1. Background

The dataset contains transactions made by credit cards in September 2013 by European cardholders. This dataset presents transactions that occurred in two days. The dataset has two classes, the positive class represents fraudulent transactions. It contains only numeric input variables which are the result of a PCA transformation. Unfortunately, due to confidentiality issues. Features $V_1, V_2, \ldots, V_{28}$ 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. Feature $class$ is the **response variable** and it takes value 1 in case of fraud and 0 otherwise.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import math
import random
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
```

executed in 2.30s, finished 07:34:24 2022-03-17

2. Exploratory Data Analysis

Load your data and perform simple exploratory data analysis (EDA).The data is located in the Files section of the canvas course page.

It is called hw2_credit_card_fraud.zip. It is a standard csv file. Answer the following:

2.1. What do you notice about the dataset?

In [2]:
```
credit = pd.read_csv("C:\\Users\\tianj\\Desktop\\" + \
                     "MATH 5430 Machine Learning for Finance\\" + \
                     "MATH 5430 - HW2\\creditcard.csv", index_col = "Time")
```

executed in 2.17s, finished 07:34:26 2022-03-17

First of all, we can see that there is no missing value in this dataset.

In [3]:
```
credit.isnull().sum()
```

executed in 29ms, finished 07:34:26 2022-03-17

Out[3]:
```
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       0
V26       0
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

We can also gather basic statistics of the dataset, such as mean, standard deviation, quantiles and etc.

In [4]:
```
credit.describe()
```

executed in 337ms, finished 07:34:26 2022-03-17

Out[4]:

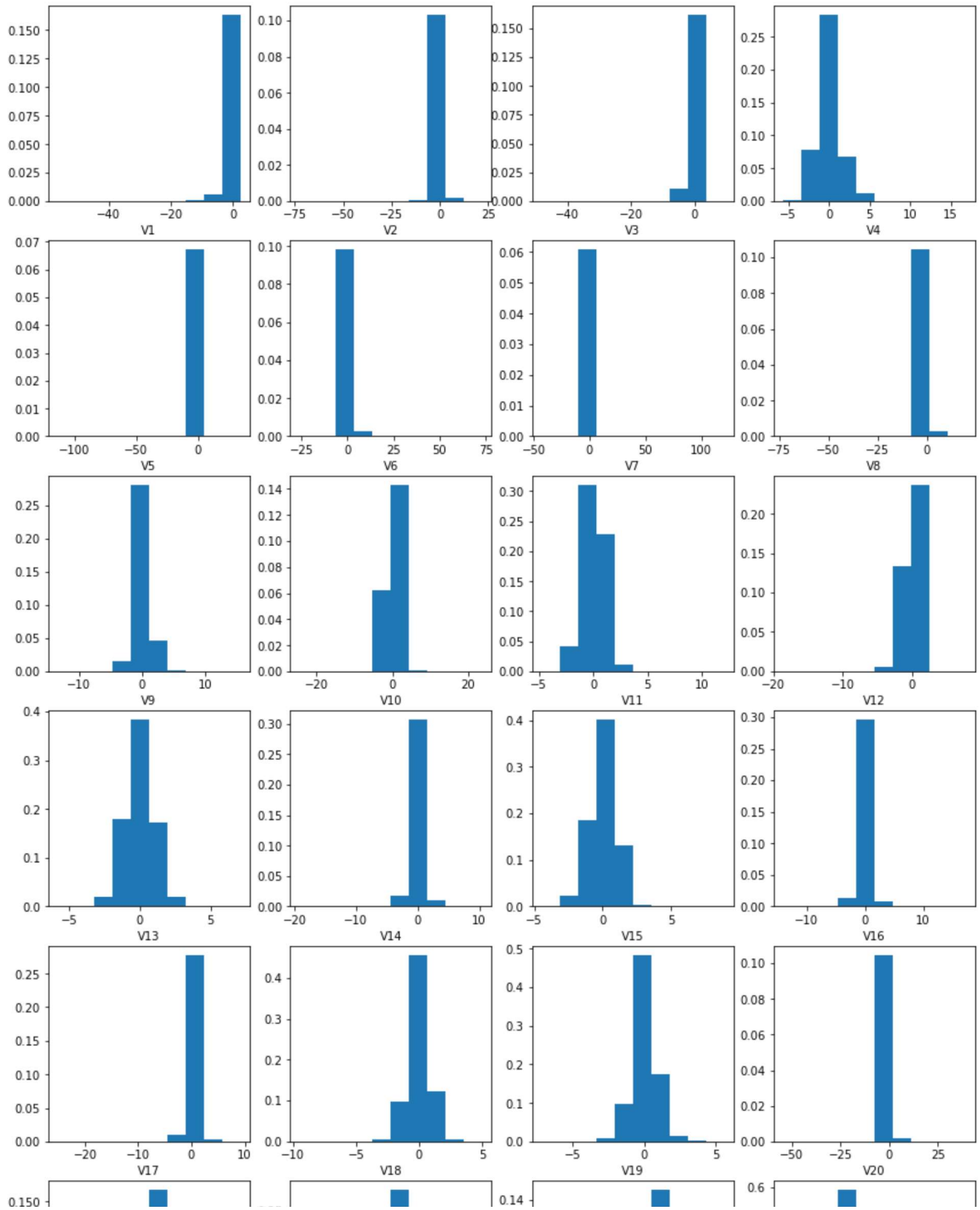
	V1	V2	V3	V4	V5	V6	
count	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.8
mean	3.918649e-15	5.682686e-16	-8.761736e-15	2.811118e-15	-1.552103e-15	2.040130e-15	-1.6
std	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.2
min	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.3
25%	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.5
50%	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.0
75%	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.7
max	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.2

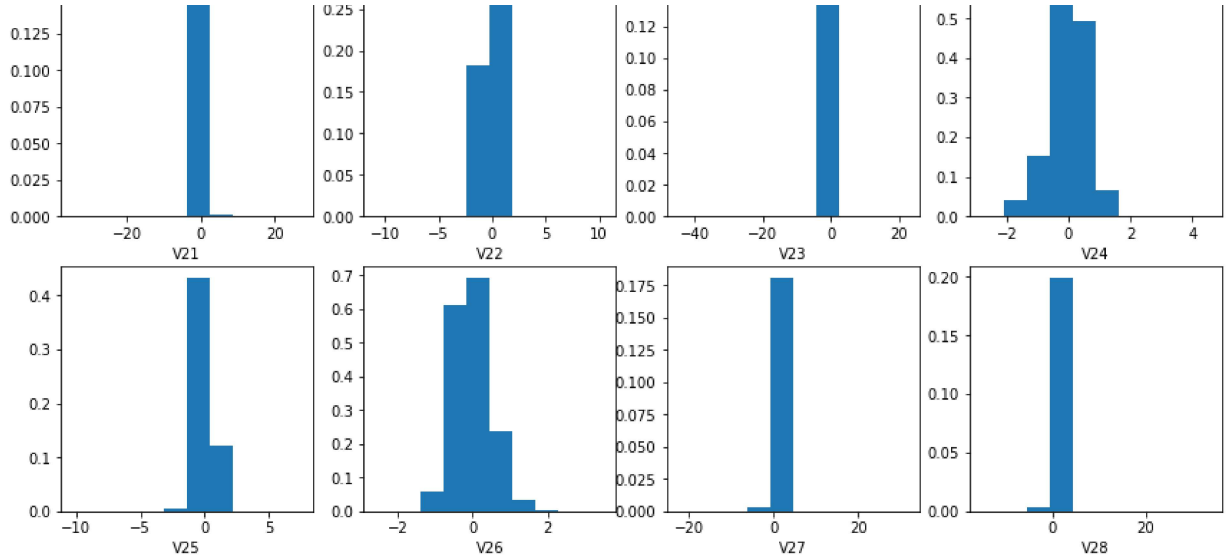
8 rows × 30 columns

From the information table above, for features $V_1, V_2, \ldots, V_{28}$, we can observe that their magnitudes vary from 10^{-15} to 10^{-16}, the distributions are concentrated around 0, and their means are also close to 0 . We can draw histograms to verify the assumption.

In [5]:
```
fig, axs = plt.subplots(7, 4)
plt.subplots_adjust(right = 2, top = 5)

which_col = 0
for row in range(7):
    for col in range(4):
        axs[row, col].hist(credit.iloc[:, which_col], density = True)
        axs[row, col].set(xlabel = credit.iloc[:, which_col].name)
        which_col += 1
plt.show()
```

executed in 2.80s, finished 07:34:29 2022-03-17





The above histogram matrix shows very strong evidence in support of our assumption, and therefore, we can believe that these factors are factorized (i.e.they are mapped to have similar scales), and there may even be bell-shaped distributions.

In [6]:
```
print(f"The number of people who didn't fraud is: " + str(credit[credit["Class"] == 0].sha
print(f"The number of people who frauded is: " + str(credit[credit["Class"] == 1].shape[0]
```

executed in 59ms, finished 07:34:29 2022-03-17

```
The number of people who didn't fraud is: 284315
The number of people who frauded is: 492
```

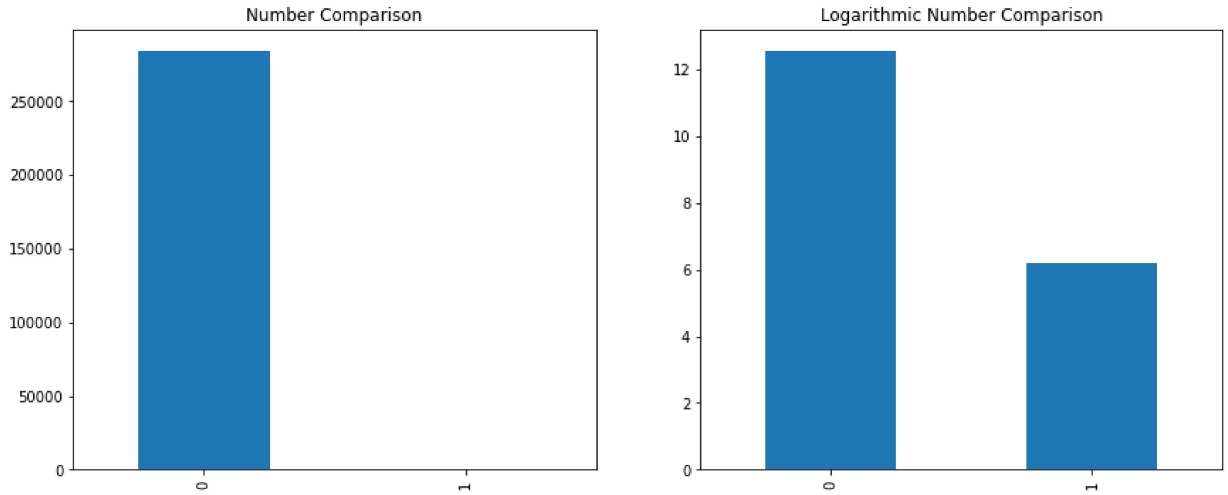
The above computations shows a huge difference between the exact numbers of fraudulent records and honest records. Hence, I am presenting this difference graphically as two bar plots as below.

I used the true number in the first barplot, as you can see, nothing displayed for class 1 other than void.

Then I took logarithmic values for both classes in the second barplot on the left. Although it looks better now, we have to admit the existence of a significant discrepancy between the sizes of class 0 and class 1. Hence, it shows very strong evidence of data inbalance in this dataset.

In [7]:
```
fig, axs = plt.subplots(1, 2)

plt.subplots_adjust(right = 2, top = 1.2)
pd.Series([credit[credit["Class"] == 0].shape[0],
           credit[credit["Class"] == 1].shape[0]]).plot.bar(ax=axs[0])
axs[0].set(title = "Number Comparison")
pd.Series([math.log(credit[credit["Class"] == 0].shape[0]),
           math.log(credit[credit["Class"] == 1].shape[0])]).plot.bar(ax=axs[1])
axs[1].set(title = "Logarithmic Number Comparison")
plt.show()
```

executed in 338ms, finished 07:34:29 2022-03-17



In [8]:
```
CovMat = pd.DataFrame(np.cov([credit[colname] for colname in credit.columns]), index = cr
CovMat.columns = credit.columns
CovMat
```

executed in 261ms, finished 07:34:30 2022-03-17

Out[8]:

	V1	V2	V3	V4	V5	V6	
V1	3.836489e+00	1.935454e-16	-2.145055e-15	-7.299171e-16	3.752756e-16	7.798616e-16	-
V2	1.935454e-16	2.726820e+00	7.005845e-17	-2.203434e-16	3.654927e-18	6.418700e-16	-
V3	-2.145055e-15	7.005845e-17	2.299029e+00	-7.258119e-16	-1.398964e-16	3.018216e-15	2.7
V4	-7.299171e-16	-2.203434e-16	-7.258119e-16	2.004684e+00	-3.624699e-15	-7.682611e-16	-
V5	3.752756e-16	3.654927e-18	-1.398964e-16	-3.624699e-15	1.905081e+00	1.123123e-15	2.5
V6	7.798616e-16	6.418700e-16	3.018216e-15	-7.682611e-16	1.123123e-15	1.774946e+00	-
V7	-7.473558e-16	-2.698409e-16	2.773184e-16	-1.724427e-16	2.523626e-16	-6.386766e-18	1.5
V8	1.462770e-16	-2.211418e-16	1.468442e-17	1.022681e-15	1.009231e-15	-1.037450e-15	2.2
V9	3.009602e-16	-2.091666e-16	3.831335e-16	9.234865e-16	1.477251e-15	2.006343e-16	1.(
V10	4.055191e-17	-3.065648e-16	3.068817e-16	-1.676526e-16	5.387815e-16	2.283269e-16	1.4
V11	6.832951e-16	-2.510798e-16	-6.380841e-17	-6.149259e-16	9.469538e-16	1.133651e-15	-
V12	4.374996e-16	-5.599398e-16	1.463940e-16	-3.033714e-16	4.727683e-16	2.666475e-16	-
V13	-7.444891e-17	2.209422e-16	3.312004e-16	-1.249411e-16	-4.442488e-16	-3.632473e-16	1.9
V14	1.468965e-15	-4.714232e-16	7.446313e-16	1.137643e-16	3.894908e-17	3.288436e-16	4.1
V15	-1.628341e-16	1.676526e-17	-5.815485e-16	-4.730199e-17	4.455941e-17	-6.949600e-16	-
V16	3.844768e-16	-1.476940e-17	6.790507e-16	5.064506e-16	2.191351e-15	-2.145554e-16	1.(
V17	2.683440e-16	-7.965495e-16	1.419736e-16	-3.857008e-16	5.807644e-16	8.662052e-17	1.8
V18	2.392636e-16	2.877039e-16	4.917467e-17	-2.358114e-16	7.879039e-16	2.163517e-16	2.8

	V1	V2	V3	V4	V5	V6	
V19	2.848493e-16	4.829992e-17	5.432926e-16	-2.402023e-16	-5.845560e-17	-6.680159e-16	-
V20	8.736200e-16	3.416920e-16	-1.118756e-16	-1.476940e-16	-2.273138e-16	2.594624e-17	8.1
V21	-1.617697e-16	1.724427e-16	-1.636824e-16	-6.386766e-17	-5.890974e-17	1.149618e-16	-
V22	3.552126e-16	2.067716e-16	-9.407293e-16	3.562618e-16	-1.925545e-17	3.772184e-17	-
V23	2.052484e-16	2.422980e-16	-5.763254e-17	2.418988e-16	8.631363e-17	-1.177560e-17	-
V24	-7.059339e-17	-1.834199e-16	-1.138296e-16	1.307291e-16	-8.093952e-16	-8.791783e-16	-
V25	4.664801e-16	3.954556e-16	5.949333e-17	3.159454e-16	-1.132384e-17	3.065648e-16	1.4
V26	-1.475008e-16	1.742390e-16	-4.690944e-16	-2.219401e-16	1.440735e-17	-1.658563e-16	-
V27	7.428721e-17	-2.845105e-16	3.019253e-16	6.177201e-17	2.463351e-16	-6.306932e-17	-
V28	2.280463e-16	-2.278270e-16	3.588894e-16	7.185112e-18	-1.572082e-16	1.788295e-16	-
Amount	-1.115566e+02	-2.194854e+02	-7.997555e+01	3.496456e+01	-1.333808e+02	7.197093e+01	1.2
Class	-8.243501e-03	6.260047e-03	-1.214993e-02	7.846318e-03	-5.443715e-03	-2.414579e-03	-

30 rows × 30 columns

In [9]:

```
StdMat = pd.DataFrame(np.dot(pd.DataFrame(map(np.std, [credit[colname] for colname in cre
                             pd.DataFrame(map(np.std, [credit[colname] for colname in cre
                        index = credit.columns)
StdMat.columns = credit.columns
StdMat
```

executed in 106ms, finished 07:34:30 2022-03-17

Out[9]:

	V1	V2	V3	V4	V5	V6	V7	V
V1	3.836476	3.234400	2.969872	2.773246	2.703474	2.609505	2.423082	2.339
V2	3.234400	2.726810	2.503796	2.338028	2.279205	2.199983	2.042816	1.972
V3	2.969872	2.503796	2.299021	2.146810	2.092799	2.020056	1.875743	1.810
V4	2.773246	2.338028	2.146810	2.004677	1.954241	1.886314	1.751556	1.691
V5	2.703474	2.279205	2.092799	1.954241	1.905074	1.838856	1.707488	1.648
V6	2.609505	2.199983	2.020056	1.886314	1.838856	1.774940	1.648138	1.591
V7	2.423082	2.042816	1.875743	1.751556	1.707488	1.648138	1.530395	1.477
V8	2.339366	1.972238	1.810937	1.691041	1.648496	1.591196	1.477521	1.426
V9	2.151879	1.814174	1.665801	1.555513	1.516378	1.463671	1.359106	1.312
V10	2.132718	1.798021	1.650968	1.541663	1.502876	1.450638	1.347004	1.300
V11	1.999259	1.685506	1.547656	1.445190	1.408831	1.359862	1.262713	1.219
V12	1.957125	1.649984	1.515039	1.414733	1.379140	1.331202	1.236101	1.193
V13	1.949433	1.643499	1.509084	1.409173	1.373719	1.325970	1.231243	1.188
V14	1.877591	1.582932	1.453470	1.357241	1.323094	1.277105	1.185868	1.144
V15	1.792819	1.511464	1.387848	1.295963	1.263357	1.219445	1.132328	1.093
V16	1.716307	1.446959	1.328618	1.240655	1.209441	1.167402	1.084003	1.046
V17	1.663587	1.402513	1.287807	1.202545	1.172291	1.131543	1.050706	1.014
V18	1.641726	1.384083	1.270884	1.186743	1.156886	1.116674	1.036899	1.001
V19	1.594452	1.344227	1.234289	1.152570	1.123573	1.084519	1.007041	0.972
V20	1.510002	1.273031	1.168915	1.091525	1.064063	1.027078	0.953703	0.920
V21	1.438704	1.212922	1.113722	1.039986	1.013821	0.978582	0.908672	0.877
V22	1.421424	1.198353	1.100345	1.027494	1.001644	0.966828	0.897758	0.866
V23	1.223123	1.031173	0.946838	0.884151	0.861906	0.831947	0.772513	0.745
V24	1.186274	1.000107	0.918312	0.857514	0.835939	0.806883	0.749239	0.723
V25	1.021022	0.860788	0.790388	0.738059	0.719490	0.694481	0.644867	0.622
V26	0.944533	0.796303	0.731177	0.682768	0.665590	0.642455	0.596558	0.575
V27	0.790590	0.666519	0.612008	0.571489	0.557110	0.537746	0.499329	0.482
V28	0.646530	0.545067	0.500489	0.467353	0.455595	0.439759	0.408342	0.394

	V1	V2	V3	V4	V5	V6	V7	V
Amount	489.907488	413.024032	379.244536	354.135959	345.226252	333.226620	309.420899	298.730
Class	0.081339	0.068574	0.062966	0.058797	0.057318	0.055325	0.051373	0.049

30 rows × 30 columns

In [10]:
```
CorrMat = CovMat / StdMat
CorrMat
```

executed in 58ms, finished 07:34:30 2022-03-17

Out[10]:

	V1	V2	V3	V4	V5	V6	
V1	1.000004e+00	5.983965e-17	-7.222720e-16	-2.631995e-16	1.388123e-16	2.988543e-16	
V2	5.983965e-17	1.000004e+00	2.798089e-17	-9.424330e-17	1.603597e-18	2.917614e-16	
V3	-7.222720e-16	2.798089e-17	1.000004e+00	-3.380885e-16	-6.684655e-17	1.494125e-15	1
V4	-2.631995e-16	-9.424330e-17	-3.380885e-16	1.000004e+00	-1.854786e-15	-4.072816e-16	-9
V5	1.388123e-16	1.603597e-18	-6.684655e-17	-1.854786e-15	1.000004e+00	6.107728e-16	1
V6	2.988543e-16	2.917614e-16	1.494125e-15	-4.072816e-16	6.107728e-16	1.000004e+00	
V7	-3.084319e-16	-1.320926e-16	1.478446e-16	-9.845116e-17	1.477976e-16	-3.875140e-18	1.
V8	6.252848e-17	-1.121273e-16	8.108739e-18	6.047642e-16	6.122134e-16	-6.519940e-16	1
V9	1.398593e-16	-1.152958e-16	2.299996e-16	5.936861e-16	9.741969e-16	1.370761e-16	1
V10	1.901419e-17	-1.705013e-16	1.858799e-16	-1.087479e-16	3.585003e-16	1.573976e-16	1
V11	3.417741e-16	-1.489640e-16	-4.122907e-17	-4.254982e-16	6.721558e-16	8.336517e-16	
V12	2.235420e-16	-3.393607e-16	9.662721e-17	-2.144372e-16	3.427994e-16	2.003057e-16	
V13	-3.819004e-17	1.344340e-16	2.194711e-16	-8.866275e-17	-3.233913e-16	-2.739483e-16	1
V14	7.823673e-16	-2.978165e-16	5.123127e-16	8.382027e-17	2.943788e-17	2.574915e-16	3
V15	-9.082569e-17	1.109207e-17	-4.190291e-16	-3.649950e-17	3.527063e-17	-5.698987e-16	
V16	2.240140e-16	-1.020720e-17	5.110955e-16	4.082124e-16	1.811871e-15	-1.837888e-16	9
V17	1.613044e-16	-5.679446e-16	1.102445e-16	-3.207370e-16	4.954099e-16	7.655078e-17	1
V18	1.457390e-16	2.078661e-16	3.869327e-17	-1.987047e-16	6.810558e-16	1.937465e-16	2

	V1	V2	V3	V4	V5	V6	
V19	1.786503e-16	3.593136e-17	4.401666e-16	-2.084058e-16	-5.202653e-17	-6.159560e-16	
V20	5.785554e-16	2.684083e-16	-9.570895e-17	-1.353098e-16	-2.136281e-16	2.526220e-17	8
V21	-1.124413e-16	1.421714e-16	-1.469689e-16	-6.141205e-17	-5.810666e-17	1.174780e-16	
V22	2.498992e-16	1.725465e-16	-8.549405e-16	3.467287e-16	-1.922385e-17	3.901609e-17	
V23	1.678067e-16	2.349731e-16	-6.086844e-17	2.735945e-16	1.001427e-16	-1.415426e-17	
V24	-5.950849e-17	-1.834004e-16	-1.239552e-16	1.524514e-16	-9.682463e-16	-1.089598e-15	
V25	4.568759e-16	4.594112e-16	7.527107e-17	4.280762e-16	-1.573871e-17	4.414299e-16	2
V26	-1.561627e-16	2.188099e-16	-6.415611e-16	-3.250595e-16	2.164598e-17	-2.581603e-16	
V27	9.396420e-17	-4.268600e-16	4.933358e-16	1.080897e-16	4.421657e-16	-1.172846e-16	
V28	3.527232e-16	-4.179794e-16	7.170781e-16	1.537406e-17	-3.450614e-16	4.066535e-16	
Amount	-2.277095e-01	-5.314108e-01	-2.108812e-01	9.873201e-02	-3.863576e-01	2.159819e-01	3
Class	-1.013477e-01	9.128897e-02	-1.929615e-01	1.334480e-01	-9.497463e-02	-4.364331e-02	

30 rows × 30 columns

In the above several steps, I derive the matrix of covariance $\left(\text{i.e. CovMat} = (Cov(V_i, V_j))_{i,j\in\{1,\ldots,30\}}\right)$, and the matrix of pair-wise multiplications of standard deviations $\left(\text{i.e. StdMat} = \left(\sigma(V_i)\cdot\sigma(V_j)\right)_{i,j\in\{1,\ldots,30\}}\right)$. Here, I just simply denote the columns of **Amount** and **Class** as V_{29} and V_{30} respectively in convenience.

Since we know that:

$$\text{Corr}\,(X, Y) = \frac{\text{Cov}\,(X, Y)}{\sigma_X \cdot \sigma_Y}$$

Therefore we can derive the matrix of correlations (i.e. **CorrMat**) through simply dividing **CovMat** by **StdMat** element-wisely.

The output of **CorrMat** shows that PCA was implemented successfully. At least for the $V_1, V_2, \ldots, V_{28}$, it seems that there are no two columns who are linear dependent on each other within our dataset, or we have confidence to say that there exists almost no collinearity between any two of our features.

1.2. What is the problem with what you observed and how can that affect the training of a ML model?

First of all, as I stated above, there is huge discrepancy between the numbers of two classes, and for class 0 (honest records) we only have 492 rows of data, hence the dataset doesn't contain enough examples for label 0. Hence, the trained model may loss accuracy and have relatively worse performance in predicting class 0.

Besides, since previous operations with PCA only dealt with $V_1, V_2, \ldots, V_{28}$ which excluding the variable "Amount" and "Class". From the Correlation table generated above, we can also oberve some pairs of variables that exists a relatively high correlation (absolute value) between them. For instance, the correlation between V_2 and **Amount** is around **-0.531411**, which is significantly higher than the correlations among those principle components generated by PCA.

In [11]:
```
def excludeSelfCorr(x):
    return x if int(round(x, 2)) != 1 else None


descCorr = list(map(lambda x: excludeSelfCorr(x),
                    sorted(np.abs([i for lst in CorrMat.values for i in lst]), reverse=Tr

for i in range(len(descCorr)-1, -1, -1):
    if descCorr[i] is None:
        descCorr.pop(i)
    else:
        continue

descCorr.sort(reverse=True)
descCorr[:10]
```

executed in 28ms, finished 07:34:30 2022-03-17

Out[11]:
```
[0.5314108051908335,
 0.5314108051908335,
 0.39731267320589464,
 0.39731267320589464,
 0.38635761272240715,
 0.38635761272240715,
 0.3394045962464592,
 0.3394045962464592,
 0.3264822135706092,
 0.3264822135706092]
```

2.3. What is the difference and trade off of oversampling vs undersampling?

Oversampling and undersampling are both methods in adjusting unbalanced distributions of classes in a dataset. They can be almost regarded as one same method towards two opposite directions, but they also have some tiny differences. Oversampling is a method to adjust the class with deficient data up to a desired sample size, and undersampling is exactly the opposite, which is to adjust the

data up to a desired sample size, and undersampling is exactly the opposite, which is to adjust the class with abundant data down to a desired sample size. For instance, in a survey with respect to class A and class B, suppose we get much more data of class A than class B. For oversampling, we can simply choose the data from class B randomly and repeatedly, until the sample size of B is reached our desired level (**Perfect Balance**, that each class has exactly $\frac{N}{n}$ data in total sample, where N is the total sample size, and n is the number of distinct classes). Similarly, for undersampling, we can simply select the data from class A randomly and repeatedly as well, until we get a sample whose size is equal to the size of class B. From these two method, we can both accomplish our goal to balance the distributions of classes in the original dataset.

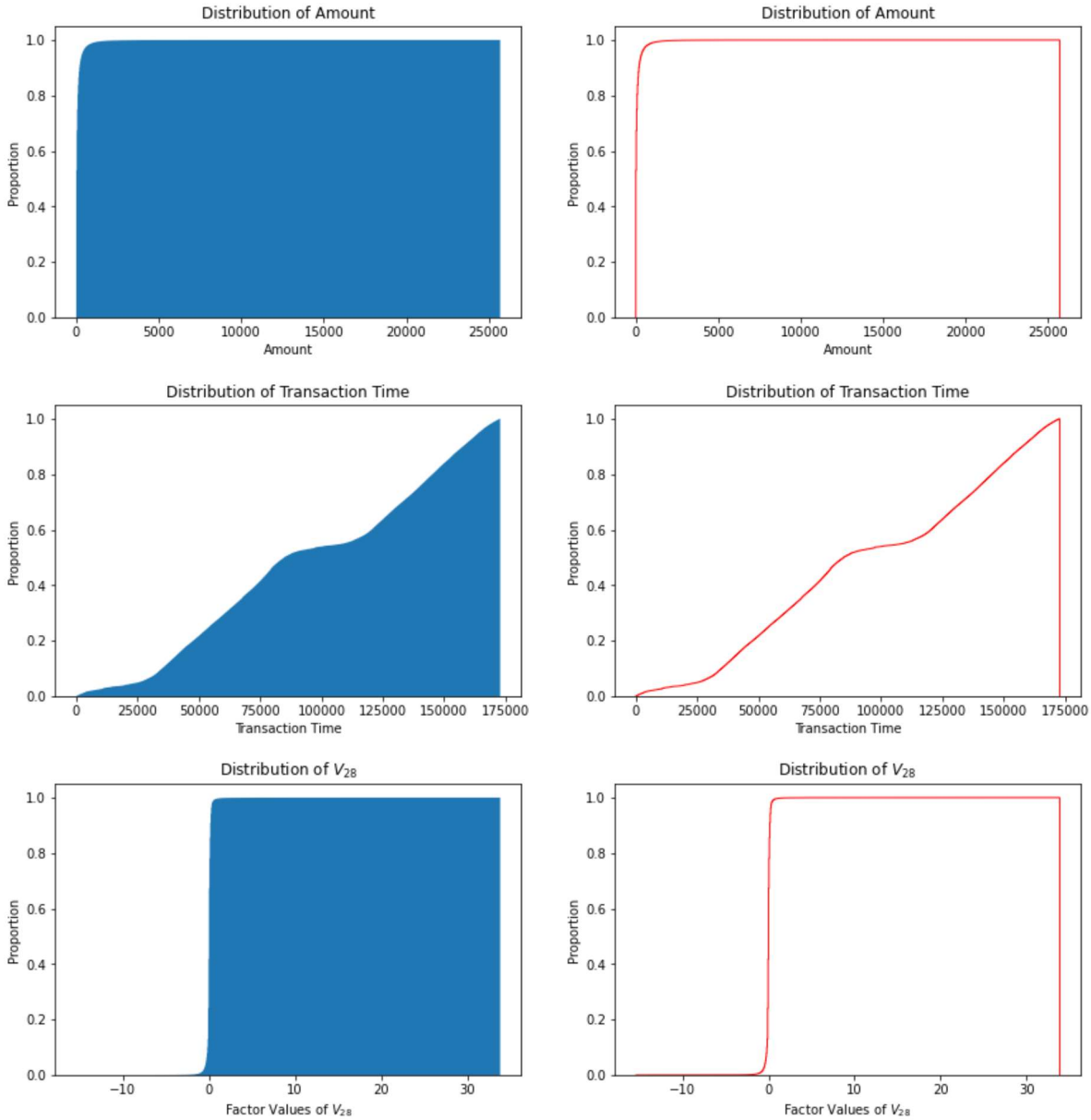
Notice that there are other tiny differences between oversampling and undersampling. Continued with the above example, since the amount of data from class A is much larger than the amount of data from class B, one can be easily deduced is that we must select some repeated data of B from the original dataset by oversampling method, and we must dropped some data of A from the original dataset by undersampling method. Otherwise our desired balance would not be reached.

We have to make some trade-offs between oversampling and undersampling, because none of them is a perfect strategy, and relying on either of them indiscriminatively may result in terrible outcomes. As stated above, applying the oversampling method means that we have to pick up some repeated data of B from the original dataset, and this is just purely equivalent to replicating some of raws for multiple times. Hence, if there are some significant unexpected errors such as outliers and noises among these raws, the process of oversampling would hence enlarge these errors. By constrast, applying the undersamplng method means that we have to give up many valuable data of A in our original dataset, and also reduce the confidence of our conclusion as well as the accuracy and predicting ability of our model.

2.4. Plot the distributions for Amount, Transaction Time, and one of the other variables. What do you see and does it make sense? Plot two of the V_x variables together one on the x axis and the other on the y axis, what do you see?

In [12]:
```
# Plot the distributions for Amount, Transaction Time and $V_{28}$.
fig, axs = plt.subplots(3, 2)
plt.subplots_adjust(right=2, top=3, hspace=0.3)

axs[0, 0].hist(credit["Amount"].values, density=True, bins=1000,
               cumulative=True,  histtype="stepfilled")
axs[0, 0].set(title="Distribution of Amount",
              xlabel="Amount", ylabel="Proportion")
axs[0, 1].hist(credit["Amount"].values, density=True, bins=1000,
               cumulative=True,  histtype="step", color="red")
axs[0, 1].set(title="Distribution of Amount",
              xlabel="Amount", ylabel="Proportion")

axs[1, 0].hist(credit.index, density=True, bins=1000,
               cumulative=True, histtype="stepfilled")
axs[1, 0].set(title="Distribution of Transaction Time",
              xlabel="Transaction Time", ylabel="Proportion")
axs[1, 1].hist(credit.index, density=True, bins=1000,
               cumulative=True, histtype="step", color="red")
axs[1, 1].set(title="Distribution of Transaction Time",
              xlabel="Transaction Time", ylabel="Proportion")

axs[2, 0].hist(credit["V28"].values, density=True, bins=1000,
               cumulative=True, histtype="stepfilled")
axs[2, 0].set(title="Distribution of $V_{28}$",
              xlabel="Factor Values of $V_{28}$", ylabel="Proportion")
axs[2, 1].hist(credit["V28"].values, density=True, bins=1000,
               cumulative=True, histtype="step", color="red")
axs[2, 1].set(title="Distribution of $V_{28}$",
              xlabel="Factor Values of $V_{28}$", ylabel="Proportion", )

plt.show()
```

executed in 898ms, finished 07:34:31 2022-03-17



What do you see and does it make sense?

From cumulative probability (actually it should be frequency) function plots above, we can see that the distribution functions for variable **Amount** and **V_{28}** take approximately to 1 at 1000 and 1 respectively, which represents that, almost all sample data of **Amount** and **V_{28}** are less than 1000 and 1 respectively.

This idea makes sense since it can be verified by the following codes, and we know that **V_{28}** is a normalized variable, so it is reasonble to have almost $V_{28} < 1$.

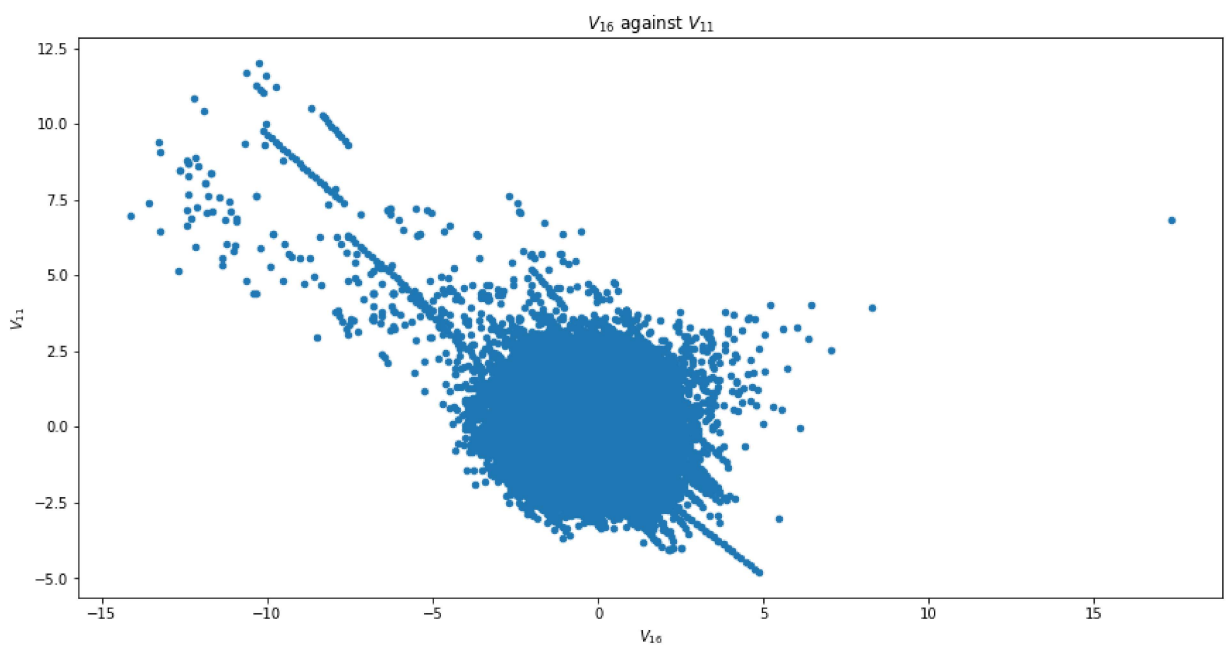
In [13]:
```
# Percentage of "Amount" sample data where values are less than 1000.
print(len(credit["Amount"][credit["Amount"] < 1000]) / len(credit["Amount"]))

# Percentage of "$V_{28}$" sample data where values are less than 1.
print(len(credit["V28"][credit["V28"] < 1]) / len(credit["V28"]))
```

executed in 28ms, finished 07:34:31 2022-03-17

```
0.9892242817065592
0.9964572499973666
```

In [14]:
```
# Plot $V_{16}$ against $V_{11}$.
fig, ax = plt.subplots()
plt.subplots_adjust(right=2, top=1.5)

ax.scatter(np.array(credit["V16"].values), np.array(credit["V11"].values), s=20)
ax.set(title="$V_{16}$ against $V_{11}$",
       xlabel="$V_{16}$", ylabel="$V_{11}$")

plt.show()
```

executed in 773ms, finished 07:34:32 2022-03-17



I plot V_{16} on the x-axis and V_{11} on the y-axis. From the plot, we can see that the data pairs (V_{16}, V_{11}) are centrosymmetic to point $(0, 0)$, with a circle shape of the joint distribution. Besides, the plot suggests a negative linear relationship between variables V_{16} and V_{11} (that "straight line" passes through the circle). However, this does not reflect on correlation as shown below. The "contradictory" phenomenon may have a possible and reasonable explanation that, the amount of data which formed the "trend line" are negligible, compared to the amount of data which form the "circle".

2.5. What other thoughts do you have about the data, that if you had time you would investigate?

Multicollinearity.

Judging whether there is a multicollinearity relationship among the selected variables is so vital to our next step and the model, although we rejected the existence of extreme dependency (or collinearity). The existence of multicollinearity means that the design matrix X is singular or approximates to singular, then we may not get a unique solution for our model parameters, or just get a solution which is emtreme and ridiculous. If we detect any possible multicollinearity relationship, in next step we can discuss how to eliminate it, such as simply remove those variables or apply modified model like ridge regression.

3. Classifying credit fraud

3.1. Split your data into train/test (Hint: from sklearn.model_selection import train_test_split).

In [15]:
```
split_result = train_test_split(credit)
train, test = split_result[0], split_result[1]
```

executed in 214ms, finished 07:34:32 2022-03-17

3.2. What is the balance within the Class variable? (i.e. how many positives and negatives are there, is this relationship balanced?)

In [16]:
```
# Slightly modify the code that I written for the first question.
print(f"The number of positives is: " + str(credit[credit["Class"] == 0].shape[0]))
print(f"The number of negatives is: " + str(credit[credit["Class"] == 1].shape[0]))
```

executed in 60ms, finished 07:34:32 2022-03-17

```
The number of positives is: 284315
The number of negatives is: 492
```

In [17]:
```
# Compute the proportion of negatives.
credit[credit["Class"] == 1].shape[0] / credit.shape[0]
```

executed in 14ms, finished 07:34:32 2022-03-17

Out[17]: 0.001727485630620034

Negatives only take about $0.17275\% < 1\%$ in the total data size. The dataset is not only imbalanced, by our definition, it is defined as an "extremely imbalanced dataset".

3.3. Perform a sampling method, based on your analysis above I.4, on only the training set.

In [18]:
```
train, test = split_result[0], split_result[1]
def OverUnderSampling(data: pd.DataFrame) -> list:
    # Smaller sample size
    n = data[data["Class"] == 1].shape[0]

    # Larger sample
    negative = data[data["Class"] == 0]

    # Larger sample size
    negRows = negative.shape[0]

    # Calculate the weight
    weight = {0: negRows + n/ n, 1: 1}

    # Downsampling the negative sample
    negRows = random.sample(list(range(negRows)), n)

    # Get the new samples for nagative and positive classes
    negative, positive = negative.iloc[negRows, :], data[data["Class"] == 1]

    return [positive, negative, weight]

samp_result = OverUnderSampling(train)
positive, negative, weight = samp_result[0], samp_result[1], samp_result[2]
train = pd.concat([positive, negative])
```

executed in 59ms, finished 07:34:32 2022-03-17

In [19]:
```
positive.head()
```

executed in 28ms, finished 07:34:32 2022-03-17

Out[19]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9
Time									
84204.0	-0.937843	3.462889	-6.445104	4.932199	-2.233983	-2.291561	-5.695594	1.338825	-4.3223
13126.0	-2.880042	5.225442	-11.063330	6.689951	-5.759924	-2.244031	-11.199975	4.014722	-3.4293
46149.0	-1.346509	2.132431	-1.854355	2.116998	-1.070378	-1.092671	-2.230986	1.036425	-1.8955
34687.0	-0.860827	3.131790	-5.052968	5.420941	-2.494141	-1.811287	-5.479117	1.189472	-3.9082
55279.0	-5.753852	0.577610	-6.312782	5.159401	-1.698320	-2.683286	-7.934389	2.373550	-3.0730

5 rows × 30 columns

In [20]:
```
negative.head()
```
executed in 28ms, finished 07:34:32 2022-03-17

Out[20]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9
Time									
124380.0	1.769860	-0.862830	0.392016	1.537788	-1.048688	0.918484	-1.304324	0.423186	2.116
119322.0	2.138299	-0.016122	-1.783028	0.078105	0.600378	-0.517711	0.192577	-0.269254	0.617
67949.0	-2.343576	0.448862	1.877067	-1.878909	-1.233022	2.539945	-1.863212	1.898426	0.715
47841.0	1.151107	0.458571	0.896045	2.738490	-0.272516	-0.120358	0.010854	-0.064326	-0.339
38645.0	1.026473	-0.304240	0.782724	0.773048	-0.867229	-0.507320	-0.181058	-0.139676	0.463

5 rows × 30 columns

In [21]:
```
print(positive.shape[0] == negative.shape[0])
print(set(positive["Class"]), set(negative["Class"]))
```
executed in 13ms, finished 07:34:32 2022-03-17

```
True
{1} {0}
```

In [22]:
```
X_train, y_train = train.iloc[:, :29], train.iloc[:, 29:]["Class"]

X_test, y_test = test.iloc[:, :29], test.iloc[:, 29:]["Class"]
```
executed in 29ms, finished 07:34:32 2022-03-17

3.4. Train a Logistic Lasso Regression model (Hint: from sklearn.linear_model import LogisticRegression).

In [23]:
```
clf = LogisticRegression(penalty="l1", solver="liblinear", class_weight=weight).fit(X_trai
y_pred = clf.predict(X_test)
```
executed in 14ms, finished 07:34:32 2022-03-17

In [24]:
```
print(sum(y_pred), sum(y_test))
```
executed in 14ms, finished 07:34:32 2022-03-17

```
105 121
```