University of Zurich[UZH]

[Machine Learning Approach to Polkadot's Validator Selection Algorithm]

[Master Thesis]

Submitted in partial fulfillment of the requirements for the degree of [Master of Science in Data Science]

Author

Ben Domenic James Murphy

Matriculation Number: [16-714-925]

Email: bendomenicjames.murphy@uzh.ch

Supervisors

Prof. Dr Claudio J. Tessone

Dr. Matija Piškorec

Blockchains and Distributed Ledger Technologies (BDLT)

Department of Informatics

University of Zurich

Date of Submission: [15.07.2023]

Executive Summary

Polkadot's validator selection process employs an iterative algorithm, which is dependent on the size of the staking system. As Polkadot's staking network is growing, I propose a machine learning alternative approach to the current implementation, that is more independent of scale. The algorithm, the sequential Phragmén, aims to reduce a graph of nominator-validator edges to a subset of validators, the active set, and distribute the stake backing them, as evenly as possible. The goal of this thesis is to produce superior results, consequently improving the overall security or to provide solutions of equal quality in faster time.

In order to achieve the goal, a pipeline is setup, that gathers data and transforms it such that it is suitable for machine learning models. Predictions are made, which are adjusted to fit the requirements set by Polkadot. The adjusted results are scored and ultimately compared to the solutions discovered by sequential Phragmén.

An analysis of the training data reveals, that the active set remains highly static, with only 10 validators on average changing from era to era. This lack of diversity raised concerns regarding potential attack vectors for adversaries. Furthermore, it was observed that many nominators are acting inefficiently. Many of them do not execute their right to nominate up to 16 validators, which would maximize their chance of having a validator included in the active set. Additionally, many of them include validators, which are not eligible targets. This occurs since nominators frequently ignore their duty to actively tend to their validator preferences. They set them once and do not update them. Eligible validators become inactive (intentionally or unintentionally) and consequently remain as part of the nominators preferences.

The prediction task was split up into three models: The first model predicts the next active set, the second model predicts the sum of stake each validator receives and the third predicts the individual stake distribution. The results show, that the first two models are trained well and produce satisfactory results. However, the learning curves of the third model reveal a bias, which make the predictions sub-optimal. The source of the bias is likely the substantial changes in target values introduced by a slight shift of active set.

We conclude that it is unlikely to outperform the sequential Phragmén using a supervised approach under the described conditions. Therefore, we recommend exploring an unsupervised approach for further research. Furthermore, we recommend the development of a tool for nominators, that could increase the convenience and the security of the overall staking system as a consequence.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Polkadot is a blockchain that implements the Nominated Proof-of-Stake (NPoS) consensus protocol. This section aims to provide essential information about the mechanisms and concepts used in Polkadot, offering both a high-level overview and in-depth descriptions of individual components.
In Polkadot's protocol exist two types of delegators: *validators* and *nominators*. The validators are responsible for producing new blocks on the chain, while both nominators and validators have the option to stake their tokens. Now in order to earn rewards for staking, the validators must be elected into the *active set*. The election process, described in section 1.2, is currently derived by the *Sequential Phragmén* algorithm. It is an iterative method for electing a committee with proportional justified representation (Brill et al. 2017). The election of the active set occurs once every *era*, approximately every 24 hours on Polkadot's *relay chain*.
In NPoS, validators have the option to stake their own *DOT* (native token of Polkadot) or they may be backed by nominators. Nominators may choose to back up to 16 validators, however, only the validators ending up in the active set accumulate the stake i.e. the winner(s) takes all.
Which validator to back is not a trivial decision for the nominator. There are many factors, which could influence the outcome. The most common motivation is to maximize their gains by choosing validators with low commission and high probability to be elected. These types of validators are however frequently *oversubscribed*. Validators currently can only pay out the top 512 nominators, hence the term oversubscribed. Another, less common, motivation might be a strict preference for decentralization of the staking network over monetary profits. Consequently, the ideal pool of validators may differ from nominator to nominator.

The motivation for this thesis arises from the fact that Polkadot's staking network is growing. As of June 2023, the limit of nominators being able to actively participate is set to 22'500, each of them

electing up to 16 validators. This leads to a graph with 360'000 edges. Consequently, many nominators, who do not have the sufficient funds to stake, will not be included in the *snapshot*, ultimately leading to no opportunity to earn any rewards. The reason for this limit is due to the time complexity of the current implementation, the sequential Phragmén algorithm. $\mathcal{O}(|E| \cdot k)$ E = nr of edges, k = nr of candidates elected (Cevallos and Stewart 2020). As the graph size increases, the algorithm's computational time also increases. Therefore, the goal was to leverage the capabilities provided by machine learning models to improve the solution discovery process and furthermore, allow Polkadot to extend its limits indefinitely to include more nominators with less funds, ultimately increasing overall decentralization of the *validator pools*.

In autumn 2022, Florian Rüeggsegger and I have completed the Master's project for the Blockchain and Distributed Ledger Technologies Group (BDLT) at the University of Zurich. In said project, we have set up a data pipeline for the Blockchain Observatory (BCO) for Polkadot's relay chain. Consequently, there is access to Polkadots historical data, which will be used to fulfill the tasks of the master's thesis.

1.2 Polkadot

Polkadot is a *substrate* based blockchain, that validated its first block, known as the *genesis block*, on the 26th of May 2020. It runs on a NPoS consensus mechanism, which, similarly to the Proof of Stake (PoS) mechanism, relies on the participants of the chain to stake their currency to validate blocks. The NPoS extends said mechanism with the nominator role. They may stake their currency and select up to 16 validators, they consider trustworthy. Trustworthy, in the context of Polkadot, means that validators ensure not to engage in any offenses during their election window. Offenses include activities such as going offline, attacking the network or running modified software (*Polkadot Support Slashing* n.d.).
In order to actively participate as a nominator, certain conditions need to be met. Active participation entails being included in the snapshot, which will be used to calculate the next set of active validators. If they fail to do so, neither the nominator nor the validator can earn rewards.
Firstly, they must stake a (dynamic) minimum of 370 DOT(= $2035) (as of June 2023) (*Polkadot Wiki (learn staking)* n.d.). However this amount is subject to change due to new nominators entering or leaving the staking network (hence dynamic).
Secondly, staking the dynamic minimum might be insufficient. The nominators must ensure that they are in the correct position within the so-called *bags list*. This is particularly important for nominators, which are close to the dynamic minimum staking amount. The bags list is a concept of how Polkadot efficiently places the nominators into semi-sorted lists. The bags represent ranges of stakes in which the different nominators can be placed in. It is considered semi-sorted, since the order within the bags depends on the point in time a nominator gets inserted (see figure 1.1).

Moving up or down the list can be achieved by a couple of ways:

- Bonding/Unbonding DOT triggers an automatic *rebag* extrinsic.
- Increasing/decreasing stake via rewards/slash requires a manual call of the rebag extrinsic.
- Manual call of the *putInFrontOf* extrinsic of the module *voterList* triggers a sorting of the bag the nominator is in.

Lastly, even if a nominator fulfills the requirements above and the validator, they have elected, made it in to the active set, they may still not receive rewards. This is due to the fact, that the nominator elected a validator, which is oversubscribed. To repeat, a validator is considered oversubscribed, when they have more than 512 nominators backing it. The nominator not receiving rewards is staking less than the other 512. It is advised to not select validators, which are considered oversubscribed, to bond more or to join *nomination pools* (*Polkadot Wiki (learn staking)* n.d.). Nomination pools[1] is a relatively new concept, which allows nominators with little funds to team up and act as one nominator.

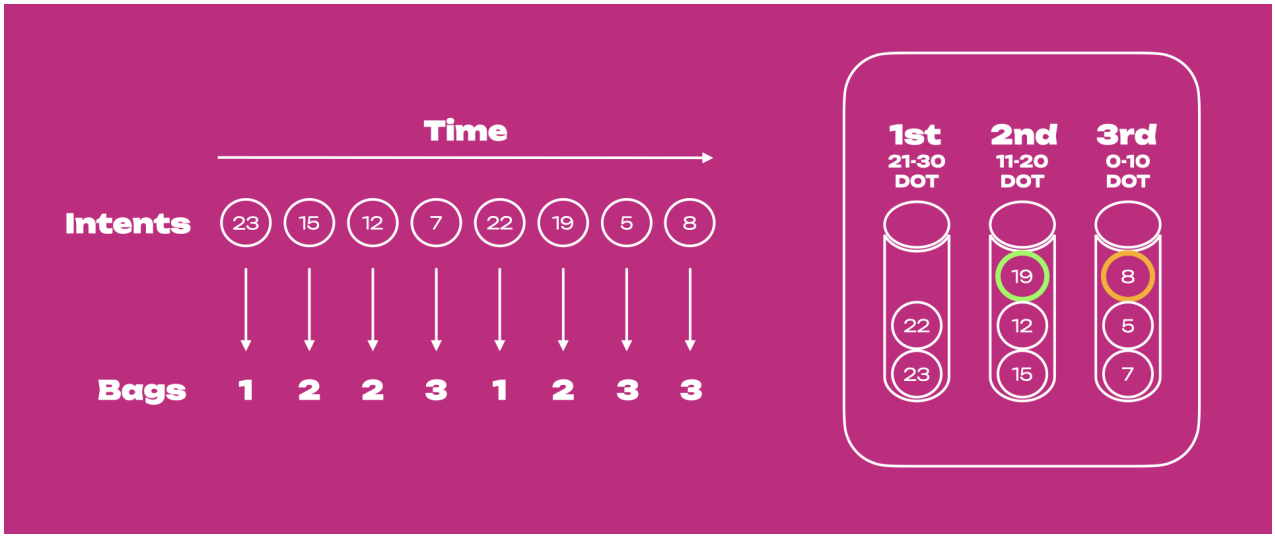


Figure 1.1 – Nominator insertion into bags over time

Source: https://wiki.polkadot.network/docs/learn-staking-advanced#bags-list

The election process starts at the beginning of the signed phase (see figure 1.2). The nominators are selected by going through the bags until the limit of 22'500 is reached (*Polkadot Wiki (learn staking)* n.d.). This graph of nominators with edges to the validators is called *snapshot*. It can be retrieved from the blockchain via the storage query module: `electionProviderMultiPhase`, function: `snapshot`.

[1] https://wiki.polkadot.network/docs/learn-nomination-pools

During the signed phase *staking miners* have the chance to compete for a reward, which is currently set to 1 DOT (as of June 2023), by calculating and submitting their solutions. Staking miner is an experimental role that was introduced in order to reduce the computational load on the nodes (*Polkadot Wiki (learn Staking advanced* n.d.). After calculating a solution the staking miner can submit it in a *compact* format to the chain via extrinsic `submit` of the module: `electionProviderMultiPhase`. The validators submit their solution, claimed score and place a deposit. The deposit is composed of a fixed amount (40 DOT as of June 2023) + a variable fee (0.0000097656 DOT/KB as of June 2023). The signed phase lasts approximately one hour, after which no more solutions can be posted by staking miners. The solutions are then ordered by their claimed score. Once a correct one is accepted, the others will be discarded and the winning staking miner will be rewarded. Only the correct solutions that have been submitted will get their deposit refunded. If there exist no good solutions, the process moves to the unsigned phase, during which the *off-chain workers*, which are the validators nodes, try to provide solutions. If still no good solutions have been posted, the fallback function, the on-chain Phragmén, would be executed. This is considered an emergency solution and should not be invoked since it slows down the average block mining time of 6 seconds. To clarify: A good solution in this context means correct. A correct solution constitutes that all the nominator-validator edges must be present in the snapshot. At the end of the unsigned phase, the function `elect` is invoked.

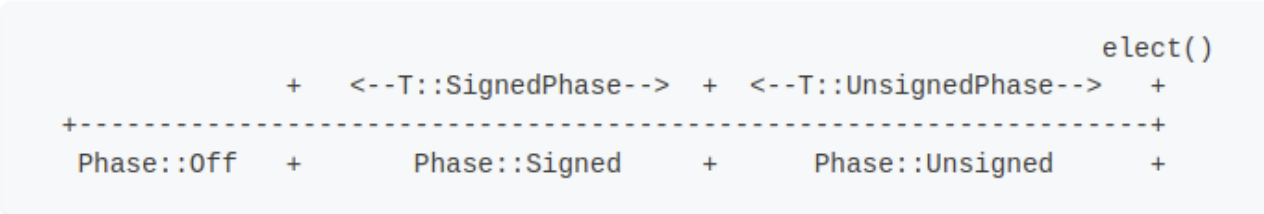


Figure 1.2 – Era on Polkadot: Different Phases

Source: https://wiki.polkadot.network/docs/learn-staking-advanced

1.3 Baseline Case

In order to justify the use for an algorithm such as sequential Phagmén, we can compare it to a baseline case. The baseline we propose is a proportional distribution of the stakes to the respective validators. First however we need to define the score metrics, that Polkadot uses to differentiate the quality of the solutions. The following metrics are calculated. The order and direction are crucial:

1. Maximise: Minimum Sum of Stake of Validator
2. Maximise: Sum of Stake Active Set
3. Minimise: Variance of Stake Active Set

In figure 1.3 we have 5 nominators, their bonds in brackets and 4 validators. The arrows indicate the nominators voting preferences. The goal is to select 3 validators for the next active set. Nominator 2 (N2) has staked 20 DOT and declared his intent to back validator 2 (V2) and validator 3 (V3). Now by applying the proportional distribution rule set, we arrive at a solution that has the score:

1. Minimum Sum of Stake of Validator = 17.5
2. Sum of Stake Active Set = 75
3. Variance of Stake Active Set = 81.25

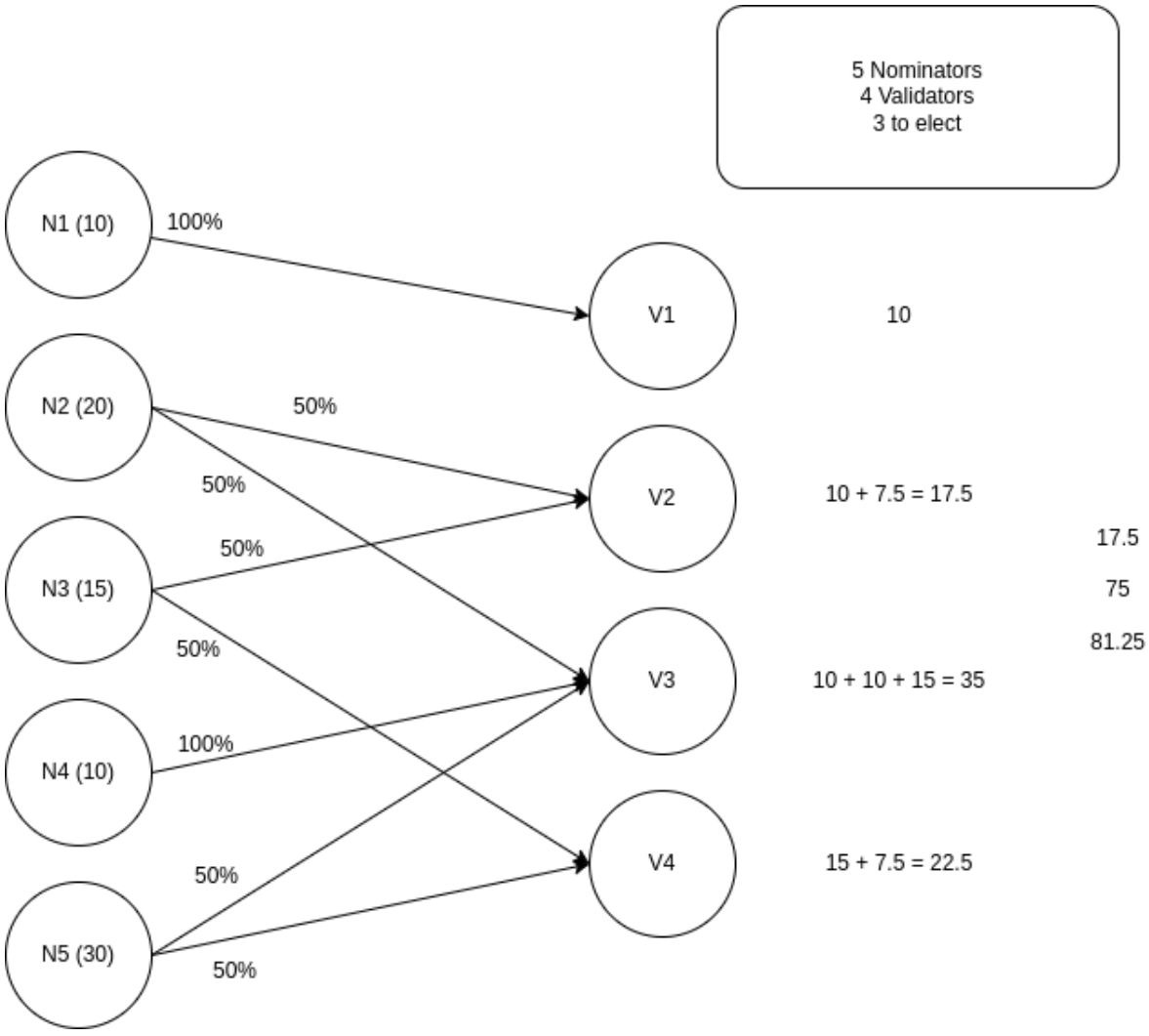


Figure 1.3 – Example by Paimani n.d. Proportional Distribution

Comparatively, applying the sequential Phragmén algorithm to the same snapshot we arrive at the outcome described in figure 1.4.
Applying the score metrics to this new distribution, we arrive at a solution with score:

1. Minimum Sum of Stake of Validator = 19
2. Sum of Stake Active Set = 75
3. Variance of Stake Active Set = 38.5

Looking back at the table (1.3), we have increased the minimum sum of stake of validator from 17.5 to 19 and decreased the variance from 81.25 to 38.5, which is both desirable. The sum of stake remains the same, as we have selected the same active set.

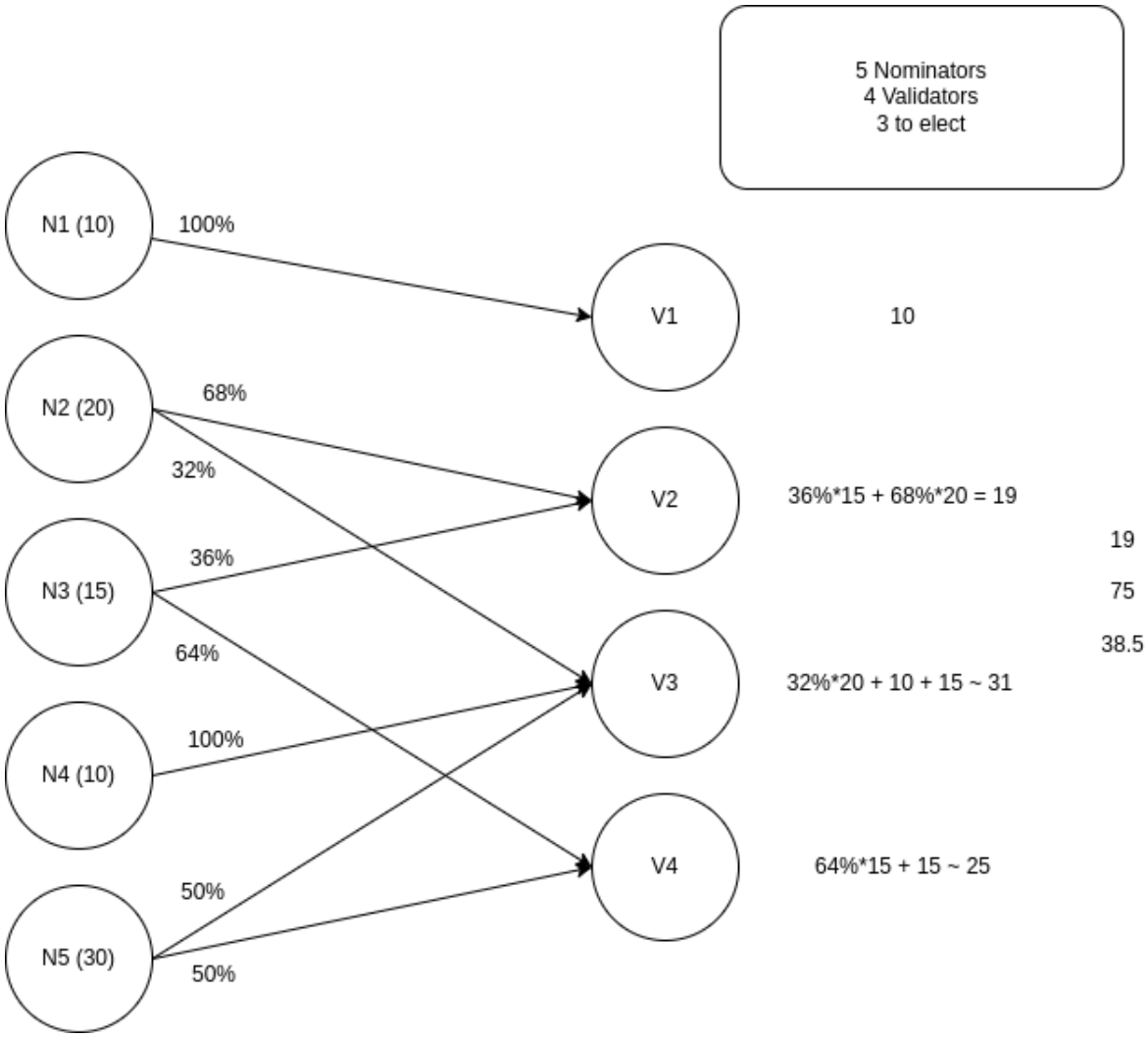


Figure 1.4 – Example by Paimani n.d. Sequential Phragmen Result

1.4 Sequential Phragmén

The sequential Phragmén approach was first introduced by Phragmén and Thiele in 1894–1895 (Janson 2018). It attempts to balance the stake as evenly as possible among the active set since in NPoS all the validators receive (approximately) an equal reward. The score metrics (1.3) seek to achieve this goal.
The explanation of how the algorithm works is divided into two parts: The basic Phragmén and the weighted Phragmén. In the Basic Phragmén all stakes are set to unity value, meaning the individual votes are unweighted. In the weighted Phragmén weights are introduced to demonstrate how this affects the election outcome. Finally, we show how the results may be optimized, which corresponds to the balancing iterations in the Rust implementation.
Both the basic and weighted following examples can be found on the Polkadot Wiki[2].

1.4.1 Basic Phragmén

The basic Phragmén implementation, as outlined in Brill et al. 2017, describes the following scenario: A group of voters (nominators), candidates (validators), their voting preferences and a limited amount of places in the active set. There are a couple of rules:

- voters must vote for at least one candidate.
- voters can not vote for all candidates. They may state their preference for max: *n-1 candidates*.
- voters vote weight is equal. All of them are set to 1.

Algorithm

1. Voters state their voting intentions, meaning they declare which candidates they vote for. These may not be changed once the snapshot is taken.
2. A so-called *load* is initialized for every voter and set to 0.
3. The candidate with the *least average (mean) cost* of submitted voting intentions wins the next seat. This is including the load assuming they win.
4. The voters, that had the winning candidate in their preference set, get *1/n* added to their load.
5. The loads of the voters, that supported this rounds winning candidate, are set to the average of the losing voters loads.
6. If all seats are taken the process is complete, else repeat from step 3.

[2] https://wiki.polkadot.network/docs/learn-phragmen

```
Open Seats: 3

Candidates:   A B C D  L0
-------------------------
Voter V1:       X      0
Voter V2:         X X  0
Voter V3:       X   X  0
Voter V4:     X X      0
Voter V5:       X X X  0
```

(a) Example Basic Phragmén round 0

```
Filled seats: 1 (B)
Open Seats: 2

Candidates:   A B C D  L0 L1
----------------------------
Voter V1:       X      0  1/4
Voter V2:         X X  0  0
Voter V3:       X   X  0  1/4
Voter V4:     X X      0  1/4
Voter V5:       X X X  0  1/4
```

(b) Example Basic Phragmén round 1

Figure 1.5 – Basic Phragmén Round 0 and 1

Source: https://wiki.polkadot.network/docs/learn-phragmen

Example

In round 0 (1.5a), voter V1 has stated his preference for candidate B, voter V2 for C and D, and so on. L0 is the load at round 0, which is, as the algorithm step 2 states, set to 0.

In round 1 (1.5b), the candidate with the most votes will win, since the loads are equal. In this scenario it is candidate B. The loads of all the voters, that have B in their preferences, will be set to 1/n, which is 1/4. Now there an averaging step would take place, however it is skipped in round 1, since it does not affect the results.

Moving on to the second round (1.6a). The calculations for the three remaining candidates, A, C and D, are as such:

- A: $(1/4 + 1/1) \ / \ 1 = 5/4$
- C: $((0 + 1/2) + (1/4 + 1/2)) \ / \ 2 = 5/8$
- D: $((0 + 1/3) + (1/4 + 1/3) + (1/4 + 1/3)) \ / \ 3 = 1/2$

Thus, D wins the next seat. The loads of the winning voters are now set to the average of the losing voters loads $((1/4 + 1/4) \ / \ 2 = 1/2$. The losing voters loads are carried over to the next round.

In the last round (1.6b), candidate A and C remain. The calculations are as following:

- A: $(1/4 + 1/1) \ / \ 1 = 5/4$
- C: $((1/2 + 1/2) + (1/2 + 1/2)) \ / \ 2 = 1$

```
Filled seats: 2 (B, D)
Open Seats: 1

Candidates:   A B C D  L0 L1  L2
-----------------------------------
Voter V1:       X      0  1/4 1/4
Voter V2:         X X  0  0   1/2
Voter V3:       X   X  0  1/4 1/2
Voter V4:     X X      0  1/4 1/4
Voter V5:       X X X  0  1/4 1/2
```

(a) Example Basic Phragmén round 2

```
Filled seats: 3 (B, D, C)
Open Seats: 0

Candidates:   A B C D  L0 L1  L2  L3
--------------------------------------
Voter V1:       X      0  1/4 1/4 1/4
Voter V2:         X X  0  0   1/2 1
Voter V3:       X   X  0  1/4 1/2 1/2
Voter V4:     X X      0  1/4 1/4 1/4
Voter V5:       X X X  0  1/4 1/2 1
```

(b) Example Basic Phragmén round 3

Figure 1.6 – Basic Phragmén Round 2 and 3

Source: https://wiki.polkadot.network/docs/learn-phragmen

Again, the lowest average wins, making C the winner of the last seat. An interesting property of this example: Summing up the loads of each round, leads to the number of seats that have been filled up until that point. So round 0 sums up to 0, round 2 sums up to 2.

1.4.2 Weighted Phragmén

By incorporating weights to the voters preferences, we can accurately describe how Polkadot elects the active set for each era. The weights reflect the amount of DOT, the nominators have bonded. It may seem anti-democratic to give voters with higher amount of DOT more voting power. It is unfortunately unavoidable due the fact, that a high-power voter can simply create more accounts, thus achieving the same influence (*Polkadot Wiki (learn Phragmen)* n.d.).

Algorithm

Unlike the basic Phragmén, we have to keep track of both the candidates (validators) and voters (nominators) scores.

1. Same as in basic Phragmén, get the snapshot.
2. Use it to create a weighted graph mapping the voters to the candidates, the edges being the total *potential* they may receive. The sum of all potentials is referred to as the *approval stake*.
3. For all not elected candidates a score is calculated. The formula is: *1 / approval stake*.
4. For every voter, the score of their candidates must be updated. The formula is: *candidate_score + (voter_budget * voter_load / candidate_approval_stake)*
5. The candidate with the lowest score is elected and removed from the pool.

6. Update the loads which are connected to the winning candidate. The weight is set to *the score of the candidate minus the voters load*. The voters load are set to the candidates score.
7. If all seats are filled continue, else return to step 3.
8. Now the stake distribution is calculated. The formula for each voter is as such: *voter_budget * edge_load / voter_load*

Example

```
Filled seats: 0
Open Seats: 3

Candidates:    A B C D E  L0
-----------------------------
Voter V1 (1):  X X        0
Voter V2 (2):  X X        0
Voter V3 (3):  X          0
Voter V4 (4):    X X X    0
Voter V5 (5):  X     X    0
```

(a) Example Weighted Phragmén round 0

```
Candidates:    A B C D E  L0 L1
-----------------------------------
Voter V1 (1):  X X        0  0.091
Voter V2 (2):  X X        0  0.091
Voter V3 (3):  X          0  0.091
Voter V4 (4):    X X X    0  0
Voter V5 (5):  X     X    0  0.091
```

(b) Example Weighted Phragmén round 1

Figure 1.7 – Weighted Phragmén Round 0 and 1

Source: https://wiki.polkadot.network/docs/learn-phragmen

As in round 0 (1.7b), we have voters like V1 stating their preference for candidate A and B, initial loads and three open seats. What differs is the weight of the votes, defined in brackets next to the voters. So voter V1 has a weight of 1, V2 weight of 2 and so on.

Now we calculate the approval stake for all candidates and the resulting score (The values are rounded to three decimals):

- A: $1 / (1 + 2 + 3 + 5) = 0.091$
- B: $1 / (1 + 2 + 4) = 0.143$
- C: $1 / 4 = 0.25$
- D: $1 / (4 + 5) = 0.111$
- E: $1 / 0 = \text{N/A}$

According to step 4, the scores must be updated, however any new addition will be 0 since the initial loads are 0. We select the candidate with the lowest score, A, and set the loads of the voters, that have the winner in their preference, to said score. Furthermore, E can be ignored from here on, since their approval stake is 0.
All the voters who had a winning candidate will get a higher load. This is by design to reduce their future influence. The edges are set to the score minus current voter load, which will be used to derive the stake distribution in the final step. To reiterate:

- *edge_load* = *elected_candidate_score* - *voter_load*
- *voter_load* = *elected_candidate_score*

In our example this means that all the winning voters will have an edge_load of 0.091 - 0 = 0.091 and a new voter_load of 0.091 (fig: 1.7b).
Before electing the new candidate, the scores have to be updated as described in step 4.

- V1 updates B to 0.156 (= 0.143 + ((1 * 0.091) / 7)
- V2 updates B to 0.182 (= 0.156 + ((2 * 0.091) / 7)
- V4 updates B to 0.182 (= 0.182 + ((4 * 0) / 7)
- V4 updates C to 0.25 (= 0.25 + ((4 * 0) / 4)
- V4 updates D to 0.111 (= 0.111 + ((4 * 0) / 9)
- V5 updates D to 0.162 (= 0.111 + ((5 * 0.091) / 9)

Thus the final candidates scores are the following:

- Candidate B: 0.182
- Candidate C: 0.25
- Candidate D: 0.162

Clearly, candidate D has the lowest score and is elected. The reasoning being two-fold: for one the candidate D had a high potential stake backing him and voter 4 did not get penalized in the previous round, therefore not adding to the candidate score in the updating part. Finally, the loads have to be updated of the voters who selected the winning candidate.

```
Filled seats: 2 (A, D)
Open Seats: 1

Candidates:    A B C D E  L0 L1    L2
-----------------------------------
Voter V1 (1):  X X        0  0.091 0.091
Voter V2 (2):  X X        0  0.091 0.091
Voter V3 (3):  X          0  0.091 0.091
Voter V4 (4):    X X X    0  0     0.162
Voter V5 (5):  X     X    0  0.091 0.162
```

(a) Example Weighted Phragmén round 2

```
Filled seats: 3 (A, D, B)
Open Seats: 0

Candidates:    A B C D E  L0 L1    L2    L3
-----------------------------------------
Voter V1 (1):  X X        0  0.091 0.091 0.274
Voter V2 (2):  X X        0  0.091 0.091 0.274
Voter V3 (3):  X          0  0.091 0.091 0.091
Voter V4 (4):    X X X    0  0     0.162 0.274
Voter V5 (5):  X     X    0  0.091 0.162 0.162
```

(b) Example Weighted Phragmén round 3

Figure 1.8 – Weighted Phragmén Round 2 and 3

Source: https://wiki.polkadot.network/docs/learn-phragmen

For the last round, we again start with the initial scores, and update them accordingly:

- V1 updates B to 0.156
- V2 updates B to 0.182
- V4 updates B to 0.274
- V4 updates C to 0.412

Having the lowest score, the candidate B wins the last round. The loads get adjusted one last time before moving on to step 8. The resulting weighted graph looks as such:

- Nominator: V1
  - Edge to A load= 0.091
  - Edge to B load= 0.183
- Nominator: V2
  - Edge to A load= 0.091
  - Edge to B load= 0.183
- Nominator: V3
  - Edge to A load= 0.091
- Nominator: V4
  - Edge to B load= 0.113
  - Edge to D load= 0.162

- Nominator: V5
  - Edge to A load= 0.091
  - Edge to D load= 0.071

Applying the formula in step 8 allows us to calculate the individual voter stake distribution. For example Voter V1 would be:

- 1 * 0.091 / 0.274 = 0.332
- 1 * 0.183 / 0.274 = 0.668

Applying this to the rest gives us:

- V1 supports: A with stake: 0.332 and B with stake: 0.668.
- V2 supports: A with stake: 0.663 and B with stake: 1.337.
- V3 supports: A with stake: 3.0.
- V4 supports: B with stake: 1.642 and D with stake: 2.358.
- V5 supports: A with stake: 2.813 and D with stake: 2.187.

resulting in final backing stakes of:

- A is elected with stake 6.807.
- D is elected with stake 4.545.
- B is elected with stake 3.647.

1.4.3 Optimizations

The rationale for further optimization is the following (*Polkadot Wiki (learn Phragmen)* n.d.):

- Reduction of nominator-validator edges. This optimizes the size of the solution posted and minimizes future reward transactions, consequently reducing the load on the network as a whole. In the best case scenario, every nominator would end up backing one validator. Network slowdown has occurred due to the large number of payout transactions and can be seen as an attack vector for adversaries (*Polkadot Wiki (learn Phragmen)* n.d.).
- Improving the distribution of stake. This works towards improving the overall score of the solution. An example demonstrating that an increase in score leads to an increase in security is the following: An active set composed of 5 validators with the stakes: {1000, 20, 10, 10, 10} can be attacked by a malicious validator with 11 tokens. Furthermore, they can achieve the majority with 33 tokens. By applying the even distribution, we compute an active set with stakes of {210, 210, 210, 210, 210}. The attacker now requires 633 tokens to claim majority (600 more than in the previous case). Achieving perfect balance is unlikely, however approaching it increases the entry barrier for any potential attacker of the network (*Polkadot Wiki (learn Phragmen)* n.d.).

Chapter 2

Methodology

This chapter describes the methodology applied to the prediction task of this thesis. The process was split up into three separate models to avoid a complex multi-target regression/classification task.
The first step was to acquire the data via storage queries and calculate the target variable via sequential Phragmén. This data was then transformed to fit the first prediction task.
The first model is a classifier, Active Set Predictor (ASP), designed to predict the next active set. Model 2 is a regression model, Expected Sum Stake Predictor (ESSP), predicting the expected sum of stake of a specific validator. Model 3 is a regression model, Stake Distribution Predictor (SDP), predicting the stake distribution of a nominator. By splitting the prediction task into three steps the overall complexity was reduced and the individual models could focus on their prediction tasks, allowing for better results.
The code of this project is published at `https://gitlab.com/moerfii/polkadot_ML_sequentialPhragmen`

2.1 Pipeline

The full Pipeline consists of multiple processing and predicting stages. After the prediction stage of SDP is complete, the results are adjusted to meet the criteria of correctness. The adjusted distributions of stake are scored via the three defined metrics 1.3 and compared to the stored solution (if one exists). Since it was often the case, that no solution was posted in the earlier eras the solutions get compared (by default) to the solution derived by the local sequential Phragmén. The predicted solution and its score are saved for further analysis. If a user of this pipeline desires to act as a staking miner, they must transform it into the compact assignment format to submit the solution to the chain.

2.1.1 Data Acquisition

The data required for the prediction task is available in the snapshot, which can be obtained via storage query `snapshot`. It is important to note, that storage queries have a default `history depth` of 84, which defines how many eras in the past, the data can be retrieved. So in order to query historical data prior to 84 eras, it is mandatory to pass the respective *block hash*. Fortunately, the `substrate-interface` written for python provides the possibility to pass it conveniently as an argument. It is worth nothing that some extra node configuration is required when querying via storage queries due to the size of the message returned. When launching the node one must add these commands for successful execution: `-ws-max-out-buffer-capacity 1000 -rpc-max-payload 1000`.
Further increasing the convenience of accessing historical data was made possible by Florian Rüegsegger. His data collection thesis provides a table in a postgreSQL database (Rüegsegger 2023). The data can be acquired via the python scripts implemented.
Lastly, if the user decides to run this script for future era predictions, they have the possibility to subscribe to storage queries. This will pull any snapshot data as soon as it is made available.

The configuration file of the script enables the user to define the era they would like to predict. The script will acquire the nine previous eras by default. For example: if one sets the era to 909, the necessary data for eras 900-909 is gathered. The `main.py` then starts its process by performing checks whether or not the requested snapshots are available. If unavailable, they will be gathered via query of the postgreSQL database. Another set of checks are making sure the sequential Phragmén was applied to the training eras.

In order to train our models on the best possible solutions the sequential Phragmén algorithm implemented in Rust is run locally for an extended period of time. The rust script gets called via sub-process library from the python script. The default is set to 400 iterations of balancing, since running it for 10'000 iterations delivered similar if not equal results. Running it for 400 iterations takes about one minute.

Lastly, two dictionaries are created and stored. They contain the index-to-address mapping. This is necessary, if the user wants to act as a staking miner to provide a valid solution. A couple of optimizations for the solution format result in the previously mentioned compact assignment format: Indices are used instead of addresses to save space. The amount one validator receives is scaled using *perU16* (Parts per 65535) format. If there are two validators for one nominator, only the first nominator has a value in perU16. It is assumed that the remaining validator receives the remainder. The purpose of the compact assignment is the reduction in required memory of the solution file.

2.1.2 Active Set Predictor Processing

In the first stage the snapshots get individually processed. Each snapshot contains two dictionaries: `snapshot['voters]` and `snapshot['targets']`. The `snapshot['voters]` contains information about the nominator, namely how much he staked (known as bond) and his voter preferences. The

`snapshot['targets']` contains the list of validators, which are eligible to validate in the current era. This second dictionary is used to check, whether a validator selected by the nominator is among the eligible targets. It is common, that a nominator has set his preferences in the past and did not make an effort to update them. In some cases, validators did not successfully state their intention to validate because the execution of the extrinsic failed. Figure 2.3a shows that roughly half of the nominators do not actively manage their voter preferences. An in-depth analysis of this issue follows in 2.1.2.

In a next step the voters dictionary gets transformed into a `pandas dataframe`. The list of preferences gets converted into individual rows in order to circumvent the *curse of dimensionality*. Excluding this step would result in over 1000 columns representing individual validators.

nominator	bond	list of validators
15j4...	3.2e17	['12Mq...', '12eZ...', '16Fw...', ...]
16ZL...	2.7e17	['1v7f...', '16FR...', '15Xv...', ...]
14Ns...	2.0e17	['1ZHN...', '1bjt...', '14d2...', ...]
16DG...	1.2e17	['12wo...', '1nTf...', '15th...', ...]
16GM...	1.0e17	['15Zv...', '14MD...', '15Co...', ...]

Table 2.1 – Snapshot raw data prior to processing

- nominator: The address of the nominator.
- bond: The amount this nominator is staking.
- list of validators: The nominators voting intentions. He may select up to 16 validators.

After processing the data looks as such:

validator	overall_total_bond	overall_proportional_bond	nominator_count	elected_current_era	elected_previous_era	era
111B8...	2.2e16	4.7e15	232	1	1	904
1123R...	5.7e15	5.8e14	314	0	0	904
1124R...	1.3e15	1.8e14	120	0	0	904
112A6...	1.2e14	1.5e13	27	0	0	904
112EM...	4.1e15	3.6e14	97	0	0	904

Table 2.2 – Active Set Predictor Dataframe

- validator: address of validator.
- overall_total_bond: Sum of all potential stake they may receive. Summed up this is more than the stake available.

- overall_proportional_bond: Sum of all proportional stake they may receive. Proportional means if a nominator selects 2 validators they receive a proportional share of 50% each. Summed up this equals to the stake available.
- nominator_count: How many nominators have selected the validator in this era.
- elected_current_era: This is the target variable for ASP. Binary column stating whether the validator made it into the active set.
- elected_previous_era: Binary column stating whether the validator has made it into the active set in the previous era.
- era: Index stating what era the data is from. Used for splitting data, not used for training purposes.

Active Set Predictor Prediction Stage

To predict the next era, a total of nine eras have to be processed and six have to be predicted. This is due to the fact, that we treat the eras as a time series prediction. Thus a *model shift* is implemented in order to prevent any *data leakage*. Data leakage can be described as information of the target variable being included in the training set. This may lead to the model delivering results, which do not accurately represent its the capabilities.
For each era we prepare the model by defining the path to the processed data, defining the target column, which is the `elected_current_era` binary column, defining the features and the test era. The dataframe gets split up into the previously defined features and target columns.
The model of choice, with the pre-configured hyperparameters gets elected. Then the data gets further divided into the training and test data set. *Feature scaling* is applied to the data, which is crucial to give the features with different magnitudes an equal weight. Certain features such as the stake would otherwise dominate due to their scale.
Once the preparations are completed, the model is fitted to the training data. Predictions are made and returned in the predict_proba format to extract the probabilities. These can be used as a feature in the subsequent models. The predictions are evaluated by appropriate scoring metrics and stored to a Comma-separated values (CSV) file.

Exploratory Data Analysis (Active Set Predictor)

The dataframe resulting from the transformation contains 1206 rows (for era 904) and 7 columns. Each row represents an eligible validator, that was nominated at least once for the current era. The resulting dataframe is approximately 0.07 MB in size, while the initial snapshot JavaScript Object Notation (JSON) file takes roughly 18MB in memory.

	count
elected_current_era	
0	909
1	297

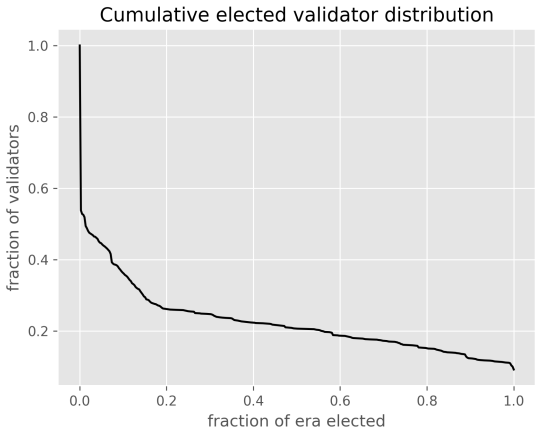
Table 2.3 – Label Count in one Era

We can clearly see that 297 winners per era are present (table 2.3), as defined in the requirements. Looking across one year of eras, it becomes evident that a big share of validators never win, while approximately 10% of the participating validators consistently win in every era (figure 2.1a). This lead to a deeper investigation into the staking dynamics of Polkadot by comparing the *absolute winners* (those who win every time) with the *absolute losers* (the validators that never win).
There is a clear divide between the two groups in terms of bond amount (figure 2.2a), emphasizing the importance of having sufficient stake behind a validator in order to increase the likelihood of getting elected. This fact is further supported by the difference in nominator count (figure 2.2b). A big share of the winners are clearly oversubscribed, which means many nominators will not be paid for their staking activity. Surprisingly, even one of the absolute losers is oversubscribed. Further investigation on subscan[1] reveals that this validator was popular prior to the eras analyzed. This validator halted his activity and the nominators backing them have not updated their preferences. This prompted the hypothesis that many nominators operate in a *set and forget* manner.
Figure 2.1b shows that the active set experiences minimal changes from era to era. This finding further enforces the notion that nominators are in fact acting passively. The correlation matrix (figure 2.5) reveals a high correlation of the target variable elected_current_era and elected_previous_era, which further enforces the observation that the active set constellation is static. This consistency can prove to be problematic for the overall staking system of Polkadot. It is problematic since it goes against the fundamental principle of blockchain which is decentralization. By having an active set that is more or less consistent it becomes by definition centralized to this subset of validators. It greatly increases the risk for collusion, thus compromising the security of the network.

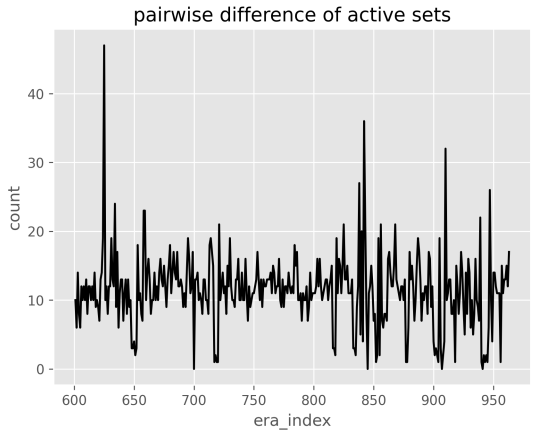
As a sidenote: The correlation matrix reveals a high correlation between the proportional and total bonds, therefore we could safely drop the total bond column, since it has a slightly lower correlation to our target variable.

The figure 2.4 depicts a concerning trend of validators being included in voter preferences, which are not eligible to validate in the given era. This highlights the fact that Polkadot's staking network has room for improvement. If this trend continues, this can have severe implications for the overall

[1] https://polkadot.subscan.io/account/1hJdgnAPSjfuHZFHzcorPnFvekSHihK9jdNPWHXgeuL7zaJ

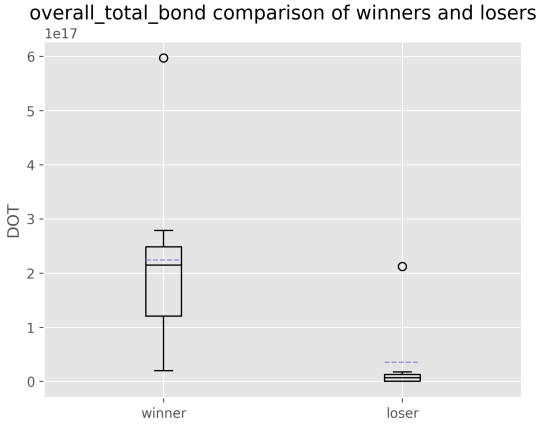


(a) Cumulative density plot of elected validators across 1 year

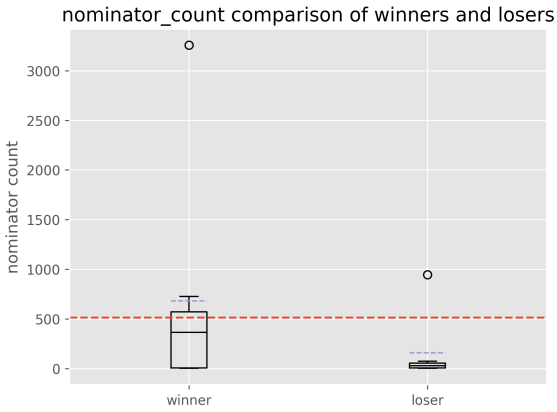


(b) Pairwise Differences of Active Sets across one year of eras

Figure 2.1 – Polkadots active set composition remains roughly equal

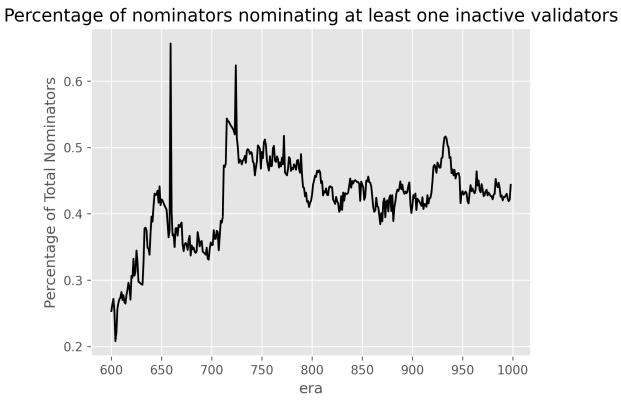


(a) Comparison of Potential Bonds

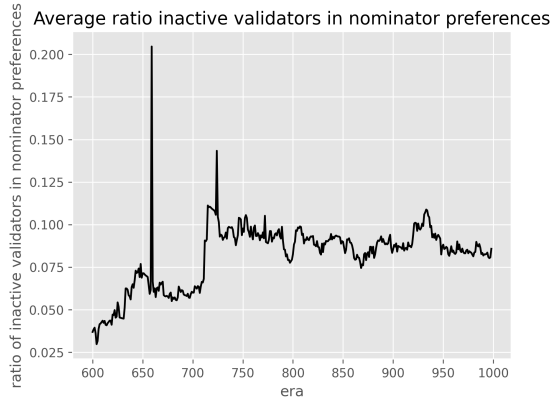


(b) Above horizontal line is oversubscribed (count > 512).

Figure 2.2 – Comparison of absolute winners (won every era) vs absolute losers (lost every era)



(a) Percentage of nominators across 400 eras



(b) Ratio depends on size of nominator preferences

Figure 2.3 – Inefficiencies of Polkadots Staking System

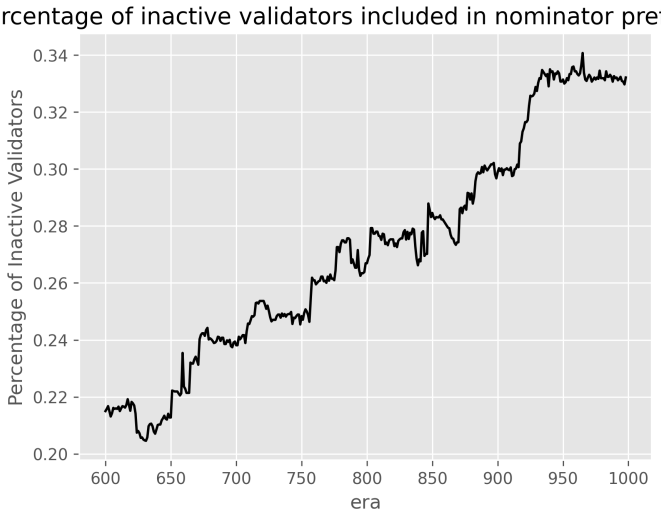


Figure 2.4 – Steady increase of share of validators that are inactive but still included in preference set

performance and security of the network. Firstly, the bond can be considered wasted if a nominator ends up having a preference set consisting only of inactive validators. Secondly, newer validators with less support will never be elected due to the lack of stake supporting them. This can lead to a system that consistently elects the same set of validators (which is currently happening as mentioned above). This consistency could lead to a potential attack vector of Polkadot. In order to address this issue it requires nominators to be informed of the necessity to update their preferences regularly. Since this process is quite labor intensive I recommend the development of a tool, which could automate this task. Said tool could allow nominators to define criteria such as commission, slashed rate and levels of activity. The program would then search the top 100 validators that best match the criteria. The user would have to sign off the transaction confirming the preference for these validators. The system would rotate the preference set of 16 validators randomly from said pool. A tool that removes the tedious labor of updating the active set could significantly improve the efficiency and performance of Polkadots staking system overall.

The spike in era 660 in figure 2.3a and figure 2.3b prompted further investigation. It was discovered, that one of the biggest validator network, Zugcapital, was *chilled* for one era[2]. Validators may be chilled if they are inactive, however they only get slashed if 10% of the validators are simultaneously inactive (*Polkadot FAQ* n.d.). As a consequence, they were made unavailable in the `snapshot['targets']` section of the next era explaining the big spike. Looking at one of the validator pages[3] it is evident, they missed out on participating in the active set of era 661. The Zugcapital network reactivated their validation intentions in the subsequent era.

[2]The block can be found here: https://polkadot.subscan.io/block/9556763

[3]https://polkadot.subscan.io/era?address=1zugcaaABVRXtyepKmwNR4g5iH2NtTNVBz1McZ81p91uAm8&page=19

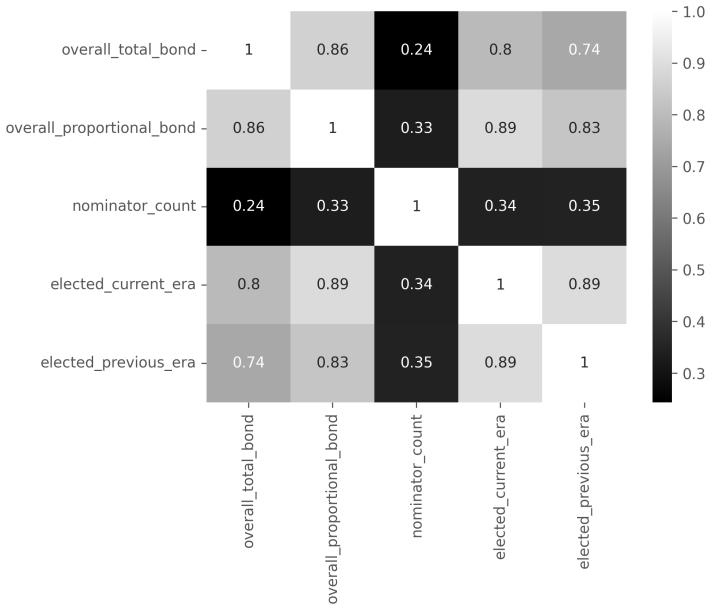


Figure 2.5 – Correlation matrix of Active Set Predictor dataframe

2.1.3 Expected Sum Stake Predictor Processing

In this stage the processing splits up into two directions. For one, the data that is required for the training dataset gets processed. The training dataframe is reduced to include the active set derived by sequential Phragmén. The testing dataframe on the other hand includes the active set predicted by ASP. The purpose of this differentiation is to evaluate the performance of the pure machine learning approach without relying on sequential Phragmén.

There is a second distinction made in this part of the pipeline. Both the dataframe for ESSP and SDP make use of the features derived in this stage, however the dataframe for ESSP requires an extra aggregation step. The proportional, total and solution bonds and validator frequency are being summed up, validator centrality and probablity of selection features, as well as the era are reduced to one value (since they are all the same for the same validator). All nominator features are dropped since they add no information to the validators. The dataframes for both models are transformed into a nominator-based row structure.

It is important to mention that due to the aggregation step the dataframe for one era is being reduced to 297 rows. Having three eras as a training set is not sufficient in this case. To circumvent this issue the ESSP is not retrained for every prediction. Instead it is pre-trained on 200 eras, which ensures it does not overfit. Since the features included are not affected by any temporal position, this predictor should produce stable results in the future.

The processed dataframe has the following structure:

nominator	validator	era	proportional_bond	total_bond	number_of_validators
15j4...	12Mq...	990	2.0e16	3.2e17	16
15j4...	12eZ...	990	2.0e16	3.2e17	16
15j4...	16Fw...	990	2.0e16	3.2e17	16
15j4...	1eLU...	990	2.0e16	3.2e17	16
15j4...	168b...	990	2.0e16	3.2e17	16

solution_bond	validator_frequency_current_era	prev_min_stake	prev_sum_stake
8.1e15	4.0	1.7e16	6.1e18
2.1e16	1.0	1.7e16	6.1e18
2.1e16	1.0	1.7e16	6.1e18
1.5e16	2.0	1.7e16	6.1e18
2.1e16	1.0	1.7e16	6.1e18

prev_variance_stake	overall_proportional_bond	overall_total_bond
7.3e29	4.2e16	4.6e17
7.3e29	2.0e16	3.2e17
7.3e29	2.0e16	3.2e17
7.3e29	2.9e16	3.7e17
7.3e29	2.0e16	3.2e17

avg_proportional_bond	avg_total_bond	nominator_index	validator_index
1.0e16	1.1e17	0	21901
2.0e16	3.2e17	0	21905
2.0e16	3.2e17	0	21914
1.4e16	1.8e17	0	21902
2.0e16	3.2e17	0	21913

nominator_centrality	validator_centrality	probability_of_selection
0.0	0.0	1.0
0.0	0.0	1.0
0.0	0.0	1.0
0.0	0.0	1.0
0.0	0.0	1.0

Table 2.4 – Expected Sum Stake Predictor Dataframe

- nominator: Address of nominator.
- validator: Address of validator.
- era: Current era.
- proportional_bond: Proportional share of nominator stake adjusted to elected validators.
- total_bond: total Potential bond validator can receive from nominator.
- number_of_validators: Nominator out-degree. How many validators did the nominator select.
- solution_bond: Target variable. Calculated by sequential Phragmén. Ratio of how much the validator should receive of nominator.
- validator_frequency_current_era: Validator in-degree. How often did a validator get elected by any nominator.
- prev_min_stake: score of previous era solution.
- prev_sum_stake: score of previous era solution.
- prev_var_stake: score of previous era solution.
- overall_proportional_bond: sum of proportional bond validator has received.
- overall_total_bond: sum of potential bond validator can received.
- avg_proportional_bond: average of proportional bond validator has received.
- avg_total_bond: average of total potential bond validator may receive.
- nominator_index: index of nominator, sorted by bond
- validator_index: index of validator, sorted by proportional bond
- nominator_centrality: degree centrality of nominator
- validator_centrality: degree centrality of validator
- probability_of_selection: prediction of ASP

Exploratory Data Analysis (Expected Sum Stake Predictor)

The analysis of the nominator out-degree (figure 2.6), also known as the nominator preferences, provide some insights into the staking characteristics and processing of the data. It reveals that only approximately a third of the nominators aim to maximize their chances by nominating the maximum of 16 validators. This could be an indicator that many nominators do not follow monetary maximization strategies or that there is a general misunderstanding of the underlying staking system in place.

As previously mentioned, the processing removes all the validators, which did not get predicted by ASP or elected by sequential Phragmén. This can be seen in the shift of the column furthest to the right (voting for 16 validators), downwards to the left. Only a small share of nominators is able to distribute their stake among 16 validators, which is by design of the optimization step (1.4.3). The removal of the losing set of validators can also be observed in the shift in figure 2.7. The dotted line shows, where validators are considered oversubscribed, where only the top 512 nominators will be receiving staking rewards.

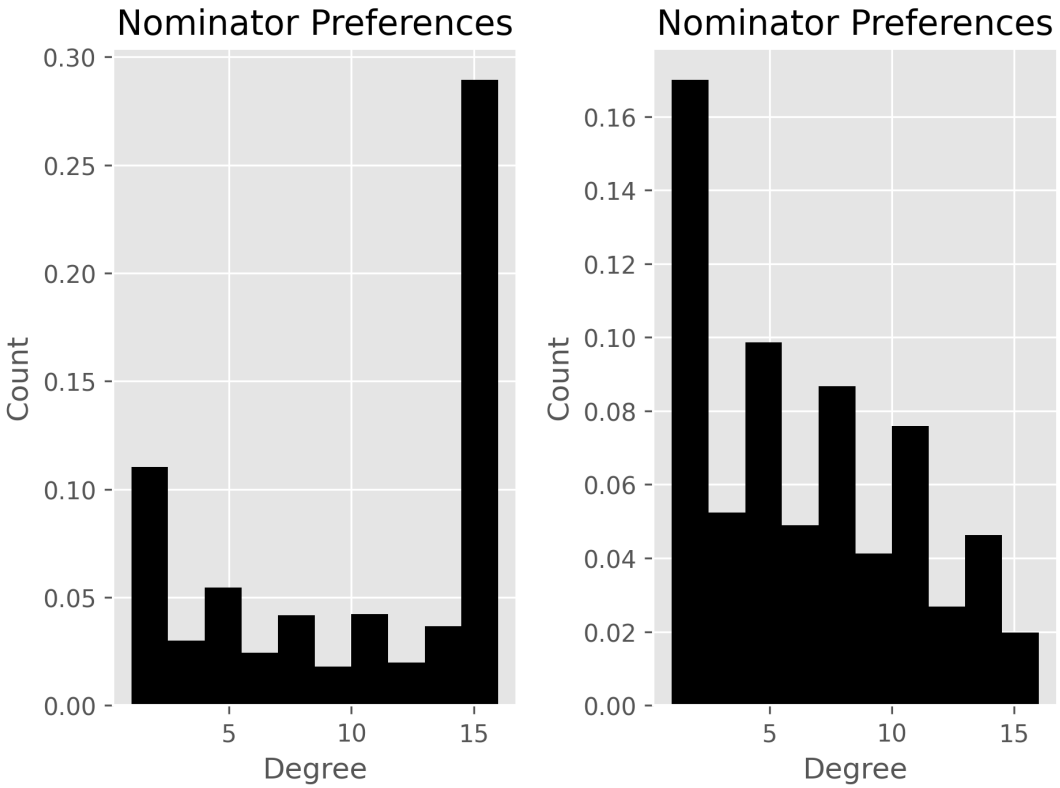


Figure 2.6 – Nominator out-degree distribution snapshot (left) and processed (right)

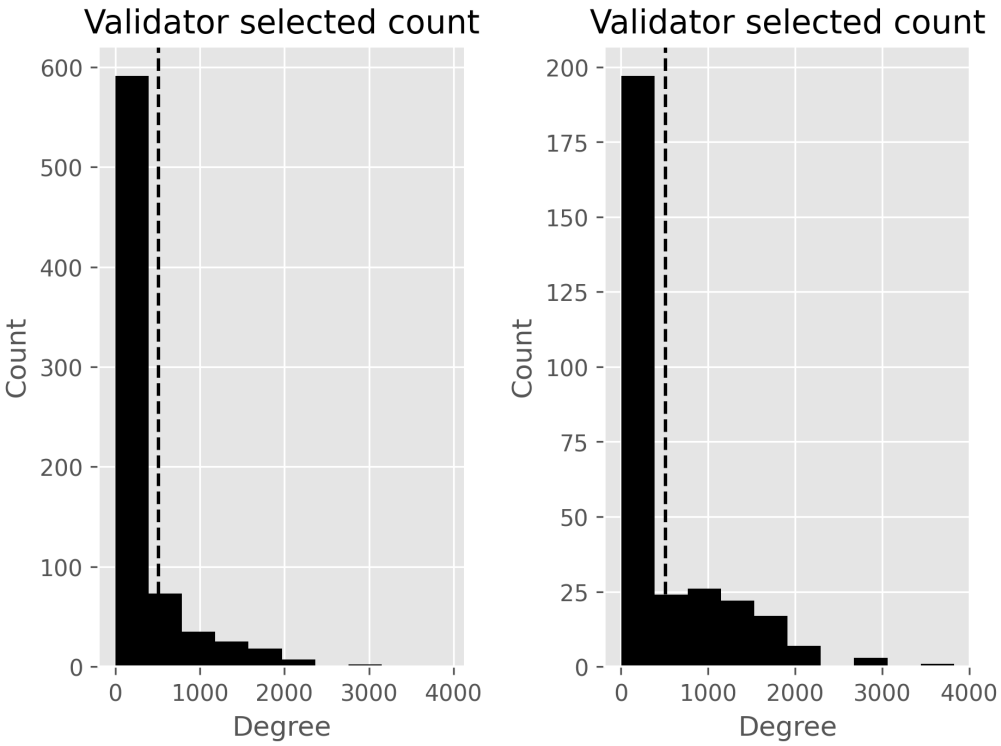
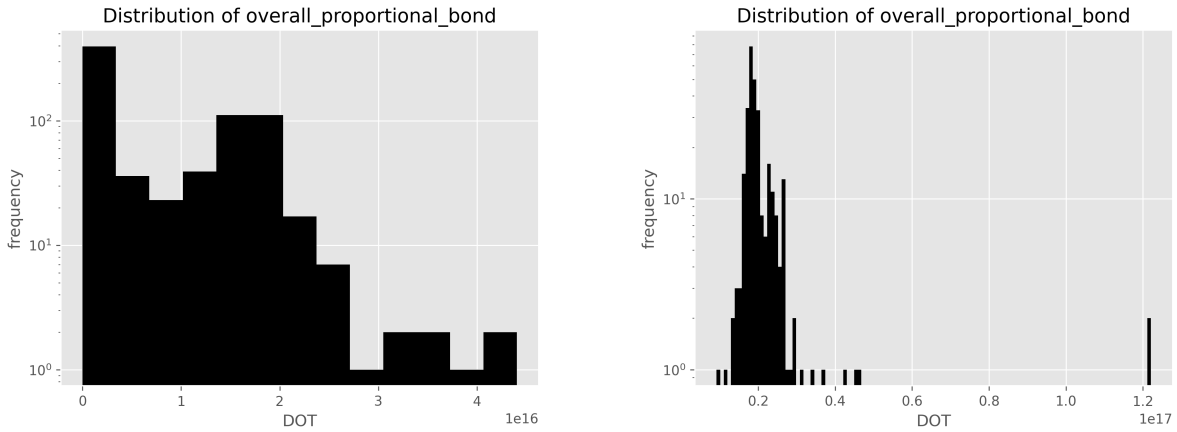


Figure 2.7 – Validator in-degree distribution snapshot (left) and processed (right)



(a) Distribution of overall_proportional_bond era 994

(b) Caption

Figure 2.8 – Overall Proportional Bond distribution prior (left) and post (right) to processing.

The distribution of the overall proportional bond shown in figure 2.8 can be considered bimodal. Comparing the two figures we can observe a big shift in stake upwards. This is shown by the disappearing lower aggregation and the jump in the scale of a magnitude. This can be explained due to the dropout of many validators not making it to the active set. The proportional shares of each validator naturally increase. Of course, there is a share of the stake, which is not accumulated by

any winning validator of the active set. This share will usually be the nominators with lower stakes. With the introduction of nominator pools, they have the opportunity to be included. This should make the system more efficient overall, since less stake is excluded.

2.1.4 Stake Distribution Predictor Processing

In this last processing stage, only one more column, the predictions made by ESSP (the expected sum of stake), is added to the dataframe.

2.1.5 Adjustment

The adjustment phase is a crucial stage in the pipeline, since it ensures that the predictions made by the SDP are globally correct. To reiterate, this hard constraint is defined as the sum of the predictions must equal the bond of the nominator. Any solution not complying will be rejected.
Let us consider the example shown in the dataframe 2.5, the Nominator X votes for validators y and z and has bonded a total of 100 DOT. The solution derived by sequential Phragmén dictates that y receives 60, z receives 40. However, the prediction by SDP predicts an equal split of 40 each. When summing up the predictions and subtracting it from the total bond results in 20 DOT missing. Therefore, the solution must be adjusted in order to be considered correct.

	nominator	validator	total_bond	solution_bond	prediction
0	X	y	100	60	40
1	X	z	100	40	40

Table 2.5 – Example Dataframe for Adjustment

There have been various strategies and sub-strategies implemented to address this issue. They can be run in three different configurations. From fastest to slowest: vectorized, multiprocessing and sequential. In the sequential and multiprocessing the order of nominators, when adjusting, can make a big difference in the final result. The resulting score is inverse to the speed of strategy. The sequential method usually produces the highest quality, however it takes approximately 10 minutes to execute.
list of (sub-)strategies:

- `even_split`. Strategy: Groups by nominator, sums up predictions, takes difference of bond and sum, divides into equal parts and adds/subtracts of all validators. Introduces many edge cases.

- `proportional_adjustment`. Strategy: Groups by nominator, sums up predictions, normalizes the predictions, multiplies the bond with the individual ratios.
- `negative_preadjustment`. Sub-strategy: Sets all negative predictions to 1. We do not set it to 0 to avoid division by 0 zero errors down the line.
- `expected_sum_stake_adjustment`. Sub-strategy: Can be used in combination with either even-split or proportional-split. Dataframe gets grouped by validator. Sum up the predictions and take the difference to the expected_sum_stake. Subtract all the differences by the minimum in order to have no negative values. Then groupby nominator, normalize the differences, add the resulting ratios to the ratios derived by the main strategies or take the maximum of either and normalize again.
- `CVXPY`. Python library designed for solving convex optimisation problems. Objective function: L2 norm of the difference between the predicted values and the original predictions. Always adds up to the corresponding target value (total bond), while also minimizing the difference between the original predictions and the optimized predictions.

It must be stated that there is no evident reason to prefer a combination of strategies over the other. The decision results in testing which configuration is most consistent over the eras. Multiple configurations could be applied to one era to produce the best result. However this approach would go against the spirit of being a pure machine learning approach. The proportional adjustment strategy in combination with expected sum sub-strategy was the approach that seemed to enhance the results while respecting the predictions made by the model.

2.1.6 Scoring and Comparison

In this step, we calculate the relevant score metrics 1.3 by grouping the dataframe by validator. We calculate the minimum sum of stakes, the overall sum of stakes and the variance of the sum of stakes. We then continue to load the stored solution. Contained in it is a claimed score, which we compare to the predicted one in a sequential manner. If we perform worse in any metric the solution is discarded, as Polkadot would do.
The predicted solutions and scores get saved for further analysis.

2.1.7 Utilties

Various utilities have been implemented in order to make the project as user-friendly as possible. First of all *make* files have been configured such that the user can *pull* the repository, setup a new `virtual environment` with the command `python -m venv nameofvenv`, activate it (`source nameofvenv/bin/activate`) and call `make install`. This will install all the dependencies for the

production version. This is the lightweight version. For developers there is the install-dev option, which installs further dependencies for processes such as unit testing. A user can run `make help` to see the full command list provided.
There are configuration JSON files in every module. If the user decides to use them separately he may do so. The main `config.json` is split up in 4 sections: data gathering, model1 - model3. A template can be found in the appendix (A.2).
Ruff is a rust based linter for python scripts. it was used to evade any code quality issues. After fixing the warnings and issues *black* was used for auto-formatting.
Lastly pyproject.toml was used to configure all the developer tools in one place.

2.2 Model Selection

There is a pallet of different models available for the user to select. The implementations used for this project were provided by *sklearn*[4]. In order to justify the main model choice of this project, we have to introduce a couple of topics. Firstly, Decision Trees (DTs) is a supervised learning method than can be used for both classification and regression. A big advantage of DTs is that they do not make any assumptions about the underlying distribution of the data. It aims to partition the data into subsets based on features and attributes. A tree consists of internal nodes representing the features and leaf nodes, which can be labels in the case of a classifier or values in the case of a regressor. DTs are generally considered being able to capture complex relationships in data, however they are prone to *overfitting* (*SciKit Decision Trees* n.d.). Overfitting occurs when a model can only repeat the predictions on data it has previously seen, however fails to provide useful results for unseen data. Gradient boosting is a method that is applied to another machine learning algorithm, in this case the DTs. In short, it tries to reduce the error by aggregating many weak models up until a stopping criterion is met. This can be a maximum number of iterations or an early stopping measure designed to avoid overfitting (*Google Machine Learning Course* n.d.). Light Gradient Boosting Machine (LGBM) is an implementation of a Gradient Boosting Decision Tree (GBDT). The authors claim that LGBM is 20 times faster than other implementation such as *XGBoost*, while achieving almost the same accuracy (Ke et al. 2017). It does this by implementing two techniques: Gradient-based One Side Sampling (GOSS) and Exclusive Feature Bundling (EFB).
GOSS is a method that aims to focus on samples that contribute more to the gain in information. This gain is encoded as a gradient. Now during the sampling it is more likely to drop instances with smaller gradients. This in turn speeds up the learning process significantly (Ke et al. 2017). EFB is a feature reduction method that aims to bundle mutually exclusive sparse columns. These features, such as one-hot-encodings, most often do not have non-zero values in the same row, making it almost lossless (Ke et al. 2017).

These two techniques give LGBM the edge over the other models. The python implementation

[4] https://scikit-learn.org/stable/supervised_learning.html

ensures that is does not overfit by allowing for an introduction of an evaluation set during training and setting of an early stopping parameter. These reasons lead to LGBM being the model selected for the prediction tasks in this thesis.

2.2.1 Hyperparameter tuning

Hyperparameters are possible configurations of a specific machine learning model. They play a crucial role in determining the capabilities of a model. They are set by the data scientist prior to the training process (rather than learned by the model). They define the architecture, learning rate and complexity. For example, in DTs a hyperparameter can be the number of branches. In a neural network this would coincide with the number of nodes or layers within. The possibilities for configuration vary from model to model (*AWS (What is hyperparameter tuning?)* N.d.).
The process of discovering the optimal values is called *hyperparameter tuning*. It involves a process, where models are trained and evaluated with various configurations of the hyperparameters available. The models yielding the best results is the setting that will be applied to the model for future predictions. This iterative process can be done manually or with the help of a framework which systematically explores the hyperparameter space (*AWS (What is hyperparameter tuning?)* N.d.).
This process enables data scientists to fine-tune the performance thus achieving optimal results. An example illustrating the importance: if the learning rate of a model is set to high, the training may take a short time, however the results could be better. In the opposite case, where the learning rate is too low, the model might never converge (*AWS (What is hyperparameter tuning?)* N.d.).
As of today there exist several techniques for finding the optimal configuration. The most popular are: *Bayesian optimization*, *grid search*, and *randomized search*.
Bayesian optimization is based on *Bayes' theorem*, thus including the probability of an event dependent on prior events, to build a probabilistic model of the hyperparameters. Regression analysis is applied to find the optimal configuration (*AWS (What is hyperparameter tuning?)* N.d.).
Grid search tries out all different combinations of configurations possible. This is naturally the computationally most expensive approach (*AWS (What is hyperparameter tuning?)* N.d.).
Random search, similarly to grid search, does not have a sense of direction of improvement. It is less exhaustive but can yield better results with less iterations (assuming the number of hyperparameters is small) than grid search (*AWS (What is hyperparameter tuning?)* N.d.).

As previously mentioned this process can be automated with the help of frameworks. Optuna is a hyperparameter optimization framework written for python that uses state-of-the-art algortihms to efficiently find the optimal configurations. It allows for easy parallelization, speeding up the process even more (*Optuna* n.d.). The default search strategy used is a Bayesian optimization technique. The specific implementation is called *Tree-structured Parzen Estimator algorithm*[5].

[5] https://optuna.readthedocs.io/en/stable/reference/samplers/generated/optuna.samplers.TPESampler.html#optuna.samplers.TPESampler

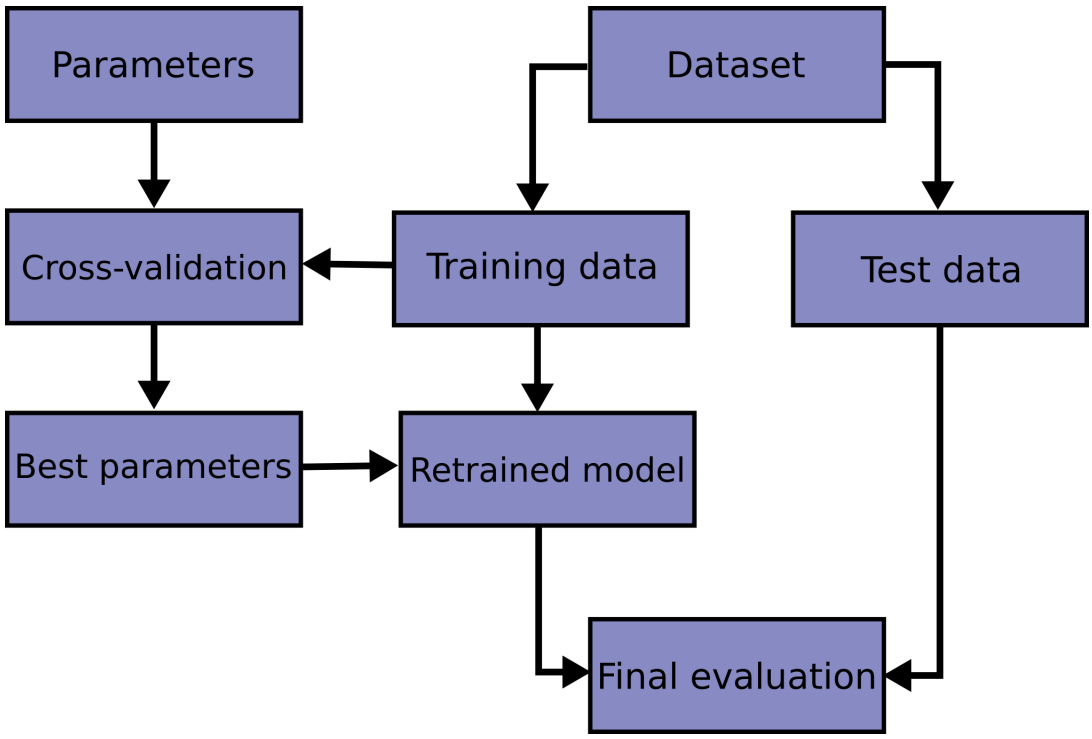


Figure 2.9 – Cross validation Process

Source: https://scikit-learn.org/stable/modules/cross_validation.html

The list of hyperparameters that are available for tuning are listed in the official documentation[6]. The subset that was included in the tuning process is listed in figure 2.11. To prevent overfitting cross-validation was applied. Cross-validation is a technique that holds out part of the training data as a *validation set* (figure 2.9). Since the training now occurs without any knowledge of the *test set*, the model can be more generalizable for unseen data. Now having partitioned the data into three sets, the data available for training might be insufficient. In order to combat this issue one can apply a *k-fold cross-validation* approach. As seen in figure 2.10 it is similar to the basic technique, however it is repeated k-times. In the example shown k is 5. So the training data is split up into 5 folds. k-1 folds are used for fitting the model, while the remainder is used for evaluation. The performance of the model is derived by averaging out the scores.

To account for the time series nature of the training data, the mixing of folds, as described previously, does not work without introducing data leakage. Hence a shift in the data had to be introduced.
For example, if we consider the era range of 910-915 and assume that three previous eras are required to predict one era, we require the range of 907-915. The first iteration would take eras 907-909 to fit the model for era 910, second iteration 908-910 for era 911 and so on. This technique ensures that the data used for training precedes the target era, thus preventing any data leakage.
For the SDP the user may select to minimize the Mean Squared Error (MSE) or maximize the lowest

[6] https://lightgbm.readthedocs.io/en/latest/pythonapi/lightgbm.LGBMClassifier.html

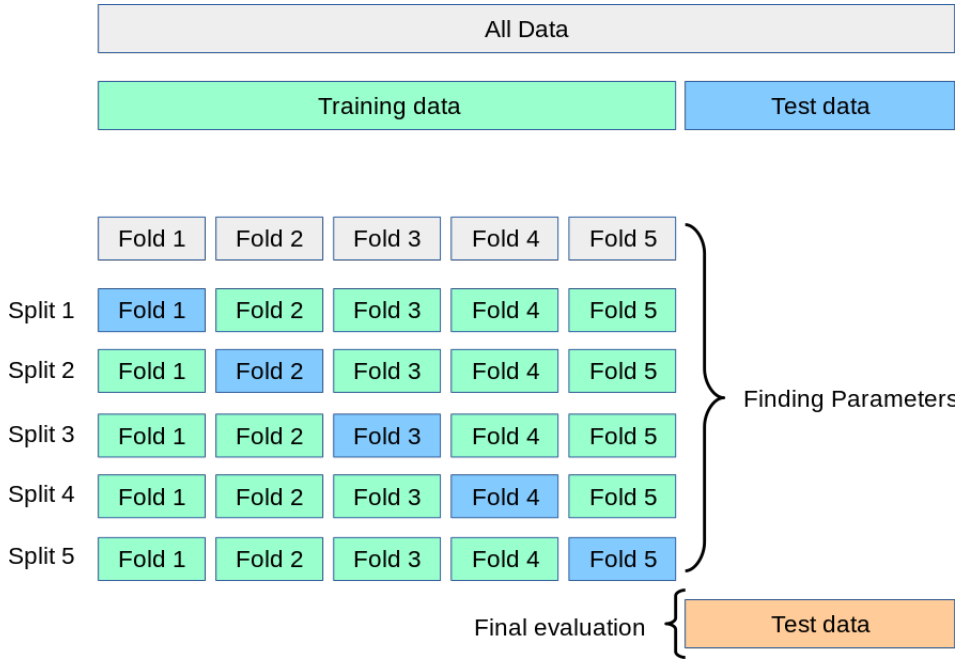


Figure 2.10 – K=5 - fold cross-validation

Source: https://scikit-learn.org/stable/modules/cross_validation.html

sum of stake. The second evaluation technique was introduced to potentially outperform the solution provided by sequential Phragmén rather than approach it.
The hyperparameter configurations discovered by optuna were included in the models and are not adjusted when predicting the test eras in order to show the true performance of the trained model.
Figure 2.11 shows the the comparative hyperparameter importance. For the ASP it seems that the learning_rate was most significant in producing results with the highest accuracy across five folds. However, this is highly dependent on the number of iterations executed and what search strategy was implemented. Figure 2.12 shows the different configurations of hyperparameters the specific search strategy used. The darker the path the higher the resulting accuracy. We can clearly see that the Bayesian technique is trying to improve upon previous configurations which produced good results instead of randomly mixing them.
Each of the three models got their individual tunings in order to maximize performance.
Optuna provides a dashboard to display visualisations regarding the tuning process. It can be displayed with the following command: `optuna-dashboard "sqlite:///db.sqlite3"`.

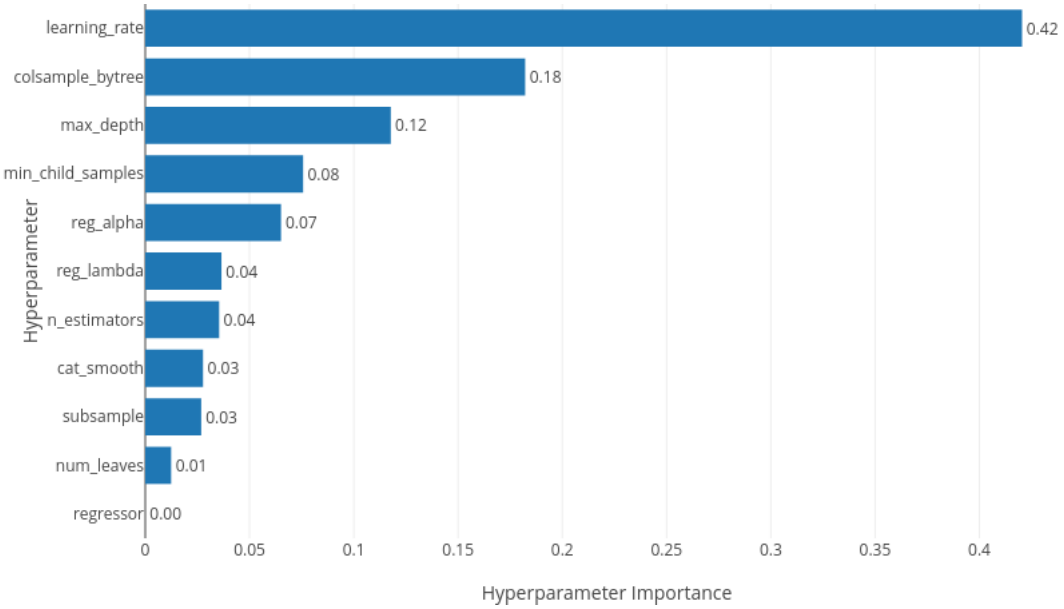


Figure 2.11 – Hyperparameter Importance in LGBM for Expected Sum Stake Predictor

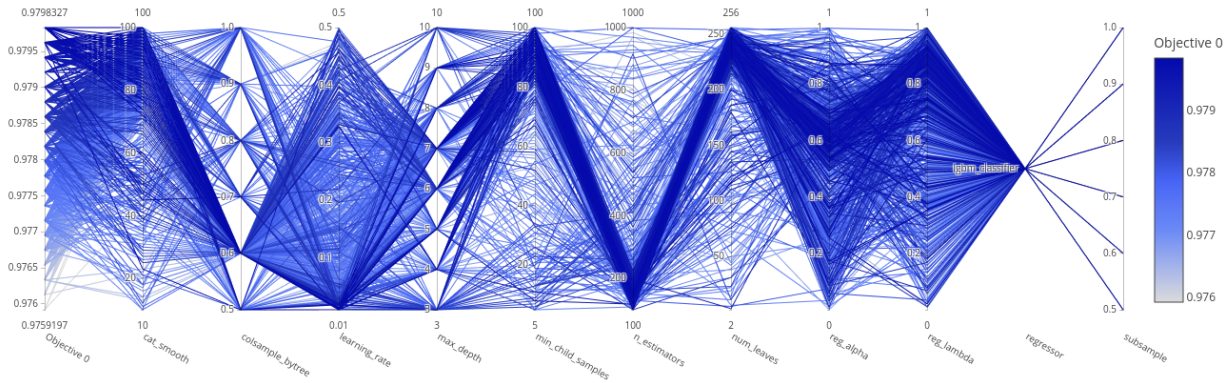


Figure 2.12 – Different Hyperparameter configurations tried out in 1000 trials.

Chapter 3

Results

This chapter displays the performance of the individual models. To provide a true insight into their performance, eras past 1000 have been reserved for the final predictions. This is supposed to simulate the performance on previously *unseen* data. The training has been performed on eras prior to 1000. To reiterate, the first model, ASP, is a classifier aiming to predict the future active sets. The second model, ESSP, is a regressor aiming to predict the expected sum of stake each validator receives. The last model, SDP, is a regressor aiming to predict the nominators stake distribution. Additionally, the final model, SDP, will display the scores it achieved and how it compared to the sequential Phragmén over the course of one month.

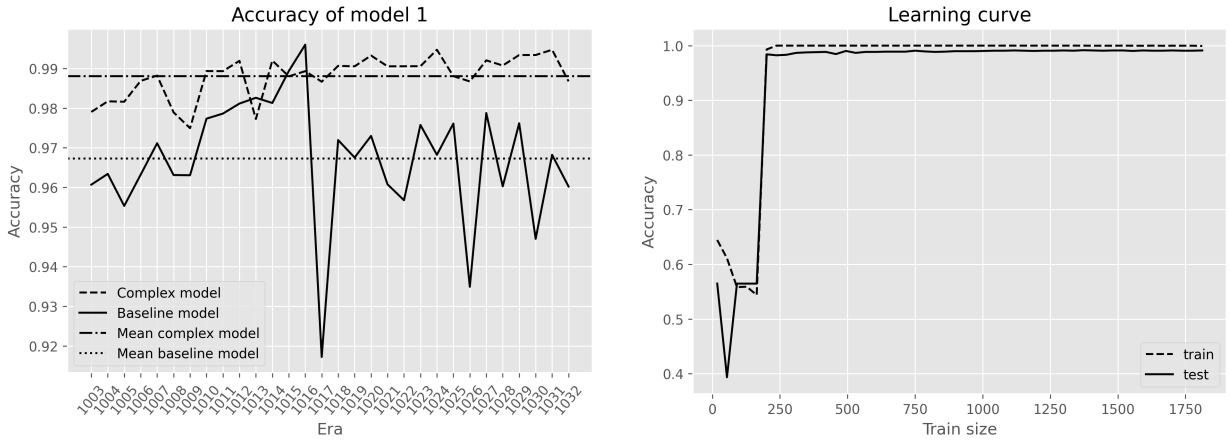
3.1 Active Set Predictor

Figure 3.1a depicts the accuracy of both the basic and complex model. The complex model contains all features that were engineered and achieves a mean accuracy close to 99%. The baseline model, which only includes one feature, namely the binary column elected_previous_era, achieves a mean accuracy close to 97%. This performance aligns closely with the insights provided in section 2.1.2. There, we showed that the active set does not change significantly. Only 10 validators on average of the active set changes from era to era in figure 2.1b. 10 validators of 297 corresponds to 3.3% which justifies the baseline models (mean) accuracy of 97%.
Figure 3.1b shows the *learning curves* for the ASP. Learning curves can give insights into the predictive capabilities of a model. It contains two curves, one for the training and one for the test set. Their direction and relationship demonstrate the four states a model can be in, namely good fit, underfit, overfit and not representative. For the ASP it shows signs of a good fit, due to the fact that the accuracy is high for both the training and test learning curves. They rise quickly, remain in close proximity to each other and remain stable across the increase in training size.

Lastly, to provide another measure of quality we generate a classification report including three evaluation metrics. Unsurprisingly, the resulting values in figure 3.2 are high.

- Precision is defined as: *tp*[1] */ (tp +* *fp*[2] *)*. This metric is supposed to give an insight into the capabilities of the model to avoid labeling a negative result as a positive one (*SciKit Model evaluation* n.d.).
- Recall is defined as: *tp / (tp + fn)*. This metric demonstrates how well the model is able to find the positive results (*SciKit Model evaluation* n.d.).
- F1 is the weighted harmonic mean of the previous two metrics, with 1 being the best and 0 being the worst evaluation (*SciKit Model evaluation* n.d.).

$$F1 = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$



(a) Accuracy of model 1 across one month

(b) Learning Curve

Figure 3.1 – Active Set Predictor Performance

[1]true positives: predicted true and reality is true
[2]false positives: predicted true and reality is false

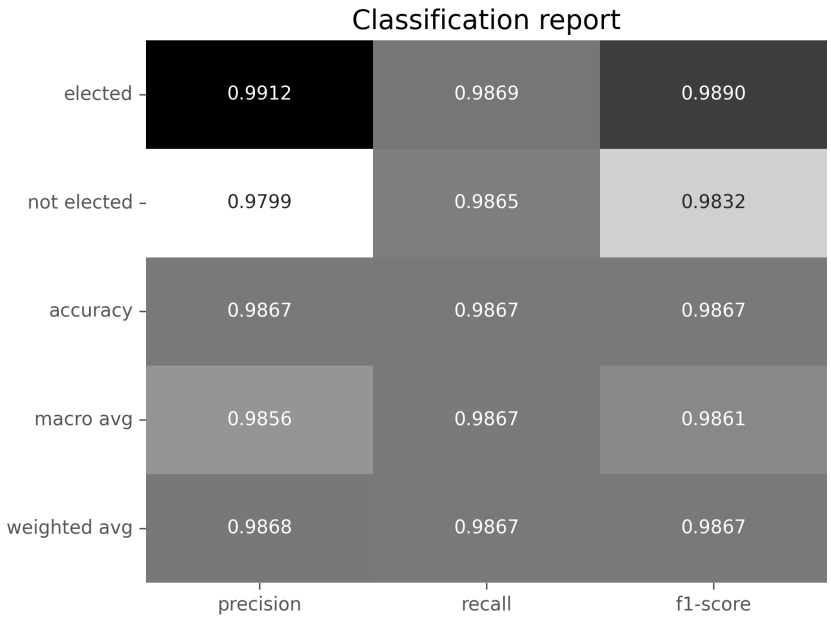


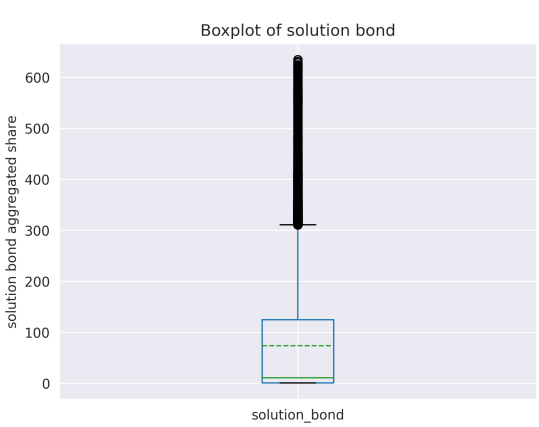
Figure 3.2 – Classification report Active Set Predictor

3.2 Expected Sum Stake Predictor

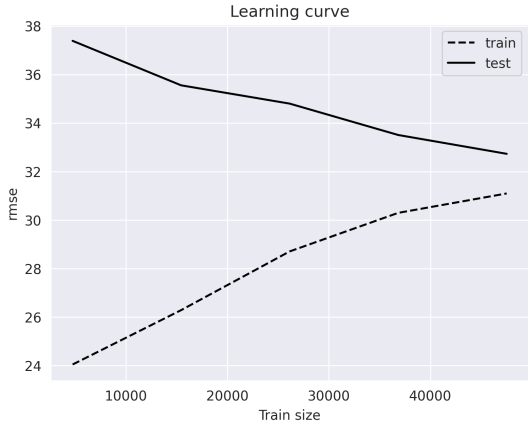
Figure 3.4a depicts a mean Root Mean Squared Error (RMSE) (Definition: 3.1) of approximately 33. The RMSE is considered more sensitive to outliers, due to the errors being squared. It is considered a conservative metric for model evaluation. To demonstrate to what extent the outliers can inflate the error term, we also provide the prediction error as the Mean Absolute Error (MAE)(Definition: 3.2). The mean has sunken from 33 to 20.
To give some context to the error terms, we provide the boxplot of the target variable 3.3a. The target variable has a mean value of 73.39 with a standard deviation of 106.43.
Lastly, the learning curves, which use the RMSE as a metric, (figure 3.3b), show a slight bias by the training error not being close to 0 and low variance by converging. The low variance can also be observed in figure 3.4a by the small amount of fluctuation of prediciton errors.
This analysis suggests that this second model has an acceptable to good ability to generalize.

$$\text{RMSE}(y, \hat{y}) = \sqrt{\frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{N}} \tag{3.1}$$

$$\text{MAE}(y, \hat{y}) = \frac{\sum_{i=0}^{N-1}|y_i - \hat{y}_i|}{N} \tag{3.2}$$

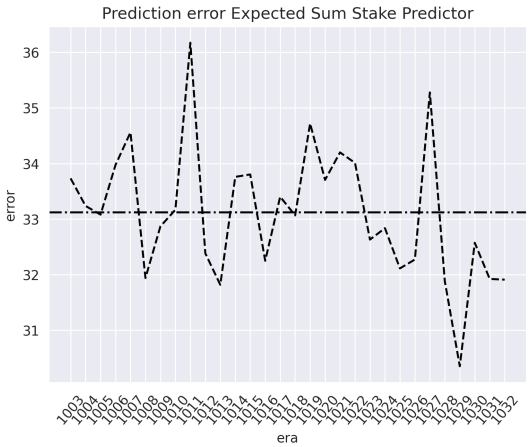


(a) Aggregated Share of Solution Bond Boxplot

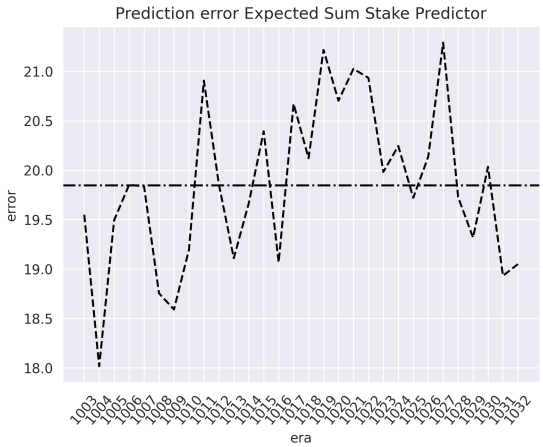


(b) Learning curve with RMSE

Figure 3.3 – Expected Sum Stake Predictor Performance



(a) RMSE across one month



(b) MAE across one month

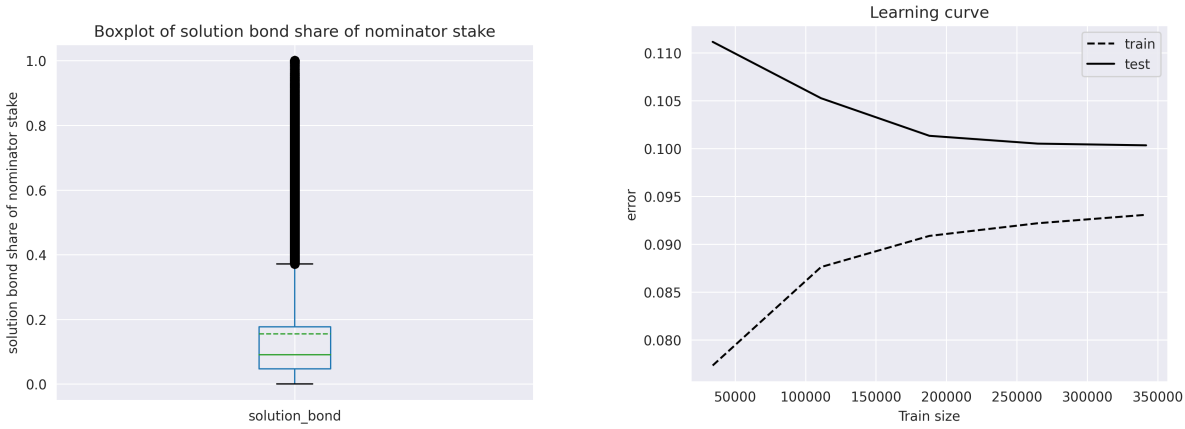
Figure 3.4 – Expected Sum Stake Predictor Performance

3.3 Stake Distribution Predictor

Figure 3.6a shows a mean RMSE of approximately 0.1 and a mean MAE of 0.069. Putting these figures in context with the help of the boxplot in figure 3.5a, we find the level of precision to be insufficient for making accurate predictions. The target variable is now the ratio of stake the validator receives of one nominator. Making a mistake of 0.1 (conservative MSE) on average, when the values range from 1 to 0, is insufficient.

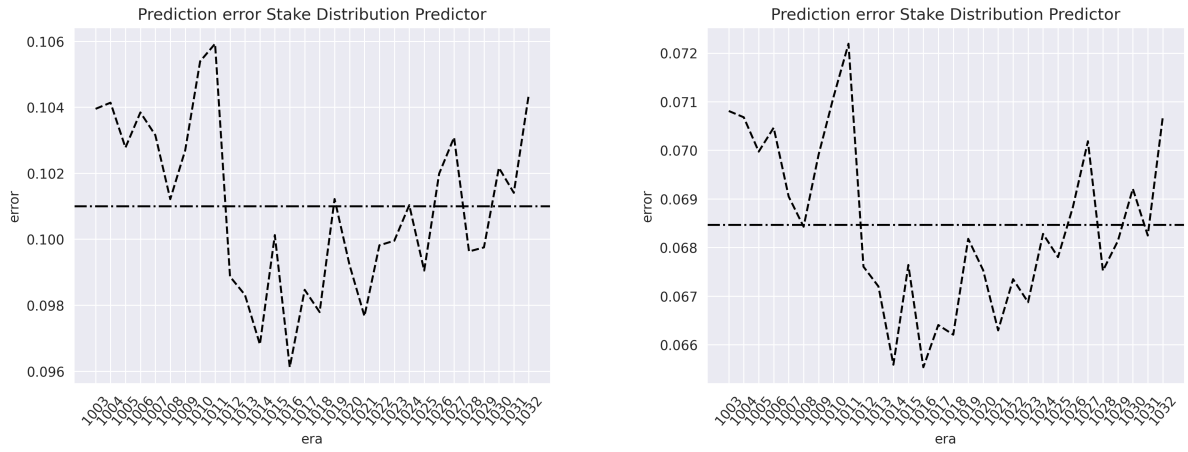
Looking at the learning curve (figure 3.5b), we see that the model is trained well. It has a gap between the train and test curve, which can be considered to be the generalization error. The gap between the train error and 0 is the bias of the model. It suggests that the model requires more features that

can supply it with crucial information to make accurate predictions.
The main issue is, that there is no inherent grouping possible without exploding the dimensions to a space where the model cannot learn. In an optimal scenario the model would recognize, that the nominator stake must be distributed among the validators they elected (the sum of stake must equal the bonded amount). Unfortunately, the model is optimizing on a row level and not on a grouped level.
Another source of bias is that a slight change in active set may produce a significant change in value of the target variable. The sequential Phragmén might redistribute the stake completely than in the prior era. The model now receives the identical input data for a row and a big change in target variable. This can offset the predictive capabilities significantly.



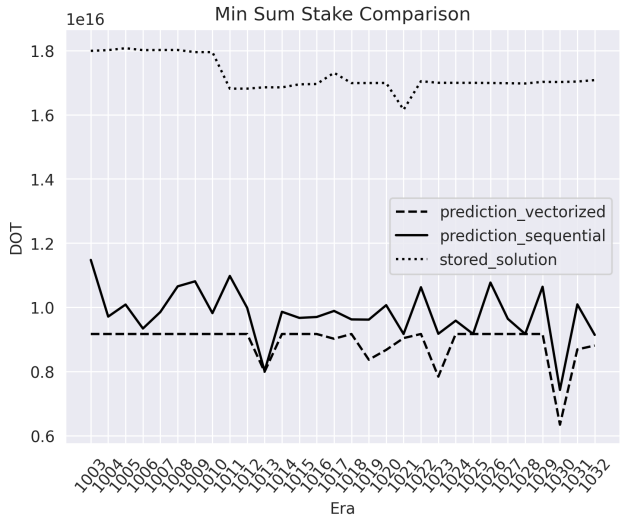
(a) Share of Total bond Boxplot **(b)** Learning curve

Figure 3.5 – Boxplot and Learning Curve for Stake Distribution Predictor

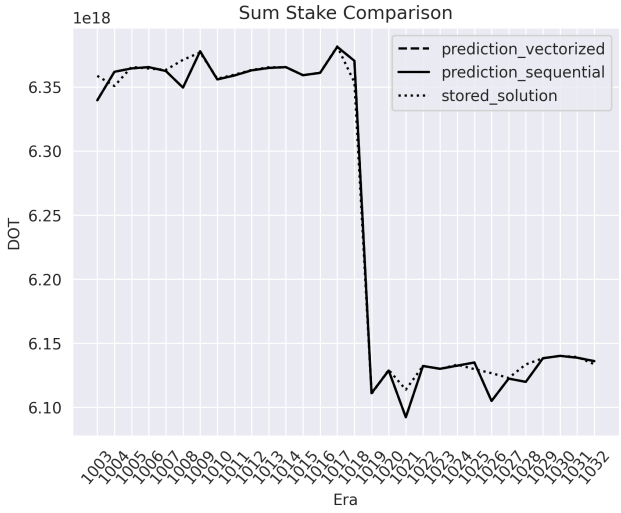


(a) RMSE **(b)** MAE

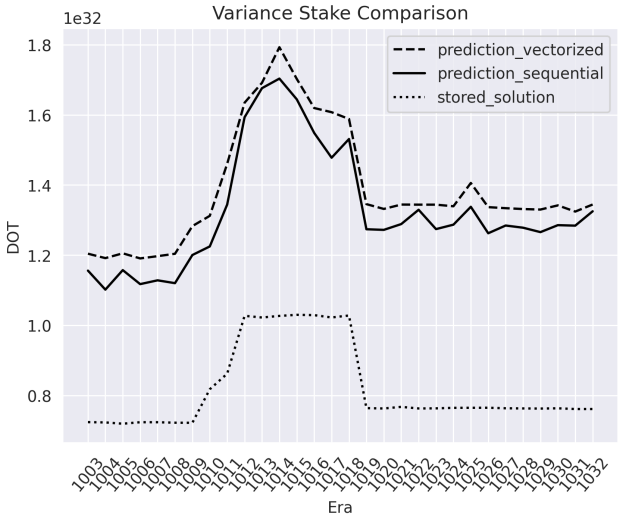
Figure 3.6 – Prediction error across one month



(a) (Maximise) Min Sum Stake



(b) (Maximise) Sum Stake



(c) (Minimise) Variance of Stake

Figure 3.7 – Scores of sequential Phragmén, predicted solutions vectorized and sequential adjustment

Now looking at the score comparison across our test month in figure 3.7 we see that we did not outperform the model when it comes to maximizing the minimal sum of stakes or minimizing the variance of stakes. However, we did outperform it in approximately a third of the month in maximising the overall sum of stake. Furthermore, we can clearly see the difference it makes in how the adjustment process takes place. The sequential adjustment technique has more of an opportunity to make use of the expected sum predictions to approach a more optimal solution. CVXPY produced the best results, however they are not included since the process can take over an hour, which makes it infeasible.

Unfortunately, if we recall how the solutions get scored (scored in order of minimum sum, sum stake, variance stake) all solutions, which are part of this test era, would be fail to beat the sequential Phragmén approach. In the training set prior to era 1000, we did successfully beat it a handful of times. Those victories were purely coincidental. However, they did show that the solutions provided by sequential Phragmén are indeed not optimal and can be improved upon.

The key-issue is that if the model was able to perfectly predict the target variable, we would only be as good as the sequential Phragmén. In a supervised approach, the models must be trained on global, or close to, optima.

Lastly, the scope of optimization is not in sync. We can reduce this prediction problem to three scopes. For one the local, row based one, secondly, the group of validators, which are elected by one nominator, and lastly, globally, the active set as a whole. While the model is focusing on a local level, it ignores the more global relationship the different predictions have with each other and as a consequence fails to achieve meaningful predictions.

Chapter 4

Conclusion

As Polkadot is growing, it becomes evident that a need for scaling of the staking system emerges. The current implementation, the sequential Phragmén, has its inherent limitations due to its size dependent time complexity. The limitations are reflected in the maximum number of nominators (22'500) that are allowed to participate, each of them selecting a maximum of 16 validators. The goal was to deliver a machine learning approach that can address the problem by providing equal or better solutions, while scaling better than the current implementation.

A pipeline was developed that gathers data via storage queries or postgreSQL database. It transforms it such that it is suitable for machine learning models. The overall prediction task was broken down into three models. The first two are delivering great results, allowing for accurate predictions of future eras. These results can be used as features and adjustment guide for subsequent models.

The third and final prediction stage, the SDP, proved to be more difficult. Although the model itself was trained reasonably well, there was a bias, which resulted in the production of sub-optimal solutions. The key discovery made is that the scope of optimization cannot be fully captured by a supervised model. While the model is aiming to predict on a local level, the optimization should simultaneously take place on a global level. Futhermore, a slight change of the active set may change the values of the target variable, the distribution of bond, completely for that specific nominator-validator row. This is likely to be the cause of the bias present in the model.

Outperforming the sequential Phragmén with a supervised model approach seems to be difficult, due to the fact that a model perfectly predicting the target variable can only be as good as the sequential Phragmén. Therefore, to successfully predict better solution one must either apply various solvers to find more optimal solutions than the ones discovered by sequential Phragmén or implement an unsupervised model approach that uses the three score metrics as a fitness function.

4.1 Future Work

Diving deeper into the data allowed us to closely analyze the staking system and its intricacies. It showed that its highly complex and most likely misunderstood by the average user. We were surprised at how little the active set composition changes from era to era. Additionally, it became evident that the nominators do not actively tend to their validator preferences, since many of them included inactive validators. To address this issue it is suggested that an auxiliary tool is developed. This tool would evaluate the requirements defined by nominators, such as comission, activity levels, nominator count and more, and return a list of validators which would match them best. It would be optimal, if Polkadot enables the nominator to submit this *extended preference list*. Polkadot could reduce this extended list each era to a subset that fills the 16 slots available. This could ultimately increase the convenience, reduce the need to be active and ultimately increase the overall efficiency and security of Polkadots staking system by introducing a higher diversity to the active sets.

Working with CVXPY showed that the solution (derived by sequential Phragmén) with the current composition of the active set is very close to (or possibly at) the global optimum. This suggested to me, that if there is an opportunity to outperform the sequential Phragmén, it will be with a different active set composition. As I mentioned in the conclusion, that in order to make the supervised model approach work, there is a need for the best possible solutions. Different solvers could be applied to the snapshots in order to find them. My recommendation is applying them to different active sets than the ones set by sequential Phragmén. If better solutions exist, it would inadvertently be beneficial for the staking system overall. If it is not possible to gather superior solutions, an unsupervised machine learning approach seems more suitable. I would then recommend a reinforcement learning approach using the three score metrics 1.3 as a fitness function.

There is a clear need for more sophisticated adjustment strategies. The implementations in this project aim to keep the adjustments as true as possible to the initial predictions. They improve the score of the solution, however there is a lack of justification why a certain strategy works better than the other. So there should be work in developing a good approach or at least work in justification as to why one works better than the other.

Of course one could tackle the problem from the other side and try to implement constraints for the model, such that the predictions already fit the criteria. However, from what I have discovered there exist only row wise constraints, meaning that we could set a constraint for that the prediction must be positive and smaller or equal to the total bond available. However, this approach still requires adjustment due to the predictions of a group of validators, elected by one nominator, possibly exceeding the available bond to them.

Acknowledgment

I would like to thank Prof. Dr Claudio J. Tessone for giving me the opportunity to write my Master's thesis in the Blockchain and Distributed Ledger Technologies Group.

I would also like to thank my supervisor Dr. Matija Piškorec for his continuous support throughout this thesis. His expertise and feedback were extremely valuable and contributed to the success of this thesis.

I would also like to express my gratitude to my family and friends for their love and full support throughout my studies. Without them this journey would have been significantly more difficult, if not impossible. Thank you for enabling me to overcome all the challenges that were posed to me during my studies.

Appendices

Appendix A

Appendix

A.1 List of Acronyms

BCO Blockchain Observatory 2

PoS Proof of Stake 2

NPoS Nominated Proof-of-Stake 1

JSON JavaScript Object Notation 18

CSV Comma-separated values 18

BDLT Blockchain and Distributed Ledger Technologies Group 2

MSE Mean Squared Error 31

RMSE Root Mean Squared Error 36

ASP Active Set Predictor 15

A.2 Configuration File main.py

```
{
  "username": "username",
  "password": "password",
  "database": "database",

  "start_block": "11314529",
  "end_block": "14184902",
  "block_numbers_path": "./data_collection/block_numbers/events.json",

  "era": 1033,
  "block_numbers": "./data_collection/block_numbers/new_block_numbers_dataframe.parquet",
  "config_path": "config.json",
  "intermediate_results_path": "./data_collection/data/intermediate_results/",


  "model_1_path": "./data_collection/data/processed_data/model_1_data",
```

```
  "model_1": "lgbm_classifier",
  "features_1": [
    "overall_total_bond",
    "overall_proportional_bond",
    "nominator_count",
    "elected_previous_era",
    "era"
  ],
  "target_1": "elected_current_era",


  "model_2_path": "./data_collection/data/processed_data/model_2_data",
  "model_2": "lgbm_model_2",
  "model_2_load": "./models/trained_models/lgbm.pkl",
  "scaler_2_load": "./models/trained_models/lgbm_scaler.pkl",
  "features_2": [
    "validator",
    "proportional_bond",
    "total_bond",
    "validator_frequency_current_era",
    "probability_of_selection",
    "validator_centrality",
    "era"
  ],
  "target_2": "solution_bond",



    "model_3_path": "./data_collection/data/processed_data/model_3_data",
    "model_3": "lgbm_model_3",
  "features_3": [
    "nominator",
    "validator",
    "proportional_bond",
    "total_bond",
    "overall_total_bond",
    "overall_proportional_bond",
    "era",
```

```
    "number_of_validators",
    "validator_frequency_current_era",
    "average_proportional_bond",
    "average_total_bond",
    "nominator_index",
    "validator_index",
    "nominator_centrality",
    "validator_centrality",
    "probability_of_selection",
    "expected_sum_stake"
  ],
    "target_3": "solution_bond",
    "compare": "./data_collection/data/calculated_solutions_data/",
    "plot": null,
  "save": null,

  "node": {
    "url": "wss://rpc.polkadot.io",
    "ss58_format": 0,
    "type_registry_preset": "polkadot"
  },

}
```

A.3 pyproject.toml file

This is boilerplate code found on github. [1]

```
    [build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "data_collection"
version = "0.1"
readme = "README.md"
requires-python = ">=3.8.10"
```

[1] https://github.com/duarteocarmo/boilerplate

```
dependencies = ["numpy>=1.23.5", "pandas>=1.5.2", "substrate-interface>=1.4.2", "pyarrow>=11.

[project.optional-dependencies]
dev = [
  "black>=23.1.0",
  "ruff>=0.0.47",
  "pip-tools>=6.12.3",
]

[tool.ruff]
line-length = 79
ignore = ["E501"]
exclude = [".env", ".venv", "venv", "notebooks"]

[tool.coverage.paths]
source = ["src"]

[tool.coverage.run]
branch = true
relative_files = true

[tool.coverage.report]
show_missing = true
fail_under = 80

[tool.black]
line-length = 79
extend-exclude = '''
/(
  | .env
  | .venv
  | venv
  | notebooks
)/
'''
```

A.4 Make File

This is boilerplate code found on github. [2]

```
    .PHONY: install clean lint format

## Install for production
install:
        @echo ">> Installing dependencies"
        python -m pip install --upgrade pip
        python -m pip install -e .

## Install for development
install-dev: install
        python -m pip install -e ".[dev]"

## Build dependencies
build:
        pip-compile --generate-hashes --resolver=backtracking --output-file=requirements.txt
        pip-compile --generate-hashes --resolver=backtracking --extra=dev --output-file=requi

## Delete all temporary files
clean:
        rm -rf .ipynb_checkpoints
        rm -rf **/.ipynb_checkpoints
        rm -rf .pytest_cache
        rm -rf **/.pytest_cache
        rm -rf __pycache__
        rm -rf **/__pycache__
        rm -rf build
        rm -rf dist

## Lint using ruff
ruff:
        ruff .

## Format files using black
format:
```

[2] https://github.com/duarteocarmo/boilerplate

```
        ruff . --fix
        black .

## Run tests
test:
        pytest --cov=src --cov-report xml --log-level=WARNING --disable-pytest-warnings

## Run checks (ruff + test)
check:
        ruff check .
        black --check .




################################################################################
# Self Documenting Commands                                                    #
################################################################################

.DEFAULT_GOAL := help

# Inspired by <http://marmelab.com/blog/2016/02/29/auto-documented-makefile.html>
# sed script explained:
# /^##/:
#         * save line in hold space
#         * purge line
#         * Loop:
#                 * append newline + line to hold space
#                 * go to next line
#                 * if line starts with doc comment, strip comment character off and loop
#         * remove target prerequisites
#         * append hold space (+ newline) to line
#         * replace newline plus comments by `---`
#         * print line
# Separate expressions are necessary because labels cannot be delimited by
# semicolon; see <http://stackoverflow.com/a/11799865/1968>
.PHONY: help
help:
```

```
	@echo "$$(tput bold)Available commands:$$(tput sgr0)"
	@sed -n -e "/^## / { \
		h; \
		s/.*//; \
		:doc" \
		-e "H; \
		n; \
		s/^## //; \
		t doc" \
		-e "s/:.*//; \
		G; \
		s/\\n## /---/; \
		s/\\n/ /g; \
		p; \
	}" ${MAKEFILE_LIST} \
	| awk -F '---' \
		-v ncol=$$(tput cols) \
		-v indent=19 \
		-v col_on="$$(tput setaf 6)" \
		-v col_off="$$(tput sgr0)" \
	'{ \
		printf "%s%*s%s ", col_on, -indent, $$1, col_off; \
		n = split($$2, words, " "); \
		line_length = ncol - indent; \
		for (i = 1; i <= n; i++) { \
			line_length -= length(words[i]) + 1; \
			if (line_length <= 0) { \
				line_length = ncol - indent - length(words[i]) - 1; \
				printf "\n%*s ", -indent, " "; \
			} \
			printf "%s ", words[i]; \
		} \
		printf "\n"; \
	}' \
	| more $(shell test $(shell uname) = Darwin && echo '--no-init --raw-control-chars')
```

Bibliography

AWS (What is hyperparameter tuning?) (N.d.). Accessed: 2023-07-08. URL: https://aws.amazon.com/what-is/hyperparameter-tuning/.

Brill, Markus et al. (Feb. 2017). "Phragmén's Voting Methods and Justified Representation". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 31.1. DOI: 10.1609/aaai.v31i1.10598. URL: https://ojs.aaai.org/index.php/AAAI/article/view/10598.

Cevallos, Alfonso and Alistair Stewart (2020). "Validator election in nominated proof-of-stake". In: *CoRR* abs/2004.12990. arXiv: 2004.12990. URL: https://arxiv.org/abs/2004.12990.

Google Machine Learning Course (n.d.). Accessed: 2023-07-08. URL: https://developers.google.com/machine-learning/decision-forests/intro-to-gbdt.

Janson, Svante (2018). *Phragmén's and Thiele's election methods*. arXiv: 1611.08826 [math.HO].

Ke, Guolin et al. (2017). "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., pp. 3149–3157. ISBN: 9781510860964.

Optuna (n.d.). Accessed: 2023-07-08. URL: https://optuna.org/.

Paimani, Kian (n.d.). URL: https://www.youtube.com/watch?v=MjOvVhc1oXw&t=2590s&ab_channel=ParityTech.

Polkadot FAQ (n.d.). Accessed: 2023-07-08. URL: https://support.polkadot.network/support/solutions/articles/65000181959-staking-faq-s.

Polkadot Support Slashing (n.d.). Accessed: 2023-06-16. URL: https://support.polkadot.network/support/solutions/articles/65000110858-what-does-it-mean-to-get-slashed-%5C#What-could-cause-your-tokens-to-get-slashed?.

Polkadot Wiki (learn Phragmen) (n.d.). Accessed: 2023-06-22. URL: https://wiki.polkadot.network/docs/learn-phragmen.

Polkadot Wiki (learn Staking advanced (n.d.). Accessed: 2023-06-29. URL: https://wiki.polkadot.network/docs/learn-staking-advanced.

Polkadot Wiki (learn staking) (n.d.). Accessed: 2023-06-20. URL: https://wiki.polkadot.network/docs/learn-staking.

Rüegsegger, Florian (2023). "Inter-chain Data collection pipeline for the Polkadot ecosystem". unpublished.

SciKit Decision Trees (n.d.). Accessed: 2023-07-08. URL: https://scikit-learn.org/stable/modules/tree.html.

SciKit Model evaluation (n.d.). Accessed: 2023-07-08. URL: https://scikit-learn.org/0.15/modules/model_evaluation.html.

Universität Zürich[UZH]

Institut für Informatik

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel (inklusive generativer KI wie z.B. ChatGPT) angefertigt habe. Mir ist bekannt, dass ich die volle Verantwortung für die Wissenschaftlichkeit des vorgelegten Textes selbst übernehme, auch wenn (nach schriftlicher Absprache mit der betreuenden Professorin resp. dem betreuenden Professor) KI-Hilfsmittel eingesetzt und deklariert wurden. Alle Stellen, die wörtlich oder sinngemäss aus veröffentlichten oder nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Zürich, 14.7.23 B. Murphy

Unterschrift Student:in

5/5 31.05.2023/db/gs/nl