BLOCKCHAIN ANAMOLY DETECTION

Machine Learning

GROUP 5

Dr Shih Yu Chang

Harivardhana Naga Naidu Polireddi (017437238)
Karthik Nimmagadda (016996148)
Sahithi Reddy Chandiri (017445038)
Shreya Sree Matta (017440618)
Yashasvi Kotra (017466436)

INTRODUCTION

Our primary goal is to develop a sophisticated machine learning system designed to identify and prevent fraudulent transactions within blockchain networks. This project is crucial as it addresses the growing need for enhanced security measures in blockchain technologies, where anomalies can indicate potential fraud, security breaches, or other unauthorized activities. By leveraging advanced anomaly detection techniques, we aim to safeguard the integrity and reliability of blockchain transactions, ensuring they remain secure, transparent, and efficient. Our approach combines rigorous data analysis with innovative machine learning algorithms to detect irregular patterns and behaviors effectively, enhancing the overall trustworthiness of the blockchain infrastructure.

DATASET

1. Dataset Description
   - Title: Bitcoin Transaction Network Metadata (2011-2013)
   - Source: IEEE Dataport
   - Content: This dataset consists of metadata for Bitcoin transactions from 2011 to 2013, including transaction IDs, timestamps, wallet addresses, transaction values, and transaction inputs and outputs.

WORKFLOW

Business Understanding

We define the project's objectives and requirements .This phase focuses on identifying the key challenges that the anomaly detection system needs to address, such as detecting fraudulent transactions and ensuring data integrity within the blockchain. The outcome will be a clear scope and expected benefits of the project.

Data Understanding

We Gather and analyze blockchain transaction logs to identify normal and anomalous patterns. Assess data quality, structure, and potential issues like missing values or outliers. This insight is critical for guiding subsequent preprocessing and modeling efforts.

Data Preprocessing

Cleaning and transforming the data by addressing missing values, removing duplicates, and filtering irrelevant information. Use normalization, scaling, and feature engineering to enhance the data's relevance for anomaly detection, focusing on features that indicate unusual behavior.

Data Modeling

We choose suitable machine learning models like Isolation Forest, K-Means, and other models for anomaly detection.

Evaluation

Evaluation of the models using metrics such as accuracy, precision, recall, and F1-score to assess their performance. Refine and adjust the models based on their results on validation and test sets to improve generalization to new data.

Data Preprocessing

Checked for any missing values / null values. We have no null values in our dataset.
Detected Outliers using Z-score method where we have detected 69205 outliers.

```
tx_hash                0
indegree               0
outdegree              0
in_btc                 0
out_btc                0
total_btc              0
mean_in_btc            0
mean_out_btc           0
in_malicious           0
out_malicious          0
is_malicious           0
out_and_tx_malicious   0
all_malicious          0
dtype: int64
```

```
[3]: df.head()
```

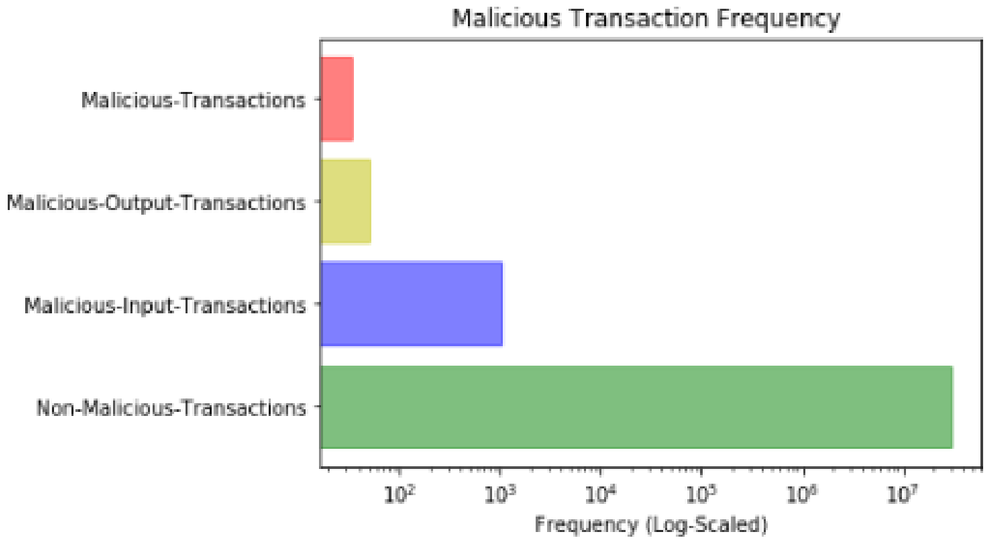
Out[3]:

	tx_hash	indegree	outdegree	in_btc	out_btc	total_btc	mean_in_btc	mean_out_btc	in_malicious	out_malicious
0	0437cd7f8525ceed2324359c2d0ba26006d92d856a9c20...	0	1	0.0	50.0	50.0	0.0	50.0	0	0
1	f4184fc596403b9d638783cf57adfe4c75c605f6356fbc...	1	2	50.0	50.0	100.0	50.0	25.0	0	0
2	ea44e97271691990157559d0bdd9959e02790c34db6c00...	1	1	10.0	10.0	20.0	10.0	10.0	0	0
3	a16f3ce4dd5deb92d98ef5cf8afeaf0775ebca408f708b...	1	1	40.0	30.0	70.0	40.0	30.0	0	0
4	591e91f809d716912ca1d4a9295e70c3e78bab077683f7...	1	2	30.0	30.0	60.0	30.0	15.0	0	0

Feature Engineering

Feature Transformation: Apply logarithmic scaling to features like 'in_btc' and 'out_btc' to normalize the data and reduce skewness.
Used RobustScaler for having least effect of outliers.

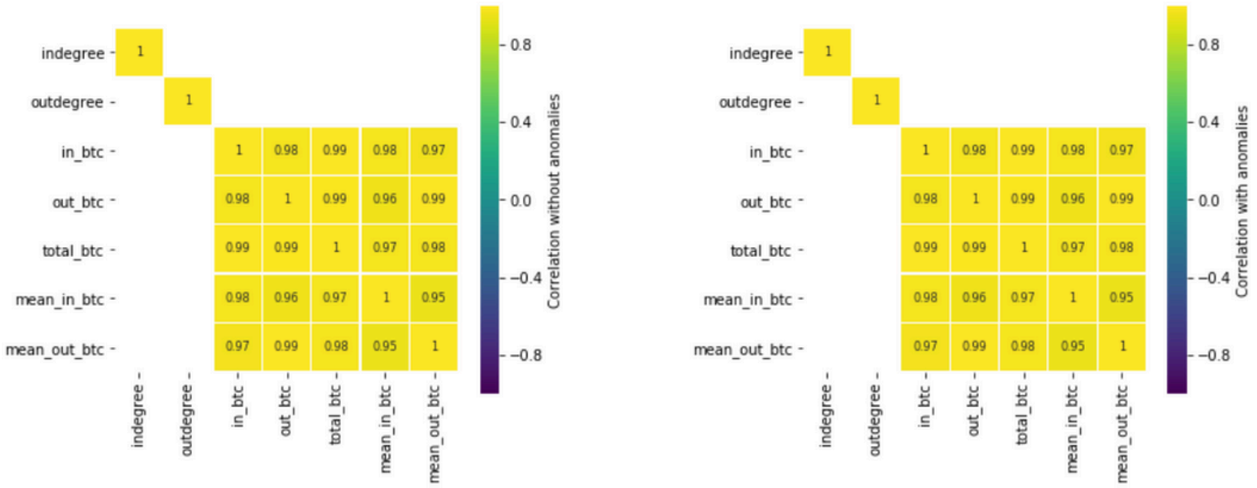
```
o_df2 = pd.DataFrame()
o_df2['indegree'] = df['indegree']
o_df2['outdegree'] = df['outdegree']
o_df2['in_btc'] =  np.log1p(df['in_btc'])
o_df2['out_btc']= np.log1p(df['out_btc'])
o_df2['total_btc']= np.log1p(df['total_btc'])
o_df2['mean_in_btc']= np.log1p(df['mean_in_btc'])
o_df2['mean_out_btc']= np.log1p(df['mean_out_btc'])
o_df2['tx_hash'] = df['tx_hash']
o_df2['in_malicious'] = df['in_malicious']
o_df2['out_malicious'] = df['out_malicious']
o_df2['is_malicious'] = df['is_malicious']
o_df2['out_and_tx_malicious'] = df['out_and_tx_malicious']
o_df2['all_malicious'] = df['all_malicious']

o_df2.describe()
```



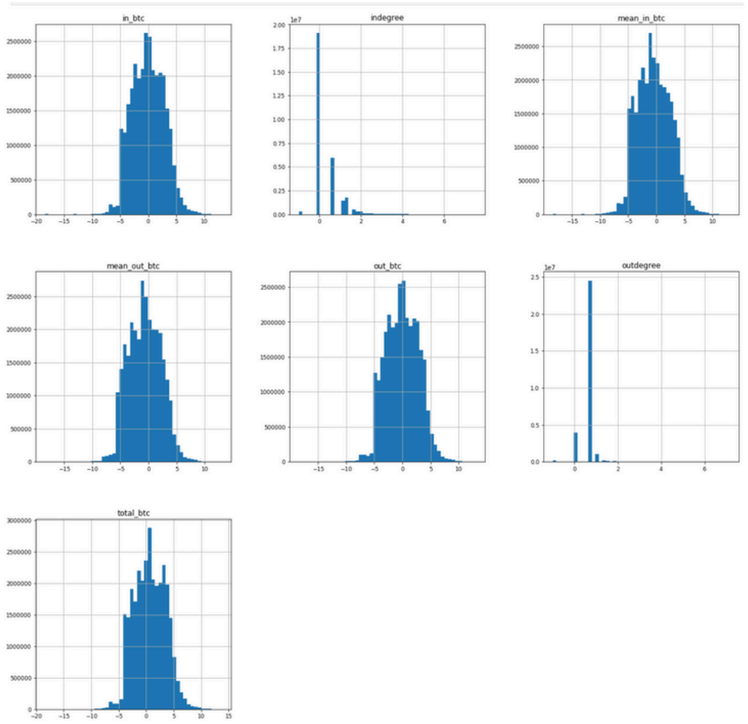
Out[3]:

e	in_btc	out_btc	total_btc	mean_in_btc	mean_out_btc	in_malicious	out_malicious	is_malicious	out_and_tx_malicious	all_malicious
1	0.0	50.0	50.0	0.0	50.0	0	0	0	0	0
2	50.0	50.0	100.0	50.0	25.0	0	0	0	0	0
1	10.0	10.0	20.0	10.0	10.0	0	0	0	0	0
1	40.0	30.0	70.0	40.0	30.0	0	0	0	0	0
2	30.0	30.0	60.0	30.0	15.0	0	0	0	0	0

EDA



The correlation heatmap reveals a strong linear relationship among transaction metrics such as incoming and outgoing Bitcoin amounts and the number of transactions. It shows that higher transaction connections correlate with higher transaction values. Anomalies included in the analysis do not significantly alter these relationships, suggesting that typical transaction behaviors are maintained even when outliers are present. This insight is essential for identifying consistent patterns and potential anomalies in blockchain transactions.

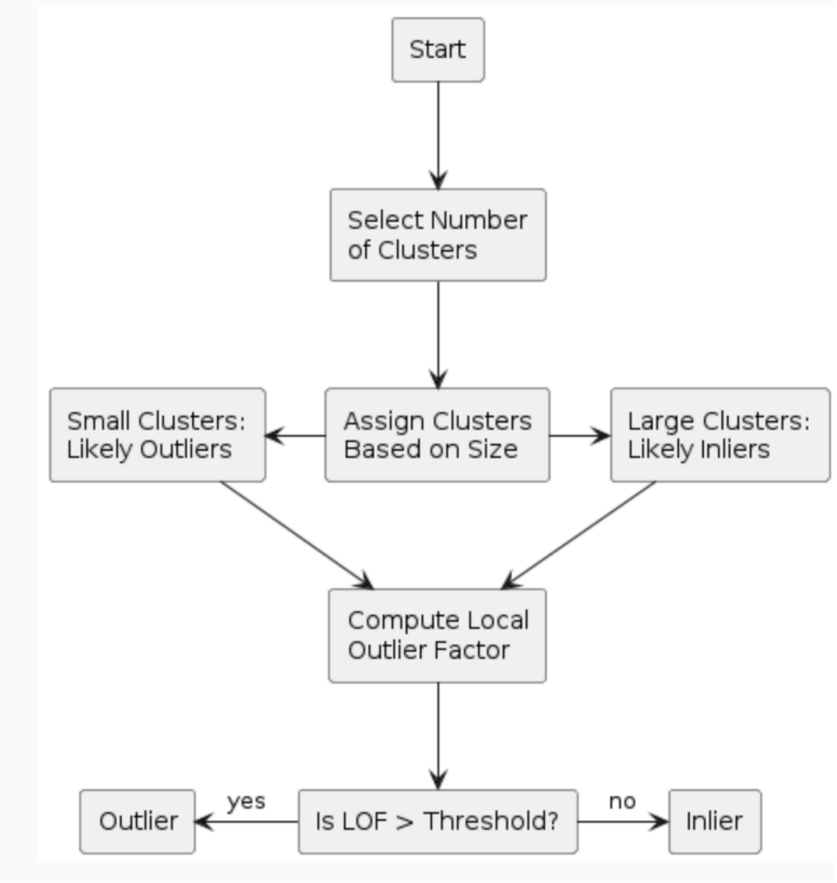


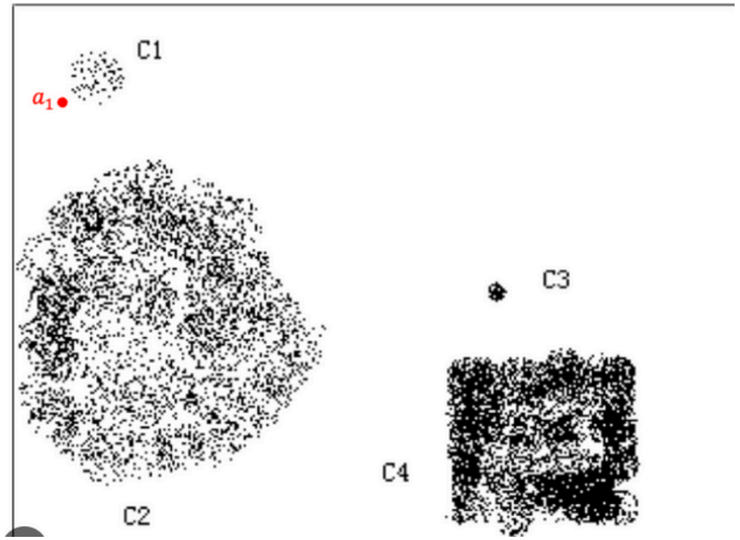
Most values follow a normal distribution with notable outliers, suggesting typical transaction behaviors and potential anomalies. The 'Indegree' and 'Outdegree' metrics, which show a high skewness, highlight few nodes with unusually high transactions, potentially flagging important areas for fraud detection.

Connectivity-Based Outlier Factor (CBLOF)

- It defines anomalies as a combination of local distances to nearby clusters and the size of the clusters to which the data point belongs.
  It first clusters data points into large or small clusters.
- Data points of a small cluster next to a nearby large cluster are identified as outliers.
  The local outliers may not be a singular point, but a small group of isolated points.
- CBLOF builds at the “cluster” level and LOF builds at the single data point level.
- First it assigns a data point to one and only one cluster. K-means is therefore a common cluster algorithm for CBLOF.
- The second step ranks clusters according to cluster size from large to small and get the cumulative data counts. The third step calculates the distance of a data point to the centroid and the outlier score.

CBLOF

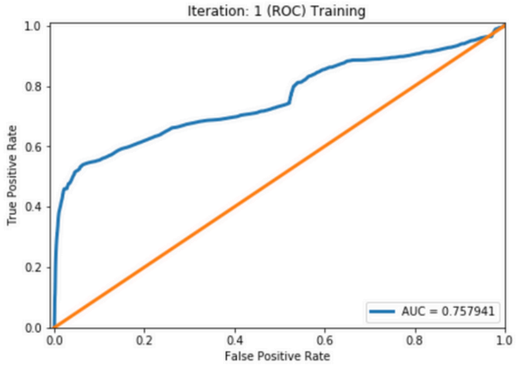

Start
Select Number of Clusters
Assign Clusters Based on Size
Small Clusters: Likely Outliers
Large Clusters: Likely Inliers
Compute Local Outlier Factor
Is LOF > Threshold?
yes
no
Outlier
Inlier

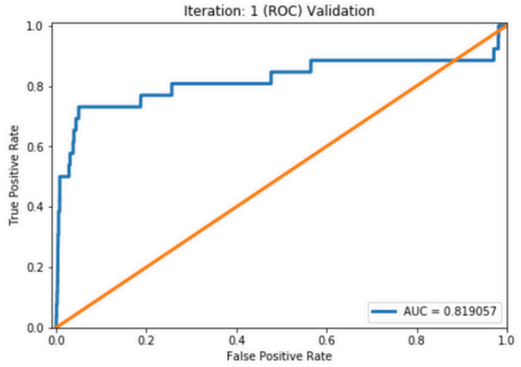

C1
a_1
C2
C3
C4

CBLOF

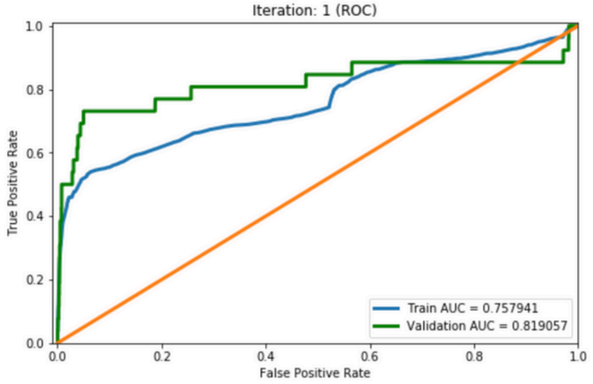
```
evaluation_results_training = {}
evaluation_results_test = {}
models = {}

for index in range(0, iterations):
    sampled_Xtrain, sampled_ytrain = fetch_training_data(training_sample_size)
    X_train, y_train = balance_classes(sampled_Xtrain, sampled_ytrain, oversampling_fraction)
    print(f"Iteration {index + 1} underway...")

    start_time = time.time()
    classifier = train_model(X_train)
    end_time = time.time()
    store_model(classifier, model_path + f'i={index + 1}.h5')
    predicted_labels = classifier.labels_  # Binary labels (0: inliers, 1: outliers)
    anomaly_scores = classifier.decision_scores_
    duration_training = round(end_time - start_time, 4)
    models[index + 1] = classifier
    evaluation_results_training[index + 1] = assess_performance(y_train, predicted_labels, anomaly_scores, duration_training)

    print(f"Iteration {index + 1} complete")

    start_test = time.time()
    predicted_test_labels = classifier.predict(original_Xtest)
    test_scores = classifier.decision_function(original_Xtest)
    end_test = time.time()
    duration_test = round(end_test - start_test, 4)
    evaluation_results_test[index + 1] = assess_performance(original_ytest, predicted_test_labels, test_scores, duration_test)

    print(f"Iteration: ({index + 1}/{iterations}) Training Time: {duration_training} seconds Testing Time: {duration_test} seconds")

print("Modeling process across iterations is complete!")
```

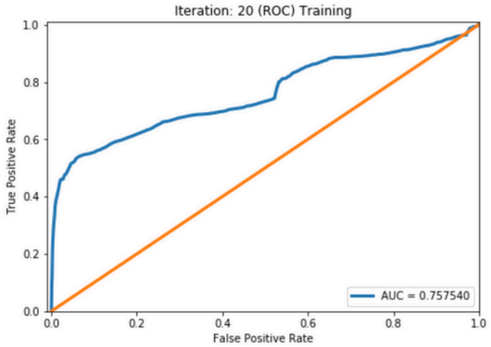
We trained a Clustering-Based Local Outlier Factor (CBLOF) model iteratively to detect anomalies in a dataset.Each iteration involves sampling the training data, oversampling the minority class, fitting the model, and evaluating its performance.The model training process is repeated for 20 iterations and number of clusters are 8.The model's performance improves iteratively by balancing the dataset and adjusting to the training data.Various evaluation metrics such as accuracy, precision, recall, F1-score, and ROC-AUC are computed to assess the model's performance on both training and testing datasets.The iterative process ensures that the model is robust and can effectively identify outliers in the dataset.

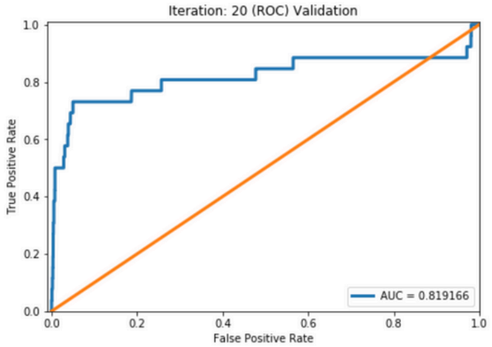




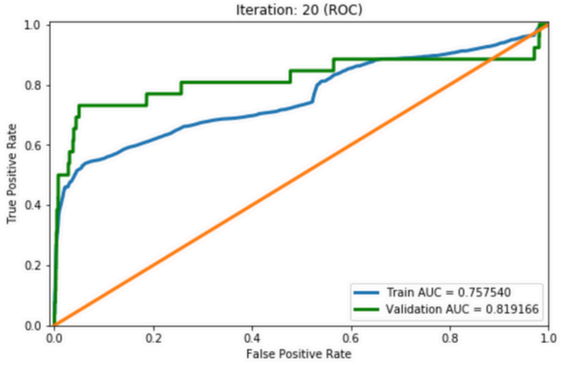
<Figure size 432x288 with 0 Axes>





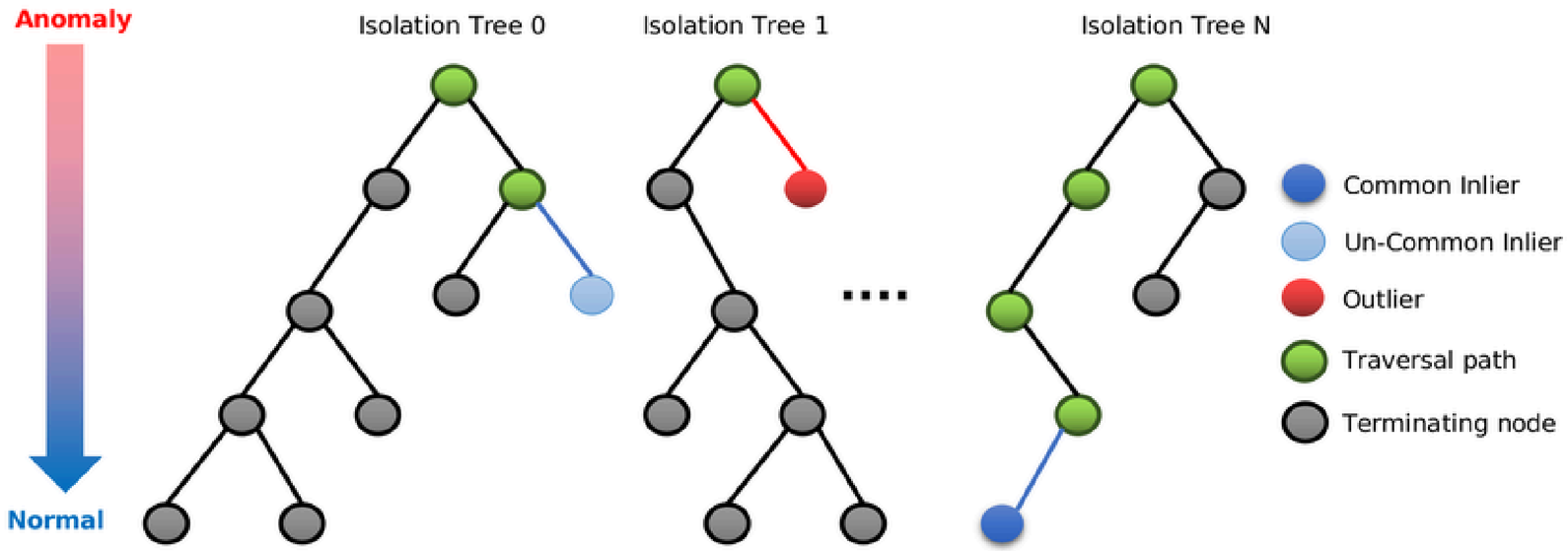


<Figure size 432x288 with 0 Axes>



The ROC curves here compare the true positive rate and false positive rate across various threshold settings during both training and validation phases. The curves show a trade-off between precision and recall, indicating how the choice of threshold impacts the model's ability to distinguish between classes. The area under the curve (AUC) values provide a quantitative measure of the model’s overall ability to discriminate between positive and negative classes across all thresholds. Higher AUC values in validation suggest the model generalizes well, maintaining a good balance between sensitivity (true positive rate) and specificity (false positive rate), which is crucial for effective anomaly detection in practical scenarios.

ISOLATION FOREST



In our project, we're utilizing the Isolation Forest algorithm to detect anomalies within blockchain transactions, capitalizing on its ability to effectively isolate atypical instances. We start by sampling and oversampling the data to manage class imbalances, making rare fraudulent transactions more prevalent for model training. The Isolation Forest model identifies anomalies based on the ease with which instances can be isolated, assigning scores that reflect their anomaly status. We adjust thresholds on these scores to optimize the balance between precision and recall, fine-tuning the model across multiple iterations to enhance detection capabilities.

```python
training_evaluations = {}
test_evaluations = {}
model = {}


for i in range(0, iterations):
    sample_Xtrain, sample_ytrain = get_training_sample(training_sample_size)
    X_train, y_train = oversample_minority_class(sample_Xtrain, sample_ytrain, oversampling_fraction)
    print("Iteration "+ str(i+1) + " in progress...")

    #Modeling
    start = time.time()
    clf = fit_model(X_train)
    end = time.time()
    save_model(clf, model_path+'i='+str(i+1)+'.h5')
    y_train_pred = clf.labels_  # binary labels (0: inliers, 1: outliers)
    y_train_scores = clf.decision_scores_  # raw outlier scores

    training_time = round(end - start, 4)
    model[i+1] = clf
    training_evaluations[i+1] = evaluate(y_train, y_train_pred, y_train_scores, training_time)

#     print("Iteration "+ str(i+1)+ " Trained")

    test_start = time.time()
    y_test_pred = clf.predict(original_Xtest)
    y_test_scores = clf.decision_function(original_Xtest)
    test_end = time.time()
    testing_time = round(test_end - test_start, 4)
    test_evaluations[i+1] = evaluate(original_ytest, y_test_pred, y_test_scores, testing_time)

    print("Iteration: (" + str((i+1)) + "/" + str(iterations) + ")    Training Time: "+str(training_time)+" seconds     Testing Time: "+ str(t

print("Iterative Modeling Completed!")
```

It begins by repeatedly sampling and oversampling data to balance class distribution, followed by training the model on this prepared data. Each iteration involves timing the model training and prediction phases, saving the model, and evaluating its performance on both training and testing sets to record metrics such as accuracy and outlier detection rates. Results are stored in dictionaries for each iteration, providing insights into the model's effectiveness and computational efficiency across different runs. This systematic approach helps optimize the model by adjusting parameters and improving its ability to identify anomalies accurately.

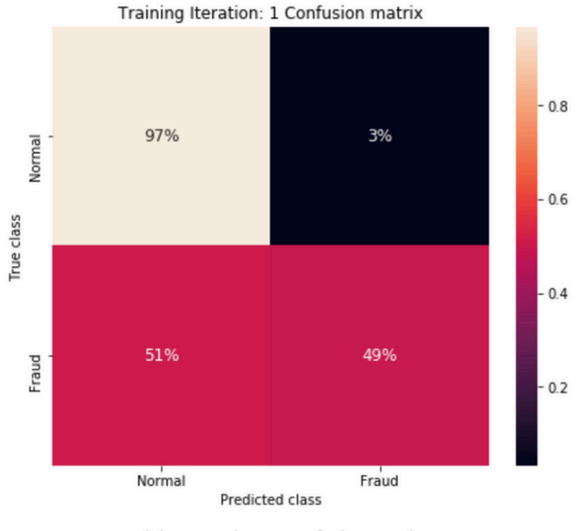
```python
import time
from hyperopt import Trials, STATUS_OK, fmin, tpe, hp
space = {
    'n_estimators': hp.uniform('n_estimators', 5, 100),
    'outliers_fraction': hp.uniform('outliers_fraction', 0.05, 0.25)
}
random_state = 42
training_sample_size = int(len(original_ytrain) / 10)
sample_Xtrain, sample_ytrain = get_training_sample(training_sample_size)
print('Training With:')
print(pd.value_counts(sample_ytrain))

def hyper_param_opt(space):
    start = time.time()
    n_estimators = int(space['n_estimators'])
    outliers_fraction = space['outliers_fraction']
    X_train, y_train = oversample_minority_class(sample_Xtrain, sample_ytrain, outliers_fraction)
    clf = IForest(n_estimators=n_estimators, contamination=outliers_fraction, random_state=random_state, n_jobs=-1)
    clf.fit(X_train)
    y_train_pred = clf.labels_
    score = f1_score(y_train, y_train_pred, average="macro")
    val = 1 - score
    end = time.time()
    print('time: ' + str(round(end - start, 4)) + ' seconds')
    print('params: ' + str(space))
    print('val: ' + str(val))
    print('')
    return {'loss': val, 'status': STATUS_OK}
trials = Trials()
best = fmin(fn=hyper_param_opt, space=space, algo=tpe.suggest, max_evals=100, trials=trials)

print('\n\n\nbest: {}'.format(best))
```

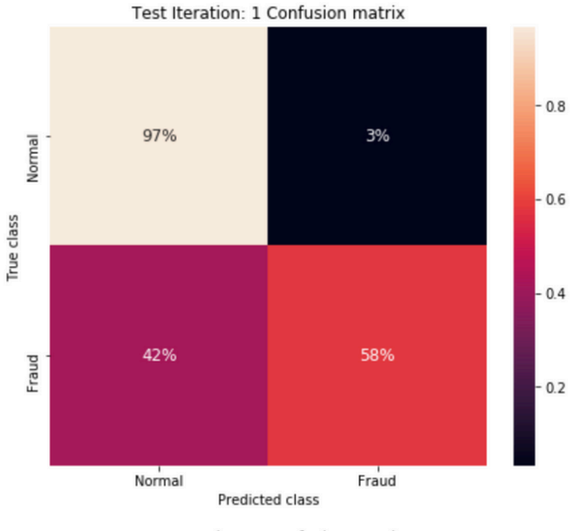
we used the Hyperopt library to optimize hyperparameters for an isolation forest model. We defined a search space for the number of estimators and the outlier fraction, which are key parameters for the model. For each iteration, we took a subset of our training data, performed oversampling to balance the classes, and then trained the isolation forest model. We evaluated the model's performance using the macro F1 score and used the time taken for each fit as part of our performance metrics. The optimization process iterated through different combinations of hyperparameters to find the best model configuration based on the lowest loss, which was one minus the F1 score.

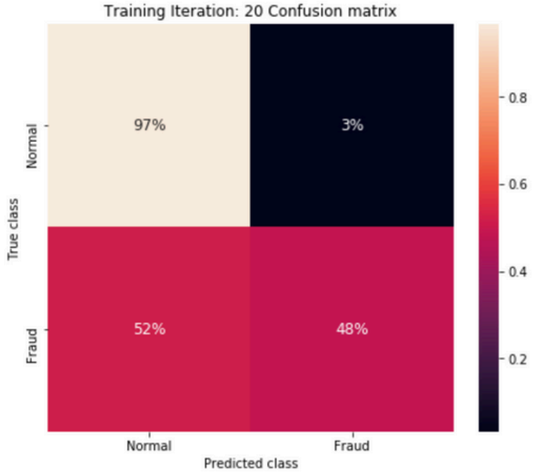
High Recall, Lower Precision: The model excels at detecting positive (anomalous) instances with a high recall rate, ensuring that most fraudulent transactions are captured. However, it struggles with precision, resulting in a higher number of false positives.

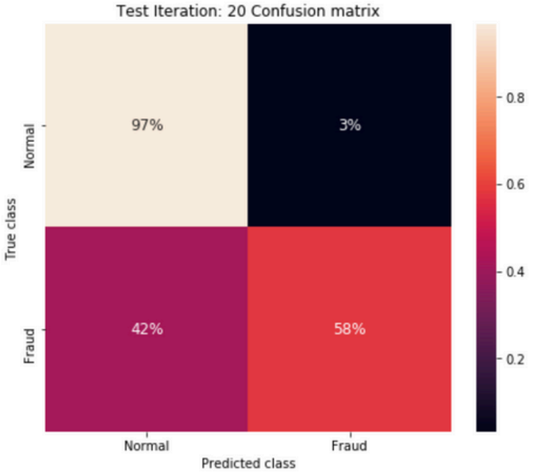
Hyperparameter Tuning: Enhancements in model performance were driven by optimizing hyperparameters such as n_estimators, max_depth, and min_samples_split. This tuning helped balance the model, reducing overfitting and underfitting.

Confusion Matrix Insights: The confusion matrix displays a significant number of true positives, affirming the model's effectiveness at identifying anomalies. Nonetheless, the presence of false positives suggests the model could benefit from further refinement to improve precision.

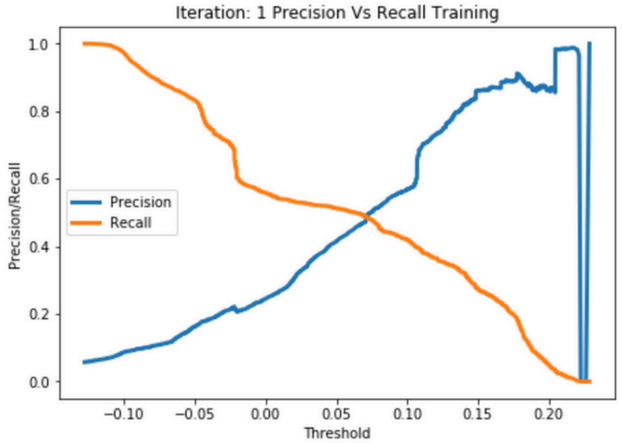


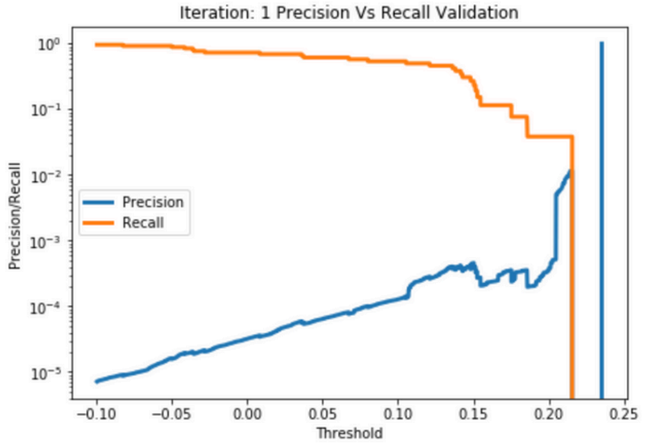


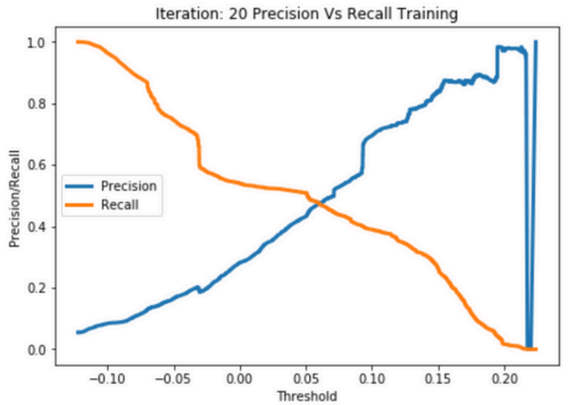


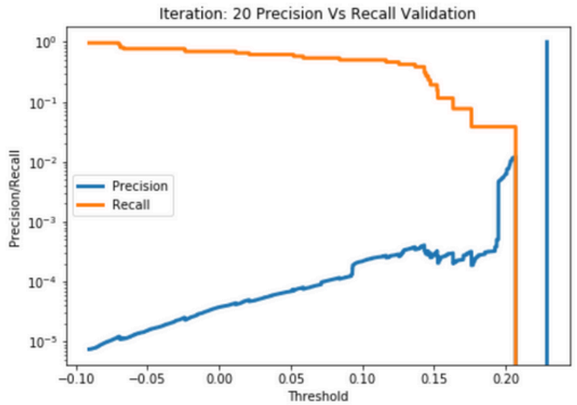


Initially, the model identifies 51% of fraud correctly in training but only 42% in testing, suggesting some overfitting. By the twentieth iteration, these figures slightly improve to 52% in training and 58% in testing, indicating a progressive but modest enhancement in detecting fraud while maintaining robustness against normal transactions.









In training, precision peaks and then drops, while recall decreases as thresholds increase, showing a trade-off between identifying more anomalies and maintaining accuracy. During validation, precision and recall remain stable until they split at higher thresholds, highlighted by the logarithmic scale used for better visibility of changes at low thresholds. These trends are critical for adjusting the model to balance between detecting anomalies and reducing false positives.

K-MEANS

The K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible.

- The '*means*' in the K-means refers to averaging of the data; that is, finding the centroid.

The objective is to partition data into 'k' clusters, minimizing the intra-cluster variance. It seeks to form groups where data points within each cluster are more similar to each other than to those in other clusters.

Advantages:
K Means is faster as compare to other clustering technique.
It provides strong coupling between the data points.
Disadvantages:
K Means cluster do not provide clear information regarding the quality of clusters.
Different initial assignment of cluster centroid may lead to different clusters.
Also, K Means algorithm is sensitive to noise.

K-MEANS

K-means Clustering Setup:

- Cluster Initialization: The K-means algorithm is initialized with a specified number of clusters. K-means clustering is an unsupervised learning algorithm that identifies clusters in the data based on feature similarity.

- Model Training: The scaled and preprocessed data is fed into the K-means model to learn how to cluster similar transactions together.

Cluster Analysis:

- Cluster Merging: After clustering, you have logic to merge specific clusters based on predefined conditions. This might be done to consolidate clusters that are very similar or to fine-tune the separation between normal and anomalous transactions.

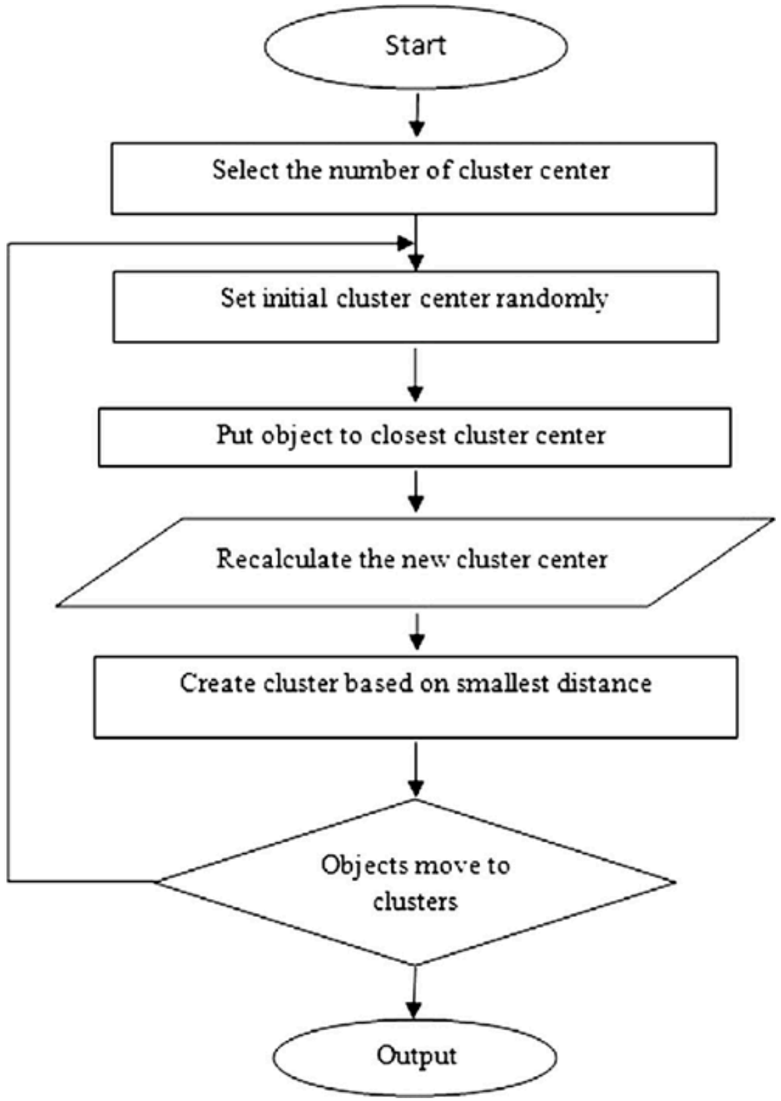
Evaluation Metrics:

- Various metrics such as accuracy, precision, recall, F1-score, and ROC-AUC score are calculated to evaluate the performance of the clustering model. These metrics provide insights into how well the model is performing in terms of identifying anomalies.

Confusion Matrix Visualization:

- Visual Representation: Confusion matrices for both training and testing sets are visualized using heatmaps. These visualizations help in understanding the true positive, false positive, true negative, and false negative rates of the model.

Model Architecture/ Data Flow



Model Implementation

•Initialized variables: Models, sse, calinski_harabaz

-Models: Dictionary to store MiniBatchKmeans model objects (No. of clusters)

-Sse- Dictionary to store sum of squared distances of samples to their closest cluster center for each k.

-calinski_harabaz- t store index for each k, a metric that evaluates model by measuring ratio between cluster dispersion.

•Clustering with Mini Batches:

-Initially it selects the cluster centers

-Selects the number of clusters to form and number of clusters to generate

-Selecting number of times to run with different centroid seeds. Maximim iterations to run until best output is 15000 and the batch size being 256.

•Model Fitting and Storing: We trained the model using kmeans.fit(original_Xtrain), stored through models dictionary.

•Label extraction: Extract labels of each point which indicates to which cluster a point is assigned.

•Inertia Calculation: sse[k] = kmeans.inertia_: Calculates and stores the sum of squared distances of samples to their nearest cluster center for each k. Lower values of inertia mean that the clusters are more dense and well separated, which typically suggests a model with better defined clusters.

Tabular Train Data Evaluation

```
Model with K=2
==================================

Predicted for K=2
Cluster 1:       Malicious: 14 (17.073%)

Cluster 2:       Malicious: 68 (82.927%)
```

```
Model with K=3
==================================

Predicted for K=3
Cluster 1:       Malicious: 56 (68.293%)

Cluster 2:       Malicious: 6 (7.317%)

Cluster 3:       Malicious: 20 (24.39%)
```

```
Model with K=9
==================================

Predicted for K=9
Cluster 1:       Malicious: 1 (1.22%)

Cluster 2:       Malicious: 24 (29.268%)

Cluster 3:       Malicious: 4 (4.878%)

Cluster 4:       Malicious: 21 (25.61%)

Cluster 5:       Malicious: 1 (1.22%)

Cluster 6:       Malicious: 19 (23.171%)

Cluster 7:       Malicious: 3 (3.659%)

Cluster 8:       Malicious: 2 (2.439%)

Cluster 9:       Malicious: 7 (8.537%)
```

```
Model with K=12
==================================

Predicted for K=12
Cluster 1:       Malicious: 1 (1.22%)

Cluster 2:       Malicious: 20 (24.39%)

Cluster 3:       Malicious: 7 (8.537%)

Cluster 4:       Malicious: 0 (0.0%)

Cluster 5:       Malicious: 1 (1.22%)

Cluster 6:       Malicious: 4 (4.878%)

Cluster 7:       Malicious: 5 (6.098%)

Cluster 8:       Malicious: 3 (3.659%)

Cluster 9:       Malicious: 2 (2.439%)

Cluster 10:      Malicious: 20 (24.39%)

Cluster 11:      Malicious: 0 (0.0%)

Cluster 12:      Malicious: 19 (23.171%)
```

Tabular Test Data Evaluation

```
Model with K=2
==================================

Predicted for K=2
Cluster 1:       Malicious: 3 (11.538%)

Cluster 2:       Malicious: 23 (88.462%)
```

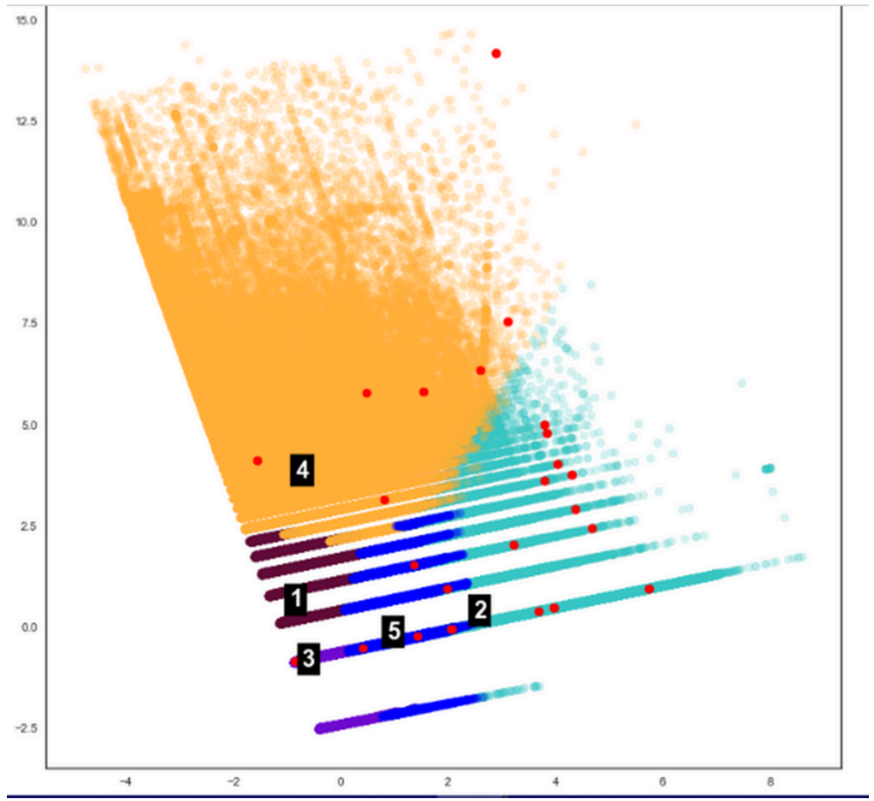
```
Model with K=3
==================================

Predicted for K=3
Cluster 1:       Malicious: 19 (73.077%)

Cluster 2:       Malicious: 2 (7.692%)

Cluster 3:       Malicious: 5 (19.231%)
```

```
Model with K=9
==================================

Predicted for K=9
Cluster 1:       Malicious: 1 (3.846%)

Cluster 2:       Malicious: 10 (38.462%)

Cluster 3:       Malicious: 1 (3.846%)

Cluster 4:       Malicious: 5 (19.231%)

Cluster 5:       Malicious: 1 (3.846%)

Cluster 6:       Malicious: 7 (26.923%)

Cluster 7:       Malicious: 0 (0.0%)

Cluster 8:       Malicious: 1 (3.846%)

Cluster 9:       Malicious: 0 (0.0%)
```

```
Model with K=12
==================================

Predicted for K=12
Cluster 1:       Malicious: 1 (3.846%)

Cluster 2:       Malicious: 4 (15.385%)

Cluster 3:       Malicious: 0 (0.0%)

Cluster 4:       Malicious: 1 (3.846%)

Cluster 5:       Malicious: 1 (3.846%)

Cluster 6:       Malicious: 1 (3.846%)

Cluster 7:       Malicious: 2 (7.692%)

Cluster 8:       Malicious: 0 (0.0%)

Cluster 9:       Malicious: 1 (3.846%)

Cluster 10:      Malicious: 8 (30.769%)

Cluster 11:      Malicious: 0 (0.0%)

Cluster 12:      Malicious: 7 (26.923%)
```

2 D Visualization

```
#set font size of labels on matplotlib plots
plt.rc('font', size=16)

#set style of plots
sns.set_style('white')

#define a custom palette
customPalette = ['#630C3A', '#39C8C6', '#720cd2', '#FFB139', '#0004FF', '#04FF00', '#00C3FF', '#FFD800', '#00F3FF', '#7000F
sns.set_palette(customPalette)
#sns.palplot(customPalette)

groups = {}
for i in range(0, k):
    groups[i] = i

#create a new figure
plt.figure(figsize=(12,12))

#Loop through labels and plot each cluster
for i, label in enumerate(groups.keys()):
    #add data points
    plt.scatter(x=df_pca.loc[df_pca['cluster']==label, 'pca_0'],
                y=df_pca.loc[df_pca['cluster']==label,'pca_1'],
                color=customPalette[i],
                alpha=0.20)
    #add label
    plt.annotate((int(label)+1),
             df_pca.loc[df_pca['cluster']==label,['pca_0','pca_1']].mean(),
             horizontalalignment='center',
             verticalalignment='center',
             size=20, weight='bold',
             color='white',
             backgroundcolor='#000000')

    #mark anomalies
    plt.scatter(x=df_pca.loc[df_pca['is_anomaly']==1, 'pca_0'],
                y=df_pca.loc[df_pca['is_anomaly']==1,'pca_1'],
                color='#FF0000',
                alpha=1)
```



Training and Testing Evaluation

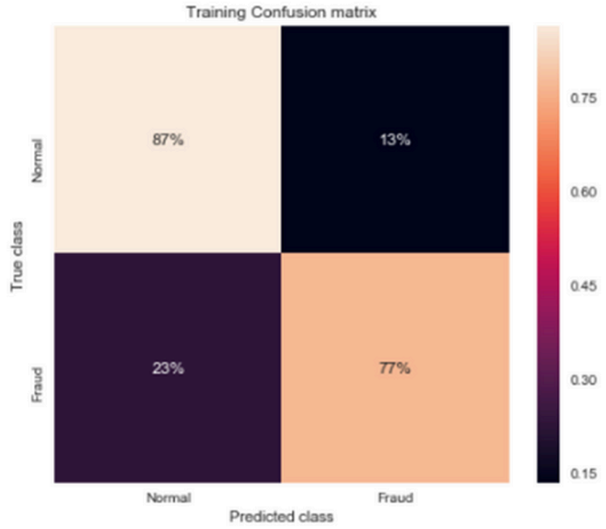
```
For K-means (K=5)
Balanced Accuracy: 0.8167213
Macro Precision: 0.5000092
Macro Recall: 0.8167213
Macro F1: 0.4638692

Normal Accuracy: 0.8651496
Normal Precision: 1.93e-05
Normal Recall: 0.7682927
Normal F1: 3.86e-05
```

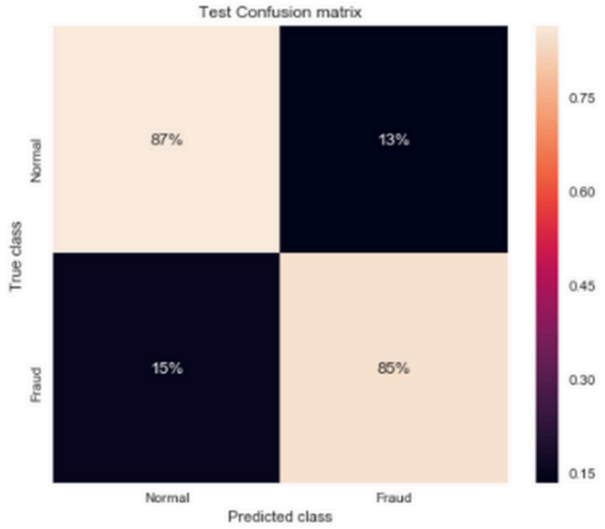
	precision	recall	f1-score	support
0	1.00	0.87	0.93	24198425
1	0.00	0.77	0.00	82
micro avg	0.87	0.87	0.87	24198507
macro avg	0.50	0.82	0.46	24198507
weighted avg	1.00	0.87	0.93	24198507

```
For K-means (K=5)
Balanced Accuracy: 0.8557088
Macro Precision: 0.5000131
Macro Recall: 0.8557088
Macro F1: 0.4639096

Normal Accuracy: 0.8652636
Normal Precision: 2.7e-05
Normal Recall: 0.8461538
Normal F1: 5.4e-05
```

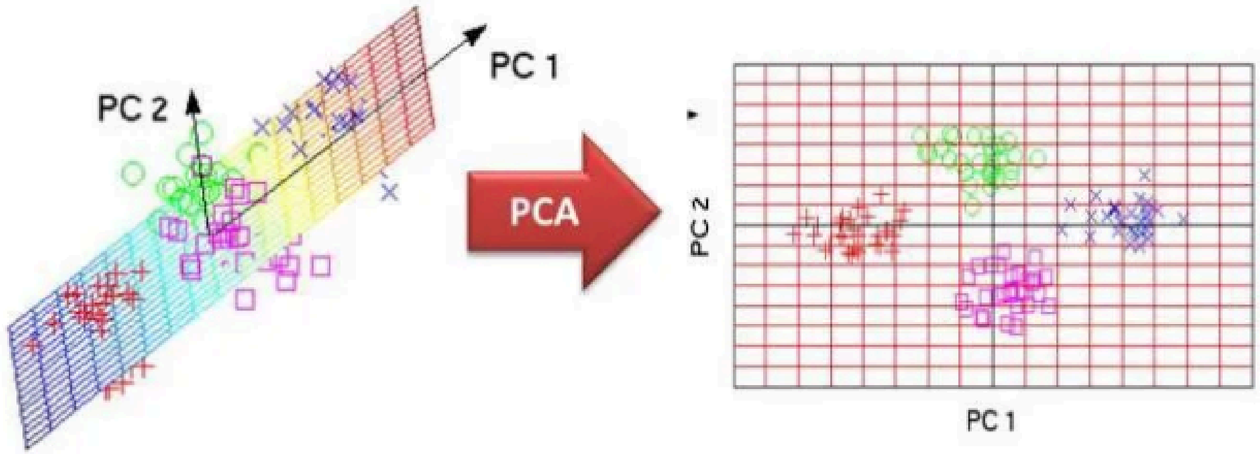
	precision	recall	f1-score	support
0	1.00	0.87	0.93	6049601
1	0.00	0.85	0.00	26
micro avg	0.87	0.87	0.87	6049627
macro avg	0.50	0.86	0.46	6049627
weighted avg	1.00	0.87	0.93	6049627

Confusion Matrix for training and testing





Principal Component Analysis(PCA)



- Principal Component Analysis (PCA) is a statistical technique commonly used for dimensionality reduction, but it can also be applied to anomaly detection in blockchain data, especially in scenarios where the data is unlabelled, meaning there's no prior information about which transactions are normal or anomalous.

Principal Component Analysis(PCA)

- In the context of blockchain anomaly detection, PCA works by transforming the original features into a new set of orthogonal components called principal components. These principal components are ordered by the amount of variance they explain in the data, with the first principal component explaining the most variance, the second explaining the second most, and so on.

- The intuition behind using PCA for anomaly detection is that normal transactions in a blockchain dataset should exhibit some patterns or regularities, and anomalies would deviate from these patterns.

- By reducing the dimensionality of the data through PCA, we can focus on the principal components that capture the most significant variations in the data, potentially highlighting anomalies as outliers in this reduced-dimensional space.

Principal Component Analysis(PCA)

In [6]:
```
# Log scale to normalize the data
scaled_df = df.copy()

scaled_df['indegree'] = np.log1p(scaled_df['indegree'])
scaled_df['outdegree']= np.log1p(scaled_df['outdegree'])
scaled_df['in_btc'] = np.log1p(scaled_df['in_btc'])
scaled_df['out_btc']= np.log1p(scaled_df['out_btc'])
scaled_df['total_btc']= np.log1p(scaled_df['total_btc'])
scaled_df['mean_in_btc']= np.log1p(scaled_df['mean_in_btc'])
scaled_df['mean_out_btc']= np.log1p(scaled_df['mean_out_btc'])

# RobustScaler is less prone to outliers.
from sklearn.preprocessing import RobustScaler
rob_scaler = RobustScaler()

scaled_df['indegree'] = rob_scaler.fit_transform(scaled_df['indegree'].values.reshape(-1,1))
scaled_df['outdegree'] = rob_scaler.fit_transform(scaled_df['outdegree'].values.reshape(-1,1))
scaled_df['in_btc'] = rob_scaler.fit_transform(scaled_df['in_btc'].values.reshape(-1,1))
scaled_df['out_btc'] = rob_scaler.fit_transform(scaled_df['out_btc'].values.reshape(-1,1))
scaled_df['total_btc'] = rob_scaler.fit_transform(scaled_df['total_btc'].values.reshape(-1,1))
scaled_df['mean_in_btc'] = rob_scaler.fit_transform(scaled_df['mean_in_btc'].values.reshape(-1,1))
scaled_df['mean_out_btc'] = rob_scaler.fit_transform(scaled_df['mean_out_btc'].values.reshape(-1,1))
```

We preprocessed the data by performing log transformation and robust scaling on columns, which are common techniques used to prepare numerical data when dealing with skewed distributions and outliers.

Principal Component Analysis(PCA)

- Then we have split the preprocessed data into training and testing sets for machine learning model training and evaluation.

Get Training Sample

In [12]:
```
from sklearn.utils import shuffle

original_train_df = pd.concat([original_Xtrain, original_ytrain], axis=1)
original_train_normal_df = original_train_df[original_train_df.out_and_tx_malicious == 0]
original_train_fraud_df = original_train_df[original_train_df.out_and_tx_malicious == 1]

def get_training_sample(n):
    normal_sample = original_train_normal_df.sample(n)
    final_sample = pd.concat([normal_sample, original_train_fraud_df])
    final_sample = shuffle(final_sample)

    return final_sample.iloc[:,:-1].values, final_sample.iloc[:,-1].values
```

We have done this code to address class imbalance by creating a balanced training dataset with an equal number of samples from both normal and fraudulent transactions. This can help prevent the model from being biased towards the majority class and improve its performance on detecting fraudulent transactions

Principal Component Analysis(PCA)

Get Test Sample

In [13]:
```
original_test_df = pd.concat([original_Xtest, original_ytest], axis=1)
original_test_normal_df = original_test_df[original_test_df.out_and_tx_malicious == 0]
original_test_fraud_df = original_test_df[original_test_df.out_and_tx_malicious == 1]

def get_test_sample(n):
    normal_sample = original_test_normal_df.sample(n)
    final_sample = pd.concat([normal_sample, original_test_fraud_df])
    final_sample = shuffle(final_sample)
    return final_sample.iloc[:,:-1].values, final_sample.iloc[:,-1].values
```

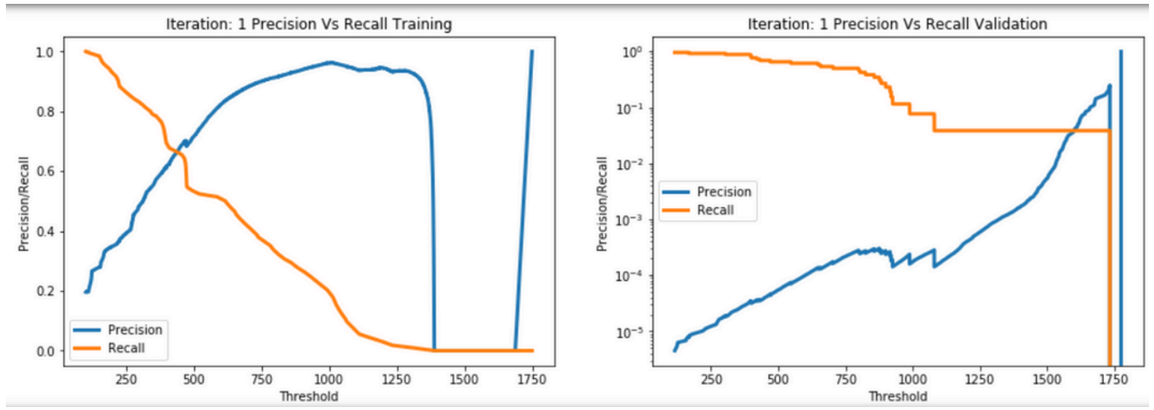
This code is to create a balanced test dataset with an equal number of samples from both normal and fraudulent transactions, facilitating a fair evaluation of the model's performance on both classes.

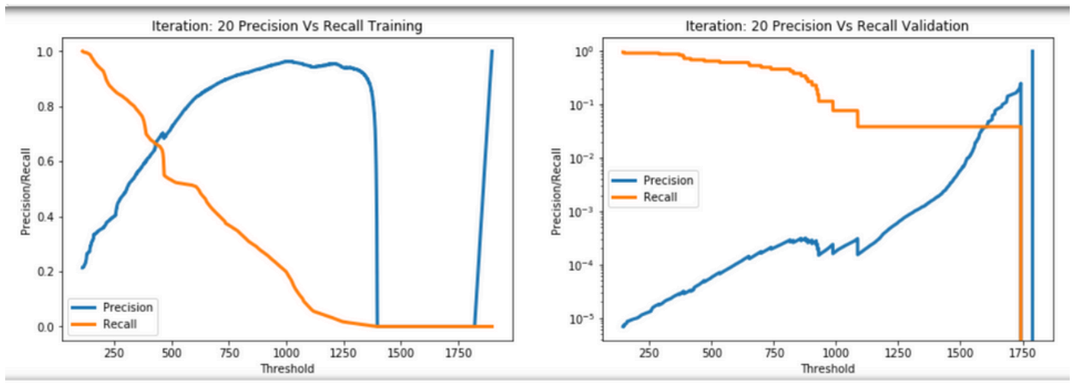
We also done oversampling of the minority class using the SMOTE algorithm, which can help address class imbalance issues in binary classification tasks.

Principal Component Analysis(PCA)

- We have done evaluation metrics for assessing the performance of a binary classification model. These metrics capture different aspects of model performance, including balanced accuracy, precision, recall.
- Then we trained a PCA model for outlier detection using the PyOD library.
- We have done evaluation metrics obtained during model training and testing into DataFrames. Each row in the DataFrames corresponds to a specific iteration, and each column represents a different evaluation metric.

Precision vs Recall

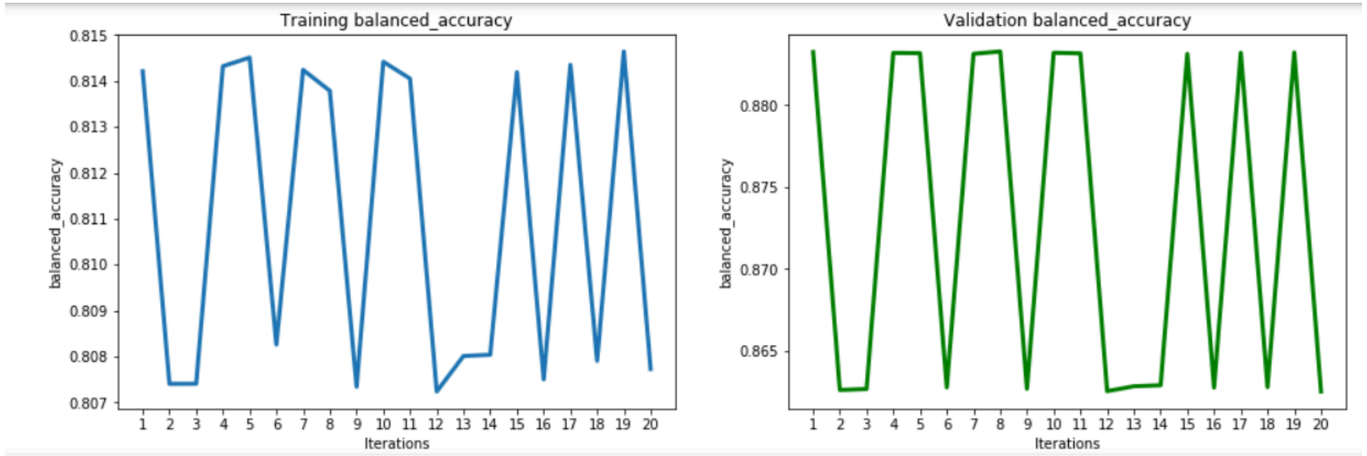




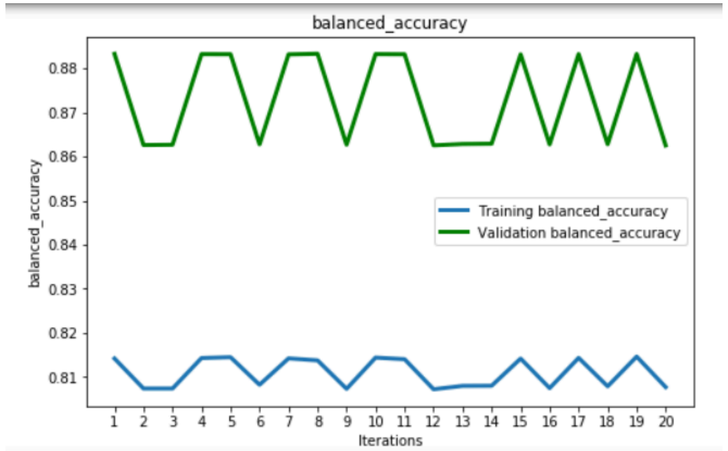
Principal Component Analysis(PCA)

- In Iteration 1, the Precision and Recall curves for validation show more fluctuations, indicating potential overfitting or instability.
- In Iteration 20, the curves are more stable, with fewer fluctuations, indicating better generalization and less overfitting.

Balanced Accuracy



Principal Component Analysis(PCA)



The first set indicates higher variability and instability, while the second set demonstrates improved stability and consistency, particularly in the training process. This comparison suggests that the model training has potentially improved, leading to more reliable performance metrics.

Autoencoder

Autoencoders, under the unsupervised wing of neural networks, enable the learning of efficient codings. The autoencoder learns to encode data into an n-dimensional representation and then reconstructs the original data. This feature makes autoencoders useful to perform anomaly detection as it infers deviations from normal data patterns. The destructive points in an extremely imbalanced dataset are identified using a deep autoencoder in this study.

```
Training features shape: (24198507, 7)
Testing features shape: (6049627, 7)
Training labels shape: (24198507,)
Testing labels shape: (6049627,)
```

Information of the Split Dattaset for modelling

Implementation of Autoencoder Model

Data Loading and Preprocessing

The notebook starts by importing necessary libraries such as **numpy**, **pandas**, **sklearn**, and **keras**. The dataset is then loaded into a pandas DataFrame.

Data Splitting

The dataset is divided into two types of sets: the training and testing sets, so that the model can further evaluate the unseen data. Artificial examples of the training set from the minority class (malicious data points) are generated so as to overcome the class imbalance.

Building the Autoencoder Model

The autoencoder model is defined using the Keras library. The model architecture consists of an encoder and a decoder:

- Encoder: Four dense layers with 7, 7, 6, and 4 neurons respectively, all using the tanh activation function.
- Decoder: Three dense layers with 4, 6, and 7 neurons respectively, using the tanh activation function for the first two layers and sigmoid for the last layer.

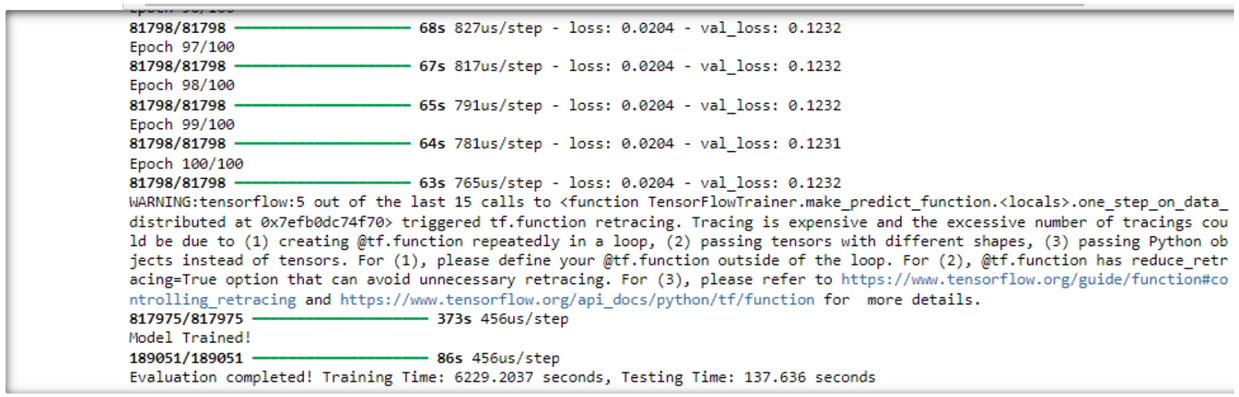
Autoencoder model implementation contd...

Train the Model Autoencoder

Compile the model using MSLE loss as the loss function and the Adam optimizer. Model training is set for 100 epochs using a batch size of 256. The model learns how to encode the input data into a lower dimension and decodes it back to the original format.

Model Evaluation

Some of the key following metrics are checked to evaluate model performance during training: balanced accuracy, macro-precision, macro-recall, macro F1 score, area under the receiver operating characteristic curve, and others. The same metrics are computed on training and test sets to confirm the capability of the model in generalizing new data.



Epoch's used in modelling for better accuracy

	Iteration	Accuracy	Balanced-Accuracy	Macro-Precision	Macro-Recall
0	1	0.921746	0.738544	0.72201	0.738544

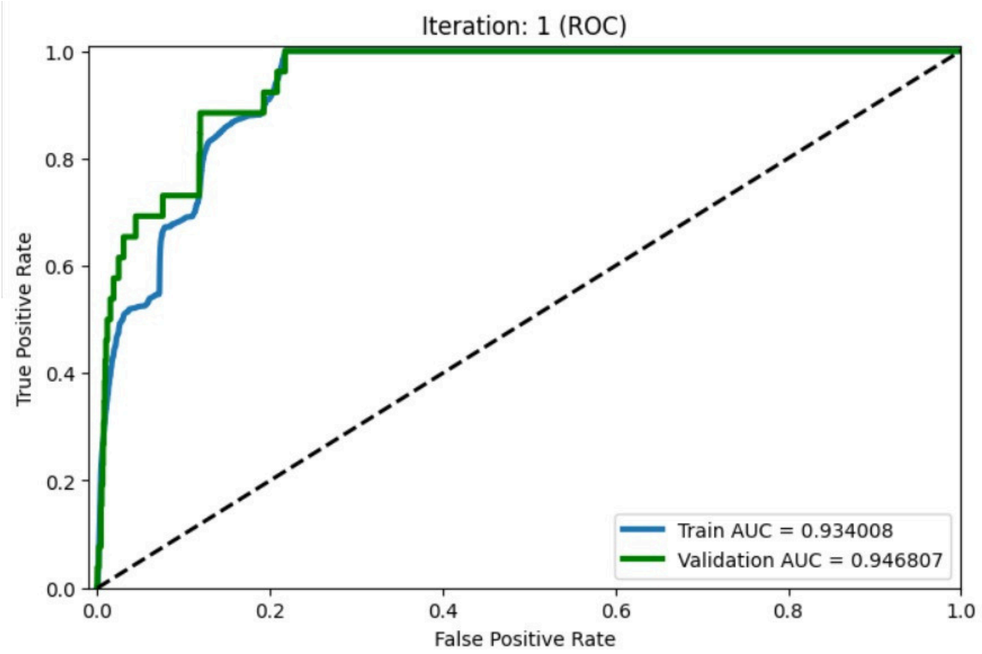
	Macro-F1	Macro-ROC	Precision	Recall	F1	ROC	Time
0	0.729884	0.934008	0.483269	0.522748	0.502234	0.934008	6229.2037

Evaluation for Training and Testing Datasets

	Iteration	Accuracy	Balanced-Accuracy	Macro-Precision	Macro-Recall
0	1	0.918973	0.824871	0.500019	0.824871

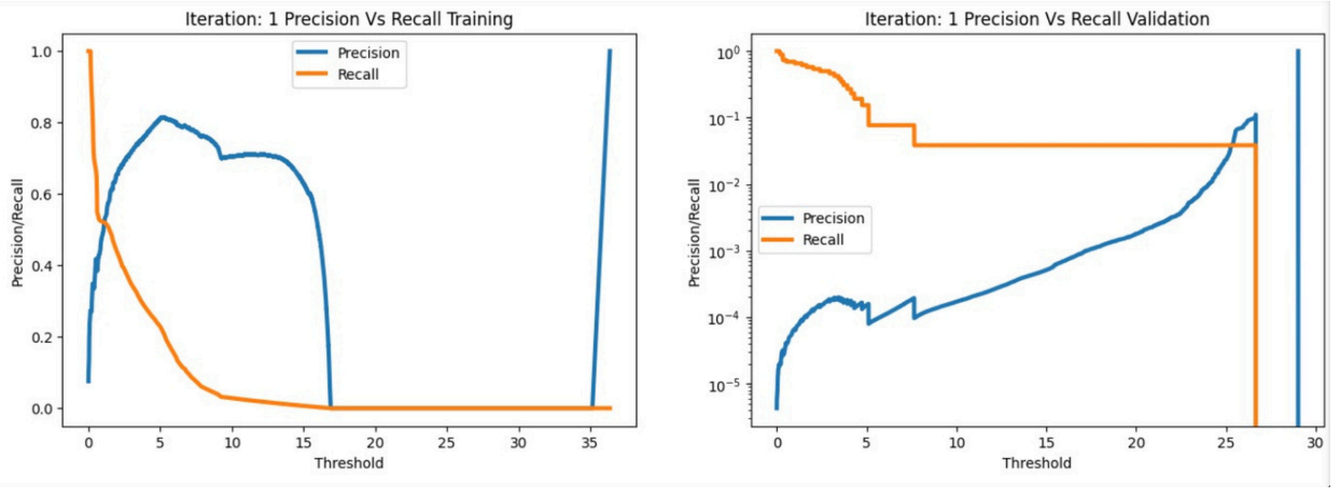
	Macro-F1	Macro-ROC	Precision	Recall	F1	ROC	Time
0	0.478927	0.946807	0.000039	0.730769	0.000077	0.946807	137.636

ROC Curve



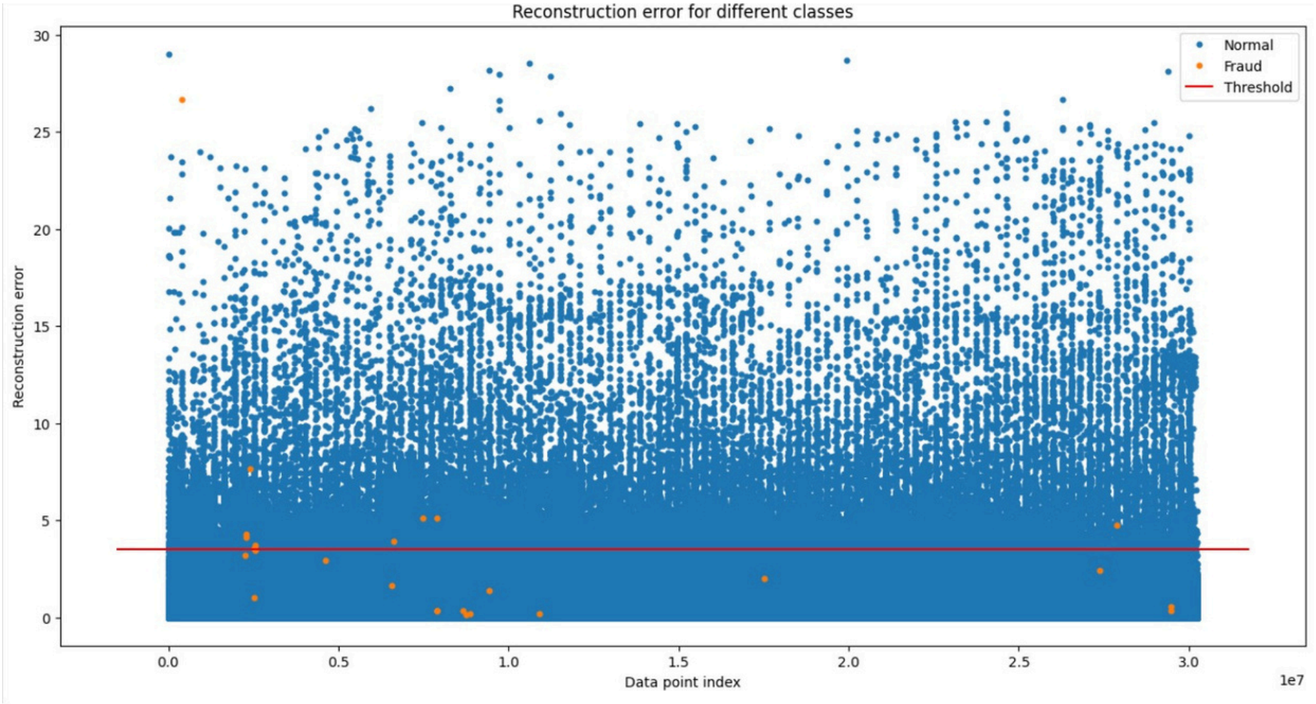
a higher AUC score for a model usually represents a better model. Recall of the autoencoder model shows a satisfying result as recall refers to the percentage of total relevant results correctly classifed by the algorithm.

Precision VS Recall



the trade-off between precision and recall of the model at different thresholds for training and test data.
A value of threshold slightly below 4 seems reasonably satisfactory.
threshold value slightly below
4 seems to cater to a signicant number of malicious data points while not misclassifying majority of non-malicious data points.

Reconstruction error



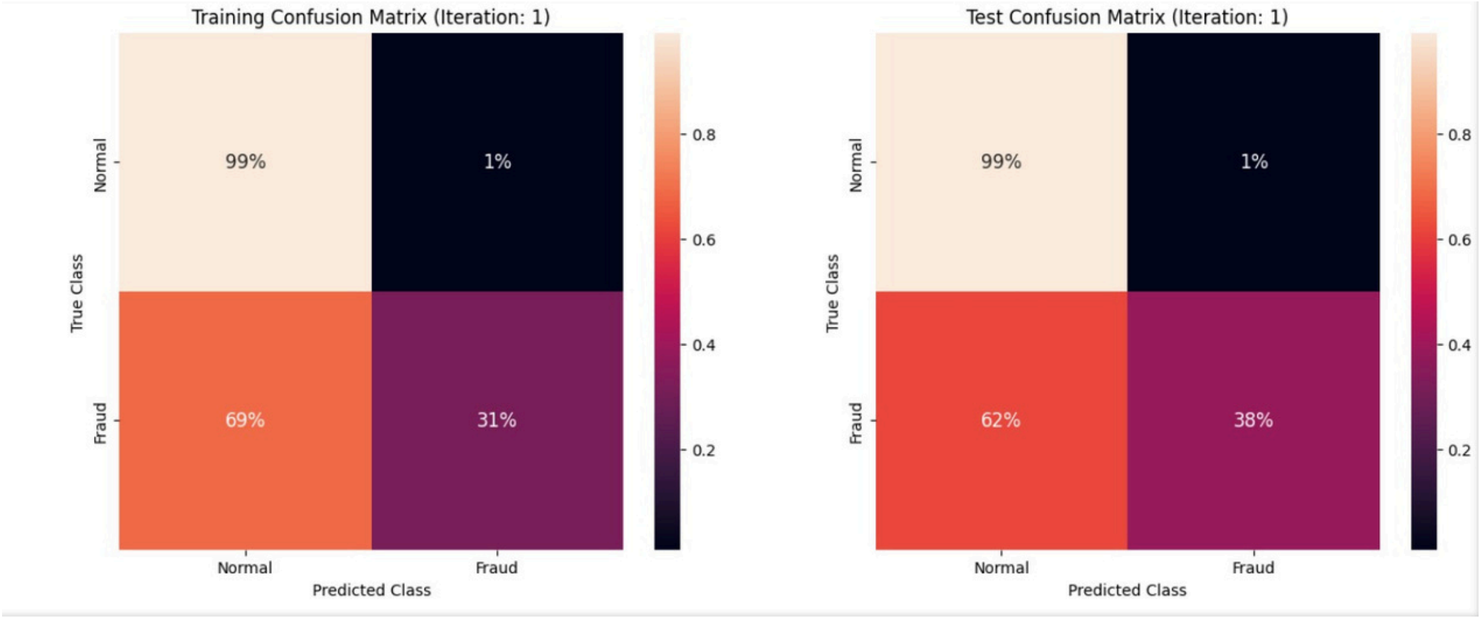
Visual representation of reconstruction error of the test set
from the model predicition to predict the malicious transactions.

The reconstruction plot for the error shows most normal points have been well reconstructed by the autoencoder: That is to say, they have small values for the reconstruction error, and all the points lie quite far below the threshold line. As such, since most of the data points representing fraud are indicatory of higher values of the reconstruction error and spread above the threshold, it is able to effectively show them as being anomalies. The red threshold line separates normal and fraudulent data.

Confusion Matrix

The model built has been able to detect 38% (10 out of 16) of the malicious data points in test data and 31% (617,804 out of 1,358,965) of them in the training data.

The model has classified approximately 69% of the cases in the training data and 62% (10 out of 16) in the test data as false-positive cases. In contrast, the true positive has been rated at a higher form 99% (Training: 23,986,576 out of 211,849 and Test: 5,996,738 out of 52,863) used in the test and training data 23,986,576 of the normal transaction out of 23,988,425 were properly classified, while for the test data, 5,996,738 of the normal transaction out of 5,996,754 were properly identified.



Conclusion

- In conclusion, the Blockchain Anomaly Detection project successfully applied sophisticated data preprocessing techniques and advanced machine learning models to enhance the security framework of blockchain transactions. Through meticulous exploratory data analysis, the project effectively prepared and normalized the transaction data, setting a robust foundation for model training.

- The employment of various unsupervised machine learning models, such as Isolation Forest, Autoencoder, CBLOF, PCA, K-means demonstrated significant capabilities in identifying anomalous transactions that could potentially indicate fraudulent activity.
- Autoencoder outperforms from the rest of the models followed by ensemble model. Iforest is the best fit to detect the anamolies.

- This project not only advances the technological measures in blockchain security but also provides a valuable contribution to ongoing research in the field.

Future Scope

Integration with Additional Blockchains:

While your current focus might be on Bitcoin transactions, expanding the system to monitor and analyze transactions across various blockchains (such as Ethereum, Ripple, etc.) can broaden its utility and impact.

Real-Time Monitoring and Response:

Developing capabilities for real-time anomaly detection and automatic response mechanisms can significantly increase the system's effectiveness in preventing fraudulent activities as they happen.

Advanced Machine Learning Models:

Incorporating more advanced machine learning and deep learning techniques could improve the accuracy and efficiency of anomaly detection. Techniques like neural networks or reinforcement learning might be explored for this purpose.

User Behavior Analysis:

Extending the system to analyze user behavior patterns over time to predict and detect anomalous behaviors before they result in actual transactions could preemptively enhance security measures.

Thank You