Big Data and Information Analytics
©American Institute of Mathematical Sciences
Volume **1**, Number **4**, October **2016**

doi:10.3934/bdia.2016013

pp. **309–340**

A TESTBED TO ENABLE COMPARISONS BETWEEN COMPETING APPROACHES FOR COMPUTATIONAL SOCIAL CHOICE

John A. Doucette*

University of Waterloo & New College of Florida
5800 Bayshore Road
Sarasota, FL 34234, USA

Robin Cohen

University of Waterloo
200 University Avenue West
Waterloo, ON N2L 3G1, Canada

(Communicated by the Aijun An)

Abstract. Within artificial intelligence, the field of computational social choice studies the application of AI techniques to the problem of group decision making, especially through systems where each agent submits a vote taking the form of a total ordering over the alternatives (a preference). Reaching a reasonable decision becomes more difficult when some agents are unwilling or unable to rank all the alternatives, and appropriate voting systems must be devised to handle the resulting incomplete preference information. In this paper, we present a detailed testbed which can be used to perform information analytics in this domain. We illustrate the testbed in action for the context of determining a winner or putting candidates into ranked order, using data from realworld elections, and demonstrate how to use the results of the testbed to produce effective comparisons between competing algorithms.

1. **Introduction.**

This paper is in the topic area of computational social choice, which concerns the computational study of facilitating group decision making among intelligent agents. The particular focus is on enabling votes from agents to be effectively aggregated in order to select the winner from amongst a set of possible candidates, or to put the candidates into a ranked order, to reflect the collective preferences of the community. While one application may be political elections, where votes are registering an individual's preference for a specific leader or representative, it is important to note that the paradigm of voting and aggregating votes is applicable as well to any environment where a group decision is to be made (for example, enabling a collection of robots to determine where each will be operating; selecting the primary recommendation for an individual by examining preferences recorded by a set of peers).

2010 *Mathematics Subject Classification.* Primary: 91B14, 91B10; Secondary: 68T37.

Key words and phrases. Information analytics for voting systems, computational social choice, machine learning, partial preferences, testbeds.

The authors acknowledge support from the Natural Sciences and Engineering Research Council of Canada, the Ontario Graduate Scholarship Program, and the University of Waterloo.

* Corresponding author: John A. Doucette.

In this work, we describe in full a testbed designed to enable comparisons between competing algorithms for aggregating votes from a community of agents, when individual agents are unwilling or unable to provide the complete information usually required by group decision making systems. Particular features of the work include: careful selection of a set of metrics for comparing algorithms, concern for a proper interface for users, decisions about implementation that enable comparisons to be made efficiently, and detailed consideration of the various options that should be admitted, within the framework.

We illustrate the testbed operating with data drawn from real-world elections, to compare three existing techniques for group decision making with incomplete information. This is an application where one can imagine vast stores of data, which may be challenging to manage and to analyze. We briefly comment on the reasons why it is difficult to design computational systems to enable group decisions, towards the end of the paper, to reinforce the value of the testbed that has been designed, for current researchers and practitioners.

2. **The Prefmine system.**

In our research, we have been developing models to impute missing data in ballots expressing voters' ranked preferences [12]. This is done in order to facilitate group decision making. A common application area for group decision making is that of real-world political elections, where a group is trying to determine a candidate to be chosen to represent the population. When introducing our models, we have provided empirical data to validate the effectiveness of our algorithms. These results involved 100 repetitions of each experimental condition, with 4 voting rules, 11 electoral datasets containing tens of thousands of ballots in total, and 4 different systems that could be used to address the impact of missing preference information on the outcome of the election. In total 4,400 decision problems were considered, and 17,600 decisions were made. When performing experiments on this scale, it pays to have a reliable system, in which one can be certain of the integrity of the data, experimental methodology, and results. Such a system was constructed largely from scratch, but in a carefully tested and designed fashion. The resulting system has several features that may be of great interest to practitioners and to other researchers. First, since the system contains implementations of the imputation-based approach to social choice under several classification systems [12], and also of the Minimax Regret approach [29], a worst-case approach, and several other algorithms, it can serve as a ready means of comparison for practitioners interested in evaluating their own approaches to this problem domain, or in making electoral decisions using any of the implemented techniques. Researchers may also be interested in the system as an independent implementation of existing algorithms for social choice, given that typical scientific software has disagreement in the output results on the order of 10% between different implementations of the same algorithm [20, 19], perhaps because of poor testing practices adopted in the development of most scientific software [25]. By comparing the results of several independent implementations, researchers can be surer of their results, and uncover issues in their own implementations. Second, the system contains implementations of four voting rules, and is readily extensible to accommodate many more. Practitioners interested in using the system to decide elections can compare the results under several different voting systems, with or without additional systems like MMR or the imputation-based approach augmenting them. Further, the implementations of these systems adopt a simple parallelization strategy to provide quick evaluation. Finally, other researchers may be interested in the experimental design available in this system, which leverages real world datasets to create plausible problems where the ground truth is known, but concealed from the algorithms under evaluation. This could be a very potent evaluation tool for other new algorithms intended for the same problem domain.

The testbed system is called "Prefmine", since it facilitates data mining over the Preflib repository of datasets [32] (although the system could be readily extended to work with other online repositories). The Preflib repository contains dozens of datasets, each of which contains the (usually incomplete) ballots from real human elections, in contests from politics, sports, and non-governmental organizations. Prefmine is not the first system intended to mine preference data, though it fulfills a different niche than other methods. Web-based social choice systems like Pnyx [4], Spliddit [17], Whale^3 [3], Democratix [6] and RoboVote[1] provide user friendly implementations of social choice functions that may be difficult to implement or operate correctly, to assist with popularizing these techniques. Ordinary users can submit preferences to the systems and obtain results from sophisticated Condorcet extensions or other rules that are automatically selected according to expert knowledge, in order to fit the users' problem domain. The systems each offer some specialized benefits. For example, Democratix implements answer set programming [15, 16] to select winners using voting rules that are NP-Complete, while Whale^3 boasts an extremely simple user interface. Preflib itself provides a set of associated tools for generating synthetic data for social choice [33], as well as data for matching domains [10].

Additionally there are several frameworks designed to facilitate machine learning over preference data. LPCforSOS [22] implements a binary classification approach similar to the one adopted in the initial implementation of the imputation-based approach to social choice. The SVM-rank toolkit [24, 23] provides a means to apply the popular Support Vector Machine algorithm [7] to learning rankings. There are also more general frameworks that allow the application and integration of several approaches, including the Preference Learning Toolbox [14], a recent Java-based toolkit, and WEKA-LR [2], an extension for the popular Weka machine learning toolkit [18] that allows models to output rankings instead of classes during classification.

Prefmine should be understood as an integrated framework offering the features of a preference mining toolkit (i.e. learning algorithms capable of dealing with rankings) alongside features more typical of a toolkit for computational social choice (i.e. efficient implementations of voting rules, synthetic data generation routines). In this respect, it is novel compared with the approaches described above. Prefmine implements a combination of machine learning approaches similar to the label-ranking algorithms of LPCforSOS, alongside more general learning algorithms, and problem specific approaches like MMR that do not use machine learning, and instead optimize the social choice decision process directly. Additionally, Prefmine provides efficient implementations of several voting systems, and an extensible framework for implementing many more, alongside algorithms for the generation of synthetic data according to several common preference distributions. Prefmine should be viewed as complementary to existing systems, bringing together two different sets of functionality in a single package, while simultaneously providing seamless access to the data stored in Preflib itself.

[1]A forthcoming system with anonymous authors. See robovote.org.

2.1. **System design.** Prefmine is designed both to facilitate experiments, and to ensure that they are conducted correctly. This broad design goal can be expanded into six objectives:

1. Data should be seamlessly obtained from Preflib, or other online repositories with similar formats.
2. The system should store the original data in a read-only format, so that experimenters cannot accidentally change it.
3. Problem instances should be easy to generate from both real-world and synthetic data.
4. It should be easy to run arbitrary combinations of social choice algorithms on a given problem instance, and easy to collect arbitrary performance measures.
5. The system should make use of all available computational resources (to avoid having experimenters implement their own error-prone parallelism).
6. The system should be easy for a new user to apply.

Seamlessly obtaining data from Preflib entails both fetching the desired datasets in a straightforward manner, and processing them from their current format in a representation more suitable for the imputation-based approach to social choice. It is important that this process be automated to ensure consistency, because manual processing can very easily introduce errors into the source data. For this reason also, Prefmine dynamically fetches data from the Preflib repository each time a new experiment is run, rather than storing it locally where it might be subject to (inadvertent) corruption by the experimenter. This is marginally slower than storing the data on a local disk, but the datasets on Preflib, and indeed, most social choice sets in general, are relatively small. They contain on the order of tens or at most hundreds of thousands of votes, and at most dozens of candidates. Preflib also uses a simple compression format to store the data more efficiently. The result is that fetching the data is a rapid step when modern network connections are utilized, lasting no more than a second or two.

Once the data has been obtained from Preflib, Prefmine stores it as an immutable data structure, using a language with true immutability (the `D` programming language). It is therefore not possible for an experimenter's own (novel) social choice algorithm to inadvertently modify the data, as attempting to do so will trigger a fault. This ensures that, for example, performance measures based on aggregates over a ground-truth ordering are never computed over data that has been modified by the methods under assessment. It also facilitates parallel processing of the original dataset, because there is no chance of race conditions arising from data being mutated.

Prefmine adopts an extensible framework to allow easy incorporation of new problem generation algorithms, social choice functions, imputation algorithms, and performance measures. The system uses a plugin-style architecture, where new features can be added without changing the core experiment, analysis, and data loading code at all, preserving the integrity of these components. The system also boasts a simple user interface featuring both a graphical mode and "headless" operation via the commandline. The interface also makes use of the plugin architecture, allowing users to seamlessly access newly developed functionality through the graphical interface.

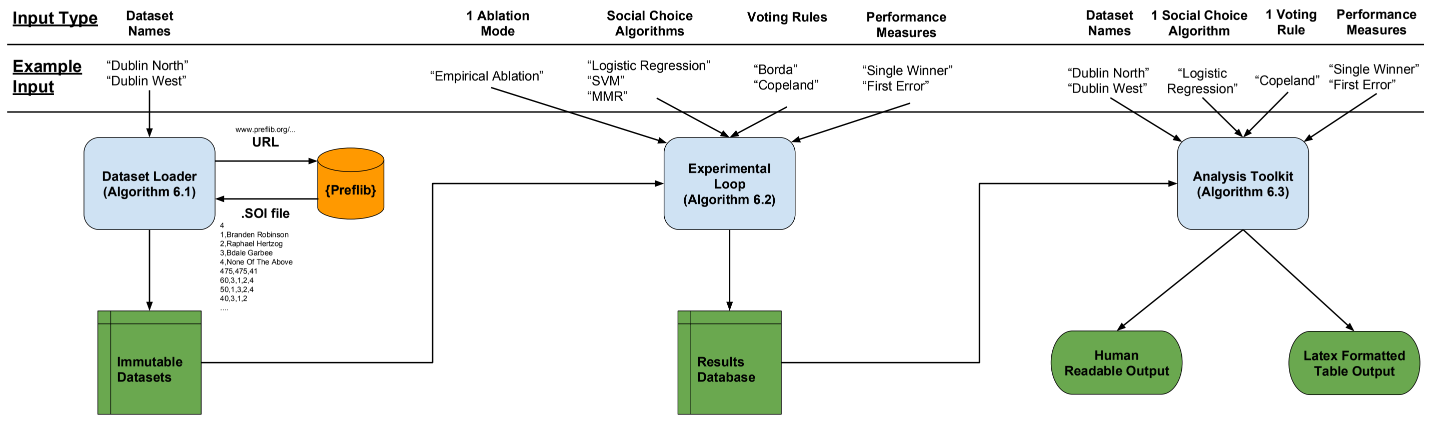
Prefmine also seeks to use all computational resources available on the system upon which it is run. Modern commercial desktop computers, as well as servers, typically feature a multi-core architecture in which properly parallelized code can run an order of magnitude faster than code which runs sequentially. Importantly, parallelization is a highly error-prone process, and the subtle bugs it produces can yield slightly (or wildly) incorrect computations without giving off overt signs to the user like a program crash. Indeed, many development versions of Preflib exhibited these properties. With this in mind, Prefmine seeks to minimize the need for experimenters to consider parallelism when adding extensions to the system (e.g. new imputation methods, voting rules, etc.), and instead to parallelize the process inside the core experimental setup, making maximal use of computational resources. Although many of the results generated in [12] used a parallel version of Prefmine, recent updates to the core design of the D programming language (in which Prefmine is implemented) have necessitated a rewrite of the parallel processing code[2]. As a result, the current version of Prefmine uses only one CPU core at a time, and is significantly slower.

2.2. **An algorithmic description of Prefmine.** The previous few pages situated the Prefmine system within the context of other similar systems, and outlined the broad design goals of the system and the reasoning behind them. This subsection provides an algorithmic description of the core Prefmine system, the main experiment loop. The loop operates over a large set of parameters, which will be described first. The remainder of this section is dedicated to a “user manual”-style description of Prefmine, including both examples showing how the system is to be used, and explaining all of the possible settings present in the current version of the system. In many cases these settings are simply instantiations of algorithms presented in the existing research literature. A reader interested only in the general design of the system can safely skip Subsection 2.4 through to the start of Section 3.

Prefmine's core consists of three components: a system for *loading* electoral data in .soi format [32] (i.e. one of several format's available for data from the Preflib repository, and a common native format for ballots) and storing them immutably; an *experimental loop* which can dynamically generate new problem instances from stored datasets, decide them using arbitrary combinations of imputation algorithms and voting rules, or using comparison algorithms like Minimax Regret, and assess the quality of the resulting decisions relative to the ground truth data; and an *analysis toolkit* that can produce both human readable and $\LaTeX$-formatted summaries of the results from running the experimental loop. The three components are depicted together in Figure 1.

The Dataset Loader component of Preflib is summarized in Algorithm 1. The loader accepts a string corresponding to the name of a dataset, and looks up the appropriate URL on the Preflib website, obtaining a .soi file. The .soi format is a **S**trict **O**rder, **I**ncomplete list representation of the ballots in an election. The start of the file contains metadata: the number of candidates, a mapping from candidate names to numbers, and information about the total number of ballots in the set. After this, the file contains one line per *unique* ballot in the election. The first number on a line indicates the number of voters who cast this ballot. The remainder of the line is a comma-separated list of numbers, which are mapped to candidates according to the mapping in the metadata. Candidates that appear earlier (i.e. closer to the start of the line) in this list strictly precede those who come later. The list need not contain every candidate. The interpretation of unranked candidates

[2]In particular, the removal of semi-immutable maps has rendered much of the core Prefmine code non-threadsafe.



```
4
1,Branden Robinson
2,Raphael Hertzog
3,Bdale Garbee
4,None Of The Above
475,475,41
60,3,1,2,4
50,1,3,2,4
40,3,1,2
....
```

FIGURE 1. A graphical depiction of the data flow in Prefmine, with example input arguments. Rounded boxes denote algorithms, presented in full in the text. Cylinders show external data stores. Sharp squares denote immutable internal data stores, output by one of the system's algorithms. Ovals show the system's final output. Processing begins with the leftmost algorithm (the Dataset Loader), which downloads data from the Preflib repository, and then formats it as an immutable database. The immutable datasets are then passed to the middle algorithm (The Experimental Loop), which runs experiments on them according to its other input parameters, producing an immutable results database. The rightmost algorithm (The Analysis Toolkit) can be used to generate human readable and Latex-formatted tables by querying the results database. Queries are constructed from the input parameters. Existing valid parameter settings are discussed later in the text, but most parameter sets are easily extensible.

depends on the election, but in many cases the original ballots were effectively interpreted as top orders. In its current implementation, Prefmine interprets all datasets obtained in .soi format as top orders. The runtime of the Dataset Loader algorithm is in $O(|B||C|)$, where B is the set of ballots and C is the set of candidates.

After obtaining a .soi file, the loader processes it into an immutable datastore. Candidate mapping is stored as an immutable hashmap. Each line is converted into a "Datum" data structure. A Datum is a map from positions to sets of candidates. If and only if the (partial) ordering implied by a given line could be completed such that a candidate c_i were placed in a position p_j, will the corresponding Datum structure return a set of candidates containing c_i when queried with key p_j. The loader's immutable datastore is called a "Data" instance, and contains metadata (the total number of ballots; the Candidate mapping), and a list of Datum structures corresponding to the list of orderings from the original .soi file. The entire Data instance is output as an immutable structure, meaning none of the components can be changed in any way. Data can only be read, not written.

Once the Dataset Loader has completed its task, the Experimental Loop algorithm (described in Algorithm 2) takes over, and performs a series of repetitions according to an experimental design specified by the passed parameters. Each iteration of the outermost loop is a single repetition of the experiment. A new problem instance is generated by making a mutable copy of the *complete* rows from an immutable Dataset instance that was generated by the Dataset Loader (Algorithm 1), and then ablating it according to the single Ablation Mode parameter's setting. This parameter can currently be configured to either ablate the data according to a distribution learned from the corresponding immutable Dataset instance, or a user-specified vector of probabilities. Both modes are discussed in the next subsection, along with examples of their use.

After the new problem instance has been generated, each member of the list of Social Choice Algorithms is given a deep copy of the problem instance. If the method relies on the generation of imputations, it generates an imputation, and each of the listed Voting Rules is run on the resulting imputation to generate a set of orderings (one per voting rule). If the method does not rely on the generation of an imputation (e.g. Minimax Regret), then it is instead called with each voting rule in turn as an argument to obtain these orderings. Subsection 2.5 discusses the implementation details of this part of the system and shows example uses. Additionally, each of the listed voting rules is run once on the true-preferences of the problem instance to obtain the "correct" orderings. Finally, every member of the list of Performance Measures is applied, comparing each ordering produce by a Social Choice Algorithm under each voting rule, to the corresponding "correct" ordering. The results are stored in an immutable database under a composite key that encodes the dataset, Social Choice Algorithm, Voting Rule, and Performance Measure that were used, as well as the repetition number from the main loop. The entire main loop is then repeated. In the final step of the algorithm, the output database is rendered immutable in the final step to ensure that the data are not tampered with by the system inadvertently[3]. The total runtime of the algorithm is highly dependent on the runtimes of the parameters it is passed. If k datasets, all with ballot sets smaller than $|B|$ over candidate sets smaller than $|C|$ are passed, and l voting rules, all taking less than $O(S(B))$ time to evaluate over the most complex

[3]In practice the database is written to disk, and the files marked as read only after the run is complete.

Algorithm 1 An algorithmic description of the Dataset Loader component of the Prefmine system. The Dataset Loader accepts a string corresponding to the name of a dataset, and then downloads the corresponding .soi file from the Preflib repository [32]. The file is then processed into an immutable Data instance which is produced as output.

```
procedure DatasetLoader(DatasetName)
    Let url ← lookupURL(DatasetName)
    Let $S$ be a stream obtained by opening url.
    Let $|C|$ ← S.nextLine()
    Let CandMap = ∅
    for i = 0; $i < |C|$; i++ do
        Let {key, name} = split(S.nextLine(),“,”)
        Let CandMap[key] = name
    end for
    // The last line of the metadata contains the number of ballots.
    Let $|B|$ ← split(S.nextLine(), “,”)[0]
    Dataset output = ∅
    while S.hasNextLine() do
        //Each remaining line is parsed into a top order.
        tokens ← split(S.nextLine(),“,”)
        numBallots ← tokens[0]
        Datum d = ∅
        notAssigned = CandMap.keys()
        for i=1; $i <$ —tokens—; i++ do
            d[i] = tokens[i]
            notAssigned \ = tokens[i]
        end for
        for i= —tokens—; i ¡ —C—; i++ do
            d[i] = notAssigned
        end for
        for i = 0; i ¡ numBallots; i++ do
            output ← append(output, d.duplicate())
        end for
    end while
     return cast(Immutable Dataset) output
end procedure
```

dataset, and m Social Choice Algorithms all taking less than $O(M(B))$ time, and n performance measures, all taking less than $O(P(o_1, o_2))$ time, are passed, then the total runtime will be in $O(\text{NumReps} \cdot k(\text{ablate}(B) + ml(S(B) + M(B) + n)))$

The final component of the Prefmine system is the Analysis Toolkit, an algorithm that allows the analysis of results generated by the Experimental Loop. The toolkit allows the user to specify an output database from a run of the Experimental loop, a social choice algorithm and a voting rule, as well as a list of performance measures. The toolkit then generates a tabular summary of the mean performance of that social choice algorithm under that voting rule, with respect to each performance measure. Sample standard deviations are also reported. The table is output both in a human-readable format, and as a Latex tabular environment that can be

Algorithm 2 An algorithmic description of the Experimental Loop component of the Prefmine system. The Experimental Loop accepts an Immutable Dataset (produced using Algorithm 1), an ablation mode, a list of social choice algorithms (e.g. imputation-based, Minimax Regret), a list of voting rules (e.g. **Borda**, **Copeland**, a list of performance measures (e.g. Single Winner Error, First Error Location), and a number of repetitions for the experiment. For each repetition of the experiment, a new problem instance is generated using the specified ablation mode. Then every social choice algorithm is run under every voting rule to produce an outcome. The outcome is compared to the ground truth outcome for the voting rule under consideration using every performance measure. The results are written to an output database, which is rendered read-only as the final step.

```
procedure ExperimentalLoop(ImmutableDatasets, AblationMode, ListOfSCAlgs, ListOfVotingRules, ListOfPrefMeasures, NumReps)
    Let Output = $\emptyset$
    for Rep = 0; Rep ¡ NumReps; Rep++ do
        for all Dataset in ImmutableDatasets do
            Let Problem = AblationMode(Dataset.copy())
            for all Alg in ListOfSCAlgs do
                for all Rule in ListOfVotingRules do
                    if Alg uses Imputation then
                        Outcome = Rule(Alg(Problem.expressedPrefs.copy()))
                    else
                        Outcome = Alg(Problem.expressedPrefs.copy(), Rule)
                    end if
                    CorrectOutcome = Rule(Problem.truePrefs)
                    for all Measure in ListOfPerfMeasures do
                        Key = concatenateNames(Dataset, Alg,Rule,Measure,Rep)
                        Output[Key] = Measure(Outcome, CorrectOutcome)
                    end for
                end for
            end for
        end for
    end for
     return cast(Immutable) Output
end procedure
```

dropped into a document with minimal editing. The Analysis Toolkit's behaviour is summarized in Algorithm 3. The algorithm's runtime is in $O(\text{NumReps} \cdot kn)$, where k is the number of datasets, and n is the number of performance measures, assuming that printing is a constant time operation, and NumReps is the largest number of entries in the database for any single key.

This concludes the presentation of the Prefmine system at a high level. Note that the algorithms as presented contain some inefficient components to facilitate their presentation. For example, Algorithm 2 repeatedly recomputes the value Alg(Problem.expressedPrefs.copy()), when in practice this is computed once and cached. Additionally, the algorithms are all presented as sequential when (at least in earlier versions of the system) parallel processing took place. In the past, parallel processing took place within individual algorithms. For example, when training a

Algorithm 3 An algorithmic description of the Analysis Toolkit component of the Prefmine system. The Analysis Toolkit accepts an output database produced by the Experimental Loop component of the system (Algorithm 2). The user also specifies a particular social choice algorithm and voting rule, as well as a list of datasets and performance measures. The toolkit computes a table where each row corresponds to a dataset, and each column to a performance measure. The value in a particular table cell will be the mean and standard deviation of the selected social choice algorithm under the selected voting rule, with respect to the corresponding performance measure (i.e. column) and dataset (i.e. row). The resulting table is then output both in a human readable format and as a Latex tabular environment.

```
procedure AnalysisToolkit(OutputDatabase, SCAlg, VotingRule, ListOfPrefMeasures, ListOfDatasets)
    Let Table = $\emptyset$
    for all Dataset in ListOfDatasets do
        for all Measure in ListOfPerfMeasures do
            Key = concatenateNames(Dataset,SCAlg,VotingRule,Measure, *)
            Results = OutputDatabase[key]
            Mean = computeMean(Results)
            Stdev = computerStandardDeviation(Results)
            Table[Dataset][Measure][“mean”] = Mean
            Table[Dataset][Measure][“stdev”] = Stdev
        end for
    end for
    Print(Table)
    PrintLatex(Table)
end procedure
```

chained classifier based on logistic regression, many classifiers could be trained in parallel from the same dataset. After changes in the implementation language took place during 2015, this feature ceased to function properly and was removed. Future versions of Prefmine will likely include parallelism in the outermost loop of Algorithm 2 (i.e. the Reps loop) instead. Although this is not always maximally efficient, it will ensure thread safety and should produce a significant speedup when the slowest social choice algorithms are run. Additional parallelism was previously present in the computation of the social choice functions, which adopted a map-reduce paradigm [8] to efficiently compute the outcome.

The remainder of this section serves as a reference manual for Prefmine, describing its use, features, and how to extend the existing code base, with examples. The features and usage are both presented via the graphical user interface.

2.3. Using Prefmine.

Prefmine is implemented in the D programming language. To compile Prefmine, users must install the Digital Mars D compiler `dmd`[4], and the `dub` package manager[5]. After obtaining the Prefmine source code from the author, users can run `dub` in the top-level directory of the project source to compile and execute the code. `dub` will automatically fetch and install all other required libraries. `dub` will produce an executable named `prefmine` in the top-level directory

[4]http://www.digitalmars.com/d/1.0/dmd-linux.html

[5]For Debian-based operating systems: http://d-apt.sourceforge.net/

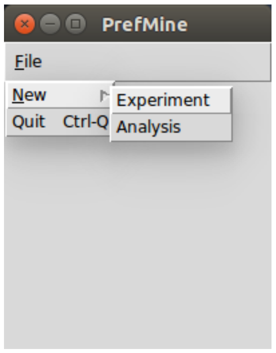


FIGURE 2. The initial Prefmine window, from which users can launch a new experiment, or run the analysis toolkit.

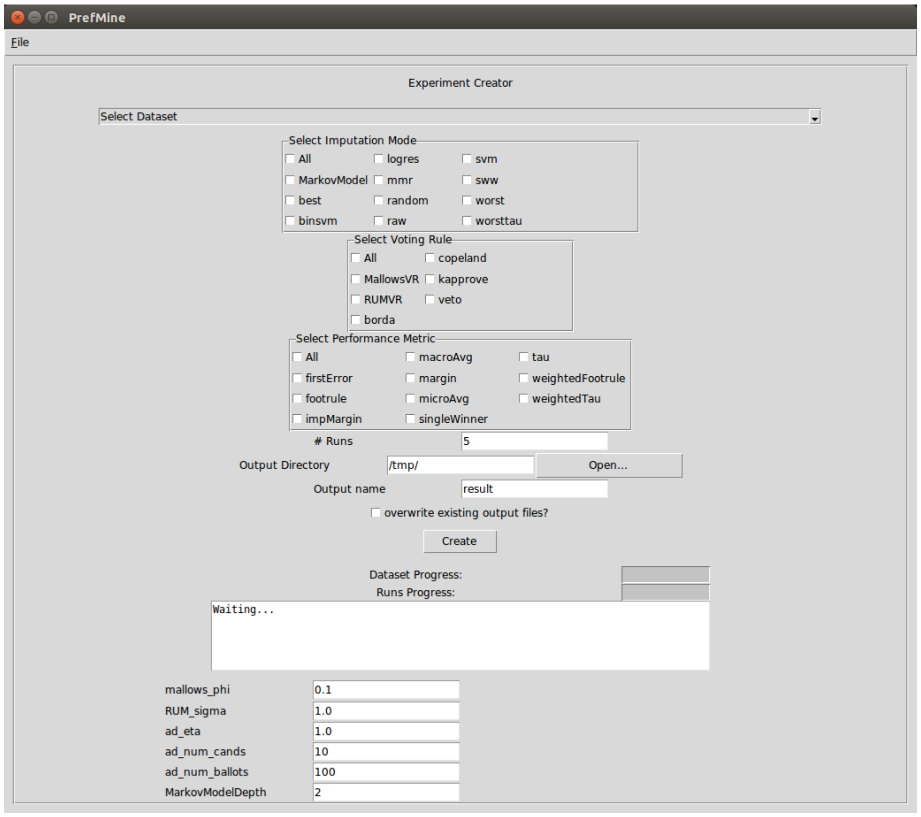
of the project source code after a successful compilation. Running the `prefmine` executable will open a small window, shown in Figure 2. Users can elect to either start a new experiment or run the analysis toolkit, by selecting the corresponding options from the drop down menu, as shown.

Starting a new experiment will display the window shown in Figure 3, which allows the user to configure which datasets will be loaded, and to configure the inputs to both the Dataset Loader and Experimental Loop portions of the system. The configuration of the experiment is detailed later in this section. Additionally, the user can configure the directory to write the experimental loop's output database to, and indicate whether to overwrite or append to any existing database present at that location. After selecting the settings they prefer (e.g. Figure 4), the user presses the "Create" button. Datasets are loaded, and the experimental loop begins running. Two progress bars in the lower right indicate the progress of the experimental loop, and the large textbox provides a summary of this progress (Figure 5). When the experiment is complete, the user should close the application.

The process for analysis of the data is similar. Instead of selecting "Experiment" from the File menu at the start, one selects "Analysis". A very similar window will be constructed, displayed in Figure 6. Radio buttons limit the user to the selection of a single imputation method and a single voting rule, while the familiar checkbox-style interface allows the selection of multiple performance measures. All datasets stored in the results database at the specified path will be presented in the results table.

2.4. **Dataset generation.** In Prefmine, the selection of datasets for the Dataset Loader to fetch is accomplished through a dropdown menu at the top of the experiment window, as seen in Figure 7. The menu is used to avoid any possible confusion on the part of the experimenter about which dataset an experiment is being run on: selection of the specified name ensures that data is fetched from the corresponding URL. The menu is populated dynamically from the Preflib repository's website, so all available datasets are listed[6]. In addition to each individual set of electoral data, it is possible to run a Prefmine experiment on any set of related elections at one, if they are stored together on Preflib as a single collection of data. For example, the

[6]In the example windows, this feature has been disabled by the author to avoid searching through such a large list.



Runs: 5
Output Directory: /tmp/
Output name: result

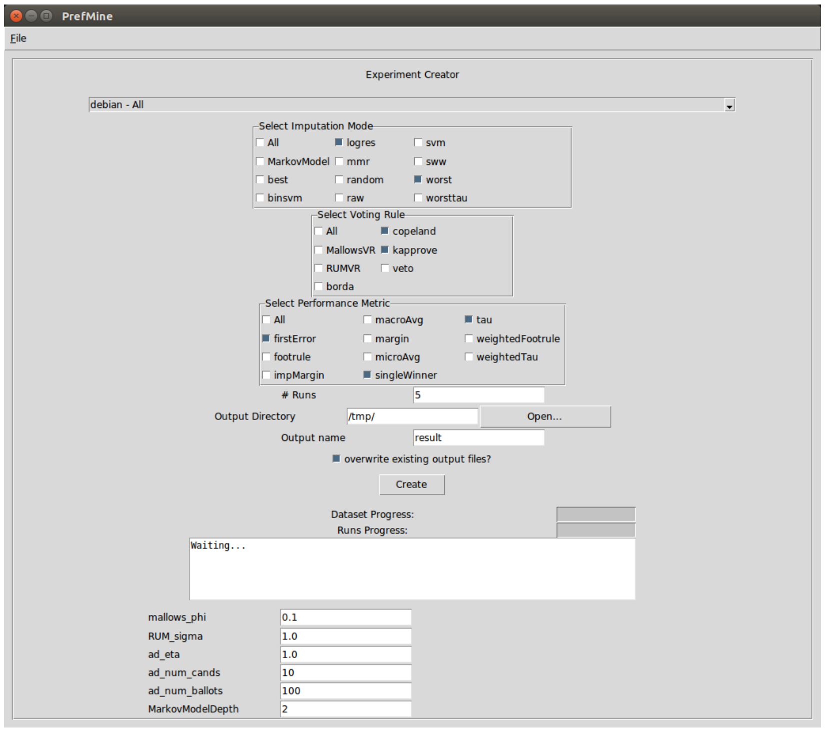
```
Waiting...
```

mallows_phi	0.1
RUM_sigma	1.0
ad_eta	1.0
ad_num_cands	10
ad_num_ballots	100
MarkovModelDepth	2

FIGURE 3. The experiment configuration window in Prefmine, from which the user can configure input parameters for both the Dataset Loader and Experimental Loop portions of the system.

7 Debian elections and 3 Irish elections are grouped together, and so experiments can be run on these 3 or 7 sets all at once. Additionally, two synthetic dataset generators are listed in the dropdown menu: RUM and NoisedMallows. The RUM option is an implementation of a Random Utility Model [30] for generating problem instances, based on [1]. This model assumes that each voter's utility for a given candidate is sampled from a Gaussian distribution. All voters sample utilities from the same Gaussian distribution for a particular candidate, but each candidate has its own distribution. The Gaussian distribution for each candidate has mean sampled uniformly from $(0, |C|)$, and standard deviation equal to the `RUM_sigma` parameter setting, which is configured near the bottom of the experiment window. A voter's true preferences are implied by the utility they have sampled for each candidate: the candidate with the highest sampled utility is ranked first, and the candidate with the lowest sampled utility is ranked last.

The other synthetic data generator is simply a standard Mallows model [31]. The central ordering is a permutation of the candidates selected uniformly at random. The dispursion parameter ϕ is set using the `mallows_phi` parameter near the bottom of the experimenter window. Synthetic ballots are sampled from the Mallows model using a re-implementation of the efficient algorithm from [28].



debian - All

Select Imputation Mode

☐ All	☑ logres	☐ svm
☐ MarkovModel	☐ mmr	☐ sww
☐ best	☐ random	☑ worst
☐ binsvm	☐ raw	☐ worsttau

Select Voting Rule

☐ All	☑ copeland
☐ MallowsVR	☑ kapprove
☐ RUMVR	☐ veto
☐ borda	

Select Performance Metric

☐ All	☐ macroAvg	☑ tau
☑ firstError	☐ margin	☐ weightedFootrule
☐ footrule	☐ microAvg	☐ weightedTau
☐ impMargin	☑ singleWinner	

# Runs	5
Output Directory	/tmp/
Output name	result

☑ overwrite existing output files?

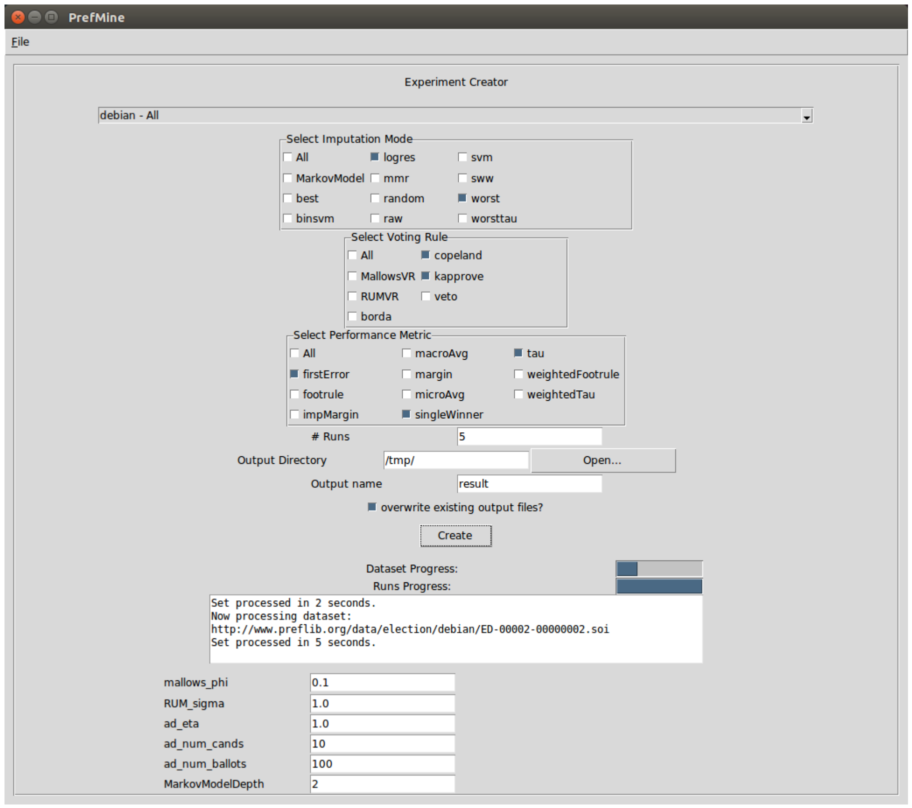
Dataset Progress:
Runs Progress:

```
Waiting...
```

mallows_phi	0.1
RUM_sigma	1.0
ad_eta	1.0
ad_num_cands	10
ad_num_ballots	100
MarkovModelDepth	2

FIGURE 4. An example of an experiment configured using Prefmine's experiment configuration window. The experiment will run over the seven Debian datasets from the Preflib repository. Logistic Regression and a Worst Case Imputation approach will be applied to each problem instance, and the **Copeland** voting rule will be used to decide outcomes. Performance will be assessed under the Single Winner Error, First Error Location, and Kendall Correlation (τ) performance measures. 5 replications will be performed, and the output database will be stored in /tmp/, overwriting any existing results there. Since more than one dataset is being processed, the output filename parameter is ignored. The parameters below the output textbox ("Waiting...") are used to configure synthetic data generators or imputation methods that are not used, and so are ignored. Pressing the "Create" button will start the experiment.

Both the Mallows and RUM synthetic data generators share a number of parameters set within the Prefmine experiment window. Collectively these are called the *artificial data* parameters, and are denoted with the prefix `ad_`. All artificial data parameters must be set in order to use a synthetic data generator. The `ad_num_cands` and `ad_num_ballots` parameters respectively specify the number of candidates who will compete in the synthetic election, and the number of ballots that will be cast in the election. The `ad_eta` parameter allows the construction of simple top-ordered ballots, rather than the totally-ordered ballots that are generated by the two methods. The probability of generating a top-ordered ballot of



debian - All

Select Imputation Mode: All, [✓ logres], svm, MarkovModel, mmr, sww, best, random, [✓ worst], binsvm, raw, worsttau

Select Voting Rule: All, [✓ copeland], MallowsVR, [✓ kapprove], RUMVR, veto, borda

Select Performance Metric: All, macroAvg, [✓ tau], [✓ firstError], margin, weightedFootrule, footrule, microAvg, weightedTau, impMargin, [✓ singleWinner]

# Runs	5
Output Directory	/tmp/
Output name	result

[✓ overwrite existing output files?]

```
Set processed in 2 seconds.
Now processing dataset:
http://www.preflib.org/data/election/debian/ED-00002-00000002.soi
Set processed in 5 seconds.
```

mallows_phi	0.1
RUM_sigma	1.0
ad_eta	1.0
ad_num_cands	10
ad_num_ballots	100
MarkovModelDepth	2

FIGURE 5. An example of a Prefmine experiment in progress, using the settings from Figure 4. Note the progress bars showing the fraction of datsets processed (top) and runs completed on this dataset (bottom). The Textbox summarizes progress, including runtimes required to complete each dataset.

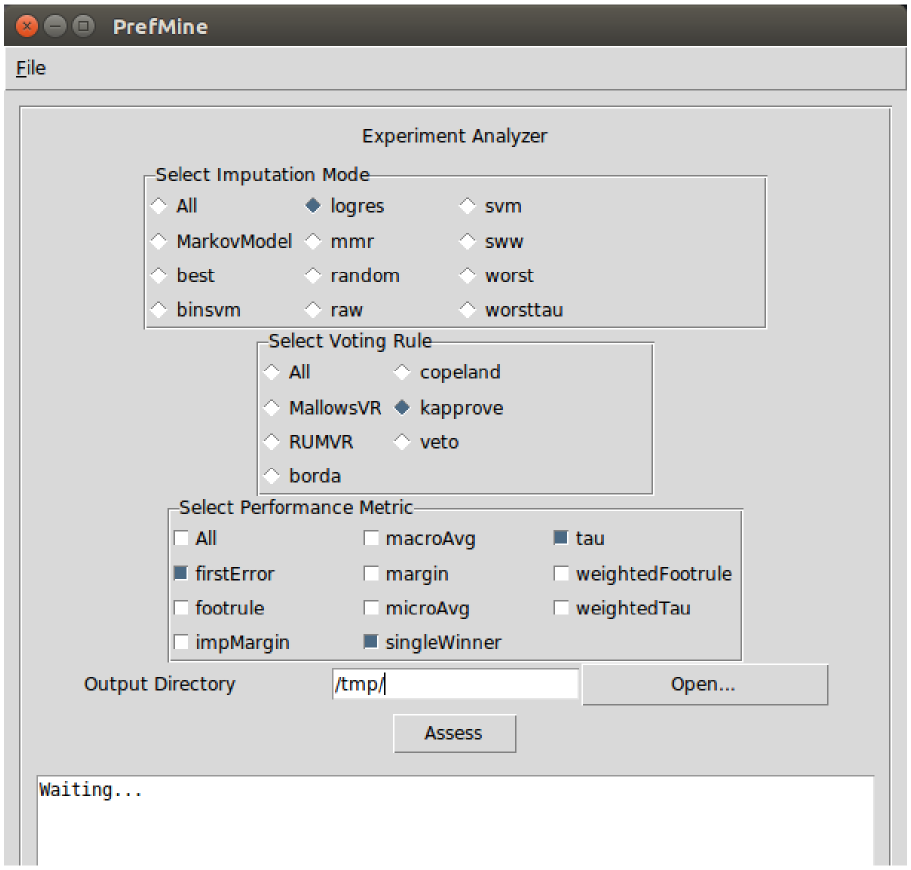
length at least k is given by η^{k-1} (i.e. in expectation a fraction η of ballots rank at least two preferences, η^2 rank at least 3, and so on).

In addition to selecting datasets, Prefmine offers two methods for ablating datasets to generate problem instances[7]. In the first, a user specifies a cumulative density function for the probability that a ballot has at least k candidates ranked, for every $1 < k \leq |C|$, and the system ablates the ballots such that this distribution is observed in expectation. In the other, such a distribution is learned empirically from the original dataset, and the ablation process is identical.

2.5. Imputation Modes.

After selecting a dataset or synthetic data generation method, the Prefmine user should select one or more "Imputation Modes" from the array of checkboxes located directly beneath the dataset selection menu. To select more than one imputation mode at a time, the user simple checks the boxes beside multiple methods. Figure 8 shows a closeup of the array of checkboxes, which currently includes twelve different methods. Some of these are actually comparison

[7]The interface shown in this paper does not allow selection between them, as this created additional clutter. The choice is easily made using the commandline interface to Prefmine.


PrefMine

File

Experiment Analyzer

Select Imputation Mode
All [✓ logres] svm
MarkovModel mmr sww
best random worst
binsvm raw worsttau

Select Voting Rule
All copeland
MallowsVR [✓ kapprove]
RUMVR veto
borda

Select Performance Metric
All macroAvg [✓ tau]
[✓ firstError] margin weightedFootrule
footrule microAvg weightedTau
impMargin [✓ singleWinner]

Output Directory /tmp/ Open...

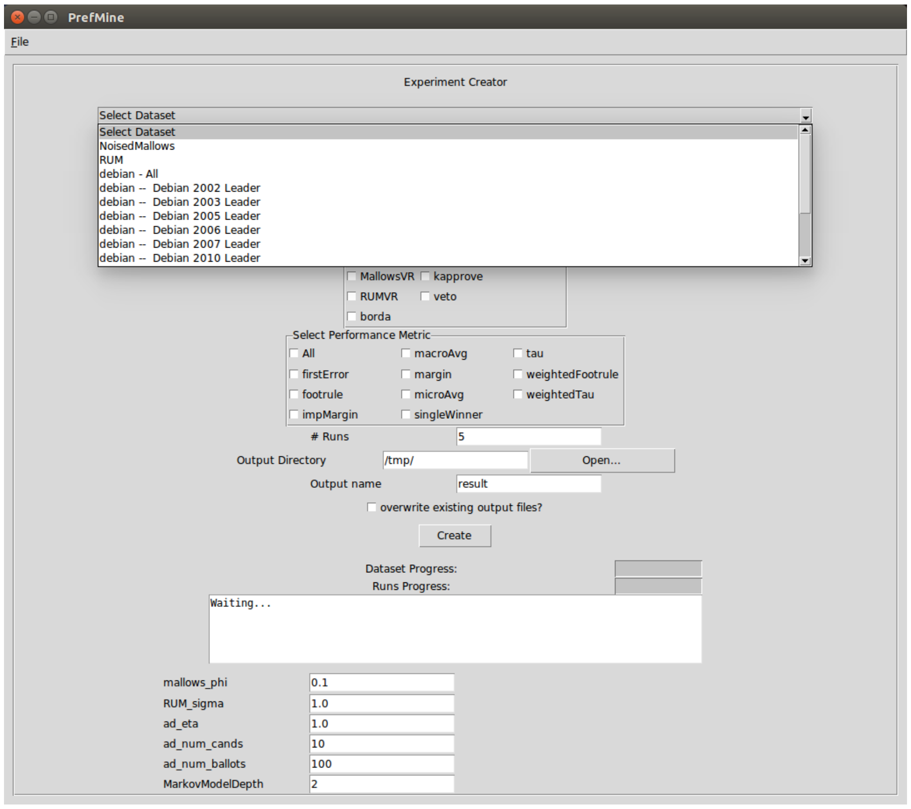
Assess

```
Waiting...
```

FIGURE 6. The Prefmine analysis window. The window is similar to the experiment window, but with a reduced set of options, and larger space to view the output.

algorithms and do not use imputation, despite the name. The details of the methods are itemized below.

- **All**: Behaves identically to selecting every other checkbox in the array.
- **MarkovModel**: This is a Markov-Tree based imputation method. The intuitive idea behind the model is that in domains like voting, ranked alternatives can be thought of as embedded in a small feature space (for example, the familiar left-right political axis). A ranking can be thought of as a **trajectory** through such a space, moving from candidate to candidate, and so can be learned with a variant of a Markov Tree. A Markov Tree itself is simply a different factoring of the distributions in a conventional Markov Model.
- **best**: The "best case" imputation method. The method peeks at the true preferences of users, and imputes each expressed ballot with the correct values. It can be useful to detect whether ties are present in the orderings produced by the true preferences of users, an especially common problem when the **Copeland** voting rule is used.



Runs: 5
Output Directory: /tmp/
Output name: result

```
Waiting...
```

mallows_phi: 0.1
RUM_sigma: 1.0
ad_eta: 1.0
ad_num_cands: 10
ad_num_ballots: 100
MarkovModelDepth: 2

FIGURE 7. The location of the dataset selection dropdown menu in Prefmine's experiment window.

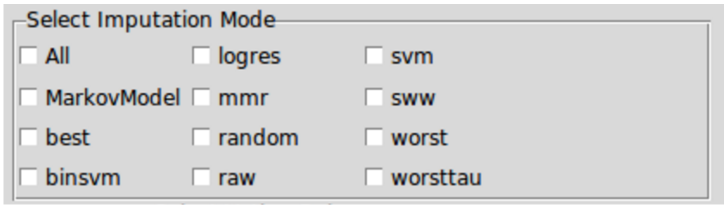


FIGURE 8. The Imputation Method selection box in Prefmine's experiment window.

- **binsvm**: The standard imputation-based approach to social choice described by [12], using a binary Support Vector Machine [7] as the base classifier. A set of SVMs are learned to predict the candidate that should be imputed at each position. Since this is a multi-class classification problem, one SVM is trained to predict membership in each class, a one-versus-all approach [34]. Contrast with the **svm** option, which uses a one-versus-one approach. The individual SVM models are trained using the popular libsvm C++ implementation [5], which is linked directly with Prefmine's D code via a custom interface. Parameters for the SVM are selected using cross validation, with Linear, Polynomial (degree 3), and Radial Basis Function kernels considered alongside every combination of parameter values for $C \in (2^{-5}, 2^{15})$ and $\gamma \in (2^{-15}, 2^5)$, spaced at equal *factors* of 2^4 (i.e. 2^{-15}, 2^{-11}, 2^{-7}, and so on). In the current implementation this can take a tremendous amount of time compared with other imputation methods. A subsampling approach maybe implemented in future to remedy this.

- **logres**: The standard imputation-based approach to social choice described in [12], using logistic regression as the base classifier. This is identical to the model described in [12], including the choice of features, feature selection algorithms, and parameterization. The implementation is a custom version of the conjugate gradient descent algorithm [21] designed to operate efficiently over the standard Prefmine data format.
- **mmr**: An implementation of the Minimax Regret algorithm [29], used as a comparison method and described in [12]. This implementation handles partial orders when used with scoring vector-based voting rules (e.g. **Borda**), but only with top-orders on **Copeland**.
- **random**: A randomized imputation approach, in the spirit of the MLE approach [40]. Starting from the top of the ballot, each position that could be held by more than one candidate is considered (i.e. each position n where the voter has not definitely ranked a candidate in n^{th} place). A candidate is selected uniformly at random from the set of candidates eligible for this position, and this candidate is ruled out for all other positions. The process is then repeated until every candidate has been assigned to a position. For top-orders, this selects a suffix for each ballot uniformly at random.
- **raw**: No imputation is performed. The ballots are passed directly through to the voting rules, in their original, incomplete, forms. This is useful primarily when testing a new ablation model, but could also be used to decide elections directly as a comparison.
- **svm**: As binsvm above (including parameter selection procedures and runtimes), but using libsvm's default one-versus-one approach to multiclass classification instead of the one-versus-all approach used by binsvm and logres.
- **SWW**: Single Winner Worstcase: A comparison method like MMR that computes the set of *possible* winners, per [40], and assigns a score of 1 to members of the set, and 0 to all other members. Essentially it computes an upper bound on the worst possible single winner distance. Current implementations work only for monotonic scoring rules, since other voting rules are NP-Hard to compute possible winners for.
- **worst**: Peeks at voters' true preferences, and imputes the *reverse* ordering (i.e. each candidate goes as far from its true position as possible, while still producing an ordering consistent with the voter's expressed preferences).
- **worsttau**: The system used as a benchmark for the difficulty of different datasets in [12]. It imputes each ballot with the opposite of the correct *aggregate ordering* under the **Borda** voting rule. That is, with the opposite of the ordering that is returned by **Borda** under consideration when run on the true preferences of the voters, which the method peeks at. Contrast with **worst** and **SWW** above. This creates a profile where candidates that were ranked highly *on average* are ranked low consistently, and vice-versa. It is a good heuristic approximation of the SWW method for many voting rules, and is a lower-bound on the worst possible distances, rather than an upper bound

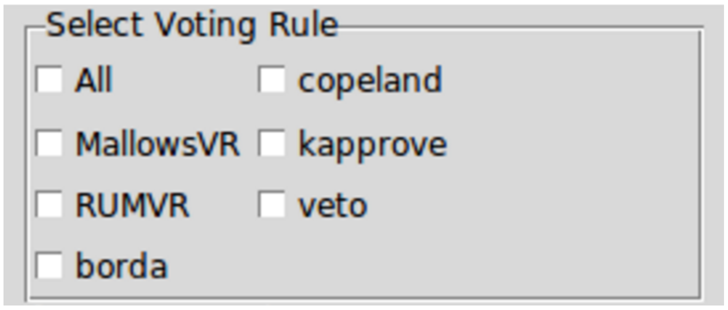


FIGURE 9. The Voting Rules selection box in Prefmine's experiment window.

(i.e. there may exist profiles that are even worse, especially for non-monotonic rules).

Extending the set of "Imputation Methods" is a straightforward proposition. An imputation method must accept a profile of incomplete ballots and produce a completed one. A comparison method must accept both a profile of incomplete ballots and a list of voting rules. It must output a score for each candidate under each rule, such that candidates with higher scores under a given rule come before those with lower scores. In both cases, after writing a function satisfying the above constraints, users can add its name to a list which is used to dynamically generate the box depicted in Figure 8. The system will automatically generate appropriate calls to the function in the experimental loop, storage of its results, and so on.

2.6. **Voting rules.** After picking one or more imputation methods, Prefmine users should pick one or more voting rules for their experiments. The Voting Rule selection box is an array of checkboxes directly below the Imputation Methods box. The current implementation of Prefmine contains seven voting rules, two of which are used only for synthetic data. As with the Imputation Method box, a user who wants to sequentially run several voting rules on the same data simply checks more than one box. The Voting Rule selection box is shown in Figure 9. The various options are itemized below.

- **All**: Equivalent to checking all other boxes.
- **MallowsVR**: A "voting rule" to use in conjunction with synthetic data generated by the Mallows Model. When selected, models with the ability to aggregate rankings directly (like the Markov Model, or MMR), rather than imputing ballots and then using a separate voting rule, will have their estimated rankings directly compared to the true central ranking of the Mallows distribution that was used to generate the current problem instance, rather than using an ordinary voting rule. Activating this function when a dataset other than **NoisedMallows** is selected will trigger a fault.
- **RUMVR**: A "voting rule" to use in conjunction with synthetic data generated by the Random Utilty Model. When selected, models with the ability to aggregate rankings directly (like the Markov Model, or MMR) will have their estimated rankings directly compared to the true central ranking of the RUM that was used to generate the current problem instance, rather than using an ordinary voting rule. Activating this function when a dataset other than **RUM** is selected will trigger a fault.

- **borda**: An implementation of the **Borda** count voting rule. A candidate receives exactly k points for appearing ahead of k other candidates on a ballot.
- **copeland**: An implementation of the **Copeland** voting rule. A candidate receives 1 point for each pairwise runoff contest it wins against other candidates.
- **kapprove**: An implementation of $\frac{|C|}{2}$-approval. A candidate receives 1 point for appearing in the top half of the positions on a ballot (round down), and 0 otherwise.
- **veto**: An implementation of the Veto voting rule. A candidate receives 1 point for every ballot on which it is not ranked last.

Note that none of these voting rules express behaviour in the case of ties. This is because when Prefmine is used for experimental purposes, detection of ties is important (e.g. if the correct ordering contains no ties, but the winning candidate selected by an imputation method was tied with someone else, then this result may have a different interpretation even if the method has picked the correct winner via tie-breaking). Additionally, all voting rules assign a score to each candidate. This is to facilitate measurement of the errors in candidate's scores under rules like Borda. Voting rules that are not based on assigning a score to each candidate can simply assign each candidate a score equal to the number of candidates behind them in whatever ranking is decided upon. Indeed, this is precisely what is done for the current implementations of the MallowsVR and RUMVR rules.

Additionally, note that the implementations of these rules use a simple Map-Reduce [8] approach for the borda, kapprove, veto, and Copeland options. In each case, since the ballots in a profile express independent information, the set of ballots can be partitioned into independent subsets, one for each available CPU core. Aggregate scores can be computed (in the case of Copeland, these are computed for a given pairwise contest, not the overall ranking) by separate threads for each subset. The aggregate scores for each thread can then be combined to obtain the final result. This feature was present in an earlier version of Prefmine, but has been disabled following changes in the implementation language in 2015.

As with the addition of new Imputation Methods, Prefmine's plugin-style architecture allows users to readily add new voting rules of their own. A voting rule must accept a profile of complete ballots (i.e. total orderings). It may optionally accept a profile of partial orderings as well. It must produce a mapping from each candidate to a score, such that candidates with higher scores are ahead of those with lower scores.

2.7. **Performance measures.** After selecting the desired dataset, imputation methods, and voting rules, the user need only select one or more performance measures before starting the experiment (other parameters have functional defaults). Prefmine currently implements eleven different performance measures. Users select desired performance measures from the Performance Metric selection box, located directly below the Voting Rule selection box in the experiment window. This box is pictured in Figure 10. As with the other selection boxes, the Performance Metric selection box contains an array of checkboxes. Users can click on a checkbox to select a performance measure. Clicking on multiple checkboxes allows the user to select several methods. Note that although the name of the box suggests that all options are Metrics, this is not formally true. For example, the **tau** options is a correlation. The eleven possible options are summarized below. Throughout the

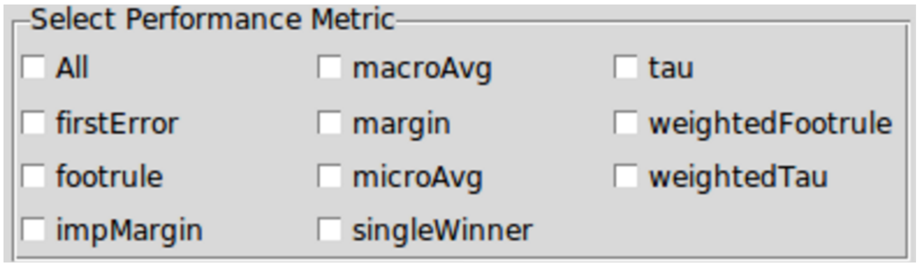


FIGURE 10. The Performance Metric selection box in Prefmine's experiment window.

summary, o_1 is the overall ordering of the candidates under an imputation method of interest, and o_2 is the ordering under the true preferences of voters. Additionally, S_1 and S_2 are used to denote the *scores* for the candidates, as output by the voting rules described in the previous subsection.

- **All**: Equivalent to selecting all other options simultaneously.
- **firstError**: The First Error Location measurement, defined as $\delta_{\text{FE}}(o_1, o_2) = \arg\max_i \forall j < i, o_{1,j} = o_{2,j}$. Returns the location of the highest ranked candidate in o_1 that has been placed in an incorrect position. Proportionate to the length of the prefix under which two sequences agree. This measurement is useful in conjunction with measurements like **tau** below, which give an overall view of the similarities between two sequences.
- **footrule**: Computes Spearman's Footrule distance [9, 37], the underlying (but un-normalized measurement used in the non-parametric Spearman correlation. Formally, this is $\delta_{SF}(o_1, o_2) = \sum_{i=1}^{|C|} |\text{Pos}(o_1, c_i) - \text{Pos}(o_2, c_i)|$, the summed number of positions between where a candidate ought to be placed (i.e. the position of the candidate in sequence o_2), and where the candidate has actually been placed (i.e. its position in o_1). This is also similar to the Single Winner Error, but computed for every candidate, not just the winner. It is a less commonly used alternative for τ.
- **impMargin**: Computes the margin by which the winner picked by the imputation method loses to the true winner, in the ground truth elections. That is: $\delta_{IM}(o_1, S_2) = \arg\max_{c \in C} S_2(c) - S_2(o_1)$. Used to measure the margins of victory, which are predictors of performance for some imputation methods.
- **macroAvg**: Computes the average percentage error in the score of a candidate: $\delta_{mac}(S_1, S_2) = \frac{1}{|C|} \sum_{c \in C} |(S_1(c)/S_2(c)) - 1|$. Can be used to compare the average bias towards different candidates, independent of their popularity.
- **margin**: As **impMargin** above, but instead computes the margin by which the true winner has lost to the winner picked by the imputation method, in the imputation method's election. That is: $\delta_{Marg}(S_1, o_2) = \arg\max_{c\ inC} S_1(c) - S_1(o_2)$. Used to measure the margins of victory, which are predictors of performance for some imputation methods.
- **microAvg**: Computes the overall percentage error in the scores of candidates collectively:

$$\delta_{mic}(S_1, S_2) = \frac{\sum_{c \in C} |S_1(c) - S_2(c)|}{\sum_{c \in C} S_1(c)}$$

Can be used to compare the total error in the imputation, but will favour methods that are more accurate on popular candidates (which will have much higher scores) even if they are less accurate on unpopular candidates (which will have smaller scores).

- **singleWinner**: Computes the Single Winner Error for the imputation method: the rank of the winner chosen by the imputation method in the ground truth election. $\delta_w(o_1, o_2) = |\{c \in C | c\ o_2\ o_1(1)\}|$, where $o_1(1)$ is the candidate ranked first in the imputation method's ordering.
- **tau**: Computes the Kendall Correlation τ between the two outcome orderings [26].

$$\tau(o_1, o_2) = \frac{\sum_{c_i \in C} \sum_{c_j \in C \setminus c_i} I((c_i\ o_1\ c_j)\ =\ (c_i\ o_2\ c_j) \wedge (c_j\ o_1\ c_i)\ =\ (c_j\ o_2\ c_i))}{|C|(|C|-1)}$$

where I is a binary indicator variable. This measures the fraction of pairwise comparisons on which the two orders agree, and thus facilitates the evaluation of the ability for competing methods to recover the overall ordering of the candidates, rather than just determining the winner.

- **weightedFootrule**: A variant of **footrule** above, where errors are weighted by the (correct) position of each candidate. This effectively penalizes methods that make mistakes in highly ranked candidates, and benefits those that make mistakes only in the ordering of lower ranked candidates. Formally: $\delta_{SF}(o_1, o_2) = \sum_{i=1}^{|C|} |\mathrm{Pos}(o_1, c_i) - \mathrm{Pos}(o_2, c_i)| \times \mathrm{Pos}(o_2, c_i)$.
- **weightedTau**: A variant of **tau** above, where errors are weighted by the position of each candidate.

$$\tau(o_1, o_2) =$$

$$2\frac{\sum_{(c_i,c_j) \in (C \times C), i \neq j} I((c_i\ o_1\ c_j) = (c_i\ o_2\ c_j) \wedge (c_j\ o_1\ c_i) = (c_j\ o_2\ c_i)) \times \mathrm{Pos}(o_1, c_i)}{|C|(|C|-1)(|C|+1)}$$

As with the other features of Prefmine, the set of performance measures is easily extended. A performance measure must accept a mapping of candidates to scores, as output by a voting rule, and must output a scalar value. The code base readily facilitates conversion between scores and orderings. After writing a new performance measure, adding it to the user interface simply entails adding its name to a list, which will be used to populate the Performance Metric selection box at runtime.

2.8. **Sample output.** Tables 1 and 2 show example outputs from an experiment performed with the Prefmine system to compare two competing approaches to group decision making with incomplete information. All of the sets from the Debian and Irish groupings of Preflib.org were selected as datasets, with the Imputation Modes selected were logres and mmr. The Copeland voting rule was used, and the firstError, singleWinner, and tau performance metrics were selected. 1600 runs were performed on each of the Debian sets, while only 400 were performed on the larger Irish ones. Prefmine outputs a latex formatted table for each method, and the resulting tables are shown below.

2.9. **Extending the system.** Prefmine is envisioned as an experimental platform for other researchers to extend, facilitating the application of machine learning algorithms to the extensive Preflib repository. At present there are no plans to add further imputation methods, voting rules, or performance measures to Prefmine, although these additions would be easily accomplished. Instead, future work will

	First Error	Single Winner	τ	—C—	% Missing
Debian 2002	4.00 ± 0.00	0.00 ± 0.00	1.00 ± 0.00	4	11.9
Debian 2003	4.94 ± 0.49	0.01 ± 0.12	0.99 ± 0.07	5	13.6
Debian 2005	7.00 ± 0.00	0.00 ± 0.00	1.00 ± 0.00	7	15.5
Debian 2006	6.35 ± 0.76	0.00 ± 0.00	0.94 ± 0.03	8	14.8
Debian 2007	9.00 ± 0.00	0.00 ± 0.00	0.90 ± 0.08	9	19.1
Debian 2010	5.00 ± 0.00	0.00 ± 0.00	1.00 ± 0.00	5	11.0
Debian 2012	4.00 ± 0.00	0.00 ± 0.00	1.00 ± 0.00	4	13.2
Debian Logo	5.18 ± 1.36	0.00 ± 0.00	0.75 ± 0.11	8	40.0
Dublin North	4.01 ± 0.07	0.00 ± 0.00	0.85 ± 0.01	12	58.5
Dublin West	3.85 ± 1.88	0.82 ± 0.38	0.81 ± 0.06	9	50.8
Meath	1.00 ± 0.00	3.54 ± 0.32	0.74 ± 0.02	14	66.8

Table 1. Table showing the mean firstError, singleWinner, and tau measures for the instantiation of the imputation-based approach using *logistic regression* on the **Copeland** social choice function. Reported values are the mean over many problem instances, and reported measurement errors are the sample standard deviations. The rightmost columns report the number of candidates, and the percentage of missing information in each set.

	First Error	Single Winner	τ	—C—	% Missing
Debian 2002	4.00 ± 0.00	0.00 ± 0.00	1.00 ± 0.00	4	11.9
Debian 2003	4.40 ± 0.84	0.02 ± 0.09	0.90 ± 0.11	5	13.6
Debian 2005	6.57 ± 1.54	0.04 ± 0.13	0.99 ± 0.02	7	15.5
Debian 2006	6.00 ± 0.10	0.00 ± 0.00	0.93 ± 0.00	8	14.8
Debian 2007	2.05 ± 0.37	0.00 ± 0.00	0.87 ± 0.03	9	19.1
Debian 2010	3.04 ± 1.39	0.00 ± 0.00	0.82 ± 0.14	5	11.0
Debian 2012	4.00 ± 0.00	0.00 ± 0.00	1.00 ± 0.00	4	13.2
Debian Logo	5.44 ± 1.00	0.00 ± 0.00	0.74 ± 0.12	8	40.0
Dublin North	3.98 ± 0.14	0.00 ± 0.00	-0.09 ± 0.03	12	58.5
Dublin West	3.00 ± 0.00	0.72 ± 0.13	0.52 ± 0.05	9	50.8
Meath	2.97 ± 0.17	0.00 ± 0.00	-0.46 ± 0.04	14	66.8

Table 2. Table showing the mean firstError, singleWinner, and tau measures for the *Minimax Regret* approach on the **Copeland** social choice function. Reported values are the mean over many problem instances, and reported measurement errors are the sample standard deviations. The rightmost columns report the number of candidates, and the percentage of missing information in each set.

focus on adding additional features to the software, to further streamline its application and to improve runtimes.

At present, users of Prefmine may add new approaches written in the D programming language directly. C and C++ code can also be linked directly to Prefmine, but requires users to write a simple interface file. Automating the construction of such an interface would broaden the appeal of Prefmine significantly, as C and C++ are much more commonly used languages.

Prefmine also lacks graphing capabilities, requiring users transport their results to external software in order to visualize the differences between the different methods. Adding in direct support for graphing through a third-party application like the `plot.ly` cloud service or Python's matplotlib could allow users to quickly compare the results of different methods, and perhaps even construct custom graphs in inside a Prefmine instance.

Finally, Prefmine currently lacks proper parallel processing facilities, and consequently is much slower than it ought to be. While in the short term this will be addressed by enabling parallel repetitions of the experiment, in the long run general purpose frameworks like OpenCL [38] could be incorporated into Prefmine, allowing much faster computation.

3. Lessons for practitioners.

The previous section described the Prefmine experimental testbed. The testbed was carefully constructed to minimize the potential for experimental error, and to streamline the addition of new imputation methods, voting rules, and performance measures. Prior to the creation of Prefmine, experiments were performed using a multi-part experimental framework, which performed many of the same steps, but which also suffered from systematic problems with the replication and storage of results and data, parallel processing of data, and extensions of the system to incorporate additional techniques. Despite this, these earlier systems provided a number of important findings that were later incorporated into Prefmine, particularly the choice of learning algorithms available in Prefmine (logistic regression and two SVM variants), as well as the internal algorithm for feature selection (information gain with a fixed feature set size). This section describes first the deficiencies of the earlier experimental setup that were solved by Prefmine (briefly), and then the experiment design and results that were used to select the learning and feature selection algorithms used within the Prefmine system.

3.1. Experimental robustness.

A major advantage of Prefmine over the earlier system is the experimental robustness. The earlier system relied on a large number of modular programs written in three languages, and invoked by a series of interconnecting scripts. The result was a brittle system, where changes in one component often produced subtle errors in other, distantly related, parts. The lack of a comprehensive testing framework meant that often errors were uncovered only longer after their introduction, invalidating a large number of earlier runs. Prefmine overcomes this deficiency by combining a plugin-style architecture with integrated unit tests in the core experimental framework, incorporating the lessons learned during the initial system's development. The plugin-style architecture is also the feature that facilitates Prefmine's extensability, allowing it to support many times more experimental settings than the earlier system, and to easily integrate yet more.

The earlier system also taught important lessons regarding data integrity. Since the system worked on locally cached copies of the Preflib datasets, and since the introduction of new features frequently produced incorrect behaviour in unrelated parts of the system, data were often corrupted without the experimenter's knowledge. Eventual discovery of these errors necessitated rerunning earlier experiments, and dramatically increased development times. In contrast, Prefmine stores no data locally: fresh copies are procured from the Preflib repository at the start of each experiment. Additionally, the first step in any experiment is to render the stored copied of the data immutable (i.e. read-only). The last step is for all results to be written out into an immutable database. This prevents systematic corruption of the data, and further de-couples the different components of the system. Each algorithm is provided with its own read-write copy of a problem instance, which ensures that when a new algorithm is developed, the benchmark results from older ones are not influenced, even if the new algorithm is corrupting its own data. As a result, development times are greatly reduced, and adding new features to the system has taken less (rather than more) time as the capabilities of the framework have expanded.

The earlier system also provided lessons in the importance of integrating parallelism directly into the experimental system, rather than adding it piecemeal throughout its components. The original system frequently suffered race conditions resulting from the use of multiple components accessing the same (mutable) datasets simultaneously, and the only parallelism that was manageable in later versions of the system was through entirely separate installations that were started as separate processes. In contrast, Prefmine was implemented in a language that supported extensive static analysis, immutable data types, and simple parallel structures (D), which facilitated simple, error free, parallel processing in earlier Prefmine versions (though not in the current version).

Although these lessons seem relatively obvious in retrospect, they constitute an important reason for future experimenters to consider using (or extending) Prefmine, rather than constructing their own testbeds. At present there is no unified testbed for use with Preflib, and correctly managing the data can be a laborious process, prone to many of the issues outlined above. By extending Prefmine, practitioners instead can jump straight to the implementation of their new algorithm or voting rule, confident in the integrity and efficiency of the resulting experiment design.

3.2. **Feature selection and algorithm choice.** Although the earlier system was error-prone and laborious to construct, it was used for a number of initial experiments on the imputation-based approach to social choice, some of which have not since been replicated with the more reliable Prefmine system. The experiment most likely to be of interest to readers is one that set out to answer the questions "Which conventional machine learning algorithm is best suited to predicting voters' unstated preferences?" and "How large should the feature set for the imputation-based approach to social choice need to be, and how should it be selected?". The results from this experiment later influenced the choice of classification algorithms, feature sets, and feature selection algorithms used in Prefmine.

To answer these questions, an experiment design much like the one used in Prefmine was adopted. 10 datasets were selected (the eight Debian sets from Preflib, and Dublin North and West from the Irish sets). These sets were picked because they represent real-world preferences that humans expressed as top-orders for use in a ranked ballot voting system. The details of the datasets in question are provided in [12], but basically these are data generated from real-world elections held in three Irish counties in 2002, and a number of leadership elections held for the Debian Society over the years 2002–2012. Problem instances were generated by discarding incomplete ballots, and then ablating the completed ones in a fashion consistent with the distribution of missingness in the original dataset, exactly as is done in Prefmine's standard ablation mode. Only the **Borda** voting rule was considered in these experiments.

The experiments of [12] measured performance in terms of the correctness of the outcomes, which was a sensible choice for comparing dissimilar approaches like

MMR and the imputation-based approach. However, since this is essentially comparing competing variants of the same approach (i.e. variants of the imputation-based approach with different classifiers used), a different set of measurements were adopted. Two measurements were used. The first was the error in the **Borda** scores of the candidates after the imputation took place, a measurement that amounts to the microAvg setting in Prefmine, discussed above. Let S_1 and S_1 be vectors indexing candidates to Borda scores (i.e. positive integers). The Borda Count Error (BCE) of S_1 with respect to S_2 is given by:

$$\mathrm{BCE}(S_1, S_2) = \frac{\sum_{c \in C} |S_1(c) - S_2(c)|}{\sum_{c \in C} S_2(c)}$$

If the Borda scores (i.e. average ballot positions) are computed for an imputed preferences of voters, and for the voters' true preferences, then the BCE of the scores for the imputed preferences with respect to the scores for the true preferences is a measure of how *accurately* the imputation method has predicted the preferences of voters. Note that there exist other methods, like simply counting the error rates of the classification methods, that could be adopted instead. However, a classifier making errors that cancel out in aggregate is generally preferable to one with a lower absolute error rate making errors that do not cancel out, a concept captured nicely by looking at the errors in the aggregate totals, rather than on individual ballots. For succinctness, $\mathrm{BCE}(S_1, S_2)$ will be written simply as BCE, denoting the error between the aggregates of a method and the ground truth preferences.

Related to this preference for methods that are less *biased* in the mistakes they make, the second measurement used in this experiment is the popularity bias of the methods, a feature that was found to be problematic in some earlier, smaller scale, experiments. In particular, earlier experiments indicated that imputation-based methods tended to under-estimate the aggregate scores for unpopular candidates, and overestimate them for popular ones. If a method tends to make errors that penalize unpopular candidates, then even if it performs well, its use might be questionable when the goal is to make "fair" decisions, because candidates will be treated unequally *by the system*, not merely by voters. Of course, this is a matter of degree. Certainly unpopular candidates may also view **Plurality** (i.e. the common electoral system when voters simply name their first preference, and the most commonly named preference wins) as a system that discriminates against them, though in a rather different manner.

In some cases bias of this kind may be a desirable feature: candidates none of the voters know much about may after all be a poor choice. However, in other applications the fact that voters do not know much about a given alternative is coincidental. For example, in the case of the robotic mining swarm, firms that do not know about a given region may do so because they are recent entrants to the area, not because the region is a fringe or undesirable alternative. In any case, methods with less bias clearly ought to be preferred to those with more, since the former are usable in a broader class of applications. Formally, bias is defined as the Pearson Correlation between the true (i.e. "correct") scores of a candidate, and the error in the candidate's score:

$$\mathrm{Bias}(S_1, S_2) = \mathrm{cor}(S_1 - S_2, S_2)$$

that is, the correlation between the elementwise difference (n.b. not the absolute value of the difference) of the two scoring vectors, and the second scoring vector.

Again, Bias is frequently used without arguements to denote Bias(S_1, S_2), where S_1 is the Borda scores of the candidates under the imputed preferences, and S_2 is the true Borda scores of the candidates.

The experiment compared three different imputation algorithms, each under three different feature selection treatments. The three algorithms were a multinomial logistic regression model, a support vector machine, and a Naive Bayes model. The multinomial regression model works much like logistic regression, but with a multinomial output instead of a binary one. It trains a single multinomial log-linear model using a neural network from the nnet package in R [35] The model has no important parameters to tune, and operates more or less automatically. Predictions from the model were generated with the predict function in R, using the undocumented type="prob" argument to provide a distribution over all the classes for each record.

The second model was a simple Naive Bayes classifier, taken from the e1071 package in R [11]. The model was configured to use Laplace smoothing with value 1, and predicted new values using the R predict function, with the type="raw" argument. To ensure the creation of a valid model, several additional preprocessing steps were performed before providing data to Naive Bayes. First, a second copy of every record was added, to avoid the situation where a class has only one example, which produced strange behaviours. Second, a very small amount of uniformly random noise ($\in \pm 10^-4$) was added to every feature for every ballot. This avoided the situation where attributes had nonzero variance overall, but zero variance when conditioned on a particular class, which produced unhandled exceptions in the model. We selected this model because of its extreme simplicity, and also because of the intuitive notion that models based on effectively counting the votes were likely to produce a good performance. This thought was eventually borne out in the Markov Tree models, though in a rather more principled way, by designing a model that learns patterns in the votes by counting them, but with a clearer semantic meaning to the counts.

The final model was the support vector machine (SVM), using the libsvm implementation [5]. The standard svm-train and svm-predict tools were wrapped with a Perl script. The script performed a search over the SVM parameter space for each training set, and then produced a model using the parameterization with the highest cross validation accuracy. The search used the polynomial (degree 3), RBF, and sigmoid kernels. For each kernel, the search performed a grid search over the parameter space of $C \in (2^{-5}, 2^{15})$ and $\gamma \in (2^{-15}, 2^{-5})$, in steps of 2^4 . A large step size was used because of limited computational resources. Parameter selection used a randomly selected subset of 500 data points to improve run times. Additionally, the Perl script monitored the polynomial kernel, which failed to converge on certain datasets. Experimentation with the parameter ranges did not change this behaviour, so the script was configured to stop searching with the polynomial kernel after the first instance where it failed to converge on any given dataset. This provided approximately a twenty-fold decrease in run times for the SVM. Overall, this method served as the inspiration for the **svm** Imputation Method in Prefmine (not **binsvm**). There are slight differences in the Prefmine implementation however.

Since all three learning algorithms were to be used in the imputation-based approach to social choice, features over the ballot needed to be constructed. Features included the rank (position) of each candidate on the ballot, if known; a three valued indicator variable for each pair of candidates (i, j), indicating whether i appeared before j, after j, or the ordering of the two candidates was unknown; and a set of variables indicating the magnitude of the difference in position of two candidates on the voter's ballot. For instance, if candidate i is the voter's fifth preference, and candidate j is the voter's eighth preference, then the distance between them was $5 - 8 = -3$. Features were generated using a number of Perl and bash scripts that operated over the ballots. Files containing the resulting features were stored for later use.

Two alternative feature sets were generated as well. The first set was constructed by running principal component analysis (PCA) on the original feature set [13]. PCA is a dimensionality reduction technique, in which the original feature set is compressed into a smaller set of features, each of which is a linear combination of the original feature set. Using PCA allows much (often most) of the original feature set's information to be represented in a smaller set of features, which can in turn reduce the runtime of classification algorithms applied to the data.

Before applying PCA to a dataset represented using the original feature set, values that indicated missing data were replaced with the mean of the column in which they appeared. This is a standard stem to avoid bias. If a column was entirely comprised of missing values, or if it had variance 0, it was removed entirely prior to performing PCA, to avoid faults in the PCA implementation that was used. For each training file, all components with magnitudes at least 10% of the first principal component (i.e. containing at least 10% of the information content of the most informative component) were captured, and created an alternative file containing only the reconstructions of each row within the resulting subspace. The PCA calls subtracted the mean from each column automatically, and also normalized the data prior to computing the components. R was used [39] for the preprocessing described in this step, and the R `prcomp` implementation of PCA was then applied.

The second feature set was obtained by applying an information gain filter to the data, and selecting just the ten most predictive features under this metric, using the FSelector package in R [36], and with the same preprocessing steps as the PCA feature selection. In a classification context, the information gain of a feature is the change in entropy produced by partitioning the data on that feature. It is based on the Kullback-Leibler divergence measure [27]. Let $H(Y)$ be the overall entropy a dataset with features X and labels Y[8]. Then the information gain of feature X_i is:

$$\mathrm{IG}(X_i, X, Y) = H(Y) - \sum_{v \in \mathrm{values}(X_i)} \frac{|\{x_j \in X | X_{ij} = v\}|}{|X|} H(\{y \in Y | X_{ij} = v)\})$$

The 10 original features with the highest information gain were selected and used as the third feature treatment. This constant size set of features was included to explore the possibility of speeding up the machine learning algorithms with a greatly reduced version of the original feature space, rather than trying to construct new features as with the PCA approach. An advantage of the information gain approach is that, since the original features already had clear human-interpretable meanings, models learned from subsets of this set of feature ought to be easy to interpret as well. In contrast, models learned using the PCA feature set might be more difficult to interpret. As a final step before providing data to a classifier, any remaining missing values were imputed with the mean of the corresponding column. The

[8]i.e. a measure of how much variety there is in the labels of the dataset: higher if the dataset is split more or less evenly among all the classes, lower if one class dominates the set.

data were also centered and normalized, and any columns with 0 variance were dropped. These steps were taken to ensure that the classifiers were able to process the data, as the support vector machine proved especially temperamental when given datasets that violated any of the mentioned properties. These techniques were automatically integrated into the support vector machine implementation that appears in Prefmine.

Each of the three algorithms (SVM, Naive Bayes, and Multinomial Logistic Regression) was run under each of the three feature treatments for each of the ten datasets considered. Every combination of dataset, feature set and algorithm was run with ten different problem instances. This small number of replications was used because the system was very slow (especially when PCA was used, and because of the parameter selection component of the SVM). Performance is summarized in Tables 3 and 4. Bold values indicate the best performance under each dataset. If there are multiple bolded values for a single set, this indicates that the two values are statistically indistinguishable. To compare measurements between two machine learning methods, a Student's t-test was used over the paired differences between the value of the measurements for the two methods on the same training and test data, with the null hypothesis that the mean difference between the measurements is zero. It was not immediately clear that the assumptions required for use of the t-test would be satisfied with such a small dataset. In particular, the normality assumption was not assured. To mitigate this, a Shapiro-Wilk test was applied to the distribution of differences, prior to analysis, and a non-parametric test was used instead if the data were not consistent with a normal distribution.

In terms of the Borda Count Error, the combination of the SVM and information gain feature selection method appears best. The SVM has statistically significantly better performance than the other two methods on every set except Debian 2002 and Debian 2012, which were both shown to have minimal room for errors in any case in [12]. The SVM performance using the information gain selected features was generally at least as good as using the other two sets, and on Dublin North and Debian 2003, was statistically better. This was true despite the information gain feature set requiring the lowest runtimes overall, on account of having the smallest feature spaces.

The Bias results in Table 4 provide an interesting contrast. The non-SVM methods consistently exhibit lower bias than the SVM across the easy Debian sets, though they are not statistically different on the Debian Logo and Dublin North sets. The SVM results are better on Debian West. Bias also appears to be slightly worse for the SVM when used with the information gain feature sets. It appears the SVM+IG combination is imputing with a bias for more popular candidates, which leads to a more accurate imputation over all. It is also interesting to note that Dublin North in general exhibits much higher bias values than any of the other sets. Performance when ordering the less popular candidates on this set was also lower for the imputation-based approach than for other methods in the experiments of [12]. It appears that the patterns present in the data lead methods to impute more popular candidates. This is further illustrated by measuring the diversity of the sets of ballots in Dublin North and Dublin West, shown in Figure 11. The ballots of the Dublin North set exhibit much more diversity of opinion, meaning more patterns must be learned to impute them accurately.

Overall, these results suggested only a small number of features were needed to ensure good performance from machine learning models, and that using information

	SVM			NB			MN		
	PCA	IG	plain	PCA	IG	plain	PCA	IG	plain
Deb. 2002	0.008	0.007	0.008	**0.003**	**0.003**	**0.003**	**0.003**	**0.003**	**0.003**
Deb. 2003	0.009	**0.005**	0.009	0.009	0.009	0.009	0.009	0.009	0.009
Deb. 2005	**0.014**	**0.013**	**0.016**	0.031	0.031	0.031	0.031	0.031	0.031
Deb. 2006	**0.013**	**0.013**	**0.015**	0.031	0.031	0.031	0.031	0.031	0.031
Deb. 2007	**0.033**	**0.031**	**0.033**	0.043	0.043	0.043	0.043	0.043	0.043
Deb. 2010	**0.004**	**0.004**	**0.004**	0.005	0.005	0.005	0.005	0.005	0.005
Deb. 2012	0.011	0.011	0.011	**0.003**	**0.003**	**0.003**	**0.003**	**0.003**	**0.003**
Deb. Logo	**0.006**	**0.008**	**0.006**	0.011	0.011	0.011	0.011	0.011	0.011
North	1.084	**0.923**	1.08	1.546	1.546	1.546	1.546	1.546	1.546
West	**0.549**	**0.458**	0.575	0.654	0.654	0.654	0.654	0.654	0.654

TABLE 3. A summary of the results from the preliminary experiment comparing feature selection methods and classifiers for use with the imputation based approach to resolving social choice with incomplete information. Performance is reported under the Borda Error measure, which is related to the classifier's accuracy in imputing ballots. Performance which is statistically indistinguishable from the best on any given dataset has been rendered in bold.

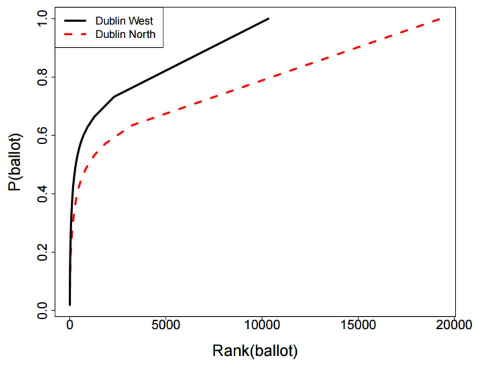


FIGURE 11. Empirical cumulative density functions for unique ballots in Dublin North and Dublin West. The x-axis shows the ranking of ballots from most to least common. The y-axis shows the cumulative proportion of voters who cast each ballot.

gain to select them was a reasonable approach. As a result, all classifier-based imputation methods in Prefmine use information gain feature selection, though with a value of 30 rather than 10, since this is still adequately fast for most applications, and provides a slight performance boost on some sets. SVM was added to Prefmine, while the Naive Bayes classifier and the multinomial model were not, on the basis of these performance results.

4. **Summary.** This paper described the Prefmine system, a general testbed for evaluating different approaches to the problem of social choice with partial information. The testbed system implements many different approaches to this problem, many different voting rules, and many different performance measures, allowing

	SVM			NB			MN		
	PCA	IG	plain	PCA	IG	plain	PCA	IG	plain
Deb. 2002	0.151	0.155	0.159	0.109	0.109	0.109	0.109	0.109	0.109
Deb. 2003	0.088	**0.030**	0.051	**0.009**	**0.009**	**0.009**	**0.009**	**0.009**	**0.009**
Deb. 2005	0.255	0.182	0.309	**0.031**	**0.031**	**0.031**	**0.031**	**0.031**	**0.031**
Deb. 2006	0.171	**0.040**	0.203	**0.011**	**0.011**	**0.011**	**0.011**	**0.011**	**0.011**
Deb. 2007	0.280	**0.170**	0.296	**0.065**	**0.065**	**0.065**	**0.065**	**0.065**	**0.065**
Deb. 2010	**0.078**	0.162	**0.064**	**0.099**	**0.099**	**0.099**	**0.099**	**0.099**	**0.099**
Deb. 2012	0.34	0.35	0.32	**0.12**	**0.12**	**0.12**	**0.12**	**0.12**	**0.12**
Deb. Logo	0.056	0.034	0.026	0.007	0.007	0.007	0.007	0.007	0.007
North	0.331	0.323	0.322	0.391	0.391	0.391	0.391	0.391	0.391
West	**0.020**	0.101	**0.026**	0.041	0.041	0.041	0.041	0.041	0.041

Table 4. A summary of the results from the preliminary experiment comparing feature selection methods and classifiers for use with the imputation based approach to resolving social choice with incomplete information. Performance is reported under the Bias measure, which is the correlation between the classifier's error in imputing a given candidate and the popularity of that candidate. Performance which is statistically indistinguishable from the best on any given dataset has been rendered in bold.

practitioners to easily evaluate algorithms on their own datasets. Additionally, the system's plugin-style architecture allows future experimenters to easily add a new approach to Prefmine, and then compare it with existing methods, without needing to implement their own error-prone code to obtain, pre-process, or evaluate data. Prefmine also provides intuitive and easy to read results summaries for the user, and efficiently makes use of parallel computing resources.

In addition to showcasing Prefmine, the paper contains a detailed description of the system, including descriptions of all user settings, and pictures illustrating how to use Prefmine. An initial study described near the end of the paper demonstrates the benefits of many of Prefmine's features, and provides insights into the design of Prefmine, and why future users might prefer using it to creating their own systems.

Our work can be viewed as enabling information analytics for the application of elections (and for any environment where group decision making among agents faces the challenge of addressing missing data values in voters' preferences).

REFERENCES

[1] H. Azari, D. Parkes and L. Xia, Random Utility Theory for Social Choice, in *Advances in Neural Information Processing Systems*, NIPS Foundation, 2012, 126–134.

[2] A. Balz and R. Senge, WEKA-LR: A Label Ranking Extension for WEKA, URL `http://www.uni-marburg.de/fb12/kebi/research/software/labelrankdoc.pdf`.

[3] S. Bouveret, Whale3 - WHich ALternative is Elected, URL `http://strokes.imag.fr/whale3/`.

[4] F. Brandt, G. Chabin and C. Geist, Pnyx: A Powerful and User-friendly Tool for Preference Aggregation, in *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, International Foundation for Autonomous Agents and Multiagent Systems, 2015, 1915–1916.

[5] C.-C. Chang and C.-J. Lin, LIBSVM: a library for support vector machines, *ACM Transactions on Intelligent Systems and Technology (TIST)*, **2** (2011), 27–66.

[6] G. Charwat and A. Pfandler, Democratix: A Declarative Approach to Winner Determination, in *Algorithmic Decision Theory*, Springer, 2015, 253–269.

[7] C. Cortes and V. Vapnik, Support-vector networks, *Machine Learning*, **20** (1995), 273–297.

[8] J. Dean and S. Ghemawat, MapReduce: simplified data processing on large clusters, *Communications of the ACM*, **51** (2008), 107–113.

[9] P. Diaconis and R. L. Graham, Spearman's footrule as a measure of disarray, *Journal of the Royal Statistical Society. Series B (Methodological)*, 262–268.

[10] J. P. Dickerson, A. D. Procaccia and T. Sandholm, Price of fairness in kidney exchange, in *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, International Foundation for Autonomous Agents and Multiagent Systems, 2014, 1013–1020.

[11] E. Dimitriadou, K. Hornik, F. Leisch, D. Meyer and A. Weingessel, Misc functions of the Department of Statistics (e1071), TU Wien, *R package*, **1** (2008), 5–24.

[12] J. A. Doucette, K. Larson and R. Cohen, Conventional machine learning for social choice, in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI Press, 2015.

[13] G. H. Dunteman, *Principal Components Analysis*, no. 69 in Quantitative Applications in the Social Sciences, Sage, 1989.

[14] V. E. Farrugia, H. P. Martínez and G. N. Yannakakis, The preference learning toolbox, *arXiv preprint arXiv:1506.01709*.

[15] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub and M. Schneider, Potassco: The Potsdam answer set solving collection, *AI Communications*, **24** (2011), 107–124.

[16] M. Gelfond and V. Lifschitz, Classical negation in logic programs and disjunctive databases, *New Generation Computing*, **9** (1991), 365–385.

[17] J. Goldman and A. D. Procaccia, Spliddit: Unleashing fair division algorithms, *ACM SIGecom Exchanges*, **13** (2015), 41–46.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann and I. H. Witten, The WEKA data mining software: an update, *ACM SIGKDD explorations newsletter*, **11** (2009), 10–18.

[19] L. Hatton, The T-experiments: errors in scientific software, in *Quality of Numerical Software*, Springer, 1997, 12–31.

[20] L. Hatton and A. Roberts, How accurate is scientific software?, *IEEE Transactions on Software Engineering*, **20** (1994), 785–797.

[21] M. R. Hestenes and E. Stiefel, Methods of Conjugate Gradients for Solving Linear Systems1, *Journal of Research of the National Bureau of Standards*, **49**.

[22] E. Hüllermeier and J. Fürnkranz, Learning from label preferences, in *Proceedings of the 14th International Conference on Discovery Science*, Springer, 2011, 2–17.

[23] T. Joachims, Optimizing search engines using clickthrough data, in *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 2002, 133–142.

[24] T. Joachims, Training linear SVMs in linear time, in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 2006, 217–226.

[25] D. Kelly and R. Sanders, The challenge of testing scientific software, in *In Proceedings of the 2008 Conference for the Association for Software Testing*, AST, 2008, 30–36.

[26] M. G. Kendall, A new measure of rank correlation, *Biometrika*, **30** (1938), 81–93.

[27] S. Kullback and R. A. Leibler, On information and sufficiency, *The Annals of Mathematical Statistics*, **22** (1951), 79–86.

[28] T. Lu and C. Boutilier, Learning Mallows models with pairwise preferences, in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, 145–152.

[29] T. Lu and C. Boutilier, Robust approximation and incremental elicitation in voting protocols, in *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence*, AAAI Press, 2011, 287–213.

[30] R. D. Luce, *Individual Choice Behavior: A Theoretical Analysis*, Wiley, 1959.

[31] C. L. Mallows, Non-null ranking models. I, *Biometrika*, **44** (1957), 114–130.

[32] N. Mattei and T. Walsh, Preflib: A library for preferences http://www. preflib. org, in *Algorithmic Decision Theory*, Springer, 2013, 259–270.

[33] N. Matti, PrefLib-Tools: A small and lightweight set of Python tools for working with and generating data from www.PrefLib.org., URL `https://github.com/nmattei/PrefLib-Tools`.

[34] R. Rifkin and A. Klautau, In defense of one-vs-all classification, *The Journal of Machine Learning Research*, **5** (2004), 101–141.

[35] B. Ripley and W. Venables, nnet: Feed-forward neural networks and multinomial log-linear models, *R package version*, **7**.

[36] P. Romanski, FSelector: Selecting attributes, *Vienna: R Foundation for Statistical Computing*.

[37] C. Spearman, 'Footrule' for measuring correlation, *British Journal of Psychology, 1904-1920*, **2** (1906), 89–108.

[38] J. E. Stone, D. Gohara and G. Shi, OpenCL: A parallel programming standard for heterogeneous computing systems, *Computing in Science & Engineering*, **12** (2010), 66–73.

[39] R. C. Team, *R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. 2013*, ISBN 3-900051-07-0, 2014.

[40] L. Xia and V. Conitzer, Determining Possible and Necessary Winners under Common Voting Rules Given Partial Orders., in *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, AAAI Press, 2008, 196–201.

E-mail address: `jdoucette@ncf.edu`
E-mail address: `rcohen@uwaterloo.ca`