**Import Section:**

In this section, the necessary libraries and modules are imported to facilitate the implementation of the machine learning project.

|  |
| --- |
| import pandas as pd  import numpy as np  from sklearn.ensemble import HistGradientBoostingRegressor  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import LabelEncoder  from sklearn.metrics import mean\_squared\_error  import matplotlib.pyplot as plt  import seaborn as sns  from sklearn.impute import SimpleImputer |

**Data Analysis Section:**

In this section, the provided code performs the initial steps of data analysis, including loading the dataset, selecting relevant columns, and filtering rows based on specific criteria.

|  |
| --- |
| # Read the Excel file into a DataFrame  file\_path = '/Users/mdabdurrazzak/Downloads/Final\_UK\_Food\_Product.xlsx'  data = pd.read\_excel(file\_path, engine='openpyxl')  # Select specific columns from the DataFrame  selected\_columns = [  'product\_name', 'energy\_100g', 'fat\_100g', 'saturated-fat\_100g', 'carbohydrates\_100g',  'sugars\_100g', 'fiber\_100g', 'proteins\_100g', 'salt\_100g', 'sodium\_100g', 'main\_category', 'nutrition-score-uk\_100g', 'brands'  ]  # Filter rows where 'countries\_en' contains 'United Kingdom' and selected columns are not null  selected\_data = data[data['countries\_en'].str.contains('United Kingdom', na=False)][selected\_columns] |

The provided code snippet calculates and displays basic statistics for the selected dataset. This is an essential step in understanding the distribution and characteristics of the chosen nutritional and product-related features.

|  |
| --- |
| # Display basic statistics for the selected data  print(selected\_data.describe()) |

A number of food items

Description automatically generated with medium confidence

Pairwise Scatter Plot of Nutritional Features:

The provided code generates a pairwise scatter plot illustrating the relationships between selected nutritional features. Each point on the scatter plot represents a data point, and the position of the points in the plots provides insights into the correlations or patterns between pairs of features. The size of the figure has been adjusted to be more compact, with a width of 8 inches and a height of 6 inches.

|  |
| --- |
| # Pairwise scatter plot for selected nutritional features with a smaller figure size  plt.figure(figsize=(8, 6))  sns.pairplot(selected\_data[['energy\_100g', 'fat\_100g', 'carbohydrates\_100g', 'proteins\_100g']])  plt.suptitle('Pairwise Scatter Plot of Nutritional Features', y=1.02)  plt.show() |

A group of blue dots

Description automatically generated

**Label Encoding for Categorical Columns:**

The provided code snippet applies label encoding to non-numeric columns in the selected\_data DataFrame. Label encoding is a preprocessing technique that assigns a unique numerical label to each unique category in a categorical column.

|  |
| --- |
| # Apply label encoding to non-numeric columns  label\_encoders = {}  for column in selected\_data.columns:  if selected\_data[column].dtype == 'object':  label\_encoders[column] = LabelEncoder()  selected\_data[column] = label\_encoders[column].fit\_transform(selected\_data[column]) |

**Generating Synthetic Prices Function:**

The provided code defines a function, generate\_synthetic\_prices, that creates synthetic prices for products based on a normal distribution with a specified mean and standard deviation. It then adds a new 'price' column to the selected\_data DataFrame.

|  |
| --- |
| # Define a function to generate synthetic prices  def generate\_synthetic\_prices(num\_products):  # Assuming a normal distribution of prices with a mean and standard deviation  mean\_price = 3.0 # Adjust based on your assumptions  std\_dev\_price = 1.0 # Adjust based on your assumptions  # Generate synthetic prices  synthetic\_prices = np.random.normal(mean\_price, std\_dev\_price, num\_products)    # Ensure prices are non-negative  synthetic\_prices = np.maximum(synthetic\_prices, 0)    return synthetic\_prices  # Add a new column 'price' to the DataFrame  selected\_data['price'] = generate\_synthetic\_prices(len(selected\_data))  # Display the DataFrame with the new 'price' column  print(selected\_data[['product\_name', 'price']]) |

A screenshot of a computer

Description automatically generated

**Generating Realistic Carbon Footprint Values Function:**

The provided code defines a function, generate\_realistic\_carbon\_footprint, which generates synthetic carbon footprint values for products based on a normal distribution around a specified average. It then adds a new 'carbon\_footprint' column to the selected\_data DataFrame.

|  |
| --- |
| # Function to generate realistic carbon footprint values  def generate\_realistic\_carbon\_footprint():  # Define an average carbon footprint (in kgCO2e per kg)  average\_carbon\_footprint = 2.0  # Generate carbon footprint values with some variation  return np.random.normal(average\_carbon\_footprint, 0.5)  # Create a new column 'carbon\_footprint' in the DataFrame  selected\_data['carbon\_footprint'] = selected\_data.apply(lambda row: generate\_realistic\_carbon\_footprint(), axis=1)  # Display the DataFrame with the new column  print(selected\_data[['product\_name', 'carbon\_footprint']]) |

A screenshot of a computer

Description automatically generated

**Inverse Transforming and Categorizing Products:**

The code snippet begins by undoing label encoding for categorical columns in the selected\_data DataFrame. This ensures the restoration of original categorical values for interpretability. Subsequently, a product categorization function assigns products to predefined categories based on keywords in the 'main\_category' attribute, creating a new 'product\_category' column. Finally, one-hot encoding is applied to convert categorical attributes into binary columns. This concise process enhances the dataset's structure, providing both original categorical values and a structured representation suitable for machine learning tasks.

|  |
| --- |
| # Inverse transform the label encoded columns after imputation  for column in selected\_data.columns:  if column in label\_encoders:  selected\_data[column] = label\_encoders[column].inverse\_transform(selected\_data[column])  # Define categories for product categorization  categories = {  'Category A': ['organic', 'healthy', 'natural'],  'Category B': ['snacks', 'chips', 'sweets'],  'Category C': ['beverages', 'drinks', 'juices'],  'Category D': ['dairy', 'cheese', 'milk'],  'Category E': ['meat', 'sausage', 'chicken']  }  # Function to categorize products based on 'main\_category' attribute  def categorize\_product(row, categories):  for category, keywords in categories.items():  for keyword in keywords:  if keyword.lower() in str(row['main\_category']).lower():  return category  return 'Other'  # Apply categorization to the DataFrame and create a new column 'product\_category'  selected\_data['product\_category'] = selected\_data.apply(lambda row: categorize\_product(row, categories), axis=1)  # One-hot encode categorical columns  selected\_data = pd.get\_dummies(selected\_data, columns=['main\_category']) |

A graph showing a number of different colored squares

Description automatically generated

**User Input:**

This code segment allows users to input a product name or category, filters products based on the input, and then calculates and assigns a combined score to the filtered products. The combined score is derived from the user-defined weights and the predictions made by the specified machine learning model. This functionality assists users in exploring and selecting products based on their personalized preferences, considering factors such as nutrition, cost, and carbon footprint.

|  |
| --- |
| # Define a generic function to calculate the combined score for all products  def calculate\_combined\_score(selected\_data, nutrition\_weight, cost\_weight, carbon\_footprint\_weight, model):  # Features for the model  features = selected\_data[['nutrition-score-uk\_100g', 'price', 'carbon\_footprint']]  # Target variable  target = selected\_data['combined\_score']  # Handle missing values in the target variable  imputer = SimpleImputer(strategy='mean')  target = imputer.fit\_transform(target.values.reshape(-1, 1)).flatten()  # Train the specified model to predict user preferences  model.fit(features, target)  # Predict user preferences for all products  predicted\_nutrition\_scores = model.predict(features)  # Combine the predicted scores with price and carbon footprint  combined\_scores = (  nutrition\_weight \* predicted\_nutrition\_scores +  cost\_weight \* selected\_data['price'] +  carbon\_footprint\_weight \* selected\_data['carbon\_footprint']  )  # Add the combined scores to the DataFrame  selected\_data.loc[:, 'combined\_score'] = combined\_scores  # User-defined weights for nutrition, cost, and carbon footprint  nutrition\_weight = 1.0  cost\_weight = -0.5  carbon\_footprint\_weight = -0.2  # User input (product name or part of the name)  user\_input = input("Enter a product name or category: ")  # Filter products based on user input  filtered\_products = selected\_data[  selected\_data['product\_name'].str.contains(user\_input, case=False) |  selected\_data['product\_category'].str.contains(user\_input, case=False)  ] |

The histogram showcases the distribution of combined scores for user-filtered products, emphasizing the aggregation of predicted nutrition scores, product prices, and carbon footprints based on user-defined weights. The x-axis represents the combined score, offering insight into the distribution's central tendency and spread. The sky-blue color denotes the frequency of products within each combined score range. This visualization serves as a concise tool for users to discern the overall pattern of product evaluations, aiding in the identification of prevalent score ranges and the variability in user preferences across the selected products.

|  |
| --- |
| plt.figure(figsize=(12, 6))  sns.histplot(filtered\_products['combined\_score'], bins=20, kde=True, color='skyblue')  plt.title('Distribution of Combined Scores')  plt.xlabel('Combined Score')  plt.ylabel('Frequency')  plt.show() |
|  |

A graph of a bar graph

Description automatically generated with medium confidence

The scatter plot above juxtaposes product prices against their respective combined scores, allowing users to discern relationships between cost and overall product evaluation. Each data point represents a product, with the x-axis denoting the product price and the y-axis indicating the combined score. The green color accentuates the data points, aiding in the visual distinction of price-combined score associations. This visualization is instrumental in helping users identify potential patterns or trends, providing valuable insights into how product pricing influences their personalized combined scores. Users can efficiently navigate and make informed decisions regarding products based on their preferences, taking both nutritional aspects and cost considerations into account.

|  |
| --- |
| plt.figure(figsize=(10, 6))  sns.scatterplot(x='price', y='combined\_score', data=filtered\_products, color='green')  plt.title('Price vs. Combined Score')  plt.xlabel('Product Price')  plt.ylabel('Combined Score')  plt.show() |

A graph showing a number of green dots

Description automatically generated

**Machine Learning Models:**

**HistGradientBoostingRegressor:**

The algorithm, HistGradientBoostingRegressor, minimizes a designated loss function, often Mean Squared Error (MSE) for regression, measuring the average squared difference between predicted (ŷ) and actual values (y). Utilizing gradient descent, it adjusts tree weights iteratively to optimize predictions. Decision trees, the base learners, are constructed to reduce residual errors by recursively splitting data based on features. The distinctive Histogram-Based Approach discretizes continuous features into bins, reducing computational complexity. In summary, the algorithm combines gradient boosting with efficient histogram-based techniques, making it apt for high-dimensional datasets. Its strength lies in creating a robust predictive model through multiple decision trees' aggregation.

MSE=*n*1​∑*i*=1*n*​(*yi*​−*y*^​*i*​)2

Gradient Descent:

Adjust parameters iteratively using computed gradients to minimize the loss function.

Gradient Descent:Adjust parameters iteratively using computed gradients to minimize the loss function.

Decision Trees:

Construct trees to recursively split data, creating leaf nodes for predicted values.

Histogram-Based Approach: Discretize continuous features into bins, enhancing computational efficiency.

The code segment employs the HistGradientBoostingRegressor algorithm, a machine learning model, to generate personalized product recommendations. It initiates by preparing features and target variables for filtered products and addressing missing values through mean imputation. Subsequently, the dataset undergoes a split into training and testing sets, facilitating model training and evaluation. The HistGradientBoostingRegressor model is instantiated and trained on the training set. The mean squared error is then computed to assess the model's accuracy on the test set. Beyond evaluation, the code leverages the trained model to predict scores for all filtered products, identifying and presenting the top 5 recommended products based on their combined scores. This comprehensive functionality empowers users to gauge model performance, understand its predictive accuracy through the mean squared error metric, and make informed product choices aligned with their preferences.

|  |
| --- |
| # Separate features and target  X = filtered\_products[['nutrition-score-uk\_100g', 'price', 'carbon\_footprint']]  y = filtered\_products['combined\_score']  # Handle missing values in features  imputer\_features = SimpleImputer(strategy='mean')  X = imputer\_features.fit\_transform(X)  # Split data into training and testing sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  # Create an instance of HistGradientBoostingRegressor  hist\_model = HistGradientBoostingRegressor(random\_state=42)  # Train the model  hist\_model.fit(X\_train, y\_train)  # Predict on the test set  y\_pred = hist\_model.predict(X\_test)  # Evaluate the model  mse = mean\_squared\_error(y\_test, y\_pred)  print(f'Mean Squared Error: {mse}')  # Display the top 5 recommended products  filtered\_products\_copy = filtered\_products.copy() # Create a copy to avoid SettingWithCopyWarning  filtered\_products\_copy['predicted\_score'] = hist\_model.predict(X)  top\_products\_hist = filtered\_products\_copy.nlargest(5, 'predicted\_score')[['product\_name', 'product\_category', 'predicted\_score']]  print("Filtered and Top 5 Recommended Products using HistGradientBoostingRegressor:")  print(top\_products\_hist[['product\_name', 'product\_category', 'predicted\_score']]) |

A screen shot of a computer

Description automatically generated

**Learning Curve:**

The learning curve depicts the model's performance evolution with an increasing number of training examples. In the plot, the x-axis represents the number of training examples, while the y-axis denotes the scores. The blue line illustrates the average training score, showcasing how well the model fits the training data. Simultaneously, the orange line represents the average validation score, reflecting the model's generalization to unseen data. As training examples increase, the learning curve provides insights into potential overfitting or underfitting. A convergence of training and validation scores indicates an optimal model. In this specific case, the learning curve indicates how the HistGradientBoostingRegressor's performance evolves concerning the amount of available training data, assisting users in assessing its learning behavior and generalization capabilities.

|  |
| --- |
| train\_sizes, train\_scores, test\_scores = learning\_curve(hist\_model, X, y, cv=5)  plt.figure(figsize=(10, 6))  plt.plot(train\_sizes, np.mean(train\_scores, axis=1), label='Training Score')  plt.plot(train\_sizes, np.mean(test\_scores, axis=1), label='Validation Score')  plt.title('Learning Curve')  plt.xlabel('Training Examples')  plt.ylabel('Score')  plt.legend()  plt.show() |

A graph of a line

Description automatically generated with medium confidence

**RandomForestRegressor :**

RandomForestRegressor, a versatile ensemble learning algorithm, builds a forest of decision trees for regression tasks. The ensemble construction involves two key components: bootstrap sampling and random feature selection. Bootstrap sampling ensures diversity by randomly selecting subsets of the training data with replacement for each tree, while random feature selection introduces variability at each node split. Decision trees within the forest are constructed independently, aiming to minimize Mean Squared Error (MSE) by recursively splitting nodes. The split criteria identify optimal points to reduce impurity within nodes. Ensemble aggregation entails averaging predictions from all trees, a process that smoothens predictions and mitigates overfitting. The algorithm is initialized by sampling data and selecting features randomly for each tree. Tree growth follows predefined stopping criteria, such as maximum depth. In predicting new inputs, the algorithm aggregates predictions from all trees. The Mean Squared Error serves as a common evaluation metric, quantifying the average squared difference between actual and predicted values. RandomForestRegressor excels in providing a robust and adaptable ensemble, making it well-suited for regression tasks across diverse datasets.

The code first separates features (nutritional scores, prices, and carbon footprints) and the target variable (combined scores) for filtered products. Missing values in features are handled using mean imputation. The data is then split into training and testing sets. An instance of the RandomForestRegressor is created, trained on the training set, and evaluated using MSE on the test set. Finally, the code identifies and displays the top 5 recommended products based on predicted combined scores, enabling users to make informed choices aligned with their preferences.

|  |
| --- |
| # Separate features and target  X = filtered\_products[['nutrition-score-uk\_100g', 'price', 'carbon\_footprint']]  y = filtered\_products['combined\_score']  # Handle missing values in features  imputer\_features = SimpleImputer(strategy='mean')  X = imputer\_features.fit\_transform(X)  # Split data into training and testing sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  # Create an instance of RandomForestRegressor  rf\_model = RandomForestRegressor(random\_state=42)  # Train the model  rf\_model.fit(X\_train, y\_train)  # Predict on the test set  y\_pred = rf\_model.predict(X\_test)  # Evaluate the model  mse = mean\_squared\_error(y\_test, y\_pred)  print(f'Mean Squared Error: {mse}')  # Display the top 5 recommended products  filtered\_products\_copy = filtered\_products.copy() # Create a copy to avoid SettingWithCopyWarning  filtered\_products\_copy['predicted\_score'] = rf\_model.predict(X)  top\_products\_rf = filtered\_products\_copy.nlargest(5, 'predicted\_score')[['product\_name', 'product\_category', 'predicted\_score']]  print("Filtered and Top 5 Recommended Products using RandomForestRegressor:")  print(top\_products\_rf[['product\_name', 'product\_category', 'predicted\_score']]) |

A screenshot of a computer

Description automatically generated

**Learning Plot:**

|  |
| --- |
| train\_sizes, train\_scores, test\_scores = learning\_curve(rf\_model, X, y, cv=5)  plt.figure(figsize=(10, 6))  plt.plot(train\_sizes, np.mean(train\_scores, axis=1), label='Training Score')  plt.plot(train\_sizes, np.mean(test\_scores, axis=1), label='Validation Score')  plt.title('Learning Curve')  plt.xlabel('Training Examples')  plt.ylabel('Score')  plt.legend()  plt.show() |

A graph with a line

Description automatically generated

**The Residual Plot**: depicted in a scatterplot with predicted values on the x-axis and residuals on the y-axis, serves as a diagnostic tool for assessing the performance of the RandomForestRegressor. Each point represents an observation, and the vertical distance from the point to the horizontal axis indicates the difference between the actual and predicted values (residual). A well-behaved plot should exhibit a random scatter of points without discernible patterns, indicating a balanced model. Patterns, trends, or uneven spreads may signal issues such as underfitting, overfitting, or nonlinear relationships. The Residual Plot aids in validating model assumptions, identifying regions of strong and weak performance, and ensuring the reliability of predictions.

|  |
| --- |
| plt.figure(figsize=(10, 6))  sns.scatterplot(x=y\_pred, y=y\_test - y\_pred, color='skyblue')  plt.title('Residual Plot')  plt.xlabel('Predicted Values')  plt.ylabel('Residuals')  plt.show() |

A graph with blue dots

Description automatically generated

**validation section :**

In the validation section, missing values in the features are handled using mean imputation with SimpleImputer. The data is then split into training and testing sets using the train\_test\_split function. Two regression models, namely HistGradientBoostingRegressor and RandomForestRegressor, are trained on the training set and evaluated on the testing set. The mean squared error (MSE) is computed for each model to quantify the squared differences between the actual and predicted target values. In the current validation, the HistGradientBoostingRegressor yields a mean squared error of approximately 10.67, while the RandomForestRegressor achieves a slightly lower mean squared error of approximately 9.58. These metrics provide a quantitative measure of the models' performance on the validation set, with lower MSE values indicating better predictive accuracy.

|  |
| --- |
| # Handle missing values in features  imputer\_features = SimpleImputer(strategy='mean')  features = imputer\_features.fit\_transform(features)  # Split data into training and testing sets  features\_train, features\_test, target\_train, target\_test = train\_test\_split(  features, target, test\_size=0.2, random\_state=42  )  # Train and evaluate HistGradientBoostingRegressor  hist\_model.fit(features\_train, target\_train)  hist\_predictions = hist\_model.predict(features\_test)  hist\_mse = mean\_squared\_error(target\_test, hist\_predictions)  print(f'Mean Squared Error for HistGradientBoostingRegressor: {hist\_mse}')  # Train and evaluate RandomForestRegressor  rf\_model.fit(features\_train, target\_train)  rf\_predictions = rf\_model.predict(features\_test)  rf\_mse = mean\_squared\_error(target\_test, rf\_predictions)  print(f'Mean Squared Error for RandomForestRegressor: {rf\_mse}') |

Root Mean Squared Error (RMSE) is computed for both the HistGradientBoostingRegressor and the RandomForestRegressor models. RMSE is calculated by taking the square root of the mean squared error (MSE), providing a measure of the average magnitude of the prediction errors. For the HistGradientBoostingRegressor, the RMSE is approximately 3.27, while the RandomForestRegressor achieves a slightly lower RMSE of approximately 3.10. These RMSE values offer an interpretable metric for the average size of the prediction errors, allowing for a comparative assessment of the models' performance in the validation phase. Lower RMSE values indicate better predictive accuracy and suggest that the models are providing predictions that are closer to the actual target values.

|  |
| --- |
| hist\_rmse = np.sqrt(hist\_mse)  print(f'Root Mean Squared Error for HistGradientBoostingRegressor: {hist\_rmse}')  # Compute RMSE for RandomForestRegressor  rf\_rmse = np.sqrt(rf\_mse)  print(f'Root Mean Squared Error for RandomForestRegressor: {rf\_rmse}') |