Task 3

Contains solutions for both part A and B

By : Naimish Mani B

Table Of Contents

Preface

Initially, after extracting the dataset, I realised that I don't have enough memory to load the entire dataset. Hence, in an attempt to circumvent this issue, I ended up implementing my own generator to yield the data one line at a time, and also wrote my own classifiers to work with this method.

There were a couple of issues with this approach, though. For one, it loads elements into memory sequentially. This is a problem wherever we use batches, because we cannot gurantee there being a good, uniform split of elements from each class in the batch. This might lead to accuracy issues. But it can be mitigated by performing random access on the file, by maybe using seek() to read a specific line in the file, and then randomly reading the lines to make the training set. This might ensure uniformity across batches. But, not all algorithms work on batches. And since I was implementing them on my own, there's more scope for failure here.

So instead, I ended up using Colab for this task, after getting permission to do so from Shravan Chaudhrai.

NOTE:

There were times when the notebook got disconnected due to internet stability issues. In those instances, I let the initial "Understanding the Dataset" code stay in it's prior state without re-running the cells, because those plots and values are constant for the dataset.

Libraries Used

- Pandas
- Seaborn
- Matplotlib
- Scikit-Learn
- NumPy
- Time

Setup

We start off by downloading the dataset using 'gdown', and then extracting it with gzip. After verifying that the datsaet has been extracted properly, we can now work on understanding the dataset.

In [1]:
```
# Run this to figure out where to restart from each time instance crashes.
!ls
```

```
HIGGS_6M.csv  sample_data
```

In [1]:
```
# Download the dataset
!gdown --id 1Nl-RU5HggCSWWqINN4gbxHAVzER6T2Po

# Extract it
!gzip -d /content/HIGGS_6M.csv.gz

# Verify if dataset is there
!ls

# Try reading a line from it
with open('HIGGS_6M.csv', 'r') as fh:
  print(fh.readline())
```

```
Downloading...
From: https://drive.google.com/uc?id=1Nl-RU5HggCSWWqINN4gbxHAVzER6T2Po
To: /content/HIGGS_6M.csv.gz
1.56GB [00:10, 142MB/s]
HIGGS_6M.csv  sample_data
1.0000000000000000e+00,8.69293128906250000e-01,-6.35081827640533447e-01
,2.25690260529518127e-01,3.27470064163208007e-01,-6.89993202686309814e-0
1,7.54202246665954589e-01,-2.48573139309883117e-01,-1.09206390380859375e
+00,0.00000000000000000e+00,1.37499213218688965e+00,-6.53674185276031494
e-01,9.30349111557006836e-01,1.10743606090545654e+00,1.13890433311462402
3e+00,-1.57819831371307373e+00,-1.04698538780212402e+00,0.0000000000000000
00e+00,6.57929539680480957e-01,-1.04545699432492256e-02,-4.57671694457530
9753e-02,3.10196137428283691e+00,1.35376000404357910e+00,9.79563117027282
7148e-01,9.78076159954071045e-01,9.20004844665527344e-01,7.21657454967498
7793e-01,9.88750934600830078e-01,8.76678347587585449e-01
```

Understanding the Dataset

Now that we've downloaded the dataset, our next task is to load it. For that, we'll be using Pandas. We'll also be visualising various features of the dataset too, using seaborn. So let's import the required libraries first.

In [2]:
```
import pandas as pd
import seaborn as sn
import matplotlib.pyplot as plt

# Loads the dataframe
df = pd.read_csv("HIGGS_6M.csv", header=None)

df.head()
```

Out[2]:

	0	1	2	3	4	5	6	7	
0	1.0	0.869293	-0.635082	0.225690	0.327470	-0.689993	0.754202	-0.248573	-1.0920
1	1.0	0.907542	0.329147	0.359412	1.497970	-0.313010	1.095531	-0.557525	-1.5882
2	1.0	0.798835	1.470639	-1.635975	0.453773	0.425629	1.104875	1.282322	1.3816
3	0.0	1.344385	-0.876626	0.935913	1.992050	0.882454	1.786066	-1.646778	-0.9423
4	1.0	1.105009	0.321356	1.522401	0.882808	-1.205349	0.681466	-1.070464	-0.9218

In [ ]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6000000 entries, 0 to 5999999
Data columns (total 29 columns):
 #   Column  Dtype  
---  ------  -----  
 0   0       float64
 1   1       float64
 2   2       float64
 3   3       float64
 4   4       float64
 5   5       float64
 6   6       float64
 7   7       float64
 8   8       float64
 9   9       float64
 10  10      float64
 11  11      float64
 12  12      float64
 13  13      float64
 14  14      float64
 15  15      float64
 16  16      float64
 17  17      float64
 18  18      float64
 19  19      float64
 20  20      float64
 21  21      float64
 22  22      float64
 23  23      float64
 24  24      float64
 25  25      float64
 26  26      float64
 27  27      float64
 28  28      float64
dtypes: float64(29)
memory usage: 1.3 GB
```

Now, let's see what the range of the columns in the dataset are.

In [ ]:

```
df.describe().T
```

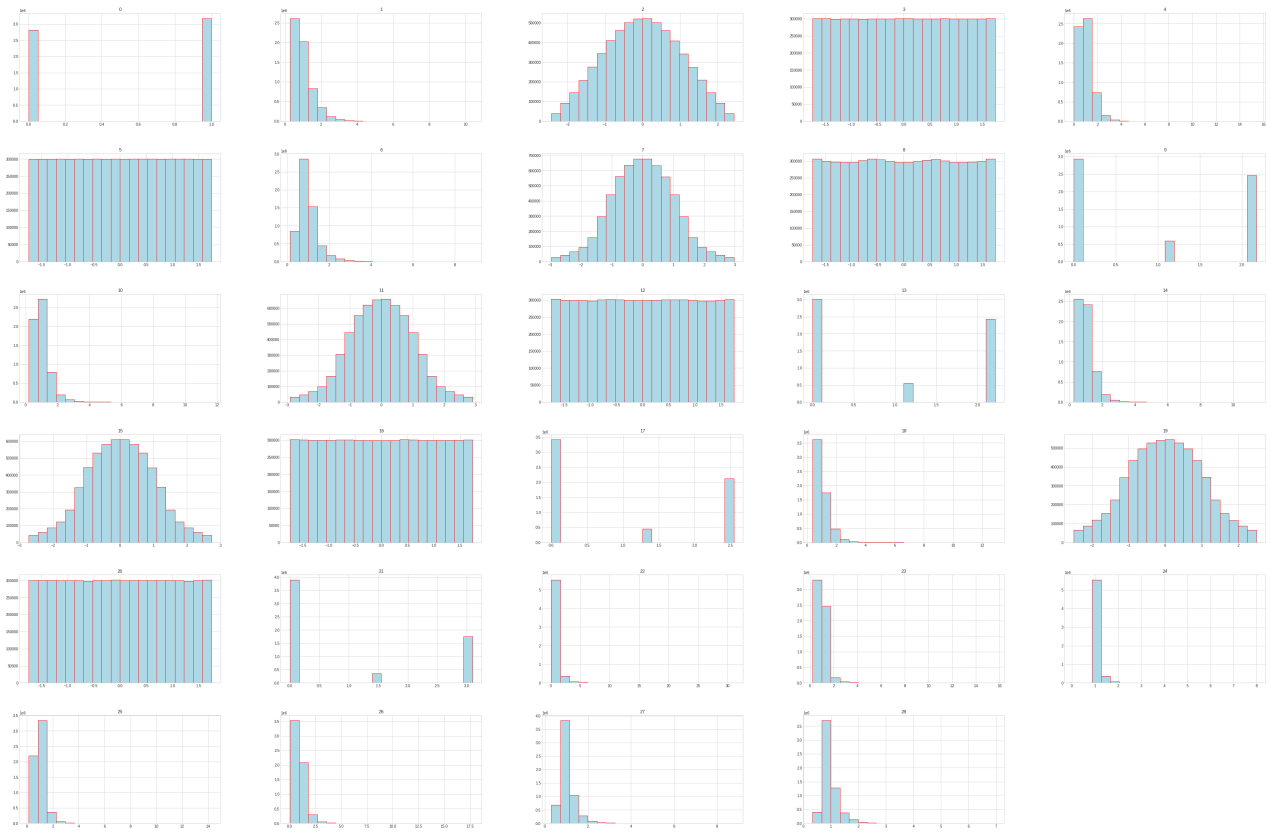
Out[ ]:

	count	mean	std	min	25%	50%	75%	ma
0	6000000.0	0.529724	0.499116	0.000000	0.000000	1.000000	1.000000	1.00000
1	6000000.0	0.991139	0.564896	0.274697	0.590387	0.853188	1.236043	10.39601
2	6000000.0	-0.000006	1.008915	-2.434976	-0.739296	0.000920	0.738214	2.43486
3	6000000.0	0.000363	1.006282	-1.742508	-0.871376	0.000971	0.870994	1.74323
4	6000000.0	0.998190	0.599945	0.000394	0.576591	0.891350	1.292679	15.39682
5	6000000.0	0.000032	1.006247	-1.743944	-0.871147	0.000378	0.871482	1.74325
6	6000000.0	0.990886	0.475080	0.137686	0.678993	0.894636	1.170740	8.84861
7	6000000.0	-0.000536	1.009190	-2.969725	-0.687245	-0.001016	0.686204	2.96967
8	6000000.0	-0.000089	1.005880	-1.741237	-0.868096	-0.000496	0.868313	1.74145
9	6000000.0	1.000497	1.027797	0.000000	0.000000	1.086538	2.173076	2.17307
10	6000000.0	0.992473	0.500014	0.188981	0.656335	0.889886	1.201749	11.64708
11	6000000.0	0.000372	1.009539	-2.913090	-0.694472	0.000060	0.695564	2.91321
12	6000000.0	-0.000203	1.006107	-1.742372	-0.870179	-0.000203	0.869318	1.74317
13	6000000.0	0.999499	1.049401	0.000000	0.000000	0.000000	2.214872	2.21487
14	6000000.0	0.992142	0.487452	0.263608	0.650853	0.897249	1.221447	11.42381
15	6000000.0	0.000128	1.008500	-2.729663	-0.698898	0.000173	0.700154	2.73000
16	6000000.0	-0.000210	1.006410	-1.742069	-0.871689	-0.000197	0.871395	1.74288
17	6000000.0	1.000315	1.193803	0.000000	0.000000	0.000000	2.548224	2.54822
18	6000000.0	0.985946	0.505809	0.365354	0.617281	0.867990	1.220930	12.88256
19	6000000.0	0.000055	1.007736	-2.497265	-0.713357	0.000372	0.714102	2.49800
20	6000000.0	0.000010	1.006360	-1.742691	-0.872034	-0.000264	0.871051	1.74337
21	6000000.0	0.999507	1.400049	0.000000	0.000000	0.000000	3.101961	3.10196
22	6000000.0	1.034293	0.673313	0.083492	0.790630	0.894954	1.024868	31.07619
23	6000000.0	1.024772	0.380471	0.198676	0.846215	0.950652	1.083482	15.63785
24	6000000.0	1.050405	0.164043	0.083049	0.985749	0.989775	1.020313	7.99273
25	6000000.0	1.009734	0.397441	0.132006	0.767542	0.916531	1.142169	14.26243
26	6000000.0	0.973029	0.525559	0.048125	0.673866	0.873443	1.138591	17.76285
27	6000000.0	1.033050	0.365336	0.295112	0.819417	0.947368	1.140515	8.77991
28	6000000.0	0.959765	0.313336	0.330721	0.770378	0.871874	1.059126	6.96690

Now that we have a rough understanding of the range of the dataset, the next step is to see the variation within each column in the dataset. That has been plotted below.

Plot all columns

In [ ]:
```
plt.style.use('seaborn-whitegrid')

# Plot all in same plot
df.hist(bins=20, figsize=(60, 40), color='lightblue', edgecolor = 'red')
plt.show()
```



From the above plots, we can see clearly that some columns follow a binomial distribution, some follow a normal distribution and others a gaussian distribution. There are also some columns which seem to have just three distinct values (9, 13, 17 and 21). Next, let's see the correlation coefficients between all columns through a correlation matrix.

For good measure, we'll also calculate the skewness of the columns down below.

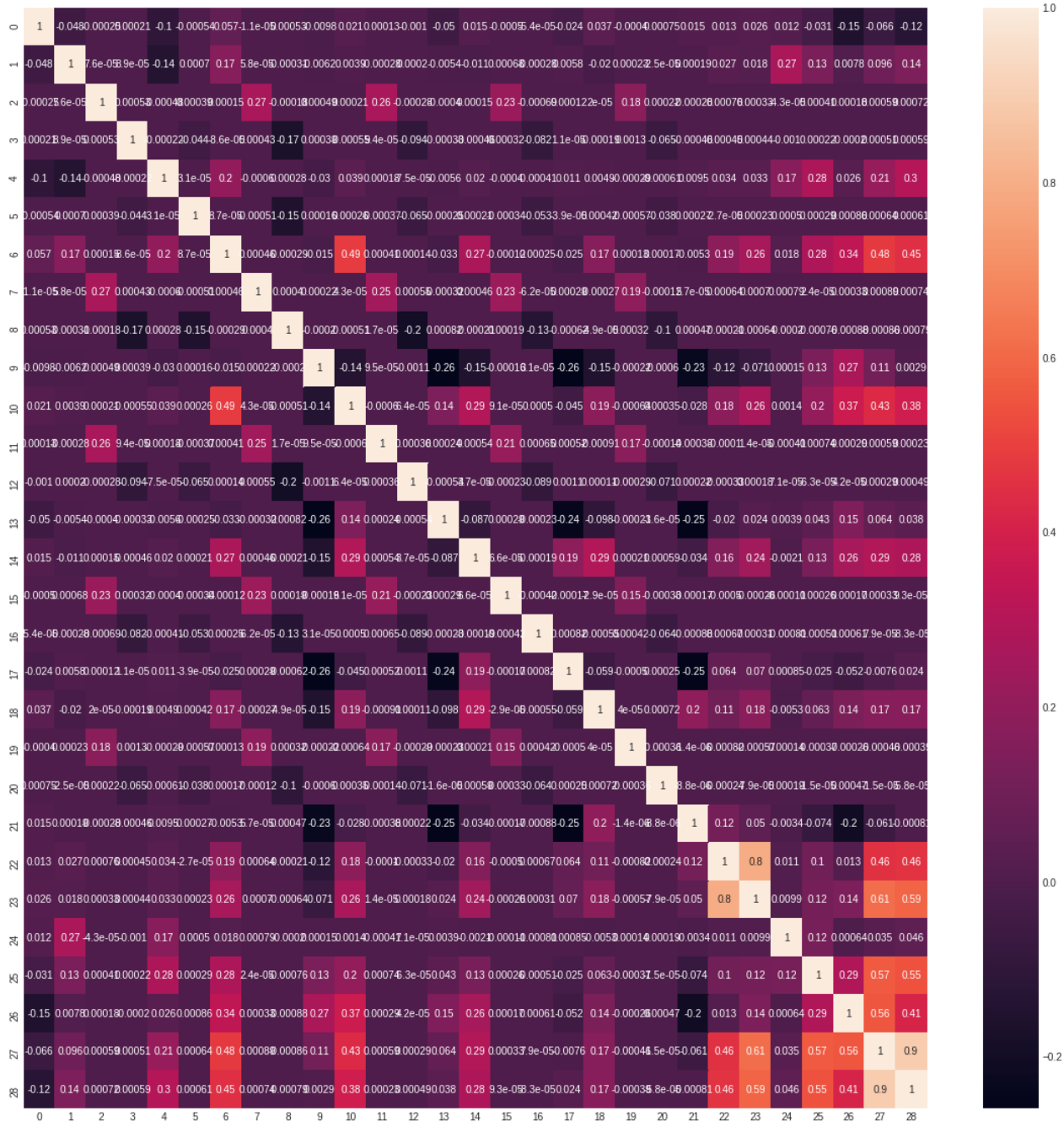
In [ ]:
```
skewValue = df.skew()
print("skewValue of dataframe attributes: ", skewValue)
```

```
skewValue of dataframe attributes:  0    -0.119107
1     1.750431
2    -0.000024
3    -0.000612
4     1.491011
5    -0.001000
6     1.906270
7     0.000307
8     0.000359
9     0.158172
10    1.973255
11    0.000865
12    0.000418
13    0.195109
14    1.702261
15   -0.001123
16    0.000909
17    0.438865
18    1.723900
19   -0.000511
20    0.000227
21    0.759401
22    6.462415
23    4.969472
24    4.581439
25    2.853695
26    2.433943
27    2.690596
28    2.547707
dtype: float64
```

Draw Correlation Matrix

In [ ]:

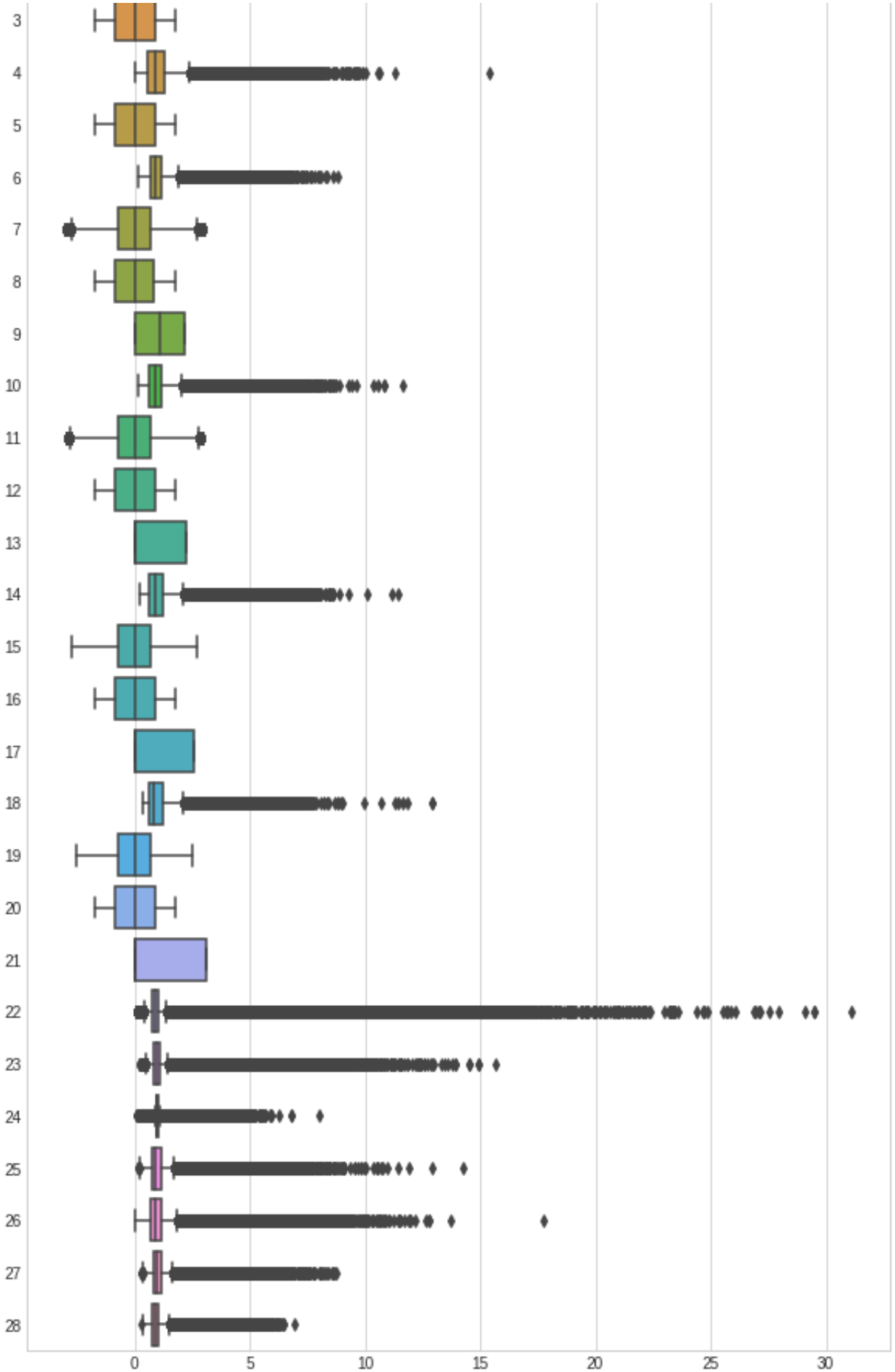
```
corrMatrix = df.corr()

plt.figure(figsize=(20,20))
sn.heatmap(corrMatrix, annot=True)
plt.show()
```



From the correlation matrix, it becomes very clear that almost all of the original features generated (the first 22 columns) are clearly independent of eachother. This implies that we'll have to use AutoEncoders for dimensionality reduction, and traditional techniques might not offer much of a solution.

In [ ]:

```
plt.figure(figsize=(10, 18))
ax = sn.boxplot(data=df, orient='h')
```


3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
0
5
10
15
20
25
30

Pair Plot Analysis

To identify the kind of classifiers and dimensionality reduction techniques that would be best suited for the dataset, we can perform pair-plot analysis on the dataset.

But the dataset has 6M rows, which is alot. But from the above plots, it is quite apparant that the columns tend to follow typical distributions (Gaussian, Normal and Binomial) to the t. This means that we can sample a relatively small subset for further tasks, without suffering from too much a loss in terms of accuracy.

So, from the 6M rows, we randomly sample 120k rows to make a 'small' version of the dataset. This small dataset is then split into :

- 100k rows (training data)
- 20k rows (testing data)

Also, we drop the generated final 7 rows from the dataset for now, and only work with the essential data.

In [4]:
```
# Randomly Sample 120k rows
shortData = df.sample(120000)
# Drop final 7 columns
shortData = shortData.drop(shortData.columns[28],axis=1)
shortData = shortData.drop(shortData.columns[27],axis=1)
shortData = shortData.drop(shortData.columns[26],axis=1)
shortData = shortData.drop(shortData.columns[25],axis=1)
shortData = shortData.drop(shortData.columns[24],axis=1)
shortData = shortData.drop(shortData.columns[23],axis=1)
shortData = shortData.drop(shortData.columns[22],axis=1)

# Print DF shape
print(shortData.shape)

# Show first five rows of sample data
shortData.head()
```

```
(120000, 22)
```

Out[4]:

	0	1	2	3	4	5	6	7
1814828	0.0	0.458072	-0.248416	1.408655	2.488697	-1.177061	0.694383	0.743637
2118867	0.0	0.370593	0.966123	1.625050	0.270872	1.111200	0.569339	0.480236
1203429	0.0	1.586506	-1.425944	-1.658724	0.569361	-1.569877	0.711513	0.989214
2079585	0.0	0.587825	-1.274005	1.142876	0.618646	-1.436803	0.570805	-0.517916
616804	1.0	1.475419	-0.592227	-0.606707	0.756023	-0.672388	0.653709	0.402008

But again, even this is too large to perform pair plot analysis on. So, we randomly sample just 100 values from this short version of the dataset, and work from there.

Now, let's perform pair plot analysis on this sample.

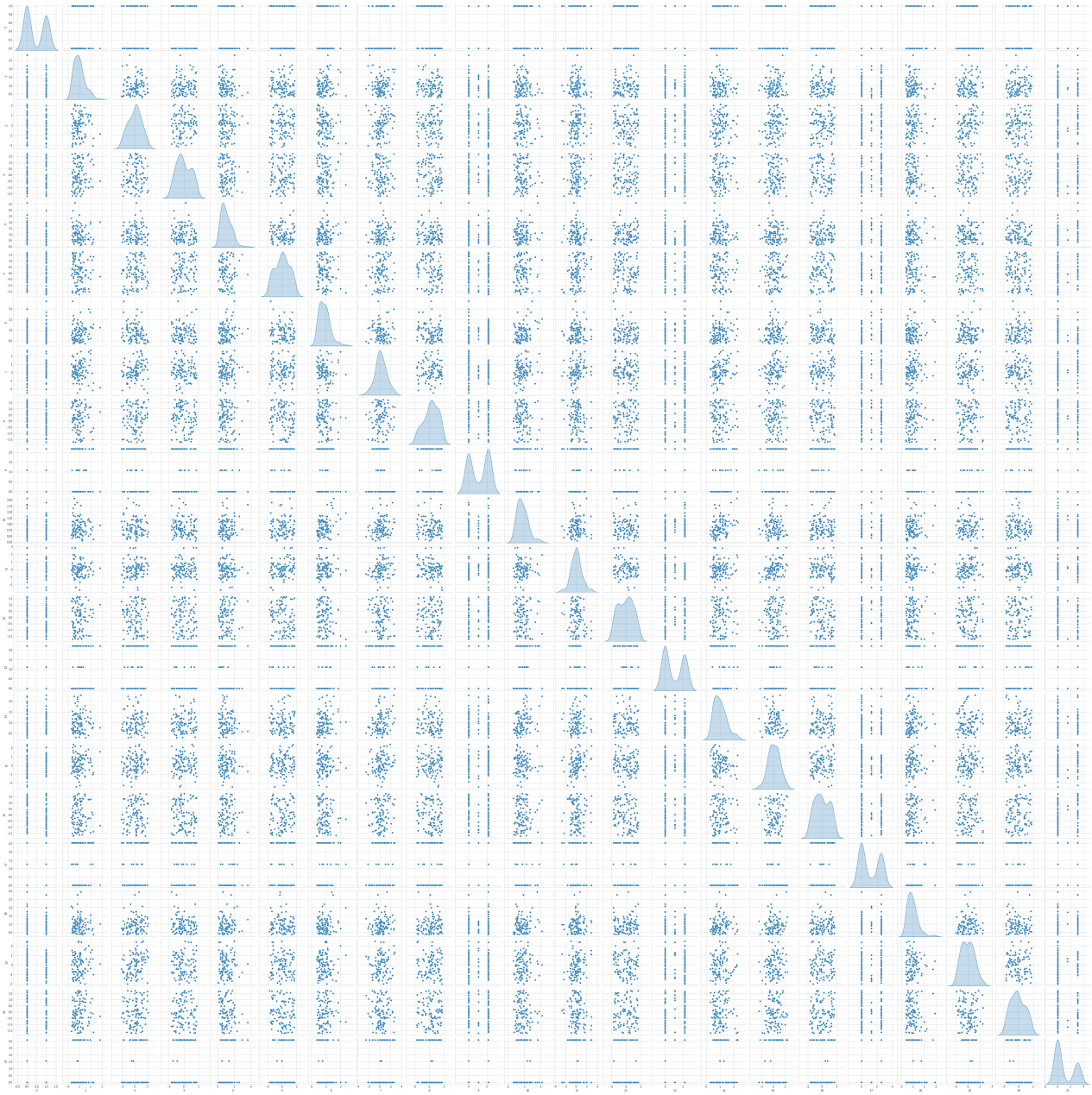
In [ ]:
```
pairPlotData = shortData.sample(100)
# Show head
pairPlotData.head()
```

Out[ ]:

	0	1	2	3	4	5	6	7
4379028	1.0	0.534204	0.702177	-1.014530	1.657363	-0.270533	0.961235	-1.862648
1013337	0.0	2.229966	-0.570800	1.555138	0.767058	0.486526	1.296975	-0.308977
2935225	0.0	0.566962	-0.021481	-1.139929	0.837163	0.640455	0.713345	0.812953
2565353	1.0	0.972144	0.957357	0.212928	0.696754	-1.107602	1.189153	0.626790
492671	1.0	0.427326	-1.058758	1.430294	0.183106	-0.958200	0.896285	0.670360

In [ ]:
```
# Plot it
sn.pairplot(pairPlotData, diag_kind="kde")
```

Out[ ]:
```
<seaborn.axisgrid.PairGrid at 0x7f35cb609ed0>
```



Next, let's split the data into features (x) and labels (y)

In [5]:

```
x = shortData.drop(shortData.columns[0],axis=1)
y = shortData[0]

print(x.head())
print(y.head())
```

```
                1         2         3   ...        19        20        21
1814828  0.458072 -0.248416  1.408655   ...  0.721597 -1.312634  0.000000
2118867  0.370593  0.966123  1.625050   ...  0.883165  0.583051  3.101961
1203429  1.586506 -1.425944 -1.658724   ... -1.084797  0.454312  0.000000
2079585  0.587825 -1.274005  1.142876   ... -1.249696  0.103052  0.000000
616804   1.475419 -0.592227 -0.606707   ...  0.161940  0.764507  0.000000

[5 rows x 21 columns]
1814828    0.0
2118867    0.0
1203429    0.0
2079585    0.0
616804     1.0
Name: 0, dtype: float64
```

Now, let's split the dataset `shortData` up into the training and testing sets.

In [5]:

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.2, random_state=3
)

print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(4800000, 21)
(1200000, 21)
(4800000,)
(1200000,)
```

Classification

Now that we have our training and testing datasets, we can move ahead with building classifiers with it.

We can try building the following classifiers with sklearn, and plot their corresponding ROC curves.

- Logistic Regression
- K Nearest Neighbors
- Decision Tree
- Random Forest
- Naive Bayes Classifier
- Multi Layer Perceptrons
- AdaBoost

In [3]:

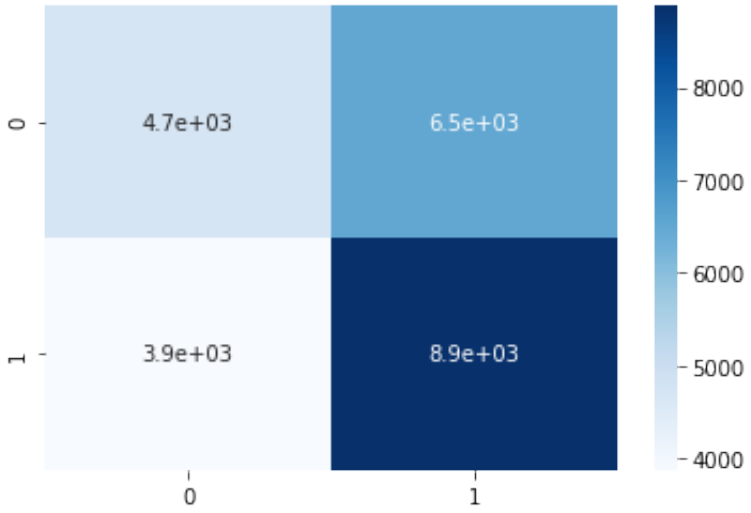
```
# Import time
from time import time
```

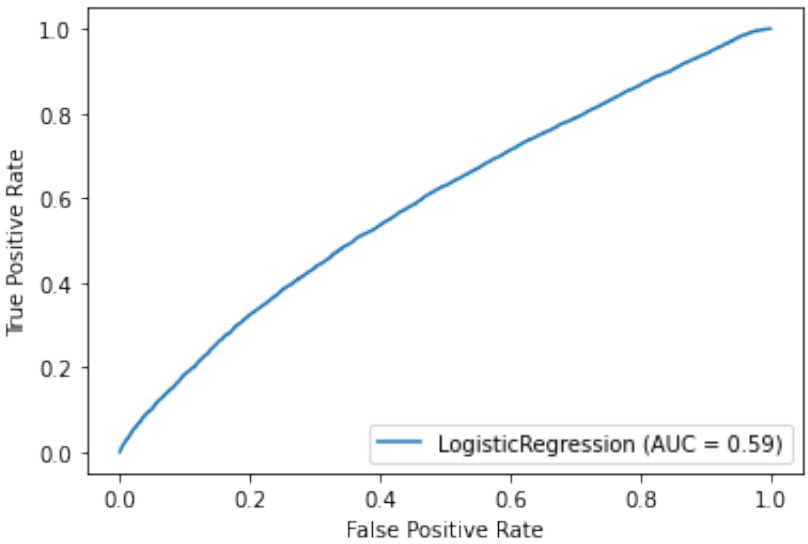
```
# Import the classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier

# Import the metrics
from sklearn.metrics import precision_recall_curve, auc, roc_curve, plot_r
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score, recall_score, accuracy_score

# Function to evaluate the model
def evaluate(clf, X, y):

  # Confusion matrix
  # Get Predictions
  y_pred = clf.predict(X)
  confMat = confusion_matrix(y, y_pred)

  # Calculate Values
  pr = precision_score(y, y_pred)
  rc = recall_score(y, y_pred)
  acc = accuracy_score(y, y_pred)

  # Print them
  print("Confusion Matrix:\n", confMat)
  print("Precision: ", pr)
  print("Recall: ", rc)
  print("Accuracy: ", acc)

  # Plot the matrix
  sn.heatmap(confMat, cmap="Blues", annot=True)
  plt.show()

  print("-------------------------------------------------------------")

  # ROC - AUC
  # Get the probabilities
  y_predict_proba = clf.predict_proba(X)[:, 1]

  # Calculate the values
  precision, recall, thresholds = precision_recall_curve(y, y_predict_proba
  fpr, tpr, thresholds = roc_curve(y, y_predict_proba)

  # Plot the curve
  plot_roc_curve(clf, X, y)
  plt.show()

  # Print values of area under the ROC and PR curves
  print('ROC AUC', auc(fpr, tpr), '\n', 'PR AUC', auc(recall, precision))
```

Logistic Regression

In [24]:
```
t1 = time()
# Create model
clf = LogisticRegression()
# Train it
clf.fit(x_train, y_train)
# Evaluate it
evaluate(clf, x_test, y_test)
print("Execution took ", time()-t1, "s")
```

```
Confusion Matrix:
 [[4721 6525]
 [3875 8879]]
Precision:  0.5764087250064919
Recall:  0.6961737494119492
Accuracy:  0.5666666666666667
```





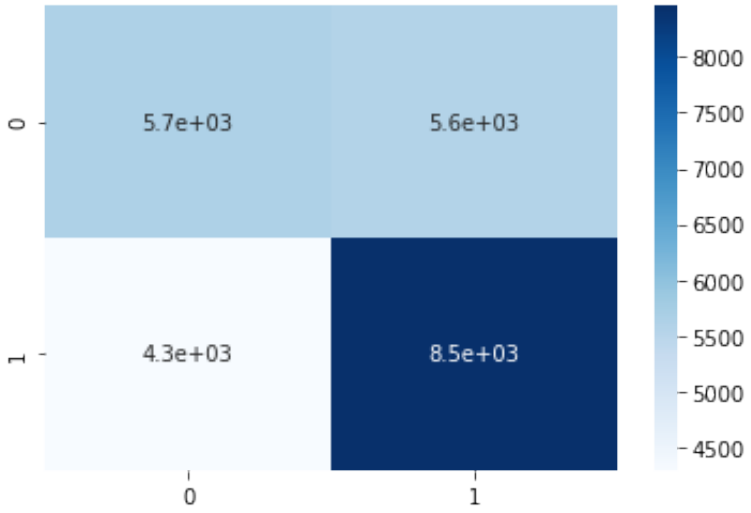

```
ROC AUC 0.594064766142976
 PR AUC 0.6202330766696245
Execution took  1.2451484203338623 s
```

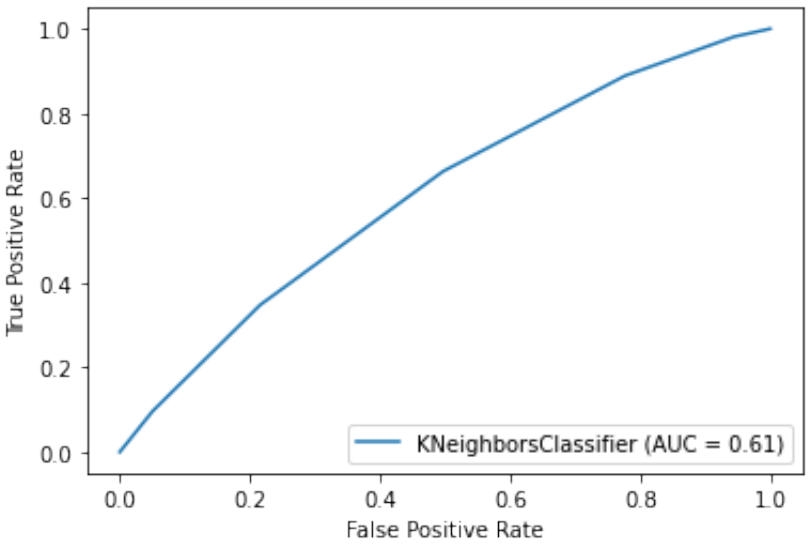
From this, we can see that logistic regression performs *slightly* better than random.

K-Nearest Neighbours

In [26]:
```
t1 = time()
# Create model
clf = KNeighborsClassifier(5)
# Train it
clf.fit(x_train, y_train)
# Evaluate it
evaluate(clf, x_test, y_test)
print("Execution took ", time()-t1, "s")
```

```
Confusion Matrix:
 [[5656 5590]
 [4296 8458]]
Precision:  0.6020785876993167
Recall:  0.6631644974125764
Accuracy:  0.5880833333333333
```






```
ROC AUC 0.6101635119385643
 PR AUC 0.6379985511502878
Execution took  774.771154165268 s
```

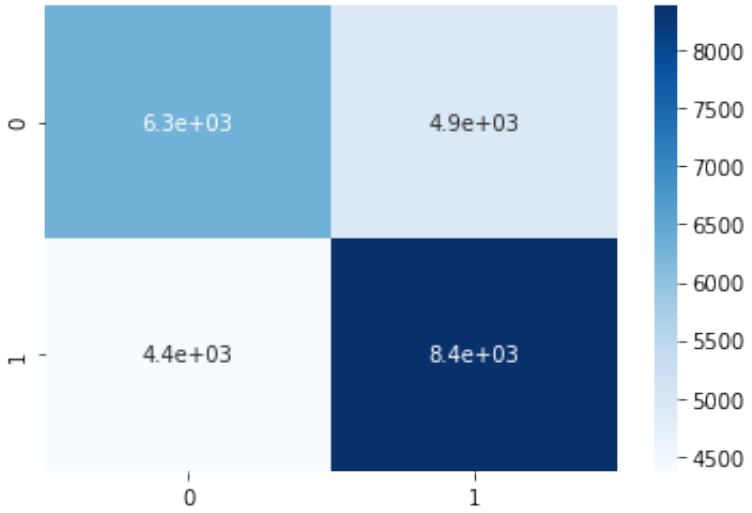
From this, we can see that the KNN classification method too, does not fare much better than random guessing.

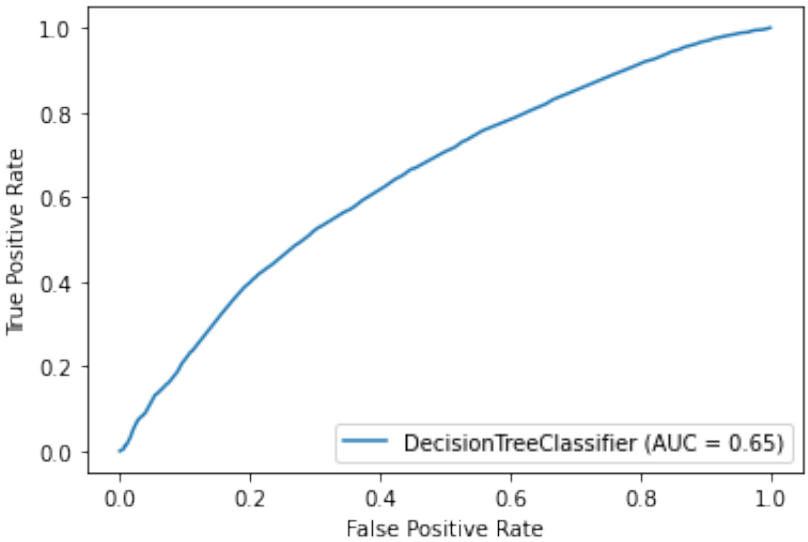
Decision Tree

In [25]:

```
t1 = time()
# Create model
clf = DecisionTreeClassifier(max_depth=10)
# Train it
clf.fit(x_train, y_train)
# Evaluate it
evaluate(clf, x_test, y_test)
print("Execution took ", time()-t1, "s")
```

```
Confusion Matrix:
 [[6299 4947]
 [4372 8382]]
Precision:  0.6288543776727437
Recall:  0.6572055825623334
Accuracy:  0.6117083333333333
```





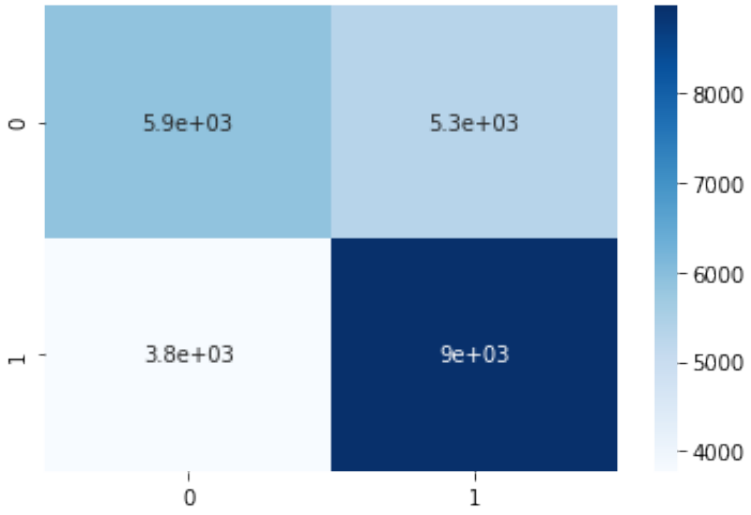

```
ROC AUC 0.6494379957750419
 PR AUC 0.6533941050833969
Execution took  2.678687572479248 s
```

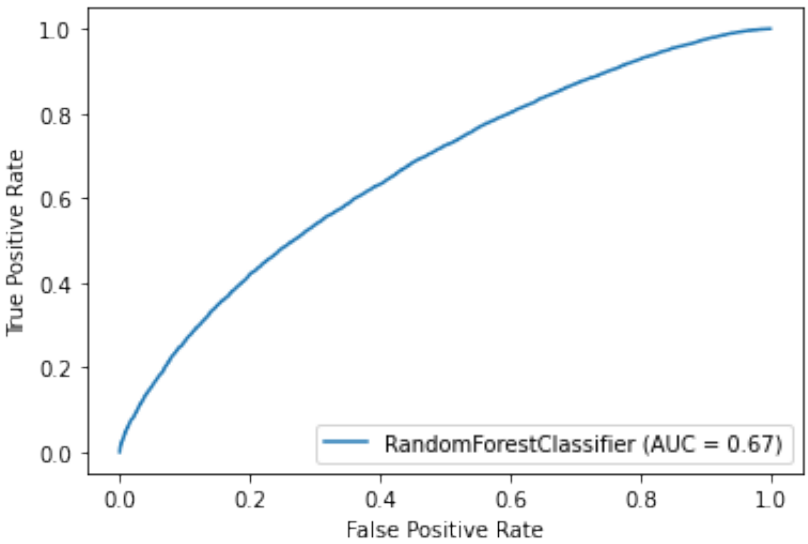
From this, we can see that a Decision Tree (with a depth of 10) also only performs slightly better than random guessing, but does much better than the Logistic Regression and KNNs.

Random Forest

In [27]:
```
t1 = time()
# Create model
clf = RandomForestClassifier(max_depth=15, n_estimators=15, max_features='
# Train it
clf.fit(x_train, y_train)
# Evaluate it
evaluate(clf, x_test, y_test)
print("Execution took ", time()-t1, "s")
```

```
Confusion Matrix:
 [[5909 5337]
 [3775 8979]]
Precision:  0.627200335289187
Recall:  0.7040144268464795
Accuracy:  0.6203333333333333
```





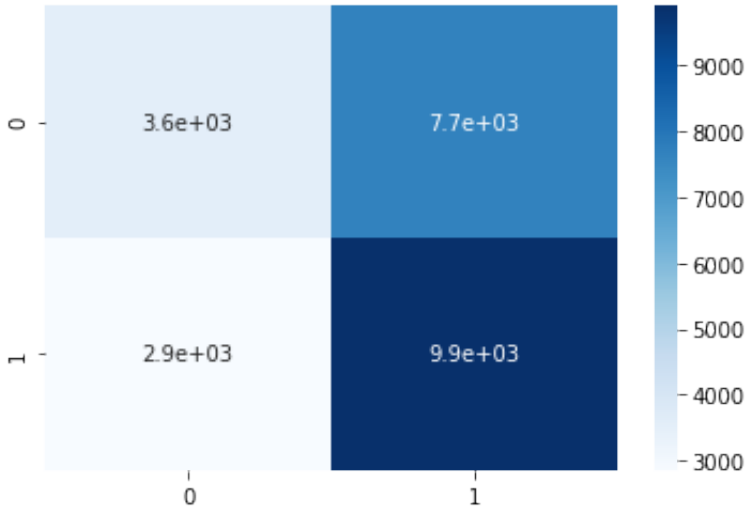

```
ROC AUC 0.6683998681907245
 PR AUC 0.6887949197787688
Execution took  6.956610441207886 s
```

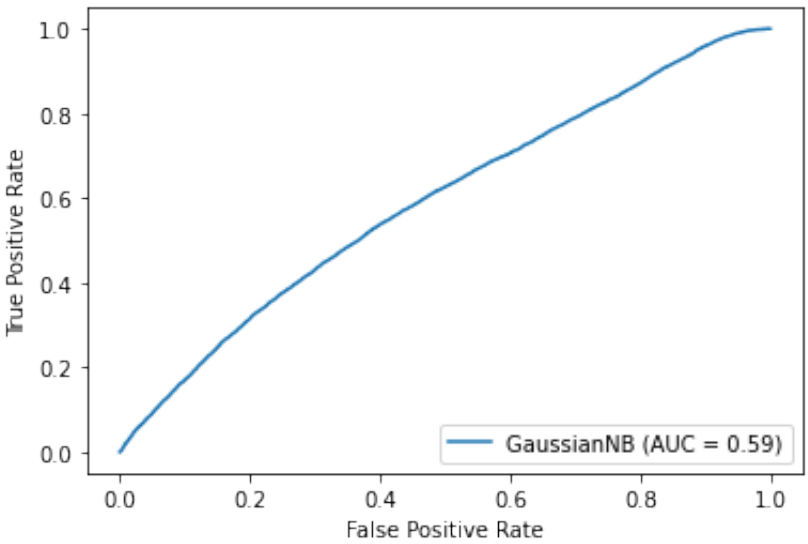
From this, we can see that the Random Forest has performed the best so far, but it too isn't a great classifier, with an AUC of just 0.67.

Naive Bayes Classifier

In [28]:
```
t1 = time()
# Create model
clf = GaussianNB()
# Train it
clf.fit(x_train, y_train)
# Evaluate it
evaluate(clf, x_test, y_test)
print("Execution took ", time()-t1, "s")
```

```
Confusion Matrix:
 [[3561 7685]
 [2854 9900]]
Precision:  0.5629798123400626
Recall:  0.776227066018504
Accuracy:  0.560875
```






```
ROC AUC 0.5926038246944444
 PR AUC 0.610965947063866
Execution took  0.5450377464294434 s
```

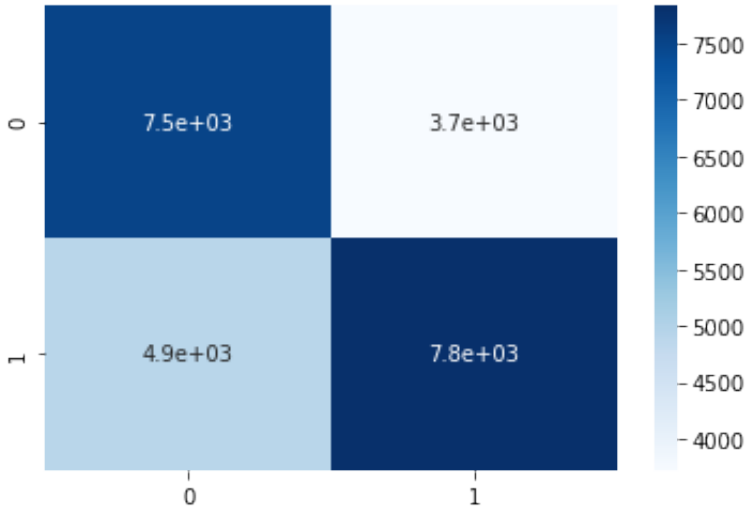
This classifier also performs poorly with the task at hand.

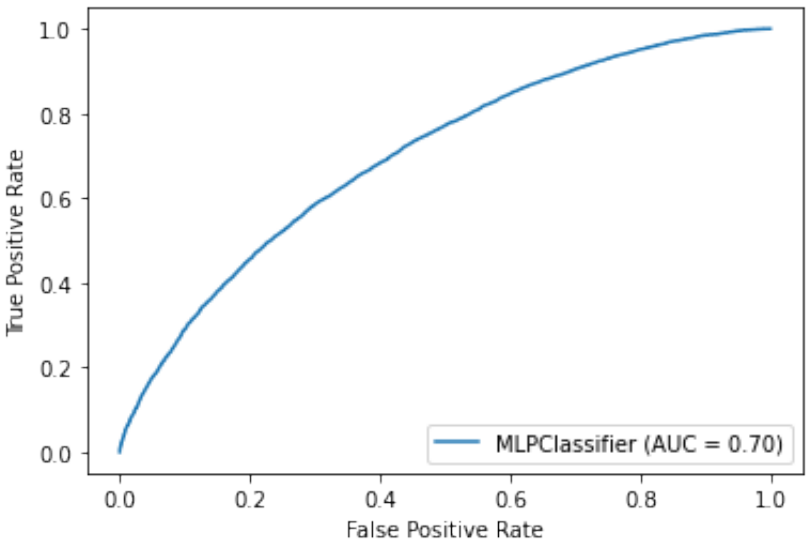
Multi Layer Perceptron

In [29]:

```
t1 = time()
# Create model
clf = MLPClassifier(max_iter=1000)
# Train it
clf.fit(x_train, y_train)
# Evaluate it
evaluate(clf, x_test, y_test)
print("Execution took ", time()-t1, "s")
```

```
Confusion Matrix:
 [[7525 3721]
 [4919 7835]]
Precision:  0.6780027691242645
Recall:  0.6143170769954525
Accuracy:  0.64
```





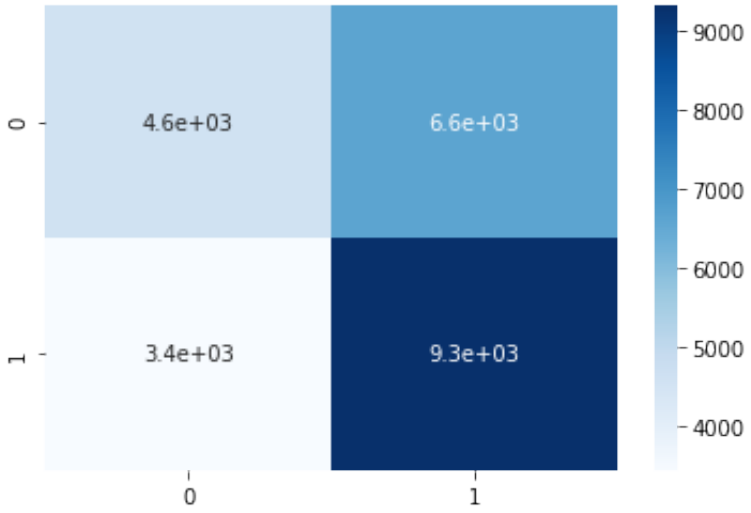

```
ROC AUC 0.7006138903227133
 PR AUC 0.7125871079815211
Execution took  85.41898965835571 s
```

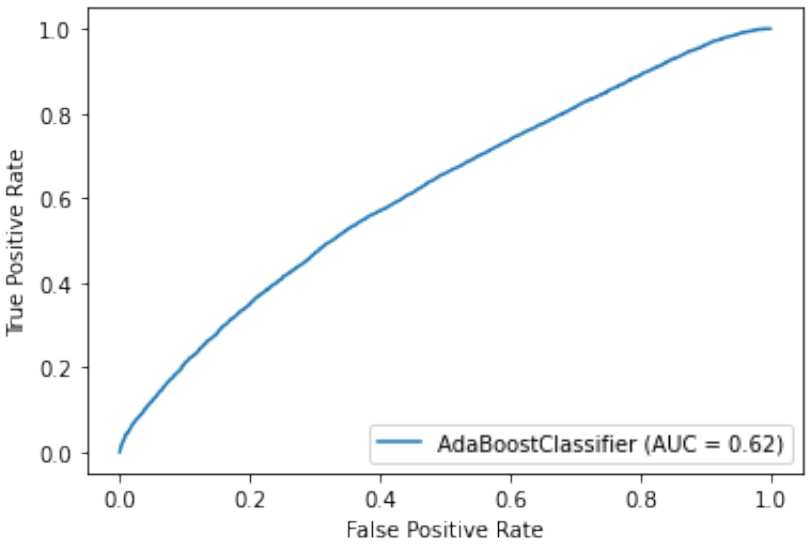
So far, this classifier has performed the best.

AdaBoost

In [30]:
```
t1 = time()
# Create model
clf = AdaBoostClassifier()
# Train it
clf.fit(x_train, y_train)
# Evaluate it
evaluate(clf, x_test, y_test)
print("Execution took ", time()-t1, "s")
```

```
Confusion Matrix:
 [[4603 6643]
 [3436 9318]]
Precision:  0.5837980076436313
Recall:  0.7305943233495374
Accuracy:  0.5800416666666667
```






```
ROC AUC 0.6188446185218303
 PR AUC 0.6453742539270508
Execution took  13.581999063491821 s
```

AdaBoost too has not performed well on the data.

Conclusion

Summary of all the model's performances:

1. Logistic Regression
   ```
   Precision:  0.5764087250064919
   Recall:  0.6961737494119492
   Accuracy:  0.5666666666666667
   ROC AUC 0.594064766142976
   PR AUC 0.6202330766696245
   ```
2. K-Nearest Neighbours
   ```
   Precision:  0.6020785876993167
   Recall:  0.6631644974125764
   Accuracy:  0.5880833333333333
   ROC AUC 0.6101635119385643
   PR AUC 0.6379985511502878
   ```
3. Decision Tree
   ```
   Precision:  0.6288543776727437
   Recall:  0.6572055825623334
   Accuracy:  0.6117083333333333
   ROC AUC 0.6494379957750419
   PR AUC 0.6533941050833969
   ```
4. Random Forest
   ```
   Precision:  0.627200335289187
   Recall:  0.7040144268464795
   Accuracy:  0.6203333333333333
   ROC AUC 0.6683998681907245
   PR AUC 0.6887949197787688
   ```
5. Naive Bayes Classifier
   ```
   Precision:  0.5629798123400626
   Recall:  0.776227066018504
   Accuracy:  0.560875
   ROC AUC 0.5926038246944444
   PR AUC 0.610965947063866
   ```
6. Multi Layer Perceptron
   ```
   Precision:  0.6780027691242645
   Recall:  0.6143170769954525
   Accuracy:  0.64
   ROC AUC 0.7006138903227133
   PR AUC 0.7125871079815211
   ```
7. AdaBoost
   ```
   Precision:  0.5837980076436313
   Recall:  0.7305943233495374
   Accuracy:  0.5800416666666667
   ROC AUC 0.6188446185218303
   PR AUC 0.6453742539270508
   ```

From this, we can conclude that the Multi Layer Perceptron performed the best of the 7

classifiers tested.

Future Work

So far, we've only used scikit learn's inbuilt classifiers for the task, on a very limited portion of the dataset. From this, we could clearly see that it was the multi-layer perceptron that performed the best on this limited dataset. So, the next natural steps are:

- Build a feed-forward network in a deep-learning framework (TensorFlow / Keras).
- Use larger versions of the dataset.
- From the plot of all columns, it seems like we could replace a few columns (9, 13, 17, 21) with one-hot encodings which map their values to a simpler input.
- We can try training the models on scaled data which lies within the [0, 1] range.

Dimensionality Reduction

Next, we look at dimensionality reduction techniques which we can apply on the dataset.

To see if this works properly, we'll be testing the reduced dataset with the Multi Layer Perceptron from above.

From what we had seen in the "Understanding the Dataset" phase, it is evident that we cannot apply the following techniques:

1. Low Variance Filter
2. High Correlation Filter

Also, given that the Random Forest model demonstrated a mediocre performance, it doesn't make much sense to use it for feature selection and subsequent reduction.

Hence, we go for the following technique instead

- Principal Component Analysis
- Neural Auto Encoder

PCA

First, we bring all the features onto the same scale.

In [54]:

```
import numpy as np
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
x_std =  sc.fit_transform(x) # x is the full shortData's indepent features

# Print the covariance matrix once, before we proceed
cov_matrix = np.cov(x_std.T)
print("cov_matrix shape:",cov_matrix.shape)
print("Covariance_matrix",cov_matrix)
```

```
cov_matrix shape: (21, 21)
Covariance_matrix [[ 1.00000833e+00  5.72030784e-04  7.63847401e-04 -1.4219
7220e-01
   3.15669967e-03  1.63546774e-01 -4.01262825e-03 -9.72087201e-04
  -8.27160766e-03  2.87064714e-03 -3.86642628e-04  8.84698954e-04
  -4.68132504e-03 -9.94146128e-03  2.74567398e-03 -5.72972061e-03
   6.33101142e-03 -1.45760538e-02  1.91428549e-03 -4.42067671e-03
   6.65140244e-03]
 [ 5.72030784e-04  1.00000833e+00 -1.85103975e-03  1.76482840e-04
  -2.96255955e-03 -1.02624404e-03  2.65643483e-01  2.13501578e-04
   1.49889706e-03 -2.76653743e-03  2.63176356e-01  1.72879252e-03
   1.63938412e-03 -1.02865621e-03  2.26266395e-01  1.70148417e-03
   3.15235091e-03 -3.02524902e-03  1.78611323e-01 -8.99279434e-04
  -5.75251370e-03]
 [ 7.63847401e-04 -1.85103975e-03  1.00000833e+00  1.64696313e-03
  -4.74863000e-02  4.20335977e-03 -2.93895937e-03 -1.68808115e-01
  -7.83559471e-04  2.96745028e-03 -3.07945052e-03 -9.01104199e-02
  -1.16239825e-03  4.05536825e-03 -3.30455942e-03 -8.46132119e-02
  -2.58132216e-03  5.98850900e-04  2.82989247e-04 -6.40925213e-02
   1.47505924e-03]
 [-1.42197220e-01  1.76482840e-04  1.64696313e-03  1.00000833e+00
   2.13520419e-03  1.95842716e-01 -7.65046551e-04 -3.23354246e-03
  -2.99861694e-02  3.56153824e-02 -1.42770739e-03  1.64862054e-03
  -3.67909458e-03  1.91401919e-02 -1.63596815e-03  2.24449692e-03
   1.54537879e-02  1.74982225e-03 -4.38697252e-03 -3.85402715e-03
   8.00075423e-03]
 [ 3.15669967e-03 -2.96255955e-03 -4.74863000e-02  2.13520419e-03
   1.00000833e+00  5.19784282e-03  1.34980630e-03 -1.56359826e-01
  -1.15369501e-03  2.36810163e-03  1.16536871e-03 -6.18757877e-02
  -4.87659317e-04  3.72229755e-03 -1.70689537e-03 -5.38205359e-02
   2.86813168e-03  1.61545994e-03 -2.11462471e-03 -3.55912299e-02
  -7.71372947e-04]
 [ 1.63546774e-01 -1.02624404e-03  4.20335977e-03  1.95842716e-01
   5.19784282e-03  1.00000833e+00 -2.52372918e-03 -2.50121269e-03
  -1.40310952e-02  4.84302756e-01 -4.13416227e-04  3.25527323e-03
  -2.89558612e-02  2.66127300e-01  3.24586125e-03 -2.60233192e-03
  -2.74477269e-02  1.65497903e-01  2.63277697e-03 -1.70995162e-03
  -5.82655827e-03]
 [-4.01262825e-03  2.65643483e-01 -2.93895937e-03 -7.65046551e-04
   1.34980630e-03 -2.52372918e-03  1.00000833e+00  2.47930681e-03
  -4.80495725e-03  1.02400748e-03  2.43966251e-01 -2.11164793e-03
   3.09514775e-03  2.18251890e-03  2.27206085e-01  1.37356375e-03
  -1.85945731e-03  3.64691541e-03  1.90800725e-01 -1.16274695e-03
   6.97822536e-05]
 [-9.72087201e-04  2.13501578e-04 -1.68808115e-01 -3.23354246e-03
  -1.56359826e-01 -2.50121269e-03  2.47930681e-03  1.00000833e+00
   2.26892057e-03 -5.40455960e-03  1.22284682e-03 -1.99728590e-01
```

```
 -1.60562728e-03 -2.68456137e-03 -7.82234417e-04 -1.35220491e-01
 -2.22143601e-03  8.64202496e-04  2.35824835e-03 -1.02632821e-01
  4.01456643e-03]
[-8.27160766e-03  1.49889706e-03 -7.83559471e-04 -2.99861694e-02
 -1.15369501e-03 -1.40310952e-02 -4.80495725e-03  2.26892057e-03
  1.00000833e+00 -1.38258971e-01 -7.71796241e-04  3.19665551e-03
 -2.61806287e-01 -1.45489835e-01  1.91314888e-04  2.45407743e-03
 -2.51689119e-01 -1.46851692e-01  1.14298019e-03 -1.82892988e-03
 -2.34283142e-01]
[ 2.87064714e-03 -2.76653743e-03  2.96745028e-03  3.56153824e-02
  2.36810163e-03  4.84302756e-01  1.02400748e-03 -5.40455960e-03
 -1.38258971e-01  1.00000833e+00 -4.07044112e-03  2.54659380e-04
  1.43626020e-01  2.93775076e-01 -5.08007808e-03 -7.61701801e-03
 -4.57361353e-02  1.86027381e-01  2.80300594e-04  6.20113658e-03
 -3.04116675e-02]
[-3.86642628e-04  2.63176356e-01 -3.07945052e-03 -1.42770739e-03
  1.16536871e-03 -4.13416227e-04  2.43966251e-01  1.22284682e-03
 -7.71796241e-04 -4.07044112e-03  1.00000833e+00  4.36786655e-03
  5.91669108e-03 -1.58582613e-04  2.10985076e-01  1.19885734e-03
  1.49880951e-04 -2.42362691e-03  1.73626381e-01  2.26754323e-03
 -1.73130824e-03]
[ 8.84698954e-04  1.72879252e-03 -9.01104199e-02  1.64862054e-03
 -6.18757877e-02  3.25527323e-03 -2.11164793e-03 -1.99728590e-01
  3.19665551e-03  2.54659380e-04  4.36786655e-03  1.00000833e+00
  2.00736717e-03 -3.22413726e-03 -2.09929717e-04 -9.03055420e-02
 -2.81196718e-03 -2.04195442e-03 -1.92490022e-03 -7.03899719e-02
 -3.18152403e-03]
[-4.68132504e-03  1.63938412e-03 -1.16239825e-03 -3.67909458e-03
 -4.87659317e-04 -2.89558612e-02  3.09514775e-03 -1.60562728e-03
 -2.61806287e-01  1.43626020e-01  5.91669108e-03  2.00736717e-03
  1.00000833e+00 -8.41953306e-02  4.97597268e-03 -1.97448743e-03
 -2.47555362e-01 -9.92822040e-02  4.45997696e-03 -1.07103365e-03
 -2.45140596e-01]
[-9.94146128e-03 -1.02865621e-03  4.05536825e-03  1.91401919e-02
  3.72229755e-03  2.66127300e-01  2.18251890e-03 -2.68456137e-03
 -1.45489835e-01  2.93775076e-01 -1.58582613e-04 -3.22413726e-03
 -8.41953306e-02  1.00000833e+00 -5.17662736e-04 -8.57904004e-04
  1.86973082e-01  2.94855819e-01  3.57175694e-03 -2.76257434e-04
 -3.59374725e-02]
[ 2.74567398e-03  2.26266395e-01 -3.30455942e-03 -1.63596815e-03
 -1.70689537e-03  3.24586125e-03  2.27206085e-01 -7.82234417e-04
  1.91314888e-04 -5.08007808e-03  2.10985076e-01 -2.09929717e-04
  4.97597268e-03 -5.17662736e-04  1.00000833e+00  4.18381783e-03
  2.17222804e-03  9.56042767e-04  1.42865000e-01  8.49133409e-04
  1.18722378e-04]
[-5.72972061e-03  1.70148417e-03 -8.46132119e-02  2.24449692e-03
 -5.38205359e-02 -2.60233192e-03  1.37356375e-03 -1.35220491e-01
  2.45407743e-03 -7.61701801e-03  1.19885734e-03 -9.03055420e-02
 -1.97448743e-03 -8.57904004e-04  4.18381783e-03  1.00000833e+00
  9.26959437e-04 -3.52084541e-03  1.82113109e-03 -5.95291252e-02
  1.44054252e-03]
[ 6.33101142e-03  3.15235091e-03 -2.58132216e-03  1.54537879e-02
  2.86813168e-03 -2.74477269e-02 -1.85945731e-03 -2.22143601e-03
 -2.51689119e-01 -4.57361353e-02  1.49880951e-04 -2.81196718e-03
 -2.47555362e-01  1.86973082e-01  2.17222804e-03  9.26959437e-04
  1.00000833e+00 -5.85167355e-02 -1.04200450e-03  6.64900549e-03
 -2.56136456e-01]
[-1.45760538e-02 -3.02524902e-03  5.98850900e-04  1.74982225e-03
  1.61545994e-03  1.65497903e-01  3.64691541e-03  8.64202496e-04
 -1.46851692e-01  1.86027381e-01 -2.42362691e-03 -2.04195442e-03
 -9.92822040e-02  2.94855819e-01  9.56042767e-04 -3.52084541e-03
```

```
 -5.85167355e-02  1.00000833e+00  1.03151875e-03 -2.63912721e-03
  1.93356887e-01]
[ 1.91428549e-03  1.78611323e-01  2.82989247e-04 -4.38697252e-03
 -2.11462471e-03  2.63277697e-03  1.90800725e-01  2.35824835e-03
  1.14298019e-03  2.80300594e-04  1.73626381e-01 -1.92490022e-03
  4.45997696e-03  3.57175694e-03  1.42865000e-01  1.82113109e-03
 -1.04200450e-03  1.03151875e-03  1.00000833e+00  1.62349407e-03
 -6.73066697e-03]
[-4.42067671e-03 -8.99279434e-04 -6.40925213e-02 -3.85402715e-03
 -3.55912299e-02 -1.70995162e-03 -1.16274695e-03 -1.02632821e-01
 -1.82892988e-03  6.20113658e-03  2.26754323e-03 -7.03899719e-02
 -1.07103365e-03 -2.76257434e-04  8.49133409e-04 -5.95291252e-02
  6.64900549e-03 -2.63912721e-03  1.62349407e-03  1.00000833e+00
 -5.89103215e-03]
[ 6.65140244e-03 -5.75251370e-03  1.47505924e-03  8.00075423e-03
 -7.71372947e-04 -5.82655827e-03  6.97822536e-05  4.01456643e-03
 -2.34283142e-01 -3.04116675e-02 -1.73130824e-03 -3.18152403e-03
 -2.45140596e-01 -3.59374725e-02  1.18722378e-04  1.44054252e-03
 -2.56136456e-01  1.93356887e-01 -6.73066697e-03 -5.89103215e-03
  1.00000833e+00]]
```

Next, we calculate the eigenvectors and eigenvalues, and sort them in descending order.

In [55]:

```
# Calculate
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
print('Eigen Vectors:\n', eigenvectors)
print('Eigen Values:\n', eigenvalues)

# Make a set of (eigenvalue, eigenvector) pairs
eig_pairs = [(eigenvalues[index], eigenvectors[:,index]) for index in range
# Sort them
eig_pairs.sort()
eig_pairs.reverse()
print(eig_pairs)

# Extract the descending ordered eigenvalues and eigenvectors
eigvalues_sorted = [eig_pairs[index][0] for index in range(len(eigenvalues
eigvectors_sorted = [eig_pairs[index][1] for index in range(len(eigenvalues

# Print it out to see if it worked properly
print('Eigenvalues in descending order:\n', eigvalues_sorted)
```

```
Eigen Vectors:
 [[ 6.04309744e-02  2.03796851e-03  4.72527718e-03 -2.56018776e-01
   6.11150229e-03  8.70644837e-02 -4.12893117e-02 -3.62921846e-02
   1.46877603e-01  6.05441869e-02  3.48462460e-01  5.51107158e-02
   1.03620874e-03 -1.09523728e-03  1.57785117e-02 -5.04787226e-01
  -7.14191488e-01  2.36965963e-02 -4.00857078e-02 -5.58106993e-02
  -4.52823070e-03]
 [-1.31487458e-02  4.81284651e-01 -6.89608343e-04  3.01891157e-03
  -4.33148092e-03  5.45749064e-03  5.30895698e-03  4.99051061e-03
   6.16583094e-03  7.31684733e-03  1.20284157e-01 -7.83293607e-01
  -1.60505762e-01 -3.00665445e-01 -1.53394338e-01 -1.26079220e-02
   4.83429831e-03 -9.12901893e-04 -6.04170138e-03  7.50511720e-03
   2.15369205e-03]
 [ 8.31138770e-03 -6.93260905e-03 -1.39617902e-03 -1.40122801e-02
  -3.97028937e-01 -3.43489688e-03 -8.49782914e-03 -3.36263020e-03
  -1.18112806e-01  2.55817482e-01 -3.20530652e-03  6.97733351e-03
   6.03461890e-03 -3.88262782e-03 -1.02202515e-02 -1.38281747e-03
```

```
 -2.40045820e-02 -2.12444298e-01 -3.48938035e-01  5.10418632e-01
 -5.78233090e-01]
[ 1.34443275e-01 -6.08857475e-04  1.76246229e-03 -2.61866655e-01
  9.78495526e-03  8.67983731e-02 -6.40765293e-02 -6.89727428e-03
  9.50473931e-02  5.66509908e-02  3.69885641e-01  5.36764047e-02
  1.35333991e-02  1.29880008e-02  3.10657844e-02 -5.14994134e-01
  6.90829306e-01  5.25007609e-02  3.35057976e-03  5.51649684e-02
  5.71655582e-03]
[ 8.94052766e-03 -2.48740489e-03  9.39495186e-04 -1.23084422e-02
 -3.26183002e-01  9.28711917e-04 -1.55777280e-02  1.43924764e-02
 -1.35200104e-01  2.84108136e-01 -7.76749664e-03 -9.55118369e-03
  6.90009750e-03  2.90953440e-03  6.65594545e-03  4.29424950e-02
 -2.19008964e-02  7.70096796e-01  3.73887635e-01 -1.01445536e-01
 -2.19570254e-01]
[ 5.02564999e-01  1.12895344e-02 -2.88029131e-02  6.81869247e-01
 -2.46970578e-02 -1.28634266e-01 -1.51225332e-01 -9.84719270e-02
  3.34321278e-01  1.46903527e-01 -1.59050547e-01 -1.65697735e-02
 -5.14611490e-03 -2.50186345e-03  5.55311849e-03 -2.69032046e-01
 -9.88024540e-03  1.19633714e-02 -6.42858197e-03 -1.23457916e-02
 -1.96083015e-03]
[-7.50069342e-03  4.77830222e-01 -4.07362480e-03  6.60934162e-03
 -2.21827562e-03  1.70641326e-03  1.36750095e-02 -5.26530836e-03
 -4.46833579e-03 -2.31425648e-03 -2.90496936e-02  4.81097217e-01
 -7.26979730e-01 -8.29092350e-02 -5.67483310e-02  1.10508002e-02
  9.83051416e-03  1.15472542e-02  1.92375327e-03  4.22672277e-03
 -6.92338109e-03]
[-8.07367611e-03  3.25161078e-03  1.33602604e-03 -2.22177196e-02
 -5.65653705e-01  2.70760783e-03  4.66980913e-02 -2.08233610e-02
  3.46853287e-01 -7.20261259e-01 -7.21049531e-03  9.76671702e-04
  8.42513143e-03 -1.78334828e-03 -2.16602224e-03  2.86319082e-03
  1.96936239e-03  1.34662725e-01 -2.36312288e-02  1.19149604e-01
  6.87501866e-02]
[-2.63775849e-01 -9.73970142e-03 -5.08803535e-01 -1.13286414e-01
  7.03547646e-03  3.74178372e-02 -3.09470270e-02 -5.65419821e-02
  6.83496014e-01  3.22815863e-01  7.90810846e-02  1.52784674e-02
  5.13249305e-03  3.75335187e-03 -1.74389970e-02  2.68704697e-01
  4.03383576e-02 -2.03016013e-02  1.93513333e-02 -5.72963884e-03
 -1.22779628e-02]
[ 5.17216064e-01  1.03220898e-02 -5.45688749e-03 -6.10478321e-01
  2.42311972e-02 -2.09779545e-01 -2.56104811e-01 -1.55416629e-01
  8.42795720e-02  2.01023039e-02 -4.47993444e-01 -6.33806235e-02
 -2.88448096e-02 -1.93345885e-02 -1.12646461e-02  1.03876739e-01
 -1.86171408e-02 -1.93957039e-02  1.15284160e-02 -1.06174888e-02
 -3.86065766e-03]
[-1.22316925e-02  4.64572462e-01  2.34791360e-03 -5.00576551e-03
  9.61223816e-03 -9.69127305e-03  5.61873004e-03 -2.63487738e-03
 -2.25010122e-03  6.50606836e-03 -6.33093039e-02  3.66137472e-01
  6.17228787e-01 -4.97118563e-01 -1.31486825e-01 -9.42769257e-03
  2.87178418e-03  4.50586457e-03  7.21671141e-03  6.45201292e-03
  7.55903049e-03]
[-7.24320720e-04  7.31134491e-04 -2.20677414e-04 -1.70073519e-02
 -4.40902136e-01  4.80466924e-03 -3.23035125e-02 -6.89717657e-03
 -1.59138037e-01  3.68927533e-01 -6.78783340e-03  5.11127658e-03
 -4.25824138e-03  5.26554753e-03  1.14064211e-02  9.73192316e-03
 -1.96558710e-02 -4.44339649e-02 -1.33881308e-01  1.90663087e-01
  7.65367959e-01]
[ 1.02221434e-02  1.39542936e-02 -5.06749693e-01  1.14058738e-01
 -7.82563144e-03  8.95178645e-02 -5.96845802e-01 -3.55991866e-01
 -3.72846456e-01 -2.08874589e-01  2.19843299e-01  2.50114415e-02
  1.55061396e-02  1.09516545e-02 -6.69075025e-03  7.89286296e-02
 -1.95223955e-02  4.12624244e-03 -9.70555765e-03 -9.82460514e-03
```

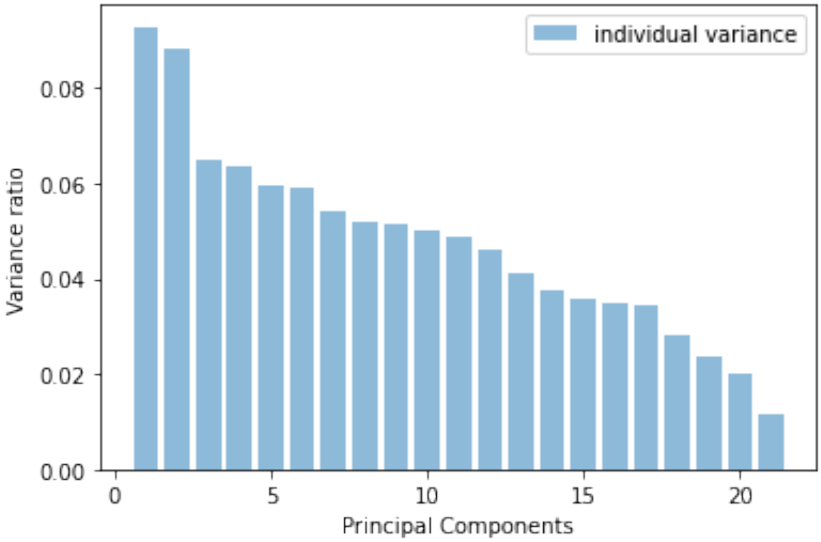
```
  -6.54700421e-03]
 [ 4.74090541e-01  1.25775751e-02 -1.36409572e-02 -4.93853918e-03
   2.18142193e-03  7.55753300e-01  5.33111420e-02  2.80053102e-01
   1.80920117e-02 -3.33210627e-03  1.46719417e-01  2.27476987e-02
   1.80795572e-02  1.91105044e-03 -1.29943262e-02  3.13603925e-01
  -2.82483600e-02 -1.73403762e-02 -9.12171988e-03 -3.81686236e-03
   2.04288791e-03]
 [-9.29993930e-03  4.29786397e-01  6.18426366e-03 -1.06884889e-02
  -2.41128645e-03  2.32152153e-03  9.53697555e-03 -1.96343289e-03
   1.25177402e-03  5.93454287e-03 -3.43903426e-02 -7.55622711e-03
   2.20464112e-01  7.79614485e-01 -3.95597765e-01 -2.62770559e-02
  -2.92953320e-03 -2.40076729e-03 -1.83255419e-03 -7.19330894e-03
  -1.17917434e-03]
 [-8.14694677e-03  5.17518929e-03  1.38627052e-03 -1.72012392e-02
  -3.66045755e-01 -3.77006985e-03  6.79868893e-03  1.13824435e-02
  -5.17691810e-02  9.65755028e-02 -3.48209382e-03  4.45371543e-03
   2.74862271e-03 -1.30425874e-02  3.58125495e-03 -4.09283866e-03
   7.98508694e-02 -1.65812925e-01 -3.66867611e-01 -8.16132734e-01
  -1.36560983e-01]
 [ 7.69899831e-02  4.43939172e-03 -5.01391694e-01 -1.51747186e-02
  -6.31447129e-04 -2.55031424e-01 -1.43343366e-02  7.80253712e-01
  -1.46197909e-01 -8.82358982e-02 -9.65313493e-02 -9.00412240e-03
  -5.86151504e-03  3.99030162e-04  5.18111537e-03 -1.66731776e-01
  -2.79192699e-02  1.61144270e-02 -2.38602895e-02  1.34770633e-02
   1.32834153e-02]
 [ 3.87178852e-01  6.29920526e-03 -6.94347795e-02 -5.61613669e-03
  -5.83144207e-03 -4.90547824e-01  3.62168842e-01 -1.04142899e-01
  -6.70381703e-02 -1.53412372e-02  5.72470278e-01  6.53816533e-02
   3.10407496e-02  2.08125465e-02 -1.29611226e-02  3.50825861e-01
  -2.12538706e-02 -2.08372890e-02  1.23685398e-02 -4.07794677e-03
   9.56532101e-03]
 [-5.36005821e-03  3.72258174e-01 -1.43633754e-03 -3.76257567e-03
   2.08680338e-03 -2.29537745e-03 -1.68874220e-03 -8.67118820e-04
   1.05139147e-02  7.66002679e-04 -1.03095683e-02 -5.43675282e-02
   1.15166452e-01  2.15237262e-01  8.92665489e-01  3.52967056e-02
  -1.12541492e-02 -1.20145987e-02 -6.73777689e-04  5.42881516e-03
  -1.42679398e-02]
 [ 1.45727451e-03  1.14015248e-03 -1.30781693e-03 -5.41618858e-03
  -2.93277755e-01  9.72314749e-03 -1.88302392e-02  2.67224695e-02
  -4.33606103e-02  5.77078067e-02  1.75209942e-02  2.98270151e-03
  -4.32279147e-03 -3.95026857e-03 -3.01413305e-03 -5.91814749e-02
  -7.88032421e-03 -5.55515166e-01  7.62939299e-01 -7.48167780e-02
  -8.59378823e-02]
 [ 8.44565804e-02 -9.17423311e-03 -4.76410336e-01 -2.14286896e-02
   2.48776961e-03  1.94390569e-01  6.40354348e-01 -3.67665950e-01
  -1.88535105e-01 -3.78383896e-02 -2.85507135e-01 -4.22965249e-02
  -8.09967100e-03 -1.06077924e-02  1.48977340e-02 -2.47730230e-01
   7.47160589e-03  7.30696079e-03  1.16977748e-02 -1.36540952e-04
   5.84263405e-03]]
Eigen Values:
 [1.95084819 1.85805902 0.24578343 0.42459227 0.49507822 0.59326259
 1.36482363 1.33936985 1.24274343 1.24975633 0.73059873 0.72800433
 0.75213919 0.79360774 0.86790455 0.96756747 1.14041728 1.02688461
 1.05601115 1.08088484 1.09183813]
[(1.9508481917521363, array([ 0.06043097, -0.01314875,  0.00831139,  0.1344
4327,  0.00894053,
        0.502565  , -0.00750069, -0.00807368, -0.26377585,  0.51721606,
       -0.01223169, -0.00072432,  0.01022214,  0.47409054, -0.00929994,
       -0.00814695,  0.07698998,  0.38717885, -0.00536006,  0.00145727,
        0.08445658])), (1.858059018594852, array([ 0.00203797,  0.48128465,
-0.00693261, -0.00060886, -0.0024874 ,
```

```
        0.01128953,  0.47783022,  0.00325161, -0.0097397 ,  0.01032209,
        0.46457246,  0.00073113,  0.01395429,  0.01257758,  0.4297864 ,
        0.00517519,  0.00443939,  0.00629921,  0.37225817,  0.00114015,
       -0.00917423])), (1.3648236325070462, array([-0.04128931,  0.00530896
, -0.00849783, -0.06407653, -0.01557773,
       -0.15122533,  0.01367501,  0.04669809, -0.03094703, -0.25610481,
        0.00561873, -0.03230351, -0.5968458 ,  0.05331114,  0.00953698,
        0.00679869, -0.01433434,  0.36216884, -0.00168874, -0.01883024,
        0.64035435])), (1.339369849678526, array([-0.03629218,  0.00499051,
-0.00336263, -0.00689727,  0.01439248,
       -0.09847193, -0.00526531, -0.02082336, -0.05654198, -0.15541663,
       -0.00263488, -0.00689718, -0.35599187,  0.2800531 , -0.00196343,
        0.01138244,  0.78025371, -0.1041429 , -0.00086712,  0.02672247,
       -0.36766595])), (1.2497563300791545, array([ 0.06054419,  0.00731685
,  0.25581748,  0.05665099,  0.28410814,
        0.14690353, -0.00231426, -0.72026126,  0.32281586,  0.0201023 ,
        0.00650607,  0.36892753, -0.20887459, -0.00333211,  0.00593454,
        0.0965755 , -0.0882359 , -0.01534124,  0.000766  ,  0.05770781,
       -0.03783839])), (1.2427434342899308, array([ 0.1468776 ,  0.00616583
, -0.11811281,  0.09504739, -0.1352001 ,
        0.33432128, -0.00446834,  0.34685329,  0.68349601,  0.08427957,
       -0.0022501 , -0.15913804, -0.37284646,  0.01809201,  0.00125177,
       -0.05176918, -0.14619791, -0.06703817,  0.01051391, -0.04336061,
       -0.18853511])), (1.140417283992402, array([-0.71419149,  0.0048343 ,
-0.02400458,  0.69082931, -0.0219009 ,
       -0.00988025,  0.00983051,  0.00196936,  0.04033836, -0.01861714,
        0.00287178, -0.01965587, -0.0195224 , -0.02824836, -0.00292953,
        0.07985087, -0.02791927, -0.02125387, -0.01125415, -0.00788032,
        0.00747161])), (1.0918381293928954, array([-0.00452823,  0.00215369
, -0.57823309,  0.00571656, -0.21957025,
       -0.00196083, -0.00692338,  0.06875019, -0.01227796, -0.00386066,
        0.00755903,  0.76536796, -0.006547  ,  0.00204289, -0.00117917,
       -0.13656098,  0.01328342,  0.00956532, -0.01426794, -0.08593788,
        0.00584263])), (1.0808848400720503, array([-5.58106993e-02,  7.5051
1720e-03,  5.10418632e-01,  5.51649684e-02,
       -1.01445536e-01, -1.23457916e-02,  4.22672277e-03,  1.19149604e-01,
       -5.72963884e-03, -1.06174888e-02,  6.45201292e-03,  1.90663087e-01,
       -9.82460514e-03, -3.81686236e-03, -7.19330894e-03, -8.16132734e-01,
        1.34770633e-02, -4.07794677e-03,  5.42881516e-03, -7.48167780e-02,
       -1.36540952e-04])), (1.0560111548119322, array([-4.00857078e-02, -6.
04170138e-03, -3.48938035e-01,  3.35057976e-03,
        3.73887635e-01, -6.42858197e-03,  1.92375327e-03, -2.36312288e-02,
        1.93513333e-02,  1.15284160e-02,  7.21671141e-03, -1.33881308e-01,
       -9.70555765e-03, -9.12171988e-03, -1.83255419e-03, -3.66867611e-01,
       -2.38602895e-02,  1.23685398e-02, -6.73777689e-04,  7.62939299e-01,
        1.16977748e-02])), (1.0268846095348936, array([ 0.0236966 , -0.0009
129 , -0.2124443 ,  0.05250076,  0.7700968 ,
        0.01196337,  0.01154725,  0.13466272, -0.0203016 , -0.0193957 ,
        0.00450586, -0.04443396,  0.00412624, -0.01734038, -0.00240077,
       -0.16581292,  0.01611443, -0.02083729, -0.0120146 , -0.55551517,
        0.00730696])), (0.9675674729748748, array([-0.50478723, -0.01260792
, -0.00138282, -0.51499413,  0.0429425 ,
       -0.26903205,  0.0110508 ,  0.00286319,  0.2687047 ,  0.10387674,
       -0.00942769,  0.00973192,  0.07892863,  0.31360392, -0.02627706,
       -0.00409284, -0.16673178,  0.35082586,  0.03529671, -0.05918147,
       -0.24773023])), (0.8679045475101522, array([ 0.01577851, -0.15339434
, -0.01022025,  0.03106578,  0.00665595,
        0.00555312, -0.05674833, -0.00216602, -0.017439  , -0.01126465,
       -0.13148683,  0.01140642, -0.00669075, -0.01299433, -0.39559777,
        0.00358125,  0.00518112, -0.01296112,  0.89266549, -0.00301413,
        0.01489773])), (0.793607744075387, array([-1.09523728e-03, -3.00665
```

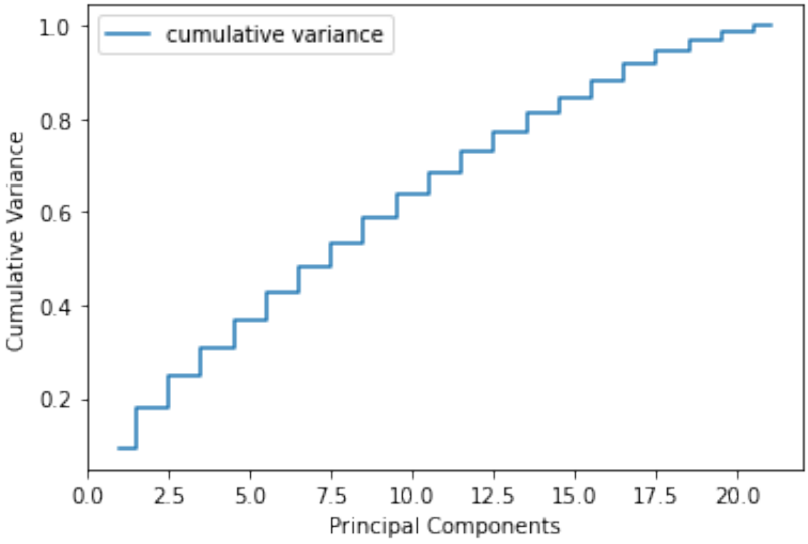
```
445e-01, -3.88262782e-03,  1.29880008e-02,
        2.90953440e-03, -2.50186345e-03, -8.29092350e-02, -1.78334828e-03,
        3.75335187e-03, -1.93345885e-02, -4.97118563e-01,  5.26554753e-03,
        1.09516545e-02,  1.91105044e-03,  7.79614485e-01, -1.30425874e-02,
        3.99030162e-04,  2.08125465e-02,  2.15237262e-01, -3.95026857e-03,
       -1.06077924e-02])), (0.7521391924764491, array([ 0.00103621, -0.1605
0576,  0.00603462,  0.0135334 ,  0.0069001 ,
       -0.00514611, -0.72697973,  0.00842513,  0.00513249, -0.02884481,
        0.61722879, -0.00425824,  0.01550614,  0.01807956,  0.22046411,
        0.00274862, -0.00586152,  0.03104075,  0.11516645, -0.00432279,
       -0.00809967])), (0.7305987250350349, array([ 0.34846246,  0.12028416
, -0.00320531,  0.36988564, -0.0077675 ,
       -0.15905055, -0.02904969, -0.0072105 ,  0.07908108, -0.44799344,
       -0.0633093 , -0.00678783,  0.2198433 ,  0.14671942, -0.03439034,
       -0.00348209, -0.09653135,  0.57247028, -0.01030957,  0.01752099,
       -0.28550714])), (0.7280043330838262, array([ 0.05511072, -0.78329361
,  0.00697733,  0.0536764 , -0.00955118,
       -0.01656977,  0.48109722,  0.00097667,  0.01527847, -0.06338062,
        0.36613747,  0.00511128,  0.02501144,  0.0227477 , -0.00755623,
        0.00445372, -0.00900412,  0.06538165, -0.05436753,  0.0029827 ,
       -0.04229652])), (0.5932625897104592, array([ 0.08706448,  0.00545749
, -0.0034349 ,  0.08679837,  0.00092871,
       -0.12863427,  0.00170641,  0.00270761,  0.03741784, -0.20977954,
       -0.00969127,  0.00480467,  0.08951786,  0.7557533 ,  0.00232152,
       -0.00377007, -0.25503142, -0.49054782, -0.00229538,  0.00972315,
        0.19439057])), (0.49507822201001817, array([ 0.0061115 , -0.0043314
8, -0.39702894,  0.00978496, -0.326183  ,
       -0.02469706, -0.00221828, -0.56565371,  0.00703548,  0.0242312 ,
        0.00961224, -0.44090214, -0.00782563,  0.00218142, -0.00241129,
       -0.36604576, -0.00063145, -0.00583144,  0.0020868 , -0.29327776,
        0.00248777])), (0.42459227401910804, array([-0.25601878,  0.0030189
1, -0.01401228, -0.26186666, -0.01230844,
        0.68186925,  0.00660934, -0.02221772, -0.11328641, -0.61047832,
       -0.00500577, -0.01700735,  0.11405874, -0.00493854, -0.01068849,
       -0.01720124, -0.01517472, -0.00561614, -0.00376258, -0.00541619,
       -0.02142869])), (0.24578342585721696, array([ 4.72527718e-03, -6.896
08343e-04, -1.39617902e-03,  1.76246229e-03,
        9.39495186e-04, -2.88029131e-02, -4.07362480e-03,  1.33602604e-03,
       -5.08803535e-01, -5.45688749e-03,  2.34791360e-03, -2.20677414e-04,
       -5.06749693e-01, -1.36409572e-02,  6.18426366e-03,  1.38627052e-03,
       -5.01391694e-01, -6.94347795e-02, -1.43633754e-03, -1.30781693e-03,
       -4.76410336e-01]))]
Eigenvalues in descending order:
 [1.9508481917521363, 1.858059018594852, 1.3648236325070462, 1.339369849678
526, 1.2497563300791545, 1.2427434342899308, 1.140417283992402, 1.091838129
3928954, 1.0808848400720503, 1.0560111548119322, 1.0268846095348936, 0.9675
674729748748, 0.8679045475101522, 0.793607744075387, 0.7521391924764491, 0.
7305987250350349, 0.7280043330838262, 0.5932625897104592, 0.495078222010018
17, 0.42459227401910804, 0.24578342585721696]
```

Now, let's plot this.

In [64]:
```
# The sum of all eigenvalues
tot = sum(eigenvalues)

# Variance explained by each eigenvector
var_explained = [(i / tot) for i in sorted(eigenvalues, reverse=True)]
# Cumulative variance
cum_var_exp = np.cumsum(var_explained)

plt.bar(range(1,22), var_explained, alpha=0.5, align='center', label='indiv
plt.ylabel('Variance ratio')
plt.xlabel('Principal Components')
plt.legend(loc = 'best')
plt.show()
```



In [69]:
```
plt.step(range(1,22),cum_var_exp, where= 'mid', label='cumulative variance
plt.ylabel('Cumulative Variance')
plt.xlabel('Principal Components')
plt.legend(loc = 'best')
plt.show()
```



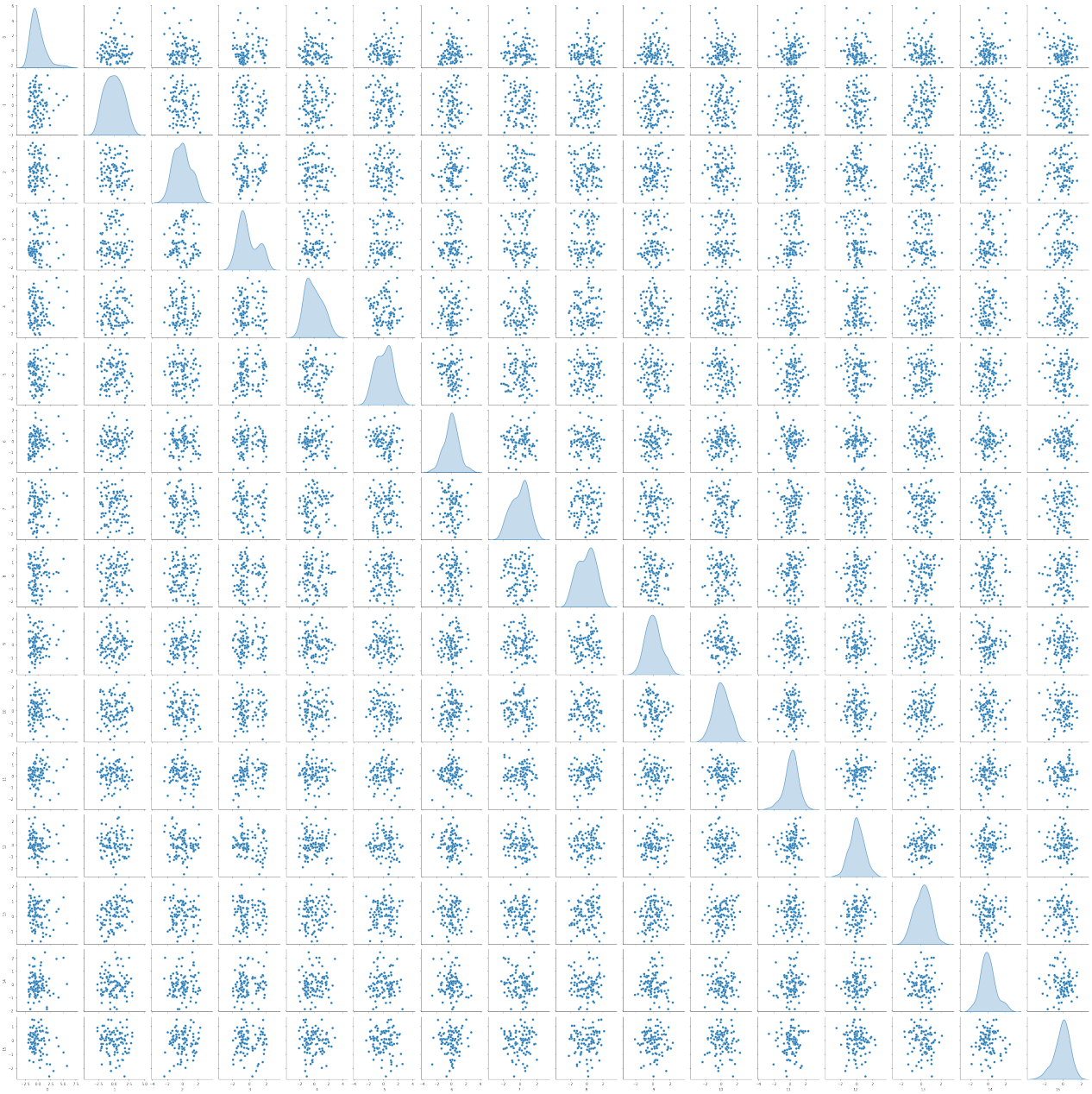
In [70]:
```
print(var_explained)
print(cum_var_exp)
```

```
[0.0928967587944701, 0.08847826356046178, 0.06499105995127502, 0.0637789851
553863, 0.05951171025919385, 0.05917776562355454, 0.054305132405477886, 0.0
51991858606753205, 0.05147027774754206, 0.05028582641518932, 0.048898859626
81655, 0.04607425761488572, 0.041328443570107454, 0.037790530032262855, 0.0
35815853554754536, 0.034790125557729826, 0.034666584113383365, 0.0282503640
8835929, 0.02357495696943658, 0.020218511226198, 0.011703875126761978]
[0.09289676 0.18137502 0.24636608 0.31014507 0.36965678 0.42883454
 0.48313968 0.53513153 0.58660181 0.63688764 0.6857865  0.73186076
 0.7731892  0.81097973 0.84679558 0.88158571 0.91625229 0.94450266
 0.96807761 0.98829612 1.        ]
```

From this, we can see that the last 5 dimensions seem to have the least impact on the variance of the dataset (the remaining 16 contribute towards almost 90% of the variance).

Hence, we'll get rid of them, and rebuild the classifier on this to see the effects of this reduction on its performance.

In [76]:
```
# p_reduce represents reduced mathematical space
p_reduce = np.array(eigvectors_sorted[0:16])   # Reducing from 21 to 16 di
x_16D = np.dot(x_std,p_reduce.T)   # projecting original data into princip
pairplot_16D = pd.DataFrame(x_16D).sample(100)  # converting array to data

# Pairplot
sn.pairplot(pairplot_16D, diag_kind="kde")
plt.show()
```

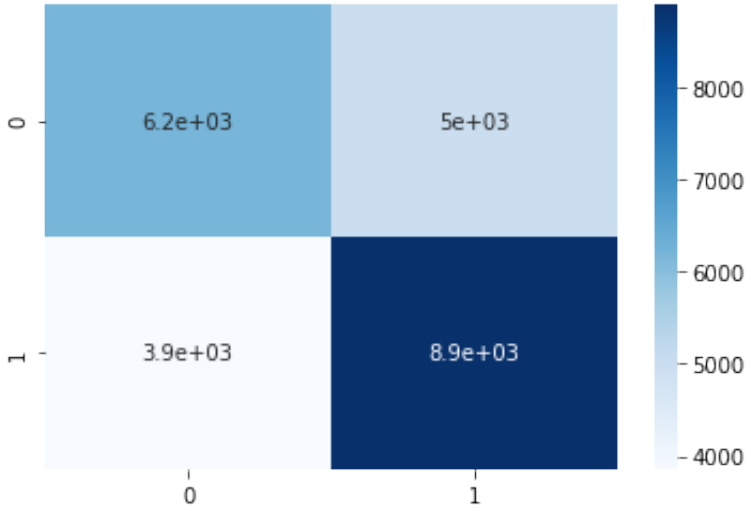


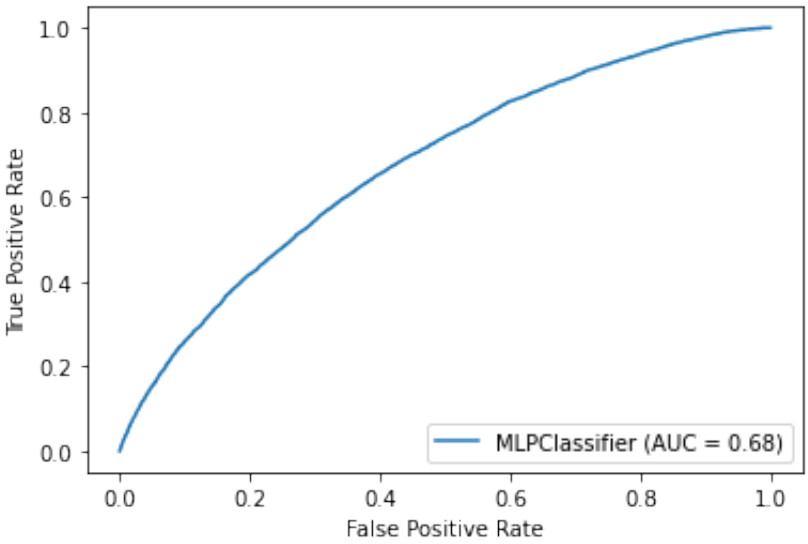
Now, let's train the MLP classifier again on this data, to see what kind of impact it has on it's accuracy and performance.

In [77]:

```
t1 = time()

# Split dataset for train and test
x_16D_train, x_16D_test, y_train, y_test = train_test_split(
    x_16D, y, test_size=0.2, random_state=3
)

print(x_16D_train.shape)
print(x_16D_test.shape)
print(y_train.shape)
print(y_test.shape)


# Create model
clf = MLPClassifier(max_iter=1000)
# Train it
clf.fit(x_16D_train, y_train)
# Evaluate it
evaluate(clf, x_16D_test, y_test)
print("Execution took ", time()-t1, "s")
```

```
(96000, 16)
(24000, 16)
(96000,)
(24000,)
Confusion Matrix:
 [[6230 5016]
 [3856 8898]]
Precision:  0.6394997843898232
Recall:  0.6976634781245099
Accuracy:  0.6303333333333333
```






```
ROC AUC 0.6776320671687397
 PR AUC 0.6876381037450132
Execution took  136.37257933616638 s
```

MLP's performance on original data, before reduction:

```
Precision:  0.6780027691242645
Recall:  0.6143170769954525
Accuracy:  0.64
ROC AUC 0.7006138903227133
 PR AUC 0.7125871079815211
```

MLP's performance on data reduced by 5 dimensions:

```
Precision:  0.6394997843898232
Recall:  0.6976634781245099
Accuracy:  0.6303333333333333
ROC AUC 0.6776320671687397
 PR AUC 0.6876381037450132
```

From this, we can see that despite a reduction of 23.8095 % from the original dataset, the accuracy of the classifier has not decreased by much.

Neural Auto Encoder

As autoencoders tend to perform better with more training data, we'll be using the full dataset here. After that, we'll use a small sample (120k rows) from it for training the classifier to validate the network.

In [6]:

```
# Get full data
full_data = df

# Drop last 7 columns
full_data = full_data.drop(full_data.columns[28],axis=1)
full_data = full_data.drop(full_data.columns[27],axis=1)
full_data = full_data.drop(full_data.columns[26],axis=1)
full_data = full_data.drop(full_data.columns[25],axis=1)
full_data = full_data.drop(full_data.columns[24],axis=1)
full_data = full_data.drop(full_data.columns[23],axis=1)
full_data = full_data.drop(full_data.columns[22],axis=1)

# Normalise all data
full_data = full_data.apply(lambda x: (x-x.min())/(x.max()-x.min()))

# Split into features and lables
x = full_data.drop(full_data.columns[0],axis=1)
y = full_data[0]

print(x.shape)
print(y.shape)

# Split into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.2, random_state=3
)

print(x_train.shape)
print(x_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(6000000, 21)
(6000000,)
(4800000, 21)
(1200000, 21)
(4800000,)
(1200000,)
```

Next, let's build an autoencoder to train on this data

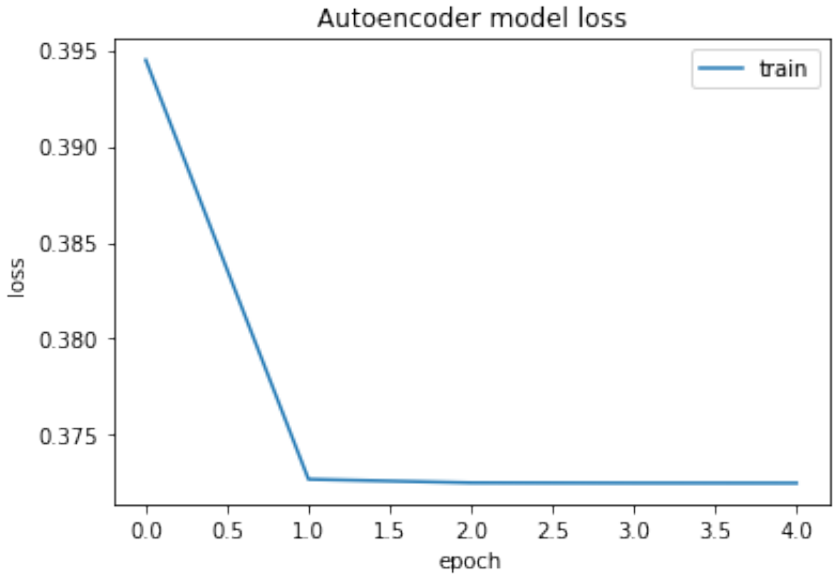
In [8]:

```
from keras.models import Sequential, Model
from keras.layers import Activation, Dense, Dropout, Input
from keras import optimizers, regularizers
```

In [9]:

```
encoding_dim = 16

# Define input layer
input_data = Input(shape=(x_train.shape[1],))

# Define encoding layer
encoded = Dense(encoding_dim, activation='relu')(input_data)

# Define decoding layer
decoded = Dense(x_train.shape[1], activation='sigmoid')(encoded)

# Create the autoencoder model
autoencoder = Model(input_data, decoded)

#Compile the autoencoder model
autoencoder.compile(optimizer='adam',
                    loss='binary_crossentropy')

#Fit to train set, validate with dev set and save to hist_auto for plottin
ae = autoencoder.fit(x_train, x_train,
                epochs=5,
                batch_size=1000,
                shuffle=True
)

# Summarize history for loss
plt.figure()
plt.plot(ae.history['loss'])
plt.title('Autoencoder model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()

# Create a separate model (encoder) in order to make encodings (first part
encoder = Model(input_data, encoded)

# Create a placeholder for an encoded input
encoded_input = Input(shape=(encoding_dim,))
# Retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# Create the decoder model
decoder = Model(encoded_input, decoder_layer(encoded_input))
```

```
Epoch 1/5
4800/4800 [==============================] - 12s 2ms/step - loss: 0.4382
Epoch 2/5
4800/4800 [==============================] - 11s 2ms/step - loss: 0.3729
Epoch 3/5
4800/4800 [==============================] - 11s 2ms/step - loss: 0.3725
Epoch 4/5
4800/4800 [==============================] - 11s 2ms/step - loss: 0.3725
Epoch 5/5
4800/4800 [==============================] - 11s 2ms/step - loss: 0.3725
```



Next, we build a new model which takes this data as input.

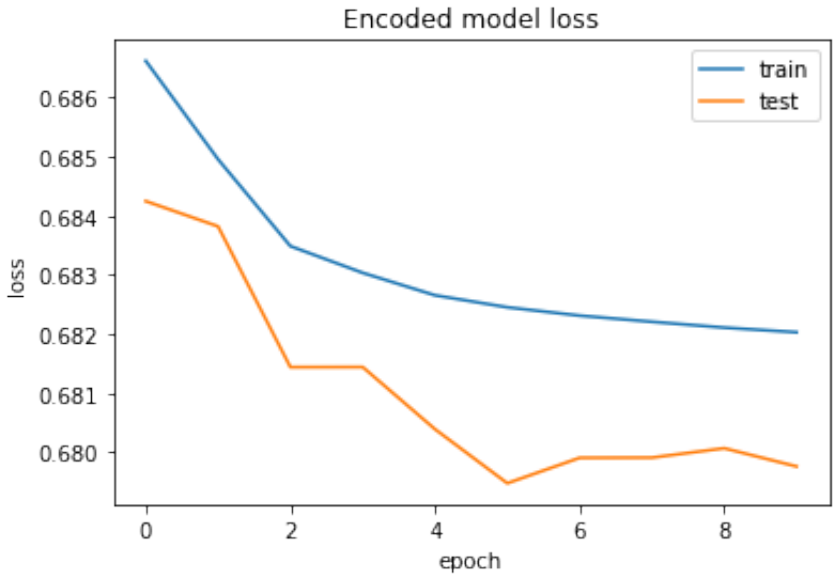
In [10]:

```
# Encode data set from above using the encoder
encoded_train_x = encoder.predict(x_train)
encoded_test_x = encoder.predict(x_test)

model = Sequential()
model.add(Dense(16, input_dim=encoded_train_x.shape[1],
                kernel_initializer='normal',
                activation="relu"
                )
          )
model.add(Dropout(0.2))
model.add(Dense(1))
model.add(Activation("sigmoid"))
model.compile(loss="binary_crossentropy", optimizer='adam')
```

In [115…

```
history = model.fit(encoded_train_x, y_train, validation_split=0.2, epochs=

# Summarize history for loss
plt.figure()
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Encoded model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()
```

```
Epoch 1/10
60000/60000 [==============================] - 77s 1ms/step - loss: 0.6877
- val_loss: 0.6842
Epoch 2/10
60000/60000 [==============================] - 77s 1ms/step - loss: 0.6852
- val_loss: 0.6838
Epoch 3/10
60000/60000 [==============================] - 76s 1ms/step - loss: 0.6836
- val_loss: 0.6814
Epoch 4/10
60000/60000 [==============================] - 76s 1ms/step - loss: 0.6831
- val_loss: 0.6814
Epoch 5/10
60000/60000 [==============================] - 76s 1ms/step - loss: 0.6827
- val_loss: 0.6804
Epoch 6/10
60000/60000 [==============================] - 76s 1ms/step - loss: 0.6824
- val_loss: 0.6795
Epoch 7/10
60000/60000 [==============================] - 76s 1ms/step - loss: 0.6824
- val_loss: 0.6799
Epoch 8/10
60000/60000 [==============================] - 76s 1ms/step - loss: 0.6823
- val_loss: 0.6799
Epoch 9/10
60000/60000 [==============================] - 76s 1ms/step - loss: 0.6821
- val_loss: 0.6801
Epoch 10/10
60000/60000 [==============================] - 78s 1ms/step - loss: 0.6821
- val_loss: 0.6798
```

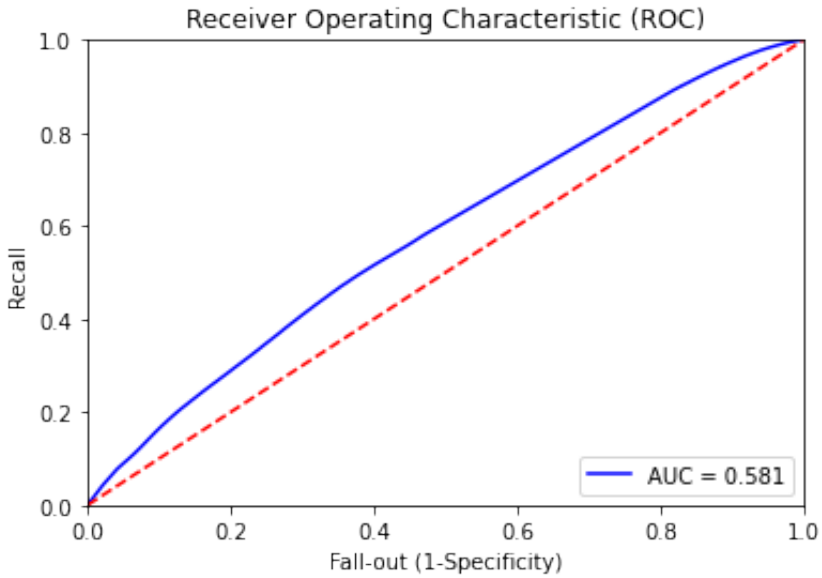


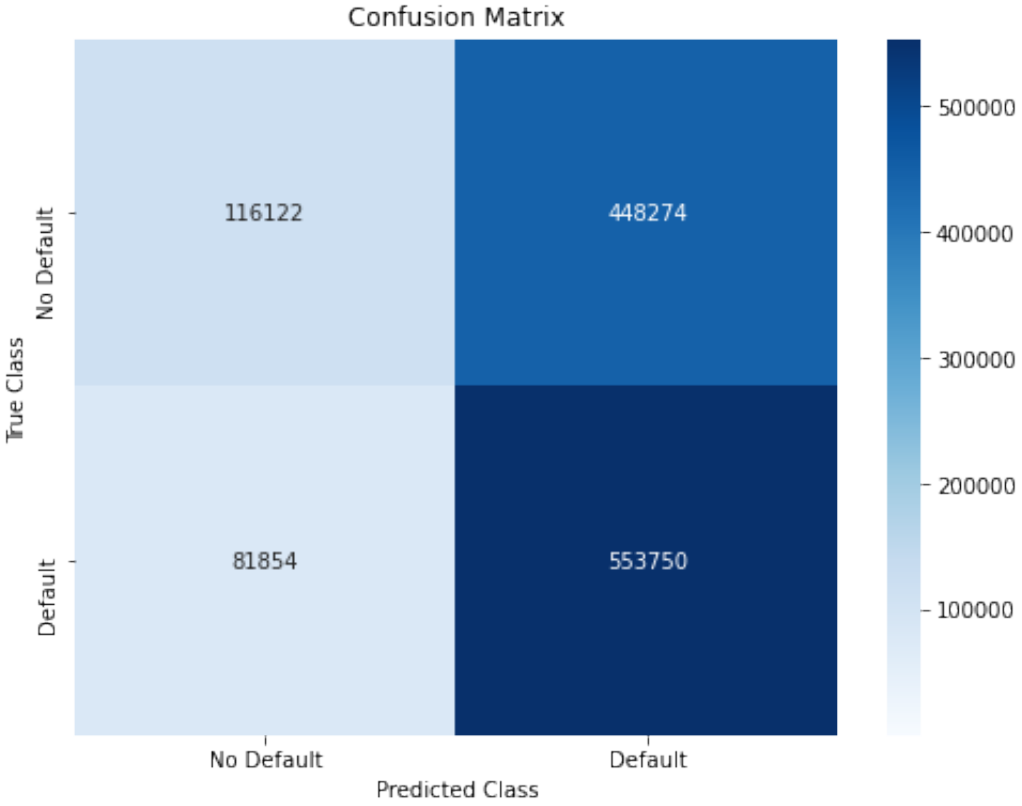
Now, we visualise the new classifier's performance.

In [118…

```
# Predict on test set
predictions_NN_prob = model.predict(encoded_test_x)
predictions_NN_prob = predictions_NN_prob[:,0]

# Turn probability to 0-1 binary output
predictions_NN_01 = np.where(predictions_NN_prob > 0.5, 1, 0)

# Accuracy
acc_NN = accuracy_score(y_test, predictions_NN_01)
print('Overall accuracy:', acc_NN)

# Area Under Curve
false_positive_rate, recall, thresholds = roc_curve(y_test, predictions_NN_
roc_auc = auc(false_positive_rate, recall)
plt.figure()
plt.title('Receiver Operating Characteristic (ROC)')
plt.plot(false_positive_rate, recall, 'b', label = 'AUC = %0.3f' %roc_auc)
plt.legend(loc='lower right')
plt.plot([0,1], [0,1], 'r--')
plt.xlim([0.0,1.0])
plt.ylim([0.0,1.0])
plt.ylabel('Recall')
plt.xlabel('Fall-out (1-Specificity)')
plt.show()

# Confusion Matrix
cm = confusion_matrix(y_test, predictions_NN_01)
labels = ['No Default', 'Default']
plt.figure(figsize=(8,6))
sn.heatmap(cm,xticklabels=labels, yticklabels=labels, annot=True, fmt='d',
plt.title('Confusion Matrix')
plt.ylabel('True Class')
plt.xlabel('Predicted Class')
plt.show()
```

```
Overall accuracy: 0.5582266666666666
```





From the above graphs, we can see that the classifier we had built to work with the autoencoder performs poorly at the classification task. This can be attributed to two reasons:

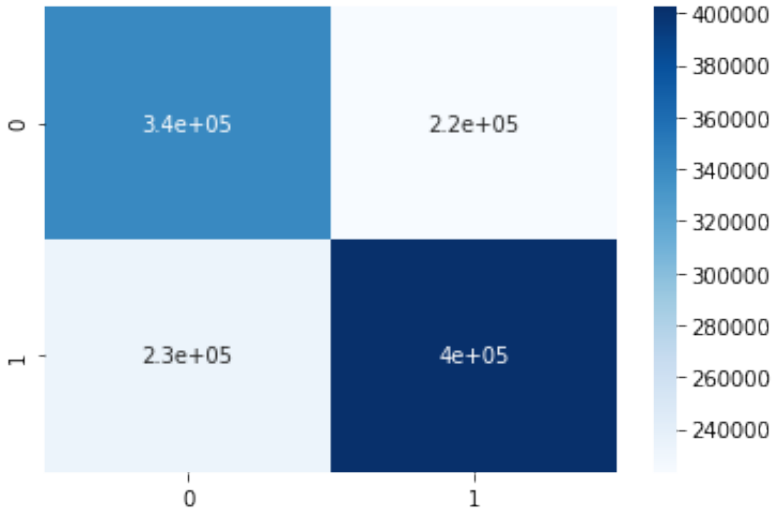
1. The autoencoder we built was a very simple one, so maybe it was unable to capture all the required features comprehensively.
2. The classifier we built is also very basic, and hence it could not perform as well as a deeper classifier could have.

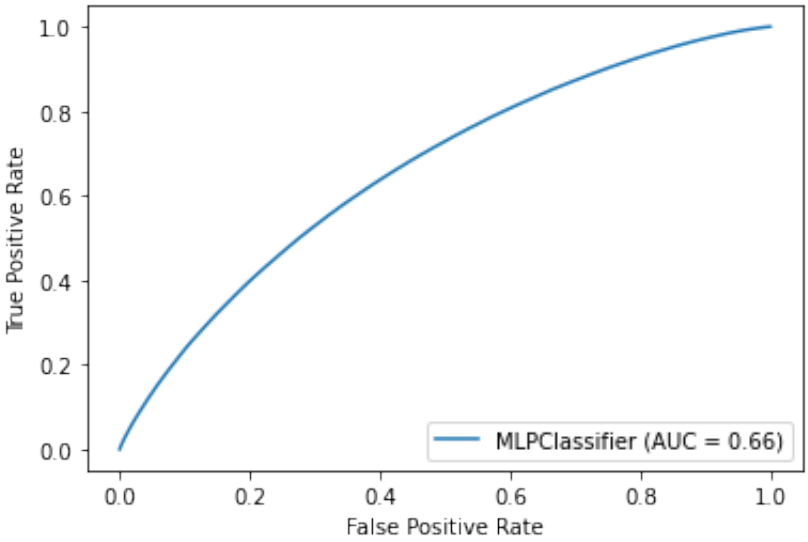
We can figure out which of the two reasons is more likely to be the contributing factor through means of a simple test.

In [11]:

```
# Use sklearn's mlp to figure out which of the above is the more likely re
t1 = time()
# Create model
clf = MLPClassifier(max_iter=100, verbose=True)
# Train it using the autoencoder's encodings
clf.fit(encoded_train_x, y_train)
# Evaluate it
evaluate(clf, encoded_test_x, y_test)
print("Execution took ", time()-t1, "s")
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/neural_network/_multilayer_p
erceptron.py:573: UserWarning: Training interrupted by user.
  warnings.warn("Training interrupted by user.")
Confusion Matrix:
 [[341119 223277]
 [233144 402460]]
Precision:  0.643177565015334
Recall:  0.6331929943801486
Accuracy:  0.6196491666666667
```






```
ROC AUC 0.6635604206843982
 PR AUC 0.6707186585465506
Execution took  4981.915852546692 s
```

Let's compare all three versions of the MLP network:

1. Normal MLP's performance
```
Precision:  0.6780027691242645
Recall:  0.6143170769954525
Accuracy:  0.64
ROC AUC 0.7006138903227133
PR AUC 0.7125871079815211
```
2. MLP's performance on PCA reduced data:

```
Precision:  0.6394997843898232
Recall:  0.6976634781245099
Accuracy:  0.6303333333333333
ROC AUC 0.6776320671687397
PR AUC 0.6876381037450132
```
3. MLP on AE reduced data:

```
Precision:  0.643177565015334
Recall:  0.6331929943801486
Accuracy:  0.6196491666666667
ROC AUC 0.6635604206843982
PR AUC 0.6707186585465506
```

From this, we can infer that the basic Auto Encoder performs reasonably well, with results comparable to that from PCA.

Future Work

We've used PCA and an Auto Encoder for dimensionality reduction, and it has performed reasonably well. The next natural steps would be to

1. Perform LDA, as another dimensionality reduction technique.
2. Perform PCA on the latent vector from the Deep Autoencoder to further reduce the dimensionality of the task at hand.
3. Build better Auto Encoders for the task.
4. Run on an instance with more RAM. During training, the instance crashed multiple times due to which we had to stop at just 5 epochs.
5. Use better classifiers to benchmark the AutoEncoder.

Concluding Thoughts

For the major part of this exercise, we were only working with 21 of the 28 provided columns. Also, we built our classifiers for Part A on an even smaller sample set from the provided data. With this in mind, these were our findings:

Observations :

1. A multi-layer perceptron performed the best at the classification task.
2. PCA provided 23% reduction by eliminating 5 columns, without having a significant impact on the Multi Layer Perceptron's performance.
3. The AutoEncoder's performance is comparable to that offered through PCA.

Inferences :

1. Since the MLP performed the best, the next logical thing to do is to build better deep neural networks using frameworks like TensorFlow or PyTorch.
2. PCA can be used as a dimensionality reduction technique.
3. The AutoEncoder's performance can be improved by using better networks.

Future work :

1. Try LDA and see how it compares to PCA for dimensionality reduction.
2. Build better classifiers to benchmark the effects of dimensionality reduction.
3. Build better Auto Encoders to achieve better dimensionality reduction.
4. Try replacing the columns 9, 13, 17 and 21 with one-hot encodings and evaluate the impact it has on models.
5. Use scaled data on all classifiers, instead of just for dimensionality reduction.