In my project, the initial step involved loading the dataset to begin the analysis. I used Python's Pandas library, which is efficient for data manipulation and analysis. With Pandas, I imported the dataset from a CSV file. This approach allows me to quickly access and inspect the data. Loading the data was as simple as using pd.read_csv(), which read the contents of the CSV file into a DataFrame - a two-dimensional data structure with labeled axes (rows and columns).

After loading the data, I needed to split it into training and validation sets. This is a crucial step in any machine learning project as it helps in validating the model's performance and prevents overfitting. I utilized Scikit-learn's train_test_split function for this purpose. This function provides a fast and efficient way to divide the data into separate sets - typically, a larger portion for training and a smaller portion for validation. I decided on an 80-20 split, allocating 80% of the data for training and 20% for validation. This split ratio is widely used in machine learning tasks as it usually provides a sufficient amount of data for learning while keeping enough data aside to validate and test the model's generalizability to new data.

Imbalanced datasets can lead to biased models, as they tend to favor the majority class. To address this, I employed a technique known as Synthetic Minority Over-sampling Technique (SMOTE). SMOTE works by creating synthetic samples from the minority class, thus balancing the class distribution. It's crucial to apply this technique only to the training data to avoid introducing artificial bias in the model's validation phase as the requirement describes. This approach helped me ensure that the model learned to recognize patterns from both classes equally, improving its ability to generalize and make accurate predictions on unseen data.

After finishing the above data preparation, I started the model training. The reason I did those seven classifications is that I wanted to find the best three models to match the requirements for doing the prediction. And based on the result, I did an analysis for each model.

Logistic Regression

Pros: Simple and interpretable.

Performs well when the relationship between features and the target variable is approximately linear.

Cons: In this case, it completely failed to predict Class 1 (bankruptcy cases), as indicated by an F1-Score of 0.0000 for Class 1.

The rationale for Choosing: It was chosen for its simplicity and interpretability. However, its inability to predict the minority class makes it less suitable for the final selection.

K-Nearest Neighbors (KNN)

Pros: Intuitive and non-parametric, good for classification when the decision boundary is irregular.

Competitive cross-validation accuracy.

Cons: Like Logistic Regression, it completely failed to predict Class 1. Can be computationally expensive with large datasets.

Rationale for Choosing: KNN was selected for its effectiveness in capturing non-linear relationships. Nevertheless, its poor performance in predicting the minority class is a significant drawback.

Decision Tree

Pros: Highly interpretable. Can capture non-linear relationships between features and the target.

Cons: Moderate success in predicting Class 1, better than the previous models but still relatively low.

Cons: Prone to overfitting. Lower cross-validation accuracy compared to other models.

Rationale for Choosing: Its ability to model non-linear relationships and interpretability were key factors. The moderate success in predicting Class 1 suggests some potential but might require further tuning or ensemble methods for improvement.

Support Vector Machine (SVM)

Pros: Effective in high-dimensional spaces. Good for cases where the margin of separation is clear.

Cons: Failed to predict Class 1, similar to Logistic Regression and KNN. Can be computationally intensive with large datasets and complex kernels.

Rationale for Choosing: SVM was chosen for its effectiveness in high-dimensional spaces and versatility with different kernels. However, its performance in the minority class was disappointing.

Gradient Boosting

Pros: Robust to overfitting. Good performance in both accuracy and AUC, indicating a strong ability to differentiate between classes.

Cons: Can be slow to train due to the sequential nature of boosting. Requires careful tuning to prevent overfitting.

Rationale for Choosing: Its strong performance across most metrics, including a reasonable F1-Score for Class 1, makes it a strong candidate for the final selection.

XGBoost

Pros: High performance across all metrics. Excellent in handling various types of data, scalable and efficient.

Cons: Like Gradient Boosting, it can be complex and requires careful tuning. Less interpretable compared to simpler models.

Rationale for Choosing: Chosen for its high efficiency and effectiveness in both accuracy and AUC. Its good performance in predicting the minority class makes it a top contender.

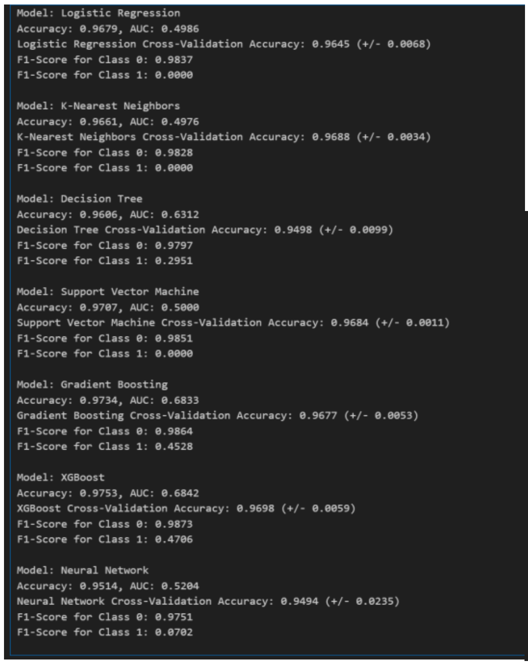
Neural Network

Pros: Highly flexible and capable of modeling complex nonlinear relationships.

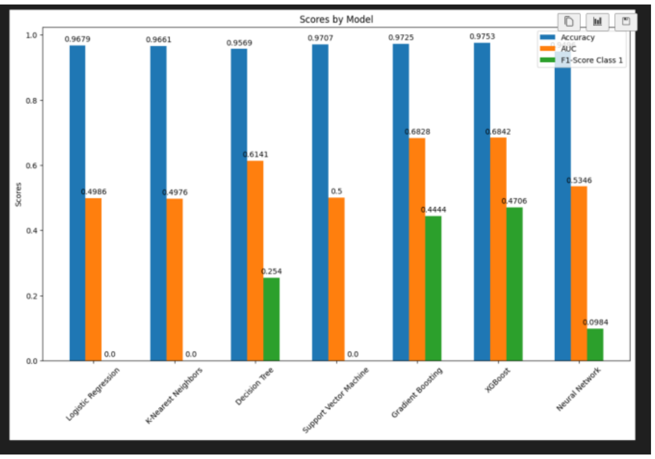
Cons: Managed to predict Class 1, albeit with a low F1-Score. Requires careful architecture design and tuning. Less interpretable and can overfit if not regularized properly.

Rationale for Choosing: Its flexibility and potential in capturing complex patterns were the main reasons for its selection. However, its lower performance compared to ensemble methods and difficulty in tuning are notable.

Based on the accuracy, AUC, and particularly the F1-Scores for both classes, I identified XGBoost, Gradient Boosting, and Decision Tree as the top three models. These models showed a better balance in predicting both classes, which is crucial for our objective of accurately identifying cases of bankruptcy. XGBoost and Gradient Boosting, in particular, demonstrated strong overall performance, while the Decision Tree provided a reasonable trade-off between performance and interpretability.



```
Model: Logistic Regression
Accuracy: 0.9679, AUC: 0.4986
Logistic Regression Cross-Validation Accuracy: 0.9645 (+/- 0.0068)
F1-Score for Class 0: 0.9837
F1-Score for Class 1: 0.0000

Model: K-Nearest Neighbors
Accuracy: 0.9661, AUC: 0.4976
K-Nearest Neighbors Cross-Validation Accuracy: 0.9688 (+/- 0.0034)
F1-Score for Class 0: 0.9828
F1-Score for Class 1: 0.0000

Model: Decision Tree
Accuracy: 0.9606, AUC: 0.6312
Decision Tree Cross-Validation Accuracy: 0.9498 (+/- 0.0099)
F1-Score for Class 0: 0.9797
F1-Score for Class 1: 0.2951

Model: Support Vector Machine
Accuracy: 0.9707, AUC: 0.5000
Support Vector Machine Cross-Validation Accuracy: 0.9684 (+/- 0.0011)
F1-Score for Class 0: 0.9851
F1-Score for Class 1: 0.0000

Model: Gradient Boosting
Accuracy: 0.9734, AUC: 0.6833
Gradient Boosting Cross-Validation Accuracy: 0.9677 (+/- 0.0053)
F1-Score for Class 0: 0.9864
F1-Score for Class 1: 0.4528

Model: XGBoost
Accuracy: 0.9753, AUC: 0.6842
XGBoost Cross-Validation Accuracy: 0.9698 (+/- 0.0059)
F1-Score for Class 0: 0.9873
F1-Score for Class 1: 0.4706

Model: Neural Network
Accuracy: 0.9514, AUC: 0.5204
Neural Network Cross-Validation Accuracy: 0.9494 (+/- 0.0235)
F1-Score for Class 0: 0.9751
F1-Score for Class 1: 0.0702
```



In my approach to selecting the best model and tune-tuning hyperparameters, I began by identifying three promising machine learning models: Decision Tree Classifier, Gradient Boosting Classifier, and XGBoost Classifier. For each model, I employed a systematic approach to identify the most effective set of hyperparameters. My aim was to optimize each model's performance, balancing between overfitting and underfitting, and ensuring robust generalization to unseen data.

Decision Tree Classifier:

Max Depth (max_depth): I set the maximum depth of the tree to control its complexity. A deeper tree can capture more detailed data patterns, but too much depth risks overfitting. Finding the right balance was crucial for model performance.

Minimum Samples for Split (min_samples_split): This parameter determines how many samples are needed to split a node. Setting it higher helps in preventing the model from learning overly specific patterns, thus avoiding overfitting.

Minimum Samples at Leaf Nodes (min_samples_leaf): By setting the minimum number of samples required at a leaf node, I ensured that the tree doesn't grow too complex and overfit the training data. It's a critical parameter for generalizing the model to new data.

Gradient Boosting Classifier:

Number of Estimators (n_estimators): This represents the number of trees in the ensemble. More trees can increase the model's ability to capture complex patterns, but too many trees can lead to overfitting.

Maximum Depth of Trees (max_depth): Similar to the Decision Tree, the depth of trees in Gradient Boosting was crucial. Deeper trees are more expressive but can easily overfit.

Minimum Samples for Split and Leaf Nodes (min_samples_split, min_samples_leaf): These parameters work to prevent overfitting by controlling the growth of each tree in the ensemble.

Learning Rate (learning_rate): A critical parameter in boosting, it scales the contribution of each tree. I had to balance between a learning rate that's too high, leading to rapid learning but potentially missing subtleties, and too low, requiring more trees but improving model robustness.

XGBoost Classifier:

Number of Estimators (n_estimators) and Maximum Depth (max_depth): As in Gradient Boosting, these parameters define the complexity of the model, influencing both its ability to learn from the training data and its capacity to generalize.

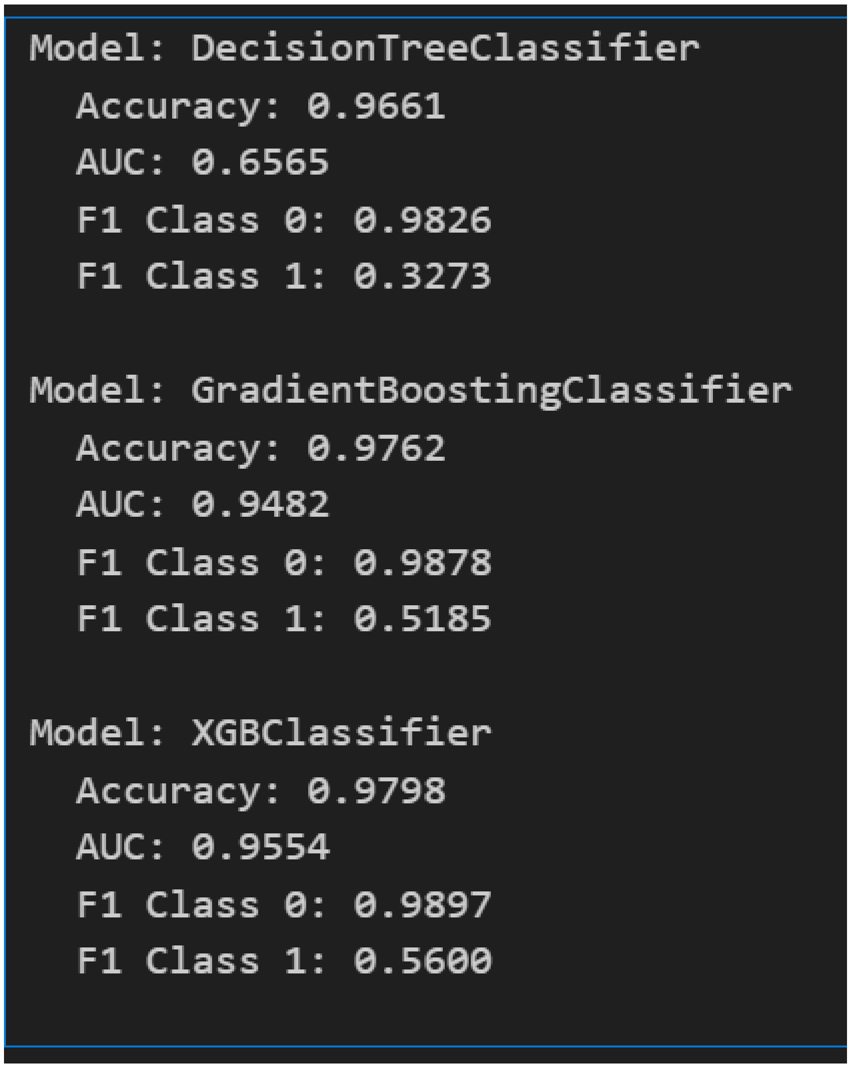
Minimum Child Weight (min_child_weight): This parameter helps to control overfitting by setting a threshold on the sum of instance weight (hessian) needed in a child.

Gamma (gamma): A regularization parameter that encourages simpler models, reducing the risk of overfitting.

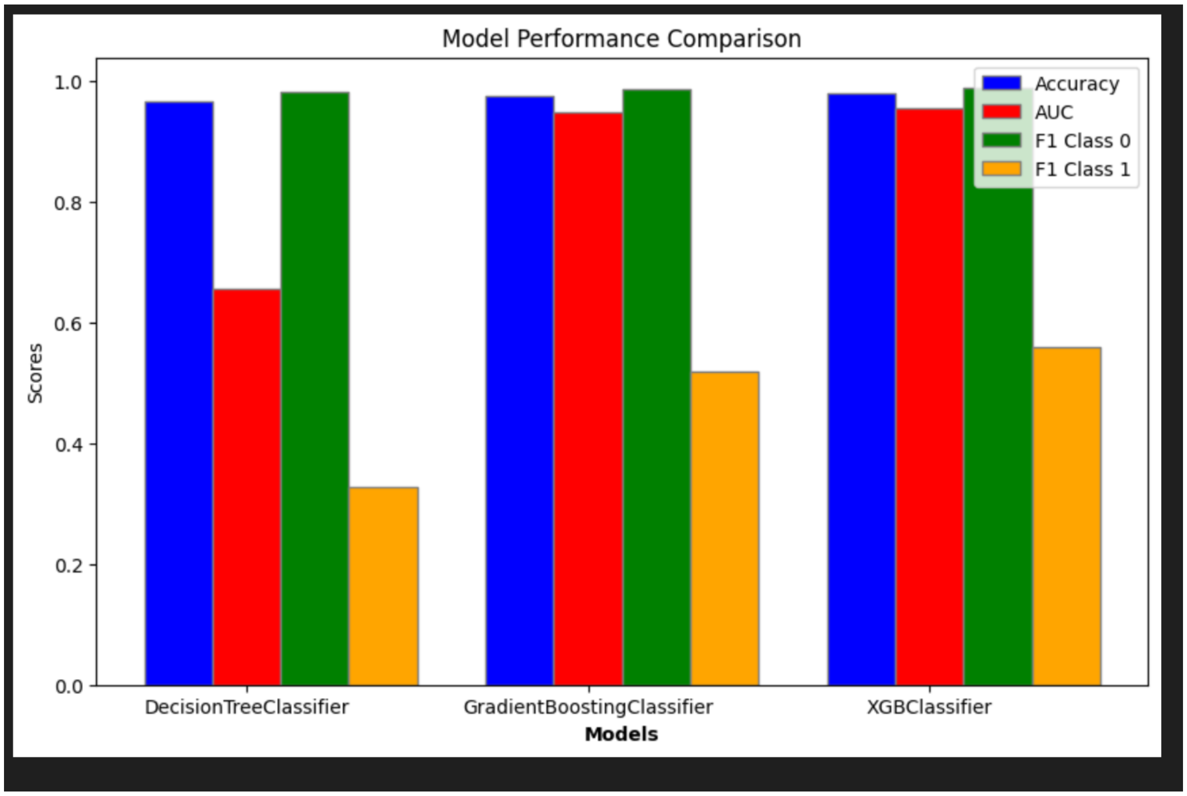
Learning Rate (learning_rate): The learning rate in XGBoost plays a similar role as in Gradient Boosting, affecting the speed and robustness of learning.

Subsample (subsample) and Column Sample by Tree (colsample_bytree): These parameters define the fraction of samples and features used for each tree, essential for regularization and preventing overfitting.

The tuning of these parameters was conducted through an exhaustive Grid Search with Cross-Validation, allowing me to systematically explore a range of values and combinations. The objective was to maximize each model's performance metrics while ensuring that they remained generalizable to unseen data.



```
Model: DecisionTreeClassifier
  Accuracy: 0.9661
  AUC: 0.6565
  F1 Class 0: 0.9826
  F1 Class 1: 0.3273

Model: GradientBoostingClassifier
  Accuracy: 0.9762
  AUC: 0.9482
  F1 Class 0: 0.9878
  F1 Class 1: 0.5185

Model: XGBClassifier
  Accuracy: 0.9798
  AUC: 0.9554
  F1 Class 0: 0.9897
  F1 Class 1: 0.5600
```



After completing the grid searches and analyzing the models' performances, I chose the XGBoost Classifier with its specified parameters for its superior performance in terms of accuracy, AUC, and F1 scores. This model not only showed a high level of accuracy but also demonstrated a strong ability to generalize, as evidenced by its performance on the validation set. The comprehensive parameter tuning played a crucial role in achieving this level of performance.