Pnyx: a user-friendly and powerful aggregation tool

a term–project paper by

Guillaume Chabin
guillaume.chabin@tum.de
student id: 03650708

■ 15 September 2014

Interdisciplinary project at the
Institute for Decision Sciences & Systems (DSS)
Technische Universitaet Muenchen, Germany

Assessor and Supervisor:
Prof. Dr. Felix Brandt
Institute for Decision Sciences & Systems (DSS)
Technische Universitaet Muenchen, Germany

Tutor and Advisor:
Christian Geist, M.Sc.
Institute for Decision Sciences & Systems (DSS)
Technische Universitaet Muenchen, Germany

Abstract

The aggregation of the preferences within a group of individuals is a common task. In 1951, Kenneth J. Arrow formalized a mathematical approach of this problem. He built the foundations of Social Choice Theory. Since then, scientists focused mostly on theoretical studies of decision schemes. It is only recently that computational aspects are taken into considerations, and there is now a lack of tools computing preference aggregations. The Pnyx Project tries to address this need by developing a user-friendly, powerful and open source aggregation tool.

Pnyx is a web-based application that allows the user to run customized polls, from the collection of individual preferences to the computation of a collective preference. It is also possible to export the collected preference profiles under the PrefLib standards to increase the compatibility with other tools in the field of social choice theory. This paper presents the major aspects of this aggregation tool from a theoretical approach based on social choice theory, and both functional and technical aspects with the introduction of the functionalities and their implementations.

Contents

1 Introduction

Collective decision-making, or at least the aggregation of the preferences within a group of individuals is a common task. A typical and straightforward example is the political election but the application of this field goes way beyond that. It goes from very advanced algorithmic topics in distributed artificial intelligence to very practical use in everyday life in both private and professional environments with the raise of collaborative projects.

More formally, the aggregation of preferences aims to answer a simple question. How can we represent the preferences (in the form of relation) of a set of individuals over some alternatives by a collective preference relation? This elementary problem is at the heart of social choice theory and has been taken into consideration since the first democracies. We can mention writings of Pliny the Young during Ancient Rome (1st century AD), of the Catalan philosopher Ramon Llull or of the Marquis Nicolas de Condorcet who was a very active philosopher and mathematician during the Age of Enlightenment in France. The scientific discipline came later in 1951 with the work of Kenneth J. Arrow who built the mathematical foundations. He introduced a theoretical framework to analyze the aggregation of preferences. Since them, most of the classical work focused on the result of theoretical possibility of properties such as fairness requirements of some decision rules. However, most of the results were impossibility results. For instance Arrow showed that an aggregation method with a list of basic fairness property leads to a dictatorial choice rule.

Recently, scientists in the field of *computational social choice* have brought new perspectives using methods from computer science and considering computational aspects of social choice. In addition to the existence, the computation of the aggregated preferences are now a point of interest. However there are nowadays too fews IT solutions to collect and aggregate the preferences. The Pnyx Project aims to address this need providing a universal and easy-to-use toll that support the process from the collection to the aggregation.

1.1 Existing solutions and motivations

At the early time of the Pnyx project, we realized a benchmark of 17 existing tools related to preferences aggregation. For each of them, we took into consideration the interface, the user experience and also the specific features related to preference aggregation. From this, we clearly saw a lack of appropriate tool on the web. The results of our benchmark identified 3 groups of tools that perform, often partially, preference aggregation is presented in figure 1.

The first group of apps can be considered as preferences collector. It regroups tools known by a large public such as the scheduling tool *Doodle* or more generally survey management tools like *Survey Monkey*. In general, these tools offer a good user experience with a very intuitive interface. However they do not aggregate the collected preferences or sometimes compute only very simple statistics. Using these tools to collect preference profiles results in misusage. We will see in this paper that there are different ways to express individual preferences and only a few of them are supported by such tools. Often the issue of anonymity regarding the poll organizer and other participants is not really well addressed.

The second group of tools are rather social choice tools that are limited to

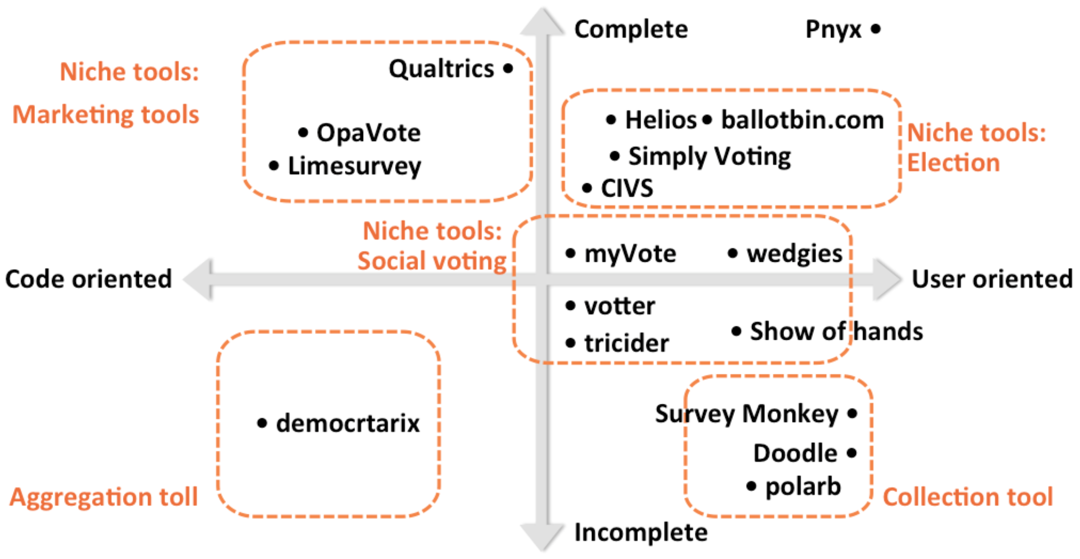


Figure 1: Matrix of existing solutions

winner determination. These tools allow the user to compute the aggregation of a given preference profile (a set of preference relations among a set of individuals). If numerous choice rules and preference domains are supported, such tools often require background knowledge in social choice theory to collect the preference profile initially. An interesting and open source project that goes in this direction is the *Democratix project* from TU Wien [1] that almost deals with winner determination through Answer Set Programming.

The last group contains apps that collect and aggregate the preferences but they are still very basic in the choice rules and the preference type supported. As *Wedgies*, most of these applications are commercial services targeting the niche of voting among existing social network platforms like Twitter or Facebook. Some other tools are designed for a specific purpose and will bring a better answer than Pnyx. Hence *Helios* is election oriented. It offers less flexibility than Pnyx but can handle cryptographic issue, which is something that Pnyx does not support currently.

The main goal of Pnyx is not to replace all this tools but to provide a user-friendly, flexible and open source tool that supports all preference aggression needs from the collection of individual preferences to the computation of the aggregated preference.

2 A theoretical approach of Pnyx

2.1 Preliminaries

In this section, we introduce some prerequisites to *social choice theory* and notations used in this paper.

2.1.1 Alternatives, voters and preference relations

Let A be a feasible set of alternatives with $|A| = m$ and $N = \{1, 2, \ldots, n\}$ be a finite set of voters. In general, an alternative is denoted by *x, y* or *z.* The set of all feasible subsets of A is denoted by $F(A)$. Each voter or agent is modeled by a preference relation over the set of alternatives denoted R[1]. In particular R_i denotes the preference relation of the voter i. We assume this relation to be *complete* (i.e. $\forall x, y, z \in A, xRy \wedge yRz \implies xRz$) and in most cases *transitive*, (i.e. $\forall x, y \in A, xRy \vee yRx$). The relation R can be interpreted as "at least as good as". The set of all complete preferences over A is denoted by $\mathcal{R}(A)$. If xRy and $\neg yRx$ we speak about strict preference, denoted by xPy. We denote n_{xy} the set of voters that strictly prefer x to y (e.g. $n_{xy} = \{i \in N, xPy\}$). The indifference between x and y, (i.e. xRy and yRx) is dented by w xIy or yIx.

Later in the document we introduce linear order of preference preference relation. This is relation *complete*, *transitive* and *asymmetric* (i.e. $\forall x, y \in A, xRy \implies \neg yRx$).

For a preference relation R and a feasible set A of alternatives, we introduce the *Maximal set*: $\max(R, A) = \{x \in A : \nexists y \in A, yPx\}$. The maximal set contains the best elements of A according to R and in general, it does not need to be a singleton. Moreover, if R is a transitive relation, this set is never empty. By completeness of R, the maximal set is also defined by $\max(R, A) = \{x \in A : \forall y \in A, xRy\}$.

A preference profile is defined as a set of individual preferences. $R_N = (R_1, R_2, \ldots, R_n) \in \mathcal{R}(A)^N$. We model an election by the triple (A, N, R_N) where A represents all the *alternatives* e.g. candidates, V the set of *voters* and R_N the *preference profile* containing the *preference relations* of each voters. Finally we introduce the *lottery*, which is a probability distribution over alternatives. The set of lotteries over a set of alternative A is denoted by $\Delta(A)$.

Definition More formally a lottery $\mathcal{L}$ over A is defined as follows

$$\mathcal{L} : A \to [0, 1] \text{ such that } \sum_{x \in A} \mathcal{L}(x) = 1$$

2.1.2 Preference aggregation

This subsection brings formal definitions around the *aggregation* of a given preference profile. Pnyx implements three different types of aggregation functions: t*he social choice function, the social welfare function and the lottery extensions.*

Definition A social choice function (SCF) is defined as follows:

$$f : \mathcal{R}(A)^N \times F(A) \to F(A) \text{ such that } \forall R_N, A, f(R_N, A) \subseteq A$$

A SCF is a function that maps a preference profile and a feasible set of alternatives to a subset of the given feasible set. The output can be considered

[1]The preference relation is sometimes denoted by $\succeq$ in literature

as the socially preferred alternatives and is often named the choice set. If the SCF returns a unique alternative for any preference profile, it is called *resolute*. To ensure the resoluteness, Pnyx breaks ties.
Pnyx also uses the so called social decision schemes (SDS) or randomized social choice functions. These functions map a preference profile to a lottery over a fixed set of alternatives.

Definition A social decision scheme (SDS) is defined as follows:

$$f : \mathcal{R}(A)^N \times F(A) \to \Delta(A)$$

Compared to SCFs, the randomization of SDS brings appealing properties. However, the extension of preference relations to lotteries is not trivial and will not be addressed in this paper. For more details please refer to [2].
The last group of aggregation methods supported by Pnyx are social welfare functions (SWF). It maps a preference profile and a feasible set of alternatives to a complete preference relation. This collective relation represents the preference of the group of voters and is denoted by R

Definition A social welfare function (SWF) is a function f such that:

$$f : \mathcal{R}(A)^N \to \mathcal{R}(A) \text{ such that } \mathcal{R}(A) \text{ is transitive}$$

2.2 Individual preferences and aggregated preferences

As already mentioned, Pnyx allows to collect individual preferences among a set of voters. We selected 5 types of individual preferences that fits real use cases. Here is a brief description of all cases - in increasing order of constraints.

Complete preference: Each voter has to specify all the pairwise comparisons among the set of alternatives. For each comparison one can enter a strict preference xPy, yPx or an indifference xIy.

Complete preorder of preference: [2]To the former ballot, we add a transitivity constrain. In practice, this individual preference is a ranking over the alternatives allowing indifferences.

Linear order of preference: This is a refinement of the complete preorder. The individual preferences are complete transitive and asymmetric (e.g. ties are not allowed anymore)

Dichotomous preferences: A preference relation R on A is dichotomous if for all alternatives $x, y, z \in A$, xPy implies that either zIx or zIy. Each voter distinguishes between only two equivalence classes.

Dichotomous preferences with a unique best alternative: In addition to dichotomous preferences, each voter can only select a unique alternative among all the alternatives. $\exists x \in A, \forall y \in A \setminus \{x\}, xPy$.

For the collective preferences Pnyx supports 3 different types:

[2]sometimes called bucket ordered preference

Unique winner: The aggregated preference is represented by a unique alternative. This is the result of a resolute SCF and represents the most preferred alternative.

Lottery: The aggregated preference is represented by lottery over the alternatives. This is the result of a SDS and represents the probability for each alternative to be the most preferred alternative.

Linear order of preference: The aggregated preference is represented by a linear order. This is the result of a SWF

We mentioned in the previous section that the solutions studied in the benchmark don't support all kind of individual and collective preferences. Table 1 gives a global picture of the situation, and it clearly shows that Pnyx is the only aggregation tool supporting as many preference types.

Table 1: Support of the different preferences types

input output	**Dichotomous preferences with a unique best alt.**	**Dichotomous preferences**	**Asymmetric transitive complete preferences**	**Transitive complete preferences**	**Complete preferences**
Unique winner	Pnyx, OpaVote, ballotbin	Pnyx, OpaVote, ballotbin	Pnyx, OpaVote, ballotbin	Pnyx	Pnyx
Lottery	Pnyx, Votter, show of hands, wedgies, poll daddy, SurveyMonkey, Simply Voting, Qualtrics	Pnyx, SurveyMonkey, Qualtrics	Pnyx, SurveyMonkey	Pnyx	Pnyx
Asymmetric transitive complete preferences	Pnyx	Pnyx	Pnyx, myvote, polldaddy	Pnyx	Pnyx
Transitive complete preferences	None	None	None	CIVS	None

The notion of individual and collective preferences may appear abstract. Table 2 is an illustration of real-life use-cases that would fit particularly well the different tuples (input & output) supported.

2.3 Supported choice rules

In the following we recall the voting rules supported by Pnyx and then present a brief analysis of the most advanced choice rules.

2.3.1 Definitions

The selection of the choice rule is induced by the input type and output type chosen by the user. By hiding this step to the user, we want the tool to be more straightforward and to avoid the requirement of any extra knowledge

Table 2: Real-life scenario to

input output	**Dichotomous preferences with a unique best alt.**	**Dichotomous preferences**	**Asymmetric transitive complete preferences**	**Transitive complete preferences**	**Complete preferences**
Unique winner	Elect a representative	Decision in a committee	Chose the name of a project	Select the activity of the next team building trip	Elect the favorite meal of a target audience
Lottery	Survey for the presidential election	Allocation of a common good	Survey for the next team-building activity	Who will win the next football world cup?	Survey to select the name of a new brand
Asymmetric transitive complete preferences	Jury of Pitch contest	Jury of American Idol	Movie rankings	Song rankings	Car rankings by a target audience

in Social Choice Theory. Table 3 is the matrix that illustrates the assignment of the choice rule for each possible tuple.

Table 3: Pnyx choice rules matrix

input output	**Dichotomous preferences with a unique best alt.**	**Dichotomous preferences**	**Asymmetric transitive complete preferences**	**Transitive complete preferences**	**Complete preferences**
Unique winner *	Plurality	Approval voting	Borda's scores	Bucket Borda's rule	Young's scoring rule
Lottery	Random dictatorship	Nash's solution	Maximal lottery	Maximal lottery	Maximal lottery
Asymmetric transitive complete preferences*	Plurality scores	Approval voting scores	Kemeny's rule	Kemeny's rule	Kemeny's rule

* Ties are broken by a rule specified during the poll creation.(It is either a randomized or a customized lexicographic order).

Plurality: This is a very basic and ubiquitous SCF. It selects the alternatives that are ranked first by most voters. Despite its simplicity this choice rule satisfies several interesting properties such as *anonymity* (each voter is treated equally) , *neutrality* (each alternative is treated equally) but it fails to be *strategyproof* and *Pareto optimal.* A SCF is *strategyproof* if a voter never benefits in lying to get a better outcome, under the assumption that he knows the preferences of all other voters. A SCF is *Pareto optimal* if an alternative will not be chosen if there exists another alternative such that all voters prefer the latter to the former.

Approval voting: Approval voting selects the alternatives that gathered most approvals. It leads to the same results as plurality but the terminology *approval* *voting* is only used for dichotomous profile. It is also known to be $\text{Max}(R_M, A)$, where R_M is the pairwise majority relation (i.e. $xR_My \iff n_{xy} \geq n_{yx}$). It is also a very uncontroversial choice rule because it satisfies *neutrality*, *anonymity*, *participation* (no voter benefits from not voting) and *strategyproofness.*

Borda's rule: For this choice rule the individual preferences should be linear orders. It is a (positional) scoring rule. For a fixed number of alternatives m, this class of choice rule is characterized by a scoring vector $s = (s_1, \dots, s_m)$ and if a voter ranks an alternative at the i^{th} position, it gets s_i points. The scoring rule chooses those alternatives for which the accumulated score is maximal.

$$f(R_N, A) = \arg\max_{x \in A} s(x, A) \text{ where } s(x, A) = \sum_{i \in N} s_{|y \in A : yR_ix|}$$

For Borda's rule, $s = (|A| - 1, |A| - 2, \dots, 0)$. It selects the alternatives with the highest average rank in individual rankings. From the definition we can see directly that scoring rules are *anonymous* and *neutral.* Borda's rule also satisfies *participation.*

Bucket Borda's rule: As presented by John Cullinan, Samuel K. Hsiao and David Polett [5], we use a generalization of Borda's rule to complete preorders of preference. Given a preference profile R_N,

$$f(R_N, A) = \arg\max_{x \in A} s(x, A) \text{ where } s(x, A) = \sum_{i \in N} 2|y \in A : xP_iy| + |y \in A \backslash \{x\} : xI_iy|$$

Young's scoring rule: It is a rule introduced by Young while he was studying Borda's rule [6]. This is a generalization of both former rules that supports complete binary relations.

$$f(R_N, A) = \arg\max_{x \in A} s(x, A) \text{ where } s(x, A) = \sum_{y \in A \backslash \{x\}} n_{xy} - n_{yx}$$

Kemeny's rule:

Kemeny's rule returns all rankings that maximize pairwise agreements with the individual preferences. As a consequence, we need complete individual preferences to compute the output. Kemeny's rule satisfies very appealing axiomatic properties, and has been described or studied by many scholars: Kemeny in 1959, Young in the 1980s and we can find seminal work of Condorcet in 1785.

Like the former choice rules, Kemeny's rule satisfies *neutrality* and *anonymity.* In 1998, Condorcet showed that Kemeny's rule is the maximum-likelihood SPF for any $p \in [0 : 1]$. To explain what is a maximum likelihood SPF, let first introduce a probabilistic model of binary preference relation. Let's assume that each voter has to evaluate a binary relation which has an intrinsic order and that each voter has the probability p to guess the truth. We can assume that $p \geq 0.5$. The choice rule is a maximum likelihood SPF for the given p if it yields those rankings that are most likely to be "true".

Another characterization of Kemeny's rule is that it leads to a maximal weight acyclic majority subgraph. To compute Kemeny's rankings, we used the

MIP formulation of this characteristic:

$$\begin{aligned}
\underset{x_{ij}}{\text{minimize}} \quad & \sum_{i,j\in A} w_{ij}x_{ji} \text{ where } w_{ij} = \max(n_{ij} - n_{ji}, 0) \\
\text{subject to} \quad & x_{ii} = 0, \ \forall i \in A \\
& x_{ij} + x_{ji} = 1, \ \forall i \neq j \in A \\
& x_{ij} + x_{jk} + x_{ki}, \geq 1 \ \forall i \neq j \neq k \in A \\
& x_{ij} \in \{0, 1\}, \forall i, j \in A
\end{aligned}$$

The objective function minimizes the pairwise disagreement and the constraint forces the output to be transitive, asymmetric and complete. As you can expect from the exponential number of constraints, Kemeny ranking problem is NP complete. More formally, Bartholdi et al. showed [4] that deciding whether there exists a ranking with which Kemeny score is at least 1 is NP- complete. As a consequence, finding a Kemeny ranking is NP-hard.

Nash's solution:

Nash's solution is computed when the individual preferences are dichotomous and the aggregation is a lottery. We select Nash's solution based on Bogomolnaia's reflexion [3]. The general idea is to maximize a collective utility function with a particular shape. The choice of the utility function leads to different solutions and has naturally an impact on the axiomatic properties of SDS. For Nash's solution, the objective function is a logarithm. Here his the convex optimization problem to compute the lottery.

$$\begin{aligned}
\underset{p}{\text{maximize}} \quad & \sum_{i\in N} log(U_i \cdot p) \\
& \text{where } U_i \text{ is the approval vector of voter } i \\
\text{subject to} \quad & 0 \leq p_i \ \forall i \in A \\
& \sum_{i\in A} p_i = 1
\end{aligned}$$

The approval vector is a 0-1 vector where each approved alternative is labeled with 1 and with 0 otherwise. This solution is naturally *neutral* and *anonymous* and the concavity of the logarithm brings very interesting properties for decision making under dichotomous preference, in particular *ex-ante efficiency and fair welfare share.*

Ex-ante Efficiency: It could be view as a probabilistic interpretation of *Pareto optimality* for dichotomous preferences. As Pareto optimality, it guarantees given a Nash's solution p, there is no other solution p' that will satisfy better at least one voter without negative impacts on some other voter's utility. More formally:

$$\forall p', \{U \cdot p' \geq U \cdot p\} \Rightarrow U \cdot p = U \cdot p'$$

Fair Welfare Share: It guarantees a lower bound on the utility of all agents who are not completely indifferent. And more precisely we have

$$\forall i \in N, U_i = 0 \Rightarrow U_i \cdot p \geq \frac{1}{|N|}$$

However Nash's solution fails to be strategyproof. Indeed, Bogomolnaia and al. showed [3] that assuming $|A| \geq 5$ and $|N| \geq 17$, an anonymous and neutral mechanism cannot be ex-ante efficient, strategyproof, and meet the fair welfare share.

Maximal Lottery: Maximal Lotteries are recent mechanism, initially introduced by Kreweras in 1965 and Fishburn in 1984. It is an extension of maximal alternatives (weak Condorcet winners) to lotteries. x is maximal if and only if, $\forall y \in A, g(x,y) = |\{i \in N, xR_iy\}| - |\{i \in N, yR_ix\}| \geq 0$. Hence, for $p, q \in \Delta(A)$, we define $g(p,q) = \sum_{x,y \in A} g(x,y)p(x)q(y)$, and p is a maximal lottery if and only if $\forall q \in \Delta(A), g(p,q) \geq 0$. An interpretation of an ML p can be a lottery where the expected number of voters who are happier with q's outcome than with p's is never greater than the number of voters who are happier with p's outcome than with q's.

From the definition, we can easily show that the set of Maximal Lotteries given A is convex. It is even a polytope, but it can be large, and during the computation we try to approximate the barycenter to act as a tie break.

Rather than using the latter definition, we use an approach based on game theory and *v. Neumann's theorem* (1928), where g is seen as a *symmetric two-player zero-sum game*. From that we can get the existence of maximal lottery and also that Maximal Lotteries are mixed minimax strategies of the plurality game. Hence we can simply compute MLs by linear programing. For more details, please refer to the reference [?].

$$\begin{aligned} &\forall a \in A \\ &\qquad \text{maximize} \quad p_a \\ &\qquad \text{subject to} \quad \sum i \in A p_i n_j, \ \forall j \in A \\ &\qquad\qquad\qquad\quad 0 \leq p_i, \ \forall i \in A \\ &\qquad\qquad\qquad\quad \sum_{i \in A} p_i = 1 \end{aligned}$$

Plurality game being a *zero-sum game*, the LP problem is reduced to a feasibility problem and the objective function can be arbitrarily chosen. The computation is processed in 2 steps. First, we compute for each alternative the ML maximizing its probability in the lottery. Then, we take the barycenter of all computed lotteries. Thus we are approaching the gravity center, but there is no guarantee to hit it.

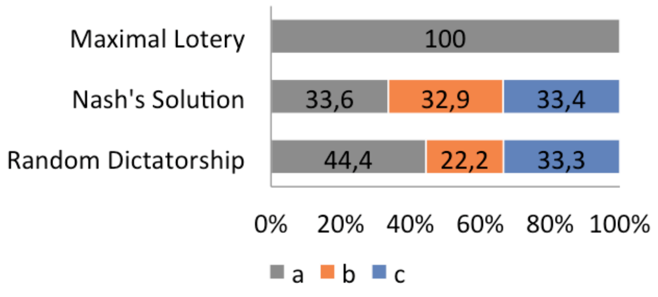
2.3.2 Relation and generalization

In this paragraph, we study the relation between the choice rules and try to figure out whether there is some generalization relations within the decision matrix. The problem is the following: for a given output type, and an equivalent preference profile, do we have the same outcome within a row of the decision matrix?

Unique winner: As already mentioned above, the choices rules selected for unique winner return the same output given a preference profile.

4	3	2
a	c	b
b,c	a,b	a,c

(a) Preference profile



(b) The different aggregation

Figure 2: Generalization for lotteries

Asymmetric transitive complete preferences: In theory, we also have the relation of generalization from left to right. Indeed, the majority relation R_m is transitive on dichotomous profile. Therefore, the ranking given by approval scores and plurality scores is transitive and is de facto a Kemeny ranking. In practice Pnyx performs a tie breaking, and as explained later, the outcome may differ if the tie breaking is not performed similarly.

Lottery: Lotteries outcomes fail to satisfy this generalization. Figure 2 shows a counterexample with 9 voters and 3 alternatives. First we can remark that the maximal lottery is degenerated while the others are not. In this example a is a *Condorcet winner* (i.e. $\max(A) = \{a\}$) and a Condorcet winner is always selected by the ML. On the other hand, Nash's solution and random dictatorship guarantee the alternatives b and c some non null score. Hence MLs are *Condorcet extension* (i.e. the choice rule select only the Condorcet winner when it exists) and the others are not. We can also see that Nash's solution is less discriminating than the random dictatorship mechanism. This is due to the logarithm, which overweights the votes of participants with only few alternatives selected. It reduces the impact of the approval scores compare to the random dictatorship. This could intuitively explains the *fair welfare share* property of Nash's solution.

3 A generic use case

Before detailing the implementation, we would like to introduce in this section the user interface and the general workflow of Pnyx with a generic use case. Santa would like to know what present to offer to the Lucky family for Christmas. He would like to ask each family member to rank 3 possible gifts. Santa logs in to his Pnyx account. From the dashboard he can access the creation

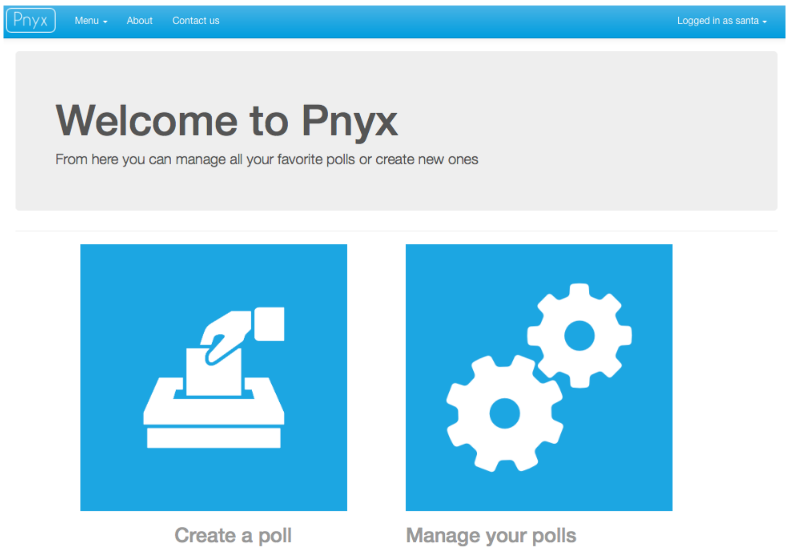
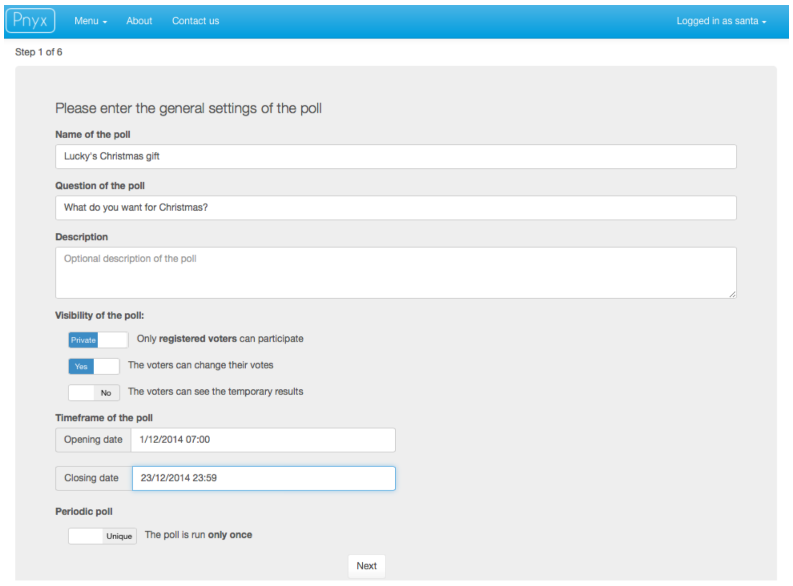


Figure 3: Pnyx dashboard

page simply by clicking the corresponding icon (see figure 3).



Step 1 of 6

Please enter the general settings of the poll

Name of the poll: Lucky's Christmas gift

Question of the poll: What do you want for Christmas?

Description: Optional description of the poll

Visibility of the poll:
- Private — Only **registered voters** can participate
- Yes — The voters can change their votes
- No — The voters can see the temporary results

Timeframe of the poll
- Opening date: 1/12/2014 07:00
- Closing date: 23/12/2014 23:59

Periodic poll
- Unique — The poll is run **only once**

Figure 4: Create a new poll: general settings

The creation of a new poll is split into 6 steps, for each one, there is a dedicated form to fill up. The first step (see figure 4) deals with the general settings of the poll. He specifies the name and the question of the poll. The poll will be private (only the Luckys will be able to vote, and only Santa and the family will see the results). He specifies the opening date and the closing date and that the poll is not periodic.

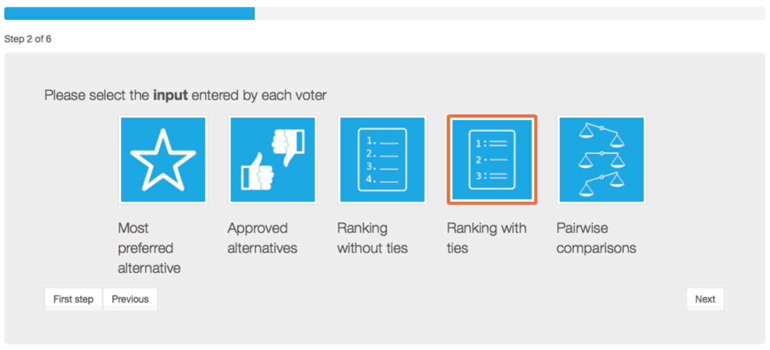


Figure 5: Create a new poll: input type selection

The second step selects the individual preference type. IN this particular case, indifference between gifts may happen and the appropriate individual preference type seems to be a complete preorder.The third step is the choice of the collective preference. Santa would like to have a unique winner. (See figures 5 & 6)

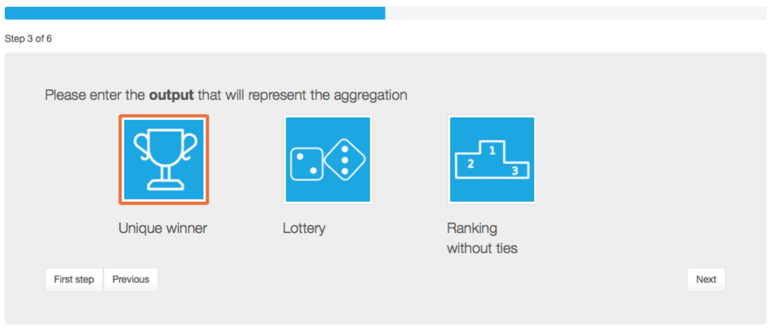


Figure 6: Create a new poll: output type selection

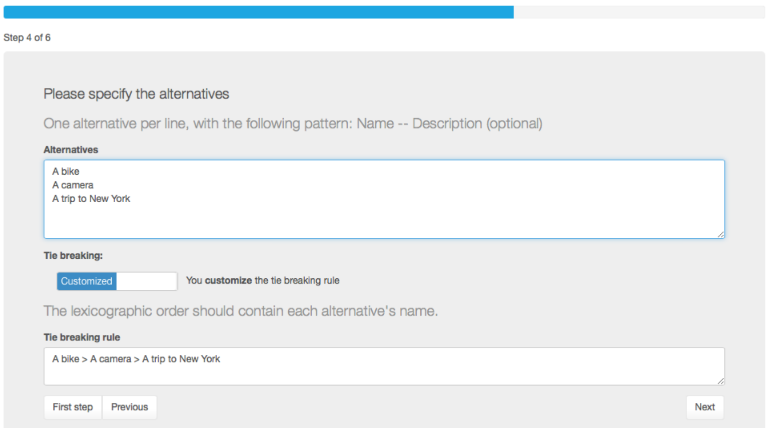
In the fourth step (figure 7) Santa specifies the alternatives (i.e. the possible gifts): a bike, a camera and a trip to New York. He also specifies the tie-breaking rule. He could have chosen a randomized rule but he has his own idea about the rule and customized it. A bike is difficult to put in his sledge and the trip is hard to organize. Therefore he choses the rule: A camera $>$ A trip to New York $>$ A bike.

Then he needs to specify the emails of each participants for them to receive the link to vote and to access the final result. (See figure 8)

The final step is a preview of the new poll that Santa is about to create. Just before the creation, he is able to check every think and if he wants, he can go backward to update the forms. He confirms the preview, and the poll is created.

Now, he can manage his poll from the *manage existing polls* page. From there he will be able to email the participant, modify the poll, extract the preference profile, access the results and a summary of the poll.

Once the poll is created, each member of the family receives an email with


Step 4 of 6

Please specify the alternatives

One alternative per line, with the following pattern: Name -- Description (optional)

Alternatives

```
A bike
A camera
A trip to New York
```

Tie breaking:

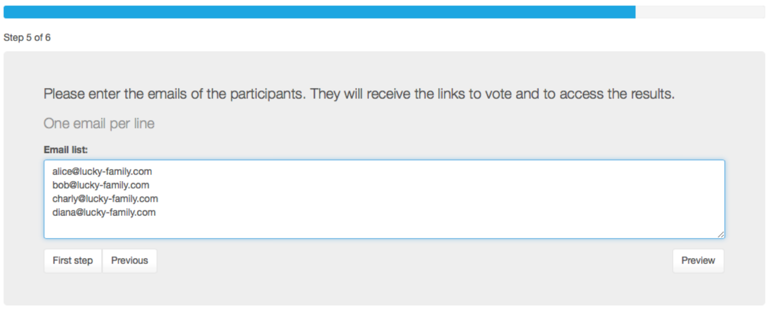
Customized — You **customize** the tie breaking rule

The lexicographic order should contain each alternative's name.

Tie breaking rule

```
A bike > A camera > A trip to New York
```

Figure 7: Create a new poll: alternatives definition


Step 5 of 6

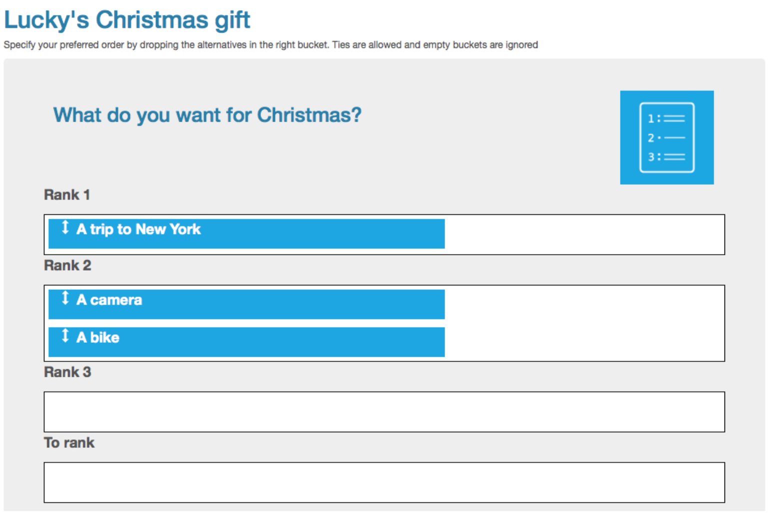
Please enter the emails of the participants. They will receive the links to vote and to access the results.

One email per line

Email list:

```
alice@lucky-family.com
bob@lucky-family.com
charly@lucky-family.com
diana@lucky-family.com
```

Figure 8: Create a new poll: participants declaration


Lucky's Christmas gift

Specify your preferred order by dropping the alternatives in the right bucket. Ties are allowed and empty buckets are ignored

What do you want for Christmas?

Rank 1

- A trip to New York

Rank 2

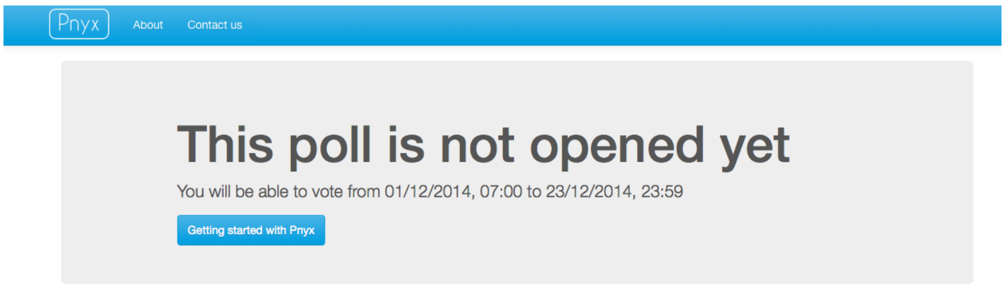
- A camera
- A bike

Rank 3

To rank

Figure 9: A ballot for bucket preferences

the timeframe, a link to vote and a link to see the results. Mr Lucky clicks on the link of the ballot. If the time frame is right, he can see the ballot shown in figure 9. Otherwise the poll is not opened yet or already closed, and he is redirected to an error page (figure 10).

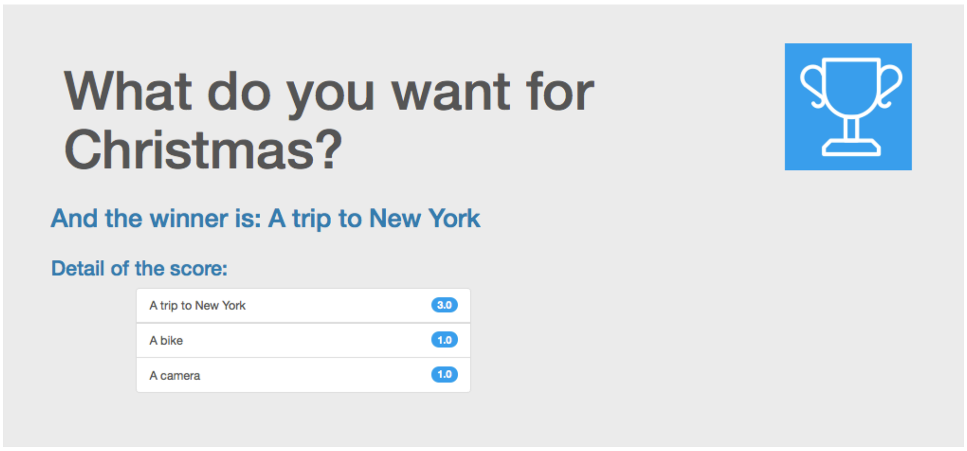


This poll is not opened yet

You will be able to vote from 01/12/2014, 07:00 to 23/12/2014, 23:59

Figure 10: A redirection page

During the poll Santa can see the temporary results, but because in he didn't set the temporary results to visible the participants cannot. But once the poll is closed, Santa and all the family can see the final results like in figure 11.



What do you want for Christmas?

And the winner is: A trip to New York

Detail of the score:

A trip to New York	3.0
A bike	1.0
A camera	1.0

Figure 11: A results page

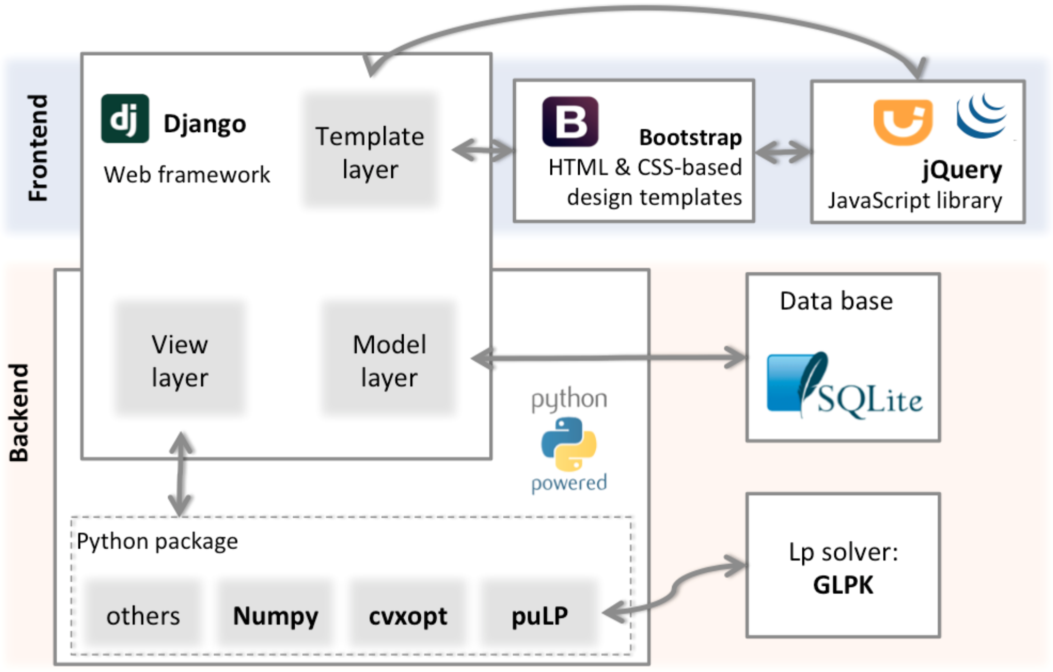
4 Technical and functional documentation

This chapter details the front-end and the back-end implementations of Pnyx. The goal is to allow a reader with basic programming skills to understand the implementation of Pnyx so that he will have a big picture of the tool and he will be able to keep on working with the project. You will also find here useful links to find more details about the technical aspects of Pnyx. We mentioned in the introduction that Pnyx is open source. It is currently under the permissive MIT license [7] and the source code is available on a Github repository: github.com/gchabin/pnyx. Please note that some technical dependencies may use other open source licenses with their own conditions.

4.1 General introduction and technical dependencies

Pnyx is a web application developed in Python 2.7 [8] with several technical dependencies, figure 12 draws a big picture of the tool and explains how modules are interacting together.

Figure 12: General architecture of Pnyx



From a general point of view, we can distinguish 2 major elements in the program: the front-end and the back-end. The front-end is the part of the application visible by the users, and the back-end is the hidden part of Pnyx handling all the logic of the application.

On the diagram you can see the distribution of the technical core of the application in both sides. You can easily remark that the module called *Django* seems to play an important role. *Django* will be introduced in the next subsection, but from now we can consider *Django* as the web framework that build the skeleton of the application, linking the front-end and the back-end together. The 2 remaining modules part of the front-end are *jQuery* and *Bootstrap*. The former contains standard JavaScript libraries. Pnyx uses *jQuery JavaScript Library* and *jQuery UI* which is an extension of *jQuery* explicitly designed for the user interface interactions. The latter is an HTML, CSS, and JS framework very popular to develop web applications. *Bootstrap* allows the developer to design nice user interfaces without deep knowledge in web design. It contains

HTML and CSS-based templates for typography, forms, buttons, navigation and other interface components that are used as building blocks to create each page. *Bootstrap* [9] provides a few easy ways to quickly get started. In addition to the bootstrap library Pnyx used a customized theme from *bootswatch.com*

In addition to Django, the back-end contains the model layer, an LP solver and some python modules. The model layer stores all the data of the application and is detailed in the next section. The LP solver is used to compute *maximal lotteries* and in *Kemeny's rule* to perform Mixed Integer Programming. The current LP solver is the *GNU Linear Programming Kit*[11]. To integrate the solver in the code, we use *puLP* [10]. It is an LP modeler written in python imported as a package. It creates LP files and calls directly the LP solver *GLPK*. It is also compatible with other LP solvers such as *COIN* , *CPLEX* or *GUROBI*. Other classical modules for python programming are also used in the back end. We can mention *NumPy*[12], used for scientific computing and *CvxOpt*[13] used for convex optimization used in the computation of the Nash's solution. Please refer to the official documentation [14] for more details. It contains a useful user documentation with numerous examples for different kinds of optimization problems.

4.2 The architecture of Pnyx as a Django application

From a bird's-eye view Django is the web framework used to build the application and is implemented in Python. Pnyx implements the version 1.6 of Django, it was the latest stable release at the time of the development. Django comes up with several functionalities that are regularly used in web application and make the development of a web application, quicker, easier and less redundant. The scientific aspect of Pnyx has been a key factor in the choice of Python as a web programming language. From that, Django is a natural framework to build the application. The Django project provides a short but complete tutorial[16] that introduces most of the features used in Pnyx. We assume the reader to have the corresponding knowledge of Django and the documentation details the scenarios where the use of the framework is slightly more advanced.

Django uses the *Model, View, Template* model, which is very similar to the classical *MVC* model.

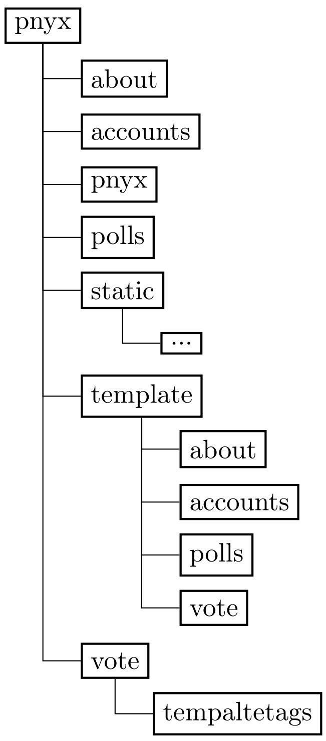
The *model layer* is an abstraction layer used to structure and to manipulate the data of the application. This built-in functionality handles the integration of the databases in the application: from the creation of the tables and their relationships to the queries.

The *template layer* is related to the front-end side of the application. It provides a designer-friendly syntax to render the information to be presented to the user. It makes the html development more responsive and dynamic. Like in Python, template code supports local variables, if statement or for loops...

The *view layer* encapsulates the logic responsible for processing a user's request and for returning the response. For more information about *Django*, please refer to the official documentation[15].

To understand the architecture of a Django application(figure 13), you need to know that a such application is divided into several sub applications. Each sub application represents a feature of the application and has a dedicated directory, views, templates, urls namespace... Here is the list of the sub applications

Figure 13: Directory tree of Pnyx



used by Pnyx.

built in apps: Built-in applications not represented in figure 13, that support classical functions such that the wizard, the admin user-interface, the authentication...
django.contrib.admin, *django.contrib.auth*, *django.contrib.contenttypes*, *django.contrib.sessions*, *django.contrib.messages*, *django.contrib.staticfiles*, *django.contrib.formtools*

about: handles the *about* page. This is a trivial sub application with only one page.

accounts: handles the pages and the logic behind the authentication with the help of the built in modules.

pnyx: The eponym sub application is taking care of the smooth running of Pnyx. It contains the namespaces declaration for the urls, the settings file *settings.py* and the *wsgi.py* file which is a specification for simple and universal interface between the web server and the application.

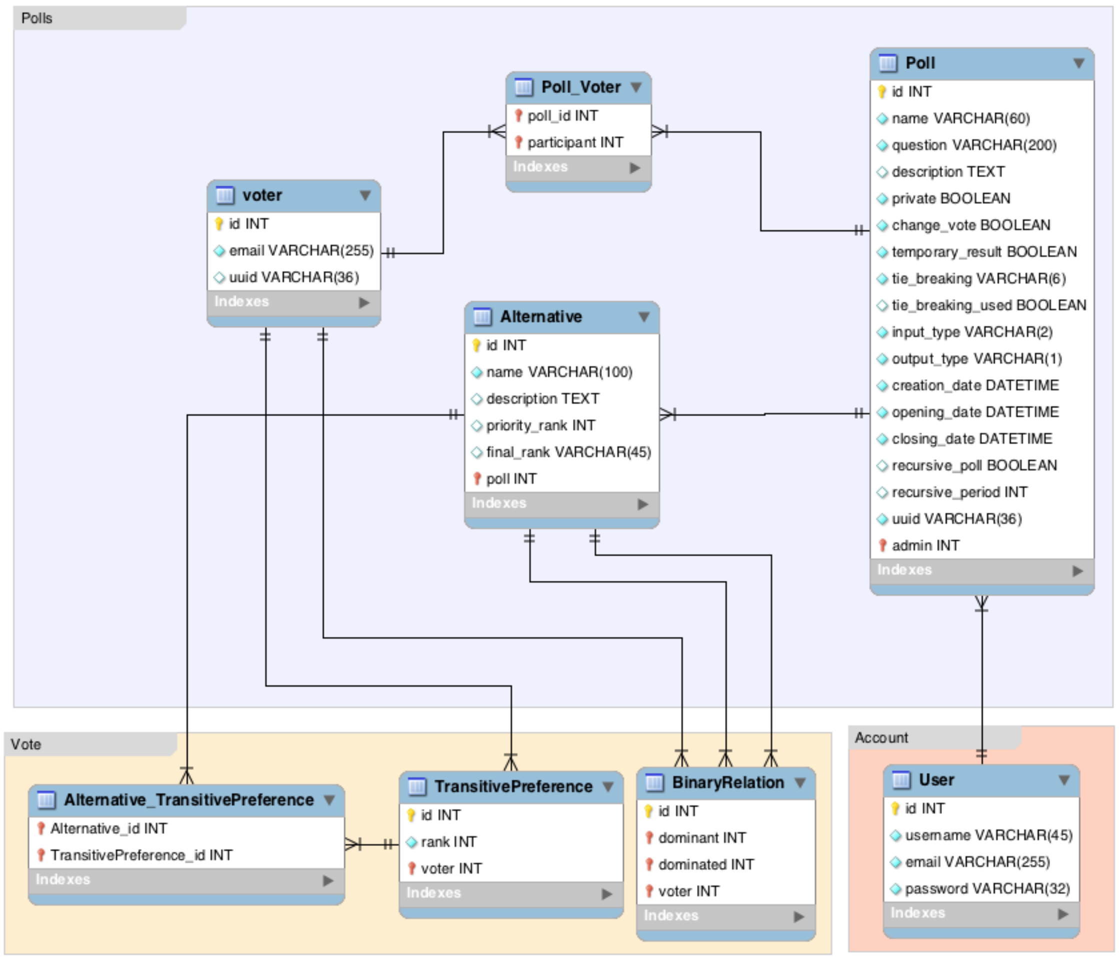
polls: handles the pages and the logic related to the poll creation and the poll management. In other word it is where the user is acting as a poll administrator and need to be registered.

vote: handles the pages and the logic related to the collection and the aggregation of the preferences and the results pages. It's the side of Pnyx where the user is acting as voter.

As we can see in figure 13 presenting the directory tree, there is a folder for each sub application but also additional folders named *static* and *template*. The former contains all the external references used by the application like source code of libraries or images and the latter is a dedicated folder to the template files.

4.3 The model layer

Figure 14: Database architecture schema



In this section we present the model layer. We will see how each aspect of a poll is defined in the database. The current version of Pnyx uses a LiteSQL database contained in the file *pnyx/db.sqlite3*. Given the scope of Pnyx, the application has to interact with different types of object detailed in figure 14. They are 5 main tables: the Poll's table, the Alternative's table, the Voter's table, the Transitive Preference's table and the Binary Relation's table. The objects are relatively straightforward and most of the features of a poll correspond to an attribute in a table. In the diagram, not colored diamonds are fields where the NULL value is allowed, the yellow keys are primary keys and the red keys are foreign keys (for one-to-many relationships). The join tables *Alternative_ TransitivePrefrerence* and *Poll_ Voter* are automatically created by Django to support the *many-to-many relations*.

A *poll*, is defined by a question, a name and may contain an optional description. It is created and managed by a unique user: the *admin*. A poll can be either *public* or *private*, and during private polls we may authorized voters to change their votes (*change_ vote* attribute). If *temporary_ result* is set to *True*, the voters can access the temporary results. To break ties, we use a lexicographic order that is either randomly defined (*tie_ breaking* set to *random*) or customized by the admin (*tie_ breaking* set to *custom*). The *tie_ breaking_ used* attribute indicates wether the final aggregation has required a tie break. There are 3 states: *True*, *False* or *Unknown*. The poll object stored also the input type and the output type, which fully specifies the choice rule. Each poll stores several date-time: the creation date, the opening date and the closing date. The attributes *recursive_ poll* and *recursive_ period* are optional and dedicated to periodic polls. More details are given in the next paragraphs.

Naturally a poll needs alternatives. Each one is defined by a name, an optional description and its poll. The relation between an alternative and a poll is symbolized by a one-to-many relation. Alternatives of a same poll cannot share the name. In addition to alternatives, a poll requires voters. Each voter is defined by a unique uuid and an email address. By default the email address is *unknown_ voter@pnyx.com*, it is the one used for public polls. A many-to-many relation links voters and polls. From that it is possible to exctrat for each poll the list of participants and vice versa.

We will now detail the implementation of visibility, vote and timeframe restrictions that are more complex and that require more explanations.

4.3.1 How poll visibility restriction is supported

A poll can be either private or public. The idea behind is to restrict the access of the ballot and the results of private polls to registered voters. The implementation is based on a token system (named here *voter_ uuid*) in the url that identifies the voter and authorizes him the access the page.

Code 1: Implementation of the visibility restriction in vote/urls.py

```
url(r'^(?P<pk>\d+)/(?P<voter_uuid>[a-z0-9\-]+)/temp-results/$',
    views.temporary_results, name = 'temp_results'),
```

Code 2: Implementation of the visibility restriction in vote/views.py

```
def temporary_results(request, pk, voter_uuid):

  poll = get_object_or_404(Poll, pk = pk)
```

```
      if not (poll.temporary_result or poll.admin == request.user):
#return temporary result not available for this poll
       return HttpResponseRedirect(reverse('vote:no_temp_results',
           kwargs = {'pk': poll.pk, 'voter_uuid': voter_uuid}))

# check if the voter is valid
elif voter_uuid == 'public' and not poll.private:
            pass
elif poll.admin == request.user and voter_uuid == 'admin':
            pass
elif poll.private:
            try:
      voter = get_voter_by_uuid(voter_uuid)
   poll.participant.all().get(uuid = voter_uuid)
      except (KeyError, Voter.DoesNotExist):
             return HttpResponseRedirect(
                        reverse('vote:unauthorized', kwargs =
                            {'pk': poll.pk, 'voter_uuid':
                            voter_uuid}))
else:
  return HttpResponseServerError("The voter UUID is not valid")
return compute_and_display_results(request, poll, False)
```

In this example we can see the 3 possible values of the token.

public: This token has to be present in url if the poll is public. The urls containing *public* are simple and shared by every voter.

a voter uuid: If the poll is private, the url doesn't contain the *private* keyword but rather the uuid of the voter who is accessing the page. This uuid is a complex, unique and randomly generated string that turns the url into a personal link.

admin: The token *admin* provides to the admin some additional access rights. For instance he is always able to see the temporary results. To perform the authentication in this case, the *admin* keyword should be present in the url and the logged-in user has to be the creator of the poll.

Once the voter is identified, and depending on the characteristics of the poll, the back-end either displays a requested page or redirects to an error page.

4.3.2 How vote restriction is supported

If the poll is private there is the possibility to allow the voters to change their votes after the submission. The technical solution is also based on a token system in the url, therefore we only describes briefly the workflow. Assume a registered voter of a private poll tries to access the ballot. The back-end identifies him with the token *voter_ uuid* and looks in the database for preferences matching the poll and the voter.

If no preference is found, the default ballot is displayed. Otherwise, we check if the voter is allowed to change his vote. If so, we load the data in the context and display the ballot with the previous vote, otherwise the voter is redirected to an *already voted* page.

Code 3: Implementation of the vote restriction in vote/views.py

```
def get_ballot_view(request, pk, voter_uuid):

        poll = get_object_or_404(Poll, pk=pk)
 ballot_data = {}
 # check if the voter is valid
        if voter_uuid == 'public' and not poll.private:
           pass
        elif poll.private:
            try:
                voter = get_voter_by_uuid(voter_uuid)
                 poll.participant.all().get(uuid = voter_uuid)
                 if poll.input_type == 'Bi':
                                       ...
                else:
                        previous_vote =
                             TransitivePreference.objects.filter(voter =
                             voter, alternative__poll =
                             poll.pk).order_by('rank')
                if len(previous_vote) != 0 and not poll.change_vote:
                 # the vote is already saved and the voter cannot change
                      the vote
                        return HttpResponseRedirect(
                                reverse('vote:already_voted', kwargs =
                                     {'pk': poll.pk, 'voter_uuid':
                                     voter_uuid}))
                elif len(previous_vote) !=0:
                        logger.debug("previous vote retrieved from
                             database: " + str(previous_vote))
                        # the vote is already saved and the voter CAN
                             change the vote
                        # get the data from the vote and parse it to the
                             ballot
                        for pref in list(previous_vote):
                            ballot_data[pref.rank] = pref.alternative.all()
            except (KeyError, Voter.DoesNotExist):
                return HttpResponseRedirect(
                                        reverse('vote:unauthorized', kwargs
                                            = {'pk': poll.pk, 'voter_uuid':
                                            voter_uuid}))
            else:
                 return HttpResponseServerError("Invalid voter UUID")
                        ...
        return render_to_response(template_name,
                {"poll": poll,
                'voter_uuid':voter_uuid,
                'ballot_data':ballot_data},
                RequestContext(request))
```

When the user submits his vote, a similar check is performed to prevent him to vote twice with the same ballot.

4.3.3 How timeframe and periodicity are handled

As mentioned above each poll is delimited by an opening date and a closing date. Before displaying any ballot or processing a vote, the back-end checks if the poll is running or not. If not the user is redirected to the appropriate informative page (the poll is closed or to opened yet).

Code 4: Implementation of the visibility restriction in vote/views.py

```
def get_ballot_view(request, pk, voter_uuid):

        poll = get_object_or_404(Poll, pk=pk)
        ...
        if poll.closing_date < timezone.now():
            #return the poll is closed
            return HttpResponseRedirect(reverse('vote:poll_closed',
                        kwargs = {'pk': pk , 'voter_uuid': voter_uuid}))
        elif poll.opening_date > timezone.now():
            #return the poll not opened yet
            return
                HttpResponseRedirect(reverse('vote:poll_not_opened_yet',
                        kwargs = {'pk': pk , 'voter_uuid': voter_uuid}))

        input_type = poll.input_type
        template_name = get_ballot_template_name(input_type)
        if template_name is None:
            return HttpResponseServerError("the input type " + input_type
                + " is not supported")
        return render_to_response(template_name,
                        {"poll": poll,
                        'voter_uuid':voter_uuid,
                        'ballot_data':ballot_data},
                        RequestContext(request))
```

With the recursive attribute set to *True* a poll can run periodically, every Monday from 10 am to 6 pm for instance. Pnyx simply repeats the timeframe according a period specified during the poll creation. In practice the back-end updates the timeframe to the next one when the first user looks at the final result of the previous timeframe.

Code 5: Implementation of the periodic polls in vote/views.py

```

def results(request, pk, voter_uuid):
        poll = get_object_or_404(Poll, pk=pk)
        ...
        #check if the poll is recursive
        if not poll.recursive_poll:
            ...
        else:
            if (timezone.now() < poll.closing_date and timezone.now() >
                poll.opening_date)\
               or (timezone.now() < poll.opening_date and
                   poll.alternative_set.all()[0].final_rank != None):
               #return the result are not available
```

```
                return HttpResponseRedirect(
                        reverse('vote:poll_not_closed_yet', kwargs =
                            {'pk': pk, 'voter_uuid': voter_uuid}))
            else:
                if timezone.now() > poll.closing_date:
                #update the timeframe
                    poll.closing_date = poll.closing_date +
                        datetime.timedelta(days = poll.recursive_period)
                    poll.opening_date = poll.opening_date +
                                datetime.timedelta(days =
                                    poll.recursive_period)
                    poll.save()
                    logger.debug("Time frame of the repeated poll " +
                        str(poll.pk) + " updated")
        return compute_and_display_results(request, poll, True)
```

As you can see in the previous extract of code, one can access the result only after the first iteration and when the poll is not running. Unlike basic polls, periodic polls require to compute the result every time a user consult the *result* page.

4.4 Implementation of admin features (polls app)

4.4.1 Handle a simple form: Email participant

Pnyx user interface allows the poll administrator to email the participants at once. To do so, the user has simply to fill a form and submit it, the back-end extracts the subject, the message and sends the email to the participants. More generally, forms are used each time the user needs to parse some data to the application especially to interact with the database. Implementing form processing from scratch results to a lot of repeated code. If the workflow is as simple as this use case, we can use the built-in solution *FormView* from the *django.views.generic* module. Here are main steps to implement this feature.

First we need to create the form object from the *dajngo.forms* module. The inherited method *is_valid()* specifies the rules to validate the form and the error messages to display in case of invalidity. By default django checks if some required input is lacking.

Code 6: a simple form implementation in polls/models.py

```
class EmailParticipantForm(forms.Form):
 subject = forms.CharField()
 message = forms.CharField(widget = forms.Textarea)

 def is_valid(self):
        valid = super(SetUpUpcomingPollForm, self).is_valid()
        ... #add customized rules
        return valid
```

Then we create a subclass of a *from view* which is mapped to a url, a template and a form. In *form_vaild()* we specify what to do once the form is valid and submitted.

Code 7: a simple form implementation in polls/views.py

```
class EmailParticipant(generic.FormView):
        template_name = 'polls/email_participant.html'
        form_class = EmailParticipantForm

        def form_valid(self, form):
  # This method is called when valid form data has been POSTed.
  # It should return an HttpResponse.
         participant_list = list(get_object_or_404(Poll, pk =
              self.kwargs['pk']).participant.all())
  email_list = ()
         for participant in participant_list:
          email_list += (participant.email),
  if email_list:
   #notify the participants
   send_mail(form.cleaned_data['subject'],
                form.cleaned_data['message'],
                self.request.user.email,
                email_list,
                fail_silently = False)
  return HttpResponseRedirect(reverse('polls:detail', kwargs =
       self.kwargs))
```

From the user perspective the process is straightforward. First, one gets an empty form by loading the *email participant* page, then fills the form up and submits it. The back-end calls *is_ valid()*. If the form is invalid, the user is redirected to the form page with the errors notified. Otherwise the back-end calls *form_ valid()*.

The emails are sent and the user is redirected to a confirmation page. Sending emails with Django is relatively easy, we just need to set up the functionality in the settings file and to call the method *send_ email()* from the *django.core.mail* module. Here is how looks like the settings:

Code 8: Settings to send emails in pnyx/settings.py

```
#Email settings
EMAIL_HOST = 'smtp.gmail.com'
EMAIL_PORT = "587"
EMAIL_HOST_USER = 'email@gmail.com'
EMAIL_HOST_PASSWORD = 'password'
EMAIL_USE_TLS = True

DEFAULT_FROM_EMAIL = EMAIL_HOST_USER
SERVER_EMAIL = EMAIL_HOST_USER
```

4.4.2 Handle a multipage form: Create a poll

Poll creation is mostly reduces to data rendering and the related form is lengthy. For a smoother user experience, we split the form in 5 web pages using a *form wizard*: the general settings, the input type, the output type, the alternatives definition and the participants specification. To do so, Django also provides a built-in feature: the *SessionWizardView* in the *django.contrib.formtools.wizard.views* module and works as follows:

From the user perspective, once he is on the *create a poll* page, the user fills and submits the 5 forms step by step. After each submission, the form is validated with *is_ valid()* and after the last form, the user see a preview of poll is about to create, he can either confirm the creation or navigate backward to make some changes.

From the back-end side, it works slightly differently. The app maintains the state on the back-end side so the processing can be delayed until the final submission. Here is how we implemented such the wizard. First of all, we had to define a form for each step as a subclass of *django.forms* and its validation rules. Then we created a *WizardView* subclass that wraps all the forms together and specifies what to do after the final submission in the method *done()*.

Code 9: a form wizard implementation in polls/views.py

```
class CreatePollWizardView(SessionWizardView):

 form_list = [list of the forms]

 def get_template_names(self):
  return [TEMPLATES[self.steps.current]]

 def get_form_initial(self, step):
  ...

 def get_context_data(self, form, **kwargs):
  context = super(CreatePollWizardView, self).get_context_data(form =
      form, **kwargs)
                ...
               return context

 def done(self, form_list, **kwargs):
  ...
  return HttpResponseRedirect(reverse('polls:create_poll_confirmation',
      kwargs={'pk': poll.pk}))
```

Finally to make the form reachable, we mapped the *CreatePollWizardView.as_ view()* method to a url.

Code 10: Mapping of the url of a form wizard in polls/urls.py

```
url(r'^poll/add/$', views.CreatePollWizardView.as_view(),
    name='create_poll'),
```

The last requirement to use form wizard is to add *django.contrib.formtools* in the INSTALLED_APPS in the settings file.

Code 11: Sentings for form wizard in pnyx/settings.py

```
INSTALLED_APPS = (
 ...
 'django.contrib.formtools', #for form wizard
 )
```

4.4.3 Handle a form manually: Change poll settings

Every poll has a page where its administrator can modify its attributes. This form evolves during the life cycle of the poll such that, depending on the status of the poll (running, closed or upcoming), the admin is not able to change the same parameters. For closed polls, one can only change the name and the description. For a running poll, the admin can change the name, the description but also add new voters, extend the timeframe or change the visibility of the temporary results. If the poll is not opened yet, one can change every attribute of the poll except its periodicity. To handle the update process, we couldn't use built-in forms like in the poll creation. The dynamic behavior depending on the status of the poll (upcoming, running or closed) is relatively specific and not supported by Django's built-in features. Instead, we used a basic view. It is simply a method that takes a Web request and returns a Web response.

Strictly speaking the view is the method *change_settings_view()* but the logic is handled by inner methods. The goal of *change_settings_view()* is to realize the dynamic rooting and to redirect to the appropriate form and template. The form processing takes place in the inner methods.

Code 12: change_setting implementation in polls/views.py

```
@login_required
def change_settings_view(request, pk, *args, **kwargs):

 poll = Poll.objects.get(pk = pk)
        if not poll.admin == request.user:
         logger.debug("user and admin are different: no right to see the
             details")
         return HttpResponseRedirect(reverse('polls:no_right'))
        elif poll.opening_date >= timezone.now():
  #the poll is not opened yet
               return set_upcoming_poll(poll, request, pk)
        elif poll.closing_date <= timezone.now():
  #the poll is closed
               return set_closed_poll(poll, request, pk)
        else:
  #the poll is opened
  return set_opened_poll(poll, request, pk)
```

Code 13: Declaration of the url for the change_settings page in polls/urls.py

```
url(r'^mypolls/(?P<pk>\d+)/settings/$', views.change_settings_view,
    name='setup'),
```

Here is an example of the inner method called when the poll is closed. It is relatively simple, the form being limited to only 2 text fields.

Code 14: manual implementation of a form in polls/views.py

```
def set_closed_poll(poll, request, pk):
        if request.method == 'POST':
             set_up_poll_form = SetUpClosedPollForm(request.POST)
             if set_up_poll_form.is_valid():
```

```
            # update if the value changed
                if poll.name !=
                    set_up_poll_form.cleaned_data['poll_name']:
                    poll.name =
                        set_up_poll_form.cleaned_data['poll_name']
                if poll.description !=
                    set_up_poll_form.cleaned_data['poll_description']:
                    poll.description =
                        set_up_poll_form.cleaned_data['poll_description']
            poll.save()
            logger.debug("Closed poll " + str(pk) + " updated")
            return
                HttpResponseRedirect(reverse('polls:update_confirmation',
                kwargs = {'pk': pk}))

        else:
            data = {'poll_name': poll.name, 'poll_description':
                poll.description,}
            set_up_poll_form = SetUpClosedPollForm(data)
            return render(request,
                'polls/change_closed_poll_setting.html', {
                     'set_up_poll_form': set_up_poll_form, 'poll_pk':
                         pk})
```

In this example we can see that when the user accesses the web page for the first time (GET request), a form containing the name and the description of the poll is created. Once the form is submitted (POST request), we create a form object from the submitted form. If the form is valid, the changes are saved in the database and the user is redirected to a confirmation page. Else the user is redirected to the form page with notifications about the errors. The other inner methods share the same logic.

4.4.4 Generic views: informative pages

In addition to the editing pages, the poll administrator can access 3 types of informative pages inherited from the *django.views.generic* module. The simplest one is a page where the text is static and independent from the context (like *no authorization* page). Django provides a solution to handle the simple behavior: the *TemplateView*. All we need to do is to create a subclass of the *Template View*, attaches a template to it and map the classed based view to a url.

Code 15: a TemplateView implementation in polls/views.py

```
class NoRightView(generic.TemplateView):
        template_name = 'polls/no_right.html'
```

Code 16: declaration of the url of a TemplateView in polls/urls.py

```
url(r'^forbidden/$', views.NoRightView.as_view(), name = 'no_right'),
```

A slightly more complex but usual case is to print detailed information about a particular object, detailing the attributes of a poll for instance. The content depends now on the url. Django provides a solution to reduce the coding: the *generic detail view*. We need to create a subclass of the detail view, to attache a template to it and to map the view to a url.

Django will extract the *pk* attribute from the url and get the poll from the database with the corresponding *primary key*. It is important that the attributes of the regular expression in the url and of the object attribute from the database share the same name.

Code 17: a DetailView implementation in polls/views.py

```
class DetailView(generic.DetailView):

 model = Poll
 template_name = 'polls/detail.html'

 def get_context_data(self, **kwargs):
  context = super(DetailView, self).get_context_data(**kwargs)
  ...
  return context
```

Code 18: declaration of the url of a DetailView in polls/urls.py

```
url(r'^mypolls/(?P<pk>\d+)/$', views.DetailView.as_view(),
    name='detail'),
```

Pnyx uses also a *Detail view* to extract and print the preference profile. We use this to extract easily the related poll. Then in the *get_ context_ data* method, we extract all the preferences related to the poll and process them to get the preference profile under the PrefLib standards.

The last use case where we use informative pages is when we need to display a list of items like in the *my polls* page. To handle all the database interactions, Django comes up with the generic *List View*. Similarly to the *Detail View*, we need to map a url to the view and to specify a template. The url does not characterize the query anymore, it is now specified in the *get_ queryset()* method.

Code 19: a ListView implementation in polls/views.py

```
class ManagePollView(generic.ListView):
 template_name = 'polls/manage_poll.html'
 context_object_name = 'current_poll_list' #name of the list in the
     template view

 def get_queryset(self):
  #return the running polls
  return Poll.objects.filter(
   admin = self.request.user
   ).filter(
    opening_date__lte= timezone.now()
   ).filter(
    closing_date__gte = timezone.now()
   ).order_by(
    '-creation_date'
   )
```

```
def get_context_data(self, **kwargs):
 context = super(ManagePollView, self).get_context_data(**kwargs)
              ... #add closed and upcoming polls in the context
              return context
```

Code 20: declaration of the url of a ListView in polls/urls.py

```
url(r'^mypolls/$', views.ManagePollView.as_view(), name='manage_poll'),
```

4.5 Implementation of voter features

This subsection deals with the functionalities related to the voting process regarding the voters and the polls.

4.5.1 Workflow to display the ballot page

The processing of the ballot view is very similar to the update poll view. It is also a basic view with a dynamic behavior. There are 5 types of ballot for the 5 supported individual preferences but only one url. However the post processing is slightly more complex. Here is global picture of the workflow when a voter access the ballot and then submit his vote.

First the method *get_ ballot_ view():* is called. The different verifications (visibility, vote and timeframe restrictions) are performed and the ballot is displayed. If necessary the data of the previous vote is loaded into the template in a dictionary called *data_ ballot*.

Once the voter submits his ballot, the view *vote()* is called. The back-end performs some verifications and then processes the data with the appropriate inner method *process_ vote_ [input_ type]*. In this method the expressed preferences are saved in database by creating the corresponding *Binary Relation* instances if the individual preference type is a complete relation or *Transitive Preference* instances otherwise. Then, the voter is redirected to a confirmation page displaying the link of the final results and if necessary the link of the temporary results.

Before detailing the implementation of each ballot, here is a bird's-eye view of the ballot implantation. In *HTTP* programming, the data exchange between the user and the server is contained in *HTML input*, the design of such inputs is standardized and the interaction limited. To make the ballot more user-friendly and intuitive, we used the *jQuery User Interface* javascript library on top of these inputs. The voter is not directly interacting with *html inputs* but with *selectable* or *sortable* objects. A Javascript layer based on *jQuery* catches the interactions and creates the corresponding hidden inputs. These inputs will de sent back at the submission of the form. The drawback of this design is that *jQuery UI* is not supported by mobile devices by default. To overcome this difficulty we use the javascript library *jQuery UI Touch Punch* It is a small but unofficial hack that enables the use of touch events on web sites using the jQuery UI.

Complete binary relation

This ballot is the most general ballot and has the most complex implementation. We create all pairwise comparisons between alternatives. For each comparison, we have one sortable element that can be dragged and dropped in 3 different buckets; one for each possible preference xPy, xIy or yPx. To link the 3 buckets together and to create the hidden inputs, each comparison is associated wit a javascript function.

Code 21: Template code for binary relations ballot in templates/ballot_complete_binary_relation.html

```
<div class="col-md-1 connectedSortable_{{ pk1_pk2 }} sortable"
    id="sortable_{{ pk1_pk2 }}_1">
 {% if ballot_data|keyvalue:pk1_pk2 == pk1 %}
 ... # 1st bucket
</div>
<div class="col-md-1 connectedSortable_{{ pk1_pk2 }} sortable"
    id="sortable_{{ pk1_pk2 }}_2"> # 2nd bucket
 {% if not ballot_data|keyvalue:pk1_pk2 %}
 <div class="raw ui-state-default" id="{{ pk1_pk2 }}">
  <div class="col-md-12">
   <p><span class="glyphicon glyphicon-resize-horizontal"></span></p>
  </div>
 </div>
 {% endif %}
</div>
<div class="col-md-1 connectedSortable_{{ pk1_pk2 }} sortable"
    id="sortable_{{ pk1_pk2 }}_3">
 {% if ballot_data|keyvalue:pk1_pk2 == pk2 %}
 ... # 3rd bucket

<div id="hidden-choice">
 ...
 {# contains all hidden alternative input for each alternative selected
     #}
 {% if ballot_data %}
  {# init the ballot with the previous vote inputs #}
 {% else %}
  {# init the ballot with indifference #}
  {% for alternative_1 in poll.alternative_set.all %}
   {% for alternative_2 in poll.alternative_set.all %}
    {% if alternative_1.pk < alternative_2.pk %}
     <input type="hidden"
      id = "id_comparison_{{ alternative_1.pk }}_{{ alternative_2.pk }}"
      name = "comparison_{{ alternative_1.pk }}_{{ alternative_2.pk }}"
      value="-1">
     {% endif %}
   {% endfor %}
  {% endfor %}
</div>
...
```

The javascript code is very similar to the example of the next ballot, please refer to it for more details.

Transitive complete preferences

In this ballot the user specifies his preferences using *sortable* elements. He is supposed to drag and drop each alternative in the corresponding bucket. There is one bucket for each rank and within the same bucket we assumed a complete indifference. One extra bucket containing the alternative to rank is used to initiate the ballot.

Code 22: Template code for complete and transitive preferences in templates/ballot_complete_preorder.html

```
<form class="form-horizontal" role = "form" action="{% url 'vote:vote'
    poll.id voter_uuid %}" method="post">
...
 {% for alternative in poll.alternative_set.all %}
  <p><b>Rank {{ forloop.counter }} </b></p>
  <div class="container connectedSortable sortable" id="sortable_{{
      forloop.counter }}">
  {% if ballot_data %}
   ... #load the data
  {% endif %}
  </div>
 {% endfor %}
 {# init all alternatives in a "to rank" bucket #}
 <p><b>To rank </b></p>
  <div class="container connectedSortable sortable"
      id="sortable_to_rank">
  {% if not ballot_data %}
  {% for alternative in poll.alternative_set.all %}
   <div class="row ui-state-default" id="{{ alternative.pk }}">
    <div class="col-sm-7 ui-widget-content">
     <p><span class="glyphicon glyphicon-resize-vertical"></span>
         <strong>{{ alternative.name }}...</p>
    </div>
   </div>
  {% endfor %}
  {% endif %}
  </div>
 </div>
 <div id="hidden-choice">
  {# contains all hidden alternative input for each alternative selected
      #}
  {% if ballot_data %}
  ... create the hidden input data
  {% endif %}
 ...
```

As you can see in the template code, there is one javascript method to linked all the buckets together and to create the hidden inputs every time the ballot is updated, (i.e. every time an alternative is dropped).

Code 23: JavaScript code for complete and transitive preference of the ballot templates/ballot_complete_preorder.html

```
$(function() {
 {% for alternative in poll.alternative_set.all %}
  $( "#sortable_{{ forloop.counter }}" ).sortable({
   connectWith: ".connectedSortable",
   deactivate: function() {
    $( "#hidden-choice > input[name='choice_{{ forloop.counter
        }}']").remove( );
    $( ".ui-state-default", this ).each(function() {
     var value = $(this).attr('id') ;
     if (value != null ) {
      $('<input>').attr({
       type: 'hidden',
       name: 'choice_{{ forloop.counter }}' ,
       value: value
      }).appendTo('#hidden-choice');
     }
    });
   }
  });
  $( "#sortable_{{ forloop.counter }}" ).disableSelection();
 {% endfor %}

 $( "#sortable_to_rank" ).sortable({
  connectWith: ".connectedSortable",
 });
 $( "#sortable_to_rank" ).disableSelection();
});
```

Linear order of preference

Linear orders prevent the voter to express indifferences. Therefore buckets are not useful anymore, the individual preference is expressed by ordering vertically the alternatives. To do so, the voter drags and drops sortable alternatives and the implementation is very similar to the previous ballot so the code is not detailed here.

Dichotomous preferences

Dichotomous ballot differs from the previous one by its implementation. It deals with a different object from the *jQuery UI* library: the *selectable*.

Code 24: Template code for dichotomous preferences in templates/ballot_dichotomous.html

```
...
<div class="container" id="selectable">
 {% for alternative in poll.alternative_set.all %}
  <div class="row" id="{{ alternative.pk }}">
   <div class="col-sm-7 ui-widget-content" >
    <p><strong>{{ alternative.name }} ...</strong></p>
   </div>
  </div>
 {% endfor %}
</div>
...
<div id="hidden-choice">
 {# Will contain all hidden input for each alternative selected #}
  {% for alternative in ballot_data.1 %}
   <input type="hidden" name="choice" value="{{ alternative.pk }}">
  {% endfor %}
 </div>
</div>
...
```

The voter has simply to select the alternatives he would like to approve. A javascript method creates/erases the corresponding hidden input every time an alternative is selected / unselected. If the voter is revoting, a init function is run at first to reload the ballot.

Code 25: JavaScript code for dichotomous preferences in templates/ballot_dichotomous.html.html

```
...
$(function() {
           //init the form from the hidden input
           $( "input[name='choice']" , this ).each(function() {
               var pk = $(this).attr('value') ;
               $( "#"+pk).addClass("ui-selected");
           });

           $( "#selectable" ).bind( "mousedown", function ( e ) {
               e.metaKey = true;
           }).selectable({
               //load the selection in the input
               stop: function() {
                   $( "#hidden-choice > input").remove( );
                   $( ".ui-selected", this ).each(function() {
                       var value = $(this).attr('id') ;
                       if ( value != null ) {
                           $('<input>').attr({
                               type: 'hidden',
                               name: 'choice',
                               value: value
                           }).appendTo('#hidden-choice');
                       }
                   });
               }
           });
       });
```

Dichotomous preferences with a unique best alternative

The last kind of ballot is very similar to dichotomous profile. There is only one add-on to prevent the voter to select severals alternatives. We added a javascript function that does the following: every time an alternative is selected, it unselects all the other alternatives.

Code 26: JavaScript code for unique winner ballot in templates/ballot_most_preferred_alternative.html

```
...
$(function() {
            //init the form from the hidden input
            $( "input[name='choice']" , this ).each(function() {
                var pk = $(this).attr('value') ;
                $( "#"+pk).addClass("ui-selected");
            });
            $("#selectable").selectable({
                selected: function(event, ui) { // prevent to select
                     several alternative
                    $(ui.selected).addClass("ui-selected").siblings().removeClass("ui-selected");
                },
                stop: function() {
                    $( "#hidden-choice > input").remove( );
                    $( ".ui-selected", this ).each(function() {
                        var value = $(this).attr('id') ;
                        if ( value != null) {
                            $('<input>').attr({
                                type: 'hidden',
                                name: 'choice',
                                value: value
                            }).appendTo('#hidden-choice');
                        }
                    });
                }
            });
        });
...
```

4.5.2 Workflow to display the results pages

Another important page of Pnyx is the *results page* where the final or temporary results are published. Except the computation of the aggregated preference profile the result processing is relatively basics, mostly because there is not interaction with the user. Temporary results and final results are reachable from two different urls, but the processing and the workflow are similar and the templates shared. There is one template for each kind of aggregated preference *(results_best_alternative.html, results_lottery.html, results_ranking.html)*. The code is very simple and does not contain javascript.

Here is global picture of the workflow when a voter access the final results of a poll. First the view method *results()* is called. Timeframe and visibility verifications described above are performed. Then the back-end checks if the final results have already been computed. If so the method *display_results()* is called. It basically extracts the final results from the database, and display it to the user. Otherwise, the method *compute_and_display_results()* is called with the parameter *save_ result* to *True*. The appropriate choice rule is performed, then in *process_ result_ [output_ type]*, the computed result is processed and the final results of the poll are saved in the database by writing in each alternative its rank or its lottery score. Then the final results are displayed.

Code 27: Mapping of urls for the results pages in vote/urls.py

```
url(r'^(?P<pk>\d+)/(?P<voter_uuid>[a-z0-9\-]+)/temp-results/$',
    views.temporary_results, name = 'temp_results'),
url(r'^(?P<pk>\d+)/(?P<voter_uuid>[a-z0-9\-]+)/results/$',
    views.results, name = 'results'),
```

If the voter requests the temporary results, the workflow is relatively similar. The few differences are :

- the view method called is *temporary_ results()*
- *compute_ and_ display_ results()* is called with the parameter *save_ result* to *False* so that the results are not saved in the database.

4.5.3 How the voting rules are implemented

The aggregation is performed by an inner method specific for each rule. It is called in the *compute_ and_ display_ results()* which does a dynamic rooting. We will not detail all choice rules but only the most interesting implementations. The *Plurality* (and *approval voting*, *plurality score*), *Borda's score*, *Borda's extension to preorders*, *Young's scoring rule* and *random dictatorship* are relatively straightforward and based on basics python programming and *NumPy*. For this rules the tie break is well implemented in the sense that the back-end can really detect when it is necessary to break ties, that it is performed according the lexicographic order chosen by the voter. On the other hand, *PuLP* and *CvxOpt* act as a black box and Pnyx back-end is not able to check if the optimization problems induces a tie break, the status is *UNKNOWN* in the database. The *tie_ breaking_ used* is updated in consequence and the ballot mentions if and how a tie break is performed.

Kemeny's rule & Mixed Integer Programing

As explained in section 2, the Kemeny's rule is computed by MIP. Here is the implementation of the rule based on *PuLP*.

Code 28: Mixed Integer Programing and Kemeny's rule implementation in vote/views.py

```
def kemeny(request, poll, penalty_weights, index_array, save_result):
 ...
 n_alternatives = penalty_weights.shape[0]
 # The prob variable is created to contain the problem data
 prob = pulp.LpProblem("Kemeny Problem", pulp.LpMinimize)
 # The variable are created
 Sequence = ["{0:1d}".format(x) for x in range(n_alternatives)] #convert
     range() to a list of string
```

```
edge = pulp.LpVariable.dicts("Edge", (Sequence, Sequence), 0, 1,
    pulp.LpInteger)
# The objective function is added to 'prob'
prob += pulp.lpSum([edge[r][c] * penalty_weights[r, c] for r in
    Sequence for c in Sequence]), " weight of the graph"
# Creation of the constrain
for i in Sequence:
       prob += edge[i][i] == 0, "No loop of size 1:: " + i
# constraints for every pair
for i, j in combinations(range(n_alternatives), 2):
              prob += edge[str(i)][str(j)] + edge[str(j)][str(i)] == 1,
                  "No loop of size 2: " + str(i) + "<=>" + str(j)
# and for every cycle of length 3
for i, j, k in combinations(range(n_alternatives), 3):
 prob += edge[str(i)][str(j)] + edge[str(j)][str(k)] +
     edge[str(k)][str(i)] >= 1, "No loop of size 3: (" + str(
   i) + "=>" + str(j) + "=>" + str(k) + "=>" + str(i) + ")"
 prob += edge[str(i)][str(k)] + edge[str(k)][str(j)] +
     edge[str(j)][str(i)] >= 1, "No loop of size 3: (" + str(
   i) + "=>" + str(k) + "=>" + str(j) + "=>" + str(i) + ")"
# The problem is solved using puLP's choice of Solver
prob.solve()
       ... #the result is processed
```

Using an LP solver raises the performance of the computation but the major drawback is that tie breaking is not controlled by Pnyx anymore. The LP solver returns one ranking that maximize the Kemeny score but we are not aware of the existence of other rankings that would also have a maximal score. Thus we cannot perform tie break and the voter is informed in the result page.

Maximal lottery & Linear Programing

We mentioned in section 2, that to approximate the barycenter of Maximal lotteries, we compute one ML for each alternative. In practice the method *maximal_lottery()* calls the method *sub_maximal_lottery* to resolve the each LP sub problem linked to a given alternative. We also use *puLP* and *GLPK*. Here is the implementation of an LP sub-problem.

Code 29: Linear Programing and Maximal Lottery implementation in vote/views.py

```
def sub_maximal_lottery( objective_index, payoff_matrix,
    alternative_pk_array):
 #the maximal lottery are mixed maximin strategies of the plurality game
 n_alternatives = payoff_matrix.shape[0]
 # The prob variable is created to contain the problem data
 prob = pulp.LpProblem("Maximal Lottery Problem", pulp.LpMaximize)
 # The variable are created
 Sequence = ["{0:1d}".format(x) for x in range(n_alternatives)] #convert
     range() to a list of string
 p = pulp.LpVariable.dicts("P", Sequence, lowBound = 0)
 # The objective function is added to 'prob' first
 prob += p[str(objective_index)], "maximize the probality for
```

```
    alternative " + str(objective_index)
 # Creation of the constrain
prob += pulp.lpSum([p[i] for i in Sequence]) == 1 , "Propability
    distribution"
 # Utility greater than the security level in case of every pure
     strategy of the opponent
for j in range(n_alternatives):
 prob += pulp.lpSum([p[str(i)]*payoff_matrix[i,j] for i in
     range(n_alternatives)]) >= 0 , \
  "utility >= security level if pure strategy " + str(j)
 # The problem is solved using PuLP's choice of Solver
prob.solve()
 # Create the lottery object and return
      ...
```

It worths to mention that *strict maximal lotteries*, a refinement of maximal lotteries, are implemented but they are not used.

Nash's solution & convex optimization

Nash's solution uses a non linear convex optimization. It is computed by the *CvxOpt* package for convex optimization based on Python's extensive standard library, (the *NumPy* module is required). CvxOpt is a free software under the GNU General Public License. Here is the implementation of the optimization problem.

Code 30: Convex optimization and Nash's Solution implementation in vote/views.py

```
def nash(request, poll, utilitaian_matrix, alternative_pk_array,
    save_result):
 # Settup of the optimizer
 cvxopt.solvers.options['show_progress'] = False
 cvxopt.solvers.options['maxiters'] = 500

 def F(x = None, z = None):
  if x is None:
   return 0, cvxopt.matrix(1./n_alternatives, (n_alternatives,1))
  if min(x) <= 0.0:
         return None
  f = -sum(np.log(np.dot(utilitaian_matrix.T,x)))
   #grad(f) = U^T *(1/(U_1*x),..,1/(U_N)*x))
  Df =
      -cvxopt.matrix(np.dot(utilitaian_matrix,(np.dot(utilitaian_matrix.T,x)
      ** -1)).T)
        if z is None:
         return f, Df
  H = cvxopt.matrix(0., (n_alternatives,n_alternatives))
        for i in range(0, n_alternatives):
               for j in range(0,n_alternatives):
                      if i>=j:
     U_ijUjn =
         np.matrix([utilitaian_matrix[i,n]*utilitaian_matrix[j,n]
         for n in range(0, n_voter)])
     H[i,j] = np.dot(U_ijUjn, (np.dot(utilitaian_matrix.T, x) **
         -2))
  return f, Df, H

 #constrains to define a distribution
 G = -cvxopt.matrix(np.identity(n_alternatives))
 h = cvxopt.matrix(0.,(n_alternatives,1))
 A = cvxopt.matrix(1.,(1, n_alternatives))
 b = cvxopt.matrix(1.,(1,1))

 sol = cvxopt.solvers.cp(F, A = A, b = b, G=G, h=h)
 opt_value = sol['x']
 ...
```

The argument F is a function that evaluates the objective and nonlinear constraint functions. CvxOpt's documentation explains how to define this method.

4.6 The implementation of side features

Pnyx implements other general features related to the accounts management. These are functionalities are provided by *Django* authentication system in the module *django.contrib.auth.* A user can create an account, change his password on the platform, or ask to reset it. In that case, he will receive an email containing a token link where he will be able to enter a new password. More details can be found in *Django* doc.

4.7 Deployment with Apache and Wsgi

Pnyx is currently deployed on an *Apache* 2 server[17]. The current solution used a linux virtual machine as a web server. We briefly detaile the few steps to perform to deploy the application. For more information please see *Django project*'s official page or the *Apache* 2 documentation. If you don't have experiences with apache server, here is a tutorial that explains how to setup the *Django* server http://youtu.be/hBMVVruB9Vs

Before deploying Pnyx please check the following:

- The project folder is in the */var/www* folder
- The DEBUG mode in the setting file, is turned off in production mode.
- If the database is as *SQLite* database, the project folder and the file *db.sqllite3* should have both the writing rights. You can see it by running the command *ls -al* and change it with the command *chmod.* Once you are in the appropriate directory run *sudo chmod 777 db.sqllite3* for instance.

If you want to edit a file from the virtual machine, you can use the command-line editor Vim [18]. Please note that to modify any file in the */var/www/* directory you need super user rights. To have it, you need to add *sudo* before any command in the command-line (like in the example above).

References

[1] Gunther Charwat and Andreas Pfandler: *DEMOCRATIX: A Declarative Approach to Winner Determination*, Fifth International Workshop on Computational Social Choice (COMSOC 2014), democratix.dbai.tuwien.ac.at/

[2] Haris Aziz , Felix Brandt and Markus Brill: *On the tradeoff between economic efficiency and strategyproofness in randomized social choice*, In Proceedings of the 12th International Joint Conference on Autonomous Agents and MultiAgent Systems. (AAMAS). IFAAMAS, 2013. Forthcoming

[3] Anna Bogomolnaia, Herve Moulin and Richard Stong: *Collective choice under dichotomous preferences*, Journal of Economic Theory, Volume 122, Issue 2, June 2005, Pages 165-184

[4] J. J. Bartholdi III, C. A. Tovey, and M. A. Trick: *The computational difficulty of manipulating an election*, Social Choice and Welfare, 6:227?241, 1989

[5] John Cullinan , Samuel Hsiao and David Polett: *A Borda count for partially ordered ballots*, Social Choice and Welfare, vol. 42, issue 4, pages 913-926, 2014

[6] H.P. Young: *An Axiomatization of Borda's rule*, Journal of Economic Theory 9, 43-52, 1974

[7] MIT license information: opensource.org/licenses/MIT

[8] Python documentation : docs.python.org/2/

[9] Bootstraps tutorials: getbootstrap.com/getting-started

[10] PuLP official page: pypi.python.org/pypi/PuLP

[11] GNU Linear Programming Kit official page: gnu.org/software/glpk/

[12] NumPy official page: numpy.org

[13] CvxOpt official page :CvxOpt.org/

[14] CvxOpt documentation :CvxOpt.org/userguide/

[15] Django project's official tutorial: docs.djangoproject.com/en/1.6/

[16] Django project's official page: docs.djangoproject.com/en/1.6/intro/tutorial01/

[17] Apache HTTP sevrer project: httpd.apache.org

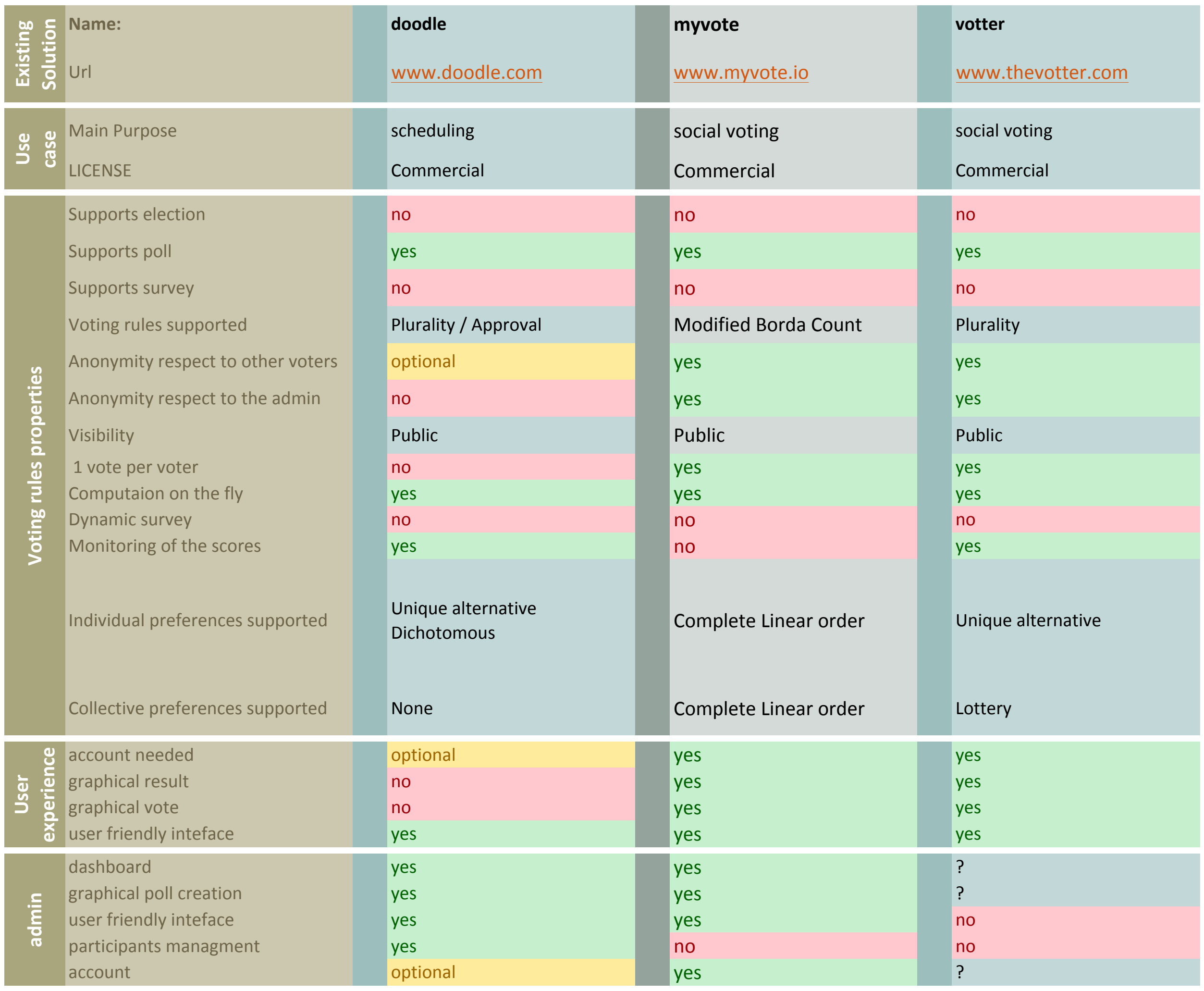
[18] Vim command line editor: www.vim.org

Appendix

A Benchmark of existing solutions

You will find here a table summerazing our benchmark over several tools related to preference aggregation that are available online, and for free.

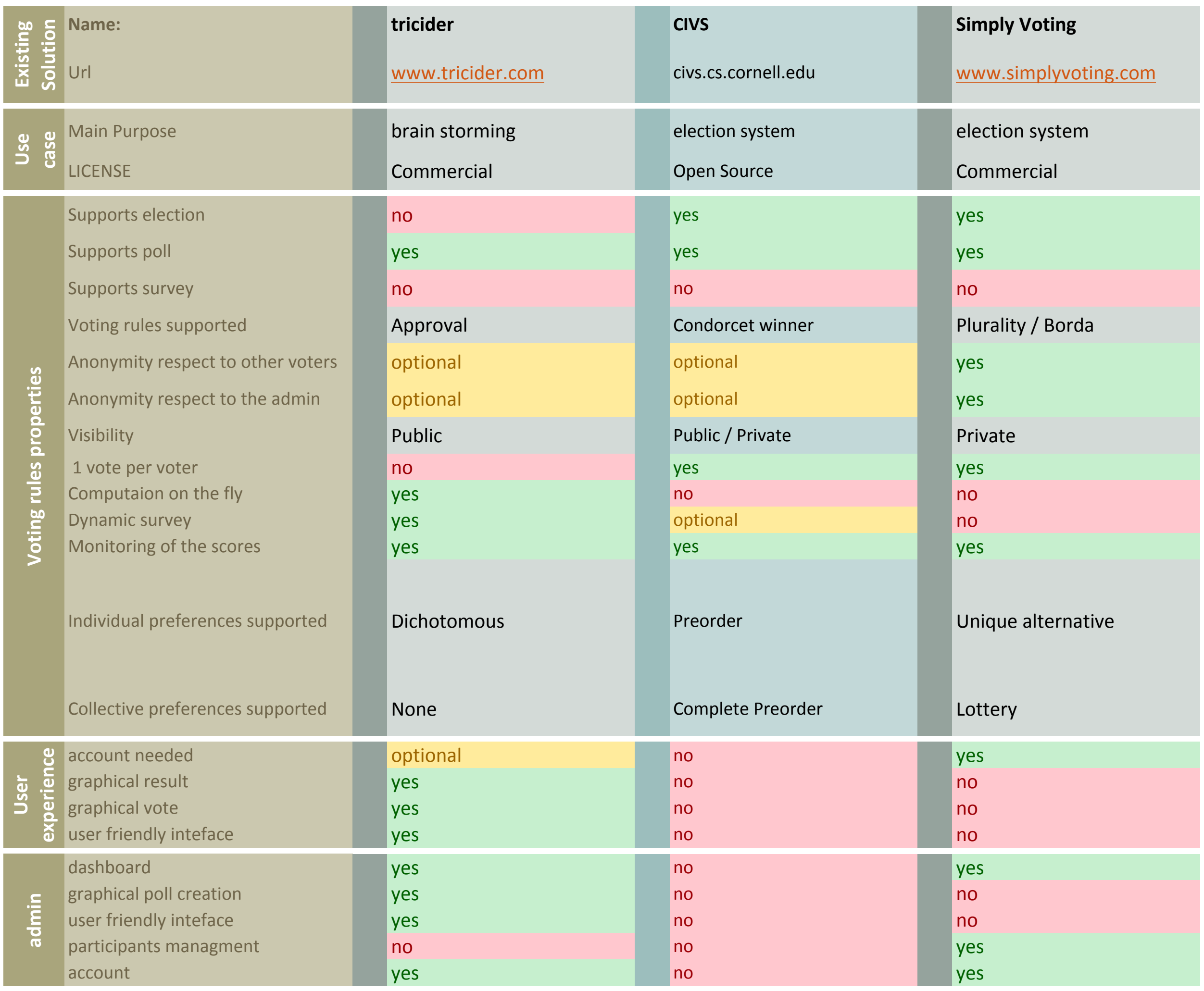
Figure 15: Detail of the benchmark of existing solutions



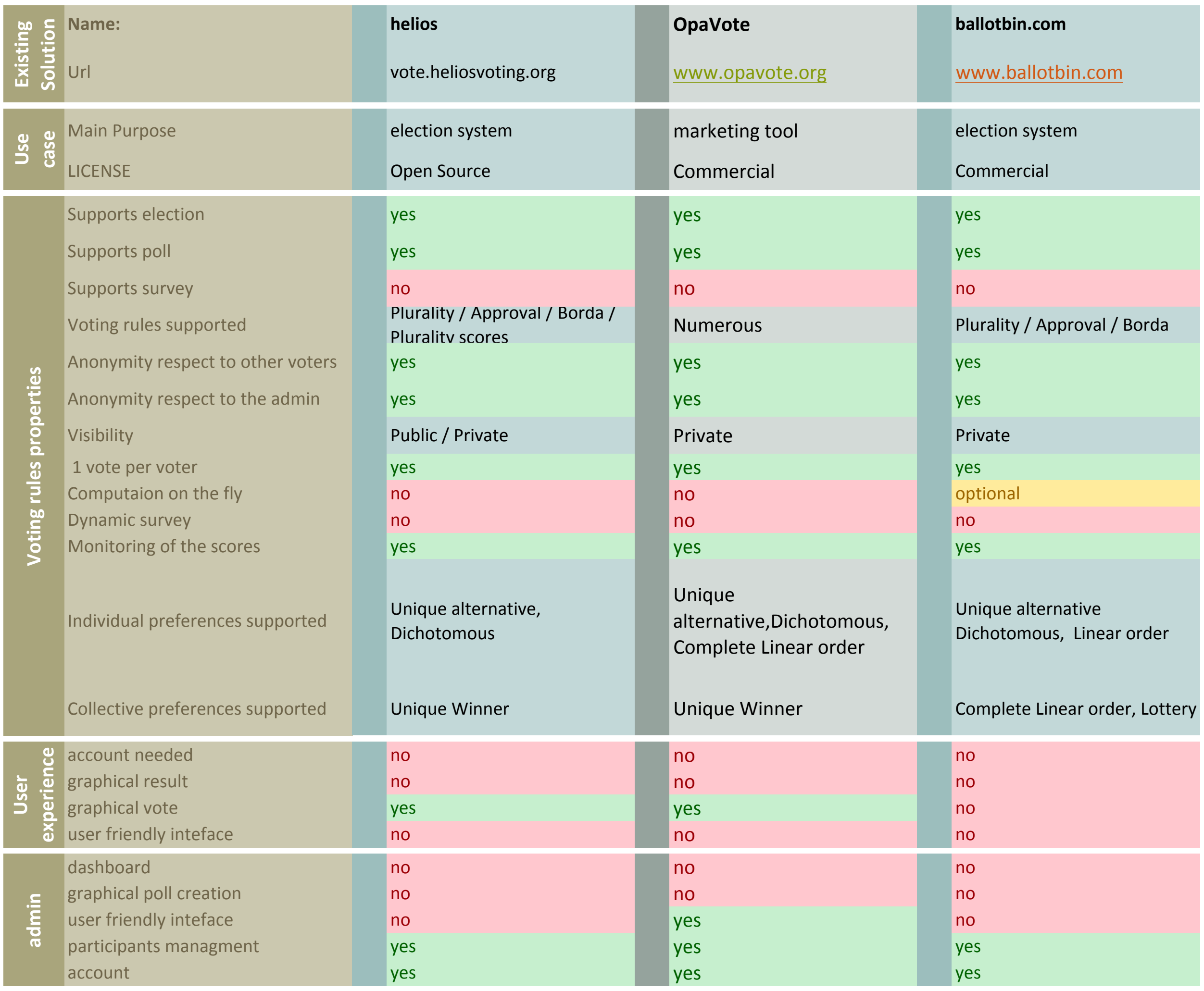
Existing Solution	Name:	doodle	myvote	votter
	Url	www.doodle.com	www.myvote.io	www.thevotter.com
Use case	Main Purpose	scheduling	social voting	social voting
	LICENSE	Commercial	Commercial	Commercial
Voting rules properties	Supports election	no	no	no
	Supports poll	yes	yes	yes
	Supports survey	no	no	no
	Voting rules supported	Plurality / Approval	Modified Borda Count	Plurality
	Anonymity respect to other voters	optional	yes	yes
	Anonymity respect to the admin	no	yes	yes
	Visibility	Public	Public	Public
	1 vote per voter	no	yes	yes
	Computaion on the fly	yes	yes	yes
	Dynamic survey	no	no	no
	Monitoring of the scores	yes	no	yes
	Individual preferences supported	Unique alternative Dichotomous	Complete Linear order	Unique alternative
	Collective preferences supported	None	Complete Linear order	Lottery
User experience	account needed	optional	yes	yes
	graphical result	no	yes	yes
	graphical vote	no	yes	yes
	user friendly inteface	yes	yes	yes
admin	dashboard	yes	yes	?
	graphical poll creation	yes	yes	?
	user friendly inteface	yes	yes	no
	participants managment	yes	no	no
	account	optional	yes	?

Existing Solution	**Name:**	**show of hands**	**wedgies**	**polldaddy**
	Url	www.showofhands.mobi	www.wedgies.com	www.polldaddy.com
Use case	Main Purpose	social voting	social voting	survey management
	LICENSE	Commercial	Commercial	Commercial
Voting rules properties	Supports election	no	no	no
	Supports poll	yes	yes	yes
	Supports survey	no	no	yes
	Voting rules supported	Plurality	Plurality	Plurality / Approval / Borda
	Anonymity respect to other voters	yes	yes	yes
	Anonymity respect to the admin	yes	yes	no
	Visibility	Public	Public	Public / Private
	1 vote per voter	yes	yes	optional
	Computaion on the fly	yes	yes	yes
	Dynamic survey	no	no	yes
	Monitoring of the scores	yes	yes	yes
	Individual preferences supported	Unique alternative	Unique alternative	Unique alternative, Complete Linear order
	Collective preferences supported	Lottery	Lottery	Lottery, Complete Linear order
User experience	account needed	yes	yes	optional
	graphical result	yes	yes	yes
	graphical vote	yes	yes	yes
	user friendly inteface	yes	yes	yes
admin	dashboard	yes	yes	yes
	graphical poll creation	no	yes	yes
	user friendly inteface	yes	yes	yes
	participants managment	no	no	yes
	account	yes	no	yes

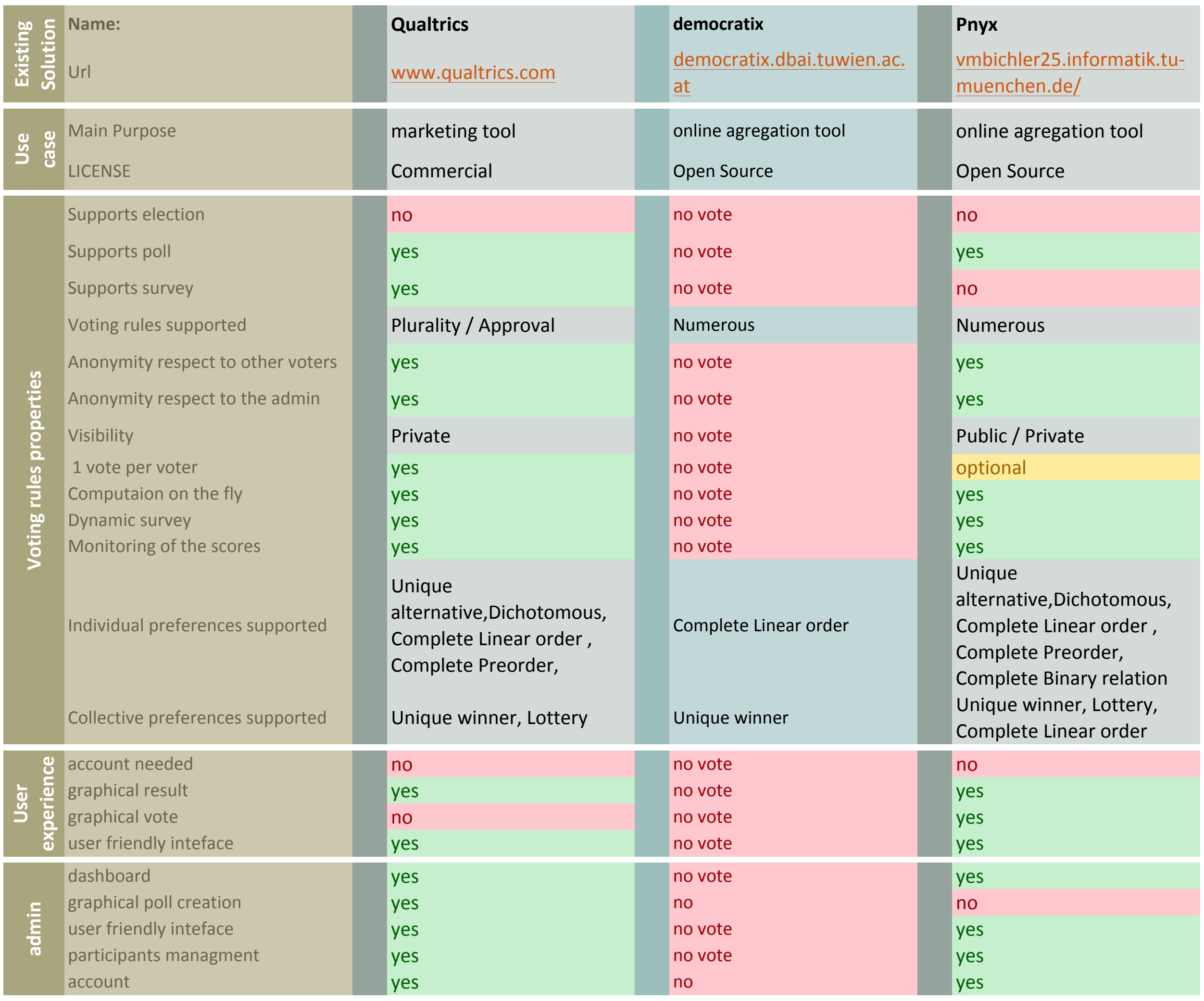
Existing Solution	**Name:**	**polarb**	**SurveyMonkey**	**limesurvey**
	Url	www.polarb.com	www.surveymonkey.com	www.limesurvey.org
Use case	Main Purpose	embed polls	survey management	marketing tool
	LICENSE	Commercial	Commercial	Open Source
Voting rules properties	Supports election	no	no	no
	Supports poll	yes	yes	yes
	Supports survey	no	yes	yes
	Voting rules supported	Plurality	Plurality / Approval	Plurality / Approval
	Anonymity respect to other voters	yes	yes	yes
	Anonymity respect to the admin	yes	yes	no
	Visibility	Public	Public	Public
	1 vote per voter	yes	no	no
	Computaion on the fly	yes	yes	yes
	Dynamic survey	no	yes	yes
	Monitoring of the scores	yes	yes	yes
	Individual preferences supported	Unique alternative	Unique alternative, Dichotomous, Complete Linear order	Unique alternative, Dichotomous, Complete Linear order
	Collective preferences supported	None	Lottery	Noene
User experience	account needed	no	no	no
	graphical result	yes	no	no
	graphical vote	yes	yes	yes
	user friendly inteface	yes	no	no
admin	dashboard	no	no	no
	graphical poll creation	yes	yes	yes
	user friendly inteface	yes	yes	no
	participants managment	no	yes	no
	account	yes	yes	yes



Existing Solution	**Name:**	**tricider**	**CIVS**	**Simply Voting**
	Url	www.tricider.com	civs.cs.cornell.edu	www.simplyvoting.com
Use case	Main Purpose	brain storming	election system	election system
	LICENSE	Commercial	Open Source	Commercial
Voting rules properties	Supports election	no	yes	yes
	Supports poll	yes	yes	yes
	Supports survey	no	no	no
	Voting rules supported	Approval	Condorcet winner	Plurality / Borda
	Anonymity respect to other voters	optional	optional	yes
	Anonymity respect to the admin	optional	optional	yes
	Visibility	Public	Public / Private	Private
	1 vote per voter	no	yes	yes
	Computaion on the fly	yes	no	no
	Dynamic survey	yes	optional	no
	Monitoring of the scores	yes	yes	yes
	Individual preferences supported	Dichotomous	Preorder	Unique alternative
	Collective preferences supported	None	Complete Preorder	Lottery
User experience	account needed	optional	no	yes
	graphical result	yes	no	no
	graphical vote	yes	no	no
	user friendly inteface	yes	no	no
admin	dashboard	yes	no	yes
	graphical poll creation	yes	no	no
	user friendly inteface	yes	no	no
	participants managment	no	no	yes
	account	yes	no	yes



Existing Solution	**Name:**	**helios**	**OpaVote**	**ballotbin.com**
	Url	vote.heliosvoting.org	www.opavote.org	www.ballotbin.com
Use case	Main Purpose	election system	marketing tool	election system
	LICENSE	Open Source	Commercial	Commercial
Voting rules properties	Supports election	yes	yes	yes
	Supports poll	yes	yes	yes
	Supports survey	no	no	no
	Voting rules supported	Plurality / Approval / Borda / Plurality scores	Numerous	Plurality / Approval / Borda
	Anonymity respect to other voters	yes	yes	yes
	Anonymity respect to the admin	yes	yes	yes
	Visibility	Public / Private	Private	Private
	1 vote per voter	yes	yes	yes
	Computaion on the fly	no	no	optional
	Dynamic survey	no	no	no
	Monitoring of the scores	yes	yes	yes
	Individual preferences supported	Unique alternative, Dichotomous	Unique alternative,Dichotomous, Complete Linear order	Unique alternative Dichotomous, Linear order
	Collective preferences supported	Unique Winner	Unique Winner	Complete Linear order, Lottery
User experience	account needed	no	no	no
	graphical result	no	no	no
	graphical vote	yes	yes	no
	user friendly inteface	no	no	no
admin	dashboard	no	no	no
	graphical poll creation	no	no	no
	user friendly inteface	no	yes	no
	participants managment	yes	yes	yes
	account	yes	yes	yes



Existing Solution	Name:	Qualtrics	democratix	Pnyx
	Url	www.qualtrics.com	democratix.dbai.tuwien.ac.at	vmbichler25.informatik.tu-muenchen.de/
Use case	Main Purpose	marketing tool	online agregation tool	online agregation tool
	LICENSE	Commercial	Open Source	Open Source
Voting rules properties	Supports election	no	no vote	no
	Supports poll	yes	no vote	yes
	Supports survey	yes	no vote	no
	Voting rules supported	Plurality / Approval	Numerous	Numerous
	Anonymity respect to other voters	yes	no vote	yes
	Anonymity respect to the admin	yes	no vote	yes
	Visibility	Private	no vote	Public / Private
	1 vote per voter	yes	no vote	optional
	Computaion on the fly	yes	no vote	yes
	Dynamic survey	yes	no vote	yes
	Monitoring of the scores	yes	no vote	yes
	Individual preferences supported	Unique alternative,Dichotomous, Complete Linear order , Complete Preorder,	Complete Linear order	Unique alternative,Dichotomous, Complete Linear order , Complete Preorder, Complete Binary relation
	Collective preferences supported	Unique winner, Lottery	Unique winner	Unique winner, Lottery, Complete Linear order
User experience	account needed	no	no vote	no
	graphical result	yes	no vote	yes
	graphical vote	no	no vote	yes
	user friendly inteface	yes	no vote	yes
admin	dashboard	yes	no vote	yes
	graphical poll creation	yes	no	no
	user friendly inteface	yes	no vote	yes
	participants managment	yes	no vote	yes
	account	yes	no	yes