2024

Case Study Five

PREDICT BANKRUPTCY

LANI LEWIS

SOUTHERN METHODIST UNIVERSITY

0 - 11

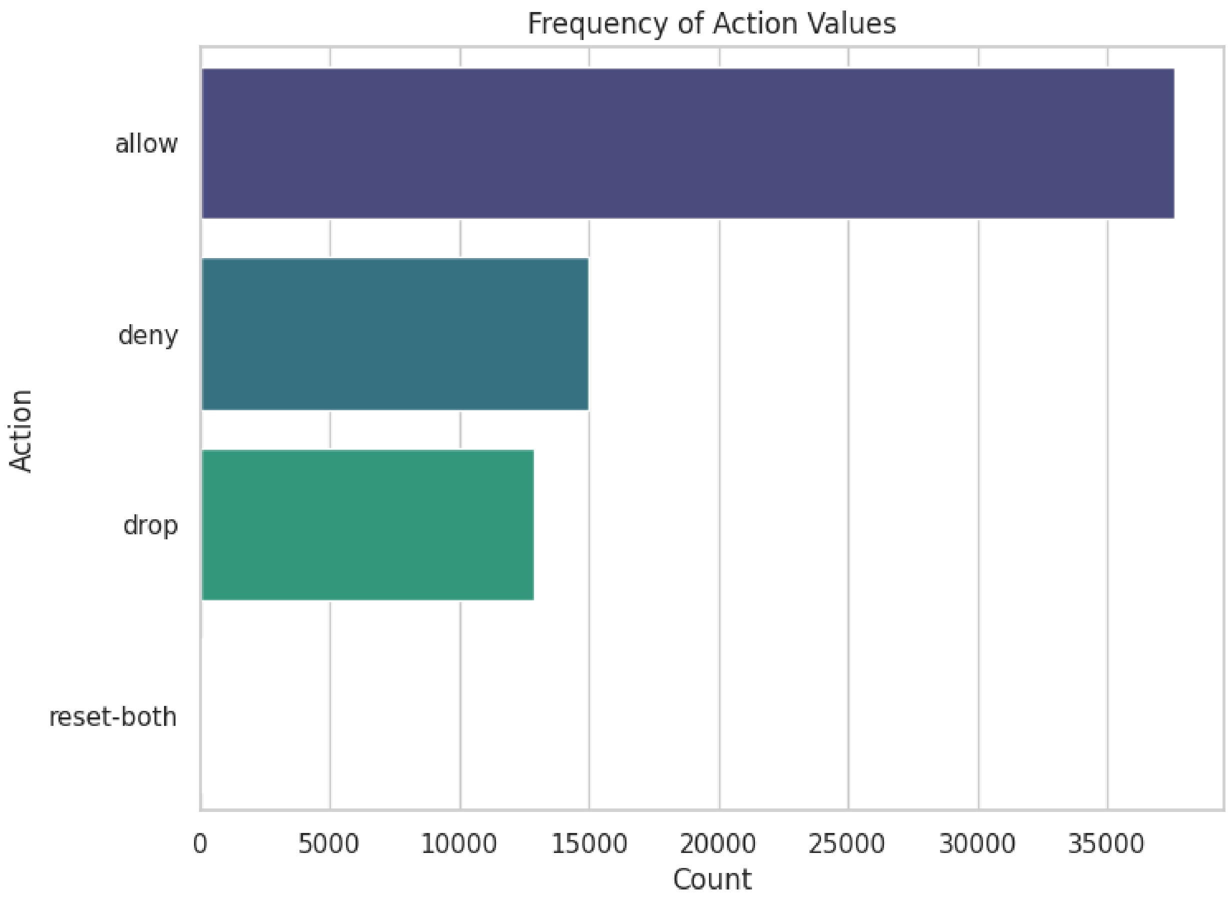
TABLE OF CONTENTS

ABSTRACT

In the realm of cybersecurity, managing firewall actions efficiently is paramount for organizations facing high volumes of network traffic. Currently, many companies expend substantial manpower and resources determining whether to accept or deny access to incoming requests—a process well suited for automation. This study proposes a robust program aimed at automating the decision-making process for firewall actions, leveraging historical data to filter, classify, and automatically respond to network requests. I seek to develop a model that prioritizes both high accuracy and operational speed. Utilizing Support Vector Machines (SVM) and Stochastic Gradient Descent (SGD) methodologies, the program will not only automate the classification of network traffic into four distinct actions but also optimize firewall management. The efficacy of my model will be demonstrated through a detailed report on its accuracy and performance, highlighting its potential to manage firewall traffic and reduce the burden on IT infrastructure.

INTRODUCTION

In the initial phase of my analysis, I examined the structure and characteristics of the dataset housed in a DataFrame of 65,532 entries across 12 columns, consuming approximately 6.0+ MB of memory. The primary focus was on understanding the distribution of my target variable, 'Action', which revealed significant class imbalances. Predominantly, the 'allow' action characterized the majority of the network traffic allowed through the firewall, followed by the 'deny' action which represented a considerable portion of the traffic being explicitly blocked. Less frequent were the 'drop' actions, which silently discard traffic without any sender notification, and the 'reset-both' actions, which abruptly terminate connections. The latter, being the least common, poses the greatest challenge for classification due to its rarity.

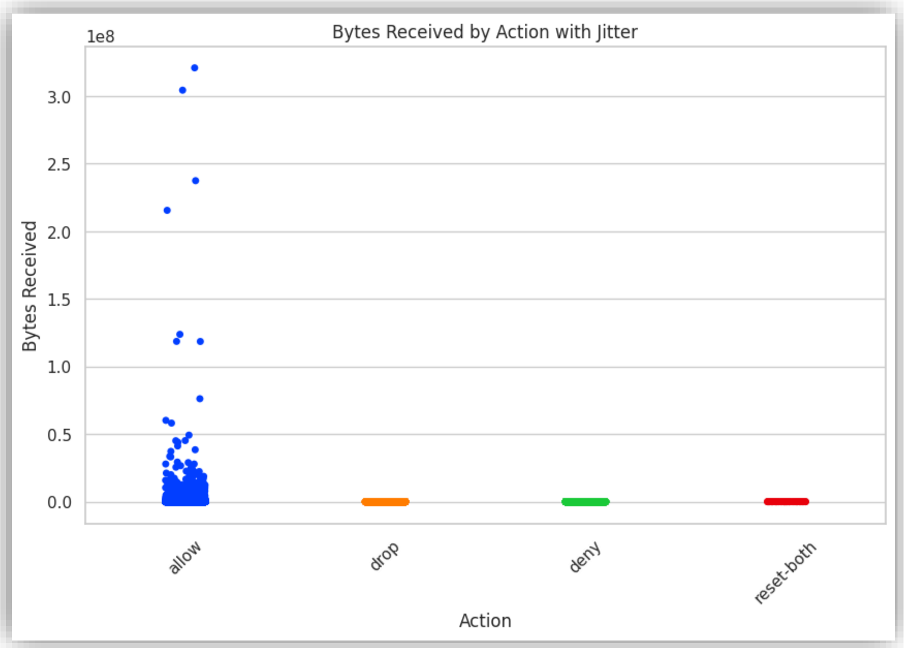


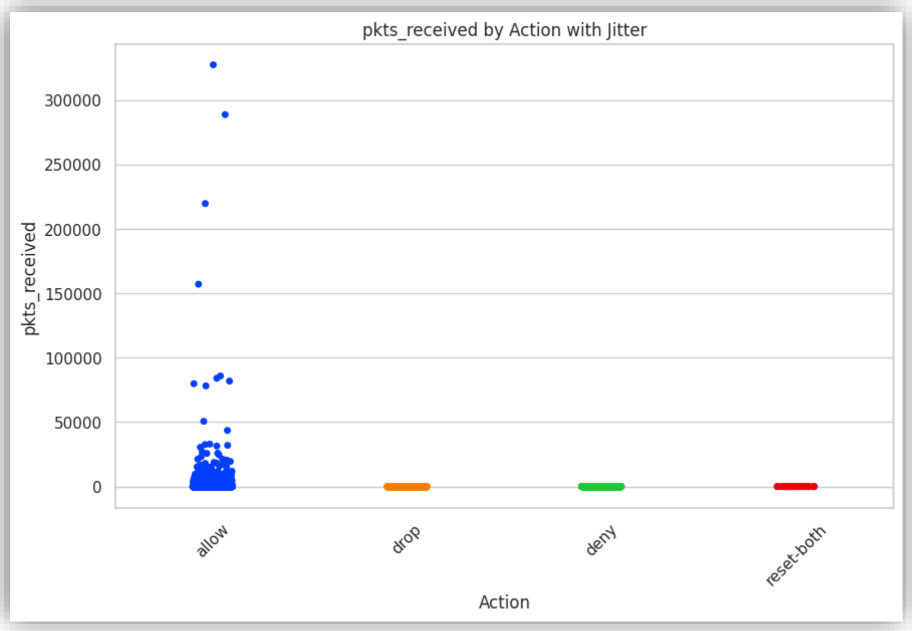
These imbalances will require monitoring as they have the potential to bias my models towards the more prevalent classes, jeopardizing the accuracy of predictions for less frequent actions like 'reset-both'. To address this, I will leverage techniques like Support Vector Machines (SVM) and Stochastic Gradient Descent (SGD), to enhance my classification approach.

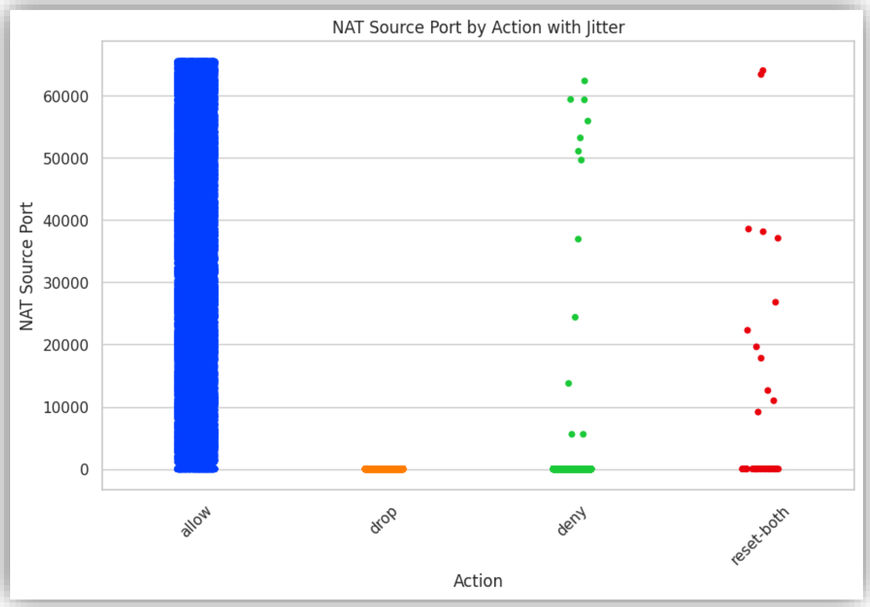
HANDLE MISSING VALUES

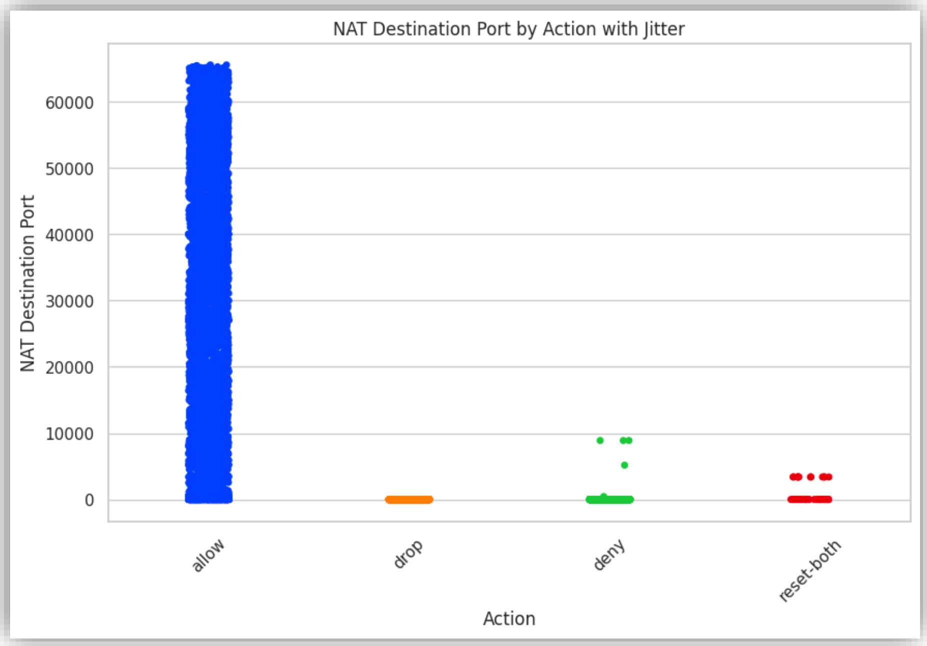
Next with this data I needed to approach the issue of missing values, recognizing the presence of zeros. For columns such as 'Source Port' and 'Destination Port', where missing data was minimal, I employed the mean of the columns to replace missing values, ensuring minimal distortion in the overall data distribution. This straightforward and efficient approach assumes that the missing values are randomly distributed, which may not always be the case. For more complex features with a higher percentage of missing values, such as 'NAT Source Port' and 'NAT Destination Port', I utilized the K-Nearest Neighbors (KNN) imputation technique. This method is advantageous as it estimates missing values based on the resemblance to the nearest instances in the feature space, thereby potentially offering more accurate imputations.

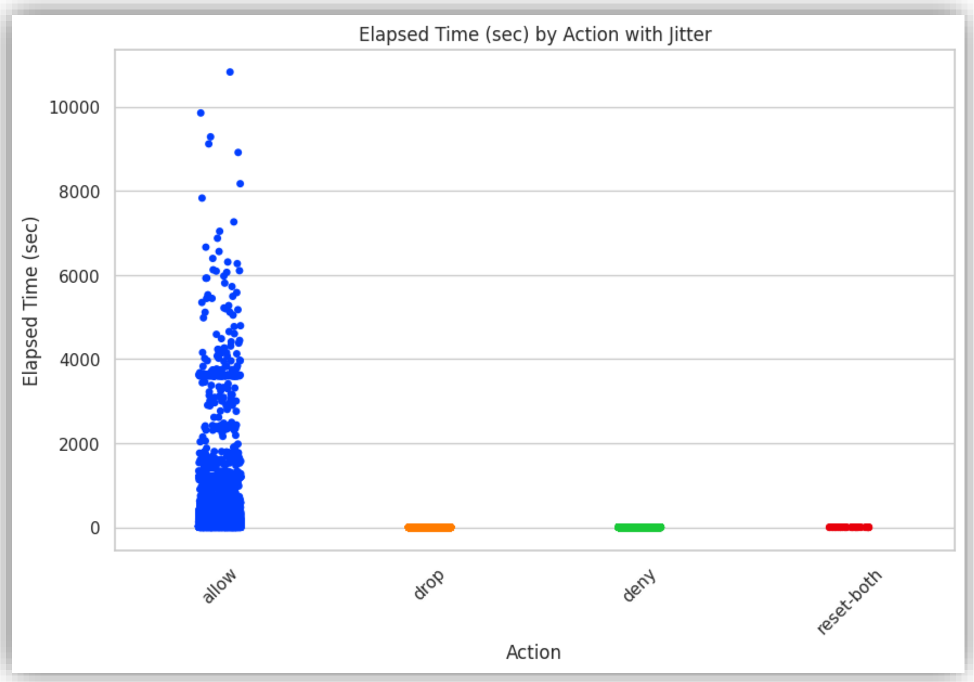
Metric	**Count of Missing Values**	**Percentage of Missing Values**
Source Port	173	0.27%
Destination Port	173	0.27%
NAT Source Port	28,432	43.38%
NAT Destination Port	28,432	43.38%
Bytes Received	31,574	48.18%
Elapsed Time (sec)	28,265	43.13%
pkts_received	31,574	48.18%


Bytes Received by Action with Jitter
1e8
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Bytes Received
allow
drop
deny
reset-both
Action


pkts_received by Action with Jitter
300000
250000
200000
150000
100000
50000
0
pkts_received
allow
drop
deny
reset-both
Action


NAT Source Port by Action with Jitter
NAT Source Port
60000
50000
40000
30000
20000
10000
0
allow
drop
deny
reset-both
Action


NAT Destination Port by Action with Jitter
NAT Destination Port
60000
50000
40000
30000
20000
10000
0
allow
drop
deny
reset-both
Action



MANIPULATE ATTRIBUTES

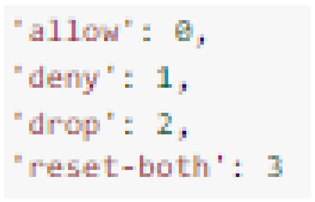
In my workflow, after identifying the relevant columns for scaling—'Bytes', 'Bytes Sent', 'Bytes Received', 'Packets', 'Elapsed Time (sec)', 'pkts_sent', and 'pkts_received'—I employed the StandardScaler to normalize these features. This normalization adjusts each feature to have a mean of zero and a standard deviation of one, thus ensuring no single feature influences the model due to its numerical scale. The transformed data was then merged back with the original dataframe.

Following this, I categorized network ports based on the frequency of occurrence within my dataset to ensure the models could handle the multiple port numbers for each specific port type. I embarked on a methodical analysis and reclassification of port numbers to better understand and prepare the data for modeling. Initially, I determined the frequency of each 'Source Port', 'Destination Port', 'NAT Source Port', and 'NAT Destination Port'. Recognizing the need for clearer categorization, I then grouped these ports based on specific frequency thresholds.

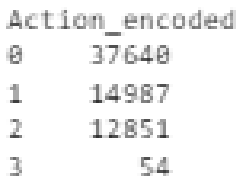
For instance, ports appearing less than a certain number of times were categorized under a common label (-1) to denote their infrequent usage, thereby simplifying the model's learning process. This was done separately for each type of port, ensuring that the thresholds were tailored to the distribution and importance of each port type. The 'Source Port', for example, had a lower threshold compared to the 'NAT Destination Port' due to its differing role and frequency in network traffic.

This reclassification not only aids in managing the vast data by reducing the number of categories the model needs to learn but also improves the interpretability and effectiveness of the predictive model. By focusing on significant ports and grouping the less frequent ones, the model can better generalize across similar types of network activity without overfitting to noise or rare occurrences.

In the final preparation stage of my dataset for predictive modeling, I addressed the 'Action' column by converting its categorical labels into numerical integers. I began by displaying the unique actions—'allow', 'drop', 'deny', 'reset-both'—to ensure all categories were accurately captured. Subsequently, I mapped each action to a unique integer (allow: 0, deny: 1, drop: 2, reset-both: 3), thereby transforming the 'Action' column into a new 'Action_encoded' column. This numerical representation not only simplifies the data but also enhances the efficiency and interpretability of subsequent modeling processes, ensuring that algorithms can effectively learn and make predictions based on these encoded actions. After encoding, I verified the transformation by examining the distribution of encoded values, confirming the data's integrity and readiness for modeling. Additionally, it is important to note the observed imbalance in my data during this transformation.



```
'allow': 0,
'deny': 1,
'drop': 2,
'reset-both': 3
```



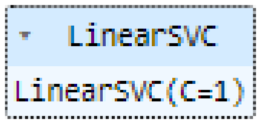
```
Action_encoded
0    37640
1    14987
2    12851
3       54
```

Now I am ready to split the data for subsequent modeling. Initially, I ensured the target variable 'Action_encoded' was properly formatted by confirming its type was represented as an integer. I then stored this into my new target vector named 'y'. Subsequently, I selected my scaled columns from the main DataFrame, which included features like 'Bytes', 'Bytes Sent', 'Bytes Received', as well as my newly created port groups. This selection was stored in a new DataFrame named 'X'. I always like to check the shape of the new DataFrame to confirm that the correct columns are included. Here, I see I have 65,532 entries and 11 features.

LINEAR SVC

For my initial model, I chose to use LinearSVC. This implementation offers flexibility in choosing penalties, loss functions and scales well with large datasets. Key differences between LinearSVC and SVC include the default loss function and how intercept regularization is handled by the two implementations. One of the main reasons I preferred LinearSVC over the regular SVC is its speed, which is crucial as it allows ample time to fine-tune the models for optimal performance.

Starting with a basic Linear SVC model I trained it on my dataset (X, y) and performed predictions. The accuracy of this initial model was calculated to be approximately 82.48%.



```
LinearSVC
LinearSVC(C=1)
```

To assess the model's performance further, I generated a confusion matrix, which helped visualize the true versus predicted classifications, showing that while the model performed well in some classes, there were notable discrepancies in others, especially in minority classes.

I also conducted cross-validation using stratified k-folds to ensure that each fold was a good representative of the whole, particularly important given the class imbalances in my dataset. The cross-validation scores varied around 83-87%, indicating fairly, consistent performance across different subsets of the data.

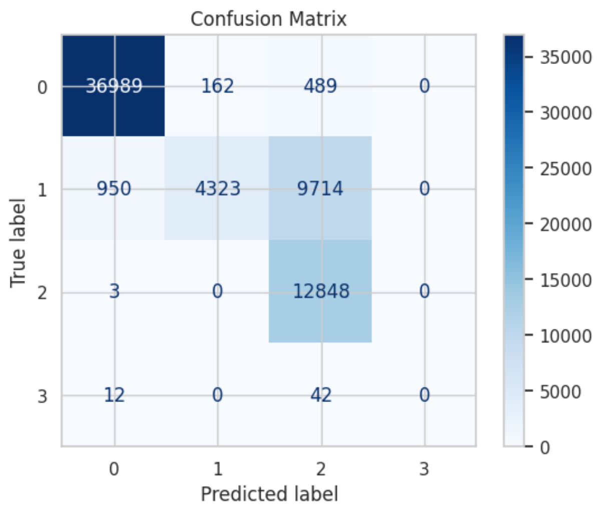
For my initial model tuning effort, I focused on optimizing the regularization parameter C for my LinearSVC model. By iterating over a range of C values, I C = 1 X 10-6 as the most effective, yielding an accuracy of 82.32% on training data. This result, coupled with cross-validation scores ranging from 62.94% to 82.29%, suggests that while the model is somewhat consistent, there is room for improvement.

In my latest adjustments to the LinearSVC model, I focused on optimizing the intercept scaling to enhance model performance. By testing values from 1 to 10, I found that an intercept scaling of 9.0 provided the highest accuracy at 81.48%. Further fine-tuning and adjustments to the model settings are still required.

Next, I focused on adjusting the tolerance parameter of the LinearSVC model. Starting with a range from 1 x 10-7 to 1 x 10-1, I used cross-validation to identify the tolerance that maximizes model accuracy. The best tolerance found was 1 x 10-7, which achieved an average cross-validation accuracy ranging from 63.66% to 83.78%. Subsequently, I fitted the LinearSVC model using this optimal tolerance, alongside the previously identified best intercept scaling, achieving a training accuracy of approximately 64.19%.

In the most recent phase of model tuning, I explored optimizing class weights after prior adjustments to regularization and intercept scaling. By testing different class weights, from equal distribution to weights inversely proportional to class frequencies, I aimed to correct the model's bias towards more frequent classes. This strategy involved cross-validation to determine the best weighting scheme that improves model performance, particularly for less frequent but crucial categories. The optimal configuration used balanced weights of {0: 1, 1: 10}, achieving an accuracy of 82.65% on validation sets. Cross-validation scores showed consistent improvements, and the refined model displayed a more balanced performance across different classes. Currently, this model appears to have the best overall performance. While there are still areas for improvement, such as increasing the representation of the target actions 'drop' and 'reset-both', I believe achieving accuracy results in the 90% range might be possible.

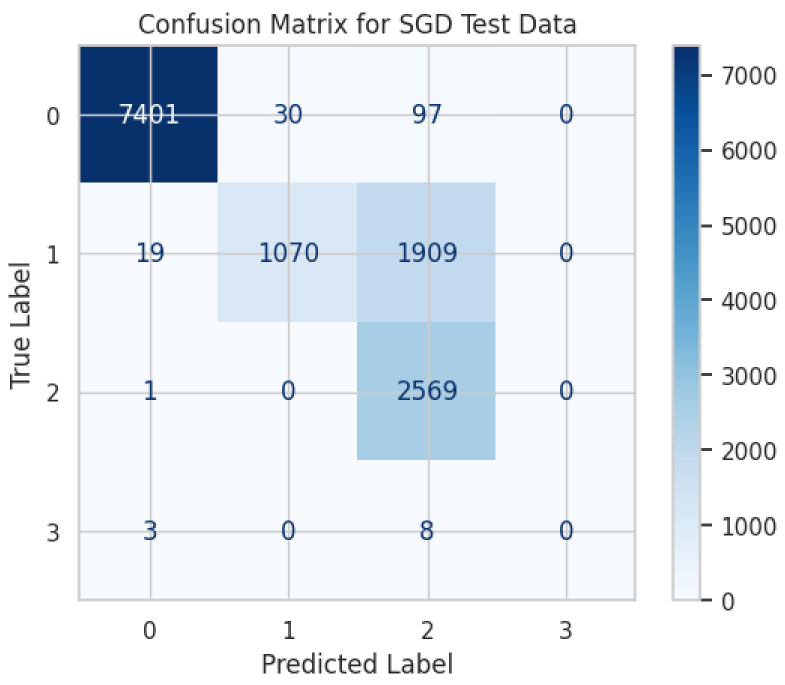
- **Fold 1**: 0.7845
- **Fold 2**: 0.8085
- **Fold 3**: 0.7907
- **Fold 4**: 0.6412
- **Fold 5**: 0.7841



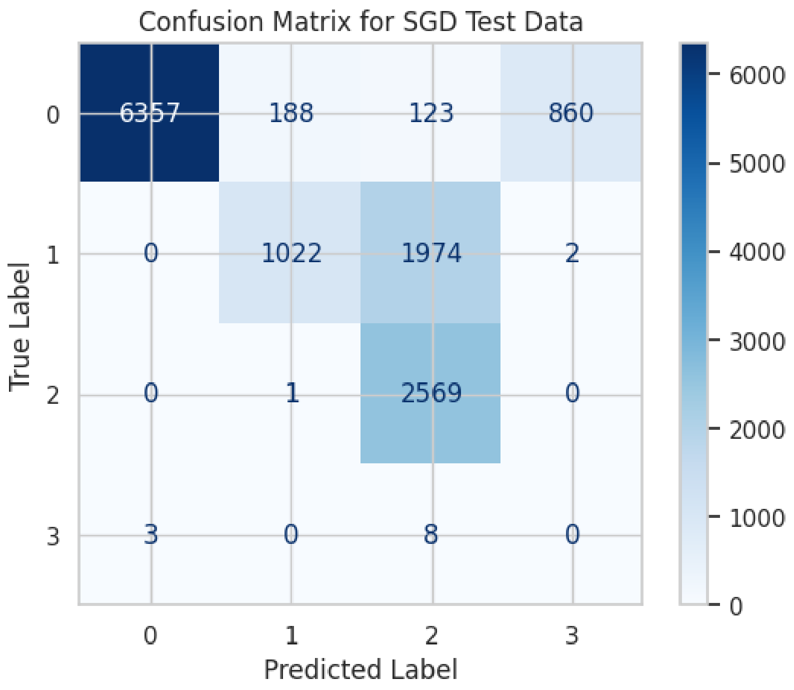
SGD

As I left off with my SVC model, there was an imbalance in my data, complicating classification predictions for the less common target actions 'drop' and 'reset-both'. I then turned to the SGDClassifier to help with this classification fine-tuning. This model is supposed to be faster and more robust in handling imbalanced data. Initially, I started with a hinge loss function, splitting my data into 80% training and 20% testing with stratified sampling to maintain the proportion of classes across the train and test sets. This resulted in a training set of 52,425 samples and a testing set of 13,107 samples, each featuring 11 predictors. After training, my SGD model achieved an accuracy of 82.67% on the test data, indicating strong baseline performance.

As I continued to optimize my model, I focused next on tuning the SGD classifier by experimenting with different learning rates and regularization parameters. Initially, I set up a grid of possible values, considering 'optimal', 'invscaling', and 'adaptive' for learning rates and varying alpha values to test regularization strength. For each combination, I used cross-validation to gauge the performance of the SGD model. If a learning rate required a specific eta value, I set it; accordingly, otherwise, I defaulted to 0.01. This approach allowed me to identify the combination that maximized accuracy, providing a robust way to select the best parameters for my model. The best parameters led to an accuracy score of 84.23%, as evidenced by the resulting confusion matrix, which displayed how the model's predictions spread across the different classes.



In my last tuning effort for the SGD classifier, I experimented with different maximum iteration numbers and the use of early stopping to improve model performance. I tested two configurations for maximum iterations—500 and 1000—both with and without early stopping. The results showed that setting 'max_iter' to 500 with 'early_stopping' enabled yielded the best accuracy of approximately 72.46%. This setup appears to optimize the training process efficiently, preventing overfitting while ensuring timely convergence. If time allowed, I would continue to refine these parameters to enhance the model's precision further.



CONCLUSION

In the realm of cybersecurity, managing firewall actions efficiently is paramount for organizations handling high volumes of network traffic. This study proposes a robust program aimed at automating the decision-making process for firewall actions by leveraging historical data to filter, classify, and automatically respond to network requests using Support Vector Machines (SVM) and Stochastic Gradient Descent (SGD) methodologies.

In conclusion, the tuning efforts on both the SVC and SGD models have provided significant insights and improvements in model performance, particularly in handling the imbalanced data inherent to the classification tasks. After various adjustments to regularization parameters and intercept scaling, the SVC model showed a notable increase in accuracy, especially when class weights were optimized to address the imbalance. The best parameters identified for the SVC model led to a moderate degree of precision.

Switching to the SGDClassifier, the model's robustness against imbalanced data was leveraged by experimenting with different learning rates, regularization strengths, and training strategies, including early stopping. Optimized for hinge loss, the SGD model demonstrated considerable adaptability and efficiency, achieving an accuracy of approximately 84.23%, with learning rate adjustments and iterative enhancements proving beneficial.

The computational efficiency of both SVM and SGD models was evaluated on Google Colab using a T4 GPU, completing the exploratory data analysis, model tuning, and adjustments in roughly 15-20 minutes. This rapid processing underscores the effectiveness of powerful hardware accelerators in machine learning workflows, enabling quick iterations and timely optimizations. This combination of methodical tuning and robust computational resources forms a potent approach to effectively tackle complex, imbalanced datasets. This project highlights the model's potential to manage firewall traffic more efficiently, significantly reducing the burden on IT infrastructure.