Generation of synthetic tabular data using VAE

In [1]:
```
%load_ext watermark
%watermark -p tensorflow,pandas -z -v -n -m -w
```

```
Python implementation: CPython
Python version       : 3.10.5
IPython version      : 8.4.0

tensorflow: 2.9.1
pandas    : 1.4.2

Compiler    : MSC v.1929 64 bit (AMD64)
OS          : Windows
Release     : 10
Machine     : AMD64
Processor   : Intel64 Family 6 Model 142 Stepping 9, GenuineIntel
CPU cores   : 4
Architecture: 64bit

Watermark: 2.3.1
```

In [2]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [3]:
```
#Calculating the computing time
import time
start = time.time()
import datetime
print("Start Time:" ,datetime.datetime.fromtimestamp(start).strftime('%Y-%m-%d %H:%M:%S'))
```

```
Start Time: 2022-08-03 01:10:51
```

In [4]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import nn, optim
from torch.autograd import Variable

import pandas as pd
import numpy as np
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
```

In [5]:
```
path = r"C:\Users\Home\Jupyter\Datasets\creditcard.csv"
device = torch.device('cpu')
```

In [6]:
```
data_original = pd.read_csv(path, sep =",")
data_original = pd.DataFrame(data_original)
del data_original['Time']
data_original
```

Out[6]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V21	V22	V23	V
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	...	-0.018307	0.277838	-0.110474	0.0669
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	...	-0.225775	-0.638672	0.101288	-0.3398
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	...	0.247998	0.771679	0.909412	-0.6892
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	...	-0.108300	0.005274	-0.190321	-1.1755
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	...	-0.009431	0.798278	-0.137458	0.1412
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
284802	-11.881118	10.071785	-9.834783	-2.066656	-5.364473	-2.606837	-4.918215	7.305334	1.914428	4.356170	...	0.213454	0.111864	1.014480	-0.5093
284803	-0.732789	-0.055080	2.035030	-0.738589	0.868229	1.058415	0.024330	0.294869	0.584800	-0.975926	...	0.214205	0.924384	0.012463	-1.0162
284804	1.919565	-0.301254	-3.249640	-0.557828	2.630515	3.031260	-0.296827	0.708417	0.432454	-0.484782	...	0.232045	0.578229	-0.037501	0.6401
284805	-0.240440	0.530483	0.702510	0.689799	-0.377961	0.623708	-0.686180	0.679145	0.392087	-0.399126	...	0.265245	0.800049	-0.163298	0.1232
284806	-0.533413	-0.189733	0.703337	-0.506271	-0.012546	-0.649617	1.577006	-0.414650	0.486180	-0.915427	...	0.261057	0.643078	0.376777	0.0087

284807 rows × 30 columns

In [7]:
```
columns = data_original.columns
```

Build Data Loader

In [8]:
```python
def load_and_standardize_data(path):
    df = pd.read_csv(path) # read in from csv
    del df['Time']
    df = df.values.reshape(-1, df.shape[1]).astype('float32')
    X_train, X_test = train_test_split(df, test_size=0.3, random_state=42) # randomly split
    # Standardize features by removing the mean and scaling to unit variance. z = (x - u) / s
    scaler = preprocessing.StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)
    return X_train, X_test, scaler
```

In [9]:
```python
from torch.utils.data import Dataset, DataLoader
class DataBuilder(Dataset):
    def __init__(self, path, train=True):
        self.X_train, self.X_test, self.standardizer = load_and_standardize_data(path)
        if train:
            self.x = torch.from_numpy(self.X_train)
            self.len=self.x.shape[0]
        else:
            self.x = torch.from_numpy(self.X_test)
            self.len=self.x.shape[0]
        del self.X_train
        del self.X_test
    def __getitem__(self,index):
        return self.x[index]
    def __len__(self):
        return self.len
```

In [10]:
```python
#Dataset
data_set=DataBuilder(path)
traindata_set=DataBuilder(path, train=True)
testdata_set=DataBuilder(path, train=False)
#Loader
trainloader=DataLoader(dataset=traindata_set,batch_size=1024)
testloader=DataLoader(dataset=testdata_set,batch_size=1024)
```

In [11]:
```python
type(trainloader.dataset.x), type(testloader.dataset.x)
```

Out[11]:
```
(torch.Tensor, torch.Tensor)
```

In [12]:
```python
trainloader.dataset.x.shape, testloader.dataset.x.shape
```

Out[12]:
```
(torch.Size([199364, 30]), torch.Size([85443, 30]))
```

In [13]:
```python
trainloader.dataset.x
```

Out[13]:
```
tensor([[-1.1668, -0.2865,  0.5392,  ..., -0.4486, -0.3397, -0.0423],
        [-0.1592, -2.4354, -2.2454,  ...,  0.3352,  4.2777, -0.0423],
        [-0.9221, -0.3388,  1.4944,  ...,  0.2675, -0.0534, -0.0423],
        ...,
        [-0.0740,  0.5967,  1.0054,  ..., -0.5957, -0.3284, -0.0423],
        [-1.5029,  1.4133, -1.6661,  ...,  1.0198, -0.3397, -0.0423],
        [ 0.6296, -0.4692,  0.2541,  ...,  0.1172,  0.0936, -0.0423]])
```

VAE Model

In [14]:
```
class Autoencoder(nn.Module):
    def __init__(self,D_in,H=50,H2=12,latent_dim=3):

        #Encoder
        super(Autoencoder,self).__init__()
        self.linear1=nn.Linear(D_in,H)
        self.lin_bn1 = nn.BatchNorm1d(num_features=H)
        self.linear2=nn.Linear(H,H2)
        self.lin_bn2 = nn.BatchNorm1d(num_features=H2)
        self.linear3=nn.Linear(H2,H2)
        self.lin_bn3 = nn.BatchNorm1d(num_features=H2)

        # Latent vectors mu and sigma
        self.fc1 = nn.Linear(H2, latent_dim)
        self.bn1 = nn.BatchNorm1d(num_features=latent_dim)
        self.fc21 = nn.Linear(latent_dim, latent_dim)
        self.fc22 = nn.Linear(latent_dim, latent_dim)

        # Sampling vector
        self.fc3 = nn.Linear(latent_dim, latent_dim)
        self.fc_bn3 = nn.BatchNorm1d(latent_dim)
        self.fc4 = nn.Linear(latent_dim, H2)
        self.fc_bn4 = nn.BatchNorm1d(H2)

        # Decoder
        self.linear4=nn.Linear(H2,H2)
        self.lin_bn4 = nn.BatchNorm1d(num_features=H2)
        self.linear5=nn.Linear(H2,H)
        self.lin_bn5 = nn.BatchNorm1d(num_features=H)
        self.linear6=nn.Linear(H,D_in)
        self.lin_bn6 = nn.BatchNorm1d(num_features=D_in)

        self.relu = nn.ReLU()

    def encode(self, x):
        lin1 = self.relu(self.lin_bn1(self.linear1(x)))
        lin2 = self.relu(self.lin_bn2(self.linear2(lin1)))
        lin3 = self.relu(self.lin_bn3(self.linear3(lin2)))

        fc1 = F.relu(self.bn1(self.fc1(lin3)))

        r1 = self.fc21(fc1)
        r2 = self.fc22(fc1)

        return r1, r2

    def decode(self, z):
        fc3 = self.relu(self.fc_bn3(self.fc3(z)))
        fc4 = self.relu(self.fc_bn4(self.fc4(fc3)))

        lin4 = self.relu(self.lin_bn4(self.linear4(fc4)))
        lin5 = self.relu(self.lin_bn5(self.linear5(lin4)))
        return self.lin_bn6(self.linear6(lin5))

    def reparameterize(self, mu, logvar):
        if self.training:
            std = logvar.mul(0.5).exp_()
            eps = Variable(std.data.new(std.size()).normal_())
            return eps.mul(std).add_(mu)
        else:
            return mu

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar
```

In [15]:
```
class customLoss(nn.Module):
    def __init__(self):
        super(customLoss, self).__init__()
        self.mse_loss = nn.MSELoss(reduction="sum")

    def forward(self, x_recon, x, mu, logvar):
        loss_MSE = self.mse_loss(x_recon, x)
        loss_KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp()) #Kullback-Leibler Divergence (KL-divergence)
        return loss_MSE + loss_KLD
#the KL loss is equivalent to the sum of all the KL divergences between the component Xi~N(μi, σi²) in X, and the standard normal[.
```

In [16]:
```
D_in = data_set.x.shape[1]
H = 50 #layer 1
H2 = 12 # Layer 2
model = Autoencoder(D_in, H, H2).to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3) # LR = 0.001
loss_mse = customLoss()
```

In [17]:
```python
epochs = 2000
log_interval = 50
val_losses = []
train_losses = []
test_losses = []
```

Notes: When the Bernoulli distribution is modeled, the MSE or the binary cross-entropy could be used. When a normal distribution is modeled, the log-likelihood is often applied.

In [18]:
```python
def train(epoch):
    model.train()
    train_loss = 0
    for batch_idx, data in enumerate(trainloader):
        data = data.to(device)
        optimizer.zero_grad()
        reconstructed_data, mu, logvar = model(data)
        loss = loss_mse(reconstructed_data, data, mu, logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
    if epoch % 100 == 0:
        print('====> Epoch: {} Average training loss: {:.4f}'.format(
            epoch, train_loss / len(trainloader.dataset)))
        train_losses.append(train_loss / len(trainloader.dataset))
```

In [19]:
```python
def test(epoch):
    with torch.no_grad():
        test_loss = 0
        for batch_idx, data in enumerate(testloader):
            data = data.to(device)
            optimizer.zero_grad()
            reconstructed_data, mu, logvar = model(data)
            loss = loss_mse(reconstructed_data, data, mu, logvar)
            test_loss += loss.item()
            if epoch % 100 == 0:
                print('====> Epoch: {} Average test loss: {:.4f}'.format(
                    epoch, test_loss / len(testloader.dataset)))
            test_losses.append(test_loss / len(testloader.dataset))
```

In [20]:
```python
train_start_time =time.time()
```

In [21]:
```python
for epoch in range(1, epochs + 1):
    train(epoch)
    test(epoch)
```

```
====> Epoch: 100 Average training loss: 19.8206
====> Epoch: 100 Average test loss: 0.2235
====> Epoch: 100 Average test loss: 0.4618
====> Epoch: 100 Average test loss: 0.6891
====> Epoch: 100 Average test loss: 0.9145
====> Epoch: 100 Average test loss: 1.1418
====> Epoch: 100 Average test loss: 1.3655
====> Epoch: 100 Average test loss: 1.6169
====> Epoch: 100 Average test loss: 1.8945
====> Epoch: 100 Average test loss: 2.1299
====> Epoch: 100 Average test loss: 2.3795
====> Epoch: 100 Average test loss: 2.6286
====> Epoch: 100 Average test loss: 2.8770
====> Epoch: 100 Average test loss: 3.1441
====> Epoch: 100 Average test loss: 3.4321
====> Epoch: 100 Average test loss: 3.6678
====> Epoch: 100 Average test loss: 3.9052
====> Epoch: 100 Average test loss: 4.1738
====> Epoch: 100 Average test loss: 4.3865
```

In [22]:
```python
train_end_time =time.time()
```

In [23]:
```python
train_time = train_end_time - train_start_time
print("Total Training Time for",epochs, "epochs:", round(train_time) , "seconds")
```

```
Total Training Time for 2000 epochs: 7762 seconds
```

In [24]:
```python
scaler = trainloader.dataset.standardizer
```

In [25]:
```python
with torch.no_grad():
    for batch_idx, data in enumerate(testloader):
        data = data.to(device)
        optimizer.zero_grad()
        reconstructed_data, mu, logvar = model(data) # a vector of means, μ, and another vector of standard deviations, σ.
```

In [26]:
```
reconstructed_data.size()
```

Out[26]:
```
torch.Size([451, 30])
```

In [27]:
```
sigma = torch.exp(logvar/2)
```

In [28]:
```
mu[1], sigma[1]
```

Out[28]:
```
(tensor([-3.4837e-04, -1.1919e+00, -5.7458e-02]),
 tensor([1.0002, 0.1018, 0.1291]))
```

In [29]:
```
mu.mean(axis=0)
```

Out[29]:
```
tensor([ 0.0003,  0.0020, -0.0041])
```

In [30]:
```
 sigma.mean(axis=0)
```

Out[30]:
```
tensor([1.0000, 0.1468, 0.2148])
```

In [31]:
```
# sample z from q
synthetic_data_size = 284807
q = torch.distributions.Normal(mu.mean(axis=0), sigma.mean(axis=0))
z = q.rsample(sample_shape=torch.Size([synthetic_data_size])) # q-->latent matrix | Sampling z in VAE
```

In [32]:
```
z.shape,z
```

Out[32]:
```
(torch.Size([284807, 3]),
 tensor([[ 0.3545,  0.0731,  0.1653],
         [-0.2588,  0.0064, -0.0070],
         [ 0.8590,  0.0331,  0.0838],
         ...,
         [ 0.1314,  0.0511, -0.2231],
         [-1.5953, -0.1105,  0.0707],
         [-0.1516,  0.2999,  0.1075]]))
```

In [33]:
```
with torch.no_grad():pred = model.decode(z).cpu().numpy()
pred # predicted values
```

Out[33]:
```
array([[-0.01484537,  0.25238967,  0.6424196 , ...,  0.12296152,
        -0.1916731 ,  0.8117676 ],
       [ 0.55061555, -0.10713053, -0.4920885 , ...,  0.04719353,
        -0.10092729,  0.7340698 ],
       [ 0.28929353,  0.00336981, -0.05528414, ...,  0.12634945,
        -0.13681   ,  0.7548218 ],
       ...,
       [ 0.39189148, -0.2234497 , -0.1039629 , ...,  0.06050539,
        -0.13704401,  0.71447754],
       [ 0.24216676, -0.01840353, -0.04524434, ...,  0.21961164,
        -0.25507182,  0.83496094],
       [ 0.34177613,  0.08081293, -0.1050787 , ..., -0.4050517 ,
        -0.06141329,  0.6227417 ]], dtype=float32)
```

In [34]:
```
synthetic_data = scaler.inverse_transform(pred)
synthetic_data.shape
```

Out[34]:
```
(284807, 30)
```

In [35]:
```
synthetic_data = pd.DataFrame(synthetic_data, columns = columns)
synthetic_data.head(20)
```

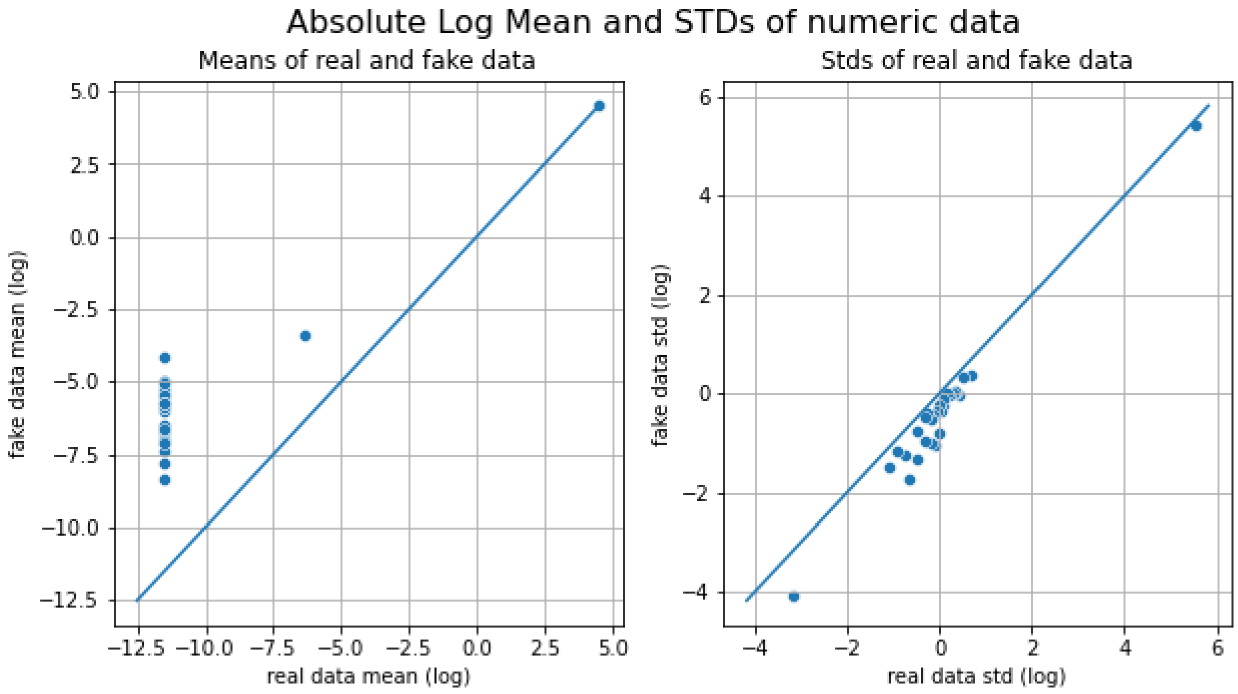
Out[35]:

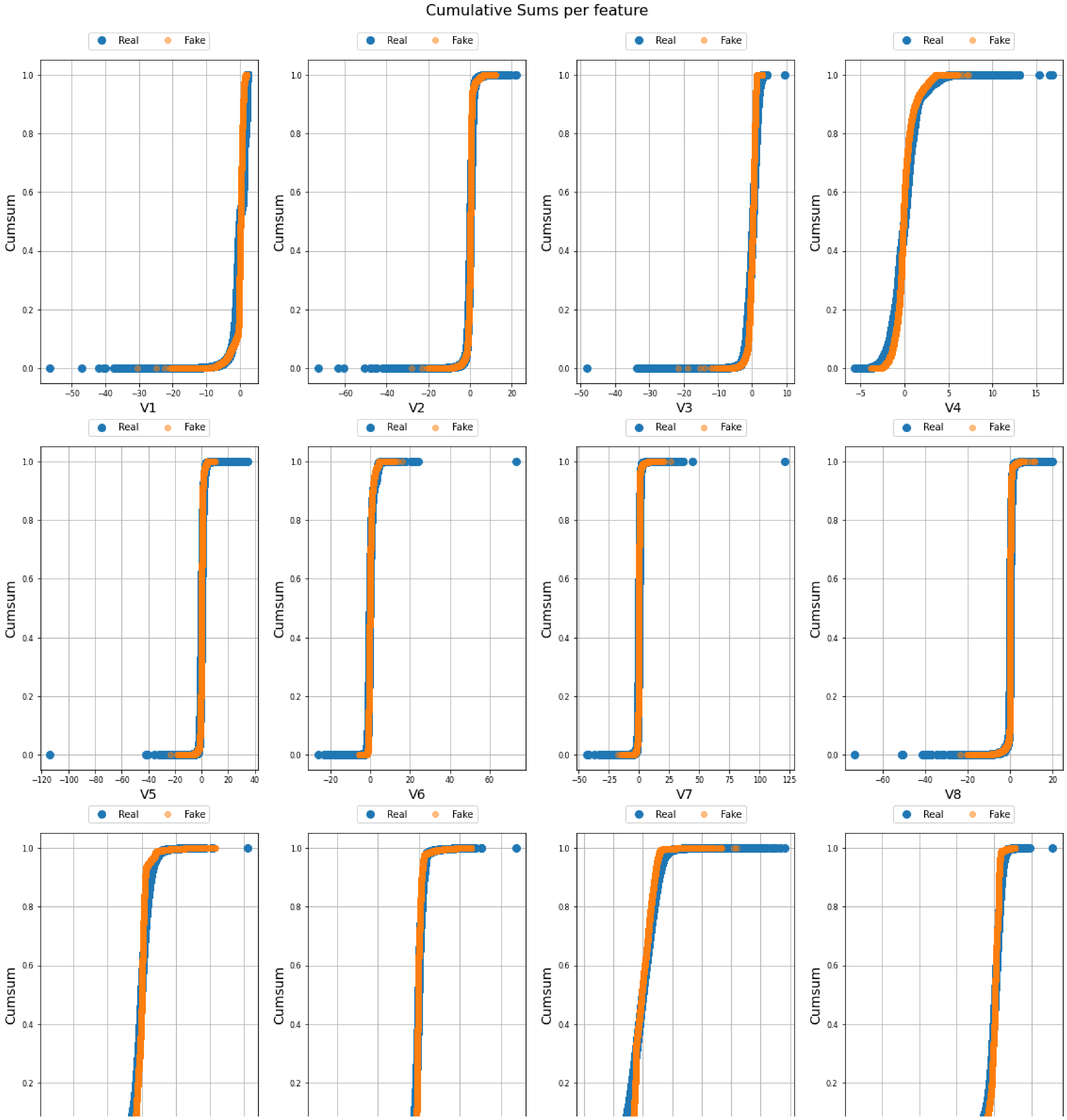
	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V21	V22	V23	V24
0	-0.030602	0.418624	0.974931	0.959757	0.009973	-0.054692	0.210411	0.153848	-0.031649	-0.204386	...	0.048854	0.233896	0.003994	0.098860
1	1.078240	-0.181012	-0.743187	0.049971	0.209390	-0.585835	0.195955	-0.052234	0.586924	-0.260113	...	0.163829	0.502646	-0.036751	-0.125002
2	0.565800	0.003289	-0.081683	0.265607	0.115479	-0.445406	0.182086	0.002341	0.396881	-0.324847	...	0.108658	0.387974	-0.033867	0.024618
3	-2.505154	2.268930	-0.394400	-0.123339	0.344359	2.294253	-4.728662	-8.792536	-0.793346	-0.956390	...	-2.942264	0.728317	-0.690392	-0.094305
4	-0.203337	-0.591250	-1.266944	-1.418267	3.827471	4.260494	-1.546661	-0.275008	0.469770	-0.612009	...	-0.722724	-0.154449	0.144457	1.132957
5	-0.254810	-0.273160	0.939553	-1.238551	-0.408545	-0.170572	-0.136938	0.152519	-0.641571	0.018446	...	0.207197	0.422584	0.008617	0.188607
6	0.769540	-1.551693	0.259168	-2.112123	-0.436174	1.075957	-1.338953	0.329851	-1.596198	1.129621	...	-0.148317	-0.100204	0.052451	0.086733
7	-0.235930	-0.230763	0.753444	-0.175164	0.090774	0.502533	-0.105852	0.547952	0.108082	-0.351145	...	-0.051436	-0.102162	-0.032319	-0.273030
8	0.276898	-0.173869	0.556815	-0.946122	-0.763508	-0.628725	-0.225931	0.036170	-0.544115	0.052018	...	0.124275	0.291242	0.069427	0.155563
9	0.486398	0.168587	-0.121543	0.361280	0.338876	-0.205306	0.188836	0.119312	0.150753	-0.206046	...	0.112486	0.392031	-0.052190	-0.089536
10	-0.171381	0.484090	0.550623	0.193778	0.243152	0.007093	0.203905	0.230649	-0.130176	-0.311636	...	-0.132071	-0.068611	-0.027779	0.057731
11	-6.169681	5.213397	-3.933545	-1.165274	-2.042171	-1.396624	-1.254021	3.117203	1.681513	1.794170	...	-0.170428	-0.119136	0.542279	-0.142010
12	0.417158	0.073372	0.164471	0.373113	0.073615	-0.374493	0.167984	0.026645	0.320808	-0.327854	...	0.089980	0.348221	-0.016977	0.059511
13	0.408577	-0.018702	-0.150884	-0.118539	0.079073	-0.907186	0.104593	-0.056970	0.161179	-0.264482	...	-0.203062	-0.384461	0.194066	0.103146
14	0.174266	0.231572	0.456944	0.706086	0.264000	-0.156053	0.201073	0.153719	-0.005376	-0.186889	...	0.103170	0.314029	-0.053101	0.010881
15	1.262154	-0.768404	-0.425106	-0.362269	-0.137982	-0.228964	-0.266408	0.068544	-0.202482	0.403123	...	0.079689	0.118653	0.002588	-0.384218
16	-5.867027	2.678400	0.780549	-2.096145	3.519388	-0.447244	5.570286	-4.728125	6.809417	9.524776	...	-1.617824	-0.437543	-0.045833	-0.114836
17	0.700190	0.235111	-0.067197	2.023877	1.099232	2.189086	-0.178510	0.675691	-0.377686	0.664753	...	0.280321	0.631397	-0.095442	-0.568869
18	0.366020	0.726324	-0.660009	0.017941	0.453604	-0.992755	0.577518	-0.040847	0.098313	-0.693667	...	-0.063734	-0.102689	0.085543	0.120883
19	0.759007	-1.471549	0.164099	-2.121075	-0.267988	1.312158	-1.359754	0.371628	-1.529812	1.080820	...	-0.123667	-0.063948	0.040006	0.156312

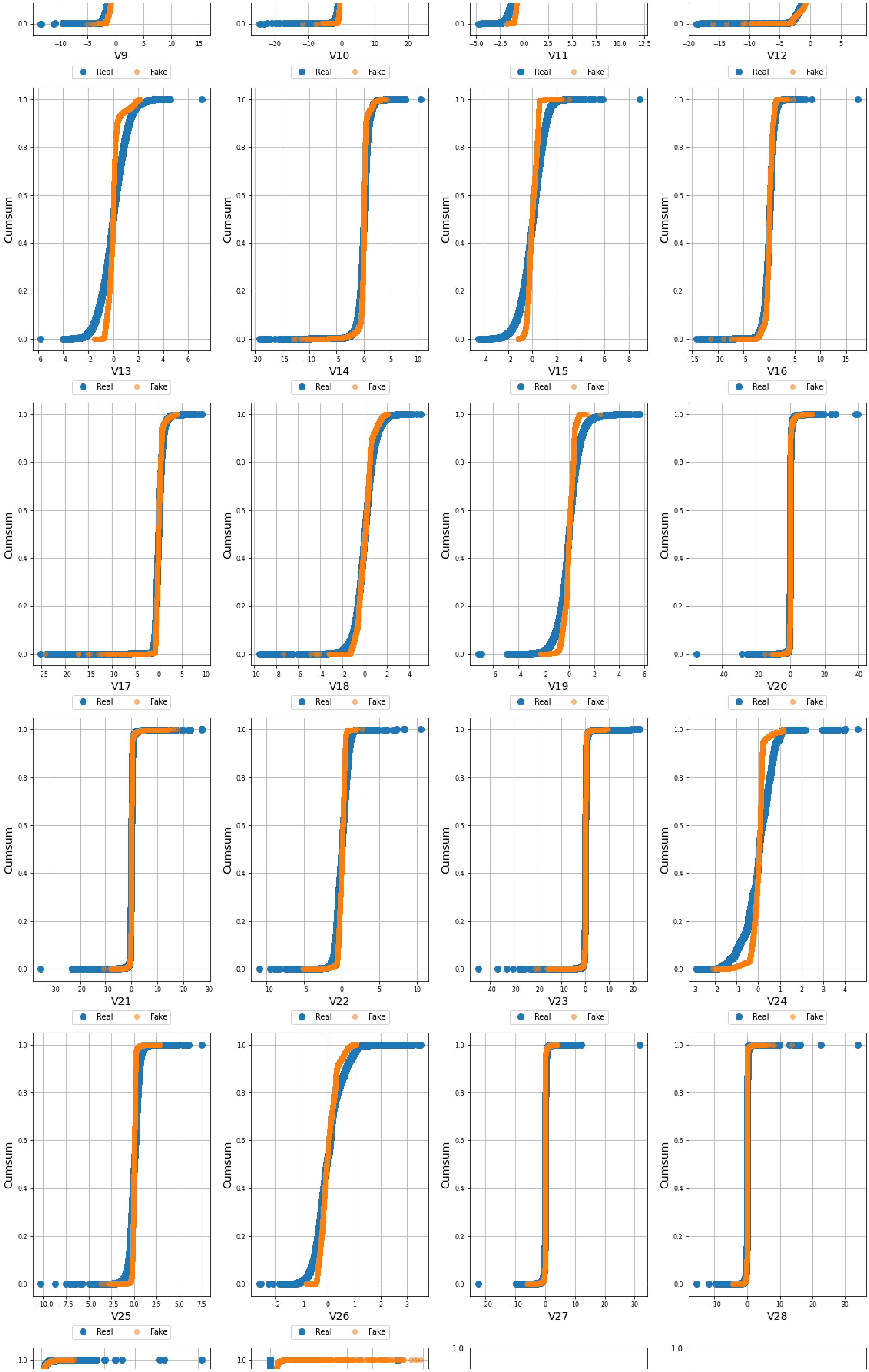
20 rows × 30 columns

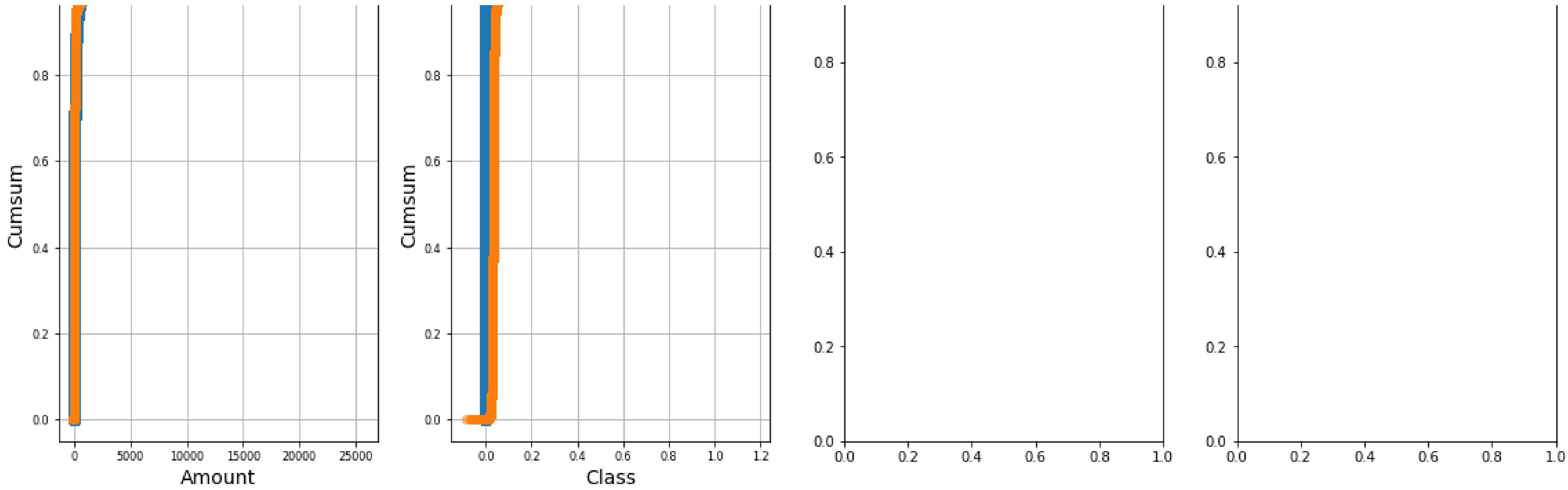
In [36]:
```
from table_evaluator import load_data, TableEvaluator

print(len(data_original), len(synthetic_data))
table_evaluator =  TableEvaluator(data_original, synthetic_data)
table_evaluator.visual_evaluation()
```

284807 284807

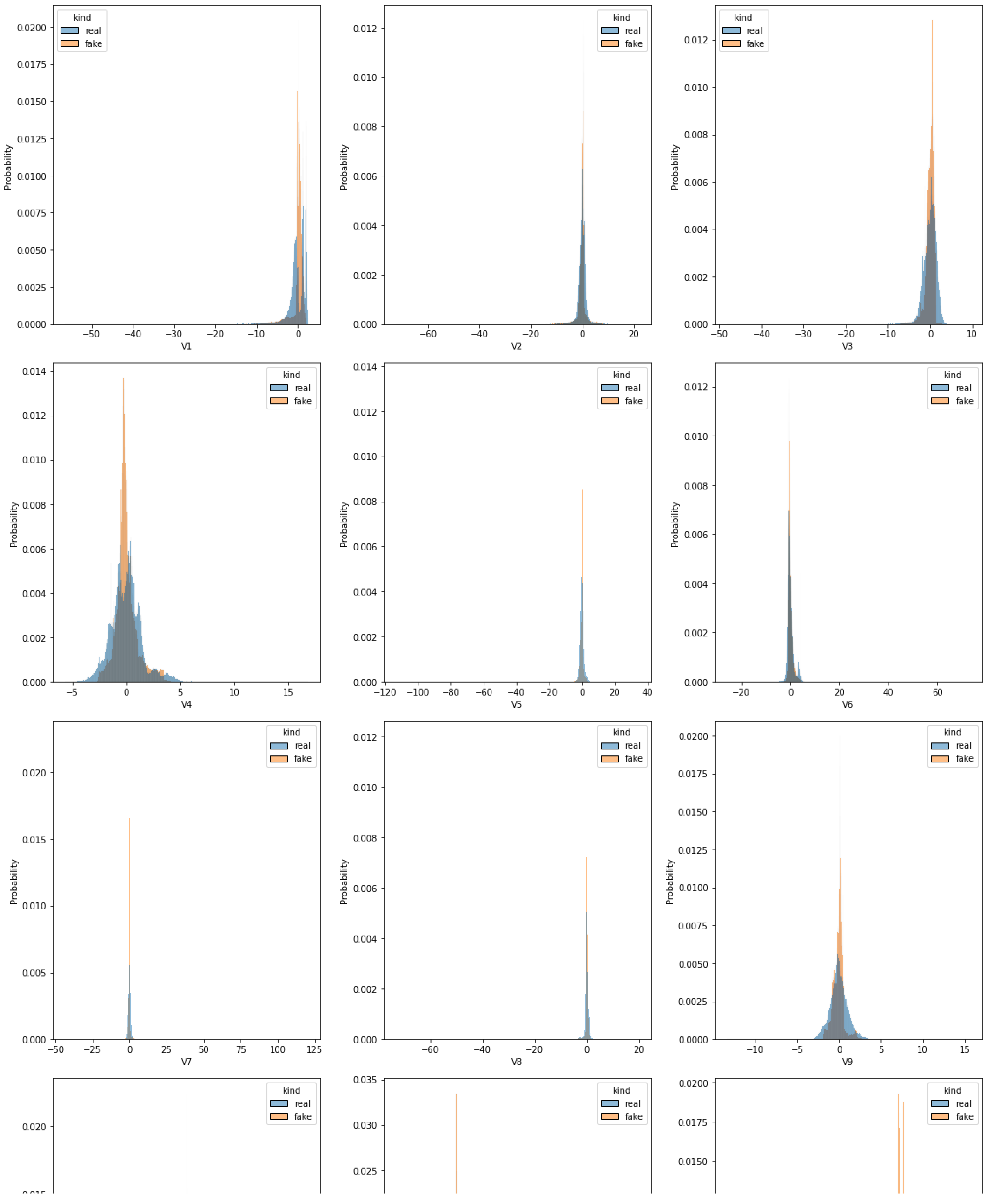


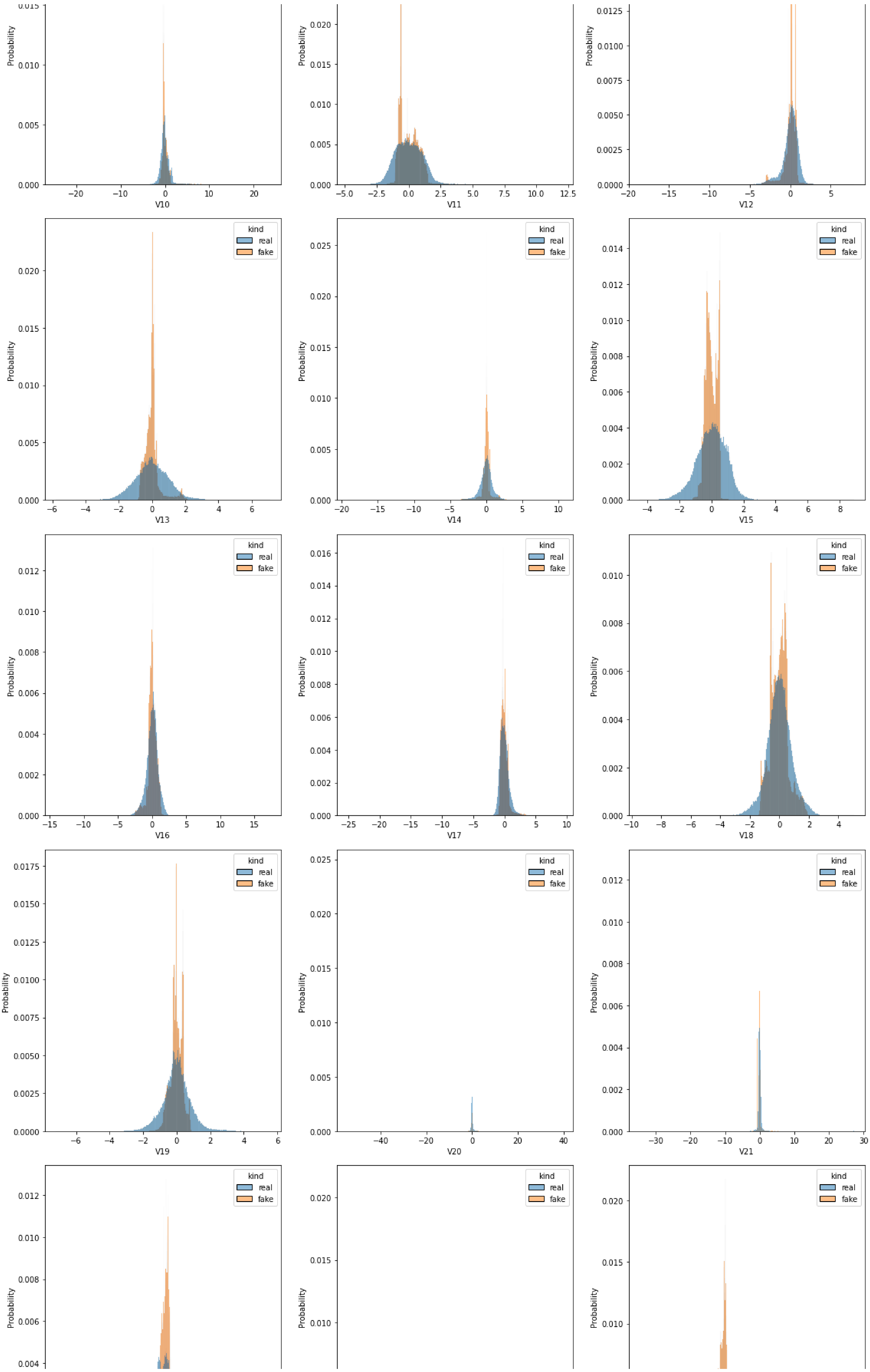



V9
V10
V11
V12
V13
V14
V15
V16
V17
V18
V19
V20
V21
V22
V23
V24
V25
V26
V27
V28
Cumsum
Real
Fake

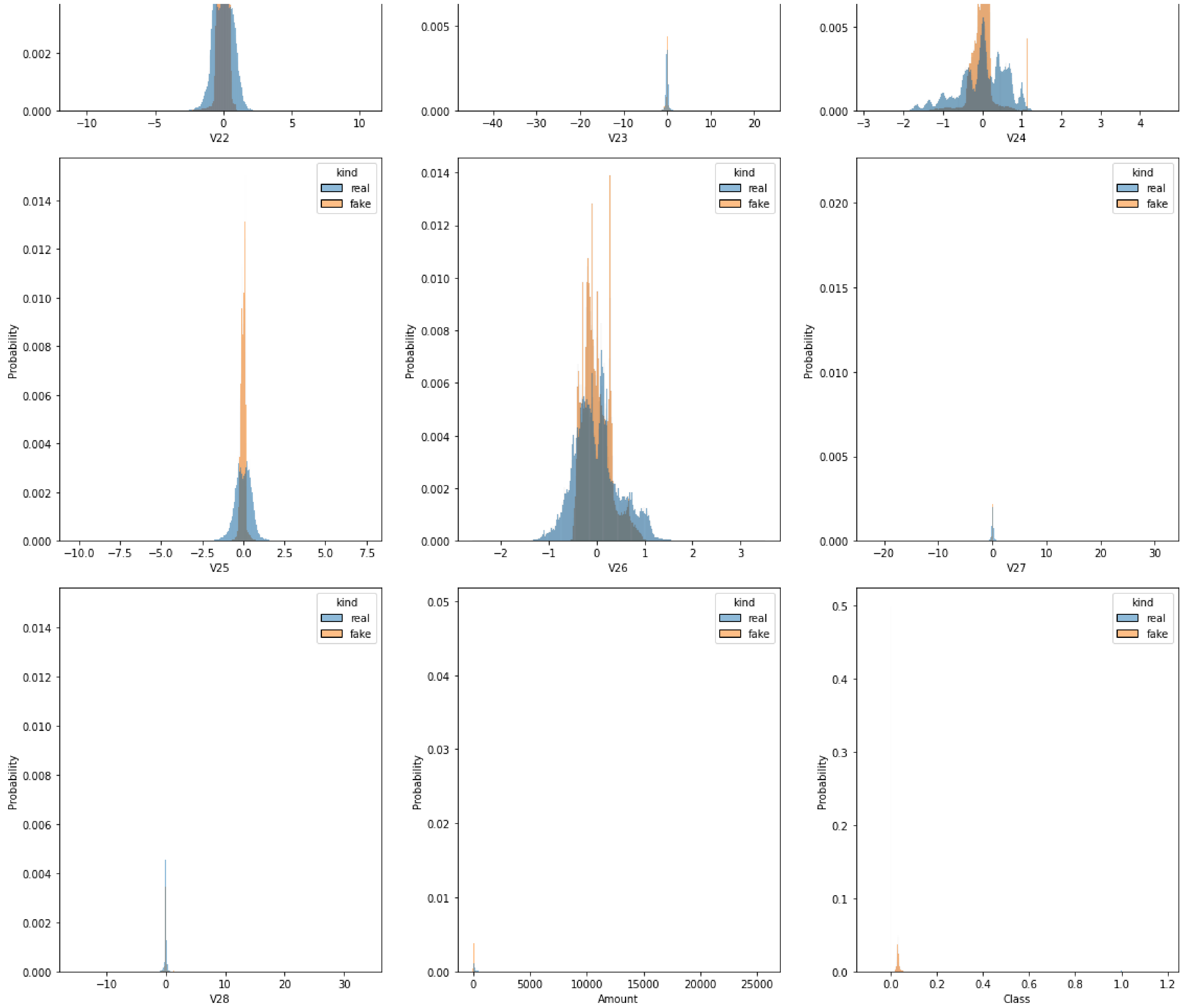


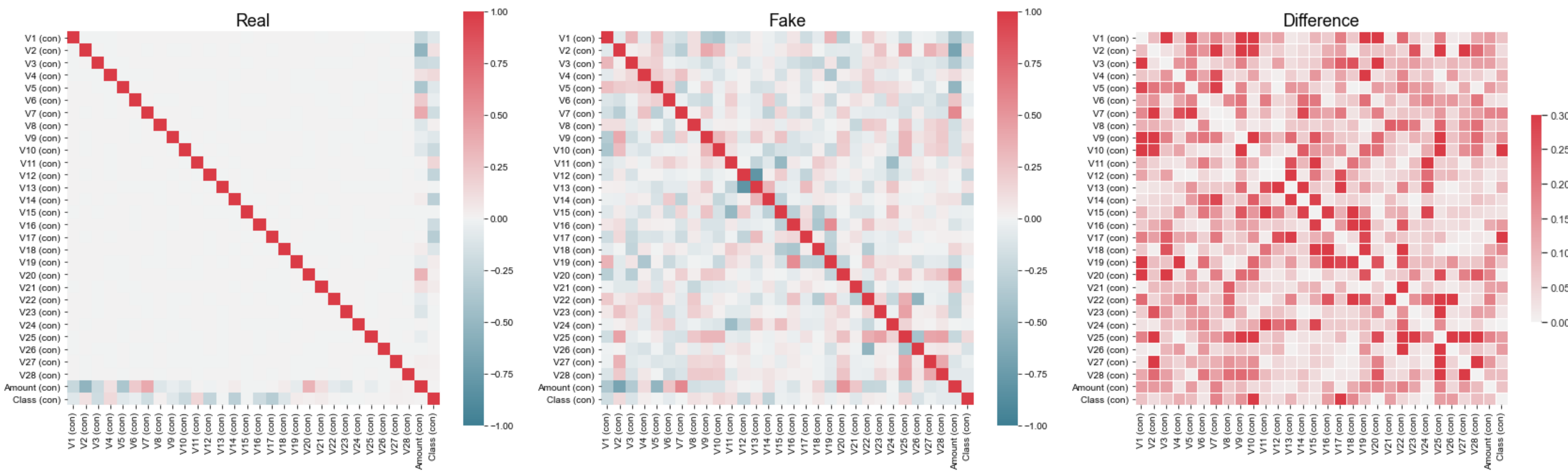
Distribution per feature

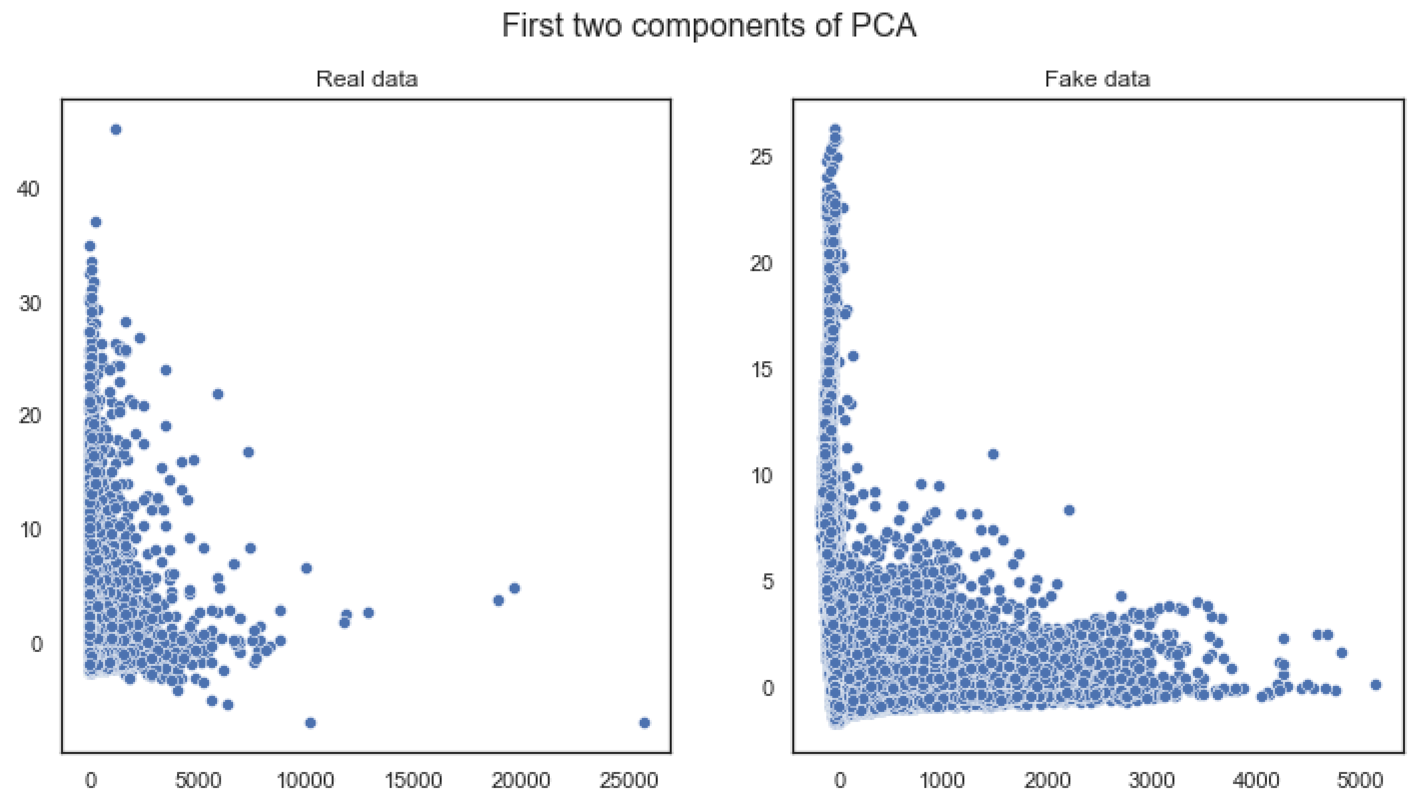




Probability
V10
V11
V12
V13
V14
V15
V16
V17
V18
V19
V20
V21
kind
real
fake


V22
V23
V24
Probability
kind
real
fake
V25
V26
V27
V28
Amount
Class


Real
Fake
Difference



In [54]:
```
VAE_Synthetic_Data = datetime.datetime.now().strftime("VAE_Synthetic_Data %d-%m-%Y(%H.%M Hrs).csv")
synthetic_data.to_csv(VAE_Synthetic_Data)
```

In [39]:
```
end = time.time()
total_time = end - start
print("End Time:" ,datetime.datetime.fromtimestamp(end).strftime('%Y-%m-%d %H:%M:%S'))
print("Total Run Time:", round(total_time/3600) , "Hours")
```

```
End Time: 2022-08-03 19:19:05
Total Run Time: 18 Hours
```

+===+

Notes:

https://harvard-iacs.github.io/2019-CS109B/labs/lab10/VAE/ (https://harvard-iacs.github.io/2019-CS109B/labs/lab10/VAE/)

https://jhui.github.io/2017/03/06/Variational-autoencoders/ (https://jhui.github.io/2017/03/06/Variational-autoencoders/)

https://medium.com/@olivia.liang032/how-to-measure-statistical-similarity-on-tabular-data-demonstrated-using-synthetic-data-66a1aa60084d (https://medium.com/@olivia.liang032/how-to-measure-statistical-similarity-on-tabular-data-demonstrated-using-synthetic-data-66a1aa60084d)

https://towardsdatascience.com/visualising-high-dimensional-datasets-using-pca-and-t-sne-in-python-8ef87e7915b (https://towardsdatascience.com/visualising-high-dimensional-datasets-using-pca-and-t-sne-in-python-8ef87e7915b)

It is highly recommended to use another dimensionality reduction method (e.g. PCA for dense data or TruncatedSVD for sparse data) to reduce the number of dimensions to a reasonable amount (e.g. 50) if the number of features is very high.

https://colab.research.google.com/github/tvhahn/Manufacturing-Data-Science-with-Python/blob/master/Metal%20Machining/1.B_building-vae.ipynb#scrollTo=t6mNH0b6RnlU (https://colab.research.google.com/github/tvhahn/Manufacturing-Data-Science-with-Python/blob/master/Metal%20Machining/1.B_building-vae.ipynb#scrollTo=t6mNH0b6RnlU)

In [ ]: