CREDIT CARD FRAUD DETECTION

CONTEXT

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

The dataset contains transactions made by credit cards in September 2013 by European cardholders. This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. It contains only numerical input variables which are the result of a PCA transformation. Unfortunately, due to confidentiality issues, we cannot provide the original features and more background information about the data. Features V1, V2, ... V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

Objective

- To predict whether a card transaction is fraudulent or not.
- Which variables are most significant.

In [1]:

```
import warnings
warnings.filterwarnings("ignore")

# Libraries to help with reading and manipulating data

import pandas as pd
import numpy as np

# Library to split data
from sklearn.model_selection import train_test_split

#Libraries to help with model building
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier, RandomF

# Libaries to help with data visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Removes the limit from the number of displayed columns and rows.
```

```
# This is so I can see the entire dataframe when I  print it
pd.set_option("display.max_columns", None)
# pd.set_option('display.max_rows', None)
pd.set_option("display.max_rows", 200)

# To build linear model for statistical analysis and prediction
import statsmodels.stats.api as sms
from statsmodels.stats.outliers_influence import variance_inflation_factor
import statsmodels.api as sm
from statsmodels.tools.tools import add_constant
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# To get diferent metric scores
from sklearn import metrics
from sklearn.metrics import accuracy_score, recall_score, precision_score, roc_auc_s
from sklearn.model_selection import GridSearchCV

# For pandas profiling
from pandas_profiling import ProfileReport
```

Import Dataset

In [2]:

```
transactions = pd.read_csv("creditcard.csv")
```

View the first and last 5 rows of the dataset.

In [3]:

```
transactions
```

Out[3]:

	Time	V1	V2	V3	V4	V5	V6	V7	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.09
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.08
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.24
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.37
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.27
...	...	...	...	...	...	...	...	...	
284802	172786.0	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.30
284803	172787.0	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.29
284804	172788.0	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.70
284805	172788.0	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.67
284806	172792.0	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.41

284807 rows × 31 columns

In [4]:

```
# copying data to another variable to avoid any changes to original data
data = transactions.copy()
```

Understand the shape of the dataset.

In [5]:
```
data.shape
```

Out[5]:
```
(284807, 31)
```

- The dataset has 284807 rows and 31 columns

Check the data types of the columns for the dataset.

In [6]:
```
data.info()
```

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

- Every column in the data is a numeric column.
- There are no null values in the data.

Summary of the dataset.

In [7]:
```
data.describe().T
```

Out[7]:

	count	mean	std	min	25%	50%	7
Time	284807.0	9.481386e+04	47488.145955	0.000000	54201.500000	84692.000000	139320.500

	count	mean	std	min	25%	50%	7
V1	284807.0	3.918649e-15	1.958696	-56.407510	-0.920373	0.018109	1.315
V2	284807.0	5.682686e-16	1.651309	-72.715728	-0.598550	0.065486	0.803
V3	284807.0	-8.761736e-15	1.516255	-48.325589	-0.890365	0.179846	1.027
V4	284807.0	2.811118e-15	1.415869	-5.683171	-0.848640	-0.019847	0.743
V5	284807.0	-1.552103e-15	1.380247	-113.743307	-0.691597	-0.054336	0.611
V6	284807.0	2.040130e-15	1.332271	-26.160506	-0.768296	-0.274187	0.398
V7	284807.0	-1.698953e-15	1.237094	-43.557242	-0.554076	0.040103	0.570
V8	284807.0	-1.893285e-16	1.194353	-73.216718	-0.208630	0.022358	0.327
V9	284807.0	-3.147640e-15	1.098632	-13.434066	-0.643098	-0.051429	0.597
V10	284807.0	1.772925e-15	1.088850	-24.588262	-0.535426	-0.092917	0.453
V11	284807.0	9.289524e-16	1.020713	-4.797473	-0.762494	-0.032757	0.739
V12	284807.0	-1.803266e-15	0.999201	-18.683715	-0.405571	0.140033	0.618
V13	284807.0	1.674888e-15	0.995274	-5.791881	-0.648539	-0.013568	0.662
V14	284807.0	1.475621e-15	0.958596	-19.214325	-0.425574	0.050601	0.493
V15	284807.0	3.501098e-15	0.915316	-4.498945	-0.582884	0.048072	0.648
V16	284807.0	1.392460e-15	0.876253	-14.129855	-0.468037	0.066413	0.523
V17	284807.0	-7.466538e-16	0.849337	-25.162799	-0.483748	-0.065676	0.399
V18	284807.0	4.258754e-16	0.838176	-9.498746	-0.498850	-0.003636	0.500
V19	284807.0	9.019919e-16	0.814041	-7.213527	-0.456299	0.003735	0.458
V20	284807.0	5.126845e-16	0.770925	-54.497720	-0.211721	-0.062481	0.133
V21	284807.0	1.473120e-16	0.734524	-34.830382	-0.228395	-0.029450	0.186
V22	284807.0	8.042109e-16	0.725702	-10.933144	-0.542350	0.006782	0.528
V23	284807.0	5.282512e-16	0.624460	-44.807735	-0.161846	-0.011193	0.147
V24	284807.0	4.456271e-15	0.605647	-2.836627	-0.354586	0.040976	0.439
V25	284807.0	1.426896e-15	0.521278	-10.295397	-0.317145	0.016594	0.350
V26	284807.0	1.701640e-15	0.482227	-2.604551	-0.326984	-0.052139	0.240
V27	284807.0	-3.662252e-16	0.403632	-22.565679	-0.070840	0.001342	0.091
V28	284807.0	-1.217809e-16	0.330083	-15.430084	-0.052960	0.011244	0.078
Amount	284807.0	8.834962e+01	250.120109	0.000000	5.600000	22.000000	77.165
Class	284807.0	1.727486e-03	0.041527	0.000000	0.000000	0.000000	0.000

- The minimum value for amount is 0, while the maximum value is 25691.16
- The minimum value for time is 0, while the maximum value is 172792.00

EDA

Univariate analysis

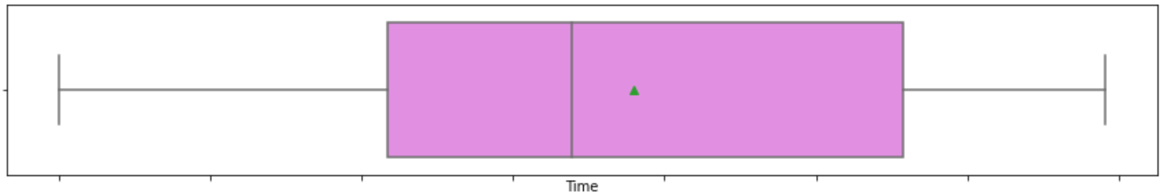
In [8]:

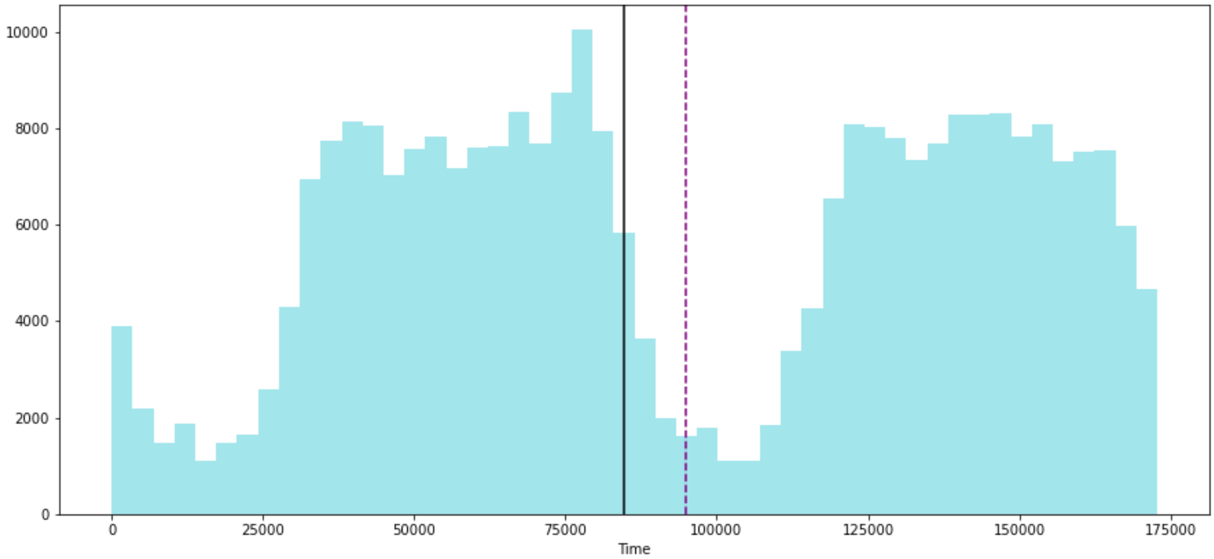
```
# While doing uni-variate analysis of numerical variables we want to study their cen
# and dispersion.
# Let us write a function that will help us create boxplot and histogram for any inp
# variable.
# This function takes the numerical column as the input and returns the boxplots
# and histograms for the variable.
# Let us see if this help us write faster and cleaner code.
def histogram_boxplot(feature, figsize=(15,10), bins = None):
    """ Boxplot and histogram combined
    feature: 1-d feature array
    figsize: size of fig (default (9,8))
    bins: number of bins (default None / auto)
    """
    f2, (ax_box2, ax_hist2) = plt.subplots(nrows = 2, # Number of rows of the subplo
                                           sharex = True, # x-axis will be shared am
                                           gridspec_kw = {"height_ratios": (.25, .75
                                           figsize = figsize
                                           ) # creating the 2 subplots
    sns.boxplot(feature, ax=ax_box2, showmeans=True, color='violet') # boxplot will
    sns.distplot(feature, kde=F, ax=ax_hist2, bins=bins,color = 'orange') if bins el
    ax_hist2.axvline(np.mean(feature), color='purple', linestyle='--') # Add mean to
    ax_hist2.axvline(np.median(feature), color='black', linestyle='-') # Add median
```

Observations on Time

In [9]:

```
histogram_boxplot(data['Time'])
```

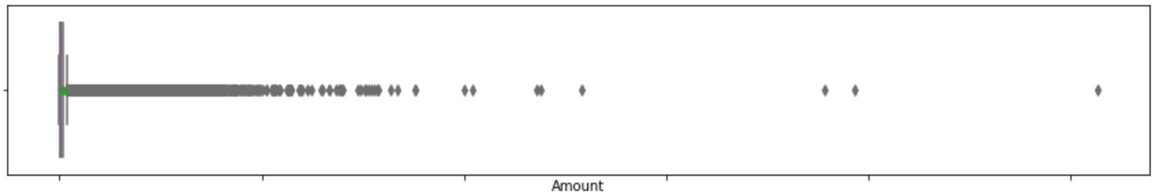


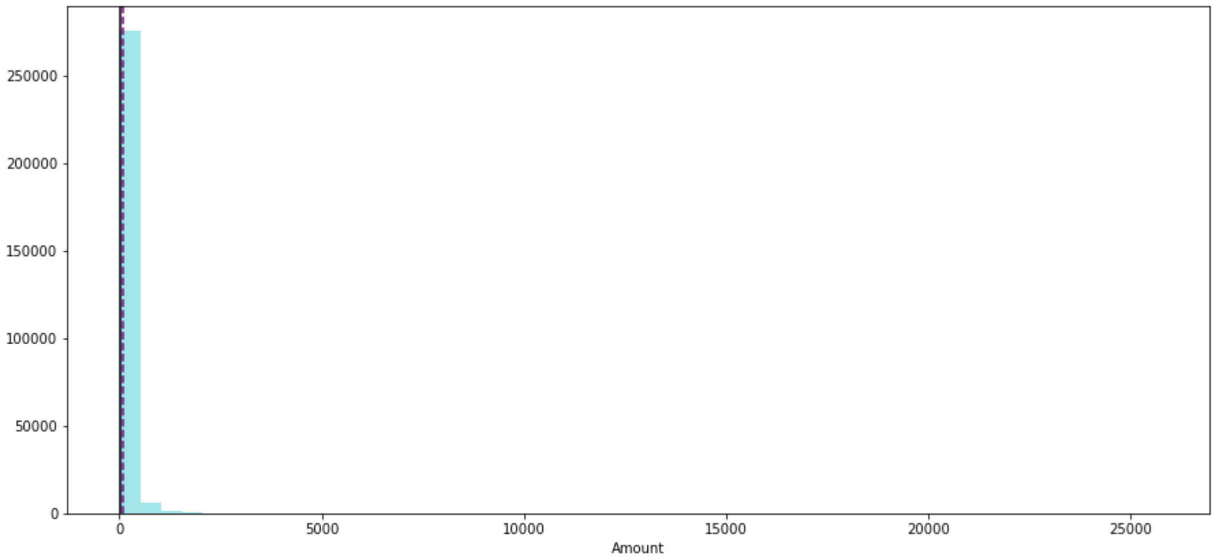


Time column is symmetrical at around 100,000 and at around 12,500

Observations on Amount

In [10]:
```
histogram_boxplot(data['Amount'])
```

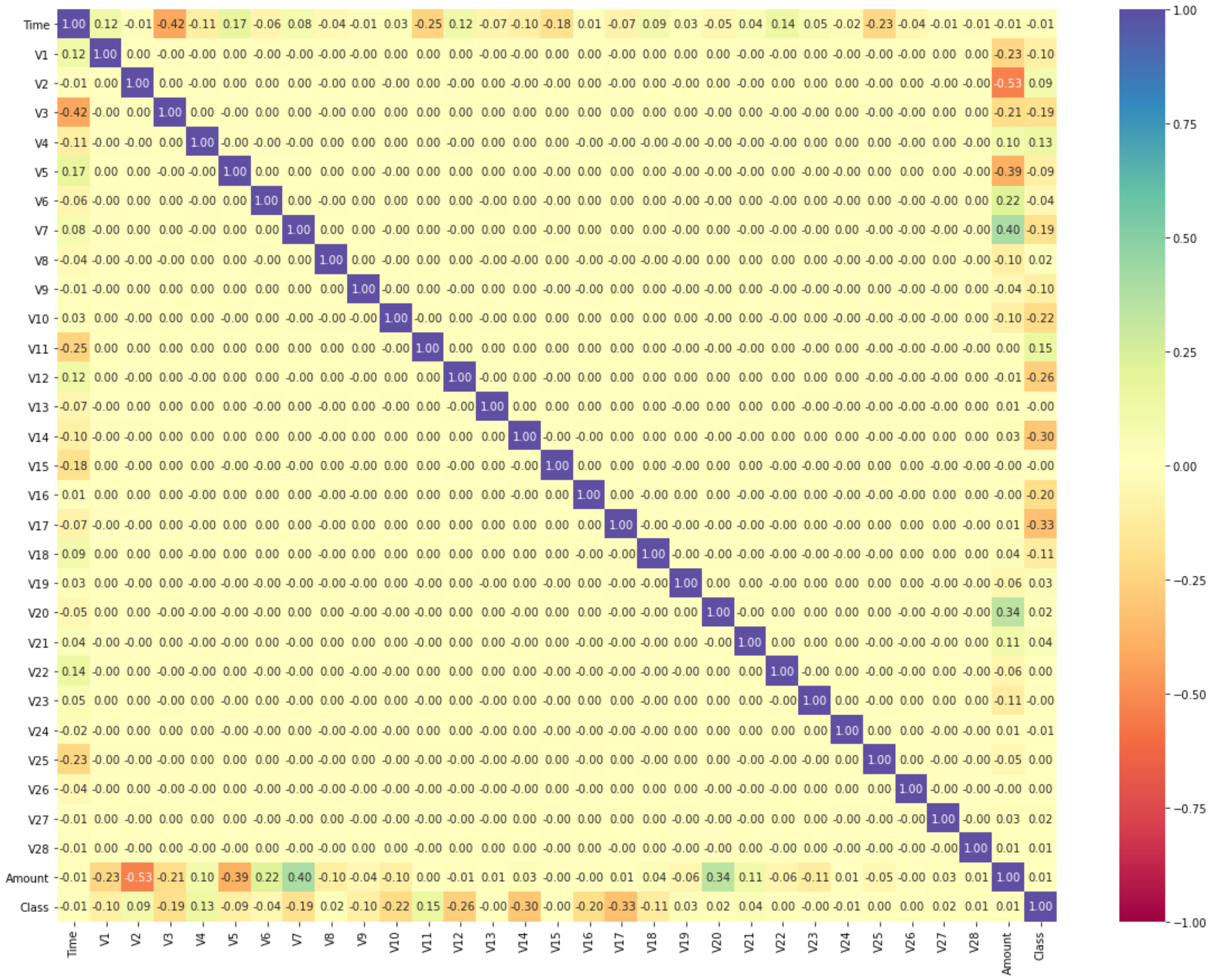




- A lot of outliers are present in the amount columns, these values are potential fraudulent activities
- The same can be done for other colums from V1 to V28

Bivariate Analysis

In [11]:
```
plt.figure(figsize=(20,15))
sns.heatmap(data.corr(),annot=True,vmin=-1,vmax=1,fmt='.2f',cmap='Spectral')
plt.show()
```

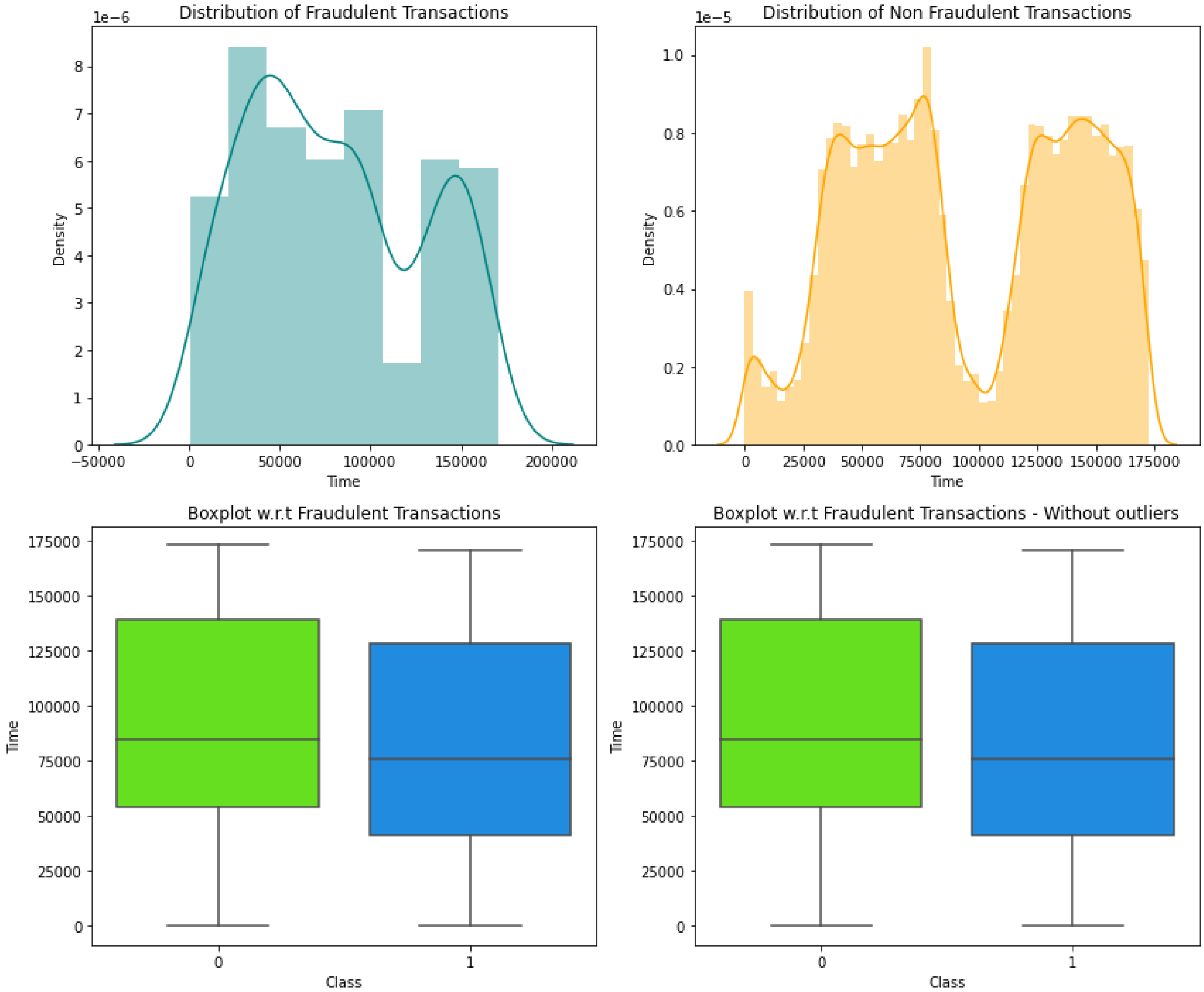


- The features are not very correlated with one another.

In [12]:
```
### Function to plot distributions and Boxplots of transactions
def plot(x,target='Class'):
    fig,axs = plt.subplots(2,2,figsize=(12,10))
    axs[0, 0].set_title('Distribution of Fraudulent Transactions')
    sns.distplot(data[(data[target] == 1)][x],ax=axs[0,0],color='teal')
    axs[0, 1].set_title("Distribution of Non Fraudulent Transactions")
    sns.distplot(data[(data[target] == 0)][x],ax=axs[0,1],color='orange')
    axs[1,0].set_title('Boxplot w.r.t Fraudulent Transactions')
    sns.boxplot(data[target],data[x],ax=axs[1,0],palette='gist_rainbow')
    axs[1,1].set_title('Boxplot w.r.t Fraudulent Transactions - Without outliers')
    sns.boxplot(data[target],data[x],ax=axs[1,1],showfliers=False,palette='gist_rain
    plt.tight_layout()
    plt.show()
```

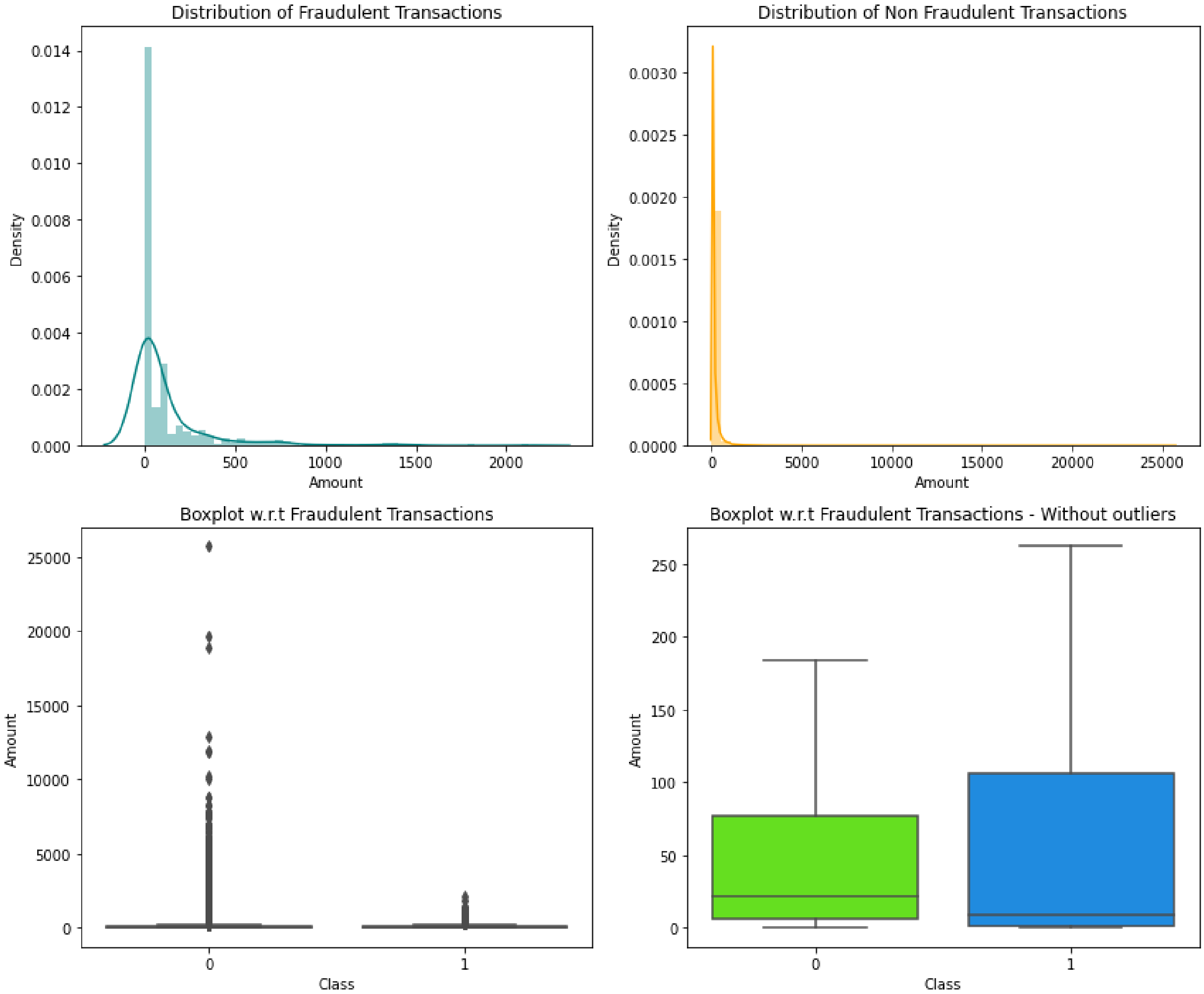
Distribution of time column

In [13]:
```
plot('Time')
```



Distribution of amount column

In [15]:
```
plot('Amount')
```



Building the model

Model evaluation criterion

Model can make wrong predictions as:

1. Predicting a transaction is fraudulent wehereas it is not
2. Predicting a transaction is not fraudulent whereas it is fraudulent

Which case is more important?

- Predicting a transaction is not fraudulent whereas it is fraudulent

How to reduce this loss i.e need to reduce False Negatives?

- The bank or institution would want `Recall` to be maximized, greater the Recall higher the chances of minimizing false negatives. Hence, the focus should be on increasing Recall or minimizing the false negatives.

Fist, let's create two functions to calculate different metrics and confusion matrix, so that we don't have to use the same code repeatedly for each model.

In [20]:

```
##  Function to calculate different metric scores of the model - Accuracy, Recall an
def get_metrics_score1(model,train,test,train_y,test_y,flag=True):
    '''
```

```
    model : classifier to predict values of X

    '''
    # defining an empty list to store train and test results

    score_list=[]

    pred_train = model.predict(train)
    pred_test = model.predict(test)

    pred_train = np.round(pred_train)
    pred_test = np.round(pred_test)

    train_acc = accuracy_score(pred_train,train_y)
    test_acc = accuracy_score(pred_test,test_y)

    train_recall = recall_score(train_y,pred_train)
    test_recall = recall_score(test_y,pred_test)

    train_precision = precision_score(train_y,pred_train)
    test_precision = precision_score(test_y,pred_test)

    score_list.extend((train_acc,test_acc,train_recall,test_recall,train_precision,t

   # If the flag is set to True then only the following print statements will be dis

    if flag == True:
        print("Accuracy on training set : ",accuracy_score(pred_train,train_y))
        print("Accuracy on test set : ",accuracy_score(pred_test,test_y))
        print("Recall on training set : ",recall_score(train_y,pred_train))
        print("Recall on test set : ",recall_score(test_y,pred_test))
        print("Precision on training set : ",precision_score(train_y,pred_train))
        print("Precision on test set : ",precision_score(test_y,pred_test))
        print("ROC-AUC Score on training set:",metrics.roc_auc_score(train_y,pred_tr
        print("ROC-AUC Score on test set:",metrics.roc_auc_score(test_y,pred_test))
    return score_list # returning the list with train and test scores
```

In [21]:

```
## Defining a function for better visualization of confusion matrix
def make_confusion_matrix(y_actual,y_predict,labels=[1, 0]):
    '''
    y_predict: prediction of class
    y_actual : ground truth
    '''
    cm=confusion_matrix( y_predict,y_actual, labels=[1, 0])
    data_cm = pd.DataFrame(cm, index = [i for i in ["1","0"]],
                  columns = [i for i in ['1','0']])
    group_counts = ["{0:0.0f}".format(value) for value in
                cm.flatten()]
    group_percentages = ["{0:.2%}".format(value) for value in
                         cm.flatten()/np.sum(cm)]
    labels = [f"{v1}\n{v2}" for v1, v2 in
              zip(group_counts,group_percentages)]
    labels = np.asarray(labels).reshape(2,2)
    plt.figure(figsize = (7,5))
    sns.heatmap(data_cm, annot=labels,fmt='')
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

Logistic Regression

Let's build model using Statsmodels

Before making the model, first let's check if our variables has multicollinearity

- There are different ways of detecting (or testing) multi-collinearity, one such way is Variation Inflation Factor.
- **Variance Inflation factor**: Variance inflation factors measure the inflation in the variances of the regression coefficients estimates due to collinearities that exist among the predictors. It is a measure of how much the variance of the estimated regression coefficient βk is “inflated”by the existence of correlation among the predictor variables in the model.
- General Rule of thumb: If VIF is 1 then there is no correlation among the kth predictor and the remaining predictor variables, and hence the variance of βk is not inflated at all. Whereas if VIF exceeds 5, we say there is moderate VIF and if it is 10 or exceeding 10, it shows signs of high multi-collinearity. But the purpose of the analysis should dictate which threshold to use.

In [14]:

```
X = data.drop(['Class'], axis=1)
Y = data[['Class']]

#Splitting data in train and test sets
X_train, X_test, y_train, y_test = train_test_split(X,Y, test_size=0.30, random_stat
```

In [15]:

```
# dataframe with numerical column only
num_feature_set = X.copy()
num_feature_set = add_constant(num_feature_set)
num_feature_set = num_feature_set.astype(float)
```

In [16]:

```
vif_series1 = pd.Series([variance_inflation_factor(num_feature_set.values,i) for i i
print('Series before feature selection: \n\n{}\n'.format(vif_series1))
```

```
Series before feature selection:

const     10.065370
Time       1.879865
V1         1.651908
V2         4.422390
V3         1.877342
V4         1.138061
V5         2.859316
V6         1.571530
V7         2.929040
V8         1.131633
V9         1.023894
V10        1.126333
V11        1.115328
V12        1.030070
V13        1.008474
V14        1.031854
V15        1.063421
V16        1.000448
V17        1.010701
V18        1.031048
V19        1.039643
V20        2.399180
V21        1.140305
```

```
V22        1.089101
V23        1.158142
V24        1.000806
V25        1.130801
V26        1.003359
V27        1.010105
V28        1.001433
Amount    12.116701
dtype: float64
```

- Amount and V2 seems to be highly correlated, so we will drop one of them depending on which has less effect on making predictions.

In [17]:

```
X_train, X_test, y_train, y_test = train_test_split(num_feature_set, Y, test_size=0.
```

In [22]:

```
logit = sm.Logit(y_train, X_train.astype(float))
lg = logit.fit()

print(lg.summary())

print('')
# Let's check model performances for this model
scores_LR = get_metrics_score1(lg,X_train,X_test,y_train,y_test,flag=True)
```

```
Optimization terminated successfully.
         Current function value: 0.003877
         Iterations 13
                           Logit Regression Results
==============================================================================
Dep. Variable:                  Class   No. Observations:               199364
Model:                          Logit   Df Residuals:                   199333
Method:                           MLE   Df Model:                           30
Date:                Mon, 28 Jun 2021   Pseudo R-squ.:                  0.7044
Time:                        10:18:04   Log-Likelihood:                -772.85
converged:                       True   LL-Null:                       -2614.8
Covariance Type:            nonrobust   LLR p-value:                     0.000
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -8.1197      0.286    -28.420      0.000      -8.680      -7.560
Time       -7.872e-06   2.75e-06     -2.862      0.004   -1.33e-05   -2.48e-06
V1             0.0929      0.051      1.837      0.066      -0.006       0.192
V2            -0.0250      0.066     -0.377      0.706      -0.155       0.105
V3            -0.0183      0.065     -0.279      0.780      -0.146       0.110
V4             0.6626      0.085      7.810      0.000       0.496       0.829
V5             0.1335      0.080      1.667      0.096      -0.024       0.291
V6            -0.1124      0.092     -1.221      0.222      -0.293       0.068
V7            -0.1056      0.077     -1.368      0.171      -0.257       0.046
V8            -0.1442      0.039     -3.742      0.000      -0.220      -0.069
V9            -0.5118      0.132     -3.879      0.000      -0.770      -0.253
V10           -0.8191      0.107     -7.661      0.000      -1.029      -0.610
V11           -0.1302      0.100     -1.307      0.191      -0.325       0.065
V12            0.2746      0.118      2.333      0.020       0.044       0.505
V13           -0.4406      0.105     -4.189      0.000      -0.647      -0.234
V14           -0.6528      0.081     -8.062      0.000      -0.812      -0.494
V15           -0.0603      0.103     -0.583      0.560      -0.263       0.142
V16           -0.1957      0.144     -1.358      0.175      -0.478       0.087
V17           -0.0724      0.082     -0.882      0.378      -0.233       0.088
V18            0.0078      0.149      0.052      0.958      -0.285       0.301
V19            0.1119      0.113      0.991      0.322      -0.109       0.333
V20           -0.4982      0.095     -5.223      0.000      -0.685      -0.311
V21            0.4589      0.075      6.106      0.000       0.312       0.606
V22            0.7242      0.164      4.416      0.000       0.403       1.046
V23           -0.1048      0.071     -1.485      0.138      -0.243       0.034
```

```
V24          -0.0259      0.179     -0.145      0.885      -0.376       0.325
V25          -0.1586      0.156     -1.019      0.308      -0.464       0.146
V26           0.1407      0.222      0.634      0.526      -0.294       0.576
V27          -0.9115      0.133     -6.847      0.000      -1.172      -0.651
V28          -0.3228      0.100     -3.214      0.001      -0.520      -0.126
Amount        0.0010      0.000      2.202      0.028       0.000       0.002
==============================================================================

Possibly complete quasi-separation: A fraction 0.34 of observations can be
perfectly predicted. This might indicate that there is complete
quasi-separation. In this case some parameters will not be identified.

Accuracy on training set :  0.9992676711943982
Accuracy on test set :  0.9991573329588147
Recall on training set :  0.6666666666666666
Recall on test set :  0.5777777777777777
Precision on training set :  0.8981132075471698
Precision on test set :  0.8387096774193549
ROC-AUC Score on training set: 0.8332654965235728
ROC-AUC Score on test set: 0.788800972163611
```

In [24]:

```
X_train1 = X_train.drop('Amount', axis = 1)
X_test1 = X_test.drop('Amount', axis = 1)

logit1 = sm.Logit(y_train, X_train1.astype(float))
lg1 = logit1.fit()
```

```
Optimization terminated successfully.
         Current function value: 0.003891
         Iterations 13
```

In [25]:

```
# Let's check model performances for this model
scores_LR = get_metrics_score1(lg1,X_train1,X_test1,y_train,y_test,flag=True)
```

```
Accuracy on training set :  0.9992526233422283
Accuracy on test set :  0.9991807403766253
Recall on training set :  0.6582633053221288
Recall on test set :  0.5851851851851851
Precision on training set :  0.8969465648854962
Precision on test set :  0.8494623655913979
ROC-AUC Score on training set: 0.829063815851304
ROC-AUC Score on test set: 0.7925105369823332
```

In [27]:

```
X_train2 = X_train.drop('V2', axis = 1)
X_test2 = X_test.drop('V2', axis = 1)

logit2 = sm.Logit(y_train, X_train2)
lg2 = logit2.fit()
```

```
Optimization terminated successfully.
         Current function value: 0.003877
         Iterations 13
```

In [28]:

```
# Let's check model performances for this model
scores_LR = get_metrics_score1(lg2,X_train2,X_test2,y_train,y_test,flag=True)
```

```
Accuracy on training set :  0.999247607391505
Accuracy on test set :  0.9991573329588147
Recall on training set :  0.6554621848739496
Recall on test set :  0.5777777777777777
Precision on training set :  0.896551724137931
Precision on test set :  0.8387096774193549
ROC-AUC Score on training set: 0.8276632556272142
ROC-AUC Score on test set: 0.788800972163611
```

The models performance improved when the 'Amount' column was dropped for lg1. So we will use lg1 as base model.

Let's check VIF score again

In [23]:

```
num_feature_set = num_feature_set.drop(['Amount'], axis = 1)
vif_series1 = pd.Series([variance_inflation_factor(num_feature_set.values,i) for i i
print('Series before feature selection: \n\n{}\n'.format(vif_series1))
```

```
Series before feature selection:

const     8.493197
Time      1.879717
V1        1.025906
V2        1.000211
V3        1.330979
V4        1.020827
V5        1.056305
V6        1.007464
V7        1.013490
V8        1.002566
V9        1.000141
V10       1.001762
V11       1.115321
V12       1.029065
V13       1.008164
V14       1.018333
V15       1.063262
V16       1.000266
V17       1.010099
V18       1.015374
V19       1.001578
V20       1.004863
V21       1.003762
V22       1.039010
V23       1.004916
V24       1.000492
V25       1.102120
V26       1.003223
V27       1.000050
V28       1.000167
dtype: float64
```

The multicolinearity from the columns have been removed and now have VIFs less than 5

Metrics of final logistic model (lg1)

In [30]:

```
print(lg1.summary())

print('')

# Model performance

scores_LR = get_metrics_score1(lg1,X_train1.astype(float),X_test1.astype(float),y_tr
```

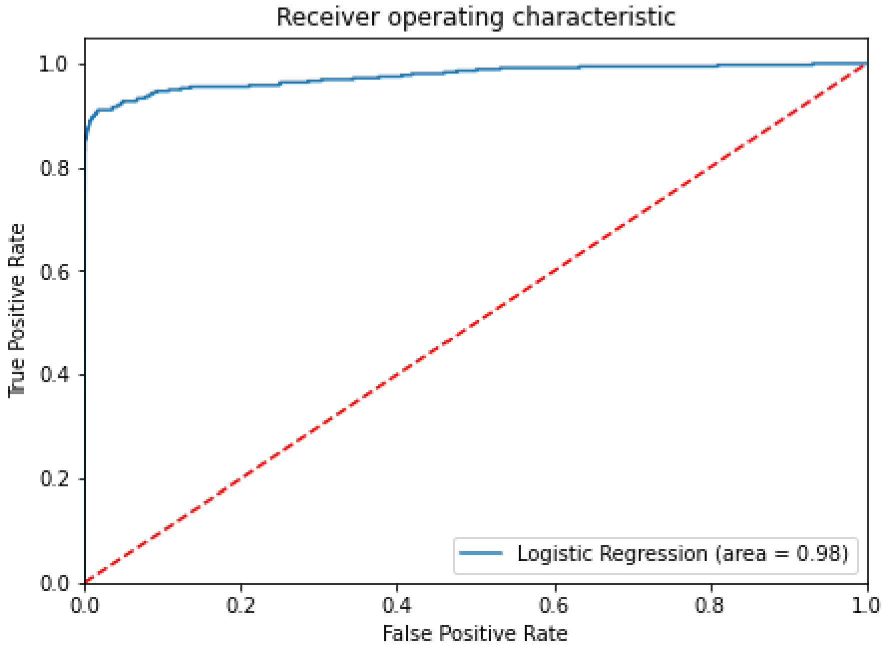
```
                           Logit Regression Results
==============================================================================
Dep. Variable:                  Class   No. Observations:               199364
Model:                          Logit   Df Residuals:                   199334
Method:                           MLE   Df Model:                           29
Date:                Mon, 28 Jun 2021   Pseudo R-squ.:                  0.7034
Time:                        15:58:20   Log-Likelihood:                -775.63
converged:                       True   LL-Null:                       -2614.8
```

```
Covariance Type:            nonrobust   LLR p-value:                     0.000
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -8.0214      0.283    -28.349      0.000      -8.576      -7.467
Time       -7.777e-06   2.75e-06     -2.825      0.005   -1.32e-05   -2.38e-06
V1             0.0698      0.050      1.392      0.164      -0.028       0.168
V2            -0.0812      0.057     -1.430      0.153      -0.193       0.030
V3            -0.0363      0.064     -0.564      0.572      -0.162       0.090
V4             0.6732      0.085      7.914      0.000       0.506       0.840
V5             0.0765      0.076      1.009      0.313      -0.072       0.225
V6            -0.0802      0.091     -0.877      0.380      -0.259       0.099
V7            -0.0275      0.069     -0.400      0.689      -0.162       0.107
V8            -0.1615      0.037     -4.390      0.000      -0.234      -0.089
V9            -0.5037      0.135     -3.729      0.000      -0.768      -0.239
V10           -0.8243      0.112     -7.356      0.000      -1.044      -0.605
V11           -0.1291      0.100     -1.292      0.196      -0.325       0.067
V12            0.2610      0.118      2.207      0.027       0.029       0.493
V13           -0.4434      0.105     -4.217      0.000      -0.649      -0.237
V14           -0.6430      0.082     -7.849      0.000      -0.804      -0.482
V15           -0.0660      0.104     -0.637      0.524      -0.269       0.137
V16           -0.2135      0.144     -1.481      0.139      -0.496       0.069
V17           -0.0739      0.083     -0.894      0.371      -0.236       0.088
V18            0.0355      0.149      0.238      0.812      -0.257       0.328
V19            0.0889      0.113      0.788      0.431      -0.132       0.310
V20           -0.3209      0.079     -4.045      0.000      -0.476      -0.165
V21            0.5049      0.075      6.721      0.000       0.358       0.652
V22            0.7074      0.164      4.304      0.000       0.385       1.030
V23           -0.1331      0.084     -1.586      0.113      -0.298       0.031
V24           -0.0274      0.179     -0.153      0.879      -0.379       0.324
V25           -0.1901      0.157     -1.211      0.226      -0.498       0.117
V26            0.1280      0.224      0.571      0.568      -0.311       0.567
V27           -0.8812      0.163     -5.399      0.000      -1.201      -0.561
V28           -0.2386      0.112     -2.123      0.034      -0.459      -0.018
==============================================================================

Possibly complete quasi-separation: A fraction 0.34 of observations can be
perfectly predicted. This might indicate that there is complete
quasi-separation. In this case some parameters will not be identified.

Accuracy on training set :  0.9992526233422283
Accuracy on test set :  0.9991807403766253
Recall on training set :  0.6582633053221288
Recall on test set :  0.5851851851851851
Precision on training set :  0.8969465648854962
Precision on test set :  0.8494623655913979
ROC-AUC Score on training set: 0.829063815851304
ROC-AUC Score on test set: 0.7925105369823332
```

- ROC-AUC on training set

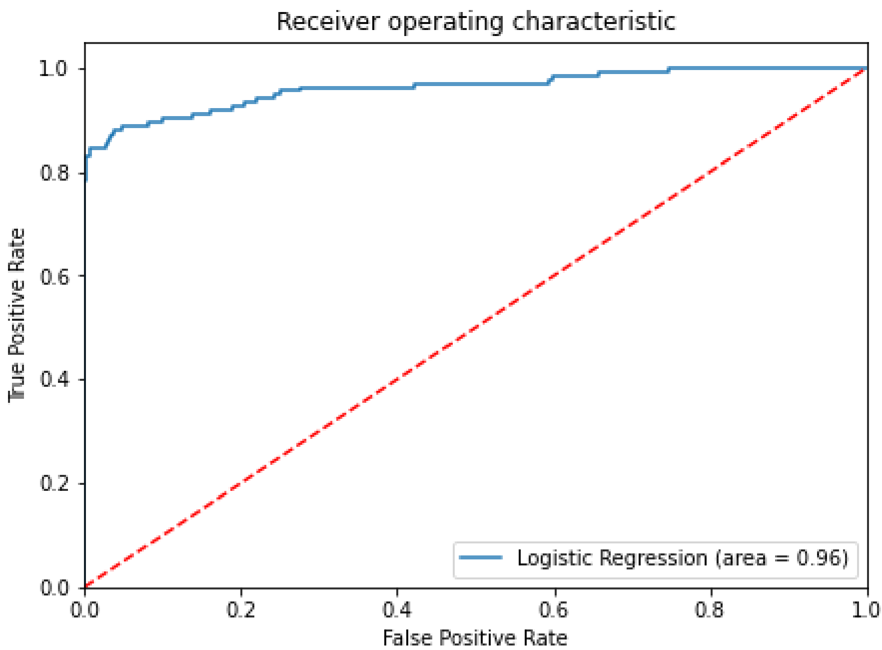
In [68]:

```
logit_roc_auc_train = roc_auc_score(y_train, lg1.predict(X_train1))
fpr, tpr, thresholds = roc_curve(y_train, lg1.predict(X_train1))
plt.figure(figsize=(7,5))
plt.plot(fpr, tpr, label='Logistic Regression (area = %0.2f)' % logit_roc_auc_train)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic')
plt.legend(loc="lower right")
plt.show()
```



- ROC-AUC on test set

In [69]:
```
logit_roc_auc_test = roc_auc_score(y_test, lg1.predict(X_test1))
fpr, tpr, thresholds = roc_curve(y_test, lg1.predict(X_test1))
plt.figure(figsize=(7,5))
plt.plot(fpr, tpr, label='Logistic Regression (area = %0.2f)' % logit_roc_auc_test)
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic')
plt.legend(loc="lower right")
plt.show()
```



Coefficient interpretations

- Coefficients of V1, V4, V5, V12, V18, V19, V21, V22, V26 are positive, so an increase in the value of these columns increases the chances that a transaction is fraudulent.
- Coefficients for V2, V3, V6, V7, V8, V9, V10, V11, V13, V14, V15, V16, V17, V20, V23, V24, V25, V27, V28, and time are negative meaning that an increase in its values decreases the chances of a transacction being fraudulent.
- V10, V14, V27, V22, V21 are the most important variables from the logistic regression model.
- Similarly we can interpret for other attributes.

- Logistic Regression model is giving a generalized performance on training and test set.
- ROC-AUC score of 0.96 on training and test set is quite good.

Model Performance Improvement

- Let's see if the recall score can be improved further, by changing the model threshold using AUC-ROC Curve.

Optimal threshold using AUC-ROC curve

In [70]:

```
# Optimal threshold as per AUC-ROC curve
# The optimal cut off would be where tpr is high and fpr is low
fpr, tpr, thresholds = metrics.roc_curve(y_test, lg1.predict(X_test1.astype(float)))

optimal_idx = np.argmax(tpr - fpr)
optimal_threshold = thresholds[optimal_idx]
print(optimal_threshold)
```

```
0.0017854847952167836
```

In [71]:

```
# Model prediction with optimal threshold
pred_train_opt = (lg1.predict(X_train1.astype(float)) > optimal_threshold).astype(in
pred_test_opt = (lg1.predict(X_test1.astype(float)) > optimal_threshold).astype(int)

print('Accuracy on train data:',accuracy_score(y_train, pred_train_opt) )
print('Accuracy on test data:',accuracy_score(y_test, pred_test_opt))

print('Recall on train data:',recall_score(y_train, pred_train_opt) )
print('Recall on test data:',recall_score(y_test, pred_test_opt))

print('Precision on train data:',precision_score(y_train, pred_train_opt) )
print('Precision on test data:',precision_score(y_test, pred_test_opt))

print('ROC-AUC Score on train data:',roc_auc_score(y_train, pred_train_opt) )
print('ROC-AUC Score on test data:',roc_auc_score(y_test, pred_test_opt))
```

```
Accuracy on train data: 0.9618637266507494
Accuracy on test data: 0.9616586496260665
Recall on train data: 0.9159663865546218
Recall on test data: 0.8740740740740741
Precision on train data: 0.041392405063291136
Precision on test data: 0.034942256440627775
ROC-AUC Score on train data: 0.9389562243767194
ROC-AUC Score on test data: 0.9179356631916767
```

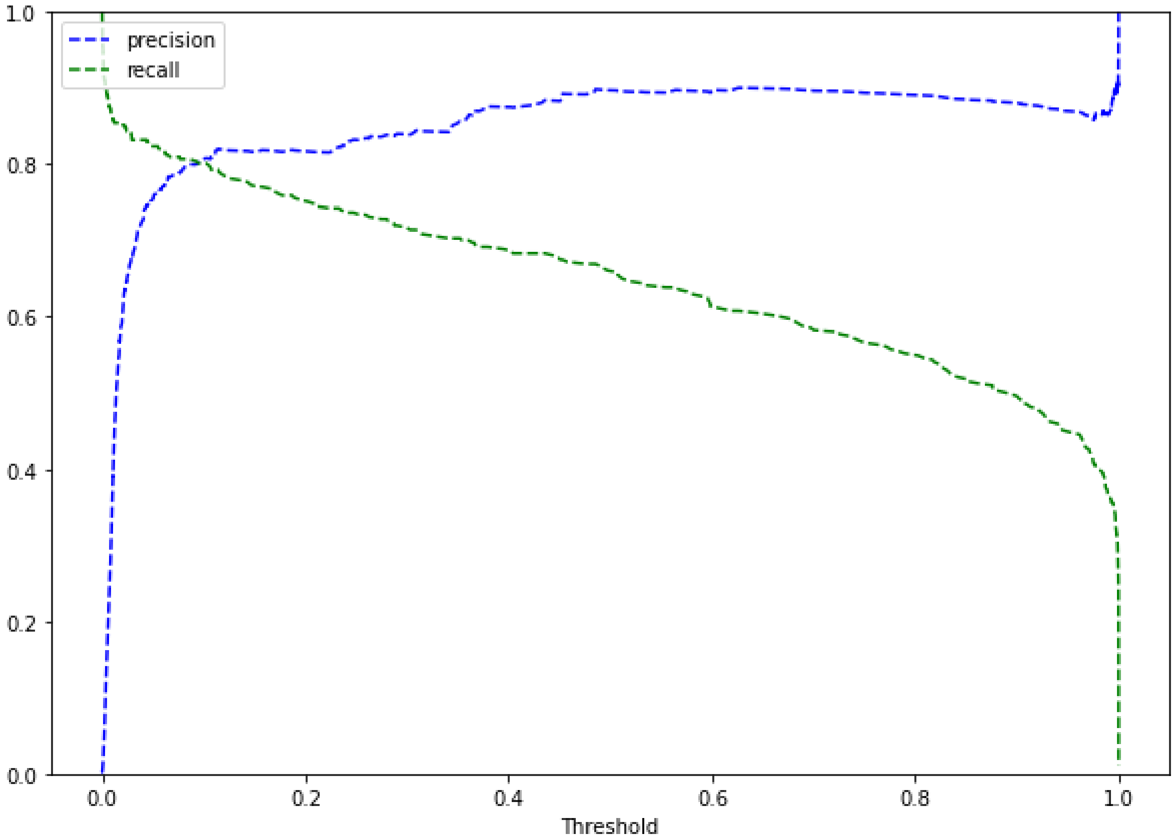
- Recall increased from on the test set to 0.874, as compared to the previous model which was 0.585.

- As we will decrease the threshold value, Recall will keep on increasing but the Precision will decrease, but that's not right because it will lead to loss of resources, we need to choose an optimal balance between recall and precision.
- Area under the curve is has decreased as compared to the initial model.

Let's use Precision-Recall curve and see if we can find a better threshold

In [72]:

```
y_scores=lg1.predict(X_train1)
prec, rec, tre = precision_recall_curve(y_train, y_scores,)

def plot_prec_recall_vs_tresh(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], 'b--', label='precision')
    plt.plot(thresholds, recalls[:-1], 'g--', label = 'recall')
    plt.xlabel('Threshold')
    plt.legend(loc='upper left')
    plt.ylim([0,1])
plt.figure(figsize=(10,7))
plot_prec_recall_vs_tresh(prec, rec, tre)
plt.show()
```



- At 0.085 threshold we get a higher recall and a good precision.

In [73]:

```
optimal_threshold = 0.085

# Model prediction with optimal threshold
pred_train_opt = (lg1.predict(X_train1.astype(float)) > optimal_threshold).astype(in
pred_test_opt = (lg1.predict(X_test1.astype(float)) > optimal_threshold).astype(int)

#Model performance with optimal threhold

print('Accuracy on train data:',accuracy_score(y_train, pred_train_opt) )
print('Accuracy on test data:',accuracy_score(y_test, pred_test_opt))
```

```
print('Recall on train data:',recall_score(y_train, pred_train_opt))
print('Recall on test data:',recall_score(y_test, pred_test_opt))

print('Precision on train data:',precision_score(y_train, pred_train_opt) )
print('Precision on test data:',precision_score(y_test, pred_test_opt))

print('ROC-AUC Score on train data:',roc_auc_score(y_train, pred_train_opt) )
print('ROC-AUC Score on test data:',roc_auc_score(y_test, pred_test_opt))
```

```
Accuracy on train data: 0.9992927509480147
Accuracy on test data: 0.9992509626300574
Recall on train data: 0.8067226890756303
Recall on test data: 0.7407407407407407
Precision on train data: 0.8
Precision on test data: 0.7751937984496124
ROC-AUC Score on train data: 0.9031804463784538
ROC-AUC Score on test data: 0.8702003980348332
```

- Model is performing well on training and test set.
- Model has given a balanced performance, if the company wishes to maintain a balance between recall and precision this model can be used.
- Area under the curve has decreased as compared to the initial model but the performance is generalized on training and test set.

Decision Trees

Split data

In [64]:
```
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=.30, random_stat
```

Build Decision Tree Model

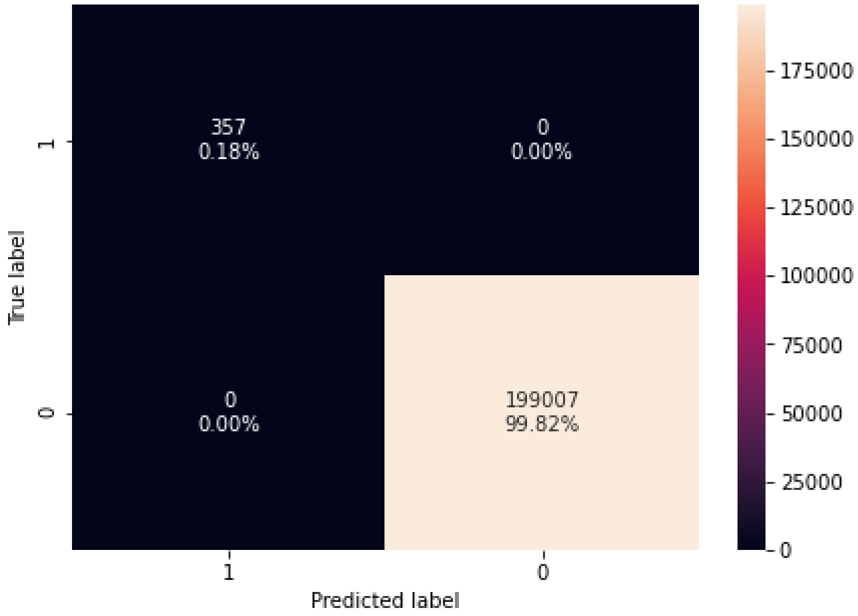
We will build our model using the DecisionTreeClassifier function. Using default 'gini' criteria to split. Other option include 'entropy'.

In [65]:
```
dTree = DecisionTreeClassifier(criterion = 'gini', random_state=1)
dTree.fit(X_train, y_train)
```

Out[65]:
```
DecisionTreeClassifier(random_state=1)
```

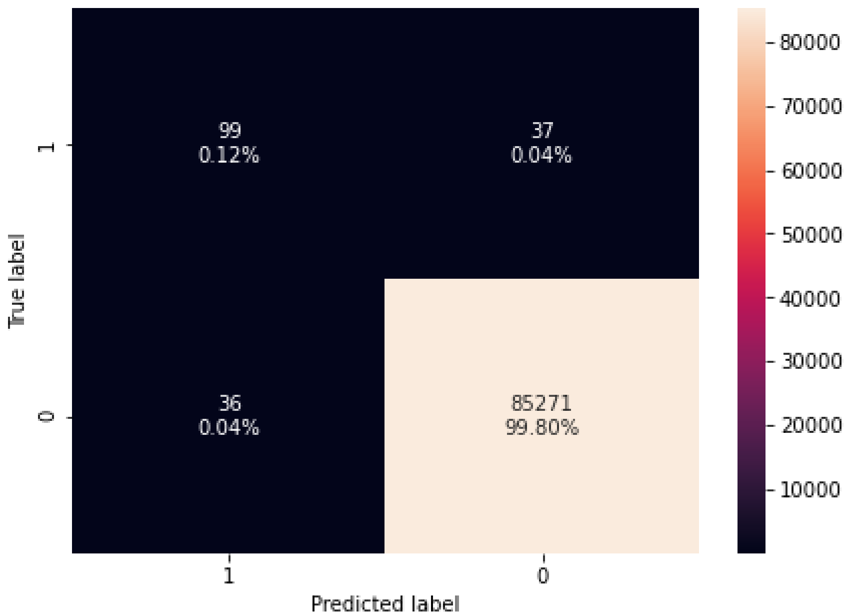
In [66]:
```
prob_train = dTree.predict_proba(X_train)
pred_train = dTree.predict(X_train)

prob_test = dTree.predict_proba(X_test)
pred_test = dTree.predict(X_test)
```

In [67]:
```
# Let us make confusion matrix on train set
make_confusion_matrix(y_train,pred_train)
```



In [68]:

```
# let us make confusion matrix on test set
make_confusion_matrix(y_test,pred_test)
```



In [69]:

```
# Let's check model performances for this model
score_DT = get_metrics_score1(dTree, X_train, X_test, y_train, y_test, flag=True)
```

```
Accuracy on training set :  1.0
Accuracy on test set :  0.9991456292499094
Recall on training set :  1.0
Recall on test set :  0.7333333333333333
Precision on training set :  1.0
Precision on test set :  0.7279411764705882
```

- Model has performed very well on training and test set.
- There's slight overfitting in terms of recall and recall, let's see if pruning methods can help in improving the metrics.
- Area under the curve is also 0.72794 is not so good.

Visualizing the Decision Tree

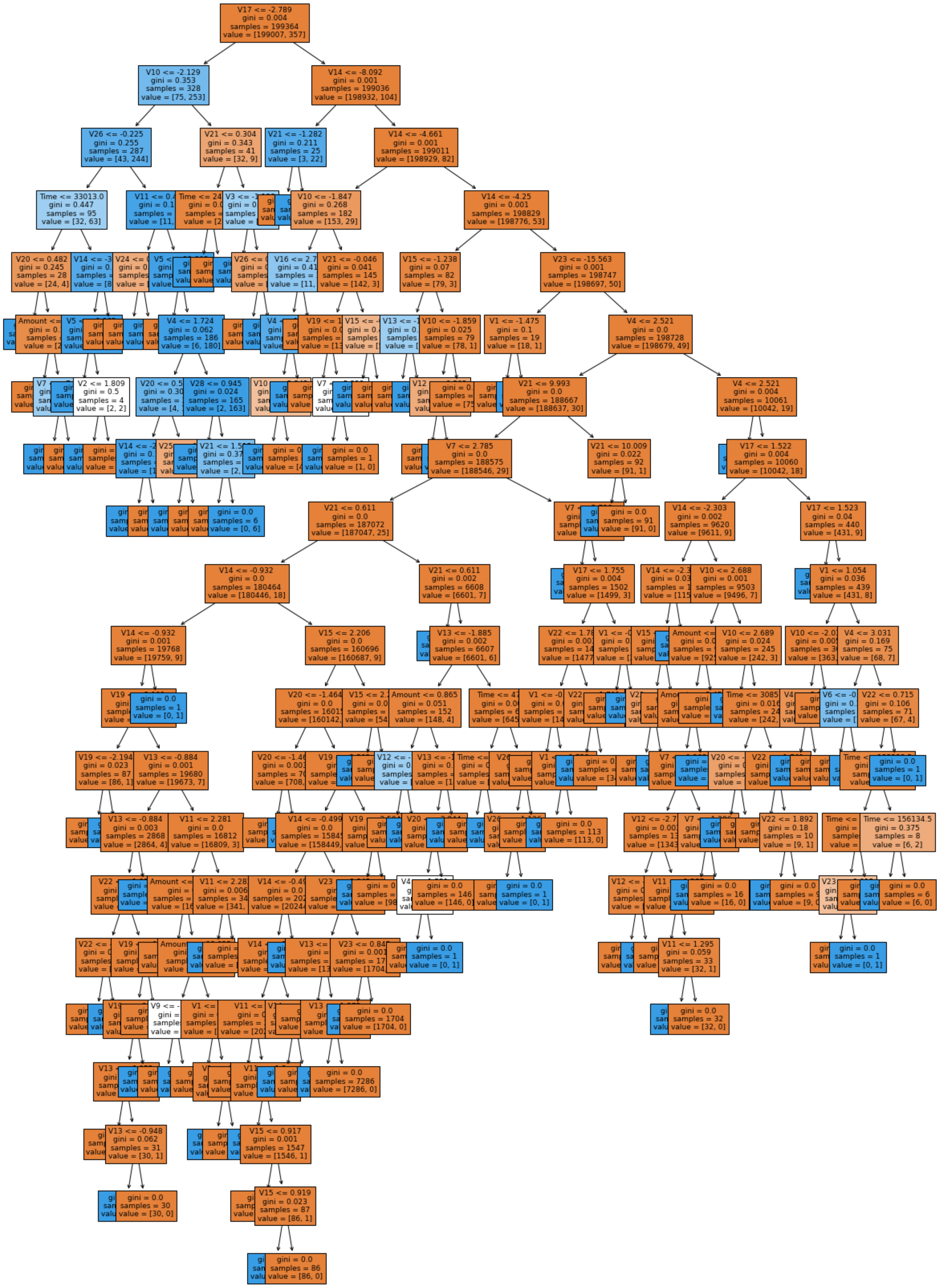
In [70]:

```
feature_names = list(X.columns)
print(feature_names)
```

```
['Time', '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', 'V27', 'V28', 'Amount']
```

In [71]:

```
plt.figure(figsize=(20,30))
out = tree.plot_tree(dTree,feature_names=feature_names,filled=True,fontsize=9,node_i
#below code will add arrows to the decision tree split if they are missing
for o in out:
     arrow = o.arrow_patch
     if arrow is not None:
        arrow.set_edgecolor('black')
        arrow.set_linewidth(1)
plt.show()
```



In [72]:

```
# Text report showing the rules of a decision tree -

print(tree.export_text(dTree,feature_names=feature_names,show_weights=True))
```

```
|--- V17 <= -2.79
|   |--- V10 <= -2.13
|   |   |--- V26 <= -0.22
|   |   |   |--- Time <= 33013.00
|   |   |   |   |--- V20 <= 0.48
```

```
|   |   |   |   |   |--- weights: [0.00, 2.00] class: 1
|   |   |   |   |--- V20 >  0.48
|   |   |   |   |   |--- Amount <= 94.99
|   |   |   |   |   |   |--- weights: [23.00, 0.00] class: 0
|   |   |   |   |   |--- Amount >  94.99
|   |   |   |   |   |   |--- V7 <= -7.56
|   |   |   |   |   |   |   |--- weights: [0.00, 2.00] class: 1
|   |   |   |   |   |   |--- V7 >  -7.56
|   |   |   |   |   |   |   |--- weights: [1.00, 0.00] class: 0
|   |   |   |--- Time >  33013.00
|   |   |   |   |--- V14 <= -3.09
|   |   |   |   |   |--- V5 <= -0.06
|   |   |   |   |   |   |--- weights: [0.00, 57.00] class: 1
|   |   |   |   |   |--- V5 >  -0.06
|   |   |   |   |   |   |--- V2 <= 1.81
|   |   |   |   |   |   |   |--- weights: [0.00, 2.00] class: 1
|   |   |   |   |   |   |--- V2 >  1.81
|   |   |   |   |   |   |   |--- weights: [2.00, 0.00] class: 0
|   |   |   |   |--- V14 >  -3.09
|   |   |   |   |   |--- weights: [6.00, 0.00] class: 0
|   |   |--- V26 >  -0.22
|   |   |   |--- V11 <= 0.47
|   |   |   |   |--- V24 <= 0.12
|   |   |   |   |   |--- weights: [3.00, 0.00] class: 0
|   |   |   |   |--- V24 >  0.12
|   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |--- V11 >  0.47
|   |   |   |   |--- V5 <= -22.81
|   |   |   |   |   |--- weights: [2.00, 0.00] class: 0
|   |   |   |   |--- V5 >  -22.81
|   |   |   |   |   |--- V4 <= 1.72
|   |   |   |   |   |   |--- V20 <= 0.54
|   |   |   |   |   |   |   |--- V14 <= -2.66
|   |   |   |   |   |   |   |   |--- weights: [0.00, 16.00] class: 1
|   |   |   |   |   |   |   |--- V14 >  -2.66
|   |   |   |   |   |   |   |   |--- weights: [1.00, 0.00] class: 0
|   |   |   |   |   |   |--- V20 >  0.54
|   |   |   |   |   |   |   |--- V25 <= -0.11
|   |   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |   |--- V25 >  -0.11
|   |   |   |   |   |   |   |   |--- weights: [3.00, 0.00] class: 0
|   |   |   |   |   |--- V4 >  1.72
|   |   |   |   |   |   |--- V28 <= 0.95
|   |   |   |   |   |   |   |--- weights: [0.00, 157.00] class: 1
|   |   |   |   |   |   |--- V28 >  0.95
|   |   |   |   |   |   |   |--- V21 <= 1.51
|   |   |   |   |   |   |   |   |--- weights: [2.00, 0.00] class: 0
|   |   |   |   |   |   |   |--- V21 >  1.51
|   |   |   |   |   |   |   |   |--- weights: [0.00, 6.00] class: 1
|   |--- V10 >  -2.13
|   |   |--- V21 <= 0.30
|   |   |   |--- Time <= 24961.50
|   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |--- Time >  24961.50
|   |   |   |   |--- weights: [28.00, 0.00] class: 0
|   |   |--- V21 >  0.30
|   |   |   |--- V3 <= -1.14
|   |   |   |   |--- weights: [0.00, 7.00] class: 1
|   |   |   |--- V3 >  -1.14
|   |   |   |   |--- V26 <= 0.72
|   |   |   |   |   |--- weights: [4.00, 0.00] class: 0
|   |   |   |   |--- V26 >  0.72
|   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|--- V17 >  -2.79
|   |--- V14 <= -8.09
|   |   |--- V21 <= -1.28
|   |   |   |--- weights: [3.00, 0.00] class: 0
|   |   |--- V21 >  -1.28
|   |   |   |--- weights: [0.00, 22.00] class: 1
```

```
|   |--- V14 >  -8.09
|   |   |--- V14 <= -4.66
|   |   |   |--- V10 <= -1.85
|   |   |   |   |--- V16 <= 2.75
|   |   |   |   |   |--- V4 <= 1.85
|   |   |   |   |   |   |--- V10 <= -2.54
|   |   |   |   |   |   |   |--- weights: [0.00, 2.00] class: 1
|   |   |   |   |   |   |--- V10 >  -2.54
|   |   |   |   |   |   |   |--- weights: [4.00, 0.00] class: 0
|   |   |   |   |   |--- V4 >  1.85
|   |   |   |   |   |   |--- weights: [0.00, 24.00] class: 1
|   |   |   |   |--- V16 >  2.75
|   |   |   |   |   |--- weights: [7.00, 0.00] class: 0
|   |   |   |--- V10 >  -1.85
|   |   |   |   |--- V21 <= -0.05
|   |   |   |   |   |--- V19 <= 1.44
|   |   |   |   |   |   |--- weights: [136.00, 0.00] class: 0
|   |   |   |   |   |--- V19 >  1.44
|   |   |   |   |   |   |--- V7 <= 5.32
|   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |--- V7 >  5.32
|   |   |   |   |   |   |   |--- weights: [1.00, 0.00] class: 0
|   |   |   |   |--- V21 >  -0.05
|   |   |   |   |   |--- V15 <= -0.09
|   |   |   |   |   |   |--- weights: [0.00, 2.00] class: 1
|   |   |   |   |   |--- V15 >  -0.09
|   |   |   |   |   |   |--- weights: [5.00, 0.00] class: 0
|   |   |--- V14 >  -4.66
|   |   |   |--- V14 <= -4.25
|   |   |   |   |--- V15 <= -1.24
|   |   |   |   |   |--- V13 <= -1.77
|   |   |   |   |   |   |--- weights: [1.00, 0.00] class: 0
|   |   |   |   |   |--- V13 >  -1.77
|   |   |   |   |   |   |--- weights: [0.00, 2.00] class: 1
|   |   |   |   |--- V15 >  -1.24
|   |   |   |   |   |--- V10 <= -1.86
|   |   |   |   |   |   |--- V12 <= -0.50
|   |   |   |   |   |   |   |--- weights: [3.00, 0.00] class: 0
|   |   |   |   |   |   |--- V12 >  -0.50
|   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |--- V10 >  -1.86
|   |   |   |   |   |   |--- weights: [75.00, 0.00] class: 0
|   |   |   |--- V14 >  -4.25
|   |   |   |   |--- V23 <= -15.56
|   |   |   |   |   |--- V1 <= -1.48
|   |   |   |   |   |   |--- weights: [18.00, 0.00] class: 0
|   |   |   |   |   |--- V1 >  -1.48
|   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |--- V23 >  -15.56
|   |   |   |   |   |--- V4 <= 2.52
|   |   |   |   |   |   |--- V21 <= 9.99
|   |   |   |   |   |   |   |--- V7 <= 2.79
|   |   |   |   |   |   |   |   |--- V21 <= 0.61
|   |   |   |   |   |   |   |   |   |--- V14 <= -0.93
|   |   |   |   |   |   |   |   |   |   |--- V14 <= -0.93
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 9
|   |   |   |   |   |   |   |   |   |   |--- V14 >  -0.93
|   |   |   |   |   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |   |   |   |--- V14 >  -0.93
|   |   |   |   |   |   |   |   |   |   |--- V15 <= 2.21
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 10
|   |   |   |   |   |   |   |   |   |   |--- V15 >  2.21
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 2
|   |   |   |   |   |   |   |   |--- V21 >  0.61
|   |   |   |   |   |   |   |   |   |--- V21 <= 0.61
|   |   |   |   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |   |   |   |--- V21 >  0.61
|   |   |   |   |   |   |   |   |   |   |--- V13 <= -1.88
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 5
```

```
|   |   |   |   |   |   |   |   |   |   |--- V13 >  -1.88
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 4
|   |   |   |   |   |   |   |--- V7 >  2.79
|   |   |   |   |   |   |   |   |--- V7 <= 2.79
|   |   |   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |   |   |--- V7 >  2.79
|   |   |   |   |   |   |   |   |   |--- V17 <= 1.75
|   |   |   |   |   |   |   |   |   |   |--- V22 <= 1.79
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 3
|   |   |   |   |   |   |   |   |   |   |--- V22 >  1.79
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 2
|   |   |   |   |   |   |   |   |   |--- V17 >  1.75
|   |   |   |   |   |   |   |   |   |   |--- V1 <= -0.37
|   |   |   |   |   |   |   |   |   |   |   |--- weights: [22.00, 0.00] class: 0
|   |   |   |   |   |   |   |   |   |   |--- V1 >  -0.37
|   |   |   |   |   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |--- V21 >  9.99
|   |   |   |   |   |   |   |--- V21 <= 10.01
|   |   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |   |--- V21 >  10.01
|   |   |   |   |   |   |   |   |--- weights: [91.00, 0.00] class: 0
|   |   |   |   |   |--- V4 >  2.52
|   |   |   |   |   |   |--- V4 <= 2.52
|   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |--- V4 >  2.52
|   |   |   |   |   |   |   |--- V17 <= 1.52
|   |   |   |   |   |   |   |   |--- V14 <= -2.30
|   |   |   |   |   |   |   |   |   |--- V14 <= -2.32
|   |   |   |   |   |   |   |   |   |   |--- V15 <= -1.91
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 2
|   |   |   |   |   |   |   |   |   |   |--- V15 >  -1.91
|   |   |   |   |   |   |   |   |   |   |   |--- weights: [110.00, 0.00] class: 0
|   |   |   |   |   |   |   |   |   |--- V14 >  -2.32
|   |   |   |   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |   |   |--- V14 >  -2.30
|   |   |   |   |   |   |   |   |   |--- V10 <= 2.69
|   |   |   |   |   |   |   |   |   |   |--- Amount <= 2.48
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 6
|   |   |   |   |   |   |   |   |   |   |--- Amount >  2.48
|   |   |   |   |   |   |   |   |   |   |   |--- weights: [7895.00, 0.00] class: 0
|   |   |   |   |   |   |   |   |   |--- V10 >  2.69
|   |   |   |   |   |   |   |   |   |   |--- V10 <= 2.69
|   |   |   |   |   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |   |   |   |   |--- V10 >  2.69
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 4
|   |   |   |   |   |   |   |--- V17 >  1.52
|   |   |   |   |   |   |   |   |--- V17 <= 1.52
|   |   |   |   |   |   |   |   |   |--- weights: [0.00, 1.00] class: 1
|   |   |   |   |   |   |   |   |--- V17 >  1.52
|   |   |   |   |   |   |   |   |   |--- V1 <= 1.05
|   |   |   |   |   |   |   |   |   |   |--- V10 <= -2.03
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 2
|   |   |   |   |   |   |   |   |   |   |--- V10 >  -2.03
|   |   |   |   |   |   |   |   |   |   |   |--- weights: [356.00, 0.00] class: 0
|   |   |   |   |   |   |   |   |   |--- V1 >  1.05
|   |   |   |   |   |   |   |   |   |   |--- V4 <= 3.03
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 2
|   |   |   |   |   |   |   |   |   |   |--- V4 >  3.03
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 5
```

In [73]:

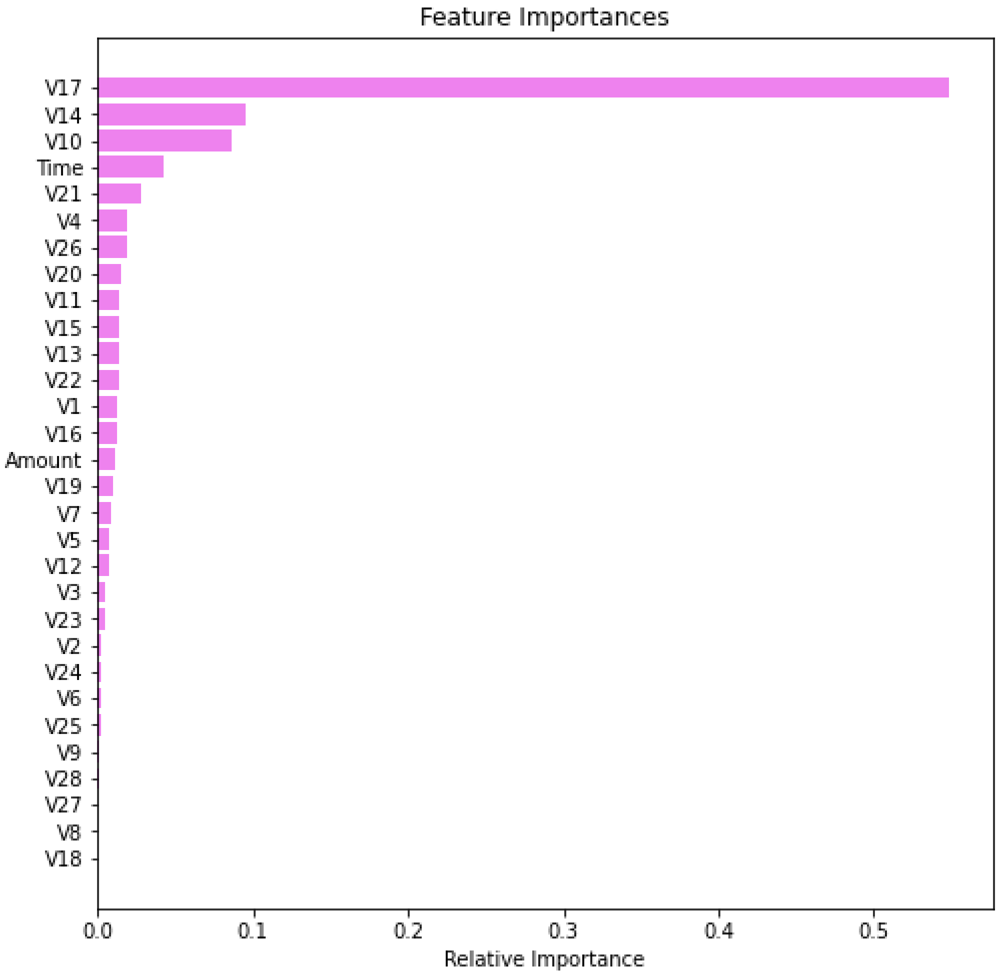
```
# importance of features in the tree building ( The importance of a feature is compu
#(normalized) total reduction of the criterion brought by that feature. It is also k

print (pd.DataFrame(dTree.feature_importances_, columns = ["Imp"], index = X_train.c
```

```
             Imp
V17     0.549237
```

```
V14       0.095711
V10       0.086110
Time      0.042388
V21       0.028209
V4        0.019497
V26       0.018988
V20       0.015216
V11       0.013917
V15       0.013532
V13       0.013315
V22       0.013269
V1        0.012384
V16       0.011963
Amount    0.011054
V19       0.009804
V7        0.008890
V5        0.007822
V12       0.006775
V3        0.005238
V23       0.004823
V2        0.002806
V6        0.002105
V24       0.002105
V25       0.002105
V9        0.001403
V28       0.001335
V18       0.000000
V8        0.000000
V27       0.000000
```

In [74]:

```
importances = dTree.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(8,8))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



Observation

- V17, V14, V10 are the most important features from the decision tree
- V18, V8, V9 have very little importance
- But people having V17 less than -2.789, V10 less than and V26 less than -0.22, have greater chances of being a fraudulent transaction.

Reducing overfitting (Regularization)

- In general, the deeper you allow your tree to grow, the more complex your model will become because you will have more splits and it captures more information about the data and this is one of the root causes of overfitting

Let's try Grid search

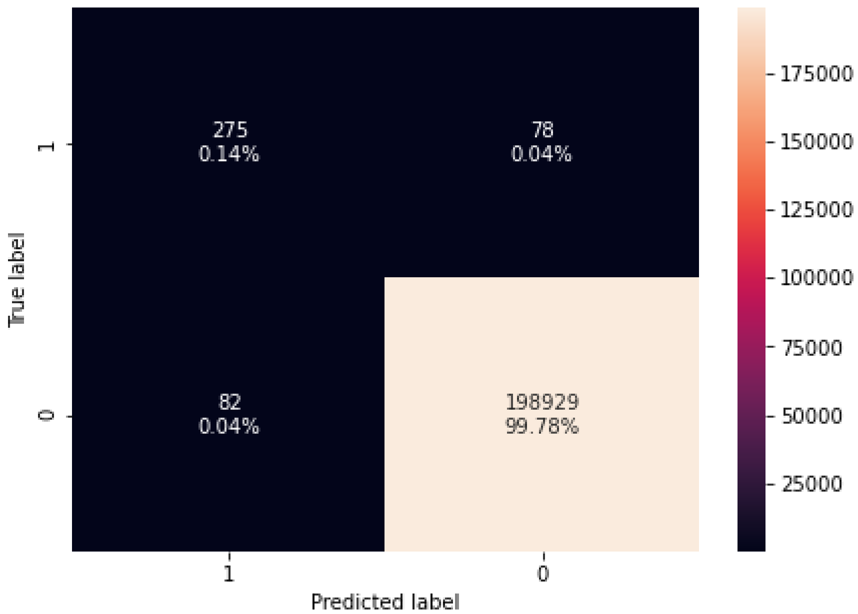
- Hyperparameter tuning is also tricky in the sense that there is no direct way to calculate how a change in then hyperparameter value will reduce the loss of your model, so we usually resort to experimentation. i.e we'll use Gridsearch
- Grid search is a tuning technique that attempts to compute the optimum values of hyperparameters.
- It is an exhaustive search that is performed on a the specific parameter values of a model.
- The parameters of the estimator/model used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

In [75]:
```
# Choose the type of classifier.
estimator = DecisionTreeClassifier(random_state=1)

# Grid of parameters to choose from
parameters = {'max_depth': np.arange(6,15),
              'min_samples_leaf': [1, 2, 5, 7, 10],
              'max_leaf_nodes' : [2, 3, 5, 10],
             }

# Type of scoring used to compare parameter combinations
acc_scorer = metrics.make_scorer(metrics.recall_score)

# Run the grid search
grid_obj = GridSearchCV(estimator, parameters, scoring=acc_scorer,cv=5)
grid_obj = grid_obj.fit(X_train, y_train)

# Set the clf to the best combination of parameters
estimator = grid_obj.best_estimator_

# Fit the best algorithm to the data.
estimator.fit(X_train, y_train)
```

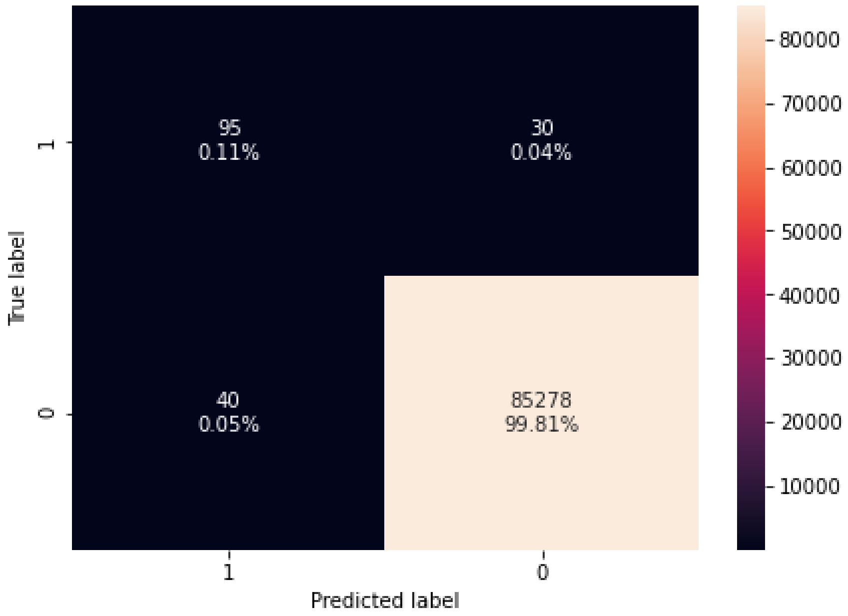
Out[75]:
```
DecisionTreeClassifier(max_depth=6, max_leaf_nodes=3, random_state=1)
```

In [76]:
```
prob_train = estimator.predict_proba(X_train)
pred_train = estimator.predict(X_train)

prob_test = estimator.predict_proba(X_test)
pred_test = estimator.predict(X_test)
```

In [77]:
```
# Let us make confusion matrix on train set
make_confusion_matrix(y_train, pred_train)
```



In [78]:
```
# Let us make confusion matrix on test set
make_confusion_matrix(y_test,pred_test)
```



In [79]:

```
# Let's check model performances for this model
scores_DT = get_metrics_score2(estimator,X_train,X_test,y_train,y_test,flag=True)
```

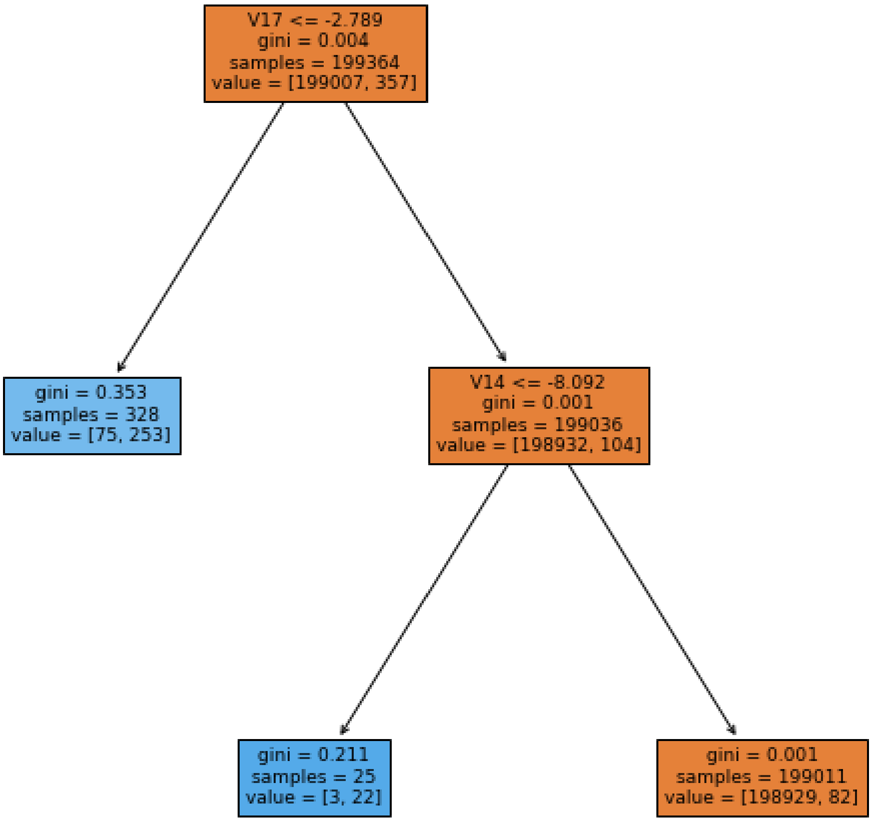
```
Accuracy on training set :  0.999197447884272
Accuracy on test set :  0.9991807403766253
Recall on training set :  0.7703081232492998
Recall on test set :  0.7037037037037037
Precision on training set :  0.7790368271954674
Precision on test set :  0.76
ROC-AUC Score on training set: 0.8849580886186752
ROC-AUC Score on test set: 0.8516760184012963
```

The overfitting in terms of precision and recall has been reduced, while the ROC-AUC curve has reduced to 0.85167

The value of precision and recall for both train and test respectively have values close to each other, indicating the removal of overfitting

In [80]:

```
plt.figure(figsize=(10,10))
out = tree.plot_tree(estimator,feature_names=feature_names,filled=True,fontsize=9,no
#below code will add arrows to the decision tree split if they are missing
for o in out:
     arrow = o.arrow_patch
     if arrow is not None:
        arrow.set_edgecolor('black')
        arrow.set_linewidth(1)
plt.show()
```



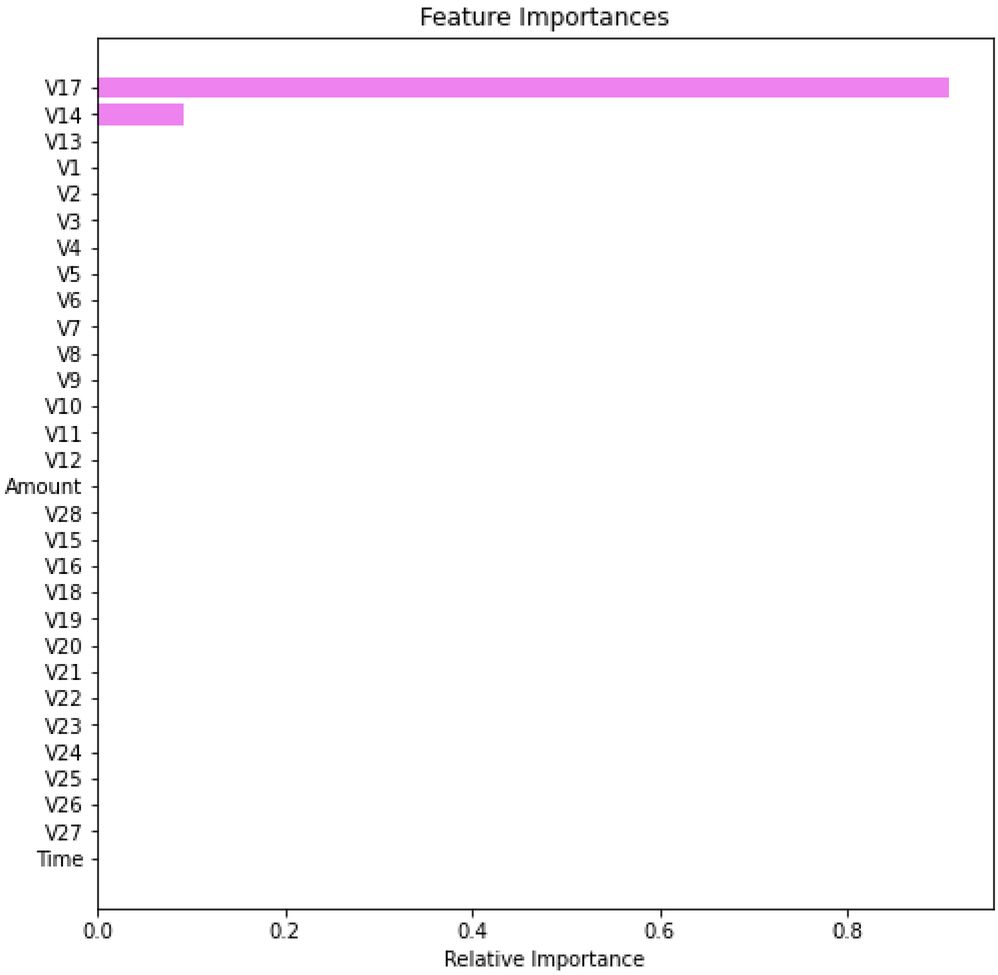
In [81]:

```
# Text report showing the rules of a decision tree -

print(tree.export_text(estimator,feature_names=feature_names,show_weights=True))
```

```
|--- V17 <= -2.79
|   |--- weights: [75.00, 253.00] class: 1
|--- V17 >  -2.79
|   |--- V14 <= -8.09
|   |   |--- weights: [3.00, 22.00] class: 1
|   |--- V14 >  -8.09
|   |   |--- weights: [198929.00, 82.00] class: 0
```

In [82]:

```
importances = estimator.feature_importances_
indices = np.argsort(importances)

plt.figure(figsize=(8,8))
plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='violet', align='center')
plt.yticks(range(len(indices)), [feature_names[i] for i in indices])
plt.xlabel('Relative Importance')
plt.show()
```



- Decision tree after pre-pruning has given similar feature importance and decision rules.

Comparing Model Performance

In [83]:

```
# defining list of model
models = [lg1]

# defining empty lists to add train and test results
acc_train = []
acc_test = []
recall_train = []
recall_test = []
precision_train = []
precision_test = []
# looping through all the models to get the metrics score - Accuracy, Recall and Pre
for model in models:

    j = get_metrics_score1(model, X_train1, X_test1, y_train, y_test, False)
    acc_train.append(j[0])
    acc_test.append(j[1])
    recall_train.append(j[2])
    recall_test.append(j[3])
    precision_train.append(j[4])
    precision_test.append(j[5])
```

In [84]:

```
# defining list of model
models = [dTree,estimator]
```

```
# Looping through all the models to get the metrics score - Accuracy, Recall and Pre
for model in models:

    j = get_metrics_score2(model,X_train,X_test,y_train,y_test,False)
    acc_train.append(j[0])
    acc_test.append(j[1])
    recall_train.append(j[2])
    recall_test.append(j[3])
    precision_train.append(j[4])
    precision_test.append(j[5])
```

In [133…

```
comparison_frame = pd.DataFrame({'Model':['Logistic Regression',
                                          'Decision Tree','Decision Tree(pre-pruned)
                                          'Train_Accuracy': acc_train,'Test_Accuracy
                                          'Train_Recall':recall_train,'Test_Recall':
                                          'Train_Precision':precision_train,'Test_Pr


new_row = {'Model':'Logistic Regression with precision-recall curve threshold', 'Tra
           'Train_Recall':0.8067226890756303,'Test_Recall':0.7407407407407407,'Train
           'Test_Precision':0.7751937984496124}

new_row1 = {'Model':'Logistic Regression with optimal threshold', 'Train_Accuracy':0
           'Train_Recall':0.9159663865546218,'Test_Recall':0.8740740740740741,'Train
           'Test_Precision':0.034942256440627775}


#append rows to the dataframe
comparison_frame = comparison_frame.append(new_row, ignore_index=True)
comparison_frame = comparison_frame.append(new_row1, ignore_index=True)

comparison_frame
```

Out[133…

	Model	Train_Accuracy	Test_Accuracy	Train_Recall	Test_Recall	Train_Precision	Test_Precision
0	Logistic Regression	0.999253	0.999181	0.658263	0.585185	0.896947	0.849462
1	Decision Tree	1.000000	0.999146	1.000000	0.733333	1.000000	0.727941
2	Decision Tree(pre-pruned)	0.999197	0.999181	0.770308	0.703704	0.779037	0.760000
3	Logistic Regression with precision-recall curv...	0.999293	0.999251	0.806723	0.740741	0.800000	0.775194
4	Logistic Regression with optimal threshold	0.961864	0.961659	0.915966	0.874074	0.041392	0.034942

From the models above,the best model for giving us the highest recall is the logistic regression optimal threshold, followed by the logistic regression with

precision-recall curve. But if the bank wants an harmonic of both the precision and the recall, then the pruned decision tree can be used.