CSYE 7374 Parallel Machine Learning & AI

Detecting Unknown Mining Pools and their Behaviour within Bitcoin using Parallelized Machine Learning

Presented By

- Srinjoy Chakravarty
- Srishti Ashok Mishra

Northeastern University

INTRODUCTION

- **What is Blockchain?**
  A system in which ***blocks*** of cryptocurrency ***transactions*** are linked together through cryptographic ***hashes*** via a ***peer-to-peer*** network

1. **Node** — computing ***peer*** within the ***network***
2. **Transaction** — immutable ledger ***record*** indicating a ***transfer*** of ***value***
3. **Block** — *data-structure* used to batch transactions and ***relay*** them ***peer-to-peer***
4. **Chain** — a ***sequence*** of blocks in ***chronological*** order
5. **Cryptocurrency** — algorithmic value token generated outside central bank or governmental purview
6. **Miners** — powerful nodes which create new monetary supply
7. **Consensus** — decentralized code-driven governance to validate all network activity

Definitions

Unknown Mining Pool Detection

Definitions[1] :

- **Pooled Mining**: Pooled mining "pools" all of the resources of the clients in that pool to generate the solution to a given block. When the pool solves a block, the 6.25 BTC generated by that block's solution is split and distributed between the pools participants.

- **Solo Mining**: Solo mining is when a miner performs the mining operations alone without joining a pool. All mined blocks are generated to the miner's credit.

Cause: Unknown Mining Pools are **not** necessarily one **coordinated private** pool. Rather, they could be a combination of solo miners and group miners that hide their pool

Motivation: Shadow mining may allow a group of miners to *hide their hash rate* or *protect proprietary hardware* (sub 5 nanometre ASIC chips)

Result: The lack of transparency poses a danger to the health of the bitcoin network as it may allow for game-theoretic attacks discussed later

OBJECTIVE

Background: By analyzing the different **characteristics** of bitcoin transactions and identifying the behaviour of **publicly-known** Bitcoin pools in picking them to mine, we build a Machine Learning Classifier that is able to predict whether a transaction is mined by a **mining pool** or **not**

Motivation: Imagine if Nakamoto had moved one of his coins (he hasn't moved his coins in years) and our classifier flagged it to have mined by an **Unknown Mining Pool**. Our deep-dive analysis would certain be useful.

Result: The confusion matrix from our predictive model allows us to use **'False Positives'** to identify transactions that have been mined by a **Shadow/ Dark** mining pool

Reasoning: *data points (transactions) where our model predicts* ***mined_by_mining_pool == true****, but in actuality have been labelled by the network as* ***mined_by_mining_pool == false*** *(WHY? LACK OF INFO??)*

Objective: identify **node addresses** flagged to be *'unknown'* by our prediction classifiers *(false-positive)* and do further data analytics *(node centrality metrics etc.)* to unravel **malicious** network activity

Dataset

Unlike other data, the blockchain never sleeps…

Since our dataset is huge and dynamic (changes every 10 minutes), we are indebted to the rigorous ETL framework[1] built by Google [1] to utilize their specialty BigQuery service on live blockchain data.

The dataset we used ranged from a monster **1.0 to 1.3 GB**!

This allowed us to get an incredible **accuracy** rates of **99.8395885589584%** (Stacked Ensemble Classifiers) and **99.8375925642544%** (Tensorflow Neural Net)

We used **26 unique features** of a bitcoin transaction ranging from how much **total monetary value** the transaction contains to **how long** the **inputs** used in the sat **idle** before being included in the transaction!

Analysis Methodology

We use a combination of different Classification Algorithms and an Artificial Neural Network (ANN) to train our prediction models.

1. Serial Model 1 = Random Forest (SKLearn)

- We check the accuracy of our classifier by plotting a Precision-Recall Curve and calculating the area under it
- We use a Confusion Matrix to list out false positives and false negatives
- These transactions are then further investigated with network analysis

2. Serial Model 2 = Gaussian Naive Bayes (SKLearn)

3. Serial AI Model = Artificial Neural Net (TensorFlow)

4. Parallelized ML Model = Multi-Node Stacked Classifier over Distributed Memory

(using Open-source MPI)

Optimization Techniques Used

To achieve maximum accuracy we use a Stacking technique where we split into **4 MPI Processes** where each process runs a ML classifier (layer1) on our given dataset. Each process then fits the test data and returns its predictions back the the **Master MPI Process.**

```python
def classify(X_train, X_test, y_train, y_test):
    # classification
    algorithm = None
    classification_time_start = MPI.Wtime()
    if rank == 0:
        algorithm = 'ridge'
        clf0 = RidgeClassifier()
        st.fit(clf0, X_train, y_train)
        classification_output = st.predict(clf0, X_test)
        pass
    elif rank == 1:
        algorithm = 'randomForest'
        clf1 = RandomForestClassifier(n_estimators=10)
        st.fit(clf1, X_train, y_train)
        classification_output = st.predict(clf1, X_test)
        pass
    elif rank == 2:
        algorithm = 'lda'
        clf2 = LinearDiscriminantAnalysis()
        st.fit(clf2, X_train, y_train)
        classification_output = st.predict(clf2, X_test)
        pass
    elif rank == 3:
        algorithm = 'GaussianNaiveBayes'
        clf3 = GaussianNB()
        st.fit(clf3, X_train, y_train)
        classification_output = st.predict(clf3, X_test)
        pass

    classification_time_end = MPI.Wtime()
    classification_time = classification_time_end - classification_time_start
    print(
        f'[TIME] Process {rank} finished classification by {algorithm} algorithm with time: {classification_time}')
    return classification_output

```

The master MPI process **receives** each of the 4 predictions and puts them through **voting mechanism** to come up with predictions for every row (x-test) of transaction data by using a **majority vote** is used to determine **y-pred** (layer2)

```python
    def vote(self, preds):
        np_preds = np.array(preds).T
        result_list = list()
        for row in np_preds:
            result_list.append(np.argmax(np.bincount(row)))
        return np.array(result_list)

```

ANALYSIS

Model storage, joblib vs pickle vs parquet

For 1.2 gigabytes of Bitcoin Transaction Data

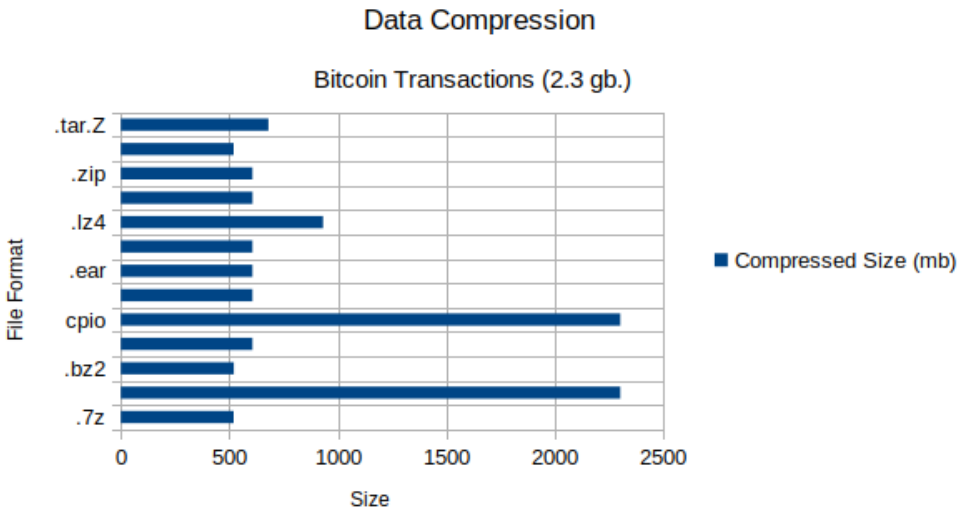
Big Data Storage Comparison (1.2 GB Bitcoin Transaction Data)	Write to disk	Read to disk	Filesize (mb)
Parquet (multi-threaded)	49.5127	70.133	725.7
Feather (multi-threaded)	56.2501	78.925	823.6
Pickle	49.813	13.8363	2100
HDF5*	Fails	Fails	Unfair to compare

*HDF5 has superior Read/Write speeds than both Parquet and Feather but begins to fail once the dataset grows too large

Dataset Compression (Post - ETL)

We tried 27 different data compression techniques to minimize the size of our dataset (1.3+ GB) and reduce network latency when passing data between local workstations, servers and high-performance cluster nodes. The file format with the **Highest Compression Ratio** was **.7zip** with **Space Saving Score** of **78.35%**

Space Saving = (Compressed Size) / (l
0.7835 = 1 - 519.3 / 2,982.4



Saving the Model so you can Throw Away the Data (Pickle vs JobLib)

The idea behind any Data Science is to be able to build a model and throw away all the data.

We compared data write speeds of both Pickle vs JobLib (the palatalization library) behind sci-kit learning (written in Cython) and found that joblib was usually significantly faster on large numpy arrays because it has a special handling for the array buffers of the numpy data structure.

```
from joblib import dump, load
dump(rf_classifier_8_cpus, 'model_rf_8_cpu.joblib')
```

```
['model_rf_8_cpu.joblib']
```

Model Training Parallelization

Hardware Setup (4-node CPU cluster each with 180 GB RAM)
> $ srun -p short --cpus-per-task 4 --ntasks 4 --nodes 4 --pty --export=ALL --mem=180Gb --time=08:00:00 /bin/bash

We created a virtual anaconda environment to install our custom software
> $

We used MPI for Python to interface with HPC shared nodes

> $ conda install -c anaconda mpi4py

We used OpenMP version 3.1.2
> $ module load openmpi/3.1.2

```
[INFO] Program runned in 4 processes
[INFO] Hello from process number 0
[INFO] Bcasting data from the root process (0)
[TIME] Master process (0) finished Bcasting data with time 86.51921606063843
[TIME] Process 0 finished classification by ridge algorithm with time: 24.678951025009155
[ACCURANCY] Final accurancy for test-train is 0.9983958855895836
[INFO] Stacking classifier finish work with time: 203.12623000144958
[INFO] Hello from process number 1
[TIME] Process 1 finished receive bcasted data with time 87.64578890800476
[TIME] Process 1 finished classification by randomForest algorithm with time: 98.60505700111389
[INFO] Hello from process number 3
[TIME] Process 3 finished receive bcasted data with time 88.20838403701782
[TIME] Process 3 finished classification by GaussianNaiveBayes algorithm with time: 17.277169942855835
/home/s.chakravarty/.conda/envs/parallel_flow_env/lib/python3.7/site-packages/sklearn/utils/deprecation.
py:143: FutureWarning: The sklearn.linear_model.ridge module is  deprecated in version 0.22 and will be
removed in version 0.24. The corresponding classes / functions should instead be imported from sklearn.l
inear_model. Anything that cannot be imported from sklearn.linear_model is now part of the private API.
  warnings.warn(message, FutureWarning)
[INFO] Hello from process number 2
[TIME] Process 2 finished receive bcasted data with time 87.7812020778656
[TIME] Process 2 finished classification by lda algorithm with time: 31.150506019592285
```

Performance Gains in Model Training gained with Distributed Memory Parallelization

How Stacked Classifier Works
Each process trains on the datasets and builds a model
Each process then provides a probabilistic prediction with .fit()

Vote Function[1]
The Master Process then takes a vote on which model to use for prediction (binary 0 or 1)

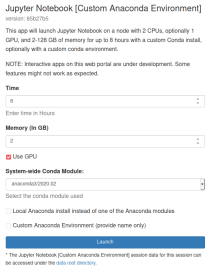
Improvements
Bin count gives you the ratio of which model voted for which outcome [0 | 4], [1 | 3], **[2 | 2]**, [3 | 1], [0 |4]
Unfortunately, since its an even number of models argmax() defaults to the first occurrence i.e. False
In this case we can take 2 approaches

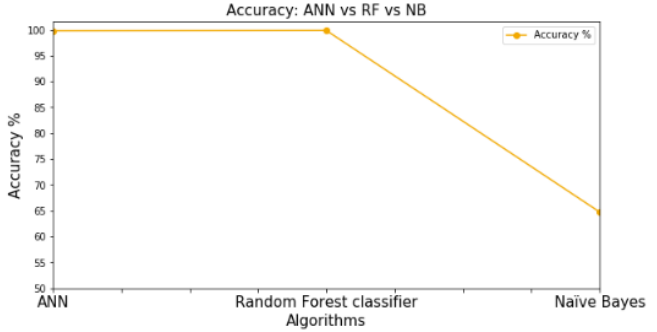
1) Numpy argmax - random tie breaking
2) Odd number of Classifiers (5 MPI Processes)

ML Model Accuracy Comparison

When the accuracy is compared, we see that our Random Forest Classifier performs the best with our data (as long as we use around 100 trees) while our Artificial Neural Net only reaches the same level of accuracy when we use a **high number** of **Epochs**! The high number of Epochs means the entire training of duration including multiple rounds back propagation takes a much longer time!

We tried to use a GPU setup via OpenOnDemand (OOD) on HPC to speed up our Notebook and found some reduction in training times.





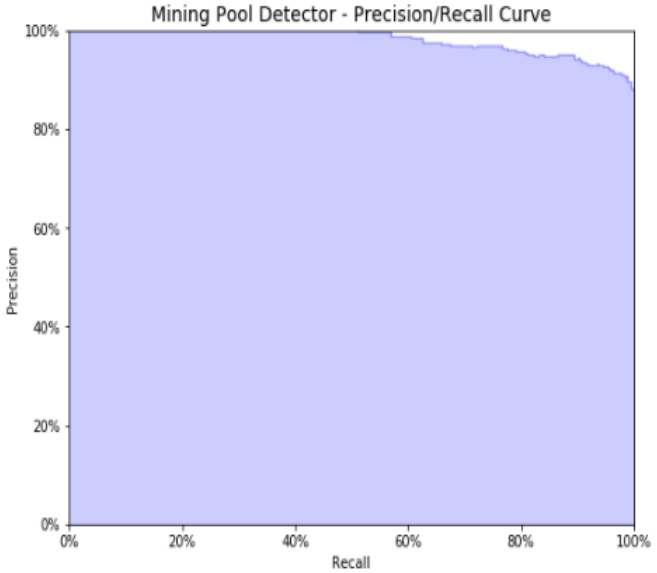

```
time_start = time.time()
ann.fit(np.array(x_train_ann), y_train_ann, batch_size = batch_size, epochs = epochs, validation_data = (x_test_ann, y_test_
print('Time elapsed: {} seconds' .format(time.time() - time_start))
```

```
Train on 4008025 samples, validate on 1002007 samples
Epoch 1/30
4008025/4008025 [==============================] - 55s 14us/step - loss: 0.0115 - binary_accuracy: 0.9967 - val_loss: 0.0062 -
val_binary_accuracy: 0.9980
Epoch 2/30
4008025/4008025 [==============================] - 51s 13us/step - loss: 0.0064 - binary_accuracy: 0.9980 - val_loss: 0.0061 -
val_binary_accuracy: 0.9980
Epoch 3/30
4008025/4008025 [==============================] - 34s 8us/step - loss: 0.0064 - binary_accuracy: 0.9980 - val_loss: 0.0060 - v
al_binary_accuracy: 0.9980
Epoch 4/30
4008025/4008025 [==============================] - 34s 9us/step - loss: 0.0063 - binary_accuracy: 0.9980 - val_loss: 0.0063 - v
al_binary_accuracy: 0.9980
Epoch 5/30
4008025/4008025 [==============================] - 34s 9us/step - loss: 0.0063 - binary_accuracy: 0.9980 - val_loss: 0.0059 - v
al_binary_accuracy: 0.9980
Epoch 6/30
4008025/4008025 [==============================] - 35s 9us/step - loss: 0.0064 - binary_accuracy: 0.9980 - val_loss: 0.0061 - v
al_binary_accuracy: 0.9981
Epoch 7/30
4008025/4008025 [==============================] - 34s 8us/step - loss: 0.0064 - binary_accuracy: 0.9980 - val_loss: 0.0063 - v
al_binary_accuracy: 0.9980
```

Prediction Accuracy Curve

- An ideal system with high precision and high recall will return many results, with all results labeled correctly

- Our curve when applied on a heavily trained model should great Precision even at the highest end of the Recall spectrum.



Artificial Neural Network Speed Analysis

HPC Nvidia GPU (Open OnDemand on Discovery)

```
(base) [s.chakravarty@c2176 ~]$ nvidia-smi
Sun Aug  9 07:17:04 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.33.01    Driver Version: 440.33.01    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           Off  | 00000000:05:00.0 Off |                    0 |
| N/A   31C    P0    68W / 149W |      0MiB / 11441MiB |     86%      Default |
+-------------------------------+----------------------+----------------------+
```

Octa-core i7 10th gen CPU

```
infinitybook - IP 192.168.0.70/24 Pub 75.69.107.191 Uptime: 3 days, 15:16:13

CPU  [  7.8%]   CPU \      7.8% MEM -   62.9% SWAP -    0.2% LOAD    8-core
MEM  [ 62.9%]   user:     4.2% total:  62.5G total:   8.00G 1 min:    0.65
SWAP [  0.2%]   system:   2.4% used:   39.3G used:    14.5M 5 min:    0.92
                idle:    92.2% free:   23.2G free:    7.99G 15 min:   1.03

NETWORK       Rx/s   Tx/s   TASKS 390 (2073 thr), 1 run, 302 slp, 87 oth
enp59s0f1       0b     0b
lo              0b     0b   CPU%   MEM%      PID USER       THR  NI S
wlp64s0      184Kb   20Kb   13.0   0.1      1952 eve        4     0 S plank
                            4.7    0.2      1859 eve        20    0 S budgie
DefaultGateway        7ms   4.3    0.4      1424 eve        10    0 S /usr/l

2020-08-09 07:21:50 EDT/s
```

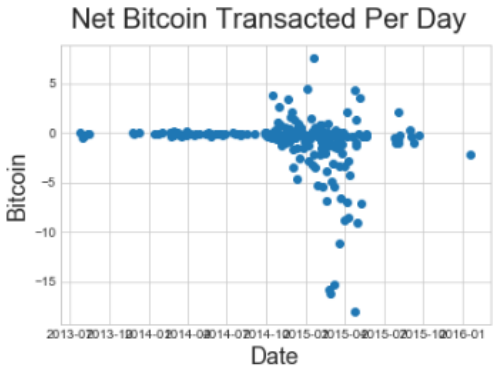
Dual-core CPU

OS Name	Microsoft Windows 10 Home Single Language
Version	10.0.18363 Build 18363
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	DESKTOP-DB50DUU
System Manufacturer	HP
System Model	HP Pavilion Notebook
System Type	x64-based PC
System SKU	Z4Q39PA#ACJ
Processor	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2712 Mhz, 2 Core(s), 4 Logical Pr...
BIOS Version/Date	Insyde F.25, 12/12/2016
SMBIOS Version	2.8
Embedded Controller Version	83.14
BIOS Mode	UEFI
BaseBoard Manufacturer	HP
BaseBoard Product	8216
BaseBoard Version	83.14
Platform Role	Mobile
Secure Boot State	Off
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\Windows
System Directory	C:\Windows\system32

Network Analysis on a chosen Unknown Bitcoin Node

- We decided to investigate our chosen unknown node address: **1PuVaCWwDK8VmFj4EN8e2EgxvDpjeyJmRx**

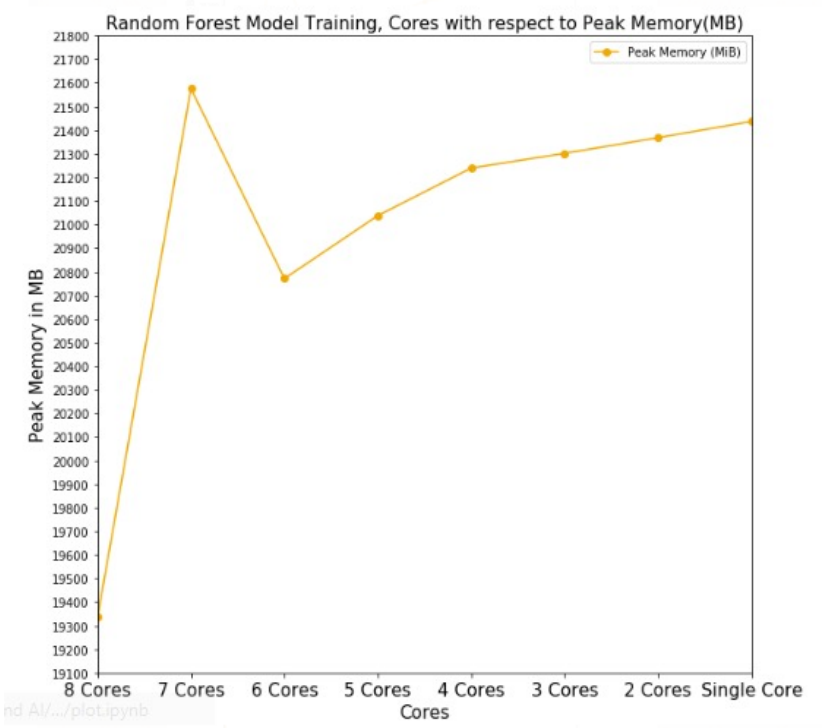
- **Exploratory Data Analysis**
  a. Unique addresses that were included in the transaction history of our chosen Node wallet
  b. Top addresses that our **Unknown Node wallet** interacted with
  c. Transaction activity of our chosen **Unknown Node** across time
  d. Node centrality analysis of the wallet address our node sent most Bitcoins to



Memory Profiling

We analysed how increasing the number of CPU cores allow us reduce the peak memory usage when training our machine learning models on huge datasets.

Here is an example of Peak Memory over multiple cores when using a Random Forest Training Algorithm



Future Optimization Plan

1. Use SKompiler[1] to transform trained SKLearn models into symbolic python (Sympy) expressions which, in turn, can be translated to C++, Javascript, Rust and Julia code. **Benchmark model training speed** of each language.

2. Transpile[2] trained scikit-learn models to C, PHP, Java and Ruby code. **Benchmark model training speed** of each language.

3. **Prediction:** *Julia code may have the highest model training performance.*

[1] konstantint/SKompiler: A tool for compiling trained SKLearn models into other representations (such as SQL, Sympy or Excel formulas)

[2] nok/sklearn-porter: Transpile trained scikit-learn estimators to C, Java, JavaScript and others.