Multi-Class Prediction of Obesity Risk

Adnan Sardi[1], Vu Minh Dung[2], and Vu Tuan Long[3]

[1]Data Analyst, Toronto, Canada
[2]2nd year as university student, Hanoi, Vietnam
[3]Mechanical Engineer, Neuilly-sur-Seine, France

April 2024

Contents

1 Abstract 3

2 Introduction 3

3 Preparation & Checking 5

4 Training dataset preparation 6
- 4.1 Data cleaning & EDA 7
- 4.2 Feature engineering 16
- 4.3 Data transformation 17

5 Predective model for obesity risk dataset 19
- 5.1 Logistic Regression model 21
- 5.2 KNN model 22
- 5.3 SVC model 23
- 5.4 Naive Bayes model 24
- 5.5 Random Forest model 25
- 5.6 XGBoost model 25
  - 5.6.1 Train XGBoost model & Parameter Tunning 27

6 Results 28

7 Conclusion 28

8 Appendix 29

1. Abstract

Obesity is a huge problem in our modern society hitting more than 650 million adults and even 1.9 billion are overweight as written World Health Organization and these numbers may rise in the future. Obesity affects cardiovascular disease which is the first reason for death. The goals of this project are to use various factors to predict obesity risk in individuals, enhance our data analytical skills and develop our data project management and teamwork.

2. Introduction

Ever wondered what factors contribute most to obesity risk? We, a data science team of three, embarked on a collaborative quest to tackle this very question! This project, hosted on the renowned Kaggle[1] platform as a competition, presented a rich dataset brimming with insights waiting to be unearthed.

Having previously teamed up on a successful data project, Vu Minh Dung, Vu Tuan Long and I shared this experience fostered a seamless workflow, allowing us to navigate the intricacies of data analysis with efficiency. Before diving headfirst into the data, we meticulously laid the foundation for our exploration. This initial phase involved in importing essential libraries: Just like a builder gathers the necessary tools, we began by importing the essential Python libraries for data manipulation and analysis. Libraries such as:

- import pandas as pd
- import numpy as np
- import matplotlib.pyplot as plt
- import matplotlib.cm as cm
- import seaborn as sns
- from sklearn.model_selection import train_test_split, GridSearchCV
- from sklearn.ensemble import RandomForestClassifier
- from sklearn.linear_model import LogisticRegression
- from sklearn.naive_bayes import GaussianNB
- from sklearn.neighbors import KNeighborsClassifier
- from sklearn.svm import SVC

[1] https://www.kaggle.com/competitions/playground-series-s4e2/

- from xgboost import XGBClassifier
- from sklearn.metrics import classification_report
- import warnings

Now let's explain each library we have imported and their purpose.

Pandas (pd): A powerful tool for data manipulation and analysis, featuring high-performance DataFrames and Series for tabular data. Ideal for data cleaning, transformation, and numerical computations.

NumPy (np): Essential for numerical computing in Python, providing efficient multi-dimensional arrays and operations, crucial for large dataset analyses in scientific computing and machine learning.

Matplotlib.pyplot (plt): A fundamental library for creating a variety of static visualizations such as line charts, histograms, and scatter plots.

Seaborn (sns): Built atop matplotlib, this high-level library enhances the creation of statistical graphics, focusing on aesthetics and clarity.

Train_test_split: A function from sklearn.model_selection that splits data into training and testing sets to avoid overfitting and enhance model reliability.

GridSearchCV: Also from sklearn.model_selection, this tool facilitates hyperparameter tuning by automating the testing of different parameter combinations and optimizing model settings.

Machine Learning Models: Includes implementations like RandomForestClassifier, LogisticRegression, and XGBClassifier from scikit-learn. These models are used to identify patterns in data and make predictions on new, unseen datasets, chosen based on the problem type and data traits.

Classification_report: A function from sklearn.metrics that produces a report detailing a classification model's performance, including precision, recall, F1-score, and support metrics for each class, helping to highlight model strengths and areas for improvement.

```
train_df_path = '/kaggle/input/playground-series-s4e2/train.csv'
train_df = pd.read_csv(train_df_path)
```

```
test_df_path = '/kaggle/input/playground-series-s4e2/test.csv'
test_df = pd.read_csv(test_df_path)
```

As a final step of our introduction we loaded our data into data_train and data_test. These initial steps were crucial in preparing the data for further analysis.

3. Preparation & Checking

During this phase the goal is to figure out our dataset includes which types of features we have, start to do some hypotheses and check if we need to clean or replace some missing values. Pandas offers us many functions to analyze our dataset such as head(), info(), describe(), isnull() and so on. Below you can visualize which we used.

```
train_df.head()
test_df.head()
```

```
print(f'1. train_df columns: {train_df.shape[1]}')
print(f'   train_df length (no. example): {train_df.shape[0]}')
print('')
print(f'2. test_df columns: {test_df.shape[1]}')
print(f'   test_df length (no. example): {test_df.shape[0]}')
```

```
train_df.info()
test_df.info()
```

The outputs of these scripts are available in the Appendix section. After visualizing our outputs we can make some first considerations. The only difference between the train and test datasets is the number of rows, which train dataset contains more rows than the test dataset (20758 against 13840) and a further difference is the missing feature of "NObeyesdad" which is the target for our prediction. With the function .head() we can watch various different features such as:

- id, an identifier for each individual or record.
- Gender, the gender of the individual (e.g., male or female).
- Age, the age of the individual.
- Height, the height of the individual.
- Weight, the weight of the individual.
- family_history_with_overweight, indicates whether the individual has a family history of overweight or obesity.
- FAVC, whether the individual consumes high-calorie food frequently (FAVC stands for Frequent Consumption of High Caloric Food).

- FCVC, frequency of consumption of vegetables.
- NCP, number of main meals per day.
- CAEC, consumption of food between meals (CAEC may stand for something like Consumption of Additional Energy Content).
- SMOKE, indicates whether the individual smokes or not.
- CH2O, daily water consumption.
- SCC, calories consumption monitoring.
- FAF, physical activity frequency.
- TUE, time spent on entertainment screen time (e.g., TV, computer).
- CALC, caloric intake monitoring.
- MTRANS. mode of transportation used by the individual.
- NObeyesdad, obesity level or class (e.g., underweight, normal weight, overweight, obesity class I, II, III).

All of these features may be divided into 3 data types such as numerical, categorial and identifier, also there are no missing values in any of the columns, which simplifies preprocessing. General considerations for further analysis are encoding categorical data. Both datasets contain several categorical features that will need to be encoded to prepare for any machine learning model and feature engineering. We considered creating new features that could help improve the predictive power of our models, such as BMI (Body Mass Index) derived from Height and Weight.

4. Training dataset preparation

This phase ensures that the dataset is clean, well-understood, and appropriately formatted, which are essential for building robust and accurate predictive models. The preparation of the training dataset involves several key steps:

Data Cleaning & Exploratory Data Analysis (EDA): This initial step focuses on identifying and rectifying any issues in the dataset, such as correcting data entry errors, and removing outliers. EDA is performed to uncover patterns, anomalies, dependencies, and potentially informative relationships between the variables.

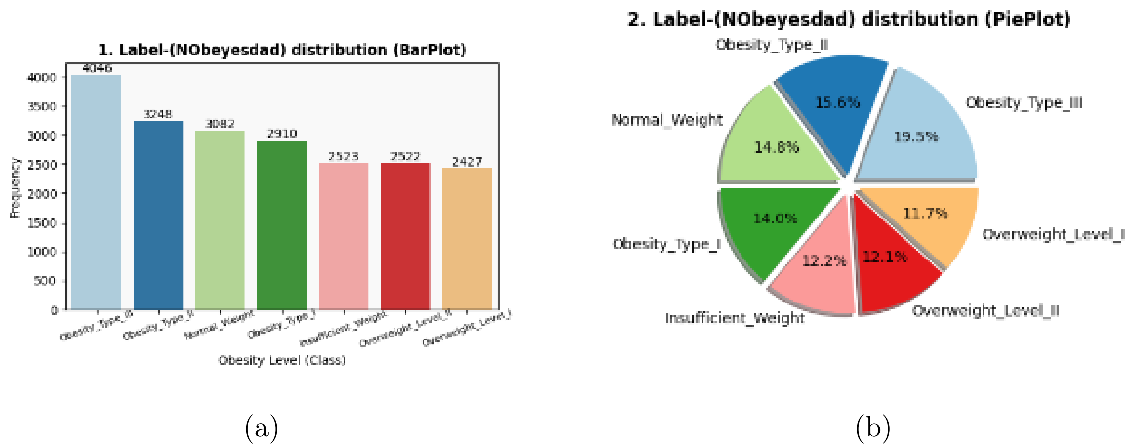
Feature Engineering: This process involves creating new features from existing ones to increase the predictive power of the machine learning models. The goal is to utilize domain knowledge and insights gained during EDA to craft features that help in better representing the problem to predictive models.

Data Transformation: The final step involves transforming the data into a format that can be effectively used by machine learning algorithms. This includes scaling and normalizing numerical data, encoding categorical variables, and potentially applying more complex transformations like PCA (Principal Component Analysis) or bucketing.

4.1 Data cleaning & EDA

```
label_counts = train_df['NObeyesdad'].value_counts()

fig, axs = plt.subplots(1,2,figsize=(12,4))

sns.barplot(x=label_counts.index, y=label_counts.values,
            ax=axs[0], palette='Paired')
axs[0].set_title('1. Label-(NObeyesdad) distribution (BarPlot)',
                 fontweight='bold')
axs[0].set_ylabel('Frequency')
axs[0].set_xlabel('Obesity Level (Class)')
axs[0].tick_params(axis='x', rotation=20, labelsize=8, pad=-5)
for idx, value in enumerate(label_counts):
    axs[0].text(idx, value, str(value), ha='center', va='bottom')
axs[0].set_facecolor('#F9F9F9')

explode = (0.08,0.08,0.08,0.08,0.08,0.08,0.08)
axs[1].pie(labels=label_counts.index, x=label_counts.values,
            autopct='%1.1f%%', explode=explode,
            colors=sns.color_palette('Paired', len(label_counts)),
            shadow=True)
axs[1].set_title('2. Label-(NObeyesdad) distribution (PiePlot)',
                 fontweight='bold')
axs[1].set_aspect('equal')

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

Here below are the visualizations.



(a) (b)

Figure 1: Both charts represent NObeyesdad. In the second chart (b) we have a pie chart with every percentage of NObeyesdad while the other chart (a) is a comparison with all of them.

The bar plot shows that 'Obesity_Type_I' is the most frequent class with a count of 4,364, making it the highest bar in the plot. The second most common class is 'Obesity_Type_III' with 3,248 instances. 'Obesity_Type_II' follows closely with 3,082 occurrences. 'Overweight_Level_I' and 'Overweight_Level_II' have a similar number of instances, 2,573 and 2,522 respectively. 'Normal_Weight' is slightly less common, with 2,470 instances. The least represented class is 'Insufficient_Weight', with the smallest bar showing 2,427 cases. In the pie chart, the largest portion of the dataset is made up of 'Obesity_Type_III', constituting 19.5% of the total instances. 'Obesity_Type_I' represents 15.6% of the data, despite having the highest count in the bar plot, which indicates the dataset has a somewhat even distribution of classes. 'Normal_Weight' makes up 14.8%, while 'Obesity_Type_II' is 14.0% of the dataset. 'Overweight_Level_I' and 'Overweight_Level_II' are quite close in proportion, at 12.2% and 11.7% respectively. The 'Insufficient_Weight' category, while the least frequent, still comprises a significant 12.1% of the data. Another important phase is that we need to label encoding our target, here is the code.

```
target_map = {
    'Obesity_Type_III': 6,
    'Obesity_Type_II': 5,
    'Obesity_Type_I': 4,
    'Overweight_Level_II': 3,
    'Overweight_Level_I': 2,
    'Normal_Weight': 1,
    'Insufficient_Weight': 0
}

train_df['label_encode'] = train_df['NObeyesdad'].map(target_map)
```

It's essential to understand the distribution of our features variable, as this can significantly influence the performance of our predictive models. Here there is a distribution function.

```
def dis_plot(data, col_list, dtype):
    plt.figure(figsize=(18,15))

    for i, col in enumerate(col_list):
        plt.subplot(5,3,i+1)
        if dtype == 'object':
            value_count_col = data[col].value_counts()
            sns.countplot(data=data, x=col, palette='colorblind')
            for i, value in enumerate(value_count_col):
                plt.text(i, value, f'{value/len(data)*100:.2f}%',
                ha='center', va='bottom', fontdict=
                {'color':'black'})
        else:
            sns.histplot(data=data, x=col, kde=True,
            palette='colorblind')

        plt.xticks(rotation=30)
        plt.title(f'{col.title()} distribution',
                        fontweight='bold')

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

```
non_num_train = [col for col in train_df if
                    train_df[col].dtype == 'object' and col !=
                    'NObeyesdad']

dis_plot(data=train_df, col_list=non_num_train, dtype='object')
```

```
non_num_test = [col for col in test_df if
                    test_df[col].dtype == 'object']

dis_plot(data=test_df, col_list=non_num_test, dtype='object')
```

Here is the distribution for all non-numerical features in the train dataset, for the test dataset is available in the Appendix section.

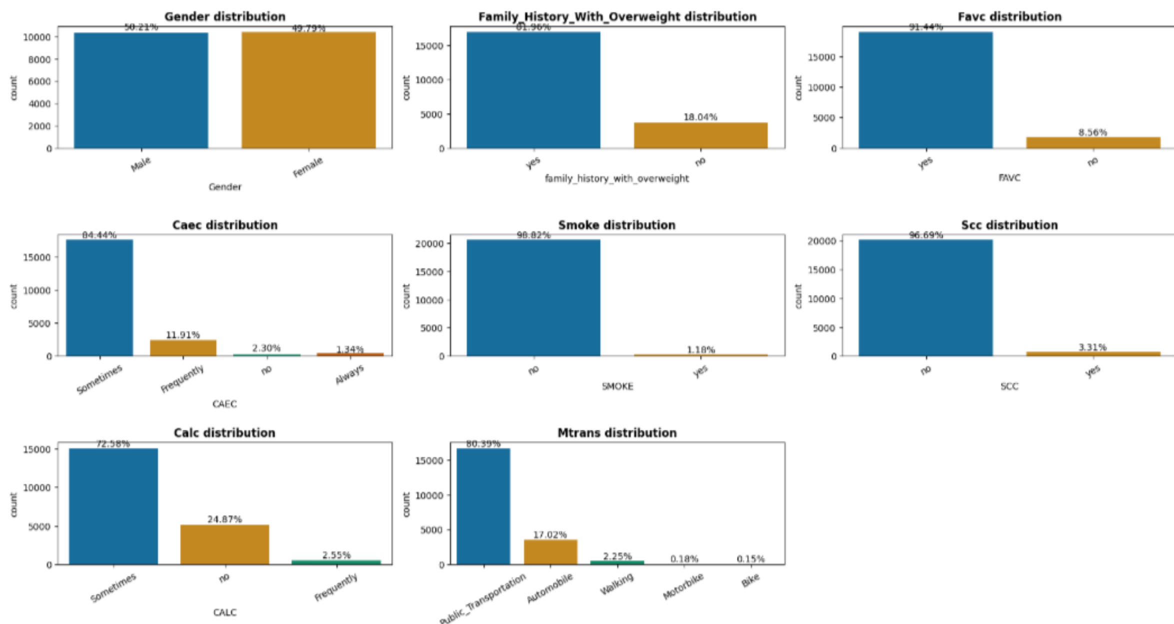


Figure 2: Plotting for non-numeric columns train_df.

The majority are Female (59%), while Male accounts for approximately 41%, showing a slight skew towards female participants. A significant majority (about 82%) have a family history of overweight, which suggests that genetic factors may be significant in this dataset. Most participants (91%) frequently consume high-caloric food, indicating that diet could be an influential factor in obesity levels. The vast majority (84%) consume food between meals sometimes, while only a small fraction do so frequently (around 12%) or always (about 2-3%). The smoking habit is not prevalent, with about 98% of the participants not smoking. Very few participants (3%) monitor their calorie consumption,

suggesting a potential area of intervention for weight management. The consumption of alcohol is mostly 'Sometimes' (72%), with 'no' consumption at nearly 25% and 'Frequently' being quite rare (2.5%). The overwhelming majority use public transportation (80%). Other means of transportation are much less common, with automobiles around 17%, and walking, motorbike, and bike combined are less than 3%.
Now let's analyze the numerical features of the train dataset. Plotting of the test dataset is available in the Appendix section.

```
num_train = [col for col in train_df if train_df[col].dtype
                != object and col != 'id']

dis_plot(data=train_df, col_list=num_train, dtype='numeric')
```

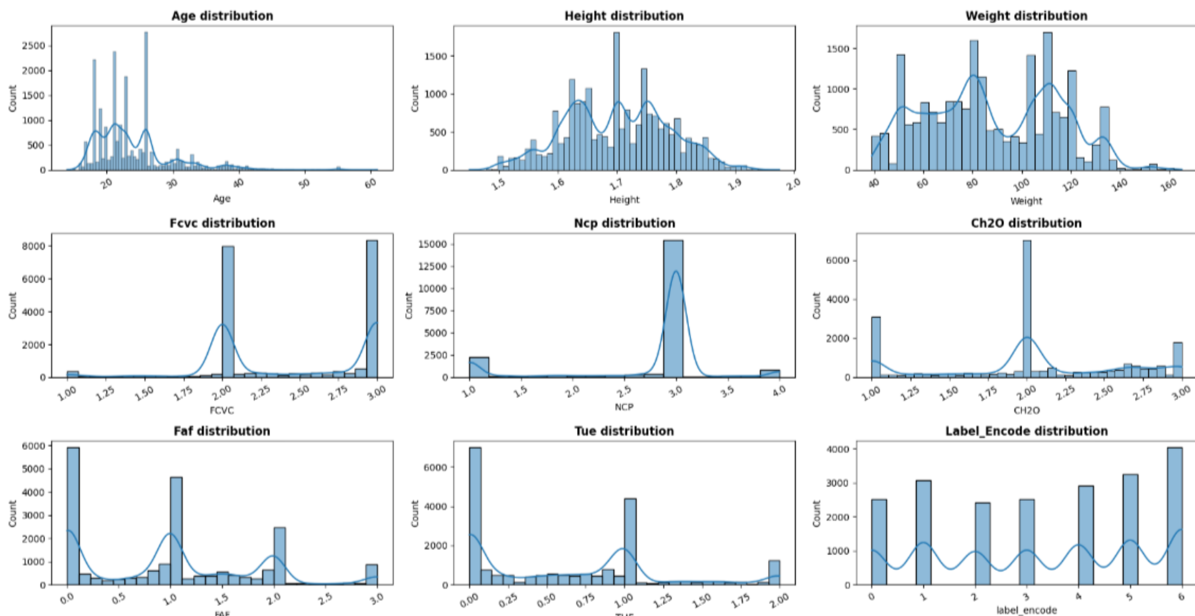


Figure 3: Plotting for numeric columns train_df.

Age Distribution: The distribution of age shows multiple peaks, which could suggest different groups within the dataset, or age ranges where participants are more commonly found. Most participants are concentrated in the 20-40 age range. Height Distribution: The height distribution appears approximately normal but with multiple peaks, suggesting there are distinct groupings of height within the population. Weight Distribution: The weight distribution is more spread out and has multiple peaks, similar to age and height, indicating the presence of various weight groups within the dataset. FCVC Distribution: There is a large peak at the highest rating, indicating a high frequency of vegetable consumption among most participants. There are also smaller peaks around the lower ratings, indicating fewer individuals consuming vegetables less frequently. NCP Distribution: The distribution is sharply peaked around 3 meals a day, which is typical for

most individuals. Other values have a considerably lower frequency, suggesting that most people adhere to a three-meal pattern. CH2O Distribution: Most participants consume around 2 litres of water a day, which corresponds to the sharp peak. Other amounts are much less common. FAF Distribution: There's a significant peak at 0, indicating a large number of individuals with no physical activity. There's another peak between 1 and 2, suggesting another group that engages in physical activity with moderate frequency. TUE Distribution: The distribution shows a large number of participants with very low technology usage times, but also peaks at higher usage times, indicating variability in the population's technology use. Label_Encode Distribution: The bars are unevenly distributed, which suggests an imbalance among the encoded categories. Some categories are more frequent than others.

```
num_col = [col for col in train_df if train_df[col].dtype
            != object and col != 'id' and col != 'label_encode']

plt.figure(figsize=(12,10))
for i, col in enumerate(num_col):
    plt.subplot(5,3,i+1)
    sns.kdeplot(data=train_df, x=col, label='train_df')
    sns.kdeplot(data=test_df, x=col, label='test_df')
    plt.title(f'{col.title()} distribution', fontweight='bold')
    plt.legend()
    plt.tight_layout()
```

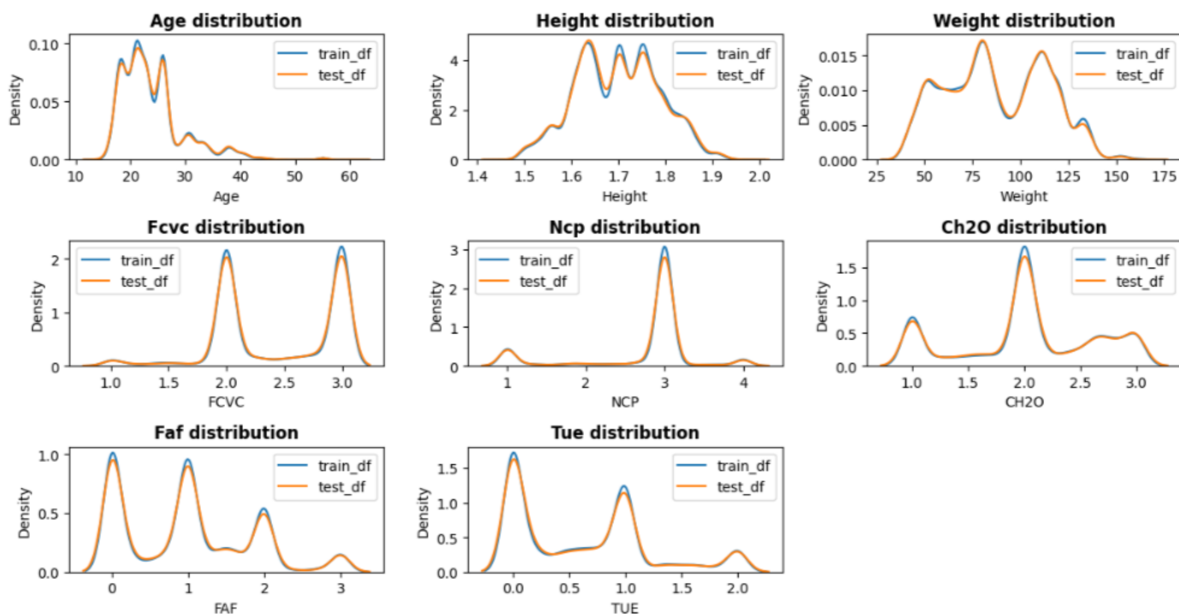


Figure 4: Plot numeric columns for both train_df & test_df.

Age Distribution: Both the training and testing sets have very similar distributions for

age, with multiple peaks. This is indicative of a similar age composition in both datasets. Height Distribution: The distributions of height are closely aligned between the train and test sets, suggesting consistency in the way height is distributed among both groups. Weight Distribution: Weight also shows a near-identical pattern between the datasets, ensuring that any model developed from the training set should generalize well to the test set regarding this feature. FCVC Distribution: The density plots for the frequency of vegetable consumption show that both datasets have similar patterns, with a strong preference towards higher consumption. NCP Distribution: Number of main meals per day appears consistent between the train and test sets, peaking around three meals, which is a common eating pattern. CH2O Distribution: Water consumption shows similar behaviours between the two datasets, with the main peak around 2 liters, which aligns with common hydration recommendations. FAF Distribution: The distribution of physical activity frequency aligns closely between the training and testing sets, with prominent peaks at the lower activity levels and around the midpoint of the scale. TUE Distribution: Time spent using technology devices is similarly distributed between the train and test sets, suggesting that participants' technology use habits are well-represented in both.

```
non_num_train = [col for col in train_df if train_df[col].dtype
=='object' and col != 'NObeyesdad']

plt.figure(figsize=(12,12))
for i, col in enumerate(non_num_train):
    plt.subplot(5,3,i+1)
    sns.violinplot(data=train_df, x=col, y='label_encode',
palette='colorblind')
    plt.title(f'{col} vs label_encode')
    plt.xticks(rotation=20)
    plt.tight_layout()
```

1. Gender vs label_encode: There seems to be a distribution of encoded labels for both genders, with perhaps a slightly wider spread for females, indicating a greater diversity in body weight categories.

2. family_history_with_overweight vs label_encode: Looks at whether having a family history of overweight correlates with the body weight categories. The spread of 'yes' seems to be more towards the higher end of the encoded labels, suggesting that family history might correlate with higher body weight categories.

3. FAVC vs label_encode: Appears to compare the frequency of high caloric food

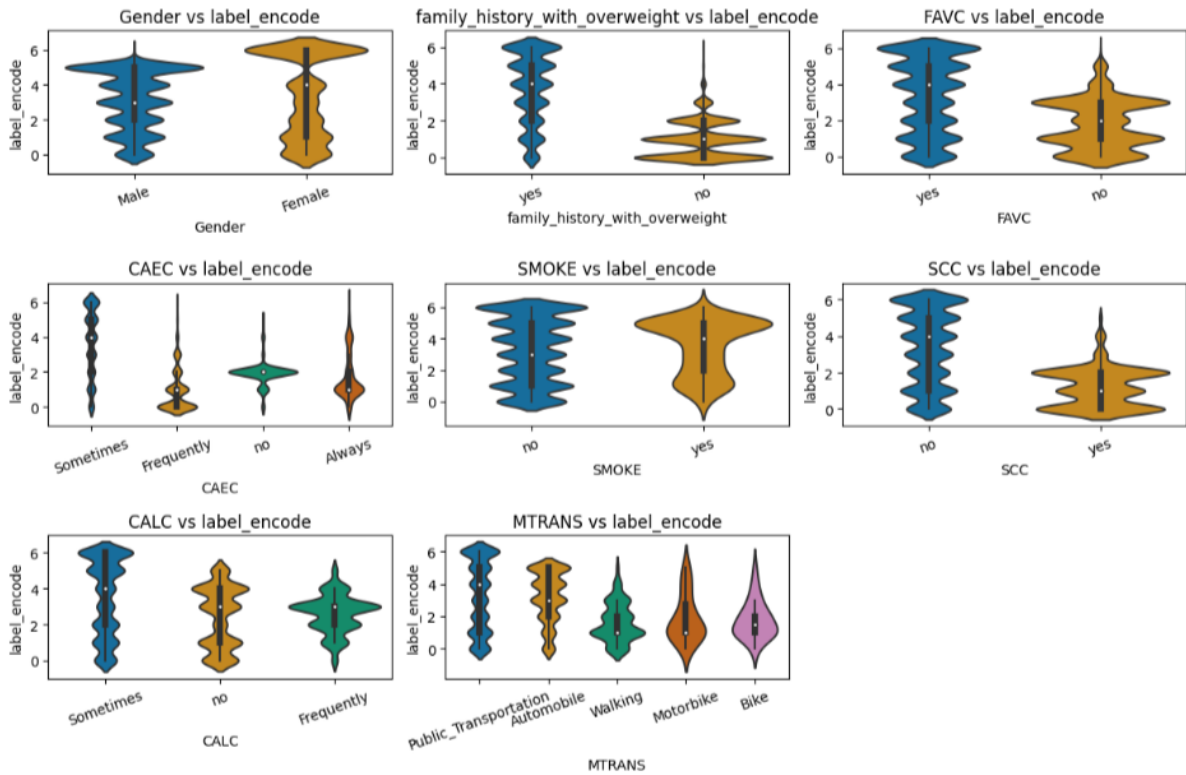


Figure 5: Non-numeric_col vs 'label_encode' train_df.

consumption with body weight categories. Those who consume high caloric food 'yes' seem to have a higher concentration of labels corresponding to overweight or obesity.

4. CAEC vs label_encode: This could stand for something like consumption of food between meals, with categories like 'sometimes', 'frequently', 'always', and 'no'. The distribution of labels varies with these categories, possibly indicating different impacts on body weight.

5. SMOKE vs label_encode: Compares smokers and non-smokers. The distribution seems to be more or less similar across the encoded labels, which might suggest a smaller impact of smoking on the weight categories, at least from this visual.

6. SCC vs label_encode: Could relate to something like self-control concerning eating habits. 'No' self-control seems to have a higher median label encode value than 'yes', suggesting a potential relationship with higher body weight categories.

7. CALC vs label_encode: Might denote the consumption of alcohol, with the categories 'sometimes', 'no', 'frequently'. The distribution suggests that those who consume alcohol 'sometimes' or 'no' seem to spread across the body weight categories differently than those who consume 'frequently'.

8. MTRANS vs label_encode: Looks at the modes of transportation used, with options like 'Public_Transportation', 'Automobile', 'Walking', 'Motorbike', and 'Bike'.

The distribution of weight categories seems to vary with the mode of transportation, which could point to a correlation between physical activity involved in transportation and body weight categories.

```
num_train = [col for col in train_df if (train_df[col].dtype
                        != 'object') and (col != 'id') and
                        (col != 'label_encode')]

plt.figure(figsize=(12,10))
for i, col in enumerate(num_train):
    plt.subplot(5,3,i+1)
    sns.scatterplot(data=train_df, x=col, y='label_encode',
    palette='cubehelix')

    plt.title(f'{col} vs label_encode')
    plt.tight_layout()
```

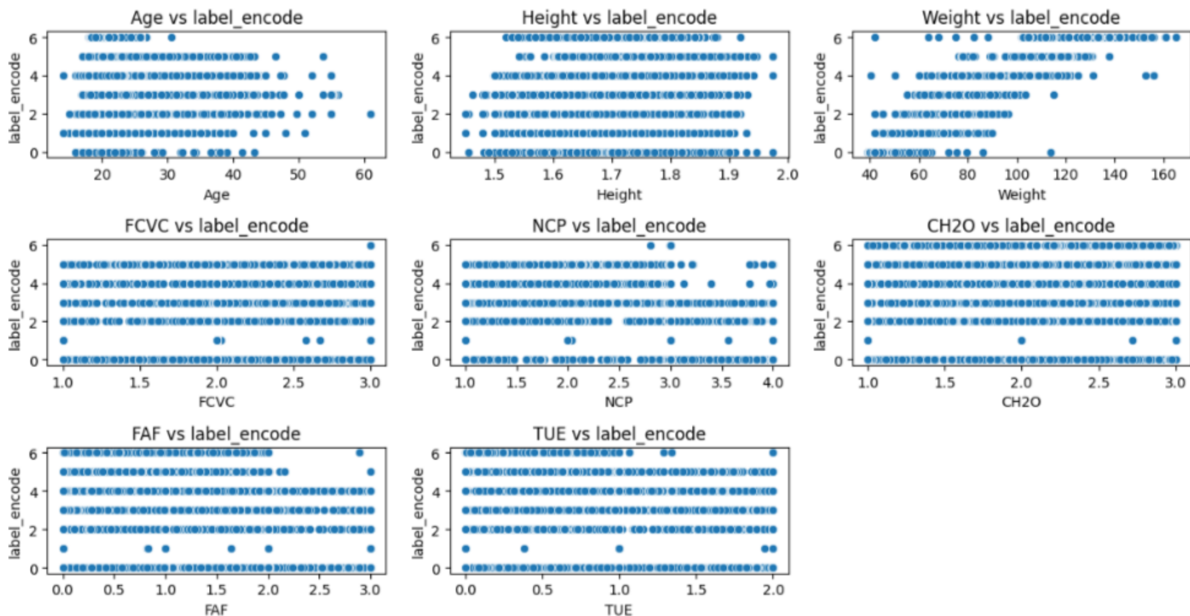


Figure 6: Numeric_col vs 'label_encode' train_df.

1. Age vs label_encode: Shows data points scattered across different ages for each label category. It looks like all age groups are represented across the weight categories, with a relatively even distribution.

2. Height vs label_encode: Height is plotted against weight categories. The data seems to be clustered by height within the weight categories, suggesting there might be a trend where certain heights correspond to certain weight categories.

3. Weight vs label_encode: The relationship between body weight and the weight categories is quite expected – as the actual weight increases, the weight category encoded label also increases.

4. FCVC vs label_encode: This could be related to the frequency of vegetable consumption. The points are spread across all levels of the encoded label, indicating that individuals across all weight categories report various frequencies of vegetable consumption.

5. NCP vs label_encode: The scatter points are evenly distributed, suggesting a similar pattern of meal frequency across all body weight categories.

6. CH2O vs label_encode: There's a spread of intake levels across all weight categories, with no clear pattern of water consumption associated with the weight categories.

7. FAF vs label_encode: Like with CH2O, the points are distributed across all levels of the encoded label, which implies that individuals of all body weight categories have varying levels of physical activity.

8. TUE vs label_encode: Again, we see a broad spread across all weight categories without an immediately obvious pattern.

4.2 Feature engineering

In the feature engineering section of our dataset, the inclusion of Body Mass Index (BMI) is pivotal as it serves as a universally accepted metric combining weight and height to categorize individuals across a spectrum from underweight to obesity, calculated using the formula BMI = weight (kg) / height (m)$\hat{2}$. This new feature offers a robust correlation with the 'label_encode' target variable, which encodes various body weight statuses, thus providing a potential predictor with high relevance for obesity-related outcomes.

```
train_df['BMI'] = train_df['Weight'] / (train_df['Height']**2)
train_df.head()
```

OKE	CH2O	SCC	FAF	TUE	CALC	MTRANS	NObeyesdad	label_encode	BMI
	2.763573	no	0.000000	0.976473	Sometimes	Public_Transportation	Overweight_Level_II	3	28.259565
	2.000000	no	1.000000	1.000000	no	Automobile	Normal_Weight	1	23.422091
	1.910378	no	0.866045	1.673584	no	Public_Transportation	Insufficient_Weight	0	17.126706
	1.674061	no	1.467863	0.780199	Sometimes	Public_Transportation	Obesity_Type_III	6	44.855798
	1.979848	no	1.967973	0.931721	Sometimes	Public_Transportation	Overweight_Level_II	3	25.599151

Figure 7: BMI for train_df.

The BMI values show a wide range of body weight statuses, from underweight to severe

obesity, demonstrating the diversity of the individuals' physical characteristics in the dataset. This range of BMI values can be pivotal for health risk assessments and targeted interventions if the dataset is used for healthcare or wellness applications.

4.3 Data transformation

```
non_num_col = [col for col in train_df if train_df[col].dtype
                    == 'object']

for col in non_num_col:
    print(train_df[col].value_counts())
    print('----------------------------------------')
```

```
CAEC_map = {'no': 0,
            'Sometimes': 1,
            'Frequently': 2,
            'Always': 3}

CALC_map = {'no': 0,
        'Sometimes': 1,
        'Frequently': 2}



train_df['CAEC'] = train_df['CAEC'].map(CAEC_map)
train_df['CALC'] = train_df['CALC'].map(CALC_map)

test_df['CAEC'] = test_df['CAEC'].map(CAEC_map)
test_df['CALC'] = test_df['CALC'].map(CALC_map)
```

```
cate_col = [col for col in train_df if train_df[col].dtype ==
            'object' and col != 'NObeyesdad']

encode_train = pd.get_dummies(train_df[cate_col],
                              drop_first=True)
train_df.drop(columns=cate_col, axis=1, inplace=True)
train_df = pd.concat([train_df, encode_train], axis=1)

encode_test = pd.get_dummies(test_df[cate_col],
                             drop_first=True)
test_df.drop(columns=cate_col, axis=1, inplace=True)
test_df = pd.concat([test_df, encode_test], axis=1)
```

1. Identification and Analysis of Non-Numerical Data: First, we identified non-numerical (object type) columns in the train_df dataset using a list comprehension. For each of these columns, we printed the value counts, providing a summary of the distribution of categories within each variable. This is an essential step for understanding the composition of categorical data, and it prepares us for making informed decisions on how to encode these variables for numerical analysis.

2. Encoding Categorical Variables with Ordinal Values: We then mapped ordinal categorical features, specifically 'CAEC' and 'CALC', to numerical values using predefined dictionaries (CAEC_map and CALC_map). This encoding reflects the inherent order within the categories, with 'no' as the base category assigned a 0, 'Sometimes' a 1, 'Frequently' a 2, and 'Always' a 3 for 'CAEC', while 'CALC' doesn't include an 'Always' category. These mappings were applied to both train_df and test_df, ensuring that the machine learning algorithms can interpret these ordinal features as quantitative data.

3. One-Hot Encoding of Remaining Categorical Variables: Finally, for the remaining categorical variables that do not have an ordinal nature, we implemented one-hot encoding. This technique converts categorical variables into a format that can be provided to machine learning algorithms to better predict the outcome variable without imposing an arbitrary ordinal relationship. Using pd.get_dummies, we transformed categorical columns into multiple binary columns, with each category represented as a separate column. We also dropped the first binary column (drop_first=True) to avoid the dummy variable trap caused by multicollinearity. After one-hot encoding, we removed the original categorical columns from train_df and test_df and concatenated the new binary columns to these dataframes. This step ensures that our models have purely numerical input that accurately represents the information within the categorical variables.

5. Predective model for obesity risk dataset

```
X = train_df.drop(columns=['id','NObeyesdad','label_encode'],
                                        axis=1)
y = train_df['label_encode']

X_test = test_df.drop(columns=['id'])
```

This snippet is crucial for setting up the data for subsequent modelling. It prepares the feature set X and the target variable y for the training process, as well as the feature set X_test for evaluation. X: Features for the training dataset. The code removes columns that are not useful for prediction (id, NObeyesdad) or are target labels (label_encode). This focuses the model training on relevant data. y: Target variable for the training dataset. It uses label_encode, which is likely a preprocessed form of the NObeyesdad column that encodes obesity categories into numerical labels suitable for machine learning models. X_test: Features for the testing dataset. It drops the id column, aligning the test data structure with the training data, ensuring that the models can be applied directly during the evaluation phase.

```
X_train, X_val, y_train, y_val =
    train_test_split(X, y, test_size=0.2, random_state=0)
```

Splits the training data further into training and validation sets, a common practice to evaluate model performance internally before using the test set. The function train_test_split is used to randomly divide the data into a training set (80% of the data) and a validation set (20% of the data), as defined by test_size=0.2. random_state=0 ensures the split is reproducible, meaning the same random selection of rows for each set can be achieved each time the code is run, which is essential for debugging and comparing model performance over iterations.

```
class train_model():
    def __init__(self, model, X_train, X_test,
                 y_train, y_test):
        self.model = model

        self.X_train = X_train
        self.X_test = X_test
        self.y_train = y_train
        self.y_test = y_test

        self.model.fit(self.X_train, self.y_train)

        print(f'1. Model: {model}')


    def model_eval(self):
        y_pred = self.model.predict(self.X_test)

        model_report =
            classification_report(self.y_test, y_pred)

        print('')
        print(f'2. {self.model} model report:')
        print(model_report)
```

Defines a Python class train_model that encapsulates the functionality for training a machine learning model and evaluating its performance. Constructor (__init__): Initializes the model with the provided training and testing datasets, then fits the model to the training data. This step includes immediate output of the model details upon initialization, which helps in identifying the model configuration during runtime. Model Evaluation (model_eval): Makes predictions on the test dataset and generates a classification report, which includes precision, recall, f1-score, and support for each class. This function is critical for understanding how well the model performs across different obesity categories, providing insights necessary for refining the model and selecting the best performer for deployment.

5.1 Logistic Regression model

```
logit_model = train_model(LogisticRegression(),
                          X_train, X_val, y_train, y_val)

logit_model.model_eval()
```

```
1. Model: LogisticRegression()

2. LogisticRegression() model report:
              precision    recall  f1-score   support

           0       0.78      0.80      0.79       478
           1       0.62      0.68      0.65       630
           2       0.53      0.43      0.48       472
           3       0.47      0.34      0.39       510
           4       0.53      0.52      0.52       582
           5       0.77      0.87      0.82       673
           6       0.84      0.92      0.88       807

    accuracy                           0.68      4152
   macro avg       0.65      0.65      0.65      4152
weighted avg       0.66      0.68      0.67      4152
```

Figure 8: Model: LogisticRegression().

The classification report for the Logistic Regression model indicates an overall accuracy of 68%, with the best individual class performance observed in class 6 (precision of 0.84 and recall of 0.92), and relatively lower performance in classes 2 and 3, suggesting that further model tuning is required to improve predictions for these classes.

5.2 KNN model

```
knn_model = train_model(KNeighborsClassifier(),
                        X_train, X_val, y_train, y_val)

knn_model.model_eval()
```

```
1. Model: KNeighborsClassifier()

2. KNeighborsClassifier() model report:
              precision    recall  f1-score   support

           0       0.89      0.93      0.91       478
           1       0.86      0.83      0.84       630
           2       0.74      0.76      0.75       472
           3       0.75      0.79      0.77       510
           4       0.89      0.82      0.86       582
           5       0.96      0.95      0.95       673
           6       0.98      0.99      0.99       807

    accuracy                           0.88      4152
   macro avg       0.87      0.87      0.87      4152
weighted avg       0.88      0.88      0.88      4152
```

Figure 9: Model: KNeighborsClassifier().

The KNeighborsClassifier model's classification report shows high accuracy at 88% with the model performing exceptionally well in classifying class 6 (precision of 0.98 and recall of 0.99) and demonstrating strong predictive capabilities across all classes, as reflected by the consistent F1-scores.

5.3 SVC model

```
svc_model = train_model(SVC(), X_train, X_val, y_train, y_val)

svc_model.model_eval()
```

```
1. Model: SVC()

2. SVC() model report:
              precision    recall  f1-score   support

           0       0.83      0.95      0.89       478
           1       0.84      0.72      0.78       630
           2       0.64      0.64      0.64       472
           3       0.63      0.71      0.67       510
           4       0.81      0.79      0.80       582
           5       0.93      0.89      0.91       673
           6       0.95      0.97      0.96       807

    accuracy                           0.82      4152
   macro avg       0.81      0.81      0.81      4152
weighted avg       0.82      0.82      0.82      4152
```

Figure 10: Model: SVC().

The SVC model achieves an overall accuracy of 82%, with high precision and recall for class 6, indicating robust performance in identifying the highest obesity risk category, while classes 2 and 3 show comparatively lower scores, suggesting potential areas for model improvement.

5.4 Naive Bayes model

```
nb_model = train_model(GaussianNB(), X_train,
                       X_val, y_train, y_val)

nb_model.model_eval()
```

```
1. Model: GaussianNB()

2. GaussianNB() model report:
              precision    recall  f1-score   support

           0       0.74      0.94      0.83       478
           1       0.81      0.55      0.66       630
           2       0.63      0.37      0.47       472
           3       0.50      0.64      0.56       510
           4       0.67      0.64      0.66       582
           5       0.80      0.95      0.87       673
           6       0.97      1.00      0.98       807

    accuracy                           0.75      4152
   macro avg       0.73      0.73      0.72      4152
weighted avg       0.75      0.75      0.74      4152
```

Figure 11: Model: GaussianNB().

The GaussianNB model shows an accuracy of 75%, excelling in classifying class 6 with near-perfect precision and recall, although the model's lower performance on class 2 indicates that certain categories may be less distinguishable or require additional feature refinement.

5.5 Random Forest model

```
rf_model = train_model(RandomForestClassifier(),
                       X_train, X_val, y_train, y_val)

rf_model.model_eval()
```

```
1. Model: RandomForestClassifier()

2. RandomForestClassifier() model report:
              precision    recall  f1-score   support

           0       0.92      0.93      0.92       478
           1       0.86      0.87      0.87       630
           2       0.79      0.76      0.78       472
           3       0.79      0.83      0.81       510
           4       0.90      0.87      0.88       582
           5       0.97      0.96      0.97       673
           6       1.00      1.00      1.00       807

    accuracy                           0.90      4152
   macro avg       0.89      0.89      0.89      4152
weighted avg       0.90      0.90      0.90      4152
```

Figure 12: Model: RandomForestClassifier().

5.6 XGBoost model

```
xgb_model = train_model(XGBClassifier(),
                        X_train, X_val, y_train, y_val)

xgb_model.model_eval()
```

```
1. Model: XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, objective='multi:softprob', ...)
```

Figure 13

```
2. XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, objective='multi:softprob', ...) model report:
              precision    recall  f1-score   support
```

Figure 14

The XGBoost model, yields a high accuracy of 90% with the strongest classification results seen in class 6 (precision and recall both at 1.00), indicating a highly effective model particularly for identifying the most at-risk obesity category.

0	0.93	0.94	0.93	478
1	0.87	0.88	0.88	630
2	0.79	0.77	0.78	472
3	0.79	0.84	0.81	510
4	0.91	0.87	0.89	582
5	0.97	0.96	0.96	673
6	0.99	1.00	1.00	807
accuracy			0.90	4152
macro avg	0.89	0.89	0.89	4152
weighted avg	0.90	0.90	0.90	4152

Figure 15: Model: XGBClassifier.

5.6.1 Train XGBoost model & Parameter Tunning

```
param_grid = {
    'n_estimators':[50, 100],
    'max_depth':[3, 5, 7, 9],
    'learning_rate':[0.01, 0.1],
    'gamma':[0.1, 0.2]
}

grid_search = GridSearchCV(XGBClassifier(tree_method='gpu_hist'),
param_grid, cv=3)
grid_search.fit(X, y)

print(f'1/ Best Parameter: {grid_search.best_params_}.')
print(f'2/ Best Accuracy: {grid_search.best_score_}.')

xgb = XGBClassifier(**grid_search.best_params_)
xgb.fit(X, y)
```

```
1/ Best Parameter: {'gamma': 0.2, 'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 100}.
2/ Best Accuracy: 0.9027364123419886.
```

Figure 16: Output code.

6. Results

During preparation & checking the team conducted a comprehensive review of the dataset to identify the types of features available, and initiate hypothesis testing. The use of functions like head(), info(), and describe() facilitated a deep dive into the data's structure and contents, setting the stage for effective data preprocessing. This thorough data checking ensured that the foundational data used in the models was reliable and robust, a critical step for accurate predictive modeling. In training dataset preparation the section was divided into three crucial parts. The first was data cleaning & exploratory data analysis (EDA) where we analyzed significant patterns and potential relationships within the data using different types of charts. Feature Engineering: New features were created to enhance the dataset's predictive capabilities, notably the introduction of the Body Mass Index (BMI), which provided a strong predictive correlation with obesity risk categories. Data Transformation: Data was transformed into a format suitable for machine learning, including encoding categorical variables and normalizing numerical data, making it ready for effective model training. In the predictive model for obesity risk dataset the predictive modeling stage evaluated various algorithms, with XGBoost showing standout performance. The XGBoost model achieved a remarkable accuracy rate of 90%, significantly higher than other models tested. It was particularly adept at predicting the highest-risk category of obesity (class 6), achieving both precision and recall rates of 100%. This indicates the model's exceptional ability to identify and classify the most at-risk individuals accurately. Detailed parameter tuning was conducted with XGBoost through a systematic grid search, optimizing settings such as the number of estimators, maximum depth, learning rate, and gamma values. This fine-tuning contributed significantly to the model's success, ensuring it was well-adjusted to the specific nuances of the dataset. The results highlighted the superior accuracy and efficiency of the XGBoost model in the predictive analysis of obesity risk. The detailed analysis and comparison with other models demonstrated XGBoost's robustness in handling complex classification tasks and its potential applicability in healthcare settings for predictive diagnostics.

7. Conclusion

The project conclusively demonstrated the effectiveness of advanced machine learning techniques, especially the XGBoost model, in predicting obesity risk based on lifestyle and biometric data. The success of this model offers promising directions for future research and potential application in health monitoring and preventive health interventions. This study not only advanced the team's data analytical skills and project management capabilities but also contributed valuable insights into the fight against obesity, under-

scoring the critical role of data science in addressing major public health issues.

8. Appendix

	id	Gender	Age	Height	Weight	family_history_with_overweight	FAVC	FC'
0	0	Male	24.443011	1.699998	81.669950	yes	yes	2.0
1	1	Female	18.000000	1.560000	57.000000	yes	yes	2.0
2	2	Female	18.000000	1.711460	50.165754	yes	yes	1.8
3	3	Female	20.952737	1.710730	131.274851	yes	yes	3.0
4	4	Male	31.641081	1.914186	93.798055	yes	yes	2.6

(a)

	id	Gender	Age	Height	Weight	family_history_with_overweight	FAVC
0	20758	Male	26.899886	1.848294	120.644178	yes	yes
1	20759	Female	21.000000	1.600000	66.000000	yes	yes
2	20760	Female	26.000000	1.643355	111.600553	yes	yes
3	20761	Male	20.979254	1.553127	103.669116	yes	yes
4	20762	Female	26.000000	1.627396	104.835346	yes	yes

(b)

Figure 17: Outputs of the function .head() for respectively : (a) train_df.head(), (b) test_df.head().

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20758 entries, 0 to 20757
Data columns (total 18 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   id                              20758 non-null  int64
 1   Gender                          20758 non-null  object
 2   Age                             20758 non-null  float64
 3   Height                          20758 non-null  float64
 4   Weight                          20758 non-null  float64
 5   family_history_with_overweight  20758 non-null  object
 6   FAVC                            20758 non-null  object
 7   FCVC                            20758 non-null  float64
 8   NCP                             20758 non-null  float64
 9   CAEC                            20758 non-null  object
 10  SMOKE                           20758 non-null  object
 11  CH2O                            20758 non-null  float64
 12  SCC                             20758 non-null  object
 13  FAF                             20758 non-null  float64
 14  TUE                             20758 non-null  float64
 15  CALC                            20758 non-null  object
 16  MTRANS                          20758 non-null  object
 17  NObeyesdad                      20758 non-null  object
dtypes: float64(8), int64(1), object(9)
memory usage: 2.9+ MB
```

(a)

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13840 entries, 0 to 13839
Data columns (total 17 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   id                              13840 non-null  int64
 1   Gender                          13840 non-null  object
 2   Age                             13840 non-null  float64
 3   Height                          13840 non-null  float64
 4   Weight                          13840 non-null  float64
 5   family_history_with_overweight  13840 non-null  object
 6   FAVC                            13840 non-null  object
 7   FCVC                            13840 non-null  float64
 8   NCP                             13840 non-null  float64
 9   CAEC                            13840 non-null  object
 10  SMOKE                           13840 non-null  object
 11  CH2O                            13840 non-null  float64
 12  SCC                             13840 non-null  object
 13  FAF                             13840 non-null  float64
 14  TUE                             13840 non-null  float64
 15  CALC                            13840 non-null  object
 16  MTRANS                          13840 non-null  object
dtypes: float64(8), int64(1), object(8)
memory usage: 1.8+ MB
```

(b)

Figure 18: Outputs of the function .info() for respectively : (a) train_df.info(), (b) test_df.info().

```
1. train_df columns: 18
   train_df length (no. example): 20758

2. test_df columns: 17
   test_df length (no. example): 13840
```

Figure 19: Output of .shape function applied to train and test datasets.

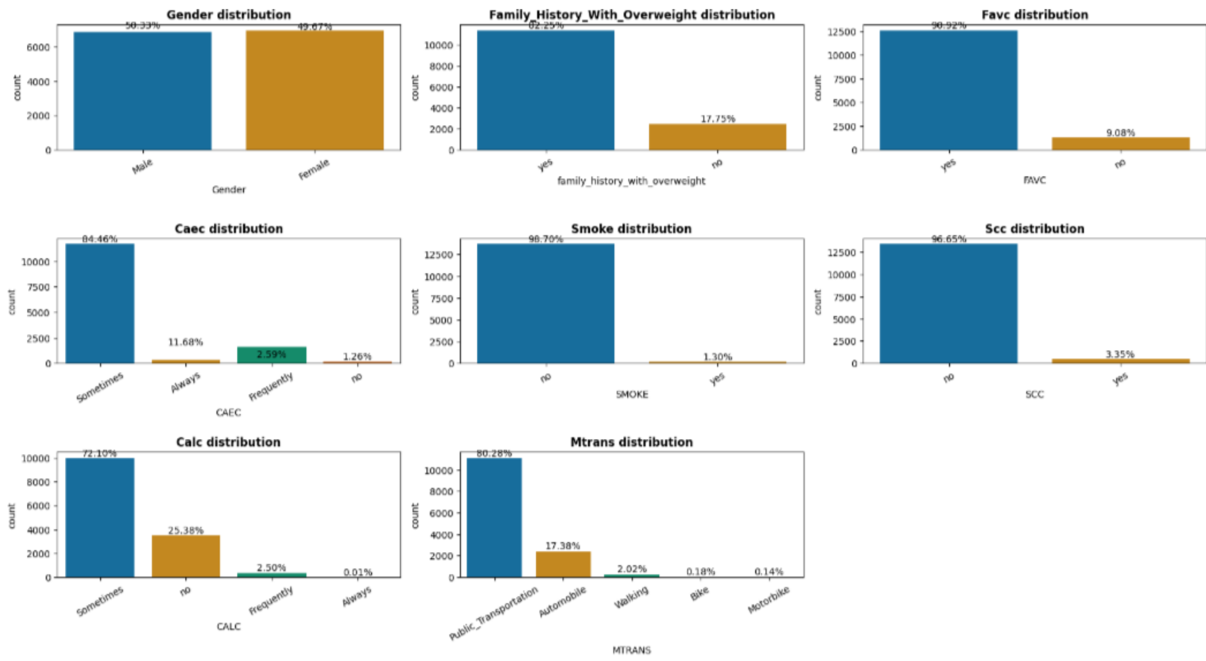


Figure 20: Plotting for non-numeric columns test_df.

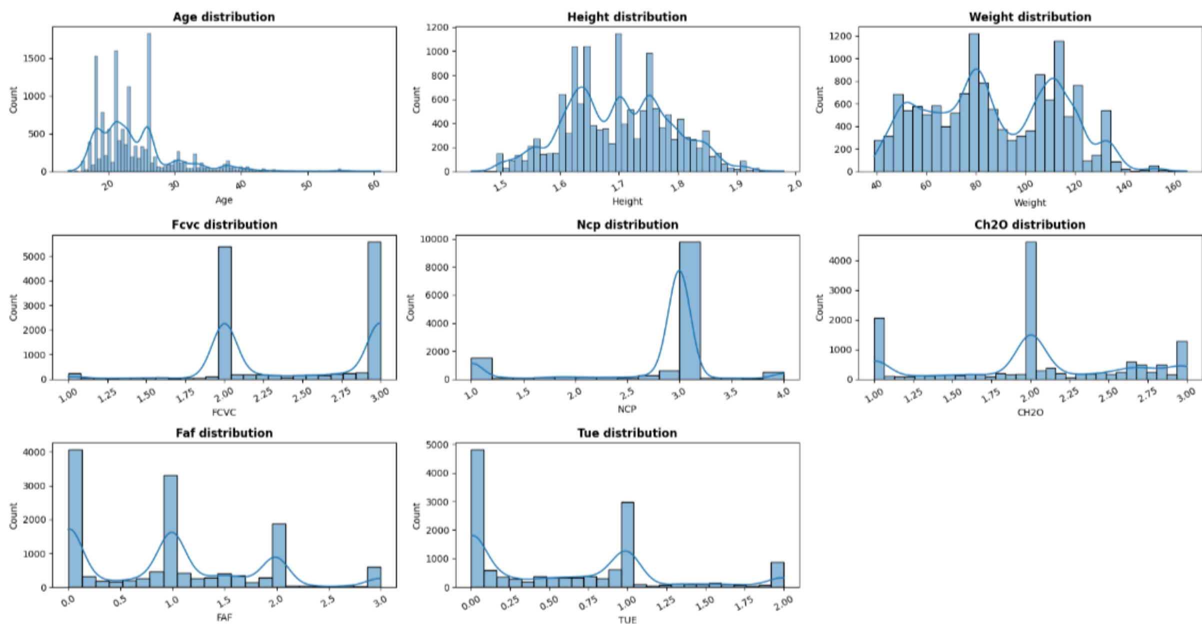


Figure 21: Plotting for numeric columns test_df.

:	NCP	CAEC	SMOKE	CH2O	SCC	FAF	TUE	CALC	MTRANS	BMI
8616	3.000000	Sometimes	no	2.825629	no	0.855400	0.000000	Sometimes	Public_Transportation	35.315411
0000	1.000000	Sometimes	no	3.000000	no	1.000000	0.000000	Sometimes	Public_Transportation	25.781250
0000	3.000000	Sometimes	no	2.621877	no	0.000000	0.250502	Sometimes	Public_Transportation	41.324115
0000	2.977909	Sometimes	no	2.786417	no	0.094851	0.000000	Sometimes	Public_Transportation	42.976937
0000	3.000000	Sometimes	no	2.653531	no	0.000000	0.741069	Sometimes	Public_Transportation	39.584143

Figure 22: BMI for test_df.