Credit card fraud detection using Naive Bayes

Hanah Chang

1. Introduction

In this project, we are going to detect fraudulent credit card transactions using Naive Bayes algorithm. The dataset is from (https://www.kaggle.com/mlg-ulb/creditcardfraud/home). It contains 284,807 credit card transactions made in September 2013 by European cardholders. It includes 30 variables and 28 of which are principal components obtained from PCA. Our class is binary variable, 1 in case of fraud 0 otherwise. The author says the data contains only numerical input variables which are the result of a Principal Component Analysis(PCA) transformation due to confidentiality issues.

PCA is a mathmetical procedure where it transforms variables into smaller number of uncorrelated variables(PC), by which it reduces dimensions of the data without losing any information. To put it simply, we can assume the principal components from the dataset (v1, v2, v3…) are new variables which are consist of important fraction of all original variables.

Although we may not know what are the original features as well as the relationship between each attribute and our class, we can still use this data to achieve our goal, which is predict frauds accurately.

2. Data & Libaray

From min, max and std values, we know that variable 'time' and 'amount' need normalization / scaling.

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

In [3]:
```
df = pd.read_csv("creditcard.csv")
```

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

Out[4]:

	Time	V1	V2	V3	V4	V5
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05
mean	94813.859575	3.919560e-15	5.688174e-16	-8.769071e-15	2.782312e-15	-1.552563e-15
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01

8 rows × 31 columns

In [6]:
```
df.head(10)
```

Out[6]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.
5	2.0	-0.425966	0.960523	1.141109	-0.168252	0.420987	-0.029728	0.476201	0.260314	-0.
6	4.0	1.229658	0.141004	0.045371	1.202613	0.191881	0.272708	-0.005159	0.081213	0.
7	7.0	-0.644269	1.417964	1.074380	-0.492199	0.948934	0.428118	1.120631	-3.807864	0.
8	7.0	-0.894286	0.286157	-0.113192	-0.271526	2.669599	3.721818	0.370145	0.851084	-0.
9	9.0	-0.338262	1.119593	1.044367	-0.222187	0.499361	-0.246761	0.651583	0.069539	-0.

10 rows × 31 columns

3. Explanatory Analysis

There are 492 fraud cases out of total 284,807 transactions. Which is only 0.173% of all transactions. The dataset is highly skewed, and it will be interesting too see if Naive Bayes perform well with this type of data.

In [7]:
```
df['Class'].value_counts()
```

Out[7]:
```
0    284315
1       492
Name: Class, dtype: int64
```

In [8]:
```
print( 'fraud transactions % =', (len(df[df['Class']==1]) / len(df[df['C
lass']==0]) )*100,"%")
```

```
fraud transactions % = 0.17304750013189596 %
```

We are skipping correlation analysis because by definition, PCA solves multicollinearity among predictor variables. Next, we are going to use Kernel Density Estimation (KDE) plot. KDE is a non-parametric way to estimate the probability density function of a random variable. To put it simply, it allows us to estimate what’s the share of data that falls into a particular interval.

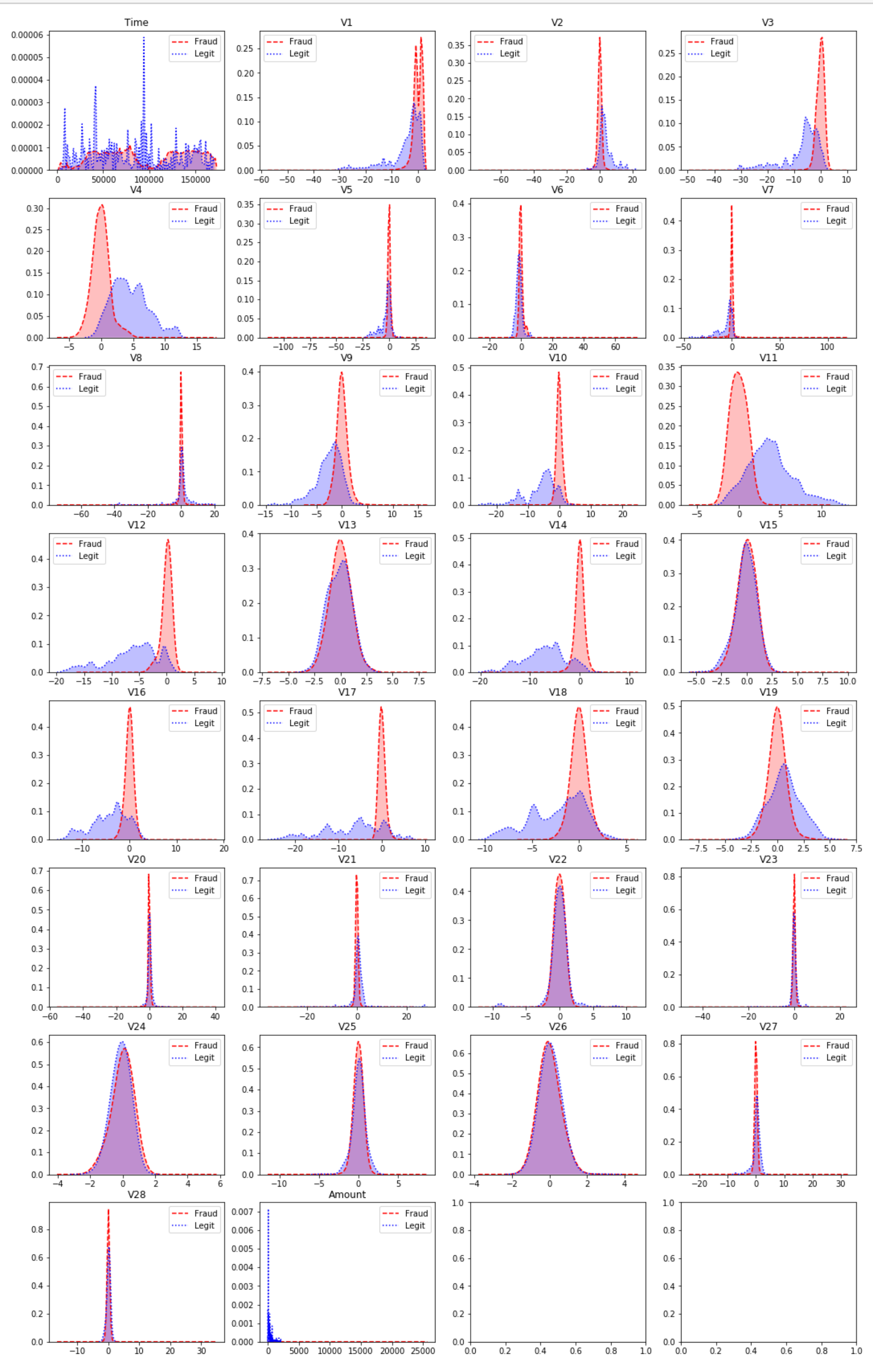
By looking at below KDE plots, we know that the distribution of Fraud / Legit datapoints for 12 variables - 'Time','V8','V13','V15','V20','V22','V23','V24','V25','V26','V27', 'V28' - are largely overlaps. It means that it might be difficult to classify Fraud and Legit based on these variables.

In [9]:
```
colnames = df.drop('Class',axis=1).columns.values
fraud = df[df.Class ==0]
legit = df[df.Class ==1]
```

In [11]:
```
i = 1
plt.subplots(8,4,figsize=(18,30))

for col in colnames:
    plt.subplot(8,4,i)
    sns.kdeplot(fraud[col], bw = 0.4, label = "Fraud", shade=True, color
="r", linestyle="--")
    sns.kdeplot(legit[col], bw = 0.4, label = "Legit", shade=True, color
= "b", linestyle=":")
    plt.title(col, fontsize=12)
    i = i + 1

plt.show()
```



4. Data Cleaning

First we are going to normalize our 'amount' variable, then drop 12 variables - 'Time','V8','V13','V15','V20','V22','V23','V24','V25','V26','V27', 'V28' - since the distribution of fraud/legit are highly overlaps and it might be difficult to classify Fraud and Legit based on these variables.

In [13]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
df['Amount_scaled'] = sc.fit_transform(df['Amount'].values.reshape(-1,1
))
df['Amount_scaled'].head(5)
```

Out[13]:
```
0    0.244964
1   -0.342475
2    1.160686
3    0.140534
4   -0.073403
Name: Amount_scaled, dtype: float64
```

In [17]:
```
df.drop(['Time','V8','V13','V15','V20','V22','V23','V24','V25','V26','V2
7','V28'], axis = 1, inplace = True)
```

5. Naive Bayes - Training

We are going to split our data into 80% training data and 20% testing data

In [21]:
```
X = df.drop(['Class'], axis = 1)
y = df['Class']
```

In [22]:
```
X.head()
```

Out[22]:

	V1	V2	V3	V4	V5	V6	V7	V9	V10
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.363787	0.090794
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	-0.255425	-0.166974
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	-1.514654	0.207643
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	-1.387024	-0.054952
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	0.817739	0.753074

In [23]:
```
y.head()
```

Out[23]:
```
0    0
1    0
2    0
3    0
4    0
Name: Class, dtype: int64
```

In [24]:
```
from sklearn.model_selection import train_test_split
```

In [40]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=123)
```

In [41]:
```
print("X_train:", X_train.shape)
print("y_train:",y_train.shape)
print("X_test:",X_test.shape)
print("y_test:",y_test.shape)
```

```
X_train: (227845, 19)
y_train: (227845,)
X_test: (56962, 19)
y_test: (56962,)
```

Next, we fit Gaussian Naive Bayes model from sklearn.naive_bayes because our predictors are continues variables. From the confusion matrix, we can see that the model misclassified 58 legit cases as fraud cases when using training dataset.

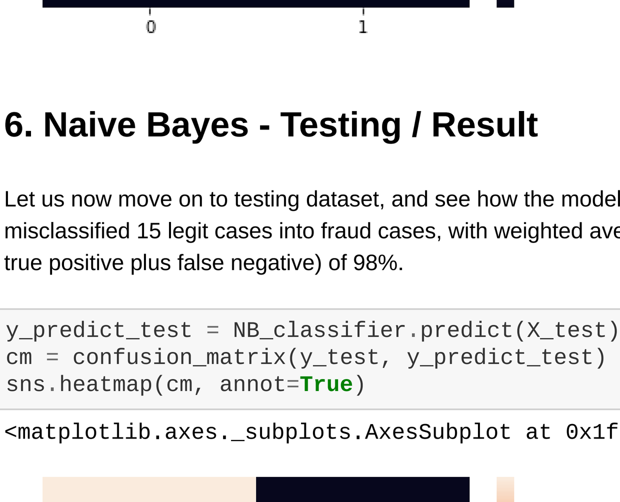
In [42]:
```
from sklearn.naive_bayes import GaussianNB
NB_classifier = GaussianNB()
NB_classifier.fit(X_train, y_train)
```

Out[42]:
```
GaussianNB(priors=None, var_smoothing=1e-09)
```

In [43]:
```
from sklearn.metrics import classification_report, confusion_matrix
```

In [44]:
```
y_predict_train = NB_classifier.predict(X_train)
y_predict_train
cm = confusion_matrix(y_train, y_predict_train)
sns.heatmap(cm, annot=True)
```

Out[44]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1f7db710b88>
```

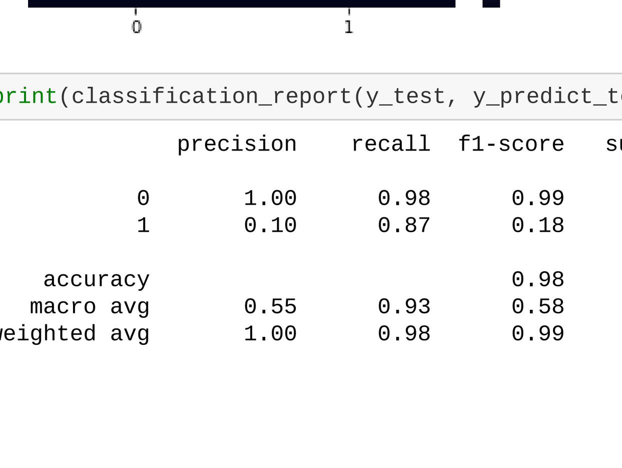


6. Naive Bayes - Testing / Result

Let us now move on to testing dataset, and see how the model performs with unseen data. It misclassified 15 legit cases into fraud cases, with weighted average recall score (true positive / true positive plus false negative) of 98%.

In [45]:
```
y_predict_test = NB_classifier.predict(X_test)
cm = confusion_matrix(y_test, y_predict_test)
sns.heatmap(cm, annot=True)
```

Out[45]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1f7dba34588>
```



In [46]:
```
print(classification_report(y_test, y_predict_test))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     56847
           1       0.10      0.87      0.18       115

    accuracy                           0.98     56962
   macro avg       0.55      0.93      0.58     56962
weighted avg       1.00      0.98      0.99     56962
```