Part 1: Dataset Loading

Our goal is to study whether demographic and dietary factors are associated with dietary supplement use, and whether these variables can help predict supplement use. To do this, we import three core components for two time periods: demographic data, dietary intake data, and dietary supplement data.

We use the 2021–2023 cycle as the main dataset for model development and evaluation, and we also load the 2017–March 2020 pre-pandemic cycle as an external dataset for testing generalization.

Although the DR1TOT dataset records only a single day of dietary intake, it is used as an approximation of individuals' usual dietary patterns.

```
import pandas as pd
import numpy as np
np.random.seed(42)

# education, income, and household characteristics.
demo = pd.read_sas("DEMO_L.xpt")
# day 1 total nutrient intake data from the dietary interview.
dr1tot = pd.read_sas("DR1TOT_L.xpt")
# total dietary supplement use data.
dsqtot = pd.read_sas("DSQTOT_L.xpt")

# Previous cycle: 2017-March 2020 NHANES
demo_prev = pd.read_sas("P_DEMO.xpt")
dr1tot_prev = pd.read_sas("P_DR1TOT.xpt")
dsqtot_prev = pd.read_sas("P_DSQTOT.xpt")
```

Part 2: Preprocessing

```
demo_cols = [
    "SEQN",

    # Basic demographics
    "RIDAGEYR",      # age
    "RIAGENDR",      # gender
    "RIDRETH3",      # race/ethnicity, 7 also means missing
```

```
    # Socioeconomic
    "DMDEDUC2",      # education, 7 and 9 also means missing
    "INDFMPIR",      # income-to-poverty ratio
    "DMDHHSIZ",      # household size

    # Marital status (lifestyle proxy)
    "DMDMARTZ"       # 77 and 99 also means missing
]

diet_cols = [
    "SEQN",

    # Energy
    "DR1TKCAL",

    # Macronutrients
    "DR1TPROT",
    "DR1TCARB",
    "DR1TTFAT",

    # Quality indicators
    "DR1TSUGR",   # sugar
    "DR1TFIBE",   # fiber

    # Micronutrients (very important for supplements!)
    "DR1TVC",   # vitamin C
    "DR1TVD",   # vitamin D
    "DR1TCALC",   # calcium
    "DR1TIRON",   # iron
    "DR1TMAGN",   # magnesium
    "DR1TZINC",   # zinc

    # Eating behavior
    "DR1TNUMF"    # number of foods reported
]

supp_cols = [
    "SEQN",
    "DSDCOUNT",   # number of supplements, 77 and 99 also means missing

]
```

```
demo_prev_cols =[
    "SEQN",

    # Basic demographics
```

```
    "RIDAGEYR",      # age
    "RIAGENDR",      # gender
    "RIDRETH3",      # race/ethnicity, 7 also means missing

    # Socioeconomic
    "DMDEDUC2",      # education, 7 and 9 also means missing
    "INDFMPIR",      # income-to-poverty ratio

    # Marital status (lifestyle proxy)
    "DMDMARTZ"       # 77 and 99 also means missing
]

diet_prev_cols = diet_cols

supp_prev_cols = supp_cols
```

```
def replace_special_missing_codes(df, col_code_map):
    """
    Replace NHANES-style special missing codes with NaN.
    col_code_map: dict of {column_name: [codes_to_replace]}
    """
    df = df.copy()
    for col, codes in col_code_map.items():
        if col in df.columns:
            df[col] = df[col].replace(codes, np.nan)
    return df

def safe_divide(numerator, denominator):
    """Divide safely and return NaN when denominator is 0 or missing."""
    result = numerator / denominator
    result = result.replace([np.inf, -np.inf], np.nan)
    return result
```

```
demo_sub = demo[demo_cols].copy()
diet_sub = dr1tot[diet_cols].copy()
supp_sub = dsqtot[supp_cols].copy()

# Clean NHANES special missing codes
demo_missing_codes = {
    "RIDRETH3": [7],
    "DMDEDUC2": [7, 9],
    "DMDMARTZ": [77, 99]
}

supp_missing_codes = {
    "DSDCOUNT": [77, 99]
}
```

```
demo_sub = replace_special_missing_codes(demo_sub, demo_missing_codes)
supp_sub = replace_special_missing_codes(supp_sub, supp_missing_codes)
supp_sub["DSDCOUNT"] = supp_sub["DSDCOUNT"].mask(supp_sub["DSDCOUNT"] < 1, 0)

# Merge the data and drop rows with missing target
df = (
    demo_sub
    .merge(diet_sub, on="SEQN", how="inner")
    .merge(supp_sub, on="SEQN", how="inner")
)
df = df.dropna(subset=["DSDCOUNT"]).copy().reset_index(drop=True)
```

```
demo_prev_sub = demo_prev[demo_prev_cols].copy()
diet_prev_sub = dr1tot_prev[diet_prev_cols].copy()
supp_prev_sub = dsqtot_prev[supp_prev_cols].copy()

demo_prev_sub = replace_special_missing_codes(demo_prev_sub, demo_missing_codes)
supp_prev_sub = replace_special_missing_codes(supp_prev_sub, supp_missing_codes)
supp_prev_sub["DSDCOUNT"] =
supp_prev_sub["DSDCOUNT"].mask(supp_prev_sub["DSDCOUNT"] < 1, 0)

# Merge the data and drop rows with missing target
df_prev = (
    demo_prev_sub
    .merge(diet_prev_sub, on="SEQN", how="inner")
    .merge(supp_prev_sub, on="SEQN", how="inner")
)
df_prev = df_prev.dropna(subset=["DSDCOUNT"]).copy().reset_index(drop=True)
```

We identified extreme inconsistencies in dietary records where macronutrient-derived energy greatly exceeded reported total caloric intake. These cases were removed using a plausibility filter based on macronutrient-to-energy ratios.

Calories from macronutrients are calculated using standard nutritional conversions:

Protein: 4 kcal per gram

Carbohydrates: 4 kcal per gram

Fat: 9 kcal per gram

We define low-intake indicators using threshold values motivated by nutritional interpretation, so that the indicator reflects whether intake is meaningfully low rather than simply below average.

```
df["macro_kcal"] = (
    df["DR1TPROT"] * 4 +
    df["DR1TCARB"] * 4 +
    df["DR1TTFAT"] * 9
```

```
)

df["macro_ratio"] = df["macro_kcal"] / df["DR1TKCAL"]

df = df[(df["macro_ratio"] > 0.5) & (df["macro_ratio"] < 1.5)]

# Macronutrient ratios, multipliers are applied to convert grams to calories
df["protein_ratio"] = safe_divide(df["DR1TPROT"] * 4, df["DR1TKCAL"])
df["carb_ratio"]    = safe_divide(df["DR1TCARB"] * 4, df["DR1TKCAL"])
df["fat_ratio"]     = safe_divide(df["DR1TTFAT"] * 9, df["DR1TKCAL"])

# Diet quality indicators, multipliers are applied to convert grams to calories
df["fiber_density"] = safe_divide(df["DR1TFIBE"] * 2, df["DR1TKCAL"])
df["sugar_density"] = safe_divide(df["DR1TSUGR"] * 4, df["DR1TKCAL"])

# Micronutrient low-intake indicators
df["low_vitamin_c"] = (df["DR1TVC"] < 75).astype(float)
df["low_calcium"] = (df["DR1TCALC"] < 1000).astype(float)
df["low_iron"] = (df["DR1TIRON"] < 8).astype(float)
df["low_vitamin_d"] = (df["DR1TVD"] < 15).astype(float)
df["low_magnesium"] = (df["DR1TMAGN"] < 310).astype(float)
df["low_zinc"] = (df["DR1TZINC"] < 8).astype(float)

# Age groups
df["age_group"] = pd.cut(
        df["RIDAGEYR"],
        bins=[0, 18, 35, 50, 65, 80],
        labels=["child", "young_adult", "adult", "middle_age", "senior"],
        right=False
)

# Binary classification target
df["supp_use"] = (df["DSDCOUNT"] > 0).astype(int)

# Regression target
df["supp_count"] = df["DSDCOUNT"]

# Multi-class target
df["supp_usage_count"] = pd.cut(
    df["DSDCOUNT"],
    bins=[-1, 0, 5, np.inf],
    labels=["none", "1-5", "5+"]
)
```

```
df_prev["macro_kcal"] = (
    df_prev["DR1TPROT"] * 4 +
    df_prev["DR1TCARB"] * 4 +
```

```
    df_prev["DR1TTFAT"] * 9
)

df_prev["macro_ratio"] = df_prev["macro_kcal"] / df_prev["DR1TKCAL"]

df_prev = df_prev[(df_prev["macro_ratio"] > 0.5) & (df_prev["macro_ratio"] <
1.5)]

# Macronutrient ratios, multipliers are applied to convert grams to calories
df_prev["protein_ratio"] = safe_divide(df_prev["DR1TPROT"] * 4,
df_prev["DR1TKCAL"])
df_prev["carb_ratio"]    = safe_divide(df_prev["DR1TCARB"] * 4,
df_prev["DR1TKCAL"])
df_prev["fat_ratio"]     = safe_divide(df_prev["DR1TTFAT"] * 9,
df_prev["DR1TKCAL"])

# Diet quality indicators, multipliers are applied to convert grams to calories
df_prev["fiber_density"] = safe_divide(df_prev["DR1TFIBE"] * 2,
df_prev["DR1TKCAL"])
df_prev["sugar_density"] = safe_divide(df_prev["DR1TSUGR"] * 4,
df_prev["DR1TKCAL"])

# Micronutrient low-intake indicators
df_prev["low_vitamin_c"] = (df_prev["DR1TVC"] < 75).astype(float)
df_prev["low_calcium"] = (df_prev["DR1TCALC"] < 1000).astype(float)
df_prev["low_iron"] = (df_prev["DR1TIRON"] < 8).astype(float)
df_prev["low_vitamin_d"] = (df_prev["DR1TVD"] < 15).astype(float)
df_prev["low_magnesium"] = (df_prev["DR1TMAGN"] < 310).astype(float)
df_prev["low_zinc"] = (df_prev["DR1TZINC"] < 8).astype(float)

# Age groups
df_prev["age_group"] = pd.cut(
        df_prev["RIDAGEYR"],
        bins=[0, 18, 35, 50, 65, 80],
        labels=["child", "young_adult", "adult", "middle_age", "senior"],
        right=False
)

# Binary classification target
df_prev["supp_use"] = (df_prev["DSDCOUNT"] > 0).astype(int)

# Regression target
df_prev["supp_count"] = df_prev["DSDCOUNT"]

# Multi-class target
df_prev["supp_usage_count"] = pd.cut(
    df_prev["DSDCOUNT"],
    bins=[-1, 0, 5, np.inf],
```

```
    labels=["none", "1-5", "5+"]
)
```

```
# The following maps numbers in columns to what they actually means

gender_map = {
    1: "Male",
    2: "Female"
}

race_map = {
    1: "Mexican American",
    2: "Other Hispanic",
    3: "Non-Hispanic White",
    4: "Non-Hispanic Black",
    6: "Non-Hispanic Asian",
}

education_map = {
    1: "Less than 9th grade",
    2: "9-11th grade",
    3: "High school / GED",
    4: "Some college / AA",
    5: "College graduate or above"
}

marital_map = {
    1: "Married/Living with partner",
    2: "Widowed/Divorced/Separated",
    3: "Never married"
}

df["gender_label"] = df["RIAGENDR"].map(gender_map)
df["race_label"] = df["RIDRETH3"].map(race_map)
df["education_label"] = df["DMDEDUC2"].map(education_map)
df["marital_label"] = df["DMDMARTZ"].map(marital_map)

df_prev["gender_label"] = df_prev["RIAGENDR"].map(gender_map)
df_prev["race_label"] = df_prev["RIDRETH3"].map(race_map)
df_prev["education_label"] = df_prev["DMDEDUC2"].map(education_map)
df_prev["marital_label"] = df_prev["DMDMARTZ"].map(marital_map)
```

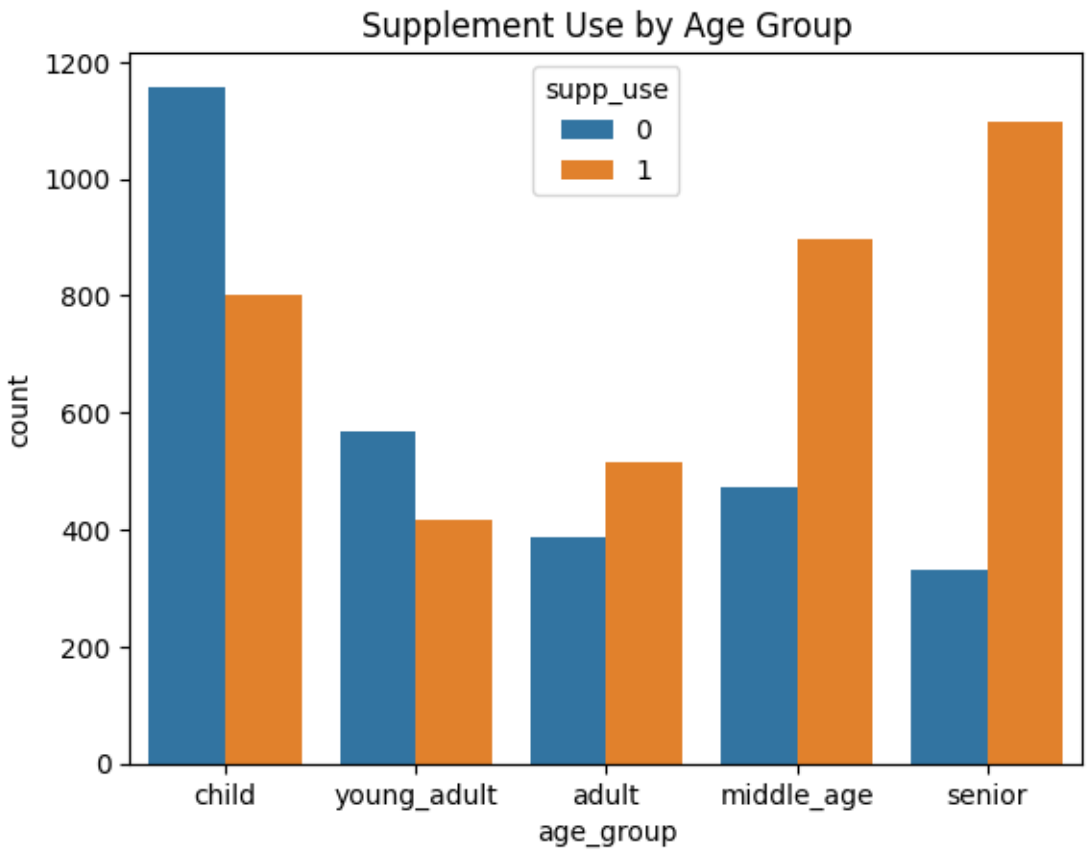
```
df_before_impute = df.copy()

# Separate numeric and categorical columns
num_cols = df.select_dtypes(include=[np.number]).columns.tolist()
cat_cols = df.select_dtypes(include=["object", "category"]).columns.tolist()
```

```
# Do not impute SEQN or the targets
exclude_from_impute = {"SEQN", "DSDCOUNT", "supp_use", "supp_count"}
num_impute_cols = [c for c in num_cols if c not in exclude_from_impute]

# Median imputation for numeric
for col in num_impute_cols:
    df[col] = df[col].fillna(df[col].median())

# Mode imputation for categorical
for col in cat_cols:
    if df[col].isna().any():
        mode_vals = df[col].mode(dropna=True)
        df[col] = df[col].fillna(mode_vals.iloc[0])
```

```
df_prev_before_impute = df_prev.copy()

# Separate numeric and categorical columns
num_cols_prev = df_prev.select_dtypes(include=[np.number]).columns.tolist()
cat_cols_prev = df_prev.select_dtypes(include=["object",
"category"]).columns.tolist()

# Do not impute SEQN or the targets
num_impute_cols_prev = [c for c in num_cols_prev if c not in exclude_from_impute]

# Median imputation for numeric
for col in num_impute_cols_prev:
    df_prev[col] = df_prev[col].fillna(df_prev[col].median())

# Mode imputation for categorical
for col in cat_cols_prev:
    if df_prev[col].isna().any():
        mode_vals_prev = df_prev[col].mode(dropna=True)
        df_prev[col] = df_prev[col].fillna(mode_vals_prev.iloc[0])
```

Part 3: Exploratory Data Analysis

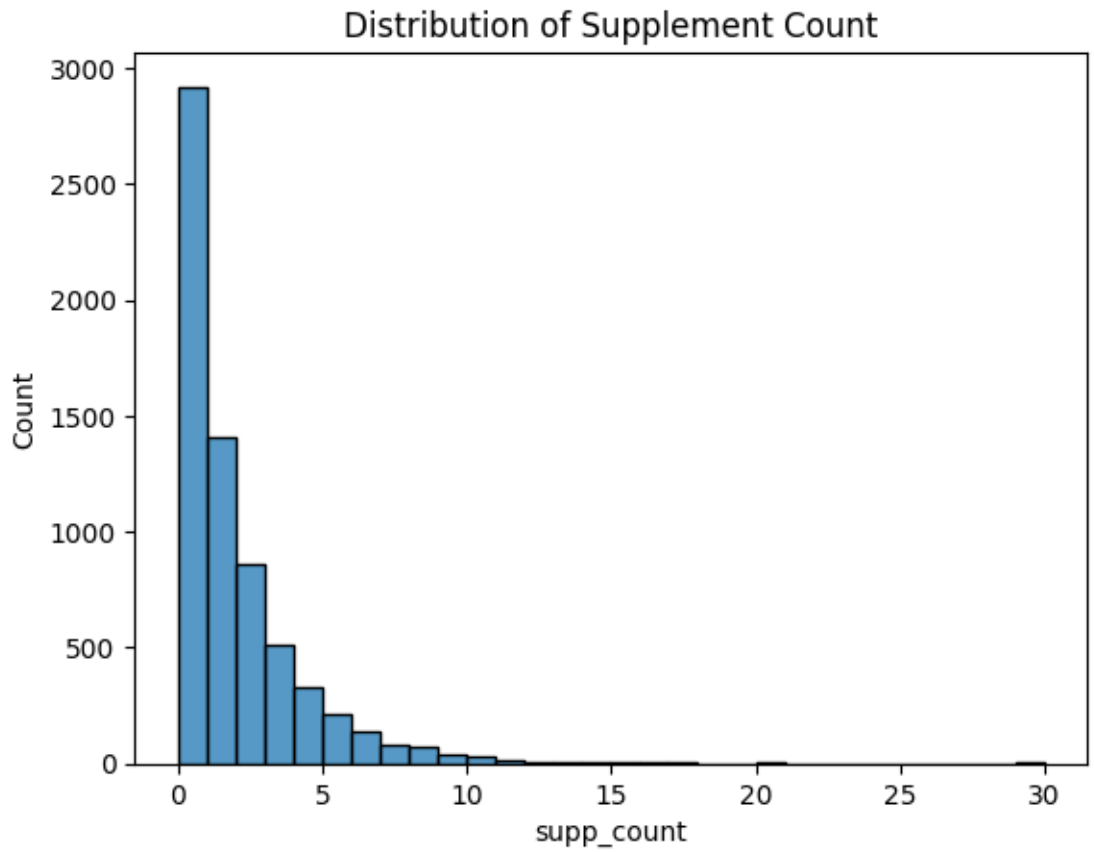
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.countplot(x="age_group", hue="supp_use", data=df)
plt.title("Supplement Use by Age Group")
plt.show()
```



Younger individuals (especially children) are less likely to use supplements, while older groups—particularly middle-aged and senior individuals—are much more likely to report supplement use.

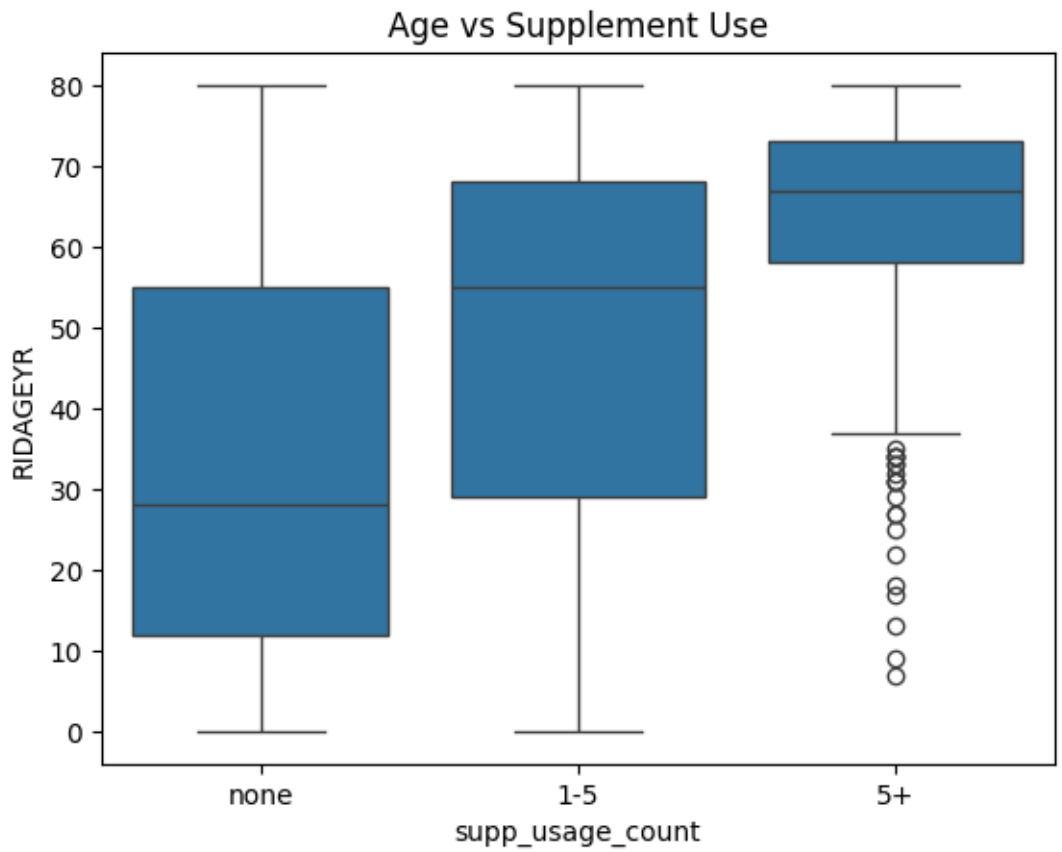
This suggests that age is strongly associated with supplement behavior, possibly reflecting increased health awareness or medical needs in older populations. As a result, age is expected to be an important predictor in the classification models.

```
sns.histplot(df["supp_count"], bins=30)
plt.title("Distribution of Supplement Count")
plt.show()
```



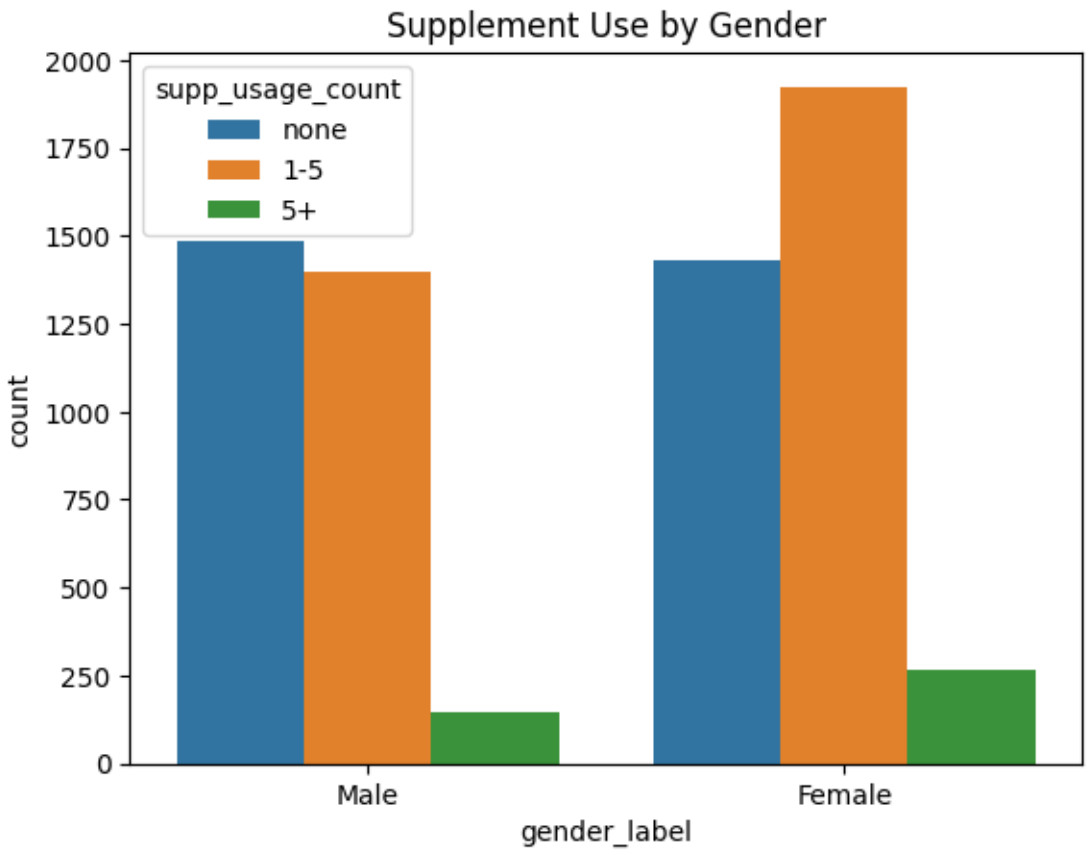
The distribution of supplement count is highly right-skewed, with most individuals taking few or no supplements, and a small number taking many supplements. This indicates that high supplement usage (e.g., 5+) is relatively rare, which may lead to class imbalance in the multiclass setting.

```
sns.boxplot(x="supp_usage_count", y="RIDAGEYR", data=df)
plt.title("Age vs Supplement Use")
plt.show()
```



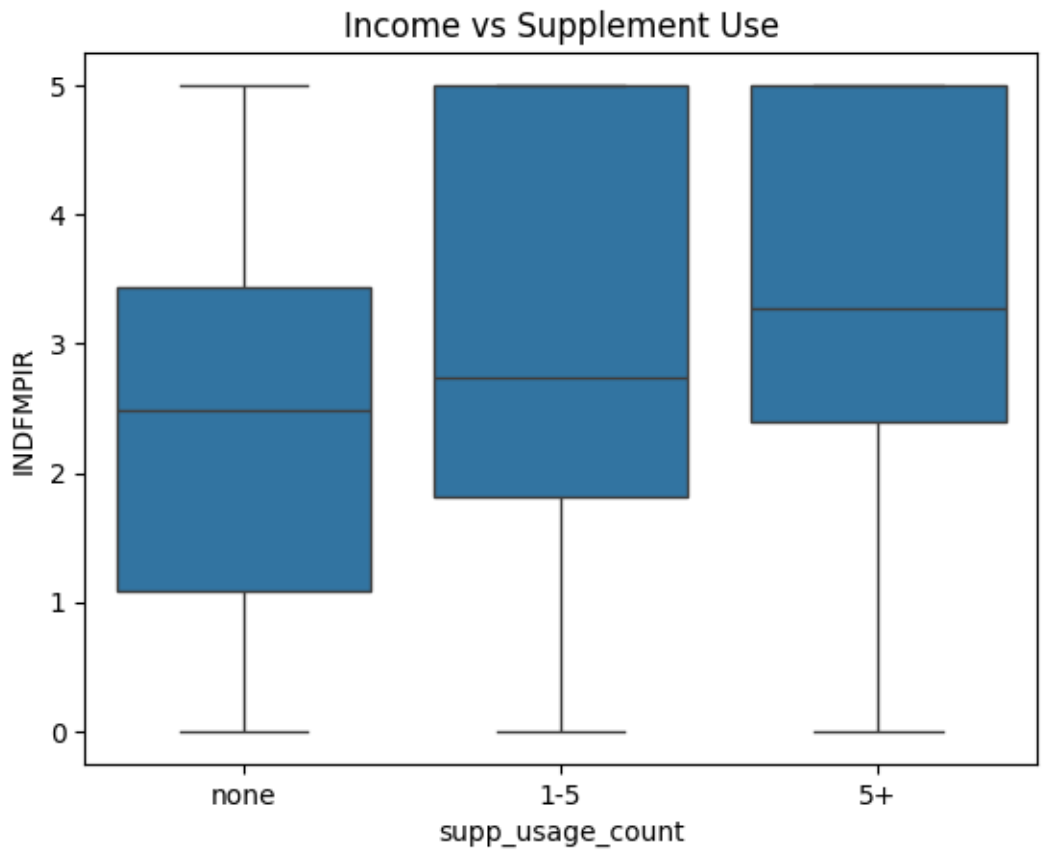
Individuals who take more supplements tend to be older. The median age increases steadily from the "none" group to the "5+" group. This reinforces the earlier finding that age is strongly associated with supplement use.

```
sns.countplot(x="gender_label", hue="supp_usage_count", data=df)
plt.title("Supplement Use by Gender")
plt.show()
```



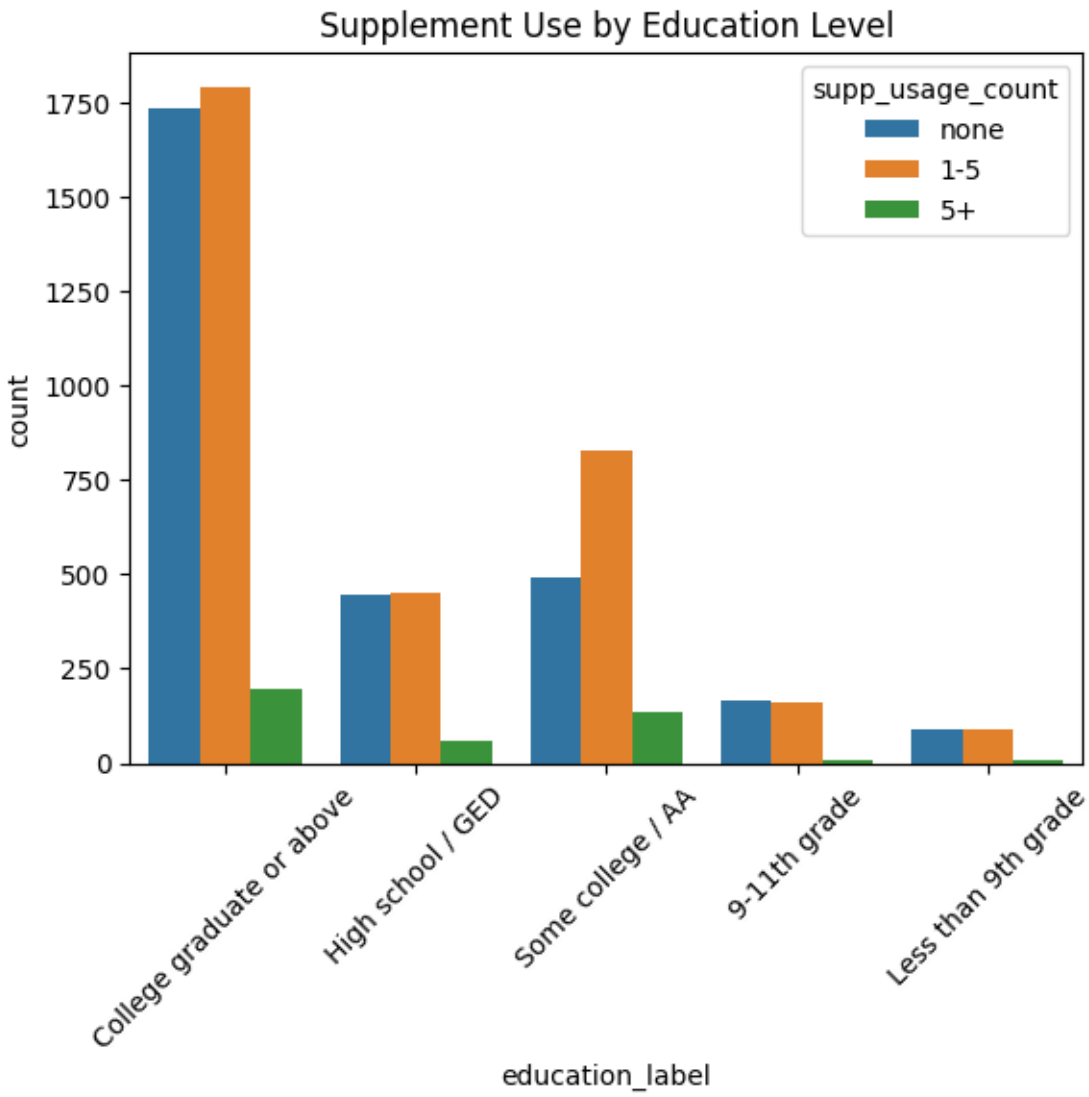
Females are slightly more likely than males to use supplements. This suggests that gender plays a role in supplement behavior, potentially reflecting differences in health awareness or lifestyle. However, the difference is moderate, indicating that gender alone is unlikely to fully explain supplement use.

```
sns.boxplot(x="supp_usage_count", y="INDFMPIR", data=df)
plt.title("Income vs Supplement Use")
plt.show()
```



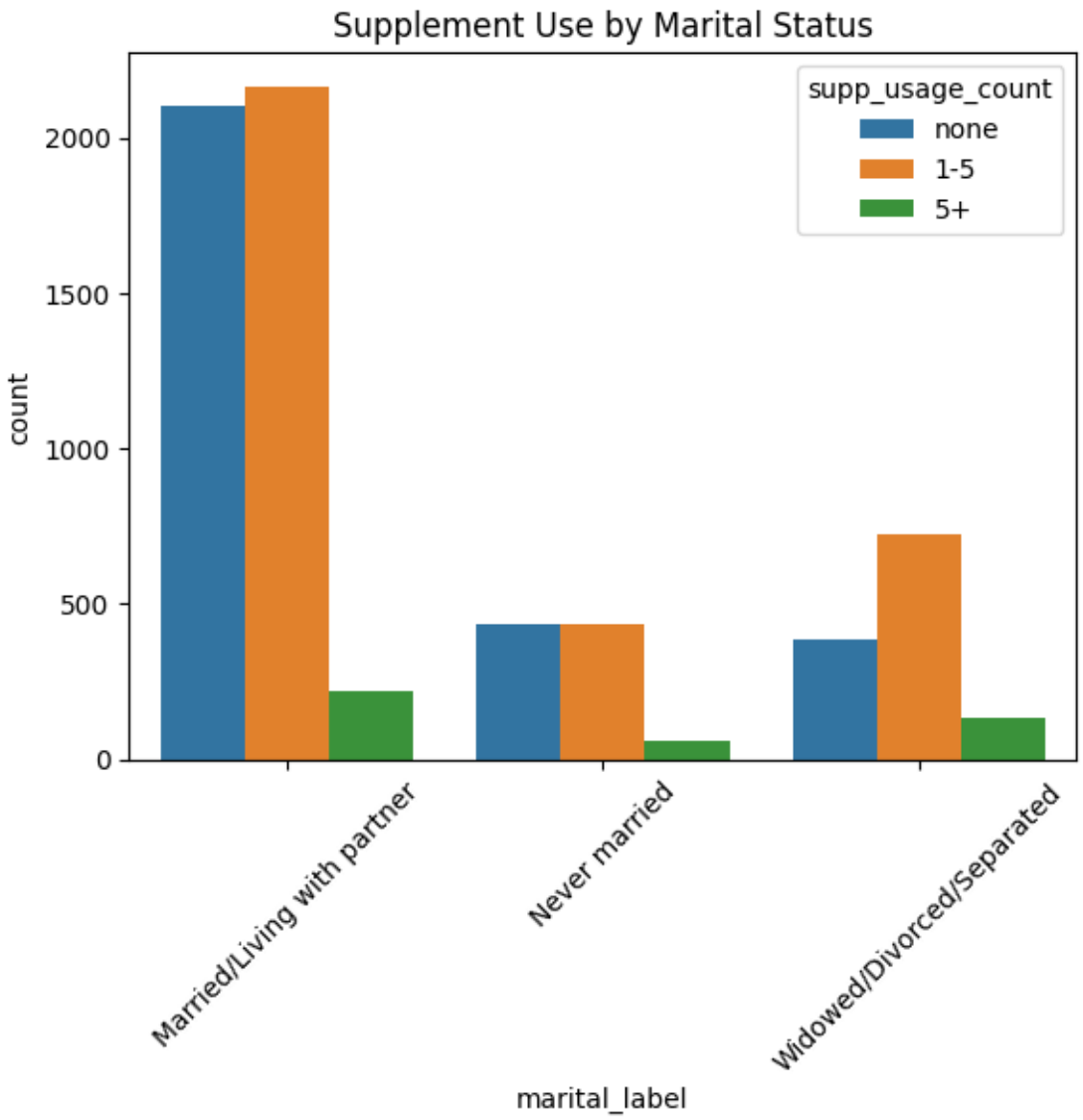
Individuals who take more supplements tend to have higher income (as measured by PIR). This suggests that supplement use is associated with higher socioeconomic status, likely because supplements are discretionary health-related purchases.

```
sns.countplot(x="education_label", hue="supp_usage_count", data=df)
plt.title("Supplement Use by Education Level")
plt.xticks(rotation=45)
plt.show()
```



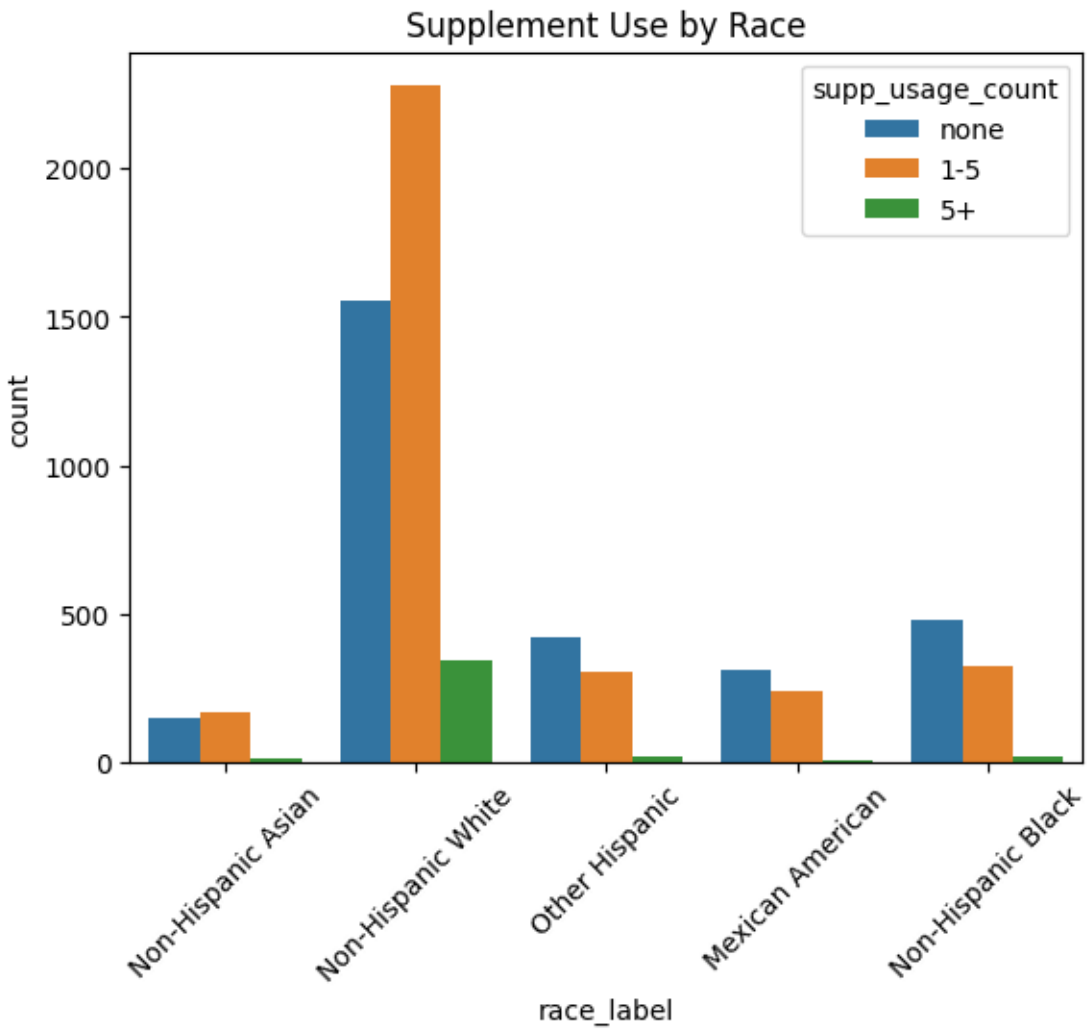
The differences across education levels are less pronounced compared to other variables. While some variation exists, no strong pattern clearly distinguishes supplement users from non-users. This suggests that education levels may have a weaker association with supplement use and may contribute less to predictive performance compared to variables like age or income.

```
sns.countplot(x="marital_label", hue="supp_usage_count", data=df)
plt.title("Supplement Use by Marital Status")
plt.xticks(rotation=45)
plt.show()
```



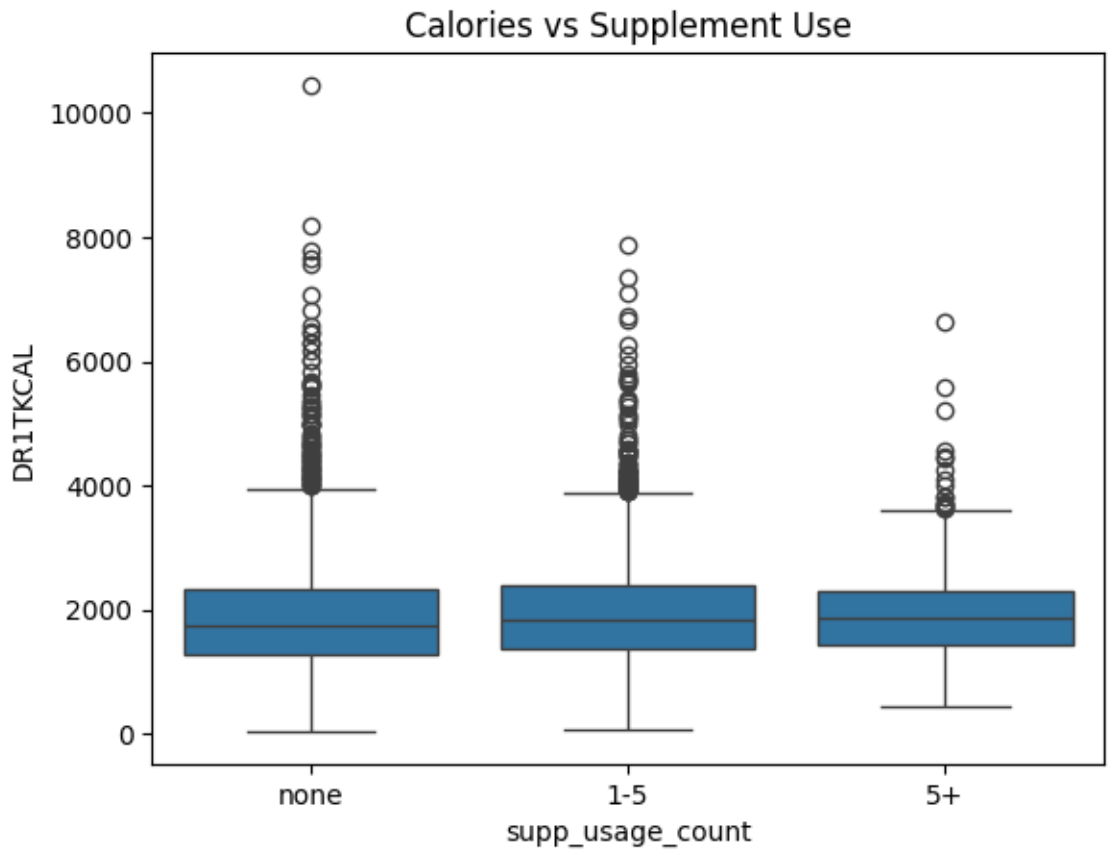
The differences across marital status groups are less pronounced compared to other variables. While some variation exists, no strong pattern clearly distinguishes supplement users from non-users. This suggests that marital status may have a weaker association with supplement use and may contribute less to predictive performance compared to variables like age or income.

```
sns.countplot(x="race_label", hue="supp_usage_count", data=df)
plt.title("Supplement Use by Race")
plt.xticks(rotation=45)
plt.show()
```



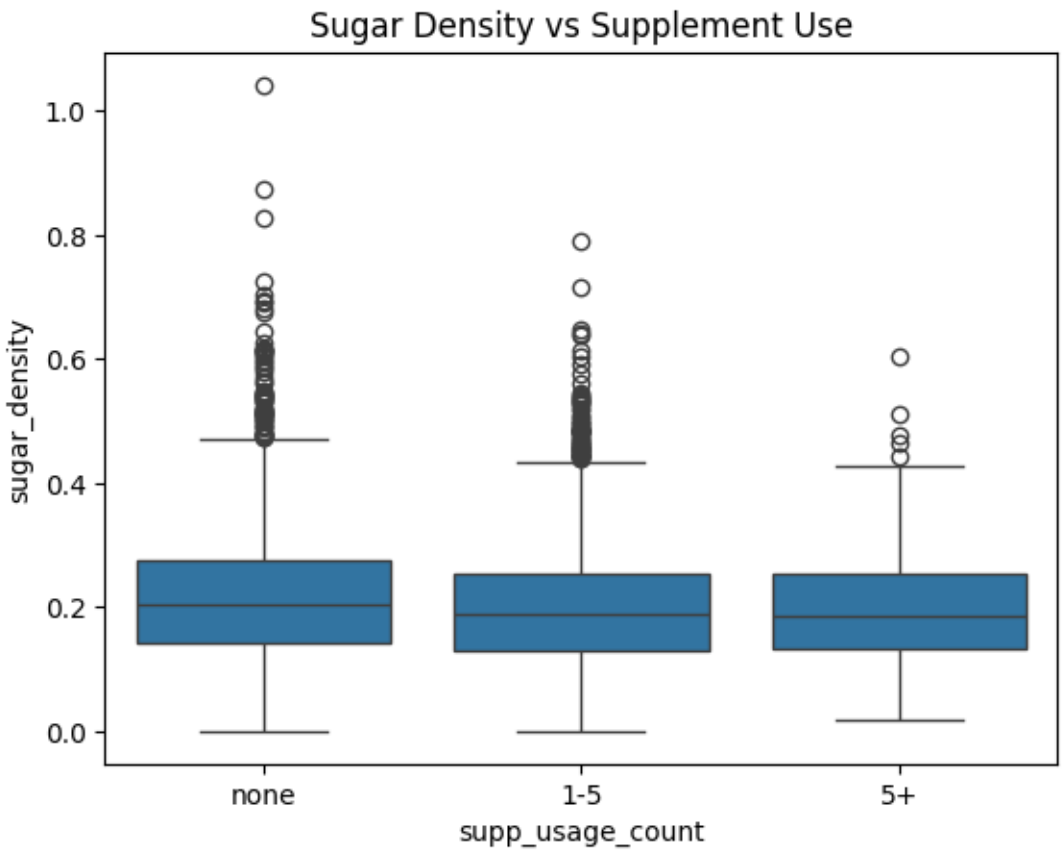
The chart shows variation in supplement use across racial groups. For example, Non-Hispanic White individuals appear to have higher supplement usage compared to some other groups. However, these differences should be interpreted cautiously, as they may reflect underlying socioeconomic, cultural, or access-related factors rather than direct causal effects.

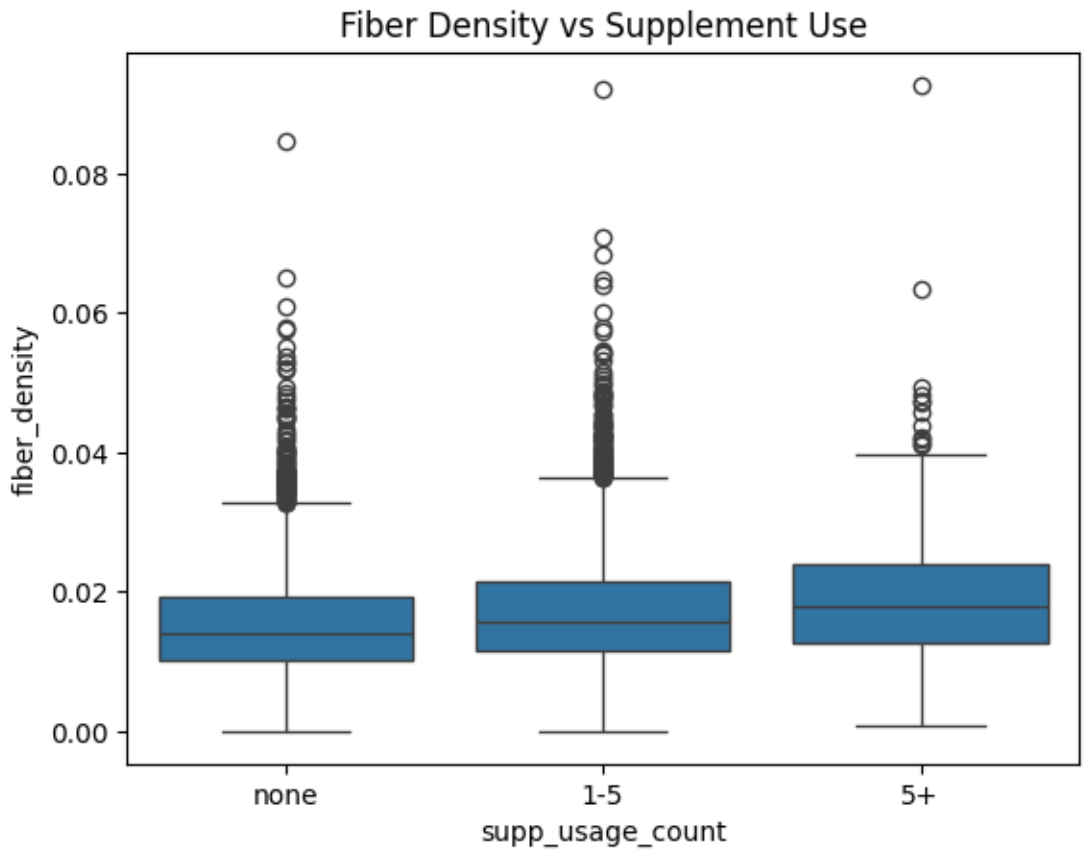
```
sns.boxplot(x="supp_usage_count", y="DR1TKCAL", data=df)
plt.title("Calories vs Supplement Use")
plt.show()
```



The boxplots show that total calorie intake is relatively similar across supplement use groups. This suggests that total energy intake alone does not strongly distinguish supplement users from non-users.

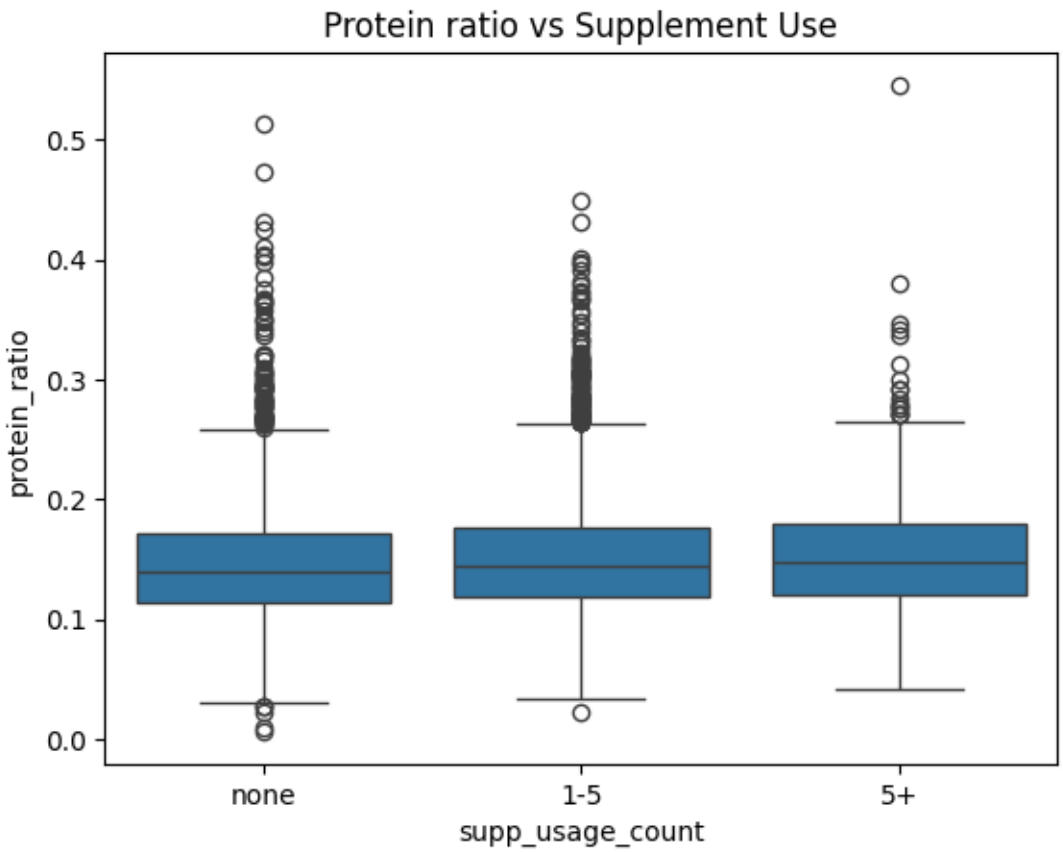
```
sns.boxplot(x="supp_usage_count", y="sugar_density", data=df)
plt.title("Sugar Density vs Supplement Use")
plt.show()

sns.boxplot(x="supp_usage_count", y="fiber_density", data=df)
plt.title("Fiber Density vs Supplement Use")
plt.show()
```

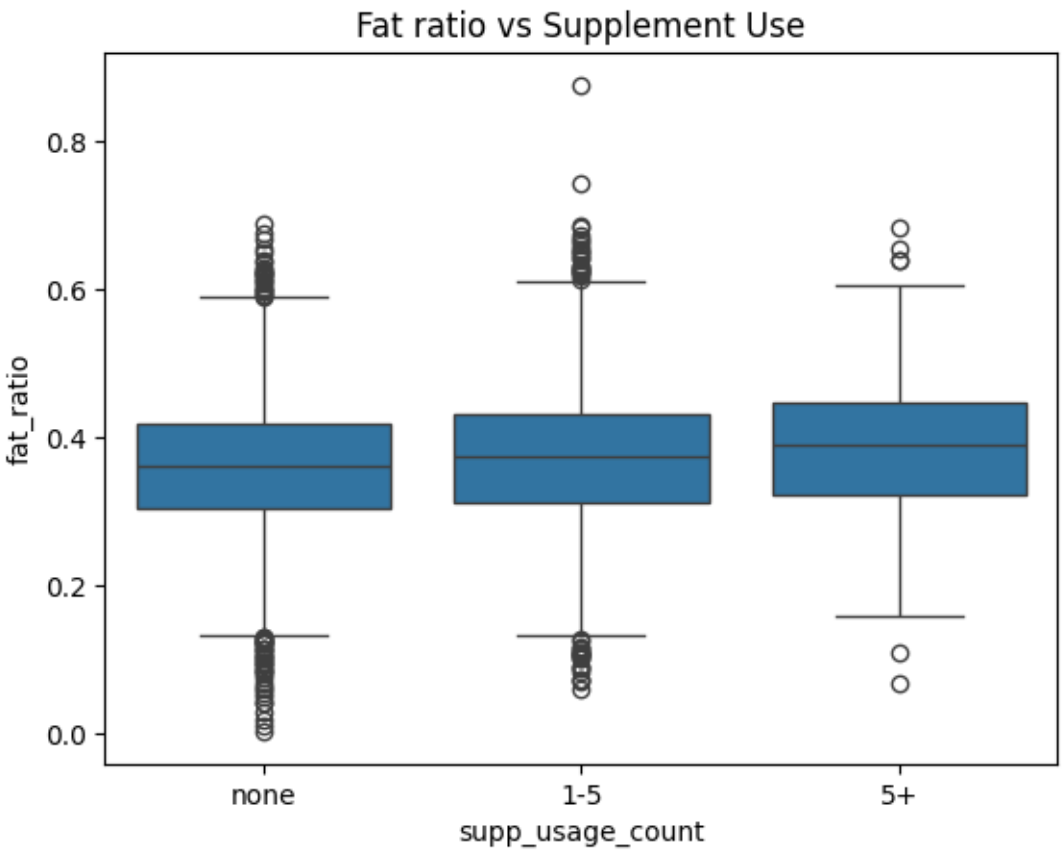

Sugar Density vs Supplement Use
sugar_density
1.0
0.8
0.6
0.4
0.2
0.0
none
1-5
5+
supp_usage_count

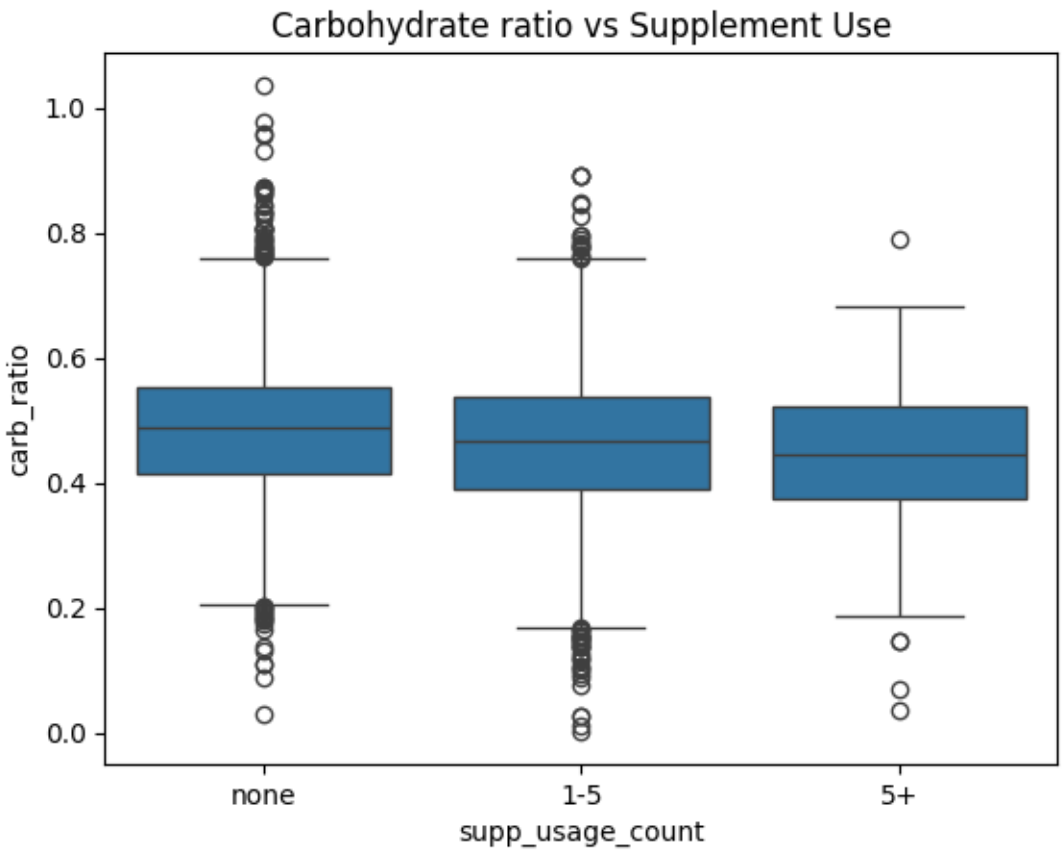


Sugar density appears slightly lower for individuals who take more supplements, and fiber appears higher, although the distributions overlap considerably. This may suggest that supplement users have slightly healthier dietary patterns, but the effect is weak. Sugar/fiber intake alone is unlikely to be a strong predictor.

```
sns.boxplot(x="supp_usage_count", y="protein_ratio", data=df)
plt.title("Protein ratio vs Supplement Use")
plt.show()

sns.boxplot(x="supp_usage_count", y="fat_ratio", data=df)
plt.title("Fat ratio vs Supplement Use")
plt.show()

sns.boxplot(x="supp_usage_count", y="carb_ratio", data=df)
plt.title("Carbohydrate ratio vs Supplement Use")
plt.show()
```


Protein ratio vs Supplement Use
protein_ratio
0.0
0.1
0.2
0.3
0.4
0.5
none
1-5
5+
supp_usage_count


Fat ratio vs Supplement Use
fat_ratio
0.0
0.2
0.4
0.6
0.8
none
1-5
5+
supp_usage_count

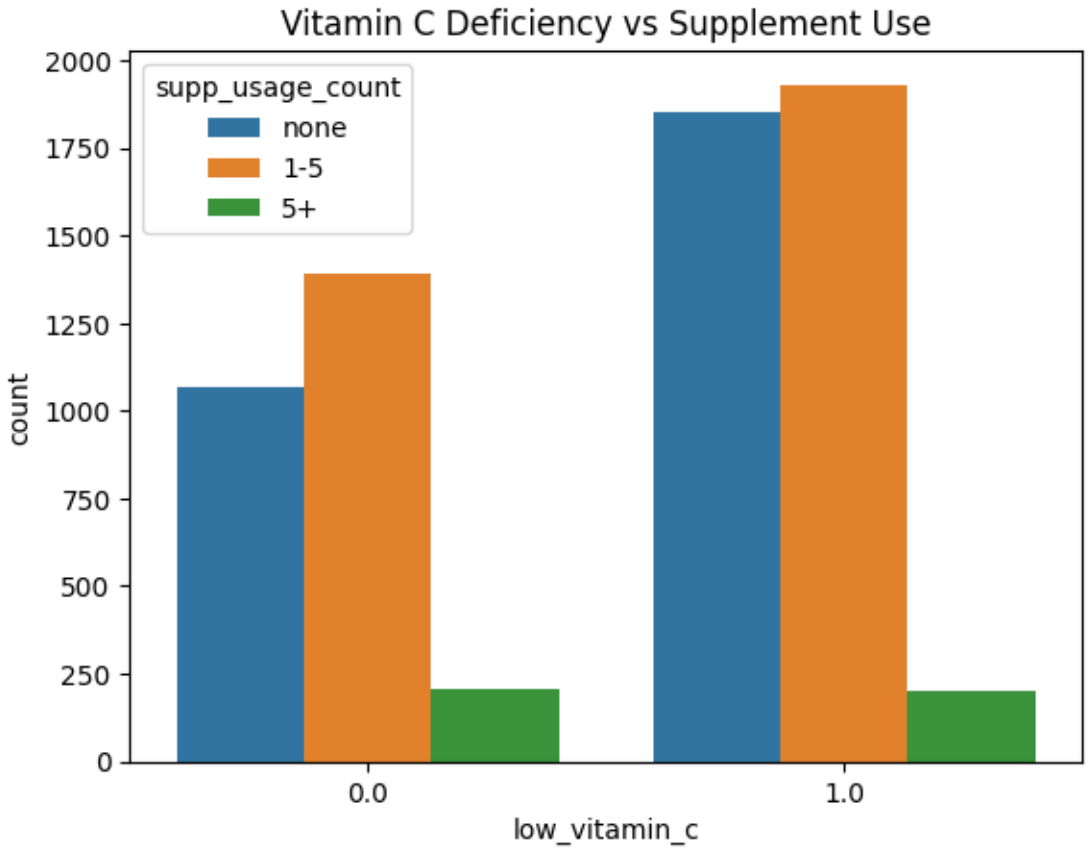


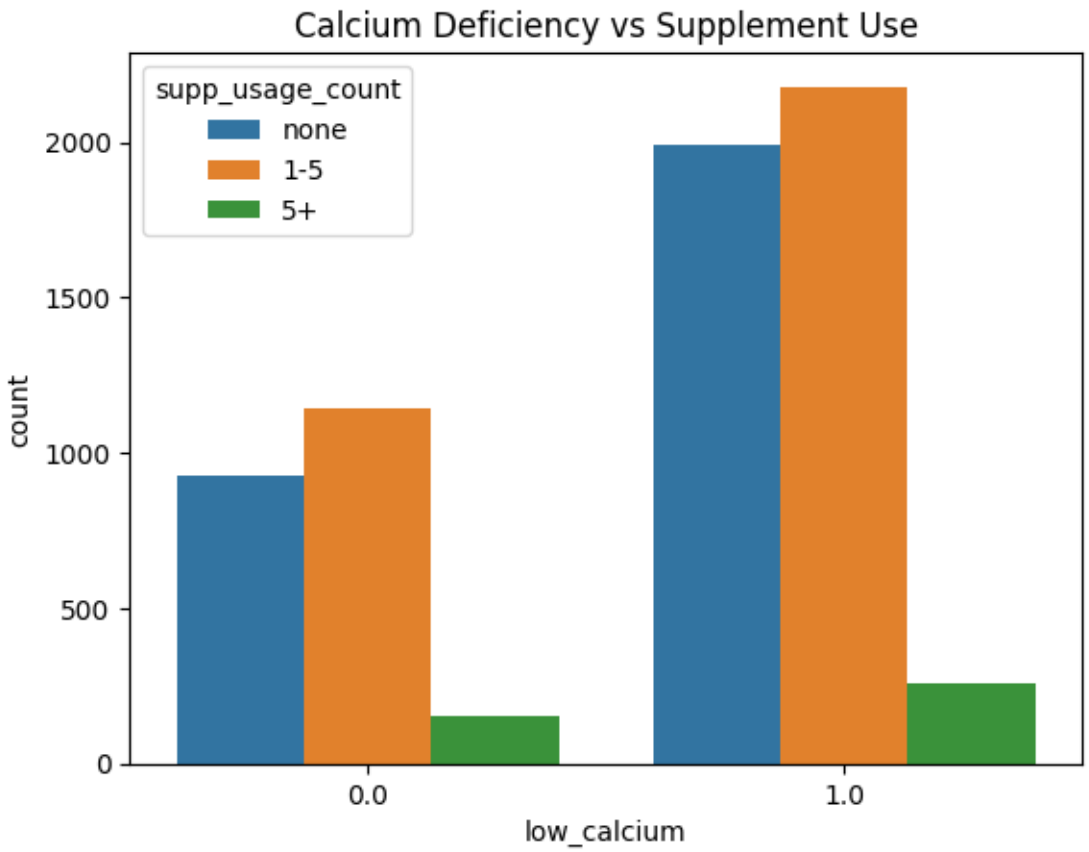
Protein ratio shows only minor differences across groups, fat ratio shows a slight increase for higher supplement use groups, and carbohydrate ratio tends to decrease slightly as supplement use increases. However, the differences are small and distributions overlap.

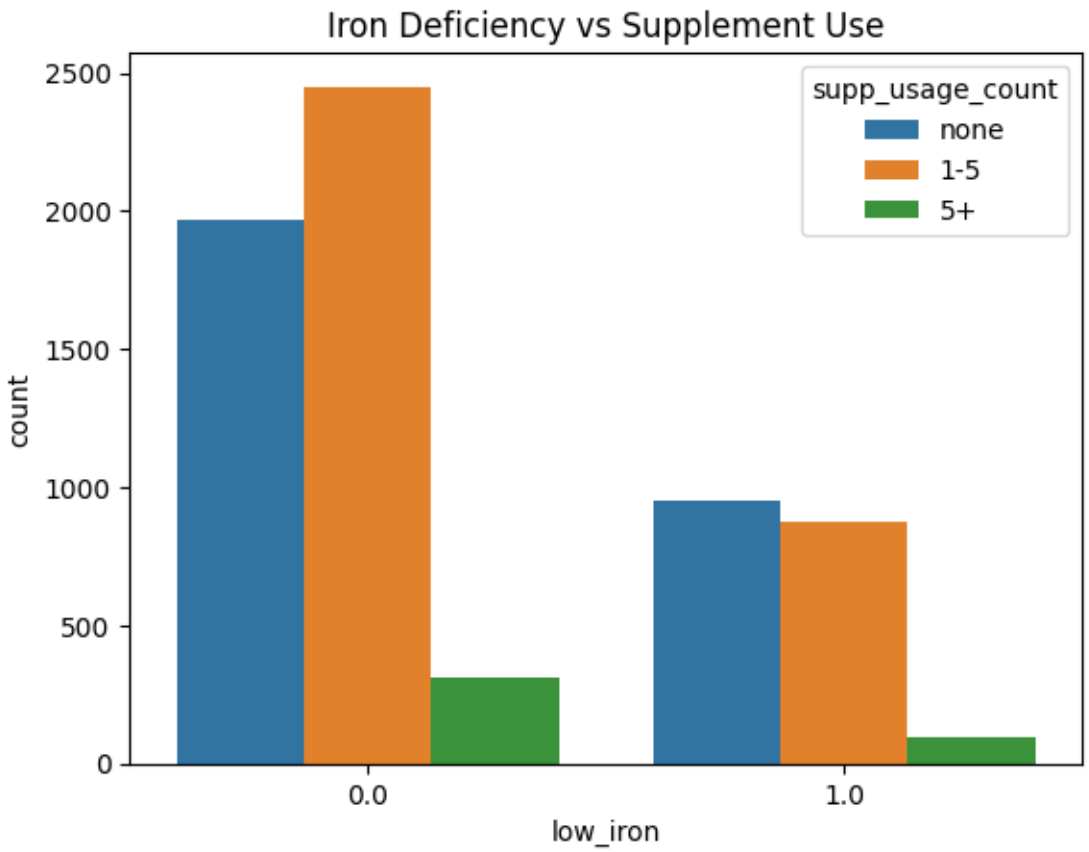
This suggests only a weak/none association, meaning all proportions alone is unlikely to strongly predict supplement use.

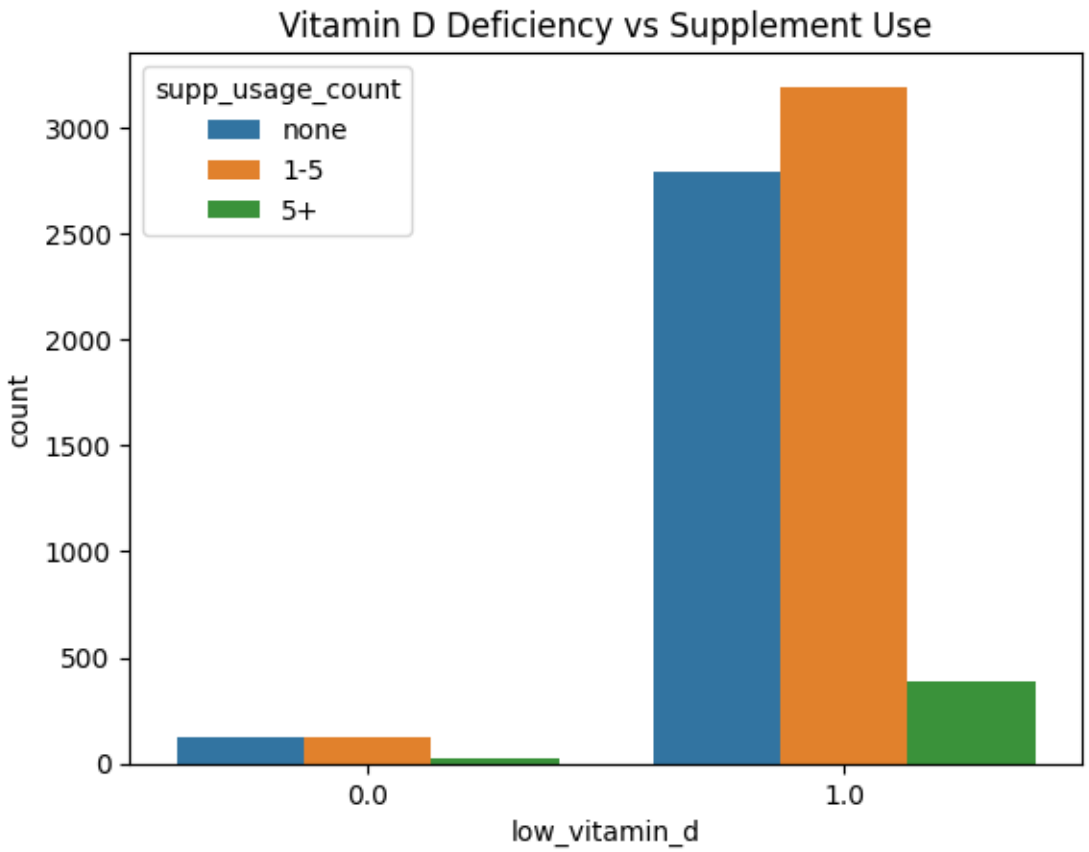
```
sns.countplot(x="low_vitamin_c", hue="supp_usage_count", data=df)
plt.title("Vitamin C Deficiency vs Supplement Use")
plt.show()

sns.countplot(x="low_calcium", hue="supp_usage_count", data=df)
plt.title("Calcium Deficiency vs Supplement Use")
plt.show()

sns.countplot(x="low_iron", hue="supp_usage_count", data=df)
plt.title("Iron Deficiency vs Supplement Use")
plt.show()

sns.countplot(x="low_vitamin_d", hue="supp_usage_count", data=df)
plt.title("Vitamin D Deficiency vs Supplement Use")
plt.show()
```

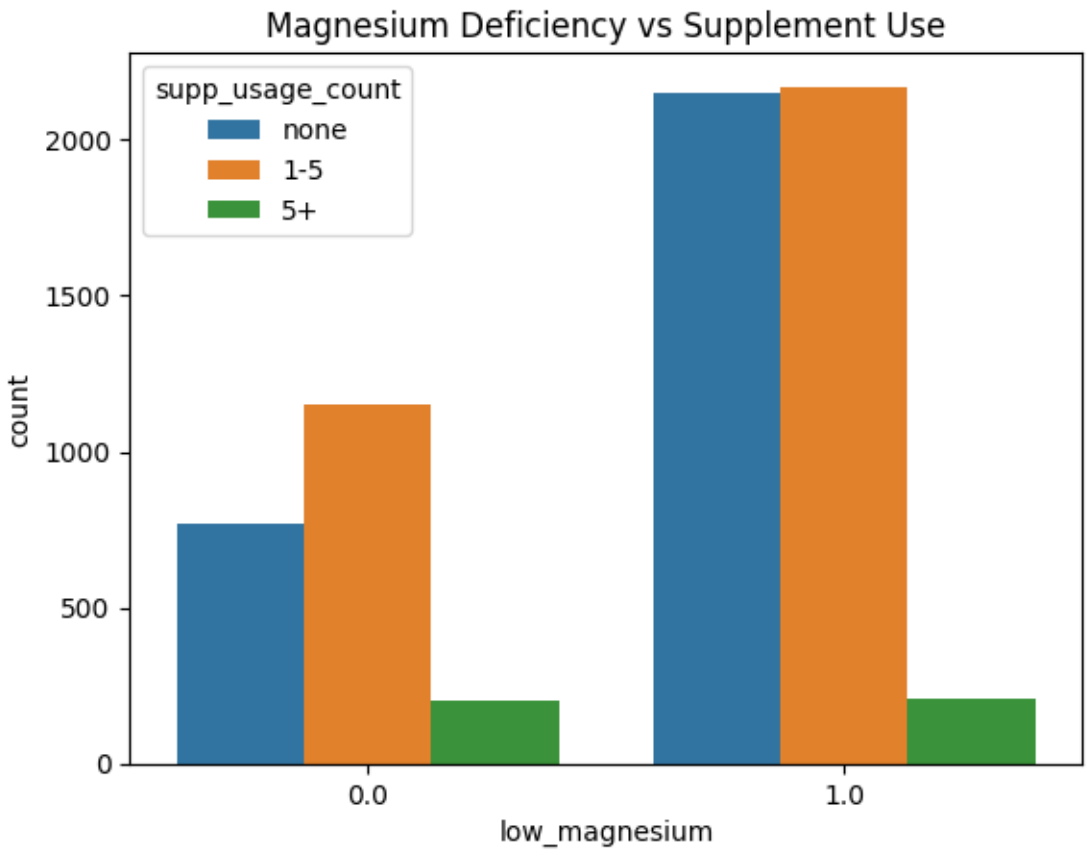
```
sns.countplot(x="low_magnesium", hue="supp_usage_count", data=df)
plt.title("Magnesium Deficiency vs Supplement Use")
plt.show()

sns.countplot(x="low_zinc", hue="supp_usage_count", data=df)
plt.title("Zinc Deficiency vs Supplement Use")
plt.show()
```

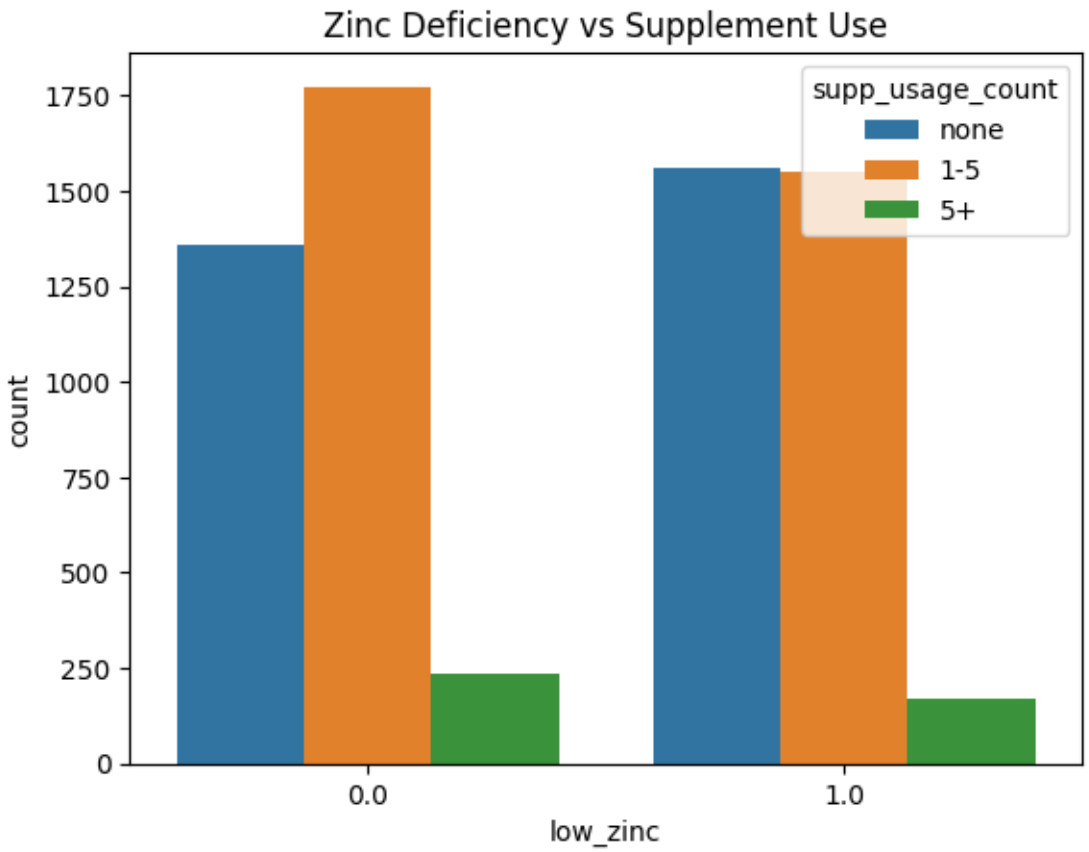



Calcium Deficiency vs Supplement Use
supp_usage_count
none
1-5
5+
2000
1500
1000
500
0
count
0.0
1.0
low_calcium


Iron Deficiency vs Supplement Use
supp_usage_count
none
1-5
5+
count
2500
2000
1500
1000
500
0
0.0
1.0
low_iron


Vitamin D Deficiency vs Supplement Use
supp_usage_count
none
1-5
5+
3000
2500
2000
1500
1000
500
0
count
0.0
1.0
low_vitamin_d


Magnesium Deficiency vs Supplement Use
supp_usage_count
none
1-5
5+
2000
1500
1000
500
0
count
0.0
1.0
low_magnesium



All low-intake indicators suggests only a weak association, meaning them alone is unlikely to strongly predict supplement use.

```
cluster_features = [
    # Demographics
    "RIDAGEYR",
    "INDFMPIR",

    # Macronutrients (composition)
    "protein_ratio",
    "carb_ratio",
    "fat_ratio",

    # Diet quality
    "fiber_density",
    "sugar_density",

    # Micronutrients (RAW intake - important!)
    "DR1TVC",
    "DR1TCALC",
    "DR1TIRON",
    "DR1TVD",
    "DR1TMAGN",
```

```
    "DR1TZINC"
]
```

```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans

# Standardize the features
X = df[cluster_features].copy()
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
# Fit the clusters
kmeans = KMeans(n_clusters=4, max_iter=300, random_state=42)
df["cluster"] = kmeans.fit_predict(X_scaled)
```

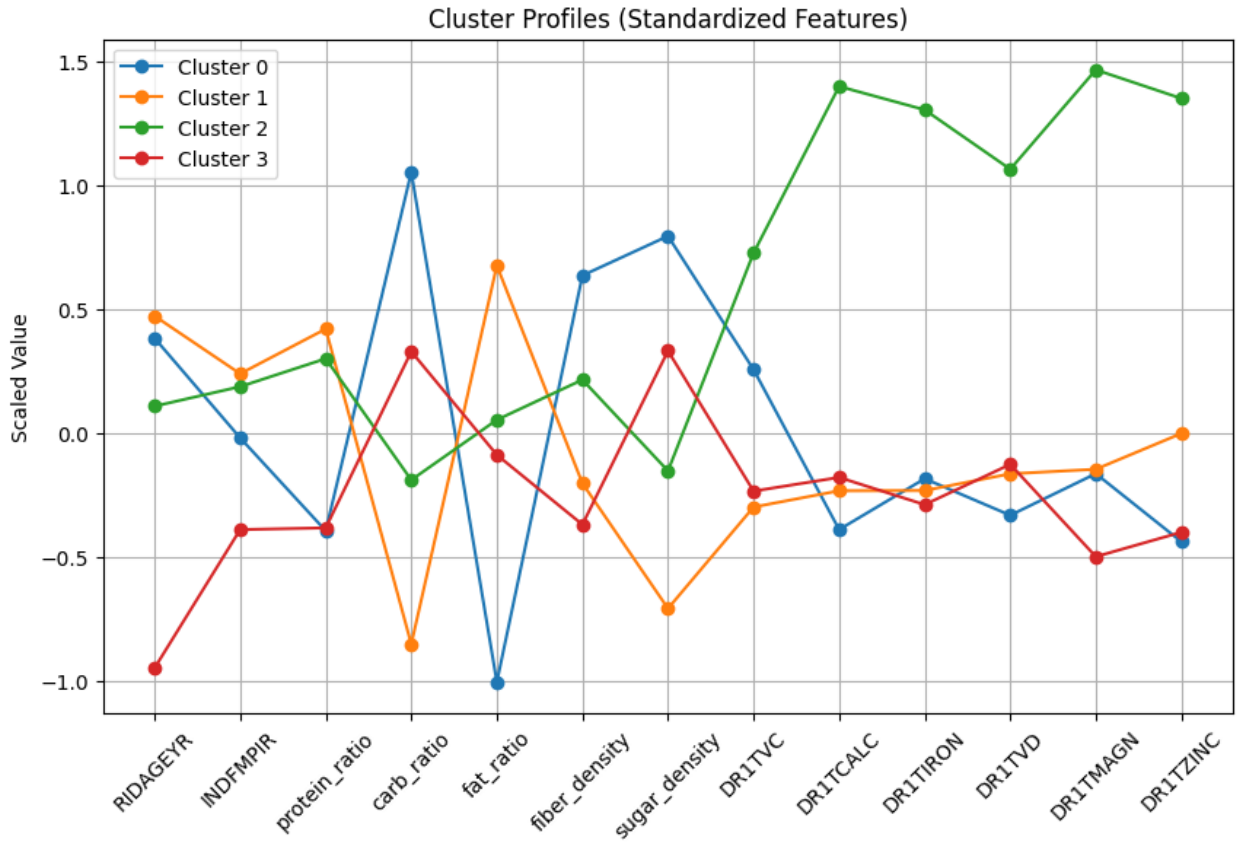
```
# make a scaled dataframe for plotting / profiling
df_scaled = pd.DataFrame(X_scaled, columns=cluster_features, index=df.index)
df_scaled["cluster"] = df["cluster"]
```

```
# Cluster nutrients profile
cluster_profile = df_scaled.groupby("cluster")[cluster_features].mean()
cluster_profile
```

cluster	RIDAGEYR	INDFMPIR	protein_ratio	carb_ratio	fat_ratio	fiber_density	sugar_density	I
0	0.384693	-0.017500	-0.392321	1.054170	-1.002665	0.637535	0.796654	0
1	0.474204	0.241188	0.422009	-0.849780	0.677642	-0.200733	-0.705825	-
2	0.110284	0.189116	0.302131	-0.187346	0.055221	0.216916	-0.150653	0
3	-0.945954	-0.387609	-0.380621	0.330761	-0.086081	-0.367198	0.333541	-

```
import matplotlib.pyplot as plt
import seaborn as sns

# Line plot
plt.figure(figsize=(10, 6))

for cluster in cluster_profile.index:
    plt.plot(
        cluster_features,
        cluster_profile.loc[cluster],
        marker='o',
        label=f"Cluster {int(cluster)}"
    )
```

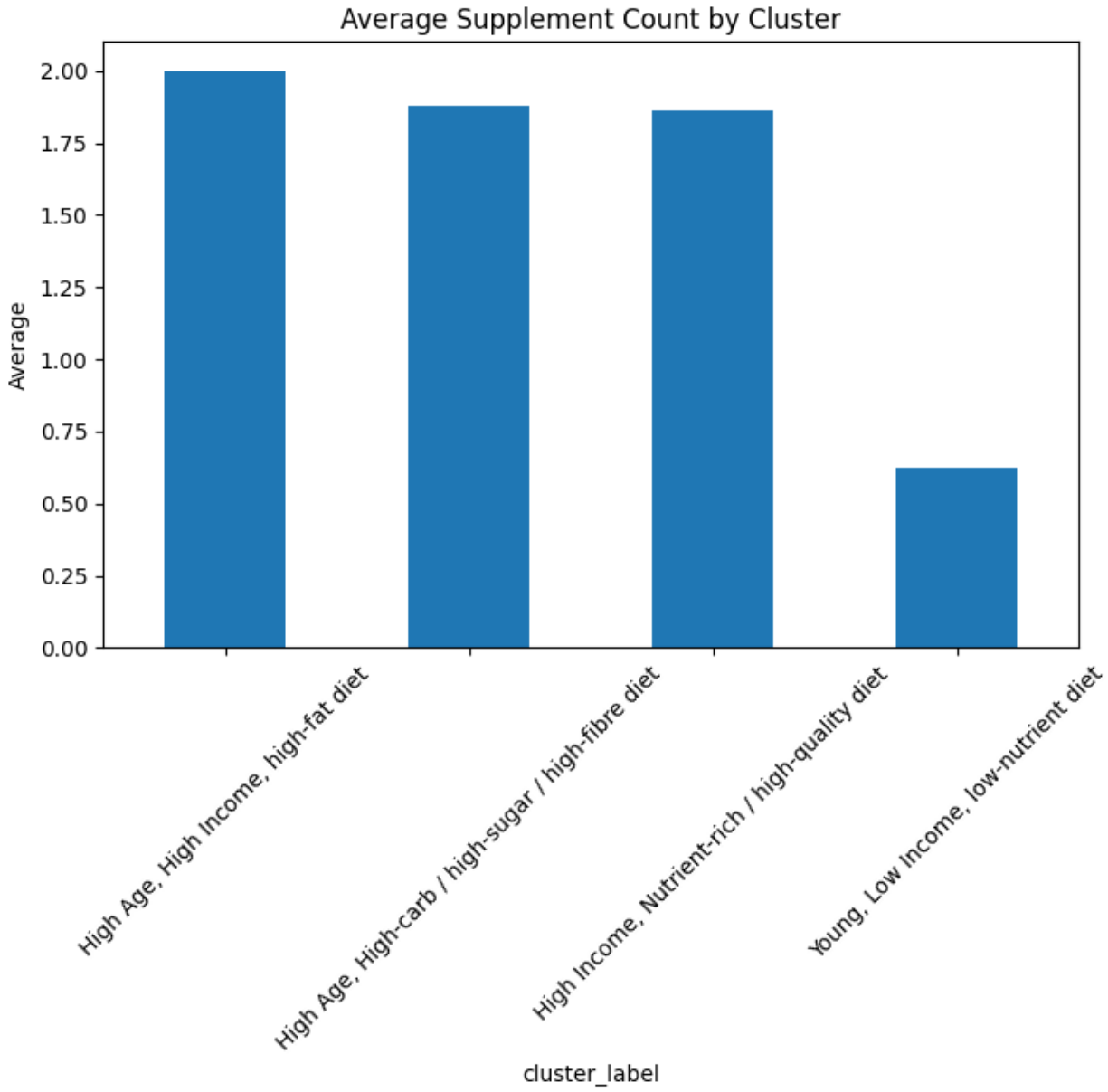
```
plt.xticks(rotation=45)
plt.ylabel("Scaled Value")
plt.title("Cluster Profiles (Standardized Features)")
plt.legend()
plt.grid(True)
plt.show()
```




```
# Label the clusters according to feature
cluster_names = {
    0: "High Age, High-carb / high-sugar / high-fibre diet",
    1: "High Age, High Income, high-fat diet",
    2: "High Income, Nutrient-rich / high-quality diet",
    3: "Young, Low Income, low-nutrient diet"
}

df_scaled["cluster_label"] = df_scaled["cluster"].map(cluster_names)
df["cluster_label"] = df["cluster"].map(cluster_names)
```

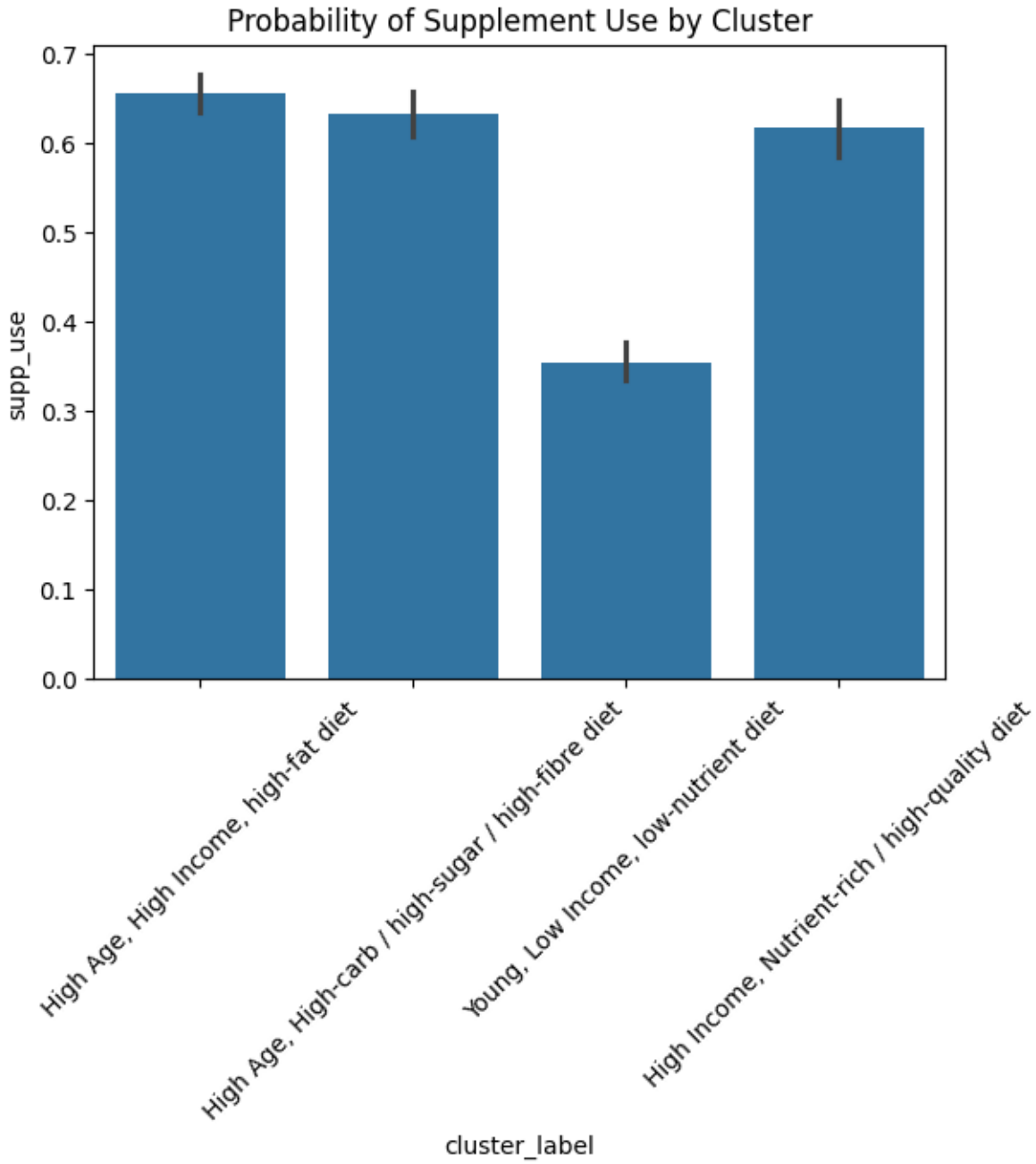
```
cluster_behavior = df.groupby("cluster_label")["supp_count"].mean()
```

```
# Supplement behavior plot
cluster_behavior.plot(kind="bar", figsize=(8, 5))
plt.title("Average Supplement Count by Cluster")
plt.ylabel("Average")
plt.xticks(rotation=45)
plt.show()
```



Supplement use is less common among individuals with both lower socioeconomic status and poorer overall nutrient intake. In contrast, older or higher-income groups tend to use more supplements on average, regardless of whether their diet is high-fat, high-sugar, or nutrient-rich. This reinforces the idea that demographic and socioeconomic factors may be at least as important as diet quality in explaining supplement behavior.

```
# Probability of using supplement by cluster
sns.barplot(x="cluster_label", y="supp_use", data=df)
plt.title("Probability of Supplement Use by Cluster")
plt.xticks(rotation=45)
plt.show()
```



The probability of supplement use is highest in the older, higher-income clusters and lowest in the young, low-income, low-nutrient cluster. This result suggests that supplement use is not driven only by nutritional deficiency. In fact, people with relatively better diets and higher socioeconomic

status may still be more likely to use supplements, possibly because supplement use reflects health awareness, purchasing power, or preventive health behavior. Meanwhile, the group with lower nutrient intake is actually less likely to use supplements, which is an important finding for interpreting the relationship between diet quality and supplement behavior.

Part 4: Modeling and Evaluation

We first use logistic regression to predict whether an individual uses dietary supplements. This is an appropriate baseline model because the target variable, supp_use, is binary, indicating whether a participant uses supplements or not.

```
# Define the binary target variable:
# 1 = supplement user, 0 = non-user
target_col = "supp_use"

demo_core_features = [
    "RIDAGEYR",     # age
    "RIAGENDR",     # gender
    "DMDEDUC2",     # education
    "INDFMPIR",     # family poverty-income ratio
    "DMDMARTZ"      # marital status
]

# Additional demographic feature that is not included in past cycle dataset,
# therefore used only in the expanded model
demo_expanded_only = [
    "DMDHHSIZ"      # household size
]

diet_raw_features = [
    "DR1TKCAL",   # total calories
    "DR1TPROT",   # total protein
    "DR1TCARB",   # total carbohydrates
    "DR1TTFAT",   # total fat
    "DR1TSUGR",   # total sugars
    "DR1TFIBE",   # total fiber
    "DR1TVC",     # vitamin C
    "DR1TVD",     # vitamin D
    "DR1TCALC",   # calcium
    "DR1TIRON",   # iron
    "DR1TMAGN",   # magnesium
    "DR1TZINC",   # zinc
    "DR1TNUMF"    # number of foods and beverages consumed
]

core_features = (
    demo_core_features
```

```
    + diet_raw_features
)

expanded_features = core_features + demo_expanded_only

# Remove duplicates while preserving order,
# in case any variable accidentally appears more than once.
core_features = list(dict.fromkeys(core_features))
expanded_features = list(dict.fromkeys(expanded_features))
```

```
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.metrics import (
    accuracy_score, balanced_accuracy_score, precision_score, recall_score,
    f1_score, roc_auc_score, confusion_matrix, classification_report
)

def split_numeric_categorical(df, feature_list):
    """
    Separate features into numeric and categorical columns.

    This is needed because numeric and categorical variables require
    different preprocessing steps before fitting logistic regression.
    """
    numeric_cols = []
    categorical_cols = []

    for col in feature_list:
        if pd.api.types.is_numeric_dtype(df[col]):
            numeric_cols.append(col)
        else:
            categorical_cols.append(col)

    return numeric_cols, categorical_cols
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import OneHotEncoder

def run_logistic_model(df, feature_list, target_col, test_size=0.2,
random_state=42):
    """
    Fit a logistic regression model for supplement use prediction.

    Steps:
    1. Select predictors and target
    2. Split predictors into numeric and categorical groups
```

```
    3. Preprocess each type appropriately
    4. Train/test split
    5. Fit logistic regression
    6. Evaluate performance using several classification metrics
    """
    X = df[feature_list].copy()
    y = df[target_col].copy()

    numeric_cols, categorical_cols = split_numeric_categorical(df, feature_list)

    preprocessor = ColumnTransformer(
        transformers=[
            ("num", Pipeline([
                # fill missing values with the median
                ("imputer", SimpleImputer(strategy="median")),
                # standardize to mean 0 and variance 1
                ("scaler", StandardScaler())
            ]), numeric_cols),
            ("cat", Pipeline([
                # fill missing values with the most frequent category
                ("imputer", SimpleImputer(strategy="most_frequent")),
                # convert categories into dummy variables
                ("onehot", OneHotEncoder(drop="first", handle_unknown="ignore"))
            ]), categorical_cols)
        ]
    )

    # Build the full modeling pipeline:
    # preprocessing first, then logistic regression.
    model = Pipeline([
        ("preprocessor", preprocessor),
        # help the model converge, since many predictors are included
        ("classifier", LogisticRegression(max_iter=2000,
class_weight="balanced"))
    ])

    X_train, X_test, y_train, y_test = train_test_split(
        X, y,
        test_size=test_size,
        random_state=random_state,
        stratify=y
    )

    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)
    y_prob = model.predict_proba(X_test)[:, 1]
```

```
    results = {
        "model": model,
        "X_train": X_train,
        "X_test": X_test,
        "y_train": y_train,
        "y_test": y_test,
        "y_pred": y_pred,
        "y_prob": y_prob,
        "metrics": {
            "accuracy": accuracy_score(y_test, y_pred),
            "balanced_accuracy": balanced_accuracy_score(y_test, y_pred),
            "precision": precision_score(y_test, y_pred, zero_division=0),
            "recall": recall_score(y_test, y_pred, zero_division=0),
            "f1": f1_score(y_test, y_pred, zero_division=0),
            "roc_auc": roc_auc_score(y_test, y_prob)
        },
        "confusion_matrix": confusion_matrix(y_test, y_pred),
        "classification_report": classification_report(y_test, y_pred,
        zero_division=0)
    }

    return results
```

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split

def run_tree_model(df, feature_list, target_col, test_size=0.2, random_state=42):
    """
    Fit a decision tree classifier for supplement use prediction.

    Compared with logistic regression, decision trees can capture
    nonlinear relationships and interactions more flexibly.
    """
    X = df[feature_list].copy()
    y = df[target_col].copy()

    numeric_cols, categorical_cols = split_numeric_categorical(df, feature_list)

    preprocessor = ColumnTransformer(
        transformers=[
            ("num", Pipeline([
                ("imputer", SimpleImputer(strategy="median"))
            ]), numeric_cols),
            ("cat", Pipeline([
                ("imputer", SimpleImputer(strategy="most_frequent")),
                ("onehot", OneHotEncoder(drop="first", handle_unknown="ignore"))
            ]), categorical_cols)
```

```
    ]
)

# Build the full pipeline:
# preprocessing first, then decision tree classification
model = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier", DecisionTreeClassifier(
        max_depth=4, # Limit the depth of the tree to reduce overfitting
        # Require at least 20 observations in each terminal node
        # so the tree does not create overly specific splits
        min_samples_leaf=20,
        # Give more balanced weight to both classes during training
        class_weight="balanced",
        random_state=random_state # Ensure reproducible results
    ))
])

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=test_size,
    random_state=random_state,
    stratify=y
)

model.fit(X_train, y_train)

y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1]

results = {
    "model": model,
    "X_train": X_train,
    "X_test": X_test,
    "y_train": y_train,
    "y_test": y_test,
    "y_pred": y_pred,
    "y_prob": y_prob,
    "metrics": {
        "accuracy": accuracy_score(y_test, y_pred),
        "balanced_accuracy": balanced_accuracy_score(y_test, y_pred),
        "precision": precision_score(y_test, y_pred, zero_division=0),
        "recall": recall_score(y_test, y_pred, zero_division=0),
        "f1": f1_score(y_test, y_pred, zero_division=0),
        "roc_auc": roc_auc_score(y_test, y_prob)
    },
    "confusion_matrix": confusion_matrix(y_test, y_pred),
    "classification_report": classification_report(y_test, y_pred,
    zero_division=0)
```

```
    }

    return results
```

```
from sklearn.dummy import DummyClassifier

def run_dummy_model(df, feature_list, target_col, test_size=0.2,
random_state=42):
    """
    Fit a dummy baseline classifier.

    This model ignores the predictors and simply predicts the most
    frequent class observed in the training data.
    """
    X = df[feature_list].copy()
    y = df[target_col].copy()

    numeric_cols, categorical_cols = split_numeric_categorical(df, feature_list)

    # Apply the same preprocessing structure as in the other models
    # so that model comparisons remain consistent.
    preprocessor = ColumnTransformer(
        transformers=[
            ("num", Pipeline([
                ("imputer", SimpleImputer(strategy="median"))
            ]), numeric_cols),
            ("cat", Pipeline([
                ("imputer", SimpleImputer(strategy="most_frequent")),
                ("onehot", OneHotEncoder(drop="first", handle_unknown="ignore"))
            ]), categorical_cols)
        ]
    )

    model = Pipeline([
        ("preprocessor", preprocessor),
        ("classifier", DummyClassifier(strategy="most_frequent"))
    ])

    X_train, X_test, y_train, y_test = train_test_split(
        X, y,
        test_size=test_size,
        random_state=random_state,
        stratify=y
    )

    model.fit(X_train, y_train)
```

```
    y_pred = model.predict(X_test)

    y_prob = model.predict_proba(X_test)[:, 1]

    results = {
        "model": model,
        "X_train": X_train,
        "X_test": X_test,
        "y_train": y_train,
        "y_test": y_test,
        "y_pred": y_pred,
        "y_prob": y_prob,
        "metrics": {
            "accuracy": accuracy_score(y_test, y_pred),
            "balanced_accuracy": balanced_accuracy_score(y_test, y_pred),
            "precision": precision_score(y_test, y_pred, zero_division=0),
            "recall": recall_score(y_test, y_pred, zero_division=0),
            "f1": f1_score(y_test, y_pred, zero_division=0),
            "roc_auc": roc_auc_score(y_test, y_prob)
        },
        "confusion_matrix": confusion_matrix(y_test, y_pred),
        "classification_report": classification_report(y_test, y_pred,
        zero_division=0)
    }

    return results
```

```
# fit all three models using both core feature set and expanded feature set
logit_core_results = run_logistic_model(df, core_features, target_col)
logit_expanded_results = run_logistic_model(df, expanded_features, target_col)
tree_core_results = run_tree_model(df, core_features, target_col)
tree_expanded_results = run_tree_model(df, expanded_features, target_col)
dummy_core_results = run_dummy_model(df, core_features, target_col)
dummy_expanded_results = run_dummy_model(df, expanded_features, target_col)
```

```
# Combine the evaluation metrics from all models into a single comparison table.
comparison_df = pd.DataFrame([
    {"Feature_Set": "Core", "Model": "Dummy Baseline",
**dummy_core_results["metrics"]},
    {"Feature_Set": "Expanded", "Model": "Dummy Baseline",
**dummy_expanded_results["metrics"]},
    {"Feature_Set": "Core", "Model": "Logistic Regression",
**logit_core_results["metrics"]},
    {"Feature_Set": "Expanded", "Model": "Logistic Regression",
**logit_expanded_results["metrics"]},
    {"Feature_Set": "Core", "Model": "Decision Tree",
**tree_core_results["metrics"]},
```

```
    {"Feature_Set": "Expanded", "Model": "Decision Tree",
**tree_expanded_results["metrics"]},
])

comparison_df
```

	Feature_Set	Model	accuracy	balanced_accuracy	precision	recall	f1	roc_au
0	Core	Dummy Baseline	0.560902	0.500000	0.560902	1.000000	0.718690	0.5000
1	Expanded	Dummy Baseline	0.560902	0.500000	0.560902	1.000000	0.718690	0.5000
2	Core	Logistic Regression	0.694737	0.694044	0.741477	0.699732	0.720000	0.7466
3	Expanded	Logistic Regression	0.688722	0.688310	0.737143	0.691689	0.713693	0.7500
4	Core	Decision Tree	0.664662	0.668536	0.730769	0.636729	0.680516	0.7235
5	Expanded	Decision Tree	0.664662	0.668536	0.730769	0.636729	0.680516	0.7245

The dummy baseline achieves an accuracy of around 0.56 but a ROC AUC of 0.50, indicating the dataset is slightly unbalanced.

Logistic regression performs best overall, with accuracy and balanced accuracy both around 0.69 and a ROC AUC close to 0.75. This indicates strong and well-balanced classification performance.

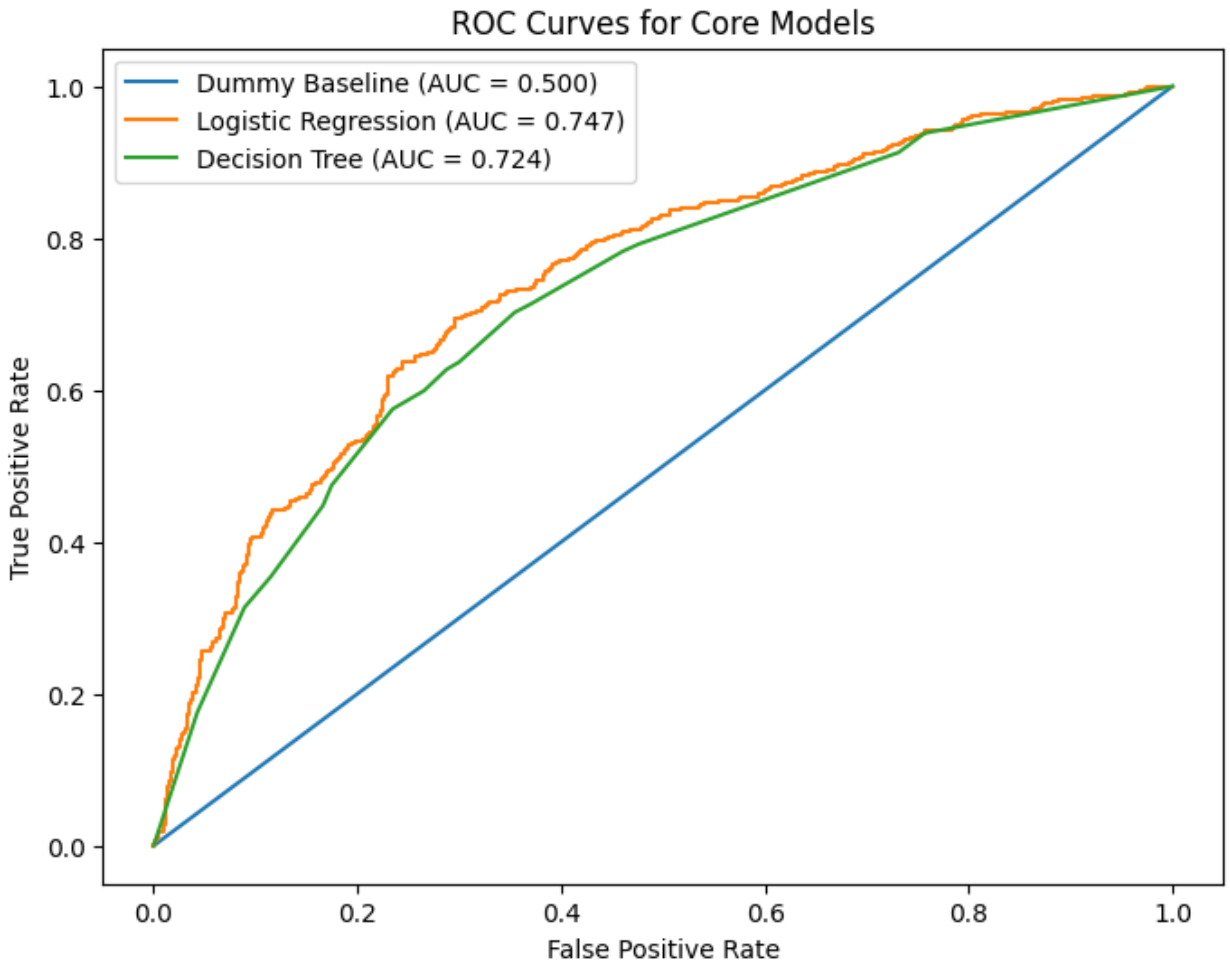
The decision tree performs slightly worse, with accuracy around 0.66 and ROC AUC around 0.72. While it captures some nonlinear patterns, it does not outperform logistic regression.

Expanding the feature set (by including household size) does not lead to meaningful improvement in model performance. In fact, logistic regression shows a slight decrease in accuracy (from 0.695 to 0.689), while the decision tree performance remains almost unchanged.

This suggests that the additional variable does not provide substantial predictive signal beyond the core demographic and dietary features. Therefore, the simpler core feature set is sufficient for this task.

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

plt.figure(figsize=(8, 6))

# Plot ROC curves for the core models only.
# ROC curves show how well each model separates supplement users
# from non-users across different classification thresholds.
for name, results in {
    "Dummy Baseline": dummy_core_results,
    "Logistic Regression": logit_core_results,
    "Decision Tree": tree_core_results
}.items():
    fpr, tpr, _ = roc_curve(results["y_test"], results["y_prob"])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, label=f"{name} (AUC = {roc_auc:.3f})")
```

```
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curves for Core Models")
plt.legend()
plt.show()
```

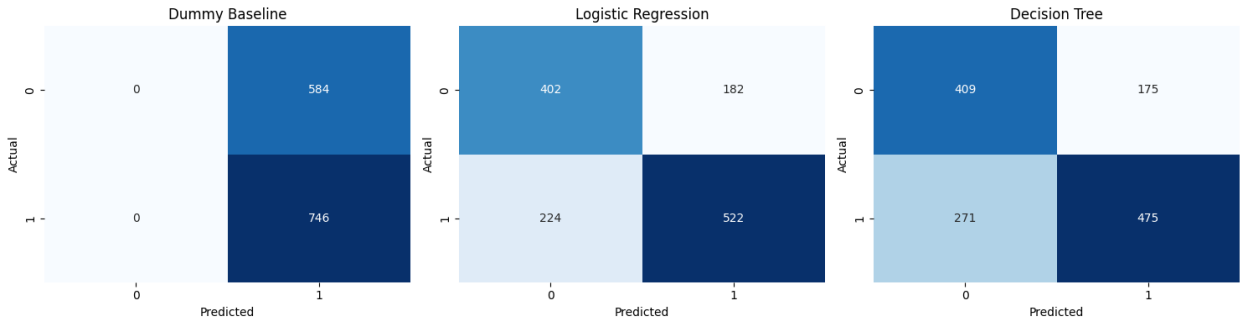


The ROC curve shows that logistic regression consistently outperforms the decision tree across most classification thresholds. Its curve lies above the decision tree and well above the diagonal baseline, resulting in the highest AUC (0.747).

Overall, logistic regression provides the best trade-off between true positive rate and false positive rate.

```
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Create side-by-side confusion matrices for the core models.
# This helps compare how each model classifies supplement users and non-users.
```

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

model_dict = {
    "Dummy Baseline": dummy_core_results,
    "Logistic Regression": logit_core_results,
    "Decision Tree": tree_core_results
}

for ax, (name, results) in zip(axes, model_dict.items()):
    cm = results["confusion_matrix"]
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False, ax=ax)
    ax.set_title(name)
    ax.set_xlabel("Predicted")
    ax.set_ylabel("Actual")

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



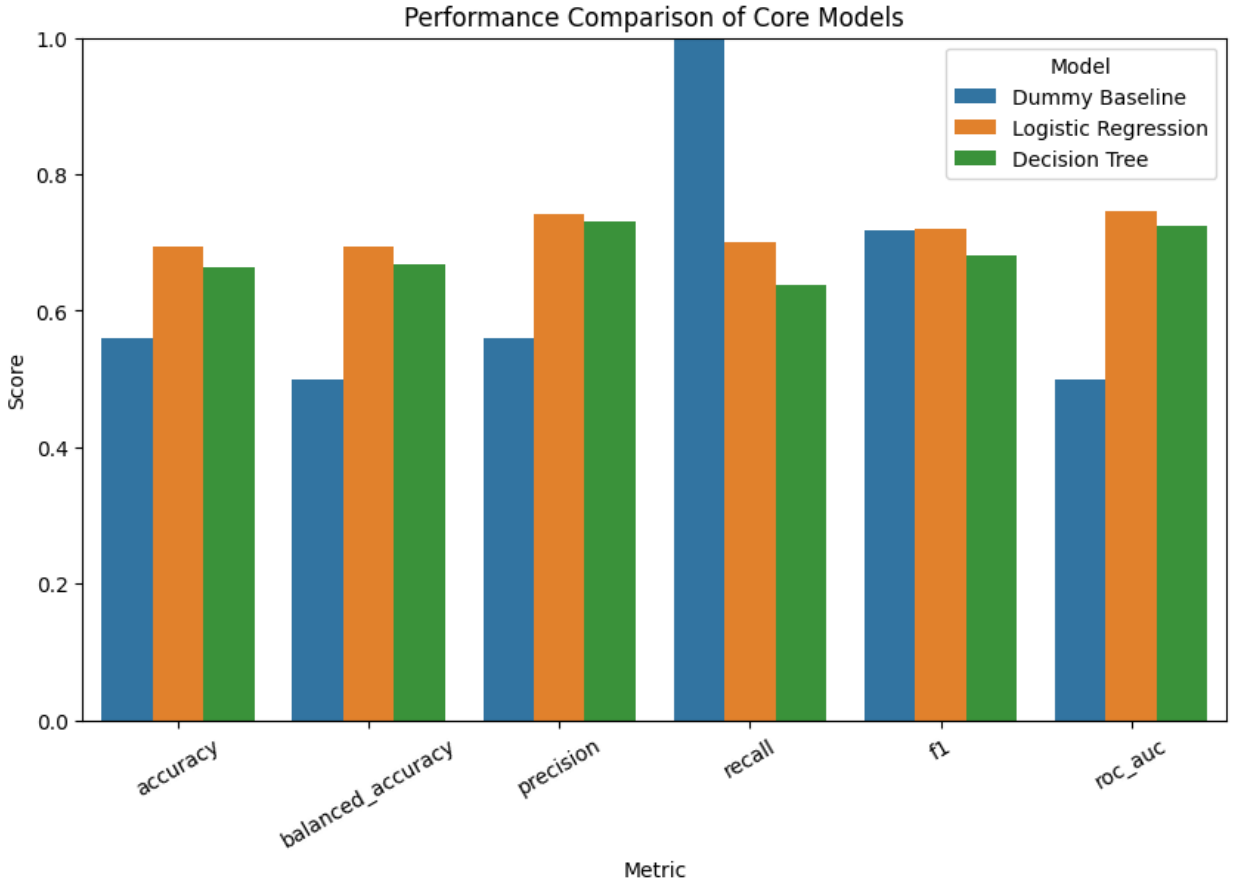
The dummy baseline predicts all observations as supplement users.

Logistic regression achieves a more balanced performance, produces fewer false positives and false negatives compared to the other models, indicating better overall classification quality.

The decision tree shows slightly worse performance, with more false negatives than logistic regression. This suggests it is less effective at identifying actual supplement users.

```
import pandas as pd

# Create a small summary table containing the evaluation metrics
# for the three core models only
viz_df = pd.DataFrame([
    {"Model": "Dummy Baseline", **dummy_core_results["metrics"]},
    {"Model": "Logistic Regression", **logit_core_results["metrics"]},
    {"Model": "Decision Tree", **tree_core_results["metrics"]}
])

# Reshape the table from wide format to long format
viz_df_long = viz_df.melt(id_vars="Model", var_name="Metric", value_name="Score")
```

```
plt.figure(figsize=(10, 6))
sns.barplot(data=viz_df_long, x="Metric", y="Score", hue="Model")
plt.title("Performance Comparison of Core Models")
plt.ylim(0, 1)
plt.xticks(rotation=30)
plt.show()
```

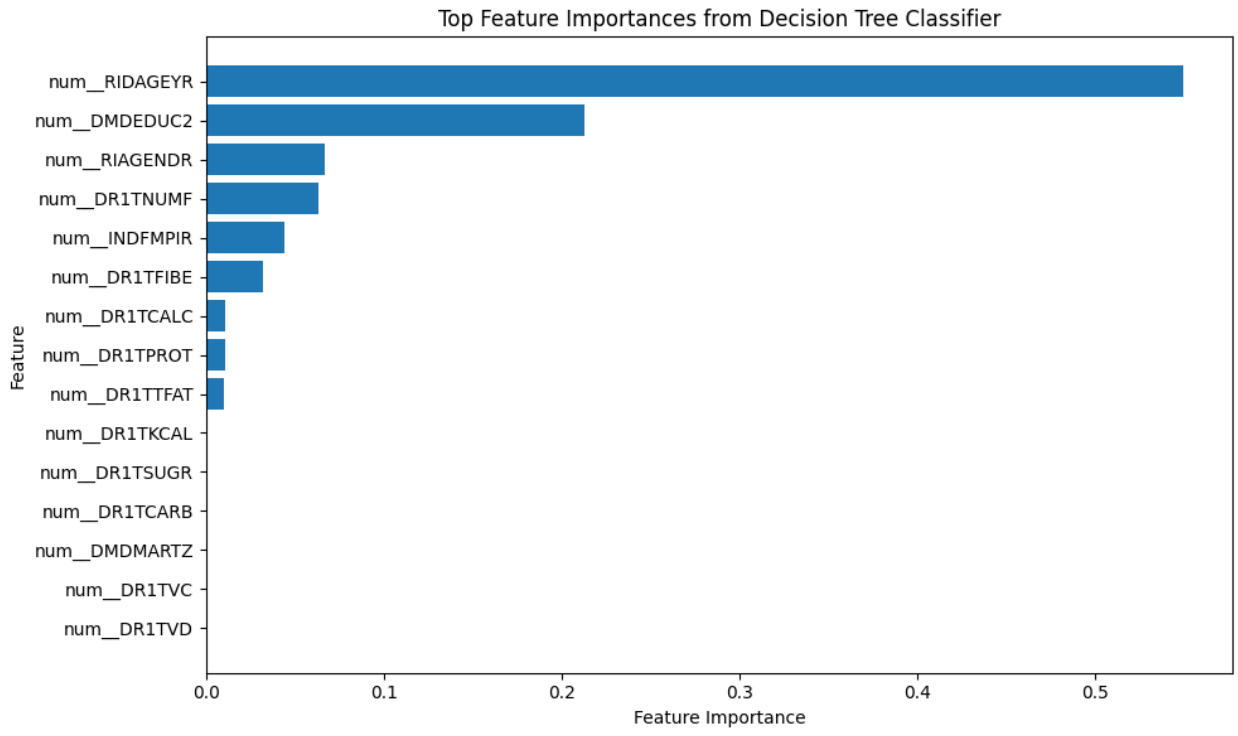


Logistic regression outperforms all other models across most evaluation metrics. It achieves the highest accuracy, balanced accuracy, F1-score, and ROC AUC, indicating the most consistent and reliable performance. The decision tree performs moderately well but is consistently slightly below logistic regression.

The dummy baseline performs poorly, despite having high recall. This confirms that relying on a single metric such as recall can be misleading.

```
# Get the fitted preprocessor and decision tree
preprocessor = tree_core_results["model"].named_steps["preprocessor"]
tree_model = tree_core_results["model"].named_steps["classifier"]

# Get feature names after preprocessing
```

```
feature_names = preprocessor.get_feature_names_out()

# Get feature importances from the decision tree
importances = tree_model.feature_importances_

# Create a dataframe for plotting
importance_df = pd.DataFrame({
    "feature": feature_names,
    "importance": importances
})

# Sort by importance
importance_df = importance_df.sort_values("importance", ascending=False)

# Keep top 15 most important features for readability
top_n = 15
top_features = importance_df.head(top_n)

# Plot
plt.figure(figsize=(10, 6))
plt.barh(top_features["feature"][::-1], top_features["importance"][::-1])
plt.xlabel("Feature Importance")
plt.ylabel("Feature")
plt.title("Top Feature Importances from Decision Tree Classifier")
plt.tight_layout()
plt.show()
```



The feature importance plot shows that age (RIDAGEYR) is by far the most influential predictor of supplement use, with a significantly higher importance than all other variables.

Education level (DMDEDUC2) is the second most important feature, followed by gender (RIAGENDR) and the number of foods consumed (DR1TNUMF). In contrast, most dietary intake variables, such as calories, fat, and sugar, have very low importance.

External Validation on a Previous NHANES Cycle

To evaluate whether the models generalize beyond the 2021–2023 cycle, we test the trained core models on data from an earlier NHANES cycle (2017–March 2020 pre-pandemic). This serves as an external validation step.

Unlike the original train-test split, this evaluation checks whether the patterns learned from the more recent data remain useful when applied to a different time period. If performance remains similar, it suggests that the model captures stable relationships rather than overfitting to one specific cycle.

```
# Use the logistic regression model trained on the main dataset
# to make predictions on the previous cycle data
X_prev_core = df_prev_before_impute[core_features].copy()
y_prev = df_prev_before_impute[target_col].copy()

prev_pred_logit = logit_core_results["model"].predict(X_prev_core)
# Predict probabilities for supplement use = 1
prev_prob_logit = logit_core_results["model"].predict_proba(X_prev_core)[:, 1]

from sklearn.metrics import accuracy_score, balanced_accuracy_score,
precision_score, recall_score, f1_score, roc_auc_score

logit_external_metrics = {
    "accuracy": accuracy_score(y_prev, prev_pred_logit),
    "balanced_accuracy": balanced_accuracy_score(y_prev, prev_pred_logit),
    "precision": precision_score(y_prev, prev_pred_logit, zero_division=0),
    "recall": recall_score(y_prev, prev_pred_logit, zero_division=0),
    "f1": f1_score(y_prev, prev_pred_logit, zero_division=0),
    "roc_auc": roc_auc_score(y_prev, prev_prob_logit)
}
```

```
# Use the decision tree model trained on the main dataset
# to make predictions on the previous cycle data
prev_pred_tree = tree_core_results["model"].predict(X_prev_core)
prev_prob_tree = tree_core_results["model"].predict_proba(X_prev_core)[:, 1]

tree_external_metrics = {
    "accuracy": accuracy_score(y_prev, prev_pred_tree),
    "balanced_accuracy": balanced_accuracy_score(y_prev, prev_pred_tree),
    "precision": precision_score(y_prev, prev_pred_tree, zero_division=0),
    "recall": recall_score(y_prev, prev_pred_tree, zero_division=0),
```

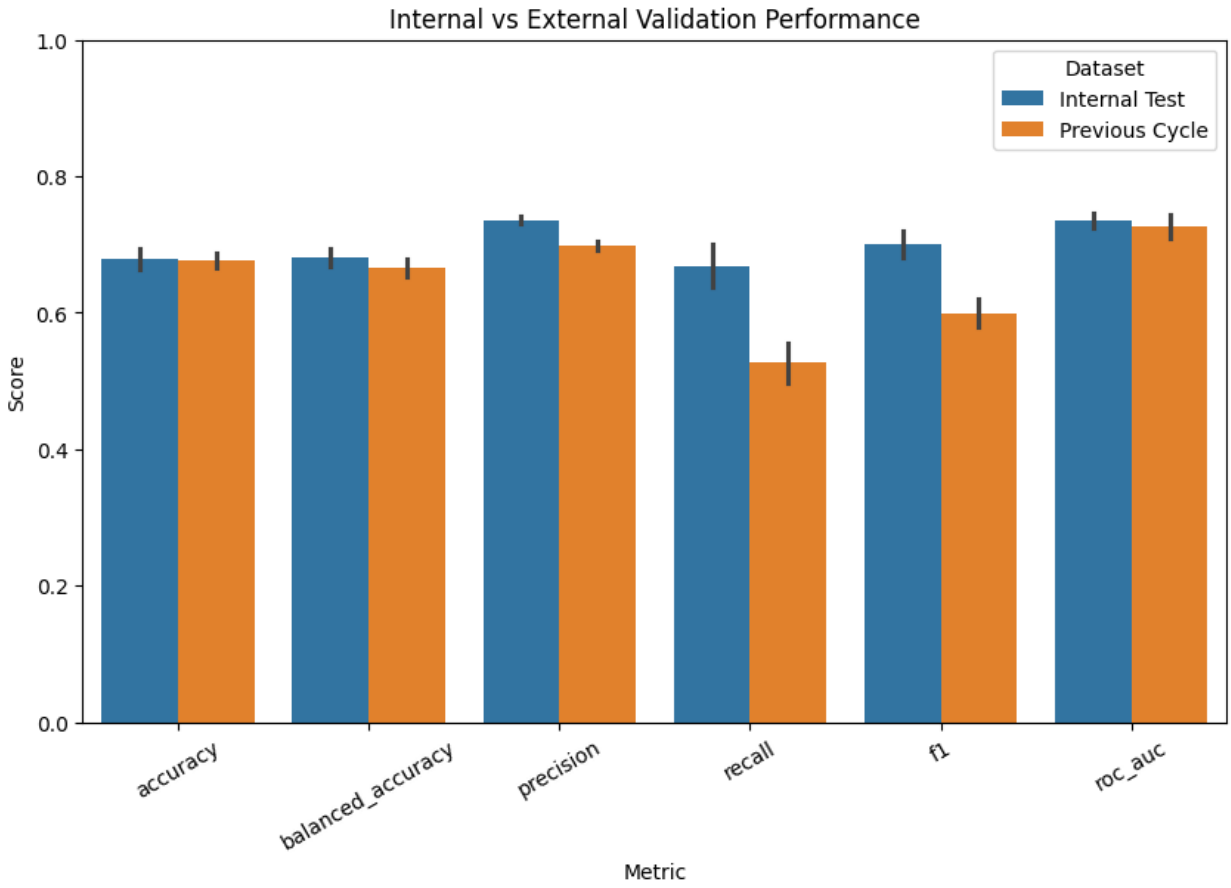
```
    "f1": f1_score(y_prev, prev_pred_tree, zero_division=0),
    "roc_auc": roc_auc_score(y_prev, prev_prob_tree)
}
```

```
external_df = pd.DataFrame([
    {"Model": "Logistic Regression", "Dataset": "Previous cycle",
**logit_external_metrics},
    {"Model": "Decision Tree", "Dataset": "Previous cycle",
**tree_external_metrics},
])

external_df
```

	Model	Dataset	accuracy	balanced_accuracy	precision	recall	f1	roc_
0	Logistic Regression	Previous cycle	0.687581	0.677867	0.704100	0.554775	0.620581	0.744
1	Decision Tree	Previous cycle	0.666235	0.653923	0.691033	0.497893	0.578776	0.708

The external validation results show that logistic regression still performs better than the decision tree on the previous NHANES cycle. Its accuracy, balanced accuracy, F1-score, and ROC AUC are all slightly higher, suggesting better generalization across time.

```
external_summary = pd.DataFrame([
    {"Model": "Logistic Regression", "Dataset": "Internal Test",
**logit_core_results["metrics"]},
    {"Model": "Decision Tree", "Dataset": "Internal Test",
**tree_core_results["metrics"]},
    {"Model": "Logistic Regression", "Dataset": "Previous Cycle",
**logit_external_metrics},
    {"Model": "Decision Tree", "Dataset": "Previous Cycle",
**tree_external_metrics},
])

ext_long = external_summary.melt(
    id_vars=["Model", "Dataset"],
    var_name="Metric",
    value_name="Score"
)

plt.figure(figsize=(10, 6))
sns.barplot(data=ext_long, x="Metric", y="Score", hue="Dataset")
plt.title("Internal vs External Validation Performance")
plt.ylim(0, 1)
plt.xticks(rotation=30)
plt.show()
```



This chart shows that model performance is fairly similar between the internal test set and the previous cycle. The small drop in some metrics suggests a modest loss in performance, but overall the logistic regression model remains stable and generalizes reasonably well.

Multi-class classification

```
# Select features
features = [
    "RIDAGEYR",
    "INDFMPIR",
    "DMDHHSIZ",
    "protein_ratio",
    "carb_ratio",
    "fat_ratio",
    "fiber_density",
    "sugar_density",
    "DR1TVC",
    "DR1TCALC",
    "DR1TIRON",
    "DR1TVD",
    "DR1TMAGN",
    "DR1TZINC"
```

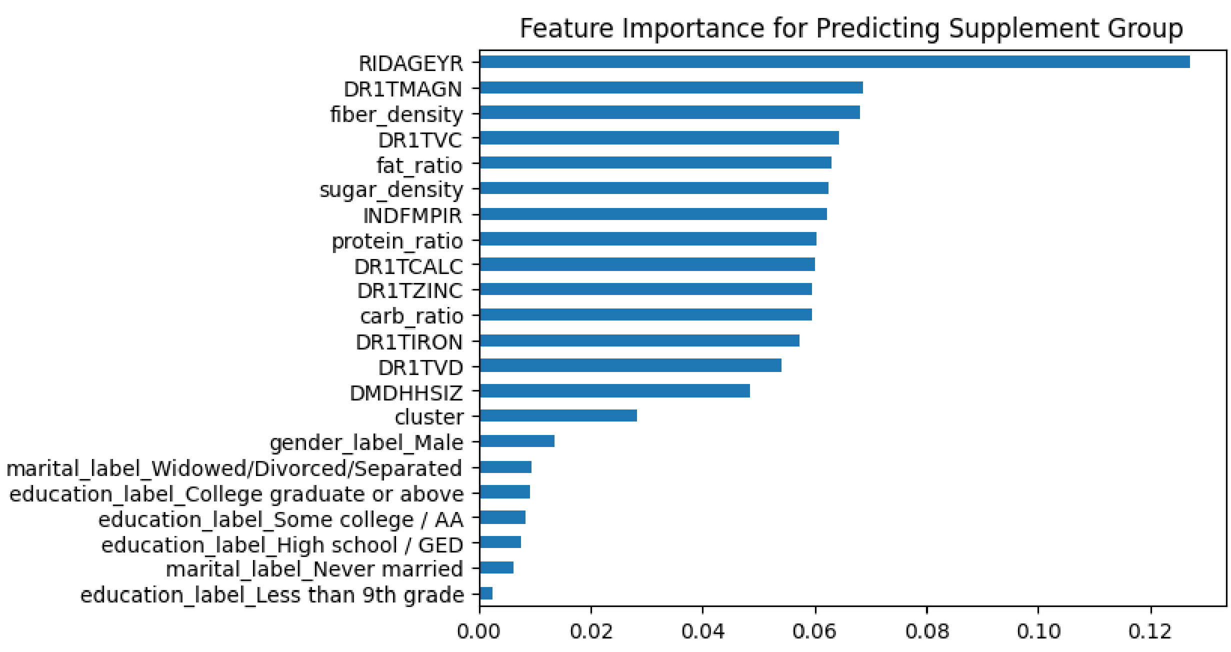
```
]

categorical_cols = [
    "gender_label",
    "education_label",
    "marital_label"
]

# Encode categorical features
df_encoded = pd.get_dummies(df, columns=categorical_cols, drop_first=True)
features += [col for col in df_encoded.columns if any(x in col for x in
categorical_cols)]

features.append("cluster")
```

```
# Train model (Random Forest)
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

X = df_encoded[features]
y = df_encoded.loc[X.index, "supp_usage_count"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Fit 500 decision trees with balanced class weight
model = RandomForestClassifier(n_estimators=500, random_state=1,
class_weight="balanced")
model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

         1-5       0.61      0.74      0.67       677
          5+       0.00      0.00      0.00        82
        none       0.65      0.58      0.61       571

    accuracy                           0.63      1330
   macro avg       0.42      0.44      0.43      1330
weighted avg       0.59      0.63      0.60      1330
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:1565:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels
with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:1565:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels
with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
/usr/local/lib/python3.12/dist-packages/sklearn/metrics/_classification.py:1565:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels
with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
importance = pd.Series(model.feature_importances_, index=features)
importance.sort_values().plot(kind="barh")
plt.title("Feature Importance for Predicting Supplement Group")
plt.show()
```




```
colab2pdf()
```