AIT511: Course Project 1

Obesity Level Classification Report

Team Name: MT2025065

Member: Keyur Sanjaykumar Padiya

Course: Machine Learning

Date: October 26, 2025

Contents

1 Introduction

Maintaining a healthy lifestyle is becoming increasingly difficult. This dataset investigates how a person's weight category relates to their daily routines, eating habits, physical activity, and demographic information. Building models that can correctly categorize people into groups like inadequate weight, normal weight, overweight, or obesity levels is the task assigned to participants. Features like age, gender, family history, food consumption patterns, physical activity, technology use, and modes of transportation are all included in the dataset. It is a realistic and difficult problem that combines machine learning, behavioral science, and healthcare because of these various factors. Your objective is to create predictive models that can reveal hidden trends in lifestyle choices and advance knowledge of the risk factors for obesity and overweight.

2 Data processing

2.1 Exploratory Data Analysis (EDA)

I did an initial analysis of the dataset. The analysis yielded that the data was *high-quality*, had no missing values across all 15,533 samples and 17 features. The features were in a good mix of 8 numerical and 9 categorical attributes.

2.1.1 Initial Analysis

The data loading confirmed the dataset's high integrity. I performed a basic check on the data's structure, completeness, and feature types.

Data Summary As summarized in Table 1, the dataset consists of 15,533 samples and 17 predictive features, with *zero missing or null values.* This high-quality, complete dataset allowed me to proceed directly to analysis without needing any data imputation.

Table 1: High-Level Data Summary

Attribute	**Count**
Total Samples (Rows)	15,533
Total Features (Columns)	17
Missing (Null) Values	0

Data Dictionary The 17 features are a mix of 8 numerical attributes (floats or integers) and 9 categorical attributes (objects/text). These features, described in Table 2, provide a comprehensive view of each individual's demographics, habits, and physical state.

2.1.2 Target Variable Analysis

The target variable, `WeightCategory`, is a categorical feature with 7 distinct classes. As shown in Figure 1, the classes are relatively. Analysis of the `WeightCategory` target variable confirms it's a classification problem with 7 distinct classes, making regression

Table 2: Data Dictionary and Feature Descriptions

Feature Name	**Data Type**	**Description**
`Age`	Numerical	Age of the individual
`Gender`	Categorical	Gender (Male/Female)
`Height`	Numerical	Height in meters
`Weight`	Numerical	Weight in kilograms
`family_history_...`	Categorical	Family history of overweight (yes/no)
`FAVC`	Categorical	Frequent consumption of high-caloric food (yes/no)
`FCVC`	Numerical	Frequency of vegetable consumption (1-3)
`NCP`	Numerical	Number of main meals (1-4)
`CAEC`	Categorical	Consumption of food between meals (e.g., Sometimes, no)
`SMOKE`	Categorical	Smokes (yes/no)
`CH2O`	Numerical	Daily water consumption (1-3 Liters)
`SCC`	Categorical	Calories consumption monitoring (yes/no)
`FAF`	Numerical	Physical activity frequency (0-3 days/week)
`TUE`	Numerical	Time using technology (0-2 hours/day)
`CALC`	Categorical	Alcohol consumption (e.g., Sometimes, no)
`MTRANS`	Categorical	Transportation method (e.g., Public, Walking)
`WeightCategory`	Categorical	**Target Variable** (7 classes)

models inappropriate. This balance provides adequate samples for the model to learn patterns for each category without needing complex resampling techniques like SMOTE.

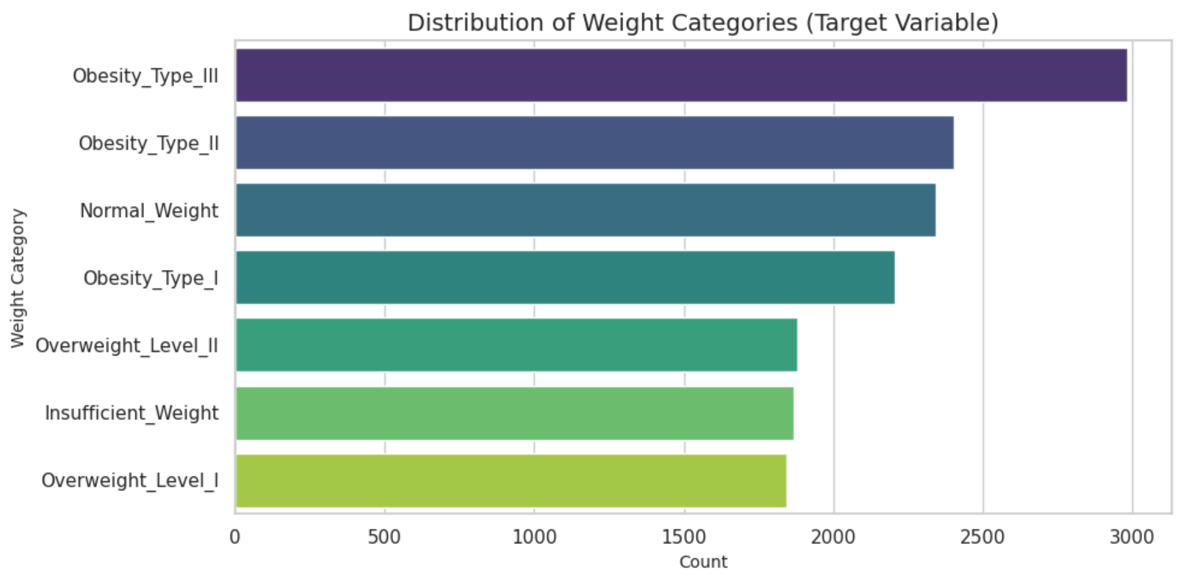

Figure 1: Distribution of the Target Variable (WeightCategory)

2.1.3 Numerical Feature Analysis

Aside from target variable and other categorical features, we have 8 numerical features too. Table 3 provides a summary of the descriptive statistics for these features, illustrating the central tendency and spread, such as the wide range observed in \texttt{Weight} (39kg to 165kg).

Table 3: Descriptive Statistics for Numerical Features

Statistic	**Age**	**Height**	**Weight**	**FCVC**	**NCP**	**CH2O**	**FAF**	**TUE**
Count	15533	15533	15533	15533	15533	15533	15533	15533
Mean	23.82	1.70	87.79	2.44	2.76	2.03	0.98	0.61
Std Dev	5.66	0.09	26.37	0.53	0.71	0.61	0.84	0.60
Min	14.00	1.45	39.00	1.00	1.00	1.00	0.00	0.00
25% (Q1)	20.00	1.63	66.00	2.00	3.00	1.80	0.01	0.00
50% (Median)	22.77	1.70	84.00	2.34	3.00	2.00	1.00	0.57
75% (Q3)	26.00	1.76	111.60	3.00	3.00	2.53	1.58	1.00
Max	61.00	1.98	165.06	3.00	4.00	3.00	3.00	2.00

Analysis of the numerical features revealed varied distributions; histograms showed some features like `Weight` are roughly *normal*, while `FAF` (physical activity) is *skewed* and some other features which can be seen in Figure 2. This variability highlights that non-parametric models like Decision Trees or XGBoost can better capture these mixed feature distributions.

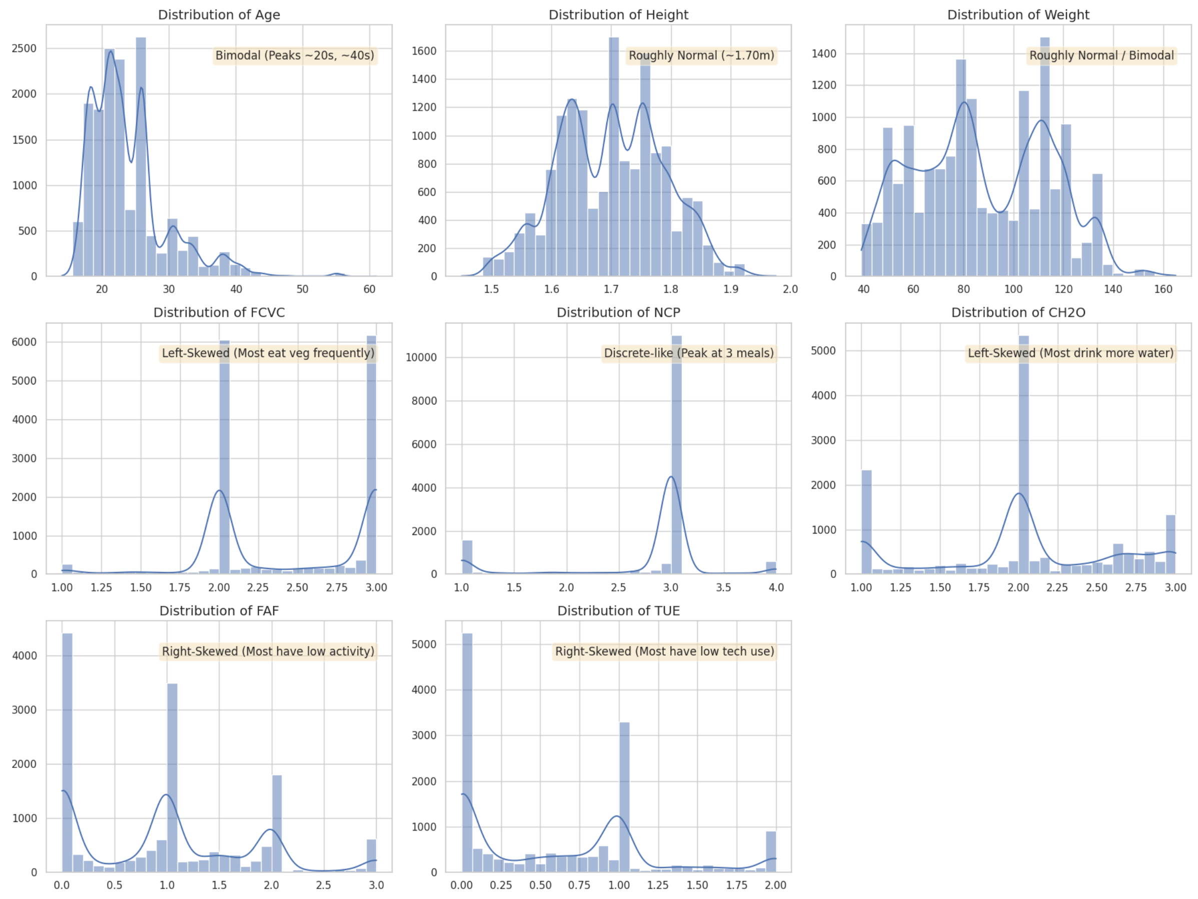

Figure 2: Distribution of the WeightCategory

Box plots (Figure 4) highlight `Weight` as the dominant predictor, with its distribution almost completely separating the target classes. `Age` and `Height` show distinct trends. Features like `FAF` demonstrates complex non-linear relationships sharing similar median values for both low and high weight categories. This analysis is evident for non-linearity and it strongly justifies the point of using non-linear Classification models such as *Decision Trees*, *Random Forests*, and *Gradient Boosting/XGBoost.*

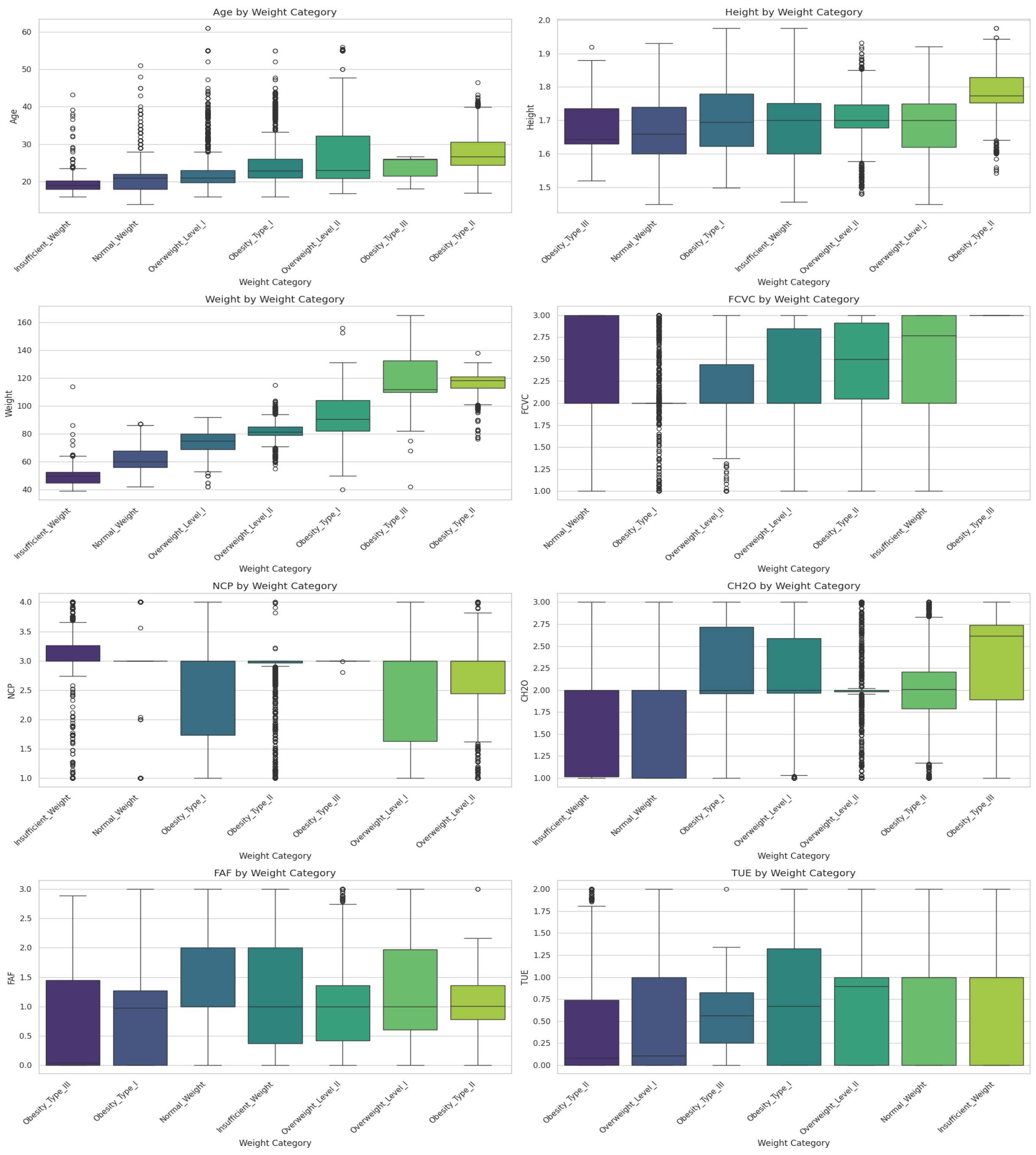

Figure 3: Distribution of Numerical Features Across Weight Categories

2.1.4 Categorical Feature Analysis

The 9 categorical features also proved to be highly predictive. As seen in Figure **??**, features like `family_history_with_overweight` and `MTRANS` (Transportation) have a very strong correlation with obesity. For example, individuals with `Obesity_Type_III` almost exclusively have a family history and predominantly use `Public_Transportation`. These strong categorical signals are ideal for tree-based models.

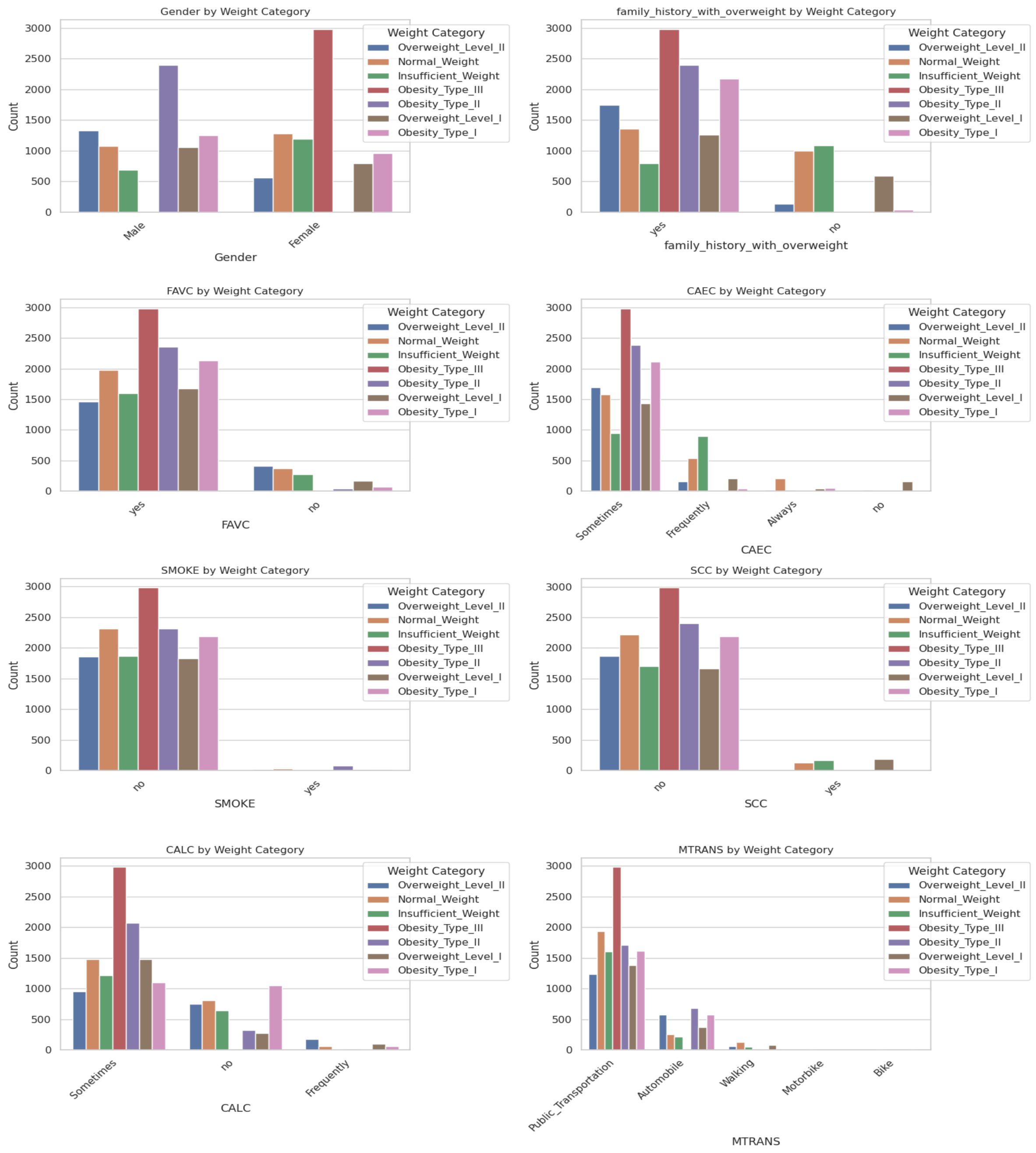

Figure 4: categorical$_{f}eatures_{v}s_{t}arget$

2.1.5 Feature Importance and Correlation

A correlation heatmap (Figure 5) showed moderate multicollinearity between some numerical features (e.g., ‘Height‘ and ‘Weight‘). While this would be a problem for linear models, it is easily handled by tree-based ensembles.

To confirm my findings, I trained a Random Forest model on the full dataset to rank all features. The resulting importance plot (Figure **??**) confirms the EDA’s findings: `Weight`, `Age`, and `Height` are the top 3 predictors, followed by a mix of lifestyle features like `FCVC`, `Gender`, and `CH2O`. Importantly, even the weakest features (like ‘SMOKE‘ and ‘SCC‘) showed non-zero importance, justifying my decision to keep all features for the final model.

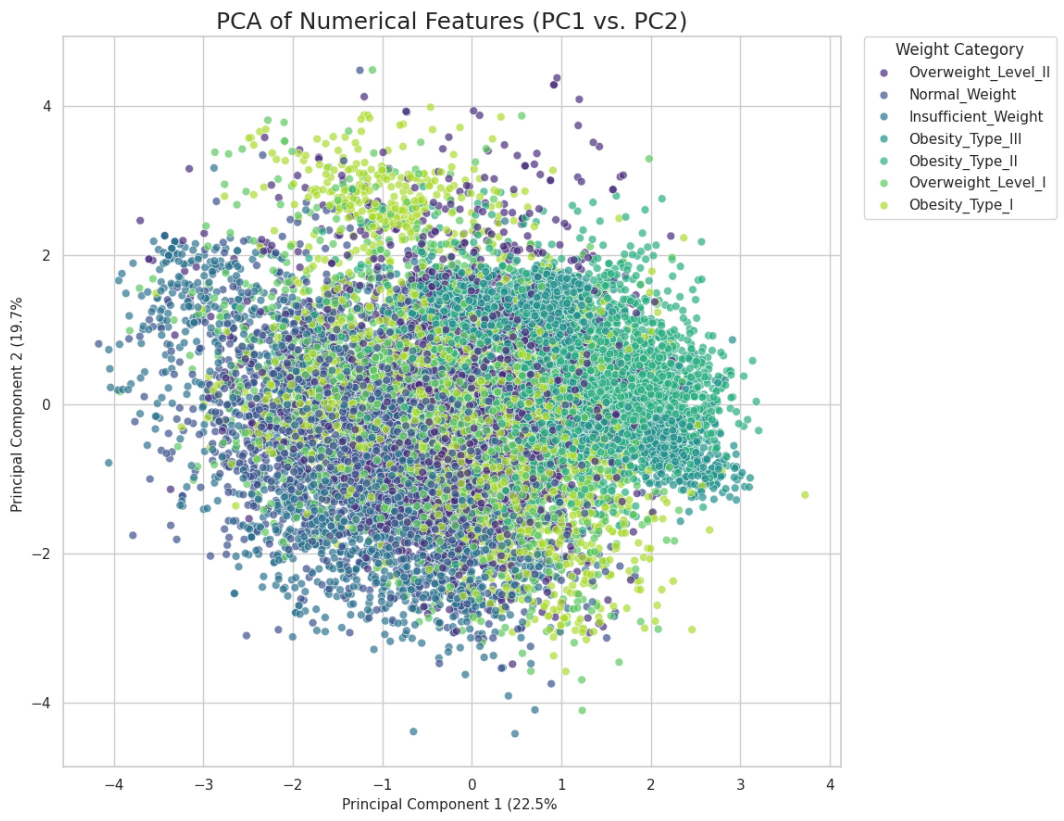

Figure 5: Distribution of the WeightCategory

2.2 Data Preprocessing Pipeline

Data preprocessing encompasses a series of techniques used to clean, transform, and organize raw data before feeding it into machine learning models. Quality preprocessing directly impacts model performance, accuracy, and reliability.Data preprocessing is a critical step in machine learning pipelines that directly impacts model performance.

2.2.1 Data Loading & Exploration

By examining the shape and types, you can check if the dataset is imbalanced, contains enough observations for learning, and if the data types are compatible with your model. The raw training and test datasets were successfully loaded, and non-predictive

id columns were removed. Features (X) and the target (y) were cleanly separated, establishing distinct datasets for training the preprocessors and the final model, and for generating predictions.

2.2.2 Identify Feature Types

Before applying specific transformations, we must identify which columns are numerical and which are categorical. This allows us to apply the correct preprocessing step (scaling or encoding) to each type. The script correctly identified 8 numerical and 9 categori-

cal features based on their data types, enabling the application of appropriate, distinct transformations (scaling vs. encoding) in the subsequent steps.

2.2.3 Handle Categorical Features (One-Hot Encoding)

The EDA identified 9 categorical features. Tree-based models like XGBoost require numerical input. One-Hot Encoding converts these text-based categories into binary (0/1) columns, allowing the model to process them. Using drop_first=True avoids creating perfectly collinear features. Crucially, the columns must be aligned between the training and test sets after encoding to ensure they have the exact same structure. Applying One-Hot

Encoding... Training data shape after OHE: (15533, 22) Test data shape after OHE: (5225, 22) Aligning columns between train and test sets... Columns aligned. Shape after alignment: X=(15533, 22), Xtestofficial=(5225, 22)

2.2.4 Handle Numerical Features

The 8 numerical features have different scales. While XGBoost is somewhat robust to this, applying Standard Scaling standardizes the features (mean=0, std dev=1). This can sometimes help the gradient boosting process converge more smoothly. The scaler is fitted only on the training data and then used to transform both train and test sets to prevent data leakage. I did not use RobustScaler instead of StandardScaler because

the dataset lacks significant outliers, making mean-variance scaling sufficiently effective. Additionally, StandardScaler is computationally faster and more commonly adopted for features without heavy skewness or extreme values.

2.2.5 Handle Target Variable

The target variable WeightCategory needs to be converted from text labels to integers (0, 1, 2... 6) for XGBoost's multi:softmax objective. Label Encoding achieves this mapping. Label Encoding successfully transformed the 7 distinct text labels of the WeightCategory

target variable into the required integer format (0-6) for XGBoost's multi-class classification objective (multi:softmax). The "multi:softmax" objective is used for multi-class

classification problems where each sample belongs to exactly one class and that's why we used it here. It directly predicts the class label by applying the softmax function to output mutually exclusive class predictions, making it suitable for multi-class tasks with discrete labels.

3 Model used

The Project objective is to predict discrete WeightCategory labels, which is a multi-class classification problem. Models such as Linear Regression, L1 Regularization, L2 Regularization and Polynomial Regression are designed for predicting continuous numerical values and are inappropriate for this problem. The Exploratory Data Analysis confirmed the categorical nature of the target variable, so we will focusing more on classification algorithms.

3.1 Naive Bayes Classifier

I implemented the Naive Bayes classifier as a baseline probabilistic approach. But, it achieved a relatively low accuracy because of violation of its assumption of feature independence. EDA showed correlations between some features and thus violating the principles of Naive Bayes Classifier.

3.2 K-Nearest Neighbors (KNN) (using PCA)

I implemented the K-Nearest Neighbors (KNN) classifier, applying Principal Component Analysis (PCA). This approach yielded a low accuracy as the fundamental issue likely lies in PCA itself. PCA's feature transformation dilutes the strong weighted predictors like Weight and merges them with low weighted components.

3.3 Bagging

Then, I explored first ever ensemble method, the Bagging Classifier which aims to improve stability and accuracy by training multiple trees on random subsets of data and combining their predictions.Bagging effectively captures the dataset's complex patterns and reduces overfitting by averaging predictions from multiple decision trees trained on different data subsets. I was able to breakthrough 90+ percent accuracy using bagging but soon reached a threshold because of it's randomness and no good pattern recognition of features.

3.4 Random Forest

Random Forest's is a special type of Bagging method only, and due to same reason of decorrelation due to random independent tree building, it also didn't give quite upgrade in the accuracy. Because of this reason I had to move towards boosting mechanisms which sequentially focuses on misclassified data for improving the accuracy.

3.5 Adaptive(ADA) Boosting

I implemented AdaBoost as the first boosting technique, which focuses sequentially on samples that previous weak learners misclassified. It's best accuracy was lower than Bagging and Random Forest, which suggested that while boosting's focus on errors was good, more advanced boosting methods like Gradient Boosting and XGBoost might be useful because of their focus on the residual errors. They might handle the complexity of data more effectively and could achieve higher accuracy.

3.6 Gradient Boosting

Following AdaBoost, I implemented the Gradient Boosting method, which builds trees iteratively, with each new tree attempting to predict and correct the *residual errors* made by the ensemble so far. Although my peak accuracy was improved, but optimizing Gradient Boosting proved computationally intensive and required extensive tuning around some parameters, it was hard to consistently reach and stabilize near its peak performance. Due to these inefficiencies, I transitioned to XGBoost, an optimized and regularized implementation of gradient boosting which is known for its efficiency and ability to often achieve superior accuracies.

3.7 Extreme Gradient (XG) Boosting

Finally, I focused on XGBoost, an advanced and highly optimized boosting method. XGBoost incorporates several key improvements into gradient boosting like regularization (L1 and L2, often called alpha and lambda) to prevent overfitting and optimized tree construction. These properties make it well-suited for datasets like this one, which has complex interactions, non-linear relationships, and a mix of strong and weak predictors, as identified during the EDA. I was able to achieve the peak accuracy of this project because of extensive hyperparameter tuning (will be discussed in the next section).

4 Hyper parameter tuning

For simpler models like Naive Bayes, KNN with PCA, Bagging, and AdaBoost, manual tuning was sufficient to achieve reasonable performance. However, when working with XGBoost, a gradient boosting model with numerous interdependent hyperparameters, manual tuning became impractical. I initially attempted GridSearchCV to systematically explore parameter combinations, but this approach proved highly inefficient. A single GridSearch run with a modest parameter grid took several hours to complete, and the resulting accuracy improvements were minimal. The exhaustive nature of GridSearch meant it wasted significant computational resources evaluating poor parameter combinations that were clearly suboptimal early in training. Given these limitations, I decided to transition to Optuna, a Bayesian optimization framework that uses intelligent search strategies to explore the parameter space more efficiently.

Optuna offered several key advantages over GridSearchCV for tuning XGBoost. Unlike GridSearch which tests every combination exhaustively, Optuna employs Tree-structured Parzen Estimator (TPE) sampling, which learns from previous trials to intelligently suggest promising parameter regions. This Bayesian approach means each trial informs the next, leading to faster convergence toward optimal parameters. Second, I implemented MedianPruner to automatically terminate unpromising trials early during cross-validation, saving substantial computational time without sacrificing search quality. My tuning workflow used Repeated Stratified K-Fold cross-validation (5 folds, 2 repeats) to ensure robust validation accuracy estimates and prevent overfitting to any single validation split. I systematically tuned critical XGBoost parameters including gamma (regularization on leaf node splits, range: 0.3-0.7), max_depth (tree depth limit, range: 7-9), learning_rate (shrinkage factor, range: 0.02-0.06), reg_alpha (L1 regularization, range: 0.1-0.5), reg_lambda (L2 regularization, range: 1.2-12.0), n_estimators (number of boosting rounds, range: 600-800), subsample (row sampling ratio), and colsample_bytree (column sampling ratio).

During the tuning process, I encountered several significant challenges. Initially, my wide search ranges (e.g., max_depth: 5-12, learning_rate: 0.005-0.1) led to extremely long search times, taking 4-5 hours for 100 trials with limited accuracy improvements. Additionally, I discovered a disconnect between cross-validation accuracy and test set accuracy, my best CV score was 0.9092, but the corresponding test accuracy was only 0.9107, while other trials achieved higher test accuracy (0.9132) despite lower CV scores. This taught me that relying solely on validation metrics can be misleading. I also attempted SMOTE oversampling to handle class imbalance, but it paradoxically decreased both validation and test accuracy, forcing me to abandon that approach. Furthermore, introducing scale_pos_weight for class weighting did not improve overall accuracy as expected. Through iterative refinement, I narrowed my search ranges based on observed patterns particularly identifying that gamma values between 0.62-0.66 and learning rates around 0.025-0.032 produced consistently strong test results. This second-phase targeted search with tighter ranges converged quickly and ultimately discovered my best parameters, achieving a final test accuracy of 0.91322, validating the benefits of intelligent, data-driven hyperparameter optimization over brute-force approaches.

4.1 Hyperparameter Tuning ranges for XGBoost:-

4.1.1 Manual Tuning 1

First I did some manual tunings to test the ranges like above :-

Changed Parameter	**Setting**	**Accuracy**
Baseline	Basic settings	0.90473
Learning rate	0.01 LR	0.90666
Learning rate + Subsample	0.01 LR, 0.9 subsample	0.90936
Learning rate + Subsample + Colsample	0.01 LR, 0.9 subsample, 0.9 colsample_bytree	0.90666
Learning rate + Subsample + Max depth	0.01 LR, 0.9 subsample, max_depth=7	0.90505
Learning rate + Subsample + Max depth	0.01 LR, 0.9 subsample, max_depth=5	0.90183
Number of trees	2000 trees, 0.01 LR, 0.9 subsample	0.90602
Number of trees	1500 trees, 0.01 LR, 0.9 subsample	0.90602
Number of trees	1200 trees, 0.01 LR, 0.9 subsample	0.90602
Learning rate	0.005 LR, 0.9 subsample	0.90005
Random state	0.01 LR, 0.9 subsample, random_state=50	0.90537
Learning rate	0.02 LR, 0.9 subsample	0.90634
Learning rate	0.015 LR, 0.9 subsample	0.90505
Subsample	0.01 LR, 0.95 subsample	0.90763
Subsample	0.01 LR, 1 subsample	0.90666
Colsample	0.01 LR, 0.9 subsample, 0.7 colsample_bytree	0.90537

Table 4: XGBoost parameter tuning experiments (other parameters fixed: *n_estimators* = 1000, *learning_rate* = 0.05, *max_depth* = 6, *subsample* = 0.8, *colsample_bytree* = 0.8).

From this I got the best output accuracy of **.90936** for the hyperparameters:-

objective = 'multi:softmax',
num_class = len(y.unique()),
n_estimators = 1000,
learning_rate = 0.01,
max_depth = 6,
subsample = 0.8,)
colsample_bytree = 0.8,
use_label_encoder = False,
eval_metric = 'mlogloss',
random_state = 42,
early_stopping_rounds = 50

4.1.2 GridSearchCV 1

I started with basic GridSearchCV thinking exhaustive search would find optimal parameters. I used a simple fixed grid:

$$\text{param\_grid} = \begin{cases} \text{n\_estimators} & : [1000, 1100, 1200] \\ \text{learning\_rate} & : [0.009, 0.01, 0.011] \\ \text{max\_depth} & : [5, 6, 7] \\ \text{subsample} & : [0.90, 0.92, 0.95] \\ \text{colsample\_bytree} & : [0.90, 0.92, 0.95] \end{cases}$$

These Hyperparameters gave me an accuracy of .90771 which wasn't good and it took a lot of time, hence I went for a better search methods like Optuna Search and RandomizedSearchCV.

4.1.3 Optuna Search 1

As I moved forward, I introduced new Regularization hyperparameters like Alpha and Lambda which control model complexity.

$$\text{param\_search} = \begin{cases} \text{n\_estimators} & : [900, 1200] \\ \text{learning\_rate} & : [0.009, 0.011] \\ \text{max\_depth} & : [5, 7] \\ \text{subsample} & : [0.90, 0.95] \\ \text{colsample\_bytree} & : [0.90, 0.95] \\ \text{reg\_alpha} & : [0.8, 1.0] \\ \text{reg\_lambda} & : [0.8, 1.2] \end{cases}$$

This gave an Validation Accuracy of 0.90581 and an Output/Test Accuracy: 0.90761, so no improvements.

4.1.4 RandomizedSearchCV search 1

I tried RandomizedSearchCV with a much wider range across ALL parameters:

$$\text{param\_dist} = \begin{cases} \text{n\_estimators} & : [800, 900, 1000, 1100, 1200] \\ \text{learning\_rate} & : [0.001, 0.1] \\ \text{max\_depth} & : [4, 10] \\ \text{subsample} & : [0.6, 1.0] \\ \text{colsample\_bytree} & : [0.6, 1.0] \\ \text{min\_child\_weight} & : [1, 10] \\ \text{reg\_alpha} & : [0.0, 0.6] \\ \text{reg\_lambda} & : [0.5, 4.0] \\ \text{gamma} & : [0.0, 0.5] \end{cases}$$

This gave an Validation Accuracy of 0.90431 and an Output/Test Accuracy: 0.90661, so no improvements.

4.1.5 Optuna Search 2

Then I tried to search the best values of lambda and alpha in hope for accuracy jump with these parameters:

$$\text{params} = \begin{cases} \text{n\_estimators} & : 1000 \\ \text{learning\_rate} & : 0.01 \\ \text{max\_depth} & : 6 \\ \text{subsample} & : 0.8 \\ \text{colsample\_bytree} & : 0.8 \\ \text{reg\_alpha} & : [0.15, 0.55] \\ \text{reg\_lambda} & : [0.8, 3.2] \end{cases}$$

This gave an Validation Accuracy of 0.905749 and an Output/Test Accuracy: 0.91019, so a breakthrough was achieved. Focusing on fewer parameters with Optuna allowed it to find high-quality solutions.

4.1.6 Optuna Search 3

I tweaked some things in this iteration, I used use very low learning rate with many trees in order to improve the accuracy to balance the higer number of trees:

$$\text{params} = \begin{cases} \text{n\_estimators} & : [1500, 2000] \\ \text{learning\_rate} & : [0.003, 0.007] \\ \text{max\_depth} & : [5, 7] \\ \text{subsample} & : [0.75, 0.85] \\ \text{colsample\_bytree} & : [0.75, 0.85] \\ \text{reg\_alpha} & : [0.1, 0.5] \\ \text{reg\_lambda} & : [0.8, 2.5] \end{cases}$$

Due to higher number of trees, this search took a lot of time and there wasn't enough increase in the accuracy.

4.1.7 Optuna Search 4

I broadened the search significantly after realizing Stages 3-6 needed a different approach. Parameter *n_estimators* with range 500-900 seemed reasonable. I expanded max_depth to 12 because maybe shallower trees might limit performance :

$$\text{params} = \begin{cases} \text{n\_estimators} & : [500, 900] \\ \text{learning\_rate} & : [0.01, 0.05] \\ \text{max\_depth} & : [6, 12] \\ \text{subsample} & : [0.6, 0.95] \\ \text{colsample\_bytree} & : [0.3, 0.8] \\ \text{min\_child\_weight} & : [1, 5] \\ \text{reg\_alpha} & : [0.0, 1.0] \\ \text{reg\_lambda} & : [0.1, 50.0] \end{cases}$$

This gave an Validation Accuracy of 0.90800 and an Output/Test Accuracy: 0.91157, so a breakthrough was achieved. A Solid improvement, so I learned that Moderate ranges with intelligent exploration worked better than both too narrow and too random.

4.1.8 Optuna Search 5

Then i did one another optuna search but no improvement was found so, I introduced one extra parameter *gamma* which is a regularization parameter that controls whether a node will be split in a tree or not. For starters I did some manual tunings on my best parameters till now in the range of:

$$\text{base\_params} = \begin{cases} \text{n\_estimators} & : 1390 \\ \text{learning\_rate} & : 0.00680 \\ \text{max\_depth} & : 8 \\ \text{subsample} & : 0.85676 \\ \text{colsample\_bytree} & : 0.50965 \\ \text{min\_child\_weight} & : 7 \\ \text{reg\_alpha} & : 0.15512 \\ \text{reg\_lambda} & : 3.51511 \end{cases}$$

$$\gamma\text{\_range} = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]$$

The results were quite surprising and shocking at the same time, as *0.5* value of *gamma* gave an Validation Accuracy of 0.90856 and an Output/Test Accuracy: 0.91239. This was a huge jump in accuracy. So my hunt for the best value of *gamma* began.

4.1.9 Optuna Search 6

Then I tried one more search centered around the best parameters I found previously, but it's top accurate parameters were just overfitting, luckily I got one more parameter with an accuracy of .91239. Then for reducing the overfitting, I tried to reduce model complexity by reducing trees and balancing other parameters with this range:

$$\text{params} = \begin{cases} \text{n\_estimators} & : [500, 900] \\ \text{max\_depth} & : [6, 10] \\ \text{learning\_rate} & : [0.015, 0.030] \\ \text{reg\_alpha} & : [0.3, 1.0] \\ \text{reg\_lambda} & : [20.0, 50.0] \\ \text{gamma} & : [0.5, 1.0] \\ \text{min\_child\_weight} & : [3, 8] \\ \text{subsample} & : [0.70, 0.85] \\ \text{colsample\_bytree} & : [0.50, 0.70] \end{cases}$$

The results were not good, got a validation accuracy of 0.90668 and Output/Test Accuracy of 0.90716.

4.1.10 Optuna Search 7

Previous failures led me realise that I needed a balanced approach, not too little complexity, not much. and So, i tried to keep my trees and depth low and would balance my regularization values accordingly:

$$\text{params} = \begin{cases} \text{max\_depth} & : [7, 12] \\ \text{n\_estimators} & : [550, 700] \\ \text{learning\_rate} & : [0.001, 0.5] \\ \text{reg\_alpha} & : [0.15, 0.5] \\ \text{reg\_lambda} & : [10.0, 25.0] \\ \text{gamma} & : [0.3, 0.7] \\ \text{min\_child\_weight} & : [1, 3] \\ \text{subsample} & : [0.60, 0.90] \\ \text{colsample\_bytree} & : [0.4, 0.7] \end{cases}$$

Luckily, I Found the sweet spot! This is where an output/test accuracy of 0.91267 first appeared, So i was in the right path.

4.1.11 Optuna Search 8

After this peak performance followed a huge range of overfitting parameters, I did 2 searches in hope for an improvement but it didn't came. So I analyzed top 10 accurate trials of my last 3 optuna searches and decided to make a tight ranged optuna search for the best parameters like:

$$\text{params} = \begin{cases} \text{n\_estimators} & : [550, 900] \\ \text{max\_depth} & : [7, 13] \\ \text{learning\_rate} & : [0.008, 0.035] \\ \text{reg\_alpha} & : [0.10, 0.5] \\ \text{reg\_lambda} & : [5.0, 25.0] \\ \text{gamma} & : [0.25, 1] \\ \text{min\_child\_weight} & : [1, 6] \\ \text{subsample} & : [0.60, 0.95] \\ \text{colsample\_bytree} & : [0.35, 1] \end{cases}$$

The results were almost the same, got a validation accuracy of 0.90887 and Output/Test Accuracy of 0.91267(from top 10 accuracies).

4.1.12 Optuna Search 9

I maintained that good habit of always checking the top 10 accuracies of an optuna search which led to my best and the final accuracy till date. I did and extensive optuna search on the top 10 accuracies of the last search and I prioritized my top accuracies till date and made this optuna search range :

$$\text{params} = \begin{cases} \text{n\_estimators} & : [600, 800] \\ \text{learning\_rate} & : [0.02, 0.06] \\ \text{max\_depth} & : [7, 9] \\ \text{subsample} & : [0.5, 0.9] \\ \text{colsample\_bytree} & : [0.42, 0.52] \\ \text{min\_child\_weight} & : [1, 2] \\ \text{gamma} & : [0.3, 0.7] \\ \text{reg\_alpha} & : [0.1, 0.5] \\ \text{reg\_lambda} & : [1.2, 12.0] \end{cases}$$

The results were really good as it worked, got avalidation accuracy of 0.90897 and Output/Test Accuracy of 0.91132(from top 10 accuracies). The most optimal hyperparameters with a 0.91132 output/test accuracies are:-

’n_estimators’ : 685,
’learning_rate’ : 0.03106,
’max_depth’ : 9,
’subsample’ : 0.5036,
’colsample_bytree’ : 0.4609,
’min_child_weight’ : 1,
’gamma’ : 0.6262,
’reg_alpha’ : 0.4449,
’reg_lambda’ : 1.2909

5 Performance evaluation

Table 5: **Performance Comparison Across All Models**

Model	**Validation Accuracy (%)**	**Precision**	**Recall**	**F1-score**	**Test Accuracy (%)**
Naive Bayes	63.31	0.63	0.63	0.63	63.31
KNN (with PCA)	71.76	0.72	0.72	0.72	71.76
Bagging Classifier	89.86	0.89	0.89	0.89	90.69
Random Forest	89.41	0.88	0.88	0.88	90.58
AdaBoost	89.19	0.88	0.88	0.88	89.61
Gradient Boosting	90.45	0.90	0.90	0.90	91.02
XGBoost	90.92	0.90	0.90	0.90	91.32

Due to its combination of exceptional accuracy, flexibility, and computational efficiency on structured/tabular data, XGBoost consistently outperforms other algorithms for classification problems like obesity prediction, as supported by both validation and test performance metrics in the comparative analysis.

6 Conclusion

XGBoost stands out as the best performer because it offers high accuracy, efficient handling of both numerical and categorical features, and robust resistance to overfitting through built-in regularization. It supports parallel processing, scales well to large datasets, and efficiently manages missing values and outliers. The algorithm is highly customizable, provides interpretable feature importance scores, and is heavily optimized for speed and memory efficiency.

7 GitHub Repository Link

The full source code for this project is available at: `https://github.com/KeyPad717/Obesity-Level-Classification`