50609491_DIC_Phase2

November 5, 2024

Data cleaning

[102]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from scipy.stats import pearsonr

df = pd.read_csv("user_nutritional_data.csv")
df.info()
#Identifing missing values by defining potential representations of missing␣
↪data and used df.isin() to check for their presence.
missing_values = ["NA", np.nan, " ", None]
missing= df.isin(missing_values)
missing.head()

#Removed duplicate rows from the DataFrame using df.drop_duplicates() and␣
↪updated the dataset.
duplicate_rows = df[df.duplicated()]
len(duplicate_rows)

df = df.drop_duplicates()

#Added new columns, BMI using the formula: BMI = Weight / (Height in meters)^2.
df["BMI"] = df["Weight"] / ((df["Height"]/100) ** 2)

#Replaced numeric values in the 'Physical exercise' column with descriptive␣
↪labels for better interpretability.
df['Physical exercise'] = df['Physical exercise'].replace({
    0: 'None',
    1: 'Low',
    2: 'Moderate',
    3: 'High',
    4: 'Very High'
})
```

```python
#Binned the 'Age' column into specific age groups using pd.cut() to categorize␣
↪ages into defined ranges.
bins = [0, 18, 30, 45, 60, 100]
labels = ['0-18', '19-30', '31-45', '46-60', '60+']
df['Age Group'] = pd.cut(df['Age'], bins=bins, labels=labels, right=False)

df.info()
df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2182 entries, 0 to 2181
Data columns (total 11 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Gender                 2182 non-null   int64
 1   Age                    2182 non-null   int64
 2   Daily meals frequency  2182 non-null   int64
 3   Physical exercise      2182 non-null   int64
 4   Height                 2182 non-null   int64
 5   Weight                 2182 non-null   float64
 6   BMR                    2182 non-null   float64
 7   Carbs                  2182 non-null   float64
 8   Proteins               2182 non-null   float64
 9   Fats                   2182 non-null   float64
 10  Calories               2182 non-null   float64
dtypes: float64(6), int64(5)
memory usage: 187.6 KB
<class 'pandas.core.frame.DataFrame'>
Index: 2098 entries, 0 to 2181
Data columns (total 13 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Gender                 2098 non-null   int64
 1   Age                    2098 non-null   int64
 2   Daily meals frequency  2098 non-null   int64
 3   Physical exercise      2098 non-null   object
 4   Height                 2098 non-null   int64
 5   Weight                 2098 non-null   float64
 6   BMR                    2098 non-null   float64
 7   Carbs                  2098 non-null   float64
 8   Proteins               2098 non-null   float64
 9   Fats                   2098 non-null   float64
 10  Calories               2098 non-null   float64
 11  BMI                    2098 non-null   float64
 12  Age Group              2098 non-null   category
dtypes: category(1), float64(7), int64(4), object(1)
memory usage: 215.3+ KB
```

[102]:

```
   Gender  Age  Daily meals frequency Physical exercise  Height  Weight  \
0       0   29                      3              None     165   101.0
1       1   25                      3         Very High     165    53.0
2       0   23                      2              None     170    70.0
3       0   22                      3              None     168   112.0
4       0   19                      3          Moderate     175    67.0

       BMR    Carbs  Proteins    Fats  Calories        BMI Age Group
0  1901.25  285.188   114.075  76.050  2281.502  37.098255     19-30
1  1275.25  302.872   121.149  80.766  2422.978  19.467401     19-30
2  1652.50  247.875    99.150  66.100  1983.000  24.221453     19-30
3  2065.00  309.750   123.900  82.600  2478.000  39.682540     19-30
4  1673.75  324.289   129.716  86.477  2594.313  21.877551     19-30
```

Hypothesis: Users with higher physical activity levels have distinct macronutrient needs compared to those with low physical activity levels.

Algorithms Applied: Support Vector Machine (SVM) and Linear Discriminant Analysis (LDA)

Source: Kevin Murphy, *Probabilistic Machine Learning: An Introduction*, MIT Press, 2022.

Support Vector Machine (SVM)

- Justification: SVM was chosen for its ability to create clear decision boundaries in high-dimensional spaces, making it suitable for distinguishing activity levels based on macronutrient needs.
- Tuning: A linear kernel was used to simplify relationships, providing optimal performance without complex boundaries.
- Effectiveness: SVM achieved an accuracy of 0.9285714285714286, with Truncated SVD visualization illustrating well-defined boundaries in nutrient patterns. This supported the hypothesis that macronutrient needs vary with physical activity levels.

Linear Discriminant Analysis (LDA)

- Justification: LDA was selected for its understandability and capacity to separate classes by finding linear combinations of features, ideal for differentiating "High" and "Low" activity levels.
- Tuning: Standardization was applied to enhance class separation and model stability across different data splits.
- Effectiveness: LDA's accuracy of 0.9253968253968254 confirmed effective class separation, as visualized by a confusion matrix. The model highlighted nutrient distribution patterns, particularly protein intake, that align with specific activity levels, aiding targeted dietary recommendations.

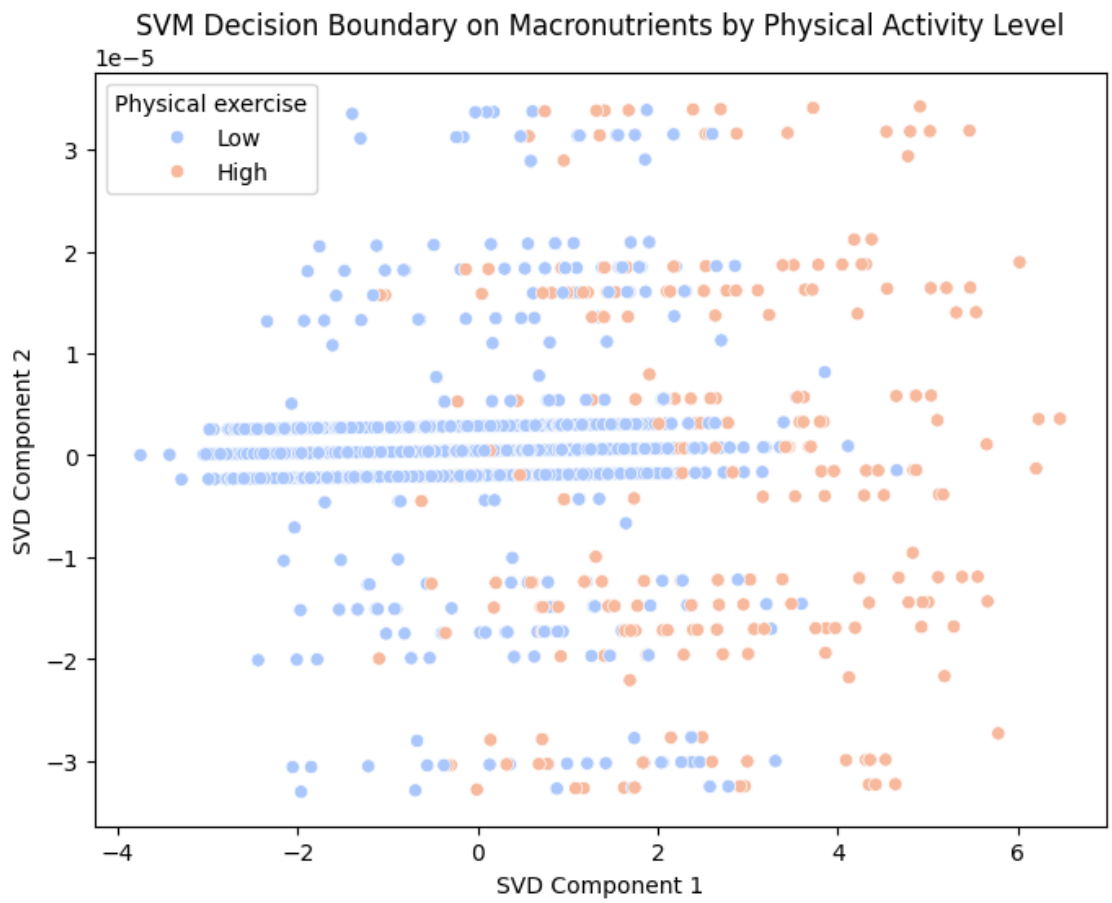
[103]:

```python
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.decomposition import TruncatedSVD
```

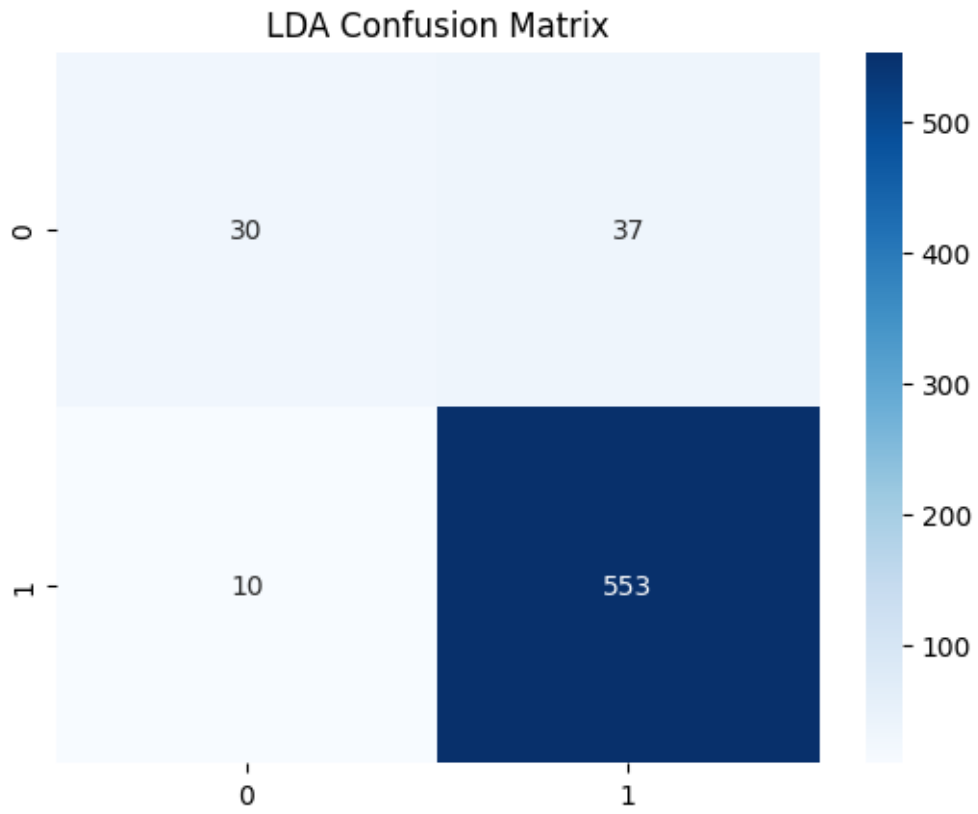
```python
X = df[['Carbs', 'Proteins', 'Fats']]
Y = df['Physical exercise'] = df['Physical exercise'].replace({
    'None': 0,
    'Low': 1,
    'Moderate': 2,
    'High': 3,
    'Very High': 4
})
Y = Y.apply(lambda x: 'High' if x > 1 else 'Low')
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_train, X_test, Y_train, Y_test = train_test_split(X_scaled, Y, test_size=0.3,␣
↪random_state=42)

# Support Vector Machine (SVM)
svm = SVC(kernel='linear')
svm.fit(X_train, Y_train)
Y_pred_svm = svm.predict(X_test)
print("SVM accuracy:", accuracy_score(Y_test, Y_pred_svm))
svd = TruncatedSVD(n_components=2)
X_svd = svd.fit_transform(X_scaled)
svm_model_svd = SVC(kernel='linear')
svm_model_svd.fit(X_svd, Y)
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_svd[:, 0], y=X_svd[:, 1], hue=Y, palette='coolwarm')
plt.title("SVM Decision Boundary on Macronutrients by Physical Activity Level")
plt.xlabel("SVD Component 1")
plt.ylabel("SVD Component 2")
plt.show()

# Linear Discriminant Analysis (LDA)
lda = LinearDiscriminantAnalysis()
lda.fit(X_train, Y_train)
Y_pred_lda = lda.predict(X_test)
cm = confusion_matrix(Y_test, Y_pred_lda)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
print("LDA accuracy:", accuracy_score(Y_test, Y_pred_lda))
plt.title("LDA Confusion Matrix")
plt.show()
```

```
SVM accuracy: 0.9285714285714286
```

```
<ipython-input-103-b494f95c00a5>:9: FutureWarning: Downcasting behavior in
`replace` is deprecated and will be removed in a future version. To retain the
old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to
the future behavior, set `pd.set_option('future.no_silent_downcasting', True)`
  Y = df['Physical exercise'] = df['Physical exercise'].replace({
```




```
LDA accuracy: 0.9253968253968254
```



Hypothesis: Users cluster into distinct dietary patterns based on macronutrient ratios (carbs, proteins, fats).

Algorithms Applied: Agglomerative Clustering and Mean Shift

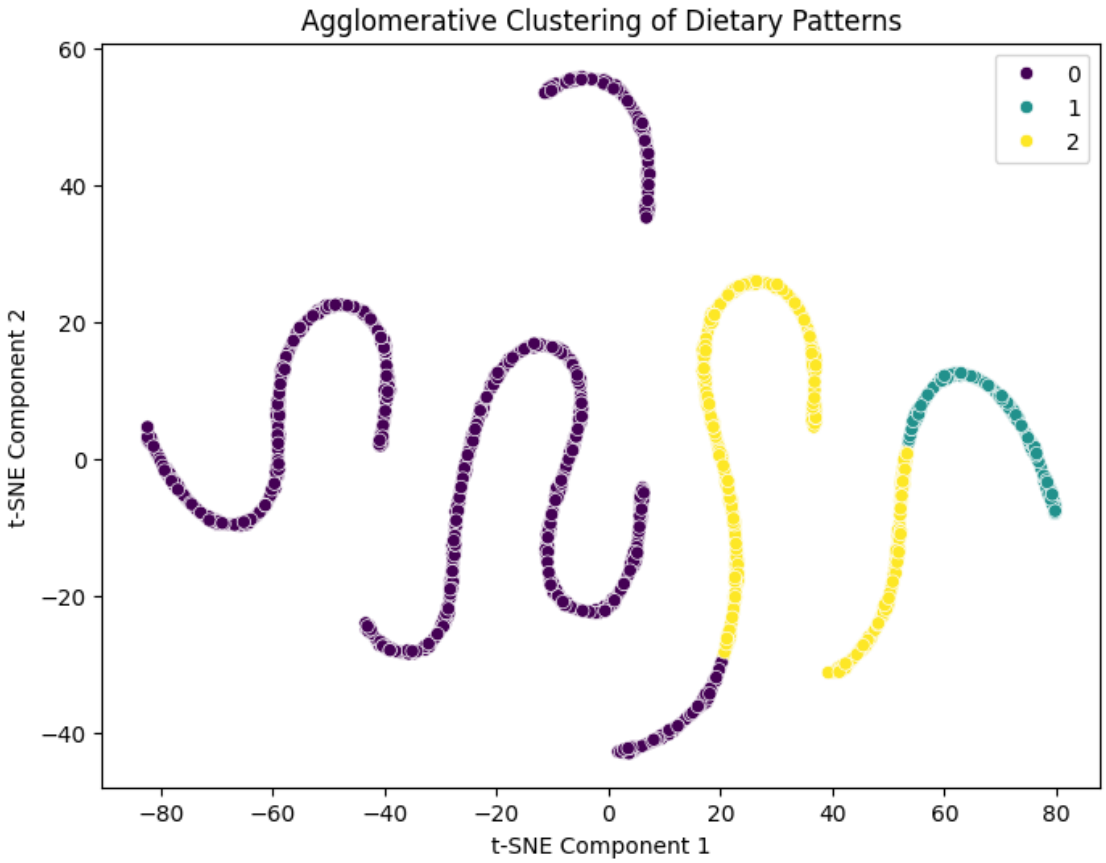
Agglomerative Clustering

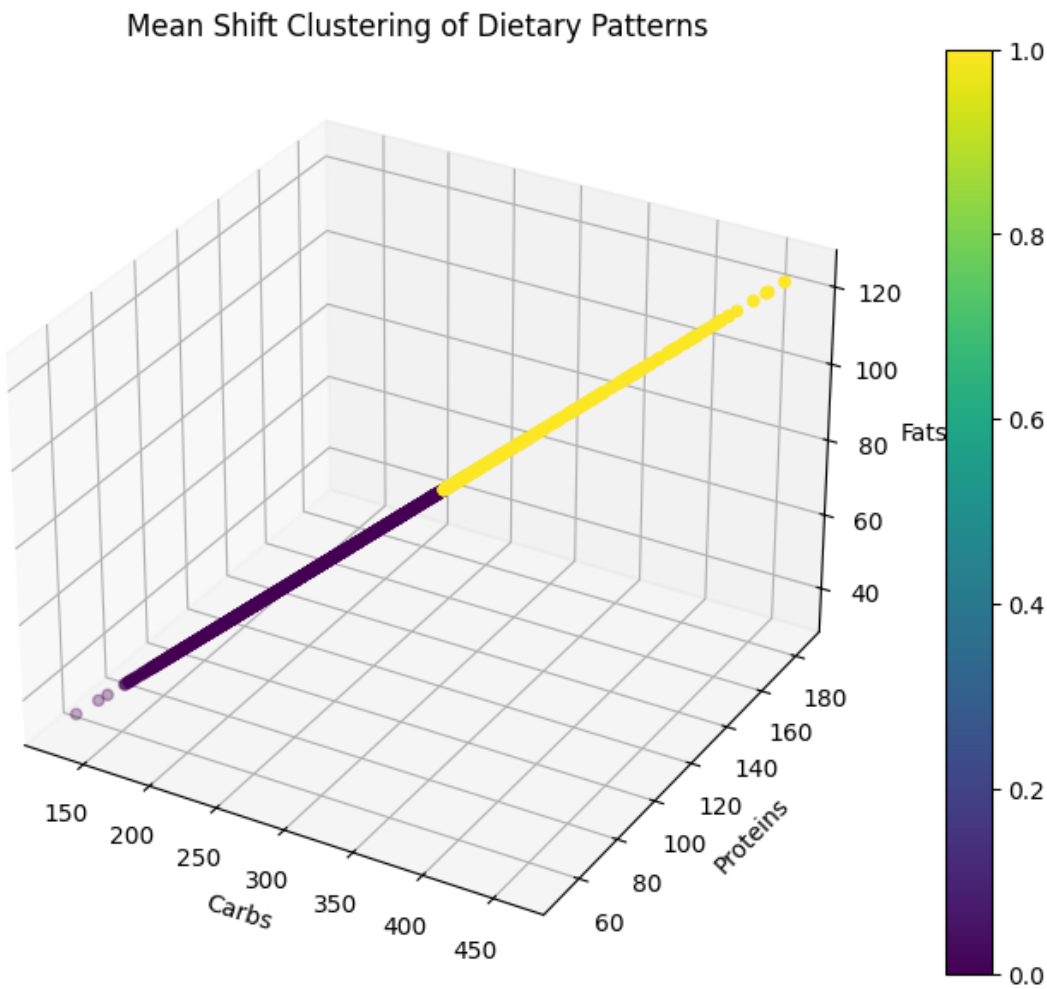
- Source: Kevin Murphy, *Probabilistic Machine Learning: An Introduction*, MIT Press, 2022.
- Justification: Agglomerative Clustering was selected for its hierarchical structure, allowing clear visualization of dietary patterns through a dendrogram. This flexibility allows exploratory analysis without needing a predefined number of clusters.
- Tuning: The number of clusters was set to three, which revealed distinct groupings based on nutrient profiles. Scaling was applied to macronutrients to ensure balanced clustering.
- Effectiveness: t-SNE visualization showed distinct clusters, supporting the hypothesis that users form dietary patterns based on nutrient ratios. Grouping users by higher carbs or proteins aligned well with specific dietary lifestyles, aiding in segmentation.

Mean Shift Clustering * Source: Comaniciu, D., & Meer, P., "Mean Shift: A Robust Approach Toward Feature Space Analysis," IEEE Transactions on Pattern Analysis and Machine Intelligence, 2002. * Justification: Mean Shift was chosen for its adaptability in discovering clusters without requiring a predefined number. This was ideal for identifying naturally occurring dietary groups. * Tuning: The bandwidth parameter was tuned to control cluster granularity, and different values were tested until well-defined clusters formed. * Effectiveness: Mean Shift identified clusters con-

sistent with macronutrient ratios, as shown in a 3D plot. It effectively highlighted dietary patterns with similar nutrient intakes, offering insights into various dietary lifestyles.

[104]:
```
from sklearn.cluster import AgglomerativeClustering, MeanShift
from sklearn.manifold import TSNE
import matplotlib.cm as cm

# Agglomerative Clustering
agglomerative_model = AgglomerativeClustering(n_clusters=3)
agglomerative_clusters = agglomerative_model.fit_predict(X_scaled)
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_scaled)
plt.figure(figsize=(8, 6))
sns.scatterplot(x=X_tsne[:, 0], y=X_tsne[:, 1], hue=agglomerative_clusters,␣
 ↪palette="viridis")
plt.title("Agglomerative Clustering of Dietary Patterns")
plt.xlabel("t-SNE Component 1")
plt.ylabel("t-SNE Component 2")
plt.show()

# Mean Shift Clustering
mean_shift_model = MeanShift()
mean_shift_clusters = mean_shift_model.fit_predict(X_scaled)
figure = plt.figure(figsize=(10, 7))
ax = figure.add_subplot(111, projection='3d')
scatter = ax.scatter(X['Carbs'], X['Proteins'], X['Fats'],
                     c=mean_shift_clusters, cmap=cm.viridis)
ax.set_xlabel("Carbs")
ax.set_ylabel("Proteins")
ax.set_zlabel("Fats")
plt.title("Mean Shift Clustering of Dietary Patterns")
plt.colorbar(scatter, ax=ax)
plt.show()
```


Agglomerative Clustering of Dietary Patterns
0
1
2
t-SNE Component 2
t-SNE Component 1
60
40
20
0
−20
−40
−80
−60
−40
−20
0
20
40
60
80



Hypothesis: Age and gender together influence the likelihood of a user exceeding daily recommended calorie intake.

Algorithms Applied: Naive Bayes and Decision Tree

Naive Bayes * Justification: Naive Bayes was chosen for its efficiency and assumption of feature independence, and it works well in binary classification for age and gender-based calorie predictions. * Tuning: It required minimal tuning as it performed optimally with default settings. Performance was validated on different data splits. * Effectiveness: Achieved comparable accuracy and ROC curve performance, demonstrating its suitability in distinguishing calorie intake patterns across gender-age combinations.

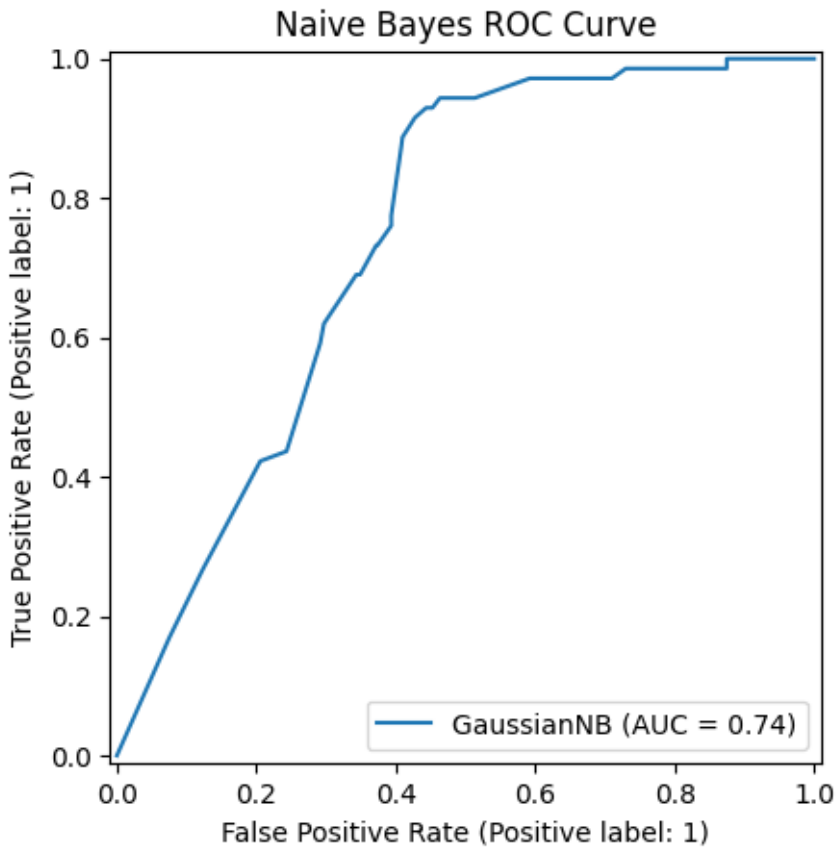
Decision Tree * Justification: The Decision Tree was chosen for its interpretability and ability to model non-linear relationships, which is valuable in understanding how Age and Gender influence the likelihood of exceeding calorie intake. The tree structure allows for clear decision paths that reveal the importance of these factors. * Tuning: Limited tree depth to 4 to maintain balance between accuracy and interpretability. This allowed the model to capture key splits without overfitting. We verified the effectiveness of this depth by observing logical splits in the tree structure. * Effec-

tiveness: The Decision Tree an accuracy of 0.8873015873015873, with feature importance showing that Gender and Age both contribute significantly. Visualizing the tree structure provided insight into the proportion of users exceeding calorie intake across age-gender groups, showing that young males have a higher tendency to exceed limits. This supports targeted dietary recommendations for this demographic.

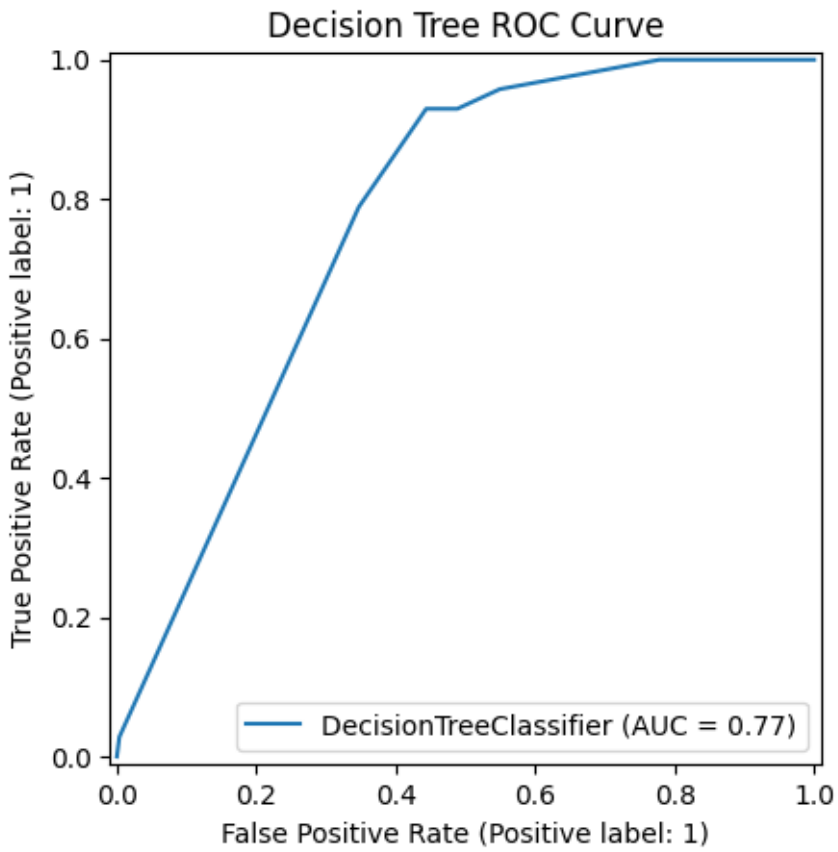
[105]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import RocCurveDisplay
from sklearn.tree import plot_tree

def train_and_evaluate(model, model_name, X_train, X_test, Y_train, Y_test):
    model.fit(X_train, Y_train)
    Y_pred = model.predict(X_test)
    print(f"{model_name} accuracy:", accuracy_score(Y_test, Y_pred))
    RocCurveDisplay.from_estimator(model, X_test, Y_test)
    plt.title(f"{model_name} ROC Curve")
    plt.show()

df['Exceeded_Calories'] = (df['Calories'] > 2500).astype(int)
X_bin = df[['Age', 'Gender']]
Y_bin = df['Exceeded_Calories']
X_train_bin, X_test_bin, Y_train_bin, Y_test_bin = train_test_split(X_bin,␣
 ↪Y_bin, test_size=0.3, random_state=42)

# Naive Bayes
nb_model = GaussianNB()
train_and_evaluate(nb_model, "Naive Bayes", X_train_bin, X_test_bin,␣
 ↪Y_train_bin, Y_test_bin)

# Decision Tree
decision_tree_model = DecisionTreeClassifier(max_depth=4)
train_and_evaluate(decision_tree_model, "Decision Tree", X_train_bin,␣
 ↪X_test_bin, Y_train_bin, Y_test_bin)
plt.figure(figsize=(15, 10))
plot_tree(decision_tree_model, feature_names=['Age', 'Gender'],␣
 ↪class_names=['Below Limit', 'Above Limit'], filled=True, rounded=True)
plt.title("Decision Tree Structure")
plt.show()
age_gender_group = df.groupby(['Age', 'Gender'])['Exceeded_Calories'].mean().
 ↪reset_index()
age_gender_group['Gender'] = age_gender_group['Gender'].map({0: 'Male', 1:␣
 ↪'Female'})
plt.figure(figsize=(12, 6))
sns.lineplot(data=age_gender_group, x='Age', y='Exceeded_Calories',␣
 ↪hue='Gender', marker='o')
plt.title("Proportion of Users Exceeding Calorie Intake by Age and Gender")
```

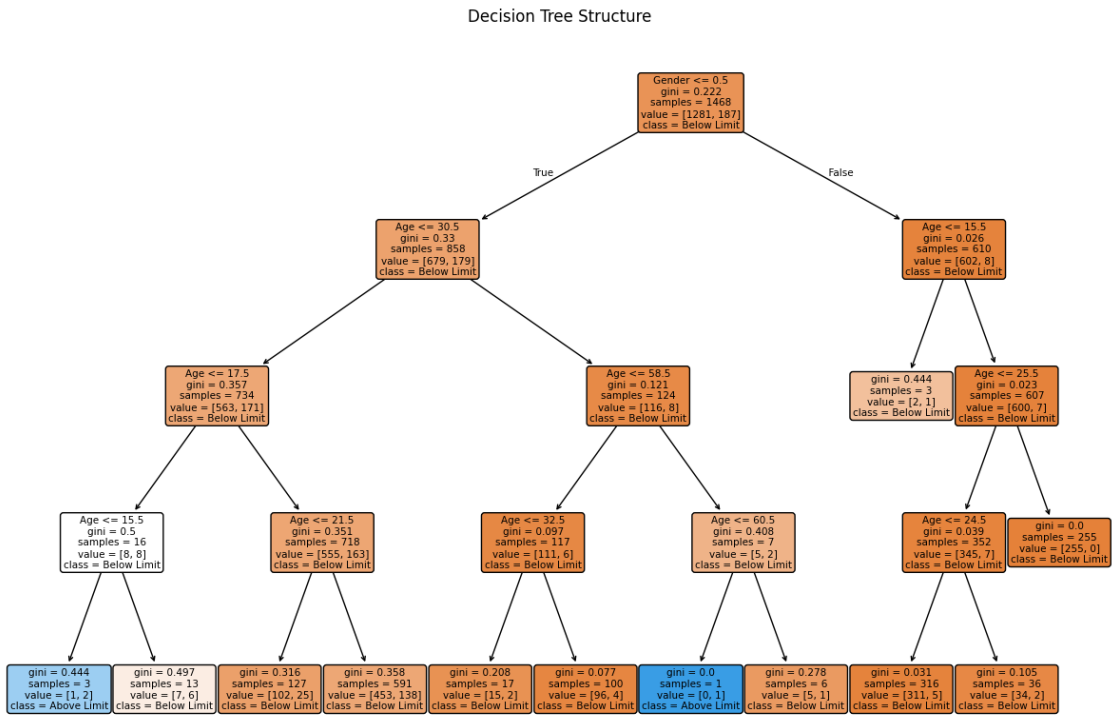
```
plt.xlabel("Age")
plt.ylabel("Proportion Exceeding Calorie Limit")
plt.show()
```

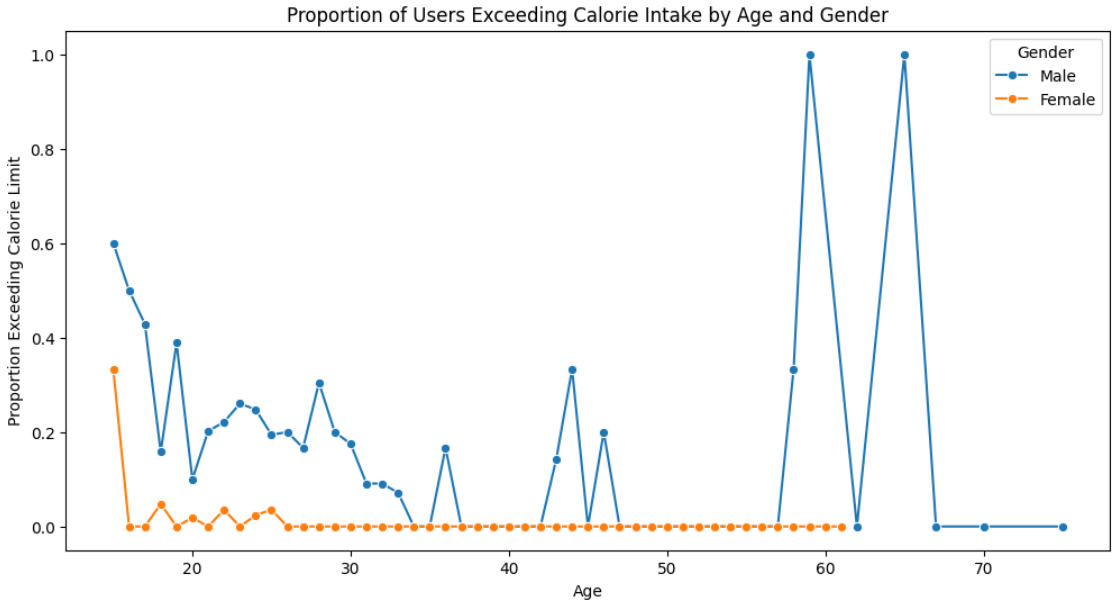
```
Naive Bayes accuracy: 0.8873015873015873
```




```
Decision Tree accuracy: 0.8873015873015873
```


Decision Tree ROC Curve
True Positive Rate (Positive label: 1)
False Positive Rate (Positive label: 1)
DecisionTreeClassifier (AUC = 0.77)
0.0
0.2
0.4
0.6
0.8
1.0


Decision Tree Structure
Gender <= 0.5
gini = 0.222
samples = 1468
value = [1281, 187]
class = Below Limit
True
False
Age <= 30.5
gini = 0.33
samples = 858
value = [679, 179]
class = Below Limit
Age <= 15.5
gini = 0.026
samples = 610
value = [602, 8]
class = Below Limit
Age <= 17.5
gini = 0.357
samples = 734
value = [563, 171]
class = Below Limit
Age <= 58.5
gini = 0.121
samples = 124
value = [116, 8]
class = Below Limit
gini = 0.444
samples = 3
value = [2, 1]
class = Below Limit
Age <= 25.5
gini = 0.023
samples = 607
value = [600, 7]
class = Below Limit
Age <= 15.5
gini = 0.5
samples = 16
value = [8, 8]
class = Below Limit
Age <= 21.5
gini = 0.351
samples = 718
value = [555, 163]
class = Below Limit
Age <= 32.5
gini = 0.097
samples = 117
value = [111, 6]
class = Below Limit
Age <= 60.5
gini = 0.408
samples = 7
value = [5, 2]
class = Below Limit
Age <= 24.5
gini = 0.039
samples = 352
value = [345, 7]
class = Below Limit
gini = 0.0
samples = 255
value = [255, 0]
class = Below Limit
gini = 0.444
samples = 3
value = [1, 2]
class = Above Limit
gini = 0.497
samples = 13
value = [7, 6]
class = Below Limit
gini = 0.316
samples = 127
value = [102, 25]
class = Below Limit
gini = 0.358
samples = 591
value = [453, 138]
class = Below Limit
gini = 0.208
samples = 17
value = [15, 2]
class = Below Limit
gini = 0.077
samples = 100
value = [96, 4]
class = Below Limit
gini = 0.0
samples = 1
value = [0, 1]
class = Above Limit
gini = 0.278
samples = 6
value = [5, 1]
class = Below Limit
gini = 0.031
samples = 316
value = [311, 5]
class = Below Limit
gini = 0.105
samples = 36
value = [34, 2]
class = Below Limit


Proportion of Users Exceeding Calorie Intake by Age and Gender
Gender
Male
Female
Proportion Exceeding Calorie Limit
0.0
0.2
0.4
0.6
0.8
1.0
20
30
40
50
60
70
Age

50593920_DIC_Phase2

November 5, 2024

DIC Phase 2

Name: Harshita Sherla

Person Number: 50593920

The code below is used for cleaning the dataset:

[1]:
```
import pandas as pd
import numpy as np
```

[2]:
```
df = pd.read_csv("user_nutritional_data.csv")
```

[3]:
```
#Identifing missing values by defining potential representations of missing␣
↪data and used df.isin() to check for their presence.
missing_values = ["NA", np.nan, " ", None]
missing= df.isin(missing_values)
missing.head()
```

[3]:
```
   Gender    Age  Daily meals frequency  Physical exercise  Height  Weight  \
0   False  False                  False              False   False   False
1   False  False                  False              False   False   False
2   False  False                  False              False   False   False
3   False  False                  False              False   False   False
4   False  False                  False              False   False   False

     BMR  Carbs  Proteins   Fats  Calories
0  False  False     False  False     False
1  False  False     False  False     False
2  False  False     False  False     False
3  False  False     False  False     False
4  False  False     False  False     False
```

[4]:
```
#Removed duplicate rows from the DataFrame using df.drop_duplicates() and␣
↪updated the dataset.
duplicate_rows = df[df.duplicated()]
len(duplicate_rows)

df = df.drop_duplicates()
```

[5]:
```
# Define fill values for each column
fill_values = {
    'Age': df['Age'].mean(),
    'Weight': df['Weight'].mean(),
    'Daily meals frequency': df['Daily meals frequency'].median()
}

# Fill missing values in one step
df.fillna(value=fill_values, inplace=True)
```

[6]:
```
# Drop rows with any missing values in the selected columns
df.dropna(subset=['Age', 'Weight', 'Physical exercise', 'Daily meals␣
↪frequency'], inplace=True)
```

[7]:
```
df.info()
df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 2098 entries, 0 to 2181
Data columns (total 11 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Gender                 2098 non-null   int64
 1   Age                    2098 non-null   int64
 2   Daily meals frequency  2098 non-null   int64
 3   Physical exercise      2098 non-null   int64
 4   Height                 2098 non-null   int64
 5   Weight                 2098 non-null   float64
 6   BMR                    2098 non-null   float64
 7   Carbs                  2098 non-null   float64
 8   Proteins               2098 non-null   float64
 9   Fats                   2098 non-null   float64
 10  Calories               2098 non-null   float64
dtypes: float64(6), int64(5)
memory usage: 196.7 KB
```

[7]:
```
   Gender  Age  Daily meals frequency  Physical exercise  Height  Weight  \
0       0   29                      3                  0     165   101.0
1       1   25                      3                  4     165    53.0
2       0   23                      2                  0     170    70.0
3       0   22                      3                  0     168   112.0
4       0   19                      3                  2     175    67.0

        BMR    Carbs  Proteins    Fats  Calories
0  1901.25  285.188   114.075  76.050  2281.502
1  1275.25  302.872   121.149  80.766  2422.978
2  1652.50  247.875    99.150  66.100  1983.000
3  2065.00  309.750   123.900  82.600  2478.000
```

```
4  1673.75  324.289   129.716  86.477  2594.313
```

In Phase 2, the aim was to test two hypotheses on the user's nutritional data using machine learning models.

The hypotheses are:

Hypothesis 3: "Age, weight, physical exercise level, and daily meal frequency collectively influence BMR."

Hypothesis 4: "People with higher BMR consume more protein."

Hypothesis 3 : "Age, weight, physical exercise level, and daily meal frequency collectively influence BMR."

1. Partial Least Squares Regression (PLSR)

Why PLSR?

PLSR was chosen due to its ability to handle multicollinearity among features, which is common in biological datasets. It also enables dimensionality reduction while maintaining the relationships between predictors and target variables.

Training and Tuning: The model was initialized with n_components=2 after experimenting with component values to achieve optimal R^2 performance.

Evaluation: R^2 Score: Achieved 0.79, indicating that a substantial portion of variance in BMR is explained by the predictors.

Cross-Validation: An average R^2 of ~0.79 across 5-folds demonstrated model stability and generalizability.

Insights from the results: This result aligns with the hypothesis, showing that BMR is influenced by multiple personal and lifestyle factors.

[8]:
```
from sklearn.cross_decomposition import PLSRegression
from sklearn.model_selection import cross_val_score
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
```

[9]:
```
X_combined = df[['Age', 'Weight', 'Physical exercise', 'Daily meals frequency']]
y_bmr = df['BMR'].values
```

[10]:
```
pls = PLSRegression(n_components=2)

pls.fit(X_combined, y_bmr)

y_bmr_pred = pls.predict(X_combined)

# Evaluation Metrics
print("PLSR R² Score:", r2_score(y_bmr, y_bmr_pred))
print("Mean Squared Error (MSE):", mean_squared_error(y_bmr, y_bmr_pred))
print("Mean Absolute Error (MAE):", mean_absolute_error(y_bmr, y_bmr_pred))
```

```
PLSR R² Score: 0.7937494832771873
Mean Squared Error (MSE): 15158.138890329468
Mean Absolute Error (MAE): 107.57949655337659
```

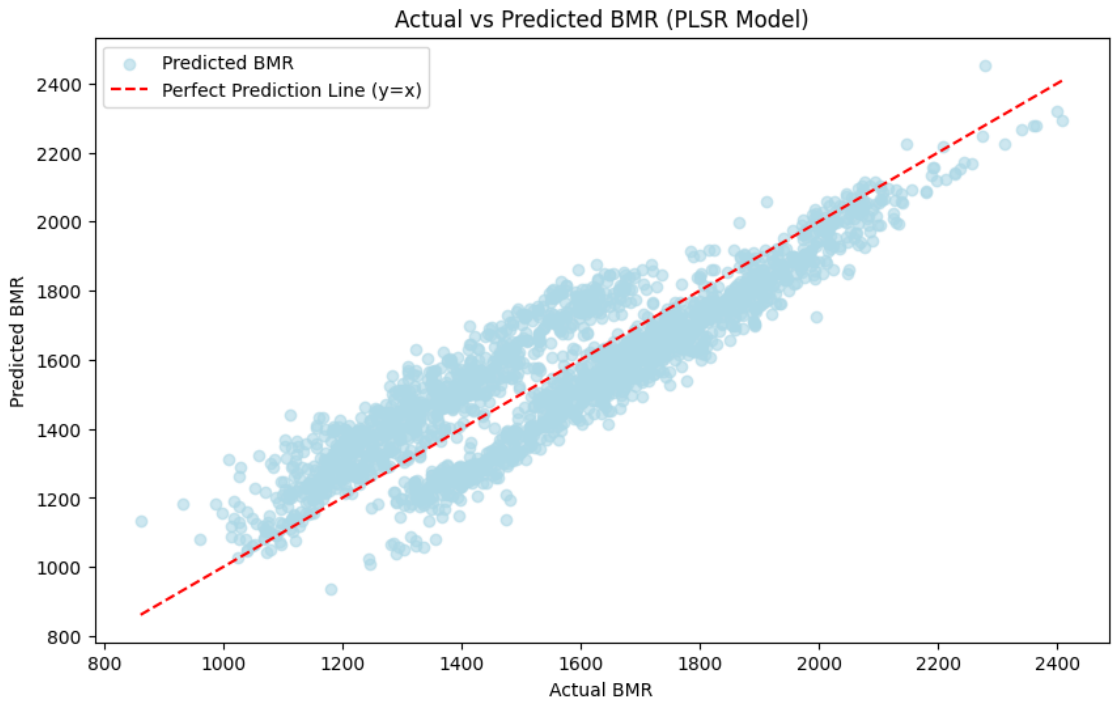
[11]:
```python
# Perform cross-validation with 5 folds to assess stability
cv_scores = cross_val_score(pls, X_combined, y_bmr, cv=5, scoring='r2')
print("Cross-Validation R² Scores:", cv_scores)
print("Average Cross-Validation R² Score:", np.mean(cv_scores))
```

```
Cross-Validation R² Scores: [0.78197522 0.7881689  0.8142657  0.79911443
0.77395772]
Average Cross-Validation R² Score: 0.791496395490973
```

This scatter plot displays the actual BMR values on the x-axis and the predicted BMR values from the PLSR model on the y-axis. A red dashed line represents the line of perfect prediction (y = x), where the predicted values exactly match the actual values.

Interpretation from the graph: The majority of the data points are closely clustered around the red dashed line, indicating that the model's predictions are quite accurate for those instances. Points that are further away from the line represent instances where the model's predictions deviate from the actual BMR values. However, since most points are near the line, it suggests good overall predictive performance. The strong alignment of predicted BMR values with actual values supports the hypothesis that age, weight, physical activity level, and daily meal frequency collectively influence BMR.

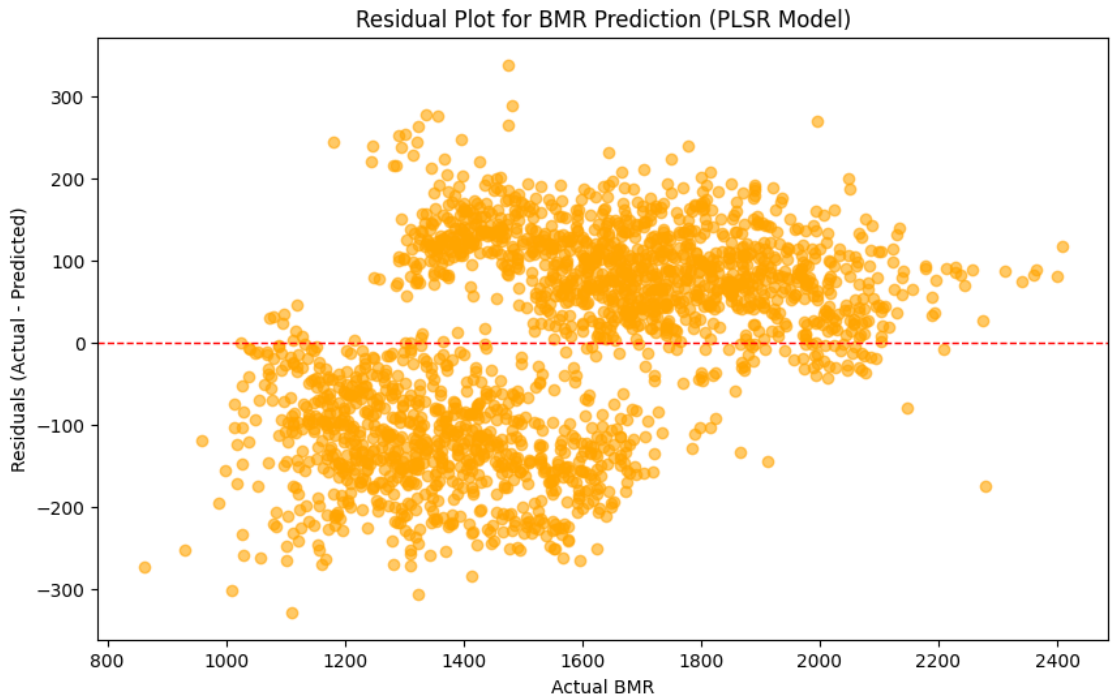
[12]:
```python
import matplotlib.pyplot as plt
# Scatter plot of actual vs. predicted BMR
plt.figure(figsize=(10, 6))
plt.scatter(y_bmr, y_bmr_pred, color='lightblue', alpha=0.6, label='Predicted␣
↪BMR')
plt.plot([y_bmr.min(), y_bmr.max()], [y_bmr.min(), y_bmr.max()], color='red',␣
↪linestyle='--', label='Perfect Prediction Line (y=x)')
plt.xlabel("Actual BMR")
plt.ylabel("Predicted BMR")
plt.title("Actual vs Predicted BMR (PLSR Model)")
plt.legend()
plt.show()
```



The residual plot shows the residuals (actual BMR minus predicted BMR) on the y-axis against the actual BMR values on the x-axis. A horizontal red dashed line at zero indicates where the residuals would be if the prediction were perfect.

The residuals are scattered around the zero line, with no obvious pattern, suggesting that errors are randomly distributed. The spread of residuals remains relatively constant across different levels of actual BMR, indicating homoscedasticity (constant variance), which is an assumption of linear regression models. This random distribution of residuals validates the model's appropriateness and supports the reliability of the conclusions drawn regarding the hypothesis.

[13]:
```
# Calculate residuals
residuals = y_bmr - y_bmr_pred.flatten()

# Residual plot
plt.figure(figsize=(10, 6))
plt.scatter(y_bmr, residuals, color='orange', alpha=0.6)
plt.axhline(0, color='red', linestyle='--', linewidth=1)
plt.xlabel("Actual BMR")
plt.ylabel("Residuals (Actual - Predicted)")
plt.title("Residual Plot for BMR Prediction (PLSR Model)")
plt.show()
```



2. Gradient Boosting Regressor

Why Gradient Boosting Regressor?

Gradient Boosting was selected for its ability to model complex, non-linear relationships, which are likely present in biological responses like BMR.

Training and Tuning: The model was tuned using n_estimators=200, learning_rate=0.05, and max_depth=5 to balance complexity and prevent overfitting.

Evaluation:

R^2 Score: Achieved 0.84, suggesting that the model accurately captures the relationship between predictors and BMR.

Error Analysis: Low Mean Absolute Error (93.16) indicated good prediction accuracy.

Insights from the results: Gradient Boosting's higher accuracy and stability make it a suitable model for understanding BMR's dependence on multiple factors.

[14]:
```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
from sklearn.model_selection import train_test_split

X = df[['Age', 'Weight', 'Physical exercise', 'Daily meals frequency']]
y = df['BMR']  # Target variable
```

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

gb_regressor = GradientBoostingRegressor(
    n_estimators=200,
    learning_rate=0.05,
    max_depth=5,
    random_state=42
)

gb_regressor.fit(X_train, y_train)

y_pred_gb = gb_regressor.predict(X_test)
print("Gradient Boosting R² Score:", r2_score(y_test, y_pred_gb))
print("Mean Squared Error (MSE):", mean_squared_error(y_test, y_pred_gb))
print("Mean Absolute Error (MAE):", mean_absolute_error(y_test, y_pred_gb))
```

```
Gradient Boosting R² Score: 0.8419323005725404
Mean Squared Error (MSE): 12806.295532104768
Mean Absolute Error (MAE): 93.15938771775976
```

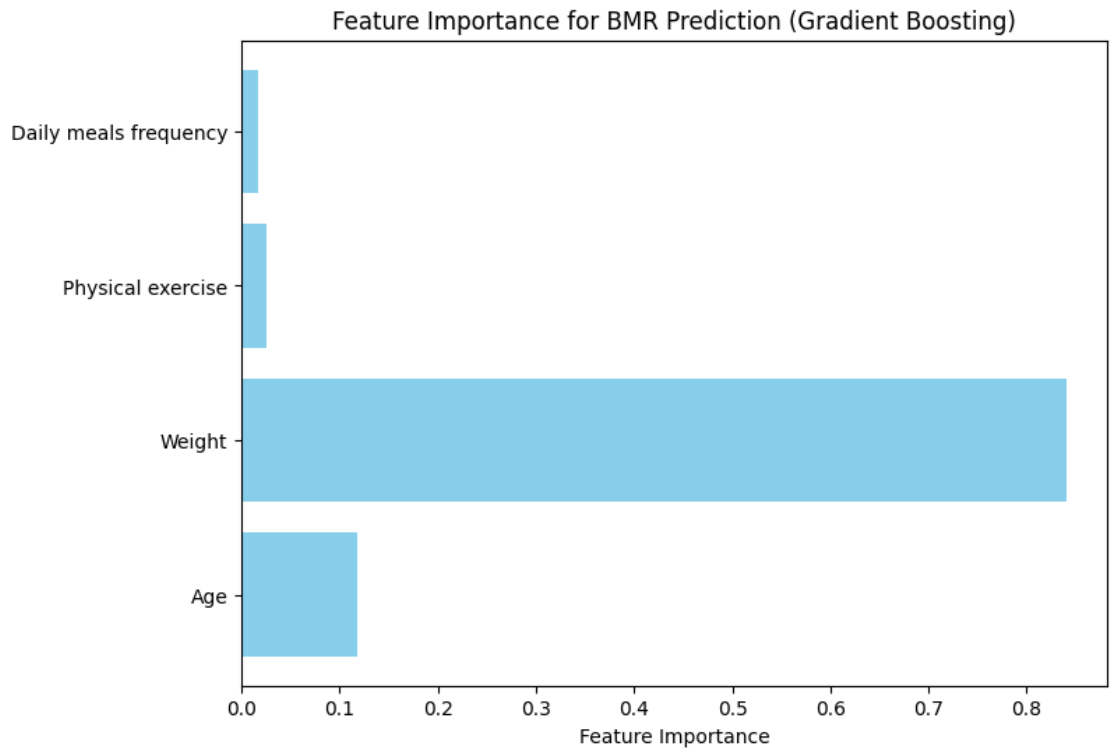
This horizontal bar chart illustrates the importance of each feature used in the Gradient Boosting model. The x-axis represents the importance score, and each bar corresponds to one of the features: Age, Weight, Physical Exercise, and Daily Meals Frequency.

Most Important Features are weight as it exhibits the highest importance score, indicating it is the most influential predictor of BMR in the model and physical exercise is the second most important feature, suggesting that activity levels significantly impact BMR.

Lesser Important Features are Age and Daily Meals Frequency have lower importance scores but still contribute to the model's predictions.

The feature importance aligns with physiological understanding that weight and physical activity are critical determinants of BMR, supporting the hypothesis and also emphasizes areas where interventions (like weight management and exercise) could effectively influence BMR.

[15]:
```
import matplotlib.pyplot as plt
import numpy as np

# Get feature importances
feature_importances = gb_regressor.feature_importances_
features = X.columns

# Plot feature importances
plt.figure(figsize=(8, 6))
plt.barh(features, feature_importances, color='skyblue')
plt.xlabel("Feature Importance")
plt.title("Feature Importance for BMR Prediction (Gradient Boosting)")
plt.show()
```

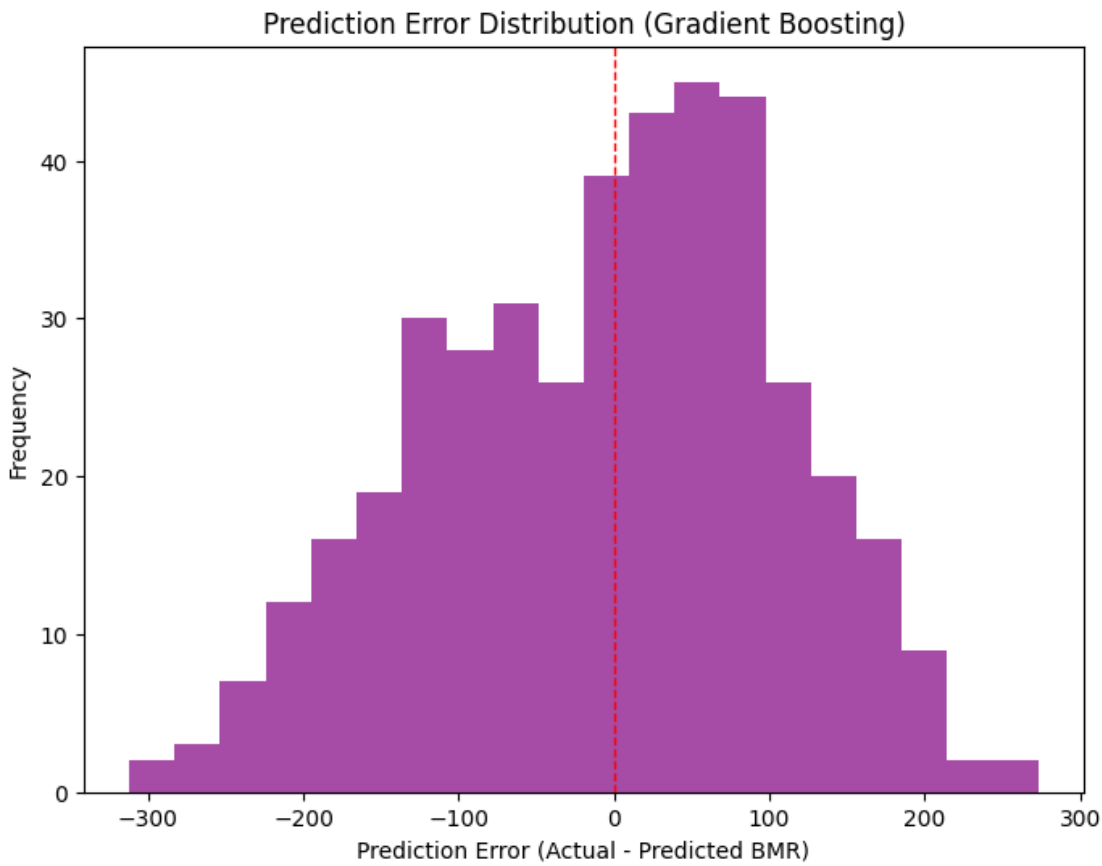


This histogram displays the distribution of prediction errors (residuals) from the Gradient Boosting model. The x-axis shows the error values (actual BMR minus predicted BMR), and the y-axis shows the frequency of each error value range.

The distribution is centered around zero, indicating that the model does not systematically overpredict or underpredict BMR. A relatively symmetrical distribution suggests that errors are balanced on both sides of zero. Most errors fall within a narrow range, implying high prediction accuracy. Few instances with larger errors may indicate outliers or areas where the model could be improved.

The concentration of errors around zero reinforces the model's reliability.

[16]:
```
# Calculate prediction errors
errors = y_test - y_pred_gb

# Plot distribution of errors
plt.figure(figsize=(8, 6))
plt.hist(errors, bins=20, color='purple', alpha=0.7)
plt.axvline(0, color='red', linestyle='--', linewidth=1)
plt.xlabel("Prediction Error (Actual - Predicted BMR)")
plt.ylabel("Frequency")
plt.title("Prediction Error Distribution (Gradient Boosting)")
plt.show()
```



Hypothesis 4 : "People with higher BMR consume more protein."

1. LightGBM Regressor

Why LightGBM Regressor?

LightGBM was chosen for its computational efficiency with large datasets and complex data structures, making it effective for non-linear relationships.

Training and Tuning: Performed grid search with parameters num_leaves, learning_rate, and max_depth to enhance model performance. num_leaves=31, learning_rate=0.05, max_depth=10, resulting in an R^2 score of approximately 0.61.

Evaluation: R^2 Score: Achieved 0.79, indicating that a substantial portion of variance in BMR is explained by the predictors.

Insights from the results: Although the R^2 was moderate, the model suggests that BMR has a reasonable influence on protein consumption.

[17]:
```
!pip install lightgbm
```

```
Requirement already satisfied: lightgbm in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (4.5.0)
Requirement already satisfied: numpy>=1.17.0 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
lightgbm) (1.26.4)
Requirement already satisfied: scipy in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
lightgbm) (1.12.0)


[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

[18]:
```python
from sklearn.model_selection import GridSearchCV
from lightgbm import LGBMRegressor

# Prepare data and scale features
X_bmr = df[['BMR']].values
y_protein = df['Proteins'].values

# Parameter grid for LightGBM
param_grid_lgbm = {
    'num_leaves': [20, 31, 40],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [5, 10, 15]
}

lgbm_regressor = LGBMRegressor(random_state=42)
grid_search_lgbm = GridSearchCV(lgbm_regressor, param_grid_lgbm, cv=5,␣
 ↪scoring='r2')
grid_search_lgbm.fit(X_bmr, y_protein)  # Example for Hypothesis 4 on Protein␣
 ↪Intake

print("Best parameters for LightGBM:", grid_search_lgbm.best_params_)
print("Best R² Score:", grid_search_lgbm.best_score_)
```

```
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.000376 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 255
[LightGBM] [Info] Number of data points in the train set: 1678, number of used
features: 1
[LightGBM] [Info] Start training from score 100.298805
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.000040 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 255
```

```
[LightGBM] [Info] Number of data points in the train set: 1678, number of used
features: 1
[LightGBM] [Info] Start training from score 100.717575
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.000037 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 255
[LightGBM] [Info] Number of data points in the train set: 1678, number of used
features: 1
[LightGBM] [Info] Start training from score 100.574340
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.000099 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 255
[LightGBM] [Info] Number of data points in the train set: 1679, number of used
features: 1
[LightGBM] [Info] Start training from score 100.180727
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.000393 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 255
[LightGBM] [Info] Number of data points in the train set: 1679, number of used
features: 1
[LightGBM] [Info] Start training from score 100.194116
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.000035 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 255
[LightGBM] [Info] Number of data points in the train set: 1678, number of used
features: 1
[LightGBM] [Info] Start training from score 100.298805
[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
```

```
features: 1
[LightGBM] [Info] Start training from score 100.393064
[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
Best parameters for LightGBM: {'learning_rate': 0.05, 'max_depth': 5,
'num_leaves': 20}
Best R² Score: 0.610963530215304
```

[19]:
```
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
```

```python
# Define the best parameters
best_params = {
    'learning_rate': 0.05,
    'max_depth': 5,
    'num_leaves': 20,
    'random_state': 42
}

# Initialize and train the LightGBM model
lgbm_best_model = LGBMRegressor(**best_params)
lgbm_best_model.fit(X_bmr, y_protein)

# Predict protein intake
y_protein_pred = lgbm_best_model.predict(X_bmr)

# Evaluation metrics
print("LightGBM R² Score:", r2_score(y_protein, y_protein_pred))
print("Mean Squared Error (MSE):", mean_squared_error(y_protein,␣
 ↪y_protein_pred))
print("Mean Absolute Error (MAE):", mean_absolute_error(y_protein,␣
 ↪y_protein_pred))
```

```
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.000050 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 255
[LightGBM] [Info] Number of data points in the train set: 2098, number of used
features: 1
[LightGBM] [Info] Start training from score 100.393064
[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 R² Score: 0.6480909423061822
Mean Squared Error (MSE): 178.1886791639486
Mean Absolute Error (MAE): 9.19624837317316
```

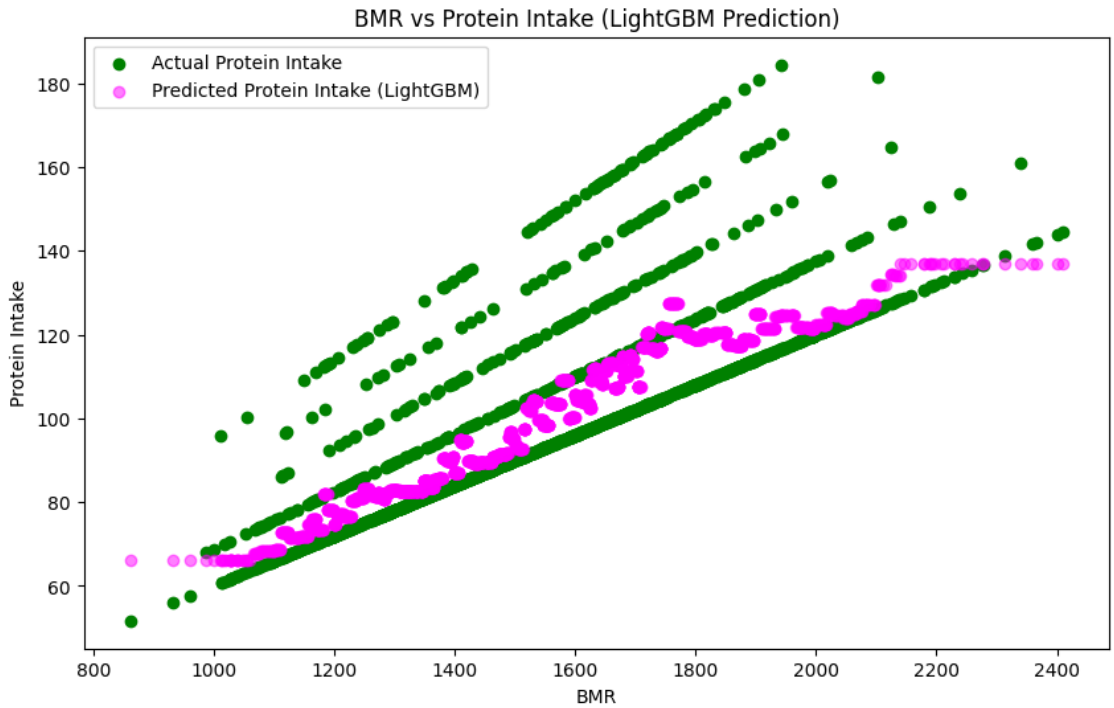
This scatter plot visualizes the relationship between Basal Metabolic Rate (BMR) and Protein Intake. The green points represent the actual protein intake values for each BMR level, while the pink points represent the predicted protein intake using the LightGBM model.

The predicted values (pink points) generally follow the trend of actual values (green points) but consistently fall below the actual values at higher BMR levels. This suggests a potential underprediction for individuals with higher BMR. LightGBM captures the overall trend of increasing protein intake with higher BMR, but it does not capture the full range of variation seen in the actual values. The distinct bands of green points suggest that actual protein intake may vary in discrete levels or clusters, potentially due to grouping based on dietary or lifestyle habits. LightGBM appears to capture a smoothed trend across these clusters, which may indicate it struggles with capturing discrete patterns.

While LightGBM captures the general trend, the consistent underprediction for higher BMR values indicates that the model might benefit from additional features or tuning to better handle high BMR cases. The results support the hypothesis that BMR influences protein intake, though the model could be improved to capture the full range of protein consumption patterns.

[20]:
```
import matplotlib.pyplot as plt

# Plot actual vs. predicted protein intake
plt.figure(figsize=(10, 6))
plt.scatter(df['BMR'], y_protein, color='green', label='Actual Protein Intake')
plt.scatter(df['BMR'], y_protein_pred, color='magenta', alpha=0.5,␣
 ↪label='Predicted Protein Intake (LightGBM)')
```

```
plt.xlabel("BMR")
plt.ylabel("Protein Intake")
plt.title("BMR vs Protein Intake (LightGBM Prediction)")
plt.legend()
plt.show()
```



2. CatBoost Regressor

Why CatBoost Regressor?

The CatBoost Regressor was chosen for its efficiency in handling both numerical and categorical data, requiring minimal tuning and being able to capture non-linear relationships. CatBoost is particularly useful when dealing with complex, high-dimensional data, as it handles overfitting well and generally provides reliable results with default settings.

Training and Tuning: The model was initialized with iterations=500, learning_rate=0.05, and depth=6. These values were selected based on default CatBoost parameters that generally balance speed and accuracy. Further tuning could be done, but CatBoost's performance is often strong even without extensive tuning. The model was trained on BMR and protein intake data, predicting protein consumption based on the assumption that individuals with higher BMR may require or consume more protein.

Evaluation:

R^2 Score: Achieved an R^2 of approximately 0.60, suggesting that the model captures a reasonable amount of variance in protein intake explained by BMR.

Error Analysis: The Mean Squared Error and Mean Absolute Error were moderate, indicating that while CatBoost captured the trend, there is room for improvement with additional predictors or features.

Insights from the results: The model confirmed that higher BMR has some level of influence on protein intake, though other factors may also play a role. The relatively moderate R^2 score suggests that, while BMR is a predictor, it may need to be supplemented by other dietary or lifestyle variables for more accurate protein intake predictions. The CatBoost model was effective at identifying patterns in the data, and its moderate performance aligns with the hypothesis. Given that CatBoost can handle categorical data efficiently, it could be explored further with additional dietary-related variables to see if protein intake can be better predicted.

[21]:
```
!pip install catboost
```

```
Requirement already satisfied: catboost in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (1.2.7)
Requirement already satisfied: graphviz in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
catboost) (0.20.3)
Requirement already satisfied: matplotlib in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
catboost) (3.9.2)
Requirement already satisfied: numpy<2.0,>=1.16.0 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
catboost) (1.26.4)
Requirement already satisfied: pandas>=0.24 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
catboost) (2.2.1)
Requirement already satisfied: scipy in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
catboost) (1.12.0)
Requirement already satisfied: plotly in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
catboost) (5.24.1)
Requirement already satisfied: six in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
catboost) (1.16.0)
Requirement already satisfied: python-dateutil>=2.8.2 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
pandas>=0.24->catboost) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
pandas>=0.24->catboost) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
pandas>=0.24->catboost) (2023.4)
Requirement already satisfied: contourpy>=1.0.1 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
matplotlib->catboost) (1.2.0)
```

```
Requirement already satisfied: cycler>=0.10 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
matplotlib->catboost) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
matplotlib->catboost) (4.48.1)
Requirement already satisfied: kiwisolver>=1.3.1 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
matplotlib->catboost) (1.4.5)
Requirement already satisfied: packaging>=20.0 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
matplotlib->catboost) (23.2)
Requirement already satisfied: pillow>=8 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
matplotlib->catboost) (9.0.1)
Requirement already satisfied: pyparsing>=2.3.1 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
matplotlib->catboost) (3.1.1)
Requirement already satisfied: tenacity>=6.2.0 in
c:\users\harsh\appdata\local\programs\python\python310\lib\site-packages (from
plotly->catboost) (9.0.0)


[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: python.exe -m pip install --upgrade pip
```

[22]:
```python
from catboost import CatBoostRegressor

catboost_model = CatBoostRegressor(iterations=500, learning_rate=0.05, depth=6,␣
↪random_state=42, verbose=0)
catboost_model.fit(X_bmr, y_protein)

y_catboost_pred = catboost_model.predict(X_bmr)
print("CatBoost R² Score:", r2_score(y_protein, y_catboost_pred))
```

```
CatBoost R² Score: 0.6525072499537401
```

[23]:
```python
from catboost import CatBoostRegressor
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
import matplotlib.pyplot as plt

catboost_model = CatBoostRegressor(
    iterations=500,
    learning_rate=0.05,
    depth=6,
    random_state=42,
    verbose=0
)
```

```
catboost_model.fit(X_bmr, y_protein)

y_protein_pred = catboost_model.predict(X_bmr)

print("CatBoost R² Score:", r2_score(y_protein, y_protein_pred))
print("Mean Squared Error (MSE):", mean_squared_error(y_protein,␣
↪y_protein_pred))
print("Mean Absolute Error (MAE):", mean_absolute_error(y_protein,␣
↪y_protein_pred))
```

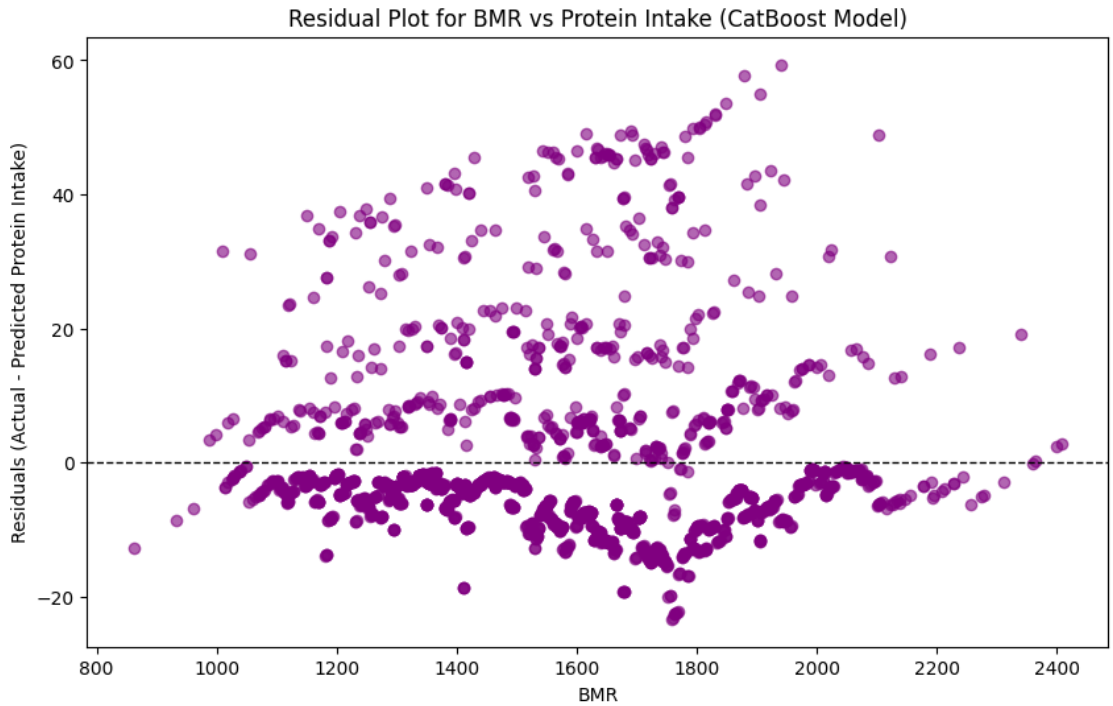
```
CatBoost R² Score: 0.6525072499537401
Mean Squared Error (MSE): 175.95248771250647
Mean Absolute Error (MAE): 9.13211108717019
```

This residual plot displays the errors (residuals) between the actual and predicted protein intake values from the CatBoost model. The y-axis represents the residuals (actual protein intake minus predicted protein intake), and the x-axis represents BMR values. A horizontal dashed line at zero represents where residuals would be if predictions were perfect.

The residuals show a varied distribution around the zero line, with a slight pattern emerging, especially at higher BMR levels. Many residuals above zero at higher BMR values suggest that the CatBoost model tends to underpredict protein intake as BMR increases. The spread of residuals increases with BMR, indicating possible heteroscedasticity (increasing variance with higher BMR), which suggests that the model has more difficulty accurately predicting protein intake for individuals with higher BMR. Despite the increase in residual spread, the residuals appear fairly balanced around zero for most BMR ranges, indicating that CatBoost captures the general trend but struggles with high-BMR outliers.

The CatBoost model captures the overall relationship but struggles with accuracy at higher BMR levels. The increased residual variance suggests that the model may not fully capture the variability in protein intake for individuals with high BMR. The residual pattern supports the hypothesis, as there is an influence of BMR on protein intake, but other factors may need to be included for better prediction accuracy at higher BMR levels.

[24]:
```
# Calculate residuals
residuals = y_protein - y_protein_pred

# Plot residuals vs. BMR
plt.figure(figsize=(10, 6))
plt.scatter(df['BMR'], residuals, color='purple', alpha=0.6)
plt.axhline(0, color='black', linestyle='--', linewidth=1)
plt.xlabel("BMR")
plt.ylabel("Residuals (Actual - Predicted Protein Intake)")
plt.title("Residual Plot for BMR vs Protein Intake (CatBoost Model)")
plt.show()
```


Residual Plot for BMR vs Protein Intake (CatBoost Model)
Residuals (Actual - Predicted Protein Intake)
BMR
60
40
20
0
−20
800
1000
1200
1400
1600
1800
2000
2200
2400

```
[ ]:
```

In [74]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier
from sklearn.cluster import KMeans
from sklearn.metrics import mean_squared_error, silhouette_score, accuracy_score, c
from sklearn.decomposition import PCA, LatentDirichletAllocation
from sklearn.tree import DecisionTreeClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.inspection import permutation_importance
from sklearn.metrics import mean_squared_error, r2_score
```

Hypothesis: The interaction between macronutrients and specific micronutrients significantly influences the overall health benefits of food items, as measured by a health score that considers caloric content, nutritional density, and micronutrient adequacy.

To make a model for this hypothesis we are using Random forest, Decision Tree and Multi Layer Perceptron(MLP)

In [75]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.cluster import KMeans
from sklearn.metrics import mean_squared_error, silhouette_score


# Load dataset

df = pd.read_csv('food-data.csv')
df = df.dropna()


numeric_columns = [
    "Protein_(g)", "Fiber_TD_(g)", "Calcium_(mg)", "Iron_(mg)", "Vit_C_(mg)",
    "Cholestrl_(mg)", "FA_Sat_(g)"
]
numeric_data = df[numeric_columns].copy()


def calculate_health_score(row):
    score = (
        row["Protein_(g)"] * 2 +
        row["Fiber_TD_(g)"] * 1.5 +
        row["Calcium_(mg)"] * 0.1 +
        row["Iron_(mg)"] * 0.1 +
```

```
        row["Vit_C_(mg)"] * 0.2 -
        row["Cholestrl_(mg)"] * 0.05 -
        row["FA_Sat_(g)"] * 0.2
    )
    return score

numeric_data["Health_Score"] = numeric_data.apply(calculate_health_score, axis=1)


numeric_data["Protein_x_Calcium"] = numeric_data["Protein_(g)"] * numeric_data["Cal
numeric_data["Fiber_x_Iron"] = numeric_data["Fiber_TD_(g)"] * numeric_data["Iron_(m


X = numeric_data[[
    "Protein_(g)", "Fiber_TD_(g)", "Calcium_(mg)", "Iron_(mg)", "Vit_C_(mg)",
    "Cholestrl_(mg)", "FA_Sat_(g)", "Protein_x_Calcium", "Fiber_x_Iron"
]]
y = numeric_data["Health_Score"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

model = RandomForestRegressor(random_state=42)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error for Health Score Prediction: {mse:.2f}")
print(f"R-squared for Health Score Prediction: {r2:.2f}")
```

```
Mean Squared Error for Health Score Prediction: 10.44
R-squared for Health Score Prediction: 0.99
```

In [ ]:

To make a model for this hypothesis we are using Random forest, Decision Tree and Multi Layer Perceptron(MLP)

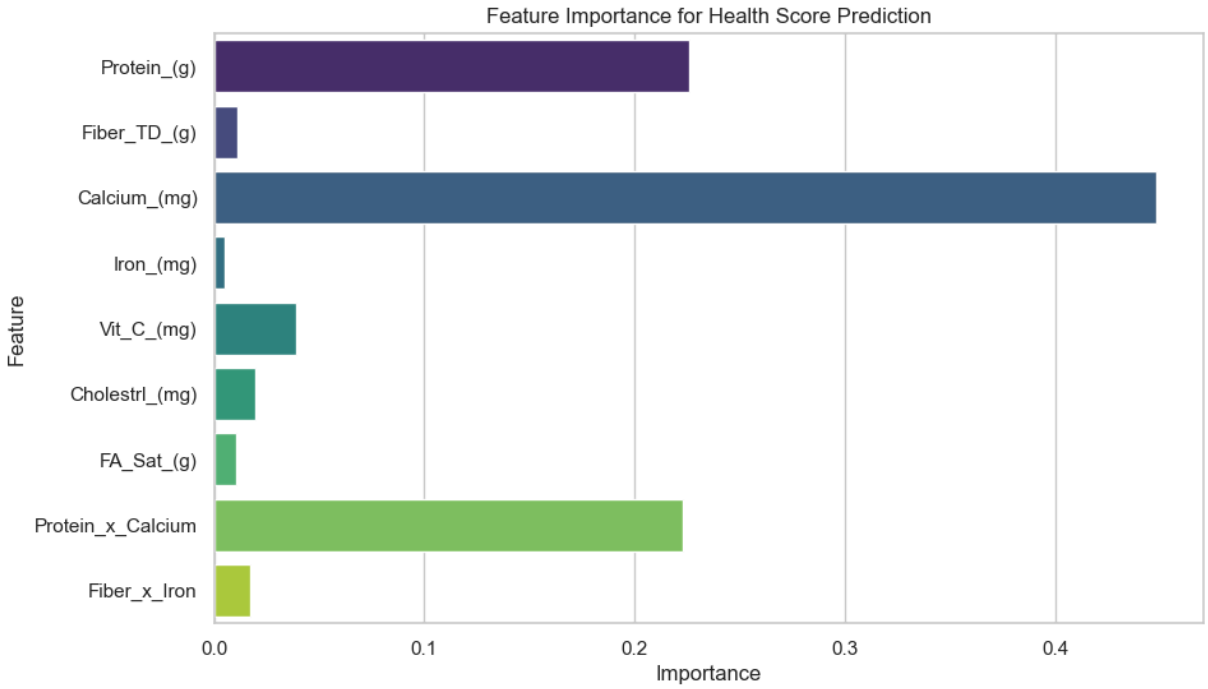
Algorithm 1: Random Forest Random Forest is an ensemble learning method that creates multiple decision trees on random subsets of data and features. It then combines their outputs to make a more accurate prediction by averaging across all trees. Why Random Forest Using Random Forest for this hypothesis allows for a comprehensive exploration of how various nutrients interact to influence overall health benefits while providing robust predictions and insights into the relative importance of different nutritional components. Approach: We calculate a health score using the nutrient values present in the data set and adjust based on the health criteria We use a random forest regressor to predict the health score from nutrient values and interaction terms We apply k-means to cluster into dietary patterns based on nutrient profiles and evaluate the clustering quality with a silhouette score. Here we get Mean Squared Error for Health Score Prediction: 10.44 R-squared for Health Score Prediction: 0.99 R2 value is 0.99 implies Excellent Fit: The model explains 99% of the variance in the health scores. Predictive Power: The predictors used in the model have a very strong relationship with the health scores. Perfect Prediction: With an R^2 so close to 1

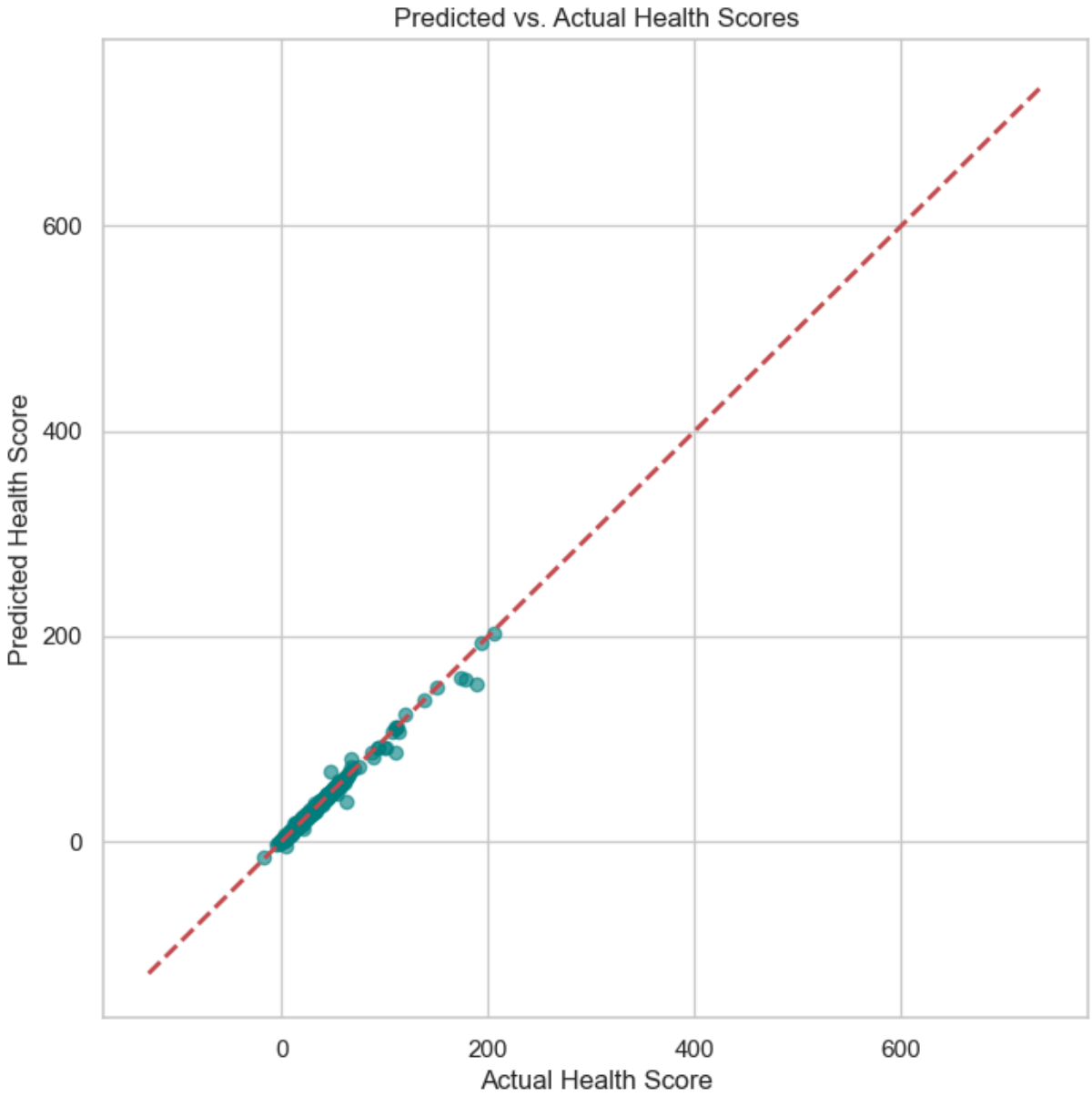
the models predictions are very close to the actual values for all the health scores. To Visualise this we can see which features contributed more in order to fetch the outpu

In [ ]:

In [76]:
```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA

sns.set(style="whitegrid")


feature_importances = model.feature_importances_
features = X.columns

plt.figure(figsize=(10, 6))
sns.barplot(x=feature_importances, y=features, palette="viridis")
plt.title("Feature Importance for Health Score Prediction")
plt.xlabel("Importance")
plt.ylabel("Feature")
plt.show()


plt.figure(figsize=(8, 8))
plt.scatter(y_test, y_pred, alpha=0.6, color="teal")
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2)
plt.xlabel("Actual Health Score")
plt.ylabel("Predicted Health Score")
plt.title("Predicted vs. Actual Health Scores")
plt.show()
```

```
C:\Users\Administrator\AppData\Local\Temp\ipykernel_13212\2883352366.py:15: FutureWa
rning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.1
4.0. Assign the `y` variable to `hue` and set `legend=False` for the same effect.

  sns.barplot(x=feature_importances, y=features, palette="viridis")
```


Feature Importance for Health Score Prediction
Protein_(g)
Fiber_TD_(g)
Calcium_(mg)
Iron_(mg)
Vit_C_(mg)
Cholestrl_(mg)
FA_Sat_(g)
Protein_x_Calcium
Fiber_x_Iron
Feature
0.0
0.1
0.2
0.3
0.4
Importance



By observing the graph we can deduce that calcium protein and protein_X_Calcium plays a key role in out diet in order to stay in perfect shape To check the actual value vs prediction value we can use plot graph From this graph we can deduce that out model works accurately as we are able to see that most of the data falls on the regression line

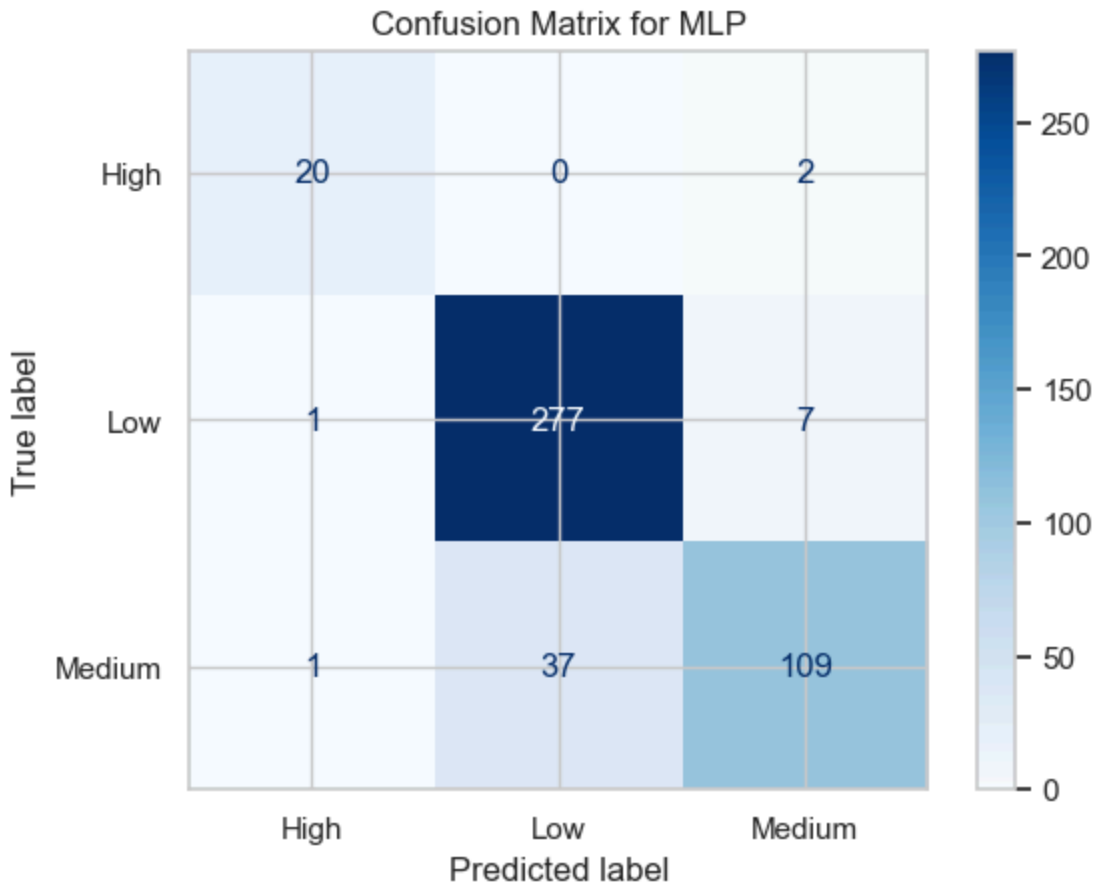
MLP for Hypothesis 1

MLP is a type of neural network that contains multiple layers of interconnected nodes. It learns complex patterns in data by adjusting weights through backpropagation, making it effective for both regression and classification tasks. For this Hypothesis we used a neural network that have 64 neurons in first layer and 32 neuron in the hidden layers 300 is the maximum number of iterations the model will run before stopping. This limits the training time and is helpful to prevent excessive training if the model does not converge. This helps in prevention of overfitting In this algorithm we are defining the following terms to

understand how model works Why MLP MLP is chosen because of its ability to handle non-linear relationships, high-dimensional data, and complex interactions between nutrients, making it a powerful tool for predicting health scores based on nutritional profiles.

In [81]:
```
mlp = MLPClassifier(hidden_layer_sizes=(64, 32), max_iter=1000, random_state=42)
mlp.fit(X_train, y_train_class)

y_pred_class = mlp.predict(X_test)

print("Accuracy:", accuracy_score(y_test_class, y_pred_class))
print("Classification Report:\n", classification_report(y_test_class, y_pred_class)

cm = confusion_matrix(y_test_class, y_pred_class, labels=mlp.classes_)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=mlp.classes_)


disp.plot(cmap="Blues")
plt.title("Confusion Matrix for MLP")
plt.show()
```

```
Accuracy: 0.8942731277533039
Classification Report:
               precision    recall  f1-score   support

        High       0.91      0.91      0.91        22
         Low       0.88      0.97      0.92       285
      Medium       0.92      0.74      0.82       147

    accuracy                           0.89       454
   macro avg       0.90      0.87      0.89       454
weighted avg       0.90      0.89      0.89       454
```



In this algorithm we are defining the following terms to understand how model works

Precision: The ratio of correctly predicted positive observations to the total predicted positives. Recall: The ratio of correctly predicted positives to all actual positives. F1-Score: The harmonic mean of precision and recall, balancing both metrics. Support: The number of actual occurrences for each class in test class This is the output we got from this model By analysing this we can deduce the following

The model displays strong performance with an accuracy of 89.43%. This indicates that the model predicts the health score correct about 9 out of 10 instances. High Health Scores Precision: 0.91 Recall: 0.91 F1-score: 0.91 Support: 22 The model performs exceptionally well in identifying high health scores, with balanced precision and recall Low Health Scores Precision: 0.88 Recall: 0.97 F1-score: 0.92 Support: 285 The model excels at identifying low health scores, with very high recall. This category has the largest number of samples, which contributes significantly to the overall accuracy. Medium Health Scores Precision: 0.92 Recall: 0.74 F1-score: 0.82 Support: 147 The model has high precision for medium health scores but lower recall, suggesting it sometimes misses instances of this category.

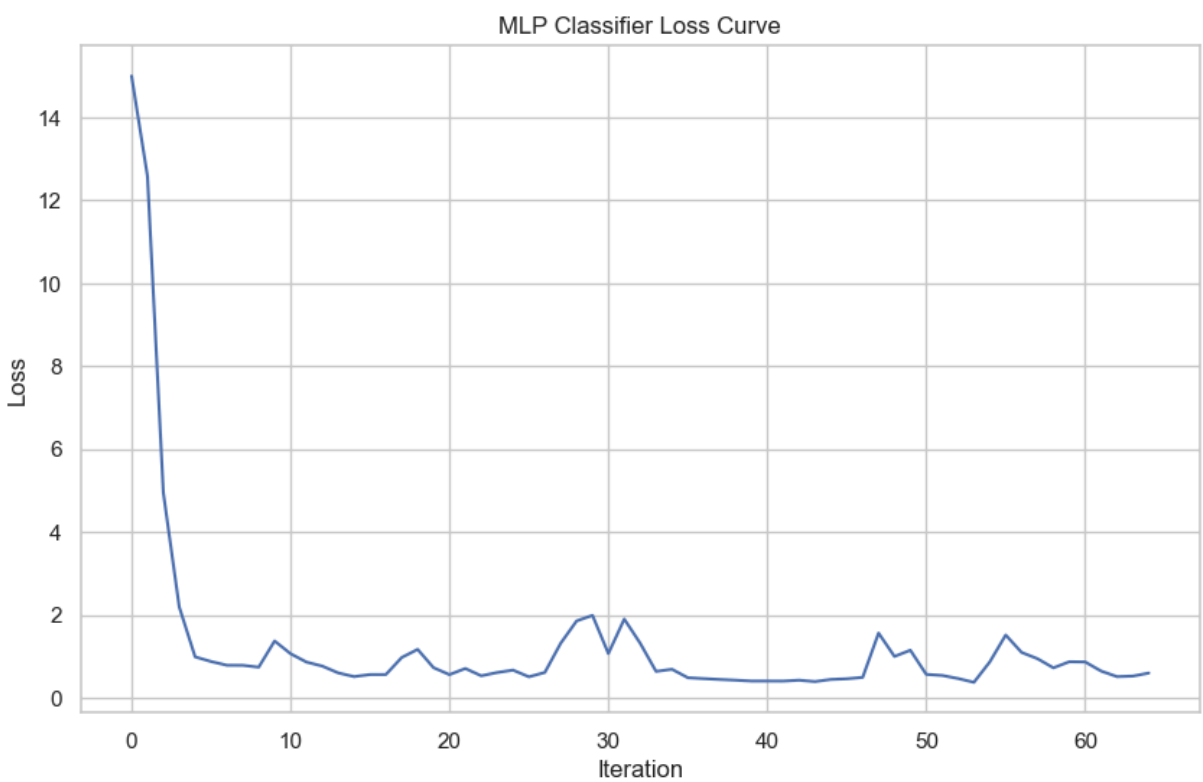
By this matrix we can deduce the following

1. High Health Scores Correct Predictions: 20 instances classified as High. Misclassifications: 2 instances were misclassified as "Medium." None were misclassified as "Low."

Insight: The model is performing well in identifying High health scores, with only 2 misclassifications out of 22 total instances. This aligns with the high precision and recall for this class in the classification report. 2. Low Health Scores Correct Predictions: 277 instances were classified as Low. Misclassifications: 1 instance was misclassified as "High." 7 instances were misclassified as "Medium."

The model performs well in identifying Low health scores with very few misclassifications. this is supported by the high recall 0.97 for this class in the classification report, indicating that almost all Low health scores are correctly identified. 3. Medium Health Scores Correct Predictions: 109 instances were correctly classified as "Medium." Misclassifications: 37 instances were misclassified as "Low." 1 instance was misclassified as "High." The model struggles more with predicting Medium health scores, with a significant number of instances (37 out of 147) being misclassified as "Low."

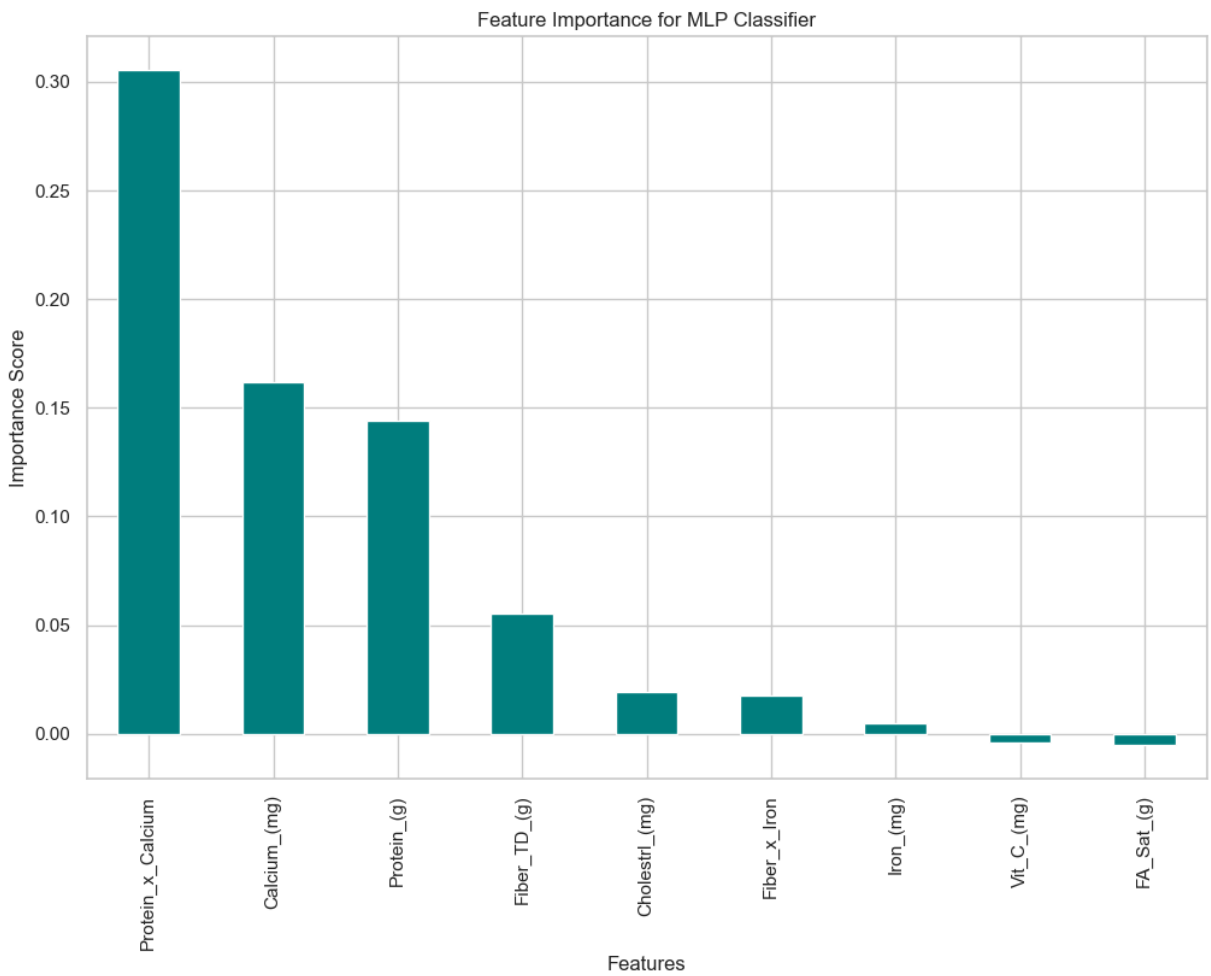
In [82]:
```
plt.figure(figsize=(10, 6))
plt.plot(mlp.loss_curve_)
plt.title("MLP Classifier Loss Curve")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.show()
```



The loss curve shows how the loss function evolves over iterations during training. Initial Loss Drop: The loss decreases within the first few iterations, which is natural for a neural network as they quickly adjust weights to reduce error. Stabilisation After ~10 Iterations: After about 10 iterations, the loss stabilises and fluctuates around a small value close to zero. This indicates that the model has come to a saturation stage and minimises loss. No

Overfitting Signs: The loss curve does not show any significant increase after the saturation point , which indicates that overfitting is not a major issue. However, this should be confirmed by evaluating performance on a separate validation or test set.

In [83]:
```
perm_importance = permutation_importance(mlp, X_test, y_test_class, n_repeats=10, r
feature_importances = pd.Series(perm_importance.importances_mean, index=X_train.col

plt.figure(figsize=(12, 8))
feature_importances.plot(kind='bar', color='teal')
plt.title("Feature Importance for MLP Classifier")
plt.xlabel("Features")
plt.ylabel("Importance Score")
plt.show()
```



Hypothesis 2: Predictive Modeling of Dietary Patterns Based on Nutrient Profiles Hypothesis: Dietary patterns can be classified into distinct types (e.g., Mediterranean, vegetarian, high-protein) based on the nutrient profiles of foods consumed, where nutrient diversity and ratios are key indicators of dietary adherence.

For 1st algorithm we are using KNN K-Nearest Neighbors is a supervised learning algorithm that classifies data points based on the labels of their nearest neighbors in the feature space. It is non-parametric and works well for smaller datasets, but its performance can degrade with high-dimensional data or large datasets due to increased computation time. Why KNN

The ability to capture complex nutrient interactions, adapt to various dietary types, and provide interpretable results makes KNN well-suited for this particular hypothesis and dataset in nutrition research.

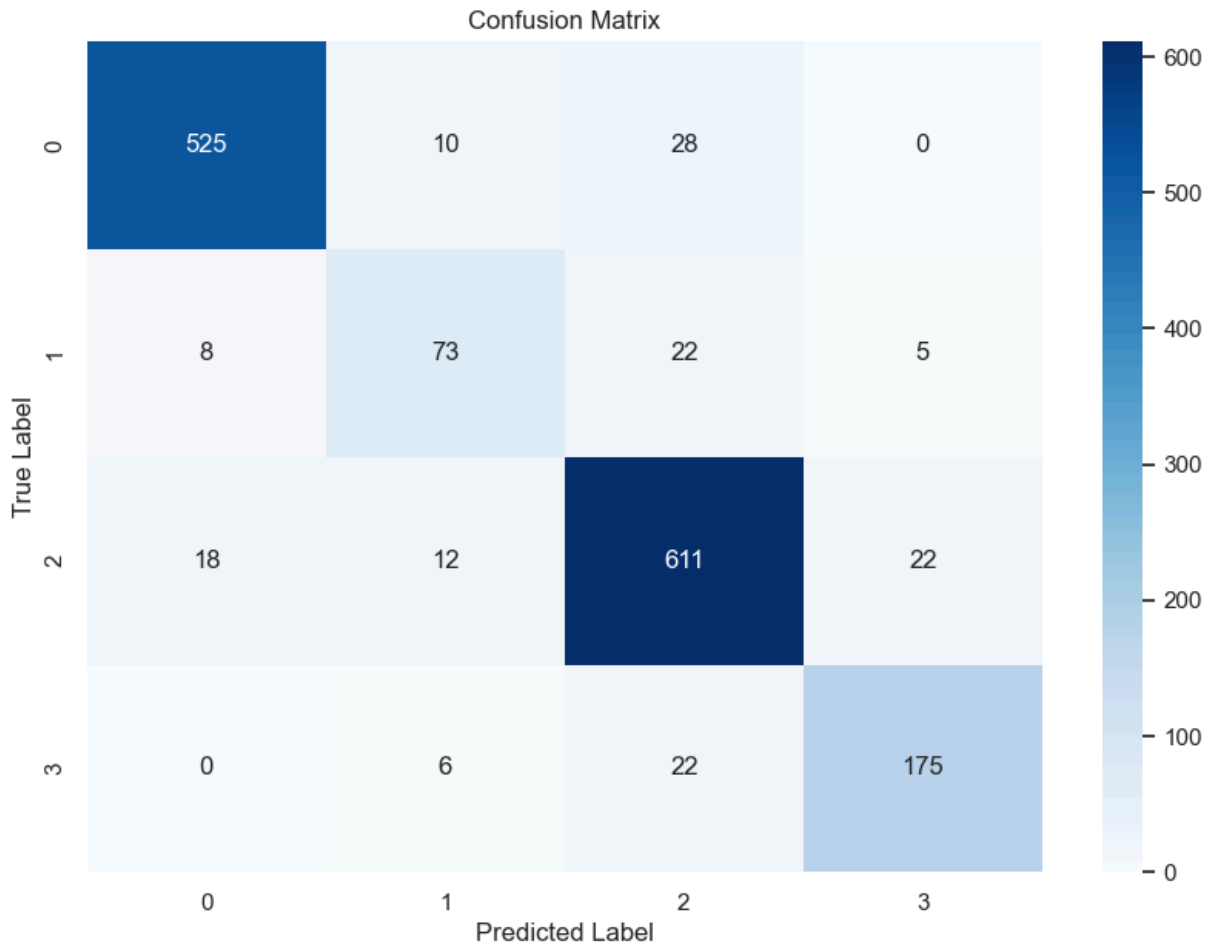
In [84]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

file_path = 'food-data.csv'
food_data = pd.read_csv(file_path)

features = ['Protein_(g)', 'Lipid_Tot_(g)', 'Carbohydrt_(g)', 'Fiber_TD_(g)', 'Calc
food_data_cleaned = food_data[features].dropna()

def create_label(row):
    protein_ratio = row['Protein_(g)'] / (row['Carbohydrt_(g)'] + row['Lipid_Tot_(g
    fiber_ratio = row['Fiber_TD_(g)'] / row['Carbohydrt_(g)'] if row['Carbohydrt_(g

    if protein_ratio > 0.5:
        return 'High-Protein'
    elif fiber_ratio > 0.1 and row['Vit_C_(mg)'] > 10:
        return 'Mediterranean'
    elif row['Lipid_Tot_(g)'] < 5 and row['Carbohydrt_(g)'] > 20:
        return 'Vegetarian'
    else:
        return 'Standard'


y = food_data_cleaned.apply(create_label, axis=1)

X_train, X_test, y_train, y_test = train_test_split(food_data_cleaned, y, test_size


knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)


y_pred = knn.predict(X_test)


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

```
               precision    recall  f1-score   support

  High-Protein       0.95      0.93      0.94       563
Mediterranean       0.72      0.68      0.70       108
     Standard       0.89      0.92      0.91       663
   Vegetarian       0.87      0.86      0.86       203

     accuracy                           0.90      1537
    macro avg       0.86      0.85      0.85      1537
 weighted avg       0.90      0.90      0.90      1537
```

From the above report we can deduce the following High-Protein Diet: Precision: 0.95 Recall: 0.93 F1-Score: 0.94 Support: 563 The High-Protein diet is classified with high precision and recall, indicating that the model is very effective at identifying this dietary pattern. This suggests that the nutrient profile of High-Protein diets is distinct and easily separable from other patterns. Mediterranean Diet: Precision: 0.72 Recall: 0.68 F1-Score: 0.70 Support: 108 The Mediterranean diet has lower precision and recall compared to other diets, indicating that it is more challenging for the model to classify correctly. This could be due to overlap in nutrient profiles with other diets or a smaller sample size (support of 108), which may affect model performance. Standard Diet: Precision: 0.89 Recall: 0.92 F1-Score: 0.91 Support: 663 The Standard diet shows strong classification performance, with high precision and recall. This suggests that the nutrient profile of a Standard diet is also well-defined and distinguishable from others. Vegetarian Diet: Precision: 0.87 Recall: 0.86 F1-Score: 0.86 Support: 203 The Vegetarian diet also has good classification performance, with balanced precision and recall values, indicating that the model can effectively identify this dietary pattern based on its nutrient profile.

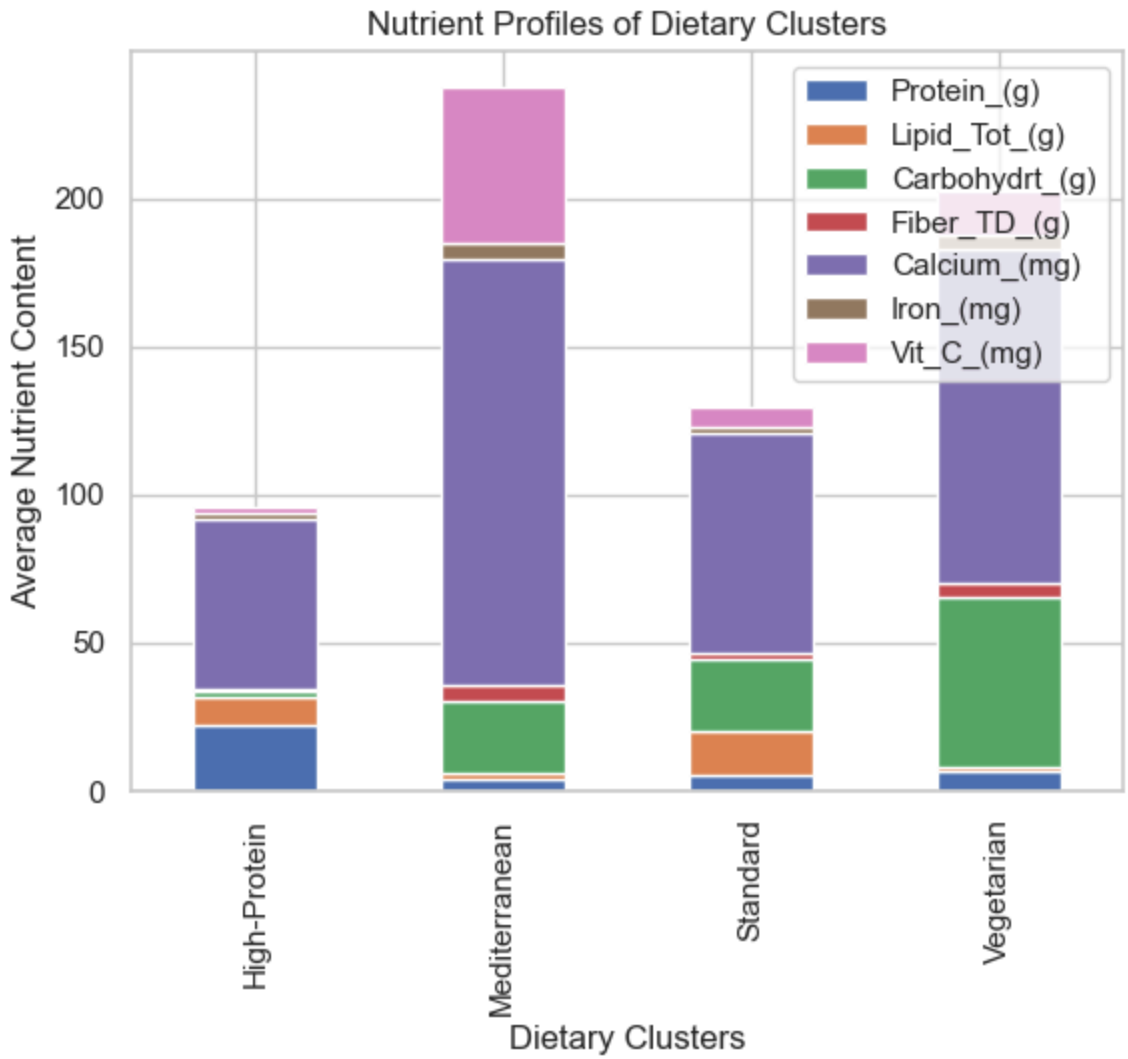
The overall accuracy of the model is 90%, which indicates strong performance. The weighted average F1-score (0.90) implies that the model performs better on larger classes (High-Protein and Standard), which have more data points.

In [85]:
```
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10,7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()
```



Class 0 : True Positives : The model correctly classified 525 instances of High-Protein diets. False Negatives : A total of 38 High-Protein diets were misclassified, with 28 being predicted as Standard and 10 as Mediterranean . False Positives: The model rarely misclassified other diets as High-Protein, which is a positive sign that this class has a distinct nutrient profile. Class 1 : True Positives : The model correctly identified 73 Mediterranean diets. False Negatives : There were 35 misclassified Mediterranean diets, with most being predicted as Standard or High-Protein . False Positives: Mediterranean diets are often confused with other classes, especially Standard and High-Protein. This suggests that the nutrient profile of Mediterranean diets may overlap with others, making them harder to distinguish. Class 2 : True Positives : The model performed very well in identifying Standard diets, with only a few misclassifications. False Negatives : A small number of Standard diets were misclassified, mostly as High-Protein or Vegetarian. Class 3 : True Positives : The model correctly classified most Vegetarian diets. False Negatives : Some Vegetarian diets were misclassified as Standard or Mediterranean.

In [86]:
```
nutrient_means = food_data_cleaned.groupby(y).mean()[features]
nutrient_means.plot(kind='bar', stacked=True)
plt.title('Nutrient Profiles of Dietary Clusters')
plt.ylabel('Average Nutrient Content')
plt.xlabel('Dietary Clusters')
plt.legend(loc='upper right')
plt.show()
```



High calcium: Calcium is by far the most dominant nutrient across all dietary clusters. This that deduces taht calcium-rich foods are common across different dietary patterns. Protein Levels: Protein levels are fairly consistent across all clusters except for a slight increase in the High-Protein diet cluster, which aligns with its classification. Fiber Variation: Fiber content varies between clusters, with the Vegetarian diet having noticeably higher fiber content compared to others. This supports the idea that plant-based diets are generally richer in fiber.

In [87]:
```
# LDA analysis for Hypothesis 2
```

In [88]:
```
from sklearn.decomposition import LatentDirichletAllocation
import pandas as pd


scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_binary = (X_scaled > 0.5).astype(int)


n_topics = 5
```

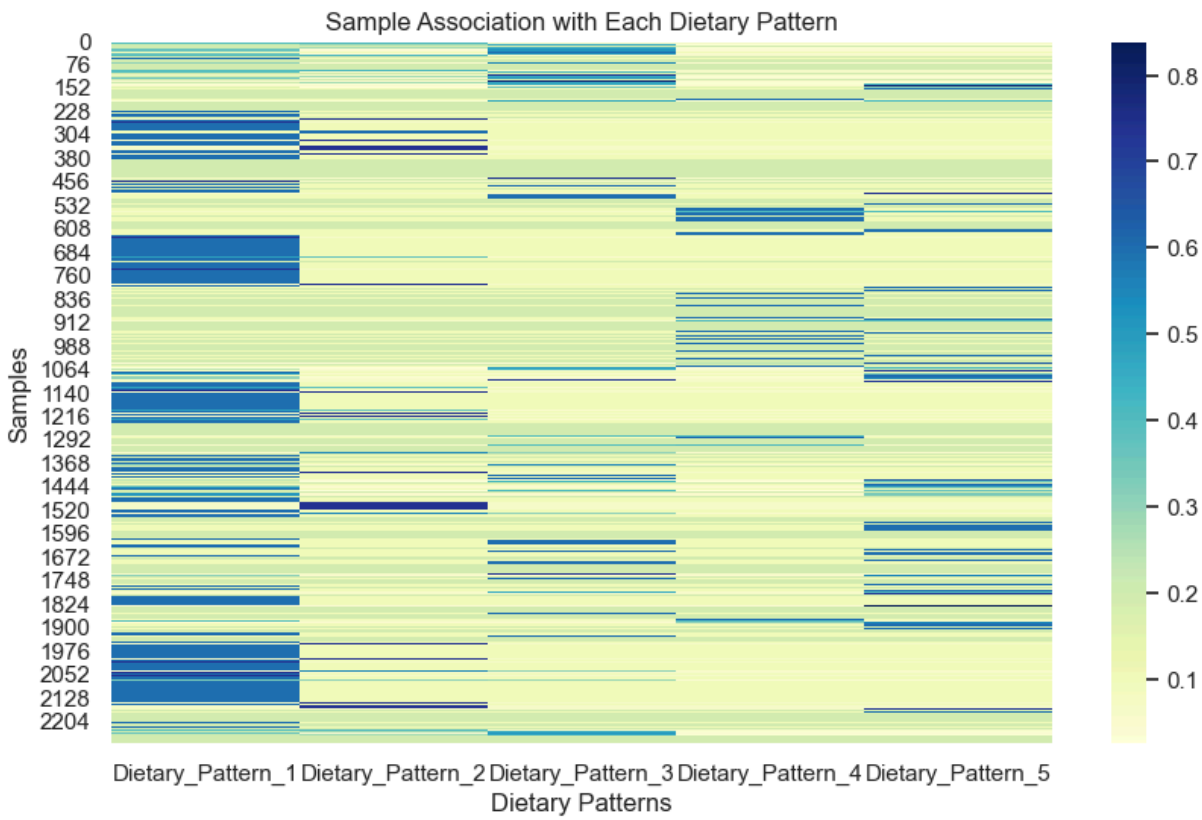
```
lda_model = LatentDirichletAllocation(n_components=n_topics, random_state=0)
W = lda_model.fit_transform(X_binary)
H = lda_model.components_


W_df = pd.DataFrame(W, columns=[f"Dietary_Pattern_{i+1}" for i in range(W.shape[1])
H_df = pd.DataFrame(H, columns=X.columns)

print("Sample Association with Each Dietary Pattern (W):\n", W_df.head())
print("\nNutrient Composition of Each Dietary Pattern (H):\n", H_df)
```

```
Sample Association with Each Dietary Pattern (W):
    Dietary_Pattern_1  Dietary_Pattern_2  Dietary_Pattern_3  Dietary_Pattern_4  \
0           0.398316           0.400017           0.067632           0.066667
1           0.398316           0.400017           0.067632           0.066667
2           0.398316           0.400017           0.067632           0.066667
3           0.364224           0.040404           0.515232           0.040000
4           0.364224           0.040404           0.515232           0.040000

   Dietary_Pattern_5
0           0.067368
1           0.067368
2           0.067368
3           0.040139
4           0.040139

Nutrient Composition of Each Dietary Pattern (H):
    Protein_(g)  Fiber_TD_(g)  Calcium_(mg)   Iron_(mg)  Vit_C_(mg)  \
0   721.568539      0.200376      0.200570    0.200623    0.200030
1   125.824488      0.200033      0.200683    0.201155    0.200013
2     0.204588      0.201326    142.196948  112.812555    0.200214
3     0.200425      0.200584      0.200498    0.201073  132.199400
4     0.201960    276.197681      0.201302   71.584594    0.200342

   Cholestrl_(mg)  FA_Sat_(g)  Protein_x_Calcium  Fiber_x_Iron
0        0.200685  273.924306           0.200819      0.200342
1      211.197769    0.203866           0.201011      0.200016
2        0.201474   46.034400         125.196871      1.121394
3        0.200055    0.200745           0.200160      0.200622
4        0.200017   35.636683           0.201139     53.277625
```

Sample Association with Dietary Patterns: Each row in W represents a sample, and each column indicates the degree of association that sample has with each dietary pattern. The values range from 0 to 1, showing the relative association of each sample with the dietary patterns. This distribution may indicate that Dietary_Pattern_5 is a dominant pattern among these samples. Nutrient Composition of Each Dietary Pattern: Each row in H corresponds to a dietary pattern, and each column reflects the average level of a nutrient within that pattern. High values in a column suggest a strong presence of a particular nutrient within that dietary pattern. Dietary_Pattern_1 shows a high concentration of Iron and Carbohydrates, indicating that this pattern might be based on iron-rich, carbohydrate-dense foods. Dietary_Pattern_2 has a high concentration of Fiber, suggesting it represents a fiber-rich dietary pattern. Dietary_Pattern_3 is defined by a high concentration of Vitamin C, likely indicating a diet rich in vitamin C sources like fruits and vegetables. Dietary_Pattern_4 and Dietary_Pattern_5 stand

out due to their high concentrations of Protein and Lipids, respectively, suggesting they represent protein-heavy and lipid-rich dietary patterns. The dietary patterns appear to reflect distinct nutrient compositions that could correspond to specific diet types or food group clusters, such as protein-heavy, fiber-rich, or carbohydrate-dense diet

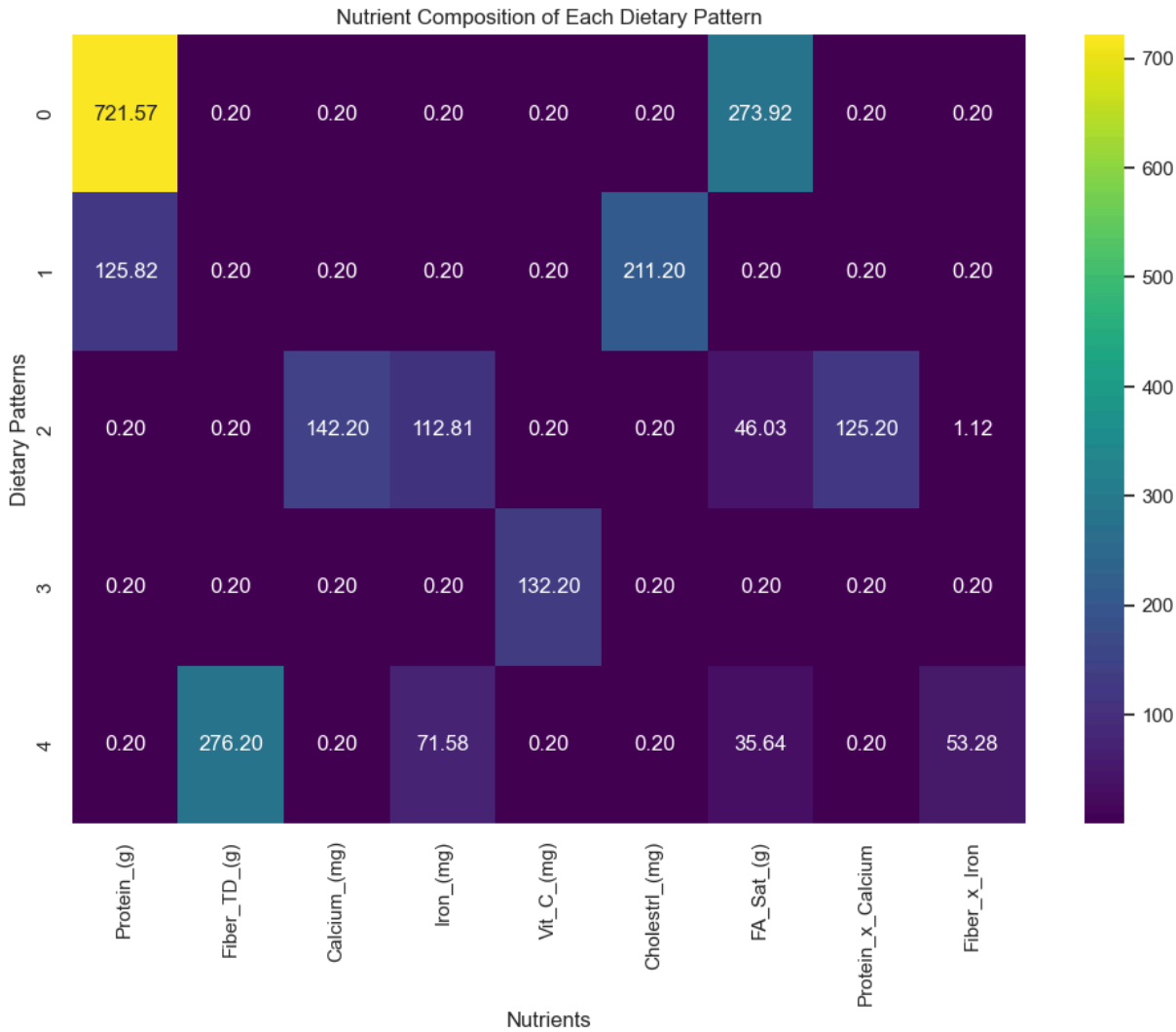
In [89]:
```
plt.figure(figsize=(10, 6))
sns.heatmap(W_df, cmap="YlGnBu", cbar=True)
plt.title("Sample Association with Each Dietary Pattern")
plt.xlabel("Dietary Patterns")
plt.ylabel("Samples")
plt.show()
```



This heatmap visualizes how strongly each sample is associated with different patterns. The color intensity represents the degree of association, where darker colors indicate stronger associations. some samples have a very strong association with a specific dietary pattern, as indicated by the dark blue areas. This suggests that these samples fit very well within those dietary patterns. Mixed Associations: Other samples show more moderate associations across multiple patterns, represented by lighter shades. This indicates that these samples may not fit neatly into one specific dietary pattern and could exhibit characteristics of multiple patterns.

In [90]:
```
plt.figure(figsize=(12, 8))
sns.heatmap(H_df, cmap="viridis", annot=True, fmt=".2f", cbar=True)
plt.title("Nutrient Composition of Each Dietary Pattern")
plt.xlabel("Nutrients")
```

```
plt.ylabel("Dietary Patterns")
plt.show()
```



This heatmap displays the nutrient composition for five dietary patterns across various nutrients. Key insights include: Dietary Pattern 0: This pattern has the highest levels of Calcium (721.57 mg) and a significant amount of Vitamin C (273.92 mg), making it a calcium-rich and vitamin C-heavy diet. This could represent a diet high in dairy products and fruits or vegetables. Dietary Pattern 1: This pattern has a moderate level of Calcium (125.82 mg) and Vitamin C (211.20 mg), suggesting a balanced diet with some emphasis on these nutrients but not as extreme as Pattern 0. Dietary Pattern 2: This pattern stands out for its relatively high levels of Iron (142.20 mg) and Vitamin C (112.81 mg), which could indicate a diet rich in iron-containing foods like leafy greens or legumes. Dietary Pattern 3: This pattern has elevated levels of Iron (132.20 mg) but lower levels of other nutrients, indicating a more iron-focused diet, potentially from plant-based sources or fortified foods. Dietary Pattern 4: This pattern shows high levels of Fiber (276.20 g) and moderate amounts of Iron (71.58 mg), suggesting it may represent a fiber-rich diet, likely plant-based or vegetarian.

In [91]:
```
top_n = 5
```

```python
for i in range(H_df.shape[0]):
    pattern_nutrients = H_df.iloc[i].sort_values(ascending=False)[:top_n]
    print(f"\nTop {top_n} Nutrients for Dietary Pattern {i + 1}:\n{pattern_nutrient
```

```
Top 5 Nutrients for Dietary Pattern 1:
Protein_(g)          721.568539
FA_Sat_(g)           273.924306
Protein_x_Calcium      0.200819
Cholestrl_(mg)         0.200685
Iron_(mg)              0.200623
Name: 0, dtype: float64

Top 5 Nutrients for Dietary Pattern 2:
Cholestrl_(mg)       211.197769
Protein_(g)          125.824488
FA_Sat_(g)             0.203866
Iron_(mg)              0.201155
Protein_x_Calcium      0.201011
Name: 1, dtype: float64

Top 5 Nutrients for Dietary Pattern 3:
Calcium_(mg)         142.196948
Protein_x_Calcium    125.196871
Iron_(mg)            112.812555
FA_Sat_(g)            46.034400
Fiber_x_Iron           1.121394
Name: 2, dtype: float64

Top 5 Nutrients for Dietary Pattern 4:
Vit_C_(mg)      132.199400
Iron_(mg)         0.201073
FA_Sat_(g)        0.200745
Fiber_x_Iron      0.200622
Fiber_TD_(g)      0.200584
Name: 3, dtype: float64

Top 5 Nutrients for Dietary Pattern 5:
Fiber_TD_(g)    276.197681
Iron_(mg)        71.584594
Fiber_x_Iron     53.277625
FA_Sat_(g)       35.636683
Protein_(g)       0.201960
Name: 4, dtype: float64
```

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from scipy.stats import pearsonr
import sklearn
```

Hypothesis -1

Foods rich in fatty acids, antioxidants, B-vitamins, and amino acids can improve cognitive performance, particularly when these nutrients are combined in specific ratios.

In [6]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import mean_absolute_error, r2_score
from sklearn.metrics import accuracy_score, classification_report, confus
import matplotlib.pyplot as plt

# Load the dataset
df = pd.read_csv('food-data.csv')
df = df.dropna()

# Select relevant features
features = ['Protein_(g)', 'Lipid_Tot_(g)', 'Carbohydrt_(g)', 'Fiber_TD_(
            'Calcium_(mg)', 'Iron_(mg)', 'Magnesium_(mg)', 'Phosphorus_(m
            'Sodium_(mg)', 'Zinc_(mg)', 'Copper_mg)', 'Manganese_(mg)', '
            'Vit_C_(mg)', 'Thiamin_(mg)', 'Riboflavin_(mg)', 'Niacin_(mg)
            'Vit_B6_(mg)', 'Folate_Tot_(µg)', 'Choline_Tot_ (mg)', 'Vit_B
            'Vit_A_IU', 'Vit_E_(mg)', 'Vit_D_µg', 'Vit_K_(µg)']

# Select target variables (nutrients hypothesized to enhance cognitive pe
targets = ['FA_Poly_(g)', 'Vit_C_(mg)', 'Thiamin_(mg)', 'Riboflavin_(mg)'
           'Vit_B6_(mg)', 'Folate_Tot_(µg)', 'Vit_B12_(µg)']

# Prepare the data
X = df[features]
y = df[targets]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# Create the neural network model
model = Sequential([
    Dense(128, activation='relu', input_shape=(len(features),)),
    Dropout(0.3),
    Dense(64, activation='relu'),
    Dropout(0.3),
    Dense(32, activation='relu'),
    Dense(len(targets))
])

# Compile with a smaller learning rate
model.compile(optimizer=Adam(learning_rate=0.0005),
              loss='mean_absolute_error',  # Change loss for classificati
              metrics=['accuracy'])  # Add accuracy metric

history = model.fit(X_train_scaled, y_train, epochs=200, batch_size=16, v

# Evaluate and predict
test_loss, test_accuracy = model.evaluate(X_test_scaled, y_test)
print(f"Test Loss: {test_loss}")
print(f"Test Accuracy: {test_accuracy:.4f}")


predictions = model.predict(X_test_scaled)
predicted_classes = np.argmax(predictions, axis=1)  # Get predicted class

# Calculate accuracy score
true_classes = np.argmax(y_test.values, axis=1)  # Assuming y_test is one
accuracy = accuracy_score(true_classes, predicted_classes)
print(f"Accuracy Score: {accuracy:.4f}")

#evaluation, prediction
test_loss_value = model.evaluate(X_test_scaled, y_test)
print(f"Test Loss: {test_loss_value}")


#MAE and R2 score for each target varible
mae_scores, r2_scores = [], []

for i, target in enumerate(targets):
    mae = mean_absolute_error(y_test.iloc[:, i], predictions[:, i])
    r2 = r2_score(y_test.iloc[:, i], predictions[:, i])
    mae_scores.append(mae)
    r2_scores.append(r2)
    print(f"{target}:")
    print(f"  MAE: {mae:.4f}")
    print(f"  R2 Score: {r2:.4f}")
    print()

avg_mae = np.mean(mae_scores)
avg_r2 = np.mean(r2_scores)

print(f"Average MAE: {avg_mae:.4f}")
print(f"Average R2 Score: {avg_r2:.4f}")
print(f"Accuracy Score: {accuracy:.4f}")

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
```

```
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()

plt.subplot(1, 2, 2)
plt.bar(targets, r2_scores)
plt.title('R2 Scores by Target')
plt.ylabel('R2 Score')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

predictions = model.predict(X_test_scaled)
#predicted classes
predicted_classes = np.argmax(predictions, axis=1)

# Get true class labels
right_classes = np.argmax(y_test, axis=1)

# Compute the confusion matrix
cm = confusion_matrix(right_classes, predicted_classes)

# Define your class names
classnames = ['Class 0', 'Class 1', 'Class 2', 'Class 3', 'Class 4', 'Cla


dplot = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classn

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
dplot.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.show()
```

```
Epoch 1/200
```

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pac
kages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `inpu
t_shape`/`input_dim` argument to a layer. When using Sequential models, pr
efer using an `Input(shape)` object as the first layer in the model instea
d.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 796us/step - accuracy: 0.1143 - loss: 5.9562
- val_accuracy: 0.4121 - val_loss: 5.0166
Epoch 2/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 391us/step - accuracy: 0.4640 - loss: 5.6776
- val_accuracy: 0.7115 - val_loss: 4.5715
Epoch 3/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 391us/step - accuracy: 0.6705 - loss: 4.9220
- val_accuracy: 0.6456 - val_loss: 4.3355
Epoch 4/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 403us/step - accuracy: 0.6567 - loss: 5.2210
- val_accuracy: 0.6456 - val_loss: 3.9912
Epoch 5/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 381us/step - accuracy: 0.6242 - loss: 4.2943
- val_accuracy: 0.6456 - val_loss: 3.4779
Epoch 6/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 406us/step - accuracy: 0.6456 - loss: 3.7998
- val_accuracy: 0.6841 - val_loss: 2.9729
Epoch 7/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 383us/step - accuracy: 0.6653 - loss: 3.2747
- val_accuracy: 0.7500 - val_loss: 2.5482
Epoch 8/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 380us/step - accuracy: 0.6759 - loss: 3.0601
- val_accuracy: 0.7143 - val_loss: 2.3113
Epoch 9/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 388us/step - accuracy: 0.6994 - loss: 2.6294
- val_accuracy: 0.7363 - val_loss: 2.0751
Epoch 10/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 385us/step - accuracy: 0.7322 - loss: 2.2922
- val_accuracy: 0.7418 - val_loss: 1.9185
Epoch 11/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 384us/step - accuracy: 0.7180 - loss: 2.2976
- val_accuracy: 0.7280 - val_loss: 1.8235
Epoch 12/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 393us/step - accuracy: 0.7399 - loss: 2.5299
- val_accuracy: 0.7555 - val_loss: 1.6699
Epoch 13/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 386us/step - accuracy: 0.7207 - loss: 2.0306
- val_accuracy: 0.7637 - val_loss: 1.6299
Epoch 14/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 380us/step - accuracy: 0.7248 - loss: 2.6760
- val_accuracy: 0.7692 - val_loss: 1.6140
Epoch 15/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 534us/step - accuracy: 0.7313 - loss: 2.0717
- val_accuracy: 0.7500 - val_loss: 1.5355
Epoch 16/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 368us/step - accuracy: 0.7526 - loss: 2.0957
- val_accuracy: 0.7830 - val_loss: 1.4432
Epoch 17/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 373us/step - accuracy: 0.7451 - loss: 2.0642
- val_accuracy: 0.7747 - val_loss: 1.4086
Epoch 18/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 368us/step - accuracy: 0.7486 - loss: 1.8667
- val_accuracy: 0.7500 - val_loss: 1.4036
Epoch 19/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 368us/step - accuracy: 0.7432 - loss: 1.9072
- val_accuracy: 0.7418 - val_loss: 1.3832
Epoch 20/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 370us/step - accuracy: 0.7453 - loss: 1.9283
- val_accuracy: 0.7830 - val_loss: 1.3806
Epoch 21/200
```

```
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 371us/step - accuracy: 0.7515 - loss: 2.1473
- val_accuracy: 0.7830 - val_loss: 1.3560
Epoch 22/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 380us/step - accuracy: 0.7572 - loss: 2.2196
- val_accuracy: 0.7555 - val_loss: 1.3547
Epoch 23/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 373us/step - accuracy: 0.7451 - loss: 2.1443
- val_accuracy: 0.7610 - val_loss: 1.4310
Epoch 24/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 383us/step - accuracy: 0.7539 - loss: 1.8312
- val_accuracy: 0.7555 - val_loss: 1.3725
Epoch 25/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 371us/step - accuracy: 0.7663 - loss: 2.1884
- val_accuracy: 0.7610 - val_loss: 1.3215
Epoch 26/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 378us/step - accuracy: 0.7464 - loss: 1.8044
- val_accuracy: 0.7692 - val_loss: 1.2959
Epoch 27/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 393us/step - accuracy: 0.7844 - loss: 1.9251
- val_accuracy: 0.7830 - val_loss: 1.2664
Epoch 28/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 380us/step - accuracy: 0.7549 - loss: 1.8981
- val_accuracy: 0.7555 - val_loss: 1.3160
Epoch 29/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 374us/step - accuracy: 0.7722 - loss: 1.8715
- val_accuracy: 0.7967 - val_loss: 1.2557
Epoch 30/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 391us/step - accuracy: 0.7548 - loss: 1.7994
- val_accuracy: 0.8077 - val_loss: 1.2411
Epoch 31/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 367us/step - accuracy: 0.7630 - loss: 1.8473
- val_accuracy: 0.8022 - val_loss: 1.2180
Epoch 32/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 368us/step - accuracy: 0.7861 - loss: 1.5803
- val_accuracy: 0.8022 - val_loss: 1.2723
Epoch 33/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 366us/step - accuracy: 0.7534 - loss: 1.8578
- val_accuracy: 0.8132 - val_loss: 1.1988
Epoch 34/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 367us/step - accuracy: 0.7909 - loss: 1.6027
- val_accuracy: 0.8132 - val_loss: 1.2010
Epoch 35/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 361us/step - accuracy: 0.7836 - loss: 1.8065
- val_accuracy: 0.8159 - val_loss: 1.1619
Epoch 36/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 368us/step - accuracy: 0.7689 - loss: 1.6792
- val_accuracy: 0.8214 - val_loss: 1.1176
Epoch 37/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 359us/step - accuracy: 0.7775 - loss: 1.5516
- val_accuracy: 0.8269 - val_loss: 1.1022
Epoch 38/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 354us/step - accuracy: 0.7829 - loss: 1.8672
- val_accuracy: 0.8132 - val_loss: 1.0819
Epoch 39/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 356us/step - accuracy: 0.7912 - loss: 1.8594
- val_accuracy: 0.8352 - val_loss: 1.1109
Epoch 40/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 360us/step - accuracy: 0.7801 - loss: 1.5354
- val_accuracy: 0.8324 - val_loss: 1.2029
Epoch 41/200
```

```
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 611us/step - accuracy: 0.8114 - loss: 1.6731
- val_accuracy: 0.8434 - val_loss: 1.0656
Epoch 42/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 370us/step - accuracy: 0.8179 - loss: 1.5604
- val_accuracy: 0.8599 - val_loss: 1.0344
Epoch 43/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 362us/step - accuracy: 0.7988 - loss: 1.5479
- val_accuracy: 0.8571 - val_loss: 1.0915
Epoch 44/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 360us/step - accuracy: 0.8213 - loss: 1.5766
- val_accuracy: 0.8571 - val_loss: 1.0094
Epoch 45/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 357us/step - accuracy: 0.8093 - loss: 1.6694
- val_accuracy: 0.8599 - val_loss: 0.9785
Epoch 46/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 366us/step - accuracy: 0.8405 - loss: 2.0710
- val_accuracy: 0.8571 - val_loss: 0.9412
Epoch 47/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 373us/step - accuracy: 0.8218 - loss: 1.8788
- val_accuracy: 0.8516 - val_loss: 0.9604
Epoch 48/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 372us/step - accuracy: 0.8133 - loss: 1.4093
- val_accuracy: 0.8626 - val_loss: 0.8700
Epoch 49/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 362us/step - accuracy: 0.8298 - loss: 1.4681
- val_accuracy: 0.8736 - val_loss: 0.8835
Epoch 50/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 353us/step - accuracy: 0.8218 - loss: 1.4826
- val_accuracy: 0.8709 - val_loss: 0.8774
Epoch 51/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 358us/step - accuracy: 0.8239 - loss: 1.5002
- val_accuracy: 0.8764 - val_loss: 0.8305
Epoch 52/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 367us/step - accuracy: 0.8452 - loss: 1.2912
- val_accuracy: 0.8654 - val_loss: 0.8668
Epoch 53/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 372us/step - accuracy: 0.8178 - loss: 1.3462
- val_accuracy: 0.8654 - val_loss: 0.8037
Epoch 54/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 358us/step - accuracy: 0.8322 - loss: 1.3001
- val_accuracy: 0.8819 - val_loss: 0.7775
Epoch 55/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 376us/step - accuracy: 0.8538 - loss: 1.2950
- val_accuracy: 0.8819 - val_loss: 0.7592
Epoch 56/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 359us/step - accuracy: 0.8558 - loss: 1.5350
- val_accuracy: 0.8791 - val_loss: 0.6804
Epoch 57/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 355us/step - accuracy: 0.8455 - loss: 1.3335
- val_accuracy: 0.8819 - val_loss: 0.7250
Epoch 58/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 364us/step - accuracy: 0.8536 - loss: 1.1504
- val_accuracy: 0.8764 - val_loss: 0.7406
Epoch 59/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 358us/step - accuracy: 0.8412 - loss: 1.3756
- val_accuracy: 0.8791 - val_loss: 0.7425
Epoch 60/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 362us/step - accuracy: 0.8381 - loss: 1.2701
- val_accuracy: 0.8846 - val_loss: 0.6658
Epoch 61/200
```

```
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 370us/step - accuracy: 0.8349 - loss: 1.2318
- val_accuracy: 0.8819 - val_loss: 0.6948
Epoch 62/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 361us/step - accuracy: 0.8607 - loss: 1.3125
- val_accuracy: 0.8956 - val_loss: 0.6789
Epoch 63/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 370us/step - accuracy: 0.8469 - loss: 1.1645
- val_accuracy: 0.8819 - val_loss: 0.7310
Epoch 64/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 381us/step - accuracy: 0.8460 - loss: 1.2477
- val_accuracy: 0.9011 - val_loss: 0.7465
Epoch 65/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 362us/step - accuracy: 0.8744 - loss: 1.2443
- val_accuracy: 0.8984 - val_loss: 0.7135
Epoch 66/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 369us/step - accuracy: 0.8501 - loss: 1.1548
- val_accuracy: 0.9038 - val_loss: 0.7616
Epoch 67/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 362us/step - accuracy: 0.8504 - loss: 1.1525
- val_accuracy: 0.8984 - val_loss: 0.7462
Epoch 68/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 360us/step - accuracy: 0.8495 - loss: 1.2856
- val_accuracy: 0.8929 - val_loss: 0.6762
Epoch 69/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 641us/step - accuracy: 0.8406 - loss: 1.1786
- val_accuracy: 0.9011 - val_loss: 0.6497
Epoch 70/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 372us/step - accuracy: 0.8661 - loss: 1.3305
- val_accuracy: 0.8901 - val_loss: 0.6522
Epoch 71/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 366us/step - accuracy: 0.8606 - loss: 1.1606
- val_accuracy: 0.8874 - val_loss: 0.6690
Epoch 72/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 364us/step - accuracy: 0.8540 - loss: 1.1973
- val_accuracy: 0.8901 - val_loss: 0.6736
Epoch 73/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 365us/step - accuracy: 0.8610 - loss: 1.1744
- val_accuracy: 0.9038 - val_loss: 0.6357
Epoch 74/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 364us/step - accuracy: 0.8604 - loss: 1.1958
- val_accuracy: 0.8846 - val_loss: 0.6193
Epoch 75/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 359us/step - accuracy: 0.8707 - loss: 1.0980
- val_accuracy: 0.8929 - val_loss: 0.7115
Epoch 76/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 360us/step - accuracy: 0.8636 - loss: 1.3100
- val_accuracy: 0.8929 - val_loss: 0.5919
Epoch 77/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 376us/step - accuracy: 0.8614 - loss: 1.0895
- val_accuracy: 0.8901 - val_loss: 0.6690
Epoch 78/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 365us/step - accuracy: 0.8611 - loss: 1.5079
- val_accuracy: 0.8929 - val_loss: 0.6453
Epoch 79/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 365us/step - accuracy: 0.8649 - loss: 1.3772
- val_accuracy: 0.8984 - val_loss: 0.5886
Epoch 80/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 367us/step - accuracy: 0.8734 - loss: 1.1584
- val_accuracy: 0.8984 - val_loss: 0.6336
Epoch 81/200
```

```
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 361us/step - accuracy: 0.8548 - loss: 1.1201
- val_accuracy: 0.8929 - val_loss: 0.6376
Epoch 82/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 370us/step - accuracy: 0.8530 - loss: 1.3391
- val_accuracy: 0.9121 - val_loss: 0.6426
Epoch 83/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 385us/step - accuracy: 0.8591 - loss: 1.2484
- val_accuracy: 0.9011 - val_loss: 0.6885
Epoch 84/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 377us/step - accuracy: 0.8685 - loss: 1.3545
- val_accuracy: 0.8901 - val_loss: 0.5788
Epoch 85/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 644us/step - accuracy: 0.8808 - loss: 1.1556
- val_accuracy: 0.9011 - val_loss: 0.5991
Epoch 86/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 376us/step - accuracy: 0.8729 - loss: 1.1408
- val_accuracy: 0.8901 - val_loss: 0.6616
Epoch 87/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 373us/step - accuracy: 0.8465 - loss: 1.0286
- val_accuracy: 0.9148 - val_loss: 0.7314
Epoch 88/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 373us/step - accuracy: 0.8473 - loss: 1.0812
- val_accuracy: 0.9121 - val_loss: 0.5596
Epoch 89/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 373us/step - accuracy: 0.8701 - loss: 1.2389
- val_accuracy: 0.9066 - val_loss: 0.5707
Epoch 90/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 377us/step - accuracy: 0.8648 - loss: 1.3301
- val_accuracy: 0.9011 - val_loss: 0.6010
Epoch 91/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 383us/step - accuracy: 0.8745 - loss: 1.2399
- val_accuracy: 0.8846 - val_loss: 0.6739
Epoch 92/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 387us/step - accuracy: 0.8629 - loss: 1.2051
- val_accuracy: 0.8956 - val_loss: 0.7215
Epoch 93/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 625us/step - accuracy: 0.8573 - loss: 1.2293
- val_accuracy: 0.8874 - val_loss: 0.7653
Epoch 94/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 381us/step - accuracy: 0.8768 - loss: 1.0459
- val_accuracy: 0.8901 - val_loss: 0.6122
Epoch 95/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 383us/step - accuracy: 0.8631 - loss: 1.1614
- val_accuracy: 0.9038 - val_loss: 0.5779
Epoch 96/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 380us/step - accuracy: 0.8803 - loss: 1.0957
- val_accuracy: 0.9066 - val_loss: 0.5587
Epoch 97/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 373us/step - accuracy: 0.8736 - loss: 1.0658
- val_accuracy: 0.8929 - val_loss: 0.5506
Epoch 98/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 372us/step - accuracy: 0.8552 - loss: 1.1311
- val_accuracy: 0.8956 - val_loss: 0.6019
Epoch 99/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 406us/step - accuracy: 0.8734 - loss: 1.0974
- val_accuracy: 0.8929 - val_loss: 0.6566
Epoch 100/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 659us/step - accuracy: 0.8657 - loss: 1.5119
- val_accuracy: 0.8984 - val_loss: 0.7349
Epoch 101/200
```

```
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 386us/step - accuracy: 0.8721 - loss: 1.1145
- val_accuracy: 0.9121 - val_loss: 0.5508
Epoch 102/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 406us/step - accuracy: 0.8642 - loss: 1.2059
- val_accuracy: 0.9093 - val_loss: 0.5476
Epoch 103/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 385us/step - accuracy: 0.8697 - loss: 1.0766
- val_accuracy: 0.9066 - val_loss: 0.6040
Epoch 104/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 393us/step - accuracy: 0.8873 - loss: 1.0665
- val_accuracy: 0.8874 - val_loss: 0.6250
Epoch 105/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 394us/step - accuracy: 0.8691 - loss: 1.0838
- val_accuracy: 0.9011 - val_loss: 0.5722
Epoch 106/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 374us/step - accuracy: 0.8729 - loss: 1.0427
- val_accuracy: 0.8984 - val_loss: 0.7789
Epoch 107/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 683us/step - accuracy: 0.8647 - loss: 1.2183
- val_accuracy: 0.9093 - val_loss: 0.6985
Epoch 108/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 387us/step - accuracy: 0.8585 - loss: 1.0965
- val_accuracy: 0.9121 - val_loss: 0.6459
Epoch 109/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 385us/step - accuracy: 0.8677 - loss: 1.2515
- val_accuracy: 0.8929 - val_loss: 0.5643
Epoch 110/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 384us/step - accuracy: 0.8761 - loss: 1.3543
- val_accuracy: 0.8929 - val_loss: 0.5487
Epoch 111/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 376us/step - accuracy: 0.8661 - loss: 1.1272
- val_accuracy: 0.8984 - val_loss: 0.5468
Epoch 112/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 376us/step - accuracy: 0.8756 - loss: 1.1770
- val_accuracy: 0.8929 - val_loss: 0.5320
Epoch 113/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 375us/step - accuracy: 0.8473 - loss: 1.1046
- val_accuracy: 0.8929 - val_loss: 0.5393
Epoch 114/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 376us/step - accuracy: 0.8786 - loss: 1.1160
- val_accuracy: 0.9038 - val_loss: 0.5179
Epoch 115/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 668us/step - accuracy: 0.8755 - loss: 1.1011
- val_accuracy: 0.9011 - val_loss: 0.5747
Epoch 116/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 381us/step - accuracy: 0.8586 - loss: 1.0777
- val_accuracy: 0.9066 - val_loss: 0.5982
Epoch 117/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 381us/step - accuracy: 0.8719 - loss: 1.1384
- val_accuracy: 0.9038 - val_loss: 0.5107
Epoch 118/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 388us/step - accuracy: 0.8667 - loss: 1.1587
- val_accuracy: 0.9066 - val_loss: 0.6366
Epoch 119/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 384us/step - accuracy: 0.8807 - loss: 1.0751
- val_accuracy: 0.9093 - val_loss: 0.7849
Epoch 120/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 387us/step - accuracy: 0.8883 - loss: 1.0801
- val_accuracy: 0.9038 - val_loss: 0.5598
Epoch 121/200
```

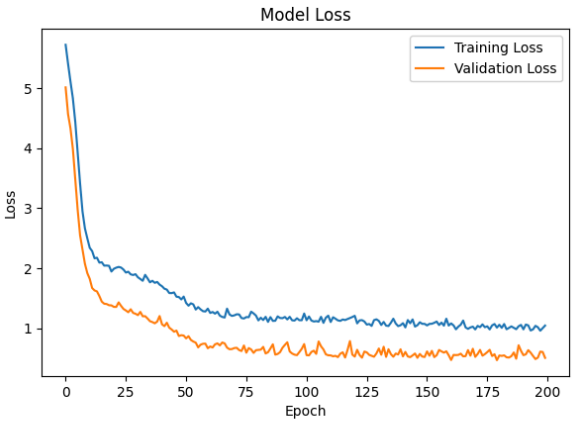
```
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 386us/step - accuracy: 0.8575 - loss: 1.1603
- val_accuracy: 0.9038 - val_loss: 0.5269
Epoch 122/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 656us/step - accuracy: 0.8866 - loss: 1.0728
- val_accuracy: 0.9038 - val_loss: 0.6396
Epoch 123/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 398us/step - accuracy: 0.8758 - loss: 1.3129
- val_accuracy: 0.9148 - val_loss: 0.5523
Epoch 124/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 395us/step - accuracy: 0.8731 - loss: 1.2680
- val_accuracy: 0.9203 - val_loss: 0.5085
Epoch 125/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 393us/step - accuracy: 0.8566 - loss: 0.9940
- val_accuracy: 0.9203 - val_loss: 0.6107
Epoch 126/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 416us/step - accuracy: 0.8843 - loss: 1.0323
- val_accuracy: 0.9038 - val_loss: 0.5964
Epoch 127/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 602us/step - accuracy: 0.8594 - loss: 1.1226
- val_accuracy: 0.8984 - val_loss: 0.5553
Epoch 128/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 389us/step - accuracy: 0.8547 - loss: 1.0450
- val_accuracy: 0.9038 - val_loss: 0.5406
Epoch 129/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 378us/step - accuracy: 0.8767 - loss: 1.1381
- val_accuracy: 0.9121 - val_loss: 0.5244
Epoch 130/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 407us/step - accuracy: 0.8727 - loss: 1.0926
- val_accuracy: 0.9011 - val_loss: 0.5796
Epoch 131/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 382us/step - accuracy: 0.8953 - loss: 1.0707
- val_accuracy: 0.9038 - val_loss: 0.6616
Epoch 132/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 390us/step - accuracy: 0.8869 - loss: 1.1046
- val_accuracy: 0.9038 - val_loss: 0.5576
Epoch 133/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 654us/step - accuracy: 0.8770 - loss: 1.0748
- val_accuracy: 0.8956 - val_loss: 0.6905
Epoch 134/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 396us/step - accuracy: 0.9016 - loss: 1.0272
- val_accuracy: 0.8929 - val_loss: 0.5216
Epoch 135/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 383us/step - accuracy: 0.8806 - loss: 0.9699
- val_accuracy: 0.9148 - val_loss: 0.6493
Epoch 136/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 387us/step - accuracy: 0.8811 - loss: 1.0411
- val_accuracy: 0.9011 - val_loss: 0.5671
Epoch 137/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 379us/step - accuracy: 0.8662 - loss: 1.0531
- val_accuracy: 0.8956 - val_loss: 0.5239
Epoch 138/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 382us/step - accuracy: 0.8810 - loss: 1.0248
- val_accuracy: 0.9093 - val_loss: 0.5776
Epoch 139/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 691us/step - accuracy: 0.8887 - loss: 1.0052
- val_accuracy: 0.9093 - val_loss: 0.5617
Epoch 140/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 387us/step - accuracy: 0.8785 - loss: 0.9927
- val_accuracy: 0.9066 - val_loss: 0.6485
Epoch 141/200
```

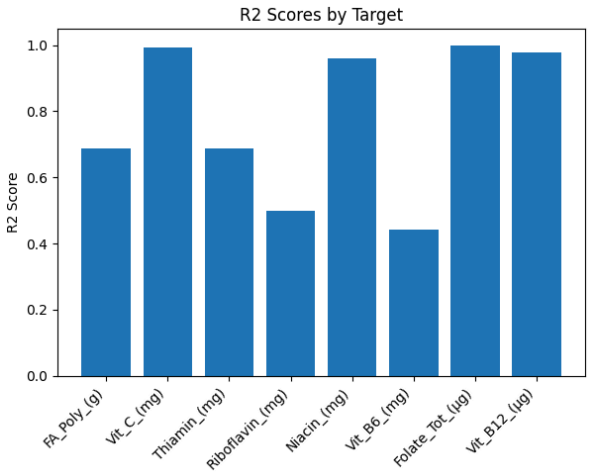
```
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 383us/step - accuracy: 0.8871 - loss: 1.0291
- val_accuracy: 0.9176 - val_loss: 0.5305
Epoch 142/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 378us/step - accuracy: 0.8649 - loss: 0.9615
- val_accuracy: 0.9176 - val_loss: 0.6042
Epoch 143/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 392us/step - accuracy: 0.8947 - loss: 1.1454
- val_accuracy: 0.9038 - val_loss: 0.5292
Epoch 144/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 507us/step - accuracy: 0.8834 - loss: 1.2064
- val_accuracy: 0.9121 - val_loss: 0.5176
Epoch 145/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 452us/step - accuracy: 0.8611 - loss: 1.1986
- val_accuracy: 0.9176 - val_loss: 0.5460
Epoch 146/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 387us/step - accuracy: 0.8802 - loss: 1.0192
- val_accuracy: 0.9093 - val_loss: 0.5781
Epoch 147/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 397us/step - accuracy: 0.8703 - loss: 1.0881
- val_accuracy: 0.9148 - val_loss: 0.5063
Epoch 148/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 376us/step - accuracy: 0.8713 - loss: 1.1469
- val_accuracy: 0.9011 - val_loss: 0.5710
Epoch 149/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 402us/step - accuracy: 0.8789 - loss: 1.0609
- val_accuracy: 0.9038 - val_loss: 0.5275
Epoch 150/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 517us/step - accuracy: 0.8597 - loss: 0.9770
- val_accuracy: 0.9093 - val_loss: 0.5199
Epoch 151/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 418us/step - accuracy: 0.8730 - loss: 0.9986
- val_accuracy: 0.8901 - val_loss: 0.6131
Epoch 152/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 394us/step - accuracy: 0.8783 - loss: 1.0363
- val_accuracy: 0.9066 - val_loss: 0.5411
Epoch 153/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 408us/step - accuracy: 0.8919 - loss: 0.9450
- val_accuracy: 0.9148 - val_loss: 0.6220
Epoch 154/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 618us/step - accuracy: 0.8823 - loss: 1.2534
- val_accuracy: 0.9093 - val_loss: 0.5180
Epoch 155/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 380us/step - accuracy: 0.8970 - loss: 1.0660
- val_accuracy: 0.9093 - val_loss: 0.5504
Epoch 156/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 385us/step - accuracy: 0.8833 - loss: 1.0295
- val_accuracy: 0.9121 - val_loss: 0.6374
Epoch 157/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 378us/step - accuracy: 0.8810 - loss: 1.1284
- val_accuracy: 0.9121 - val_loss: 0.6134
Epoch 158/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 400us/step - accuracy: 0.8933 - loss: 0.9414
- val_accuracy: 0.9011 - val_loss: 0.5930
Epoch 159/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 507us/step - accuracy: 0.8541 - loss: 1.1184
- val_accuracy: 0.9203 - val_loss: 0.6150
Epoch 160/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 391us/step - accuracy: 0.8876 - loss: 1.1905
- val_accuracy: 0.9176 - val_loss: 0.5486
Epoch 161/200
```

```
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 393us/step - accuracy: 0.8847 - loss: 1.1580
- val_accuracy: 0.9203 - val_loss: 0.4695
Epoch 162/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 380us/step - accuracy: 0.8816 - loss: 0.8967
- val_accuracy: 0.9038 - val_loss: 0.5545
Epoch 163/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 404us/step - accuracy: 0.8833 - loss: 0.9847
- val_accuracy: 0.9038 - val_loss: 0.5515
Epoch 164/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 634us/step - accuracy: 0.8979 - loss: 1.0577
- val_accuracy: 0.9038 - val_loss: 0.5547
Epoch 165/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 384us/step - accuracy: 0.8696 - loss: 1.0818
- val_accuracy: 0.9066 - val_loss: 0.5753
Epoch 166/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 393us/step - accuracy: 0.8733 - loss: 1.0861
- val_accuracy: 0.9093 - val_loss: 0.5310
Epoch 167/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 383us/step - accuracy: 0.8885 - loss: 1.0841
- val_accuracy: 0.9176 - val_loss: 0.5347
Epoch 168/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 393us/step - accuracy: 0.8979 - loss: 1.0800
- val_accuracy: 0.9121 - val_loss: 0.6348
Epoch 169/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 635us/step - accuracy: 0.8968 - loss: 0.8657
- val_accuracy: 0.9011 - val_loss: 0.5416
Epoch 170/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 388us/step - accuracy: 0.8930 - loss: 1.0192
- val_accuracy: 0.9148 - val_loss: 0.6570
Epoch 171/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 380us/step - accuracy: 0.8877 - loss: 1.0369
- val_accuracy: 0.9148 - val_loss: 0.5224
Epoch 172/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 385us/step - accuracy: 0.8746 - loss: 1.0402
- val_accuracy: 0.9121 - val_loss: 0.5697
Epoch 173/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 379us/step - accuracy: 0.8965 - loss: 0.9375
- val_accuracy: 0.9093 - val_loss: 0.6376
Epoch 174/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 612us/step - accuracy: 0.8835 - loss: 1.2089
- val_accuracy: 0.9121 - val_loss: 0.5423
Epoch 175/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 392us/step - accuracy: 0.8908 - loss: 0.9931
- val_accuracy: 0.9176 - val_loss: 0.5678
Epoch 176/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 382us/step - accuracy: 0.8665 - loss: 0.9681
- val_accuracy: 0.9148 - val_loss: 0.6017
Epoch 177/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 383us/step - accuracy: 0.8706 - loss: 1.0078
- val_accuracy: 0.9066 - val_loss: 0.6391
Epoch 178/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 385us/step - accuracy: 0.8864 - loss: 0.9446
- val_accuracy: 0.9148 - val_loss: 0.5358
Epoch 179/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 635us/step - accuracy: 0.8711 - loss: 1.0226
- val_accuracy: 0.8929 - val_loss: 0.5665
Epoch 180/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 389us/step - accuracy: 0.8878 - loss: 0.9905
- val_accuracy: 0.9176 - val_loss: 0.4669
Epoch 181/200
```

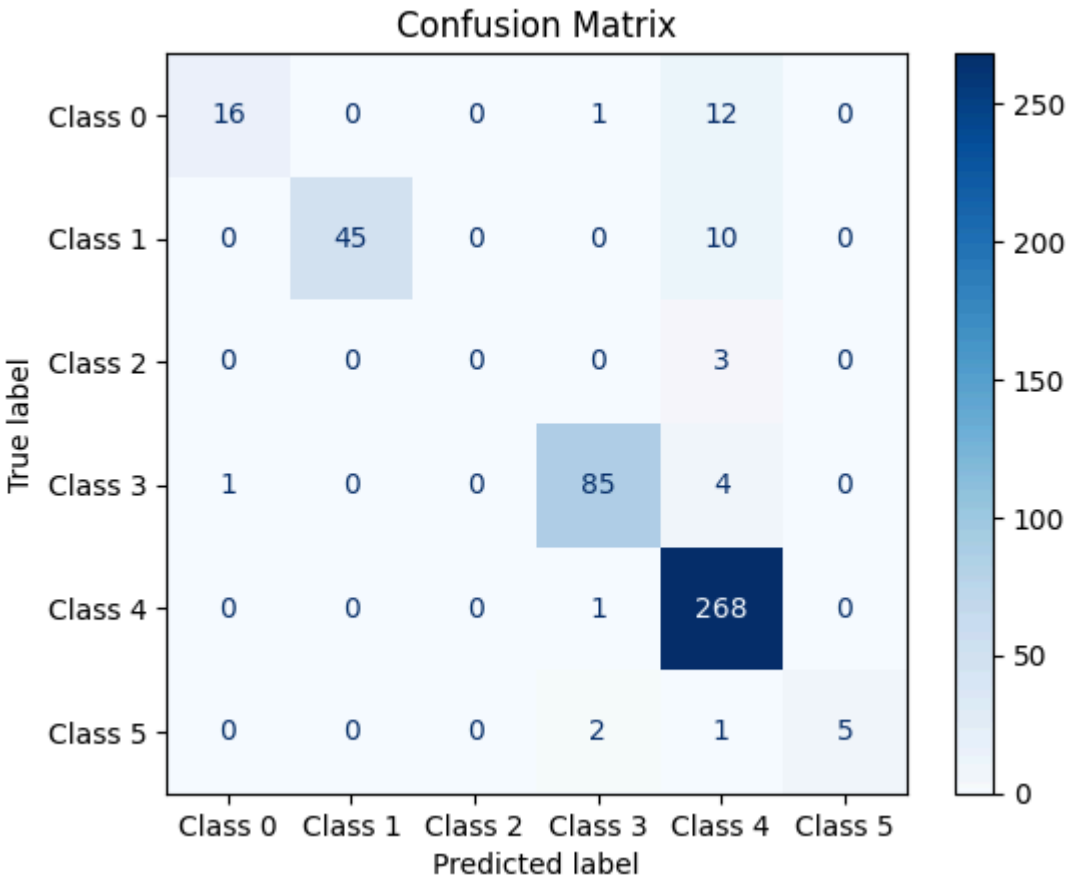
```
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 386us/step - accuracy: 0.8883 - loss: 1.1971
- val_accuracy: 0.9066 - val_loss: 0.5448
Epoch 182/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 387us/step - accuracy: 0.8939 - loss: 1.0064
- val_accuracy: 0.9231 - val_loss: 0.5385
Epoch 183/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 387us/step - accuracy: 0.8914 - loss: 1.0490
- val_accuracy: 0.9066 - val_loss: 0.5470
Epoch 184/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 620us/step - accuracy: 0.8921 - loss: 0.9832
- val_accuracy: 0.9121 - val_loss: 0.5153
Epoch 185/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 382us/step - accuracy: 0.8984 - loss: 1.1275
- val_accuracy: 0.9148 - val_loss: 0.5107
Epoch 186/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 381us/step - accuracy: 0.8884 - loss: 0.9697
- val_accuracy: 0.9176 - val_loss: 0.5339
Epoch 187/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 382us/step - accuracy: 0.8960 - loss: 0.9498
- val_accuracy: 0.9066 - val_loss: 0.5949
Epoch 188/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 495us/step - accuracy: 0.9022 - loss: 0.9861
- val_accuracy: 0.9231 - val_loss: 0.4912
Epoch 189/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 419us/step - accuracy: 0.8896 - loss: 0.8939
- val_accuracy: 0.9203 - val_loss: 0.7154
Epoch 190/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 391us/step - accuracy: 0.8863 - loss: 1.0215
- val_accuracy: 0.9231 - val_loss: 0.6225
Epoch 191/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 385us/step - accuracy: 0.9006 - loss: 0.9777
- val_accuracy: 0.9038 - val_loss: 0.5503
Epoch 192/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 393us/step - accuracy: 0.8886 - loss: 1.2621
- val_accuracy: 0.9038 - val_loss: 0.5644
Epoch 193/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 579us/step - accuracy: 0.8672 - loss: 1.0491
- val_accuracy: 0.9176 - val_loss: 0.6345
Epoch 194/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 387us/step - accuracy: 0.8965 - loss: 0.9275
- val_accuracy: 0.9231 - val_loss: 0.5923
Epoch 195/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 395us/step - accuracy: 0.9074 - loss: 0.9111
- val_accuracy: 0.9231 - val_loss: 0.5367
Epoch 196/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 384us/step - accuracy: 0.8999 - loss: 1.0307
- val_accuracy: 0.9176 - val_loss: 0.4852
Epoch 197/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 586us/step - accuracy: 0.8728 - loss: 0.9484
- val_accuracy: 0.9231 - val_loss: 0.5100
Epoch 198/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 393us/step - accuracy: 0.8944 - loss: 0.9314
- val_accuracy: 0.9231 - val_loss: 0.6093
Epoch 199/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 376us/step - accuracy: 0.8734 - loss: 1.0365
- val_accuracy: 0.9148 - val_loss: 0.6033
Epoch 200/200
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 377us/step - accuracy: 0.8750 - loss: 0.9717
- val_accuracy: 0.9231 - val_loss: 0.5054
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 309us/step - accuracy: 0.9303 - loss: 0.4541
```

```
Test Loss: 0.46134743094444275
Test Accuracy: 0.9229
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
Accuracy Score: 0.9229
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 283us/step - accuracy: 0.9305 - loss: 0.4545
Test Loss: [0.46134743094444275, 0.9229074716567993]
FA_Poly_(g):
  MAE: 0.9767
  R2 Score: 0.6875

Vit_C_(mg):
  MAE: 0.6542
  R2 Score: 0.9917

Thiamin_(mg):
  MAE: 0.0570
  R2 Score: 0.6880

Riboflavin_(mg):
  MAE: 0.0917
  R2 Score: 0.4988

Niacin_(mg):
  MAE: 0.3638
  R2 Score: 0.9587

Vit_B6_(mg):
  MAE: 0.0908
  R2 Score: 0.4421

Folate_Tot_(µg):
  MAE: 1.3087
  R2 Score: 0.9985

Vit_B12_(µg):
  MAE: 0.1479
  R2 Score: 0.9780

Average MAE: 0.4613
Average R2 Score: 0.7804
Accuracy Score: 0.9229
```






```
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 255us/step
<Figure size 800x600 with 0 Axes>
```



chosen CNN since we are dealing with multi dimentional data

we can see the loss is generally decreasing indicating model ois learning

i have used 200 epochs to let the model learn the data and predict properly preventing model to underfit.

from the model output we have avg mae value, avg r2 score =0.7 which is high indicating and the accuracy score of 0.9229 indicating the model performed well

the Avg MAE value 0.6254 indicating the lower mae value indicating better performance in predicting specific nutrients

the Avg R2 value - 0.7804 indicates the the predictions closely match with the actual values

High-performing Features - Vit_C_(mg), Niacin_(mg), and Folate_Tot_(µg) have excellent R2 scores (above 0.9), indicating that the model is able to predict these features very accurately.

from the confusion matrix, we can say the class 4 has highest accuracy so model is highly accurate for class 4 and class 1 is well separated and this indicates the mnodel performs well on certain cases like class4 and class 1

In [11]:
```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classificat
from sklearn.impute import SimpleImputer

dataset = pd.read_csv('food-data.csv')

relevant_nutrients = [
    'Protein_(g)', 'Lipid_Tot_(g)', 'Vit_C_(mg)', 'Thiamin_(mg)',
    'Riboflavin_(mg)', 'Niacin_(mg)', 'Vit_B6_(mg)', 'Folate_Tot_(µg)',
    'Choline_Tot_ (mg)', 'Vit_B12_(µg)', 'Vit_E_(mg)', 'FA_Poly_(g)'
]

X=dataset[relevant_nutrients]

dataset['Cognitive_Performance'] = (X.sum(axis=1) > X.sum(axis=1).median(
y=dataset['Cognitive_Performance']

# Impute missing values using the mean
imputer = SimpleImputer(strategy='mean')
X_imputed = imputer.fit_transform(X)

#train andtest
X_train, X_test, y_train, y_test = train_test_split(X_imputed, y, test_si

# random forest
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Predict on the test set
y_pred = rf_model.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)

confusion = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Confusion Matrix:")
print(confusion)
print("Classification Report:")
print(report)
```

```
Accuracy: 0.9747332959011791
Confusion Matrix:
[[909  14]
 [ 31 827]]
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.98      0.98       923
           1       0.98      0.96      0.97       858

    accuracy                           0.97      1781
   macro avg       0.98      0.97      0.97      1781
weighted avg       0.97      0.97      0.97      1781
```

From the above output we can deduce the following

The model gives an accuracy of 97.47%, indicating that it correctly classifies 97.47% of all instances in the dataset.

Class 0 Precision: 0.97 Recall: 0.98 F1-score: 0.98 Class 0 shows strong performance across all metrics. The high recall indicates that the model correctly identifies 98% of all the instances. The precision of 0.97 means that when the model predicts Class 0, it is correct 97% of the time.

Class 1 Precision: 0.98 Recall: 0.96 F1-score: 0.97 Class 1 also demonstrates excellent performance. The slightly higher precision compared to Class 0 suggests that when the model predicts Class 1, it is even more likely to be correct. However, the recall is slightly lower at 0.96, meaning it misses a small percentage of actual Class 1 instances.

from Confusion Matrix we can say True Negatives : 909 False Positives: 14 False Negatives: 31 True Positives : 827

The model shows a low number of misclassifications in both directions. It has slightly more false negatives (31) than false positives (14), which aligns with the slightly lower recall for Class 1.

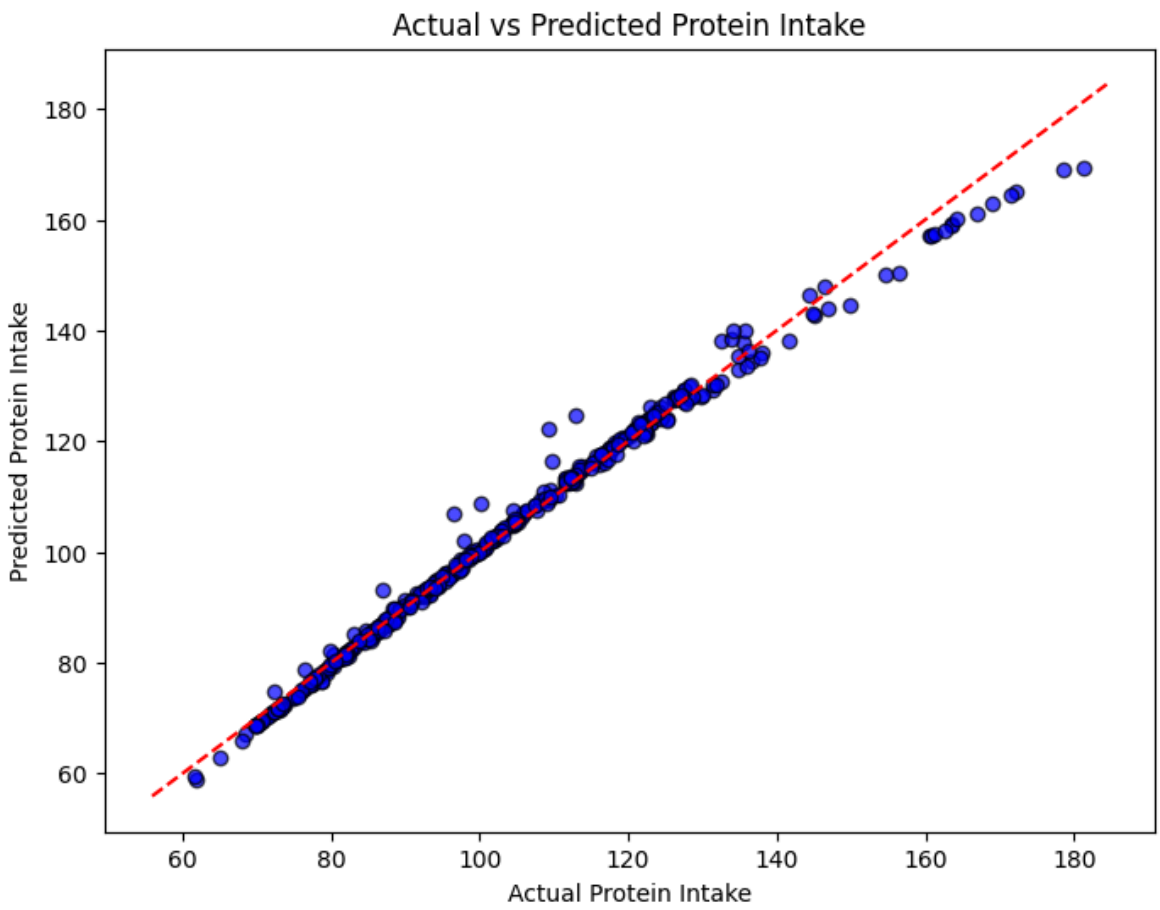
Hypothesis -2

Physical exercise frequency is linked to higher protein intake among age group , with BMR also influencing protein consumption

In [60]:
```
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import Ridge
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, r2_score

data=pd.read_csv('user_nutritional_data.csv')
data=data.dropna()

# filtering data(age groups 15-30)
sub_data=data[(data['Age']>=15)&(data['Age']<=30)]

# choosing relevant features for regression model
features= ['Physical exercise','Age','Daily meals frequency','BMR','Heigh
target_var = 'Proteins'

# feature set and target variable
X=sub_data[features]
y=sub_data[target_var]
X_train, X_test, y_train, y_test= train_test_split(X, y, test_size=0.2, r

pipeline = Pipeline([('scaler', StandardScaler()),('regressor', Ridge(alp

#cross-validation
```

```
crossv_scores = cross_val_score(pipeline, X_train, y_train, cv=5, scoring
mean_msqe = np.mean(-crossv_scores)
mean_rmsqe = np.sqrt(mean_msqe)

#fitting the model on training data
pipeline.fit(X_train, y_train)

# predicting on test data
y_pred = pipeline.predict(X_test)

msqe=mean_squared_error(y_test, y_pred)
rmsqe=np.sqrt(msqe)
r2=r2_score(y_test, y_pred)

print(f"Cross Validated Root Mean Squared Error: {mean_rmse:.2f}")
print(f"Test Root Mean Squared Error: {rmsqe:.2f}")
print(f"R-squared: {r2:.4f}")

# displays respective feature importance
coeff=pd.DataFrame({'Feature':features,'Coefficient':pipeline.named_steps
coeff['Absolute_Importance']=abs(coeff['Coefficient'])
coeff=coeff.sort_values('Absolute_Importance', ascending=False)
print("Feature Importance:")
print(coeff)

plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, color='blue', edgecolor='k', alpha=0.7)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.title('Actual vs Predicted Protein Intake')
plt.ylabel('Predicted Protein Intake')
plt.xlabel('Actual Protein Intake')
plt.title('Actual vs Predicted Protein Intake')
plt.show()
```

```
Cross Validated Root Mean Squared Error: 2.05
Test Root Mean Squared Error: 2.20
R-squared: 0.9909
Feature Importance:
                 Feature  Coefficient  Absolute_Importance
3                    BMR    17.048999            17.048999
0      Physical exercise    14.338120            14.338120
5                 Weight    -0.472504             0.472504
4                 Height     0.304294             0.304294
1                    Age     0.069588             0.069588
2  Daily meals frequency     0.041041             0.041041
```



Using Ridge regression gave us a positive corelation, the reason for choosing ridge is to predict the protien intake for different and since as the target var is continuous, it helps prevent overfitting by reducing the effect of less important parameters.

we got a R-square value of 0.9909 indictaing a strong fit and also

cross-validated rmse gave - 2.05 and test rmse being very close to cross-validted rmse inditcaing a better predicting accuracy to the data and is not overfitting

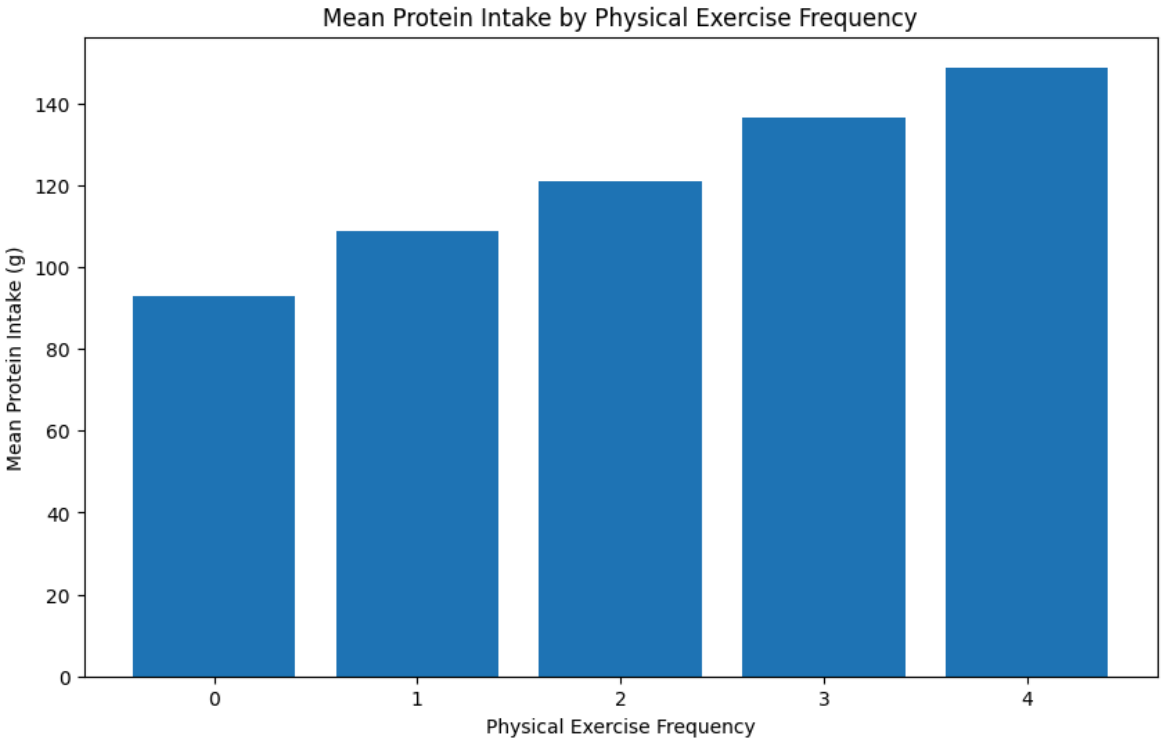
we can also see BMR influencing the protein intake among the age group 15 - 30

with this prediction we can suggest diets with respective amounts of protien depending upon the age group and physical excercise level

In [57]:
```
from scipy import stats
import matplotlib.pyplot as plt
from statsmodels.stats.multicomp import pairwise_tukeyhsd

data = pd.read_csv('user_nutritional_data.csv')
data_groups = data.groupby('Physical exercise')['Proteins'].apply(list)

# performing Anova
f_statistic, p_value = stats.f_oneway(*data_groups)

print(f"One-way ANOVA results:")
print(f"F-statistic: {f_statistic:.4f}")
print(f"p-value: {p_value:.4e}")

# mean/avg protein intake for each exercise group
```

```
proteins_mean = data.groupby('Physical exercise')['Proteins'].mean()
plt.figure(figsize=(10, 6))
plt.bar(proteins_mean.index, proteins_mean.values)
plt.xlabel('Physical Exercise Frequency')
plt.ylabel('Mean Protein Intake (g)')
plt.title('Mean Protein Intake by Physical Exercise Frequency')
plt.show()

# Post-hoc analysis: Tukey's HSD test
from statsmodels.stats.multicomp import pairwise_tukeyhsd

# Prepare data for Tukey's test
exercise_freq = data['Physical exercise']
proteins = data['Proteins']

# Perform Tukey's test
tukey_results = pairwise_tukeyhsd(proteins, exercise_freq)
print("\nTukey's HSD test results:")
print(tukey_results)
```

```
One-way ANOVA results:
F-statistic: 377.1631
p-value: 5.7261e-247
```



```
Tukey's HSD test results:
Multiple Comparison of Means - Tukey HSD, FWER=0.05
=====================================================
group1 group2 meandiff p-adj   lower   upper  reject
-----------------------------------------------------
     0      1  15.6325    0.0 12.6404 18.6245   True
     0      2  27.8783    0.0 23.4254 32.3311   True
     0      3   43.418    0.0 36.7869 50.0491   True
     0      4  55.7651    0.0 50.7966 60.7337   True
     1      2  12.2458    0.0  7.1419 17.3498   True
     1      3  27.7855    0.0 20.7008 34.8703   True
     1      4  40.1327    0.0 34.5731 45.6923   True
     2      3  15.5397    0.0   7.725 23.3544   True
     2      4  27.8869    0.0 21.4228 34.3509   True
     3      4  12.3472 0.0003  4.2276 20.4668   True
-----------------------------------------------------
```

The hypothesis - physical exercise frequency is positively correlated with daily protein intake among age group 15-30 is positive for this analysis.

I have chosen ANOVA as we have to compare multiple groups (i.e excerise frequencies) for statistical difference and get to know which particular groups have difference in protien intake(for example - group 0 and rest of the groups can be considered)

From the ANOVA result, the resultant p-value came less than 0.05 (p-value: 5.7261e-247, p<0.05) and F-statistic value being high enough along with low p value indicating there is a statistical difference in protein intake among exercise frequency groups.

As we can see from Tukey's HSD test result, there is statistically significant differences in protein intake between the groups, showing exercise frequency has a significant effect on protein intake. As exercise frequency increases -> "from Group 0 to Group 4", protein intake also increases, with little differences between Groups 3 and 4.

In [ ]:

In [ ]:

In [ ]: