Assignment 5: A Beginner's Guide To Fraudulent Transactions

Ash

2018/12/06

1 Loading & Cleaning The Data

First, we have to clean and organize the data into a more suitable format for modeling. Dates of transaction in the original data are being changed in to day, month and year by slicing the strings, then the month and year are encoded into one column since we are interested in the number of transaction in a month. Of course, we can also collapse months in different year into only 12 datapoints, but that will result in a smaller dataset for density modeling. Days are maintained, and will be used to model the probability a transaction will happen in a day.

```
#Loading relevant libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors.kde import KernelDensity
import pandas as pd
pd.options.display.float_format = '{:,.2f}'.format

#Loading the data
data = pd.read_csv('/Users/ash/Downloads/anonymized.csv')

#Clean the date in a more processable format
def clean_date(obj):
    obj = str(obj)
    date = obj[:2]
    year = obj[-4:]
    month = obj[2:-4]
    if month == 'Jan':
        month = 1
    elif month == 'Feb':
        month = 2
    elif month == 'Mar':
        month = 3
    elif month == 'Apr':
        month = 4
    elif month == 'May':
```

```
        month = 5
    elif month == 'Jun':
        month = 6
    elif month == 'Jul':
        month = 7
    elif month == 'Aug':
        month = 8
    elif month == 'Sep':
        month = 9
    elif month == 'Oct':
        month = 10
    elif month == 'Nov':
        month = 11
    elif month == 'Dec':
        month = 12
    return date+'/'+str(month)+'_'+year

data['Date'] = data['Date'].apply(clean_date)
dumb = pd.DataFrame(data['Date'].str.split('/',2).tolist())
data = pd.concat([data, dumb], axis=1)

data.columns = ['Date', 'Amount', 'Day', 'Month-Year']
data = data.drop(columns=['Date'])
data.head()

--->Output:

Figure 1
```

	Amount	Day	Month-Year
0	54,241.35	25	5_2016
1	54,008.83	29	5_2017
2	54,008.82	30	6_2017
3	52,704.37	05	1_2017
4	52,704.36	23	2_2017

Figure 1: Cleaned Data

2 Density Models

For the following density models Gaussian mixtures are chosen because they exhibit both smoothing qualities controlled by the bandwidth and nice interpretation of the bandwidth as the variance of the Gaussian, and how this variance controls how much effect each datapoint has for its neighboring datapoints. Bandwidth in all model are set

to 3, since using visual inspection lower bandwidth seems to have less smoothing effect for this specific dataset (there are more spikes, which probably means the density estimation is estimating more of a local fluctuation rather than more global trend) and higher bandwidth seems to smooth the data too much (e.g., smoothing multimodal data into unimodal density, which is undesirable). An extension might be to use cross validation for the density model by first estimating the density for a part of the data then sample from that density to compare with the held-out part of the data sufficient statistics (like position of the means or variance in multimodal situation), thereby tweaking the bandwidth to a more optimal value. Furthermore, experimenting with different density function that might better express periodicity (given this is a time-dependent financial data) in the data.

2.1 Number Of Transaction In A Month

Sampling from this density distribution will return the likely number of transaction for an average month, given the data we have seen. We do not model the time-dependencies of transaction (for example, we treat the number of transaction in January of 2014 the same as January of 2016, but in reality this assumption might not hold given the historical transactions likely relate to time).

```
#Density model of the number of transaction in a month

#Count the number of transaction by pandas
hist_data = np.array(data['Month-Year'].value_counts())

#Model the counts by Gaussians
kde_1 = KernelDensity(kernel='gaussian', bandwidth=3).fit(hist_data.reshape
    (-1,1))

#Create density model
x = np.linspace(min(hist_data),max(hist_data), 500).reshape(-1, 1)
y = np.exp(kde_1.score_samples(x))

plt.figure(figsize=(12,8))
plt.hist(hist_data, density=True, bins=15, label='Data', alpha=0.7)
plt.plot(x, y, label='Density Function')
plt.legend()
plt.xlabel('Number of transaction')
plt.title('Kernel Density Modeling Of Number Of Transaction In A Month')
plt.show()

--->Output:

Figure 2
```

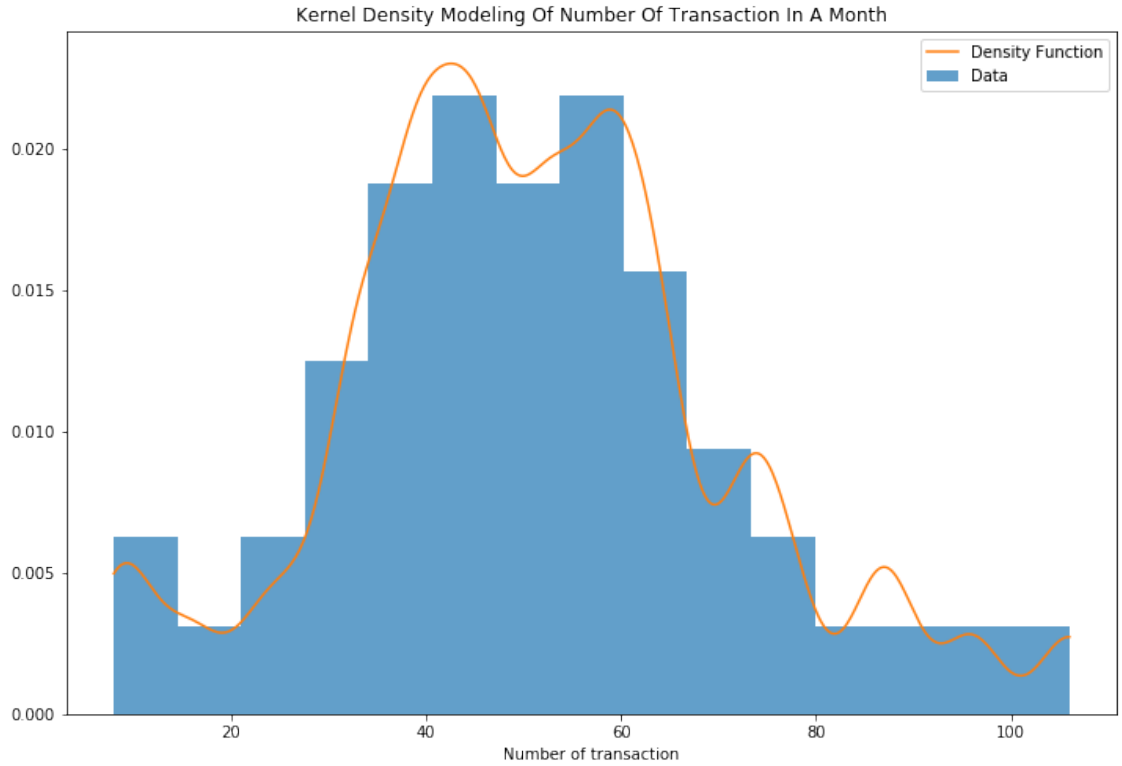


Figure 2: Number Of Transaction In A Month

2.2 Day Of Transaction In A Month

After counting the number of transaction in a specific day (from 1 to 31, note that certain months have fewer than 31 days, so this would represent the average of all months, rather than of a specific month), we model the density of these counts for each day. Sampling from this density distribution will give a (float) day, so roughly this model expresses how likely a transaction will occur in a specific day: a typical transaction, as seen in figure 3, will likely occur at the beginning or the end of the month since the density seems to be bimodal.

```
#Density model of the number of transaction in a day

hist_data = np.array(data['Day'], dtype=float)

kde_2 = KernelDensity(kernel='gaussian', bandwidth=3).fit(hist_data.reshape
    (-1,1))
x = np.linspace(min(hist_data),max(hist_data), 500).reshape(-1, 1)
y = np.exp(kde_2.score_samples(x))

plt.figure(figsize=(12,8))
plt.hist(hist_data, density=True, bins=15, label='Data', alpha=0.7)
```

```
plt.plot(x, y, label='Density Function')
plt.legend()
plt.xlabel('Day')
plt.title('Kernel Density Modeling Of Day Of The Month A Transaction Will
    Occur')
plt.show()

---->Output:

Figure 3
```

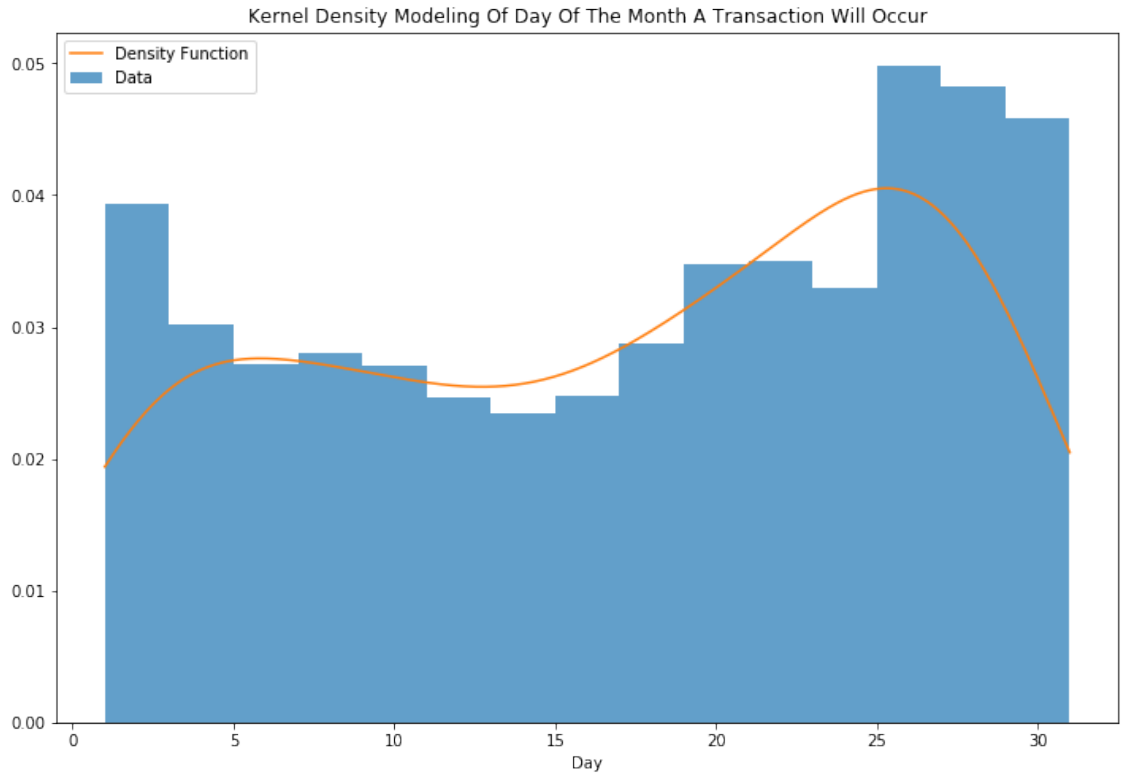


Figure 3: Day Of The Month A Transaction Will Occur

2.3 Transaction Size

Transaction size (the amount of money incoming or outgoing) is included in this model, regardless of whether it's coming out or going in (we'll model the outgoing or incoming transactions separately later). Sampling from this model will return the likely size of the transaction for any particular month and day, which again might not be realistic given the time-dependencies of transaction size (for example, toward the end of November/beginning of December the size of outgoing transaction might become larger because people go shopping for Black Friday or for Christmas gifts, compared to the

usual situation where people go grocery weekly).

```
#Density model of the transaction size

hist_data = np.array(np.log(np.absolute(data['Amount'])))

kde_3 = KernelDensity(kernel='gaussian', bandwidth=3).fit(hist_data.reshape
    (-1,1))
x = np.linspace(min(hist_data),max(hist_data), 500).reshape(-1, 1)
y = np.exp(kde_3.score_samples(x))

plt.figure(figsize=(12,8))
plt.hist(hist_data, density=True, bins=15, label='Data', alpha=0.7)
plt.plot(x, y, label='Density Function')
plt.legend()
plt.xlabel('Log of transaction size')
plt.title('Kernel Density Modeling Of Log Of Transaction Size')
plt.show()

---->Output:

Figure 4
```

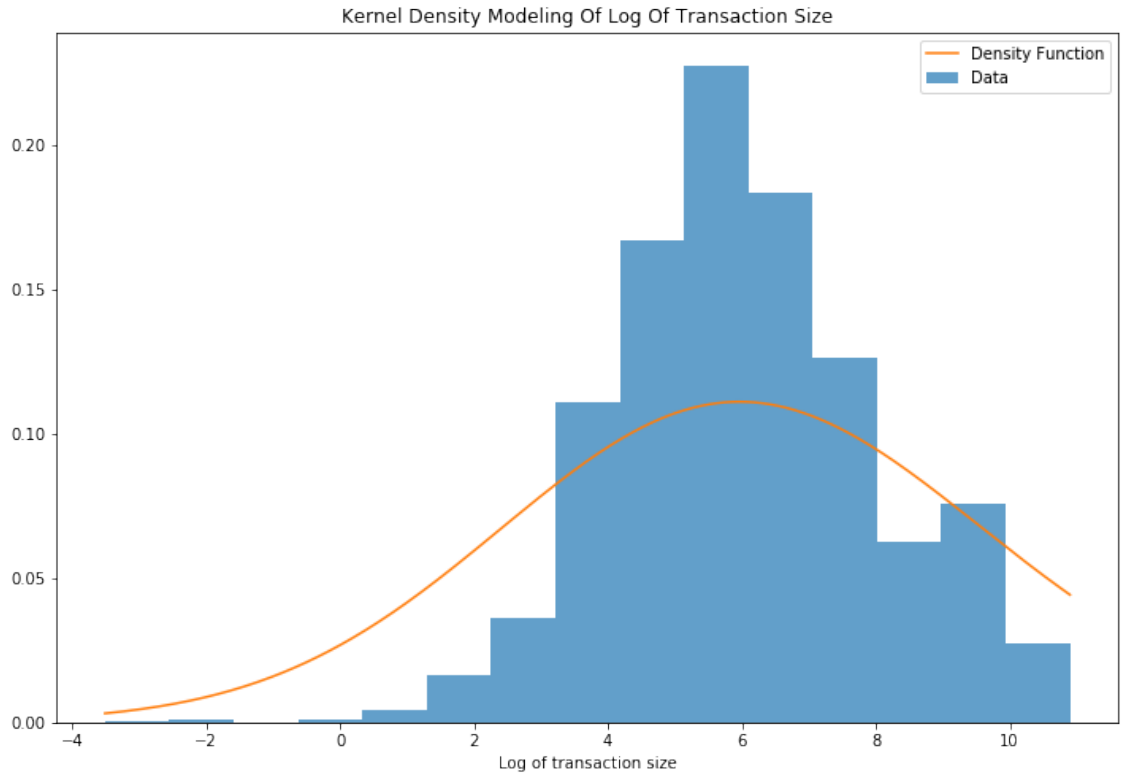


Figure 4: Log Of Transaction Size

3 Sampling Fake Transactions

3.1 In-Out Modeling

To create a convincing set of monthly transaction, first we need a model to decide whether a transaction is incoming or outgoing. Of course, it's possible to model the number of incoming and outgoing transaction for a day in a month using density modeling, but that would create a slight problem of when sampling for the total number of transaction in that month with the first model above, we might have mismatch and therefore the total number of transaction differs from the sum of the incoming and outgoing transaction. Thus, it makes sense to model this in/out decision as a proportion, or a probability of a transaction, in a day in a month, is in or out. We use a beta distribution as the prior for the probability of being an incoming transaction, and a binomial distribution as the likelihood of observing certain number of incoming transactions per month for two reasons. First, assuming that for a specific day, the probability of sampling an incoming transaction is independent of the probability of sampling another incoming transaction later in the same day. This, again, might not hold in reality for situations like recurrent payments to services/salaries (the outgoing transaction will occur anyway, regardless of the incoming transaction and the model, with perfect certainty at a day in a month) or holidays like mentioned above (for example there are days when there is very few incoming transaction because people go out and buy a lot of things, like on Black Fridays). However, disregard special cases like thoses (because the density models built did not capture those as well), we can justify this assumption by arguing that for a particular day, the in/out quality of the current transaction now does not depends on the next transaction. Second, the beta distribution is a conjugate prior for the binomial likelihood, so it will simplify calculations. We also use an uninformative prior: Beta(alpha=1, beta=1) to make the initial guess uniform for all probability.

```
#Modeling in/out transaction probability for a day

#Count the number of in/out transaction
in_out = data['Amount'] > 0

data['In_Out'] = in_out

_in = data.loc[data['In_Out'] == True]['Day'].value_counts()
_out = data.loc[data['In_Out'] == False]['Day'].value_counts()

from scipy.stats import beta

#Update a beta prior according to the conjugate property
a_prior = 1
b_prior = 1
a_post = a_prior + np.sum(_in)
b_post = b_prior + np.sum(_out)

prior = beta(a_prior, b_prior)
posterior = beta(a_post, b_post)
```

```
21
22 #Plot the prior and posterior
23 x_io = np.linspace(0,1,100)
24 y_io_1 = prior.pdf(x_io)
25 y_io_2 = posterior.pdf(x_io)
26
27 plt.figure(figsize=(12,8))
28 plt.plot(x_io, y_io_1, label='Prior')
29 plt.plot(x_io, y_io_2, label='Posterior')
30 plt.title('Posterior & Prior Of Probability Of In/Out Transaction')
31 plt.show()
32
33 ---->Output:
34
35 Figure 5
```

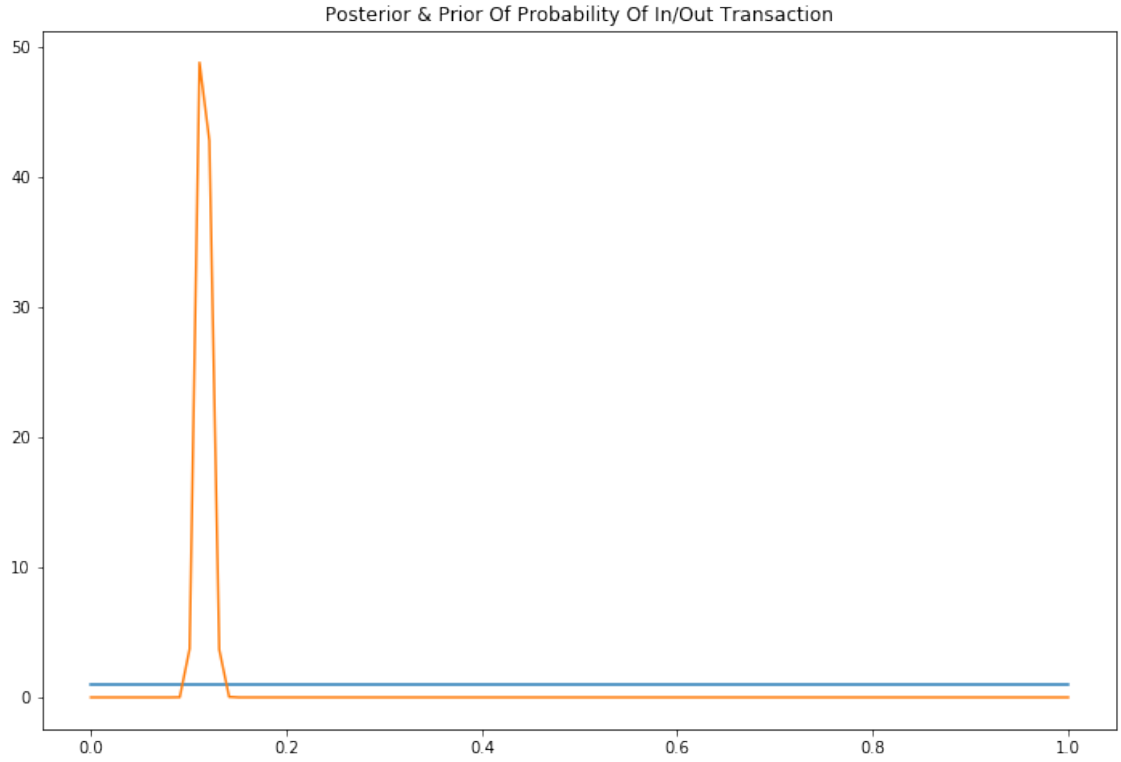


Figure 5: Posterior & Prior For The Probability Of Transaction Being Incoming

We can see the posterior peaks at around 0.12, and now we can create a fake set of monthly transaction data:

```
1 #Creating fake transaction data for a month
2
3 def create_fake(month, date, size, in_out):
4     #Sample the total number of transaction for the month
5     n_trans = month.sample(1)
```

```
6
7     #For each transaction, sample a date and a size for the transaction
8     date_trans, size_trans = [], []
9     for _ in range(int(n_trans)):
10        _date = int(date.sample(1))
11        if _date < 1:
12            _date = 1
13        elif _date > 31:
14            _date = 31
15        date_trans.append(_date)
16        size_trans.append(np.exp(size.sample(1))[0][0])
17
18    #For the month, sample a proportion of that month transaction will be
      incoming
19    p = in_out.rvs(size=1)
20    for i in range(int(p*len(size_trans))):
21        size_trans[i] = -size_trans[i]
22
23    #Creating fake data
24    d = {
25        'Amount': size_trans,
26        'Day': date_trans
27    }
28    fake_data = pd.DataFrame(data=d)
29
30    return fake_data.sample(frac=1).reset_index(drop=True)
31
32 fake = create_fake(kde_1, kde_2, kde_3, posterior)
33 fake.head()
34
35
36 ---->Output:
37
38 Figure 6
```

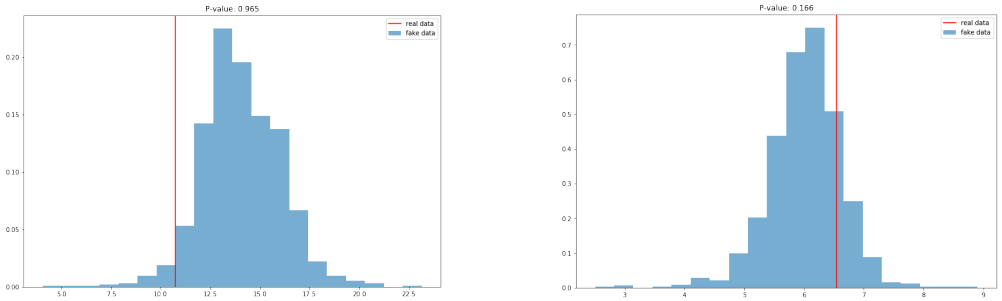
3.2 Detect Fraudulent Transaction

Since we were making a lot of unrealistic assumptions when building the model to generate a fake set of monthly transaction, there are some ways for a forensic account to detect fraudulent transactions. To begin with, the accountant can simply check the numerical form of the transaction size: usually for transactions between realistic parties, transaction size is probably rounded up to .00 or .50 (to 10 cents), because normally people would not send each other $10.67. So the accountant can simply count the "odd" transactions in the fake set to compare it to a normal set of transaction to see that the fake set was generated by a simple continuous variable model. Second, the accountant can look at recurrent transaction or periodicity in a real set of transaction (which can be salary, debt payments, etc.) which won't be presented in the fake data since we did not model time-dependency in any way.

Of course, we can improve our model by modifying our assumptions and changing our

	Amount	Day
0	5,180.27	19
1	281.01	9
2	604.96	10
3	2.83	24
4	43.17	14

Figure 6: Fake Transaction Data



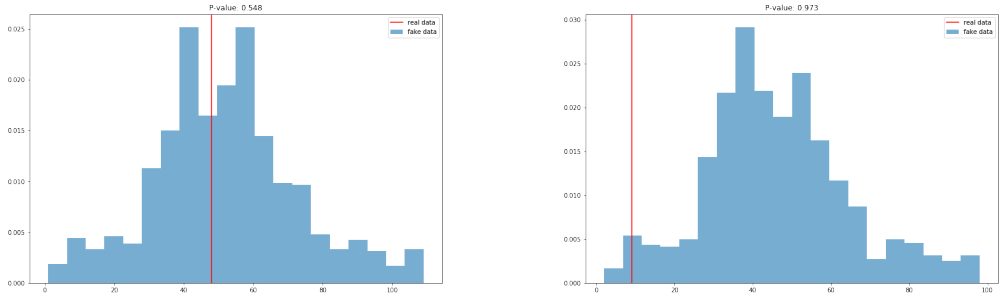
(a) Max (b) Mean

Figure 7: Test Statistics For 1000 Sampled Set Of Fake Data, Compared To The Real Data

models accordingly. One way to do this is to perform predictive tests, since we have access to the fake data generating mechanism. To do a predictive test, first we create a large number of fake data, then for each fake dataset we calculate some sort of test statistics and compare it to the real data test statistics to see how likely the calculated test statistics on the real dataset likely stem from the same data generating mechanism. For example, in figure 7 we create 1000 sets of fake monthly transactions and for each set we calculate both the mean and the max transaction size to compare to the mean and max log of transaction size of a month in the real data. We can see that for the max (7a) around 96% of the sampled fake data maxes are greater than the real data max, suggesting that we are using an assumption that makes this aspect of the fake data different from the real data (in a statistically significant way, since among 1000 sampled test statistics, 960 of them have value different significantly from the real value), whereas for the mean (7b) the p-value is around 16%, suggesting that we likely make

assumptions that led to a data generating mechanism that modeled the mean of the real data correctly, since the real mean is closer to the most sampled mean. This might be because in our density model we did not account for the large variance observed in the real data. Another interesting deficit of the current model can be seen when we look at the total number of transaction in a month (figure 8a) and the total number of incoming transaction in a month (figure 8b). While in 8a the total number of transaction sampled is very close to the real value (p-value is around 0.5, suggesting that it's very likely that the real data follow the same generating mechanism as our model, for this particular test statistics), in 8b the number of incoming transaction in the sampled sets differs significantly from the real value (p-value around 0.97) suggesting that the naive assumption of the beta model for the proportion of incoming transaction is seriously wrong.

```
#Test statistics

def test_stats_1(data):
    return np.max(np.log(np.abs(data['Amount'])))

_test_stats = []
for _ in range(1000):
    _fake = create_fake(kde_1, kde_2, kde_3, posterior)
    _test_stats.append(test_stats_1(_fake))

dumb = [_test_stats[i] > test_stats_1(month_data) for i in range(len(
    _test_stats))]

plt.figure(figsize=(12,8))
plt.hist(_test_stats, density=True, bins=20, alpha=0.6, label='fake data')
plt.axvline(test_stats_1(month_data), color='r', label='real data')
plt.legend()
plt.title('P-value: {}'.format(sum(dumb)/len(dumb)))
plt.show()

def test_stats_2(data):
    return np.mean(np.log(np.abs(data['Amount'])))

_test_stats = []
for _ in range(1000):
    _fake = create_fake(kde_1, kde_2, kde_3, posterior)
    _test_stats.append(test_stats_2(_fake))

dumb = [_test_stats[i] > test_stats_2(month_data) for i in range(len(
    _test_stats))]
plt.figure(figsize=(12,8))
plt.hist(_test_stats, density=True, bins=20, alpha=0.6, label='fake data')
plt.axvline(test_stats_2(month_data), color='r', label='real data')
plt.legend()
plt.title('P-value: {}'.format(sum(dumb)/len(dumb)))
plt.show()

--->Output:

```



(a) Total No Of Transaction (b) Incoming No Of Transaction

Figure 8: Test Statistics For 1000 Sampled Set Of Fake Data, Compared To The Real Data

```
38 Figure 7
```

4 Benford's Law

Interestingly, the real data follows Benford's law whereas the fake data follows it only loosely (figure 9).

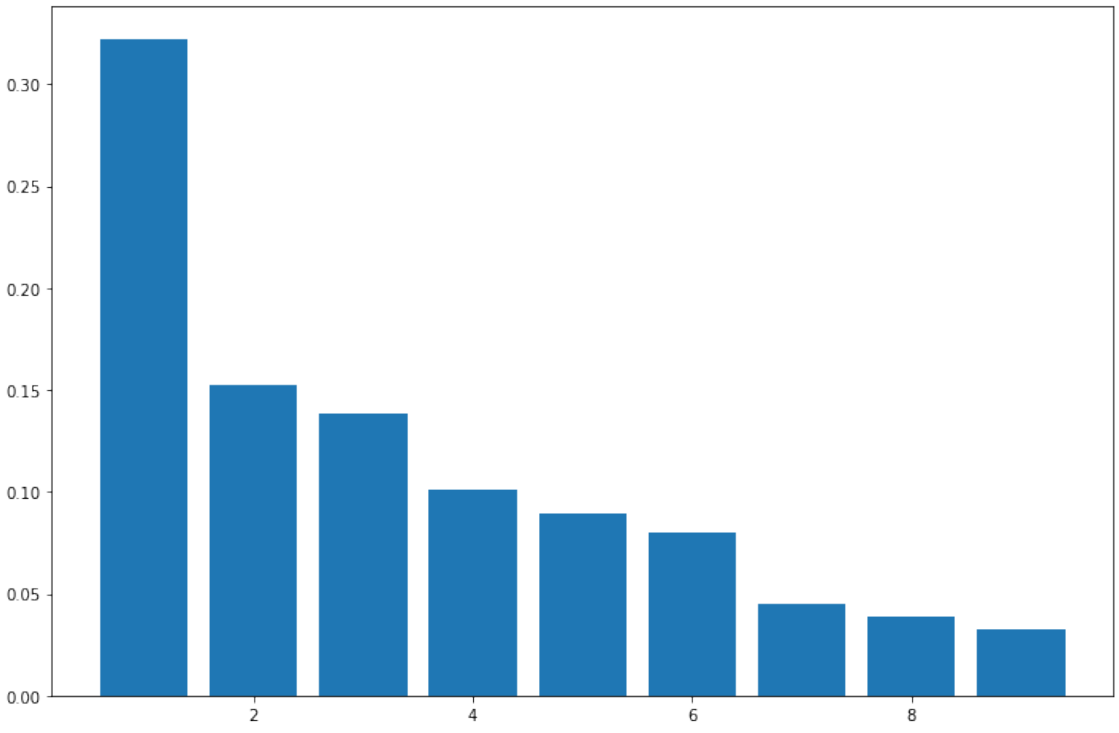


Figure 9: Benford's Law For Real Data

```
#Benford's Law
import math
def first_digit(num):
    num = np.absolute(num)
    if num < 1:
        return np.nan
    else:
        num = str(num)
        return int(num[0])
digits = data['Amount'].apply(first_digit)

plt.figure(figsize=(12,8))
plt.bar(range(1,10),digits.value_counts()/sum(digits.value_counts()))
plt.show()

--->Output:

Figure 9
```