**MODULE 1 - ASSIGNMENT 2 - REPORT**

**SANTANDER CUSTOMER TRANSACTION PREDICTION**

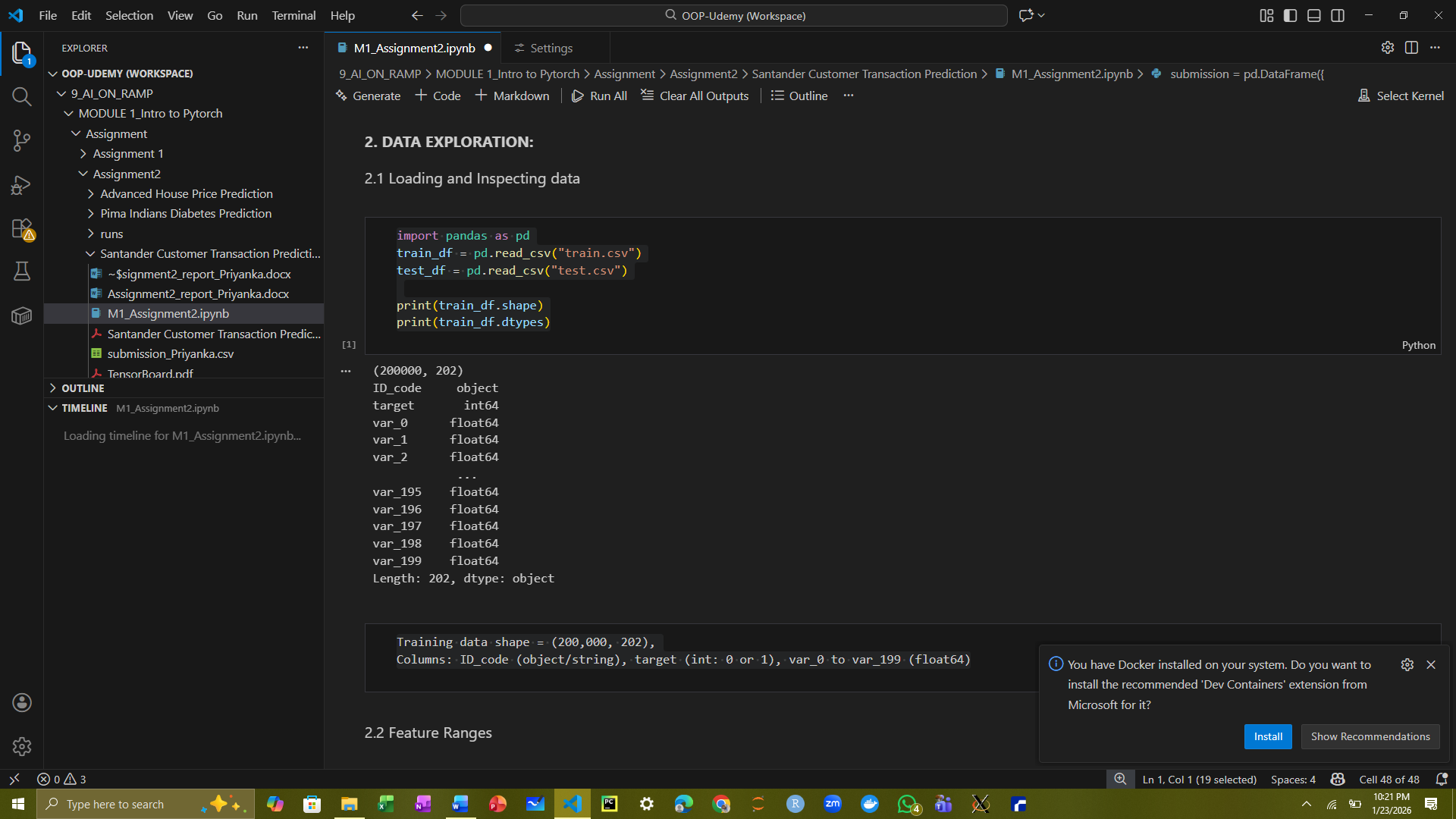
* ***Priyanka Prem Kumar***

1. **DESCRIPTION OF THE DATASET:**

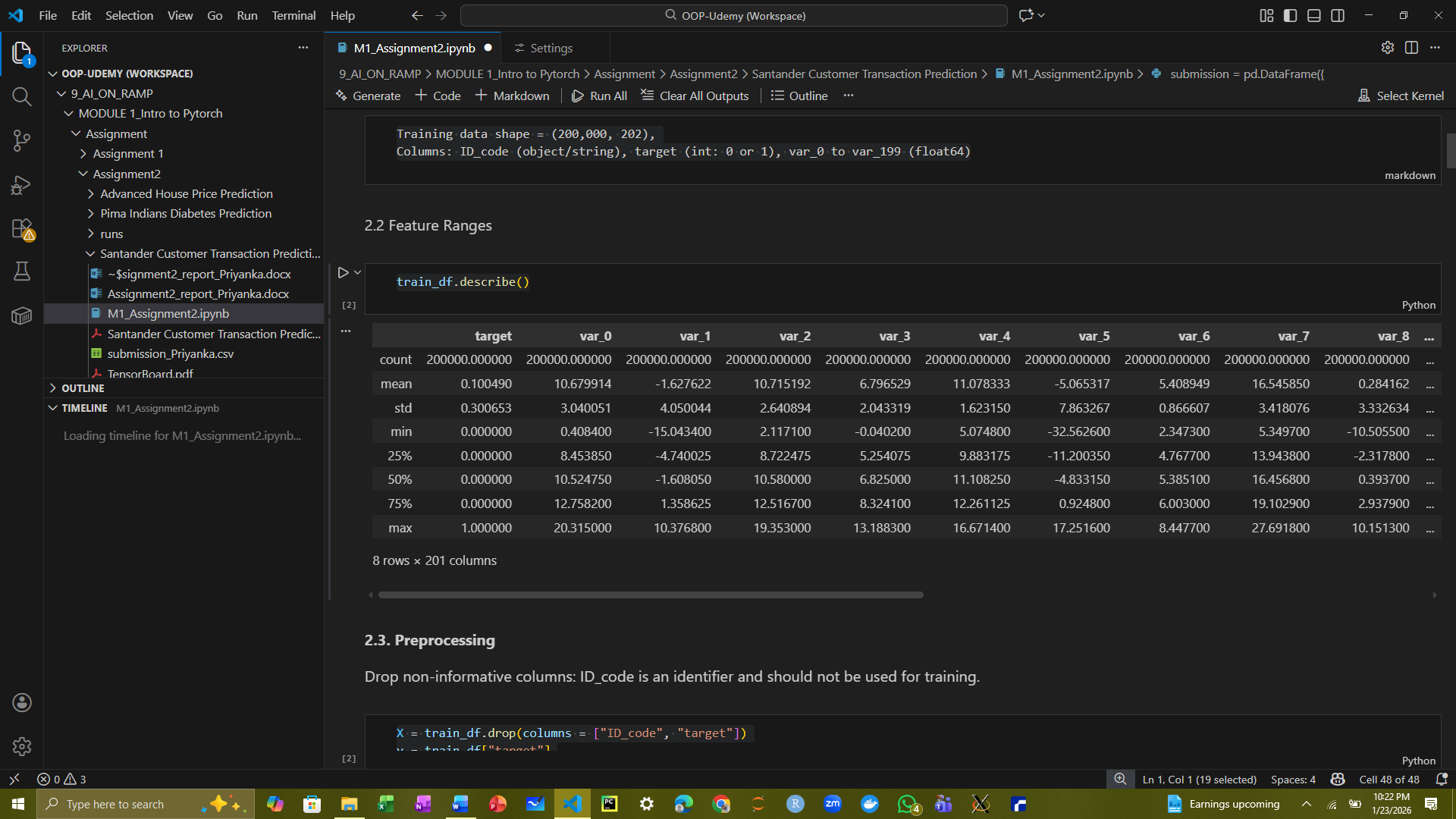
The Santander Customer Transaction Dataset is a binary classification dataset released by Santander Bank and commonly used in machine learning competitions. It contains anonymized customer-level data where the objective is to predict whether a customer will make a specific transaction in the future. The training dataset consists of approximately 200,000 samples with 200 numerical features (var\_0 to var\_199), along with an ID\_code for identification and a binary target variable indicating transaction occurrence. The test dataset has a similar structure but excludes the target column. A notable characteristic of the dataset is its strong class imbalance, with roughly 90% of samples belonging to the negative class (target = 0) and about 10% to the positive class (target = 1).

1. **DATA EXPLORATION**
   1. **Loading and inspecting data**

The inspection begins with imports the pandas library and reads the training and test datasets from CSV files into pandas Dataframes named train\_df and test\_df. It then displays the data types of each column. This provides a quick overview of the dataset’s size, structure, and the type of data contained in each feature, which is essential for planning preprocessing and model building.

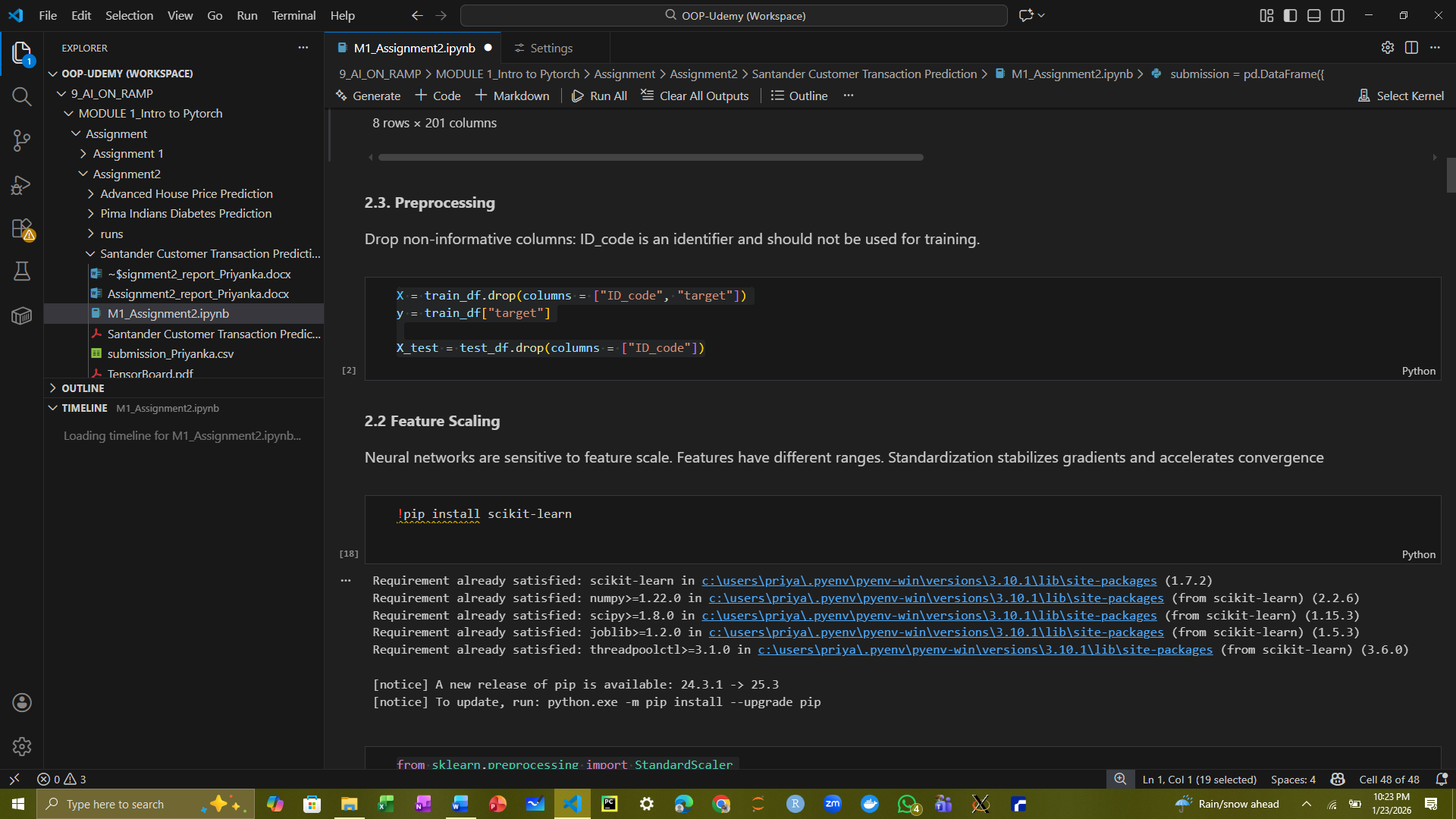


* 1. Feature ranges



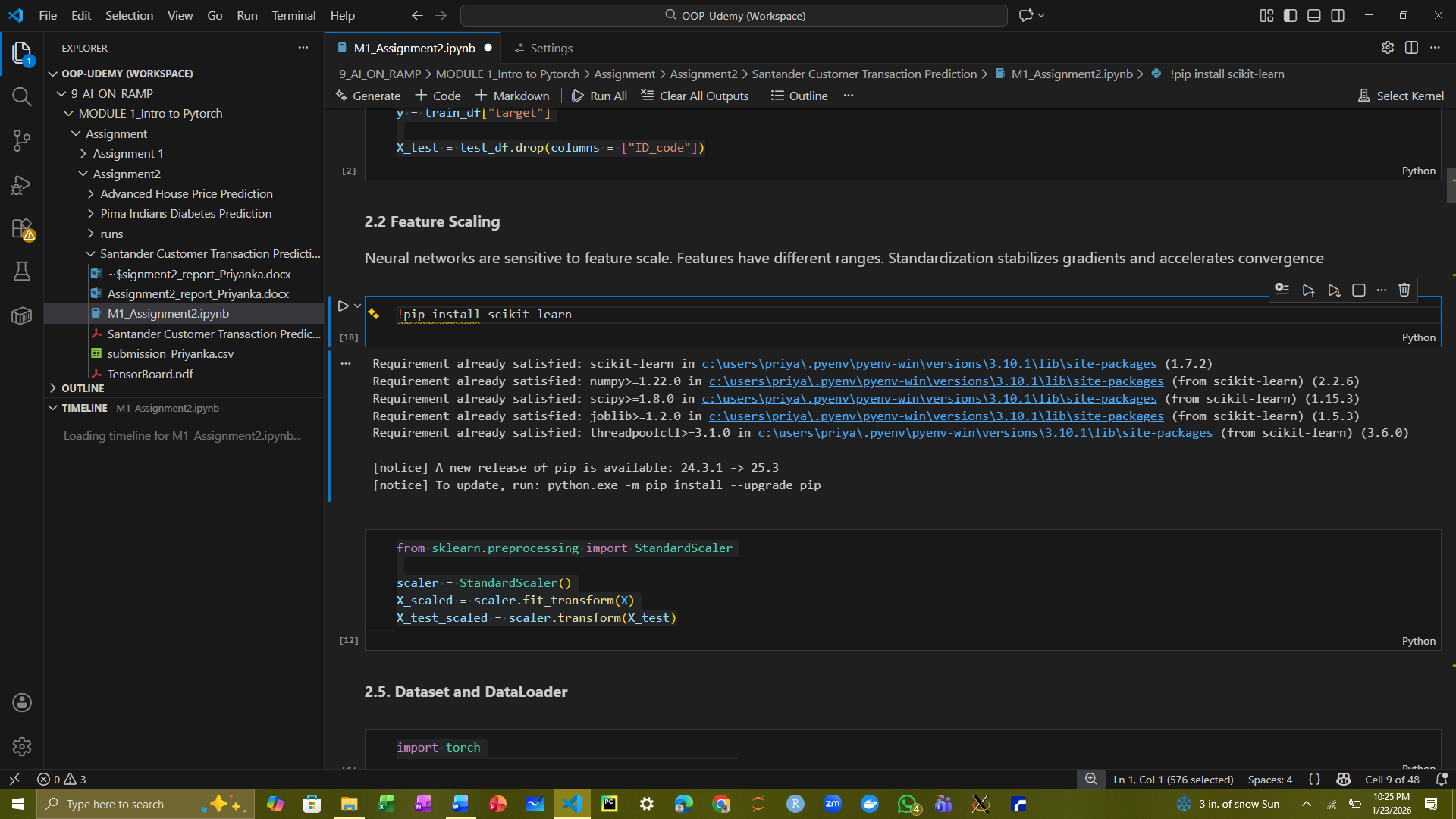
The train\_df.describe() generates a statistical summary of the numeric columns in the training dataset. It provides key metrics such as count, mean, standard deviation, minimum, maximum, and the 25th, 50th (median), and 75th percentiles. This summary helps to understand the distribution, range, and scale of each feature, identify potential outliers, and guide preprocessing decisions before training the model.

* 1. Preprocessing



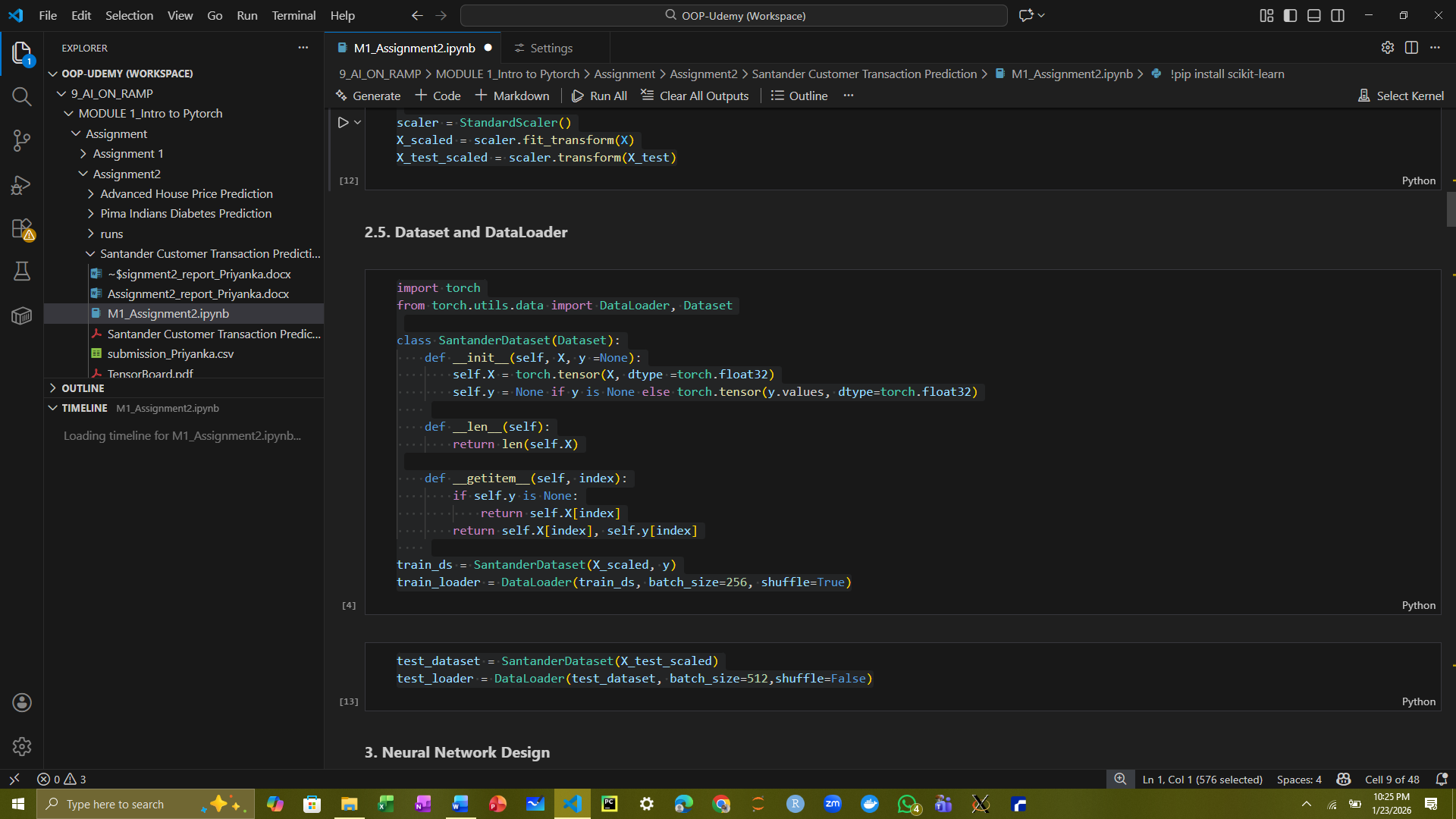
This code part separates the features and target variable from the datasets. In the training set, X contains all the input features, obtained by dropping the ID\_code and target columns, while y contains the target labels. For the test set, X\_test includes only the feature columns, with ID\_code removed since it is not needed for model training or inference. This prepares the data for preprocessing and input into the neural network.

* 1. Feature scaling



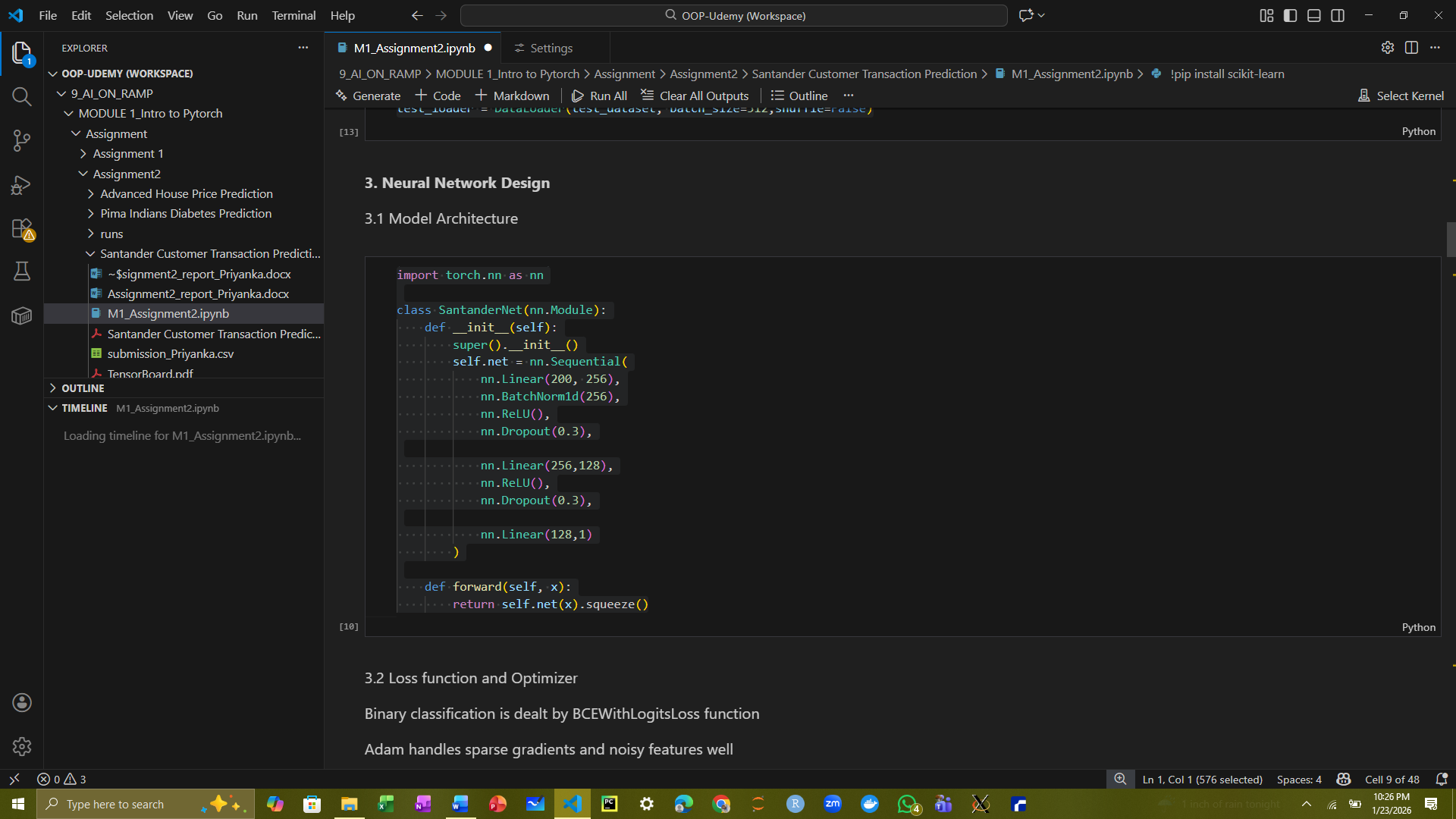
This code standardizes the feature values using StandardScaler from scikit-learn. The scaler is first fit to the training features X and transforms them to have zero mean and unit variance, producing X\_scaled. The same scaling parameters are then applied to the test features X\_test using transform, resulting in X\_test\_scaled. Standardizing the features ensures that all inputs are on a similar scale, which helps the neural network train more effectively and converge faster.

* 1. Dataset and DataLoader



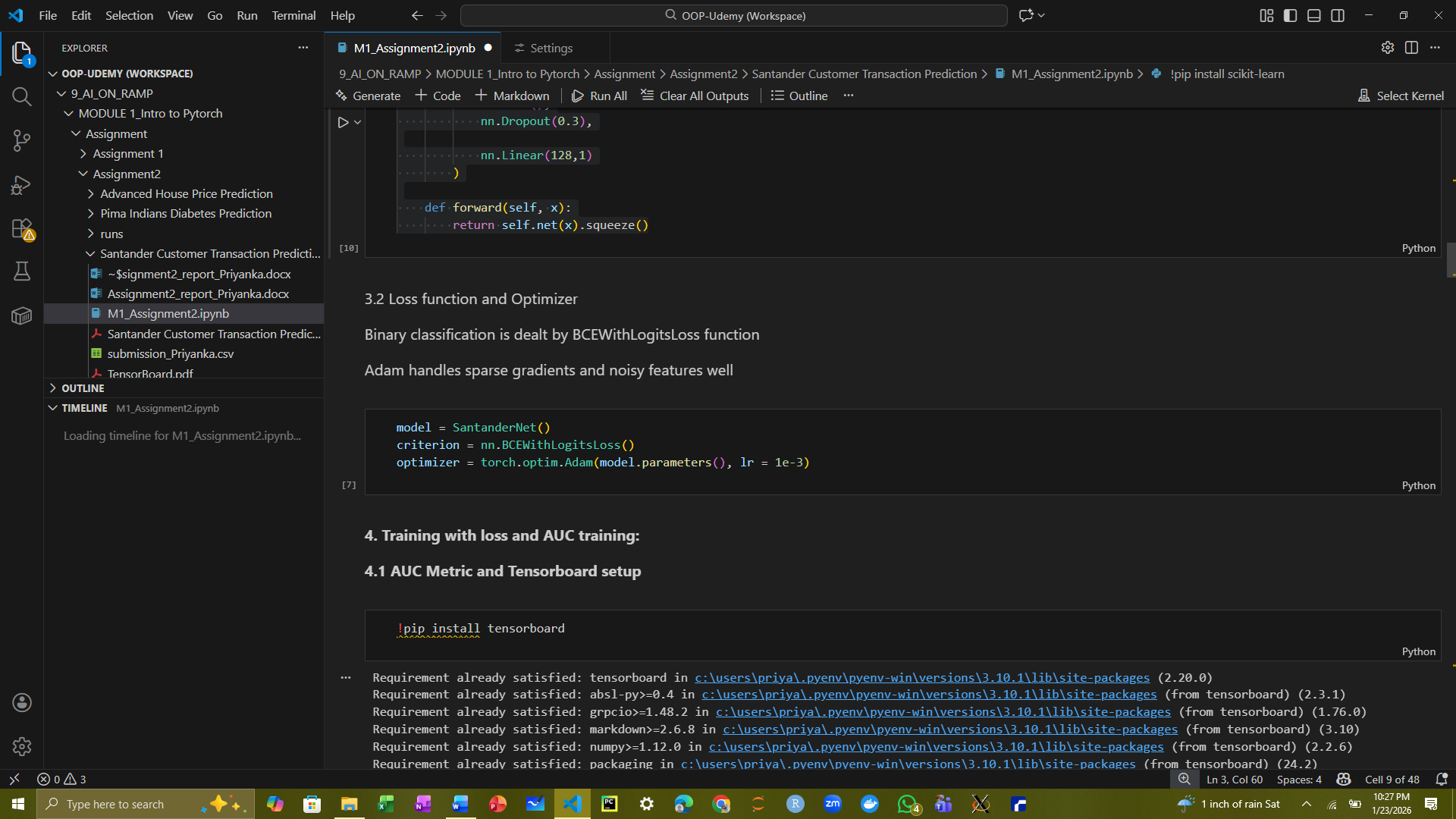
This code defines a custom PyTorch dataset class SantanderDataset to handle both training and test data. The class converts input features X and target labels y into PyTorch tensors of type float32. For training, a DataLoader is created with a batch size of 256 and shuffling enabled to feed the neural network during training. For the test set, a separate DataLoader is created with a batch size of 512 and no shuffling.

1. NEURAL NETWORK DESIGN
   1. Model architecture



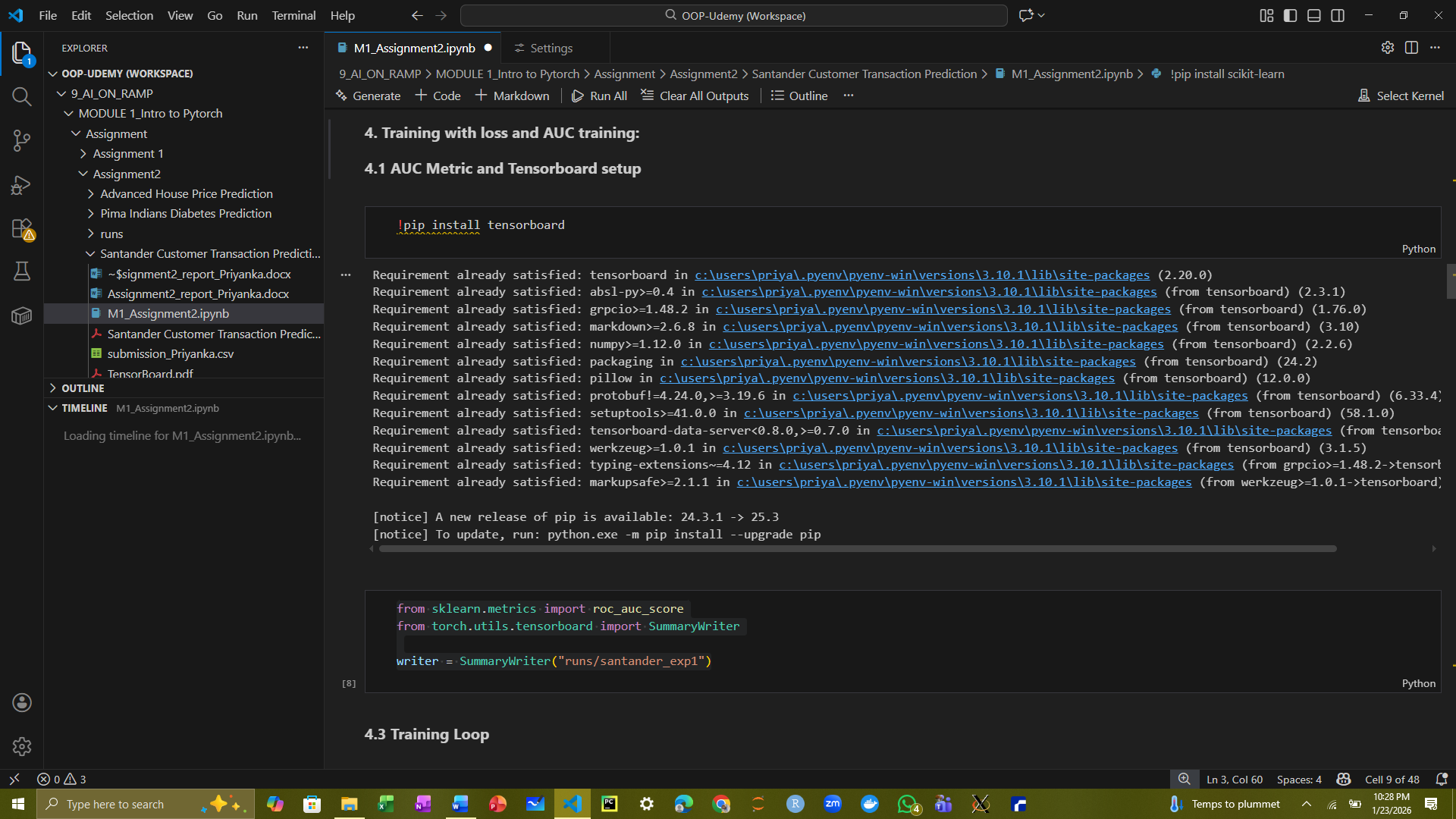
This code defines a neural network class SantanderNet using PyTorch’s nn.Module. The network consists of three fully connected (Linear) layers with sizes 200→256→128→1, appropriate for the 200 input features in the dataset. Batch normalization is applied after the first layer to stabilize and accelerate training, and ReLU activations introduce non-linearity. Dropout layers with a rate of 0.3 are included after the first two hidden layers to reduce overfitting. The final layer outputs a single value for each sample, and .squeeze() ensures the output has the correct shape for binary classification. This network is designed to predict the probability of the target class after applying sigmoid activation.

* 1. Loss function and optimizer



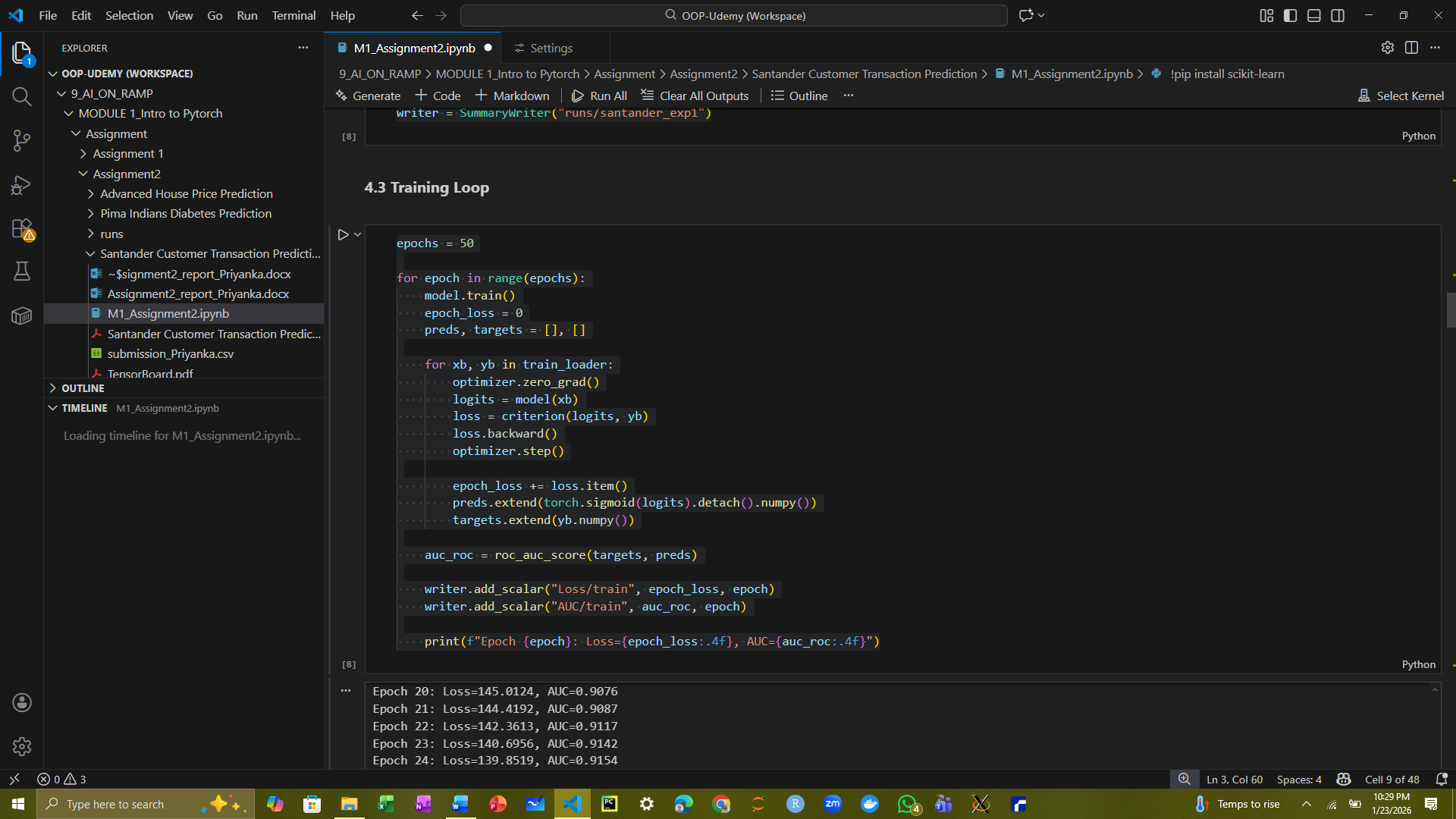
This code initializes the neural network, loss function, and optimizer for training. The loss function nn.BCEWithLogitsLoss() is used for binary classification, combining a sigmoid activation and binary cross-entropy in a numerically stable way. The optimizer is set to Adam with a learning rate of 0.001, which adapts the learning rate for each parameter and helps the model converge efficiently during training.

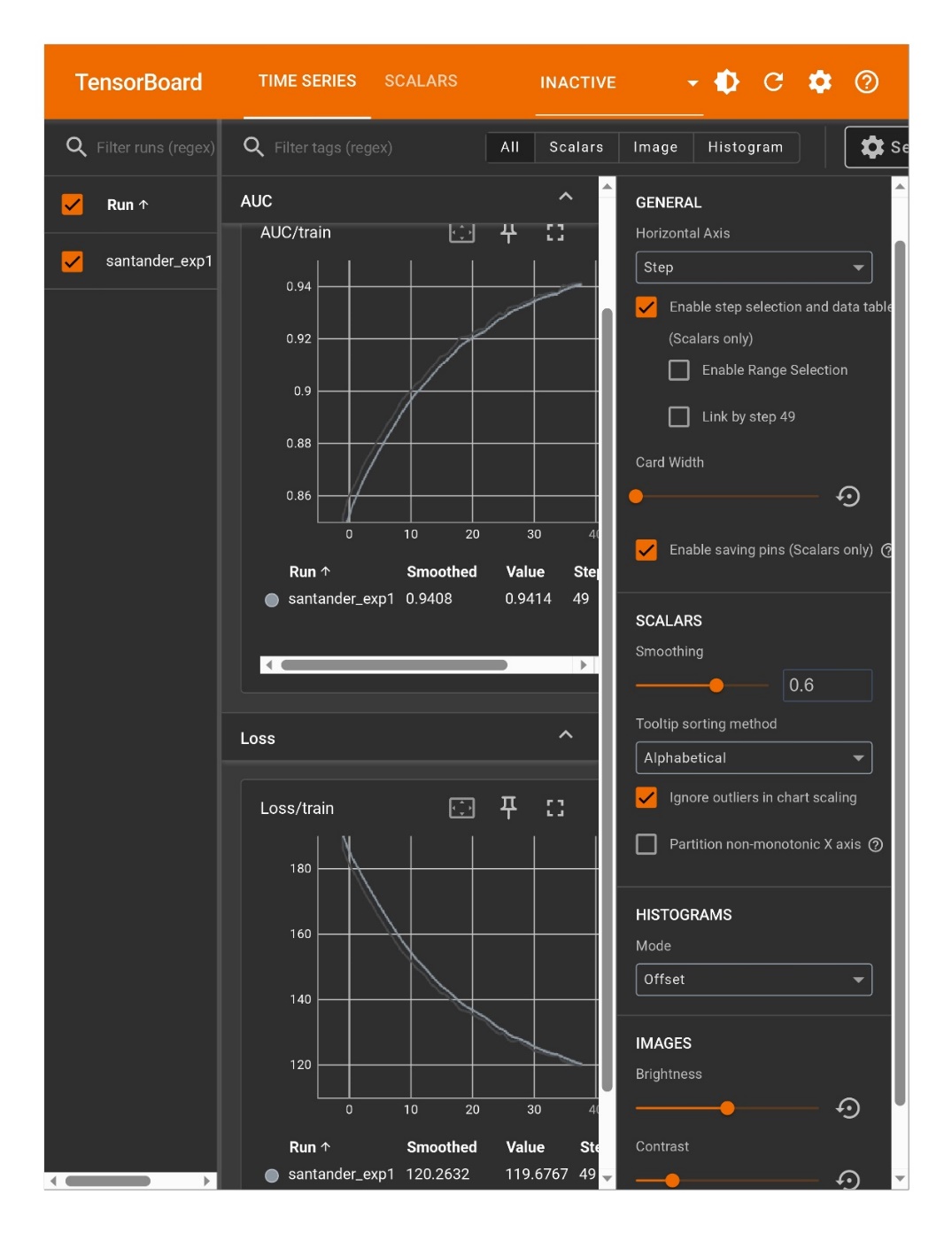
1. TRAINING WITH LOSS AND AUC
   1. Auc metric and tensorboard setup



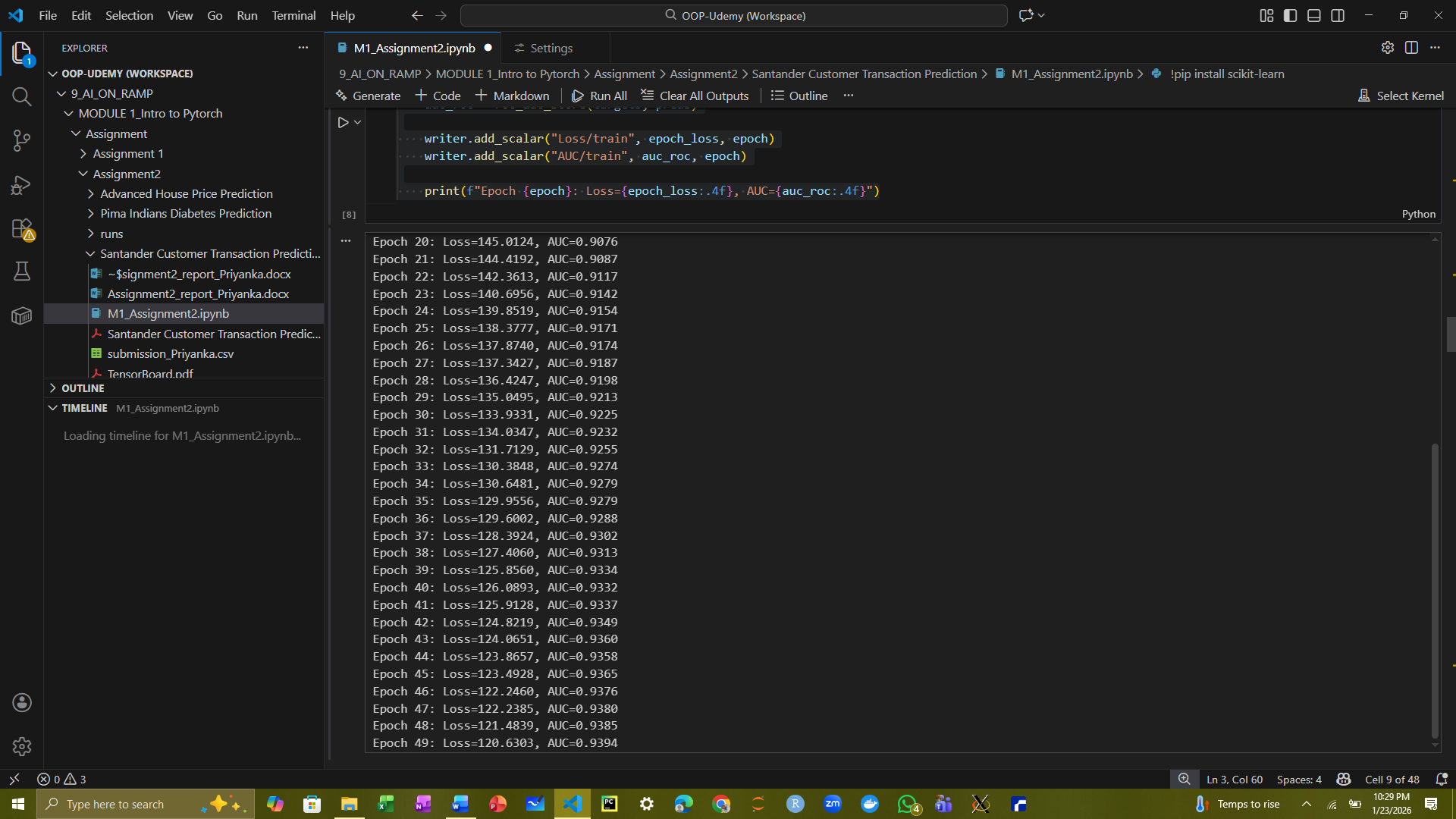
This code imports tools for evaluating model performance and visualizing training progress. roc\_auc\_score from scikit-learn is used to calculate the area under the ROC curve (AUC), which measures the model’s ability to distinguish between the two target classes. SummaryWriter from PyTorch’s tensorboard module is used to log training metrics, such as loss and AUC, so they can be visualized in TensorBoard. The writer is initialized with the directory "runs/santander\_exp1", where the logs will be stored. This setup enables monitoring the model’s performance across epochs during training.

* 1. Training loop

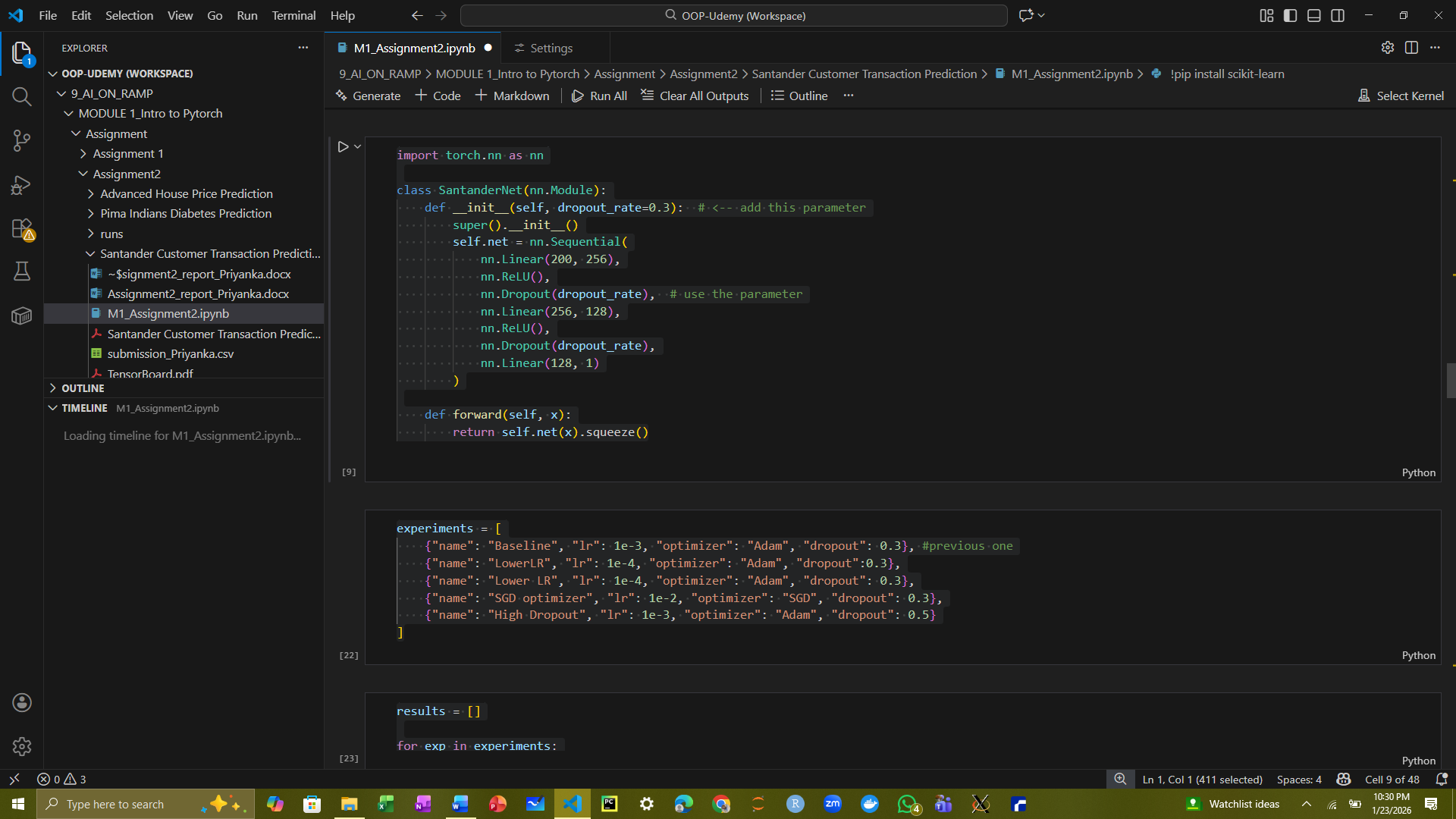


Next we implement the training loop for the neural network over 50 epochs. For each epoch, the model is set to training mode, and the loss is accumulated while iterating over batches from train\_loader. Predictions are obtained by applying a sigmoid to the model outputs, and both predictions and true labels are stored to calculate the ROC–AUC score at the end of each epoch. Training metrics, including the epoch loss and AUC, are logged to TensorBoard using SummaryWriter for visualization. 

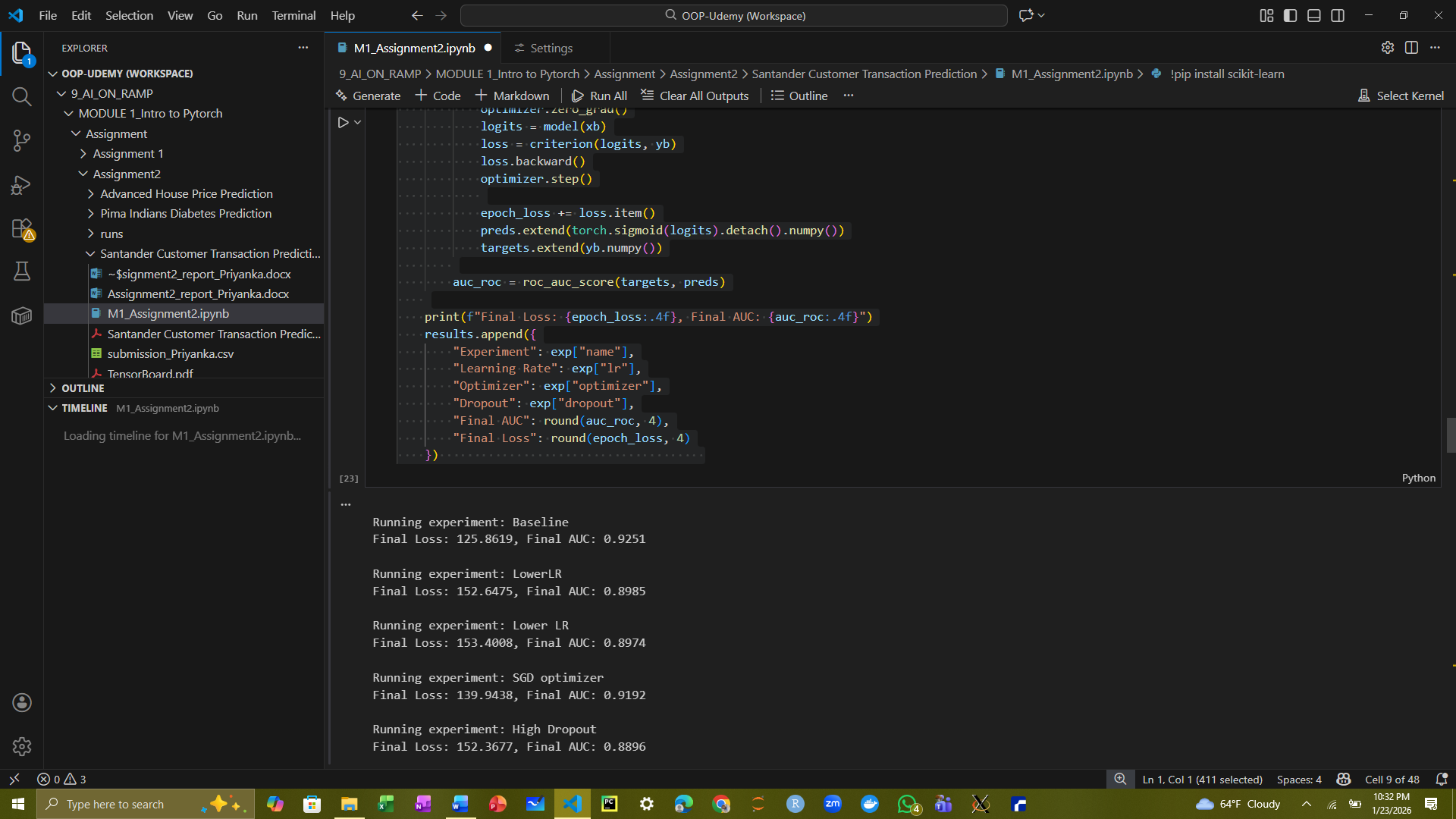
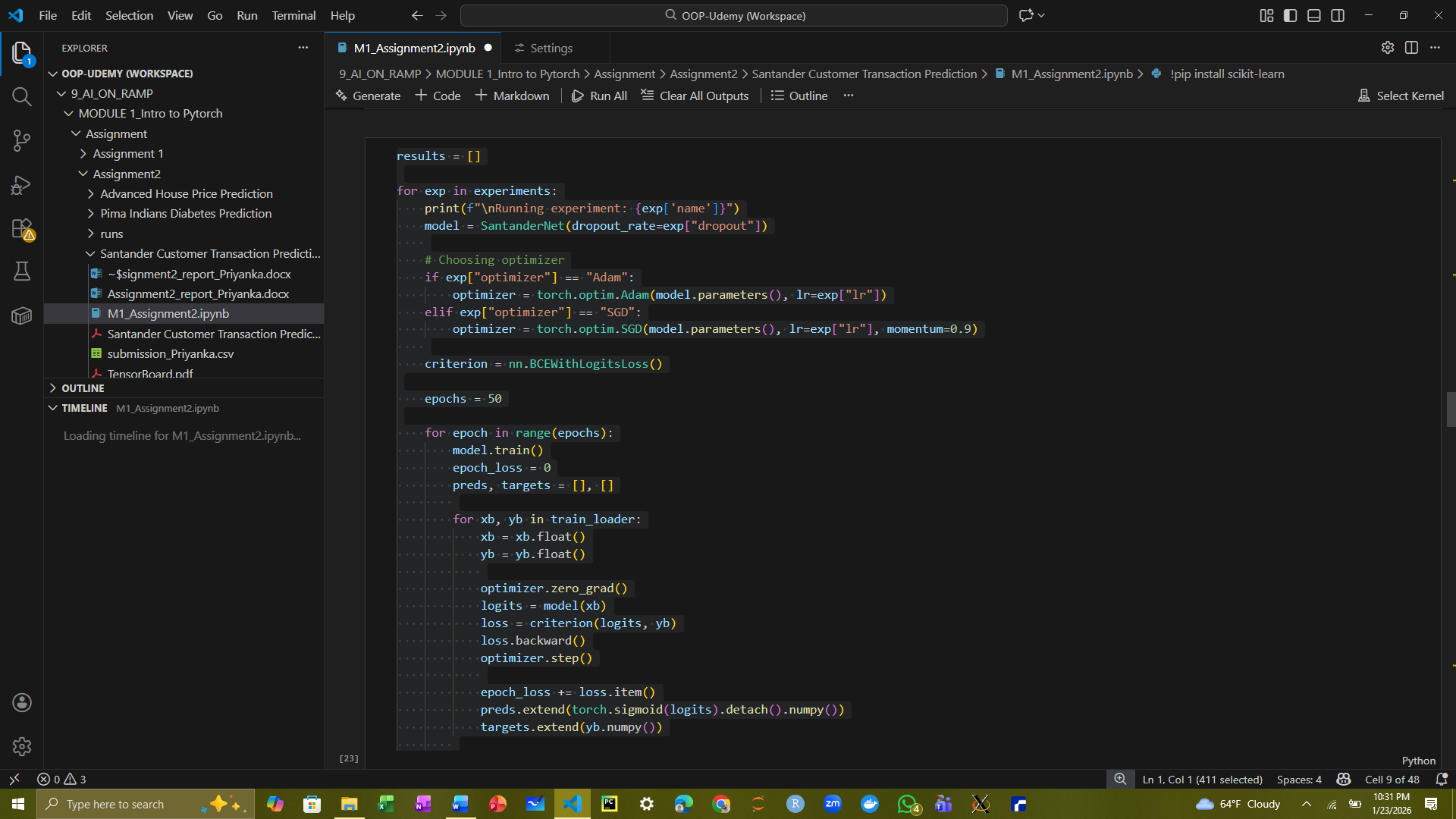
The printed output shows that the loss steadily decreases and the AUC consistently increases across epochs, reaching a final training AUC of approximately 0.9394, indicating that the model is learning effectively and distinguishing between the two target classes with high AUC.



1. OPTIMIZATION USING HYPERPARAMETERS
   1. Try different parameters (like training epochs, learning rate, etc.) of your model, and also try different optimizers. Check whether these changes can improve the model's performance.



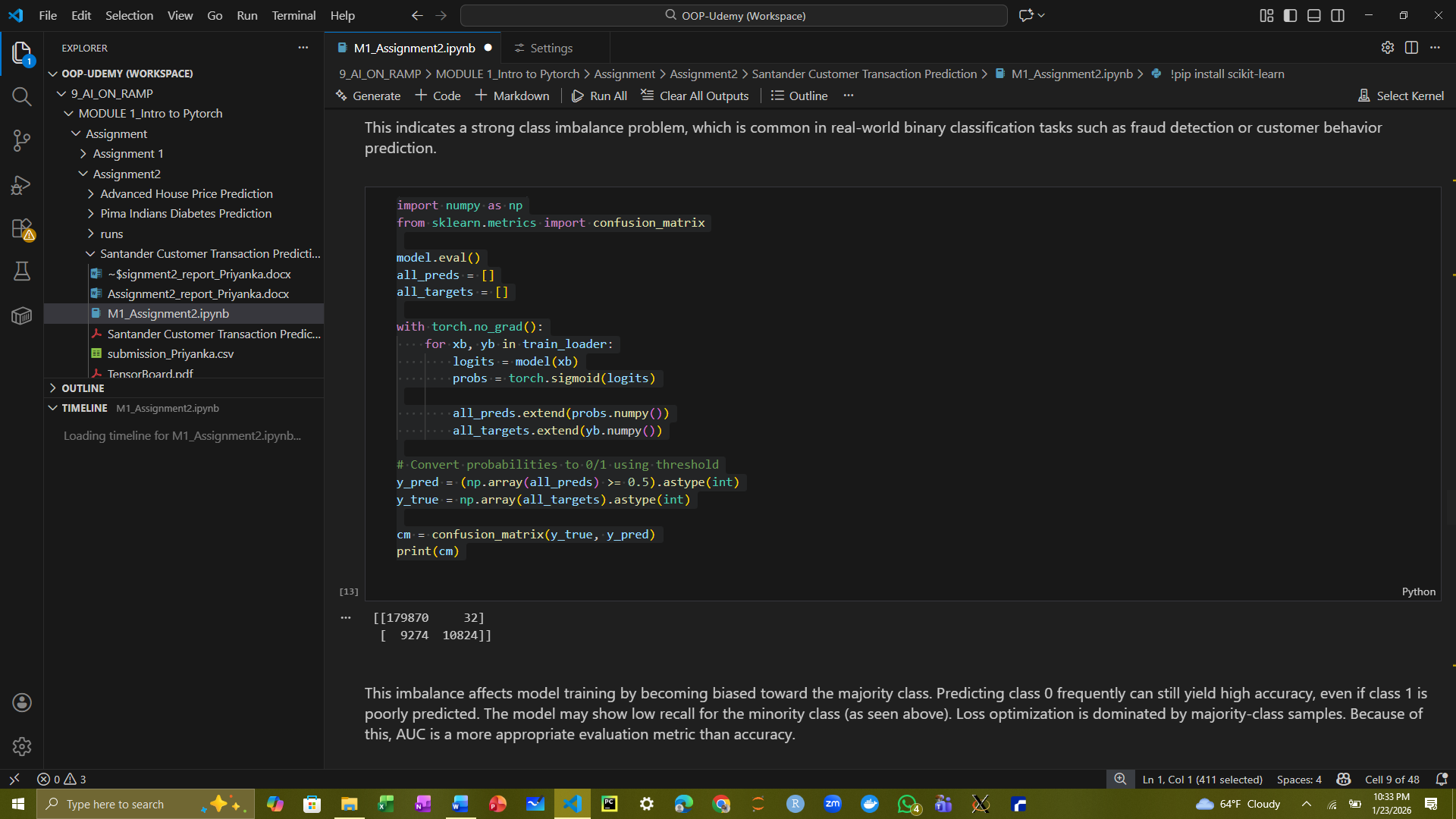
Here we run a series of experiments to evaluate how different hyperparameters affect the neural network’s performance. Each experiment varies the learning rate, optimizer, or dropout rate while keeping other settings the same. The model is trained for 50 epochs for each configuration, and the final loss and ROC–AUC score are recorded.



The results show that the baseline model with Adam optimizer, learning rate 0.001, and 0.3 dropout achieved the best performance (AUC = 0.9251). Lowering the learning rate or increasing the dropout reduced performance, while using SGD instead of Adam slightly decreased the AUC. These observations indicate that the model is sensitive to learning rate and regularization, and careful tuning of hyperparameters is important to maximize its ability to distinguish between the two target classes.

* 1. Write a code to count the amount of data samples with different target labels. What do you observe about the distribution of the target labels? How did this situation affect your model training and performance?

The code counts the number of samples for each target label in the training dataset using value\_counts(). The output shows that the dataset is highly imbalanced, with 179,902 samples labeled as class 0 and only 20,098 samples labeled as class 1. This means roughly 90% of the data belongs to the majority class (0) and only 10% to the minority class (1). Such a strong class imbalance can affect model training and performance, as the neural network may become biased toward predicting the majority class. This scenario is common in real-world binary classification tasks, such as fraud detection or customer behavior prediction, and often requires techniques like class weighting, oversampling, or undersampling to improve model learning for the minority class.



This code evaluates the trained model on the training data and computes the confusion matrix. First, the model is set to evaluation mode with model.eval(), and predictions are generated using a sigmoid activation to get probabilities. These probabilities are converted to class labels (0 or 1) using a threshold of 0.5. The confusion\_matrix function from scikit-learn compares the true labels (y\_true) with the predicted labels (y\_pred).

The output can be interpreted as follows:

* **True Negatives (TN): 179,870** — class 0 samples correctly predicted.
* **False Positives (FP): 32** — class 0 samples incorrectly predicted as class 1.
* **False Negatives (FN): 9,274** — class 1 samples incorrectly predicted as class 0.
* **True Positives (TP): 10,824** — class 1 samples correctly predicted.

This shows the model performs very well on the majority class (0) but makes more errors on the minority class (1), which is expected due to the strong class imbalance in the dataset.

* 1. What techniques can you use to solve this issue to improve model training, and make the training data received by the model more balanced? Do you think these techniques will improve your model’s performance? Why or why not?

The Techniques used to handle imbalance:

1. Class weighting in the loss function: Assign higher penalty to misclassification of minority-class samples

2. Oversampling the minority class: Duplicate or resample minority samples more frequently during training

3. Undersampling the majority class: Reduce the number of majority-class samples

4. Synthetic data generation (e.g., SMOTE): Generate artificial minority-class samples

References:

1. Goodfellow, I., Bengio, Y., & Courville, A. (2016).Deep Learning. MIT Press. Chapter 6: Optimization for Training Deep Models.

2. Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-sampling Technique.

Journal of Artificial Intelligence Research, 16, 321–357.

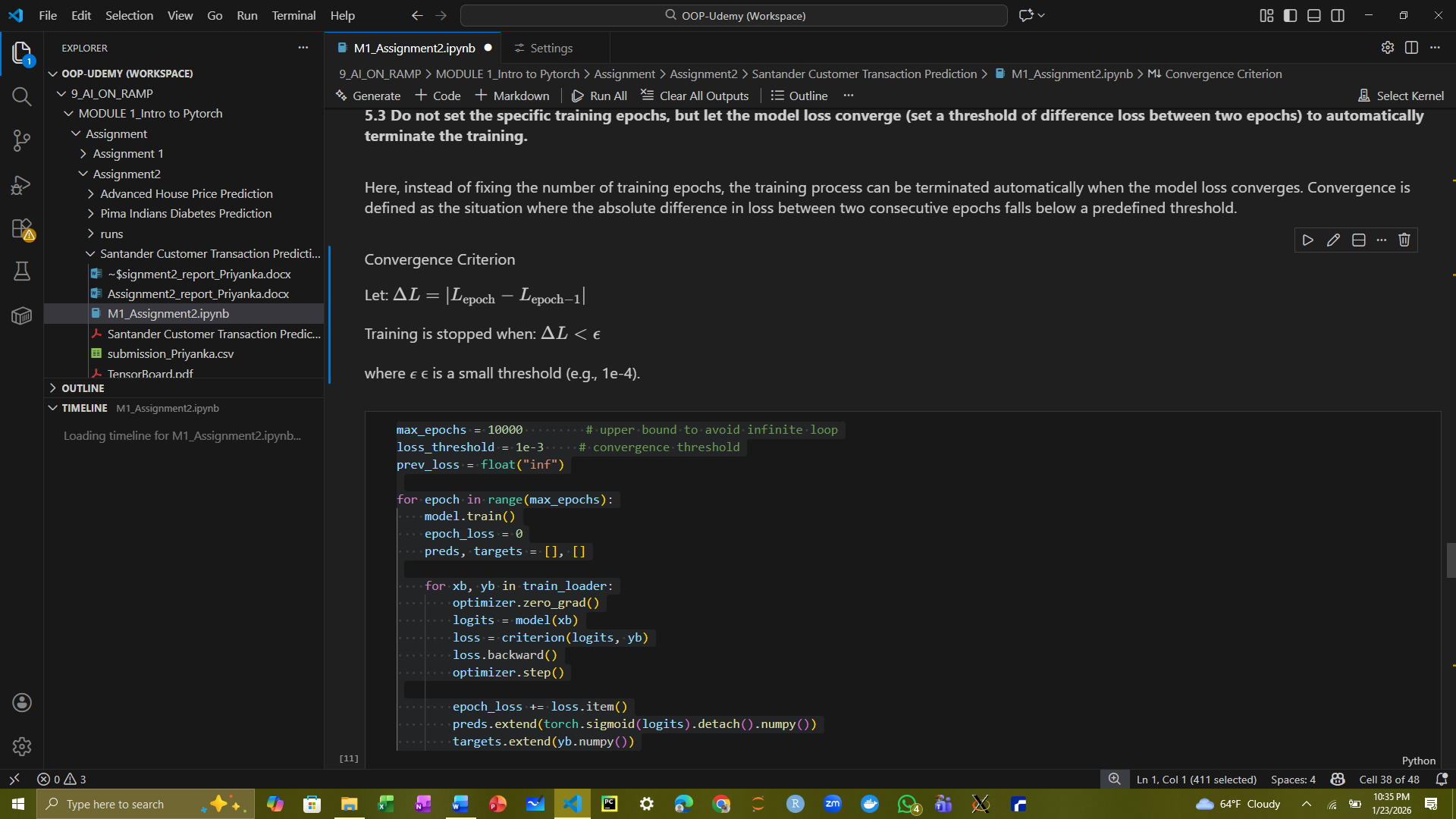
3. He, H., & Garcia, E. A. (2009). Learning from Imbalanced Data. IEEE Transactions on Knowledge and Data Engineering, 21(9), 1263–1284.

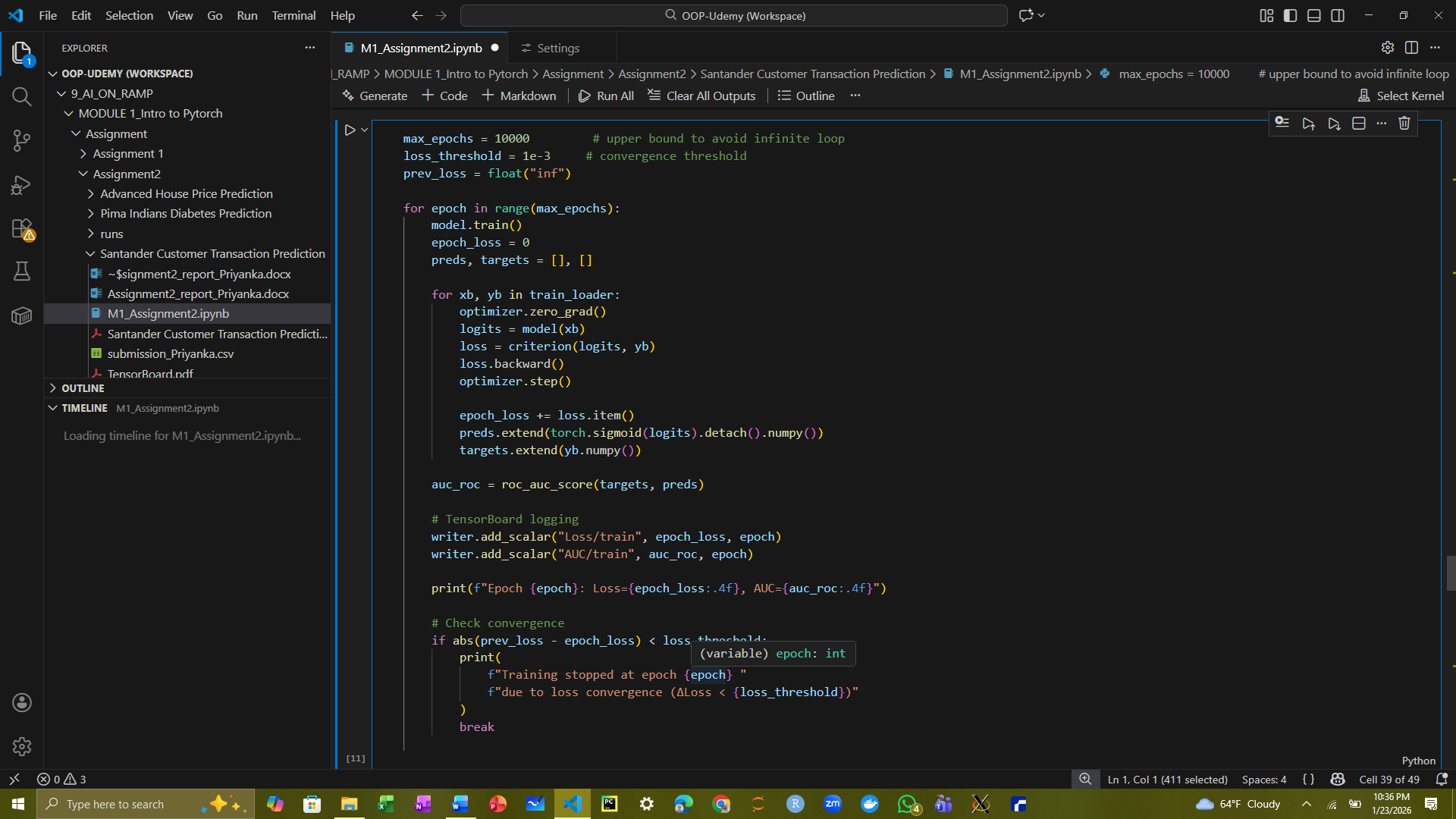
4. Han, H., Wang, W.-Y., & Mao, B.-H. (2005). Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning.

Yes, these techniques can improve model performance, particularly in minority-class recall, AUC score, and model robustness. However, they may also introduce trade-offs like oversampling can cause overfitting and undersampling may discard useful information

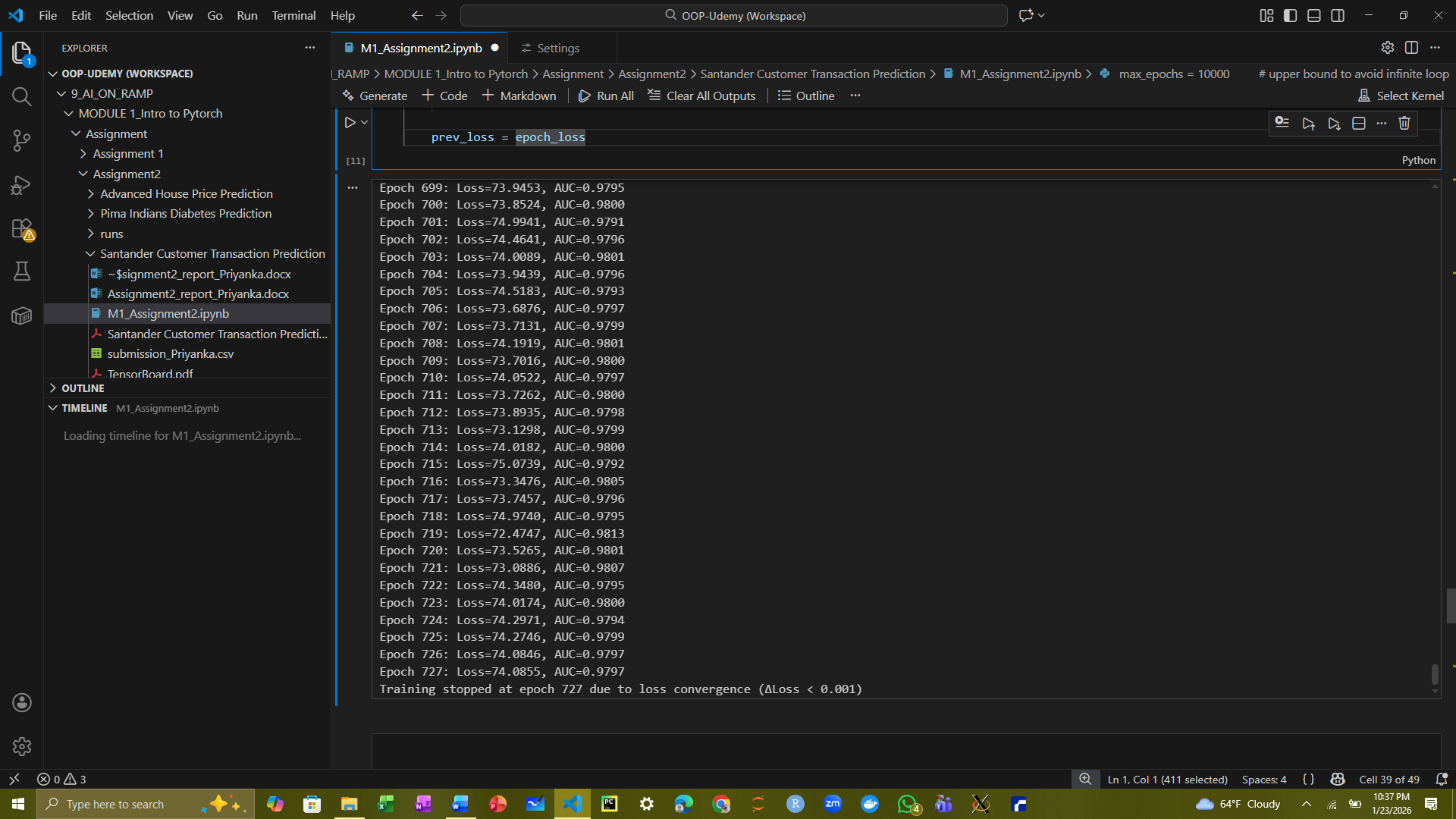
* 1. Do not set the specific training epochs, but let the model loss converge (set a threshold of difference loss between two epochs) to automatically terminate the training

Here, instead of fixing the number of training epochs, the training process can be terminated automatically when the model loss converges. Convergence is defined as the situation where the absolute difference in loss between two consecutive epochs falls below a predefined threshold.





This code modifies the training loop to stop automatically when the model’s loss converges, rather than training for a fixed number of epochs. The max\_epochs sets an upper bound to prevent infinite loops, while loss\_threshold defines the minimum change in loss (ΔLoss) between consecutive epochs that counts as convergence. During each epoch, the model is trained on batches from train\_loader, and the loss and predictions are recorded. The ROC–AUC score is calculated for evaluation. After each epoch, the difference between the previous and current epoch loss is checked. If the change is smaller than loss\_threshold, training stops early, and a message is printed indicating the epoch at which convergence occurred. This approach ensures that the model trains efficiently.



The training loop ran for 727 epochs before automatically stopping due to convergence of the loss, as specified by the threshold ΔLoss < 0.001. Across these epochs, the loss fluctuated around 73–75, and the ROC–AUC consistently remained high, around 0.980, indicating that the model learned to distinguish between the two classes very effectively. The early stopping mechanism prevented unnecessary training beyond this point, saving computation time while ensuring the model reached a stable performance. This shows that automatic convergence-based stopping can be an efficient alternative to manually specifying a fixed number of epochs.

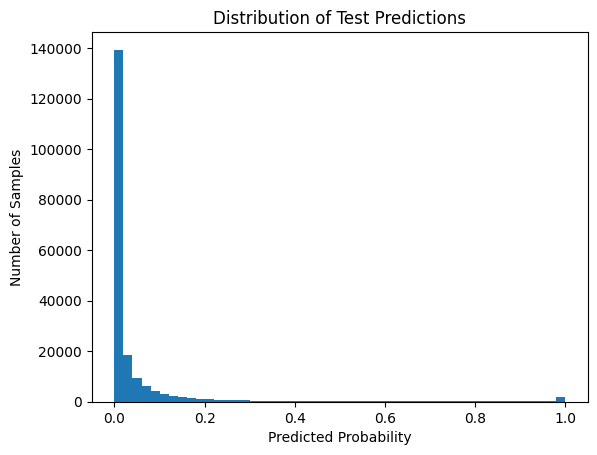
1. INFERENCE USING TEST DATA AND SUBMIT TO KAGGLE
   1. Running Inference on Test Data

Since the baseline model was identified as the best-performing configuration, it is used to generate predictions on the test dataset. The model is set to evaluation mode with model.eval() to disable dropout and batch normalization updates. Using torch.no\_grad() ensures that gradient calculations are skipped, reducing memory usage and speeding up inference. For each batch in test\_loader, the network outputs logits, which are converted to probabilities with a sigmoid activation. These predicted probabilities are collected in the test\_preds list.

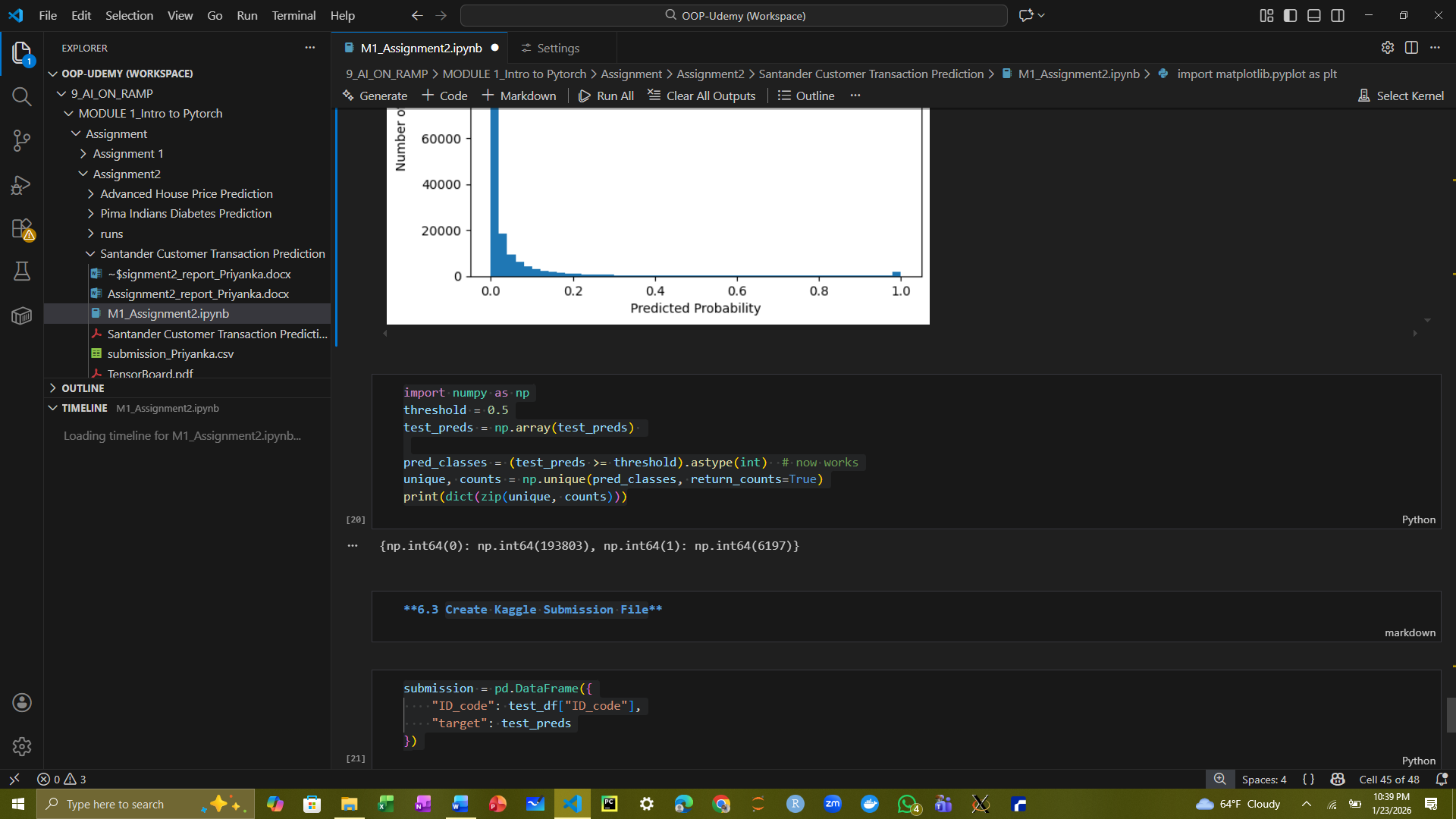


* 1. Count how many predictions would be class 1 if threshold=0.5

From the histogram you shared, it looks like the majority of predictions are close to 0, so the number of predicted class 1 samples will likely be very small.



The code converts the predicted probabilities to class labels using a threshold of 0.5 and counts how many predictions fall into each class.

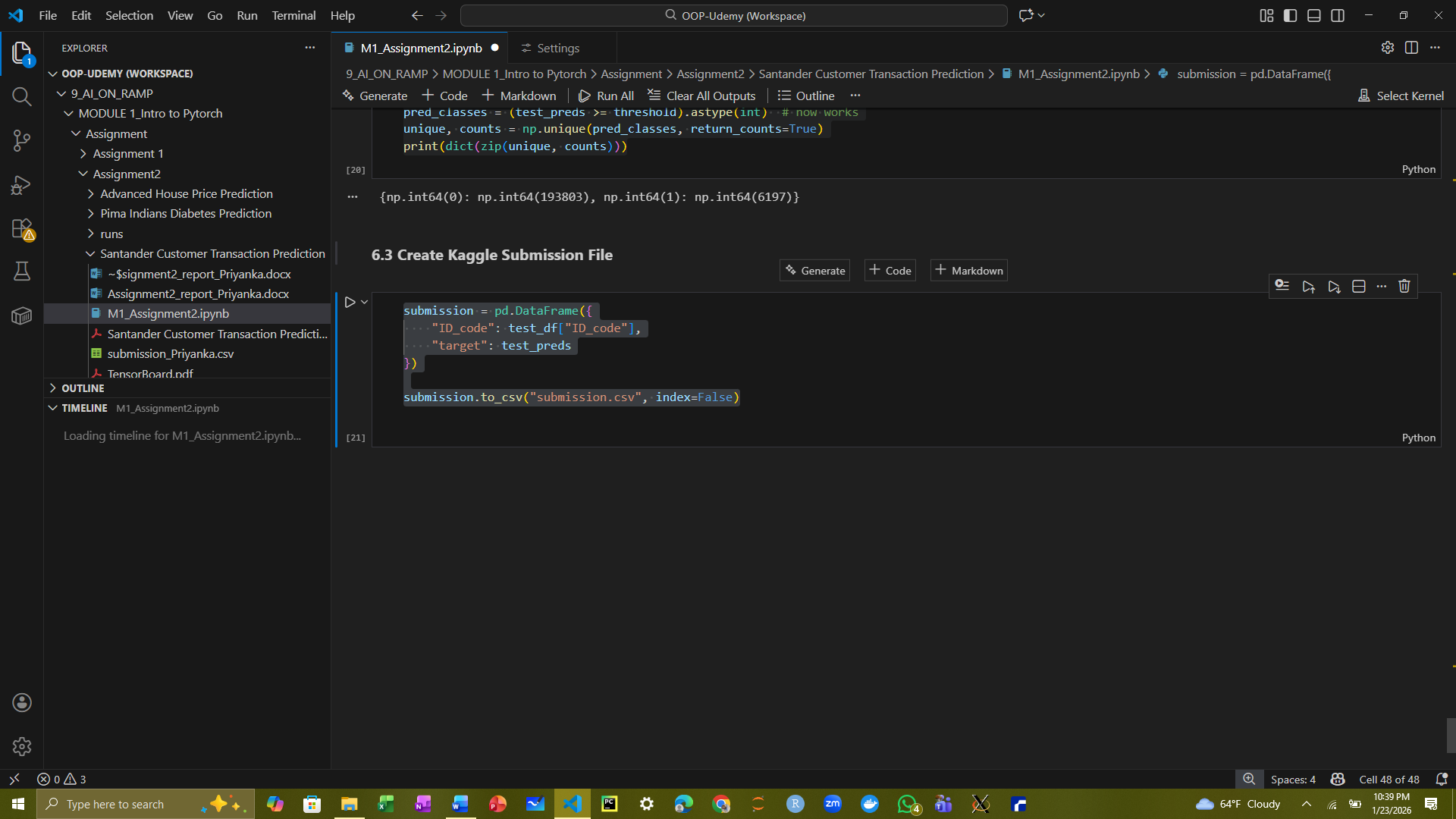


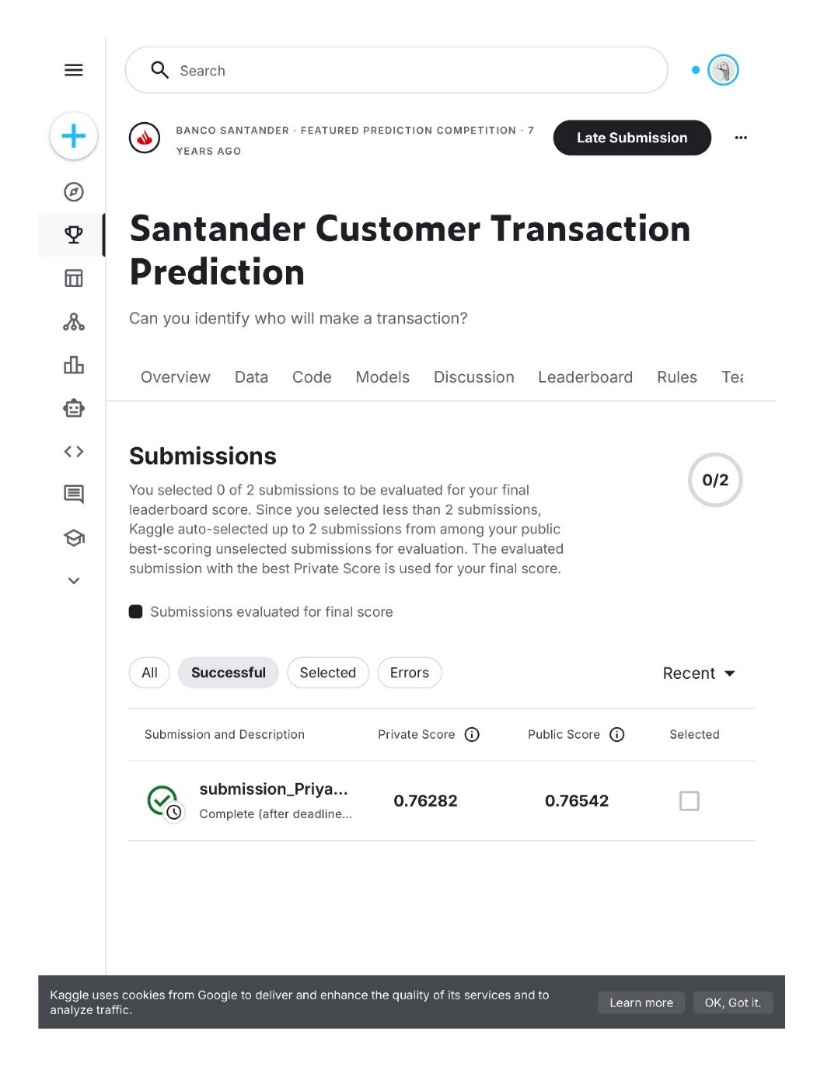
This means:

* **193,803 samples** were predicted as class 0
* **6,197 samples** were predicted as class 1

It shows that most of the test samples are predicted as class 0, which aligns with the class imbalance in the training data and the distribution seen in your test prediction histogram.

* 1. Create Kaggle Submission File:





\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*