Project (CCPS 844) - Farrukh Aziz

Please click the links below to jump to the relevant area

Introduction

This project aims to apply various **supervised** and **unsupervised** machine learning algorithms.

The following steps are included within the project.

In [80]:

```
#pk.eyJ1IjoiZjhhem16IiwiYSI6ImNqb3plOWp6MjA0bXIzcnFxczZ1bjdrbmwifQ.5qd5W4B06UUZc20Jax12OA
import pandas as pd, numpy as np, matplotlib.pyplot as plt, time, plotly.plotly as py, plotly.graph_objs as go, seaborn
from scipy.cluster.hierarchy import linkage, dendrogram
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from joblib import Parallel, delayed
from ipywidgets import FloatProgress
import matplotlib.pyplot as plt
import multiprocessing
from IPython.core.display import display, HTML
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
from plotly.graph_objs import *
from plotly import tools
import cufflinks as cf
from collections import Counter
from geopy.distance import great_circle
from shapely.geometry import MultiPoint
from sklearn import metrics
from sklearn.datasets.samples_generator import make_blobs
from sklearn.model_selection import StratifiedKFold
from sklearn.feature_selection import RFECV
from sklearn import decomposition
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report,confusion_matrix
from sklearn import metrics

%matplotlib inline
%load_ext autotime

init_notebook_mode(connected=True)

# A progress bar for long running processes
# pass in total tickes needed, then update by adding 1 to object created by this function.
def __progressbar(ticks):
    __bar = FloatProgress(min=0, max=ticks)
    display(__bar)
    return __bar

import warnings; warnings.simplefilter('ignore')
import seaborn as sns

from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import BernoulliNB
from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LinearRegression


def run_classifiers(X, y, num_splits, rnd_state, __bar):

    seed = 1
    # prepare models
    models = []
    models.append(('LR', LogisticRegression()))
    models.append(('KNN', KNeighborsClassifier()))
    models.append(('LSVM', LinearSVC()))
    models.append(('SVM', SVC()))
    models.append(('DTC', DecisionTreeClassifier()))
    models.append(('BAG', BaggingClassifier()))
    models.append(('RF', RandomForestClassifier()))
    models.append(('GNB', GaussianNB()))
    models.append(('MNB', MultinomialNB()))
    models.append(('BNB', BernoulliNB()))

    # evaluate each model in turn
    results = []
    names = []
    scoring = 'accuracy'
    for name, model in models:
        kfold = model_selection.KFold(n_splits=num_splits, random_state=seed)
        cv_results = model_selection.cross_val_score(model, X, y, cv=kfold, scoring=scoring)
        results.append(cv_results)
        names.append(name)
        msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
        print(msg)
        __bar.value += 1
    return results

```

```python
from pylab import rcParams

def draw_confusion_matrix(y_test, y_pred):

    rcParams['figure.figsize'] = 5, 5
    faceLabels = ['No Fraud (0)','Fraud (1)']
    mat = confusion_matrix(y_test, y_pred)
    sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
                xticklabels = faceLabels, cmap="BuPu", linecolor='black', linewidths=1,
                yticklabels = faceLabels)
    plt.xlabel('Actual')
    plt.ylabel('Predicted')
    accuracy = metrics.accuracy_score(y_test, y_pred)
    precision = metrics.precision_score(y_test, y_pred)
    recall = metrics.recall_score(y_test, y_pred)
    display(HTML('<b>Accuracy</b> = {:.2f}'.format(accuracy * 100)))
    display(HTML('<b>Precision</b> = {:.2f}'.format(precision * 100)))
    display(HTML('<b>Recall</b> = {:.2f}'.format(recall * 100)))
    plt.show()

    return accuracy, precision, recall

class RandomForestClassifierWithCoef(RandomForestClassifier):
    def fit(self, *args, **kwargs):
        super(RandomForestClassifierWithCoef, self).fit(*args, **kwargs)
        self.coef_ = self.feature_importances_
```

```
The autotime extension is already loaded. To reload it, use:
  %reload_ext autotime

time: 14 ms
```

1. Dataset Selection

The dataset I have selected is from **Kaggle**. It is Paysim synthetic dataset of mobile money transactions. Each step represents an hour of simulation. It can be downloaded from the following URL:

https://www.kaggle.com/ntnu-testimon/paysim1/downloads/paysim1.zip/2 (https://www.kaggle.com/ntnu-testimon/paysim1/downloads/paysim1.zip/2)

It has the following attributes:

1. step: Maps a unit of time in the real world. In this case 1 step is 1 hour of time.
2. type: CASH-IN, CASH-OUT, DEBIT, PAYMENT and TRANSFER.
3. amount: Amount of the transaction in local currency.
4. nameOrig: Customer who started the transaction.
5. oldbalanceOrg: Initial balance before the transaction.
6. newbalanceOrig: Customer's balance after the transaction.
7. nameDest: Recipient ID of the transaction.
8. oldbalanceDest: Initial recipient balance before the transaction.
9. newbalanceDest: Recipient's balance after the transaction.
10. isFraud: Identifies a fraudulent transaction (1) and non fraudulent (0).
11. isFlaggedFraud: Flags illegal attempts to transfer more than 200.000 in a single transaction.

In [6]:

```python
mobile_txns_file = "ps_transactions_log.csv"
df_txns_full = pd.read_csv(mobile_txns_file)
df_txns_full.head(5)
```

Out[6]:

	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
0	1	PAYMENT	9839.64	C1231006815	170136.0	160296.36	M1979787155	0.0	0.0	0	0
1	1	PAYMENT	1864.28	C1666544295	21249.0	19384.72	M2044282225	0.0	0.0	0	0
2	1	TRANSFER	181.00	C1305486145	181.0	0.00	C553264065	0.0	0.0	1	0
3	1	CASH_OUT	181.00	C840083671	181.0	0.00	C38997010	21182.0	0.0	1	0
4	1	PAYMENT	11668.14	C2048537720	41554.0	29885.86	M1230701703	0.0	0.0	0	0

```
time: 13 s
```

2. Attribute Analysis

As we can see below, **type, nameOrig** and **nameDest** are objects, meaning they will have to be converted to labels (levels) to be used in some algorithms. **step** is int, however, it also needs to be encoded because it doesn't reprent a numeric value, rather a ordinal (categorical) value.

Are there any null values? If they are, they need to be cleaned up. Turns out that the data has already been cleaned up, there are no null values.

In [7]:
```
df_txns_full.isnull().any()
```

Out[7]:
```
step              False
type              False
amount            False
nameOrig          False
oldbalanceOrg     False
newbalanceOrig    False
nameDest          False
oldbalanceDest    False
newbalanceDest    False
isFraud           False
isFlaggedFraud    False
dtype: bool

time: 1.45 s
```

In [8]:
```
display(df_txns_full.describe())
display(df_txns_full.info())
```

	step	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
count	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06	6.362620e+06
mean	2.433972e+02	1.798619e+05	8.338831e+05	8.551137e+05	1.100702e+06	1.224996e+06	1.290820e-03	2.514687e-06
std	1.423320e+02	6.038582e+05	2.888243e+06	2.924049e+06	3.399180e+06	3.674129e+06	3.590480e-02	1.585775e-03
min	1.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
25%	1.560000e+02	1.338957e+04	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
50%	2.390000e+02	7.487194e+04	1.420800e+04	0.000000e+00	1.327057e+05	2.146614e+05	0.000000e+00	0.000000e+00
75%	3.350000e+02	2.087215e+05	1.073152e+05	1.442584e+05	9.430367e+05	1.111909e+06	0.000000e+00	0.000000e+00
max	7.430000e+02	9.244552e+07	5.958504e+07	4.958504e+07	3.560159e+08	3.561793e+08	1.000000e+00	1.000000e+00

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6362620 entries, 0 to 6362619
Data columns (total 11 columns):
step              int64
type              object
amount            float64
nameOrig          object
oldbalanceOrg     float64
newbalanceOrig    float64
nameDest          object
oldbalanceDest    float64
newbalanceDest    float64
isFraud           int64
isFlaggedFraud    int64
dtypes: float64(5), int64(3), object(3)
memory usage: 534.0+ MB

None

time: 2.12 s
```

It also shows that this dataset has over 6 million rows, we will reduce the number of rows to a smaller subset of data, so that the processing can be optimized.

In [9]:
```
df_fraud = df_txns_full[df_txns_full['isFraud'] == 1]
df_legit = df_txns_full[df_txns_full['isFraud'] == 0]
print("Total fraud txns: {}".format(len(df_fraud)))
print("Total legit txns: {}".format(len(df_legit)))

df_legit_sub = df_legit.head(100000 - len(df_fraud))
df_txns = pd.concat([df_legit_sub, df_fraud], axis = 0).reset_index(drop=True)
print("Selected subset: {}".format(len(df_txns)))

# shuffle rows so that fraud and legit rows are mixed
df_txns = df_txns.sample(frac=1).reset_index(drop=True)

# Save data for easy load
df_txns.to_pickle('df_txns.pkl')
```

```
Total fraud txns: 8213
Total legit txns: 6354407
Selected subset: 100000
time: 641 ms
```

isFlaggedFraud appears to be a redundant attribute. By definition, any transcation that is over $200,000 is marked as isFlaggedFraud. Let's check whether there are any transaction that is marked as isFlaggedFraud but is not marked with flag isFraud.

In [10]:
```
df_txns = pd.read_pickle('df_txns.pkl')
print(len(df_txns[(df_txns['isFlaggedFraud'] == 1) & (df_txns['isFraud'] != 1)]))
```

```
0
time: 36.2 ms
```

Drop the column since it is in fact redundant with isFraud

In [11]:
```
df_txns_clean1 = df_txns.drop(['isFlaggedFraud'], axis = 1)
```

```
time: 5.27 ms
```

nameOrig column is unique categorical identifier, therefore, it will not add any information to our model, therefore, it can be dropped. Similarly, ***nameDest*** is a categorical values that repeats only twice on average, doesn't provide much variance and can be dropped.

In [12]:
```
print(len(df_txns_clean1))
print(len(df_txns_clean1['nameOrig'].unique()))
print(len(df_txns_clean1['nameDest'].unique()))
df_txns_cln = df_txns_clean1.drop(['nameOrig', 'nameDest'], axis = 1)
```

```
100000
100000
55440
time: 28.9 ms
```

In [13]:
```
df_txns_cln.to_pickle('df_txns_cln.pkl')
```

```
time: 14.8 ms
```

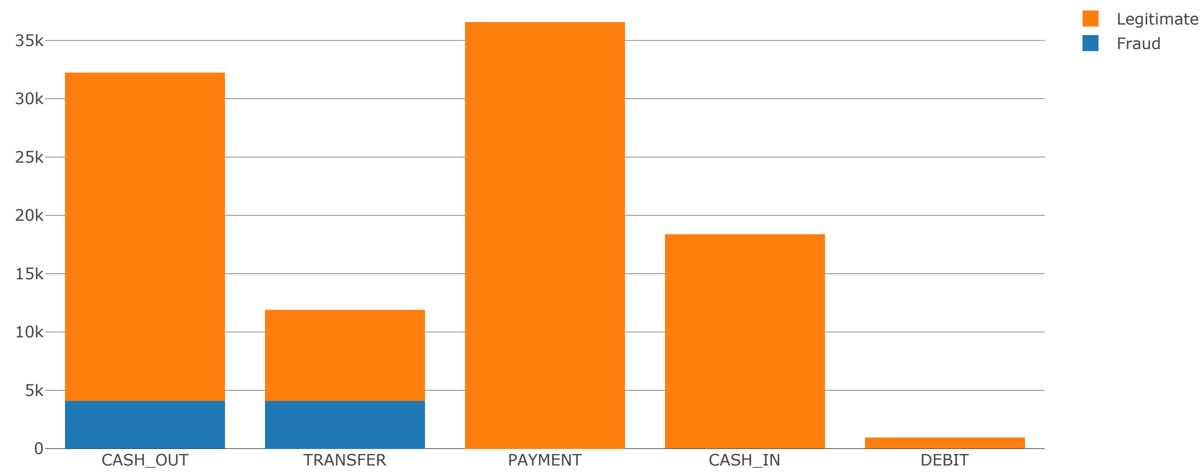
---

3. Data Visualization

type is a categorical variable with 5 different categories, lets visualize them.

In [14]:

```
df_txns = pd.read_pickle('df_txns_cln.pkl')
countsFraud = df_txns[df_txns['isFraud']==1]['type'].value_counts()
countsLegit = df_txns[df_txns['isFraud']==0]['type'].value_counts()

data = [go.Bar(
            x=countsFraud.index,
            y=countsFraud.values,
            name = 'Fraud'
    ),
        go.Bar(
            x=countsLegit.index,
            y=countsLegit.values,
            name = 'Legitimate'
    )]
layout = go.Layout(barmode='stack')
fig = go.Figure(data=data, layout=layout)
iplot(fig)
```



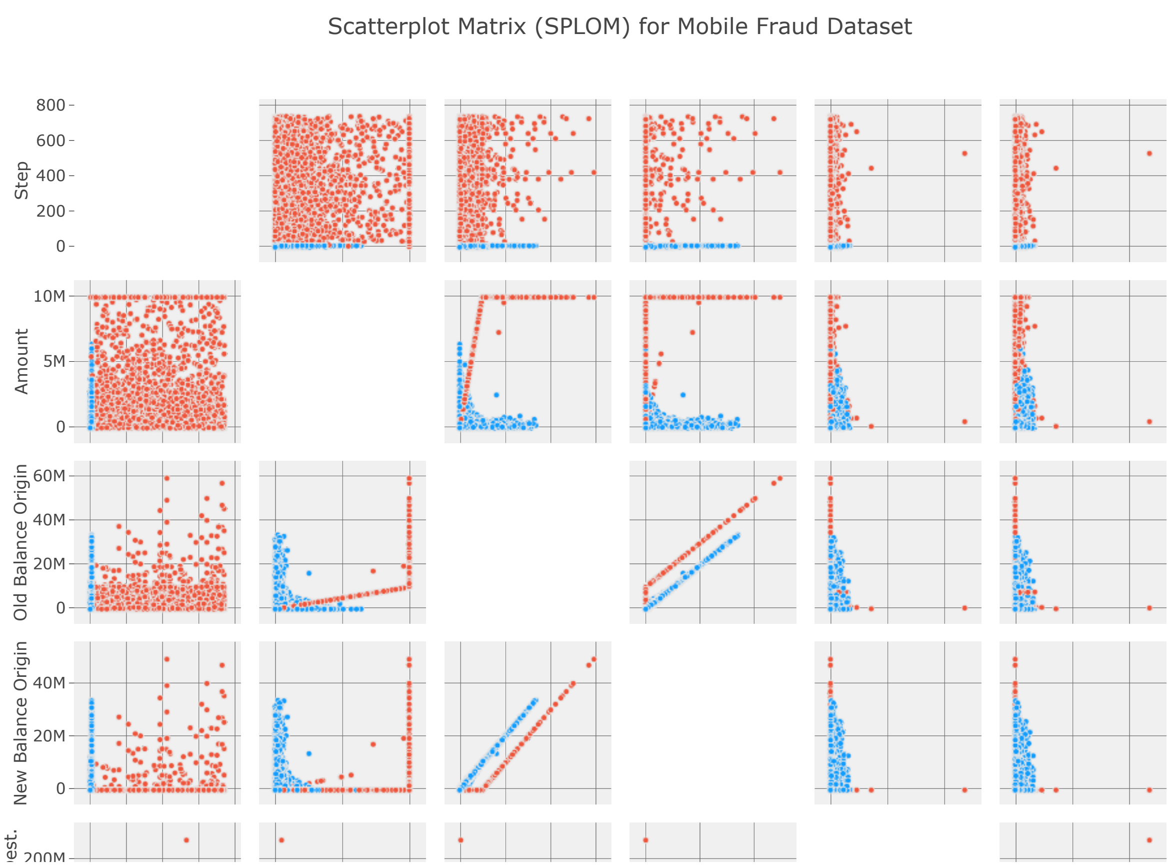
Export to plot.ly »

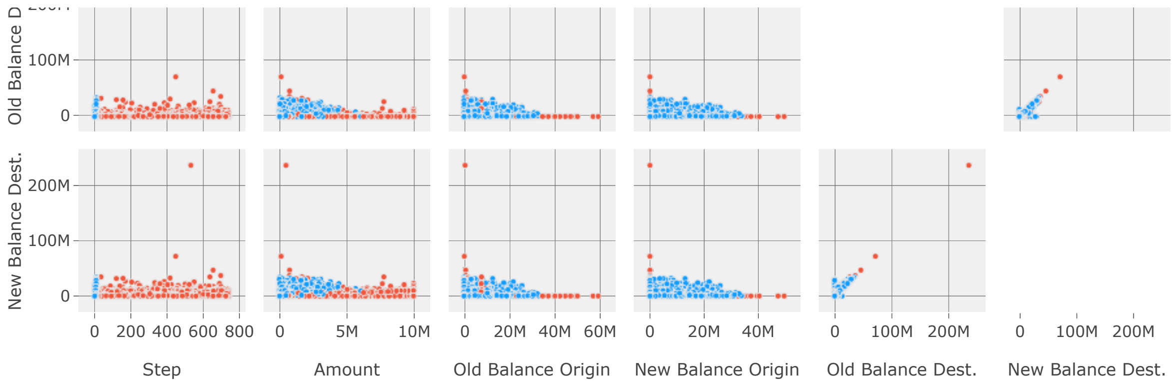
```
time: 738 ms
```

The bar graph above shows that the only two type of transactions that suffer fraud are ***CASH-OUT*** and ***TRANSFER***.

In [15]:

```python
textd = ['isLegit' if cl==0 else 'isFraud' for cl in df_txns['isFraud']]
color_vals = [0  if cl==0 else 1 for cl in df_txns['isFraud']]
pl_colorscaled = [[0., '#119dff'],
                  [0.5, '#119dff'],
                  [0.5, '#ef553b'],
                  [1, '#ef553b']]
traced = go.Splom(dimensions=[dict(label='Step', values=df_txns['step']),
                              dict(label='Amount', values=df_txns['amount']),
                              dict(label='Old Balance Origin', values=df_txns['oldbalanceOrg']),
                              dict(label='New Balance Origin', values=df_txns['newbalanceOrig']),
                              dict(label='Old Balance Dest.', values=df_txns['oldbalanceDest']),
                              dict(label='New Balance Dest.', values=df_txns['newbalanceDest'])],
                  marker=dict(color=color_vals,
                              size=5,
                              colorscale=pl_colorscaled,
                              line=dict(width=0.5,
                                        color='rgb(230,230,230)') ),
                  text=textd,
                  diagonal=dict(visible=False))
axisd = dict(showline=False,
           zeroline=False,
           gridcolor='#fff',
           ticklen=4,
           titlefont=dict(size=13))
title = "Scatterplot Matrix (SPLOM) for Mobile Fraud Dataset"
layout = go.Layout(title=title,
                   dragmode='select',
                   width=1000,
                   height=1000,
                   autosize=False,
                   hovermode='closest',
                   plot_bgcolor='rgba(240,240,240, 0.95)',
                   xaxis1=dict(axisd),
                   xaxis2=dict(axisd),
                   xaxis3=dict(axisd),
                   xaxis4=dict(axisd),
                   xaxis5=dict(axisd),
                   xaxis6=dict(axisd),
                   yaxis1=dict(axisd),
                   yaxis2=dict(axisd),
                   yaxis3=dict(axisd),
                   yaxis4=dict(axisd),
                   yaxis5=dict(axisd),
                   yaxis6=dict(axisd))

fig = dict(data=[traced], layout=layout)
iplot(fig, filename='large')
```





Export to plot.ly »

```
time: 5.4 s
```

step is an oridinal (categorical) variable. It means it can be encoded using label encoder.

In [16]:
```
lbl_encoder = LabelEncoder()

df_txns['step'] = lbl_encoder.fit_transform(df_txns['step'])
```

```
time: 7.05 ms
```

type is a categorical variable as well, however, it is nominal (not ordinal), therefore, simply applying Label Encoder won't work. We will have to create dummies out it or apply One Hot Encoder.

In [17]:
```
df_dummies = pd.get_dummies(df_txns['type'])
df_txns_d = df_txns.merge(df_dummies, left_index=True, right_index=True)
df_txns_d = df_txns_d.drop(['type'], axis = 1)

df_txns_d.to_pickle('df_txns_d.pkl')
df_txns_d.head(5)
```

Out[17]:

	step	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	isFraud	CASH_IN	CASH_OUT	DEBIT	PAYMENT	TRANSFER
0	316	322991.82	322991.82	0.00	0.00	0.00	1	0	0	0	0	1
1	7	10151.23	49964.00	39812.77	0.00	0.00	0	0	0	0	1	0
2	8	1362303.35	0.00	0.00	1493707.41	0.00	0	0	0	0	0	1
3	7	69093.52	20212987.30	20282080.82	7999977.05	7930883.53	0	1	0	0	0	0
4	8	151280.92	7669110.25	7820391.17	257041.24	105760.33	0	1	0	0	0	0

```
time: 36.9 ms
```

4. Unsupervised Machine Learning

1. K-Means Clustering

Initially, we will start with **K = 2** clusters, do the analysis and repeat with appropriate clusters

In [18]:
```
from sklearn.cluster import KMeans

df_txns_d = pd.read_pickle('df_txns_d.pkl').drop(['isFraud'], axis = 1)

k_means = KMeans(n_clusters = 2)

k_means.fit(df_txns_d)
```

Out[18]:
```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=2, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
```

```
time: 843 ms
```

Find labels of the K-Means predictions

In [19]:
```python
labels = k_means.labels_

print(labels)
```

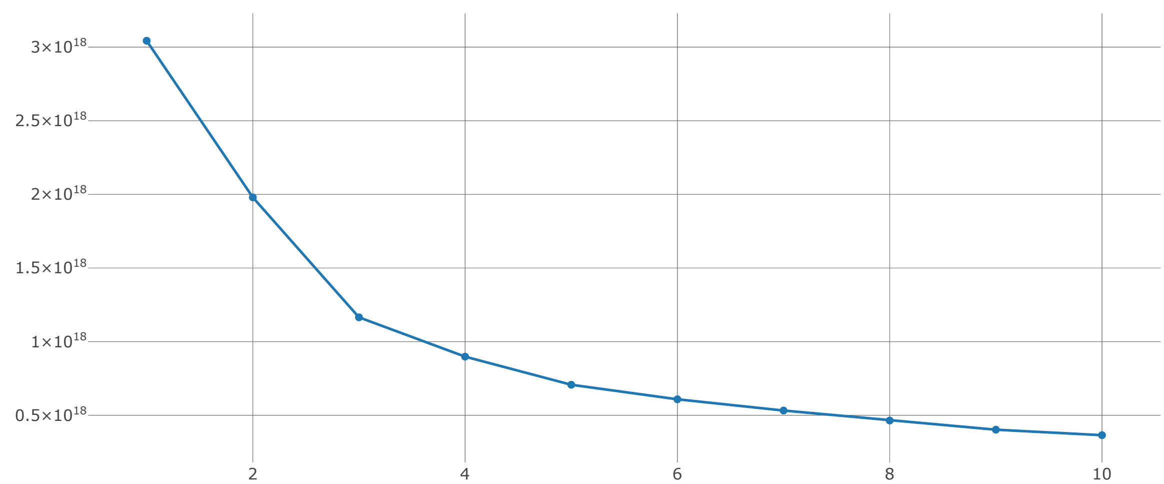
```
[0 0 0 ... 0 1 0]
time: 956 µs
```

We can use inertia plot to find the best value for **K** parameter. Lower value of inertia corresponds to smaller clusters.

In [20]:
```python
n = list(range(1,11))
inertia = []

for k in n:
    k_means = KMeans(n_clusters = k)
    k_means.fit(df_txns_d)
    inertia.append(k_means.inertia_)
print(inertia)
```

```
[3.043355109395162e+18, 1.9798418781012442e+18, 1.1652894075927675e+18, 8.981842093894879e+17, 7.073619540534374e+17, 6.
086238671884495e+17, 5.3230685618731136e+17, 4.6441703290762464e+17, 4.021963159789477e+17, 3.649224591718327e+17]
time: 20.2 s
```

In [21]:
```python
iplot([{
    'x': n,
    'y': inertia,
    'name': "K-Means K vs Inertia"
}])
```



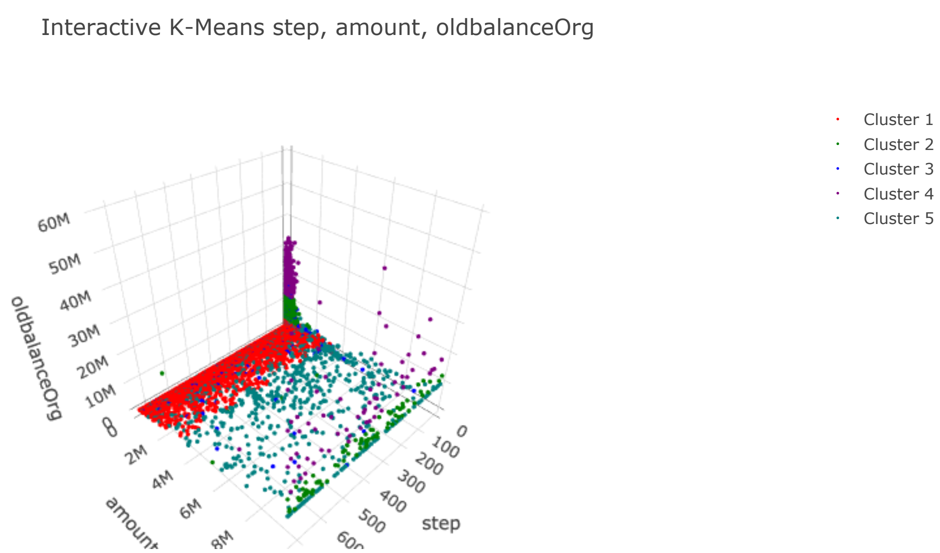
Export to plot.ly »

```
time: 198 ms
```

This graph above shows that at **K = 5**, the drop in inertia slows down. Therefore, it is the ideal value for **K**.

In [22]:
```python
k_means = KMeans(n_clusters = 5)

k_means.fit(df_txns_d)

df_txns_d['labels'] = k_means.labels_

print(k_means.cluster_centers_)

# graph in 3d for various combinations of columns that may yeild some insight
cols = [(0,1,2),(0,1,3),(0,1,4),(1,2,3),(1,4,5)]
colors = ['red','green','blue','purple','teal']

for lim in cols:
    data = []

    col_sets = df_txns_d.columns[[lim[0],lim[1],lim[2]]].values
    print(col_sets)

    for cluster in range(len(df_txns_d['labels'].unique())):
        # current cluster data subset with 3 columns only
        c_data = df_txns_d[df_txns_d['labels'] == cluster][col_sets]

        scatterPlot = dict(
            type = 'scatter3d',
            mode = "markers",
            name = "Cluster " + str(cluster + 1),
            x = c_data.values[:,0], y = c_data.values[:,1], z = c_data.values[:,2],
            marker = dict( size=2, color=colors[cluster])
        )

        data.append(scatterPlot)

    layout = dict(
        title = 'Interactive K-Means ' + ', '.join(col_sets),
        scene = dict(
            xaxis = dict( zeroline=True, title=col_sets[0] ),
            yaxis = dict( zeroline=True, title=col_sets[1] ),
            zaxis = dict( zeroline=True, title=col_sets[2] ),
        )
    )
    iplot(dict(data = data, layout=layout))
```

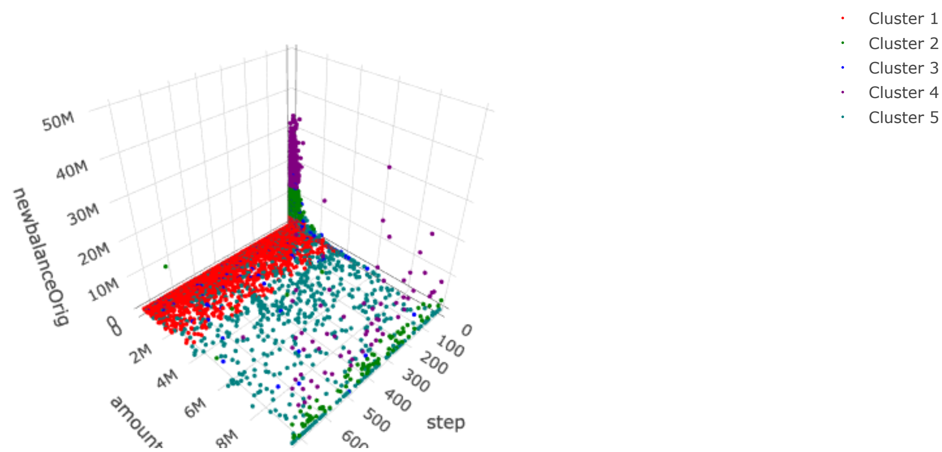
```
[[ 3.73488177e+01  1.85179930e+05  2.37774024e+05  1.77068418e+05
   2.25642969e+05  3.93116212e+05  1.14379997e-01  3.25099785e-01
   1.04092106e-02  4.41457289e-01  1.08653718e-01]
 [ 1.91521672e+01  4.22518779e+05  6.77158915e+06  6.63710370e+06
   9.86185197e+05  1.04969382e+06  9.29449739e-01  6.76298801e-03
   1.53704273e-04  2.92038119e-02  3.44297571e-02]
 [ 2.41087449e+01  5.78672783e+05  1.37940114e+06  1.32521044e+06
   1.35011540e+07  1.52686219e+07  2.88173992e-01  4.42229271e-01
   8.15586769e-03  1.85407245e-14  2.61440870e-01]
 [ 3.74071970e+01  7.77518780e+05  1.95536787e+07  1.91169740e+07
   1.56213057e+06  1.66948359e+06  9.35606061e-01  9.46969697e-04
  -6.59194921e-17  8.77076189e-15  6.34469697e-02]
 [ 5.01928498e+01  9.67025560e+05  8.52156135e+05  3.44805654e+05
   3.66179695e+06  5.64360455e+06  1.61327524e-01  5.68939006e-01
   7.43208611e-03 -4.79061235e-14  2.62301384e-01]]
['step' 'amount' 'oldbalanceOrg']
```



Export to plot.ly »

['step' 'amount' 'newbalanceOrig']

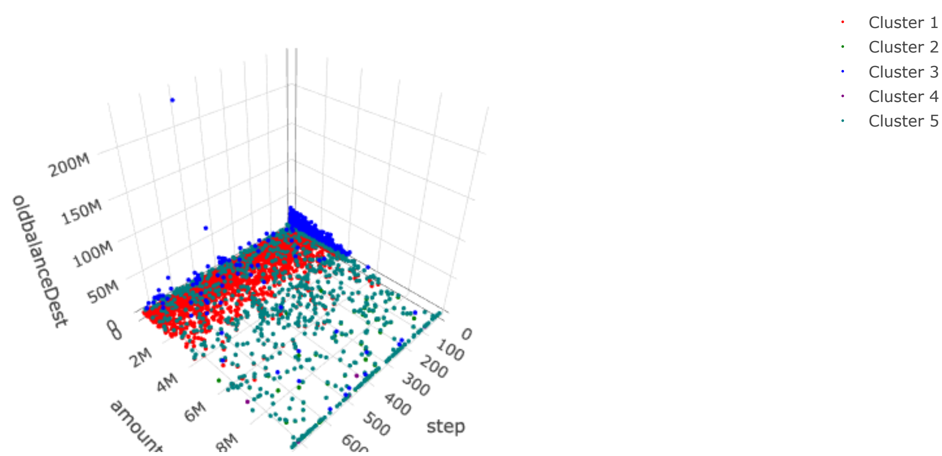
Interactive K-Means step, amount, newbalanceOrig



Export to plot.ly »

['step' 'amount' 'oldbalanceDest']

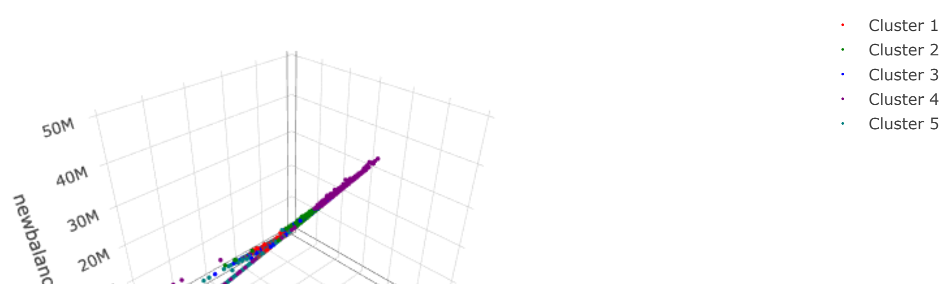
Interactive K-Means step, amount, oldbalanceDest

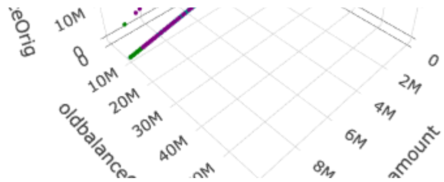


Export to plot.ly »

['amount' 'oldbalanceOrg' 'newbalanceOrig']

Interactive K-Means amount, oldbalanceOrg, newbalanceOrig

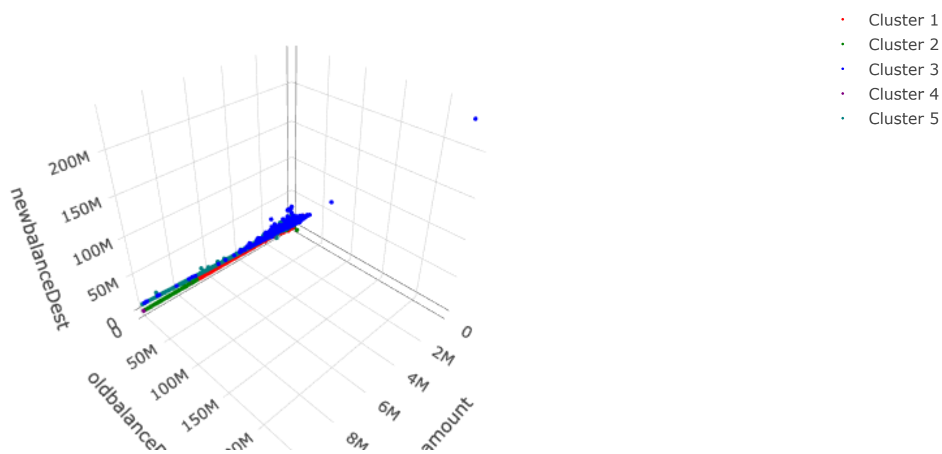




Export to plot.ly »

```
['amount' 'oldbalanceDest' 'newbalanceDest']
```

Interactive K-Means amount, oldbalanceDest, newbalanceDest

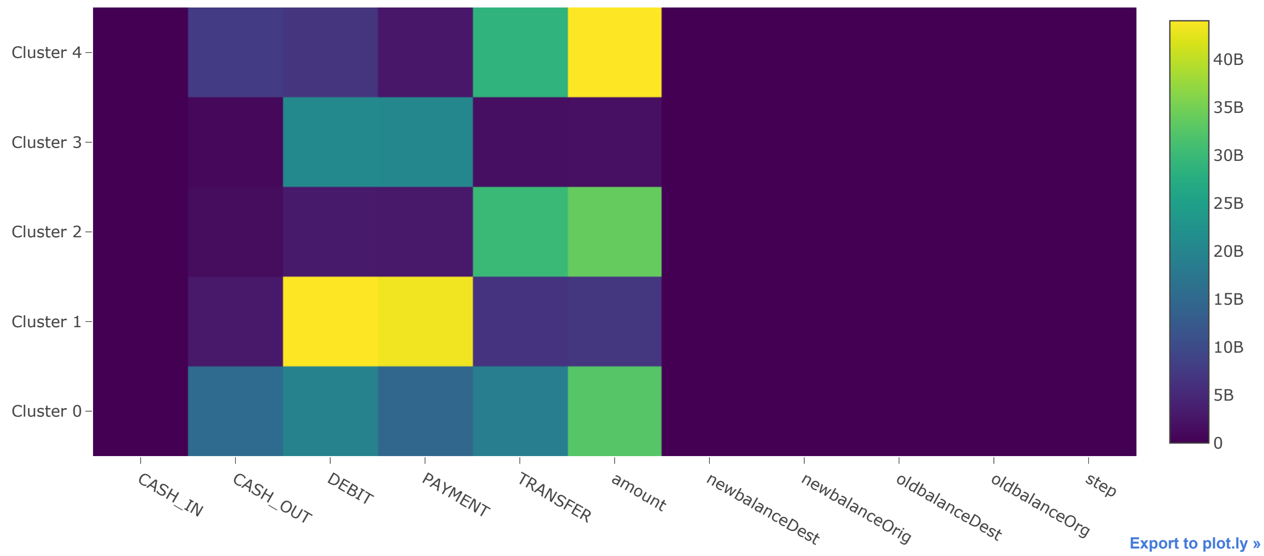


Export to plot.ly »

```
time: 10.1 s
```

Draw a heatmap with total clustered points for each attributes to observe which cluster contains most information regarding which attribute

In [23]:
```
clusters = ['Cluster ' + str(x) for x in list(range(0,5))]

sums = df_txns_d.groupby(['labels'], sort=True).sum()

data = [go.Heatmap( z=sums.values.tolist(),
                    y= clusters,
                    x= df_txns_d.columns.difference(['labels']).values,
                    colorscale='Viridis')]

iplot(data)
```




```
time: 134 ms
```

The above graphs and heatmap reveal that K-Means has clustered most of the data by amount and the 5 different types of transactions. This reveals that the amounts strongly correlate with the types of transactions.

2. Heirarchical Clustering

Heirarchical Clustering is performed by type of each transaction for a random sample of 50 transactions

In [24]:
```
df_txns_h = pd.read_pickle('df_txns_cln.pkl')
df_txns_h.head(5)
```

Out[24]:

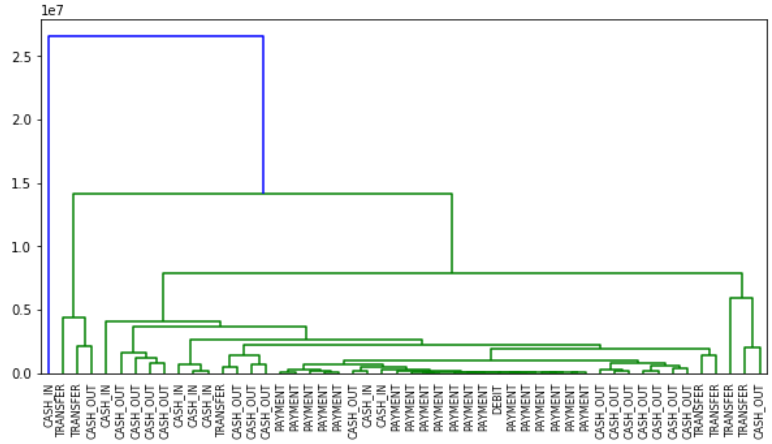
	step	type	amount	oldbalanceOrg	newbalanceOrig	oldbalanceDest	newbalanceDest	isFraud
0	317	TRANSFER	322991.82	322991.82	0.00	0.00	0.00	1
1	8	PAYMENT	10151.23	49964.00	39812.77	0.00	0.00	0
2	9	TRANSFER	1362303.35	0.00	0.00	1493707.41	0.00	0
3	8	CASH_IN	69093.52	20212987.30	20282080.82	7999977.05	7930883.53	0
4	9	CASH_IN	151280.92	7669110.25	7820391.17	257041.24	105760.33	0

```
time: 16.3 ms
```

In [25]:
```
df_txns_h_sub = df_txns_h.sample(50, random_state=2)
txn_types = list(df_txns_h_sub.pop('type'))
samples = df_txns_h_sub.values
mergings = linkage(samples, method='complete')
```

```
time: 5.33 ms
```

In [26]:
```
plt.figure(figsize=(10, 5))

dendrogram(mergings,
           labels=txn_types
)
plt.show()
```



time: 217 ms

Heirarchical clustering shows a strong relationship between CASH-IN and CASH-OUT transactions. TRANSFER transactions are in the middle of these transactions. PAYMENTS and DEBIT have a distant relationship with other 3 types.

---

5. Feature Selection

1. Cross-Validation Score (All Classifiers w/ K = 10)

Data must be scaled to the same level before applying cross-validation for classifiers.

In [27]:
```
df_txns_d = pd.read_pickle('df_txns_d.pkl')

scaler = MinMaxScaler()

y = df_txns_d.pop('isFraud').values
X = df_txns_d

X_scaled = scaler.fit_transform(X)
X_scaled
```

Out[27]:
```
array([[4.27027027e-01, 3.22991820e-02, 5.42068643e-03, ...,
        0.00000000e+00, 0.00000000e+00, 1.00000000e+00],
       [9.45945946e-03, 1.01512300e-03, 8.38532620e-04, ...,
        0.00000000e+00, 1.00000000e+00, 0.00000000e+00],
       [1.08108108e-02, 1.36230335e-01, 0.00000000e+00, ...,
        0.00000000e+00, 0.00000000e+00, 1.00000000e+00],
       ...,
       [1.21621622e-02, 1.87495000e-03, 1.26588398e-02, ...,
        0.00000000e+00, 1.00000000e+00, 0.00000000e+00],
       [9.45945946e-03, 4.90225580e-02, 8.19862548e-02, ...,
        0.00000000e+00, 0.00000000e+00, 0.00000000e+00],
       [0.00000000e+00, 2.00079100e-03, 0.00000000e+00, ...,
        0.00000000e+00, 1.00000000e+00, 0.00000000e+00]])
```

time: 27.2 ms

Calculate cross-validation accuracy score for each classifier with Folds(K) = 10

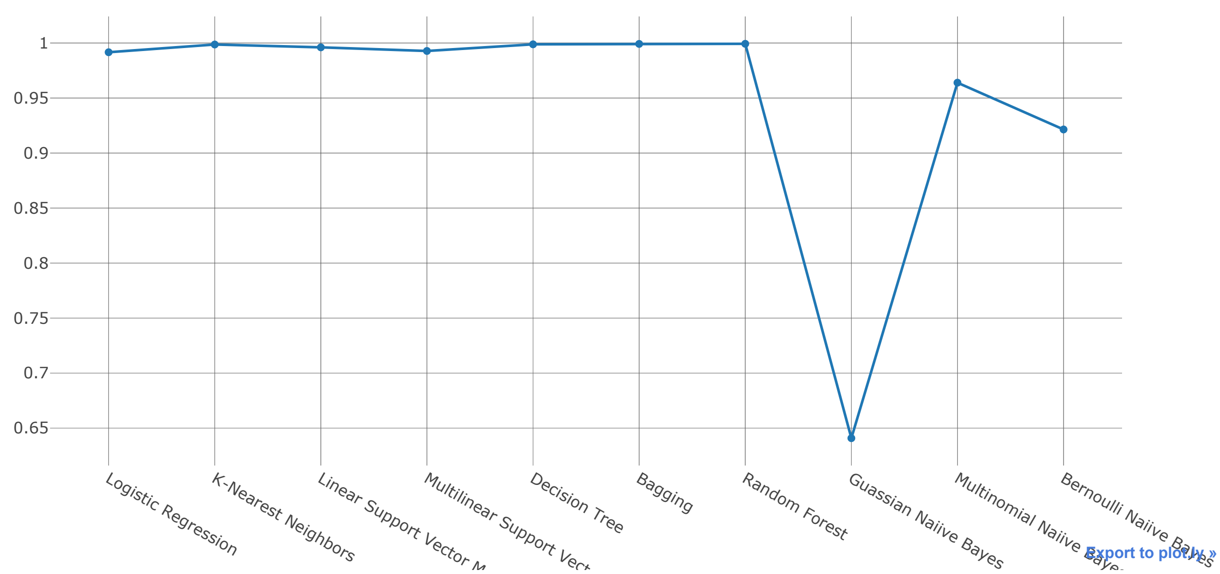
In [28]:
```
progress_bar = __progressbar(10)
result = run_classifiers(X = X_scaled, y = y, num_splits = 10, rnd_state = 1, __bar = progress_bar)
```

```
LR: 0.991620 (0.000815)
KNN: 0.998640 (0.000367)
LSVM: 0.996060 (0.000550)
SVM: 0.992760 (0.000554)
DTC: 0.998810 (0.000298)
BAG: 0.999180 (0.000275)
RF: 0.999280 (0.000194)
GNB: 0.640850 (0.004636)
MNB: 0.963940 (0.001551)
BNB: 0.921430 (0.002845)
time: 2min 46s
```

In [29]:
```
pd.DataFrame({'results': [result]}, columns=['results']).to_pickle('df_cv.pkl')
```

```
time: 4.39 ms
```

In [31]:
```
cross_val_results = pd.read_pickle('df_cv.pkl')['results'][0]

models = ['Logistic Regression','K-Nearest Neighbors','Linear Support Vector Machine',\
          'Multilinear Support Vector Machine','Decision Tree','Bagging','Random Forest',\
          'Guassian Naiive Bayes','Multinomial Naiive Bayes','Bernoulli Naiive Bayes']
mean_cross_val = []
for x in cross_val_results:
    mean_cross_val.append(np.mean(x))
mean_cross_val

iplot([{
    'x': models,
    'y': mean_cross_val,
    'name': "Cross Validation Mean"
}], filename='cufflinks/classifiers-cmp')
```




```
time: 81 ms
```

It is clear from the graph above that ***Random Forest*** has the highest accuracy based on cross-validation score.

2 - Recursive Feature Elimination by Cross-Validation (RFECV)

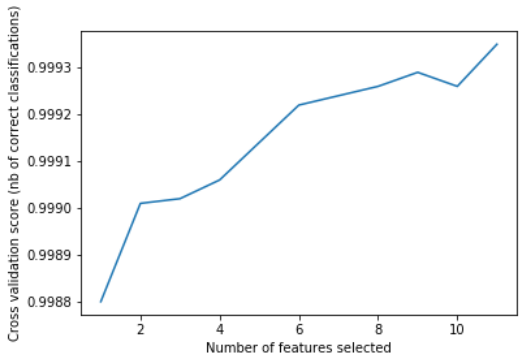
We will perform RFECV using Random Forest Model as it has scored the highest in cross-validation score.

Accuracy Curve

In [33]:

```
nb=RandomForestClassifierWithCoef()

rfecv = RFECV(estimator=nb, step=1, cv=StratifiedKFold(10),
              scoring='accuracy')
rfecv.fit(X, y)
print(type(rfecv.grid_scores_))
plt.figure()
plt.xlabel("Number of features selected")
plt.ylabel("Cross validation score (nb of correct classifications)")
plt.plot(range(1, len(rfecv.grid_scores_) + 1), rfecv.grid_scores_)
plt.show()
```

```
<class 'numpy.ndarray'>
```




```
time: 52.4 s
```

In [36]:

```
df_ranks = pd.DataFrame({'cols': X.columns,'rank': rfecv.ranking_}).\
sort_values(['rank']).reset_index(drop=True)

df_ranks
```

Out[36]:

	cols	rank
0	step	1
1	amount	1
2	oldbalanceOrg	1
3	newbalanceOrig	1
4	oldbalanceDest	1
5	newbalanceDest	1
6	CASH_IN	1
7	CASH_OUT	1
8	DEBIT	1
9	PAYMENT	1
10	TRANSFER	1

```
time: 6.45 ms
```

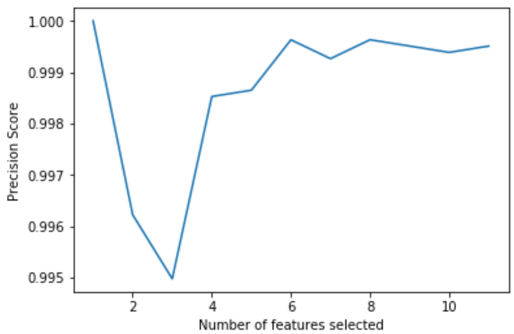
From ranking = 1 for all attributes of the data, it is clear that all attributes must be used for analysis and none of them can be dropped from accuracy point of view.

Precision Curve

In [37]:

```
nb=RandomForestClassifierWithCoef()

rfecv = RFECV(estimator=nb, step=1, cv=StratifiedKFold(10),
              scoring='precision')
rfecv.fit(X, y)
print(type(rfecv.grid_scores_))
plt.figure()
plt.xlabel("Number of features selected")
plt.ylabel("Precision Score")
plt.plot(range(1, len(rfecv.grid_scores_) + 1), rfecv.grid_scores_)
plt.show()
```

```
<class 'numpy.ndarray'>
```




```
time: 56.3 s
```

In [38]:

```
df_ranks = pd.DataFrame({'cols': X.columns,'rank': rfecv.ranking_}).\
sort_values(['rank']).reset_index(drop=True)

df_ranks
```

Out[38]:

	cols	rank
0	step	1
1	newbalanceDest	2
2	TRANSFER	3
3	oldbalanceOrg	4
4	amount	5
5	oldbalanceDest	6
6	newbalanceOrig	7
7	CASH_OUT	8
8	CASH_IN	9
9	PAYMENT	10
10	DEBIT	11

```
time: 6.5 ms
```

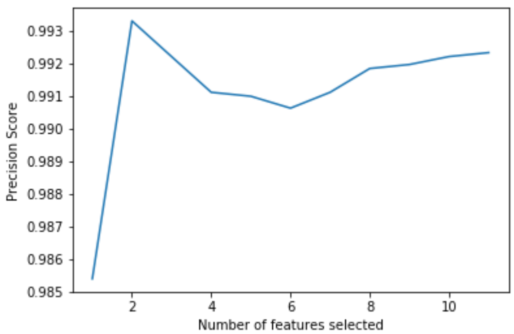
This graph shows that highest precision can be acheive by selection first 8 attributes

Recall Curve

In [39]:

```
nb=RandomForestClassifierWithCoef()

rfecv = RFECV(estimator=nb, step=1, cv=StratifiedKFold(10),
               scoring='recall')
rfecv.fit(X, y)
print(type(rfecv.grid_scores_))
plt.figure()
plt.xlabel("Number of features selected")
plt.ylabel("Precision Score")
plt.plot(range(1, len(rfecv.grid_scores_) + 1), rfecv.grid_scores_)
plt.show()
```

```
<class 'numpy.ndarray'>
```




```
time: 59.3 s
```

In [40]:

```
df_ranks = pd.DataFrame({'cols': X.columns,'rank': rfecv.ranking_}).\
sort_values(['rank']).reset_index(drop=True)

df_ranks
```

Out[40]:

	cols	rank
0	step	1
1	amount	1
2	oldbalanceOrg	2
3	TRANSFER	3
4	newbalanceDest	4
5	oldbalanceDest	5
6	PAYMENT	6
7	CASH_IN	7
8	CASH_OUT	8
9	newbalanceOrig	9
10	DEBIT	10

```
time: 6.59 ms
```

It shows that highest Recall is possible with just two attributes, step and amount.

6. Dimensionality Reduction

Principal Component Analysis

We will pick **6** components for component analysis and compare performance against original components.

In [81]:
```python
df_txns_d = pd.read_pickle('df_txns_d.pkl')

y = df_txns_d.pop('isFraud').values
X = df_txns_d

pca = decomposition.PCA(n_components=6)
pca.fit(X)
X_pca = pca.transform(X)
X_pca
```

Out[81]:
```
array([[-1.71613684e+06, -4.49702952e+05, -1.34653903e+05,
         2.10437460e+05, -6.77625152e+04,  2.59276299e+02],
       [-1.86134574e+06, -3.43542559e+05,  2.02640350e+05,
         4.35204092e+04, -6.40043943e+04, -2.90143009e+01],
       [-1.21578559e+06,  4.35688576e+05, -2.69455633e+05,
         1.50415714e+06,  7.62600285e+05, -1.42658871e+01],
       ...,
       [-1.10584929e+06, -9.82726361e+05,  2.23818880e+05,
         2.46837838e+04, -6.81142456e+04, -2.63698989e+01],
       [ 4.84669985e+06, -3.55867639e+06,  5.20989328e+05,
         3.54477232e+05,  4.70853867e+05,  2.54156525e+01],
       [-1.90941070e+06, -3.02445521e+05,  1.96822222e+05,
         4.62233518e+04, -5.22464366e+04, -3.51002626e+01]])
```

time: 148 ms

7. Test, Train, Split

In [82]:
```python
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1, test_size = 0.3)
X_pca_train, X_pca_test, y_pca_train, y_pca_test = train_test_split(X_pca, y, random_state=1, test_size = 0.3)
```

time: 15.5 ms

8. Data Scaling

In [83]:
```python
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()
X_train_scaled = sc.fit_transform(X_train)
X_test_scaled = sc.transform(X_test)

sc = StandardScaler()
X_pca_train_scaled = sc.fit_transform(X_pca_train)
X_pca_test_scaled = sc.transform(X_pca_test)

# Some classifiers such as Multinomial Naiive Bayes don't accept negative values
# therefore, MinMaxScaler with default range of 0 to 1 is used.
scm = MinMaxScaler()
X_train_mm = scm.fit_transform(X_train)
X_test_mm = scm.transform(X_test)

scm = MinMaxScaler()
X_pca_train_mm = scm.fit_transform(X_pca_train)
X_pca_test_mm = scm.transform(X_pca_test)

```

time: 59.5 ms

9. Supervised Machine Learning

In [98]:
```python
models = ['Logistic Regression','K-Nearest Neighbors','Linear Support Vector Machine',\
          'Multilinear Support Vector Machine','Decision Tree','Bagging','Random Forest',\
          'Guassian Naiive Bayes','Multinomial Naiive Bayes','Bernoulli Naiive Bayes']

df_stats = pd.DataFrame(models, columns=['model'])
df_stats.set_index('model')
```

Out[98]:

model
Logistic Regression
K-Nearest Neighbors
Linear Support Vector Machine
Multilinear Support Vector Machine
Decision Tree
Bagging
Random Forest
Guassian Naiive Bayes
Multinomial Naiive Bayes
Bernoulli Naiive Bayes

```
time: 5.33 ms
```

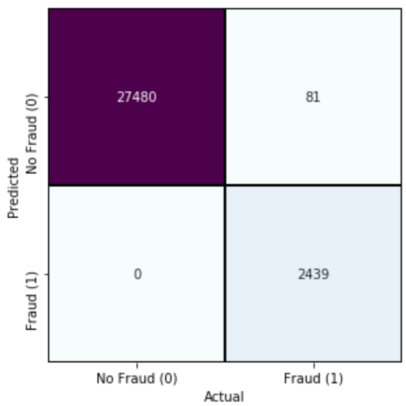
1. Logistic Regression

In [99]:
```
model = LogisticRegression()

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)

accuracy, precision, recall = draw_confusion_matrix(y_test, y_pred)

display(HTML('<b>With PCA</b>'))

model.fit(X_pca_train_scaled, y_train)

y_pca_pred = model.predict(X_pca_test_scaled)

accuracy, precision, recall = draw_confusion_matrix(y_test, y_pca_pred)


```

Accuracy = 99.73

Precision = 100.00

Recall = 96.79

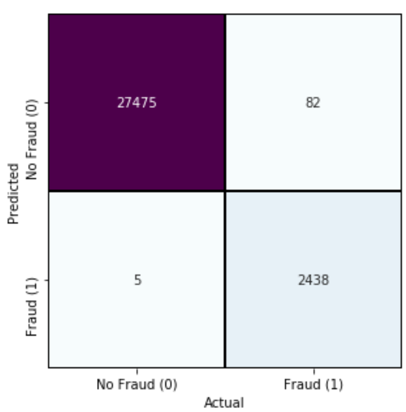


With PCA

Accuracy = 99.71

Precision = 99.80

Recall = 96.75




```
time: 534 ms
```

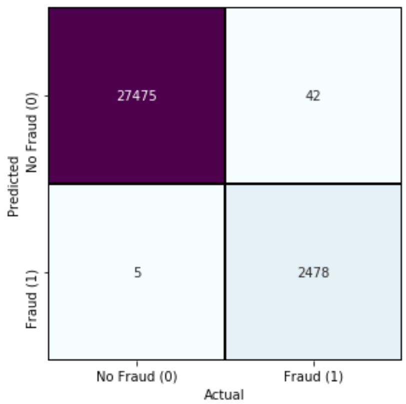
2. K-Nearest Neighbors (KNN)

In [87]:
```
model = KNeighborsClassifier(n_neighbors=3)

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)

draw_confusion_matrix(y_test, y_pred)

display(HTML('<b>With PCA</b>'))

model.fit(X_pca_train_scaled, y_train)

y_pca_pred = model.predict(X_pca_test_scaled)

draw_confusion_matrix(y_test, y_pca_pred)
```

Accuracy = 99.84

Precision = 99.80

Recall = 98.33



With PCA

Accuracy = 99.80

Precision = 99.44

Recall = 98.13

Out[87]:
```
(0.9979666666666667, 0.9943707277844793, 0.9813492063492063)
```

```
time: 3.19 s
```

3. Support Vector Machine

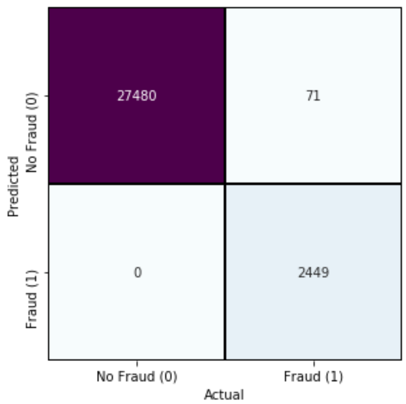
In [89]:

```
model = SVC()

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)

draw_confusion_matrix(y_test, y_pred)

display(HTML('<b>With PCA</b>'))

model.fit(X_pca_train_scaled, y_train)

y_pca_pred = model.predict(X_pca_test_scaled)

draw_confusion_matrix(y_test, y_pca_pred)
```

Accuracy = 99.76

Precision = 100.00

Recall = 97.18

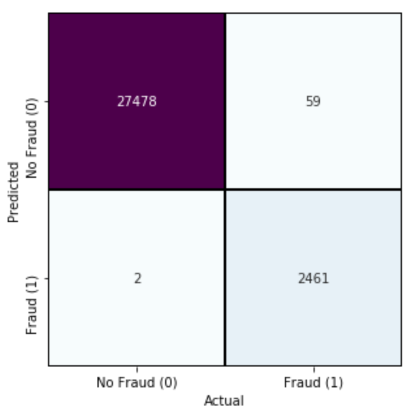


With PCA

Accuracy = 99.80

Precision = 99.92

Recall = 97.66



Out[89]: (0.9979666666666667, 0.999187982135607, 0.9765873015873016)

```
time: 5.51 s
```

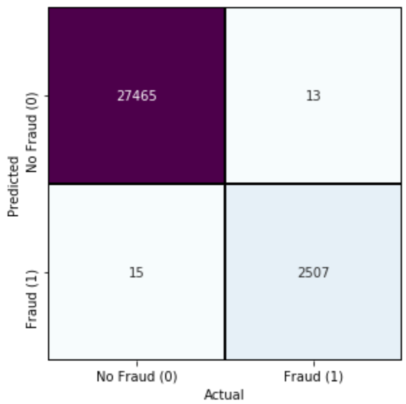
4. Decision Tree

In [90]:
```
model = DecisionTreeClassifier()

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)

draw_confusion_matrix(y_test, y_pred)

display(HTML('<b>With PCA</b>'))

model.fit(X_pca_train_scaled, y_train)

y_pca_pred = model.predict(X_pca_test_scaled)

draw_confusion_matrix(y_test, y_pca_pred)
```

Accuracy = 99.91

Precision = 99.41

Recall = 99.48



With PCA

Accuracy = 99.69

Precision = 98.10

Recall = 98.17

Out[90]:
```
(0.9968666666666667, 0.9809674861221253, 0.9817460317460317)
```

```
time: 946 ms
```

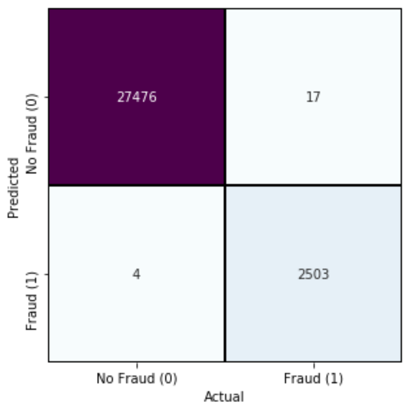
5. Bagging (Boosting Aggregations)

In [91]:
```
model = BaggingClassifier()

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)

draw_confusion_matrix(y_test, y_pred)

display(HTML('<b>With PCA</b>'))

model.fit(X_pca_train_scaled, y_train)

y_pca_pred = model.predict(X_pca_test_scaled)

draw_confusion_matrix(y_test, y_pca_pred)

```

Accuracy = 99.93

Precision = 99.84

Recall = 99.33

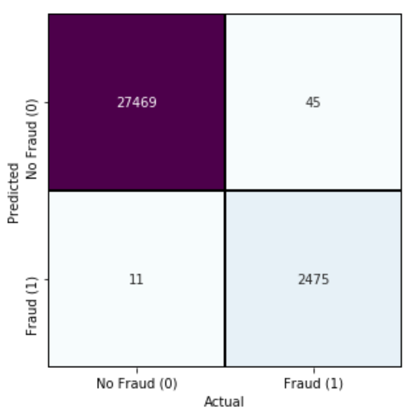


With PCA

Accuracy = 99.81

Precision = 99.56

Recall = 98.21



Out[91]:
```
(0.9981333333333333, 0.995575221238938, 0.9821428571428571)
time: 4.85 s
```

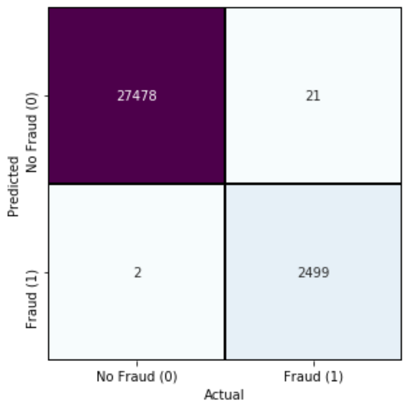
6. Random Forest

In [92]:
```
model = RandomForestClassifier()

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)

draw_confusion_matrix(y_test, y_pred)

display(HTML('<b>With PCA</b>'))

model.fit(X_pca_train_scaled, y_train)

y_pca_pred = model.predict(X_pca_test_scaled)

draw_confusion_matrix(y_test, y_pca_pred)
```

Accuracy = 99.92

Precision = 99.92

Recall = 99.17



With PCA

Accuracy = 99.80

Precision = 99.64

Recall = 97.94

Out[92]: (0.9979666666666667, 0.9963665724666936, 0.9793650793650793)

time: 1.56 s

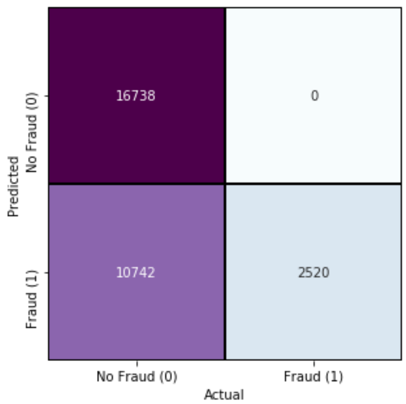
7. Guassian Naiive Baye's

In [93]:
```
model = GaussianNB()

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)

draw_confusion_matrix(y_test, y_pred)

display(HTML('<b>With PCA</b>'))

model.fit(X_pca_train_scaled, y_train)

y_pca_pred = model.predict(X_pca_test_scaled)

draw_confusion_matrix(y_test, y_pca_pred)
```

Accuracy = 64.19

Precision = 19.00

Recall = 100.00

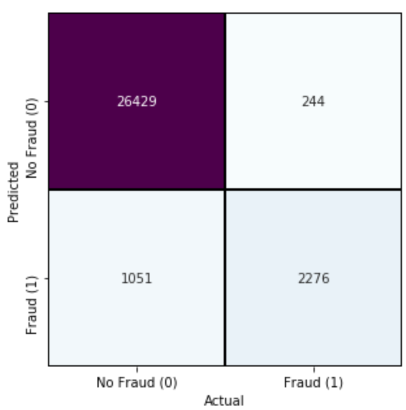


With PCA

Accuracy = 95.68

Precision = 68.41

Recall = 90.32



Out[93]: (0.9568333333333333, 0.6840997896002404, 0.9031746031746032)

time: 224 ms

8. Multinomial Naiive Baye's

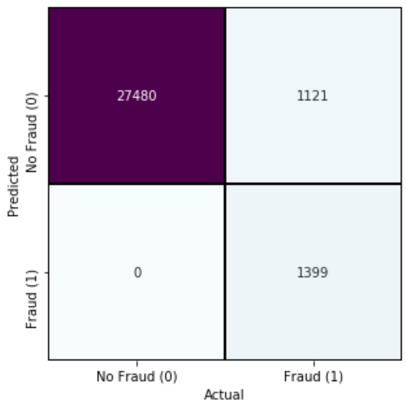
In [95]:

```
model = MultinomialNB()

model.fit(X_train_mm, y_train)

y_pred = model.predict(X_test_mm)

draw_confusion_matrix(y_test, y_pred)

display(HTML('<b>With PCA</b>'))

model.fit(X_pca_train_mm, y_train)

y_pca_pred = model.predict(X_pca_test_mm)

draw_confusion_matrix(y_test, y_pca_pred)
```

Accuracy = 96.26

Precision = 100.00

Recall = 55.52



With PCA

Accuracy = 91.60

Precision = 0.00

Recall = 0.00

Out[95]:

```
(0.916, 0.0, 0.0)

time: 234 ms
```

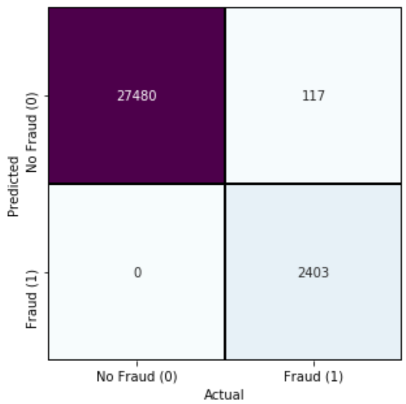
9. Bernoulli Naiive Baye's

In [96]:
```
model = BernoulliNB()

model.fit(X_train_scaled, y_train)

y_pred = model.predict(X_test_scaled)

draw_confusion_matrix(y_test, y_pred)

display(HTML('<b>With PCA</b>'))

model.fit(X_pca_train_scaled, y_train)

y_pca_pred = model.predict(X_pca_test_scaled)

draw_confusion_matrix(y_test, y_pca_pred)
```

Accuracy = 99.61

Precision = 100.00

Recall = 95.36

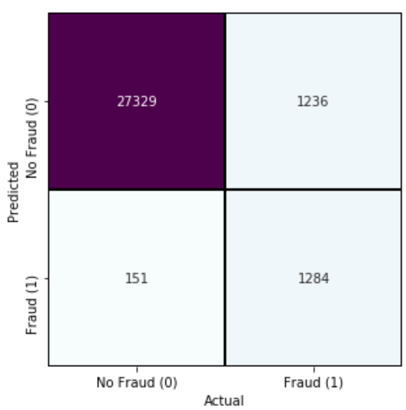


With PCA

Accuracy = 95.38

Precision = 89.48

Recall = 50.95



Out[96]:
```
(0.9537666666666667, 0.894773519163763, 0.5095238095238095)
```

```
time: 306 ms
```

Classification Conclusion

It appears that in my case, **Pricinpal Component Analysis** almost always performed worst than normally scaled data.
Best performing Classifier was **Bagging (Boosting Aggregation) Classifier** with accuracy of **99.93%**
Worst performing Classifier was **Guassian Naiive Baye's** with accuracy of **64.19%**

10. Linear Regression

For Linear Regression, we need a continuous value as label attribute. For this purpose, we will pick ***amount*** as **labeled** attirbtue.

In [54]:
```
df_txns_d = pd.read_pickle('df_txns_d.pkl')

y = df_txns_d.pop('amount')
X = df_txns_d
```

```
time: 11.2 ms
```

In [55]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1, test_size = 0.3)

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)

```

```
(70000, 11)
(30000, 11)
(70000,)
(30000,)
time: 11.8 ms
```

In [59]:
```
lr = LinearRegression()
lr.fit(X_train, y_train)

print(lr.intercept_)
list(zip(X_train.columns.values, lr.coef_))
```

```
202830.01613722602
```

Out[59]:
```
[('step', 46.94401197097731),
 ('oldbalanceOrg', 0.9401305822278108),
 ('newbalanceOrig', -0.9415730607357066),
 ('oldbalanceDest', -0.07166328667110494),
 ('newbalanceDest', 0.08185511218513569),
 ('isFraud', -342010.0931797835),
 ('CASH_IN', 112757.84263853624),
 ('CASH_OUT', -73918.61823189294),
 ('DEBIT', -220531.175252213),
 ('PAYMENT', -198050.53450234505),
 ('TRANSFER', 379742.4853479149)]
```

```
time: 13.6 ms
```

In [60]:
```
y_pred = lr.predict(X_test)
print(y_pred.shape)
```

```
(30000,)
time: 2.33 ms
```

In [66]:
```
MAE = metrics.mean_squared_error(y_test, y_pred)
MSE = metrics.mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(metrics.mean_squared_error(y_test, y_pred))

display(HTML('<b>Mean Absolute Error</b> (MAE) = {}'.format(MAE)))
display(HTML('<b>Mean Squared Error</b> (MSE) = {}'.format(MSE)))
display(HTML('<b>Root Mean Squared Error</b> (RMSE) = {}'.format(RMSE)))
```

Mean Absolute Error (MAE) = 68989132497.02155

Mean Squared Error (MSE) = 68989132497.02155

Root Mean Squared Error (RMSE) = 262657.82397831127

```
time: 5.61 ms
```

Errors shown above are clearly large, thefore, linear regression does not reveal accurate information about amount of transaction from rest of the transaction data.