This is the CLASSIFICATION jupyter notebook

In [1]:
```
# general libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import keras
```

In [2]:
```
# models
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier as RT
from keras.models import Sequential
from keras.layers import Dense
```

In [3]:
```
# preprocessing and set up
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.model_selection import train_test_split as tts
from imblearn.over_sampling import RandomOverSampler as ROS
from imblearn.under_sampling import RandomUnderSampler as RUS
from sklearn.model_selection import GridSearchCV
```

In [4]:
```
# scoring
from sklearn.metrics import roc_auc_score as ARS, precision_score as PS, ac
from sklearn.metrics import recall_score as RS, f1_score as FS, confusion_m
from sklearn.metrics import roc_curve as RC, roc_auc_score as RAS
```

In [5]:
```
ccfdf = pd.read_csv('creditcard.csv')
ccfdf.dropna()
ccfdf.info()
ccfdf.describe()
print(ccfdf["Class"].value_counts())
print(ccfdf.shape)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64
 3   V3      284807 non-null  float64
 4   V4      284807 non-null  float64
 5   V5      284807 non-null  float64
 6   V6      284807 non-null  float64
 7   V7      284807 non-null  float64
 8   V8      284807 non-null  float64
 9   V9      284807 non-null  float64
 10  V10     284807 non-null  float64
 11  V11     284807 non-null  float64
 12  V12     284807 non-null  float64
 13  V13     284807 non-null  float64
 14  V14     284807 non-null  float64
 15  V15     284807 non-null  float64
```

```
 16  V16     284807 non-null  float64
 17  V17     284807 non-null  float64
 18  V18     284807 non-null  float64
 19  V19     284807 non-null  float64
 20  V20     284807 non-null  float64
 21  V21     284807 non-null  float64
 22  V22     284807 non-null  float64
 23  V23     284807 non-null  float64
 24  V24     284807 non-null  float64
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
 30  Class   284807 non-null  int64
dtypes: float64(30), int64(1)
memory usage: 67.4 MB
0    284315
1       492
Name: Class, dtype: int64
(284807, 31)
```

In [6]:

```
y = ccfdf["Class"]
X = ccfdf.drop("Class",axis = 1)
print(y.value_counts())
```

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

In [7]:

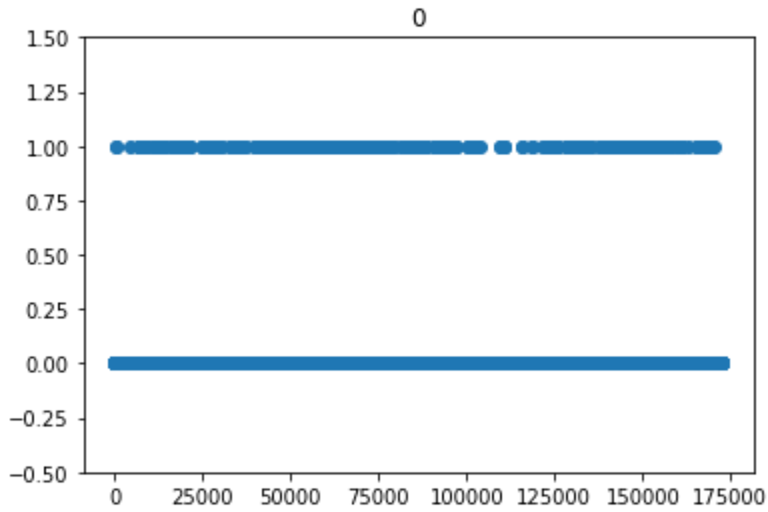
```
X.info()
```

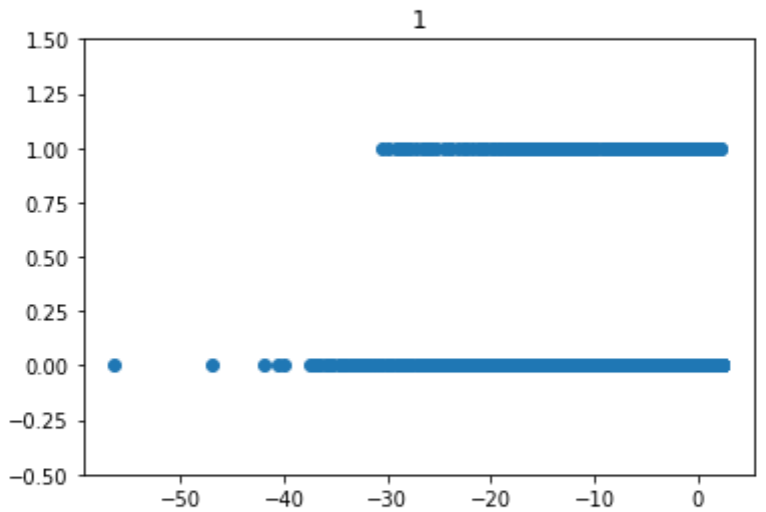
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 30 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Time    284807 non-null  float64
 1   V1      284807 non-null  float64
 2   V2      284807 non-null  float64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

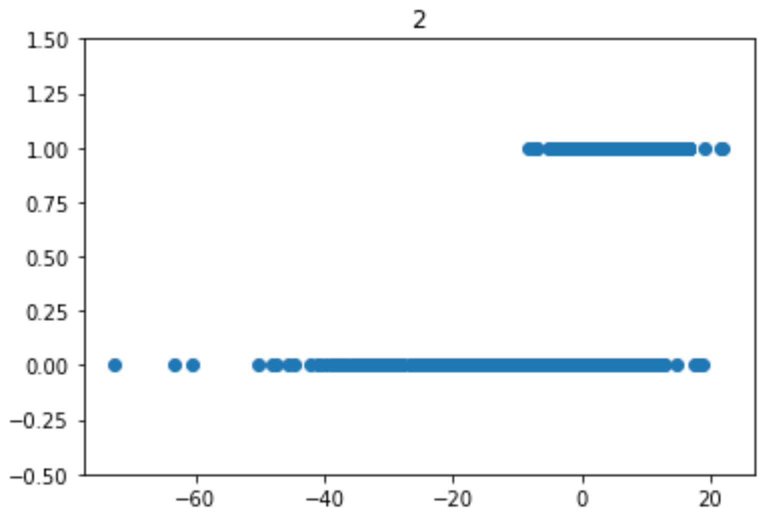
```
 25  V25     284807 non-null  float64
 26  V26     284807 non-null  float64
 27  V27     284807 non-null  float64
 28  V28     284807 non-null  float64
 29  Amount  284807 non-null  float64
dtypes: float64(30)
memory usage: 65.2 MB
```

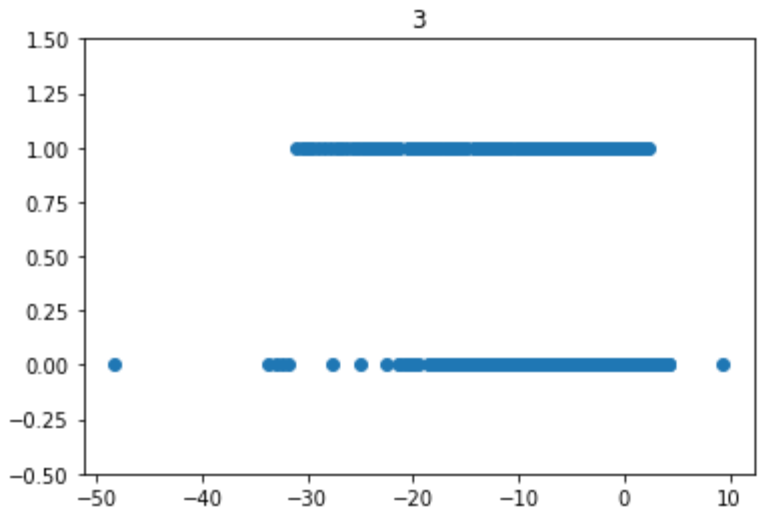
In [8]:

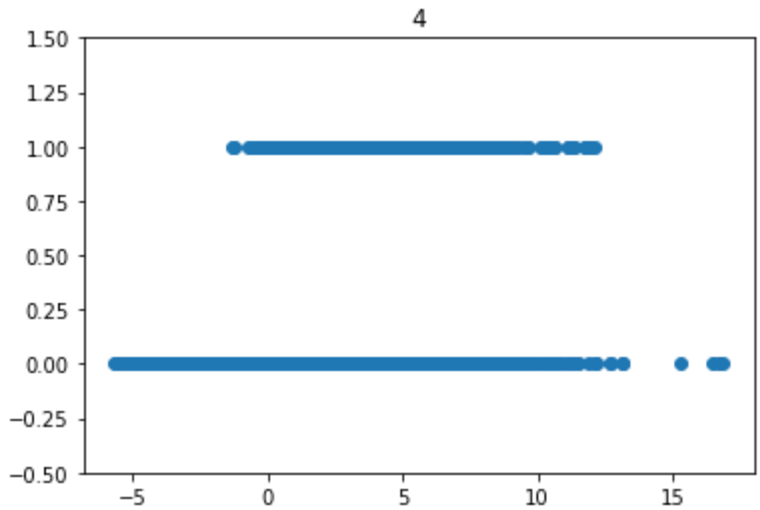
```
for i in range (0,30):
    plt.title("%d" % i)
    plt.scatter(X.iloc[:,[i]],y)
    plt.ylim(-.5,1.5)
    plt.show()
```

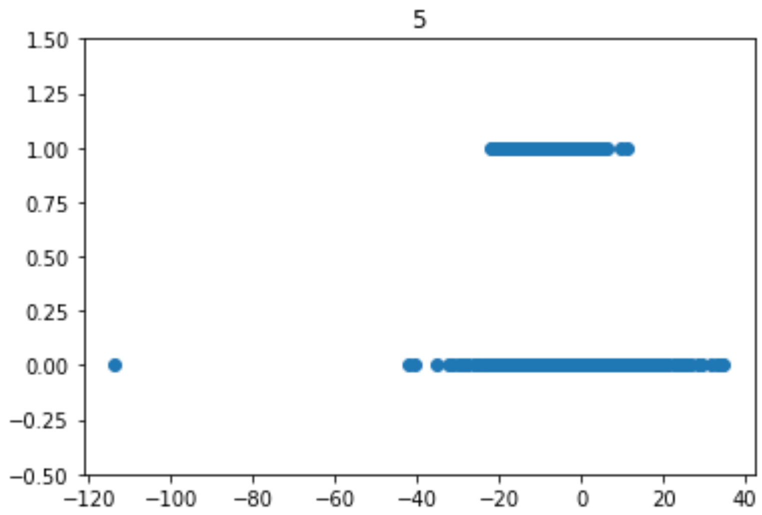


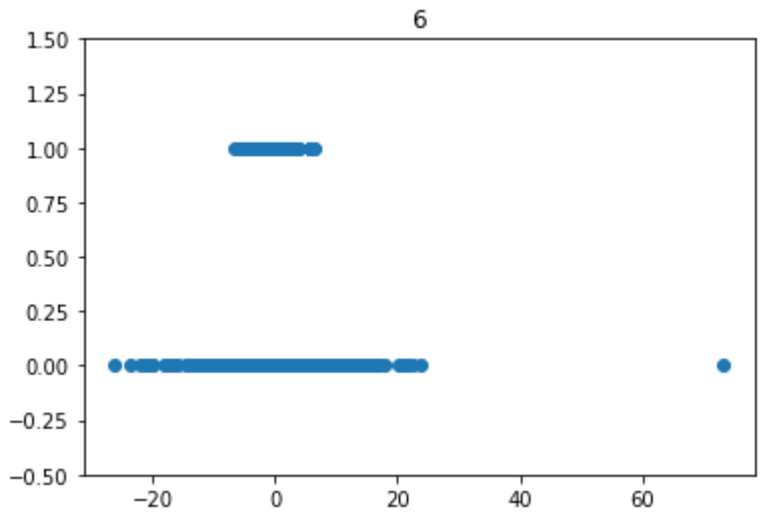


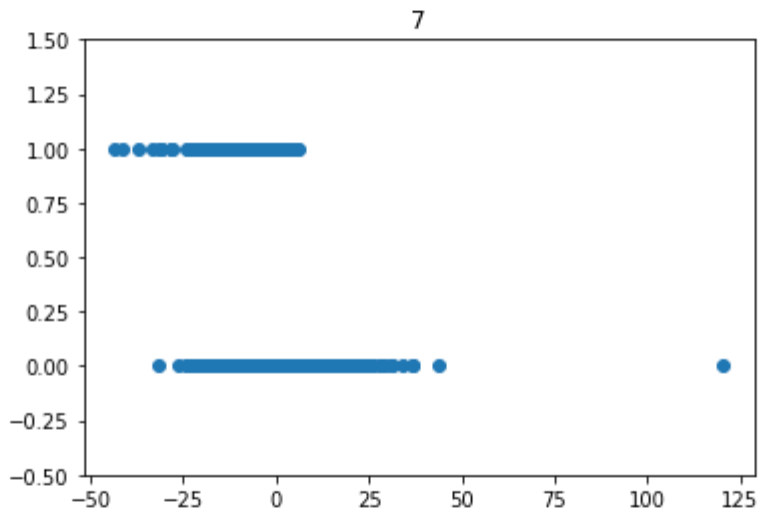

2
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−60
−40
−20
0
20

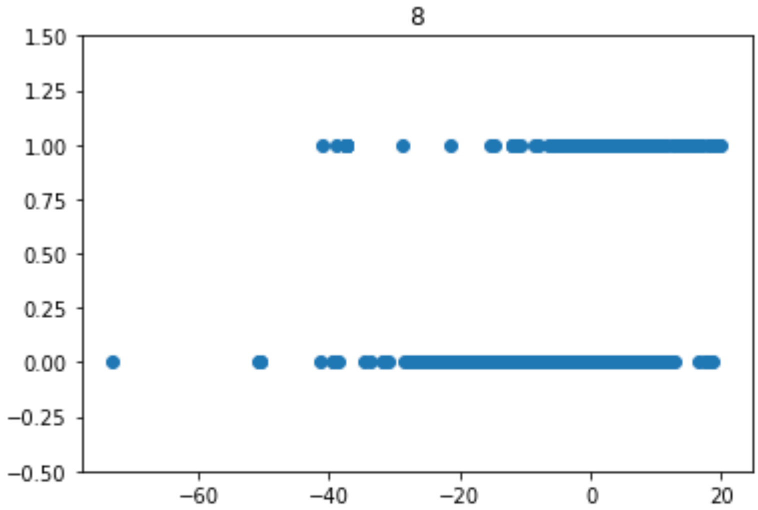

3
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−50
−40
−30
−20
−10
0
10

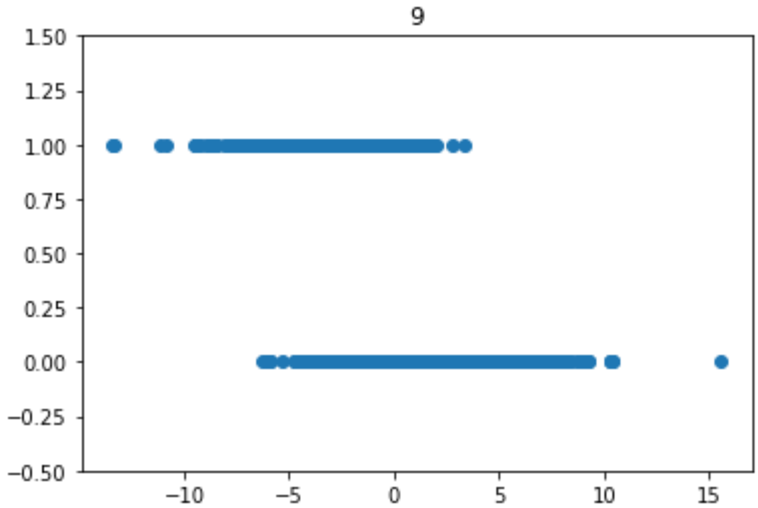

4
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−5
0
5
10
15

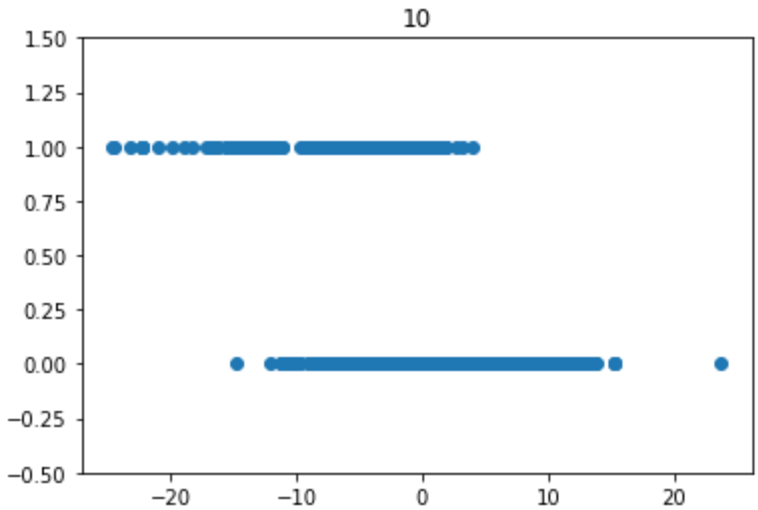

5
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−120
−100
−80
−60
−40
−20
0
20
40

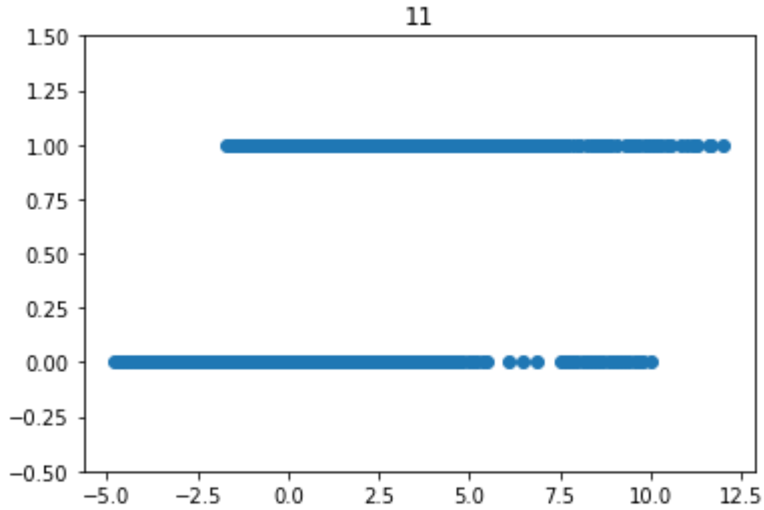

6
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−20
0
20
40
60

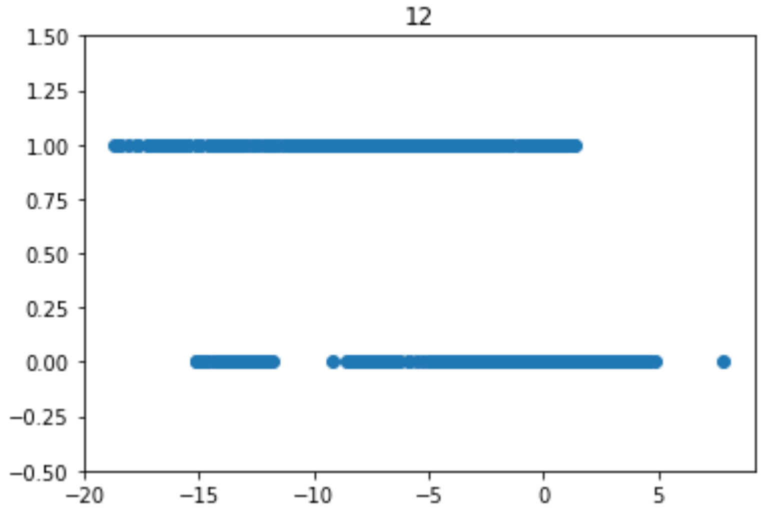

7
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−50
−25
0
25
50
75
100
125

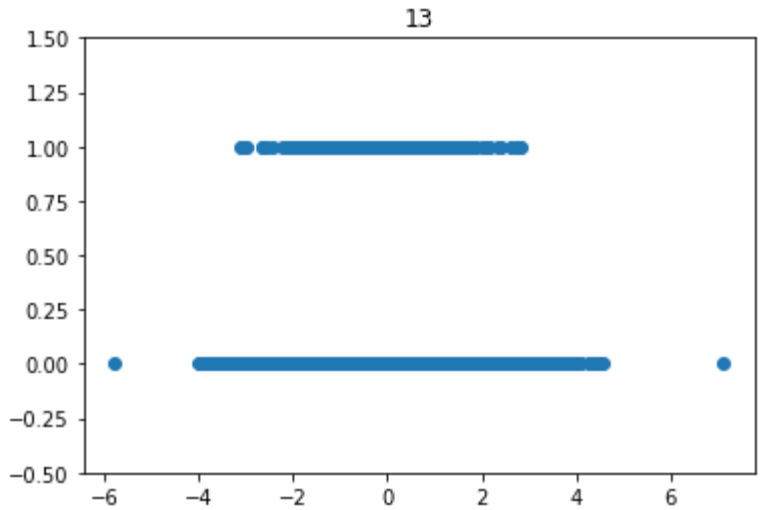

8
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−60
−40
−20
0
20

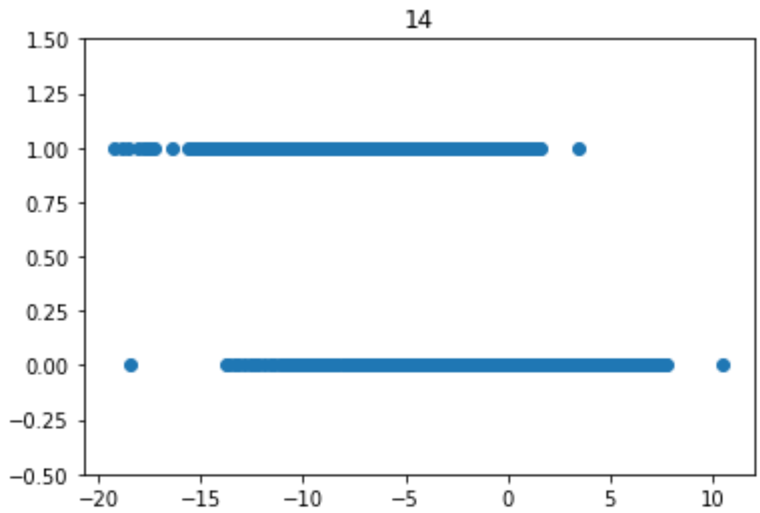

9
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−10
−5
0
5
10
15

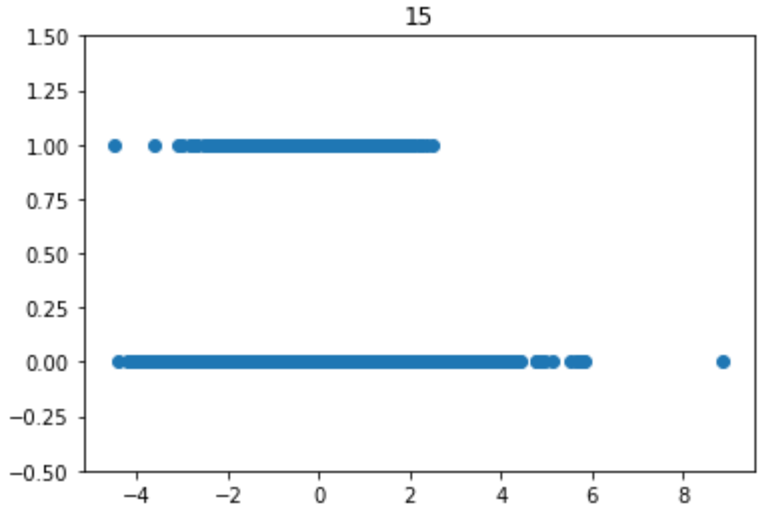

10
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−20
−10
0
10
20

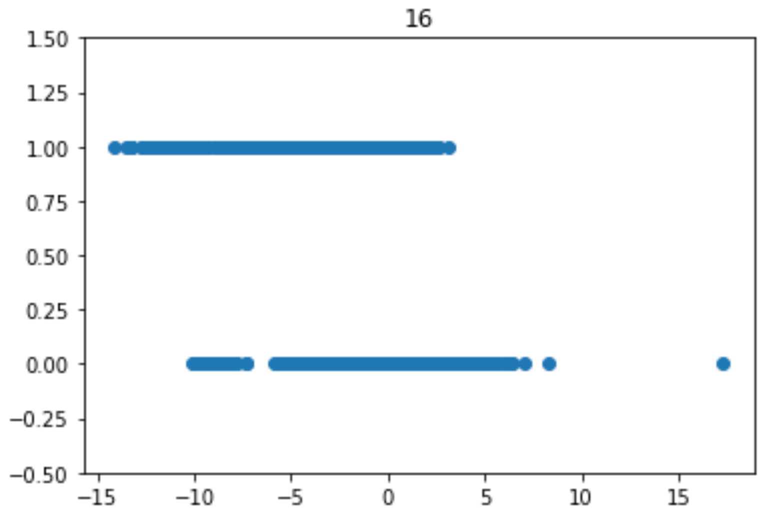

11
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5

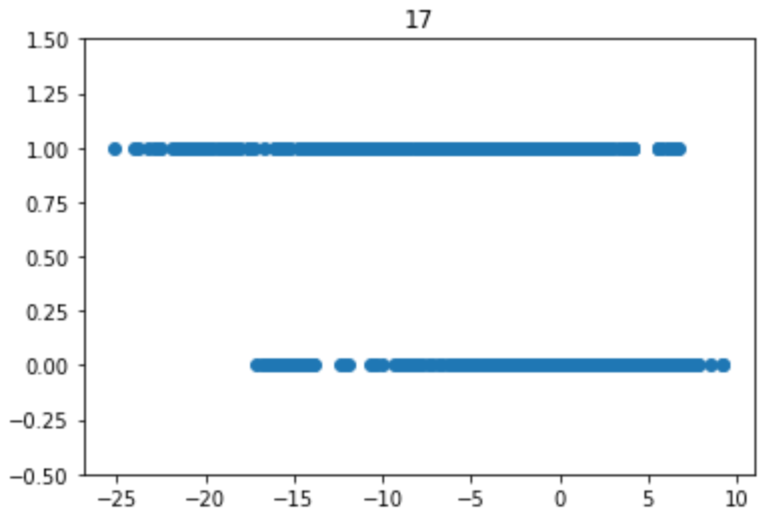

12
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−20
−15
−10
−5
0
5

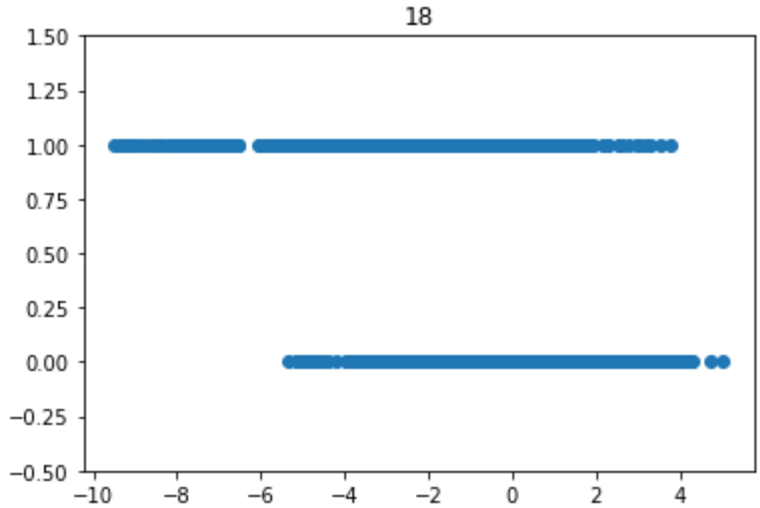

13
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−6
−4
−2
0
2
4
6

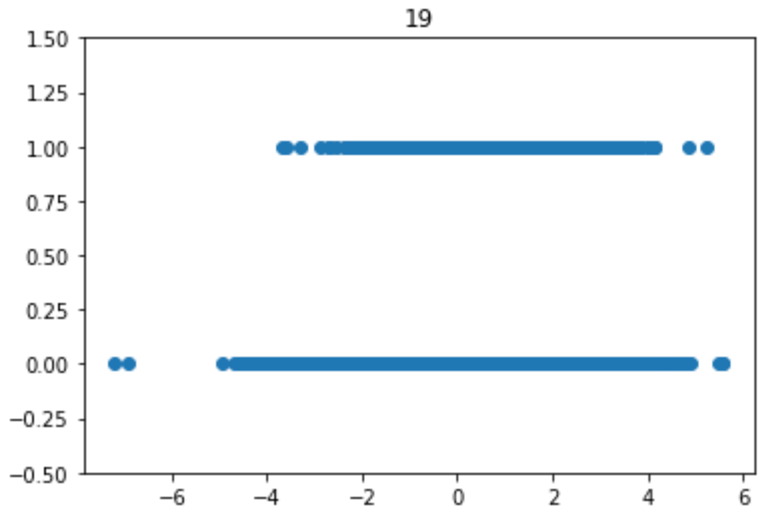

14
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−20
−15
−10
−5
0
5
10

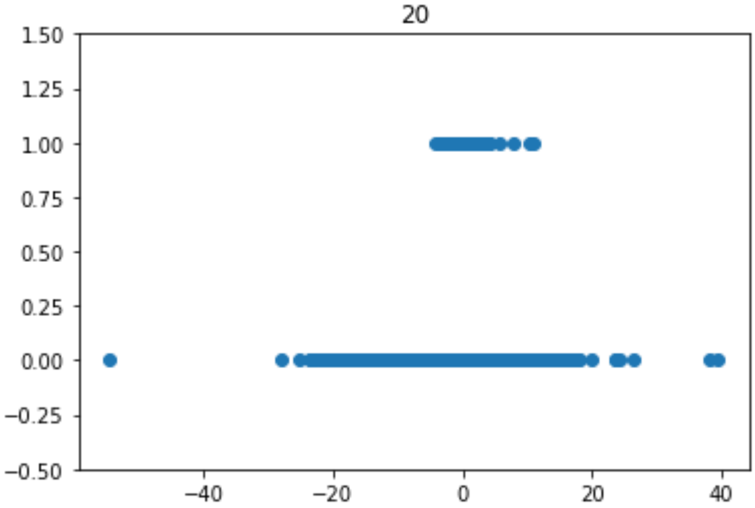

15
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−4
−2
0
2
4
6
8

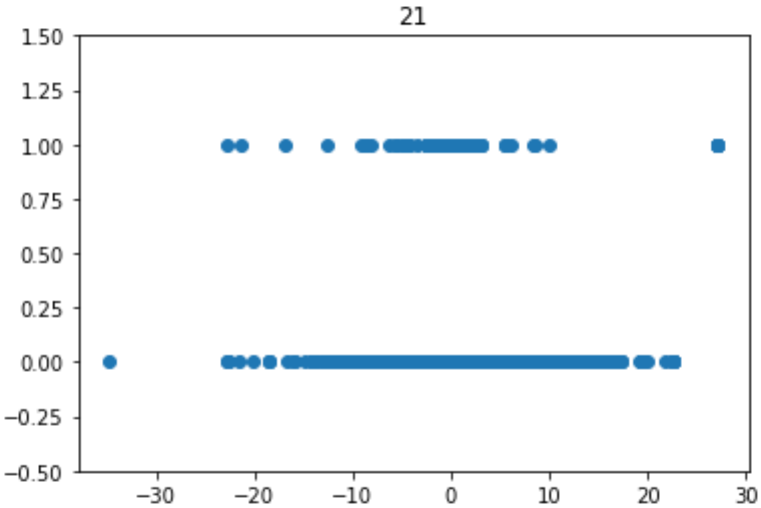

16
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−15
−10
−5
0
5
10
15

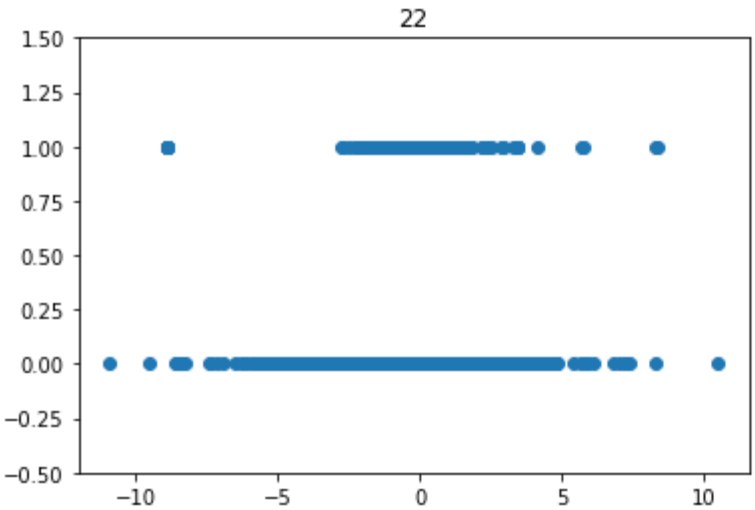

17
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−25 −20 −15 −10 −5 0 5 10

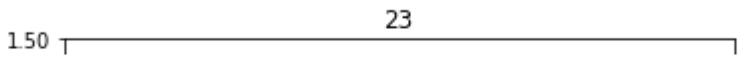

18
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−10 −8 −6 −4 −2 0 2 4


19
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−6 −4 −2 0 2 4 6


20
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−40
−20
0
20
40


21
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−30
−20
−10
0
10
20
30


22
1.50
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−10
−5
0
5
10



> Looking at these graphs graph "0" is not really indicative of Fraud, because the min and max of legitimate and fraudulent credit card usage is almost the same and the gaps of time where there are no fraudulent credit card use are so minimal it would do nothing to actually help train the modle. Also logically how would time affect someone trying to steal money?

In [9]:

```
X = X.drop("Time",axis = 1)
X.info()
X = X.values
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 284807 entries, 0 to 284806
Data columns (total 29 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   V1      284807 non-null  float64
 1   V2      284807 non-null  float64
 2   V3      284807 non-null  float64
 3   V4      284807 non-null  float64
 4   V5      284807 non-null  float64
 5   V6      284807 non-null  float64
 6   V7      284807 non-null  float64
 7   V8      284807 non-null  float64
 8   V9      284807 non-null  float64
 9   V10     284807 non-null  float64
 10  V11     284807 non-null  float64
 11  V12     284807 non-null  float64
 12  V13     284807 non-null  float64
 13  V14     284807 non-null  float64
 14  V15     284807 non-null  float64
 15  V16     284807 non-null  float64
 16  V17     284807 non-null  float64
 17  V18     284807 non-null  float64
 18  V19     284807 non-null  float64
 19  V20     284807 non-null  float64
 20  V21     284807 non-null  float64
 21  V22     284807 non-null  float64
 22  V23     284807 non-null  float64
 23  V24     284807 non-null  float64
 24  V25     284807 non-null  float64
 25  V26     284807 non-null  float64
 26  V27     284807 non-null  float64
 27  V28     284807 non-null  float64
 28  Amount  284807 non-null  float64
dtypes: float64(29)
memory usage: 63.0 MB
```

In [10]:

```
print(X.shape)
```

```
(284807, 29)
```

> I am not really sure which is really better: oversampling or undersampling. From seemingly common sense it should be oversampling because one keeps all the original data, but i am not an expert. So, let us find out by using both.

In [11]:

```
# time to train because i want to go on a side of caution instead of 8:2 ra
x_tr_f, x_te, y_tr_f, y_te = tts(X,y, test_size = 0.3)
#over and under samplers
ros = ROS(random_state = 42) # why 42? why not 42 is the better question
rus = RUS(random_state = 42)
# resampling the training sets
xtr_os_f, ytr_os_f = ros.fit_resample(x_tr_f,y_tr_f) # (x) or (y) (TR)ain
xtr_us_f, ytr_us_f = rus.fit_resample(x_tr_f,y_tr_f) # (x) or (y) (TR)ain
# now for splitting for neural network i will revert to 8:2 for validation
xtr_os, xtr_os_v, ytr_os, ytr_os_v = tts(xtr_os_f, ytr_os_f, test_size=0.2)
xtr_us, xtr_us_v, ytr_us, ytr_us_v = tts(xtr_us_f, ytr_us_f, test_size=0.2)
```

In [12]:

```
# Scaling for SVC and DNN, since decision tree aren't sensitive to feature
#oversampling
scaler_os = StandardScaler()
sc_xtr_os_f = scaler_os.fit_transform(xtr_os_f)
sc_xtr_os = scaler_os.transform(xtr_os)
sc_xte_os = scaler_os.transform(x_te)
sc_xte_os_v = scaler_os.transform(xtr_os_v)
#undersampling
scaler_us = StandardScaler()
sc_xtr_us_f = scaler_us.fit_transform(xtr_us_f)
sc_xtr_us = scaler_us.transform(xtr_us)
sc_xte_us = scaler_us.transform(x_te)
sc_xte_us_v = scaler_us.transform(xtr_us_v)
```

Random Forest

> Since Random Forests are insensitive to feature scaling due to being tree based, i will use xtr_os_f, ytr_os_f, xtr_us_f, ytr_us_f, x_te, and y_te since DT's also do not need validation sets

In [13]:

```
print(xtr_os_f.shape, ytr_os_f.shape)
print(xtr_us_f.shape, ytr_us_f.shape)
```

```
(398064, 29) (398064,)
(664, 29) (664,)
```

> I will be usign grid search to find the best params but will limit the sizes of X and y when doing so because i want an idea of what is the best and do not have the time to sit though almost 400,000 samples being tests only to then do it again later, I'd rather to a large enough number like 4000 - roughly a 1000th of the training data but like 4000 is still 4000 - to use to get a good idea of the params, maybe not the best but still better than whatever I could come up with no idea to begin with.

In [14]:

```
# lets use gridsearch to find best params
os_params = {'n_estimators':np.arange(100,200,20), 'criterion':('gini','ent
us_params = {'n_estimators':np.arange(100,200,20), 'criterion':('gini','ent
# first attempt at low precision fix: max features params is an afterthough
# second attempt at low precision fix: array of n_estimators up to 500 from
os_GS = GridSearchCV(RT(), os_params, refit = False)
os_GS.fit(xtr_os_f[:4000,:], ytr_os_f[:4000])
print("oversampled params: ")
print(os_GS.best_params_)

us_GS =  GridSearchCV(RT(), us_params, refit = False)
us_GS.fit(xtr_us_f, ytr_us_f)
print("undersampled params: ")
print(us_GS.best_params_)
```

```
oversampled params:
{'criterion': 'gini', 'max_features': 'sqrt', 'n_estimators': 100}
undersampled params:
{'criterion': 'gini', 'max_features': 'sqrt', 'max_leaf_nodes': 18, 'n_esti
mators': 100}
```

In [15]:

```
os_rt = RT(n_estimators = 100, criterion = 'gini', max_features = 'sqrt')
us_rt = RT(n_estimators = 100, criterion = 'gini', max_features = 'sqrt', m
```

In [16]:

```
os_rt.fit(xtr_os_f, ytr_os_f)
us_rt.fit(xtr_us_f, ytr_us_f)
```

Out[16]:

```
RandomForestClassifier(max_features='sqrt', max_leaf_nodes=18)
```

In [17]:

```
os_rt_pred = os_rt.predict(x_te)
us_rt_pred = us_rt.predict(x_te)
```

In [18]:

```
print("For Oversampled: ")
os_rt_cm = CM (y_te,os_rt_pred)
print(os_rt_cm)
print('\n')
print("For Undersampled: ")
us_rt_cm = CM (y_te,us_rt_pred)
print(us_rt_cm)
```

```
For Oversampled:
[[85276     7]
 [   35   125]]


For Undersampled:
[[83611  1672]
 [   17   143]]
```

In [19]:

```
print("For Oversampled: ")
print('Accuracy: {}'.format(AS(y_te, os_rt_pred)))
print('Precision: {}'.format(PS(y_te, os_rt_pred)))
print('Recall: {}'.format(RS(y_te, os_rt_pred)))
print('F1 Score: {}'.format(FS(y_te, os_rt_pred)))
print('\n')
print("For Undersampled: ")
print('Accuracy: {}'.format(AS(y_te, us_rt_pred)))
print('Precision: {}'.format(PS(y_te, us_rt_pred)))
print('Recall: {}'.format(RS(y_te, us_rt_pred)))
print('F1 Score: {}'.format(FS(y_te, us_rt_pred)))
```

```
For Oversampled:
Accuracy: 0.9995084442259752
Precision: 0.946969696969697
Recall: 0.78125
F1 Score: 0.8561643835616438


For Undersampled:
Accuracy: 0.9802324356588603
Precision: 0.07878787878787878
Recall: 0.89375
F1 Score: 0.1448101265822785
```

> So before adding the max_features param the precision was about 0.1 and 0.06 meaning there was an increase but not enough to make a good model since now it is 0.14 and 0.07. I will try to see if adding more trees now for fixing precision before adding it to param list by setting n_estimators to 200 if there is improvement, i'll add n_estimators to params
>
> adding more trees via setting n_estimators to 200 from its default 100 increased precision cores from 0.14 and 0.07 to 0.195 and 0.114, adding n_estimators to params, and will try removing max leaf node models and if it helps i'll from params
>
> max leaf nodes removal made oversampled set precision rise greatly in fact to 0.92, but the under sampled dropped to 0.089 so i will split params into os_params and us_params for the different grid searches
>
> i am not sure what else i could be doing to improve the undersampled training set results, but considering at least the over samples training set results are working with a 99% accuracy and nearly 85% F1 score i am going to consider that and win.

In [20]:

```
# lets plot the roc curves and document the area under the curve
os_rt_fpr, os_rt_tpr, _ = RC(y_te, os_rt.predict_proba(x_te)[:,1])
os_rt_auc = RAS(y_te, os_rt_pred)
plt.plot(os_rt_fpr, os_rt_tpr, linestyle='--', label = "Oversampled(AUC =%.

us_rt_fpr, us_rt_tpr, _ = RC(y_te, us_rt.predict_proba(x_te)[:,1])
us_rt_auc = RAS(y_te, us_rt_pred)
plt.plot(us_rt_fpr, us_rt_tpr, linestyle='--', label = "Undersampled(AUC =%
plt.legend()
plt.show()
```