Spring 2018

Bitcoin Transaction Fee Estimation Using Mempool State and Linear Perceptron Machine Learning Algorithm

Abdullah Al-Shehabi
San Jose State University

Bitcoin Transaction Fee Estimation Using Mempool State and Linear Perceptron Machine Learning Algorithm

A Thesis Presented To

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Abdullah Al-Shehabi

May 2018

The Designated Thesis Committee Approves The Thesis Titled

Bitcoin Transaction Fee Estimation Using Mempool State and Linear Perceptron Machine Learning Algorithm

by

Abdullah Al-Shehabi

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2018

Dr. Robert Chun	Department of Computer Science SJSU
Dr. Katerina Potika	Department of Computer Science SJSU
M.S. Mark Erhardt	BitGo Software Engineer

Acknowledgments

I would like to to thank Dr. Robert Chun for his guidance and support throughout my project. I would like to thank my committee members M.S. Mark Erhardt and Dr. Katerina Potika for their review of the project. A huge thanks to Mark who helped me design and reiterate the project from a deeply technical perspective. A special thanks to my parents and siblings for their patience and support for me along the way. A huge and special thanks to my fiance Tasnim for the continued support and encouragement throughout my toughest times. And most importantly thanking Allah for his blessings upon me.

Abstract

Bitcoin, the world's most valued cryptocurrency, uses a network of computers across the globe to create an immutable transaction record on a public ledger known as the blockchain. The blockchain consists of a series of timestamped blocks, where each block contains a series of transactions selected for inclusion in the block, generally based on how high of a fee the transaction allocates to the party responsible for confirming the transaction. Estimating an appropriate fee for Bitcoin transactions is a challenge for many transacting parties using Bitcoin as a digital currency. This work aims to help Bitcoin users save funds in their transaction fees when building multisig transactions by providing fee estimates that referenced the current state of the unconfirmed transaction pool using the perceptron machine learning classification algorithm.

Table of Contents

List of Figures

List of Charts

List of Tables

1. Introduction

Blockchain and cryptocurrency technology has been taking the computer science industry by storm for the past several years. The technology revolves around a large network of computers known as nodes, each having their own copy of the blockchain, where a new block is added to a node's copy of the blockchain after fulfilling a set of requirements to validate it. Previous blocks on the blockchain are immutable and each new block depends on information that references its predecessor, making a long chain of dependant blocks. In the case of Bitcoin (BTC), the most popular cryptocurrency at the time of this work, the blocks on the blockchain contain a series of transactions reflecting the transfer of funds. In most cases, any time a transaction is included in a block, it includes a transaction fee. The fee is collected by the node, or miner, that created the new block that includes the transaction in it. Determining the fee to be paid to the miners of the new block is up to the transaction owner. This raises the challenge for the transaction owner of selecting a transaction fee that is not too high nor too low in order for their transaction to be confirmed in a reasonable timeframe for them. This project is meant to introduce a different approach for estimating transaction fees that will include the transaction in a future block that the transaction owner had intended.

2. Background

2.1 What is the Blockchain

Cryptocurrency was initially invented in order to have a public decentralized system by which the currency can exchange owners and no one is able to modify coin ownership records. This is done via the blockchain, which is a decentralized public accounting journal of all the cryptocurrency's transactions that have ever taken place. The blockchain contains a history of all transactions that have taken place over time. This is done by storing transactions into blocks and blocks are linked together to form a chain where each block is dependant on data from the

previous block. The block is merely a piece of data that contains valid transactions and some other information discussed in more detail later.

Once the transactions are validated and a valid block is built, a node would add the new block to its local copy of the blockchain. In the case a valid block is received from another neighboring node on the network, a node would validate the block and then append it to its local blockchain and begin attempting to create the next block. Each block contains a cryptographic hash of the previous block as part of its permanent data. Any slight modification to the records of one block will create an invalid block which requires mining a new copy of the blockchain and proof of work as the rest of the chain will not support the invalid block. Due to the nature of cryptographic hashes, the change of one bit will create a completely different hash value. Bitcoin uses double SHA-256 as its hashing algorithm, which basically means that it will hash any piece of binary data twice before using it as the hash value. The blockchain therefore has immutable data that is complete with all the information regarding user coin addresses and their funds right from the first block ever created, known as the genesis block [1].

2.2 Transactions

If Bitcoin as a currency was dissected to analyze its functionality, it would be evident that the root of it all comes down to transactions. The purpose of a currency is to keep track of value and value transfers. In cash systems, this is done through the physical exchange of notes or coins being exchanged between parties. A similar concept applies to digital currency, where the owner of the digital coins has ownership of the coins as long as they have the private key capable of signing a transaction spending the coins. Whenever an owner of the coins spends their coins which are locked via their private key they create a transaction that marks their coins as spent and assigns those coins to a new address [14]. The new coin address has its own private key which then becomes the only key capable of spending those coins.

2.2.1 Transaction Components

Transactions let Bitcoin users spend Satoshis as the whole protocol accounts are in Satoshis. The ratio is one hundred million Satoshis per Bitcoin and people abstract the raw Satoshi count into Bitcoin as it is easier to relate the value of traditional currencies to Bitcoin than it is to Satoshis. A transaction is broken into inputs and outputs, where each transaction must have at least one input and one output. Each input is in fact a reference to Satoshis that are the outputs of a previous transactions known as unspents [1]. Each output remains in the unspent transaction output set until it is used as an input in another transaction. Any coins sent to a new address become unspent coins on that address usually denoted and referred to as UTXO's (Unspent Transaction Outputs). The balance of Satoshis in a wallet refers to the total of Satoshis waiting in one or more unspent(s) managed by the wallet software. Along with the transaction id, an output has an explicitly defined index based on its location in the transaction's list of outputs. An output has the amount of Satoshis which it can spend along with the pubkey script which if fulfilled, allows the coins to be spent [18]. In a transaction, the inputs use the transaction id and the output index to identify the output being spent. The output also has the signature script which specifies the parameters needed to spend the output. The amount of Satoshis spent in a transaction is higher than the amount in the outputs as the difference in funds is the transaction fee paid to the miner confirming the transaction.

2.2.2 Multisig Transactions

Multisig transactions are transactions that spend transaction outputs which require multiple signatures to be used. In order to broadcast a valid transaction, it is required for more than one key to sign the inputs which are multisig UTXO's. This feature is natively supported in Bitcoin but may not be supported in other digital currencies. The way this works is that when an address is generated, the rules set in the signature script specify that more than one signature with different private keys are needed. In the case of multisig addresses, the signature script will specify the threshold signatures needed to spend the coins on the address. Since multiple signatures with different private keys are needed, these types of transaction inputs are generally

used to have multiple parties each with their own private key have control over whether funds are spent or not.

2.2.3 Transactions and Blocks

Every block created on the blockchain must include at least one transaction, and specifically the coinbase transaction [1]. The first transaction in the block must be a coinbase transaction, which is how new coins are created and added to the available coins on the network. A coinbase transaction does not contain any source for the coins that it spends as it only contains a destination address for the funds. In the case of the coinbase transaction, the coins that were created cannot be spent in any transaction for at least another 100 blocks. This rule will be explained later in section 2.5. All transactions are recorded into blocks in binary "raw transaction" format. The "raw transaction" format is hashed to create the transaction id. In the figure 1 below, the data blocks is the level at which the raw transactions lie, each transaction is hashed into its transaction id and made into the leaves of the tree. Using these transaction ids at the lowest level of the tree, a Merkle tree is constructed by iteratively pairing each transaction id with one other transaction id and then hashing them to create another higher level of the tree. The hash of the pair is hashed with another pair's hash and the process continues with hash pairs until one hash value is created known as the Merkle root. In the case of an odd number of transactions in the block, the lone transaction id is paired with itself and hashed [10].

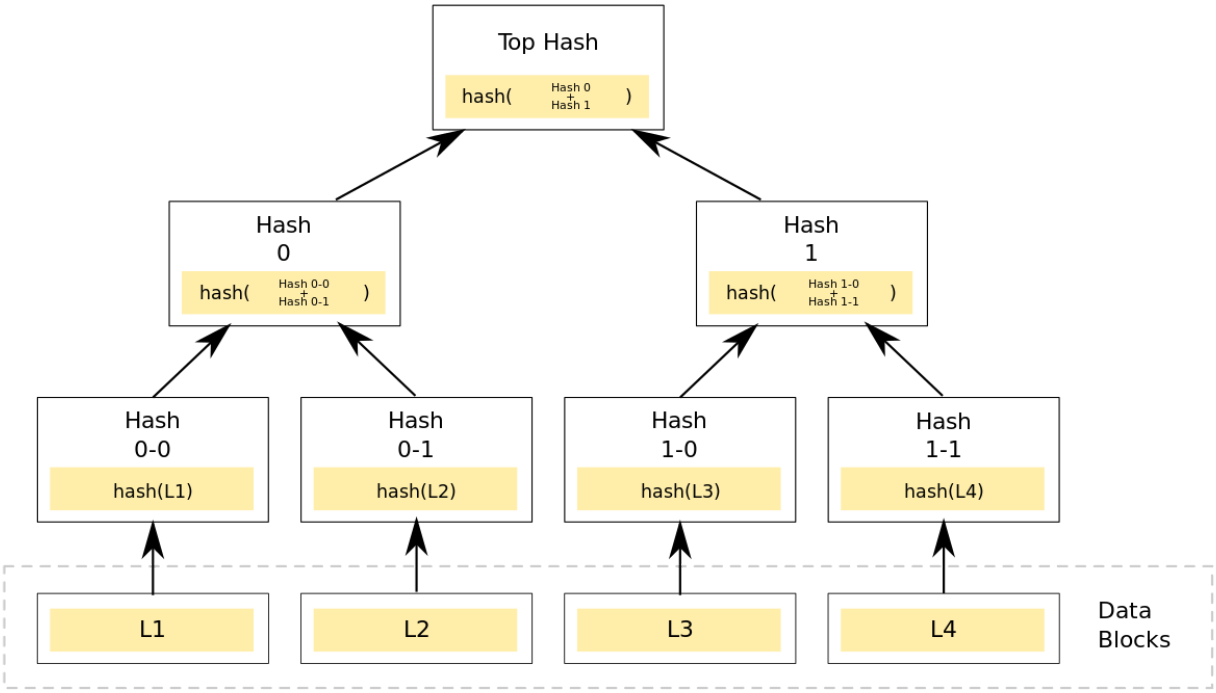

Figure 1: Merkle tree hashing setup. [5]

2.2.4 Transaction Fees

Miners generate blocks with the main intent of gaining profits from it, since Bitcoin is a digital currency where a miner that generates a block is rewarded in coins and transaction fees. As more space becomes available in blocks, the average cost for space on the blockchain becomes less. Miners responsible for generating new blocks will make less profit if less transactions are competing for confirmation, which makes space on the chain too abundant and transacting parties will pay less fees in order to have their transaction confirmed. However, there is a balance between the profit miners can make from fees and the block size. As space in blocks becomes abundant, it is necessary that there remains demand for transaction confirmation on the chain as that will keep the cost of fees higher and more profitable for the miners [3]. If miners no longer make profit off of mining for the Bitcoin blockchain then they may move to mining other cryptocurrencies and mining support for the chain decreases [19]. For the time being however,

the reward from generating a block heavily outweighs the profit from fees. The Bitcoin project depends on the support of miners to exist and insuring the mining network exists is key to the currency's survival. The two methods by which miners are rewarded is by transaction fees and mining rewards.

Fees play a key part in generating the miners profit; the average revenue per block over the course of 2017 has been averaging around 14 BTC per block. With the block reward currently set to 12.5 BTC's. The highest profit made up till the time of this work was 19.79 BTC and the lowest non empty block reward was 12.87 BTC [12]. As can be seen from these numbers, a large percentage of the mining profit can be made by the miners from fees alone. Since the block reward is halved approximately every four years, or every 210,000 blocks, for Bitcoin miners to continue support for the currency will require fees that can still provide them with profit. Thus, increasing the block size is something that will be quite probable since more transactions need to be processed as the coin becomes more widespread and adopted by many but only while the miners can still make profit off of mining and fees.

Fees and block rewards are what provide the Bitcoin currency with the processing support it needs, however high cost fees will also deter people from using the coin as a means of transacting if they are too high. The assumption used in this research project is that miners will select transactions to include in their block based upon the highest amount of profit that they can make. This is generally true unless a mining pool has an agreement with other parties that include out-of-band payments for transactions that are mined. If those agreements exist then those transactions will be outliers that do not have similar transaction fees to the ones in the same block. Transactions with extremely high fees are also considered outliers since they would be much higher than the average fee rate for transactions in the block. Bitcoin users will minimize network fees since it is in their best interest to save their funds. Even though miners benefit from higher transaction fees, the Bitcoin users are negatively affected by the high fees. This is a whole other issue on its own, but the main concern of this research project is to produce a method of fee estimation which will help users save on transaction fees.

2.2.5 Transaction Size

Another major component related to transactions is the raw size and the virtual size (vSize) of a transaction. The size of a block is limited to 4,000,000 weight units which corresponds to 1,000,000 virtual bytes (vBytes) [22]. The vBytes and vSize of a transaction are always equal as they refer to the same thing, which is the number of virtual bytes. The size of the transaction is the raw size of the data in bytes, while the vSize is a calculated value derived on the type of inputs for the transaction. A transaction without a single Segwit input has a calculated weight based on the following function:

$$4 * number\ of\ bytes = weight\ units\ of\ transaction$$

Segwit is an abbreviation to a Bitcoin feature known as Segregated Witness introduced in 2015 by Pieter Wuille. This feature moves the coin ownership verification script from the script signature to a new part of the transaction called the witness [22]. As such the transaction weight is calculated a little differently so that it is still backwards compatible, since the weight is what is used to limit the number of transactions that are included into the block. However, Segwit transactions contain two different components which have different proportions. For example, a Segwit transaction can be 1000 bytes in raw data size and have 300 of those 1000 bytes be in the witness script component. Or the transaction can have 500 of the 1000 bytes be in the witness script component, as such the transaction weight is different every time depending on how the bytes are distributed among the components. The way the weight is calculated for Segwit transactions is using the following formula

$$witness\ script\ bytes * 3 + number\ of\ bytes\ not\ in\ witness\ script = transaction\ weight$$

As such, Segwit transactions are backwards compatible with the Bitcoin protocol running on legacy nodes running older software. The weight limit applies to both, but the weight of Segwit transactions is less than that of transactions spending non Segwit inputs. When a miner is looking to select a transaction to include in the block, they see two transactions with the same fee but different weights. If they have the same number of inputs but one only uses Segwit inputs and the other uses non Segwit inputs, then the miner will see that the Segwit transaction has a lighter

weight and will make more profit by including the transaction with the lighter weight and a "higher fee rate" even though the actual number of bytes in the Segwit transaction will be higher than the legacy transaction.

2.3 Generating a Block

A block contains up to 4,000,000 weight units. A simple transaction containing one input and two outputs is about 1000 weight units and 226 in actual raw data bytes and so a block can hold up to 4000 transactions [22]. The reason there are two outputs in a simple transaction is because Bitcoin takes in all the inputs in a transaction and spends them sending an amount to another address and sending the change to a change address. The change address can be any address, as it is just another location to store the coins.

When a block is built and all the transactions in it are hashed to create the Merkle root, in order for the block to be accepted by other nodes, it must satisfy the difficulty criteria for Bitcoin. Proof of work is a way to guarantee that a large amount of computational effort was placed into generating the block. It is also easily verifiable whether the block has that Merkle root or not just by recreating the block and hashing all the transaction pairs up to the root. Bitcoin's proof of work model specifies that in order for a block to be generated, the block hash should be less that a certain value known as the target [7]. That value is determined based on how powerful the hashing power of the network is. The more computers attempting to create a block where the hash is less than the target value, the faster a block can be created. The Bitcoin blockchain core code specifies that the difficulty (the ratio between maximum possible target and actual target) be reset every 2016 blocks to a new value depending how long it took on average to mine a block [7]. Bitcoin's ruleset wants to ensure that the average time between each block on the blockchain is 10 minutes and so if the average time is less than 10 minutes then that means the difficulty to mine a block needs to be raised and vice versa.

In order to raise the difficulty, the new target needs to be a smaller number than the current target value. The reason this works is because the Merkle root is a SHA256 cryptographic hash value. Which means it is impossible to determine what the value will be prior

to hashing all transactions in the block and it is impossible to find a transaction set that will produce a hash value less than the difficulty by reversing the hash. Imagine that a block was created using 2000 different transactions. The transactions are hashed and the Merkle root is produced. The block header and all transactions are hashed to create the new block's hash value.

A block consists of multiple parts which are the magic number, block size, transaction counter, all transactions, and block header [11]. A block's id is the hash of the block header, of which the Merkle root is part of. The magic number the size of four bytes with a value always set to 0xD9B4BEF9. This magic number is a way to identify the file/data structure of incoming data which in the case of the bitcoin network is used by nodes to identify a block that is being propagated across the network. The block size has a length of 4 bytes and is a number that indicates the number of bytes that make up the block. The transaction counter is a number that consists of 1 to 9 bytes worth, that is used as part of the Bitcoin protocol. The transactions included in a block are the signed transactions which show the source and destination of funds. The last component of a block is the block header which consists of six components.

Field	Purpose	Size (Bytes)
Version	Block version number	4
hashPrevBlock	256-bit hash of the previous block header	32
hashMerkleRoot	256-bit hash based on all of the transactions in the block	32
Time	Current timestamp as seconds since 1970-01-01T00:00 UTC	4
Bits	Current target in compact format	4
Nonce	32-bit number (starts at 0)	4

Figure 2: Block header components [11]

The first component in the block header is the version number. The version number has 4 bytes reserved for it and it indicates which set of block validation criteria to follow. Next follows the previous block header hash and it is used as a way of making each block dependant on its predecessor. Changing one transaction in a previous block will completely change the Merkle root due to cryptographic hash behavior and since each block uses its predecessor's Merkle root,

this makes every block in the blockchain have a hash value dependant on the previous block, creating a long blockchain of dependencies. Thus the blocks on the blockchain once accepted across the network become infeasible to change, since even changing one bit of data in any block will create a drastically different blockchain that would need its own proof of work and will have difficulty catching up to the rest of the network's chain. The third component of the block header is a timestamp which references the time the block has been hashed. A nonce is also part of the block header and goes up to a 4 byte value. The nonce is used to allow for a node to hash a block template with a different value every time so that it may generate a block hash that meets the network difficulty requirement. Then comes the nBits, which is an encoded version of the target difficulty value.

The block hash is a value that is represented using hexadecimal notation. The difficulty is a number determined by the Bitcoin network and if the block hash is larger than that number, then the block must be rehashed, changing some parameters while doing so, until we get a block hash that is less than the difficulty value. The block hashes and difficulty values are very large 32 byte numbers. An example of a possible Double SHA256 hash is 10b71be629f471644dc83da-8d4bbb280a39d403f9a5eb0b46b559d9da5ab8a9a for a block, the amount of computational value to get a SHA256 hash that is less than the difficulty value of lets say 1452839779145 (The difficulty value on Nov 5th, 2017) is enormous since it is highly determined by luck. One such block hash that is less than the difficulty value is block 492272 of the Bitcoin blockchain which is 000000000000000004d0ddef91c8fa1ff4b21466c54bdd750aa28b13fb744e7 [23].

Mining is what the Bitcoin community refers to as the act of attempting to find and publish the new block by producing the proof of work [6]. The miners are nodes on the network that extract transactions from the pool of unconfirmed transactions, known as the mempool, in order to include them in the block they are creating. Every node on the network has its own mempool from which it selects transactions [9]. A large amount of computing power is done in order to generate the block as many block hash values are produced and checked if they are less than the difficulty value. As soon as a valid block hash is produced, the miner relays the new block to the node network. If a miner receives a new block from its neighboring nodes, it verifies

that the block is valid and then appends to its copy of the blockchain, then starts mining a new block.

Since there are a lot of miners and a lot of computing power on the Bitcoin network consisting of miners, the probability of one miner being the lucky one to get a valid block that adheres to the difficulty value is extremely slim. As such, miners decided to create mining pools, where the pool consists of many nodes that work together on generating a new block. As mentioned before, every block includes a coinbase transaction which allows for new coins to be added to the Bitcoin network. The miners which create the block, assign the coins from the coinbase transaction to themselves as a reward. But in the case a miner is part of a mining pool, the reward gets distributed among the miners based on the amount of hashing power they contributed to mining the new block. The pooling of resources reduces the variance in payout.

2.5 Blockchain Splits

A common question that arises is what happens when two miners hash a valid block and broadcast it across the network at the same time. Since once a miner receives a block from a neighboring node and validates it, it adds it to its copy of the chain and starts mining the next block, and if another block shows up as a valid block to the one it just appended to its chain, the node ignores it and continues mining a new one to continue with in its local copy of the chain. This creates a case where a chain split occurs among miners on the node network, where a portion of the miners have a version of the chain that is different than the other miners. What happens in this scenario is that the chain with the majority of the miners will move faster than the other chain, since it has more miners and more hashing power it will be able to create blocks faster. An encoded rule followed by Bitcoin nodes was implemented by Bitcoins creator Satoshi which shows that longest chain is Bitcoin. When a longer chain is presented to a node, the node reverts back to the last shared block and then reorganizes so that it is identical to the longer chain. The longer chain means it is a chain that accumulated more processing power to create than the current one. This means that when a fork happens among the nodes, the longest chain will be the one that prevails, and so the miners with a shorter version of the chain tend to

abandon their chain and hop back onto mining for the longest chain. This is the reason behind the rule that coinbase transactions cannot be spent until 100 blocks have passed. Since new coins added to the Bitcoin network should not be used as inputs of a transaction until it is certain that the reward of mining a block was indeed given to the spender and not a reward that was given in an abandoned fork of the chain.

3. Related Works

In this chapter we will be looking at what other research and solutions have been proposed or implemented in the determining what transaction fees to use when broadcasting a transaction. As mentioned earlier in section 2.2 regarding transactions, the transaction fee is the difference in funds between a transactions inputs and a transactions outputs. The fees are paid to the miners who generate the block and miners will generally include a transaction in their block if it gives them the most profit. As such, the higher a transaction fee the more likely it is to be included in a block sooner and confirmed on the blockchain.

3.1 Optimizing Fee Estimation via the Mempool State

Several methods exist that help decide what fee to use in a transaction and in this section, Karl-Johan Alm from DG Lab's take on fee estimation shall be explored. Alm gave a conference talk at Scaling Bitcoin 2017 at Stanford University where many professionals and developers from the Bitcoin community have met to discuss proposals for improving Bitcoin's scalability. At the conference, Alm discussed his method of using mempool state to create transactions with fees that are as low as possible. Setting a transaction fee to be higher than all other transactions for faster confirmation is not generally the wisest decision to make when deciding on what fee rate to use in a transaction. Many transactions are being held in the mempool for the miners to take from when mining a new block, therefore Alm proposes to consider the state of the mempool when deciding on what transaction fee rate to set for a new transaction being broadcasted and using Bitcoin's replace by fee (RBF) feature to bump up the fee in order to confirm the transaction in the intended target block.

3.1.1 Replace By Fee

Bitcoin transactions are irreversible after confirmation but may remain unconfirmed indefinitely if they were never included in a block. The reason the transactions may end up stuck mostly lies on the fact that other transactions took precedence over the transaction in question and were included in the blocks instead. This raises the issue of having a transaction being stuck for a long time hindering the use of the digital currency as a payment method since parties involved will not be able to confirm the sending or receival of their funds. This issue can be solved using Bitcoin's RBF feature which allows the spender to create a new transaction with the same inputs (outputs may be different), but reduce the funds being sent to the outputs to a lower value, effectively producing a higher fee for the miners to profit from [13]. The new transaction can then be broadcasted to all nodes across the network and most nodes with a copy of that transaction in their mempool will only replace the transaction with the updated one if it signaled the use of RBF [23].

3.1.2 Mempool State and Fee Estimate

Transactions sit in the mempool for as long as it takes for a miner to finally include them in a block. The time spent in the mempool has no impact on whether the transaction is mined or not. Which is a confusing point since a lot of Bitcoin services use the number of blocks till confirmation to tell users with low fees how long their transactions will take to confirm. The reason those estimates are not reliable is because they do not use the current state of the mempool to determine whether the fee is suitable or not, but instead use historical data and the average fee used in the most recently mined block. Replacing by fee will however allow the sender of the transaction to check the transactions in the mempool and adjust their fee accordingly so that they can reach their target block effectively [2]. This also allows the transacting party to underpay on their fees and try to save funds by publishing a transaction with a lower fee rate and once they see enough transactions in the mempool that may beat them to their target block, they use RBF to update their fee while attempting to not spend too much.

3.1.3 Optimization

This method can be optimized by taking the transactions in the mempool and sorting them by descending fee rates, and dividing groups based on what block target they would be placed in. The next part that needs to be done is determine which target block the transaction should be confirmed in and select a fee rate that will most likely confirm the transaction in that target block. The fee for the new transaction will then be set to be slightly higher than that value so that it can be guaranteed to confirm in the target block. As new transactions enter the mempool, the same process is to be repeated if the target block becomes out of reach due to the transaction having a lower fee than necessary to make the cut for that block. This cannot be done too frequently however, since there is a minimum fee increase per RBF update done for the transaction.

3.1.4 Advantages

Alm mentions a few key advantages to his fee estimation method that make his method a great solution to the Bitcoin transaction fee estimation problem. Replacing by fee is already an implemented feature in the Bitcoin protocol and enabling it while updating the fee based on transaction fee rates currently being used in the mempool can save, according to Alm, up to 80% on fees compared to the already implemented fee estimation algorithms. It is reliable and extremely safe as it will avoid the transacting party from making a transaction with an extremely high transaction fee [2]. In the cases where the mempool transaction fee average drops compared to the previous block then this method of optimizing fee rates used based on the state of the mempool will provide a much lower fee estimate than fee estimation algorithms that use historical data. This method of fee estimation also provides much more accurate estimates on when a transaction will be included in a block, since transactions are primarily included in blocks in the order of highest fees.

3.1.5 Disadvantages

This method of fee estimation has solid advantages, and the issue does not lie in the algorithm itself as it can produce extremely low fees that include the transaction in the targeted block. A transaction using this method must be flagged as replaceable since otherwise if a new transaction enters the mempool trying to spend the same coins then the node will consider it an attempt at double spending the same funds and will decline it. The major disadvantage for this technique of publishing transactions is that many Bitcoin services will not track RBF payments as the transaction can be completely changed when republished, sending the coins to a different location. As such, the funds will only be credited to a service user if the transaction is confirmed. Also, some services may not be configured properly for this type of transaction [23]. A disadvantage of using such a method of attaching fees to transactions is that transactions using this method will have an additional computational cost as the transaction needs to be signed again, broadcasted multiple times, relayed across the network and verified by miners every time it is broadcasted. This method may in fact require checking the mempool a large number of times in order to not underestimate the transaction fee needed and missing the target block. By depending solely on the mempool state, this method has no information regarding previous trends and can only produce an accurate fee estimation after analyzing the mempool several times.

The issue lies in multisig transactions, since the main effort in this project is to reduce fees for multisig transactions which require multiple signatures to allow spending the funds. In order to allow for multisig transactions to make use of RBF, multiple transactions need to be signed with varying fees [15]. Then as the state of the mempool changes, the signed transactions can be broadcasted to replace the ones with lower fees. However, this method does not work very well in the case of multiple parties holding keys for multisig transaction unspents. That means that the user will only have their fee estimates based on one instance of the mempool and not updated instances. The aim of this work is to produce fee estimates that will save the transacting parties in a multisig transaction as much as possible in fees while only using one snapshot of the mempool compared to the many an RBF transaction may use.

3.2 Bitcoin Core Fee Estimation Algorithm

Bitcoin Core is what the community calls the codebase used to control the currency. The Bitcoin Core codebase has two methods of fee estimation, one of which is economical and the other called conservative. At the time of this writing, Bitcoin Core v0.16 has been released and v0.15 was released prior to it with an improved fee estimation algorithm than what was previously used. The improvement does not change the logic of the fee estimation algorithm but does a better job ignoring outlying historical data and providing more accurate estimates. The algorithm pre v0.15 will be discussed first, then the details of all improvements with post v0.15 will be delved into.

3.2.1 Buckets and Targets

Bitcoin transaction fee rates fall within a range of 1 satoshi per byte up to several hundreds of satoshis per byte and can go much higher depending on the demand for space on the blockchain. Keeping track of all the fee rates attached to every single transaction requires a lot of storage space and computational power. Therefore, the first thing the Bitcoin Core fee estimation algorithm does is group the transaction fee rates into buckets where each bucket reflect a certain range of fee rates. The algorithm attempts to make very large range of fee estimates so that the lowest fee bucket has a range between 1 and 1.1 satoshis per byte, basically a ten percent increase. The next bucket has a range with another ten percent increase between 1.1 to 1.21 satoshis per byte, it goes on this way till 10,000 satoshis per byte. The second thing the algorithm tracks is the time until confirmation, which is the number of blocks between when a transaction first entered the mempool till it is confirmed. The target block is the idea of broadcasting a transaction with a fee rate that you expect will have it included within a certain number of blocks. For pre v0.15 Bitcoin Core the algorithm kept track of transactions that had a target block between 1 and 25 blocks.

3.2.2 Estimating Fees for a Target Block

To estimate how much fees a transaction requires to confirm within a certain number of blocks, a historic record of transactions in the mempool and the blocks is stored. The information stored includes the number of transactions that have entered the mempool in each fee rate bucket, and the number of transactions that successfully made it onto the blockchain within their target number of blocks. Using this information, Bitcoin Core can forecast the probability that a transaction with a certain fee rate would be able to achieve its target block.

Bitcoin Core has dynamic fee rates where fee rates change over time and therefore the fee estimation algorithm generates different fee rates depending on the current fee rate trend. Newer blocks are more likely to provide more accurate future fee rates as the value of the currency changes. As a consequence, newer blocks receive higher weight when it comes to using them for historical fee estimates than older blocks. The algorithm uses an exponentially weighted moving average. Every time a new block is generated, the weight of older blocks gets multiplied by a decay value. The decay value is set to 0.998 in Bitcoin Core pre v0.15. This means that as a new block is added to the chain, the weight of the fee rates used for estimation is multiplied by 0.988. A block from 2.4 days ago, about 346 blocks ago, would have a weight that is half of the weight as the newest block. And a block from 7.2 days ago, approximately 1037 blocks ago, would have a weight that is one eighth of the newest block. These weights are independent of the target block value used for fee estimation mentioned earlier, the weights are used to get a relevant fee estimate while the target block is used to determine the accuracy of fee estimates.

Bitcoin Core has the API call estimateSmartFee {target block number}, which takes in a target block number and returns the fee rate that should be used if the user wishes to confirm their transaction within that target block. The algorithm then looks at the bucket with the highest fees and verifies that for that fee, the probability of being confirmed in that number of blocks is at least 95%. Remember that a bucket contains a range where the highest fee rate is 10% larger than the lowest fee rate in that bucket. As such, the algorithm starts by looking at the highest fee rate bucket, and checks that if a fee rate in the range of that bucket's fee rates was used for the transaction, that the transaction has a 95% probability it will confirm before or at the target

block. If a fee rate that falls in the highest fee rate bucket does not have a 95% or higher probability of confirming within the target block, then that is very bad news, since even using an extremely high fee is not going to confirm the transaction. This is not really a possible situation, unless miners were intentionally ignoring high fee rate transactions which will definitely skew the algorithm estimates as the algorithm depends entirely on the premise that miners select transactions to include in a block by highest fee rate.

After the algorithm checks the first bucket with the highest fee rates and finds that the transaction has a 95% or higher possibility of confirming before or at the target block, it moves on to the next bucket of fee rates and does the same. The algorithm continues from bucket to bucket checking if the transaction has a 95% or higher probability of being confirmed before or at the target block if it uses a fee rate in the range of that bucket. Once the algorithm finds a bucket where the fee rate has a probability less than 95% for confirming the transaction, it stops and goes back the the previous bucket where the transaction did have a high chance of confirming before or at the target block and uses the median fee rate value from that bucket as the recommended fee rate.

3.2.3 Improvements made in Bitcoin Core post v0.15

The improvements that have been implemented that made the algorithm smarter about making sure that enough data points have been gathered in order to provide an accurate estimate. The algorithm in pre v0.15 was susceptible to extreme outliers, and with the update the algorithm can detect the outliers and if not enough data was found for the fee estimates of a target block then a fallback default fee rate value is returned to be used. The fee rate buckets were also smaller in size, providing more accurate fee rates for target blocks, and compared to the older version of the algorithm, there no longer is a limit of up to 25 blocks for the target, instead the algorithm allows for a fee rate estimate for up to a 1008th target block, which is about one week out. The new update also allows for two types of fee rate estimate modes, a conservative or economical fee estimate. The conservative estimate uses larger historical estimate data which is less susceptible to rapid changes in market fee rates. The economical uses fee estimates for a

shorter historical data period and may provide more cost saving estimates in times of low transaction activity.

The algorithm is repeated three times over three different time ranges and historical data periods. The smallest time range period provides a fee rate estimates for a target block up to 12 blocks using a decay rate of 0.962. The second time range, which uses a decay rate of 0.9952, can provide estimates for a target block up to 48 blocks out. The last period, which uses a decay rate of 0.99931, can provide fee rate estimates for a target block up to 1008 blocks. Along with the different decay rates for each fee rate estimate mode, the threshold for probability of confirmation for a target block changes as well. The smallest time range for confirmation uses a 60% threshold for the probability, the second uses an 85% threshold and the last uses a 95%.

Another major improvement in Bitcoin Core post v0.15 is that the fee rate buckets are no longer spaces at 10% interval increases, but instead use 5% interval changes. This creates more fee rate buckets which will provide more accurate fee rate estimates for target blocks. An interesting modification that was also made is that if a success threshold fails to land in a certain target bucket, the algorithm will attempt to get back above the threshold by adding more data points from the lower fee rate bucket. Along with the 5% fee rate change for a buckets fee rate range values, this modification allows for extreme outliers to carry a lesser weight in their bucket and would skew the data points much less. The last major update is that transactions that failed to confirm due to low fees or other reasons are now considered as part of the fee estimation logic, which leads to more accurate fee rate estimates that will help users avoid having their transactions get stuck.

3.2.4 Advantages

The largest advantage of using this approach is that it uses historical data analysis to produce the fee estimate. Since historical analysis can provide insight on what sort of fees are needed to have a transaction included in a block, Bitcoin Core decided to include this fee estimation algorithm as the main method for fee estimation. It is the most widely used method for fee estimation by most Bitcoin transacting parties. As most Bitcoin users use this fee

estimation method, that means that most users will have similar estimates for attempting to reach a target block. If the majority of users have similar fees based on historical data then the fee estimate has a very high possibility of landing the transaction in the intended block. There would be very few transactions which compete with the fees set by the Bitcoin Core fee estimation algorithm, thus most fee estimates produced will fall near each other for intended blocks.

3.2.5 Disadvantages

The problem with using this method of fee estimation is that it will not adapt to major changes in network activity. Transactions that have been broadcasted to the network during high network traffic will have many transactions like it competing for space on the blockchain and thus may be delayed for quite some time as more and more transactions enter the mempool with higher fees. Or if the network traffic was suddenly much less than before, then the algorithm will create fee estimates which are too high for the current network state and will cost the users more than necessary. In the case of high network activity then the transactions may be delayed for quite some time due to inadequate fee estimates. Bitcoin Core's algorithm is totally blind to the state of the mempool which can cause unwanted behavior, unlike the fee bumping algorithm which produces fee estimates that are highly dependant on the current state of the mempool.

One example of this algorithm performing very poorly was during early November of last year, 2017. Due to a new cryptocurrency with the same hashing algorithm having a moment of profitability that was higher than that of Bitcoin, many miners switched over to mining the other currency. This resulted in the hash rate for Bitcoin decreasing by 50% and blocks being produced on average every 20 minutes compared to every 10 minutes. That meant that there would be less block space available and less transactions could confirm. In this scenario, the Bitcoin Core algorithm did not account for the state of the network, many transactions ended up being stuck and inadequate fees were used. The fee rates were much higher than the historical data had predicted due to space on the blockchain becoming more scarce. The core fee estimation algorithm failed to adapt to the network activity and was completely useless at this point and was resulting in users vastly underpaying in fee costs.

4. Hypothesis

After analyzing the two techniques of generating fee estimates for use in transactions, one using historical data [8] and the other the mempool state [2], this work will discuss a different approach which uses the mempool along with a weight vector to predict the necessary fee rate capable of confirming a transaction in the intended time frame. The technique will use one snapshot of the mempool to generate the fee estimates. The reason one snapshot of the mempool is needed is because we are targeting generating fee estimates specific for multisig transactions. These transactions require that multiple parties sign a transaction and if the private keys are held my multiple entities it becomes difficult to sign the transaction more than once like in the technique discussed earlier [2]. In order to use replace by fee for multisig transactions while the private keys are held by multiple parties, one must pre-sign multiple transactions with different fee rates so that the transactions can be updated as the state of the mempool changes [15]. As a result, multisig transactions cannot fully take advantage of the replace by fee feature. The other method, which Bitcoin Core uses for generating fee estimates [8], can cause users to overpay or underpay quite easily when the historical data becomes irrelevant to the current state of the network. This is because the algorithm is completely blind to what's in the mempool [16].

By using the perceptron machine learning algorithm to generate weight vectors for different block target ranges, it becomes possible to produce fee estimates that save Bitcoin users from overpaying on fees. For multisig transactions which cannot fully utilize the replace by fee feature, this approach can help produce more accurate fee estimates. The machine learning algorithm adjusts overtime to trends in the mempool since it modifies the weights overtime as more data passes through based on a value called the learning rate [20]. It is possible to just use the mempool content and the precalculated weights that change according to the fee trends to predict fee rates necessary for the confirmation of a transaction.

The weighted values for fee rate estimates of future blocks will be derived from mathematical analysis of historical mempool snapshots. By taking snapshots of the mempool and

updating the recorded transactions to include confirmation time, it is possible to analyze the relationship between the fee rate, confirmation time, and miner preference in snapshots to create weight values that can be used to generate fee estimates. The miner preference is assumed based on how the fee rate of a transaction compares to other transactions in the mempool. Once the weight values are derived, the algorithm will be able to combine the mempool state with the machine learning model's weights to generate the fee estimates.

Snapshots of the mempool can be rebuilt using records of mempool contents over time. An array of transactions is the result of rebuilding the mempool and the array is sorted by fee rate. The array is then looped through while using a "size" variable to keep track of the cumulative vSize of the transactions. Every time the cumulative size returns 0 after the modulo by 1 million is taken, we would have a section of the mempool which adds up to 1 million vBytes that can be considered a block that will be created in the near future. Depending on the size of the mempool, it is possible to create several groups of transactions where we can assume each group will be a future block.

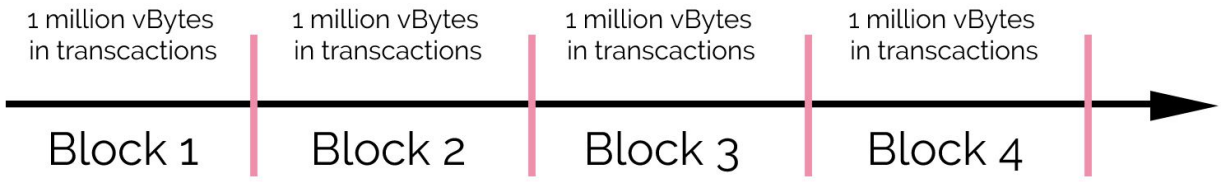

Figure 3: Mempool sectioned into blocks, each slice is a maximum of 1 million vBytes.

The user then selects a target block they would like their transaction to confirm in. The target block number is then taken from the mempool future blocks and the fee rate of the transaction with the lowest fee rate is taken. The fee rate is then incremented slightly and checked against the weights generated by the perceptron algorithm to determine whether the transaction will likely confirm with that fee rate in the intended target block or not. If the transaction with the fee rate is checked against the weights and the weights return true, then the fee rate is returned. Otherwise the fee rate will be increased and checked again over and over until the prediction

returns true. The fee rate determined to confirm the transaction in the intended target block is then returned to the user.

5. Data Collection

The process by which data collection was done was rather a tedious one. Initially when scoping what was needed for data collection, it was intended to use a public API for retrieving mempool content. It quickly became apparent however when trying to use Blockchain.info's public API, that the limits for public use prevented collecting sufficient data. The API capped responses to ten transactions per call and did not support pagination. When other API's were investigated, it became apparent that similar limits were in place and that it would be infeasible to use a third party API for the large amount of data collection necessary for this project. Instead, a Bitcoin node was set up to directly collect the mempool data from the source. The Bitcoin node was operated in pruned mode to reduce the necessary disk footprint. A pruned node does not keep the entire copy of the blockchain but only retains the latest part of the blockchain. A pruned node must keep a minimum of 550 MB worth of blockchain data [21].

5.1 Amazon Web Services Bitcoin Node

Since the mempool state is individual to each node and ephemeral, mempool data had to be collected continuously. An Amazon Web Service EC2 server instance was used to ensure continuous operation of the Bitcoin node. The server instance was set up with Bitcoin Core 0.15 and act as a node on the Bitcoin network responsible for mining new blocks. Since the node would be capable of generating new blocks it would have its own mempool. The server was used to collect the transactions contained in the mempool into a database in addition to running the Bitcoin Core instance. The mempool data was collected by polling Bitcoin Core service through the command line interface that is available as part of the software package.

Along with the EC2 instance, an AWS relational database was set up on the EC2 server instance to which transaction data and fee data was stored. The database was setup with two tables, one of which contains mempool transaction history and the other to store fee estimates

made via Bitcoin Core's fee estimation algorithm at every block height of mempool snapshots. When Bitcoin Core is first run, it synchronizes with the network's current state. To that end, it downloads and verifies the complete blockchain. Initially, an AWS's t2 micro tier EC2 instance with 1 GB of RAM and a 1 GHz processor was used. It took over a week for the node to validate half of the blockchain height with the t2 micro specs. The chain is at the time of this writing over 150 GB in size worth of raw data. When the validation reached about block 320,000, the node became extremely slow at validation and was processing only a few blocks in a ten minute time frame. At that point, the node was validating slower than new blocks were found by the Bitcoin network, meaning that it would never catch up to the chaintip. It was necessary to upgrade the instance to a higher spec system. As such a new server instance was spawned and set up with a 2 Ghz processor and 4 GB of ram. The new server instance quickly synced and validated the chain within two days, which allowed to commence the mempool transaction history data collection.

5.2 Database Setup

In order to record the transaction and fee data of the blockchain to use in the creating a training set for the perceptron machine learning algorithm, the database was setup with two tables. One table captured transaction data, and the other fee estimation data from Bitcoin Core's algorithm. The transaction data table was named mempool_history and its purpose is to capture snapshots of the node's entire mempool at every block height. The table has six columns, starting with transaction id, and timestamp, current chain height, fee rate, transaction size, and transaction confirmation height. When a snapshot of the mempool is taken, only the transaction id and current chain height are recorded, the other fields are filled using another script which will be described in section 5.4 regarding retrieving transaction data. The table is indexed by transaction id, which are unique, and the chain height, that is because when snapshots are taken, transaction ids are checked against the database so that they are not recorded twice. The reason the chain height is indexed is because during the training of the machine learning model, queries are run to retrieve each snapshot of the mempool by chain block height.

	id	txid	timestamp	feerate	currentheight	size	confirmHeight
1	70001	443fb4ada3189a0d05861fcc37e73ccc19db685…	2018-03-15 02:44:59	12	513575	142	513576
2	70002	f795a268277311fe00784ead0292b3201d13b52…	2018-03-15 02:44:59	12	513575	142	513576
3	70003	5a70a4440197f379df143d4686fa88a5535fbda…	2018-03-15 02:44:59	12	513575	142	513576
4	70004	2c6f8ce910a6d51b317d5f39e126478488b192f…	2018-03-15 02:44:59	12	513575	142	513576
5	70005	cf9e5e1c011cceea5bafe8bcfb11236b276f775…	2018-03-15 02:44:59	12	513575	167	513576
6	70006	488aede43403aadbecdaabcb51747b58cc76d69…	2018-03-15 02:44:59	12	513575	167	513576
7	70007	5563a417ae70060303d34ce6b045933f1a0a862…	2018-03-15 02:44:59	12	513575	167	513576
8	70008	87f912e1e3252e205117b957931b00e2115a309…	2018-03-15 02:44:59	12	513575	167	513576
9	70009	d858f571d568d05c8d34dcf3e636da57baf60c6…	2018-03-15 02:44:59	12	513575	167	513576
10	70010	09effc76d0d72d3962e6bad48539806e94d6707…	2018-03-15 02:44:59	12	513575	167	513576

Figure 4: Mempool history database table segment of 10 entries.

The fee_estimates table contains four columns. The block target, block_height of fee estimate, fee rate per kilobyte and timestamp. The block target field is an indicator of what confirmation time is intended by the recorded fee rate. Block height is the moment in time in which the fee estimates were taken. The fee estimate is taken as a fee rate per kilobyte in BTC value not in Satoshis, unlike the mempool history table which stores the fee rate in Satoshis. The table is indexed by chain height for quick fee estimates retrieval for a mempool snapshot being analyzed.

	id	block_target	timestamp	block_height	fee_rate_kb
1	1010	3	2018-04-05 14:47:54	516753	0.00005609
2	1011	4	2018-04-05 14:47:54	516753	0.00005047
3	1012	5	2018-04-05 14:47:54	516753	0.00005047
4	1013	6	2018-04-05 14:47:54	516753	0.00005047
5	1014	7	2018-04-05 14:47:54	516753	0.00005047
6	1015	8	2018-04-05 14:47:54	516753	0.00005047
7	1016	9	2018-04-05 14:47:54	516753	0.00004985
8	1017	10	2018-04-05 14:47:54	516753	0.00001765
9	1018	11	2018-04-05 14:47:54	516753	0.00001185
10	1019	12	2018-04-05 14:47:54	516753	0.00001

Figure 5: Fee estimates database table segment of 10 entries.

5.3 Recording the Mempool Transaction History

Bitcoin's command line tool contained a method by the name of 'getrawmempool' which prints to console a list of all transaction ids for the transactions currently in the mempool. The easiest method of capturing that data was to write it to a file and reading that file to retrieve the transaction ids. Python was the language of choice for retrieval of the mempool transaction content. A system command is issued every 2.5 minutes to write the current mempool transaction

ids to a file. The file is then read via the same python script and parsed into a string array with the entirety of the mempool's transaction record. The block height of the chain is also retrieved using BitGo's public API at the time of the snapshot of the mempool is taken. The reason this is done is so that the time taken for the transaction to be confirmed can be taken into account, as it relates directly to the fee paid by the transaction creator. In order to not register the same transaction over and over again with each snapshot, the script queries the mempool_history table for the transaction ids it just retrieved from the mempool. Since the mempool_history table is indexed by transaction ids, this is a relatively cheap operation. The database query checks the database for the transaction ids that were read from the mempool, a list is returned with the ones currently with a database record. The list of transactions currently in the database is converted into a set. The list of all transaction ids currently in the mempool is then looped through and checked against the set for whether a database entry exists or not. In the case a database entry does exist, then the loop continues to the next transaction. In the case that a database entry does not exist, then the transaction id is added to the database along with the current chain height in order to mark the period of time in which the transaction was first seen in the mempool.

In addition to entering each transaction from the mempool into the database, the fee estimates based on Bitcoin Core's fee estimation algorithm are produced using the Bitcoin command line tool command estimateSmartFee. The command is looped through with a different target block each time to produce fee estimates for target blocks 2 to 19. The estimates are retrieved once every new block is created and stored in the fee_estimates table of the database along with the block height and target block. The fee_estimates table is indexed by block height making it extremely efficient at returning fee estimates for a certain period of the blockchain. The need for fee estimation data was with the intent of comparing the fee estimates of this works algorithm and Bitcoin Core's fee estimation algorithm.

The fee estimate data from Bitcoin Core is to be used as a baseline for this work's fee estimation technique. This will be done by checking transactions in a mempool snapshot and comparing the fee rates used against the baseline to determine what target block was intended by the transaction. Then the fee estimates built using this works algorithm will be generated as well

and used to determine what target block was intended by the mempool snapshot's transactions. The fee rates will then be compared and confirmation time checked to determine how accurate each were for achieving the intended target block. This will be the technique used to determine the effectiveness of each algorithm at determining the optimal fee rate for achieving confirmation within the intended time frame.

5.4 Filling in Missing Data

The data retrieved up until this point only contains transaction ids and when they were first seen in the mempool. Fee estimate data was recorded as well for comparison against this work's fee estimation technique. But in order to build a training set that can be used by the machine learning model that will aid in predicting fee rates based on target blocks, it is necessary to record every transactions confirmation height, fee rate, and vSize. The transaction ids were stored into the database along with the current chain block height, as a method of determining at what point in time the transaction entered the mempool. This created a large data set of database entries with only transaction ids and the block height they were first seen at. The database entries were unique in the sense that each transaction only had one database entry. The reason this was done was in order to save on storage space and make the table more efficient in terms of querying since entries would be indexed by transaction ids.

This raises an issue however related to rebuilding the mempool to use during the creation of the of the training dataset. This will be discussed in more detail in section 6.1. In order to prevent the snapshot script from attempting to create a database entry for a transaction that already exists in the database, the algorithm would take a snapshot of the mempool and then reformat the transaction ids to use them in a database query. The script would query the database for the transaction ids it saw in the mempool and the query would return the ids that already contain a database entry. The transaction ids returned by the database are stored in a set data structure to be quickly referenced later. The script would then get the current blockchain height and loop through all transactions in the mempool snapshot, checking if they already have a database entry by checking for them in the set. If no database entry exists then the transaction id

is logged with the current chain height, otherwise the script ignores it and moves on to the next transaction it sees in the mempool snapshot.

The database now contains unique transaction ids with an idea of the first time they were seen in the mempool. These entries are then processed and queried by another script. This script is meant to retrieve the transaction data after confirmation which was called getTxData solely meant to check blockchain data for the transactions that are in the database. The script uses BitGo's SDK and BitGo server endpoint which retrieves transaction data from BitGo's Bitcoin full node local copy of the blockchain. Full nodes is the term used for Bitcoin network nodes that keep an entire copy of the chain, hence they have the ability to view an entire transactions details at any point in time upon request. The script checks for any transaction in the database mempool history table that does not have a height for when it was confirmed. It then loops through all these transactions and sends API requests to get the transaction details from the full nodes. The API request returns the block the transaction was confirmed in and the vSize of the transaction, which are promptly added to the database transaction entry. After the getTxData script is completed, all the transaction entries in the database contain the data necessary to build the training set. In the case that transaction data was missing from the blockchain altogether, which is possible if a transaction becomes invalid due to another transaction replacing it using RBF (replace-by-fee) which was discussed earlier in section 3.1, then a flag was set on the confirmation height which set it to -1. The learning algorithm ignored database entries which had the confirmation either set to NULL meaning they were not processed yet by getTxData or they have a -1 flag meaning the entry is missing critical data.

Script code is available in appendix B, under getTxData.js. The script is a node js script, which assumes BitGo's SDK is already available in the node environment. In addition to having the SDK, an environment variable containing a developer access token is needed to access the API endpoint. The access token needs to be for a BitGo admin account as it is used to access BitGo's internal nodes not available for public use. The developer token is also locked to the IP address of the AWS instance running the script.

6. Training the Machine Learning Model

The machine learning algorithm used for this research effort is the perceptron classification algorithm. The algorithm is inspired by how information is processed by a single neural cell. The cell accepts input signals via dendrites which then pass an electrical signal to the cell body. This concept is applied within the perceptron algorithm when a training set is used as input signals that are then weighted into a linear function called the activation function.

$$activation = sum(weight_i * x_i) + bias$$

The activation is then transformed into a prediction using the transfer function such as a step transfer function.

$$prediction = 1 \; if \; activation >= 0 \; else \; 0$$

The perceptron algorithm then can classify a data point into one of two classes, 0 or 1. Whether it is true or false, or whether it will activate the neuron or not. A linear equation can be used to separate the two classes in a 2D plane. The algorithm is closely related to a linear regression or logistic regression algorithm that makes predictions in a similar way using a weighted sum of inputs [22]. The weights of the perceptron algorithm are estimated using a training data set using stochastic gradient descent. The weighted values are then built into a vector which is used to classify data points. Gradient descent is a process by which the user minimizes a function by following the gradients of the cost function. This requires knowing the form of the cost and the derivative so when given a point you are able to find out the gradient and move in that direction. In machine learning, this technique used to evaluate and update weights every iteration through the training set is called stochastic gradient descent, where the error is minimized against the training data.

Optimization of the weights happens by showing the model a training instance one at a time, where the model makes a prediction and the error is calculated. The model is then updated to reduce the error next prediction. The procedure is used to find a set of weights which can be

used for prediction with the smallest error in the model based off of the training data. Every iteration uses the latest weight to as well as a learning rate and error to generate a new weight.

$$weight = previous\ weight + learning\ rate * (expected - predicted) * x$$

The learning rate is a configuration that is set by the model creator while the difference between expected and predicted value is the prediction error. The learning rate is a way of telling the perceptron algorithm how much to weigh in the error every training instance iteration. X is the input value of a training data instance.

6.1 Building the Training Set

The training set used for this model was built using snapshots of the mempool over the period of a few days. When building the training data set, each data instance included a block number and what slice of the block they took up. Transactions were sorted by fee rate and split into blocks based on the vSize of the transaction, since every block takes up 1,000,000 vBytes, it was possible to split the mempool into blocks and give them further resolution of about 10 slices per block. This creates a vector for each database entry, since the training algorithm needs training instances to be in the form of a vector array. The idea behind the algorithm is to allow it to classify whether a transaction will confirm or not based on the fee rate and some context regarding the mempool. It was decided to have the vector contain four pieces of data for the training instances. The fee rate, the future block it was assigned when the mempool was split into blocks, the slice it was part of for that block, and whether the transaction confirmed or not in that block. Determining whether the transaction confirmed in the block it was predicted to fall into when the mempool was split into blocks was done by comparing confirmation height with the block number it should have confirmed in added to the current chain height.

The training data set was created by recreating the mempool overtime using the snapshots. Since the snapshots taken of the mempool only included new transactions added at the new height and did not account for older transactions still in the mempool, this meant that it was necessary to track transactions that have been confirmed and those that are still present at the new height. In order to create a virtual mempool at every new block of the chain, a global

variable was made called mempool which was used by the method generate_learning_set. The mempool global variable is a dictionary where the key is the transaction id and the value is the database entry containing the confirmation height, fee rate and vSize of the transaction. The training set is also available as a global variable, and for every block height, the mempool contents would be used to add a new training instance to the learning instances dataset.

The method generate_training_set takes in two variables, the previous block height and the current one. The reason for this was because the data set was missing some block heights, even though confirmations were consistent, which would cause the mempool to overflow with transactions it would assume were unconfirmed but in fact were. The first step of generate_training_set is to get all database entries for transactions that were first seen at a the current height. Then for every one of those transactions, it would add them to the mempool global object. The next step was to query for all transactions that were confirmed between the previous block height and the current chain height. The confirmed transactions are then looped through and removed from the mempool global object as they are now part of a block and no longer awaiting confirmation.

At this point of the data processing, the mempool is converted into a list of dictionary objects, where each object is a transaction database entry object. The list is then sorted on the fee rate, which means we have a long list of transactions that can be used to represent an estimate of the next set of blocks that will be mined. For every transaction in this mempool transaction list, the block it will land is determined by the size of the mempool up to this block. A total size variable is used to sum up all the transaction vSizes and that value is used to determine where the transaction sits in terms of what block and slice of the block it would confirm in. Two constant variables are used, called BLOCK_SIZE and SLICE_SIZE, where the block size is equal to 1,000,000 vBytes and the slice size is equal to 100,000 vBytes. When looking at the accumulating vSize of the mempool while looping through the transactions. The current transaction vSize is added to the total size, which gets divided by the block_size constant to determine what block the transaction would sit in. The total size is also divided by the slice size and modulo 10 for 10 slices per block, and that would give us the slice number. A constant of 1

gets added to the slice and block number since, the block number will be confirming at block_height plus 1 more block. The last field that gets calculated for each transaction is whether it hit the target block or not, which is calculated by looking at the block height it was estimated to land in and compare that with the actual confirmation block height. If the confirmation block height is equal to or less than the estimated confirmation block height, then the transaction has hit its target. The training instance then includes the four fields fee rate, estimated confirmation block, estimated block slice, and whether it landed or not based on those estimates in the form of a vector. The code that builds the training instances is in appendix C.

Once the training instance was built, it was added to a large array of training instances which was called the training set. The training set is then passed through to our training model which is part of a class called learn in learn.py. The learn class contains several methods, where the constructor takes in the learning data set in the form of an array of vectors upon instantiation. Method train_weights is then called which uses the weights global variable to run the gradient descent calculations across the training set. The gradient descent calculation uses the predict method which also uses the weights global variable to make a prediction. In the case the prediction and the expected value are the same, not change is made to the weights, but in the case they are not the same then the error is calculated and the weights updated. The learn class is available in appendix D.The learn class is used by the processData python script which generates the training data set.

6.2 Validating Data Set

Below are some charts which summarize the data set used to teach the machine learning algorithm. In chart 1, it is possible to see the number transactions that were in the virtual mempool built from our snapshots. We can compare our records to the mempool of a third party (Blockchain.info [24]) at every block height registered in chart 2. When compared to Blockchain info's mempool size over the same time span as our node's mempool we see that it has the same trends. The reason the comparison is shown is to validate our algorithms technique of recreating a virtual mempool post data collection.

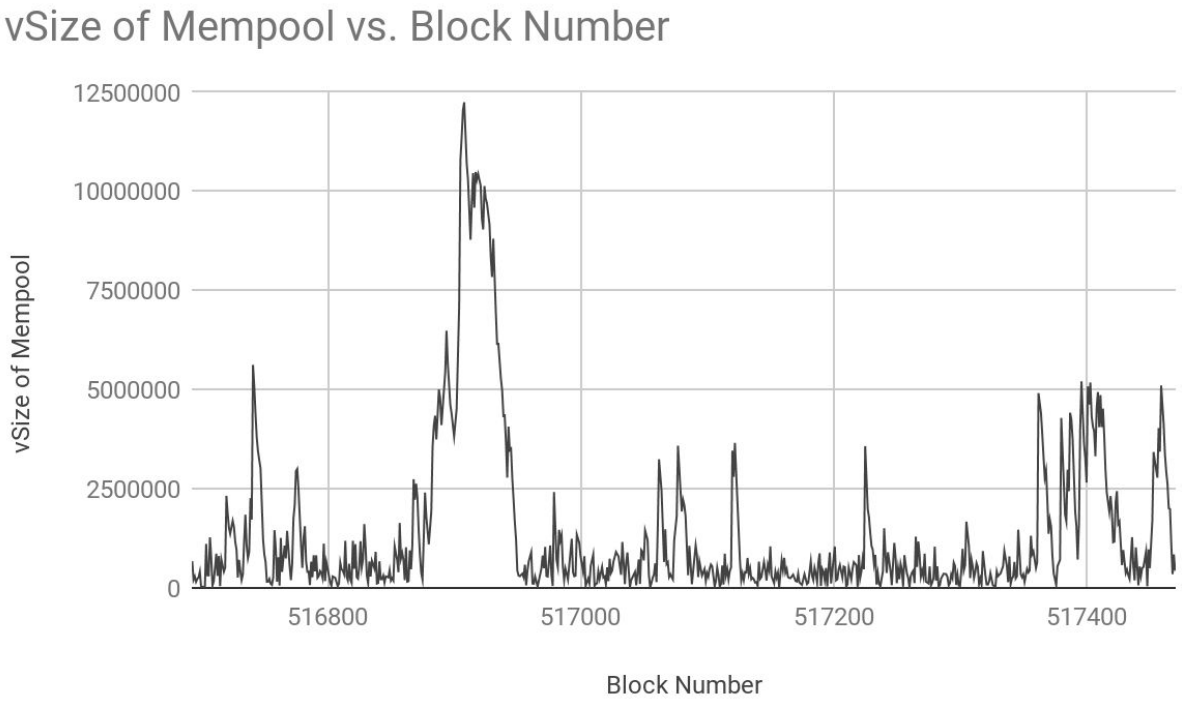

Chart 1: vSize of our mempool at certain block heights.

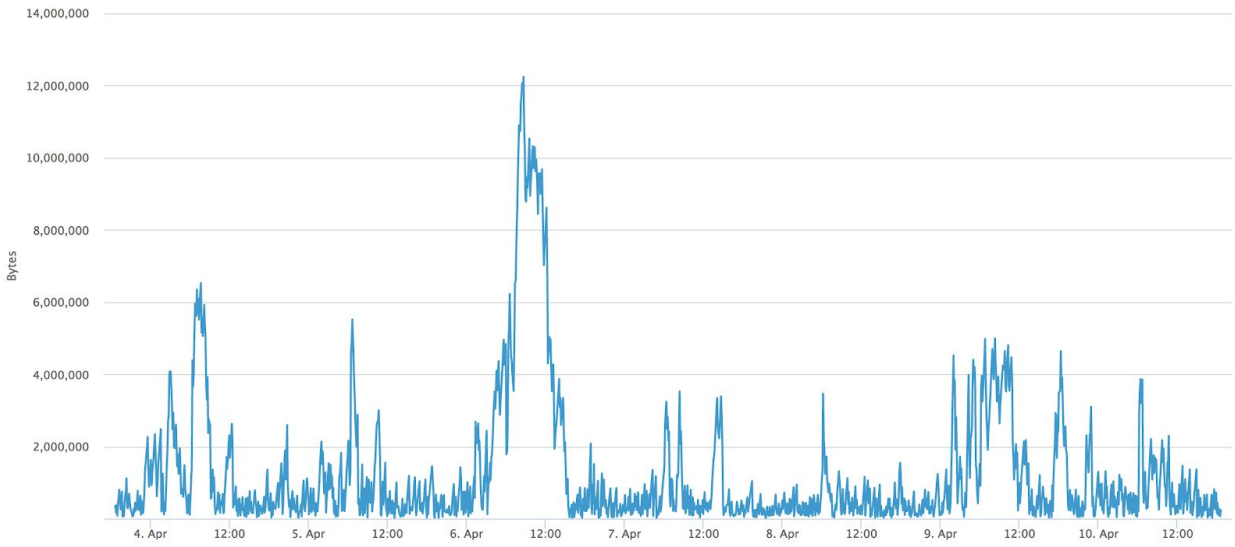

Chart 2: vSize of the mempool of Blockchain Info's Bitcoin nodes [24].

In chart 1 it is possible to see the vSize of the mempool at every block height. The numbers in chart 1 are a good indicator of how full the mempool was at that block height. The larger the vSize, the more blocks that can be generated using the mempool contents. Which means that if the mempool contained a vSize of 4 million bytes, then it is possible to create a training set based off of transactions with 4 different estimated confirmation blocks. When there are more transactions in the mempool that can serve as training data for higher target blocks, we have more data to train our weights for higher target blocks. So if the mempool contained transactions that can fill up to 20 blocks, then weight groups with a higher target number will have training data to use from that mempool snapshot. If the mempool had less amounts of transactions that can only fill up to 2 blocks for an extended period of time, then the weights between blocks 1 and 4 will have more training data to learn from while other weights will remain unchanged for that extended period of time and will have less training data to use. It's also possible to compare the number of transactions that other Bitcoin nodes on the network contained with the one our mempool contained and it is evident that the number of transactions in the mempool on our node also carry a similar trend to the number of transactions on other nodes on the network. From these two charts, it can be concluded that our data set created from our Bitcoin node and used for training the machine learning model is consistent with the rest of the Bitcoin network.

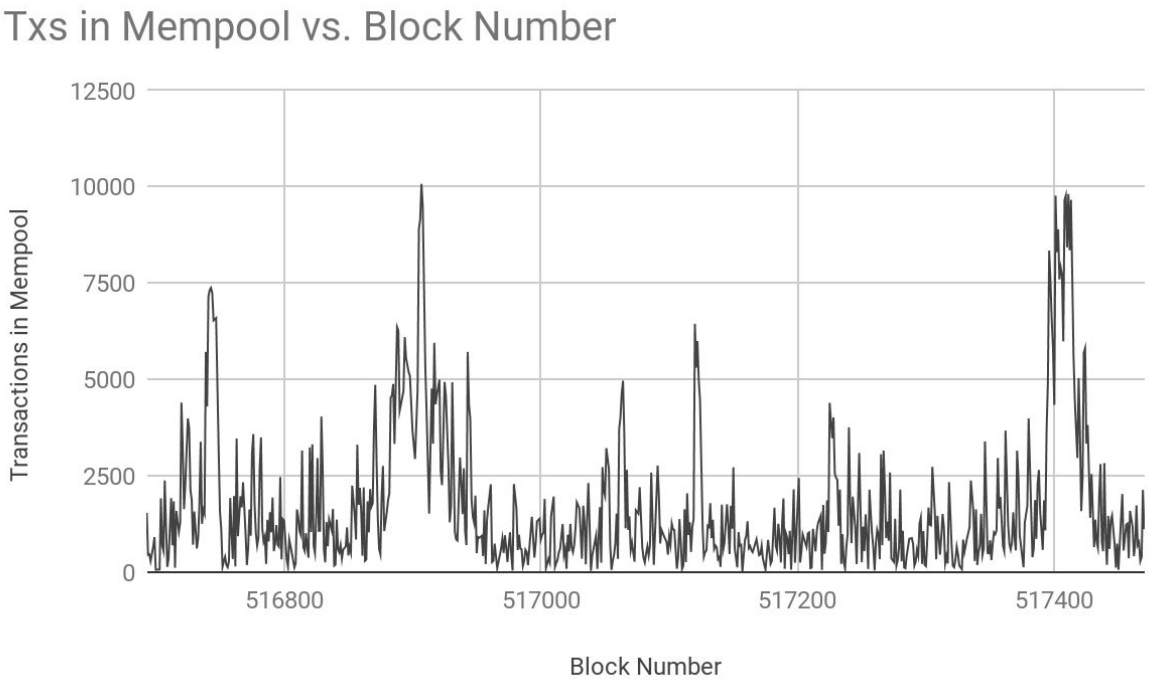

Chart 3: Number of transactions in our mempool at certain block heights.

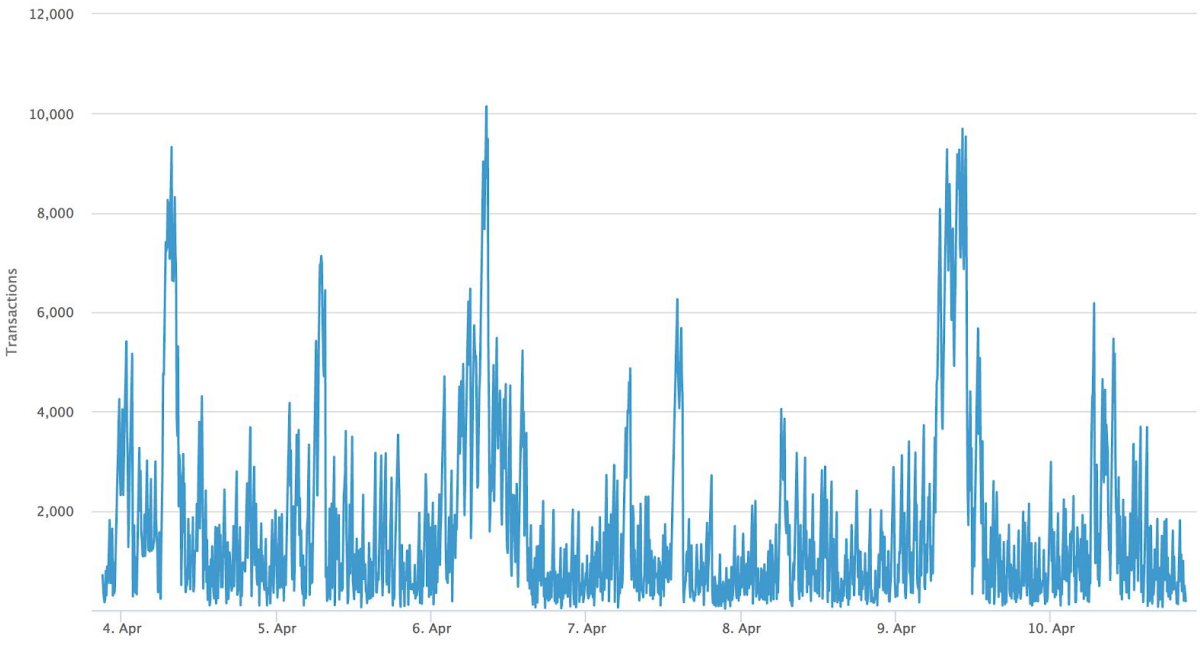

Chart 4: Blockchain Info's transactions in mempool overtime. [24]

6.3 Training the model

In order to provide different weights for different target blocks, four different training sets were built using the data collected from the mempool. Each with a different target block in mind. The the data collected was separated into four different classifications which were target blocks in range 1 to 4, 5 to 8, 9 to 12 and 13+. The four different classifications allowed for four different weights to be calculated. Each weight vector was created using a data set containing transactions with a confirmation time in the range of target blocks for the weight. If a transaction confirmed within 6 blocks from the first time it was seen in the mempool then 6 training instances would be built from it. The first instance would claim the confirmation time was 6 blocks and would be used to build the weight vector for confirmation time between 5 to 8 blocks. The second instance would be 5 target blocks and it would be used to build the weight vector with a confirmation time between 5 and 8 as well. The third instance would be that the transaction would confirm 4 blocks later and so it would be used as a training instance for the weight vector of range 1 to 4 target blocks. This continues until confirmation time of one target block, giving training instances for each target block range.

The training instances were added into four different lists. Each list was used as the training data set for one machine learning model. A different model was created for the four different target block ranges and trained with the training set of that target block range. As a result four different weight vectors were generated, each of which would be used to predict if a transaction would confirm or not depending on the intended target block.

7. Results

In terms of classification, the results showed that the linear separator to a certain extent was capable of distinguishing whether a transaction will confirm or not in the block it was estimated to confirm in. The estimated confirmation block was determined based on the technique discussed in the hypothesis, section 4, and described using figure 3. Which was by sorting the transactions in a snapshot of the mempool by fee rate, and then splitting the list of

transactions into blocks of 1,000,000 vBytes. It is assumed that the blocks would then form the next blocks that are appended on to the blockchain. Based on what block a transaction was part of, a training instance was created for the weight of the target block equal to that transactions confirmation time. All training instances were fed into different learning models depending on their confirmation time. Four models were used, each specific to a different range of confirmation times for transactions. The four models resulted in a set of four different weights and biases. Using those weights to predict whether a transaction would confirm or not proved possible to a certain extent. The weights trained for estimating whether a transaction would confirm within 1 to 4 blocks was with the highest accuracy of 90.72%. This was most likely due to the fact that the next blocks to be generated will most likely contain many of the transactions that are currently in the mempool snapshots with the highest fee rates. Moving beyond the first few blocks drastically drops the accuracy of classification, since the weights and bias used for transactions that confirmed within 5 to 8 blocks had an accuracy of 33.09%. For transactions that confirmed between 9 to 12 blocks, the accuracy of the weights was 17.45%. And for confirmation after 13+ blocks since the transaction broadcasted, the accuracy was a low 11.43%.

In chart 7 below, the model was run against our snapshots of the mempool in order to show the difference in fee estimates. The chart shows the comparison of Bitcoin Core's fee estimation algorithm against our own and against the lowest fee per block. The chart is highlighting fee estimates with a target block of 2, since Bitcoin Core's algorithm starts estimates at a target block of 2. An estimate for a target block of 2 has the highest confidence in terms of confirming in the intended time frame. The reason for this method of comparison is to show that our fee estimation was producing fee estimates lower than Bitcoin Core's fee estimates but at certain points would produce estimates that were magnitudes higher than Bitcoin Core's estimates. The reason this was occuring was because the linear perceptron model would need certain components of a transaction instance to be higher than others in order to activate the model and produce a signal saying that the transaction will confirm. In this case, some instances needed a very high transaction fee in order to be predicted to confirm within the intended target block.

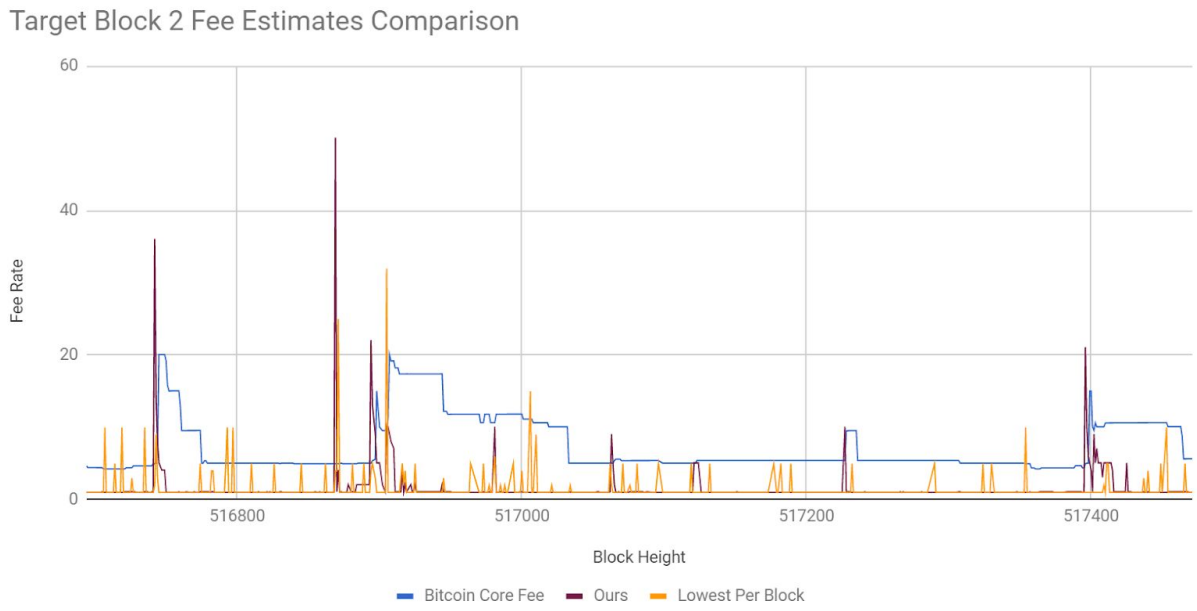

Chart 5: Our fee estimates vs Bitcoin Core's vs the lowest fee rate in the block.

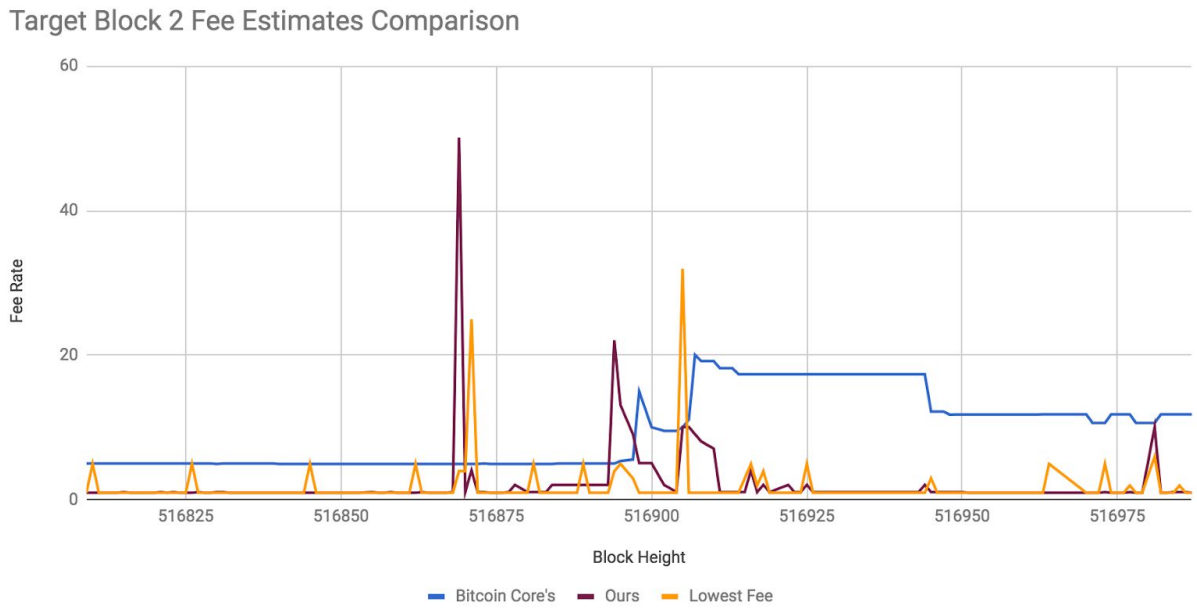

Chart 6: A zoomed in portion of chart 5 showing a more detailed view of fee rate differences.

Block Height	Target Block	Bitcoin Core	Ours	Lowest Fee in Block
516738	2	4.686	1	1
516739	2	4.686	1	1
516740	2	4.686	1.1	1
516741	2	4.985	1.1	1
516742	2	4.987	36.1	2
516743	2	5.045	14.1	9
516744	2	6.99	8.1	3
516745	2	20.101	5.1	1
516747	2	20.1	4.1	1
516749	2	20.1	4.1	1
516750	2	19.21	1.1	1
516751	2	15.862	1	1
516752	2	15.05	1	1

Table 1: A sample of the data plotted in Chart 7 showing the comparison in fee rates.

Table 1 shows an example of best case scenario fee estimation by our fee estimation algorithm and also a worst case scenario. The best case scenario is the case where the predicted fee rate is much lower than Bitcoin Core's fee estimate. And the fee rate is either slightly higher or the same rate as the lowest rate used by a transaction in that target block. The worst case scenario however, is when our estimated fee rate was magnitudes higher than Bitcoin Core's fee estimate while the lowest fee rate in the target block is lower than the Bitcoin Core fee estimate as depicted in chart 6. What can be seen from the above table, is that fee rates created by our algorithm did not seem to always provide fee rates that were lower than Bitcoin Core's fee estimation algorithm. In addition to that, the fee rate estimates at times were much higher than both Bitcoin Core's fee estimates and the lowest fee in the target block. This means that the algorithm was not providing estimates that were meant to save the user funds on transaction fees consistently.

8. Conclusion

The original intention of this work was to create a fee estimation algorithm by which the user is able to produce fee estimates for an intended target block based on a single snapshot of the mempool. The fee estimates would be created using precalculated weights for a linear perceptron classification algorithm. The problem with this approach is that the fee estimation algorithm was the linear perceptron classification model using a single node training model for each range of target blocks. It was only realized through experimentation and analysis of the results that the approach was incapable of providing accurate fee estimates on a per block target basis. The is predominantly due to the distribution of fee rate vs confirmation time of Bitcoin transactions, shown in chart 5 and chart 6 below. The charts show the count of transactions vs the fee rate used which were confirmed within 1 block for chart 5 and 2 blocks for chart 6. It should be noted that in both charts, any transaction with a fee rate of 50 satoshis per byte or higher was excluded due to their wide distribution and infrequent count. The majority of transactions had paid a fee rate less than 50 satoshis per byte and therefore it was acceptable only looking at those. When plotting the number of transactions per fee rate that have confirmed within just a block or two since broadcasting, it becomes clear that the linear separator would have difficulty classifying transactions based off of fee rate alone.

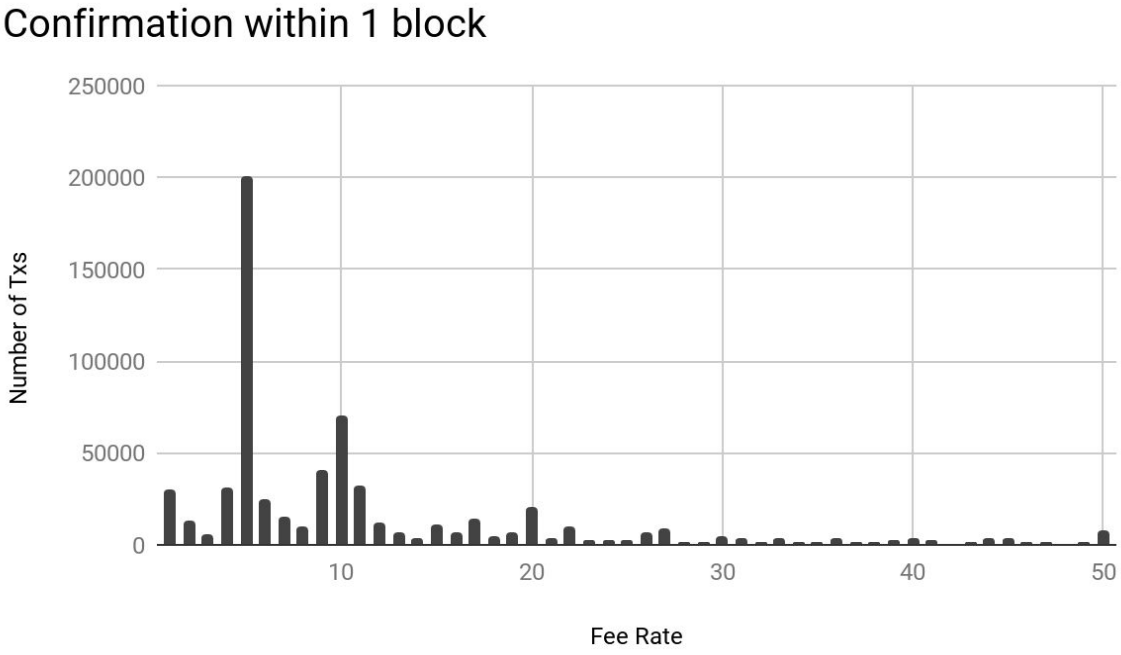

Chart 7: Count of transactions confirmed within 1 block vs fee rate.

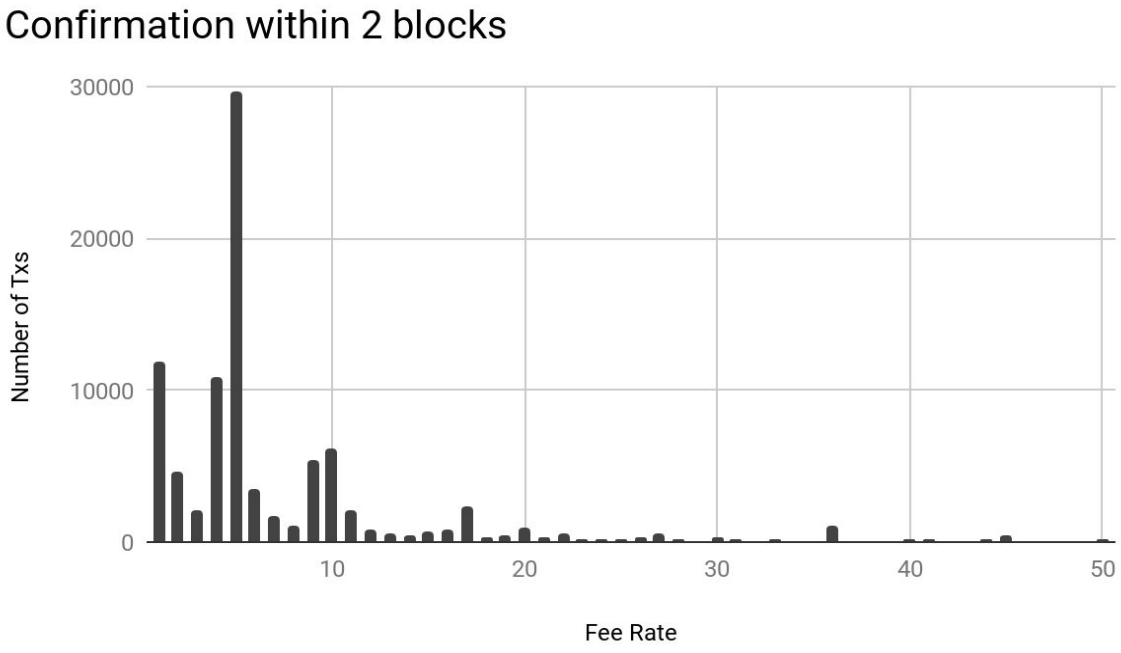

Chart 8: Count of transactions confirmed within 2 blocks vs fee rate.

That is why the training set for the linear separator included context regarding where the transaction places in the mempool in terms of miner preference. The transactions were sorted by fee rate, then the estimated target block of a transaction and the slice it would occupy were included in the training instance. Those fields were added in order to give the classifier more fields to build the weight vectors upon. However, the linear perceptron machine learning algorithm with a single node was not capable of creating a set of vectors capable of accurate predictions. The main reason for that was lack of learning capacity of the model. Our fee estimation algorithm used four nodes, each with a different set of weights and biases in addition to their very own training set. However, for a model using one node it is difficult to generate a classification metric for a problem consisting of a nonlinear fee rate vs confirmation time distribution. Fee rates depend on demand for confirmation on the blockchain. Therefore, it is best for a learning model to solve the problem of fee estimation for Bitcoin using deep learning. A multilayer perceptron machine learning model is an example of a deep learning training model where hidden layers of nodes, each with their own sets of weights and biases would most likely be better suited at determining transaction confirmation times. The training model used by this work was a single node machine learning model that was incapable of providing fee rates that would save the user funds in transactions consistently and instead produced unpredictable behavior that makes the algorithm an unreliable source of Bitcoin fee estimates.

9. Future Work

The main issue involved with our technique was that it was a single layer linear perceptron machine learning model using a single node with one set of weights and a bias. The activation function of such an algorithm is a linear line that splits data points into two sides. One side of the line will consist of data points that do not activate the the model and will return a 0. The other side will contain the data points that will activate the model and will return a 1. The result of the activation function equal to 0 refers to a transaction that will not confirm and a 1 refers to a transaction that will confirm. Fee estimation is a multidimensional problem requiring a solution that can factor in the different aspects that impact whether a transaction will confirm or not in the intended target block. The biggest factor that impacts confirmation time is the fee

paid in the transaction to the miners, but it relates directly to the other transactions competing against it. Looking at what fees are being paid by other transactions in the mempool and selecting a fee rate that competes for the target block was the original intention of this work. However, the oversight was the use of a single node perceptron machine learning model. It is difficult to produce a model that can solve a multi dimensional problem accurately with a single node machine learning model rather than a model containing many nodes in a network.

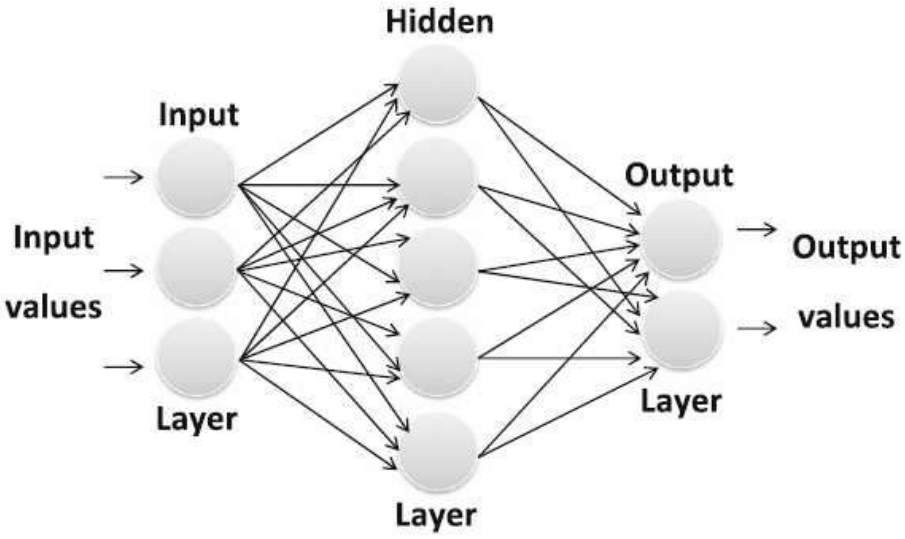

Figure 6: Multilayer perceptron example network [27].

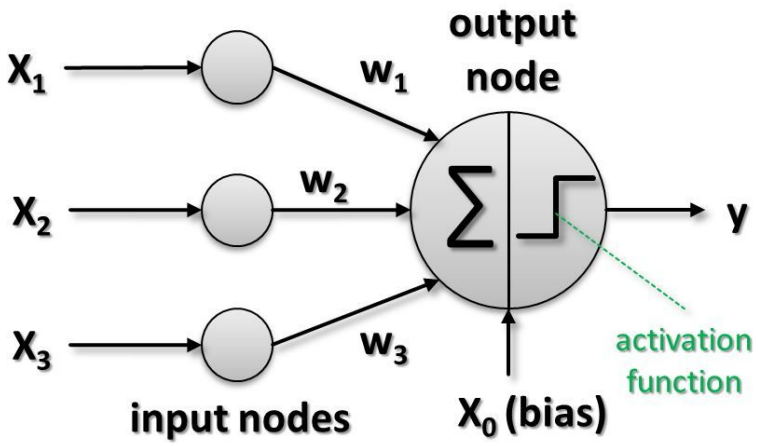

Figure 7: Single node linear perceptron model used in this work [28].

In figure 6, a multilayer machine learning model is shown, where many nodes can exist in a hidden layer. It is possible to add as many nodes as one wants into the hidden layer. The number of nodes impacts performance of the model and how much computing power is necessary for creating accurate predictions. It is also possible to have multiple hidden layers where each node, in every layer, uses its own weights and biases and receives inputs from the previous layer. Figure 7 shows what a single node and single layer perceptron machine learning model looks like. This work used four single layer perceptron models that used their own sets of weights and biases. But they were not connected together or being fed the same inputs like in a multilayer model. The four nodes were instead fed separate inputs and trained for different block targets. Using a multilayer model for training against our dataset would use the same input feed for all the nodes in the hidden layer. The weights and biases would be created according to the input feed and the output would be a generated after computations by the nodes in the hidden layer are done using the many different weights and biases.

In order to expand on the work done in this research effort, it is imperative that a multilayer perceptron machine learning model is used. A multilayer perceptron machine learning model is more capable in terms of classification. What multilayer means in this context is that there are multiple layers containing nodes in a network, that process the inputs in different phases. In a simple multilayer neural network, there would be three layers known as the input layer, hidden layer, and the output layer. The input layer has as many nodes as the number of inputs, which pass in the input to the hidden layer equipped with weights and biases. The simplest neural network, has one node in the hidden layer which directly applies the weights and bias to produce a single output. The single layer linear perceptron algorithm used in this work was of the simplest form, consisting of a single node in the hidden layer. Although not intuitive, a single layer perceptron is still a multilayer network, it just consists of a single node [26], while the multilayer perceptron consists of many nodes in multiple layers. The multilayer perceptron uses a network of nodes all equipped with their own set of weights and biases, compared to to the single layer model which only uses one set of weights and a bias [26].

Considering it is possible to train a much larger neural network with many different weights and biases to classify transactions, the output can also be the expected target block. In terms of inputs, the training instances should contain the estimated confirmation block, slice, and fee rate. The training instances would then be passed through the network with the training labels that indicate the confirmation time of the transaction just like we have done with the single layer perceptron algorithm. Optimization can be made in determining the number of hidden layers and nodes per layer, as well as the number of iterations the training instances are passed into the network. Once the model produces a high enough percentage of accurate classification, the same technique used for fee estimation discussed in the hypothesis, section 4, is to be used for producing fee estimates for a target block.

10. References

[1] Narayanan, Arvind, et al. *Bitcoin and Cryptocurrency Technologies: a Comprehensive Introduction*. Princeton University Press, 2016.

[2] Alm, Karl-Johan. *Mempool optimized fees, and the correlation between user costs, miner incentives, and block capacity.* DG-Labs. [Online] Available: https://bc-2.jp/mempool.pdf.

[3] Peter R. *A Transaction Fee Market Exists Without a Block Size Limit.* [Online] Available: https://scalingbitcoin.org/papers/transaction-fee-market-exists-without-block-size-limit.pdf.

[4] "Blockchain." Investopedia. https://www.investopedia.com/terms/d/distributed-applications-apps.asp. Accessed on: Nov 1st, 2017.

[5] "Merkle tree." [Online] Available: https://en.wikipedia.org/wiki/Merkle_tree. Accessed on: Nov 12th, 2017.

[6] "Proof of work." Bitcoin Wiki. [Online] Available: https://en.bitcoin.it/wiki/Proof_of_work. Accessed on: Nov 12th, 2017.

[7] "Difficulty." Bitcoin Wiki. [Online] Available: https://en.bitcoin.it/wiki/Difficulty. Accessed on: Nov 12th, 2017.

[8] Marcos, Alex. "High level description Bitcoin Core's fee estimation algorithm." [Online] Available: https://gist.github.com/morcos/d3637f015bc4e607e1fd10d8351e9f41. April 21st 2017. Accessed on: Oct 24th, 2017.

[9] Biegel, Ofir. "What is the Bitcoin mempool." 99bitcoins.com. [Online] Available: https://99bitcoins.com/what-is-bitcoin-mempool/. Nov 6th 2017. Accessed on: Nov 12th, 2017.

[10] "Bitcoin developer reference." Bitcoin.org. [Online] Available: https://bitcoin.org/en/developer-reference. Accessed on: Nov 8th, 2017.

[11] "Block." Bitcoin Wiki. [Online] Available: https://en.bitcoin.it/wiki/Block. Accessed on: Nov 14th, 2017.

[12] "Miner Revenue per Block." SmartBit.com. [Online] Available: https://www.smartbit.com.au/charts/miner- revenue-per-block Accessed on: Dec 1st, 2017.

[13] "What is replace by fee." TheMerkle.com. [Online] Available: https://themerkle.com/what-is-replace-by-fee/. Accessed on: Nov 17th, 2017.

[14] Wuille, Pieter. "How is a transaction output marked as spent." StackExchange.com. [Online] Available: https://bitcoin.stackexchange.com/questions/28225/how-is-a-transaction-output-marked-as-spent. July 12th, 2014. Accessed on: Nov 16th, 2017.

[15] Lopp, Jameson. “The challenges of Bitcoin transaction fee estimation.” Blog.BitGo.com. [Online] Available: https://blog.bitgo.com/the-challenges-of-bitcoin-transaction-fee-estimation -e47a64a61c72. May 10th, 2017. Accessed on: Oct 24th, 2017.

[16] Newbury, John. “An Introduction to Bitcoin Core Fee Estimations.” BitcoinTechTalk.com. [Online] Available: https://bitcointechtalk.com/an-introduction-to-bitcoin-core-fee-estimation-27920880ad0 Sept 12th, 2017. Accessed on: Nov 19th, 2017.

[17] Newbury, John. “What’s New in Bitcoin Core v0-15.” BitcoinTechTalk.com. [Online] Available: https://bitcointechtalk.com/whats-new-in-bitcoin-core-v0-15-part-2-41b6d0493136. Sept 15th, 2017. Accessed on: Nov 19th, 2017.

[18] “Pubkey Script.” Bitcoin.org. [Online] Available: https://bitcoin.org/en/glossary/pubkey-script. Accessed on: Nov 30th, 2017.

[19] Ian Wong, Joon. “One metric is far more important to Bitcoin users than its price.” QZ.com. [Online] Available: https://qz.com/1127817/bitcoin-cash-bch-price-could-lead-to-bitcoin-death-spiral/. November 13th. 2017. Accessed Dec 2nd, 2017.

[20] Brownlee Jason, “How to implement the perceptron algorithm from scratch in python.” MachineLearningMastery.com. [Online] Available: https://machinelearningmastery.com/implement-perceptron-algorithm-scratch-python/. November 2nd, 2016. Accessed March 12th, 2018.

[21] Buntinx, Jean-Pierre. “Pros and cons on Bitcoin block pruning” news.bitcoin.com. [Online] Available: https://news.bitcoin.com/pros-and-cons-on-bitcoin-block-pruning/. Accessed on: March 17th, 2018.

[22] Song, Jimmy. “Understanding Segwit Blocksize.” Medium.com. [Online] Available: https://medium.com/@jimmysong/understanding-segwit-block-size-fd901b87c9d4. July 3rd, 2017. Accessed on: March 19th, 2018.

[23] Korsakov, Igor. “Bitcoin replace-by-fee guide: fix stuck transactions, do double spends.” Medium.com. [Online] Available: https://medium.com/@overtorment/bitcoin-replace-by-fee-guide-e10032f9a93f. Oct 25th, 2017. Accessed on: Mar 31st, 2018.

[24] “Mempool Size.” Blockchain.info. [Online] Available: https://blockchain.info/charts/mempool-size?timespan=1week. Accessed on: April 10th, 2018.

[25] “Mempool Count.” Blockchain.info. [Online] Available: https://blockchain.info/charts/mempool -count. Accessed on: April 10th, 2018.

[26] Brownlee Jason, “Crash course on multi-layer perceptron neural network.” MachineLearningMastery.com. [Online] Available:

https://machinelearningmastery.com/neural-networks-crash-course/. May 16th, 2016. Accessed April 17th, 2018.

[27] Alnaymat, Ghazi, et al. *Classification of VoIP and non-VoIP traffic using machine learning approaches*. ResearchGate.net. [Online] Available: https://www.researchgate.net/figure/MultiLayer-Perceptron-MLP-sturcture-334-MultiLayer-Perceptron-Classifier-MultiLayer_fig2_309592737. Oct 31st, 2016. Accessed on: April 21st, 2018.

[28] "Neural Network." Big-Data.tips. [Online] Available: http://www.big-data.tips/neural-network. Jun 22nd, 2016. Accessed on: April 21st, 2018.

11. Appendices

Appendix A

Read Mempool Python Script: readMempool.py
Assumes script is running on a Bitcoin Node and database is setup as in section 5.2.

```
import requests
import MySQLdb
import os
import ast
import time

# Global variables
db = None
cur = None

# Constants
base_url = 'https://www.bitgo.com/api/v2/btc/'
fee_url = 'https://www.bitgo.com/api/v1/tx/fee?numBlocks=2'

# Connects to db
def establish_connection():
  if (db == None):
     connection = MySQLdb.connect(host="************************************",
                          user="**********",
                          passwd="*********",
                          db="**********")
     cursor = connection.cursor(MySQLdb.cursors.DictCursor)
     return connection, cursor
  else:
     return db, cur

# Uses BitGo's public API to get chain height
def get_height():
  block_url = base_url + 'public/block/latest'
  latest = requests.get(block_url).json()
  return latest.get('height')

# Prepares a tx summary for insertion into db
def format_insert_query(summary):
  return '(txid, feerate, currentheight) VALUES ("' + \
        summary.get('txid') + '", ' + \
        str(-1) + ', ' + \
        str(summary.get('currentheight')) + ')'
```

```
# Reads mempool and returns an array of the txids
def read_mempool():
  os.system('../bitcoin-0.15.1/bin/bitcoin-cli getrawmempool > rawMempool.txt')
  mempool = open('rawMempool.txt', 'r').read()
  return ast.literal_eval(mempool)

# Check if a tx is already in db and if it isn't add it
def write_to_db_txs(tx_array, current_height):
  cur.execute('SELECT txid FROM mempool_history WHERE txid IN ' + str(tx_array).replace('[',
'(').replace(']', ')'))
  db.commit()
  db_result = cur.fetchall()
  dbtxs = set(map(lambda entry: entry['txid'], db_result))

  for txid in tx_array:
    if txid in dbtxs:
      continue
    summary = { 'txid': txid }
    summary['currentheight'] = current_height
    values = format_insert_query(summary)
    cur.execute('INSERT INTO mempool_history ' + values)
  print 'Processed: ' + str(len(tx_array)) + ' pending txs'
  db.commit()

# Adds fee estimates at block height to database
def write_fees_to_db(fees, chain_height):
  values = 'VALUES (\"' + str(fees) + '\",' + str(chain_height) + ')'
  cur.execute('INSERT INTO fee_estimates (fee_data, block_height)' + values )
  db.commit()

# Start reading mempool
while (True):
  db, cur = establish_connection()
  mempool = read_mempool()
  current_height = get_height()
  write_to_db_txs(mempool, current_height)
  time.sleep(150)
```

Appendix B

Retrieve Transaction Data from BitGo API: getTxData.js
Assumes access token is available as described in section 5.4.

```
const BitGoJS = require('bitgo');
const Promise = require('bluebird');
```

```
const colors = require('colors');
const _ = require('lodash');
const mysql = require('promise-mysql');
const co = Promise.coroutine;

co(function *() {
 // Use bitgo SDK to get tx data
 let bitgo = new BitGoJS.BitGo({env: 'prod', accessToken: process.env.TOKEN_AZ});
 let db = yield mysql.createConnection({
   host: '****************************************', user: **********, password: '***********',
   database: '******'
 });

 const txs = yield db.query('SELECT * FROM mempool_history WHERE feerate IN (-2, -1)');
 for (const tx of txs) {
   const url = bitgo.coin('btc').url('/admin/tx/' + tx.txid);
   let bitgoTx;
   try {
     yield Promise.delay(10);
     bitgoTx = yield bitgo.get(url).result();
   } catch (e) {
     console.log('Failed to get tx'.red);
     yield db.query(`UPDATE mempool_history SET feerate = -2 WHERE txid = '${tx.txid}'`);
     continue;
   }
   const feerate = parseInt(bitgoTx.fee / bitgoTx.size);
   const height = bitgoTx.blockHeight ? ', confirmHeight = ' + bitgoTx.blockHeight : '';
   const updated = yield db.query(`UPDATE mempool_history SET feerate = ${feerate}, size = ${bitgoTx.size}`
+ height + ` WHERE txid = '${tx.txid}'`);
 }
 db.end();
})();
```

Appendix C

Build Training Set for machine learning algorithm: processData.py

```
from __future__ import division
import MySQLdb
import sys

# Global variables
db = None
cur = None

BLOCK_SIZE = 1000000
SLICE_SIZE = 100000

# Connects to db
def establish_connection():
  if (db == None):
     connection = MySQLdb.connect(host="**************************",
                          user="********",
                          passwd="************",
                          db="*******")
     cursor = connection.cursor(MySQLdb.cursors.DictCursor)
     return connection, cursor
  else:
     return db, cur

def get_all_txs_entered_at_height(block_height):
  query = '''SELECT
          txid, feerate, size, currentheight, confirmheight
          FROM
          mempool_history
          WHERE
          feerate NOT IN(-2, -1)
          AND size IS NOT NULL
          AND confirmheight IS NOT NULL
          AND confirmheight > 0
          AND currentheight = ''' + str(block_height)
  cur.execute(query)
  db.commit()
  return list(cur.fetchall())

def get_all_confirmed_txs(prev_height, block_height):
  query = '''SELECT
          txid
          FROM
          mempool_history
```

```
            WHERE
            confirmheight <= '" + str(block_height) + \
            ' AND confirmheight > ' + str(prev_height)
    cur.execute(query)
    db.commit()
    return list(cur.fetchall())

def get_fee_estimates(block_height):
    query = '" SELECT
             *
             FROM
             fee_estimates
             WHERE
             block_height = '" + str(block_height)
    cur.execute(query)
    db.commit()
    return list(cur.fetchall())

# Gets lowest transaction fee in block
def get_lowest_fee_rate_in_block(block_height):
    query = '"SELECT
            feerate
            FROM
            mempool_history
            WHERE
            confirmheight = '" + str(block_height) + ' ORDER BY feerate ASC'
    cur.execute(query)
    db.commit()
    lowest_fee_list = list(cur.fetchall())
    if lowest_fee_list:
        return str(lowest_fee_list[0].get('feerate'))
    else:
        return 'No Data'

# Based on fee, find the expected number of blocks needed for confirmation
def get_estimated_confirmation_time(fee, fee_estimates):
    min_diff = sys.maxint
    block_estimate = sys.maxint
    for key in fee_estimates:
        if (abs(fee - fee_estimates[key]) < min_diff):
            block_estimate = key
            min_diff = abs(fee - fee_estimates[key])
    return int(block_estimate)

# Run a virtual mempool based on historical snapshots
mempool = {}
training_set = {
    '1-4': [],
```

```
    '5-8': [],
    '9-12': [],
    '13+': []
    }

result_csv = 'Block Height|Txs in Mempool|Confirmed Txs|vSize of Mempool\n'
# Build mock x number of next blocks by sorting by fee
def generate_training_set(result_csv, prev_height, height):
    txs_snapshot = get_all_txs_entered_at_height(height)
    for new_tx in txs_snapshot:
        mempool[new_tx.get('txid')] = new_tx

    confirmed_txs = get_all_confirmed_txs(prev_height, height)
    for confirmed_tx in confirmed_txs:
        mempool.pop(confirmed_tx.get('txid'), None)

    virtual_mempool = [value for key, value in mempool.items()]
    tx_list = sorted(virtual_mempool, key=lambda tx: int(tx.get('feerate')), reverse=True)

    total_size = 0
    for tx in tx_list:
        total_size += tx.get('size')
        block = total_size / BLOCK_SIZE + 1
        slice = (total_size / SLICE_SIZE) % 10 + 1

        feerate = tx.get('feerate')
        hit_target = 1 if tx.get('confirmheight') <= block + height else 0

        instance = [feerate, block, slice, hit_target]
        confirm_time = tx.get('confirmheight') - tx.get('currentheight')
        if confirm_time <= 4:
            training_set['1-4'].append(instance)
        elif confirm_time <= 8:
            training_set['5-8'].append(instance)
        elif confirm_time <= 12:
            training_set['9-12'].append(instance)
        else:
            training_set['13+'].append(instance)

    result_csv += str(height) + '|' + str(len(mempool)) + '|' + str(len(confirmed_txs)) + '|' + str(total_size) + '\n'
    return result_csv

# Compare Bitcoin Core fees to our technique
def compare_fees(models, prev_height, height):
    txs_snapshot = get_all_txs_entered_at_height(height)
    for new_tx in txs_snapshot:
        mempool[new_tx.get('txid')] = new_tx
```

```
  confirmed_txs = get_all_confirmed_txs(prev_height, height)
  for confirmed_tx in confirmed_txs:
    mempool.pop(confirmed_tx.get('txid'), None)

  virtual_mempool = [value for key, value in mempool.items()]
  tx_list = sorted(virtual_mempool, key=lambda tx: int(tx.get('feerate')), reverse=True)

  fee_estimates = get_fee_estimates(height)
  core_fee_estimates = {block_estimate['block_target']: (block_estimate['fee_rate_kb'])*100000 for
block_estimate in fee_estimates}

  our_fee_estimates = {}

  total_size = 0
  cur_block = 1
  prev_block = 1
  index = 0
  for tx in tx_list:
    total_size += tx.get('size')
    cur_block = total_size // BLOCK_SIZE + 1
    fee = 0
    instance = [fee, prev_block, 10]
    if cur_block != prev_block:
      will_confirm = False
      count = 0
      while not will_confirm:
        fee += tx_list[index-1].get('feerate') + .1
        instance[0] = fee
        count += 1
        if prev_block <= 4:
          will_confirm = models['1-4'].predict_using_weight(instance, models['1-4'].weights)
        elif prev_block <= 8:
          will_confirm = models['5-8'].predict_using_weight(instance, models['5-8'].weights)
        elif prev_block <= 12:
          will_confirm = models['9-12'].predict_using_weight(instance, models['9-12'].weights)
        else:
          will_confirm = models['13+'].predict_using_weight(instance, models['13+'].weights)

      our_fee_estimates[prev_block] = fee
    elif index == len(tx_list)-1:
      # Add estimation for last block
      fee = 0
      will_confirm = False
      instance = [fee, cur_block, slice]
      while not will_confirm:
        fee += tx_list[index-1].get('feerate') + .1
        instance[0] = fee
        if prev_block <= 4:
```

```
            will_confirm = models['1-4'].predict_using_weight(instance, models['1-4'].weights)
          elif prev_block <= 8:
            will_confirm = models['5-8'].predict_using_weight(instance, models['5-8'].weights)
          elif prev_block <= 12:
            will_confirm = models['9-12'].predict_using_weight(instance, models['9-12'].weights)
          else:
            will_confirm = models['13+'].predict_using_weight(instance, models['13+'].weights)

        our_fee_estimates[cur_block] = fee

      prev_block = cur_block
      index += 1

  return { 'core': core_fee_estimates, 'ours': our_fee_estimates }

# Generate weights
from learn import learn

db, cur = establish_connection()

cur.execute('SELECT DISTINCT currentheight FROM mempool_history WHERE currentheight > 516690
ORDER BY currentheight ASC')
db.commit()
block_heights = cur.fetchall()
block_heights = map(lambda block: int(block.get('currentheight')), block_heights)

prev_height = block_heights[0]
for index in range(1, len(block_heights)):
  height = block_heights[index]
  print "Processing mempool at height: " + str(height)
  try:
    result_csv = generate_training_set(result_csv, prev_height, height)
  except MySQLdb.OperationalError as e:
    db = None
    db, cur = establish_connection()
    print "Lost connection and reconnected, running block " + str(height) + " again!"
    result_csv = generate_training_set(prev_height, height)
  prev_height = height

models = {
  '1-4': learn(training_set['1-4']),
  '5-8': learn(training_set['5-8']),
  '9-12': learn(training_set['9-12']),
  '13+': learn(training_set['13+'])
}

# Train model
for key, model in models.iteritems():
```

```
  model.train_weights()

# Test weights against Bitcoin Core
prev_height = block_heights[0]
fee_csv1 = 'Block Height|Target Block|Bitcoin Core Fee|Ours|Lowest Per Block\n'
fee_csv2 = 'Block Height|Target Block|Bitcoin Core Fee|Ours|Lowest Per Block\n'
for index in range(1, len(block_heights)):
  height = block_heights[index]
  fee_estimates = compare_fees(models, prev_height, height)
  lowest_fee = get_lowest_fee_rate_in_block(height)
  fee_csv1 += str(height) + '|1|None|' + str(fee_estimates['ours'].get(1)) + '|' + lowest_fee + '\n'
  fee_csv2 += str(height) + '|2|' + str(fee_estimates['core'].get('2')) + '|' + str(fee_estimates['ours'].get(2)) + '|' +
lowest_fee + '\n'
  prev_height = height

with open('fee_estimates1.csv', 'w') as file:
  file.write(fee_csv1)

with open('fee_estimates2.csv', 'w') as file:
  file.write(fee_csv2)

with open('stats.csv', 'w') as file:
  file.write(result_csv)

with open('accuracy.txt', 'w') as file:
  text = 'Training Set Size:'
  for key, value in models.iteritems():
    text += key + '\n'
    prediction = value.test_prediction()
    text += 'Correct: ' + str(prediction[0]) + '\tTotal: ' + str(prediction[1]) + '\n'
    text += '----------\n'
  file.write(text)

with open('generated_weights.txt', 'w') as file:
  weights_generated = ''
  for key, value in models.iteritems():
    weights_generated += key + ': ' + str(value.weights) + '\n'
  file.write(weights_generated)
```