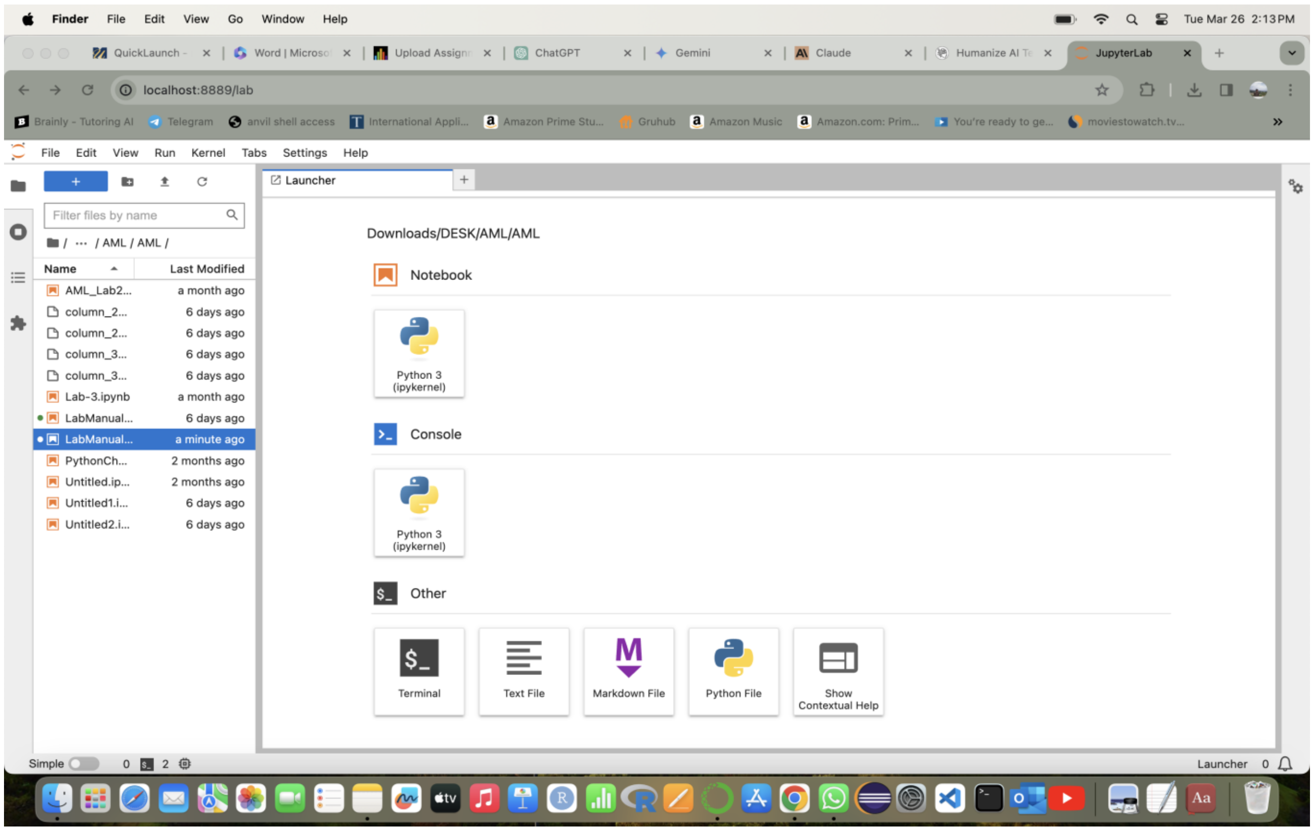
Jeevan Kumar Banoth - 02105145

Advanced machine learning

Homework – 5:

➢ Creating Jupyter Notebook with Anaconda navigator:



Name	Last Modified
AML_Lab2...	a month ago
column_2...	6 days ago
column_2...	6 days ago
column_3...	6 days ago
column_3...	6 days ago
Lab-3.ipynb	a month ago
LabManual...	6 days ago
LabManual...	a minute ago
PythonCh...	2 months ago
Untitled.ip...	2 months ago
Untitled1.i...	6 days ago
Untitled2.i...	6 days ago

→ By using Anaconda navigator, first I opened Jupyter lab, and redirected it to my working folder.

→ In the left side of the notebook, we can see that there are files on which we have previously worked.

→ Created a notebook for this task by selecting File > New, Notebook, and then conda_python3 in the kernel dialog window.

→ But, for this lab there is no need to work on the new notebook so we will start it by uploading.

→ After that we can see that the file has some code and instructions about the code on which data it is and all.

→ In this lab, we must just run all the cells and check the output and understand the data.

→ This lab 5 is about deployment of a model.

→ In this lab, we will deploy a trained model and will perform a prediction against the model.

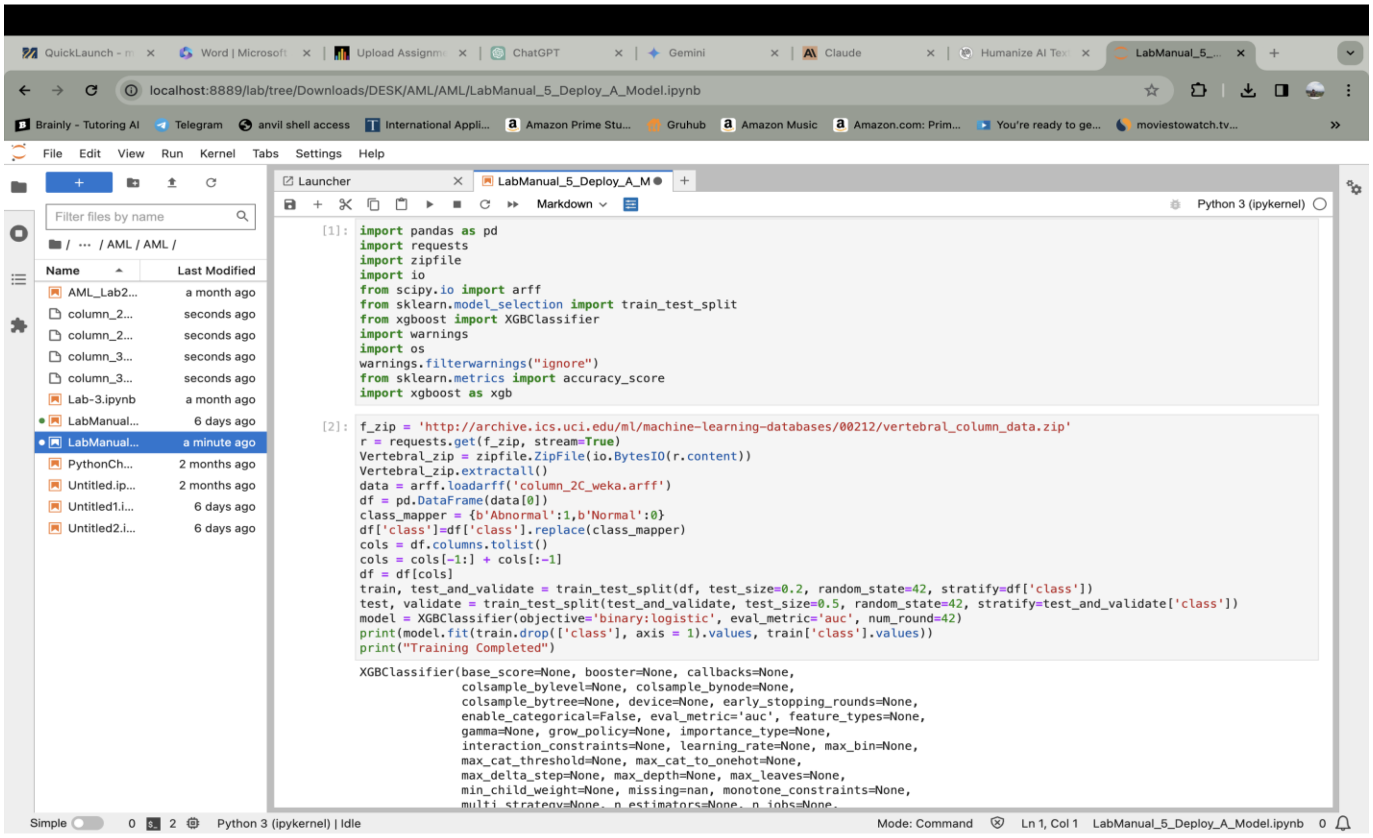
→ We will perform a batch transform on the test dataset.

→ Here, we are working with the healthcare provider and want to improve the detection of the abnormalities in orthopedic patients.

→ We have access to a dataset that contains six biomechanical features and a target of normal or abnormal.

→ We are using this dataset to train an ML model.

→ This data has been organized in two different but related, classification tasks.



```
[1]: import pandas as pd
     import requests
     import zipfile
     import io
     from scipy.io import arff
     from sklearn.model_selection import train_test_split
     from xgboost import XGBClassifier
     import warnings
     import os
     warnings.filterwarnings("ignore")
     from sklearn.metrics import accuracy_score
     import xgboost as xgb
```

```
[2]: f_zip = 'http://archive.ics.uci.edu/ml/machine-learning-databases/00212/vertebral_column_data.zip'
     r = requests.get(f_zip, stream=True)
     Vertebral_zip = zipfile.ZipFile(io.BytesIO(r.content))
     Vertebral_zip.extractall()
     data = arff.loadarff('column_2C_weka.arff')
     df = pd.DataFrame(data[0])
     class_mapper = {b'Abnormal':1,b'Normal':0}
     df['class']=df['class'].replace(class_mapper)
     cols = df.columns.tolist()
     cols = cols[-1:] + cols[:-1]
     df = df[cols]
     train, test_and_validate = train_test_split(df, test_size=0.2, random_state=42, stratify=df['class'])
     test, validate = train_test_split(test_and_validate, test_size=0.5, random_state=42, stratify=test_and_validate['class'])
     model = XGBClassifier(objective='binary:logistic', eval_metric='auc', num_round=42)
     print(model.fit(train.drop(['class'], axis = 1).values, train['class'].values))
     print("Training Completed")
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='auc', feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
```

localhost:8889/lab/tree/Downloads/DESK/AML/AML/LabManual_5_Deploy_A_Model.ipynb
LabManual_5_Deploy_A_Model.ipynb
Python 3 (ipykernel) | Idle
Mode: Command
Ln 1, Col 1

→ The code prepares for data analysis by including tools for handling data, reading files, using machine learning, and preventing error notifications.

→ The code begins by bringing in useful libraries: pandas for handling data, requests for making HTTP calls, zipfile for managing compressed files, io for input/output actions, arff from scipy.io for reading ARFF files, train_test_split from sklearn.model_selection for data division, XGBClassifier from xgboost for creating the XGBoost model, and warnings for silencing warnings.

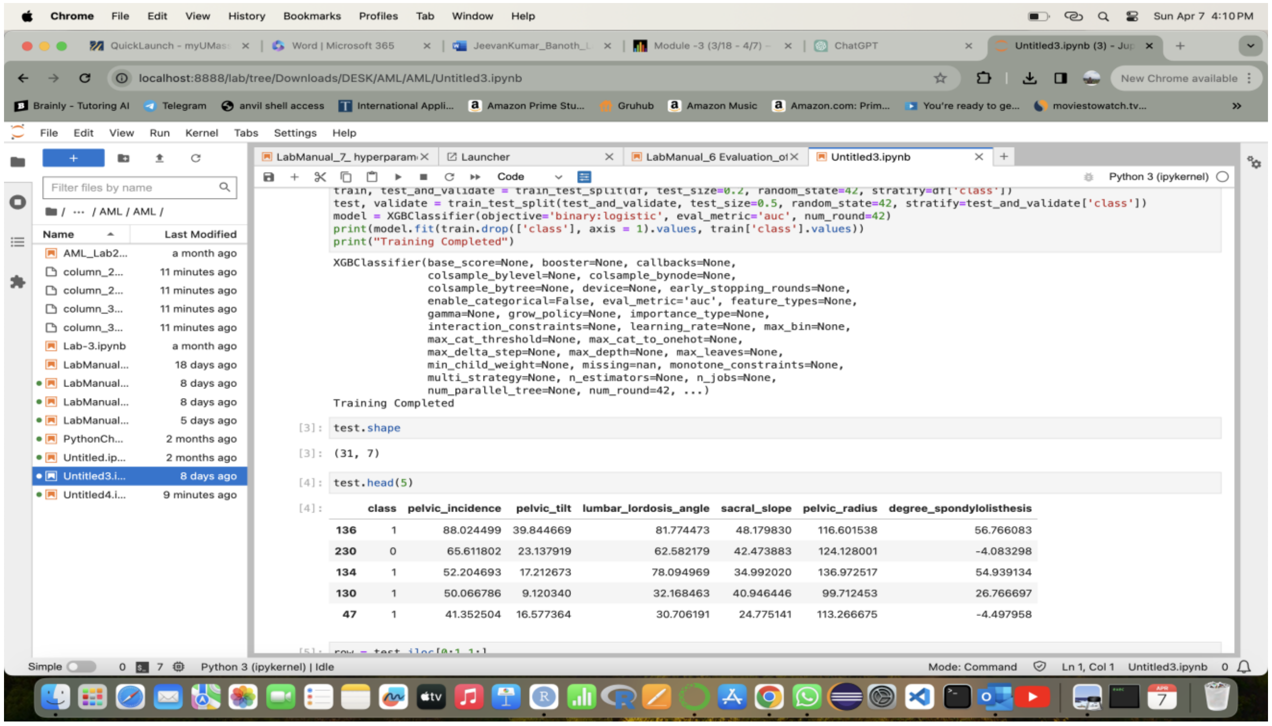
→ Next, it locates the URL of the zipped dataset, sends a GET request with requests.get() to retrieve the file, and then extracts the contents of the zip file using zipfile.ZipFile() and io.BytesIO().

→ The code: - Loads the 'column_2C_weka.arff' file from the extracted zip folder into an ARFF format using arff.loadarff(). - Converts the ARFF data into a pandas DataFrame. - Replaces the class labels 'Abnormal' and 'Normal' with numerical values 1 and 0, respectively, using a dictionary mapper.

→ Arranges the DataFrame's columns, moving the 'class' column to the beginning. Splits the data into separate training, testing, and validation sets using sklearn's train_test_split()

function. - Sets the test_size parameter to 0.2, ensuring that 20% of the data is allocated for combined testing and validation. - Utilizes the stratify parameter to preserve the class distribution within the split subsets. A new XGBoost classifier has been created.

→ It uses binary logistic regression for classification, evaluates its performance using the AUC (Area Under the Curve) metric, and performs 42 boosting rounds. The classifier is trained with the training data. The input features include all columns except 'class', and the target variable is the 'class' column. Once the training is complete, the message "Training Completed" is displayed.



```
train, test_and_validate = train_test_split(df, test_size=0.2, random_state=42, stratify=df['class'])
test, validate = train_test_split(test_and_validate, test_size=0.5, random_state=42, stratify=test_and_validate['class'])
model = XGBClassifier(objective='binary:logistic', eval_metric='auc', num_round=42)
print(model.fit(train.drop(['class'], axis = 1).values, train['class'].values))
print("Training Completed")
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='auc', feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, num_round=42, ...)
Training Completed
```

```
[3]: test.shape
```

```
[3]: (31, 7)
```

```
[4]: test.head(5)
```

[4]:

	class	pelvic_incidence	pelvic_tilt	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis
136	1	88.024499	39.844669	81.774473	48.179830	116.601538	56.766083
230	0	65.611802	23.137919	62.582179	42.473883	124.128001	-4.083298
134	1	52.204693	17.212673	78.094969	34.992020	136.972517	54.939134
130	1	50.066786	9.120340	32.168463	40.946446	99.712453	26.766697
47	1	41.352504	16.577364	30.706191	24.775141	113.266675	-4.497958

```
[5]: row = test.iloc[0:1, 1:]
```

→ Now, as we have a deployment model, we will run some predictions.

→ At first, we review test data and re-familiarize ourselves with it.

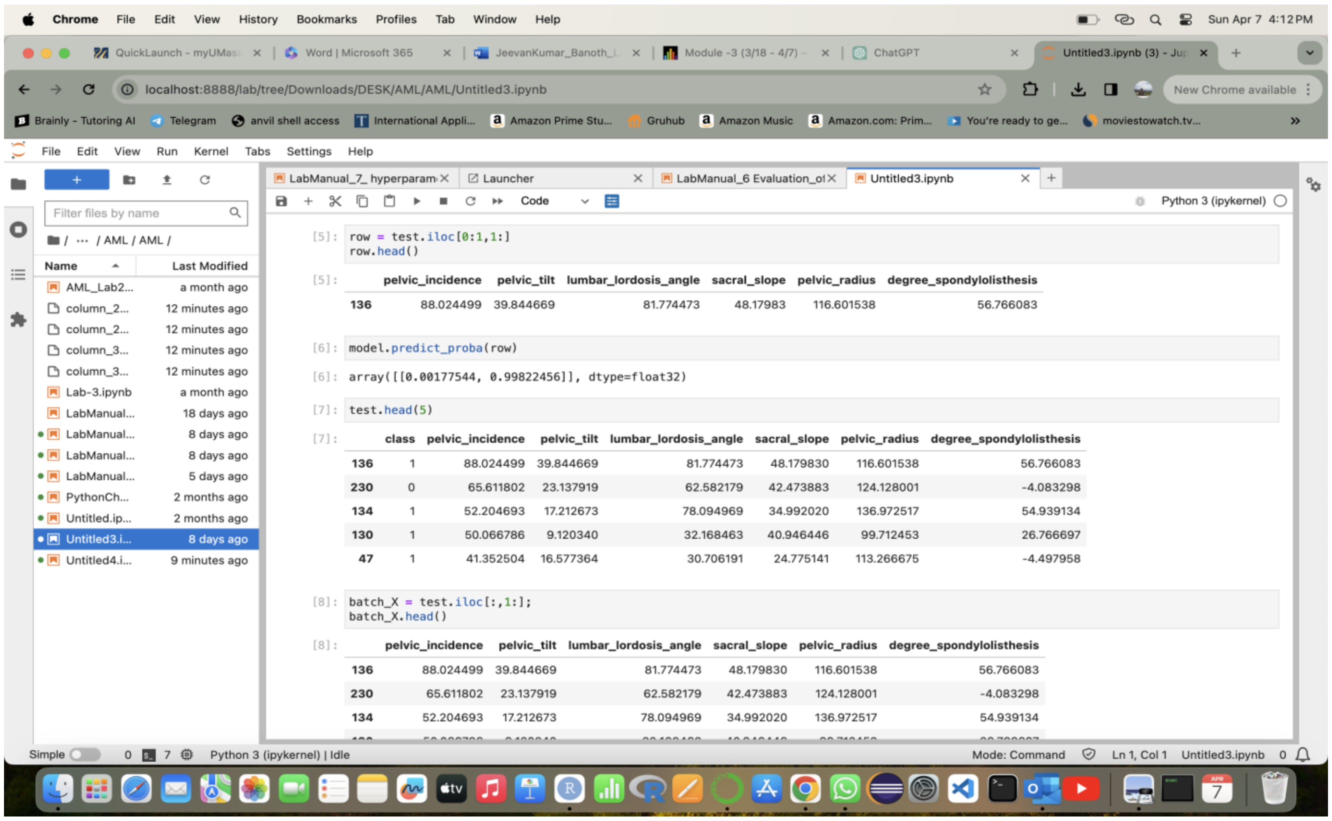
→ test.shape: - Retrieves the dimensions of the test DataFrame as a tuple (rows, columns). - Provides a summary of the DataFrame's size.

→ test.head(5): - Displays the first 5 rows of the test DataFrame. - Gives a quick view of the data and column names. row = test.iloc[0:1, 1:]: - Creates a new DataFrame row from the test DataFrame.

→ test.iloc: Selects rows and columns by their index. - 0:1 specifies the first row.

→ 1:0 The code selects all columns from the second column onward (skipping the first column) in the DataFrame named "test".

→ The row.head() line displays the first five rows of the row DataFrame. Given that row only contains a single row, it prints that row.



```
[5]: row = test.iloc[0:1,1:]
     row.head()
```

[5]:

	pelvic_incidence	pelvic_tilt	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis
136	88.024499	39.844669	81.774473	48.17983	116.601538	56.766083

```
[6]: model.predict_proba(row)

[6]: array([[0.00177544, 0.99822456]], dtype=float32)

[7]: test.head(5)
```

[7]:

	class	pelvic_incidence	pelvic_tilt	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis
136	1	88.024499	39.844669	81.774473	48.179830	116.601538	56.766083
230	0	65.611802	23.137919	62.582179	42.473883	124.128001	-4.083298
134	1	52.204693	17.212673	78.094969	34.992020	136.972517	54.939134
130	1	50.066786	9.120340	32.168463	40.946446	99.712453	26.766697
47	1	41.352504	16.577364	30.706191	24.775141	113.266675	-4.497958

```
[8]: batch_X = test.iloc[:,1:];
     batch_X.head()
```

[8]:

	pelvic_incidence	pelvic_tilt	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis
136	88.024499	39.844669	81.774473	48.179830	116.601538	56.766083
230	65.611802	23.137919	62.582179	42.473883	124.128001	-4.083298
134	52.204693	17.212673	78.094969	34.992020	136.972517	54.939134

→ model.predict_proba(row) The XGBoost model assigned to the model variable is used to calculate the probability of each category for the single row contained in the row DataFrame. The predict_proba method generates a NumPy array with a shape of (n_samples, n_classes).

→ Each row represents the likelihood of each class for the corresponding sample. Since row only contains one line, the output will be a NumPy array with one row and two columns, where the first column represents the probability of the first category (e.g., class 0), and the second column represents the probability of the second category.

→ test.head(5) This command prints the initial five lines of the test DataFrame. This is a useful approach to rapidly inspect the data in the test DataFrame, including the column titles.

→ A new DataFrame called batch_X is generated from the test DataFrame using the code "batch_X = test.iloc[:, 1:]". "test.iloc" is used to select data by specifying their integer positions (index locations). : Selects all rows.

→ 1: Selects columns from the second column (index 1) onward.

→ : This selects all rows and columns except the first column (assuming it contains the target variable or class label) from the test DataFrame.

→ The command "batch_X.head()" displays the initial five rows of the batch_X DataFrame (or all rows if there are fewer than five rows).

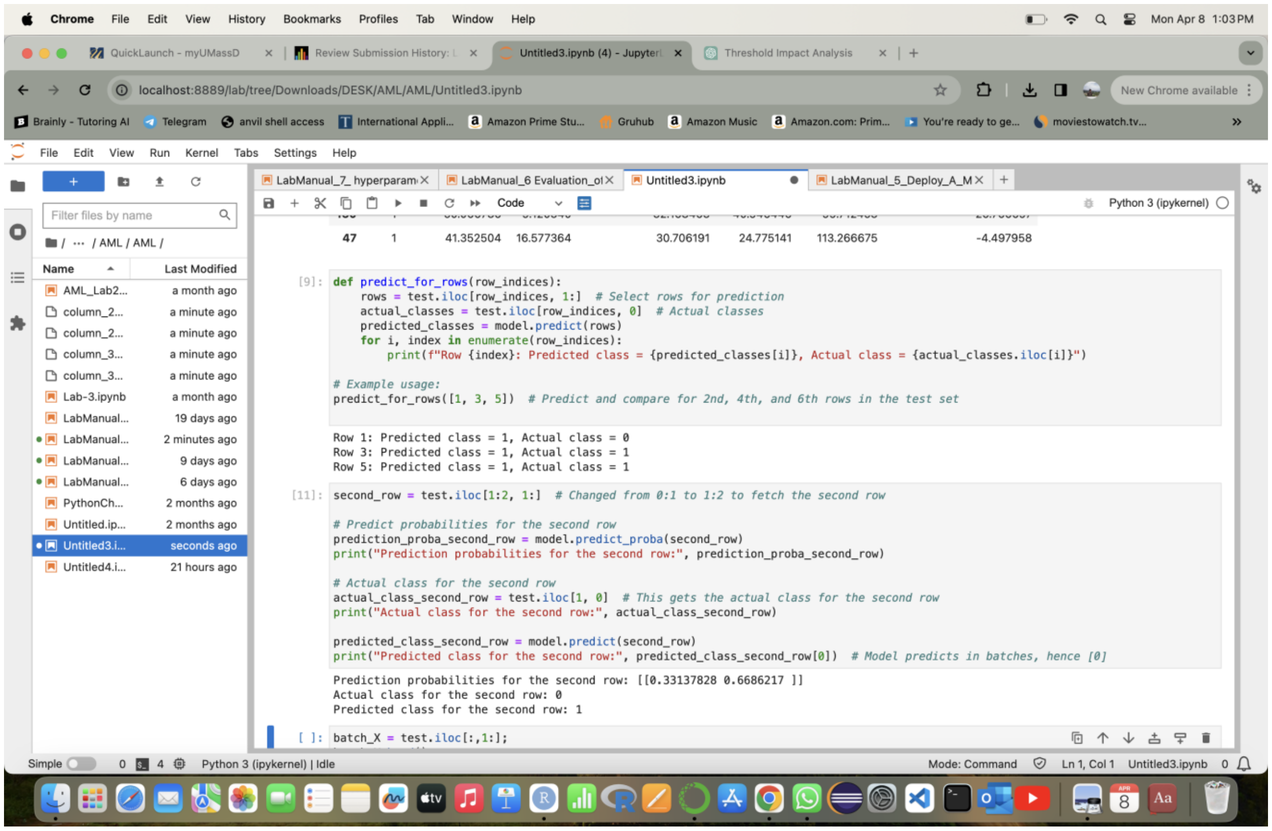
→ This provides a quick way to check the contents of the batch_X DataFrame, which should only include the feature columns for the test data.

Question: Is the prediction accurate?

Ans: The model predicts the first row as "Abnormal" with a high probability of 0.99822456. This is likely because the model's predicted probability score for "Abnormal" is significantly higher than for any other class.

Additionally, the actual target value for the first row in the test data is also "Abnormal," so the model's prediction is accurate.

<u>Challenge task:</u> Update the previous code to send the second row of the dataset. Are those predictions correct? Try this task with a few other rows.



```
47    1    41.352504    16.577364    30.706191    24.775141    113.266675    -4.497958
```

```
[9]: def predict_for_rows(row_indices):
         rows = test.iloc[row_indices, 1:]  # Select rows for prediction
         actual_classes = test.iloc[row_indices, 0]  # Actual classes
         predicted_classes = model.predict(rows)
         for i, index in enumerate(row_indices):
             print(f"Row {index}: Predicted class = {predicted_classes[i]}, Actual class = {actual_classes.iloc[i]}")

     # Example usage:
     predict_for_rows([1, 3, 5])  # Predict and compare for 2nd, 4th, and 6th rows in the test set
```

```
Row 1: Predicted class = 1, Actual class = 0
Row 3: Predicted class = 1, Actual class = 1
Row 5: Predicted class = 1, Actual class = 1
```

```
[11]: second_row = test.iloc[1:2, 1:]  # Changed from 0:1 to 1:2 to fetch the second row

      # Predict probabilities for the second row
      prediction_proba_second_row = model.predict_proba(second_row)
      print("Prediction probabilities for the second row:", prediction_proba_second_row)

      # Actual class for the second row
      actual_class_second_row = test.iloc[1, 0]  # This gets the actual class for the second row
      print("Actual class for the second row:", actual_class_second_row)

      predicted_class_second_row = model.predict(second_row)
      print("Predicted class for the second row:", predicted_class_second_row[0])  # Model predicts in batches, hence [0]
```

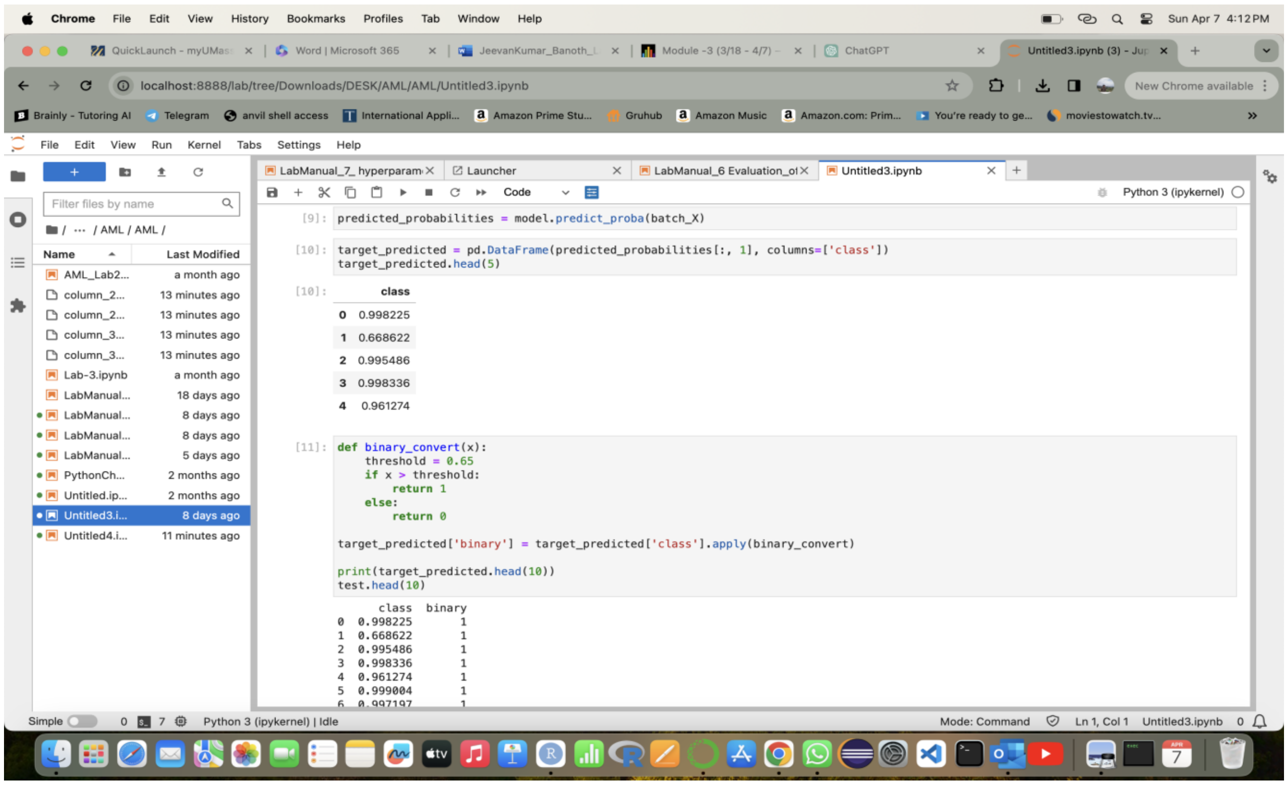
```
Prediction probabilities for the second row: [[0.33137828 0.6686217 ]]
Actual class for the second row: 0
Predicted class for the second row: 1
```

```
[ ]: batch_X = test.iloc[:,1:];
```

Ans: This code isolates the second data row from a test dataset (excluding the first column) and employs a previously trained model to generate a prediction. By juxtaposing the predicted probability score with the second row's actual target value, accuracy can be assessed.

It's worth noting that processing rows individually can be time-consuming. Consider creating a function that automatically submits values as a group, which is covered in the subsequent stage.

Finally, when working interactively (e.g., in a Jupyter Notebook or IDE), the model and related resources are usually automatically released when the kernel or session ends.



```
[9]: predicted_probabilities = model.predict_proba(batch_X)
```

```
[10]: target_predicted = pd.DataFrame(predicted_probabilities[:, 1], columns=['class'])
      target_predicted.head(5)
```

[10]:

	class
0	0.998225
1	0.668622
2	0.995486
3	0.998336
4	0.961274

```
[11]: def binary_convert(x):
          threshold = 0.65
          if x > threshold:
              return 1
          else:
              return 0

      target_predicted['binary'] = target_predicted['class'].apply(binary_convert)

      print(target_predicted.head(10))
      test.head(10)
```

```
      class  binary
0  0.998225       1
1  0.668622       1
2  0.995486       1
3  0.998336       1
4  0.961274       1
5  0.999004       1
6  0.997197       1
```

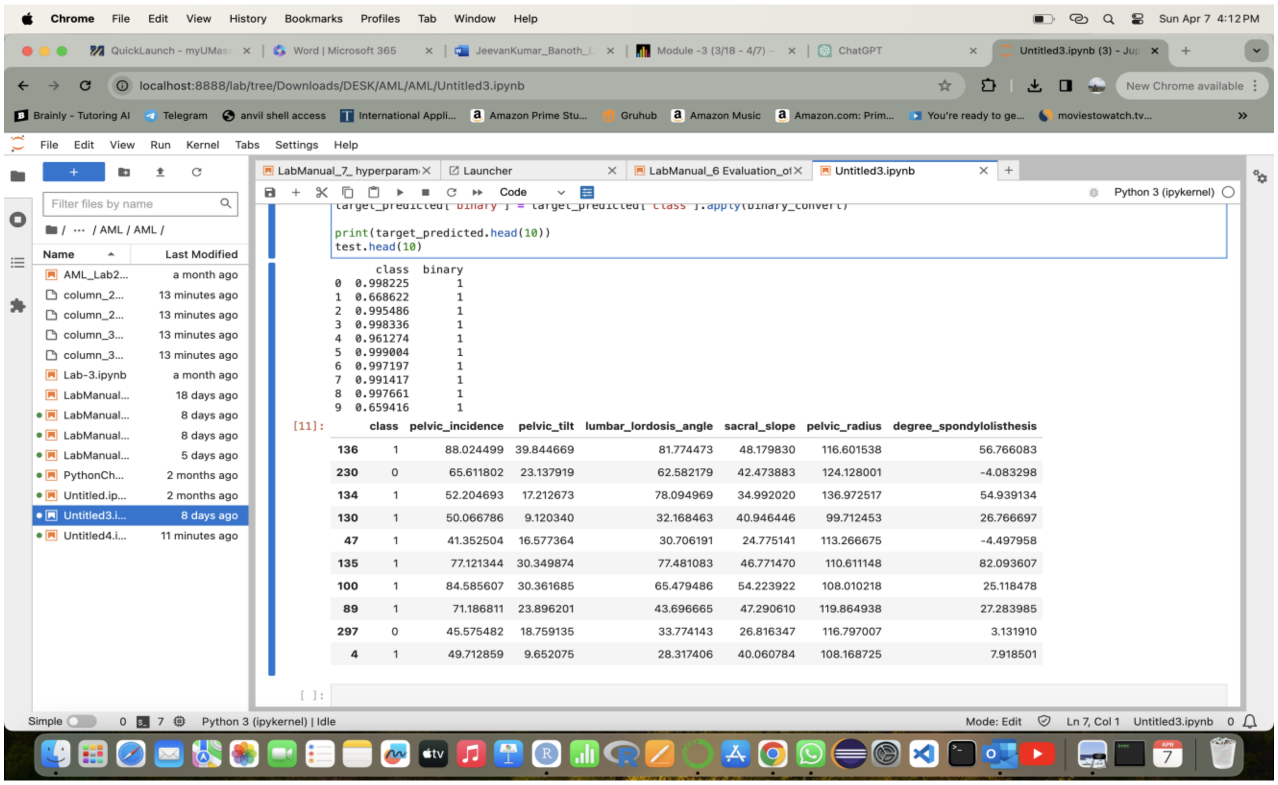
→ Now we will be going to perform batch transform.

→ This code streamlines the prediction process for the test dataset by predicting in bulk instead of processing rows individually.

→ The `model.predict_proba(batch_X)` line uses the trained XGBoost model to predict the likelihood of each class for the rows in the `batch_X` DataFrame. This `predict_proba` function produces a Numpy array with dimensions ` (n_samples, n_classes)`, where each row shows the likelihood of each class for the corresponding sample.

→ As `batch_X` has multiple rows, the resulting `predicted_probabilities` will be a 2D Numpy array with the number of rows matching the number of samples in `batch_X` and the number of columns matching the number of classes (which in this example is 2: 0 and 1).

→ The line `target_predicted = pd.DataFrame(predicted_probabilities[: 1], columns=['class']) ` generates a new Pandas DataFrame called `target_predicted`. This DataFrame contains the probabilities for class 1 (`predicted_probabilities[: 1] `) for each sample in `batch_X`, with the column labeled as 'class'.



```
target_predicted['binary'] = target_predicted['class'].apply(binary_convert)

print(target_predicted.head(10))
test.head(10)
```

```
      class  binary
0  0.998225       1
1  0.668622       1
2  0.995486       1
3  0.998336       1
4  0.961274       1
5  0.999004       1
6  0.997197       1
7  0.991417       1
8  0.997661       1
9  0.659416       1
```

[11]:

	class	pelvic_incidence	pelvic_tilt	lumbar_lordosis_angle	sacral_slope	pelvic_radius	degree_spondylolisthesis
136	1	88.024499	39.844669	81.774473	48.179830	116.601538	56.766083
230	0	65.611802	23.137919	62.582179	42.473883	124.128001	-4.083298
134	1	52.204693	17.212673	78.094969	34.992020	136.972517	54.939134
130	1	50.066786	9.120340	32.168463	40.946446	99.712453	26.766697
47	1	41.352504	16.577364	30.706191	24.775141	113.266675	-4.497958
135	1	77.121344	30.349874	77.481083	46.771470	110.611148	82.093607
100	1	84.585607	30.361685	65.479486	54.223922	108.010218	25.118478
89	1	71.186811	23.896201	43.696665	47.290610	119.864938	27.283985
297	0	45.575482	18.759135	33.774143	26.816347	116.797007	3.131910
4	1	49.712859	9.652075	28.317406	40.060784	108.168725	7.918501

→ The code includes a function, `binary_convert`, that takes a parameter `x`. Within this function, a threshold of 0.65 is set.

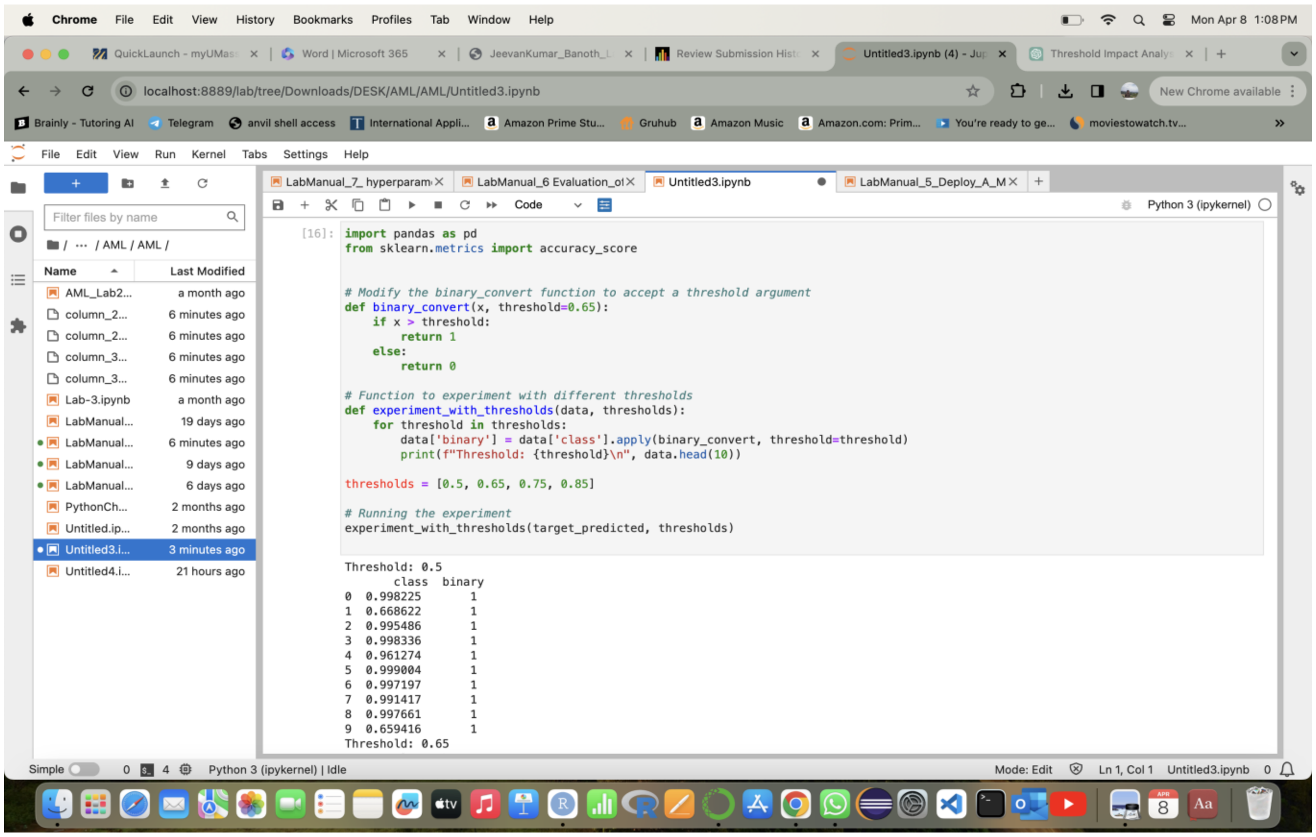
→ It checks if `x` is greater than this threshold. If it is, the function outputs 1; otherwise, it outputs 0.

→ The code then introduces a new column called `'binary'` to the `target_predicted` DataFrame. It uses the `apply` function to apply the `binary_convert` function to the `'class'` column of `target_predicted`.

→ The resulting values are stored in the `'binary'` column. Finally, the code displays the first 10 rows of the `target_predicted` DataFrame, including both the `'class'` and `'binary'` columns, using `print(target_predicted.head(10)) `.

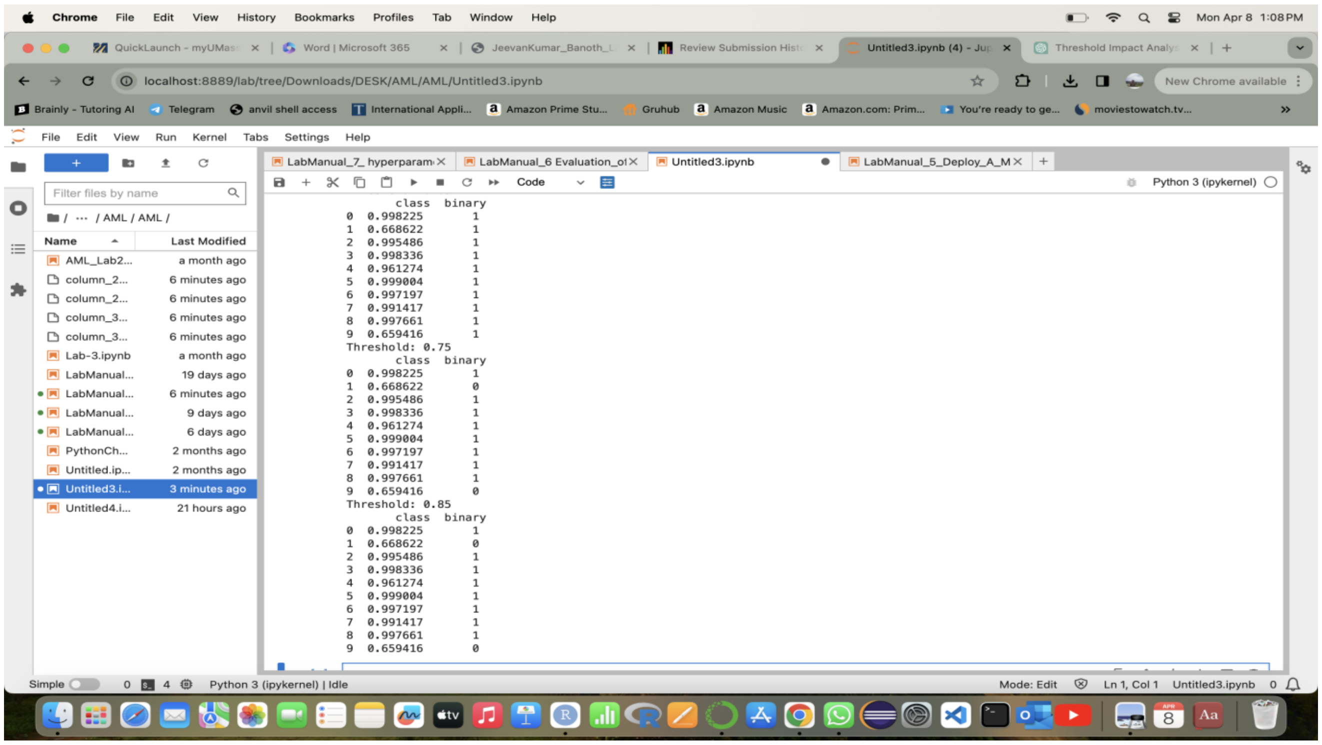
<u>Challenge task:</u> Experiment with changing the value of the threshold. Does it impact the results?

Ans:



```
[16]: import pandas as pd
      from sklearn.metrics import accuracy_score

      # Modify the binary_convert function to accept a threshold argument
      def binary_convert(x, threshold=0.65):
          if x > threshold:
              return 1
          else:
              return 0

      # Function to experiment with different thresholds
      def experiment_with_thresholds(data, thresholds):
          for threshold in thresholds:
              data['binary'] = data['class'].apply(binary_convert, threshold=threshold)
              print(f"Threshold: {threshold}\n", data.head(10))

      thresholds = [0.5, 0.65, 0.75, 0.85]

      # Running the experiment
      experiment_with_thresholds(target_predicted, thresholds)
```

```
Threshold: 0.5
      class  binary
0  0.998225       1
1  0.668622       1
2  0.995486       1
3  0.998336       1
4  0.961274       1
5  0.999004       1
6  0.997197       1
7  0.991417       1
8  0.997661       1
9  0.659416       1
Threshold: 0.65
```



```
      class  binary
0  0.998225       1
1  0.668622       1
2  0.995486       1
3  0.998336       1
4  0.961274       1
5  0.999004       1
6  0.997197       1
7  0.991417       1
8  0.997661       1
9  0.659416       1
Threshold: 0.75
      class  binary
0  0.998225       1
1  0.668622       0
2  0.995486       1
3  0.998336       1
4  0.961274       1
5  0.999004       1
6  0.997197       1
7  0.991417       1
8  0.997661       1
9  0.659416       0
Threshold: 0.85
      class  binary
0  0.998225       1
1  0.668622       0
2  0.995486       1
3  0.998336       1
4  0.961274       1
5  0.999004       1
6  0.997197       1
7  0.991417       1
8  0.997661       1
9  0.659416       0
```

Adjusting the threshold level affects the classification results. A lower threshold allows more cases to be labeled as abnormal, increasing potential false positives. Conversely, a higher threshold classifies more cases as normal, potentially leading to more false negatives.

Finding a suitable threshold is crucial to strike a balance between false positives and false negatives. The specific issue and the relative significance of each error type should guide this decision.

Creating evaluation metrics in the following step offers a deeper evaluation of the model's performance and can assist in deciding whether to fine-tune the model or consider alternative approaches.

Conclusion:

- The data preparation process involves:
    - 1. Acquiring a dataset on vertebral column data from the UCI Machine Learning Repository.
    - 2. Importing the data into a pandas DataFrame.
    - 3. Converting the target variable into binary values.
    - 4. Reorganizing the data columns.
- The data is further divided into separate training and testing/validation sets using a method called stratified sampling to ensure balanced class representation.
- A classifier based on the XGBoost algorithm is then created with predefined parameters optimized for binary classification and AUC (Area Under Curve) as the performance metric.
- The XGBoost classifier is trained using the training data, and the training process is monitored through a printed message.
- Finally, the trained model is evaluated by performing predictions on individual rows and the entire test set.
- A function defines a process to turn predicted probabilities (scores) into clear "yes" or "no" predictions by comparing them to a threshold value.
- This function is used on the predicted probabilities, and the resulting predictions are added as a new column.
- The code then shows both the binary predictions and the real targets from the test set side-by-side.
- This lets you see how well the model is doing and compare the predictions to the actual values.
- This code is a good starting point for more analysis, like calculating metrics to evaluate the model, adjusting its settings, or trying out other machine learning methods.
- It shows how important it is to prepare data carefully, choose the right model, and evaluate predictions in a machine learning process.
- This code demonstrates a practical approach to creating a binary classification model using XGBoost.

- It also lays the groundwork for more complex machine learning techniques and refining the performance of the model.
- This code provides a detailed workflow for binary classification using XGBoost, covering the following steps:
- Display original test set rows, including predicted probabilities and actual labels.
- Evaluate model performance on the test set using accuracy, precision, and other metrics.
- Optimize XGBoost hyperparameters using the validation set to improve model performance.
- Adjust classification thresholds based on the misclassification costs in your specific application.
- The code serves as a foundation for binary classification with XGBoost and includes guidance on:
    - Data preparation
    - Model training
    - Prediction generation
    - Necessary evaluations
- We can customize this code to explore different data features and optimize the model according to your needs.

--------X-------