Detecting Suspicious Transactions for Anti-Money Laundering

Kyle Greenberg (kkg35), Christopher Chan (cmc455)
ORIE 4741

Objective/Research Question

What are strong indicative factors of money laundering?

Introduction

According to the IMF (International Money Fund), up to 5 percent of global GDP is estimated to be involved in money laundering (~USD 4 trillion). This incredibly large figure comes to show just how pervasive and damaging this criminal activity is on the integrity of global financial systems. In addition to terrorism financing and drug trafficking, money laundering can also involve government corruption and political instability. Being able to correctly identify fraudulent transactions and money laundering could alleviate such pressures on both countries and companies alike. In recent years, criminals have been using increasingly more and more sophisticated techniques of money laundering, making the need to train models that can detect illicit transactions even more vital. Money laundering detection also is made significantly harder by difficulties in the acquisition of sufficient data that is correctly labeled while also covering a sufficient range of transactions. Current anti-money laundering efforts worldwide only help the interception and recovery of an estimated .1% of all laundered money, which leaves an incredible amount of room for growth and improvement. We believe that a model feasibly could be extended to outperform this low percentage. This report describes our analysis of a synthetic dataset generated through a simulation by IBM in order to explore potential methods of aiding money laundering detection.

Description of IBM Anti-Money Laundering Dataset

In the real world, access to financial transaction data is very limited and highly restricted as financial institutions tend to keep their data private. Even when real financial transaction data is made public, due to false positives and false negatives, labeling transactions as laundering or not laundering can be problematic and inaccurate. IBM generated a synthetic dataset which aims to avoid

Filename	Sample Size
HI-Large_Trans.csv	(n = 179,702,229)
HI-Medium_Trans.csv	(n = 31,898,238)
HI-Small_Trans.csv	(n = 5,078,345)
LI-Large_Trans.csv	(n = 176,066,557)
LI-Medium_Trans.csv	(n = 31,251,483)

Table 1. The filename and sample size of the sets provided by IBM.

these problems. This generated model involves data with individuals, companies, and banks that interact with each other in the form of financial transactions. Individuals and companies have accounts and send money through a bank. The data in this dataset is labeled to detect illicit transactions, with a small portion of transactions labeled as laundering, the rest not. The generator models the entire money laundering cycle from placement or the source of illicit funds, layering or mixing the illicit funds into the financial system, and finally integration or the spending of the illicit funds. Since this dataset models the entire laundering cycle, the generator has the foundation necessary to properly label individual transactions as laundering or not laundering. The generated models an entire financial ecosystem allowing for training and analysis to be done on a more holistic set of data, whereas real data oftentimes only includes information coming from a small subset of this financial ecosystem. Therefore, gaining insights that may apply in a more broader sense may be possible.

IBM released 6 datasets divided into two groups of three with group HI having a higher rate of illicit transactions and group LI having a lower rate of illicit transactions. In each group there are 3 datasets of small, medium and large sizes. Alongside the sample sets are 6 text files each containing a set of transactions alongside within the sample sets they correspond to, with labels indicating the money laundering strategy being used for them. A potentially problematic aspect of the dataset is immediately noticeable: The potential for operation on such a large-scale dataset to be too demanding for our current hardware. Given these constraints, out of the 6 datasets, we ultimately decided to use the ‘HI-Small_Trans.csv’ dataset. The fact that there is a higher illicit rate allows for easier training and the smaller size of the dataset requires less computing resources.

Problem Solving Approach / Data Transformation

We first began by converting the ‘HI-Small_Trans.csv’ to the feather file format. The reason we did this was because this file format would help us to significantly reduce load times and allow for overall easier training. Some of the main benefits of using this file format is its lightweight, minimal API which allows for pushing data frames in and out of memory as simple as possible in addition to an extremely high read and write performance. The feather file format

is not designed for long term data storage, but for our purposes of short term storage of data frames for analysis, the feather file format was our best option.

From the ‘HI-Small_Trans.feather’ file we created multiple resulting dataset paths, which are nodes_path, bank_bounds_path, edges_path, ys_path, subgraphs_path as a way to define the dataset being used. We loaded in the data to data frames: ‘nodes_df’, ‘banks_df’, ‘edges_df’, ‘ys_df’, and ‘subgraphs_df’. The dataframe ‘nodes_df’ contains nodes indexed in order by both bank and account, ‘banks_df’ contains the associated lower bounds and upper bound on the index of a node when provided with a bank, ‘edges_df’ contains the edge information for any transaction, with index pointers to its start and end nodes, ‘ys_df’ contains the desired information to predict whether or not a transaction is money laundering, and ‘subgraphs_df’ contains information pertaining to subgraphs. These data frames can be used for our subsequent data analysis.

Next, in order to be able to make the data usable, we created an efficient and optimized search function capable of returning the index of a node.The function takes as input the provided bank account and account number. The function leverages the lower and upper bounds for each bank from ‘banks_df’ to narrow down the search range and minimize the number of comparisons required to find the node index.

For data visualization, we landed on the utilization of the Graphviz python library. This library allows for the easy visualization of graphs of nodes and edges connecting these nodes.

The text file ‘HI-Small_Patterns.txt’ that corresponds to ‘HI-Small_Trans.csv’ file gives us sample data for illicit transactions. In this file there is data corresponding to laundering attempts using any of the eight patterns of laundering. These eight patterns include the graph shapes: bipartite, fan-in, fan-out, cycle, gather-scatter, scatter-gather, stack, and random. Given this data we can potentially analyze common patterns in the shapes of graphs that could give insight into whether or not any given transaction is illicit.

We wrote a function that takes the file path as input and generates a directed graph visualization of illicit transactions based on the data in the file. Using the graphviz library, we created a digraph object with nodes and edges that represent transactions. After extracting relevant transactions, it iterates over them to identify the start and end nodes and adds them to

the visualization. The final result is a graph rendered as a PDF file titled ‘illict_transactions_example_graph.pdf’. This function allowed us to conveniently analyze and visualize patterns common amongst illicit transactions.

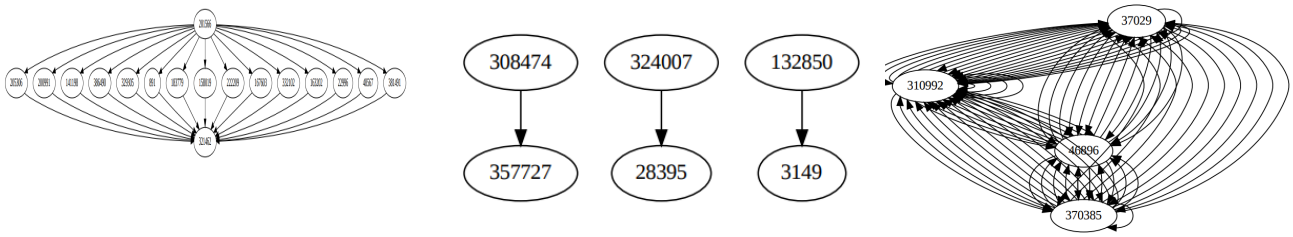

Figure 1. Illicit Transaction Example Graphs

A few of the example graphs are shown above. It can be seen that the shape of these illicit transaction graphs vary greatly in terms of the number of nodes and edges involved, whether or not there are cycles. Some of the illicit transactions above involve only two accounts, whereas others in the file involve dozens. There are a few more common graph shapes, however, this small number of examples is a microcosm to the large complexity and nuance involved in identifying higher proportions of illicit transactions, since many of them do not have consistent traits.

The next task was to create a function capable of splitting these example graphs into subgraphs. The reason for doing this is because subgraphs allow for large complex graphs to be broken down into smaller and more manageable components. In these smaller and more localized graphs, laundering transactions can be more easily identified for a few different reasons. The first reason is community detection, meaning nodes connected or clustered closely together tend to have stronger connections as opposed to nodes many edges away from each other in a graph, secondly, subgraphs can processed much more efficiently, and finally subgraphs can be visualized in a way that is much less confusing to analyze. We defined a function to split graphs into disconnected subgraphs by way of iteratively processing nodes and their transaction partners to identify the relevant subgraphs. We stored these resulting subgraphs for later analysis in the dataframe ‘subgraphs_df’. The function is also able to identify relevant information such as the total number of nodes, total number of subgraphs, and total number of nodes within the

subgraphs. Running the function found that there were 515088 total nodes, those of which were all contained in subgraphs. There were 114139 subgraphs to parse through.

We next calculated the net changes (deltas) in currency balances for each node based on the transactions recorded in the graph. The change in currency balances helps to allow for the detection of unexpected currency related patterns, large transactions in specific currencies, inconsistencies in currency conversions which can all be indications of laundering.

We made a series of final transformations in order to obtain a singular final DataFrame. We created a new DataFrame titled ‘final_df’ which is a copy of ‘edges_df.’ The purpose of this was to have a finalized DataFrame containing all of the information required for analysis and predictions. We converted the ‘Timestamp’ column in ‘edges_df’ DataFrame to numerical values. We added node information to ‘final_df’ and dropped the following unwanted columns from ‘final_df’: 'Payment Currency', 'Receiving Currency', 'Account End', 'Account Start', 'transaction partners End', 'transaction partners Start'. We transformed The 'Payment Format' column in the 'final_df' DataFrame into boolean features by creating a new column for each unique payment format. Each column indicates whether the corresponding payment format is present in the 'Payment Format' column. We added a target variable for ‘Is Laundering’ originally from the ‘ys_df’ DataFrame.

Data Analysis

After transforming our data, we were able to get our dataset ready for analysis through binary classification. Due to the highly unbalanced nature of the dataset, if the classification scoring while training the model was not adjusted to account for the low likelihood of a transaction being a money laundering attempt, then the model would not ever predict that a transaction was money laundering. As such, we train using the balanced accuracy score as our training metric in order to obtain some actual predictions. We wanted to evaluate the performance of our model hoping to obtain a score of around 0.6, a figure that others seem to use as a cutoff for when a model is doing a decent enough job. The way we did this was we fit 100 different decision tree classifiers to samples of 1,000,000 transactions and their data from the dataset and averaged the resulting balanced accuracy score in order to obtain an idea of what sort

of starting score we are looking at for our model. For each of the 100 tests, we utilized a train/test split of 70 to 30 after taking a random subset of transactions from 'final_df'. The resulting average balanced accuracy score that we were able to achieve was 0.603 meaning that on average our model is doing a decent job. This gives us insight into the fact that our model is indeed worth using.

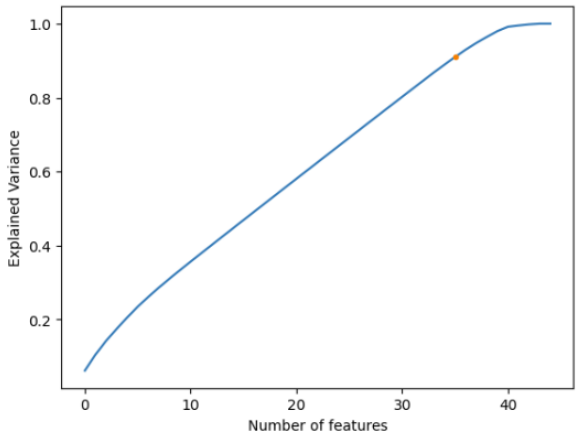

Figure 2. Plot of Expected Variance vs. Number of Features

To further make improvements to our model next we transformed X with PCA or Principal Component Analysis before fitting the decision tree. We attempted to use PCA on standardized features to hopefully see improvements as PCA helps in reducing dimensionality of the feature matrix by reducing irrelevant noise in the data and choosing only the most important features. We then used 5-fold cross validation to obtain a new balanced accuracy score and reduce overfitting. We performed 5-fold cross validation by averaging the balanced accuracy from each of the 5 folds after the model is tested on each fold. 5-fold cross validation helps us to achieve a more robust estimate of the model's performance, mitigating the possible issues relating to variance and bias that may arise from evaluating the model on a single train-test split. The plot on the left shows the cumulative expected variance ratio as a function of the number of features with a threshold of .9, visualizing how PCA worked here to find the principal components.

We achieved with our new model a balanced accuracy score of .598. This gives us continued confidence that continuing to add more features using different techniques will continue to allow us to improve our model for anti money laundering purposes.

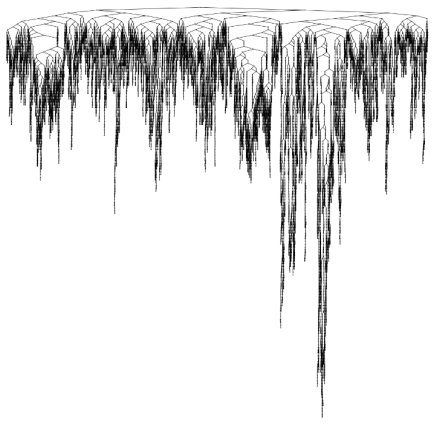

Figure 3. Final Decision Tree

Since we reached the threshold that we have a passable metric of money laundering on this dataset, some next steps to improve further upon our model would be to add detection for specific shapes that are more common such as fan out and fan in detection to see if that yields better results.

Could Our Model Be A Weapon of Math Destruction?

A weapon of math destruction is defined as a predictive model whose outcome is not easily measurable, whose predictions can have negative consequences, and that creates self-fulfilling (or defeating) feedback loops. Our model is unlikely to be a weapon of mass destruction (WMD), but the following discussion should be taken into consideration.

While the dataset used in the model can be separated into train and test examples, it is important to note that the data itself was generated synthetically by IBM. Thus, the model may be affected by inherent biases within the dataset. The generator creates and labels money laundering activity by employing known tactics such as placement, layering, and integration. However, this approach can fail to reflect more recent or less-known laundering methods, making it difficult to validate the model's accuracy.

Although it is unlikely, unauthorized parties could use our model to identify the key factors that label a transaction as illegitimate. This could potentially enable them to develop new tactics to evade detection. Furthermore, repeated detection could motivate parties to find alternative routes that leave no trace, making it challenging for our model to analyze. However, this cannot be considered a self-defeating loop as the coevolution of jurisdiction and crime has always existed. Our aim with this model is to stay ahead of offending parties.

Fairness of Our Model

Fairness in big data is an ever prevalent issue and opens the question of why should algorithm designers think about fairness? The answer is that algorithms bias available information and can have large unintended and often unobservable negative consequences that could be harmful to certain groups of people. Therefore, algorithm designers need to decide what information should be used in the model to minimize or eliminate unfairness.

Fairness is significant to our algorithm as the negative consequences could impartially affect communities of different socioeconomic status. The dataset includes information such as the type of currency of the account and the location of the bank. The amount of illicit drug seizures can differ greatly between communities geographically adjacent to the perpetrators and have a greater chance of being falsely labeled by the model. Since illegal activities often take place in disadvantaged or impoverished areas, the model is more likely to label false positives on such communities. Frequent repeats of this error could negatively affect the person's credit score or create a barrier for impoverished communities.

Conclusion

Money laundering continues to be an incredibly pervasive issue in our current society. Techniques in improving anti-money laundering detection currently have many obstacles and drawbacks. We attempted to train a model utilizing a dataset of synthetic financial transactions to see if we could find a desired result. We attempted to use different analysis techniques such as train-test validate, decision tree binary classification, principal component analysis, and 5-cross validation. We can conclude that our model does produce an acceptable result given that the dataset is highly imbalanced. We were able to, with decent accuracy, classify illicit transactions giving us insight into the fact that our model is worth continuing to add features to for further training.

References

https://www.kaggle.com/datasets/ealtman2019/ibm-transactions-for-anti-money-laundering-aml

https://stephenallwright.com/balanced-accuracy/

https://scikit-learn.org/stable/

Appendix A (Individual Contributions)

Kyle and Christopher both split two main tasks of writing code and performing analysis and writing. Kyle primarily worked more on producing code, while Christopher performed analysis and gathered insights from the code and completed the write up for this final report.

Appendix B (Code)

Load Dataset in Graph Form:

Upload the files located in the Graph_Data folder to the notebook before running!

Code:

Imports, Configuration, Loading the dataset:

```
import pandas as pd
import pyarrow

# constants:
DATASET = 'HI-Small_Trans'
# Reasoning behind dataset choice: Higher illicit rate allows easier
training,
# while smaller size requires less computating resources.

# Paths of data files:
# File organization Path constant:
PATH = './'
# Feather format to improve load times and allow for easier training:
FORMAT = '.feather'
# Resulting dataset paths:
nodes_path = PATH + DATASET + '_nodes' + FORMAT
bank_bounds_path = PATH + DATASET + '_bank_bounds' + FORMAT
edges_path = PATH + DATASET + '_edges' + FORMAT
ys_path = PATH + DATASET + '_ys' + FORMAT
subgraphs_path = PATH + DATASET + '_subgraphs' + FORMAT

import graphviz
import dask.dataframe as dd
# Load Data:
# nodes_df contains nodes indexed in order by both bank and account:
nodes_df =
dd.from_pandas(pd.read_feather(nodes_path),npartitions=32).compute()
# banks_df contains the associated lower bound and upper bound on the
index of a node when provided with a bank:
banks_df =
dd.from_pandas(pd.read_feather(bank_bounds_path).set_index('id'),
npartitions=32).compute()
# edges_df contains the edge information for any transaction, with
index pointers to its start and end nodes
edges_df = dd.from_pandas(pd.read_feather(edges_path),
npartitions=32).compute()
# ys_df contains the desired information to predict, whether or not a
transaction is Money Laundering:
ys_df = dd.from_pandas(pd.read_feather(ys_path),
```

```
npartitions=32).compute()
# subgraphs_df contains the subgraph information:
subgraphs_df = dd.from_pandas(pd.read_feather(subgraphs_path),
npartitions=32).compute()
```

Efficient search capability for the index of a node:

```
def get_node_index(bank, acc):
    # Get inclusive bounds on index:
    [lb, ub] = banks_df.loc[bank][['lb', 'ub']]
    # While lower bound is less than upper bound, the node hasn't been
found:
    while lb < ub:
        # Current index is the average of the lower and upper bound,
rounded down:
        i = round((lb+ub)/2)
        # if current node acc is less than the desired node acc:
        if nodes_df.iloc[i]['Account'] < acc:
            # Then any nodes between lb and i inclusive's acc are less
than acc, so:
            lb = i + 1
        # else if current node matches desired node, return i as node
has been found:
        elif nodes_df.iloc[i]['Account'] == acc:
            return i
        # otherwise current node acc is greater than desired node acc:
        else:
            ub = i - 1
    # Make sure the current node matches the desired node:
    i = int((lb+ub)/2 - (lb+ub)%2)
    if int(nodes_df.iloc[i]['Bank']) == int(bank) and
int(nodes_df.iloc[i]['Account'] == acc):
        return i
    # Throw an error otherwise:
    else:
        print(lb, ub)
        raise Exception('NodeNotFound')
```

Tests for the above function:

```
 import numpy as np
 def run_tests(num_tests=10000):
     for i in
np.random.randint(low=0,high=len(nodes_df),size=num_tests):
         node = nodes_df.iloc[i]
         assert (get_node_index(int(node['Bank']),
str(node['Account'])) == i)
     print('passed', num_tests,'random tests')
 run_tests()
```

```
passed 10000 random tests
```

Generate a visualization of the initial examples provided:

```
 import graphviz

 # Load file containing laundering attempts done under certain known
patterns:
 laundering_attempts = pd.Series([i for i in open('./HI-
Small_Patterns.txt')])
 # Remove unnecessary newline symbols:
 laundering_attempts = laundering_attempts.map(lambda s: s.replace('\
n', ''))
 # Remove empty lines:
 laundering_attempts =
laundering_attempts[laundering_attempts.map(lambda s: s !=
'')].reset_index(drop=True)
 # Format of lines that denote the start of a laundering attempt:
 beginining_format='BEGIN LAUNDERING ATTEMPT'
 # Format of lines that denote the end of a laundering attempt:
 ending_format='END LAUNDERING ATTEMPT'

 # Accumulators:
 beginings = []
 endings = []
 # Get indices of lines denoting starts and ends of laundering
attempts
 for i, v in laundering_attempts.items():
   if beginining_format in v:
     beginings += [i]
   elif ending_format in v:
     endings += [i]
 # Show that length is the same:
 assert len(beginings) == len(endings), 'INCORRECT FORMAT'

 # Organize the lines by which laundering attempts they belong to
 temp = []
 for i in range(len(beginings)):
   start = beginings[i]
   end = endings[i]
   temp += [laundering_attempts.iloc[start:end+1]]
 laundering_attempts = temp

 # Generate a Digraph render of the graph that consists of the
transactions
 # within the file, and right to pdf:
 dot = graphviz.Digraph()
 for laundering_attempt in laundering_attempts:
   transactions = laundering_attempt.iloc[1:-1]
   for transaction in transactions:
     values = transaction.split(',')
     start = str(get_node_index(int(values[1]), values[2]))
```

```
    end = str(get_node_index(int(values[3]), values[4]))
    dot.node(start)
    dot.node(end)
    dot.edge(start, end)
dot.render('illicit_transaction_example_graph',format='pdf')
```

```
(process:1552): GLib-GIO-WARNING **: 13:05:14.129: Unexpectedly, UWP
app `Clipchamp.Clipchamp_2.5.15.0_neutral__yxz26nhyzhsrt' (AUMId
`Clipchamp.Clipchamp_yxz26nhyzhsrt!App') supports 41 extensions but
has no verbs

(process:1552): GLib-GIO-WARNING **: 13:05:14.521: Unexpectedly, UWP
app `Microsoft.ScreenSketch_11.2303.17.0_x64__8wekyb3d8bbwe' (AUMId
`Microsoft.ScreenSketch_8wekyb3d8bbwe!App') supports 29 extensions but
has no verbs

'illicit_transaction_example_graph.pdf'
```

Split into subgraphs, add relevant data:

```
# finding transaction partners of each node:
print('finding transaction partners. This will likely take a couple
minutes.')
transaction_partners = []
for i in range(len(nodes_df)):
    transaction_partners += [set([])]

# Add transaction partners according to the edge
def process_edge(edge, transaction_partners):
    s, e = edge['Start Node'], edge['End Node']
    transaction_partners[s].add(e)
    transaction_partners[e].add(s)

# Make a call for each edge to be processed:
dd.from_pandas(edges_df, npartitions=32).apply(
    lambda e: process_edge(e, transaction_partners),
    meta=(None, 'object'),
    axis=1
).compute()
nodes_df['transaction partners'] = transaction_partners

print(
    'processing nodes_df will require',
    len(nodes_df),
    'iterations. This will likely take a couple minutes.'
)

nodes_df['addable'] = True

def split_graph():
```

```
    # subgraphs is an accumulator array for sets of nodes that make
    # up disconnected subgraphs within the dataset:
    subgraphs = []
    # current index counter:
    i = 0
    # current subgraph initial value:
    subgraph = []
    # horizon for the current subgraph:
    horizon = list()
    # settled nodes for the current subgraph:
    settled = []
    # While there are nodes that are not yet accounted for:
    while True:
        # if no nodes are in the horizon:
        if len(horizon) == 0:
            # if there are nodes in the subgraph:
            if len(subgraph) > 0:
                # add the current subgraph to the list of subgraphs:
                subgraphs += [subgraph.copy()]
            # Make a new list of subgraphs:
            subgraph = []
            # increment check index until the current node is addable
to the horizon:
            while not nodes_df.loc[i,'addable']:
                i += 1
                # if i has exited the index of nodes_df, then all
nodes are not addable,
                # so finish:
                if i == len(nodes_df):
                    return subgraphs
            # add the current node to the list of horizon nodes to
process:
            horizon.append(i)
            # mark the current node as no longer being addable:
            nodes_df.loc[i, 'addable'] = False
        # get the current node:
        curr = horizon.pop()
        # add the current node to the subgraph:
        subgraph += [curr]
        # add all addable transaction partners of the node to the
list of horizon nodes to process
        # and mark them as no longer being addable:
        for partner in nodes_df.loc[curr, 'transaction partners']:
            if nodes_df.loc[partner, 'addable']:
                horizon.append(partner)
                nodes_df.loc[partner,'addable'] = False

subgraphs = split_graph()

print('finished!')
```

```
 print('total nodes:', len(nodes_df))
 print('total subgraphs:', len(subgraphs))
 print(
     'total nodes in subgraphs (should match total nodes):',
     sum([len(subgraph) for subgraph in subgraphs])
 )

 subgraphs = pd.DataFrame({'nodes' : subgraphs})

 print('sorting subgraphs')
 # mapping subgraphs to series:
 subgraphs['nodes'] = subgraphs['nodes'].map(
     lambda ls: pd.Series(ls).sort_values().reset_index(drop=True)
 )

 print('Making changes to nodes')
 subgraph_ids = subgraphs['nodes'].explode()
 temp = pd.Series(subgraph_ids.index.tolist(), index =
subgraph_ids.values.tolist())
 subgraph_ids = pd.Series(nodes_df.index.tolist())
 nodes_df['subgraph'] = subgraph_ids.map(lambda x: temp[x])
 # remove redundant column:
 nodes_df = nodes_df.drop(columns=['addable'])

 print('Making changes to edges')
 # map subgraph back to edges from nodes:
 edges_df['subgraph'] = edges_df['Start Node'].map(lambda n:
nodes_df.loc[n,'subgraph'])

 print('Updating subgraph data')
 # Re-map subgraphs['nodes'] column to lists:
 subgraphs['nodes'] = subgraphs['nodes'].map(lambda ser:
ser.values.tolist())
 # Get edge indices and number of edges and nodes for each subgraph:
 grouped_temp = edges_df.reset_index().groupby('subgraph')
 edge_count = grouped_temp['index'].count()
 subgraphs['num_edges'] = grouped_temp['index'].count()
 subgraphs['edges'] = grouped_temp['index'].apply(lambda g:
g.values.tolist())
 subgraphs['num_nodes'] = subgraphs['nodes'].map(lambda x:len(x))

 # write changes to file:
 print('writing changes')
 subgraphs.to_feather(subgraphs_path)
 nodes_df.to_feather(nodes_path)
 edges_df.to_feather(edges_path)
 print('done')
```

```
finding transaction partners. This will likely take a couple minutes.
processing nodes_df will require 515088 iterations. This will likely
```

```
take a couple minutes.
finished!
total nodes: 515088
total subgraphs: 114139
total nodes in subgraphs (should match total nodes): 515088
sorting subgraphs
Making changes to nodes
Making changes to edges
Updating subgraph data
writing changes
done
```

Get change in accounts for nodes:

```
 import numpy as np
 currencies = list(
     set(
         edges_df['Receiving Currency'].tolist() +
         edges_df['Payment Currency'].tolist()
     )
 )
 num_currencies = len(list(set(currencies)))
 num_nodes = len(nodes_df)
 temp = np.zeros(shape=(num_nodes, num_currencies))
 c = {}
 c_rev = {}
 for i in range(len(currencies)):
   c[currencies[i]] = i
   c_rev[i] = currencies[i]
 def handle_edge(e):
   temp[e['Start Node'], c[e['Payment Currency']]] += e['Amount Paid']
   temp[e['End Node'], c[e['Receiving Currency']]] -= e['Amount
Received']
 dd.from_pandas(edges_df, npartitions=32).apply(
     handle_edge, axis=1, meta=(None,'object')
 ).compute()
 deltas = pd.DataFrame(temp).rename(columns=c_rev)
 for col in deltas.columns:
   nodes_df['Delta ' + col] = deltas[col]
 nodes_df.to_feather(nodes_path)

edges_df['Payment Format'].unique()

array(['Reinvestment', 'Cheque', 'Credit Card', 'ACH', 'Cash', 'Wire',
       'Bitcoin'], dtype=object)
```

Making final transformations to obtain a singular final dataframe:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import balanced_accuracy_score
```

```
# Convert Timestamp entries todatetime representation:
edges_df['Timestamp'] = pd.to_datetime(edges_df['Timestamp'])
# Get all unique transaction times in order:
temp =
pd.Series(edges_df['Timestamp'].unique()).sort_values().reset_index(dr
op=True)
# Get the first time in the dataset:
min_time = min(temp)
# define a temporary series that can be used to map times to numerical
values:
temp = pd.Series(
    temp.map(
        lambda t: (t - min_time)/np.timedelta64(1,'s')
    ).values.tolist(),
    index = temp.map(lambda t: str(t)).values.tolist()
)
# convert Timestamp entries to numerical values:
edges_df['Timestamp'] = edges_df['Timestamp'].map(lambda t:
temp[str(t)])
# Make final_df, a dataframe for containing all info in the end:
final_df = edges_df.copy()
# Remove this column from being added as we dont want to add it again:
temp = nodes_df.drop(columns=['subgraph'])
# Add columns containing information about the Start Node to the
matrix:
final_df = final_df.join(
    temp,
    on='Start Node',
    how='left'
)
# Add columns containing information about the End Node to the matrix:
final_df = final_df.join(
    temp,
    on='End Node',
    how='left',
    lsuffix=' End',
    rsuffix=' Start'
)

# remove unwanted columns from final_df:
drop_columns = [
    'Payment Currency',
    'Receiving Currency',
    'Account End',
    'Account Start',
    'transaction partners End',
    'transaction partners Start'
]
# transform payment format into boolean features:
for form in final_df['Payment Format'].unique():
```

```
    final_df[form] = final_df['Payment Format'].map(lambda s: s ==
form)
final_df = final_df.drop(columns = ['Payment Format'])
final_df = final_df.drop(columns=drop_columns)

Y_col = 'Is Laundering'
# Add y column to final_df:
final_df[Y_col] = ys_df[Y_col]
```

Tests for Decision Tree Classifier Fitting

Due to the highly unbalanced nature of the dataset, if the scoring while training the model was not adjusted to account for the low likelihood of a transaction being a money laundering attempt, then the model would not ever predict that a transaction was money laundering. As such, we train using the balanced accuracy score as our training metric in order to obtain some actual predictions.

The cell below fits 100 different decision tree classifiers to samples from the dataset and averages the resulting balanced accuracy score in order to obtain an idea of what sort of starting score we are looking at for our model.

```
from sklearn.tree import DecisionTreeClassifier
# specify number of times to run:
num_tests=100
# make accumulators:
balanced_scores = []
models = []
# specify size of sample to draw each time:
sample_size=1000000
for i in range(num_tests):
    # take a subset of transactions and split them into a train and
test set:
    train, test = train_test_split(final_df.sample(sample_size),
test_size=0.3)
    # separate columns and matrixes for fitting decision tree:
    X_cols = list(final_df.columns[:-1])
    X_train, X_test = train[X_cols], test[X_cols]
    Y_train, Y_test = train[Y_col], test[Y_col]
    # fit the decision tree:
    model =
DecisionTreeClassifier(class_weight='balanced').fit(X_train, Y_train)
    # append the model and the balanced accuracy score:
    models += [model]
    balanced_scores += [balanced_accuracy_score(model.predict(X_test),
Y_test)]
results = pd.Series(balanced_scores)
print('average balanced accuracy score:', results.mean())
```

```
average balanced accuracy score: 0.6036636688889316
```

Making Improvements:

The cell below first transforms X with PCA, before fitting the decision tree.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.decomposition import PCA
from sklearn.model_selection import cross_val_score
import graphviz
from sklearn.tree import export_graphviz
from sklearn.preprocessing import  StandardScaler
import matplotlib.pyplot as plt

# separate columns and matrixes for fitting decision tree:
X_cols = list(final_df.columns[:-1])
X  = final_df[X_cols]
Y = final_df[Y_col]

# apply PCA to x
    # scale X before PCA
scaler = StandardScaler()
scaler.fit(X)
scaled = scaler.transform(X)
    # decompose x according to PCA:
pca = PCA()
pca.fit(scaled)
decomposed = pca.transform(X)
    # plot results and select cutoff:
threshold = 0.9
explained_var = np.cumsum(pca.explained_variance_ratio_)
plt.plot(explained_var)
selected = min([j for j in range(len(explained_var)) if
explained_var[j] > threshold])
plt.plot([selected], [explained_var[selected]], '.', label='selected
cutoff')
plt.xlabel('Number of features')
plt.ylabel('Explained Variance')
    # redo PCA but only with desired number of components:
pca = PCA(n_components=selected)
pca.fit(scaled)
decomposed = pca.transform(X)
# fit the decision tree:
print('fitting tree')

tree = DecisionTreeClassifier(class_weight='balanced')
tree.fit(decomposed, Y)
# cross validate
print('final balanced accuracy score (through 5-fold cross
validation):',cross_val_score(tree, X, Y,
            scoring="balanced_accuracy",
            cv=5).mean())
```

```
C:\Users\chris\anaconda3\envs\ORIE4741\lib\site-packages\sklearn\
base.py:432: UserWarning: X has feature names, but PCA was fitted
without feature names
  warnings.warn(
C:\Users\chris\anaconda3\envs\ORIE4741\lib\site-packages\sklearn\
base.py:432: UserWarning: X has feature names, but PCA was fitted
without feature names
  warnings.warn(

fitting tree
final balanced accuracy score (through 5-fold cross validation):
0.5984658573914015
```