

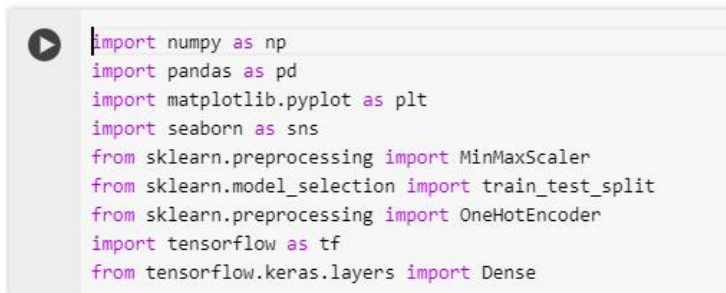
## Accredian Assignment

### Info about the dataset:

- **step** - maps a unit of time in the real world. In this case 1 step is 1 hour of time. Total steps 744 (30 days simulation).
  - **type** - CASH-IN, CASH-OUT, DEBIT, PAYMENT and TRANSFER.
  - **amount** - amount of the transaction in local currency.
  - **nameOrig** - customer who started the transaction
  - **oldbalanceOrg** - initial balance before the transaction
  - **newbalanceOrig** - new balance after the transaction
  - **nameDest** - customer who is the recipient of the transaction
  - **oldbalanceDest** - initial balance recipient before the transaction. Note that there is not information for customers that start with M (Merchants).
  - **newbalanceDest** - new balance recipient after the transaction. Note that there is no information for customers that start with M (Merchants).
  - **isFraud** - This is the transactions made by the fraudulent agents inside the simulation. In this specific dataset the fraudulent behavior of the agents aims to profit by taking control or customers accounts and try to empty the funds by transferring to another account and then cashing out of the system.
  - **isFlaggedFraud** - The business model aims to control massive transfers from one account to another and flags illegal attempts. An illegal attempt in this dataset is an attempt to transfer more than 200.000 in a single transaction.
- 

### Info about the colab file:

The notebook starts by importing the necessary libraries, which include:



```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
import tensorflow as tf
from tensorflow.keras.layers import Dense
```

The code snippet imports several libraries commonly used for data analysis and machine learning, including NumPy, Pandas, Matplotlib, Seaborn, Scikit-learn, TensorFlow, and Keras.

Here's a more detailed breakdown of the code:

- The first lines import libraries like NumPy (numpy), Pandas (pandas), Matplotlib (matplotlib.pyplot as plt), and Seaborn (seaborn as sns). These libraries are commonly used for data manipulation and visualization.
  - The next lines import functionalities from scikit-learn for data preprocessing, including MinMaxScaler for normalization and train\_test\_split to split data into training and testing sets.
  - Then, it imports OneHotEncoder from scikit-learn for data preprocessing, likely to encode categorical data.
  - TensorFlow (tensorflow as tf) library is imported, a popular library for machine learning.
  - Finally, from TensorFlow's Keras API, a dense layer (Dense) is imported, likely to be used for building a neural network model.
-

```
df = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/Fraud.csv")
```

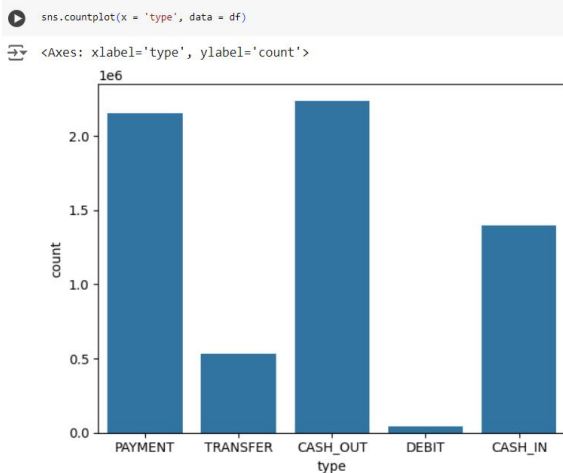
```
df.head(10)
```

	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
0	1	PAYMENT	9839.64	C1231006815	170136.00	160296.36	M1979787155	0.0	0.00	0	0
1	1	PAYMENT	1864.28	C1666544295	21249.00	19384.72	M2044282225	0.0	0.00	0	0
2	1	TRANSFER	181.00	C1305486145	181.00	0.00	C553264065	0.0	0.00	1	0
3	1	CASH_OUT	181.00	C840083671	181.00	0.00	C38997010	21182.0	0.00	1	0
4	1	PAYMENT	11668.14	C2048537720	41554.00	29885.86	M1230701703	0.0	0.00	0	0
5	1	PAYMENT	7817.71	C90045638	53860.00	46042.29	M573487274	0.0	0.00	0	0
6	1	PAYMENT	7107.77	C154988899	183195.00	176087.23	M408069119	0.0	0.00	0	0
7	1	PAYMENT	7861.64	C1912850431	176087.23	168225.59	M633326333	0.0	0.00	0	0
8	1	PAYMENT	4024.36	C1265012928	2671.00	0.00	M1176932104	0.0	0.00	0	0
9	1	DEBIT	5337.77	C712410124	41720.00	36382.23	C195600860	41898.0	40348.79	0	0

- The code snippet describes the process of loading the data or the dataset on google colab for which pandas, which is one of the famous machine learning libraries is used for loading the dataset and reading the dataset. First 10 rows of the dataset are displayed using the head() function.

```
df['newbalanceDest'] = df['newbalanceDest'].fillna(method = 'ffill')
df['isFraud'] = df['isFraud'].fillna(method = 'ffill')
df['isFlaggedFraud'] = df['isFlaggedFraud'].fillna(method = 'ffill')
```

- This code snippet describes the handling of the missing values in the dataset : line of code is imputing missing values in the column 'newbalanceDest' of the DataFrame 'df' using forward fill method.
- df['newbalanceDest']: This selects the 'newbalanceDest' column from the DataFrame df.fillna(method='ffill'): This function fills missing values (NaN) in the selected column 'newbalanceDest' using the 'ffill' method. The 'ffill' method replaces missing values with the value from the previous row.



This code snippet is used for plotting this bar graph

The bars on the graph represent different transaction types and the height of each bar shows the number of transactions of that type. Where 'CASH\_OUT' is the one which is used for maximum number of times, after that 'PAYMENT', 'CASH\_IN', 'TRANSFER' and lastly 'DEBIT'.

```
[ ] df['isFraud'].value_counts()
```

```
isFraud
0      6354407
1        8213
Name: count, dtype: int64
```

```
[ ] df['isFlaggedFraud'].value_counts()
```

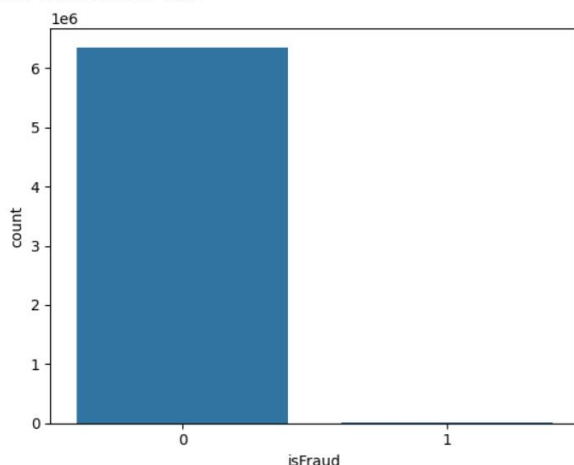
```
isFlaggedFraud
0      6362604
1         16
Name: count, dtype: int64
```

This code snippet describes the total number of transactions which are fraud (indicated by: 1) and which are safe or legit (indicated by: 0), the second code cell indicates the total transaction which is flagged fraud means that the fraud detection software has flagged them as fraud or not fraud.

---

```
sns.countplot(x = 'isFraud', data = df)
print("1.0: Transaction is Fraud \n0.0: Transaction is legit")
```

```
1.0: Transaction is Fraud
0.0: Transaction is legit
```



The code snippet is for creating the visualization that shows a bar plot created using the Seaborn library. The plot visualizes the distribution of fraudulent and legitimate transactions in a dataset. Here is a detailed description of the image and its implications:

### **Plot Type: Count plot (Bar plot)**

#### **X-axis: isFraud**

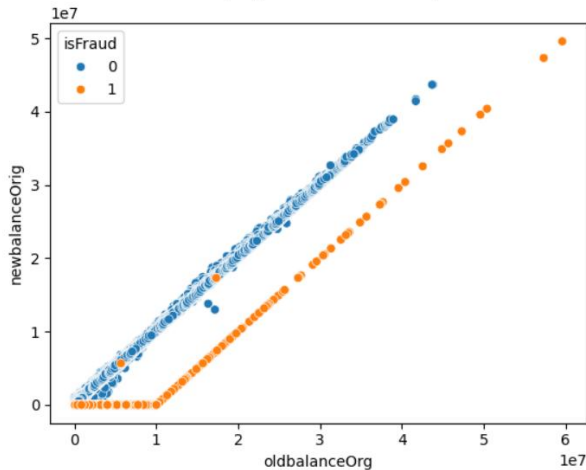
- This axis represents the binary classification of transactions.
- 0 indicates a legitimate transaction.
- 1 indicates a fraudulent transaction.

#### **Y-axis: count**

- This axis represents the number of transactions for each category (isFraud).
-

```
sns.scatterplot(x = 'oldbalanceOrig', y = 'newbalanceOrig', data = df, hue = 'isFraud')
```

```
<Axes: xlabel='oldbalanceOrig', ylabel='newbalanceOrig'>
```



This snippet shows a scatter plot generated using Seaborn's scatterplot function. This plot visualizes the relationship between the old balance and the new balance of the origin account, with a distinction made between fraudulent and legitimate transactions.

#### Description of the Plot:

- **X-axis (oldbalanceOrig):** Represents the old balance of the origin account before the transaction.
- **Y-axis (newbalanceOrig):** Represents the new balance of the origin account after the transaction.

**Hue (isFraud):** This differentiates between fraudulent (1) and legitimate (0) transactions using color:

- **0 (blue dots):** Legitimate transactions.
- **1 (orange dots):** Fraudulent transactions.

#### Insights:

##### Legitimate Transactions (Blue Dots):

Most legitimate transactions lie along the line where newbalanceOrig is slightly less than oldbalanceOrig, indicating a reduction in balance post-transaction. There is a clear linear pattern suggesting that the new balance is generally consistent with the transaction amount deducted from the old balance.

##### Fraudulent Transactions (Orange Dots):

Fraudulent transactions form a distinct pattern where the new balance is often zero or significantly lower than the original balance, indicating an unusual depletion of funds. There is a noticeable clustering of fraudulent transactions at lower values of newbalanceOrig, suggesting that these transactions often result in the account balance being drained or reduced to very low amounts.

```
[ ] new_type = pd.get_dummies(df['type'], dtype = int)
```

```
[ ] df1 = pd.DataFrame(new_type)
```

```
[ ] df = pd.concat([df,df1], axis = 1)
```

This code snippet deals with the processing of vectorization of string values given in the 'type' attribute of the data frame. After vectorizing the values and converting the values into the data frame this new data frame is concatenated with the older or the original data frame. For concatenation concat() of pandas libraries is used. For vectorization get\_dummies() function of the pandas library is used which converts the string value or words into binary format (either 0 or 1). Vectorized values are converted to data frame by DataFrame() function of the pandas library.

```
X = df.drop(['isFraud', 'nameOrig', 'nameDest'], axis = 1)
y = df[['isFraud']]
```

**The predictor variable selected are:**

step    amount    oldbalanceOrg    newbalanceOrig

oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud	CASH_IN	CASH_OUT	DEBIT	PAYMENT	TRANSFER
----------------	----------------	---------	----------------	---------	----------	-------	---------	----------

**Reason for selecting these is:** as we are trying to predict whether the transaction is fraud or not so all the values which are related to amount, balance, or money are best to predict whether the transactions are fraud or not. The attributes 'nameOrig', 'nameDest' are not so much important because they are just the names of the customers starting and ending the transaction respectively hence not that much import to predict the fraudulent.

**Target variable is:** 'isFraud' which will predict and classify the transaction as fraudulent or not.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
```

This code snippet splits the dataset into training and testing data  
The description of the training, testing and validation data is:

### 1. Training Data

The training data is used to train the machine learning model. The model learns the patterns and relationships within the data during the training phase.

- **Purpose:** To fit the model and determine the parameters (weights in the case of algorithms like logistic regression, or splits in the case of decision trees).
- **Usage:** The model is repeatedly exposed to this data to learn the underlying patterns.

### 2. Validation Data

The validation data is used to tune the model and select the best parameters. It helps in evaluating the model performance during the training process without overfitting to the training data.

- **Purpose:** To validate the model's performance and fine-tune the model's hyperparameters (e.g., learning rate, number of trees in a forest, etc.).
- **Usage:** The validation set is used during the model training phase to periodically check the model's performance and make adjustments. Techniques like k-fold cross-validation can be used where the training data is split multiple times to create different validation sets.

### 3. Testing Data

The testing data is used to evaluate the final performance of the model after it has been trained and validated. This set has not been used during the model building or validation process.

- **Purpose:** To provide an unbiased evaluation of the final model.
- **Usage:** After the model has been trained and validated, the testing data is used to assess how well the model generalizes to new, unseen data. The performance metrics obtained from this set give an indication of the model's effectiveness.

```

▶ scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.fit_transform(X_test)

```

The code snippet you sent is related to data preprocessing in Python using the scikit-learn library. The specific part uses MinMaxScaler to scale features in a dataset. Here's a breakdown of the code:

- 1. Import:** The code assumes MinMaxScaler is already imported from scikit-learn library.
- 2. Scaler Object:** A MinMaxScaler object is created and assigned to the variable scaler. This object helps scale features in a dataset to a specific range. By default, it scales features to be between 0 and 1.
- 3. Train Data Scaling:** The code fits the scaler to the training data (X\_train) using the fit\_transform method. This method calculates the minimum and maximum values in each feature of the training data, and it uses those values to transform each sample in the training data to a new range. The scaled training data is stored in the variable X\_train\_scaled.
- 4. Test Data Scaling:** The code then uses the already fitted scaler (scaler) to transform the test data (X\_test) using the transform method. This method applies the same scaling parameters (minimum and maximum values) learned from the training data to transform the test data to the same range. The scaled test data is stored in the variable X\_test\_scaled.

```

▶ tf.random.set_seed(42)
classifier_model = tf.keras.models.Sequential()
classifier_model.add(tf.keras.layers.Dense(units = 100, activation = 'relu', input_shape = (12,)))
classifier_model.add(tf.keras.layers.Dropout(0.3))
classifier_model.add(tf.keras.layers.Dense(units = 50, activation = 'relu'))
classifier_model.add(tf.keras.layers.Dropout(0.3))
classifier_model.add(tf.keras.layers.Dense(units = 50, activation = 'relu'))
classifier_model.add(tf.keras.layers.Dense(units = 1, activation = 'sigmoid'))

[ ] classifier_model.compile(optimizer = 'SGD', loss = 'binary_crossentropy', metrics = 'accuracy')

[ ] #calculating the epochs on training and validation dataset
from tensorflow.keras.callbacks import EarlyStopping
Early_Stop = EarlyStopping()
epochs_hist = classifier_model.fit(X_train_scaled,y_train,epochs = 5, validation_data = (X_test_scaled, y_test), batch_size = 125, callbacks=[Early_Stop])

```

#### • Setting Random Seed:

- The first line `tf.random.set_seed(42)` sets a random seed for TensorFlow. This helps ensure reproducibility of the results by making sure the model weights are initialized the same way each time the code is run.

#### • Model Initialization:

- The second line `classifier_model = tf.keras.models.Sequential()` creates a sequential model object, which is a linear stack of layers.

#### • Adding Layers:

- The next lines starting from `classifier_model.add(...)` define the layers of the neural network:
  - The first layer is a Dense layer with 100 units and a 'relu' activation function. The `input_shape` argument specifies that the input data will have 12 features.
  - A Dropout layer with a rate of 0.3 is added after the first hidden layer. Dropout helps prevent overfitting by randomly dropping out a certain percentage of neurons during training.
  - Two more hidden layers are added, each with 50 units and a 'relu' activation function. Another dropout layer with a rate of 0.3 is added after the second hidden layer.
  - The final layer is a Dense layer with 1 unit and a 'sigmoid' activation function. Since it's a binary classification task (classifying between two classes), the output layer has one unit. The 'sigmoid' activation function typically squashes the output values between 0 and 1, making it suitable for binary classification tasks.

#### • Compiling the Model:

- The last two lines compile the model:

`classifier_model.compile(optimizer='SGD', loss='binary_crossentropy', metrics=['accuracy'])` configures the learning process.

- `optimizer='SGD'` sets the optimizer to be Stochastic Gradient Descent (SGD), a common algorithm for training neural networks.
  - `loss='binary_crossentropy'` specifies the loss function as binary crossentropy, which is appropriate for binary classification tasks.
  - `metrics=['accuracy']` tells the model to track the accuracy metric during training.
- 

## **Reasons for using the neural network for the fraud classification:**

### **1. Ability to Handle Complex Patterns and Non-Linearity**

- **Deep Learning Capabilities:** Neural networks, especially deep neural networks, can capture complex, non-linear relationships in data. Fraudulent activities often involve intricate patterns that are not easily captured by traditional linear models.
- **Feature Interactions:** Neural networks can automatically learn and represent feature interactions without explicit manual feature engineering, which is crucial for detecting subtle fraudulent behaviors.

### **2. Scalability with Large Datasets**

- **Handling Large Volumes of Data:** Neural networks are well-suited for training on large datasets, which is common in fraud detection scenarios involving millions of transactions.
- **Parallel Processing:** With modern hardware accelerators like GPUs and TPUs, neural networks can efficiently process and learn from vast amounts of data, providing scalability and speed.

### **3. Flexibility and Adaptability**

- **Model Flexibility:** Neural networks can be tailored for various types of data, whether structured, unstructured, or a mix of both. This flexibility allows for integrating additional data sources like text or images, enhancing the fraud detection capabilities.
- **Continuous Learning:** Neural networks can be updated and retrained continuously as new fraud patterns emerge, making them adaptable to evolving fraud tactics.

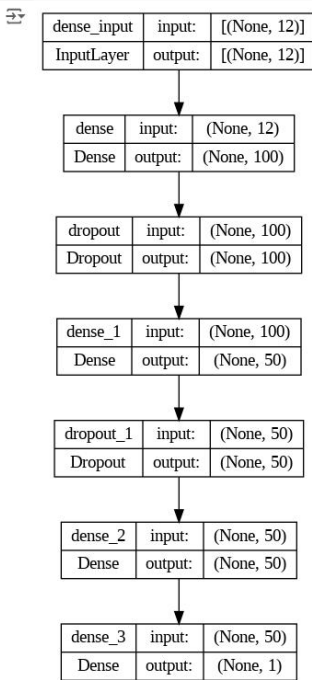
### **4. Robustness to Noise and Irregularities**

- **Resilience to Data Noise:** Neural networks, especially those with regularization techniques like dropout, can be robust against noisy data and outliers, which are common in real-world fraud detection.
- **Feature Hierarchies:** By learning hierarchical feature representations, neural networks can discern important features from noisy data, improving detection accuracy.

### **5. Improved Detection Performance**

- **High Accuracy:** Neural networks have demonstrated superior performance in various domains, including fraud detection, often achieving higher accuracy and recall rates compared to traditional machine learning models.
  - **Advanced Architectures:** Architectures like Convolutional Neural Networks (CNN) for pattern recognition, Recurrent Neural Networks (RNNs) for sequential data, and their hybrids can provide state-of-the-art performance in detecting fraud.
-

```
import keras
keras.utils.plot_model(classifier_model, show_shapes=True)
```



### Code Analysis:

The Python code defines a sequential neural network model using Keras. Here's a breakdown of the code:

### Setting Random Seed:

The first line sets a random seed to ensure reproducibility of the results.

### Model Initialization:

A sequential model object is created.

### Adding Layers:

- The code defines the layers of the neural network:
- A dense layer with 100 units and a relu activation is added. This layer takes the input data with 12 features and transforms it into 100 internal units using a rectified linear activation function.
- A dropout layer with a rate of 0.3 is added to prevent overfitting.
- Two more hidden layers are added, each with 50 units and a relu activation function, followed by dropout layers with a rate of 0.3.
- The final layer has one unit and a sigmoid activation function. Since it's a binary classification task, the output layer has one unit. The sigmoid function maps the output between 0 and 1.

### Compiling the Model:

The model is compiled using SGD optimizer, binary cross-entropy loss (as there are only two classes 0 and 1), and accuracy metric.

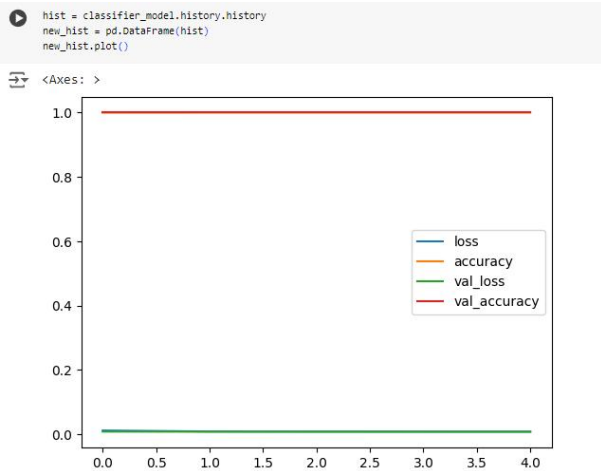
### Visualization Analysis:

The visualization complements the information from the code. It shows a flowchart-like representation of the neural network architecture:

- **Input Layer:** The model starts with an input layer represented by a box labeled "dense\_input". This layer takes data with 12 features, which aligns with the input\_shape argument in the code.



- **Hidden Layers:** There are three hidden layers, each represented by a rectangular box. The first two hidden layers have 100 and 50 units respectively, as specified in the code with Dense layers. The third and fourth layers (counting the final layer) also have 50 units each.
- **Dropout Layers:** The visualization includes dropout layers symbolized by hexagonal boxes placed after the first, second, and third hidden layers. These layers match the dropout rates (0.3) specified in the code.
- **Output Layer:** The final layer is a dense layer with one unit and likely a sigmoid activation function (not explicitly shown in the visualization but implied based on the code).



- This graph is a line graph showing the accuracy and loss of a machine learning model over multiple epochs (training iterations). The code snippet is generating this visualization. The first line imports libraries commonly used for data analysis and visualization, including matplotlib.pyplot (plt) for plotting. The next lines likely define labels for the x and y axes of the graph.
- The following lines (new\_hist['accuracy']...) plot the training and validation accuracy (accuracy and val\_accuracy) on the y-axis against epochs on the x-axis. Similarly, the loss (loss and val\_loss) is plotted on the same graph.

```
[ ] #evaluating the model on the testing dataset.
evaluation = classifier_model.evaluate(X_test,y_test)
print('test accuracy: {}'.format(evaluation[1]))
```

```
39767/39767 [=====] - 66s 2ms/step - loss: 2931.5188 - accuracy: 0.9987
test accuracy:0.9987285137176514
```

```
[ ] y_test_predict = classifier_model.predict(X_test)
y_train_predict = classifier_model.predict(X_train)
y_test_predict = y_test_predict > 0.5
y_train_predict = y_train_predict > 0.5
```

```
39767/39767 [=====] - 63s 2ms/step
159066/159066 [=====] - 249s 2ms/step
```

- The first line evaluation = classifier\_model.evaluate(X\_test, y\_test) likely evaluates the performance of the classifier model (classifier\_model) on the testing data (X\_test and y\_test). The evaluate function likely returns a list of metrics, and the assignment stores these metrics in the variable evaluation.
- The second line print('test accuracy: {}'.format(evaluation[1])) prints the test accuracy. It retrieves the test accuracy from the second element ([1]) of the evaluation list and formats it into a string using the .format() method.
- The second code cell predict the data from the testing data and training data.

```
print("Training Report:\n", classification_report(y_train_predict, y_train))
print("Testing Report:\n", classification_report(y_test_predict, y_test))
```

```
Training Report:
              precision    recall  f1-score   support

   False         1.00        1.00        1.00    5090045
   True          0.00        0.00        0.00         51

 accuracy          1.00    5090096
 macro avg         0.50        0.50        0.50    5090096
 weighted avg       1.00        1.00        1.00    5090096

Testing Report:
              precision    recall  f1-score   support

   False         1.00        1.00        1.00   1272512
   True          0.00        0.00        0.00         12

 accuracy          1.00   1272524
 macro avg         0.50        0.50        0.50   1272524
 weighted avg       1.00        1.00        1.00   1272524
```

**classification\_report:** This function is used to create a classification report, which is a text summary of the performance of a classification model. It compares the true labels (`y_train` and `y_test`) with the predicted labels (`y_train_predict` and `y_test_predict`) for both the training and testing data.

**y\_train, y\_test:** These represent the true labels for the training and testing data, respectively.

**y\_train\_predict, y\_test\_predict:** These represent the labels predicted by the model for the training and testing data, respectively.

**print:** This function is used to output the classification reports to the console.

## Classification Report:

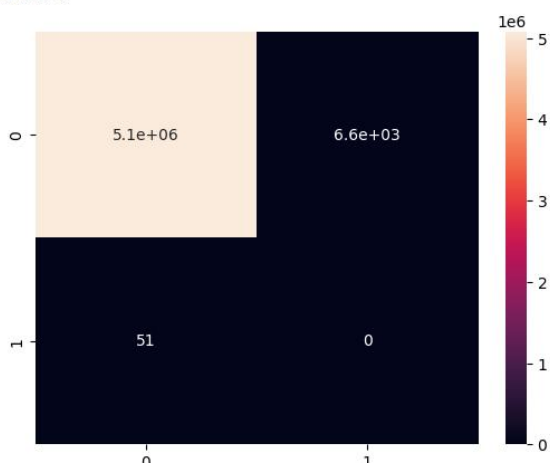
The classification report provides detailed information about the model's performance on a multi-class classification task. Here's an explanation of the different metrics included in the report (assuming you have multiple classes):

- **Support:** This shows the total number of true instances for each class.
- **Precision:** This is the ratio of correctly predicted positive cases to the total predicted positive cases (including false positives).
- **Recall:** This is the ratio of correctly predicted positive cases to the total true positive cases (including false negatives).
- **F1-score:** This is the harmonic mean of precision and recall, combining both metrics into a single score.
- **Accuracy:** This is the ratio of correctly predicted instances to the total number of instances (sometimes not included in the report)

```
[ ] from sklearn.metrics import confusion_matrix, classification_report
cm = confusion_matrix(y_train_predict, y_train)
cm2 = confusion_matrix(y_test_predict, y_test)
```

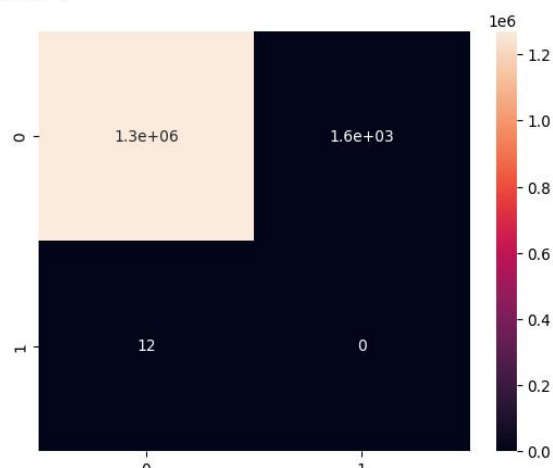
```
[ ] sns.heatmap(cm, annot = True)
```

<Axes: >



```
[ ] sns.heatmap(cm2, annot = True)
```

<Axes: >



### Code Snippet (Creating Confusion Matrix):

The Python code snippet appears to be generating a confusion matrix using the `confusion_matrix` function from the scikit-learn library. Here's a breakdown of the code

**from sklearn.metrics import confusion\_matrix:** This line imports the `confusion_matrix` function from scikit-learn.  
**cm = confusion\_matrix(y\_train, y\_train\_predict):** This line creates a confusion matrix for training data.  
**cm2 = confusion\_matrix(y\_test, y\_test\_predict):** This line creates a confusion matrix for testing data.

- `y_test`: This represents the true labels for the testing data.
- `y_train`: This represents the true labels for the training data.
- `y_test_predict`: These are the labels predicted by the model for the testing data.
- `Y_train_predict`: These are the labels predicted by the model for the training data.
- The result of this line is stored in the variable `cm`.

### Confusion Matrix Visualization:

The confusion matrix visualization is a table that summarizes the performance of a classification model on a dataset. Let's analyze the different elements of the confusion matrix:

- **Rows:** The rows represent the actual labels (ground truth) of the data points.
- **Columns:** The columns represent the labels predicted by the model.
- **Diagonal Values:** The diagonal values represent the correctly classified instances. Ideally, these values should be high for a good model.
- **Off-Diagonal Values:** The off-diagonal values represent the incorrectly classified instances. These cells show how many data points were misclassified as a particular class.

---

### Prevention Measures for Infrastructure Update

1. **Real-Time Monitoring:** Implement real-time transaction monitoring systems to detect suspicious activities.
2. **Anomaly Detection Systems:** Use machine learning models for anomaly detection.
3. **Two-Factor Authentication:** Enhance security with two-factor authentication for high-risk transactions.
4. **Regular Audits:** Conduct regular audits and penetration testing.
5. **Data Encryption:** Ensure all sensitive data is encrypted during transmission and storage.

---

### Determining the Effectiveness of Implemented Actions

To determine if the actions are effective:

1. **Continuous Monitoring:** Track the number of fraudulent transactions before and after implementation.
  2. **Performance Metrics:** Regularly evaluate model performance metrics such as precision, recall, and ROC-AUC.
  3. **User Feedback:** Collect feedback from customers about the security measures.
  4. **Incident Reports:** Analyze incident reports to identify any reduction in fraud attempts.
-