Digital Finance Fraud Detection

1. Import Libraries

In [1]:
```
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
from sklearn.model_selection import train_test_split, learning_curve
from sklearn.metrics import average_precision_score
from xgboost.sklearn import XGBClassifier
from xgboost import plot_importance, to_graphviz
```

In [2]:
```
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

Import data and correct spelling of original column headers for consistency

In [3]:
```
df = pd.read_csv('../input/PS_20174392719_1491204439457_log.csv')
df = df.rename(columns={'oldbalanceOrg':'oldBalanceOrig', 'newbalanceOrig':'newBalance
                        'oldbalanceDest':'oldBalanceDest', 'newbalanceDest':'newBalanc
print(df.head())
```

```
   step      type    amount     nameOrig  oldBalanceOrig  newBalanceOrig  \
0     1   PAYMENT   9839.64  C1231006815        170136.0       160296.36
1     1   PAYMENT   1864.28  C1666544295         21249.0        19384.72
2     1  TRANSFER    181.00  C1305486145           181.0            0.00
3     1  CASH_OUT    181.00   C840083671           181.0            0.00
4     1   PAYMENT  11668.14  C2048537720         41554.0        29885.86

      nameDest  oldBalanceDest  newBalanceDest  isFraud  isFlaggedFraud
0  M1979787155             0.0             0.0        0               0
1  M2044282225             0.0             0.0        0               0
2   C553264065             0.0             0.0        1               0
3    C38997010         21182.0             0.0        1               0
4  M1230701703             0.0             0.0        0               0
```

Test if there are any missing values in the DataFrame. It turns out there are no obvious missing values, but, as we will see below, this does not rule out the presence of proxies indicated by a numerical value like 0.

In [4]:
```
df.isnull().values.any()
```

Out[4]:
```
False
```

2. Exploratory Data Analysis

From this section until section 4, we exclusively manipulate the data using DataFrame methods. This approach provides a succinct means to gain insights into the dataset. Subsequent sections will present more elaborate visualizations.

2.1. Which types of transactions are fraudulent?

We observe that among the five types of transactions, fraud is identified only in two: 'TRANSFER,' involving money sent to a customer/fraudster, and 'CASH_OUT,' where funds are transferred to a merchant who then pays the customer/fraudster in cash. Remarkably, the quantity of fraudulent TRANSFERs nearly matches that of fraudulent CASH_OUTs.

In [5]:
```
print('\n The types of fraudulent transactions are {}'.format(\
list(df.loc[df.isFraud == 1].type.drop_duplicates().values))) # only 'CASH_OUT'
                                                              # & 'TRANSFER'

dfFraudTransfer = df.loc[(df.isFraud == 1) & (df.type == 'TRANSFER')]
dfFraudCashout = df.loc[(df.isFraud == 1) & (df.type == 'CASH_OUT')]

print ('\n The number of fraudulent TRANSFERs = {}'.\
       format(len(dfFraudTransfer))) # 4097

print ('\n The number of fraudulent CASH_OUTs = {}'.\
       format(len(dfFraudCashout))) # 4116
```

```
 The types of fraudulent transactions are ['TRANSFER', 'CASH_OUT']

 The number of fraudulent TRANSFERs = 4097

 The number of fraudulent CASH_OUTs = 4116
```

2.2. What determines whether the feature *isFlaggedFraud* gets set or not?

It appears that the origin of the isFlaggedFraud feature is unclear, which contrasts with the provided description. Out of 6 million entries, only 16 instances trigger the isFlaggedFraud feature, and these occurrences do not seem to correlate with any explanatory variables.

The data describes isFlaggedFraud as being triggered when an attempt is made to 'TRANSFER' an 'amount' greater than 200,000. However, as demonstrated below, isFlaggedFraud can remain inactive despite this condition being met.

In [6]:
```
print('\nThe type of transactions in which isFlaggedFraud is set: \
{}'.format(list(df.loc[df.isFlaggedFraud == 1].type.drop_duplicates())))
                                                              # only 'TRANSFER'

dfTransfer = df.loc[df.type == 'TRANSFER']
dfFlagged = df.loc[df.isFlaggedFraud == 1]
dfNotFlagged = df.loc[df.isFlaggedFraud == 0]

print('\nMin amount transacted when isFlaggedFraud is set= {}'\
                                    .format(dfFlagged.amount.min())) # 353874.22

print('\nMax amount transacted in a TRANSFER where isFlaggedFraud is not set=\
 {}'.format(dfTransfer.loc[dfTransfer.isFlaggedFraud == 0].amount.max())) # 92445516.6
```

```
The type of transactions in which isFlaggedFraud is set: ['TRANSFER']

Min amount transacted when isFlaggedFraud is set= 353874.22

Max amount transacted in a TRANSFER where isFlaggedFraud is not set= 92445516.64
```

Can the values of oldBalanceDest and newBalanceDest determine the activation of isFlaggedFraud?

In every TRANSFER where isFlaggedFraud is activated, the old balance matches the new balance in both the origin and destination accounts. This parity is presumably due to the transaction being halted.

Notably, oldBalanceDest equals 0 in every such transaction. However, as demonstrated below, the presence of isFlaggedFraud is not solely determined by oldBalanceDest and newBalanceDest being 0 in TRANSFERS, as these conditions do not consistently indicate the activation of isFlaggedFraud.

In [7]:
```
print('\nThe number of TRANSFERs where isFlaggedFraud = 0, yet oldBalanceDest = 0 and\
 newBalanceDest = 0: {}'.\
format(len(dfTransfer.loc[(dfTransfer.isFlaggedFraud == 0) & \
(dfTransfer.oldBalanceDest == 0) & (dfTransfer.newBalanceDest == 0)])))  # 4158
```

```
The number of TRANSFERs where isFlaggedFraud = 0, yet oldBalanceDest = 0 and newBalan
ceDest = 0: 4158
```

The activation of isFlaggedFraud cannot be determined by thresholding oldBalanceOrig due to the overlapping range of values. This range overlaps with the values found in TRANSFERs where isFlaggedFraud is not activated (as shown below). It's important to note that newBalanceOrig need not be considered since it is updated only after the transaction, while isFlaggedFraud would be activated before the transaction occurs.

In [8]:
```
print('\nMin, Max of oldBalanceOrig for isFlaggedFraud = 1 TRANSFERs: {}'.\
format([round(dfFlagged.oldBalanceOrig.min()), round(dfFlagged.oldBalanceOrig.max())])

print('\nMin, Max of oldBalanceOrig for isFlaggedFraud = 0 TRANSFERs where \
oldBalanceOrig = \
newBalanceOrig: {}'.format(\
[dfTransfer.loc[(dfTransfer.isFlaggedFraud == 0) & (dfTransfer.oldBalanceOrig \
== dfTransfer.newBalanceOrig)].oldBalanceOrig.min(), \
round(dfTransfer.loc[(dfTransfer.isFlaggedFraud == 0) & (dfTransfer.oldBalanceOrig \
               == dfTransfer.newBalanceOrig)].oldBalanceOrig.max())]))
```

```
Min, Max of oldBalanceOrig for isFlaggedFraud = 1 TRANSFERs: [353874.0, 19585040.0]

Min, Max of oldBalanceOrig for isFlaggedFraud = 0 TRANSFERs where oldBalanceOrig = ne
wBalanceOrig: [0.0, 575668.0]
```

Can the activation of isFlaggedFraud be influenced by customers transacting more than once? Notably, there are no duplicate customer names within transactions where isFlaggedFraud is activated. However, transactions without isFlaggedFraud exhibit duplicate customer names.

Interestingly, it's revealed that the originators of transactions triggering isFlaggedFraud have transacted only once. Additionally, a small number of destination accounts involved in

transactions with isFlaggedFraud activated have engaged in multiple transactions.

In [9]:
```
print('\nHave originators of transactions flagged as fraud transacted more than \
once? {}'\
.format((dfFlagged.nameOrig.isin(pd.concat([dfNotFlagged.nameOrig, \
                                    dfNotFlagged.nameDest])))).any())) # False

print('\nHave destinations for transactions flagged as fraud initiated\
 other transactions? \
{}'.format((dfFlagged.nameDest.isin(dfNotFlagged.nameOrig)).any())) # False

# Since only 2 destination accounts of 16 that have 'isFlaggedFraud' set have been
# destination accounts more than once,
# clearly 'isFlaggedFraud' being set is independent of whether a
# destination account has been used before or not

print('\nHow many destination accounts of transactions flagged as fraud have been \
destination accounts more than once?: {}'\
.format(sum(dfFlagged.nameDest.isin(dfNotFlagged.nameDest)))) # 2
```

```
Have originators of transactions flagged as fraud transacted more than once? False

Have destinations for transactions flagged as fraud initiated other transactions? Fal
se

How many destination accounts of transactions flagged as fraud have been destination
accounts more than once?: 2
```

It's evident that transactions marked with isFlaggedFraud occur across all values of step, much like the complementary set of transactions. Hence, isFlaggedFraud shows no correlation with step and therefore appears to be unrelated to any explanatory variable or feature within the dataset.

Conclusion: While isFraud is consistently activated when isFlaggedFraud is set, given the seemingly random and infrequent nature of isFlaggedFraud being triggered just 16 times, this feature can be deemed insignificant. Therefore, it can be safely discarded from the dataset without compromising the information contained within.

2.3. Are expected merchant accounts accordingly labelled?

It was previously indicated that CASH_IN transactions entail customers being paid by merchants whose names are prefixed by 'M'. However, upon examination of the current data, there is no evidence of merchants conducting CASH_IN transactions directed towards customers.

In [10]:
```
print('\nAre there any merchants among originator accounts for CASH_IN \
transactions? {}'.format(\
(df.loc[df.type == 'CASH_IN'].nameOrig.str.contains('M')).any())) # False
```

```
Are there any merchants among originator accounts for CASH_IN transactions? False
```

Likewise, it was mentioned that CASH_OUT transactions involve paying a merchant. However, upon inspection of CASH_OUT transactions, there are no merchants listed among the destination accounts.

In [11]:
```
print('\nAre there any merchants among destination accounts for CASH_OUT \
transactions? {}'.format(\
(df.loc[df.type == 'CASH_OUT'].nameDest.str.contains('M')).any())) # False
```

```
Are there any merchants among destination accounts for CASH_OUT transactions? False
```

In fact, there are no merchants listed among any originator accounts. Merchants exclusively appear in the destination accounts for all PAYMENTS.

In [12]:
```
print('\nAre there merchants among any originator accounts? {}'.format(\
      df.nameOrig.str.contains('M').any())) # False

print('\nAre there any transactions having merchants among destination accounts\
 other than the PAYMENT type? {}'.format(\
(df.loc[df.nameDest.str.contains('M')].type != 'PAYMENT').any())) # False
```

```
Are there merchants among any originator accounts? False

Are there any transactions having merchants among destination accounts other than the
PAYMENT type? False
```

In conclusion: Across all transactions, the account labels nameOrig and nameDest exhibit an unexpected pattern with the 'M' merchant prefix.

2.4. Are there account labels common to fraudulent TRANSFERs and CASH_OUTs?

As per the data description, the typical modus operandi for committing fraud involves initiating a TRANSFER to a fraudulent account, which subsequently carries out a CASH_OUT. In this process, the fraudulent account serves as both the destination in a TRANSFER and the originator in a CASH_OUT. However, upon examination of the data, it's evident that there is no commonality between accounts in such a manner within fraudulent transactions. Thus, the expected modus operandi is not reflected in the data.

In [13]:
```
print('\nWithin fraudulent transactions, are there destinations for TRANSFERS \
that are also originators for CASH_OUTs? {}'.format(\
(dfFraudTransfer.nameDest.isin(dfFraudCashout.nameOrig)).any())) # False
dfNotFraud = df.loc[df.isFraud == 0]
```

```
Within fraudulent transactions, are there destinations for TRANSFERS that are also or
iginators for CASH_OUTs? False
```

Is it possible for destination accounts involved in fraudulent TRANSFERs to initiate undetected CASHOUTs labeled as genuine? The data reveals the existence of three such accounts.

In [14]:
```
print('\nFraudulent TRANSFERs whose destination accounts are originators of \
genuine CASH_OUTs: \n\n{}'.format(dfFraudTransfer.loc[dfFraudTransfer.nameDest.\
isin(dfNotFraud.loc[dfNotFraud.type == 'CASH_OUT'].nameOrig.drop_duplicates())]))
```

```
Fraudulent TRANSFERs whose destination accounts are originators of genuine CASH_OUTs:

         step      type      amount     nameOrig  oldBalanceOrig  \
1030443    65  TRANSFER  1282971.57  C1175896731      1282971.57
6039814   486  TRANSFER   214793.32  C2140495649       214793.32
6362556   738  TRANSFER   814689.88  C2029041842       814689.88

         newBalanceOrig     nameDest  oldBalanceDest  newBalanceDest  isFraud  \
1030443             0.0  C1714931087             0.0             0.0        1
6039814             0.0   C423543548             0.0             0.0        1
6362556             0.0  C1023330867             0.0             0.0        1

         isFlaggedFraud
1030443               0
6039814               0
6362556               0
```

However, among these three accounts, two of them initially engage in genuine CASH_OUT transactions. Subsequently, as evidenced by the time steps recorded, they receive fraudulent TRANSFERs. This sequence indicates that fraudulent transactions are not solely discernible based on the nameOrig and nameDest features.

In [15]:
```
print('\nFraudulent TRANSFER to C423543548 occured at step = 486 whereas \
genuine CASH_OUT from this account occured earlier at step = {}'.format(\
dfNotFraud.loc[(dfNotFraud.type == 'CASH_OUT') & (dfNotFraud.nameOrig == \
                          'C423543548')].step.values)) # 185
```

```
Fraudulent TRANSFER to C423543548 occured at step = 486 whereas genuine CASH_OUT from
this account occured earlier at step = [185]
```

Considering the observations made in section 2.3 regarding the unexpected encoding of merchant accounts in the nameOrig and nameDest features, we have decided to drop these features from the data due to their lack of meaningful information.

3. Data cleaning

Referring to the exploratory data analysis (EDA) in section 2, we've established that fraud exclusively occurs in 'TRANSFER's and 'CASH_OUT's. Consequently, we've assembled only the relevant data in X for further analysis.

In [16]:
```
X = df.loc[(df.type == 'TRANSFER') | (df.type == 'CASH_OUT')]

randomState = 5
np.random.seed(randomState)

#X = X.loc[np.random.choice(X.index, 100000, replace = False)]

Y = X['isFraud']
del X['isFraud']

# Eliminate columns shown to be irrelevant for analysis in the EDA
X = X.drop(['nameOrig', 'nameDest', 'isFlaggedFraud'], axis = 1)

# Binary-encoding of labelled data in 'type'
X.loc[X.type == 'TRANSFER', 'type'] = 0
```

```
X.loc[X.type == 'CASH_OUT', 'type'] = 1
X.type = X.type.astype(int) # convert dtype('O') to dtype(int)
```

3.1. Imputation of Latent Missing Values

There are numerous transactions within the data where the destination account exhibits zero balances both before and after a non-zero amount is transacted. The proportion of such transactions, where zero likely indicates a missing value, is significantly higher in fraudulent cases (50%) compared to genuine transactions (0.06%).

In [17]:
```
Xfraud = X.loc[Y == 1]
XnonFraud = X.loc[Y == 0]
print('\nThe fraction of fraudulent transactions with \'oldBalanceDest\' = \
\'newBalanceDest\' = 0 although the transacted \'amount\' is non-zero is: {}'.\
format(len(Xfraud.loc[(Xfraud.oldBalanceDest == 0) & \
(Xfraud.newBalanceDest == 0) & (Xfraud.amount)]) / (1.0 * len(Xfraud))))

print('\nThe fraction of genuine transactions with \'oldBalanceDest\' = \
newBalanceDest\' = 0 although the transacted \'amount\' is non-zero is: {}'.\
format(len(XnonFraud.loc[(XnonFraud.oldBalanceDest == 0) & \
(XnonFraud.newBalanceDest == 0) & (XnonFraud.amount)]) / (1.0 * len(XnonFraud))))
```

```
The fraction of fraudulent transactions with 'oldBalanceDest' = 'newBalanceDest' = 0
although the transacted 'amount' is non-zero is: 0.4955558261293072

The fraction of genuine transactions with 'oldBalanceDest' = newBalanceDest' = 0 alth
ough the transacted 'amount' is non-zero is: 0.0006176245277308345
```

As zero destination account balances strongly indicate the possibility of fraud, we refrain from imputing the account balance (prior to the transaction) using statistics or distribution-based methods with an adjustment for the transaction amount. Employing such techniques could obscure this fraud indicator and potentially misclassify fraudulent transactions as genuine. Therefore, we opt to replace the value of 0 with -1, which will be more beneficial for an appropriate machine-learning (ML) algorithm designed to detect fraud.

In [18]:
```
X.loc[(X.oldBalanceDest == 0) & (X.newBalanceDest == 0) & (X.amount != 0), \
      ['oldBalanceDest', 'newBalanceDest']] = - 1
```

Similarly, the data contains multiple transactions where the originating account showcases zero balances both before and after a non-zero transaction. However, in this scenario, the proportion of such transactions is notably smaller in fraudulent cases (0.3%) compared to genuine transactions (47%). Following a similar rationale to the previous case, rather than imputing a numerical value, we opt to replace the value of 0 with a null value.

In [19]:
```
X.loc[(X.oldBalanceOrig == 0) & (X.newBalanceOrig == 0) & (X.amount != 0), \
      ['oldBalanceOrig', 'newBalanceOrig']] = np.nan
```

4. Feature-engineering

Building upon the possibility of zero balances as a differentiator between fraudulent and genuine transactions, we extend the data-imputation process described in section 3.1. This

extension involves creating two new features (columns) to record errors in both the originating and destination accounts for each transaction. These newly created features prove to be significant in achieving optimal performance from the ML algorithm we ultimately employ.

In [20]:
```
X['errorBalanceOrig'] = X.newBalanceOrig + X.amount - X.oldBalanceOrig
X['errorBalanceDest'] = X.oldBalanceDest + X.amount - X.newBalanceDest
```

5. Data visualization

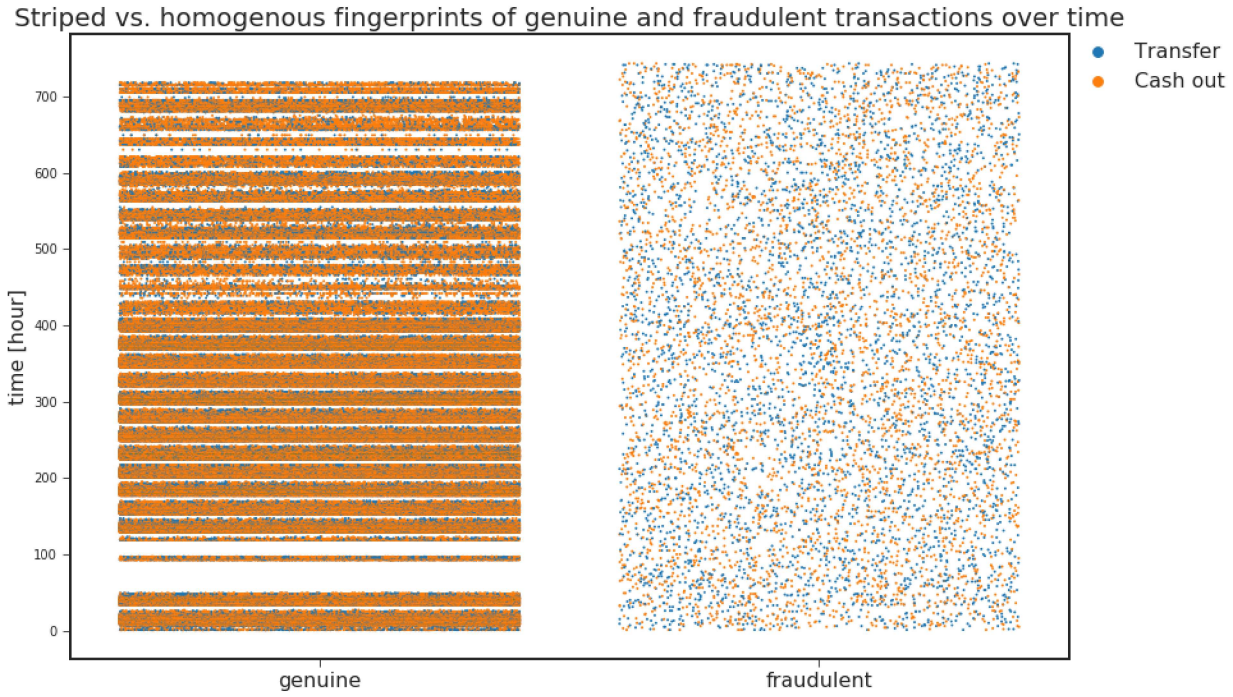
To ascertain whether the data contains ample information for a robust prediction using a ML algorithm, a direct visualization comparing fraudulent and genuine transactions is deemed the most effective method. In line with this principle, I have employed several visualization techniques below to highlight these differences.

In [21]:
```
limit = len(X)

def plotStrip(x, y, hue, figsize = (14, 9)):

    fig = plt.figure(figsize = figsize)
    colours = plt.cm.tab10(np.linspace(0, 1, 9))
    with sns.axes_style('ticks'):
        ax = sns.stripplot(x, y, \
             hue = hue, jitter = 0.4, marker = '.', \
             size = 4, palette = colours)
        ax.set_xlabel('')
        ax.set_xticklabels(['genuine', 'fraudulent'], size = 16)
        for axis in ['top','bottom','left','right']:
            ax.spines[axis].set_linewidth(2)

        handles, labels = ax.get_legend_handles_labels()
        plt.legend(handles, ['Transfer', 'Cash out'], bbox_to_anchor=(1, 1), \
               loc=2, borderaxespad=0, fontsize = 16);
    return ax
```

5. 1. Dispersion over time

The presented plot illustrates how fraudulent and genuine transactions display distinctive patterns when observing their dispersion over time. It's evident that fraudulent transactions are more uniformly distributed over time compared to genuine transactions. Additionally, a notable observation is that in genuine transactions, CASH-OUTs surpass TRANSFERs, unlike the balanced distribution between them observed in fraudulent transactions. It's important to note that the width of each 'fingerprint' in the plot is determined by the 'jitter' parameter in the plotStrip function, aimed at segregating and visualizing transactions occurring simultaneously but at different time points.

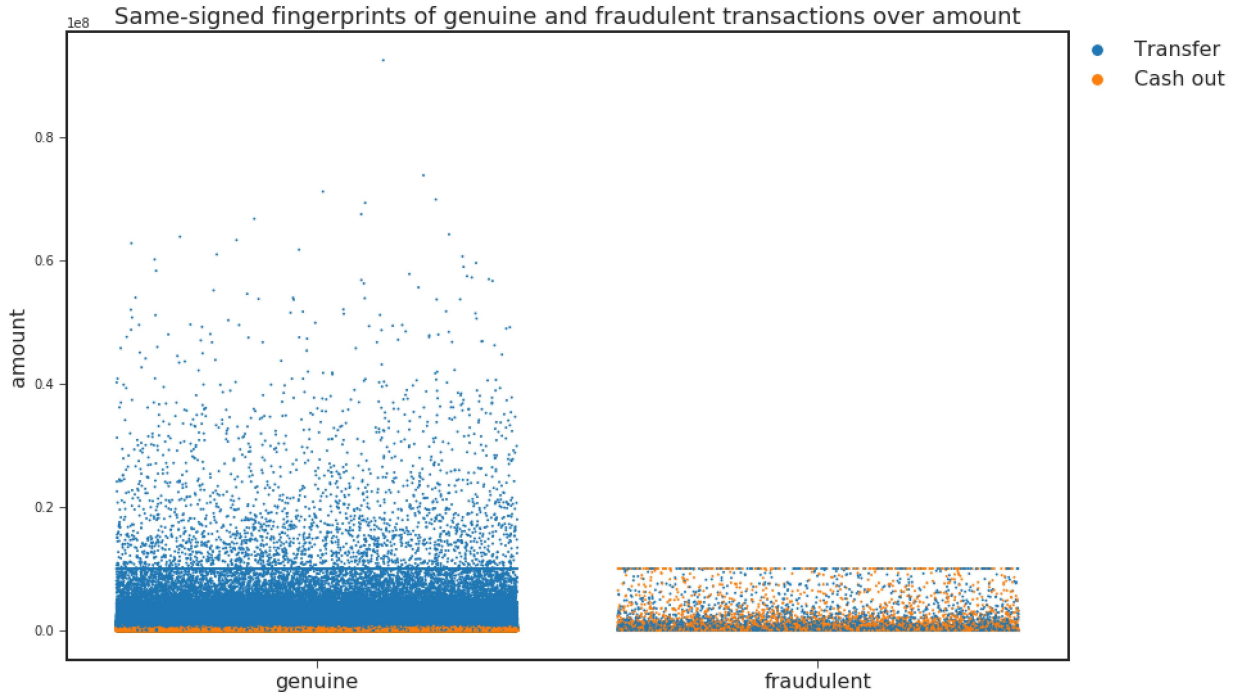
In [22]:
```
ax = plotStrip(Y[:limit], X.step[:limit], X.type[:limit])
ax.set_ylabel('time [hour]', size = 16)
ax.set_title('Striped vs. homogenous fingerprints of genuine and fraudulent \
transactions over time', size = 20);
```



5. 2. Dispersion over amount

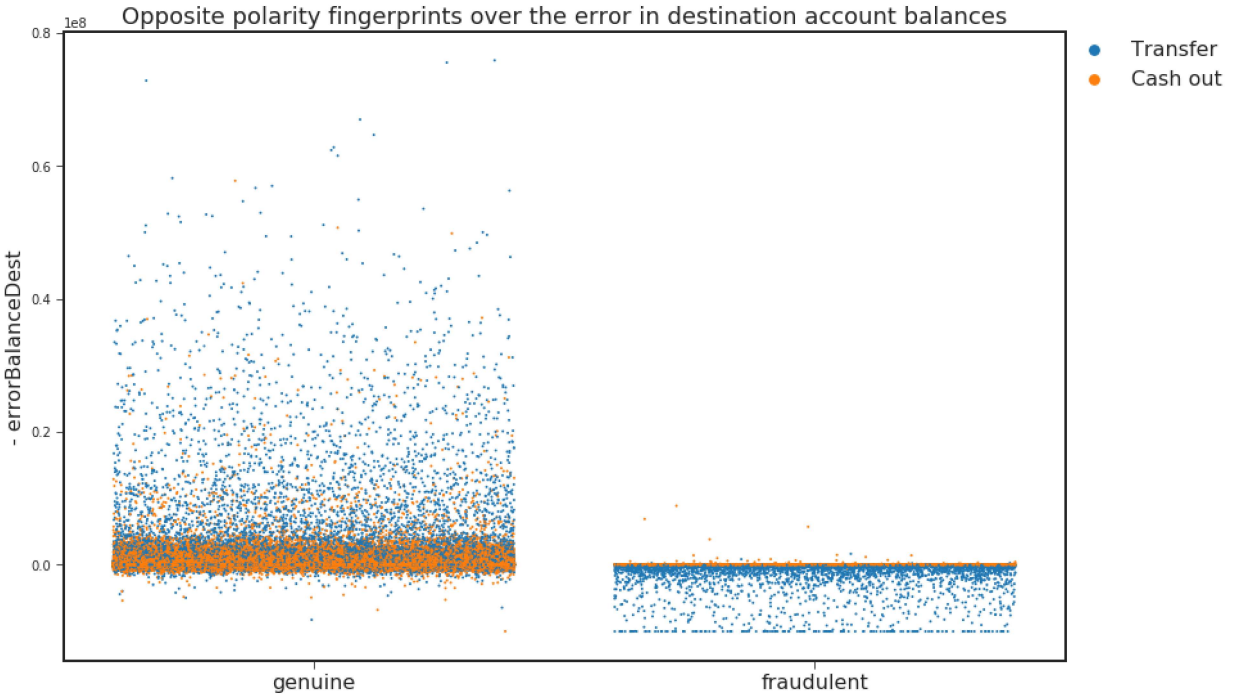
The two plots presented below demonstrate that while the original amount feature can identify the presence of fraud in a transaction, the newly introduced errorBalanceDest feature proves to be more effective in distinguishing fraudulent transactions.

In [23]:
```
limit = len(X)
ax = plotStrip(Y[:limit], X.amount[:limit], X.type[:limit], figsize = (14, 9))
ax.set_ylabel('amount', size = 16)
ax.set_title('Same-signed fingerprints of genuine \
and fraudulent transactions over amount', size = 18);
```



5. 3. Dispersion over error in balance in destination accounts

In [24]:
```
limit = len(X)
ax = plotStrip(Y[:limit], - X.errorBalanceDest[:limit], X.type[:limit], \
               figsize = (14, 9))
ax.set_ylabel('- errorBalanceDest', size = 16)
ax.set_title('Opposite polarity fingerprints over the error in \
destination account balances', size = 18);
```

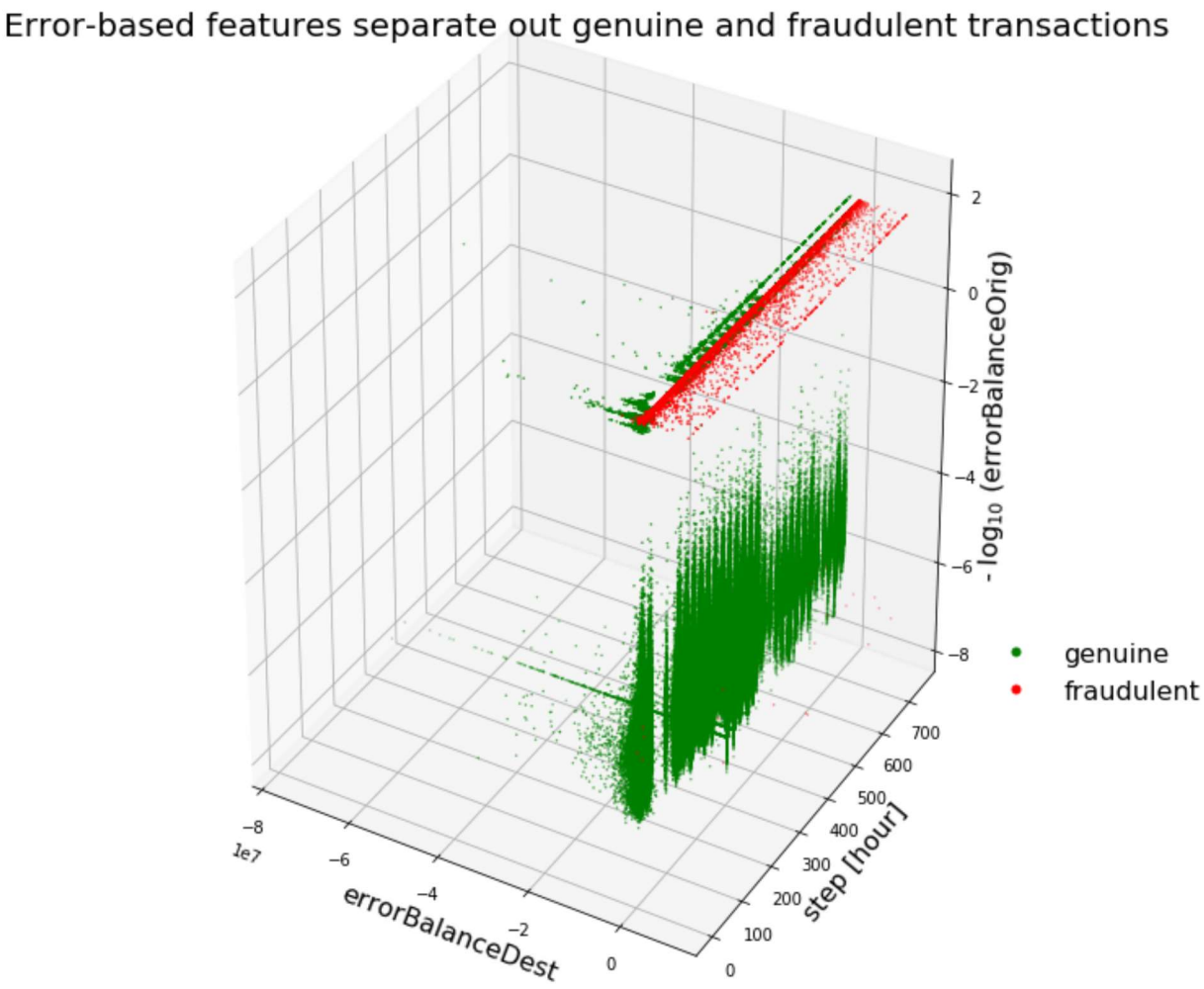


5. 4. Separating out genuine from fraudulent transactions

The 3D plot depicted below provides the most effective distinction between fraudulent and non-fraudulent data by incorporating both engineered error-based features. Evidently, the original step feature shows inefficacy in segregating fraudulent transactions. Additionally, it's notable to observe the striped nature of genuine data over time, as previously anticipated in the figure detailed in section 5.1.

In [25]:
```
# Long computation in this cell (~2.5 minutes)
x = 'errorBalanceDest'
y = 'step'
z = 'errorBalanceOrig'
zOffset = 0.02
limit = len(X)

sns.reset_orig() # prevent seaborn from over-riding mplot3d defaults

fig = plt.figure(figsize = (10, 12))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X.loc[Y == 0, x][:limit], X.loc[Y == 0, y][:limit], \
  -np.log10(X.loc[Y == 0, z][:limit] + zOffset), c = 'g', marker = '.', \
  s = 1, label = 'genuine')

ax.scatter(X.loc[Y == 1, x][:limit], X.loc[Y == 1, y][:limit], \
```

```
  -np.log10(X.loc[Y == 1, z][:limit] + zOffset), c = 'r', marker = '.', \
  s = 1, label = 'fraudulent')

ax.set_xlabel(x, size = 16);
ax.set_ylabel(y + ' [hour]', size = 16);
ax.set_zlabel('- log$_{10}$ (' + z + ')', size = 16)
ax.set_title('Error-based features separate out genuine and fraudulent \
transactions', size = 20)

plt.axis('tight')
ax.grid(1)

noFraudMarker = mlines.Line2D([], [], linewidth = 0, color='g', marker='.',
                          markersize = 10, label='genuine')
fraudMarker = mlines.Line2D([], [], linewidth = 0, color='r', marker='.',
                          markersize = 10, label='fraudulent')

plt.legend(handles = [noFraudMarker, fraudMarker], \
           bbox_to_anchor = (1.20, 0.38 ), frameon = False, prop={'size': 16});
```



5. 5. Fingerprints of genuine and fraudulent transactions

The distinct difference between fraudulent and genuine transactions is prominently showcased through a smoking gun and comprehensive evidence found in their respective correlations, as

observed in the heatmaps below.

In [26]:
```
Xfraud = X.loc[Y == 1] # update Xfraud & XnonFraud with cleaned data
XnonFraud = X.loc[Y == 0]

correlationNonFraud = XnonFraud.loc[:, X.columns != 'step'].corr()
mask = np.zeros_like(correlationNonFraud)
indices = np.triu_indices_from(correlationNonFraud)
mask[indices] = True

grid_kws = {"width_ratios": (.9, .9, .05), "wspace": 0.2}
f, (ax1, ax2, cbar_ax) = plt.subplots(1, 3, gridspec_kw=grid_kws, \
                                      figsize = (14, 9))

cmap = sns.diverging_palette(220, 8, as_cmap=True)
ax1 =sns.heatmap(correlationNonFraud, ax = ax1, vmin = -1, vmax = 1, \
    cmap = cmap, square = False, linewidths = 0.5, mask = mask, cbar = False)
ax1.set_xticklabels(ax1.get_xticklabels(), size = 16);
ax1.set_yticklabels(ax1.get_yticklabels(), size = 16);
ax1.set_title('Genuine \n transactions', size = 20)

correlationFraud = Xfraud.loc[:, X.columns != 'step'].corr()
ax2 = sns.heatmap(correlationFraud, vmin = -1, vmax = 1, cmap = cmap, \
 ax = ax2, square = False, linewidths = 0.5, mask = mask, yticklabels = False, \
    cbar_ax = cbar_ax, cbar_kws={'orientation': 'vertical', \
                                 'ticks': [-1, -0.5, 0, 0.5, 1]})
ax2.set_xticklabels(ax2.get_xticklabels(), size = 16);
ax2.set_title('Fraudulent \n transactions', size = 20);

cbar_ax.set_yticklabels(cbar_ax.get_yticklabels(), size = 14);
```

6. Machine Learning to Detect Fraud in Skewed Data

Following the evidence obtained from the aforementioned plots, it's apparent that the data now includes features enabling the clear detection of fraudulent transactions. However, a significant hurdle remains for training a robust ML model, primarily stemming from the highly imbalanced nature of the data.

In [27]:
```
print('skew = {}'.format( len(Xfraud) / float(len(X)) ))
```

```
skew = 0.002964544224336551
```

Considering the highly skewed nature of the data, I opt to use the area under the precision-recall curve (AUPRC) instead of the conventional area under the receiver operating characteristic (AUROC). The AUPRC is preferred as it offers greater sensitivity to variations between algorithms and their parameter settings compared to the AUROC.

Addressing the imbalanced nature of the data, various methods were explored, including undersampling the majority class and oversampling the minority class using techniques such as SMOTE from the 'imblearn' library. However, it was found that the best results were achieved using an ML algorithm based on ensembles of decision trees, inherently suited for imbalanced data.

This approach not only accommodates missing values within the data but also leverages parallel processing for speed. Among the algorithms tested, the extreme gradient-boosted (XGBoost) algorithm marginally outperformed random forest. Notably, XGBoost and similar algorithms provide the flexibility to assign more weight to the positive class, enabling adjustment for the data skew.

Split the data into training and test sets in a 80:20 ratio

In [28]:
```
trainX, testX, trainY, testY = train_test_split(X, Y, test_size = 0.2, \
                                                random_state = randomState)
```

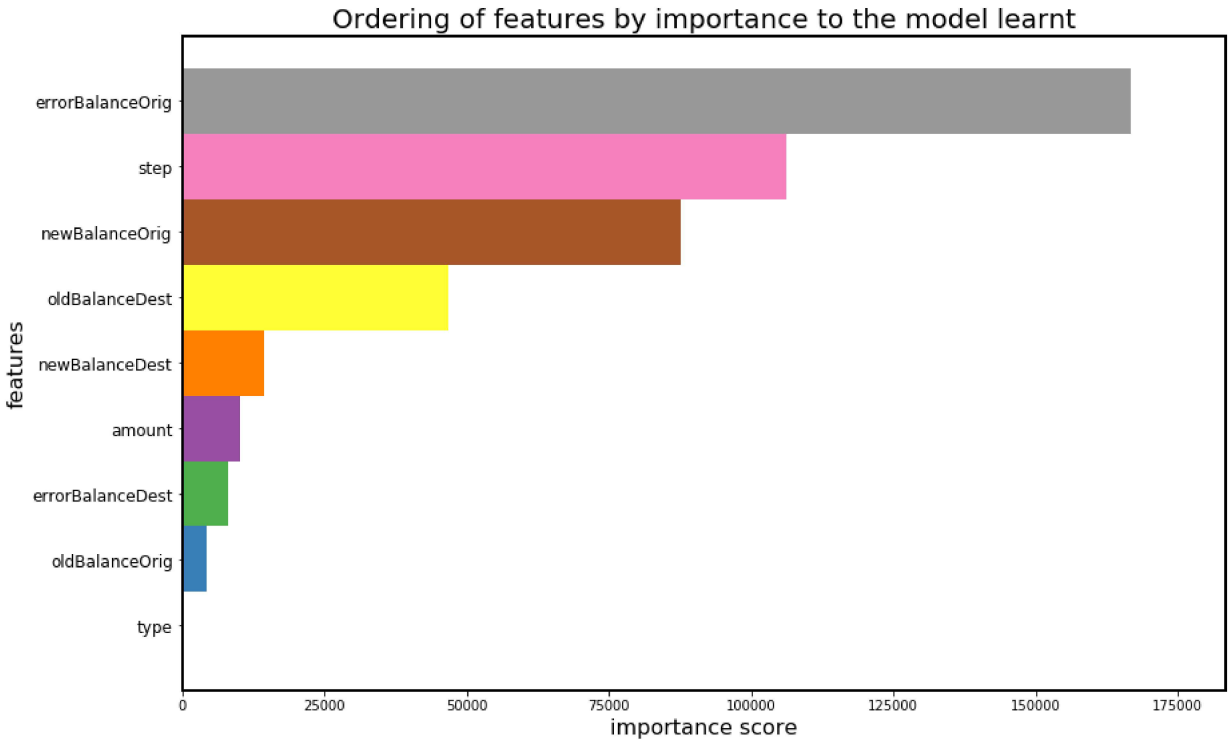
In [29]:
```
# Long computation in this cell (~1.8 minutes)
weights = (Y == 0).sum() / (1.0 * (Y == 1).sum())
clf = XGBClassifier(max_depth = 3, scale_pos_weight = weights, \
                    n_jobs = 4)
probabilities = clf.fit(trainX, trainY).predict_proba(testX)
print('AUPRC = {}'.format(average_precision_score(testY, \
                                                  probabilities[:, 1])))
```

```
AUPRC = 0.9986361116985445
```

6.1. What are the important features for the ML model?

The depicted figure below illustrates the significance of the newly created feature, errorBalanceOrig, in our model. The features are arranged based on the count of samples affected by splits on each feature, demonstrating that errorBalanceOrig stands out as the most influential feature for the model.

In [30]:
```
fig = plt.figure(figsize = (14, 9))
ax = fig.add_subplot(111)

colours = plt.cm.Set1(np.linspace(0, 1, 9))

ax = plot_importance(clf, height = 1, color = colours, grid = False, \
                     show_values = False, importance_type = 'cover', ax = ax);
for axis in ['top','bottom','left','right']:
            ax.spines[axis].set_linewidth(2)

ax.set_xlabel('importance score', size = 16);
ax.set_ylabel('features', size = 16);
ax.set_yticklabels(ax.get_yticklabels(), size = 12);
ax.set_title('Ordering of features by importance to the model learnt', size = 20);
```

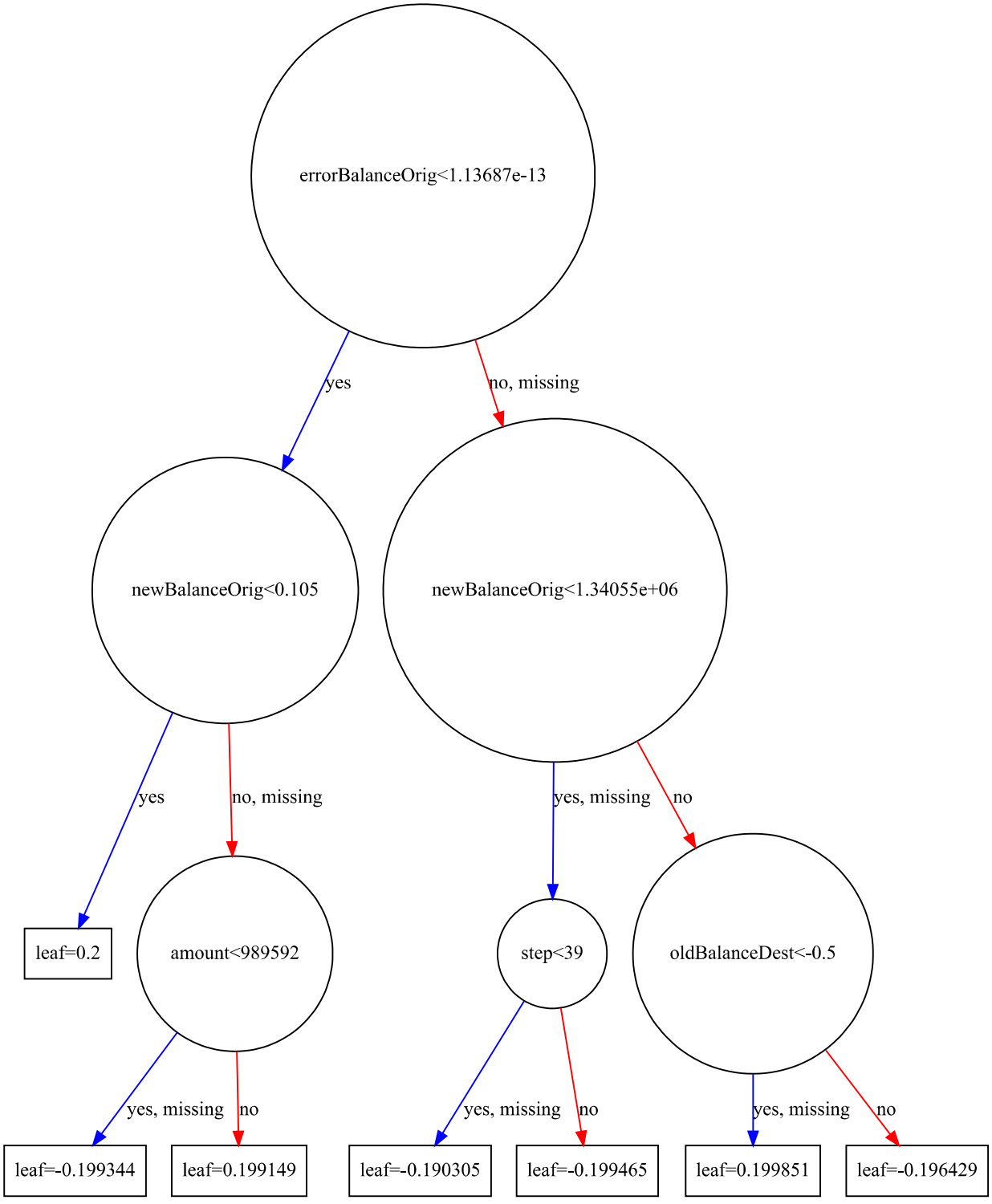


6.2. Visualization of ML model

As anticipated due to its significant influence on the model, the root node in the displayed decision tree below is indeed represented by the feature errorBalanceOrig.

In [31]:
```
to_graphviz(clf)
```

Out[31]:

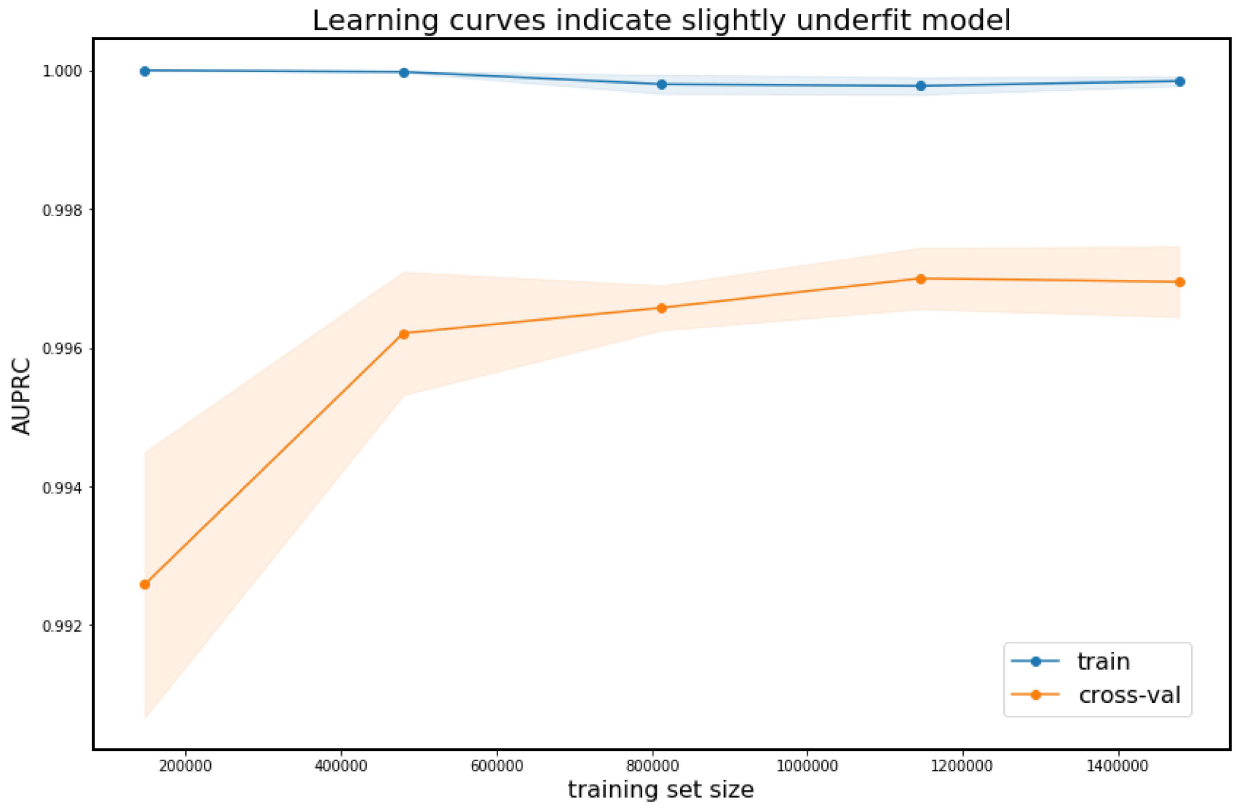


6.3. Bias-variance tradeoff

The model we've developed shows a certain degree of bias and slight underfitting, as highlighted by the plateau observed in the AUPRC while increasing the size of the training set in the cross-validation curve displayed below. To further enhance the model's performance, a straightforward method involves increasing the max_depth parameter of the XGBClassifier. This adjustment may lead to longer training times but is likely to improve performance. Additionally, tweaking other parameters of the classifier such as reducing min_child_weight and lowering reg_lambda can help address the effects of this mild underfitting.

In [32]:
```
# Long computation in this cell (~6 minutes)

trainSizes, trainScores, crossValScores = learning_curve(\
XGBClassifier(max_depth = 3, scale_pos_weight = weights, n_jobs = 4), trainX,\
                                          trainY, scoring = 'average_precision')
```

In [33]:
```
trainScoresMean = np.mean(trainScores, axis=1)
trainScoresStd = np.std(trainScores, axis=1)
crossValScoresMean = np.mean(crossValScores, axis=1)
crossValScoresStd = np.std(crossValScores, axis=1)

colours = plt.cm.tab10(np.linspace(0, 1, 9))

fig = plt.figure(figsize = (14, 9))
plt.fill_between(trainSizes, trainScoresMean - trainScoresStd,
    trainScoresMean + trainScoresStd, alpha=0.1, color=colours[0])
plt.fill_between(trainSizes, crossValScoresMean - crossValScoresStd,
    crossValScoresMean + crossValScoresStd, alpha=0.1, color=colours[1])
plt.plot(trainSizes, trainScores.mean(axis = 1), 'o-', label = 'train', \
         color = colours[0])
plt.plot(trainSizes, crossValScores.mean(axis = 1), 'o-', label = 'cross-val', \
         color = colours[1])

ax = plt.gca()
for axis in ['top','bottom','left','right']:
    ax.spines[axis].set_linewidth(2)

handles, labels = ax.get_legend_handles_labels()
plt.legend(handles, ['train', 'cross-val'], bbox_to_anchor=(0.8, 0.15), \
               loc=2, borderaxespad=0, fontsize = 16);
plt.xlabel('training set size', size = 16);
plt.ylabel('AUPRC', size = 16)
plt.title('Learning curves indicate slightly underfit model', size = 20);
```



7. Conclusion

At the beginning of our analysis, we extensively examined the data to identify features that could be eliminated and those that could be meaningfully engineered. The subsequent visualizations provided confirmation that the data could be effectively discriminated with the aid of these new features.

Addressing the significant skew in the data, we selected an appropriate metric and utilized an ML algorithm based on an ensemble of decision trees. This particular algorithm is notably suited for handling strongly imbalanced classes. The approach taken in this analysis could potentially be broadly applicable to a range of similar problems.