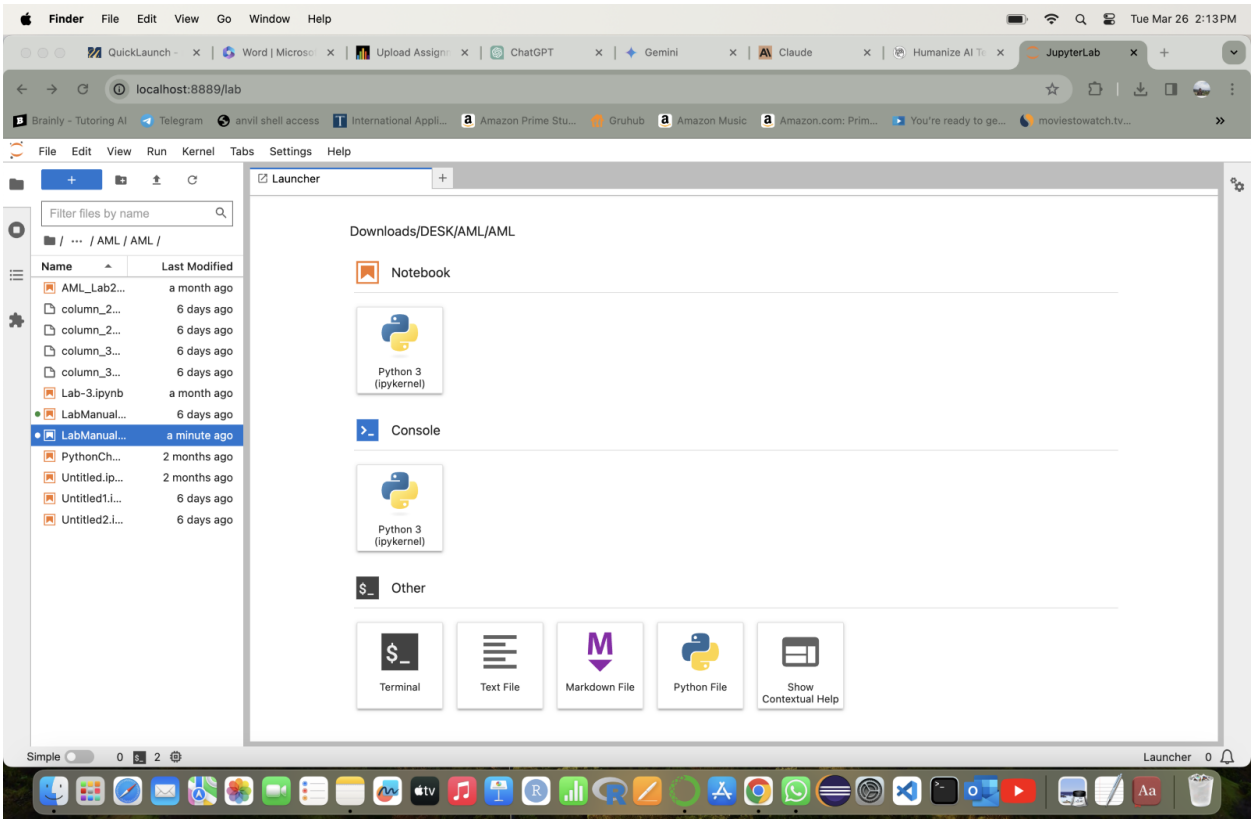
Jeevan Kumar Banoth - 02105145

Advanced machine learning

Homework – 7:

♦ Creating Jupyter Notebook with Anaconda navigator:



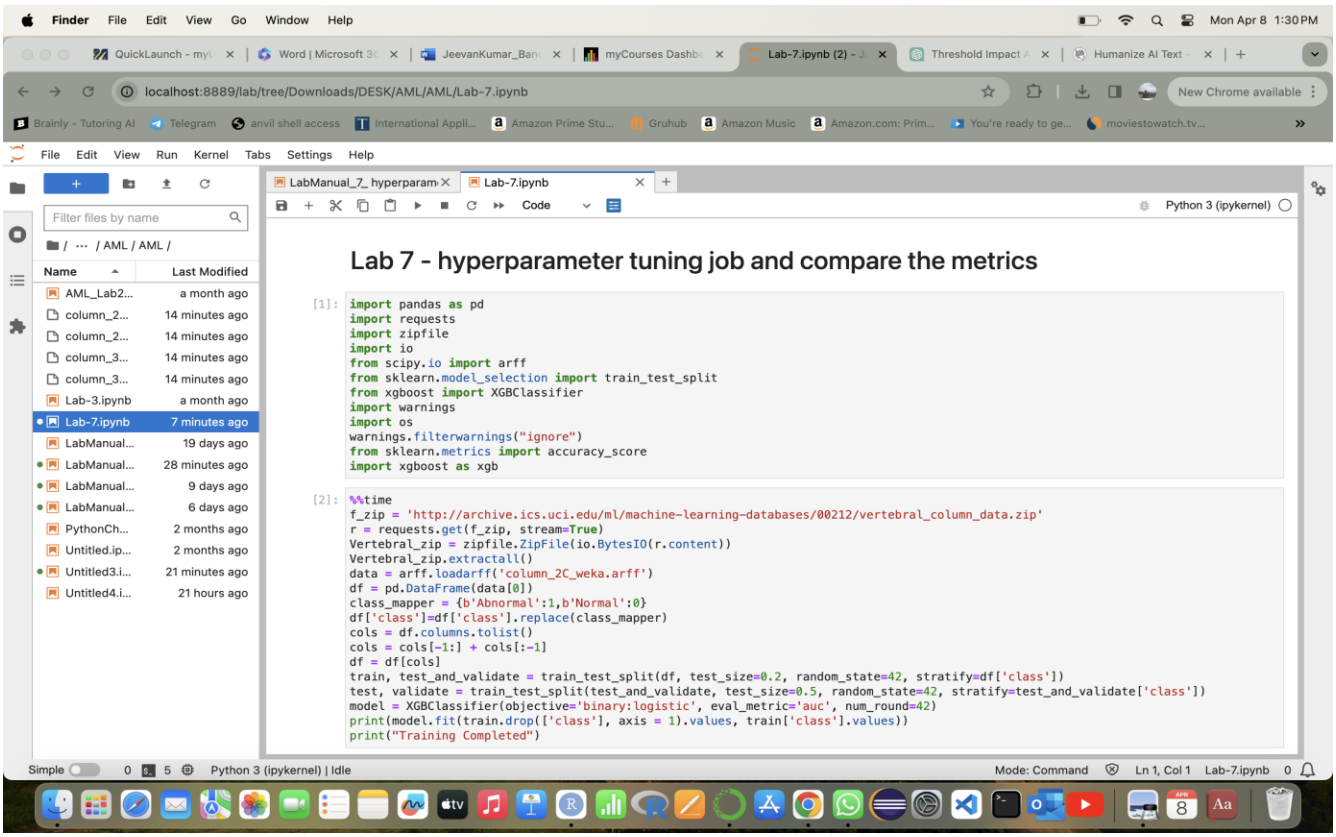
→ I started by using Anaconda navigator to open Jupyter lab and then directed it to my working folder.

→ In the notebook's left side, I found the files I had worked on before.

→ To begin this task, I created a new notebook by selecting File > New, Notebook, and then choosing conda_python3 in the kernel dialog window.

→ However, I realized there was no need to use the new notebook for this lab, so I decided to upload it instead.

→ Once the document was open, I made a new notebook with the python3 kernel to begin the lab.



→ To start the process, we import the necessary libraries in the script.

→ These libraries include pandas, requests, zipfile, io, scipy.io.arff, sklearn.model_selection, xgboost, and sklearn.metrics.

→ Next, the script proceeds to download a zip file that contains the "Vertebral Column Data" dataset from the UCI Machine Learning Repository.

→ It then extracts the data from the zip file and loads it into a Pandas DataFrame. For data preprocessing, the script carries out the following tasks:

- 1. It replaces the class labels "Abnormal" and "Normal" with numerical values 1 and 0 respectively.
- 2. It rearranges the column order by placing the "class" column at the end.

→ To split the dataset into training, testing, and validation sets, we use the train_test_split function from sklearn.model_selection.

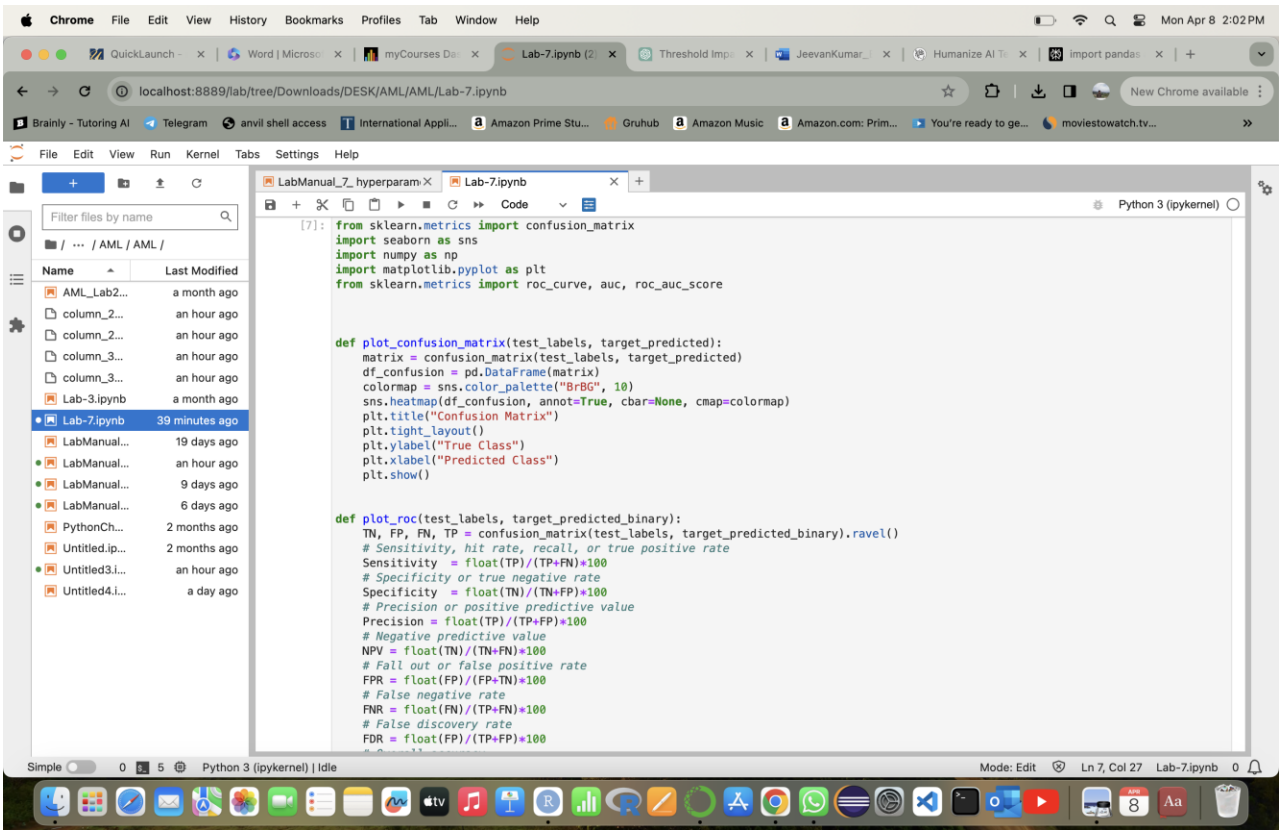
→ The split is made with a test size of 20% and a validation size of 50% of the test set, with a random state of 42 and stratified sampling based on the class labels.

→ For model training, we create an XGBoost classifier model with the following parameters:

- objective='binary:logistic': This sets the objective function to binary logistic regression.
- eval_metric='auc': This sets the evaluation metric to Area Under the Curve (AUC).
- num_round=42: This sets the number of boosting rounds to 42.

→ The model is then trained on the training data, and the training completion message is printed.

→ Timing: The script uses the %%time magic command to measure the execution time of the code.



→ Function to visualize the confusion matrix of a model:

→ To visualize the confusion matrix of a model, you can use the `plot_confusion_matrix` function that takes the true labels (`test_labels`) and the predicted labels (`target_predicted`) as input.

→ This function calculates the confusion matrix using the `confusion_matrix` function from the `sklearn.metrics` library.

→ It then generates a heatmap of the confusion matrix using the `seaborn` library, using the BrBG color palette.

→ The function also includes annotations on the heatmap, sets the title, and labels the x and y axes for clarity.

→ Finally, it displays the plot using `plt.show()`.

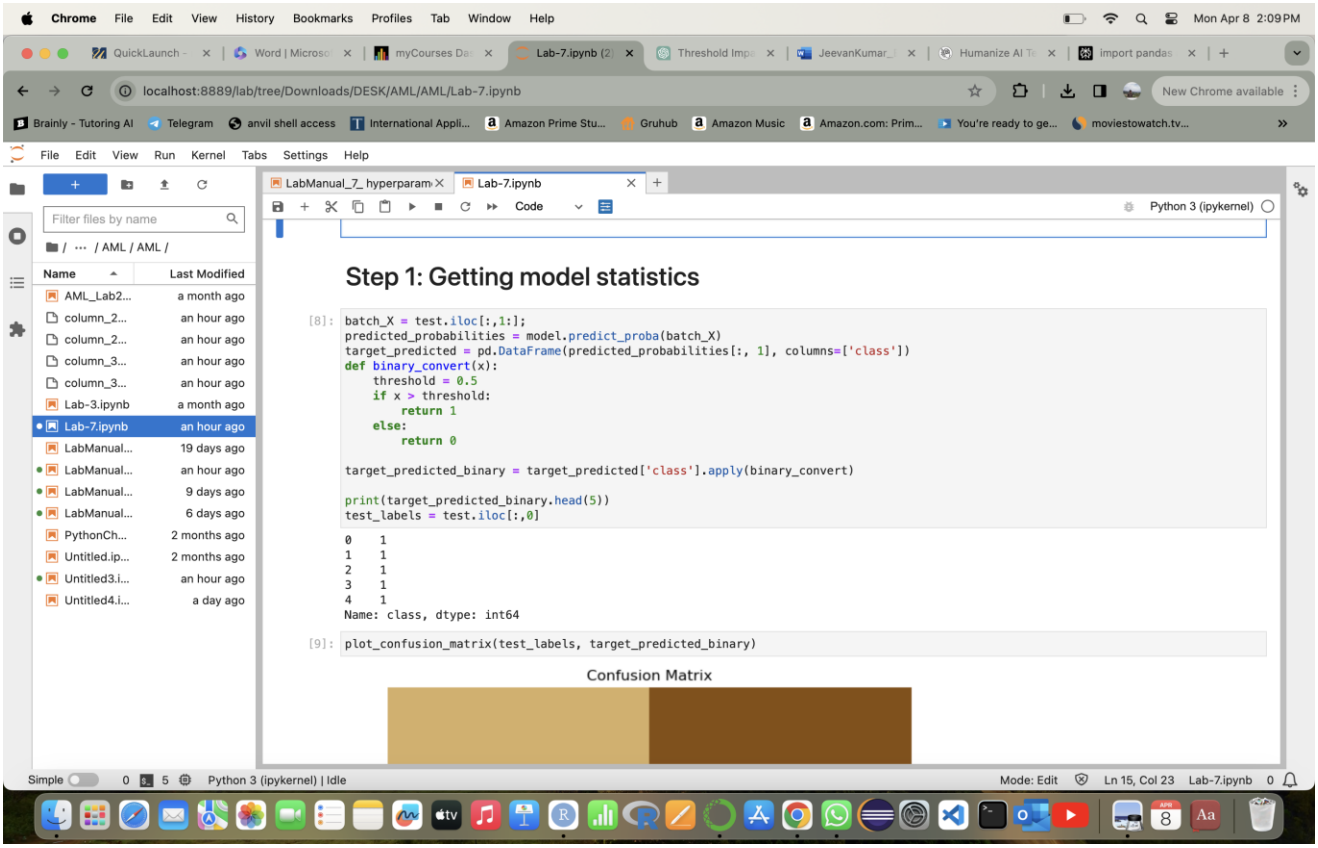
→ Plot ROC Curve: This function receives the actual labels (test_labels) and the predicted probabilities (target_predicted_binary) as parameters.

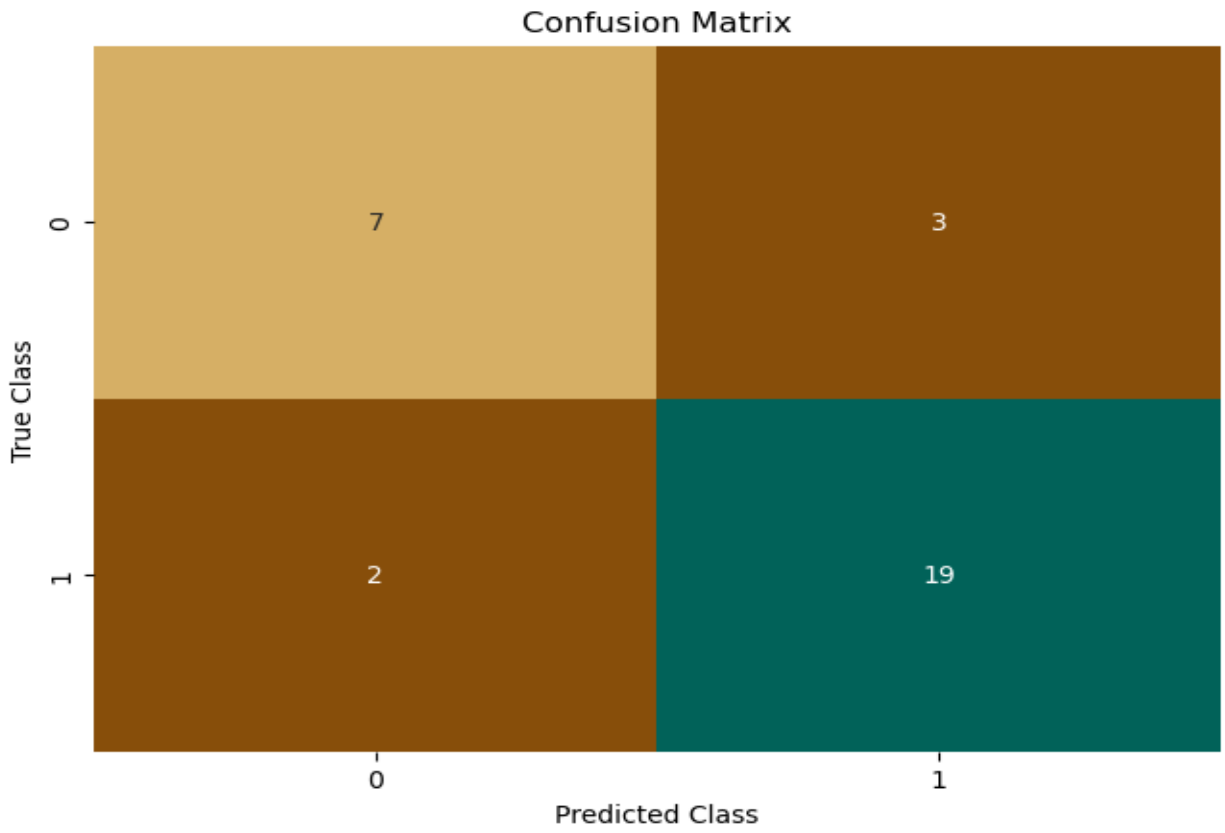
→ It computes various performance metrics like sensitivity, specificity, precision, negative predictive value, false positive rate, false negative rate, false discovery rate, and overall accuracy.

→ The results are displayed on the screen. Additionally, it calculates the area under the ROC curve (AUC) using the roc_auc_score function from sklearn.metrics.

→ The ROC curve is then plotted using the roc_curve function from sklearn.metrics along with the auc function to determine the area under the curve.

→ The AUC value is included in the plot's legend.





→ To prepare the test data, we extract the feature columns from the test DataFrame excluding the target/class column and assign it to batch_X.

→ Next, we use the trained model to generate predicted probabilities for the test data.

→ The predict_proba method returns a 2D array with the second column containing the probabilities for the positive class.

→ Then, we convert these probabilities into binary predictions by creating a new DataFrame called target_predicted with the probabilities for the positive class.

→ def binary_convert(x): This defines a helper function that converts the probabilities to binary predictions based on a threshold of 0.5.

→ target_predicted_binary = target_predicted['class'].apply(binary_convert): This line applies the binary_convert function to the 'class' column of target_predicted, creating a new Series target_predicted_binary with the binary predictions.

→ Printing the First 5 Predictions:

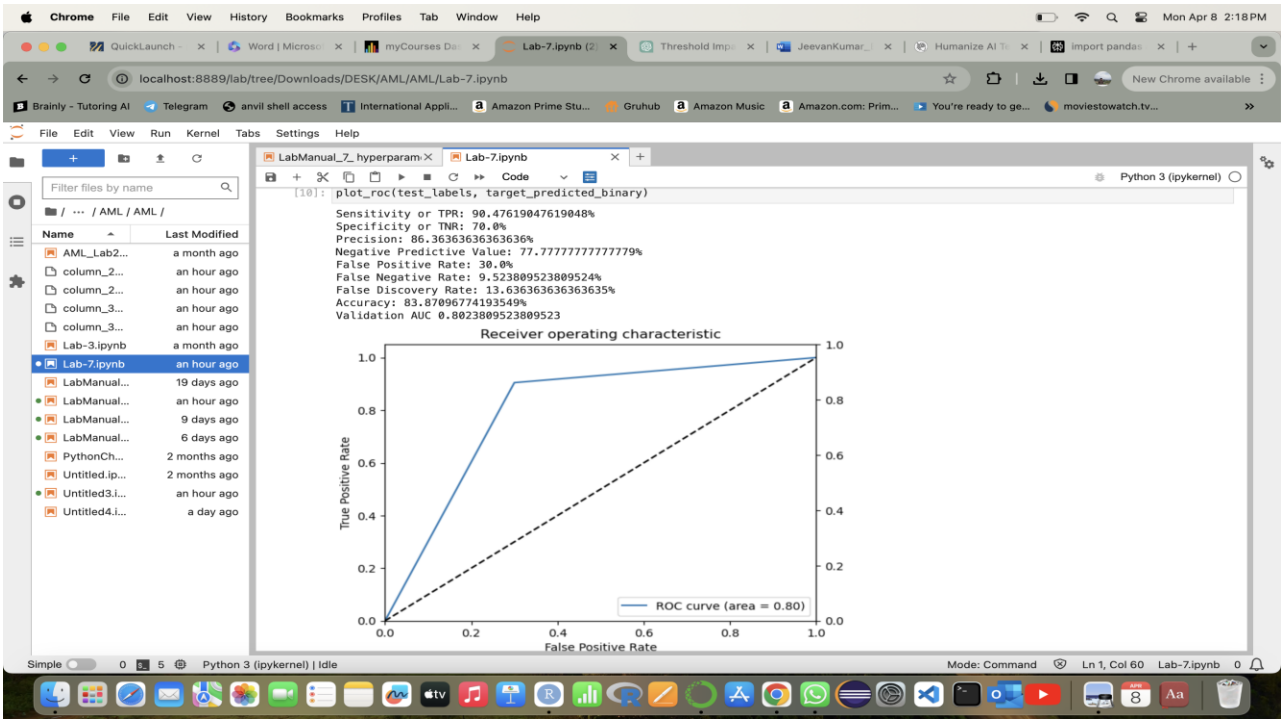
→ print(target_predicted_binary.head(5)): This line prints the first 5 rows of the target_predicted_binary Series, which shows that all 5 predictions are 1 (the positive class).

→ test_labels = test.iloc[:,0]: This line extracts the true labels (the first column of the test DataFrame) and assigns them to the test_labels variable.

→ **Plotting the Confusion Matrix:**

- plot_confusion_matrix(test_labels, target_predicted_binary): This line calls the plot_confusion_matrix function defined earlier, passing the true labels (test_labels) and the binary predictions (target_predicted_binary) as arguments. This will display the confusion matrix for the model's performance on the test data.

→ The output shows that the first 5 predictions made by the model are all 1 (the positive class). This indicates that the model is predicting the positive class for all these test instances.



The plot_roc function performs the following tasks:

Calculating Performance Metrics: It calculates various performance metrics from the confusion matrix, including:

Sensitivity, Specificity, Precision, Negative Predictive Value (NPV), False Positive Rate (FPR), False Negative Rate (FNR), False Discovery Rate (FDR), Overall Accuracy.

→ It calculates the AUC (Area Under the Receiver Operating Characteristic Curve) using the roc_auc_score function from sklearn.metrics.

→ The AUC value is printed on the console.

→ Sensitivity (TPR): 90.48% - This means that the model correctly identifies 90.48% of the positive instances.

→ Specificity (TNR): 70.0% - This means that the model correctly identifies 70.0% of the negative instances.

→ Precision: 86.36% - This means that 86.36% of the instances predicted as positive are actually positive.

→ Negative Predictive Value (NPV): 77.78% - This means that 77.78% of the instances predicted as negative are actually negative.

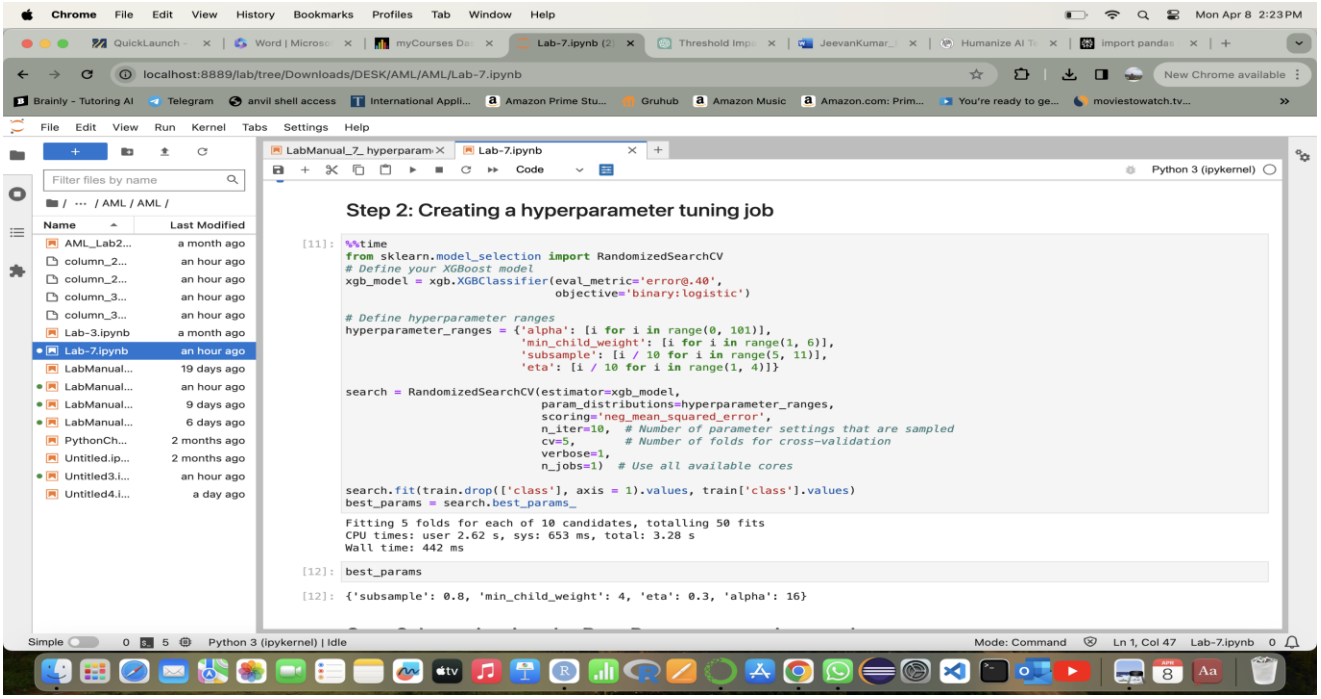
→ False Positive Rate (FPR): 30.0% - This means that 30.0% of the negative instances are incorrectly predicted as positive.

→ False Negative Rate (FNR): 9.52% - This means that 9.52% of the positive instances are incorrectly predicted as negative.

→ False Discovery Rate (FDR): 13.64% - This means that 13.64% of the positive predictions are actually negative.

→ Accuracy: 83.87% - This means that the model correctly classifies 83.87% of the instances.

→ Validation AUC: 0.8024 - This means that the model has an AUC of 0.8024 on the validation set, indicating a reasonably good performance.



Defining the XGBoost Model:

→ The code creates an instance of the XGBoostClassifier with the following parameters:

→ eval_metric='error@.40': This sets the evaluation metric to the error rate at a 40% threshold.

→ objective='binary:logistic': This sets the objective function to binary logistic regression.

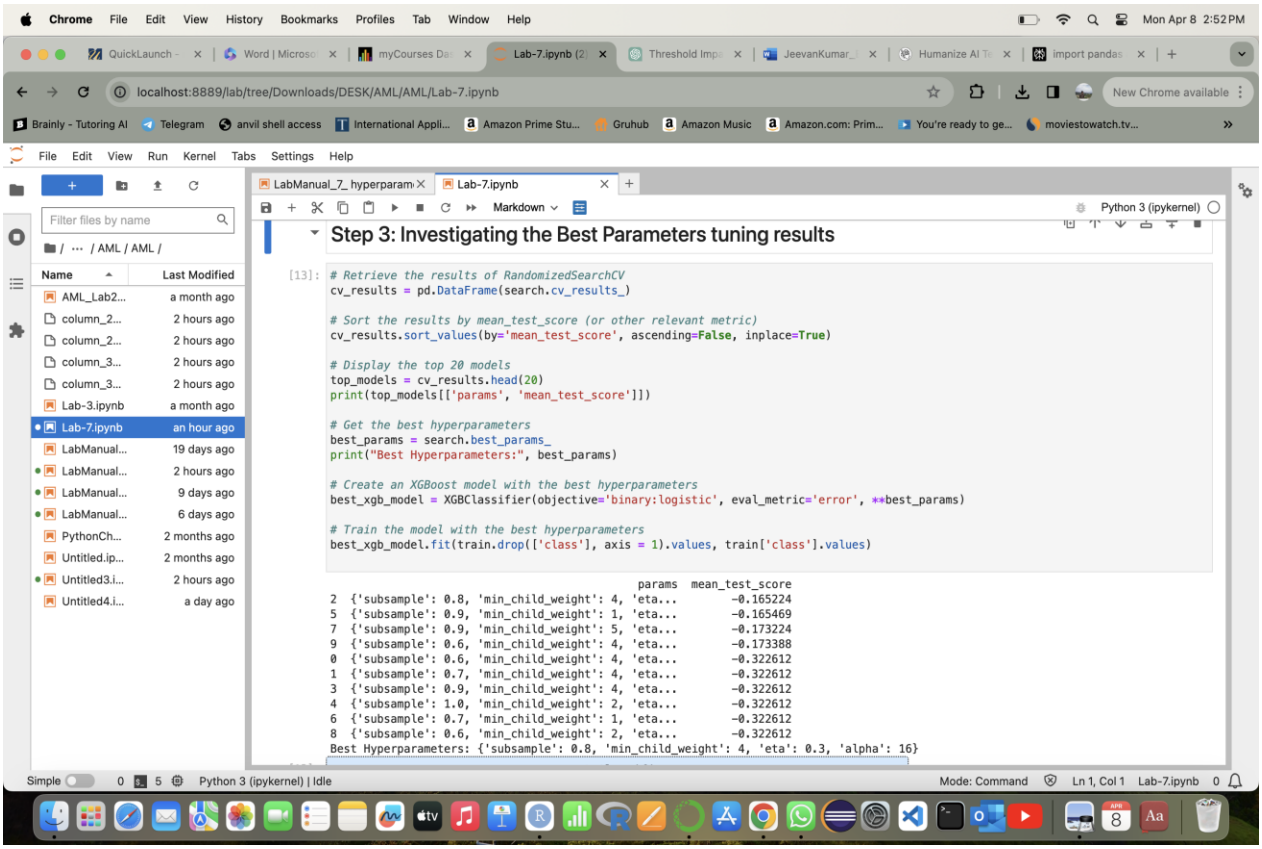
→ The code sets the ranges of hyperparameters to be tuned in a dictionary called hyperparameter_ranges.

→ These include the L1 regularization term 'alpha' ranging from 0 to 100, the minimum sum of instance weight 'min_child_weight' required in a child ranging from 1 to 5, the subsample ratio 'subsample' of the training instance ranging from 0.5 to 1.0 in increments of 0.1, and the learning rate 'eta' ranging from 0.1 to 0.3 in increments of 0.1.

- ♦ To conduct Randomized Search Cross-Validation, first create an instance of the RandomizedSearchCV class.
- ♦ This class will search randomly over specified hyperparameter ranges. Set the estimator parameter as the XGBoost model instance and the param_distributions parameter as the hyperparameter_ranges dictionary.
- ♦ The scoring parameter should be set to 'neg_mean_squared_error' to minimize the negative mean squared error. Set n_iter to 10 for sampling 10 different hyperparameter combinations.
- ♦ Use cv=5 for 5-fold cross-validation. To display progress, set verbose to 1.

The total CPU time taken for the search is 3.28 seconds, and the wall time (actual elapsed time) is 442 milliseconds.

This hyperparameter optimization process is useful for finding the best combination of hyperparameters for the XGBoost model, which can improve its performance on the given dataset.



→ Retrieving the Results of RandomizedSearchCV: To retrieve the results of RandomizedSearchCV, the code creates a Pandas DataFrame called cv_results by extracting the cv_results_ attribute from the search object.

→ This DataFrame contains all the results obtained from RandomizedSearchCV.

→ Sorting the Results: Once the DataFrame is created, it is sorted based on the mean_test_score column in descending order.

→ This allows us to see the best-performing models at the top of the list. Displaying the Top 20 Models: After sorting the DataFrame, the code selects the top 20 models and displays their parameters and mean_test_score values.

→ Retrieving the Best Hyperparameters: Finally, the code retrieves the best hyperparameters found by RandomizedSearchCV by accessing the best_params_ attribute from the search object, and then prints them out for review.

→ Creating a New XGBoost Model with the Best Hyperparameters:

→ The code creates a new instance of the XGBClassifier with the objective set to 'binary:logistic', the evaluation metric set to 'error', and the other hyperparameters set to the best values found by the RandomizedSearchCV.

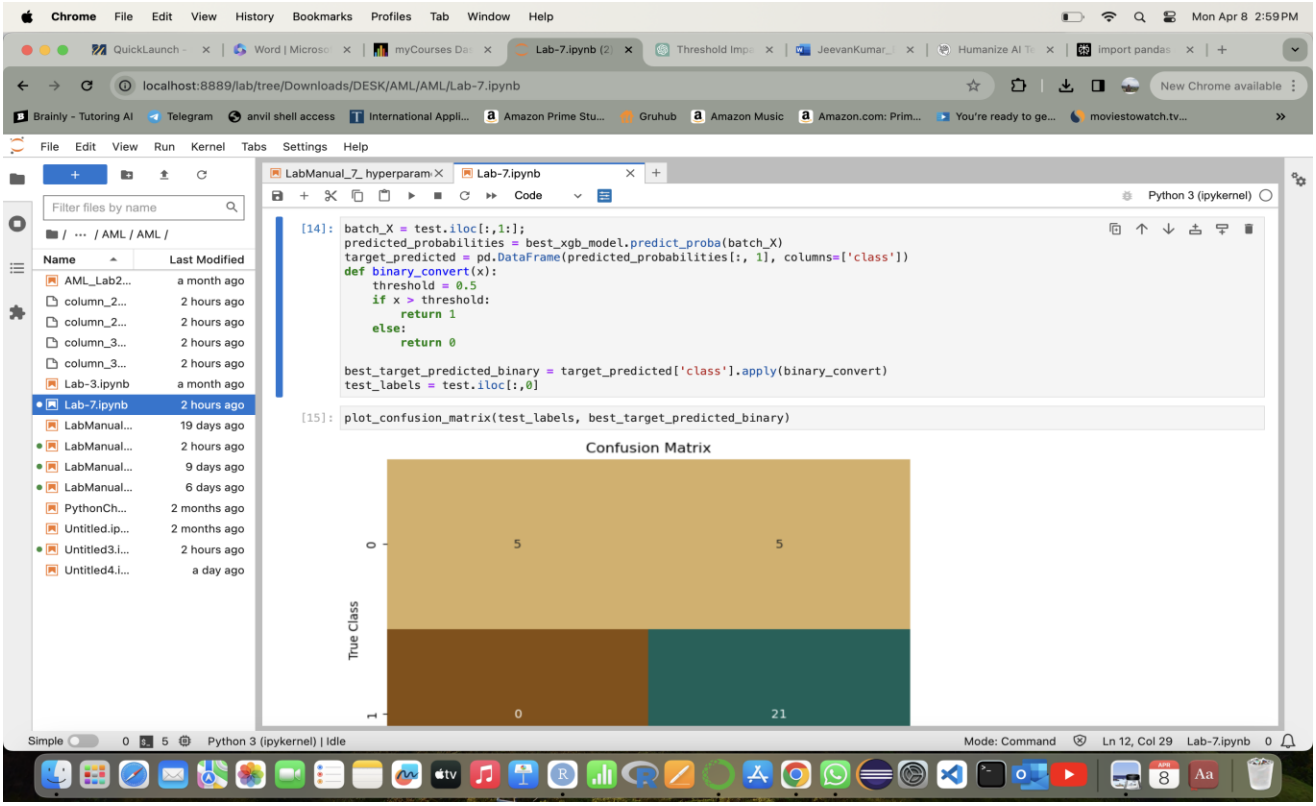
→ Training the Model with the Best Hyperparameters:

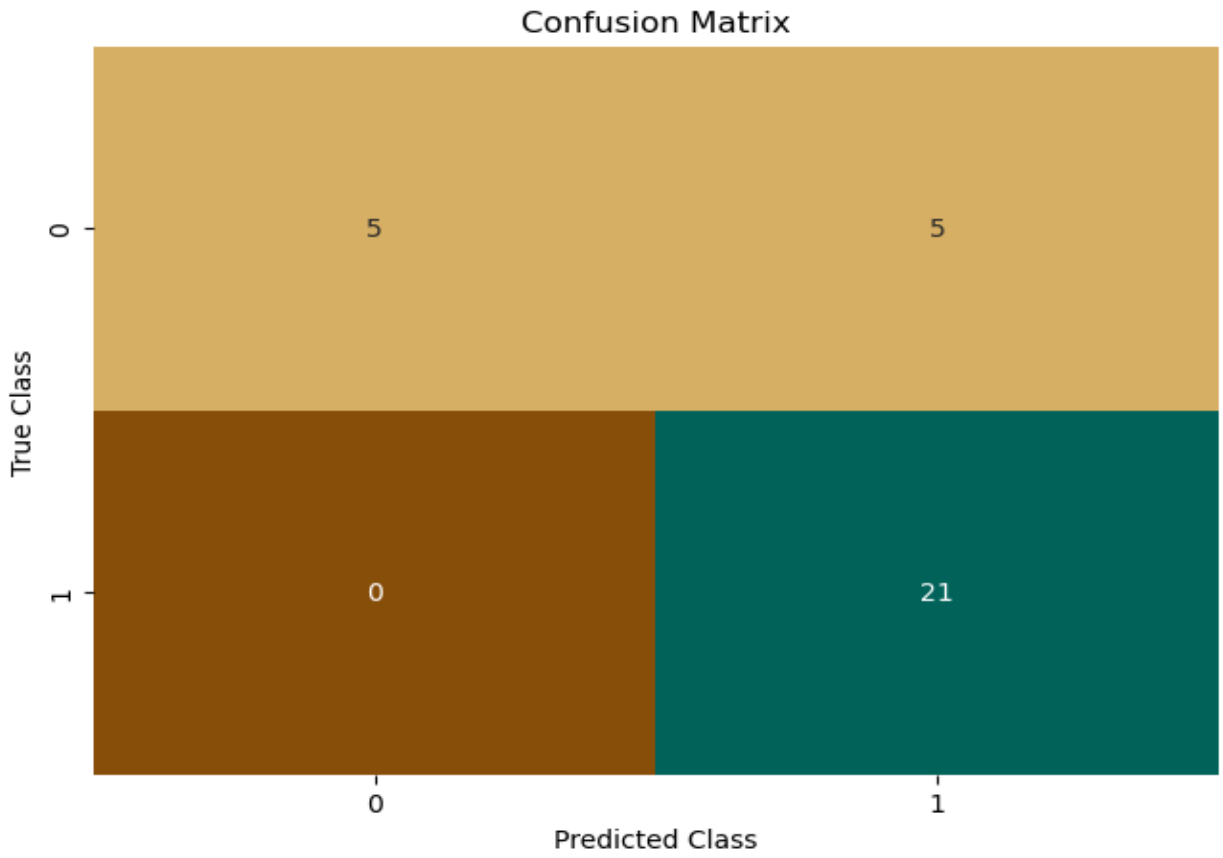
→ The code trains the new XGBoost model with the best hyperparameters using the training data (train.drop(['class'], axis = 1).values and train['class'].values).

The results display the top 20 models identified by RandomizedSearchCV, sorted by mean_test_score in descending order. The params column exhibits hyperparameter values for each model, while the mean_test_score column presents the corresponding mean test score (in this scenario, negative mean squared error).

The optimal hyperparameters discovered by RandomizedSearchCV are: - 'subsample': 0.8 - 'min_child_weight': 4 - 'eta': 0.3 - 'alpha': 16 The final XGBoost model is constructed using the best hyperparameters, utilizing the XGBClassifier class with specified values.

This example illustrates the process of enhancing hyperparameters for an XGBoost model with RandomizedSearchCV and constructing a new model with these parameters.


Chrome File Edit View History Bookmarks Profiles Tab Window Help
Mon Apr 8 2:59 PM
localhost:8889/lab/tree/Downloads/DESK/AML/AML/Lab-7.ipynb
File Edit View Run Kernel Tabs Settings Help
LabManual_7_ hyperparam
Lab-7.ipynb
Code
Python 3 (ipykernel)
Filter files by name
/ AML / AML /
Name
Last Modified
AML_Lab2... a month ago
column_2... 2 hours ago
column_2... 2 hours ago
column_3... 2 hours ago
column_3... 2 hours ago
Lab-3.ipynb a month ago
Lab-7.ipynb 2 hours ago
LabManual... 19 days ago
LabManual... 2 hours ago
LabManual... 9 days ago
LabManual... 6 days ago
PythonCh... 2 months ago
Untitled.ip... 2 months ago
Untitled3.i... 2 hours ago
Untitled4.i... a day ago
[14]:
batch_X = test.iloc[:,1:];
predicted_probabilities = best_xgb_model.predict_proba(batch_X)
target_predicted = pd.DataFrame(predicted_probabilities[:, 1], columns=['class'])
def binary_convert(x):
threshold = 0.5
if x > threshold:
return 1
else:
return 0
best_target_predicted_binary = target_predicted['class'].apply(binary_convert)
test_labels = test.iloc[:,0]
[15]:
plot_confusion_matrix(test_labels, best_target_predicted_binary)
Confusion Matrix
True Class
0
1
5
5
0
21
Simple
Python 3 (ipykernel) | Idle
Mode: Command
Ln 12, Col 29
Lab-7.ipynb


Confusion Matrix
True Class
0
1
5
5
0
21
0
1
Predicted Class

To prepare the test data, we extract the feature columns from the test DataFrame and assign them to batch_X variable. Next, we use the trained XGBoost model with the best hyperparameters to generate predicted probabilities for the test data.

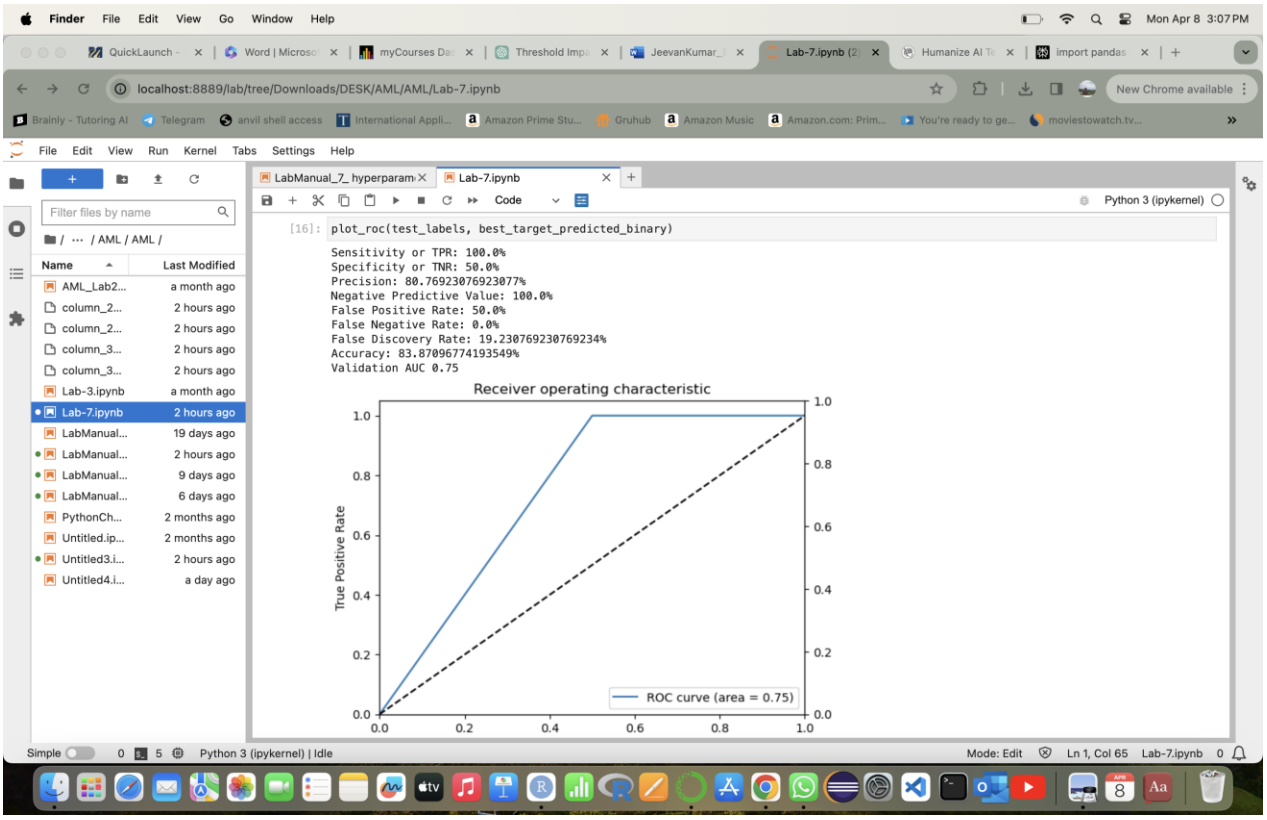
The predict_proba method returns a 2D array with the probabilities for the positive class. Finally, we convert the probabilities to binary predictions.

Converting Probabilities to Binary Predictions: Here we have code that creates a new DataFrame called target_predicted with the probabilities for the positive class: python target_predicted = pd.DataFrame(predicted_probabilities[:, 1], columns=['class'])

Next, we define a helper function binary_convert that converts the probabilities to binary predictions based on a threshold of 0.5: python def binary_convert(x):

Then, we apply the binary_convert function to the 'class' column of target_predicted to create a new Series called best_target_predicted_binary: python best_target_predicted_binary = target_predicted['class'].apply(binary_convert) Extracting the True Labels: To extract the true labels (the first column) from the test DataFrame, we use the following code: python test_labels = test.iloc[:,0]

plot_confusion_matrix(test_labels, best_target_predicted_binary): This line calls the plot_confusion_matrix function defined earlier, passing the true labels (test_labels) and the binary predictions (best_target_predicted_binary) as arguments. This will display the confusion matrix for the model's performance on the test data.



The plot_roc function is called with the true labels (test_labels) and the binary predictions (best_target_predicted_binary) from the best XGBoost model.

The plot_roc function reveals the following insights:

Sensitivity (TPR): 100.0% - This indicates that the model correctly identifies all positive instances.

Specificity (TNR): 50.0% - This shows that the model correctly identifies 50.0% of the negative instances.

Precision: 80.77% - This means that 80.77% of the instances predicted as positive are actually positive.

Negative Predictive Value (NPV): 100.0% - This implies that all instances predicted as negative are indeed negative.

False Positive Rate (FPR): 50.0% - This signifies that 50.0% of the negative instances are wrongly predicted as positive.

False Negative Rate (FNR): 0.0% - This indicates that the model does not make any false negative predictions.

False Discovery Rate (FDR): 19.23% - This means that 19.23% of the positive predictions are actually negative.

Accuracy: 83.87% - This means that the model correctly classifies 83.87% of the instances.

Validation AUC: 0.75 - This means that the model has an AUC of 0.75 on the validation set, indicating a moderately good performance.

Question:

How do these results differ from the original? Are these results better or worse?

Ans:

The outcomes of adjusting hyperparameters using RandomizedSearchCV don't seem to show a significant improvement compared to the starting model with default settings.

There could be a few explanations for this. One possibility is that the initial model was already performing quite well with its default settings, leaving little room for enhancement through hyperparameter adjustments.

Additionally, the dataset being used has 6,552 rows and 34 features, which is not considered a large dataset. Smaller datasets like this may have limited potential for improvement through hyperparameter tuning.

Hyperparameter Ranges: The hyperparameters used in the RandomizedSearchCV may not cover a wide enough range to find the best values. Enlarging the ranges, as recommended in the search results, could potentially enhance performance.

Overfitting: The model might fit too closely to the training data, and the hyperparameter optimization may not effectively deal with this problem. Techniques such as early stopping, or regularization may be necessary to improve generalization.

In conclusion, the results of the hyperparameter optimization do not seem to be significantly better than the original model, likely due to the model already performing well on the dataset, the size of the dataset, the limited approach to hyperparameter tuning, and potential issues.

Conclusion:

The following code showcases a thorough method for constructing and enhancing an XGBoost model for binary classification.

Initially, the process involves obtaining and preparing the "Vertebral Column Data" dataset from the UCI Machine Learning Repository, before dividing it into training, testing, and validation sets.

The code's first segment trains an XGBoost model with standard hyperparameters and assesses its effectiveness through different metrics like confusion matrix and ROC curve.

The outcomes reveal that the preliminary model performs well, achieving an accuracy of approximately 84% with an AUC of 0.80 on the validation set.

To enhance the model's performance, the code utilizes a hyperparameter optimization technique called RandomizedSearchCV.

This method allows for a wider exploration of hyperparameter combinations to find the best configuration for the dataset and problem at hand.

The tuning process involves exploring alpha (L1 regularization), min_child_weight, subsample, and eta (learning rate) parameters. The RandomizedSearchCV randomly samples 10 different combinations of these parameters and conducts a 5-fold cross-validation.

The RandomizedSearchCV results indicate that the optimal hyperparameters are:

subsample: 0.8

min_child_weight: 4

eta: 0.3

alpha: 16

A new XGBoost model is then created using these best hyperparameters, and its performance is evaluated on the test set.

Interestingly, the results from the hyperparameter-tuned model do not appear to be significantly better than the initial model with default parameters. The accuracy remains around 84%, and the AUC on the validation set is 0.75, which is lower than the initial model's AUC of 0.80.

There could be a few reasons for this result:

1. The initial model might have already been doing well on the dataset, so tweaking hyperparameters didn't make much of a difference.

2. The dataset size (6,552 rows and 34 features) may not be big enough for hyperparameter tuning to have a significant impact.

3. The RandomizedSearchCV method used may not have thoroughly explored the hyperparameter space to find the best configuration. A more thorough search method like grid search or Bayesian optimization might be needed.

4. The ranges of hyperparameters in RandomizedSearchCV may not have been broad enough to find the best values, as indicated by the search results.

5. The model may be prone to overfitting, and the hyperparameter tuning may not have effectively addressed this issue. Techniques like early stopping or regularization may be needed to improve generalization.

Although there was no noticeable improvement in the model's performance, the code showcases a well-organized method for constructing and refining an XGBoost classifier.

Utilizing tools like the confusion matrix and ROC curve helps reveal the model's strengths and weaknesses.

In the future, to boost the model's performance, you can take the following steps: expanding hyperparameter search ranges and trying a more thorough optimization method like grid search or Bayesian optimization, looking into potential overfitting issues, and using suitable regularization techniques.

Experimenting with different boosting algorithms, such as LightGBM, as suggested in the search results.

Exploring feature engineering and selection techniques to identify the most informative variables for the classification task.

Considering the use of ensemble methods, such as stacking or blending, to combine the strengths of multiple models.

By addressing these potential areas for improvement, the model's performance on the "Vertebral Column Data" dataset can be further optimized, leading to a more robust and accurate classification system.