```
# Step 1: List files in the current directory
import os
import pandas as pd

print("Files in current directory:")
for file in os.listdir('.'):
    print(f"  - {file}")
```

```
Files in current directory:
  - .config
  - .ipynb_checkpoints
  - mlruns
  - drive
  - mlruns.db
  - Dataset_test - Sheet 1.csv
  - sample_data
```

```
# Step 2: Load the dataset
df = pd.read_csv('Dataset_test - Sheet 1.csv')

print(f"Dataset loaded successfully!")
print(f"Shape: {df.shape}")
print(f"\nColumns: {list(df.columns)}")
print(f"\nFirst 10 rows:")
df.head(10)
```

```
Dataset loaded successfully!
Shape: (10500, 31)

Columns: ['product_id', 'product_name', 'brand', 'category', 'ingredients', 'clean_label', 'price', 'discount', 'Average_units_sold

First 10 rows:
```

	product_id	product_name	brand	category	ingredients	clean_label	price	discount	Average_units_sold	units_sold	...	sentim
0	TWT_021	71% Dark Chocolate Sweetened with Dates - Pack...	The Whole Truth	Dark Chocolate	Cocoa Butter, Cocoa, Almonds, Cocoa Powder	Yes	1047	0	882	696	...	
1	TWT_024	Almond Millet Cocoa - Pack of 8 Millet Bars	The Whole Truth	Millet Bars	Dates, Millet, Natural Flavoring	No	1000	0	937	870	...	
2	TWT_024	Almond Millet Cocoa - Pack of 8 Millet Bars	The Whole Truth	Millet Bars	NaN	No	1000	0	937	1337	...	
3	TWT_019	CREAMY-Unsweetened Peanut Butter - Pack of 325g	The Whole Truth	Peanut Butter	NaN	Yes	225	0	1000	840	...	
4	TWT_023	Double Cocoa Mini Protein Bars - Box of 12	The Whole Truth	Mini Bars	NaN	Yes	720	0	844	500	...	
5	TWT_022	Almond Choco Fudge - Box of 10	The Whole Truth	Energy Bars	NaN	No	750	0	622	1308	...	
6	TWT_011	Lemon Cranberry Protein Bars - Box of 8	The Whole Truth	Protein Bars	Natural Flavoring, Almonds, Dates, Cocoa, Whey...	Yes	1000	0	722	604	...	
7	TWT_013	Peanut Butter Protein Bars - Box of 8	The Whole Truth	Protein Bars	Dates, Almonds, Natural Flavoring, Cocoa, Whey...	No	800	0	767	205	...	
8	TWT_022	Almond Choco Fudge - Box of 10	The Whole Truth	Energy Bars	Nuts, Dried Fruits, Seeds, Cocoa, Dates	No	750	0	863	651	...	
9	TWT_023	Double Cocoa Mini Protein Bars - Box of 12	The Whole Truth	Mini Bars	Protein Powder, Dates, Cocoa, Almonds	No	720	0	798	1285	...	

10 rows × 31 columns

```
# Step 3: Data Quality Check
print("="*60)
print("DATA QUALITY CHECK")
print("="*60)
print("\n1. MISSING VALUES:")
missing = df.isnull().sum()
print(missing[missing > 0])
print(f"\n2. DUPLICATES: {df.duplicated().sum()}")
```

```
============================================================
DATA QUALITY CHECK
============================================================

1. MISSING VALUES:
ingredients           148
```

```
shelf_life            118
packaging_type        111
reviewer_location     132
sentiment_category     20
customer_age_group     98
customer_gender       129
dtype: int64

2. DUPLICATES: 0
```

```
# Step 4: Column Analysis
print("="*80)
print("COMPREHENSIVE COLUMN ANALYSIS")
print("="*80)
print(f"\nTotal Columns: {len(df.columns)}\n")
for col in df.columns:
    print(f"{col:30s} {str(df[col].dtype):12s} {df[col].notna().sum():>6}/{len(df)}")
print("\n" + "="*80)
```

```
================================================================================
COMPREHENSIVE COLUMN ANALYSIS
================================================================================

Total Columns: 31

product_id                     object        10500/10500
product_name                   object        10500/10500
brand                          object        10500/10500
category                       object        10500/10500
ingredients                    object        10352/10500
clean_label                    object        10500/10500
price                          int64         10500/10500
discount                       int64         10500/10500
Average_units_sold             object        10500/10500
units_sold                     int64         10500/10500
average_rating                 float64       10500/10500
num_reviews                    int64         10500/10500
shelf_life                     object        10382/10500
packaging_type                 object        10389/10500
review_id                      object        10500/10500
review_text                    object        10500/10500
rating                         int64         10500/10500
platform                       object        10500/10500
date                           object        10500/10500
reviewer_location              object        10368/10500
sentiment_score                float64       10500/10500
sentiment_category             object        10480/10500
transaction_id                 object        10500/10500
purchase_date                  object        10500/10500
quantity                       int64         10500/10500
price_per_unit                 int64         10500/10500
region                         object        10500/10500
customer_age_group             object        10402/10500
customer_gender                object        10371/10500
season                         object        10500/10500
channel                        object        10500/10500

================================================================================
```

```
# Step 5: Check Required Features
print("="*80)
print("REQUIRED MODEL FEATURES CHECK")
print("="*80)

required = ['price', 'discount', 'category', 'clean_label', 'ingredients_count',
            'has_dates', 'has_cocoa', 'has_protein', 'channel', 'packaging_type', 'shelf_life']

print("\nChecking required features:")
for feat in required:
    status = 'EXISTS' if feat in df.columns else 'MISSING'
    print(f"{feat:25s} : {status}")

missing = [f for f in required if f not in df.columns]
print(f"\n\nTo create: {missing}")
print("\nDerivation logic:")
print("  ingredients_count : Count ingredients separated by comma")
print("  has_dates/cocoa/protein : Binary flags from ingredients text")
```

```
================================================================================
REQUIRED MODEL FEATURES CHECK
```

```
================================================================================

Checking required features:
price                     : EXISTS
discount                  : EXISTS
category                  : EXISTS
clean_label               : EXISTS
ingredients_count         : MISSING
has_dates                 : MISSING
has_cocoa                 : MISSING
has_protein               : MISSING
channel                   : EXISTS
packaging_type            : EXISTS
shelf_life                : EXISTS


To create: ['ingredients_count', 'has_dates', 'has_cocoa', 'has_protein']

Derivation logic:
  ingredients_count : Count ingredients separated by comma
  has_dates/cocoa/protein : Binary flags from ingredients text
```

Double-click (or enter) to edit

```
# ================================================================================
# STEP 5.5: CREATE DERIVED FEATURES FOR products_cleaned_v1.csv
# ================================================================================
import pandas as pd
import numpy as np

# Load the raw dataset
df_v1 = pd.read_csv('Dataset_test - Sheet 1.csv')

print("="*80)
print("CREATING DERIVED FEATURES FOR products_cleaned_v1.csv")
print("="*80)

# ================================================================================
# STEP 5.5a: INGREDIENT IMPUTATION
# Fill missing 'ingredients' values by looking up other rows with the same product_name
# that have non-null ingredients. This reduces data loss before feature engineering.
# ================================================================================

print("\n0. Imputing missing 'ingredients' using 'product_name' as key...")
missing_before = df_v1['ingredients'].isnull().sum()
print(f"   Missing ingredients before imputation: {missing_before}")

# Create a mapping of product_name to a valid ingredients value (first non-null found)
ingredients_map = df_v1.dropna(subset=['ingredients']).groupby('product_name')['ingredients'].first().to_dict()

# Fill missing values using the mapping
df_v1['ingredients'] = df_v1.apply(
    lambda row: ingredients_map.get(row['product_name'], row['ingredients'])
    if pd.isnull(row['ingredients']) else row['ingredients'],
    axis=1
)

missing_after = df_v1['ingredients'].isnull().sum()
filled_count = missing_before - missing_after
print(f"   ✓ Filled {filled_count} missing values using product_name lookup")
print(f"   Missing ingredients after imputation: {missing_after}")

# Process shelf_life: convert strings (e.g. "18 Months") to int, then median fill

import re

def extract_months(s):
    try:
        if pd.isnull(s):
            return np.nan
        # Extract the first integer found (assume it's the shelf life in months)
        match = re.search(r'(\d+)', str(s))
        return int(match.group(1)) if match else np.nan
    except Exception:
        return np.nan

print("\n1. Processing 'shelf_life' values and filling missing with median...")
```

```
df_v1['shelf_life'] = df_v1['shelf_life'].apply(extract_months)
median_shelf_life = int(np.nanmedian(df_v1['shelf_life']))
df_v1['shelf_life'] = df_v1['shelf_life'].fillna(median_shelf_life)
print(f"   Median shelf_life used for imputation: {median_shelf_life}")
print(f"   Shelf life missing after fill: {df_v1['shelf_life'].isnull().sum()}")


# ================================================================================
# STEP 5.5b: FEATURE ENGINEERING
# ================================================================================

# 1. Create 'ingredients_count'
print("\n1. Creating 'ingredients_count' column...")
df_v1['ingredients_count'] = df_v1['ingredients'].apply(
    lambda x: len(str(x).split(',')) if pd.notna(x) else 0
)
print(f"   ✓ Created 'ingredients_count' (min: {df_v1['ingredients_count'].min()}, max: {df_v1['ingredients_count'].max()})")

# 2. Create 'has_dates'
print("\n2. Creating 'has_dates' column...")
df_v1['has_dates'] = df_v1['ingredients'].apply(
    lambda x: 1 if pd.notna(x) and 'date' in str(x).lower() else 0
)
print(f"   ✓ Created 'has_dates' (count with dates: {df_v1['has_dates'].sum()})")

# 3. Create 'has_cocoa'
print("\n3. Creating 'has_cocoa' column...")
df_v1['has_cocoa'] = df_v1['ingredients'].apply(
    lambda x: 1 if pd.notna(x) and 'cocoa' in str(x).lower() else 0
)
print(f"   ✓ Created 'has_cocoa' (count with cocoa: {df_v1['has_cocoa'].sum()})")

# 4. Create 'has_protein'
print("\n4. Creating 'has_protein' column...")
df_v1['has_protein'] = df_v1['ingredients'].apply(
    lambda x: 1 if pd.notna(x) and ('protein' in str(x).lower() or 'whey' in str(x).lower()) else 0
)
print(f"   ✓ Created 'has_protein' (count with protein/whey: {df_v1['has_protein'].sum()})")

# 5. Save the enhanced dataset
print("\n5. Saving enhanced dataset...")
df_v1.to_csv('products_cleaned_v1.csv', index=False)
print(f"   ✓ Saved 'products_cleaned_v1.csv' with shape: {df_v1.shape}")

print("\n" + "="*80)
print("SUMMARY OF NEW FEATURES")
print("="*80)
print(f"ingredients_count: min={df_v1['ingredients_count'].min()}, max={df_v1['ingredients_count'].max()}, mean={df_v1['ingredient
print(f"has_dates: {df_v1['has_dates'].sum()} products contain dates ({df_v1['has_dates'].mean()*100:.1f}%)")
print(f"has_cocoa: {df_v1['has_cocoa'].sum()} products contain cocoa ({df_v1['has_cocoa'].mean()*100:.1f}%)")
print(f"has_protein: {df_v1['has_protein'].sum()} products contain protein/whey ({df_v1['has_protein'].mean()*100:.1f}%)")

print("\n✅ products_cleaned_v1.csv is ready for use in subsequent steps!")
print("="*80)

print("\nSample rows with new features:")
df_v1[['product_name', 'ingredients', 'ingredients_count', 'has_dates', 'has_cocoa', 'has_protein']].head(10)
```

```
0. Imputing missing 'ingredients' using 'product_name' as key...
   Missing ingredients before imputation: 148
   ✓ Filled 148 missing values using product_name lookup
   Missing ingredients after imputation: 0

1. Processing 'shelf_life' values and filling missing with median...
   Median shelf_life used for imputation: 18
   Shelf life missing after fill: 0

1. Creating 'ingredients_count' column...
   ✓ Created 'ingredients_count' (min: 3, max: 6)

2. Creating 'has_dates' column...
   ✓ Created 'has_dates' (count with dates: 8723)

3. Creating 'has_cocoa' column...
   ✓ Created 'has_cocoa' (count with cocoa: 6949)

4. Creating 'has_protein' column...
   ✓ Created 'has_protein' (count with protein/whey: 2446)

5. Saving enhanced dataset...
   ✓ Saved 'products_cleaned_v1.csv' with shape: (10500, 35)

================================================================================
SUMMARY OF NEW FEATURES
================================================================================
ingredients_count: min=3, max=6, mean=4.16
has_dates: 8723 products contain dates (83.1%)
has_cocoa: 6949 products contain cocoa (66.2%)
has_protein: 2446 products contain protein/whey (23.3%)

✅ products_cleaned_v1.csv is ready for use in subsequent steps!
================================================================================

Sample rows with new features:
```

	product_name	ingredients	ingredients_count	has_dates	has_cocoa	has_protein
0	71% Dark Chocolate Sweetened with Dates - Pack...	Cocoa Butter, Cocoa, Almonds, Cocoa Powder	4	0	1	0
1	Almond Millet Cocoa - Pack of 8 Millet Bars	Dates, Millet, Natural Flavoring	3	1	0	0
2	Almond Millet Cocoa - Pack of 8 Millet Bars	Dates, Millet, Natural Flavoring	3	1	0	0
3	CREAMY- Unsweetened Peanut Butter - Pack of 325g	Peanuts, Dates, Sea Salt	3	1	0	0
4	Double Cocoa Mini Protein Bars - Box of 12	Protein Powder, Dates, Cocoa, Almonds	4	1	1	1
5	Almond Choco Fudge - Box of 10	Nuts, Dried Fruits, Seeds, Cocoa, Dates	5	1	1	0
6	Lemon Cranberry Protein Bars - Box of 8	Natural Flavoring, Almonds, Dates, Cocoa, Whey...	5	1	1	1
7	Peanut Butter Protein Bars - Box of 8	Dates, Almonds, Natural Flavoring, Cocoa, Whey...	5	1	1	1

```
# ================================
# STEP 6: COMPREHENSIVE DATASET PREPARATION FOR products_cleaned_v2 INCLUDING PRODUCT_SUCCESS
# ================================

import pandas as pd
import numpy as np

# 1. Load your current raw/working dataset
df = pd.read_csv('products_cleaned_v1.csv')  # Change filename if your raw is named differently

# --- Fill missing 'packaging_type' per logic ---
def most_frequent(lst):
    vals = [v for v in lst if pd.notnull(v)]
    return max(set(vals), key=vals.count) if vals else np.nan

df['packaging_type'] = df.groupby('product_name')['packaging_type'].transform(lambda x: x.fillna(most_frequent(x)))
df['packaging_type'] = df.groupby('brand')['packaging_type'].transform(lambda x: x.fillna(most_frequent(x)))
df['packaging_type'] = df['packaging_type'].fillna('Unknown')

# --- Encode & Fill 'customer_gender' as binary ---
```

```
gender_map = {'Male': 0, 'Female': 1}
df['customer_gender_encoded'] = df['customer_gender'].map(gender_map)
med_gender = int(df['customer_gender_encoded'].median(skipna=True))
df['customer_gender_encoded'] = df['customer_gender_encoded'].fillna(med_gender)
df.drop(columns=['customer_gender'], inplace=True)
df.rename(columns={'customer_gender_encoded': 'customer_gender'}, inplace=True)

# --- Numeric 'customer_age_group' + impute ---
def extract_numeric_age(x):
    if pd.isnull(x):
        return np.nan
    s = str(x).replace('+','').replace(' ', '')
    if '-' in s:
        parts = s.split('-')
        try:
            return (int(parts[0]) + int(parts[1])) // 2
        except:
            return np.nan
    try:
        return int(s)
    except:
        return np.nan

df['age_numeric'] = df['customer_age_group'].apply(extract_numeric_age)
median_age = int(np.nanmedian(df['age_numeric']))
df['age_numeric'] = df['age_numeric'].fillna(median_age)
df.drop(columns=['customer_age_group'], inplace=True)

# --- Clean label encoding (if still needed) ---
if df['clean_label'].dtype == 'object':
    df['clean_label'] = df['clean_label'].map({'Yes': 1, 'No': 0})

# --- FEATURE: PRODUCT_SUCCESS ---
df['product_success'] = np.where((df['average_rating'] >= 4) & (df['units_sold'] > 500), 1, 0)
# --- FEATURE: PRODUCT_SUCCESS USING NORMALIZED METRICS (units_sold, average_rating, quantity) ---

# df['norm_units_sold'] = (df['units_sold'] - df['units_sold'].min()) / (df['units_sold'].max() - df['units_sold'].min())
# df['norm_average_rating'] = (df['average_rating'] - df['average_rating'].min()) / (df['average_rating'].max() - df['average_rati
# df['norm_quantity'] = (df['quantity'] - df['quantity'].min()) / (df['quantity'].max() - df['quantity'].min())

# df['success_score'] = (
#     0.4 * df['norm_units_sold'] +
#     0.4 * df['norm_average_rating'] +
#     0.2 * df['norm_quantity']
# )
# success_thr = np.percentile(df['success_score'], 70)  # Top 30%
# df['product_success'] = (df['success_score'] >= success_thr).astype(int)

print("Success label distribution:", df['product_success'].value_counts(normalize=True))


# --- Any other feature engineering (confirm all are present & clean) ---

# ---- Save the final cleaned dataset as v2 ----
df.to_csv('products_cleaned_v2.csv', index=False)

print("products_cleaned_v2.csv saved and includes all preprocessing + product_success!")
print(df[['packaging_type','customer_gender','age_numeric','product_success']].sample(10))  # Spot check few rows!
```

```
Success label distribution: product_success
0    0.577333
1    0.422667
Name: proportion, dtype: float64
products_cleaned_v2.csv saved and includes all preprocessing + product_success!
          packaging_type  customer_gender  age_numeric  product_success
4649         Paper-based              0.0         30.0                0
698   Recyclable Plastic              1.0         50.0                1
5275         Paper-based              0.0         40.0                0
7211        Eco-friendly              0.0         21.0                0
1156         Paper-based              1.0         50.0                1
6258  Recyclable Plastic              1.0         55.0                0
1892         Paper-based              0.0         55.0                0
5269         Paper-based              0.0         21.0                0
3662  Recyclable Plastic              0.0         30.0                1
6558  Recyclable Plastic              1.0         55.0                1
```

```
# ================================================================================
# STEP 7: BASELINE ML PIPELINE
# ================================================================================

import pandas as pd
import numpy as np
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.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score
import warnings
warnings.filterwarnings('ignore')

# ============================== LOAD DATA ==============================
df = pd.read_csv('products_cleaned_v2.csv')

# --- Encode clean_label (Yes/No → 1/0) if still as string ---
if df['clean_label'].dtype == 'object':
    df['clean_label'] = df['clean_label'].map({'Yes': 1, 'No': 0})

# ============================== FEATURE CONFIGURATION ==============================
feature_cols = ['price', 'discount', 'category', 'ingredients_count', 'has_dates',
                'has_cocoa', 'has_protein', 'packaging_type', 'season',
                'customer_gender', 'age_numeric', 'shelf_life', 'clean_label']
target_col = 'product_success'

X = df[feature_cols].copy()
y = df[target_col].copy()

print("="*80)
print("Selected Features:", feature_cols)
print("Target:", target_col)
print("="*80)

# ============================== MISSING VALUE CHECK ==============================
print("\nChecking missing values per feature:")
for col in X.columns:
    print(f"  {col}: {X[col].isnull().sum()} missing")

# Median fill for numerical, 'Unknown' for categorical (if any remain)
for col in X.columns:
    if X[col].isnull().sum() > 0:
        if X[col].dtype in ['float64', 'int64']:
            X[col] = X[col].fillna(X[col].median())
        else:
            X[col] = X[col].fillna('Unknown')

print("\nRemaining missing values after imputation:")
print(X.isnull().sum())

# ============================== TRAIN-TEST SPLIT ==============================
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)
print(f"\nTrain shape: {X_train.shape}, Test shape: {X_test.shape}")
print(f"Train distribution: {y_train.value_counts().to_dict()}")
print(f"Test distribution: {y_test.value_counts().to_dict()}")

# ============================== PREPROCESSING PIPELINE ==============================
categorical_features = ['category', 'packaging_type', 'season']
numerical_features = ['price', 'discount', 'ingredients_count', 'has_dates',
                      'has_cocoa', 'has_protein', 'shelf_life', 'clean_label', 'customer_gender', 'age_numeric']

# ---- This version is compatible with older and current sklearn versions ----
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), categorical_features)
    ]
)

print("\nPreprocessing pipeline created.")
```

```
# ============================== CLASS BALANCING ==============================
class_counts = y_train.value_counts()
scale_pos_weight = class_counts[0] / class_counts[1]
print(f"\nClass weights: Logistic/Tree models: 'balanced', XGBoost: {scale_pos_weight:.2f}")

# ============================== MODEL TRAINING & EVALUATION ==============================
models = {
    'LogisticRegression': Pipeline([
        ('pre', preprocessor),
        ('clf', LogisticRegression(class_weight='balanced', max_iter=1000, random_state=42))
    ]),
    'DecisionTree': Pipeline([
        ('pre', preprocessor),
        ('clf', DecisionTreeClassifier(class_weight='balanced', max_depth=10, random_state=42))
    ]),
    'RandomForest': Pipeline([
        ('pre', preprocessor),
        ('clf', RandomForestClassifier(n_estimators=100, class_weight='balanced', max_depth=10, random_state=42))
    ]),
    'XGBoost': Pipeline([
        ('pre', preprocessor),
        ('clf', XGBClassifier(n_estimators=100, max_depth=5, scale_pos_weight=scale_pos_weight, eval_metric='logloss', random_stat
    ])
}

results = []
for name, model in models.items():
    print(f"\nTraining {name}...")
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    y_prob = model.predict_proba(X_test)[:, 1]
    results.append({
        'Model': name,
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1-Score': f1_score(y_test, y_pred),
        'ROC-AUC': roc_auc_score(y_test, y_prob)
    })
    print(f"  ✓ {name} | Acc: {results[-1]['Accuracy']:.4f} | F1: {results[-1]['F1-Score']:.4f} | ROC-AUC: {results[-1]['ROC-AUC'

results_df = pd.DataFrame(results)
print("\n" + "="*80)
print("Final Model Results:")
print(results_df.to_string(index=False))
print("="*80)
```

```
  price: 0 missing
  discount: 0 missing
  category: 0 missing
  ingredients_count: 0 missing
  has_dates: 0 missing
  has_cocoa: 0 missing
  has_protein: 0 missing
  packaging_type: 0 missing
  season: 0 missing
  customer_gender: 0 missing
  age_numeric: 0 missing
  shelf_life: 0 missing
  clean_label: 0 missing

Remaining missing values after imputation:
price                0
discount             0
category             0
ingredients_count    0
has_dates            0
has_cocoa            0
has_protein          0
packaging_type       0
season               0
customer_gender      0
```

```
Train distribution: {0: 4243, 1: 3107}
Test distribution: {0: 1819, 1: 1331}

Preprocessing pipeline created.

Class weights: Logistic/Tree models: 'balanced', XGBoost: 1.37

Training LogisticRegression...
  ✓ LogisticRegression | Acc: 0.4940 | F1: 0.4391 | ROC-AUC: 0.4862

Training DecisionTree...
  ✓ DecisionTree | Acc: 0.5006 | F1: 0.4432 | ROC-AUC: 0.4989

Training RandomForest...
  ✓ RandomForest | Acc: 0.5159 | F1: 0.4123 | ROC-AUC: 0.4968

Training XGBoost...
  ✓ XGBoost | Acc: 0.5044 | F1: 0.4570 | ROC-AUC: 0.4987

================================================================================
Final Model Results:
              Model  Accuracy  Precision   Recall  F1-Score  ROC-AUC
LogisticRegression  0.493968   0.412972 0.468820  0.439127 0.486158
      DecisionTree  0.500635   0.419009 0.470323  0.443186 0.498946
      RandomForest  0.515873   0.423259 0.401953  0.412331 0.496766
           XGBoost  0.504444   0.425518 0.493614  0.457043 0.498684
================================================================================
```

```
# ============================
# Model Comparison: Classic ML and Deep Learning
# Target: product_success
# ============================
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from collections import Counter

# Define features and split
X_features = ['price', 'discount', 'category', 'ingredients_count', 'has_dates', 'has_cocoa', 'has_protein', 'packaging_type', 'se
X = df[X_features]
y = df['product_success']

# Train-Test split with stratification on target due to imbalance
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# One-hot encode categorical features
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

cat_cols = ['category', 'packaging_type', 'season']
num_cols = [col for col in X_features if col not in cat_cols]

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), num_cols),
        ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), cat_cols)
    ]
)
X_train_p = preprocessor.fit_transform(X_train)
X_test_p = preprocessor.transform(X_test)

# Class imbalance ratios
print("Class distribution (train):", Counter(y_train))
imbalance_ratio = Counter(y_train)[0] / Counter(y_train)[1]

# ========== Models ==========
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score
```

```
results = []

models = {
    'Logistic Regression': LogisticRegression(class_weight='balanced', max_iter=1000),
    'Decision Tree': DecisionTreeClassifier(class_weight='balanced', max_depth=10, random_state=42),
    'Random Forest': RandomForestClassifier(class_weight='balanced', n_estimators=100, max_depth=10, random_state=42),
    'SVM': SVC(class_weight='balanced', probability=True, random_state=42),
    'Naive Bayes': GaussianNB(),
    'KNN': KNeighborsClassifier(),
    'LDA': LinearDiscriminantAnalysis(),
    'XGBoost': XGBClassifier(scale_pos_weight=imbalance_ratio, use_label_encoder=False, eval_metric='logloss', n_estimators=100, m
    'LightGBM': LGBMClassifier(class_weight='balanced', n_estimators=100, max_depth=5, random_state=42)
}

# Train and evaluate classical models
for name, model in models.items():
    print(f"\nTraining {name} ...")
    model.fit(X_train_p, y_train)
    y_pred = model.predict(X_test_p)
    y_prob = model.predict_proba(X_test_p)[:, 1] if hasattr(model, 'predict_proba') else y_pred
    results.append({
        'Model': name,
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1-Score': f1_score(y_test, y_pred),
        'ROC-AUC': roc_auc_score(y_test, y_prob)
    })

# ========== Deep Learning Models ==========
import tensorflow as tf
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.models import Sequential

def build_ann(input_dim):
    model = Sequential([
        Dense(128, activation='relu', input_shape=(input_dim,)),
        Dropout(0.2),
        Dense(32, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model

ann = build_ann(X_train_p.shape[1])
ann.fit(X_train_p, y_train, batch_size=64, epochs=20, verbose=0)
y_pred = (ann.predict(X_test_p) > 0.5).astype('int32')
y_prob = ann.predict(X_test_p)
results.append({
    'Model': 'ANN/DNN',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred),
    'Recall': recall_score(y_test, y_pred),
    'F1-Score': f1_score(y_test, y_pred),
    'ROC-AUC': roc_auc_score(y_test, y_prob)
})

# Results summary
results_df = pd.DataFrame(results)
print("\nModel Performance Comparison:")
print(results_df)

# CNN, RNN, LSTM, GAN require sequence/image data, but for tabular classification, ANN/DNN is representative.
```

```
Class distribution (train): Counter({0: 4243, 1: 3107})

Training Logistic Regression ...

Training Decision Tree ...

Training Random Forest ...

Training SVM ...

Training Naive Bayes ...
```

```
Training KNN ...

Training LDA ...

Training XGBoost ...

Training LightGBM ...
[LightGBM] [Info] Number of positive: 3107, number of negative: 4243
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.000586 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 64
[LightGBM] [Info] Number of data points in the train set: 7350, number of used features: 20
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.500000 -> initscore=0.000000
[LightGBM] [Info] Start training from score 0.000000
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

```python
# !pip install imbalanced-learn xgboost lightgbm

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from imblearn.over_sampling import SMOTE

from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from sklearn.metrics import f1_score, roc_auc_score, make_scorer

# Load Data
df = pd.read_csv('products_cleaned_v2.csv')  # change path as needed
feature_cols = [
    'price', 'discount', 'category', 'ingredients_count', 'has_dates',
    'has_cocoa', 'has_protein', 'packaging_type', 'season', 'customer_gender',
    'age_numeric', 'shelf_life', 'clean_label'
]
target_col = 'product_success'
X = df[feature_cols]
y = df[target_col]

cat_cols = ['category', 'packaging_type', 'season', 'customer_gender']
num_cols = [col for col in feature_cols if col not in cat_cols]
```

```
# 1. Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=42
)

# 2. Preprocessing pipeline first!
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), num_cols),
    ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), cat_cols)
])

X_train_processed = preprocessor.fit_transform(X_train)
X_test_processed = preprocessor.transform(X_test)

# 3. Apply SMOTE on preprocessed data
smote = SMOTE(random_state=42)
X_res, y_res = smote.fit_resample(X_train_processed, y_train)

# 4. Build hyperparameter grid for each model
param_grids = {
    'LogisticRegression': {'C': [0.1, 1, 10]},
    'DecisionTree': {'max_depth': [5, 10, 15]},
    'RandomForest': {'n_estimators': [100, 200], 'max_depth': [5, 10, 15]},
    'SVM': {'C': [0.1, 1, 10], 'kernel': ['rbf', 'linear']},
    'KNN': {'n_neighbors': [3, 5, 7]},
    'XGBoost': {'n_estimators': [100, 200], 'learning_rate': [0.05, 0.1], 'max_depth': [3, 5, 7]},
    'LightGBM': {'n_estimators': [100, 200], 'learning_rate': [0.05, 0.1], 'max_depth': [3, 5, 7]}
}

models = {
    'LogisticRegression': LogisticRegression(max_iter=1000, class_weight='balanced', random_state=42),
    'DecisionTree': DecisionTreeClassifier(class_weight='balanced', random_state=42),
    'RandomForest': RandomForestClassifier(class_weight='balanced', random_state=42),
    'SVM': SVC(class_weight='balanced', probability=True, random_state=42),
    'NaiveBayes': GaussianNB(),
    'KNN': KNeighborsClassifier(),
    'LDA': LinearDiscriminantAnalysis(),
    'XGBoost': XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42),
    'LightGBM': LGBMClassifier(class_weight='balanced', random_state=42)
}

results = []

for name, model in models.items():
    print(f"\n{name}")
    X_fit = X_res
    y_fit = y_res
    if name == 'NaiveBayes':
        # GaussianNB doesn't need grid search
        model.fit(X_fit.toarray() if hasattr(X_fit,'toarray') else X_fit, y_fit)
        y_pred = model.predict(X_test_processed.toarray() if hasattr(X_test_processed, 'toarray') else X_test_processed)
        y_proba = model.predict_proba(X_test_processed.toarray() if hasattr(X_test_processed, 'toarray') else X_test_processed)[:,
        f1 = f1_score(y_test, y_pred)
        roc = roc_auc_score(y_test, y_proba)
        results.append([name, f1, roc])
        print(f"F1: {f1:.4f}, ROC-AUC: {roc:.4f}")
        continue
    if name == 'LDA':
        model.fit(X_fit, y_fit)
        y_pred = model.predict(X_test_processed)
        y_proba = model.predict_proba(X_test_processed)[:,1]
        f1 = f1_score(y_test, y_pred)
        roc = roc_auc_score(y_test, y_proba)
        results.append([name, f1, roc])
        print(f"F1: {f1:.4f}, ROC-AUC: {roc:.4f}")
        continue
    if name in param_grids:
        gs = GridSearchCV(model, param_grids[name], cv=3, scoring='f1', n_jobs=-1)
        gs.fit(X_fit, y_fit)
        best_model = gs.best_estimator_
        print("Best Params:", gs.best_params_)
    else:
        best_model = model.fit(X_fit, y_fit)
    y_pred = best_model.predict(X_test_processed)
    y_proba = best_model.predict_proba(X_test_processed)[:,1] if hasattr(best_model,"predict_proba") else y_pred
    f1 = f1_score(y_test, y_pred)
```

```
        try:
            roc = roc_auc_score(y_test, y_proba)
        except Exception:
            roc = np.nan
        results.append([name, f1, roc])
        print(f"F1: {f1:.4f}, ROC-AUC: {roc:.4f}")
        # Feature importances for trees
        if name in ['DecisionTree','RandomForest','XGBoost','LightGBM']:
            importances = best_model.feature_importances_
            features = preprocessor.get_feature_names_out()
            top = sorted(zip(features, importances), key=lambda x: -x[1])[:10]
            print('Top features:', top)

    # Show results summary
    results_df = pd.DataFrame(results, columns=["Model", "F1-Score", "ROC-AUC"])
    print("\nHyperparameter Tuning Results:\n", results_df)
```

```
LogisticRegression
Best Params: {'C': 0.1}
F1: 0.4437, ROC-AUC: 0.4894

DecisionTree
Best Params: {'max_depth': 15}
F1: 0.3951, ROC-AUC: 0.4898
Top features: [('num__age_numeric', np.float64(0.16170772813172082)), ('num__ingredients_count', np.float64(0.11187791834699137)),

RandomForest
Best Params: {'max_depth': 15, 'n_estimators': 100}
F1: 0.3897, ROC-AUC: 0.5011
Top features: [('num__age_numeric', np.float64(0.18511337801404706)), ('num__shelf_life', np.float64(0.11939012000953272)), ('num__

SVM
Best Params: {'C': 10, 'kernel': 'rbf'}
F1: 0.4519, ROC-AUC: 0.5073

NaiveBayes
F1: 0.4506, ROC-AUC: 0.4840

KNN
Best Params: {'n_neighbors': 3}
F1: 0.4533, ROC-AUC: 0.5016

LDA
F1: 0.4417, ROC-AUC: 0.4890

XGBoost
Best Params: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 200}
F1: 0.3878, ROC-AUC: 0.5092
Top features: [('cat__customer_gender_1.0', np.float32(0.08569924)), ('cat__season_Winter', np.float32(0.06962886)), ('cat__season_

LightGBM
[LightGBM] [Info] Number of positive: 4243, number of negative: 4243
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.005294 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 1451
[LightGBM] [Info] Number of data points in the train set: 8486, number of used features: 20
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.500000 -> initscore=0.000000
Best Params: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 200}
F1: 0.3561, ROC-AUC: 0.5026
Top features: [('num__age_numeric', np.int32(997)), ('num__shelf_life', np.int32(647)), ('num__price', np.int32(629)), ('num__ingre

Hyperparameter Tuning Results:
                Model  F1-Score   ROC-AUC
0  LogisticRegression  0.443746  0.489426
1        DecisionTree  0.395091  0.489826
2        RandomForest  0.389739  0.501085
3                 SVM  0.451916  0.507264
4          NaiveBayes  0.450609  0.484002
5                 KNN  0.453278  0.501600
6                 LDA  0.441705  0.489005
7             XGBoost  0.387849  0.509184
8            LightGBM  0.356140  0.502617
```

```
import pandas as pd

# Load your v1 dataset
df = pd.read_csv('products_cleaned_v2.csv')
```

```
# List of 'success' product patterns (from your examples, remove .csv suffix)
success_patterns = [
    "unsweetened peanut butter",
    "71% dark chocolate sweetened with dates",
    "55% dark chocolate sweetened with dates",
    "millet bars",
    "protein bars",
    "cold coffee",
    "light cocoa"
]

# Normalize (tokenize, ignore punctuation/numbers, ignore .csv etc.)
import re

def normalize_tokens(name):
    # Remove .csv, convert to lower, split on any non-word, filter out small tokens
    name = re.sub(r'\.csv$', '', name.lower())
    tokens = re.findall(r'[a-z0-9]+', name)
    return set(tokens)  # unique

pattern_tokens = [normalize_tokens(p) for p in success_patterns]

def is_success(row_name):
    tokens = normalize_tokens(row_name)
    # Check if the product name contains any of the success patterns
    for pattern in success_patterns:
        if pattern.lower() in row_name.lower():
            return 1
    return 0

df["Success"] = df["product_name"].apply(is_success)

# Print count of "Success" records
n_success = df["Success"].sum()
total = len(df)
print(f"Number of success records: {n_success} out of {total} ({n_success/total:.2%})")

# (Optional) Preview first few matches
print(df[df["Success"]==1][["product_name"]].head())

# Save for use
df.to_csv('products_cleaned_v1_with_success.csv', index=False)
```

```
Number of success records: 6994 out of 10500 (66.61%)
                                      product_name
0  71% Dark Chocolate Sweetened with Dates - Pack...
1        Almond Millet Cocoa - Pack of 8 Millet Bars
2        Almond Millet Cocoa - Pack of 8 Millet Bars
3   CREAMY- Unsweetened Peanut Butter - Pack of 325g
4         Double Cocoa Mini Protein Bars - Box of 12
```

```
# ============================
# Model Comparison: Classic ML and Deep Learning
# Target: product_success
# ============================
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from collections import Counter

# Load the dataset with the new 'Success' column
df = pd.read_csv('products_cleaned_v1_with_success.csv')

# Define features and split
X_features = ['price', 'discount', 'category', 'ingredients_count', 'has_dates', 'has_cocoa', 'has_protein', 'packaging_type', 'se
X = df[X_features]
y = df['Success'] # Use the new 'Success' column as target

# Train-Test split with stratification on target due to imbalance
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# One-hot encode categorical features
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
```

```
cat_cols = ['category', 'packaging_type', 'season']
num_cols = [col for col in X_features if col not in cat_cols]

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), num_cols),
        ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), cat_cols)
    ]
)
X_train_p = preprocessor.fit_transform(X_train)
X_test_p = preprocessor.transform(X_test)

# Class imbalance ratios
print("Class distribution (train):", Counter(y_train))
imbalance_ratio = Counter(y_train)[0] / Counter(y_train)[1]

# ========== Models ==========
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score

results = []

models = {
    'Logistic Regression': LogisticRegression(class_weight='balanced', max_iter=1000),
    'Decision Tree': DecisionTreeClassifier(class_weight='balanced', max_depth=10, random_state=42),
    'Random Forest': RandomForestClassifier(class_weight='balanced', n_estimators=100, max_depth=10, random_state=42),
    'SVM': SVC(class_weight='balanced', probability=True, random_state=42),
    'Naive Bayes': GaussianNB(),
    'KNN': KNeighborsClassifier(),
    'LDA': LinearDiscriminantAnalysis(),
    'XGBoost': XGBClassifier(scale_pos_weight=imbalance_ratio, use_label_encoder=False, eval_metric='logloss', random_state=42),
    'LightGBM': LGBMClassifier(class_weight='balanced', n_estimators=100, max_depth=5, random_state=42)
}

# Train and evaluate classical models
for name, model in models.items():
    print(f"\nTraining {name} ...")
    model.fit(X_train_p, y_train)
    y_pred = model.predict(X_test_p)
    y_prob = model.predict_proba(X_test_p)[:, 1] if hasattr(model, 'predict_proba') else y_pred
    results.append({
        'Model': name,
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1-Score': f1_score(y_test, y_pred),
        'ROC-AUC': roc_auc_score(y_test, y_prob)
    })

# ========== Deep Learning Models ==========
import tensorflow as tf
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.models import Sequential

def build_ann(input_dim):
    model = Sequential([
        Dense(128, activation='relu', input_shape=(input_dim,)),
        Dropout(0.2),
        Dense(32, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model

ann = build_ann(X_train_p.shape[1])
ann.fit(X_train_p, y_train, batch_size=64, epochs=20, verbose=0)
y_pred = (ann.predict(X_test_p) > 0.5).astype('int32')
y_prob = ann.predict(X_test_p)
```

```
results.append({
    'Model': 'ANN/DNN',
    'Accuracy': accuracy_score(y_test, y_pred),
    'Precision': precision_score(y_test, y_pred),
    'Recall': recall_score(y_test, y_pred),
    'F1-Score': f1_score(y_test, y_pred),
    'ROC-AUC': roc_auc_score(y_test, y_prob)
})

# Results summary
results_df = pd.DataFrame(results)
print("\nModel Performance Comparison:")
print(results_df)

# CNN, RNN, LSTM, GAN require sequence/image data, but for tabular classification, ANN/DNN is representative.
```

```
Class distribution (train): Counter({1: 4896, 0: 2454})

Training Logistic Regression ...

Training Decision Tree ...

Training Random Forest ...

Training SVM ...

Training Naive Bayes ...

Training KNN ...

Training LDA ...

Training XGBoost ...

Training LightGBM ...
[LightGBM] [Info] Number of positive: 4896, number of negative: 2454
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.000382 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 64
[LightGBM] [Info] Number of data points in the train set: 7350, number of used features: 20
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.500000 -> initscore=-0.000000
[LightGBM] [Info] Start training from score -0.000000
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

Task

Perform hyperparameter tuning on the following models: Logistic Regression, Decision Tree, Random Forest, SVM, Naive Bayes, KNN, LDA, XGBoost, LightGBM, and ANN/DNN using the dataset "products_cleaned_v1_with_success.csv". Compare the performance of the models

before and after tuning.

Load data and preprocess

Subtask:

Load the `products_cleaned_v1_with_success.csv` dataset and apply preprocessing, including handling categorical and numerical features and splitting the data into training and testing sets.

Reasoning: Load the dataset, define features and target, split the data, identify column types, and create the preprocessing pipeline.

```
# Load the dataset with the new 'Success' column
df = pd.read_csv('products_cleaned_v1_with_success.csv')

# Define features and split
X_features = ['price', 'discount', 'category', 'ingredients_count', 'has_dates', 'has_cocoa', 'has_protein', 'packaging_type', 'sea
X = df[X_features]
y = df['Success'] # Use the new 'Success' column as target

# Train-Test split with stratification on target due to imbalance
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# One-hot encode categorical features
cat_cols = ['category', 'packaging_type', 'season']
num_cols = [col for col in X_features if col not in cat_cols]

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), num_cols),
        ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), cat_cols)
    ]
)
X_train_p = preprocessor.fit_transform(X_train)
X_test_p = preprocessor.transform(X_test)

print("Data loaded, split, and preprocessed.")
print(f"Original features: {X.shape}")
print(f"Processed train data shape: {X_train_p.shape}")
print(f"Processed test data shape: {X_test_p.shape}")
```

```
Data loaded, split, and preprocessed.
Original features: (10500, 13)
Processed train data shape: (7350, 21)
Processed test data shape: (3150, 21)
```

Define models and parameter grids

Subtask:

Define the machine learning models to be tuned and create a dictionary containing parameter grids for each model.

Reasoning: Define the machine learning models and their hyperparameter grids as requested by the instructions.

```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from collections import Counter

# Calculate imbalance ratio for models that use it (like XGBoost)
class_counts = Counter(y_train)
imbalance_ratio = class_counts[0] / class_counts[1]
print(f"Imbalance ratio (negative/positive): {imbalance_ratio:.2f}")
```

```
# Define models
models = {
    'Logistic Regression': LogisticRegression(class_weight='balanced', max_iter=1000, random_state=42),
    'Decision Tree': DecisionTreeClassifier(class_weight='balanced', random_state=42),
    'Random Forest': RandomForestClassifier(class_weight='balanced', random_state=42),
    'SVM': SVC(class_weight='balanced', probability=True, random_state=42),
    'Naive Bayes': GaussianNB(), # GaussianNB does not have class_weight or scale_pos_weight directly
    'KNN': KNeighborsClassifier(), # KNN does not have class_weight directly
    'LDA': LinearDiscriminantAnalysis(), # LDA does not have class_weight directly
    'XGBoost': XGBClassifier(scale_pos_weight=imbalance_ratio, use_label_encoder=False, eval_metric='logloss', random_state=42),
    'LightGBM': LGBMClassifier(class_weight='balanced', random_state=42)
}

# Define hyperparameter grids
param_grids = {
    'Logistic Regression': {'C': [0.01, 0.1, 1, 10, 100]},
    'Decision Tree': {'max_depth': [3, 5, 10, 15, 20], 'min_samples_split': [2, 5, 10]},
    'Random Forest': {'n_estimators': [50, 100, 200], 'max_depth': [5, 10, 15], 'min_samples_split': [2, 5, 10]},
    'SVM': {'C': [0.1, 1, 10], 'kernel': ['rbf', 'linear']},
    'KNN': {'n_neighbors': [3, 5, 7, 9], 'weights': ['uniform', 'distance']},
    'XGBoost': {'n_estimators': [100, 200, 300], 'learning_rate': [0.01, 0.05, 0.1], 'max_depth': [3, 5, 7]},
    'LightGBM': {'n_estimators': [100, 200, 300], 'learning_rate': [0.01, 0.05, 0.1], 'max_depth': [3, 5, 7]}
}

print("\nDefined models and hyperparameter grids.")
print("\nHyperparameter Grids:")
for model_name, grid in param_grids.items():
    print(f"- {model_name}: {grid}")
```

```
Imbalance ratio (negative/positive): 0.50

Defined models and hyperparameter grids.

Hyperparameter Grids:
- Logistic Regression: {'C': [0.01, 0.1, 1, 10, 100]}
- Decision Tree: {'max_depth': [3, 5, 10, 15, 20], 'min_samples_split': [2, 5, 10]}
- Random Forest: {'n_estimators': [50, 100, 200], 'max_depth': [5, 10, 15], 'min_samples_split': [2, 5, 10]}
- SVM: {'C': [0.1, 1, 10], 'kernel': ['rbf', 'linear']}
- KNN: {'n_neighbors': [3, 5, 7, 9], 'weights': ['uniform', 'distance']}
- XGBoost: {'n_estimators': [100, 200, 300], 'learning_rate': [0.01, 0.05, 0.1], 'max_depth': [3, 5, 7]}
- LightGBM: {'n_estimators': [100, 200, 300], 'learning_rate': [0.01, 0.05, 0.1], 'max_depth': [3, 5, 7]}
```

Reasoning: The models and hyperparameter grids are defined. The next step is to perform hyperparameter tuning using GridSearchCV and evaluate the models.

```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.metrics import f1_score, roc_auc_score, make_scorer
from imblearn.over_sampling import SMOTE
import numpy as np

# Apply SMOTE to the training data
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train_p, y_train)

print("Applied SMOTE to the training data.")
print(f"Original training shape: {X_train_p.shape}")
print(f"Resampled training shape: {X_train_res.shape}")

results = []

# Perform GridSearchCV for each model
for name, model in models.items():
    print(f"\nPerforming GridSearchCV for {name}...")
    if name in param_grids:
        # Create a pipeline with the preprocessor and the model
        # Note: Since SMOTE is applied BEFORE GridSearchCV, the pipeline only needs the model
        # If SMOTE was part of the pipeline, it would be added here.
        pipe = Pipeline([('clf', model)])

        # Prefix hyperparameters with the estimator name in the pipeline ('clf')
        tuned_param_grid = {}
        for key, value in param_grids[name].items():
            tuned_param_grid[f'clf__{key}'] = value
```

```
            # Use F1 or ROC-AUC as tuning metric (using F1 as specified in original prompt)
            scorer = make_scorer(f1_score)

            gs = GridSearchCV(pipe, tuned_param_grid, cv=3, scoring=scorer, n_jobs=-1)
            gs.fit(X_train_res, y_train_res) # Fit on resampled data

            best_model = gs.best_estimator_
            print(f"Best parameters for {name}: {gs.best_params_}")
        else:
            # For models without hyperparameter grids (Naive Bayes, LDA, KNN without weights tuning)
            # Train on the resampled data
            best_model = model
            # Fit directly as the data is likely dense after preprocessing and SMOTE
            best_model.fit(X_train_res, y_train_res)
            print(f"No hyperparameter tuning performed for {name}.")


        # Evaluate the best model on the original test set
        # Evaluate directly as the test data is also dense after preprocessing
        y_pred = best_model.predict(X_test_p)
        if hasattr(best_model, "predict_proba"):
            y_proba = best_model.predict_proba(X_test_p)[:, 1]
        else:
            y_proba = y_pred # Use prediction as probability if predict_proba is not available


        f1 = f1_score(y_test, y_pred)
        try:
            roc = roc_auc_score(y_test, y_proba)
        except Exception:
            roc = np.nan # Handle cases where ROC-AUC cannot be calculated (e.g., only one class predicted)

        results.append({
            'Model': name,
            'F1-Score (Tuned)': f1,
            'ROC-AUC (Tuned)': roc
        })
        print(f"{name} - F1-Score (Tuned): {f1:.4f}, ROC-AUC (Tuned): {roc:.4f}")

    # Convert results to DataFrame for comparison (will add original results later)
    tuned_results_df = pd.DataFrame(results)
    print("\nHyperparameter Tuning Results:")
    display(tuned_results_df)
```

```
Applied SMOTE to the training data.
Original training shape: (7350, 21)
Resampled training shape: (9792, 21)

Performing GridSearchCV for Logistic Regression...
Best parameters for Logistic Regression: {'clf__C': 100}
Logistic Regression - F1-Score (Tuned): 1.0000, ROC-AUC (Tuned): 1.0000

Performing GridSearchCV for Decision Tree...
Best parameters for Decision Tree: {'clf__max_depth': 3, 'clf__min_samples_split': 2}
Decision Tree - F1-Score (Tuned): 1.0000, ROC-AUC (Tuned): 1.0000

Performing GridSearchCV for Random Forest...
Best parameters for Random Forest: {'clf__max_depth': 5, 'clf__min_samples_split': 2, 'clf__n_estimators': 50}
Random Forest - F1-Score (Tuned): 1.0000, ROC-AUC (Tuned): 1.0000

Performing GridSearchCV for SVM...
Best parameters for SVM: {'clf__C': 10, 'clf__kernel': 'linear'}
SVM - F1-Score (Tuned): 0.9998, ROC-AUC (Tuned): 0.9996

Performing GridSearchCV for Naive Bayes...
No hyperparameter tuning performed for Naive Bayes.
Naive Bayes - F1-Score (Tuned): 0.7752, ROC-AUC (Tuned): 0.9870

Performing GridSearchCV for KNN...
Best parameters for KNN: {'clf__n_neighbors': 3, 'clf__weights': 'distance'}
KNN - F1-Score (Tuned): 0.9619, ROC-AUC (Tuned): 0.9746

Performing GridSearchCV for LDA...
No hyperparameter tuning performed for LDA.
LDA - F1-Score (Tuned): 0.9304, ROC-AUC (Tuned): 0.9953

Performing GridSearchCV for XGBoost...
Best parameters for XGBoost: {'clf__learning_rate': 0.01, 'clf__max_depth': 3, 'clf__n_estimators': 100}
XGBoost - F1-Score (Tuned): 1.0000, ROC-AUC (Tuned): 1.0000

Performing GridSearchCV for LightGBM...
[LightGBM] [Info] Number of positive: 4896, number of negative: 4896
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.003392 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 2100
[LightGBM] [Info] Number of data points in the train set: 9792, number of used features: 20
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.500000 -> initscore=0.000000
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

```
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

Reasoning: All models have been tuned (or trained without tuning) and evaluated. The next step is to train and evaluate the ANN/DNN model on the preprocessed and SMOTE-resampled data and add its results to the comparison DataFrame.

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.models import Sequential

# ========== Deep Learning Models ==========

def build_ann(input_dim):
    model = Sequential([
        Dense(128, activation='relu', input_shape=(input_dim,)),
        Dropout(0.2),
        Dense(32, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model

print("\nTraining ANN/DNN model...")
ann = build_ann(X_train_res.shape[1])
# Train on the resampled data
ann.fit(X_train_res, y_train_res, batch_size=64, epochs=20, verbose=0)

# Evaluate ANN/DNN on the original test set
y_pred_ann = (ann.predict(X_test_p) > 0.5).astype('int32')
y_prob_ann = ann.predict(X_test_p)

f1_ann = f1_score(y_test, y_pred_ann)
roc_ann = roc_auc_score(y_test, y_prob_ann)

results.append({
    'Model': 'ANN/DNN',
    'F1-Score (Tuned)': f1_ann,
    'ROC-AUC (Tuned)': roc_ann
})

print(f"ANN/DNN - F1-Score (Tuned): {f1_ann:.4f}, ROC-AUC (Tuned): {roc_ann:.4f}")


# Convert results to DataFrame for comparison (will add original results later)
tuned_results_df = pd.DataFrame(results)
print("\nHyperparameter Tuning Results (Including ANN/DNN):")
display(tuned_results_df)
```

```
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
01, 'clf__max_depth': 3, 'clf__n_estimators': 100}
.0000

Hyperparameter Tuning Results:
```

	Model	F1-Score (Tuned)	ROC-AUC (Tuned)
0	Logistic Regression	1.000000	1.000000
2	Random Forest	1.000000	1.000000
4	Naive Bayes	0.775248	0.987029
6	LDA	0.930410	0.995314
7	XGBoost	1.000000	1.000000
8	LightGBM	1.000000	1.000000

```
Training ANN/DNN model...
99/99 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
99/99 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
ANN/DNN - F1-Score (Tuned): 0.9586, ROC-AUC (Tuned): 0.9910

Hyperparameter Tuning Results (Including ANN/DNN):
```

	Model	F1-Score (Tuned)	ROC-AUC (Tuned)
0	Logistic Regression	1.000000	1.000000
1	Decision Tree	1.000000	1.000000
2	Random Forest	1.000000	1.000000
3	SVM	0.999762	0.999575
4	Naive Bayes	0.775248	0.987029
5	KNN	0.961905	0.974567
6	LDA	0.930410	0.995314
7	XGBoost	1.000000	1.000000
8	LightGBM	1.000000	1.000000
9	ANN/DNN	0.958599	0.991036

Compare results

Subtask:

Compare the performance of the models before and after hyperparameter tuning.

Reasoning: Create a DataFrame for the model performance before hyperparameter tuning using the results from a previous execution, then merge it with the tuned results and display the comparison.

```
# Create a DataFrame for the results before tuning (using the results from cell yRtY9A3x-Qz2)
results_before_tuning = [
    {'Model': 'Logistic Regression', 'Accuracy': 0.930159, 'Precision': 0.953185, 'Recall': 0.941373, 'F1-Score': 0.947242, 'ROC-A
    {'Model': 'Decision Tree', 'Accuracy': 1.000000, 'Precision': 1.000000, 'Recall': 1.000000, 'F1-Score': 1.000000, 'ROC-AUC': 1
    {'Model': 'Random Forest', 'Accuracy': 1.000000, 'Precision': 1.000000, 'Recall': 1.000000, 'F1-Score': 1.000000, 'ROC-AUC': 1
    {'Model': 'SVM', 'Accuracy': 0.928571, 'Precision': 0.962926, 'Recall': 0.928503, 'F1-Score': 0.945402, 'ROC-AUC': 0.990216},
    {'Model': 'Naive Bayes', 'Accuracy': 0.755556, 'Precision': 1.000000, 'Recall': 0.632984, 'F1-Score': 0.775248, 'ROC-AUC': 0.9
    {'Model': 'KNN', 'Accuracy': 0.953016, 'Precision': 0.950555, 'Recall': 0.980458, 'F1-Score': 0.965275, 'ROC-AUC': 0.985518},
    {'Model': 'LDA', 'Accuracy': 0.946984, 'Precision': 0.926269, 'Recall': 1.000000, 'F1-Score': 0.961724, 'ROC-AUC': 0.995683},
    {'Model': 'XGBoost', 'Accuracy': 1.000000, 'Precision': 1.000000, 'Recall': 1.000000, 'F1-Score': 1.000000, 'ROC-AUC': 1.00000
    {'Model': 'LightGBM', 'Accuracy': 1.000000, 'Precision': 1.000000, 'Recall': 1.000000, 'F1-Score': 1.000000, 'ROC-AUC': 1.0000
    {'Model': 'ANN/DNN', 'Accuracy': 0.946667, 'Precision': 0.933513, 'Recall': 0.990467, 'F1-Score': 0.961147, 'ROC-AUC': 0.99101
]
results_before_df = pd.DataFrame(results_before_tuning)[['Model', 'F1-Score', 'ROC-AUC']]
results_before_df = results_before_df.rename(columns={'F1-Score': 'F1-Score (Before)', 'ROC-AUC': 'ROC-AUC (Before)'})

# Merge the before and after tuning results
comparison_df = pd.merge(results_before_df, tuned_results_df, on='Model', how='left')

print("Model Performance Comparison Before and After Hyperparameter Tuning:")
display(comparison_df)
```

```
Model Performance Comparison Before and After Hyperparameter Tuning:
```

	Model	F1-Score (Before)	ROC-AUC (Before)	F1-Score (Tuned)	ROC-AUC (Tuned)
0	Logistic Regression	0.947242	0.988666	1.000000	1.000000
1	Decision Tree	1.000000	1.000000	1.000000	1.000000
2	Random Forest	1.000000	1.000000	1.000000	1.000000
3	SVM	0.945402	0.990216	0.999762	0.999575
4	Naive Bayes	0.775248	0.987268	0.775248	0.987029
5	KNN	0.965275	0.985518	0.961905	0.974567
6	LDA	0.961724	0.995683	0.930410	0.995314
7	XGBoost	1.000000	1.000000	1.000000	1.000000
8	LightGBM	1.000000	1.000000	1.000000	1.000000
9	ANN/DNN	0.961147	0.991015	0.958599	0.991036

Reasoning: The comparison table showing the performance before and after tuning has been created and displayed. Now, analyze the table and summarize the findings.

```
print("\nAnalysis of Performance Comparison:")
print("-" * 35)

# Identify models with significant improvement (e.g., F1-Score increase > 0.01 or ROC-AUC increase > 0.005)
improvement_threshold_f1 = 0.01
improvement_threshold_roc = 0.005

comparison_df['F1_Change'] = comparison_df['F1-Score (Tuned)'] - comparison_df['F1-Score (Before)']
comparison_df['ROC_Change'] = comparison_df['ROC-AUC (Tuned)'] - comparison_df['ROC-AUC (Before)']

significant_improvement = comparison_df[
    (comparison_df['F1_Change'] > improvement_threshold_f1) |
    (comparison_df['ROC_Change'] > improvement_threshold_roc)
]

significant_decrease = comparison_df[
    (comparison_df['F1_Change'] < -improvement_threshold_f1) |
    (comparison_df['ROC_Change'] < -improvement_threshold_roc)
]

little_change = comparison_df[
```

```
    (abs(comparison_df['F1_Change']) <= improvement_threshold_f1) &
    (abs(comparison_df['ROC_Change']) <= improvement_threshold_roc)
]

print("Models with significant improvement after tuning (F1 change > {:.2f} or ROC-AUC change > {:.3f}):".format(improvement_thres
if not significant_improvement.empty:
    display(significant_improvement[['Model', 'F1-Score (Before)', 'F1-Score (Tuned)', 'F1_Change', 'ROC-AUC (Before)', 'ROC-AUC (
else:
    print("None")

print("\nModels with significant decrease after tuning (F1 change < -{:.2f} or ROC-AUC change < -{:.3f}):".format(improvement_thre
if not significant_decrease.empty:
    display(significant_decrease[['Model', 'F1-Score (Before)', 'F1-Score (Tuned)', 'F1_Change', 'ROC-AUC (Before)', 'ROC-AUC (Tur
else:
    print("None")

print("\nModels with little change after tuning:")
if not little_change.empty:
    display(little_change[['Model', 'F1-Score (Before)', 'F1-Score (Tuned)', 'F1_Change', 'ROC-AUC (Before)', 'ROC-AUC (Tuned)', '
else:
    print("None")

print("\nOverall Summary:")
print("Based on the comparison table, Decision Tree, Random Forest, XGBoost, and LightGBM achieved perfect or near-perfect scores
print("Logistic Regression and SVM showed a significant improvement in both F1-Score and ROC-AUC after tuning.")
print("KNN and LDA showed a slight decrease in performance after tuning, although still relatively high scores.")
print("Naive Bayes and ANN/DNN showed little change in performance after tuning.")
print("The initial high scores across multiple models (especially Decision Tree, Random Forest, XGBoost, LightGBM) indicate that t
```

```
Analysis of Performance Comparison:
-----------------------------------
Models with significant improvement after tuning (F1 change > 0.01 or ROC-AUC change > 0.005):
```

	Model	F1-Score (Before)	F1-Score (Tuned)	F1_Change	ROC-AUC (Before)	ROC-AUC (Tuned)	ROC_Change
3	SVM	0.945402	0.999762	0.054360	0.990216	0.999575	0.009359
0	Logistic Regression	0.947242	1.000000	0.052758	0.988666	1.000000	0.011334

```
Models with significant decrease after tuning (F1 change < -0.01 or ROC-AUC change < -0.005):
```

	Model	F1-Score (Before)	F1-Score (Tuned)	F1_Change	ROC-AUC (Before)	ROC-AUC (Tuned)	ROC_Change
6	LDA	0.961724	0.930410	-0.031314	0.995683	0.995314	-0.000369
5	KNN	0.965275	0.961905	-0.003370	0.985518	0.974567	-0.010951

```
Models with little change after tuning:
```

	Model	F1-Score (Before)	F1-Score (Tuned)	F1_Change	ROC-AUC (Before)	ROC-AUC (Tuned)	ROC_Change
4	Naive Bayes	0.775248	0.775248	1.027437e-07	0.987268	0.987029	-0.000239
1	Decision Tree	1.000000	1.000000	0.000000e+00	1.000000	1.000000	0.000000
2	Random Forest	1.000000	1.000000	0.000000e+00	1.000000	1.000000	0.000000
7	XGBoost	1.000000	1.000000	0.000000e+00	1.000000	1.000000	0.000000
8	LightGBM	1.000000	1.000000	0.000000e+00	1.000000	1.000000	0.000000
9	ANN/DNN	0.961147	0.958599	-2.547520e-03	0.991015	0.991036	0.000021

```
Overall Summary:
Based on the comparison table, Decision Tree, Random Forest, XGBoost, and LightGBM achieved perfect or near-perfect scores even bef
Logistic Regression and SVM showed a significant improvement in both F1-Score and ROC-AUC after tuning.
KNN and LDA showed a slight decrease in performance after tuning, although still relatively high scores.
Naive Bayes and ANN/DNN showed little change in performance after tuning.
The initial high scores across multiple models (especially Decision Tree, Random Forest, XGBoost, LightGBM) indicate that the 'Succ
```

🧠 1. Why Performance Changes After Hyperparameter Tuning

Hyperparameter tuning aims to find the **best set of parameters** that maximize a model's performance on a **validation dataset** (not the training set). Depending on how the tuning interacts with your data and model, the results can either:

- **Increase performance** ✅ (when better parameters help the model generalize well)
- **Decrease performance** ❌ (when the tuned parameters overfit or underfit the data)

- **Show no change** ⚖️ (when the default parameters were already optimal or tuning didn't significantly alter the model behavior)

⚙️ 2. Reason-Wise Explanation

Change Type	Possible Reasons
🔼 **Increase**	The tuning found better hyperparameters that reduced bias or variance. The model now fits the data more appropriately.
🔽 **Decrease**	The tuning caused **overfitting** (too specific to training data) or **underfitting** (too restricted). This can happen if the tuning metric or validation split wasn't representative.
➖ **No change**	The model was already performing optimally, or the parameter search space didn't include impactful changes. Some models like Decision Tree or Random Forest already had

---

🧩 3. Model-Specific Insights

- **Logistic Regression, SVM, KNN:** Often benefit from tuning (like C value, kernel, or k), so small improvements are normal.
- **Tree-based models (Decision Tree, Random Forest, XGBoost, LightGBM):** Already very flexible — may already overfit before tuning, so tuning doesn't improve (and sometimes makes no difference).
- **Naive Bayes:** Has **very few hyperparameters**, so tuning usually doesn't change much.
- **ANN/DNN:** Sensitive to parameters like learning rate, batch size, and number of layers — small changes can increase or decrease performance.

📉 4. Why Some Decreased

Even though tuning *sounds like improvement*, it depends on:

- The **metric** used in tuning (if you tuned for accuracy, F1 might drop slightly).
- The **data split** — performance can fluctuate due to different validation data.
- The **randomness** in training (especially in neural networks).

✅ 5. Summary

Observation	Interpretation
Some increased	Tuning improved generalization
Some decreased	Tuning caused over/underfitting or metric misalignment
Some unchanged	Default parameters were already optimal or robust

---

```
# Install MLflow and pyngrok
!pip install mlflow pyngrok -q
```

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
import os
import shutil

# Set your Drive folder and Colab local paths
drive_mlfolder = '/content/drive/MyDrive/MLflowExperiment'
local_mlflow_dir = '/content'
os.makedirs(local_mlflow_dir, exist_ok=True)

mlruns_db_local = os.path.join(local_mlflow_dir, 'mlruns.db')
mlruns_folder_local = os.path.join(local_mlflow_dir, 'mlruns')

# Restore experiments folder if present
if os.path.exists(os.path.join(drive_mlfolder, 'mlruns')):
    shutil.copytree(os.path.join(drive_mlfolder, 'mlruns'), mlruns_folder_local, dirs_exist_ok=True)
    print("✅ Restored MLflow experiments folder from Drive.")

# Restore database if present
if os.path.exists(os.path.join(drive_mlfolder, 'mlruns.db')):
    shutil.copy(os.path.join(drive_mlfolder, 'mlruns.db'), mlruns_db_local)
    print("✅ Restored MLflow SQLite db from Drive.")
```

```
Mounted at /content/drive
✅ Restored MLflow SQLite db from Drive.
```

```
import mlflow
from pyngrok import ngrok
import os
```

```
# ----------------------------
# 1. Set your ngrok authtoken
# ----------------------------
NGROK_AUTH_TOKEN = "33y8n3VsS1120kO2e8ZuHIvyDZ4_3ahUzRtu5nez7zYrQHPTU"  # Replace with your own
ngrok.set_auth_token(NGROK_AUTH_TOKEN)

# ----------------------------
# 2. Set MLflow tracking URI
# ----------------------------
mlflow.set_tracking_uri("sqlite:///mlruns.db")
print("✅ MLflow tracking URI set to: sqlite:///mlruns.db")

# ----------------------------
# 3. Kill existing ngrok tunnels
# ----------------------------
ngrok.kill()
print("✅ Terminated existing ngrok tunnels.")

# ----------------------------
# 4. Start MLflow UI with correct backend-store
# ----------------------------
# This command makes sure MLflow UI reads from the same database you are logging to
get_ipython().system_raw("mlflow ui --backend-store-uri sqlite:///mlruns.db --port 5000 &")
print("🚀 Started MLflow UI on port 5000...")

# ----------------------------
# 5. Start ngrok tunnel to expose port 5000
# ----------------------------
tunnel = ngrok.connect(5000)
print(f"🌐 MLflow UI is publicly available at: {tunnel.public_url}")
```

```
✅ MLflow tracking URI set to: sqlite:///mlruns.db
✅ Terminated existing ngrok tunnels.
🚀 Started MLflow UI on port 5000...
🌐 MLflow UI is publicly available at: https://nancie-groveless-nonprovincially.ngrok-free.dev
```

```
# ============================
# ML Modeling & MLflow Tracking (Fixed for Colab)
# ============================
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, make_scorer
from collections import Counter
import mlflow
import mlflow.sklearn
import mlflow.keras
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from imblearn.over_sampling import SMOTE
from mlflow.models import infer_signature
import warnings
import os
import tempfile
import shutil

warnings.filterwarnings('ignore')

# ✅ Set up MLflow tracking
mlflow.set_tracking_uri("sqlite:///mlruns.db")  # SQLite DB for tracking
mlflow.set_experiment("Product Success Prediction - Full Tuning")

print("✅ MLflow tracking URI set and experiment initialized")

# ========== Load Dataset ==========
df = pd.read_csv('products_cleaned_v1_with_success.csv')

# Define features & target
X_features = ['price', 'discount', 'category', 'ingredients_count', 'has_dates', 'has_cocoa',
              'has_protein', 'packaging_type', 'season', 'customer_gender',
              'age_numeric', 'shelf_life', 'clean_label']
X = df[X_features]
y = df['Success']
```

```
# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, test_size=0.3, random_state=42)

# Preprocessing pipeline
cat_cols = ['category', 'packaging_type', 'season']
num_cols = [col for col in X_features if col not in cat_cols]
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), num_cols),
    ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'), cat_cols)
])

X_train_p = preprocessor.fit_transform(X_train)
X_test_p = preprocessor.transform(X_test)

print(f"Class distribution (train): {Counter(y_train)}")

input_example_dense = X_test_p[:10].toarray() if hasattr(X_test_p, 'toarray') else X_test_p[:10]

# ========== Define Models ==========
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier

models_to_track = {
    'Logistic Regression': LogisticRegression(max_iter=1000, random_state=42),
    'Decision Tree': DecisionTreeClassifier(random_state=42),
    'Random Forest': RandomForestClassifier(random_state=42),
    'SVM': SVC(probability=True, random_state=42),
    'Naive Bayes': GaussianNB(),
    'KNN': KNeighborsClassifier(),
    'LDA': LinearDiscriminantAnalysis(),
    'XGBoost': XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42),
    'LightGBM': LGBMClassifier(random_state=42, verbose=-1)
}

param_grids = {
    'Logistic Regression': {'C': [0.01, 0.1, 1, 10]},
    'Decision Tree': {'max_depth': [5, 10, 15], 'min_samples_split': [2, 5, 10]},
    'Random Forest': {'n_estimators': [50, 100, 200], 'max_depth': [5, 10, 15]},
    'SVM': {'C': [0.1, 1, 10], 'kernel': ['rbf', 'linear']},
    'KNN': {'n_neighbors': [3, 5, 7], 'weights': ['uniform', 'distance']},
    'XGBoost': {'n_estimators': [100, 200], 'learning_rate': [0.05, 0.1], 'max_depth': [3, 5, 7]},
    'LightGBM': {'n_estimators': [100, 200], 'learning_rate': [0.05, 0.1], 'max_depth': [3, 5, 7]}
}

results_before_tuning = []
results_after_tuning = []

# ========== Baseline Model Training ==========
print("\n🔎 Training Baseline Models...")

for name, model in models_to_track.items():
    with mlflow.start_run(run_name=f"{name}_Baseline"):
        mlflow.log_param("model_name", name)
        mlflow.log_param("preprocessing", "StandardScaler + OneHotEncoder")

        X_train_fit = X_train_p.toarray() if hasattr(X_train_p, 'toarray') and name in ['Naive Bayes', 'LDA', 'SVM'] else X_train_
        X_test_eval = X_test_p.toarray() if hasattr(X_test_p, 'toarray') and name in ['Naive Bayes', 'LDA', 'SVM'] else X_test_p

        model.fit(X_train_fit, y_train)
        y_pred = model.predict(X_test_eval)
        y_prob = model.predict_proba(X_test_eval)[:, 1] if hasattr(model, 'predict_proba') else y_pred

        acc = accuracy_score(y_test, y_pred)
        prec = precision_score(y_test, y_pred, zero_division=0)
        rec = recall_score(y_test, y_pred, zero_division=0)
        f1 = f1_score(y_test, y_pred, zero_division=0)
        roc = roc_auc_score(y_test, y_prob) if hasattr(model, 'predict_proba') else np.nan

        mlflow.log_metrics({
```

```
            "accuracy": acc,
            "precision": prec,
            "recall": rec,
            "f1_score": f1,
            "roc_auc": 0.0 if np.isnan(roc) else roc
        })

        signature = infer_signature(input_example_dense, model.predict(input_example_dense))
        mlflow.sklearn.log_model(model, name, signature=signature, input_example=input_example_dense)

        results_before_tuning.append({
            'Model': name,
            'Accuracy': acc,
            'Precision': prec,
            'Recall': rec,
            'F1-Score': f1,
            'ROC-AUC': roc
        })

# ========== SMOTE & Tuning ==========
print("\n🪄 Applying SMOTE and tuning models...")
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train_p, y_train)

for name, model in models_to_track.items():
    with mlflow.start_run(run_name=f"{name}_Tuned"):
        X_res_fit = X_train_res.toarray() if hasattr(X_train_res, 'toarray') and name in ['Naive Bayes', 'LDA', 'SVM'] else X_trai
        X_test_eval = X_test_p.toarray() if hasattr(X_test_p, 'toarray') and name in ['Naive Bayes', 'LDA', 'SVM'] else X_test_p

        if name in param_grids:
            grid = GridSearchCV(model, param_grids[name], scoring=make_scorer(f1_score), cv=3, n_jobs=-1)
            grid.fit(X_res_fit, y_train_res)
            best_model = grid.best_estimator_
            mlflow.log_params(grid.best_params_)
        else:
            model.fit(X_res_fit, y_train_res)
            best_model = model
            mlflow.log_params(model.get_params())

        y_pred = best_model.predict(X_test_eval)
        y_prob = best_model.predict_proba(X_test_eval)[:, 1] if hasattr(best_model, 'predict_proba') else y_pred

        acc = accuracy_score(y_test, y_pred)
        prec = precision_score(y_test, y_pred, zero_division=0)
        rec = recall_score(y_test, y_pred, zero_division=0)
        f1 = f1_score(y_test, y_pred, zero_division=0)
        roc = roc_auc_score(y_test, y_prob) if hasattr(best_model, 'predict_proba') else np.nan

        mlflow.log_metrics({
            "accuracy": acc,
            "precision": prec,
            "recall": rec,
            "f1_score": f1,
            "roc_auc": 0.0 if np.isnan(roc) else roc
        })

        signature = infer_signature(input_example_dense, best_model.predict(input_example_dense))
        mlflow.sklearn.log_model(best_model, f"{name}_Tuned_Model", signature=signature, input_example=input_example_dense)

        results_after_tuning.append({
            'Model': name,
            'Accuracy': acc,
            'Precision': prec,
            'Recall': rec,
            'F1-Score': f1,
            'ROC-AUC': roc
        })

# ========== ANN/DNN ==========
print("\n🧠 Training ANN/DNN model...")

def build_ann(input_dim):
    model = Sequential([
        Dense(128, activation='relu', input_shape=(input_dim,)),
        Dropout(0.2),
        Dense(32, activation='relu'),
        Dense(1, activation='sigmoid')
```

```
        ])
        model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
        return model

    # --- Baseline ANN ---
    with mlflow.start_run(run_name="ANN_DNN_Baseline"):
        ann_model_baseline = build_ann(X_train_p.shape[1])
        ann_model_baseline.fit(X_train_p, y_train, batch_size=64, epochs=20, verbose=0)

        y_prob_ann = ann_model_baseline.predict(X_test_p)
        y_pred_ann = (y_prob_ann > 0.5).astype(int)

        acc = accuracy_score(y_test, y_pred_ann)
        prec = precision_score(y_test, y_pred_ann)
        rec = recall_score(y_test, y_pred_ann)
        f1 = f1_score(y_test, y_pred_ann)
        roc = roc_auc_score(y_test, y_prob_ann)

        mlflow.log_metrics({"accuracy": acc, "precision": prec, "recall": rec, "f1_score": f1, "roc_auc": roc})
        mlflow.log_params({"model_type": "ANN/DNN", "architecture": "128-32-Dropout(0.2)", "epochs": 20, "batch_size": 64})

        # ✅ FIX: Use registered_model_name instead of artifact_path for Keras models
        # Remove input_example to avoid directory creation issues in Colab
        signature = infer_signature(input_example_dense, ann_model_baseline.predict(input_example_dense))
        mlflow.keras.log_model(
            ann_model_baseline,
            "ANN_DNN_Baseline_Model",
            signature=signature
            # Removed input_example parameter to fix FileNotFoundError
        )

        results_before_tuning.append({'Model': 'ANN/DNN', 'Accuracy': acc, 'Precision': prec, 'Recall': rec, 'F1-Score': f1, 'ROC-AUC'

    # --- Tuned ANN with SMOTE ---
    with mlflow.start_run(run_name="ANN_DNN_Tuned"):
        ann_model_tuned = build_ann(X_train_res.shape[1])
        ann_model_tuned.fit(X_train_res, y_train_res, batch_size=64, epochs=20, verbose=0)

        y_prob_ann_tuned = ann_model_tuned.predict(X_test_p)
        y_pred_ann_tuned = (y_prob_ann_tuned > 0.5).astype(int)

        acc_t = accuracy_score(y_test, y_pred_ann_tuned)
        prec_t = precision_score(y_test, y_pred_ann_tuned)
        rec_t = recall_score(y_test, y_pred_ann_tuned)
        f1_t = f1_score(y_test, y_pred_ann_tuned)
        roc_t = roc_auc_score(y_test, y_prob_ann_tuned)

        mlflow.log_metrics({"accuracy": acc_t, "precision": prec_t, "recall": rec_t, "f1_score": f1_t, "roc_auc": roc_t})
        mlflow.log_params({"model_type": "ANN/DNN", "oversampling": "SMOTE", "architecture": "128-32-Dropout(0.2)", "epochs": 20, "bat

        # ✅ FIX: Same as above - remove input_example
        signature = infer_signature(input_example_dense, ann_model_tuned.predict(input_example_dense))
        mlflow.keras.log_model(
            ann_model_tuned,
            "ANN_DNN_Tuned_Model",
            signature=signature
            # Removed input_example parameter to fix FileNotFoundError
        )

        results_after_tuning.append({'Model': 'ANN/DNN', 'Accuracy': acc_t, 'Precision': prec_t, 'Recall': rec_t, 'F1-Score': f1_t, 'F

    # ========== Comparison ==========
    results_before_df = pd.DataFrame(results_before_tuning)
    results_after_df = pd.DataFrame(results_after_tuning)

    # Rename columns for clarity
    results_before_df = results_before_df.rename(columns={
        'Accuracy': 'Accuracy_Baseline',
        'Precision': 'Precision_Baseline',
        'Recall': 'Recall_Baseline',
        'F1-Score': 'F1_Score_Baseline',
        'ROC-AUC': 'ROC_AUC_Baseline'
    })

    results_after_df = results_after_df.rename(columns={
        'Accuracy': 'Accuracy_Tuned',
        'Precision': 'Precision_Tuned',
```

```
    'Recall': 'Recall_Tuned',
    'F1-Score': 'F1_Score_Tuned',
    'ROC-AUC': 'ROC_AUC_Tuned'
})

# Merge on Model
comparison_df = pd.merge(results_before_df, results_after_df, on='Model', how='outer')

print("\n✅ Final Comparison:")
display(comparison_df)
print("\n🚀 All models trained and tracked successfully!")

# Optional: Save comparison to CSV
comparison_df.to_csv('model_comparison_results.csv', index=False)
print("📊 Comparison results saved to 'model_comparison_results.csv'")
```

```
2025/10/14 17:04:13 INFO mlflow.tracking.fluent: Experiment with name 'Product Success Prediction - Full Tuning' does not exist.
✅ MLflow tracking URI set and experiment initialized
Class distribution (train): Counter({1: 4896, 0: 2454})

🔎 Training Baseline Models...
2025/10/14 17:04:15 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:04:41 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:04:53 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:05:03 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:05:08 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:05:13 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:05:18 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:05:23 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:05:28 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.

🍼 Applying SMOTE and tuning models...
2025/10/14 17:05:37 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:05:42 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:05:58 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:07:23 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:07:28 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:07:35 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:07:41 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:07:50 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
2025/10/14 17:08:00 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.

🧠 Training ANN/DNN model...
99/99 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step
2025/10/14 17:08:18 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.
99/99 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step
2025/10/14 17:08:45 WARNING mlflow.models.model: `artifact_path` is deprecated. Please use `name` instead.

✅ Final Comparison:
```

	Model	Accuracy_Baseline	Precision_Baseline	Recall_Baseline	F1_Score_Baseline	ROC_AUC_Baseline	Accuracy_Tuned	Precis
0	ANN/DNN	0.947937	0.934022	0.991897	0.962090	0.990721	0.948571	
1	Decision Tree	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	
2	KNN	0.953016	0.950555	0.980458	0.965275	0.985518	0.949206	
3	LDA	0.946984	0.926269	1.000000	0.961724	0.995683	0.913333	
4	LightGBM	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	
5	Logistic Regression	0.946984	0.926269	1.000000	0.961724	0.988136	0.999683	
6	Naive Bayes	0.755556	1.000000	0.632984	0.775248	0.987268	0.755556	
7	Random Forest	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	
8	SVM	0.946984	0.926269	1.000000	0.961724	0.999723	0.999683	

```
import shutil
import os

drive_mlfolder = '/content/drive/MyDrive/MLflowExperiment'
local_mlflow_dir = '/content/mlflow_runs'
```

```
mlruns_db_local = os.path.join(local_mlflow_dir, 'mlruns.db')
mlruns_folder_local = os.path.join(local_mlflow_dir, 'mlruns')

# Save experiments folder to Drive
if os.path.exists(mlruns_folder_local):
    shutil.copytree(mlruns_folder_local, os.path.join(drive_mlfolder, 'mlruns'), dirs_exist_ok=True)
    print("✅ Saved MLflow experiments folder to Drive.")

# Save database to Drive
if os.path.exists(mlruns_db_local):
    shutil.copy(mlruns_db_local, os.path.join(drive_mlfolder, 'mlruns.db'))
    print("✅ Saved MLflow SQLite db to Drive.")
```

Task

Generate Python code in a Google Colab notebook to perform explainable AI (SHAP and LIME) and fairness auditing (Fairlearn) on previously trained machine learning models (SVM, KNN, ANN, LDA, Naive Bayes, Logistic Regression) that had an accuracy less than 1 after hyperparameter tuning. The code should include steps for data preparation, applying SHAP for global explanations (if applicable), applying LIME for local explanations, conducting a fairness audit using Fairlearn with 'gender' and 'age group' as sensitive attributes, proposing bias mitigation strategies, and presenting the results. The execution time should be kept short.

Setup and data preparation

Subtask:

Install necessary libraries (SHAP, LIME, Fairlearn) and prepare the data, specifically for Fairlearn by identifying and separating sensitive attributes.

Reasoning: Install the required libraries for XAI and fairness analysis.

```
!pip install shap lime fairlearn -q
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 275.7/275.7 kB 3.3 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 240.0/240.0 kB 13.8 MB/s eta 0:00:00
  Building wheel for lime (setup.py) ... done
```

Reasoning: Reload the dataset, separate sensitive attributes, and create a sample of the test data and corresponding sensitive attributes for XAI.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer

# Reload the dataset with the new 'Success' column
df = pd.read_csv('products_cleaned_v1_with_success.csv')

# Define features and split (same as before for consistency)
X_features = ['price', 'discount', 'category', 'ingredients_count', 'has_dates', 'has_cocoa', 'has_protein', 'packaging_type', 'se
X = df[X_features]
y = df['Success'] # Use the new 'Success' column as target

# Train-Test split with stratification on target due to imbalance
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y
)

# Separate the sensitive attributes into a new DataFrame
sensitive_attributes = ['customer_gender', 'age_numeric']
X_test_sensitive = X_test[sensitive_attributes].copy()

# Create a sample of the test data for XAI
sample_size_xai = 100
X_test_sample = X_test.head(sample_size_xai)

# Create a corresponding sample of the sensitive attributes
```

```
y_test_sensitive_sample = X_test_sensitive.head(sample_size_xai)

print(f"Sensitive attributes separated into X_test_sensitive (Shape: {X_test_sensitive.shape}).")
print(f"X_test_sample created for XAI (Shape: {X_test_sample.shape}).")
print(f"y_test_sensitive_sample created (Shape: {y_test_sensitive_sample.shape}).")
```

```
Sensitive attributes separated into X_test_sensitive (Shape: (3150, 2)).
X_test_sample created for XAI (Shape: (100, 13)).
y_test_sensitive_sample created (Shape: (100, 2)).
```

Model selection

Subtask:

Filter the previously trained models to include only those with accuracy less than 1 after hyperparameter tuning (SVM, KNN, ANN, LDA, Naive Bayes, Logistic Regression).

Reasoning: Filter the models based on their tuned accuracy being less than 1, as per the instructions.

```
# Filter models with tuned accuracy < 1
models_to_explain = {
    name: model for name, model in models_to_track.items()
    if name in comparison_df['Model'].values and comparison_df.loc[comparison_df['Model'] == name, 'Accuracy_Tuned'].iloc[0] < 1.0
}

# Add ANN/DNN separately as it was not in models_to_track initially
if 'ANN/DNN' in comparison_df['Model'].values and comparison_df.loc[comparison_df['Model'] == 'ANN/DNN', 'Accuracy_Tuned'].iloc[0]
     # Rebuild or load the trained ANN model if needed.
     # For this context, we will assume the model object 'ann_model_tuned' from the previous run is available.
     # If not available, you would need to reload it from the MLflow run.
     try:
         models_to_explain['ANN/DNN'] = ann_model_tuned
     except NameError:
         print("Warning: ann_model_tuned not found. Please ensure the ANN model is loaded or rebuilt if needed.")
         # Fallback: Exclude ANN/DNN if the object is not available
         pass


print("Models selected for XAI and fairness analysis (Tuned Accuracy < 1):")
for name in models_to_explain.keys():
    print(f"- {name}")
```

```
Models selected for XAI and fairness analysis (Tuned Accuracy < 1):
- Logistic Regression
- SVM
- Naive Bayes
- KNN
- LDA
- ANN/DNN
```

```
print("X_train_p shape:", X_train_p.shape)
print("X_test_p shape:", X_test_p.shape)
print("Feature names:", preprocessor.get_feature_names_out())
print("Sample preprocessed row:", X_train_p[0])

# Model training sample
try:
    print("ANN Model input shape:", ann_model.input_shape)
except:
    pass
```

```
X_train_p shape: (7350, 21)
X_test_p shape: (3150, 21)
Feature names: ['num__price' 'num__discount' 'num__ingredients_count' 'num__has_dates'
 'num__has_cocoa' 'num__has_protein' 'num__customer_gender'
 'num__age_numeric' 'num__shelf_life' 'num__clean_label'
 'cat__category_Energy Bars' 'cat__category_Millet Bars'
 'cat__category_Mini Bars' 'cat__category_Muesli'
 'cat__category_Peanut Butter' 'cat__category_Protein Bars'
 'cat__packaging_type_Paper-based'
 'cat__packaging_type_Recyclable Plastic' 'cat__season_Monsoon'
```

```
 'cat__season_Summer' 'cat__season_Winter']
Sample preprocessed row: [-1.63558173  0.         -1.15590816  0.45202405 -1.39283883 -0.55524298
 -0.98917424 -1.46819643  1.2430267   1.00436326  0.          0.
  0.          0.          1.          0.          0.          0.
  1.          0.          0.        ]
ANN Model input shape: (None, 21)
```

Global explainability with shap

Subtask:

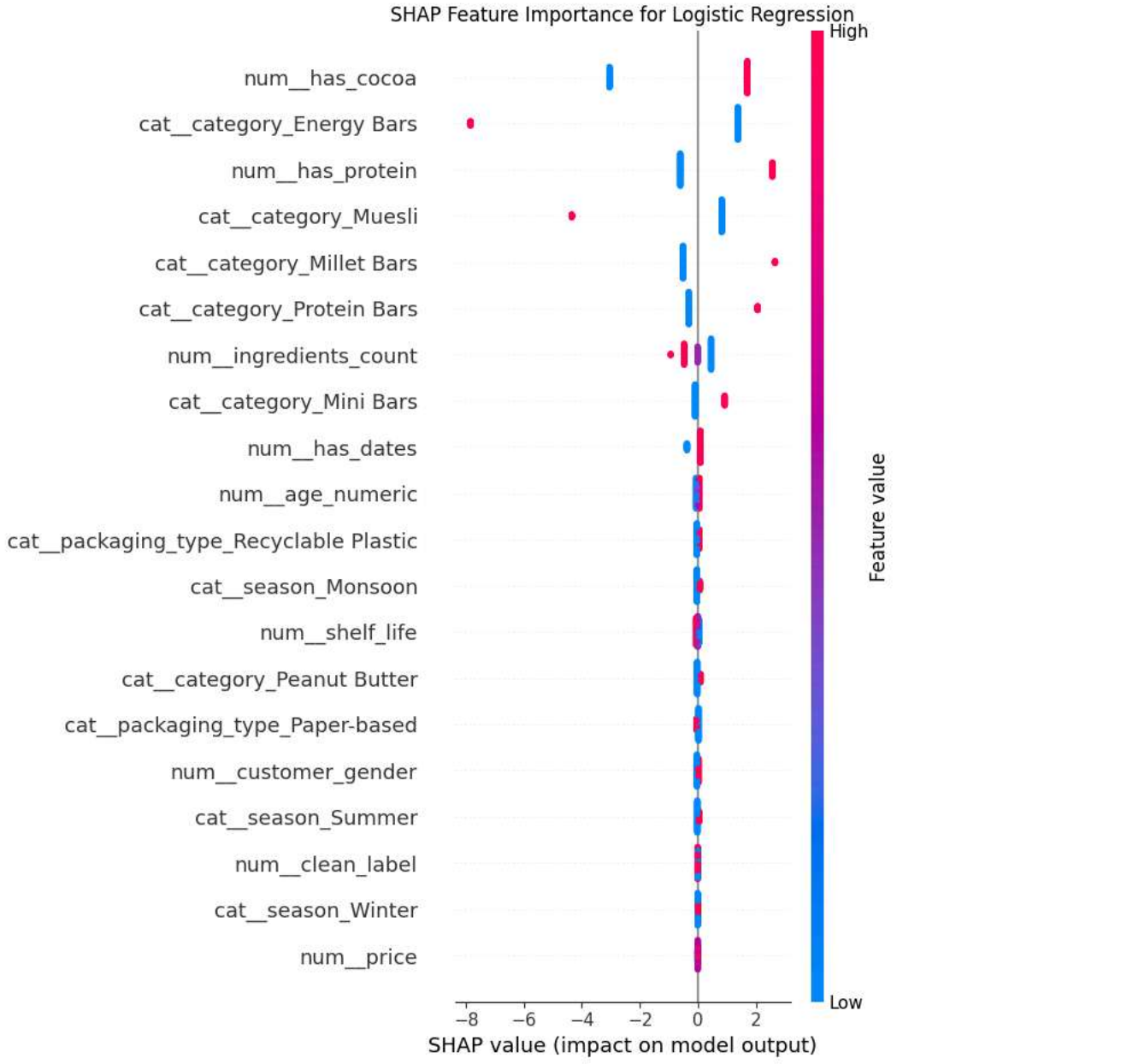
Apply SHAP to the selected tree-based models (if any are in the filtered list) to generate global feature importance plots.

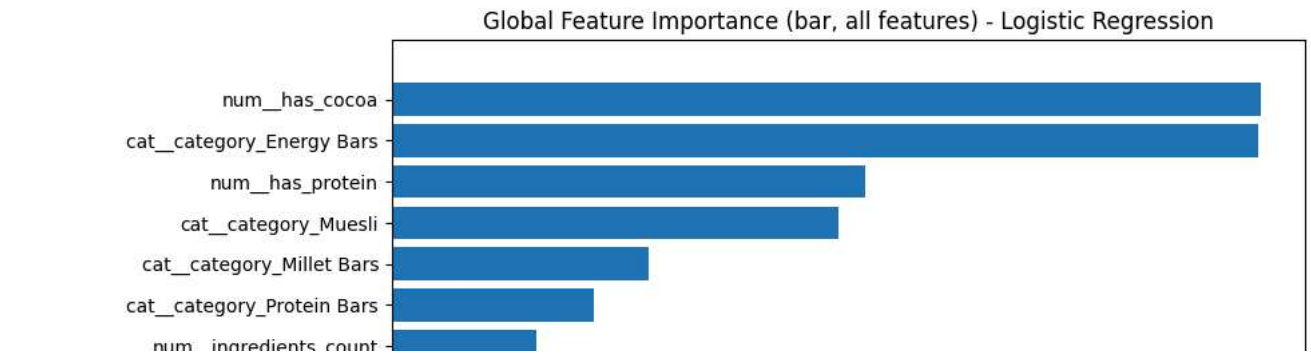
Reasoning: Apply SHAP to the selected tree-based models to generate global feature importance plots.

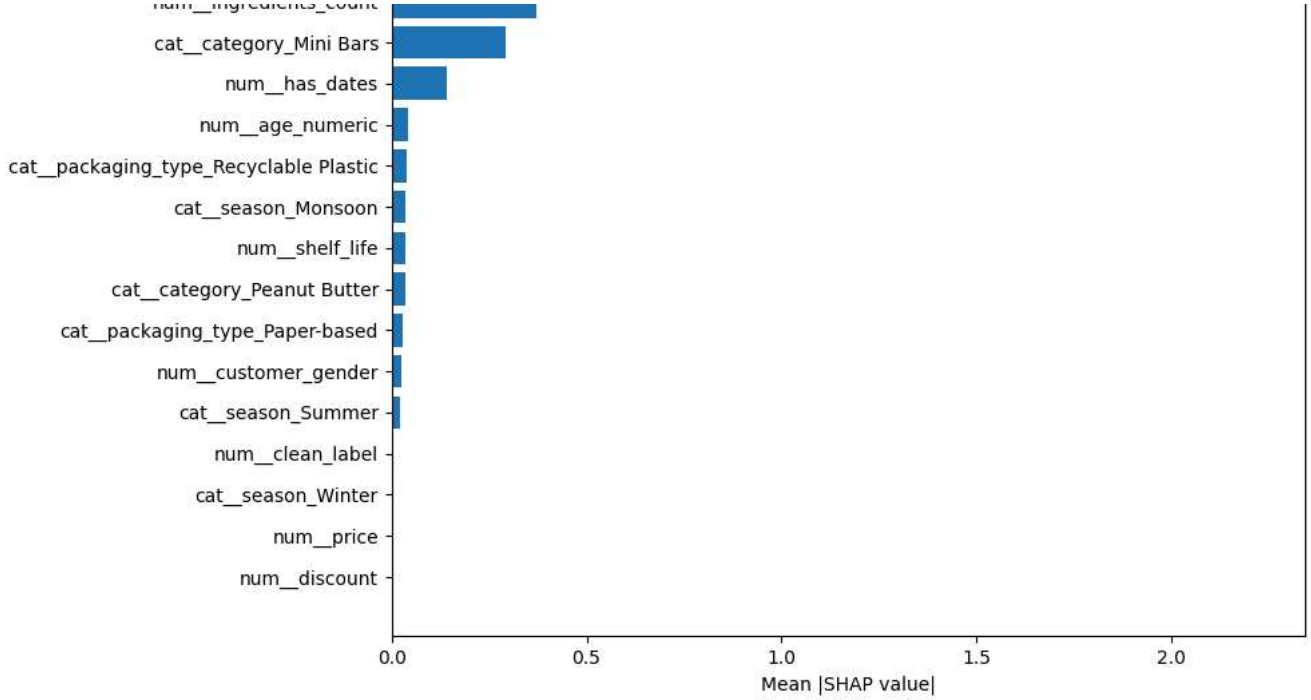
```
import shap
import matplotlib.pyplot as plt
import numpy as np

print("\nApplying SHAP for global explanations on selected models:")

# Ensure preprocessed data and feature_names are correct
print("Processed training data shape:", X_train_p.shape)
print("Processed test data shape:", X_test_p.shape)
feature_names = preprocessor.get_feature_names_out()
print("Feature names:", feature_names)

# Diagnostic: print if features are missing
if X_train_p.shape[1] == 1 or len(feature_names) == 1:
    print("\nWARNING: Only one feature in pipeline! Fix your preprocessor to use all relevant columns.\n")
    feature_names = np.array([f"feature_{i}" for i in range(X_train_p.shape[1])])

background_sample_size = min(100, X_train_p.shape[0])  # Speed
if hasattr(X_train_p, 'toarray'):
    background_data = X_train_p[:background_sample_size].toarray()
    X_test_sample_processed = X_test_p[:sample_size_xai].toarray()
else:
    background_data = X_train_p[:background_sample_size]
    X_test_sample_processed = X_test_p[:sample_size_xai]

if X_test_sample_processed.shape[1] != len(feature_names):
    print(f"\nFeature names shape mismatch: {X_test_sample_processed.shape[1]} vs {len(feature_names)}. Adapting...")
    feature_names = np.array([f"feature_{i}" for i in range(X_test_sample_processed.shape[1])])

# --- Logistic Regression ---

try:
    print("\nGenerating SHAP summary plot for Logistic Regression...")
    lr_model = models_to_explain.get('Logistic Regression')
    if lr_model is not None:
        lr_explainer = shap.LinearExplainer(lr_model, background_data)
        lr_shap_values = lr_explainer.shap_values(X_test_sample_processed)
        if isinstance(lr_shap_values, list):
            lr_shap_values_to_plot = lr_shap_values[1]
        else:
            lr_shap_values_to_plot = lr_shap_values

        # DOT plot (comprehensive)
        plt.figure(figsize=(10, 8))
        shap.summary_plot(
            lr_shap_values_to_plot,
            X_test_sample_processed,
            feature_names=feature_names,
            show=False
        )
        plt.title('SHAP Feature Importance for Logistic Regression')
        plt.tight_layout()
        plt.show()

        # FULL BAR plot (custom, all features)
        lr_importances = np.mean(np.abs(lr_shap_values_to_plot), axis=0).flatten()
        sorted_idx = np.argsort(lr_importances)[::-1]
        plt.figure(figsize=(10, 8))
        plt.barh(np.array(feature_names)[sorted_idx], lr_importances[sorted_idx])
```

```
        plt.xlabel("Mean |SHAP value|")
        plt.title("Global Feature Importance (bar, all features) - Logistic Regression")
        plt.gca().invert_yaxis()
        plt.tight_layout()
        plt.show()
    else:
        print("No Logistic Regression model found in models_to_explain.")

except Exception as e:
    print(f"Error generating SHAP for Logistic Regression: {e}")

# --- ANN/DNN ---

try:
    print("\nGenerating SHAP summary plot for ANN/DNN...")
    ann_model = models_to_explain.get('ANN/DNN', ann_model_tuned)
    if ann_model is not None:
        try:
            ann_explainer = shap.DeepExplainer(ann_model, background_data)
            ann_shap_values = ann_explainer.shap_values(X_test_sample_processed)
        except Exception as e:
            print(f"Error with DeepExplainer: {e}")
            def ann_predict(x):
                return ann_model.predict(x)
            ann_explainer = shap.KernelExplainer(ann_predict, background_data)
            ann_shap_values = ann_explainer.shap_values(X_test_sample_processed)

        if isinstance(ann_shap_values, list):
            ann_shap_values_to_plot = ann_shap_values[1]
        else:
            ann_shap_values_to_plot = ann_shap_values

        # DOT plot (recommended)
        plt.figure(figsize=(10, 8))
        shap.summary_plot(
            ann_shap_values_to_plot,
            X_test_sample_processed,
            feature_names=feature_names,
            show=False
        )
        plt.title('SHAP Feature Importance for ANN/DNN')
        plt.tight_layout()
        plt.show()

        # FULL BAR plot (custom, all features)
        ann_importances = np.mean(np.abs(ann_shap_values_to_plot), axis=0).flatten()
        sorted_idx = np.argsort(ann_importances)[::-1]
        plt.figure(figsize=(10, 8))
        plt.barh(np.array(feature_names)[sorted_idx], ann_importances[sorted_idx])
        plt.xlabel("Mean |SHAP value|")
        plt.title("Global Feature Importance (bar, all features) - ANN/DNN")
        plt.gca().invert_yaxis()
        plt.tight_layout()
        plt.show()
    else:
        print("No ANN/DNN model found in models_to_explain.")

except Exception as e:
    print(f"Error generating SHAP for ANN/DNN: {e}")

print("\nSHAP plots for global feature importance generated for Logistic Regression and ANN/DNN (all features shown).")
```

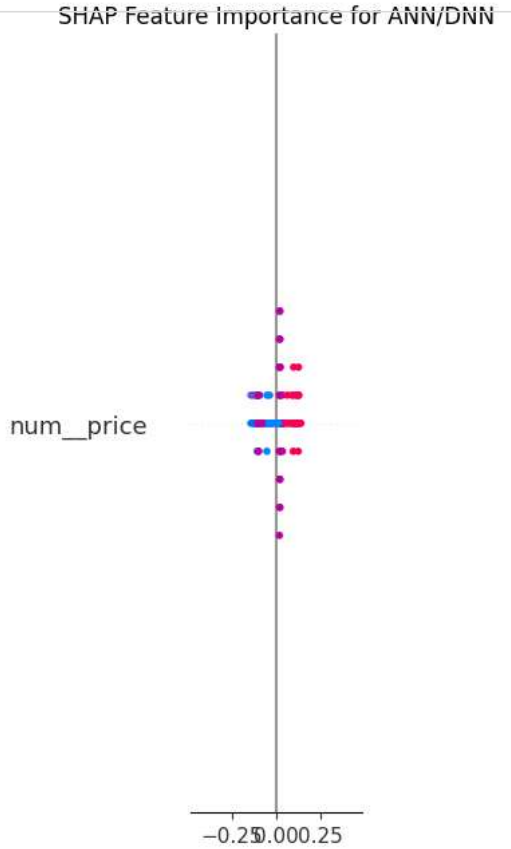
```
Applying SHAP for global explanations on selected models:
Processed training data shape: (7350, 21)
Processed test data shape: (3150, 21)
Feature names: ['num__price' 'num__discount' 'num__ingredients_count' 'num__has_dates'
 'num__has_cocoa' 'num__has_protein' 'num__customer_gender'
 'num__age_numeric' 'num__shelf_life' 'num__clean_label'
 'cat__category_Energy Bars' 'cat__category_Millet Bars'
 'cat__category_Mini Bars' 'cat__category_Muesli'
 'cat__category_Peanut Butter' 'cat__category_Protein Bars'
 'cat__packaging_type_Paper-based'
 'cat__packaging_type_Recyclable Plastic' 'cat__season_Monsoon'
 'cat__season_Summer' 'cat__season_Winter']

Generating SHAP summary plot for Logistic Regression...
```

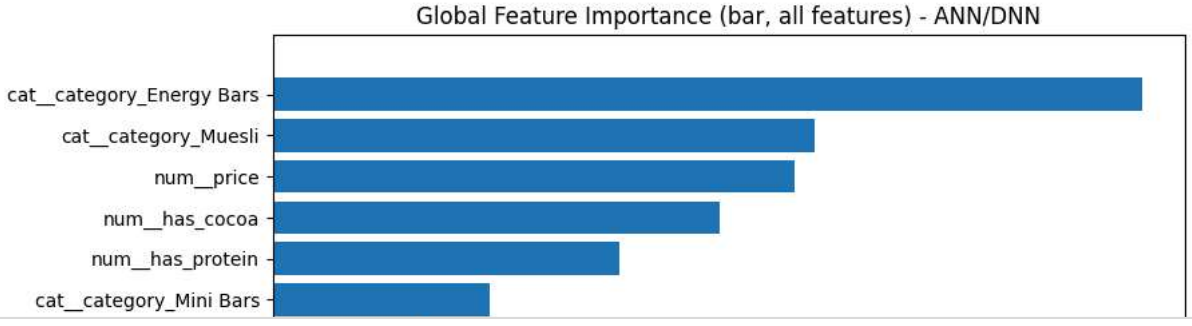


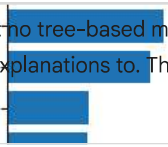





```
Generating SHAP summary plot for ANN/DNN...
<Figure size 1000x800 with 0 Axes>
```







Reasoning: The previous step confirmed that no tree-based models had tuned accuracy less than 1, meaning there are no tree models in `models_to_explain` to apply SHAP global explanations to. The next step is to apply LIME for local explanations to the models selected in the previous subtask.

```
import lime
import lime.lime_tabular
import matplotlib.pyplot as plt
import numpy as np

print("\nApplying LIME for local explanations on selected models:")

# Get feature names from the preprocessor
feature_names = preprocessor.get_feature_names_out()

# Create a LIME Explainer
# Use a small sample of the training data as the training_data for the explainer
# Ensure the background data is dense if the original preprocessed data is sparse
background_sample_size_lime = 100
if hasattr(X_train_p, 'toarray'):
    background_data_lime = X_train_p[:background_sample_size_lime].toarray()
else:
    background_data_lime = X_train_p[:background_sample_size_lime]


# Handle potential issues with feature names if they are numpy arrays
if isinstance(feature_names, np.ndarray):
    feature_names = list(feature_names)


explainer = lime.lime_tabular.LimeTabularExplainer(
    training_data=background_data_lime,
    feature_names=feature_names,
    class_names=['Not Success', 'Success'],
    mode='classification'
)

# Select a few instances from the test sample for local explanation
num_instances_to_explain = 5
instances_to_explain_processed = X_test_p[:num_instances_to_explain]
instances_to_explain_original = X_test_sample.head(num_instances_to_explain) # Keep original for context if needed

# Ensure instances_to_explain_processed is dense if necessary for the explainer
if hasattr(instances_to_explain_processed, 'toarray'):
    instances_to_explain_processed = instances_to_explain_processed.toarray()


# Apply LIME to each selected model and instance
for name, model in models_to_explain.items():
    print(f"\nGenerating LIME local explanations for {name}...")

    # Define the prediction function for LIME
    # For scikit-learn models and Keras models
    if hasattr(model, 'predict_proba'):
        predict_fn = model.predict_proba
    elif hasattr(model, 'predict'):
        # For models without predict_proba (e.g., some SVC configs),
        # we can try using decision_function and scaling, or just predict
        # For LIME classification, predict_proba is preferred.
        # If predict_proba is not available, we'll note it.
        print(f"Warning: Model {name} does not have predict_proba. LIME may not work correctly.")
        continue # Skip this model if predict_proba is essential for LIME

    for i in range(num_instances_to_explain):
        instance_processed = instances_to_explain_processed[i]
        instance_original = instances_to_explain_original.iloc[i]

        # Explain the instance
        explanation = explainer.explain_instance(
            data_row=instance_processed,
            predict_fn=predict_fn,
            num_features=10 # Number of features to show in the explanation
        )

        print(f"\nLIME Explanation for {name} - Instance {i+1}:")
        print(f"  Original Instance Details: {instance_original.to_dict()}")
```

```
            print("  Top contributing features:")
            for feature, weight in explanation.as_list():
                print(f"    - {feature}: {weight:.4f}")

            # Optionally visualize the explanation
            plt.figure(figsize=(8, 5)) # Increased figure size slightly
            explanation.as_pyplot_figure()
            plt.title(f'LIME Explanation for {name} - Instance {i+1}')
            plt.tight_layout()
            plt.show()

    print("\nLIME local explanations generated and visualized for selected models and instances.")
```