Multivariate Gaussian for Fraud Detection

Part 2(cont. from last week)

Recap

- Our distribution follows Gaussian/Normal distribution, so MVG can be applied
- How does our normal and fraudulent data look?
- General algorithm for MVG
- Equation

Aim

- Application of algorithm in code (calculating cov_matrix and probability matrix)
- Training the algorithm and testing it
- Behavior of fraud data on the algorithm
- Evaluation of metrics (precision, recall and f1 score)
- Application of inbuilt python code of MVG

Equations

- Two important parts:
  - Covariance Matrix (represented as Σ in probability eq)

$$\text{Cov}(X,Y) = \frac{\sum_{i=1}^{n}(X_i-\bar{X})(Y_i-\bar{Y})}{n}$$

  - Probability (threshold over which MVG is classified)

$$p(x;\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right).$$

Implementation of covariance matrix

Covariance matrix is an nxn square matrix, where except the diagonal, all elements are **zero.**

The diagonal element represents how much that element is correlated to the other elements.

Output is Sigma(the nxn matrix)

In [12]:
```
def covariance_matrix(X):
    m=len(X)
    mu = X.mean()
    Sigma=0
    for i in range(m):
        Sigma += np.outer(X[i] - mu, X[i] - mu)
    return Sigma / m
```

In [13]:
```
cov_mat = covariance_matrix(X_train)
cov_mat_inv = np.linalg.pinv(cov_mat)
cov_mat_det = np.linalg.det(cov_mat)
np.matrix(cov_mat).shape
```

Out[13]: (28, 28)

Result of covariance matrix

```
print (cov_mat)
```

```
[[  3.75981019e+00   1.20350036e-01  -1.15915582e-01   5.22064627e-02
   -6.04683980e-02  -1.23058867e-02  -1.18869626e-01  -5.41417174e-03
   -1.90189891e-02  -7.70002362e-02   3.24216943e-02  -7.88415048e-02
   -3.30620404e-03  -6.98364320e-02  -9.18628576e-03  -6.12225727e-02
   -1.02448641e-01  -4.83679623e-02   2.64333175e-02  -1.93710693e-02
    3.43907715e-03   5.25675937e-04   1.47048608e-02  -6.51392317e-03
    9.59823541e-03  -3.24473171e-04  -1.41704007e-02   3.76333053e-02]
 [  1.20350036e-01   2.72737080e+00   1.10432146e-01  -4.85098340e-02
    5.58800651e-02   4.14749933e-03   5.86204251e-02   8.42376856e-03
    3.68817411e-02   6.80997098e-02  -3.06421491e-02   4.86667799e-02
   -3.85129065e-03   3.74127914e-02  -5.04201207e-03   4.31366748e-02
    6.27912040e-02   2.97448464e-02   2.99325670e-03  -5.28727980e-02
   -3.13158179e-02   1.30740683e-02   1.97618043e-02  -2.89006026e-04
    7.64825546e-03   5.75934650e-03  -8.51310368e-03   3.56751456e-02]
```

These are first 2 rows.
There are 28 such rows(since n=28)
And 28 columns.

Other than the diagonal elements, the others are negligible.

Diagonal elements:

Useful elements from Sigma are just the diagonal elements. The rest of the elements ideally are 0, here they are some e-02 values due to minute internal calculation

```
array([ 3.75981019,  2.7273708 ,  2.14381036,  1.97349562,  1.83304134,
        1.75823126,  1.39794885,  1.34655203,  1.19072044,  1.08820473,
        1.00809036,  0.89938214,  0.98865687,  0.81074572,  0.83841067,
        0.72292018,  0.56915296,  0.68468254,  0.66818608,  0.60365491,
        0.50527429,  0.52228353,  0.37419929,  0.37031409,  0.26969463,
        0.23442655,  0.15651947,  0.10436598])
```

Implementation of Probability

```
def multi_gauss(x):
    n = len(cov_mat)
    return (np.exp(-0.5 * np.dot(x, np.dot(cov_mat_inv, x.T)))
            / (2. * np.pi)**(n/2.)
            / np.sqrt(cov_mat_det))
```

Result of probability on Fraud vs. Non-Fraud

```
# Check out some resulting probablilities
 for Fraud examples
#X_Fraud = data.query("Class==1").
drop(["Amount","Class","Time"],1)
for i in range(10):
    print(multi_gauss(X_Fraud.iloc[i]))
```

```
1.12978540699e-93
3.15886104559e-19
8.87924044514e-100
0.0
5.76259488396e-60
3.63225880102e-287
9.69708784719e-287
5.43682791792e-174
4.33979036493e-173
1.95110214052e-261
```

```
# Check out some resulting probablilities
 for genuine examples
#X_Legit = data.query("Class==0").
drop(["Amount","Class","Time"],1)
for i in range(10):
    print(multi_gauss(X_Legit.iloc[i]))
```

```
2.15198754521e-12
1.1814392916e-11
8.30583484946e-19
9.58512332595e-15
7.66965019308e-13
1.69292093954e-11
3.84029460532e-12
8.67823100024e-23
2.71842595314e-14
4.40835128148e-12
```

Testing for hard coded values

```
# Picking out 100 training examples to test how many are misclassified as false
 positive
ptrain_result = np.apply_along_axis(multi_gauss, 1, X_train.head(100))
```

For threshold, eps=1e-13, we have 49 results falsely classified as +ve.

```
In [21]: eps = 1e-13
         sum(ptrain_result < eps)

Out[21]: 49
```

As threshold increases, ie, as eps increases, more transactions are classified as +ve.

```
In [22]: eps = 2e-13
         sum(ptrain_result < eps)

Out[22]: 56
```

Testing for calculated values

In [27]:
```
eps = max([multi_gauss(x) for x in fraud_pca_data.values])
print(eps)
print(sum(ptrain_result < eps))
```

```
3.54226914278e-12
87
```

In [28]:
```
eps = min([multi_gauss(x) for x in fraud_pca_data.values])
print(eps)
print(sum(ptrain_result < eps))
```

```
0.0
0
```

max([multi_gauss(x) for x in fraud_pca_data.values])

For each value in fraud_pca_data, calculating the prob for multi_gauss, and taking max and min to see how it changes

Metrics

```
recall, prec, F1 = stats(X_valid, y_valid, eps)
print("For a boundary of:", eps)
print("Recall:", recall)
print("Precision:", prec)
print("F1-score:", F1)
```

```
For a boundary of: 2.7832564593e-12
Recall: 1.0
Precision: 0.00386901953383
F1-score: 0.00770821583004
```

Precision = TP/TP+FP
High precision relates to the low false positive rate.

Recall(Sensitivity) = TP/TP+FN

F1 Score = 2*(Recall * Precision) / (Recall + Precision)

As we had seen in the prev slide, our false +ve was high, hence precision is low. This is because our Fraud and Non-Fraud data is very similar.

Inbuilt function

```
from scipy.stats import multivariate_normal
var = multivariate_normal.pdf(X_test_2,X_test_2.mean(), cov_mat)
```

Inbuilt function: scipy.stats.multivariate_normal()

Parameters:
x : array_like quantiles, with the last axis of x denoting the components.
mean : array_like, Mean of the distribution (default zero)
cov : array_like, Covariance matrix of the distribution (default one)

Result of Inbuilt function

```
eps=var[1]
recall, prec, F1 = stats(X_valid, y_valid, eps)
print("For a boundary of:", eps)
print("Recall:", recall)
print("Precision:", prec)
print("F1-score:", F1)
```

```
For a boundary of: 6.29871123545e-12
Recall: 1.0
Precision: 0.00359943813649
F1-score: 0.00717305729698
```

Conclusion

High recall means (extremely) low precision

- this might be ok if on spot security measures are cheaply implemented.
  For example an extra verification online for cases that seem suspicious.
- problematic if the flagged cases have to be reviewed by hand.

Need to include Time into data

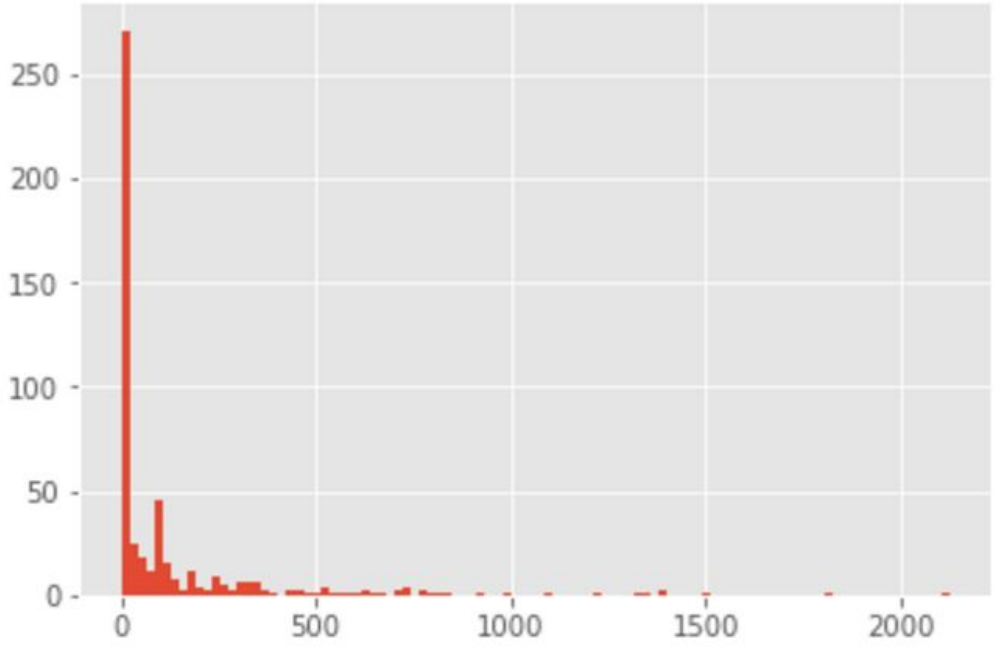
Need to analyze Amount.

The two last cases are problematic within the Multivariate Gaussian approach as

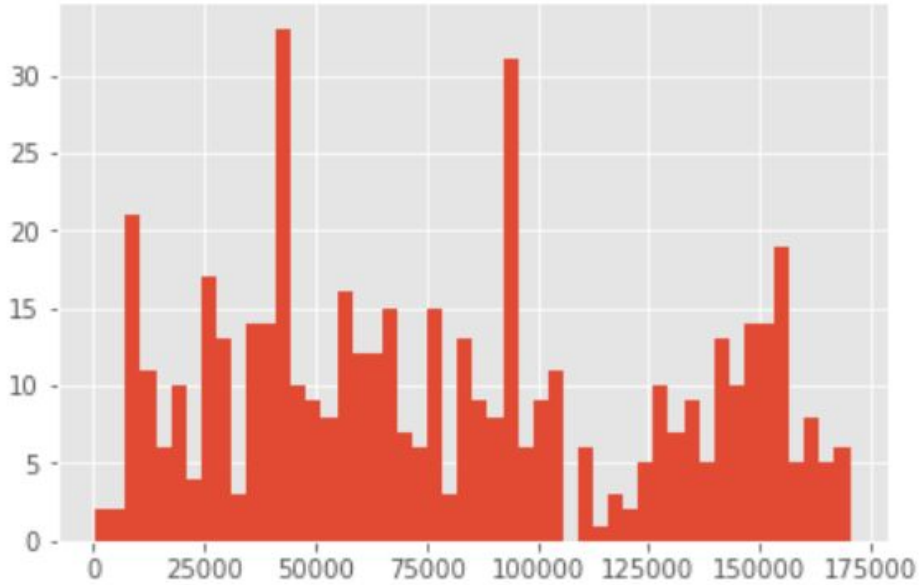
the provided data are not normal distributed.

Fraudulent Time and Amount

```
fraud_data["Amount"].hist(bins=100);
```



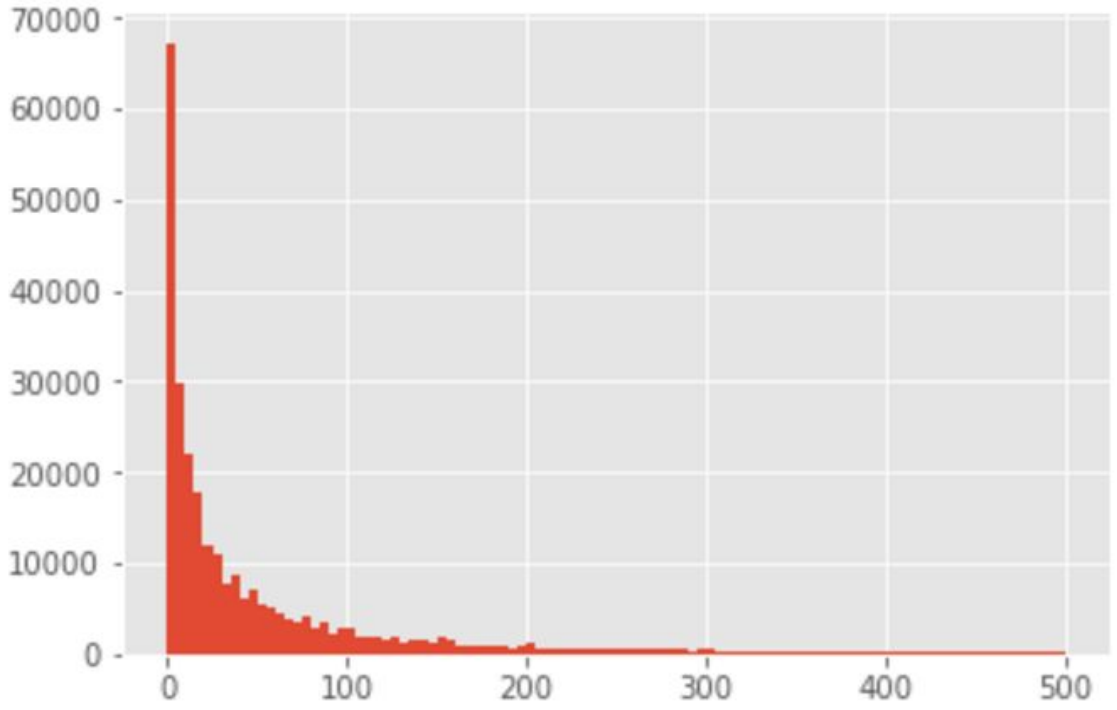

```
fraud_data["Time"].hist(bins=50);
```



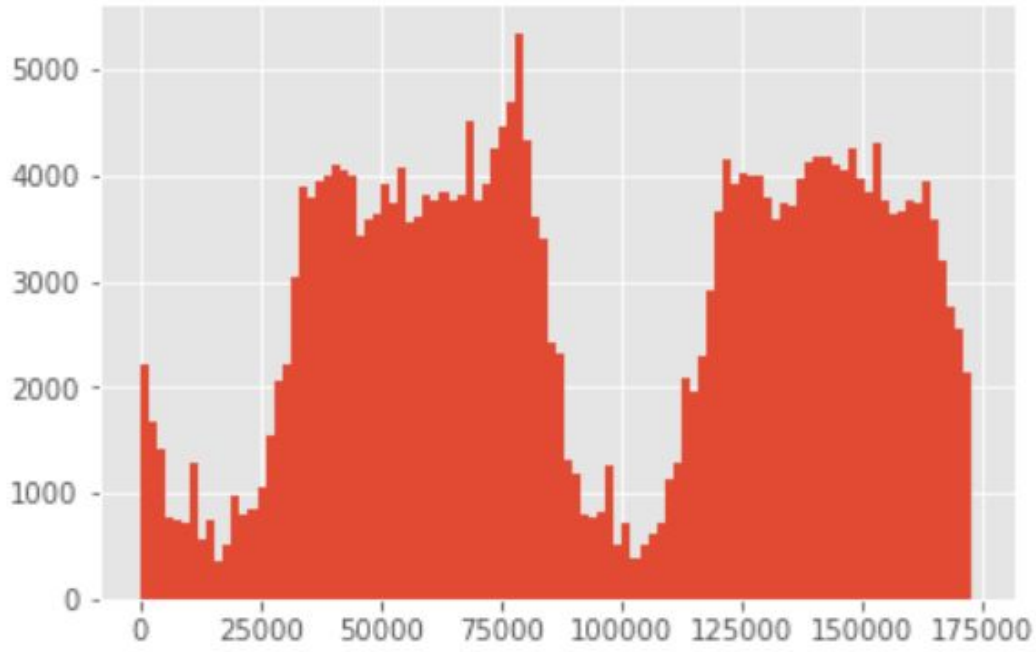
MVG cannot be used for fraudulent Amount and Time as they are not normal distribution.

Non Fraudulent Time and Amount

```
normal_data["Amount"].hist(bins=100);
```




```
normal_data["Time"].hist(bins=100);
```



MVG cannot be used for genuine Amount and Time either as they are not normal distribution.