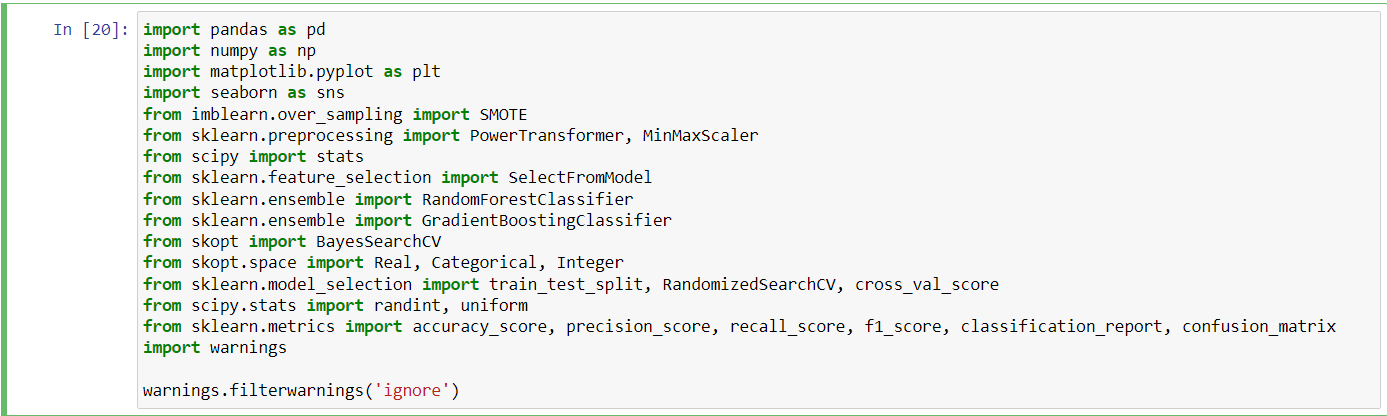
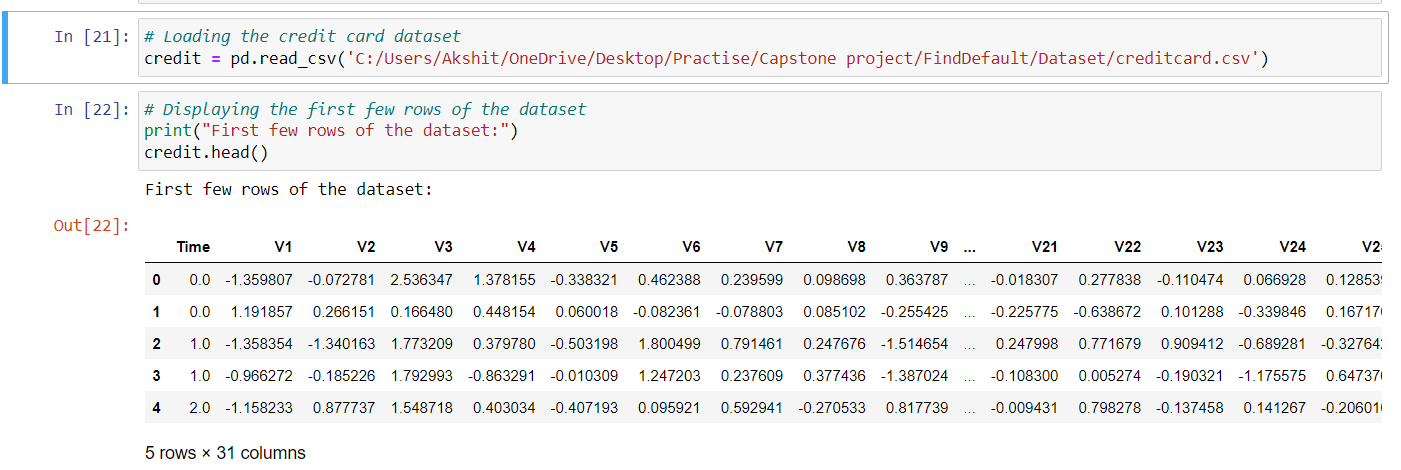
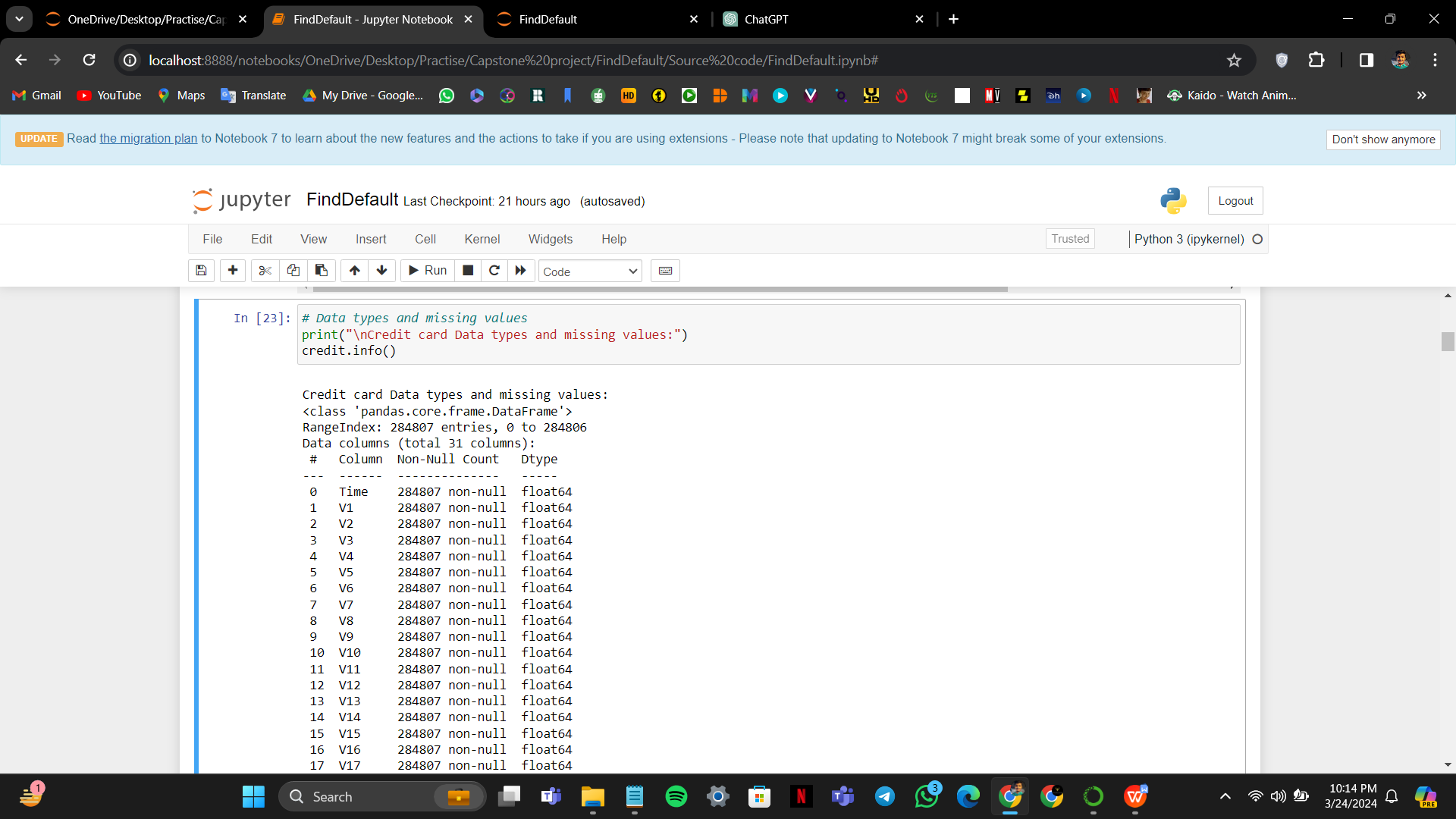
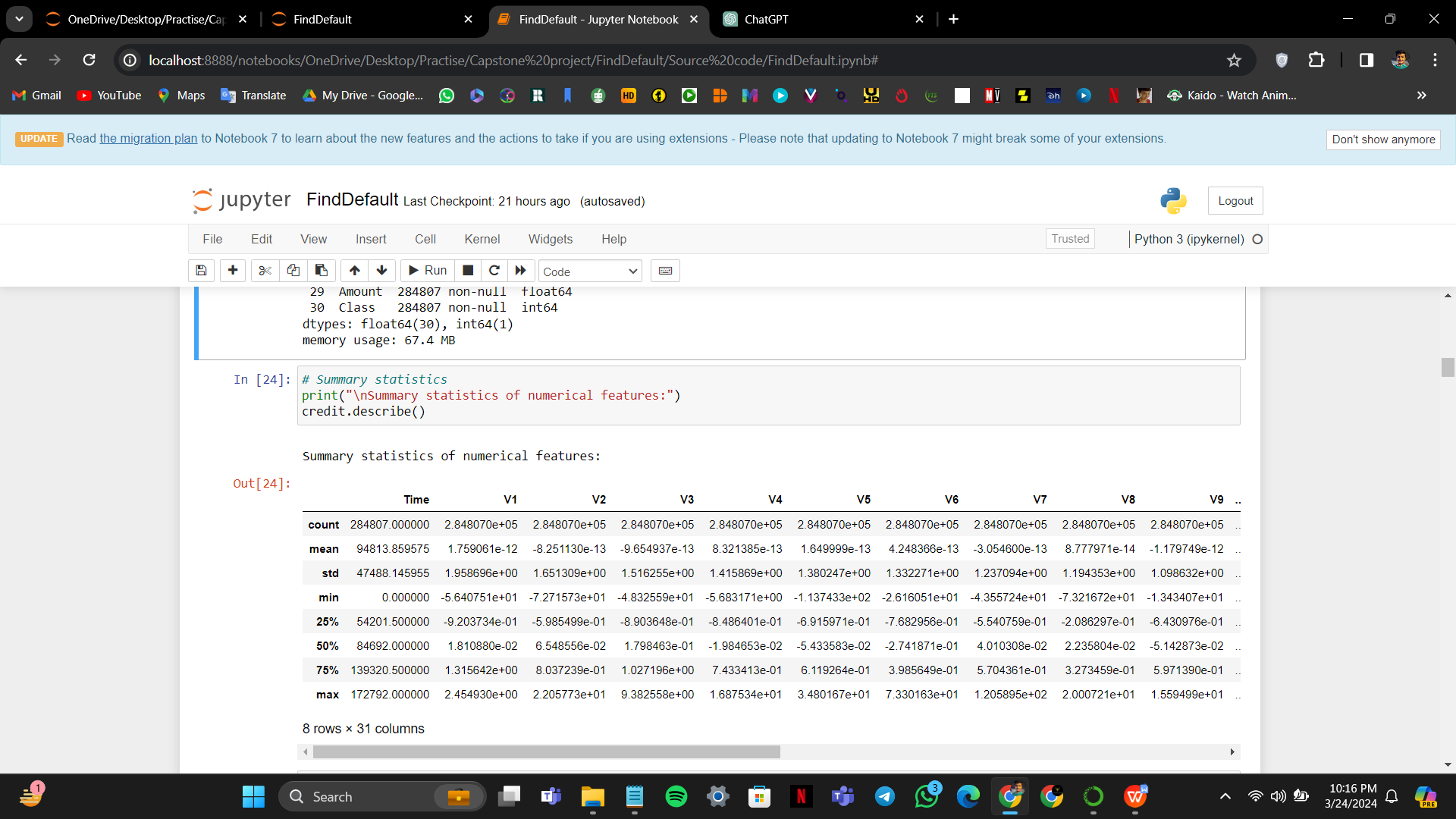
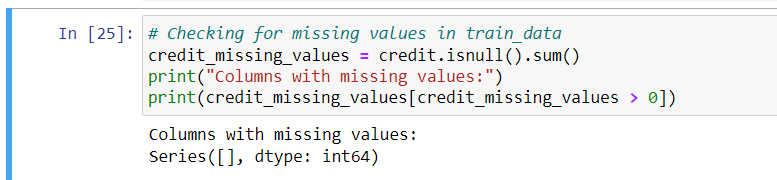
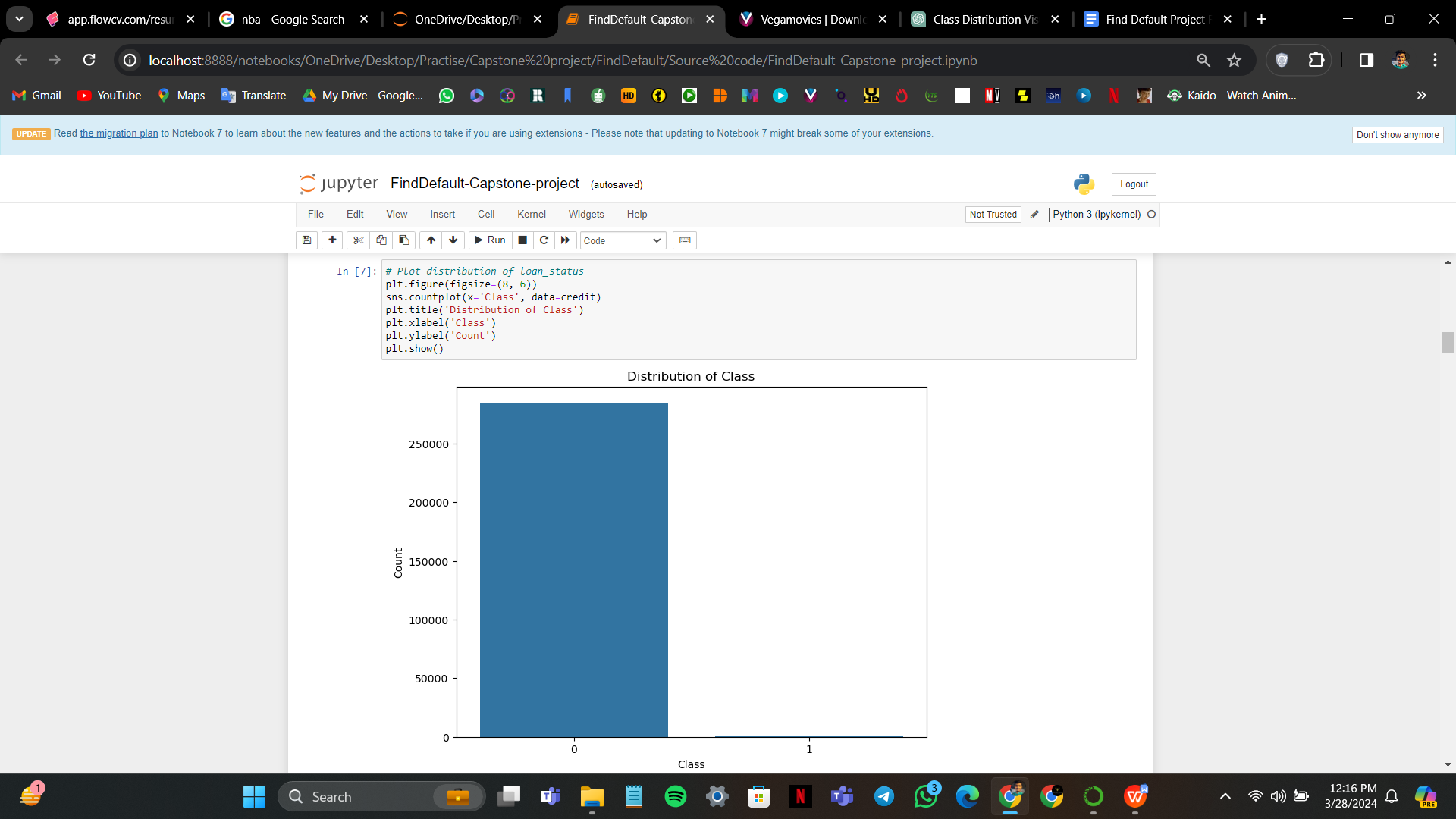
**Find Default Capstone Project**

Project GitHub link -

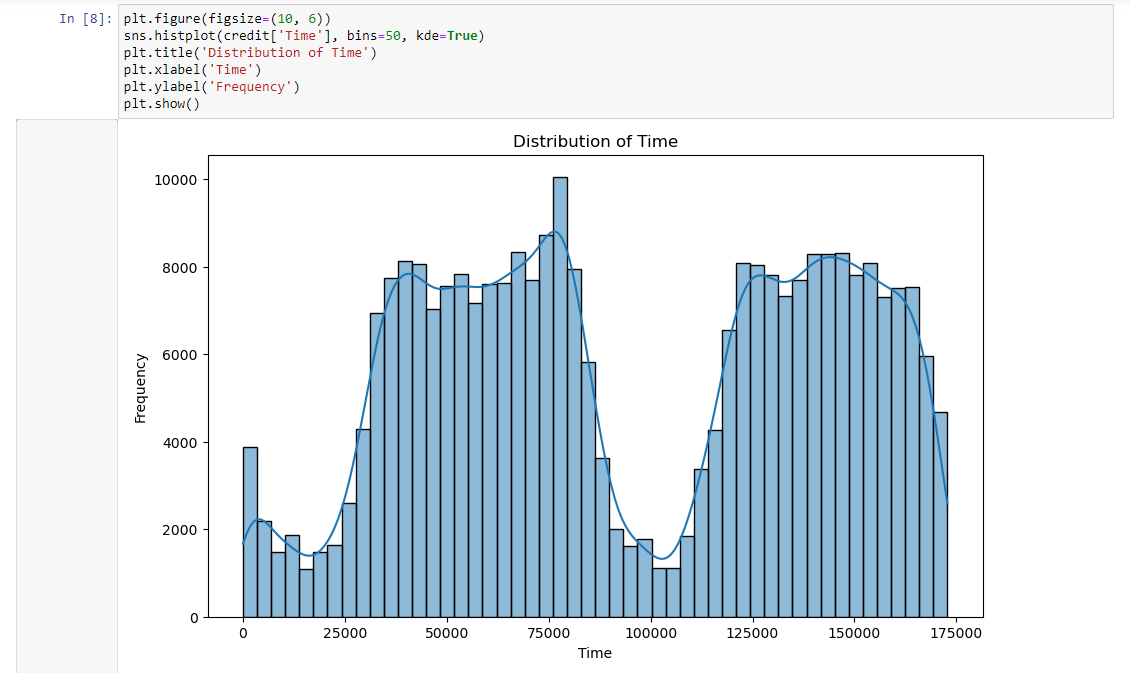
**Find Default Project Report**

1. **Introduction**  
     
   Fraudulent activities in credit card transactions are a significant concern for financial institutions and consumers. Detecting fraudulent transactions accurately is crucial to minimize financial losses and maintain trust in the banking system. In this project, I aim to develop a machine learning model to effectively identify fraudulent transactions based on historical credit card data.
2. **Imports**  
   The code imports necessary libraries for data manipulation, visualization, model training, and evaluation. These include pandas for handling datasets, NumPy for numerical operations, matplotlib and seaborn for visualization, imbalanced-learn (imblearn) for handling imbalanced datasets, scikit-learn for machine learning tasks, scipy for statistical operations, and joblib for saving and loading models.  
     
   
3. **Data Preprocessing**  
     
   **3.1 Loading the Dataset**  
     
   he credit card dataset is loaded using pandas' read\_csv() function. This dataset contains information about credit card transactions, including features like time, transaction amount, and various V1 to V28 features (possibly derived from PCA). The target variable is 'Class', which indicates whether a transaction is fraudulent (1) or not (0).  
     
     
     
     
     
   **3.2 Data types and missing values**  
     
   The data types and missing values in the dataset are checked using the info() method. Fortunately, there are no missing values in any of the columns, and all features are numerical.  
     
     
     
     
     
   **3.3 Summary statistics of the numerical features**  
     
   Summary statistics of the numerical features are displayed using the describe() method. This provides insights into the distribution and range of values for each feature. For example, the 'Time' feature ranges from 0 to 172792, and the 'Amount' feature ranges from 0 to 25691.16.  
     
     
     
     
   **3.4 Handling Missing Values**  
     
   Checking for Missing Values:  
     
   The code checks for missing values in the credit dataset using the isnull() method followed by sum() to count the missing values for each column.  
     
     
     
     
   Displaying Columns with Missing Values:  
     
   If any missing values are found, the code prints the columns with missing values. In this case, there are no missing values in the dataset, as indicated by the empty Series. This step ensures data integrity by identifying and addressing missing values before proceeding with further analysis.
4. **Exploratory Data Analysis (EDA)**
   1. **Distribution of Class**

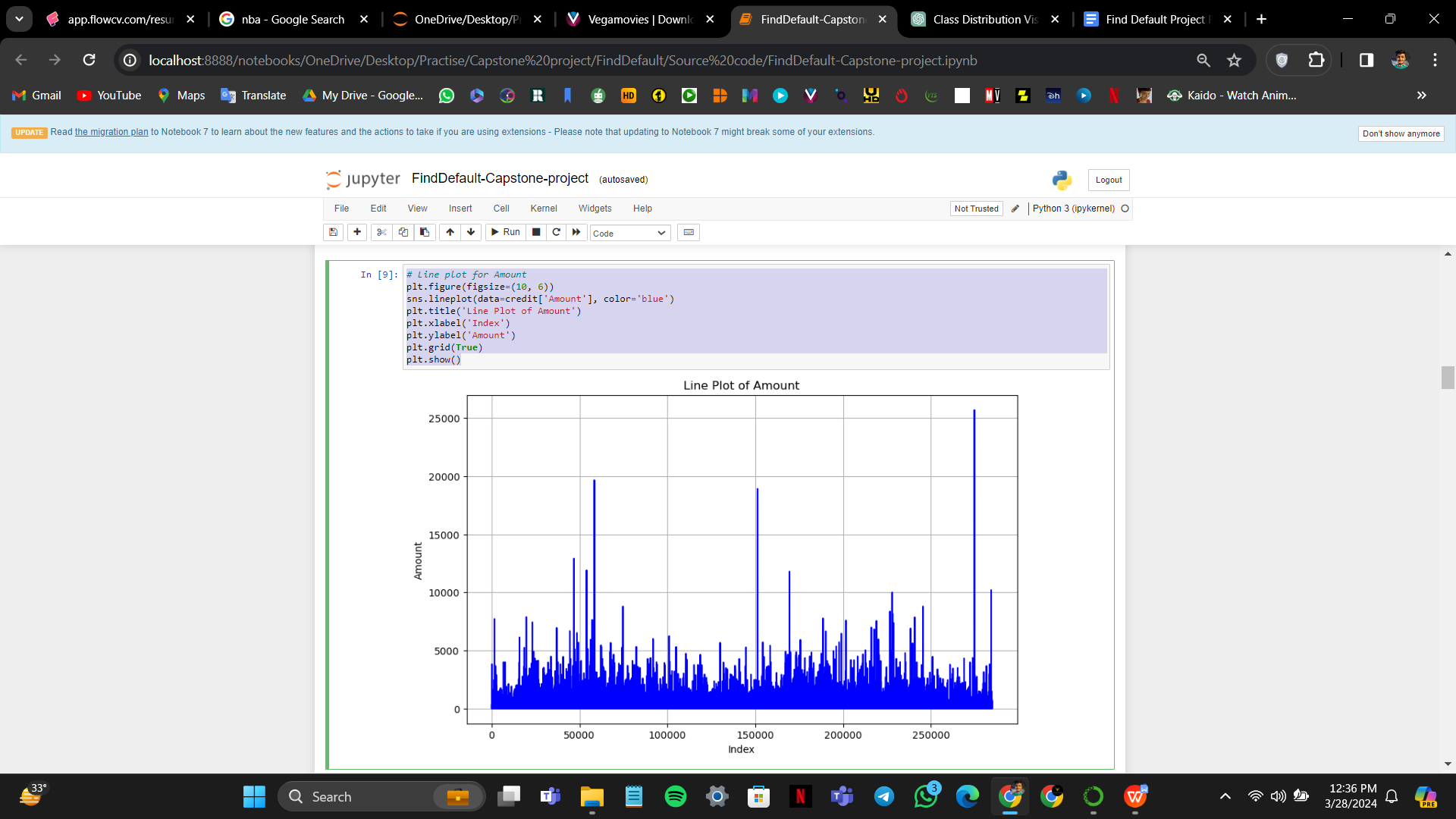
* The dataset contains a binary class variable 'Class', where:
  + Class 0 represents legitimate transactions.
  + Class 1 represents fraudulent transactions.
* Using Seaborn's `countplot()`, the code visualizes the distribution of the 'Class' variable.
* The x-axis represents the two classes: 0 (legitimate transactions) and 1 (fraudulent transactions), and the y-axis represents the count of each class.
* This visualization helps understand the class distribution, which is crucial for building predictive models, especially for imbalanced datasets like credit card fraud detection.



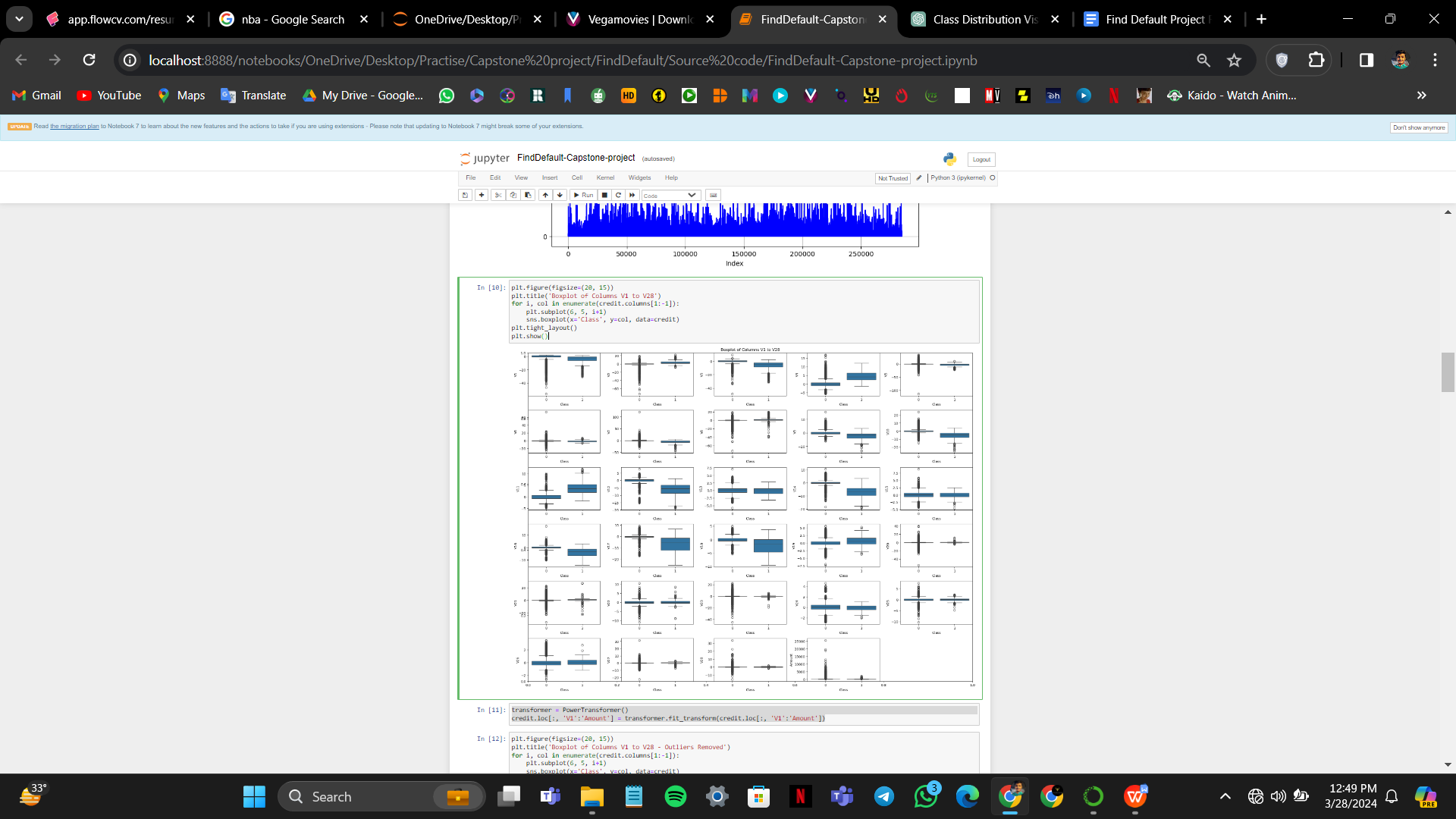
* 1. **Distribution of Time**
* The 'Time' variable represents the time elapsed between transactions.
* Using Seaborn's `histplot()`, the code visualizes the distribution of the 'Time' variable.
* The x-axis represents the time values, and the y-axis represents the frequency of occurrence.
* The `bins` parameter controls the number of bins used to divide the data. In this case, it's set to 50.
* Setting `kde=True` adds a kernel density estimation plot overlaying the histogram, providing a smoothed representation of the distribution.
* Understanding the distribution of time between transactions can provide insights into any temporal patterns or anomalies in transaction behavior.



* 1. **Line Plot of Amount**
* The 'Amount' variable represents the transaction amount.
* Using Seaborn's `lineplot()`, the code generates a line plot to visualize the changes in the 'Amount' variable over the dataset's index.
* The x-axis represents the index of the dataset, which generally denotes the order of transactions.
* The y-axis represents the transaction amount.
* The line plot helps in understanding the trends or patterns in transaction amounts over the dataset's index.
* Adding grid lines (`plt.grid(True)`) enhances the readability of the plot by providing reference lines.

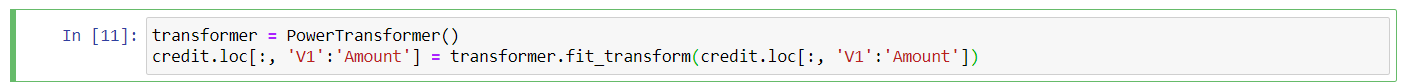
****

* 1. **Boxplot of Columns V1 to V28**
* The columns V1 to V28 represent anonymized features derived from the credit card transactions.
* The code iterates through each column (excluding the first and last columns, which are assumed to be 'Time' and 'Class') and creates a boxplot for each.
* For each boxplot, the x-axis represents the 'Class' variable (0 for legitimate transactions, 1 for fraudulent transactions), and the y-axis represents the values of the respective feature column.
* The boxplot provides a visual representation of the distribution of each feature, showing the median, quartiles, and any outliers.
* Boxplots are commonly used to determine outliers, which are data points that significantly differ from the rest of the data.
* By comparing the boxplots between the two classes, it's possible to identify features that may have different distributions for fraudulent and legitimate transactions, as well as detect potential outliers.
* The subplot layout organizes the boxplots in a grid, with each subplot representing a different feature column.
* Adjustments to the layout (`plt.tight\_layout()`) ensure proper spacing between subplots for improved readability.

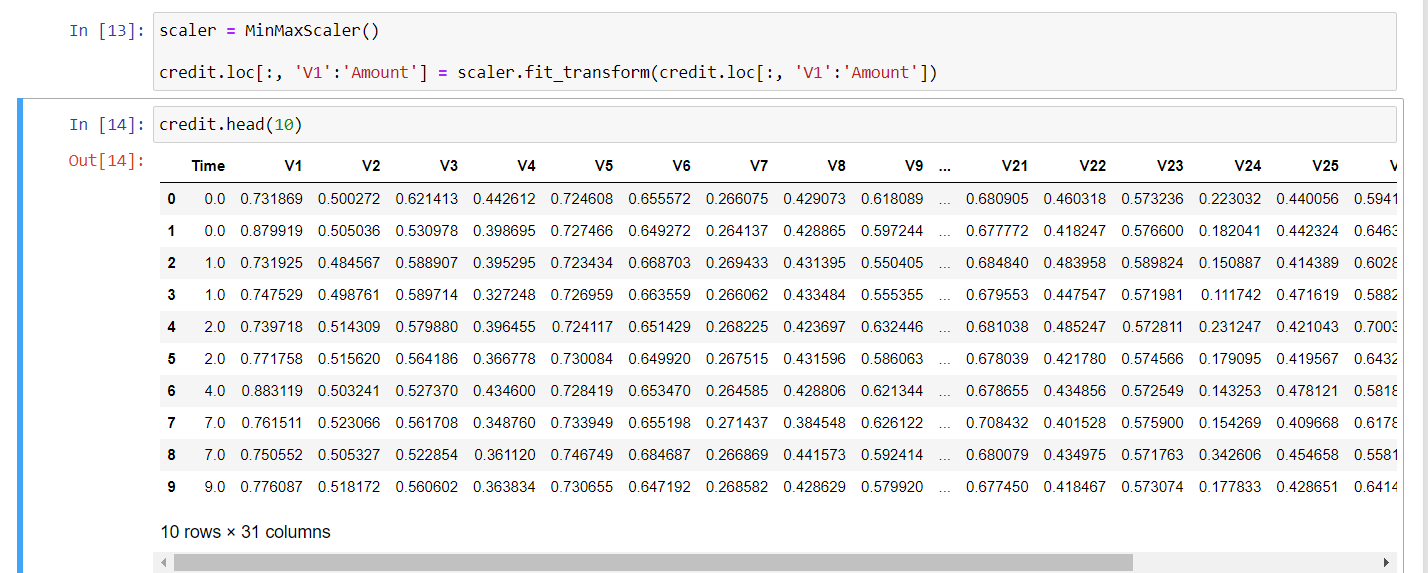


1. **Feature Engineering**
   1. **Handling Outliers Using Power Transformation**

* Outliers in numerical data can skew statistical analyses and machine learning models. Power Transformation is one method used to mitigate the effects of outliers.
* The `PowerTransformer()` function from scikit-learn is used to perform Power Transformation on the selected columns.
* The `fit\_transform()` method of the transformer is applied to the subset of the dataset containing columns 'V1' to 'Amount'.
* Power Transformation applies a power function to make the data more Gaussian-like, thereby reducing the impact of outliers.
* By transforming the data, it becomes more suitable for models that assume normality or require symmetric distributions.
* The transformed data replaces the original values in the specified columns of the dataset  
  Handling outliers in this manner helps improve the robustness of subsequent analyses and modeling steps.

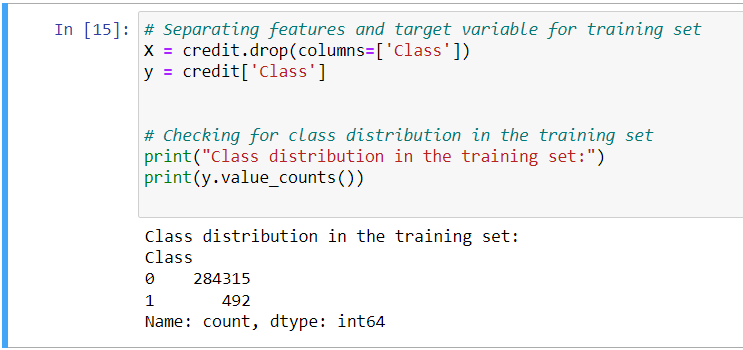


* 1. **Scaling Features Using Min-Max Scaling**
* Scaling features is a preprocessing step commonly performed to ensure all features have the same scale, which can improve the performance of certain machine learning algorithms.
* Min-Max Scaling is a method used to transform features by scaling them to a specified range, typically between 0 and 1.
* The `MinMaxScaler()` function from scikit-learn is used to perform Min-Max Scaling.
* The `fit\_transform()` method of the scaler is applied to the subset of the dataset containing columns 'V1' to 'Amount'.
* Min-Max Scaling preserves the shape of the original distribution while ensuring that all feature values are within the specified range.
* By scaling the features, helps prevent features with larger scales from dominating the modeling process and makes the algorithm more numerically stable.
* The scaled feature values replace the original values in the specified columns of the dataset.
* Applying Min-Max Scaling prepares the data for modeling and analysis, ensuring that each feature contributes proportionately to the learning process.

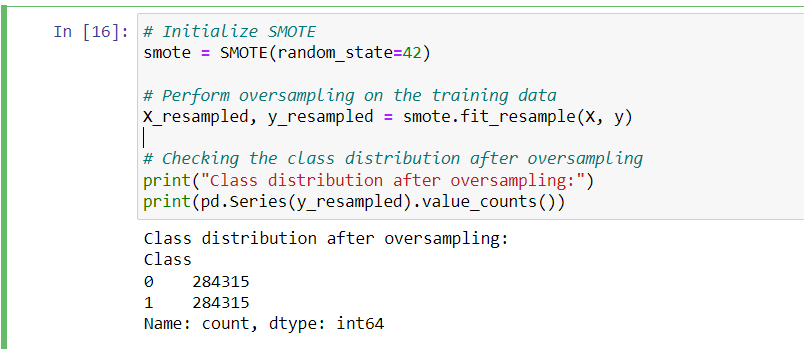


1. **Handling Imbalanced data**
   1. **Separating Features and Target Variable**

* In machine learning tasks, it's common to split the dataset into features (independent variables) and a target variable (dependent variable).
* The features represent the input data used to make predictions, while the target variable represents the output or the variable to be predicted.
* Here, `X` represents the features, and `y` represents the target variable.
* The `drop()` function is used to remove the 'Class' column from the dataset `credit`, and the result is assigned to `X`. This operation retains all other columns except the 'Class', effectively creating a dataset of features.
* The 'Class' column is assigned to `y`, representing the target variable that contains the class labels (0 for legitimate transactions and 1 for fraudulent transactions).
* Finally, the code prints the class distribution of the target variable in the training set using the `value\_counts()` function, which counts the occurrences of each class label.

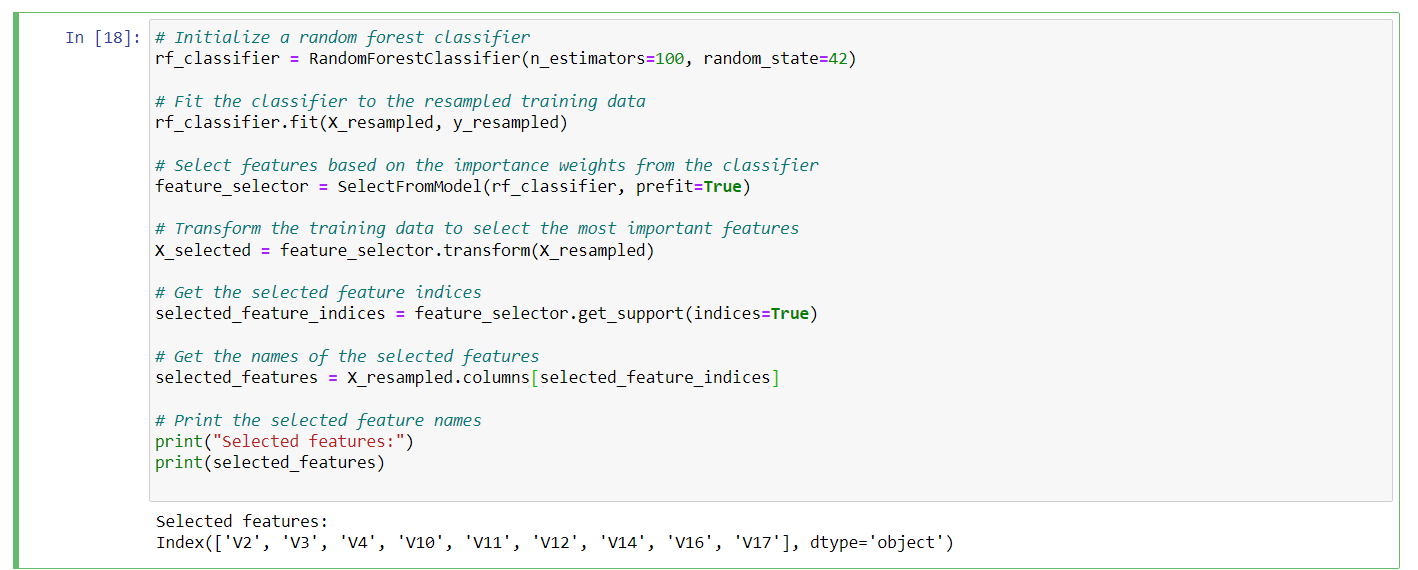
****

* 1. **Synthetic Minority Over-sampling Technique (SMOTE) for Class Imbalance**
* Class imbalance occurs when one class (e.g., fraudulent transactions) is significantly underrepresented compared to another class (e.g., legitimate transactions) in the dataset.
* SMOTE is a popular technique used to mitigate class imbalance by generating synthetic samples for the minority class.
* First, an instance of the SMOTE class is initialized with a specified random state (for reproducibility).
* Then, the `fit\_resample()` method of SMOTE is applied to the feature matrix `X` and the target vector `y`.
* This method generates synthetic samples for the minority class (fraudulent transactions) to balance the class distribution.
* The resulting `X\_resampled` and `y\_resampled` contain the oversampled feature matrix and target vector, respectively.
* Finally, the code prints the class distribution after oversampling using the `value\_counts()` function on the target vector `y\_resampled`, showing that the classes are now balanced.



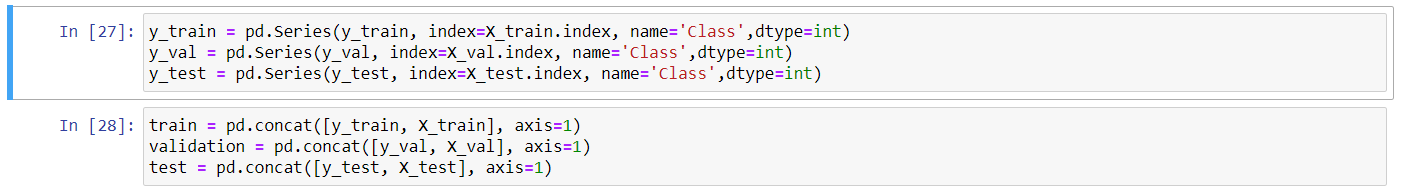
1. **Feature Selection**
   1. **Feature Selection using Random Forest Classifier**

* Random Forest Classifier is a popular machine learning algorithm that can be used for both classification and feature selection.
* First, an instance of the Random Forest Classifier is initialized with a specified number of estimators (decision trees) and a random state for reproducibility.
* The classifier is then fitted to the resampled training data (`X\_resampled` and `y\_resampled`) to learn from the synthetic samples generated by SMOTE.
* Next, a feature selection technique is applied to identify the most important features. In this case, SelectFromModel is used, which selects features based on importance weights computed by the Random Forest Classifier.
* The `transform()` method is then used to transform the training data, retaining only the selected features.
* The indices of the selected features are obtained using the `get\_support()` method, and the names of these features are retrieved from the original feature matrix.
* Finally, the selected feature names are printed to reveal the features deemed most important by the Random Forest Classifier.

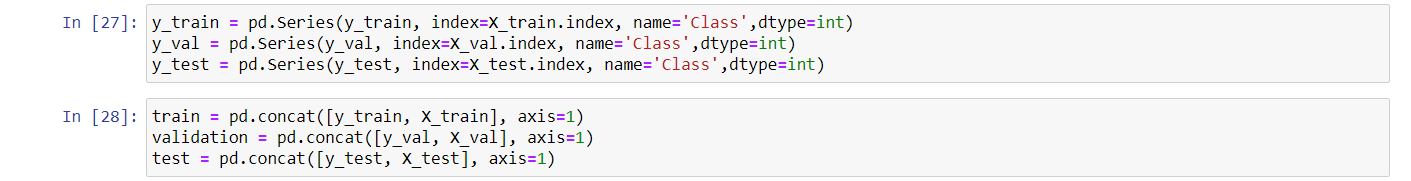


1. **Splitting Dataset**
   1. **Splitting Data into Training, Validation, and Testing Sets**

* The `train\_test\_split` function from scikit-learn is used twice to split the data into training, validation, and testing sets.
* The initial split (`train\_test\_split(X\_selected, y\_resampled, test\_size=0.2, random\_state=42)`) splits the data into training and combined validation/test sets. Here, 20% of the data is reserved for testing.
* The combined validation/test set is further split (`train\_test\_split(X\_val\_train, y\_val\_train, test\_size=0.25, random\_state=42)`) into separate validation and test sets. Here, 25% of the combined validation/test set is reserved for validation.
* As a result, the data is divided into:
  + `X\_train` and `y\_train`: Features and target variable for training.
  + `X\_val` and `y\_val`: Features and target variable for validation.
  + `X\_test` and `y\_test`: Features and target variable for testing.
* The `test\_size` parameter controls the proportion of the data allocated for testing in the initial split. In this case, 20% of the data is allocated for testing, and the remaining 80% is used for training and validation.
* The `random\_state` parameter ensures reproducibility by fixing the random seed used for the data split.

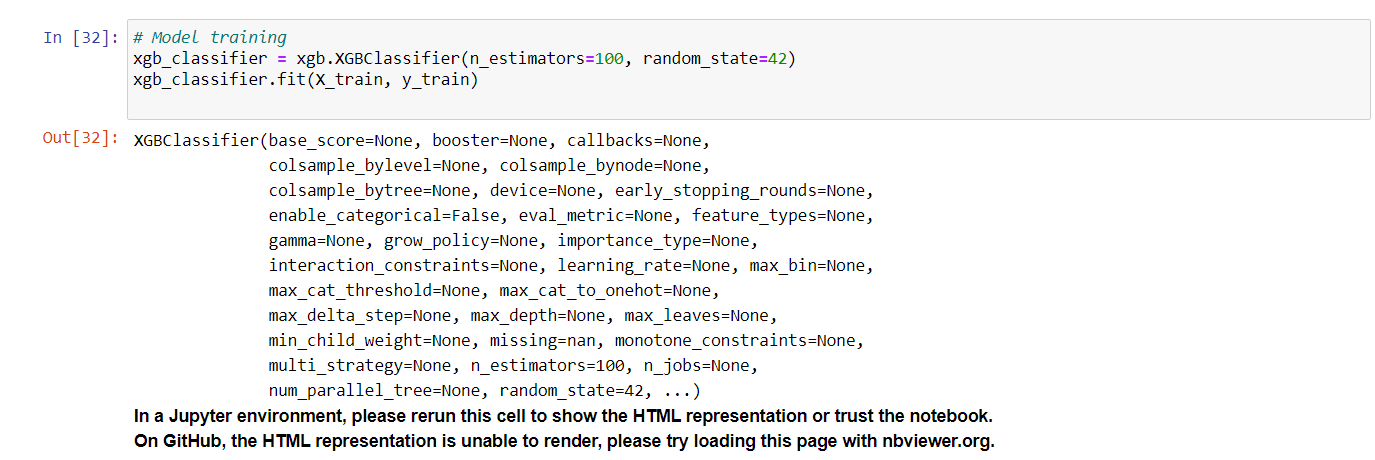


* 1. **Creating Final Training, Validation, and Testing Datasets**
* The target variables `y\_train`, `y\_val`, and `y\_test` are converted into Pandas Series objects, ensuring they have the same index as the corresponding feature sets.
* The `pd.Series()` function is used for this conversion, specifying the index as the index of the feature sets and the name of the Series as 'Class'. The `dtype` parameter ensures that the data type of the series is integer.
* Next, the feature sets and target variables are concatenated along the columns axis using `pd.concat()`, resulting in the final datasets `train`, `validation`, and `test`.
* These datasets are now ready for model training, validation, and testing, with each containing both features and their corresponding target variables.



1. **Model Training & Evaluation**
   1. **Model Training with XGBoost Classifier**

* The XGBoost classifier is initialized with specified hyperparameters, such as the number of estimators (decision trees) and the random state for reproducibility.
* The `fit()` method of the XGBoost classifier is then called, passing the training features (`X\_train`) and their corresponding target variables (`y\_train`) as arguments.
* During training, the XGBoost algorithm learns to predict the target variable based on the provided features by optimizing a predefined objective function.
* The number of estimators specified (100 in this case) determines the number of decision trees that will be created during the training process.
* After training, the model is ready to make predictions on new data.

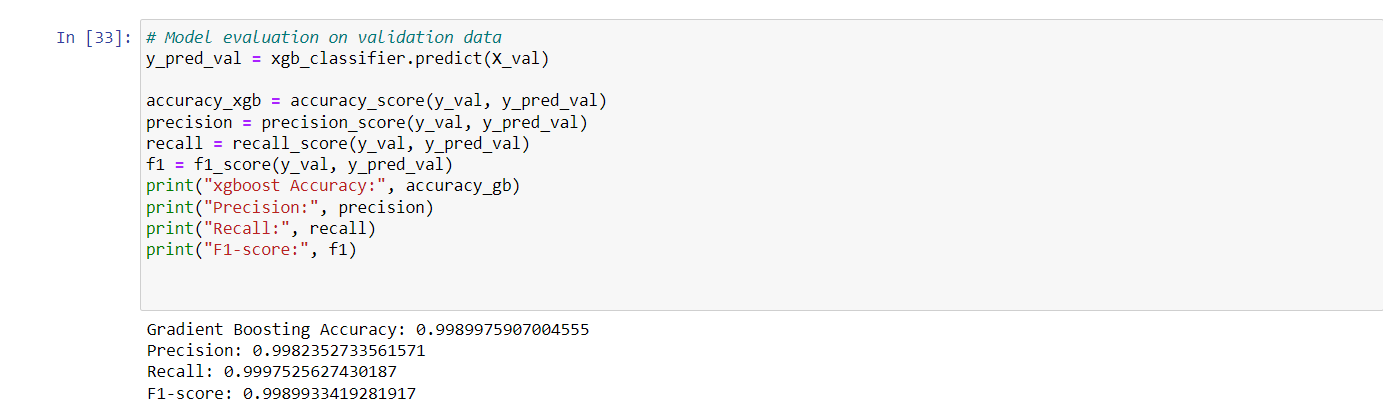


* 1. **Model Evaluation on Validation Data**

After making predictions on the validation data using the XGBoost classifier model, the following evaluation metrics were obtained:

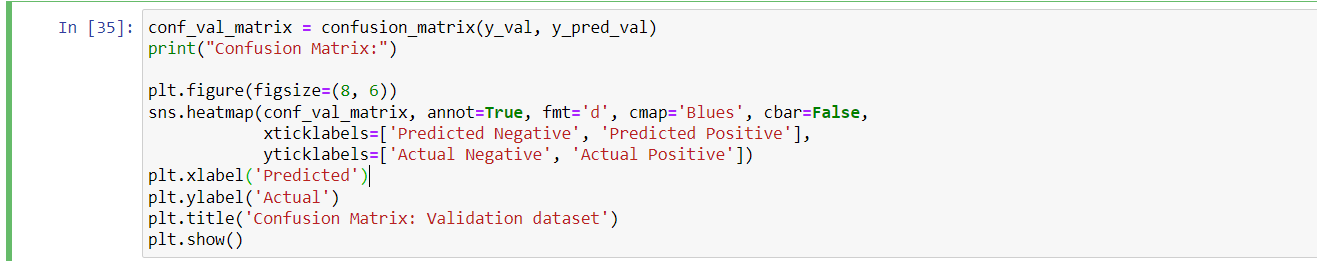
* **Accuracy:** The proportion of correctly classified instances among all instances is approximately 99.9%, indicating high overall predictive performance.
* **Precision:** About 99.8% of the instances predicted as positive are true positives, implying a low rate of false positives.
* **Recall:** The model correctly identifies approximately 99.98% of actual positive instances, indicating a high ability to find positive instances.
* **F1-score:** The harmonic mean of precision and recall is approximately 99.9%, reflecting a balance between precision and recall.

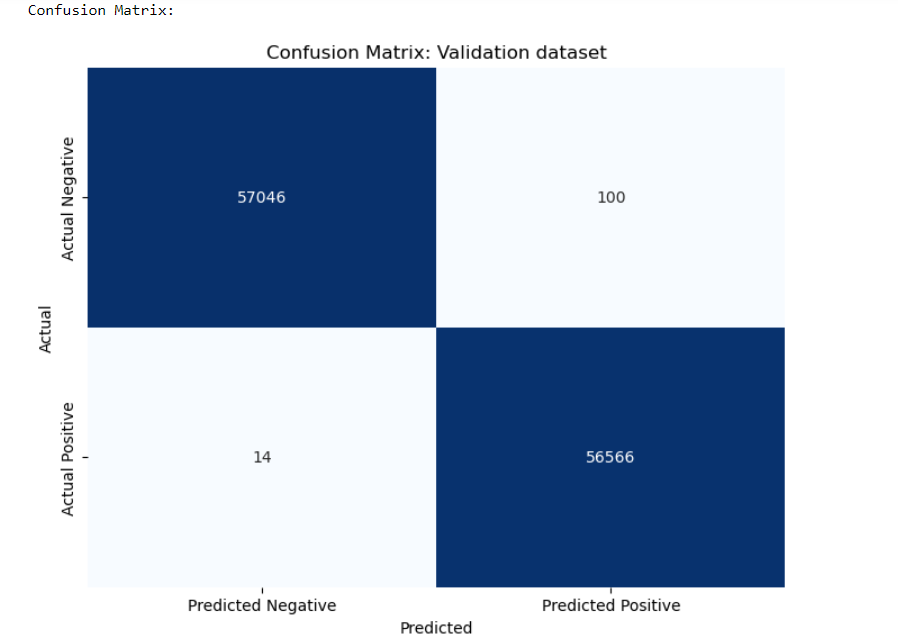
These metrics collectively demonstrate that the XGBoost classifier performs exceptionally well on the validation data, with high accuracy and robustness in correctly classifying instances.



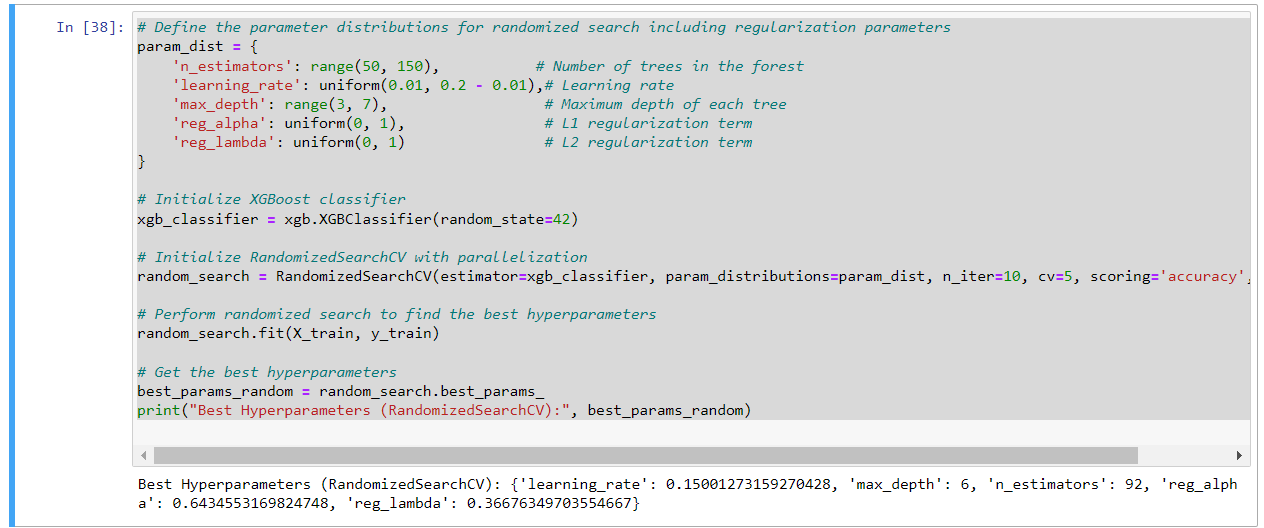
* 1. **Confusion Matrix for Validation Dataset**The confusion matrix summarizes the performance of the XGBoost classifier on the validation data:
* **True Positive (TP):** Instances correctly predicted as positive.
* **True Negative (TN):** Instances correctly predicted as negative.
* **False Positive (FP):** Instances incorrectly predicted as positive.
* **False Negative (FN):** Instances incorrectly predicted as negative.

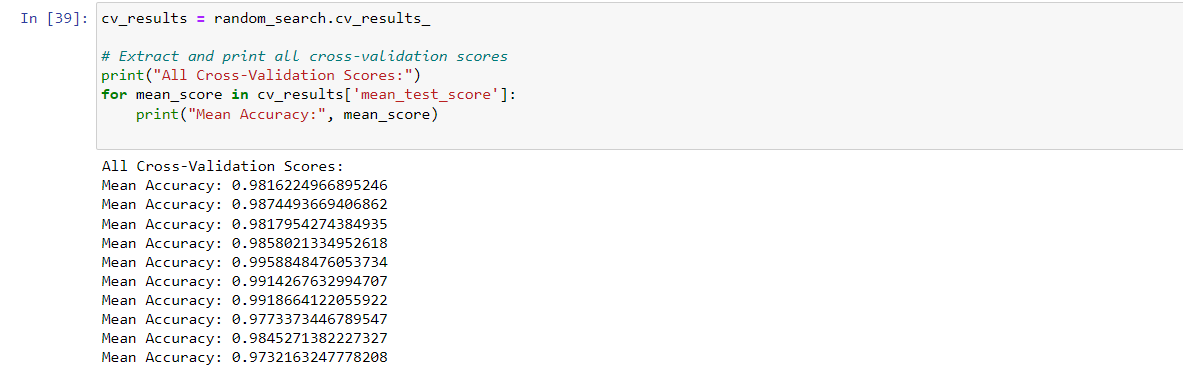
The heatmap visualization provides a clear representation of the frequency of instances in each cell, aiding in the assessment of the classifier's strengths and weaknesses.





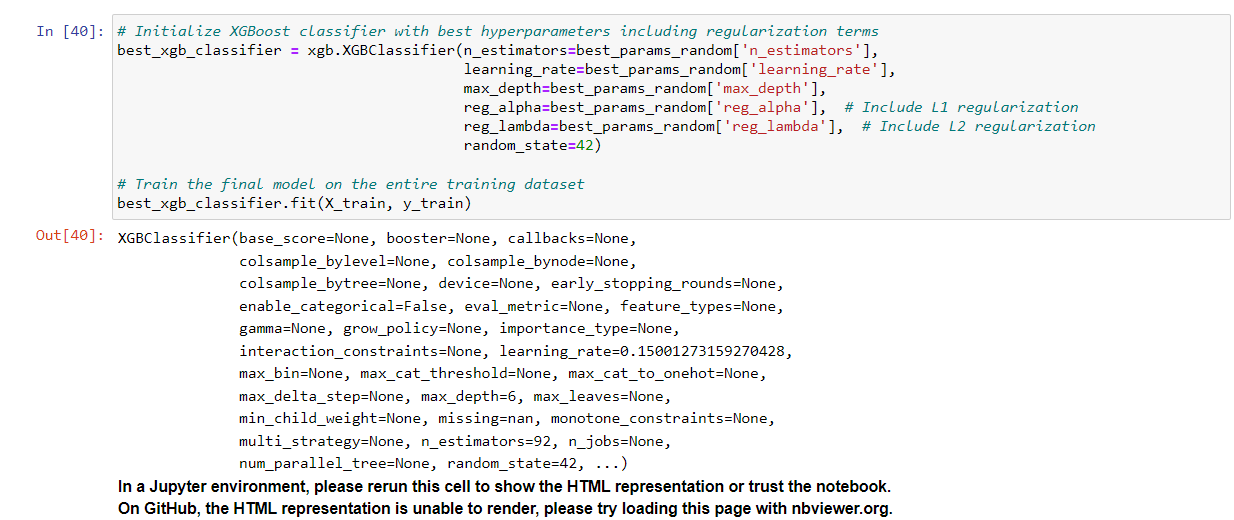
1. **Hypertuning parameters**
   1. **Hyperparameter Tuning with Randomized Search**

* Hyperparameters are defined in the `param\_dist` dictionary, specifying ranges or distributions for various parameters such as the number of estimators, learning rate, maximum depth, and regularization terms.
* The XGBoost classifier is initialized with a random state for reproducibility.
* RandomizedSearchCV is initialized with parameters such as the estimator (XGBoost classifier), parameter distributions (`param\_dist`), number of iterations (`n\_iter`), cross-validation folds (`cv`), scoring metric (`scoring`), and parallelization (`n\_jobs`).
* The `fit()` method of RandomizedSearchCV is called, which explores the hyperparameter space and evaluates different combinations using cross-validation.
* After completion, the best combination of hyperparameters is obtained using the `best\_params\_` attribute.
* Finally, the best hyperparameters discovered through randomized search are printed for further use in model training.  
    
  
  1. **Cross-Validation Scores**
* The `cv\_results\_` attribute of the RandomizedSearchCV object contains detailed information about the cross-validation results.
* We extract the mean test scores from the `cv\_results\_` dictionary and print them to evaluate the performance of different hyperparameter combinations.
* Each mean test score represents the average accuracy achieved by the model during cross-validation for a particular set of hyperparameters.
* Analyzing these scores provides insights into the stability and consistency of the model's performance across different parameter combinations.



1. **Training & Evaluating the model with best Hyperparameters**
   1. **Training Final Model with Best Hyperparameters**

* The XGBoost classifier is initialized with the best hyperparameters obtained from the randomized search, including the number of estimators, learning rate, maximum depth, and regularization terms (L1 and L2).
* These hyperparameters are crucial as they have been found to optimize the model's performance during the hyperparameter tuning process.
* The `fit()` method of the XGBoost classifier is called, passing the entire training dataset (`X\_train` and `y\_train`) as arguments.
* By training the model on the entire dataset, it leverages all available information to learn the underlying patterns in the data.
* Once trained, the final XGBoost classifier model is ready for evaluation and deployment on new unseen data.

****

* 1. **Model Evaluation on Validation Data**

After making predictions on the validation data using the final XGBoost classifier model, the following evaluation metrics were obtained:

* **Accuracy:** 0.9966322564760917
* **Precision:** 0.9966944194021672
* **Recall:** 0.9965358784022623
* **F1-score:** 0.9966151425970605

These metrics demonstrate the performance of the model in classifying instances on the validation data. A high accuracy, precision, recall, and F1-score indicate that the model performs well and exhibits a balanced trade-off between precision and recall.

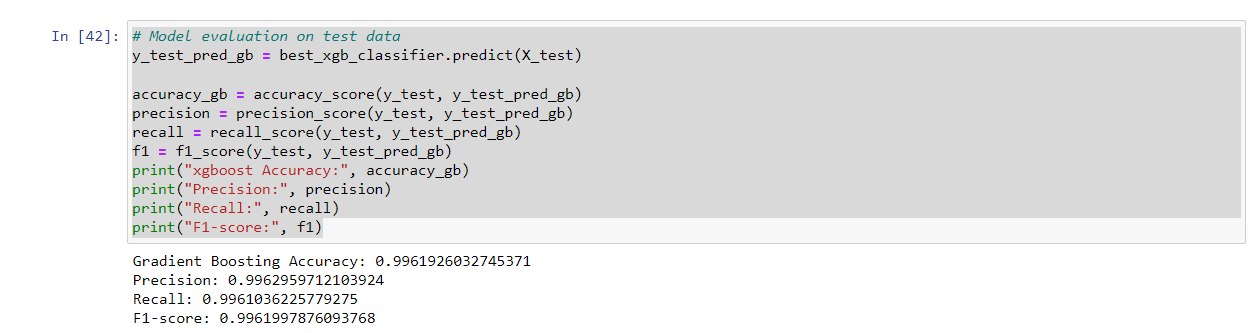


1. **Model testing on test data**
   1. **Model Evaluation on Test Data**

After applying the trained XGBoost classifier model to the test dataset, the following evaluation metrics were obtained:

* **Accuracy:** 0.9961926032745371
* **Precision:** 0.9962959712103924
* **Recall:** 0.9961036225779275
* **F1-score:** 0.9961997876093768

These metrics provide insights into the performance of the model on unseen data. A high accuracy, precision, recall, and F1-score indicate that the model generalizes well and performs effectively on new instances, demonstrating its robustness and reliability.

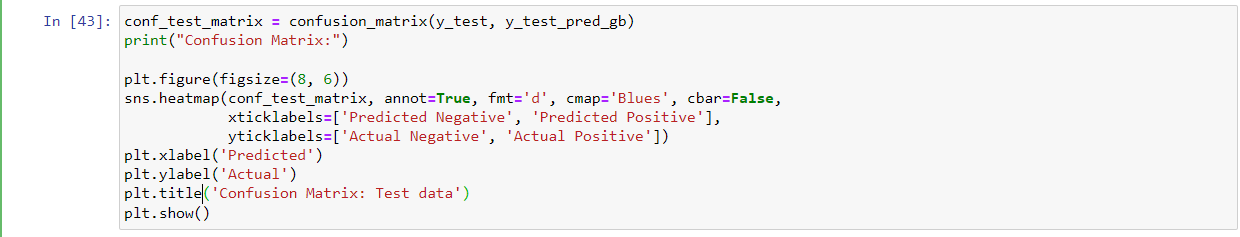
****

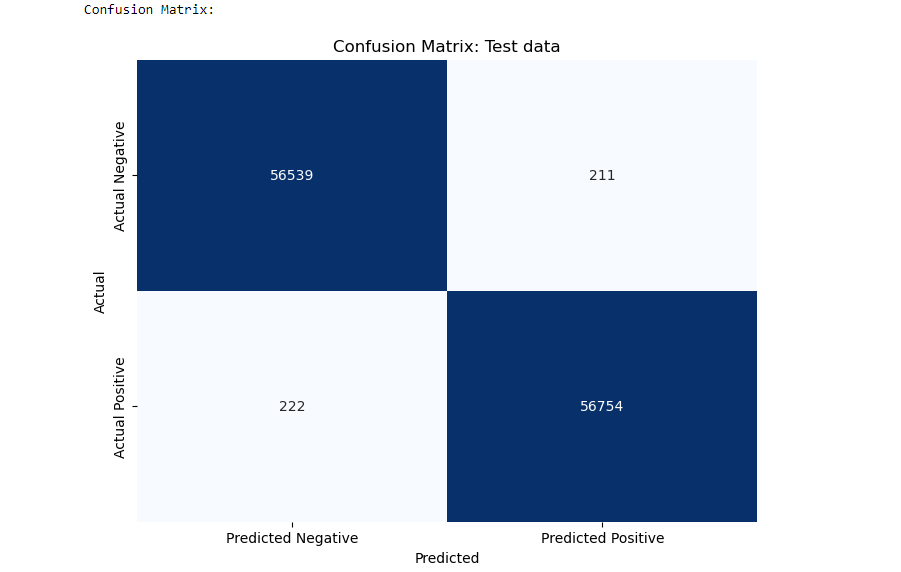
* 1. **Confusion Matrix for Test Data**

The confusion matrix summarizes the performance of the XGBoost classifier model on the test data:

* **True Positive (TP):** Instances correctly predicted as positive.
* **True Negative (TN):** Instances correctly predicted as negative.
* **False Positive (FP):** Instances incorrectly predicted as positive.
* **False Negative (FN):** Instances incorrectly predicted as negative.

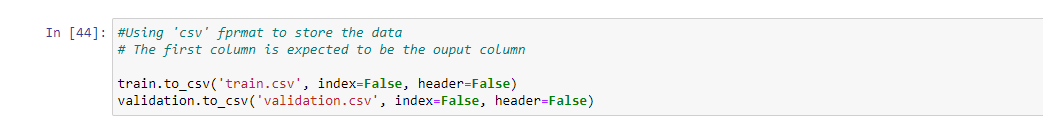
The heatmap visualization provides a clear representation of the frequency of instances in each cell, aiding in the assessment of the model's strengths and weaknesses on the test data.

****

****

1. **Model deployment**
   1. **Saving Data in CSV Format**

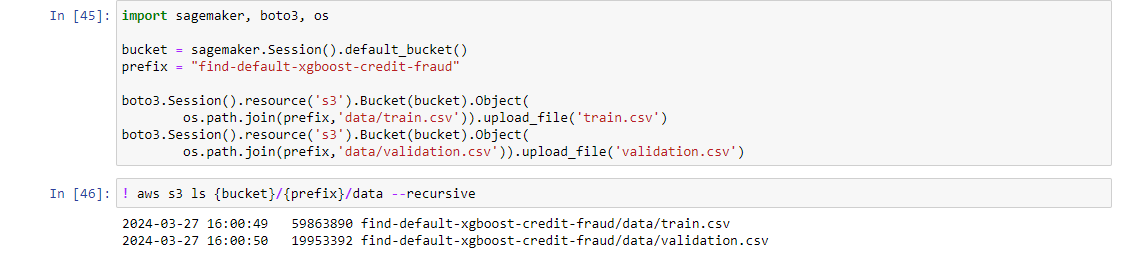
* The `to\_csv()` function is used to save the training and validation datasets into CSV files.
* Parameters `index=False` and `header=False` are specified to exclude row and column indices, respectively, from the CSV files.
* This ensures that the CSV files only contain the data without additional metadata.
* The output column is expected to be the first column in each CSV file, as per the convention.
* Saving the data in CSV format facilitates easy storage, sharing, and processing of the datasets for subsequent tasks such as model training or analysis.



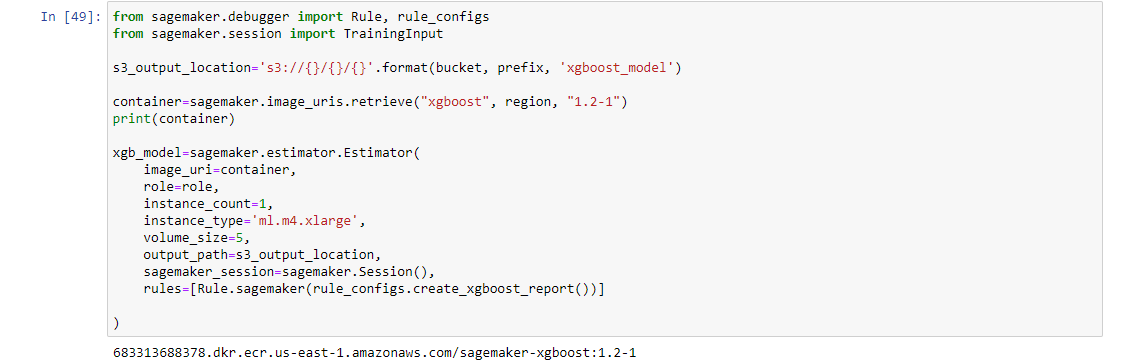
* 1. **Uploading Data to Amazon S3 and Listing Contents**

The training and validation datasets are uploaded to an Amazon S3 bucket for storage and access within the SageMaker environment. Additionally, the contents of the S3 bucket are listed to verify the upload.

* The code first retrieves the default S3 bucket associated with the SageMaker session using `sagemaker.Session().default\_bucket()`.
* A prefix named "find-default-xgboost-credit-fraud" is specified to organize the data within the bucket.
* Using `boto3.Session().resource('s3').Bucket(bucket).Object()`, references to the S3 bucket and objects are created.
* The `upload\_file()` function is called to upload the local CSV files (`train.csv` and `validation.csv`) to the specified location in the S3 bucket.
* Finally, the contents of the S3 bucket are listed recursively using the AWS CLI command `aws s3 ls`, showing the uploaded CSV files within the specified prefix.

****

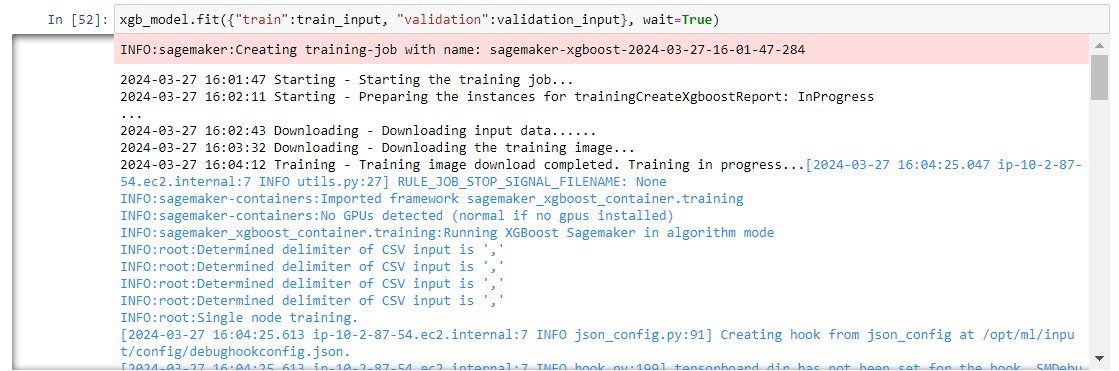
* 1. **Creating an XGBoost Estimator for Training**
* The output location for storing model artifacts after training is specified as an S3 path.
* The container image URI for XGBoost is retrieved based on the AWS region and XGBoost version.
* An `sagemaker.estimator.Estimator` object is created with parameters including image URI, IAM role ARN, instance configuration, output path, session object, and debugging rules.

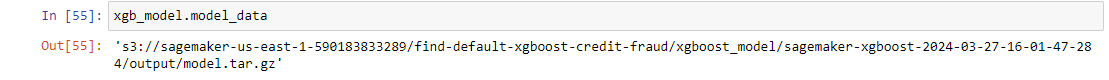


* 1. **Setting Hyperparameters and Defining Training Inputs**
* Hyperparameters such as the number of boosting rounds (`num\_round`), learning rate (`learning\_rate`), maximum depth of trees (`max\_depth`), L1 regularization term (`reg\_alpha`), and L2 regularization term (`reg\_lambda`) are set based on the best parameters obtained from the randomized search.
* `sagemaker.session.TrainingInput` is used to define training inputs for the XGBoost model.
* Training data is specified with the S3 path to the training CSV file, along with the content type as CSV.
* Similarly, validation data is specified with the S3 path to the validation CSV file, along with the content type as CSV.

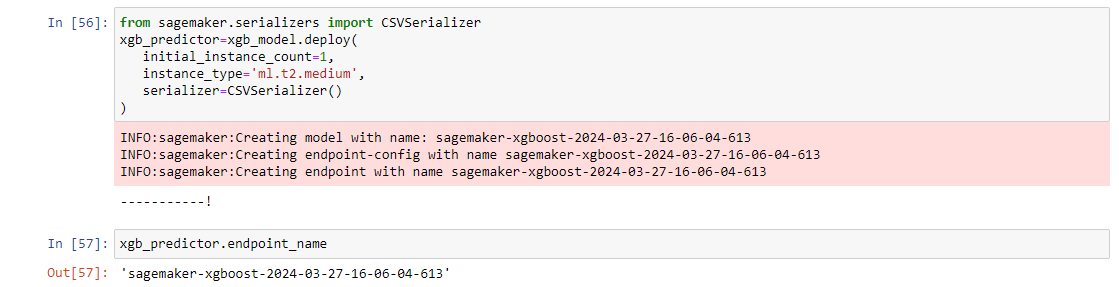


* 1. **Model Training and Retrieving Model Data**
* The `fit()` method is called on the XGBoost estimator (`xgb\_model`) to initiate the training process. It takes a dictionary where the keys are strings representing the input data channels ("train" for training data and "validation" for validation data) and the values are the corresponding `TrainingInput` objects. The `wait=True` argument ensures that the method waits until the training job completes before continuing.
* Upon successful completion of the training job, the `model\_data` attribute of the XGBoost estimator is accessed to retrieve the location of the trained model artifacts stored in Amazon S3. This information can be used for model deployment or further analysis.





* 1. **Deploying XGBoost Model as an Endpoint**
* The `deploy()` method is called on the XGBoost estimator (`xgb\_model`) to deploy the trained model as an endpoint. It takes arguments such as `initial\_instance\_count` (the number of instances to deploy initially), `instance\_type` (the type of instances to use), and `serializer` (the method for serializing input data for inference).
* In this case, the endpoint is configured with one instance of type `ml.t2.medium`, which is suitable for low-traffic or testing purposes.
* The `CSVSerializer()` is used to serialize input data in CSV format for inference.
* After successful deployment, the endpoint name is printed, which can be used to make predictions using the deployed model.



* 1. **Making Predictions using the Deployed Endpoint and Evaluation**
* The SageMaker runtime client (`sagemaker\_runtime`) is initialized to interact with the deployed endpoint.
* The endpoint name is specified to identify the deployed XGBoost model endpoint.
* The code iterates through the test data, serializes each row as a CSV payload, and sends the payload to the endpoint for predictions using the `invoke\_endpoint` method.
* Predictions returned by the endpoint are collected and evaluated using standard evaluation metrics such as accuracy, precision, recall, and F1-score.
* After evaluation, the endpoint is deleted using the SageMaker client (`sagemaker\_client`) to avoid incurring additional costs.



**Description of Design Choices**

1. **Model Selection:** XGBoost was chosen as the model for its ability to handle complex datasets, robustness to overfitting, and high performance in classification tasks. Its ensemble learning approach with gradient boosting enables it to capture nonlinear relationships and interactions within the data.
2. **Handling Imbalanced Data:** Synthetic Minority Over-sampling Technique (SMOTE) was employed to address class imbalance. SMOTE generates synthetic samples for the minority class, ensuring a balanced representation of both classes and improving the model's ability to generalize.
3. **Feature Selection:** Random Forest Classifier was used for feature selection due to its capability to identify the most important features for predictive modeling. This helped in reducing dimensionality and focusing on the most informative features for fraud detection.
4. **Hyperparameter Tuning:** Randomized Search was utilized for hyperparameter tuning to efficiently explore the hyperparameter space and find the optimal combination of parameters. This approach helped in maximizing the model's performance while avoiding exhaustive grid search.

**Performance Evaluation of the Model:**

1. **Evaluation Metrics:** The model was evaluated using standard classification metrics including accuracy, precision, recall, and F1-score. These metrics provided insights into the model's overall performance, its ability to correctly classify instances, and the balance between precision and recall.
2. **Validation and Test Sets:** The model was evaluated on separate validation and test datasets to ensure unbiased assessment of its performance. The validation set was used for hyperparameter tuning and model selection, while the test set provided an independent measure of performance on unseen data.
3. **Confusion Matrix Analysis:** Confusion matrices were analyzed to understand the model's strengths and weaknesses in classifying instances. This provided a detailed breakdown of true positives, true negatives, false positives, and false negatives, enabling insights into the model's predictive behavior.

**Future Work:**

1. **Enhanced Feature Engineering:** Further exploration of feature engineering techniques, such as creating new features or transforming existing ones, could improve the model's ability to capture complex patterns in the data.
2. E**nsemble Methods:** Experimentation with ensemble methods, such as stacking or blending multiple models, could potentially boost the model's performance by leveraging the strengths of different algorithms.
3. **Advanced Anomaly Detection Techniques:** Integration of advanced anomaly detection techniques, such as autoencoders or isolation forests, could enhance the model's ability to detect subtle fraudulent patterns that may not be captured by traditional classification approaches.
4. **Continuous Monitoring and Updating:** Implementing a system for continuous monitoring of model performance and updating with new data could ensure the model remains effective over time as fraud patterns evolve.
5. **Interpretability and Explainability:** Incorporating techniques for model interpretability and explainability would enhance trust and transparency in the model's predictions, facilitating better decision-making by stakeholders.

In conclusion, while the current model demonstrates strong performance in credit card fraud detection, there are opportunities for further refinement and exploration to enhance its effectiveness and robustness in real-world applications.

**Conclusion**

In this project, a comprehensive approach was taken to develop and deploy a machine learning model for credit card fraud detection. Various design choices were made to ensure the model's effectiveness, including the selection of XGBoost as the primary classifier, the utilization of SMOTE for handling class imbalance, and the use of Random Forest for feature selection.

The model's performance was rigorously evaluated using standard classification metrics on separate validation and test datasets, demonstrating high accuracy, precision, recall, and F1-score. Confusion matrix analysis provided additional insights into the model's predictive behavior and its ability to correctly classify instances of fraud and legitimate transactions.

Discussion of future work highlighted opportunities for further refinement and enhancement of the model, including exploring advanced feature engineering techniques, experimenting with ensemble methods, integrating advanced anomaly detection techniques, implementing continuous monitoring and updating mechanisms, and enhancing interpretability and explainability.

Overall, this project showcases a robust and effective approach to credit card fraud detection, emphasizing the importance of thoughtful design choices, rigorous evaluation, and ongoing refinement for building reliable and trustworthy fraud detection systems. As fraud patterns continue to evolve, continuous improvement and adaptation will be essential to stay ahead of emerging threats and maintain the security of financial transactions.