos nuestro dataset en 25% para el test y 75% para el entrenamiento.

## 2. Construcción de pipeline

- En este punto se construyen unas pipe lines de series para iterar los diferentes modelos en una lista .
- En este caso se construirán para aplicar los modelos de LinearRegression(), DecisionTreeRegressor(), RandomForestRegressor(), KNeighborsRegressor(),XGBRegressor()

```
1 # Building pipelines of standard scaler and model for varios regressors.
2
3 pipeline_lr=Pipeline([("scalar1",StandardScaler()),
4 | | | | | | | | | | ("lr_classifier",LinearRegression())])
5
6 pipeline_dt=Pipeline([("scalar2",StandardScaler()),
7 | | | | | | | | | | ("dt_classifier",DecisionTreeRegressor())])
8
9 pipeline_rf=Pipeline([("scalar3",StandardScaler()),
10 | | | | | | | | | | ("rf_classifier",RandomForestRegressor())]) ### Aquí vemos el random Forest.
11
12
13 pipeline_kn=Pipeline([("scalar4",StandardScaler()),
14 | | | | | | | | | | ("rf_classifier",KNeighborsRegressor())])
15
16
17 pipeline_xgb=Pipeline([("scalar5",StandardScaler()),
18 | | | | | | | | | | ("rf_classifier",XGBRegressor())]) ### Aquí vemos el XGboost
19
```

- Creamos una lista con las variables del pipe line.

```
19
20 # List of all the pipelines
21 pipelines = [pipeline_lr, pipeline_dt, pipeline_rf, pipeline_kn, pipeline_xgb]
22
```

## 3. A través de un ciclo se entrenan todos los modelos con el pipeline.

```
25
26 # Fit the pipelines
27 for pipe in pipelines:
28 | | pipe.fit(X_train, y_train)
```

## 4. Obtener la media de la validación cruzada en el conjunto de entrenamiento para todos los modelos en términos de error cuadrático medio negativo.

```
[ ] 1 cv_results_rms = []
    2 for i, model in enumerate(pipelines):
    3 | | cv_score = cross_val_score(model, X_train,y_train,scoring="neg_root_mean_squared_error", cv=10)
    4 | | cv_results_rms.append(cv_score)
    5 | | print("%s: %f " % (pipe_dict[i], cv_score.mean()))
```

- Se crea una lista vacía llamada `cv_results_rms` que almacenará los resultados de la validación cruzada para cada modelo.
- Este bucle recorre cada modelo en `pipelines` (una lista de modelos predefinida) y su índice `i`. `pipelines` contiene los modelos de regresión que se desean evaluar.
- `cross_val_score` realiza una validación cruzada de 10 folds (`cv=10`) para cada modelo en `X_train` e `y_train` (el conjunto de datos de entrenamiento).
- `scoring="neg_root_mean_squared_error"` especifica que se usará el error cuadrático medio de la raíz negativa (negative RMSE) como métrica. Este valor negativo es una convención de `scikit-learn`, que devuelve valores negativos para las métricas de error con el fin de que, al maximizar la métrica, minimicemos el error.
- `cv_results_rms.append(cv_score)` agrega los valores de `cv_score` para cada modelo a la lista `cv_results_rms`.
- `print("%s: %f" % (pipe_dict[i], cv_score.mean()))` imprime el nombre del modelo (`pipe_dict[i]`) y el promedio de RMSE entre los folds de validación cruzada (`cv_score.mean()`). Como el RMSE es negativo (por el ajuste en `scoring`), el valor mostrado también es negativo.

5. Elegir el modelo con la mejor puntuación de validación cruzada.

```
LinearRegression: -1362.991893
DecisionTree: -754.914061
RandomForest: -551.650474
KNeighbors: -821.085886
XGBRegressor: -549.985803
```

Los resultados de cada modelo se muestran con su promedio de RMSE negativo:

- LinearRegression: -1362.991893
- DecisionTree: -754.914061
- RandomForest: -551.650474
- KNeighbors: -821.085886
- XGBRegressor: -549.985803

El Root Mean Squared Error (RMSE) mide el promedio de la magnitud de los errores en las predicciones del modelo, siendo más sensible a los errores grandes (debido a que eleva al cuadrado las diferencias). Cuanto menor sea el valor del RMSE, mejor es el rendimiento del modelo, ya que indica que las predicciones están más cerca de los valores reales.

- XGBRegressor y RandomForest tienen los valores de RMSE más bajos, fueron los modelos con el mejor puntaje en la validación cruzada.
- LinearRegression tuvo el mayor RMSE, lo que sugiere que sus predicciones tienen un mayor error promedio respecto a los valores reales en comparación con los otros modelos.


6. Ajustar el mejor modelo en el conjunto de entrenamiento y obtener los resultados.

Generamos la predicción con el modelo elegido.

```
[ ] 1 # Model prediction on test data
    2 pred = pipeline_xgb.predict(X_test)
```

- Se definen las métricas de evaluación y se obtienen los resultados

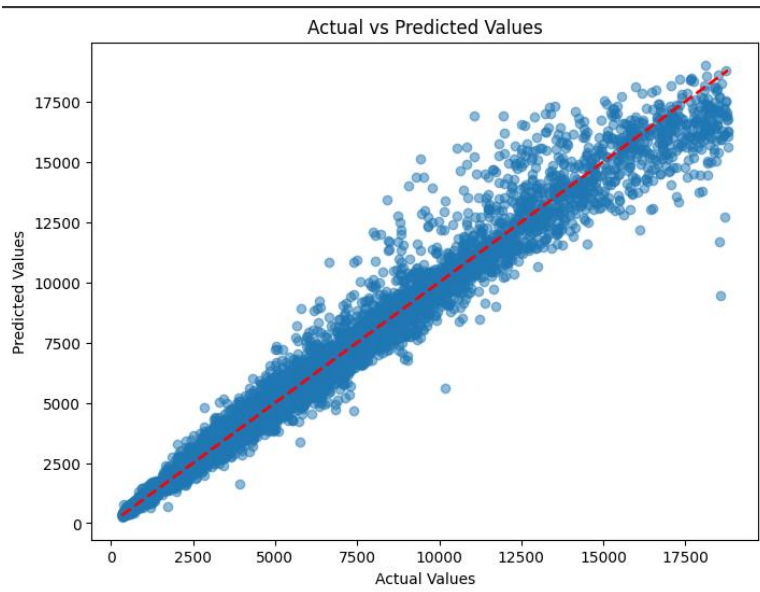
```
1 # Model Evaluation
2 print("R^2:", metrics.r2_score(y_test, pred))
3 print("Adjusted R^2:", 1 - (1 - metrics.r2_score(y_test, pred)) * (len(y_test) - 1) / (len(y_test) - X_test.shape[1] - 1))
4 print("MAE:", metrics.mean_absolute_error(y_test, pred))
5 print("MSE:", metrics.mean_squared_error(y_test, pred))
6 print("RMSE:", np.sqrt(metrics.mean_squared_error(y_test, pred)))
```

 R^2: 0.9807432889938354  
 Adjusted R^2: 0.9807304225944995  
 MAE: 275.9039350515303  
 MSE: 306372.1540923359  
 RMSE: 553.5089467139044

- La métrica R2 muestra que el modelo XGB explica el 98% de la variación del conjunto de datos para la variable dependiente precio en las respectivas variables independientes.
- El MAE es el error absoluto medio y mide la diferencia promedio entre las predicciones y los valores reales sin tener en cuenta la dirección del error. Un MAE de 275 significa que, en promedio, el modelo se desvía por 275 unidades del valor real.
- El RMSE es la raíz cuadrada del MSE y proporciona una medida del error en la misma escala que los datos originales. Un RMSE de 553 indica que, en promedio, el modelo se desvía por esa cantidad de unidades en comparación con los valores reales.

- Generamos una grafica para comparar los valores reales en el eje x y los valores predichos en el eje y .

```
1 import matplotlib.pyplot as plt
2
3 # Scatter plot for actual vs predicted values
4 plt.figure(figsize=(8, 6))
5 plt.scatter(y_test, pred, alpha=0.5)
6 plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color='red', linestyle='--', linewidth=2)
7 plt.xlabel('Actual Values')
8 plt.ylabel('Predicted Values')
9 plt.title('Actual vs Predicted Values')
10 plt.show()
11
```



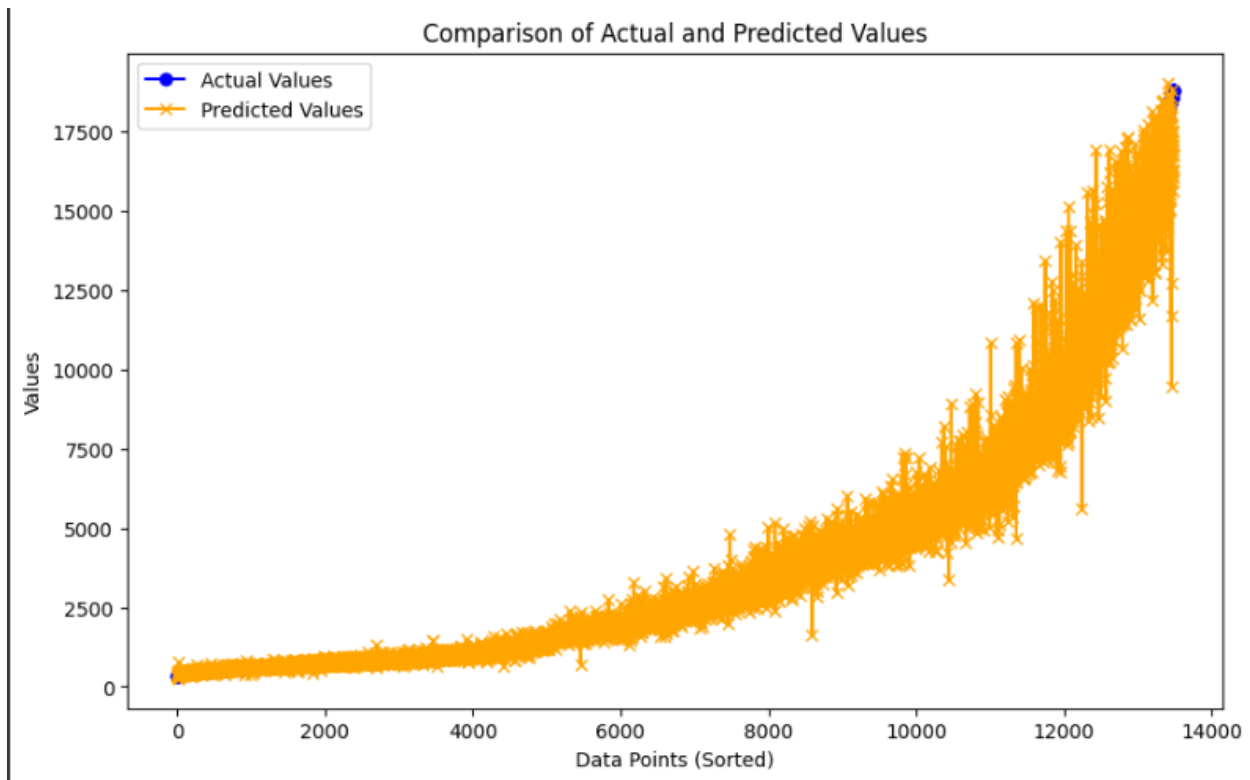
- La mayoría de los puntos están muy cerca de la línea roja, lo que indica que los valores predichos por el modelo están muy alineados con los valores reales.
- se pueden observar algunos puntos que están un poco más alejados. Estos representan casos en los que el modelo no predice con tanta precisión.
- La alineación de los puntos en torno a la línea indica que el modelo captura bien la relación lineal (o casi lineal) entre las características y la variable objetivo. Esto refuerza que el modelo está generalizando correctamente y no solo está ajustándose al conjunto de entrenamiento.

- Se genera una grafica para comprar los valores reales y los predichos en función del modelo de manera ordenada, mostrando las diferencias a lo largo de los datos.

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 # Sorting the values for a better visual comparison
5 y_test_sorted, pred_sorted = zip(*sorted(zip(y_test, pred)))
6
7 # Line plot for actual vs predicted values
8 plt.figure(figsize=(10, 6))
9 plt.plot(y_test_sorted, label='Actual Values', color='blue', marker='o')
10 plt.plot(pred_sorted, label='Predicted Values', color='orange', marker='x')
11 plt.xlabel('Data Points (Sorted)')
12 plt.ylabel('Values')
13 plt.title('Comparison of Actual and Predicted Values')
14 plt.legend()
15 plt.show()
16

```



- se utiliza zip y sorted para ordenar los valores reales (y\_test) y los valores predichos (pred). Esto permite una visualización más clara de la tendencia general, al alinear los datos en orden ascendente.



- plt.plot se usa para trazar los valores reales (y\_test\_sorted) en color azul y los valores predichos (pred\_sorted) en color naranja. Cada punto está marcado con un símbolo (círculo para los valores reales y "x" para los predichos) para facilitar la comparación visual.
- La línea de los valores predichos (naranja) sigue de cerca la línea de los valores reales (azul), lo cual es una buena señal de que el modelo predice bien la mayoría de los puntos. La cercanía entre ambas líneas indica que el modelo está capturando correctamente la tendencia de los datos.
- En los valores más bajos, las líneas están muy cerca entre sí, indicando que el modelo predice con bastante precisión en este rango. Esto es un buen indicador de que el modelo generaliza bien en el rango inferior de los datos.
- La tendencia general de crecimiento de los valores predichos y reales es muy similar, lo que sugiere que el modelo captura correctamente la relación entre las variables predictoras y la variable objetivo.

Notebook desarrollado en el documento.

<https://colab.research.google.com/drive/1Jn5pcfTRNeiYNl3cXK4kcLR-8S2iOned#scrollTo=h0QhhjZvnhtw&uniqifier=3>