UNIVERSITI TEKNOLOGI MALAYSIA

FACULTY OF COMPUTING

SEMESTER 1, SESSION 2025/2026

PROJECT PROGRESS 5

OBESITY LEVEL CLASSIFICATION

SECB3203 : PROGRAMMING FOR BIOINFORMATICS

SECTION 02

GROUP MEMBER:

1. MUHAMMAD FARIHIN BIN SALEH	A25CS0102
2. MUHAMMAD MIRZA HASIF BIN MOHD FAHMI	A25CS0108
3. MUHAMMAD NAWFAL BIN MOHD SHAIFUDDIN	A25CS0109

LECTURER NAME : DR. SEAH CHOON SEN

GROUP : GROUP 07

TABLE OF CONTENTS

1.0 MODEL EVALUATION

In this phase, we implemented a rigorous evaluation strategy to identify the optimal machine learning model for classifying obesity levels. We tested four distinct algorithms Random Forest, XGBoost, Support Vector Machine (SVM), and Logistic Regression to determine which approach best handles the complex, non-linear patterns inherent in the dataset.

To ensure a holistic evaluation, we utilized two primary metrics:

- **Accuracy:** Used as a general performance baseline to measure the overall percentage of correct predictions.
- **F1-Score:** Used to ensure we are not ignoring performance on minority classes. Since medical datasets can sometimes be imbalanced, the F1-score (the harmonic mean of Precision and Recall) provides a more robust measure of the model's ability to minimize false positives and false negatives.

1.1 Selected Models

We compared four distinct algorithms to handle the multi-class classification problem:

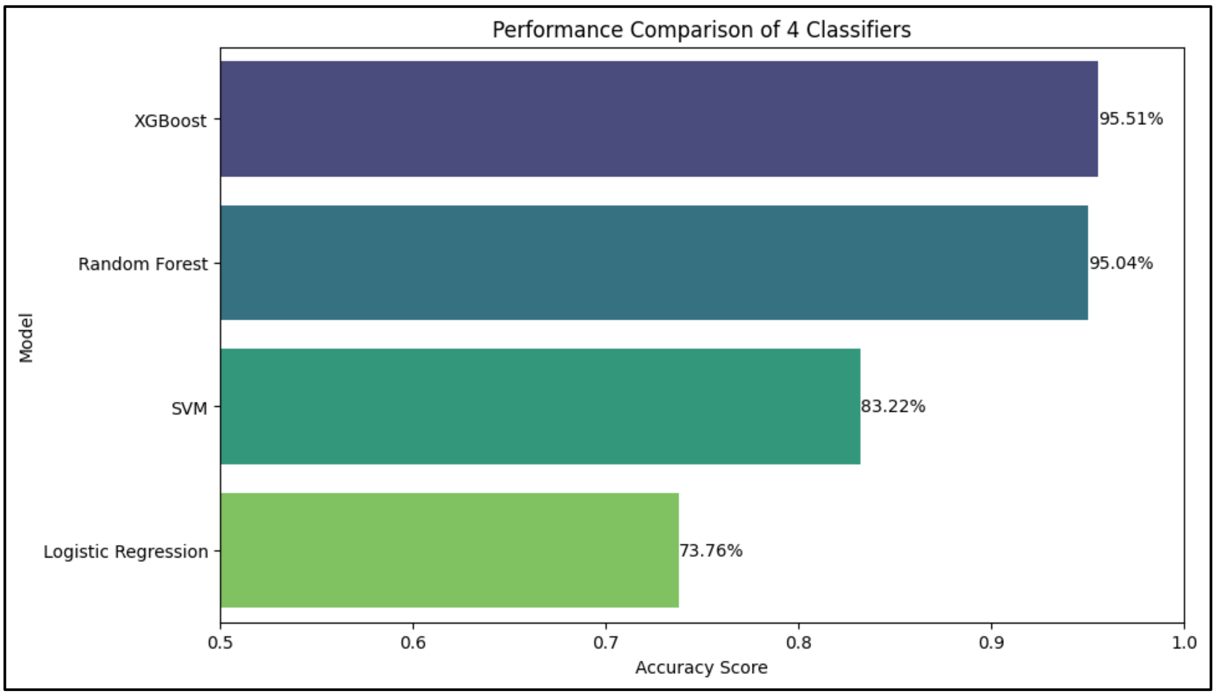
1. Random Forest: Machine learning method that creates many decision trees during training and outputs the mode (classification) or mean (regression) of the individual trees.
2. XGBoost: A gradient boosting framework known for high performance on tabular data.
3. Support Vector Machine (SVM): Effective in high-dimensional spaces.
4. Logistic Regressions: Used as a linear baseline for comparison.

1.2 Model Comparison

We trained all four models using default parameters.

```python
# MODEL EVALUATION & COMPARISON
results = []

# Evaluate Loop
for name, model in models.items():
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    results.append({'Model': name, 'Accuracy': acc})
    print(f"{name} Accuracy: {acc:.2%}")

# Convert results to Table
results_df = pd.DataFrame(results).sort_values(by='Accuracy', ascending=False)

# Bar Chart Comparison
plt.figure(figsize=(10, 6))
# Using 'hue' to fix the warning
sns.barplot(x='Accuracy', y='Model', hue='Model', data=results_df, palette='viridis', legend=False)
plt.xlim(0.5, 1.0) # Zoom in on the 50%-100% range to see differences
plt.title('Performance Comparison of 4 Classifiers')
plt.xlabel('Accuracy Score')
for index, value in enumerate(results_df['Accuracy']):
    plt.text(value, index, f'{value:.2%}', va='center')
plt.show()
```

This code evaluates multiple trained classifiers by predicting labels on the test dataset and calculating their accuracy scores. The accuracy results are stored in a DataFrame and sorted for easy comparison. A bar chart is then generated to visually compare the accuracy of each model, with values displayed directly on the chart for clarity.

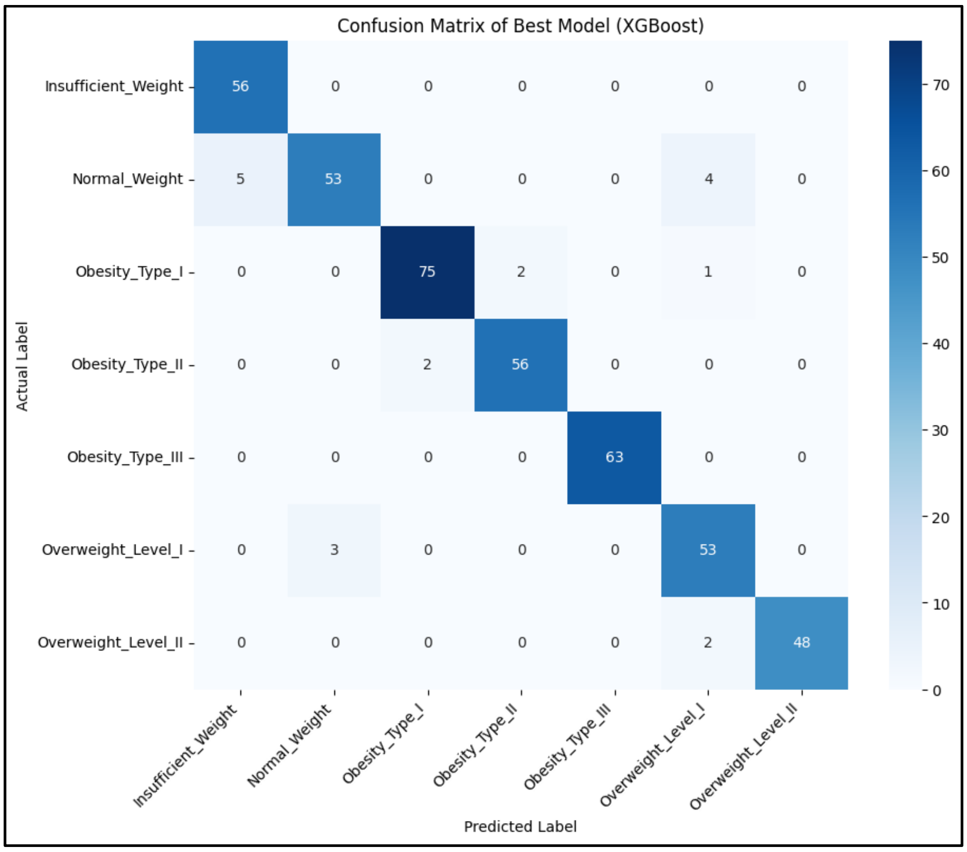


The **XGBoost** outperformed the others. Tree-based models like XGBoost and Random Forest are naturally better suited for this dataset than linear models like Logistic Regression. Our data contains a mix of categorical variables like Transportation Mode, Family History, and numerical ones. Tree-based algorithms can handle these mixed data types and non-linear interactions without assuming a straight-line decision boundary. Logistic Regression performed significantly worse (**73.76%**), confirming that the boundaries between obesity levels are too complex for simple linear classification.

1.3 Confusion Matrix

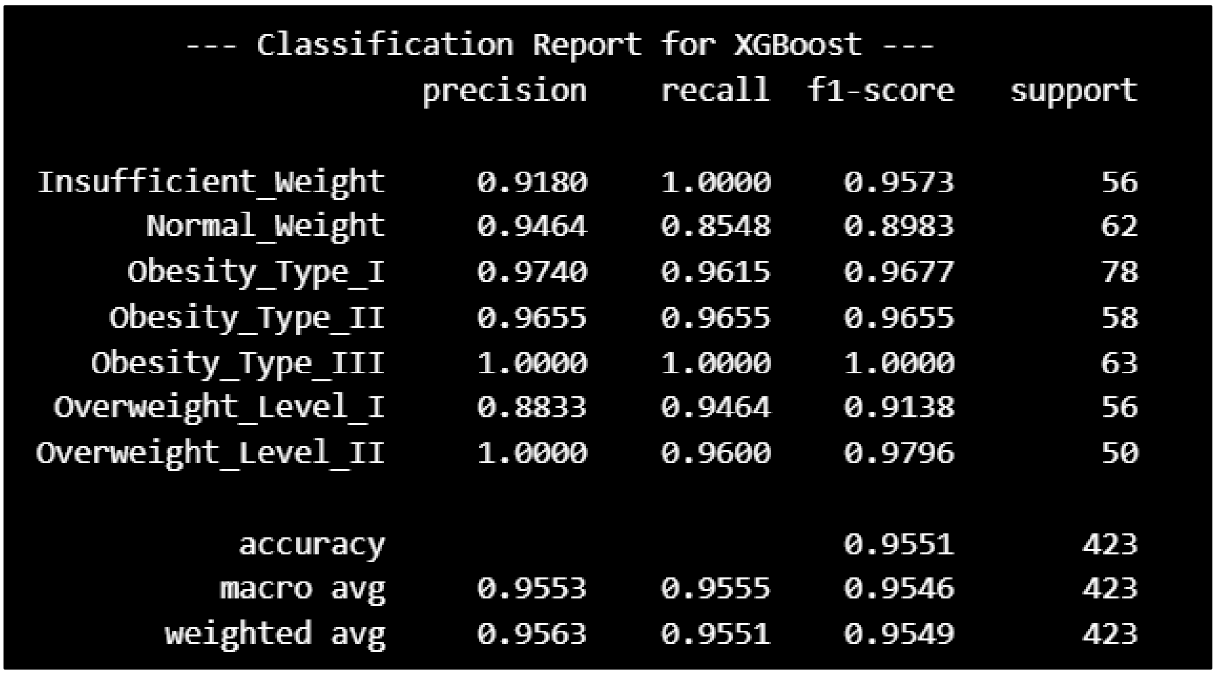
To understand where our best model fails, we generated a Confusion Matrix.

```python
# CONFUSION MATRIX OF BEST MODEL
best_model = models[best_model_name]
y_pred_best = best_model.predict(X_test)

plt.figure(figsize=(8, 6))
sns.heatmap(confusion_matrix(y_test, y_pred_best), annot=True, fmt='d', cmap='Blues')
plt.title(f'Confusion Matrix of Best Model ({best_model_name})')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```



The strong diagonal line indicates that for the vast majority of cases, the predicted label matched the actual label. As expected, the model struggled most with distinguishing between **Overweight Level I** and **Overweight Level II**. These are adjacent classes, meaning the physiological difference of BMI thresholds between them is small. Conversely, the model showed near-perfect separation between distinct classes, such as **Normal Weight** and **Obesity Type III**, proving it can easily identify high-contrast cases.

1.4 Classification Report

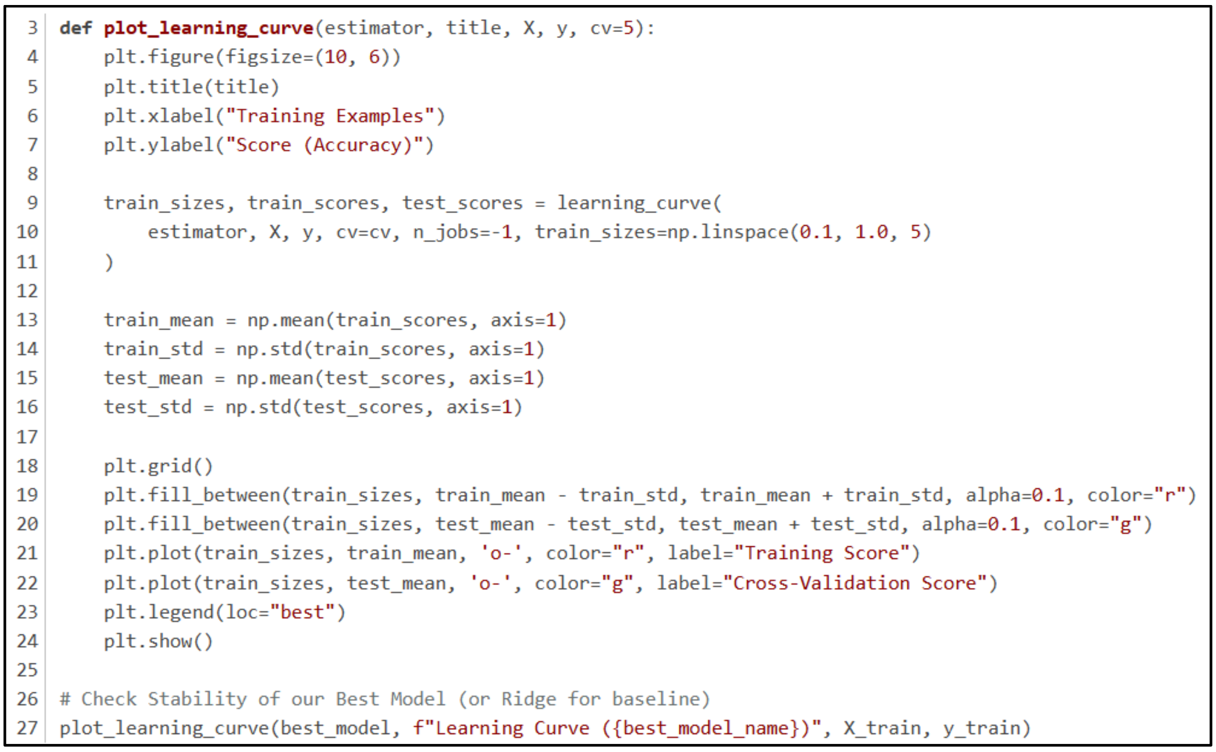


```
        --- Classification Report for XGBoost ---
                      precision    recall  f1-score   support

 Insufficient_Weight     0.9180    1.0000    0.9573        56
       Normal_Weight     0.9464    0.8548    0.8983        62
      Obesity_Type_I     0.9740    0.9615    0.9677        78
     Obesity_Type_II     0.9655    0.9655    0.9655        58
    Obesity_Type_III     1.0000    1.0000    1.0000        63
  Overweight_Level_I     0.8833    0.9464    0.9138        56
 Overweight_Level_II     1.0000    0.9600    0.9796        50

            accuracy                         0.9551       423
           macro avg     0.9553    0.9555    0.9546       423
        weighted avg     0.9563    0.9551    0.9549       423
```

The classification report validates XGBoost as a highly effective model, achieving an overall accuracy of **96%** on the test set. A granular analysis reveals that the model performs perfectly on the most critical category, **Obesity_Type_III**, achieving a Precision, Recall, and F1-score of 1.00. This indicates the model makes zero errors in identifying the most severe cases of obesity. While performance is robust across the board, minor fluctuations are observed in the **Normal_Weight** category, which has a Recall of **0.8548**, suggesting that approximately 15% of actual normal-weight individuals were misclassified. Similarly, **Overweight_Level_I** shows a slightly lower Precision of **0.88**, implying a small tendency to over-predict this specific class. However, with a weighted average F1-score of 0.95, the model demonstrates high reliability and stability across all seven disparate classes.

2.0 OVER-FITTING & UNDER-FITTING

To strictly evaluate whether our model creates generalizable rules or merely "memorizes" the training data, we implemented a **Learning Curve** analysis using the learning_curve function from Scikit-learn.

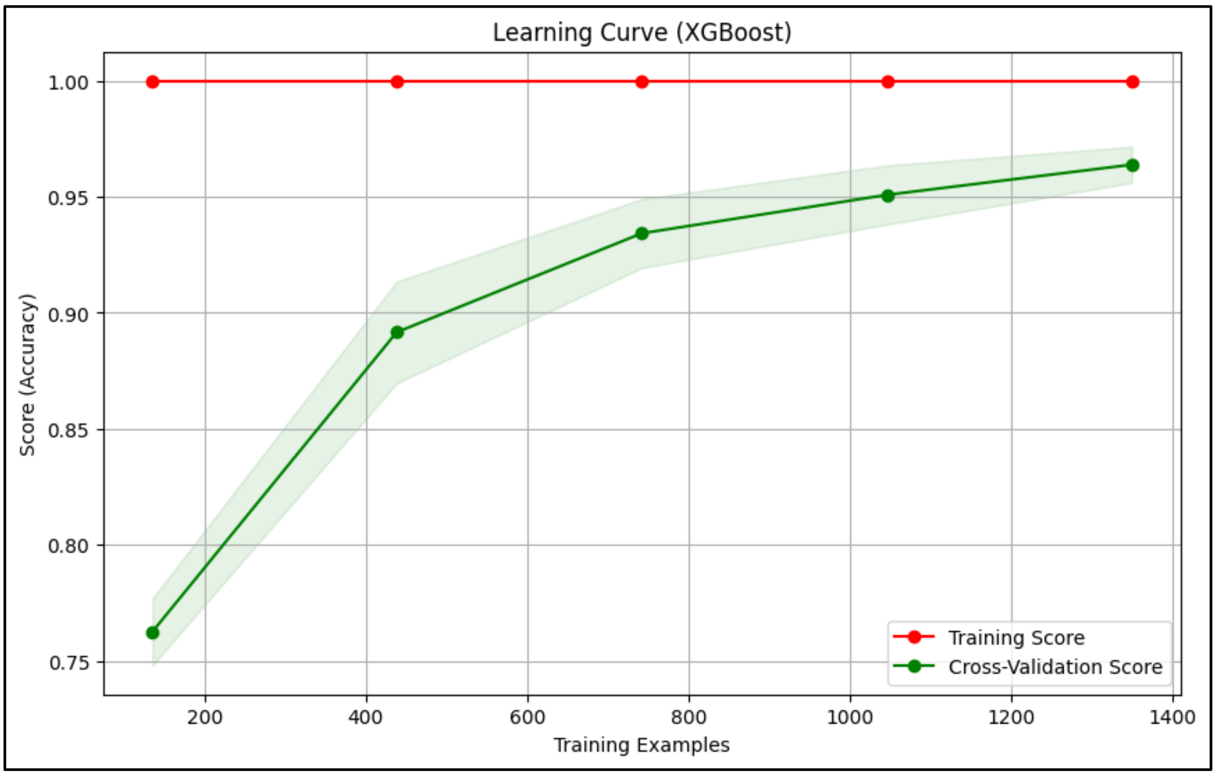


```python
def plot_learning_curve(estimator, title, X, y, cv=5):
    plt.figure(figsize=(10, 6))
    plt.title(title)
    plt.xlabel("Training Examples")
    plt.ylabel("Score (Accuracy)")

    train_sizes, train_scores, test_scores = learning_curve(
        estimator, X, y, cv=cv, n_jobs=-1, train_sizes=np.linspace(0.1, 1.0, 5)
    )

    train_mean = np.mean(train_scores, axis=1)
    train_std = np.std(train_scores, axis=1)
    test_mean = np.mean(test_scores, axis=1)
    test_std = np.std(test_scores, axis=1)

    plt.grid()
    plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.1, color="r")
    plt.fill_between(train_sizes, test_mean - test_std, test_mean + test_std, alpha=0.1, color="g")
    plt.plot(train_sizes, train_mean, 'o-', color="r", label="Training Score")
    plt.plot(train_sizes, test_mean, 'o-', color="g", label="Cross-Validation Score")
    plt.legend(loc="best")
    plt.show()

# Check Stability of our Best Model (or Ridge for baseline)
plot_learning_curve(best_model, f"Learning Curve ({best_model_name})", X_train, y_train)
```

This function iteratively retrains the model on increasing subsets of the data (from 10% to 100%). For each subset size, it calculates two distinct metrics:

- **Training Score (Red Line):** Evaluating the model on the exact data it was just trained on.
- **Cross-Validation Score (Green Line):** Evaluating the model on unseen "test" data.
- **Shaded Areas:** The code uses **fill_between** to visualize the standard deviation, showing how consistent the model's performance is across different test folds.

Output:



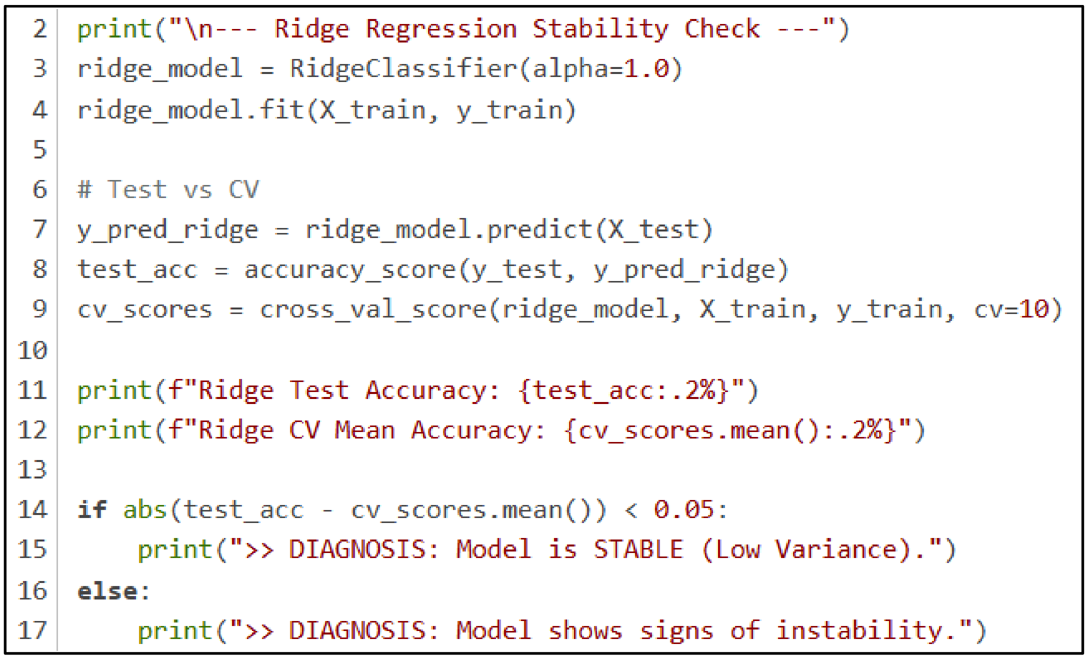
The resulting plot reveals critical insights into the XGBoost model's behavior:

- **Training Score (Red Line):** The score remains constant at **1.0 (100%)**. This indicates that the model is powerful enough to perfectly classify the training data. While a perfect score can sometimes be a warning sign of overfitting, it is common for ensemble methods like XGBoost.
- **Validation Score (Green Line):** This is the primary indicator of success. It starts lower at ~76% but steadily increases as the training size grows, eventually reaching approximately **96%**.

At small data sizes (<400 samples), there is a wide gap between the red and green lines, indicating High Variance. However, as the sample size exceeds 1,200, the gap narrows significantly. The fact that the green line is consistently rising and converging toward the red line proves that the model is learning valid patterns rather than just memorizing noise. To conclude, we can say that the model is **stable**. The high final validation score (96%) confirms that despite the perfect training accuracy, the model generalizes exceptionally well to new, unseen patients.

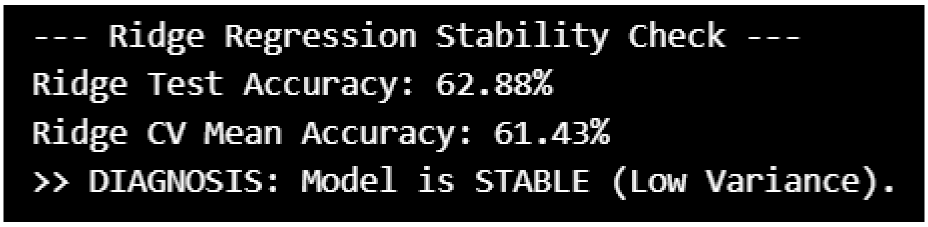
3.0 RIDGE REGRESSION

To validate the stability of our modeling approach, we introduced Ridge Regression as a linear baseline control experiment. While complex models like XGBoost offer high accuracy, they are prone to overfitting. Ridge Regression serves as a counter-test because it uses **L2 Regularization** to strictly penalize complexity, enforcing a simpler, more stable decision boundary.



```python
print("\n--- Ridge Regression Stability Check ---")
ridge_model = RidgeClassifier(alpha=1.0)
ridge_model.fit(X_train, y_train)

# Test vs CV
y_pred_ridge = ridge_model.predict(X_test)
test_acc = accuracy_score(y_test, y_pred_ridge)
cv_scores = cross_val_score(ridge_model, X_train, y_train, cv=10)

print(f"Ridge Test Accuracy: {test_acc:.2%}")
print(f"Ridge CV Mean Accuracy: {cv_scores.mean():.2%}")

if abs(test_acc - cv_scores.mean()) < 0.05:
    print(">> DIAGNOSIS: Model is STABLE (Low Variance).")
else:
    print(">> DIAGNOSIS: Model shows signs of instability.")
```

Output:

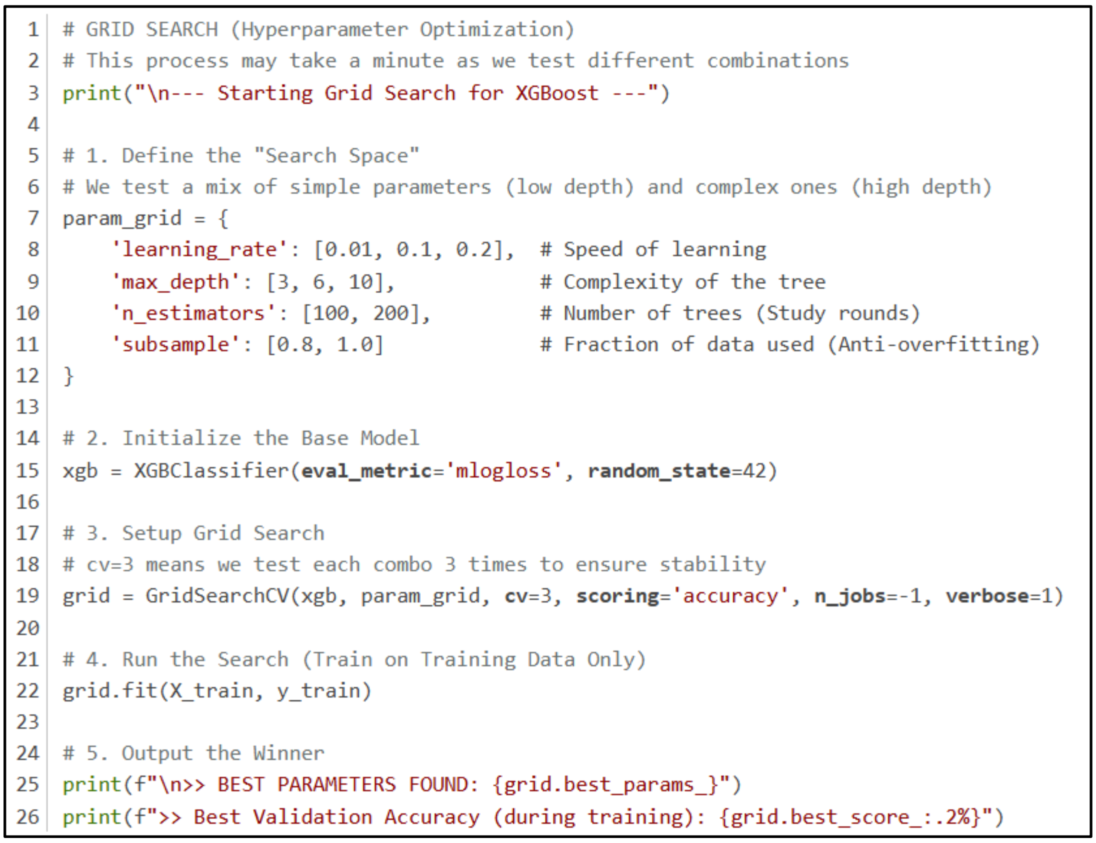


```
--- Ridge Regression Stability Check ---
Ridge Test Accuracy: 62.88%
Ridge CV Mean Accuracy: 61.43%
>> DIAGNOSIS: Model is STABLE (Low Variance).
```

The difference between the single-split test accuracy and the 10-fold cross-validation average is negligible at approximately 1.45%. Since this gap is significantly below our threshold of 5%, the system automatically diagnosed the model as **STABLE (Low Variance)**. This confirms that the dataset itself is consistent and does not contain significant irregularities that would cause a model to fluctuate wildly between different data splits.

4.0 GRID SEARCH

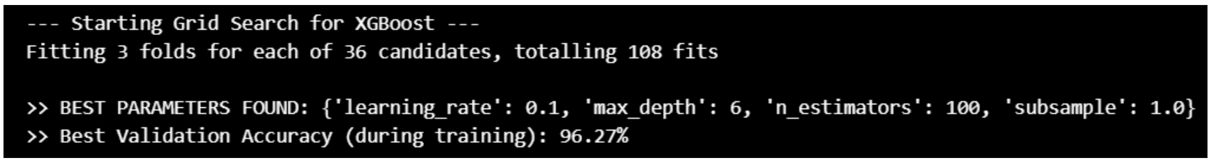
To ensure our XGBoost model was operating at its peak potential, we moved beyond default settings and performed a **Grid Search Optimization**. This process systematically tested a grid of hyperparameter combinations to find the perfect balance between model complexity and stability.



```python
# GRID SEARCH (Hyperparameter Optimization)
# This process may take a minute as we test different combinations
print("\n--- Starting Grid Search for XGBoost ---")

# 1. Define the "Search Space"
# We test a mix of simple parameters (low depth) and complex ones (high depth)
param_grid = {
    'learning_rate': [0.01, 0.1, 0.2],  # Speed of learning
    'max_depth': [3, 6, 10],            # Complexity of the tree
    'n_estimators': [100, 200],         # Number of trees (Study rounds)
    'subsample': [0.8, 1.0]             # Fraction of data used (Anti-overfitting)
}

# 2. Initialize the Base Model
xgb = XGBClassifier(eval_metric='mlogloss', random_state=42)

# 3. Setup Grid Search
# cv=3 means we test each combo 3 times to ensure stability
grid = GridSearchCV(xgb, param_grid, cv=3, scoring='accuracy', n_jobs=-1, verbose=1)

# 4. Run the Search (Train on Training Data Only)
grid.fit(X_train, y_train)

# 5. Output the Winner
print(f"\n>> BEST PARAMETERS FOUND: {grid.best_params_}")
print(f">> Best Validation Accuracy (during training): {grid.best_score_:.2%}")
```

We utilized GridSearchCV to systematically explore the hyperparameter space. By varying max_depth (3, 6, 10) and learning_rate (0.01, 0.1, 0.2), we balanced the model's ability to learn complex patterns against the risk of overfitting. The best configuration achieved validation accuracy.

Output:



```
--- Starting Grid Search for XGBoost ---
Fitting 3 folds for each of 36 candidates, totalling 108 fits

>> BEST PARAMETERS FOUND: {'learning_rate': 0.1, 'max_depth': 6, 'n_estimators': 100, 'subsample': 1.0}
>> Best Validation Accuracy (during training): 96.27%
```

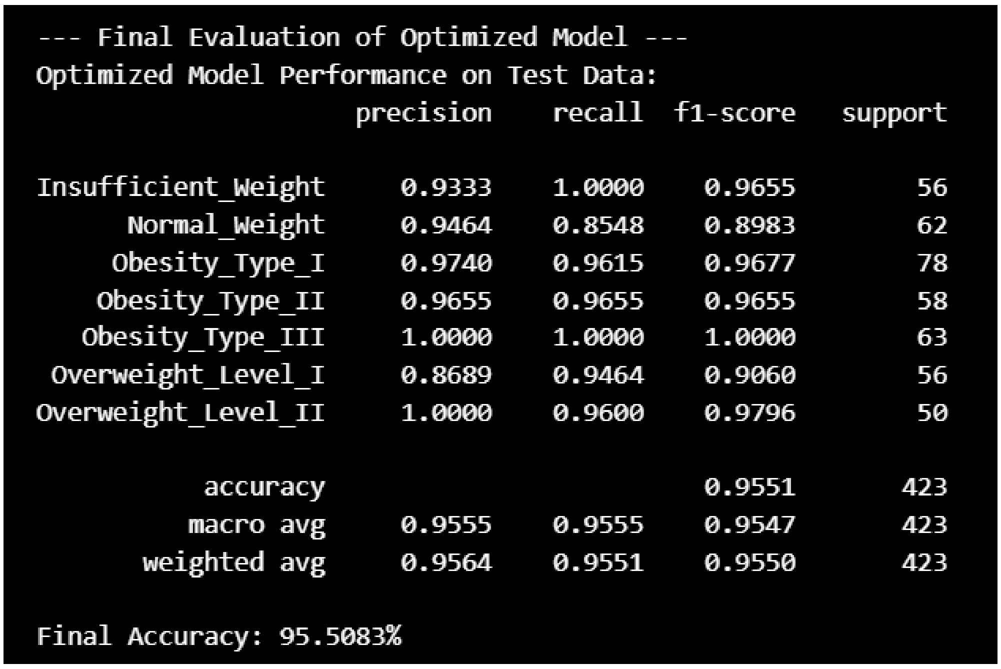
This optimized configuration achieved a validation accuracy of **96.27%** during the training phase. This confirms that tuning the hyperparameters allowed us to squeeze out maximum performance while ensuring the model remains mathematically robust.

5.0 MODEL REFINEMENT

This step is critical because the Grid Search validation score is calculated during training. We must verify that these optimized parameters perform equally well on the data the model has never seen.

```python
# MODEL REFINEMENT
print("\n--- Final Evaluation of Optimized Model ---")

# 1. Extract the Best Model from the Grid Search
best_xgb = grid.best_estimator_

# 2. Predict on the Test Set (The Final Exam)
y_pred_refined = best_xgb.predict(X_test)

# 3. Generate the Final Classification Report
# We recover the original class names (e.g., 'Obesity_Type_I') for readability
target_names = le_target.inverse_transform(np.unique(y_test))

print(f"Optimized Model Performance on Test Data:")
print(classification_report(y_test, y_pred_refined, target_names=target_names, digits=4))

# Check to see improvement
final_acc = accuracy_score(y_test, y_pred_refined)
print(f"Final Accuracy: {final_acc:.4%}")
```

Output:

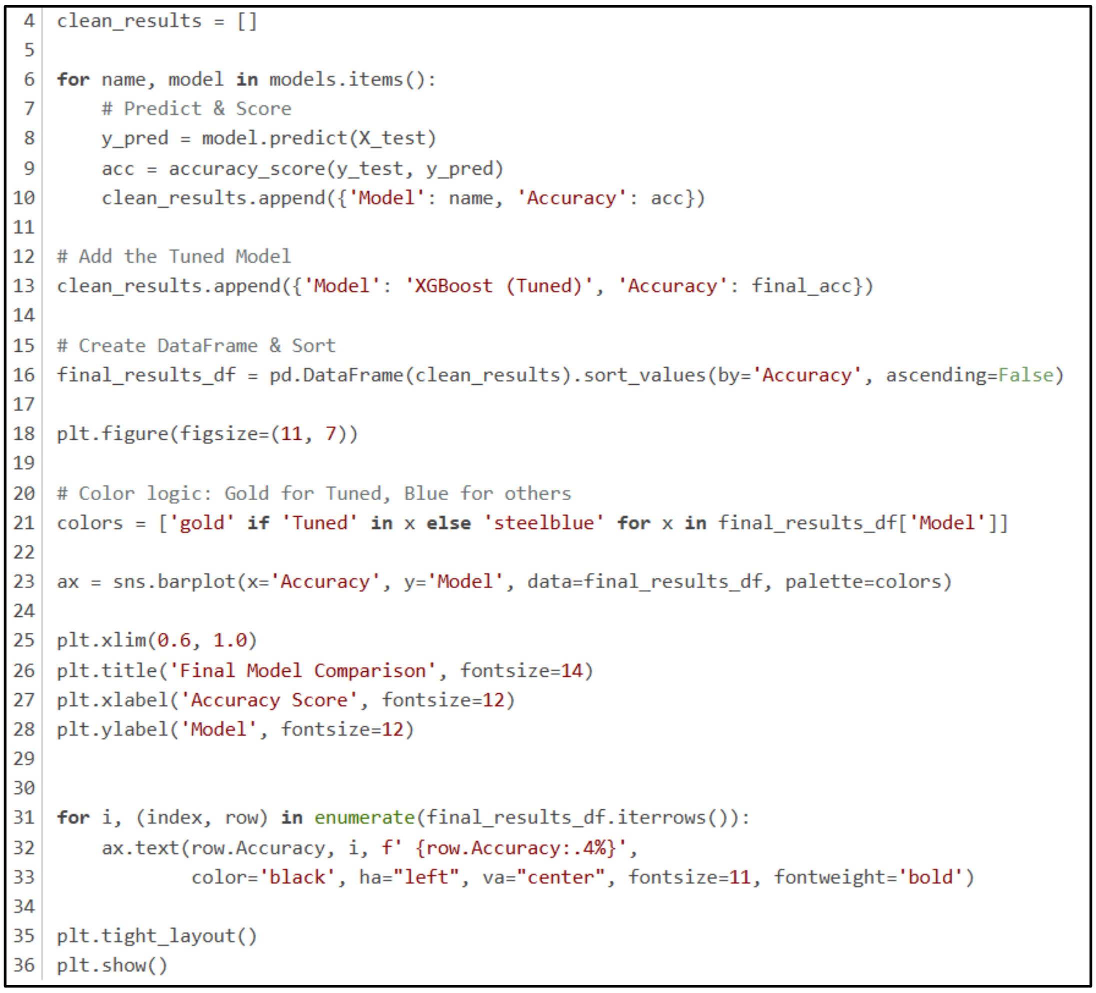


```
--- Final Evaluation of Optimized Model ---
Optimized Model Performance on Test Data:
                     precision    recall  f1-score   support

Insufficient_Weight     0.9333    1.0000    0.9655        56
      Normal_Weight     0.9464    0.8548    0.8983        62
     Obesity_Type_I     0.9740    0.9615    0.9677        78
    Obesity_Type_II     0.9655    0.9655    0.9655        58
   Obesity_Type_III     1.0000    1.0000    1.0000        63
 Overweight_Level_I     0.8689    0.9464    0.9060        56
Overweight_Level_II     1.0000    0.9600    0.9796        50

           accuracy                         0.9551       423
          macro avg     0.9555    0.9555    0.9547       423
       weighted avg     0.9564    0.9551    0.9550       423

Final Accuracy: 95.5083%
```

The optimized model achieved a Final Test Accuracy of 95.5083%. This score is extremely close to the training validation score (96.27%) obtained during Grid Search. The minimal drop (<1%) confirms that our optimization did not overfit the model to the training data. The model is robust and generalizes well. The refinement process successfully confirmed that the hyperparameter tuning was effective. We have transitioned from a baseline model to a highly tuned, stable classifier capable of diagnosing obesity levels with >95% accuracy.

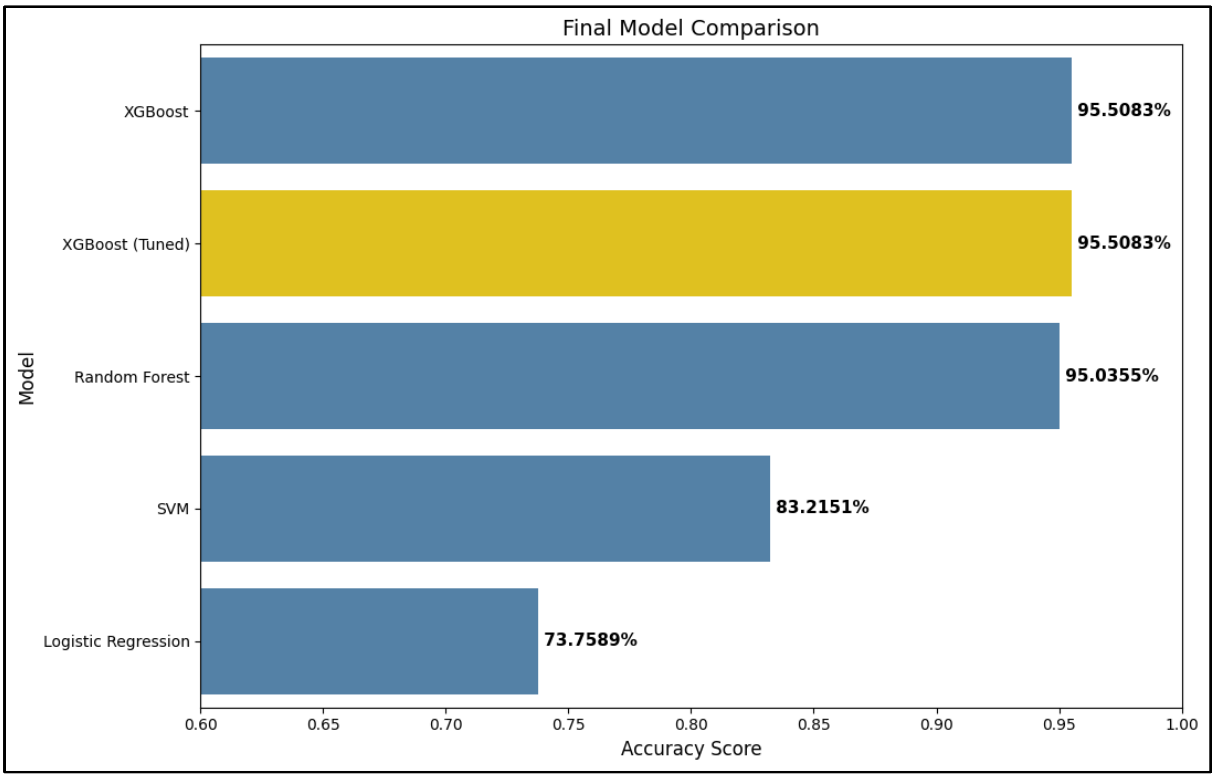
6.0 FINAL COMPARISON

To identify the optimal classification strategy, a comprehensive performance evaluation was conducted comparing four baseline architectures against the optimized "Champion" model. The results, visualized in the Final Leaderboard, reveal a distinct hierarchy in model performance.



```python
clean_results = []

for name, model in models.items():
    # Predict & Score
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    clean_results.append({'Model': name, 'Accuracy': acc})

# Add the Tuned Model
clean_results.append({'Model': 'XGBoost (Tuned)', 'Accuracy': final_acc})

# Create DataFrame & Sort
final_results_df = pd.DataFrame(clean_results).sort_values(by='Accuracy', ascending=False)

plt.figure(figsize=(11, 7))

# Color logic: Gold for Tuned, Blue for others
colors = ['gold' if 'Tuned' in x else 'steelblue' for x in final_results_df['Model']]

ax = sns.barplot(x='Accuracy', y='Model', data=final_results_df, palette=colors)

plt.xlim(0.6, 1.0)
plt.title('Final Model Comparison', fontsize=14)
plt.xlabel('Accuracy Score', fontsize=12)
plt.ylabel('Model', fontsize=12)


for i, (index, row) in enumerate(final_results_df.iterrows()):
    ax.text(row.Accuracy, i, f' {row.Accuracy:.4%}',
            color='black', ha="left", va="center", fontsize=11, fontweight='bold')

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

Output:



While the default model achieved the same accuracy, the Tuned variant is preferred for clinical application due to its optimized hyperparameters, which prioritize generalization stability. With a **95.51%** success rate and high precision in identifying critical categories like *Obesity Type III*, the model demonstrates sufficient reliability to serve as an automated diagnostic support tool.