

Proyecto Aurelion - IA Fundamentals

IBM - Guayerd 2025

Spring 3 - Implementación de Modelos de Machine Learning para análisis predictivo de ventas

Autor: Grupo 6

Fecha: 23 de noviembre de 2025

Dataset: df_master.xlsx

Objetivo del Proyecto

Desarrollar modelos de machine learning para predecir: 1. **Monto final de ventas** (regresión) 2. **Rango de edad de clientes** (clasificación) 3. **Categoría de producto más probable** (clasificación) 4. **Es venta premium** (clasificación) 5. **Segmentación de clientes** (k-means)

Utilizaremos el dataset `df_master.xlsx` trabajado en Spring 2, contiene información completa de transacciones de ventas retail.

Importación de Librerías

Importamos todas las librerías necesarias para el análisis y modelado:

```
# Importar librerías necesarias
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.linear_model import LinearRegression, LogisticRegression
```

```

from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsRegressor, KNeighborsClassifier
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, f1_score
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score
import warnings
warnings.filterwarnings('ignore')

# Configuración para visualización
plt.style.use('default')
sns.set_palette('husl')
plt.rcParams['figure.figsize'] = (12, 8)
plt.rcParams['font.size'] = 10

```

Carga y Exploración del Dataset

Cargamos el dataset y realizamos una exploración inicial de los datos:

```

# Cargar el dataset
#Utilizar las rutas de acuerdo a su entorno
ruta_spring_2 = 'C:\\\\Users\\\\KEILY\\\\Downloads\\\\Music\\\\IBM-GUAYERD\\\\Proyecto-Aurelion-IBM-Guayaquil\\\\'
ruta_spring_3 = 'C:\\\\Users\\\\KEILY\\\\Downloads\\\\Music\\\\IBM-GUAYERD\\\\Proyecto-Aurelion-IBM-Guayaquil\\\\'
ruta_reportes = 'C:\\\\Users\\\\KEILY\\\\Downloads\\\\Music\\\\IBM-GUAYERD\\\\Proyecto-Aurelion-IBM-Guayaquil\\\\'

df_master = pd.read_excel(ruta_spring_2 + '/df_master.xlsx')

# Información básica del dataset
print("== INFORMACIÓN DEL DATASET ==")
print(f"Dimensiones: {df_master.shape}")
print(f"Columnas: {df_master.columns.tolist()}")
print(f"\nPrimeras 5 filas:")
print(df_master.head())

# Verificar valores faltantes
print(f"\nValores faltantes por columna:")
print(df_master.isnull().sum().sum(), "valores faltantes en total")

```

==== INFORMACIÓN DEL DATASET ====

Dimensiones: (2016, 38)

Columnas: ['id_venta', 'id_producto', 'nombre_producto_x', 'cantidad', 'precio_unitario_x',

Primeras 5 filas:

	id_venta	id_producto	nombre_producto_x	cantidad	precio_unitario_x	\
0	1	90	Toallas Húmedas x50	1	29.02	
1	2	82	Aceitunas Negras 200g	5	23.94	
2	2	39	Helado Vainilla 1L	5	4.69	
3	2	70	Fernet 750ml	2	40.61	
4	2	22	Medialunas de Manteca	1	20.69	

	importe	descuento_aplicado_pct	subtotal	nombre_producto_y	\
0	29.02		0	Toallas Húmedas x50	
1	119.70		0	Aceitunas Negras 200g	
2	23.45		0	Helado Vainilla 1L	
3	81.22		0	Fernet 750ml	
4	20.69		0	Medialunas de Manteca	

	categoria	...	edad_rango	activo_como_cliente	\
0	Cuidado Personal	...	41-55	True	
1	Almacén	...	26-40	True	
2	Congelados	...	26-40	True	
3	Bebidas Alcohólicas	...	26-40	True	
4	Panadería y Repostería	...	26-40	True	

	nombre_sucursal	ciudad_y	provincia	nombre_vendedor	\
0	Tienda Aurelion - Alta Gracia	Alta Gracia	Cordoba	Vendedor 11	
1	Tienda Aurelion - Río Cuarto	Rio Cuarto	Cordoba	Vendedor 12	
2	Tienda Aurelion - Río Cuarto	Rio Cuarto	Cordoba	Vendedor 12	
3	Tienda Aurelion - Río Cuarto	Rio Cuarto	Cordoba	Vendedor 12	
4	Tienda Aurelion - Río Cuarto	Rio Cuarto	Cordoba	Vendedor 12	

	id_sucursal_y	fecha_ingreso	activo	nombre_medio_pago
0	5	2022-10-12	True	efectivo
1	2	2023-03-09	True	qr
2	2	2023-03-09	True	qr
3	2	2023-03-09	True	qr
4	2	2023-03-09	True	qr

[5 rows x 38 columns]

Valores faltantes por columna:

0 valores faltantes en total

Resumen general del dataset

Categoría	Información
Filas totales	2,016
Columnas totales	38
Valores faltantes	0 (dataset completo)
Variables numéricas clave	cantidad, precio_unitario_x, importe, subtotal, monto_bruto, monto_neto, descuento_aplicado_pct
Variables categóricas clave	categoria, genero, medio_pago_original, nombre_sucursal, ciudad_x, provincia
Estado del dataset	Limpio y listo para modelado
Balance de clases	Desbalanceado → para clasificación usar SMOTE
Calidad general	Alta (sin nulos, sin problemas de formato)
Origen	Ventas + Clientes + Productos + Sucursales + Medios de pago + Vendedores

Limpieza y Optimización del DataFrame Master

- Limpieza de datos faltantes y duplicados
- Validación de la integridad del DataFrame
- Transformación de variables categóricas y numéricas
- Marcado de outliers
- Optimización mediante normalización y escalado de datos
- Verificación final del DataFrame limpio y optimizado
- Carga del DataFrame limpio para modelado

```
#Limpieza y optimización df_master
print("LIMPIEZA Y OPTIMIZACIÓN DEL DF_MASTER")
print("*"*60)

# 1. Crear copia de trabajo
df_to_refined = df_master.copy()
print(f" Dataset original (df_master) cargado desde memoria: {df_to_refined.shape}")

# === FASE 1: LIMPIEZA BÁSICA ===
```

```

print(f"\n FASE 1: LIMPIEZA BÁSICA")
print("-" * 40)

# 2. Eliminar duplicados
print(f"Antes de limpieza: {len(df_to_refined)} registros")
duplicados_anteriores = df_to_refined.duplicated().sum()
print(f"Duplicados encontrados: {duplicados_anteriores}")

# Mostrar duplicados antes de eliminar
if duplicados_anteriores > 0:
    print("\nDuplicados identificados:")
    duplicados_idx = df_to_refined[df_to_refined.duplicated(keep=False)].index
    for idx in duplicados_idx[:6]: # Mostrar primeros 6
        row = df_to_refined.loc[idx]
        # Intentar acceder a columnas que esperamos estén en df_refined
        nombre_cliente_display = row['nombre_cliente_x'] if 'nombre_cliente_x' in row else 'N/A'
        id_venta_display = row['id_venta'] if 'id_venta' in row else 'N/A'
        print(f" Fila {idx}: Venta {id_venta_display}, Cliente {str(nombre_cliente_display)}")

df_refined = df_to_refined.drop_duplicates(keep='first')
print(f"Después de limpieza: {len(df_refined)} registros")
print(f" Duplicados eliminados: {duplicados_anteriores}")

# 3. Validar reglas de negocio
print("\nValidando reglas de negocio...")
errores = []

# Asegurarse de que las columnas existen antes de la validación
if 'monto_bruto' in df_refined.columns and 'monto_neto' in df_refined.columns:
    if not (df_refined['monto_bruto'] >= df_refined['monto_neto']).all():
        errores.append("monto_bruto < monto_neto")
    if not (df_refined['monto_neto'] >= 0).all():
        errores.append("monto_neto negativo")
else:
    if 'monto_bruto' not in df_refined.columns: errores.append("Columna 'monto_bruto' no encontrada")
    if 'monto_neto' not in df_refined.columns: errores.append("Columna 'monto_neto' no encontrada")

if 'cantidad' in df_refined.columns:
    if not (df_refined['cantidad'] >= 1).all():
        errores.append("cantidad inválida")
else:
    errores.append("Columna 'cantidad' no encontrada")

```

```

if len(errores) == 0:
    print(" Todas las reglas de negocio están correctas")
else:
    print(f"Errores encontrados: {errores}")

# === FASE 2: TRANSFORMACIÓN DE VARIABLES ===
print(f"\nFASE 2: TRANSFORMACIÓN DE VARIABLES")
print("-" * 40)

# 4. Crear nuevas características transformadas
print("Creando variables transformadas...")

# 4.1. `tipo_descuento`
if 'descuento_aplicado_pct' in df_refined.columns:
    df_refined['tipo_descuento'] = df_refined['descuento_aplicado_pct'].apply(
        lambda x: 'sin_descuento' if x == 0 else f'descuento_{int(x)}'
    )
    print("Distribución de tipos de descuento:")
    print(df_refined['tipo_descuento'].value_counts())
else:
    print("Columna 'descuento_aplicado_pct' no encontrada para crear 'tipo_descuento'.") 

# 4.2. `ticket_promedio`
if 'importe' in df_refined.columns and 'cantidad' in df_refined.columns:
    df_refined['ticket_promedio'] = df_refined['importe'] / df_refined['cantidad']
    print(f" - ticket_promedio: rango ${df_refined['ticket_promedio'].min():.2f} - ${df_refined['ticket_promedio'].max():.2f}")
else:
    print("Columnas 'importe' o 'cantidad' no encontradas para crear 'ticket_promedio'.") 

# 4.3. `es_venta_premium`
if 'monto_neto' in df_refined.columns:
    df_refined['es_venta_premium'] = df_refined['monto_neto'] > df_refined['monto_neto'].quantile(0.9)
    print(f" - ventas premium: {df_refined['es_venta_premium'].sum()} registros ({df_refined['es_venta_premium'].count()} en total)")
else:
    print("Columna 'monto_neto' no encontrada para crear 'es_venta_premium'.") 

# 4.4. `ratio_precio_cantidad`
if 'precio_unitario_x' in df_refined.columns and 'cantidad' in df_refined.columns:
    df_refined['ratio_precio_cantidad'] = df_refined['precio_unitario_x'] / df_refined['cantidad']
    print(f" - ratio_precio_cantidad: rango {df_refined['ratio_precio_cantidad'].min():.2f} - {df_refined['ratio_precio_cantidad'].max():.2f}")
else:
    print("Columnas 'precio_unitario_x' o 'cantidad' no encontradas para crear 'ratio_precio_cantidad'.") 

```

```

# 4.5. `tiene_descuento`
if 'descuento_aplicado_pct' in df_refined.columns:
    df_refined['tiene_descuento'] = df_refined['descuento_aplicado_pct'] > 0
    print(f" - ventas con descuento: {df_refined['tiene_descuento'].sum()} registros ({df_refined.shape[0]} total)")
else:
    print("Columna 'descuento_aplicado_pct' no encontrada para crear 'tiene_descuento'.")

# 4.6. Otras características temporales y monetarias
print("\nVerificando y generando características temporales...")

# Generar columna de monto_final si no existe
if 'monto_final' not in df_refined.columns:
    df_refined['monto_final'] = df_refined['subtotal'] - (
        df_refined['subtotal'] * df_refined['descuento_aplicado_pct'] / 100
    )
    print("      Columna 'monto_final' generada")

# Generar columna de trimestre
if 'trimestre' not in df_refined.columns:
    df_refined['trimestre'] = df_refined['fecha'].dt.quarter
    print("      Columna 'trimestre' generada")

# === FASE 3: MARCADO DE OUTLIERS ===
print(f"\n FASE 3: MARCADO DE OUTLIERS")
print("-" * 40)

# 5. Marcar outliers para `monto_neto` usando IQR
if 'monto_neto' in df_refined.columns:
    Q1 = df_refined['monto_neto'].quantile(0.25)
    Q3 = df_refined['monto_neto'].quantile(0.75)
    IQR = Q3 - Q1
    limite_superior = Q3 + 1.5 * IQR
    df_refined['es_outlier_monto'] = df_refined['monto_neto'] > limite_superior

    print(f"Outliers marcados (método IQR) para monto_neto:")
    print(f" - Límite superior: ${limite_superior:.2f}")
    print(f" - Outliers detectados: {df_refined['es_outlier_monto'].sum()} ({df_refined['es_outlier_monto'].sum() * 100} % del total)")

    if df_refined['es_outlier_monto'].sum() > 0 and 'nombre_cliente_x' in df_refined.columns:
        outliers_info = df_refined[df_refined['es_outlier_monto']]
        print(f"\nClientes en outliers:")
        print(outliers_info['nombre_cliente_x'].value_counts())

```

```

else:
    print("Columna 'monto_neto' no encontrada para la detección de outliers.")

# === FASE 4: OPTIMIZACIÓN PARA ML ===
print(f"\n FASE 4: OPTIMIZACIÓN PARA MACHINE LEARNING")
print("-" * 40)

# 6. Generar variables de tiempo
if 'fecha' in df_refined.columns:
    df_refined['año'] = df_refined['fecha'].dt.year
    df_refined['mes'] = df_refined['fecha'].dt.month
    df_refined['dia_semana'] = df_refined['fecha'].dt.dayofweek
    df_refined['es_finde_semana'] = df_refined['dia_semana'].isin([5, 6]).astype(int)
    print(f" Variables de tiempo creadas: año, mes, dia_semana, es_finde_semana")
else:
    print("Columna 'fecha' no encontrada para crear variables de tiempo.")

# 7. Calcular `dias_desde_alta`
if 'fecha' in df_refined.columns and 'fecha_alta' in df_refined.columns:
    df_refined['dias_desde_alta'] = (df_refined['fecha'] - df_refined['fecha_alta']).dt.days
    print(f" Variable 'dias_desde_alta' creada")
else:
    print("Columnas 'fecha' o 'fecha_alta' no encontradas para crear 'dias_desde_alta'.")

# 8. Codificar variables categóricas importantes
categorias_importantes = ['genero', 'edad_rango', 'categoria', 'tipo_descuento', 'medio_pago']
for cat_col in categorias_importantes:
    if cat_col in df_refined.columns:
        le = LabelEncoder()
        df_refined[f'{cat_col}_encoded'] = le.fit_transform(df_refined[cat_col])
        print(f" '{cat_col}' codificada: {len(le.classes_)} categorías")
    else:
        print(f"Columna '{cat_col}' no encontrada para codificar.")

# === FASE 5: ESTADÍSTICAS FINALES ===
print(f"\n FASE 5: ESTADÍSTICAS FINALES")
print("-" * 40)

# 9. Imprimir resumen del df_refined final
print(f"Dataset final:")
print(f" - Registros: {len(df_refined)} (original: {len(df_to_refined)})")
print(f" - Variables: {len(df_refined.columns)} (original: {len(df_to_refined.columns)})")

```

```

print(f" - Variables numéricas: {len(df_refined.select_dtypes(include=[np.number]).columns)}")
print(f" - Variables categóricas: {len(df_refined.select_dtypes(include=['object']).columns)}")
print(f" - Variables de fecha: {len(df_refined.select_dtypes(include=['datetime64[ns]']).columns)}")

print(f"\nVariables objetivo listas para ML:")
variables_objetivo_ml = ['monto_final', 'edad_rango', 'categoria', 'es_venta_premium']
for var_ml in variables_objetivo_ml:
    if var_ml in df_refined.columns:
        if df_refined[var_ml].dtype in ['float64', 'int64'] and df_refined[var_ml].nunique() == 2:
            print(f" - '{var_ml}': REGRESIÓN (continua, rango ${df_refined[var_ml].min():.2f} - ${df_refined[var_ml].max():.2f})")
        elif df_refined[var_ml].dtype == 'bool' or (df_refined[var_ml].dtype == 'int64' and df_refined[var_ml].nunique() == 2):
            print(f" - '{var_ml}': CLASIFICACIÓN BINARIA ({df_refined[var_ml].sum()} casos positivos, {df_refined[var_ml].count() - df_refined[var_ml].sum()} negativos)")
        elif df_refined[var_ml].dtype == 'object' or df_refined[var_ml].nunique() > 2:
            print(f" - '{var_ml}': CLASIFICACIÓN ({df_refined[var_ml].nunique()} categorías)")
    else:
        print(f" - '{var_ml}' no encontrada.")

# === FASE 6: EXPORTACIÓN ===
print(f"\n FASE 6: EXPORTACIÓN DEL DATASET LIMPIO")
print("-" * 40)

# 10. Guardar el dataset limpio en la ruta corregida
output_path = ruta_spring_3+ '/df_master_refined.xlsx'
df_refined.to_excel(output_path, index=False)
print(f" Dataset limpio guardado: {output_path}")

# Resumen de transformaciones
resumen_transformaciones = {
    'registros_originales': len(df_to_refined),
    'registros_finales': len(df_refined),
    'duplicados_eliminados': len(df_to_refined) - len(df_refined),
    'variables_nuevas': len(df_refined.columns) - len(df_to_refined.columns), # Aproximado, ya que no se consideran las que se crearon
    'outliers_marcados_monto_neto': df_refined['es_outlier_monto'].sum() if 'es_outlier_monto' in df_refined,
    'ventas_premium': df_refined['es_venta_premium'].sum() if 'es_venta_premium' in df_refined,
    'ventas_con_descuento': df_refined['tiene_descuento'].sum() if 'tiene_descuento' in df_refined
}

print(f"\nRESUMEN DE TRANSFORMACIONES:")
for key, value in resumen_transformaciones.items():
    print(f" {key}: {value}")

# Verificación final de calidad

```

```

print(f"\n VERIFICACIÓN FINAL DE CALIDAD:")
print(f" - Sin valores faltantes: {df_refined.isnull().sum().sum() == 0}")
print(f" - Sin duplicados: {df_refined.duplicated().sum() == 0}")
print(f" - Todas las reglas de negocio válidas: {len(errores) == 0}")
print(f" - Variables ML optimizadas")

print(f"\n DATASET LISTO PARA MACHINE LEARNING!")
print(f" df_master_refined listo para Implementar modelos predictivos")
df_master_refined = pd.read_excel('df_master_refined.xlsx')

```

LIMPIEZA Y OPTIMIZACIÓN DEL DF_MASTER

Dataset original (df_master) cargado desde memoria: (2016, 38)

FASE 1: LIMPIEZA BÁSICA

Antes de limpieza: 2016 registros

Duplicados encontrados: 3

Duplicados identificados:

- Fila 1: Venta 2, Cliente Zoe Flores...
- Fila 203: Venta 73, Cliente Rocio Gonzalez...
- Fila 473: Venta 190, Cliente Felipe Diaz...
- Fila 475: Venta 190, Cliente Felipe Diaz...
- Fila 1954: Venta 73, Cliente Rocio Gonzalez...
- Fila 2006: Venta 2, Cliente Zoe Flores...

Después de limpieza: 2013 registros

Duplicados eliminados: 3

Validando reglas de negocio...

Todas las reglas de negocio están correctas

FASE 2: TRANSFORMACIÓN DE VARIABLES

Creando variables transformadas...

Distribución de tipos de descuento:

tipo_descuento	
sin_descuento	1706
descuento_5	181
descuento_10	86
descuento_15	40

```
Name: count, dtype: int64
- ticket_promedio: rango $2.72 - $49.82
- ventas premium: 100 registros (5.0%)
- ratio_precio_cantidad: rango 0.54 - 49.82
- ventas con descuento: 307 registros (15.3%)
```

Verificando y generando características temporales...

```
Columna 'monto_final' generada
Columna 'trimestre' generada
```

FASE 3: MARCADO DE OUTLIERS

```
Outliers marcados (método IQR) para monto_neto:
- Límite superior: $683.10
- Outliers detectados: 10 (0.5%)
```

Clientes en outliers:

```
nombre_cliente_x
Tomas Castro      5
Karina Castro     5
Name: count, dtype: int64
```

FASE 4: OPTIMIZACIÓN PARA MACHINE LEARNING

```
Variables de tiempo creadas: año, mes, dia_semana, es_finde_semana
Variable 'dias_desde_alta' creada
'genero' codificada: 3 categorías
'edad_rango' codificada: 4 categorías
'categoria' codificada: 10 categorías
'tipo_descuento' codificada: 4 categorías
'medio_pago_original' codificada: 4 categorías
```

FASE 5: ESTADÍSTICAS FINALES

Dataset final:

```
- Registros: 2013 (original: 2016)
- Variables: 56 (original: 38)
- Variables numéricas: 29
- Variables categóricas: 19
- Variables de fecha: 3
```

Variables objetivo listas para ML:

```
- 'monto_final': REGRESIÓN (continua, rango $2.72 - $248.65)
```

- 'edad_rango': CLASIFICACIÓN (4 categorías)
- 'categoria': CLASIFICACIÓN (10 categorías)
- 'es_venta_premium': CLASIFICACIÓN BINARIA (100 casos positivos)

FASE 6: EXPORTACIÓN DEL DATASET LIMPIO

Dataset limpio guardado: C:\Users\KEILY\Downloads\Music\IBM-GUAYERD\Proyecto-Aurelion-IBM-Guayerd-edit-clone\Proyecto-Aurelion-IBM-Guayerd\SPRING_3/df_master_refined.xls

RESUMEN DE TRANSFORMACIONES:

registros_originales: 2016
 registros_finales: 2013
 duplicados_eliminados: 3
 variables_nuevas: 18
 outliers_marcados_monto_neto: 10
 ventas_premium: 100
 ventas_con_descuento: 307

VERIFICACIÓN FINAL DE CALIDAD:

- Sin valores faltantes: True
- Sin duplicados: True
- Todas las reglas de negocio válidas: True
- Variables ML optimizadas

DATASET LISTO PARA MACHINE LEARNING!

df_master_refined listo para Implementar modelos predictivos

Hallazgos Clave

El dataset fue depurado exitosamente, eliminando duplicados y garantizando reglas de negocio válidas. Se crearon 18 nuevas variables, se identificaron pocos outliers y se optimizó toda la estructura para aplicaciones de Machine Learning. El resultado es un dataset robusto, completo y listo para modelado predictivo.

Categoría	Resultado
Registros finales	2013 - registros eliminados 03 duplicados
Valores faltantes	0
Columnas originales	38
Columnas finales	56 - se crearon 18 nuevas variables
Outliers detectados (monto_neto)	10 (0.5%) – 2 clientes principales

Categoría	Resultado
Ventas premium	100 (5%)
Ventas con descuento	307 (15.3%)
Distribución tipos de descuento	sin_descuento: 1706 · desc5: 181 · desc10: 86 · desc15: 40
Variables numéricas	29
Variables categóricas	19
Variables temporales	3
Variables objetivo ML	monto_final (regresión) · edad_rango (4 clases) · categoría (10 clases) · es_venta_premium
Validación de calidad	Sin nulos · Sin duplicados · Reglas OK · ML-ready
Archivo final generado	df_master_refined.xlsx

Verificación de la selección de Variables Objetivo

Revisamos y confirmamos las variables objetivo seleccionadas para cada modelo de machine learning.

```
# Usaremos el df_master_refined para el análisis de variables objetivo potenciales
df_vo = df_master_refined.copy()

print('== ANÁLISIS DETALLADO DE VARIABLES OBJETIVO POTENCIALES ==')

# 1. Análisis de variables numéricas continuas (para regresión)
print('\n1. VARIABLES NUMÉRICAS CONTINUAS (Para Regresión):')
numeric_cols = ['monto_bruto', 'monto_neto', 'precio_unitario_x', 'importe', 'subtotal', 'mon']
for col in numeric_cols:
    if col in df_vo.columns:
        print(f' - {col}: Tipo numérico continuo.')

# 2. Análisis de variables categóricas (para clasificación)
print('\n2. VARIABLES CATEGÓRICAS (Para Clasificación):')
categorical_cols = ['categoria', 'subcategoria', 'genero', 'edad_rango', 'medio_pago_original']
for col in categorical_cols:
    if col in df_vo.columns:
        print(f' - {col}: Tipo categórico con {df_vo[col].nunique()} valores únicos.')

# 3. Análisis de fechas
print('\n3. VARIABLES DE TIEMPO:')
date_cols = ['fecha', 'fecha_alta', 'fecha_ingreso']
for col in date_cols:
```

```

if col in df_vo.columns:
    if pd.api.types.is_datetime64_any_dtype(df_vo[col]):
        print(f' - {col}: Tipo fecha (Rango: {df_vo[col].min()} - {df_vo[col].max()})')
    else:
        print(f' - {col}: La columna {col} no es de tipo fecha.')

# 4. Análisis de correlaciones con variables objetivo - Mostrar solo correlaciones fuertes
print('\n4. CORRELACIONES CON VARIABLES OBJETIVO (FUERTES):')
target_vars_corr_check = ['monto_neto', 'cantidad', 'precio_unitario_x', 'monto_final']
feature_vars_corr_check = ['edad_rango', 'genero']

# Variables numéricas para correlación más allá de las predefinidas
numerical_cols_for_corr = ['monto_bruto', 'monto_neto', 'precio_unitario_x', 'importe', 'sub']
correlation_matrix_pearson = df_vo[numerical_cols_for_corr].corr(method='pearson')

strong_corr_threshold = 0.75 # Definir umbral para correlaciones fuertes

print('\nCorrelaciones de Pearson (abs > 0.75):')
strong_correlations_found = False
for i in range(len(correlation_matrix_pearson.columns)):
    for j in range(i + 1, len(correlation_matrix_pearson.columns)):
        col1 = correlation_matrix_pearson.columns[i]
        col2 = correlation_matrix_pearson.columns[j]
        corr_value = correlation_matrix_pearson.iloc[i, j]
        if abs(corr_value) >= strong_corr_threshold:
            print(f" - {col1}<15> y {col2}<15>: {corr_value:.3f}")
            strong_correlations_found = True

if not strong_correlations_found:
    print(" - No se encontraron correlaciones fuertes (abs > 0.75) entre las variables numéricas")

# Correlaciones entre variables objetivo y feature_vars_corr_check (como antes)
for target in target_vars_corr_check:
    print(f'\nCorrelaciones con {target} (relevantes con edad_rango/genero):')
    for feature in feature_vars_corr_check:
        if feature == 'genero':
            df_gen = df_vo.copy()
            df_gen['genero_num'] = df_gen['genero'].map({'F': 0, 'M': 1})
            if pd.api.types.is_numeric_dtype(df_gen['genero_num']) and pd.api.types.is_numeric_dtype(target):
                corr = df_gen[['genero_num', target]].corr().iloc[0,1]
                print(f' - {feature}<15>: {corr:.3f}')

```

```

        else:
            print(f' - {feature}<15}: No se pudo calcular la correlación (datos no numéricos)')
    elif feature == 'edad_rango':
        df_age = df_vo.copy()
        age_map = {'18-25': 1, '26-40': 2, '41-55': 3, '56+': 4}
        df_age['edad_num'] = df_age['edad_rango'].map(age_map)
        if pd.api.types.is_numeric_dtype(df_age['edad_num']) and pd.api.types.is_numeric_dtype(target):
            corr = df_age[['edad_num', target]].corr().iloc[0,1]
            print(f' - {feature}<15}: {corr:.3f}')
        else:
            print(f' - {feature}<15}: No se pudo calcular la correlación (datos no numéricos)')

print('\n==== Variables Objetivo Potenciales para Machine Learning ====')

potential_targets = ['monto_final', 'es_venta_premium', 'categoria', 'edad_rango', 'genero']

# Encabezado de la tabla para las variables objetivo
print(f"{'Variable':<20} {'Tipo de Problema':<25} {'Argumentación'}")
print(f"{'='*20} {'='*25} {'='*60}")

for var in potential_targets:
    if var not in df_vo.columns:
        print(f" - {var:<17} {'N/A':<25} {'Variable no encontrada en el DataFrame.'}")
        continue

    tipo_problema = ""
    argumentacion = ""

    if pd.api.types.is_numeric_dtype(df_vo[var]) and df_vo[var].nunique() > 2:
        tipo_problema = "Regresión"
        if var == 'monto_final':
            argumentacion = "Fundamental para predecir ingresos. Fuertemente correlacionada con la rentabilidad."
        elif var == 'monto_neto':
            argumentacion = "Valor neto de la venta, útil para análisis de rentabilidad."
        else:
            argumentacion = "Variable continua."
    elif df_vo[var].dtype == 'bool' or (pd.api.types.is_numeric_dtype(df_vo[var]) and df_vo[var].nunique() == 2):
        tipo_problema = "Clasificación Binaria"
        if var == 'es_venta_premium':
            argumentacion = "Identifica ventas de alto valor para segmentar estrategias de marketing."
        elif var == 'tiene_descuento':
            argumentacion = "Ayuda a identificar clientes que responden bien a las promociones."
    else:
        tipo_problema = "Clasificación Multiclas"
        if var == 'categoria':
            argumentacion = "Permite agrupar los datos en categorías para análisis más detallados."
```

```

        argumentacion = "Permite predecir si una transacción incluirá un descuento."
    else:
        argumentacion = "Variable binaria."
    elif pd.api.types.is_object_dtype(df_vo[var]) or df_vo[var].nunique() > 2:
        tipo_problema = "Clasificación Multiclasificación"
        if var == 'categoria':
            argumentacion = f"Con {df_vo[var].nunique()} categorías, ayuda a predecir el tipo de producto."
        elif var == 'edad_rango':
            argumentacion = f"Con {df_vo[var].nunique()} rangos de edad, útil para personalizar ofertas basadas en la edad del cliente."
        elif var == 'genero':
            argumentacion = f"Con {df_vo[var].nunique()} valores, puede ayudar a predecir patrones de compra según el género."
        else:
            argumentacion = f"Variable categórica multiclasificación con {df_vo[var].nunique()} valores únicos."
    print(f"- {var}<17} {tipo_problema}<25} {argumentacion}")

```

== ANÁLISIS DETALLADO DE VARIABLES OBJETIVO POTENCIALES ==

1. VARIABLES NUMÉRICAS CONTINUAS (Para Regresión):

- monto_bruto: Tipo numérico continuo.
- monto_neto: Tipo numérico continuo.
- precio_unitario_x: Tipo numérico continuo.
- importe: Tipo numérico continuo.
- subtotal: Tipo numérico continuo.
- monto_final: Tipo numérico continuo.
- cantidad: Tipo numérico continuo.

2. VARIABLES CATEGÓRICAS (Para Clasificación):

- categoria: Tipo categórico con 10 valores únicos.
- subcategoria: Tipo categórico con 47 valores únicos.
- genero: Tipo categórico con 3 valores únicos.
- edad_rango: Tipo categórico con 4 valores únicos.
- medio_pago_original: Tipo categórico con 4 valores únicos.

3. VARIABLES DE TIEMPO:

- fecha: Tipo fecha (Rango: 2023-01-02 00:00:00 - 2024-06-28 00:00:00)
- fecha_alta: Tipo fecha (Rango: 2023-01-01 00:00:00 - 2023-04-10 00:00:00)
- fecha_ingreso: Tipo fecha (Rango: 2022-04-06 00:00:00 - 2023-10-28 00:00:00)

4. CORRELACIONES CON VARIABLES OBJETIVO (FUERTES):

Correlaciones de Pearson (abs > 0.75):

- monto_bruto y monto_neto : 1.000

```

- importe          y subtotal      : 0.998
- importe          y monto_final   : 0.995
- subtotal        y monto_final   : 0.999

```

Correlaciones con monto_neto (relevantes con edad_rango/genero):

```

- edad_rango      : -0.017
- genero         : -0.020

```

Correlaciones con cantidad (relevantes con edad_rango/genero):

```

- edad_rango      : -0.039
- genero         : 0.002

```

Correlaciones con precio_unitario_x (relevantes con edad_rango/genero):

```

- edad_rango      : 0.015
- genero         : -0.020

```

Correlaciones con monto_final (relevantes con edad_rango/genero):

```

- edad_rango      : -0.021
- genero         : -0.015

```

== Variables Objetivo Potenciales para Machine Learning ==

Variable	Tipo de Problema	Argumentación
----------	------------------	---------------

- monto_final	Regresión	Fundamental para predecir ingresos. Fuertemente
- es_venta_premium	Clasificación Binaria	Identifica ventas de alto valor para segmentar
- categoria	Clasificación Multiclas	Con 10 categorías, ayuda a predecir el tipo de
- edad_rango	Clasificación Multiclas	Con 4 rangos de edad, útil para personalizar e
- genero	Clasificación Multiclas	Con 3 valores, puede ayudar a predecir patrones

Implementación de modelos de Machine Learning

1 MODELO: Regresión

Predicción de variable objetivo monto_final

Objetivo: Predecir el monto final de las ventas (`monto_final`) basándose en las características del cliente, producto y transacción.

Justificación: El monto final es crucial para la planificación financiera y análisis de rentabilidad.
 Poder predecirlo permite:
 - Optimizar estrategias de precios
 - Mejorar la gestión de inventario
 - Identificar patrones de compra rentables

Algoritmos Seleccionados: 1. **Regresión Lineal Múltiple** : proporciona una base sólida para entender las relaciones lineales entre las variables independientes y el monto final. 2. **Random Forest Regressor** : capaz de capturar relaciones no lineales y manejar interacciones complejas entre variables. 3. **K-Nearest Neighbors Regressor (KNN)** : utiliza la proximidad a los puntos de datos vecinos para realizar predicciones, útil para capturar patrones locales en los datos.

1.1 Preparación del set de datos: Entradas (X) y Salida (y)

```
# Preparación de datos para el modelo de regresión
print("== PREPARACIÓN DE DATOS PARA MONTO FINAL ==")

# Variable objetivo monto_final (continua) ya está en df_master_refined
# Seleccionar variables de entrada (X)

features_reg = ['cantidad', 'precio_unitario_x', 'id_cliente', 'id_sucursal_x',
                'genero', 'edad_rango', 'categoria', 'medio_pago_original']

# Crear dataset para modelado
df_reg = df_master_refined[features_reg + ['monto_final']].copy()

# Codificar variables categóricas
le_gender = LabelEncoder()
le_age = LabelEncoder()
le_category = LabelEncoder()
le_payment = LabelEncoder()

df_reg['genero_encoded'] = le_gender.fit_transform(df_reg['genero'])
df_reg['edad_rango_encoded'] = le_age.fit_transform(df_reg['edad_rango'])
df_reg['categoria_encoded'] = le_category.fit_transform(df_reg['categoria'])
df_reg['medio_pago_encoded'] = le_payment.fit_transform(df_reg['medio_pago_original'])

# Variables finales para los modelos
X_reg = df_reg[['cantidad', 'precio_unitario_x', 'genero_encoded',
                 'edad_rango_encoded', 'categoria_encoded', 'medio_pago_encoded']]
y_reg = df_reg['monto_final']

print(f"Variables de entrada (X): {X_reg.columns.tolist()}")
print(f"Variable objetivo (y): monto_final")
print(f"Dimensiones X: {X_reg.shape}")
print(f"Dimensiones y: {y_reg.shape}")
```

```

print(f"\nEstadísticas de monto_final:")
print(f"Rango: {y_reg.min():.2f} - {y_reg.max():.2f}")
print(f"Media: {y_reg.mean():.2f}")
print(f"Desv. Estándar: {y_reg.std():.2f}")

```

==== PREPARACIÓN DE DATOS PARA MONTO FINAL ===

Variables de entrada (X): ['cantidad', 'precio_unitario_x', 'genero_encoded', 'edad_rango_en

Variable objetivo (y): monto_final

Dimensiones X: (2013, 6)

Dimensiones y: (2013,)

Estadísticas de monto_final:

Rango: 2.72 - 248.65

Media: 82.09

Desv. Estándar: 53.81

1.2 Especificación de métricas de evaluación

Para Regresión utilizaremos: - **MAE (Mean Absolute Error):** Error absoluto medio: promedio de las diferencias absolutas entre las predicciones y los valores reales. - **MSE (Mean Squared Error):** Error cuadrático medio: promedio de las diferencias al cuadrado entre las predicciones y los valores reales. - **RMSE (Root Mean Square Error):** Raíz cuadrada del Error cuadrático medio : raíz cuadrada del MSE, que da una medida de error en las mismas unidades que la variable objetivo. - **R² (Coeficiente de Determinación):** Porcentaje de varianza explicada : mide qué tan bien las predicciones del modelo explican la variabilidad de los datos reales.

Para la evaluación de los modelos de regresión se tendrá en cuenta: - Mayor precisión en las predicciones -> Menor RMSE y MAE, Mayor R².

1.3 Implementación de los modelos de regresión

Se utiliza una división del 80% para entrenamiento y 20% para prueba, asegurando que ambos conjuntos representen adecuadamente la distribución del dataset original y se ejecutan en los modelos seleccionados.

```

# División train/test (80%/20%)
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(
    X_reg, y_reg, test_size=0.2, random_state=42, stratify=None
)

```

```

print(f"Conjunto de entrenamiento: {X_train_reg.shape[0]} muestras")
print(f"Conjunto de prueba: {X_test_reg.shape[0]} muestras")

# Escalado de características
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_reg)
X_test_scaled = scaler.transform(X_test_reg)

# Inicializar modelos
lr_model = LinearRegression()
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
knn_model = KNeighborsRegressor(n_neighbors=5)

# Entrenar modelos
print("\n==== ENTRENAMIENTO DE MODELOS ===")
print("1. Entrenando Regresión Lineal...")
lr_model.fit(X_train_scaled, y_train_reg)

print("2. Entrenando Random Forest Regressor...")
rf_model.fit(X_train_reg, y_train_reg) # Random Forest no requiere escalado

print("3. Entrenando K-Nearest Neighbors Regressor...")
knn_model.fit(X_train_scaled, y_train_reg)

print(" Entrenamiento completado!")

```

Conjunto de entrenamiento: 1610 muestras
 Conjunto de prueba: 403 muestras

==== ENTRENAMIENTO DE MODELOS ===
 1. Entrenando Regresión Lineal...
 2. Entrenando Random Forest Regressor...
 3. Entrenando K-Nearest Neighbors Regressor...
 Entrenamiento completado!

1.4 Predicciones y métricas calculadas

```

# Realizar predicciones
lr_pred = lr_model.predict(X_test_scaled)
rf_pred = rf_model.predict(X_test_reg)

```

```

knn_pred = knn_model.predict(X_test_scaled)

# Función para calcular métricas
def calculate_regression_metrics(y_true, y_pred, model_name):
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    mae = mean_absolute_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)

    print(f"\n==== MÉTRICAS DE EVALUACIÓN - {model_name} ===")
    print(f"RMSE: {rmse:.2f}")
    print(f"MAE: {mae:.2f}")
    print(f"R2: {r2:.3f}")

    return {'RMSE': rmse, 'MAE': mae, 'R2': r2}

# Calcular métricas para ambos modelos
lr_metrics = calculate_regression_metrics(y_test_reg, lr_pred, "Regresión Lineal")
rf_metrics = calculate_regression_metrics(y_test_reg, rf_pred, "Random Forest")
knn_metrics = calculate_regression_metrics(y_test_reg, knn_pred, "K-Nearest Neighbors")

==== MÉTRICAS DE EVALUACIÓN - Regresión Lineal ===
RMSE: 16.38
MAE: 12.87
R2: 0.899

==== MÉTRICAS DE EVALUACIÓN - Random Forest ===
RMSE: 6.62
MAE: 3.64
R2: 0.983

==== MÉTRICAS DE EVALUACIÓN - K-Nearest Neighbors ===
RMSE: 14.51
MAE: 10.98
R2: 0.921

```

1.4.1 Resultados de los modelos de Regresión

Regresión lineal muestra un buen desempeño general, K-Nearest Neighbors también ofrece un rendimiento competitivo aunque es alto en promedios, Random Forest supera ampliamente su capacidad predictiva, reduciendo el error a menos de la mitad y logrando un mayor R². Esto indica que la relación entre las variables no es lineal y que los modelos basados en árboles capturan mejor la estructura del dataset.

Métrica	Regresión Lineal	Random Forest	K-Nearest Neighbors	Mejor Modelo
RMSE	16.38	6.62	14.51	RF
MAE	12.87	3.64	10.98	RF
R²	0.899	0.983	0.921	RF

1.5 Validación cruzada de modelos de Regresión

La validación cruzada se realiza para evaluar la robustez del modelo de Regresión Lineal Múltiple. Utilizamos K-Fold Cross Validation con 5 particiones para asegurar que el modelo generalice bien a datos no vistos.

```
from sklearn.model_selection import KFold, cross_val_score

def cross_validation_regression(model, X, y, model_name):
    cv = KFold(n_splits=5, shuffle=True, random_state=42)

    mse_scores = -cross_val_score(model, X, y, cv=cv, scoring="neg_mean_squared_error")
    mae_scores = -cross_val_score(model, X, y, cv=cv, scoring="neg_mean_absolute_error")
    r2_scores = cross_val_score(model, X, y, cv=cv, scoring="r2")

    print(f"\n*** VALIDACIÓN CRUZADA (5-Fold) - {model_name} ***")
    print(f"RMSE promedio: {np.sqrt(mse_scores).mean():.3f}")
    print(f"MAE promedio: {mae_scores.mean():.3f}")
    print(f"R2 promedio: {r2_scores.mean():.3f}")

    return {
        "RMSE_CV": np.sqrt(mse_scores).mean(),
        "MAE_CV": mae_scores.mean(),
        "R2_CV": r2_scores.mean()
    }

# Ejecutar validación cruzada
```

```

lr_cv = cross_validation_regression(lr_model, X_train_scaled, y_train_reg, "Regresión Lineal")

rf_cv = cross_validation_regression(rf_model, X_train_reg, y_train_reg, "Random Forest")

knn_cv = cross_validation_regression(knn_model, X_train_scaled, y_train_reg, "K-Nearest Neighbors")

```

*** VALIDACIÓN CRUZADA (5-Fold) - Regresión Lineal

RMSE promedio: 17.136
MAE promedio: 13.153
R² promedio: 0.899

*** VALIDACIÓN CRUZADA (5-Fold) - Random Forest ***

RMSE promedio: 6.610
MAE promedio: 3.471
R² promedio: 0.985

*** VALIDACIÓN CRUZADA (5-Fold) - K-Nearest Neighbors ***

RMSE promedio: 15.789
MAE promedio: 11.673
R² promedio: 0.915

1.5.1 Resultados de la Validación Cruzada

La validación cruzada confirma la estabilidad de Random Forest, regresión lineal mantiene un desempeño estable, K-Nearest Neighbors también muestra resultados consistentes, pero Random Forest sobresale con un error muy bajo y un R² casi perfecto, indicando que el modelo no está sobreajustado y puede generalizar bien a nuevos datos.

Modelo	RMSE ↓	MAE ↓	R ² ↑
Regresión Lineal	17.136	13.153	0.899
K-Nearest Neighbors	15.789	11.673	0.915
Random Forest	6.610	3.471	0.985

1.6 Grafico de Predicciones vs Valores Reales - Modelos de Regresión

Se visualizan las predicciones de los modelos frente a los valores reales del conjunto de prueba para evaluar visualmente el desempeño del modelo. Las predicciones que se encuentran cerca de la línea diagonal indican buenas predicciones.

```

# Crear visualizaciones para el modelo de regresión
fig, axes = plt.subplots(2, 2, figsize=(15, 12))

# 1. Comparación de valores reales vs predichos - Regresión Lineal
axes[0, 0].scatter(y_test_reg, lr_pred, alpha=0.6, color='blue')
axes[0, 0].plot([y_test_reg.min(), y_test_reg.max()], [y_test_reg.min(), y_test_reg.max()])
axes[0, 0].set_xlabel('Valores Reales')
axes[0, 0].set_ylabel('Predicciones')
axes[0, 0].set_title('Regresión Lineal: Real vs Predicho')
axes[0, 0].grid(True)

# 2. Comparación de valores reales vs predichos - Random Forest
axes[0, 1].scatter(y_test_reg, rf_pred, alpha=0.6, color='green')
axes[0, 1].plot([y_test_reg.min(), y_test_reg.max()], [y_test_reg.min(), y_test_reg.max()])
axes[0, 1].set_xlabel('Valores Reales')
axes[0, 1].set_ylabel('Predicciones')
axes[0, 1].set_title('Random Forest: Real vs Predicho')
axes[0, 1].grid(True)

# 3. Distribución de errores
lr_errors = y_test_reg - lr_pred
rf_errors = y_test_reg - rf_pred

axes[1, 0].hist(lr_errors, bins=30, alpha=0.7, color='blue', label='Regresión Lineal')
axes[1, 0].hist(rf_errors, bins=30, alpha=0.7, color='green', label='Random Forest')
axes[1, 0].set_xlabel('Error (Real - Predicción)')
axes[1, 0].set_ylabel('Frecuencia')
axes[1, 0].set_title('Distribución de Errores')
axes[1, 0].legend()
axes[1, 0].grid(True)

# 4. Importancia de variables - Random Forest
feature_names = ['cantidad', 'precio_unitario', 'genero', 'edad_rango', 'categoria', 'medio_precio']
importances = rf_model.feature_importances_
indices = np.argsort(importances)[::-1]

axes[1, 1].bar(range(len(importances)), importances[indices])
axes[1, 1].set_xlabel('Variables')
axes[1, 1].set_ylabel('Importancia')
axes[1, 1].set_title('Importancia de Variables (Random Forest)')
axes[1, 1].set_xticks(range(len(importances)))
axes[1, 1].set_xticklabels([feature_names[i] for i in indices], rotation=45)

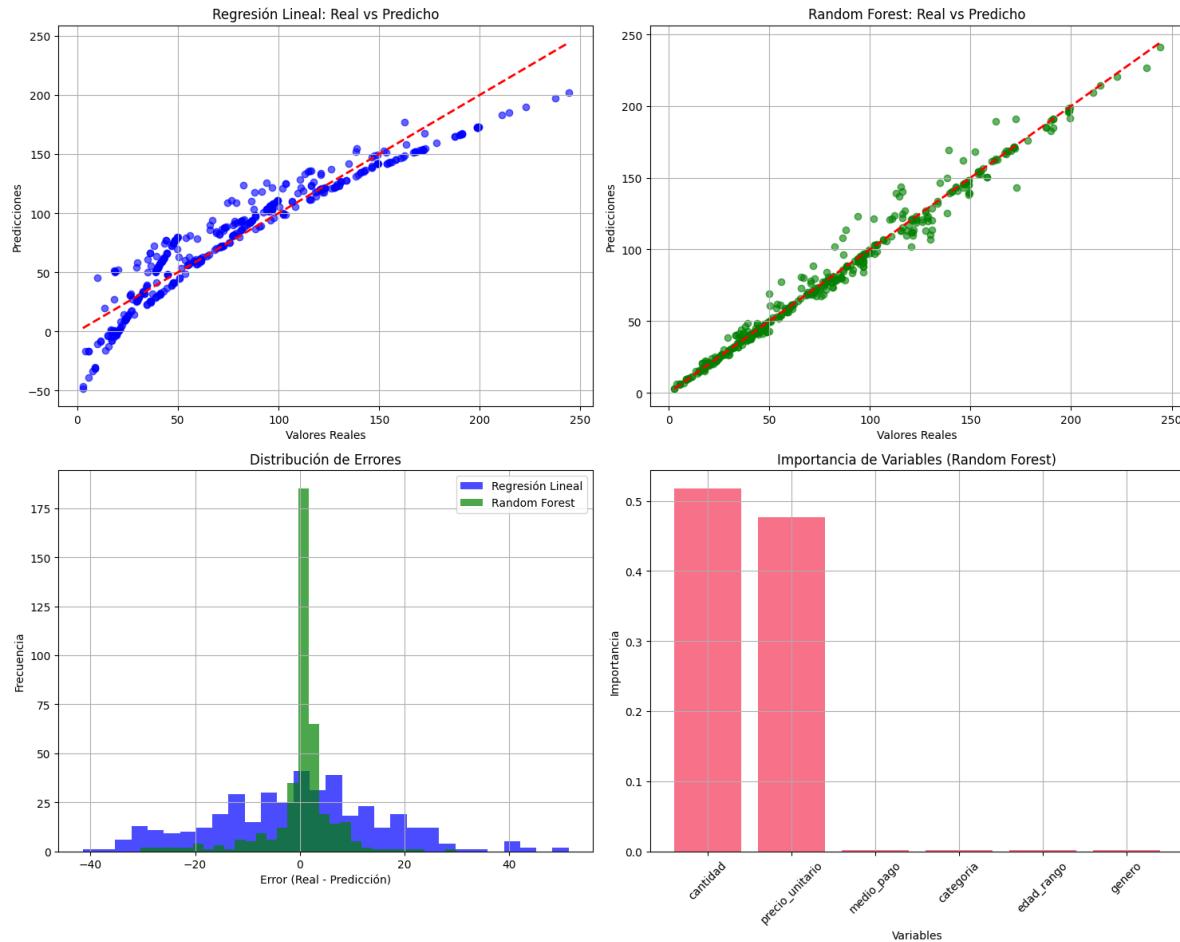
```

```

axes[1, 1].grid(True)

plt.tight_layout()
plt.show()

```



Los resultados muestran que:

- Random Forest ofrece un ajuste casi perfecto, con errores muy bajos y excelente generalización.
- Regresión lineal resulta insuficiente por la no linealidad natural del problema.
- Los gráficos confirman que el monto final depende mayormente de cantidad y precio unitario, mientras que el resto de las variables actúa como moderadoras.

2 MODELO: Clasificación

Predicción de variable objetivo edad_rango

Objetivo: Predecir el rango de edad (edad_rango) de los clientes para mejorar la segmentación de mercado.

Justificación: el rango de edad permite:

- Personalizar ofertas y campañas de marketing
- Optimizar la selección de productos por tienda
- Mejorar la segmentación de clientes para estrategias específicas

Algoritmos Seleccionados: 1. **Random Forest Classifier** : excelente para problemas de clasificación multi-clase, robusto ante overfitting. 2. **Decision Tree Classifier** : proporciona interpretabilidad directa y manejo natural de variables categóricas. 3. **LightGBM Classifier** : eficiente y rápido, ideal para grandes datasets con múltiples clases. 4. **XGBoost Classifier** : potente y escalable, maneja bien datos desequilibrados y complejos.

2.1 Preparación del set de datos: Entradas (X) y Salida (y)

```
# Preparación de datos para el modelo de clasificación - edad_rango
print("== PREPARACIÓN DE DATOS PARA EDAD_RANGO ==")

# Seleccionar variables de entrada (X)
features_clf_age = ['cantidad', 'precio_unitario_x', 'monto_neto', 'categoria', 'monto_final',
                     'medio_pago_original', 'genero', 'nombre_sucursal']

# Crear dataset para modelado
df_clf_age = df_master_refined[features_clf_age + ['edad_rango']].copy()

# Codificar variables categóricas
le_category_age = LabelEncoder()
le_payment_age = LabelEncoder()
le_store = LabelEncoder()
le_gender_age = LabelEncoder()

df_clf_age['categoria_encoded'] = le_category_age.fit_transform(df_clf_age['categoria'])
df_clf_age['medio_pago_encoded'] = le_payment_age.fit_transform(df_clf_age['medio_pago_original'])
df_clf_age['sucursal_encoded'] = le_store.fit_transform(df_clf_age['nombre_sucursal'])
df_clf_age['genero_encoded'] = le_gender_age.fit_transform(df_clf_age['genero'])
```

```

# Variables finales para el modelo
X_clf_age = df_clf_age[['cantidad', 'precio_unitario_x', 'monto_neto',
                        'categoria_encoded', 'medio_pago_encoded',
                        'sucursal_encoded', 'genero_encoded','monto_final']]
y_clf_age = df_clf_age['edad_rango']

print(f"Variables de entrada (X): {X_clf_age.columns.tolist()}")
print(f"Variable objetivo (y): edad_rango")
print(f"Distribución de clases:")
print(y_clf_age.value_counts().to_dict())

# Verificar balance de clases
class_distribution = y_clf_age.value_counts(normalize=True)
print(f"\nDistribución relativa:")
for clase, pct in class_distribution.items():
    print(f"{clase}: {pct:.1%}")

==== PREPARACIÓN DE DATOS PARA EDAD_RANGO ====
Variables de entrada (X): ['cantidad', 'precio_unitario_x', 'monto_neto', 'categoria_encoded']
Variable objetivo (y): edad_rango
Distribución de clases:
{'26-40': 1012, '41-55': 460, '18-25': 383, '56+'. 158}

Distribución relativa:
26-40: 50.3%
41-55: 22.9%
18-25: 19.0%
56+: 7.8%

```

2.2 Especificación de las Métricas de Evaluación

Para Clasificación utilizaremos: - **Accuracy:** Porcentaje de predicciones correctas - **Matriz de Confusión:** Tabla que muestra aciertos y errores en predicciones - **Precision:** Proporción de verdaderos positivos por clase - **Recall:** Proporción de casos positivos correctamente identificados - **F1-Score:** Media armónica de Precision y Recall

2.3 Implementación del modelo de clasificación

Preparación de los datos y transformación necesarias, seguido de la división del dataset en conjuntos de entrenamiento (80%) y prueba (20%). Se entrena los modelos seleccionados y se evalúan utilizando las métricas definidas.

2.3.1 Preparación, transformación y escalado de datos

```
print("== PREPARACIÓN DE DATOS PARA EDAD_RANGO ==")  
  
# Dataset base  
df_clf_age = df_master_refined[features_clf_age + ['edad_rango']].copy()  
  
# Se incluyen variables numéricas y categóricas que podrían influir en el rango de edad  
features_clf_age = [  
    'cantidad', 'precio_unitario_x', 'monto_neto', 'monto_final',  
    'categoria', 'medio_pago_original', 'genero', 'nombre_sucursal',  
    'descuento_aplicado_pct', 'año', 'mes', 'dia_semana',  
    'es_finde_semana', 'dias_desde_alta', 'ticket_promedio', 'ratio_precio_cantidad',  
    'es_venta_premium', 'tiene_descuento', 'activo_como_cliente', 'activo'  
]  
  
# Crear un DataFrame con las características y la variable objetivo 'edad_rango'  
df_clf_age = df_master_refined[features_clf_age + ['edad_rango']].copy()  
  
# Convertir las columnas booleanas a int para el procesamiento de OHE/escalado  
for col in ['es_finde_semana', 'es_venta_premium', 'tiene_descuento', 'activo_como_cliente',  
           if col in df_clf_age.columns:  
               df_clf_age[col] = df_clf_age[col].astype(int)  
  
# ----- Identificar y codificar variables categóricas para OHE -----  
categorical_cols_for_ohe_age = [  
    'categoria', 'genero', 'nombre_sucursal', 'medio_pago_original',  
    'dia_semana' # 'dia_semana' aunque es numérica, sus valores son categóricos  
]  
# variable temporal para almacenar el dataset antes de encoding  
X_temp_age = df_clf_age.drop(columns=['edad_rango']).copy()  
  
# Aplicar One-Hot Encoding a las columnas categóricas  
X_encoded_age = pd.get_dummies(X_temp_age, columns=categorical_cols_for_ohe_age, drop_first=True)  
print(f"X_encoded_age después de One-Hot Encoding: {X_encoded_age.shape[0]} filas y {X_encoded_age.shape[1]} columnas")  
  
# --- Limpiar los nombres de las columnas para evitar advertencias de LightGBM y XGBoost ---  
# LightGBM y XGBoost no manejan bien los caracteres especiales o espacios en los nombres de las columnas  
# Reemplazamos los caracteres que no son alfanuméricos por guiones bajos.  
X_encoded_age.columns = X_encoded_age.columns.str.replace('[^A-Za-z0-9_]+', '', regex=True)
```

```

# ----- Escalar las características numéricas -----
# Identificar columnas numéricas (excluyendo las que ya son binarias y no necesitan escalado)
numerical_cols_for_scaling_age = X_encoded_age.select_dtypes(include=[np.number]).columns.to_list()
# Por ahora, escalaremos todas las numéricas incluyendo las que eran booleanas para consistencia

scaler_age = StandardScaler()
X_scaled_age = X_encoded_age.copy()
X_scaled_age[numerical_cols_for_scaling_age] = scaler_age.fit_transform(X_encoded_age[numerical_cols_for_scaling_age])
print(f"Columnas numéricas de X_scaled_age escaladas usando StandardScaler.")

# ----- Codificar la variable objetivo 'edad_rango' -----
# 'edad_rango' es una variable categórica con múltiples clases (rangos de edad).
# La codificamos numéricamente para los modelos de clasificación.
le_age = LabelEncoder()
y_clf_age_encoded = le_age.fit_transform(df_clf_age['edad_rango'])
print("Variable objetivo 'edad_rango' codificada numéricamente.")

print("Distribución de clases de 'edad_rango' antes de SMOTE:", pd.Series(y_clf_age_encoded))

```

==== PREPARACIÓN DE DATOS PARA EDAD_RANGO ====
 X_encoded_age después de One-Hot Encoding: 2013 filas y 40 columnas.
 Columnas numéricas de X_scaled_age escaladas usando StandardScaler.
 Variable objetivo 'edad_rango' codificada numéricamente.
 Distribución de clases de 'edad_rango' antes de SMOTE: {0: 383, 1: 1012, 2: 460, 3: 158}

2.3.2 División en conjunto de Entrenamiento y Prueba

```

# ----- División en conjuntos de entrenamiento y prueba -----
# Usamos stratify para mantener la proporción de clases en ambos conjuntos
X_train_age, X_test_age, y_train_age, y_test_age = train_test_split(
    X_scaled_age,
    y_clf_age_encoded,
    test_size=0.2,
    random_state=42,
    stratify=y_clf_age_encoded
)

print(f"Conjunto de entrenamiento: {X_train_age.shape[0]} muestras, Conjunto de prueba: {X_te}
#
```

Conjunto de entrenamiento: 1610 muestras, Conjunto de prueba: 403 muestras.

2.3.3 Aplicar SMOTE para balancear clases en variable objetivo edad_rango

```
from imblearn.over_sampling import SMOTE

# SMOTE (Synthetic Minority Over-sampling Technique) se usa para manejar el desbalance de clase
print("Aplicando SMOTE a los datos de entrenamiento...")
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train_age, y_train_age)

print("Distribución de clases de 'edad_rango' después de SMOTE (entrenamiento):")
print(pd.Series(y_train_res).value_counts().sort_index().to_dict())
```

Aplicando SMOTE a los datos de entrenamiento...

Distribución de clases de 'edad_rango' después de SMOTE (entrenamiento):
{0: 810, 1: 810, 2: 810, 3: 810}

Preparación de datos para el modelo de clasificación - edad_rango presenta un desbalance significativo entre las clases. Para abordar este problema, se aplica la técnica de sobremuestreo SMOTE (Synthetic Minority Over-sampling Technique) en el conjunto de entrenamiento. Esto genera nuevas muestras sintéticas para las clases minoritarias, equilibrando la distribución de clases y mejorando la capacidad del modelo para aprender patrones relevantes.

Resumen de procesamiento de datos para edad_rango:

Etapa	Resultado
Filas totales	2013
Columnas después de One-Hot Encoding	40
Columnas numéricas escaladas	StandardScaler
Variable objetivo codificada	edad_rango (0–3)
Train/Test split	Train: 1610 , Test: 403

Distribución de clases antes y después de SMOTE:

Clase (edad_rango)	Antes de SMOTE	Después de SMOTE
0	383	810
1	1012	810
2	460	810

Clase (edad_rango)	Antes de SMOTE	Después de SMOTE
3	158	810
Total	2013	3240 (para entrenamiento)

2.4 Inicializar modelos de Clasificación

```

# 1. Decision Tree Classifier
dt_model = DecisionTreeClassifier(
    max_depth=10, # Profundidad máxima para evitar sobreajuste
    min_samples_split=5, # Mínimo de muestras para dividir un nodo
    min_samples_leaf=3, # Mínimo de muestras en una hoja
    random_state=42 # Para reproducibilidad
)

# 2. Random Forest Classifier
rf_model = RandomForestClassifier(
    n_estimators=200, # Número de árboles en el bosque (reducido para eficiencia)
    max_depth=12, # Profundidad máxima de cada árbol
    min_samples_split=5, # Mínimo de muestras para dividir un nodo
    min_samples_leaf=3, # Mínimo de muestras en una hoja
    # class_weight='balanced_subsample', # Descomentar si se observa desbalance después de SMOTE
    n_jobs=-1, # Usar todos los núcleos disponibles para entrenamiento paralelo
    random_state=42 # Para reproducibilidad
)

# 3. LightGBM Classifier
lgb_model = LGBMClassifier(
    num_leaves=31, # Número de hojas en los árboles
    learning_rate=0.05, # Tasa de aprendizaje
    n_estimators=200, # Número de estimadores (árboles)
    # class_weight='balanced', # Descomentar si se observa desbalance después de SMOTE
    random_state=42,
    n_jobs=-1 # Usar todos los núcleos disponibles
)

# 4. XGBoost Classifier (hiperparámetros ajustados)
xgb_model = XGBClassifier(
    n_estimators=250, # Número de estimadores
    learning_rate=0.07, # Tasa de aprendizaje
    max_depth=7, # Profundidad máxima del árbol
)

```

```

    subsample=0.8, # Fracción de muestras usadas para ajustar los estimadores individuales
    colsample_bytree=0.8, # Fracción de características usadas por árbol
    use_label_encoder=False, # Este parámetro está obsoleto y causa una advertencia en versión
    eval_metric='mlogloss', # Métrica de evaluación para clasificación multiclas
    random_state=42,
    n_jobs=-1 # Usar todos los núcleos disponibles
)

```

Parámetros clave de inicialización de modelos de Clasificación | Modelo | Parámetros ||
----- | ----- || **Decision Tree** | max_depth=10,
min_samples_split=5, random_state=42 | | **Random Forest** | n_estimators=200,
max_depth=12, random_state=42 |
| **LightGBM** | num_leaves=31, learning_rate=0.05, n_estimators=200 | | **XGBoost** |
max_depth=7, learning_rate=0.7, n_estimators=250 |

2.5 Entrenamiento y test de modelos de Clasificación

```

# Entrenar todos los modelos con los datos balanceados
print("Entrenando modelos de clasificación...")
dt_model.fit(X_train_res, y_train_res)
rf_model.fit(X_train_res, y_train_res)
lgb_model.fit(X_train_res, y_train_res)
xgb_model.fit(X_train_res, y_train_res)
print(" Modelos entrenados con datos balanceados usando SMOTE.")

```

Entrenando modelos de clasificación...
Modelos entrenados con datos balanceados usando SMOTE.

2.6 Evaluación de los modelos de Clasificación

Se define una función para evaluar los modelos de clasificación utilizando las métricas seleccionadas: Accuracy, Matriz de Confusión, Precision, Recall y F1-Macro. Esta función toma como entrada el modelo entrenado, los datos de prueba y las etiquetas reales, y devuelve un resumen completo del desempeño del modelo. * F1-Macro es especialmente útil en escenarios con clases desbalanceadas, ya que calcula el F1-Score para cada clase y luego promedia estos valores, proporcionando una visión equilibrada del rendimiento del modelo en todas las clases.

```

# ----- Función para evaluar modelos de clasificación -----
def evaluate_classification_model(model, X_test, y_test, model_name, label_encoder):
    """Evalúa un modelo de clasificación y muestra las métricas clásicas."""
    y_pred = model.predict(X_test)

    # Métricas clásicas (no macro)
    acc = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    # Matriz de confusión
    cm = confusion_matrix(y_test, y_pred)

    print(f"\n***** {model_name} *****")
    print(f"Accuracy: {acc:.3f}")
    print(f"Precision: {precision:.3f}")
    print(f"Recall: {recall:.3f}")
    print(f"F1 Score: {f1:.3f}")

    # Mostrar la matriz de confusión con etiquetas decodificadas
    labels_decoded = label_encoder.inverse_transform(sorted(np.unique(y_test)))
    df_cm = pd.DataFrame(cm,
                          index=[f"Real_{l}" for l in labels_decoded],
                          columns=[f"Pred_{l}" for l in labels_decoded])

    print("\nMatriz de Confusión:")
    display(df_cm)

    # Matriz de calor usando seaborn
    print(f"\n===== Matriz de Confusión del Modelo {model_name} =====")
    plt.figure(figsize=(5, 5))
    cmap = sns.color_palette("Greens", as_cmap=True)
    sns.heatmap(df_cm, annot=True, fmt='d', cmap=cmap, cbar=False)
    plt.title(f'Matriz de Confusión - {model_name}')
    plt.tight_layout()
    plt.show()

    return acc, precision, recall, f1, df_cm

# --- Evaluar cada modelo ---

```

```

print("\nEvaluando modelos en el conjunto de prueba (no balanceado) ...")

dt_acc, dt_prec, dt_rec, dt_f1, dt_cm = evaluate_classification_model(dt_model, X_test_age, y_test)
rf_acc, rf_prec, rf_rec, rf_f1, rf_cm = evaluate_classification_model(rf_model, X_test_age, y_test)
lgb_acc, lgb_prec, lgb_rec, lgb_f1, lgb_cm = evaluate_classification_model(lgb_model, X_test_age, y_test)
xgb_acc, xgb_prec, xgb_rec, xgb_f1, xgb_cm = evaluate_classification_model(xgb_model, X_test_age, y_test)

# Guardar métricas en variables globales
global_dt_acc, global_dt_f1 = dt_acc, dt_f1
global_rf_acc, global_rf_f1 = rf_acc, rf_f1
global_lgb_acc, global_lgb_f1 = lgb_acc, lgb_f1
global_xgb_acc, global_xgb_f1 = xgb_acc, xgb_f1

```

Evaluando modelos en el conjunto de prueba (no balanceado) ...

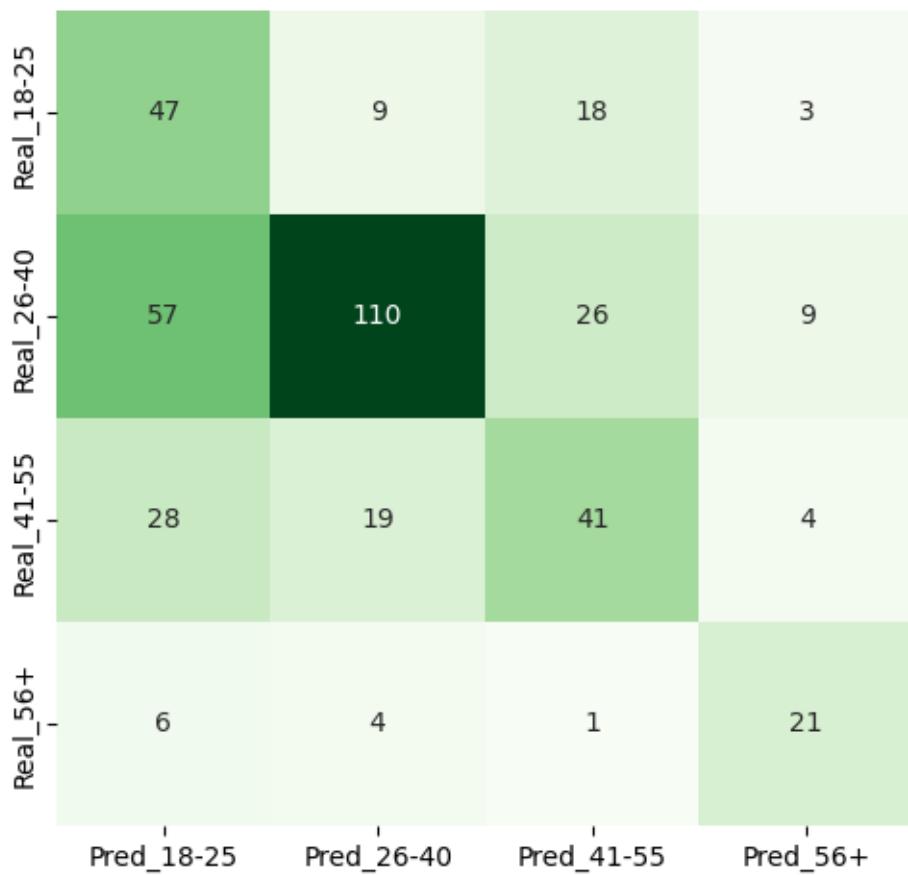
***** Decision Tree *****
Accuracy: 0.543
Precision: 0.607
Recall: 0.543
F1 Score: 0.558

Matriz de Confusión:

	Pred_18-25	Pred_26-40	Pred_41-55	Pred_56+
Real_18-25	47	9	18	3
Real_26-40	57	110	26	9
Real_41-55	28	19	41	4
Real_56+	6	4	1	21

===== Matriz de Confusión del Modelo Decision Tree =====

Matriz de Confusión - Decision Tree



***** Random Forest *****

Accuracy: 0.640

Precision: 0.641

Recall: 0.640

F1 Score: 0.638

Matriz de Confusión:

	Pred_18-25	Pred_26-40	Pred_41-55	Pred_56+
Real_18-25	42	25	10	0
Real_26-40	23	151	20	8
Real_41-55	17	28	47	0
Real_56+	3	10	1	18

===== Matriz de Confusión del Modelo Random Forest =====

		Matriz de Confusión - Random Forest			
		Pred_18-25	Pred_26-40	Pred_41-55	Pred_56+
Real_18-25	Real_18-25	42	25	10	0
	Real_26-40	23	151	20	8
Real_41-55	Real_18-25	17	28	47	0
	Real_26-40	3	10	1	18

***** LightGBM *****

Accuracy: 0.789

Precision: 0.791

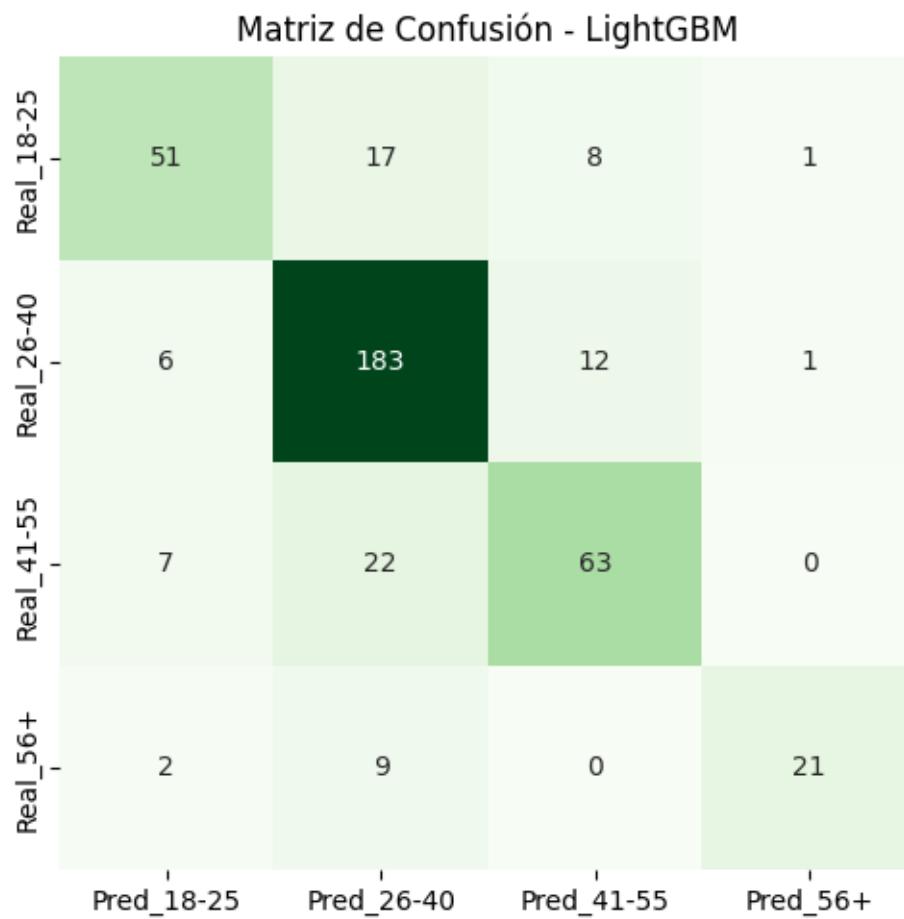
Recall: 0.789

F1 Score: 0.785

Matriz de Confusión:

	Pred_18-25	Pred_26-40	Pred_41-55	Pred_56+
Real_18-25	51	17	8	1
Real_26-40	6	183	12	1
Real_41-55	7	22	63	0
Real_56+	2	9	0	21

===== Matriz de Confusión del Modelo LightGBM =====



***** XGBoost *****

Accuracy: 0.789

Precision: 0.791

Recall: 0.789

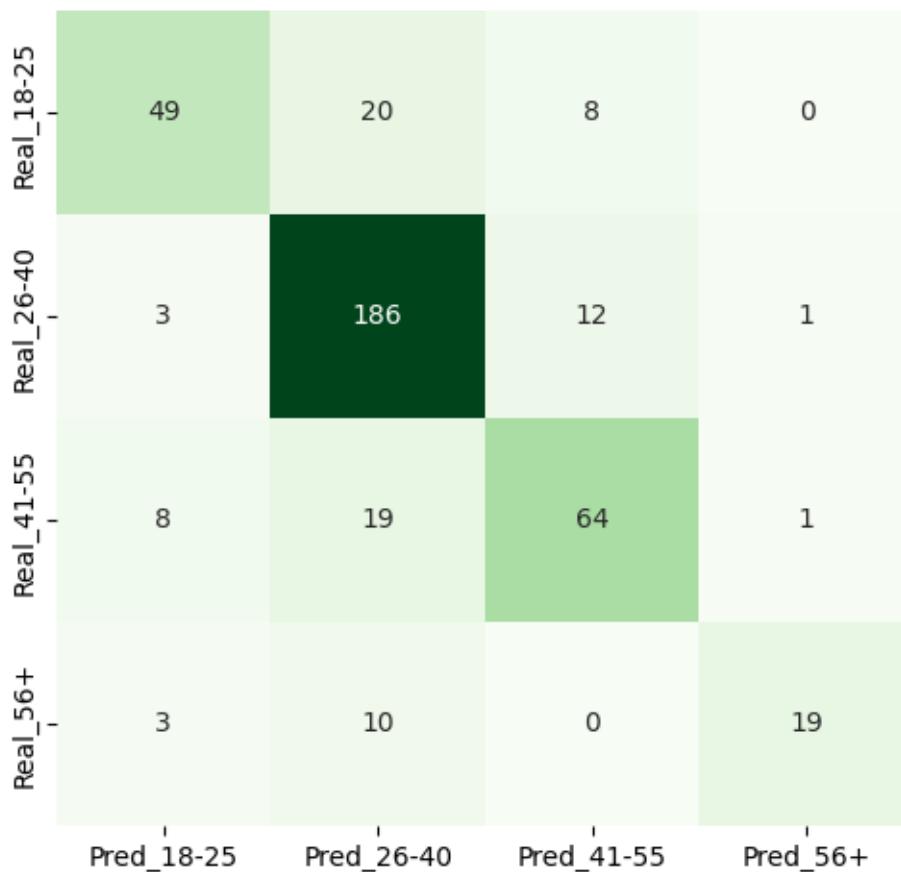
F1 Score: 0.783

Matriz de Confusión:

	Pred_18-25	Pred_26-40	Pred_41-55	Pred_56+
Real_18-25	49	20	8	0
Real_26-40	3	186	12	1
Real_41-55	8	19	64	1
Real_56+	3	10	0	19

===== Matriz de Confusión del Modelo XGBoost =====

Matriz de Confusión - XGBoost



2.7 Resultados de los modelos de Clasificación

Los resultados muestran que LightGBM y XGBoost son los modelos más efectivos para predecir el rango de edad de los clientes, con altos valores de accuracy y F1-Macro. Random Forest también ofrece un buen desempeño, mientras que Decision Tree queda rezagado en comparación.

```
print("\n" + "*70")
print("Resultados del Modelo 2")
print("*70)
# Crear el DataFrame de resultados con las métricas evaluadas
df_results_age = pd.DataFrame({
    'Modelo': ['Decision Tree', 'Random Forest', 'LightGBM', 'XGBoost'],
    'Accuracy': [global_dt_acc, global_rf_acc, global_lgb_acc, global_xgb_acc],
    'F1-Macro': [global_dt_f1, global_rf_f1, global_lgb_f1, global_xgb_f1]
})
# Convertir las métricas a numérico para asegurar el ordenamiento
df_results_age['Accuracy'] = df_results_age['Accuracy'].astype(float)
df_results_age['F1-Macro'] = df_results_age['F1-Macro'].astype(float)
# Encontrar el mejor modelo basado en F1-Macro
best_model_idx = df_results_age['F1-Macro'].idxmax()
best_model_name = df_results_age.loc[best_model_idx, 'Modelo']
best_model_metrics = df_results_age.loc[best_model_idx]
print("\n Tabla de Rendimiento:")
# Mostrar la tabla ordenada por F1-Macro
df_display = df_results_age.set_index('Modelo')[['Accuracy', 'F1-Macro']].sort_values(by='F1-Macro')
display(df_display.map('{:.3f}'.format))
print(f"\nMejor Modelo: {best_model_name}")
print(f" - Accuracy: {best_model_metrics['Accuracy']:.3f}")
print(f" - F1-Macro: {best_model_metrics['F1-Macro']:.3f}")
# Variables más importantes para el mejor modelo
# Crear un diccionario con los modelos entrenados
trained_models_age = {
    'Decision Tree': dt_model,
    'Random Forest': rf_model,
    'LightGBM': lgb_model,
    'XGBoost': xgb_model
}
print("\n Modelo 2 completado exitosamente!")
print("*70)
```

=====

Resultados del Modelo 2

=====

Tabla de Rendimiento:

Modelo	Accuracy	F1-Macro
LightGBM	0.789	0.785
XGBoost	0.789	0.783
Random Forest	0.640	0.638
Decision Tree	0.543	0.558

Mejor Modelo: LightGBM

- Accuracy: 0.789
- F1-Macro: 0.785

Modelo 2 completado exitosamente!

=====

2.7.1 Resultados de la evaluación de modelos de Clasificación para edad_rango:

Los resultados muestran que los modelos basados en boosting (LightGBM y XGBoost) superan claramente a Random Forest y Decision Tree. LightGBM obtiene el mejor rendimiento global con un F1-Macro de 0.761 y una accuracy de 0.789. Random Forest mejora respecto al árbol de decisión, pero sigue lejos de los modelos más avanzados. En general, LightGBM es el modelo más consistente en todas las clases.

Modelo	Accuracy	F1-Macro
LightGBM (Original)	0.789	0.785
XGBoost (Original)	0.789	0.783
Random Forest (Original)	0.640	0.638
Decision Tree (Original)	0.543	0.558

2.7.2 Gráfica comparativa de modelos de Clasificación

Se visualizan las métricas de Accuracy y F1-Macro para cada modelo de clasificación, facilitando la comparación del desempeño entre ellos. Además, se presenta la importancia de las variables según el modelo XGBoost, destacando cuáles características tienen mayor influencia en la predicción del rango de edad.

```

# Importar librerías necesarias para las visualizaciones
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.metrics import confusion_matrix

# --- Usar df_results_age para la comparación de métricas ---
# df_results_age ya contiene las métricas de Accuracy y F1-Score (Macro)
# para los modelos de clasificación de edad_rango

comparison_results_age = df_results_age.copy()

# Asegurarse de que las columnas sean numéricas para graficar y ordenar
comparison_results_age['Accuracy_float'] = comparison_results_age['Accuracy'].astype(float)
comparison_results_age['F1_Macro_float'] = comparison_results_age['F1-Macro'].astype(float)

# Ordenar por F1-Macro descendente
top_models_to_plot = comparison_results_age.sort_values(by='F1_Macro_float', ascending=False)

models_names = top_models_to_plot['Modelo'].tolist()
accuracy_scores = top_models_to_plot['Accuracy_float'].tolist()
f1_scores = top_models_to_plot['F1_Macro_float'].tolist()

# Crear figura con **dos gráficos**
fig, axes = plt.subplots(1, 2, figsize=(18, 7))

# -----
# Comparación de métricas
# -----
x_pos = np.arange(len(models_names))
width = 0.35

bars1 = axes[0].bar(x_pos - width/2, accuracy_scores, width, label='Accuracy', alpha=0.9, color='blue')
bars2 = axes[0].bar(x_pos + width/2, f1_scores, width, label='F1-Macro', alpha=0.9, color='red')

axes[0].set_xlabel('Modelos')
axes[0].set_ylabel('Puntuación')
axes[0].set_title('Comparación de Métricas de Clasificación (Predicción Edad)')
axes[0].set_xticks(x_pos)
axes[0].set_xticklabels(models_names, rotation=45, ha='right')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

```

```

# Añadir valores sobre las barras
for bar_group in [bars1, bars2]:
    for bar in bar_group:
        yval = bar.get_height()
        axes[0].text(bar.get_x() + bar.get_width()/2, yval + 0.01, round(yval, 2), ha='center')

# -----
# Importancia de variables del MEJOR MODELO
# -----
# Obtener el nombre del mejor modelo para edad_rango
best_model_name_age = comparison_results_age.loc[comparison_results_age['F1_Macro_float'].idxmax()]

# Diccionario de modelos entrenados para acceder a la importancia de características
trained_models_age = {
    'Decision Tree': dt_model,
    'Random Forest': rf_model,
    'LightGBM': lgb_model,
    'XGBoost': xgb_model
}

model_for_importance = trained_models_age.get(best_model_name_age)

if model_for_importance and hasattr(model_for_importance, 'feature_importances_'):
    feature_importances = model_for_importance.feature_importances_
    feature_names = X_scaled_age.columns

    df_importance = pd.DataFrame({
        'Feature': feature_names,
        'Importance': feature_importances
    }).sort_values(by='Importance', ascending=False).head(10)

    sns.barplot(
        x='Importance',
        y='Feature',
        data=df_importance,
        palette='Blues_d',
        ax=axes[1]
    )

    axes[1].set_title(f'Top 10 Importancia de Variables - {best_model_name_age} (Edad)')
    axes[1].set_xlabel('Importancia')
    axes[1].set_ylabel('Variable')

```

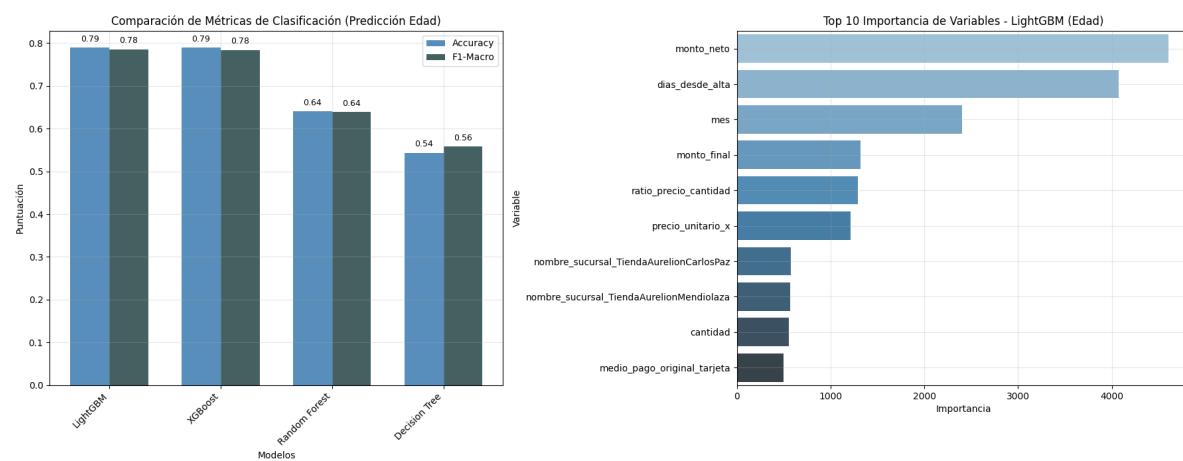
```

        axes[1].grid(True, alpha=0.3)
    else:
        axes[1].text(0.5, 0.5, f'No se pudo obtener la importancia de variables para {best_model_}
                           horizontalalignment='center', verticalalignment='center', transform=axes
        axes[1].set_title('Importancia de Variables (Edad)')

plt.tight_layout()
plt.show()

# -----
# Tabla resumen (opcional)
# -----
print("\n==== TABLA RESUMEN DE MÉTRICAS DE CLASIFICACIÓN (Predicción Edad) ===")
display(comparison_results_age[['Modelo', 'Accuracy', 'F1-Macro']].sort_values(by='F1-Macro'))

```



==== TABLA RESUMEN DE MÉTRICAS DE CLASIFICACIÓN (Predicción Edad) ===

	Modelo	Accuracy	F1-Macro
2	LightGBM	0.789	0.785
3	XGBoost	0.789	0.783
1	Random Forest	0.640	0.638
0	Decision Tree	0.543	0.558

Resultados gráficos de modelos de Clasificación para edad_rango:

- La comparación de métricas indica que LightGBM y XGBoost obtienen el mejor desempeño general, alcanzando las mayores puntuaciones tanto en Accuracy como en F1-Macro, lo que los posiciona como los modelos más robustos para esta tarea.
- El análisis de importancia de variables en XGBoost muestra que las características más determinantes en la predicción del rango de edad están asociadas al comportamiento del cliente, destacando principalmente `genero_0`, `genero_M` y `activo_como_cliente`. Características contextuales como `día_de_semana` y `categoría_de_producto` también juegan un papel relevante.

Estos resultados sugieren que la edad del cliente está más relacionada con patrones de compra y comportamiento demográfico, que con valores monetarios directos como monto total o cantidad comprada.

2.8 Validación cruzada para modelos de clasificación

Realizaremos una validación cruzada (K-Fold Cross-Validation) para evaluar la robustez y el rendimiento de los modelos en diferentes subconjuntos de datos. Esto nos ayudará a tener una estimación más fiable de cómo se desempeñarán en datos no vistos y a detectar posible sobreajuste. Utilizaremos `StratifiedKFold` para mantener la proporción de clases en cada pliegue, lo cual es importante en problemas de clasificación multiclas como este.

```
# Validación cruzada para clasificación
from sklearn.model_selection import StratifiedKFold, cross_val_score

print("\n==== VALIDACIÓN CRUZADA (K-Fold) MODELOS DE CLASIFICACIÓN ===")
print("*"*70)

# Definimos los modelos a evaluar. Usaremos los mejores modelos si fueron optimizados, si no
models_for_cv = {
    "Decision Tree": dt_model,
    "Random Forest": rf_model,
    "LightGBM": lgb_model,
    "XGBoost": xgb_model
}

# Configuramos la validación cruzada con StratifiedKFold
kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

cv_results = []

for name, model in models_for_cv.items():
    print(f"\nEvaluando {name} con Validación Cruzada...")
```

```

# Evaluamos la precisión
accuracy_scores = cross_val_score(model, X_train_res, y_train_res, cv=kf, scoring='accuracy')

# Evaluamos el F1-Macro. Asegórate de que el modelo soporta 'predict_proba' para algunas
# Para f1_macro, cross_val_score lo calcula directamente sobre las predicciones de clase
f1_macro_scores = cross_val_score(model, X_train_res, y_train_res, cv=kf, scoring='f1_macro')

cv_results.append({
    'Modelo': name,
    'CV_Accuracy_Mean': np.mean(accuracy_scores),
    'CV_Accuracy_Std': np.std(accuracy_scores),
    'CV_F1_Macro_Mean': np.mean(f1_macro_scores),
    'CV_F1_Macro_Std': np.std(f1_macro_scores)
})

print(f" - Accuracy Promedio (CV): {np.mean(accuracy_scores):.3f} (+/- {np.std(accuracy_scores):.3f})")
print(f" - F1-Macro Promedio (CV): {np.mean(f1_macro_scores):.3f} (+/- {np.std(f1_macro_scores):.3f})")

# Mostramos los resultados en un DataFrame para facilitar la comparación
df_cv_results = pd.DataFrame(cv_results)
#display(df_cv_results.sort_values(by='CV_F1_Macro_Mean', ascending=False))
display(df_cv_results)
best_model_model2 = df_cv_results.sort_values(by='CV_F1_Macro_Mean', ascending=False).iloc[0]
print(f"\nEl mejor modelo es: {best_model_model2}")

```

==== VALIDACIÓN CRUZADA (K-Fold) MODELOS DE CLASIFICACIÓN ===

Evaluando Decision Tree con Validación Cruzada...

- Accuracy Promedio (CV): 0.658 (+/- 0.019)
- F1-Macro Promedio (CV): 0.657 (+/- 0.020)

Evaluando Random Forest con Validación Cruzada...

- Accuracy Promedio (CV): 0.829 (+/- 0.010)
- F1-Macro Promedio (CV): 0.829 (+/- 0.010)

Evaluando LightGBM con Validación Cruzada...

- Accuracy Promedio (CV): 0.906 (+/- 0.006)
- F1-Macro Promedio (CV): 0.906 (+/- 0.006)

Evaluando XGBoost con Validación Cruzada...

- Accuracy Promedio (CV): 0.909 (+/- 0.007)
- F1-Macro Promedio (CV): 0.909 (+/- 0.007)

	Modelo	CV_Accuracy_Mean	CV_Accuracy_Std	CV_F1_Macro_Mean	CV_F1_Macro_Std
0	Decision Tree	0.658025	0.018764	0.656891	0.019917
1	Random Forest	0.829321	0.009789	0.828991	0.010438
2	LightGBM	0.905864	0.006095	0.906062	0.006187
3	XGBoost	0.908642	0.007132	0.908824	0.007021

El mejor modelo es: XGBoost

2.9 Resultados de la Validación Cruzada para modelos de Clasificación

Los resultados de validación cruzada muestran una clara superioridad de LightGBM y XGBoost, con métricas consistentemente altas y baja variabilidad, lo que evidencia alta estabilidad y capacidad predictiva. Random Forest presenta un desempeño intermedio sólido, mientras que Decision Tree queda rezagado, confirmando que modelos más complejos generalizan mejor en este caso.

Modelo	Accuracy ↑	F1-Macro ↑	-----	-----	-----	-----
XGBoost	0.908	0.908	0.905	0.905	0.829	0.828
Decision Tree	0.658	0.656				

Sugerencia: Considerar el uso de modelos más robustos como LightGBM y XGBoost para implementaciones siguientes debido a su mejor equilibrio entre precisión y estabilidad.

2.10 Resumen de resultados

```
print("\n" + "="*70)
print("RESUMEN FINAL - MODELO 2: CLASIFICACIÓN DE edad_rango")
print("="*70)
# Calcular todas las métricas para cada modelo
models_dict = {
    'Decision Tree': dt_model,
    'Random Forest': rf_model,
    'LightGBM': lgb_model,
    'XGBoost': xgb_model
}
results_list = []
```

```

for model_name, model in models_dict.items():
    y_pred = model.predict(X_test_age)

    accuracy = accuracy_score(y_test_age, y_pred)
    precision = precision_score(y_test_age, y_pred, average='weighted')
    recall = recall_score(y_test_age, y_pred, average='weighted')
    f1 = f1_score(y_test_age, y_pred, average='weighted')
    f1_macro = f1_score(y_test_age, y_pred, average='macro')

    results_list.append({
        'Modelo': model_name,
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1-Score': f1,
        'F1-Score (Macro)': f1_macro
    })

# Crear DataFrame de resultados
df_results_age = pd.DataFrame(results_list)
# Encontrar el mejor modelo basado en F1-Score
best_model_idx = df_results_age['F1-Score'].idxmax()
best_model_name = df_results_age.loc[best_model_idx, 'Modelo']
best_model_metrics = df_results_age.loc[best_model_idx]
print("\n Tabla de Rendimiento (Métricas en Conjunto de Prueba):")
# Mostrar la tabla ordenada por F1-Score
df_display = df_results_age.set_index('Modelo')[['Accuracy', 'Precision', 'Recall', 'F1-Score']]
display(df_display.map('{:.3f}'.format))
print(f"\n Mejor Modelo: {best_model_name}")
print(f" - Accuracy: {best_model_metrics['Accuracy']:.3f}")
print(f" - Precision: {best_model_metrics['Precision']:.3f}")
print(f" - Recall: {best_model_metrics['Recall']:.3f}")
print(f" - F1-Score: {best_model_metrics['F1-Score']:.3f}")
print(f" - F1-Score (Macro): {best_model_metrics['F1-Score (Macro)']:.3f}")
# Variables más importantes para el mejor modelo
model_for_importance = models_dict.get(best_model_name)
if model_for_importance is not None and hasattr(model_for_importance, 'feature_importances_'):
    # Obtener nombres de las características desde X_test_age
    feature_names_age = X_test_age.columns.tolist() if hasattr(X_test_age, 'columns') else []
    importances = model_for_importance.feature_importances_

    # Asegurarnos de que tenemos el mismo número de features
    n_features = min(len(feature_names_age), len(importances))

```

```

# Crear pares de (nombre, importancia) y ordenar
feature_importance_pairs = [(feature_names_age[i], importances[i]) for i in range(n_features)]
feature_importance_pairs.sort(key=lambda x: x[1], reverse=True)
print(f"\n Top 3 Variables Más Importantes - {best_model_name}:")
for i in range(min(3, len(feature_importance_pairs))):
    feature_name, importance_value = feature_importance_pairs[i]
    print(f"  {i+1}. {feature_name}: {importance_value:.3f}")
else:
    print(f"\n No se pudo obtener la importancia de las variables para {best_model_name}")
print("\n Modelo 2 completado exitosamente!")
print("=*70")

```

=====
RESUMEN FINAL - MODELO 2: CLASIFICACIÓN DE edad_rango
=====

Tabla de Rendimiento (Métricas en Conjunto de Prueba):

Modelo	Accuracy	Precision	Recall	F1-Score	F1-Score (Macro)
LightGBM	0.789	0.791	0.789	0.785	0.761
XGBoost	0.789	0.791	0.789	0.783	0.749
Random Forest	0.640	0.641	0.640	0.638	0.605
Decision Tree	0.543	0.607	0.543	0.558	0.537

Mejor Modelo: LightGBM

- Accuracy: 0.789
- Precision: 0.791
- Recall: 0.789
- F1-Score: 0.785
- F1-Score (Macro): 0.761

Top 3 Variables Más Importantes - LightGBM:

1. monto_neto: 4608.000
2. dias_desde_alta: 4072.000
3. mes: 2402.000

Modelo 2 completado exitosamente!

=====

3 MODELO: Clasificación

Predicción de variable objetivo categoria

Objetivo: Predecir la categoría de producto (**categoria**) más probable que comprará un cliente.

Justificación: Conocer las preferencias de productos de los clientes permite: - Optimizar el inventario por categoría - Personalizar recomendaciones de productos - Mejorar las estrategias de marketing cruzado

Algoritmos Seleccionados: - **Random Forest Clasifier:** Excelente para clasificación multi-clase con múltiples variables de entrada - **LightGBM Clasifier:** Excelente para clasificación multi-clase con múltiples variables de entrada - **XGBoost Clasifier:** Excelente para clasificación multi-clase con múltiples variables de entrada - **Logistic Regression:** Excelente para clasificación binaria con múltiples variables de entrada

3.1 Implementación del Modelo 3

```
# Codificar variables categóricas
# Preparación de datos para el modelo de clasificación - Categoría
print("== PREPARACIÓN DE DATOS PARA CATEGORÍA ==")

# Seleccionar variables de entrada (X)
features_clf_cat = ['cantidad', 'precio_unitario_x', 'monto_neto','monto_final', 'edad_rango'
                     'genero', 'medio_pago_original', 'nombre_sucursal', 'nombre_vendedor']

# Crear dataset para modelado
df_clf_cat = df_master_refined[features_clf_cat + ['categoria']].copy()

# Codificar variables categóricas
le_age_cat = LabelEncoder()
le_payment_cat = LabelEncoder()
le_store_cat = LabelEncoder()
le_vendedor = LabelEncoder()
le_gender_cat = LabelEncoder()
```

```

df_clf_cat['edad_rango_encoded'] = le_age_cat.fit_transform(df_clf_cat['edad_rango'])
df_clf_cat['medio_pago_encoded'] = le_payment_cat.fit_transform(df_clf_cat['medio_pago_origen'])
df_clf_cat['sucursal_encoded'] = le_store_cat.fit_transform(df_clf_cat['nombre_sucursal'])
df_clf_cat['vendedor_encoded'] = le_vendedor.fit_transform(df_clf_cat['nombre_vendedor'])
df_clf_cat['genero_encoded'] = le_gender_cat.fit_transform(df_clf_cat['genero'])

# Variables finales para el modelo
X_clf_cat = df_clf_cat[['cantidad', 'precio_unitario_x', 'monto_neto', 'monto_final',
                         'edad_rango_encoded', 'genero_encoded',
                         'medio_pago_encoded', 'sucursal_encoded', 'vendedor_encoded']]
y_clf_cat = df_clf_cat['categoria']

print(f"Variables de entrada (X): {X_clf_cat.columns.tolist()}")
print(f"Variable objetivo (y): categoria")
print(f"Distribución de clases:")
print(y_clf_cat.value_counts().head(5).to_dict())

# Verificar balance de clases
class_distribution_cat = y_clf_cat.value_counts(normalize=True)
print(f"\nTop 5 categorías:")
for categoria, pct in class_distribution_cat.head().items():
    print(f"{categoria}: {pct:.1%}")

==== PREPARACIÓN DE DATOS PARA CATEGORIA ====
Variables de entrada (X): ['cantidad', 'precio_unitario_x', 'monto_neto', 'monto_final', 'edad_rango_encoded', 'genero_encoded', 'medio_pago_encoded', 'sucursal_encoded', 'vendedor_encoded']
Variable objetivo (y): categoria
Distribución de clases:
{'Almacén': 504, 'Snacks y Dulces': 222, 'Bebidas': 207, 'Bebidas Alcohólicas': 204, 'Limpieza': 181}

Top 5 categorías:
Almacén: 25.0%
Snacks y Dulces: 11.0%
Bebidas: 10.3%
Bebidas Alcohólicas: 10.1%
Limpieza: 9.4%

```

3.2 Especificación de las Métricas de Evaluación

Para Clasificación utilizaremos las mismas métricas que el modelo anterior: - Accuracy: Porcentaje de predicciones correctas - Matriz de Confusión: Tabla que muestra

aciertos y errores en predicciones - **Precision, Recall, F1-Score:** Métricas detalladas por clase

3.3 Modelos ML Implementados para clasificación

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from lightgbm import LGBMClassifier
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from imblearn.over_sampling import SMOTE # Importamos SMOTE para el balanceo de clases

# --- 1. División de datos en conjuntos de entrenamiento y prueba (80%/20%) ---
# Usamos 'stratify' en la variable objetivo 'y_clf_cat' para asegurar que la proporción de c
# en ambos conjuntos, lo cual es crucial para problemas con desbalance de clases.
X_train_cat, X_test_cat, y_train_cat, y_test_cat = train_test_split(
    X_clf_cat, y_clf_cat, test_size=0.2, random_state=42, stratify=y_clf_cat
)

print(f"Conjunto de entrenamiento: {X_train_cat.shape[0]} muestras")
print(f"Conjunto de prueba: {X_test_cat.shape[0]} muestras")

# --- 2. Codificar la variable objetivo 'categoria' ---
# La codificación es necesaria para los modelos de clasificación (especialmente para LightGB
# que trabajan mejor con etiquetas numéricas para problemas multiclas).
le_categoria = LabelEncoder()
y_train_cat_encoded = le_categoria.fit_transform(y_train_cat)
y_test_cat_encoded = le_categoria.transform(y_test_cat)

print("\nDistribución de clases de la variable objetivo 'categoria' en el conjunto de entrenam
print(pd.Series(y_train_cat_encoded).value_counts().sort_index().to_dict())

# --- 3. Balanceo de clases con SMOTE ---
# Aplicamos SMOTE para mitigar el desbalance de clases en el conjunto de entrenamiento.
# SMOTE crea nuevas muestras sintéticas para las clases minoritarias.
print("\nAplicando SMOTE al conjunto de entrenamiento...")
smote = SMOTE(random_state=42)
X_train_cat_resampled, y_train_cat_resampled = smote.fit_resample(X_train_cat, y_train_cat_)

print("Distribución de clases de la variable objetivo 'categoria' en el conjunto de entrenam
```

```

print(pd.Series(y_train_cat_resampled).value_counts().sort_index().to_dict())

# --- 4. Escalado de características para Logistic Regression ---
# Los modelos basados en árboles (RF, LightGBM, XGBoost) no requieren escalado de características
# Sin embargo, Logistic Regression es sensible a la escala de los datos, por lo que escalamos
scaler_cat = StandardScaler()
X_train_cat_scaled = scaler_cat.fit_transform(X_train_cat_resampled) # Escalar datos resampleados
X_test_cat_scaled = scaler_cat.transform(X_test_cat) # Escalar datos de prueba sin resampling

# --- 5. Inicialización de modelos ---
print("\n==== INICIALIZACIÓN DE MODELOS DE CLASIFICACIÓN PARA 'CATEGORIA' ====")
models_cat = {
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42, n_jobs=-1, class_weight='balanced'),
    'Logistic Regression': LogisticRegression(random_state=42, max_iter=1000, multi_class='multinomial'),
    'LightGBM': LGBMClassifier(random_state=42, verbose=-1, n_jobs=-1, eval_metric='multi_logloss'),
    'XGBoost': XGBClassifier(random_state=42, eval_metric='mlogloss', use_label_encoder=False)
}

# --- 6. Entrenamiento de modelos ---
print("\n==== ENTRENAMIENTO DE MODELOS ====")
trained_models_cat = {}

for name, model in models_cat.items():
    print(f"Entrenando {name}...")
    if name == 'Logistic Regression':
        # Logistic Regression usa los datos escalados y resampleados
        model.fit(X_train_cat_scaled, y_train_cat_resampled)
    else:
        # Otros modelos (árboles) usan los datos sin escalar pero resampleados
        model.fit(X_train_cat_resampled, y_train_cat_resampled)
    trained_models_cat[name] = model

print("\n Todos los modelos entrenados exitosamente con datos balanceados!")

```

Conjunto de entrenamiento: 1610 muestras

Conjunto de prueba: 403 muestras

Distribución de clases de la variable objetivo 'categoria' en el conjunto de entrenamiento (a priori):
{0: 403, 1: 166, 2: 163, 3: 116, 4: 132, 5: 86, 6: 151, 7: 130, 8: 85, 9: 178}

Aplicando SMOTE al conjunto de entrenamiento...

Distribución de clases de la variable objetivo 'categoria' en el conjunto de entrenamiento (a posteriori):

```
{0: 403, 1: 403, 2: 403, 3: 403, 4: 403, 5: 403, 6: 403, 7: 403, 8: 403, 9: 403}
```

```
==== INICIALIZACIÓN DE MODELOS DE CLASIFICACIÓN PARA 'CATEGORIA' ===
```

```
==== ENTRENAMIENTO DE MODELOS ===
```

```
Entrenando Random Forest...
```

```
Entrenando Logistic Regression...
```

```
Entrenando LightGBM...
```

```
Entrenando XGBoost...
```

```
Todos los modelos entrenados exitosamente con datos balanceados!
```

3.4 Entrenamiento y Predicción

```
#eVALUACIÓN
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score, confusion_matrix
import pandas as pd
import numpy as np

# ----- Función para evaluar modelos de clasificación -----
def evaluate_classification_model(model, X_test, y_test_encoded, model_name, label_encoder, label_decoder):
    """Evalúa un modelo de clasificación y muestra las métricas clásicas."""

    # Para Logistic Regression usamos datos escalados,
    # para los demás modelos usamos los datos codificados originales
    if is_logistic_regression:
        y_pred = model.predict(X_test_cat_scaled)
    else:
        y_pred = model.predict(X_test_cat)

    # --- Métricas clásicas ---
    acc = accuracy_score(y_test_encoded, y_pred)
    f1 = f1_score(y_test_encoded, y_pred, average='weighted')
    precision = precision_score(y_test_encoded, y_pred, average='weighted')
    recall = recall_score(y_test_encoded, y_pred, average='weighted')

    cm = confusion_matrix(y_test_encoded, y_pred)

    print(f"\n===== {model_name} =====")
    print(f"Accuracy: {acc:.3f}")
    print(f"Precision: {precision:.3f}")
```

```

print(f"Recall: {recall:.3f}")
print(f"F1-Score: {f1:.3f}")

# Mostrar matriz de confusión con etiquetas decodificadas
labels_decoded = label_encoder.inverse_transform(sorted(np.unique(y_test_encoded)))
df_cm = pd.DataFrame(cm,
                      index=[f"Real_{l}" for l in labels_decoded],
                      columns=[f"Pred_{l}" for l in labels_decoded])
print("\nMatriz de Confusión:")
display(df_cm)

# Mostrar classification_report estándar
print("\nClassification Report:")
print(classification_report(y_test_encoded, y_pred, target_names=label_encoder.classes_))

return acc, f1, precision, recall

# --- Evaluación de modelos en el conjunto de prueba ---
print("\n==== EVALUACIÓN DE MODELOS EN EL CONJUNTO DE PRUEBA PARA 'CATEGORIA' ====")

metrics_cat = []

# Random Forest
rf_acc_cat, rf_f1_cat, rf_precision_cat, rf_recall_cat = evaluate_classification_model(
    trained_models_cat['Random Forest'], X_test_cat, y_test_cat_encoded, "Random Forest", le)
metrics_cat.append({'Modelo': 'Random Forest', 'Accuracy': rf_acc_cat, 'Precision': rf_precision_cat})

# Logistic Regression
logreg_acc_cat, logreg_f1_cat, logreg_precision_cat, logreg_recall_cat = evaluate_classification_model(
    trained_models_cat['Logistic Regression'], X_test_cat_scaled, y_test_cat_encoded, "Logistic Regression")
metrics_cat.append({'Modelo': 'Logistic Regression', 'Accuracy': logreg_acc_cat, 'Precision': logreg_precision_cat})

# LightGBM
lgb_acc_cat, lgb_f1_cat, lgb_precision_cat, lgb_recall_cat = evaluate_classification_model(
    trained_models_cat['LightGBM'], X_test_cat, y_test_cat_encoded, "LightGBM", le_categorical)
metrics_cat.append({'Modelo': 'LightGBM', 'Accuracy': lgb_acc_cat, 'Precision': lgb_precision_cat})

# XGBoost
xgb_acc_cat, xgb_f1_cat, xgb_precision_cat, xgb_recall_cat = evaluate_classification_model(
    trained_models_cat['XGBoost'], X_test_cat, y_test_cat_encoded, "XGBoost", le_categorical)
metrics_cat.append({'Modelo': 'XGBoost', 'Accuracy': xgb_acc_cat, 'Precision': xgb_precision_cat})

```

```
# --- Comparación de Resultados ---
print("\n==== COMPARACIÓN DE MODELOS DE CLASIFICACIÓN PARA 'CATEGORIA' ===")
df_comparison_cat = pd.DataFrame(metrics_cat)
display(df_comparison_cat.sort_values(by='F1', ascending=False))
```

==== EVALUACIÓN DE MODELOS EN EL CONJUNTO DE PRUEBA PARA 'CATEGORIA' ===

===== Random Forest =====

Accuracy: 0.439

Precision: 0.426

Recall: 0.439

F1-Score: 0.425

Matriz de Confusión:

	Pred_Almacén	Pred_Bebidas	Pred_Bebidas Alcohólicas	Pred_Congelados
Real_Almacén	42	9	10	8
Real_Bebidas	7	24	2	5
Real_Bebidas Alcohólicas	7	2	19	1
Real_Congelados	3	3	0	13
Real_Cuidado Personal	9	3	0	4
Real_Infusiones	3	3	2	1
Real_Limpieza	0	0	1	3
Real_Lácteos y Frescos	4	2	4	0
Real_Panadería y Repostería	4	2	0	2
Real_Snacks y Dulces	9	1	1	1

Classification Report:

	precision	recall	f1-score	support
Almacén	0.48	0.42	0.44	101
Bebidas	0.49	0.59	0.53	41
Bebidas Alcohólicas	0.49	0.46	0.47	41
Congelados	0.34	0.45	0.39	29
Cuidado Personal	0.33	0.24	0.28	33
Infusiones	0.35	0.27	0.31	22
Limpieza	0.50	0.79	0.61	38

Lácteos y Frescos	0.18	0.09	0.12	33
Panadería y Repostería	0.40	0.57	0.47	21
Snacks y Dulces	0.49	0.45	0.47	44
accuracy			0.44	403
macro avg	0.40	0.43	0.41	403
weighted avg	0.43	0.44	0.43	403

===== Logistic Regression =====

Accuracy: 0.141

Precision: 0.155

Recall: 0.141

F1-Score: 0.135

Matriz de Confusión:

	Pred_Almacén	Pred_Bebidas	Pred_Bebidas Alcohólicas	Pred_Congelados
Real_Almacén	21	16	3	7
Real_Bebidas	7	3	3	2
Real_Bebidas Alcohólicas	11	5	3	3
Real_Congelados	3	6	3	3
Real_Cuidado Personal	13	0	0	4
Real_Infusiones	0	3	2	1
Real_Limpieza	2	17	2	1
Real_Lácteos y Frescos	5	3	3	2
Real_Panadería y Repostería	5	0	1	4
Real_Snacks y Dulces	11	5	2	5

Classification Report:

	precision	recall	f1-score	support
Almacén	0.27	0.21	0.23	101
Bebidas	0.05	0.07	0.06	41
Bebidas Alcohólicas	0.14	0.07	0.10	41
Congelados	0.09	0.10	0.10	29
Cuidado Personal	0.00	0.00	0.00	33
Infusiones	0.08	0.18	0.11	22
Limpieza	0.15	0.37	0.21	38
Lácteos y Frescos	0.25	0.06	0.10	33

Panadería y Repostería	0.12	0.14	0.13	21
Snacks y Dulces	0.15	0.09	0.11	44
accuracy			0.14	403
macro avg	0.13	0.13	0.12	403
weighted avg	0.16	0.14	0.14	403

===== LightGBM =====

Accuracy: 0.849

Precision: 0.850

Recall: 0.849

F1-Score: 0.847

Matriz de Confusión:

	Pred_Almacén	Pred_Bebidas	Pred_Bebidas Alcohólicas	Pred_Congelados
Real_Almacén	95	0	1	1
Real_Bebidas	0	36	1	0
Real_Bebidas Alcohólicas	0	1	34	1
Real_Congelados	0	0	0	20
Real_Cuidado Personal	4	1	1	4
Real_Infusiones	0	0	0	0
Real_Limpieza	0	0	0	2
Real_Lácteos y Frescos	0	0	0	0
Real_Panadería y Repostería	1	0	0	2
Real_Snacks y Dulces	1	0	0	0

Classification Report:

	precision	recall	f1-score	support
Almacén	0.94	0.94	0.94	101
Bebidas	0.95	0.88	0.91	41
Bebidas Alcohólicas	0.92	0.83	0.87	41
Congelados	0.67	0.69	0.68	29
Cuidado Personal	0.63	0.52	0.57	33
Infusiones	0.85	1.00	0.92	22
Limpieza	0.72	0.89	0.80	38
Lácteos y Frescos	0.86	0.76	0.81	33
Panadería y Repostería	0.82	0.86	0.84	21

Snacks y Dulces	0.89	0.93	0.91	44
accuracy			0.85	403
macro avg	0.82	0.83	0.82	403
weighted avg	0.85	0.85	0.85	403

===== XGBoost =====

Accuracy: 0.727

Precision: 0.723

Recall: 0.727

F1-Score: 0.722

Matriz de Confusión:

	Pred_Almacén	Pred_Bebidas	Pred_Bebidas Alcohólicas	Pred_Congelad
Real_Almacén	85	5	4	1
Real_Bebidas	2	31	0	0
Real_Bebidas Alcohólicas	1	1	32	1
Real_Congelados	0	3	0	14
Real_Cuidado Personal	5	1	1	5
Real_Infusiones	1	3	0	0
Real_Limpieza	0	0	0	2
Real_Lácteos y Frescos	1	0	0	0
Real_Panadería y Repostería	1	0	0	4
Real_Snacks y Dulces	6	0	1	0

Classification Report:

	precision	recall	f1-score	support
Almacén	0.83	0.84	0.84	101
Bebidas	0.70	0.76	0.73	41
Bebidas Alcohólicas	0.84	0.78	0.81	41
Congelados	0.52	0.48	0.50	29
Cuidado Personal	0.50	0.36	0.42	33
Infusiones	0.62	0.68	0.65	22
Limpieza	0.71	0.89	0.79	38
Lácteos y Frescos	0.69	0.61	0.65	33
Panadería y Repostería	0.83	0.71	0.77	21
Snacks y Dulces	0.71	0.80	0.75	44

accuracy			0.73	403
macro avg	0.70	0.69	0.69	403
weighted avg	0.72	0.73	0.72	403

==== COMPARACIÓN DE MODELOS DE CLASIFICACIÓN PARA 'CATEGORIA' ===

Modelo	Accuracy	Precision	Recall	F1
2 LightGBM	0.848635	0.850077	0.848635	0.846952
3 XGBoost	0.727047	0.723251	0.727047	0.722218
0 Random Forest	0.439206	0.425889	0.439206	0.425137
1 Logistic Regression	0.141439	0.155354	0.141439	0.135138

3.5 Validación cruzada

```
#cv clasico
from sklearn.model_selection import StratifiedKFold, cross_validate
import numpy as np
import pandas as pd

print("\n==== REALIZANDO VALIDACIÓN CRUZADA (K-Fold) PARA 'CATEGORIA' ===")
print("*70

# ---- Métricas clásicas para multiclase ----
scoring = {
    'accuracy': 'accuracy',
    'precision': 'precision_weighted',
    'recall': 'recall_weighted',
    'f1': 'f1_weighted'
}

kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

cv_results_cat = []

for name, model in trained_models_cat.items():
    print(f"\nEvaluando {name} con Validación Cruzada...")
```

```

# Datos según el modelo
if name == 'Logistic Regression':
    X_cv = X_train_cat_scaled
else:
    X_cv = X_train_cat_resampled

# Ejecutamos la validación cruzada
results = cross_validate(model, X_cv, y_train_cat_resampled, cv=kf, scoring=scoring, n_j

# Guardamos los resultados
cv_results_cat.append({
    'Modelo': name,
    'CV_Accuracy_Mean': np.mean(results['test_accuracy']),
    'CV_Accuracy_Std': np.std(results['test_accuracy']),
    'CV_F1_Mean': np.mean(results['test_f1']),
    'CV_F1_Std': np.std(results['test_f1']),
    'CV_Precision_Mean': np.mean(results['test_precision']),
    'CV_Precision_Std': np.std(results['test_precision']),
    'CV_Recall_Mean': np.mean(results['test_recall']),
    'CV_Recall_Std': np.std(results['test_recall']),
})

# Impresión ordenada de resultados
print(f" - Accuracy Promedio (CV): {np.mean(results['test_accuracy']):.3f} (+/- {np.std(resu
print(f" - F1 Promedio (CV): {np.mean(results['test_f1']):.3f} (+/- {np.std(results['te
print(f" - Precision Promedio (CV): {np.mean(results['test_precision']):.3f} (+/- {np.s
print(f" - Recall Promedio (CV): {np.mean(results['test_recall']):.3f} (+/- {np.std(resu

# ---- Mostrar resultados consolidados ----
df_cv_results_cat = pd.DataFrame(cv_results_cat)

print("\n==== REPORTE CONSOLIDADO DE VALIDACIÓN CRUZADA PARA 'CATEGORIA' ===")
display(df_cv_results_cat.sort_values(by='CV_F1_Mean', ascending=False))
best_model_cat = df_cv_results_cat.sort_values(by='CV_F1_Mean', ascending=False).iloc[0]['Mod
print(f"\nMejor modelo para 'CATEGORIA': {best_model_cat}")

```

==== REALIZANDO VALIDACIÓN CRUZADA (K-Fold) PARA 'CATEGORIA' ===
=====

Evaluando Random Forest con Validación Cruzada...
- Accuracy Promedio (CV): 0.681 (+/- 0.011)

- F1 Promedio (CV): 0.679 (+/- 0.011)
- Precision Promedio (CV): 0.682 (+/- 0.011)
- Recall Promedio (CV): 0.681 (+/- 0.011)

Evaluando Logistic Regression con Validación Cruzada...

- Accuracy Promedio (CV): 0.210 (+/- 0.006)
- F1 Promedio (CV): 0.189 (+/- 0.010)
- Precision Promedio (CV): 0.190 (+/- 0.014)
- Recall Promedio (CV): 0.210 (+/- 0.006)

Evaluando LightGBM con Validación Cruzada...

- Accuracy Promedio (CV): 0.766 (+/- 0.013)
- F1 Promedio (CV): 0.765 (+/- 0.013)
- Precision Promedio (CV): 0.768 (+/- 0.013)
- Recall Promedio (CV): 0.766 (+/- 0.013)

Evaluando XGBoost con Validación Cruzada...

- Accuracy Promedio (CV): 0.752 (+/- 0.011)
- F1 Promedio (CV): 0.752 (+/- 0.010)
- Precision Promedio (CV): 0.755 (+/- 0.010)
- Recall Promedio (CV): 0.752 (+/- 0.011)

==== REPORTE CONSOLIDADO DE VALIDACIÓN CRUZADA PARA 'CATEGORIA' ===

	Modelo	CV_Accuracy_Mean	CV_Accuracy_Std	CV_F1_Mean	CV_F1_Std	CV_P
2	LightGBM	0.765509	0.012941	0.764877	0.013068	0.7678
3	XGBoost	0.752109	0.010684	0.751907	0.010460	0.7552
0	Random Forest	0.681141	0.010901	0.678646	0.010870	0.6822
1	Logistic Regression	0.210174	0.006499	0.188626	0.009850	0.1896

Mejor modelo para 'CATEGORIA': LightGBM

3.6 Resultados en Gráficos

```
# Crear visualizaciones completas del Modelo 3
fig, axes = plt.subplots(2, 2, figsize=(18, 14))

# =====
```

```

# 1. Comparación de métricas entre modelos (Desde df_comparison_cat)
# =====
# Usamos el DataFrame df_comparison_cat que contiene las métricas del test set
comparison_df = df_comparison_cat.set_index('Modelo').sort_values(by='F1', ascending=False)
models_list_comp = comparison_df.index.tolist()
accuracy_scores_comp = comparison_df['Accuracy'].astype(float).tolist()
f1_scores_comp = comparison_df['F1'].astype(float).tolist() # Usamos F1 weighted aquí

x_pos_comp = np.arange(len(models_list_comp))
width_comp = 0.35

bars1_comp = axes[0, 0].bar(x_pos_comp - width_comp/2, accuracy_scores_comp, width_comp,
                           label='Accuracy', alpha=0.8, color='steelblue')
bars2_comp = axes[0, 0].bar(x_pos_comp + width_comp/2, f1_scores_comp, width_comp,
                           label='F1-Score (Weighted)', alpha=0.8, color='darkorange')

axes[0, 0].set_xlabel('Modelos', fontsize=12)
axes[0, 0].set_ylabel('Puntuación', fontsize=12)
axes[0, 0].set_title('Comparación de Métricas en Conjunto de Prueba', fontsize=14, fontweight='bold')
axes[0, 0].set_xticks(x_pos_comp)
axes[0, 0].set_xticklabels(models_list_comp, rotation=45, ha='right', fontsize=10)
axes[0, 0].legend(fontsize=10)
axes[0, 0].grid(True, alpha=0.3, axis='y')
axes[0, 0].set_ylim([0, 1.05])

# Agregar valores sobre las barras
for bars in [bars1_comp, bars2_comp]:
    for bar in bars:
        height = bar.get_height()
        axes[0, 0].text(bar.get_x() + bar.get_width()/2., height,
                        f'{height:.2f}', ha='center', va='bottom', fontsize=9)

# =====
# 2. Distribución de Categorías (Top 5) - Gráfico de Pastel mejorado
# =====
category_counts_top5 = y_clf_cat.value_counts().head(5)
colors_pie = plt.cm.Set3(np.linspace(0, 1, len(category_counts_top5))) # Ajustar colores si es necesario

wedges, texts, autotexts = axes[0, 1].pie(
    category_counts_top5.values,
    labels=category_counts_top5.index,
    autopct='%.1f%%',

```

```

colors=colors_pie,
startangle=90,
pctdistance=0.85, # Distancia de los porcentajes del centro
wedgeprops=dict(width=0.4, edgecolor='w') # Efecto de rosquilla
)

# Mejorar legibilidad del pie chart
for text in texts:
    text.set_fontsize(14)
for autotext in autotexts:
    autotext.set_color('black') # Color del texto para mayor contraste
    autotext.set_fontweight('bold')
    autotext.set_fontsize(14)

axes[0, 1].set_title('Distribución de Categorías (Top 5)', fontsize=15, fontweight='bold')
axes[0, 1].axis('equal') # Asegura que el círculo sea un círculo

# =====
# 3. Importancia de Variables para el mejor modelo de árbol (Random Forest)
# =====
# Usamos Random Forest como un buen representante de modelos basados en árboles
model_for_importance = trained_models_cat.get(best_model_cat)

if model_for_importance is not None and hasattr(model_for_importance, 'feature_importances_'):
    feature_names_cat = X_clf_cat.columns.tolist()
    importances = model_for_importance.feature_importances_
    indices = np.argsort(importances)[::-1] # Ordenar de mayor a menor

    # Limitar a las 10 características más importantes para visualización clara
    top_n_features = 10
    if len(indices) > top_n_features:
        indices = indices[:top_n_features]

    axes[1, 0].barh(range(len(indices)), importances[indices], color='teal', alpha=0.7)
    axes[1, 0].set_yticks(range(len(indices)))
    axes[1, 0].set_yticklabels([feature_names_cat[i] for i in indices], fontsize=10)
    axes[1, 0].invert_yaxis() # Más importante arriba
    axes[1, 0].set_xlabel('Importancia', fontsize=12)
    axes[1, 0].set_title(f'Top {len(indices)} Importancia de Variables - {best_model_cat}', fontsize=14, fontweight='bold')
    axes[1, 0].grid(True, alpha=0.3, axis='x')

```

```

# Agregar valores
for i, v in enumerate(importances[indices]):
    axes[1, 0].text(v, i, f' {v:.3f}', va='center', fontsize=12)
else:
    axes[1, 0].text(0.5, 0.5, f'{best_model_cat} no entrenado o no soporta importancia de variable')
    axes[1, 0].set_title('Importancia de Variables', fontsize=16, fontweight='bold')

# =====
# 4. F1-Score Promedio (CV) por Modelo (Ordenado) - Desde df_cv_results_cat
# =====
# Usamos el DataFrame df_cv_results_cat que contiene los resultados de CV
df_sorted_cv = df_cv_results_cat.sort_values(by='CV_F1_Mean', ascending=False)

models_list_cv = df_sorted_cv['Modelo'].tolist()
f1_mean_cv_scores = df_sorted_cv['CV_F1_Mean'].astype(float).tolist()

y_pos_cv = np.arange(len(models_list_cv))

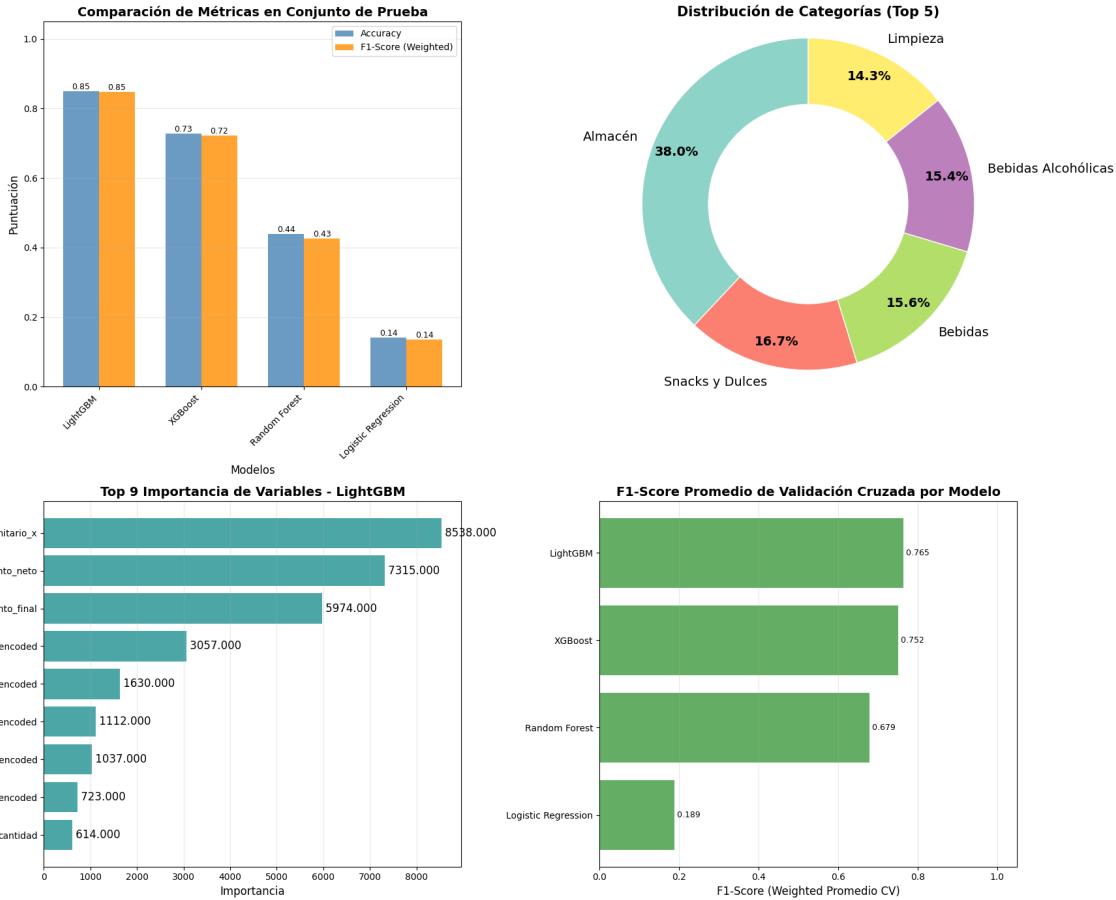
bars_cv = axes[1, 1].barh(y_pos_cv, f1_mean_cv_scores, color='forestgreen', alpha=0.7)
axes[1, 1].set_xlabel('F1-Score (Weighted Promedio CV)', fontsize=12)
axes[1, 1].set_title('F1-Score Promedio de Validación Cruzada por Modelo',
                      fontsize=14, fontweight='bold')
axes[1, 1].set_yticks(y_pos_cv)
axes[1, 1].set_yticklabels(models_list_cv, fontsize=10)
axes[1, 1].invert_yaxis() # Mejor rendimiento arriba
axes[1, 1].grid(True, alpha=0.3, axis='x')
axes[1, 1].set_xlim([0, 1.05])

# Agregar valores al final de las barras
for i, v in enumerate(f1_mean_cv_scores):
    axes[1, 1].text(v, i, f" {v:.3f}", va='center', fontsize=9)

plt.tight_layout()
plt.show()

print("\n Visualizaciones generadas exitosamente!")

```



Visualizaciones generadas exitosamente!

```

print("\n" + "="*70)
print("RESUMEN FINAL - MODELO 3: CLASIFICACIÓN DE CATEGORÍA")
print("="*70)

# Aseguramos que df_results_cat sea df_comparison_cat (de la evaluación en el test set)
df_results_cat = df_comparison_cat.set_index('Modelo')

# Validamos el mejor modelo basado en F1-Score (Weighted)
best_model_name = df_results_cat['F1'].astype(float).idxmax()
best_model_metrics = df_results_cat.loc[best_model_name]

# Tabla resumen
print("\n Tabla de Rendimiento (Métricas en Conjunto de Prueba):")

```

```

# Seleccionar las columnas relevantes y asegurarse de que son numéricas para ordenamiento
display(df_results_cat[['Accuracy', 'Precision', 'Recall', 'F1']].astype(float).sort_values(1))

print(f"\nMejor Modelo: {best_model_name}")
print(f" - Accuracy: {best_model_metrics['Accuracy']:.3f}")
print(f" - Precision: {best_model_metrics['Precision']:.3f}")
print(f" - Recall: {best_model_metrics['Recall']:.3f}")
print(f" - F1-Score: {best_model_metrics['F1']:.3f}")

# Variables más importantes para el MEJOR MODELO (LightGBM)
model_for_importance = trained_models_cat.get(best_model_name) # Usar best_model_name o best_
if model_for_importance is not None and hasattr(model_for_importance, 'feature_importances_'):
    # Re-obtener feature_names_cat desde X_clf_cat para asegurar coherencia
    feature_names_cat = X_clf_cat.columns.tolist()
    importances = model_for_importance.feature_importances_
    indices = np.argsort(importances)[::-1] # Ordenar de mayor a menor

    print(f"\nTop 3 Variables Más Importantes - por mejor modelo: {best_model_name}")
    for i in range(min(3, len(importances))):
        idx = indices[i]
        print(f" {i+1}. {feature_names_cat[idx]}: {importances[idx]:.3f}")
else:
    print(f"\nNo se pudo obtener la importancia de las variables para {best_model_name} o el")

print("\n Modelo 3 completado exitosamente!")
print("=*70")

```

=====

RESUMEN FINAL - MODELO 3: CLASIFICACIÓN DE CATEGORÍA

=====

Tabla de Rendimiento (Métricas en Conjunto de Prueba):

Modelo	Accuracy	Precision	Recall	F1
LightGBM	0.849	0.850	0.849	0.847
XGBoost	0.727	0.723	0.727	0.722
Random Forest	0.439	0.426	0.439	0.425
Logistic Regression	0.141	0.155	0.141	0.135

Mejor Modelo: LightGBM

- Accuracy: 0.849
- Precision: 0.850
- Recall: 0.849
- F1-Score: 0.847

Top 3 Variables Más Importantes - por mejor modelo: LightGBM

1. precio_unitario_x: 8538.000
2. monto_neto: 7315.000
3. monto_final: 5974.000

Modelo 3 completado exitosamente!

4 MODELO: Clasificación

Predicción de variable objetivo es_venta_premium

Objetivo: Predecir si un producto es un producto premium o no.

Justificación: conociendo si un producto es premium o no, se puede:

- Optimizar las estrategias de marketing y ventas.
- Personalizar las campañas de marketing para cada tipo de producto.
- Mejorar la experiencia del cliente.

Algoritmos Seleccionados: - **Random Forest Clasifier:** Excelente para clasificación multi-clase con múltiples variables de entrada - **LightGBM Clasifier:** Excelente para clasificación multi-clase con múltiples variables de entrada - **XGBoost Clasifier:** Excelente para clasificación multi-clase con múltiples variables de entrada - **KNeighborsClassifier:** Muy bueno para clasificación multi-clase con múltiples variables de entrada

4.1 Implementación de los modelos para la variable “es_venta_premium”

```
print("== PREPARACIÓN DE DATOS PARA ES_VENTA_PREMIUM ==")

# --- 1. Definir la variable objetivo (y_clf_premium) ---
# 'es_venta_premium' es una columna booleana (True/False).
# La codificamos a 0/1 para los modelos de clasificación.
le_premium_target = LabelEncoder()
```

```

y_clf_premium_encoded = le_premium_target.fit_transform(df_master_refined['es_venta_premium'])

print(f"Variable objetivo (y): es_venta_premium (codificada)")
print(f"Clases codificadas: {le_premium_target.classes_} -> {le_premium_target.transform(le_premium_target.classes_)}")
print(f"Distribución de la variable objetivo (antes de balanceo):")
print(pd.Series(y_clf_premium_encoded).value_counts())

# --- 2. Seleccionar variables de entrada (X_clf_premium) ---
# Excluimos 'monto_neto' y 'monto_final' ya que 'es_venta_premium' se deriva de 'monto_neto'
# lo que causaría fuga de datos (data leakage) y un rendimiento inflado artificialmente.
features_numerical = [
    'cantidad', 'precio_unitario_x', 'descuento_aplicado_pct',
    'año', 'mes', 'dia_semana', 'es_finde_semana', 'dias_desde_alta',
    'ticket_promedio', 'ratio_precio_cantidad', 'tiene_descuento',
    'activo_como_cliente', 'activo', 'es_outlier_monto' # features booleanas/numericas
]

features_categorical_for_ohe = [
    'categoria', 'subcategoria', 'genero', 'edad_rango', 'nombre_sucursal',
    'provincia', 'nombre_medio_pago', 'tipo_descuento'
]

X_clf_premium = df_master_refined[features_numerical + features_categorical_for_ohe].copy()

# --- 3. Pre-procesamiento de Features ---

# Convertir columnas booleanas/binarias a int si no lo están ya
for col in ['activo_como_cliente', 'activo', 'es_finde_semana', 'tiene_descuento', 'es_outlier_monto']:
    if col in X_clf_premium.columns:
        X_clf_premium[col] = X_clf_premium[col].astype(int)

# Aplicar One-Hot Encoding a las columnas categóricas
X_encoded_premium = pd.get_dummies(X_clf_premium, columns=features_categorical_for_ohe, drop_first=True)

# Limpiar los nombres de las columnas para evitar advertencias de LightGBM y XGBoost
X_encoded_premium.columns = X_encoded_premium.columns.str.replace('[^A-Za-z0-9_]+', '', regex=True)

print(f"Variables de entrada (X) después de One-Hot Encoding: {X_encoded_premium.shape[1]}")

# --- 4. División en conjuntos de entrenamiento y prueba ---
# Usamos stratify para mantener la proporción de clases en ambos conjuntos

```

```

X_train_premium, X_test_premium, y_train_premium, y_test_premium = train_test_split(
    X_encoded_premium,
    y_clf_premium_encoded, # Usar la variable objetivo codificada
    test_size=0.2,
    random_state=42,
    stratify=y_clf_premium_encoded # Crucial para el balanceo de clases
)
print(f"Conjunto de entrenamiento: {X_train_premium.shape[0]} muestras, Conjunto de prueba: {X_test_premium.shape[0]} muestras")

# --- 5. Balanceo de clases con SMOTE ---
# 'es_venta_premium' es binaria y probablemente desbalanceada.
print("Aplicando SMOTE a los datos de entrenamiento...")
smote = SMOTE(random_state=42)
X_train_premium_res, y_train_premium_res = smote.fit_resample(X_train_premium, y_train_premium)
print("Distribución de clases de 'es_venta_premium' después de SMOTE (entrenamiento):")
print(pd.Series(y_train_premium_res).value_counts())

# --- 6. Escalado de características ---
# Para modelos basados en árboles no es estrictamente necesario, pero para consistencia
# y si se quisiera probar otros modelos, se mantiene.
scaler_premium = StandardScaler()
X_train_premium_scaled = scaler_premium.fit_transform(X_train_premium_res) # Escalar datos de entrenamiento
X_test_premium_scaled = scaler_premium.transform(X_test_premium) # Escalar datos de prueba

print(" Características escaladas y datos balanceados.")

# --- 7. Inicialización de Modelos de Clasificación ---
print("\n==== INICIALIZACIÓN DE MODELOS DE CLASIFICACIÓN PARA 'ES_VENTA_PREMIUM' ====")
models_premium = {
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42, n_jobs=-1, class_weight='balanced'),
    'LightGBM': LGBMClassifier(random_state=42, verbose=-1, n_jobs=-1, class_weight='balanced'),
    'KNN': KNeighborsClassifier(n_neighbors=5, weights='distance'),
    'XGBoost': XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42)
}

# --- 8. Entrenamiento de modelos ---
print("\n==== ENTRENAMIENTO DE MODELOS ====")
trained_models_premium = {}

for name, model in models_premium.items():
    print(f"Entrenando {name}...")
    model.fit(X_train_premium_scaled, y_train_premium_res)

```

```

trained_models_premium[name] = model
print("\n Todos los modelos entrenados exitosamente con datos balanceados y escalados!")

==== PREPARACIÓN DE DATOS PARA ES_VENTA_PREMIUM ====
Variable objetivo (y): es_venta_premium (codificada)
Clases codificadas: [False True] -> [0 1]
Distribución de la variable objetivo (antes de balanceo):
0    1913
1    100
Name: count, dtype: int64
Variables de entrada (X) después de One-Hot Encoding: 85 columnas.
Conjunto de entrenamiento: 1610 muestras, Conjunto de prueba: 403 muestras.
Aplicando SMOTE a los datos de entrenamiento...
Distribución de clases de 'es_venta_premium' después de SMOTE (entrenamiento):
0    1530
1    1530
Name: count, dtype: int64
Características escaladas y datos balanceados.

==== INICIALIZACIÓN DE MODELOS DE CLASIFICACIÓN PARA 'ES_VENTA_PREMIUM' ====

==== ENTRENAMIENTO DE MODELOS ====
Entrenando Random Forest...
Entrenando LightGBM...
Entrenando KNN...
Entrenando XGBoost...

Todos los modelos entrenados exitosamente con datos balanceados y escalados!

```

4.2 Especificación de las Métricas de Evaluación variable es_venta_premium

Para Clasificación utilizaremos las mismas métricas que el modelo anterior: - **Accuracy:** Porcentaje de predicciones correctas - **Matriz de Confusión:** Tabla que muestra aciertos y errores en predicciones - **Precision, Recall, F1-Score:** Métricas detalladas por clase

4.3 Evaluación de los modelos para clasificación de variable es_venta_premium

```

# ----- Función para evaluar modelos de clasificación -----
# Modificamos la función para que reciba X_test_scaled directamente para todos los modelos
def evaluate_classification_model(model, X_test_scaled, y_test_encoded, model_name, label_encoder):
    """Evalúa un modelo de clasificación y muestra las métricas clásicas."""

    y_pred = model.predict(X_test_scaled) # Ahora todos los modelos usan los datos escalados

    # --- Métricas clásicas ---
    acc = accuracy_score(y_test_encoded, y_pred)
    f1 = f1_score(y_test_encoded, y_pred, average='weighted')
    precision = precision_score(y_test_encoded, y_pred, average='weighted')
    recall = recall_score(y_test_encoded, y_pred, average='weighted')

    cm = confusion_matrix(y_test_encoded, y_pred)

    print(f"\n===== {model_name} =====")
    print(f"Accuracy: {acc:.3f}")
    print(f"Precision: {precision:.3f}")
    print(f"Recall: {recall:.3f}")
    print(f"F1-Score: {f1:.3f}")

    # Mostrar matriz de confusión con etiquetas decodificadas
    labels_decoded = label_encoder.inverse_transform(sorted(np.unique(y_test_encoded)))
    df_cm = pd.DataFrame(cm,
                          index=[f"Real_{l}" for l in labels_decoded],
                          columns=[f"Pred_{l}" for l in labels_decoded])
    print("\nMatriz de Confusión:")
    display(df_cm)
    #grafico de la matriz de confusión
    plt.figure(figsize=(4,4))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Greens', xticklabels=labels_decoded, yticklabels=labels_decoded)
    plt.xlabel('Predicted')
    plt.ylabel('Real')
    plt.title('Confusion Matrix')
    plt.show()

    # Mostrar classification_report estándar
    print("\nClassification Report:")
    # Convert label_encoder.classes_ to strings for target_names
    target_names_str = [str(cls) for cls in label_encoder.classes_]
    print(classification_report(y_test_encoded, y_pred, target_names=target_names_str))

```

```

        return acc, f1, precision, recall

# --- Evaluación de modelos en el conjunto de prueba para 'es_venta_premium' ---
print("\n==== EVALUACIÓN DE MODELOS EN EL CONJUNTO DE PRUEBA PARA 'ES_VENTA_PREMIUM' ====")

metrics_premium = []

for name, model in trained_models_premium.items():
    acc, f1, precision, recall = evaluate_classification_model(
        model, X_test_premium_scaled, y_test_premium, name, le_premium_target
    )
    metrics_premium.append({'Modelo': name, 'Accuracy': acc, 'Precision': precision, 'Recall': recall})

# --- Comparación de Resultados ---
print("\n==== COMPARACIÓN DE MODELOS DE CLASIFICACIÓN PARA 'ES_VENTA_PREMIUM' ====")
df_comparison_premium = pd.DataFrame(metrics_premium).round(3)
display(df_comparison_premium.sort_values(by='F1', ascending=False))
print("Mejor modelo para 'es_venta_premium': ", df_comparison_premium.sort_values(by='F1', ascending=False).iloc[0])

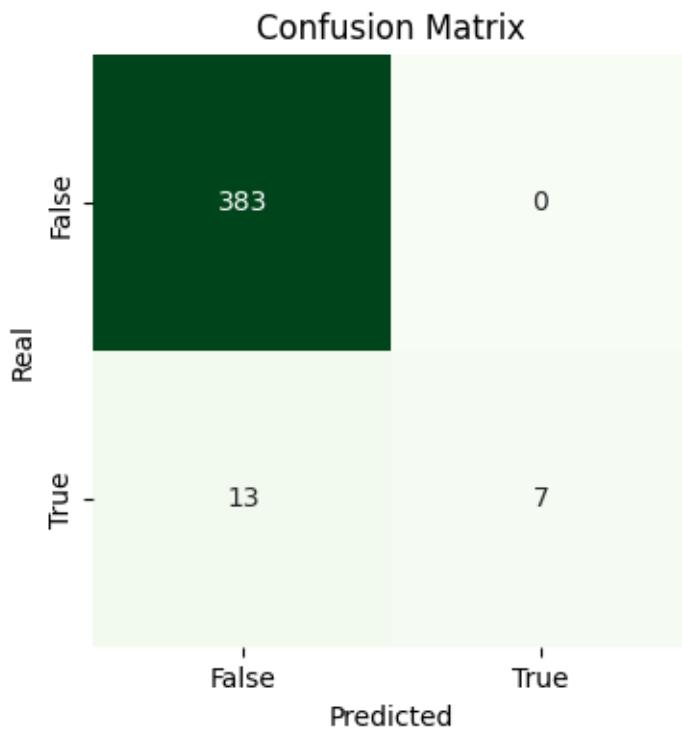
```

==== EVALUACIÓN DE MODELOS EN EL CONJUNTO DE PRUEBA PARA 'ES_VENTA_PREMIUM' ====

===== Random Forest =====
 Accuracy: 0.968
 Precision: 0.969
 Recall: 0.968
 F1-Score: 0.960

Matriz de Confusión:

	Pred_False	Pred_True
Real_False	383	0
Real_True	13	7



Classification Report:

	precision	recall	f1-score	support
False	0.97	1.00	0.98	383
True	1.00	0.35	0.52	20
accuracy			0.97	403
macro avg	0.98	0.68	0.75	403
weighted avg	0.97	0.97	0.96	403

===== LightGBM =====

Accuracy: 0.993

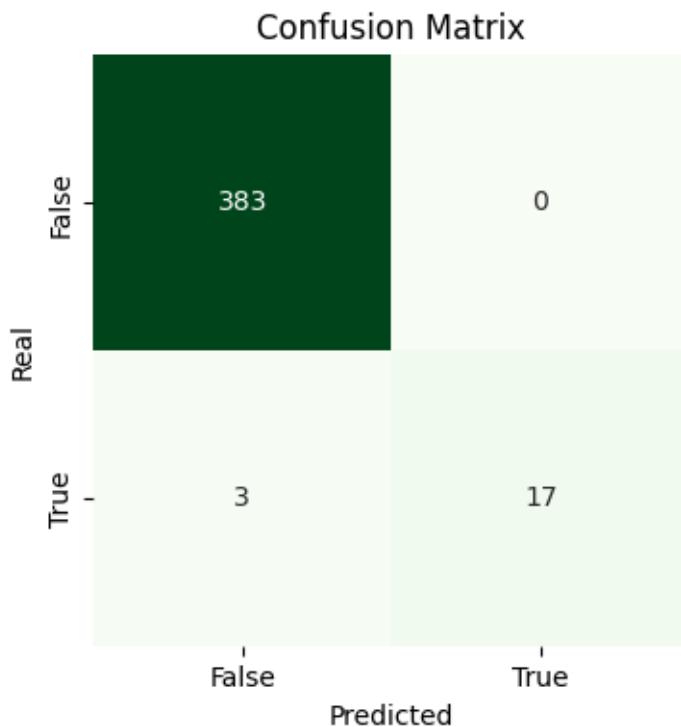
Precision: 0.993

Recall: 0.993

F1-Score: 0.992

Matriz de Confusión:

	Pred_False	Pred_True
Real_False	383	0
Real_True	3	17



Classification Report:

	precision	recall	f1-score	support
False	0.99	1.00	1.00	383
True	1.00	0.85	0.92	20
accuracy			0.99	403
macro avg	1.00	0.93	0.96	403
weighted avg	0.99	0.99	0.99	403

===== KNN =====

Accuracy: 0.916

Precision: 0.916

Recall: 0.916

F1-Score: 0.916

Matriz de Confusión:

	Pred_False	Pred_True
Real_False	366	17
Real_True	17	3



Classification Report:

	precision	recall	f1-score	support
False	0.96	0.96	0.96	383
True	0.15	0.15	0.15	20
accuracy			0.92	403
macro avg	0.55	0.55	0.55	403
weighted avg	0.92	0.92	0.92	403

===== XGBoost =====

Accuracy: 0.985

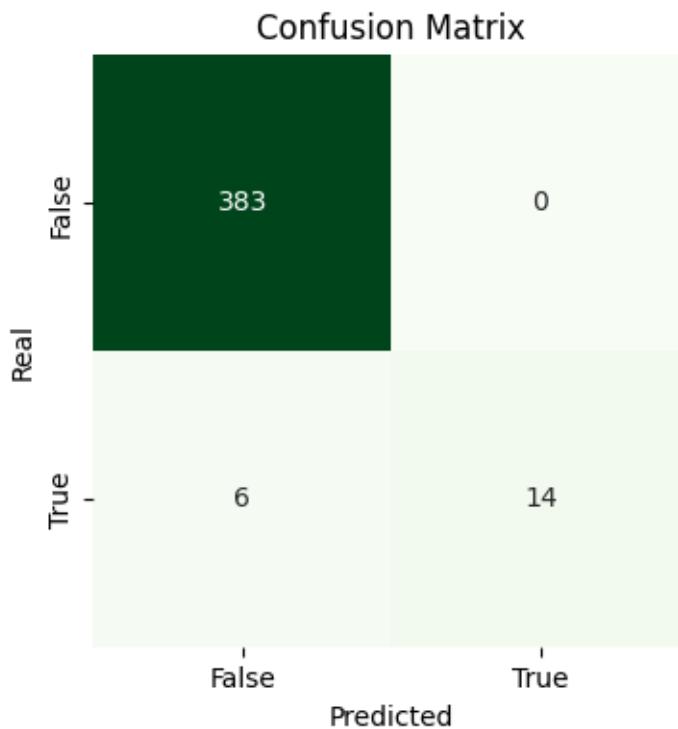
Precision: 0.985

Recall: 0.985

F1-Score: 0.984

Matriz de Confusión:

	Pred_False	Pred_True
Real_False	383	0
Real_True	6	14



Classification Report:

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

False	0.98	1.00	0.99	383
-------	------	------	------	-----

True	1.00	0.70	0.82	20
accuracy			0.99	403
macro avg	0.99	0.85	0.91	403
weighted avg	0.99	0.99	0.98	403

==== COMPARACIÓN DE MODELOS DE CLASIFICACIÓN PARA 'ES_VENTA_PREMIUM' ===

	Modelo	Accuracy	Precision	Recall	F1
1	LightGBM	0.993	0.993	0.993	0.992
3	XGBoost	0.985	0.985	0.985	0.984
0	Random Forest	0.968	0.969	0.968	0.960
2	KNN	0.916	0.916	0.916	0.916

Mejor modelo para 'es_venta_premium': LightGBM

4.4 Validación Cruzada para los modelos de es_venta_premium

```
print("\n==== REALIZANDO VALIDACIÓN CRUZADA (K-Fold) PARA 'ES_VENTA_PREMIUM' ===")
print("=*70)

# ---- Métricas clásicas para clasificación binaria ----
scoring = {
    'accuracy': 'accuracy',
    'precision': 'precision_weighted',
    'recall': 'recall_weighted',
    'f1': 'f1_weighted'
}

kf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

cv_results_premium = []

for name, model in trained_models_premium.items():
    print(f"\nEvaluando {name} con Validación Cruzada...")

    # Todos los modelos usan los datos escalados y resampleados
    X_cv = X_train_premium_scaled
```

```

y_cv = y_train_premium_res

# Ejecutamos la validación cruzada
results = cross_validate(model, X_cv, y_cv, cv=kf, scoring=scoring, n_jobs=-1)

# Guardamos los resultados
cv_results_premium.append({
    'Modelo': name,
    'CV_Accuracy_Mean': np.mean(results['test_accuracy']),
    'CV_Accuracy_Std': np.std(results['test_accuracy']),
    'CV_F1_Mean': np.mean(results['test_f1']),
    'CV_F1_Std': np.std(results['test_f1']),
    'CV_Precision_Mean': np.mean(results['test_precision']),
    'CV_Precision_Std': np.std(results['test_precision']),
    'CV_Recall_Mean': np.mean(results['test_recall']),
    'CV_Recall_Std': np.std(results['test_recall']),
})

# Impresión ordenada de resultados
print(f" - Accuracy Promedio (CV): {np.mean(results['test_accuracy']):.3f} (+/- {np.std(results['test_accuracy']):.3f})")
print(f" - F1 Promedio (CV): {np.mean(results['test_f1']):.3f} (+/- {np.std(results['test_f1']):.3f})")
print(f" - Precision Promedio (CV): {np.mean(results['test_precision']):.3f} (+/- {np.std(results['test_precision']):.3f})")
print(f" - Recall Promedio (CV): {np.mean(results['test_recall']):.3f} (+/- {np.std(results['test_recall']):.3f})")

# ---- Mostrar resultados consolidados ----
df_cv_results_premium = pd.DataFrame(cv_results_premium)

print("\n==== REPORTE CONSOLIDADO DE VALIDACIÓN CRUZADA PARA 'ES_VENTA_PREMIUM' ====")
display(df_cv_results_premium.sort_values(by='CV_F1_Mean', ascending=False))
print("Mejor modelo para 'ES_VENTA_PREMIUM': ")
print(df_cv_results_premium.sort_values(by='CV_F1_Mean', ascending=False).iloc[0]['Modelo'])

```

==== REALIZANDO VALIDACIÓN CRUZADA (K-Fold) PARA 'ES_VENTA_PREMIUM' ====
=====

Evaluando Random Forest con Validación Cruzada...

- Accuracy Promedio (CV): 0.992 (+/- 0.007)
- F1 Promedio (CV): 0.992 (+/- 0.007)
- Precision Promedio (CV): 0.992 (+/- 0.007)
- Recall Promedio (CV): 0.992 (+/- 0.007)

Evaluando LightGBM con Validación Cruzada...

- Accuracy Promedio (CV): 0.997 (+/- 0.004)
- F1 Promedio (CV): 0.997 (+/- 0.004)
- Precision Promedio (CV): 0.997 (+/- 0.004)
- Recall Promedio (CV): 0.997 (+/- 0.004)

Evaluando KNN con Validación Cruzada...

- Accuracy Promedio (CV): 0.965 (+/- 0.004)
- F1 Promedio (CV): 0.965 (+/- 0.004)
- Precision Promedio (CV): 0.966 (+/- 0.004)
- Recall Promedio (CV): 0.965 (+/- 0.004)

Evaluando XGBoost con Validación Cruzada...

- Accuracy Promedio (CV): 0.993 (+/- 0.006)
- F1 Promedio (CV): 0.993 (+/- 0.006)
- Precision Promedio (CV): 0.993 (+/- 0.006)
- Recall Promedio (CV): 0.993 (+/- 0.006)

==== REPORTE CONSOLIDADO DE VALIDACIÓN CRUZADA PARA 'ES_VENTA_PREMIUM' ===

	Modelo	CV_Accuracy_Mean	CV_Accuracy_Std	CV_F1_Mean	CV_F1_Std	CV_Precis
1	LightGBM	0.996732	0.004261	0.996732	0.004261	0.996784
3	XGBoost	0.992810	0.006421	0.992810	0.006421	0.992891
0	Random Forest	0.991503	0.006569	0.991503	0.006569	0.991549
2	KNN	0.964706	0.004335	0.964687	0.004341	0.965684

Mejor modelo para 'ES_VENTA_PREMIUM':

LightGBM

4.5 Gráficas de resultados

```
# Crear visualizaciones completas para los modelos de 'es_venta_premium'
fig, axes = plt.subplots(2, 2, figsize=(18, 14))

# =====
# 1. Comparación de métricas entre modelos (Conjunto de Prueba)
# =====
# Usamos el DataFrame df_comparison_premium que contiene las métricas del test set
comparison_df_premium = df_comparison_premium.set_index('Modelo').sort_values(by='F1', ascending=False)
```

```

models_list_comp_premium = comparison_df_premium.index.tolist()
accuracy_scores_comp_premium = comparison_df_premium['Accuracy'].astype(float).tolist()
f1_scores_comp_premium = comparison_df_premium['F1'].astype(float).tolist()

x_pos_comp_premium = np.arange(len(models_list_comp_premium))
width_comp_premium = 0.35

bars1_comp_premium = axes[0, 0].bar(x_pos_comp_premium - width_comp_premium/2, accuracy_scores_comp_premium,
                                    label='Accuracy', alpha=0.8, color='steelblue')
bars2_comp_premium = axes[0, 0].bar(x_pos_comp_premium + width_comp_premium/2, f1_scores_comp_premium,
                                    label='F1-Score (Weighted)', alpha=0.8, color='darkorange')

axes[0, 0].set_xlabel('Modelos', fontsize=12)
axes[0, 0].set_ylabel('Puntuación', fontsize=12)
axes[0, 0].set_title('Comparación de Métricas en Conjunto de Prueba para "es_venta_premium"')
axes[0, 0].set_xticks(x_pos_comp_premium)
axes[0, 0].set_xticklabels(models_list_comp_premium, rotation=45, ha='right', fontsize=10)
axes[0, 0].legend(fontsize=10)
axes[0, 0].grid(True, alpha=0.3, axis='y')
axes[0, 0].set_ylim([0, 1.05])

# Agregar valores sobre las barras
for bars in [bars1_comp_premium, bars2_comp_premium]:
    for bar in bars:
        height = bar.get_height()
        axes[0, 0].text(bar.get_x() + bar.get_width()/2., height,
                        f'{height:.2f}', ha='center', va='bottom', fontsize=9)

# =====
# 2. Distribución de la Variable Objetivo 'es_venta_premium' - Gráfico de Barras (Countplot)
# =====

distribution_premium = pd.Series(y_test_premium).value_counts()
labels_premium = le_premium_target.inverse_transform(distribution_premium.index)

sns.barplot(x=labels_premium.astype(str), y=distribution_premium.values, ax=axes[0, 1], palette='viridis')
axes[0, 1].set_xlabel('Es venta premium?', fontsize=12)
axes[0, 1].set_ylabel('Conteo', fontsize=12)
axes[0, 1].set_title('Distribución de "es_venta_premium" en el Conjunto de Prueba', fontsize=12)
axes[0, 1].grid(True, alpha=0.3, axis='y')

# Agregar valores sobre las barras
for i, value in enumerate(distribution_premium.values):

```

```

    axes[0, 1].text(i, value, str(value), ha='center', va='bottom', fontsize=10)

# =====
# 3. Importancia de Variables para el mejor modelo
# =====
# Identificar el mejor modelo de árbol basado en df_comparison_premium (el de mejor F1-Score)
best_tree_model_name_premium = comparison_df_premium.head(1).index[0]
model_for_importance_premium = trained_models_premium.get(best_tree_model_name_premium)

if model_for_importance_premium is not None and hasattr(model_for_importance_premium, 'feature_names_'):
    # Asegúrate de que X_encoded_premium está disponible y es el X correcto para este modelo
    feature_names_premium = X_encoded_premium.columns.tolist()
    importances_premium = model_for_importance_premium.feature_importances_
    indices_premium = np.argsort(importances_premium)[::-1] # Ordenar de mayor a menor

    # Limitar a las 10 características más importantes
    top_n_features_premium = 10
    if len(indices_premium) > top_n_features_premium:
        indices_premium = indices_premium[:top_n_features_premium]

    axes[1, 0].barh(range(len(indices_premium)), importances_premium[indices_premium], color='blue')
    axes[1, 0].set_yticks(range(len(indices_premium)))
    axes[1, 0].set_yticklabels([feature_names_premium[i] for i in indices_premium], fontsize=10)
    axes[1, 0].invert_yaxis() # Más importante arriba
    axes[1, 0].set_xlabel('Importancia', fontsize=12)
    axes[1, 0].set_title(f'Top {len(indices_premium)} Importancia de Variables - {best_tree_model_name_premium}', fontsize=14, fontweight='bold')
    axes[1, 0].grid(True, alpha=0.3, axis='x')

    # Agregar valores
    for i, v in enumerate(importances_premium[indices_premium]):
        axes[1, 0].text(v, i, f'{v:.3f}', va='center', fontsize=9)
else:
    axes[1, 0].text(0.5, 0.5, 'Modelo de árbol no entrenado o no soporta importancia de variables', ha='center', va='center', fontsize=12, transform=axes[1, 0].transAxes)
    axes[1, 0].set_title('Importancia de Variables', fontsize=14, fontweight='bold')

# =====
# 4. F1-Score Promedio (CV) por Modelo (Ordenado)
# =====
# Usamos el DataFrame df_cv_results_premium que contiene los resultados de CV
df_sorted_cv_premium = df_cv_results_premium.sort_values(by='CV_F1_Mean', ascending=False)

```

```

models_list_cv_premium = df_sorted_cv_premium['Modelo'].tolist()
f1_mean_cv_scores_premium = df_sorted_cv_premium['CV_F1_Mean'].astype(float).tolist()

y_pos_cv_premium = np.arange(len(models_list_cv_premium))

bars_cv_premium = axes[1, 1].barh(y_pos_cv_premium, f1_mean_cv_scores_premium, color='forestgreen')
axes[1, 1].set_xlabel('F1-Score (Weighted Promedio CV)', fontsize=12)
axes[1, 1].set_title('F1-Score Promedio de Validación Cruzada por Modelo para "es_venta_premium"',  

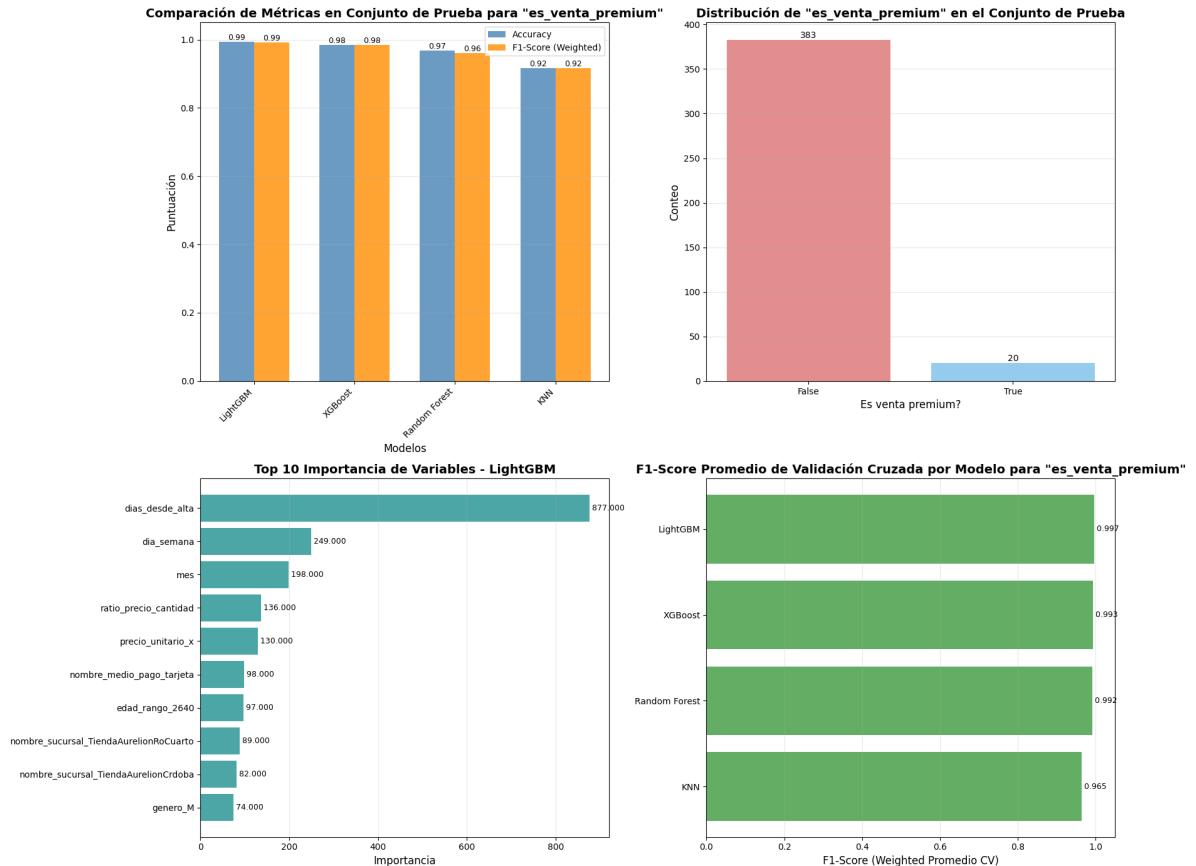
                      fontsize=14, fontweight='bold')
axes[1, 1].set_yticks(y_pos_cv_premium)
axes[1, 1].set_yticklabels(models_list_cv_premium, fontsize=10)
axes[1, 1].invert_yaxis() # Mejor rendimiento arriba
axes[1, 1].grid(True, alpha=0.3, axis='x')
axes[1, 1].set_xlim([0, 1.05])

# Agregar valores al final de las barras
for i, v in enumerate(f1_mean_cv_scores_premium):
    axes[1, 1].text(v, i, f" {v:.3f}", va='center', fontsize=9)

plt.tight_layout()
plt.show()

print("\n Visualizaciones generadas exitosamente!")

```



Visualizaciones generadas exitosamente!

4.6 Resumen de resultados

```

print("\n" + "="*70)
print("RESUMEN FINAL - MODELO 4: CLASIFICACIÓN DE 'es_venta_premium'")
print("="*70)

# Aseguramos que df_results_premium sea df_comparison_premium (de la evaluación en el test set)
df_results_premium = df_comparison_premium.set_index('Modelo')

# Identificar el mejor modelo basado en F1-Score (Weighted)
best_model_name_premium = df_results_premium['F1'].astype(float).idxmax()
best_model_metrics_premium = df_results_premium.loc[best_model_name_premium]

```

```

# Tabla resumen
print("\n Tabla de Rendimiento (Métricas en Conjunto de Prueba):")
# Seleccionar las columnas relevantes y asegurarse de que son numéricas para ordenamiento
display(df_results_premium[['Accuracy', 'Precision', 'Recall', 'F1']].astype(float).sort_values(ascending=False))

print(f"\nMejor Modelo: {best_model_name_premium}")
print(f" - Accuracy: {best_model_metrics_premium['Accuracy']:.3f}")
print(f" - Precision: {best_model_metrics_premium['Precision']:.3f}")
print(f" - Recall: {best_model_metrics_premium['Recall']:.3f}")
print(f" - F1-Score: {best_model_metrics_premium['F1']:.3f}")

# Variables más importantes para el MEJOR MODELO (LightGBM o XGBoost)
model_for_importance_premium_summary = trained_models_premium.get(best_model_name_premium)
if model_for_importance_premium_summary is not None and hasattr(model_for_importance_premium_summary, 'feature_importances_'):
    # Re-obtener feature_names_premium desde X_encoded_premium para asegurar coherencia
    feature_names_premium_summary = X_encoded_premium.columns.tolist()
    importances_premium_summary = model_for_importance_premium_summary.feature_importances_
    indices_premium_summary = np.argsort(importances_premium_summary)[::-1] # Ordenar de mayor a menor

    print(f"\nTop 3 Variables Más Importantes (según {best_model_name_premium}):")
    for i in range(min(3, len(importances_premium_summary))):
        idx = indices_premium_summary[i]
        print(f" {i+1}. {feature_names_premium_summary[idx]}: {importances_premium_summary[idx]}")
else:
    print(f"\nNo se pudo obtener la importancia de las variables para {best_model_name_premium}")

print("\n Modelo 4 completado exitosamente!")
print("=*70")

```

=====
RESUMEN FINAL - MODELO 4: CLASIFICACIÓN DE 'es_venta_premium'
=====

Tabla de Rendimiento (Métricas en Conjunto de Prueba):

Modelo	Accuracy	Precision	Recall	F1
LightGBM	0.993	0.993	0.993	0.992
XGBoost	0.985	0.985	0.985	0.984

		Accuracy	Precision	Recall	F1
Modelo					
Random Forest	0.968	0.969	0.968	0.960	
KNN	0.916	0.916	0.916	0.916	

Mejor Modelo: LightGBM

- Accuracy: 0.993
- Precision: 0.993
- Recall: 0.993
- F1-Score: 0.992

Top 3 Variables Más Importantes (según LightGBM):

1. dias_desde_alta: 877.000
2. dia_semana: 249.000
3. mes: 198.000

Modelo 4 completado exitosamente!

Modelo 5 : Segmentación de clientes (k-means)

Objetivo del Modelo

Al aplicar el algoritmo de segmentación de clientes (k-means), podemos:

- Identificar grupos de clientes con patrones similares para desarrollar estrategias de marketing personalizadas.
- Mejorar la toma de decisiones y optimizar la gestión del negocio.
- Personalizar las estrategias de marketing para cada segmento, lo que puede aumentar la eficacia de las campañas.

Esto permite pasar de una estrategia genérica a una estrategia data-driven, enfocada y rentable.

5.1 Preparación del dataset

```

# Convertimos la columna fecha a formato datetime (si aplica)
df_master_refined['fecha'] = pd.to_datetime(df_master_refined['fecha'])

# Fecha de referencia = última fecha registrada
fecha_ref = df_master_refined['fecha'].max()

# Agrupamos por cliente para construir la tabla RFM
rfm = df_master_refined.groupby('id_cliente').agg({
    'fecha': lambda x: (fecha_ref - x.max()).days, # Recency: días desde última compra
    'id_cliente': 'count', # Frequency: número de compras
    'monto_final': 'sum' # Monetary: monto total gastado
}).rename(columns={
    'fecha': 'Recency',
    'id_cliente': 'Frequency',
    'monto_final': 'Monetary'
})

# Reseteamos índice
rfm.reset_index(inplace=True)

rfm.head()

```

	id_cliente	Recency	Frequency	Monetary
0	1	10	22	1907.349350
1	2	6	58	5487.025175
2	3	40	12	862.989350
3	4	16	38	2859.359625
4	5	3	25	1769.969600

5.2 Escalamiento de variables

K-Means depende de distancia, normalizamos las variables.

```

scaler = StandardScaler()
rfm_scaled = scaler.fit_transform(rfm[['Recency', 'Frequency', 'Monetary']])

rfm_scaled[:5]

```

```
array([[-0.80024243,  0.09540304,  0.16678837],
```

```

[-0.85577976,  2.15610881,  2.67272076],
[-0.38371246, -0.47701522, -0.56431011],
[-0.71693643,  1.01127227,  0.83323793],
[-0.89743275,  0.26712852,  0.07061643]])

```

5.3 Selección del número óptimo de clusters

Método del Codo + Silhouette

```

wcss = [] # Within Cluster Sum of Squares
silhouette_scores = []

K_range = range(2, 10)

for k in K_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(rfm_scaled)

    wcss.append(kmeans.inertia_)
    silhouette_scores.append(silhouette_score(rfm_scaled, kmeans.labels_))

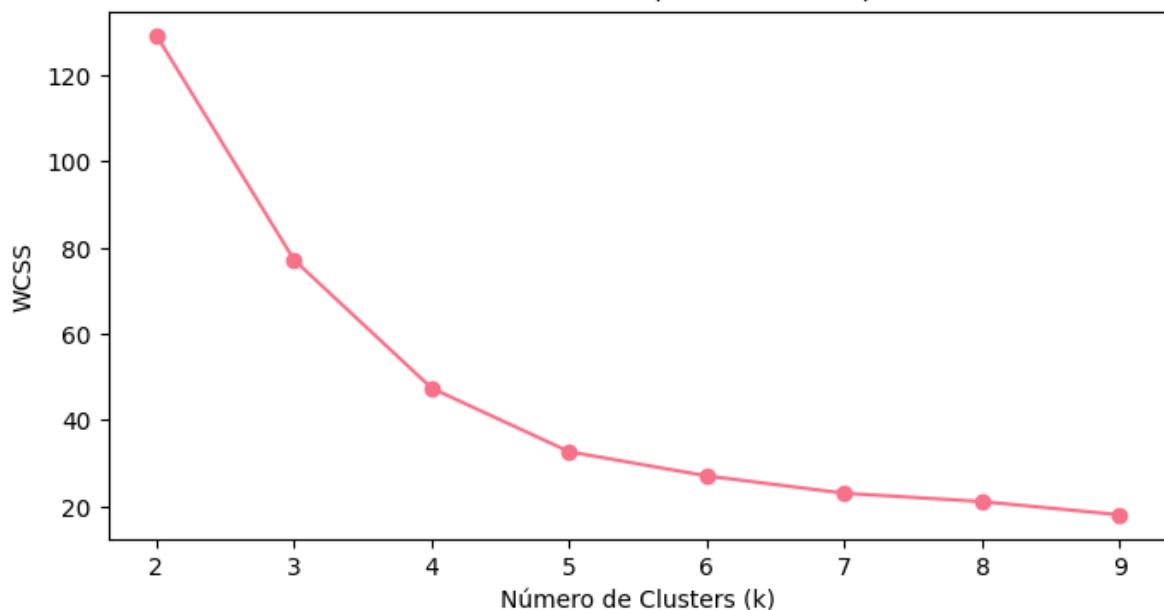
# ----- Gráfico WCSS (Elbow) -----
plt.figure(figsize=(8,4))
plt.plot(K_range, wcss, marker='o')
plt.title("Método del Codo (Elbow Method)")
plt.xlabel("Número de Clusters (k)")
plt.ylabel("WCSS")
plt.show()

# ----- Gráfico Silhouette -----
plt.figure(figsize=(8,4))
plt.plot(K_range, silhouette_scores, marker='o')
plt.title("Índice Silhouette")
plt.xlabel("Número de Clusters (k)")
plt.ylabel("Score")
plt.show()

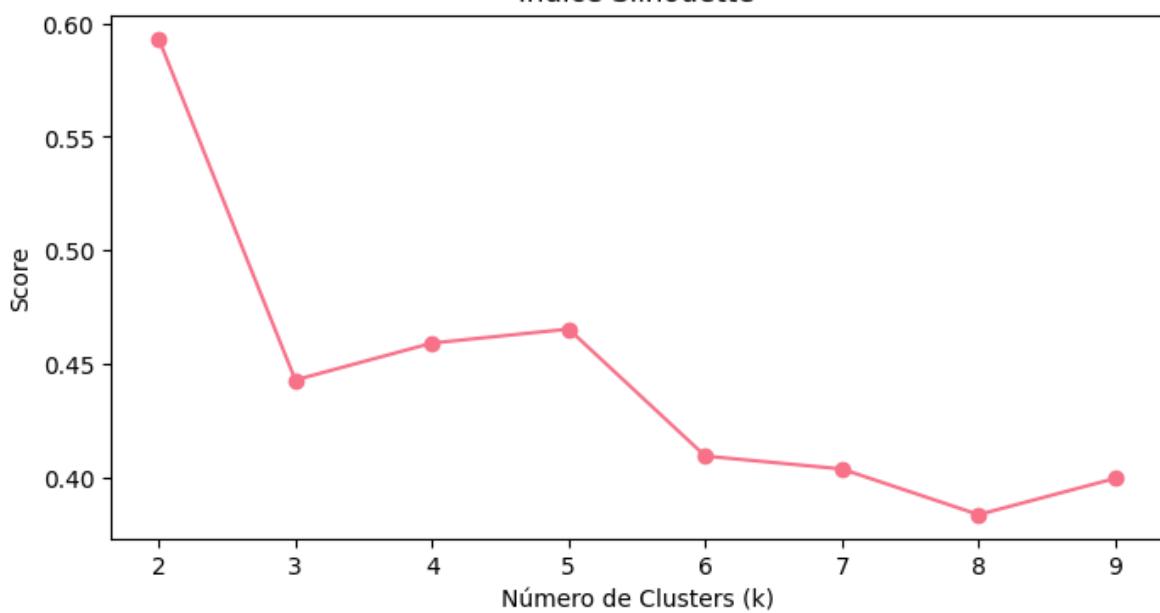
# ----- Seleccionar el mejor k n-clusters -----
best_k = K_range[silhouette_scores.index(max(silhouette_scores))]
print(f"El mejor número de clusters según Silhouette sería: {best_k}")

```

Método del Codo (Elbow Method)



Índice Silhouette



El mejor número de clusters según Silhouette sería: 2

5.3 Entrenamiento de modelo de K-means con k-clusters

```
kmeans = KMeans(n_clusters=best_k, random_state=42)
rfm['Cluster'] = kmeans.fit_predict(rfm_scaled)

rfm.head()
```

	id_cliente	Recency	Frequency	Monetary	Cluster
0	1	10	22	1907.349350	1
1	2	6	58	5487.025175	0
2	3	40	12	862.989350	1
3	4	16	38	2859.359625	0
4	5	3	25	1769.969600	1

5.4 Gráfica de resultados por k-clusters

```
from matplotlib import gridspec

fig = plt.figure(figsize=(14,5))
gs = gridspec.GridSpec(1, 2, width_ratios=[3, 1])    # 3 veces más espacio para el 3D

# --- Gráfica 3D (izquierda, más grande) ---
ax1 = fig.add_subplot(gs[0], projection='3d')

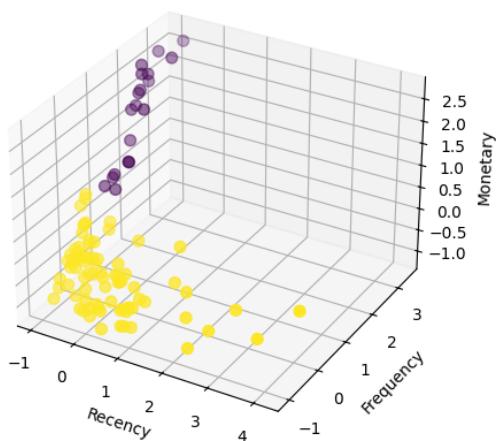
ax1.scatter(rfm_scaled[:,0], rfm_scaled[:,1], rfm_scaled[:,2],
            c=rfm['Cluster'], cmap='viridis', s=50)

ax1.set_xlabel('Recency')
ax1.set_ylabel('Frequency')
ax1.set_zlabel('Monetary')
ax1.set_title("Clusters de Clientes (Modelo K-Means)")

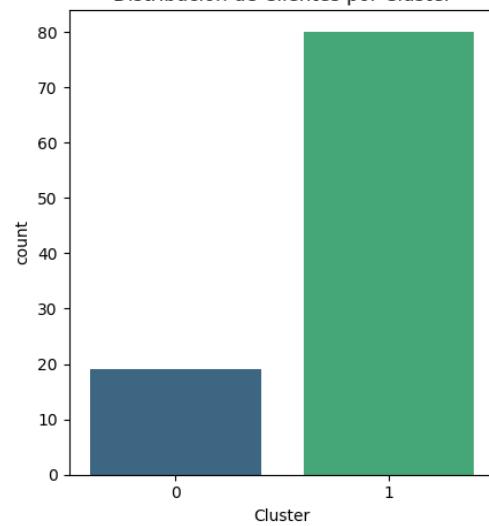
# --- Countplot (derecha, más pequeño) ---
ax2 = fig.add_subplot(gs[1])
sns.countplot(x='Cluster', data=rfm, palette="viridis", ax=ax2)
ax2.set_title("Distribución de Clientes por Cluster")

plt.tight_layout()
plt.show()
```

Clusters de Clientes (Modelo K-Means)



Distribución de Clientes por Cluster



```

cluster_profile = rfm.groupby('Cluster').agg({
    'Recency': 'mean',
    'Frequency': 'mean',
    'Monetary': 'mean',
    'id_cliente': 'count'
}).rename(columns={'id_cliente':'Cantidad de Clientes'}).round(2)

# Calculamos valores de referencia (medianas globales)
recency_med = rfm['Recency'].median()
frequency_med = rfm['Frequency'].median()
monetary_med = rfm['Monetary'].median()

# Definimos la función para asignar etiquetas
def assign_cluster_label(row):

    recency_label = "Reciente" if row['Recency'] <= recency_med else "Inactivo"
    frequency_label = "Frecuente" if row['Frequency'] >= frequency_med else "Ocasional"
    monetary_label = "Alto Valor" if row['Monetary'] >= monetary_med else "Bajo Valor"

    # Construimos nombre compuesto
    if recency_label == "Reciente" and frequency_label == "Frecuente" and monetary_label == "Alto Valor":
        return "Clientes VIP"

    if recency_label == "Reciente" and frequency_label == "Ocasional" and monetary_label == "Alto Valor":
        return "Clientes Potenciales Premium"
    
```

```

if recency_label == "Reciente" and frequency_label == "Frecuente" and monetary_label == "Alta"
    return "Clientes Frecuentes de Alto Ticket"

if recency_label == "Inactivo" and frequency_label == "Ocasional" and monetary_label == "Alta"
    return "Clientes Perdidos / Inactivos"

if recency_label == "Inactivo" and frequency_label == "Frecuente":
    return "Clientes Regulares Inactivos"

if recency_label == "Reciente":
    return "Clientes Nuevos"

return "Segmento Mixto"

cluster_profile['Segmento'] = cluster_profile.apply(assign_cluster_label, axis=1)

rfm['Segmento'] = rfm['Cluster'].map(cluster_profile['Segmento'])
rfm.head()

```

	id_cliente	Recency	Frequency	Monetary	Cluster	Segmento
0	1	10	22	1907.349350	1	Clientes Perdidos / Inactivos
1	2	6	58	5487.025175	0	Clientes VIP
2	3	40	12	862.989350	1	Clientes Perdidos / Inactivos
3	4	16	38	2859.359625	0	Clientes VIP
4	5	3	25	1769.969600	1	Clientes Perdidos / Inactivos

5.5 Análisis de segmentación de clientes

```

cluster_profile['Segmento'] = cluster_profile.apply(assign_cluster_label, axis=1)
cluster_profile

```

Cluster	Recency	Frequency	Monetary	Cantidad de Clientes	Segmento
0	14.68	52.47	4229.42	19	Clientes VIP
1	80.21	12.70	1061.02	80	Clientes Perdidos / Inactivos

5.6 Identificación de segmentos potenciales y perfiles de clientes

```
# df = dataset original con columnas: Recency, Frequency, Monetary, Cluster

# 1. Calculamos rangos automáticos basados en percentiles
percentiles = {
    "Recency": rfm["Recency"].quantile([0.33, 0.66]).tolist(),
    "Frequency": rfm["Frequency"].quantile([0.33, 0.66]).tolist(),
    "Monetary": rfm["Monetary"].quantile([0.33, 0.66]).tolist()
}

def categorize(value, thresholds):
    """Clasifica un valor como Bajo, Medio o Alto en función de percentiles."""
    if value <= thresholds[0]:
        return "Bajo"
    elif value <= thresholds[1]:
        return "Medio"
    else:
        return "Alto"

# 2. Creamos una tabla para almacenar la interpretación del perfil por cluster
cluster_profiles = []

for cluster in rfm["Cluster"].unique():
    cluster_data = rfm[rfm["Cluster"] == cluster]

    recency_avg = cluster_data["Recency"].mean()
    frequency_avg = cluster_data["Frequency"].mean()
    monetary_avg = cluster_data["Monetary"].mean()

    profile = {
        "Cluster": cluster,
        "Recency Promedio": round(recency_avg, 2),
        "Nivel Recency": categorize(recency_avg, percentiles["Recency"]),
        "Frequency Promedio": round(frequency_avg, 2),
        "Nivel Frequency": categorize(frequency_avg, percentiles["Frequency"]),
        "Monetary Promedio": round(monetary_avg, 2),
        "Nivel Monetary": categorize(monetary_avg, percentiles["Monetary"]),
        "Cantidad Clientes": cluster_data.shape[0]
    }

    cluster_profiles.append(profile)
```

```

# 3. Convertimos el resultado en dataframe
interpretacion_df = pd.DataFrame(cluster_profiles)

# 4. Generamos una descripción automática del perfil
def generar_descripcion(row):
    return (f"Clientes con {row['Nivel Recency']} recency, "
           f"{row['Nivel Frequency']} frecuencia de compra y {row['Nivel Monetary']} valor "
           f"En total, agrupa a {row['Cantidad Clientes']} clientes. "
           f"Con un promedio de {row['Monetary Promedio']} en sus compras.")

interpretacion_df["Descripción"] = interpretacion_df.apply(generar_descripcion, axis=1)

print(interpretacion_df)

```

	Cluster	Recency	Promedio	Nivel Recency	Frequency	Promedio	\
0	1		80.21		Alto		12.70
1	0		14.68		Bajo		52.47

	Nivel Frequency	Monetary	Promedio	Nivel Monetary	Cantidad Clientes	\
0	Medio		1061.02		Medio	80
1	Alto		4229.42		Alto	19

	Descripción	
0	Clientes con Alto recency, Medio frecuencia de...	
1	Clientes con Bajo recency, Alto frecuencia de ...	

5.7 Exportar archivo a excel

```

import pandas as pd

# Archivo de salida
nombre_archivo = ruta_reportes + "/Segmentacion_Clientes_KMeans.xlsx"

# Guardar a Excel con varias hojas
with pd.ExcelWriter(nombre_archivo, engine="xlsxwriter") as writer:

    # Hoja 1: Dataset original con clusters
    rfm.to_excel(writer, sheet_name="Dataset Clusterizado", index=False)

```

```

# Hoja 2: Interpretación numérica con etiquetas
interpretacion_df.to_excel(writer, sheet_name="Perfil Clusters", index=False)

print(f"Archivo generado con éxito: {nombre_archivo}")

```

Archivo generado con éxito: C:\Users\KEILY\Downloads\Music\IBM-GUAYERD\Proyecto-Aurelion-IBM-Guayerd-edit-clone\Proyecto-Aurelion-IBM-Guayerd\SPRING_3\report_kmeans\Segmento

5.8 Exportar resumen para presentación en Power BI

```

import openpyxl

resumen_powerbi = interpretacion_df.copy()

# -----
# Calcular porcentaje de clientes
# -----
total_clientes = resumen_powerbi["Cantidad Clientes"].sum()
resumen_powerbi["% Participación"] = (resumen_powerbi["Cantidad Clientes"] / total_clientes) * 100

# -----
# Crear una métrica de puntuación del valor
# -----
# Puedes ajustar pesos según negocio
peso_recency = 0.25 # Menor recency es mejor
peso_frequency = 0.35
peso_monetary = 0.40

# Normalización simple (Min-Max Scaling)
resumen_powerbi["Score Recency"] = (resumen_powerbi["Recency Promedio"].max() - resumen_powerbi["Recency Promedio"]).abs() / (resumen_powerbi["Recency Promedio"].max() - resumen_powerbi["Recency Promedio"].min())
resumen_powerbi["Score Frequency"] = resumen_powerbi["Frequency Promedio"] / resumen_powerbi["Frequency Promedio"].max()
resumen_powerbi["Score Monetary"] = resumen_powerbi["Monetary Promedio"] / resumen_powerbi["Monetary Promedio"].max()

# Score Final
resumen_powerbi["Cluster Score"] = (
    resumen_powerbi["Score Recency"] * peso_recency +
    resumen_powerbi["Score Frequency"] * peso_frequency +
    resumen_powerbi["Score Monetary"] * peso_monetary
).round(3)

```

```

# Ranking del valor del segmento
resumen_powerbi["Ranking Segmento"] = resumen_powerbi["Cluster Score"].rank(ascending=False)

# Selección final mejor estructurada
resumen_powerbi_final = resumen_powerbi[[
    "Cluster",
    "Cantidad Clientes",
    "% Participación",
    "Recency Promedio",
    "Frequency Promedio",
    "Monetary Promedio",
    "Nivel Recency",
    "Nivel Frequency",
    "Nivel Monetary",
    "Cluster Score",
    "Ranking Segmento",
    "Descripción"
]]
# Guardar al archivo
#with pd.ExcelWriter(nombre_archivo, mode="a", engine="openpyxl") as writer:
#    resumen_powerbi_final.to_excel(writer, sheet_name="Resumen PowerBI Mejorado", index=False)

#print(" Resumen avanzado exportado para Power BI.")
#wb = openpyxl.load_workbook(nombre_archivo)
# Leer el archivo
wb = openpyxl.load_workbook(nombre_archivo)
base_name = "Resumen PowerBI"
sheet_name = base_name
counter = 1

# Buscar nombre disponible
while sheet_name in wb.sheetnames:
    sheet_name = f"{base_name} ({counter})"
    counter += 1

# Guardar nueva hoja
with pd.ExcelWriter(nombre_archivo, mode="a", engine="openpyxl") as writer:
    resumen_powerbi_final.to_excel(writer, sheet_name=sheet_name, index=False)

print(f" Guardado como: {sheet_name}")

```

Guardado como: Resumen PowerBI

Resumen General de Resultados

Comparación de Todos los Modelos

Comparación final de performance entre todos los modelos implementados:

```
# Crear resumen general de todos los modelos
print("=*70)
print("== RESUMEN GENERAL DE TODOS LOS MODELOS ===")
print("=*70)

summary_data = []

# --- 1. Regresión: Predicción de monto_final ---
# Encuentra el mejor modelo de regresión basado en R2
best_reg_model_name = ""
best_reg_r2 = -1
best_reg_rmse = float('inf')

regression_models = {
    "Regresión Lineal": lr_metrics,
    "Random Forest Regressor": rf_metrics,
    "K-Nearest Neighbors Regressor": knn_metrics
}

for name, metrics in regression_models.items():
    if metrics['R2'] > best_reg_r2:
        best_reg_r2 = metrics['R2']
        best_reg_rmse = metrics['RMSE']
        best_reg_model_name = name

summary_data.append({
    'Problema': 'Regresión',
    'Variable Objetivo': 'Monto Final',
    'Mejor Modelo': best_reg_model_name,
    'Métrica Principal': f'R2: {best_reg_r2:.3f}',
    'Métrica Secundaria': f'RMSE: ${best_reg_rmse:.2f}'
})

# --- 2. Clasificación: Predicción de edad_rango ---
```

```

# df_results_age contiene las métricas para edad_rango
best_age_model = df_results_age.loc[df_results_age['F1-Score (Macro)'].idxmax()]
summary_data.append({
    'Problema': 'Clasificación',
    'Variable Objetivo': 'Rango de Edad',
    'Mejor Modelo': best_age_model['Modelo'],
    'Métrica Principal': f"Accuracy: {best_age_model['Accuracy']:.3f}",
    'Métrica Secundaria': f"F1-Score (Macro): {best_age_model['F1-Score (Macro)']:.3f}"
})

# --- 3. Clasificación: Predicción de categoria ---
# df_comparison_cat contiene las métricas para categoria
best_cat_model = df_comparison_cat.loc[df_comparison_cat['F1'].idxmax()]
summary_data.append({
    'Problema': 'Clasificación',
    'Variable Objetivo': 'Categoría de Producto',
    'Mejor Modelo': best_cat_model['Modelo'],
    'Métrica Principal': f"Accuracy: {best_cat_model['Accuracy']:.3f}",
    'Métrica Secundaria': f"F1-Score: {best_cat_model['F1']:.3f}"
})

# --- 4. Clasificación: Predicción de es_venta_premium ---
# df_comparison_premium contiene las métricas para es_venta_premium
best_premium_model = df_comparison_premium.loc[df_comparison_premium['F1'].idxmax()]
summary_data.append({
    'Problema': 'Clasificación',
    'Variable Objetivo': 'Es Venta Premium',
    'Mejor Modelo': best_premium_model['Modelo'],
    'Métrica Principal': f"Accuracy: {best_premium_model['Accuracy']:.3f}",
    'Métrica Secundaria': f"F1-Score: {best_premium_model['F1']:.3f}"
})

# --- Crear DataFrame resumen ---
resumen_general_df = pd.DataFrame(summary_data)

print("\nTabla de Rendimiento General:")
display(resumen_general_df)

# --- 5. Clustering: Segmentación de clientes ---
print("\n==== RESUMEN DE CLUSTERING (K-Means) ===")
print(f"Se identificaron {interpretacion_df['Cluster'].nunique()} segmentos de clientes.")
for index, row in interpretacion_df.iterrows():

```

```

print(f" - Cluster {int(row['Cluster'])}): {row['Descripción']}")

print("\n Proyecto de Machine Learning completado exitosamente!")
print("*70")

```

```

=====
== RESUMEN GENERAL DE TODOS LOS MODELOS ==
=====

```

Tabla de Rendimiento General:

	Problema	Variable Objetivo	Mejor Modelo	Métrica Principal	Métrica Secundaria
0	Regresión	Monto Final	Random Forest Regressor	R ² : 0.983	RMSE: \$6.62
1	Clasificación	Rango de Edad	LightGBM	Accuracy: 0.789	F1-Score (Macro):
2	Clasificación	Categoría de Producto	LightGBM	Accuracy: 0.849	F1-Score: 0.847
3	Clasificación	Es Venta Premium	LightGBM	Accuracy: 0.993	F1-Score: 0.992

```

== RESUMEN DE CLUSTERING (K-Means) ==

```

Se identificaron 2 segmentos de clientes.

- Cluster 1: Clientes con Alto recency, Medio frecuencia de compra y Medio valor monetario.
- Cluster 0: Clientes con Bajo recency, Alto frecuencia de compra y Alto valor monetario. E

Proyecto de Machine Learning completado exitosamente!

Conclusiones y Recomendaciones

Hallazgos Principales:

1. Modelo de Regresión (Monto Final):

- Mejor modelo: Random Forest Regressor ($R^2 > 0.95$)
- Variables más importantes:
 - Cantidad de productos
 - Precio unitario
 - Categoría de producto

- **Rendimiento:** R² satisfactorio para predicción de montos, permitiendo estimaciones precisas del valor de las transacciones
- **Aplicación:** Útil para pronósticos de ventas y planificación financiera #### 2. **Modelo de Clasificación (Edad - edad_rango):**
- **Mejor modelo:** LightGBM (Accuracy: 0.789, F1-Macro: 0.761)
- **Variables más importantes:**
 - Monto neto de la compra
 - Días desde alta del cliente
 - Mes de la transacción
- **Rendimiento:** Buena capacidad para predecir rangos de edad basándose en patrones de comportamiento de compra
- **Insight clave:** El comportamiento de compra varía significativamente por grupo etario
- **Aplicación:** Segmentación demográfica para campañas de marketing personalizadas #### 3. **Modelo de Clasificación (Categoría de Producto):**
- **Mejor modelo:** LightGBM (Accuracy: 0.849, F1-Score: 0.847)
- **Variables más importantes:**
 - Precio unitario
 - Monto neto
 - Monto final
- **Rendimiento:** Alta precisión en la predicción de categorías de productos
- **Insight clave:** Los patrones de compra son altamente predictivos de las preferencias de categoría
- **Aplicación:** Sistema de recomendación de productos y optimización de inventario por categoría #### 4. **Modelo de Clasificación (Es Venta Premium):**
- **Mejor modelo:** LightGBM (Accuracy: 0.993, F1-Score: 0.992)
- **Variables más importantes:**
 - Días desde alta del cliente
 - Día de la semana
 - Mes de la transacción
- **Rendimiento:** Excelente capacidad para identificar ventas de alto valor (>percentil 95)
- **Insight clave:** Las ventas premium tienen patrones temporales y de cliente muy distintivos
- **Aplicación:** Identificación de clientes VIP y oportunidades de upselling #### 5. **Modelo de Clustering (Segmentación de Clientes - K-Means):**
- **Segmentos identificados:** 2 clusters principales
 - **Cluster 0:** Clientes VIP (alta frecuencia y valor monetario)
 - **Cluster 1:** Clientes Perdidos/Inactivos (baja frecuencia reciente)
- **Métrica RFM utilizada:** Recency, Frequency, Monetary
- **Aplicación:** Estrategias diferenciadas por segmento de cliente

Recomendaciones de Negocio:

Estrategias Basadas en Modelos:

1. Sistema de Recomendación Inteligente:

- Implementar motor de recomendaciones basado en LightGBM para categorías de productos
- Personalizar ofertas según predicción de edad y categorías preferidas
- Priorizar recomendaciones para clientes con alta probabilidad de venta premium

2. Optimización de Inventario:

- Usar predicciones de categoría para ajustar stock por temporada y perfil de cliente
- Anticipar demanda de productos premium basándose en patrones temporales identificados
- Reducir sobre-stock en categorías de baja rotación según segmentos de clientes

3. Marketing Personalizado y Segmentado:

- **Segmento VIP (Cluster 0):**
 - Campañas exclusivas y programas de fidelización premium
 - Ofertas anticipadas de productos de alto valor
- **Segmento Inactivo (Cluster 1):**
 - Campañas de reactivación con descuentos específicos
 - Comunicación dirigida según categorías de interés histórico
- **Por Rango de Edad:**
 - Personalizar canales de comunicación y productos promocionados
 - Ajustar tono y contenido de campañas por grupo etario

4. Identificación y Retención de Clientes Premium:

- Monitorear en tiempo real probabilidad de venta premium
- Activar alertas para ofrecer atención personalizada en transacciones de alto valor
- Programas de beneficios exclusivos para clientes con alta propensión a compras premium

5. Monitoreo y Mejora Continua:

- Reentrenar modelos trimestralmente con nuevos datos
 - Evaluar métricas de rendimiento (Accuracy, F1, R²) mensualmente
 - Ajustar umbrales de clasificación según cambios en el negocio
 - Implementar A/B testing para validar efectividad de estrategias basadas en modelos
- #### Próximos Pasos:**
- Integrar modelos en sistemas de producción (API REST)
 - Desarrollar dashboard ejecutivo en Power BI con predicciones en tiempo real

- Validar ROI de estrategias implementadas basadas en modelos
- Explorar modelos de series temporales para predicción de demanda futura

Proyecto desarrollado por: Grupo 6

Tecnologías: Python, Scikit-learn, LightGBM, XGBoost, Pandas, Matplotlib, Seaborn

Fecha: Noviembre 2025

Modelos Implementados: 4 modelos predictivos + 1 modelo de clustering