R2B2

Release 0.1.0

R2B2 Team

Jul 06, 2021

CONTENTS

Round-by-Round and Ballot-by-Ballot election audits: a workbench for exploration of risk-limiting audits.

CHAPTER

ONE

R2B2

Round-by-Round and Ballot-by-Ballot risk limiting audit library.

1.1 Subpackages

1.1.1 r2b2.simulation

R2B2 Simulations

Submodules

r2b2.simulation.filip_athena

Athena/Minerva Simulations using Filip Zagorski's athena library.

Module Contents

class r2b2.simulation.filip_athena.**FZMinervaOneRoundRisk**(*alpha*, *reported*, *sample_size*, *election_file*, *reported_name*, *db_mode=True*, *db_host='localhost'*, *db_name='r2b2'*, *db_port=27017*, **args*, ***kwargs*)

Bases: *r2b2.simulator.Simulation*

Simulate a 1-round Minerva audit for a given sample size to compute risk limit.

trial(*self*, *seed*)
Execute a 1-round minerva audit from Filip's athena code.

analyze(*self*)
Analyze the simulation trials.

run(*self*, *n: int*)
Execute n trials of the simulation.

Executes n simulation trials by generating a random seed, running a trial with the given seed, and writing the trial entry to the trials collection.

Parameters **n** (*int*) – Number of trials to execute and write to database.

get_seed(*self*)
Generate a random seed.

Note: This method generates 8 random bytes using os sources of randomness. If a different source of randomness is desired, overwrite the method per implementation.

output(*self*, *fd: str = None*)
Write output of simulation to JSON file.

Parameters **fd** (*str*) – filename to write output to. If no file is passed, formatted JSON is simply printed.

output_audit(*self*)
Create audit output in JSON format.

Note: This functionality is separated into a method so specific audit implementations may override it and customize their output in non-database mode.

1.2 Submodules

1.2.1 r2b2.__main__

Entrypoint module, in case you use *python -mr2b2*.

Why does this file exist, and why __main__? For more info, read:

- https://www.python.org/dev/peps/pep-0338/
- https://docs.python.org/2/using/cmdline.html#cmdoption-m
- https://docs.python.org/3/using/cmdline.html#cmdoption-m

1.2.2 r2b2.athena

Athena audit module.

Module Contents

class r2b2.athena.**Athena**(*alpha: float*, *delta: float*, *max_fraction_to_draw: float*, *contest:* r2b2.contest.Contest)
Bases: *r2b2.audit.Audit*

Athena audit implementation.

An Athena audit is a type of risk-limiting audit that accounts for round-by-round auditor decisions. For a given sample size (in the context of a round schedule), the audit software calculates a minimum number of votes for the reported winner that must be found in the sample to stop the audit and confirm the reported outcome.

Variables

- **alpha** (*float*) – Risk limit. Alpha represents the chance that, given an incorrectly called election, the audit will fail to force a full recount.

- **max_fraction_to_draw** (*float*) – The maximum number of ballots the auditors are willing to draw as a fraction of the ballots in the contest.
- **delta** (*float*) – Delta value.

Initialize an Athena audit.

get_min_sample_size(*self*, *sub_audit:* r2b2.audit.PairwiseAudit, *min_sprob: float = 10 ** - 6*)
Computes the minimum sample size that has a stopping size (kmin). Here we find a practical minimum instead of the theoretical minimum (BRAVO's minimum) to avoid floating-point imprecisions in the later convolution process.

Parameters

- **sub_audit** (*PairwiseAudit*) – Compute minimum sample size for this subaudit.
- **min_sprob** (*float*) – Round sizes with below min_sprob stopping probability are excluded.

Returns *int* – The minimum sample size of the audit, adherent to the min_sprob.

satisfactory_sample_size(*self*, *left*, *right*, *sprob*, *num_dist*, *denom_dist*)
Helper method that returns True if the round size satisfies the stopping probability.

next_sample_size(*self*, **args*, ***kwargs*)
Generate estimates of possible next sample sizes.

Note: To be used during live/interactive audit execution.

stopping_condition_pairwise(*self*, *pair: str*, *verbose: bool = False*) → bool
Check, without finding the kmin, whether the audit is complete.

Parameters **pair** (*str*) – Dictionary key referencing pairwise subaudit. Evaluate the stopping condition for this subaudit.

Returns *bool* – Whether or not the pairwise stopping condition has been met.

next_min_winner_ballots_pairwise(*self*, *sub_audit:* r2b2.audit.PairwiseAudit) → int
Compute stopping size for a given subaudit.

Parameters **sub_audit** (*PairwiseAudit*) – Compute next stopping size for this subaudit.

Returns *int* – Stopping size for most recent round.

compute_min_winner_ballots(*self*, *sub_audit:* r2b2.audit.PairwiseAudit, *rounds: List[int]*, **args*, ***kwargs*)
Compute the minimum number of winner ballots for a round schedule.

Extend the audit's round schedule with the passed (partial) round schedule, and then extend the audit's minimum number of winner ballots schedule with the corresponding minimums to meet the stopping condition.

Parameters

- **sub_audit** (*PairwiseAudit*) – Compute minimum winner ballots for this Pairwise subaudit.
- **rounds** (*List[int]*) – A (partial) round schedule of the audit.

find_kmin(*self*, *sub_audit:* r2b2.audit.PairwiseAudit, *sample_size: int*, *append: bool*)
Search for a kmin (minimum number of winner ballots) satisfying all stopping criteria.

Parameters

- **sub_audit** (*PairwiseAudit*) – Find kmin for this subaudit.
- **sample_size** (*int*) – Sample size to find kmin for.

- **append** (*bool*) – Optionally append the kmins to the min_winner_ballots list. This may not always be desirable here because, for example, appending happens automatically outside this method during an interactive audit.

compute_all_min_winner_ballots(*self*, *sub_audit:* r2b2.audit.PairwiseAudit, *max_sample_size: int* = *None*, **args*, ***kwargs*)

Compute the minimum number of winner ballots for the complete (that is, ballot-by-ballot) round schedule.

Note: Due to limited convolutional precision, results may be off somewhat after the stopping probability very nearly equals 1.

Parameters

- **sub_audit** (*PairwiseAudit*) – Compute minimum winner ballots for this pairwise subaudit.
- **max_sample_size** (*int*) – Optionally set the maximum sample size to generate stopping sizes (kmins) up to. If not provided the maximum sample size is determined by max_frac_to_draw and the total contest ballots.

Returns None, kmins are appended to the min_winner_ballots list.

compute_risk(*self*, *votes_for_winner: int*, *pair: str*, **args*, ***kwargs*)

Return the hypothetical (Minerva) p-value if votes_for_winner were obtained in the most recent round.

get_risk_level(*self*)

Return the risk level of an interactive Athena audit.

Non-interactive and bulk Athena audits are not considered here since the sampled number of reported winner ballots is not available.

__repr__(*self*)

String representation of Audit class.

Note: Can (and perhaps should) be overwritten in subclass.

__str__(*self*)

Human readable string (i.e. printable) representation of Audit class.

Note: Can (and perhaps should) be overwritten in subclass.

---

current_dist_null(*self*)

Update distribution_null for each sub audit for current round.

_current_dist_null_pairwise(*self*, *sub_audit: PairwiseAudit*, *bulk_use_round_size=False*)

Update distribution_null for a single PairwiseAudit

Parameters

- **sub_audit** (*PairwiseAudit*) – Pairwise subaudit for which to update distribution.
- **bulk_use_round_size** (*bool*) – Optional argument used by bulk methods. Since the bulk methods do not sample ballots for candidates during the rounds, this flag simply uses the round schedule as the round draw (instead of the pairwise round draw) for updating the distribution. Default is False.

current_dist_reported(*self*)

Update distribution_reported_tally for each subaudit for current round.

_current_dist_reported_pairwise(*self*, *sub_audit: PairwiseAudit*, *bulk_use_round_size=False*)
Update dist_reported for a single PairwiseAudit.

Parameters

- **sub_audit** (*PairwiseAudit*) – Pairwise subaudit for which to update distriution.
- **bulk_use_round_size** (*bool*) – Optional argument used by bulk methods. Since the bulk methods do not sample ballots for candidates during the rounds, this flag simply uses the round schedule as the round draw (instead of the pairwise round draw) for updating the distribution. Default is False.

truncate_dist_null(*self*)
Update risk schedule and truncate null distribution for each subaudit.

_truncate_dist_null_pairwise(*self*, *pair: str*)
Update risk schedule and truncate null distribution for a single subaudit.

Parameters **pair** (*str*) – Dictionary key for subaudit (within the audit's subaudits) to truncate distribution and update risk schedule.

truncate_dist_reported(*self*)
Update stopping prob schedule and truncate reported distribution for each subaudit.

_truncate_dist_reported_pairwise(*self*, *pair*)
Update stopping prob schedule and truncate reported distribution for a single subaudit.

Parameters **pair** (*str*) – Dictionary key for subaudit (within the audit's subaudits) to truncate distribution and update stopping prob schedule.

__get_interval(*self*, *dist: List[float]*)
Get relevant interval [l, u] of given distribution.

Find levels l and u such that cdf(l) < tolerance and 1 - cdf(u) < tolerance. The purpose of this is to improve efficiency in the current_dist_* functions for audits without replacement where almost all of the hypergeometric distribution falls in a fraction of its range, i.e. between l and u.

Note: cdf() in this context does not require cdf(infinity) = 1, although the distribution should sum very closely to 1.

---

asn(*self*, *pair: str*)
Compute ASN as described in BRAVO paper for pair of candidates.

Given the fractional margin for the reported winner and the risk limit (alpha) produce the average number of ballots sampled during the audit.

Parameters **pair** (*str*) – Dictionary key referencing pairwise audit in audit's subaudits.

Returns *int* – ASN value.

execute_round(*self*, *sample_size: int*, *sample: dict*, *verbose: bool = False*) → bool
Execute a single, non-interactive audit round.

Executes 1 round of the audit, given its current state. If the audit is stopped, its state will not be modified.

Warning: This method assumes the audit is in the correct state to be executed. If multiple executions of a full audit will be run the same audit object, make sure to call reset on the audit object between full executions.

Parameters

- **sample_size** (*int*) – Total ballots sampled by the end of this round (cumulative).

- **sample** (*dict*) – Sample counts for each candidate by the end of this round (cumulative).

Returns *bool* – True if the audit met its stopping condition by this round.

run(*self*, *verbose: bool = False*)
Begin interactive audit execution.

Begins the interactive version of the audit. While computations for different audits will vary, the process for executing each one is the same. This provides a process for selecting a sample size, determining if the ballots found for the reported winner in that sample size meet the stopping condition(s), and if not continuing with the audit. As the audit proceeds, data including round sizes, ballots for the winner in each round size, and per round risk and stopping probability are stored.

_reset(*self*)
Reset attributes modified during run().

stopping_condition(*self*, *verbose: bool = False*) → bool
Determine if the audits stopping condition has been met.

The audit stopping condition is met if and only if each pairwise stopping condition is met.

next_min_winner_ballots(*self*, *verbose: bool = False*)
Compute next stopping size of given (actual) sample sizes for all subaudits.

1.2.3 r2b2.audit

Abstract module defining an Audit framework.

Module Contents

class r2b2.audit.**PairwiseAudit**(*sub_contest:* r2b2.contest.PairwiseContest)
Store audit information for pairwise comparison.

The PairwiseAudit class hold the audit data for a single pair of candidates.

Variables

- **sub_contest** (PairwiseContest) – Pairwise contest data for relevant pair of candidates.
- **min_sample_size** (*int*) – The smallest valid sample size. The minimum round size where kmin <= round
- **risk_schedule** (*List[float]*) – Schedule of risk assocaited with each previous round. Corresponds to tail of null distribution.
- **stopping_prob_schedule** (*List[float]*) – Schedule of stopping probabilities associated with each previous round. Corresponds to tail of reported tally distribution.
- **pvalue_schedule** (*List[float]*) – Schedule of pvalues associated with each previous round. Corresponds to ratio of risk and stopping probability.
- **distribution_null** (*Dict[str, List[float]]*) – Current distribution associated with a tied election for each pairwise subcontest.
- **distribution_reported_tally** (*Dict[str, List[float]]*) – Current distribution associated with reported tally for each pairwise subcontest.
- **min_winner_ballots** (*List[int]*) – List of stopping sizes (kmin values) for each round.
- **stopped** (*bool*) – Indicates if pairwise audit has stopped.

__repr__(*self*)
Return repr(self).

__str__(*self*)
Return str(self).

get_pair_str(*self*)
Get winner-loser pair as string used as dictionary key in Audit.

class r2b2.audit.**Audit**(*alpha: float*, *beta: float*, *max_fraction_to_draw: float*, *replacement: bool*, *contest:* r2b2.contest.Contest)

Bases: abc.ABC

Abstract Base Class to define a general Audit object type.

The Audit class is an abstract base class which defines the general structure and properties of a risk-limiting audit. Individual RLAs are subclasses of the Audit class.

Variables

- **alpha** (*float*) – Risk limit. Alpha represents the chance that given an incorrectly called election, the audit will fail to go to a full recount.
- **beta** (*float*) – the worst case chance of causing an unnecessary full recount. For many RLAs, beta will simply be set to 0 and will not appear to be a parameter.
- **max_fraction_to_draw** (*float*) – The maximum total number of ballots auditors are willing to draw during the course of the audit.
- **replacement** (*bool*) – Indicates if the audit sampling should be done with (true) or without (false) replacement.
- **rounds** (*List[int]*) – List of round sizes (i.e. sample sizes).
- **sample_ballots** (*Dict[str, List[int]]*) – Dictionary mapping candidates to sample counts drawn throughout audit. Sample counts are cumulative.
- **pvalue_schedule** (*List[float]*) – Schedule of pvalues for overall audit in each round. In each round, the overall pvalue is the maximum pvalue of all subaudits.
- **contest** (Contest) – Contest on which to run the audit.
- **sub_audits** (*Dict[str, PairwiseAudit]*) – Dict of PairwiseAudits wthin audit indexed by loser.
- **stopped** (*bool*) – Indicates if the audit has stopped (i.e. met the risk limit).

Create an instance of an Audit.

Note: This should only be called when initializing a subclass as the Audit class is an abstract class.

__repr__(*self*)
String representation of Audit class.

Note: Can (and perhaps should) be overwritten in subclass.

__str__(*self*)
Human readable string (i.e. printable) representation of Audit class.

Note: Can (and perhaps should) be overwritten in subclass.

`current_dist_null`(*self*)
Update distribution_null for each sub audit for current round.

`_current_dist_null_pairwise`(*self*, *sub_audit:* PairwiseAudit, *bulk_use_round_size=False*)
Update distribution_null for a single PairwiseAudit

Parameters

- **sub_audit** (*PairwiseAudit*) – Pairwise subaudit for which to update distribution.
- **bulk_use_round_size** (*bool*) – Optional argument used by bulk methods. Since the bulk methods do not sample ballots for candidates during the rounds, this flag simply uses the round schedule as the round draw (instead of the pairwise round draw) for updating the distribution. Default is False.

`current_dist_reported`(*self*)
Update distribution_reported_tally for each subaudit for current round.

`_current_dist_reported_pairwise`(*self*, *sub_audit:* PairwiseAudit, *bulk_use_round_size=False*)
Update dist_reported for a single PairwiseAudit.

Parameters

- **sub_audit** (*PairwiseAudit*) – Pairwise subaudit for which to update distriution.
- **bulk_use_round_size** (*bool*) – Optional argument used by bulk methods. Since the bulk methods do not sample ballots for candidates during the rounds, this flag simply uses the round schedule as the round draw (instead of the pairwise round draw) for updating the distribution. Default is False.

`truncate_dist_null`(*self*)
Update risk schedule and truncate null distribution for each subaudit.

`_truncate_dist_null_pairwise`(*self*, *pair: str*)
Update risk schedule and truncate null distribution for a single subaudit.

Parameters **pair** (*str*) – Dictionary key for subaudit (within the audit's subaudits) to truncate distribution and update risk schedule.

`truncate_dist_reported`(*self*)
Update stopping prob schedule and truncate reported distribution for each subaudit.

`_truncate_dist_reported_pairwise`(*self*, *pair*)
Update stopping prob schedule and truncate reported distribution for a single subaudit.

Parameters **pair** (*str*) – Dictionary key for subaudit (within the audit's subaudits) to truncate distribution and update stopping prob schedule.

`__get_interval`(*self*, *dist: List[float]*)
Get relevant interval [l, u] of given distribution.

Find levels l and u such that cdf(l) < tolerance and 1 - cdf(u) < tolerance. The purpose of this is to improve efficiency in the current_dist_* functions for audits without replacement where almost all of the hypergeometric distribution falls in a fraction of its range, i.e. between l and u.

Note: cdf() in this context does not require cdf(infinity) = 1, although the distribution should sum very closely to 1.

asn(*self*, *pair: str*)
Compute ASN as described in BRAVO paper for pair of candidates.

Given the fractional margin for the reported winner and the risk limit (alpha) produce the average number of ballots sampled during the audit.

Parameters **pair** (*str*) – Dictionary key referencing pairwise audit in audit's subaudits.

Returns *int* – ASN value.

execute_round(*self*, *sample_size: int*, *sample: dict*, *verbose: bool = False*) → bool
Execute a single, non-interactive audit round.

Executes 1 round of the audit, given its current state. If the audit is stopped, its state will not be modified.

Warning: This method assumes the audit is in the correct state to be executed. If multiple executions of a full audit will be run the same audit object, make sure to call reset on the audit object between full executions.

Parameters

- **sample_size** (*int*) – Total ballots sampled by the end of this round (cumulative).
- **sample** (*dict*) – Sample counts for each candidate by the end of this round (cumulative).

Returns *bool* – True if the audit met its stopping condition by this round.

run(*self*, *verbose: bool = False*)
Begin interactive audit execution.

Begins the interactive version of the audit. While computations for different audits will vary, the process for executing each one is the same. This provides a process for selecting a sample size, determining if the ballots found for the reported winner in that sample size meet the stopping condition(s), and if not continuing with the audit. As the audit proceeds, data including round sizes, ballots for the winner in each round size, and per round risk and stopping probability are stored.

_reset(*self*)
Reset attributes modified during run().

abstract get_min_sample_size(*self*, *sub_audit:* PairwiseAudit)
Get the minimum valid sample size in a sub audit

Parameters **sub_audit** (*PairwiseAudit*) – Get minimum sample size for this sub_audit.

abstract next_sample_size(*self*, **args*, ***kwargs*)
Generate estimates of possible next sample sizes.

Note: To be used during live/interactive audit execution.

stopping_condition(*self*, *verbose: bool = False*) → bool
Determine if the audits stopping condition has been met.

The audit stopping condition is met if and only if each pairwise stopping condition is met.

abstract stopping_condition_pairwise(*self*, *pair: str*, *verbose: bool = False*) → bool
Determine if pairwise subcontest meets stopping condition.

Parameters **pair** (*str*) – Dictionary key referencing pairwise audit in audit's sub_audits.

Returns *bool* – If the pairwise subaudit has stopped.

next_min_winner_ballots(*self*, *verbose: bool = False*)
Compute next stopping size of given (actual) sample sizes for all subaudits.

abstract next_min_winner_ballots_pairwise(*self*, *sub_audit:* PairwiseAudit) → int
Compute next stopping size of given (actual) sample size for a subaudit.

Parameters **sub_audit** (*PairwiseAudit*) – Compute next stopping size for this subaudit.

Note: To be used during live/interactive audit execution.

abstract compute_min_winner_ballots(*self*, *sub_audit:* PairwiseAudit, *progress: bool = False*, **args*, ***kwargs*)
Compute the stopping size(s) for any number of sample sizes for a given subaudit.

abstract compute_all_min_winner_ballots(*self*, *sub_audit:* PairwiseAudit, *progress: bool = False*, **args*, ***kwargs*)
Compute all stopping sizes from the minimum sample size on for a given subaudit.

abstract compute_risk(*self*, *sub_audit:* PairwiseAudit, **args*, ***kwargs*)
Compute the current risk level of a given subaudit.

Returns Current risk level of the audit (as defined per audit implementation).

abstract get_risk_level(*self*, **args*, ***kwargs*)
Find the risk level of the audit, that is, the smallest risk limit for which the audit as it has panned out would have already stopped.

Returns *float* – Risk level of the realization of the audit.

1.2.4 r2b2.brla

Bayesian Risk-Limiting Audit module.

Module Contents

class r2b2.brla.**BayesianRLA**(*alpha: float*, *max_fraction_to_draw: float*, *contest:* r2b2.contest.Contest, *reported_winner: str = None*)
Bases: *r2b2.audit.Audit*

Baysian Risk-Limiting Audit implementation.

A Bayesian Risk-Limit Audit implementation as defined by Vora, et. al. for auditing 2-candidate plurality elections. For a given sample size, the audit software calculates a minimum number of votes for the reported winner that must be found in the sample to stop the audit and confirm the reported outcome.

Variables

- **alpha** (*float*) – Risk limit. Alpha represents the chance, that given an incorrectly called election, the audit will fail to force a full recount.
- **max_fraction_to_draw** (*int*) – The maximum total number of ballots auditors are willing to draw during the course of the audit.
- **rounds** (*List[int]*) – The round sizes used during the audit.
- **contest** (Contest) – Contest to be audited.
- **prior** (*np.ndarray*) – Prior distribution for worst-case election.

Initialize a Bayesian RLA.

get_min_sample_size(*self*, *sub_audit:* r2b2.audit.PairwiseAudit)
Get the minimum valid sample size in a sub audit

Parameters **sub_audit** (*PairwiseAudit*) – Get minimum sample size for this sub_audit.

__str__(*self*)
Human readable string (i.e. printable) representation of Audit class.

Note: Can (and perhaps should) be overwritten in subclass.

stopping_condition_pairwise(*self*, *pair: str*, *verbose: bool = False*) → bool
Determine if pairwise subcontest meets stopping condition.

Parameters **pair** (*str*) – Dictionary key referencing pairwise audit in audit's sub_audits.

Returns *bool* – If the pairwise subaudit has stopped.

next_min_winner_ballots_pairwise(*self*, *sub_audit: r2b2.audit.PairwiseAudit*, *sample_size: int = 0*) → int
Compute the stopping size requirement for a given subaudit and round.

Parameters

- **sample_size** (*int*) – Current round size, i.e. number of ballots to be sampled in round
- **sub_audit** (*PairwiseAudit*) – Pairwise subaudit to get stopping size requirement for.

Returns *int* – The minimum number of votes cast for the reported winner in the current round size in order to stop the audit during that round. If round size is invalid, -1.

compute_priors(*self*) → numpy.ndarray
Compute prior distribution of worst case election for each pairwise subaudit.

compute_risk(*self*, *sub_audit: r2b2.audit.PairwiseAudit*, *votes_for_winner: int = None*, *current_round: int = None*, **args*, ***kwargs*) → float
Compute the risk level given current round size, votes for winner in sample, and subaudit.

The risk level is computed using the normalized product of the prior and posterior distributions. The prior comes from compute_prior() and the posterior is the hypergeometric distribution of finding votes_for_winner from a sample of size current_round taken from a total size of contest_ballots. The risk is defined as the lower half of the distribution, i.e. the portion of the distribution associated with an incorrectly reported outcome.

Parameters

- **sample** (*int*) – Votes found for reported winner in current round size.
- **current_round** (*int*) – Current round size.
- **sub_aduit** (*PairwiseAudit*) – Subaudit to generate risk value.

Returns *float* – Value for risk of given sample and round size.

next_sample_size(*self*)
Generate estimates of possible next sample sizes.

Note: To be used during live/interactive audit execution.

compute_min_winner_ballots(*self*, *sub_audit: r2b2.audit.PairwiseAudit*, *rounds: List[int]*, *progress: bool = False*, **args*, ***kwargs*)
Compute the minimum number of winner ballots for a list of round sizes.

Compute a list of minimum number of winner ballots that must be found in the corresponding round (sample) sizes to meet the stopping condition.

Parameters

- **sub_audit** (*PairwiseAudit*) – Subaudit specifying which pair of candidates to run for.

- **rounds** (*List[int]*) – List of round sizes.
- **progress** (*bool*) – If True, a progress bar will display.

Returns *List[int]* – List of minimum winner ballots to meet the stopping conditions for each round size in rounds.

`compute_all_min_winner_ballots`(*self*, *sub_audit: r2b2.audit.PairwiseAudit*, *max_sample_size: int = None*, *progress: bool = False*, **args*, ***kwargs*)

Compute the minimum winner ballots for all possible sample sizes.

Parameters

- **max_sample_size** (*int*) – Optional. Set maximum sample size to generate stopping sizes up to. If not provided the maximum sample size is determined by max_fraction_to_draw and the total contest ballots.
- **progress** (*bool*) – If True, a progress bar will display.

Returns

List[int] –

List of minimum winner ballots to meet the stopping condition for each round size in the range [min_sample_size, max_sample_size].

`get_risk_level`(*self*, **args*, ***kwargs*)

Find the risk level of the audit, that is, the smallest risk limit for which the audit as it has panned out would have already stopped.

Returns *float* – Risk level of the realization of the audit.

`__repr__`(*self*)

String representation of Audit class.

Note: Can (and perhaps should) be overwritten in subclass.

`current_dist_null`(*self*)

Update distribution_null for each sub audit for current round.

`_current_dist_null_pairwise`(*self*, *sub_audit: PairwiseAudit*, *bulk_use_round_size=False*)

Update distribution_null for a single PairwiseAudit

Parameters

- **sub_audit** (*PairwiseAudit*) – Pairwise subaudit for which to update distribution.
- **bulk_use_round_size** (*bool*) – Optional argument used by bulk methods. Since the bulk methods do not sample ballots for candidates during the rounds, this flag simply uses the round schedule as the round draw (instead of the pairwise round draw) for updating the distribution. Default is False.

`current_dist_reported`(*self*)

Update distribution_reported_tally for each subaudit for current round.

`_current_dist_reported_pairwise`(*self*, *sub_audit: PairwiseAudit*, *bulk_use_round_size=False*)

Update dist_reported for a single PairwiseAudit.

Parameters

- **sub_audit** (*PairwiseAudit*) – Pairwise subaudit for which to update distriution.

- **bulk_use_round_size** (*bool*) – Optional argument used by bulk methods. Since the bulk methods do not sample ballots for candidates during the rounds, this flag simply uses the round schedule as the round draw (instead of the pairwise round draw) for updating the distribution. Default is False.

`truncate_dist_null`(*self*)
Update risk schedule and truncate null distribution for each subaudit.

`_truncate_dist_null_pairwise`(*self*, *pair: str*)
Update risk schedule and truncate null distribution for a single subaudit.

Parameters **pair** (*str*) – Dictionary key for subaudit (within the audit's subaudits) to truncate distribution and update risk schedule.

`truncate_dist_reported`(*self*)
Update stopping prob schedule and truncate reported distribution for each subaudit.

`_truncate_dist_reported_pairwise`(*self*, *pair*)
Update stopping prob schedule and truncate reported distribution for a single subaudit.

Parameters **pair** (*str*) – Dictionary key for subaudit (within the audit's subaudits) to truncate distribution and update stopping prob schedule.

`__get_interval`(*self*, *dist: List[float]*)
Get relevant interval [l, u] of given distribution.

Find levels l and u such that cdf(l) < tolerance and 1 - cdf(u) < tolerance. The purpose of this is to improve efficiency in the current_dist_* functions for audits without replacement where almost all of the hypergeometric distribution falls in a fraction of its range, i.e. between l and u.

Note: cdf() in this context does not require cdf(infinity) = 1, although the distribution should sum very closely to 1.

`asn`(*self*, *pair: str*)
Compute ASN as described in BRAVO paper for pair of candidates.

Given the fractional margin for the reported winner and the risk limit (alpha) produce the average number of ballots sampled during the audit.

Parameters **pair** (*str*) – Dictionary key referencing pairwise audit in audit's subaudits.

Returns *int* – ASN value.

`execute_round`(*self*, *sample_size: int*, *sample: dict*, *verbose: bool = False*) → bool
Execute a single, non-interactive audit round.

Executes 1 round of the audit, given its current state. If the audit is stopped, its state will not be modified.

Warning: This method assumes the audit is in the correct state to be executed. If multiple executions of a full audit will be run the same audit object, make sure to call reset on the audit object between full executions.

Parameters

- **sample_size** (*int*) – Total ballots sampled by the end of this round (cumulative).
- **sample** (*dict*) – Sample counts for each candidate by the end of this round (cumulative).

Returns *bool* – True if the audit met its stopping condition by this round.

run(*self*, *verbose: bool = False*)
Begin interactive audit execution.

Begins the interactive version of the audit. While computations for different audits will vary, the process for executing each one is the same. This provides a process for selecting a sample size, determining if the ballots found for the reported winner in that sample size meet the stopping condition(s), and if not continuing with the audit. As the audit proceeds, data including round sizes, ballots for the winner in each round size, and per round risk and stopping probability are stored.

_reset(*self*)
Reset attributes modified during run().

stopping_condition(*self*, *verbose: bool = False*) → bool
Determine if the audits stopping condition has been met.

The audit stopping condition is met if and only if each pairwise stopping condition is met.

next_min_winner_ballots(*self*, *verbose: bool = False*)
Compute next stopping size of given (actual) sample sizes for all subaudits.

1.2.5 r2b2.cli

R2B2's command line interface offers significant out-of-the-box functionality with respect to executing audits and generating audit data without requiring the user to write a single line of Python.

Note: Why does this file exist, and why not put this in __main__?

You might be tempted to import things from __main__ later, but that will cause problems: the code will get executed twice:

- When you run *python -m r2b2* python will execute `__main__.py` as a script. That means there won't be any `r2b2.__main__` in `sys.modules`.
- When you import __main__ it will get executed again (as a module) because there's no `r2b2.__main__` in `sys.modules`.

Also see (1) from http://click.pocoo.org/5/setuptools/#setuptools-integration

---

Module Contents

class r2b2.cli.**IntList**
Bases: `click.ParamType`

Represents the type of a parameter. Validates and converts values from the command line or Python into the correct type.

To implement a custom type, subclass and implement at least the following:

- The `name` class attribute must be set.
- Calling an instance of the type with `None` must return `None`. This is already implemented by default.
- *`convert()`* must convert string values to the correct type.
- *`convert()`* must accept values that are already the correct type.
- It must be able to convert a value if the `ctx` and `param` arguments are `None`. This can occur when converting prompt input.

convert(*self*, *value*, *param*, *ctx*)
Convert the value to the correct type. This is not called if the value is `None` (the missing value).

This must accept string values from the command line, as well as values that are already the correct type. It may also convert other compatible types.

The `param` and `ctx` arguments may be `None` in certain situations, such as when converting prompt input.

If the value cannot be converted, call *fail()* with a descriptive message.

Parameters

- **value** – The value to convert.
- **param** – The parameter that is using this type to convert its value. May be `None`.
- **ctx** – The current context that arrived at this value. May be `None`.

to_info_dict(*self*) → Dict[str, Any]
Gather information that could be useful for a tool generating user-facing documentation.

Use `click.Context.to_info_dict()` to traverse the entire CLI structure.

New in version 8.0.

get_metavar(*self*, *param: click.core.Parameter*) → Optional[str]
Returns the metavar default for this param if it provides one.

get_missing_message(*self*, *param: click.core.Parameter*) → Optional[str]
Optionally might return extra information about a missing parameter.

New in version 2.0.

split_envvar_value(*self*, *rv: str*) → Sequence[str]
Given a value from an environment variable this splits it up into small chunks depending on the defined envvar list splitter.

If the splitter is set to *None*, which means that whitespace splits, then leading and trailing whitespace is ignored. Otherwise, leading and trailing splitters usually lead to empty items being included.

fail(*self*, *message: str*, *param: Optional[click.core.Parameter] = None*, *ctx: Optional[click.core.Context] = None*) → NoReturn
Helper method to fail with an invalid value message.

shell_complete(*self*, *ctx: click.core.Context*, *param: click.core.Parameter*, *incomplete: str*) → List[click.shell_completion.CompletionItem]
Return a list of `CompletionItem` objects for the incomplete value. Most types do not provide completions, but some do, and this allows custom types to provide custom completions as well.

Parameters

- **ctx** – Invocation context for this command.
- **param** – The parameter that is requesting completion.
- **incomplete** – Value being completed. May be empty.

New in version 8.0.

r2b2.cli.**interactive**(*election_mode*, *election_file*, *contest_file*, *audit_type*, *risk_limit*, *max_fraction_to_draw*, *verbose*)
Executes an audit round by round.

Depending on what options are passed to the interactive command, users may be prompted for contest results, audit type, risk limit, and/or maximum fraction of contest ballots to draw when initializing the contest and audit to run.

During execution, users will enter each round size and results of the round's sample and subsequently receive information about the current state of the audit. The process continues until either the stopping conditions are met or the audit reaches the maximum sample size.

For information on each option run

```
$ r2b2 interactive --help
```

Example

Contest results can be passed as a JSON file rather than entering the data through the prompt:

```
$ r2b2 interactive --contest-file example_contest.json
```

Tip: To generate a template contest JSON file run:

```
$ r2b2 template contest
```

Example

Audit parameters can be passed in as options rather than entering through the prompt:

```
$ r2b2 interactive --audit-type brla --risk-limit 0.1 --max-fraction-to-draw 0.2
$ r2b2 interactive -a brla -r 0.1 -m 0.2    // Shortened equivalent
```

Example

Election mode allows users to enter all the results from an election then select a contest from the election to audit:

```
$ r2b2 interactive -e
$ r2b2 interactive -e --election-file // pass election results as JSON file.
```

Warning: Election mode simply allows you to enter an entire election's data, then select one one contest from that election to run. Auditing multiple contests from an election concurrently is not implemented.

r2b2.cli.**bulk**(*audit_type*, *risk_limit*, *max_fraction_to_draw*, *contest_file*, *output*, *round_list*, *full_audit_limit*, *pair*, *verbose*)

Bulk auditing mode generates stopping sizes for a given fixed round schedule.

Either provide a list of round sizes for which to generate stopping sizes or generate a ballot by ballot list of stopping sizes from the minimum valid sample size to the default maximum sample size or a specified maximum sample size.

Parameters

- **contest_file** – Contest results as JSON file.
- **audit_type** – Which audit type to use to generate stopping sizes.
- **risk_limit** – Risk limit (alpha) of audit.

- **max_fraction_to_draw** – Maximum fraction of contest ballots that could be drawm during the audit. Sets the default maximum size of the ballot by ballot output.

Tip: To generate a template contest JSON file, run:

```
$ r2b2 template contest
```

Returns Formatted list of rounds and their associated stopping sizes. Default execution is ballot by ballot from minimum valid sample size to the maximum sample size of audit.

Example

To generate stopping sizes for a specific set of round sizes, provide the round sizes as a space separated list of integers enclosed by quotes using the round list option:

```
$ r2b2 bulk -l '100 200 300' contest.json brla 0.1 0.5
```

Example

To generate a ballot by ballot result from the minimum valid sample size to a specific maximum (i.e. not the maximum fraction to draw of the audit), run:

```
$ r2b2 bulk -f 221 contest.json brla 0.1 0.5
```

Example

To write the results to a file instead of to stdout, run:

```
$ r2b2 bulk -o output.txt contest.json brla 0.1 0.5
```

Tip: Generating large or compute heavy data sets can take some time. To estimate run times, use the verbose flag to display a progress bar:

```
$ r2b2 bulk -v contest.json brla 0.1 0.5
```

---

r2b2.cli.**template**(*style*, *output*)

Generate JSON templates for possible input formats.

Example

To create a contest results JSON file, first generate the template as a new JSON file:

```
$ r2b2 template -o my_contest.json contest
```

Now the file my_contest.json will be created and contain:

```
{
    "contest_ballots" : 100,
    "tally" : {
            "CandidateA" : 50,
            "CandidateB" : 50
    },
    "num_winners" : 1,
    "reported_winners" : ["CandidateA"],
    "contest_type" : "PLURALITY"
}
```

Simply repopulate the fields with your contest results.

1.2.6 r2b2.contest

Contest module for handling individual contest data.

Module Contents

class r2b2.contest.**ContestType**

Bases: enum.Enum

Enum indicating what type of vote variation was used in the contest.

__repr__(*self*)
Return repr(self).

__str__(*self*)
Return str(self).

__dir__(*self*)
Default dir() implementation.

__format__(*self*, *format_spec*)
Default object formatter.

__hash__(*self*)
Return hash(self).

__reduce_ex__(*self*, *proto*)
Helper for pickle.

name(*self*)
The name of the Enum member.

value(*self*)
The value of the Enum member.

classmethod _convert(*cls*, *name*, *module*, *filter*, *source=None*)
Create a new Enum subclass that replaces a collection of global constants

class r2b2.contest.**PairwiseContest**(*reported_winner: str*, *reported_loser: str*, *reported_winner_ballots: int*, *reported_loser_ballots: int*)

Simple 2-candidate, no irrelevant ballot sub contests of a Contest.

class r2b2.contest.**Contest**(*contest_ballots: int*, *tally: Dict[str, int]*, *num_winners: int*, *reported_winners: List[str]*, *contest_type:* ContestType)

Contest information from a single contest within an Election.

Variables

- **contest_ballots** (*int*) – Total number of ballots cast in the contest.
- **irrelevant_ballots** (*int*) – Number of ballots not attributed to a candidate in the tally.
- **candidates** (*List[str]*) – List of candidates in the contest sorted (descending) by tally.
- **num_candidates** (*int*) – Number of candidates in the contest.
- **num_winners** (*int*) – Number of winners desired from contest.
- **reported_winners** (*List[str]*) – Reported winners from contest. Must be candidates from list of candidates, and length should match number of winners. Stored in same order as sorted candidates.
- **contest_type** (ContestType) – What type of contest is this?
- **tally** (*Dict[str, int]*) – Reported tally from contest as a dictionary of candidates to reported votes received.
- **winner_prop** (*float*) – Proportion of ballots cast for reported winner. Currently for first winner listed in reported winners.
- **sub_contests** (*Dict[str, Dict[str, List[int]]]*) – Collection of pairwise sub-contests for each (reported winner, candidate) pair where the reported winner has more than 50% of the total sub-contest ballots, i.e. where the reported winner has a greater reported tally than the other candidate. These pairs provide the two-candidate, no irrelevant ballots assumption required by some audits.

__repr__(*self*)

String representation of Contest class.

__str__(*self*)

Human readable string representation of audit class.

to_json(*self*)

Generate dict representation of Contest for use in a JSON file.

1.2.7 r2b2.election

Election module handles data associated with an Election or collection of Contests.

Module Contents

class r2b2.election.**Election**(*name: str*, *total_ballots: int*, *contests: Dict[str,* r2b2.contest.Contest*]*)
Election information extracted from reported results.

A class to encompass all data from an entire election. The election's key data structue is a list of Contest objects which hold the relevant data from each contest within the election.

Variables

- **name** (*str*) – Election name.
- **total_ballots** (*int*) – Total ballots cast in entire election.
- **contests** (*Dict[str, Contest]*) – dict of contests within the election with names as keys.

__repr__(*self*)
Return repr(self).

__str__(*self*)
Return str(self).

1.2.8 r2b2.minerva

Minerva audit module.

Module Contents

class r2b2.minerva.**Minerva**(*alpha: float*, *max_fraction_to_draw: float*, *contest:* r2b2.contest.Contest)
Bases: *r2b2.audit.Audit*

Minerva audit implementation.

A Minerva audit is a type of risk-limiting audit that accounts for round-by-round auditor decisions. For a given sample size (in the context of a round schedule), the audit software calculates a minimum number of votes for the reported winner that must be found in the sample to stop the audit and confirm the reported outcome.

Variables

- **alpha** (*float*) – Risk limit. Alpha represents the chance that, given an incorrectly called election, the audit will fail to force a full recount.
- **max_fraction_to_draw** (*float*) – The maximum number of ballots the auditors are willing to draw as a fraction of the ballots in the contest.
- **contest** (Contest) – Contest to be audited.

Initialize a Minerva audit.

get_min_sample_size(*self*, *sub_audit:* r2b2.audit.PairwiseAudit, *min_sprob: float = 10 ** - 6*)
Computes the minimum sample size that has a stopping size (kmin). Here we find a practical minimum instead of the theoretical minimum (BRAVO's minimum) to avoid floating-point imprecisions in the later convolution process.

Parameters

- **sub_audit** (*PairwiseAudit*) – Get minimum sample size for this subaudit.
- **min_sprob** (*float*) – Round sizes with below min_sprob stopping probability are excluded.

Returns *int* – The minimum sample size of the audit, adherent to the min_sprob.

satisfactory_sample_size(*self*, *left*, *right*, *sprob*, *num_dist*, *denom_dist*)
Helper method that returns True if the round size satisfies the stopping probability.

kmin_search_upper_bound(*self*, *n*, *sub_audit:* r2b2.audit.PairwiseAudit)
The Minerva kmin is no greater than the BRAVO kmin, so the latter serves as an upper bound for a kmin binary search.

(Solve for k: (p/.5)^k * ((1-p)/.5)^(n-k) > 1/alpha)

sample_size_kmin(*self*, *left*, *right*, *num_dist*, *denom_dist*, *sum_num_right*, *sum_denom_right*, *orig_right*)
Finds a kmin with a binary search given the twin distributions.

find_sprob(*self*, *n*, *sub_audit:* r2b2.audit.PairwiseAudit)
Helper method to find the stopping probability of a given prospective round size.

binary_search_estimate(*self*, *left*, *right*, *sprob*, *sub_audit:* r2b2.audit.PairwiseAudit)
Method to use binary search approximation to find a round size estimate.

next_sample_size_gaussian(*self*, *sprob=0.9*)
This is a rougher but quicker round size estimate for very narrow margins.

next_sample_size(*self*, *sprob=0.9*, *verbose=False*, **args*, ***kwargs*)
Attempt to find a next sample size estimate no greater than 10000. Failing that, try to find an estimate no greater than 20000, and so on.

Parameters

- **sprob** (*float*) – Compute next sample for this stopping probability.
- **verbose** (*bool*) – If true, the kmin and stopping probability of the next sample size will be returned in addition to the next sample size itself.

Returns

Return maxmimum next sample size estimate across all pairwise subaudits. If verbose, return information as specified above.

_next_sample_size_pairwise(*self*, *sub_audit:* r2b2.audit.PairwiseAudit, *sprob=0.9*)
Compute next sample size for a single pairwise subaudit.

Parameters

- **sub_audit** (*PairwiseAudit*) – Compute the sample size for this sub_audit.
- **sprob** (*float*) – Get the sample size for this stopping probability.

Returns Estimate in the format [sample size, kmin, stopping probability].

stopping_condition_pairwise(*self*, *pair: str*, *verbose: bool = False*) → bool
Check, without finding the kmin, whether the subaudit is complete.

Parameters **pair** (*str*) – Dictionary key referencing pairwise subaudit to evaluate.

Returns *bool* – Whether or not the pairwise stopping condition has been met.

next_min_winner_ballots_pairwise(*self*, *sub_audit:* r2b2.audit.PairwiseAudit) → int
Compute stopping size for a given subaudit.

Parameters **sub_audit** (*PairwiseAudit*) – Compute next stopping size for this subaudit.

Returns *int* – Stopping size for most recent round.

compute_min_winner_ballots(*self*, *sub_audit:* r2b2.audit.PairwiseAudit, *rounds: List[int]*, **args*, ***kwargs*)
Compute the minimum number of winner ballots for a round schedule of a pairwise audit.

Extend the audit's round schedule with the passed (partial) round schedule, and then extend the audit's minimum number of winner ballots schedule with the corresponding minimums to meet the stopping condition.

Parameters

- **sub_audit** (*PairwiseAudit*) – Compute minimum winner ballots for this Pairwise subaudit.
- **rounds** (*List[int]*) – A (partial) round schedule of the audit.

`find_kmin`(*self*, *sub_audit:* r2b2.audit.PairwiseAudit, *sample_size: int*, *append: bool*)
Search for a kmin (minimum number of winner ballots) satisfying all stopping criteria.

Parameters

- **sub_audit** (*PairwiseAudit*) – Find kmin for this subaudit.
- **sample_size** (*int*) – Sample size to find kmin for.
- **append** (*bool*) – Optionally append the kmins to the min_winner_ballots list. This may not always be desirable here because, for example, appending happens automatically outside this method during an interactive audit.

`compute_all_min_winner_ballots`(*self*, *sub_audit:* r2b2.audit.PairwiseAudit, *max_sample_size: int = None*, **args*, ***kwargs*)
Compute the minimum number of winner ballots for the complete (that is, ballot-by-ballot) round schedule.

Note: Due to limited convolutional precision, results may be off somewhat after the stopping probability very nearly equals 1.

Parameters

- **sub_audit** (*PairwiseAudit*) – Compute minimum winner ballots for this pairwise subaudit.
- **max_sample_size** (*int*) – Optionally set the maximum sample size to generate stopping sizes (kmins) up to. If not provided the maximum sample size is determined by max_frac_to_draw and the total contest ballots.

Returns None, kmins are appended to the min_winner_ballots list.

`compute_risk`(*self*, *votes_for_winner: int*, *pair: str*, **args*, ***kwargs*)
Return the hypothetical pvalue if votes_for_winner were obtained in the most recent round.

`get_risk_level`(*self*)
Return the risk level of an interactive Minerva audit.

Non-interactive and bulk Minerva audits are not considered here since the sampled number of reported winner ballots is not available.

`__repr__`(*self*)
String representation of Audit class.

Note: Can (and perhaps should) be overwritten in subclass.

`__str__`(*self*)
Human readable string (i.e. printable) representation of Audit class.

Note: Can (and perhaps should) be overwritten in subclass.

---

current_dist_null(*self*)
Update distribution_null for each sub audit for current round.

_current_dist_null_pairwise(*self*, *sub_audit: PairwiseAudit*, *bulk_use_round_size=False*)
Update distribution_null for a single PairwiseAudit

Parameters

- **sub_audit** (*PairwiseAudit*) – Pairwise subaudit for which to update distribution.
- **bulk_use_round_size** (*bool*) – Optional argument used by bulk methods. Since the bulk methods do not sample ballots for candidates during the rounds, this flag simply uses the round schedule as the round draw (instead of the pairwise round draw) for updating the distribution. Default is False.

current_dist_reported(*self*)
Update distribution_reported_tally for each subaudit for current round.

_current_dist_reported_pairwise(*self*, *sub_audit: PairwiseAudit*, *bulk_use_round_size=False*)
Update dist_reported for a single PairwiseAudit.

Parameters

- **sub_audit** (*PairwiseAudit*) – Pairwise subaudit for which to update distriution.
- **bulk_use_round_size** (*bool*) – Optional argument used by bulk methods. Since the bulk methods do not sample ballots for candidates during the rounds, this flag simply uses the round schedule as the round draw (instead of the pairwise round draw) for updating the distribution. Default is False.

truncate_dist_null(*self*)
Update risk schedule and truncate null distribution for each subaudit.

_truncate_dist_null_pairwise(*self*, *pair: str*)
Update risk schedule and truncate null distribution for a single subaudit.

Parameters **pair** (*str*) – Dictionary key for subaudit (within the audit's subaudits) to truncate distribution and update risk schedule.

truncate_dist_reported(*self*)
Update stopping prob schedule and truncate reported distribution for each subaudit.

_truncate_dist_reported_pairwise(*self*, *pair*)
Update stopping prob schedule and truncate reported distribution for a single subaudit.

Parameters **pair** (*str*) – Dictionary key for subaudit (within the audit's subaudits) to truncate distribution and update stopping prob schedule.

__get_interval(*self*, *dist: List[float]*)
Get relevant interval [l, u] of given distribution.

Find levels l and u such that cdf(l) < tolerance and 1 - cdf(u) < tolerance. The purpose of this is to improve efficiency in the current_dist_* functions for audits without replacement where almost all of the hypergeometric distribution falls in a fraction of its range, i.e. between l and u.

Note: cdf() in this context does not require cdf(infinity) = 1, although the distribution should sum very closely to 1.

---

asn(*self*, *pair: str*)
Compute ASN as described in BRAVO paper for pair of candidates.

Given the fractional margin for the reported winner and the risk limit (alpha) produce the average number of ballots sampled during the audit.

Parameters **pair** (*str*) – Dictionary key referencing pairwise audit in audit's subaudits.

Returns *int* – ASN value.

execute_round(*self*, *sample_size: int*, *sample: dict*, *verbose: bool = False*) → bool
Execute a single, non-interactive audit round.

Executes 1 round of the audit, given its current state. If the audit is stopped, its state will not be modified.

Warning: This method assumes the audit is in the correct state to be executed. If multiple executions of a full audit will be run the same audit object, make sure to call reset on the audit object between full executions.

Parameters

- **sample_size** (*int*) – Total ballots sampled by the end of this round (cumulative).
- **sample** (*dict*) – Sample counts for each candidate by the end of this round (cumulative).

Returns *bool* – True if the audit met its stopping condition by this round.

run(*self*, *verbose: bool = False*)
Begin interactive audit execution.

Begins the interactive version of the audit. While computations for different audits will vary, the process for executing each one is the same. This provides a process for selecting a sample size, determining if the ballots found for the reported winner in that sample size meet the stopping condition(s), and if not continuing with the audit. As the audit proceeds, data including round sizes, ballots for the winner in each round size, and per round risk and stopping probability are stored.

_reset(*self*)
Reset attributes modified during run().

stopping_condition(*self*, *verbose: bool = False*) → bool
Determine if the audits stopping condition has been met.

The audit stopping condition is met if and only if each pairwise stopping condition is met.

next_min_winner_ballots(*self*, *verbose: bool = False*)
Compute next stopping size of given (actual) sample sizes for all subaudits.

1.2.9 r2b2.simulator

R2B2 Simulation Module.

Module Contents

class r2b2.simulator.**DBInterface**(*host='localhost'*, *port=27017*, *name='r2b2'*, *user='reader'*, *pwd='icanread'*)
Class for handling MongoDB operations.

audit_lookup(*self*, *audit_type: str*, *alpha: float*, *qapp: dict = None*, **args*, ***kwargs*)
Find/Create an audit in database.

Searches through database for an existing audit entry with the given parameters. If none exists, an audit entry is created for the parameters.

Parameters

- **audit_type** (*str*) – Name of audit, for example: 'minerva', 'brla', etc.
- **alpha** (*float*) – Risk-limit of audit.
- **qapp** (*dict*) – Optional parameter that appends dict to mongo query.

Returns ObjectID of new or existing audit entry.

`contest_lookup`(*self*, *contest:* r2b2.contest.Contest, *qapp: dict = None*, **args*, ***kwargs*)
Find/Create a contest in database.

Searches through database for an existing contest entry with the given parameters. If none exists, a contest entry is created.

Parameters

- **contest** (*r2b2.contest.Contest*) – Contest with attributes to be used in the database query.
- **qapp** (*dict*) – Optional parameter that appends dict to mongo query.

Returns ObjectID of new of existing contest entry.

`simulation_lookup`(*self*, *audit*, *reported*, *underlying*, *invalid*, *qapp: dict = None*, **args*, ***kwargs*)
Find/Create a simulation in database.

Searches through database for an existing simulation entry with the given parameters. If none exists, a simulation entry is created.

Parameters

- **audit** – ObjectID of audit entry (from audits collection) used in the simulation.
- **reported** – ObjectID of reported contest entry (from contests collection) used in the simulation.
- **underlying** – Description of the underlying contest used in the simulation. Could be an ObjectID from the contests table, could simply be a string indicating a tie, depends on the specific simulation.
- **qapp** (*dict*) – Optional parameter that appends dict to mongo query.

Returns ObjectID of new or existing simulation entry.

`trial_lookup`(*self*, *sim_id*, **args*, ***kwargs*)
Find all trials for a given simulation ObjectID

`write_trial`(*self*, *entry*)
Write a trial document into the trials collection.

`update_analysis`(*self*, *sim_id*, *entry*)
Update analysis in simulation document.

`class r2b2.simulator.Simulation`(*audit_type: str*, *alpha: float*, *reported:* r2b2.contest.Contest, *underlying*, *invalid: bool*, *db_mode=True*, *db_host='localhost'*, *db_port=27017*, *db_name='r2b2'*, *user='reader'*, *pwd='icanread'*, **args*, ***kwargs*)

Bases: `abc.ABC`

Abstract Base Class to define a simulation.

Variables

- **db_mode** (*bool*) – Indicates if simulation is running in Database mode or local mode.
- **audit_type** (*str*) – Indicates what type of audit is simulated.

- **alpha** (*float*) – Risk-limit of simulation.
- **audit_id** (*str*) – ObjectID of audit entry from audits collection in MongoDB.
- **reported** (Contest) – Reported contest results that are audited during simulation.
- **reported_id** (*str*) – ObjectID of reported contest entry from contests collection in MongoDB.
- **underlying** (*str*) – Indicates the true underlying contest results ballots are drawn from during the simulation. This might be an ObjectID similar to reported_id, it might be a string simply indicating that the underlying distribution is a tie. This field is specified by a specific simulation implementation.
- **sim_id** (*str*) – ObjectID of simulation from simulations collection in MongoDB defined by the reported contest, underlying contest, and audit.
- **trials** – List of trials performed in run() method. Trials are dicts formatted for JSON output or MongoDB document entry.

run(*self*, *n: int*)
Execute n trials of the simulation.

Executes n simulation trials by generating a random seed, running a trial with the given seed, and writing the trial entry to the trials collection.

Parameters **n** (*int*) – Number of trials to execute and write to database.

get_seed(*self*)
Generate a random seed.

Note: This method generates 8 random bytes using os sources of randomness. If a different source of randomness is desired, overwrite the method per implementation.

---

output(*self*, *fd: str = None*)
Write output of simulation to JSON file.

Parameters **fd** (*str*) – filename to write output to. If no file is passed, formatted JSON is simply printed.

output_audit(*self*)
Create audit output in JSON format.

Note: This functionality is separated into a method so specific audit implementations may override it and customize their output in non-database mode.

---

abstract trial(*self*, *seed*)
Execute a single trial given a random seed.

abstract analyze(*self*, **args*, ***kwargs*)
Analyze the simulation trials.

r2b2.simulator.**histogram**(*values: List*, *xlabel: str*, *bins='auto'*)
Create a histogram for a given dataset.

CHAPTER

TWO

INSTALLATION

At the command line:

```
pip install r2b2
```

You can also install the in-development version with:

```
pip install https://github.com/gwexploratoryaudits/r2b2/archive/master.zip
```

CHAPTER

THREE

USAGE

To use R2B2 in a project:

```
import r2b2
```

CHAPTER

FOUR

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.2 Documentation improvements

R2B2 could always use more documentation, whether as part of the official R2B2 docs, in docstrings, or even on the web in blog posts, articles, and such.

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/gwexploratoryaudits/r2b2/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

4.4 Development

For more specific development guidelines and standards see the Design Guide. For guidelines specific to developing a new audit see the Audit Design Guide.

To set up *r2b2* for local development:

1. Fork r2b2 (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:gwexploratoryaudits/r2b2.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git commit
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

4.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

4.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER

FIVE

AUTHORS

- R2B2 Team

CHAPTER

SIX

CHANGELOG

6.1 0.1.0 (2020-02-11)

- skeleton via cookiecutter-pylibrary commit to master Removed pypy builds, may want to add back for performance Can add mac/osx builds later if desired

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

N

O

P

R

S

T

U

V

W