data pre-processing for k-mean clustering

September 23, 2019

1 *Data pre-processing for k-mean clustering*

[83]:
```
import pandas as pd
from datetime import timedelta
import os
import seaborn as sns
import numpy as np
from matplotlib import pyplot as plt
print(os.getcwd())
```

```
/home/hans/python_codes/DataCamp/data_pre-processing_fo_k-mean_clustering
```

1.1 Advantiges of k-mean clustering

- one of the most popular unsupervised learning method
- Simple and fast
- Worsk well (with certain assumptions about the data)

1.2 Key k-mean assumptions

- Symmetric distribution of variables (not skewed)
- Variables with same average values
- Variables with same variance

1.3 Variables on the same scale

- K-means assumes equal mean
- And equal variance
- It's not the case with RFM Data

Dataset

[84]:
```
datamart_rfm = pd.read_excel('../data/datamart.xlsx')
```

[85]:
```
datamart_rfm.describe()
```

[85]:

```
         CustomerID      Recency  MonetaryValue    Frequency            R  \
count   4372.000000  4372.000000    4372.000000  4372.000000  4372.000000
mean   15299.677722    92.047118    1898.459701    93.053294     2.514181
std     1722.390705   100.765435    8219.345141   232.471608     1.124804
min    12346.000000     1.000000   -4287.630000     1.000000     1.000000
25%    13812.750000    17.000000     293.362500    17.000000     2.000000
50%    15300.500000    50.000000     648.075000    42.000000     3.000000
75%    16778.250000   143.000000    1611.725000   102.000000     4.000000
max    18287.000000   374.000000  279489.020000  7983.000000     4.000000

                 F            M  RFM_Segment    RFM_Score
count  4372.000000  4372.000000  4372.000000  4372.000000
mean      2.487420     2.500000   278.792315     7.501601
std       1.119114     1.118162   118.763354     2.828144
min       1.000000     1.000000   111.000000     3.000000
25%       1.000000     1.750000   211.000000     5.000000
50%       2.000000     2.500000   311.000000     7.000000
75%       3.000000     3.250000   411.000000    10.000000
max       4.000000     4.000000   444.000000    12.000000
```

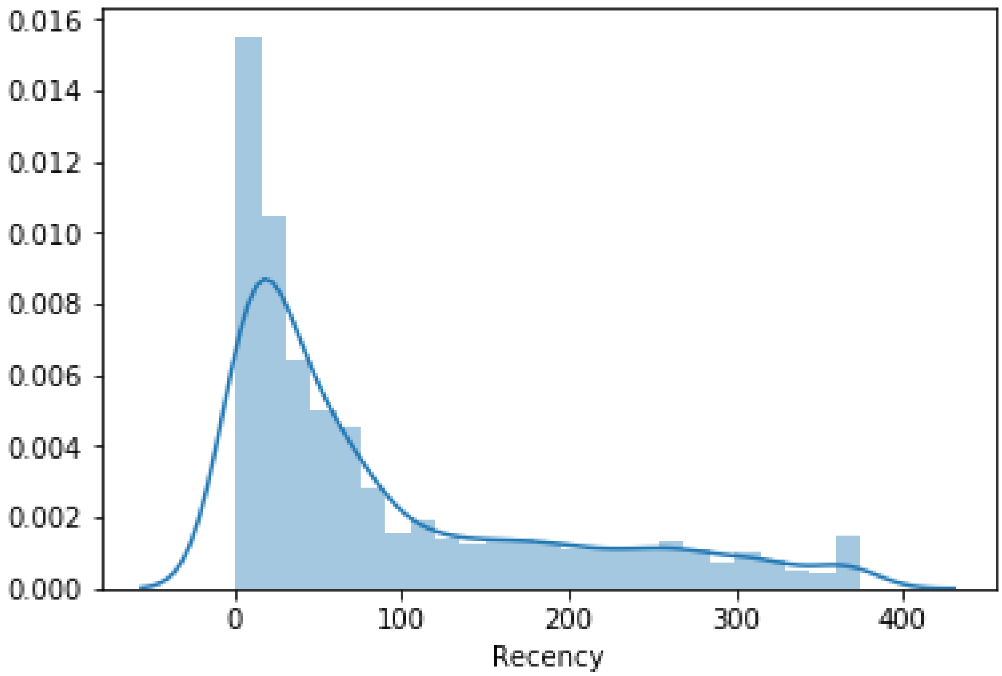
1.4 Identifying skewness

- Visual analysis of the distribution
- If it has a tail - it's skewed

1.4.1 Exploring distribution of Rencency

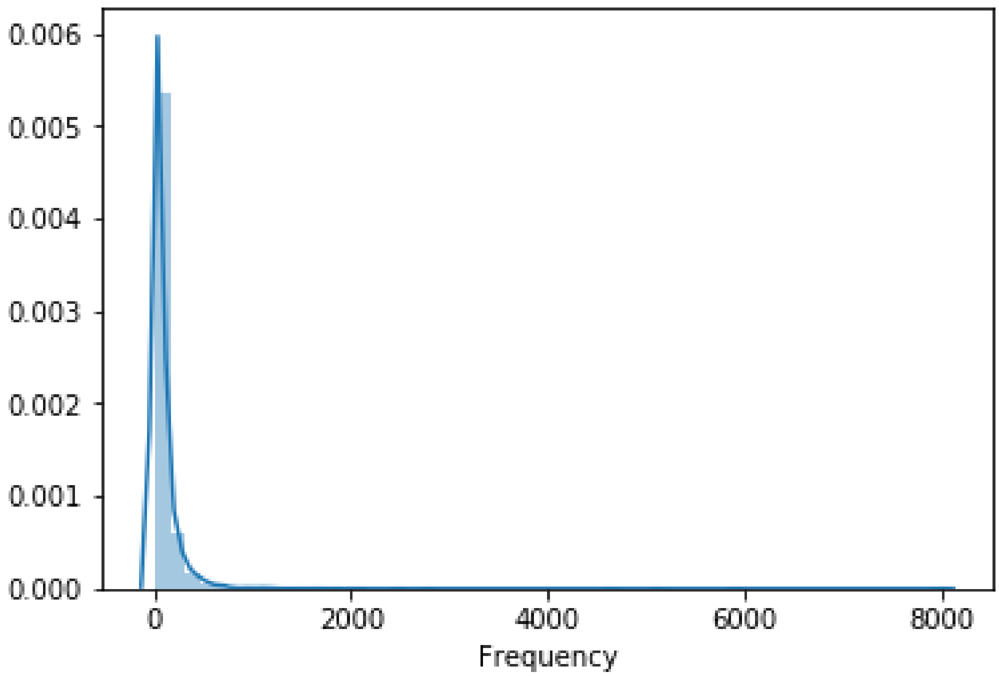
[86]:

```
sns.distplot(datamart_rfm['Recency'])
plt.show()
```



1.4.2 Exploring distribution of Frequency

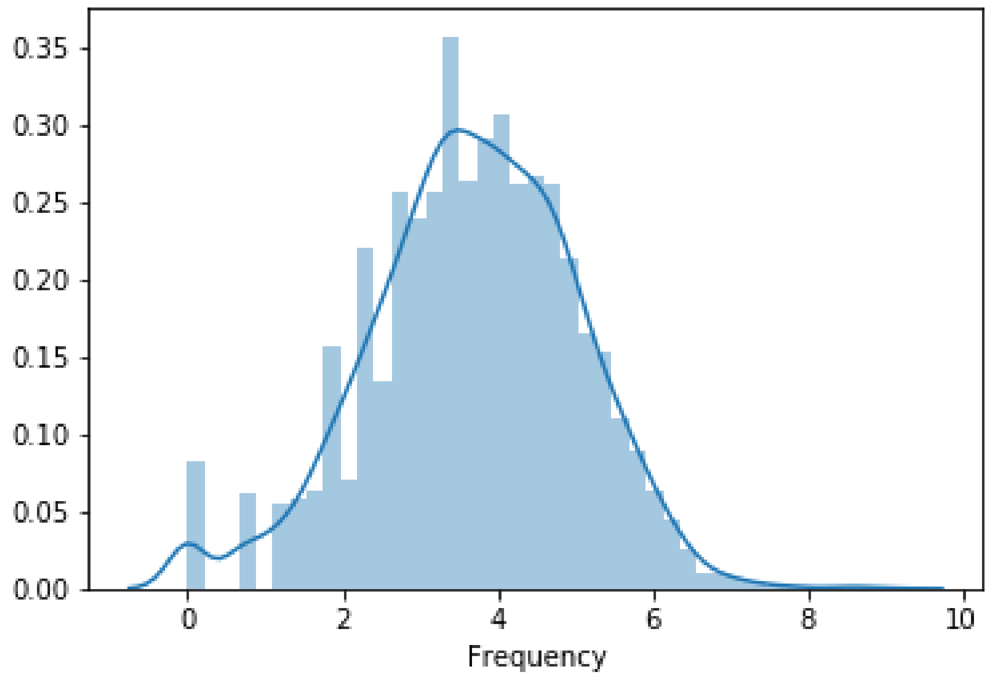
[87]:
```
sns.distplot(datamart_rfm['Frequency'])
plt.show()
```



1.5 Data transfomations to manage skewness

- Logarithmic transformations (positive values only)

[88]:
```
frequency_log = np.log(datamart_rfm['Frequency'])
sns.distplot(frequency_log)
plt.show()
```



1.6 Dealing with negative values

- Adding a constant before log transformation
- Cube root transformation

1.7 Centering and scaling variables

1.7.1 Identifying an issue

- Analyze key statistics of the dataset
- Compare mean and standard deviation

[89]:
```
datamart_rfm.describe().round(1)
```

[89]:

	CustomerID	Recency	MonetaryValue	Frequency	R	F	M	\
count	4372.0	4372.0	4372.0	4372.0	4372.0	4372.0	4372.0	
mean	15299.7	92.0	1898.5	93.1	2.5	2.5	2.5	
std	1722.4	100.8	8219.3	232.5	1.1	1.1	1.1	
min	12346.0	1.0	-4287.6	1.0	1.0	1.0	1.0	
25%	13812.8	17.0	293.4	17.0	2.0	1.0	1.8	
50%	15300.5	50.0	648.1	42.0	3.0	2.0	2.5	
75%	16778.2	143.0	1611.7	102.0	4.0	3.0	3.2	
max	18287.0	374.0	279489.0	7983.0	4.0	4.0	4.0	

```
       RFM_Segment  RFM_Score
count       4372.0     4372.0
mean         278.8        7.5
std          118.8        2.8
min          111.0        3.0
25%          211.0        5.0
50%          311.0        7.0
75%          411.0       10.0
max          444.0       12.0
```

1.7.2 Centering variables with different means

- K-means works well on variables with the same mean
- Centering variables is done by subtracting average value from each observation

[90]:
```
datamart_centered = datamart_rfm - datamart_rfm.mean()
datamart_centered.describe().round(2)
```

[90]:
```
       CustomerID  Recency  MonetaryValue  Frequency        R        F  \
count     4372.00  4372.00        4372.00    4372.00  4372.00  4372.00
mean         0.00    -0.00          -0.00       0.00    -0.00     0.00
std       1722.39   100.77        8219.35     232.47     1.12     1.12
min      -2953.68   -91.05       -6186.09     -92.05    -1.51    -1.49
25%      -1486.93   -75.05       -1605.10     -76.05    -0.51    -1.49
50%          0.82   -42.05       -1250.38     -51.05     0.49    -0.49
75%       1478.57    50.95        -286.73       8.95     1.49     0.51
max       2987.32   281.95      277590.56    7889.95     1.49     1.51

             M  RFM_Segment  RFM_Score
count  4372.00      4372.00    4372.00
mean      0.00        -0.00       0.00
std       1.12       118.76       2.83
min      -1.50      -167.79      -4.50
25%      -0.75       -67.79      -2.50
50%       0.00        32.21      -0.50
75%       0.75       132.21       2.50
max       1.50       165.21       4.50
```

1.7.3 Scaling variables with different variance

- K-means works better on variables with the same variance / standard deviation
- Scaling variablkes is done by dividing them by standard deviation of each

[91]:
```
datamart_scaled = datamart_rfm / datamart_rfm.std()
datamart_scaled.describe().round(2)
```

[91]:
```
       CustomerID  Recency  MonetaryValue  Frequency        R        F  \
count     4372.00  4372.00        4372.00    4372.00  4372.00  4372.00
mean         8.88     0.91           0.23       0.40     2.24     2.22
std          1.00     1.00           1.00       1.00     1.00     1.00
min          7.17     0.01          -0.52       0.00     0.89     0.89
25%          8.02     0.17           0.04       0.07     1.78     0.89
50%          8.88     0.50           0.08       0.18     2.67     1.79
75%          9.74     1.42           0.20       0.44     3.56     2.68
max         10.62     3.71          34.00      34.34     3.56     3.57

             M  RFM_Segment  RFM_Score
count  4372.00      4372.00    4372.00
mean      2.24         2.35       2.65
std       1.00         1.00       1.00
min       0.89         0.93       1.06
25%       1.57         1.78       1.77
50%       2.24         2.62       2.48
75%       2.91         3.46       3.54
max       3.58         3.74       4.24
```

1.8 Combining centering and scaling

- Subtract mean and divide by standard deviation manually
- Or use a scaler from 'scikit-learn' library (returns numpy.ndarray object)

[92]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(datamart_rfm)
datamart_normalized = scaler.transform(datamart_rfm)
```

[93]:
```
print ('mean: ', datamart_normalized.mean(axis=0).round(2))
print ('std: ', datamart_normalized.std(axis=0).round(2))
```

```
mean:  [-0. -0. -0.  0.  0.  0.  0. -0.  0.]
std:  [1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

1.9 *Sequence of structuring pre-processing steps*

1.9.1 Why the sequence matters ?

- Log transformation only works with positive values
- Normalization forces data to have negative values and **log** wil not work

1.9.2 Sequence

1. Unskew the data - log transformation
2. Standardize the same average values
3. Scale to the same standard diviation
4. Store as separate array to be used for clustering

1.10 Coding the sequence

- Unskew the data with log transformation

[94]:
```
import numpy as np
#datamart_rfm = pd.read_excel('../data/datamart.xlsx')
datamart_log = np.log(datamart_rfm[['Frequency','Recency']])
```

- Normalize the variables with **StandardScaler**

[95]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(datamart_log)
```

[95]:
```
StandardScaler(copy=True, with_mean=True, with_std=True)
```

- Store it seperately for clustering

[96]:
```
datamart_normalized = scaler.transform(datamart_log)
```

[97]:
```
np.save('../data/datamart_normalized',datamart_normalized)
```

[ ]:
```

```