CREDIT CARD FRAUD DETECTION USING DEEP LEARNING

Author： Boya Zeng

Problem Statement

Credit card fraud represents a significant challenge to financial security and customer trust. The project aims to develop a deep learning model that can effectively identify and classify transactions as fraudulent or legitimate, helping to mitigate losses due to fraud.

In [26]:
```
# !pip install keras-tuner
# !pip install imblearn
# !pip install -U imbalanced-learn
# !pip uninstall imbalanced-learn -y
# !pip install imbalanced-learn

# !pip install -U numpy scipy scikit-learn
```

About Dataset

Dataset: Credit Card Fraud Detection

Context It is important that credit card companies are able to recognize fraudulent credit card transactions so that customers are not charged for items that they did not purchase.

Content The dataset contains transactions made by credit cards in September 2013 by European cardholders. This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions.

It contains only numerical input variables which are the result of a PCA transformation. Unfortunately, due to confidentiality issues, we cannot provide the original features and more background information about the data. Features V1, V2, ... V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

In [203…
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from keras import regularizers
from sklearn.model_selection import train_test_split

from tensorflow.keras.regularizers import l2
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix,average_precision_score
from sklearn import preprocessing


from tensorflow import keras
from keras_tuner import RandomSearch

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from keras.layers import Input, Dense, BatchNormalization, Dropout
from keras.models import Model, Sequential
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.callbacks import EarlyStopping

from sklearn import metrics
from sklearn.metrics import roc_curve, roc_auc_score, auc, precision_recall_curve,  precision_recall_fscore_support
```

Dataset Attributes

V1 - V28 : Numerical features that are a result of PCA transformation.

Time : Seconds elapsed between each transaction and the 1st transaction.

Amount : Transaction amount.

Class : Fraud or otherwise (1 or 0)

Load and Explore the Data

In [2]:
```python
data = pd.read_csv("/Users/boyazeng/Downloads/Bose ML & Predictive Analytics/creditcard.csv")
#data["Time"] = data["Time"].apply(lambda x : x / 3600 % 24)


# Display the first few rows of the dataset
data.head()
```

Out[2]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V2
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	-0.11047
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	0.10128
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	0.90941
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	-0.19032
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	-0.13745

5 rows × 31 columns

In [35]:
```python
data.shape
```

Out[35]:
```
(284807, 31)
```

In [36]:
```python
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64  
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
```

The dataset consists of 28 anonymized variables, 1 "amount" variable, 1 "time" variable and 1 target variable - Class. Let's look at the distribution of target.

In [83]:
```python
# Check for missing values
data.isnull().sum()
```

Out[83]:
```
Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       0
V26       0
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

- No null values present in the data!

In [39]:
```python
# Statistical summary
data.describe()
```

Out[39]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8
count	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05
mean	-3.065637e-16	1.168375e-15	3.416908e-16	-1.379537e-15	2.074095e-15	9.604066e-16	1.487313e-15	-5.556467e-16	1.213481e-16
std	1.000002e+00	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1.194353e+00
min	-1.996583e+00	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7.321672e+01
25%	-8.552120e-01	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2.086297e-01
50%	-2.131453e-01	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2.235804e-02
75%	9.372174e-01	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3.273459e-01
max	1.642058e+00	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2.000721e+01

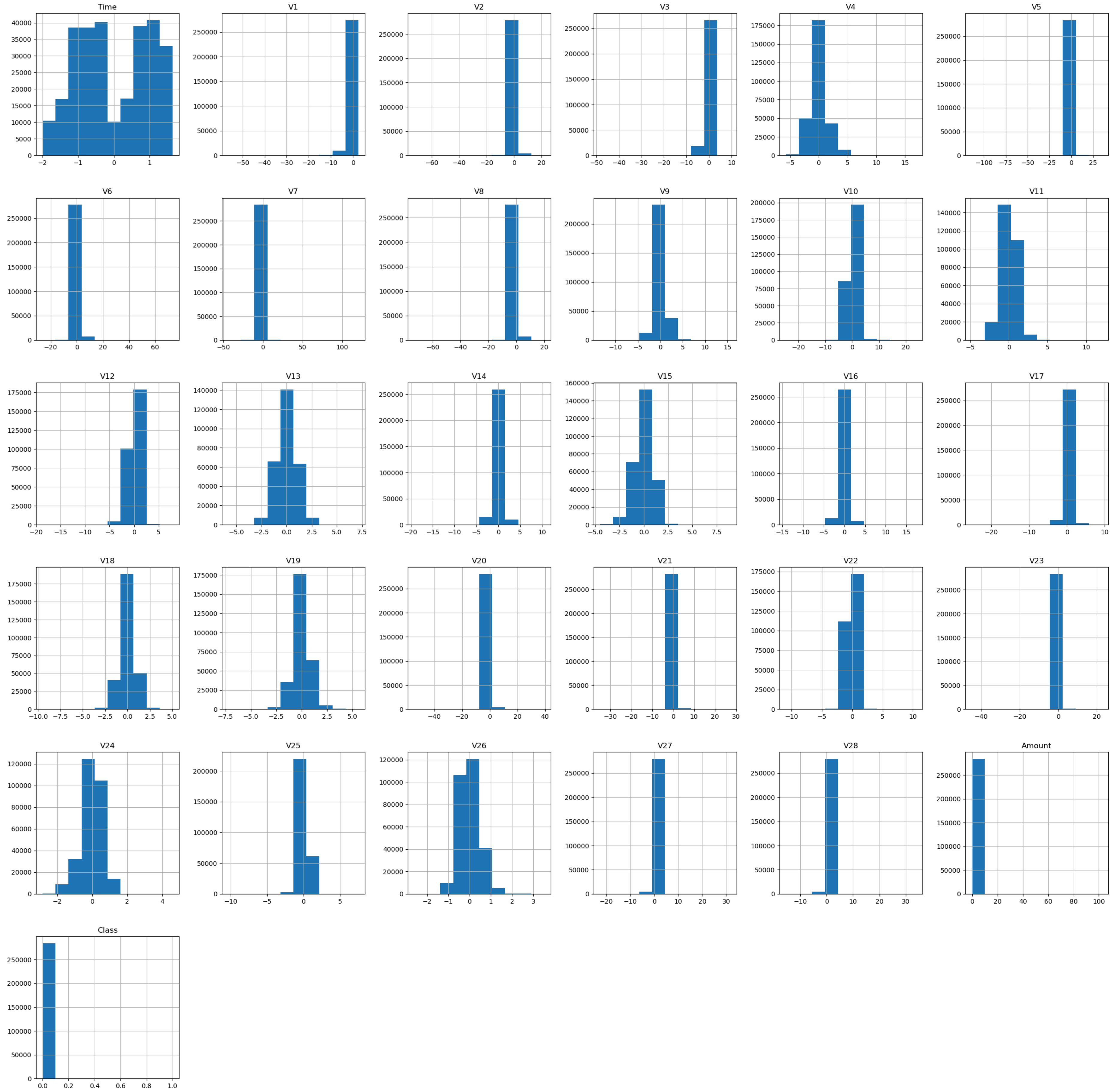
8 rows × 31 columns

Exploratory Data Analysis

The dataset contains 284,807 transactions, of which only 492 (0.17%) are fraudulent, indicating a highly imbalanced dataset.

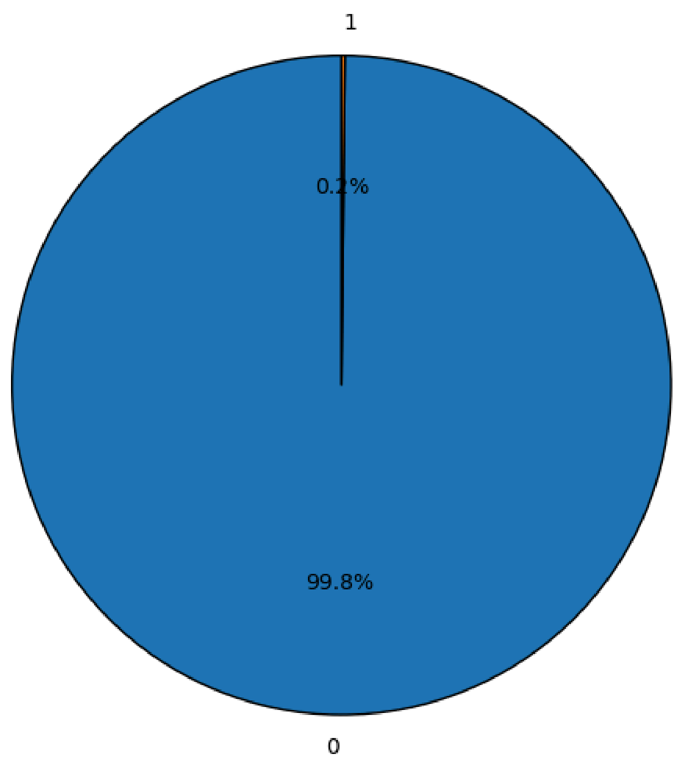
The visual analysis of 'Amount' shows a right-skewed distribution, typical for transactional data, where most transactions involve smaller amounts.

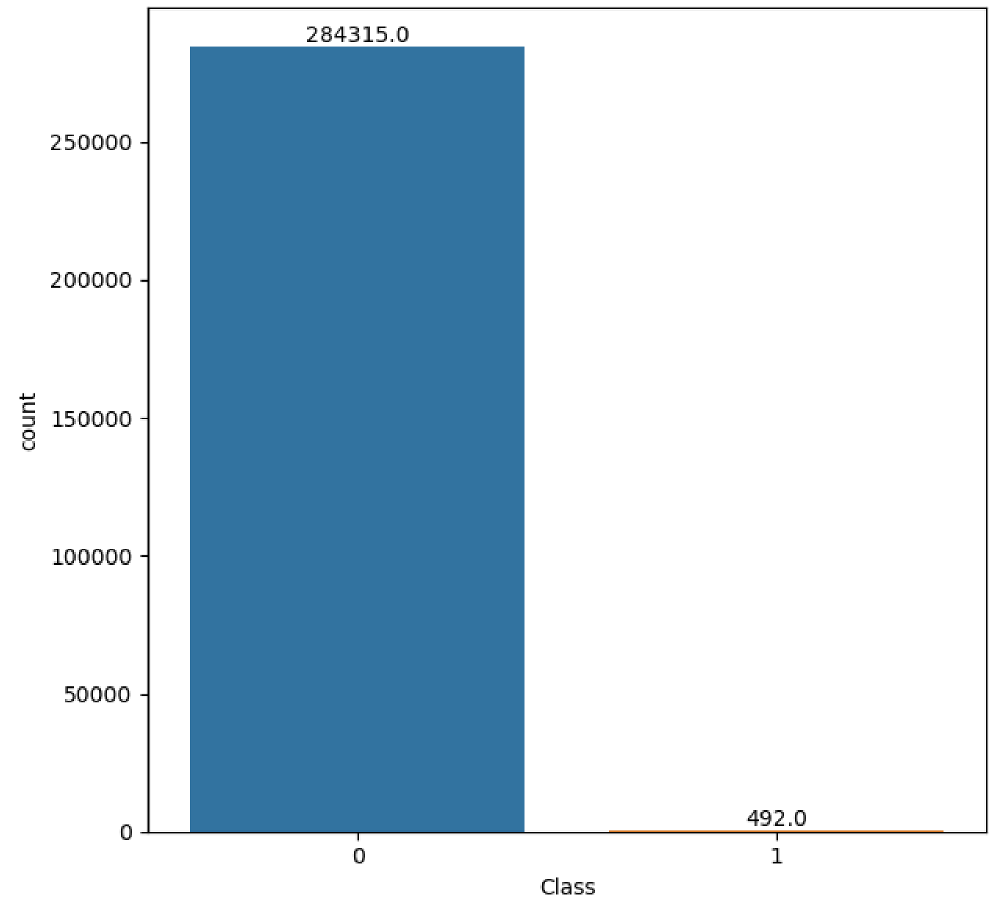
In [52]:
```python
data.hist(figsize=(30,30))
plt.show()
```



In [50]:

```python
fraud = len(data[data['Class'] == 1]) / len(data) * 100
nofraud = len(data[data['Class'] == 0]) / len(data) * 100
fraud_percentage = [nofraud,fraud]

plt.figure(figsize=(12, 6))
# Pie chart
fraud_percentage = data['Class'].value_counts(normalize=True)
plt.subplot(1, 2, 1)
plt.pie(fraud_percentage, labels=fraud_percentage.index, autopct='%1.1f%%', startangle=90,
        wedgeprops={'edgecolor': 'black', 'linewidth': 1, 'antialiased': True})

# Countplot
plt.subplot(1, 2, 2)
ax = sns.countplot(x='Class', data=data)  # Use 'x=' to specify the column name explicitly
for rect in ax.patches:
    ax.text(rect.get_x() + rect.get_width() / 2, rect.get_height() + 2, f'{rect.get_height()}',
            ha='center', va='bottom')

plt.tight_layout()
plt.show()
```






```
In [4]: # Check class distribution
        #data['Class'].value_counts()
        vc = data['Class'].value_counts().to_frame().reset_index()
        vc['percent'] = vc["count"].apply(lambda x : round(100*float(x) / len(data), 2))
        vc = vc.rename(columns = {"index" : "Target", "count" : "Count"})
        vc
```

Out[4]:

	Class	Count	percent
0	0	284315	99.83
1	1	492	0.17

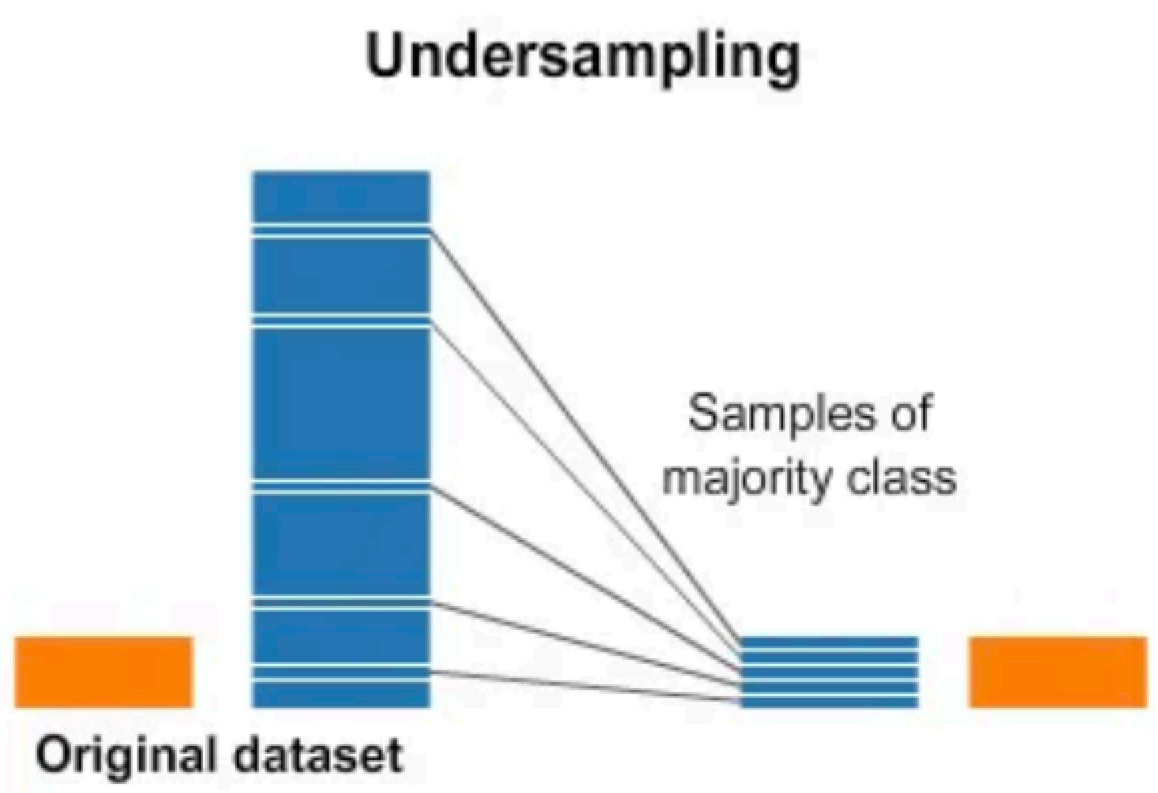
```
In [51]: #sns.countplot(x="Class", data=data)
```

The dataset is highly imbalanced ! It's a big problem because classifiers will always predict the most common class without performing any analysis of the features and it will have a high accuracy rate, obviously not the correct one. To change that, I will proceed to random undersampling.

The simplest undersampling technique involves randomly selecting examples from the majority class and deleting them from the training dataset. This is referred to as random undersampling.

Although simple and effective, a limitation of this technique is that examples are removed without any concern for how useful or important they might be in determining the decision boundary between the classes. This means it is possible, or even likely, that useful information will be deleted.

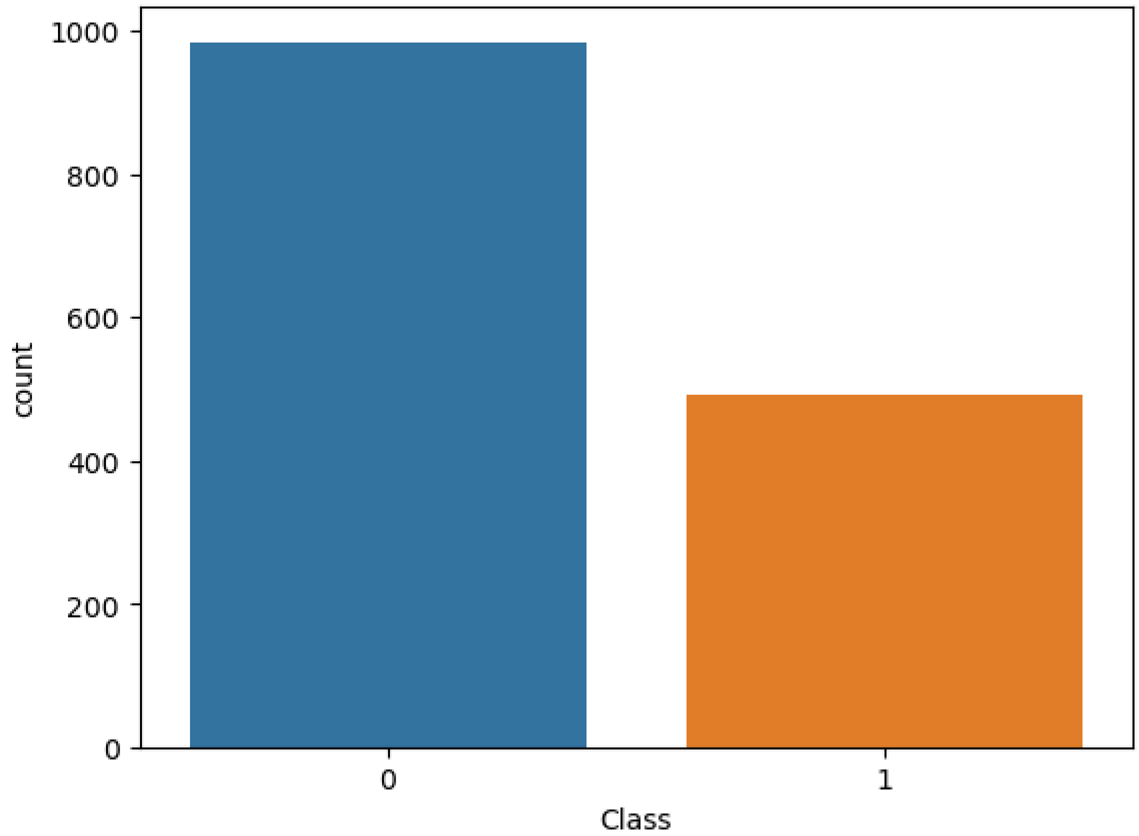
Balancing the Dataset： Under-sampling the Majority Class



In [143…
```
# Random Under-sampling

# Calculate number of instances in each class
count_class_0, count_class_1 = data['Class'].value_counts()

# Divide by class
df_class_0 = data[data['Class'] == 0]
df_class_1 = data[data['Class'] == 1]

# Random under-sampling
df_class_0_under = df_class_0.sample(count_class_1 * 2, random_state=42)  # Change multiplier to control class ratio

# Combine the data back
under_sample = pd.concat([df_class_0_under, df_class_1], axis=0)

# Shuffle the dataset to avoid any inherent ordering
under_sample = under_sample.sample(frac=1, random_state=42)
```

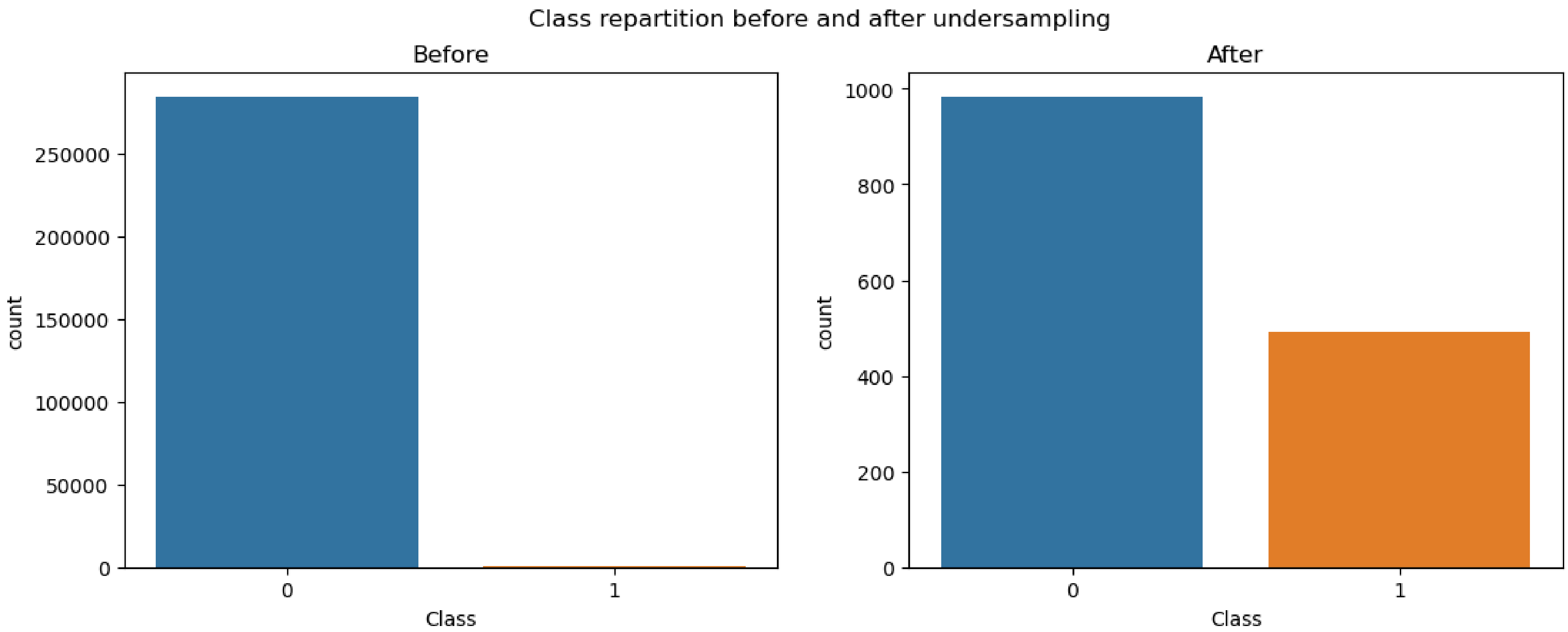
In [144…
```
sns.countplot(x="Class", data=under_sample)
```

Out[144]:
```
<Axes: xlabel='Class', ylabel='count'>
```



In [145…
```
#visualizing undersampling results
fig, axs = plt.subplots(ncols=2, figsize=(13,4.5))
sns.countplot(x="Class", data=data, ax=axs[0])
sns.countplot(x="Class", data=under_sample, ax=axs[1])

fig.suptitle("Class repartition before and after undersampling")
a1=fig.axes[0]
a1.set_title("Before")
a2=fig.axes[1]
a2.set_title("After")
```

Out[145]:
```
Text(0.5, 1.0, 'After')
```



Data Preprocessing

Standardation

In [146…
```
# Normalize 'Time' and 'Amount'
scaler = StandardScaler()
under_sample[['Time', 'Amount']] = scaler.fit_transform(under_sample[['Time', 'Amount']])
```

Split the Dataset

In [147…
```
# Separate features and target
X = under_sample.drop('Class', axis=1)
y = under_sample['Class']


# Split the dataset 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, stratify=y)
```

In [148…
```
X_train.shape
```

Out[148]:
```
(1180, 30)
```

In [149…
```
y_train.shape
```

Out[149]:
```
(1180,)
```

In [150…
```
X_test.shape
```

Out[150]:
```
(296, 30)
```

In [151…
```
y_test.shape
```

Out[151]:
```
(296,)
```

Building and Training the Neural Network

Assumptions and Limitations

- Assumptions: PCA-transformed features sufficiently capture the dynamics necessary for detecting fraud. The labels in the dataset are accurate and there are no misclassifications.
- Limitations: PCA components are not interpretable, which means understanding which features most influence model decisions is difficult. Models trained on past data may not perform well on new, unseen fraudulent tactics.

Multilayer Neural Network with Tensorflow/Keras

We'll build a simple neural network using TensorFlow and Keras for binary classification.

- Implement Dropout: Dropout is a regularization technique where randomly selected neurons are ignored during training. This means they are “dropped-out” randomly. This technique forces the network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons.
- Use Weight Regularization: Apply L1 or L2 regularization, which adds a penalty on the size of the weights to the loss function. This discourages learning overly complex models.
- Batch Normalization: This technique normalizes the input layer by adjusting and scaling activations. It can help to speed up training and has some regularization effects.
- Early Stopping： Stop training as soon as the validation performance starts deteriorating, despite improvements in training performance.

In [207…
```
#train the model
model = Sequential()
model.add(Dense(32, input_shape=(30,), activation='relu')),
model.add(Dense(8, activation='relu')),
model.add(Dense(4, activation='relu')),
model.add(Dense(1, activation='sigmoid'))
```

```
/Users/boyazeng/anaconda3/lib/python3.11/site-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [243…
```
#train the model
model = Sequential()
model.add(Dense(32, input_shape=(30,), activation='relu', kernel_regularizer=l2(0.01))),
model.add(BatchNormalization()),
model.add(Dropout(0.5)),
model.add(Dense(16, activation='relu', kernel_regularizer=l2(0.01))),
model.add(Dropout(0.5)),
model.add(Dense(8, activation='relu', kernel_regularizer=l2(0.01))),
model.add(Dropout(0.5)),
model.add(Dense(4, activation='relu', kernel_regularizer=l2(0.01))),
```

```
model.add(Dropout(0.5)),
model.add(Dense(1, activation='sigmoid'))
```

```
/Users/boyazeng/anaconda3/lib/python3.11/site-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

In [244…

```
opt = tf.keras.optimizers.Adam(learning_rate=0.001) #optimizer
# Compile the model
model.compile(optimizer=opt, loss=tf.keras.losses.BinaryCrossentropy(), metrics=['accuracy']) #metrics
```

Implementing Random Search and Early Stopping

In [245…

```
from kerastuner.tuners import RandomSearch

# Define the model building function for Keras Tuner
def build_model(hp):
    model = Sequential()
    model.add(Dense(units=hp.Int('units', min_value=32, max_value=512, step=32),
                    activation='relu', input_dim=X_train.shape[1]))
    model.add(Dense(units=hp.Int('units', min_value=32, max_value=512, step=32),
                    activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(optimizer=tf.keras.optimizers.Adam(
        hp.Float('learning_rate', min_value=1e-4, max_value=1e-2, sampling='LOG')),
        loss='binary_crossentropy',
        metrics=['accuracy'])
    return model

# Initialize RandomSearch
tuner = RandomSearch(
    build_model,
    objective='val_accuracy',
    max_trials=5,  # Set a low number for demonstration; increase it for real use
    executions_per_trial=3,
    directory='output',
    project_name='CreditCardFraud'
)

# Early stopping callback
early_stop = EarlyStopping(monitor='val_loss',  min_delta=0, patience=5, verbose=1, restore_best_weights=True)

# Execute the search
tuner.search(X_train, y_train, epochs=50, validation_data=(X_test, y_test), callbacks=[early_stop])

# Get the best model
best_model = tuner.get_best_models(num_models=1)[0]
```

```
Reloading Tuner from output/CreditCardFraud/tuner0.json
/Users/boyazeng/anaconda3/lib/python3.11/site-packages/keras/src/saving/saving_lib.py:415: UserWarning: Skipping variable loading for optimizer 'adam', because it has 2 variables whereas the saved optimizer has 14 variables.
  saveable.load_own_variables(weights_store.get(inner_path))
```

In [246…

```
#earlystopper = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy', min_delta=0, patience=15, verbose=1,mode='au
```

In [247…

```
# Fit the best model
history = best_model.fit(X_train, y_train, epochs=50, validation_data=(X_test, y_test), callbacks=[early_stop],valida
history_dict = history.history
```

```
Epoch 1/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9577 - loss: 0.1610 - val_accuracy: 0.9696 - val_loss: 0.0955
Epoch 2/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 922us/step - accuracy: 0.9529 - loss: 0.1299 - val_accuracy: 0.9696 - val_loss: 0.0701
Epoch 3/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9677 - loss: 0.0785 - val_accuracy: 0.9764 - val_loss: 0.0654
Epoch 4/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9750 - loss: 0.0680 - val_accuracy: 0.9764 - val_loss: 0.0643
Epoch 5/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 988us/step - accuracy: 0.9788 - loss: 0.0545 - val_accuracy: 0.9797 - val_loss: 0.0642
Epoch 6/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9846 - loss: 0.0478 - val_accuracy: 0.9797 - val_loss: 0.0637
Epoch 7/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9876 - loss: 0.0493 - val_accuracy: 0.9797 - val_loss: 0.0642
Epoch 8/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 998us/step - accuracy: 0.9889 - loss: 0.0434 - val_accuracy: 0.9797 - val_loss: 0.0650
Epoch 9/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 960us/step - accuracy: 0.9919 - loss: 0.0326 - val_accuracy: 0.9797 - val_loss: 0.0665
Epoch 10/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 989us/step - accuracy: 0.9891 - loss: 0.0303 - val_accuracy: 0.9797 - val_loss: 0.0676
Epoch 11/50
37/37 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9902 - loss: 0.0397 - val_accuracy: 0.9797 - val_loss: 0.0691
Epoch 11: early stopping
Restoring model weights from the end of the best epoch: 6.
```

In [248…

```
# history = model.fit(X_train.values, y_train.values, epochs = 6, batch_size=5, validation_split = 0.15, verbose = 0,
#                      callbacks = [earlystopper])
#history_dict = history.history
```

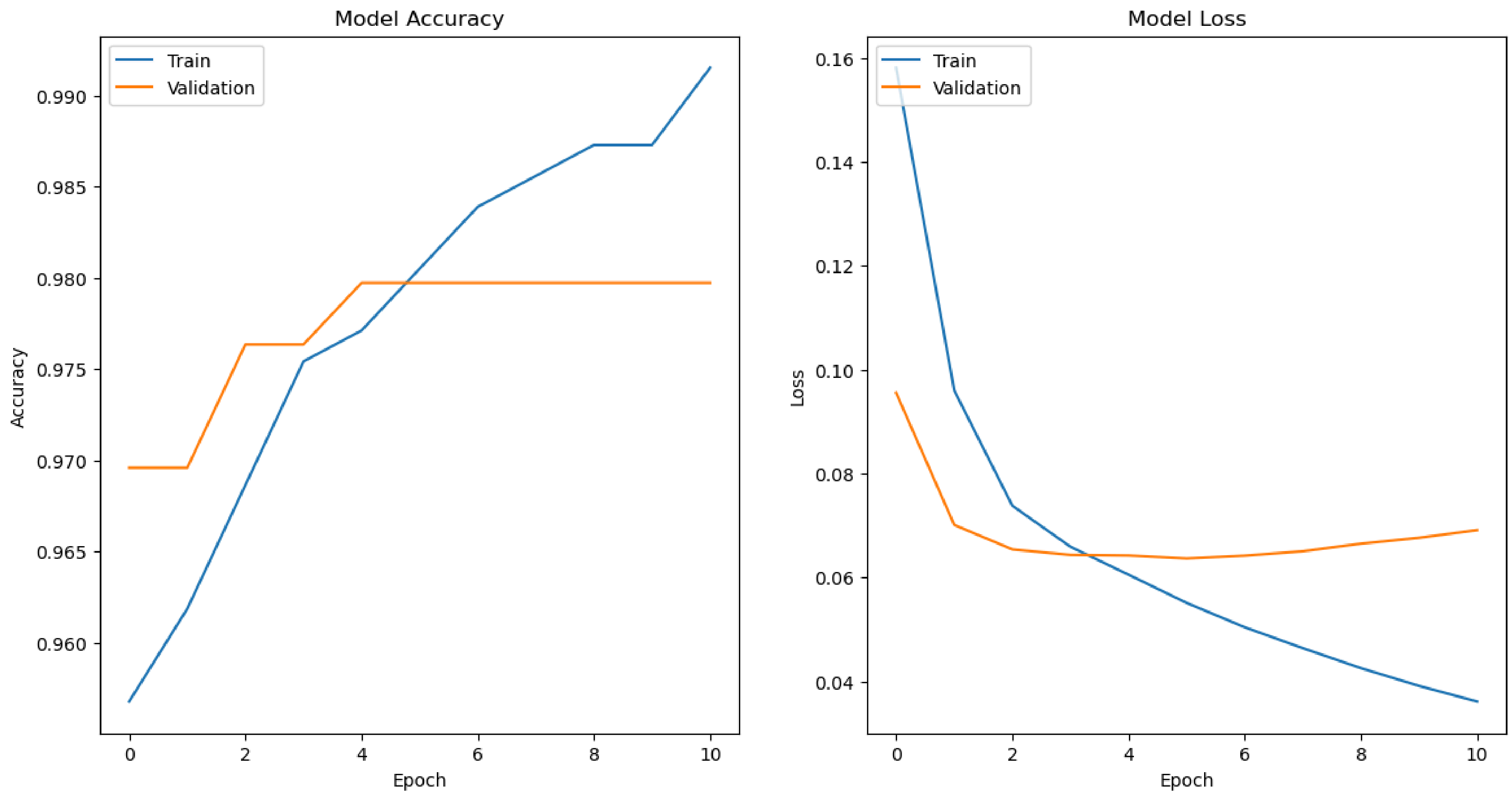
Model Evaluation

In [249…
```
# Evaluate the model
loss, accuracy = best_model.evaluate(X_test, y_test)
print("Loss:", loss)
print("Accuracy:", accuracy)
```
```
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 636us/step - accuracy: 0.9886 - loss: 0.0502
Loss: 0.06365001201629639
Accuracy: 0.9797297120094299
```

In [250…
```
# Predict probabilities
y_pred_proba = best_model.predict(X_test)

# Predict classes
y_pred = (y_pred_proba > 0.5).astype(int)

# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')

# Precision, Recall, F1-Score
precision, recall, f1_score, _ = precision_recall_fscore_support(y_test, y_pred, average='binary')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1-Score: {f1_score}')
```
```
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
Accuracy: 0.9797297297297297
Precision: 0.979381443298969
Recall: 0.9595959595959596
F1-Score: 0.9693877551020408
```

Overfitting/Underfitting:

In [251…
```
# Plot accuracy and loss
plt.figure(figsize=(14, 7))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train')
plt.plot(history.history['val_accuracy'], label='Validation')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='upper left')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train')
plt.plot(history.history['val_loss'], label='Validation')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper left')
plt.show()
```



Accuracy Graph Analysis:

- Training Accuracy increases consistently, which is indicative of the model learning from the training data.
- Validation Accuracy also increases and closely follows the training accuracy. This is generally a good sign as it suggests that the model is generalizing well on unseen data.

- Gap Between Training and Validation Accuracy: The small gap between the training and validation accuracy is favorable as it implies that there is no significant overfitting. Both curves are ascending, which suggests that the model could potentially improve with more training epochs.

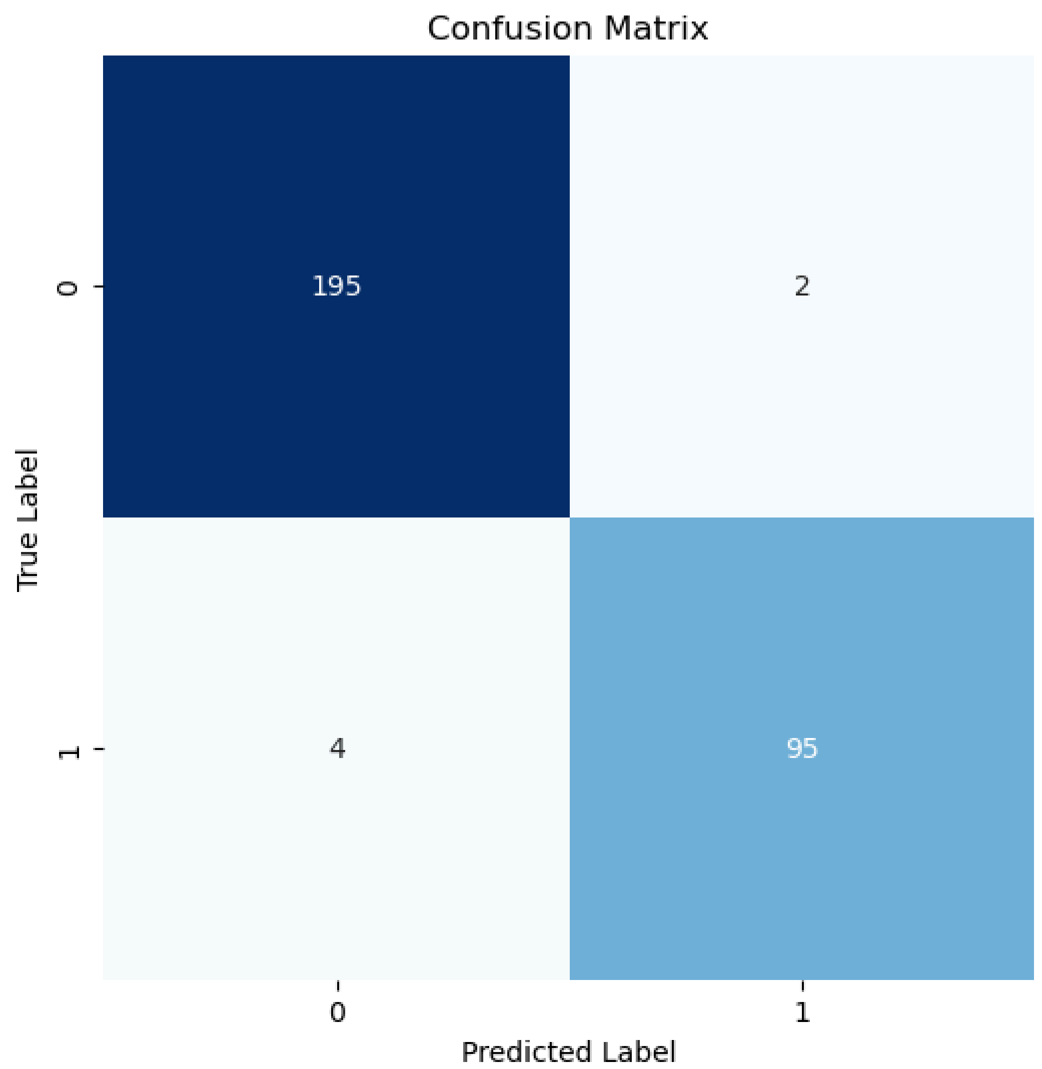
Loss Graph Analysis:

- Training Loss decreases steadily, showing that the model is increasingly fitting the training data better.
- Validation Loss decreases initially and then starts to increase slightly towards the latest epochs. This upward trend in validation loss while the training loss continues to decrease could be an early sign of overfitting. The model may be starting to learn specifics about the training data that do not generalize to the validation data.

In [198…

```
# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(6, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", cbar=False)
plt.title('Confusion Matrix')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()
```

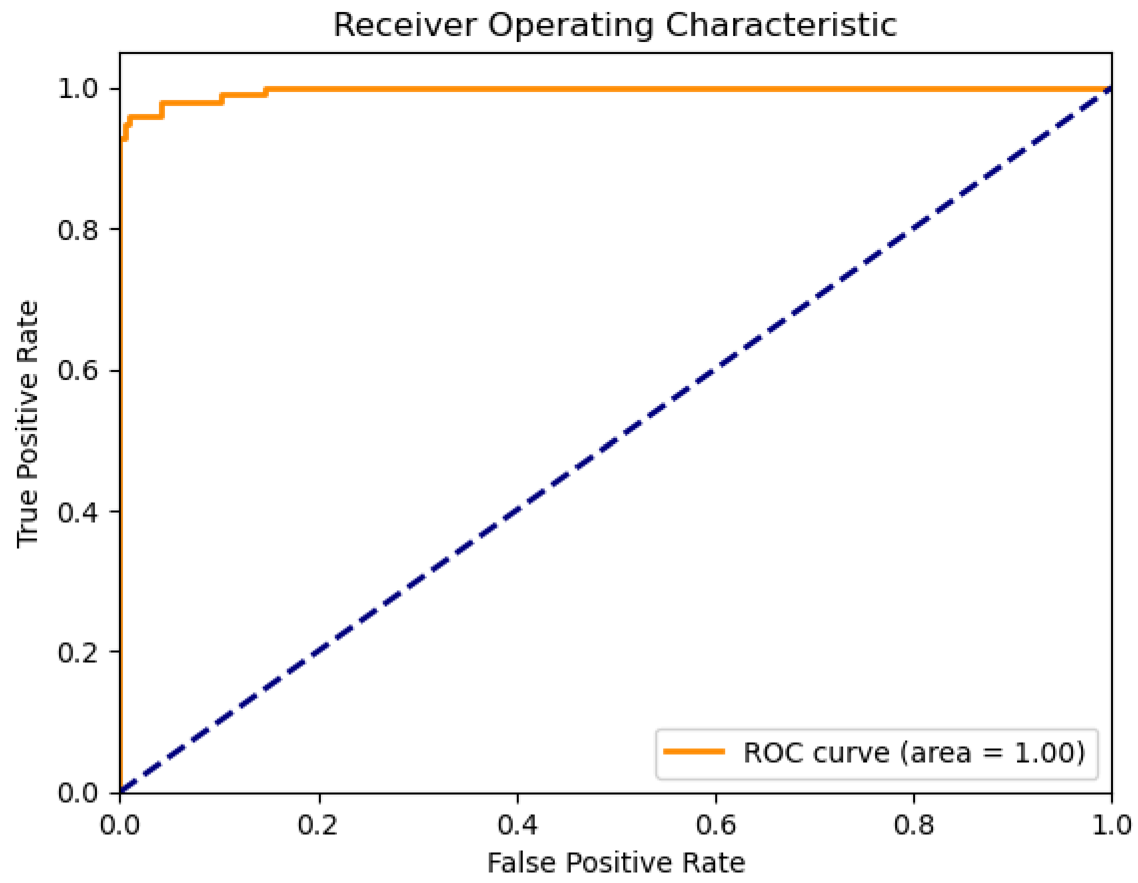
```
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
```



The confusion matrix indicates that the model has performed well in classifying the classes. It shows a good balance of precision and recall, with few false positives and false negatives. However, the false negatives (Type II errors) could be critical depending on the application since they represent missed fraud detections.

In [205…

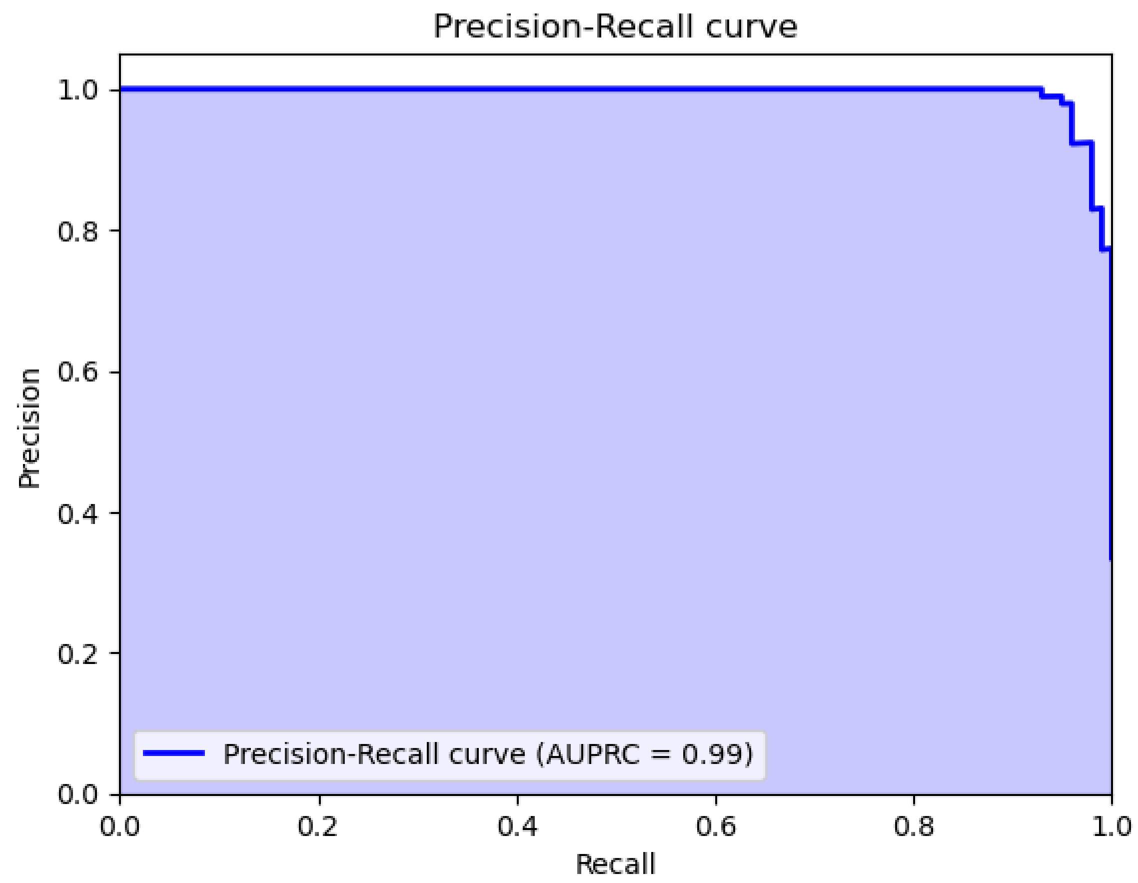
```
# ROC curve and AUC
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```



The ROC curve shows an excellent performance with an AUC of 1.00. This suggests perfect discrimination between positive and negative classes. However, such a perfect score may sometimes be suspect in practical scenarios, as it could imply overfitting especially if the test data is not representative or similarly 'easy' as the training set.

In [200…

```
# Precision-Recall curve and AUPRC
precision, recall, _ = precision_recall_curve(y_test, y_pred_proba)
auprc = average_precision_score(y_test, y_pred_proba)
plt.figure()
plt.plot(recall, precision, color='blue', lw=2, label='Precision-Recall curve (AUPRC = %0.2f)' % auprc)
plt.fill_between(recall, precision, step='post', alpha=0.2, color='b')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.title('Precision-Recall curve')
plt.legend(loc="lower left")
plt.show()
```



The Precision-Recall curve also shows excellent performance, nearly reaching the top right corner, with an AUPRC close to 1 (0.99). This indicates both high recall and high precision, demonstrating the model's ability to handle the imbalanced data effectively.

In [ ]:

```
# # Save the model
# best_model.save('autoencoder_fraud_detection.h5')

# # Load the model (in a different file or system)
# from tensorflow.keras.models import load_model
# loaded_model = load_model('autoencoder_fraud_detection.h5')
```

In [ ]: