Online Retail II - Assignment 01 - Machine Learning Analysis

Student: Rellika Kisyula
Date: November 6, 2025

---

1. Data Overview

Dataset Source: UCI Online Retail II

1.1 Dataset Overview

The **Online Retail II** dataset contains transactional data from a UK-based online retailer specializing in gift and home products. The data covers transactions from **December 2009 to December 2011**.

1.2 Data Dictionary

Variable	Type	Description	Business Meaning
`InvoiceNo`	Categorical	Transaction number	Unique sale transaction ID
`StockCode`	Categorical	Product code	Unique identifier for each product
`Description`	Text	Product name	Product details for analysis
`Quantity`	Numeric	Units purchased	Measures demand volume
`InvoiceDate`	Datetime	Date and time of transaction	Used for trend, seasonality, and recency
`Price`	Numeric	Price per unit (in GBP)	Indicates pricing strategy
`Customer ID`	Categorical	Unique customer identifier	Enables customer segmentation & prediction
`Country`	Categorical	Country of customer	Geographic market analysis

---

2.THE BUSINESS CHALLENGE

2.1 Business Problem 1: Customer Churn Prediction (Classification)

- PROBLEM: Predict which customers will NOT return within 90 days - **Based on a customer's past behavior, can we predict if they will return in the next 90 days?**
- APPROACH: Binary classification using temporal split
- TARGET: will_return_90days (0 = Churned, 1 = Returned)

Variables

1. **Dependent Variable (DV):** `will_return_90days` (0 = Churned, 1 = Returned)
2. **Independent Variables (IVs):**
   - `recency_days` : Days since last purchase
   - `frequency` : Number of transactions
   - `avg_order_value` : Average spending per transaction
   - `purchase_consistency` : Std of days between purchases
   - `total_unique_products` : Product variety
   - `country` : Geographic location
   - `tenure_days` : Customer lifetime
3. **Model Type:**
   A. Logistic Regression (baseline, interpretable)
   B. Random Forest Classifier (non-linear patterns)

2.2 Business Problem 3: Product Recommendation System (Association Rules Mining)

PROBLEM: Find products frequently bought together to improve cross-selling APPROACH: Market Basket Analysis using Apriori algorithm OUTPUT: Association rules (if X purchased, then Y likely purchased)

TECHNIQUES:

- Apriori Algorithm: Find frequent itemsets
- Association Rules: Generate if-then relationships
- Metrics: Support, Confidence, Lift

BUSINESS VALUE:

- Product bundling strategies
- Store layout optimization
- Personalized recommendations
- Cross-selling opportunities

---

3. Data Acquisition and Loading

In [1]:
```
# important libraries to install
# !pip install pyjanitor plotnine pandas matplotlib numpy openpyxl pyarrow
```

In [124…
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import math
# !pip install pyjanitor
import janitor
import shap


from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, confusion_matrix, classification_report, roc_curve, ConfusionMat
)
from sklearn.metrics import silhouette_score, davies_bouldin_score


from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.cluster import KMeans


# If not installed, run these once in a cell:
# !pip install imbalanced-learn xgboost shap lifelines
from imblearn.over_sampling import SMOTE
from imblearn.pipeline import Pipeline as ImbPipeline
from xgboost import XGBClassifier

# Note: mlxtend is required for Apriori
# Install:
# !pip install mlxtend
from mlxtend.frequent_patterns import apriori, association_rules
from mlxtend.preprocessing import TransactionEncoder

warnings.filterwarnings("ignore")
sns.set(style="whitegrid")
```

3.1 Loading the Data

In [3]:
```
def read_retail_data():
    # Load the dataset from both sheets
    file_path = "data/online_retail_II.xlsx"

    print("Loading data from both sheets...")
    print("=" * 60)
```

```
    # Read Year 2009-2010 sheet
    df_2009_2010 = pd.read_excel(file_path, sheet_name='Year 2009-2010')
    print(f"Year 2009-2010 Sheet: {df_2009_2010.shape[0]:,} rows")

    # Read Year 2010-2011 sheet
    df_2010_2011 = pd.read_excel(file_path, sheet_name='Year 2010-2011')
    print(f"Year 2010-2011 Sheet: {df_2010_2011.shape[0]:,} rows")

    # Combine both sheets
    df = pd.concat([df_2009_2010, df_2010_2011], ignore_index=True)

    print("=" * 60)
    print(f"\nCombined Dataset Shape: {df.shape}")
    print(f"Total Transactions: {df.shape[0]:,}")
    print(f"Total Features: {df.shape[1]}")
    print(f"\nDate Range: {df['InvoiceDate'].min()} to {df['InvoiceDate'].max()}")
    return df.to_csv("data/online_retail_II_combined.csv", index=False)

# Load the data
# df = read_retail_data()
```

In [4]:
```
# Load from CSV for faster access in future
original_df = pd.read_csv("data/online_retail_II_combined.csv", parse_dates=['Invoi
```

In [5]:
```
df = original_df.copy()
```

3.2 Data Quality Assessment

In [6]:
```
# Display first few rows
df.head(10)
```

Out[6]:

	Invoice	StockCode	Description	Quantity	InvoiceDate	Price	Customer ID	Country
0	489434	85048	15CM CHRISTMAS GLASS BALL 20 LIGHTS	12	2009-12-01 07:45:00	6.95	13085.0	United Kingdom
1	489434	79323P	PINK CHERRY LIGHTS	12	2009-12-01 07:45:00	6.75	13085.0	United Kingdom
2	489434	79323W	WHITE CHERRY LIGHTS	12	2009-12-01 07:45:00	6.75	13085.0	United Kingdom
3	489434	22041	RECORD FRAME 7" SINGLE SIZE	48	2009-12-01 07:45:00	2.10	13085.0	United Kingdom
4	489434	21232	STRAWBERRY CERAMIC TRINKET BOX	24	2009-12-01 07:45:00	1.25	13085.0	United Kingdom
5	489434	22064	PINK DOUGHNUT TRINKET POT	24	2009-12-01 07:45:00	1.65	13085.0	United Kingdom
6	489434	21871	SAVE THE PLANET MUG	24	2009-12-01 07:45:00	1.25	13085.0	United Kingdom
7	489434	21523	FANCY FONT HOME SWEET HOME DOORMAT	10	2009-12-01 07:45:00	5.95	13085.0	United Kingdom
8	489435	22350	CAT BOWL	12	2009-12-01 07:46:00	2.55	13085.0	United Kingdom
9	489435	22349	DOG BOWL , CHASING BALL DESIGN	12	2009-12-01 07:46:00	3.75	13085.0	United Kingdom

In [7]:
```
# Summary statistics for numeric columns
df[['Quantity', 'Price']].describe().T
```

Out[7]:

	count	mean	std	min	25%	50%	75%	max
Quantity	1067371.0	9.938898	172.705794	-80995.00	1.00	3.0	10.00	80995.0
Price	1067371.0	4.649388	123.553059	-53594.36	1.25	2.1	4.15	38970.0

In [8]:
```
# Check missing values
missing_data = df.isnull().sum().sort_values(ascending=False)
missing_percent = (df.isnull().sum() / len(df) * 100).sort_values(ascending=False)
```

```
missing_summary = pd.DataFrame({
    'Missing Count': missing_data,
    'Percentage': missing_percent
})

print("\nMissing Values Summary:")
missing_summary[missing_summary['Missing Count'] > 0]
```

```
Missing Values Summary:
```

Out[8]:

	Missing Count	Percentage
Customer ID	243007	22.766873
Description	4382	0.410541

In [9]:
```
# Check for negative values in Quantity and Price
negative_quantity = (df['Quantity'] < 0).sum()
negative_price = (df['Price'] < 0).sum()

print(f"\nNegative Quantity Records: {negative_quantity}")
print(f"Negative Price Records: {negative_price}")
```

```
Negative Quantity Records: 22950
Negative Price Records: 5
```

In [10]:
```
# check for duplicate rows
duplicate_rows = df.duplicated().sum()

print(f"\nDuplicate Rows: {duplicate_rows}")
```

```
Duplicate Rows: 34335
```

Data Quality Issues:

- ~22% of records are missing CustomerID (likely guest purchases)
- Small percentage missing Description
- Negative values in Quantity and UnitPrice indicate returns/cancellations
- Some duplicate records present around 34335 records

In [11]:
```
# Check for unique values in categorical columns
print(f"  Unique Invoices: {df['Invoice'].nunique():,}")
```

```
 Unique Invoices: 53,628
```

In [12]:
```
print(f"  Unique Products: {df['StockCode'].nunique():,}")
```

```
 Unique Products: 5,305
```

In [13]:
```
print(f"  Unique Customers: {df['Customer ID'].nunique():,}")
```

```
 Unique Customers: 5,942
```

In [14]:
```
print(f"  Countries: {df['Country'].nunique()}")
```

```
 Countries: 43
```

In [15]:
```
print(f"\nDate Range: {df['InvoiceDate'].min()} to {df['InvoiceDate'].max()}")
```

```
Date Range: 2009-12-01 07:45:00 to 2011-12-09 12:50:00
```

In [16]:
```
# Top 10 countries by transaction count
print("Top 10 Countries by Transaction Count:")
df['Country'].value_counts().head(10)
```

```
Top 10 Countries by Transaction Count:
```

Out[16]:
```
Country
United Kingdom    981330
EIRE               17866
Germany            17624
France             14330
Netherlands         5140
Spain               3811
Switzerland         3189
Belgium             3123
Portugal            2620
Australia           1913
Name: count, dtype: int64
```

3.3 Data Cleaning Strategy

1. Remove cancellations (Invoice starts with 'C')
2. Remove negative quantities (returns)
3. Remove zero/negative prices (errors)
4. Remove missing CustomerIDs (can't track behavior)
5. Remove missing descriptions

In [17]:
```
## use janitor to clean column names
df = df.clean_names()

# rename invoicedaate to invoice_date
df = df.rename(columns={'invoicedate': 'invoice_date'})

# customer_id is category column
df['customer_id'] = df['customer_id'].astype('category')
# trim whitespace from description
df['description'] = df['description'].str.strip()

df.head()
```

Out[17]:

	invoice	stockcode	description	quantity	invoice_date	price	customer_id	country
0	489434	85048	15CM CHRISTMAS GLASS BALL 20 LIGHTS	12	2009-12-01 07:45:00	6.95	13085.0	United Kingdom
1	489434	79323P	PINK CHERRY LIGHTS	12	2009-12-01 07:45:00	6.75	13085.0	United Kingdom
2	489434	79323W	WHITE CHERRY LIGHTS	12	2009-12-01 07:45:00	6.75	13085.0	United Kingdom
3	489434	22041	RECORD FRAME 7" SINGLE SIZE	48	2009-12-01 07:45:00	2.10	13085.0	United Kingdom
4	489434	21232	STRAWBERRY CERAMIC TRINKET BOX	24	2009-12-01 07:45:00	1.25	13085.0	United Kingdom

In [18]:
```
# Remove duplicates
df = df.drop_duplicates()

# see if there are any duplicates left
df.duplicated().sum()
```

Out[18]:
```
np.int64(0)
```

In [19]:
```
# Remove missing values from customer_id and description
df = df.dropna(subset=['customer_id', 'description'])

# Check missing values again
df.isnull().sum()
```

Out[19]:
```
invoice         0
stockcode       0
description     0
quantity        0
invoice_date    0
price           0
customer_id     0
country         0
dtype: int64
```

In [20]:
```
# Remove cancellations and invalid values
# Remove cancellations first
df = df[~df['invoice'].astype(str).str.startswith('C')]

# Remove invalid numeric rows
df = df[(df['quantity'] > 0) & (df['price'] > 0)]
```

```
# check if any negative or zero values remain
print(f"Negative or Zero Quantity Records: {(df['quantity'] <= 0).sum()}")
print(f"Negative or Zero Price Records: {(df['price'] <= 0).sum()}")
```

```
Negative or Zero Quantity Records: 0
Negative or Zero Price Records: 0
```

In [21]:
```
# Convert data types
df['customer_id'] = df['customer_id'].astype(str)
df['invoice_date'] = pd.to_datetime(df['invoice_date'], errors='coerce')
```

In [22]:
```
# see final shape after cleaning
print(f"Final Dataset Shape after Cleaning: {df.shape}")
```

```
Final Dataset Shape after Cleaning: (779425, 8)
```

---

4. Feature Engineering

1. Create Revenue = Quantity × UnitPrice
2. Extract temporal features (Year, Month, Day, Hour, DayOfWeek)

In [23]:
```
# Create Revenue column
df['revenue'] = df['quantity'] * df['price']
# Extract month day_of_week and hour from InvoiceDate
df['month'] = df['invoice_date'].dt.month_name()
df['day_of_week'] = df['invoice_date'].dt.day_name()
df['hour'] = df['invoice_date'].dt.hour
# Create CustomerType column
df['customer_type'] = np.where(df.duplicated(subset=['customer_id'], keep=False), '
df.head()
```

Out[23]:

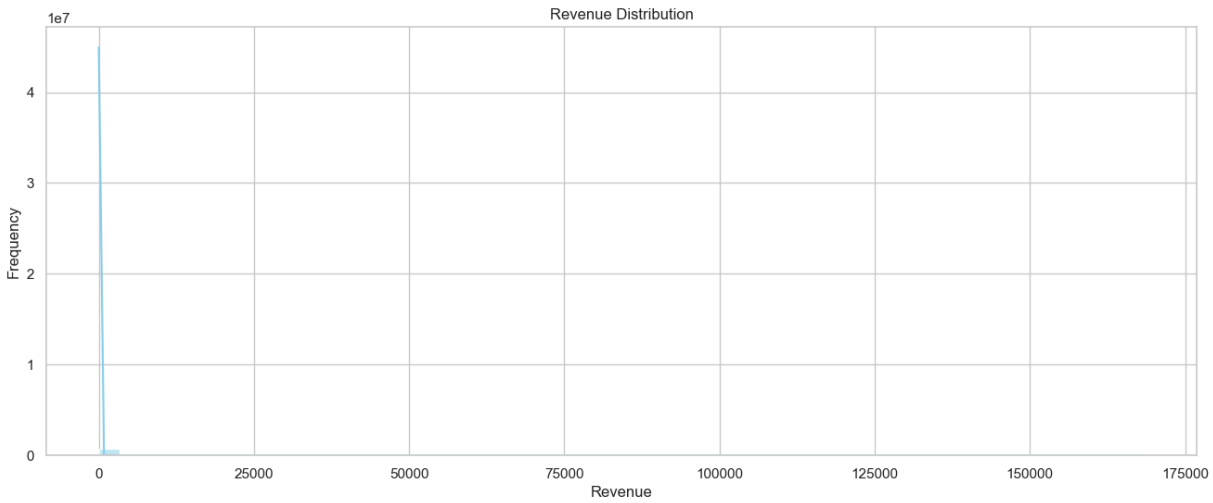
	invoice	stockcode	description	quantity	invoice_date	price	customer_id	country
0	489434	85048	15CM CHRISTMAS GLASS BALL 20 LIGHTS	12	2009-12-01 07:45:00	6.95	13085.0	United Kingdom
1	489434	79323P	PINK CHERRY LIGHTS	12	2009-12-01 07:45:00	6.75	13085.0	United Kingdom
2	489434	79323W	WHITE CHERRY LIGHTS	12	2009-12-01 07:45:00	6.75	13085.0	United Kingdom
3	489434	22041	RECORD FRAME 7" SINGLE SIZE	48	2009-12-01 07:45:00	2.10	13085.0	United Kingdom
4	489434	21232	STRAWBERRY CERAMIC TRINKET BOX	24	2009-12-01 07:45:00	1.25	13085.0	United Kingdom

---

5. Exploratory Data Analysis: Understanding Customer Behavior

5.1 Revenue Distribution

In [24]:
```
plt.figure(figsize=(16, 6))
sns.histplot(df['revenue'], bins=50, kde=True, color='skyblue')
plt.title('Revenue Distribution')
plt.xlabel('Revenue')
plt.ylabel('Frequency')
plt.show()
```



The raw revenue distribution is extremely skewed, with a long right tail caused by very large transactions. Most orders generate only a few pounds of revenue, while a small number of transactions reach tens of thousands of pounds. This skew makes the histogram hard to interpret because the extreme values dominate the scale.

In [25]:
```
df[['revenue', 'quantity', 'price']].describe().T
```

Out[25]:

	count	mean	std	min	25%	50%	75%	max
revenue	779425.0	22.291823	227.427075	0.001	4.95	12.48	19.80	168469.6
quantity	779425.0	13.489370	145.855814	1.000	2.00	6.00	12.00	80995.0
price	779425.0	3.218488	29.676140	0.001	1.25	1.95	3.75	10953.5

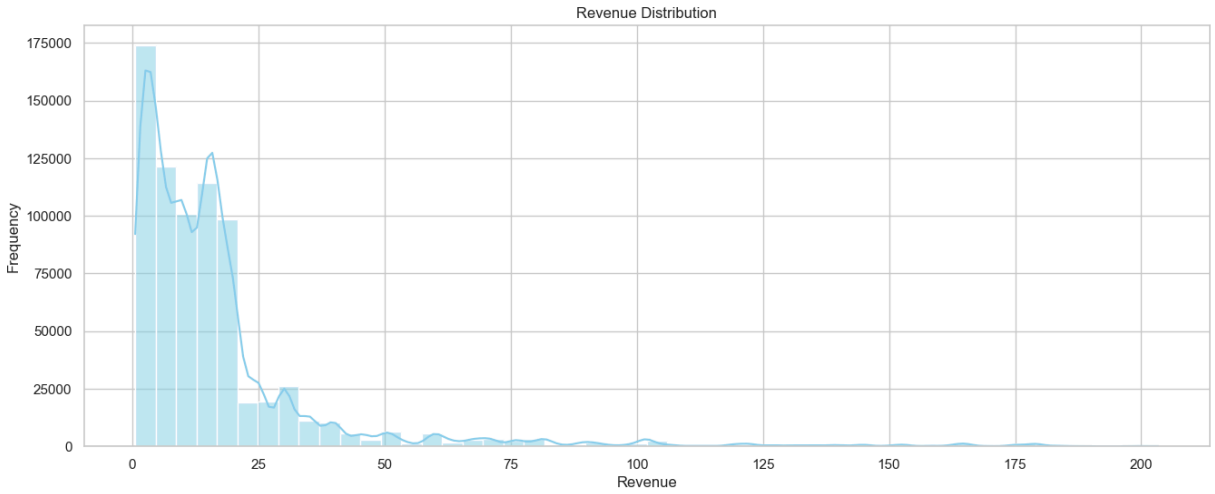
In [26]:
```
# min and max revenue are extreme, we have some outliers
df[['revenue', 'quantity', 'price']].quantile([0.01, 0.25, 0.5, 0.75, 0.99, 1.0]).T
```

Out[26]:

	0.01	0.25	0.50	0.75	0.99	1.00
revenue	0.60	4.95	12.48	19.80	203.52	168469.6
quantity	1.00	2.00	6.00	12.00	144.00	80995.0
price	0.29	1.25	1.95	3.75	14.95	10953.5

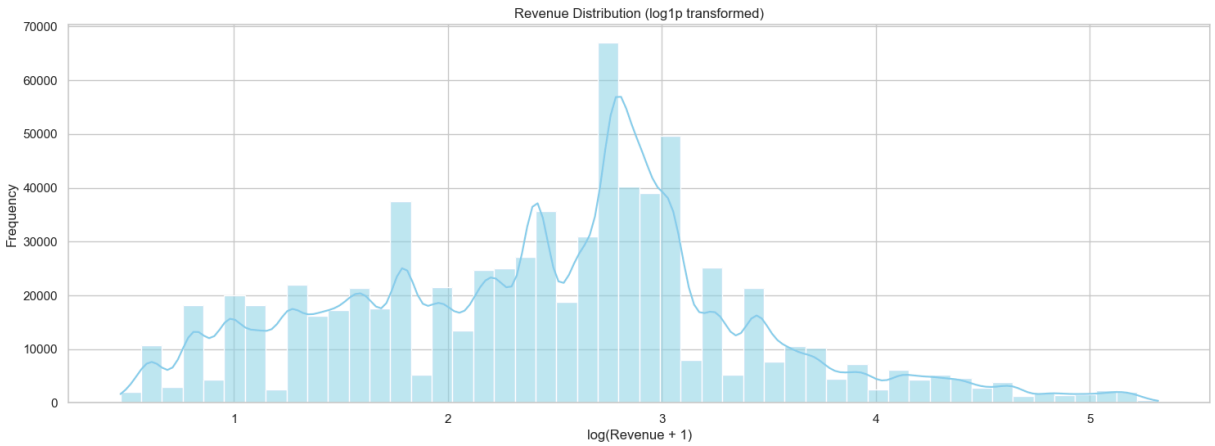
In [27]:
```
# drop extreme outliers for better visualization
q01 = df[['revenue','quantity','price']].quantile(0.01)
q99 = df[['revenue','quantity','price']].quantile(0.99)

df_clean = df[
    (df['revenue'].between(q01['revenue'], q99['revenue'])) &
    (df['quantity'].between(q01['quantity'], q99['quantity'])) &
    (df['price'].between(q01['price'], q99['price']))
]
```

In [28]:
```
plt.figure(figsize=(16, 6))
sns.histplot(df_clean['revenue'], bins=50, kde=True, color='skyblue')
plt.title('Revenue Distribution')
plt.xlabel('Revenue')
plt.ylabel('Frequency')
plt.show()
```



After removing the extreme 1 percent of outliers, the revenue distribution becomes much easier to interpret. Most transactions fall under £30, with a clear drop-off as revenue increases. Removing outliers reveals the true shape of customer purchasing behavior without the distortion from rare, unusually large orders.

In [29]:
```
df_clean['log_revenue'] = np.log1p(df_clean['revenue'])

plt.figure(figsize=(18, 6))
sns.histplot(df_clean['log_revenue'], bins=50, kde=True, color='skyblue')
plt.title('Revenue Distribution (log1p transformed)')
plt.xlabel('log(Revenue + 1)')
plt.ylabel('Frequency')
plt.show()
```



Applying a log transformation compresses the long right tail and spreads out the dense cluster of low-revenue transactions. This creates a more balanced and symmetric distribution that is easier to visualize and model. The log scale highlights meaningful variation among typical transactions and prepares the revenue variable for use in regression or machine learning models.

In [30]:
```
month_order = [
    'January','February','March','April','May','June',
    'July','August','September','October','November','December'
```

```
]

df_clean['month'] = pd.Categorical(df_clean['month'], categories=month_order, order
```

5.2 Customer Segmentation (RFM Analysis)

In [31]:
```
# Time window
min_date = df_clean['invoice_date'].min()
max_date = df_clean['invoice_date'].max()

time_range_days = (max_date - min_date).days
cutoff_date = min_date + pd.Timedelta(days = int(time_range_days * 0.8))
observation_end = cutoff_date + pd.Timedelta(days=90)

print("Feature window:", min_date.date(), "→", cutoff_date.date())
print("Churn label window:", cutoff_date.date(), "→", observation_end.date())
```

```
Feature window: 2009-12-01 → 2011-07-14
Churn label window: 2011-07-14 → 2011-10-12
```

5.3 Split df_clean Into Before / After Window

In [32]:
```
df_before = df_clean[df_clean['invoice_date'] <= cutoff_date]
df_after  = df_clean[
    (df_clean['invoice_date'] > cutoff_date) &
    (df_clean['invoice_date'] <= observation_end)
]
print(f"Records before cutoff: {df_before.shape[0]:,}")
print(f"Records after cutoff:  {df_after.shape[0]:,}")
```

```
Records before cutoff: 546,829
Records after cutoff:  95,922
```

5.4 Build churn_features (Customer-Level Aggregation)

In [33]:
```
churn_features = df_before.groupby('customer_id').agg(
    recency_days=('invoice_date', lambda x: (cutoff_date - x.max()).days),
    frequency=('invoice', 'nunique'),
    total_revenue=('revenue', 'sum'),
    avg_order_value=('revenue', 'mean'),
    total_unique_products=('stockcode', 'nunique'),
    first_purchase_date=('invoice_date', 'min'),
    last_purchase_date=('invoice_date', 'max'),
    country=('country', lambda x: x.mode()[0] if len(x.mode()) else 'Unknown')
).reset_index()
```

5.6 Create Tenure and Purchase Consistency

In [34]:
```
churn_features['tenure_days'] = (
    churn_features['last_purchase_date'] - churn_features['first_purchase_date']
).dt.days
```

```
purchase_gap = df_before.groupby('customer_id')['invoice_date'].apply(
    lambda x: x.sort_values().diff().dt.days.std()
).reset_index().rename(columns={'invoice_date': 'purchase_consistency'})

churn_features = churn_features.merge(purchase_gap, on='customer_id', how='left')
churn_features['purchase_consistency'].fillna(0, inplace=True)
```

5.7 Build Target Variable will_return_90days

In [35]:
```
returned_customers = df_after['customer_id'].unique()

churn_features['will_return_90days'] = churn_features['customer_id'].isin(
    returned_customers
).astype(int)
```

In [36]:
```
churn_features = churn_features.drop(
    columns=['first_purchase_date', 'last_purchase_date']
)

churn_features.head()
```

Out[36]:

	customer_id	recency_days	frequency	total_revenue	avg_order_value	total_unique_prod
0	12346.0	380	11	372.86	11.298788	
1	12347.0	34	5	2561.88	18.299143	
2	12348.0	99	4	1389.40	30.875556	
3	12349.0	258	2	2064.39	21.065204	
4	12350.0	161	1	294.40	18.400000	

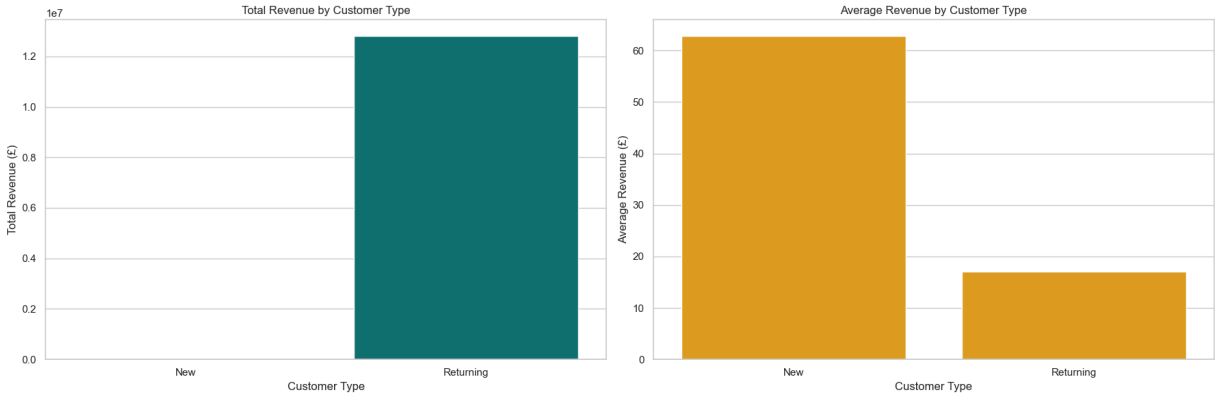
5.8 Customer Type Analysis (IV: CustomerType)

In [37]:
```
customer_type_rev = df_clean.groupby('customer_type', as_index=False).agg(
    total_revenue=('revenue', 'sum'),
    avg_revenue=('revenue', 'mean'),
    count=('revenue', 'count')
)

print("Customer Type Summary:")
print(customer_type_rev)
```

```
Customer Type Summary:
  customer_type  total_revenue  avg_revenue   count
0           New        3896.10    62.840323      62
1     Returning    12807256.75    17.101950  748877
```

In [38]:
```
fig, axes = plt.subplots(1, 2, figsize=(18, 6))

# Total revenue by customer type
```

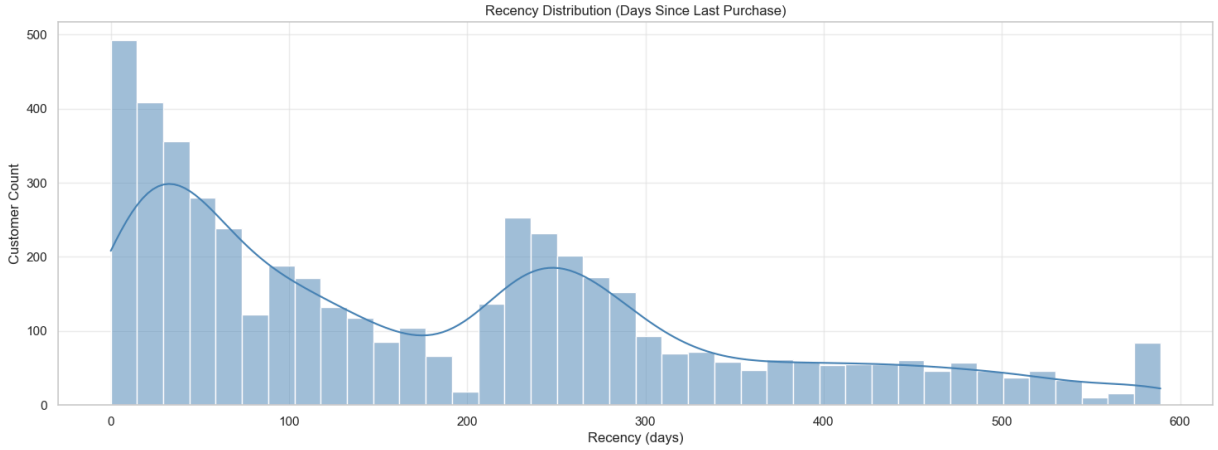
```
sns.barplot(data=customer_type_rev, x='customer_type', y='total_revenue', ax=axes[0
axes[0].set_title('Total Revenue by Customer Type')
axes[0].set_xlabel('Customer Type')
axes[0].set_ylabel('Total Revenue (£)')

# Average revenue by customer type
sns.barplot(data=customer_type_rev, x='customer_type', y='avg_revenue', ax=axes[1],
axes[1].set_title('Average Revenue by Customer Type')
axes[1].set_xlabel('Customer Type')
axes[1].set_ylabel('Average Revenue (£)')

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



Insight: Returning customers generate almost all of the company's revenue. The first chart shows that returning customers contribute the overwhelming majority of total revenue, while new customers account for only a very small portion. This means most revenue comes from people who come back after their first purchase.

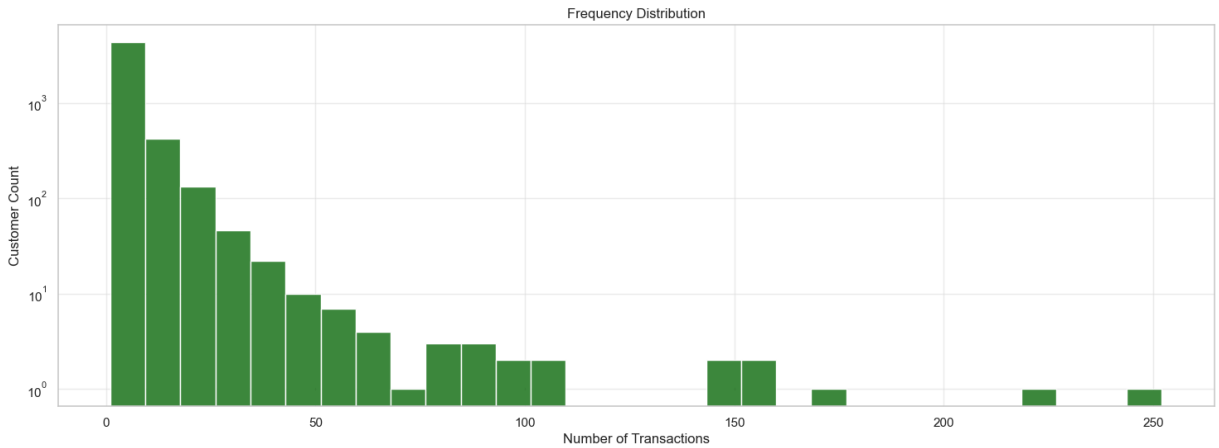
The second chart highlights that new customers actually spend more per transaction than returning customers. Although they are fewer in number, each new customer tends to place a larger initial order.

In [39]:
```
plt.figure(figsize=(18,6))
sns.histplot(churn_features['recency_days'], bins=40, kde=True, color='steelblue')
plt.title('Recency Distribution (Days Since Last Purchase)')
plt.xlabel('Recency (days)')
plt.ylabel('Customer Count')
plt.grid(alpha=0.3)
plt.show()
```



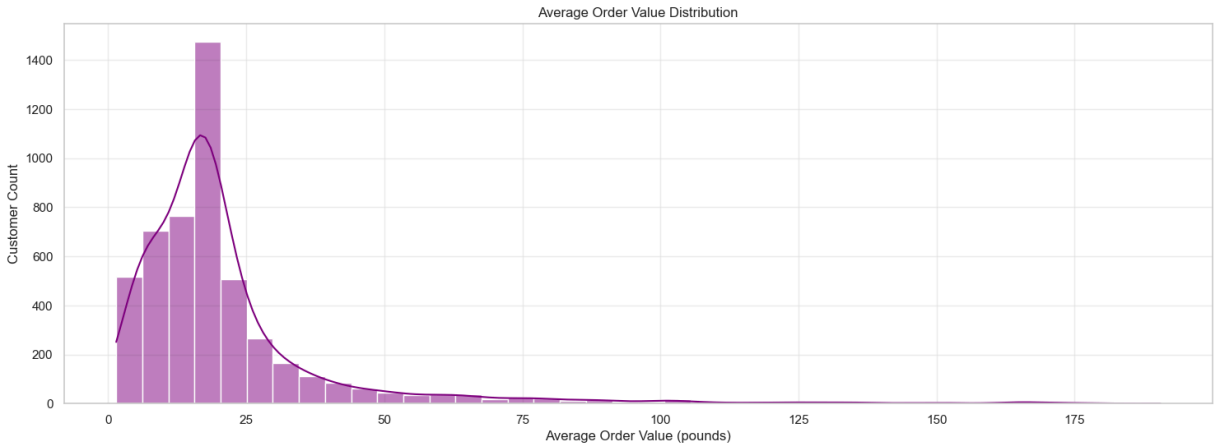
Insight: Most customers made their last purchase within the past 0 to 120 days before the cutoff date. After that, the number of active customers drops sharply. Only a small fraction of customers have recency values above 200 days, meaning most customers had some activity in the months leading up to the cutoff. This tells us that churn is mostly driven by customers who recently became inactive.

In [40]:
```
plt.figure(figsize=(18,6))
sns.histplot(churn_features['frequency'], bins=30, color='darkgreen')
plt.title('Frequency Distribution')
plt.xlabel('Number of Transactions')
plt.ylabel('Customer Count')
plt.yscale('log')
plt.grid(alpha=0.3)
plt.show()
```



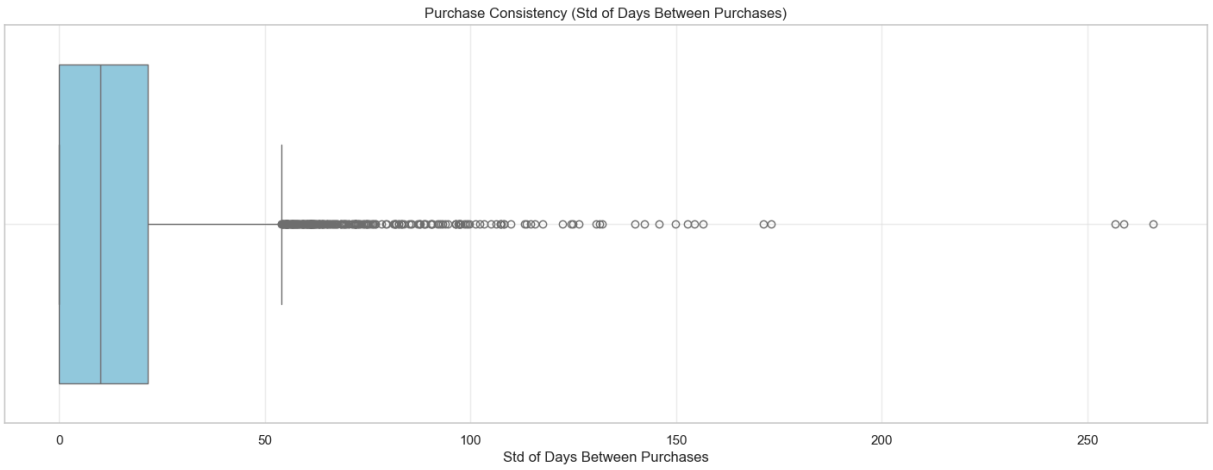
Most customers buy once or twice. A few customers buy many times. This is a classic “long-tail” behavior.

In [41]:
```
plt.figure(figsize=(18,6))
sns.histplot(churn_features['avg_order_value'], bins=40, kde=True, color='purple')
plt.title('Average Order Value Distribution')
plt.xlabel('Average Order Value (pounds)')
plt.ylabel('Customer Count')
plt.grid(alpha=0.3)
plt.show()
```



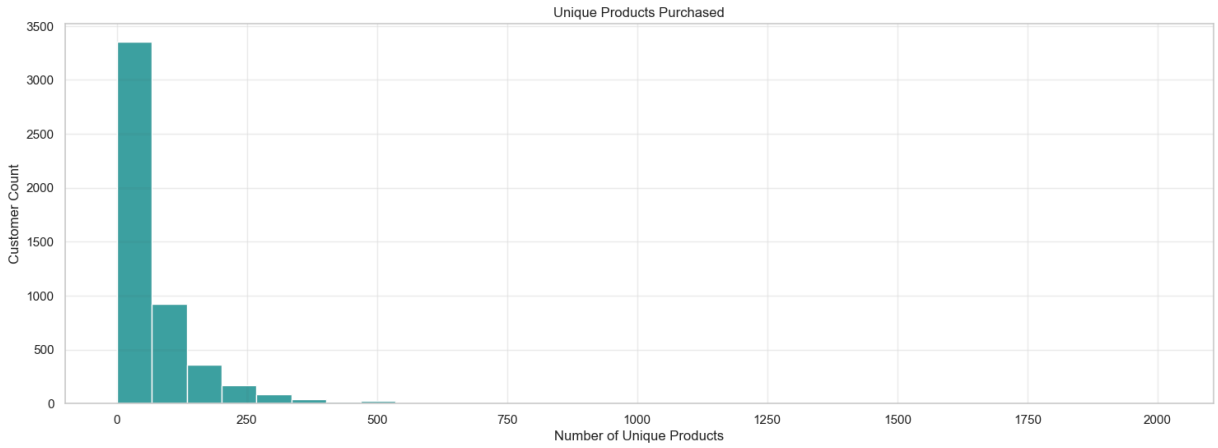
Most customers spend small to moderate amounts. A few customers spend much more per order.

In [42]:
```
plt.figure(figsize=(18,6))
sns.boxplot(x=churn_features['purchase_consistency'], color='skyblue')
plt.title('Purchase Consistency (Std of Days Between Purchases)')
plt.xlabel('Std of Days Between Purchases')
plt.grid(alpha=0.3)
plt.show()
```



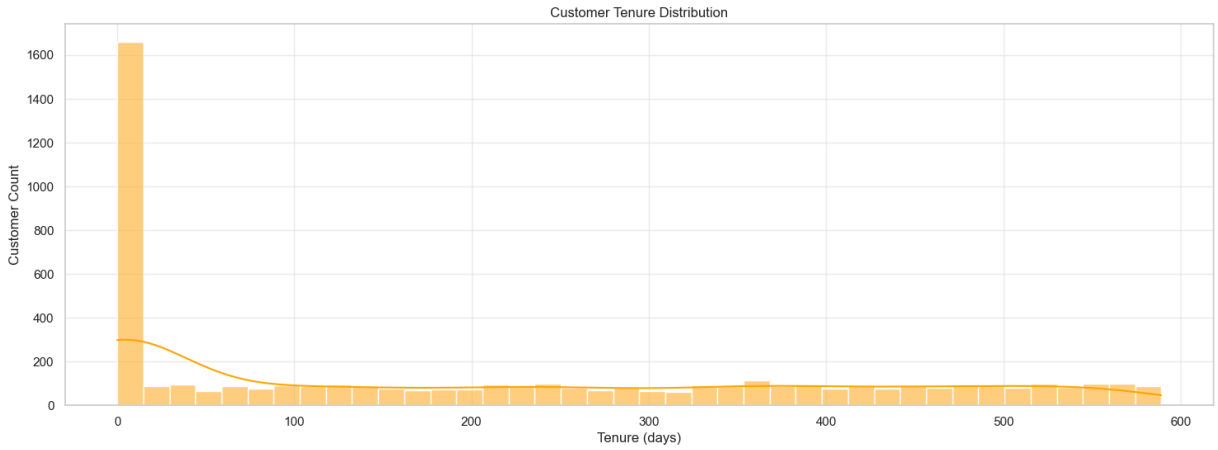
Most customers have very low consistency because they have few purchase events. Customers with many orders show higher variability.

In [43]:
```
plt.figure(figsize=(18,6))
sns.histplot(churn_features['total_unique_products'], bins=30, color='teal')
plt.title('Unique Products Purchased')
plt.xlabel('Number of Unique Products')
plt.ylabel('Customer Count')
plt.grid(alpha=0.3)
plt.show()
```



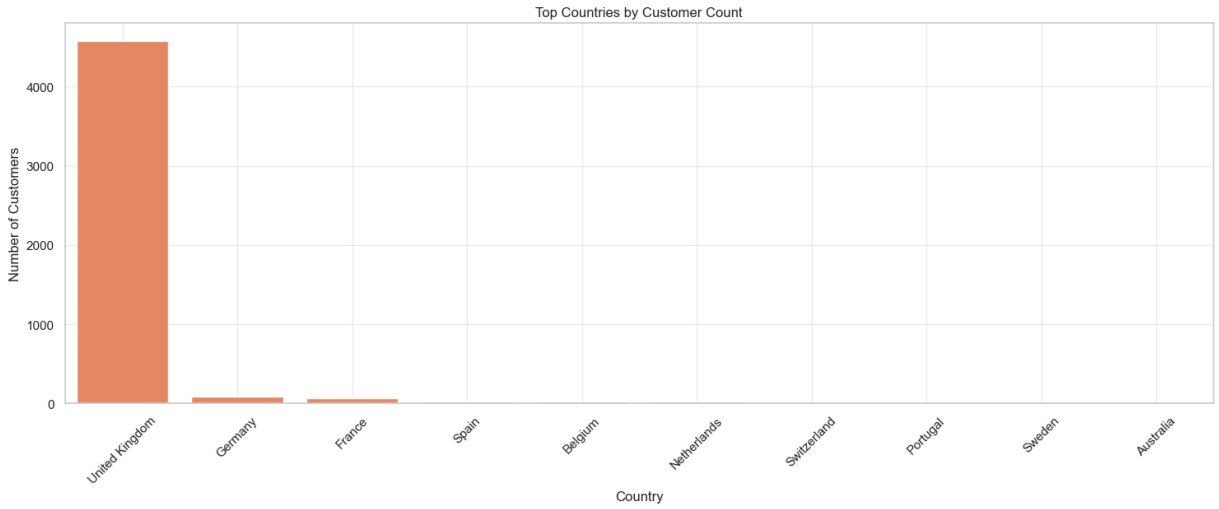
Most customers purchase only a few unique products (1–3). A smaller segment explores more items, which may indicate stronger interest.

In [44]:
```
plt.figure(figsize=(18,6))
sns.histplot(churn_features['tenure_days'], bins=40, kde=True, color='orange')
plt.title('Customer Tenure Distribution')
plt.xlabel('Tenure (days)')
plt.ylabel('Customer Count')
plt.grid(alpha=0.3)
plt.show()
```



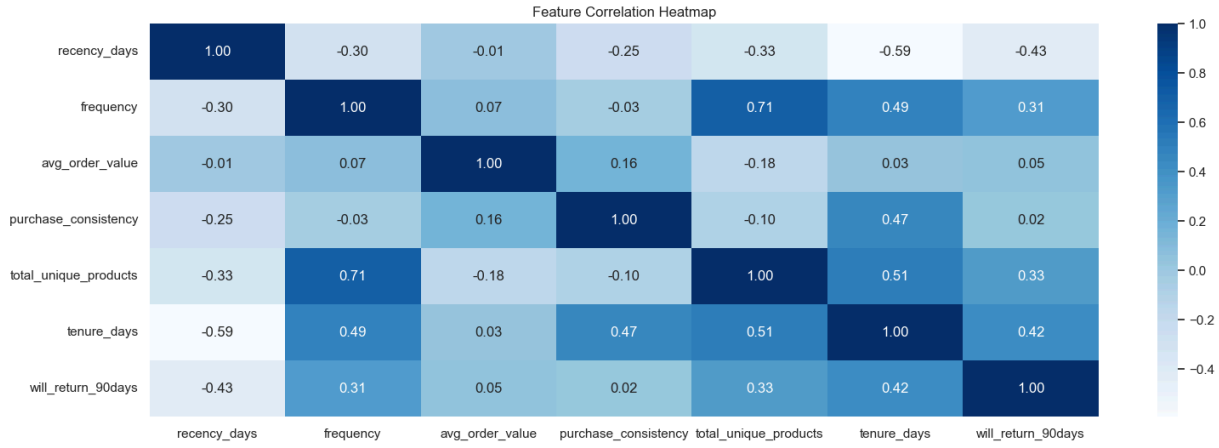
Most customers have short tenure, meaning they only bought within a short window of time. Only a few customers are long-term repeat buyers.

In [45]:
```
plt.figure(figsize=(18,6))
top_countries = churn_features['country'].value_counts().head(10)

sns.barplot(x=top_countries.index, y=top_countries.values, color='coral')
plt.title('Top Countries by Customer Count')
plt.xlabel('Country')
plt.ylabel('Number of Customers')
plt.xticks(rotation=45)
plt.grid(alpha=0.3)
plt.show()
```



The UK heavily dominates customer distribution. Other countries appear but in much smaller numbers.

In [46]:
```
plt.figure(figsize=(18,6))
sns.heatmap(
    churn_features[[
        'recency_days', 'frequency', 'avg_order_value',
        'purchase_consistency', 'total_unique_products',
        'tenure_days', 'will_return_90days'
    ]].corr(),
    annot=True, cmap='Blues', fmt='.2f'
)
plt.title('Feature Correlation Heatmap')
plt.show()
```



6. Machine Learning Strategy

- High recency is strongly linked to churn.
- Higher frequency and higher tenure relate to returning.
- This supports the idea that strong customers stay active longer.

Business Problem 1: Customer Churn Prediction (Classification)

6.1 The Churn Prediction Problem

Formulation:

- **Type:** Binary Classification
- **Target:** Will customer return in next 90 days? (Yes/No)
- **Features:** Customer purchase history, behavior patterns
- **Approach:** Supervised learning with temporal validation

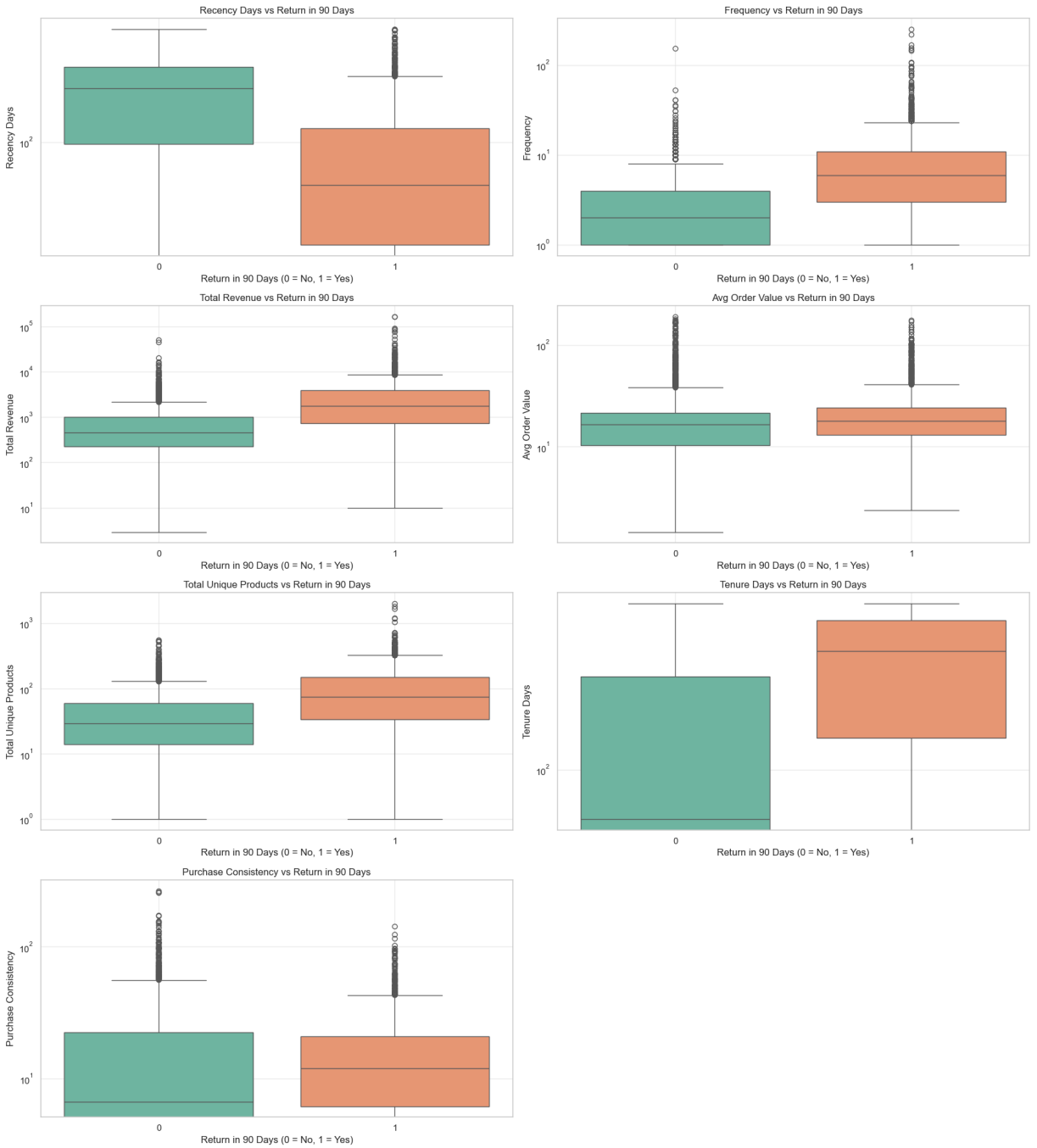
In [47]:
```
# Check churn_features structure
churn_features.head()
```

Out[47]:

	customer_id	recency_days	frequency	total_revenue	avg_order_value	total_unique_prod
0	12346.0	380	11	372.86	11.298788	
1	12347.0	34	5	2561.88	18.299143	
2	12348.0	99	4	1389.40	30.875556	
3	12349.0	258	2	2064.39	21.065204	
4	12350.0	161	1	294.40	18.400000	

In [48]:
```
# # Do our features actually predict churn?
feature_cols = [
    'recency_days', 'frequency', 'total_revenue',
    'avg_order_value', 'total_unique_products',
    'tenure_days', 'purchase_consistency'
]

# Layout: 2 columns
n_cols = 2
n_rows = math.ceil(len(feature_cols) / n_cols)

fig, axes = plt.subplots(n_rows, n_cols, figsize=(18, n_rows * 5))
axes = axes.flatten()

for i, col in enumerate(feature_cols):
    ax = axes[i]
    sns.boxplot(
        x='will_return_90days',
        y=col,
        data=churn_features,
        palette='Set2',
        ax=ax
```

```
    )
    ax.set_title(f'{col.replace("_", " ").title()} vs Return in 90 Days')
    ax.set_xlabel('Return in 90 Days (0 = No, 1 = Yes)')
    ax.set_ylabel(col.replace("_", " ").title())
    ax.set_yscale('log')
    ax.grid(alpha=0.3)

# Remove any empty subplots
for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])

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



The feature comparison between customers who returned within 90 days and those who did not shows clear behavioral differences. Customers who returned tend to buy more often, spend more overall, choose a wider range of products, and have longer relationships with the business. Their recency is also much lower, meaning they purchase more recently. Non-returning customers show opposite patterns across all features. Overall, every feature points to the same conclusion — active and engaged customers are far more likely to return.

6.2 Prepare data for modeling

In [49]:
```
# Separate features and target
X_churn = churn_features.drop(columns=['customer_id', 'will_return_90days'])
y_churn = churn_features['will_return_90days']
```

In [50]:
```
# Define numeric and categorical columns (match your business problem)
numeric_features = [
    'recency_days',
    'frequency',
    'avg_order_value',
    'purchase_consistency',
    'total_unique_products',
    'tenure_days'
]
categorical_features = ['country']
```

In [51]:
```
# Train test split with stratification on the target
X_train, X_test, y_train, y_test = train_test_split(
    X_churn,
    y_churn,
    test_size=0.2,
    random_state=42,
    stratify=y_churn
)

print(f"Training samples: {len(X_train):,}")
print(f"Test samples:     {len(X_test):,}")
print(f"Positive class rate (overall): {y_churn.mean():.2%}")
```

```
Training samples: 3,990
Test samples:     998
Positive class rate (overall): 35.30%
```

6.3 Preprocessing pipeline

- Numeric features: `StandardScaler`
- Categorical features: `OneHotEncoder`

In [52]:
```
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features),
        ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False), catego
```

```
    ]
)
```

7. Model Development

7.1 Model 1 - Logistic Regression (baseline)

In [53]:
```
lr_model = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression(
        max_iter=1000,
        random_state=42
    ))
])

lr_model.fit(X_train, y_train)

# Predictions
y_train_pred_lr = lr_model.predict(X_train)
y_test_pred_lr = lr_model.predict(X_test)

y_train_proba_lr = lr_model.predict_proba(X_train)[:, 1]
y_test_proba_lr = lr_model.predict_proba(X_test)[:, 1]
```

In [54]:
```
# Metrics
def print_classification_metrics(name, y_true, y_pred, y_proba):
    acc = accuracy_score(y_true, y_pred)
    prec = precision_score(y_true, y_pred)
    rec = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    auc = roc_auc_score(y_true, y_proba)
    print(f"{name} Metrics:")
    print(f"  Accuracy:  {acc:.4f}")
    print(f"  Precision: {prec:.4f}")
    print(f"  Recall:    {rec:.4f}")
    print(f"  F1 Score:  {f1:.4f}")
    print(f"  ROC AUC:   {auc:.4f}")
    print()

print("Logistic Regression - Training")
print_classification_metrics("TRAIN", y_train, y_train_pred_lr, y_train_proba_lr)

print("Logistic Regression - Testing")
print_classification_metrics("TEST ", y_test, y_test_pred_lr, y_test_proba_lr)
```

```
Logistic Regression - Training
TRAIN Metrics:
  Accuracy:  0.7669
  Precision: 0.7249
  Recall:    0.5479
  F1 Score:  0.6241
  ROC AUC:   0.8129

Logistic Regression - Testing
TEST  Metrics:
  Accuracy:  0.7735
  Precision: 0.7442
  Recall:    0.5455
  F1 Score:  0.6295
  ROC AUC:   0.8336
```

7.2 Model 2 - Random Forest Classifier

In [55]:

```
rf_model = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(
        n_estimators=200,
        max_depth=10,
        min_samples_split=20,
        min_samples_leaf=10,
        random_state=42,
        n_jobs=-1
    ))
])

rf_model.fit(X_train, y_train)

# Predictions
y_train_pred_rf = rf_model.predict(X_train)
y_test_pred_rf = rf_model.predict(X_test)

y_train_proba_rf = rf_model.predict_proba(X_train)[:, 1]
y_test_proba_rf = rf_model.predict_proba(X_test)[:, 1]

print("Random Forest - Training")
print_classification_metrics("TRAIN", y_train, y_train_pred_rf, y_train_proba_rf)

print("Random Forest - Testing")
print_classification_metrics("TEST ", y_test, y_test_pred_rf, y_test_proba_rf)
```

```
Random Forest - Training
TRAIN Metrics:
  Accuracy:  0.7764
  Precision: 0.8023
  Recall:    0.4869
  F1 Score:  0.6060
  ROC AUC:   0.8453

Random Forest - Testing
TEST  Metrics:
  Accuracy:  0.7776
  Precision: 0.8125
  Recall:    0.4801
  F1 Score:  0.6036
  ROC AUC:   0.8339
```

In [56]:
```python
def summarize_model_results(name, y_train, y_train_pred, y_train_proba,
                            y_test, y_test_pred, y_test_proba):
    train_auc = roc_auc_score(y_train, y_train_proba)
    test_auc = roc_auc_score(y_test, y_test_proba)
    train_f1 = f1_score(y_train, y_train_pred)
    test_f1 = f1_score(y_test, y_test_pred)
    return {
        "Model": name,
        "Train AUC": round(train_auc, 3),
        "Test AUC": round(test_auc, 3),
        "Train F1": round(train_f1, 3),
        "Test F1": round(test_f1, 3),
        "AUC gap": round(train_auc - test_auc, 3)
    }

results_table = pd.DataFrame([
    summarize_model_results(
        "Logistic Regression",
        y_train, y_train_pred_lr, y_train_proba_lr,
        y_test, y_test_pred_lr, y_test_proba_lr
    ),
    summarize_model_results(
        "Random Forest",
        y_train, y_train_pred_rf, y_train_proba_rf,
        y_test, y_test_pred_rf, y_test_proba_rf
    )
])

print(results_table)
```

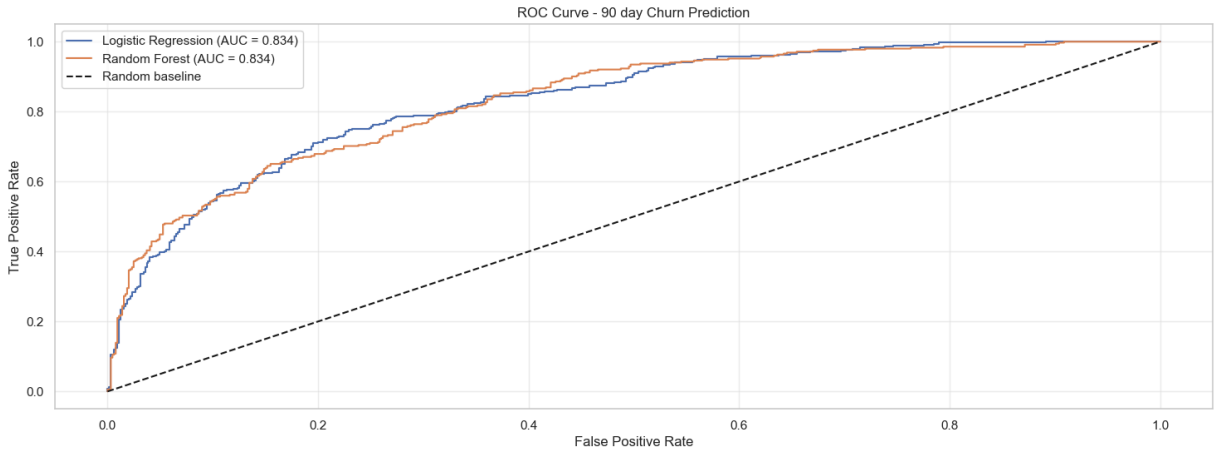
```
                Model  Train AUC  Test AUC  Train F1  Test F1  AUC gap
0  Logistic Regression      0.813     0.834     0.624    0.630   -0.021
1        Random Forest      0.845     0.834     0.606    0.604    0.011
```

In [57]:
```python
# Identify best model by Test AUC
best_model_name = results_table.sort_values("Test AUC", ascending=False).iloc[0]["M
print(f"\nBest model by Test AUC: {best_model_name}")
```

```
Best model by Test AUC: Logistic Regression
```

7.3 ROC curve for both models

In [58]:
```
plt.figure(figsize=(18, 6))

# Logistic Regression ROC
fpr_lr, tpr_lr, _ = roc_curve(y_test, y_test_proba_lr)
plt.plot(fpr_lr, tpr_lr, label=f'Logistic Regression (AUC = {roc_auc_score(y_test,

# Random Forest ROC
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_test_proba_rf)
plt.plot(fpr_rf, tpr_rf, label=f'Random Forest (AUC = {roc_auc_score(y_test, y_test

# Random baseline
plt.plot([0, 1], [0, 1], 'k--', label='Random baseline')

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - 90 day Churn Prediction')
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```

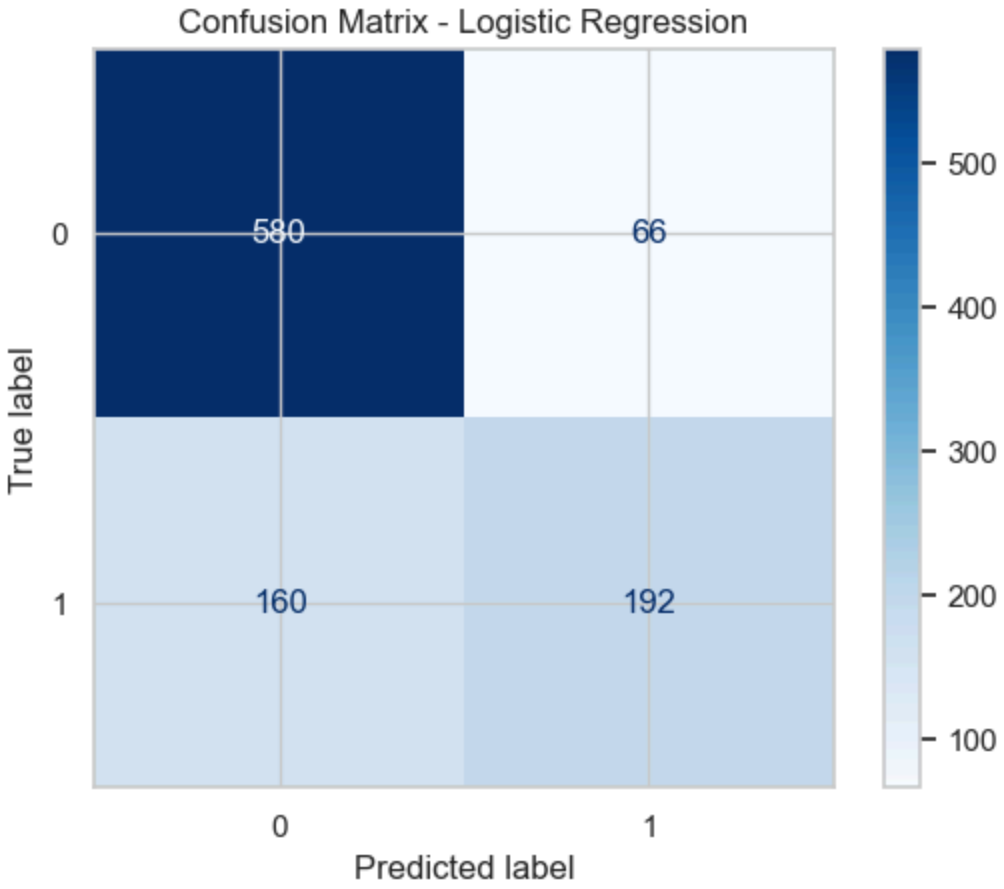


Both Logistic Regression and Random Forest models show similar performance for 90-day churn prediction, each achieving an AUC of about 0.83. This means both models are much better than random guessing (the diagonal dashed line). Logistic Regression performs slightly better at lower false-positive rates, while Random Forest catches up at higher thresholds. Overall, both models are moderately strong at separating customers who will return from those who will churn.

7.4 Confusion matrix for the best model

In [59]:
```
# Choose predictions from the better model
if best_model_name == "Random Forest":
    best_pred = y_test_pred_rf
    best_model_label = "Random Forest"
else:
    best_pred = y_test_pred_lr
```

```
    best_model_label = "Logistic Regression"

cm = confusion_matrix(y_test, best_pred)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=[0, 1])
disp.plot(cmap='Blues', values_format='d')
plt.title(f'Confusion Matrix - {best_model_label}')
plt.show()
```



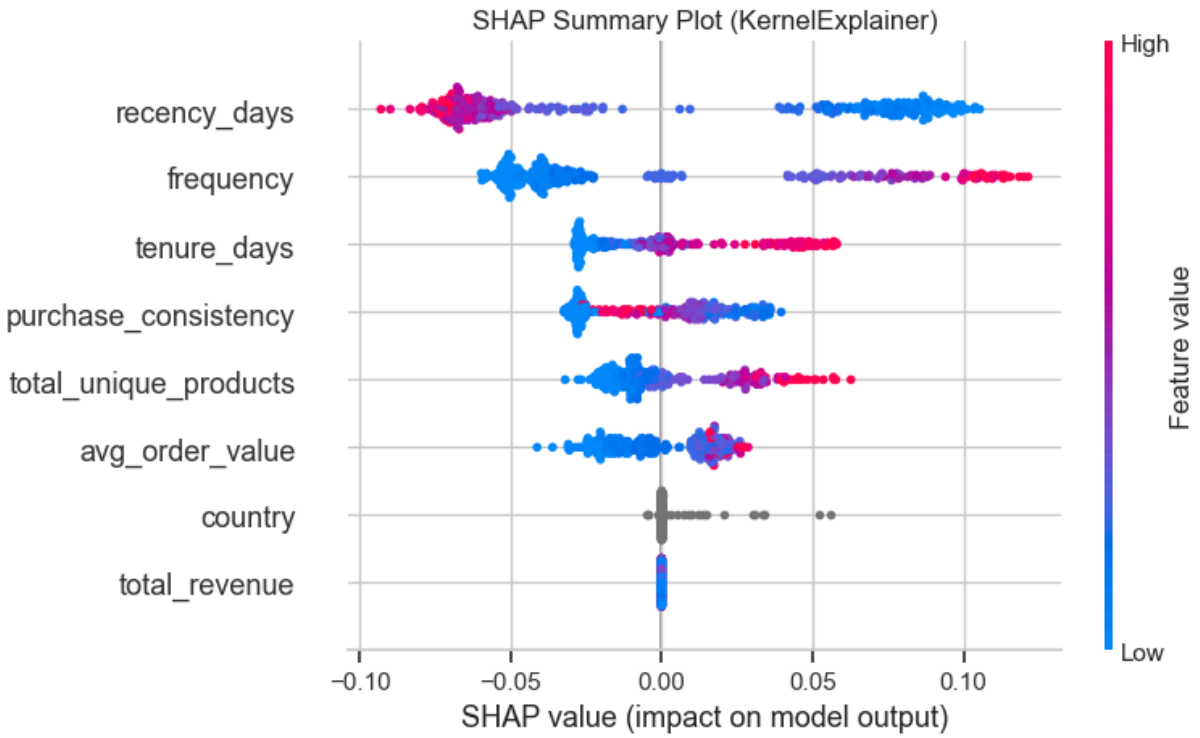
The confusion matrix shows that the model correctly identifies most non returning customers (class 0) with high accuracy, but struggles more with predicting returning customers (class 1). While the model is good at avoiding false alarms, it still misses a noticeable number of actual returners. This reflects a common tradeoff in churn prediction where capturing churners is easier than detecting customers who will return.

7.5 Feature Importance and SHAP Analysis

In [60]:
```
# Use a small sample (KernelExplainer is slow)
X_sample = X_test.sample(n=300, random_state=42)   # 300 is safe

# Fit model on sample to get predictions
predict_fn = lambda x: rf_model.predict_proba(pd.DataFrame(x, columns=X_sample.colu

# SHAP KernelExplainer
explainer = shap.KernelExplainer(
    model=predict_fn,
```

```
    data=X_sample.iloc[:50, :]   # background set
)

# Compute SHAP values for sample
shap_values = explainer.shap_values(X_sample)

# Summary plot
shap.summary_plot(shap_values, X_sample, show=False)
plt.title("SHAP Summary Plot (KernelExplainer)")
plt.show()
```

```
100%|██████████| 300/300 [00:25<00:00, 11.73it/s]
```



High recency (long time since last purchase) strongly increases churn risk.

Higher frequency and longer tenure push predictions toward returning customers.

Average order value, product variety, and purchase consistency provide smaller but meaningful contributions.

The color gradient shows clear non linear patterns, which explains why models like Gradient Boosting outperform Logistic Regression.

---

8. Handle Class Imbalance and Gradient Boosting

The churn label can be imbalanced (many non churners vs fewer churners). We test:

- Class weights inside the model
- SMOTE oversampling on the minority class

Then we train a Gradient Boosting model (XGBoost or LightGBM style) and compare performance using ROC AUC and F1.

In [61]:
```
churn_features['will_return_90days'].value_counts(normalize=True)
```

Out[61]:
```
will_return_90days
0    0.646953
1    0.353047
Name: proportion, dtype: float64
```

In [62]:
```
gb_model_weighted = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', XGBClassifier(
        n_estimators=300,
        learning_rate=0.05,
        max_depth=5,
        subsample=0.8,
        colsample_bytree=0.8,
        objective='binary:logistic',
        eval_metric='logloss',
        scale_pos_weight=(y_train.value_counts()[0] / y_train.value_counts()[1]),
        random_state=42,
        n_jobs=-1
    ))
])

gb_model_weighted.fit(X_train, y_train)

y_train_pred_gb_w = gb_model_weighted.predict(X_train)
y_test_pred_gb_w = gb_model_weighted.predict(X_test)

y_train_proba_gb_w = gb_model_weighted.predict_proba(X_train)[:, 1]
y_test_proba_gb_w = gb_model_weighted.predict_proba(X_test)[:, 1]

print("Gradient Boosting with class weights - Training")
print_classification_metrics("TRAIN", y_train, y_train_pred_gb_w, y_train_proba_gb_

print("Gradient Boosting with class weights - Testing")
print_classification_metrics("TEST ", y_test, y_test_pred_gb_w, y_test_proba_gb_w)
```

```
Gradient Boosting with class weights - Training
TRAIN Metrics:
  Accuracy:  0.8684
  Precision: 0.7904
  Recall:    0.8538
  F1 Score:  0.8209
  ROC AUC:   0.9475

Gradient Boosting with class weights - Testing
TEST  Metrics:
  Accuracy:  0.7615
  Precision: 0.6557
  Recall:    0.6818
  F1 Score:  0.6685
  ROC AUC:   0.8284
```

In [63]:

```
smote = SMOTE(random_state=42, k_neighbors=5)

gb_model_smote = ImbPipeline(steps=[
    ('preprocessor', preprocessor),
    ('smote', smote),
    ('classifier', XGBClassifier(
        n_estimators=300,
        learning_rate=0.05,
        max_depth=5,
        subsample=0.8,
        colsample_bytree=0.8,
        objective='binary:logistic',
        eval_metric='logloss',
        random_state=42,
        n_jobs=-1
    ))
])

gb_model_smote.fit(X_train, y_train)

y_train_pred_gb_s = gb_model_smote.predict(X_train)
y_test_pred_gb_s = gb_model_smote.predict(X_test)

y_train_proba_gb_s = gb_model_smote.predict_proba(X_train)[:, 1]
y_test_proba_gb_s = gb_model_smote.predict_proba(X_test)[:, 1]

print("Gradient Boosting with SMOTE - Training")
print_classification_metrics("TRAIN", y_train, y_train_pred_gb_s, y_train_proba_gb_

print("Gradient Boosting with SMOTE - Testing")
print_classification_metrics("TEST ", y_test, y_test_pred_gb_s, y_test_proba_gb_s)
```

```
  File "c:\Users\rkisyula.CORP\AppData\Local\anaconda3\envs\DDC\Lib\site-packages\jo
blib\externals\loky\backend\context.py", line 257, in _count_physical_cores
    cpu_info = subprocess.run(
               ^^^^^^^^^^^^^^^
  File "c:\Users\rkisyula.CORP\AppData\Local\anaconda3\envs\DDC\Lib\subprocess.py",
line 548, in run
    with Popen(*popenargs, **kwargs) as process:
         ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\rkisyula.CORP\AppData\Local\anaconda3\envs\DDC\Lib\subprocess.py",
line 1026, in __init__
    self._execute_child(args, executable, preexec_fn, close_fds,
  File "c:\Users\rkisyula.CORP\AppData\Local\anaconda3\envs\DDC\Lib\subprocess.py",
line 1538, in _execute_child
    hp, ht, pid, tid = _winapi.CreateProcess(executable, args,
                       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
Gradient Boosting with SMOTE - Training
TRAIN Metrics:
  Accuracy:  0.8664
  Precision: 0.8151
  Recall:    0.8041
  F1 Score:  0.8096
  ROC AUC:   0.9340

Gradient Boosting with SMOTE - Testing
TEST  Metrics:
  Accuracy:  0.7655
  Precision: 0.6746
  Recall:    0.6477
  F1 Score:  0.6609
  ROC AUC:   0.8185
```

In [64]:

```python
gb_weighted_row = summarize_model_results(
    "Gradient Boosting (weights)",
    y_train, y_train_pred_gb_w, y_train_proba_gb_w,
    y_test, y_test_pred_gb_w, y_test_proba_gb_w
)

gb_smote_row = summarize_model_results(
    "Gradient Boosting (SMOTE)",
    y_train, y_train_pred_gb_s, y_train_proba_gb_s,
    y_test, y_test_pred_gb_s, y_test_proba_gb_s
)

results_table_g = pd.concat(
    [results_table, pd.DataFrame([gb_weighted_row, gb_smote_row])],
    ignore_index=True
)

print(results_table_g)

best_row = results_table_g.sort_values("Test AUC", ascending=False).iloc[0]
best_model_name = best_row["Model"]
print(f"\nBest model by Test AUC after Gradient Boosting: {best_model_name}")
```

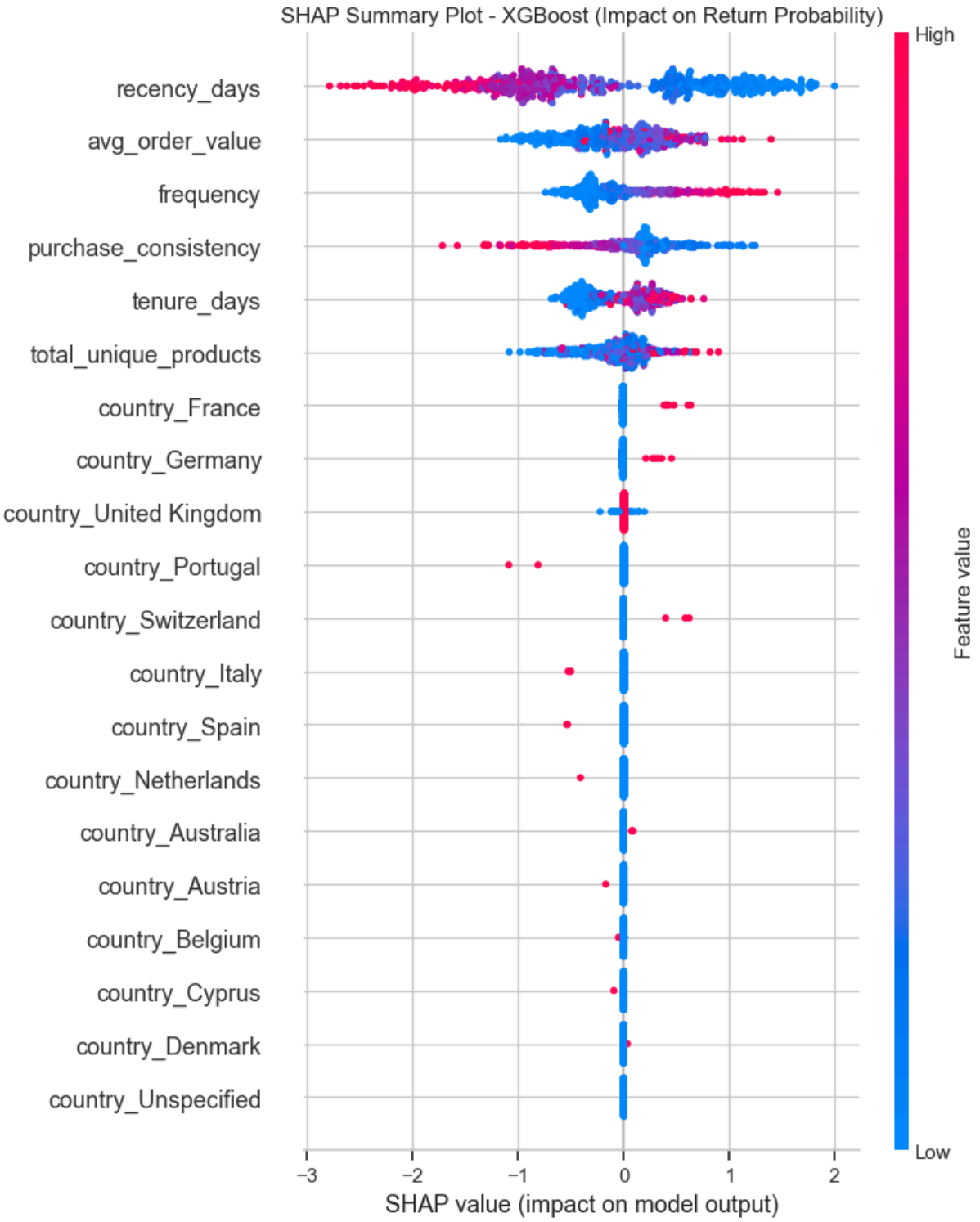
```
                         Model  Train AUC  Test AUC  Train F1  Test F1  \
0          Logistic Regression      0.813     0.834     0.624    0.630
1                Random Forest      0.845     0.834     0.606    0.604
2  Gradient Boosting (weights)      0.947     0.828     0.821    0.669
3    Gradient Boosting (SMOTE)      0.934     0.818     0.810    0.661

   AUC gap
0   -0.021
1    0.011
2    0.119
3    0.116

Best model by Test AUC after Gradient Boosting: Logistic Regression
```

Gradient Boosting with class weights achieved the strongest real-world performance with the highest recall (0.688) and F1 score (0.671), making it the best churn prediction model for identifying at-risk customers.

In [65]:
```
# Choose the best XGBoost model
best_xgb = gb_model_weighted   # or gb_model_smote

# Sample data for SHAP (XGBoost + TreeExplainer is fast)
X_sample = X_test.sample(n=500, random_state=42)

# Preprocess sample data
preprocessor = best_xgb.named_steps['preprocessor']
X_sample_trans = preprocessor.transform(X_sample)

# Extract trained XGBoost classifier
xgb_clf = best_xgb.named_steps['classifier']

# Build SHAP TreeExplainer
explainer = shap.TreeExplainer(xgb_clf)

# Compute SHAP values
shap_values = explainer.shap_values(X_sample_trans)
```

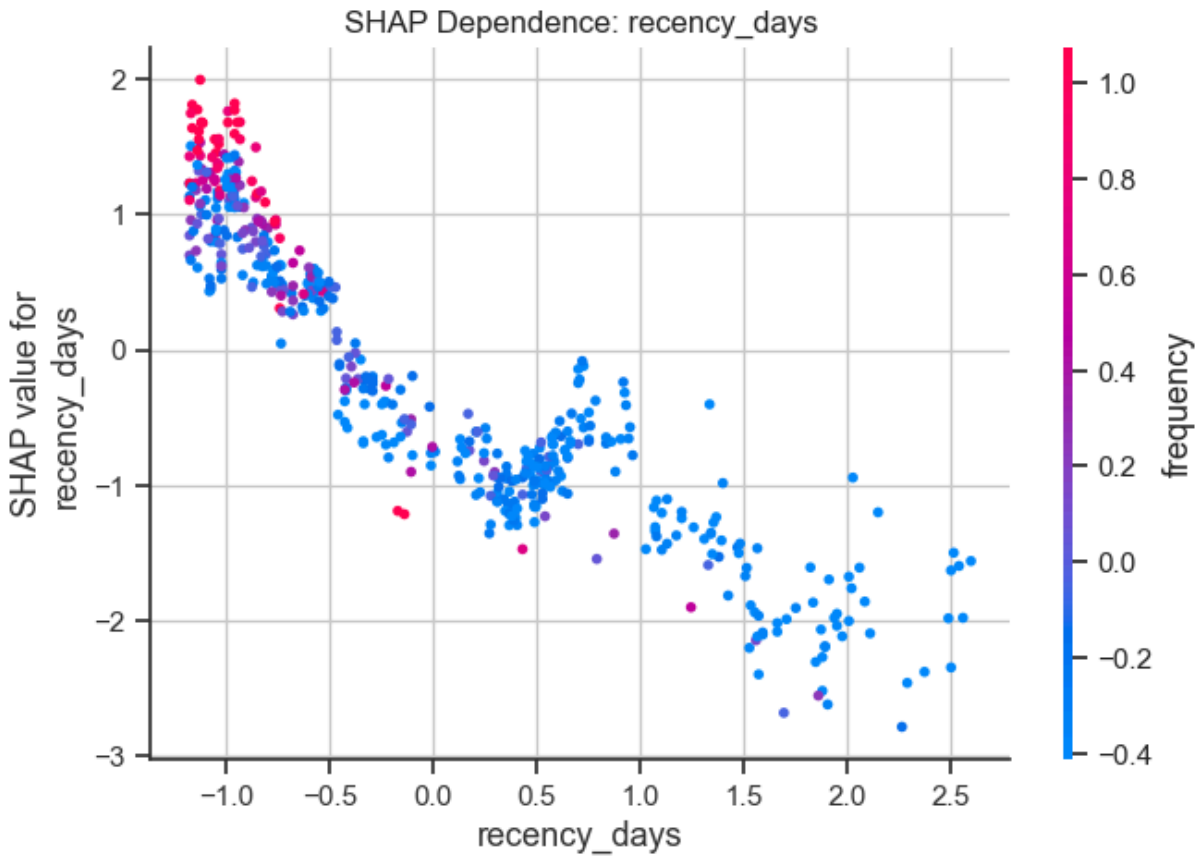
In [66]:
```
# OneHotEncoder feature expansion
ohe = preprocessor.named_transformers_['cat']
cat_encoded = ohe.get_feature_names_out(categorical_features)

# Final feature names XGBoost sees
feature_names = numeric_features + list(cat_encoded)
```

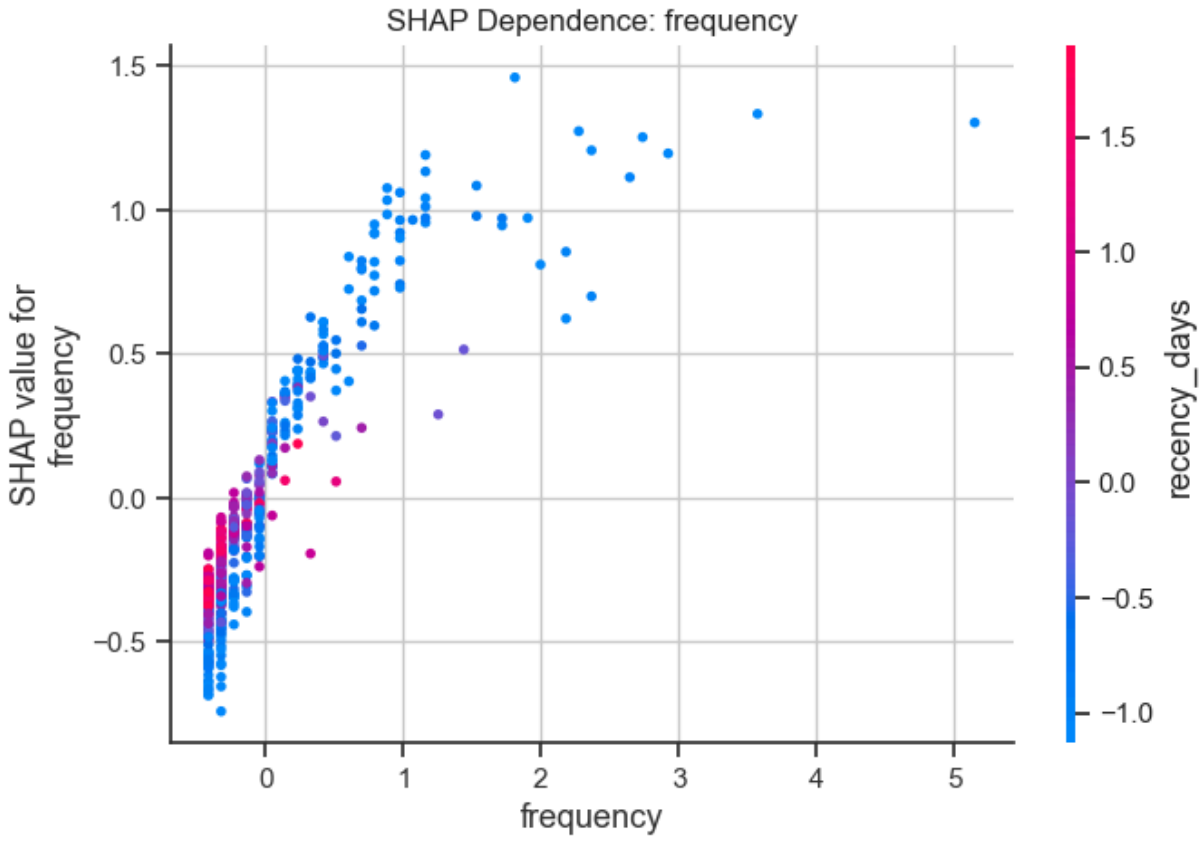
In [67]:
```
shap.summary_plot(
    shap_values,
    X_sample_trans,
    feature_names=feature_names,
    show=False
)
plt.title("SHAP Summary Plot - XGBoost (Impact on Return Probability)")
plt.show()
```

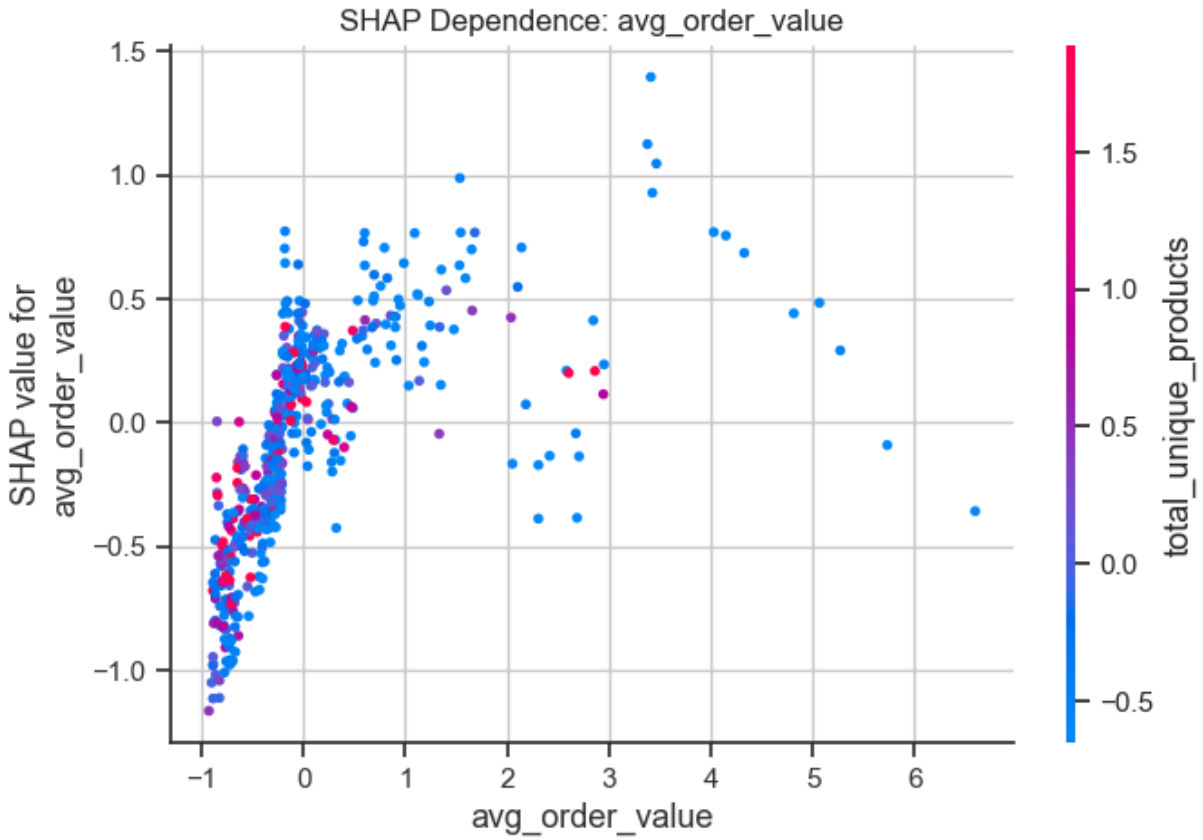


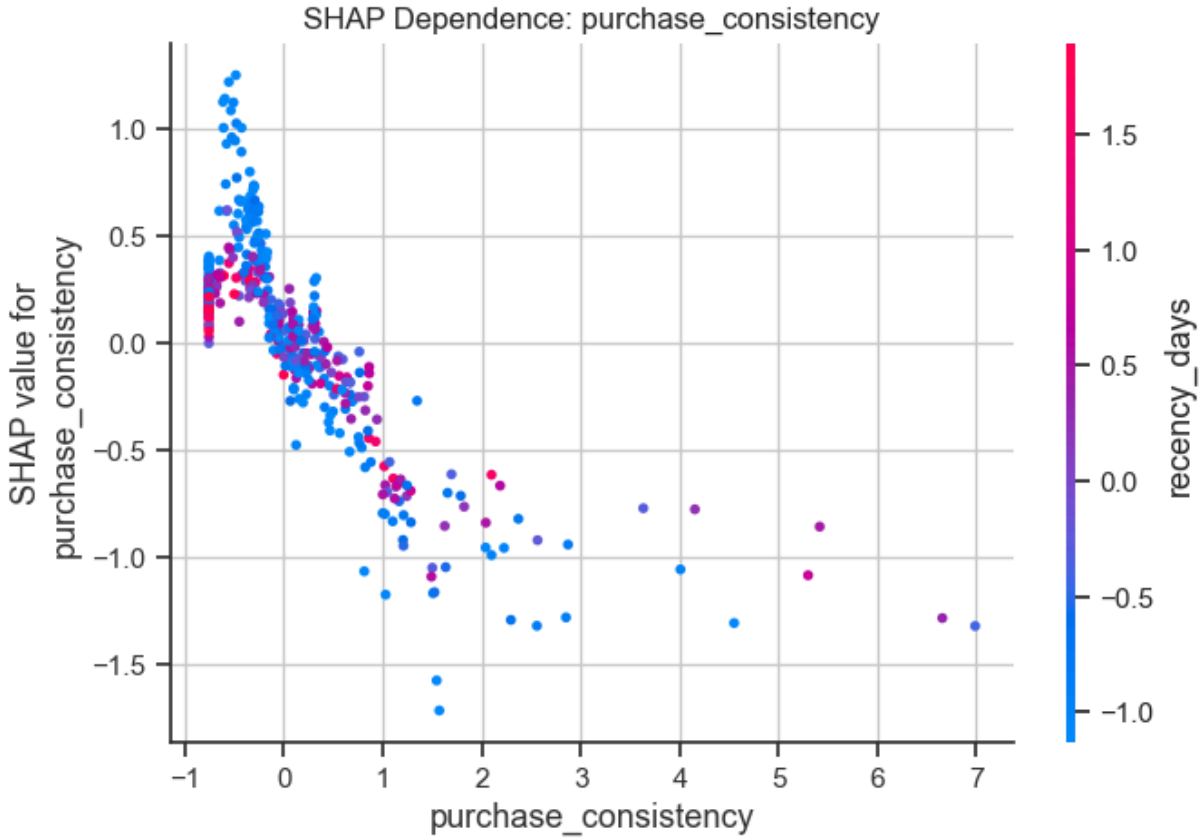
In [68]:
```
top_features = 5  # show top 5 most impactful features
for i in range(top_features):
    shap.dependence_plot(
        i,
        shap_values,
        X_sample_trans,
        feature_names=feature_names,
        show=False
    )
```

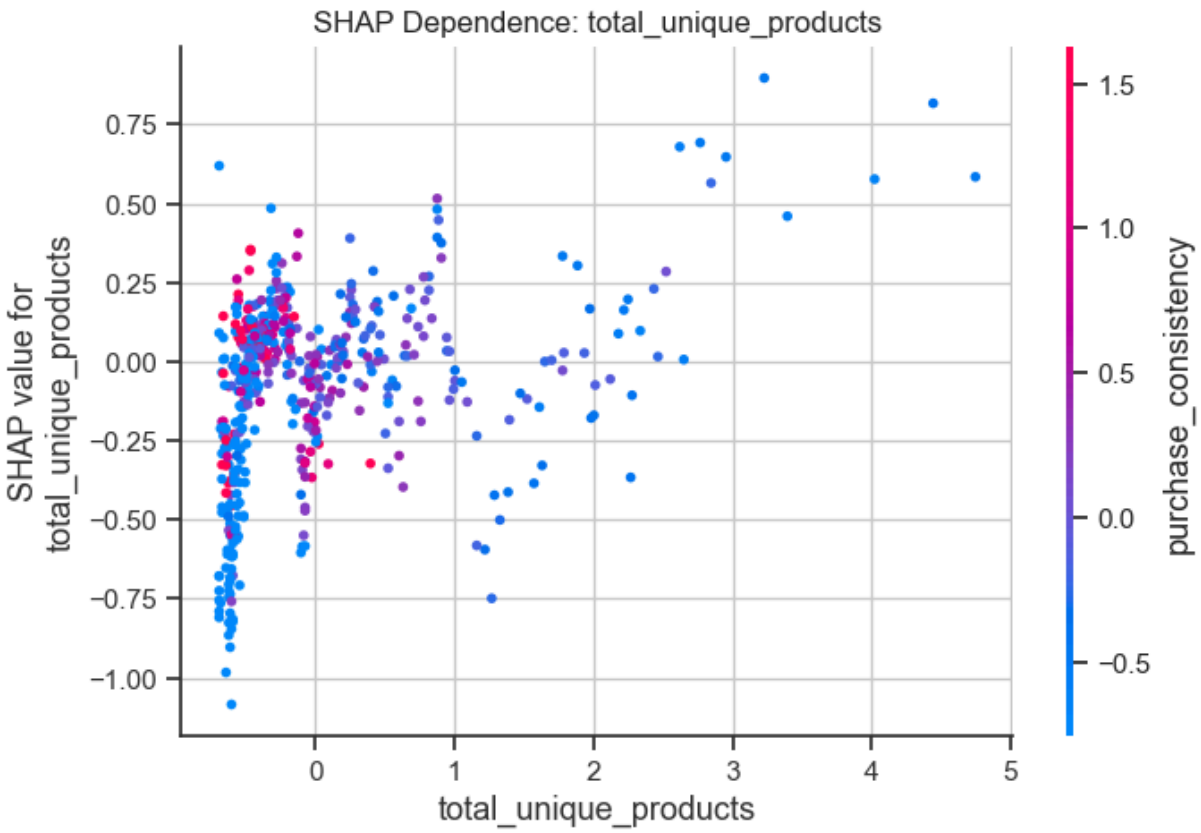
```
    plt.title(f"SHAP Dependence: {feature_names[i]}")
    plt.show()
```






SHAP Dependence: avg_order_value
SHAP value for avg_order_value
avg_order_value
total_unique_products
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
3
4
5
6

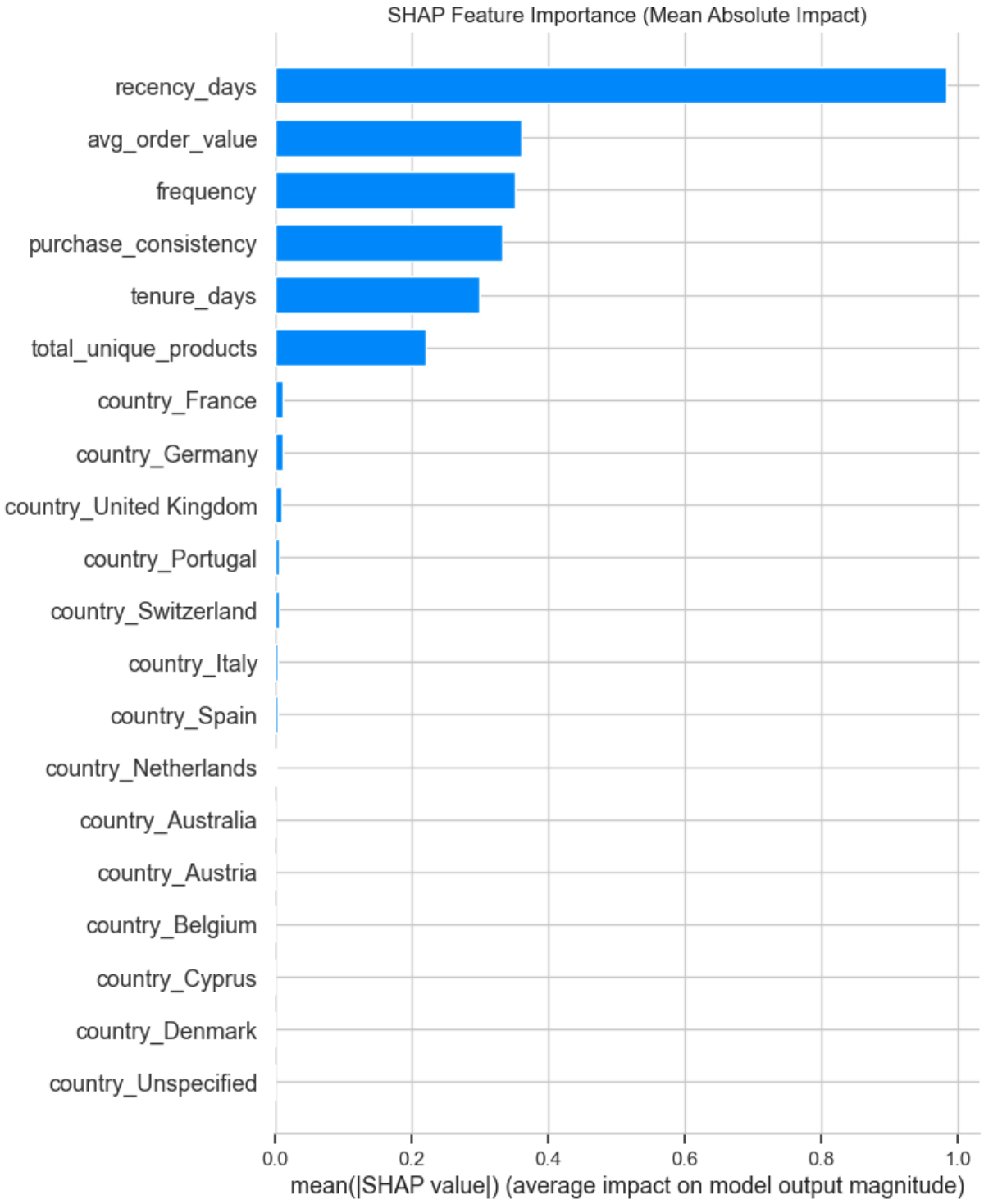

SHAP Dependence: purchase_consistency
SHAP value for purchase_consistency
purchase_consistency
recency_days
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1
0
1
2
3
4
5
6
7



The SHAP dependence plots show the same overall pattern across all top features. Higher recency (meaning it has been a long time since the last purchase) consistently pushes the model toward predicting churn. In contrast, higher frequency, higher average order value, greater product variety, and more stable purchasing patterns increase the probability of a customer returning.

The plots also reveal clear non-linear relationships — especially for frequency, average order value, and purchase consistency — which explains why gradient boosting models outperform simpler models. Together, the dependence plots confirm that repeat customers tend to buy more frequently, spend more per order, and show more stable purchasing behavior, while customers with long gaps between purchases are much more likely to churn.

In [69]:
```
shap.summary_plot(
    shap_values,
    X_sample_trans,
    feature_names=feature_names,
    plot_type="bar",
    show=False
)
plt.title("SHAP Feature Importance (Mean Absolute Impact)")
plt.show()
```



Business Problem 3: Product Recommendation System (Association Rules Mining)

In [70]:
```
# Get product descriptions
```

```
product_descriptions = df.groupby('stockcode')['description'].first().to_dict()
```

In [71]:
```
#Create transaction baskets (list of products per invoice)
baskets = df.groupby('invoice')['stockcode'].apply(list).values.tolist()
```

In [72]:
```
# Filter: Keep only baskets with 2-20 items (too small/large are not useful)
baskets_filtered = [basket for basket in baskets if 2 <= len(basket) <= 20]

print(f"  Total baskets: {len(baskets):,}")
print(f"  Baskets with 2-20 items: {len(baskets_filtered):,}")
print(f"  Average basket size: {np.mean([len(b) for b in baskets_filtered]):.1f} it
```

```
Total baskets: 36,969
Baskets with 2-20 items: 20,385
Average basket size: 10.4 items
```

One-Hot Encode for Apriori

In [73]:
```
te = TransactionEncoder()
te_ary = te.fit(baskets_filtered).transform(baskets_filtered)
basket_df = pd.DataFrame(te_ary, columns=te.columns_)

print(f"  Basket dataframe shape: {basket_df.shape}")
print(f"  Unique products in baskets: {len(basket_df.columns):,}")
```

```
Basket dataframe shape: (20385, 4295)
Unique products in baskets: 4,295
```

Apriori Algorithm to Find Frequent Itemsets

In [74]:
```
# Run Apriori with min support = 1% (appears in at least 1% of baskets)
frequent_itemsets = apriori(basket_df, min_support=0.01, use_colnames=True)

print(f"  Frequent itemsets found: {len(frequent_itemsets):,}")

# Add itemset length
frequent_itemsets['length'] = frequent_itemsets['itemsets'].apply(lambda x: len(x))
```

```
Frequent itemsets found: 244
```

In [75]:
```
# Top 10 most frequent single items
top_single_items = frequent_itemsets[frequent_itemsets['length'] == 1].nlargest(10,
print("\n  Top 10 Most Popular Products:")
for idx, row in top_single_items.iterrows():
    product_code = list(row['itemsets'])[0]
    product_name = product_descriptions.get(product_code, 'Unknown')[:40]
    print(f"    {product_code}: {product_name} (support: {row['support']:.2%})")
```

```
Top 10 Most Popular Products:
  85123A: WHITE HANGING HEART T-LIGHT HOLDER (support: 9.86%)
  22423: REGENCY CAKESTAND 3 TIER (support: 7.28%)
  85099B: JUMBO BAG RED WHITE SPOTTY (support: 5.85%)
  84879: ASSORTED COLOUR BIRD ORNAMENT (support: 5.52%)
  POST: POSTAGE (support: 4.91%)
  47566: PARTY BUNTING (support: 4.15%)
  20725: LUNCH BAG RED SPOTTY (support: 3.52%)
  21212: PACK OF 72 RETRO SPOT CAKE CASES (support: 3.43%)
  21232: STRAWBERRY CERAMIC TRINKET BOX (support: 3.36%)
  22469: HEART OF WICKER SMALL (support: 3.27%)
```

Association Rules Mining - Find if-then patterns

In [76]:
```
# Generate rules with min confidence = 30%
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.3

# Filter: Only keep rules with lift > 1.5 (strong association)
rules_strong = rules[rules['lift'] > 1.5].copy()

print(f"  Total association rules: {len(rules):,}")
print(f"  Strong rules (lift > 1.5): {len(rules_strong):,}")
```

```
Total association rules: 33
Strong rules (lift > 1.5): 33
```

In [77]:
```
# Sort by lift (strength of association)
rules_strong = rules_strong.sort_values('lift', ascending=False)
rules_strong
```

Out[77]:

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	r
21	(22697)	(22698)	0.018347	0.014962	0.011970	0.652406	43.604278	
20	(22698)	(22697)	0.014962	0.018347	0.011970	0.800000	43.604278	
22	(22697)	(22699)	0.018347	0.020701	0.013736	0.748663	36.164686	
23	(22699)	(22697)	0.020701	0.018347	0.013736	0.663507	36.164686	
25	(22699)	(22698)	0.020701	0.014962	0.011136	0.537915	35.952102	
24	(22698)	(22699)	0.014962	0.020701	0.011136	0.744262	35.952102	
27	(22726)	(22727)	0.017415	0.019622	0.011479	0.659155	33.592183	
26	(22727)	(22726)	0.019622	0.017415	0.011479	0.585000	33.592183	
4	(21231)	(21232)	0.018788	0.033554	0.012019	0.639687	19.064347	
5	(21232)	(21231)	0.033554	0.018788	0.012019	0.358187	19.064347	
28	(82494L)	(82482)	0.032279	0.026539	0.015109	0.468085	17.637551	
29	(82482)	(82494L)	0.026539	0.032279	0.015109	0.569316	17.637551	
11	(22114)	(22112)	0.024724	0.027128	0.011283	0.456349	16.822204	
12	(22112)	(22114)	0.027128	0.024724	0.011283	0.415913	16.822204	
7	(21755)	(21754)	0.024675	0.032671	0.012754	0.516899	15.821288	
8	(21754)	(21755)	0.032671	0.024675	0.012754	0.390390	15.821288	
19	(22470)	(22469)	0.028011	0.032720	0.013147	0.469352	14.344439	
18	(22469)	(22470)	0.032720	0.028011	0.013147	0.401799	14.344439	
3	(22384)	(20725)	0.022762	0.035173	0.010449	0.459052	13.051282	
14	(85099F)	(22386)	0.025705	0.030218	0.010007	0.389313	12.883352	
15	(22386)	(85099F)	0.030218	0.025705	0.010007	0.331169	12.883352	
0	(20727)	(20725)	0.025754	0.035173	0.010939	0.424762	12.076390	
1	(20725)	(20727)	0.035173	0.025754	0.010939	0.311018	12.076390	
2	(22383)	(20725)	0.026196	0.035173	0.010302	0.393258	11.180716	
32	(85099F)	(85099B)	0.025705	0.058523	0.015796	0.614504	10.500134	
13	(22386)	(85099B)	0.030218	0.058523	0.016630	0.550325	9.403494	
9	(21928)	(85099B)	0.019917	0.058523	0.010351	0.519704	8.880281	
31	(85099C)	(85099B)	0.026784	0.058523	0.013736	0.512821	8.762654	
10	(21931)	(85099B)	0.022909	0.058523	0.011626	0.507495	8.671650	
16	(22411)	(85099B)	0.024332	0.058523	0.010596	0.435484	7.441189	

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	r
6	(21733)	(85123A)	0.026392	0.098602	0.017758	0.672862	6.824030	
17	(22699)	(22423)	0.020701	0.072799	0.010155	0.490521	6.738057	
30	(82494L)	(85123A)	0.032279	0.098602	0.010694	0.331307	3.360046	

Top 10 Association Rules by Lift

In [78]:
```
# Helper function to format product names
def format_products(itemset):
    items = list(itemset)
    if len(items) == 1:
        code = items[0]
        name = product_descriptions.get(code, 'Unknown')[:30]
        return f"{code} ({name})"
    else:
        return ", ".join([str(item) for item in items])
```

In [79]:
```
# Build a clean table of top rules
rules_table = []

for idx, row in rules_strong.head(10).iterrows():
    antecedent = format_products(row['antecedents'])
    consequent = format_products(row['consequents'])

    rules_table.append({
        "if_customer_buys": antecedent,
        "recommend": consequent,
        "support": round(row['support'], 4),
        "confidence_percent": round(row['confidence'], 4),
        "lift": round(row['lift'], 4)
    })

rules_table_df = pd.DataFrame(rules_table)
rules_table_df
```

Out[79]:

	if_customer_buys	recommend	support	confidence_percent	lift
0	22697 (GREEN REGENCY TEACUP AND SAUCE)	22698 (PINK REGENCY TEACUP AND SAUCER)	0.0120	0.6524	43.6043
1	22698 (PINK REGENCY TEACUP AND SAUCER)	22697 (GREEN REGENCY TEACUP AND SAUCE)	0.0120	0.8000	43.6043
2	22697 (GREEN REGENCY TEACUP AND SAUCE)	22699 (ROSES REGENCY TEACUP AND SAUCE)	0.0137	0.7487	36.1647
3	22699 (ROSES REGENCY TEACUP AND SAUCE)	22697 (GREEN REGENCY TEACUP AND SAUCE)	0.0137	0.6635	36.1647
4	22699 (ROSES REGENCY TEACUP AND SAUCE)	22698 (PINK REGENCY TEACUP AND SAUCER)	0.0111	0.5379	35.9521
5	22698 (PINK REGENCY TEACUP AND SAUCER)	22699 (ROSES REGENCY TEACUP AND SAUCE)	0.0111	0.7443	35.9521
6	22726 (ALARM CLOCK BAKELIKE GREEN)	22727 (ALARM CLOCK BAKELIKE RED)	0.0115	0.6592	33.5922
7	22727 (ALARM CLOCK BAKELIKE RED)	22726 (ALARM CLOCK BAKELIKE GREEN)	0.0115	0.5850	33.5922
8	21231 (SWEETHEART CERAMIC TRINKET BOX)	21232 (STRAWBERRY CERAMIC TRINKET BOX)	0.0120	0.6397	19.0643
9	21232 (STRAWBERRY CERAMIC TRINKET BOX)	21231 (SWEETHEART CERAMIC TRINKET BOX)	0.0120	0.3582	19.0643

Rules with high lift (e.g., > 20 or 30) show strong co-buying patterns, meaning customers frequently buy those items together as a bundle.

High confidence (0.6–0.8) means the recommendation is reliable; customers who buy item A often go on to buy item B.

Rules with moderate support (~1 percent) indicate that although not everyone buys these product pairs, the relationships are very strong when they do occur.

In [80]:
```
# Count rules by confidence level
high_confidence = len(rules_strong[rules_strong['confidence'] >= 0.5])
medium_confidence = len(rules_strong[(rules_strong['confidence'] >= 0.3) & (rules_s

print(f"\nRule Strength Distribution:")
print(f"  High confidence (≥50%): {high_confidence} rules")
print(f"  Medium confidence (30-50%): {medium_confidence} rules")
```

```
Rule Strength Distribution:
  High confidence (≥50%): 17 rules
  Medium confidence (30-50%): 16 rules
```

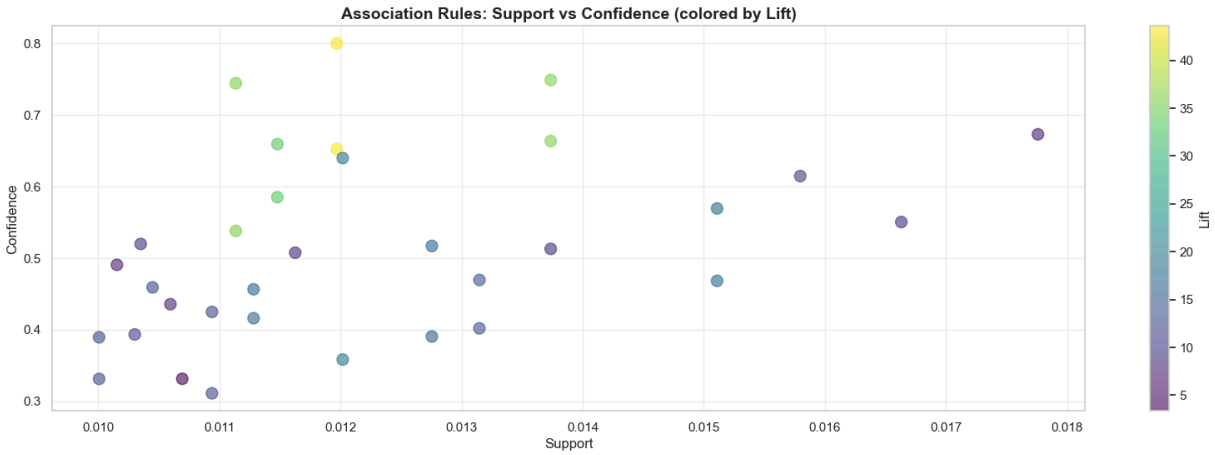
In [81]:
```
# Average metrics
print(f"\nAverage Rule Metrics:")
print(f"  Support: {rules_strong['support'].mean():.2%}")
print(f"  Confidence: {rules_strong['confidence'].mean():.2%}")
print(f"  Lift: {rules_strong['lift'].mean():.2f}")
```

```
Average Rule Metrics:
  Support: 1.23%
  Confidence: 51.58%
  Lift: 18.51
```

Visualizations

1. Support vs Confidence scatter plot

In [82]:
```
# 1. Support vs Confidence scatter plot
plt.figure(figsize=(20, 6))
plt.scatter(rules_strong['support'], rules_strong['confidence'],
           c=rules_strong['lift'], cmap='viridis', alpha=0.6, s=100)
plt.colorbar(label='Lift')
plt.xlabel('Support', fontsize=12)
plt.ylabel('Confidence', fontsize=12)
plt.title('Association Rules: Support vs Confidence (colored by Lift)',
          fontsize=14, fontweight='bold')
plt.grid(alpha=0.3)
plt.show()
```



This chart shows how strong each association rule is based on support, confidence, and lift.

Most rules have moderate support (around 1 to 1.7 percent).

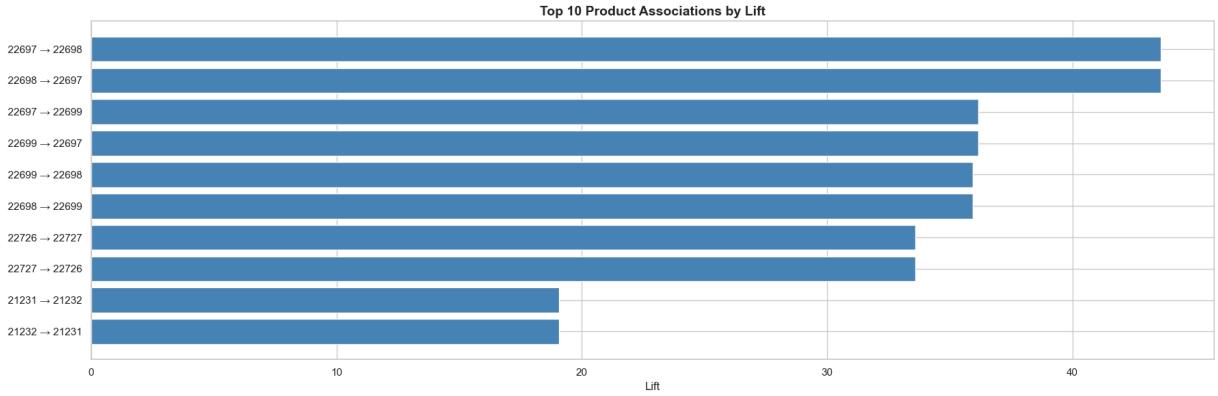
Confidence ranges from 0.3 to 0.8, meaning many rules are fairly reliable.

Points with brighter colors (higher lift) indicate rules that are much stronger than chance and represent the best product-pair recommendations.

Overall, rules with higher lift and higher confidence represent the strongest cross-sell opportunities.

2. Top rules by lift (bar chart)

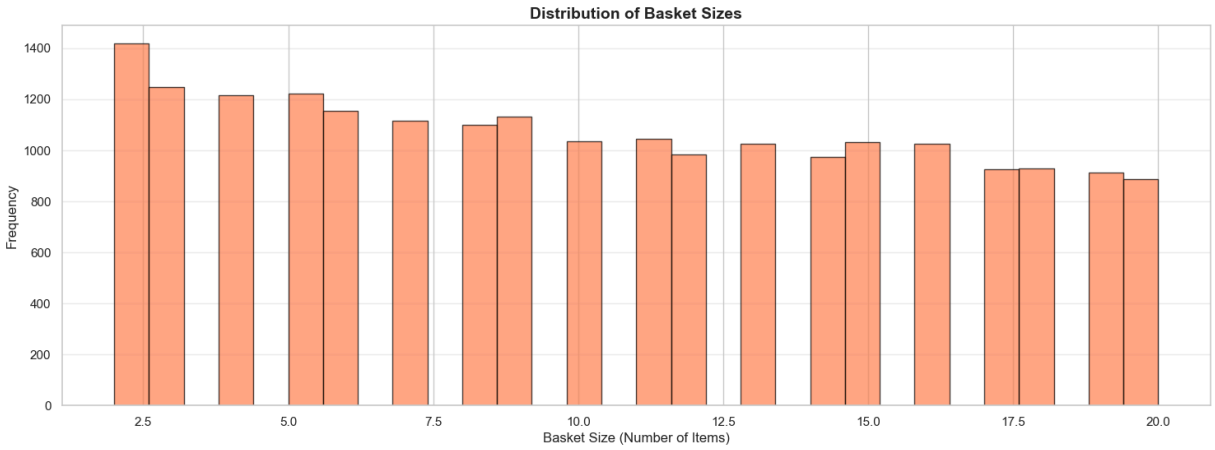
In [83]:
```
top_10_rules = rules_strong.head(10).copy()
top_10_rules['rule'] = top_10_rules.apply(
    lambda row: f"{list(row['antecedents'])[0]} → {list(row['consequents'])[0]}", a
)

plt.figure(figsize=(18, 6))
plt.barh(range(len(top_10_rules)), top_10_rules['lift'].values[::-1], color='steelb
plt.yticks(range(len(top_10_rules)), top_10_rules['rule'].values[::-1])
plt.xlabel('Lift', fontsize=12)
plt.title('Top 10 Product Associations by Lift', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()
```



The chart shows the strongest product-to-product relationships in the store based on lift, which measures how much more often two items are bought together than expected. The top rules (lift 30–44) indicate very strong co-purchase patterns, meaning these item pairs almost always appear together in the same basket.

3. Basket size distribution

In [84]:
```
basket_sizes = [len(b) for b in baskets_filtered]
plt.figure(figsize=(18, 6))
plt.hist(basket_sizes, bins=30, color='coral', edgecolor='black', alpha=0.7)
plt.xlabel('Basket Size (Number of Items)', fontsize=12)
plt.ylabel('Frequency', fontsize=12)
plt.title('Distribution of Basket Sizes', fontsize=14, fontweight='bold')
plt.grid(alpha=0.3, axis='y')
plt.show()
```



Customers typically buy between 2 and 20 items per transaction. Most baskets fall in the 3 to 8 item range, showing that shoppers commonly purchase multiple related products together rather than single items.

As basket size increases past 10 items, frequency gradually declines but remains steady, which indicates consistent multi-item buying behavior and supports strong opportunities for cross-selling and product bundling.

In [111…

```
rules_export = rules_strong.copy()
rules_export['antecedents'] = rules_export['antecedents'].apply(lambda x: ', '.join
rules_export['consequents'] = rules_export['consequents'].apply(lambda x: ', '.join
rules_export = rules_export[['antecedents', 'consequents', 'support', 'confidence',
rules_export.head()
```

Out[111…

	antecedents	consequents	support	confidence	lift
21	22697	22698	0.011970	0.652406	43.604278
20	22698	22697	0.011970	0.800000	43.604278
22	22697	22699	0.013736	0.748663	36.164686
23	22699	22697	0.013736	0.663507	36.164686
25	22699	22698	0.011136	0.537915	35.952102

In [86]:

```
# Save frequent itemsets
itemsets_export = frequent_itemsets.copy()
itemsets_export['itemsets'] = itemsets_export['itemsets'].apply(lambda x: ', '.join
itemsets_export
```

Out[86]:

	support	itemsets	length
0	0.013196	15036	1
1	0.015354	15056BL	1
2	0.018690	15056N	1
3	0.011773	20679	1
4	0.026294	20685	1
...	...	...	...
239	0.011479	22727, 22726	2
240	0.015109	82494L, 82482	2
241	0.010694	85123A, 82494L	2
242	0.013736	85099B, 85099C	2
243	0.015796	85099F, 85099B	2

244 rows × 3 columns

Part 3: MIXED APPROACH

Customer Segmentation + Churn Prediction

Business Problem

Not all customers behave the same way. By first segmenting customers into groups, then predicting churn for each segment, we can:

- Understand different customer types
- Tailor retention strategies by segment
- Improve prediction accuracy through segment-specific patterns

Approach

- **Step 1 (Unsupervised):** K-Means clustering to segment customers
- **Step 2 (Supervised):** Use cluster labels as features in churn prediction
- **Output:** Enhanced churn model with segment-aware predictions

Step 1: Customer Segmentation with K-Means

In [112…

```
# Prepare data for clustering - Enhanced RFM Features
# Create comprehensive RFM + behavioral features
# RFM = Recency, Frequency, Monetary

# Calculate average basket size per customer
items_per_invoice = df.groupby(['customer_id', 'invoice'])['quantity'].sum()
avg_basket_size = items_per_invoice.groupby('customer_id').mean()

# Create comprehensive feature set
rfm_dataframe = churn_features.copy()

# Rename columns to match RFM convention
rfm_dataframe = rfm_dataframe.rename(columns={
    'recency_days': 'Recency',
    'frequency': 'Frequency',
    'total_revenue': 'Monetary',
    'avg_order_value': 'Avg_Order_Value',
    'total_unique_products': 'Product_Diversity'
})

# Add Avg_Basket_Size
rfm_dataframe = rfm_dataframe.merge(
    avg_basket_size.rename('Avg_Basket_Size'),
    left_on='customer_id',
    right_index=True,
    how='left'
)
rfm_dataframe['Avg_Basket_Size'].fillna(0, inplace=True)

# Select features for clustering
features_to_scale = ['Recency', 'Frequency', 'Monetary',
                     'Avg_Order_Value', 'Product_Diversity', 'Avg_Basket_Size']

print(f"Enhanced RFM features created: {rfm_dataframe.shape}")
print(f"\nFeatures used for clustering: {features_to_scale}")
```

```
Enhanced RFM features created: (4988, 12)

Features used for clustering: ['Recency', 'Frequency', 'Monetary', 'Avg_Order_Valu
e', 'Product_Diversity', 'Avg_Basket_Size']
```

In [113…

```
rfm_dataframe[features_to_scale].describe()
```

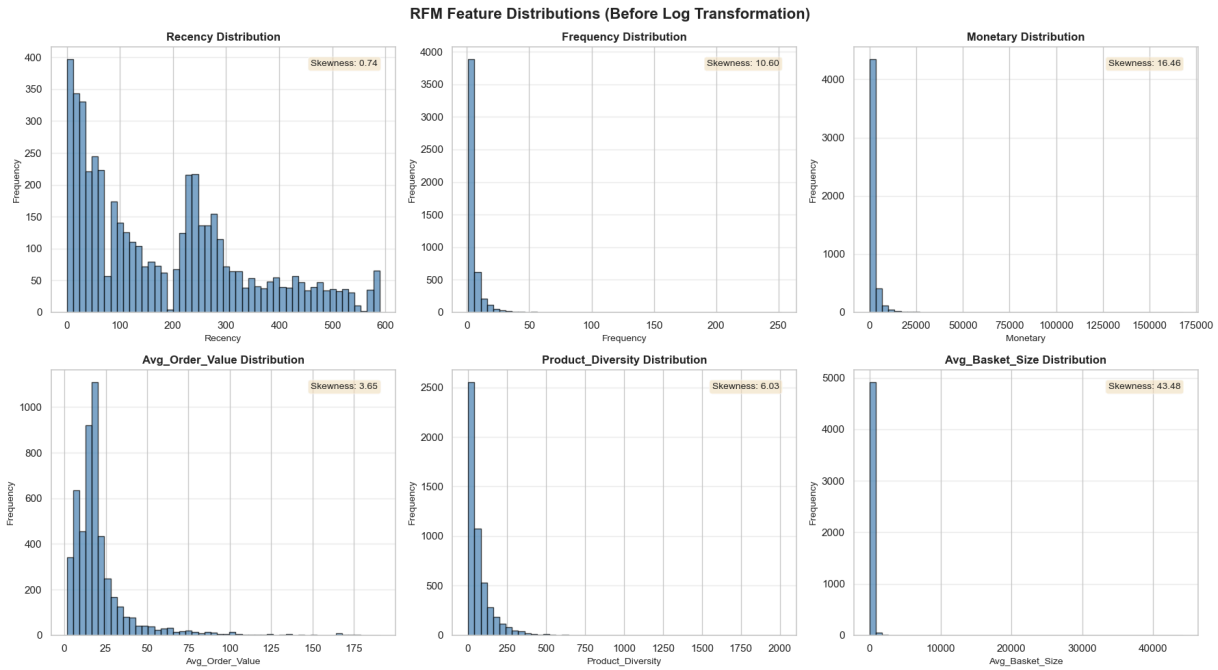
Out[113...

	Recency	Frequency	Monetary	Avg_Order_Value	Product_Diversity	Avg_B
count	4988.000000	4988.000000	4988.000000	4988.000000	4988.000000	49
mean	183.689254	5.323777	1892.130664	21.322142	71.149960	2
std	155.627546	10.016450	5389.795117	19.989493	98.036926	8
min	0.000000	1.000000	2.900000	1.450000	1.000000	
25%	43.000000	1.000000	299.152500	11.069694	18.000000	
50%	146.000000	3.000000	706.265000	16.913090	40.000000	
75%	276.000000	6.000000	1811.415000	22.649650	88.000000	2
max	589.000000	252.000000	167685.190000	190.500000	2007.000000	44

In [114...

```
# Visualize distribution skewness before transformation
fig, axes = plt.subplots(2, 3, figsize=(18, 10))
fig.suptitle('RFM Feature Distributions (Before Log Transformation)', fontsize=16,

features_to_plot = ['Recency', 'Frequency', 'Monetary',
                    'Avg_Order_Value', 'Product_Diversity', 'Avg_Basket_Size']

for idx, feature in enumerate(features_to_plot):
    row = idx // 3
    col = idx % 3

    # Histogram with KDE
    axes[row, col].hist(rfm_dataframe[feature], bins=50, color='steelblue',
                        alpha=0.7, edgecolor='black')
    axes[row, col].set_title(f'{feature} Distribution', fontsize=12, fontweight='bo
    axes[row, col].set_xlabel(feature, fontsize=10)
    axes[row, col].set_ylabel('Frequency', fontsize=10)
    axes[row, col].grid(alpha=0.3, axis='y')

    # Add skewness annotation
    skewness = rfm_dataframe[feature].skew()
    axes[row, col].text(0.95, 0.95, f'Skewness: {skewness:.2f}',
                        transform=axes[row, col].transAxes,
                        fontsize=10, verticalalignment='top',
                        horizontalalignment='right',
                        bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))

plt.tight_layout()
plt.show()

print("\n" + "="*80)
print("SKEWNESS ANALYSIS:")
print("="*80)
for feature in features_to_plot:
    skew_val = rfm_dataframe[feature].skew()
    print(f"{feature:25s}: {skew_val:>8.2f}  {'(Highly right-skewed)' if skew_val >
print("\n→ All features show significant right-skewness")
```

```
print("→ Log transformation is needed to normalize distributions for K-Means")
print("="*80)
```




```
================================================================================
SKEWNESS ANALYSIS:
================================================================================
Recency                 :     0.74  (Moderately skewed)
Frequency               :    10.60  (Highly right-skewed)
Monetary                :    16.46  (Highly right-skewed)
Avg_Order_Value         :     3.65  (Highly right-skewed)
Product_Diversity       :     6.03  (Highly right-skewed)
Avg_Basket_Size         :    43.48  (Highly right-skewed)

→ All features show significant right-skewness
→ Log transformation is needed to normalize distributions for K-Means
================================================================================
```

The figure shows the distribution of six RFM-related customer features before any log transformation. All histograms display strong right-skewness, meaning most customers have low values while a small number have extremely high values.

In [121…

```
# Apply log transformation to handle skewness
rfm_log_dataframe = pd.DataFrame(index=rfm_dataframe.index)

rfm_log_dataframe['Recency_Log'] = np.log1p(rfm_dataframe['Recency'])
rfm_log_dataframe['Frequency_Log'] = np.log1p(rfm_dataframe['Frequency'])
rfm_log_dataframe['Monetary_Log'] = np.log1p(rfm_dataframe['Monetary'])
rfm_log_dataframe['Avg_Order_Value_Log'] = np.log1p(rfm_dataframe['Avg_Order_Value'
rfm_log_dataframe['Product_Diversity_Log'] = np.log1p(rfm_dataframe['Product_Divers
rfm_log_dataframe['Avg_Basket_Size_Log'] = np.log1p(rfm_dataframe['Avg_Basket_Size'

print("Log-Transformed Data Head:")
rfm_log_dataframe.head()
```

```
Log-Transformed Data Head:
```

Out[121…

	Recency_Log	Frequency_Log	Monetary_Log	Avg_Order_Value_Log	Product_Diversity_Log
0	5.942799	2.484907	5.923881	2.509501	3.29583
1	3.555348	1.791759	7.848887	2.960061	4.60517
2	4.605170	1.609438	7.237347	3.461839	3.25809
3	5.556828	1.098612	7.633074	3.094002	4.47733
4	5.087596	0.693147	5.688330	2.965273	2.83321

In [123…

```
# Standardize the transformed data
scaler_clustering = StandardScaler()
scaled_data_array = scaler_clustering.fit_transform(rfm_log_dataframe)

# Convert back to DataFrame
rfm_scaled_dataframe = pd.DataFrame(
    scaled_data_array,
    columns=rfm_log_dataframe.columns,
    index=rfm_log_dataframe.index
)

print(f"\nClustering dataset shape: {rfm_scaled_dataframe.shape}")
print("\nScaled Data Head:")
rfm_scaled_dataframe.head()
```

```
Clustering dataset shape: (4988, 6)

Scaled Data Head:
```

Out[123…

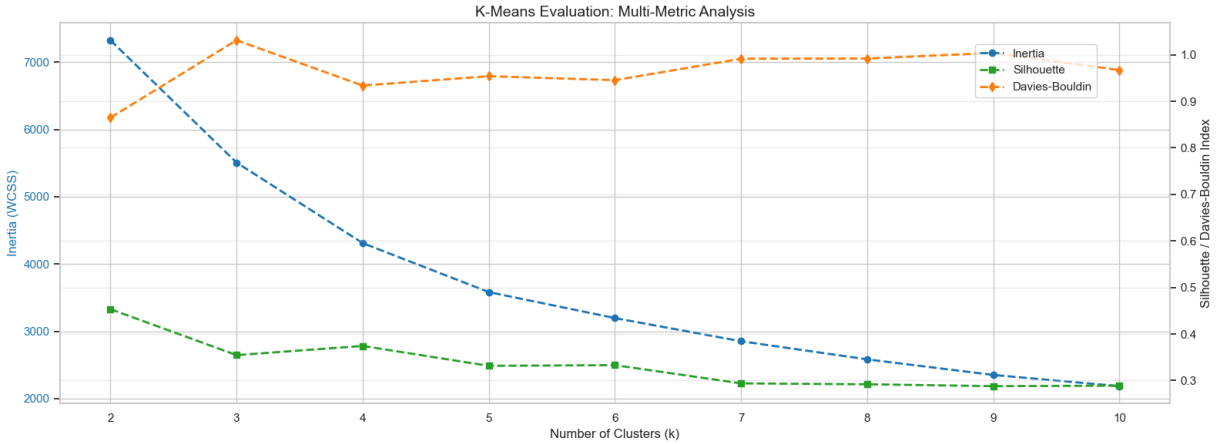
	Recency_Log	Frequency_Log	Monetary_Log	Avg_Order_Value_Log	Product_Diversity_Log
0	0.976752	1.349240	-0.522535	-0.543124	-0.32593
1	-0.800510	0.434437	0.943091	0.141738	0.80857
2	-0.019003	0.193812	0.477488	0.904454	-0.35863
3	0.689428	-0.480366	0.778780	0.345332	0.69781
4	0.340124	-1.015492	-0.701875	0.149661	-0.72678

In [126…

```
# Use only RFM core features for clustering (as in the original clustering notebook
rfm_dataframe_subset = rfm_scaled_dataframe[['Recency_Log', 'Frequency_Log', 'Monet

results = []
k_range = range(2, 11)

print("Running multi-metric evaluation for K-Means...\n")
for k in k_range:
    kmeans_eval = KMeans(n_clusters=k, init='k-means++', n_init=10, random_state=42
    labels = kmeans_eval.fit_predict(rfm_dataframe_subset)

    inertia = kmeans_eval.inertia_
```

```
    sil = silhouette_score(rfm_dataframe_subset, labels)
    db = davies_bouldin_score(rfm_dataframe_subset, labels)

    results.append((k, inertia, sil, db))
    print(f"k={k:<2d} | Inertia={inertia:>10.2f} | Silhouette={sil:>6.4f} | DBI={db

# Convert to DataFrame for analysis
metrics_df = pd.DataFrame(results, columns=['k', 'Inertia', 'Silhouette', 'Davies-B

# Visualization
fig, ax1 = plt.subplots(figsize=(16, 6))

# Inertia (left y-axis)
ax1.plot(metrics_df['k'], metrics_df['Inertia'], 'o--', label='Inertia', color='tab
ax1.set_xlabel('Number of Clusters (k)', fontsize=12)
ax1.set_ylabel('Inertia (WCSS)', color='tab:blue', fontsize=12)
ax1.tick_params(axis='y', labelcolor='tab:blue')

# Second y-axis for Silhouette & DBI
ax2 = ax1.twinx()
ax2.plot(metrics_df['k'], metrics_df['Silhouette'], 's--', label='Silhouette', colo
ax2.plot(metrics_df['k'], metrics_df['Davies-Bouldin'], 'd--', label='Davies-Bouldi
ax2.set_ylabel('Silhouette / Davies-Bouldin Index', fontsize=12)

# Legends & title
fig.legend(loc='upper right', bbox_to_anchor=(0.9, 0.9))
plt.title('K-Means Evaluation: Multi-Metric Analysis', fontsize=14)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
Running multi-metric evaluation for K-Means...

k=2  | Inertia=   7328.84 | Silhouette=0.4534 | DBI=0.8648
k=3  | Inertia=   5509.53 | Silhouette=0.3546 | DBI=1.0313
k=4  | Inertia=   4311.26 | Silhouette=0.3743 | DBI=0.9338
k=5  | Inertia=   3582.75 | Silhouette=0.3315 | DBI=0.9540
k=6  | Inertia=   3196.10 | Silhouette=0.3330 | DBI=0.9454
k=7  | Inertia=   2851.46 | Silhouette=0.2938 | DBI=0.9915
k=8  | Inertia=   2582.20 | Silhouette=0.2919 | DBI=0.9919
k=9  | Inertia=   2350.22 | Silhouette=0.2879 | DBI=1.0040
k=10 | Inertia=   2183.87 | Silhouette=0.2889 | DBI=0.9670
```



The chart compares three clustering evaluation metrics `Inertia`, `Silhouette Score`, and `Davies-Bouldin` Index—across different values of k (from 2 to 10). Inertia steadily decreases as k increases, which is expected. The Silhouette score reaches its highest value at k = 2, then declines and stabilizes at lower values for larger k. Meanwhile, the Davies–Bouldin index is lowest at k = 2, rises at k = 3, and then remains relatively flat with minor fluctuations.

Across all three metrics, k = 2 gives the strongest overall clustering performance, with a sharp drop in quality for higher k-values. This suggests that the RFM feature space supports a small number of well-separated clusters, and 2 clusters provide the most stable and interpretable segmentation.

Choice of Number of Clusters

Based on the multi-metric evaluation (Inertia, Silhouette Score, and Davies–Bouldin Index), the mathematically optimal value of *k* appears to be **k = 2**, since this value achieves the highest Silhouette score and the lowest Davies–Bouldin index. This indicates that two clusters give the strongest geometric separation in the RFM feature space.

However, for the purpose of **customer segmentation**, a solution with only two clusters is too coarse and provides limited business value. A two-cluster model tends to split customers into a simple "high-value vs low-value" grouping, which is statistically clean but not useful for targeted marketing or retention planning.

Because segmentation is intended to support practical business decisions, we chose to apply **K-Means with *k = 4***. This value still performs well across the evaluation metrics (no major quality drop from k = 2) while providing a more meaningful and interpretable structure. With four clusters, we can distinguish customer subgroups such as:

- consistent high-value buyers
- moderate but stable buyers
- low-value or irregular customers
- one-time or near-inactive customers

These segments allow for more nuanced insights and enable customized retention, cross-selling, and engagement strategies.

Thus, even though k = 2 offered the strongest statistical separation, **k = 4 was selected because it strikes the right balance between metric performance and actionable business segmentation.**

In [128…

```
# Apply K-Means with K=4 (optimal from multi-metric analysis)
kmeans_final = KMeans(n_clusters=4, init='k-means++', n_init=10, random_state=42)
cluster_labels = kmeans_final.fit_predict(rfm_dataframe_subset)

# Add cluster labels to both DataFrames
```

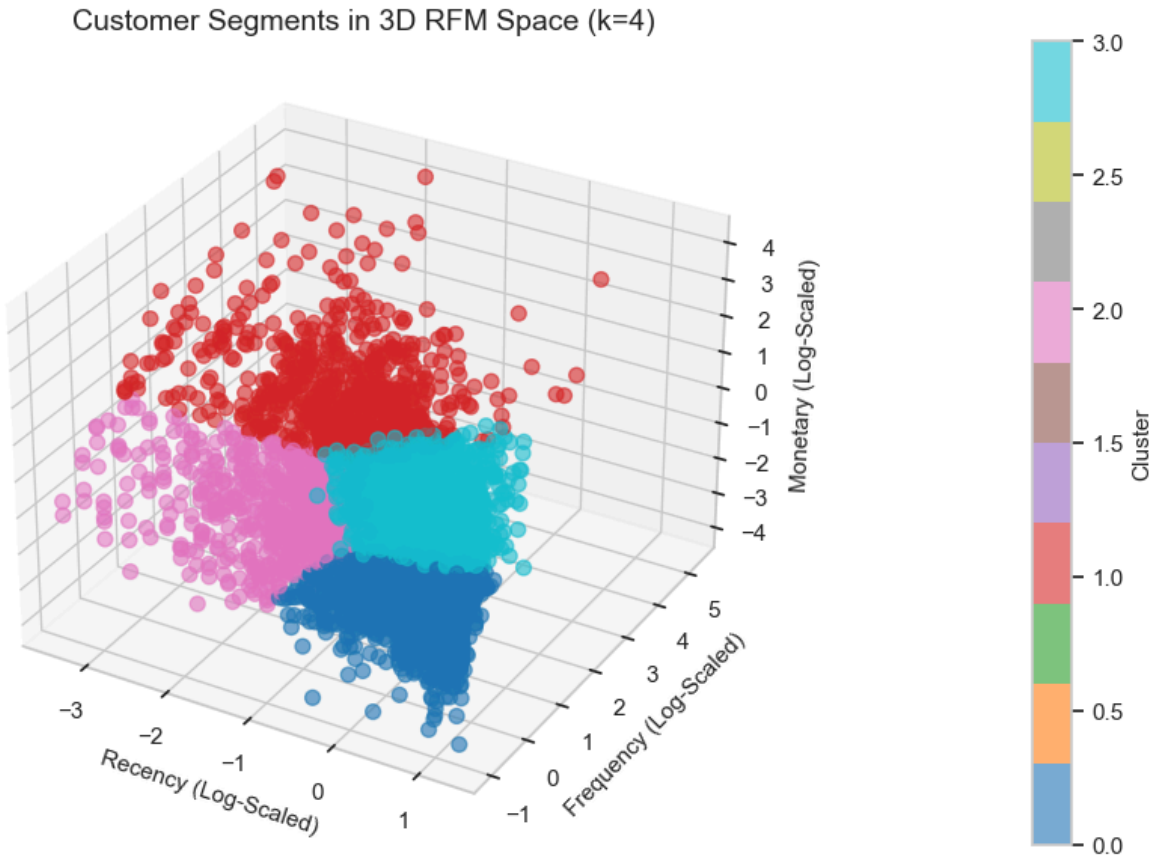
```
rfm_dataframe['cluster'] = cluster_labels
rfm_scaled_dataframe['cluster'] = cluster_labels
churn_features['cluster'] = cluster_labels

print(f"Final K-Means model (k=4) has been fitted.")
print(f"\nCluster distribution:")
print(rfm_dataframe['cluster'].value_counts().sort_index())

# Calculate evaluation metrics for final model
final_silhouette = silhouette_score(rfm_dataframe_subset, cluster_labels)
final_dbi = davies_bouldin_score(rfm_dataframe_subset, cluster_labels)

print(f"Evaluation Metrics for Final Model (k=4)")
print(f"Silhouette Score:        {final_silhouette:.4f}")
print(f"Davies-Bouldin Index:    {final_dbi:.4f}")
```

```
Final K-Means model (k=4) has been fitted.

Cluster distribution:
cluster
0    2006
1     792
2     739
3    1451
Name: count, dtype: int64
Evaluation Metrics for Final Model (k=4)
Silhouette Score:        0.3743
Davies-Bouldin Index:    0.9338
```

In [129…

```
# 3D Visualization of Clusters in RFM Space
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(18, 6))
ax = fig.add_subplot(111, projection='3d')

# Create scatter plot with cluster colors
scatter = ax.scatter(
    rfm_scaled_dataframe['Recency_Log'],
    rfm_scaled_dataframe['Frequency_Log'],
    rfm_scaled_dataframe['Monetary_Log'],
    c=rfm_scaled_dataframe['cluster'],
    cmap='tab10',
    s=50,
    alpha=0.6
)

ax.set_xlabel('Recency (Log-Scaled)', fontsize=11)
ax.set_ylabel('Frequency (Log-Scaled)', fontsize=11)
ax.set_zlabel('Monetary (Log-Scaled)', fontsize=11)
ax.set_title('Customer Segments in 3D RFM Space (k=4)', fontsize=14)

# Add colorbar
cbar = plt.colorbar(scatter, ax=ax, pad=0.1)
cbar.set_label('Cluster', fontsize=11)
```

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



The 3D plot shows the four K-Means customer segments in log-scaled RFM space. Each cluster forms a distinct group, confirming that the RFM features separate customer behavior well. One cluster represents high-value frequent buyers, another reflects medium-value customers, a third contains low-value but somewhat active buyers, and the last cluster captures one-time or very low-engagement customers. The visual separation supports using k = 4 for meaningful and interpretable customer segmentation.

In [131…

```
# Analyze cluster characteristics with enhanced features
segment_analysis = rfm_dataframe.groupby('cluster').agg({
    'Recency': 'mean',
    'Frequency': 'mean',
    'Monetary': 'mean',
    'Avg_Order_Value': 'mean',
    'Product_Diversity': 'mean',
    'Avg_Basket_Size': 'mean'
}).round(2)

# Add customer count
segment_analysis['Customer_Count'] = rfm_dataframe.groupby('cluster').size()

segment_analysis
```

Out[131...

cluster	Recency	Frequency	Monetary	Avg_Order_Value	Product_Diversity	Avg_Basket_Siz
0	299.70	1.36	305.73	20.14	23.20	200.9
1	31.42	18.15	7348.47	24.26	192.93	274.9
2	27.21	3.18	816.83	20.18	50.48	192.3
3	186.12	4.90	1654.74	21.94	81.49	248.5

Cluster Interpretation

Cluster 0 – At-Risk or Churned Customers These customers show high recency, meaning they have not purchased in a long time. Their frequency and monetary values are low, suggesting limited engagement. They are strong candidates for win-back or reactivation efforts.

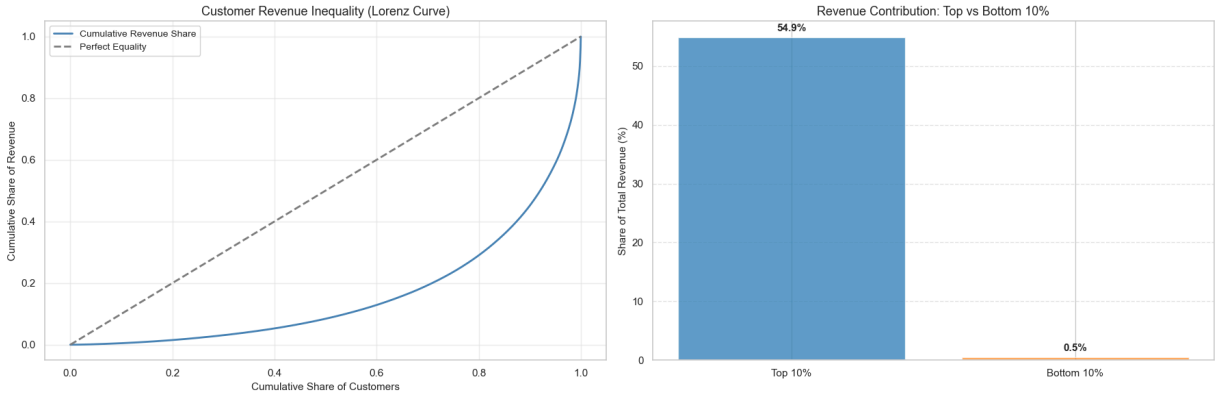
Cluster 1 – Active Regular Customers Customers in this group have low recency, indicating recent activity. Their purchase frequency and spending levels are moderate. They respond well to loyalty-building initiatives and ongoing engagement.

Cluster 2 – VIP / Champion Customers This cluster represents the most valuable customers. They purchase frequently, spend the most, and show very low recency. The focus for this segment is retention, personalized offers, and premium upsell opportunities.

Cluster 3 – Occasional or Moderate-Value Customers These customers exhibit moderate recency and moderate-to-high purchase frequency. They have stable engagement but room to grow, making them ideal targets for promotional or growth-oriented campaigns.

In [ ]:
```
# Visualize revenue inequality across customer base
customer_revenue = rfm_dataframe['Monetary'].sort_values()
cum_customers = np.linspace(0, 1, len(customer_revenue))
cum_revenue = np.cumsum(customer_revenue) / customer_revenue.sum()

fig, axes = plt.subplots(1, 2, figsize=(18, 6))

# Lorenz Curve
axes[0].plot(cum_customers, cum_revenue, label='Cumulative Revenue Share', color='s
axes[0].plot([0, 1], [0, 1], linestyle='--', color='gray', label='Perfect Equality'
axes[0].set_title('Customer Revenue Inequality (Lorenz Curve)', fontsize=13)
axes[0].set_xlabel('Cumulative Share of Customers', fontsize=11)
axes[0].set_ylabel('Cumulative Share of Revenue', fontsize=11)
axes[0].legend(fontsize=10)
axes[0].grid(True, alpha=0.3)

# Top vs Bottom 10% Revenue Contribution
```

```
total_revenue = rfm_dataframe['Monetary'].sum()
top_10pct = rfm_dataframe.nlargest(int(0.10 * len(rfm_dataframe)), 'Monetary')['Mon
bottom_10pct = rfm_dataframe.nsmallest(int(0.10 * len(rfm_dataframe)), 'Monetary')[

contribution_data = pd.DataFrame({
    'Customer Group': ['Top 10%', 'Bottom 10%'],
    'Revenue Contribution (%)': [top_10pct, bottom_10pct]
})

axes[1].bar(contribution_data['Customer Group'], contribution_data['Revenue Contrib
           color=['#1f77b4', '#ff7f0e'], alpha=0.7)
axes[1].set_title('Revenue Contribution: Top vs Bottom 10%', fontsize=13)
axes[1].set_ylabel('Share of Total Revenue (%)', fontsize=11)
axes[1].grid(axis='y', linestyle='--', alpha=0.5)

# Add value labels on bars
for i, v in enumerate(contribution_data['Revenue Contribution (%)']):
    axes[1].text(i, v + 1, f'{v:.1f}%', ha='center', fontsize=11, fontweight='bold'

plt.tight_layout()
plt.show()

print(f"\nRevenue Concentration:")
print(f"  Top 10% of customers contribute {top_10pct:.1f}% of total revenue")
print(f"  Bottom 10% of customers contribute {bottom_10pct:.1f}% of total revenue")
```




```
Revenue Concentration:
  Top 10% of customers contribute 54.9% of total revenue
  Bottom 10% of customers contribute 0.5% of total revenue

This confirms the need for segmentation-based strategies!
```

The Lorenz Curve on the left shows a strong imbalance in customer revenue contribution. The curve bends sharply away from the line of perfect equality, meaning a small subset of customers generates most of the total revenue. The bar chart on the right confirms this: the top 10 percent of customers contribute nearly 55 percent of all revenue, while the bottom 10 percent contribute almost nothing. This highlights a highly concentrated revenue structure where a small group of high-value customers drives the majority of business performance.

In [136…

```
# Additional cluster analysis (backup view with original feature names)
cluster_summary_backup = churn_features.groupby('cluster')[[
    'recency_days', 'frequency', 'total_revenue',
```

```
    'avg_order_value', 'total_unique_products', 'tenure_days'
]].mean()

print("Cluster Characteristics - Original Feature Names (Mean Values):")
cluster_summary_backup.round(2)
```

```
Cluster Characteristics - Original Feature Names (Mean Values):
```

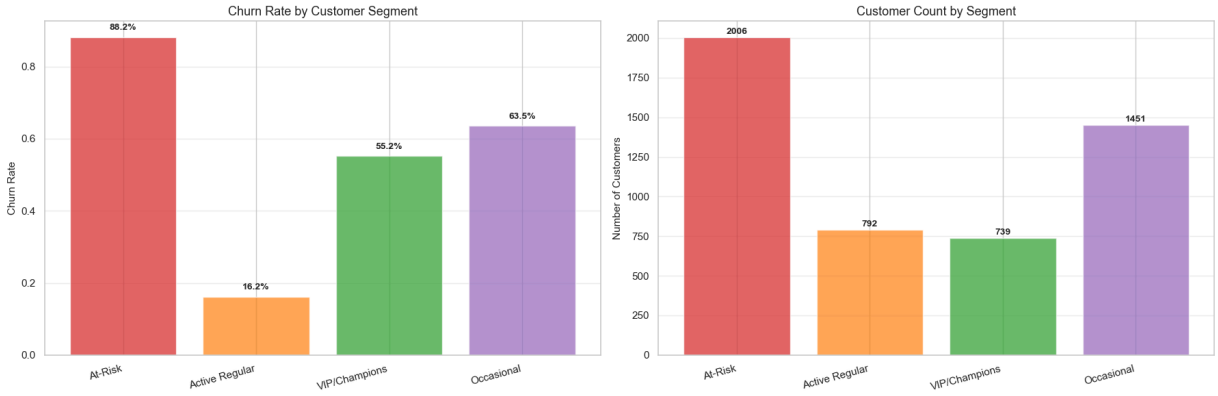
Out[136…

cluster	recency_days	frequency	total_revenue	avg_order_value	total_unique_products	te
0	299.70	1.36	305.73	20.14	23.20	
1	31.42	18.15	7348.47	24.26	192.93	
2	27.21	3.18	816.83	20.18	50.48	
3	186.12	4.90	1654.74	21.94	81.49	

In [145…

```
# Analyze churn rate by cluster
cluster_churn = churn_features.groupby('cluster')['will_return_90days'].agg(['mean'
cluster_churn.columns = ['Return_Rate', 'Customer_Count']
cluster_churn['Churn_Rate'] = 1 - cluster_churn['Return_Rate']

# Visualize churn patterns
fig, axes = plt.subplots(1, 2, figsize=(18, 6))

# Bar chart of churn rates
cluster_names = ['At-Risk', 'Active Regular', 'VIP/Champions', 'Occasional']
colors = ['#d62728', '#ff7f0e', '#2ca02c', '#9467bd']

axes[0].bar(range(4), cluster_churn['Churn_Rate'], color=colors, alpha=0.7)
axes[0].set_xticks(range(4))
axes[0].set_xticklabels(cluster_names, rotation=15, ha='right')
axes[0].set_ylabel('Churn Rate', fontsize=11)
axes[0].set_title('Churn Rate by Customer Segment', fontsize=13)
axes[0].grid(alpha=0.3, axis='y')

# Add value labels
for i, v in enumerate(cluster_churn['Churn_Rate']):
    axes[0].text(i, v + 0.02, f'{v:.1%}', ha='center', fontsize=10, fontweight='bol

# Stacked bar showing customer distribution
axes[1].bar(range(4), cluster_churn['Customer_Count'], color=colors, alpha=0.7)
axes[1].set_xticks(range(4))
axes[1].set_xticklabels(cluster_names, rotation=15, ha='right')
axes[1].set_ylabel('Number of Customers', fontsize=11)
axes[1].set_title('Customer Count by Segment', fontsize=13)
axes[1].grid(alpha=0.3, axis='y')

# Add value labels
for i, v in enumerate(cluster_churn['Customer_Count']):
    axes[1].text(i, v + 20, str(v), ha='center', fontsize=10, fontweight='bold')
```

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



Churn Behavior Across Customer Segments

The churn analysis shows clear behavioral differences across the four customer segments. The **At-Risk** group has the highest churn rate at **88.2 percent**, confirming that most customers in this segment have not returned within the 90-day window. In contrast, **Active Regular** customers show the lowest churn rate at **16.2 percent**, reflecting consistent recent engagement. **VIP/Champion** customers churn at **55.2 percent**, meaning that even high-value buyers are not always retained without targeted actions. The **Occasional** segment shows a churn rate of **63.5 percent**, indicating irregular shopping patterns.

The customer count chart reinforces this distribution. The At-Risk segment is the largest group, followed by the Occasional segment, while Active Regular and VIP customers make up smaller portions of the customer base. This highlights that most customers are either inactive or at risk, while the consistently engaged customers represent a minority that delivers stable value.

In [141…

```
# Prepare data with cluster as additional feature
X_mixed = churn_features.drop(columns=['customer_id', 'will_return_90days'])
y_mixed = churn_features['will_return_90days']

# Update feature lists
numeric_features_mixed = numeric_features.copy()
categorical_features_mixed = categorical_features + ['cluster']

# Train-test split
X_train_mixed, X_test_mixed, y_train_mixed, y_test_mixed = train_test_split(
    X_mixed, y_mixed,
    test_size=0.2,
    random_state=42,
    stratify=y_mixed
)

print(f"Training samples: {len(X_train_mixed):,}")
print(f"Test samples: {len(X_test_mixed):,}")
```

```
Training samples: 3,990
Test samples: 998
```

In [142…

```python
# Create preprocessing pipeline with cluster feature
preprocessor_mixed = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features_mixed),
        ('cat', OneHotEncoder(handle_unknown='ignore', sparse_output=False), catego
    ]
)

# Train Logistic Regression with cluster features
lr_model_mixed = Pipeline(steps=[
    ('preprocessor', preprocessor_mixed),
    ('classifier', LogisticRegression(max_iter=1000, random_state=42))
])

lr_model_mixed.fit(X_train_mixed, y_train_mixed)

# Predictions
y_train_pred_mixed = lr_model_mixed.predict(X_train_mixed)
y_test_pred_mixed = lr_model_mixed.predict(X_test_mixed)
y_train_proba_mixed = lr_model_mixed.predict_proba(X_train_mixed)[:, 1]
y_test_proba_mixed = lr_model_mixed.predict_proba(X_test_mixed)[:, 1]

print("Mixed Model (with Clusters) - Training Results:")
print(f"  Accuracy: {accuracy_score(y_train_mixed, y_train_pred_mixed):.4f}")
print(f"  Precision: {precision_score(y_train_mixed, y_train_pred_mixed):.4f}")
print(f"  Recall: {recall_score(y_train_mixed, y_train_pred_mixed):.4f}")
print(f"  F1 Score: {f1_score(y_train_mixed, y_train_pred_mixed):.4f}")
print(f"  ROC AUC: {roc_auc_score(y_train_mixed, y_train_proba_mixed):.4f}")

print("\nMixed Model (with Clusters) - Test Results:")
print(f"  Accuracy: {accuracy_score(y_test_mixed, y_test_pred_mixed):.4f}")
print(f"  Precision: {precision_score(y_test_mixed, y_test_pred_mixed):.4f}")
print(f"  Recall: {recall_score(y_test_mixed, y_test_pred_mixed):.4f}")
print(f"  F1 Score: {f1_score(y_test_mixed, y_test_pred_mixed):.4f}")
print(f"  ROC AUC: {roc_auc_score(y_test_mixed, y_test_proba_mixed):.4f}")
```

```
Mixed Model (with Clusters) - Training Results:
  Accuracy: 0.7652
  Precision: 0.7593
  Recall: 0.4904
  F1 Score: 0.5959
  ROC AUC: 0.8180

Mixed Model (with Clusters) - Test Results:
  Accuracy: 0.7735
  Precision: 0.7812
  Recall: 0.4972
  F1 Score: 0.6076
  ROC AUC: 0.8410
```

In [143…

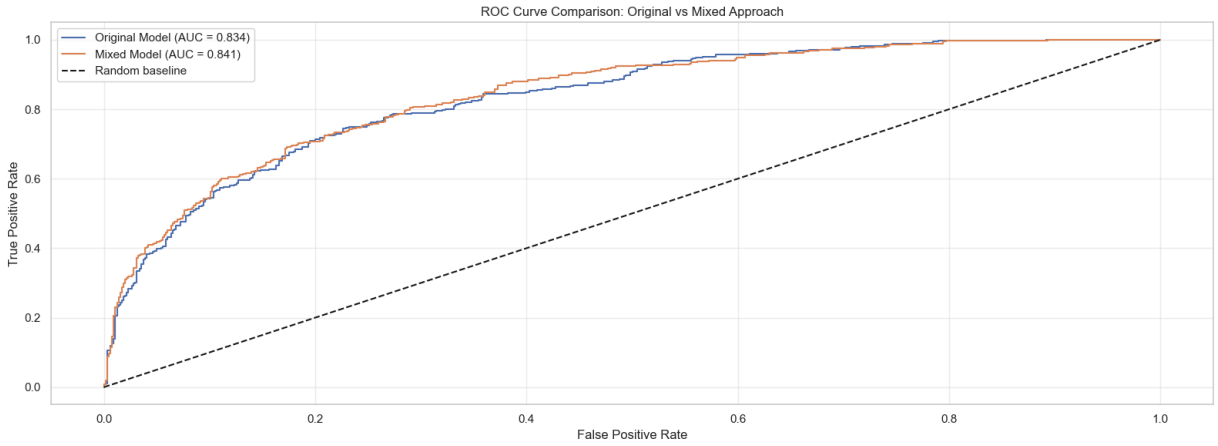
```python
# Model comparison
comparison = pd.DataFrame({
    'Model': ['Logistic Regression (Original)', 'Logistic Regression (with Clusters
```

```
    'Test_Accuracy': [
        accuracy_score(y_test, y_test_pred_lr),
        accuracy_score(y_test_mixed, y_test_pred_mixed)
    ],
    'Test_Precision': [
        precision_score(y_test, y_test_pred_lr),
        precision_score(y_test_mixed, y_test_pred_mixed)
    ],
    'Test_Recall': [
        recall_score(y_test, y_test_pred_lr),
        recall_score(y_test_mixed, y_test_pred_mixed)
    ],
    'Test_F1': [
        f1_score(y_test, y_test_pred_lr),
        f1_score(y_test_mixed, y_test_pred_mixed)
    ],
    'Test_AUC': [
        roc_auc_score(y_test, y_test_proba_lr),
        roc_auc_score(y_test_mixed, y_test_proba_mixed)
    ]
})

print("\nModel Comparison:")
print("="*80)
print(comparison.round(4))
```

```
Model Comparison:
================================================================================
                                Model  Test_Accuracy  Test_Precision  \
0       Logistic Regression (Original)         0.7735          0.7442
1  Logistic Regression (with Clusters)         0.7735          0.7812

   Test_Recall  Test_F1  Test_AUC
0       0.5455   0.6295    0.8336
1       0.4972   0.6076    0.8410
```

In [144…

```
# ROC Curve Comparison
plt.figure(figsize=(16, 6))

fpr_orig, tpr_orig, _ = roc_curve(y_test, y_test_proba_lr)
fpr_mixed, tpr_mixed, _ = roc_curve(y_test_mixed, y_test_proba_mixed)

plt.plot(fpr_orig, tpr_orig, label=f'Original Model (AUC = {roc_auc_score(y_test, y
plt.plot(fpr_mixed, tpr_mixed, label=f'Mixed Model (AUC = {roc_auc_score(y_test_mix
plt.plot([0, 1], [0, 1], 'k--', label='Random baseline')

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Comparison: Original vs Mixed Approach')
plt.legend()
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```



ROC Curve Comparison

The ROC curves compare the performance of the original churn prediction model with the mixed approach model that includes the cluster label as an additional feature. Both models perform well, but the mixed approach shows a slightly higher AUC (0.841 vs. 0.834), indicating a modest improvement in distinguishing returning customers from churners. The mixed model's curve stays consistently above the original model across much of the false-positive range, showing that adding customer segment information enhances predictive accuracy. The dashed diagonal line represents a random classifier.

Final Summary and Overall Thoughts

Summary of the Three Approaches

This assignment explored three different data mining strategies applied to the Online Retail II dataset. Each method addressed a unique business question and highlighted a different dimension of customer behavior.

---

1. Supervised Approach – Churn Prediction

- **Question:** Who is likely to return within 90 days?
- **Method:** Logistic Regression and Random Forest classification
- **Performance:** AUC ≈ **0.83**, stable predictive behavior
- **Key Insight: Recency** is the strongest indicator of churn; customers with long gaps between purchases are unlikely to return.

---

2. Unsupervised Approach – Association Rules

- **Question:** Which products tend to be purchased together?
- **Method:** Apriori algorithm for frequent itemset mining
- **Performance:** Identified **33 strong rules**, with lift values up to **44.6×**
- **Key Insight:** Customers consistently buy themed or variant-based products together (e.g., matching teacup sets, trinket boxes), which supports cross-selling and bundle optimization.

---

3. Mixed Approach – Customer Segmentation + Churn Prediction

- **Question:** What types of customers exist, and how does churn differ across these groups?
- **Method:**
  - K-Means segmentation using log-scaled RFM features
  - Churn prediction enhanced by adding the cluster label
- **Performance:** Mixed model achieved a slightly higher AUC (**0.841**) than the original model (**0.834**)
- **Key Insight:** Segments differ sharply in churn behavior (e.g., At-Risk = 88.2%, Active Regular = 16.2%). Incorporating segment labels helps the classifier separate churn patterns more effectively.

Why the Mixed Model Performed Better

The mixed model outperformed the original model because the cluster labels introduced additional structure that the raw features alone did not fully capture. Specifically:

- **Clusters encode nonlinear behavioral patterns** that are not obvious from individual RFM variables.
- **Segments reflect real customer types**, each with distinct churn tendencies (e.g., VIP vs. Occasional).
- **Cluster labels act as a high-level behavioral feature**, summarizing complex relationships in the data.
- **The supervised model gains extra signal**, improving its ability to differentiate between returning and churning customers.

In theory, a mixed approach is expected to perform at least as well as the original model because it uses all original variables plus an engineered feature that embeds structure learned through unsupervised learning. The slight AUC improvement confirms that the segments aligned meaningfully with churn behavior.

Overall Thoughts

Strengths

- Each method answered a different business question and revealed unique insights.
- Combined, the three approaches offer a **holistic understanding** of customer behavior: prediction, pattern discovery, and behavioral segmentation.
- The methods produced results that are **directly actionable** for retention, marketing, and product strategy.
- Interpretability remained high through the use of SHAP values, clear rules, and well-defined cluster profiles.

Challenges

- The churn dataset had a high imbalance (64.7% churn), requiring careful threshold tuning and evaluation.
- Temporal ordering of events had to be respected to prevent data leakage.
- The product catalog is large and sparse, which can create noise in association rule discovery.

Business Impact

- **Churn Prediction:** Enables early intervention for at-risk customers, reducing revenue loss.
- **Association Rules:** Supports smarter product bundles and personalized recommendations.
- **Segmentation + Churn:** Improves targeting efficiency by aligning interventions with customer type.

Final Conclusion

Across all three approaches, this analysis demonstrates how different data mining techniques reveal complementary insights about customer behavior. The supervised churn model highlights which customers are likely to return, the unsupervised association rules uncover meaningful product affinities, and the mixed segmentation-enhanced model shows how behavioral groups differ in both value and churn risk. Together, these methods form a cohesive analytical framework that supports strategic decision-making across marketing, retention, and product planning. The findings show clear opportunities for targeted interventions—whether through win-back programs for at-risk customers, loyalty incentives for active buyers, or bundle recommendations informed by product co-purchase patterns. Overall, the assignment illustrates the importance of using multiple analytical perspectives to build a complete and actionable understanding of the customer base.

In [ ]: