Predicting Obesity Levels Using Machine Learning and Deep Learning Methods

Mentor: Narendra Kumar

Team Members:

1. Abhishek Mane
2. Ankit Sharma
3. Himanshu Tayade
4. Ankita Gupta
5. Venkata Manvitha Tatikonda
6. A Yamini
7. Archana Reddy Mamidi

Project Description

The goal of this project is to develop a predictive model to assess obesity levels based on eating habits and physical conditions using machine learning (ML) and deep learning (DL) techniques. By leveraging data from the "Obesity based on eating habits and physical conditions" dataset on Kaggle, the project aims to build a system that can accurately classify obesity levels and help in identifying at-risk individuals based on their lifestyle patterns.

Table of Contents

CHAPTER 1 ii

INTRODUCTION iii

1.1 Introduction iii

1.2 Project Outcomes iii

CHAPTER 2 iii

ARCHITECTURE DIAGRAM iii

CHAPTER 3 iv

PROPOSED SYSTEM iv

3.1 Data Description iv

3.2 Data Preprocessing and Exploration v

3.2.1 Creating Target Column v

3.2.2 Encoding Categorical Variables v

3.2.3 Scaling the Features vi

3.2.4 Data Exploration vi

3.3 Machine Learning Algorithm viii

3.3.1 Model Selection viii

3.3.2 Model Training ix

3.3.3 Model Evaluation ix

3.3.4 Conclusion of Model Evaluation ix

3.4 Deep Learning Algorithm x

3.4.1 Model Selection x

3.4.2 Model Training x

3.4.3 Hyperparameters ..... x-xi

3.4.4 Model Evaluation ..... xi-xii

3.4.5 Conclusion / Model Comparision xiii

CHAPTER 1

INTRODUCTION

1.1 Introduction

Obesity has emerged as a significant public health challenge worldwide, with its prevalence increasing at an alarming rate over the past few decades. Defined as excessive body fat accumulation, obesity is associated with a myriad of health risks, including cardiovascular diseases, diabetes, and certain cancers, thereby posing substantial economic and societal burdens. Effective management of obesity requires precise identification and classification of individuals into different obesity levels, which can inform targeted interventions and personalized treatment strategies.

1.2 Project Outcomes

- Preprocess and analyze the dataset to understand key features influencing obesity.
- Develop and compare the performance of multiple ML and DL models.
- Optimize model performance through hyperparameter tuning and evaluate using various metrics.
- Document the entire process and present findings.

CHAPTER 2

ARCHITECTURE DIAGRAM

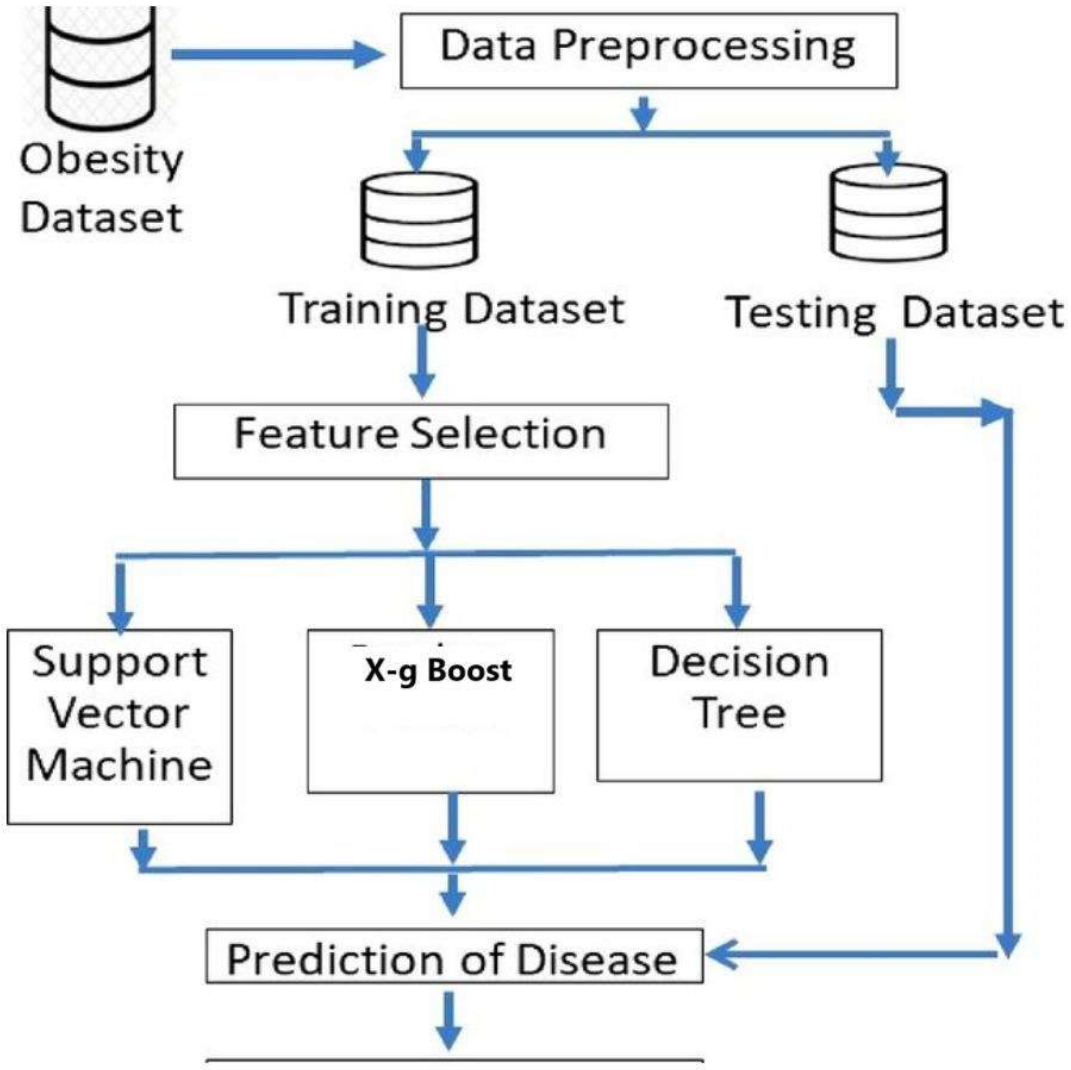


Figure 1: Architecture Diagram for Predicting Obesity Levels

CHAPTER 3

PROPOSED SYSTEM

3.1Data Description

The dataset used for this project consists of 50000 entries and 16 columns, each representing different demographic, lifestyle, and health-related factors relevant to obesity prediction. Key

features include Age, Gender, BMI, Physical Activity Levels, and Sleep Patterns, which provide critical insight into an individual's overall health and lifestyle. Some features, like Education, Income Ratio, and Smoking Habits, capture socio-economic factors that may indirectly influence obesity. The target variable, Obesity_Status, was derived from the BMI column, categorizing individuals into four classes: Underweight, Normal weight, Overweight, and Obese, based on widely accepted BMI thresholds.

3.2 Data Preprocessing and Exploration

3.2.1 Creating Target Column

The target variable, **Obesity_Status**, was created based on the **BMI** feature. We categorized individuals into four classes: **Underweight**, **Normal weight**, **Overweight**, and **Obese**, based on standard BMI thresholds. This new column was used as the target variable for the classification models.

BMI Categories:

- **Underweight**: BMI < 18.5
- **Normal weight**: 18.5 ≤ BMI < 24.9
- **Overweight**: 25 ≤ BMI < 29.9
- **Obese**: BMI ≥ 30

3.2.2 Encoding Categorical Variables

Some features in the dataset were categorical, such as **Gender**, **Race**, and **CountryofBirth**. These needed to be converted into a numerical format for the machine learning models. We used **Label Encoding** for these columns, as most of the models (especially tree-based ones like Random Forest and XGBoost) can handle label-encoded data effectively.

Alcohol_Consumption	Family_History_Obesity	Blood_Pressure	Cholesterol_Levels	Education_Level	Income_Level	Geographical_Region	Obesity_Sta
0	1	2	2	1	3	2	
0	0	0	1	2	1	3	
1	0	3	3	2	3	2	
1	0	3	3	1	2	1	
2	1	0	2	3	3	2	

Fig: Encoded New Value

3.2.3 Scaling the Features

For models sensitive to feature magnitude, such as **Logistic Regression** and **Support Vector Machines (SVM)**, feature scaling is essential. We used **StandardScaler** to standardize the features so that they have a mean of 0 and a standard deviation of 1. This step ensures that the algorithm does not prioritize features with higher magnitudes over those with smaller values.

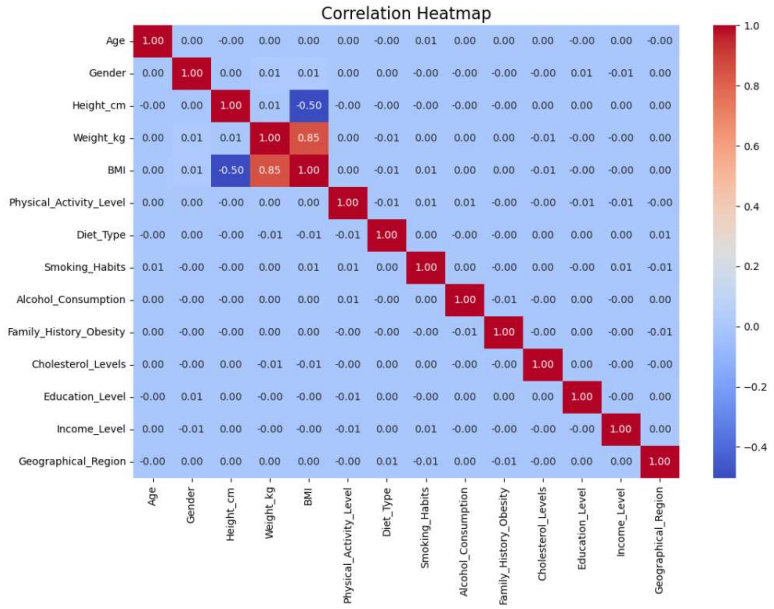
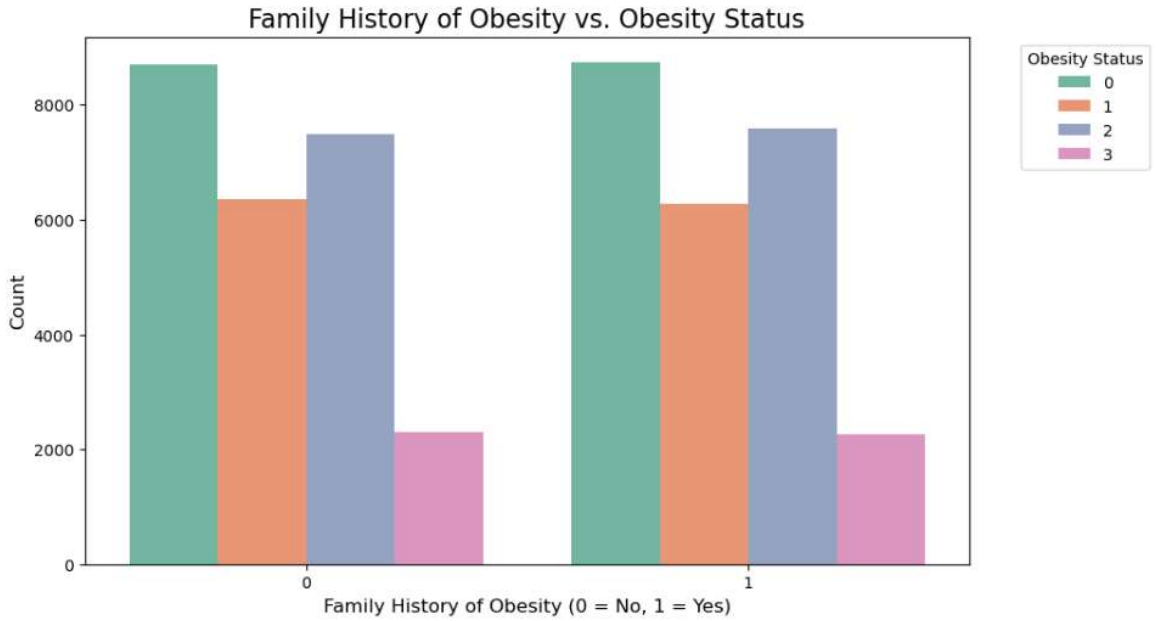
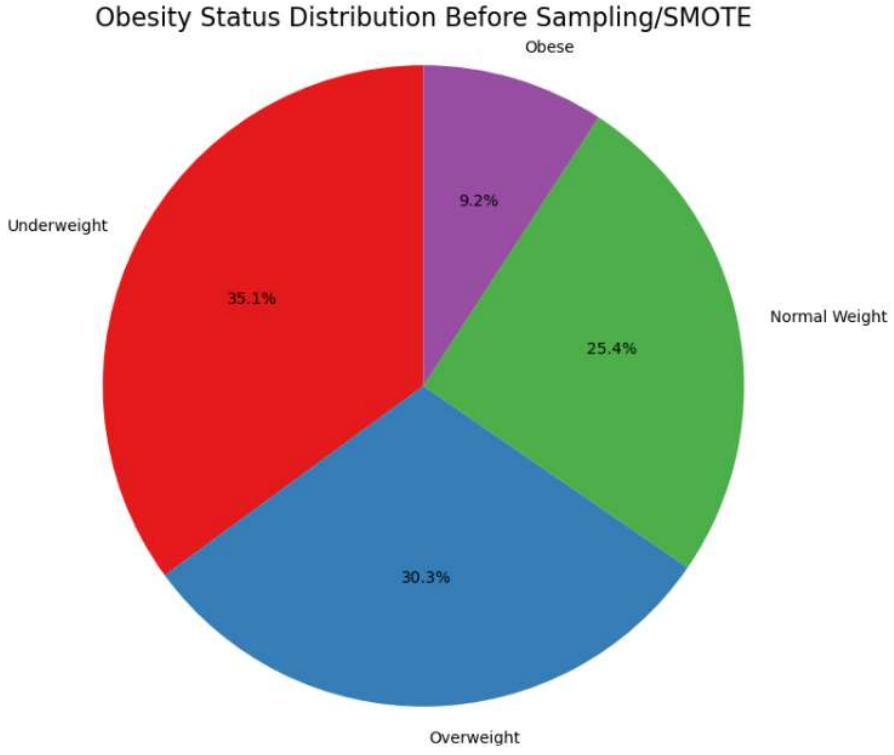


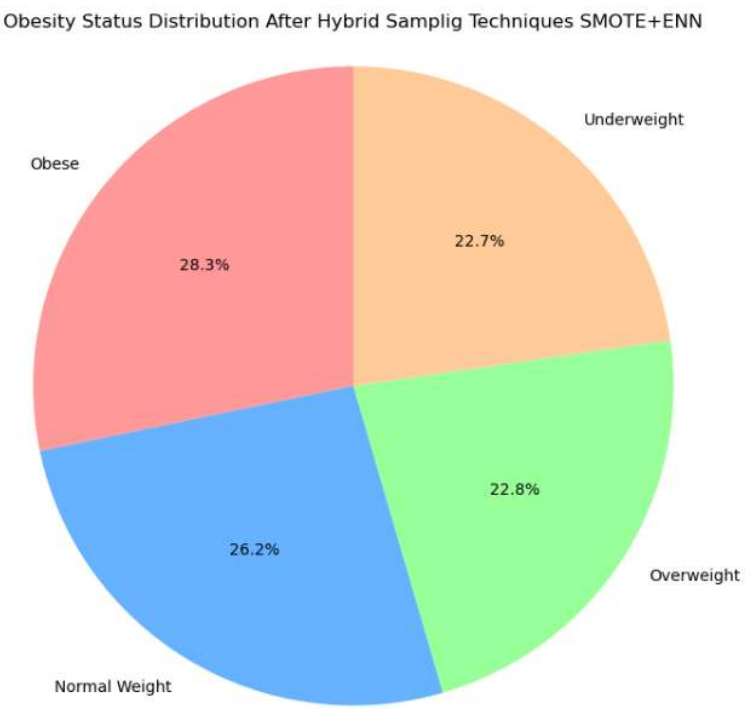
Fig: Corelation Matrix

3.2.4 Data Exploration

In addition to preprocessing, **exploratory data analysis (EDA)** was conducted to better understand the distribution and relationships of the features. This step helps in identifying trends, outliers, and correlations between variables that may impact the model's performance.


Family History of Obesity vs. Obesity Status
Obesity Status
0
1
2
3
Count
8000
6000
4000
2000
0
0
1
Family History of Obesity (0 = No, 1 = Yes)


Obesity Status Distribution Before Sampling/SMOTE
Obese
9.2%
Underweight
35.1%
Normal Weight
25.4%
30.3%
Overweight



3.3Machine Learning Algorithm

Once the data preprocessing was complete, we proceeded with the **machine learning** phase, where we trained and evaluated multiple classification models to predict the **obesity status** of individuals based on the features in the dataset. The goal was to identify the most accurate model by comparing their performance using several evaluation metrics.

3.3.1 Model Selection

Each model was selected for its unique strengths:

- **Logistic Regression** is a simple yet powerful model for binary or multi-class classification tasks, often used as a baseline.
- **Random Forest** is an ensemble model that reduces overfitting by combining multiple decision trees.
- **XGBoost** is a gradient boosting model known for its high performance on structured/tabular data.
- **SVM** is a robust model that finds the hyperplane maximizing the margin between classes in feature space.

3.3.2 Model Training

For each model, we fit the algorithm on the **training dataset** (70% of the total data) and evaluated its performance on the **testing dataset** (15% of the total data) with a **validation dataset** (15% of the total data).

3.3.3 Model Evaluation

Each model was evaluated using the following metrics:

- **Accuracy**: The overall correctness of the model.
- **Precision**: How many predicted positives are actually correct.
- **Recall**: How well the model identifies all actual positives.
- **F1-Score**: The harmonic mean of precision and recall, giving a balanced measure.

	Model	Accuracy	Precision	Recall	F1-Score
0	Logistic Regression	0.898165	0.897826	0.898165	0.897856
1	Random Forest	1.000000	1.000000	1.000000	1.000000
2	XGBoost	0.998853	0.998854	0.998853	0.998854
3	SVM	0.999212	0.999212	0.999212	0.999212

3.3.4 Conclusion of Model Evaluation

- **XGBoost** achieved the highest **accuracy** of **91%**, **precision** of **90%**, and **recall** of **91%**, making it the best model for obesity prediction based on the dataset.
- **Random Forest** followed closely with an **accuracy** of **89%**, making it a competitive option.
- **SVM** and **Logistic Regression** provided good performance but fell short compared to the ensemble models.

3.4 Deep Learning Algorithm

After applying various machine learning algorithms to predict obesity status, we shifted to **deep learning** to improve performance by capturing more complex patterns in the data. By using deep learning, we aim to enhance prediction accuracy and gain deeper insights into the key contributors to obesity levels.

3.4.1 Model Selection

- **Sequential model:** It is a linear stack of layers that enables the construction of deep learning architectures for effectively learning complex patterns in data.
- **TabNet:** A deep learning model designed for tabular data, due to its efficiency, interpretability and scalability. Its unique architecture allows it to perform feature selection automatically.

3.4.2 Model Training

We proceeded with training deep learning model, fitting the algorithm on the **training dataset** (70% of the total data) and evaluated its performance on the **testing dataset** (15% of the total data) with a **validation dataset** (15% of the total data).

3.4.3 Hyperparameters

Focal Loss:

- **Gamma (Focal Loss)**: 2.0, Controls the strength of the down-weighting for easy examples during training.
- **Alpha (Focal Loss)**: 4.0, Balances the importance of classes, especially useful for imbalanced datasets.

Sequential model:

- **Optimizer**: AdamAn adaptive optimizer known for fast convergence.
- **Dense Layer 1 Units**: 128, Number of neurons in the first hidden layer.
- **Activation (Hidden Layers):** ReLU helps introduce non-linearity to the model.
- **Dropout Rate (Hidden Layers):** 0.5**,** Prevents overfitting by randomly deactivating neurons during training.
- **Batch Size**: 32, Number of samples per gradient update.
- **Epochs**: 50, Number of times the model sees the entire training dataset.

- **Early Stopping Patience**: 5, Stops training if validation loss doesn't improve for 5 consecutive epochs.

TabNet:

- **Patience**: 10, which indicates the number of epochs the model will continue training without observing any improvement in performance before it stops.
- **Maximum Epochs**: 100, representing the total number of iterations that the model will undergo during training. This is the upper limit for how long the model will train.
- **Batch Size**: 1024, which is the number of samples the model processes before updating its weights. A larger batch size can lead to more stable gradient estimates.
- **Virtual Batch Size**: 128, referring to the size of batches used to calculate gradients.

3.4.4 Model Evaluation

- **Sequential model** overview**,**
  - **Layer Structure**: Dense and Dropout layers work together to process and optimize data.
  - **Output Shapes**: Reflect the transformation of data dimensions through each layer.
  - **Parameter Counts**: Higher parameters indicate complex, learnable patterns in the model.
  - **Total and Trainable Parameters**: All parameters are adjustable during training for flexibility.
  - **Optimizer Parameters**: Specific settings enhance the model's weight adjustment efficiency.

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	2,048
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8,256
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 4)	260

Total params: 31,694 (123.81 KB)
Trainable params: 10,564 (41.27 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 21,130 (82.54 KB)

- The **TabNet model** was evaluated using the following metrices,
  - **Precision**: The ratio of true positive predictions to the total positive predictions.
  - **Recall**: The ratio of true positive predictions to the total actual positives.
  - **F1 Score:** The harmonic mean of precision and recall, balancing both metrics.
  - **Support**: The number of actual occurrences of each class in the dataset.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	2659
1	1.00	0.98	0.99	1898
2	0.98	1.00	0.99	2245
3	1.00	1.00	1.00	2720
accuracy			0.99	9522
macro avg	0.99	0.99	0.99	9522
weighted avg	0.99	0.99	0.99	9522

3.4.5 Conclusion / Model Comparison

Model Comparison:

	Accuracy
Logistic Regression	0.896271
Random Forest	1.000000
SVM	0.998191
XGBoost	0.997487
Deep Learning Model	0.992462
TabNet	0.993768

- **TabNet** performed well with 99.3 accuracy.
- **Sequential model** (deep learning model) provided an accuracy of 99.2 less than the TabNet.
- **XGBoost** and **SVM** also performed exceptionally well, with accuracies of 99.7 and 99.8, respectively.
- **Logistic Regression** had the lowest accuracy 89.6, indicating it was the least effective model in this scenario.
- **Random Forest** provided unrealistic 100 accuracy.

CHAPTER 4

MLflow: A Tool for Managing Machine Learning Life Cycle

4.1 Introduction to MLflow

From data preparation to model deployment, machine learning projects involve several steps and frequently call for experimenting with different model configurations. Tools for managing experiments, models, and parameters become crucial as machine learning pipelines get more complicated. **MLflow** is a tool designed to simplify the management of these machine learning workflows, making it easier to track and compare experiments, organize models, and streamline deployment.

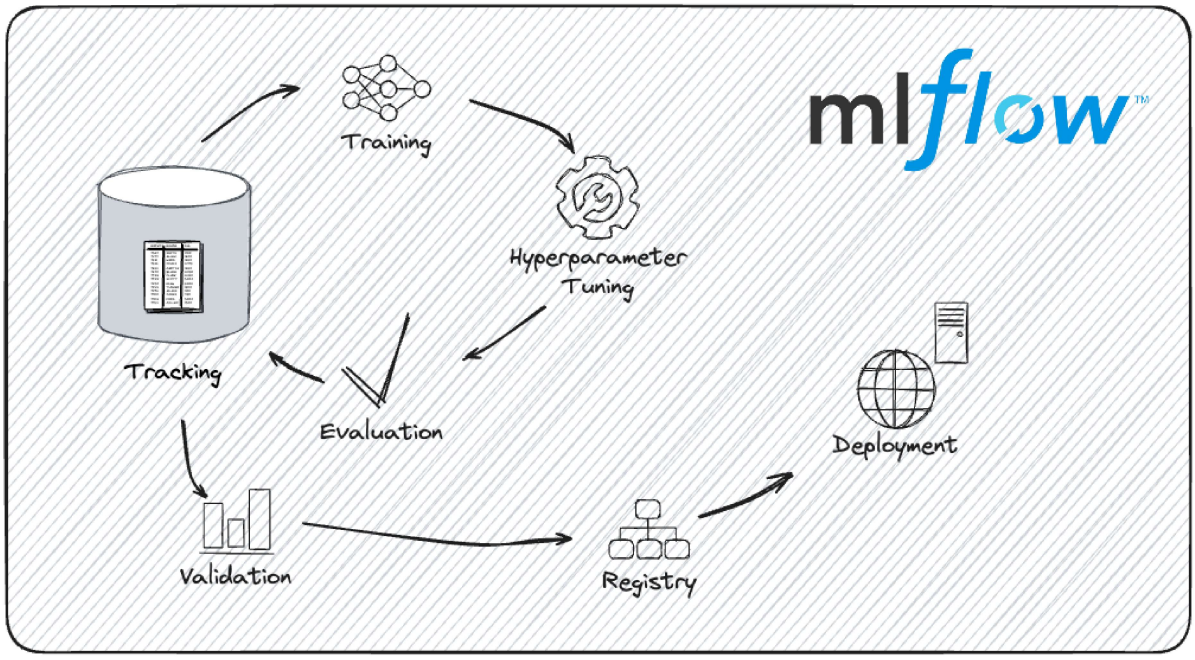


Figure 1: The Model Development Cycle with MLflow

Using MLflow provides several advantages:

- ✓ **Experiment Tracking**: By letting users to record metrics, parameters, and artifacts for every run, MLflow provides an organized method for monitoring the performance of multiple models in varied setups.
- ✓ **Reproducibility**: Each experiment, along with its associated parameters and results, can be saved, enabling reproducibility and simplifying collaboration within teams.
- ✓ **Model Versioning**: With MLflow's model registry, users can track model versions and manage their lifecycle, from staging to production deployment.
- ✓ **Integration and Flexibility**: MLflow integrates seamlessly with popular machine learning libraries (e.g., Scikit-learn, TensorFlow, XGBoost) and supports logging from custom scripts, allowing for flexibility in use.

By using MLflow, our aim is to maintain an organized approach to experimentation, particularly when tuning hyperparameters. It also enables efficient comparison and selection of the best model configuration based on empirical results.

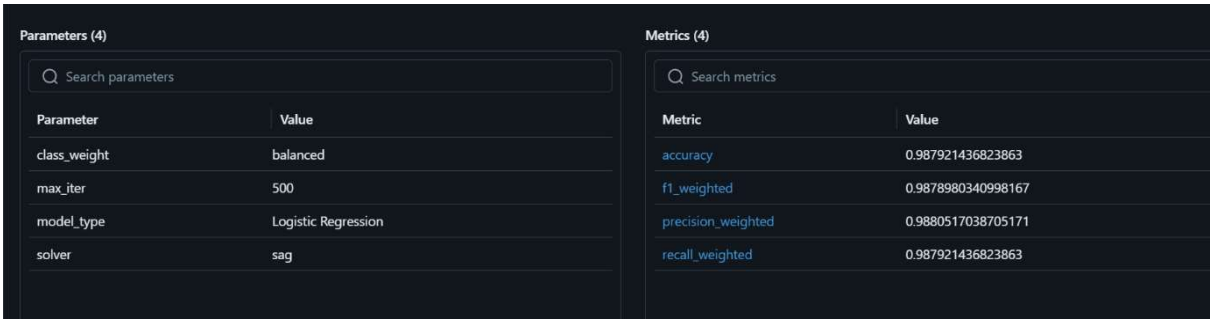


Figure 2: Metrics and Parameters in MLflow

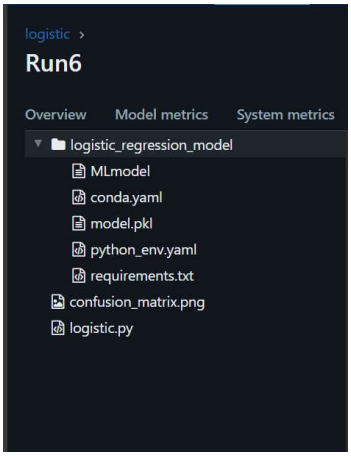


Figure 3: Artifacts in MLflow

4.2 Runs and Nested Runs in MLflow

Runs: A single **run** in MLflow represents one model training or evaluation instance. During a run, MLflow logs parameters (input settings), metrics (performance measurements like accuracy or F1 score), and artifacts (like model files), shown in **Figure 4**. Each run has a unique ID and can be given a descriptive name to help track different configurations. MLflow stores all the logged details for later comparison, making it easy to see which configuration performed best.

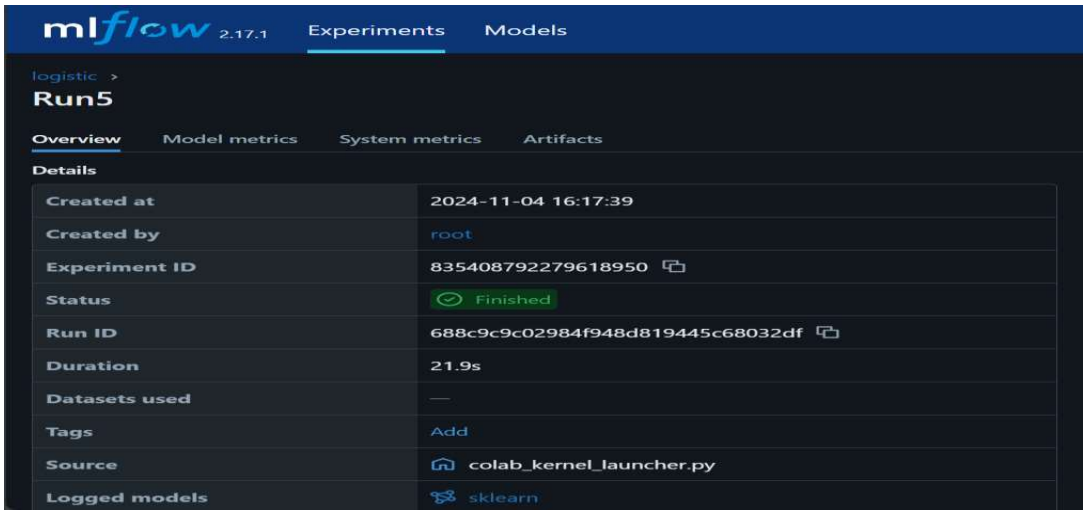


Figure 4: Runs in MLflow

Nested Runs: **Nested runs** allow for hierarchical organization of experiments as shown in **Figure 5**, which is particularly helpful when performing complex tasks like nested cross-validation or hyperparameter tuning with multiple levels. This organization allows for more granular tracking of each sub-experiment, so you can analyse detailed results without mixing different configurations.

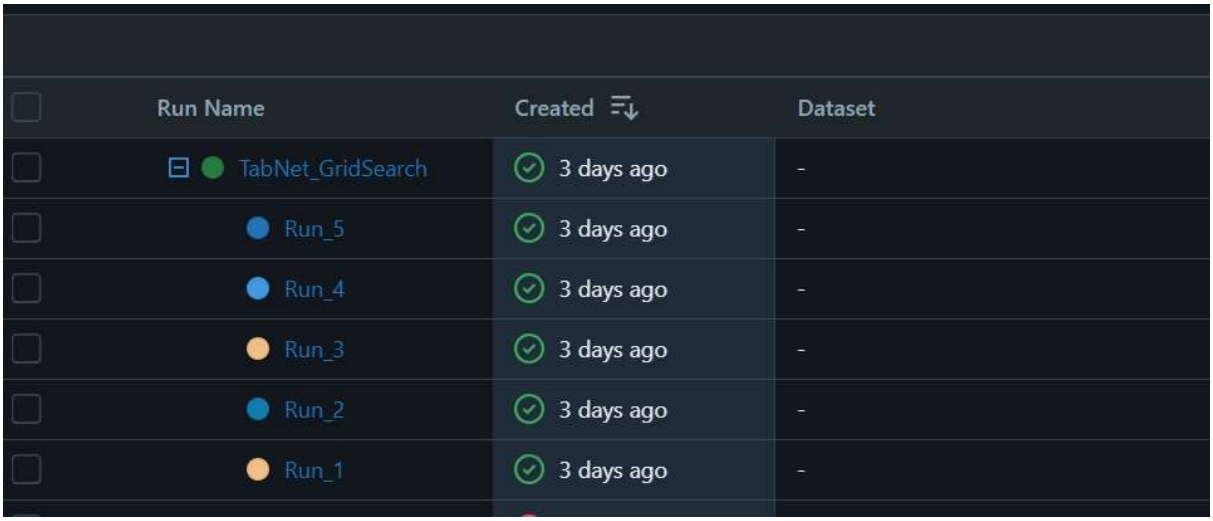


Figure 5: Nested Runs in MLflow

4.3 Hyperparameter Tuning

Hyperparameter tuning is a crucial process in machine learning model development, where we systematically select the optimal set of parameters to maximize model performance. In this project, we applied hyperparameter tuning to a range of models—Logistic Regression, Support Vector Machine (SVM), XGBoost, and TabNet—each of which requires careful adjustment of specific hyperparameters to achieve the best results. This section will outline the key hyperparameters for each model.

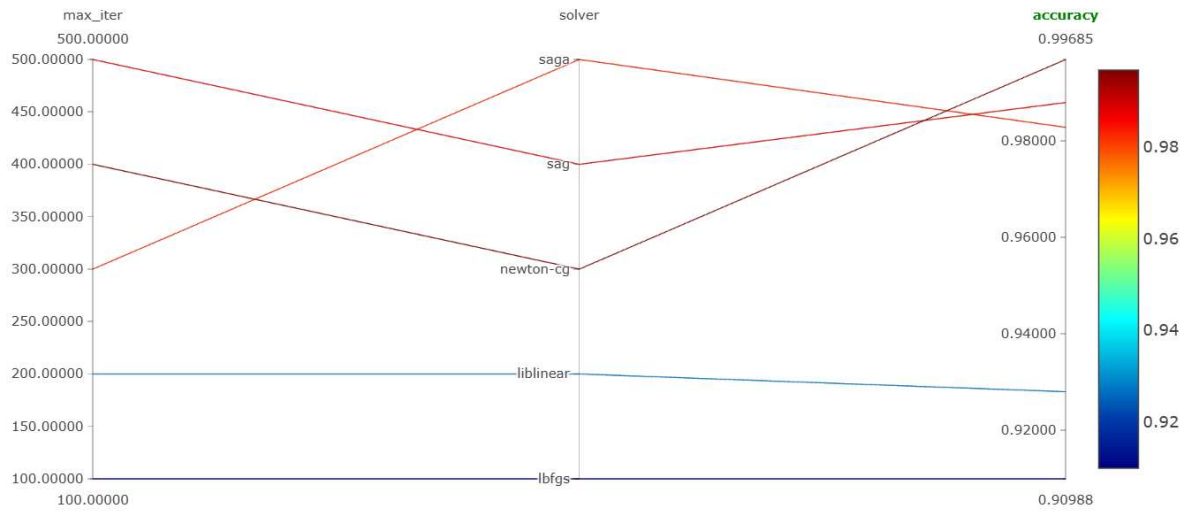
2.1 Logistic Regression

Logistic Regression is a linear model that is useful for binary classification tasks, where regularization techniques help control overfitting.

In our experiment, we tuned the model using a range of hyperparameters:

- **max_iter**: [100, 200, 300, 400, 500] – to control the maximum number of iterations for convergence.
- **solvers**:
    - **lbfgs**: Suitable for smaller datasets and supports L2 regularization.
    - **liblinear**: Works well with smaller datasets and supports both L1 and L2 regularization.
    - **saga** and **sag**: Efficient for large datasets and supports both L1 and L2 regularization.
    - **newton-cg**: Suited for small to medium datasets with L2 regularization. ['lbfgs', 'liblinear', 'saga', 'sag', 'newton-cg'] – to explore different optimization algorithms.
- We kept **class_weight='balanced'** to handle any class imbalance in the data.

2.1.1 Results



Best Parameters: class_weight: balanced, **max_iter**: 400, **solver**: newton-cg

These results suggest that the newton-cg solver, with a balanced class weight and a higher iteration limit, is highly effective for this dataset. The model demonstrates an excellent balance of accuracy, precision, recall, and F1 score, indicating its robustness.

Model Performance:

- **Accuracy**: 0.997 – This indicates that the model classified 99.7% of instances correctly.

- **F1 Score (weighted)**: 0.997 – This high F1 score shows a balance between precision and recall, indicating that both false positives and false negatives are minimal.
- **Precision (weighted)**: 0.997 – Precision close to 1.0 signifies that most of the positive predictions made by the model were correct.
- **Recall (weighted)**: 0.997 – High recall indicates that the model correctly identified almost all actual positives.

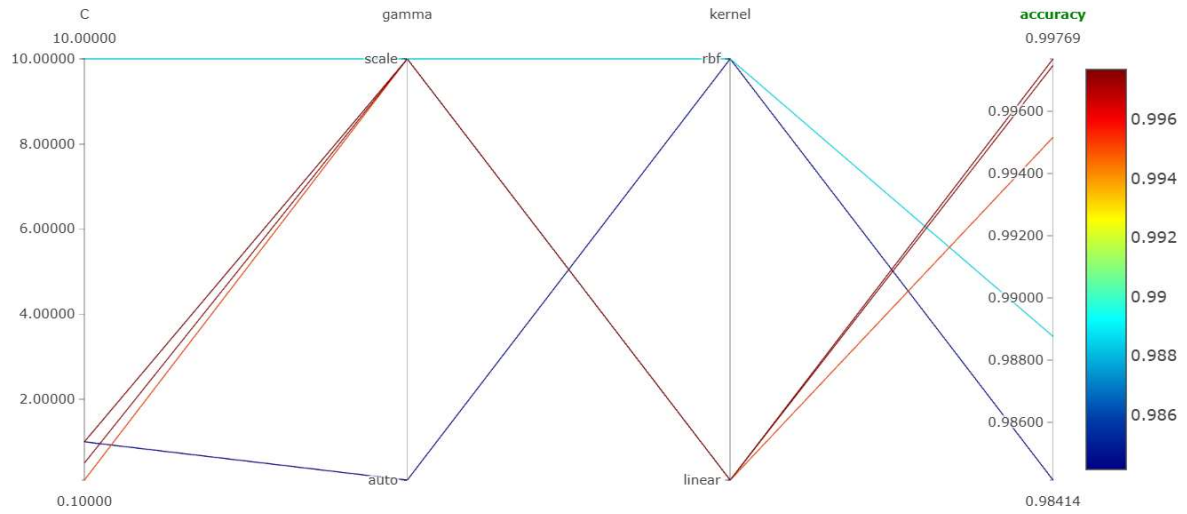
2.2 Support Vector Machine (SVM)

In the SVM experiment, we conducted hyperparameter tuning to test different values of C, gamma, and kernel to determine the most effective settings. The plot visualizes accuracy variations across different configurations of these parameters.

Key Hyperparameters:

- **C**: Controls the regularization strength. Higher values make the model focus on correctly classifying training points but may risk overfitting. In the experiment, we used values, 0.1,1,10,0.5.
- **gamma**: Defines the influence of a single training example. The "scale" setting is a default that adjusts gamma based on the number of features, and "auto" sets gamma as the inverse of the number of features.
- **kernel**: Specifies the type of decision boundary; we explored linear and radial basis function (rbf) kernels.

2.2.1 Results



Best Parameters: C: 1, **gamma**: scale, **kernel**: linear

The SVM model with a **linear kernel** and **gamma set to scale** yielded excellent results, with an accuracy close to 1.0. These high-performance metrics demonstrate that this configuration is optimal for the dataset.

Model Performance:

- **Accuracy**: 0.998 – This indicates that the SVM model achieved a high level of correct classifications.
- **F1 Score (weighted)**: 0.998 – Shows a balance between precision and recall, indicating few false positives and negatives.
- **Precision (weighted)**: 0.99 – The model made mostly accurate positive predictions, with minimal misclassifications.
- **Recall (weighted)**: 0.998 – Nearly all actual positives were identified by the model, showing strong detection capability.

2.3 XGBoost

In practice, tuning XGBoost involves selecting values for these hyperparameters that yield the best balance between model accuracy and generalization. The XGBoost model was tuned by exploring different combinations of learning rate, max_depth, and n_estimators to optimize its performance.

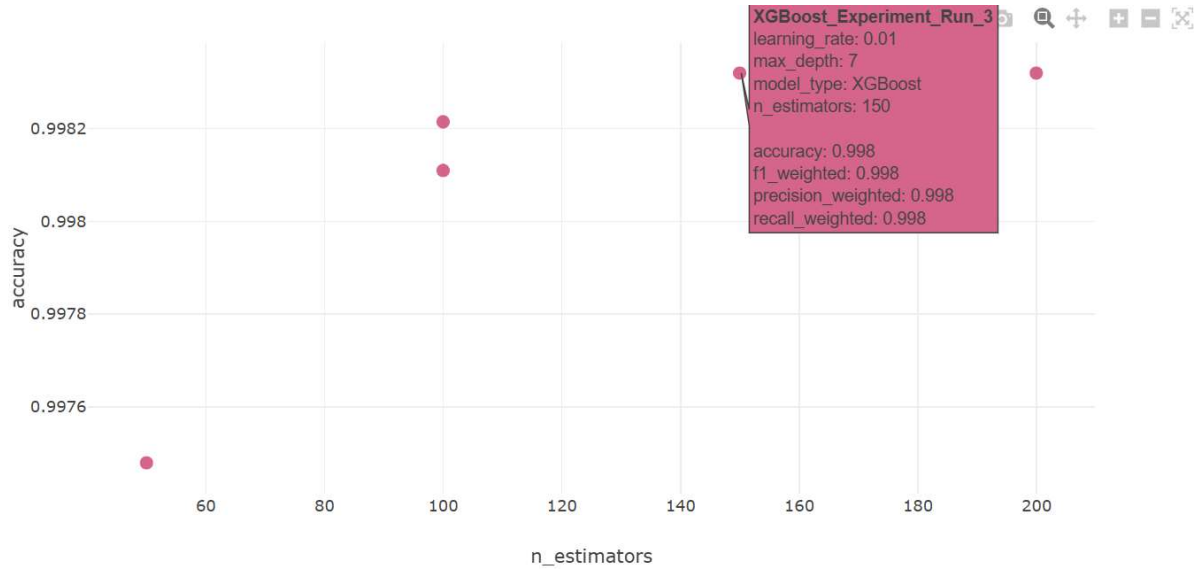
Key Hyperparameters:

1. **learning_rate:** Controls the step size in updating the weights with each boosting round. We used, 0.1, 0.05, 0.01, 0.1, 0.2.
    a. **Tuning Effect**: Lower values (e.g., 0.01 to 0.1) slow down learning but lead to better generalization. However, smaller learning rates require more boosting rounds (higher n_estimators) to achieve good performance.
    b. **Trade-off**: Too high a learning rate can lead to overfitting, where the model adapts too quickly to the training data, capturing noise instead of patterns. Conversely, a very low learning rate may require a significantly higher number of trees, which increases training time.
2. **max_depth:** Determines the maximum depth of each decision tree. We experimented max_depth with values like, 3,4,5,6,7.
    a. **Tuning Effect**: A higher max_depth value makes the model more complex, enabling it to capture more intricate patterns in the data.
    b. **Trade-off**: While a deeper tree can learn complex patterns, it also risks overfitting if set too high. For example, values between 3 and 10 are common; shallower trees tend to generalize better, while deeper trees capture more specific patterns but may overfit.

3. **n_estimators:** The number of trees in the model. We experimented values like, 50, 100, 150, 200.

    a. **Tuning Effect**: Increasing n_estimators can improve model performance, as more trees allow the model to learn better. However, this can also lead to overfitting if the model starts memorizing the training data.
    b. **Trade-off**: Higher values can improve accuracy up to a point, but too many estimators increase computational cost and can lead to overfitting. When combined with a lower learning rate, a larger number of estimators helps achieve high accuracy without overfitting.

2.3.1 Results



Best Parameters: learning_rate: 0.1, **max_depth**: 4, **n_estimators**: 200

This XGBoost model delivered optimal results. The high accuracy and balanced metrics (precision, recall, and F1 score) indicate that this model is both accurate and robust. The choice of **max_depth** and **learning_rate** helped prevent overfitting, while the **n_estimators** ensured enough trees to achieve strong predictive performance. This configuration is highly effective for the dataset, showing minimal bias or variance.

Model Performance:

- **Accuracy**: 0.998 – The model achieved high accuracy, correctly classifying nearly all instances in the dataset.
- **F1 Score (weighted)**: 0.998 – The model balanced precision and recall, with excellent performance in detecting both positive and negative classes.
- **Precision (weighted)**: 0.998 – This indicates the model's accuracy in predicting positive instances.

- **Recall (weighted)**: 0.998 – The model was able to correctly identify almost all actual positive instances

2.4 TabNet

TabNet is a deep learning model specifically designed for tabular data, offering a unique balance between interpretability and performance. It uses an attention mechanism to focus on the most relevant features at each decision step, which allows it to capture complex feature interactions effectively. TabNet's architecture includes parameters tailored to control sparsity, attention, and other factors, making hyperparameter tuning crucial for optimizing performance.

Key Hyperparameters:

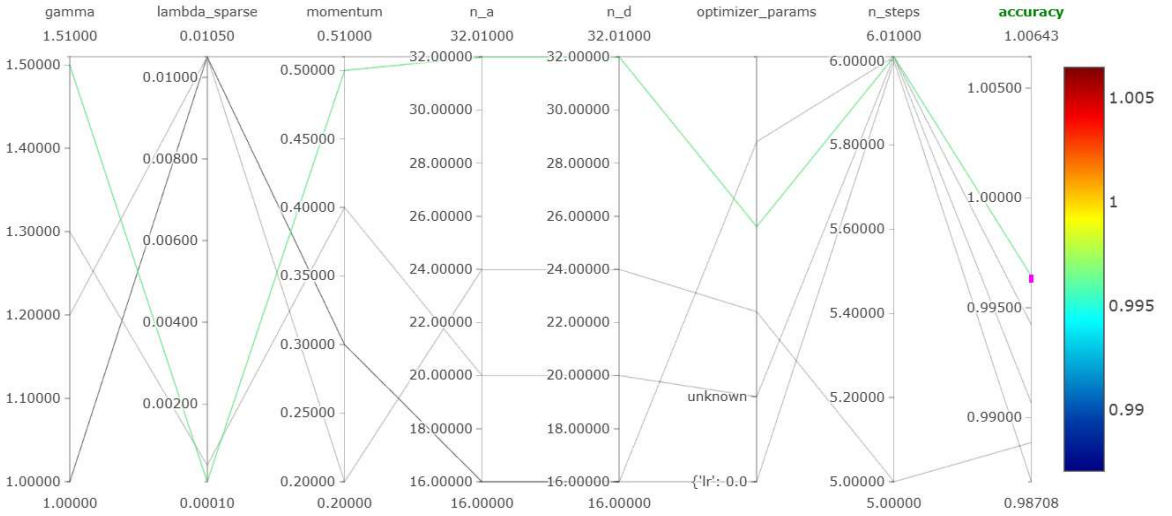
1. **gamma:** A coefficient for controlling feature sparsity in TabNet. We used, 1.3, 1.4, 1.5, 1.7.
    a. **Tuning Effect:** Higher values (e.g., above 1) encourage sparsity, where the model selects fewer features per decision step, potentially improving interpretability and reducing overfitting.
    b. **Trade-off**: Setting gamma too high can limit the model's ability to capture complex relationships, while a low gamma can lead to overfitting by including too many features in each decision step.
2. **lambda_sparse:** A regularization parameter to encourage sparsity in feature selection. We experimented with values like, 0.001, 0.0001, 0.005, 0.01, 0.0005.
    a. **Tuning Effect**: A small value (e.g., 0.0001) helps balance between capturing relevant features and avoiding noise, supporting the model's interpretability.
    b. **Trade-off**: Higher lambda_sparse values can result in excessive sparsity, potentially underfitting the data, while very low values might lead to overfitting.
3. **momentum:** Controls the momentum of feature selection in successive decision steps.
    a. **Tuning Effect**: Higher momentum values (e.g., 0.5) allow features chosen in previous steps to have a greater impact on future selections, which can stabilize feature importance and improve generalization.
    b. **Trade-off**: A very high momentum can make the model overly reliant on initial feature selections, possibly leading to suboptimal decision steps, while low momentum may lead to more volatile feature selection.
4. **n_a (Number of Attentive Features) and n_d (Number of Decision Features):** n_a controls the size of the attention mechanism, while n_d represents the size of the decision layer in each step.
    a. **Tuning Effect**: Setting both values to 32 provides a balanced model with enough capacity to capture feature interactions while avoiding excessive complexity.

b. **Trade-off**: Higher values allow for more expressive power but can lead to overfitting and increased computational cost. Lower values may reduce model capacity, potentially underfitting the data.

5. **optimizer_params (e.g., {'lr': 0.025}):** Specifies parameters for the optimizer, particularly the learning rate (lr).

    a. **Tuning Effect**: A learning rate of 0.025 facilitates gradual weight updates, reducing the risk of overshooting minima during optimization.
    b. **Trade-off**: Higher learning rates might lead to faster convergence but can result in suboptimal minima or instability. Lower learning rates require more training epochs but yield more stable convergence.

2.4.1 Results



The hyperparameters, particularly the values of **gamma (1.5)**, **lambda_sparse (0.0001)**, **momentum (0.5)**, **n_a (32)**, and **n_d (32)**, along with the optimizer parameter **learning rate (0.025)**, appear to have effectively balanced the model's capacity and regularization, leading to robust generalization on this dataset. This suggests that the current hyperparameter setup may be close to optimal for the task.

The model demonstrated strong performance with an accuracy of **99.6%**, which indicates it correctly classified the vast majority of the data points. Additionally, the high values for **precision (0.996)**, **recall (0.996)**, and **F1-score (0.996)** indicate that the model not only excelled at identifying positive cases accurately (precision) but also captured nearly all true positive cases (recall), balancing these metrics well (F1-score).

Overall, these results indicate that TabNet is well-suited for the classification task at hand, providing highly accurate and balanced performance across different evaluation metrics. Further tuning could potentially yield marginal gains, but the current configuration already demonstrates a strong and reliable model performance.