Dataset: Classification of Alzheimer's Disease Dataset (Kaggle competition).

Team Members:

- Member 1: Sung-chi (William) Wu (1006990446)
- Member 2: Xing Yu (Megan) Wang (1008446889)

Kaggle Team Name: Team WM

Final Results:

- **Ranking**: 64
- **Prediction Score**: 0.94894 (on Kaggle)

2. Problem Statement
Specific Problem:
The goal of this project is to develop a machine learning model to predict the likelihood of Alzheimer's disease in patients based on the provided dataset on each patient's demographic information, lifestyle factors, medical history, as well as other medical records.

Relevance and Importance:
Alzheimer's is a critical public health issue with significant social and economic implications. Early prediction and diagnosis can help with timely interventions, improving patients' quality of life and reducing healthcare costs. Accurate prediction models can also assist in research and resource allocation.

3. Statistical Analyses
3.1 EDA and Data Preprocessing
3.1.1 Examining Missing Values and Outliers
We started our analysis by cleaning the dataset to focus on meaningful features. Columns like PatientID and DoctorInCharge were removed because they didn't contribute to the Alzheimer's diagnosis. We checked for missing values and found none. To identify potential outliers, we used the Interquartile Range (IQR) method with a 1.5 threshold for continuous features, but no significant outliers were detected. For categorical variables, we used bar plots to visualize their distributions in training dataset. Most categories showed imbalance except for Gender. However, statistical tests revealed that only MemoryComplaints and BehavioralProblems were significantly associated with the target feature (p-value < 0.05). Since tree-based models like Random Forest, Catboost and XGBoost are naturally robust to imbalanced data, these imbalances are less concerning. Whereas, tree-based models are generally unaffected by scaling because their decision-making process does not rely on the magnitude of feature values (Loh 2011), we standardized all features to ensure fair performance when using Support Vector Machine (SVM), which is more sensitive to features scaling. Lastly, categorical features (both binary and ordinal) are converted to category data type during preprocessing for tree-based models to ensure proper handling of these features during model training.

3.1.2 Examining Class Imbalance

The dataset showed a class imbalance, with 35.37% of samples diagnosed with Alzheimer's, which caused categorical variables biased distributions. While tree-based models can handle class imbalance effectively (Buda et al. 2018), we will choose to explicitly address it by incorporating class weights later when initiating the models. For example, in Catboost, we will use the class_weights parameter, while in XGBoost, equivalent parameter like scale_pos_weight will be configured to reflect the class distribution. This ensures that the minority class is given adequate weight during training.

3.1.3 Examining Multicollinearity

While tree-based models are less sensitive to multicollinearity compared to linear models, redundant features can sometimes lead to increased noise or unnecessarily complex models. This phenomenon is explained by their reliance on feature importance calculations during the splitting process rather than direct relationships between predictors. As highlighted by Dormann et al. (2013), multicollinearity in tree-based models often has a smaller but non-negligible impact on model interpretability. To mitigate potential risks, we will assess the impact of removing features with high multicollinearity (identified through VIF) using cross-validation to ensure there are no adverse effects on performance.

3.1.4 Examining Non-Linear Relationships

Using Partial Dependence Plots (PDPs), we explored potential non-linear effects between features and the target variable. These plots revealed clear non-linear relationships for many features, including BMI, AlcoholConsumption, SleepQuality, and several others. Notably, we observed threshold effects in key features like MMSE, FunctinoalAssessment, and ADL, where their impact on the target feature shifted sharply at values of approximately 20, 4, and 4, respectively.

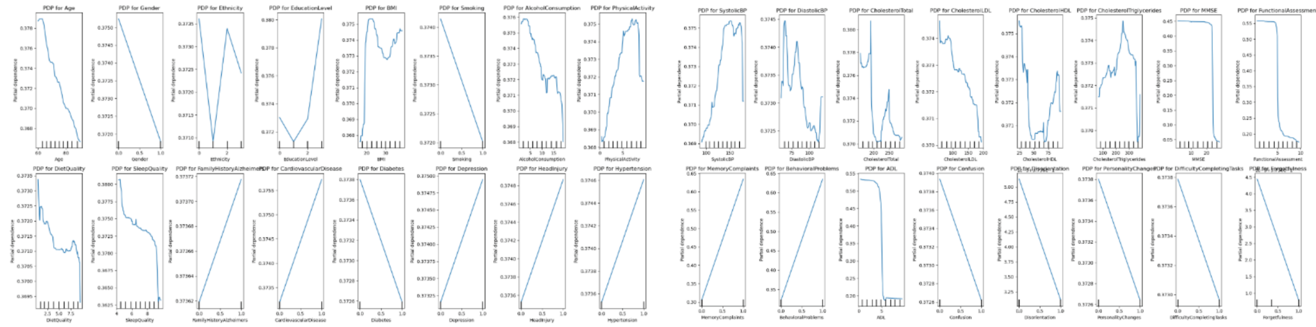

Figure 1: PDP Plots for Identifying Non-linear Relationships

3.2 Choice of Machine Learning Algorithms

3.2.1 Research Focus – Random Forest, Catboost, and XGBoost

We focus on research on tree-based models, including Random Forest, Catboost, and XGBoost, which are well-suited for our classification problem due to their inherent strengths and alignment with the dataset. Tree-based models are particularly effective for datasets with complex, non-linear relationships and mixed data types, like ours. Our EDA has identified MMSE, FunctionalAssessment, and ADL exhibiting threshold effects, as well as other continuous features exhibiting non-linear patterns. Tree-based methods naturally capture such relationships without requiring additional feature transformations. This makes them both powerful and easy to implement.

Another strength of tree-based models is their robustness to feature scaling and multicollinearity. Unlike linear models, they do not require all features to be on the same scale (Loh 2011). This allowed us to work with raw data for most features. Additionally, while multicollinearity—identified in features like BMI and SleepQuality—can sometimes add noise to predictions, it is less problematic for tree-based methods. Random Forest and gradient boosting models handle such redundancy by selecting the most informative splits during tree construction (Dormann et al. 2013). Tree-based models are also resilient to class imbalance. Our EDA identified that only 35.37% of samples diagnosed with Alzheimer's. Tree-based models can naturally handle imbalance by splitting the data based on feature importance. For example, Random Forest balances predictions by averaging across many decision trees.

Gradient boosting models like CatBoost and XGBoost extend the capabilities of tree-based methods by sequentially building trees, where each tree learns from the mistakes of the previous one. This iterative process typically results in higher accuracy than methods like Random Forest, which trains trees independently. The main reason we selected Catboost was due to its ability to handle categorical data directly without preprocessing steps like one-hot encoding. This is especially relevant for our dataset, which contains 17 categorical features (binary and ordinal). It is also designed to reduce overfitting, which is a common challenge in gradient boosting.

XGBoost is another gradient boosting model that we investigated. We selected XGBoost primarily for its speed, which enables us to experiment with multiple configurations, hyperparameters, and preprocessing techniques efficiently. This advantage is particularly important given that our model training and testing are conducted on JupyterHub, which has limited computing resources. Additionally, the speed of XGBoost is beneficial for our workflow, as we validate each model using K-fold cross-validation, which requires repeated training on different data splits. This efficiency allows us to optimize our models with shorter experimentation time.

Lastly, Random Forest was used as a baseline model to compare against Catboost and XGBoost.

3.2.2 Alternative Models

During our initial experimentation, we tested Logistic Regression and SVM as alternative models. However, their baseline performance was noticeably weaker compared to the tree-based models we ultimately focused on (Figure 2). While Logistic Regression and SVM have their strengths—such as simplicity, interpretability, and effectiveness in high-dimensional data—they struggled to capture the complex interactions and non-linear relationships present in our dataset. Consequently, we decided to exclude them from further analysis and prioritized tree-based models, which demonstrated significantly better performance in capturing these complexities.

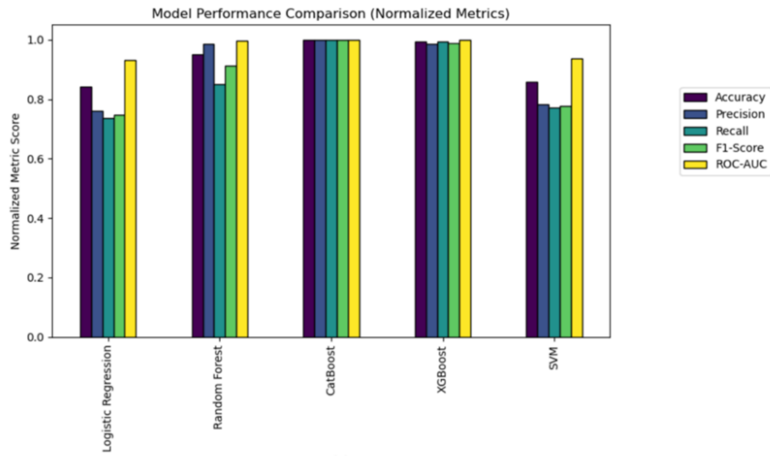

Figure 2: Baseline Models Performance Comparison

Note: Continuous features were scaled for Logistic Regression and SVM to ensure that numerical features contribute equally to model training.

3.3 Attempts to Improve Model Performance

3.3.1 Feature Engineering (Excl. Feature Selection)

Based on insights from our EDA, we implemented several feature engineering steps to enhance model performance. One key observation from the PDPs, was the presence of threshold effects in three features: FunctionalAssessment, ADL, and MMSE. These effects occurred at approximate values of 20, 4, and 4, respectively. To precisely identify the optimal threshold positions, we conducted a more rigorous analysis using the following approach.

We calculated the optimal thresholds by analyzing the gradient of the PDP curves. By identifying the points of maximum gradient change in the PDPs, we pinpointed the values where the features' impact on the target variable shifted most sharply. The results are summarized in Table 1:

Table 1: Optimal Thresholds for FunctionalAssessment, ADL, and MMSE

Feature	**Optimal Threshold**
FunctionalAssessment	4.93
ADL	4.96
MMSE	23.82

Following this analysis, we created dummy variables for MMSE, FunctionalAssessment, and ADL to reflect their respective thresholds. These dummies were tested in our models to assess their impact on predictive performance.

3.3.2 Feature Selection

In terms of feature selection, Megan and I took different approaches when investigating the best set of predictors to include in each model. As a result, different feature selection processes were applied to Random Forest, XGBoost, and CatBoost. Although our approaches differed, we shared the same goal: to identify the best set of features that would maximize each model's performance, which we can then compare later.

For Random Forest and XGBoost, manual feature selection was applied. Baseline models were initially trained with all features, and subsets were then selected based on their importance scores. Specifically, experiments were conducted using the top 17, top 11, and top 5 features. While this method is not the most robust—since it does not account for interactions or potential redundancies among features—it is a straightforward and efficient way to quickly experiment and identify potentially significant predictors. To ensure that the selected feature subsets generalized well to unseen data, models with different feature sets were validated using K-fold cross-validation.

In contrast, a recursive approach was used for feature selection with CatBoost. This method involved iteratively eliminating the least important features while simultaneously evaluating the model's performance using K-fold cross-validation. Starting with all features, the CatBoost model was trained to rank features by importance, the least important ones were removed, and the model was re-trained at each step. By validating the model at each iteration, this method ensured that the selected feature set maintained or improved the model's performance while potentially enhancing generalizability and robustness. Although this process was computationally intensive, it allowed for a more systematic exploration of feature importance and interactions compared to the manual approach. The performance of each model with different sets of features will be discussed later in the Results section.

3.3.3 Hyperparameter Tuning

Hyperparameter tuning was the final step before model validation. This order ensured we evaluated all models under their optimal conditions, making comparisons fair and meaningful. For tuning, we used Grid Search, which systematically tests combinations of hyperparameters to find the best setup. The parameters we tested included the number of iterations, learning rate, tree depth, regularization strength, and bagging temperature. Table 2 presents an example of the parameters and values that we explored using Grid Search.

Table 2: Grid Search Parameter Grid Example

Grid Search	Iterations	Learning Rate	Depth	L2_leaf_reg	Bagging Temperature
Values Tested	500, 1000	0.01, 0.05, 0.1	4, 6, 8	1, 3, 5	0.2, 0.5, 1

These values were chosen to balance thorough exploration with practical runtime limits. For instance, we tested tree depths (4, 6, 8) to explore models of varying complexity, and learning rates (0.01, 0.05, 0.1) to capture different speeds of training. Parameters like bagging temperature and regularization strength helped us examine their effects on the model's stability and ability to generalize. These choices were informed, for instance, by the CatBoost documentation, which suggests optimal tree depths between 4 and 10, with values like 6 and 10 often performing well (CatBoost Documentation n.d.).

Since Grid Search was time-consuming, we set a rule to limit when tuning was repeated. We only re-ran Grid Search if there were major changes to the feature set, like adding, removing, or transforming over 20% of the features. Previous literatures have suggested similar strategies for balancing computational efficiency and model optimization (Hutter et al., 2019). For minor adjustments to features, we retained the previously identified best hyperparameters to save time as our previous experimentation showed that smaller feature changes did not substantially alter the optimal hyperparameter values.

3.4 Model Validation

The final step of our process was model validation. We used 5-fold cross-validation to evaluate each model, instead of a single train-test split, to ensure that evaluation metrics such as ROC-AUC, accuracy, precision, recall, and F1-score reflect the model's ability to generalize rather than its fitting to a specific train-test split. During hyperparameter tuning and feature selection, we used ROC-AUC as the primary evaluation metric to optimize. This decision was informed by the dataset's class imbalance, with only 35.37% of samples diagnosed with Alzheimer's. In imbalanced datasets, accuracy can be misleading, as a model that consistently predicts the majority class can achieve high accuracy while failing to identify the minority class.

ROC-AUC addresses this limitation by evaluating the model's ability to rank predictions and distinguish between classes across all possible thresholds. Its threshold independence makes it particularly useful during hyperparameter tuning and feature selection, where the focus is on assessing the model's overall discriminative power rather than performance at a specific threshold. This approach allowed us to build robust models that effectively account for the dataset's imbalance.

That said, we later recognized that the competition specifies accuracy as the primary evaluation metric for submissions. This oversight came to light after completing most of our analysis. While this adjustment highlights the need for closer alignment with evaluation criteria, we believe our approach remains justified, as it ensured proper consideration of the minority class. Accuracy was monitored as a secondary metric throughout the analysis, and our models consistently demonstrated strong performance on this measure.

4. Results

In this section we present the results of our analysis, including feature selection, inclusion of threshold dummies, hyperparameter tuning, and the impact of ensemble methods. Our primary focus is to evaluate the predictive performance of Random Forest, Catboost, and XGBoost, alongside their baseline.

4.1 Baseline Model Performance

We began by evaluating the baseline performance of all models using the full feature set. Random Forest, CatBoost, and XGBoost demonstrated high accuracy and ROC-AUC, with Random Forest achieving the highest baseline ROC-AUC of 0.956. Catboost and XGBoost closely follow in this metric, achieving 0.941 and 0.951 respectively.

4.2 Feature Selection

Using manual selection for Random Forest and XGBoost, we tested subsets of features ranked by their importance scores (e.g., top 23, top 15, and top 9). Table 3 presents the results of our features selection. Performance remained stable or even improved slightly, indicating that many features were redundant or less informative.

Table 3: Feature Selection Results (Post-Hyperparameter Tuning)

Model	Feature Set	Mean Accuracy	Mean Precision	Mean Recall	Mean F1 Score	Mean ROC-AUC
Random Forest	All features	0.955	0.947	0.925	0.936	0.956
	Top 23	0.940	0.956	0.869	0.910	0.951
	Top 15	0.944	0.952	0.885	0.917	0.956
	Top 9	0.954	0.954	0.915	0.934	0.951
Catboost	All features	0.954	0.958	0.949	0.933	0.941
	Top 11 (via CFECV)	0.978	0.962	0.954	0.938	0.946
XGBoost	All features	0.950	0.944	0.914	0.928	0.951
	Top 30	0.947	0.942	0.906	0.923	0.952
	Top 20	0.948	0.942	0.908	0.924	0.951
	Top 15	0.948	0.937	0.914	0.925	0.952

	Top 8	0.944	0.940	0.899	0.919	0.950
	Top 5	0.944	0.941	0.898	0.919	0.952

For Random Forest and XGBoost, we were able to narrow down the feature set to the top 9 and top 5 most important features without sacrificing much of the performance. For CatBoost, we employed a recursive feature elimination approach, which identified the top 11 features. This reduced feature set significantly improved accuracy from 0.954 to 0.978 while maintaining a high ROC-AUC of 0.946. As a result, a higher model performance was achieved together with reduced model complexity.

4.3 Threshold Effects

To investigate the threshold effects observed in our EDA for FunctionalAssessment, MMSE, and ADL, we experimented with two different approaches. First, we replaced the original features with their corresponding threshold dummies (Model 4.1). Second, we retained the original features and added their threshold dummies to the feature set selected through CFECV, resulting in Model 4.3. The goal of these experiments was to assess whether incorporating threshold dummies could enhance model performance.

The results of these experiments were compared to the performance of the CatBoost model with the top 11 features selected using CFECV, which has consistently shown to be our best-performing model thus far. The CatBoost model with the top 11 features achieved an accuracy of 0.978 with high precision, recall, F1-score, and ROC-AUC values, making it the benchmark for comparison. In contrast, Model 4.1, which replaced the original features with their threshold dummies, performed notably worse. Accuracy dropped to 0.950, and all other metrics showed a corresponding decline. This suggests that replacing the original features with threshold dummies resulted in a loss of valuable information that could not be fully captured by the dummies.

Model 4.3, which included the threshold dummies alongside the original features, performed slightly better than Model 4.1 but still fell short of the performance achieved by the CatBoost model with the top 11 features. While adding threshold dummies introduced additional information about non-linear effects, this did not lead to significant improvements. Accuracy for Model 4.3 reached 0.956 but precision, recall, and F1-score were all lower compared to the baseline CatBoost model. Moreover, the ROC-AUC for Model 4.3 remained below that of the CFECV-selected model, further indicating that the additional complexity introduced by the threshold dummies did not translate into better predictions.

4.4 Hyperparameter Tuning

After determining the optimal feature sets, we proceeded to hyperparameter tuning using GridSearch to maximize the performance of our models. For CatBoost, the benefits of hyperparameter tuning were particularly notable. Recall improved from 0.933 to 0.954, and accuracy increased significantly from 0.954 to 0.978, demonstrating the model's enhanced ability to correctly identify positive cases without compromising overall predictive power. Similarly, both Random Forest and XGBoost saw measurable improvements in key metrics such as recall and F1 scores.

4.5 Ensemble Methods

To conclude the analysis, we tested ensemble methods that combined the best-performing Random Forest model (using the top 9 features) and the best-performing CatBoost model. These ensemble methods included weighted average, stacking, and soft voting, which leverage the strengths of multiple base models. The goal was to determine whether these approaches could yield additional improvements over the individual models by combining their predictions in complementary ways. Grid search was used to find the optimal weights for the weighted average ensemble model, which resulted in a weight of 0.7 to Catboost and 0.3 to Random Forest.

Table 4: Ensemble Methods Results

Model	AUC	Accuracy	Precision	Recall	F1

Weighted Average (0.7 Catboost; 0.3 RF)	0.958	0.958	0.952	0.928	0.940
Stacking	0.958	0.958	0.952	0.929	0.940
Voting	0.958	0.958	0.952	0.928	0.940

Table 4 presents the results of the three ensemble methods. The weighted average ensemble, which assigned weights of 0.7 to our best performing CatBoost model and 0.3 to Random Forest, achieved the highest AUC score of 0.956. This suggests that giving more weight to the stronger performer (CatBoost) resulted in a slight edge in discriminative power. However, the stacking and soft voting ensembles produced nearly identical performance metrics, including AUC, accuracy, precision, recall, and F1-score, with AUC values both equal to 0.958.

Table 5: Top Performing Models for each Model Type

Model	Best Feature Set	Tuned Hyperparameters	Evaluation Metrics
Random Forest	Selected Features: MMSE, FunctionalAssessment, ADL, BehaviroalProblems, MemoryComplaints, CholesterolTriglycerides, CholesterolHDL, DeitQuality, Age	- n_estimators: 100 - max_depth: 10 - min_samples_split: 5 - min_samples_leaf: 2 - max_features: None	Accuracy: 0.954 Precision: 0.954 Recall: 0.915 F1: 0.934 ROC-AUC: 0.951
Catboost	Selected Features: AlcoholConsumption, SleepQuality, SystolicBP, CholesterolTotal, CholesterolHDL, CholesterolTriglycerides, MMSE, FunctionalAssessment, MemoryComplaints, BehavioralProblems, ADL	- iterations: 500 - learning_rate: 0.01 - depth: 4 - l2_leaf_reg: 5 - bagging_temperature: 0.2	Accuracy: 0.978 Precision: 0.962 Recall: 0.954 F1: 0.938 ROC-AUC: 0.946
XGBoost	Selected Features: BehavioralProblems, MemoryComplaints, MMSE, ADL, FunctionalAssessment,	- n_estimators: 100 - learning_rate: 0.3 - max_depth: 6 - min_child_weight: 1 - gamma: 0 - subsample: 1 - colsample_bytree: 1 - lambda: 1 - alpha: 0	Accuracy: 0.944 Precision: 0.941 Recall: 0.898 F1: 0.919 ROC-AUC: 0.952
Weighted Average Ensemble (RF + Catboost)	- RF: Selected Features (see above) - Catboost: Selected Features (see above)	Combined tuned hyperparameters from RF and Catboost	Accuracy: 0.958 Precision: 0.952 Recall: 0.929 F1: 0.940 ROC-AUC: 0.958

Comparing to the best performing individual models, the weighted average ensemble model, combining the strengths of the top performing models of Random Forest and CatBoost, provided the best overall balance across all metrics. By assigning 70% weight to the CatBoost model and 30% weight to the Random Forest model, the ensemble achieved an accuracy of 0.958, precision of 0.952, recall of 0.929, and an F1 score of 0.940. Its ROC-AUC of 0.958 was the highest among all models.

5. Conclusion

This project aimed to develop a machine learning model capable of predicting the likelihood of Alzheimer's disease based on a variety of patient data. Given the critical importance of early diagnosis for improving patient outcomes and optimizing healthcare resources, the project sought to identify a robust and accurate model through feature engineering, hyperparameter tuning, and ensemble methods.

Our analysis began by comparing the baseline performance of Logistic Regression, SVM, Random Forest, CatBoost, and XGBoost models. Logistic Regression and SVM were excluded early due to their inability to handle the dataset's complex non-linear relationships and feature interactions effectively. Tree-based models, on the other hand, demonstrated superior performance, with CatBoost emerging as the strongest individual model. CatBoost achieved an accuracy of 0.978, a recall of 0.954, and an F1 score of 0.938 when paired with a selected feature set of 11 predictors identified through CFECV. Random Forest and XGBoost also delivered strong results, with Random Forest achieving an accuracy of 0.954 and XGBoost achieving 0.952.

Threshold dummies were explored for features that exhibited clear non-linear effects in our EDA, such as MMSE, FunctionalAssessment, and ADL. However, replacing or supplementing these features with threshold dummies did not improve model performance. Instead, the CatBoost model with the CFECV-selected top 11 features retained its position as the best-performing individual model, highlighting that these features' original forms contained sufficient information for accurate predictions.

Hyperparameter tuning through Grid Search significantly improved model performance. For CatBoost, recall increased from 0.933 to 0.954, and accuracy rose from 0.954 to 0.978. Random Forest and XGBoost also benefited from tuning, with notable gains in precision and F1 scores. Ensemble methods were then investigated to further enhance performance by combining the strengths of the individual models.

The weighted average ensemble model, which assigned 70% and 30% weight to each of our best performing Catboost model and Random Forest model, achieved an accuracy of 0.958, precision of 0.952, recall of 0.929, and an F1 score of 0.940. Its ROC-AUC of 0.958 was the highest among all models. This model was used to submit our final prediction results on Kaggle, achieving a score of 0.94894 and a rank of 64.

6. Limitations & Future Directions

While our study demonstrates strong predictive performance, there are several limitations to our approach. First is our focus on tree-based models, which, although effective for this dataset, may not fully capture the complexity of certain interactions between features. For example, while we leveraged PDPs to identify threshold effects for features such as MMSE, FunctionalAssessment, and ADL, the PDPs also revealed many other non-linear relationships in features like BMI, SleepQuality, and CholesterolHDL. These non-linearities were not explicitly modeled in our feature engineering steps, potentially leaving out important patterns.

Another limitation lies in our handling of interaction effects between features. Our models do not explicitly account for potential synergies or dependencies between variables, despite evidence that such interactions may play a critical role in predicting Alzheimer's risk. For instance, features like PhysicalActivity and DietQuality could interact to affect outcomes, but this was not thoroughly examined in our analysis. Future work could experiment with different interaction terms or include more sophisticated feature engineering strategies to address this limitation.

Finally, while our use of ROC-AUC as the primary evaluation metric was justified given the class imbalance in the dataset, the competition emphasized accuracy. Although our models showed competitive accuracy scores, future studies should align evaluation metrics with specific project goals or competition criteria from the outset to avoid potential misalignment. Additionally, while cross-validation mitigated overfitting risks, the limited size of our dataset means that further validation on external datasets would be valuable to confirm the generalizability of our findings.

References

Buda, Mateusz, Atsuto Maki, and Maciej A. Mazurowski. 2018. "A Systematic Study of the Class Imbalance Problem in Convolutional Neural Networks." *Neural Networks* 106: 249–59. https://doi.org/10.1016/j.neunet.2018.07.011.

CatBoost Documentation. n.d. "Parameter Tuning." Accessed November 24, 2024. https://catboost.ai/docs/en/concepts/parameter-tuning.

Dormann, C. F., Elith, J., Bacher, S., Buchmann, C., Carl, G., Carré, G., García Marquéz, J. R., Gruber, B., Lafourcade, B., Leitão, P. J., Münkemüller, T., McClean, C., Osborne, P. E., Reineking, B., Schröder, B., Skidmore, A. K., Zurell, D., & Lautenbach, S. (2013). Collinearity: A review of methods to deal with it and a simulation study evaluating their performance. *Ecography*, *36*(1), 27–46. https://doi.org/10.1111/j.1600-0587.2012.07348.x

Ke, Guolin, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree." Advances in Neural Information Processing Systems 30 (2017): 3146–3154.

Loh, Wei-Yin. "Classification and Regression Trees." *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 1, no. 1 (2011): 14–23.

Prokhorenkova, Liudmila, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. "CatBoost: Unbiased Boosting with Categorical Features." Advances in Neural Information Processing Systems 31 (2018): 6638–6648.

Appendix

1.

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:
get_ipython().system('pip install lightgbm')
get_ipython().system('pip install catboost')
get_ipython().system('pip install xgboost')

import pandas as pd
import os
from sklearn.model_selection import train_test_split, GridSearchCV
from catboost import CatBoostClassifier
from sklearn.metrics import classification_report, roc_auc_score

# Clearing the Results Log File
results_file = 'model_performance_tracking.csv'

# Clear the file (overwrite with empty DataFrame)
if os.path.exists(results_file):
    os.remove(results_file)  # Remove the file if it exists
# Create a new empty DataFrame with the required headers
results_df = pd.DataFrame(columns=['Model', 'AUC', 'Accuracy', 'Precision', 'Recall', 'F1-Score',
                                   'Hyperparameters', 'Notes'])
results_df.to_csv(results_file, index=False)  # Save the empty file

# Load training data and preprocess
data = pd.read_csv('train.csv')

# ## Preprocessing
# In[2]:
# Remove irrelevant columns
data = data.drop(columns=['PatientID', 'DoctorInCharge'])

# Split features and target
X = data.drop(columns='Diagnosis')
y = data['Diagnosis']

categorical_features = [
    'Gender', 'Ethnicity', 'EducationLevel', 'Smoking',
    'FamilyHistoryAlzheimers', 'CardiovascularDisease', 'Diabetes',
    'Depression', 'HeadInjury', 'Hypertension', 'MemoryComplaints',
    'BehavioralProblems', 'Confusion', 'Disorientation',
    'PersonalityChanges', 'DifficultyCompletingTasks', 'Forgetfulness',
]

X[categorical_features] = X[categorical_features].astype('category')


# # EDA
# ## Check if the classes are balanced
# In[3]:
print(y.mean())


# In[4]:
from sklearn.utils.class_weight import compute_class_weight

class_weights = compute_class_weight(
    class_weight='balanced',
    classes=[0, 1],
    y=y
)
class_weights = {0: class_weights[0], 1: class_weights[1]}
print(class_weights)
```

```
# ## Check for non-linear relationships
# In[5]:
from sklearn.inspection import PartialDependenceDisplay
import matplotlib.pyplot as plt
from math import ceil

model = CatBoostClassifier(class_weights=list(class_weights.values()),
                           cat_features=categorical_features,
                           random_state=42,
                           verbose=0)
model.fit(X, y)

# Select features to plot (numerical + important categorical ones)
features_to_plot = X.columns.tolist()

num_features = len(features_to_plot)
num_cols = 8 #Number of columns in the grid
num_rows = ceil(num_features / num_cols)

fig, axes = plt.subplots(num_rows, num_cols, figsize=(20, 5 * num_rows))
axes = axes.ravel()

# Plot PDPs for all features
for i, feature in enumerate(features_to_plot):
    PartialDependenceDisplay.from_estimator(
        model,
        X,
        [feature],
        kind="average",
        ax=axes[i]  # Assign subplot
    )
    axes[i].set_title(f"PDP for {feature}")

for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])

output_image = "pdp_plots.png"
plt.tight_layout()
plt.show()

# In[6]:
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier
from catboost import CatBoostClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score, accuracy_score, precision_score, recall_score, f1_score
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize models
models = {
    "Logistic Regression": LogisticRegression(max_iter=1000, random_state=42),
    "Random Forest": RandomForestClassifier(n_estimators=100, random_state=42),
    "CatBoost": CatBoostClassifier(iterations=500, learning_rate=0.05, depth=6, cat_features=list(X.select_dtypes(include='category').columns),
random_state=42, verbose=0),
    "XGBoost": XGBClassifier(use_label_encoder=False, eval_metric='logloss', enable_categorical=True, n_estimators=500, learning_rate=0.05,
max_depth=6, random_state=42),
    "SVM": SVC(probability=True, kernel='linear', random_state=42)
}

# Initialize storage for metrics
results = {
    "Model": [],
    "Accuracy": [],
    "Precision": [],
```

```
    "Recall": [],
    "F1-Score": [],
    "ROC-AUC": []
}

# Evaluate models
for name, model in models.items():
    print(f"Training {name}...")
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:, 1] if hasattr(model, "predict_proba") else None

    # Compute metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='binary')
    recall = recall_score(y_test, y_pred, average='binary')
    f1 = f1_score(y_test, y_pred, average='binary')
    roc_auc = roc_auc_score(y_test, y_proba) if y_proba is not None else None

    # Store results
    results["Model"].append(name)
    results["Accuracy"].append(accuracy)
    results["Precision"].append(precision)
    results["Recall"].append(recall)
    results["F1-Score"].append(f1)
    results["ROC-AUC"].append(roc_auc)

# Create DataFrame for results
results_df = pd.DataFrame(results)

# Normalize metrics for visualization
normalized_results = results_df.set_index("Model")
normalized_results = normalized_results / normalized_results.max()

# Plot results
normalized_results.plot(kind='bar', figsize=(10, 6), colormap='viridis', edgecolor='black')
plt.title("Model Performance Comparison (Normalized Metrics)")
plt.ylabel("Normalized Metric Score")
plt.xlabel("Model")
plt.legend(loc='lower right', bbox_to_anchor=(1.3, 0.5))
plt.tight_layout()
plt.show()


# ## Prioritize Feature Transformation for Most Important Features
# ### Identify the Top Features using Catboost Model

# In[7]:
from catboost import CatBoostClassifier
import pandas as pd

# Initialize and train the model
model = CatBoostClassifier(class_weights=list(class_weights.values()),
                           cat_features=categorical_features,
                           random_state=42,
                           verbose=0)
model.fit(X, y)

# Get feature importance
feature_importances = model.get_feature_importance(prettified=True)

# Convert to a DataFrame for easy manipulation
feature_importances_df = pd.DataFrame(feature_importances)
feature_importances_df.columns = ['Feature', 'Importance']

# Sort by importance
feature_importances_df = feature_importances_df.sort_values(by='Importance', ascending=False)

# Display the importance matrix
print(feature_importances_df)
```

```
# In[8]:
import matplotlib.pyplot as plt
# Plot feature importance
plt.figure(figsize=(12, 6))
plt.barh(feature_importances_df['Feature'], feature_importances_df['Importance'], color='skyblue')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Feature Importance')
plt.gca().invert_yaxis()  # Invert y-axis to display highest importance at the top
plt.show()


# ## Prioritize Transformation for Top 5 Features
# FunctionalAssessment, ADL, MMSE, MemoryComplaints, BehavioralProblems

# In[9]:
import matplotlib.pyplot as plt

# Define the top 5 features
top_5_features = ['FunctionalAssessment', 'ADL', 'MMSE', 'MemoryComplaints', 'BehavioralProblems']

# Set up a 3x2 grid for the plots
num_features = len(top_5_features)
num_cols = 2  # Number of columns
num_rows = 3  # Number of rows (enough to fit all 5 features)
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 10))
axes = axes.ravel()  # Flatten the axes for easier indexing

# Train the CatBoost model if not already trained
model = CatBoostClassifier(class_weights=list(class_weights.values()),
                           cat_features=categorical_features,
                           random_state=42,
                           verbose=0)
model.fit(X, y)

# Generate PDPs for the top 5 features
for i, feature in enumerate(top_5_features):
    PartialDependenceDisplay.from_estimator(
        model,
        X,
        [feature],
        kind="average",
        ax=axes[i]  # Assign each plot to a specific subplot
    )
    axes[i].set_title(f"PDP for {feature}")

for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])

# Adjust layout
plt.tight_layout()
plt.show()

# In[10]:
# Calculate correlation between FunctionalAssessment and ADL
correlation = X[['FunctionalAssessment', 'ADL']].corr()
print("Correlation Matrix:\n", correlation)

# # Baseline Model
# ## K-Fold Cross Validation
# In[11]:

from sklearn.model_selection import KFold
from sklearn.metrics import roc_auc_score, accuracy_score, precision_score, recall_score, f1_score

# Initialize CatBoost model
cat_model_baseline = CatBoostClassifier(class_weights=list(class_weights.values()),
                                        cat_features=categorical_features,
                                        random_state=42,
                                        verbose=0)
```

```
# Initialize K-Fold Cross-Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
# Lists to store metrics for each fold
auc_scores = []
accuracy_scores = []
precision_scores = []
recall_scores = []
f1_scores = []

# Perform K-Fold Cross-Validation
for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train the model
    cat_model_baseline.fit(X_train, y_train)

    # Make predictions
    y_pred = cat_model_baseline.predict(X_test)
    y_pred_proba = cat_model_baseline.predict_proba(X_test)[:, 1]

    # Calculate metrics
    auc = roc_auc_score(y_test, y_pred_proba)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='binary')
    recall = recall_score(y_test, y_pred, average='binary')
    f1 = f1_score(y_test, y_pred, average='binary')

    # Store metrics
    auc_scores.append(auc)
    accuracy_scores.append(accuracy)
    precision_scores.append(precision)
    recall_scores.append(recall)
    f1_scores.append(f1)

# Compute average performance metrics
mean_auc = sum(auc_scores) / len(auc_scores)
mean_accuracy = sum(accuracy_scores) / len(accuracy_scores)
mean_precision = sum(precision_scores) / len(precision_scores)
mean_recall = sum(recall_scores) / len(recall_scores)
mean_f1 = sum(f1_scores) / len(f1_scores)

# Print overall results
print("\nK-Fold Cross-Validation Results:")
print(f"Mean AUC: {mean_auc:.4f}")
print(f"Mean Accuracy: {mean_accuracy:.4f}")
print(f"Mean Precision: {mean_precision:.4f}")
print(f"Mean Recall: {mean_recall:.4f}")
print(f"Mean F1-Score: {mean_f1:.4f}")


# ## Log Model 1 - Baseline Model

# In[12]:
results_file = 'model_performance_tracking.csv'
if not os.path.exists(results_file):
    results_df = pd.DataFrame(columns=['Model', 'AUC', 'Accuracy', 'Precision', 'Recall', 'F1-Score',
                                       'Hyperparameters', 'Notes'])
else:
    # Load existing results
    results_df = pd.read_csv(results_file)

hyperparameters = cat_model_baseline.get_params()
new_entry = pd.DataFrame([{
    'Model': 'Model 1: Baseline Model',
    'AUC': mean_auc,
    'Accuracy': mean_accuracy,
    'Precision': mean_precision,
    'Recall': mean_recall,
```

```
    'F1-Score': mean_f1,
    'Hyperparameters': str(hyperparameters),  # Convert hyperparameters to string
    'Notes': '5-fold cross-validation with default hyperparameters'
}])
results_df = pd.concat([results_df, new_entry], ignore_index=True)
results_df.to_csv(results_file, index=False)
print("Model results logged successfully!")


# In[13]:
print(results_df)


# # Model 2
# ## Hyperparameter Tuning

# In[14]:
# # Define the parameter grid for tuning
# param_grid = {
#     'iterations': [500, 1000],
#     'learning_rate': [0.01, 0.05, 0.1],
#     'depth': [4, 6, 8],
#     'l2_leaf_reg': [1, 3, 5],
#     'bagging_temperature': [0.2, 0.5, 1],
# }

# # Wrap CatBoost in a compatible scikit-learn estimator for GridSearchCV
# cat_model_baseline = CatBoostClassifier(class_weights=list(class_weights.values()), cat_features=categorical_features, random_state=42, verbose=0)

# # Perform Grid Search
# grid_search = GridSearchCV(estimator=cat_model, param_grid=param_grid, cv=3, scoring='roc_auc', verbose=3, n_jobs=-1)
# grid_search.fit(X_train, y_train)

# # Output the best parameters and the corresponding score
# print(f"Best Hyperparameters: {grid_search.best_params_}")
# print(f"Best AUC Score from Grid Search: {grid_search.best_score_}")


# ## Retrain Model with the Resulting Hyperparameters from Hyperparameter Tuning

# In[15]:
from sklearn.model_selection import KFold
from sklearn.metrics import roc_auc_score, accuracy_score, precision_score, recall_score, f1_score

# Best hyperparameters from grid search
best_params = {
    'bagging_temperature': 0.2,
    'depth': 4,
    'iterations': 500,
    'l2_leaf_reg': 3,
    'learning_rate': 0.05,
    'random_state': 42,
    'verbose': 0
}

# Initialize CatBoost model with tuned hyperparameters
cat_model_baseline_tuned = CatBoostClassifier(class_weights=list(class_weights.values()),
                              cat_features=categorical_features,
                                   **best_params)


# Initialize K-Fold Cross-Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Lists to store metrics for each fold
auc_scores = []
accuracy_scores = []
precision_scores = []
recall_scores = []
f1_scores = []
```

```
# Perform K-Fold Cross-Validation
for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train the model
    cat_model_baseline_tuned.fit(X_train, y_train, cat_features=categorical_features)

    # Make predictions
    y_pred = cat_model_baseline_tuned.predict(X_test)
    y_pred_proba = cat_model_baseline_tuned.predict_proba(X_test)[:, 1]  # Probabilities for the positive class

    # Calculate metrics
    auc = roc_auc_score(y_test, y_pred_proba)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='binary')
    recall = recall_score(y_test, y_pred, average='binary')
    f1 = f1_score(y_test, y_pred, average='binary')

    # Store metrics
    auc_scores.append(auc)
    accuracy_scores.append(accuracy)
    precision_scores.append(precision)
    recall_scores.append(recall)
    f1_scores.append(f1)

# Compute average performance metrics
mean_auc = sum(auc_scores) / len(auc_scores)
mean_accuracy = sum(accuracy_scores) / len(accuracy_scores)
mean_precision = sum(precision_scores) / len(precision_scores)
mean_recall = sum(recall_scores) / len(recall_scores)
mean_f1 = sum(f1_scores) / len(f1_scores)

# Print overall results
print("\nK-Fold Cross-Validation Results for Tuned Model:")
print(f"Mean AUC: {mean_auc:.4f}")
print(f"Mean Accuracy: {mean_accuracy:.4f}")
print(f"Mean Precision: {mean_precision:.4f}")
print(f"Mean Recall: {mean_recall:.4f}")
print(f"Mean F1-Score: {mean_f1:.4f}")


# ## Log Model 2 - Baseline Model with Hyperparameter Tuning

# In[16]:
## Outputting to log file
import os

# File path to save and load results
results_file = 'model_performance_tracking.csv'

# Initialize results DataFrame if it doesn't exist
if not os.path.exists(results_file):
    results_df = pd.DataFrame(columns=['Model', 'AUC', 'Accuracy', 'Precision', 'Recall', 'F1-Score',
                                       'Hyperparameters', 'Notes'])
else:
    # Load existing results
    results_df = pd.read_csv(results_file)

# Log the tuned model's results
new_entry = pd.DataFrame([{
    'Model': 'Model 2: Baseline Model with Hyperparameter Tuning)',
    'AUC': mean_auc,
    'Accuracy': mean_accuracy,
    'Precision': mean_precision,
    'Recall': mean_recall,
    'F1-Score': mean_f1,
    'Hyperparameters': "{'bagging_temperature': 0.2, 'depth': 4, 'iterations': 500, 'l2_leaf_reg': 3, 'learning_rate': 0.05}",
    'Notes': '5-fold cross-validation with tuned hyperparameters'
}])
```

```
# Append the new entry to the existing results DataFrame
results_df = pd.concat([results_df, new_entry], ignore_index=True)

# Save the updated results DataFrame to the CSV file
results_df.to_csv(results_file, index=False)

print("Tuned model results logged successfully!")


# In[17]:
from IPython.display import display

# Display the DataFrame
display(results_df)


# ## Generate Predictions for Test Data - Baseline Model

# In[18]:
# Load the test dataset
test_data = pd.read_csv('test.csv')

# Extract PatientID for output
patient_ids = test_data['PatientID']
test_features = test_data.drop(columns=['PatientID', 'DoctorInCharge'], errors='ignore')

# Generate predictions for CatBoost
cat_test_predictions = cat_model_baseline.predict(test_features)
cat_output = pd.DataFrame({
    'PatientID': patient_ids,
    'Diagnosis': cat_test_predictions
})
cat_output.to_csv('predictions_catboost.csv', index=False)

print("Predictions saved to 'predictions_catboost.csv'")


# ## Generate Predictions for Test Data - Tuned Model

# In[19]:
# Load the test dataset
test_data = pd.read_csv('test.csv')

# Extract PatientID for output
patient_ids = test_data['PatientID']
test_features = test_data.drop(columns=['PatientID', 'DoctorInCharge'], errors='ignore')

# Generate predictions for CatBoost
cat_test_tuned_predictions = cat_model_baseline_tuned.predict(test_features)
cat_output = pd.DataFrame({
    'PatientID': patient_ids,
    'Diagnosis': cat_test_tuned_predictions
})
cat_output.to_csv('predictions_catboost_tuned.csv', index=False)

print("Predictions saved to 'predictions_catboost_tuned.csv'")


# ## Comparing Prediction Results

# In[20]:
# Create a DataFrame to compare predictions
comparison_df = pd.DataFrame({
    'PatientID': patient_ids,
    'Baseline_Diagnosis': cat_test_predictions,
    'Tuned_Diagnosis': cat_test_tuned_predictions
})

# Add a column to highlight differences
```

```
comparison_df['Differ'] = comparison_df['Baseline_Diagnosis'] != comparison_df['Tuned_Diagnosis']

# Print the differences summary
num_differences = comparison_df['Differ'].sum()
print(num_differences)


# ## Comparing Hyperparameters

# In[21]:
# Specify the hyperparameters to compare
hyperparameters_to_compare = [
    'bagging_temperature',
    'depth',
    'iterations',
    'l2_leaf_reg',
    'learning_rate',
    'random_state',
    'verbose'
]

# Extract the hyperparameters from the baseline and tuned models
baseline_params = cat_model_baseline.get_params()
tuned_params = cat_model_baseline_tuned.get_params()

# Extract default values for baseline parameters if not explicitly set
default_params = {
    'bagging_temperature': 1.0,
    'depth': 6,
    'iterations': 1000,
    'l2_leaf_reg': 3,
    'learning_rate': 0.03,  # Approximate default
    'random_state': None,   # Default is None if not set
    'verbose': 0            # Silent mode enabled
}

# Replace missing baseline parameters with their defaults
baseline_values = [
    baseline_params.get(param, default_params.get(param, "N/A"))
    for param in hyperparameters_to_compare
]

# Create a DataFrame to compare the selected hyperparameters
hyperparameter_comparison = pd.DataFrame({
    'Hyperparameter': hyperparameters_to_compare,
    'Baseline_Model': baseline_values,
    'Tuned_Model': [tuned_params.get(param, "N/A") for param in hyperparameters_to_compare],
    'Differ': [
        baseline_params.get(param, default_params.get(param, "N/A")) != tuned_params.get(param, "N/A")
        for param in hyperparameters_to_compare
    ]  # Highlight differences
})

# Save the comparison to a CSV file
# hyperparameter_comparison.to_csv('hyperparameter_comparison_selected.csv', index=False)

# Display the comparison
print("Selected Hyperparameter Comparison:")
print(hyperparameter_comparison.to_string(index=False))

print("\nComparison saved to 'hyperparameter_comparison_selected.csv'")


# # Model 3 - Catboost with Feature Selection & Hyperparameter Tuning
# ## Feature Selection via Catboost Feature Elimination

# In[22]:
# # Import necessary libraries
# from sklearn.model_selection import cross_val_score, KFold
# from catboost import CatBoostClassifier
```

```
# # Define the fixed list of categorical features
# categorical_features = [
#     'Gender', 'Ethnicity', 'EducationLevel', 'Smoking',
#     'FamilyHistoryAlzheimers', 'CardiovascularDisease', 'Diabetes',
#     'Depression', 'HeadInjury', 'Hypertension', 'MemoryComplaints',
#     'BehavioralProblems', 'Confusion', 'Disorientation',
#     'PersonalityChanges', 'DifficultyCompletingTasks', 'Forgetfulness'
# ]

# # Preprocess categorical features
# for col in categorical_features:
#     if col in X.columns:
#         X[col] = X[col].astype('category')  # Ensure correct type
#         if 'Unknown' not in X[col].cat.categories:  # Add 'Unknown' as a valid category
#             X[col] = X[col].cat.add_categories('Unknown')
#         X[col] = X[col].fillna('Unknown')  # Replace NaN with 'Unknown'

# # Define CatBoost model parameters
# estimator_params = {
#     'random_state': 42,
#     'verbose': 0,
#     'iterations': 500,
#     'depth': 6,
#     'learning_rate': 0.05,
#     'eval_metric': 'AUC'
# }

# # Define the manual RFECV function
# def manual_rfecv(X, y, categorical_features, estimator_params, min_features_to_select=1):
#     features = X.columns.tolist()
#     best_auc = 0
#     best_features = features.copy()

#     while len(features) > min_features_to_select:
#         print(f"Evaluating {len(features)} features...")

#         # Train model and calculate cross-validated AUC
#         model = CatBoostClassifier(cat_features=[features.index(col) for col in categorical_features if col in features],
#                                    **estimator_params)
#         scores = cross_val_score(
#             model,
#             X[features],
#             y,
#             scoring='roc_auc',
#             cv=KFold(n_splits=5, shuffle=True, random_state=42),
#             error_score='raise'
#         )
#         mean_auc = scores.mean()

#         print(f"Mean AUC with {len(features)} features: {mean_auc:.4f}")

#         # Check if this set of features is the best
#         if mean_auc > best_auc:
#             best_auc = mean_auc
#             best_features = features.copy()

#         # Identify the least important feature and remove it
#         model.fit(X[features], y)
#         feature_importances = model.get_feature_importance(type='FeatureImportance')
#         least_important_feature = features[np.argmin(feature_importances)]
#         print(f"Removing least important feature: {least_important_feature}")
#         features.remove(least_important_feature)

#     return best_features, best_auc

# # Perform Recursive Feature Elimination
# best_features, best_auc = manual_rfecv(X, y, categorical_features, estimator_params)

# # Output results
# print(f"\nBest Features ({len(best_features)}): {best_features}")
# print(f"Best Mean AUC: {best_auc:.4f}")
```

```
# ## Train Model with Selected Features
# Selected Features: AlcoholConsumption, CholesterolTotal, MMSE, FunctionalAssessment, MemoryComplaints, BehavioralProblems, ADL

# In[23]:
best_features = ['AlcoholConsumption', 'SleepQuality', 'SystolicBP', 'CholesterolTotal', 'CholesterolHDL',
              'CholesterolTriglycerides', 'MMSE', 'FunctionalAssessment', 'MemoryComplaints',
              'BehavioralProblems', 'ADL']

estimator_params = {
    'random_state': 42,
    'verbose': 0,
    'iterations': 500,
    'depth': 6,
    'learning_rate': 0.05
}

X_best = X[best_features]

# Train and validate the final model
cat_cfecv = CatBoostClassifier(class_weights=list(class_weights.values()),
                           cat_features=categorical_features,
                           **estimator_params)


# ## Hyperparameter Tuning

# In[24]:
# param_grid = {
#     'iterations': [500, 1000, 1500],  # Number of boosting iterations
#     'learning_rate': [0.01, 0.05, 0.1],  # Learning rate
#     'depth': [4, 6, 8],  # Maximum tree depth
#     'l2_leaf_reg': [1, 3, 5],  # L2 regularization
#     'bagging_temperature': [0.2, 0.5, 1]  # Bagging randomness
# }

# from sklearn.model_selection import GridSearchCV

# # Initialize the CatBoost model
# cat_cfecv = CatBoostClassifier(class_weights=list(class_weights.values()),
#                            cat_features=['MemoryComplaints', 'BehavioralProblems'],
#                            random_state=42,
#                            verbose=0)

# # Perform Grid Search on the reduced feature set
# grid_search = GridSearchCV(
#     estimator=cat_cfecv,
#     param_grid=param_grid,
#     cv=5,  # 5-fold cross-validation
#     scoring='roc_auc',  # Optimize for ROC AUC
#     verbose=3,
#     n_jobs=-1
# )

# # Fit Grid Search to the reduced feature set
# grid_search.fit(X_best, y)

# # Best parameters and score
# best_params = grid_search.best_params_
# best_score = grid_search.best_score_

# print(f"Best Hyperparameters: {best_params}")
# print(f"Best ROC AUC Score: {best_score:.4f}")


# In[25]:
from sklearn.metrics import roc_auc_score, accuracy_score, precision_score, recall_score, f1_score

# Specify best hyperparameters
best_params = {
```

```
    'bagging_temperature': 0.2,
    'depth': 4,
    'iterations': 500,
    'l2_leaf_reg': 5,
    'learning_rate': 0.01,
    'random_state': 42,
    'verbose': 0,
}

# Initialize and train the final CatBoost model
cat_cfecv_tuned = CatBoostClassifier(class_weights=list(class_weights.values()),
                          cat_features=['MemoryComplaints', 'BehavioralProblems'],
                              **best_params)
cat_cfecv_tuned.fit(X_best, y)

# Generate predictions
y_pred = cat_cfecv_tuned.predict(X_best)
y_pred_proba = cat_cfecv_tuned.predict_proba(X_best)[:, 1]  # Probabilities for the positive class

# Calculate performance metrics
mean_auc = roc_auc_score(y, y_pred_proba)
mean_accuracy = accuracy_score(y, y_pred)
mean_precision = precision_score(y, y_pred, zero_division=0)
mean_recall = recall_score(y, y_pred, zero_division=0)
mean_f1 = f1_score(y, y_pred, zero_division=0)

# Print evaluation results
print(f"cat_cfecv_tuned Metrics:")
print(f"ROC AUC: {mean_auc:.4f}")
print(f"Accuracy: {mean_accuracy:.4f}")
print(f"Precision: {mean_precision:.4f}")
print(f"Recall: {mean_recall:.4f}")
print(f"F1-Score: {mean_f1:.4f}")


# ## Log results for Model 3 - Feature Selection using CFECV

# In[26]:
# Log the model's results
hyperparameters = cat_cfecv_tuned.get_params()
new_entry = pd.DataFrame([{
    'Model': 'Model 3: Catboost with Feature Selection',
    'AUC': mean_auc,
    'Accuracy': mean_accuracy,
    'Precision': mean_precision,
    'Recall': mean_recall,
    'F1-Score': mean_f1,
    'Hyperparameters': str(hyperparameters),  # Convert hyperparameters to string
    'Notes': 'Trained with selected features and tuned hyperparameters'
}])

# Append the new entry to the existing results DataFrame
results_df = pd.concat([results_df, new_entry], ignore_index=True)

# Save the updated results DataFrame to the CSV file
results_df.to_csv(results_file, index=False)

print("Model results logged successfully!")


# ## Generate Predictions for Test Dataset

# In[27]:
# Load the test dataset
test_data = pd.read_csv('test.csv')

# Extract PatientID for output
patient_ids = test_data['PatientID']

# Subset test data to only include optimal features
test_features = test_data[best_features]
```

```
cat_cfecv_tuned_predictions = cat_cfecv_tuned.predict(test_features)
cat_output = pd.DataFrame({
    'PatientID': patient_ids,
    'Diagnosis': cat_cfecv_tuned_predictions
})
cat_output.to_csv('predictions_catboost_cfecv_tuned.csv', index=False)

print("Predictions saved to 'predictions_catboost_cfecv_tuned.csv'")


# # Model 4.1, 4.2, 4.3 - Feature Engineering (Threshold Effects)

# ## Identify the optimal threshold for FunctionalAssessment, ADL and MMSE

# In[28]:
from sklearn.inspection import PartialDependenceDisplay

# Function to calculate optimal threshold from PDP
def calculate_optimal_threshold(model, X, feature_name):
    # Generate Partial Dependence Plot data
    pdp = PartialDependenceDisplay.from_estimator(
        model, X, [feature_name], kind="average"
    )

    # Extract the PDP data
    feature_pdp = pdp.lines_[0][0].get_ydata()
    feature_values = pdp.lines_[0][0].get_xdata()

    # Calculate differences (gradient of the curve)
    differences = np.diff(feature_pdp)

    # Find the index of the largest change
    threshold_index = np.argmax(np.abs(differences))
    optimal_threshold = feature_values[threshold_index]

    return optimal_threshold

# Train the model
model = CatBoostClassifier(class_weights=list(class_weights.values()),
                                    cat_features=categorical_features,
                           random_state=42, verbose=0)
model.fit(X, y)

# Calculate thresholds for each feature
optimal_threshold_fa = calculate_optimal_threshold(model, X, "FunctionalAssessment")
optimal_threshold_adl = calculate_optimal_threshold(model, X, "ADL")
optimal_threshold_mmse = calculate_optimal_threshold(model, X, "MMSE")

print(f"Optimal threshold for FunctionalAssessment: {optimal_threshold_fa:.2f}")
print(f"Optimal threshold for ADL: {optimal_threshold_adl:.2f}")
print(f"Optimal threshold for MMSE: {optimal_threshold_mmse:.2f}")


# ## Model 4.1 - replace the original features with the transformed features

# In[29]:
from sklearn.model_selection import KFold
from sklearn.metrics import roc_auc_score, accuracy_score, precision_score, recall_score, f1_score
from catboost import CatBoostClassifier

# Replace features with binary threshold transformations
X_transformed = X.copy()
X_transformed['FunctionalAssessment'] = (X_transformed['FunctionalAssessment'] > 4.93).astype(int)
X_transformed['ADL'] = (X_transformed['ADL'] > 4.96).astype(int)
X_transformed['MMSE'] = (X_transformed['MMSE'] > 23.82).astype(int)

# Initialize CatBoost model
cat_model_41 = CatBoostClassifier(class_weights=list(class_weights.values()),
                                  cat_features=categorical_features,
                             random_state=42,
```

```
                        verbose=0)

# Initialize K-Fold Cross-Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Lists to store metrics for each fold
auc_scores = []
accuracy_scores = []
precision_scores = []
recall_scores = []
f1_scores = []

# Perform K-Fold Cross-Validation
for fold, (train_index, test_index) in enumerate(kf.split(X_transformed)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X_transformed.iloc[train_index], X_transformed.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train the model
    cat_model_41.fit(X_train, y_train)

    # Make predictions
    y_pred = cat_model_41.predict(X_test)
    y_pred_proba = cat_model_41.predict_proba(X_test)[:, 1]  # Probabilities for the positive class

    # Calculate metrics
    auc = roc_auc_score(y_test, y_pred_proba)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='binary')
    recall = recall_score(y_test, y_pred, average='binary')
    f1 = f1_score(y_test, y_pred, average='binary')

    # Store metrics
    auc_scores.append(auc)
    accuracy_scores.append(accuracy)
    precision_scores.append(precision)
    recall_scores.append(recall)
    f1_scores.append(f1)

# Compute average performance metrics
mean_auc = sum(auc_scores) / len(auc_scores)
mean_accuracy = sum(accuracy_scores) / len(accuracy_scores)
mean_precision = sum(precision_scores) / len(precision_scores)
mean_recall = sum(recall_scores) / len(recall_scores)
mean_f1 = sum(f1_scores) / len(f1_scores)

# Print overall results
print("\nK-Fold Cross-Validation Results with Transformed Features:")
print(f"Mean AUC: {mean_auc:.4f}")
print(f"Mean Accuracy: {mean_accuracy:.4f}")
print(f"Mean Precision: {mean_precision:.4f}")
print(f"Mean Recall: {mean_recall:.4f}")
print(f"Mean F1-Score: {mean_f1:.4f}")


# ## Log Model 1 - replaced the original features with their corresponding threshold dummies

# In[30]:
results_file = 'model_performance_tracking.csv'
if not os.path.exists(results_file):
    results_df = pd.DataFrame(columns=['Model', 'AUC', 'Accuracy', 'Precision', 'Recall', 'F1-Score',
                                       'Hyperparameters', 'Notes'])
else:
    # Load existing results
    results_df = pd.read_csv(results_file)

new_entry = pd.DataFrame([{
    'Model': 'Model 4.1: Replaced the original features with their corresponding threshold dummies',
    'AUC': mean_auc,
    'Accuracy': mean_accuracy,
    'Precision': mean_precision,
```

```
    'Recall': mean_recall,
    'F1-Score': mean_f1,
    'Hyperparameters': str(best_params),  # Convert hyperparameters to string
    'Notes': 'Combined features (original + binary thresholds)'
}])
results_df = pd.concat([results_df, new_entry], ignore_index=True)

results_df.to_csv(results_file, index=False)

print("Model results logged successfully!")


# ## Model 4.2 - 3 transformed features + 2 untransformed high importance features
# Investigate the model that combine the transformed the and untransformed top features

# In[31]:
from sklearn.model_selection import KFold
from sklearn.metrics import roc_auc_score, accuracy_score, precision_score, recall_score, f1_score

# Define the feature set with binary transformed and untransformed features
threshold_features = ['FunctionalAssessment_binary', 'ADL_binary', 'MMSE_binary', 'MemoryComplaints',
                     'BehavioralProblems']

# Create a new dataset with the transformed features
X_transformed_subset = X.copy()
X_transformed_subset['FunctionalAssessment_binary'] = (X_transformed_subset['FunctionalAssessment'] > 4.93).astype(int)
X_transformed_subset['ADL_binary'] = (X_transformed_subset['ADL'] > 4.96).astype(int)
X_transformed_subset['MMSE_binary'] = (X_transformed_subset['MMSE'] > 23.82).astype(int)
X_transformed_subset = X_transformed_subset[threshold_features]  # Select the specified features

# Initialize CatBoost model
cat_model_42 = CatBoostClassifier(class_weights=list(class_weights.values()),
                                  cat_features=threshold_features,
                                  random_state=42,
                                  verbose=0)

# Initialize K-Fold Cross-Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Lists to store metrics for each fold
auc_scores = []
accuracy_scores = []
precision_scores = []
recall_scores = []
f1_scores = []

# Perform K-Fold Cross-Validation
for fold, (train_index, test_index) in enumerate(kf.split(X_transformed_subset)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X_transformed_subset.iloc[train_index], X_transformed_subset.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train the model
    cat_model_42.fit(X_train, y_train)

    # Make predictions
    y_pred = cat_model_42.predict(X_test)
    y_pred_proba = cat_model_42.predict_proba(X_test)[:, 1]  # Probabilities for the positive class

    # Calculate metrics
    auc = roc_auc_score(y_test, y_pred_proba)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='binary')
    recall = recall_score(y_test, y_pred, average='binary')
    f1 = f1_score(y_test, y_pred, average='binary')

    # Store metrics
    auc_scores.append(auc)
    accuracy_scores.append(accuracy)
    precision_scores.append(precision)
    recall_scores.append(recall)
```

```
    f1_scores.append(f1)

# Compute average performance metrics
mean_auc = sum(auc_scores) / len(auc_scores)
mean_accuracy = sum(accuracy_scores) / len(accuracy_scores)
mean_precision = sum(precision_scores) / len(precision_scores)
mean_recall = sum(recall_scores) / len(recall_scores)
mean_f1 = sum(f1_scores) / len(f1_scores)

# Print overall results
print("\nK-Fold Cross-Validation Results with Transformed and Untransformed Features:")
print(f"Mean AUC: {mean_auc:.4f}")
print(f"Mean Accuracy: {mean_accuracy:.4f}")
print(f"Mean Precision: {mean_precision:.4f}")
print(f"Mean Recall: {mean_recall:.4f}")
print(f"Mean F1-Score: {mean_f1:.4f}")


# ## Log Model 4.2

# In[32]:
results_file = 'model_performance_tracking.csv'
if not os.path.exists(results_file):
    results_df = pd.DataFrame(columns=['Model', 'AUC', 'Accuracy', 'Precision', 'Recall', 'F1-Score',
                                       'Hyperparameters', 'Notes'])
else:
    # Load existing results
    results_df = pd.read_csv(results_file)

new_entry = pd.DataFrame([{
    'Model': 'Model 4.2: Threshold Dummies ',
    'AUC': mean_auc,
    'Accuracy': mean_accuracy,
    'Precision': mean_precision,
    'Recall': mean_recall,
    'F1-Score': mean_f1,
    'Hyperparameters': str(best_params),  # Convert hyperparameters to string
    'Notes': 'Combined features (original + binary thresholds)'
}])

results_df = pd.concat([results_df, new_entry], ignore_index=True)
results_df.to_csv(results_file, index=False)
print("Model results logged successfully!")


# ## Model 4.3 - 3 transformed features + their original and remaining important features identified by in Feature Selection

# In[33]:
# Define the combined feature set (original + transformed features)
combined_features = ['FunctionalAssessment', 'ADL', 'MMSE',
                     'FunctionalAssessment_binary', 'ADL_binary', 'MMSE_binary',
                     'MemoryComplaints', 'BehavioralProblems', 'AlcoholConsumption', 'CholesterolTotal',
                    'SleepQuality', 'SystolicBP', 'CholesterolHDL', 'CholesterolTriglycerides'
                    ]

cat_features_in_combined_features = ['MemoryComplaints','BehavioralProblems', 'FunctionalAssessment_binary',
                                     'ADL_binary', 'MMSE_binary'
                                    ]

X_combined = X.copy()
X_combined['FunctionalAssessment_binary'] = (X_combined['FunctionalAssessment'] > 4.93).astype(int)
X_combined['ADL_binary'] = (X_combined['ADL'] > 4.96).astype(int)
X_combined['MMSE_binary'] = (X_combined['MMSE'] > 23.82).astype(int)

X_combined = X_combined[combined_features]

best_params = {
    'bagging_temperature': 0.2,
    'depth': 4,
    'iterations': 500,
    'l2_leaf_reg': 5,
```

```
    'learning_rate': 0.01,
    'random_state': 42,
    'verbose': 0,
}

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
precision_scores = []
recall_scores = []
f1_scores = []

# Perform K-Fold Cross-Validation
for fold, (train_index, test_index) in enumerate(kf.split(X_combined)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X_combined.iloc[train_index], X_combined.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    cat_model_43 = CatBoostClassifier(class_weights=list(class_weights.values()),
                              cat_features=cat_features_in_combined_features,
                              **best_params)
    # Train the model
    cat_model_43.fit(X_train, y_train)

    # Make predictions
    y_pred = cat_model_43.predict(X_test)
    y_pred_proba = cat_model_43.predict_proba(X_test)[:, 1]  # Probabilities for the positive class

    # Calculate metrics
    auc = roc_auc_score(y_test, y_pred_proba)
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='binary')
    recall = recall_score(y_test, y_pred, average='binary')
    f1 = f1_score(y_test, y_pred, average='binary')

    # Store metrics
    auc_scores.append(auc)
    accuracy_scores.append(accuracy)
    precision_scores.append(precision)
    recall_scores.append(recall)
    f1_scores.append(f1)

# Compute average performance metrics
mean_auc = sum(auc_scores) / len(auc_scores)
mean_accuracy = sum(accuracy_scores) / len(accuracy_scores)
mean_precision = sum(precision_scores) / len(precision_scores)
mean_recall = sum(recall_scores) / len(recall_scores)
mean_f1 = sum(f1_scores) / len(f1_scores)

# Print overall results
print("\nK-Fold Cross-Validation Results with Combined Features (Original + Transformed):")
print(f"Mean AUC: {mean_auc:.4f}")
print(f"Mean Accuracy: {mean_accuracy:.4f}")
print(f"Mean Precision: {mean_precision:.4f}")
print(f"Mean Recall: {mean_recall:.4f}")
print(f"Mean F1-Score: {mean_f1:.4f}")


# In[34]:
results_file = 'model_performance_tracking.csv'

if not os.path.exists(results_file):
    results_df = pd.DataFrame(columns=['Model', 'AUC', 'Accuracy', 'Precision', 'Recall', 'F1-Score',
                                'Hyperparameters', 'Notes'])
else:
    # Load existing results
    results_df = pd.read_csv(results_file)

new_entry = pd.DataFrame([{
    'Model': 'Model 4.3: Catboost with Feature Engineering (Threshold Effects)',
```

```
    'AUC': mean_auc,
    'Accuracy': mean_accuracy,
    'Precision': mean_precision,
    'Recall': mean_recall,
    'F1-Score': mean_f1,
    'Hyperparameters': str(best_params),  # Convert hyperparameters to string
    'Notes': 'Combined features (original + binary thresholds)'
}])

results_df = pd.concat([results_df, new_entry], ignore_index=True)
results_df.to_csv(results_file, index=False)
print("Model results logged successfully!")
print(results_df)

# ## Make Predictions on Test Dataset

# In[36]:
test_data = pd.read_csv('test.csv')  # Replace with the actual file path

# Create binary threshold features for the test dataset
test_data['FunctionalAssessment_binary'] = (test_data['FunctionalAssessment'] > 4.93).astype(int)
test_data['ADL_binary'] = (test_data['ADL'] > 4.96).astype(int)
test_data['MMSE_binary'] = (test_data['MMSE'] > 23.82).astype(int)

# Define the combined feature set (original + transformed features)
combined_features = ['FunctionalAssessment', 'ADL', 'MMSE',
                     'FunctionalAssessment_binary', 'ADL_binary', 'MMSE_binary',
                     'MemoryComplaints', 'BehavioralProblems', 'AlcoholConsumption', 'CholesterolTotal',
                    'SleepQuality', 'SystolicBP', 'CholesterolHDL', 'CholesterolTriglycerides'
                    ]

cat_features_in_combined_features = ['MemoryComplaints','BehavioralProblems', 'FunctionalAssessment_binary',
                                     'ADL_binary', 'MMSE_binary'
                                    ]

# Subset the test dataset to include only the combined features
X_test_combined = test_data[combined_features]

# Train the model on the entire training dataset
cat_model_43 = CatBoostClassifier(class_weights=list(class_weights.values()),
                                  cat_features=cat_features_in_combined_features,
                                **best_params)
cat_model_43.fit(X_combined, y)

# Make predictions
y_test_predictions = cat_model_43.predict(X_test_combined)
y_test_predictions_proba = cat_model_43.predict_proba(X_test_combined)[:, 1]  # Probabilities for the positive class

# Prepare the output DataFrame
output = pd.DataFrame({
    'PatientID': test_data['PatientID'],  # Replace with the actual ID column name in the test dataset
    'Diagnosis': y_test_predictions
})

# Save the predictions to a CSV file
output_file = 'test_predictions_catboost.csv'
output.to_csv(output_file, index=False)

print(f"Predictions saved to '{output_file}'")


# # Model 5 Experimenting Ensemble Methods with XGBoost and LightGBM
# ## Model 5.1 Weighted Average Method
# ## GridSearch to find Optimal Weights

# In[37]:
# from itertools import product

# # Define possible weight values
# weight_values = np.arange(0.1, 1.1, 0.1)  # Weights from 0.1 to 1.0 in 0.1 increments
# # K-Fold Cross-Validation
```

```
# kf = KFold(n_splits=5, shuffle=True, random_state=42)

# # Function to evaluate ensemble performance for a given set of weights
# def evaluate_weights(weights, kf, X_best, X_xg_best, y):
#     auc_scores = []

#     for train_index, test_index in kf.split(X_best):
#         # Split data into train and test sets for the current fold
#         X_train_best, X_test_best = X_best.iloc[train_index], X_best.iloc[test_index]
#         y_train, y_test = y.iloc[train_index], y.iloc[test_index]

#         # Train individual models
#         cat_cfecv_tuned.fit(X_train_best, y_train)
#         xgb_model.fit(X_xg_best.iloc[train_index], y_train)
#         lgb_model.fit(X_train_best, y_train)

#         # Generate predictions
#         y_proba_cat = cat_cfecv_tuned.predict_proba(X_test_best)[:, 1]
#         y_proba_xgb = xgb_model.predict_proba(X_xg_best.iloc[test_index])[:, 1]
#         y_proba_lgb = lgb_model.predict_proba(X_test_best)[:, 1]

#         # Weighted ensemble probabilities
#         y_ensemble_proba = (
#             weights[0] * y_proba_cat +
#             weights[1] * y_proba_xgb +
#             weights[2] * y_proba_lgb
#         )

#         # Evaluate AUC
#         auc_scores.append(roc_auc_score(y_test, y_ensemble_proba))

#     return np.mean(auc_scores)

# # Grid search over weight combinations
# best_auc = 0
# best_weights = None

# for weights in product(weight_values, repeat=3):
#     if np.isclose(sum(weights), 1):  # Ensure weights sum to 1
#         mean_auc = evaluate_weights(weights, kf, X_best, X_xg_best, y)
#         if mean_auc > best_auc:
#             best_auc = mean_auc
#             best_weights = weights

# print(f"Best Weights: {best_weights}, Best AUC: {best_auc:.4f}")


# In[38]:
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from sklearn.ensemble import VotingClassifier

X_xg = X.copy()

# Prepare aligned datasets for all models
X_xg_best = X_xg[X_best.columns]  # Subset X_xg to match X_best
X_best = X[best_features]         # Already prepared for CatBoost and LightGBM

# Initialize classifiers
cat_cfecv_tuned = CatBoostClassifier(
    random_state=42, verbose=0, iterations=500, depth=6, learning_rate=0.05,
    cat_features=[X_best.columns.get_loc(col) for col in categorical_features if col in X_best.columns]
)

xgb_model = XGBClassifier(
    random_state=42, use_label_encoder=False, eval_metric='logloss',
    n_estimators=500, max_depth=6, learning_rate=0.05, enable_categorical=True
)

lgb_model = LGBMClassifier(
```

```
    random_state=42, n_estimators=500, max_depth=6, learning_rate=0.05,
    categorical_feature=[X_best.columns.get_loc(col) for col in categorical_features if col in X_best.columns],
    verbose=-1
)

# best weights for the ensemble from before
weights = {'catboost': 0.6, 'xgboost': 0.3, 'lightgbm': 0.1}

# Initialize K-Fold Cross-Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Lists to store metrics for Weighted Average Ensemble
auc_scores, accuracy_scores, precision_scores, recall_scores, f1_scores = [], [], [], [], []

# Perform K-Fold Cross-Validation
for fold, (train_index, test_index) in enumerate(kf.split(X_best)):
    print(f"Fold {fold + 1}")

    # Split data into train and test sets for the current fold
    X_train_best, X_test_best = X_best.iloc[train_index], X_best.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train individual models
    cat_cfecv_tuned.fit(X_train_best, y_train)
    xgb_model.fit(X_xg_best.iloc[train_index], y_train)
    lgb_model.fit(X_train_best, y_train)

    # Generate predictions
    y_proba_cat = cat_cfecv_tuned.predict_proba(X_test_best)[:, 1]
    y_proba_xgb = xgb_model.predict_proba(X_xg_best.iloc[test_index])[:, 1]
    y_proba_lgb = lgb_model.predict_proba(X_test_best)[:, 1]

    # Weighted ensemble probabilities
    y_ensemble_proba = (
        weights['catboost'] * y_proba_cat +
        weights['xgboost'] * y_proba_xgb +
        weights['lightgbm'] * y_proba_lgb
    )
    y_ensemble_pred = (y_ensemble_proba >= 0.5).astype(int)

    # Evaluate performance
    auc_scores.append(roc_auc_score(y_test, y_ensemble_proba))
    accuracy_scores.append(accuracy_score(y_test, y_ensemble_pred))
    precision_scores.append(precision_score(y_test, y_ensemble_pred, zero_division=0))
    recall_scores.append(recall_score(y_test, y_ensemble_pred, zero_division=0))
    f1_scores.append(f1_score(y_test, y_ensemble_pred, zero_division=0))

weighted_auc = np.mean(auc_scores)
weighted_accuracy = np.mean(accuracy_scores)
weighted_precision = np.mean(precision_scores)
weighted_recall = np.mean(recall_scores)
weighted_f1 = np.mean(f1_scores)

print("\nWeighted Average Ensemble Cross-Validation Results:")
print(f"Mean AUC: {weighted_auc:.4f}")
print(f"Mean Accuracy: {weighted_accuracy:.4f}")
print(f"Mean Precision: {weighted_precision:.4f}")
print(f"Mean Recall: {weighted_recall:.4f}")
print(f"Mean F1-Score: {weighted_f1:.4f}")

results = []
results.append({
    "Model": "Weighted Average Ensemble",
    "Mean AUC": weighted_auc,
    "Mean Accuracy": weighted_accuracy,
    "Mean Precision": weighted_precision,
    "Mean Recall": weighted_recall,
    "Mean F1-Score": weighted_f1
})

# Create and print results DataFrame
```

```
results_df = pd.DataFrame(results)
print("\nResults Summary:")
print(results_df)


# ## Model 5.2 Stacking Method

# In[39]:
# Reset indices of feature subsets to align with KFold indices
X_best = X_best.reset_index(drop=True)
X_xg_best = X_xg_best.reset_index(drop=True)
y = y.reset_index(drop=True)

# Initialize K-Fold Cross-Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Lists to store metrics
auc_scores, accuracy_scores, precision_scores, recall_scores, f1_scores = [], [], [], [], []

# Perform K-Fold Cross-Validation
for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")

    # Map indices to respective feature subsets
    X_train_best, X_test_best = X_best.iloc[train_index], X_best.iloc[test_index]
    X_train_xg_best, X_test_xg_best = X_xg_best.iloc[train_index], X_xg_best.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Generate out-of-fold predictions for stacking
    oof_stacked_features = pd.DataFrame(index=train_index, columns=['catboost', 'xgboost', 'lightgbm'])
    test_stacked_features = pd.DataFrame(index=test_index, columns=['catboost', 'xgboost', 'lightgbm'])

    # Train individual models and generate predictions
    for model, model_name, model_X_train, model_X_test in [
        (cat_cfecv_tuned, 'catboost', X_train_best, X_test_best),
        (xgb_model, 'xgboost', X_train_xg_best, X_test_xg_best),
        (lgb_model, 'lightgbm', X_train_best, X_test_best)
    ]:
        # Train model
        model.fit(model_X_train, y_train)

        # Out-of-fold predictions for training meta-model
        oof_stacked_features[model_name] = model.predict_proba(model_X_train)[:, 1]

        # Predictions for the test set in this fold
        test_stacked_features[model_name] = model.predict_proba(model_X_test)[:, 1]

    # Train meta-model on the out-of-fold predictions
    meta_model = LogisticRegression(random_state=42)
    meta_model.fit(oof_stacked_features, y_train)

    # Predict on the test set for this fold
    y_meta_proba = meta_model.predict_proba(test_stacked_features)[:, 1]
    y_meta_pred = (y_meta_proba >= 0.5).astype(int)

    # Evaluate performance
    auc_scores.append(roc_auc_score(y_test, y_meta_proba))
    accuracy_scores.append(accuracy_score(y_test, y_meta_pred))
    precision_scores.append(precision_score(y_test, y_meta_pred, zero_division=0))
    recall_scores.append(recall_score(y_test, y_meta_pred, zero_division=0))
    f1_scores.append(f1_score(y_test, y_meta_pred, zero_division=0))

# Calculate average performance metrics
stacking_auc = np.mean(auc_scores)
stacking_accuracy = np.mean(accuracy_scores)
stacking_precision = np.mean(precision_scores)
stacking_recall = np.mean(recall_scores)
stacking_f1 = np.mean(f1_scores)

# Print overall results
print("\nStacking Ensemble Cross-Validation Results:")
```

```
print(f"Mean AUC: {stacking_auc:.4f}")
print(f"Mean Accuracy: {stacking_accuracy:.4f}")
print(f"Mean Precision: {stacking_precision:.4f}")
print(f"Mean Recall: {stacking_recall:.4f}")
print(f"Mean F1-Score: {stacking_f1:.4f}")


# ## Model 5.3 Voting Method

# In[40]:
# Initialize Voting Classifier with adjusted feature sets
voting_model = VotingClassifier(estimators=[
    ('catboost', cat_cfecv_tuned),
    ('xgboost', xgb_model),
    ('lightgbm', lgb_model)
], voting='soft')

kf = KFold(n_splits=5, shuffle=True, random_state=42)
auc_scores, accuracy_scores, precision_scores, recall_scores, f1_scores = [], [], [], [], []

# Perform K-Fold Cross-Validation
for fold, (train_index, test_index) in enumerate(kf.split(X_best)):
    print(f"Fold {fold + 1}")

    # Split data into train and test sets for the current fold
    X_train_best, X_test_best = X_best.iloc[train_index], X_best.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    voting_model.fit(X_train_best, y_train)

    y_pred = voting_model.predict(X_test_best)
    y_proba = voting_model.predict_proba(X_test_best)[:, 1]

    auc_scores.append(roc_auc_score(y_test, y_proba))
    accuracy_scores.append(accuracy_score(y_test, y_pred))
    precision_scores.append(precision_score(y_test, y_pred, zero_division=0))
    recall_scores.append(recall_score(y_test, y_pred, zero_division=0))
    f1_scores.append(f1_score(y_test, y_pred, zero_division=0))

voting_auc = sum(auc_scores) / len(auc_scores)
voting_accuracy = sum(accuracy_scores) / len(accuracy_scores)
voting_precision = sum(precision_scores) / len(precision_scores)
voting_recall = sum(recall_scores) / len(recall_scores)
voting_f1 = sum(f1_scores) / len(f1_scores)

print("\nK-Fold Cross-Validation Results for Voting Ensemble:")
print(f"Mean AUC: {voting_auc:.4f}")
print(f"Mean Accuracy: {voting_accuracy:.4f}")
print(f"Mean Precision: {voting_precision:.4f}")
print(f"Mean Recall: {voting_recall:.4f}")
print(f"Mean F1-Score: {voting_f1:.4f}")


# ## Log Model 5.1, 5.2, and 5.3

# In[41]:
results_file = 'model_performance_tracking.csv'
if not os.path.exists(results_file):
    results_df = pd.DataFrame(columns=['Model', 'AUC', 'Accuracy', 'Precision', 'Recall', 'F1-Score',
                                       'Hyperparameters', 'Notes'])
else:
    # Load existing results
    results_df = pd.read_csv(results_file)

def log_results(name, auc, accuracy, precision, recall, f1, hyperparameters, notes):
    """Log results of a model or ensemble method."""
    new_entry = pd.DataFrame([{
        'Model': name,
        'AUC': auc,
        'Accuracy': accuracy,
        'Precision': precision,
```

```
        'Recall': recall,
        'F1-Score': f1,
        'Hyperparameters': hyperparameters,
        'Notes': notes
    }])
    global results_df
    results_df = pd.concat([results_df, new_entry], ignore_index=True)

# Example for Weighted Average Ensemble
log_results(
    name="Model 5.1: Weighted Average Ensemble",
    auc=weighted_auc,  # Replace with calculated metrics
    accuracy=weighted_accuracy,
    precision=weighted_precision,
    recall=weighted_recall,
    f1=weighted_f1,
    hyperparameters="{'Weights': {'CatBoost': 0.5, 'XGBoost': 0.3, 'LightGBM': 0.2}}",
    notes="Weighted average probabilities from CatBoost, XGBoost, and LightGBM"
)

# Example for Stacking Ensemble
log_results(
    name="Model 5.2: Stacking Ensemble",
    auc=stacking_auc,  # Replace with calculated metrics
    accuracy=stacking_accuracy,
    precision=stacking_precision,
    recall=stacking_recall,
    f1=stacking_f1,
    hyperparameters="{'Meta-Model': 'Logistic Regression'}",
    notes="Stacked probabilities using CatBoost, XGBoost, and LightGBM as base models"
)

# Log results for Voting Ensemble
log_results(
    name="Model 5.3: Voting Ensemble",
    auc=mean_auc,
    accuracy=mean_accuracy,
    precision=mean_precision,
    recall=mean_recall,
    f1=mean_f1,
    hyperparameters="{'Voting': 'Soft'}",  # Ensemble-specific hyperparameters
    notes="Soft voting across CatBoost, XGBoost, and LightGBM"
)

# Save the updated results DataFrame to the CSV file
results_df.to_csv(results_file, index=False)

print("Ensemble results logged successfully!")


# ## Model 5.4 Weighted Average Ensemble Method with Random Forest

# In[42]:
# from sklearn.ensemble import RandomForestClassifier

# # Initialize models
# cat_cfecv_tuned = CatBoostClassifier(
#     random_state=42, verbose=0, iterations=500, depth=6, learning_rate=0.05,
#     cat_features=[X_best.columns.get_loc(col) for col in categorical_features if col in X_best.columns]
# )

# best_rf_model = RandomForestClassifier(
#     max_depth=10,
#     max_features=None,
#     min_samples_leaf=2,
#     min_samples_split=5,
#     n_estimators=100,
#     random_state=42
# )

# # Initialize K-Fold Cross-Validation
```

```
# kf = KFold(n_splits=5, shuffle=True, random_state=42)

# # Define weights for the ensemble
# weights = {'catboost': 0.7, 'random_forest': 0.3}  # Adjust weights as needed

# # Lists to store metrics
# auc_scores, accuracy_scores, precision_scores, recall_scores, f1_scores = [], [], [], [], []

# # Perform K-Fold Cross-Validation
# for fold, (train_index, test_index) in enumerate(kf.split(X_best)):
#     print(f"Fold {fold + 1}")

#     # Split data into train and test sets for the current fold
#     X_train_best, X_test_best = X_best.iloc[train_index], X_best.iloc[test_index]
#     y_train, y_test = y.iloc[train_index], y.iloc[test_index]

#     # Train CatBoost model
#     cat_cfecv_tuned.fit(X_train_best, y_train)

#     # Train Random Forest model
#     best_rf_model.fit(X_train_best, y_train)

#     # Generate predictions
#     y_proba_cat = cat_cfecv_tuned.predict_proba(X_test_best)[:, 1]
#     y_proba_rf = best_rf_model.predict_proba(X_test_best)[:, 1]

#     # Weighted ensemble probabilities
#     y_ensemble_proba = (
#         weights['catboost'] * y_proba_cat +
#         weights['random_forest'] * y_proba_rf
#     )
#     y_ensemble_pred = (y_ensemble_proba >= 0.5).astype(int)

#     # Evaluate performance
#     auc_scores.append(roc_auc_score(y_test, y_ensemble_proba))
#     accuracy_scores.append(accuracy_score(y_test, y_ensemble_pred))
#     precision_scores.append(precision_score(y_test, y_ensemble_pred, zero_division=0))
#     recall_scores.append(recall_score(y_test, y_ensemble_pred, zero_division=0))
#     f1_scores.append(f1_score(y_test, y_ensemble_pred, zero_division=0))

# rf_weighted_auc = sum(auc_scores) / len(auc_scores)
# rf_weighted_accuracy = sum(accuracy_scores) / len(accuracy_scores)
# rf_weighted_precision = sum(precision_scores) / len(precision_scores)
# rf_weighted_recall = sum(recall_scores) / len(recall_scores)
# rf_weighted_f1 = sum(f1_scores) / len(f1_scores)

# print("\nWeighted Average Ensemble Cross-Validation Results:")
# print(f"Mean AUC: {rf_weighted_auc:.4f}")
# print(f"Mean Accuracy: {rf_weighted_accuracy:.4f}")
# print(f"Mean Precision: {rf_weighted_precision:.4f}")
# print(f"Mean Recall: {rf_weighted_recall:.4f}")
# print(f"Mean F1-Score: {rf_weighted_f1:.4f}")


# ## Model 5.5 Stacking Ensemble Method with Random Forest

# In[43]:
# # Initialize meta-model
# meta_model = LogisticRegression(random_state=42)

# # Initialize K-Fold Cross-Validation
# kf = KFold(n_splits=5, shuffle=True, random_state=42)

# # Lists to store metrics
# auc_scores, accuracy_scores, precision_scores, recall_scores, f1_scores = [], [], [], [], []

# # Perform K-Fold Cross-Validation
# for fold, (train_index, test_index) in enumerate(kf.split(X_best)):
#     print(f"Fold {fold + 1}")

#     # Split data into train and test sets for the current fold
```

```
#     X_train_best, X_test_best = X_best.iloc[train_index], X_best.iloc[test_index]
#     y_train, y_test = y.iloc[train_index], y.iloc[test_index]

#     # Train base models
#     cat_cfecv_tuned.fit(X_train_best, y_train)
#     best_rf_model.fit(X_train_best, y_train)

#     # Generate predictions for meta-model
#     y_proba_cat = cat_cfecv_tuned.predict_proba(X_test_best)[:, 1]
#     y_proba_rf = best_rf_model.predict_proba(X_test_best)[:, 1]

#     # Combine predictions as features for the meta-model
#     stacked_features = pd.DataFrame({
#         'catboost': y_proba_cat,
#         'random_forest': y_proba_rf
#     })

#     # Train meta-model
#     meta_model.fit(stacked_features, y_test)

#     # Make predictions using meta-model
#     y_meta_proba = meta_model.predict_proba(stacked_features)[:, 1]
#     y_meta_pred = (y_meta_proba >= 0.5).astype(int)

#     # Evaluate performance
#     auc_scores.append(roc_auc_score(y_test, y_meta_proba))
#     accuracy_scores.append(accuracy_score(y_test, y_meta_pred))
#     precision_scores.append(precision_score(y_test, y_meta_pred, zero_division=0))
#     recall_scores.append(recall_score(y_test, y_meta_pred, zero_division=0))
#     f1_scores.append(f1_score(y_test, y_meta_pred, zero_division=0))

# rf_stacking_auc = sum(auc_scores) / len(auc_scores)
# rf_stacking_accuracy = sum(accuracy_scores) / len(accuracy_scores)
# rf_stacking_precision = sum(precision_scores) / len(precision_scores)
# rf_stacking_recall = sum(recall_scores) / len(recall_scores)
# rf_stacking_f1 = sum(f1_scores) / len(f1_scores)

# print("\nStacking Ensemble Cross-Validation Results:")
# print(f"Mean AUC: {stacking_auc:.4f}")
# print(f"Mean Accuracy: {stacking_accuracy:.4f}")
# print(f"Mean Precision: {stacking_precision:.4f}")
# print(f"Mean Recall: {stacking_recall:.4f}")
# print(f"Mean F1-Score: {stacking_f1:.4f}")


# ## Model 5.6: Voting Ensemble Method with Random Forest

# In[44]:
# from sklearn.ensemble import VotingClassifier

# # Initialize Voting Classifier
# voting_model = VotingClassifier(estimators=[
#     ('catboost', cat_cfecv_tuned),
#     ('random_forest', best_rf_model)
# ], voting='soft')  # Use 'hard' for hard voting

# # Lists to store metrics
# auc_scores, accuracy_scores, precision_scores, recall_scores, f1_scores = [], [], [], [], []

# # Perform K-Fold Cross-Validation
# for fold, (train_index, test_index) in enumerate(kf.split(X_best)):
#     print(f"Fold {fold + 1}")

#     # Split data into train and test sets for the current fold
#     X_train_best, X_test_best = X_best.iloc[train_index], X_best.iloc[test_index]
#     y_train, y_test = y.iloc[train_index], y.iloc[test_index]

#     # Train Voting Classifier
#     voting_model.fit(X_train_best, y_train)

#     # Make predictions
```

```
#     y_proba = voting_model.predict_proba(X_test_best)[:, 1]  # Probabilities
#     y_pred = voting_model.predict(X_test_best)  # Class labels

#     # Evaluate performance
#     auc_scores.append(roc_auc_score(y_test, y_proba))
#     accuracy_scores.append(accuracy_score(y_test, y_pred))
#     precision_scores.append(precision_score(y_test, y_pred, zero_division=0))
#     recall_scores.append(recall_score(y_test, y_pred, zero_division=0))
#     f1_scores.append(f1_score(y_test, y_pred, zero_division=0))

# # Compute average performance metrics
# rf_voting_auc = sum(auc_scores) / len(auc_scores)
# rf_voting_accuracy = sum(accuracy_scores) / len(accuracy_scores)
# rf_voting_precision = sum(precision_scores) / len(precision_scores)
# rf_voting_recall = sum(recall_scores) / len(recall_scores)
# rf_voting_f1 = sum(f1_scores) / len(f1_scores)

# # Print overall results
# print("\nVoting Ensemble Cross-Validation Results:")
# print(f"Mean AUC: {voting_auc:.4f}")
# print(f"Mean Accuracy: {voting_accuracy:.4f}")
# print(f"Mean Precision: {voting_precision:.4f}")
# print(f"Mean Recall: {voting_recall:.4f}")
# print(f"Mean F1-Score: {voting_f1:.4f}")


# In[45]:
# Initialize models
cat_cfecv_tuned = CatBoostClassifier(
    random_state=42, verbose=0, iterations=500, depth=6, learning_rate=0.05,
    cat_features=[X_best.columns.get_loc(col) for col in categorical_features if col in X_best.columns]
)

best_rf_model = RandomForestClassifier(
    max_depth=10,
    max_features=None,
    min_samples_leaf=2,
    min_samples_split=5,
    n_estimators=100,
    random_state=42
)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

def evaluate_model(name, weights=None):
    """Evaluate the model using K-Fold Cross-Validation."""
    auc_scores, accuracy_scores, precision_scores, recall_scores, f1_scores = [], [], [], [], []

    for fold, (train_index, test_index) in enumerate(kf.split(X_best)):
        print(f"Fold {fold + 1} for {name}")

        # Split data
        X_train, X_test = X_best.iloc[train_index], X_best.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        # Train base models
        cat_cfecv_tuned.fit(X_train, y_train)
        best_rf_model.fit(X_train, y_train)

        # Generate predictions
        y_proba_cat = cat_cfecv_tuned.predict_proba(X_test)[:, 1]
        y_proba_rf = best_rf_model.predict_proba(X_test)[:, 1]

        # Ensemble logic
        if name == "Weighted Average Ensemble":
            y_ensemble_proba = (
                weights['catboost'] * y_proba_cat +
                weights['random_forest'] * y_proba_rf
            )
        elif name == "Stacking Ensemble":
            stacked_features = pd.DataFrame({'catboost': y_proba_cat, 'random_forest': y_proba_rf})
```

```
            meta_model.fit(stacked_features, y_test)
            y_ensemble_proba = meta_model.predict_proba(stacked_features)[:, 1]
        elif name == "Voting Ensemble":
            voting_model = VotingClassifier(estimators=[
                ('catboost', cat_cfecv_tuned),
                ('random_forest', best_rf_model)
            ], voting='soft')
            voting_model.fit(X_train, y_train)
            y_ensemble_proba = voting_model.predict_proba(X_test)[:, 1]

        y_ensemble_pred = (y_ensemble_proba >= 0.5).astype(int)

        # Evaluate performance
        auc_scores.append(roc_auc_score(y_test, y_ensemble_proba))
        accuracy_scores.append(accuracy_score(y_test, y_ensemble_pred))
        precision_scores.append(precision_score(y_test, y_ensemble_pred, zero_division=0))
        recall_scores.append(recall_score(y_test, y_ensemble_pred, zero_division=0))
        f1_scores.append(f1_score(y_test, y_ensemble_pred, zero_division=0))

    return {
        "AUC": sum(auc_scores) / len(auc_scores),
        "Accuracy": sum(accuracy_scores) / len(accuracy_scores),
        "Precision": sum(precision_scores) / len(precision_scores),
        "Recall": sum(recall_scores) / len(recall_scores),
        "F1-Score": sum(f1_scores) / len(f1_scores),
    }

# Evaluate models
weighted_results = evaluate_model("Weighted Average Ensemble", weights={'catboost': 0.7, 'random_forest': 0.3})
stacking_results = evaluate_model("Stacking Ensemble")
voting_results = evaluate_model("Voting Ensemble")

# Print results
print("\nWeighted Average Ensemble Results:", weighted_results)
print("\nStacking Ensemble Results:", stacking_results)
print("\nVoting Ensemble Results:", voting_results)


# In[46]:
# File path to save and load results
results_file = 'model_performance_tracking.csv'

# Initialize results DataFrame if it doesn't exist
if not os.path.exists(results_file):
    results_df = pd.DataFrame(columns=['Model', 'AUC', 'Accuracy', 'Precision', 'Recall', 'F1-Score',
                                      'Hyperparameters', 'Notes'])
else:
    # Load existing results
    results_df = pd.read_csv(results_file)

# Function to log results into DataFrame
def log_results(name, auc, accuracy, precision, recall, f1, hyperparameters, notes):
    """Log results of a model or ensemble method."""
    new_entry = pd.DataFrame([{
        'Model': name,
        'AUC': auc,
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1-Score': f1,
        'Hyperparameters': str(hyperparameters),
        'Notes': notes
    }])
    global results_df
    results_df = pd.concat([results_df, new_entry], ignore_index=True)

# Log results for each ensemble method
# 1. Weighted Average Ensemble
log_results(
    name="Model 5.4: Weighted Average Ensemble (Random Forest)",
    auc=weighted_results["AUC"],  # Replace with computed metrics
```

```
    accuracy=weighted_results["Accuracy"],
    precision=weighted_results["Precision"],
    recall=weighted_results["Recall"],
    f1=weighted_results["F1-Score"],
    hyperparameters="{'Weights': {'CatBoost': 0.7, 'Random Forest': 0.3}}",
    notes="Weighted average probabilities from CatBoost and Random Forest"
)

# 2. Stacking Ensemble
log_results(
    name="Model 5.5: Stacking Ensemble (Random Forest)",
    auc=stacking_results["AUC"],  # Replace with computed metrics
    accuracy=stacking_results["Accuracy"],
    precision=stacking_results["Precision"],
    recall=stacking_results["Recall"],
    f1=stacking_results["F1-Score"],
    hyperparameters="{'Meta-Model': 'Logistic Regression', 'Base Models': ['CatBoost', 'Random Forest']}",
    notes="Stacked probabilities using CatBoost and Random Forest"
)

# 3. Voting Ensemble
log_results(
    name="Model 5.6: Voting Ensemble (Random Forest)",
    auc=voting_results["AUC"],  # Replace with computed metrics
    accuracy=voting_results["Accuracy"],
    precision=voting_results["Precision"],
    recall=voting_results["Recall"],
    f1=voting_results["F1-Score"],
    hyperparameters="{'Voting': 'Soft', 'Base Models': ['CatBoost', 'Random Forest']}",
    notes="Soft voting across CatBoost and Random Forest"
)

# Save the updated results DataFrame to the CSV file
results_df.to_csv(results_file, index=False)

print("\nEnsemble results logged successfully!")
print(results_df)


# ## Generate Predictions

# In[48]:
test_data = pd.read_csv('test.csv')

# Extract PatientID for output
patient_ids = test_data['PatientID']
test_features = test_data.drop(columns=['PatientID', 'DoctorInCharge'], errors='ignore')

# Subset test features to match models
test_features_best = test_features[X_best.columns]  # For CatBoost and Random Forest

# Predictions storage
predictions = {}

# 1. Weighted Average Ensemble Predictions
cat_cfecv_tuned.fit(X_best, y)
best_rf_model.fit(X_best, y)

y_proba_cat_test = cat_cfecv_tuned.predict_proba(test_features_best)[:, 1]
y_proba_rf_test = best_rf_model.predict_proba(test_features_best)[:, 1]

weights = {'catboost': 0.7, 'random_forest': 0.3}
y_ensemble_proba_weighted = (
    weights['catboost'] * y_proba_cat_test +
    weights['random_forest'] * y_proba_rf_test
)
y_ensemble_pred_weighted = (y_ensemble_proba_weighted >= 0.5).astype(int)

predictions["Weighted Average Ensemble"] = y_ensemble_pred_weighted

# 2. Stacking Ensemble Predictions (Corrected)
```

```
stacked_features_train = pd.DataFrame({
    'catboost': cat_cfecv_tuned.predict_proba(X_best)[:, 1],
    'random_forest': best_rf_model.predict_proba(X_best)[:, 1]
})

meta_model.fit(stacked_features_train, y)

stacked_features_test = pd.DataFrame({
    'catboost': y_proba_cat_test,
    'random_forest': y_proba_rf_test
})

y_meta_proba_test = meta_model.predict_proba(stacked_features_test)[:, 1]
y_meta_pred_test = (y_meta_proba_test >= 0.5).astype(int)

predictions["Stacking Ensemble"] = y_meta_pred_test

# 3. Voting Ensemble Predictions
voting_model = VotingClassifier(estimators=[
    ('catboost', cat_cfecv_tuned),
    ('random_forest', best_rf_model)
], voting='soft')
voting_model.fit(X_best, y)

y_voting_pred_test = voting_model.predict(test_features_best)

predictions["Voting Ensemble"] = y_voting_pred_test

# Save predictions to separate files
for model_name, preds in predictions.items():
    output = pd.DataFrame({
        'PatientID': patient_ids,
        'Diagnosis': preds
    })
    file_name = f'predictions_{model_name.replace(" ", "_").lower()}.csv'
    output.to_csv(file_name, index=False)
    print(f"Predictions saved to '{file_name}'")


# ## Optimize Final Ensemble Model for Accuracy

# In[49]:
def evaluate_model(name, weights=None):
    """Evaluate the model using K-Fold Cross-Validation with threshold optimization."""
    auc_scores, accuracy_scores, precision_scores, recall_scores, f1_scores = [], [], [], [], []
    optimal_thresholds = []

    for fold, (train_index, test_index) in enumerate(kf.split(X_best)):
        print(f"Fold {fold + 1} for {name}")

        # Split data
        X_train, X_test = X_best.iloc[train_index], X_best.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]

        # Train base models
        cat_cfecv_tuned.fit(X_train, y_train)
        best_rf_model.fit(X_train, y_train)

        # Generate predictions
        y_proba_cat = cat_cfecv_tuned.predict_proba(X_test)[:, 1]
        y_proba_rf = best_rf_model.predict_proba(X_test)[:, 1]

        # Ensemble logic
        if name == "Weighted Average Ensemble":
            y_ensemble_proba = (
                weights['catboost'] * y_proba_cat +
                weights['random_forest'] * y_proba_rf
            )
        elif name == "Stacking Ensemble":
            stacked_features = pd.DataFrame({'catboost': y_proba_cat, 'random_forest': y_proba_rf})
            meta_model.fit(stacked_features, y_test)
```

```
        y_ensemble_proba = meta_model.predict_proba(stacked_features)[:, 1]
    elif name == "Voting Ensemble":
        voting_model = VotingClassifier(estimators=[
            ('catboost', cat_cfecv_tuned),
            ('random_forest', best_rf_model)
        ], voting='soft')
        voting_model.fit(X_train, y_train)
        y_ensemble_proba = voting_model.predict_proba(X_test)[:, 1]

    # Optimize threshold for accuracy
    thresholds = np.arange(0.0, 1.01, 0.01)
    best_accuracy = 0
    best_threshold = 0.5

    for threshold in thresholds:
        y_ensemble_pred = (y_ensemble_proba >= threshold).astype(int)
        accuracy = accuracy_score(y_test, y_ensemble_pred)
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_threshold = threshold

    optimal_thresholds.append(best_threshold)
    print(f"Optimal Threshold for Fold {fold + 1}: {best_threshold}")

    # Apply optimal threshold
    y_ensemble_pred = (y_ensemble_proba >= best_threshold).astype(int)

    # Evaluate performance
    auc_scores.append(roc_auc_score(y_test, y_ensemble_proba))
    accuracy_scores.append(accuracy_score(y_test, y_ensemble_pred))
    precision_scores.append(precision_score(y_test, y_ensemble_pred, zero_division=0))
    recall_scores.append(recall_score(y_test, y_ensemble_pred, zero_division=0))
    f1_scores.append(f1_score(y_test, y_ensemble_pred, zero_division=0))

  return {
    "AUC": sum(auc_scores) / len(auc_scores),
    "Accuracy": sum(accuracy_scores) / len(accuracy_scores),
    "Precision": sum(precision_scores) / len(precision_scores),
    "Recall": sum(recall_scores) / len(recall_scores),
    "F1-Score": sum(f1_scores) / len(f1_scores),
    "Optimal Threshold": np.mean(optimal_thresholds),  # Average threshold across folds
  }

# Evaluate models
weighted_results = evaluate_model("Weighted Average Ensemble", weights={'catboost': 0.7, 'random_forest': 0.3})
stacking_results = evaluate_model("Stacking Ensemble")
voting_results = evaluate_model("Voting Ensemble")

# Print results
print("\nWeighted Average Ensemble Results:", weighted_results)
print("\nStacking Ensemble Results:", stacking_results)
print("\nVoting Ensemble Results:", voting_results)

# In[50]:
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve

# Re-train models on the entire training dataset (X_best, y)
cat_cfecv_tuned.fit(X_best, y)
best_rf_model.fit(X_best, y)

# Generate predictions (probabilities) for the training dataset
y_proba_cat = cat_cfecv_tuned.predict_proba(X_best)[:, 1]
y_proba_rf = best_rf_model.predict_proba(X_best)[:, 1]

# Combine predictions using the weighted ensemble logic
weights = {'catboost': 0.7, 'random_forest': 0.3}
y_ensemble_proba_weighted = (
  weights['catboost'] * y_proba_cat +
  weights['random_forest'] * y_proba_rf
)
```

```
# Compute ROC curve on the training dataset
fpr, tpr, thresholds = roc_curve(y, y_ensemble_proba_weighted)

# Find the optimal threshold (closest to TPR=1, FPR=0)
optimal_idx = (tpr - fpr).argmax()
optimal_threshold = thresholds[optimal_idx]

# Plot the ROC curve and mark the optimal threshold
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label='ROC Curve', color='blue')
plt.scatter(fpr[optimal_idx], tpr[optimal_idx], color='red', label=f'Optimal Threshold = {optimal_threshold:.2f}')
plt.title("ROC Curve with Optimal Threshold (Weighted Average Ensemble)")
plt.xlabel("False Positive Rate (FPR)")
plt.ylabel("True Positive Rate (TPR)")
plt.legend()
plt.grid()
plt.show()


# In[51]:
test_data = pd.read_csv('test.csv')
patient_ids = test_data['PatientID']
test_features = test_data.drop(columns=['PatientID', 'DoctorInCharge'], errors='ignore')
test_features_best = test_features[X_best.columns]  # For CatBoost and Random Forest

# Train base models on the full dataset
cat_cfecv_tuned.fit(X_best, y)
best_rf_model.fit(X_best, y)

# Predictions storage
predictions = {}

# 1. Weighted Average Ensemble Predictions
y_proba_cat_test = cat_cfecv_tuned.predict_proba(test_features_best)[:, 1]
y_proba_rf_test = best_rf_model.predict_proba(test_features_best)[:, 1]

weights = {'catboost': 0.7, 'random_forest': 0.3}
y_ensemble_proba_weighted = (
    weights['catboost'] * y_proba_cat_test +
    weights['random_forest'] * y_proba_rf_test
)
# Apply the optimal threshold for Weighted Average Ensemble
optimal_threshold_weighted = 0.436
y_ensemble_pred_weighted = (y_ensemble_proba_weighted >= optimal_threshold_weighted).astype(int)
predictions["Weighted Average Ensemble"] = y_ensemble_pred_weighted

# 2. Stacking Ensemble Predictions
stacked_features_test = pd.DataFrame({
    'catboost': y_proba_cat_test,
    'random_forest': y_proba_rf_test
})
meta_model.fit(stacked_features_train, y)  # Train meta-model on full training data
y_meta_proba_test = meta_model.predict_proba(stacked_features_test)[:, 1]
# Apply the optimal threshold for Stacking Ensemble
optimal_threshold_stacking = 0.33
y_meta_pred_test = (y_meta_proba_test >= optimal_threshold_stacking).astype(int)
predictions["Stacking Ensemble"] = y_meta_pred_test

# 3. Voting Ensemble Predictions
voting_model = VotingClassifier(estimators=[
    ('catboost', cat_cfecv_tuned),
    ('random_forest', best_rf_model)
], voting='soft')
voting_model.fit(X_best, y)
y_voting_proba_test = voting_model.predict_proba(test_features_best)[:, 1]
# Apply the optimal threshold for Voting Ensemble
optimal_threshold_voting = 0.39
y_voting_pred_test = (y_voting_proba_test >= optimal_threshold_voting).astype(int)
predictions["Voting Ensemble"] = y_voting_pred_tes
# Save predictions to separate files
```

```
for model_name, preds in predictions.items():
    output = pd.DataFrame({
        'PatientID': patient_ids,
        'Diagnosis': preds
    })
    file_name = f'predictions_{model_name.replace(" ", "_").lower()}.csv'
    output.to_csv(file_name, index=False)
    print(f"Predictions saved to '{file_name}'")
```

2.

```
!pip install statsmodels
!pip install scikit-learn
!pip install xgboost
!pip install seaborn

import pandas as pd
import os
import statsmodels.api as sm
from statsmodels.iolib.summary2 import summary_col
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, roc_auc_score, roc_curve
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_selection import RFE
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
from scipy.stats import zscore
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import IsolationForest
from scipy.stats import chi2_contingency
from sklearn.model_selection import KFold
import numpy as np

relative_path1 = os.path.join('..', 'Data', 'train.csv')
abs_path1 = os.path.abspath(relative_path1)
train_df = pd.read_csv(abs_path1)

relative_path2 = os.path.join('..', 'Data', 'test.csv')
abs_path2 = os.path.abspath(relative_path2)
test_df = pd.read_csv(abs_path2)

# Splitting into Training and Testing Datasets
X = train_df.drop(['DoctorInCharge', 'PatientID', 'Diagnosis'], axis=1)
y = train_df['Diagnosis']

# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
In [4]:
test_X = test_df.drop(['DoctorInCharge', 'PatientID'], axis=1)
In [5]:
# standardize the features
scaler = StandardScaler()

test_X_scaled = scaler.fit_transform(test_X)

## 0. Check Outliers
```

In [59]:
```
category = X_train[['Gender', 'Ethnicity', 'Smoking', 'FamilyHistoryAlzheimers', 'CardiovascularDisease', 'EducationLevel',
                    'Diabetes', 'Depression', 'HeadInjury', 'Hypertension', 'MemoryComplaints', 'BehavioralProblems',
                    'Confusion', 'Disorientation', 'PersonalityChanges', 'DifficultyCompletingTasks', 'Forgetfulness']]
continuous = X_train[['Age', 'BMI', 'AlcoholConsumption', 'PhysicalActivity', 'DietQuality', 'SleepQuality', 'SystolicBP',
                      'DiastolicBP', 'CholesterolTotal', 'CholesterolLDL', 'CholesterolHDL', 'CholesterolTriglycerides',
                      'MMSE', 'FunctionalAssessment', 'ADL']]
```
IQR for continuous variables

In [60]:
```
Q1 = continuous.quantile(0.25)
Q3 = continuous.quantile(0.75)
IQR = Q3 - Q1

outliers = (continuous < (Q1 - 1.5 * IQR)) | (continuous > (Q3 + 1.5 * IQR))
outlier_counts = outliers.sum()
print(outlier_counts)
```

Test categorical variables

In [61]:
```
plt.figure(figsize=(15, 10))
for i, col in enumerate(category.columns, 1):
    plt.subplot(4, 5, i)
    sns.countplot(data=category, x=col)
    plt.title(col)
    plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

for col in category.columns:
    print(f"Target distribution for {col}:")
    print(train_df.groupby(col)['Diagnosis'].mean())
    print("\n")

for col in category.columns:
    pd.crosstab(train_df[col], train_df['Diagnosis']).plot(kind='bar', stacked=True)
    plt.title(f"{col} vs Diagnosis")
    plt.xlabel(col)
    plt.ylabel("Count")
    plt.show()

for col in category:
    contingency_table = pd.crosstab(X_train[col], y_train)
    chi2, p, _, _ = chi2_contingency(contingency_table)
    print(f"{col}: p-value = {p}")
```

1. Decision Trees

In [26]:
```
model1 = DecisionTreeClassifier(random_state=42)
model1.fit(X_train, y_train)
```

1.1 Features Selection

1.1.1 Retrieve Feature Importances

In [27]:
```
importances = model1.feature_importances_

feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': importances})
feature_importance_df = feature_importance_df.sort_values(by='Importance', ascending=False)
feature_importance_df

# All features have importance
top_features1 = feature_importance_df.head(21)['Feature'].values
X_train_selected1 = pd.DataFrame(scaler.fit_transform(X_train[top_features1]), columns=top_features1)
X_test_selected1 = pd.DataFrame(scaler.transform(X_test[top_features1]), columns=top_features1)
```
In [44]:
```
# Features have importance > 0.003
top_features2 = feature_importance_df.head(17)['Feature'].values
X_train_selected2 = pd.DataFrame(scaler.fit_transform(X_train[top_features2]), columns=top_features2)
X_test_selected2 = pd.DataFrame(scaler.transform(X_test[top_features2]), columns=top_features2)
```
In [55]:
```
# Features have importance > 0.005
top_features3 = feature_importance_df.head(13)['Feature'].values
```

```
X_train_selected3 = pd.DataFrame(scaler.fit_transform(X_train[top_features3]), columns=top_features3)
X_test_selected3 = pd.DataFrame(scaler.transform(X_test[top_features3]), columns=top_features3)
In [56]:
# Features have importance > 0.01
top_features4 = feature_importance_df.head(10)['Feature'].values
X_train_selected4 = pd.DataFrame(scaler.fit_transform(X_train[top_features4]), columns=top_features4)
X_test_selected4 = pd.DataFrame(scaler.transform(X_test[top_features4]), columns=top_features4)
In [57]:
# Features have importance > 0.1
top_features5 = feature_importance_df.head(5)['Feature'].values
X_train_selected5 = pd.DataFrame(scaler.fit_transform(X_train[top_features5]), columns=top_features5)
X_test_selected5 = pd.DataFrame(scaler.transform(X_test[top_features5]), columns=top_features5)
#### 1.1.2 Recursive Feature Elimination
rfe_selector = RFE(estimator=DecisionTreeClassifier(random_state=42), n_features_to_select=10, step=1)
X_train_rfe = rfe_selector.fit_transform(X_train, y_train)

# Check selected features
selected_features = X_train.columns[rfe_selector.support_]
In [59]:
rfe_features = selected_features.values
X_train_rfe = pd.DataFrame(scaler.fit_transform(X_train[rfe_features]), columns=rfe_features)
X_test_rfe = pd.DataFrame(scaler.transform(X_test[rfe_features]), columns=rfe_features)
### 1.2 Evaluate Models
In [60]:
# Original Model
predictions1 = model1.predict(X_test)

# Decision Tree 1
model_top21 = DecisionTreeClassifier(random_state=42)
model_top21.fit(X_train_selected1, y_train)
predictions_top21 = model_top21.predict(X_test_selected1)
# Decision Tree 2
model_top17 = DecisionTreeClassifier(random_state=42)
model_top17.fit(X_train_selected2, y_train)
predictions_top17 = model_top17.predict(X_test_selected2)
# Decision Tree 3
model_top13 = DecisionTreeClassifier(random_state=42)
model_top13.fit(X_train_selected3, y_train)
predictions_top13 = model_top13.predict(X_test_selected3)
# Decision Tree 4
model_top10 = DecisionTreeClassifier(random_state=42)
model_top10.fit(X_train_selected4, y_train)
predictions_top10 = model_top10.predict(X_test_selected4)
# Decision Tree 5
model_top5 = DecisionTreeClassifier(random_state=42)
model_top5.fit(X_train_selected5, y_train)
predictions_top5 = model_top5.predict(X_test_selected5)
# RFE Model
model_rfe = DecisionTreeClassifier(random_state=42)
model_rfe.fit(X_train_rfe, y_train)
predictions_rfe = model_rfe.predict(X_test_rfe)

# Evaluation for original model
accuracy_1 = accuracy_score(y_test, predictions1)
f1_1 = f1_score(y_test, predictions1)
roc_auc_1 = roc_auc_score(y_test, model1.predict_proba(X_test)[:, 1])

# Evaluation for top 21 features
accuracy_top21 = accuracy_score(y_test, predictions_top21)
f1_top21 = f1_score(y_test, predictions_top21)
roc_auc_top21 = roc_auc_score(y_test, model_top21.predict_proba(X_test_selected1)[:, 1])
# Evaluation for top 17 features
accuracy_top17 = accuracy_score(y_test, predictions_top17)
f1_top17 = f1_score(y_test, predictions_top17)
roc_auc_top17 = roc_auc_score(y_test, model_top17.predict_proba(X_test_selected2)[:, 1])
# Evaluation for top 13 features
accuracy_top13 = accuracy_score(y_test, predictions_top13)
f1_top13 = f1_score(y_test, predictions_top13)
roc_auc_top13 = roc_auc_score(y_test, model_top13.predict_proba(X_test_selected3)[:, 1])
# Evaluation for top 10 features
accuracy_top10 = accuracy_score(y_test, predictions_top10)
```

```
f1_top10 = f1_score(y_test, predictions_top10)
roc_auc_top10 = roc_auc_score(y_test, model_top10.predict_proba(X_test_selected4)[:, 1])
# Evaluation for top 5 features
accuracy_top5 = accuracy_score(y_test, predictions_top5)
f1_top5 = f1_score(y_test, predictions_top5)
roc_auc_top5 = roc_auc_score(y_test, model_top5.predict_proba(X_test_selected5)[:, 1])

# Evaluation for RFE features
accuracy_rfe = accuracy_score(y_test, predictions_rfe)
f1_rfe = f1_score(y_test, predictions_rfe)
roc_auc_rfe = roc_auc_score(y_test, model_rfe.predict_proba(X_test_rfe)[:, 1])


print("Original Model - Accuracy:", accuracy_1, "F1 Score:", f1_1, "ROC-AUC:", roc_auc_1)
print("Model with Top 21 Features - Accuracy:", accuracy_top21, "F1 Score:", f1_top21, "ROC-AUC:", roc_auc_top21)
print("Model with Top 17 Features - Accuracy:", accuracy_top17, "F1 Score:", f1_top17, "ROC-AUC:", roc_auc_top17)
print("Model with Top 13 Features - Accuracy:", accuracy_top13, "F1 Score:", f1_top13, "ROC-AUC:", roc_auc_top13)
print("Model with Top 10 Features - Accuracy:", accuracy_top10, "F1 Score:", f1_top10, "ROC-AUC:", roc_auc_top10)
print("Model with Top 5 Features - Accuracy:", accuracy_top5, "F1 Score:", f1_top5, "ROC-AUC:", roc_auc_top5)
print("RFE Model - Accuracy:", accuracy_rfe, "F1 Score:", f1_rfe, "ROC-AUC:", roc_auc_rfe)
```

2. Random Forests

In [24]:

```
model2 = RandomForestClassifier(random_state=42)
model2.fit(X_train, y_train)
```

2.1 Features Selection

In [25]:

```
rf_importances = model2.feature_importances_

rf_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': rf_importances})
rf_feature_importance_df = rf_feature_importance_df.sort_values(by='Importance', ascending=False)
rf_feature_importance_df
# Features have importance > 0.01
top_rf_features1 = rf_feature_importance_df.head(17)['Feature'].values
X_train_rf_selected1 = pd.DataFrame(scaler.fit_transform(X_train[top_rf_features1]), columns=top_rf_features1)
X_test_rf_selected1 = pd.DataFrame(scaler.transform(X_test[top_rf_features1]), columns=top_rf_features1)
```

In [30]:

```
# Features have importance > 0.03
top_rf_features2 = rf_feature_importance_df.head(11)['Feature'].values
X_train_rf_selected2 = pd.DataFrame(scaler.fit_transform(X_train[top_rf_features2]), columns=top_rf_features2)
X_test_rf_selected2 = pd.DataFrame(scaler.transform(X_test[top_rf_features2]), columns=top_rf_features2)
```

In [31]:

```
# Features have importance > 0.05
top_rf_features3 = rf_feature_importance_df.head(5)['Feature'].values
X_train_rf_selected3 = pd.DataFrame(scaler.fit_transform(X_train[top_rf_features3]), columns=top_rf_features3)
X_test_rf_selected3 = pd.DataFrame(scaler.transform(X_test[top_rf_features3]), columns=top_rf_features3)
```

2.2 Evaluate Models

In [65]:

```
# Original Model
predictions2 = model2.predict(X_test)

# Random Forest 1
model_rf_top17 = RandomForestClassifier(random_state=42)
model_rf_top17.fit(X_train_rf_selected1, y_train)
predictions_rf_top17 = model_rf_top17.predict(X_test_rf_selected1)

# Random Forest 2
model_rf_top11 = RandomForestClassifier(random_state=42)
model_rf_top11.fit(X_train_rf_selected2, y_train)
predictions_rf_top11 = model_rf_top11.predict(X_test_rf_selected2)

# Random Forest 3
model_rf_top5 = RandomForestClassifier(random_state=42)
model_rf_top5.fit(X_train_rf_selected3, y_train)
predictions_rf_top5 = model_rf_top5.predict(X_test_rf_selected3)

# Evaluation for original model
accuracy_2 = accuracy_score(y_test, predictions2)
f1_2 = f1_score(y_test, predictions2)
roc_auc_2 = roc_auc_score(y_test, model2.predict_proba(X_test)[:, 1])
```

```
# Evaluation for top 17 features
accuracy_rf_top17 = accuracy_score(y_test, predictions_rf_top17)
f1_rf_top17 = f1_score(y_test, predictions_rf_top17)
roc_auc_rf_top17 = roc_auc_score(y_test, model_rf_top17.predict_proba(X_test_rf_selected1)[:, 1])

# Evaluation for top 11 features
accuracy_rf_top11 = accuracy_score(y_test, predictions_rf_top11)
f1_rf_top11 = f1_score(y_test, predictions_rf_top11)
roc_auc_rf_top11 = roc_auc_score(y_test, model_rf_top11.predict_proba(X_test_rf_selected2)[:, 1])

# Evaluation for top 5 features
accuracy_rf_top5 = accuracy_score(y_test, predictions_rf_top5)
f1_rf_top5 = f1_score(y_test, predictions_rf_top5)
roc_auc_rf_top5 = roc_auc_score(y_test, model_rf_top5.predict_proba(X_test_rf_selected3)[:, 1])

print("Original Model - Accuracy:", accuracy_2, "F1 Score:", f1_2, "ROC-AUC:", roc_auc_2)
print("Model with Top 17 Features - Accuracy:", accuracy_rf_top17, "F1 Score:", f1_rf_top17, "ROC-AUC:", roc_auc_rf_top17)
print("Model with Top 11 Features - Accuracy:", accuracy_rf_top11, "F1 Score:", f1_rf_top11, "ROC-AUC:", roc_auc_rf_top11)
print("Model with Top 5 Features - Accuracy:", accuracy_rf_top5, "F1 Score:", f1_rf_top5, "ROC-AUC:", roc_auc_rf_top5)
#### 2.2.2 5-Fold Cross Validation
In [27]:
model2 = RandomForestClassifier(random_state=42)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    model2.fit(X_train, y_train)

    predictions2 = model2.predict(X_test)
    proba_predictions2 = model2.predict_proba(X_test)[:, 1]

    accuracy = accuracy_score(y_test, predictions2)
    f1 = f1_score(y_test, predictions2)
    precision = precision_score(y_test, predictions2)
    recall = recall_score(y_test, predictions2)
    auc = roc_auc_score(y_test, proba_predictions2)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

model2 = RandomForestClassifier(random_state=42)
```

```
kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

# top17

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    model2.fit(X_train, y_train)

    # Feature importance
    rf_importances = model2.feature_importances_
    rf_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': rf_importances})
    rf_feature_importance_df = rf_feature_importance_df.sort_values(by='Importance', ascending=False)

    # Select top features
    top_rf_features1 = rf_feature_importance_df.head(17)['Feature'].values
    X_train_rf_selected1 = X_train[top_rf_features1]
    X_test_rf_selected1 = X_test[top_rf_features1]

    # Train model on selected features
    model_rf_top17 = RandomForestClassifier(random_state=42)
    model_rf_top17.fit(X_train_rf_selected1, y_train)
    predictions_rf_top17 = model_rf_top17.predict(X_test_rf_selected1)
    proba_predictions_top17 = model_rf_top17.predict_proba(X_test_rf_selected1)[:, 1]

    accuracy = accuracy_score(y_test, predictions_rf_top17)
     f1 = f1_score(y_test, predictions_rf_top17)
     precision = precision_score(y_test, predictions_rf_top17)
    recall = recall_score(y_test, predictions_rf_top17)
    auc = roc_auc_score(y_test, proba_predictions_top17)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

model2 = RandomForestClassifier(random_state=42)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

# top11
```

```
for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    model2.fit(X_train, y_train)

    # Feature importance
    rf_importances = model2.feature_importances_
    rf_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': rf_importances})
    rf_feature_importance_df = rf_feature_importance_df.sort_values(by='Importance', ascending=False)

    # Select top features
    top_rf_features2 = rf_feature_importance_df.head(11)['Feature'].values
    X_train_rf_selected2 = X_train[top_rf_features2]
    X_test_rf_selected2 = X_test[top_rf_features2]

    # Train model on selected features
    model_rf_top11 = RandomForestClassifier(random_state=42)
    model_rf_top11.fit(X_train_rf_selected2, y_train)
    predictions_rf_top11 = model_rf_top11.predict(X_test_rf_selected2)
     proba_predictions_top11 = model_rf_top11.predict_proba(X_test_rf_selected2)[:, 1]

     accuracy = accuracy_score(y_test, predictions_rf_top11)
    f1 = f1_score(y_test, predictions_rf_top11)
    precision = precision_score(y_test, predictions_rf_top11)
    recall = recall_score(y_test, predictions_rf_top11)
    auc = roc_auc_score(y_test, proba_predictions_top11)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

model2 = RandomForestClassifier(random_state=42)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

# top5

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    model2.fit(X_train, y_train)
```

```
    # Feature importance
    rf_importances = model2.feature_importances_
    rf_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': rf_importances})
    rf_feature_importance_df = rf_feature_importance_df.sort_values(by='Importance', ascending=False)

    # Select top features
    top_rf_features3 = rf_feature_importance_df.head(5)['Feature'].values
    X_train_rf_selected3 = X_train[top_rf_features3]
    X_test_rf_selected3 = X_test[top_rf_features3]

    # Train model on selected features
    model_rf_top5 = RandomForestClassifier(random_state=42)
    model_rf_top5.fit(X_train_rf_selected3, y_train)
    predictions_rf_top5 = model_rf_top5.predict(X_test_rf_selected3)
    proba_predictions_top5 = model_rf_top5.predict_proba(X_test_rf_selected3)[:, 1]

    accuracy = accuracy_score(y_test, predictions_rf_top5)
    f1 = f1_score(y_test, predictions_rf_top5)
    precision = precision_score(y_test, predictions_rf_top5)
    recall = recall_score(y_test, predictions_rf_top5)
    auc = roc_auc_score(y_test, proba_predictions_top5)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

### 2.3 Improvement
#### 2.3.1 First try - Hyperparameter tuning
In [11]:
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 15, 20, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['sqrt', 'log2', None]
}

grid_search = GridSearchCV(estimator=model2, param_grid=param_grid, scoring='f1', cv=3)
grid_search.fit(X_train_rf_selected2, y_train)

print("Best Parameters:", grid_search.best_params_)

best_rf_model = RandomForestClassifier(
    max_depth=10,
    max_features=None,
    min_samples_leaf=2,
    min_samples_split=5,
    n_estimators=100,
    random_state=42
)

# Fit on the training data
best_rf_model.fit(X, y)

best_rf_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': best_rf_model.feature_importances_})
```

```
best_rf_feature_importance_df = best_rf_feature_importance_df.sort_values(by='Importance', ascending=False)
best_rf_feature_importance_df

# Features have importance > 0.0005
top_best_features0 = best_rf_feature_importance_df.head(23)['Feature'].values
X_train_best_selected0 = pd.DataFrame(X_train[top_best_features0], columns=top_best_features0)
X_test_best_selected0 = pd.DataFrame(X_test[top_best_features0], columns=top_best_features0)
In [14]:
# Features have importance > 0.001
top_best_features1 = best_rf_feature_importance_df.head(19)['Feature'].values
X_train_best_selected1 = pd.DataFrame(scaler.fit_transform(X_train[top_best_features1]), columns=top_best_features1)
X_test_best_selected1 = pd.DataFrame(scaler.transform(X_test[top_best_features1]), columns=top_best_features1)
In [16]:
# Features have importance > 0.004
top_best_features2 = best_rf_feature_importance_df.head(15)['Feature'].values
X_train_best_selected2 = pd.DataFrame(scaler.fit_transform(X_train[top_best_features2]), columns=top_best_features2)
X_test_best_selected2 = pd.DataFrame(scaler.transform(X_test[top_best_features2]), columns=top_best_features2)
In [17]:
# Features have importance > 0.01
top_best_features3 = best_rf_feature_importance_df.head(9)['Feature'].values
X_train_best_selected3 = pd.DataFrame(scaler.fit_transform(X_train[top_best_features3]), columns=top_best_features3)
X_test_best_selected3 = pd.DataFrame(scaler.transform(X_test[top_best_features3]), columns=top_best_features3)
In [57]:
# RF Best Model
predictionbest = best_rf_model.predict(X_test)

# Random Forest Best 0
model_best_top23 = RandomForestClassifier(random_state=42)
model_best_top23.fit(X_train_best_selected0, y_train)
predictions_best_top23 = model_best_top23.predict(X_test_best_selected0)

# Random Forest Best 1
model_best_top19 = RandomForestClassifier(random_state=42)
model_best_top19.fit(X_train_best_selected1, y_train)
predictions_best_top19 = model_best_top19.predict(X_test_best_selected1)

# Random Forest Best 2
model_best_top15 = RandomForestClassifier(random_state=42)
model_best_top15.fit(X_train_best_selected2, y_train)
predictions_best_top15 = model_best_top15.predict(X_test_best_selected2)

# Random Forest Best 3
model_best_top9 = RandomForestClassifier(random_state=42)
model_best_top9.fit(X_train_best_selected3, y_train)
predictions_best_top9 = model_best_top9.predict(X_test_best_selected3)
In [58]:
# Evaluation for RF best model
accuracy_best = accuracy_score(y_test, predictionbest)
f1_best = f1_score(y_test, predictionbest, average='weighted')
roc_auc_best = roc_auc_score(y_test, best_rf_model.predict_proba(X_test)[:, 1])

# Evaluation for top 23 features
accuracy_best_top23 = accuracy_score(y_test, predictions_best_top23)
f1_best_top23 = f1_score(y_test, predictions_best_top23)
roc_auc_best_top23 = roc_auc_score(y_test, model_best_top23.predict_proba(X_test_best_selected0)[:, 1])

# Evaluation for top 19 features
accuracy_best_top19 = accuracy_score(y_test, predictions_best_top19)
f1_best_top19 = f1_score(y_test, predictions_best_top19)
roc_auc_best_top19 = roc_auc_score(y_test, model_best_top19.predict_proba(X_test_best_selected1)[:, 1])

# Evaluation for top 15 features
accuracy_best_top15 = accuracy_score(y_test, predictions_best_top15)
f1_best_top15 = f1_score(y_test, predictions_best_top15)
roc_auc_best_top15 = roc_auc_score(y_test, model_best_top15.predict_proba(X_test_best_selected2)[:, 1])

# Evaluation for top 9 features
accuracy_best_top9 = accuracy_score(y_test, predictions_best_top9)
f1_best_top9 = f1_score(y_test, predictions_best_top9)
roc_auc_best_top9 = roc_auc_score(y_test, model_best_top9.predict_proba(X_test_best_selected3)[:, 1])
```

```
print("Best RF Model - Accuracy:", accuracy_best, "F1 Score:", f1_best, "ROC-AUC:", roc_auc_best)
print("Best Model with Top 23 Features - Accuracy:", accuracy_best_top23, "F1 Score:", f1_best_top23, "ROC-AUC:", roc_auc_best_top23)
print("Best Model with Top 19 Features - Accuracy:", accuracy_best_top19, "F1 Score:", f1_best_top19, "ROC-AUC:", roc_auc_best_top19)
print("Best Model with Top 15 Features - Accuracy:", accuracy_best_top15, "F1 Score:", f1_best_top15, "ROC-AUC:", roc_auc_best_top15)
print("Best Model with Top 9 Features - Accuracy:", accuracy_best_top9, "F1 Score:", f1_best_top9, "ROC-AUC:", roc_auc_best_top9)
```

In [34]:

```
best_rf_model = RandomForestClassifier(
    max_depth=10,
    max_features=None,
    min_samples_leaf=2,
    min_samples_split=5,
    n_estimators=100,
    random_state=42
)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    best_rf_model.fit(X_train, y_train)

    predictionbest = best_rf_model.predict(X_test)
    proba_predictionbest = best_rf_model.predict_proba(X_test)[:, 1]

    accuracy = accuracy_score(y_test, predictionbest)
    f1 = f1_score(y_test, predictionbest)
    precision = precision_score(y_test, predictionbest)
    recall = recall_score(y_test, predictionbest)
    auc = roc_auc_score(y_test, proba_predictionbest)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)
best_rf_model = RandomForestClassifier(
    max_depth=10,
    max_features=None,
    min_samples_leaf=2,
    min_samples_split=5,
    n_estimators=100,
    random_state=42
)

kf = KFold(n_splits=5, shuffle=True, random_state=42)
```

```
auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

# top23

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    best_rf_model.fit(X_train, y_train)

    # Feature importance
    best_rf_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': best_rf_model.feature_importances_})
    best_rf_feature_importance_df = best_rf_feature_importance_df.sort_values(by='Importance', ascending=False)

    # Select top features
    top_best_features0 = best_rf_feature_importance_df.head(23)['Feature'].values
    X_train_best_selected0 = X_train[top_best_features0]
    X_test_best_selected0 = X_test[top_best_features0]

    # Train model on selected features
    model_best_top23 = RandomForestClassifier(random_state=42)
    model_best_top23.fit(X_train_best_selected0, y_train)
    predictions_best_top23 = model_best_top23.predict(X_test_best_selected0)
    proba_predictions_best_top23 = model_best_top23.predict_proba(X_test_best_selected0)[:, 1]

    accuracy = accuracy_score(y_test, predictions_best_top23)
    f1 = f1_score(y_test, predictions_best_top23)
    precision = precision_score(y_test, predictions_best_top23)
    recall = recall_score(y_test, predictions_best_top23)
    auc = roc_auc_score(y_test, proba_predictions_best_top23)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

best_rf_model = RandomForestClassifier(
    max_depth=10,
    max_features=None,
    min_samples_leaf=2,
    min_samples_split=5,
    n_estimators=100,
    random_state=42
)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
```

```
f1_scores = []
precision_scores = []
recall_scores = []

# top15

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    best_rf_model.fit(X_train, y_train)

    # Feature importance
    best_rf_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': best_rf_model.feature_importances_})
    best_rf_feature_importance_df = best_rf_feature_importance_df.sort_values(by='Importance', ascending=False)

    # Select top features
    top_best_features2 = best_rf_feature_importance_df.head(15)['Feature'].values
    X_train_best_selected2 = X_train[top_best_features2]
    X_test_best_selected2 = X_test[top_best_features2]

    # Train model on selected features
    model_best_top15 = RandomForestClassifier(random_state=42)
    model_best_top15.fit(X_train_best_selected2, y_train)
    predictions_best_top15 = model_best_top15.predict(X_test_best_selected2)
    proba_predictions_best_top15 = model_best_top15.predict_proba(X_test_best_selected2)[:, 1]

    accuracy = accuracy_score(y_test, predictions_best_top15)
    f1 = f1_score(y_test, predictions_best_top15)
    precision = precision_score(y_test, predictions_best_top15)
    recall = recall_score(y_test, predictions_best_top15)
    auc = roc_auc_score(y_test, proba_predictions_best_top15)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

best_rf_model = RandomForestClassifier(
    max_depth=10,
    max_features=None,
    min_samples_leaf=2,
    min_samples_split=5,
    n_estimators=100,
    random_state=42
)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []
```

```
# top9

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    best_rf_model.fit(X_train, y_train)

    # Feature importance
    best_rf_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': best_rf_model.feature_importances_})
    best_rf_feature_importance_df = best_rf_feature_importance_df.sort_values(by='Importance', ascending=False)

    # Select top features
    top_best_features3 = best_rf_feature_importance_df.head(9)['Feature'].values
    X_train_best_selected3 = X_train[top_best_features3]
    X_test_best_selected3 = X_test[top_best_features3]

    # Train model on selected features
    model_best_top9 = RandomForestClassifier(random_state=42)
    model_best_top9.fit(X_train_best_selected3, y_train)
    predictions_best_top9 = model_best_top9.predict(X_test_best_selected3)
    proba_predictions_best_top9 = model_best_top9.predict_proba(X_test_best_selected3)[:, 1]

    accuracy = accuracy_score(y_test, predictions_best_top9)
    f1 = f1_score(y_test, predictions_best_top9)
    precision = precision_score(y_test, predictions_best_top9)
    recall = recall_score(y_test, predictions_best_top9)
    auc = roc_auc_score(y_test, proba_predictions_best_top9)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

### 2.3.1 Second Try - Increase the number of trees
random_forest_model = RandomForestClassifier(
    max_depth=10,
    max_features=None,
    min_samples_leaf=2,
    min_samples_split=5,
    n_estimators=200,  # Increased from 100 to 200
    random_state=42
)

# Fit the model on the training data
random_forest_model.fit(X_train, y_train)

afeature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': random_forest_model.feature_importances_})
afeature_importance_df = afeature_importance_df.sort_values(by='Importance', ascending=False)
afeature_importance_df

# Features have importance > 0.001
a_features1 = afeature_importance_df.head(19)['Feature'].values
```

```
X_train_a_selected1 = pd.DataFrame(scaler.fit_transform(X_train[a_features1]), columns=a_features1)
X_test_a_selected1 = pd.DataFrame(scaler.transform(X_test[a_features1]), columns=a_features1)

# Features have importance > 0.005
a_features2 = afeature_importance_df.head(13)['Feature'].values
X_train_a_selected2 = pd.DataFrame(scaler.fit_transform(X_train[a_features2]), columns=a_features2)
X_test_a_selected2 = pd.DataFrame(scaler.transform(X_test[a_features2]), columns=a_features2)

# Features have importance > 0.001
a_features3 = afeature_importance_df.head(9)['Feature'].values
X_train_a_selected3 = pd.DataFrame(scaler.fit_transform(X_train[a_features3]), columns=a_features3)
X_test_a_selected3 = pd.DataFrame(scaler.transform(X_test[a_features3]), columns=a_features3)
In [24]:
# RF Model
predictiona = random_forest_model.predict(X_test)

# Random Forest 1
model_a_top19 = RandomForestClassifier(random_state=42)
model_a_top19.fit(X_train_a_selected1, y_train)
predictions_a_top19 = model_a_top19.predict(X_test_a_selected1)

# Random Forest 2
model_a_top13 = RandomForestClassifier(random_state=42)
model_a_top13.fit(X_train_a_selected2, y_train)
predictions_a_top13 = model_a_top13.predict(X_test_a_selected2)

# Random Forest 3
model_a_top9 = RandomForestClassifier(random_state=42)
model_a_top9.fit(X_train_a_selected3, y_train)
predictions_a_top9 = model_a_top9.predict(X_test_a_selected3)
In [28]:
# Evaluation for RF model
accuracy_a = accuracy_score(y_test, predictiona)
f1_a = f1_score(y_test, predictiona, average='weighted')
roc_auc_a = roc_auc_score(y_test, random_forest_model.predict_proba(X_test)[:, 1])

# Evaluation for top 19 features
accuracy_a_top19 = accuracy_score(y_test, predictions_a_top19)
f1_a_top19 = f1_score(y_test, predictions_a_top19)
roc_auc_a_top19 = roc_auc_score(y_test, model_a_top19.predict_proba(X_test_a_selected1)[:, 1])

# Evaluation for top 13 features
accuracy_a_top13 = accuracy_score(y_test, predictions_a_top13)
f1_a_top13 = f1_score(y_test, predictions_a_top13)
roc_auc_a_top13 = roc_auc_score(y_test, model_a_top13.predict_proba(X_test_a_selected2)[:, 1])

# Evaluation for top 9 features
accuracy_a_top9 = accuracy_score(y_test, predictions_a_top9)
f1_a_top9 = f1_score(y_test, predictions_a_top9)
roc_auc_a_top9 = roc_auc_score(y_test, model_a_top9.predict_proba(X_test_a_selected3)[:, 1])


print("RF Model - Accuracy:", accuracy_a, "F1 Score:", f1_a, "ROC-AUC:", roc_auc_a)
print("Model with Top 19 Features - Accuracy:", accuracy_a_top19, "F1 Score:", f1_a_top19, "ROC-AUC:", roc_auc_a_top19)
print("Model with Top 13 Features - Accuracy:", accuracy_a_top13, "F1 Score:", f1_a_top13, "ROC-AUC:", roc_auc_a_top13)
print("Model with Top 9 Features - Accuracy:", accuracy_a_top9, "F1 Score:", f1_a_top9, "ROC-AUC:", roc_auc_a_top9)

## 3. XBGoost
In [7]:
model3 = XGBClassifier(random_state=42)
model3.fit(X_train, y_train)

### 3.1 Features Selection
In [8]:
xgb_importance = model3.feature_importances_

xgb_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': xgb_importance})
xgb_feature_importance_df = xgb_feature_importance_df.sort_values(by='Importance', ascending=False)
xgb_feature_importance_df

# All features have importance
```

```
top_xgb_features1 = xgb_feature_importance_df.head(30)['Feature']
X_train_xgb_selected1 = X_train[top_xgb_features1]
X_test_xgb_selected1 = X_test[top_xgb_features1]

# Features have importance > 0.013
top_xgb_features2 = xgb_feature_importance_df.head(20)['Feature']
X_train_xgb_selected2 = X_train[top_xgb_features2]
X_test_xgb_selected2 = X_test[top_xgb_features2]

# Features have importance > 0.015
top_xgb_features3 = xgb_feature_importance_df.head(15)['Feature']
X_train_xgb_selected3 = X_train[top_xgb_features3]
X_test_xgb_selected3 = X_test[top_xgb_features3]

# Features have importance > 0.02
top_xgb_features4 = xgb_feature_importance_df.head(8)['Feature']
X_train_xgb_selected4 = X_train[top_xgb_features4]
X_test_xgb_selected4 = X_test[top_xgb_features4]

# Features have importance > 0.1
top_xgb_features5 = xgb_feature_importance_df.head(5)['Feature']
X_train_xgb_selected5 = X_train[top_xgb_features5]
X_test_xgb_selected5 = X_test[top_xgb_features5]
```

3.2 Evaluate Models

In [10]:

```
# Original Model
predictions3 = model3.predict(X_test)

# XGBoost 1
model_xgb_top30 = XGBClassifier(random_state=42)
model_xgb_top30.fit(X_train_xgb_selected1, y_train)
predictions_xgb_top30 = model_xgb_top30.predict(X_test_xgb_selected1)

# XGBoost 2
model_xgb_top20 = XGBClassifier(random_state=42)
model_xgb_top20.fit(X_train_xgb_selected2, y_train)
predictions_xgb_top20 = model_xgb_top20.predict(X_test_xgb_selected2)

# XGBoost 3
model_xgb_top15 = XGBClassifier(random_state=42)
model_xgb_top15.fit(X_train_xgb_selected3, y_train)
predictions_xgb_top15 = model_xgb_top15.predict(X_test_xgb_selected3)

# XGBoost 4
model_xgb_top8 = XGBClassifier(random_state=42)
model_xgb_top8.fit(X_train_xgb_selected4, y_train)
predictions_xgb_top8 = model_xgb_top8.predict(X_test_xgb_selected4)

# XGBoost 5
model_xgb_top5 = XGBClassifier(random_state=42)
model_xgb_top5.fit(X_train_xgb_selected5, y_train)
predictions_xgb_top5 = model_xgb_top5.predict(X_test_xgb_selected5)
```

In [13]:

```
# Evaluation for original model
accuracy_3 = accuracy_score(y_test, predictions3)
f1_3 = f1_score(y_test, predictions3)
roc_auc_3 = roc_auc_score(y_test, model3.predict_proba(X_test)[:, 1])

# Evaluation for top 30 features
accuracy_xgb_top30 = accuracy_score(y_test, predictions_xgb_top30)
f1_xgb_top30 = f1_score(y_test, predictions_xgb_top30)
roc_auc_xgb_top30 = roc_auc_score(y_test, model_xgb_top30.predict_proba(X_test_xgb_selected1)[:, 1])

# Evaluation for top 20 features
accuracy_xgb_top20 = accuracy_score(y_test, predictions_xgb_top20)
f1_xgb_top20 = f1_score(y_test, predictions_xgb_top20)
roc_auc_xgb_top20 = roc_auc_score(y_test, model_xgb_top20.predict_proba(X_test_xgb_selected2)[:, 1])

# Evaluation for top 15 features
accuracy_xgb_top15 = accuracy_score(y_test, predictions_xgb_top15)
f1_xgb_top15 = f1_score(y_test, predictions_xgb_top15)
```

```
roc_auc_xgb_top15 = roc_auc_score(y_test, model_xgb_top15.predict_proba(X_test_xgb_selected3)[:, 1])

# Evaluation for top 8 features
accuracy_xgb_top8 = accuracy_score(y_test, predictions_xgb_top8)
f1_xgb_top8 = f1_score(y_test, predictions_xgb_top8)
roc_auc_xgb_top8 = roc_auc_score(y_test, model_xgb_top8.predict_proba(X_test_xgb_selected4)[:, 1])

# Evaluation for top 5 features
accuracy_xgb_top5 = accuracy_score(y_test, predictions_xgb_top5)
f1_xgb_top5 = f1_score(y_test, predictions_xgb_top5)
roc_auc_xgb_top5 = roc_auc_score(y_test, model_xgb_top5.predict_proba(X_test_xgb_selected5)[:, 1])

auc_scores.append(auc)
accuracy_scores.append(accuracy)
f1_scores.append(f1)

mean_auc = sum(auc_scores) / len(auc_scores)
mean_accuracy = sum(accuracy_scores) / len(accuracy_scores)
mean_f1 = sum(f1_scores) / len(f1_scores)

print("Original Model - Accuracy:", accuracy_3, "F1 Score:", f1_3, "ROC-AUC:", roc_auc_3)
print("Model with Top 30 Features - Accuracy:", accuracy_xgb_top30, "F1 Score:", f1_xgb_top30, "ROC-AUC:", roc_auc_xgb_top30)
print("Model with Top 20 Features - Accuracy:", accuracy_xgb_top20, "F1 Score:", f1_xgb_top20, "ROC-AUC:", roc_auc_xgb_top20)
print("Model with Top 15 Features - Accuracy:", accuracy_xgb_top15, "F1 Score:", f1_xgb_top15, "ROC-AUC:", roc_auc_xgb_top15)
print("Model with Top 8 Features - Accuracy:", accuracy_xgb_top8, "F1 Score:", f1_xgb_top8, "ROC-AUC:", roc_auc_xgb_top8)
print("Model with Top 5 Features - Accuracy:", accuracy_xgb_top5, "F1 Score:", f1_xgb_top5, "ROC-AUC:", roc_auc_xgb_top5)

#### 3.2.2 5-fold Cross Validation
In [15]:
model3 = XGBClassifier(random_state=42)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    model3.fit(X_train, y_train)

    predictions3 = model3.predict(X_test)
    proba_predictions3 = model3.predict_proba(X_test)[:, 1]

    accuracy = accuracy_score(y_test, predictions3)
    f1 = f1_score(y_test, predictions3)
    precision = precision_score(y_test, predictions3)
    recall = recall_score(y_test, predictions3)
    auc = roc_auc_score(y_test, proba_predictions3)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
```

```
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

model3 = XGBClassifier(random_state=42)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

# top30
for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    model3.fit(X_train, y_train)

    # Feature importance
    feature_importances = model3.feature_importances_
    xgb_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': feature_importances})
    xgb_feature_importance_df = xgb_feature_importance_df.sort_values(by='Importance', ascending=False)

    # Select top features
    top_xgb_features1 = xgb_feature_importance_df.head(30)['Feature']
    X_train_xgb_selected1 = X_train[top_xgb_features1]
    X_test_xgb_selected1 = X_test[top_xgb_features1]

    # Train model on selected features
    model_xgb_top30 = XGBClassifier(random_state=42)
    model_xgb_top30.fit(X_train_xgb_selected1, y_train)
    predictions_xgb_top30 = model_xgb_top30.predict(X_test_xgb_selected1)
    proba_predictions_top30 = model_xgb_top30.predict_proba(X_test_xgb_selected1)[:, 1]

    accuracy = accuracy_score(y_test, predictions_xgb_top30)
    f1 = f1_score(y_test, predictions_xgb_top30)
    precision = precision_score(y_test, predictions_xgb_top30)
    recall = recall_score(y_test, predictions_xgb_top30)
    auc = roc_auc_score(y_test, proba_predictions_top30)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

model3 = XGBClassifier(random_state=42)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
```

```
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

# top20

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    model3.fit(X_train, y_train)

    # Feature importance
    feature_importances = model3.feature_importances_
    xgb_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': feature_importances})
    xgb_feature_importance_df = xgb_feature_importance_df.sort_values(by='Importance', ascending=False)

    # Select top features
    top_xgb_features2 = importance_df.head(20)['Feature']
    X_train_xgb_selected2 = X_train[top_xgb_features2]
    X_test_xgb_selected2 = X_test[top_xgb_features2]

    # Train model on selected features
    model_xgb_top20 = XGBClassifier(random_state=42)
    model_xgb_top20.fit(X_train_xgb_selected2, y_train)
    predictions_xgb_top20 = model_xgb_top20.predict(X_test_xgb_selected2)
    proba_predictions_top20 = model_xgb_top20.predict_proba(X_test_xgb_selected2)[:, 1]

    accuracy = accuracy_score(y_test, predictions_xgb_top20)
    f1 = f1_score(y_test, predictions_xgb_top20)
    precision = precision_score(y_test, predictions_xgb_top20)
    recall = recall_score(y_test, predictions_xgb_top20)
    auc = roc_auc_score(y_test, proba_predictions_top20)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

model3 = XGBClassifier(random_state=42)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

# top15

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
```

```
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    model3.fit(X_train, y_train)

    # Feature importance
    feature_importances = model3.feature_importances_
    xgb_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': feature_importances})
    xgb_feature_importance_df = xgb_feature_importance_df.sort_values(by='Importance', ascending=False)

    # Select top features
    top_xgb_features3 = importance_df.head(15)['Feature']
    X_train_xgb_selected3 = X_train[top_xgb_features3]
    X_test_xgb_selected3 = X_test[top_xgb_features3]

    # Train model on selected features
    model_xgb_top15 = XGBClassifier(random_state=42)
    model_xgb_top15.fit(X_train_xgb_selected3, y_train)
    predictions_xgb_top15 = model_xgb_top15.predict(X_test_xgb_selected3)
    proba_predictions_top15 = model_xgb_top15.predict_proba(X_test_xgb_selected3)[:, 1]

    accuracy = accuracy_score(y_test, predictions_xgb_top15)
    f1 = f1_score(y_test, predictions_xgb_top15)
    precision = precision_score(y_test, predictions_xgb_top15)
    recall = recall_score(y_test, predictions_xgb_top15)
    auc = roc_auc_score(y_test, proba_predictions_top15)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

model3 = XGBClassifier(random_state=42)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

# top8

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    model3.fit(X_train, y_train)

    # Feature importance
    feature_importances = model3.feature_importances_
    xgb_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': feature_importances})
```

```
    xgb_feature_importance_df = xgb_feature_importance_df.sort_values(by='Importance', ascending=False)

    # Select top features
    top_xgb_features4 = importance_df.head(8)['Feature']
    X_train_xgb_selected4 = X_train[top_xgb_features4]
    X_test_xgb_selected4 = X_test[top_xgb_features4]

    # Train model on selected features
    model_xgb_top8 = XGBClassifier(random_state=42)
    model_xgb_top8.fit(X_train_xgb_selected4, y_train)
    predictions_xgb_top8 = model_xgb_top8.predict(X_test_xgb_selected4)
    proba_predictions_top8 = model_xgb_top8.predict_proba(X_test_xgb_selected4)[:, 1]

    accuracy = accuracy_score(y_test, predictions_xgb_top8)
    f1 = f1_score(y_test, predictions_xgb_top8)
    precision = precision_score(y_test, predictions_xgb_top8)
    recall = recall_score(y_test, predictions_xgb_top8)
    auc = roc_auc_score(y_test, proba_predictions_top8)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

model3 = XGBClassifier(random_state=42)

kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

# top5

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    model3.fit(X_train, y_train)

    # Feature importance
    feature_importances = model3.feature_importances_
    xgb_feature_importance_df = pd.DataFrame({'Feature': X_train.columns, 'Importance': feature_importances})
    xgb_feature_importance_df = xgb_feature_importance_df.sort_values(by='Importance', ascending=False)

    # Select top features
    top_xgb_features5 = importance_df.head(5)['Feature']
    X_train_xgb_selected5 = X_train[top_xgb_features5]
    X_test_xgb_selected5 = X_test[top_xgb_features5]

    # Train model on selected features
    model_xgb_top5 = XGBClassifier(random_state=42)
```

```
    model_xgb_top5.fit(X_train_xgb_selected5, y_train)
    predictions_xgb_top5 = model_xgb_top5.predict(X_test_xgb_selected5)
    proba_predictions_top5 = model_xgb_top5.predict_proba(X_test_xgb_selected5)[:, 1]

    accuracy = accuracy_score(y_test, predictions_xgb_top5)
    f1 = f1_score(y_test, predictions_xgb_top5)
    precision = precision_score(y_test, predictions_xgb_top5)
    recall = recall_score(y_test, predictions_xgb_top5)
    auc = roc_auc_score(y_test, proba_predictions_top5)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)
```

4. Support Vector Machine

In [38]:

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

In [43]:

```
from sklearn.svm import SVC
svm_rbf = SVC(kernel='rbf', C=1.0, gamma='scale', class_weight='balanced', random_state=42)
```

In [44]:

```
svm_rbf.fit(X_scaled, y)

from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.model_selection import cross_validate
from sklearn.metrics import make_scorer, accuracy_score, roc_auc_score, precision_score, recall_score, f1_score

# Define the pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('svm', SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42, probability=True))  # Set probability=True
])

scoring = {
    'accuracy': 'accuracy',
    'precision': make_scorer(precision_score),
    'recall': make_scorer(recall_score),
    'f1': make_scorer(f1_score),
    'roc_auc': make_scorer(roc_auc_score, needs_proba=True)
}

# Perform cross-validation
cv_results = cross_validate(pipeline, X, y, cv=5, scoring=scoring, return_train_score=False)

# Print results
print("Cross-Validation Results:")
print("Mean Accuracy:", cv_results['test_accuracy'].mean())
print("Mean Precision:", cv_results['test_precision'].mean())
print("Mean Recall:", cv_results['test_recall'].mean())
print("Mean F1 Score:", cv_results['test_f1'].mean())
print("Mean ROC-AUC:", cv_results['test_roc_auc'].mean())
```

```
kf = KFold(n_splits=5, shuffle=True, random_state=42)

auc_scores = []
accuracy_scores = []
f1_scores = []
precision_scores = []
recall_scores = []

for fold, (train_index, test_index) in enumerate(kf.split(X)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # train the original model
    model3.fit(X_train, y_train)

    predictions3 = model3.predict(X_test)
    proba_predictions3 = model3.predict_proba(X_test)[:, 1]

    accuracy = accuracy_score(y_test, predictions3)
    f1 = f1_score(y_test, predictions3)
    precision = precision_score(y_test, predictions3)
    recall = recall_score(y_test, predictions3)
    auc = roc_auc_score(y_test, proba_predictions3)

    accuracy_scores.append(accuracy)
    f1_scores.append(f1)
    auc_scores.append(auc)
    precision_scores.append(precision)
    recall_scores.append(recall)

# Compute mean metrics
mean_accuracy = np.mean(accuracy_scores)
mean_f1 = np.mean(f1_scores)
mean_auc = np.mean(auc_scores)
mean_precision = np.mean(precision_scores)
mean_recall = np.mean(recall_scores)

print("\nK-Fold Cross-Validation Results:")
print("Mean Accuracy:", mean_accuracy)
print("Mean Precision", mean_precision)
print("Mean Recall", mean_recall)
print("Mean F1 Score:", mean_f1)
print("Mean ROC-AUC:", mean_auc)

model4 = LinearSVC(C=1.0, random_state=42)
In [9]:
model4.fit(X_train, y_train)
# Original Model
predictions4 = model4.predict(X_test)

# SVM using RF top 17 features
model_svm_top17 = LinearSVC(C=1.0, random_state=42)
model_svm_top17.fit(X_train_rf_selected1, y_train)
predictions_svm_top17 = model_svm_top17.predict(X_test_rf_selected1)

# SVM using RF top 11 features
model_svm_top11 = LinearSVC(C=1.0, random_state=42)
model_svm_top11.fit(X_train_rf_selected2, y_train)
predictions_svm_top11 = model_svm_top11.predict(X_test_rf_selected2)

# SVM using RF top 5 features
model_svm_top5 = LinearSVC(C=1.0, random_state=42)
model_svm_top5.fit(X_train_rf_selected3, y_train)
predictions_svm_top5 = model_svm_top5.predict(X_test_rf_selected3)
## 4.1 Model Evaluation
In [28]:
# Evaluation for original model
accuracy_4 = accuracy_score(y_test, predictions4)
f1_4 = f1_score(y_test, predictions4, average='weighted')
```

```
# Evaluation for top 17 features
accuracy_svm_top17 = accuracy_score(y_test, predictions_svm_top17)
f1_svm_top17 = f1_score(y_test, predictions_svm_top17, average='weighted')

# Evaluation for top 11 features
accuracy_svm_top11 = accuracy_score(y_test, predictions_svm_top11)
f1_svm_top11 = f1_score(y_test, predictions_svm_top11, average='weighted')

# Evaluation for top 5 features
accuracy_svm_top5 = accuracy_score(y_test, predictions_svm_top5)
f1_svm_top5 = f1_score(y_test, predictions_svm_top5, average='weighted')

print("Original Model - Accuracy:", accuracy_4, "F1 Score:", f1_4)
print("Model with Top 17 Features - Accuracy:", accuracy_svm_top17, "F1 Score:", f1_svm_top17)
print("Model with Top 11 Features - Accuracy:", accuracy_svm_top11, "F1 Score:", f1_svm_top11)
print("Model with Top 5 Features - Accuracy:", accuracy_svm_top5, "F1 Score:", f1_svm_top5)

## 5. Reduce Multicollinearity
### 5.1 VIF
In [7]:
X_vif = X_train.iloc[:, 1:]

# Calculate VIF for each feature
vif_data = pd.DataFrame()
vif_data["Feature"] = X_vif.columns
vif_data["VIF"] = [variance_inflation_factor(X_vif.values, i) for i in range(X_vif.shape[1])]

vif_data

# Create a new DataFrame with only selected features
X_train1 = X_train.drop(['SystolicBP', 'DiastolicBP', 'CholesterolTotal', 'CholesterolLDL'], axis = 1)
X_test1 = X_test.drop(['SystolicBP', 'DiastolicBP', 'CholesterolTotal', 'CholesterolLDL'], axis = 1)
test_X1 = test_X.drop(['SystolicBP', 'DiastolicBP', 'CholesterolTotal', 'CholesterolLDL'], axis=1)
### 5.1.1 Redo Random Forest
In [9]:
model5 = RandomForestClassifier(
    max_depth=10,
    max_features=None,
    min_samples_leaf=2,
    min_samples_split=5,
    n_estimators=100,
    random_state=42
)

# Fit on the training data
model5.fit(X_train1, y_train)

rf_feature_importance_df1 = pd.DataFrame({'Feature': X_train1.columns, 'Importance': model5.feature_importances_})
rf_feature_importance_df1 = rf_feature_importance_df1.sort_values(by='Importance', ascending=False)
rf_feature_importance_df1

# Features have importance > 0.001
top_new_features1 = rf_feature_importance_df1.head(15)['Feature'].values
X_train_new_selected1 = pd.DataFrame(scaler.fit_transform(X_train1[top_new_features1]), columns=top_new_features1)
X_test_new_selected1 = pd.DataFrame(scaler.transform(X_test1[top_new_features1]), columns=top_new_features1)

# Features have importance > 0.01
top_new_features2 = rf_feature_importance_df1.head(9)['Feature'].values
X_train_new_selected2 = pd.DataFrame(scaler.fit_transform(X_train1[top_new_features2]), columns=top_new_features2)
X_test_new_selected2 = pd.DataFrame(scaler.transform(X_test1[top_new_features2]), columns=top_new_features2)

# Features have importance > 0.1
top_new_features3 = rf_feature_importance_df1.head(5)['Feature'].values
X_train_new_selected3 = pd.DataFrame(scaler.fit_transform(X_train1[top_new_features3]), columns=top_new_features3)
X_test_new_selected3 = pd.DataFrame(scaler.transform(X_test1[top_new_features3]), columns=top_new_features3)
In [12]:
# New RF Model
predictions5 = model5.predict(X_test1)

# New Random Forest 1
model_new_top15 = RandomForestClassifier(random_state=42)
```

```
model_new_top15.fit(X_train_new_selected1, y_train)
predictions_new_top15 = model_new_top15.predict(X_test_new_selected1)

# New Random Forest 2
model_new_top9 = RandomForestClassifier(random_state=42)
model_new_top9.fit(X_train_new_selected2, y_train)
predictions_new_top9 = model_new_top9.predict(X_test_new_selected2)

# New Random Forest 3
model_new_top5 = RandomForestClassifier(random_state=42)
model_new_top5.fit(X_train_new_selected3, y_train)
predictions_new_top5 = model_new_top5.predict(X_test_new_selected3)
In [13]:
# Evaluation for new RF model
accuracy_5 = accuracy_score(y_test, predictions5)
f1_5 = f1_score(y_test, predictions5, average='weighted')
roc_auc_5 = roc_auc_score(y_test, model5.predict_proba(X_test1)[:, 1])

# Evaluation for new top 15 features
accuracy_new_top15 = accuracy_score(y_test, predictions_new_top15)
f1_new_top15 = f1_score(y_test, predictions_new_top15)
roc_auc_new_top15 = roc_auc_score(y_test, model_new_top15.predict_proba(X_test_new_selected1)[:, 1])

# Evaluation for new top 9 features
accuracy_new_top9 = accuracy_score(y_test, predictions_new_top9)
f1_new_top9 = f1_score(y_test, predictions_new_top9)
roc_auc_new_top9 = roc_auc_score(y_test, model_new_top9.predict_proba(X_test_new_selected2)[:, 1])

# Evaluation for new top 5 features
accuracy_new_top5 = accuracy_score(y_test, predictions_new_top5)
f1_new_top5 = f1_score(y_test, predictions_new_top5)
roc_auc_new_top5 = roc_auc_score(y_test, model_new_top5.predict_proba(X_test_new_selected3)[:, 1])


print("New RF Model - Accuracy:", accuracy_5, "F1 Score:", f1_5, "ROC-AUC:", roc_auc_5)
print("New Model with Top 15 Features - Accuracy:", accuracy_new_top15, "F1 Score:", f1_new_top15, "ROC-AUC:", roc_auc_new_top15)
print("New Model with Top 9 Features - Accuracy:", accuracy_new_top9, "F1 Score:", f1_new_top9, "ROC-AUC:", roc_auc_new_top9)
print("New Model with Top 5 Features - Accuracy:", accuracy_new_top5, "F1 Score:", f1_new_top5, "ROC-AUC:", roc_auc_new_top5)

predictions = random_forest_model.predict(test_X)

output = pd.DataFrame({'PatientID': test_df['PatientID'], 'Diagnosis': predictions})
output.head(50)

# test_X1 = test_X[best_rf_feature_importance_df]
predictions = model_best_top15.predict(test_X[best_rf_feature_importance_df.head(15)['Feature']])
# scaler.transform(X_test[a_features1]
# best_rf_feature_importance_df.head(19)['Feature']
# Create a DataFrame with PatientID and the predictions
output1 = pd.DataFrame({'PatientID': test_df['PatientID'], 'Diagnosis': predictions})
# output1.head(50)
output1.to_csv('predictions1.csv', index=False)
```