A Decentralised Peer-Prediction Market

CS907 Dissertation Project - Dissertation

Thomas Archbold
1602581

Department of Computer Science
University of Warwick

September 3, 2020

Supervised by Matthias Englert

Contents

1 Introduction

Prediction markets are exchange-traded markets[1] which allow users to trade on the outcomes of future events as opposed to traditional financial instruments. Users participate by placing bets and buying or selling shares in the markets these bets give rise to. Since users stake their own money, market prices should indicate the true beliefs of the userbase and the perceived likelihood the events have of occurring. Different users will have different beliefs and knowledge informing their decisions, and prediction markets provide a means of aggregating information on events of interest using "the wisdom of the crowd". Shares in these markets are usually traded between \$0 and \$1, and for binary events a market will typically pay out \$1 for every share held for a positive outcome, and \$0 otherwise.

Traditional prediction markets are centralised in the sense that the system provides the bets for the users to trade in, and traders simply choose the price point and size of stake at which to participate. Since the system knows all possible bets beforehand it is simple to allow it to determine the outcomes of the markets and pay out the winnings to stakeholders accordingly. There are two issues with this centralised approach: firstly, it restricts the types of bets that can be made, since they must be explicitly offered by the market maker; secondly, it operates on trust – there is nothing to stop the central market maker from manipulating the system for their own gain. We aim for a system that avoids both of these issues.

In this project we implement a *decentralised* prediction market in which it is up to the userbase itself to define the markets and determine the outcomes of these events – these outcomes are decided upon by consensus among a group of users, known as arbiters. This removes the need for a trusted centre, however with no central moderator bets may become ambiguous or their outcomes subjective, and arbiters may still attempt to manipulate the outcome of the market for their own gain by submitting false reports. It is also important that users continue to act according to their true beliefs so we may learn about the true public sentiment on the events. We base our design on the incentive compatible outcome determination mechanism proposed by Freeman, Lahaie, and Pennock [11], which allows us to crowdsource market outcomes while incentivising users to act truthfully in all stages of the prediction market.

The rest of this dissertation is structured as follows: in Section 2 we give an overview of the current literature on algorithmic mechanism design and prediction markets, and discuss recent implementations of decentralised markets. In Section 3 we outline the motivations for undertaking the project and why we believe it to be worthwhile, and detail the successes that have been achieved so far. Section 4 covers the high-level design of the market and introduces the theoretical model upon which our implementation is based, as well as issues related to project management and ethical considerations. In Section 5 we then discuss our implementation of the prediction market, including the tools we have used to do so. Finally in Section **??** we reflect on the project's successes and failures and suggest areas for further development.

2 Background

2.1 Setup

Consider the following problem within algorithmic mechanism design known as the *information aggregation problem* [18, Ch. 26]. An individual known as the "aggregator" wishes to obtain a prediction about an uncertain variable which will be realised at some point in the future. There are a number of individuals known as the "informants" who each hold sets of information about the variable's outcome. The goal is to design a mechanism that extracts the relevant information from the informants and uses it to provide a prediction of the variable's realisation. In an ideal setting the mechanism should produce the same prediction as an omniscient forecast that has access to all information available to all informants. This may not be viable in practice, since each agent's information is private information, and so the mechanism must incentivise them to act in the desired truth-telling manner.

A prediction market is one mechanism that can provide such a forecast. In this setting the aggregator creates a financial security whose payoff is tied to the outcome of the variable. In the simpler case of binary events, such a security may pay out \$1 for each share held if the variable has a "true" or "yes" outcome, and \$0 otherwise, however markets can be created for other types including discrete ("will the result of the match be a home win, away win, or a draw?"), continuous ("what will be the highest measured temperature in Coventry in September?"), or any combination of these types. Informants are then able to participate in the market induced by the security by trading shares according to their

[1] Markets in which all transactions are routed through a central source.

beliefs: those who believe, for example, that global warming is real might buy shares at a given price in the market, "the global average temperature in 2020 will be higher than that in 2019", while deniers may be inclined to sell shares. The share price will be adjusted accordingly, and the aggregator can view the current price as the informant's combined belief of the outcome of the event.

In this model there is a set Ω of possible states of the world and at any moment the world is in exactly one state $\omega \in \Omega$, though the informants do not know which. Each informant i may however possess partial information regarding this true state, and this is represented by a partition π_i of Ω. The agent knows in which subset of this partition the true world state lies, but does not know the exact member of which is true. Given n agents, their combined information $\hat{\pi}$ is the coarsest common refinement of the partitions $\pi_1, \ldots, \pi_n$.[2]

We also assume a common prior probability distribution $P \in \Delta^{\Omega}$ which describes the probabilities that all agents assign to the different world states before receiving any information. Once each agent receives their partial information, they form their posterior beliefs by restricting the common prior to the subset of their partition in which they know the true state to lie. For our purposes, a *forecast* is an estimate of the expected value of the function $f : \Omega \to \{0, 1\}$, known as an *event*, which equals one for exactly one subset of Ω and 0 otherwise.

As mentioned, there is an ideal "omniscient" forecast that uses the distribution P restricted to the subset of $\hat{\pi}$ in which the true world state lies but which is impractical given the private nature of each agent's information. The goal is therefore a mechanism to incentivise agents to reveal their private information such that in equilibrium we achieve a forecast as close as possible to the omniscient one. Prediction markets offer the agents the chance of financial gain for revealing information regarding the expected value of $f(\omega)$, and the share price of the security can be interpreted as the collective forecast of the agents. In Section 4 we shall outline the different approaches one can take to designing the prediction market mechanism itself and the one among them, a Market Scoring Rule, that we implement. Importantly, Market Scoring Rules are one of the ways to increase liquidity in the market as the system itself assumes the opposite side to any trade, meaning users may participate even when no other user wishes to buy or sell for what they are asking or bidding. This means a trade is always able to be executed, although without care the system could make consistent losses.

2.2 Literature Review

The Iowa Electronic Markets (IEM) are real-money prediction markets developed by the University of Iowa [3] that have been running since 1988. They allow users to buy and sell contracts based on the outcome of U.S. political elections and economic indicators, and are currently offering markets for the winning party of the 2020 U.S. presidential election, the vote share between the Democratic and Republican parties in the 2020 U.S. presidential election, and the compositions of the houses of Congress, House of Representatives, and U.S. Senate after the outcome of the 2020 U.S. congressional elections. The number of markets offered is small and the topics are kept relevant to current events, meaning there is likely to be high liquidity for any security a user wishes to trade in. This has allowed the markets to predict the results of political elections with more accuracy and less error than traditional polls: for the presidential elections between 1988 and 2000, three-quarters of the time the IEM's market price on the day each poll was released was more accurate for predicting vote share than the poll itself [20, pg. 19]. These markets inspired similar markets in the forms of the Hollywood Stock Exchange, NewsFutures, and the Foresight Exchange report, which achieved similar successes despite not using real money.

An issue with these markets is that they are restrictive in the bets they offer. Although this can be beneficial in that they provide a focused and liquid market in which to trade, it leaves them potentially less interesting to interact with. A combinatorial prediction market is a solution to this and drastically increases the number of events that can be bet on and outcomes predicted by offering securities on interrelated propositions that can be combined in various ways. One example of such a market is that of *Predictalot* [6], a combinatorial prediction market developed by Yahoo! that allowed users to trade securities in the 2010 NCAA Men's Division I Basketball Tournament. The tournament sees the top 64 teams play 63 games in a knockout competition, yielding a total outcome space of size 2^{63}. *Predictalot* then kept track of the odds, computing them by scanning through all of the predictions made by users. This prediction market was the original inspiration for this project in investigating prediction markets. Using a Market Scoring Rule for such a market would involve computing a summation over the entire outcome space Ω, an intractable, #P-hard problem akin to counting the number of variable

[2] A partition α of a set X is a refinement of a partition ρ of X if every element of α is a subset of some element of ρ. In this case α is *finer* than ρ and ρ is *coarser* than α.

assignments that satisfy a CNF formula, or or the number of subsets in a list of integers that sum to zero. Instead, they employ an implementation of importance sampling, a technique for estimating properties of a particular probability distribution using only samples generated from a different distribution. This "naive" approach is then improved upon by the work of Dudík, Lahaie, and Pennock [9], who use convex optimisation and constraint generation to develop a market maker which is tractable. This approach lies somewhere between treating all securities as independent and a fully combinatorial, "ideal" market maker, propagating information among related securities for more complete information aggregation. The ways in which their odds are calculated are also natural: for example, a large bet on a team to win the entire tournament automatically increases the odds that the same team will progress past the first round, since they would not be able to win the competition without doing so. This work is then improved upon by Kroer, Dudík, Lahaie, and Balakrishnan [16], in which they use integer programming to achieve arbitrage-free trades, or always profitable risk-free trades. On top of achieving bounded loss, a crucial element behind a market mechanism operating in the real world and avoiding bankruptcy, avoiding arbitrage is desirable as it leads to more accurate forecasts: since users cannot make risk-free profits, they are forced to bet according to their true beliefs.

All examples so far have involved a centralised market mechanism. These types of systems involve a central authority providing the bets upon which users may bet and then verifying their outcome. *Decentralised* markets allow the users themselves to define their own bets and trade shares in them. These types of systems involve a central authority providing the bets upon which users may bet and then verifying their outcome. These types of markets allow the users themselves to define their own markets by providing custom bets and them trading shares in them. Several examples of decentralised markets exist and they are often implemented with cryptocurrencies. Peterson et al. [19] study the setting and use it to implement the oracle at the heart of *Augur* [1], a decentralised prediction market built upon the Ethereum blockchain that launched in 2018. It allows users to offer predictions on any topic, and markets may be either categorical, which are similar to binary markets in which the winner takes all, or scalar, which offer users a spectrum of outcomes in which to invest. For example, users may bet that the global average temperature for 2020 will lie in a certain range. As in many decentralised markets, outcomes of events are then resolved by the users, and in the case of *Augur* users are incentivised to report truthfully by way of paying reporting fees. This amounts to users depositing tokens to back their report, and token holders are then entitled to the trading fees generated. Although this persuades against manipulation, it has not been shown whether this system achieves any theoretical guarantees of truthful reporting.

As can often be the case with real-money markets, the platform had quickly devolved into an assassination market [8] – originally this referred to the case where users created markets on the deaths of certain people, which then incentivised their assassination. A user could stand to profit by placing a bet on the exact time of their death, and ensure this bet was profitable by assassinating the subject. More generally this refers to the users of a prediction market having the ability to influence a market's outcome and acting on this opportunity. Another issue with *Augur* is the option to report a market's outcome as "invalid": this is for the case where the user-made bet is too ambiguous to decided, such as, "Bayern Munich will play well against Paris Saint Germain".

Other decentralised markets based on cryptocurrencies exist, including *Omen* [5] and *Hivemind* [2]. The former is similar to *Augur* in that it allows users to create markets for any bet they like and whose outcomes are not decided by the system itself. Whereas *Augur* uses a reputation system in the form of requiring users to back their report of a market's outcome with $REP tokens, *Omen* asks the market creator to supply an "oracle" through which the outcome can be determined. They note that this oracle can even be *Augur*. Although this may solve the "invalid" outcome option for ambiguous bets, it may introduce bias into the process of outcome determination. For example, suppose a user creates a market for "The Democratic nominee will tell a lie during tonight's debate" and lists the oracle as the conservative news channel, Fox News. Users would then trade on how they think the oracle will report the outcome, and not what they believe the outcome will be themselves. An important aspect of decentralised markets must therefore be that the outcome is determined by the community, not a single source.

In contrast to *Augur*, which implements a traditional orderbook using Ethereum, *Omen* uses an automated market maker to provide liquidity to its securities. Two approaches to this include using a Market Scoring Rule to update the odds on a given event, and implementing a parimutuel market where users compete for a share of the total money wagered while the share price varies dynamically according to some cost function. Hanson [12] shows that we can use any strictly proper scoring rule to implement an automated market maker: with such scoring rules, agents maximise their expected utility by truthfully revealing their predictions. In particular, in this project we implement the peer prediction market introduced by Freeman, Lahaie, and Pennock [11], which specifies a market that uses a Market Scoring Rule to trade bets and crowdsources outcome determination, similarly to *Augur*, by asking users for reports. All they require is that the rule is strictly proper, giving plenty of choice to study the effects different rules have on user behaviour. While the choice of scoring rule is less important than the mechanism by which market outcomes are determined, a recent work by Liu, Wang, and Chen [17] introduces scoring rules for the setting where the aggregator has access only to user reports, which they call Surrogate Scoring Rules (SSRs). This appears to be an interesting avenue to further explore and adapt to a prediction market. One assumption they make, however, seems incompatible with the decentralised setting in that they require all events to be independent. Given that users can create a market for *any* bet, this condition is impossible to ensure. Since SSRs can be strictly proper under certain conditions, they may be applicable as the Market Scoring Rule in [11].

Other prediction markets existed in *PredictIt* [7] and *InTrade* [4], both offering markets for various political and economic events. However, both experienced disputes largely related to the wording of the available to trade and the ambiguity in their resolution. For example, a bet offered on *PredictIt* was, “Who will be Senate-confirmed Secretary of State on March 31, 2018?”, and although Rex Tillerson was fired in the middle of March, he was officially the secretary of state until midnight of the 31st, leading to confusion among users in what they are trading on and therefore inaccuracies in the predictions it elicited. This will be a problem inherent to any prediction market that allows users to specify their own bets, and while the work of Freeman et al. [11] evades the problem of determining the outcome by setting it to the proportion of users reporting a “yes” outcome, this does nothing to penalise the initial creator for introducing an ambiguous market.

3 Goals

3.1 Core Features

The goal of this project is to implement a truthful decentralised prediction market in which users may specify the bets on which to trade. The market outcomes will be decided by peer prediction, in which events are settled by a subset of the users known as arbiters. A user may even act as an arbiter in a market in which they themselves may hold a stake. Specifically, we seek to:

- create a web application on which users can create custom bets and trade on these markets
- implement a trading mechanism that allows users to buy and sell shares in the user-made securities using play money
- crowdsource outcome determination using reports from arbiters who may hold positions in the market
- incentivise truthful behaviour at all stages in the mechanism

These goals will largely be achieved by implementing the mechanism outlined by Freeman et al. [11], albeit wit several practical modifications. The first three goals cover core functionality of any decentralised prediction market, while the fourth is concerned with the tuning of system parameters, before and during execution, in order to ensure that users do not manipulate the mechanism. In this users are not only discouraged from attempting to “game” the system for personal gain, they are hurt for doing so.

Other non-essential features but highly desirable for strong user experience include asynchronous communication with the server in order to display up-to-date pricing information to the user without a page refresh, and the automated closing of markets. Both of these features would make the system straightforward and intuitive to use. Moreover, it would allow the system to run independently, meaning the market’s functioning is only influenced by the community, one of the key points of implementing a decentralised market.

As we will discuss in more detail in Section 4, one aspect of the mechanism is the assumption that the system knows the signal error rates when user’s receive news about a market’s outcome. This is unrealistic in practice, since we cannot hope to know the exact where each user learns about the outcome of the event nor the accuracy of said source’s reporting. Hence we also look to implement a way in which users do not need to be explicitly asked their estimates of signal accuracy, and instead this is calculated based on their past reporting history. This leaves less opportunity to game the system, the entire point of implementing this prediction market mechanism.

3.2 Stretch Features

With more time, there are plenty of additional features that could be implemented to render the system more intuitive and usable. These include the option to create different types of markets, particularly categorical ones since they would function similarly to binary markets but allow multiple related markets to be expressed more succinctly. Furthermore the option to create and sort markets by categories would help users offer their information more readily if they are especially interested in a certain topic, say politics or sport.

A useful feature to implement would be the tracking of price histories for each security. This would enable graphs to be generated so that users could be more informed on how the forecast of an event has changed over time. This would make the decision to participate at a particular price point more interesting on their part since they are not only estimating their belief on the probability that the event will have a positive outcome, but also on how the other users will act, similar to a real stock market.

Finally, an issue with the mechanism of Freeman et al. as it stands is that it does not directly punish users for creating markets on ambiguous bets. Traders may become confused about the wording, leading to different interpretations and hence different users trading on different beliefs – this is not useful for information aggregation. Although the negative effects are somewhat mitigated in that the very same community that trades in the security also decides on its outcome, there is no mechanism in place to specifically encourage clear bets. This is something that could be improved and would be useful in avoiding the market becoming swamped with overly subjective wagers.

3.3 Motivation

As discussed, there are already numerous prediction markets that exist in the literature. The Iowa Electronic Markets are the longest running and arguably most successful, but the options offered to the users on which to trade are too restrictive. This is a similar issue among all centralised markets, including InTrade, PredictIt, and the Hollywood Stock Exchange. While their success can be attributed to their narrow focus on a particular topic, it seems more interesting to be able to aggregate information from a wider variety of themes, sacrificing perhaps some of the predictive accuracy for more widespread forecasts to be made.

Decentralised prediction markets are not a new concept, however current examples in the literature lack in the functionality they offer. In the case of *Omen*, while they allow any user to create a market, they rely on a single oracle to determine the outcome of the event. This leaves a single point of "failure" in the system and leaves it open to being manipulated. For example, listing a biased news source as the oracle could have a significant effect on the event's outcome. Although this is mitigated somewhat by displaying to traders the oracle chosen, this still encourages them to trade on how they believe the oracle will report the market and not necessarily the market itself. The mechanism by which *Augur* determines market outcomes appears to be an improvement over this, in which multiple reporters from the community back their report of the market outcome with $REP tokens, thus implementing a form of reputation system. However, it does not deal with ambiguous bets elegantly, offering the option for a report a market's outcome as "invalid". Given the inevitability of such markets in a decentralised setting, this is a key weakness to *Augur*.

It is therefore well justified to implement the decentralised peer prediction mechanism of Freeman et al. This not only allows users to create markets for any event they see fit, but also crowdsources market outcomes by relying on reports from the community. Thus instead of relying on a single source of information, which leaves it vulnerable to reporting biases particularly for more ambiguous or subjective bets, it gets a more complete picture of how the users themselves, who are the ones interacting with the market, observed the outcome. This can average out the biases present in any one news source. This also seems to be a better method to deal with ambiguity than in *Augur*, since the outcome of the market can be influenced by a reporter's interpretation of a wager. Another issue with *Augur* is that it has not been shown to achieve any theoretical guarantees, which should be a key consideration in an environment in which rational selfish agents are interacting. Instead, the market we implement is proven to be incentive compatible, meaning it it in a user's best interests to report market outcomes truthfully and not attempt to manipulate it for their own gain. Although the mechanism is not budget balanced, it can be fully subsidised by a trading fee on each transaction, further making it practical and self-sufficient.

4 Design

4.1 Mechanism Overview

As mentioned our design of the prediction market is based on the peer prediction mechanism proposed by Freeman et al. [11]. In this section we will present the main ideas presented in their work and give an overview of the mechanism itself.

We are interested in setting up a prediction market for outcome of a binary random variable $X \in \{0, 1\}$. We will use the terms "market", "stock", and "security" interchangeably throughout to refer to the entity comprising a wager, such as "Arsenal will beat Tottenham", and a deadline – these two pieces of information are all we need to represent the event X. The mechanism is divided into two main stages: the market stage, where users may buy and sell shares in the securities whose deadlines have not yet passed; and the arbitration stage, where a subset of the users report on the outcome of the security and the payout price per share is computed. In a traditional prediction market on binary events, if the market's outcome was positive then stakeholders with long positions will then be paid out $1 for each share they own while those with short positions will buy back their shares at a price of $1 per share. Similarly, if the outcome was negative, long users will have lost money since they receive no money back from their initial investment, while short users will profit as they must "buy back" their shares at a price of $0 per share. This market is different in that market outcomes are set as the proportion of users that reported a positive outcome. Therefore, even if a user has gone long on a security, this must have been at or below the right price to make a profit, since it is not necessarily the case that 100% of the users reporting on the market agree on its outcome.

Since we rely heavily on user participation for the mechanism to run correctly, it is important that users act in the desired manner. This mechanism incentivises users to act truthfully in two aspects: firstly, users are encouraged to trade on their belief of the market's realised outcome, rather than, for example, how a specific news source will report on it, since this outcome is determined entirely by the community, each individual of which will have access to their own news sources of varying biases; secondly, it is in a user's best interests to report on market outcomes truthfully, since they can receive no better payoff by attempting to manipulate the system. The same is true even if an otherwise rogue reporter held a position in the market. Therefore, we are able to gather accurate public sentiment on the event itself as well as its outcome, and ambiguous securities are dealt with more gracefully.

These considerations also allow us to achieve certain useful guarantees. For example, in order to incentivise reporters to act truthfully, we must pay them more than what they would otherwise gain from attempting to manipulate the system. We can use this knowledge, coupled with what we know stakeholders are expecting to be paid out, in order to bound the amount we must pay to ensure incentive compatibility. In this way we can ensure that the system is sustainable and the system's loss is never unbounded.

In the following sections, we shall outline the mechanism we implement from a theoretical standpoint.

4.2 Market Stage

The market stage allows users to create markets for any bet they desire and specifies the how the share price reacts according to user participation. As we implement a decentralised market, we place no restriction on the bet that can be placed (only that it is binary, i.e. it is ultimately either a "yes" or a "no"). In any market, there are two sides to any trade: one side offers some number shares, while there is some party that must be willing to purchase this number of shares at this price. There are a number of options for implementing such a trading mechanism, as we will detail in the following section.

4.2.1 Market Scoring Rules

There are a number of options for implementing a market trading mechanism: these include as a continuous double auction, market call auction, or by using an automated market maker. A continuous double auction (CDA) is a method by which buyers are matched with sellers of a stock. The market maker keeps an order book that tracks the bids, submitted by those looking to buy a stock, and the asks, submitted by those looking to sell a stock. Traders arrive asynchronously and place orders, and when two opposite orders match the trade is executed. Continuous double auctions are traditionally used in highly liquid markets, where there are many bids for a given ask (and vice versa), such as the New York Stock Exchange. An issue with this approach, however, is that it relies on high liquidity – that is, it requires that there is always a willing buyer and seller for a particular price at a particular quantity of shares. Especially in a prediction market, most prediction markets have far fewer participants than a stock exchange: if Alice is willing to sell a share of stock A for \$6 but no one is willing to buy at that price, then a trade cannot be executed. This problem can be particularly severe in a combinatorial market, where a trader's attention is split between exponentially many securities. Traders may also be hesitant to participate in a market organised as a continuous double auction, since their participation at a particular price encourages others to participate at a lower price thus depriving them of profit. Prices may therefore not be informative, and instead only reflect the game traders are playing with one another.

To avoid these problems of illiquidity, one can use an automated market maker, in which a price maker is (nearly) always willing to accept both buy and sell orders at a certain price. It is possible that this price changes with the user's interaction with the market. This ensures that participants are always able to make a trade, thus "making" the market. Typically this is not used in real-money markets since always assuming the opposite side to any given trade would likely result in losses for the house; in play-money markets this is not such an issue as losses are less detrimental, but prices must still be adjusted to ensure losses are bounded. Two options arise for implementing an automated market maker: Market Scoring Rules (MSR) and parimutuel markets. We are interested in the former.

Suppose our market contains $|\Omega|$ mutually exclusive and exhaustive securities. Let q_j be the total quantity of shares held by all traders of security j, and let $\boldsymbol{q} = (q_1, \ldots, q_{|\Omega|})$. At the heart of a Market Scoring Rule is the cost function $C(\boldsymbol{q})$, which is a means of recording the total amount of money spent by traders as a function of the total number of shares in circulation. An agent wishing to purchase $q'_j - q_j$ shares of security j, increasing the number of shares of security j to q'_j, would then pay $C(\boldsymbol{q}_{-j}, q'_j) - C(\boldsymbol{q})$.[3] This model also accommodates selling shares, in which case $q'_j < q_j$. We cannot use C to quote a share price to the user, since we first need to know the quantity of shares they wish to buy or sell: instead, we use its derivative $p = \partial C / \partial q_j$, which gives the cost for purchasing an infinitesimal quantity of shares. Just as we cannot use C to quote individual share price, we cannot use p to calculate the cost of transaction, since a user's interaction with the market will automatically have an effect on share price as, in practice, they are not purchasing an infinitesimal quantity.

4.2.2 Trading fees

In addition to implementing a Market Scoring Rule to quote the share price of a given security and calculate the necessary payments for buying or selling shares in a given market, the market stage is responsible for implementing trading fees. As mentioned, the mechanism we implement is not budget balanced, meaning the market must be subsidised in order to pay the correct winnings to stakeholders when market outcomes are realised. Trading fees are therefore used to raise these subsidies. For any given market, buying shares will push the share price p upwards towards \$1, while selling shares will push it towards \$0. There are two types of transactions that a user may be involved in: one in which a trader is increasing their risk, and one in which a user is liquidating shares it has previously bought or sold. For example, suppose Alice holds ten shares in a particular security: if she were to sell up to and including ten shares she would simply be liquidating shares that have already been sold to her, while if she were to buy additional shares or sell more than ten, then she would be increasing her risk. Risk transactions are defined analogously for a user buying shares. Trading fees are only imposed on transactions in which a user increases their risk, and can be viewed as a fee on the worst-case loss incurred by the agent. Specifically, for fixed system parameter f and for transactions in which a user increases their risk, a buy transaction that pushes share price to p incurs an additional charge of fp, while a sell transaction that pushes share price to p incurs additional charge of $f(1-p)$. Users may trade shares in a given security as long as they have enough funds to make the transaction (including the fee), and as long as the deadline has not yet passed. After the market expires, stakeholders' positions are final and we then determine the outcome of the market via peer prediction in the arbitration stage.

4.3 Arbitration Stage

The arbitration stage is concerned with determining the perceived outcome of the event X from a subset of the total userbase, known as the "arbiters", who offer reports on what they observed the outcome to be. Specifically, each arbiter i receives a private signal $x_i \in \{0, 1\}$ that tells them the result of the event – this is analogous to reading the news, watching the match, even hearing about it from a friend, and will vary from market to market. The arbiter then submits a report $\hat{x}_i \in \{0, 1\}$ to the system that tells it what they believe the outcome to be. Note that since the signal they receive is private information, we

[3] We use $\boldsymbol{q}_{-j}$ to denote the vector $(q_1, \ldots, q_{j-1}, q_{j+1}, \ldots, q_{|\Omega|})$.

have no way of determining whether this report is what they truly observed, or whether they are trying to manipulate the system for their own gain. Instead, we incentivise arbiters to act truthfully by paying them a reward if their report agrees with another randomly chosen arbiter: for this we implement the "1/prior with midpoint" mechanism, which we will detail below.

Once all reports have been collected and the arbiters paid, the outcome of the market $\hat{X} \in [0, 1]$ is set to the proportion of arbiters that reported a positive outcome. This differentiates this mechanism from traditional prediction markets, in which shares of a security will pay out $1 if the event occurred, and $0 otherwise. Stakeholders are then paid out in the usual, where those with long positions are paid out $\hat{X}$ for each share owned, while those with short positions must buy them back at $\hat{X}$ per share. This should not change how traders view the security: if they have information telling them the event will occur they will still buy into the market if the share price is appropriate, while if they believe the event is unlikely they will continue to sell. The mechanism simply accommodates for the ambiguous bets possible as a result of being made by the community.

4.3.1 1/prior mechanism

We use a modified version of the 1/prior payment mechanism to reward users for submitting reports on an event's outcome and to incentivise truth telling behaviour. The original version was conceived by Jurca and Faltings [14, 15] as a means of rewarding arbiters for participation in opinion polls, another means of crowdsourcing a forecast in which users submit probabilistic estimates for the likelihood of events to occur. Witkowski [21] then generalised this to pay out different amounts depending on the signals reported by paired arbiters. For arbiters i and j with reports $\hat{x}_i$ and $\hat{x}_j$, the 1/prior mechanism pays a reward $u(\hat{x}_i, \hat{x}_j)$ as follows:

$$u(\hat{x}_i, \hat{x}_j) = \begin{cases} k\mu & \text{if } \hat{x}_i = \hat{x}_j = 0 \\ k(1-\mu) & \text{if } \hat{x}_i = \hat{x}_j = 1 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

In this, k is a parameter and μ is the common prior belief that $X = 1$. A suitable value to use for μ in our case is the closing price of the market: if users feel the event is likely to occur they will buy shares of it, pushing the share price towards $1, and if they feel it is unlikely it will be pushed towards $0. Everyone can see this price, and if it differs from a user's beliefs they will participate in the market and alter the price accordingly, making it a sensible choice for the common prior.

The modification introduced by Freeman et al. [11] of the 1/prior mechanism is simple and requires two extra values to be computed. Let μ_1^i be the probability that, given that agent i receives a positive signal of the event's outcome, another randomly chosen user also receives a positive signal. Similarly, let μ_0^i be the probability that, given that agent i receives a negative signal, another randomly chosen user receives a positive signal. We require a common value for these "update" probabilities across all agents, so we define the μ_1 and μ_0 as follows:

$$\begin{aligned} \mu_1 &:= \min_i \mu_1^i \\ \mu_0 &:= \max_i \mu_0^i \end{aligned} \tag{2}$$

The modified payment mechanism is now simply Equation 1 with μ replaced by $(\mu_1 + \mu_0)/2$. This is the "1/prior-with-midpoint" mechanism and guarantees that the incentives for arbiters are always the same, no matter the signal they receive. The arbiter with the greatest incentive to misreport – that is, an arbiter with a large stake in the market in which they are reporting – has their incentive to misreport weakly decreased by using the midpoint $(\mu_1 + \mu_0)/2$ in the payment rule as opposed using μ in the standard version.

In particular, suppose arbiter i holds a position of n_i securities in the market. We can ensure truthful reporting is a best response for i by setting the 1/prior-with-midpoint parameter k to the appropriate value such that they will receive weakly greater reward from the payment mechanism than they would by misreporting. With m arbiters and $\delta = \mu_1 - \mu_0$, truthful reporting for arbiter i is a best response if:

$$k \geq \frac{2|n_i|}{m\delta} \tag{3}$$

4.4 Tools

5 Implementation

In this section we shall detail our implementation of the peer prediction mechanism. The system consists of five independent parts, each responsible for a separate area of functionality. These are as follows:

1. Database
2. Trading
3. Arbitration
4. Server
5. User experience

5.1 Database

5.1.1 Table Definitions

The database is responsible for persistent storage. It is implemented using the Mito library provided by Quicklisp, which is an object relational mapper that provides support for MySQL, PostgreSQL, and SQLite3. This allows us to define and interact with the database tables while remaining within the Lisp ecosystem, leading to a quicker and more flexible development cycle.

We define three tables that handle all interactions with the database: `USER`, which stores all users in the system and their remaining budget; `SECURITY`, which is stores the wager, deadline, number of shares, and final outcome for every user-created security; and `USER-SECURITY`, which is responsible for the many-to-many relationship that users may have with securities, and allows us to store user positions as well as reports once the deadline has passed. The table definitions are as follows:

User	*name*	*budget*

Security	*bet*	*shares*	*deadline*	*outcome*

User-security	*user*	*security*	*shares*	*report*	*positive belief*	*negative belief*

An entry in the `USER` table consists of a username and a budget. All users currently start with $100, which is of no real consequence since the market uses play-money. Currently there is no requirement to supply a password when logging; this has been done to speed up the testing process, and if required in the future this will be a simple addition to make.

Securities are also simple entities to store, whose fields fully describe a market: *bet* stores a string that specifies the wager being made; *shares* stores the total number of shares (referred to as q in Section 4.2.1); *deadline* holds the date and time by which the event's outcome will have been realised; and *outcome* is initially null and eventually set to the payout price per share, or the fraction of arbiters reporting a positive outcome, once the deadline has passed.

The `USER-SECURITY` table is used to represent the mapping between users and securities, and is used to store a user's position in a given market as well as the outcome they have reported on its outcome, if they are indeed an arbiter for it. The columns are as follows: *user* holds a reference to an entry in the `USER` table; *security* holds a reference to an entry in the `SECURITY` table; *shares* stores the user's position in this market if they have one, otherwise 0; *report* stores the user's report on the outcome, if they are an arbiter for this security; and *positive belief* and *negative belief* represent how reliable a user's signals are, based on their previous reporting history. The manner in which we use the last two fields will be discussed in greater depth in Section 5.3.

5.1.2 Database Interface

We provide the interface with which we interact with the database in `database.lisp`. We first initialise the database by connecting to our choice of backend and then define the tables as in the previous section. We opt to use MySQL as the backend simply as it was already installed on the development machine, as well as some prior familiarity with the language. Tables are then created using the `deftable` macro supplied by Mito: syntactically, this is similar to vanilla Common Lisp's `defstruct` macro. Listing 1 shows how we define the columns and their associated datatypes. The macro defines the default accessors[4], the slots `created_at` and `updated_at`, and a primary key `id` if none is specified. As the listing shows, we can specify a previously defined table as the column datatype, in this case `user` and `security`, in order to model the foreign key relation in a straightforward manner.

```
(deftable user-security ()
          ((user :col-type user)
           (security :col-type security)
           (shares :col-type :integer
                   :initform 0)
           (report :col-type (or :integer :null))
           (positive-belief :col-type (or :double :null))
           (negative-belief :col-type (or :double :null))))
```

Listing 1: Defining the `USER-SECURITY` table in Mito

Insertion is similarly straightforward: to insert a new entry into a table we simply create an instance of the structure that is implicitly defined when calling `deftable` then call `create-dao`. To retrieve records from the database we can use either `select-dao` or `find-dao`: the former returns all records satisfying the criteria provided, while the latter returns the first match.

We design the interface to the database so that no custom queries need to be created outside of `database.lisp`. This ensures the code interacting with it can be kept as clean and simple as possible. We use another library from Quicklisp, SXQL, in order to build the more complex queries to the database. For example, Listing 2 shows how we retrieve all the securities whose deadline is yet to pass, in order of first to last to expire. We use the SXQL functions `where`, `:>`, `order-by`, and `:asc` that expand into the corresponding MySQL code for Mito to execute.

```
(defun get-expiring-securities (datetime)
  " return all securities ordered by deadline , most imminent first "
  (with-open-database
    (select-dao ’security
                (where (:> :deadline datetime))
                (order-by (:asc :deadline)))))
```

Listing 2: Retrieving active markets using Mito and SXQL

For any interaction with the database, we need to establish a connection prior to the transaction and disconnect after it is complete. This gives us the opportunity to make use of Lisp's macro system: while it is a small example, since its use is so widespread it drastically reduces the number of lines and ensures we never forget to close a database connection. We define our macro `with-open-database` as in Listing 3. Since the final statement in a function or macro definition in Lisp is the value returned by that block, we are able to open the connection, execute arbitrary code and store the final result in the variable `result`, then disconnect from the database and return the result of the transaction.

```
(defmacro with-open-database (&body code)
  " execute CODE without worrying about the connection "
  ‘(progn
     (connect-database)
     (let ((result (progn ,@code)))
       (disconnect-database)
       result)))
```

Listing 3: Defining our `with-open-database` macro

In order to enable the transactions in the following sections, when we first initialise the database we create a user with the name "bank". All money that is then to be collected from or paid out to users is done so through the bank, so that money is largely conserved. Although not so important for a play-money market, it does help to enforce a value on the currency we use as well as analyse the system's performance.

[4] Functions for accessing members of a struct.

5.2 Trading

Trading allows us to gather public sentiment on the user-defined securities, and this part of the system is responsible for setting share prices, calculating the cost of transactions, and charging fees to raise funds for the arbitration stage, to ensure arbiters are incentivised to report truthfully. These features are implemented in the files `msr.lisp` and `market.lisp`.

As discussed in Section 4.2.1, we will be using a Market Scoring Rule as our automated market maker. In order to achieve the guarantees of Freeman, Lahaie, and Pennock's mechanism we must use a scoring rule that is strictly proper, which can then be implemented as a market maker based on a convex cost function. We use the commonly-used Logarithmic Market Scoring Rule (LMSR) created by Robin Hanson [13], whose cost function is defined as follows:

$$C(\boldsymbol{q}) = b \log \left(\sum_j e^{q_j/b} \right) \tag{4}$$

This assumes that each security j is one of a collection of mutually exclusive and exhaustive outcomes. Since we are only dealing with binary events, we can compute a share price based only on the number of shares bought for the positive outcome, and assume that buying shares in the negative outcome is equivalent to selling shares in the positive outcome. In this case, we have $\boldsymbol{q} = (0 \quad q_1 - q_0)$, where q_0 and q_1 are the quantity of shares bought by agents in the negative and positive outcomes, respectively. This gives us the following cost function for LMSR in the binary setting, where $q = q_1 - q_0$:

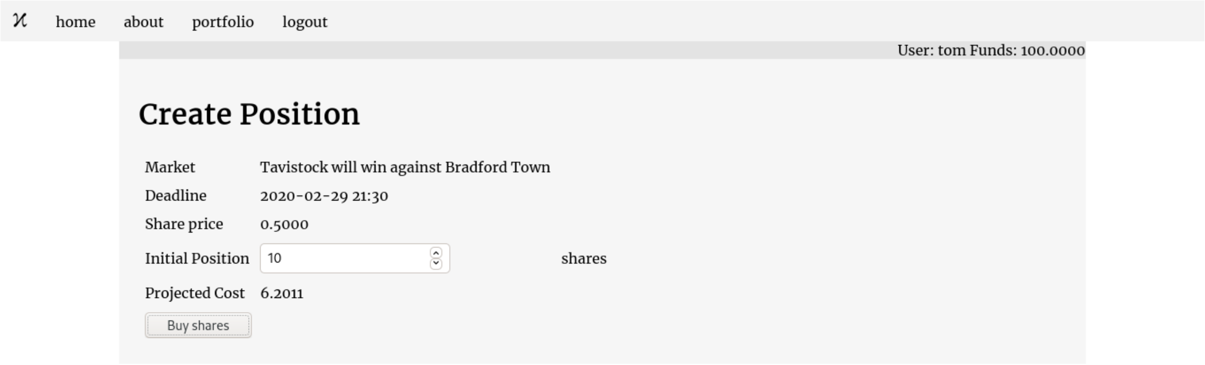
$$C_b(q) = b \log(1 + e^{q/b}) \tag{5}$$

In these cost functions, b appears as a parameter that allows us to control the responsiveness of C. A lower value of b corresponds to a more sensitive share price, meaning the price will change more quickly for smaller transactions. It also controls the market's risk of loss: for markets with $|\Omega|$ outcomes it can be shown that the maximum loss incurred by the market maker is $b \log |\Omega|$. Recall that to compute the actual cost to charge an agent for a transaction, we compute $C_b(q') - C_b(q)$ for an agent wishing to take the total quantity of shares from q to q', and this also encodes sell transactions. The share price function is the derivative of the cost function, which is in our case:

$$p_b(q) = \frac{e^{q/b}}{1 + e^{q/b}} \tag{6}$$

Upon market creation a user is only given the option to buy a positive number of shares – otherwise, it would make more sense for the user to create a market for the opposite outcome. After this, the custom security is created and is open for all other users to trade in. The share price reflects the strength of the community's opinion on the event's outcome: for example, at $q = 0$ then the community is exactly split on whether the event will have a positive outcome, and appropriately $p_b(0) = \$0.50$ for any b. Meanwhile, for $q = 20$ and $b = 10$ this means twenty more shares have been bought than sold in the market and would yield a quoted share price of $p_{10}(20) \approx \$0.88$. Users feel more confident that the event will be positive, thus pushing the price upwards. This could assure some agent that the event will have a positive outcome. Suppose they wish to buy ten more shares in the market: they would then be required to pay $C_{10}(30) - C_{10}(20) = 10 \log(\frac{1+e^3}{1+e^2}) \approx \$9.22$. Such an action in the market would push the share price to $p(30) \approx \$0.95$. Users may also be required to pay a fee on their transaction, in order to subsidise the arbitration stage. Figure 1 shows the interface for creating a new market, while Figure 2 shows an example of the transaction summary that a user is presented with after having done so.

Fees are only charged on transactions where an agent increases their risk: this means they are either buying or selling a greater number of shares than their current position. For example, a user owning ten shares and selling five would incur no extra cost since they are liquidating a position they have already invested in, while selling more than ten or buying additional shares would incur an additional cost. The fee serves a secondary purpose in bounding the price of the security away from \$0 and \$1, as a potential trader would be spending more than \$1 or less than \$0 to buy or sell the shares. For example, even if an agent were to buy an infinitesimal number of shares, meaning their transaction cost would be given by the share price function p, if the share price was \$0.99 and the transaction fee was set to 5%, they would be required to pay $\$0.99 \cdot 1.05 = 1.0395$, greater than the maximum possible payout.

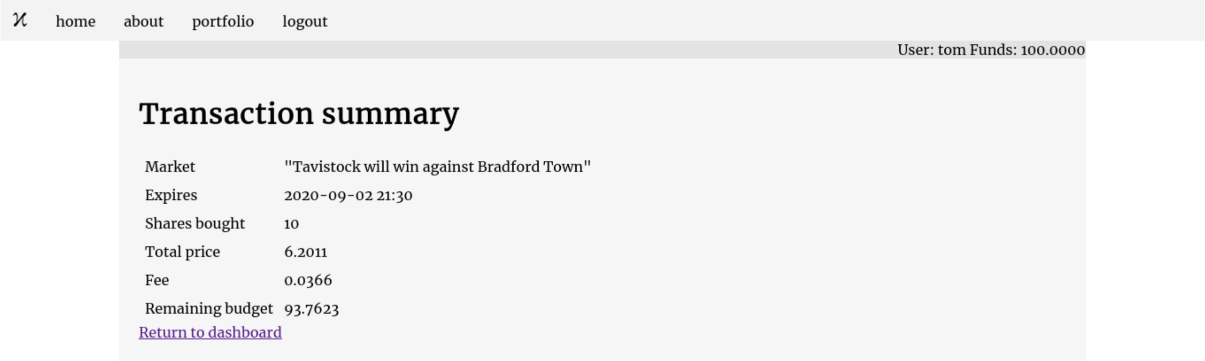


User: tom Funds: 100.0000

Create Position

Market	Tavistock will win against Bradford Town	
Deadline	2020-02-29 21:30	
Share price	0.5000	
Initial Position	10	shares
Projected Cost	6.2011	

Figure 1: The interface for creating a new market



User: tom Funds: 100.0000

Transaction summary

Market	"Tavistock will win against Bradford Town"
Expires	2020-09-02 21:30
Shares bought	10
Total price	6.2011
Fee	0.0366
Remaining budget	93.7623

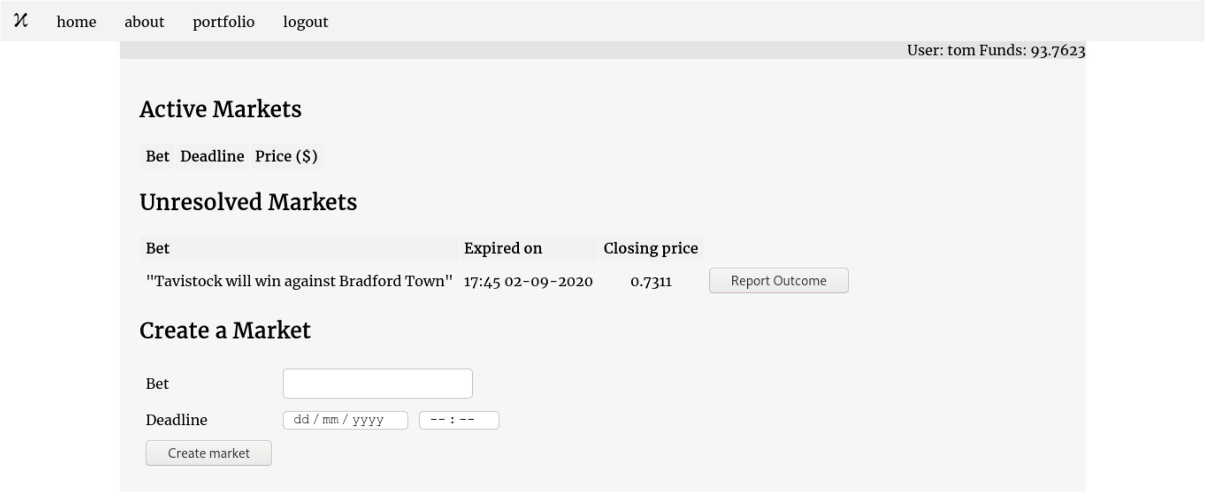
Figure 2: Users are presented with a transaction summary upon creating a new market

5.3 Arbitration

5.3.1 Computing signal reliability

The arbitration stage is where we resolve market outcomes using arbiter reports and pay out winnings to, or demand payment from, stakeholders in the security. Since the mechanism incentivises arbiters to act truthfully even if an arbiter themselves holds a stake in the market, we decide to let anyone opt in to reporting on the outcome. Once a market's deadline has passed the security will be listed as an expired market and the user is presented with the option to act as an arbiter, as in Figure 3. After doing so, they may then input their observed signal, or indeed a lie, as Figure 4 shows. Once the required number of reports have been collected, we move onto rewarding arbiters for submitting reports and computing the final outcome of the market.

Arbiters are rewarded by being paid a certain amount of money only if their report agrees with another randomly chosen arbiter. The reward is determined by the 1/prior-with-midpoint mechanism, where instead of using the common prior probability μ we use the update probabilities μ_1 and μ_0. Recall that these are the probability that, given that arbiter i receives a positive signal so too does another randomly chosen arbiter j, and the probability that, given that i receives a negative signal, another randomly chosen arbiter j receives a positive signal. Since we cannot assume arbiters to act truthfully, this is more complicated than simply counting the types of reports received. Instead, we use information about signal error rates $Pr[x_i = 1|X = 1]$ and $Pr[x_i = 1|X = 0]$ based on past market outcomes and past reporting behaviour for each arbiter i. This is a fairly involved process: first we gather all securities that have been reported on in the past by i and whose outcome has been determined; we then iterate through each one of these returned securities and determine the report that i submitted as well as its outcome, and push this into a list of results. At the end of this stage we have a list `((security report outcome) ...)` that describes the arbiter's report and the market's true (peer-determined) outcome for



User: tom Funds: 93.7623

Active Markets

Bet	Deadline	Price ($)

Unresolved Markets

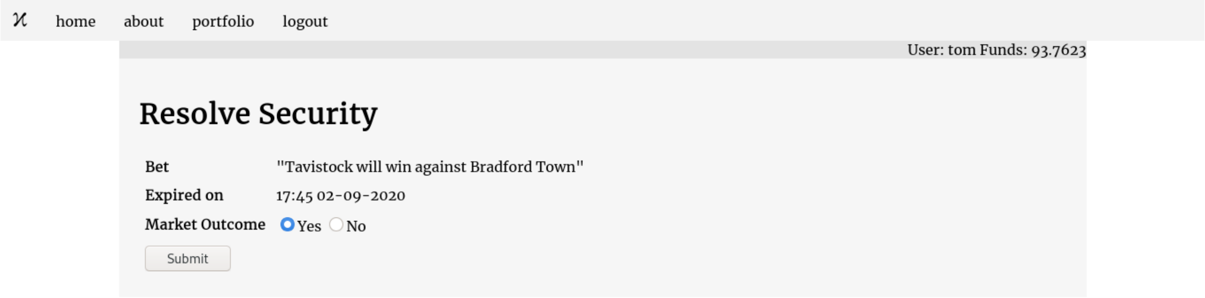
Bet	Expired on	Closing price
"Tavistock will win against Bradford Town"	17:45 02-09-2020	0.7311

Create a Market

Bet

Deadline dd / mm / yyyy -- : --

Figure 3: Users may opt in to report on market outcomes



User: tom Funds: 93.7623

Resolve Security

Bet	"Tavistock will win against Bradford Town"
Expired on	17:45 02-09-2020
Market Outcome	● Yes ○ No

Figure 4: The interface by which arbiters report market outcomes

each security for which they have acted as an arbiter. The code implementing this is show in Listing 4.

```
(defun get-securities-reported-by-user (user)
  (with-open-database
    (select-dao 'security (inner-join 'user-security
                                      :on (:= :security.id
                                              :user-security.security-id))
                 (where (:and (:= :user-security.user-id (user-id user))
                              (:not-null :security.outcome)
                              (:not-null :user-security.report))))))

(defun get-reporting-history (user)
  (let ((securities (get-securities-reported-by-user user))
        security-reports)
        (dolist (security securities)
          (with-open-database
            (let ((report (user-security-report
                            (find-dao 'user-security
                                      :user user
                                      :security security)))
              (outcome (security-outcome security)))
              (if outcome
            (push (list security report outcome) security-reports)))))
        security-reports))
```

Listing 4: Gathering a user's reporting history

We then use this history to compute the probability that, assuming the arbiter has always acted truthfully, their signal has been correct in telling them the true outcome of the event. We refer to these probabilities, $Pr[x_i = 1|X = 1]$ and $Pr[x_i = 1|X = 0]$, as a arbiter i's positive and negative signal beliefs as a remnant of a previous implement. Listing 5 shows how we calculate an arbiter's positive belief, with a similar method applying for computing the negative belief: for the arbiter's history restricted to the securities whose outcome was reported as positive by the majority of arbiters, we count the number of times this arbiter also reported a positive outcome, thus giving us the arbiter's signal error probability given we know the outcome is (more likely to have been) positive. If there have been no positive outcomes, then we assume the arbiter has a perfect signal. Calculating the negative signal belief is done in a similar manner: first we collect items of the arbiter's reporting history where the security was mostly reported as a negative outcome, then we count the number of times this arbiter submitted a positive report. Again we assume in the lack of markets with negative outcomes that the arbiter has a perfect signal. We repeat this to compute the positive and negative signal beliefs for each arbiter.

```
(defun calculate-positive-belief (reporting-history)
  ;; first only get the reports where the outcome was positive
  (let ((positive-outcomes (remove-if-not #'(lambda (x) (>= 0.5 (third x)))
                                          reporting-history))
        positive-reports)

    ;; now count the number of times the user reported a positive outcome
    (setf positive-reports (count T (mapcar #'(lambda (x) (equal 1 (second x)))
                                            positive-outcomes)))
    (if positive-outcomes
      (/ positive-reports (length positive-outcomes))
      1)))
```

Listing 5: Computing an arbiter's positive signal belief given their reporting history

We use this information about the reliability of each arbiter's signal, along with the prior probability μ that the event had a positive outcome, to compute the update probabilities μ_1 and μ_0, used in the 1/prior-with-midpoint mechanism. We use these to compute the update probabilities μ_1^i for each arbiter i, using a randomly chosen peer arbiter j, as follows:

$$\begin{aligned} \mu_1^i &= Pr[x_j = 1|x_i = 1] \\ &= Pr[x_j = 1|X = 0] \cdot Pr[X = 0|x_i = 1] + Pr[x_j = 1|X = 1] \cdot Pr[X = 1|x_i = 1] \end{aligned} \tag{7}$$

We use the same approach to compute μ_0^i for each i. The final values of μ_1 and μ_0 are then calculated by taking the minimum and maximum across all μ_1^i and μ_0^i, respectively.

5.3.2 Rewarding the arbiters

We may now pair arbiters randomly to pay them via the 1/prior-with-midpoint mechanism. We first retrieve two lists from the database: the first is the list of all arbiter reports and is of the form `arbiter-reports = ((arbiter report) ...)`; the second is a list of all arbiter signal beliefs and is of the form `arbiter-beliefs = ((arbiter positive-belief negative-belief) ...)`. Since we will be pairing arbiters randomly and still need efficient access to these values, we then create two hash tables associating an arbiter to their report and their beliefs, giving us `report[i] = report` and `beliefs[i] = (positive-belief negative-belief)` for each i. The market's outcome is simply set to the proportion of arbiters that reported a positive outcome. At this point we pay out the appropriate winnings to stakeholders, paying out money to those who hold shares and demanding money from those who have gone short. The former group are looking to be paid out at a price higher than the level at which they bought the shares, while the latter are hoping to buy back their shares at a lower level than their "in" price. Now that the market's outcome has been determined it will no longer be listed as an unresolved market on the user's dashboard.

```
(let ((reports (mapcar #'second arbiter-reports)))
  ;; the payoff of each share held is the fraction of arbiters reporting 1
  (setf outcome (float (/ (count 1 reports) (length reports)))))
```

Listing 6: Computing the market outcome

We next compute the random pairing of arbiters. Lisp has no function to shuffle a list, so we implement the Fisher-Yates [10, pg. 26-27] algorithm ourselves as the `shuffle` function in Listing 7. The random pairing is then formed by walking through the shuffled list and collecting adjacent elements as pairs.

```
(defun shuffle (lst)
  " shuffle LST randomly (without modifying it) "
  (let ((lst (copy-list lst)))
    (loop for i from (length lst) downto 2 do
          (rotatef (elt lst (random i))
                   (elt lst (1- i))))
    lst))

(defun random-pairing (lst)
  " randomly pair items from LST together if length(lst) is even "
  (if (evenp (length lst))
    (loop for (a b) on (shuffle lst) by #'cddr while b
          collect (list a b))))
```

Listing 7: Assigning arbiters to peers randomly

For each set of paired arbiters we then retrieve their reports and signal beliefs from the hash tables and compute, for each arbiter i in the pair, the values of μ_1^i and μ_0^i according to Equation 7, where the randomly chosen j is simply the partner with whom they have been paired. We push these values to a list so we may then compute the minimum value of μ_1^i and maximum value of μ_0^i across all arbiters. Finally to compute the smallest k to satisfy Equation 3 we set $k = \max_i 2|n_i|/m\delta$, and use this to pay arbiters according to Equation 1.

5.4 Server

The code from each of the separate areas of the market is then drawn together in the file `server.lisp`, in which we set up the web server, define the webpages, and call the functions from the different interfaces we provide.

We use Hunchentoot to host the web server. At startup this involves creating an instance of a Hunchentoot `easy-acceptor`, which opens up a port of our choosing to accept requests. Doing so also initialises the dispatch table, which is a list containing the functions to execute when the corresponding webpage is loaded. One way to specify webpages to the dispatch table is by using Hunchentoot's `create-prefix-dispatcher` function: this simply takes a URL and the function to execute when that URL is loaded. Here is again the opportunity to make use of Lisp's macro system: the code in Listing 9 shows a macro that defines a function and a URL of the same name, creates a dispatcher from them and pushes it to the dispatch table. This again not only saves rewriting repetitive code but also keeps the codebase clear and logical – the URL `/index` is served by a function called `index`. Now each time we wish to define a new webpage we need only to call `define-url-fn` followed by the name of the page and the instructions to execute.

```
(defmacro define-url-fn ((name) &body body)
  " creates handler NAME and pushes it to *DISPATCH-TABLE* "
  `(progn
     ;; define the handler
     (defun ,name ()
       ,@body)

     ;; add the handler to the dispatch table
     (push (create-prefix-dispatcher
             ,(format NIL "/~(~A~)" name) ',name)
           *dispatch-table*)))
```

Listing 8: Macroising URL functions

To define the content of the webpages we use CL-WHO[5], translates Lisp statements into strings of valid HTML. Since we are aiming for a uniform style across all webpages – for example, with a consistent

[5] Common Lisp With HTML Output.

navigation bar, a content section, and all the same preamble – this is another opportunity to make use of Lisp's macro system, to avoid code duplication and improve the abstraction to the code. In this vein we define the `standard-page` macro, an abridged version of which is given in Listing **??**. This allows us to concisely define webpages with a similar look and feel in a concise manner, and only requires us to specify the content that makes the page unique via the `body` parameter.

```
(defmacro standard-page ((&key title) &body body)
  " template for a standard webpage "
  ‘(with-html-output-to-string
     (*standard-output* NIL :prologue T :indent T)
     (:html :xml\:lang "en"
            :lang "en
            (:head (:title title)
                   (:link :href "/style.css" :type "text/css" :rel "stylesheet")
                   ;; rest of preamble ... )
            (:body
              (:ul :id "navbar"
                   (:li ... ))
              (:div :class "container"
                    ...
                    (:div :class "content"
                          ,@body))))))
```

Listing 9: Macroising webpage definitions

5.5 User Experience

References

[1] *Augur: the world's most accessible, no-limit betting platform.* `https://www.augur.net`. Accessed: 2020-02-16.

[2] *Hivemind.* `https://hvmd.io/`. Accessed: 2020-04-16.

[3] *IEM: Iowa Electronic Markets.* `https://iemweb.biz.uiowa.edu`. Accessed: 2020-02-16.

[4] *InTrade.* `https://intrade.com/`. Accessed: 2020-08-28.

[5] *Omen.* `https://omen.eth.link/`. Accessed: 2020-08-28.

[6] *Predictalot! (and we mean a lot).* `http://blog.oddhead.com/2010/03/05/predictalot/`. Accessed: 2020-02-16.

[7] *Predictit.* `https://www.predictit.org`. Accessed: 2020-02-16.

[8] *This new blockchain-based betting platform could cause napster-size legal headaches.* `https://www.technologyreview.com/2018/08/02/240354/this-new-ethereum-based-assassination-market-platform-could-cause-napster-size-legal/`. Accessed: 2020-08-27.

[9] M. Dudík, S. Lahaie, and D. M. Pennock, *A tractable combinatorial market maker using constraint generation*, in Proceedings of the 13th ACM Conference on Electronic Commerce, EC '12, New York, NY, USA, 2012, Association for Computing Machinery, p. 459–476.

[10] R. A. Fisher and F. Yates, *Statistical tables for biological, agricultural and medical research*, Oliver and Boyd Ltd, London, 3 ed., 1938.

[11] R. Freeman, S. Lahaie, and D. M. Pennock, *Crowdsourced outcome determination in prediction markets*, in Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, AAAI'17, AAAI Press, 2017, p. 523–529.

[12] R. Hanson, *Combinatorial Information Market Design*, Information Systems Frontiers, 5 (2003), pp. 107–119.

[13] R. Hanson, *Logarithmic markets scoring rules for modular combinatorial information aggregation*, Journal of Prediction Markets, 1 (2007), pp. 3–15.

[14] R. Jurca and B. Faltings, *Incentives for expressing opinions in online polls*, in Proceedings of the 9th ACM conference on Electronic commerce - EC '08, Chicago, Il, USA, 2008, ACM Press, pp. 119–128.

[15] ——, *Incentives for Answering Hypothetical Questions*, in Workshop on Social Computing and User Generated Content - EC '11, 2011, p. 9.

[16] C. Kroer, M. Dudík, S. Lahaie, and S. Balakrishnan, *Arbitrage-free combinatorial market making via integer programming*, in Proceedings of the 2016 ACM Conference on Economics and Computation, 2016, pp. 161–178.

[17] Y. Liu, J. Wang, and Y. Chen, *Surrogate scoring rules*, in Proceedings of the 21st ACM Conference on Economics and Computation, EC '20, New York, NY, USA, 2020, Association for Computing Machinery, p. 853–871.

[18] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani, *Algorithmic Game Theory*, Cambridge University Press, USA, 2007.

[19] J. Peterson, J. Krug, M. Zoltu, A. K. Williams, and S. Alexander, *Augur: a decentralized oracle and prediction market platform*, arXiv preprint arXiv:1501.01042, (2015).

[20] J. Surowiecki, *The Wisdom of Crowds: Why the Many are Smarter Than the Few and how Collective Wisdom Shapes Business, Economies, Societies, and Nations*, Doubleday, USA, 2004.

[21] J. Witkowski, *Robust Peer Prediction Mechanisms*, PhD thesis, 2014.