Credit Card Fraud Detection Dataset 2023

Here we look at the problem of detecting creditcard payment fraud. The dataset contains more than 550,000 credit card transactions made by European cardholders in the year 2023. The data has been anonymized to protect the cardholders' identities.

The objective is to use such historical data to develop a prediction algorithm capable of classifying new transactions as *fraud* or *not fraud*. Both Supervised and Unsupervised machine learning algorithms will be performed.

Authors:

- Federica Stefanizzi
- Matteo Greco
- Samuele Spagliardi
- Ayman Bizzou

Simplicity Alert: Just for simplicity, these are all the Python libraries that have been used during the task.

In [81]:
```
# Libraries for data manipulation
import numpy as np
import pandas as pd

# Libraries for machine learning
from sklearn.model_selection import (
    train_test_split,
    cross_val_score,
    StratifiedKFold,
)
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    accuracy_score,
    confusion_matrix,
    classification_report,
    precision_score,
    recall_score,
    f1_score,
    mean_squared_error,
    roc_curve,
    roc_auc_score,
    auc,
    precision_recall_curve,
    average_precision_score,
)
import matplotlib.pyplot as plt
import seaborn as sns
from tabulate import tabulate
import time
import random

# Libraries for specific algorithms
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import (
    RandomForestClassifier,
    BaggingClassifier,
    GradientBoostingClassifier,
    AdaBoostClassifier,
)
from sklearn.linear_model import LinearRegression, LogisticRegressi
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import LinearSVC, SVC, OneClassSVM
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier, MLPRegressor
from sklearn.datasets import make_classification, make_regression
from sklearn.decomposition import PCA
from sklearn.cluster import (
    KMeans,
    AgglomerativeClustering,
    DBSCAN,
)
from sklearn.metrics import (
    homogeneity_score,
    completeness_score,
    v_measure_score,
    silhouette_score,
    calinski_harabasz_score,
    ConfusionMatrixDisplay,
)
from sklearn.calibration import CalibrationDisplay, CalibratedClass
from sklearn.ensemble import IsolationForest

import os
from matplotlib.gridspec import GridSpec
```

Seeding

To guarantee consistent and reproducible results, a random seed is commonly employed. Seeding a random number generator ensures that each time the code is rerun, the generated random samples remain constant.

In [82]:
```
random_seed = 42
```

In [83]:
```
random.seed(random_seed)
np.random.seed(random_seed)
```

Data Preparation

In this section, data preparation steps will be performed.

Data Loading

The provided data is in a *Comma Separated Values* (CSV) file. **Fortunately**, Pandas offers built-in functions to load these files automatically.

In [84]:
```
df = pd.read_csv('/Users/federicastefanizzi/Downloads/creditcard_20
```

One of the most important prerequisites when dealing with substantial datasets like this one is to diligently examine the data prior to commencing any analysis, ensuring a comprehensive understanding before engaging in further manipulation or exploration.

In [85]:
```
df.sample(10)
```

Out[85]:

	id	V1	V2	V3	V4	V5	V6	V7
437378	437378	0.420468	-0.070194	-0.569266	0.191673	-0.009607	0.426903	-0.356728
504222	504222	-0.238944	0.250929	-0.374408	0.152938	-0.105008	-0.039028	-0.293004
4794	4794	-0.117796	-0.147961	2.130455	-0.325762	0.325616	0.271351	0.772625
388411	388411	-0.855315	0.137014	-0.628116	0.613733	-0.643573	-0.664283	-0.880040
424512	424512	0.257686	0.035247	-0.203112	0.506745	-0.242235	-0.192608	-0.289297
123536	123536	-0.413357	-0.450377	1.184996	-0.378135	1.009580	1.385973	0.525802
333319	333319	-0.905109	1.090565	-1.187502	1.927398	-1.209692	-1.471852	-1.480131
369666	369666	0.517429	-0.016506	-0.225822	0.437507	0.045921	0.111977	-0.120557
62882	62882	-0.330014	-0.161835	1.288293	-0.591877	0.289209	-0.283643	0.623377
414847	414847	-1.998296	2.563713	-1.922748	2.220722	-2.315012	0.163451	-2.788664

10 rows × 31 columns

In the provided sample of 10 rows from the dataset, it's noteworthy that there are 31 columns in total. Out of these, 28 columns have been anonymized to safeguard the privacy of individuals involved in credit card transactions. The three known features are the transaction ID (cumulative number), the transaction amount, and the class label denoted as "Class," distinguishing between **fraudulent (1)** and **non-fraudulent (0)** transactions.

Summary Statistics

This section will provide a concise overview of key numerical characteristics within the dataframe.

How many rows are there in the dataframe?

```
In [86]: len(df)
Out[86]: 568630
```

How many columns are there in the dataframe?

```
In [87]: len(df.columns)
Out[87]: 31
```

(as previously mentioned)

What are the names and datatypes in each column?

```
In [88]: df.columns
Out[88]: Index(['id', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9',
'V10',
       'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V1
9', 'V20',
       'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Am
ount',
       'Class'],
      dtype='object')
```

In [89]:
```
df.dtypes
```

Out[89]:
```
id          int64
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
```

This output provides the data types for each column in the dataframe df:

Columns 'id' and 'Class' have data type **int64**. Columns 'V1' through 'V28' and 'Amount' have data type **float64**. The float64 data type indicates that these columns store numerical values with decimal points, while int64 indicates integer values. This summary is valuable for understanding the nature of the data types within the dataframe and is essential for subsequent data analysis and processing.

Notice: The disclosed data types, uniquely consisting of float64 for numerical values and int64 for integers in the dataframe, indicate the absence of categorical variables. This absence can be presumably attributed to the anonymization process applied to the majority of features, aligning with common practices to safeguard privacy and confidentiality in sensitive datasets.

Descriptive Statistics for the Dataframe Columns

In [90]:
```
for column in df.columns:
    desc_df = df[column].describe()
    print(f"description for column '{column}': \n\n{desc_df}\n")
```

```
description for column 'id':

count    568630.000000
mean     284314.500000
std      164149.486122
min           0.000000
25%      142157.250000
50%      284314.500000
75%      426471.750000
max      568629.000000
Name: id, dtype: float64

description for column 'V1':

count    5.686300e+05
mean    -5.638058e-17
std      1.000001e+00
min     -3.495584e+00
25%     -5.652859e-01
```

id: This column functions as a unique identifier for transactions, with values ranging from 0 to 568629 in sequential order. As a transaction ID, its numerical characteristics, such as the mean, may not hold substantive relevance, as the values are assigned sequentially rather than carrying inherent numerical significance. The primary utility of this column lies in uniquely identifying individual transactions within the dataset.

V1 - V28: These columns exhibit characteristics indicative of standardization. Their means all approximate to 0 and standard deviations are all near 1. The varying ranges from minimum to maximum values across columns could imply diverse scales or units, but nothing can be stated certainly due to their anonymized nature. Quartiles offer insights into the distribution of values, aiding in understanding the spread and central tendencies of these features.

Amount: The 'Amount' column represents the monetary value associated with transactions. Descriptive statistics such as mean, median, and quartiles provide a glimpse into the central tendency and dispersion of transaction amounts. This information is crucial for understanding the distribution of monetary values within the dataset.

Class: The 'Class' column is a binary categorical variable (0 or 1), typically used for classification tasks, such as distinguishing between fraudulent and non-fraudulent transactions. The mean of 0.5 indicates a balanced distribution of classes, *suggesting an equal representation of both fraud and non-fraud instances in the dataset*. This balance is pivotal for training robust machine learning models.

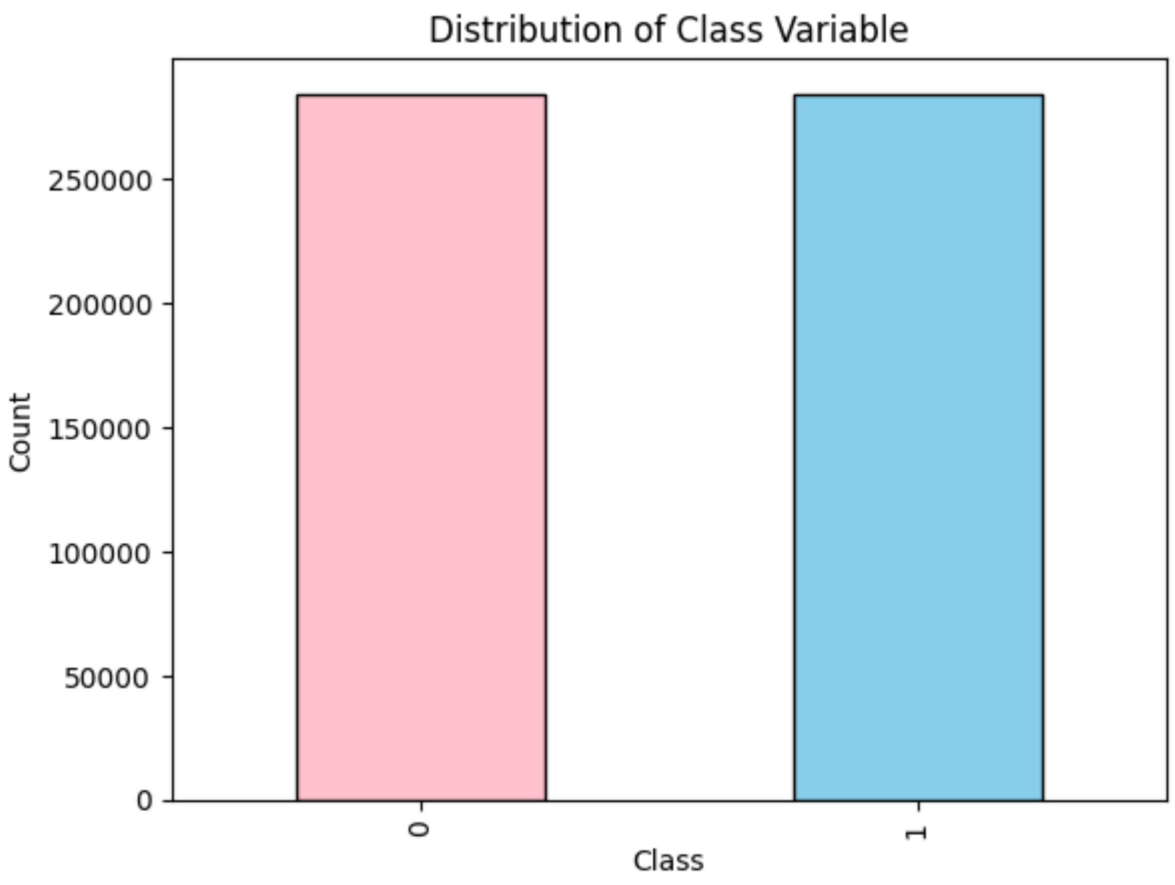
Focus on 'id' column: The 'id' column is just an index identifying the rows, and it doesn't carry any significance in the analysis or decision-making process. Since it doesn't represent any meaningful information about the data, removing it will make the dataframe cleaner and more efficient. Notably, it was observed that all transactions are denoted by an incrementally increasing ID, and a significant shift in class type occurs halfway through the dataset. This pattern suggests that the ID column, rather than being a useful feature, may introduce bias or spurious correlations in the model training process.

```
In [91]: df = df.drop('id', axis=1)
```

Data Visualization

A Glimpse Into the Target Variable 'Class'

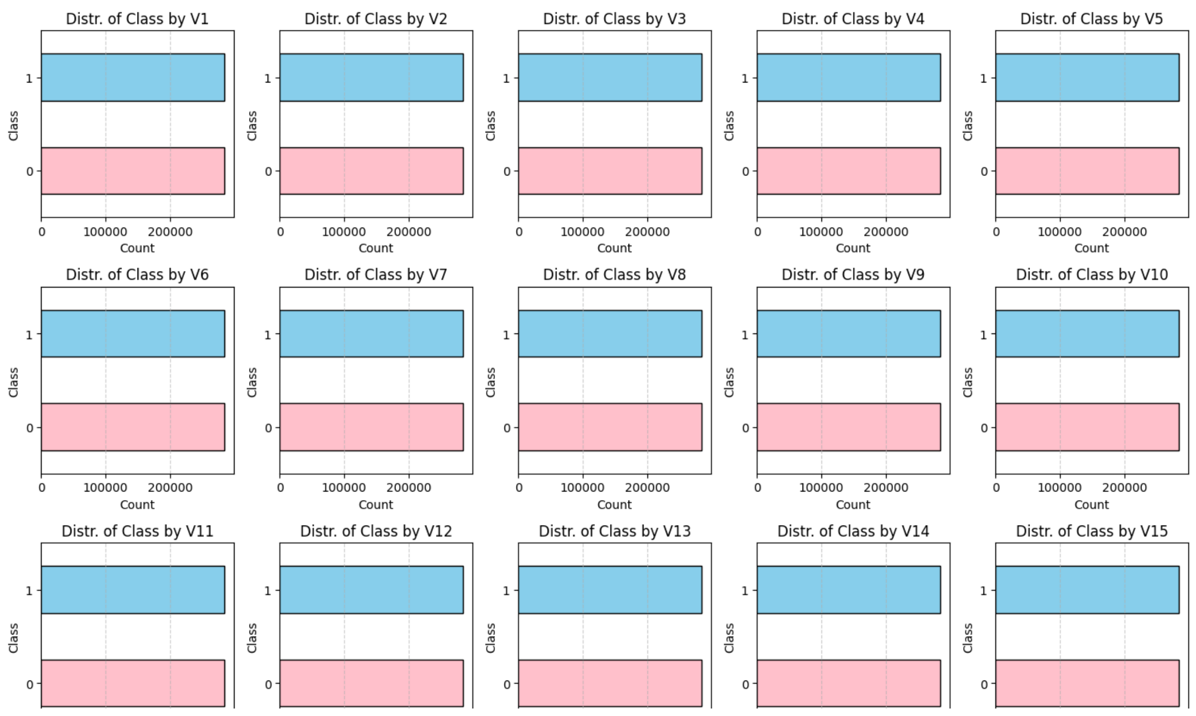
In [92]:
```
# Bar plot for the 'Class' variable
df['Class'].value_counts().plot(kind='bar', color=['pink', 'skyblue
plt.title('Distribution of Class Variable')
plt.xlabel('Class')
plt.ylabel('Count')
plt.show()
```



The equal representation of both fraud and non-fraud instances in the dataset, as indicated by the mean of the 'Class' column being 0.5 and by the barplot is essential for training machine learning models, especially for binary classification tasks like fraud detection. An equal representation helps prevent the model from being biased toward the usual majority class (non-fraud), ensuring that it learns to recognize patterns in both classes effectively. This balance contributes to a more robust and fair model, enhancing its ability to generalize well to new, unseen data.

However, more exploration will follow through the assignment, and we will evaluate how the algorithm works with imbalanced class instances in a new, highly imbalanced dataset. Imbalanced class distributions, where one class significantly outnumbers the other, can pose challenges for model training and evaluation. Therefore, further investigation will be conducted to assess the model's performance under different class distribution scenarios.

In [93]:
```
columns_to_plot = ['V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8',
                   'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17',
                   'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27',

# Calculate the number of rows and columns for subplots
num_plots = len(columns_to_plot)
num_cols = 5
num_rows = (num_plots + num_cols - 1) // num_cols

fig, axes = plt.subplots(num_rows, num_cols, figsize=(14, 3*num_row

axes = axes.flatten()

# Iterate through the specified columns and plot horizontal bar cha
for i, column in enumerate(columns_to_plot):
    ax = axes[i]
    df['Class'].value_counts().plot(kind='barh', color=['pink', 'sk
    ax.set_title(f'Distr. of Class by {column}')
    ax.set_xlabel('Count')
    ax.set_ylabel('Class')
    ax.xaxis.grid(True, linestyle='--', alpha=0.6)

for i in range(num_plots, num_cols * num_rows):
    fig.delaxes(axes[i])

# Adjust layout
plt.tight_layout()
plt.show()
```

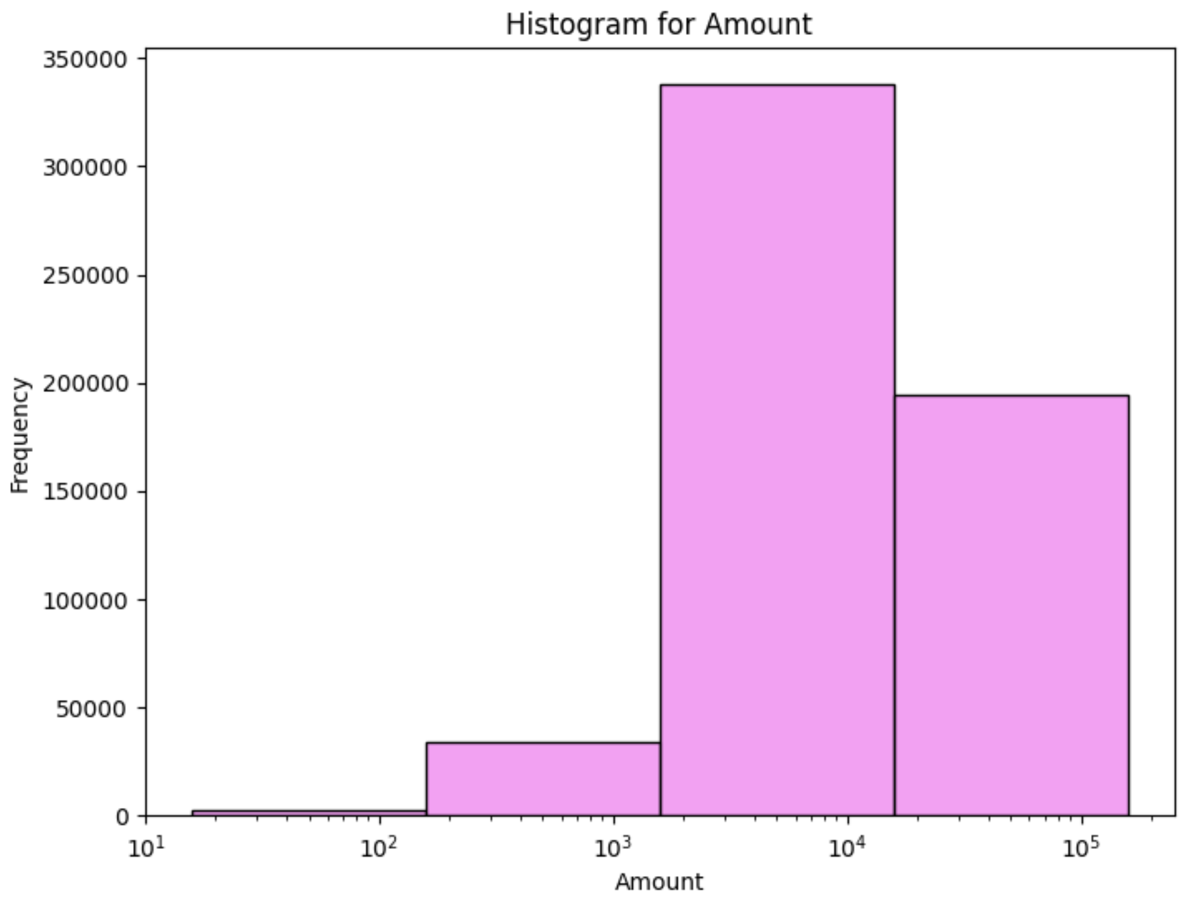


The plots above depict the distribution of the 'Class' variable across various columns. Each column represents an even distribution of the target variable, indicating that there is no necessity to eliminate any of them due to class distribution issues.

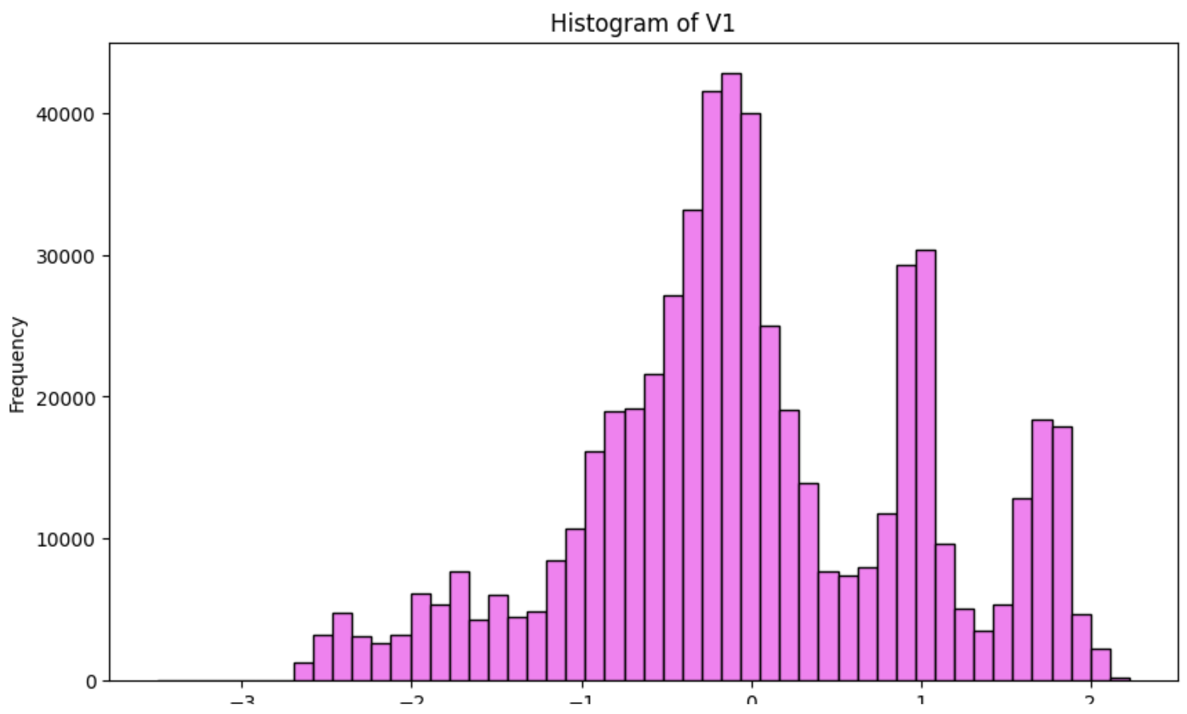
Histograms for Feature Visualization

In [328]:
```
# Histogram for 'Amount'
fig = plt.figure()
fig = plt.figure(figsize=(8, 6))
sns.histplot(data=df, x='Amount', discrete=True, log_scale=(True, F
plt.xlabel('Amount')
plt.ylabel('Frequency')
plt.title('Histogram for Amount')
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



In [329]:
```
# Histograms for selected 'V' variables (V1, V2, V3 in this example
selected_v_columns = ['V1', 'V2', 'V3']
for column in selected_v_columns:
    plt.figure(figsize=(10, 6))
    plt.hist(df[column], bins=50, color='violet', edgecolor='black'
    plt.title(f'Histogram of {column}')
    plt.xlabel(column)
    plt.ylabel('Frequency')
    plt.show()
```



The presented histograms provide a visual exploration of the distribution of the **'Amount'** feature and the first three anonymized features, denoted as **'V1,' 'V2,' and 'V3'**. Analyzing these visual representations, it becomes apparent that 'V2' and 'V3' exhibit patterns indicative of a normal distribution, where data points are concentrated around the mean and gradually taper off towards the extremes. This normal distribution suggests a symmetrical and bell-shaped curve for these features. On the other hand, interpreting the distribution of 'Amount' and 'V1' is less straightforward. The histogram for 'Amount' seems left-skewed, indicating a concentration of greater transaction amounts with a tail stretching towards smaller values. Meanwhile, 'V1' appears to have a more complex distribution that doesn't strictly adhere to a normal pattern, even though it presents a high concentration of points around the mean.

In [332]:
```
transactions_with_zero_amount =df[df['Amount'] == 0]
print(transactions_with_zero_amount)
```

```
Empty DataFrame
Columns: [V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13,
V14, V15, V16, V17, V18, V19, V20, V21, V22, V23, V24, V25, V26, V
27, V28, Amount, Class]
Index: []

[0 rows x 30 columns]
```

In this dataframe, there are no transactions with a 0 value for 'Amount'

Data Preprocessing: Dealing With Duplicates

The following code provides a summary of the distribution of unique values in each column, along with their respective counts.

In [169]:
```
for column in df.columns:
    val_c = df[column].value_counts()
    print(f"value counts for column '{column}': \n\n{val_c}\n")
```

```
value counts for column 'V1':

V1
-1.704517    3539
-2.459141    1399
-1.161336     259
-1.065576     255
-1.522526     246
             ...
-0.272852       1
 1.822123       1
 0.556931       1
 0.276802       1
-0.795144       1
Name: count, Length: 552035, dtype: int64

value counts for column 'V2':

V2
```

Assumption Alert: While inspecting the counts for the anonymized features, notice that certain rows have identical values across different columns. To address this, these identical rows have to be eliminated, based on the assumption that rows exhibiting identical values across all 'V' columns, even if they differ in the 'Amount' column, should be excluded.

In [170]:
```
columns_to_exclude = ['Amount']  # Exclude 'Amount' column from com
v_columns = [col for col in df.columns if col.startswith('V') and c

# Drop duplicates based on 'V' columns
df_no_duplicates = df.drop_duplicates(subset=v_columns, keep='first
```

In [171]:
```
original_rows, original_cols = df.shape #assigning the number of ro

# Calculate the number of eliminated rows
eliminated_rows = original_rows - df_no_duplicates.shape[0] #addres

# Print the number of eliminated rows
print(f'Number of eliminated rows: {eliminated_rows}')
```

```
Number of eliminated rows: 16595
```

In [172]:
```
df = df_no_duplicates
```

Data Preprocessing: Dealing with Correlated Features

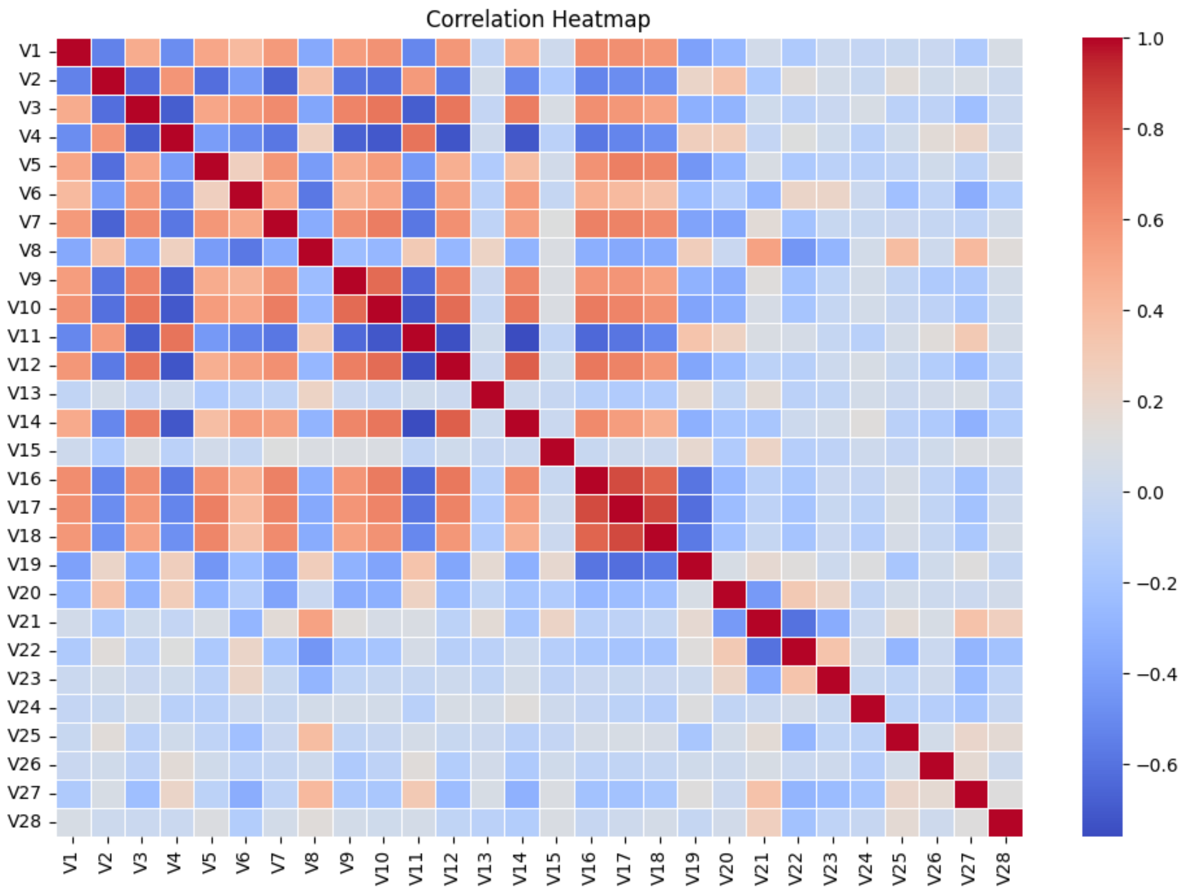
In this section, the intricacies of correlated features will be addressed, employing visualizations and methodologies to enhance the understanding and optimization of the dataset. To better visualize the interrelationships among features a correlation heatmap will be used, thus providing a graphical representation of correlation strengths.

Assumption Alert: An assumption is made that 'strong correlation' is identified when the correlation coefficient surpasses a threshold of 0.8, encompassing both positive and negative correlations. Consequently, all the features that will show a positive correlation > 0.8 and a negative correlation < -0.8 will be removed.

Step 1: Calculate the Correlation Matrix

A correlation matrix (in this case displayed through a correlation heatmap for better visualization) is a table that displays the correlation coefficients between many variables. Each cell in the table shows the correlation between two variables. The value is in the range of -1 to 1. If two variables have high correlation, it means when one variable changes, the second variable tends to change in a specific direction. If they have a low or negative correlation, it means that the variables are likely to change in opposite directions.

In [173]:
```
correlation_matrix = df[['V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7',
                         'V11', 'V12', 'V13', 'V14', 'V15', 'V16',
                         'V21', 'V22', 'V23', 'V24', 'V25', 'V26',

plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, cmap='coolwarm', annot=False, fmt="
plt.title('Correlation Heatmap')
plt.show()
```



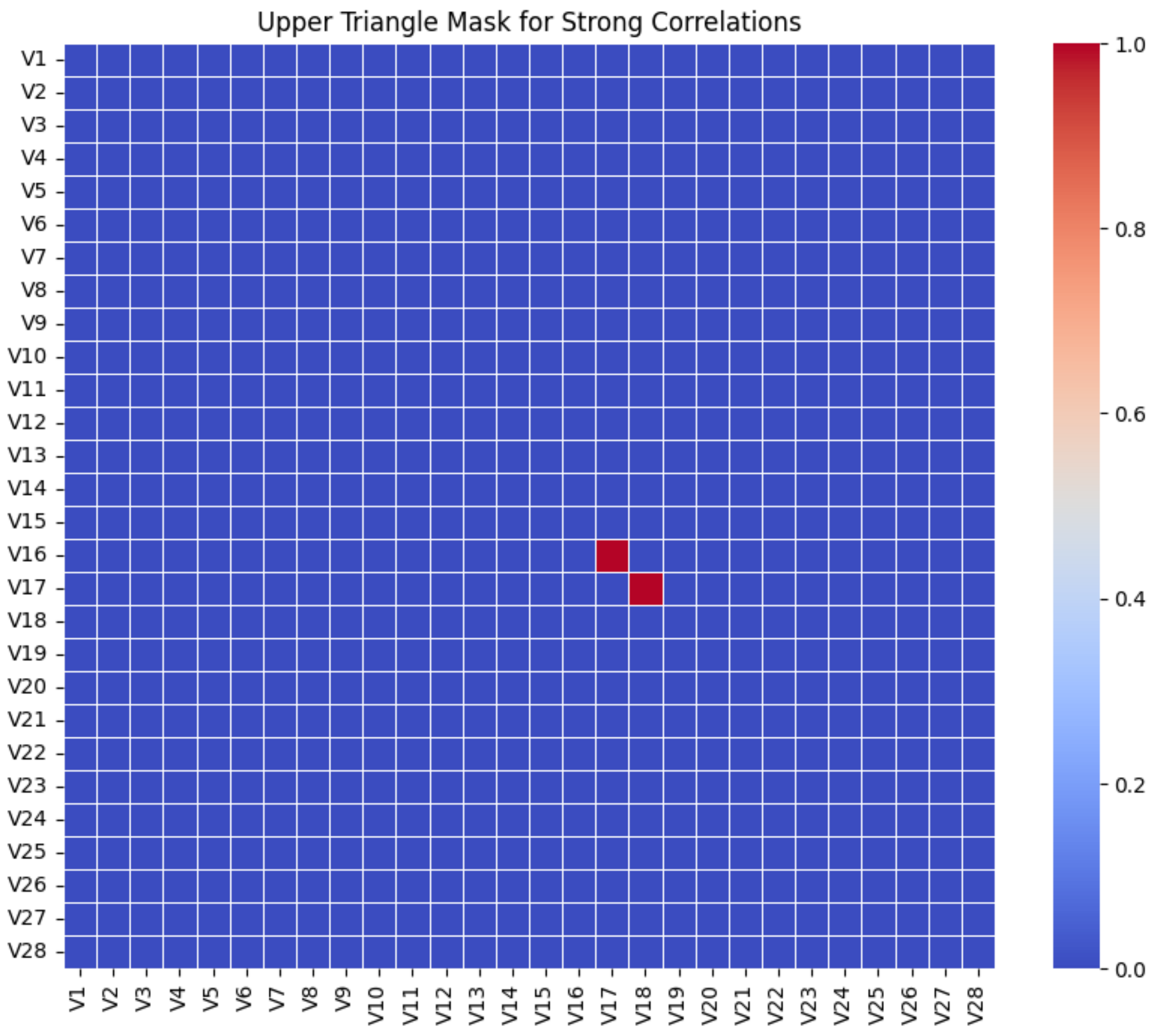
Step 2: Create a Mask for the Upper Triangle with Both Positive and Negative Correlations

In [174]:
```
upper_triangle_mask = np.triu((correlation_matrix > 0.8) | (correla
```

This line of code is designed to create a binary mask, often referred to as an **upper triangle mask**, specifically idealized for analyzing a correlation matrix. In the context of the correlation matrix, the goal is to identify and isolate the upper triangular portion of this matrix where the correlations are either notably positive (greater than 0.8) or strongly negative (less than -0.8). By converting the values in the matrix to a boolean type, the code ensures that the value 'True' represents positions in the upper triangle where the correlation is either greater than 0.8 or less than -0.8.

Notice: Only the upper triangle of the correlation matrix is considered when dealing with features. This is done to avoid redundancy and ensure that each pair of features is analyzed only once, as in a correlation matrix the lower triangle is a mirror image of the upper triangle (the correlation matrix is completely symmetrical with respect to the diagonal).

In [175]:
```
plt.figure(figsize=(10, 8))
sns.heatmap(upper_triangle_mask, cmap="coolwarm", annot=False, fmt=
plt.title("Upper Triangle Mask for Strong Correlations")
plt.show()
```



According to the upper triangle mask for strong correlations, the analysis indicates that only 2 features need to be removed from the dataset.

Convert the Boolean Mask to a DataFrame with the Same Index and Columns as Correlation Matrix

In [176]:
```
upper_triangle_df = pd.DataFrame(upper_triangle_mask, index=correla
```

This line converts the boolean mask into a dataframe. The new DataFrame has the same rows and columns as the original correlation matrix, but it contains True or False values based on whether the corresponding correlation was greater than 0.8 or less than -0.8.

In [177]:
```
upper_triangle_df.sample(10)
```

Out[177]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V19	V20	V2
V11	False	False	False	False	False	False	False	False	False	False	...	False	False	Fals
V26	False	False	False	False	False	False	False	False	False	False	...	False	False	Fals
V9	False	False	False	False	False	False	False	False	False	False	...	False	False	Fals
V6	False	False	False	False	False	False	False	False	False	False	...	False	False	Fals
V5	False	False	False	False	False	False	False	False	False	False	...	False	False	Fals
V28	False	False	False	False	False	False	False	False	False	False	...	False	False	Fals
V8	False	False	False	False	False	False	False	False	False	False	...	False	False	Fals
V15	False	False	False	False	False	False	False	False	False	False	...	False	False	Fals
V21	False	False	False	False	False	False	False	False	False	False	...	False	False	Fals
V3	False	False	False	False	False	False	False	False	False	False	...	False	False	Fals

10 rows × 28 columns

This step will set the stage for efficient subsequent operations in the process of managing correlated features.

Step 3: Find and Drop the Correlated Features, Excluding Specific Columns

In [178]:
```
columns_to_exclude = ['Class', 'Amount']
to_drop = [column for column in upper_triangle_df.columns if any(up
df_nc = df.drop(to_drop, axis=1) # Here nc stands for 'no correlati
```

These codes identify which columns in the original dataframe df have strong correlations. The columns listed in columns_to_exclude (here, 'Class' and 'Amount') are not considered for removal, due to their extreme relevance in the specific context of this task.

In [179]:
```
table = tabulate({"Columns Identified for Removal": to_drop}, heade
print(table)
```

```
Columns Identified for Removal
--------------------------------
V17
V18
```

The two columns identified for removal are V17 and V18. These columns were selected based on their strong correlation with other features in the dataset, surpassing the defined threshold of 0.8. The exclusion of these columns ensures that the remaining features in the dataset are more independent and less likely to introduce redundancy into the model.

Sample of the Resulting DataFrame

```
In [180]: df_nc.sample(10)
```

Out[180]:

	V1	V2	V3	V4	V5	V6	V7	
503464	-0.145397	0.006385	-0.555243	0.069120	0.456084	-1.223471	-0.009756	-0.0429
511638	-1.045146	-1.230559	-0.618296	0.237364	-0.059258	0.227270	-0.472652	0.3863
396034	-0.027575	0.892997	-1.019403	1.545567	0.226942	-1.277847	-0.336276	0.1530
131451	0.041983	-0.052917	1.545382	0.413735	0.113579	0.822822	0.331863	-0.0149
213008	-0.212088	-0.307199	0.240582	-0.652327	0.678867	0.043539	0.722871	-0.2593
18671	0.919359	-0.538646	1.077088	-0.002597	0.047772	0.937714	0.240045	-0.0629
488998	-1.610651	0.592321	-1.408555	1.890480	-0.514359	-2.454228	-2.576583	-0.0827
519871	-0.873272	1.031623	-0.851367	0.995529	-0.848004	-0.527854	-0.830799	0.6535
330666	1.254632	-0.169989	0.331540	0.590150	0.581801	0.452491	0.566339	-0.1886
400894	-0.502580	0.185146	-0.419529	0.530537	-0.558322	0.077892	-0.180788	0.2530

10 rows × 28 columns

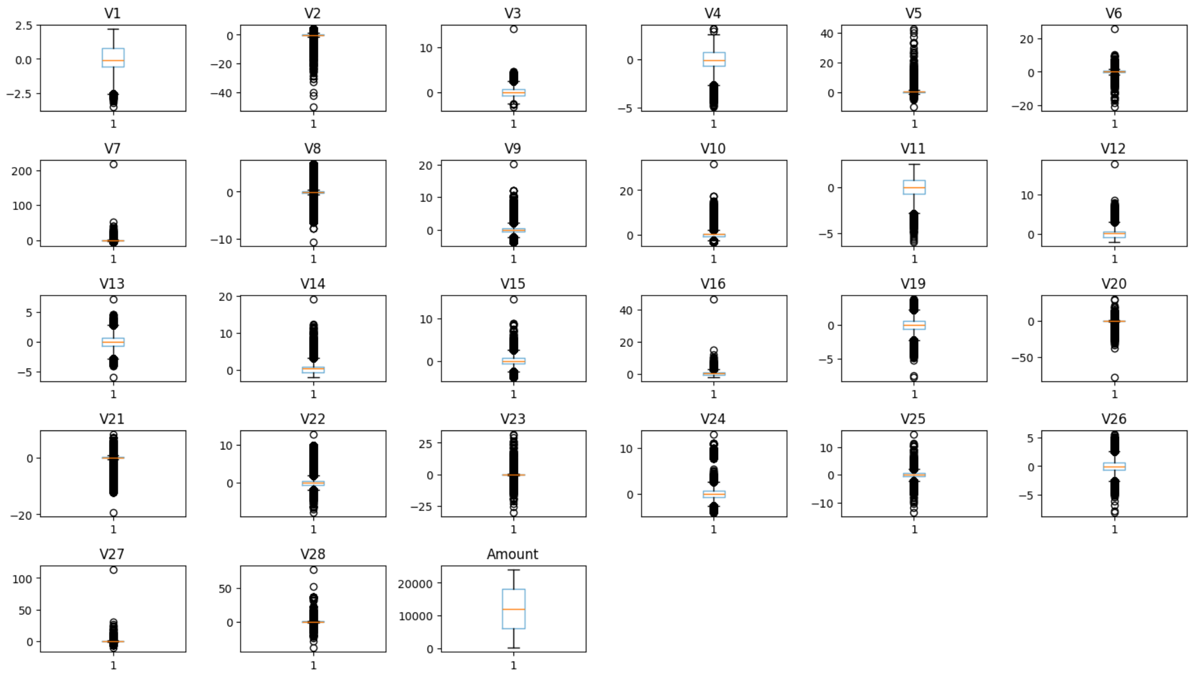
```
In [181]: df = df_nc
```

Dealing with Outliers

Boxplots

Box plots are statistical graphics that display the distribution of a dataset. Box plots show the median of the data as a line within the box, the interquartile range (IQR) as the box itself, and the data points outside the whiskers as potential outliers.

In [182]:
```
columns_to_remove = ['Class']
df_cleaned = df.drop(columns=columns_to_remove)

blues = plt.cm.Blues_r

plt.figure(figsize=(15, 10))  # Imposta le dimensioni della figura

for i, col in enumerate(df_cleaned.columns):
    plt.subplot(6, 6, i+1)
    plt.boxplot(df_cleaned[col], boxprops=dict(color=blues(0.5)))
    plt.title(col)

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



The presence of outliers can be due to variability in the measurement, or they could also be a result of the natural distribution of the data. In some cases, outliers can contain valuable information about the dataset or the process from which the data was generated. Without contextual information about the variables represented in these plots, it is impossible to determine the significance or the cause of these outliers. So, outliers will not be removed in this dataset.

Supervised Learning Models

Supervised machine learning serves as a fundamental concept in the vast domain of artificial intelligence. It constitutes an approach where models are trained to make predictions or decisions based on carefully labeled datasets. In this learning paradigm, a dataset is provided, comprising pairs of inputs and corresponding outputs. The inputs represent the distinctive features of the data, while the outputs signify the labeled target or response. The primary goal is to equip the model with the capability to discern inherent patterns, relationships, or mappings between these inputs and outputs through a dedicated training phase. Once effectively trained, the model utilizes its acquired knowledge to make predictions for new, unseen data. While supervised learning finds its application in various contexts, the focus here is specifically on its utilization for addressing the challenge of fraud detection in credit card transactions. This targeted application exemplifies the practical and impactful use of supervised learning in addressing real-world issues within the financial sector.

In the forthcoming subsections, a variety of supervised machine learning algorithms will be explored individually by computing their metrics (also observing how these metrics vary changing models' hyperparameters). Finally, the different models will be compared in order to identify the most effective and efficient among them.

Confusion Matrix:

	Predicted positive	**Predicted negative**
Labelled positive	TP	FN
Labelled negative	FP	TN

Building blocks for the metrics:

- $\text{TP} = \text{True Positive}$ (i.e., positive values correctly predicted as such)
- $\text{TN} = \text{True Negative}$ (i.e., negative values correctly predicted as such)
- $\text{FP} = \text{False Positive}$ (i.e., negative values predicted as positive)
- $\text{FN} = \text{False Negative}$ (i.e., positive values predicted as negative)

Metrics:

- $\text{accuracy} = \frac{\text{TP+TN}}{\text{TP+TN+FP+FN}}$
- $\text{precision} = \frac{\text{TP}}{\text{TP+FP}}$ (a.k.a. positive predictive value)
- $\text{recall} = \frac{\text{TP}}{\text{TP+FN}}$ (a.k.a. sensitivity, hit rate, true positive rate)
- $\text{specificity} = \frac{\text{TN}}{\text{TN+FP}}$ (a.k.a. selectivity, negative class recall true negative rate)
- $\text{F}_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}}$ (a.k.a. sensitivity, hit rate, true positive rate)

Linear Classifiers

Logistic Regression

Logistic regression is a widely used algorithm in the realm of supervised learning. It deals with numerical feature vectors and aims to predict the log odds of an event occurring for each data point, representing the (natural) logarithm of the odds, where "odds" is the ratio of the probability of success to the probability of failure. In logistic regression, the dependent variable (the outcome we are trying to predict) is binary, meaning it has only two possible outcomes (e.g., 0 or 1, true or false, success or failure, **fraud or not fraud**).

Loss Function

The negative log likelihood is the preferred loss function. The likelihood is the probability that a given set of true labels $\hat{y}_i$ would be observed, given the predicted probabilities $\hat{p}_i$.

For each data point, the logistic regression model predicts a probability $\hat{p}_i$ that the data point is *fraud or not fraud*.

The overall objective of logistic regression is to determine parameters that generate probabilities $\hat{p}_i$ maximizing the likelihood. To simplify computations, especially since many optimization methods involve computing derivatives of the loss function, the negative logarithm of the likelihood is utilized. Since maximizing the likelihood is analogous to minimizing the negative log likelihood, the negative log likelihood will be the chosen loss function.

Log Odds

The log odds can take any real value, ranging from negative to positive infinity. When p = 0.5, the odds are 1, and the log of the odds is 0. As p approaches 0, the log odds go to negative infinity and, as p approaches 1, the log odds go to positive infinity. Logistic regression assumes that the log odds of the binary outcome are a linear combination of the input features.

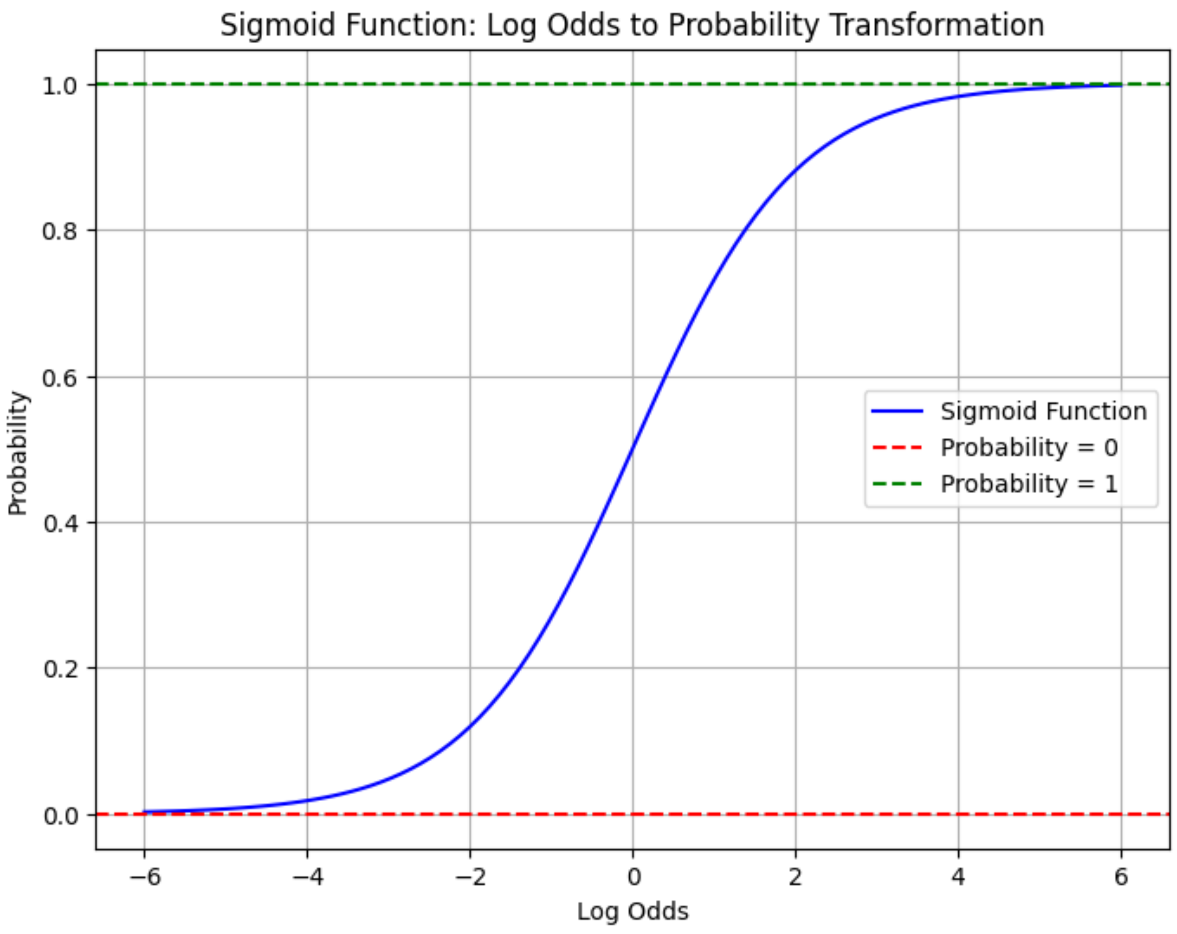
Sigmoid Function

The sigmoid function, also known as the logistic function, is used in logistic regression to model the relationship between log odds and probabilities. The key reason for this is the desire for a function that transforms linear combinations of input features into probabilities while maintaining a convenient and interpretable scale.

Notice: Why prefer a probability over a straightforward [1; 0] output? The rationale lies in constructing a probabilistic framework for predicting an output, which represents the probability of the input belonging to class 1 or 0. The motivation behind this choice is to capture and model uncertainty. Classifiers are inherently imperfect, and input data are frequently imperfect as well. Consequently, we cannot be entirely certain about the output being definitively 0 or 1. However, by providing a probability estimate, we acknowledge and quantify the inherent uncertainty in the prediction process.

The sigmoid function is a mathematical function that maps any real number x to a value between 0 and 1. It is defined as: $\sigma(x) = \frac{1}{1+e^{-x}}$. This function plays a pivotal role in logistic regression because it transforms the linear combination of input features (log odds) into probabilities. The following code generates a plot where the x-axis represents a range of log odds values, and the y-axis represents the corresponding probabilities after applying the sigmoid function. The sigmoid function ensures that the output probabilities are bounded between 0 and 1.

In [183]:
```
# Define the sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Generate a range of values for the log odds
log_odds_values = np.linspace(-6, 6, 100)  # Arbitrary values

# Calculate the corresponding probabilities using the sigmoid funct
probabilities = sigmoid(log_odds_values)

# Plotting
plt.figure(figsize=(8, 6))
plt.plot(log_odds_values, probabilities, label='Sigmoid Function',
plt.title('Sigmoid Function: Log Odds to Probability Transformation
plt.xlabel('Log Odds')
plt.ylabel('Probability')
plt.axhline(0, color='red', linestyle='--', label='Probability = 0'
plt.axhline(1, color='green', linestyle='--', label='Probability =
plt.legend()
plt.grid(True)
plt.show()
```



But...Why Not Linear Regression? Linear regression is a robust method for predicting future outcomes based on historical data. It works by creating a linear model that best fits the relationship between input variables and a real-valued response. However, when it comes to classification problems where we want to predict a category, linear regression faces challenges. The problem lies in its nature of predicting real-valued variables, while classification involves categorical outcomes. If we attempt to map categories to 0 and 1 and apply linear regression, we get a line that maps inputs to outputs, but the resulting values can fall below 0 or go beyond 1. This poses an issue because these values can't be interpreted as probabilities. Even if we treat them as scores and set a threshold for classification, this method doesn't create an effective classifier. The reason is that the squared-error loss function used in linear regression doesn't accurately represent how far points are from the classification boundary.

Train-test split

The main goal is to develop a model that is able to predict the future based on past data, so there is the need to reserve some of the labelled data to act as these "future" data in order to make possible the evaluation of the model.

In [184]:
```
X_train, X_test, y_train, y_test = train_test_split(df.drop('Class'
```

Scaling

In [185]:
```
#extract the 'Amount' column for training and testing
X_train_amount = X_train[['Amount']]
X_test_amount = X_test[['Amount']]

#initialize StandardScaler
standard_scaler = StandardScaler()

#reshape the 'Amount' column to a 2D array for training
X_train_amount_stand = standard_scaler.fit_transform(X_train_amount

#reshape the 'Amount' column to a 2D array for testing
X_test_amount_stand = standard_scaler.transform(X_test_amount.value

#replace the original 'Amount' column with the scaled values
X_train['Amount'] = X_train_amount_stand
X_test['Amount'] = X_test_amount_stand
```

The reason for using **.fit_transform** on the training set and only **.transform** on the test set when performing feature scaling, such as standardization, is to ensure that the scaling is consistent and doesn't introduce data leakage or information from the test set into the training process. Here's why this approach is followed:

Training Data: When .fit_transform is used on the training data, it calculates the mean and standard deviation of the features within the training set. These statistics are used to perform the actual scaling. It ensures that the scaling is based solely on the information present in the training data, which is what the model has learned from.

Testing Data: For the test data, the same scaling that was learned from the training data needs to be applied. It is incorrect to recalculate the mean and standard deviation based on the test data because that would introduce knowledge about the test set into the scaling process, potentially causing data leakage.

The purpose of separating the operations into .fit_transform for training data and .transform for testing data is to maintain the integrity of the training and testing process. It prevents the test data from influencing the scaling parameters and ensures that the scaling is applied consistently as if the test data were completely unseen during the training process.

Notice: Standardization has specifically been implemented on the 'Amount' column since the 'V' columns were already standardized.

In [186]:
```
X_train.describe()
```

Out[186]:

	V1	V2	V3	V4	V5	
count	414026.000000	414026.000000	414026.000000	414026.000000	414026.000000	414026
mean	0.001696	-0.021765	0.014893	-0.002996	0.010686	-0
std	0.979424	0.973642	0.995042	0.999760	0.995430	0
min	-3.010744	-42.459582	-3.183760	-4.872165	-9.952786	-18
25%	-0.550492	-0.488635	-0.631382	-0.660437	-0.280808	-0
50%	-0.095017	-0.135500	0.006465	-0.072434	0.084202	0
75%	0.795269	0.328853	0.636732	0.709600	0.441565	0
max	2.225587	4.361865	14.125834	3.201536	41.540257	26

8 rows × 27 columns

In [187]:
```
X_test.describe()
```

Out[187]:

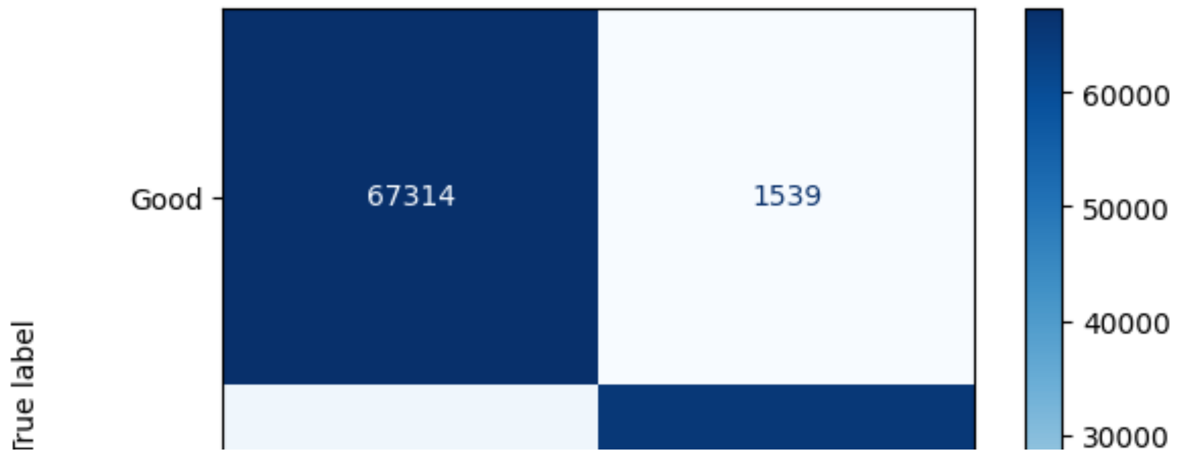
	V1	V2	V3	V4	V5	
count	138009.000000	138009.000000	138009.000000	138009.000000	138009.000000	138009
mean	0.002750	-0.023625	0.014844	-0.004603	0.010338	-0
std	0.977400	0.972645	0.992369	1.000537	0.982518	0
min	-3.495584	-49.966572	-2.517061	-4.951222	-4.298320	-21
25%	-0.550957	-0.489946	-0.631702	-0.662617	-0.279599	-0
50%	-0.093226	-0.136468	0.007712	-0.072791	0.082372	0
75%	0.795599	0.328524	0.635782	0.704218	0.440242	0
max	2.229046	4.356342	4.355693	3.139657	42.716891	9

8 rows × 27 columns

Finally, logistic regression is performed:

```
In [188]: #Logistic Regression

          start_time = time.time()

          clf =  LogisticRegression(random_state=random_seed).fit(X_train, y_

          y_pred = clf.predict(X_test)

          end_time = time.time()
          duration = end_time - start_time

          print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
          print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
          print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
          print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
          print(f"F1:          {f1_score(y_test, y_pred):.5f}")
          print(f"Model Runtime (seconds): {duration:.2f} seconds")

          ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
          plt.show()
```

```
Accuracy:    0.96042
Precision:   0.97695
Recall:      0.94326
Specificity: 0.94492
F1:          0.95981
Model Runtime (seconds): 1.05 seconds
```



Gaussian Naive Bayes

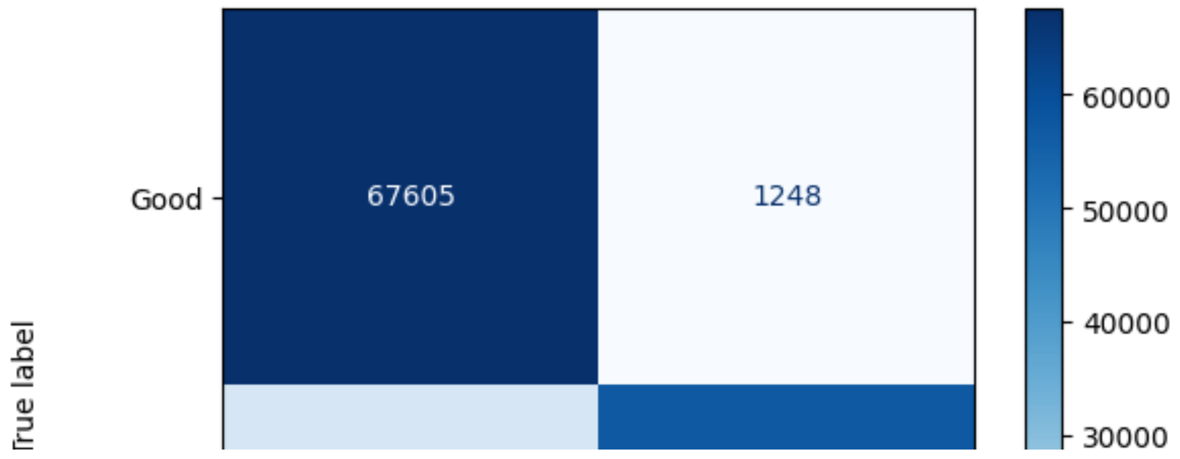
Gaussian Naive Bayes is a probabilistic classification algorithm that belongs to the Naive Bayes family. The "naive" in its name stems from the assumption it makes: it assumes that the features used for classification are conditionally independent, given the class label. In simpler terms, it treats each feature as if it contributes independently to the probability of a certain class, which is a simplification that may not always hold true in real-world scenarios.

This algorithm specifically models the distribution of each feature within each class as a Gaussian (normal) distribution. Despite its seemingly oversimplified assumption, Gaussian Naive Bayes often performs remarkably well, especially in situations where the independence assumption is not severely violated.

It's worth noting that Naive Bayes has a tendency to exhibit high bias due to its simplistic assumption. Bias refers to the model's inclination to make assumptions about the underlying patterns in the data. However, despite its high bias, Naive Bayes often works well in practice, and this is attributed to its low variance. The simplicity of the model, coupled with the independence assumption, contributes to its ability to generalize well to new, unseen data, making it a practical and efficient choice for various classification tasks.

In [189]:
```
#GaussianNB

start_time = time.time()

clf =  GaussianNB().fit(X_train, y_train)

y_pred = clf.predict(X_test)

end_time = time.time()
duration = end_time - start_time

print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
print(f"F1:          {f1_score(y_test, y_pred):.5f}")
print(f"Model Runtime (seconds): {duration:.2f} seconds")

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
plt.show()
```

```
Accuracy:    0.90417
Precision:   0.97864
Recall:      0.82681
Specificity: 0.84950
F1:          0.89634
Model Runtime (seconds): 0.18 seconds
```



Linear Support Vector Machines (SVM)

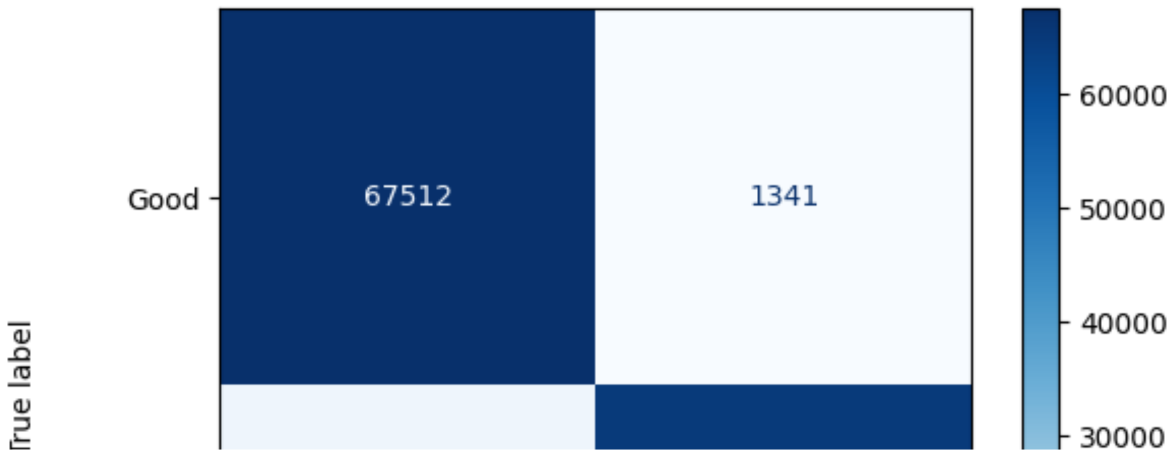
Similar to logistic regression, a support vector machine (SVM) is a basic linear classifier. This means it creates a straight line (hyperplane) in a space to separate two classes in a dataset. The key difference lies in how they learn from data.

Logistic regression uses a log-likelihood function that punishes all points based on how much they deviate from the correct probability estimate, even if they are on the correct side of the hyperplane. On the other hand, SVMs use a hinge loss. This loss function penalizes only those points that are on the wrong side of the hyperplane or very close to it on the correct side. In simpler terms, SVMs focus more on points that are challenging to classify, aiming to maximize the margin between different classes.

The SVM classifier aims to discover a hyperplane with the largest possible gap between two classes. This "gap" is known as the margin, representing the distance from the dividing line to the nearest data points on either side. Even if the data isn't perfectly separated by a straight line, SVM penalizes points inside the margin based on how far they are from it. In simpler terms, SVM works to create a wide space between different groups, considering points near the edge (support vectors) as more crucial for accurate classification.

```
In [190]: #LinearSVM

          start_time = time.time()

          clf =  LinearSVC(dual=False, random_state=random_seed).fit(X_train,

          y_pred = clf.predict(X_test)

          end_time = time.time()
          duration = end_time - start_time

          print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
          print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
          print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
          print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
          print(f"F1:          {f1_score(y_test, y_pred):.5f}")
          print(f"Model Runtime (seconds): {duration:.2f} seconds")

          ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
          plt.show()
```

```
Accuracy:    0.95933
Precision:   0.97975
Recall:      0.93823
Specificity: 0.94049
F1:          0.95854
Model Runtime (seconds): 1.35 seconds
```



Non-Linear Classifiers

In machine learning, non-linear classifiers are a category of models designed to capture complex relationships and patterns in data that cannot be effectively represented by linear decision boundaries. Unlike linear classifiers, which assume that the decision boundary is a straight line, non-linear classifiers have the flexibility to model intricate, curved, or non-linear decision boundaries. Non-linear classifiers employ various mathematical functions, known as kernels, to transform the input data into a higher-dimensional space, where non-linear patterns become more apparent.

Neural Networks

Neural networks, inspired by the intricate connectivity and computational prowess of the human brain, represent a paradigm-shifting approach in the realm of machine learning. Comprising interconnected layers of artificial neurons, these networks excel at capturing intricate patterns and relationships within data, allowing them to tackle a myriad of complex tasks across various domains. In a neural network, information flows through input layers, hidden layers, and finally culminates in an output layer, with each connection characterized by adjustable weights and biases. The process of training involves iteratively refining these weights to minimize the discrepancy between predicted and actual outcomes. In a neural network, the final layer typically contains one or more neurons, and each neuron produces an output. The number of neurons in the final layer depends on the specific problem. In this case of binary classification, there is usually a single neuron in the final layer that produces a probability score between 0 and 1, indicating the likelihood of belonging to one of the two classes. The key part of a neural network is indeed how each neuron determines its output from its inputs, and this is achieved through the use of activation functions. Activation functions introduce non-linearity into the network and help the network model complex relationships within the data. The following code utilizes the 'MLPClassifier' from scikit-learn to create a neural network classifier. By default, the MLPClassifier uses the "ReLU" (Rectified Linear Unit) activation function for hidden layers and the "Logistic" activation function for te output layer, which is equivalent to the sigmoid activation function. It produces a single value between 0 and 1, representing the probability of belonging to the positive class (this only holds for binary classification problems).

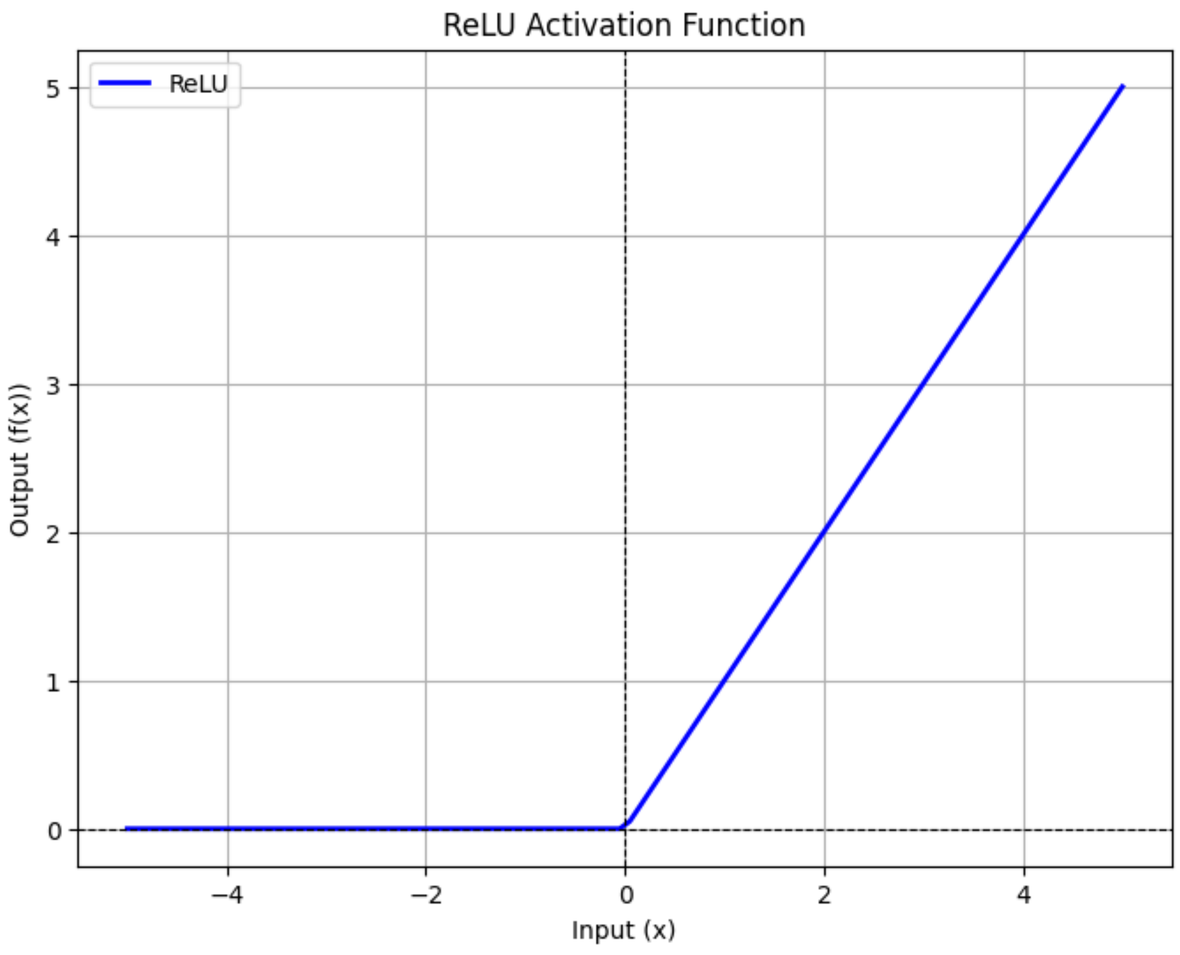
The **Rectified Linear Unit (ReLU)** activation function is one of the most widely used activation functions in artificial neural networks, particularly in deep learning models. It is a simple but effective non-linear activation function that introduces non-linearity into the model. ReLU is defined as follows:

f(x) = max(0, x)

"x" represents the input to the neuron (the weighted sum of the inputs from the previous layer). "f(x)" is the output of the neuron. If the input "x" is positive (greater than or equal to zero), the function returns "x" itself. If the input "x" is negative, the function returns zero.

Visually, this can be represented as a graph where the function is a straight line with a slope of 1 for positive values and flat at zero for negative values.

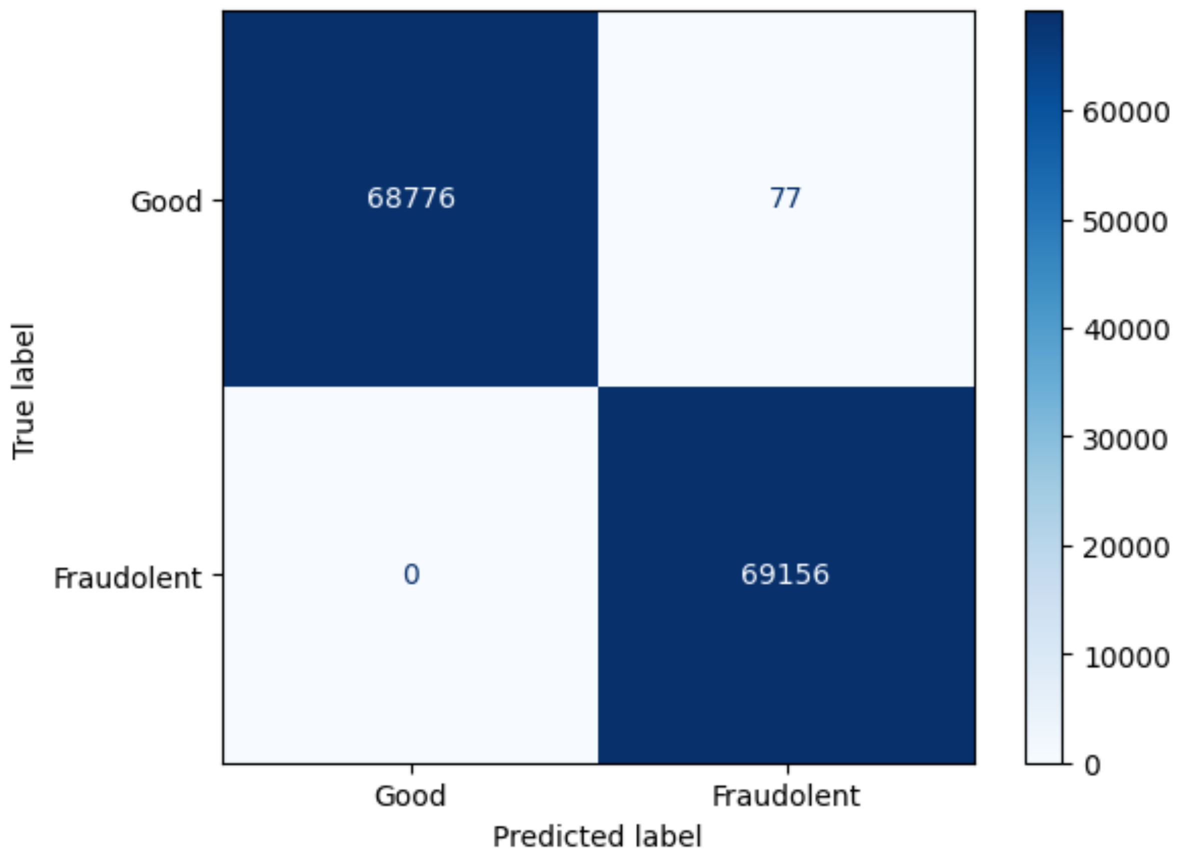
In [191]:
```
def relu(x):
    return np.maximum(0, x)

x = np.linspace(-5, 5, 100)

y = relu(x)

plt.figure(figsize=(8, 6))
plt.plot(x, y, label='ReLU', color='b', linewidth=2)
plt.axhline(0, color='k', linestyle='--', linewidth=0.8)
plt.axvline(0, color='k', linestyle='--', linewidth=0.8)
plt.xlabel('Input (x)')
plt.ylabel('Output (f(x))')
plt.title('ReLU Activation Function')
plt.legend()
plt.grid(True)
plt.show()
```



In [192]:
```
#Neural Networks

start_time = time.time()
```

```
clf = MLPClassifier(max_iter=300, random_state=random_seed)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

end_time = time.time()
duration = end_time - start_time

print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
print(f"F1:          {f1_score(y_test, y_pred):.5f}")
print(f"Model Runtime (seconds): {duration:.2f} seconds")

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
plt.show()
```

```
Accuracy:    0.99944
Precision:   0.99889
Recall:      1.00000
Specificity: 1.00000
F1:          0.99944
Model Runtime (seconds): 46.83 seconds
```



In [193]:
```
os.environ['LOKY_MAX_CPU_COUNT'] = '4'
```

K-Nearest-Neighbours

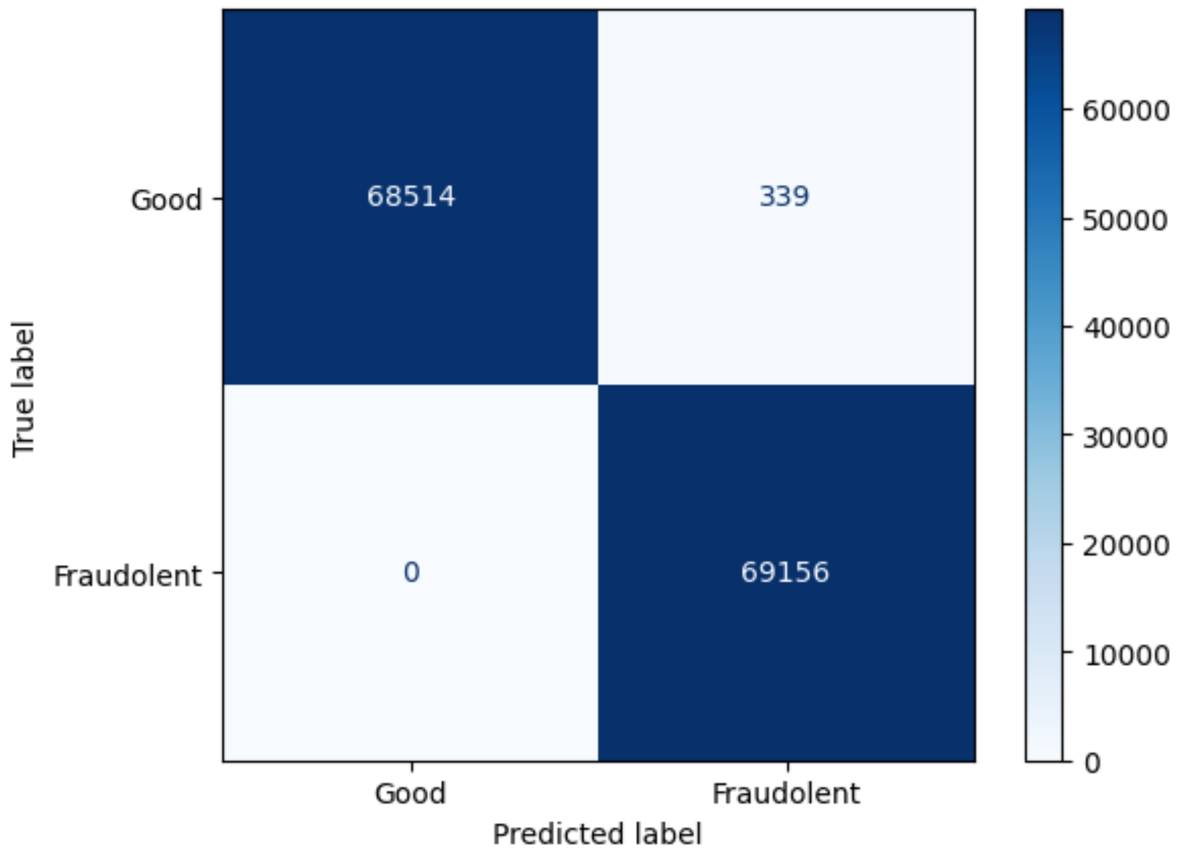
The k-nearest neighbours (k-NN) algorithm is an example of a lazy learning approach in the field of machine learning. Unlike traditional eager learning algorithms, which perform extensive computations during the training phase, lazy learning methods, such as k-NN, defer most of the work to the classification phase. During training, k-NN merely stores all the feature vectors and their corresponding labels in the model. The crux of k-NN's functionality becomes apparent during classification, where it leverages this repository of training data to make local generalizations for each test sample based on its k-nearest neighbours. This simplicity defines k-NN, making it one of the elementary machine learning algorithms.

The determination of proximity between data points in an n-dimensional feature space, where n corresponds to the feature vector's dimensionality, relies on distance metrics. Typically, Euclidean distance serves continuous variables, while Hamming distance is suitable for discrete variables. The uncomplicated nature of k-NN translates into a swift training phase, setting it apart from many other learning algorithms. However, this expedited training comes at the expense of classification processing time and space efficiency. The model's space requirement is substantial, as it needs to retain all feature vectors and labels.

In [194]:
```
#kNN

start_time = time.time()

clf = KNeighborsClassifier()
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

end_time = time.time()
duration = end_time - start_time

print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
print(f"F1:          {f1_score(y_test, y_pred):.5f}")
print(f"Model Runtime (seconds): {duration:.2f} seconds")

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
plt.show()
```

```
Accuracy:    0.99754
Precision:   0.99512
Recall:      1.00000
Specificity: 1.00000
F1:          0.99756
```

```
Model Runtime (seconds): 33.80 seconds
```



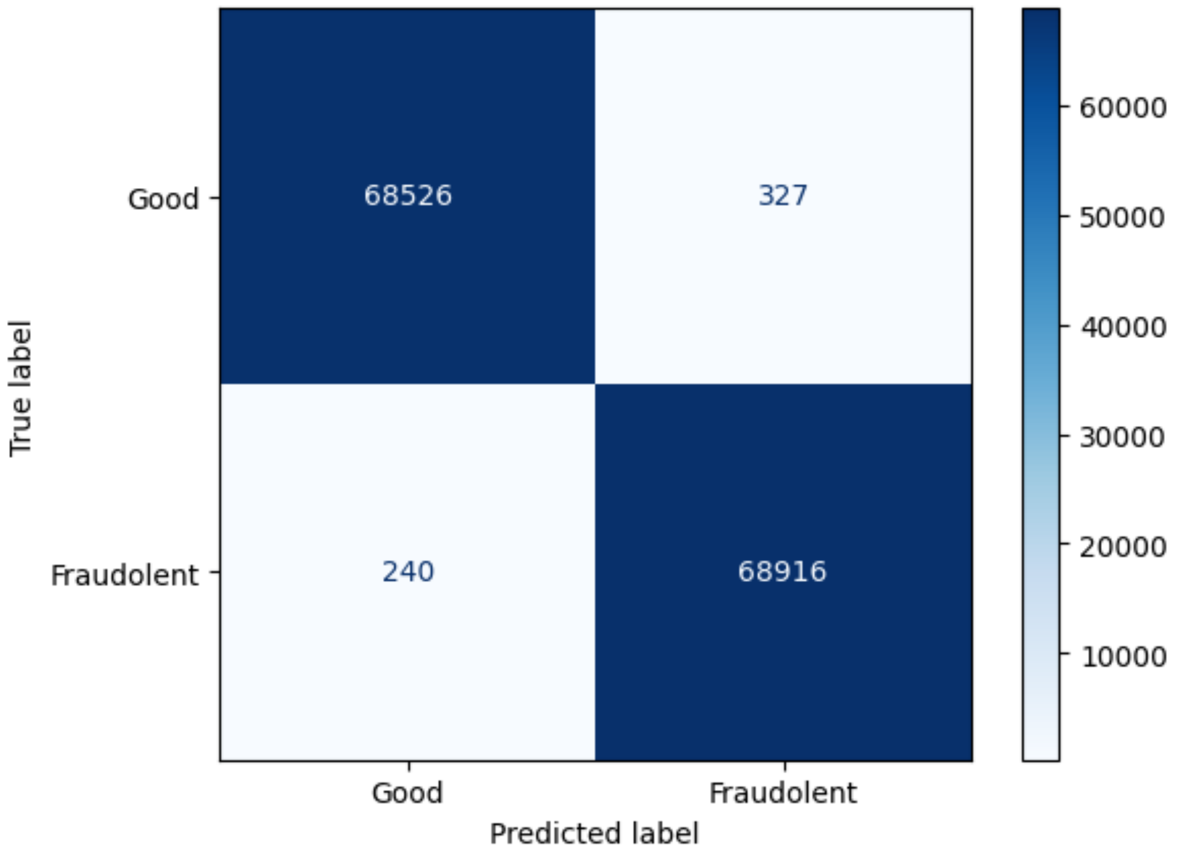
Kernelized SVM

Kernelized Support Vector Machines (SVMs) unlock the true potential of this powerful machine learning technique by embracing the so-called 'kernel trick'. At its core, a kernel takes a seemingly simple linear decision boundary and works to produce a far more complex and nonlinear boundary, applying a transformation from one vector space, V1, to another, V2, where each point in V1 is mapped to a function in V2. By introducing nonlinearity to an otherwise linear classifier, they make SVMs capable of tackling intricate classification challenges. Among the kernel choices, the radial basis function (RBF) kernel stands out as a favorite, often applied to capture intricate relationships in the data. With the RBF kernel, it's as if each data point has a sphere of influence, determining the region where points are classified in the same way.

In [195]:
```
#kSVM

start_time = time.time()

clf = SVC(kernel='rbf', random_state=random_seed)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

end_time = time.time()
```

```
duration = end_time - start_time

print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
print(f"F1:          {f1_score(y_test, y_pred):.5f}")
print(f"Model Runtime (seconds): {duration:.2f} seconds")

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
plt.show()
```

```
Accuracy:    0.99589
Precision:   0.99528
Recall:      0.99653
Specificity: 0.99651
F1:          0.99590
Model Runtime (seconds): 349.73 seconds
```



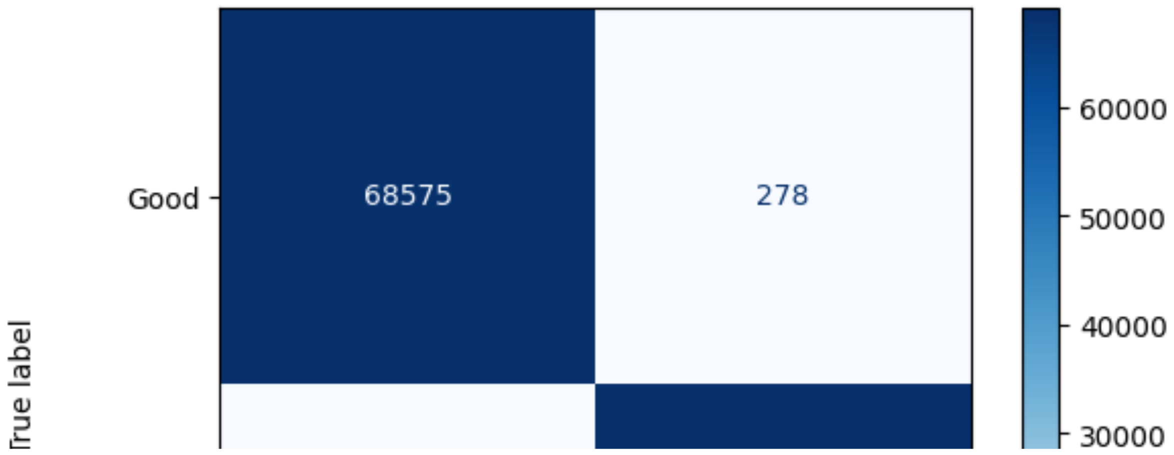
Decision Trees

Decision trees are versatile supervised learning models known for their interpretability and intuitive decision-making processes. They are binary tree structures used for classification and regression tasks, capable of handling both categorical and numerical data without requiring extensive preprocessing. The construction of a typical decision tree begins at the root, where the dataset is split into child subsets based on binary conditions. These conditions are automatically selected to maximize the quality of the split, with metrics like Gini impurity, variance reduction, and information gain being common criteria. Decision trees can grow until certain stopping conditions are met, such as achieving pure leaves, reaching a maximum depth, or having child nodes with an insufficient number of samples. The resulting tree represents a series of binary decisions leading to classifications at the leaf nodes.

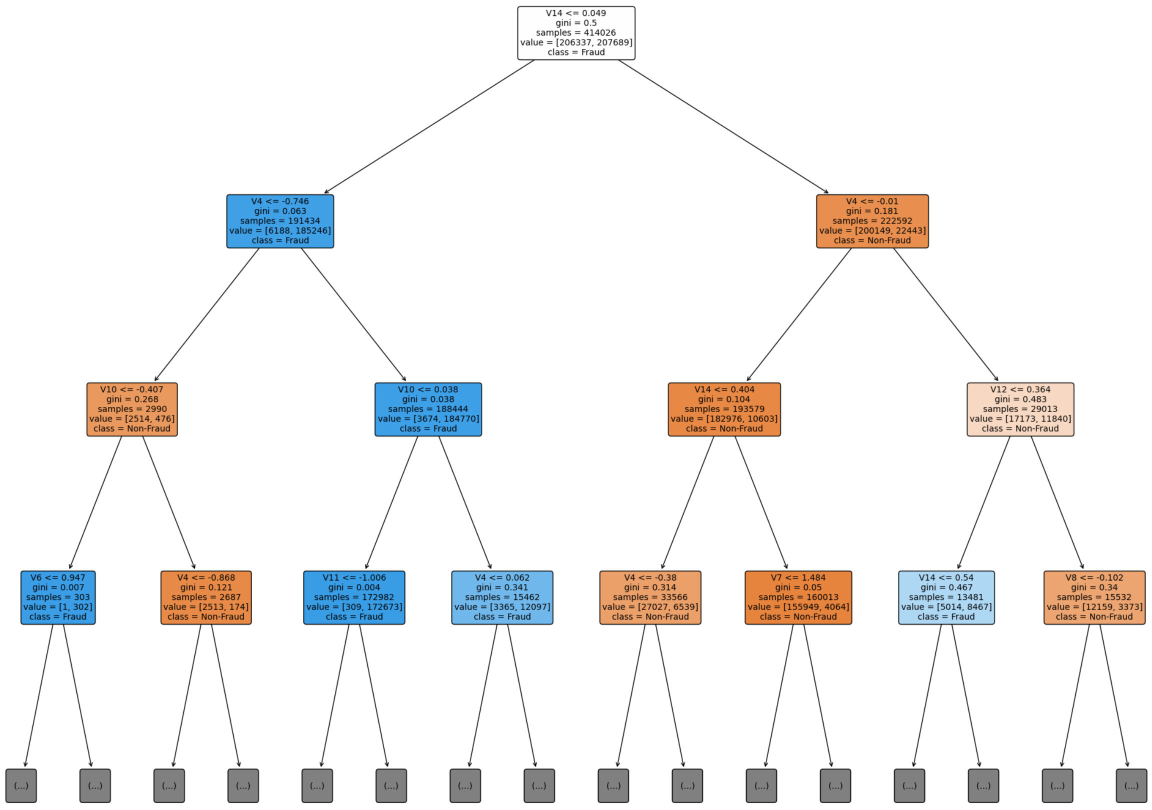
Decision trees excel in interpretability, as predictions can be explained through a series of Boolean conditions. They are efficient to train and make predictions on, especially for large datasets. However, they are prone to overfitting, requiring pruning to reduce complexity. Additionally, they may struggle with certain types of relationships and can be sensitive to minor variations in the training data. Despite these limitations, decision trees remain a valuable tool in the realm of machine learning.

In [196]:
```
#Decision Tree

start_time = time.time()

clf = DecisionTreeClassifier(random_state=random_seed)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

end_time = time.time()
duration = end_time - start_time

print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
print(f"F1:          {f1_score(y_test, y_pred):.5f}")
print(f"Model Runtime (seconds): {duration:.2f} seconds")

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
plt.show()
```

```
Accuracy:    0.99720
Precision:   0.99599
Recall:      0.99842
Specificity: 0.99841
F1:          0.99721
Model Runtime (seconds): 22.65 seconds
```



In [200]:
```python
# Visualize the decision tree
plt.figure(figsize=(25, 20))
plot_tree(clf, filled=True, feature_names=df.columns, class_names=[
plt.show()
```



The plotted decision tree is a visual representation of how the decision tree classifier clf makes predictions. The tree starts at the root and splits down to the third level of depth based on feature thresholds, leading to a decision of whether a sample is classified as 'Non-Fraud' or 'Fraud'. Each node in the tree shows:

The feature and threshold that it's splitting on (e.g., V4 <= 0.049).

The Gini impurity (gini), which is a measure of how often a randomly chosen element would be incorrectly identified.

The number of samples (samples) that fall into that node.

The distribution of the samples across the different classes (value).

The dominant class (class) at that node based on the distribution of the samples.

Ensemble Methods

Ensemble methods in machine learning are techniques that combine the predictions of multiple individual models (often called base or weak learners) to create a more accurate and robust model. The fundamental idea behind ensemble methods is to leverage the diversity and complementary strengths of multiple models to improve overall predictive performance.

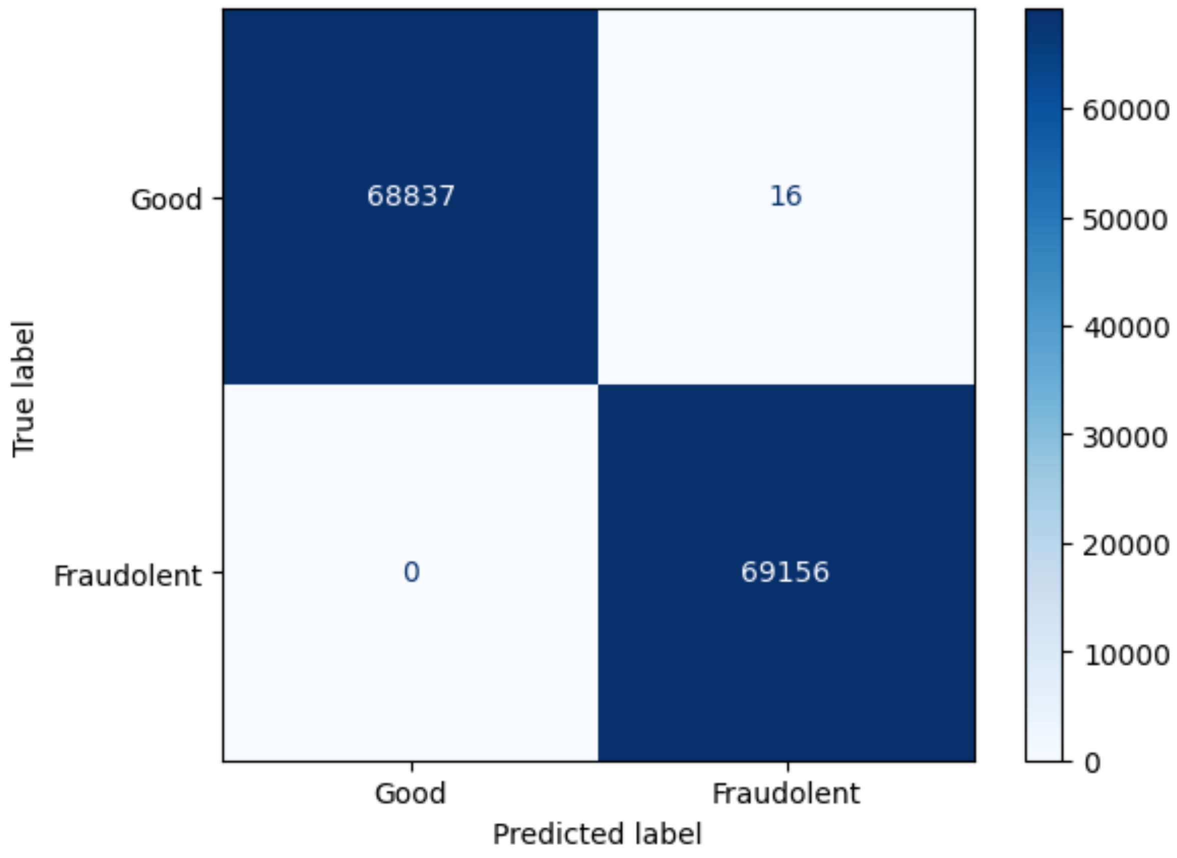
Ensemble Learning (1): Random Forest

Random Forests emerge as a prominent choice within the realm of ensembles. They are formed through the ensemble of numerous decision trees, often numbering in the tens to thousands. These individual trees independently learn from the data and collectively contribute to making predictions. In the case of classification tasks like the one of this project, each tree "votes" to determine the final prediction, while regression tasks rely on averaging the individual tree predictions. One of the key challenges addressed by the Random Forest algorithm is the risk of highly similar trees and repetitive splits, especially for influential features. This issue is tackled by introducing randomness during training, including random sampling of data points and feature selection, resulting in a diverse ensemble that significantly enhances model performance. Random Forests offer advantages such as reduced overfitting compared to single decision trees and efficient parallelized training. However, their enhanced complexity can pose challenges in terms of storage requirements and interpretability of predictions.

In [201]:
```
#Random Forest

start_time = time.time()

clf = RandomForestClassifier(random_state=random_seed)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

end_time = time.time()
duration = end_time - start_time

print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
print(f"F1:          {f1_score(y_test, y_pred):.5f}")
print(f"Model Runtime (seconds): {duration:.2f} seconds")

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
plt.show()
```

```
Accuracy:    0.99988
Precision:   0.99977
Recall:      1.00000
Specificity: 1.00000
```

```
F1:          0.99988
Model Runtime (seconds): 217.88 seconds
```



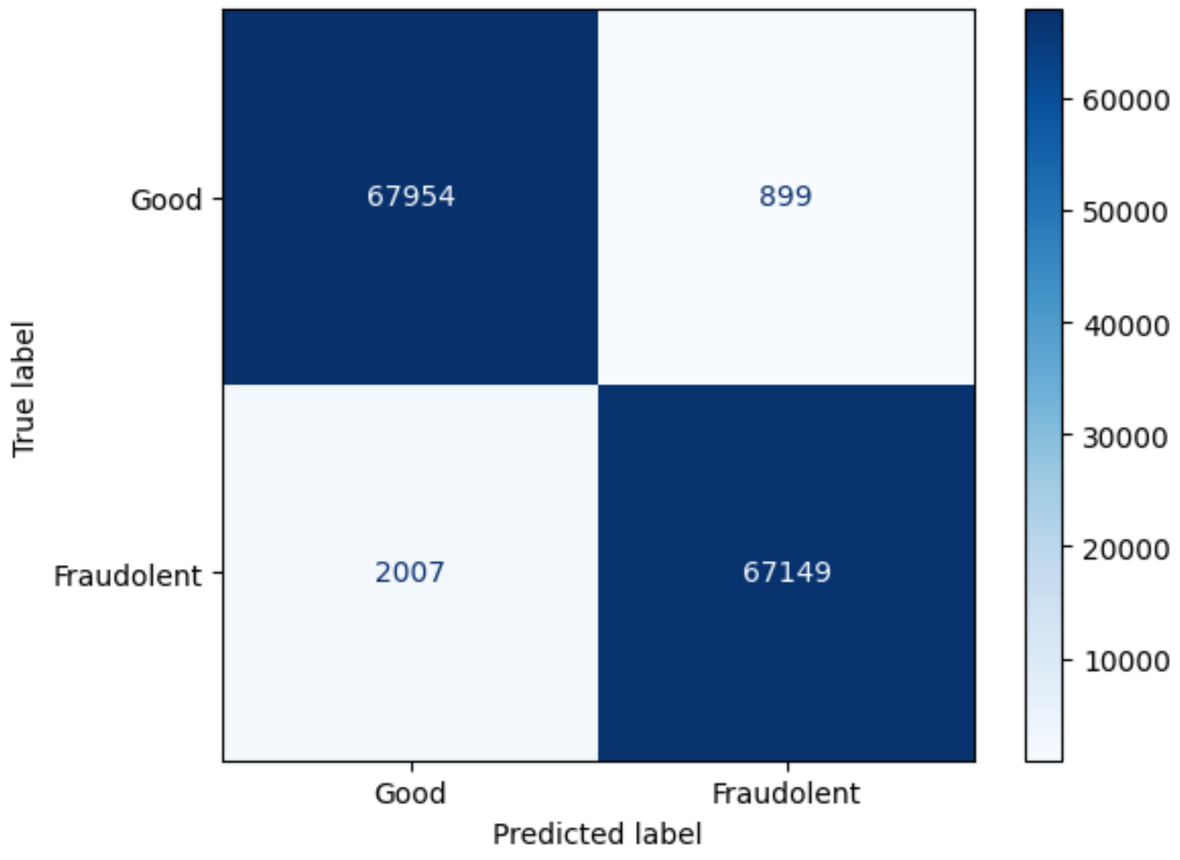
Ensemble Learning (2): Gradient Boosted Decision Trees

Gradient-boosted decision trees (GBDTs) stand out as a significant advancement in ensemble learning, offering a clever way to combine the predictions of individual decision trees for improved overall forecasting. In the world of machine learning, GBDTs have gained attention due to their reliance on gradient boosting, a technique that strategically integrates multiple modest learners through a gradient descent optimization process aimed at minimizing the loss function. The fundamental principle of GBDTs involves a step-by-step addition of individual trees to the ensemble. This addition process is guided by gradient descent, which helps optimize the model's performance. Importantly, the addition of trees stops when certain predefined conditions are met, ensuring that the model doesn't overcomplicate itself. To enhance GBDTs further, several innovations have been introduced, including limitations on tree complexity, the use of shrinkage to balance the influence of trees, subsampling methods, and regularization techniques. While GBDTs offer remarkable predictive capabilities, they also require careful handling to prevent overfitting and manage computational resources efficiently.

In [202]:
```
#GradientBoost Decision Trees

start_time = time.time()
```

```
clf = GradientBoostingClassifier(random_state=random_seed)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

end_time = time.time()
duration = end_time - start_time

print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
print(f"F1:          {f1_score(y_test, y_pred):.5f}")
print(f"Model Runtime (seconds): {duration:.2f} seconds")

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
plt.show()
```

```
Accuracy:    0.97894
Precision:   0.98679
Recall:      0.97098
Specificity: 0.97131
F1:          0.97882
Model Runtime (seconds): 379.92 seconds
```



Comparison Between Models

All the previously seen supervised learning models were constructed through diverse techniques and configurations, each one offering unique perspectives on the underlying data and exhibiting varying levels of predictive performance. To undertake a comprehensive comparison between them, two powerful visualization tools will be used: the Precision-Recall curve and the ROC (Receiver Operating Characteristic) curve. To facilitate the process of creating those graphs for the mentioned classifiers, a dictionary is established. This dictionary serves as a central repository for storing and organizing the results of each classifier's predictions and performance metrics.

In [35]:
```
#Dictionary

logging.basicConfig(level=logging.INFO)

clf_dict = {}

models = {
    'Logistic Regression': LogisticRegression(C=0.9, random_state=r
    'Gaussian NB': GaussianNB(),
    'Linear SVM': LinearSVC(dual=False, random_state=random_seed),
    'Neural Networks': MLPClassifier(hidden_layer_sizes=(50, 50), m
    'kNN': KNeighborsClassifier(n_neighbors=5),
    'RBF SVM': SVC(kernel='rbf', probability=True, random_state=ran
    'Decision Tree': DecisionTreeClassifier(random_state=random_see
    'Random Forest': RandomForestClassifier(random_state=random_see
    'GradientBoost': GradientBoostingClassifier(random_state=random
}

for model_name, model in models.items():
    start_time = time.time()
    model.fit(X_train, y_train)
    end_time = time.time()
    training_time = end_time - start_time
    logging.info(f"{model_name} trained in {training_time:.2f} seco
    clf_dict[model_name] = model
```

```
INFO:root:Logistic Regression trained in 1.20 seconds
INFO:root:Gaussian NB trained in 0.16 seconds
INFO:root:Linear SVM trained in 1.26 seconds
INFO:root:Neural Networks trained in 60.39 seconds
INFO:root:kNN trained in 0.07 seconds
INFO:root:RBF SVM trained in 2726.84 seconds
INFO:root:Decision Tree trained in 22.86 seconds
INFO:root:Random Forest trained in 220.07 seconds
INFO:root:GradientBoost trained in 382.23 seconds
```

Precision-Recall

The Precision-Recall Curve is a valuable tool in evaluating classification models. It quantifies a model's ability to accurately classify positive instances while considering the trade-off with false positives.

Remember that:

Precision is a measure of how accurately the model classifies positive instances out of all instances it predicts as positive. It is calculated as: Precision = TP / (TP + FP), where TP is the number of true positives (correctly predicted positive instances) and FP is the number of false positives (negative instances predicted as positive).

Recall, also known as sensitivity or true positive rate, measures the ability of the model to capture all positive instances in the dataset. It is calculated as: Recall = TP / (TP + FN), where TP is the number of true positives and FN is the number of false negatives (positive instances predicted as negative).

By plotting precision against recall at varying decision thresholds, it provides insights into model performance. The area under the curve (AUC-PR) is often used to compare models, with higher values indicating better performance. The curve aids in selecting the most suitable model for the desired problem.

Logistic Regression

In [36]:
```
clf = clf_dict['Logistic Regression']

y_proba = clf.predict_proba(X_test)
lr_pr, lr_rc, _ = precision_recall_curve(y_test, y_proba[:, 1])
lr_avg_prec = average_precision_score(y_test, y_proba[:, 1])
```

Gaussian NB

In [37]:
```
clf = clf_dict['Gaussian NB']

y_proba = clf.predict_proba(X_test)
gdb_pr, gdb_rc, _ = precision_recall_curve(y_test, y_proba[:, 1])
gdb_avg_prec = average_precision_score(y_test, y_proba[:, 1])
```

LinearSVM

In [38]:
```
# Does not have the .predict_proba method
clf = clf_dict['Linear SVM']

# Use decision function to get the decision scores
decision_scores = clf.decision_function(X_test)

# For precision-recall curve, directly use decision scores
lsvm_pr, lsvm_rc, _ = precision_recall_curve(y_test, decision_score

# For average precision score, use the decision scores as well
lsvm_avg_prec = average_precision_score(y_test, decision_scores)
```

Neural Networks

In [39]:
```
clf = clf_dict['Neural Networks']

y_proba = clf.predict_proba(X_test)
nn_pr, nn_rc, _ = precision_recall_curve(y_test, y_proba[:, 1])
nn_avg_prec = average_precision_score(y_test, y_proba[:, 1])
```

kNN

In [40]:
```
clf = clf_dict['kNN']

y_proba = clf.predict_proba(X_test)
knn_pr, knn_rc, _ = precision_recall_curve(y_test, y_proba[:, 1])
knn_avg_prec = average_precision_score(y_test, y_proba[:, 1])
```

RBF SVM

In [41]:
```
clf = clf_dict['RBF SVM']

y_proba = clf.predict_proba(X_test)
ksvm_pr, ksvm_rc, _ = precision_recall_curve(y_test, y_proba[:, 1])
ksvm_avg_prec = average_precision_score(y_test, y_proba[:, 1])
```

Decision Tree

In [42]:
```
clf = clf_dict['Decision Tree']

y_proba = clf.predict_proba(X_test)
dt_pr, dt_rc, _ = precision_recall_curve(y_test, y_proba[:, 1])
dt_avg_prec = average_precision_score(y_test, y_proba[:, 1])
```

Random Forest

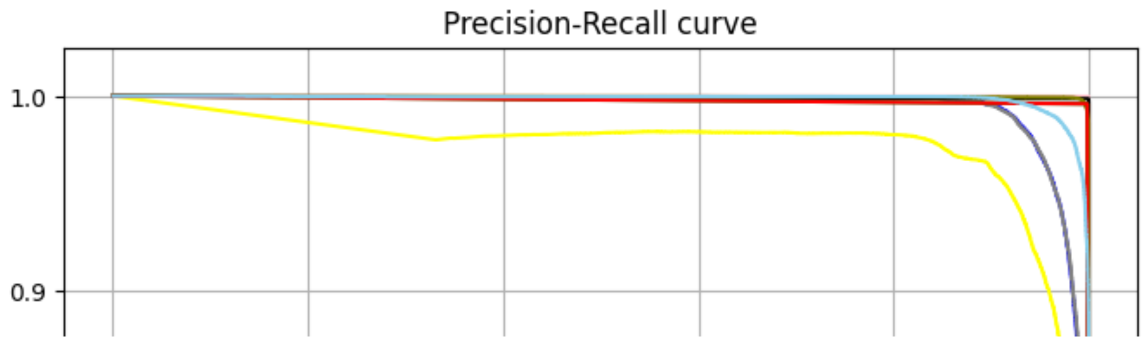
In [43]:
```
clf = clf_dict['Decision Tree']

y_proba = clf.predict_proba(X_test)
rf_pr, rf_rc, _ = precision_recall_curve(y_test, y_proba[:, 1])
rf_avg_prec = average_precision_score(y_test, y_proba[:, 1])
```

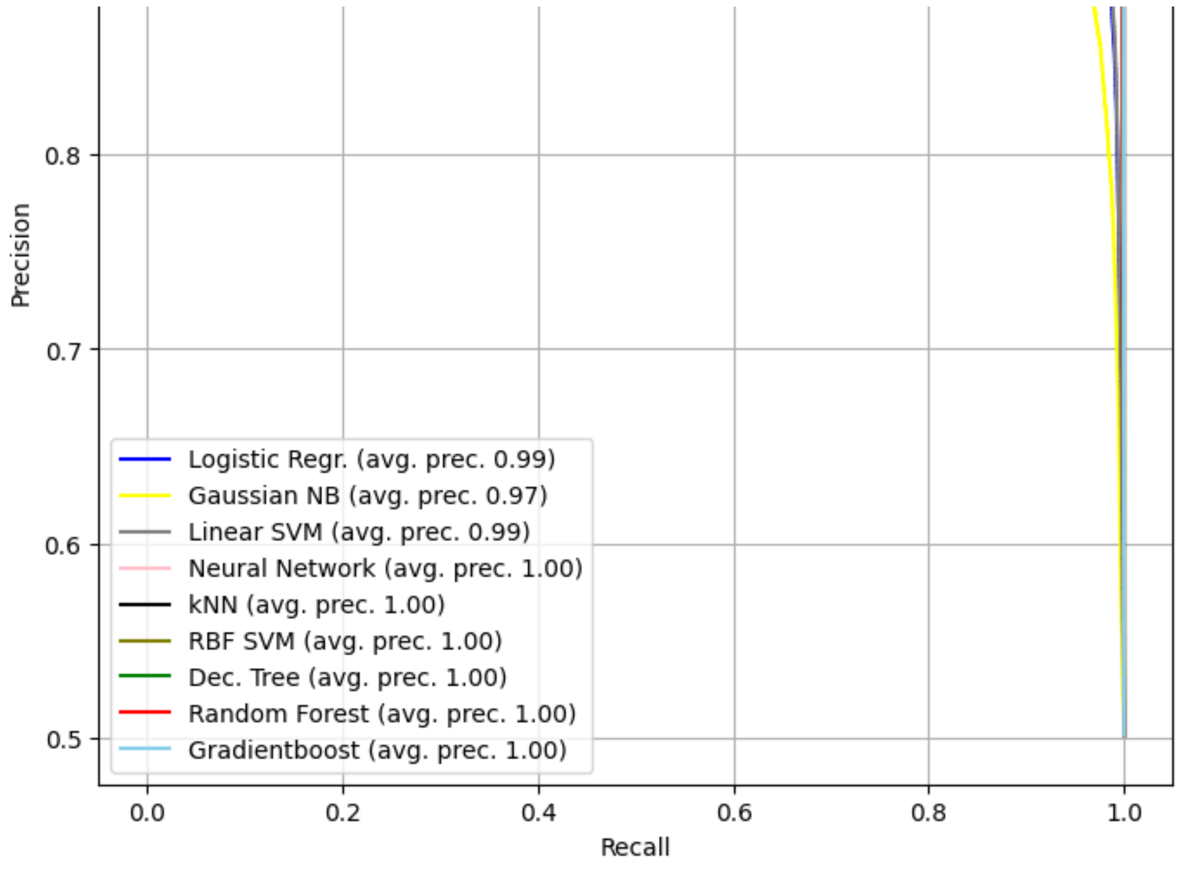
GradientBoost

In [44]:
```
clf = clf_dict['GradientBoost']

y_proba = clf.predict_proba(X_test)
gb_pr, gb_rc, _ = precision_recall_curve(y_test, y_proba[:, 1])
gb_avg_prec = average_precision_score(y_test, y_proba[:, 1])
```

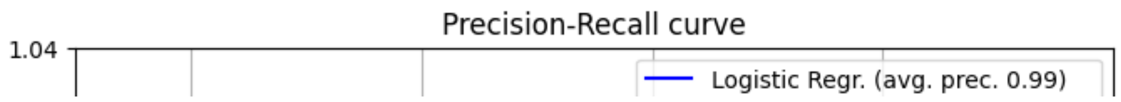
Plot the curves:

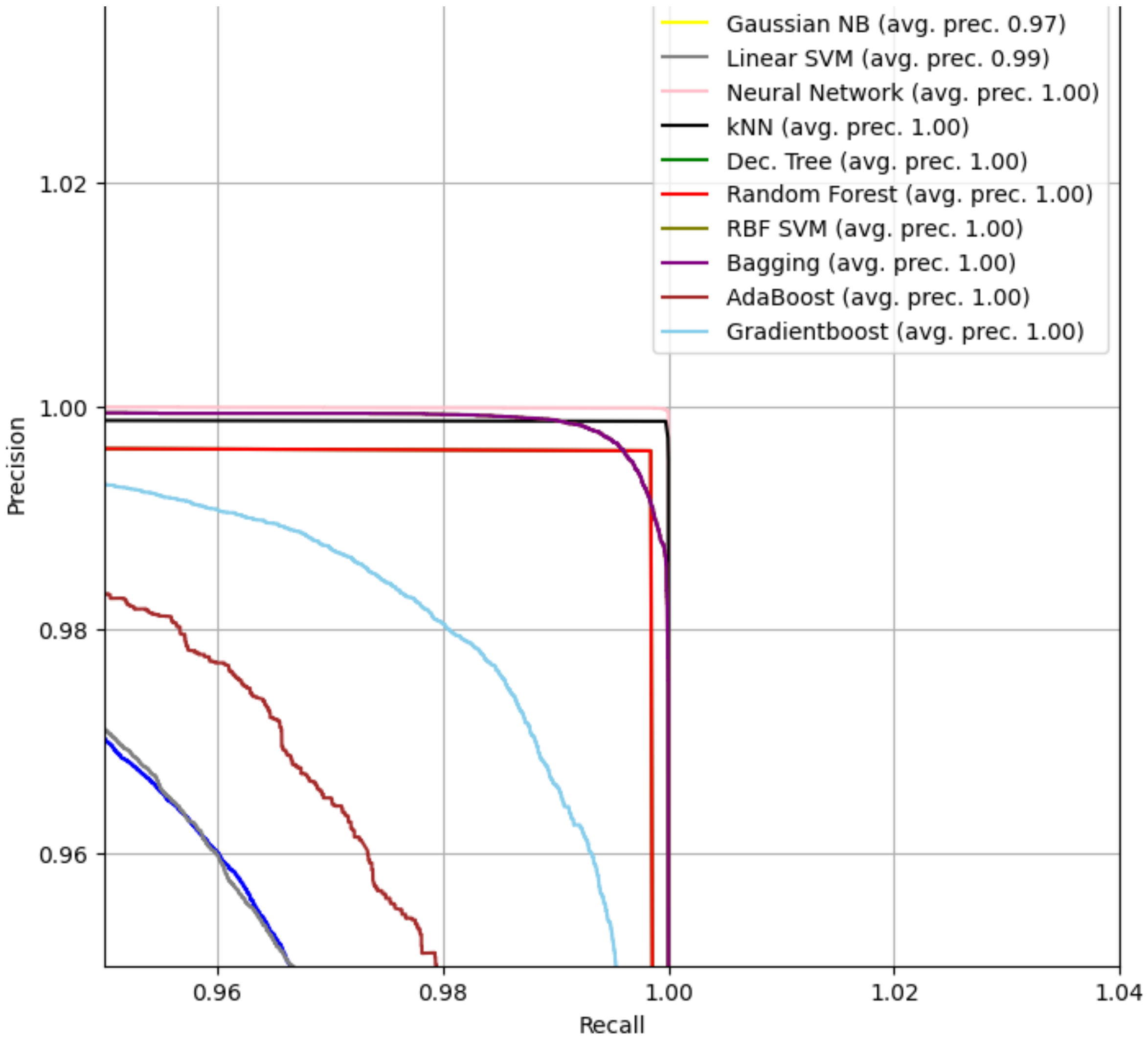
In [45]:
```
plt.figure(figsize=(8, 8))

plt.plot(lr_rc, lr_pr, c='blue', label=f'Logistic Regr. (avg. prec.
plt.plot(gdb_rc, gdb_pr, c='yellow', label=f'Gaussian NB (avg. prec
plt.plot(lsvm_rc, lsvm_pr, c='grey', label=f'Linear SVM (avg. prec.
plt.plot(nn_rc, nn_pr, c='pink', label=f'Neural Network (avg. prec.
plt.plot(knn_rc, knn_pr, c='black', label=f'kNN (avg. prec. {knn_av
plt.plot(ksvm_rc, ksvm_pr, c='olive', label=f'RBF SVM (avg. prec. {
plt.plot(dt_rc, dt_pr, c='green', label=f'Dec. Tree (avg. prec. {dt
plt.plot(rf_rc, rf_pr, c='red', label=f'Random Forest (avg. prec. {
plt.plot(gb_rc, gb_pr, c='skyblue', label=f'Gradientboost (avg. pre

plt.legend()

plt.grid()

plt.title('Precision-Recall curve')
plt.xlabel('Recall')
plt.ylabel('Precision')

plt.show()
```





In [58]:
```
plt.figure(figsize=(8, 8))

plt.plot(lr_rc, lr_pr, c='blue', label=f'Logistic Regr. (avg. prec.
plt.plot(gdb_rc, gdb_pr, c='yellow', label=f'Gaussian NB (avg. prec
plt.plot(lsvm_rc, lsvm_pr, c='grey', label=f'Linear SVM (avg. prec.
plt.plot(nn_rc, nn_pr, c='pink', label=f'Neural Network (avg. prec.
plt.plot(knn_rc, knn_pr, c='black', label=f'kNN (avg. prec. {knn_av
plt.plot(dt_rc, dt_pr, c='green', label=f'Dec. Tree (avg. prec. {dt
plt.plot(rf_rc, rf_pr, c='red', label=f'Random Forest (avg. prec. {
plt.plot(ksvm_rc, ksvm_pr, c='olive', label=f'RBF SVM (avg. prec. {
plt.plot(b_rc, b_pr, c='purple', label=f'Bagging (avg. prec. {b_avg
plt.plot(gb_rc, gb_pr, c='skyblue', label=f'Gradientboost (avg. pre

plt.xlim([0.95, 1.04])
plt.ylim([0.95, 1.04])

plt.legend()

plt.grid()

plt.title('Precision-Recall curve')
plt.xlabel('Recall')
plt.ylabel('Precision')

plt.show()
```





Although the majority of the models in the dataset demonstrate effective performance, it is evident from the precision-recall curve analysis that the neural network stands out as the best-performing model. The neural network consistently achieves the highest precision and recall, making it the top choice for this particular task, according to this evaluation.

ROC Curve

The Receiver Operating Characteristic (ROC) curve is a widely used method for comparing models in the field of machine learning. It assesses a model's performance by plotting the false positive rate ($FPR = \frac{FP}{FP+TN}$) on the x-axis and the true positive rate ($TPR = \frac{TP}{TP+FN}$), also known as Recall, on the y-axis. Each point on the curve corresponds to a specific score threshold and represents the (FPR, TPR) pair at that threshold. The area under the curve (AUC) is a crucial metric computed from the ROC curve and reflects the probability that a randomly selected positive example will receive a higher score than a randomly selected negative example. An AUC of 0.5 indicates random chance, while a higher AUC signifies better model discrimination. Notably, AUC remains robust even in situations with imbalanced sample sizes. Even though this is not the case in this project, this makes it an excellent choice for model evaluation in diverse scenarios.

Logistic Regression

In [46]:
```
clf = clf_dict['Logistic Regression']

y_proba = clf.predict_proba(X_test)
lr_fpr, lr_tpr, _ = roc_curve(y_test, y_proba[:, 1])
lr_auc= roc_auc_score(y_test, y_proba[:, 1])
```

Gaussian NB

In [47]:
```
clf = clf_dict['Gaussian NB']

y_proba = clf.predict_proba(X_test)
gdb_fpr, gdb_tpr, _ = roc_curve(y_test, y_proba[:, 1])
gdb_auc= roc_auc_score(y_test, y_proba[:, 1])
```

Linear SVM

In [48]:
```
# Assuming you have already trained the LinearSVC classifier and st
clf = clf_dict['Linear SVM']

# Use decision function to get the decision scores
decision_scores = clf.decision_function(X_test)

# For precision-recall curve, you can directly use decision scores
lsvm_fpr, lsvm_tpr, _ = roc_curve(y_test, decision_scores)

# For average precision score, you can use the decision scores as w
lsvm_auc = roc_auc_score(y_test, decision_scores)
```

Neural Networks

In [49]:
```
clf = clf_dict['Neural Networks']

y_proba = clf.predict_proba(X_test)
nn_fpr, nn_tpr, _ = roc_curve(y_test, y_proba[:, 1])
nn_auc= roc_auc_score(y_test, y_proba[:, 1])
```

kNN

In [50]:
```
clf = clf_dict['kNN']

y_proba = clf.predict_proba(X_test)
knn_fpr, knn_tpr, _ = roc_curve(y_test, y_proba[:, 1])
knn_auc= roc_auc_score(y_test, y_proba[:, 1])
```

RBF SVM

In [51]:
```
clf = clf_dict['RBF SVM']

y_proba = clf.predict_proba(X_test)
ksvm_fpr, ksvm_tpr, _ = roc_curve(y_test, y_proba[:, 1])
ksvm_auc= roc_auc_score(y_test, y_proba[:, 1])
```

Decision Tree

In [52]:
```
clf = clf_dict['Decision Tree']

y_proba = clf.predict_proba(X_test)
dt_fpr, dt_tpr, _ = roc_curve(y_test, y_proba[:, 1])
dt_auc= roc_auc_score(y_test, y_proba[:, 1])
```

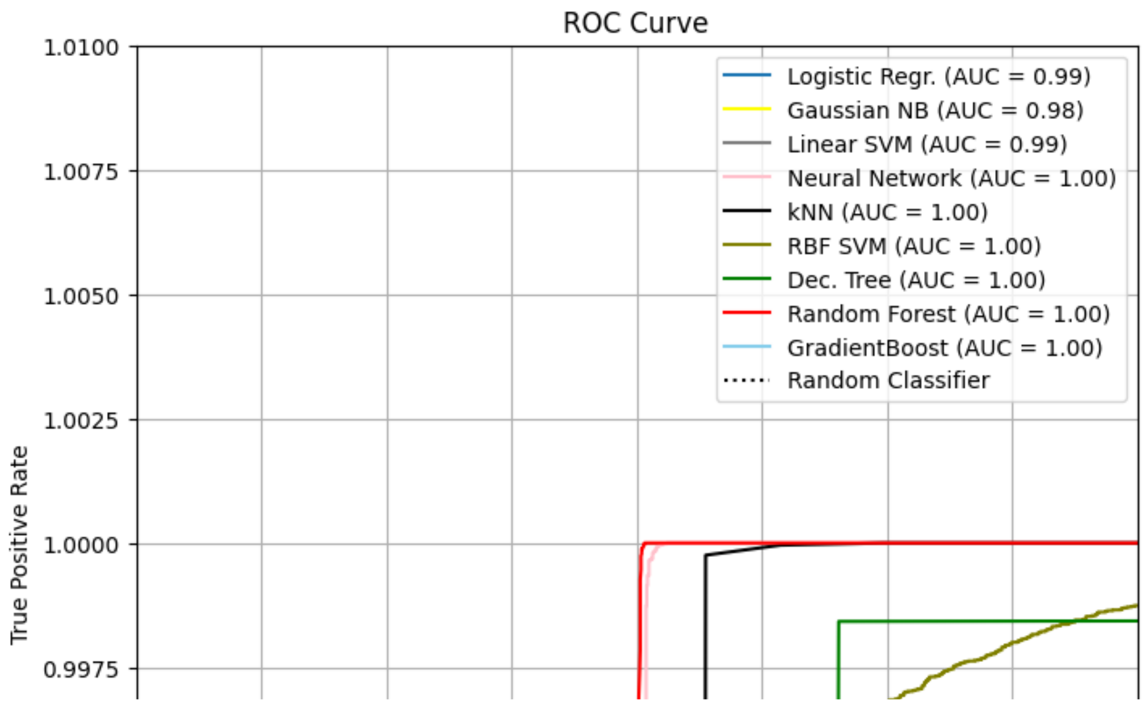
Random Forest

In [53]:
```
clf = clf_dict['Random Forest']

y_proba = clf.predict_proba(X_test)
rf_fpr, rf_tpr, _ = roc_curve(y_test, y_proba[:, 1])
rf_auc= roc_auc_score(y_test, y_proba[:, 1])
```

GradientBoost

In [55]:
```
clf = clf_dict['GradientBoost']

y_proba = clf.predict_proba(X_test)
gb_fpr, gb_tpr, _ = roc_curve(y_test, y_proba[:, 1])
gb_auc= roc_auc_score(y_test, y_proba[:, 1])
```

In [57]:
```
plt.figure(figsize=(8, 8))

plt.plot(lr_fpr, lr_tpr , c='tab:blue', label=f'Logistic Regr. (AUC
plt.plot(gdb_fpr, gdb_tpr, c='yellow', label=f'Gaussian NB (AUC = {
plt.plot(lsvm_fpr, lsvm_tpr, c='grey', label=f'Linear SVM (AUC = {l
plt.plot(nn_fpr, nn_tpr, c='pink', label=f'Neural Network (AUC = {n
plt.plot(knn_fpr, knn_tpr , c='black', label=f'kNN (AUC = {knn_auc:
plt.plot(ksvm_fpr, ksvm_tpr , c='olive', label=f'RBF SVM (AUC = {ks
plt.plot(dt_fpr, dt_tpr , c='green', label=f'Dec. Tree (AUC = {dt_a
plt.plot(rf_fpr, rf_tpr, c='red', label=f'Random Forest (AUC = {rf_
plt.plot(gb_fpr, gb_tpr, c='skyblue', label=f'GradientBoost (AUC =

plt.xlim([-0.01, 0.01])
plt.ylim([0.99, 1.01])

plt.plot([0, 1], [0, 1], 'k:', label='Random Classifier')

plt.legend()

plt.grid()

plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')

plt.show()
```



While it is notable that the majority of the models in the dataset exhibit efficient performance, the Random Forest model emerges as the top performer when evaluated based on the ROC curve. The ROC curve analysis reveals that the Random Forest consistently achieves the highest area under the curve (AUC-ROC), demonstrating its strong ability to distinguish between positive and negative classes.

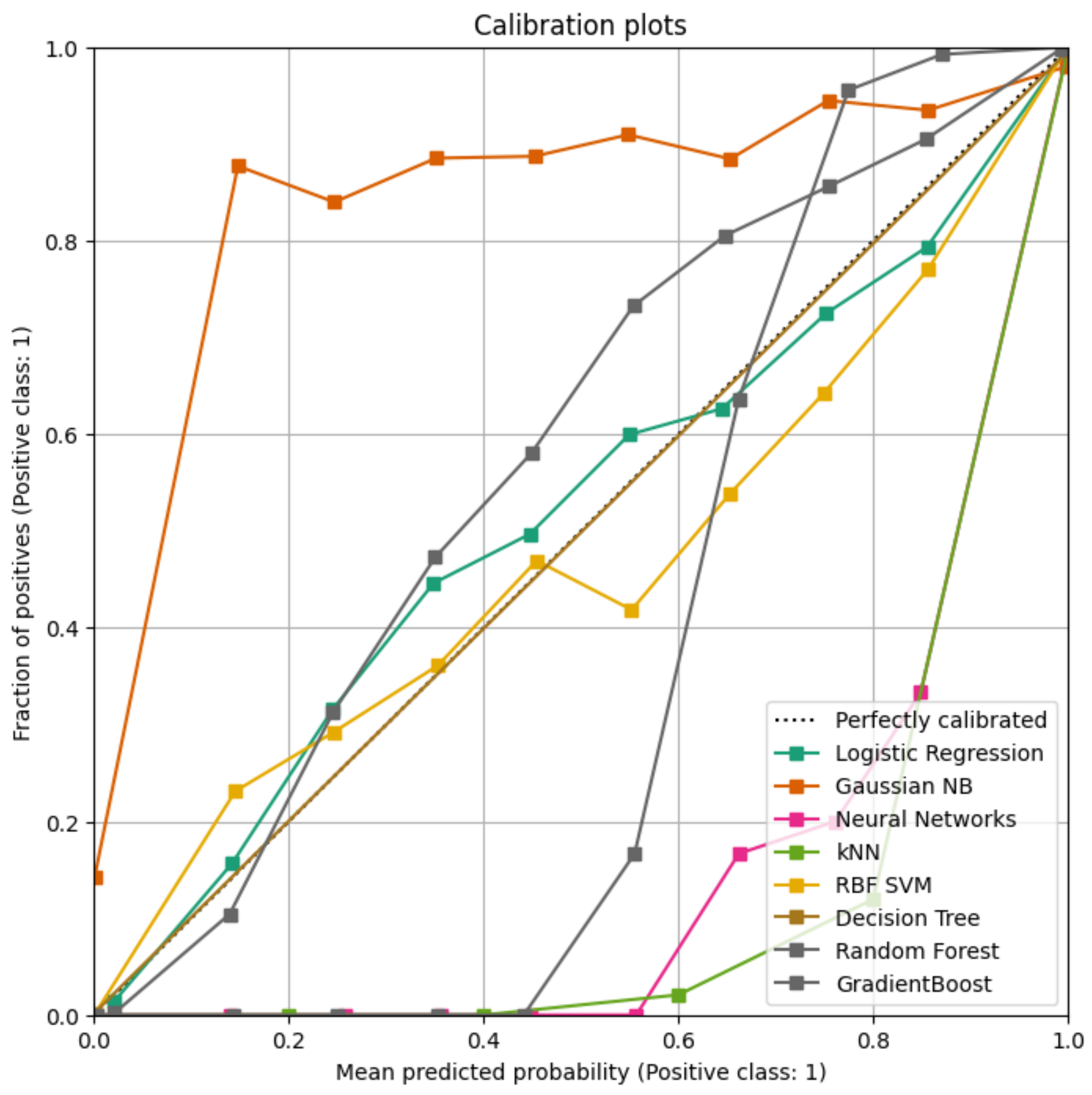
Calibration

Calibration in the context of machine learning, especially for a binary classification problem like credit card fraud detection, refers to the process of ensuring that the predicted probabilities generated by a classification model accurately reflect the actual likelihood of the predicted outcomes, which, in this case, are whether a credit card transaction is fraudulent or not.

In [58]:
```
fig = plt.figure(figsize=(8, 8))
gs = GridSpec(1, 1)
ax_calibration_curve = fig.add_subplot(gs[:1, :1])
colors = plt.cm.get_cmap("Dark2")

calibration_displays = {}

for i, (name, clf) in enumerate(clf_dict.items()):
    try:
        y_proba = clf.predict_proba(X_test)
        display = CalibrationDisplay.from_predictions(
            y_test,
            y_proba[:, 1],
            n_bins=10,
            name=name,
            ax=ax_calibration_curve,
            color=colors(i),
        )
        calibration_displays[name] = display
    except Exception:
        continue

plt.xlim([0, 1])
plt.ylim([0, 1])

plt.grid()

plt.title("Calibration plots")

plt.show()
```

```
/var/folders/vz/c9yq75ls71bg54ncvlqv26dr0000gn/T/ipykernel_6883/11
14094156.py:4: MatplotlibDeprecationWarning: The get_cmap function
was deprecated in Matplotlib 3.7 and will be removed two minor rel
eases later. Use ``matplotlib.colormaps[name]`` or ``matplotlib.co
```

```
lormaps.get_cmap(obj)`` instead.
  colors = plt.cm.get_cmap("Dark2")
```



The plot shows that a model in particular, namely Decision Tree, is perfectly calibrated. This means that its predicted probabilities align precisely with the actual occurrence of fraudulent transactions. For instance, if the perfectly calibrated decision tree model assigns a predicted probability of 0.8 to a transaction being fraudulent, this transaction is indeed fraudulent approximately 80% of the time.

In essence, a perfectly calibrated decision tree model in credit card fraud detection indicates that its predicted probabilities are not only accurate but also trustworthy confidence scores. When a model is perfectly calibrated, it demonstrates a high level of reliability in its probability estimates. This is valuable in fraud detection because it means that when the model provides a probability score, it can be used effectively to prioritize and make decisions about potentially fraudulent transactions, improving the efficiency of fraud detection systems and reducing false alarms.

Hyperparameter Tuning

Hyperparameter tuning refers to the process of systematically searching for the best combination of hyperparameter values for a given machine learning algorithm or model. Hyperparameters are parameters that are not learned from the data but are set before training and affect the learning process and model behaviour. Tuning these hyperparameters is essential because they significantly impact a model's performance.

In the previous analysis, all the models were trained using the default hyperparameter values provided by scikit-learn. However, it's often the case that the default values may not be optimal for a specific dataset or problem. Therefore, hyperparameter tuning becomes crucial to find the most effective set of hyperparameters that can improve a model's performance.

Three different comparison methods were applied to evaluate and select the best models: the Precision-Recall curve, the ROC curve, and calibration.

The Neural Network emerged as the best-performing model according to the Precision-Recall curve. This indicates that it excels in correctly identifying positive instances (e.g., fraudulent transactions) while maintaining high precision, making it suitable for tasks where precision is a critical consideration.

The Random Forest was identified as the best-performing model based on the ROC curve. This suggests that it exhibits strong discrimination capabilities between positive and negative instances, making it effective at distinguishing between fraudulent and non-fraudulent transactions.

The Decision Tree was found to be the perfectly calibrated model, indicating that its predicted probabilities align closely with actual outcomes. This is valuable for assigning reliable confidence scores to transactions, reducing false alarms in fraud detection.

Given that each of these models excelled in different aspects, it was decided to perform hyperparameter tuning on these top three models to further enhance their performance. However, it is important to underline that all the models have already demonstrated strong performance with default hyperparameter values from scikit-learn, so the expected improvement from tuning may be limited. Nonetheless, this process aims to fine-tune the models' architectures to achieve even better results.

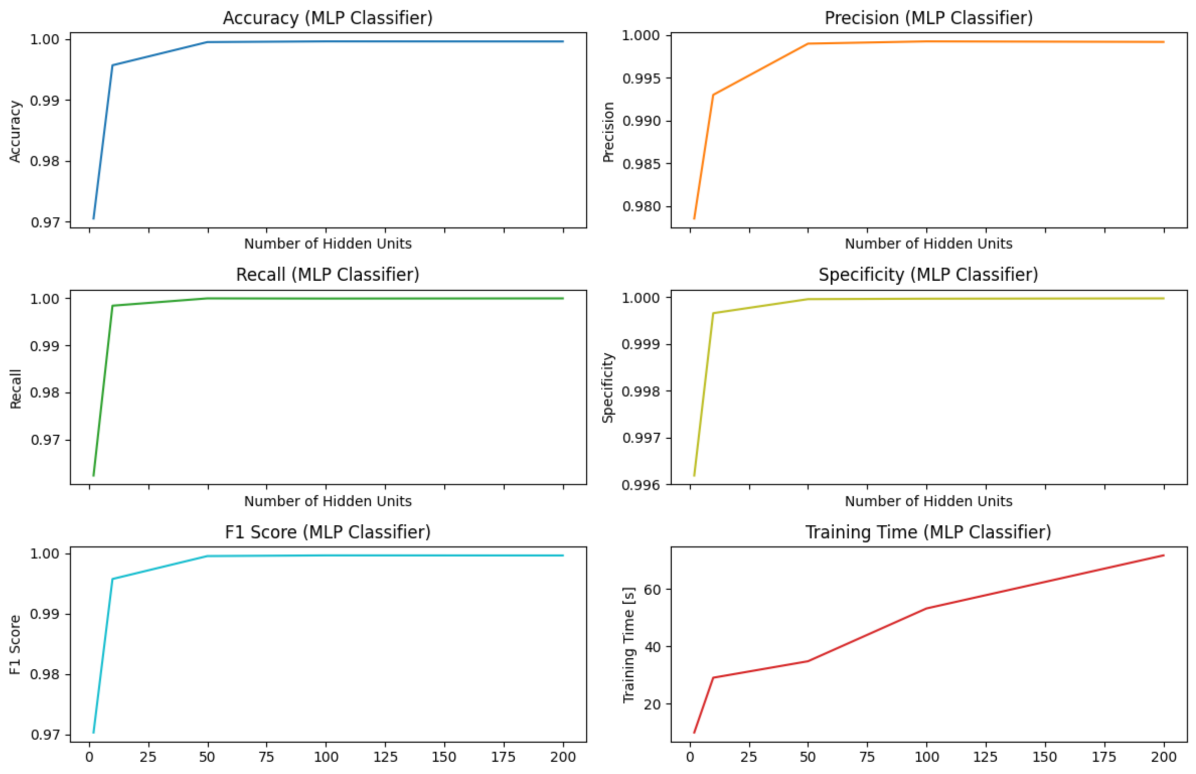
(1) Neural Network

In the context of Neural Network, hyperparameter tuning focuses on optimizing the hidden_layer_sizes hyperparameter. The provided code assesses various configurations using cross-validation and records evaluation metrics (accuracy, precision, recall, specificity, F1 score) and training times for each configuration.

In [37]:
```
# Define the list of hyperparameter values you want to test
list_h_param_values = [2, 10, 50, 100, 200]

# Initialize lists to store evaluation metrics
list_accuracy = []
list_precision = []
list_recall = []
list_specificity = []
list_f1 = []
list_training_time = []

# Perform k-fold cross-validation for each set of hyperparameters
k_fold = 5

for val in list_h_param_values:

    clf = MLPClassifier(hidden_layer_sizes=val, random_state=random

    # Measure training time
    t_start = time.time()
    clf.fit(X_train, y_train)
    t_stop = time.time()
    training_time = t_stop - t_start

    # Perform k-fold cross-validation
    cv_accuracy_scores = cross_val_score(clf, X_train, y_train, cv=
    cv_precision_scores = cross_val_score(clf, X_train, y_train, cv
    cv_recall_scores = cross_val_score(clf, X_train, y_train, cv=k_
    cv_specificity_scores = cross_val_score(clf, X_train, y_train,
    cv_f1_scores = cross_val_score(clf, X_train, y_train, cv=k_fold

    # Compute the mean of the cross-validation scores
    mean_accuracy = cv_accuracy_scores.mean()
    mean_precision = cv_precision_scores.mean()
    mean_recall = cv_recall_scores.mean()
    mean_specificity = cv_specificity_scores.mean()
    mean_f1 = cv_f1_scores.mean()


    # Store the results in the respective lists
    list_accuracy.append(mean_accuracy)
    list_precision.append(mean_precision)
    list_recall.append(mean_recall)
    list_specificity.append(mean_specificity)
    list_f1.append(mean_f1)
    list_training_time.append(training_time)

# Now there are lists with mean evaluation metrics and training tim
```

In [38]:
```
# Create the subplots for the different metrics
fig, ax = plt.subplots(3, 2, figsize=(12, 8), sharex=True)
```

```
ax[0][0].plot(list_h_param_values, list_accuracy, c='tab:blue')
ax[0][0].set_title(f'Accuracy (MLP Classifier)')
ax[0][0].set_ylabel('Accuracy')
ax[0][0].set_xlabel('Number of Hidden Units')

ax[0][1].plot(list_h_param_values, list_precision, c='tab:orange')
ax[0][1].set_title(f'Precision (MLP Classifier)')
ax[0][1].set_ylabel('Precision')
ax[0][1].set_xlabel('Number of Hidden Units')

ax[1][0].plot(list_h_param_values, list_recall, c='tab:green')
ax[1][0].set_title(f'Recall (MLP Classifier)')
ax[1][0].set_ylabel('Recall')
ax[1][0].set_xlabel('Number of Hidden Units')

ax[1][1].plot(list_h_param_values, list_specificity, c='tab:olive')
ax[1][1].set_title(f'Specificity (MLP Classifier)')
ax[1][1].set_ylabel('Specificity')
ax[1][1].set_xlabel('Number of Hidden Units')

ax[2][0].plot(list_h_param_values, list_f1, c='tab:cyan')
ax[2][0].set_title(f'F1 Score (MLP Classifier)')
ax[2][0].set_ylabel('F1 Score')
ax[2][0].set_xlabel('Number of Hidden Units')

ax[2][1].plot(list_h_param_values, list_training_time, c='tab:red')
ax[2][1].set_title(f'Training Time (MLP Classifier)')
ax[2][1].set_ylabel('Training Time [s]')
ax[2][1].set_xlabel('Number of Hidden Units')

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



In the hyperparameter tuning of our neural network, the number of hidden units significantly impacts performance metrics. A notable improvement in accuracy, precision, recall, specificity, and F1 score is observed with increasing hidden units, with the most substantial gains up to 50 units. Beyond this point, performance improvements taper off while computational time continues to rise linearly.

Hence, a configuration of around 50 hidden units (against the default value of 100 by scikit-learn) is recommended, balancing model performance with computational efficiency, which is crucial for practical deployment.

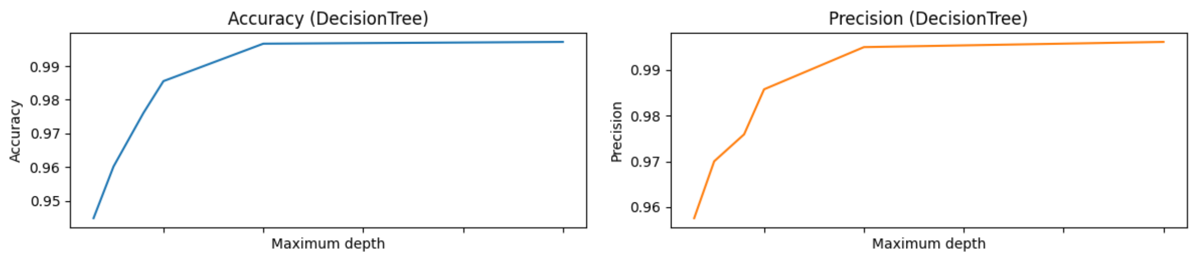
(2) Decision Trees

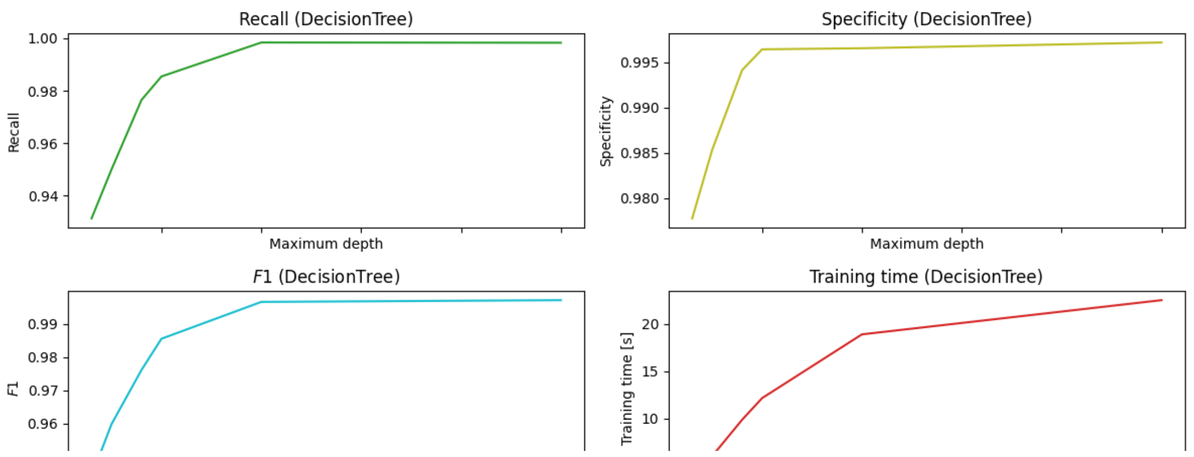
Hyperparameter tuning in decision trees involves adjusting parameters to find the model configuration that results in the best performance. One of the primary hyperparameters is max_depth, which controls the maximum depth of the tree. The default max_depth for a DecisionTreeClassifier in scikit-learn is None, allowing the tree to expand until all leaves are pure. However, allowing a decision tree to grow without any constraints on its depth (max_depth=None) can be computationally expensive and may lead to overfitting. In practice, it is often better to search for an optimal max_depth as part of hyperparameter tuning. By limiting the depth of the tree, the complexity of the model can be reduced, which often helps to improve the generalization to new data. This is indeed done using cross-validation to evaluate the impact of different depths on performance metrics.

In [39]:
```
# Define the list of hyperparameter values
list_h_param_values = [3, 5, 8, 10, 20, 50]

# Initialize lists to store evaluation metrics
list_accuracy = []
list_precision = []
list_recall = []
list_specificity = []
list_f1 = []
list_training_time = []

# Perform k-fold cross-validation for each set of hyperparameters
k_fold = 5

for val in list_h_param_values:

    clf = DecisionTreeClassifier(max_depth=val, random_state=random

    # Measure training time
    t_start = time.time()
    clf.fit(X_train, y_train)
    t_stop = time.time()
    training_time = t_stop - t_start

    # Perform k-fold cross-validation
    cv_accuracy_scores = cross_val_score(clf, X_train, y_train, cv=
    cv_precision_scores = cross_val_score(clf, X_train, y_train, cv
    cv_recall_scores = cross_val_score(clf, X_train, y_train, cv=k_
    cv_specificity_scores = cross_val_score(clf, X_train, y_train,
    cv_f1_scores = cross_val_score(clf, X_train, y_train, cv=k_fold

    # Compute the mean of the cross-validation scores
    mean_accuracy = cv_accuracy_scores.mean()
    mean_precision = cv_precision_scores.mean()
    mean_recall = cv_recall_scores.mean()
    mean_specificity = cv_specificity_scores.mean()
    mean_f1 = cv_f1_scores.mean()

    # Store the results in the respective lists
    list_accuracy.append(mean_accuracy)
    list_precision.append(mean_precision)
    list_recall.append(mean_recall)
    list_specificity.append(mean_specificity)
    list_f1.append(mean_f1)
    list_training_time.append(training_time)
```

In [40]:
```
# Define plot parameters
model_name = 'DecisionTree'
h_param_name = 'Maximum depth'
log_scale = False

# Create the subplots for the different metrics
```

```
fig, ax = plt.subplots(3, 2, figsize=(12, 8), sharex=True)

ax[0][0].plot(list_h_param_values, list_accuracy, c='tab:blue')
if log_scale:
    ax[0][1].set_xscale('log')
ax[0][0].set_title(f'Accuracy ({model_name})')
ax[0][0].set_ylabel('Accuracy')
ax[0][0].set_xlabel(f'{h_param_name}')

ax[0][1].plot(list_h_param_values, list_precision, c='tab:orange')
if log_scale:
    ax[0][1].set_xscale('log')
ax[0][1].set_title(f'Precision ({model_name})')
ax[0][1].set_ylabel('Precision')
ax[0][1].set_xlabel(f'{h_param_name}')

ax[1][0].plot(list_h_param_values, list_recall, c='tab:green')
if log_scale:
    ax[1][0].set_xscale('log')
ax[1][0].set_title(f'Recall ({model_name})')
ax[1][0].set_ylabel('Recall')
ax[1][0].set_xlabel(f'{h_param_name}')

ax[1][1].plot(list_h_param_values, list_specificity, c='tab:olive')
if log_scale:
    ax[1][1].set_xscale('log')
ax[1][1].set_title(f'Specificity ({model_name})')
ax[1][1].set_ylabel('Specificity')
ax[1][1].set_xlabel(f'{h_param_name}')

ax[2][0].plot(list_h_param_values, list_f1, c='tab:cyan')
if log_scale:
    ax[2][0].set_xscale('log')
ax[2][0].set_title(f'$F1$ ({model_name})')
ax[2][0].set_ylabel('$F1$')
ax[2][0].set_xlabel(f'{h_param_name}')

ax[2][1].plot(list_h_param_values, list_training_time, c='tab:red')
if log_scale:
    ax[2][1].set_xscale('log')
ax[2][1].set_title(f'Training time ({model_name})')
ax[2][1].set_ylabel('Training time [s]')
ax[2][1].set_xlabel(f'{h_param_name}')

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





The default value for max_depth in scikit-learn's decision tree classifier is None, which means the nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. The graphs suggest that increasing the maximum depth of the tree initially results in significant gains in various performance metrics, such as accuracy, precision, recall, specificity, and F1 score. However, these metrics plateau beyond a max_depth of 20, indicating that additional depth does not contribute to better performance and can lead to overfitting and unnecessary computational expense.

Therefore, based on the observed trends, setting the max_depth parameter around 20 may offer an optimal balance between model complexity and performance, negating the need for the default unbounded growth of the tree that could lead to overly complex models.

(3) Random Forest

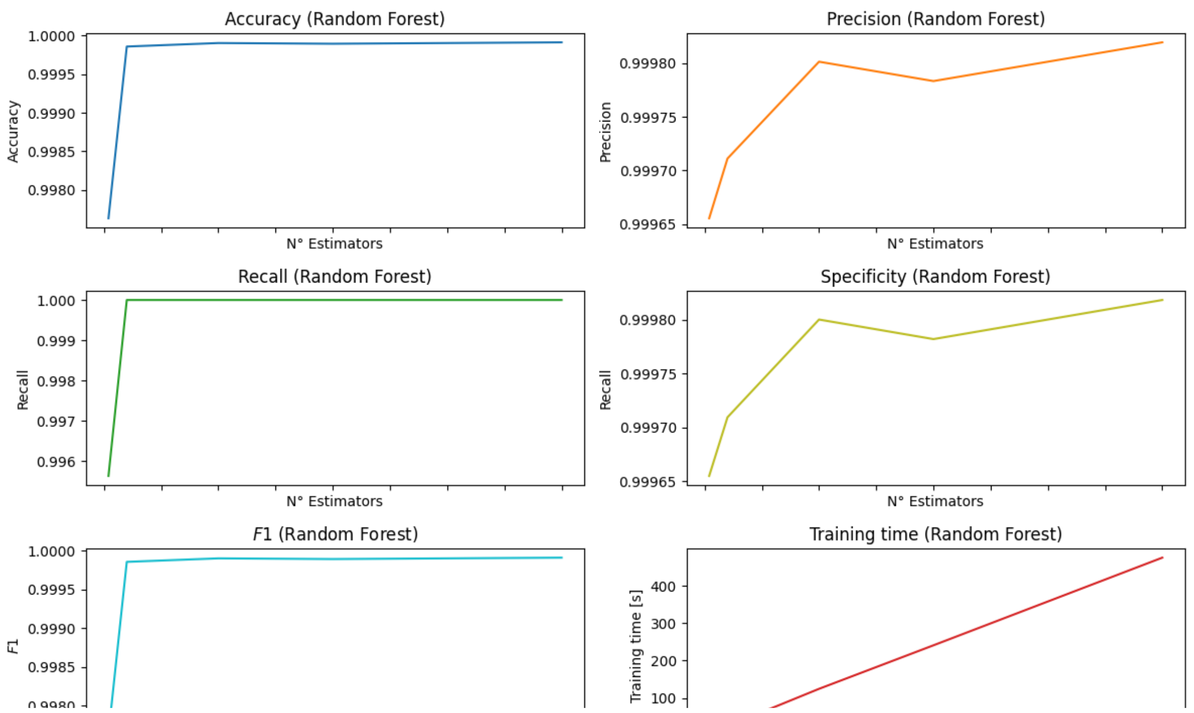
Hyperparameter tuning for Random Forest involves adjusting a set of parameters to optimize the model's performance. A key hyperparameter is n_estimators, which represents the number of trees in the forest.

Notice: We acknowledge that hyperparameter tuning is typically performed using K-fold cross-validation to ensure robustness and generalization of the model. However, in this particular case, we encountered difficulties running the code for hyperparameter tuning with Random Forest using K-fold cross-validation. As an alternative approach, we opted to directly tune the hyperparameters on the test set to illustrate the importance of finding the optimal classifier configuration for achieving the best results.

In [43]:
```
list_h_param_values = [2,10,50,100,200]

list_accuracy = []
list_precision = []
list_recall = []
list_specificity = []
list_f1 = []
list_training_time = []

for val in list_h_param_values:
    clf = RandomForestClassifier(n_estimators=val, random_state=ran

    t_start = time.time()

    clf.fit(X_train, y_train)

    t_stop = time.time()

    y_pred = clf.predict(X_test)

    list_accuracy.append(accuracy_score(y_test, y_pred))
    list_precision.append(precision_score(y_test, y_pred))
    list_recall.append(recall_score(y_test, y_pred))
    list_specificity.append(recall_score(y_test, y_pred, pos_label=
    list_f1.append(f1_score(y_test, y_pred))
    list_training_time.append(t_stop - t_start)

model_name = 'Random Forest'
h_param_name = 'N° Estimators'
log_scale = False
```

In [45]:
```
fig, ax = plt.subplots(3, 2, figsize=(12, 8), sharex=True)

ax[0][0].plot(list_h_param_values, list_accuracy, c='tab:blue')
if log_scale:
    ax[0][1].set_xscale('log')
ax[0][0].set_title(f'Accuracy ({model_name})')
ax[0][0].set_ylabel('Accuracy')
ax[0][0].set_xlabel(f'{h_param_name}')

ax[0][1].plot(list_h_param_values, list_precision, c='tab:orange')
if log_scale:
    ax[0][1].set_xscale('log')
ax[0][1].set_title(f'Precision ({model_name})')
ax[0][1].set_ylabel('Precision')
ax[0][1].set_xlabel(f'{h_param_name}')

ax[1][0].plot(list_h_param_values, list_recall, c='tab:green')
if log_scale:
    ax[0][1].set_xscale('log')
ax[1][0].set_title(f'Recall ({model_name})')
ax[1][0].set_ylabel('Recall')
```

```
ax[1][0].set_xlabel(f'{h_param_name}')

ax[1][1].plot(list_h_param_values, list_specificity, c='tab:olive')
if log_scale:
    ax[0][1].set_xscale('log')
ax[1][1].set_title(f'Specificity ({model_name})')
ax[1][1].set_ylabel('Recall')
ax[1][1].set_xlabel(f'{h_param_name}')

ax[2][0].plot(list_h_param_values, list_f1, c='tab:cyan')
if log_scale:
    ax[0][1].set_xscale('log')
ax[2][0].set_title(f'$F1$ ({model_name})')
ax[2][0].set_ylabel('$F1$')
ax[2][0].set_xlabel(f'{h_param_name}')

ax[2][1].plot(list_h_param_values, list_training_time, c='tab:red')
if log_scale:
    ax[0][1].set_xscale('log')
ax[2][1].set_title(f'Training time ({model_name})')
ax[2][1].set_ylabel('Training time [s]')
ax[2][1].set_xlabel(f'{h_param_name}')

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



Typically, increasing 'n_estimators' improves model performance because it averages more decisions from more trees, reducing variance and making the model more robust to overfitting. However, this comes with a trade-off in computational cost and time, as seen in the training time graph.

It appears that the metrics of accuracy, precision, recall, specificity, and F1 score plateau beyond a certain number of estimators, as there is little to no improvement in the metrics after the initial rise. Therefore, the best 'n_estimators' hyperparameter would be at the point just before the performance metrics plateau, where additional trees no longer contribute to a significant increase in model accuracy, but before the training time becomes excessively long.

Unsupervised Machine Learning

Unsupervised machine learning is a type of machine learning where the algorithm is not provided with labeled training data. In contrast to supervised learning, where the algorithm is trained on input-output pairs, unsupervised learning deals with unlabeled data and aims to find patterns, relationships, or structures within the data.

K-means Clustering

K-means clustering is a popular unsupervised machine learning algorithm used for partitioning a dataset into K distinct, non-overlapping subsets (clusters). The goal is to group similar data points together and assign them to clusters based on certain features or characteristics.

The main objective of k-means is to minimize the intra-cluster variance, making data points within the same cluster as similar as possible, while maximizing the inter-cluster variance, ensuring that different clusters are distinct from each other. As an iterative algorithm, k-means assigns data points to clusters and refines these assignments until convergence.

K-means is an unsupervised learning algorithm, meaning it doesn't rely on labelled data for training. Instead, it groups data points into clusters based solely on the similarity of their features. So, the 'Class' label needs to be dropped.

In [204]:
```
df
```

Out[204]:

	V1	V2	V3	V4	V5	V6	V7	
0	-0.260648	-0.469648	2.496266	-0.083724	0.129681	0.732898	0.519014	-0.1300
1	0.985100	-0.356045	0.558056	-0.429654	0.277140	0.428605	0.406466	-0.1331
2	-0.260272	-0.949385	1.728538	-0.457986	0.074062	1.419481	0.743511	-0.0955
3	-0.152152	-0.508959	1.746840	-1.090178	0.249486	1.143312	0.518269	-0.0651
4	-0.206820	-0.165280	1.527053	-0.448293	0.106125	0.530549	0.658849	-0.2126
...	...	...	...	...	...	...	...	
568625	-0.833437	0.061886	-0.899794	0.904227	-1.002401	0.481454	-0.370393	0.1896
568626	-0.670459	-0.202896	-0.068129	-0.267328	-0.133660	0.237148	-0.016935	-0.1477
568627	-0.311997	-0.004095	0.137526	-0.035893	-0.042291	0.121098	-0.070958	-0.0199
568628	0.636871	-0.516970	-0.300889	-0.144480	0.131042	-0.294148	0.580568	-0.2077
568629	-0.795144	0.433236	-0.649140	0.374732	-0.244976	-0.603493	-0.347613	-0.3408

This dataframe represents the dataset used prior to implementing supervised machine learning algorithms. It includes the 'Class' column, and notably, the 'Amount' column has not undergone standardization yet. The standardization process for the 'Amount' feature is carried out in the subsequent cell:

In [205]:
```
# Create a StandardScaler instance
scaler = StandardScaler()

# Fit and transform the 'Amount' column
df['Amount'] = scaler.fit_transform(df['Amount'].values.reshape(-1,

# Now, the 'Amount' column is standardized
```

In [206]:
```
df.groupby('Class').size()
```

Out[206]:
```
Class
0    275190
1    276845
dtype: int64
```

The class distribution is balanced across the dataframe

In [207]:
```
true_labels = df['Class'].values
true_labels
```

Out[207]:
```
array([0, 0, 0, ..., 1, 1, 1])
```

In [208]:
```
df0 = df.drop('Class', axis=1)
```

In [209]:
```
df0.sample(10)
```

Out[209]:

	V1	V2	V3	V4	V5	V6	V7	
6441	1.071131	-0.654998	0.884247	-0.772261	-0.118322	0.141360	0.208981	-0.1742
436495	-0.015866	0.412201	-0.702588	0.644499	-0.406872	-0.988559	-0.569535	0.1860
36180	0.032000	-0.378598	1.141759	-1.633759	0.187244	0.005456	0.572876	-0.1797
549327	-1.125572	0.647434	-1.174139	0.955707	-1.014681	-0.838093	-0.786721	0.7537
403227	-0.917491	-0.045286	-0.722684	0.231797	0.037503	0.432348	-0.631601	-0.6196
57149	0.271063	-0.166401	0.951616	-0.288099	0.440773	0.488754	0.530680	-0.1407
10757	1.065523	-0.589470	0.267879	-0.963775	1.010678	2.347407	0.089001	0.0611
296002	0.687698	-0.134332	0.766847	0.336897	0.398240	0.367842	0.468898	-0.1672
505566	-0.399419	-0.003737	0.351294	-0.059949	-0.067431	0.360703	-0.028634	-0.0643
446810	-0.013100	0.792307	-0.851967	1.305874	0.508757	-1.691169	-0.082156	-0.0337

10 rows × 27 columns

In [210]:
```
for column in df0.columns:
    desc_df = df[column].describe()
    print(f"description for column '{column}': \n\n{desc_df}\n")
```

```
description for column 'V1':

count    552035.000000
mean          0.001960
std           0.978918
min          -3.495584
25%          -0.550592
50%          -0.094628
75%           0.795346
max           2.229046
Name: V1, dtype: float64

description for column 'V2':

count    552035.000000
mean         -0.022230
std           0.973392
min         -49.966572
25%          -0.488964
```

K-means encounters challenges when faced with datasets of high dimensions. This phenomenon is known as the "curse of dimensionality," where all data points become approximately equidistant from each other, hindering the algorithm's effectiveness. To overcome this limitation, it is advisable to employ k-means in scenarios characterized by lower dimensions. Alternatively, when such a dataset is not available (like in this case), the performance of this algorithm can be enhanced by applying a dimensionality reduction technique, such as Principal Component Analysis (PCA).

PCA is a powerful method aimed at simplifying and summarizing the complexity of high-dimensional datasets (like the analyzed one), which is a 27-dimensional dataset). In simpler terms, PCA transforms the original features of the data into a new set of uncorrelated variables, called principal components. These components capture the most significant sources of variation in the data. By retaining only a subset of the principal components that contribute most to the variance, PCA enables a reduction in the dimensionality of the dataset. This not only addresses the curse of dimensionality but also facilitates a more efficient and meaningful application of algorithms like k-means.

In [211]:
```
# Initialize PCA with enough components to capture 95% variance
pca = PCA(n_components=0.95)

# Fit PCA on the data
df_pca = pca.fit_transform(df0)

# The number of components that hold 95% of the variance
num_components = pca.n_components_

print(f"Number of components to retain 95% of the variance: {num_co
```

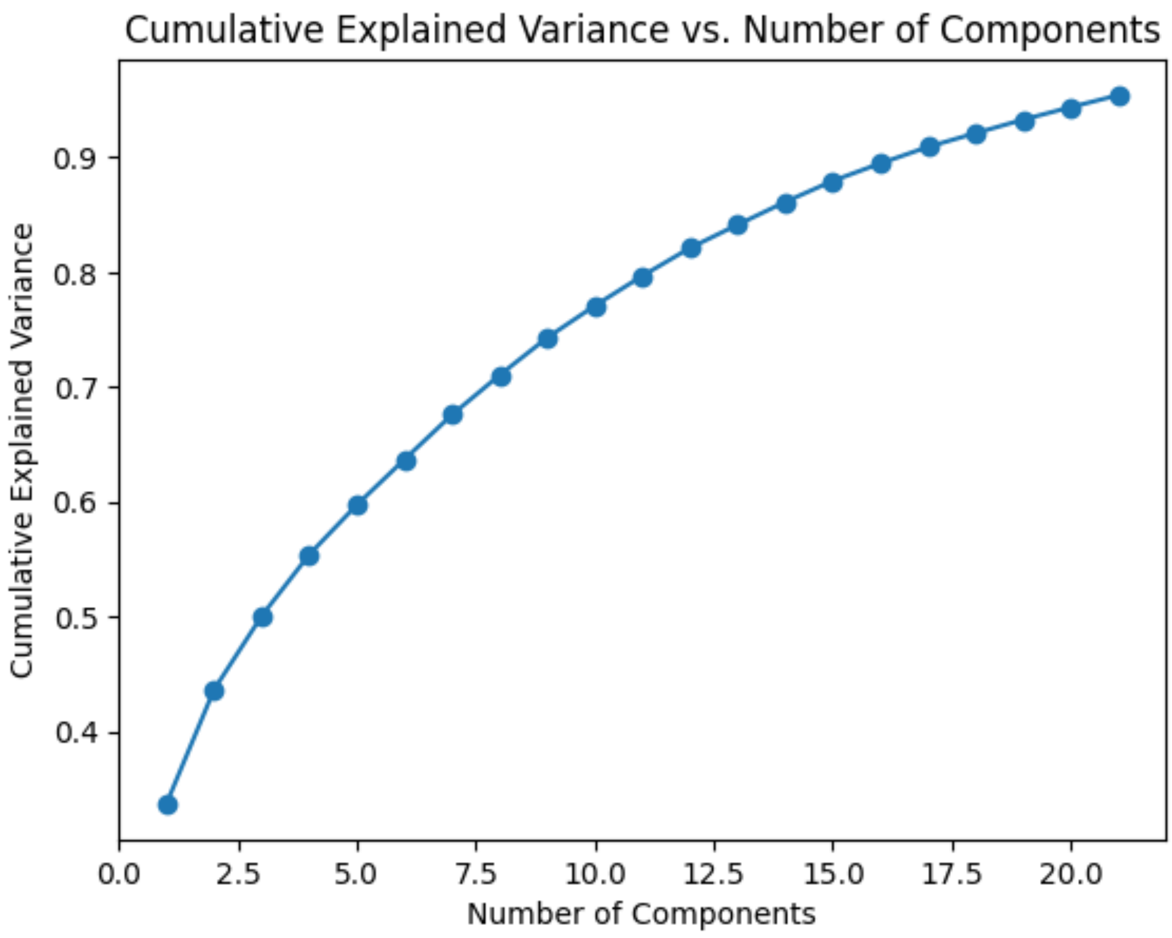
```
Number of components to retain 95% of the variance: 21
```

In [212]:
```
# Apply PCA for dimensionality reduction
n_components = 21 # Choose the desired number of components
pca = PCA(n_components=n_components)
df_pca = pca.fit_transform(df0)

# Evaluate the explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_
print(f"Explained Variance Ratio: {explained_variance_ratio}")

# 'df_pca' now contains the transformed features with reduced dimen
```

```
Explained Variance Ratio: [0.33702625 0.09932781 0.06414071 0.0535
0812 0.04325514 0.03953711
 0.03893883 0.03450695 0.03259687 0.02784273 0.02599399 0.02429325
 0.02022688 0.01983013 0.01836384 0.01520503 0.01436313 0.01197062
 0.01178995 0.01085109 0.01043603]
```

The explained variance ratio in the context of Principal Component Analysis (PCA) provides information about the proportion of variance in the original data that is retained by each principal component.

```
In [213]: cumulative_explained_variance = explained_variance_ratio.cumsum()
          plt.plot(range(1, len(cumulative_explained_variance) + 1), cumulati
          plt.xlabel('Number of Components')
          plt.ylabel('Cumulative Explained Variance')
          plt.title('Cumulative Explained Variance vs. Number of Components')
          plt.show()
```



This graph is a plot of cumulative explained variance against the number of components in a Principal Component Analysis (PCA) setup. In PCA, explained variance measures the amount of variance in the data that is captured by each principal component. The cumulative explained variance is the sum of explained variances of all components up to a certain number, indicating the total variance captured by the first N components.

The x-axis of the graph represents the number of principal components, and the y-axis represents the cumulative explained variance. As more components are added, they capture more of the variance in the dataset. The graph tends to plateau as the number of components increases, indicating that each additional component contributes less to capturing the variance.

In situations where the curve of the cumulative explained variance does not show a clear "elbow" like in this case — that is a point where the rate of increase in explained variance significantly slows down — choosing the number of components to retain can be challenging. An "elbow" usually indicates that additional components beyond this point contribute little to the explanation of the data's variance. In this case, as no elbow is visible, it has been decided on a threshold of cumulative explained variance that is wished to achieve (95%), and the smallest number of components that reach this threshold is chosen. This explains why 21 principal components are retained.

In [214]:
```
# Convert the NumPy array to a DataFrame
df_pca = pd.DataFrame(df_pca, columns=[f'PC{i}' for i in range(1, d

# Sample 10 rows from the DataFrame
sample_df_pca = df_pca.sample(10)
print(sample_df_pca)
```

```
             PC1       PC2       PC3       PC4       PC5       PC6
PC7  \
293913  1.191503  0.767004  1.694573 -0.774532 -0.334114 -0.721171
1.236631
235837  2.598775 -0.351812 -1.799949 -0.461361 -1.201580 -1.127557
0.648749
238246  3.093419  0.895395  0.620109 -3.214607 -0.110320  1.105029
-3.856209
413872 -0.678183 -0.743639  0.278419  0.000471 -0.628447 -0.775848
0.480661
324781  1.111950  0.685576  1.476062 -0.739720 -0.437634 -0.956714
0.879737
15906   5.540288  2.847101  4.148882  4.218453 -0.948889  0.102915
1.577231
212925  2.355839 -0.072141  1.372472 -0.032055  0.449441 -0.531650
-0.169443
349071 -0.287751 -1.110834 -0.819383  0.198660 -0.863821  1.301511
-2.060286
231212  2.189401 -0.114587 -0.752426 -0.655747 -0.144312  0.812893
-0.322770
503529 -0.892385 -0.790092  0.598267 -0.157356 -0.399586  0.001446
0.270892
```

```
              PC8       PC9      PC10  ...      PC12       PC13       
PC14  \
293913 -0.469585  0.096783  1.277298  ...  0.414621   0.086289  0.
034306
235837  0.810482  0.187110  0.618220  ...  0.962259   0.953549 -1.
503324
238246 -2.687573 -0.661777 -2.173013  ... -2.033113  -0.067157 -0.
892980
413872 -0.906299 -0.515482 -0.454152  ...  0.374154  -0.473721  0.
290384
324781 -0.200142  0.029290  0.892849  ...  0.466187   0.311248 -0.
229049
15906   1.214141 -2.936415 -0.156992  ...  3.385032 -15.736851 -1.
511892
212925 -1.052463  0.274517  0.931681  ... -0.178249   0.043346 -0.
420925
349071 -0.329106 -0.372212  0.032636  ...  2.063619  -0.592275 -0.
221266
231212  1.129143  0.293866  0.153216  ...  0.657292  -0.200007 -1.
303942
503529 -0.013073  0.150361 -0.515789  ...  1.296188   0.444774  0.
461899

             PC15      PC16      PC17      PC18      PC19      PC20  
PC21  
293913 -0.129029  0.403789 -0.333106  0.175218 -0.121780 -0.255956  
0.210370  
235837  0.070898 -0.237714  0.359212 -0.166896  0.273291 -0.464172  
0.404352  
238246  2.164301 -2.147405 -1.066563  2.377255 -1.470917 -0.052739 
-0.947775  
413872 -0.727271  0.142475 -0.266126  0.086344  0.090599 -0.784419 
-0.358815  
324781 -0.343563  0.208302 -0.076168 -0.168620 -0.310215 -0.327756  
0.505217  
15906   4.584398 -1.573919  0.826311 -2.059501 -2.051748  0.254532 
-2.034804  
212925 -1.151804 -1.112579 -0.327004  0.250735  0.440487  0.775196 
-0.579773  
349071  0.051325  1.387182  0.027285  0.489625  0.066660  0.153028  
0.403461  
231212 -1.114582 -0.521763  0.427428  0.561507  0.483048 -0.238587 
-0.026266  
503529  0.835667  1.154290  0.251841 -0.178253  0.600072  0.701729 
-0.472357  

[10 rows x 21 columns]
```

K-Means Clustering, with k = 2

For this binary classification task, k-means clustering is initially applied with k=2 under the assumption that the data could be divided into two distinct groups, namely fraudulent and non-fraudulent transactions.

However, clustering (including k-means) is an unsupervised learning technique primarily used for discovering the inherent groupings in data. Even in scenarios where class labels are equally distributed within a dataset (like this one), clustering might not be the optimal approach for binary classification tasks. This is primarily because clustering algorithms do not consider the actual class labels during the grouping process; they only analyze the distances between data points. Consequently, clusters identified by such algorithms may not align with the true underlying class distinctions, particularly if the classes are not naturally formed into distinct, well-separated groups based on the input features alone. Supervised learning methods, which are designed to leverage the class labels during the training phase, are typically more adept at recognizing and utilizing the complex patterns that differentiate the classes.

In [215]:
```
kmeans= KMeans(n_clusters= 2, random_state = random_seed, n_init =
```

Clustering Evaluation Metrics:

Metrics	Description
Homogeneity Score	Measures the extent to which each cluster contains only members of a single class.
Completeness Score	Measures the degree to which all members of a true class are assigned to the same cluster.
V-Measure Score	The harmonic mean of homogeneity and completeness, representing a balance between the two.
Silhouette Score	Assesses how similar an object is to its own cluster compared to other clusters.
Calinski-Harabasz Score	Measures the ratio of between-clusters dispersion to within-cluster dispersion.

Building blocks for the metrics:

- $\text{homogeneity} = \text{A higher score indicates more homogeneous clusters}$
- $$\text{completeness} = \text{A higher score indicates better assignment of data points to their own clusters}$$
- $$\text{V-measure} = \text{A higher score indicates a better balance of homogeneity and completeness}$$
- $\text{silhouette} = \text{A higher score indicates better separated clusters}$
- $$\text{calinski-harabasz} = \text{A higher score denotes clusters that are well separated and compact}$$

Homogeneity, Completeness, V-Measure:

These metrics are external indices that evaluate clustering performance based on known labels. They require the true class labels to be known a priori. Homogeneity checks if each cluster contains only members of a single class. Completeness assesses whether all members of a given class are assigned to the same cluster. V-Measure is the harmonic mean of homogeneity and completeness, offering a single score that balances the two

Silhouette Score:

This is an internal index that measures how similar an object is to its own cluster (cohesion) compared to other clusters (separation). It does not require knowledge of true labels. The silhouette score provides insight into the distance between the resulting clusters. A high silhouette score indicates that objects are well matched to their own cluster and poorly matched to neighbouring clusters.

Notice: The calculation of the silhouette score was attempted on several occasions; however, due to its significant computational expense, it was determined that the calculation would not be performed.

Calinski-Harabasz Score:

Also an internal index, this score is a ratio of the sum of between-cluster dispersion to within-cluster dispersion. Like the silhouette score, it does not require knowledge of true labels. High values of the Calinski-Harabasz score generally indicate that the clusters are dense and well separated, which is a sign of a good clustering.

K-means Clustering Evaluation

In [216]:
```
kmeans.cluster_centers_
```

Out[216]:
```
array([[-3.22265425e+00,  7.94984755e-02,  1.16403912e-01,
         2.60662620e-01, -1.05668731e-01,  5.65040833e-03,
         5.14697897e-02,  3.12754538e-02, -8.09239344e-02,
         1.55381363e-02,  2.80815515e-02,  6.74216144e-02,
        -1.18727235e-02,  2.96009456e-02, -4.74753998e-02,
         5.00480497e-02, -5.85155620e-02,  4.72351141e-03,
         4.18923799e-02, -6.03553957e-03, -3.24468162e-02],
       [ 2.00673686e+00, -4.95034555e-02, -7.24843569e-02,
        -1.62313809e-01,  6.57995931e-02, -3.51849184e-03,
        -3.20500793e-02, -1.94751286e-02,  5.03910844e-02,
        -9.67554958e-03, -1.74862955e-02, -4.19832314e-02,
         7.39310828e-03, -1.84324175e-02,  2.95627851e-02,
        -3.11647662e-02,  3.64374600e-02, -2.94131599e-03,
        -2.60862557e-02,  3.75831187e-03,  2.02045324e-02]])
```

This retrieves the coordinates of the cluster centers. Knowing the cluster centers can help understand the central points around which the data points are clustered. Unfortunately, this is not the the case here, as clustering is performed in a 21-dimensional space. Hence, visualization will not be neither possible nor helpful.

In [217]:
```
predicted_clusters= kmeans.predict(df_pca)
predicted_clusters
```

Out[217]:
```
array([1, 1, 1, ..., 1, 1, 0], dtype=int32)
```

In [218]:
```
from sklearn.metrics import homogeneity_score, completeness_score,

# Calculate the scores
homogeneity = homogeneity_score(true_labels, predicted_clusters)
completeness = completeness_score(true_labels, predicted_clusters)
v_measure = v_measure_score(true_labels, predicted_clusters)

# Print the scores
print(f"Homogeneity score   = {homogeneity:.4f}")
print(f"Completeness score  = {completeness:.4f}")
print(f"V-Measure score     = {v_measure:.4f}")
```

```
Homogeneity score   = 0.5650
Completeness score  = 0.5881
V-Measure score     = 0.5763
```

These scores indicate that while the k-means clustering has performed moderately well in grouping the dataset, there's still a significant amount of overlap between clusters, and not all members of the true classes have been perfectly captured within the clusters. This reinforces the notion that while k-means can provide insights into the general structure of the data, it may not always align perfectly with the actual binary class labels in supervised learning tasks.

In [221]:
```
calinski = calinski_harabasz_score(df_pca, predicted_clusters)
print(f"Calinski-Harabaz Index = {calinski:.4f}")
```

```
Calinski-Harabaz Index = 205398.3571
```

A score of 205398.3571 is quite high, suggesting that the two clusters created by the KMeans model with n_clusters=2 on the PCA-reduced dataset df_pca are indeed well-defined and distinct from each other. This indicates a good clustering structure.

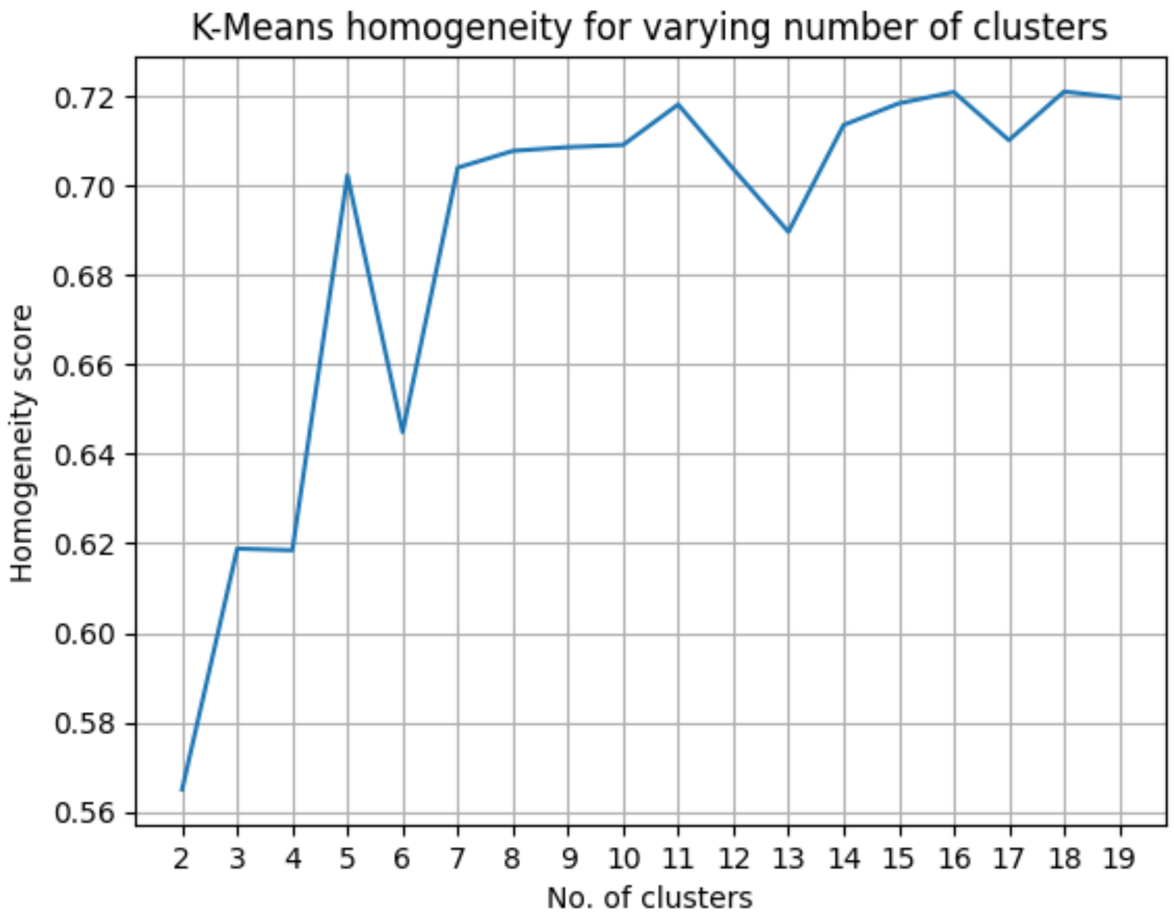
Determining the optimal number of clusters for the k-means clustering model

The following cell iterates over a range of potential cluster counts (from 2 to 19), applying the k-means algorithm with each count. It then calculates the homogeneity score for each configuration, which measures how consistent each cluster is with respect to the true labels. These scores are stored in a list, allowing for comparison to identify the number of clusters that results in the most homogeneous grouping.

In [222]:
```
n_clusters = list(range(2, 20))
homogeneity_values = []

for k in n_clusters:
    # Register start time
    t_start = time.time()

    # Fit clustering algorithm with explicit n_init value
    kmeans = KMeans(n_clusters=k, n_init=10, random_state=random_se

    # Compute score with current configuration and append it to the
    homogeneity_values.append(homogeneity_score(true_labels, kmeans

    # Register end time
    t_stop = time.time()

    # Print elapsed time
    print(f"Elapsed time: {t_stop - t_start:.5f} seconds (k: {k:2d}
```

```
Elapsed time: 3.14486 seconds (k:  2)
Elapsed time: 3.56702 seconds (k:  3)
Elapsed time: 4.48120 seconds (k:  4)
Elapsed time: 5.54105 seconds (k:  5)
Elapsed time: 5.81451 seconds (k:  6)
Elapsed time: 5.40205 seconds (k:  7)
Elapsed time: 5.72456 seconds (k:  8)
Elapsed time: 6.39246 seconds (k:  9)
Elapsed time: 6.70149 seconds (k: 10)
Elapsed time: 6.66968 seconds (k: 11)
Elapsed time: 7.26880 seconds (k: 12)
Elapsed time: 8.30647 seconds (k: 13)
Elapsed time: 8.93130 seconds (k: 14)
Elapsed time: 8.90630 seconds (k: 15)
Elapsed time: 9.94285 seconds (k: 16)
Elapsed time: 9.64651 seconds (k: 17)
Elapsed time: 11.34215 seconds (k: 18)
Elapsed time: 11.10650 seconds (k: 19)
```

In [223]:
```
plt.figure()

plt.plot(n_clusters, homogeneity_values)
plt.xticks(n_clusters)

plt.title('K-Means homogeneity for varying number of clusters')
plt.xlabel('No. of clusters')
plt.ylabel('Homogeneity score')
plt.grid()
plt.show()
```



If the plot does not show a clear elbow, like in this case, then the number of clusters that corresponds to the highest homogeneity score will be chosen. It's important to also consider the computational cost, interpretability, and practical utility of the clustering solution when many clusters are involved.

This graph plots the homogeneity values for different numbers of clusters in a k-means clustering algorithm, indicating that the number of clusters maximizing homogeneity is 16. This might imply that the clusters are not easily separable into two distinct categories. Let's see what happens when k = 16.

K-Means Clustering, with k = 16

In [224]:
```
kmeans= KMeans(n_clusters= 16, random_state = random_seed, n_init =
```

In [225]:
```
predicted_clusters= kmeans.predict(df_pca)
predicted_clusters
```

Out[225]:
```
array([ 1, 14, 14, ..., 12,  3,  0], dtype=int32)
```

In [226]:
```
from sklearn.metrics import homogeneity_score, completeness_score,

# Calculate the scores
homogeneity = homogeneity_score(true_labels, predicted_clusters)
completeness = completeness_score(true_labels, predicted_clusters)
v_measure = v_measure_score(true_labels, predicted_clusters)

# Print the scores
print(f"Homogeneity score   = {homogeneity:.4f}")
print(f"Completeness score  = {completeness:.4f}")
print(f"V-Measure score     = {v_measure:.4f}")
```

```
Homogeneity score   = 0.7209
Completeness score  = 0.2107
V-Measure score     = 0.3261
```

The increase in the homogeneity score when k=16 suggests that the clusters are more internally consistent. In other words, members of each cluster are more similar to each other than to members of other clusters.

However, the decrease in the completeness score indicates that all data points that are members of a given class are not necessarily assigned to the same cluster. A lower completeness score with a higher number of clusters can occur because the algorithm might be splitting what could be considered a single class in the context of binary classification into multiple smaller clusters.

The V-Measure is a harmonic mean of homogeneity and completeness. The decrease in this score when k=16, despite the increase in homogeneity, is due to the significant drop in the completeness score. The V-Measure aims to balance both homogeneity and completeness, but in this case, the loss in completeness outweighs the gain in homogeneity. While a higher number of clusters may better capture the nuanced structure of the data (as indicated by higher homogeneity), it might do so at the expense of grouping all members of a single class together (as indicated by lower completeness).

In [227]:
```
calinski = calinski_harabasz_score(df_pca, predicted_clusters)
print(f"Calinski-Harabaz Index = {calinski:.4f}")
```

```
Calinski-Harabaz Index = 51871.5231
```

This indicates that the clustering solution with just two clusters has a better ratio of between-clusters dispersion to within-cluster dispersion, meaning the two clusters are more distinct and separated from each other compared to the sixteen clusters scenario.

Hierarchical Clustering

Hierarchical clustering presents a distinct approach to data clustering, in contrast to the K-Means algorithm. Unlike K-Means, hierarchical clustering methods do not require an operator-selected value, 'k,' which denotes the number of desired clusters. Determining the optimal 'k' can be a challenging task, and it can significantly impact the clustering outcomes. Agglomerative hierarchical clustering, for example, constructs clusters in a bottom-up manner. Initially, each data point forms its own cluster, and pairs of clusters are progressively merged based on their similarity, often measured using distance metrics like Euclidean distance. This merging process continues until only one cluster remains. Subsequently, one can explore the layers of the resulting tree-like structure, called a dendrogram, to select the layer that yields the most suitable clustering outcome. Conversely, divisive hierarchical clustering begins with a single cluster encompassing all data points and divides clusters based on the chosen distance metric until each data point resides in its individual cluster.

Hierarchical clustering is a powerful algorithm for detecting intricate structures within data, but it comes with a significant computational cost, especially for large datasets. To address this, a common approach is to perform hierarchical clustering on a smaller, representative subset of the original dataset. An undersampled dataset containing only 5% of the data from the original dataframe (df) will be used to train the model.

In [228]:
```
undersampled_df = df.sample(frac=0.05, random_state=random_seed)  #

undersampled_indices = undersampled_df.index.tolist()  #Memorize th

#Reset index of the undersampled df
undersampled_df.reset_index(drop=True, inplace=True)

#Now, undersampled_df contains 5% of data from df
```

Then, the target label is removed to train the unsupervised machine learning algorithm.

In [229]:
```
undersampled_df0 = undersampled_df.drop('Class', axis=1) #Remove th
```

PCA is applied on the undersampled_df not containing the target label.

In [230]:
```
# Initialize PCA with enough components to capture 95% variance
pca = PCA(n_components=0.95)

# Fit PCA on the data
under_df_pca = pca.fit_transform(undersampled_df0)

# The number of components that hold 95% of the variance
num_components = pca.n_components_

print(f"Number of components to retain 95% of the variance: {num_co
```

```
Number of components to retain 95% of the variance: 21
```

In [231]:
```
# Apply PCA for dimensionality reduction
n_components = 21 # Choose the desired number of components
pca = PCA(n_components=n_components)
under_df_pca = pca.fit_transform(undersampled_df0)

# Evaluate the explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_
print(f"Explained Variance Ratio: {explained_variance_ratio}")

# 'df_pca' now contains the transformed features with reduced dimen
```

```
Explained Variance Ratio: [0.34245551 0.09520564 0.06532428 0.0539
4309 0.04332518 0.04088766
 0.03915442 0.03516019 0.03252607 0.02772608 0.0264931  0.02314137
 0.01997628 0.01917143 0.01780479 0.01536633 0.01426002 0.01242382
 0.01116221 0.01063238 0.00993416]
```

In [232]:
```
under_df_pca
```

Out[232]:
```
array([[ 1.61980694,  0.25675565, -0.16044942, ...,  0.21091545,
         0.56099981,  0.30992252],
       [ 2.19673978,  0.24826242, -0.0357739 , ...,  0.41485781,
        -0.36541061,  0.19526909],
       [ 3.03824939,  0.34742613, -0.8983387 , ..., -0.54824479,
        -0.60610867, -0.13468656],
       ...,
       [-1.20383508, -0.08619672,  0.07503606, ..., -0.0613309 ,
        -0.19497909,  0.0465232 ],
       [-3.35947994, -0.90352243,  1.39197424, ..., -0.13517723,
        -0.11300287, -0.32322013],
       [-2.50301236,  0.55993921,  1.02888286, ..., -0.15229338,
         0.40189539,  0.41777626]])
```

In [233]:
```
# Convert the NumPy array to a DataFrame
under_df_pca = pd.DataFrame(under_df_pca, columns=[f'PC{i}' for i i

# Sample 10 rows from the DataFrame
sample_df_pca = under_df_pca.sample(10)
print(sample_df_pca)
```

```
            PC1       PC2       PC3       PC4       PC5       PC6
PC7  \
2797   1.887419 -0.156557 -0.526191 -0.294005  0.030276 -1.687065
0.610024
256   -3.104852 -0.785926  0.741256  0.997857  0.961345 -0.115069
-0.067284
19004 -6.788392 -3.189068 -0.403389 -3.043377 -0.639742 -0.162461
1.402614
9785   1.739646  0.220227 -0.979799 -1.392085  1.456601  0.809514
0.259509
12650 -2.511310  0.207974 -1.211816  1.318413 -0.258351 -0.512058
1.307230
20266 -1.536492  0.207991 -0.040193  0.321566 -0.676124 -0.860903
0.355676
17732 -2.518096 -0.545110 -0.184404  0.957256  1.683070  0.553884
-0.045230
24464  2.797131 -0.194149  1.470756 -0.030775  0.115761  1.478060
-0.806232
349   -7.006903 -3.669986 -0.601501 -3.982691 -0.266363  0.586912
1.007200
```

In [234]:
```
under_df_pca
```

Out[234]:

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC
0	1.619807	0.256756	-0.160449	0.300224	-0.357850	0.642494	-0.434569	0.40595
1	2.196740	0.248262	-0.035774	0.076675	0.481111	-0.912998	0.141005	0.68598
2	3.038249	0.347426	-0.898339	-0.748906	0.732996	-0.471165	-0.367575	0.81069
3	2.785988	0.925624	-1.371520	-1.970512	1.161507	-0.319854	-1.587461	0.26819
4	1.467617	-0.268154	0.204997	0.150798	-0.013160	0.936230	-0.412706	-1.00680
...	...	...	...	...	...	...	...	.
27597	-4.029500	-0.015749	0.257479	1.624659	-1.286374	-0.203822	-0.070549	0.97442
27598	-3.178049	-0.782360	1.012858	1.334436	0.811270	-0.084272	0.513110	0.02275
27599	-1.203835	-0.086197	0.075036	0.637881	-1.402472	0.725438	-0.394954	0.64700
27600	-3.359480	-0.903522	1.391974	1.486141	0.457582	0.913404	0.012700	0.11433
27601	-2.503012	0.559939	1.028883	1.613989	-1.509866	1.055898	-0.468245	-0.56970

27602 rows × 21 columns

In [235]:
```
import os
os.environ['LOKY_MAX_CPU_COUNT'] = '4'
```

Hierarchical Clustering, with k = 2

Also in this case, the choice of starting with k = 2 clusters in hierarchical clustering for binary classification tasks is primarily driven by the nature of binary classification itself. With the objective to categorize data into two distinct classes, setting k = 2 in aligns with this binary structure.

In [268]:
```
hier= AgglomerativeClustering(n_clusters= 2).fit(under_df_pca)
```

In [269]:
```
df #this is the original dataframe, with 'Amount' standardized and
```

Out[269]:

	V1	V2	V3	V4	V5	V6	V7	
0	-0.260648	-0.469648	2.496266	-0.083724	0.129681	0.732898	0.519014	-0.1300
1	0.985100	-0.356045	0.558056	-0.429654	0.277140	0.428605	0.406466	-0.1331
2	-0.260272	-0.949385	1.728538	-0.457986	0.074062	1.419481	0.743511	-0.0955
3	-0.152152	-0.508959	1.746840	-1.090178	0.249486	1.143312	0.518269	-0.0651
4	-0.206820	-0.165280	1.527053	-0.448293	0.106125	0.530549	0.658849	-0.2126
...	...	...	...	...	...	...	...	
568625	-0.833437	0.061886	-0.899794	0.904227	-1.002401	0.481454	-0.370393	0.1896
568626	-0.670459	-0.202896	-0.068129	-0.267328	-0.133660	0.237148	-0.016935	-0.1477
568627	-0.311997	-0.004095	0.137526	-0.035893	-0.042291	0.121098	-0.070958	-0.0199
568628	0.636871	-0.516970	-0.300889	-0.144480	0.131042	-0.294148	0.580568	-0.2077
568629	-0.795144	0.433236	-0.649140	0.374732	-0.244976	-0.603493	-0.347613	-0.3408

In [270]:
```
undersampled_df #this is the undersampled (5%) dataframe, with 'Amo
```

Out[270]:

	V1	V2	V3	V4	V5	V6	V7	V
0	0.280786	-0.121753	0.080266	-0.634878	0.337487	-0.339435	0.687471	-0.12225
1	1.676371	-1.217370	0.124901	-1.497943	-0.216193	0.282256	0.116151	-0.16945
2	-0.397542	-1.339106	1.782958	-1.988351	0.807473	-0.510503	0.277306	-0.16775
3	0.141210	-0.485878	2.073377	-0.208767	0.058450	1.215183	0.249732	-0.04961
4	0.714555	-0.243008	0.277992	-0.385915	0.440426	0.065985	0.539742	-0.15435
...	...	...	...	...	...	...	...	.
27597	-1.194215	0.273000	-0.978721	0.976918	-0.560189	-0.675675	-1.190438	-0.02010
27598	-0.206089	0.878329	-1.037097	1.519706	0.226155	-1.256981	-0.316752	0.22944
27599	-0.256469	0.184256	-0.096168	0.134291	-0.084373	-0.203676	-0.135060	0.08800
27600	-0.093502	0.934110	-1.048917	1.522697	0.197708	-1.167343	-0.362378	0.22440
27601	-0.804051	0.557047	-0.918128	1.448933	0.134470	-0.784706	-0.291492	-0.20082

Hierarchical Clustering Evaluation

The following cell is used to create a subset of the original 'Class' data that aligns with the undersampled dataset. This subset is then used to evaluate and compare the results of the clustering algorithm, ensuring that the true class labels correspond to the same data points in the undersampled dataset.

In [271]:
```
# Undersample the 'Class' column by keeping the same indexes

second_undersample = df.loc[undersampled_indices]['Class'] #these a

# second_undersample contains the values of the 'Class' column unde
```

In [272]:
```
# Calculate the scores
homogeneity = homogeneity_score(second_undersample, hier.labels_)
completeness = completeness_score(second_undersample, hier.labels_)
v_measure = v_measure_score(second_undersample, hier.labels_)

# Print the scores
print(f"Homogeneity score    = {homogeneity:.4f}")
print(f"Completeness score   = {completeness:.4f}")
print(f"V-Measure score      = {v_measure:.4f}")
```

```
Homogeneity score   = 0.6821
Completeness score  = 0.6897
V-Measure score     = 0.6859
```

In comparing hierarchical and k-means clustering for k=2, hierarchical clustering demonstrates superior performance. This suggests that hierarchical clustering more effectively groups data points into consistent and complete clusters for this specific dataset, indicating a better alignment with the inherent data structure than k-means for this credit card fraud detection task.

In [273]:
```
calinski = calinski_harabasz_score(under_df_pca, hier.labels_)
print(f"Calinski-Harabaz Index = {calinski:.4f}")
```

```
Calinski-Harabaz Index = 8935.3021
```

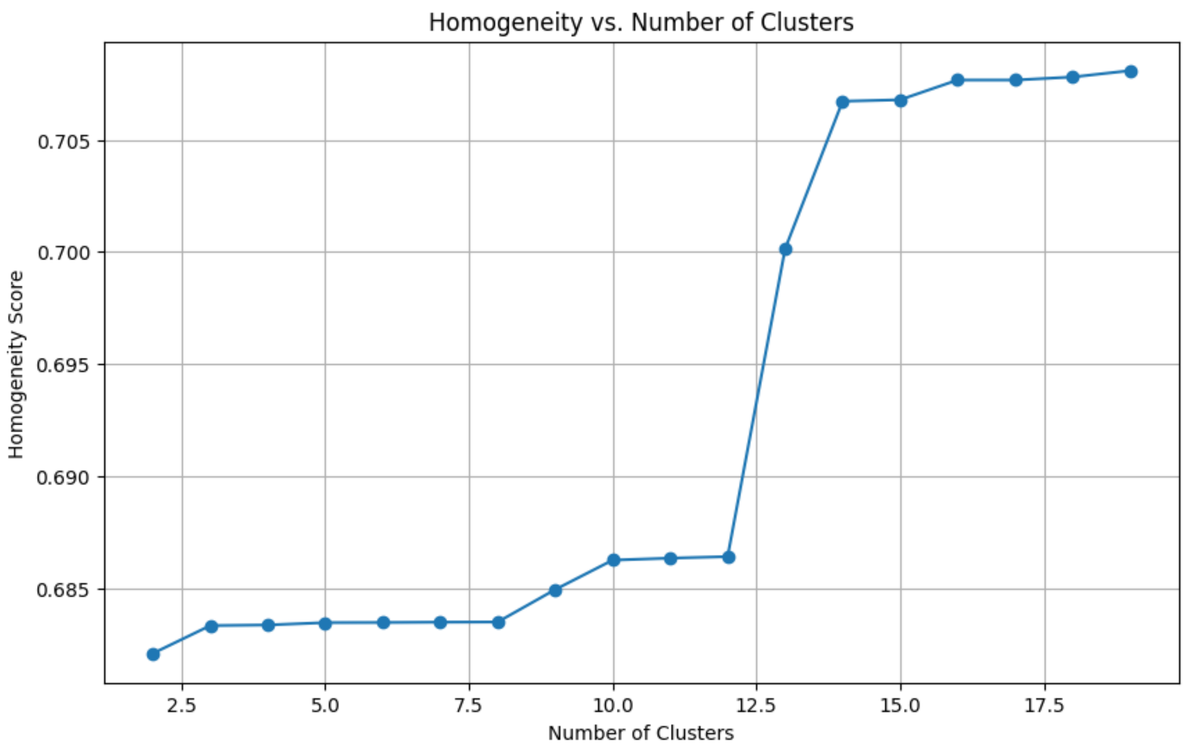
This value is considerably lower than the scores for the KMeans algorithm with two and sixteen clusters. A lower Calinski-Harabaz score typically indicates that the clusters are less dense and/or less well separated.

Determining the optimal number of clusters for the hierarchical clustering model

In [244]:
```
n_clusters = list(range(2, 20))
n_clusters

homogeneity_values = []

for k in n_clusters:
    # Register start time
    t_start = time.time()
    # Fit clustering algorithm
    hier= AgglomerativeClustering(n_clusters= k).fit(under_df_pca)
    # Compute score with current configuration and append it to the
    homogeneity_values.append(homogeneity_score(second_undersample,
    # Register end time
    t_stop = time.time()
    # Print elapsed time
    print(f"Elapsed time: {t_stop - t_start:.5f} seconds (k: {k:2d}
```

```
Elapsed time: 29.43377 seconds (k:  2)
Elapsed time: 27.96330 seconds (k:  3)
Elapsed time: 30.23306 seconds (k:  4)
Elapsed time: 27.87402 seconds (k:  5)
Elapsed time: 31.13049 seconds (k:  6)
Elapsed time: 27.78304 seconds (k:  7)
Elapsed time: 31.40133 seconds (k:  8)
Elapsed time: 26.49843 seconds (k:  9)
Elapsed time: 34.85649 seconds (k: 10)
Elapsed time: 35.55700 seconds (k: 11)
Elapsed time: 30.87342 seconds (k: 12)
Elapsed time: 32.34186 seconds (k: 13)
Elapsed time: 31.52025 seconds (k: 14)
Elapsed time: 31.84963 seconds (k: 15)
Elapsed time: 28.77700 seconds (k: 16)
Elapsed time: 28.15366 seconds (k: 17)
Elapsed time: 26.54312 seconds (k: 18)
Elapsed time: 28.23876 seconds (k: 19)
```

In [245]:
```python
# Create a plot
plt.figure(figsize=(10, 6))
plt.plot(n_clusters, homogeneity_values, marker='o', linestyle='-')
plt.title('Homogeneity vs. Number of Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('Homogeneity Score')
plt.grid(True)
plt.show()
```



The stability of the homogeneity score after the jump could imply that the optimal number of clusters, with respect to homogeneity, is found at the elbow of the curve. Let's see what happens when k = 14.

Hierarchical Clustering, with k = 14

In [257]:
```python
hier= AgglomerativeClustering(n_clusters= 14).fit(under_df_pca)
```

In [258]:
```
# Calculate the scores
homogeneity = homogeneity_score(second_undersample, hier.labels_)
completeness = completeness_score(second_undersample, hier.labels_)
v_measure = v_measure_score(second_undersample, hier.labels_)

# Print the scores
print(f"Homogeneity score   = {homogeneity:.4f}")
print(f"Completeness score  = {completeness:.4f}")
print(f"V-Measure score     = {v_measure:.4f}")
```

```
Homogeneity score   = 0.7067
Completeness score  = 0.2455
V-Measure score     = 0.3644
```

By increasing the number of clusters for hierarchical clustering, the homogeneity score increased (but not that much), which suggests that the clusters have become more pure —each cluster is likely to contain data points that are more similar to each other. However, the completeness score decreased significantly indicating that members of a single class are now spread across more clusters, rather than being grouped together. Consequently, the V-measure score, which is a balance between homogeneity and completeness, has also decreased. Essentially, as the number of clusters increased, the algorithm focused more on the internal similarity within clusters than on keeping all members of a class within a single cluster.

In [259]:
```
calinski = calinski_harabasz_score(under_df_pca, hier.labels_)
print(f"Calinski-Harabaz Index = {calinski:.4f}")
```

```
Calinski-Harabaz Index = 2621.9012
```

In comparison to higher scores obtained from other clustering methods or different numbers of clusters, this lower score may imply that having 14 clusters does not capture the data's structure as effectively. This can occur if the clusters are too many for the natural groupings in the data, causing some clusters to be very close to each other or some clusters to have high variance internally.

DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a popular clustering algorithm renowned for its effectiveness in various scenarios. It differs from other algorithms like k-means and hierarchical clustering by being density-based, identifying clusters based on the density of data points. The algorithm requires two parameters: ε (epsilon), a radius defining the neighbourhood around a point, and minPoints, the minimum number of points to form a cluster. Points are categorized as core, border, or noise points. Core points are central to clusters and have a sufficient number of neighbours within ε, border points are on the edges of clusters and are within ε of core points, and noise points are isolated data points that do not fit into any cluster. This density-based approach allows DBSCAN to discover clusters of arbitrary shapes and effectively handle noise in the data, making it a powerful clustering algorithm for various applications. The clustering process involves randomly selecting points, forming clusters around core points, and recursively including nearby core and border points. Despite its versatility, DBSCAN has limitations, such as difficulty handling clusters of varying densities, dependence on parameter selection, non-determinism, poor performance in high-dimensional spaces, and sensitivity to the data's density characteristics, especially when derived from sampled sources.

DBSCAN Clustering, with ϵ = 0.2

In [260]:
```
eps = 0.2 # The minPoints parameter has a default value of 5 in sci
dbscan = DBSCAN(eps=eps).fit(df_pca)
```

In [261]:
```
print(dbscan.labels_)
```
```
[-1 43 -1 ... -1 -1 -1]
```

In [262]:
```
# Calculate the scores
homogeneity = homogeneity_score(true_labels, dbscan.labels_)
completeness = completeness_score(true_labels, dbscan.labels_)
v_measure = v_measure_score(true_labels, dbscan.labels_)

# Print the scores
print(f"Homogeneity score   = {homogeneity:.4f}")
print(f"Completeness score  = {completeness:.4f}")
print(f"V-Measure score     = {v_measure:.4f}")
```
```
Homogeneity score   = 0.3066
Completeness score  = 0.0909
V-Measure score     = 0.1402
```

The metrics obtained from the DBSCAN clustering are not that robust, indicating that the model's current configuration may not be optimal. This could be attributed to the value assigned to the ϵ parameter, which plays a critical role in defining the scale of the clusters and directly impacts the model's ability to discern distinct groups.

A small ϵ value can lead to a higher number of clusters with many points classified as noise, whereas a larger epsilon can merge distinct clusters into a single cluster. Both cases can lead to poor performance metrics.

To potentially improve the model's performance, let's experiment with different values of ϵ.

In [263]:
```
calinski = calinski_harabasz_score(df_pca, dbscan.labels_)
print(f"Calinski-Harabaz Index = {calinski:.4f}")
```

```
Calinski-Harabaz Index = 40.4303
```

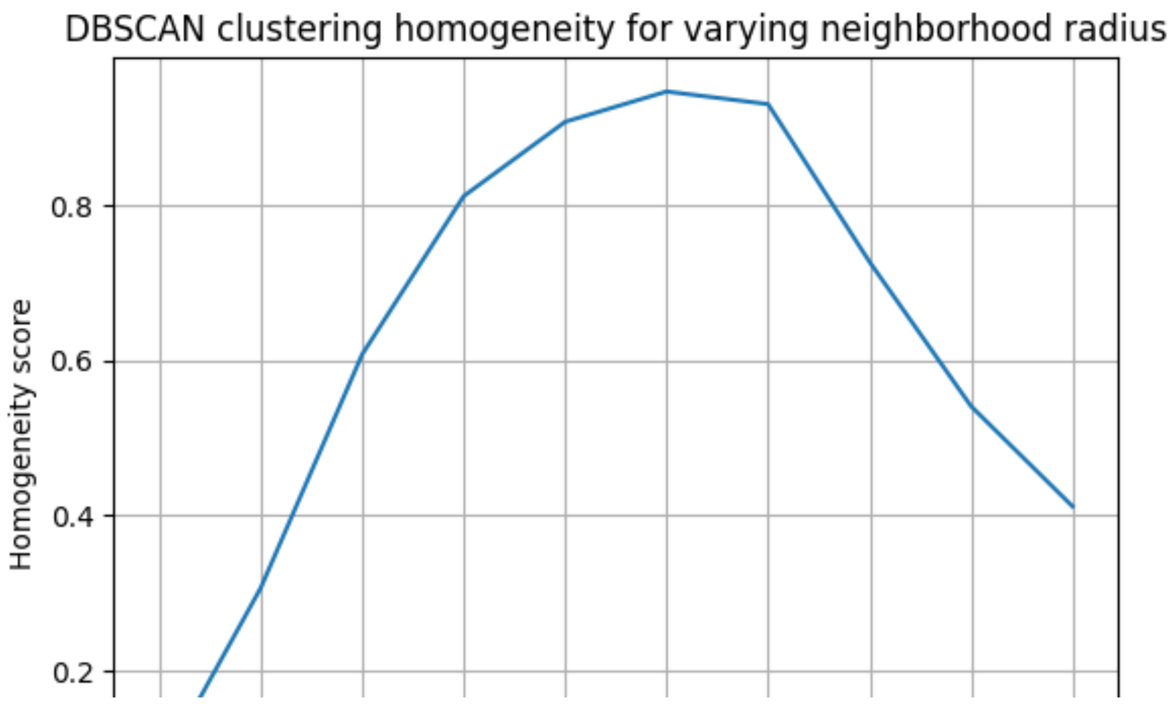
This very low Calinski-Harabaz Index suggests that the clusters identified by DBSCAN are not well-separated and likely do not exhibit high internal cohesion. This could be attributable to an improper ϵ value.

Determining the optimal ϵ for the DBSCAN clustering model

In [87]:
```python
eps = [(e + 1) / 10 for e in range(10)]

homogeneity_values = []

for e in eps:
    # Register start time
    t_start = time.time()
    # Fit clustering algorithm
    dbscan = DBSCAN(eps=e).fit(df_pca)
    # Compute score with current configuration and append it to the
    homogeneity_values.append(homogeneity_score(true_labels, dbscan
    # Register end time
    t_stop = time.time()
    # Print elapsed time
    print(f"Elapsed time: {t_stop - t_start:.5f} seconds (e: {e:.2f
```

```
Elapsed time: 129.68895 seconds (e: 0.10)
Elapsed time: 141.90465 seconds (e: 0.20)
Elapsed time: 147.88165 seconds (e: 0.30)
Elapsed time: 148.52020 seconds (e: 0.40)
Elapsed time: 160.53390 seconds (e: 0.50)
Elapsed time: 160.62813 seconds (e: 0.60)
Elapsed time: 163.23525 seconds (e: 0.70)
Elapsed time: 153.75837 seconds (e: 0.80)
Elapsed time: 155.20501 seconds (e: 0.90)
Elapsed time: 154.54170 seconds (e: 1.00)
```

In [88]:
```
plt.figure()

plt.plot(eps, homogeneity_values)
plt.xticks(eps)

plt.title('DBSCAN clustering homogeneity for varying neighborhood r
plt.xlabel('Radius ($\epsilon$)')
plt.ylabel('Homogeneity score')
plt.grid()
plt.show()
```



The peak of the curve represents the best balance between having too narrow a radius, where many points may be considered noise, and too wide a radius, which can lead to merging distinct clusters into a single cluster.

DBSCAN Clustering, with ϵ = 0.6

In [264]:
```
eps = 0.6 # The minPoints parameter has a default value of 5 in sci
dbscan = DBSCAN(eps=eps).fit(df_pca)
```

In [265]:
```
# Calculate the scores
homogeneity = homogeneity_score(true_labels, dbscan.labels_)
completeness = completeness_score(true_labels, dbscan.labels_)
v_measure = v_measure_score(true_labels, dbscan.labels_)

# Print the scores
print(f"Homogeneity score   = {homogeneity:.4f}")
print(f"Completeness score  = {completeness:.4f}")
print(f"V-Measure score     = {v_measure:.4f}")
```

```
Homogeneity score   = 0.9459
Completeness score  = 0.3195
V-Measure score     = 0.4776
```

What this means is that the model with eps=0.6 has managed to create very pure clusters, but those clusters do not capture all instances of the classes well (many classes are spread out over multiple clusters). On the other hand, the model with eps=0.2 performs poorly both in terms of cluster purity and grouping class members together.

What is the Best Clustering Model?

The performance of three clustering algorithms, namely K-Means, Hierarchical clustering, and DBSCAN, was evaluated on a credit card fraud detection dataset. The purpose of this evaluation was to determine which clustering model is better suited for binary classification problems and to discuss why clustering might not be the optimal choice for such problems. We will exclusively assess the performance using the metrics of homogeneity, completeness, and V-measure since we have access to the true labels. It's worth noting that in this comparison, k = 2 clusters were applied to align with the binary classification nature of the problem.

K-Means Clustering:

- Homogeneity Score: 0.5650
- Completeness Score: 0.5881
- V-Measure Score: 0.5763

Hierarchical Clustering (with a 5% undersampled dataset due to computational constraints):

- Homogeneity Score: 0.6821
- Completeness Score: 0.6897
- V-Measure Score: 0.6859

DBSCAN (with epsilon = 0.6, determined as the optimal epsilon):

- Homogeneity Score: 0.9459
- Completeness Score: 0.3195
- V-Measure Score: 0.4776

Model Evaluation

Among the three models:

- **DBSCAN** achieves the highest homogeneity score, indicating that it creates clusters that are more internally consistent in terms of class labels. However, it has the lowest completeness score, suggesting that it may not capture all instances of the positive class (fraud).
- **Hierarchical clustering** performs well in terms of both homogeneity and completeness, striking a balance between cluster quality and coverage of the positive class.
- **K-Means**, while providing a reasonable compromise between homogeneity and completeness, falls somewhat in between DBSCAN and Hierarchical clustering.

Choice of Model for Binary Classification

For binary classification problems like credit card fraud detection, the primary goal is to accurately identify instances of the positive class (fraud). Therefore, completeness (the ability to capture all positive instances) is a crucial metric.

In this context, **Hierarchical clustering** appears to be the most suitable choice as it achieves a reasonable trade-off between homogeneity and completeness, ensuring a good balance between clustering quality and positive instance coverage.

Why Clustering May Not Be Optimal for Binary Classification

Clustering algorithms are designed for unsupervised learning tasks where the true labels are not available. In the case of binary classification in which the class label is available, supervised learning algorithms like Neural Networks, Decision Trees, or Random Forests are generally preferred. Clustering doesn't use this class information and instead groups data points based solely on their similarities, which may not align with the binary classification task.

Clusters group data points based on overall similarity, but it doesn't necessarily distinguish between subtle differences that are essential in binary classification. Classification models are designed to learn decision boundaries that maximize the separation between classes. Moreover, evaluating the performance of clustering algorithms on binary classification tasks can be challenging. Metrics like homogeneity and completeness, commonly used for clustering evaluation, may not directly translate to the effectiveness of binary classifiers. Binary classifiers are typically evaluated using precision, recall, F1-score, ROC curves, etc.

Finally, clustering models, depending on the algorithm, may have varying levels of complexity and scalability issues. They do not scale optimally to the typical high dimensions of large-scale binary classification tasks. In such cases, simpler and more efficient models should be preferred.

Optional Part: Extra (Imbalanced) Dataset

For the optional part, an alternative credit card fraud detection dataset was chosen. The primary distinction from the previous dataset lies in the highly unbalanced distribution of class labels. This dataset contains transactions that occurred in two days, comprising 492 fraud cases out of a total of 284,807 transactions. It's important to note that this dataset exhibits a significant class imbalance, with the positive class (fraudulent transactions) representing a mere 0.172% of the entire transaction volume.

Data Preparation

In this section, data preparation steps will be performed.

Data Loading

The provided data is in a *Comma Separated Values* (CSV) file. **Fortunately**, Pandas offers built-in fucntions to load these file automatically.

```
In [420]: df = pd.read_csv(r"/Users/federicastefanizzi/Downloads/creditcard.c
```

```
In [421]: df.sample(10)
```

Out[421]:

	Time	V1	V2	V3	V4	V5	V6	V
268499	163241.0	-4.354356	-2.306397	-1.648038	-1.051172	-2.425199	-0.890446	-0.5660
62173	50163.0	0.884847	-0.585087	0.671848	1.363226	-0.802593	0.275890	-0.3537
12738	22327.0	1.247089	-0.351657	0.828671	-0.011727	-0.577409	0.560595	-0.9330
155403	105332.0	-3.586759	3.470251	-0.577418	-1.324697	-0.264627	-0.012731	-0.4919
92903	64162.0	1.214417	-0.006048	0.102805	0.273894	0.050689	0.267520	-0.2189
206388	136191.0	1.808058	-0.521598	-1.551198	0.227859	0.065169	-0.593926	0.1902
233725	147681.0	-1.286111	0.002821	0.349455	-1.489309	0.135679	-0.631661	0.4632
258149	158516.0	2.050734	-0.364010	-2.542843	-0.729357	2.388455	3.318015	-0.4791
165114	117209.0	-0.986614	1.824985	-0.916355	-0.402232	0.069265	-0.525213	-0.5401
194626	130684.0	-0.350942	0.463124	1.647337	-0.558471	-0.230180	-0.445314	0.2911

In the provided sample of 10 rows from the dataset, it's noteworthy that there are 31 columns in total. Out of these, 28 columns have been anonymized to safeguard the privacy of individuals involved in credit card transactions. The three known features are the Time, the transaction amount, and the class label denoted as "Class," distinguishing between **fraudulent (1)** and **non-fraudulent (0)** transactions.

Summary Statistics

This section will provide a concise overview of key numerical characteristics within the dataframe.

In [422]:
```
len(df)
```

Out[422]:
```
284807
```

In [423]:
```
len(df.columns)
```

Out[423]:
```
31
```

In [424]:
```
df.columns
```

Out[424]:
```
Index(['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V
9', 'V10',
       'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V1
9', 'V20',
       'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Am
ount',
       'Class'],
      dtype='object')
```

In [425]:
```
df.dtypes
```

Out[425]:
```
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
```

This output provides the data types for each column in the dataframe df:

Column 'Class' has data type **int64**. Columns 'Time', 'V1' through 'V28' and 'Amount' have data type **float64**. The float64 data type indicates that these columns store numerical values with decimal points, while int64 indicates integer values. This summary is valuable for understanding the nature of the data types within the dataframe and is essential for subsequent data analysis and processing.

Descriptive Statistics for the Dataframe Columns

In [426]:
```
for column in df.columns:
    desc_df = df[column].describe()
    print(f"description for column '{column}': \n\n{desc_df}\n")
```

```
description for column 'Time':

count    284807.000000
mean      94813.859575
std       47488.145955
min           0.000000
25%       54201.500000
50%       84692.000000
75%      139320.500000
max      172792.000000
Name: Time, dtype: float64

description for column 'V1':

count    2.848070e+05
mean     1.168375e-15
std      1.958696e+00
min     -5.640751e+01
25%     -9.203734e-01
```

Time: This column represents the time elapsed in seconds between transactions. It ranges from 0 to 172,792 seconds. The mean time between transactions is approximately 94,813.86 seconds.

V1 - V28: These columns exhibit characteristics indicative of standardization. Their means all approximate to 0 and standard deviations are all near 1. The varying ranges from minimum to maximum values across columns could imply diverse scales or units, but nothing can be stated certainly due to their anonymized nature. Quartiles offer insights into the distribution of values, aiding in understanding the spread and central tendencies of these normalized features.

Amount: The 'Amount' column represents the monetary value associated with transactions. Descriptive statistics such as mean, median, and quartiles provide a glimpse into the central tendency and dispersion of transaction amounts. This information is crucial for understanding the distribution of monetary values within the dataset.

Class: The 'Class' column is a binary categorical variable (0 or 1), typically used for classification tasks, such as distinguishing between fraudulent and non-fraudulent transactions. The mean of 0.5 indicates a balanced distribution of classes, *suggesting an equal representation of both fraud and non-fraud instances in the dataset*. This balance is pivotal for training robust machine learning models.

Data Visualization

Dealing with Duplicates

Assumption Alert: As part of the data preprocessing steps, it was observed that certain rows in the anonymized features exhibit identical values across different columns. To address this issue and maintain data integrity, these identical rows will be eliminated. This decision is based on the assumption that rows displaying identical values across all 'V' columns, regardless of any differences in the 'Amount' and 'Time' columns, should be excluded from the analysis.

In [427]:
```
for column in df.columns:
    val_c = df[column].value_counts()
    print(f"value counts for column '{column}': \n\n{val_c}\n")
```

```
value counts for column 'Time':

Time
163152.0    36
64947.0     26
68780.0     25
3767.0      21
3770.0      20
            ..
127750.0     1
62260.0      1
62259.0      1
127753.0     1
172792.0     1
Name: count, Length: 124592, dtype: int64

value counts for column 'V1':

V1
-2.055797    77
```

In [428]:
```
columns_to_exclude = ['Amount','Time']  # Exclude
v_columns = [col for col in df.columns if col.startswith('V') and c

# Drop duplicates based on 'V' columns
df_no_duplicates = df.drop_duplicates(subset=v_columns, keep='first
```

In [429]:
```
original_rows, original_cols = df.shape #assigning the number of ro

# Calculate the number of eliminated rows
eliminated_rows = original_rows - df_no_duplicates.shape[0] #addres

# Print the number of eliminated rows
print(f'Number of eliminated rows: {eliminated_rows}')
```

```
Number of eliminated rows: 9144
```

New size divided by classes:

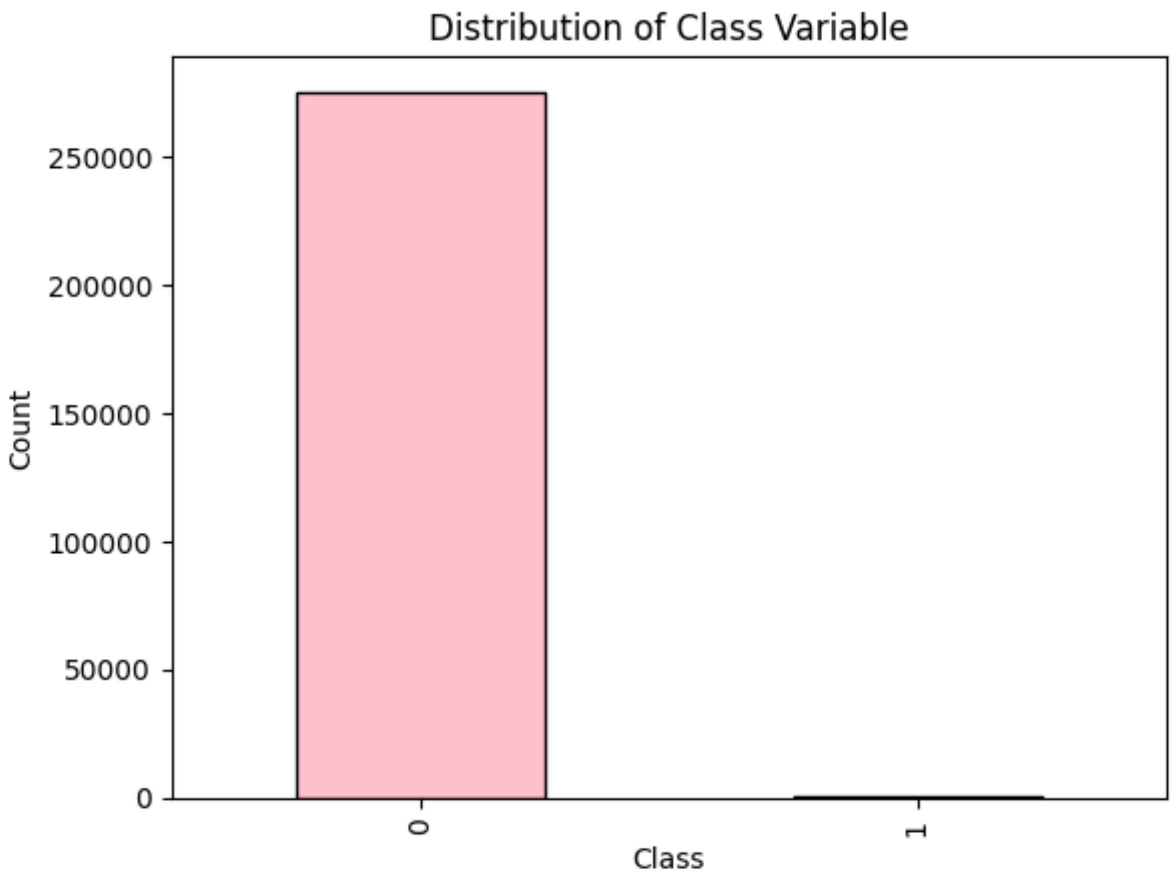
In [430]:
```
df.groupby('Class').size()
```

Out[430]:
```
Class
0    284315
1       492
dtype: int64
```

In [431]:
```
df = df_no_duplicates
```

A Glimpse Into the Target Variable 'Class'

In [432]:
```
# Bar plot for the 'Class' variable
df['Class'].value_counts().plot(kind='bar', color=['pink', 'skyblue
plt.title('Distribution of Class Variable')
plt.xlabel('Class')
plt.ylabel('Count')
plt.show()
```



As evident from the dataset, it displays a significant class imbalance, with a mere 0.172% of instances where "Class" equals 1, comprising 492 out of a total of 284,807 observations.

Creation of a New (More Balanced) Dataset

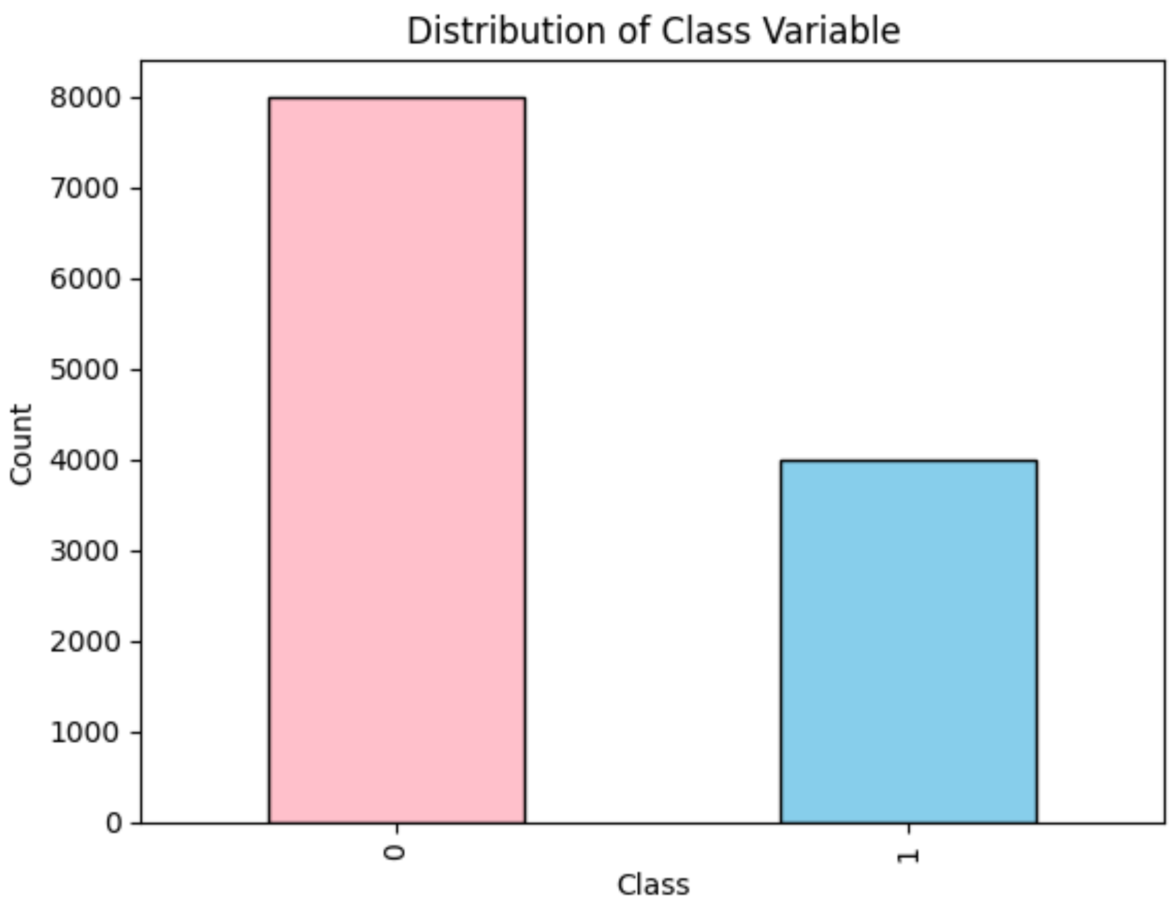
To address the significant class imbalance within the dataset, a two-fold strategy was implemented. First, an undersampling technique was applied to the non-fraudulent transactions, reducing the dataset to 8,000 instances. Subsequently, an oversampling approach was employed for the fraudulent transactions, increasing their representation to 4,000 instances. This rebalancing effort resulted in a new dataset distinct from the initial analysis, characterized by a revised distribution comprising two-thirds non-fraudulent instances to one-third fraudulent instances.

It is worth noting that this newly achieved balance may potentially introduce overfitting challenges, which will be addressed in the subsequent steps of the analysis.

In [433]:
```
# Desired number of instances for 'Class' = 1 and 'Class' = 0
desired_class_1_count = 4000
desired_class_0_count = 8000

# Sampling 4000 instances of 'Class' = 1 and 8000 instances of 'Cla
df_class_1_resampled = df[df['Class'] == 1].sample(n=desired_class_
df_class_0_resampled = df[df['Class'] == 0].sample(n=desired_class_

# Creating a balanced dataset by concatenating the sampled instance
df_resampled = pd.concat([df_class_1_resampled, df_class_0_resample

# Shuffle the concatenated DataFrame
df_resampled = df_resampled.sample(frac=1, random_state=42).reset_i

# Checking the counts
class_counts = df_resampled['Class'].value_counts()
print(class_counts)

df = df_resampled
```

```
Class
0    8000
1    4000
Name: count, dtype: int64
```

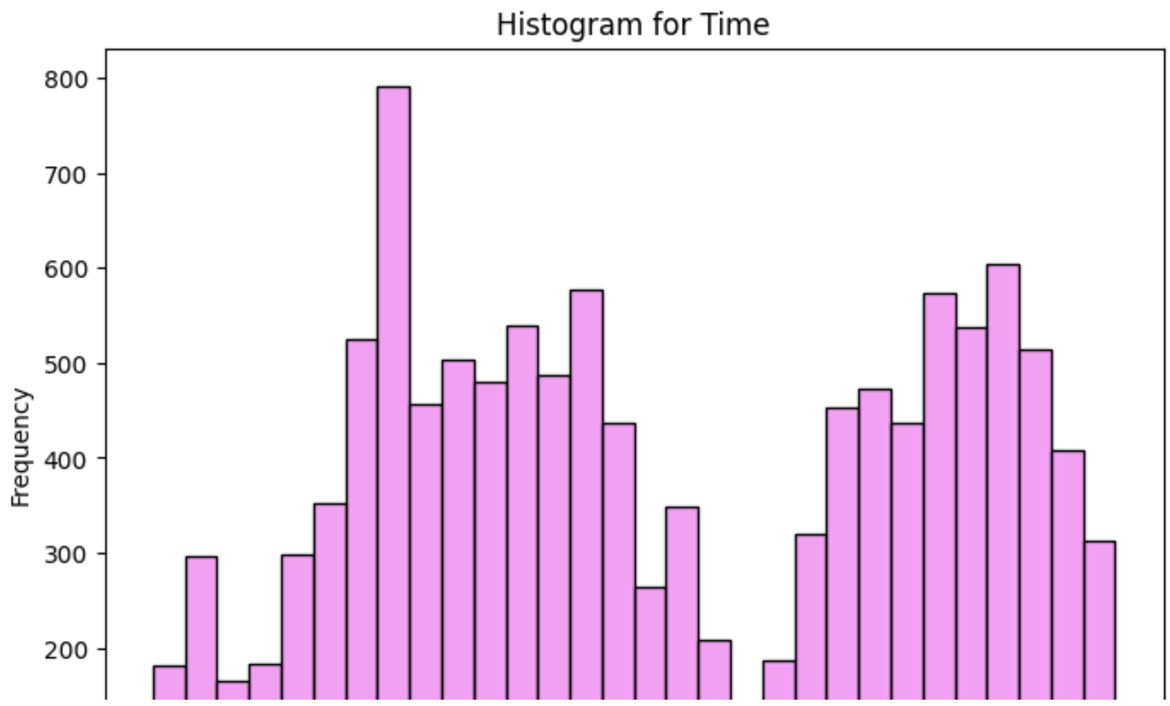
In [434]:
```
# Bar plot for the 'Class' variable
df['Class'].value_counts().plot(kind='bar', color=['pink', 'skyblue
plt.title('Distribution of Class Variable')
plt.xlabel('Class')
plt.ylabel('Count')
plt.show()
```



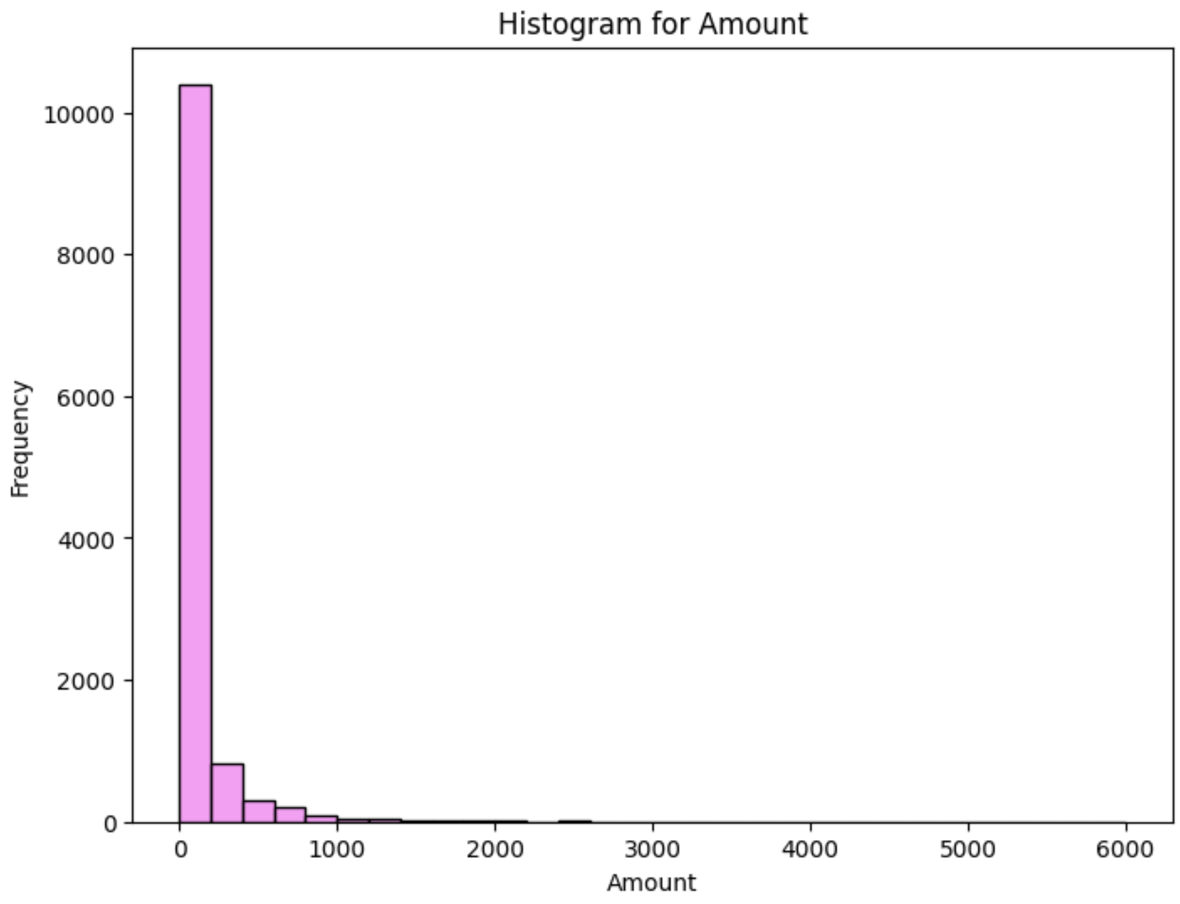
In an effort to address the substantial class imbalance within the credit card fraud detection dataset, a resampling approach was employed. By undersampling the majority class (non-fraudulent transactions) and oversampling the minority class (fraudulent transactions), a more balanced dataset was created. This undertaking offers several advantages, including the potential for improved model performance and generalization. Balancing the dataset can help mitigate bias, and facilitate better model training. However, there are noteworthy trade-offs to consider. Resampling may result in the loss of valuable information, potentially lead to overfitting on the minority class, and incur increased computational costs. Furthermore, there is a risk of data leakage and potential disruption of temporal information if present in the dataset.

Histograms for Feature Visualization

In [435]:
```
fig = plt.figure(figsize=(8, 6))
sns.histplot(data=df, x='Time', bins=30, color='violet')
plt.xlabel('Time')
plt.ylabel('Frequency')
plt.title('Histogram for Time')
plt.show()
```



In [436]:
```
fig = plt.figure(figsize=(8, 6))
sns.histplot(data=df, x='Amount', bins=30, color='violet')
plt.xlabel('Amount')
plt.ylabel('Frequency')
plt.title('Histogram for Amount')
plt.show()
```



The distribution is heavily skewed to the right, meaning most of the data points are on the lower end of the 'Amount' scale. There is a very high frequency of values in the first bin (0-1000) and the frequency decreases sharply as the amount increases.

In [437]:
```python
transactions_with_zero_amount = df[df['Amount'] == 0]
transactions_with_zero_amount
```

Out[437]:

	Time	V1	V2	V3	V4	V5	V6	V7
11	102480.0	-1.929597	4.066413	-4.865184	5.898602	-0.552493	-1.555962	-3.833623
135	90434.0	1.848433	0.373364	0.269272	3.866438	0.088062	0.970447	-0.721945
157	85285.0	-7.030308	3.421991	-9.525072	5.270891	-4.024630	-2.865682	-6.989195
175	30707.0	-0.964364	0.176372	2.464128	2.672539	0.145676	-0.152913	-0.591983
190	85285.0	-7.030308	3.421991	-9.525072	5.270891	-4.024630	-2.865682	-6.989195
...	...	...	...	...	...	...	...	...
11733	64443.0	1.079524	0.872988	-0.303850	2.755369	0.301688	-0.350284	-0.042848
11801	125658.0	0.224414	2.994499	-3.432458	3.986519	3.760233	0.165640	1.099378
11802	102671.0	-4.991758	5.213340	-9.111326	8.431986	-3.435516	-1.827565	-7.114303
11813	102669.0	-5.603690	5.222193	-7.516830	8.117724	-2.756858	-1.574565	-6.330343
11984	142961.0	0.457845	1.373769	-0.488926	2.805351	1.777386	0.100492	1.295016

244 rows × 31 columns

Within the dataset, a total of 244 instances exhibit a 0 value in the 'Amount' column. These instances shall be omitted from the dataset due to their limited relevance to the analytical objectives and their potential to introduce noise, which could adversely impact the accuracy of subsequent analyses and models. This removal is aimed at maintaining the dataset's integrity.

In [438]:
```python
df = df.drop(transactions_with_zero_amount.index)
```

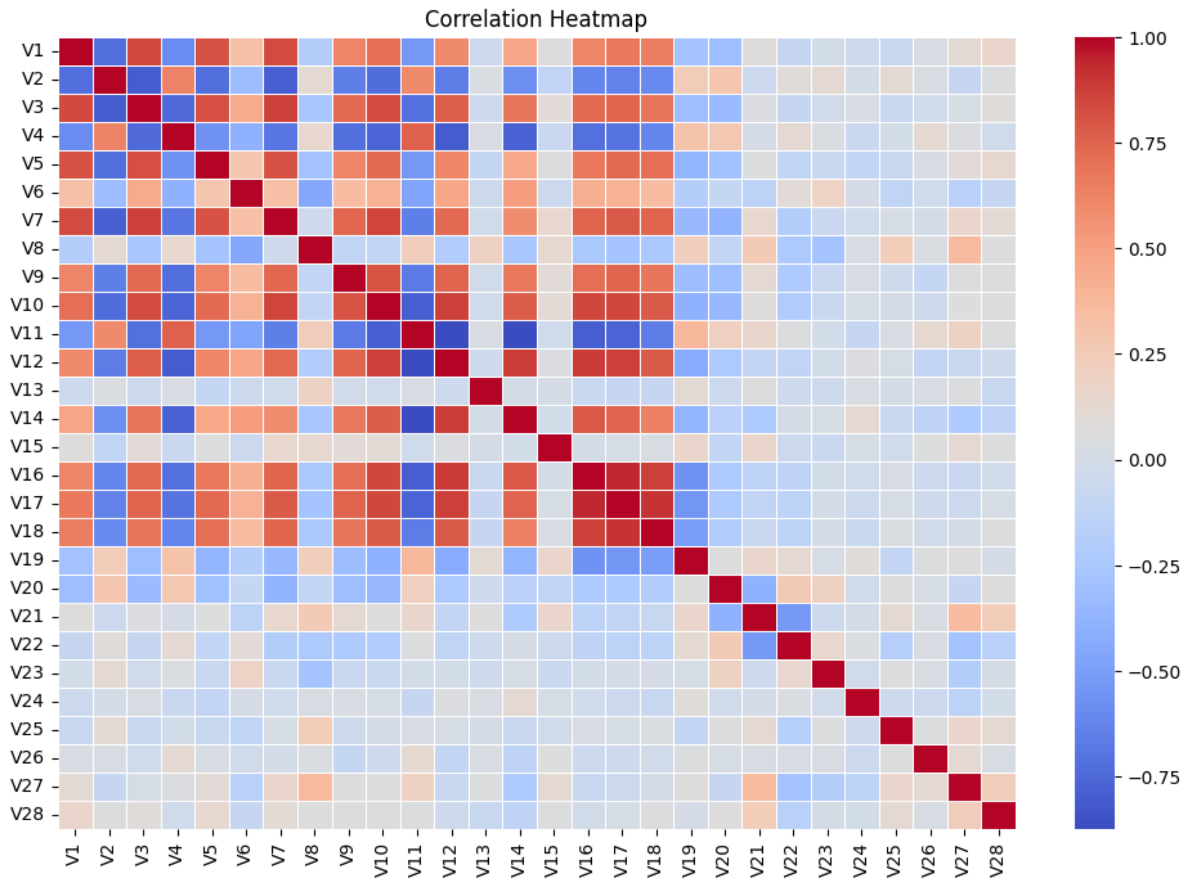
Dealing with Correlated Features

Assumption Alert: An assumption is made that 'strong correlation' is identified when the correlation coefficient surpasses a threshold of 0.8, encompassing both positive and negative correlations. Consequently, all the features that will show a positive correlation > 0.8 and a negative correlation < -0.8 will be removed.

All steps for the removal of columns displaying high correlation with other features will be executed in a manner consistent with the procedures employed in the previously analyzed dataset.

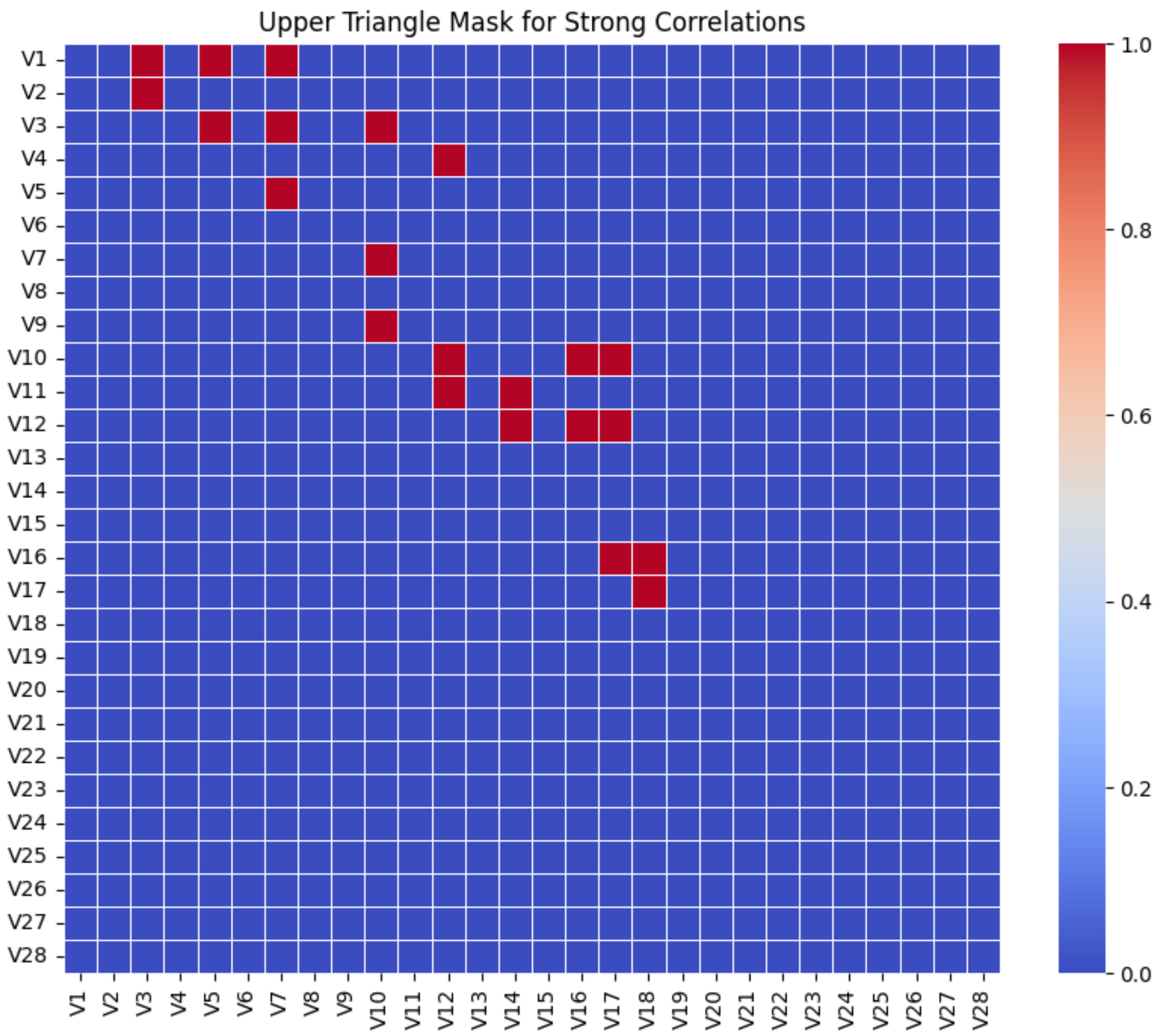
Calculate the Correlation Matrix

In [439]:
```
correlation_matrix = df[['V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7',
                         'V11', 'V12', 'V13', 'V14', 'V15', 'V16',
                         'V21', 'V22', 'V23', 'V24', 'V25', 'V26',

plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, cmap='coolwarm', annot=False, fmt="
plt.title('Correlation Heatmap')
plt.show()
```



In [440]:
```
upper_triangle_mask = np.triu((correlation_matrix > 0.8) | (correla
```

In [441]:
```
plt.figure(figsize=(10, 8))
sns.heatmap(upper_triangle_mask, cmap="coolwarm", annot=False, fmt=
plt.title("Upper Triangle Mask for Strong Correlations")
plt.show()
```



In [442]:
```
upper_triangle_df = pd.DataFrame(upper_triangle_mask, index=correla
```

In [443]:
```
columns_to_exclude = ['Class', 'Amount', 'Time']
to_drop = [column for column in upper_triangle_df.columns if any(up
df_nc = df.drop(to_drop, axis=1) # Here nc stands for 'no correlati
```

In [444]:
```
table = tabulate({"Columns Identified for Removal": to_drop}, heade
print(table)
```

```
Columns Identified for Removal
--------------------------------
V3
V5
V7
V10
V12
V14
V16
V17
V18
```

The following columns were identified for removal due to significant correlation issues.

In [445]:
```
df = df_nc
```

Dealing with Outliers

The anonymization of features makes it challenging to remove outliers because the absence of information about their variable of reference prevents the evaluation of whether an outlier is significant or not.

Supervised Learning Model

Train-test split

In [446]:
```
X_train, X_test, y_train, y_test = train_test_split(df.drop('Class'
```

In [447]:
```
len(X_test)
```

Out[447]:
```
2939
```

Scaling

In [448]:
```
X_train_amount = X_train[['Amount']]
X_test_amount = X_test[['Amount']]

standard_scaler = StandardScaler()

X_train_amount_stand = standard_scaler.fit_transform(X_train_amount

X_test_amount_stand = standard_scaler.transform(X_test_amount.value

#replace the original 'Amount' column with the scaled values
X_train['Amount'] = X_train_amount_stand
X_test['Amount'] = X_test_amount_stand
```

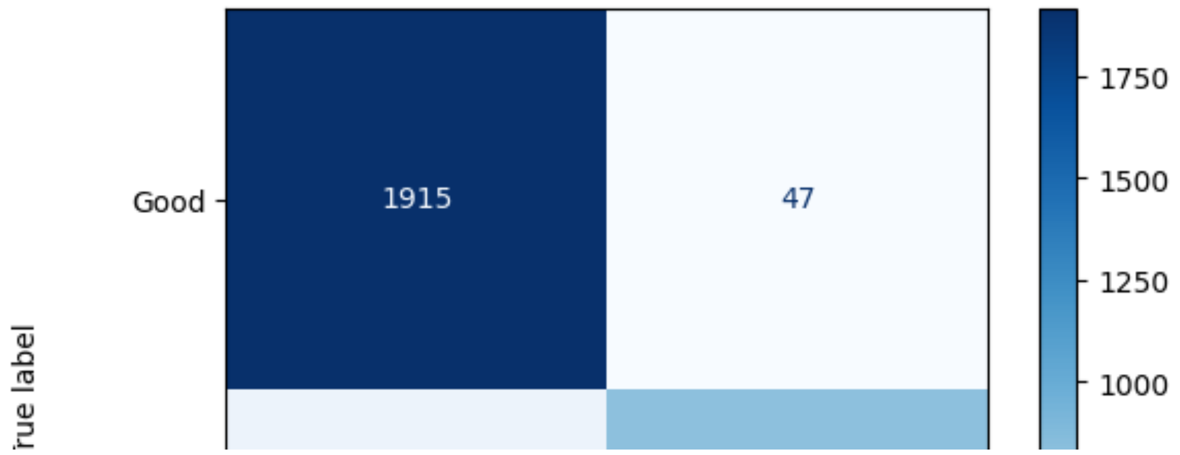
In [449]:
```
X_train_amount = X_train[['Time']]
X_test_amount = X_test[['Time']]

standard_scaler = StandardScaler()

X_train_amount_stand = standard_scaler.fit_transform(X_train_amount

X_test_amount_stand = standard_scaler.transform(X_test_amount.value

#replace the original 'Time' column with the scaled values
X_train['Time'] = X_train_amount_stand
X_test['Time'] = X_test_amount_stand
```

Notice: Standardization has specifically been implemented on the 'Amount' and 'Time' columns since the 'V' columns were already standardized.

Linear Support Vector Machines (SVM)

```
In [450]: #LinearSVM

start_time = time.time()

clf =  LinearSVC(dual=False, random_state=random_seed).fit(X_train,

y_pred = clf.predict(X_test)

end_time = time.time()
duration = end_time - start_time

print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
print(f"F1:          {f1_score(y_test, y_pred):.5f}")
print(f"Model Runtime (seconds): {duration:.2f} seconds")

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
plt.show()
```

```
Accuracy:    0.93229
Precision:   0.94610
Recall:      0.84442
Specificity: 0.92646
F1:          0.89237
Model Runtime (seconds): 0.04 seconds
```



The results indicate that the linear SVM classifier has performed well in distinguishing between 'Good' and 'Fraudulent' classes. The accuracy is high at 93.22%, meaning that the majority of predictions are correct. Precision is also high at 94.61%, indicating that when the model predicts the 'Good' class, it is usually correct. Recall is slightly lower at 84.44%, suggesting that the model misses some 'Good' instances, classifying them as 'Fraudulent'. The F1 score, which balances precision and recall, is good at 89.23%. The model is fast, with a runtime of only 0.04 seconds, making it efficient for practical use. The specificity, which indicates how well the model identifies 'Fraudulent' cases, is 92.64%, showing that it is also reliable in spotting fraudulent instances. Overall, the model demonstrates robust performance metrics.

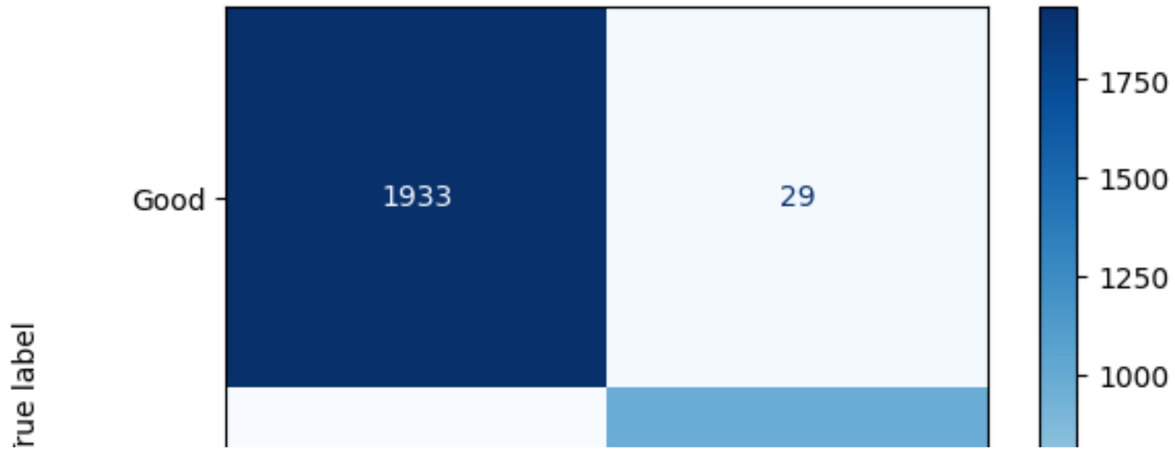
Non-Linear Classifiers

The three best-performing models identified in the previous dataset analysis have been implemented. These models comprise a Neural Network specialized for Precision-Recall, a Decision Forest optimized for Calibration, and a Random Forest model tailored for ROC Curve optimization.

Neural Networks

In [453]:
```
start_time = time.time()

clf = MLPClassifier(max_iter=300, random_state=random_seed)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

end_time = time.time()
duration = end_time - start_time

print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
print(f"F1:          {f1_score(y_test, y_pred):.5f}")
print(f"Model Runtime (seconds): {duration:.2f} seconds")

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
plt.show()
```

```
Accuracy:    0.98979
Precision:   0.97114
Recall:      0.99898
Specificity: 0.99948
F1:          0.98486
Model Runtime (seconds): 4.70 seconds
```



The accuracy is extremely high at 98.97%, meaning almost all predictions made by the model are correct. Precision is very good at 97.11%, indicating that when the model predicts an instance as 'Good', it is correct most of the time. Recall is almost perfect at 99.89%, showing that the model has identified all actual 'Good' instances with only one error. Specificity is also almost perfect at 99.94%, meaning almost every 'Fraudulent' instance was correctly identified. The F1 score is outstanding at 98.48%, suggesting an excellent balance between precision and recall. The model's runtime is 4.70 seconds, which, while longer than the previous model, is still quite efficient given the complexity of such task. Overall, the model is performing exceptionally well on the given task (probably also because of overfitting issues).

Decision Trees

In [454]:
```
#Decision Tree

start_time = time.time()

clf = DecisionTreeClassifier(random_state=random_seed)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

end_time = time.time()
duration = end_time - start_time

print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
print(f"F1:          {f1_score(y_test, y_pred):.5f}")
print(f"Model Runtime (seconds): {duration:.2f} seconds")

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
plt.show()
```

```
Accuracy:    0.99217
Precision:   0.97700
Recall:      1.00000
Specificity: 1.00000
F1:          0.98837
Model Runtime (seconds): 0.16 seconds
```

The accuracy is very high at 99.21%, which means the model correctly predicted the majority of the instances. The precision is 97.70%, indicating that when the model predicts an instance as 'Good', it is correct most of the time. The recall is perfect at 100%, which means the model correctly identified all of the true 'Good' instances. Specificity is also perfect at 100%, demonstrating that all 'Fraudulent' instances were identified correctly. The F1 score is excellent at 98.83%, which shows a strong balance between precision and recall, indicating that the model is both accurate and consistent. The model's runtime is very quick at 0.16 seconds, indicating that it is highly efficient. In summary, the Decision Tree classifier has achieved near-perfect performance on this task, with high scores across all metrics and a very quick runtime, suggesting it is an effective and efficient model for the classification task it was designed to perform.

Ensemble Learning (1): Random Forest

In [455]:
```
#Random Forest

start_time = time.time()

clf = RandomForestClassifier(random_state=random_seed, n_estimators
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

end_time = time.time()
duration = end_time - start_time

print(f"Accuracy:    {accuracy_score(y_test, y_pred):.5f}")
print(f"Precision:   {precision_score(y_test, y_pred):.5f}")
print(f"Recall:      {recall_score(y_test, y_pred):.5f}")
print(f"Specificity: {recall_score(y_pred, y_test, pos_label=0):.5f
print(f"F1:          {f1_score(y_test, y_pred):.5f}")
print(f"Model Runtime (seconds): {duration:.2f} seconds")

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_lab
plt.show()
```

```
Accuracy:    0.99796
Precision:   0.99390
Recall:      1.00000
Specificity: 1.00000
F1:          0.99694
Model Runtime (seconds): 7.42 seconds
```

Accuracy: 99.79%, indicating almost all predictions are correct. Precision: 99.39%, showing that when the model predicts an instance as 'Good', it is accurate almost every time. Recall: 100%, meaning the model correctly identified every 'Good' instance, with no 'Good' instances incorrectly marked as 'Fraudulent'. Specificity: 100%, indicating that every 'Fraudulent' instance was correctly identified, with no 'Fraudulent' instances incorrectly marked as 'Good'. F1 Score: 99.69%, which is near perfect, showing an excellent balance between precision and recall. Model Runtime: At 7.42 seconds, the runtime is reasonable for a Random Forest model, which typically requires more computational resources due to its ensemble nature. These results suggest that the Random Forest model is highly effective for this classification task, with near-perfect performance across all metrics.

Comparison Between Models

Precision-Recall, ROC Curve analysis, and calibration assessments will once again be executed on the models that have demonstrated the best performance.

In [456]:
```
#Dictionary

logging.basicConfig(level=logging.INFO)

clf_dict = {}

models = {
    'Linear SVM': LinearSVC(dual=False, random_state=random_seed),
    'Neural Networks': MLPClassifier(hidden_layer_sizes=(50, 50), m
    'Decision Tree': DecisionTreeClassifier(random_state=random_see
    'Random Forest': RandomForestClassifier(random_state=random_see
}

for model_name, model in models.items():
    start_time = time.time()
    model.fit(X_train, y_train)
    end_time = time.time()
    training_time = end_time - start_time
    logging.info(f"{model_name} trained in {training_time:.2f} seco
    clf_dict[model_name] = model
```

```
INFO:root:Linear SVM trained in 0.03 seconds
INFO:root:Neural Networks trained in 3.55 seconds
INFO:root:Decision Tree trained in 0.18 seconds
INFO:root:Random Forest trained in 1.81 seconds
```

Precision-Recall

LinearSVM

In [457]:
```
# Does not have the .predict_proba method
clf = clf_dict['Linear SVM']

# Use decision function to get the decision scores
decision_scores = clf.decision_function(X_test)

# For precision-recall curve, directly use decision scores
lsvm_pr, lsvm_rc, _ = precision_recall_curve(y_test, decision_score

# For average precision score, use the decision scores as well
lsvm_avg_prec = average_precision_score(y_test, decision_scores)
```

Neural Networks

In [458]:
```
clf = clf_dict['Neural Networks']

y_proba = clf.predict_proba(X_test)
nn_pr, nn_rc, _ = precision_recall_curve(y_test, y_proba[:, 1])
nn_avg_prec = average_precision_score(y_test, y_proba[:, 1])
```

Decision Tree

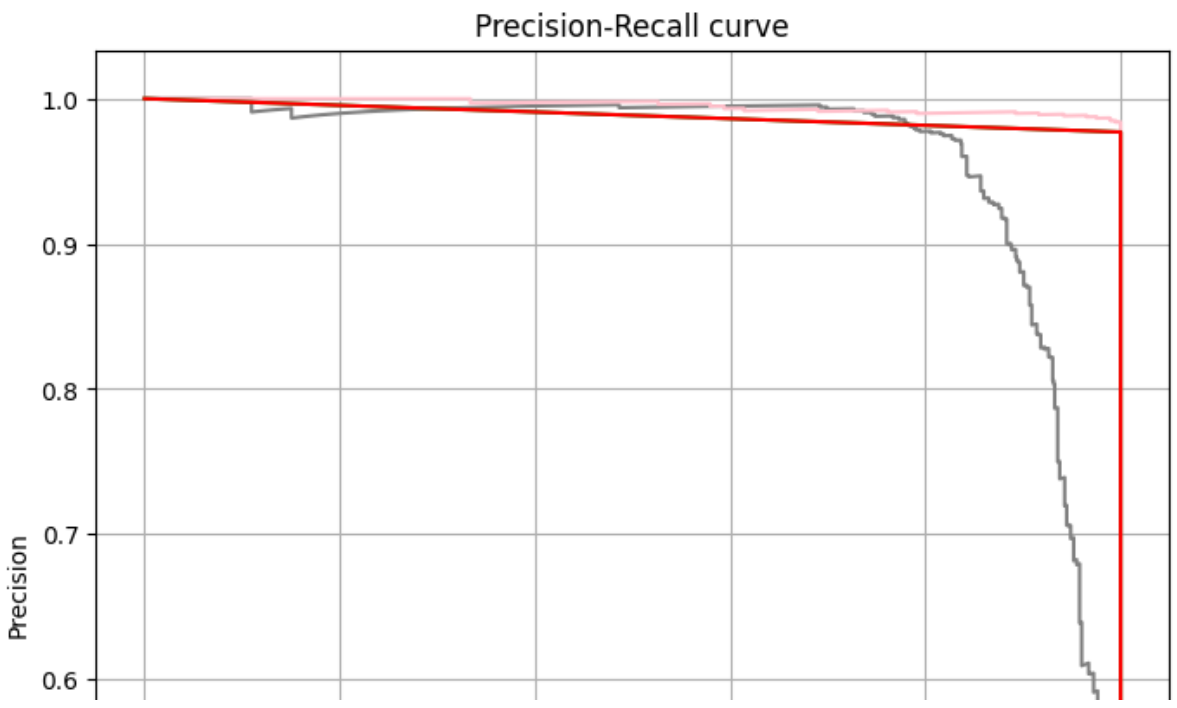
In [459]:
```
clf = clf_dict['Decision Tree']

y_proba = clf.predict_proba(X_test)
dt_pr, dt_rc, _ = precision_recall_curve(y_test, y_proba[:, 1])
dt_avg_prec = average_precision_score(y_test, y_proba[:, 1])
```

Random Forest

In [460]:
```
clf = clf_dict['Decision Tree']

y_proba = clf.predict_proba(X_test)
rf_pr, rf_rc, _ = precision_recall_curve(y_test, y_proba[:, 1])
rf_avg_prec = average_precision_score(y_test, y_proba[:, 1])
```

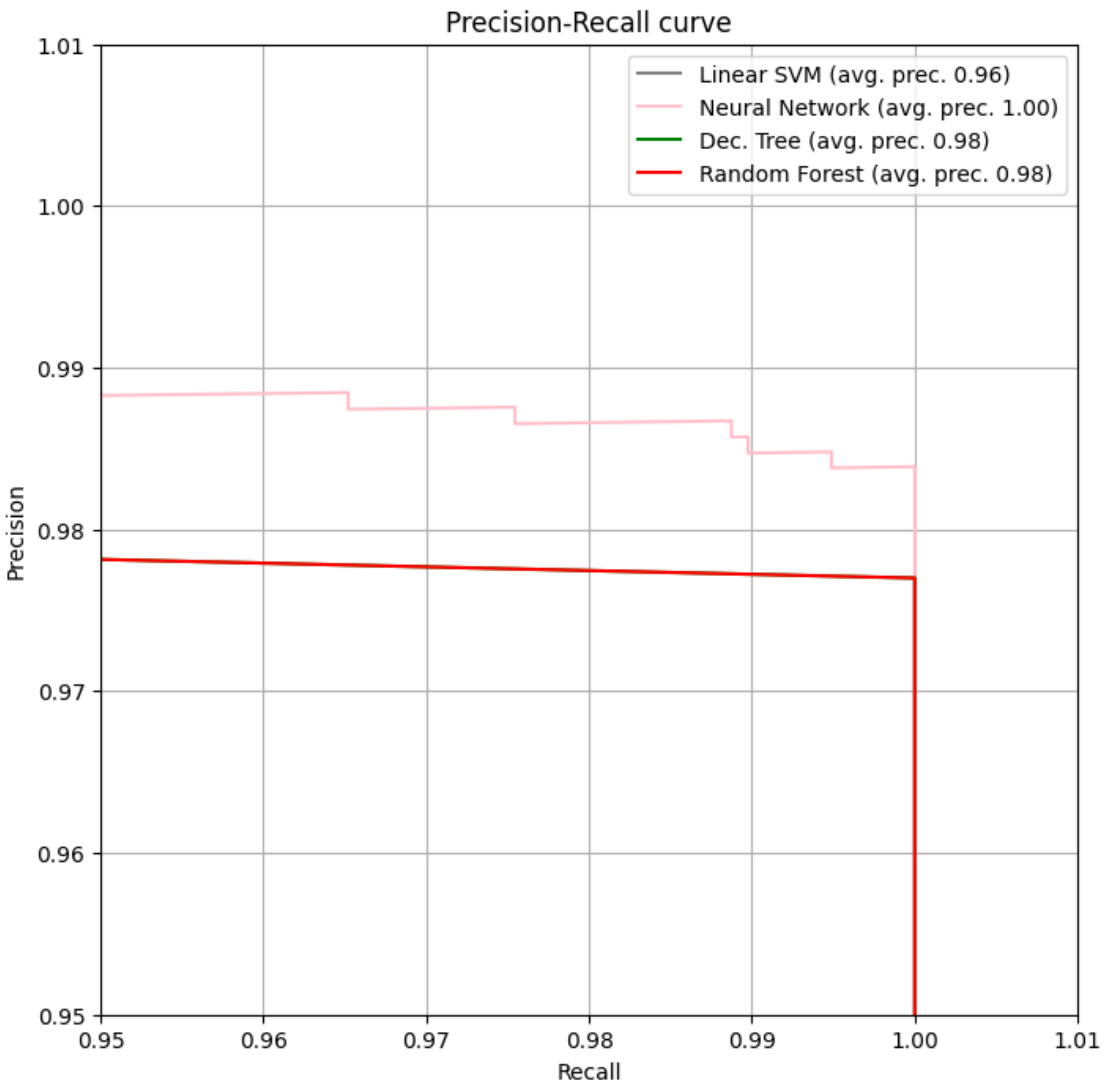
Plot the Curves

In [461]:
```
plt.figure(figsize=(8, 8))

plt.plot(lsvm_rc, lsvm_pr, c='grey', label=f'Linear SVM (avg. prec.
plt.plot(nn_rc, nn_pr, c='pink', label=f'Neural Network (avg. prec.
plt.plot(dt_rc, dt_pr, c='green', label=f'Dec. Tree (avg. prec. {dt
plt.plot(rf_rc, rf_pr, c='red', label=f'Random Forest (avg. prec. {

plt.legend()

plt.grid()

plt.title('Precision-Recall curve')
plt.xlabel('Recall')
plt.ylabel('Precision')

plt.show()
```



It is evident that the Neural Network classifier continues to exhibit superior performance in terms of Precision-Recall analysis. In contrast, both the Random Forest and Decision Tree classifiers appear to yield comparable results in this regard.

In [463]:
```
plt.figure(figsize=(8, 8))

plt.plot(lsvm_rc, lsvm_pr, c='grey', label=f'Linear SVM (avg. prec.
plt.plot(nn_rc, nn_pr, c='pink', label=f'Neural Network (avg. prec.
plt.plot(dt_rc, dt_pr, c='green', label=f'Dec. Tree (avg. prec. {dt
plt.plot(rf_rc, rf_pr, c='red', label=f'Random Forest (avg. prec. {

plt.xlim([0.95, 1.01])
plt.ylim([0.95, 1.01])
plt.legend()

plt.grid()

plt.title('Precision-Recall curve')
plt.xlabel('Recall')
plt.ylabel('Precision')

plt.show()
```



ROC Curve

LinearSVM

In [464]:
```
clf = clf_dict['Linear SVM']

# Use decision function to get the decision scores
decision_scores = clf.decision_function(X_test)

# For precision-recall curve, you can directly use decision scores
lsvm_fpr, lsvm_tpr, _ = roc_curve(y_test, decision_scores)

# For average precision score, you can use the decision scores as w
lsvm_auc = roc_auc_score(y_test, decision_scores)
```

Neural Networks

In [465]:
```
clf = clf_dict['Neural Networks']

y_proba = clf.predict_proba(X_test)
nn_fpr, nn_tpr, _ = roc_curve(y_test, y_proba[:, 1])
nn_auc= roc_auc_score(y_test, y_proba[:, 1])
```

Decision Tree

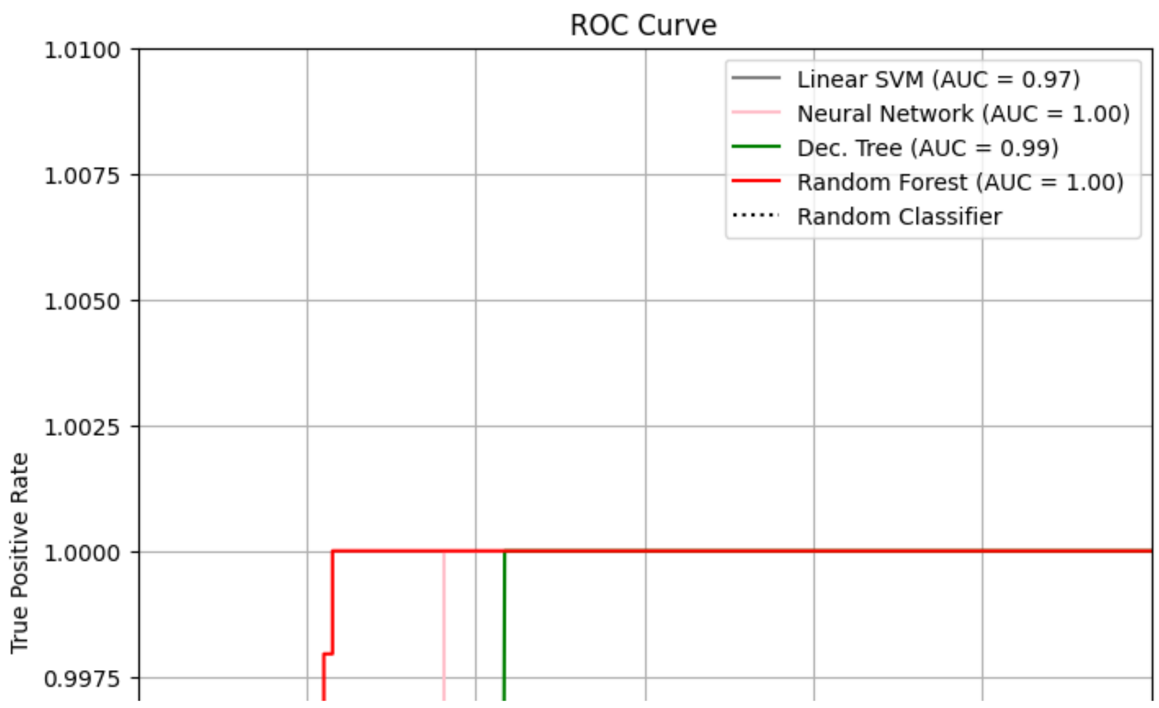
In [466]:
```
clf = clf_dict['Decision Tree']

y_proba = clf.predict_proba(X_test)
dt_fpr, dt_tpr, _ = roc_curve(y_test, y_proba[:, 1])
dt_auc= roc_auc_score(y_test, y_proba[:, 1])
```

Random Forest

In [467]:
```
clf = clf_dict['Random Forest']

y_proba = clf.predict_proba(X_test)
rf_fpr, rf_tpr, _ = roc_curve(y_test, y_proba[:, 1])
rf_auc= roc_auc_score(y_test, y_proba[:, 1])
```

Plot the Curve

In [468]:
```
plt.figure(figsize=(8, 8))

plt.plot(lsvm_fpr, lsvm_tpr, c='grey', label=f'Linear SVM (AUC = {l
plt.plot(nn_fpr, nn_tpr, c='pink', label=f'Neural Network (AUC = {n
plt.plot(dt_fpr, dt_tpr , c='green', label=f'Dec. Tree (AUC = {dt_a
plt.plot(rf_fpr, rf_tpr, c='red', label=f'Random Forest (AUC = {rf_

plt.xlim([-0.01, 0.05])
plt.ylim([0.99, 1.01])

plt.plot([0, 1], [0, 1], 'k:', label='Random Classifier')

plt.legend()

plt.grid()

plt.title('ROC Curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')

plt.show()
```



Once more, the Random Forest classifier has yielded the most favorable outcome in ROC Curve analysis, demonstrating a nearly ideal ROC curve.

Calibration

In [469]:
```
fig = plt.figure(figsize=(8, 8))
gs = GridSpec(1, 1)
ax_calibration_curve = fig.add_subplot(gs[:1, :1])
colors = plt.cm.get_cmap("Dark2")

calibration_displays = {}

for i, (name, clf) in enumerate(clf_dict.items()):
    try:
        y_proba = clf.predict_proba(X_test)
        display = CalibrationDisplay.from_predictions(
            y_test,
            y_proba[:, 1],
            n_bins=10,
            name=name,
            ax=ax_calibration_curve,
            color=colors(i),
        )
        calibration_displays[name] = display
    except Exception:
        continue

plt.xlim([0, 1])
plt.ylim([0, 1])

plt.grid()

plt.title("Calibration plots")

plt.show()
```

```
/var/folders/vz/c9yq75ls71bg54ncvlqv26dr0000gn/T/ipykernel_17192/1
114094156.py:4: MatplotlibDeprecationWarning: The get_cmap functio
n was deprecated in Matplotlib 3.7 and will be removed two minor r
eleases later. Use ``matplotlib.colormaps[name]`` or ``matplotlib.
colormaps.get_cmap(obj)`` instead.
  colors = plt.cm.get_cmap("Dark2")
```

Despite a somewhat diminished performance compared to the previous dataset, the Decision Tree model consistently demonstrates its superior calibration capabilities.

Anomaly Detection

Anomaly detection is a vital technique for uncovering unusual activities, particularly when dealing with imbalanced datasets, as exemplified in the context of credit card fraud detection. In this analysis, the focus is on a dataset that spans two days and contains a total of 284,807 transactions. Among these, a mere 492 cases are instances of fraudulent transactions. This glaring class imbalance illustrates that fraudulent transactions represent only 0.172% of the entire dataset. This is a stark contrast to the previous dataset, which had a more even distribution of classes.

Given this extreme class imbalance, anomaly detection becomes the preferred approach. It excels at identifying rare and unexpected events, such as fraudulent credit card transactions, where traditional supervised learning techniques may struggle.

In [4]:
```
df0 = pd.read_csv(r"/Users/federicastefanizzi/Downloads/creditcard.
```

In [5]:
```
df0
```

Out[5]:

	Time	V1	V2	V3	V4	V5	V6	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592
...	...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577

284807 rows × 31 columns

In [6]:
```
# Create a StandardScaler instance
scaler = StandardScaler()

# Fit and transform the 'Amount' column
df0['Amount'] = scaler.fit_transform(df0['Amount'].values.reshape(-

# Now, the 'Amount' column is standardized
```

In [7]:
```
# Create a StandardScaler instance
scaler = StandardScaler()

# Fit and transform the 'Amount' column
df0['Time'] = scaler.fit_transform(df0['Time'].values.reshape(-1, 1

# Now, the 'Amount' column is standardized
```

In [8]:
```
class_distribution = df0['Class'].value_counts()
print(class_distribution)
```

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

In [9]:
```
df0.isnull().sum()
```

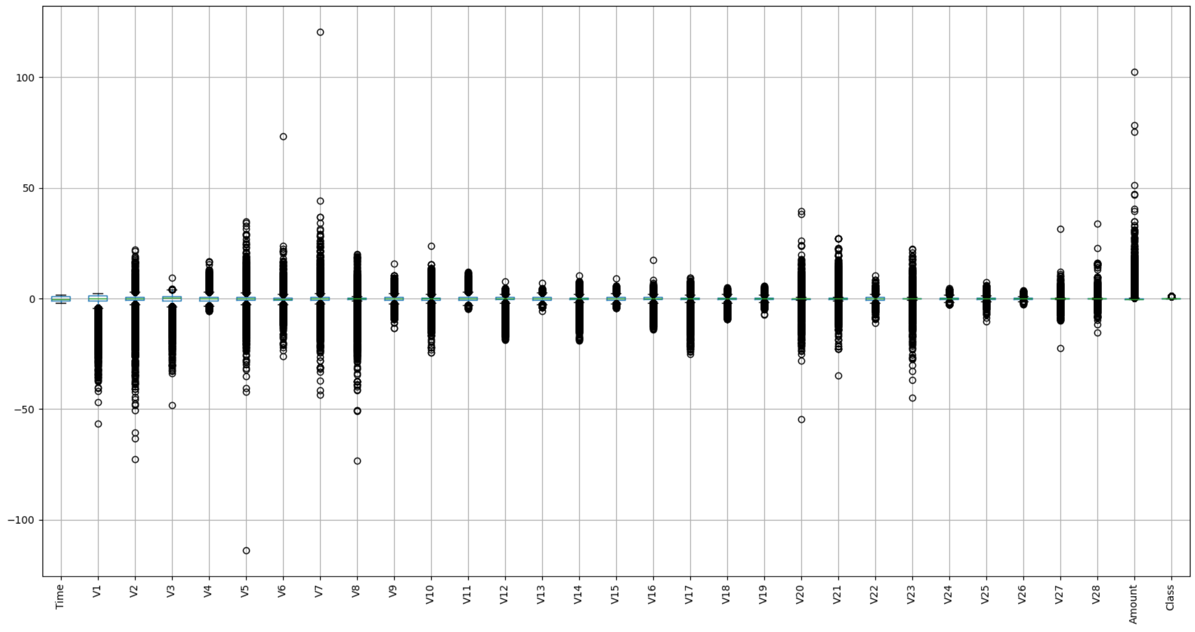
Out[9]:
```
Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
```

Based on the provided table, it is evident that there are **no missing values** present in the dataset.

Boxplot Visualization

Using a boxplot to visualize data is often done to verify the presence of outliers, which is an important step in preparing for anomaly detection. Boxplots provide a clear visual representation of the distribution of data, including the identification of potential outliers. Outliers can be data points that deviate significantly from the typical range of values and may be indicative of anomalies or unusual events.

In [10]:
```
ax = df0.boxplot(figsize=(20, 10))
plt.setp(ax.get_xticklabels(), rotation=90);
```



The median values for most variables seem to hover around the zero line, suggesting that the data might be centered or standardized around zero. However, 'Amount' stands out with a higher median value than the others even if it has been standardized (probably due to its skewness).

Outliers?

A noteworthy observation is the presence of a substantial number of outliers, as evidenced by data points lying outside the boundaries defined by the boxplot whiskers, across nearly all variables. In the cases of many columns, outliers are observed to extend significantly beyond the interquartile range (IQR). Additionally, the 'Amount' variable exhibits a substantial count of outliers characterized by elevated values.

Preparatory Steps for Anomaly Detection

Step 1: Check for missing values in the dataset

In [11]:
```
result = df0.isna().drop_duplicates()
result
```

Out[11]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23
0	False	False	False	False	False	False	False	False	False	False	...	False	False	False

1 rows × 31 columns

The variable 'result' will contain a DataFrame with duplicate rows removed, and each cell will indicate whether the corresponding cell in the original DataFrame df0 contains a missing value (True) or not (False).

Every cell in the resulting df has the value False, which means that there are no missing values in any of the columns of the original df 'df0.' In other words, all values in 'df0' are non-missing, and the dataframe is considered complete with respect to missing data.

Step 2: Assign the result to tmp_df

In [12]:
```
tmp_df = result
```

Step 3: Identify columns with missing values

In [13]:
```
missing_columns = [x for x in tmp_df.columns if len(tmp_df[tmp_df[x

if not missing_columns:
    print("Columns with Missing Values: None")
else:
    print("Columns with Missing Values:")
    for column in missing_columns:
        print(column)
```

```
Columns with Missing Values: None
```

The output message indicates that there are no columns in the dataset containing missing values.

Step 4: Remove selected features not needed for anomaly detection

In [14]:
```
true_labels = df0['Class'].values
true_labels
```

Out[14]:
```
array([0, 0, 0, ..., 0, 0, 0])
```

In [15]:
```
selected_features = ['Class', 'Time']
df0 = df0.drop(selected_features, axis=1)
```

Before performing the anomaly detection task, two key columns were removed from the dataset: the 'Class' column and the 'Time' column. This was done for specific reasons:

- Target Variable 'Class': The 'Class' column, which indicates whether a data point is normal or an anomaly, was removed. This step ensures that the model doesn't learn to detect anomalies based on the target variable itself, preventing overfitting and ensuring unbiased detection.
- Temporal Information 'Time': The 'Time' column, which represents timestamps, was also removed. While timestamps are valuable for certain analyses, they may not provide relevant information for anomaly detection. By excluding it, the model focuses solely on identifying unusual patterns without being influenced by time-related factors.

In [16]:
```
df0
```

Out[16]:

	V1	V2	V3	V4	V5	V6	V7	
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270
...	...	...	...	...	...	...	...	
284802	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305
284803	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294
284804	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708
284805	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679
284806	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414

284807 rows × 29 columns

Performing Anomaly Detection

In [17]:
```
# Set the outlier fraction
outlier_fraction = 0.01
```

In the provided step, the objective is to specify the anticipated proportion of outliers within the dataset. In this particular instance, the parameter is set to 1% (0.01), signifying an expectation that roughly 1% of the dataset's data points are anomalies.

This parameterization is important in anomaly detection as it establishes a predefined threshold for identifying anomalies within the dataset. By designating this value, the level of sensitivity the algorithm should exhibit when detecting data points that deviate significantly from the norm is established. The choice of 1% signifies a relatively low tolerance for anomalies, and it means that the algorithm will only flag data points as anomalies if they deviate significantly from what is considered normal. This cautious approach aims to minimize the chances of incorrectly classifying regular data as anomalies.

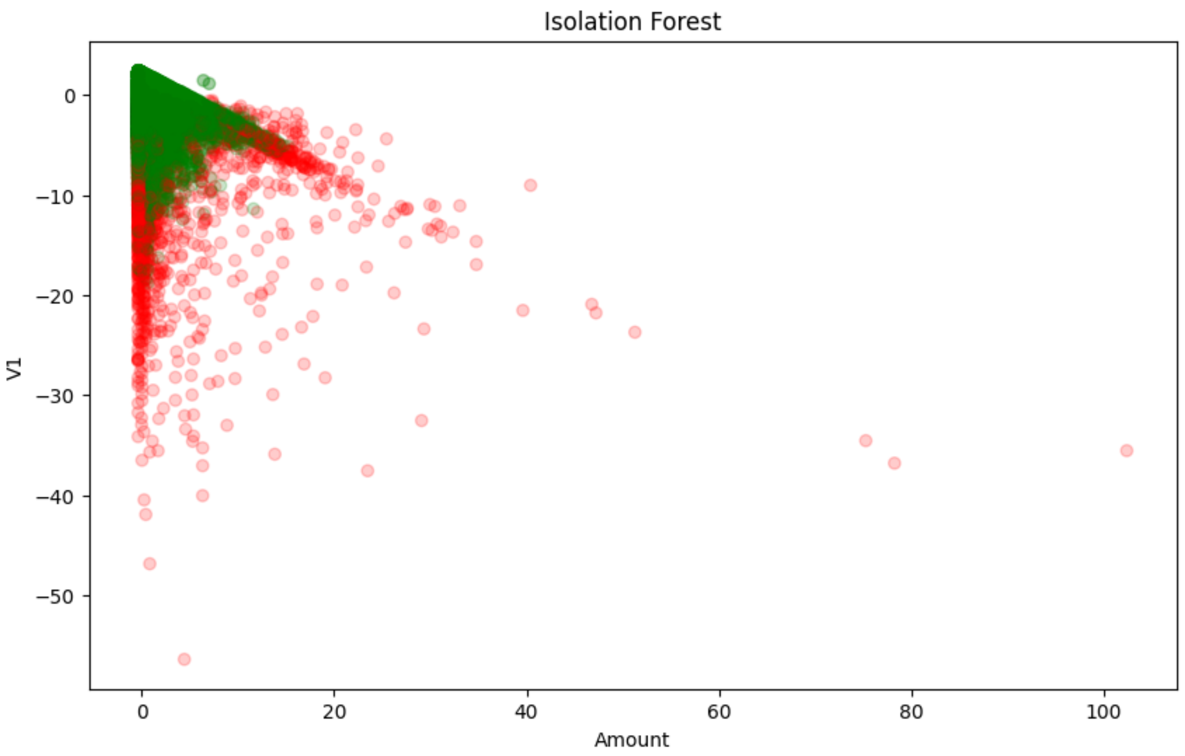
Isolation Forest

The Isolation Forest algorithm represents a robust and versatile approach to the task of anomaly detection. At its core, Isolation Forest distinguishes itself by adopting a unique perspective on anomalies—viewing them as rare and isolated data points within a larger, more homogeneous population. Unlike traditional density-based methods, Isolation Forest leverages the concept of isolation through random partitioning, effectively simulating the process of isolating anomalies. By quantifying the efficiency with which individual data points can be isolated, Isolation Forest assigns anomaly scores, providing a principled means of identifying outliers without making restrictive assumptions about the data's underlying distribution. Its scalability, ability to handle high-dimensional data, and independence from distribution assumptions render Isolation Forest a valuable tool in the realm of anomaly detection, offering insights into irregularities that may have otherwise remained concealed within complex datasets.

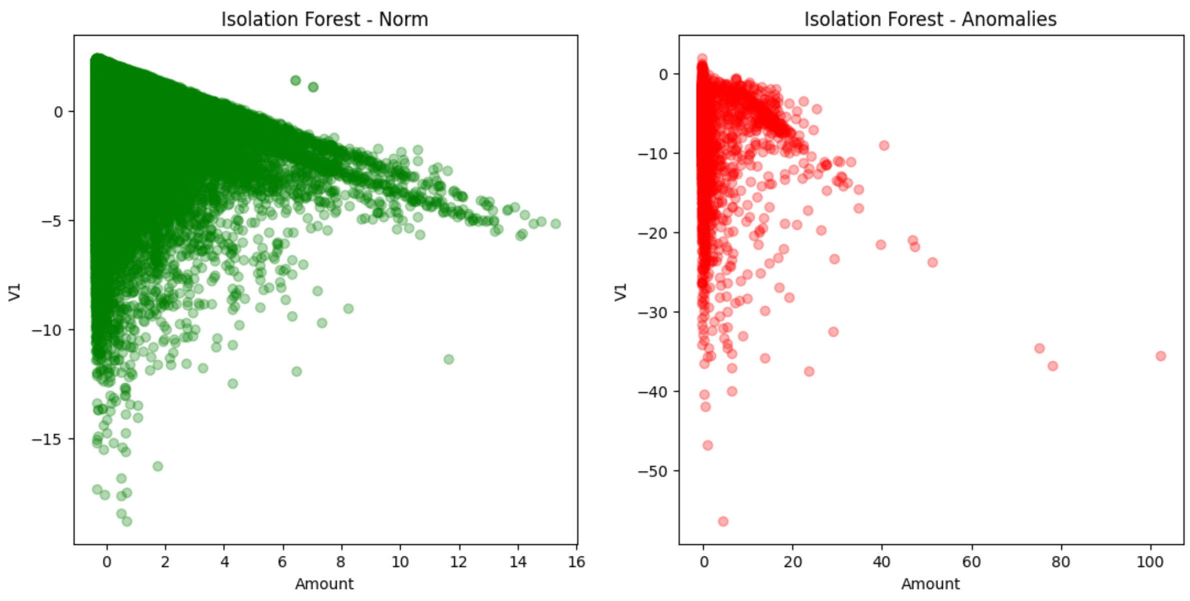
In [23]:
```
# Register start time
t_start = time.time()

iso_forest = IsolationForest(contamination=outlier_fraction, random
iso_forest.fit(df0)
y_pred = iso_forest.predict(df0)

# Register stop time
t_stop = time.time()

print(f"Fitting time: {t_stop - t_start:.2f} s")
```

```
Fitting time: 1.78 s
```

In [24]:
```python
# Check if the lengths match

colors = ['red', 'green']  # Define your two distinct colors here

if len(y_pred) == len(df0):
    # Assuming 'colors' is a list with at least 2 distinct colors
    point_colors = [colors[0] if pred == -1 else colors[1] for pred

    plt.figure(figsize=(10, 6))
    plt.scatter(df0['Amount'], df0['V1'], color=point_colors, alpha

    plt.title('Isolation Forest')
    plt.ylabel('V1')
    plt.xlabel('Amount')

    plt.show()
else:
    print("The length of y_pred should match the length of df0.")
```

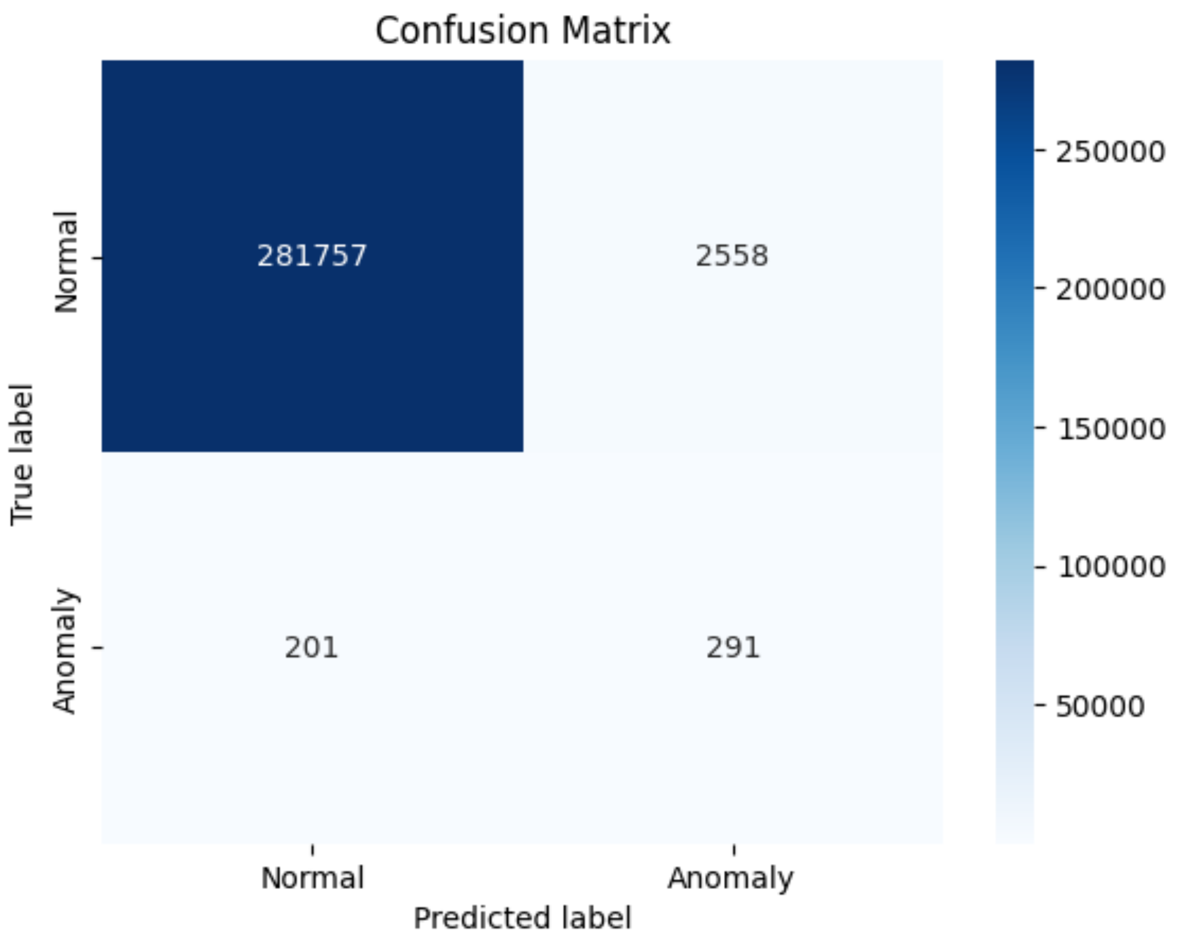


In [25]:
```
import matplotlib.pyplot as plt

if len(y_pred) == len(df0):
    fig, ax = plt.subplots(1, 2, figsize=(13,6))

    ax[0].scatter(df0[y_pred == 1]['Amount'], df0[y_pred == 1]['V1'
    ax[0].set_title('Isolation Forest - Norm')

    ax[1].scatter(df0[y_pred == -1]['Amount'], df0[y_pred == -1]['V
    ax[1].set_title('Isolation Forest - Anomalies')

    for idx in [0, 1]:
        ax[idx].set_ylabel('V1')
        ax[idx].set_xlabel('Amount')

    plt.show()
else:
    print("The length of y_pred should match the length of df0.")
```



In [27]:
```
# Convert y_pred to a binary vector where -1 becomes 1 (anomaly) an
binary_y_pred = [1 if x == -1 else 0 for x in y_pred]
```

In [28]:
```
# Compute confusion matrix
conf_matrix = confusion_matrix(true_labels, binary_y_pred)

# Plot confusion matrix
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xtickla
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.title('Confusion Matrix')
plt.show()
```



The confusion matrix presented visualizes the performance of the Isolation Forest algorithm for anomaly detection. In this matrix, the top left quadrant represents the true negatives, indicating that 281,757 normal instances were correctly identified as normal. The top right quadrant shows the false positives, with 2,558 normal instances incorrectly labeled as anomalies. The bottom left quadrant represents the false negatives, consisting of 201 instances that were actual anomalies but were predicted as normal. Lastly, the bottom right quadrant displays the true positives, where 291 anomalies were correctly identified.

The high number of true negatives suggests that the model is very effective at identifying normal instances. However, the presence of false positives indicates that there is some degree of error where normal behavior is being misclassified as anomalous. The true positives count reflects the model's capacity to correctly identify genuine anomalies, which is crucial in many applications such as fraud detection. The relatively low number of false negatives is encouraging, as it indicates that few anomalies are missed by the model. However, the balance between false positives and false negatives should be carefully considered in the context of the specific costs associated with each type of error in the application domain. The precision, recall, and F1-score derived from these values would provide further insights into the model's performance, particularly the trade-off between sensitivity (the ability to detect anomalies) and specificity (the ability to recognize normal instances).

In [29]:
```
# Compute classification report
class_report = classification_report(true_labels, binary_y_pred)
print(class_report)
```

```
              precision    recall  f1-score   support

           0       1.00      0.99      1.00    284315
           1       0.10      0.59      0.17       492

    accuracy                           0.99    284807
   macro avg       0.55      0.79      0.58    284807
weighted avg       1.00      0.99      0.99    284807
```

The classification report indicates that the Isolation Forest model has a high ability to correctly identify normal instances (label 0) with a precision of 1.00 and a recall of 0.99. This is further reflected in the perfect F1-score of 1.00 for the normal class. However, the model's ability to identify anomalies (label 1) is considerably lower, with a precision of 0.10 and a recall of 0.59, resulting in a low F1-score of 0.17.

The low precision for the anomaly class suggests that while the model can detect anomalies to some extent (as shown by the recall), it also misclassifies a lot of normal instances as anomalies. This is indicated by the low F1-score, which shows a poor balance between precision and recall for the anomaly class.

The overall accuracy of the model is high at 0.99, but this metric is skewed by the large number of normal instances, which is common in anomaly detection tasks where anomalies are rare. The macro average gives equal weight to both classes and thus drops to 0.55 for precision and 0.58 for F1-score, which suggests that the model's performance is not as robust for the less represented anomaly class.

Given the large support for the normal class compared to the relatively small support for the anomaly class, the model's performance appears to be strongly influenced by the class imbalance. This can lead to a model that is overfitted to the normal instances and underperforming in identifying true anomalies.

One Class SVM

The One-Class Support Vector Machine (One-Class SVM) is a powerful and versatile algorithm designed for the detection of anomalies or outliers within datasets. Operating on the premise that anomalies are rare occurrences, One-Class SVM seeks to create a boundary or hyperplane that encapsulates the majority of data points while minimizing the inclusion of anomalies. Unlike traditional SVMs used for classification, One-Class SVM focuses on identifying data points that deviate significantly from the norm, making it well-suited for applications where anomalies are infrequent and of particular interest. By constructing a high-dimensional separation boundary, One-Class SVM excels in scenarios like fraud detection, where isolating exceptional observations from a predominantly normal dataset is of paramount importance.

Given the computational complexity of this model, the only viable approach to its utilization involved conducting a downsampling operation on the dataframe.

In [65]:
```
df = pd.read_csv('/Users/federicastefanizzi/Downloads/creditcard.cs
```

In [66]:
```
from sklearn.utils import resample

desired_ratio = 30
df_majority = df[df['Class'] == 0]
df_minority = df[df['Class'] == 1]

n_samples_majority = len(df_minority) * desired_ratio

df_majority_undersampled = resample(df_majority,
                                    replace=False,
                                    n_samples=int(n_samples_majorit
                                    random_state=42)

df_undersampled = pd.concat([df_majority_undersampled, df_minority]

df_undersampled = df_undersampled.sample(frac=1, random_state=42)

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

```
Class
0    14760
1      492
Name: count, dtype: int64
```

In [67]:
```
df_undersampled
```

Out[67]:

	Time	V1	V2	V3	V4	V5	V6	V
78719	57664.0	1.246647	-0.042646	-1.172281	0.271970	2.177943	3.609955	-0.42765
77976	57311.0	1.141142	-0.594963	-2.180269	-1.950412	2.148106	2.607154	0.09922
218888	141510.0	-0.804532	0.561754	-0.134167	0.870744	1.484100	-0.413485	1.46713
208402	137104.0	-0.109517	1.199368	-3.604309	0.036522	1.370718	-0.875119	2.04534
248296	153875.0	-0.613696	3.698772	-5.534941	5.620486	1.649263	-2.335145	-0.90718
...	...	...	...	...	...	...	...	
174211	121820.0	0.315677	0.792154	-1.980506	0.166049	2.757871	3.954494	0.24941
137653	82265.0	-0.441724	1.006684	1.683470	0.052523	-0.299090	-1.087035	0.69616
108876	71116.0	-0.778253	0.130171	2.330368	1.743399	-0.602476	0.995023	0.31530
88007	61944.0	-2.242599	0.091694	1.335811	0.234568	1.097002	0.989207	-0.32539
279854	169135.0	-1.196644	1.930087	-3.515438	-1.333896	3.579285	2.453577	0.39347

15252 rows × 31 columns

In [68]:
```
true_labels = df_undersampled['Class'].values
true_labels
```

Out[68]:
```
array([0, 0, 0, ..., 0, 0, 0])
```

In [69]:
```
selected_features = ['Class', 'Time']
df_undersampled = df_undersampled.drop(selected_features, axis=1)
```

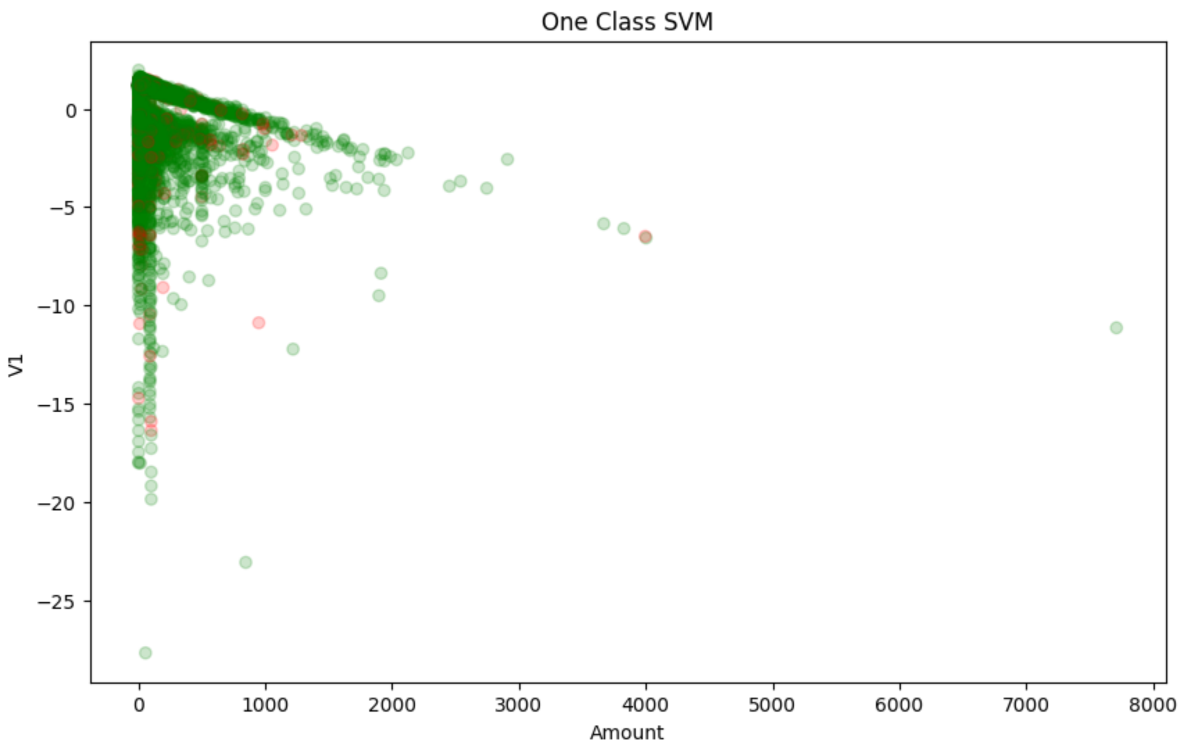
In [70]:
```
# Create a StandardScaler instance
scaler = StandardScaler()

# Fit and transform the 'Amount' column
df_undersampled['Amount'] = scaler.fit_transform(df_undersampled['A

# Now, the 'Amount' column is standardized
```

In [71]:
```
# Register start time
t_start = time.time()

# Create and fit the One-Class SVM model
y_pred_svm = OneClassSVM(nu=outlier_fraction, kernel="rbf", gamma=0

# Register stop time
t_stop = time.time()

# Calculate elapsed time
elapsed_time = t_stop - t_start
print(f"Elapsed time: {elapsed_time} seconds")
```

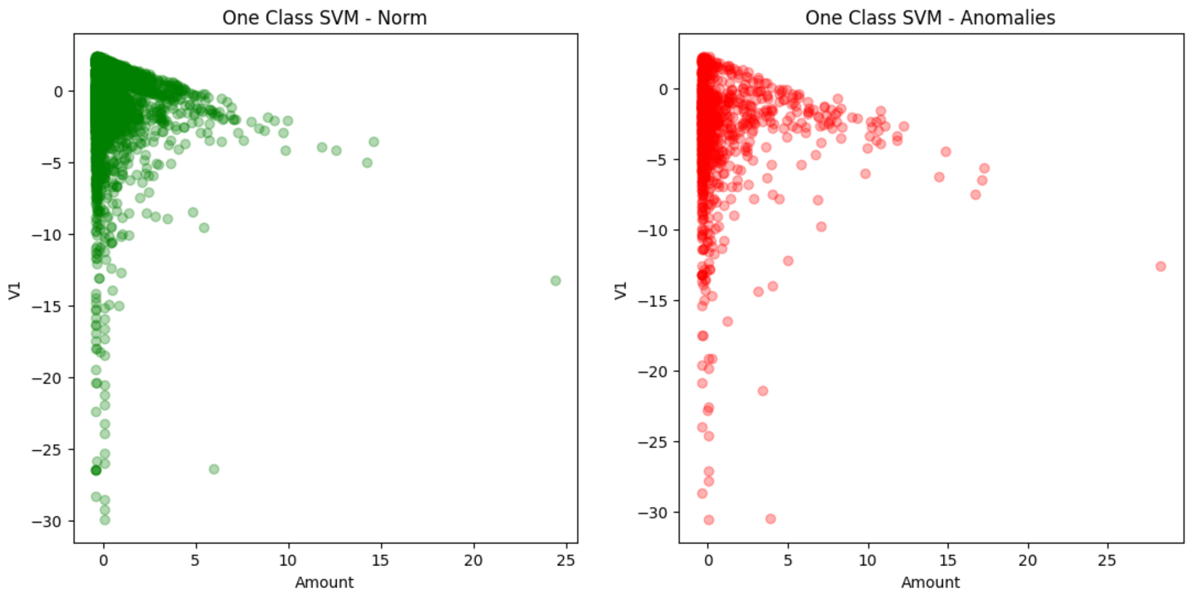
```
Elapsed time: 2.169666051864624 seconds
```

To visualize the predictions generated by the One-Class SVM on the entire original dataset, the original dataset (df) was used. This made it possible to gain insights into how the model's predictions are distributed across the complete dataset, including samples that were not part of the undersampling process.

In [72]:
```
df_subset = df.iloc[:len(y_pred_svm)]

# Create a color map for anomalies (red) and normal instances (gree
colors = ['red' if pred == -1 else 'green' for pred in y_pred_svm]

plt.figure(figsize=(10, 6))
plt.scatter(df_subset['Amount'], df_subset['V1'], c=colors, alpha=0

plt.title('One Class SVM')
plt.ylabel('V1')
plt.xlabel('Amount')

plt.show()
```

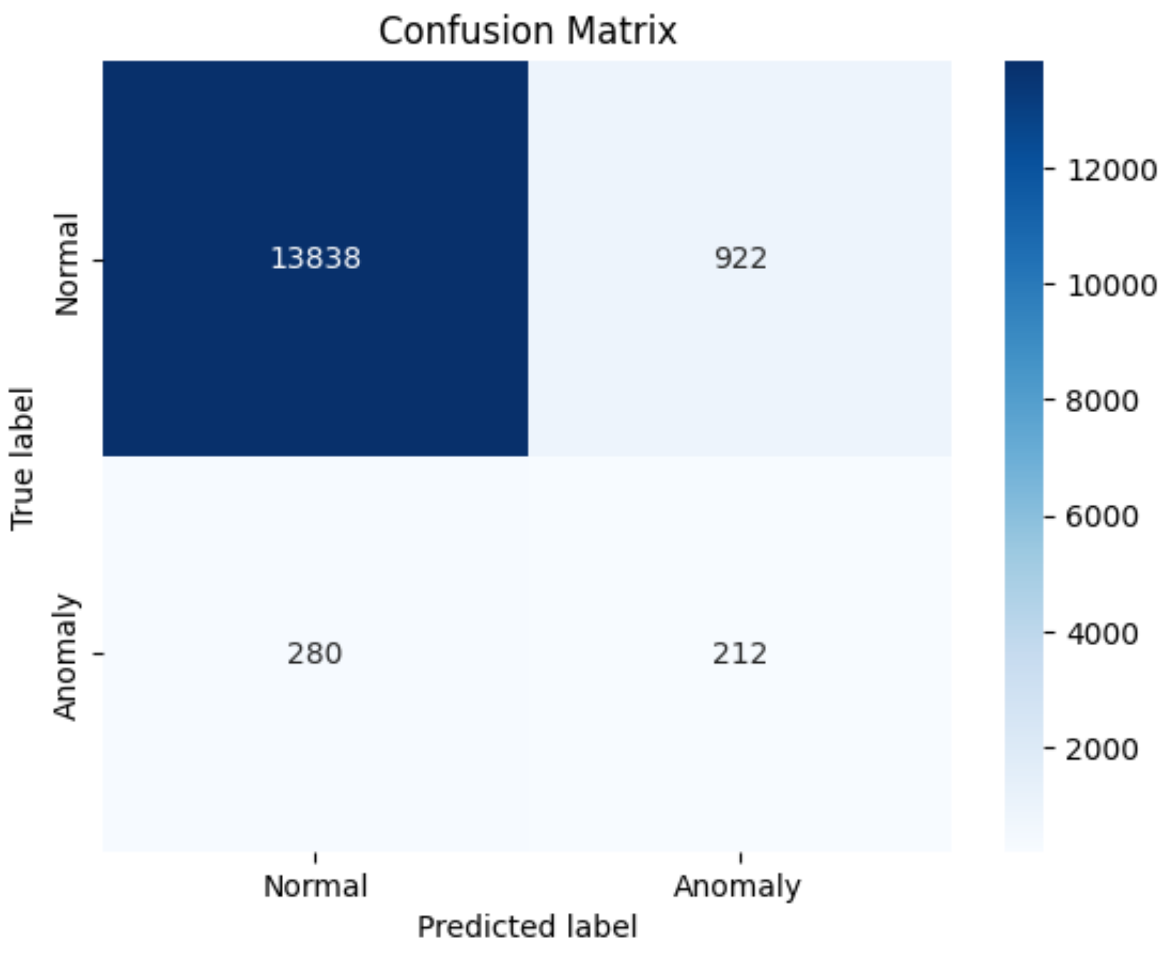


In [73]:
```
fig, ax = plt.subplots(1, 2, figsize=(13, 6))

# Using y_pred_svm for non-anomalies (normal)
ax[0].scatter(df_undersampled[y_pred_svm == 1]['Amount'], df_unders
ax[0].set_title('One Class SVM - Norm')
ax[0].set_ylabel('V1')
ax[0].set_xlabel('Amount')

# Using y_pred_svm for anomalies
ax[1].scatter(df_undersampled[y_pred_svm == -1]['Amount'], df_under
ax[1].set_title('One Class SVM - Anomalies')
ax[1].set_ylabel('V1')
ax[1].set_xlabel('Amount')

plt.show()
```



In [74]:
```
# Convert y_pred_svm to a binary vector where -1 becomes 1 (anomaly
binary_y_pred_svm = [1 if x == -1 else 0 for x in y_pred_svm]
```

In [75]:
```
# Compute confusion matrix
conf_matrix = confusion_matrix(true_labels, binary_y_pred_svm)

# Plot confusion matrix
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xtickla
plt.ylabel('True label')
plt.xlabel('Predicted label')
plt.title('Confusion Matrix')
plt.show()
```



The confusion matrix displayed reflects the performance of a One-Class Support Vector Machine (OC-SVM) for anomaly detection. Here, the true negatives are substantial (13,838), illustrating the model's competency in correctly classifying normal instances. Conversely, there are a considerable number of false positives (922), where normal instances were mistakenly identified as anomalies. This may suggest an over-sensitive model or a misshaped decision boundary. In the realm of detecting actual anomalies, the model successfully identified 212, which are the true positives, but also missed 280, indicated by the false negatives. These missed anomalies highlight a potential area for improvement in the model's sensitivity to anomalous behaviour.

While the OC-SVM model demonstrates a good ability to recognize normal behaviour, the balance between recognizing true anomalies and avoiding false alarms is not optimal. For practical applications, particularly those where failing to detect anomalies can have serious consequences, the model requires refinement. Enhancing its precision and recall is crucial, as these metrics will provide a deeper understanding of its performance, especially in terms of its ability to balance detecting anomalies against falsely identifying normal behaviour as anomalous.

In [76]:
```
# Compute classification report
class_report = classification_report(true_labels, binary_y_pred_svm
print(class_report)
```

```
              precision    recall  f1-score   support

           0       0.98      0.94      0.96     14760
           1       0.19      0.43      0.26       492

    accuracy                           0.92     15252
   macro avg       0.58      0.68      0.61     15252
weighted avg       0.95      0.92      0.94     15252
```

The classification report for the One-Class Support Vector Machine (OC-SVM) used in anomaly detection reveals a high precision (0.98) and recall (0.94) for the normal class (label 0), leading to a high F1-score (0.96). This indicates that the model is quite effective at identifying normal instances. However, for the anomaly class (label 1), the precision is low (0.19), and the recall is moderate (0.43), resulting in a much lower F1-score (0.26).

This disparity signifies that while the model has some capability to detect anomalies, it incorrectly classifies a large number of normal instances as anomalies (low precision), and also misses several actual anomalies (moderate recall). The low F1-score for the anomaly class underscores the challenge the model faces in balancing the precision and recall for this class.

Overall accuracy of the model is high (0.92), but this is primarily influenced by the model's ability to identify the normal instances, which are the majority class in the dataset. The macro average scores (precision: 0.58, recall: 0.68, F1-score: 0.61) are more reflective of the model's performance across both classes, showing that there is room for improvement, especially in correctly identifying anomalies.

The class imbalance, indicated by the 'support' values, likely affects the model's performance, suggesting a potential overfit to the normal class.

Conclusions

In conclusion, this study on credit card fraud detection emphasizes the superiority of specific machine learning models for classifying financial transactions. The availability of labeled data in the dataset guided the focus towards supervised learning algorithms, notably neural networks and random forests, which outperformed unsupervised methods. Unsupervised algorithms, particularly clustering techniques, showed suboptimal performance, likely due to their limited suitability for binary classification tasks like fraud detection. The initial analysis, performed on a dataset characterized by balanced class distribution, was further deepened through the exploration of a dataset exhibiting a more realistic, imbalanced class distribution. In both scenarios, the aforementioned supervised models maintained their effectiveness. However, attempts at anomaly detection in the imbalanced dataset led to overfitting, with the models proficiently identifying normal instances but erring significantly in anomaly detection. This highlights the complexity and challenges in effectively detecting fraudulent activities in diverse dataset conditions.

```
In [ ]:
```