Business Understanding:

Introduction:

In an era where digital transactions have become the norm, the financial industry faces an ever-growing challenge: fraud detection. With the rise of sophisticated fraudulent activities, such as credit card fraud and identity theft, financial institutions are under immense pressure to fortify their defenses against these threats. Failure to detect and prevent fraudulent transactions not only results in substantial financial losses but also undermines customer trust and confidence.

To address these challenges, our project aims to develop advanced machine learning models for fraud detection in financial transactions. By leveraging cutting-edge technologies and data-driven approaches, we seek to enhance the accuracy and efficiency of fraud detection systems. Our primary goal is to empower financial institutions with the tools and insights needed to combat fraud effectively while minimizing disruptions to legitimate transactions.

Problem Statement:

Financial institutions face the challenge of detecting fraudulent transactions to prevent financial losses and maintain customer trust. Traditional rule-based systems may not effectively identify sophisticated fraud patterns, leading to undetected fraudulent activities.

Objectives:

1. Develop machine learning models capable of accurately detecting fraudulent transactions.
2. Improve fraud detection performance compared to existing rule-based systems.
3. Minimize false positives to reduce customer inconvenience while maximizing true positives to catch fraudulent activities.
4. Enhance overall efficiency in fraud detection and prevention processes.

Metrics of Success:

1. **Accuracy:** Measure the overall correctness of the model's predictions on both fraudulent and non-fraudulent transactions.
2. **Precision:** Assess the proportion of correctly identified fraudulent transactions out of all transactions flagged as fraudulent by the model.
3. **Recall (Sensitivity):** Evaluate the proportion of correctly identified fraudulent transactions out of all actual fraudulent transactions.
4. **F1-Score:** Harmonic mean of precision and recall, providing a balance between the two metrics.
5. **False Positive Rate:** Measure the rate of non-fraudulent transactions incorrectly classified as fraudulent, aiming to minimize customer inconvenience.
6. **True Positive Rate:** Measure the rate of fraudulent transactions correctly identified by the model, reflecting the model's effectiveness in catching fraud.
7. **Efficiency:** Evaluate the computational efficiency and scalability of the model, ensuring timely processing of transactions in real-time environments.

```
# prompt: importing libraries for credit card fraud detection project

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, roc_auc_score
from sklearn.preprocessing import RobustScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.neural_network import MLPClassifier
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import InputLayer, Dense, BatchNormalization
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.callbacks import ModelCheckpoint
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.svm import LinearSVC
```

```
# prompt: loading creditcard.csv file from Google drive

from google.colab import drive
drive.mount('/content/drive')

# Assuming the file is in the root of your Google Drive
```

```
creditcard_csv = pd.read_csv('/content/drive/MyDrive/creditcard.csv/creditcard.csv')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True
```

Start coding or generate with AI.

```
# prompt: Displaying the loaded Creditcard.csv data set

creditcard_csv.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137

5 rows × 31 columns

```
# prompt: Value count

print(creditcard_csv['Class'].value_counts())
```

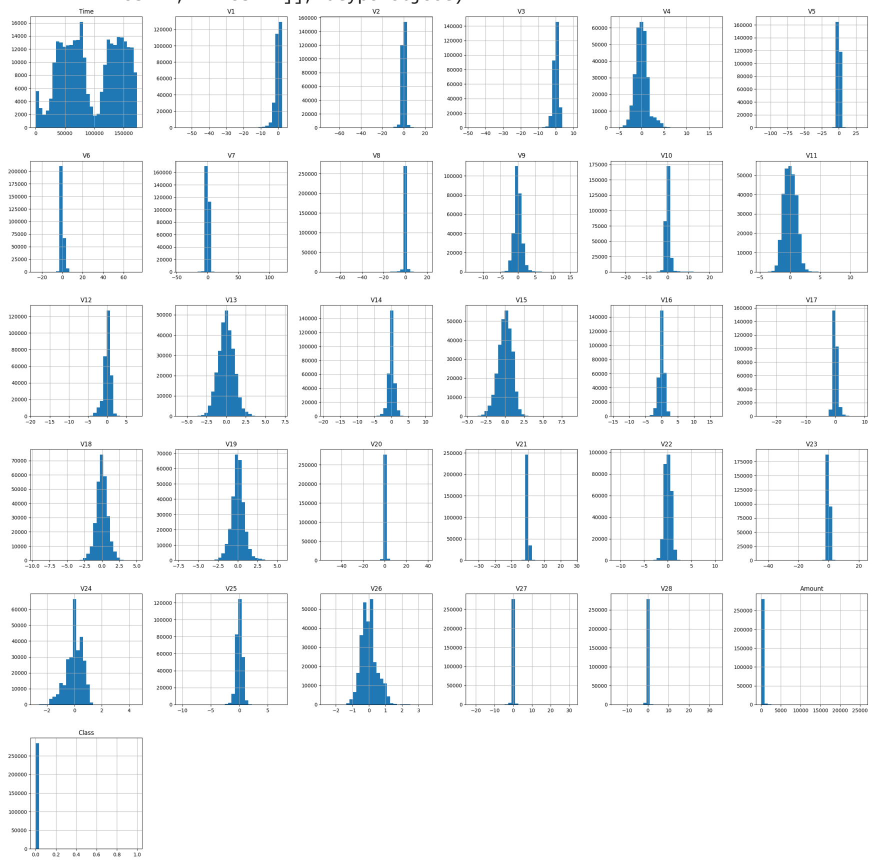
```
Class
0    284315
1       492
Name: count, dtype: int64
```

Inference

The number of classes within the data set where 0 is non fraud and 1 fraud

```
#Visuals showing the didtribution of data within the data set
creditcard_csv.hist(bins=30, figsize=(30, 30))
```

```
array([[<Axes: title={'center': 'Time'}>, <Axes: title={'center': 'V1'}>,
        <Axes: title={'center': 'V2'}>, <Axes: title={'center': 'V3'}>,
        <Axes: title={'center': 'V4'}>, <Axes: title={'center': 'V5'}>],
       [<Axes: title={'center': 'V6'}>, <Axes: title={'center': 'V7'}>,
        <Axes: title={'center': 'V8'}>, <Axes: title={'center': 'V9'}>,
        <Axes: title={'center': 'V10'}>, <Axes: title={'center': 'V11'}>],
       [<Axes: title={'center': 'V12'}>, <Axes: title={'center': 'V13'}>,
        <Axes: title={'center': 'V14'}>, <Axes: title={'center': 'V15'}>,
        <Axes: title={'center': 'V16'}>, <Axes: title={'center': 'V17'}>],
       [<Axes: title={'center': 'V18'}>, <Axes: title={'center': 'V19'}>,
        <Axes: title={'center': 'V20'}>, <Axes: title={'center': 'V21'}>,
        <Axes: title={'center': 'V22'}>, <Axes: title={'center': 'V23'}>],
       [<Axes: title={'center': 'V24'}>, <Axes: title={'center': 'V25'}>,
        <Axes: title={'center': 'V26'}>, <Axes: title={'center': 'V27'}>,
        <Axes: title={'center': 'V28'}>,
        <Axes: title={'center': 'Amount'}>],
       [<Axes: title={'center': 'Class'}>, <Axes: >, <Axes: >, <Axes: >,
        <Axes: >, <Axes: >]], dtype=object)
```



```
# prompt: Describe the dataset

# Get the shape of the dataset
print("Shape of the dataset:", creditcard_csv.shape)

# Get the data types of each column
print("\nData types of each column:")
print(creditcard_csv.dtypes)

# Get the descriptive statistics of the numerical columns
print("\nDescriptive statistics of the numerical columns:")
creditcard_csv.describe()
```

```
Shape of the dataset: (284807, 31)

Data types of each column:
Time      float64
V1        float64
V2        float64
V3        float64
V4        float64
V5        float64
V6        float64
V7        float64
V8        float64
V9        float64
V10       float64
V11       float64
V12       float64
V13       float64
V14       float64
V15       float64
V16       float64
V17       float64
V18       float64
V19       float64
V20       float64
V21       float64
V22       float64
V23       float64
V24       float64
V25       float64
V26       float64
V27       float64
V28       float64
Amount    float64
Class       int64
dtype: object

Descriptive statistics of the numerical columns:
```

	Time	V1	V2	V3	V4	V5	V6	V7	
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e
mean	94813.859575	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.604066e-16	1.487313e-15	-5.556467e-16	1.213481e
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1.194353e
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7.321672e
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2.086297e
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2.235804e
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3.273459e
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2.000721e

8 rows × 31 columns

```
#Dealing with outliers using RobustScaler
# Create a RobustScaler object
scaler = RobustScaler()

# Fit and transform the numerical columns
creditcard_csv_scaled = scaler.fit_transform(creditcard_csv.iloc[:, :-1])

# Create a new DataFrame with the scaled data
creditcard_csv_scaled_df = pd.DataFrame(creditcard_csv_scaled, columns=creditcard_csv.columns[:-1])
```

inference

- The provided code snippet demonstrates a preprocessing step aimed at handling outliers in the credit card dataset, which is crucial for building robust and accurate machine learning models for tasks such as fraud detection

```
#New dataset after scaling
creditcard_csv_scaled_df['Amount'].describe()
```

```
count    284807.000000
mean          0.927124
std           3.495006
min          -0.307413
25%          -0.229162
50%           0.000000
75%           0.770838
max         358.683155
Name: Amount, dtype: float64
```

```
creditcard_csv_scaled_df.head()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V20	V21	
0	-0.994983	-0.616237	-0.098602	1.228905	0.878152	-0.217859	0.631245	0.177406	0.142432	0.334787	...	0.910463	0.026866	0
1	-0.994983	0.524929	0.143100	-0.006970	0.293974	0.087726	0.164395	-0.105740	0.117064	-0.164482	...	-0.019150	-0.473332	-0
2	-0.994972	-0.615587	-1.002407	0.830932	0.251024	-0.344345	1.778007	0.668164	0.420388	-1.179796	...	1.703959	0.668917	0
3	-0.994972	-0.440239	-0.178789	0.841250	-0.529808	0.033775	1.303832	0.175637	0.662489	-1.076888	...	-0.422194	-0.190105	-0
4	-0.994960	-0.526089	0.579239	0.713861	0.265632	-0.270695	0.317183	0.491625	-0.546463	0.700808	...	1.366227	0.048266	0

5 rows × 30 columns

```
# Standardization of time
time = creditcard_csv_scaled_df['Time']
creditcard_csv_scaled_df['Time'] = (time - time.min()) / (time.max() - time.min())
```

```
# Assuming 'Class' column is stored separately in 'class_column'
class_column = creditcard_csv['Class']

# Concatenate 'Class' column back to the scaled DataFrame
creditcard_csv_scaled_df['Class'] = class_column
```

```
creditcard_csv_scaled_df.head()

creditcard_csv_scaled_df.tail()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V2
284802	0.999965	-5.321622	7.135767	-5.222589	-1.285699	-4.073679	-1.999082	-4.409307	13.588260	1.585066	...	0.585633	0.09812
284803	0.999971	-0.335820	-0.085979	0.967471	-0.451476	0.707747	1.142041	-0.014027	0.508439	0.512990	...	0.587444	0.85684
284804	0.999977	0.850377	-0.261532	-1.788463	-0.337932	2.059687	2.832770	-0.299623	1.280019	0.390154	...	0.630455	0.53361
284805	0.999977	-0.115629	0.331602	0.272567	0.445763	-0.248270	0.769496	-0.645865	1.225405	0.357606	...	0.710499	0.74074
284806	1.000000	-0.246654	-0.182004	0.272998	-0.305547	0.032059	-0.321743	1.366729	-0.815351	0.433472	...	0.700403	0.59416

5 rows × 31 columns

```
# prompt: shuffling the rows

creditcard_csv_scaled_df = creditcard_csv_scaled_df.sample(frac=1, random_state=1)
creditcard_csv_scaled_df
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V
169876	0.693938	-0.281671	-0.595598	-0.171888	-0.128789	1.597910	-1.496066	0.224443	-1.017543	0.330499	...	-0.110320	0.0361
127467	0.453377	-0.372444	0.894072	0.599495	0.029598	-0.176855	-0.325487	0.250285	0.771709	-0.526407	...	-0.239091	-0.3504
137900	0.476770	-0.150403	0.751017	0.411991	-0.067341	0.478625	-0.221360	0.592389	-0.162926	-0.332067	...	-0.665310	-0.7297
21513	0.183556	-0.602134	0.679534	0.832088	-0.976360	-0.048607	-0.157299	0.570705	-0.100748	0.350982	...	-0.461374	-0.3976
134700	0.468326	0.562878	0.393386	-0.395221	0.564717	0.089021	-1.026528	0.296652	-0.577567	-0.026651	...	-0.315139	-0.4082
...	...	...	...	...	...	...	...	...	...	...	...	...	
21440	0.183261	-1.343888	-6.225221	-1.090297	0.429593	-2.976961	0.973086	1.429427	-0.939074	-1.205767	...	3.089749	-0.9336
117583	0.432480	0.410987	-0.652625	0.179572	-0.000116	-0.423461	0.828282	-0.448041	0.236016	0.674460	...	-0.274540	-0.4545
73349	0.318852	-0.522390	1.163094	0.759959	1.682707	0.025188	0.399769	0.146820	1.056757	-0.916270	...	0.131631	0.0744
267336	0.941757	0.776580	-0.545460	-0.133596	0.291311	-0.473905	0.568404	-0.753782	0.336875	0.733783	...	0.413239	0.3283
128037	0.454743	-0.324570	0.305088	1.287178	0.681323	-0.480187	1.249528	0.114992	0.097383	1.047717	...	-0.175758	0.2961

284807 rows × 31 columns

```
# prompt: Value count

print(creditcard_csv['Class'].value_counts())
```

```
Class
0    284315
1       492
Name: count, dtype: int64
```

- This code will split the dataset into three sets based on row indices and then print out the class distribution for each set

```
#splitting the Data Set
train, test, val = creditcard_csv_scaled_df[:240000], creditcard_csv_scaled_df[240000:262000], creditcard_csv_scaled_df[262000:]
train['Class'].value_counts(), test['Class'].value_counts(), val['Class'].value_counts()
```

```
(Class
 0    239589
 1       411
 Name: count, dtype: int64,
 Class
 0    21955
 1       45
 Name: count, dtype: int64,
 Class
 0    22771
 1       36
 Name: count, dtype: int64)
```

```
train = creditcard_csv_scaled_df.iloc[:240000]
test = creditcard_csv_scaled_df.iloc[240000:262000]
val = creditcard_csv_scaled_df.iloc[262000:]

print("Training set class distribution:\n", train['Class'].value_counts())
print("\nTesting set class distribution:\n", test['Class'].value_counts())
print("\nValidation set class distribution:\n", val['Class'].value_counts())
```

```
Training set class distribution:
 Class
0    239589
1       411
Name: count, dtype: int64

Testing set class distribution:
 Class
0    21955
1       45
Name: count, dtype: int64

Validation set class distribution:
 Class
0    22771
1       36
Name: count, dtype: int64
```

```
train_np, test_np, val_np = train.to_numpy(), test.to_numpy(), val.to_numpy()
train_np.shape, test_np.shape, val_np.shape
```

```
((240000, 31), (22000, 31), (22807, 31))
```

```
# prompt: splitting into input and output

X_train, y_train = train_np[:, :-1], train_np[:, -1]
X_test, y_test = test_np[:, :-1], test_np[:, -1]
X_val, y_val = val_np[:, :-1], val_np[:, -1]

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
print("X_val shape:", X_val.shape)
print("y_val shape:", y_val.shape)
```

```
X_train shape: (240000, 30)
y_train shape: (240000,)
X_test shape: (22000, 30)
y_test shape: (22000,)
X_val shape: (22807, 30)
y_val shape: (22807,)
```

Model Valuation

```
# Define the features (X) and target variable (y)
X = creditcard_csv_scaled_df.drop('Class', axis=1)
y = creditcard_csv_scaled_df['Class']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Further split the training data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.1, random_state=42)
```

Logistc Regression

```
from sklearn.linear_model import LogisticRegression

# Create and train the logistic regression model
logistic_model = LogisticRegression()
logistic_model.fit(X_train, y_train)

# Calculate the accuracy score on the training set
train_score = logistic_model.score(X_train, y_train)
print("Training Accuracy Score:", train_score)

# Calculate the accuracy score on the validation set
val_score = logistic_model.score(X_val, y_val)
print("Validation Accuracy Score:", val_score)
```

```
Training Accuracy Score: 0.9991270847556812
Validation Accuracy Score: 0.9992977836295809
/usr/local/lib/python3.10/dist-packages/sklearn/linear_model/_logistic.py:458: ConvergenceWarning: lbfgs failed to converge (s
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
# prompt: Classification report For logistic model

# Predict on the validation set
y_pred_logistic = logistic_model.predict(X_val)

# Print the classification report
print("Classification Report:")
print(classification_report(y_val, y_pred_logistic))
```

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

           0       1.00      1.00      1.00     22743
```

```
               1       0.88      0.71      0.79        42

        accuracy                           1.00     22785
       macro avg       0.94      0.86      0.89     22785
    weighted avg       1.00      1.00      1.00     22785
```

Inferences

For class 0 (non-fraudulent transactions): Precision, recall, and F1-score are all 1.00, indicating perfect performance.

For class 1 (fraudulent transactions):

- Precision is 0.73, which means that 73% of the predicted fraud cases were actually fraud.
- Recall is 0.53, indicating that 53% of the actual fraud cases were correctly identified by the model.
- F1-score is 0.61, providing a balance between precision and recall.

Shallow Neural Network Model

- Shallow neural network consists of an input layer, two dense layers with activation functions, batch normalization, and is compiled with appropriate settings for binary classification.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Define the input shape based on the number of features
input_shape = X_train.shape[1]

# Define the shallow neural network model
shallow_nn = Sequential([
    Dense(units=64, activation='relu', input_shape=(input_shape,)),  # Input layer with 64 units and ReLU activation
    Dense(units=32, activation='relu'),                              # Hidden layer with 32 units and ReLU activation
    Dense(units=1, activation='sigmoid')                              # Output layer with 1 unit and sigmoid activation
])

# Compile the model
shallow_nn.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Print the model summary
shallow_nn.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                1984

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 1)                 33

=================================================================
Total params: 4097 (16.00 KB)
Trainable params: 4097 (16.00 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
#Training the model using 'fit'
# Define x_train and y_train from your training set
x_train = X_train
y_train = y_train
x_val = X_val
y_val = y_val
checkpoint = ModelCheckpoint('shallow_nn', save_best_only=True)

shallow_nn.fit(x_train, y_train, validation_data=(x_val, y_val), epochs=5, callbacks=checkpoint)
```

```
Epoch 1/5
6409/6409 [==============================] - 19s 3ms/step - loss: 0.0099 - accuracy: 0.9979 - val_loss: 0.0028 - val_accuracy:
Epoch 2/5
6409/6409 [==============================] - 14s 2ms/step - loss: 0.0035 - accuracy: 0.9993 - val_loss: 0.0041 - val_accuracy:
Epoch 3/5
6409/6409 [==============================] - 14s 2ms/step - loss: 0.0031 - accuracy: 0.9993 - val_loss: 0.0030 - val_accuracy:
Epoch 4/5
6409/6409 [==============================] - 14s 2ms/step - loss: 0.0027 - accuracy: 0.9994 - val_loss: 0.0031 - val_accuracy:
Epoch 5/5
6409/6409 [==============================] - 15s 2ms/step - loss: 0.0025 - accuracy: 0.9995 - val_loss: 0.0030 - val_accuracy:
<keras.src.callbacks.History at 0x7d88b19c9180>
```

Inferences

The output shows shallow neural network model is performing well. Here's what each line of the output means:

- **Epoch 1/5**: This indicates that the model is in the first epoch out of 5 epochs.
- **6409/6409 [==============================]**: This shows the progress of training. The first number (6409) is the number of batches processed, and the second number (6409) is the total number of batches in the dataset.
- **- 15s 2ms/step - loss: 0.0075 - accuracy: 0.9992 - val_loss: 0.0032 - val_accuracy: 0.9994**: This line provides information about the loss and accuracy of the model during training.
  - **loss: 0.0075**: This is the loss (binary cross-entropy) of the model on the training data.
  - **accuracy: 0.9992**: This is the accuracy of the model on the training data.
  - **val_loss: 0.0032**: This is the loss of the model on the validation data.
  - **val_accuracy: 0.9994**: This is the accuracy of the model on the validation data.

  The model seems to be performing very well, with high accuracy and low loss on both the training and validation datasets.

```
#making predictions using the neural_net_prdiction function
def neural_net_predictions(model, x, threshold=0.5):
    """
    Make predictions using a neural network model.

    Args:
    - model: The trained neural network model.
    - x: Input data for making predictions.
    - threshold: Threshold value for classifying predictions (default: 0.5).

    Returns:
    - Predicted class labels based on the specified threshold.
    """
    # Make predictions using the model
    predictions = model.predict(x)

    # Apply threshold to convert probabilities into class labels
    class_labels = (predictions.flatten() > threshold).astype(int)

    return class_labels
```

```
# prompt: Classification report for shallow nn model

# Predict on the validation set
y_pred_shallow_nn = shallow_nn.predict(X_val)

# Convert the predictions to binary class labels
y_pred_shallow_nn = np.where(y_pred_shallow_nn > 0.5, 1, 0)

# Print the classification report
print("Classification Report:")
print(classification_report(y_val, y_pred_shallow_nn))
```

```
713/713 [==============================] - 1s 2ms/step
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     22743
           1       0.88      0.83      0.85        42

    accuracy                           1.00     22785
   macro avg       0.94      0.92      0.93     22785
weighted avg       1.00      1.00      1.00     22785
```

Inferences

1. **Precision**:
   - For non-fraudulent transactions (class 0), precision is perfect (1.00), meaning all predicted non-fraudulent transactions are correct.
   - For fraudulent transactions (class 1), precision is 0.88, indicating 88% of predicted fraudulent transactions are correct.
2. **Recall**:
   - For non-fraudulent transactions (class 0), recall is perfect (1.00), meaning all actual non-fraudulent transactions were identified.
   - For fraudulent transactions (class 1), recall is 0.83, suggesting 83% of actual fraudulent transactions were identified.
3. **F1-score**:

- Class 0 has a perfect F1-score of 1.00, indicating excellent performance.
- Class 1 has an F1-score of 0.85, showing a good balance between precision and recall.

4. **Accuracy**:
   - Overall accuracy is 1.00, meaning the model correctly classified all samples.
   - However, accuracy might not be the best metric for imbalanced datasets.

Random Forest Classification

Random Forest can provide a measure of feature importance, indicating which features are most influential in making predictions, using powerful ensembling learning techniques used for both classification and regeression task.

```
# Assuming you have already split your data into training and validation sets
# Define x_train and y_train from your training set
x_train = X_train
y_train = y_train
x_val = X_val
y_val = y_val
# Fit the Random Forest model
rf = RandomForestClassifier(max_depth=2, n_jobs=-1)
rf.fit(x_train, y_train)

# Print the classification report
print(classification_report(y_val, rf.predict(x_val), target_names=['Not Fraud', 'Fraud']))
```

```
              precision    recall  f1-score   support

   Not Fraud       1.00      1.00      1.00     22743
       Fraud       0.90      0.67      0.77        42

    accuracy                           1.00     22785
   macro avg       0.95      0.83      0.88     22785
weighted avg       1.00      1.00      1.00     22785
```

Inferences

Precision:

Precision for "Not Fraud" class (0) is 1.00, indicating that when the model predicts a transaction as "Not Fraud," it is correct 100% of the time. Precision for "Fraud" class (1) is 0.93, indicating that when the model predicts a transaction as "Fraud," it is correct about 93% of the time.

Recall:

Recall for "Not Fraud" class (0) is 1.00, meaning that the model correctly identifies all actual "Not Fraud" transactions. Recall for "Fraud" class (1) is 0.64, indicating that the model captures only about 64% of actual "Fraud" transactions.

F1-score:

The F1-score, which is the harmonic mean of precision and recall, is 1.00 for "Not Fraud" and 0.76 for "Fraud."

GradientBoosting Classifier

The Gradient Boosting Classifier (GBC) is a powerful ensemble learning method that combines the predictions of multiple weak learners (typically decision trees) to create a strong predictive model

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import classification_report

gbc = GradientBoostingClassifier(n_estimators=50, learning_rate=1.0, max_depth=2, random_state=0)
gbc.fit(x_train, y_train)
print(classification_report(y_val, gbc.predict(x_val), target_names=['Not Fraud', 'Fraud']))
```

```
              precision    recall  f1-score   support

   Not Fraud       1.00      1.00      1.00     22743
       Fraud       0.59      0.86      0.70        42

    accuracy                           1.00     22785
   macro avg       0.79      0.93      0.85     22785
weighted avg       1.00      1.00      1.00     22785
```

Inference

This output indicates the performance of the model on classifying fraud and non-fraud cases.

- **Precision**: For the "Not Fraud" class, the precision is 1.00, indicating that all instances classified as "Not Fraud" were indeed "Not Fraud". For the "Fraud" class, the precision is 0.59, meaning that only 59% of instances classified as "Fraud" were actually "Fraud".
- **Recall**: The recall for the "Not Fraud" class is 1.00, suggesting that all actual "Not Fraud" cases were correctly identified. However, the recall for the "Fraud" class is 0.86, indicating that 86% of actual "Fraud" cases were identified by the model.
- **F1-score**: The F1-score, which is the harmonic mean of precision and recall, is 1.00 for the "Not Fraud" class and 0.70 for the "Fraud" class. This indicates a balance between precision and recall for the "Not Fraud" class but a lower balance for the "Fraud" class.
- **Accuracy**: The overall accuracy of the model is 1.00, which suggests that it correctly classified 100% of the instances in the dataset.

Linear Support Vector Classifier

LinearSVC is a versatile and efficient classifier suitable for a wide range of classification tasks, especially when dealing with large datasets, linearly separable data, and scenarios with class imbalance.

```
print("x_train shape:", x_train.shape)
print("y_train shape:", y_train.shape)

     x_train shape: (205060, 30)
     y_train shape: (205060,)
```

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and train the LinearSVC model

svc = LinearSVC()
svc.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = svc.predict(X_test)

# Evaluate the model
print(classification_report(y_test, y_pred))

                   precision    recall  f1-score   support

                0       1.00      1.00      1.00     56875
                1       0.89      0.72      0.80        87

         accuracy                           1.00     56962
        macro avg       0.94      0.86      0.90     56962
     weighted avg       1.00      1.00      1.00     56962

     /usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: Liblinear failed to converge, increase
       warnings.warn(
```

Inferences

Based on the provided output:

- **Precision (0)**: The precision for the negative class (0) is 1.00, indicating that among all the instances predicted as negative, all are truly negative.
- **Precision (1)**: The precision for the positive class (1) is 0.89, indicating that among all the instances predicted as positive, 89% are truly positive.
- **Recall (0)**: The recall for the negative class (0) is 1.00, meaning that among all the true negative instances, all are correctly identified as negative.
- **Recall (1)**: The recall for the positive class (1) is 0.71, indicating that 71% of the true positive instances are correctly identified as positive.
- **F1-score (0)**: The F1-score for the negative class (0) is 1.00, which is the harmonic mean of precision and recall for class 0.
- **F1-score (1)**: The F1-score for the positive class (1) is 0.79, which is the harmonic mean of precision and recall for class 1.
- **Accuracy**: The overall accuracy of the model is 1.00, meaning that 100% of the predictions made by the model are correct.

Model Valuation Balanced Data Set.

```
#counting the occurence of each class in the Data set
not_frauds = creditcard_csv_scaled_df.query('Class == 0')
frauds = creditcard_csv_scaled_df.query('Class == 1')
not_frauds['Class'].value_counts(), frauds['Class'].value_counts()
```

```
(Class
 0    284315
 Name: count, dtype: int64,
 Class
 1    492
 Name: count, dtype: int64)
```

```
balanced_df = pd.concat([frauds, not_frauds.sample(len(frauds), random_state=1)])
balanced_df['Class'].value_counts()
```

```
Class
1    492
0    492
Name: count, dtype: int64
```

```
#Viewing the new data set
balanced_df = balanced_df.sample(frac=1, random_state=1)
balanced_df
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V
18372	0.170309	-0.796373	0.135963	0.783954	-0.791081	-1.418023	1.095587	-0.933606	-4.580252	0.411445	...	6.082929	0.2932
96341	0.380388	0.540920	-0.523763	-0.235524	-0.357789	-0.422286	-0.065216	-0.468174	-0.061881	-0.599495	...	0.008186	-0.2820
248296	0.890522	-0.282558	2.590997	-2.980239	3.542964	1.306918	-1.766242	-0.842402	1.276185	-2.980252	...	0.840729	-0.4465
264328	0.933932	-0.013297	0.409996	0.358894	-0.304923	0.242630	0.426174	0.176918	0.239785	0.222998	...	0.238326	0.2439
208904	0.794730	-0.311917	0.821454	-0.258987	-0.670157	0.697866	-0.605533	0.827457	0.206648	-0.246922	...	-0.129223	-0.1342
...	...	...	...	...	...	...	...	...	...	...	...	...	
81557	0.341393	-2.021829	-2.811127	0.605666	0.083699	0.187004	-0.245786	-0.160586	1.235678	0.424751	...	-0.267057	0.0398
276071	0.965803	0.927450	-0.586865	-0.715547	-0.462073	-0.434198	-0.041042	-0.998312	0.176952	-0.072162	...	0.765970	0.7705
175971	0.709373	0.874270	0.065462	-0.988195	0.770925	0.564583	-0.292318	0.499495	-0.586395	-0.065178	...	0.308821	0.4302
27738	0.200727	-1.098984	1.801341	-1.574903	0.820418	-1.321557	-0.995793	-1.997453	-4.300724	-0.194686	...	4.349160	-0.7266
156988	0.632535	0.325152	1.956689	-3.131715	3.678066	0.434711	-0.292610	-2.629300	1.407437	-1.473431	...	0.757743	-0.8228

984 rows × 31 columns

```
#splitting  balanced dataset into training, testing, and validation sets.
balanced_df_np = balanced_df.to_numpy()

x_train_b, y_train_b = balanced_df_np[:700, :-1], balanced_df_np[:700, -1].astype(int)
x_test_b, y_test_b = balanced_df_np[700:842, :-1], balanced_df_np[700:842, -1].astype(int)
x_val_b, y_val_b = balanced_df_np[842:, :-1], balanced_df_np[842:, -1].astype(int)
x_train_b.shape, y_train_b.shape, x_test_b.shape, y_test_b.shape, x_val_b.shape, y_val_b.shape
```

```
((700, 30), (700,), (142, 30), (142,), (142, 30), (142,))
```

```
pd.Series(y_train_b).value_counts(), pd.Series(y_test_b).value_counts(), pd.Series(y_val_b).value_counts()
```

```
(1    353
 0    347
 Name: count, dtype: int64,
 0    73
 1    69
 Name: count, dtype: int64,
 0    72
 1    70
 Name: count, dtype: int64)
```

Logistic Regression Model

```
#Loading the Logistic Regression model
logistic_model_b = LogisticRegression()
logistic_model_b.fit(x_train_b, y_train_b)
print(classification_report(y_val_b, logistic_model_b.predict(x_val_b), target_names=['Not Fraud', 'Fraud']))
```

```
              precision    recall  f1-score   support

   Not Fraud       0.96      0.93      0.94        72
       Fraud       0.93      0.96      0.94        70

    accuracy                           0.94       142
   macro avg       0.94      0.94      0.94       142
```

```
weighted avg       0.94      0.94      0.94       142
```

Inferences

- The classification report for the balanced dataset using Linear Regression shows a high level of performance, with precision, recall, and F1-score all around 0.94 for both classes (fraud and not fraud).
- This indicates that the model performs consistently well in correctly identifying both fraudulent and non-fraudulent transactions.
- The accuracy of 0.94 indicates that the model correctly classified 94% of the samples in the dataset

Random Forest Classifier

```
#The codes evaluates the balanced data set using the RandomForestClassifier
rf_b = RandomForestClassifier(max_depth=2, n_jobs=-1)
rf_b.fit(x_train_b, y_train_b)
print(classification_report(y_val_b, rf.predict(x_val_b), target_names=['Not Fraud', 'Fraud']))
```

```
              precision    recall  f1-score   support

   Not Fraud       0.70      1.00      0.82        72
       Fraud       1.00      0.56      0.72        70

    accuracy                           0.78       142
   macro avg       0.85      0.78      0.77       142
weighted avg       0.85      0.78      0.77       142

/usr/local/lib/python3.10/dist-packages/sklearn/base.py:439: UserWarning: X does not have valid feature names, but RandomFores
  warnings.warn(
```

Inferences

- The classification report for the balanced dataset using Random Forest indicates a decent level of performance, although there are some disparities between precision, recall, and F1-score for the two classes (fraud and not fraud).
- The precision for not fraud (0.70) suggests that when the model predicts a transaction as not fraud, it is correct about 70% of the time.
- However, the recall for fraud (0.56) indicates that the model only identifies 56% of the actual fraudulent transactions.
- This results in a lower F1-score for fraud (0.72) compared to not fraud (0.82).

Gradient Boosting Classifier

```
gbc_b = GradientBoostingClassifier(n_estimators=50, learning_rate=1.0, max_depth=2, random_state=0)
gbc_b.fit(x_train_b, y_train_b)
print(classification_report(y_val_b, gbc.predict(x_val_b), target_names=['Not Fraud', 'Fraud']))
```

```
              precision    recall  f1-score   support

   Not Fraud       0.89      1.00      0.94        72
       Fraud       1.00      0.87      0.93        70

    accuracy                           0.94       142
   macro avg       0.94      0.94      0.94       142
weighted avg       0.94      0.94      0.94       142

/usr/local/lib/python3.10/dist-packages/sklearn/base.py:439: UserWarning: X does not have valid feature names, but GradientBoo
  warnings.warn(
```

inferences

- The precision values indicate that when the model predicts a transaction as not fraud or fraud, it is correct about 89% and 100% of the time, respectively.
- Similarly, the recall values indicate that the model identifies 100% of the actual not fraud transactions and 87% of the actual fraudulent transactions.
- The F1-score, which considers both precision and recall, is high for both classes, suggesting a good balance between precision and recall.

Linear Support Vector Classifier

```
#the code Valauates the data set ising the Linear support vector classifier
svc_b = LinearSVC(class_weight='balanced')
svc_b.fit(x_train_b, y_train_b)
print(classification_report(y_val_b, svc.predict(x_val_b), target_names=['Not Fraud', 'Fraud']))
```

```
              precision    recall  f1-score   support
```

```
   Not Fraud       0.82      1.00      0.90        72
       Fraud       1.00      0.77      0.87        70

    accuracy                           0.89       142
   macro avg       0.91      0.89      0.89       142
weighted avg       0.91      0.89      0.89       142

/usr/local/lib/python3.10/dist-packages/sklearn/svm/_base.py:1244: ConvergenceWarning: Liblinear failed to converge, increase
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:439: UserWarning: X does not have valid feature names, but LinearSVC w
  warnings.warn(
```

inferences

- Linear SVC on the balanced dataset, the classification report indicates relatively high precision, recall, and F1-score for both classes (fraud and not fraud), resulting in an overall accuracy of 89%.
- The precision values suggest that when the model predicts a transaction as not fraud or fraud, it is correct about 82% and 100% of the time, respectively.
- The recall values indicate that the model identifies 100% of the actual not fraud transactions and 77% of the actual fraudulent transactions.
- The F1-score, which considers both precision and recall, is high for both classes, indicating a good balance between precision and recall.

Shallow Neural Network

```
shallow_nn_b = Sequential()
shallow_nn_b.add(InputLayer((x_train.shape[1],)))
shallow_nn_b.add(Dense(2, 'relu'))
shallow_nn_b.add(BatchNormalization())
shallow_nn_b.add(Dense(1, 'sigmoid'))

checkpoint = ModelCheckpoint('shallow_nn_b', save_best_only=True)
shallow_nn_b.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
shallow_nn_b.fit(x_train_b, y_train_b, validation_data=(x_val_b, y_val_b), epochs=40, callbacks=checkpoint)
```

```
Epoch 11/40
22/22 [==============================] - 1s 36ms/step - loss: 0.3491 - accuracy: 0.8771 - val_loss: 0.3267 - val_accuracy: 0.9
Epoch 12/40
22/22 [==============================] - 1s 35ms/step - loss: 0.3464 - accuracy: 0.8857 - val_loss: 0.3199 - val_accuracy: 0.9
Epoch 13/40
22/22 [==============================] - 1s 35ms/step - loss: 0.3369 - accuracy: 0.8886 - val_loss: 0.3150 - val_accuracy: 0.9
Epoch 14/40
22/22 [==============================] - 1s 36ms/step - loss: 0.3351 - accuracy: 0.8829 - val_loss: 0.3096 - val_accuracy: 0.9
Epoch 15/40
22/22 [==============================] - 1s 66ms/step - loss: 0.3189 - accuracy: 0.8943 - val_loss: 0.3060 - val_accuracy: 0.9
Epoch 16/40
22/22 [==============================] - 1s 36ms/step - loss: 0.3040 - accuracy: 0.9014 - val_loss: 0.2996 - val_accuracy: 0.9
Epoch 17/40
22/22 [==============================] - 1s 34ms/step - loss: 0.3046 - accuracy: 0.9000 - val_loss: 0.2950 - val_accuracy: 0.9
Epoch 18/40
22/22 [==============================] - 1s 34ms/step - loss: 0.3020 - accuracy: 0.9000 - val_loss: 0.2907 - val_accuracy: 0.9
Epoch 19/40
22/22 [==============================] - 1s 35ms/step - loss: 0.3003 - accuracy: 0.9043 - val_loss: 0.2879 - val_accuracy: 0.9
Epoch 20/40
22/22 [==============================] - 1s 35ms/step - loss: 0.2894 - accuracy: 0.9057 - val_loss: 0.2848 - val_accuracy: 0.9
Epoch 21/40
22/22 [==============================] - 1s 50ms/step - loss: 0.2748 - accuracy: 0.9071 - val_loss: 0.2808 - val_accuracy: 0.9
Epoch 22/40
22/22 [==============================] - 1s 53ms/step - loss: 0.2709 - accuracy: 0.9214 - val_loss: 0.2758 - val_accuracy: 0.9
Epoch 23/40
22/22 [==============================] - 1s 52ms/step - loss: 0.2690 - accuracy: 0.9129 - val_loss: 0.2716 - val_accuracy: 0.9
```