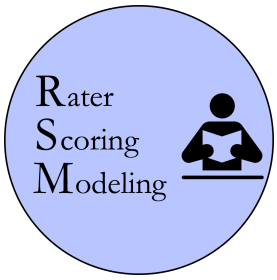
Rater Scoring Modeling Tool

Release 9.0.0

Mar 10, 2022

Contents



Automated scoring of written and spoken responses is a growing field in educational natural language processing. Automated scoring engines employ machine learning models to predict scores for such responses based on features extracted from the text/audio of these responses. Examples of automated scoring engines include MI Write for written responses and SpeechRater for spoken responses.

RSMTool is a python package which automates and combines in a *single pipeline* multiple analyses that are commonly conducted when building and evaluating automated scoring models. The output of RSMTool is a comprehensive, customizable HTML statistical report that contains the outputs of these multiple analyses. While RSMTool does make it really simple to run this set of standard analyses using a single command, it is also fully customizable and allows users to easily exclude unneeded analyses, modify the standard analyses, and even include custom analyses in the report.

We expect the *primary users* of RSMTool to be researchers working on developing new automated scoring engines or on improving existing ones. Note that RSMTool is not a scoring engine by itself but rather a tool for building and evaluating machine learning models that may be used in such engines.

The primary means of using RSMTool is via the *command-line*.

CHAPTER 1

Documentation

Note: If you use the Dash app on macOS, you can also download the complete RSMTool documentation for offline use. Go to the Dash preferences, click on "Downloads", then "User Contributed", and search for "RSMTool".

1.1 Who is RSMTool for?

We expect the primary users of RSMTool to be researchers working on developing new automated scoring engines or on improving existing ones. Here's the most common scenario.

A group of researchers already *has* a set of responses such as essays or recorded spoken responses which have already been assigned numeric scores by human graders. They have also processed these responses and extracted a set of (numeric) features using systems such as Coh-Metrix, TextEvaluator, OpenSmile, or using their own custom text/speech processing pipeline. They wish to understand how well the set of chosen features can predict the human score.

They can then run an RSMTool "experiment" to build a regression-based scoring model (using one of many available regressors) and produce a report. The report includes descriptive statistics for all their features, diagnostic information about the trained regression model, and a comprehensive evaluation of model performance on a held-out set of responses.

While they could use `R`, `PASW (SPSS)` or other tools to perform each of the RSMTool analyses individually and compile a report himself, RSMTool does all of this work for them with just a single command. Furthermore, the analyses included into the tool highlight educational measurement criteria important to building automated scoring models. If they wish, they can conduct further exploratory analysis using their preferred tools for data analysis by using the output of RSMTool as a starting point.

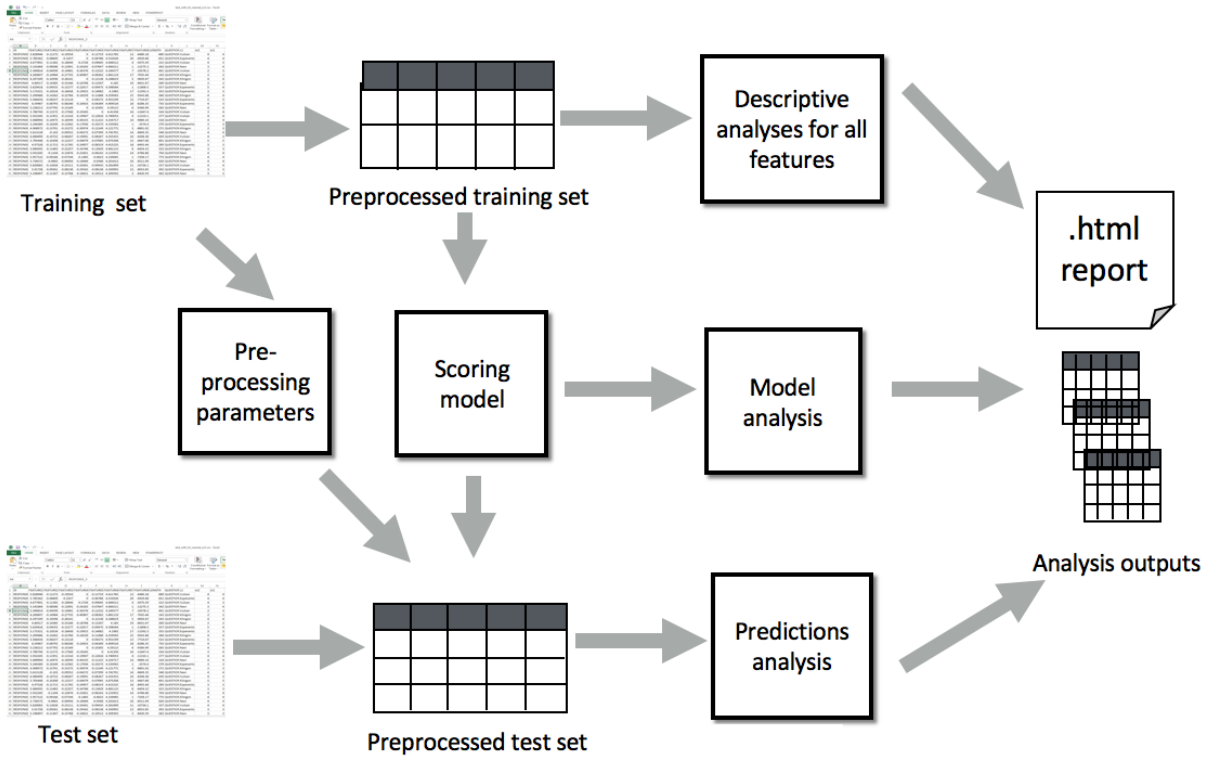
RSMTool is designed to be customizable:

1. The users can choose to either run all of the default analyses or select *only* the subset applicable to their particular study by changing the appropriate settings in a configuration file.
2. RSMTool provides explicit support for adding *custom analyses* to the report if the user has some analysis in mind that is not already provided by RSMTool. These analyses can then be automatically included in all subsequent experiments.

While training and evaluating a scoring model represents the most common use case for RSMTool, it can do a lot more for advanced users such as *evaluating predictions obtained using an external scoring engine*, *generating predictions for new data*, *generating a detailed comparison between two different scoring models*, *generating a summary report for multiple scoring models*, and *using cross-validation to better estimate the performance of scoring models*.

1.2 Overview of RSMTool Pipeline

The following figure gives an overview of the RSMTool pipeline:



As its primary input, RSMTool takes a *data file* containing a table with numeric, non-sparse features and a human scores for all responses, *pre-processes* them and lets you train a regression-based **Scoring Model** to predict the human score from the features. Available regression models include Ridge, SVR, AdaBoost, and Random Forests, among many others.

This trained model can then be used to generate scores for a held-out evaluation data whose feature values are pre-processed using the same *Pre-processing Parameters*. In addition to the raw scores predicted by the model, the **Prediction Analysis** component of the pipline generates several additional *post-processed scores* that are commonly used in automated scoring.

The primary output of RSMTool is a comprehensive, customizable HTML statistical report that contains the multiple analyses required for a comprehensive evaluation of an automated scoring model including descriptive analyses for all features, model analyses, subgroup comparisons, as well as several different *evaluation measures* illustrating model efficacy. More Details about these analyses can be found in this documentaion and in a separate technical paper.

In addition to the HTML report, RSMTool also saves the intermediate outputs of all of the performed analyses as *CSV files*.

1.2.1 Input file format

The input files for the training and evaluation data should either be in a tabular format with responses as rows and features and score(s) in the columns in the `jsonlines` format with a JSON object per responses on each line. See below for a more detailed description of the `.jsonlines` format.

RSMTool supports input files in `.csv`, `.tsv`, `.sas7bdat`, `.xlsx`, or `.jsonlines` format. The format of the file is determined based on the extension. In all cases the *output files* will be saved in `.csv` format by default (see *file format* for other intermediate file output options).

For `Excel` spreadsheets, all data must be stored in the first sheet.

Note: RSMTool 8.1 and higher no longer support `.xls` files. All Excel files must be in `.xlsx` format.

In a `.jsonlines` format file, each line corresponds to a response and is represented as a dictionary with column names as the keys and column values as the values. For example:

```
{"id":"RESPONSE_1","FEATURE1":4.9380646013,"FEATURE2":-0.0584019308,"FEATURE3":-0.
↪0846667513,"FEATURE4":-0.3167939755,"FEATURE5":-0.059868434,"FEATURE6":4.6559139785,
↪"FEATURE7":16,"FEATURE8":-7940.9812058284,"LENGTH":279,"QUESTION":"QUESTION_1","L1":
↪"Esperanto","y":3,"score2":3}
{"id":"RESPONSE_2","FEATURE1":5.3798973535,"FEATURE2":-0.0781006576,"FEATURE3":-0.
↪1592030845,"FEATURE4":-0.2715376933,"FEATURE5":-0.1073346977,"FEATURE6":4.
↪7626728111,"FEATURE7":10,"FEATURE8":-6683.4324801506,"LENGTH":434,"QUESTION":
↪"QUESTION_1","L1":"Klingon","y":4,"score2":4}
```

Although RSMTool does allow for nesting in the JSON objects on each line of a `.jsonlines` format file, the top-level keys will be ignored when parsing the files and processing the data. Therefore, in the example below, the keys `x` and `metadata` will be ignored.

```
{"id":"RESPONSE_1","y":3,"x":{"FEATURE1":4.9380646013,"FEATURE2":-0.0584019308,
↪"FEATURE3":-0.0846667513,"FEATURE4":-0.3167939755,"FEATURE5":-0.059868434,"FEATURE6
↪":4.6559139785,"FEATURE7":16.0,"FEATURE8":-7940.9812058284},"metadata":{"LENGTH
↪":279,"QUESTION":"QUESTION_1","L1":"Esperanto","score2":3}}
{"id":"RESPONSE_2","y":4,"x":{"FEATURE1":5.3798973535,"FEATURE2":-0.0781006576,
↪"FEATURE3":-0.1592030845,"FEATURE4":-0.2715376933,"FEATURE5":-0.1073346977,"FEATURE6
↪":4.7626728111,"FEATURE7":10.0,"FEATURE8":-6683.4324801506},"metadata":{"LENGTH
↪":434,"QUESTION":"QUESTION_1","L1":"Klingon","score2":4}}
```

If the file contains nesting of more than two levels, the column names for nested records beyond the top level will be generated using `.` to separate levels: For example, given the JSON object `{'foo': {'bar': {'foo2': 0, 'foo3': 0}}}`, `foo` will be ignored and the columns will be named `bar.foo2` and `bar.foo3`.

1.2.2 Feature pre-processing

Data filtering

1. Remove all training and evaluation responses that have non-numeric for any of the features (see *column selection methods* for different ways to select features).
2. Remove all training and evaluation responses with non-numeric values for human scores.

3. Optionally remove all training and evaluation responses with zero values for human scores. Zero scored responses are usually removed since in many scoring rubrics, zero scores usually indicate non-scorable responses.
4. Remove all features with values that do not change across responses (i.e., those with a standard deviation close to 0).

Data preprocessing

1. Truncate/clamp any outlier feature values, where outliers are defined as $\mu \pm 4 * \sigma$, where μ is the mean and σ is the standard deviation.
2. Apply pre-specified *transformations* to feature values.
3. Standardize all transformed feature values into z-scores.
4. Flip the signs for feature values if necessary.

Pre-processing parameters

Any held-out evaluation data on which the model is to be evaluated needs to be pre-processed in the same way as the training data. Therefore, the following parameters are computed on the training set, saved to disk, and re-used when pre-processing the evaluation set:

- Mean and standard deviation of raw feature values. These are used to compute floor and ceiling for truncating any outliers in the evaluation set;
- Any transformation and sign changes that were applied;
- Mean and standard deviation of transformed feature values. These are used to convert feature values in the evaluation set to z-scores.

1.2.3 Score post-processing

RSMTool computes six different versions of scores commonly used in different applications of automated scoring:

raw

The raw predictions generated by the model.

raw_trim

The raw predictions "trimmed" to be in the score range acceptable for the item. The scores are trimmed to be within the following range: $score_{min} - tolerance$ and $score_{max} + tolerance$. Unless specified otherwise, $score_{min}$ and $score_{max}$ are set to the lowest and highest points on the scoring scale respectively and `tolerance` is set to 0.4998.

This approach represents a compromise: it provides scores that are real-valued and, therefore, provide more information than human scores that are likely to be integer-valued. However, it also ensures that the scores fall within the expected scale.

Note: In RSMTool v6.x and earlier, the default value for `tolerance` was 0.49998 (note the extra "9"). Therefore, the `raw_trim` values computed for outliers by RSMTool v7.0 and onwards will be *different* from those computed by previous versions. If you wish to replicate results obtained with older versions, set the new *trim_tolerance* field in the experiment configuration file to 0.49998.

---

raw_trim_round

The raw_trim predictions rounded to the nearest integer.

Note: The rounding is done using `rint` function from `numpy`. See numpy documentation for treatment of values such as 1.5.

---

scale

The raw predictions rescaled to match the human score distribution on the training set. The raw scores are first converted to z-scores using the mean and standard deviation of the *machine scores* predicted for the training set. The z-scores are then converted back to "scaled" scores using the mean and standard deviation of the *human scores*, also computed on the training set.

scale_trim

The scaled scores trimmed in the same way as raw_trim scores.

scale_trim_round

The scale_trim scores scores rounded to the nearest integer.

1.3 Evaluation Metrics

This section documents the exact mathematical definitions of the primary metrics used in RSMTool for evaluating the performance of automated scoring engines. RSMTool reports also include many secondary evaluations as described in *intermediary files* and the *report sections*.

The following conventions are used in the formulas in this section:

N – total number of responses in the *evaluation set* with numeric human scores and numeric system scores. Zero human scores are, by default, excluded from evaluations unless *exclude_zero_scores* was set to `false`.

M – system score. The primary evaluation metrics in the RSMTool report are computed for *all* six types of *scores*. For some secondary evaluations, the user can choose between raw and scaled scores using the *use_scaled_predictions* configuration field for RSMTool or the *scale_with* field for RSMEval.

H – human score. The score values in *test_label_column* for RSMTool or *human_score_column* for RSMEval.

$H2$ – second human score (if available). The score values in *second_human_score_column*.

N_2 – total number of responses in the evaluation set where both H and $H2$ are available and are numeric and non-zero (unless *exclude_zero_scores* was set to `false`).

$\bar{M}$ – Mean of M ; $\bar{M} = \sum_{n=1}^{N} \frac{M_i}{N}$

$\bar{H}$ – Mean of H ; $\bar{H} = \sum_{n=1}^{N} \frac{H_i}{N}$

σ_M – Standard deviation of M ; $\sigma_M = \sqrt{\frac{\sum_{i=1}^{N} (M_i - \bar{M})^2}{N-1}}$

σ_H – Standard deviation of H ; $\sigma_H = \sqrt{\frac{\sum_{i=1}^{N}(H_i - \bar{H})^2}{N-1}}$

σ_{H2} – Standard deviation of $H2$; $\sigma_{H2} = \sqrt{\frac{\sum_{i=1}^{N_2}(H2_i - \bar{H2})^2}{N_2-1}}$

1.3.1 Accuracy Metrics (Observed score)

These metrics show how well system scores M predict observed human scores H. The computed metrics are available in the *intermediate file* `eval`, with a subset of the metrics also available in the intermediate file `eval_short`.

Percent exact agreement (rounded scores only)

Percentage responses where human and system scores match exactly.

$$A = \sum_{i=1}^{N} \frac{w_i}{N} \times 100$$

where $w_i = 1$ if $M_i = H_i$ and $w_i = 0$ if $M_i \neq H_i$

The percent exact agreement is computed using *rsmtool.utils.agreement* with `tolerance` set to `0`.

Percent exact + adjacent agreement

Percentage responses where the absolute difference between human and system scores is `1` or less.

$$A_{adj} = \sum_{i=1}^{N} \frac{w_i}{N} \times 100$$

where $w_i = 1$ if $|M_i - H_i| \leq 1$ and $w_i = 0$ if $|M_i - H_i|1$.

The percent exact + adjacent agreement is computed using *rsmtool.utils.agreement* with `tolerance` set to `1`.

Cohen's kappa (rounded scores only)

$$\kappa = 1 - \frac{\sum_{k=0}^{K-1}\sum_{j=1}^{K} w_{jk}X_{jk}}{\sum_{k=0}^{K-1}\sum_{j=1}^{K} w_{jk}m_{jk}}$$

when $k = j$ then w_{jk} = 0 and when $k \neq j$ then w_{jk} = 1

where:

- K is the number of scale score categories (maximum observed rating - minimum observed rating + 1). Note that for κ computation the values of *H* and *M* are shifted to *H-minimum_rating* and *M-minimum_rating* so that the lowest value is 0. This is done to support negative labels.
- X_{jk} is the number times where $H = j$ and $M = k$.
- m_{jk} is the percent chance agreement:

  $m_{jk} = \sum_{k=1}^{K} \frac{n_{k+}}{N}\frac{n_{+k}}{N}$, where

  - n_{k+} - total number of responses where $H_i = k$

- n_{+k} - total number of responses where $M_i = k$

Kappa is computed using skll.metrics.kappa with `weights` set to `None` and `allow_off_by_one` set to `False` (default).

Note: See this discussion for the explanation of how the SKLL implementation differs from the scikit-learn implementation. The two implementations might produce different results if the matrix contains missing labels. For example, consider the hypothetical scenario where our predictions only contain the labels `1`, `2`, and `4`. In the SKLL implementation, the missing `3` will be automatically added to the list of labels whereas in the scikit-learn implementation, the `3` would only be added if a complete list of labels was passed to the function via the optional `labels` keyword argument.

Quadratic weighted kappa (QWK)

Unlike *Cohen's kappa* which is only computed for rounded scores, quadratic weighted kappa is computed for continuous scores using the following formula:

$$QWK = \frac{2 * Cov(M, H)}{Var(H) + Var(M) + (\bar{M} - \bar{H})^2}$$

Note that in this case the variances and covariance are computed by dividing by `N` and not by `N-1`, as in other cases.

QWK is computed using *rsmtool.utils.quadratic_weighted_kappa* with `ddof` set to `0`.

See Haberman (2019) for the full derivation of this formula. The discrete case is simply treated as a special case of the continuous one.

Note: In RSMTool v6.x and earlier QWK was computed using skll.metrics.kappa with `weights` set to `"quadratic"`. Continuous scores were rounded for computation. Both formulas produce the same scores for discrete (rounded scores) but QWK values for continuous scores computed by RSMTool starting with v7.0 will be *different* from those computed by earlier versions.

Pearson Correlation coefficient (r)

$$r = \frac{\sum_{i=1}^{N}(H_i - \bar{H})(M_i - \bar{M})}{\sqrt{\sum_{i=1}^{N}(H_i - \bar{H})^2 \sum_{i=1}^{N}(M - \bar{M})^2}}$$

Pearson correlation coefficients is computed using scipy.stats.pearsonr.

If the variance of human or system scores is `0` (all scores are the same) or only one response is available, RSMTool returns `None`.

Note: In *scipy* v1.4.1 and later, the implementation uses the following formula:

$$r = \frac{H - \bar{H}}{\left\|H - \bar{H}\right\|_2} \cdot \frac{M - \bar{M}}{\left\|M - \bar{M}\right\|_2}$$

This implementation is more robust to very large values but is more likely to return a value slightly smaller than 1 (for example, 0.9999999999999998) for perfect correlation when *n* is small. See this comment for further detail.

Standardized mean difference (SMD)

This metrics ensures that the distribution of system scores is centered on a point close to what is observed with human scoring.

$$SMD = \frac{\bar{M} - \bar{H}}{\sigma_H}$$

SMD between system and human scores is computed using *rsmtool.utils.standardized_mean_difference* with the `method` argument set to `"unpooled"`.

Note: In RSMTool v6.x and earlier SMD was computed with the `method` argument set to `"williamson"` as described in Williamson et al. (2012). The values computed by RSMTool starting with v7.0 will be *different* from those computed by earlier versions.

Mean squared error (MSE)

The mean squared error of a machine score as a predictor of observed human score H:

$$MSE(H|M) = \frac{1}{N}\sum_{i=1}^{N}(H_i - M_i)^2$$

MSE is computed using sklearn.metrics.mean_squared_error.

Proportional reduction in mean squared error for observed score (R2)

$$R2 = 1 - \frac{MSE(H|M)}{\sigma_H^2}$$

R2 is computed using sklearn.metrics.r2_score. If only one response is available, RSMTool returns `None`.

1.3.2 Accuracy Metrics (True score)

According to test theory, an observed score is a combination of the true score T and a measurement error. The true score cannot be observed, but its distribution parameters can be estimated from observed scores. Such an estimation requires that two human scores be available for *at least a* subset of responses in the evaluation set since these are necessary to estimate the measurement error component.

Evaluating system against true score produces performance estimates that are robust to errors in human scores and remain stable even when human-human agreeement varies (see Loukina et al. (2020).

The true score evaluations computed by RSMTool are available in the *intermediate file* `true_score_eval`.

Proportional reduction in mean squared error for true scores (PRMSE)

PRMSE shows how well system scores can predict true scores. This metric generally varies between 0 (random prediction) and 1 (perfect prediction), although in some cases in can take negative values (suggesting a very bad fit) or exceed 1 (suggesting that the sample size is too small to reliably estimate rater error variance).

PRMSE for true scores is defined similarly to *PRMSE for observed scores*, but with the true score T used instead of the observed score H, that is, as the percentage of variance in the true scores explained by the system scores.

$$PRMSE = 1 - \frac{MSE(T|M)}{\sigma_T^2}$$

In the simple case where all responses have two human scores, $MSE(T|M)$ (**mean squared error when predicting true score with system score**) and σ_T^2 (**variance of true score**) are estimated from their observed score counterparts $MSE(H|M)$ and σ_H^2 as follows:

- $\hat{H}$ is used instead of H to compute $MSE(\hat{H}|M)$ and $\sigma_{\hat{H}}^2$. $\hat{H}$ is the average of two human scores for each response ($\hat{H}_i = \frac{H_i + H2_i}{2}$). These evaluations use $\hat{H}$ rather than H because the measurement errors for each rater are assumed to be random and, thus, can partially cancel out making the average $\hat{H}$ closer to true score T than H or $H2$.
- To compute estimates for true scores, the values for observed scores are adjusted for **variance of measurement errors** (σ_e^2) in human scores defined as:

$$\sigma_e^2 = \frac{1}{2 \times N_2} \sum_{i=1}^{N_2} (H_i - H2_i)^2$$

In the simple case, where **all responses are double-scored**, $MSE(T|M)$ is estimated as:

$$MSE(T|M) = MSE(\hat{H}|M) - \frac{1}{2}\sigma_e^2$$

and σ_T^2 is estimated as:

$$\sigma_T^2 = \sigma_{\hat{H}}^2 - \frac{1}{2}\sigma_e^2$$

The PRMSE formula implemented in RSMTool is more general and can also handle the case where the number of available ratings varies across the responses (e.g. **only a subset of responses is double-scored**). While `rsmtool` and `rsmeval` only support evaluations with two raters, the implementation of the PRMSE formula available via the *API* supports cases where some of the responses have **more than two** ratings available. The formula was derived by Matt S. Johnson and is explained in more detail in Loukina et al. (2020).

In this case, the variance of rater errors is computed as a pooled variance estimator.

We first calculate the within-subject variance of human ratings for each response, V_i, using denominator $c_i - 1$:

$$V_i = \frac{\sum_{j=1}^{c} (H_{i,j} - \bar{H}_i)^2}{c_i - 1}$$

where

- $H_{i,j}$ is the human score assigned by rater j to response i
- c_i is the total number of human scores available for response i. For double-scored responses this equals 2.
- $\bar{H}_i$ is the average human rating for response i.

We then take a weighted average of those within-responses variances:

$$\sigma_e^2 = \frac{\sum_{i=1}^{N} V_i * (c_i - 1)}{\sum_{i=1}^{N} (c_i - 1)}$$

The **true score variance** σ_T^2 is then estimated as

$$\sigma_T^2 = \frac{\sum_{i=1}^{N} c_i (\bar{H}_i - \bar{H})^2 - (N-1)\sigma_e^2}{c_\cdot - \frac{\sum_{i=1}^{N} c_i^2}{c_\cdot}}$$

where

- $c_\cdot = \sum_{i=1}^{N} c_i$ is the total number of observed human scores.
- $\bar{H}_i$ is the average human rating for response i. For responses with only one rating this will be the single human score *H*.

Mean squared error $MSE(T|M)$ is estimated as:

$$MSE(T|M) = \frac{1}{c.}\left(\sum_{i=1}^{N} c_i(\bar{H}_i - M_i)^2 - N\sigma_e^2\right)$$

The formulas are derived to ensure consistent results regardless of the number of raters and of the number of ratings availvable for each response.

PRMSE is computed using the *rsmtool.utils.prmse_true* function.

In some cases, it may be appropriate to compute variance of human errors using a different sample than the one used for main evaluations. This can be accomplished using *rsmtool.utils.variance_of_errors* and using an optional configuration field *rater_error_variance* in *rsmtool* or *rsmtool*

Note: The PRMSE formula assigns higher weight to discrepancies between system scores and human scores when human score is the average of two or more human scores than when the human score is based on a single score.

1.3.3 Fairness

Fairness of automated scores is an important component of RSMTool evaluations (see Madnani et al, 2017).

When defining an experiment, the RSMTool user has the option of specifying which subgroups should be considered for such evaluations using *subgroups* field. These subgroups are then used in all fairness evaluations.

All fairness evaluations are conducted on the evaluation set. The metrics are only computed for either *raw_trim* or *scale_trim* scores (see *score postprocessing* for further details) depending on the value of *use_scaled_predictions* in RSMTool or the value of *scale_with* in RSMEval.

Differences between standardized means for subgroups (DSM)

This is a standard evaluation used for evaluating subgroup differences. The metrics are available in the *intermediate files* `eval_by_<SUBGROUP>`.

DSM is computed as follows:

1. For each group, get the z-score for each response :math:i, using the $\bar{H}$, $\bar{M}$, σ_H, and σ_S for system and human scores for the whole evaluation set:

$$z_{H_i} = \frac{H_i - \bar{H}}{\sigma_H}$$

$$z_{M_i} = \frac{M_i - \bar{M}}{\sigma_M}$$

2. For each response i, calculate the difference between machine and human scores: $z_{M_i} - z_{H_i}$
3. Calculate the mean of the difference $z_{M_i} - z_{H_i}$ by subgroup of interest.

DSM is computed using *rsmtool.utils.difference_of_standardized_means* with:

`population_y_true_observe_mn` = $\bar{H}$ for the whole evaluation set

`population_y_pred_mn` = $\bar{M}$ for the whole evaluation set

`population_y_true_observed_sd` = σ_H for the whole evaluation set

`population_y_pred_sd` = σ_M for the whole evaluation set

Note: In RSMTool v6.x and earlier, subgroup differences were computed using *standardized mean difference* with the `method` argument set to `"williamson"`. Since the differences computed in this manner were very sensitive to score distributions, RSMTool no longer uses this function to compute subgroup differences starting with v7.0.

Additional fairness evaluations

Starting with v7.0, RSMTool includes additional fairness analyses suggested in Loukina, Madnani, & Zechner, 2019. The computed metrics from these analyses are available in *intermediate files* `fairness_metrics_by_<SUBGROUP>`.

These include:

- Overall score accuracy: percentage of variance in squared error $(M - H)^2$ explained by subgroup membership
- Overall score difference: percentage of variance in absolute error $(M - H)$ explained by subgroup membership
- Conditional score difference: percentage of variance in absolute error $(M - H)$ explained by subgroup membership when controlling for human score

Please refer to the paper for full descriptions of these metrics.

The fairness metrics are computed using *rsmtool.fairness_utils.get_fairness_analyses*.

1.3.4 Human-human agreement

If scores from a second human (*H2*) are available, RSMTool computes the following additional metrics for human-human agreement using only the N_2 responses, including only responses that contain numeric values for both the H and $H2$ columns.

The computed metrics are available in the *intermediate file* `consistency`.

Percent exact agreement

Same as *percent exact agreement for observed scores* but substituting $H2$ for M.

Percent exact + ajdacent agreement

Same as *percent adjacent agreement for observed scores* but substituting $H2$ for M and N_2 for N.

Cohen's kappa

Same as *Cohen's kappa for observed scores* but substituting $H2$ for M and N_2 for N.

Quadratic weighted kappa (QWK)

Same as *QWK for observed scores* but substituting $H2$ for M and N_2 for N.

Pearson Correlation coefficient (r)

Same as *r for observed scores* but substituting $H2$ for M and N_2 for N.

Standardized mean difference (SMD)

$$SMD = \frac{\bar{H}2 - \bar{H}1}{\sqrt{\frac{\sigma_H^2 + \sigma_{H2}^2}{2}}}$$

Unlike *SMD for human-system scores*, the denominator in this case is the "pooled" standard deviation of $H1$ and $H2$.

Therefore, SMD between two human scores is computed using *rsmtool.utils.standardized_mean_difference* with the `method` argument set to `"pooled"`.

Note: In RSMTool v6.x and earlier, SMD was computed with the `method` argument set to `"williamson"` as described in Williamson et al. (2012). Starting with v7.0, the values computed by RSMTool will be *different* from those computed by earlier versions.

1.4 Installation

Note that RSMTool currently works with Python 3.8, 3.9, and 3.10.

1.4.1 Installing with conda

Currently, the recommended way to install RSMTool is by using the `conda` package manager. If you have already installed `conda`, you can skip straight to Step 2.

1. To install `conda`, follow the instructions on this page.
2. Create a new conda environment (say, `rsmtool`) and install the RSMTool conda package for your preferred Python version. For example, for Python 3.8, run:

```
conda create -n rsmtool -c conda-forge -c ets python=3.8 rsmtool
```

3. Activate this conda environment by running `conda activate rsmtool`. You should now have all of the RSMTool command-line utilities in your path.[1]
4. From now on, you will need to activate this conda environment whenever you want to use RSMTool. This will ensure that the packages required by RSMTool will not affect other projects.

RSMTool can also be downloaded directly from GitHub.

1.4.2 Installing with pip

You can also use `pip` to install RSMTool instead of `conda`. To do so, simply run:

```
pip install rsmtool
```

Note that if you are on macOS, you will need to have the following line in your `.bashrc` for RSMTool to work properly:

```
export MPLBACKEND=Agg
```

[1] For older versions of conda, you may have to do `source activate rsmtool` on Linux/macOS and `activate rsmtool` on Windows.

1.5 Tutorial

First you'll want to *install RSMTool* and make sure you the conda environment in which you installed it is activated.

1.5.1 Workflow

Important: Although this tutorial provides feature values for the purpose of illustration, RSMTool does *not* include any functionality for feature extraction; the tool is *designed for researchers* who use their own NLP/Speech processing pipeline to extract features for their data.

Once you have the features extracted from your data, using RSMTool is fairly straightforward:

1. Make sure your data is split into a training set and a held-out evaluation set. You will need two files containing the feature values: one for each set. The files should be in one of the *supported formats*
2. Create an *experiment configuration file* describing the modeling experiment you would like to run.
3. Run that configuration file with *rsmtool* and generate the experiment HTML report as well as the *intermediate CSV files*.
4. Examine the HTML report to check various aspects of model performance.

Note that the above workflow does not use the customization features of RSMTool, e.g., *choosing which sections to include in the report* or *adding custom analyses sections* etc. However, we will stick with this workflow for our tutorial since it is likely to be the most common use case.

1.5.2 ASAP Example

Let's see how we can apply this basic RSMTool workflow to a simple example based on a 2012 Kaggle competition on automated essay scoring called the Automated Student Assessment Prize (ASAP) contest. As part of that contest, responses to 8 different essay questions written by students in grades 6-10 were provided. The responses were scored by humans and given a holistic score indicating the English proficiency of the student. The goal of the contest was to build an automated scoring model with the highest accuracy on held-out evaluation data.

For our tutorial, we will use one of the questions from this data to illustrate how to use RSMTool to train a scoring model and evaluate its performance. All of the data we refer to below can be found in the `examples/rsmtool` folder in the github repository.

Extract features

Kaggle provides the actual text of the responses along with the human scores in a `.tsv` file. The first step, as with any automated scoring approaches, is to extract features from the written (or spoken) responses that might be useful in predicting the score we are interested in. For example, some features that might be useful for predicting the English proficiency of a response are:

- Does the response contain any grammatical errors and if so, how many and of what kind since some grammatical errors are more serious than others.
- Is the response is well organized, e.g., whether there is a clear thesis statement and a conclusion, etc.
- Is the response coherent, i.e., do the ideas expressed in the various paragraphs well connected?
- Does the response contain any spelling errors?

For more examples of what features might look like, we refer the reader to this paper by Attali & Burstein[1]. For our ASAP2 data, we extracted the following four features:

- `GRAMMAR`: indicates how grammatically fluent the responses is.
- `MECHANICS`: indicates how free of mechanical errors (e.g., spelling errors) the response is.
- `DISCOURSE`: indicates whether the discourse transitions in the response make sense.
- `ORGANIZATION`: indicates whether the response is organized well.

Note: These features were extracted using proprietary NLP software. To reiterate, RSMTool does *not* include any NLP/speech components for feature extraction.

Create a configuration file

The next step is to create an *RSMTool experiment configuration file* in `.json` format.

```
1  {
2      "experiment_id": "ASAP2",
3      "train_file": "train.csv",
4      "test_file": "test.csv",
5      "model": "LinearRegression",
6      "description": "A model which uses all features and a LinearRegression.",
7      "test_label_column": "score",
8      "train_label_column": "score",
9      "use_scaled_predictions": true,
10     "trim_min": 1,
11     "trim_max": 6,
12     "id_column": "ID",
13     "second_human_score_column": "score2",
14     "length_column": "LENGTH"
15 }
```

Let's take a look at the options in our configuration file.

- **Line 2**: We define an experiment ID
- **Lines 3-4**: We list the paths to our training and evaluation files with the feature values. For this tutorial we used `.csv` format, but RSMTool also supports several other *input file formats*.
- **Line 5**: We choose to use a linear regression model to combine those features into a score.
- **Line 6**: We also provide a description which will be included in the experiment report.
- **Lines 7-8**: These two fields indicate that the human scores in the two `.csv` files are located in columns named `score`.
- **Lines 9**: Next, we indicate that we would like to use the *scaled scores* for our evaluation analyses.
- **Lines 10-11**: These fields indicate that the lowest score on the scoring scale is a 1 and the highest score is a 6. This information is usually part of the rubric used by human graders.
- **Line 12**: This field indicates that the unique IDs for the responses in the two `.csv` files are located in columns named `ID`.

[1] Attali, Y., & Burstein, J. (2006). Automated essay scoring with e-rater® V.2. Journal of Technology, Learning, and Assessment, 4(3). https://ejournals.bc.edu/index.php/jtla/article/download/1650/1492

- **Line 13**: This field indicates that scores from a second set of human graders are also available (useful for comparing the agreement between human-machine scores to the agreement between two sets of humans) and are located in the `score2` column in the test set `.csv` file.
- **Line 14**: This field indicates that response lengths are also available for the training data in a column named `LENGTH`.

Documentation for all of the available configuration options is available *here*.

Note:

1. For this example, we are using *all* of the non-metadata columns in the training and evaluation `.csv` files as features in the model. However, it is also possible to *choose specific columns* to be used for training the model.
2. You can also use our nifty capability to *automatically generate* `rsmtool` configuration files rather than creating them manually.

Run the experiment

Now that we have our features in right format and our configuration file in `.json` format, we can use the *rsmtool* command-line script to run our modeling experiment.

```
$ cd examples/rsmtool
$ rsmtool config_rsmtool.json
```

This should produce output like:

```
Output directory: /Users/nmadnani/work/rsmtool/examples/rsmtool
Loading experiment data
Reading configuration file: /Users/nmadnani/work/rsmtool/examples/rsmtool/config_
↪rsmtool.json
Reading training data: /Users/nmadnani/work/rsmtool/examples/rsmtool/train.csv
Reading evaluation data: /Users/nmadnani/work/rsmtool/examples/rsmtool/test.csv
Pre-processing training and test set features
Saving training and test set data to disk
Training LinearRegression model
The exact solution is  x = 0
Running analyses on training set
Generating training and test set predictions
Processing test set predictions
Saving training and test set predictions to disk
Scaling the coefficients and saving them to disk
Running analyses on test set predictions
Starting report generation
Merging sections
Exporting HTML
Executing notebook with kernel: python3
```

Once the run finishes, you will see the `output`, `figure`, `report`, and `feature` sub-directories in the current directory. Each of these directories contain *useful information* but we are specifically interested in the `report/ASAP2_report.html` file, which is the final RSMTool experiment report.

Examine the report

Our experiment report contains all the information we would need to determine how well our linear regression model with four features is doing. It includes:

1. Descriptive feature statistics.
2. Inter-feature correlations.
3. Model fit and diagnostics.
4. Several different metrics indicating how well the machine's scores agree with the humans'.

. . . and *much more*.

If we are not satisfied with the performance of the model, we can then take specific action, e.g., either add new features or perhaps try a more sophisticated model. And then run `rsmtool` again and examine the report. Building a scoring model is generally an iterative process.

1.5.3 What's next?

Next, you should read the detailed documentation on *rsmtool* since the tutorial only covered a very basic example. If you are interested in RSMTool but your use case is more nuanced, we have probably got you *covered*.

References

1.6 Using RSMTool

For most users, the primary means of using RSMTool will be via the command-line utility `rsmtool`. We refer to each run of `rsmtool` as an "experiment".

1.6.1 Input

`rsmtool` requires a single argument to run an experiment: the path to *a configuration file*. It can also take an output directory as an optional second argument. If the latter is not specified, `rsmtool` will use the current directory as the output directory.

Here are all the arguments to the `rsmtool` command-line script.

config_file
: The *JSON configuration file* for this experiment.

output_dir `(optional)`
: The output directory where all the files for this experiment will be stored.

-f, --force
: If specified, the contents of the output directory will be overwritten even if it already contains the output of another rsmtool experiment.

-h, --help
: Show help message and exist.

-V, --version
: Show version number and exit.

Experiment configuration file

This is a file in `.json` format that provides overall configuration options for an `rsmtool` experiment. Here's an example configuration file for `rsmtool`.

Note: To make it easy to get started with `rsmtool`, we provide a way to **automatically generate** configurations file both interactively as well as non-interactively. Novice users will find interactive generation more helpful while more advanced users will prefer non-interactive generation. See *this page* for more details.

Next, we describe all of the `rsmtool` configuration fields in detail. There are four required fields and the rest are all optional. We first describe the required fields and then the optional ones (sorted alphabetically).

experiment_id

An identifier for the experiment that will be used to name the report and all *intermediate files*. It can be any combination of alphanumeric values, must *not* contain spaces, and must *not* be any longer than 200 characters.

model

The machine learner you want to use to build the scoring model. Possible values include *built-in linear regression models* as well as all of the learners available via SKLL. With SKLL learners, you can customize the *tuning objective* and also *compute expected scores as predictions*.

train_file

The path to the training data feature file in one of the *supported formats*. Each row should correspond to a single response and contain numeric feature values extracted for this response. In addition, there should be a column with a unique identifier (ID) for each response and a column with the human score for each response. The path can be absolute or relative to the location of config file.

test_file

The path to the evaluation data feature file in one of the *supported formats*. Each row should correspond to a single response and contain numeric feature values extracted for this response. In addition, there should be a column with a unique identifier (ID) for each response and a column with the human score for each response. The path can be absolute or relative to the location of config file.

Note:

1. For both the training and evaluation files, the default behavior of `rsmtool` is to look for a column named `spkitemid` in order to get the unique IDs for each response and a column named `sc1` to get train/test labels. The optional fields *id_column* and *train_label_column* can be used to specify different names for these columns.
2. `rsmtool` also assumes that all other columns present in these files (other than those containing IDs and labels) contain feature values. If this is not the case, one can use other configuration file fields to identify columns containing non-feature information useful for various analyses, e.g., *second_human_score_column*, *flag_column*, *subgroups* et cetera below.

3. Any columns not explicitly identified in (1) and (2) will be considered feature columns and used by `rsmtool` in the model. To use only a *subset* of these remaining columns as features, one can employ the four optional fields *features*, *feature_subset_file*, *feature_subset*, and *sign*. See *selecting feature columns* for more details on how to achieve this.

---

candidate_column *(Optional)*

The name for an optional column in the training and test data containing unique candidate IDs. Candidate IDs are different from response IDs since the same candidate (test-taker) might have responded to multiple questions.

custom_sections *(Optional)*

A list of custom, user-defined sections to be included into the final report. These are IPython notebooks (`.ipynb` files) created by the user. The list must contains paths to the notebook files, either absolute or relative to the configuration file. All custom notebooks have access to some *pre-defined variables*.

description *(Optional)*

A brief description of the experiment. This will be included in the report. The description can contain spaces and punctuation. It's blank by default.

exclude_zero_scores *(Optional)*

By default, responses with human scores of 0 will be excluded from both training and evaluation set. Set this field to `false` if you want to keep responses with scores of 0. Defaults to `true`.

feature_subset *(Optional)*

Name of the pre-defined feature subset to be used if using *subset-based column selection*.

feature_subset_file *(Optional)*

Path to the feature subset file if using *subset-based column selection*.

features *(Optional)*

Path to the file with list of features if using *fine-grained column selection*. Alternatively, you can pass a `list` of feature names to include in the experiment.

file_format *(Optional)*

The format of the *intermediate files*. Options are `csv`, `tsv`, or `xlsx`. Defaults to `csv` if this is not specified.

flag_column *(Optional)*

This field makes it possible to only use responses with particular values in a given column (e.g. only responses with a value of `0` in a column called `ADVISORY`). The field takes a dictionary in Python format where the keys are the names of the columns and the values are lists of values for responses that will be used to train the model. For example, a value of `{"ADVISORY": 0}` will mean that `rsmtool` will *only* use responses for which the `ADVISORY` column has the value 0. If this field is used without `flag_column_test`, the conditions will be applied to *both* training and evaluation set and the specified columns must be present in both sets. When this field is used in conjunction with `flag_column_test`, the conditions will be applied to *training set only* and the specified columns must be present in the training set. Defaults to `None`.

Note: If several conditions are specified (e.g., `{"ADVISORY": 0, "ERROR": 0}`) only those responses which satisfy *all* the conditions will be selected for further analysis (in this example, these will be the responses where the `ADVISORY` column has a value of 0 *and* the `ERROR` column has a value of 0).

Note: When reading the values in the supplied dictionary, `rsmtool` treats numeric strings, floats and integers as the same value. Thus `1`, `1.0`, `"1"` and `"1.0"` are all treated as the `1.0`.

---

flag_column_test *(Optional)*

This field makes it possible to only use a separate Python flag dictionary for the evaluation set. If this field is not passed, and `flag_column` is passed, then the same advisories will be used for both training and evaluation sets.

When this field is used, the specified columns must be present in the evaluation set. Defaults to `None` or *flag_column'*, if `flag_column` is present. Use `flag_column_test` only if you want filtering of the test set.

Note: When used, `flag_column_test` field determines *all* filtering conditions for the evaluation set. If it is used in conjunction with `flag_column` field, the filtering conditions defined in `flag_column` will *only* be applied to the training set. If you want to apply a subset of conditions to both partitions with additional conditions applied to the evaluation set only, you will need to specify the overlapping conditions separately for each partition.

---

general_sections *(Optional)*

RSMTool provides pre-defined sections for `rsmtool` (listed below) and, by default, all of them are included in the report. However, you can choose a subset of these pre-defined sections by specifying a list as the value for this field.

- `data_description`: Shows the total number of responses in training and evaluation set, along with any responses have been excluded due to non-numeric features/scores or *flag columns*.
- `data_description_by_group`: Shows the total number of responses in training and evaluation set for each of the *subgroups* specified in the configuration file. This section only covers the responses used to train/evaluate the model.
- `feature_descriptives`: Shows the descriptive statistics for all raw feature values included in the model:
  - a table showing mean, standard deviation, min, max, correlation with human score etc.;
  - a table with percentiles and outliers; and
  - a barplot showing the number of truncated outliers for each feature.

- `features_by_group`: Shows boxplots with distributions of raw feature values by each of the *subgroups* specified in the configuration file.
- `preprocessed_features`: Shows analyses of preprocessed features:
  - histograms showing the distributions of preprocessed features values;
  - the correlation matrix between all features and the human score;
  - a barplot showing marginal and partial correlations between all features and the human score, and, optionally, response length if *length_column* is specified in the config file.
- `dff_by_group`: Differential feature functioning by group. The plots in this section show average feature values for each of the *subgroups* conditioned on human score.
- `consistency`: Shows metrics for *human-human agreement*, the difference ("degradation") between the human-human and human-system agreement, and the disattenuated human-machine correlations. This notebook is only generated if the config file specifies *second_human_score_column*.
- `model`: Shows the parameters of the learned regression model. For linear models, it also includes the standardized and relative coefficients as well as model diagnostic plots.
- `evaluation`: Shows the *standard set of evaluations* recommended for scoring models on the evaluation data:
  - a table showing human-system association metrics;
  - the confusion matrix; and
  - a barplot showing the distributions for both human and machine scores.
- `evaluation_by_group`: Shows barplots with the main evaluation metrics by each of the subgroups specified in the configuration file.
- `fairness_analyses`: Additional *fairness analyses* suggested in Loukina, Madnani, & Zechner, 2019. The notebook shows:
  - percentage of variance in squared error explained by subgroup membership
  - percentage of variance in raw (signed) error error explained by subgroup membership
  - percentage of variance in raw (signed) error explained by subgroup membership when controlling for human score
  - plots showing estimates for each subgroup for each model
- `true_score_evaluation`: evaluation of system scores against the *true scores* estimated according to test theory. The notebook shows:
  - Number of single and double-scored responses.
  - Variance of human rater errors and estimated variance of true scores
  - Mean squared error (MSE) and proportional reduction in mean squared error (PRMSE) when predicting true score with system score.
- `pca`: Shows the results of principal components analysis on the processed feature values:
  - the principal components themselves;
  - the variances; and
  - a Scree plot.

  The analysis keeps all components. The total number of components usually equals the total number of features. In cases where the total number of responses is smaller than the number of features, the number of components is the same as the number of responses.

- `intermediate_file_paths`: Shows links to all of the intermediate files that were generated while running the experiment.
- `sysinfo`: Shows all Python packages along with versions installed in the current environment while generating the report.

id_column *(Optional)*

The name of the column containing the response IDs. Defaults to `spkitemid`, i.e., if this is not specified, `rsmtool` will look for a column called `spkitemid` in the training and evaluation files.

length_column *(Optional)*

The name for the optional column in the training and evaluation data containing response length. If specified, length is included in the inter-feature and partial correlation analyses. Note that this field *should not* be specified if you want to use the length column as an actual feature in the model. In the latter scenario, the length column will automatically be included in the analyses, like any other feature. If you specify `length_column` *and* include the same column name as a feature in the *feature file*, `rsmtool` will ignore the `length_column` setting. In addition, if `length_column` has missing values or if its standard deviation is 0 (both somewhat unlikely scenarios), `rsmtool` will *not* include any length-based analyses in the report.

min_items_per_candidate *(Optional)*

An integer value for the minimum number of responses expected from each candidate. If any candidates have fewer responses than the specified value, all responses from those candidates will be excluded from further analysis. Defaults to `None`.

min_n_per_group *(Optional)*

A single numeric value or a dictionary with keys as the group names listed in the *subgroups* field and values as the thresholds for the groups. When specified, only groups with *at least* this number of instances will be displayed in the tables and plots contained **in the report**. Note that this parameter *only* affects the HTML report and the figures. For all analyses – including the computation of the population parameters – data from *all* groups will be used. In addition, the *intermediate files* will still show the results for *all* groups.

Note: If you supply a dictionary, it *must* contain a key for *every* subgroup listed in *subgroups* field. If no threshold is to be applied for some of the groups, set the threshold value for this group to 0 in the dictionary.

Note: Any provided thresholds will be applied when displaying the feature descriptive analyses conducted on the training set *and* the results of the performance analyses computed on the evaluation set.

---

predict_expected_scores *(Optional)*

If a probabilistic SKLL classifier is chosen to build the scoring model, then *expected scores* — probability-weighted averages over contiguous, numeric score points — can be generated as the machine predictions instead of the most likely score point, which would be the default for a classifier. Set this field to `true` to compute expected scores as predictions. Defaults to `false`.

Note: You may see slight differences in expected score predictions if you run the experiment on different machines or on different operating systems most likely due to very small probablity values for certain score points which can affect floating point computations.

rater_error_variance *(Optional)*

True score evaluations require an estimate of rater error variance. By default, `rsmtool` will compute this variance from double-scored responses in the data. However, in some cases, one may wish to compute the variance on a different sample of responses. In such cases, this field can be used to set the rater error variance to a precomputed value which is then used as-is by `rsmtool`. You can use the *rsmtool.utils.variance_of_errors* function to compute rater error variance outside the main evaluation pipeline.

second_human_score_column *(Optional)*

The name for an optional column in the test data containing a second human score for each response. If specified, additional information about human-human agreement and degradation will be computed and included in the report. Note that this column must contain either numbers or be empty. Non-numeric values are *not* accepted. Note also that the *exclude_zero_scores (Optional)* option below will apply to this column too.

Note: You do not need to have second human scores for *all* responses to use this option. The human-human agreement statistics will be computed as long as there is at least one response with numeric value in this column. For responses that do not have a second human score, the value in this column should be blank.

section_order *(Optional)*

A list containing the order in which the sections in the report should be generated. Any specified order must explicitly list:

1. Either *all* pre-defined sections if a value for the *general_sections* field is not specified OR the sections specified using *general_sections*, and
2. *All* custom section names specified using *custom_ sections*, i.e., file prefixes only, without the path and without the *.ipynb* extension, and
3. *All* special sections specified using *special_sections*.

select_transformations *(Optional)*

If this option is set to `true` the system will try apply feature transformations to each of the features and then choose the transformation for each feature that yields the highest correlation with human score. The possible transformations are:

- `raw`: no transformation, use original feature value
- `org`: same as raw
- `inv`: 1/x
- `sqrt`: square root

- `addOneInv`: 1/(x+1)
- `addOneLn`: ln(x+1)

We only consider transformations that produce numeric results for *all* values for a given feature column. For example, if a feature column contains a single negative value, the `sqrt` transformation will be ignored even if it would have resulted in the highest correlation with human score for the remaining values. In addition, the `inv` and `addOneInv` transformations are never used for feature columns that contain both positive and negative values.

Defaults to `false`.

See also:

It is also possible to manually apply transformations to any feature as part of the *feature column selection* process.

sign *(Optional)*

Name of the column containing expected correlation sign between each feature and human score if using *subset-based column selection*.

skll_fixed_parameters *(Optional)*

Any fixed hyperparameters to be used if a SKLL model is chosen to build the scoring model. This should be a dictionary with the names of the hyperparameters as the keys. To determine what hyperparameters are available for the SKLL learner you chose, consult the scikit-learn documentation for the learner with the same name as well as the SKLL documentation. Any values you specify here will override both the scikit-learn and SKLL defaults. The values for a key can be string, integer, float, or boolean depending on what the hyperparameter expects. Note that if this option is specified with the *built-in linear regression models*, it will simply be ignored.

skll_objective *(Optional)*

The tuning objective to use if a SKLL model is chosen to build the scoring model. Possible values are the objectives available via SKLL. Defaults to `neg_mean_squared_error` for SKLL regressors and `f1_score_micro` for SKLL classifiers. Note that if this option is specified with the *built-in linear regression models*, it will simply be ignored.

special_sections *(Optional)*

A list specifying special ETS-only sections to be included into the final report. These sections are available *only* to ETS employees via the *rsmextra* package.

standardize_features *(Optional)*

If this option is set to `false` features will not be standardized by subtracting the mean and dividing by the standard deviation. Defaults to `true`.

subgroups *(Optional)*

A list of column names indicating grouping variables used for generating analyses specific to each of those defined subgroups. For example, `["prompt, gender, native_language, test_country"]`. These subgroup columns need to be present in both training *and* evaluation data. If subgroups are specified, `rsmtool` will generate:

- description of the data by each subgroup;
- boxplots showing the feature distribution for each subgroup on the training set; and
- tables and barplots showing human-system agreement for each subgroup on the evaluation set.

test_label_column *(Optional)*

The name for the column containing the human scores in the test data. If set to to `fake`, fake scores will be generated using randomly sampled integers. This option may be useful if you only need descriptive statistics for the data and do not care about the other analyses. Defaults to `sc1`.

train_label_column *(Optional)*

The name for the column containing the human scores in the training data. If set to to `fake`, fake scores will be generated using randomly sampled integers. This option may be useful if you only need descriptive statistics for the data and do not care about the other analyses. Defaults to `sc1`.

Note: All responses with non-numeric values in either `train_label_column` or `test_label_column` and/or those with non-numeric values for relevant features will be automatically excluded from model training and evaluation. By default, zero scores in either `train_label_column` or `test_label_column` will also be excluded. See *exclude_zero_scores (Optional)* if you want to keep responses with zero scores.

---

trim_max *(Optional)*

The single numeric value for the highest possible integer score that the machine should predict. This value will be used to compute the ceiling value for *trimmed (bound)* machine scores as `trim_max` + `trim_tolerance`. Defaults to the highest observed human score in the training data or 10 if there are no numeric human scores available.

trim_min *(Optional)*

The single numeric value for the lowest possible integer score that the machine should predict. This value will be used to compute the floor value for *trimmed (bound)* machine scores as `trim_min` - `trim_tolerance`. Defaults to the lowest observed human score in the training data or 1 if there are no numeric human scores available.

trim_tolerance *(Optional)*

The single numeric value that will be used to pad the trimming range specified in `trim_min` and `trim_max`. This value will be used to compute the ceiling and floor values for *trimmed (bound)* machine scores as `trim_max` + `trim_tolerance` for ceiling value and `trim_min`-`trim_tolerance` for floor value. Defaults to 0.4998.

Note: For more fine-grained control over the trimming range, you can set `trim_tolerance` to *0* and use `trim_min` and `trim_max` to specify the exact floor and ceiling values.

---

use_scaled_predictions *(Optional)*

If set to `true`, certain evaluations (confusion matrices, score distributions, subgroup analyses) will use the scaled machine scores. If set to `false`, these evaluations will use the raw machine scores. Defaults to `false`.

Note: All evaluation metrics (e.g., kappa and pearson correlation) are automatically computed for *both* scaled and raw scores.

---

use_thumbnails *(Optional)*

If set to `true`, the images in the HTML will be set to clickable thumbnails rather than full-sized images. Upon clicking the thumbnail, the full-sized images will be displayed in a separate tab in the browser. If set to `false`, full-sized images will be displayed as usual. Defaults to `false`.

use_truncation_thresholds *(Optional)*

If set to `true`, use the `min` and `max` columns specified in the `features` file to clamp outlier feature values. This is useful if users would like to clamp feature values based on some pre-defined boundaries, rather than having these boundaries calculated based on the training set. Defaults to `false`.

Note: If `use_truncation_thresholds` is set, a `features` file _must_ be specified, and this file _must_ include `min` and `max` columns. If no `feature` file is specified or these columns are missing, an error will be raised.

1.6.2 Output

`rsmtool` produces a set of folders in the experiment output directory. This is either the current directory in which `rsmtool` is run or the directory specified as the second optional command-line argument.

report

This folder contains the final RSMTool report in HTML format as well in the form of a Jupyter notebook (a `.ipynb` file).

output

This folder contains all of the *intermediate files* produced as part of the various analyses performed, saved as `.csv` files. `rsmtool` will also save in this folder a copy of the *configuration file*. Fields not specified in the original configuration file will be pre-populated with default values.

figure

This folder contains all of the figures generated as part of the various analyses performed, saved as `.svg` files.

feature

This folder contains a `.csv` file that lists all features, signs and transformation as used in the *final* model, taking into account any manual or automatic feature selection. See *feature column selection* for more details.

1.6.3 Selecting Feature Columns

By default, `rsmtool` will use all columns included in the training and evaluation data files as features. The only exception are any columns explicitly identified in the configuration file as containing non-feature information (e.g., *id_column*, *train_label_column*, *test_label_column*, etc.)

However, there are certain scenarios in which it is useful to choose specific columns in the data to be used as features. For example, let's say that you have a large number of very different features and you want to use a different subset of features to score different types of questions on a test. In this case, the ability to easily choose the desired features for any `rsmtool` experiment becomes quite important. The alternative of manually pre-processing the data to remove the features you don't need is quite cumbersome.

There are two ways to select specific columns in the data as features:

1. **Fine-grained column selection**: In this method, you manually create a list of the columns that you wish to use as features for an `rsmtool` experiment. See *fine-grained selection* for more details.
2. **Subset-based column selection**: In this method, you can pre-define subsets of features and then select entire subsets at a time for any `rsmtool` experiment. See *subset-based selection* for more details.

While fine-grained column selection is better for a single experiment, subset-based selection is more convenient when you need to run several experiments with somewhat different subsets of features.

Warning: `rsmtool` will filter the training and evaluation data to remove any responses with non-numeric values in any of the feature columns before training the model. If your data includes a column containing string values and you do *not* use any of these methods of feature selection *nor* specify this column as the `id_column` or the `candidate_column` or a `subgroup` column, `rsmtool` will filter out *all* the responses in the data.

Fine-grained column selection

To manually select columns to be used as features, you can provide a data file in one of the *supported formats*. The file must contain a column named `feature` which specifies the names of the feature columns that should be used for scoring model building. For additional flexibility, the same file also allows you to describe transformations to be applied to the values in these feature columns before being used in the model. The path to this file should be set as an argument to `features` in the experiment configuration file. (Note: If you do not wish to perform any feature transformations, but would simply like to select certain feature columns to include, you can also pass a `list` of feature names as an arguement to `features`.)

Here's an example of what such a file might look like.

```
feature,transform,sign
feature1,raw,1
feature2,inv,-1
```

There is one required column and two optional columns.

feature

The exact name of the column in the training and evaluation data files, including capitalization. Column names cannot contain hyphens. The following strings are reserved and cannot not be used as feature column names: `spkitemid`, `spkitemlab`, `itemType`, `r1`, `r2`, `score`, `sc`, `sc1`, and `adj`. In addition, any column names provided as values for `id_column`, `train_label_column`, `test_label_column`, `length_column`, `candidate_column`, and `subgroups` may also not be used as feature column names.

transform (optional)

A transformation that should be applied to the column values before using it in the model. Possible values are:

- `raw`: no transformation, use original value
- `org`: same as raw
- `inv`: 1/x
- `sqrt`: square root
- `addOneInv`: 1/(x+1)
- `addOneLn`: ln(x+1)

Note that `rsmtool` will raise an exception if the values in the data do not allow the supplied transformation (for example, if `inv` is applied to a column which has 0 values). If you really want to use the tranformation, you must pre-process your training and evaluation data files to remove the problematic cases.

If the feature file contains no `transform` column, `rsmtool` will use the original values for all features (`raw` trasform).

sign (optional)

After transformation, the column values will be multiplied by this number, which can be either `1` or `-1` depending on the expected sign of the correlation between transformed feature and human score. This mechanism is provided to ensure that all features in the final models have a positive correlation with the score, if that is so desired by the user.

If the feature file contains no `sign` column, `rsmtool` will multiply all values by `1`.

When determining the sign, you should take into account the correlation between the original feature and the score as well as any applied transformations. For example, if you use feature which has a negative correlation with the human score and apply `sqrt` transformation, `sign` should be set to `-1`. However, if you use the same feature but apply the `inv` transformation, `sign` should now be set to `1`.

To ensure that this is working as expected, you can check the sign of correlations for both raw and processed features in the final report.

Note: You can use the fine-grained method of column selection in combination with a *model with automatic feature selection*. In this case, the features that end up being used in the final model can be found in the `.csv` file in the `feature` folder in the experiment output directory.

Subset-based column selection

For more advanced users, `rsmtool` offers the ability to assign columns to named subsets in a data file in one of the *supported formats* and then select a set of columns by simply specifying the name of that pre-defined subset.

If you want to run multiple `rsmtool` experiments, each choosing from a large number of features, generating a separate *feature file* for each experiment listing columns to use can quickly become tedious.

Instead you can define feature subsets by providing a subset definition file in one of the *supported formats* which lists *all* feature names under a column named `feature`. Each subset is an additional column with a value of either `0` (denoting that the feature does *not* belong to the subset named by that column) or `1` (denoting that the feature does belong to the subset named by that column).

Here's an example of a subset definition file, say `subset.csv`.

```
feature,A,B
feature1,0,1
feature2,1,1
feature3,1,0
```

In this example, `feature2` and `feature3` belong to a subset called "A" and `feature1` and `feature1` and `feature2` belong to a subset called "B".

This feature subset file can be provided to `rsmtool` using the *feature_subset_file* field in the configuration file. Then, to select a particular pre-defined subset of features, you simply set the *feature_subset* field in the configuration file to the name of the subset that you wish to use.

Then, in order to use feature subset "A" (`feature2` and `feature3`) in an experiment, we need to set the following two fields in our experiment configuration file:

```
{
    ...
    "feature_subset_file": "subset.csv",
    "feature_subset": "A",
    ...
}
```

Transformations

Unlike in *fine-grained selection*, the feature subset file does not list any transformations to be applied to the feature columns. However, you can automatically select transformation for each feature *in the selected subset* by applying all possible transforms and identifying the one which gives the highest correlation with the human score. To use this functionality set the *select_transformations* field in the configuration file to `true`.

Signs

Some guidelines for building scoring models require all coefficients in the model to be positive and all features to have a positive correlation with human score. `rsmtool` can automatically flip the sign for any pre-defined feature subset. To use this functionality, the feature subset file should provide the expected correlation sign between each feature and human score under a column called `sign_<SUBSET>` where `<SUBSET>` is the name of the feature subset. Then, to tell `rsmtool` to flip the sign for this subset, you need to set the *sign* field in the configuration file to `<SUBSET>`.

To understand this, let's re-examine our earlier example of a subset definition file `subset.csv`, but with an additional column.

```
feature,A,B,sign_A
feature1,0,1,+
feature2,1,1,-
feature3,1,0,+
```

Then, in order to use feature subset "A" (`feature2` and `feature3`) in an experiment with the sign of `feature3` flipped appropriately (multiplied by -1) to ensure positive correlations with score, we need to set the following three fields in our experiment configuration file:

```
{
    ...
    "feature_subset_file": "subset.csv",
    "feature_subset": "A",
    "sign": "A"
    ...
}
```

Note: If *select_transformations* is set to `true`, `rsmtool` is intelligent enough to take it into account when flipping the signs. For example, if the expected correlation sign for a given feature is negative, `rsmtool` will multiply the feature values by `-1` if the `sqrt` transform has the highest correlation with score. However, if the best transformation turns out to be `inv` – which already changes the polarity of the feature – no such multiplication will take place.

1.6.4 Intermediate files

Although the primary output of RSMTool is an HTML report, we also want the user to be able to conduct additional analyses outside of RSMTool.To this end, all of the tables produced in the experiment report are saved as files in the format as specified by `file_format` parameter in the `output` directory. The following sections describe all of the intermediate files that are produced.

Note: The names of all files begin with the `experiment_id` provided by the user in the experiment configuration file. In addition, the names for certain columns are set to default values in these files irrespective of what they were named in the original data files. This is because RSMTool standardizes these column names internally for convenience. These values are:

- `spkitemid` for the column containing response IDs.
- `sc1` for the column containing the human scores used as training labels.
- `sc2` for the column containing the second human scores, if this column was specified in the configuration file.
- `length` for the column containing response length, if this column was specified in the configuration file.
- `candidate` for the column containing candidate IDs, if this column was specified in the configuration file.

Feature values

filenames: `train_features`, `test_features`, `train_preprocessed_features`, `test_preprocessed_features`

These files contain the raw and pre-processed feature values for the training and evaluation sets. They include *only* includes the rows that were used for training/evaluating the models after filtering. For models with feature selection, these files *only* include the features that ended up being included in the model.

Note: By default RSMTool filters out non-numeric feature values and non-numeric/zero human scores from both the training and evaluation sets. Zero scores can be kept by setting the *exclude_zero_scores* to *false*.

Flagged responses

filenames: `train_responses_with_excluded_flags`, `test_responses_with_excluded_flags`

These files contain all of the rows in the training and evaluation sets that were filtered out based on conditions specified in *flag_column*.

Note: If the training/evaluation files contained columns with internal names such as `sc1` or `length` but these columns were not actually used by `rsmtool`, these columns will also be included into these files but their names will be changed to `##name##` (e.g. `##sc1##`).

Excluded responses

filenames: `train_excluded_responses`, `test_excluded_responses`

These files contain all of the rows in the training and evaluation sets that were filtered out because of feature values or scores. For models with feature selection, these files *only* include the features that ended up being included in the model.

Response metadata

filenames: `train_metadata`, `test_metadata`

These files contain the metadata columns (`id_column`, `subgroups` if provided) for the rows in the training and evaluation sets that were *not* excluded for some reason.

Unused columns

filenames: `train_other_columns`, `test_other_columns`

These files contain all of the the columns from the original features files that are not present in the `*_feature` and `*_metadata` files. They only include the rows from the training and evaluation sets that were not filtered out.

Note: If the training/evaluation files contained columns with internal names such as `sc1` or `length` but these columns were not actually used by `rsmtool`, these columns will also be included into these files but their names will be changed to `##name##` (e.g. `##sc1##`).

Response length

filename: `train_response_lengths`

If *length_column* is specified, then this file contains the values from that column for the training data under a column called `length` with the response IDs under the `spkitemid` column.

Human scores

filename: `test_human_scores`

This file contains the human scores for the evaluation data under a column called `sc1` with the response IDs under the `spkitemid` column. If `second_human_score_column` was specfied, then it also contains the values from that column under a column called `sc2`. Only the rows that were not filtered out are included.

Note: If `exclude_zero_scores` was set to `true` (the default value), all zero scores in the `second_human_score_column` will be replaced by `nan`.

Data composition

filename: `data_composition`

This file contains the total number of responses in training and evaluation set and the number of overlapping responses. If applicable, the table will also include the number of different subgroups for each set.

Excluded data composition

filenames: `train_excluded_composition`, `test_excluded_composition`

These files contain the composition of the set of excluded responses for the training and evaluation sets, e.g., why were they excluded and how many for each such exclusion.

Missing features

filename: `train_missing_feature_values`

This file contains the total number of non-numeric values for each feature. The counts in this table are based only on those responses that have a numeric human score in the training data.

Subgroup composition

filename: `data_composition_by_<SUBGROUP>`

There will be one such file for each of the specified subgroups and it contains the total number of responses in that subgroup in both the training and evaluation sets.

Feature descriptives

filenames: `feature_descriptives`, `feature_descriptivesExtra`

The first file contains the main descriptive statistics (mean,std. dev., correlation with human score etc.) for all features included in the final model. The second file contains percentiles, mild, and extreme outliers for the same set of features. The values in both files are computed on raw feature values before pre-processing.

Feature outliers

filename: `feature_outliers`

This file contains the number and percentage of outlier values truncated to [MEAN-4*SD, MEAN+4*SD] during feature pre-processing for each feature included in the final model.

Inter-feature and score correlations

filenames: `cors_orig`, `cors_processed`

The first file contains the pearson correlations between each pair of (raw) features and between each (raw) feature and the human score. The second file is the same but with the pre-processed feature values instead of the raw values.

Marginal and partial correlations with score

filenames: `margcor_score_all_data`, `pcor_score_all_data`, `` `pcor_score_no_length_all_data ``

The first file contains the marginal correlations between each pre-processed feature and human score. The second file contains the partial correlation between each pre-processed feature and human score after controlling for all other features. The third file contains the partial correlations between each pre-processed feature and human score after controlling for response length, if `length_column` was specified in the configuration file.

Marginal and partial correlations with length

filenames: `margcor_length_all_data`, `pcor_length_all_data`

The first file contains the marginal correlations between each pre-processed feature and response length, if `length_column` was specified. The second file contains the partial correlations between each pre-processed feature and response length after controlling for all other features, if `length_column` was specified in the configuration file.

Principal components analyses

filenames: `pca`, `pcavar`

The first file contains the the results of a Principal Components Analysis (PCA) using pre-processed feature values from the training set and its singular value decomposition. The second file contains the eigenvalues and variance explained by each component.

Various correlations by subgroups

Each of following files may be produced for every subgroup, assuming all other information was also available.

- `margcor_score_by_<SUBGROUP>`: the marginal correlations between each pre-processed feature and human score, computed separately for the subgroup.
- `pcor_score_by_<SUBGROUP>`: the partial correlations between pre-processed features and human score after controlling for all other features, computed separately for the subgroup.
- `pcor_score_no_length_by_<SUBGROUP>`: the partial correlations between each pre-processed feature and human score after controlling for response length (if available), computed separately for the subgroup.
- `margcor_length_by_<SUBGROUP>`: the marginal correlations between each feature and response length (if available), computed separately for each subgroup.
- `pcor_length_by_<SUBGROUP>`: partial correlations between each feature and response length (if available) after controlling for all other features, computed separately for each subgroup.

Note: All of the feature descriptive statistics, correlations (including those for subgroups), and PCA are computed *only* on the training set.

Model information

- `feature`: *pre-processing parameters* for all features used in the model.
- `coefficients`: model coefficients and intercept (for *built-in models* only).
- `coefficients_scaled`: scaled model coefficients and intercept (linear models only). Although RSMTool generates scaled scores by scaling the predictions of the model, it is also possible to achieve the same result by scaling the coefficients instead. This file shows those scaled coefficients.
- `betas`: standardized and relative coefficients (for built-in models only).
- `model_fit`: R squared and adjusted R squared computed on the training set. Note that these values are always computed on raw predictions without any trimming or rounding.
- `.model`: the serialized SKLL `Learner` object containing the fitted model (before scaling the coeffcients).
- `.ols`: a serialized object of type `pandas.stats.ols.OLS` containing the fitted model (for built-in models excluding `LassoFixedLambda` and `PositiveLassoCV`).
- `ols_summary.txt`: a text file containing a summary of the above model (for built-in models excluding `LassoFixedLabmda` and `PositiveLassoCV`)
- `postprocessing_params`: the parameters for trimming and scaling predicted scores. Useful for generating predictions on new data.

Predictions

filenames: `pred_processed`, `pred_train`

The first file contains the predicted scores for the evaluation set and the second file contains the predicted scores for the responses in the training set. Both of them contain the raw scores as well as different types of post-processed scores.

Evaluation metrics

- `eval`: This file contains the descriptives for predicted and human scores (mean, std.dev etc.) as well as the association metrics (correlation, quadartic weighted kappa, SMD etc.) for the raw as well as the post-processed scores.
- `eval_by_<SUBGROUP>`: the same information as in **_eval.csv* computed separately for each subgroup. However, rather than SMD, a difference of standardized means (DSM) will be calculated using z-scores.
- `eval_short` - a shortened version of `eval` that contains specific descriptives for predicted and human scores (mean, std.dev etc.) and association metrics (correlation, quadartic weighted kappa, SMD etc.) for specific score types chosen based on recommendations by Williamson (2012). Specifically, the following columns are included (the `raw` or `scale` version is chosen depending on the value of the `use_scaled_predictions` in the configuration file).
  - h_mean
  - h_sd
  - corr
  - sys_mean [raw/scale_trim]
  - sys_sd [raw/scale_trim]
  - SMD [raw/scale_trim]
  - adj_agr [raw/scale_trim_round]

  - exact_agr [raw/scale_trim_round]
  - kappa [raw/scale_trim_round]
  - wtkappa [raw/scale_trim]
  - sys_mean [raw/scale_trim_round]
  - sys_sd [raw/scale_trim_round]
  - SMD [raw/scale_trim_round]
  - R2 [raw/scale_trim]
  - RMSE [raw/scale_trim]

- `score_dist`: the distributions of the human scores and the rounded raw/scaled predicted scores, depending on the value of `use_scaled_predictions`.
- `confMatrix`: the confusion matrix between the the human scores and the rounded raw/scaled predicted scores, depending on the value of `use_scaled_predictions`.

Note: Please note that for raw scores, SMD values are likely to be affected by possible differences in scale.

- `true_score_eval` - evaluation of how well system scores can predict true scores.

Human-human Consistency

These files are created only if a second human score has been made available via the `second_human_score_column` option in the configuration file.

- `consistency`: contains descriptives for both human raters as well as the agreement metrics between their ratings.
- `consistency_by_<SUBGROUP>`: contains the same metrics as in `consistency` file computed separately for each group. However, rather than SMD, a difference of standardized means (DSM) will be calculated using z-scores.
- `degradation`: shows the differences between human-human agreement and machine-human agreement for all association metrics and all forms of predicted scores.

Evaluations based on test theory

- `disattenuated_correlations`: shows the correlation between human-machine scores, human-human scores, and the disattenuated human-machine correlation computed as human-machine correlation divided by the square root of human-human correlation.
- `disattenuated_correlations_by_<SUBGROUP>`: contains the same metrics as in `disattenuated_correlations` file computed separately for each group.
- `true_score_eval`: evaluations of system scores against estimated true score. Contains total counts of single and double-scored response, variance of human rater error, estimated true score variance, and mean squared error (MSE) and proportional reduction in mean squared error (PRMSE) when predicting true score using system score.

Additional fairness analyses

These files contain the results of additional fairness analyses suggested in suggested in Loukina, Madnani, & Zechner, 2019.

- `<METRICS>_by_<SUBGROUP>.ols`: a serialized object of type `pandas.stats.ols.OLS` containing the fitted model for estimating the variance attributed to a given subgroup membership for a given metric. The subgroups are defined by the *configuration file*. The metrics are `osa` (overall score accuracy), `osd` (overall score difference), and `csd` (conditional score difference).
- `<METRICS>_by_<SUBGROUP>_ols_summary.txt`: a text file containing a summary of the above model
- `estimates_<METRICS>_by_<SUBGROUP>`` : coefficients, confidence intervals and p-values estimated by the model for each subgroup.
- `fairness_metrics_by_<SUBGROUP>`: the R^2 (percentage of variance) and p-values for all models.

1.6.5 Built-in RSMTool Linear Regression Models

Models which use the full feature set

- `LinearRegression`: A model that learns empirical regression weights using ordinary least squares regression (OLS).
- `EqualWeightsLR`: A model with all feature weights set to 1.0; a naive model.
- `ScoreWeightedLR`: a model that learns empirical regression weights using weighted least sqaures. The weights are determined based on the number of responses with different score levels. Score levels with lower number of responses are assigned higher weight.
- `RebalancedLR` - empirical regression weights are rebalanced by using a small portion of positive weights to replace negative beta values. This model has no negative coefficients.

Models with automatic feature selection

- `LassoFixedLambdaThenLR`: A model that learns empirical OLS regression weights with feature selection using Lasso regression with all coefficients set to positive. The hyperparameter `lambda` is set to `sqrt(n-lg(p))` where `n` is the number of responses and `p` is the number of features. This approach was chosen to balance the penalties for error vs. penalty for two many coefficients to force Lasso perform more aggressive feature selection, so it may not necessarily achieve the best possible performance. The feature set selected by LASSO is then used to fit an OLS linear regression. Note that while the original Lasso model is constrained to positive coefficients only, small negative coefficients may appear when the coefficients are re-estimated using OLS regression.
- `PositiveLassoCVThenLR`: A model that learns empirical OLS regression weights with feature selection using Lasso regression with all coefficients set to positive. The hyperparameter `lambda` is optimized using crossvalidation for loglikehood. The feature set selected by LASSO is then used to fit an OLS linear regression. Note that this approach will likely produce a model with a large N features and any advantages of running Lasso would be effectively negated by latter adding those features to OLS regression.
- `NNLR`: A model that learns empirical OLS regression weights with feature selection using non-negative least squares regression. Note that only the coefficients are constrained to be positive: the intercept can be either positive or negative.
- `NNLRIterative`: A model that learns empirical OLS regression weights with feature selection using an iterative implementation of non-negative least squares regression. Under this implementation, an initial OLS

model is fit. Then, any variables whose coefficients are negative are dropped and the model is re-fit. Any coefficients that are still negative after re-fitting are set to zero.

- `LassoFixedLambdaThenNNLR`: A model that learns empirical OLS regression weights with feature selection using Lasso regression as above followed by non-negative least squares regression. The latter ensures that no feature has negative coefficients even when the coefficients are estimated using least squares without penalization.
- `LassoFixedLambda`: same as `LassoFixedLambdaThenLR` but the model uses the original Lasso weights. Note that the coefficients in Lasso model are estimated using an optimization routine which may produce slightly different results on different systems.
- `PositiveLassoCV`: same as *PositiveLassoCVThenLR* but using the original Lasso weights. Please note: the coefficients in Lasso model are estimated using an optimization routine which may produce slightly different results on different systems.

Note:

1. `NNLR`, `NNLRIterative`, `LassoFixedLambdaThenNNLR`, `LassoFixedLambda` and `PositiveLassoCV` all have no negative coefficients.
2. For all feature selection models, the final set of features will be saved in the `feature` folder in the experiment output directory.

1.7 Advanced Uses of RSMTool

In addition to providing the `rsmtool` utility training and evaluating regression-based scoring models, the RSMTool package also provides three other command-line utilities for more advanced users.

1.7.1 `rsmeval` - Evaluate external predictions

RSMTool provides the `rsmeval` command-line utility to evaluate existing predictions and generate a report with all the built-in analyses. This can be useful in scenarios where the user wants to use more sophisticated machine learning algorithms not available in RSMTool to build the scoring model but still wants to be able to evaluate that model's predictions using the standard analyses.

For example, say a researcher *has* an existing automated scoring engine for grading short responses that extracts the features and computes the predicted score. This engine uses a large number of binary, sparse features. She cannot use `rsmtool` to train her model since it requires numeric features. So, she uses scikit-learn to train her model.

Once the model is trained, the researcher wants to evaluate her engine's performance using the analyses recommended by the educational measurement community as well as conduct additional investigations for specific subgroups of test-takers. However, these kinds of analyses are not available in `scikit-learn`. She can use `rsmeval` to set up a customized report using a combination of existing and custom sections and quickly produce the evaluation that is useful to her.

Tutorial

For this tutorial, you first need to *install RSMTool* and make sure the conda environment in which you installed it is activated.

Workflow

`rsmeval` is designed for evaluating existing machine scores. Once you have the scores computed for all the responses in your data, the next steps are fairly straightforward:

1. Create a data file in one of the *supported formats* containing the computed system scores and the human scores you want to compare against.
2. Create an *experiment configuration file* describing the evaluation experiment you would like to run.
3. Run that configuration file with *rsmeval* and generate the experiment HTML report as well as the *intermediate CSV files*.
4. Examine the HTML report to check various aspects of model performance.

Note that the above workflow does not use any customization features , e.g., *choosing which sections to include in the report* or *adding custom analyses sections* etc. However, we will stick with this workflow for our tutorial since it is likely to be the most common use case.

ASAP Example

We are going to use the same example from 2012 Kaggle competition on automated essay scoring that we used for the *rsmtool tutorial*.

Generate scores

`rsmeval` is designed for researchers who have developed their own scoring engine for generating scores and would like to produce an evaluation report for those scores. For this tutorial, we will use the scores we generated for the ASAP2 evaluation set using *rsmtool tutorial*.

Create a configuration file

The next step is to create an *experiment configuration file* in `.json` format.

```
1  {
2      "experiment_id": "ASAP2_evaluation",
3      "description": "Evaluation of the scores generated using rsmtool.",
4      "predictions_file": "ASAP2_scores.csv",
5      "system_score_column": "system",
6      "human_score_column": "human",
7      "id_column": "ID",
8      "trim_min": 1,
9      "trim_max": 6,
10     "second_human_score_column": "human2",
11     "scale_with": "asis"
12 }
```

Let's take a look at the options in our configuration file.

- **Line 2**: We define an experiment ID.
- **Line 3**: We also provide a description which will be included in the experiment report.
- **Line 4**: We list the path to the file with the predicted and human scores. For this tutorial we used `.csv` format, but RSMTool also supports several other *input file formats*.

- **Line 5**: This field indicates that the system scores in our `.csv` file are located in a column named `system`.
- **Line 6**: This field indicates that the human (reference) scores in our `.csv` file are located in a column named `human`.
- **Line 7**: This field indicates that the unique IDs for the responses in the `.csv` file are located in columns named `ID`.
- **Lines 8-9**: These fields indicate that the lowest score on the scoring scale is a 1 and the highest score is a 6. This information is usually part of the rubric used by human graders.
- **Line 10**: This field indicates that scores from a second set of human graders are also available (useful for comparing the agreement between human-machine scores to the agreement between two sets of humans) and are located in the `human2` column in the `.csv` file.
- **Line 11**: This field indicates that the provided machine scores are already re-scaled to match the distribution of human scores. `rsmeval` itself will not perform any scaling and the report will refer to these as `scaled` scores.

Documentation for all of the available configuration options is available *here*.

Note: You can also use our nifty capability to *automatically generate* `rsmeval` configuration files rather than creating them manually.

Run the experiment

Now that we have our scores in the right format and our configuration file in `.json` format, we can use the *rsmeval* command-line script to run our evaluation experiment.

```
$ cd examples/rsmeval
$ rsmeval config_rsmeval.json
```

This should produce output like:

```
Output directory: /Users/nmadnani/work/rsmtool/examples/rsmeval
Assuming given system predictions are already scaled and will be used as such.
 predictions: /Users/nmadnani/work/rsmtool/examples/rsmeval/ASAP2_scores.csv
Processing predictions
Saving pre-processed predictions and the metadata to disk
Running analyses on predictions
Starting report generation
Merging sections
Exporting HTML
Executing notebook with kernel: python3
```

Once the run finishes, you will see the `output`, `figure`, and `report` sub-directories in the current directory. Each of these directories contain *useful information* but we are specifically interested in the `report/ASAP2_evaluation_report.html` file, which is the final evaluation report.

Examine the report

Our experiment report contains all the information we would need to evaluate the provided system scores against the human scores. It includes:

1. The distributions for the human versus the system scores.

2. Several different metrics indicating how well the machine's scores agree with the humans'.
3. Information about human-human agreement and the difference between human-human and human-system agreement.

... and *much more*.

Input

`rsmeval` requires a single argument to run an experiment: the path to *a configuration file*. It can also take an output directory as an optional second argument. If the latter is not specified, `rsmeval` will use the current directory as the output directory.

Here are all the arguments to the `rsmeval` command-line script.

`config_file`
The *JSON configuration file* for this experiment.

`output_dir` `(optional)`
The output directory where all the files for this experiment will be stored.

`-f, --force`
If specified, the contents of the output directory will be overwritten even if it already contains the output of another rsmeval experiment.

`-h, --help`
Show help message and exist.

`-V, --version`
Show version number and exit.

Experiment configuration file

This is a file in `.json` format that provides overall configuration options for an `rsmeval` experiment. Here's an example configuration file for `rsmeval`.

Note: To make it easy to get started with `rsmeval`, we provide a way to **automatically generate** configurations file both interactively as well as non-interactively. Novice users will find interactive generation more helpful while more advanced users will prefer non-interactive generation. See *this page* for more details.

Next, we describe all of the `rsmeval` configuration fields in detail. There are four required fields and the rest are all optional. We first describe the required fields and then the optional ones (sorted alphabetically).

experiment_id

An identifier for the experiment that will be used to name the report and all *intermediate files*. It can be any combination of alphanumeric values, must *not* contain spaces, and must *not* be any longer than 200 characters.

predictions_file

The path to the file with predictions to evaluate. The file should be in one of the *supported formats*. Each row should correspond to a single response and contain the predicted and observed scores for this response. In addition, there should be a column with a unique identifier (ID) for each response. The path can be absolute or relative to the location of the configuration file.

system_score_column

The name for the column containing the scores predicted by the system. These scores will be used for evaluation.

trim_min

The single numeric value for the lowest possible integer score that the machine should predict. This value will be used to compute the floor value for *trimmed (bound)* machine scores as `trim_min - trim_tolerance`.

trim_max

The single numeric value for the highest possible integer score that the machine should predict. This value will be used to compute the ceiling value for *trimmed (bound)* machine scores as `trim_max + trim_tolerance`.

Note: Although the `trim_min` and `trim_max` fields are optional for `rsmtool`, they are *required* for `rsmeval`.

---

candidate_column *(Optional)*

The name for an optional column in prediction file containing unique candidate IDs. Candidate IDs are different from response IDs since the same candidate (test-taker) might have responded to multiple questions.

custom_sections *(Optional)*

A list of custom, user-defined sections to be included into the final report. These are IPython notebooks (`.ipynb` files) created by the user. The list must contains paths to the notebook files, either absolute or relative to the configuration file. All custom notebooks have access to some *pre-defined variables*.

description *(Optional)*

A brief description of the experiment. This will be included in the report. The description can contain spaces and punctuation. It's blank by default.

exclude_zero_scores *(Optional)*

By default, responses with human scores of 0 will be excluded from evaluations. Set this field to `false` if you want to keep responses with scores of 0. Defaults to `true`.

file_format *(Optional)*

The format of the *intermediate files*. Options are `csv`, `tsv`, or `xlsx`. Defaults to `csv` if this is not specified.

flag_column *(Optional)*

This field makes it possible to only use responses with particular values in a given column (e.g. only responses with a value of `0` in a column called `ADVISORY`). The field takes a dictionary in Python format where the keys are the names of the columns and the values are lists of values for responses that will be evaluated. For example, a value of `{"ADVISORY": 0}` will mean that `rsmeval` will *only* use responses for which the `ADVISORY` column has the value 0. Defaults to `None`.

Note: If several conditions are specified (e.g., `{"ADVISORY": 0, "ERROR": 0}`) only those responses which satisfy *all* the conditions will be selected for further analysis (in this example, these will be the responses where the `ADVISORY` column has a value of 0 *and* the `ERROR` column has a value of 0).

Note: When reading the values in the supplied dictionary, `rsmeval` treats numeric strings, floats and integers as the same value. Thus `1`, `1.0`, `"1"` and `"1.0"` are all treated as the `1.0`.

---

general_sections *(Optional)*

RSMTool provides pre-defined sections for `rsmeval` (listed below) and, by default, all of them are included in the report. However, you can choose a subset of these pre-defined sections by specifying a list as the value for this field.

- `data_description`: Shows the total number of responses, along with any responses have been excluded due to non-numeric/zero scores or *flag columns*.
- `data_description_by_group`: Shows the total number of responses for each of the *subgroups* specified in the configuration file. This section only covers the responses used to evaluate the model.
- `consistency`: shows metrics for *human-human agreement*, the difference ("degradation") between the human-human and human-system agreement, and the disattenuated human-machine correlations. This notebook is only generated if the config file specifies *second_human_score_column*.
- `evaluation`: Shows the *standard set of evaluations* recommended for scoring models on the evaluation data:
  - a table showing human-system association metrics;
  - the confusion matrix; and
  - a barplot showing the distributions for both human and machine scores.
- `evaluation by group`: Shows barplots with the main evaluation metrics by each of the subgroups specified in the configuration file.
- `fairness_analyses`: Additional *fairness analyses* suggested in Loukina, Madnani, & Zechner, 2019. The notebook shows:
  - percentage of variance in squared error explained by subgroup membership
  - percentage of variance in raw (signed) error explained by subgroup membership
  - percentage of variance in raw (signed) error explained by subgroup membership when controlling for human score
  - plots showing estimates for each subgroup for each model
- `true_score_evaluation`: evaluation of system scores against the true scores estimated according to test theory. The notebook shows:
  - Number of single and double-scored responses.

  - Variance of human rater errors and estimated variance of true scores
  - Mean squared error (MSE) and proportional reduction in mean squared error (PRMSE) when predicting true score with system score.
- `intermediate_file_paths`: Shows links to all of the intermediate files that were generated while running the evaluation.
- `sysinfo`: Shows all Python packages along with versions installed in the current environment while generating the report.

human_score_column *(Optional)*

The name for the column containing the human scores for each response. The values in this column will be used as observed scores. Defaults to `sc1`.

Note: All responses with non-numeric values or zeros in either `human_score_column` or `system_score_column` will be automatically excluded from evaluation. You can use *exclude_zero_scores (Optional)* to keep responses with zero scores.

---

id_column *(Optional)*

The name of the column containing the response IDs. Defaults to `spkitemid`, i.e., if this is not specified, `rsmeval` will look for a column called `spkitemid` in the prediction file.

min_items_per_candidate *(Optional)*

An integer value for the minimum number of responses expected from each candidate. If any candidates have fewer responses than the specified value, all responses from those candidates will be excluded from further analysis. Defaults to `None`.

min_n_per_group *(Optional)*

A single numeric value or a dictionary with keys as the group names listed in the *subgroups* field and values as the thresholds for the groups. When specified, only groups with *at least* this number of instances will be displayed in the tables and plots contained **in the report**. Note that this parameter *only* affects the HTML report and the figures. For all analyses – including the computation of the population parameters – data from *all* groups will be used. In addition, the *intermediate files* will still show the results for *all* groups.

Note: If you supply a dictionary, it *must* contain a key for *every* subgroup listed in *subgroups* field. If no threshold is to be applied for some of the groups, set the threshold value for this group to 0 in the dictionary.

---

rater_error_variance *(Optional)*

True score evaluations require an estimate of rater error variance. By default, `rsmeval` will compute this variance from double-scored responses in the data. However, in some cases, one may wish to compute the variance on a different sample of responses. In such cases, this field can be used to set the rater error variance to a precomputed

value which is then used as-is by `rsmeval`. You can use the *rsmtool.utils.variance_of_errors* function to compute rater error variance outside the main evaluation pipeline.

scale_with *(Optional)*

In many scoring applications, system scores are *re-scaled* so that their mean and standard deviation match those of the human scores for the training data.

If you want `rsmeval` to re-scale the supplied predictions, you need to provide – as the value for this field – the path to a second file in one of the *supported formats* containing the human scores and predictions of the same system on its training data. This file *must* have two columns: the human scores under the `sc1` column and the predicted score under the `prediction`.

This field can also be set to `"asis"` if the scores are already scaled. In this case, no additional scaling will be performed by `rsmeval` but the report will refer to the scores as "scaled".

Defaults to `"raw"` which means that no-rescaling is performed and the report refers to the scores as "raw".

second_human_score_column *(Optional)*

The name for an optional column in the test data containing a second human score for each response. If specified, additional information about human-human agreement and degradation will be computed and included in the report. Note that this column must contain either numbers or be empty. Non-numeric values are *not* accepted. Note also that the *exclude_zero_scores (Optional)* option below will apply to this column too.

Note: You do not need to have second human scores for *all* responses to use this option. The human-human agreement statistics will be computed as long as there is at least one response with numeric value in this column. For responses that do not have a second human score, the value in this column should be blank.

---

section_order *(Optional)*

A list containing the order in which the sections in the report should be generated. Any specified order must explicitly list:

1. Either *all* pre-defined sections if a value for the *general_sections* field is not specified OR the sections specified using *general_sections*, and
2. *All* custom section names specified using *custom_ sections*, i.e., file prefixes only, without the path and without the *.ipynb* extension, and
3. *All* special sections specified using *special_sections*.

special_sections *(Optional)*

A list specifying special ETS-only sections to be included into the final report. These sections are available *only* to ETS employees via the `rsmextra` package.

subgroups *(Optional)*

A list of column names indicating grouping variables used for generating analyses specific to each of those defined subgroups. For example, `["prompt, gender, native_language, test_country"]`. These subgroup columns need to be present in the input predictions file. If subgroups are specified, `rsmeval` will generate:

- tables and barplots showing human-system agreement for each subgroup on the evaluation set.

trim_tolerance *(Optional)*

The single numeric value that will be used to pad the trimming range specified in `trim_min` and `trim_max`. This value will be used to compute the ceiling and floor values for *trimmed (bound)* machine scores as `trim_max` + `trim_tolerance` for ceiling value and `trim_min`-`trim_tolerance` for floor value. Defaults to 0.4998.

Note: For more fine-grained control over the trimming range, you can set `trim_tolerance` to *0* and use `trim_min` and `trim_max` to specify the exact floor and ceiling values.

---

use_thumbnails *(Optional)*

If set to `true`, the images in the HTML will be set to clickable thumbnails rather than full-sized images. Upon clicking the thumbnail, the full-sized images will be displayed in a separate tab in the browser. If set to `false`, full-sized images will be displayed as usual. Defaults to `false`.

Output

`rsmeval` produces a set of folders in the output directory.

report

This folder contains the final RSMEval report in HTML format as well as in the form of a Jupyter notebook (a `.ipynb` file).

output

This folder contains all of the *intermediate files* produced as part of the various analyses performed, saved as `.csv` files. `rsmeval` will also save in this folder a copy of the *configuration file*. Fields not specified in the original configuration file will be pre-populated with default values.

figure

This folder contains all of the figures generated as part of the various analyses performed, saved as `.svg` files.

Intermediate files

Although the primary output of `rsmeval` is an HTML report, we also want the user to be able to conduct additional analyses outside of `rsmeval`. To this end, all of the tables produced in the experiment report are saved as files in the format as specified by `file_format` parameter in the `output` directory. The following sections describe all of the intermediate files that are produced.

Note: The names of all files begin with the `experiment_id` provided by the user in the experiment configuration file. In addition, the names for certain columns are set to default values in these files irrespective of what they were named in the original data files. This is because RSMEval standardizes these column names internally for convenience. These values are:

- `spkitemid` for the column containing response IDs.
- `sc1` for the column containing the human scores used as observed scores
- `sc2` for the column containing the second human scores, if this column was specified in the configuration file.
- `candidate` for the column containing candidate IDs, if this column was specified in the configuration file.

Predictions

filename: `pred_processed`

This file contains the post-processed predicted scores: the predictions from the model are truncated, rounded, and re-scaled (if requested).

Flagged responses

filename: `test_responses_with_excluded_flags`

This file contains all of the rows in input predictions file that were filtered out based on conditions specified in *flag_column*.

Note: If the predictions file contained columns with internal names such as `sc1` that were not actually used by `rsmeval`, they will still be included in these files but their names will be changed to `##name##` (e.g. `##sc1##`).

Excluded responses

filename: `test_excluded_responses`

This file contains all of the rows in the predictions file that were filtered out because of non-numeric or zero scores.

Response metadata

filename: `test_metadata`

This file contains the metadata columns (`id_column`, `subgroups` if provided) for all rows in the predictions file that used in the evaluation.

Unused columns

filename: `test_other_columns`

This file contains all of the the columns from the input predictions file that are not present in the `*_pred_processed` and `*_metadata` files. They only include the rows that were not filtered out.

Note: If the predictions file contained columns with internal names such as `sc1` but these columns were not actually used by `rsmeval`, these columns will also be included into these files but their names will be changed to `##name##` (e.g. `##sc1##`).

Human scores

filename: `test_human_scores`

This file contains the human scores, if available in the input predictions file, under a column called `sc1` with the response IDs under the `spkitemid` column.

If `second_human_score_column` was specfied, then it also contains the values in the predictions file from that column under a column called `sc2`. Only the rows that were not filtered out are included.

Note: If `exclude_zero_scores` was set to `true` (the default value), all zero scores in the `second_human_score_column` will be replaced by `nan`.

Data composition

filename: `data_composition`

This file contains the total number of responses in the input predictions file. If applicable, the table will also include the number of different subgroups.

Excluded data composition

filenames: `test_excluded_composition`

This file contains the composition of the set of excluded responses, e.g., why were they excluded and how many for each such exclusion.

Subgroup composition

filename: `data_composition_by_<SUBGROUP>`

There will be one such file for each of the specified subgroups and it contains the total number of responses in that subgroup.

Evaluation metrics

- `eval`: This file contains the descriptives for predicted and human scores (mean, std.dev etc.) as well as the association metrics (correlation, quadartic weighted kappa, SMD etc.) for the raw as well as the post-processed scores.
- `eval_by_<SUBGROUP>`: the same information as in **_eval.csv* computed separately for each subgroup. However, rather than SMD, a difference of standardized means (DSM) will be calculated using z-scores.
- `eval_short` - a shortened version of `eval` that contains specific descriptives for predicted and human scores (mean, std.dev etc.) and association metrics (correlation, quadartic weighted kappa, SMD etc.) for specific score types chosen based on recommendations by Williamson (2012). Specifically, the following columns are included (the `raw` or `scale` version is chosen depending on the value of the `use_scaled_predictions` in the configuration file).
  - h_mean
  - h_sd
  - corr
  - sys_mean [raw/scale trim]
  - sys_sd [raw/scale trim]
  - SMD [raw/scale trim]
  - adj_agr [raw/scale trim_round]
  - exact_agr [raw/scale trim_round]
  - kappa [raw/scale trim_round]
  - wtkappa [raw/scale trim]
  - sys_mean [raw/scale trim_round]
  - sys_sd [raw/scale trim_round]
  - SMD [raw/scale trim_round]
  - R2 [raw/scale trim]
  - RMSE [raw/scale trim]
- `score_dist`: the distributions of the human scores and the rounded raw/scaled predicted scores, depending on the value of `use_scaled_predictions`.
- `confMatrix`: the confusion matrix between the the human scores and the rounded raw/scaled predicted scores, depending on the value of `use_scaled_predictions`.

Note: Please note that for raw scores, SMD values are likely to be affected by possible differences in scale.

Human-human Consistency

These files are created only if a second human score has been made available via the `second_human_score_column` option in the configuration file.

- `consistency`: contains descriptives for both human raters as well as the agreement metrics between their ratings.

- `consistency_by_<SUBGROUP>`: contains the same metrics as in `consistency` file computed separately for each group. However, rather than SMD, a difference of standardized means (DSM) will be calculated using z-scores.
- `degradation`: shows the differences between human-human agreement and machine-human agreement for all association metrics and all forms of predicted scores.

Evaluations based on test theory

- `disattenuated_correlations`: shows the correlation between human-machine scores, human-human scores, and the disattenuated human-machine correlation computed as human-machine correlation divided by the square root of human-human correlation.
- `disattenuated_correlations_by_<SUBGROUP>`: contains the same metrics as in `disattenuated_correlations` file computed separately for each group.
- `true_score_eval`: evaluations of system scores against estimated true score. Contains total counts of single and double-scored response, variance of human rater error, estimated true score variance, and mean squared error (MSE) and proportional reduction in mean squared error (PRMSE) when predicting true score using system score.

Additional fairness analyses

These files contain the results of additional fairness analyses suggested in suggested in Loukina, Madnani, & Zechner, 2019.

- `<METRICS>_by_<SUBGROUP>.ols`: a serialized object of type `pandas.stats.ols.OLS` containing the fitted model for estimating the variance attributed to a given subgroup membership for a given metric. The subgroups are defined by the *configuration file*. The metrics are `osa` (overall score accuracy), `osd` (overall score difference), and `csd` (conditional score difference).
- `<METRICS>_by_<SUBGROUP>_ols_summary.txt`: a text file containing a summary of the above model
- `estimates_<METRICS>_by_<SUBGROUP>`` : coefficients, confidence intervals and p-values estimated by the model for each subgroup.
- `fairness_metrics_by_<SUBGROUP>`: the R^2 (percentage of variance) and p-values for all models.

1.7.2 `rsmpredict` - Generate new predictions

RSMTool provides the `rsmpredict` command-line utility to generate predictions for new data using a model already trained using the `rsmtool` utility. This can be useful when processing a new set of responses to the same task without needing to retrain the model.

`rsmpredict` pre-processes the feature values according to user specifications before using them to generate the predicted scores. The generated scores are post-processed in the same manner as they are in `rsmtool` output.

Note: No score is generated for responses with non-numeric values for any of the features included into the model.

If the original model specified transformations for some of the features and these transformations led to `NaN` or `Inf` values when applied to the new data, `rsmpredict` will raise a warning. No score will be generated for such responses.

Tutorial

For this tutorial, you first need to *install RSMTool* and make sure the conda environment in which you installed it is activated.

Workflow

Important: Although this tutorial provides feature values for the purpose of illustration, `rsmpredict` does *not* include any functionality for feature extraction; the tool is *designed for researchers* who use their own NLP/Speech processing pipeline to extract features for their data.

`rsmpredict` allows you to generate the scores for new data using an existing model trained using RSMTool. Therefore, before starting this tutorial, you first need to complete *rsmtool tutorial* which will produce a train RSMTool model. You will also need to process the new data to extract the same features as the ones used in the model.

Once you have the features for the new data and the RSMTool model, using `rsmpredict` is fairly straightforward:

1. Create a file containing the features for the new data. The file should be in one of the *supported formats*.
2. Create an *experiment configuration file* describing the experiment you would like to run.
3. Run that configuration file with *rsmpredict* to generate the predicted scores.

Note: You do not *need* human scores to run `rsmpredict` since it does not produce any evaluation analyses. If you do have human scores for the new data and you would like to evaluate the system on this new data, you can first run `rsmpredict` to generate the predictions and then run `rsmeval` on the output of `rsmpredict` to generate an evaluation report.

ASAP Example

We are going to use the same example from 2012 Kaggle competition on automated essay scoring that we used for the *rsmtool tutorial*. Specifically, We are going to use the linear regression model we trained in that tutorial to generate scores for new data.

Note: If you have not already completed that tutorial, please do so now. You may need to complete it again if you deleted the output files.

Extract features

We will first need to generate features for the new set of responses for which we want to predict scores. For this experiment, we will simply re-use the test set from the `rsmtool` tutorial.

Note: The features used with `rsmpredict` should be generated using the *same* NLP/Speech processing pipeline that generated the features used in the `rsmtool` modeling experiment.

Create a configuration file

The next step is to create an *rsmpredict experiment configuration file* in `.json` format.

```
{
    "experiment_dir": "../rsmtool",
    "experiment_id": "ASAP2",
    "input_features_file": "../rsmtool/test.csv",
    "id_column": "ID",
    "human_score_column": "score",
    "second_human_score_column": "score2"
}
```

Let's take a look at the options in our configuration file.

- **Line 2**: We give the path to the directory containing the output of the `rsmtool` experiment.
- **Line 3**: We provide the `experiment_id` of the `rsmtool` experiment used to train the model. This can usually be read off the `output/<experiment_id>.model` file in the `rsmtool` experiment output directory.
- **Line 4**: We list the path to the data file with the feature values for the new data. For this tutorial we used `.csv` format, but RSMTool also supports several other *input file formats*.
- **Line 5**: This field indicates that the unique IDs for the responses in the `.csv` file are located in a column named `ID`.
- **Lines 6-7**: These fields indicates that there are two sets of human scores in our `.csv` file located in the columns named `score` and `score2`. The values from these columns will be added to the output file containing the predictions which can be useful if we want to evaluate the predictions using `rsmeval`.

Documentation for all of the available configuration options is available *here*.

Note: You can also use our nifty capability to *automatically generate* `rsmpredict` configuration files rather than creating them manually.

Run the experiment

Now that we have the model, the features in the right format, and our configuration file in `.json` format, we can use the *rsmpredict* command-line script to generate the predictions and to save them in `predictions.csv`.

```
$ cd examples/rsmpredict
$ rsmpredict config_rsmpredict.json predictions.csv
```

This should produce output like:

```
WARNING: The following extraneous features will be ignored: {'spkitemid', 'sc1', 'sc2
↪', 'LENGTH'}
Pre-processing input features
Generating predictions
Rescaling predictions
Trimming and rounding predictions
Saving predictions to /Users/nmadnani/work/rsmtool/examples/rsmpredict/predictions.csv
```

You should now see a file named `predictions.csv` in the current directory which contains the predicted scores for the new data in the `predictions` column.

Input

`rsmpredict` requires two arguments to generate predictions: the path to *a configuration file* and the path to the output file where the generated predictions are saved in `.csv` format.

If you also want to save the pre-processed feature values,‘‘rsmpredict‘‘ can take a third optional argument `--features` to specify the path to a `.csv` file to save these values.

Here are all the arguments to the `rsmpredict` command-line script.

config_file
The *JSON configuration file* for this experiment.

output_file
The output `.csv` file where predictions will be saved.

--features <preproc_feats_file>
If specified, the pre-processed values for the input features will also be saved in this `.csv` file.

-h, --help
Show help message and exist.

-V, --version
Show version number and exit.

Experiment configuration file

This is a file in `.json` format that provides overall configuration options for an `rsmpredict` experiment. Here's an example configuration file for `rsmpredict`.

Note: To make it easy to get started with `rsmpredict`, we provide a way to **automatically generate** configurations file both interactively as well as non-interactively. Novice users will find interactive generation more helpful while more advanced users will prefer non-interactive generation. See *this page* for more details.

Next, we describe all of the `rsmpredict` configuration fields in detail. There are three required fields and the rest are all optional. We first describe the required fields and then the optional ones (sorted alphabetically).

experiment_dir

The path to the directory containing `rsmtool` model to use for generating predictions. This directory must contain a sub-directory called `output` with the model files, feature pre-processing parameters, and score post-processing parameters. The path can be absolute or relative to the location of configuration file.

experiment_id

The `experiment_id` used to create the `rsmtool` model files being used for generating predictions. If you do not know the `experiment_id`, you can find it by looking at the prefix of the `.model` file under the `output` directory.

input_feature_file

The path to the file with the raw feature values that will be used for generating predictions. The file should be in one of the *supported formats* Each row should correspond to a single response and contain feature values for this response. In addition, there should be a column with a unique identifier (ID) for each response. The path can be absolute or

relative to the location of config file. Note that the feature names *must* be the same as used in the original `rsmtool` experiment.

Note: `rsmpredict` will only generate predictions for responses in this file that have numeric values for the features included in the `rsmtool` model.

See also:

`rsmpredict` does not require human scores for the new data since it does not evaluate the generated predictions. If you do have the human scores and want to evaluate the new predictions, you can use the *rsmeval* command-line utility.

candidate_column *(Optional)*

The name for the column containing unique candidate IDs. This column will be named `candidate` in the output file with predictions.

file_format *(Optional)*

The format of the *intermediate files*. Options are `csv`, `tsv`, or `xlsx`. Defaults to `csv` if this is not specified.

flag_column *(Optional)*

See description in the *rsmtool configuration file* for further information. No filtering will be done by `rsmpredict`, but the contents of all specified columns will be added to the predictions file using the original column names.

human_score_column *(Optional)*

The name for the column containing human scores. This column will be renamed to `sc1`.

id_column *(Optional)*

The name of the column containing the response IDs. Defaults to `spkitemid`, i.e., if this is not specified, `rsmpredict` will look for a column called `spkitemid` in the prediction file.

There are several other options in the configuration file that, while not directly used by `rsmpredict`, can simply be passed through from the input features file to the output predictions file. This can be particularly useful if you want to subsequently run *rsmeval* to evaluate the generated predictions.

predict_expected_scores *(Optional)*

If the original model was a probabilistic SKLL classifier, then *expected scores* — probability-weighted averages over a contiguous, numeric score points — can be generated as the machine predictions instead of the most likely score point, which would be the default. Set this field to `true` to compute expected scores as predictions. Defaults to `false`.

Note:

1. If the model in the original `rsmtool` experiment is an SVC, that original experiment *must* have been run with `predict_expected_scores` set to `true`. This is because SVC classifiers are fit differently if probabilistic output is desired, in contrast to other probabilistic SKLL classifiers.
2. You may see slight differences in expected score predictions if you run the experiment on different machines or on different operating systems most likely due to very small probability values for certain score points which can affect floating point computations.

---

second_human_score_column *(Optional)*

The name for the column containing the second human score. This column will be renamed to `sc2`.

standardize_features *(Optional)*

If this option is set to `false` features will not be standardized by dividing by the mean and multiplying by the standard deviation. Defaults to `true`.

subgroups *(Optional)*

A list of column names indicating grouping variables used for generating analyses specific to each of those defined subgroups. For example, `["prompt, gender, native_language, test_country"]`. All these columns will be included into the predictions file with the original names.

Output

`rsmpredict` produces a `.csv` file with predictions for all responses in new data set, and, optionally, a `.csv` file with pre-processed feature values. If any of the responses had non-numeric feature values in the original data or after applying transformations, these are saved in a file name `PREDICTIONS_NAME_excluded_responses.csv` where `PREDICTIONS_NAME` is the name of the predictions file supplied by the user without the extension.

The predictions `.csv` file contains the following columns:

- `spkitemid` : the unique resonse IDs from the original feature file.
- `sc1` and `sc2` : the human scores for each response from the original feature file (`human_score_column` and `second_human_score_column`, respectively.
- `raw` : raw predictions generated by the model.
- `raw_trim`, `raw_trim_round`, `scale`, `scale_trim`, `scale_trim_round` : raw scores *post-processed* in different ways.

1.7.3 `rsmcompare` - Create a detailed comparison of two scoring models

RSMTool provides the `rsmcompare` command-line utility to compare two models and to generate a detailed comparison report including differences between the two models. This can be useful in many scenarios, e.g., say the user wants to compare the changes in model performance after adding a new feature into the model. To use `rsmcompare`, the user must first run two experiments using either *rsmtool* or *rsmeval*. `rsmcompare` can then be used to compare the outputs of these two experiments to each other.

Note: Currently `rsmcompare` takes the outputs of the analyses generated during the original experiments and creates comparison tables. These comparison tables were designed with a specific comparison scenario in mind: comparing a baseline model with a model which includes new feature(s). The tool can certianly be used for other comparison scenarios if the researcher feels that the generated comparison output is appropriate.

`rsmcompare` can be used to compare:

1. Two `rsmtool` experiments, or
2. Two `rsmeval` experiments, or
3. An `rsmtool` experiment with an `rsmeval` experiment (in this case, only the evaluation analyses will be compared).

Note: It is strongly recommend that the original experiments as well as the comparison experiment are all done using the same version of RSMTool.

Tutorial

For this tutorial, you first need to *install RSMTool* and make sure the conda environment in which you installed it is activated.

Workflow

`rsmcompare` is designed to compare two existing `rsmtool` or `rsmeval` experiments. To use `rsmcompare` you need:

1. Two experiments that were run using *rsmtool* or *rsmeval*.
2. Create an *experiment configuration file* describing the comparison experiment you would like to run.
3. Run that configuration file with *rsmcompare* and generate the comparison experiment HTML report.
4. Examine HTML report to compare the two models.

Note that the above workflow does not use the customization features of `rsmcompare`, e.g., *choosing which sections to include in the report* or *adding custom analyses sections* etc. However, we will stick with this workflow for our tutorial since it is likely to be the most common use case.

ASAP Example

We are going to use the same example from 2012 Kaggle competition on automated essay scoring that we used for the *rsmtool tutorial*.

Run `rsmtool` (or `rsmeval`) experiments

`rsmcompare` compares the results of the two existing `rsmtool` (or `rsmeval`) experiments. For this tutorial, we will compare model trained in the *rsmtool tutorial* to itself.

Note: If you have not already completed that tutorial, please do so now. You may need to complete it again if you deleted the output files.

Create a configuration file

The next step is to create an *experiment configuration file* in `.json` format.

```
{
    "comparison_id": "ASAP2_vs_ASAP2",
    "experiment_id_old": "ASAP2",
    "experiment_dir_old": "../rsmtool/",
    "description_old": "RSMTool experiment.",
    "use_scaled_predictions_old": true,
    "experiment_id_new": "ASAP2",
    "experiment_dir_new": "../rsmtool",
    "description_new": "RSMTool experiment (copy).",
    "use_scaled_predictions_new": true
}
```

Let's take a look at the options in our configuration file.

- **Line 2**: We provide an ID for the comparison experiment.
- **Line 3**: We provide the `experiment_id` for the experiment we want to use as a baseline.
- **Line 4**: We also give the path to the directory containing the output of the original baseline experiment.
- **Line 5**: We give a short description of this baseline experiment. This will be shown in the report.
- **Line 6**: This field indicates that the baseline experiment used scaled scores for some evaluation analyses.
- **Line 7**: We provide the `experiment_id` for the new experiment. We use the same experiment ID for both experiments since we are comparing the experiment to itself.
- **Line 8**: We also give the path to the directory containing the output of the new experiment. As above, we use the same path because we are comparing the experiment to itself.
- **Line 9**: We give a short description of the new experiment. This will also be shown in the report.
- **Line 10**: This field indicates that the new experiment also used scaled scores for some evaluation analyses.

Documentation for all of the available configuration options is available *here*.

Note: You can also use our nifty capability to *automatically generate* `rsmcompare` configuration files rather than creating them manually.

Run the experiment

Now that we have the two experiments we want to compare and our configuration file in `.json` format, we can use the *rsmcompare* command-line script to run our comparison experiment.

```
$ cd examples/rsmcompare
$ rsmcompare config_rsmcompare.json
```

This should produce output like:

```
Output directory: /Users/nmadnani/work/rsmtool/examples/rsmcompare
Starting report generation
Merging sections
Exporting HTML
Executing notebook with kernel: python3
```

Once the run finishes, you will see an HTML file named `ASAP2_vs_ASAP2_report.html`. This is the final `rsmcompare` comparison report.

Examine the report

Our experiment report contains all the information we would need to compare the new model to the baseline model. It includes:

1. Comparison of feature distributions between the two experiments.
2. Comparison of model coefficients between the two experiments.
3. Comparison of model performance between the two experiments.

Note: Since we are comparing the experiment to itself, the comparison is not very interesting, e.g., the differences between various values will always be 0.

Input

`rsmcompare` requires a single argument to run an experiment: the path to *a configuration file*. It can also take an output directory as an optional second argument. If the latter is not specified, `rsmcompare` will use the current directory as the output directory.

Here are all the arguments to the `rsmcompare` command-line script.

`config_file`
The *JSON configuration file* for this experiment.

`output_dir` `(optional)`
The output directory where the report files for this comparison will be stored.

`-h, --help`
Show help message and exist.

`-V, --version`
Show version number and exit.

Experiment configuration file

This is a file in `.json` format that provides overall configuration options for an `rsmcompare` experiment. Here's an example configuration file for `rsmcompare`.

Note: To make it easy to get started with `rsmcompare`, we provide a way to **automatically generate** configurations file both interactively as well as non-interactively. Novice users will find interactive generation more helpful while more advanced users will prefer non-interactive generation. See *this page* for more details.

Next, we describe all of the `rsmcompare` configuration fields in detail. There are seven required fields and the rest are all optional. We first describe the required fields and then the optional ones (sorted alphabetically).

comparison_id

An identifier for the comparison experiment that will be used to name the report. It can be any combination of alphanumeric values, must *not* contain spaces, and must *not* be any longer than 200 characters.

experiment_id_old

An identifier for the "baseline" experiment. This ID should be identical to the `experiment_id` used when the baseline experiment was run, whether `rsmtool` or `rsmeval`. The results for this experiment will be listed first in the comparison report.

experiment_id_new

An identifier for the experiment with the "new" model (e.g., the model with new feature(s)). This ID should be identical to the `experiment_id` used when the experiment was run, whether `rsmtool` or `rsmeval`. The results for this experiment will be listed first in the comparison report.

experiment_dir_old

The directory with the results for the "baseline" experiment. This directory is the output directory that was used for the experiment and should contain subdirectories `output` and `figure` generated by `rsmtool` or `rsmeval`.

experiment_dir_new

The directory with the results for the experiment with the new model. This directory is the output directory that was used for the experiment and should contain subdirectories `output` and `figure` generated by `rsmtool` or `rsmeval`.

description_old

A brief description of the "baseline" experiment. The description can contain spaces and punctuation.

description_new

A brief description of the experiment with the new model. The description can contain spaces and punctuation.

custom_sections *(Optional)*

A list of custom, user-defined sections to be included into the final report. These are IPython notebooks (`.ipynb` files) created by the user. The list must contains paths to the notebook files, either absolute or relative to the configuration file. All custom notebooks have access to some *pre-defined variables*.

general_sections *(Optional)*

RSMTool provides pre-defined sections for `rsmcompare` (listed below) and, by default, all of them are included in the report. However, you can choose a subset of these pre-defined sections by specifying a list as the value for this field.

- `feature_descriptives`: Compares the descriptive statistics for all raw feature values included in the model:
  - a table showing mean, standard deviation, skewness and kurtosis;
  - a table showing the number of truncated outliers for each feature; and
  - a table with percentiles and outliers;
  - a table with correlations between raw feature values and human score in each model and the correlation between the values of the same feature in these two models. Note that this table only includes features and responses which occur in both training sets.
- `features_by_group`: Shows boxplots for both experiments with distributions of raw feature values by each of the *subgroups* specified in the configuration file.
- `preprocessed_features`: Compares analyses of preprocessed features:
  - histograms showing the distributions of preprocessed features values;
  - the correlation matrix between all features and the human score;
  - a table showing marginal correlations between all features and the human score; and
  - a table showing partial correlations between all features and the human score.
- `preprocessed_features_by_group`: Compares analyses of preprocessed features by subgroups: marginal and partial correlations between each feature and human score for each subgroup.
- `consistency`: Compares metrics for human-human agreement, the difference ('degradation') between the human-human and human-system agreement, and the disattenuated correlations for the whole dataset and by each of the *subgroups* specified in the configuration file.
- `score_distributions`:
  - tables showing the distributions for both human and machine scores; and
  - confusion matrices for human and machine scores.
- `model`: Compares the parameters of the two regression models. For linear models, it also includes the standardized and relative coefficients.
- `evaluation`: Compares the standard set of evaluations recommended for scoring models on the evaluation data.
- `true_score_evaluation`: compares the evaluation of system scores against the true scores estimated according to test theory. The notebook shows:
  - Number of single and double-scored responses.
  - Variance of human rater errors and estimated variance of true scores
  - Mean squared error (MSE) and proportional reduction in mean squared error (PRMSE) when predicting true score with system score.
- `pca`: Shows the results of principal components analysis on the processed feature values for the new model only:
  - the principal components themselves;

  - the variances; and
  - a Scree plot.

- `notes`: Notes explaining the terminology used in comparison reports.
- `sysinfo`: Shows all Python packages along with versions installed in the current environment while generating the report.

section_order *(Optional)*

A list containing the order in which the sections in the report should be generated. Any specified order must explicitly list:

1. Either *all* pre-defined sections if a value for the *general_sections* field is not specified OR the sections specified using *general_sections*, and
2. *All* custom section names specified using *custom_ sections*, i.e., file prefixes only, without the path and without the *.ipynb* extension, and
3. *All* special sections specified using *special_sections*.

special_sections *(Optional)*

A list specifying special ETS-only comparison sections to be included into the final report. These sections are available *only* to ETS employees via the *rsmextra* package.

subgroups *(Optional)*

A list of column names indicating grouping variables used for generating analyses specific to each of those defined subgroups.For example, `["prompt, gender, native_language, test_country"]`.

Note: In order to include subgroups analyses in the comparison report, both experiments must have been run with the same set of subgroups.

---

use_scaled_predictions_old *(Optional)*

Set to `true` if the "baseline" experiment used scaled machine scores for confusion matrices, score distributions, subgroup analyses, etc. Defaults to `false`.

use_scaled_predictions_new *(Optional)*

Set to `true` if the experiment with the new model used scaled machine scores for confusion matrices, score distributions, subgroup analyses, etc. Defaults to `false`.

Warning: For `rsmtool` and `rsmeval`, primary evaluation analyses are computed on both raw and scaled scores, but some analyses (e.g., the confusion matrix) are only computed for either raw or re-scaled scores based on the value of `use_scaled_predictions`. `rsmcompare` uses the existing outputs and does not perform any additional evaluations. Therefore if this field was set to `true` in the original experiment but is set to `false`

for `rsmcompare`, the report will be internally inconsistent: some evaluations use raw scores whereas others will use scaled scores.

use_thumbnails *(Optional)*

If set to `true`, the images in the HTML will be set to clickable thumbnails rather than full-sized images. Upon clicking the thumbnail, the full-sized images will be displayed in a separate tab in the browser. If set to `false`, full-sized images will be displayed as usual. Defaults to `false`.

Output

`rsmcompare` produces the comparison report in HTML format as well as in the form of a Jupyter notebook (a `.ipynb` file) in the output directory.

1.7.4 `rsmsummarize` - Compare multiple scoring models

RSMTool provides the `rsmsummarize` command-line utility to compare multiple models and to generate a comparison report. Unlike `rsmcompare` which creates a detailed comparison report between the two models, `rsmsummarize` can be used to create a more general overview of multiple models.

`rsmsummarize` can be used to compare:

1. Multiple `rsmtool` experiments, or
2. Multiple `rsmeval` experiments, or
3. A mix of `rsmtool` and `rsmeval` experiments (in this case, only the evaluation analyses will be compared).

Note: It is strongly recommend that the original experiments as well as the summary experiment are all done using the same version of RSMTool.

Tutorial

For this tutorial, you first need to *install RSMTool* and make sure the conda environment in which you installed it is activated.

Workflow

`rsmsummarize` is designed to compare several existing `rsmtool` or `rsmeval` experiments. To use `rsmsummarize` you need:

1. Two or more experiments that were run using *rsmtool* or *rsmeval*.
2. Create an *experiment configuration file* describing the comparison experiment you would like to run.
3. Run that configuration file with *rsmsummarize* and generate the comparison experiment HTML report.
4. Examine HTML report to compare the models.

Note that the above workflow does not use the customization features of `rsmsummarize`, e.g., *choosing which sections to include in the report* or *adding custom analyses sections* etc. However, we will stick with this workflow for our tutorial since it is likely to be the most common use case.

ASAP Example

We are going to use the same example from 2012 Kaggle competition on automated essay scoring that we used for the *rsmtool tutorial*.

Run `rsmtool` and `rsmeval` experiments

`rsmsummarize` compares the results of the two or more existing `rsmtool` (or `rsmeval`) experiments. For this tutorial, we will compare model trained in the *rsmtool tutorial* to the evaluations we obtained in the *rsmeval tutorial*.

Note: If you have not already completed these tutorials, please do so now. You may need to complete them again if you deleted the output files.

Create a configuration file

The next step is to create an *experiment configuration file* in `.json` format.

```
1 {
2   "summary_id": "model_comparison",
3   "description": "a comparison of the results of the rsmtool sample experiment,␣
  ↪rsmeval sample experiment and once again the rsmtool sample experiment",
4   "experiment_dirs": ["../rsmtool", "../rsmeval", "../rsmtool"],
5   "experiment_names":["RSMTool experiment 1", "RSMEval experiment", "RSMTool␣
  ↪experiment 2"]
6 }
```

Let's take a look at the options in our configuration file.

- **Line 2**: We provide the `summary_id` for the comparison. This will be used to generate the name of the final report.
- **Line 3**: We give a short description of this comparison experiment. This will be shown in the report.
- **Line 4**: We also give the list of paths to the directories containing the outputs of the experiments we want to compare.
- **Line 5**: Since we want to compare experiments that all used the same experiment id (`ASAP2`), we instead list the names that we want to use for each experiment in the summary report.

Documentation for all of the available configuration options is available *here*.

Note: You can also use our nifty capability to *automatically generate* `rsmsummarize` configuration files rather than creating them manually.

Run the experiment

Now that we have the list of the experiments we want to compare and our configuration file in `.json` format, we can use the *rsmsummarize* command-line script to run our comparison experiment.

```
$ cd examples/rsmsummarize
$ rsmsummarize config_rsmsummarize.json
```

This should produce output like:

```
Output directory: /Users/nmadnani/work/rsmtool/examples/rsmsummarize
Starting report generation
Merging sections
Exporting HTML
Executing notebook with kernel: python3
```

Once the run finishes, you will see a new folder `report` containing an HTML file named `model_comparison_report.html`. This is the final `rsmsummarize` summary report.

Examine the report

Our experiment report contains the overview of main aspects of model performance. It includes:

1. Brief description of all experiments.
2. Information about model parameters and model fit for all `rsmtool` experiments.
3. Model performance for all experiments.

Note: Some of the information such as model fit and model parameters are only available for `rsmtool` experiments.

Input

`rsmsummarize` requires a single argument to run an experiment: the path to *a configuration file*. You can specify which models you want to compare and the name of the report by supplying the path to *a configuration file*. It can also take an output directory as an optional second argument. If the latter is not specified, `rsmsummarize` will use the current directory as the output directory.

Here are all the arguments to the `rsmsummarize` command-line script.

config_file
: The *JSON configuration file* for this experiment.

output_dir (optional)
: The output directory where the report and intermediate `.csv` files for this comparison will be stored.

-f, --force
: If specified, the contents of the output directory will be overwritten even if it already contains the output of another rsmsummarize experiment.

-h, --help
: Show help message and exist.

-V, --version
: Show version number and exit.

Experiment configuration file

This is a file in `.json` format that provides overall configuration options for an `rsmsummarize` experiment. Here's an example configuration file for `rsmsummarize`.

Note: To make it easy to get started with `rsmsummarize`, we provide a way to **automatically generate** configurations file both interactively as well as non-interactively. Novice users will find interactive generation more helpful while more advanced users will prefer non-interactive generation. See *this page* for more details.

Next, we describe all of the `rsmsummarize` configuration fields in detail. There are two required fields and the rest are all optional. We first describe the required fields and then the optional ones (sorted alphabetically).

summary_id

An identifier for the `rsmsummarize` experiment. This will be used to name the report. It can be any combination of alphanumeric values, must *not* contain spaces, and must *not* be any longer than 200 characters.

experiment_dirs

The list of the directories with the results of the experiment. These directories should be the output directories used for each experiment and should contain subdirectories `output` and `figure` generated by `rsmtool` or `rsmeval`.

custom_sections *(Optional)*

A list of custom, user-defined sections to be included into the final report. These are IPython notebooks (`.ipynb` files) created by the user. The list must contains paths to the notebook files, either absolute or relative to the configuration file. All custom notebooks have access to some *pre-defined variables*.

description *(Optional)*

A brief description of the summary. The description can contain spaces and punctuation.

experiment_names *(Optional)*

The list of experiment names to use in the summary report and *intermediate files*. The names should be listed in the same order as the experiments in *experiment_dirs*. When this field is not specified, the report will show the original `experiment_id` for each experiment.

file_format *(Optional)*

The format of the *intermediate files* generated by `rsmsummarize`. Options are `csv`, `tsv`, or `xlsx`. Defaults to `csv` if this is not specified.

Note: In the `rsmsummarize` context, the `file_format` parameter refers to the format of the intermediate files generated by `rsmsummarize`, not the intermediate files generated by the original experiment(s) being summarized. The format of these files does not have to match the format of the files generated by the original experiment(s).

general_sections *(Optional)*

RSMTool provides pre-defined sections for `rsmsummarize` (listed below) and, by default, all of them are included in the report. However, you can choose a subset of these pre-defined sections by specifying a list as the value for this field.

- `preprocessed_features` : compares marginal and partial correlations between all features and the human score, and optionally response length if this was computed for any of the models.
- `model`: Compares the parameters of the two regression models. For linear models, it also includes the standardized and relative coefficients.
- `evaluation`: Compares the standard set of evaluations recommended for scoring models on the evaluation data.
- `true_score_evaluation`: compares the evaluation of system scores against the true scores estimated according to test theory. The notebook shows:
  - Number of single and double-scored responses.
  - Variance of human rater errors and estimated variance of true scores
  - Mean squared error (MSE) and proportional reduction in mean squared error (PRMSE) when predicting true score with system score.
- `intermediate_file_paths`: Shows links to all of the intermediate files that were generated while running the summary.
- `sysinfo`: Shows all Python packages along with versions installed in the current environment while generating the report.

section_order *(Optional)*

A list containing the order in which the sections in the report should be generated. Any specified order must explicitly list:

1. Either *all* pre-defined sections if a value for the *general_sections* field is not specified OR the sections specified using *general_sections*, and
2. *All* custom section names specified using *custom_ sections*, i.e., file prefixes only, without the path and without the *.ipynb* extension, and
3. *All* special sections specified using *special_sections*.

special_sections *(Optional)*

A list specifying special ETS-only comparison sections to be included into the final report. These sections are available *only* to ETS employees via the *rsmextra* package.

use_thumbnails *(Optional)*

If set to `true`, the images in the HTML will be set to clickable thumbnails rather than full-sized images. Upon clicking the thumbnail, the full-sized images will be displayed in a separate tab in the browser. If set to `false`, full-sized images will be displayed as usual. Defaults to `false`.

Output

`rsmsummarize` produces a set of folders in the output directory.

report

This folder contains the final `rsmsummarize` report in HTML format as well as in the form of a Jupyter notebook (a `.ipynb` file).

output

This folder contains all of the *intermediate files* produced as part of the various analyses performed, saved as `.csv` files. `rsmsummarize` will also save in this folder a copy of the *configuration file*. Fields not specified in the original configuration file will be pre-populated with default values.

figure

This folder contains all of the figures that may be generated as part of the various analyses performed, saved as `.svg` files. Note that no figures are generated by the existing `rsmsummarize` notebooks.

Intermediate files

Although the primary output of RSMSummarize is an HTML report, we also want the user to be able to conduct additional analyses outside of RSMTool. To this end, all of the tables produced in the experiment report are saved as files in the format as specified by `file_format` parameter in the `output` directory. The following sections describe all of the intermediate files that are produced.

Note: The names of all files begin with the `summary_id` provided by the user in the experiment configuration file.

Marginal and partial correlations with score

filenames: `margcor_score_all_data`, `pcor_score_all_data`, `` `pcor_score_no_length_all_data ``

The first file contains the marginal correlations between each pre-processed feature and human score. The second file contains the partial correlation between each pre-processed feature and human score after controlling for all other features. The third file contains the partial correlations between each pre-processed feature and human score after controlling for response length, if `length_column` was specified in the configuration file.

Model information

- `model_summary`

This file contains the main information about the models included into the report including:

- Total number of features
- Total number of features with non-negative coefficients
- The learner

- The label used to train the model
- `betas`: standardized coefficients (for built-in models only).
- `model_fit`: R squared and adjusted R squared computed on the training set. Note that these values are always computed on raw predictions without any trimming or rounding.

Note: If the report includes a combination of `rsmtool` and `rsmeval` experiments, the summary tables with model information will only include `rsmtool` experiments since no model information is available for `rsmeval` experiments.

Evaluation metrics

- `eval_short` - descriptives for predicted and human scores (mean, std.dev etc.) and association metrics (correlation, quadartic weighted kappa, SMD etc.) for specific score types chosen based on recommendations by Williamson (2012). Specifically, the following columns are included (the `raw` or `scale` version is chosen depending on the value of the `use_scaled_predictions` in the configuration file).
  - h_mean
  - h_sd
  - corr
  - sys_mean [raw/scale trim]
  - sys_sd [raw/scale trim]
  - SMD [raw/scale trim]
  - adj_agr [raw/scale trim_round]
  - exact_agr [raw/scale trim_round]
  - kappa [raw/scale trim_round]
  - wtkappa [raw/scale trim_round]
  - sys_mean [raw/scale trim_round]
  - sys_sd [raw/scale trim_round]
  - SMD [raw/scale trim_round]
  - R2 [raw/scale trim]
  - RMSE [raw/scale trim]

Note: Please note that for raw scores, SMD values are likely to be affected by possible differences in scale.

Evaluations based on test theory

- `true_score_eval`: evaluations of system scores against estimated true score. Contains total counts of single and double-scored response, variance of human rater error, estimated true score variance, and mean squared error (MSE) and proportional reduction in mean squared error (PRMSE) when predicting true score using system score.

1.7.5 `rsmxval` - Run cross-validation experiments

RSMTool provides the `rsmxval` command-line utility to run cross-validation experiments with scoring models. Why would one want to use cross-validation rather than just using the simple train-and-evaluate loop provided by the `rsmtool` utility? Using cross-validation can provide more accurate estimates of scoring model performance since those estimates are averaged over *multiple* train-test splits that are randomly selected based on the data. Using a single train-test split may lead to biased estimates of performance since those estimates will depend on the specific characteristics of that split. Using cross-validation is more likely to provide estimates of how well the scoring model will generalize to unseen test data, and more easily flag problems with overfitting and selection bias, if any.

Cross-validation experiments in RSMTool consist of the following steps:

1. The given training data file is first shuffled randomly (with a fixed seed for reproducibility) and then split into the requested number of folds. It is also possible for the user to provide a CSV file containing a pre-determined set of folds, e.g., from another part of the data pipeline.
2. For each fold (or train-test split), `rsmtool` is run to train a scoring model on the training split and evaluate on the test split. All of the outputs for each of the `rsmtool` runs are saved on disk and represent the per-fold performance.
3. The predictions generated by `rsmtool` for each of the folds are all combined into a single file, which is then used as input for `rsmeval`. The output of this evaluation run is saved to disk and provides a more accurate estimate of the predictive performance of a scoring model trained on the given data.
4. A summary report comparing all of the folds is generated by running `rsmsummarize` on all of the fold directories created in the Step 1 and its output is also saved to disk. This summary output can be useful to see if the performance for any of the folds stands out for any reason, which could point to a potential problem.
5. Finally, a scoring model is trained on the complete training data file using `rsmtool`, which also generates a report that contains only the feature and model descriptives. The model is what will most likely be deployed for inference assuming the analyses produced in this step and Steps 1–4 meet the stakeholders' requirements.

Tutorial

For this tutorial, you first need to *install RSMTool* and make sure the conda environment in which you installed it is activated.

Workflow

`rsmxval` is designed to run cross-validation experiments using a single file containing human scores and features. Just like `rsmtool`, `rsmxval` does not provide any functionality for feature extraction and assumes that users will extract features on their own. The workflow steps are as follows:

1. Create a data file in one of the *supported formats* containing the extracted features for each response in the data along with human score(s) assigned to it.
2. Create an *experiment configuration file* describing the cross-validation experiment you would like to run.
3. Run that configuration file with *rsmxval* and generate its *outputs*.
4. Examine the various HTML reports to check various aspects of model performance.

Note that unlike `rsmtool` and `rsmeval`, `rsmxval` currently does not support customization of the HTML reports generated in each step. This functionality may be added in future versions.

ASAP Example

We are going to use the same example from the 2012 Kaggle competition on automated essay scoring that we used for the *rsmtool tutorial*.

Extract features

We are using the same features for this data as described in the *rsmtool tutorial*.

Create a configuration file

The next step is to create an *experiment configuration file* in `.json` format.

```
{
    "experiment_id": "ASAP2_xval",
    "description": "Cross-validation with two human scores using a LinearRegression model.",
    "train_file": "train.csv",
    "folds": 3,
    "train_label_column": "score",
    "id_column": "ID",
    "model": "LinearRegression",
    "trim_min": 1,
    "trim_max": 6,
    "second_human_score_column": "score2",
    "use_scaled_predictions": true
}
```

Let's take a look at the options in our configuration file.

- **Line 2**: We define an experiment ID used to identify the files produced as part of this experiment.
- **Line 3**: We provide a description which will be included in the various reports.
- **Line 4**: We list the path to our training file with the feature values and human scores. For this tutorial, we used `.csv` format, but several other *input file formats* are also supported.
- **Line 5**: This field indicates the number of cross-validation folds we want to use. If this field is not specified, `rsmxval` uses 5-fold cross-validation by default.
- **Line 6**: This field indicates that the human (reference) scores in our `.csv` file are located in a column named `score`.
- **Line 7**: This field indicates that the unique IDs for the responses in the `.csv` file are located in a column named `ID`.
- **Line 8**: We choose to use a linear regression model to combine the feature values into a score.
- **Lines 9-10**: These fields indicate that the lowest score on the scoring scale is a 1 and the highest score is a 6. This information is usually part of the rubric used by human graders.
- **Line 11**: This field indicates that scores from a second set of human graders are also available (useful for comparing the agreement between human-machine scores to the agreement between two sets of humans) and are located in the `score2` column in the training `.csv` file.
- **Line 12**: Next, we indicate that we would like to use the *scaled scores* for all our evaluation analyses at each step.

Documentation for all of the available configuration options is available *here*.

Note: You can also use our nifty capability to *automatically generate* `rsmxval` configuration files rather than creating them manually.

Run the experiment

Now that we have our input file and our configuration file, we can use the *rsmxval* command-line script to run our evaluation experiment.

```
$ cd examples/rsmxval
$ rsmxval config_rsmxval.json output
```

This should produce output like:

```
Output directory: output
Saving configuration file.
Generating 3 folds after shuffling
Running RSMTool on each fold in parallel
Progress: 100%|| 3/3 [00:08<00:00,  2.76s/it]
Creating fold summary
Evaluating combined fold predictions
Training model on full data
```

Once the run finishes, you will see an `output` sub-directory in the current directory. Under this directory you will see multiple sub-directories, each corresponding to a different cross-validation step, as described *here*.

Examine the reports

The cross-validation experiment produces multiple HTML reports – an `rsmtool` report for each of the 3 folds (`output/folds/{01,02,03}/report/ASAP2_xval_fold{01,02,03}.html`), the evaluation report for the cross-validated predictions (`output/evaluation/report/ASAP2_xval_evaluation_report.html`), a report summarizing the salient characteristics of the 3 folds (`output/fold-summary/report/ASAP2_xval_fold_summary_report.html`), and a report showing the feature and model descriptives (`output/final-model/report/ASAP2_xval_model_report.html`). Examining these reports will provide a relatively complete picture of how well the predictive performance of the scoring model will generalize to unseen data.

Input

`rsmxval` requires a single argument to run an experiment: the path to *a configuration file*. It can also take an output directory as an optional second argument. If the latter is not specified, `rsmxval` will use the current directory as the output directory.

Here are all the arguments to the `rsmxval` command-line script.

`config_file`
: The *JSON configuration file* for this cross-validation experiment.

`output_dir` `(optional)`
: The output directory where all the sub-directories and files for this cross-validation experiment will be stored. If a non-empty directory with the same name already exists, an error will be raised.

-h, --help
Show help message and exit.

-V, --version
Show version number and exit.

Experiment configuration file

This is a file in `.json` format that provides overall configuration options for an `rsmxval` experiment. Here's an example configuration file for `rsmxval`.

Note: To make it easy to get started with `rsmxval`, we provide a way to **automatically generate** configuration files both interactively as well as non-interactively. Novice users will find interactive generation more helpful while more advanced users will prefer non-interactive generation. See *this page* for more details.

Configuration files for `rsmxval` are almost identical to `rsmtool` configuration files with only a few differences. Next, we describe the three required `rsmxval` configuration fields in detail.

experiment_id

An identifier for the experiment that will be used as part of the names of the reports and intermediate files produced in each of the steps. It can be any combination of alphanumeric values, must *not* contain spaces, and must *not* be any longer than 200 characters. Suffixes are added to this experiment ID by each of the steps for the reports and files they produce, i.e., `_fold<N>` in the per-fold `rsmtool` step where `<N>` is a two digit number, `_evaluation` by the `rsmeval` evaluation step, `_fold_summary` by the `rsmsummarize` step, and `_model` by the final full-data `rsmtool` step.

model

The machine learner you want to use to build the scoring model. Possible values include *built-in linear regression models* as well as all of the learners available via SKLL. With SKLL learners, you can customize the *tuning objective* and also *compute expected scores as predictions*.

train_file

The path to the training data feature file in one of the *supported formats*. Each row should correspond to a single response and contain numeric feature values extracted for this response. In addition, there should be a column with a unique identifier (ID) for each response and a column with the human score for each response. The path can be absolute or relative to config file's location.

Important: Unlike `rsmtool`, `rsmxval` does not accept an evaluation set and will raise an error if the `test_file` field is specified.

Next, we will describe the two optional fields that are unique to `rsmxval`.

folds *(Optional)*

The number of folds to use for cross-validation. This should be an integer and defaults to 5.

folds_file *(Optional)*

The path to a file containing custom, pre-specified folds to be used for cross-validation. This should be a `.csv` file (no other formats are accepted) and should contain only two columns: `id` and `fold`. The `id` column should contain the same IDs of the responses that are contained in `train_file` above. The `fold` column should contain an integer representing which fold the response with the `id` belongs to. IDs not specified in this file will be skipped and not included in the cross-validation at all. Just like `train_file`, this path can be absolute or relative to the config file's location. Here's an example of a folds file containing 2 folds.

Note: If *both* `folds_file` and `folds` are specified, then the former will take precedence unless it contains a non-existent path.

In addition to the fields described so far, an `rsmxval` configuration file also accepts the following optional fields used by `rsmtool`:

- `candidate_column`
- `description`
- `exclude_zero_scores`
- `feature_subset`
- `feature_subset_file`
- `features`
- `file_format`
- `flag_column`
- `flag_column_test`
- `id_column`
- `length_column`
- `min_items_per_candidate`
- `min_n_per_group`
- `predict_expected_scores`
- `rater_error_variance`
- `second_human_score_column`
- `select_transformations`
- `sign`
- `skll_fixed_parameters`
- `skll_objective`
- `standardize_features`
- `subgroups`
- `train_label_column`
- `trim_max`
- `trim_min`
- `trim_tolerance`

- use_scaled_predictions
- use_thumbnails
- use_truncation_thresholds

Please refer to these fields' descriptions on the page describing the *rsmtool configuration file*.

Output

rsmxval produces a set of folders in the output directory.

folds

This folder contains the output of each of the per-fold rsmtool experiments. It contains as many sub-folders as the number of specified folds, named 01, 02, 03, etc. Each of these numbered sub-folders contains the output of one rsmtool experiment conducted using the training split of that fold as the training data and the test split as the evaluation data. Each of the sub-folders contains the *output directories produced by rsmtool*. The report for each fold lives in the report sub-directory, e.g., the report for the first fold is found at folds/01/report/<experiment_id>_fold01_report.html, and so on. The messages that are usually printed out by rsmtool to the screen are instead logged to a file and saved to disk as, e.g., folds/01/rsmtool.log.

evaluation

This folder contains the output of the rsmeval evaluation experiment that uses the cross-validated predictions from each fold. This folder contains the *output directories produced by rsmeval*. The evaluation report can be found at evaluation/report/<experiment_id>_evaluation_report.html. The messages that are usually printed out by rsmeval to the screen are instead logged to a file and saved to disk as evaluation/rsmeval.log.

fold-summary

This folder contains the output of the rsmsummarize experiment that provides a quick summary of all of the folds in a single, easily-scanned report. The folder contains the *output directories produced by rsmsummarize*. The summary report can be found at fold-summary/report/<experiment_id>_fold_summary_report.html. The messages that are usually printed out by rsmsummarize to the screen are instead logged to a file and saved to disk as fold-summary/rsmsummarize.log.

final-model

This folder contains the output of the rsmtool experiment that trains a model on the full training data and provides a report showing the feature and model descriptives. It contains the *output directories produced by rsmtool*. The primary artifacts of this experiment are the report (final-model/report/<experiment_id>_model_report.html) and the final trained model (final-model/output/<experiment_id>_model.model). The messages that are usually printed out by rsmtool to the screen are instead logged to a file and saved to disk as final-model/rsmtool.log.

Note: Every rsmtool experiment requires both a training and an evaluation set. However, in this step, we are using the full training data to train the model and rsmxval does not use a separate test set. Therefore, we simply randomly sample 10% of the full training data as a dummy test set to make sure that rsmtool runs successfully. The

report in this step *only* contains the model and feature descriptives and, therefore, does not use this dummy test set at all. Users should ignore any intermediate files under the `final-model/output` and `final-model/figure` sub-directories that are derived from this dummy test set. If needed, the data used as the dummy test set can be found at `final-model/dummy_test.csv` (or in the *chosen format*).

In addition to these folders, `rsmxval` will also save a copy of the *configuration file* in the output directory at the same-level as the above folders. Fields not specified in the original configuration file will be pre-populated with default values.

1.8 Writing custom RSMTool sections

RSMTool allows users to include custom Jupyter notebooks in the reports generated by *rsmtool*, *rsmeval*, and *rsmcompare* using the `custom_sections` fields in their configuration files. This can be particularly useful if a researcher wants to include custom analyses specific to their own scoring engine; she can do so while still using the familiar RSMTool pipeline.

1.8.1 Available variables

When writing such notebooks, some or all of the python variables below will be available for use in the notebooks.

experiment_id

The experiment ID from the respective configuration file.

description

The description string from the respective configuration file.

model_name

The name of the model from the respective configuration file.

context

Two possible values: `rsmtool` or `rsmeval` depending on which command-line tool is being used to generated the report (not available in `rsmcompare` notebooks).

use_scaled_predictions

A boolean Python variable containing the value of the setting with the same name from the respective configuration file.

exclude_zero_scores

A boolean Python variable containing the value of the setting with the same name from the respective configuration file.

length_column *(rsmtool only)*

The name of the column in the training and/or evaluation data containing response length. `None` if not specified in the configuration file.

second_human_score_column

The name of the column in the evaluation data containing the second human score. `None` if not specified in the respective configuration file.

groups_eval

A list containing the names of metadata or subgroup columns as specified in the respective configuration file.

min_items

The minimal number of items expected from each candidate. The value if set to 0 if the user did not specify a minimal number in the respective configuration file.

features_used *(rsmtool only)*

A list containing the names of all the features that are used for training the model. [*rsmtool* only]

In addition, several pandas data frames are also available. Many of these contain the same information produced in the intermediate CSV files produced by *rsmtool* or *rsmeval*. We have made these available to authors of custom notebooks to avoid the need for reading them from disk.

df_features *(rsmtool only)*

A data frame containing information about the feature columns that were included in the final model training. Same information as in *feature.csv*.

df_betas *(rsmtool only)*

Relative and standardized coefficients (*betas.csv*)

df_train_orig *(rsmtool only)*

df_test_orig *(rsmtool only)*

Data frames containing the original training and testing data as specified in the config file, without any changes.

df_train *(rsmtool only)*

df_train_preproc *(rsmtool only)*

df_test *(rsmtool only)*

df_test_preproc *(rsmtool only)*

Data frames containing the *raw and pre-processed feature values*.

df_train_other_columns *(rsmtool only)*

df_test_other_columns

Data frames containing the *unused columns* from the training and evaluation data.

df_train_responses_with_excluded_flags *(rsmtool only)*

df_test_responses_with_excluded flags

Data frames containing the *flagged responses*.

df_train_length *(rsmtool only)*

A data frame containing response lengths under the `length` column for the training data, along with the response IDs under the `spkitemid` column. These are *only* available (a) if *length_column* was specified in the configuration file, and (b) if no values in that column are missing, and (c) if the values in that column are not distributed with a standard deviation <= 0.

df_test_human_scores

A data frame containing the two human scores for the responses in the evaluation data under the `sc1` and `sc2` columns, along with the response IDs under the `spkitemid` column. This frame is *only* available if *second_human_score_column* was specified in the config file.

Note: This data frame will contain `NaN` for the responses for which no numeric second human score was available or for which the second score was 0 and exclude_zero_scores was set to `true`.

---

df_pred_preproc

A data frame containing the *raw and post-processed predictions for the evaluation data*.

df_feature_subset_specs *(rsmtool only)*

A data frame containing the contents of *feature_subset_file* if it was specified in configuration file. `None` if not specified.

Finally, the following variables are also available but you are strong encouraged *not* to re-read the files under these directories which are already available as data frames.

output_dir

The `output` sub-directory under the experiment output directory that contains all the intermediate CSV files.

figure_dir

The `figure` sub-directory under the experiment output directory that contains all of the generated SVG and PNG figures.

Note: All dataframes apart from `df_train_orig` and `df_test_orig` contain an `spkitemid` column which contains the unique response IDs.

All data frames except the `df_*_other_columns` contain an `sc1` column which contains the human score for the responses.

`df_train_orig` and `df_test_orig` will contain the response IDs and human scores under columns with the original names, *not* `spkitemid` and `sc1`.

1.9 Auto-generating configuration files

Configuration files for *rsmtool*, *rsmeval*, *rsmcompare*, *rsmpredict*, *rsmsummarize*, and *rsmxval* can be difficult to create manually due to the large number of configuration options supported by these tools. To make this easier for users, all of these tools support *automatic* creation of configuration files, both interactively and non-interactively.

1.9.1 Interactive generation

For novice users, it is easiest to use a guided, interactive mode to generate a configuration file. For example, to generate an `rsmtool` configuration file interactively, run the following command:

```
rsmtool generate --interactive > example_rsmtool.json
```

The following screencast shows an example interactive session after the above command is run (click to play):

The configuration file `example_rsmtool.json` generated via the session is shown below:

```
{
    // Reference: https://rsmtool.readthedocs.io/en/stable/usage_rsmtool.html
↪#experiment-configuration-file
    "experiment_id": "test_rsmtool",
    "model": "LinearRegression",
    "train_file": "/Users/nmadnani/train.csv",
    "test_file": "/Users/nmadnani/test.csv",
    // OPTIONAL: replace default values below based on your data.
    "candidate_column": null,
    "custom_sections": null,
    "description": "Test experiment.",
    "exclude_zero_scores": true,
    "feature_subset": null,
    "feature_subset_file": null,
    "features": null,
    "file_format": "xlsx",
    "flag_column": null,
    "flag_column_test": null,
    "general_sections": [
        "data_description",
        "feature_descriptives",
        "preprocessed_features",
```

(continues on next page)

(continued from previous page)

```
        "consistency",
        "model",
        "evaluation",
        "true_score_evaluation",
        "pca",
        "intermediate_file_paths",
        "sysinfo"
    ],
    "id_column": "ID",
    "length_column": "length",
    "min_items_per_candidate": null,
    "min_n_per_group": null,
    "predict_expected_scores": false,
    "second_human_score_column": null,
    "section_order": null,
    "select_transformations": false,
    "sign": null,
    "skll_objective": null,
    "special_sections": null,
    "standardize_features": true,
    "subgroups": [],
    "test_label_column": "score",
    "train_label_column": "score",
    "trim_max": 1,
    "trim_min": 6,
    "trim_tolerance": 0.4998,
    "use_scaled_predictions": true,
    "use_thumbnails": false,
    "use_truncation_thresholds": false
}
```

Note: Although we use `rsmtool` in the example above, the same instructions apply to all 6 tools; simply replace `rsmtool` with `rsmeval`, `rsmcompare`, etc.

There are some configuration options that can accept multiple inputs. For example, the `experiment_dirs` option for `rsmsummarize` takes a list of `rsmtool` experiment directories for a summary report. These options are handled differently in interactive mode. To illustrate this, let's generate a configuration file for `rsmsummarize` by using the following command:

```
rsmsummarize generate --interactive > example_rsmsummarize.json
```

The following screencast shows the interactive session (click to play):

And here is the generated configuration file for `rsmsummarize`:

```
{
    // Reference: https://rsmtool.readthedocs.io/en/stable/advanced_usage.html#config-
↪file-rsmsummarize
    "summary_id": "test_summary",
    "experiment_dirs": [
        "/Users/nmadnani/work",
        "/Users/nmadnani/work/rsmtool",
        "/Users/nmadnani/work/rsmtool/tests"
    ],
    // OPTIONAL: replace default values below based on your data.
```

(continues on next page)

(continued from previous page)

```
    "custom_sections": null,
    "description": "This is a test.",
    "experiment_names": null,
    "file_format": "tsv",
    "general_sections": [
        "preprocessed_features",
        "model",
        "evaluation",
        "true_score_evaluation",
        "intermediate_file_paths",
        "sysinfo"
    ],
    "section_order": null,
    "special_sections": null,
    "subgroups": [],
    "use_thumbnails": true
}
```

Important: If you want to include subgroup information in the reports for `rsmtool`, `rsmeval`, `rsmcompare`, and `rsmxval`, you should add `--subgroups` to the command. For example, when you run `rsmeval generate --interactive --subgroups` you would be prompted to enter the subgroup column names and the `general_sections` list (if shown[1]) will also include subgroup-based sections. Since the `subgroups` option can accept multiple inputs, it is handled in the same way as the `experiment_dirs` option for `rsmsummarize` above.

We end with a list of important things to note about interactive generation:

- Carefully read the instructions and notes displayed at the top when you first enter interactive mode.
- If you do not redirect the output of the command to a file, the generated configuration file will simply be printed out.
- You may see messages like "invalid option" and "invalid file" on the bottom left while you are entering the value for a field. This is an artifact of real-time validation. For example, when choosing a training file for `rsmtool`, the message "invalid file" may be displayed while you navigate to the actual file. Once you get to a valid file, this message should disappear.
- Required fields will *not* accept a blank input (just pressing enter) and will show an error message in the bottom left until a valid input is provided.
- Optional fields will accept blank inputs since they have default values that will be used if no user input is provided. In some cases, default values are shown underlined in parentheses.
- You can also use `-i` as an alias for `--interactive` and `-g` as an alias for `--subgroups`. So, for example, if you want to interactively generate a configuration file with subgroups for `rsmtool`, just run `rsmtool generate -ig` instead of `rsmtool generate --interactive --subgroups`.
- The configuration files generated interactively contain comments (as indicated by `// ...`). While RSMTool handles JSON files with comments just fine, you may need to remove the comments manually if you wish to use these files outside of RSMTool.

[1] Recall that `rsmxval` does not support customizing the section list and, therefore, will not display the `general_sections` field in the auto-generated configuration file.

1.9.2 Non-interactive Generation

For more advanced or experienced users who want to quickly get started with a dummy configuration file that they feel comfortable editing manually, RSMTool also provides the capability to generate configuration files non-interactively. To do so, simply omit the `--interactive` switch in the commands above. For example, to generate a dummy configuration file for `rsmtool`, the command to run would be:

```
rsmtool generate > dummy_rsmtool.json
```

When running this command, the following warning would be printed out to stderr:

```
WARNING: Automatically generated configuration files MUST be edited to add values
for required fields and even for optional ones depending on your data
```

This warning explains that the generated file *cannot* be used directly as input to `rsmtool` since the required fields are filled with dummy values. This can be confirmed by looking at the configuration file the command generates:

```
{
    // Reference: https://rsmtool.readthedocs.io/en/stable/usage_rsmtool.html
↪#experiment-configuration-file
    // REQUIRED: replace "ENTER_VALUE_HERE" with the appropriate value!
    "experiment_id": "ENTER_VALUE_HERE",
    "model": "ENTER_VALUE_HERE",
    "train_file": "ENTER_VALUE_HERE",
    "test_file": "ENTER_VALUE_HERE",
    // OPTIONAL: replace default values below based on your data.
    "candidate_column": null,
    "custom_sections": null,
    "description": "",
    "exclude_zero_scores": true,
    "feature_subset": null,
    "feature_subset_file": null,
    "features": null,
    "file_format": "csv",
    "flag_column": null,
    "flag_column_test": null,
    "general_sections": [
        "data_description",
        "feature_descriptives",
        "preprocessed_features",
        "consistency",
        "model",
        "evaluation",
        "true_score_evaluation",
        "pca",
        "intermediate_file_paths",
        "sysinfo"
    ],
    "id_column": "spkitemid",
    "length_column": null,
    "min_items_per_candidate": null,
    "min_n_per_group": null,
    "predict_expected_scores": false,
    "second_human_score_column": null,
    "section_order": null,
    "select_transformations": false,
    "sign": null,
```

(continues on next page)

(continued from previous page)

```
    "skll_objective": null,
    "special_sections": null,
    "standardize_features": true,
    "subgroups": [],
    "test_label_column": "sc1",
    "train_label_column": "sc1",
    "trim_max": null,
    "trim_min": null,
    "trim_tolerance": 0.4998,
    "use_scaled_predictions": false,
    "use_thumbnails": false,
    "use_truncation_thresholds": false
}
```

Note the two comments demarcating the locations of the required and optional fields. Note also that the required fields are filled with the dummy value "ENTER_VALUE_HERE" that *must* be manually edited by the user. The optional fields are filled with default values that may also need to be further edited depending on the data being used.

Just like interactive generation, non-interactive generation is supported by all 6 tools: `rsmtool`, `rsmeval`, `rsmcompare`, `rsmpredict`, `rsmsummarize`, and `rsmxval`.

Similarly, to include subgroup information in the reports for `rsmtool`, `rsmeval`, and `rsmcompare`, just add `--subgroups` (or `-g`) to the command. Note that unlike in interactive mode, this would *only* add subgroup-based sections to the `general_sections` list in the output file. You will need to manually edit the `subgroups` option in the configuration file to enter the subgroup column names.

1.9.3 Generation API

Interactive generation is only meant for end users and can only be used via the 6 command-line tools `rsmtool`, `rsmeval`, `rsmcompare`, `rsmpredict`, `rsmsummarize`, and `rsmxval`. It cannot be used via the RSMTool API.

However, the non-interactive generation *can* be used via the API which can be useful for more advanced RSMTool users. To illustrate, here's some example Python code to generate a configuration for `rsmtool` in the form of a dictionary:

```
# import the ConfigurationGenerator class
from rsmtool.utils.commandline import ConfigurationGenerator

# instantiate it with the options as needed
#   we want a dictionary, not a string
#   we do not want to see any warnings
#   we want to include subgroup-based sections in the report
generator = ConfigurationGenerator('rsmtool',
                                   as_string=False,
                                   suppress_warnings=True,
                                   use_subgroups=True)

# generate the configuration dictionary
configdict = generator.generate()

# remember we still need to replace the dummy values
# for the required fields
configdict["experiment_id"] = "test_experiment"
configdict["model"] = "LinearRegression"
```

(continues on next page)

(continued from previous page)

```
configdict["train_file"] = "train.csv"
configdict["test_file"] = "test.csv"

# and don't forget about adding the subgroups
configdict["subgroups"] = ["GROUP1", "GROUP2"]

# make other changes to optional fields based on your data
...

# now we can use this dictionary to run an rsmtool experiment via the API
from rsmtool import run_experiment
run_experiment(configdict, "/tmp/output")
```

For more details, refer to the API documentation for the *ConfigurationGenerator* class.

1.10 API Documentation

The primary method of using RSMTool is via the command-line scripts *rsmtool*, *rsmeval*, *rsmpredict*, *rsmcompare*, and *rsmsummarize*. However, there are certain functions in the `rsmtool` API that may also be useful to advanced users for use directly in their Python code. We document these functions below.

Note: RSMTool v5.7 and older provided the API functions `metrics_helper`, `convert_ipynb_to_html`, and `remove_outliers`. These functions have now been turned into static methods for different classes.

In addition, with RSMTool v8.0 onwards, the functions `agreement`, `difference_of_standardized_means`, `get_thumbnail_as_html`, `parse_json_with_comments`, `partial_correlations`, `quadratic_weighted_kappa`, `show_thumbnail`, and `standardized_mean_difference` that `utils.py` had previously provided have been moved to new locations.

If you are using the above functions in your code and want to migrate to the new API, you should replace the following statements in your code:

```
from rsmtool.analysis import metrics_helper
metrics_helper(...)

from rsmtool.report import convert_ipynb_to_html
convert_ipynb_to_html(...)

from rsmtool.preprocess import remove_outliers
remove_outliers(...)

from rsmtool.utils import agreement
agreement(...)

from rsmtool.utils import difference_of_standardized_means
difference_of_standardized_means(...)

from rsmtool.utils import partial_correlations
partial_correlations(...)

from rsmtool.utils import quadratic_weighted_kappa
quadratic_weighted_kappa(...)

from rsmtool.utils import standardized_mean_difference
```

(continues on next page)

(continued from previous page)

```
standardized_mean_difference(...)

from rsmtool.utils import parse_json_with_comments
parse_json_with_comments(...)

from rsmtool.utils import get_thumbnail_as_html
get_thumbnail_as_html(...)

from rsmtool.utils import show_thumbnail
show_thumbnail(...)
```

with the following, respectively:

```
from rsmtool.analyzer import Analyzer
Analyzer.metrics_helper(...)

from rsmtool.reporter import Reporter
Reporter.convert_ipynb_to_html(...)

from rsmtool.preprocessor import FeaturePreprocessor
FeaturePreprocessor.remove_outliers(...)

from rsmtool.utils.metrics import agreement
agreement(...)

from rsmtool.utils.metrics import difference_of_standardized_means
difference_of_standardized_means(...)

from rsmtool.utils.metrics import partial_correlations
partial_correlations(...)

from rsmtool.utils.metrics import quadratic_weighted_kappa
quadratic_weighted_kappa(...)

from rsmtool.utils.metrics import standardized_mean_difference
standardized_mean_difference(...)

from rsmtool.utils.files import parse_json_with_comments
parse_json_with_comments(...)

from rsmtool.utils.notebook import get_thumbnail_as_html
get_thumbnail_as_html(...)

from rsmtool.utils.notebook import show_thumbnail
show_thumbnail(...)
```

Note: In RSMTool v8.0 the API for computing `PRMSE` has changed. See *rsmtool.utils.prmse*.

1.10.1 `rsmtool` Package

`rsmtool.`**`run_experiment`**(*config_file_or_obj_or_dict*, *output_dir*, *overwrite_output=False*, *logger=None*)

Run an rsmtool experiment using the given configuration.

Run rsmtool experiment using the given configuration file, object, or dictionary. All outputs are generated under `output_dir`. If `overwrite_output` is `True`, any existing output in `output_dir` is overwritten.

Parameters

- **config_file_or_obj_or_dict** (*str or pathlib.Path or dict or Configuration*) – Path to the experiment configuration file either a a string or as a `pathlib.Path` object. Users can also pass a `Configuration` object that is in memory or a Python dictionary with keys corresponding to fields in the configuration file. Given a configuration file, any relative paths in the configuration file will be interpreted relative to the location of the file. Given a `Configuration` object, relative paths will be interpreted relative to the `configdir` attribute, that _must_ be set. Given a dictionary, the reference path is set to the current directory.
- **output_dir** (*str*) – Path to the experiment output directory.
- **overwrite_output** (*bool, optional*) – If `True`, overwrite any existing output under `output_dir`. Defaults to `False`.
- **logger** (*logging object, optional*) – A logging object. If `None` is passed, get logger from `__name__`. Defaults to `None`.

Raises

- `FileNotFoundError` – If any of the files contained in `config_file_or_obj_or_dict` cannot be located.
- `IOError` – If `output_dir` already contains the output of a previous experiment and `overwrite_output` is `False`.
- `ValueError` – If the current configuration specifies a non-linear model but `output_dir` already contains the output of a previous experiment that used a linear model with the same experiment ID.

rsmtool.**run_evaluation**(*config_file_or_obj_or_dict*, *output_dir*, *overwrite_output=False*, *logger=None*)

Run an rsmeval experiment using the given configuration.

All outputs are generated under `output_dir`. If `overwrite_output` is `True`, any existing output in `output_dir` is overwritten.

Parameters

- **config_file_or_obj_or_dict** (*str or pathlib.Path or dict or Configuration*) – Path to the experiment configuration file either a a string or as a `pathlib.Path` object. Users can also pass a `Configuration` object that is in memory or a Python dictionary with keys corresponding to fields in the configuration file. Given a configuration file, any relative paths in the configuration file will be interpreted relative to the location of the file. Given a `Configuration` object, relative paths will be interpreted relative to the `configdir` attribute, that _must_ be set. Given a dictionary, the reference path is set to the current directory.
- **output_dir** (*str*) – Path to the experiment output directory.
- **overwrite_output** (*bool, optional*) – If `True`, overwrite any existing output under `output_dir`. Defaults to `False`.
- **logger** (*logging object, optional*) – A logging object. If `None` is passed, get logger from `__name__`. Defaults to `None`.

Raises

- `FileNotFoundError` – If any of the files contained in `config_file_or_obj_or_dict` cannot be located.
- `IOError` – If `output_dir` already contains the output of a previous experiment and `overwrite_output` is `False`.

`rsmtool.`**`run_comparison`**(*config_file_or_obj_or_dict*, *output_dir*)

Run an rsmcompare experiment using the given configuration.

Use the given configuration file, object, or dictionary and generate the report in the given directory.

Parameters

- **`config_file_or_obj_or_dict`** (*`str or pathlib.Path or dict or Configuration`*) – Path to the experiment configuration file either a a string or as a `pathlib.Path` object. Users can also pass a `Configuration` object that is in memory or a Python dictionary with keys corresponding to fields in the configuration file. Given a configuration file, any relative paths in the configuration file will be interpreted relative to the location of the file. Given a `Configuration` object, relative paths will be interpreted relative to the `configdir` attribute, that _must_ be set. Given a dictionary, the reference path is set to the current directory.
- **`output_dir`** (*`str`*) – Path to the experiment output directory.

Raises

- `FileNotFoundError` – If either of the two input directories in `config_file_or_obj_or_dict` do not exist.
- `FileNotFoundError` – If the directories do not contain rsmtool outputs at all.

`rsmtool.`**`run_summary`**(*config_file_or_obj_or_dict*, *output_dir*, *overwrite_output=False*, *logger=None*)

Run rsmsummarize experiment using the given configuration.

Summarize several rsmtool experiments using the given configuration file, object, or dictionary. All outputs are generated under `output_dir`. If `overwrite_output` is `True`, any existing output in `output_dir` is overwritten.

Parameters

- **`config_file_or_obj_or_dict`** (*`str or pathlib.Path or dict or Configuration`*) – Path to the experiment configuration file either a a string or as a `pathlib.Path` object. Users can also pass a `Configuration` object that is in memory or a Python dictionary with keys corresponding to fields in the configuration file. Given a configuration file, any relative paths in the configuration file will be interpreted relative to the location of the file. Given a `Configuration` object, relative paths will be interpreted relative to the `configdir` attribute, that _must_ be set. Given a dictionary, the reference path is set to the current directory.
- **`output_dir`** (*`str`*) – Path to the experiment output directory.
- **`overwrite_output`** (*`bool, optional`*) – If `True`, overwrite any existing output under `output_dir`. Defaults to `False`.
- **`logger`** (*`logging object, optional`*) – A logging object. If `None` is passed, get logger from `__name__`. Defaults to `None`.

Raises `IOError` – If `output_dir` already contains the output of a previous experiment and `overwrite_output` is `False`.

`rsmtool.`**`compute_and_save_predictions`**(*config_file_or_obj_or_dict*, *output_file*, *feats_file=None*, *logger=None*)

Run rsmpredict using the given configuration.

Generate predictions using given configuration file, object, or dictionary. Predictions are saved in `output_file`. Optionally, pre-processed feature values are saved in `feats_file`, if specified.

Parameters

- **config_file_or_obj_or_dict** (*str or pathlib.Path or dict or Configuration*) – Path to the experiment configuration file either a a string or as a `pathlib.Path` object. Users can also pass a `Configuration` object that is in memory or a Python dictionary with keys corresponding to fields in the configuration file. Given a configuration file, any relative paths in the configuration file will be interpreted relative to the location of the file. Given a `Configuration` object, relative paths will be interpreted relative to the `configdir` attribute, that _must_ be set. Given a dictionary, the reference path is set to the current directory.
- **output_file** (*str*) – The path to the output file.
- **feats_file** (*str, optional*) – Path to the output file for saving preprocessed feature values.
- **logger** (*logging object, optional*) – A logging object. If `None` is passed, get logger from `__name__`. Defaults to `None`.

Raises

- `FileNotFoundError` – If any of the files contained in `config_file_or_obj_or_dict` cannot be located.
- `FileNotFoundError` – If `experiment_dir` does not exist.
- `FileNotFoundError` – If `experiment_dir` does not contain the required output needed from an rsmtool experiment.
- `RuntimeError` – If the name of the output file does not end in ".csv", ".tsv", or ".xlsx".

From `analyzer` Module

Classes for analyzing RSMTool predictions, metrics, etc.

author Jeremy Biggs (jbiggs@ets.org)

author Anastassia Loukina (aloukina@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

class `rsmtool.analyzer.`**Analyzer**(*logger=None*)

Bases: `object`

Class to perform analysis on all metrics, predictions, etc.

static analyze_excluded_responses(*df*, *features*, *header*, *exclude_zero_scores=True*, *exclude_listwise=False*)

Compute statistics for responses excluded from analyses.

This method computes various statistics for the responses that were excluded from analyses, either in the training set or in the test set.

Parameters

- **df** (*pandas DataFrame*) – Data frame containing the excluded responses
- **features** (*list of str*) – List of column names containing the features to which we want to restrict the analyses.

- **header** (*str*) – String to be used as the table header for the output data frame.
- **exclude_zero_scores** (*bool, optional*) – Whether or not the zero-score responses should be counted in the exclusion statistics. Defaults to `True`.
- **exclude_listwise** (*bool, optional*) – Whether or not the candidates were excluded based on minimal number of responses. Defaults to `False`.

Returns **df_full_crosstab** – Two-dimensional data frame containing the exclusion statistics.

Return type pandas DataFrame

static analyze_used_predictions(*df_test*, *subgroups*, *candidate_column*)
Compute various statistics for predictions used in analyses.

Parameters

- **df_test** (*pandas DataFrame*) – Data frame containing the test set predictions.
- **subgroups** (*list of str*) – List of column names that contain grouping information.
- **candidate_column** (*str*) – Column name that contains candidate identification information.

Returns **df_analysis** – Data frame containing information about the used predictions.

Return type pandas DataFrame

static analyze_used_responses(*df_train*, *df_test*, *subgroups*, *candidate_column*)
Compute statistics for responses used in analyses.

This method computes various statistics on the responses that were used in analyses, either in the training set or in the test set.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the response information for the training set.
- **df_test** (*pandas DataFrame*) – Data frame containing the response information for the test set.
- **subgroups** (*list of str*) – List of column names that contain grouping information.
- **candidate_column** (*str*) – Column name that contains candidate identification information.

Returns **df_analysis** – Data frame containing information about the used responses.

Return type pandas DataFrame

static check_frame_names(*data_container*, *dataframe_names*)
Check that all specified dataframes are available.

This method checks to make sure all specified DataFrames are in the given data container object.

Parameters

- **data_container** (container.DataContainer) – A DataContainer object
- **dataframe_names** (*list of str*) – The names of the DataFrames expected in the DataContainer object.

Raises `KeyError` – If a given dataframe_name is not in the DataContainer object.

static check_param_names(*configuration_obj*, *parameter_names*)
Check that all specified parameters are available.

This method checks to make sure all specified parameters are in the given configuration object.

Parameters

- **configuration_obj** (configuration_parser.Configuration) – A configuration object
- **parameter_names** (*list of str*) – The names of the parameters (keys) expected in the Configuration object.

Raises KeyError – If a given parameter_name is not in the Configuration object.

static compute_basic_descriptives(*df*, *selected_features*)
Compute basic descriptive statistics for columns in the given data frame.

Parameters

- **df** (*pandas DataFrame*) – Input data frame containing the feature values.
- **selected_features** (*list of str*) – List of feature names for which to compute the descriptives.

Returns **df_desc** – DataFrame containing the descriptives for each of the features.

Return type pandas DataFrame

compute_correlations_by_group(*df*, *selected_features*, *target_variable*, *grouping_variable*, *include_length=False*)
Compute marginal and partial correlations against target variable.

This method computes various marginal and partial correlations of the given columns in the given data frame against the target variable for all data and for each level of the grouping variable.

Parameters

- **df** (*pandas DataFrame*) – Input data frame.
- **selected_features** (*list of str*) – List of feature names for which to compute the correlations.
- **target_variable** (*str*) – Feature name indicating the target variable i.e., the dependent variable
- **grouping_variable** (*str*) – Feature name that contain the grouping information
- **include_length** (*bool, optional*) – Whether or not to include the length when computing the partial correlations. Defaults to False.

Returns **df_output** – Data frame containing the correlations.

Return type pandas DataFrame

compute_degradation_and_disattenuated_correlations(*df*, *use_all_responses=True*)
Compute the degradation in performance when using system score.

This method computes the degradation in performance when using the system to predict the score instead of a second human and also the disattenuated correlations between human and system scores. These are computed as the Pearson's correlation between the human score and the system score divided by the square root of correlation between two human raters.

For this, we can compute the system performance either only on the double scored data or on the full dataset. Both options have their pros and cons. The default is to use the full dataset. This function also

assumes that the *sc2* column exists in the given data frame, in addition to *sc1* and the various types of predictions.

Parameters

- **df** (*pandas DataFrame*) – Input data frame.
- **use_all_responses** (*bool, optional*) – Use the full data set instead of only using the double-scored subset. Defaults to `True`.

Returns

- **df_degradation** (*pandas DataFrame*) – Data frame containing the degradation statistics.
- **df_correlations** (*pandas DataFrame*) – Data frame containing the human-system correlations, human-human correlations and disattenuated correlation.

static compute_disattenuated_correlations (*human_system_corr*, *human_human_corr*)

Compute disattenuated correlations between human and system scores.

These are computed as the Pearson's correlation between the human score and the system score divided by the square root of correlation between two human raters.

Parameters

- **human_system_corr** (*pandas Series*) – Series containing of pearson's correlation coefficients human-system correlations.
- **human_human_corr** (*pandas Series*) – Series containing of pearson's correlation coefficients for human-human correlations. This can contain a single value or have the index matching that of human-system correlations.

Returns df_correlations – Data frame containing the human-system correlations, human-human correlations, and disattenuated correlations.

Return type pandas DataFrame

compute_metrics (*df*, *compute_shortened=False*, *use_scaled_predictions=False*, *include_second_score=False*, *population_sd_dict=None*, *population_mn_dict=None*, *smd_method='unpooled'*, *use_diff_std_means=False*)

Compute association metrics for scores in the given data frame.

This function compute association metrics for all score types. If `include_second_score` is `True`, then it is assumed that a column called *sc2* containing a second human score is available and it should be used to compute the human-human evaluation stats and the performance degradation statistics.

If `compute_shortened` is `True`, then this function also computes a shortened version of the full human-system metrics data frame. See `filter_metrics()` for the description of the default columns included in the shortened data frame.

Parameters

- **df** (*pandas DataFrame*) – Input data frame
- **compute_shortened** (*bool, optional*) – Also compute a shortened version of the full metrics data frame. Defaults to `False`.
- **use_scaled_predictions** (*bool, optional*) – Use evaluations based on scaled predictions in the shortened version of the metrics data frame. Defaults to `False`.
- **include_second_score** (*bool, optional*) – Second human score available. Defaults to `False`.

- **population_sd_dict** (*dict, optional*) – Dictionary containing population standard deviation for each column containing human or system scores. This is used to compute SMD for subgroups. Defaults to `None`.
- **population_mn_dict** (*dict, optional*) – Dictionary containing population mean for each column containing human or system scores. This is used to compute SMD for subgroups. Defaults to `None`.
- **smd_method** (*{"williamson", "johnson", pooled", "unpooled"}, optional*) – The SMD method to use, only used if `use_diff_std_means` is `False`. All methods have the same numerator mean(*y_pred*) - mean(*y_true_observed*) and the following denominators:
  - "williamson": pooled population standard deviation of human and system scores computed based on values in `population_sd_dict`.
  - "johnson": population standard deviation of human scores computed based on values in `population_sd_dict`.
  - "pooled": pooled standard deviation of *y_true_observed* and *y_pred* for this group.
  - "unpooled": standard deviation of *y_true_observed* for this group.

  Defaults to "unpooled".
- **use_diff_std_means** (*bool, optional*) – Whether to use the difference of standardized means, rather than the standardized mean difference. This is most useful with subgroup analysis. Defaults to `False`.

Returns

- **df_human_system_eval** (*pandas DataFrame*) – Data frame containing the full set of evaluation metrics.
- **df_human_system_eval_filtered** (*pandas DataFrame*) – Data frame containing the human-human statistics but is empty if `include_second_score` is `False`.
- **df_human_human_eval** (*pandas DataFrame*) – A shortened version of the first data frame but is empty if `compute_shortened` is `False`.

compute_metrics_by_group(*df_test*, *grouping_variable*, *use_scaled_predictions=False*, *include_second_score=False*)

Compute a subset of evaluation metrics by subgroups.

This method computes a subset of evalution metrics for the scores in the given data frame by group specified in `grouping_variable`. See `filter_metrics()` above for a description of the subset that is selected.

Parameters

- **df_test** (*pandas DataFrame*) – Input data frame.
- **grouping_variable** (*str*) – Feature name indicating the column that contains grouping information.
- **use_scaled_predictions** (*bool, optional*) – Include scaled predictions when computing the evaluation metrics. Defaults to `False`.
- **include_second_score** (*bool, optional*) – Include human-human association statistics. Defaults to `False`.

Returns

- **df_human_system_by_group** (*pandas DataFrame*) – Data frame containing the correlation human-system association statistics.

- **df_human_human_by_group** (*pandas DataFrame*) – Data frame that either contains the human-human statistics or is an empty data frame, depending on whether `include_second_score` is *True*‘.

static compute_outliers(*df*, *selected_features*)

Compute number and percentage of outliers for given columns.

This method computes the number and percentage of outliers that lie outside the range mean +/- 4 SD for each of the given columns in the given data frame.

Parameters

- **df** (*pandas DataFrame*) – Input data frame containing the feature values.
- **selected_features** (*list of str*) – List of feature names for which to compute outlier information.

Returns **df_output** – Data frame containing outlier information for each of the features.

Return type pandas DataFrame

static compute_pca(*df*, *selected_features*)

Compute PCA decomposition of the given features.

This method computes the PCA decomposition of features in the data frame, restricted to the given columns. The number of components is set to be min(n_features, n_samples).

Parameters

- **df** (*pandas DataFrame*) – Input data frame containing feature values.
- **selected_features** (*list of str*) – List of feature names to be used in the PCA decomposition.

Returns

- **df_components** (*pandas DataFrame*) – Data frame containing the PCA components.
- **df_variance** (*pandas DataFrame*) – Data frame containing the variance information.

static compute_percentiles(*df*, *selected_features*, *percentiles=None*)

Compute percentiles and outliers for columns in the given data frame.

Parameters

- **df** (*pandas DataFrame*) – Input data frame containing the feature values.
- **selected_features** (*list of str*) – List of feature names for which to compute the percentile descriptives.
- **percentiles** (*list of ints, optional*) – The percentiles to calculate. If `None`, use the percentiles {1, 5, 25, 50, 75, 95, 99}. Defaults to `None`.

Returns **df_output** – Data frame containing the percentile information for each of the features.

Return type pandas DataFrame

static correlation_helper(*df*, *target_variable*, *grouping_variable*, *include_length=False*)

Compute marginal and partial correlations for all columns.

This helper method computes marginal and partial correlations of all the columns in the given data frame against the target variable separately for each level in the the grouping variable. If `include_length` is `True`, it additionally computes partial correlations of each column in the data frame against the target variable after controlling for the “length” column.

Parameters

- **df** (*pandas DataFrame*) – Input data frame containing numeric feature values, the numeric *target variable* and the *grouping variable*.
- **target_variable** (*str*) – The name of the column used as a reference for computing correlations.
- **grouping_variable** (*str*) – The name of the column defining groups in the data
- **include_length** (*bool, optional*) – If True compute additional partial correlations of each column in the data frame against *target variable* only partialling out "length" column.

Returns

- **df_target_cors** (*pandas DataFrame*) – Data frame containing Pearson's correlation coefficients for marginal correlations between features and *target_variable*.
- **df_target_partcors** (*pandas DataFrame*) – Data frame containing Pearson's correlation coefficients for partial correlations between each feature and *target_variable* after controlling for all other features. If `include_length` is set to `True`, the "length" column will not be included in the partial correlation computation.
- **df_target_partcors_no_length** (*pandas DataFrame*) – If `include_length` is set to `True`: Data frame containing Pearson's correlation coefficients for partial correlations between each feature and `target_variable` after controlling for "length". Otherwise, it will be an empty data frame.

filter_metrics (*df_metrics*, *use_scaled_predictions=False*, *chosen_metric_dict=None*)

Filter data frame to retain only the given metrics.

This method filters the data frame `df_metrics` – containing all of the metric values by all score types (raw, raw_trim etc.) – to retain only the metrics as defined in the given dictionary `chosen_metric_dict`. This dictionary maps score types ("raw", "scale", "raw_trim" etc.) to metric names. The available metric names are:

- "corr"
- "kappa"
- "wtkappa"
- "exact_agr"
- "adj_agr"
- "SMD" or "DSM", depending on what is in `df_metrics`.
- "RMSE"
- "R2"
- "sys_min"
- "sys_max"
- "sys_mean"
- "sys_sd"
- "h_min"
- "h_max"
- "h_mean"
- "h_sd"

- "N"

Parameters

- **df_metrics** (*pd.DataFrame*) – The DataFrame to filter.
- **use_scaled_predictions** (*bool, optional*) – Whether to use scaled predictions. Defaults to `False`.
- **chosen_metric_dict** (*dict, optional*) – The dictionary to map score types to metrics that should be computer for them. Defaults to `None`.

Note: The last five metrics will be the *same* for all score types. If `chosen_metric_dict` is not specified then, the following default dictionary, containing the recommended metrics, is used:

```
{"X_trim": ["N", "h_mean", "h_sd", "sys_mean", "sys_sd", "wtkappa",
              "corr", "RMSE", "R2", "SMD"],
 "X_trim_round": ["sys_mean", "sys_sd", "kappa",
                  "exact_agr", "adj_agr", "SMD"]}
```

where X = "raw" or "scale" depending on whether `use_scaled_predictions` is `False` or `True`, respectively.

---

static metrics_helper(*human_scores*, *system_scores*, *population_human_score_sd=None*, *population_system_score_sd=None*, *population_human_score_mn=None*, *population_system_score_mn=None*, *smd_method='unpooled'*, *use_diff_std_means=False*)

Compute basic association metrics between system and human scores.

Parameters

- **human_scores** (*pandas Series*) – Series containing numeric human (reference) scores.
- **system_scores** (*pandas Series*) – Series containing numeric scores predicted by the model.
- **population_human_score_sd** (*float, optional*) – Reference standard deviation for human scores. This must be specified when the function is used to compute association metrics for a subset of responses, for example, responses from a particular demographic subgroup. If `smd_method` is set to "williamson" or "johnson", this should be the standard deviation for the whole population (in most cases, the standard deviation for the whole test set). If `use_diff_std_means` is `True`, this must be the standard deviation for the whole population and `population_human_score_mn` must also be specified. Otherwise, it is ignored. Defaults to `None`.
- **population_system_score_sd** (*float, optional*) – Reference standard deviation for system scores. This must be specified when the function is used to compute association metrics for a subset of responses, for example, responses from a particular demographic subgroup. If `smd_method` is set to "williamson", this should be the standard deviation for the whole population (in most cases, the standard deviation for the whole test set). If `use_diff_std_means` is `True`, this must be the standard deviation for the whole population and `population_system_score_mn` must also be specified. Otherwise, it is ignored. Defaults to `None`.
- **population_human_score_mn** (*float, optional*) – Reference mean for human scores. This must be specified when the function is used to compute association metrics for a subset of responses, for example, responses from a particular demographic

subgroup. If `use_diff_std_means` is `True`, this must be the mean for the whole population (in most cases, the full test set) and `population_human_score_sd` must also be specified. Otherwise, it is ignored. Defaults to `None`.

- **`population_system_score_mn`** (*`float, optional`*) – Reference mean for system scores. This must be specified when the function is used to compute association metrics for a subset of responses, for example, responses from a particular demographic subgroup. If `use_diff_std_means` is `True`, this must be the mean for the whole population (in most cases, the full test set) and `population_system_score_sd` must also be specified. Otherwise, it is ignored. Defaults to `None`.
- **`smd_method`** (*`{"williamson", "johnson", "pooled", "unpooled"}, optional`*) – The SMD method to use, only used if `use_diff_std_means` is `False`. All methods have the same numerator mean(*y_pred*) - mean(*y_true_observed*) and the following denominators :
  - "williamson": pooled population standard deviation of *y_true_observed* and *y_pred* computed using `population_human_score_sd` and `population_system_score_sd`.
  - "johnson": `population_human_score_sd`.
  - "pooled": pooled standard deviation of *y_true_observed* and *y_pred* for this group.
  - "unpooled": standard deviation of *y_true_observed* for this group.

  Defaults to "unpooled".
- **`use_diff_std_means`** (*`bool, optional`*) – Whether to use the difference of standardized means, rather than the standardized mean difference. This is most useful with subgroup analysis. Defaults to `False`.

Returns

metrics – Series containing different evaluation metrics comparing human and system scores. The following metrics are included:

- *kappa*: unweighted Cohen's kappa
- *wtkappa*: quadratic weighted kappa
- *exact_agr*: exact agreement
- *adj_agr*: adjacent agreement with tolerance set to 1
- One of the following :
  - *SMD*: standardized mean difference, if `use_diff_std_means` is `False`.
  - *DSM*: difference of standardized means, if `use_diff_std_means` is `True`.
- *corr*: Pearson's r
- *R2*: r squared
- *RMSE*: root mean square error
- *sys_min*: min system score
- *sys_max*: max system score
- *sys_mean*: mean system score (ddof=1)
- *sys_sd*: standard deviation of system scores (ddof=1)
- *h_min*: min human score

- *h_max*: max human score
- *h_mean*: mean human score (ddof=1)
- *h_sd*: standard deviation of human scores (ddof=1)
- *N*: total number of responses

Return type pandas Series

run_data_composition_analyses_for_rsmeval (*data_container*, *configuration*)
Run all data composition analyses for RSMEval.

Parameters

- **data_container** (container.DataContainer) – The DataContainer object. This container must include the following DataFrames: {"test_metadata", "test_excluded"}.
- **configuration** (configuration_parser.Configuration) – The Configuration object. This configuration object must include the following parameters (keys): {"subgroups", "candidate_column", "exclude_zero_scores", "exclude_listwise"}.

Returns

- **data_container** (*container.DataContainer*) – A new DataContainer object with the following DataFrames:
  - test_excluded_composition
  - data_composition
  - data_composition_by_*
- **configuration** (*configuration_parser.Configuration*) – A new Configuration object.

run_data_composition_analyses_for_rsmtool (*data_container*, *configuration*)
Run all data composition analyses for RSMTool.

Parameters

- **data_container** (container.DataContainer) – The DataContainer object. This container must include the following DataFrames: {"test_metadata", "train_metadata","train_excluded", "test_excluded", "train_features"}.
- **configuration** (configuration_parser.Configuration) – The Configuration object. This configuration object must include the following parameters (keys): {"subgroups", "candidate_column", "exclude_zero_scores", "exclude_listwise"}.

Returns

- **data_container** (*container.DataContainer*) – A new DataContainer object with the following DataFrames:
  - test_excluded_composition
  - train_excluded_composition
  - data_composition
  - data_composition_by_*
- **configuration** (*configuration_parser.Configuration*) – A new Configuration object.

run_prediction_analyses (*data_container*, *configuration*)
Run all analyses on the system scores (predictions).

Parameters

- **data_container** (container.DataContainer) – The DataContainer object. This container must include the following DataFrames: {"train_features", "train_metadata", "train_preprocessed_features", "train_length", "train_features"}.
- **configuration** (configuration_parser.Configuration) – The Configuration object. This configuration object must include the following parameters (keys): {"subgroups", "second_human_score_column", "use_scaled_predictions"}.

Returns

- **data_container** (*container.DataContainer*) – A new DataContainer object with the following DataFrames:
  - eval
  - eval_short
  - consistency
  - degradation
  - disattenudated_correlations
  - confMatrix
  - score_dist
  - eval_by_*
  - consistency_by_*
  - disattenduated_correlations_by_*
  - true_score_eval
- **configuration** (*configuration_parser.Configuration*) – A new Configuration object.

run_training_analyses(*data_container*, *configuration*)

Run all analyses on the training data.

Parameters

- **data_container** (container.DataContainer) – The DataContainer object. This container must include the following DataFrames: {"train_features", "train_metadata", "train_preprocessed_features", "train_length", "train_features"}.
- **configuration** (configuration_parser.Configuration) – The Configuration object. This configuration object must include the following parameters (keys): {"length_column", "subgroups", "selected_features"}.

Returns

- **data_container** (*container.DataContainer*) – A new DataContainer object with the following DataFrames:
  - feature_descriptives
  - feature_descriptivesExtra
  - feature_outliers
  - cors_orig
  - cors_processed
  - margcor_score_all_data

- pcor_score_all_data
- pcor_score_no_length_all_data
- margcor_length_all_data
- pcor_length_all_data
- pca
- pcavar
- margcor_length_by_*
- pcor_length_by_*
- margcor_score_by_*
- pcor_score_by_*
- pcor_score_no_length_by_*

- **configuration** (*configuration_parser.Configuration*) – A new Configuration object.

From `comparer` Module

Classes for comparing outputs of two RSMTool experiments.

author Jeremy Biggs (jbiggs@ets.org)

author Anastassia Loukina (aloukina@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

class rsmtool.comparer.**Comparer**
Bases: object

Class to perform comparisons between two RSMTool experiments.

static compute_correlations_between_versions(*df_old*, *df_new*, *human_score='sc1'*, *id_column='spkitemid'*)

Compute correlations between old and new feature values.

This method computes correlations between old and new feature values in the two given frames as well as the correlations between each feature value and the human score.

Parameters

- **df_old** (*pandas DataFrame*) – Data frame with feature values for the 'old' model.
- **df_new** (*pandas DataFrame*) – Data frame with feature values for the 'new' model.
- **human_score** (*str, optional*) – Name of the column containing human score. Defaults to "sc1". Must be the same for both data sets.
- **id_column** (*str, optional*) – Name of the column containing id for each response. Defaults to "spkitemid". Must be the same for both data sets.

Returns

df_correlations –

Data frame with a row for each feature and the following columns:

- "N": total number of responses

- "human_old": correlation with human score in the old frame
- "human_new": correlation with human score in the new frame
- "old_new": correlation between old and new frames

Return type pandas DataFrame

Raises

- `ValueError` – If there are no shared features between the two sets.
- `ValueError` – If there are no shared responses between the two sets.

load_rsmtool_output (*filedir*, *figdir*, *experiment_id*, *prefix*, *groups_eval*)
Load all of the outputs of an rsmtool experiment.

For each type of output, we first check whether the file exists to allow comparing experiments with different sets of outputs.

Parameters

- **filedir** (*str*) – Path to the directory containing output files.
- **figdir** (*str*) – Path to the directory containing output figures.
- **experiment_id** (*str*) – Original `experiment_id` used to generate the output files.
- **prefix** (*str*) – Must be set to `scale` or `raw`. Indicates whether the score is scaled or not.
- **groups_eval** (*list*) – List of subgroup names used for subgroup evaluation.

Returns

- **files** (*dict*) – A dictionary with outputs converted to pandas data frames. If a particular type of output did not exist for the experiment, its value will be an empty data frame.
- **figs** (*dict*) – A dictionary with experiment figures.

static make_summary_stat_df (*df*)
Compute summary statistics for the data in the given frame.

Parameters **df** (*pandas DataFrame*) – Data frame containing numeric data.

Returns **res** – Data frame containing summary statistics for data in the input frame.

Return type pandas DataFrame

static process_confusion_matrix (*conf_matrix*)
Add "human" and "machine" to column names in the confusion matrix.

Parameters **conf_matrix** (*pandas DataFrame*) – data frame containing the confusion matrix.

Returns **conf_matrix_renamed** – pandas Data Frame containing the confusion matrix with the columns renamed.

Return type pandas DataFrame

From `configuration_parser` Module

Configuration parser.

Classes related to parsing configuration files and creating configuration objects.

author Jeremy Biggs (jbiggs@ets.org)

author Anastassia Loukina (aloukina@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

class rsmtool.configuration_parser.**Configuration**(*configdict*, *, *configdir=None*, *context='rsmtool'*, *logger=None*)

Bases: object

Configuration class.

Encapsulates all of the configuration parameters and methods to access these parameters.

Create an object of the *Configuration* class.

This method can be used to directly instantiate a Configuration object.

Parameters

- **configdict** (*dict*) – A dictionary of configuration parameters. The dictionary must be a valid configuration dictionary with default values filled as necessary.
- **configdir** (*str, optional, keyword-only*) – The reference path used to resolve any relative paths in the configuration object. When None, will be set during initialization to the current working directory. Defaults to None,
- **context** (*str*) – The context of the tool. One of {"rsmtool", "rsmeval", "rsmcompare", "rsmpredict", "rsmsummarize"}. Defaults to "rsmtool".
- **logger** (*logging object, optional*) – A logging object. If None is passed, get logger from __name__. Defaults to None.

check_exclude_listwise()

Check for candidate exclusion.

Check if we are excluding candidates based on number of responses, and add this to the configuration file.

Returns **exclude_listwise** – Whether to exclude list-wise.

Return type bool

check_flag_column(*flag_column='flag_column'*, *partition='unknown'*)

Make sure the column name in flag_column is correctly specified.

Get flag columns and values for filtering, if any, and convert single values to lists. Raises an exception if the column name in flag_column is not correctly specified in the configuration file.

Parameters

- **flag_column** (*str*) – The flag column name to check. Currently used names are "flag_column" or "flag_column_test". Defaults to "flag_column".
- **partition** (*str*) – The data partition which is filtered based on the flag column name. One of {"train", "test", "both", "unknown"}. Defaults to "both".

Returns **new_filtering_dict** – Properly formatted dictionary for the column name in flag_column.

Return type dict

Raises

- ValueError – If the specified value of the column name in flag_column is not a dictionary.
- ValueError – If the value of partition is not in the expected list.

- `ValueError` – If the value of `partition` does not match the `flag_column`.

configdir
Get the path to configuration directory.

Get the path to the configuration reference directory that will be used to resolve any relative paths in the configuration.

Returns **configdir** – The path to the configuration reference directory.

Return type str

context
Get the context.

copy(*deep=True*)
Return a copy of the object.

Parameters **deep** (*bool, optional*) – Whether to perform a deep copy. Defaults to `True`.

Returns **copy** – A new configuration object.

Return type *Configuration*

get(*key*, *default=None*)
Get value or default for the given key.

Parameters

- **key** (*str*) – Key to check in the Configuration object.
- **optional** (*default,*) – The default value to return, if no key exists. Defaults to `None`.

Returns The value in the configuration object dictionary.

Return type value

get_default_converter()
Get default converter dictionary for data reader.

Returns **default_converter** – The default converter for a train or test file.

Return type dict

get_names_and_paths(*keys*, *names*)
Get a list of values for the given keys.

Remove any values that are `None`.

Parameters

- **keys** (*list*) – A list of keys whose values to retrieve.
- **names** (*list*) – The default value to use if key cannot be found. Defaults to `None`.

Returns **values** – The list of values.

Return type list

Raises `ValueError` – If there are any duplicate keys or names.

get_rater_error_variance()
Get specified rater error variance, if any, and make sure it's numeric.

Returns **rater_error_variance** – Specified rater error variance.

Return type float

get_trim_min_max_tolerance()
Get trim min, trim max, and tolerance values.

Get the specified trim min and max, and trim_tolerance if any, and make sure they are numeric.

Returns

- **spec_trim_min** (*float*) – Specified trim min value.
- **spec_trim_max** (*float*) – Specified trim max value.
- **spec_trim_tolerance** (*float*) – Specified trim tolerance value.

items()
Return configuration items as a list of tuples.

Returns **items** – A list of (key, value) tuples in the configuration object.

Return type list of tuples

keys()
Return keys as a list.

Returns **keys** – A list of keys in the configuration object.

Return type list of str

pop(*key*, *default=None*)
Remove and return an element from the object having the given key.

Parameters

- **key** (*str*) – Key to pop in the configuration object.
- **optional** (*default,*) – The default value to return, if no key exists. Defaults to `None`.

Returns The value removed from the object.

Return type value

save(*output_dir=None*)
Save the configuration file to the output directory specified.

Parameters **output_dir** (*str*) – The path to the output directory. If `None`, the current directory is used. Defaults to `None`.

to_dict()
Get a dictionary representation of the configuration object.

Returns **config** – The configuration dictionary.

Return type dict

values()
Return configuration values as a list.

Returns **values** – A list of values in the configuration object.

Return type list

class rsmtool.configuration_parser.**ConfigurationParser**(*pathlike*, *logger=None*)
Bases: `object`

`ConfigurationParser` class to create `Configuration` objects.

Instantiate a `ConfigurationParser` for a given config file path.

Parameters

- **pathlike** (*str or pathlib.Path*) – A string containing the path to the configuration file that is to be parsed. A `pathlib.Path` instance is also acceptable.
- **logger** (*logging.Logger object, optional*) – Custom logger object to use, if not `None`. Otherwise a new logger is created. Defaults to `None`.

Raises

- `FileNotFoundError` – If the given path does not exist.
- `OSError` – If the given path is a directory, not a file.
- `ValueError` – If the file at the given path does not have a valid extension (".json").

logger = None

parse(*context='rsmtool'*)

Parse configuration file.

Parse the configuration file for which this parser was instantiated.

Parameters **context** (*str, optional*) – Context of the tool in which we are validating. One of: {"rsmtool", "rsmeval", "rsmpredict", "rsmcompare", "rsmsummarize", "rsmxval"}. Defaults to "rsmtool".

Returns **configuration** – A configuration object containing the parameters in the file that we instantiated the parser for.

Return type *Configuration*

classmethod process_config(*config*)

Process configuration file.

Converts fields which are read in as string to the appropriate format. Fields which can take multiple string values are converted to lists if they have not been already formatted as such.

Parameters **inplace** (*bool*) – Maintain the state of the config object produced by this method. Defaults to `True`.

Returns **config_obj** – A configuration object

Return type *Configuration*

Raises

- `NameError` – If `config` does not exist, or `config` could not be read.
- `ValueError` – If boolean configuration fields contain a value other then "true" or "false" (in JSON).

classmethod validate_config(*config*, *context='rsmtool'*)

Validate configuration file.

Ensure that all required fields are specified, add default values values for all unspecified fields, and ensure that all specified fields are valid.

Parameters

- **context** (*str, optional*) – Context of the tool in which we are validating. One of {"rsmtool", "rsmeval", "rsmpredict", "rsmcompare", "rsmsummarize"}. Defaults to "rsmtool".
- **inplace** (*bool*) – Maintain the state of the config object produced by this method. Defaults to `True`.

Returns **config_obj** – A configuration object

Return type *Configuration*

Raises `ValueError` – If config does not exist, and no config passed.

`rsmtool.configuration_parser.`**`configure`**(*context*, *config_file_or_obj_or_dict*)

Create a Configuration object.

Get the configuration for `context` from the input `config_file_or_obj_or_dict`.

Parameters

- **`context`** (*`str`*) – The context that is being configured. Must be one of "rsmtool", "rsmeval", "rsmcompare", "rsmsummarize", "rsmpredict", or "rsmxval".
- **`config_file_or_obj_or_dict`** (*`str or pathlib.Path or dict or Configuration`*) – Path to the experiment configuration file either a a string or as a `pathlib.Path` object. Users can also pass a `Configuration` object that is in memory or a Python dictionary with keys corresponding to fields in the configuration file. Given a configuration file, any relative paths in the configuration file will be interpreted relative to the location of the file. Given a `Configuration` object, relative paths will be interpreted relative to the `configdir` attribute, that _must_ be set. Given a dictionary, the reference path is set to the current directory.

Returns **configuration** – The Configuration object for the tool.

Return type *Configuration*

Raises

- `AttributeError` – If the `configdir` attribute for the Configuration input is not set.
- `ValueError` – If `config_file_or_obj_or_dict` contains anything except a string, a path, a dictionary, or a `Configuration` object.

From `container` Module

Class to encapsulate data contained in multiple pandas DataFrames.

It represents each of the multiple data sources as a "dataset". Each dataset is represented by three properties: - "name" : the name of the data set - "frame" : the pandas DataFrame that contains the actual data - "path" : the path to the file on disk from which the data was read

author Jeremy Biggs (jbiggs@ets.org)

author Anastassia Loukina (aloukina@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

`class` `rsmtool.container.`**`DataContainer`**(*datasets=None*)

Bases: `object`

Class to encapsulate datasets.

Initialize a DataContainer object.

Parameters **`datasets`** (*`list of dicts, optional`*) – A list of dataset dictionaries. Each dict should have the following keys: "name" containing the name of the dataset, "frame" containing the dataframe object representing the dataset, and "path" containing the path to the file from which the frame was read.

`add_dataset`(*dataset_dict*, *update=False*)

Add a new dataset (or update an existing one).

Parameters

- **dataset_dict** (*dict*) – The dataset dictionary to add or update with the "name", "frame", and "path" keys.
- **update** (*bool, optional*) – Update an existing DataFrame, if `True`. Defaults to `False`.

copy (*deep=True*)
Return a copy of the container object.

Parameters **deep** (*bool, optional*) – If `True`, create a deep copy of the underlying data frames. Defaults to `True`.

drop (*name*)
Drop a given dataset from the container and return instance.

Parameters **name** (*str*) – The name of the dataset to drop.

Returns

Return type self

get_frame (*name*, *default=None*)
Get the data frame given the dataset name.

Parameters

- **name** (*str*) – The name for the dataset.
- **default** (*pandas DataFrame, optional*) – The default value to return if the named dataset does not exist. Defaults to `None`.

Returns **frame** – The data frame for the named dataset.

Return type pandas DataFrame

get_frames (*prefix=None*, *suffix=None*)
Get all data frames with a given prefix or suffix in their name.

Note that the selection by prefix or suffix is case-insensitive.

Parameters

- **prefix** (*str, optional*) – Only return frames with the given prefix. If `None`, then do not exclude any frames based on their prefix. Defaults to `None`.
- **suffix** (*str, optional*) – Only return frames with the given suffix. If `None`, then do not exclude any frames based on their suffix. Defaults to `None`.

Returns **frames** – A dictionary with the data frames that contain the specified prefix and/or suffix in their corresponding names. The names are the keys and the frames are the values.

Return type dict

get_path (*name*, *default=None*)
Get the path for the dataset given the name.

Parameters

- **name** (*str*) – The name for the dataset.
- **default** (*str, optional*) – The default path to return if the named dataset does not exist. Defaults to `None`.

Returns **path** – The path for the named dataset.

Return type str

items()
Return the container items as a list of (name, frame) tuples.

Returns **items** – A list of (name, frame) tuples in the container object.

Return type list of tuples

keys()
Return the container keys (dataset names) as a list.

Returns **keys** – A list of keys (names) in the container object.

Return type list

rename(*name*, *new_name*)
Rename a given dataset in the container and return instance.

Parameters

- **name** (*str*) – The name of the current dataset in the container object.
- **new_name** (*str*) – The new name for the dataset in the container object.

Returns

Return type self

static to_datasets(*data_container*)
Convert container object to a list of dataset dictionaries.

Each dictionary will contain the "name", "frame", and "path" keys.

Parameters **data_container** (DataContainer) – The container object to convert.

Returns **datasets_dict** – A list of dataset dictionaries.

Return type list of dicts

values()
Return all data frames as a list.

Returns **values** – A list of all data frames in the container object.

Return type list

From convert_feature_json Module

rsmtool.convert_feature_json.**convert_feature_json_file**(*json_file*, *output_file*, *delete=False*)
Convert given feature JSON file into tabular format.

The specific format is inferred by the extension of the output file.

Parameters

- **json_file** (*str*) – Path to feature JSON file to be converted.
- **output_file** (*str*) – Path to CSV/TSV/XLSX output file.
- **delete** (*bool, optional*) – Whether to delete the original file after conversion. Defaults to False.

Raises

- RuntimeError – If the given input file is not a valid feature JSON file.

- `RuntimeError` – If the output file has an unsupported extension.

From `fairness_utils` Module

`rsmtool.fairness_utils.`**`get_fairness_analyses`**(*df*, *group*, *system_score_column*, *human_score_column='sc1'*, *base_group=None*)

Compute fairness analyses described in Loukina et al. 2019.

The function computes how much variance group membership explains in overall score accuracy (osa), overall score difference (osd), and conditional score difference (csd). See the paper for more details.

Parameters

- **`df`** (*pandas DataFrame*) – A dataframe containing columns with numeric human scores, columns with numeric system scores and a column with group membership.
- **`group`** (*str*) – Name of the column containing group membership.
- **`system_score_column`** (*str*) – Name of the column containing system scores.
- **`human_score_column`** (*str*) – Name of the column containing human scores.
- **`base_group`** (*str, optional*) – Name of the group to use as the reference category. Defaults to `None` in which case the group with the largest number of cases will be used as the reference category. Ties are broken alphabetically.

Returns

- **model_dict** (*dictionary*) – A dictionary with different proposed metrics as keys and fitted models as values.
- **fairness_container** (*DataContainer*) –

  A datacontainer with the following datasets:

  – "estimates_<METRIC>_by_<GROUP>" where "<GROUP>" corresponds to the given group and "<METRIC>" can be "osa", "osd" and "csd" estimates for each group computed by the respective models.

  – "fairness_metrics_by_<GROUP>" - a summary of model fits (R2 and p values).

From `modeler` Module

Class for training and predicting with built-in or SKLL models.

author Jeremy Biggs (jbiggs@ets.org)

author Anastassia Loukina (aloukina@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

class `rsmtool.modeler.`**`Modeler`**(*logger=None*)

Bases: `object`

Class to train model and generate predictions with built-in or SKLL models.

Instantiate empty instance with no learner and given logger, if any.

`create_fake_skll_learner`(*df_coefficients*)

Create a fake SKLL linear regression learner from given coefficients.

Parameters **df_coefficients** (*pandas DataFrame*) – The data frame containing the linear coefficients we want to create the fake SKLL model with.

Returns **learner** – SKLL Learner object representing a `LinearRegression` model with the specified coefficients.

Return type skll.learner.Learner

get_coefficients()
Get the coefficients of the model, if available.

Returns **coefficients** – The coefficients of the model, if available.

Return type np.array or `None`

get_feature_names()
Get the feature names, if available.

Returns **feature_names** – A list of feature names, or None if no learner was trained.

Return type list or `None`

get_intercept()
Get the intercept of the model, if available.

Returns **intercept** – The intercept of the model.

Return type float or `None`

classmethod load_from_file(*model_path*)
Load a modeler object from a file on disk.

Parameters **model_path** (*str*) – The path to the modeler file.

Returns **model** – A Modeler instance.

Return type *Modeler*

Raises `ValueError` – If `model_path` does not end with ".model".

classmethod load_from_learner(*learner*)
Create a new Modeler instance with a pre-populated learner.

Parameters **learner** (*skll.learner.Learner*) – A SKLL Learner object.

Returns **modeler** – A Modeler object.

Return type *Modeler*

Raises `TypeError` – If `learner` is not a SKLL Learner instance.

static model_fit_to_dataframe(*fit*)
Extract fit metrics from a `statsmodels` fit object into a data frame.

Parameters **fit** (*statsmodels.RegressionResults*) – Model fit object obtained from a linear model trained using `statsmodels.OLS`.

Returns **df_fit** – The output data frame with the main model fit metrics.

Return type pandas DataFrame

static ols_coefficients_to_dataframe(*coefs*)
Convert series containing OLS coefficients to a data frame.

Parameters **coefs** (*pandas Series*) – Series with feature names in the index and the coefficient values as the data, obtained from a linear model trained using `statsmodels.OLS`.

Returns **df_coef** – Data frame with two columns: the feature name and the coefficient value.

Return type pandas DataFrame

Note: The first row in the output data frame is always for the intercept and the rest are sorted by feature name.

---

predict(*df*, *min_score*, *max_score*, *predict_expected=False*)

Get raw predictions from given SKLL model on data in given data frame.

Parameters

- **df** (*pandas DataFrame*) – Data frame containing features on which to make the predictions. The data must contain pre-processed feature values, an ID column named "spkitemid", and a label column named "sc1".
- **min_score** (*int*) – Minimum score level to be used if computing expected scores.
- **max_score** (*int*) – Maximum score level to be used if computing expected scores.
- **predict_expected** (*bool, optional*) – Predict expected scores for classifiers that return probability distributions over score. This will be ignored with a warning if the specified model does not support probability distributions. Note also that this assumes that the score range consists of contiguous integers - starting at `min_score` and ending at `max_score`. Defaults to `False`.

Returns **df_predictions** – Data frame containing the raw predictions, the IDs, and the human scores.

Return type pandas DataFrame

Raises

- `ValueError` – If the model cannot predict probability distributions and `predict_expected` is set to `True`.
- `ValueError` – If the score range specified by `min_score` and `max_score` does not match what the model predicts in its probability distribution.

predict_train_and_test(*df_train*, *df_test*, *configuration*)

Generate raw, scaled, and trimmed predictions on given data.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the pre-processed training set features.
- **df_test** (*pandas DataFrame*) – Data frame containing the pre-processed test set features.
- **configuration** (configuration_parser.Configuration) – A configuration object containing "trim_max" and "trim_min" keys.

Returns

- *List of data frames containing predictions and other*
- *information.*

scale_coefficients(*configuration*)

Scale coefficients using human scores & training set predictions.

This procedure approximates what is done in operational setting but does not apply trimming to predictions.

Parameters **configuration** (configuration_parser.Configuration) – A configuration object containing the "train_predictions_mean", "train_predictions_sd", and "human_labels_sd" parameters.

Returns **data_container** – A container object containing the "coefficients_scaled" dataset. The frame for this dataset contains the scaled coefficients and the feature names, along with the intercept.

Return type *container.DataContainer*

Raises RuntimeError – If the model is non-linear and no coefficients are available.

static skll_learner_params_to_dataframe(*learner*)

Extract parameters from the given SKLL learner into a data frame.

Parameters **learner** (*skll.learner.Learner*) – A SKLL learner object.

Returns **df_coef** – The data frame containing the model parameters from the given SKLL learner object.

Return type pandas DataFrame

Note:

1. We use the coef_ attribute of the scikit-learn model underlying the SKLL learner instead of the latter's model_params attribute. This is because model_params ignores zero coefficients, which we do not want.
2. The first row in the output data frame is always for the intercept and the rest are sorted by feature name.

---

train(*configuration*, *data_container*, *filedir*, *figdir*, *file_format='csv'*)

Train the given model on the given data and save the results.

The main driver function to train the given model on the given data and save the results in the given directories using the given experiment ID as the prefix.

Parameters

- **configuration** (configuration_parser.Configuration) – A configuration object containing "experiment_id" and "model_name" parameters.
- **data_container** (container.DataContainer) – A data container object containing "train_preprocessed_features" data set.
- **filedir** (*str*) – Path to the "output" experiment output directory.
- **figdir** (*str*) – Path to the "figure" experiment output directory.
- **file_format** (*str, optional*) – The format in which to save files. One of {"csv", "tsv", "xlsx"}. Defaults to "csv".

Returns **name**

Return type SKLL Learner object

train_builtin_model(*model_name*, *df_train*, *experiment_id*, *filedir*, *figdir*, *file_format='csv'*)

Train one of the *built-in linear regression models*.

Parameters

- **model_name** (*str*) – Name of the built-in model to train.

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model. The data frame must contain the ID column named "spkitemid" and the numeric label column named "sc1".
- **experiment_id** (*str*) – The experiment ID.
- **filedir** (*str*) – Path to the *output* experiment output directory.
- **figdir** (*str*) – Path to the *figure* experiment output directory.
- **file_format** (*str, optional*) – The format in which to save files. One of {"csv", "tsv", "xlsx"}. Defaults to "csv".

Returns learner – SKLL `LinearRegression` Learner object containing the coefficients learned by training the built-in model.

Return type skll.learner.Learner

train_equal_weights_lr (*df_train*, *feature_columns*)
Train an "EqualWeightsLR" model.

This model assigns the same weight to all features.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.
- **feature_columns** (*list of str*) – A list of feature columns to use in training the model.

Returns

- **learner** (*skll.learner.Learner*) – The SKLL learner object.
- **fit** (*statsmodels.RegressionResults*) – A `statsmodels` regression results object.
- **df_coef** (*pandas DataFrame*) – Data frame containing the model coefficients.
- **used_features** (*list of str*) – A list of features used in the final model.

train_lasso_fixed_lambda (*df_train*, *feature_columns*)
Train a "LassoFixedLambda" model.

This is a Lasso model with a fixed lambda.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.
- **feature_columns** (*list of str*) – A list of feature columns to use in training the model.

Returns

- **learner** (*skll.learner.Learner*) – The SKLL learner object
- **fit** (`None`) – This is always `None` since there is no OLS model fitted in this case.
- **df_coef** (*pandas DataFrame*) – Data frame containing the model coefficients.
- **used_features** (*list of str*) – A list of features used in the final model.

train_lasso_fixed_lambda_then_lr (*df_train*, *feature_columns*)
Train a "LassoFixedLambdaThenLR" model.

First do feature selection using lasso regression with a fixed lambda and then use only those features to train a second linear regression

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.
- **feature_columns** (*list of str*) – A list of feature columns to use in training the model.

Returns

- **learner** (*skll.learner.Learner*) – The SKLL learner object
- **fit** (*statsmodels.RegressionResults*) – A `statsmodels` regression results object.
- **df_coef** (*pandas DataFrame*) – The model coefficients in a data_frame
- **used_features** (*list of str*) – A list of features used in the final model.

train_lasso_fixed_lambda_then_non_negative_lr (*df_train*, *feature_columns*)
Train an "LassoFixedLambdaThenNNLR" model.

First do feature selection using lasso regression and positive only weights. Then fit an NNLR (see above) on those features.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.
- **feature_columns** (*list of str*) – A list of feature columns to use in training the model.

Returns

- **learner** (*skll.learner.Learner*) – The SKLL learner object.
- **fit** (*statsmodels.RegressionResults*) – A statsmodels regression results object.
- **df_coef** (*pandas DataFrame*) – Data frame containing the model coefficients.
- **used_features** (*list of str*) – A list of features used in the final model.

train_linear_regression (*df_train*, *feature_columns*)
Train a "LinearRegression" model.

This model is a simple linear regression model.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.
- **feature_columns** (*list*) – A list of feature columns to use in training the model.

Returns

- **learner** (*skll.learner.Learner*) – The SKLL learner object.
- **fit** (*statsmodels.RegressionResults*) – A `statsmodels` regression results object.
- **df_coef** (*pandas DataFrame*) – Data frame containing the model coefficients.
- **used_features** (*list of str*) – A list of features used in the final model.

train_non_negative_lr (*df_train*, *feature_columns*)
Train an "NNLR" model.

To do this, we first do feature selection using non-negative least squares (NNLS) and then use only its non-zero features to train another linear regression (LR) model. We do the regular LR at the end since we want an LR object so that we have access to R^2 and other useful statistics. There should be no difference between the non-zero coefficients from NNLS and the coefficients that end up coming out of the subsequent LR.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.
- **feature_columns** (*list of str*) – A list of feature columns to use in training the model.

Returns

- **learner** (*skll.learner.Learner*) – The SKLL learner object.
- **fit** (*statsmodels.RegressionResults*) – A statsmodels regression results object.
- **df_coef** (*pandas DataFrame*) – Data frame containing the model coefficients.
- **used_features** (*list of str*) – A list of features used in the final model.

train_non_negative_lr_iterative (*df_train*, *feature_columns*)
Train an "NNLR_iterative" model.

For applications where there is a concern that standard NNLS may not converge, an alternate method of training NNLR by iteratively fitting OLS models, checking the coefficients, and dropping negative coefficients. First, fit an OLS model. Then, identify any variables whose coefficients are negative. Drop these variables from the model. Finally, refit the model. If any coefficients are still negative, set these to zero.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.
- **feature_columns** (*list of str*) – A list of feature columns to use in training the model.

Returns

- **learner** (*skll.learner.Learner*) – The SKLL learner object.
- **fit** (*statsmodels.RegressionResults*) – A statsmodels regression results object.
- **df_coef** (*pandas DataFrame*) – Data frame containing the model coefficients.
- **used_features** (*list of str*) – A list of features used in the final model.

train_positive_lasso_cv (*df_train*, *feature_columns*)
Train a "PositiveLassoCV" model.

Do feature selection using lasso regression optimized for log likelihood using cross validation. All coefficients are constrained to have positive values.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.

- **feature_columns** (*list of str*) – A list of feature columns to use in training the model.

Returns

- **learner** (*skll.learner.Learner*) – The SKLL learner object
- **fit** (`None`) – This is always `None` since there is no OLS model fitted in this case.
- **df_coef** (*pandas DataFrame*) – Data frame containing the model coefficients.
- **used_features** (*list of str*) – A list of features used in the final model.

train_positive_lasso_cv_then_lr (*df_train*, *feature_columns*)
Train a "PositiveLassoCVThenLR" model.

First do feature selection using lasso regression optimized for log likelihood using cross validation and then use only those features to train a second linear regression.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.
- **feature_columns** (*list of str*) – A list of feature columns to use in training the model.

Returns

- **learner** (*skll.learner.Learner*) – The SKLL learner object.
- **fit** (*statsmodels.RegressionResults*) – A statsmodels regression results object.
- **df_coef** (*pandas DataFrame*) – Data frame containing the model coefficients.
- **used_features** (*list of str*) – A list of features used in the final model.

train_rebalanced_lr (*df_train*, *feature_columns*)
Train a "RebalancedLR" model.

This model balances empirical weights by changing betas (adapted from here).

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.
- **feature_columns** (*list of str*) – A list of feature columns to use in training the model.

Returns

- **learner** (*skll.learner.Learner*) – The SKLL learner object.
- **fit** (*statsmodels.RegressionResults*) – A `statsmodels` regression results object.
- **df_coef** (*pandas DataFrame*) – Data frame containing the model coefficients.
- **used_features** (*list of str*) – A list of features used in the final model.

train_score_weighted_lr (*df_train*, *feature_columns*)
Train a "ScoreWeightedLR" model.

This is a linear regression model weighted by score.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.
- **feature_columns** (*list of str*) – A list of feature columns to use in training the model.

Returns

- **learner** (*skll.learner.Learner*) – The SKLL learner object
- **fit** (*statsmodels.RegressionResults*) – A statsmodels regression results object.
- **df_coef** (*pandas DataFrame*) – Data frame containing the model coefficients.
- **used_features** (*list of str*) – A list of features used in the final model.

train_skll_model (*model_name*, *df_train*, *experiment_id*, *filedir*, *figdir*, *file_format='csv'*, *custom_fixed_parameters=None*, *custom_objective=None*, *predict_expected_scores=False*)

Train a SKLL classification or regression model.

Parameters

- **model_name** (*str*) – Name of the SKLL model to train.
- **df_train** (*pandas DataFrame*) – Data frame containing the features on which to train the model.
- **experiment_id** (*str*) – The experiment ID.
- **filedir** (*str*) – Path to the "output" experiment output directory.
- **figdir** (*str*) – Path to the "figure" experiment output directory.
- **file_format** (*str, optional*) – The format in which to save files. For SKLL models, this argument does not actually change the format of the output files at this time, as no betas are computed. One of {"csv", "tsv", "xlsx"}. Defaults to "csv".
- **custom_fixed_parameters** (*dict, optional*) – A dictionary containing any fixed parameters for the SKLL model. Defaults to `None`.
- **custom_objective** (*str, optional*) – Name of custom user-specified objective. If not specified or `None`, "neg_mean_squared_error" is used as the objective. Defaults to `None`.
- **predict_expected_scores** (*bool, optional*) – Whether we want the trained classifiers to predict expected scores. Defaults to `False`.

Returns **learner_and_objective** – A 2-tuple containing a SKLL Learner object of the appropriate type and the chosen tuning objective.

Return type tuple

From `preprocessor` Module

Classes for preprocessing input data in various contexts.

author Jeremy Biggs (jbiggs@ets.org)

author Anastassia Loukina (aloukina@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

class rsmtool.preprocessor.**FeaturePreprocessor**(*logger=None*)
Bases: object

Class to preprocess features in training and testing sets.

check_model_name(*model_name*)
Check that the given model name is valid and determine its type.

Parameters **model_name** (*str*) – Name of the model.

Returns **model_type** – One of "BUILTIN" or "SKLL".

Return type str

Raises ValueError – If the model is not supported.

check_subgroups(*df*, *subgroups*)
Validate subgroup names in the given data.

Check that all subgroups, if specified, correspond to columns in the provided data frame, and replace all NaNs in subgroups values with 'No info' for later convenience.

Raises an exception if any specified subgroup columns are missing.

Parameters

- **df** (*pandas DataFrame*) – Input data frame with subgroups to check.
- **subgroups** (*list of str*) – List of column names that contain grouping information.

Returns **df** – Modified input data frame with NaNs replaced.

Return type pandas DataFrame

Raises KeyError – If the data does not contain columns for all specified subgroups.

filter_data(*df*, *label_column*, *id_column*, *length_column*, *second_human_score_column*, *candidate_column*, *requested_feature_names*, *reserved_column_names*, *given_trim_min*, *given_trim_max*, *flag_column_dict*, *subgroups*, *exclude_zero_scores=True*, *exclude_zero_sd=False*, *feature_subset_specs=None*, *feature_subset=None*, *min_candidate_items=None*, *use_fake_labels=False*)
Filter rows with zero/non-numeric values for label_column.

Check whether any features that are specifically requested in requested_feature_names are missing from the data. If no feature names are requested, the feature list is generated based on column names and subset information, if available. The function then excludes non-numeric values for any feature. It will also exclude zero scores if exclude_zero_scores is True. If the user requested to exclude candidates with less than min_candidate_items, such candidates are also excluded.

It also generates fake labels between 1 and 10 if use_fake_parameters is True. Finally, it renames the ID and label columns and splits the data into: (a) data frame with feature values and scores (b) data frame with information about subgroup and candidate (metadata) and (c) the data frame with all other columns.

Parameters

- **df** (*pandas DataFrame*) – The data frame to filter.
- **label_column** (*str*) – The label column in the data.
- **id_column** (*str*) – The ID column in the data.
- **length_column** (*str*) – The length column in the data.

- **second_human_score_column** (*str*) – The second human score column in the data.
- **candidate_column** (*str*) – The candidate column in the data.
- **requested_feature_names** (*list*) – A list of requested feature names.
- **reserved_column_names** (*list*) – A list of reserved column names.
- **given_trim_min** (*float*) – The minimum trim value.
- **given_trim_max** (*float*) – The maximum trim value.
- **flag_column_dict** (*dict*) – A dictionary of flag columns.
- **subgroups** (*list, optional*) – A list of subgroups, if any.
- **exclude_zero_scores** (*bool*) – Whether to exclude zero scores. Defaults to `True`.
- **exclude_zero_sd** (*bool, optional*) – Whether to exclude zero standard deviation. Defaults to `False`.
- **feature_subset_specs** (*pandas DataFrame, optional*) – The data frame containing the feature subset specifications. Defaults to `None`.
- **feature_subset** (*str, optional*) – The feature subset group (e.g. 'A'). Defaults to `None`.
- **min_candidate_items** (*int, optional*) – The minimum number of items needed to include candidate. Defaults to `None`.
- **use_fake_labels** (*bool, optional*) – Whether to use fake labels. Defaults to `False`.

Returns

- **df_filtered_features** (*pandas DataFrame*) – Data frame with filtered features.
- **df_filtered_metadata** (*pandas DataFrame*) – Data frame with filtered metadata.
- **df_filtered_other_columns** (*pandas DataFrame*) – Data frame with other columns filtered.
- **df_excluded** (*pandas DataFrame*) – Data frame with excluded records.
- **df_filtered_length** (*pandas DataFrame*) – Data frame with length column(s) filtered.
- **df_filtered_human_scores** (*pandas DataFrame*) – Data frame with human scores filtered.
- **df_responses_with_excluded_flags** (*pandas DataFrame*) – Data frame containing responses with excluded flags.
- **trim_min** (*float*) – The maximum trim value.
- **trim_max** (*float*) – The minimum trim value.
- **feature_names** (*list*) – A list of feature names.

filter_on_column (*df*, *column*, *id_column*, *exclude_zeros=False*, *exclude_zero_sd=False*)

Filter out rows containing non-numeric values.

Filter out the rows in the given data frame that contain non-numeric (or zero, if specified) values in the specified column. Additionally, it may exclude any columns if they have a standard deviation (σ) of 0.

Parameters

- **df** (*pandas DataFrame*) – The data frame containing the data to be filtered.

- **column** (*str*) – Name of the column from which to filter out values.
- **id_column** (*str*) – Name of the column containing the unique response IDs.
- **exclude_zeros** (*bool, optional*) – Whether to exclude responses containing zeros in the specified column. Defaults to `False`.
- **exclude_zero_sd** (*bool, optional*) – Whether to perform the additional filtering step of removing columns that have $\sigma = 0$. Defaults to `False`.

Returns

- **df_filtered** (*pandas DataFrame*) – Data frame containing the responses that were *not* filtered out.
- **df_excluded** (*pandas DataFrame*) – Data frame containing the non-numeric or zero responses that were filtered out.

Note: The columns with $\sigma = 0$ are removed from both output data frames, assuming `exclude_zero_scores` is `True`.

---

filter_on_flag_columns (*df*, *flag_column_dict*)

Filter based on specific flag columns.

Check that all flag_columns are present in the given data frame, convert these columns to strings and filter out the values which do not match the condition in `flag_column_dict`.

Parameters

- **df** (*pandas DataFrame*) – The DataFrame to filter on.
- **flag_column_dict** (*dict*) – Dictionary containing the flag column information.

Returns

- **df_responses_with_requested_flags** (*pandas DataFrame*) – Data frame containing the responses remaining after filtering using the specified flag columns.
- **df_responses_with_excluded_flags** (*pandas DataFrame*) – Data frame containing the responses filtered out using the specified flag columns.

Raises

- `KeyError` – If the columns listed in the dictionary are not actually present in the data frame.
- `ValueError` – If no responses remain after filtering based on the flag column information.

generate_feature_names (*df*, *reserved_column_names*, *feature_subset_specs*, *feature_subset*)

Generate feature names from column names in data frame.

This method also selects the specified subset of features.

Parameters

- **df** (*pandas DataFrame*) – The data frame from which to generate feature names.
- **reserved_column_names** (*list of str*) – Names of reserved columns.
- **feature_subset_specs** (*pandas DataFrame*) – Feature subset specifications.
- **feature_subset** (*str*) – Feature subset column.

Returns **feature_names** – A list of features names.

Return type list of str

preprocess_feature(*values*, *feature_name*, *feature_transform*, *feature_mean*, *feature_sd*, *exclude_zero_sd=False*, *raise_error=True*, *truncations=None*)

Remove outliers and transform the values in given numpy array.

Use the given outlier and transformation parameters.

Parameters

- **values** (*np.array*) – The feature values to preprocess.
- **feature_name** (*str*) – Name of the feature being pre-processed.
- **feature_transform** (*str*) – Name of the transformation function to apply.
- **feature_mean** (*float*) – Mean value to use for outlier detection instead of the mean of the given feature values.
- **feature_sd** (*float*) – Std. dev. value to use for outlier detection instead of the std. dev. of the given feature values.
- **exclude_zero_sd** (*bool, optional*) – Exclude the feature if it has zero standard deviation. Defaults to `False`.
- **raise_error** (*bool, optional*) – Raise an error if any of the transformations lead to "inf" values or may change the ranking of feature values. Defaults to `True`.
- **truncations** (*pandas DataFrame, optional*) – A set of pre-defined truncation values. Defaults to `None`.

Returns **transformed_feature** – Numpy array containing the transformed and clamped feature values.

Return type np.array

Raises `ValueError` – If the preprocessed feature values have zero standard deviation and `exclude_zero_sd` is set to `True`.

preprocess_features(*df_train*, *df_test*, *df_feature_specs*, *standardize_features=True*, *use_truncations=False*)

Preprocess features in given data using corresponding specifications.

Preprocess the feature values in the training and testing data frames whose specifications are contained in `df_feature_specs`. Also returns a third data frame containing the feature specifications and other information.

Parameters

- **df_train** (*pandas DataFrame*) – Data frame containing the raw feature values for the training set.
- **df_test** (*pandas DataFrame*) – Data frame containing the raw feature values for the test set.
- **df_feature_specs** (*pandas DataFrame*) – Data frame containing the various specifications from the feature file.
- **standardize_features** (*bool, optional*) – Whether to standardize the features. Defaults to `True`.
- **use_truncations** (*bool, optional*) – Whether we should use the truncation set for removing outliers. Defaults to `False`.

Returns

- **df_train_preprocessed** (*pandas DataFrame*) – Data frame with preprocessed training data.
- **df_test_preprocessed** (*pandas DataFrame*) – Data frame with preprocessed test data.
- **df_feature_info** (*pandas DataFrame*) – Data frame with feature information.

preprocess_new_data(*df_input*, *df_feature_info*, *standardize_features=True*)

Preprocess feature values using the parameters in `df_feature_info`.

For more details on what these preprocessing parameters are, see *documentation*.

Parameters

- **df_input** (*pandas DataFrame*) – Data frame with raw feature values that will be used to generate the scores. Each feature is stored in a separate column. Each row corresponds to one response. There should also be a column named "spkitemid" containing a unique ID for each response.
- **df_feature_info** (*pandas DataFrame*) – Data frame with preprocessing parameters in the following columns:
  - "feature" : the name of the feature; should match the feature names in `df_input`.
  - "sign" : 1 or -1. Indicates whether the feature value needs to be multiplied by -1.
  - "transform" : *transformation* that needs to be applied to this feature.
  - "train_mean", "train_sd" : mean and standard deviation for outlier truncation.
  - "train_transformed_mean", "train_transformed_sd" : mean and standard deviation for computing z-scores.
- **standardize_features** (*bool, optional*) – Whether the features should be standardized prior to prediction. Defaults to `True`.

Returns

- **df_features_preprocessed** (*pandas DataFrame*) – Data frame with processed feature values.
- **df_excluded** (*pandas DataFrame*) – Data frame with responses excluded from further analysis due to non-numeric feature values in the original file or after applying transformations. This data frame always contains the original feature values.

Raises

- `KeyError` – if some of the features specified in `df_feature_info` are not present in `df_input`.
- `ValueError` – If all responses have at least one non-numeric feature value and, therefore, no score can be generated for any of the responses.

process_data(*config_obj*, *data_container_obj*, *context='rsmtool'*)

Process and setup the data for an experiment in the given context.

Parameters

- **config_obj** (`configuration_parser.Configuration`) – A configuration object.
- **data_container_obj** (`container.DataContainer`) – A data container object.
- **context** (*str*) – The tool context: one of {"rsmtool", "rsmeval", "rsmpredict"}. Defaults to "rsmtool".

Returns

- **config_obj** (*configuration_parser.Configuration*) – A new configuration object.
- **data_container** (*container.DataContainer*) – A new data container object.

Raises `ValueError` – If the context is not one of {"rsmtool", "rsmeval", "rsmpredict"}.

process_data_rsmeval (*config_obj*, *data_container_obj*)

Set up rsmeval experiment by loading & preprocessing evaluation data.

This function takes a configuration object and a container object as input and returns the same types of objects as output after the loading, normalizing, and preprocessing.

Parameters

- **config_obj** (`configuration_parser.Configuration`) – A configuration object.
- **data_container_obj** (`container.DataContainer`) – A data container object.

Returns

- **config_obj** (*configuration_parser.Configuration*) – A new configuration object.
- **data_container** (*container.DataContainer*) – A new data container object.

Raises

- `KeyError` – If columns specified in the configuration do not exist in the predictions file.
- `ValueError` – If the columns containing the human scores and the system scores in the predictions file have the same name.
- `ValueError` – If the columns containing the first set of human scores and the second set of human scores in the predictions file have the same name.
- `ValueError` – If the predictions file contains the same response ID more than once.
- `ValueError` – No responses were left after filtering out zero or non-numeric values for the various columns.

process_data_rsmpredict (*config_obj*, *data_container_obj*)

Process data for rsmpredict experiments.

This function takes a configuration object and a container object as input and returns the same types of objects as output after the loading, normalizing, and preprocessing.

Parameters

- **config_obj** (`configuration_parser.Configuration`) – A configuration object.
- **data_container_obj** (`container.DataContainer`) – A data container object.

Returns

- **config_obj** (*configuration_parser.Configuration*) – A new configuration object.
- **data_congtainer** (*container.DataContainer*) – A new data container object.

Raises

- `KeyError` – If columns specified in the configuration do not exist in the data.
- `ValueError` – If data contains duplicate response IDs.

process_data_rsmtool(*config_obj*, *data_container_obj*)
Set up rsmtool experiment by loading & preprocessing train/test data.

This function takes a configuration object and a container object as input and returns the same types of objects as output after the loading, normalizing, and preprocessing.

Parameters

- **config_obj** (configuration_parser.Configuration) – A configuration object.
- **data_container_obj** (container.DataContainer) – A data container object.

Returns

- **config_obj** (*configuration_parser.Configuration*) – A new configuration object.
- **data_container** (*container.DataContainer*) – A new data container object.

Raises

- ValueError – If columns specified in the configuration do not exist in the data.
- ValueError – If the test label column and second human score columns have the same name.
- ValueError – If the length column is requested as a feature.
- ValueError – If the second human score column is requested as a feature.
- ValueError – If "use_truncations" was specified in the configuration, but no feature CSV file was found.

process_predictions(*df_test_predictions*, *train_predictions_mean*, *train_predictions_sd*, *human_labels_mean*, *human_labels_sd*, *trim_min*, *trim_max*, *trim_tolerance=0.4998*)
Process predictions to create scaled, trimmed and rounded predictions.

Parameters

- **df_test_predictions** (*pandas DataFrame*) – Data frame containing the test set predictions.
- **train_predictions_mean** (*float*) – The mean of the predictions on the training set.
- **train_predictions_sd** (*float*) – The std. dev. of the predictions on the training set.
- **human_labels_mean** (*float*) – The mean of the human scores used to train the model.
- **human_labels_sd** (*float*) – The std. dev. of the human scores used to train the model.
- **trim_min** (*float*) – The lowest score on the score point, used for trimming the raw regression predictions.
- **trim_max** (*float*) – The highest score on the score point, used for trimming the raw regression predictions.
- **trim_tolerance** (*float*) – Tolerance to be added to trim_max and substracted from trim_min. Defaults to 0.4998.

Returns **df_pred_processed** – Data frame containing the various trimmed and rounded predictions.

Return type pandas DataFrame

static remove_outliers(*values*, *mean=None*, *sd=None*, *sd_multiplier=4*)
Remove outliers from given array of values by clamping them.

Clamp any given values that are $\pm$ `sd_multiplier` (m) standard deviations (σ) away from the mean (μ). Use given `mean` and `sd` instead of computing σ and μ, if specified. The values are clamped to the interval:

$$[\mu - m * \sigma, \mu + m * \sigma]$$

Parameters

- **values** (*np.array*) – The values from which to remove outliers.
- **mean** (*int or float, optional*) – Use the given mean value when computing outliers instead of the mean from the data. Defaults to `None`.
- **sd** (*None, optional*) – Use the given std. dev. value when computing outliers instead of the std. dev. from the data. Defaults to `None`.
- **sd_multiplier** (*int, optional*) – Use the given multipler for the std. dev. when computing the outliers. Defaults to 4. Defaults to 4.

Returns **new_values** – Numpy array with the outliers clamped.

Return type np.array

remove_outliers_using_truncations(*values*, *feature_name*, *truncations*)
Remove outliers using pre-specified truncation groups.

This is different from `remove_outliers()` which calculates the outliers based on the training set.

Parameters

- **values** (*np.array*) – The values from which to remove outliers.
- **feature_name** (*str*) – Name of the feature whose outliers are being clamped.
- **truncations** (*pandas DataFrame*) – A data frame with truncation values. The features should be set as the index.

Returns **new_values** – Numpy array with the outliers clamped.

Return type numpy array

rename_default_columns(*df*, *requested_feature_names*, *id_column*, *first_human_score_column*, *second_human_score_column*, *length_column*, *system_score_column*, *candidate_column*)
Standardize column names and rename columns with reserved column names.

RSMTool reserves some column names for internal use, e.g., "sc1", "spkitemid" etc. If the given data already contains columns with these names, then they must be renamed to prevent conflict. This method renames such columns to "##NAME##", e.g., an existing column named "sc1" will be renamed to "##sc1##".

Parameters

- **df** (*pandas DataFrame*) – The data frame containing the columns to rename.
- **requested_feature_names** (*list of str*) – List of feature column names that we want to include in the scoring model.
- **id_column** (*str*) – Column name containing the response IDs.

- **first_human_score_column** (str or None.) – Column name containing the H1 scores. Should be None if no H1 scores are available.
- **second_human_score_column** (str or None) – Column name containing the H2 scores. Should be None if no H2 scores are available.
- **length_column** (str or None) – Column name containing response lengths. Should be None if lengths are not available.
- **system_score_column** (*str*) – Column name containing the score predicted by the system. This is only used for rsmeval.
- **candidate_column** (str or None) – Column name containing identifying information at the candidate level. Should be None if such information is not available.

Returns **df** – Modified input data frame with all the approximate re-namings.

Return type pandas DataFrame

select_candidates(*df*, *N*, *candidate_col='candidate'*)

Select candidates which have responses to N or more items.

Parameters

- **df** (*pandas DataFrame*) – The data frame from which to select candidates with N or more items.
- **N** (*int*) – Minimal number of items per candidate
- **candidate_col** (*str, optional*) – Name of the column which contains candidate ids. Defaults to "candidate".

Returns

- **df_included** (*pandas DataFrame*) – Data frame with responses from candidates with responses to N or more items.
- **df_excluded** (*pandas DataFrame*) – Data frame with responses from candidates with responses to less than N items.

trim(*values*, *trim_min*, *trim_max*, *tolerance=0.4998*)

Trim values in given numpy array.

The trimming uses `trim_min - tolerance` as the floor and `trim_max + tolerance` as the ceiling.

Parameters

- **values** (*list or np.array*) – The values to trim.
- **trim_min** (*float*) – The lowest score on the score point, used for trimming the raw regression predictions.
- **trim_max** (*float*) – The highest score on the score point, used for trimming the raw regression predictions.
- **tolerance** (*float, optional*) – The tolerance that will be used to compute the trim interval. Defaults to `0.4998`.

Returns **trimmed_values** – Trimmed values.

Return type np.array

class rsmtool.preprocessor.**FeatureSpecsProcessor**(*logger=None*)

Bases: object

Encapsulate feature file processing methods.

find_feature_sign (*feature*, *sign_dict*)
Get the feature sign from the feature CSV file.

Parameters

- **feature** (*str*) – The name of the feature.
- **sign_dict** (*dict*) – A dictionary of feature signs.

Returns **feature_sign_numeric** – The signed feature.

Return type float

generate_default_specs (*feature_names*)
Generate default feature "specifications" for given feature names.

The specifications are stored as a data frame with three columns "feature", "transform", and "sign".

Parameters **feature_names** (*list of str*) – List of feature names for which to generate specifications.

Returns **feature_specs** – A dataframe with feature specifications that can be saved as a *feature list file*.

Return type pandas DataFrame

Note: Since these are default specifications, the values for the "transform" column for each feature will be "raw" and the value for the "sign" column will be `1`.

---

generate_specs (*df*, *feature_names*, *train_label*, *feature_subset=None*, *feature_sign=None*)
Generate feature specifications using the feature CSV file.

Compute the specifications for "sign" and the correlation with score to identify the best transformation.

Parameters

- **df** (*pandas DataFrame*) – The input data frame from which to generate the specifications.
- **feature_names** (*list of str*) – A list of feature names.
- **train_label** (*str*) – The label column for the training data
- **feature_subset** (*pandas DataFrame, optional*) – A data frame containing the feature subset specifications. Defaults to `None`.
- **feature_sign** (*int, optional*) – The sign of the feature. Defaults to `None`.

Returns **df_feature_specs** – The output data frame containing the feature specifications.

Return type pandas DataFrame

validate_feature_specs (*df*, *use_truncations=False*)
Validate given feature specifications.

Check given feature specifications to make sure that there are no duplicate feature names and that all columns are in the right format. Add the default values for "transform" and "sign" if none are given.

Parameters

- **df** (*pandas DataFrame*) – The feature specification DataFrame to validate.
- **use_truncations** (*bool, optional*) – Whether to use truncation values. If this is `True` and truncation values are not specified, an exception is raised. Defaults to `False`.

Returns **df_specs_new** – The output data frame with normalized values.

Return type pandas DataFrame

Raises

- `KeyError` – If the input data frame does not have a "feature" column.
- `ValueError` – If there are duplicate values in the "feature" column.
- `ValueError` – if the "sign" column contains invalid values.
- `ValueError` – If `use_truncations` is set to `True`, and no "min" and "max" columns exist in the data set.

class rsmtool.preprocessor.**FeatureSubsetProcessor**(*logger=None*)

Bases: `object`

Class to encapsulate feature sub-setting methods.

check_feature_subset_file(*df*, *subset=None*, *sign=None*)

Check that feature subset file is complete and in the correct format.

Raises an exception if it finds any errors but otherwise returns nothing.

Parameters

- **df** (*pandas DataFrame*) – The data frame containing the feature subset file.
- **subset** (*str, optional*) – Name of a pre-defined feature subset. Defaults to `None`.
- **sign** (*str, optional*) – Value of the sign. Defaults to `None`.

Raises

- `ValueError` – If any columns are missing from the subset file.
- `ValueError` – If any of the columns contain invalid values.

select_by_subset(*feature_columns*, *feature_subset_specs*, *subset*)

Select feature columns using feature subset specifications.

Parameters

- **feature_columns** (*list of str*) – A list of feature columns
- **feature_subset_specs** (*pandas DataFrame*) – The feature subset specification data frame.
- **subset** (*str*) – The column to subset.

Returns **feature_names** – A list of feature names to include.

Return type list of str

From `prmse` Module

rsmtool.utils.prmse.**prmse_true**(*system*, *human_scores*, *variance_errors_human=None*)

Compute PRMSE when predicting true score from system scores.

PRMSE = Proportional Reduction in Mean Squared Error. The formula to compute PRMSE implemented in RSMTool was derived at ETS by Matthew S. Johnson. See Loukina et al. (2020) for further information about PRMSE.

Parameters

- **system** (*array-like of shape (n_samples,)*) – System scores for each response.
- **human_scores** (*array-like of shape (n_samples, n_ratings)*) – Human ratings for each response.
- **variance_errors_human** (*float, optional*) – Estimated variance of errors in human scores. If `None`, the variance will be estimated from the data. In this case at least some responses must have more than one human score. Defaults to `None`.

Returns **prmse** – Proportional reduction in mean squared error

Return type float

`rsmtool.utils.prmse.`**variance_of_errors**(*human_scores*)

Estimate the variance of errors in human scores.

Parameters **human_scores** (*array-like of shape (n_samples, n_ratings)*) – Human ratings for each response.

Returns **variance_of_errors** – Estimated variance of errors in human scores.

Return type float

From `reader` Module

Classes for reading data files (or dictionaries) into DataContainer objects.

author Jeremy Biggs (jbiggs@ets.org)

author Anastassia Loukina (aloukina@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

class `rsmtool.reader.`**DataReader**(*filepaths*, *framenames*, *file_converters=None*)

Bases: `object`

Class to generate DataContainer objects.

Initialize a DataReader object.

Parameters

- **filepaths** (*list of str*) – A list of paths to files that are to be read in.
- **framenames** (*list of str*) – A list of names for the data sets to be included in the container.
- **file_converters** (*dict of dicts, optional*) – A dictionary of file converter dicts. The keys are the data set names and the values are the converter dictionaries to be applied to the corresponding data set. Defaults to `None`.

Raises

- `AssertionError` – If `len(filepaths)` does not equal `len(framenames)`.
- `ValueError` – If `file_converters` is not a dictionary or if any of its values is not a dictionary.
- `NameError` – If a key in `file_converters` does not exist in `framenames`.
- `ValueError` – If any of the specified file paths is `None`.

static locate_files (*filepaths*, *configdir*)
Locate an experiment file, or a list of experiment files.

If the given path doesn't exist, then maybe the path is relative to the path of the config file. If neither exists, then return `None`.

Parameters

- **filepaths** (*str or list*) – Name(s) of the experiment file we want to locate.
- **configdir** (*str*) – Path to the reference configuration directory (usually the directory of the config file)

Returns **retval** – Absolute path to the experiment file or `None` if the file could not be located. If `filepaths` was a string, this method will return a string. Otherwise, it will return a list.

Return type str or list

Raises `ValueError` – If `filepaths` is not a string or a list.

read (*kwargs_dict=None*)
Read all files contained in `self.dataset_paths`.

Parameters **kwargs_dict** (*dict of dicts, optional*) – Any additional keyword arguments to pass to a particular DataFrame. These arguments will be passed to the pandas IO reader function. Defaults to `None`.

Returns **datacontainer** – A data container object.

Return type *container.DataContainer*

Raises `FileNotFoundError` – If any of the files in `self.dataset_paths` does not exist.

static read_from_file (*filename*, *converters=None*, ***kwargs*)
Read a CSV/TSV/XLSX/JSONLINES/SAS7BDAT file and return a data frame.

Parameters

- **filename** (*str*) – Name of file to read.
- **converters** (*dict, optional*) – A dictionary specifying how the types of the columns in the file should be converted. Specified in the same format as for pd.read_csv(). Defaults to `None`.

Returns **df** – Data frame containing the data in the given file.

Return type pandas DataFrame

Raises

- `ValueError` – If the file has an unsuppored extension.
- `pandas.errors.ParserError` – If the file is badly formatted or corrupt.

Note: Any additional keyword arguments are passed to the underlying pandas IO reader function.

rsmtool.reader.**read_jsonlines** (*filename*, *converters=None*)
Read a data file in .jsonlines format into a data frame.

Normalize nested jsons with up to one level of nesting.

Parameters

- **filename** (*str*) – Name of file to read.

- **converters** (*dict, optional*) – A dictionary specifying how the types of the columns in the file should be converted. Specified in the same format as for pd.read_csv(). Defaults to `None`.

Returns **df** – Data frame containing the data in the given file.

Return type pandas DataFrame

`rsmtool.reader.`**try_to_load_file**(*filename*, *converters=None*, *raise_error=False*, *raise_warning=False*, ***kwargs*)

Read a single file, if it exists.

Optionally raises an error or warning if the file cannot be found. Otherwise, returns `None`.

Parameters

- **filename** (*str*) – Name of file to read.
- **converters** (*dict, optional*) – A dictionary specifying how the types of the columns in the file should be converted. Specified in the same format as for pd.read_csv(). Defaults to `None`.
- **raise_error** (*bool, optional*) – Raise an error if the file cannot be located. Defaults to `False`.
- **raise_warning** (*bool, optional*) – Raise a warning if the file cannot be located. Defaults to `False`.

Returns **df** – DataFrame containing the data in the given file, or `None` if the file does not exist.

Return type pandas DataFrame or `None`

Raises `FileNotFoundError` – If `raise_error` is `True` and the file cannot be located.

From `reporter` Module

Classes for dealing with report generation.

author Jeremy Biggs (jbiggs@ets.org)

author Anastassia Loukina (aloukina@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

class `rsmtool.reporter.`**Reporter**(*logger=None*)

Bases: `object`

Class to generate Jupyter notebook reports and convert them to HTML.

static check_section_names(*specified_sections*, *section_type*, *context='rsmtool'*)

Validate the specified section names.

This function checks whether the specified section names are valid and raises an exception if they are not.

Parameters

- **specified_sections** (*list of str*) – List of report section names.
- **section_type** (*str*) – One of "general" or "special".
- **context** (*str, optional*) – Context in which we are validating the section names. One of {"rsmtool", "rsmeval", "rsmcompare"}. Defaults to "rsmtool".

Raises `ValueError` – If any of the section names of the given type are not valid in the context of the given tool.

static check_section_order(*chosen_sections*, *section_order*)
Check the order of the specified sections.

Parameters

- **chosen_sections** (*list of str*) – List of chosen section names.
- **section_order** (*list of str*) – An ordered list of the chosen section names.

Raises `ValueError` – If any sections specified in the order are missing from the list of chosen sections or vice versa.

static convert_ipynb_to_html(*notebook_file*, *html_file*)
Convert given Jupyter notebook (`.ipynb`) to HTML file.

Parameters

- **notebook_file** (*str*) – Path to input Jupyter notebook file.
- **html_file** (*str*) – Path to output HTML file.

Note: This function is also exposed as the *render_notebook* command-line utility.

---

create_comparison_report(*config*, *csvdir_old*, *figdir_old*, *csvdir_new*, *figdir_new*, *output_dir*)
Generate an HTML report for comparing two rsmtool experiments.

Parameters

- **config** (configuration_parser.Configuration) – A configuration object
- **csvdir_old** (*str*) – The old experiment CSV output directory.
- **figdir_old** (*str*) – The old figure output directory
- **csvdir_new** (*str*) – The new experiment CSV output directory.
- **figdir_new** (*str*) – The old figure output directory
- **output_dir** (*str*) – The output dir for the new report.

create_report(*config*, *csvdir*, *figdir*, *context='rsmtool'*)
Generate HTML report for an rsmtool/rsmeval experiment.

Parameters

- **config** (configuration_parser.Configuration) – A configuration object
- **csvdir** (*str*) – The CSV output directory.
- **figdir** (*str*) – The figure output directory
- **context** (*str*) – Context of the tool in which we are validating. One of {"rsmtool", "rsmeval"}. Defaults to "rsmtool".

Raises `KeyError` – If "test_file_location" or "pred_file_location" fields are not specified in the configuration.

create_summary_report(*config*, *all_experiments*, *csvdir*)
Generate an HTML report for summarizing the given rsmtool experiments.

Parameters

- **config** (configuration_parser.Configuration) – A configuration object

- **all_experiments** (*list of str*) – A list of experiment configuration files to summarize.
- **csvdir** (*str*) – The experiment CSV output directory.

determine_chosen_sections(*general_sections*, *special_sections*, *custom_sections*, *subgroups*, *context='rsmtool'*)

Compile a combined list of section names to be included in the report.

Parameters

- **general_sections** (*list of str*) – List of specified general section names.
- **special_sections** (*str*) – List of specified special section names, if any.
- **custom_sections** (*list of str*) – List of specified custom sections, if any.
- **subgroups** (*list of str*) – List of column names that contain grouping information.
- **context** (*str, optional*) – Context of the tool in which we are validating. One of {"rsmtool", "rsmeval", "rsmcompare"} Defaults to "rsmtool".

Returns **chosen_sections** – Final list of chosen sections that are to be included in the HTML report.

Return type list of str

Raises `ValueError` – If a subgroup report section is requested but no subgroups were specified in the configuration file.

get_ordered_notebook_files(*general_sections*, *special_sections=[]*, *custom_sections=[]*, *section_order=None*, *subgroups=[]*, *model_type=None*, *context='rsmtool'*)

Check all section names and the order of the sections.

Combine all section names with the appropriate file mapping, and generate an ordered list of notebook files that are needed to generate the final report.

Parameters

- **general_sections** (*str*) – List of specified general sections.
- **special_sections** (*list, optional*) – List of specified special sections, if any. Defaults to `[]`.
- **custom_sections** (*list, optional*) – List of specified custom sections, if any. Defaults to `[]`.
- **section_order** (*list, optional*) – Ordered list in which the user wants the specified sections. Defaults to `None`.
- **subgroups** (*list, optional*) – List of column names that contain grouping information. Defaults to `[]`.
- **model_type** (*None, optional*) – Type of the model. Possible values are {"BUILTIN", "SKLL", `None`.}. We allow `None` here so that rsmeval can use the same function. Defaults to `None`.
- **context** (*str, optional*) – Context of the tool in which we are validating. One of {"rsmtool", "rsmeval", "rsmcompare"}. Defaults to "rsmtool".

Returns **chosen_notebook_files** – List of the IPython notebook files that have to be rendered into the HTML report.

Return type list of str

get_section_file_map(*special_sections*, *custom_sections*, *model_type=None*, *context='rsmtool'*)

Map section names to IPython notebook filenames.

Parameters

- **special_sections** (*list of str*) – List of special sections.
- **custom_sections** (*list of str*) – List of custom sections.
- **model_type** (*str, optional*) – Type of the model. One of {"BUILTIN", "SKLL", None}. We allow None here so that rsmeval can use the same function. Defaults to None.
- **context** (*str, optional*) – Context of the tool in which we are validating. One of {"rsmtool", "rsmeval", "rsmcompare"}. Defaults to "rsmtool".

Returns **section_file_map** – Dictionary mapping each section name to the corresponding IPython notebook filename.

Return type dict

static locate_custom_sections(*custom_report_section_paths*, *configdir*)

Locate custom report section files.

Get the absolute paths for custom report sections and check that the files exist. If a file does not exist, raise an exception.

Parameters

- **custom_report_section_paths** (*list of str*) – List of paths to IPython notebook files representing the custom sections.
- **configdir** (*str*) – Path to the experiment configuration directory.

Returns **custom_report_sections** – List of absolute paths to the custom section notebooks.

Return type list of str

Raises FileNotFoundError – If any of the files cannot be found.

static merge_notebooks(*notebook_files*, *output_file*)

Merge the given Jupyter notebooks into a single Jupyter notebook.

Parameters

- **notebook_files** (*list of str*) – List of paths to the input Jupyter notebook files.
- **output_file** (*str*) – Path to output Jupyter notebook file

Note: Adapted from: https://stackoverflow.com/q/20454668.

From transformer Module

Class for transforming features.

author Jeremy Biggs (jbiggs@ets.org)

author Anastassia Loukina (aloukina@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

class rsmtool.transformer.**FeatureTransformer**(*logger=None*)
Bases: object

Encapsulate feature transformation methods.

apply_add_one_inverse_transform(*name*, *values*, *raise_error=True*)
Apply the "addOneInv" (add one and invert) transform to values.

Parameters

- **name** (*str*) – Name of the feature to transform.
- **values** (*np.array*) – Numpy array containing the feature values.
- **raise_error** (*bool, optional*) – If True, raises an error if the transform is applied to a feature that has zero or negative values. Defaults to True.

Returns **new_data** – Numpy array containing the transformed feature values.

Return type np.array

Raises ValueError – If the transform is applied to a feature that has negative values and raise_error is True.

apply_add_one_log_transform(*name*, *values*, *raise_error=True*)
Apply the "addOneLn" (add one and log) transform to values.

Parameters

- **name** (*str*) – Name of the feature to transform.
- **values** (*np.array*) – Numpy array containing the feature values.
- **raise_error** (*bool, optional*) – If True, raises an error if the transform is applied to a feature that has zero or negative values. Defaults to True.

Returns **new_data** – Numpy array that contains the transformed feature values.

Return type np.array

Raises ValueError – If the transform is applied to a feature that has negative values and raise_error is True.

apply_inverse_transform(*name*, *values*, *raise_error=True*, *sd_multiplier=4*)
Apply the "inv" (inverse) transform to values.

Parameters

- **name** (*str*) – Name of the feature to transform.
- **values** (*np.array*) – Numpy array containing the feature values.
- **raise_error** (*bool, optional*) –

  If True, raises an error if the transform is applied to a feature that has zero values or to a feature that has

  both positive and negative values. Defaults to True.
- **sd_multiplier** (*int, optional*) – Use this std. dev. multiplier to compute the ceiling and floor for outlier removal and check that these are not equal to zero. Defaults to 4.

Returns **new_data** – Numpy array containing the transformed feature values.

Return type np.array

Raises `ValueError` – If the transform is applied to a feature that is zero or to a feature that can have different signs, and `raise_error` is `True`.

apply_log_transform(*name*, *values*, *raise_error=True*)
Apply the "log" transform to `values`.

Parameters

- **name** (*str*) – Name of the feature to transform.
- **values** (*np.array*) – Numpy array containing the feature values.
- **raise_error** (*bool, optional*) – If `True`, raises an error if the transform is applied to a feature that has zero or negative values. Defaults to `True`.

Returns **new_data** – Numpy array containing the transformed feature values.

Return type numpy array

Raises `ValueError` – If the transform is applied to a feature that has zero or negative values and `raise_error` is `True`.

apply_sqrt_transform(*name*, *values*, *raise_error=True*)
Apply the "sqrt" transform to `values`.

Parameters

- **name** (*str*) – Name of the feature to transform.
- **values** (*np.array*) – Numpy array containing the feature values.
- **raise_error** (*bool, optional*) – If `True`, raises an error if the transform is applied to a feature that has negative values. Defaults to `True`.

Returns **new_data** – Numpy array containing the transformed feature values.

Return type np.array

Raises `ValueError` – If the transform is applied to a feature that has negative values and `raise_error` is `True`.

find_feature_transform(*feature_name*, *feature_value*, *scores*)
Identify best transformation for feature given correlation with score.

The best transformation is chosen based on the absolute Pearson correlation with human score.

Parameters

- **feature_name** (*str*) – Name of feature for which to find the transformation.
- **feature_value** (*pandas Series*) – Series containing feature values.
- **scores** (*pandas Series*) – Numeric human scores.

Returns **best_transformation** – The name of the transformation which gives the highest correlation between the feature values and the human scores. See *documentation* for the full list of transformations.

Return type str

transform_feature(*values*, *column_name*, *transform*, *raise_error=True*)
Apply given transform to all values in the given numpy array.

The values are assumed to be for the feature with the given name.

Parameters

- **values** (*numpy array*) – Numpy array containing the feature values.

- **column_name** (*str*) – Name of the feature to transform.
- **transform** (*str*) – Name of the transform to apply. One of {"inv", "sqrt", "log", "addOneInv", "addOneLn", "raw", "org"}.
- **raise_error** (*bool, optional*) – If `True`, raise a ValueError if a transformation leads to invalid values or may change the ranking of the responses. Defaults to `True`.

Returns **new_data** – Numpy array containing the transformed feature values.

Return type np.array

Raises `ValueError` – If the given transform is not recognized.

Note: Many of these transformations may be meaningless for features which span both negative and positive values. Some transformations may throw errors for negative feature values.

From `utils` Module

class rsmtool.utils.commandline.**ConfigurationGenerator**(*context*, *as_string=False*, *suppress_warnings=False*, *use_subgroups=False*)

Class to encapsulate automated batch-mode and interactive generation.

context
Name of the command-line tool for which we are generating the configuration file.

Type str

as_string
If `True`, return a formatted and indented string representation of the configuration, rather than a dictionary. Note that this only affects the batch-mode generation. Interactive generation always returns a string. Defaults to `False`.

Type bool, optional

suppress_warnings
If `True`, do not generate any warnings for batch-mode generation. Defaults to `False`.

Type bool, optional

use_subgroups
If `True`, include subgroup-related sections in the list of general sections in the configuration file. Defaults to `False`.

Type bool, optional

ConfigurationGenerator.**generate**()
Automatically generate an example configuration in batch mode.

Returns **configuration** – The generated configuration either as a dictionary or a formatted string, depending on the value of `as_string`.

Return type dict or str

rsmtool.utils.metrics.**agreement**(*score1*, *score2*, *tolerance=0*)
Compute the agreement between two raters, under given tolerance.

Parameters

- **score1** (*list of int*) – List of rater 1 scores

- **score2** (*list of int*) – List of rater 2 scores
- **tolerance** (*int, optional*) – Difference in scores that is acceptable. Defaults to 0.

Returns **agreement_value** – The percentage agreement between the two scores.

Return type float

rsmtool.utils.metrics.**difference_of_standardized_means**(*y_true_observed*, *y_pred*, *population_y_true_observed_mn=None*, *population_y_pred_mn=None*, *population_y_true_observed_sd=None*, *population_y_pred_sd=None*, *ddof=1*)

Calculate the difference between standardized means.

First, standardize both observed and predicted scores to z-scores using mean and standard deviation for the whole population. Then calculate differences between standardized means for each subgroup.

Parameters

- **y_true_observed** (*array-like*) – The observed scores for the group or subgroup.
- **y_pred** (*array-like*) – The predicted score for the group or subgroup. The predicted scores.
- **population_y_true_observed_mn** (*float, optional*) – The population true score mean. When the DSM is being calculated for a subgroup, this should be the mean for the whole population. Defaults to `None`.
- **population_y_pred_mn** (*float, optional*) – The predicted score mean. When the DSM is being calculated for a subgroup, this should be the mean for the whole population. Defaults to `None`.
- **population_y_true_observed_sd** (*float, optional*) – The population true score standard deviation. When the DSM is being calculated for a subgroup, this should be the standard deviation for the whole population. Defaults to `None`.
- **population_y_pred_sd** (*float, optional*) – The predicted score standard deviation. When the DSM is being calculated for a subgroup, this should be the standard deviation for the whole population. Defaults to None.
- **ddof** (*int, optional*) – The delta degrees of freedom. The divisor used in calculations is N - ddof, where N represents the number of elements. Defaults to 1.

Returns **difference_of_std_means** – The difference of standardized means

Return type array-like

Raises

- `ValueError` – If only one of `population_y_true_observed_mn` and `population_y_true_observed_sd` is not `None`.
- `ValueError` – If only one of `population_y_pred_mn` and `population_y_pred_sd` is not `None`.

rsmtool.utils.metrics.**partial_correlations**(*df*)

Implement the R `pcor` function from `ppcor` package in Python.

This computes partial correlations of each pair of variables in the given data frame `df`, excluding all other variables.

Parameters **df** (*pd.DataFrame*) – Data frame containing the feature values.

Returns **df_pcor** – Data frame containing the partial correlations of of each pair of variables in the given data frame `df`, excluding all other variables.

Return type pd.DataFrame

rsmtool.utils.metrics.**quadratic_weighted_kappa**(*y_true_observed*, *y_pred*, *ddof=0*)

Calculate quadratic-weighted kappa for both discrete and continuous values.

The formula to compute quadratic-weighted kappa for continuous values was developed at ETS by Shelby Haberman. See Haberman (2019) for the full derivation. The discrete case is simply treated as a special case of the continuous one.

The formula is as follows:

$QWK = \frac{2 * Cov(M, H)}{Var(H) + Var(M) + (\bar{M} - \bar{H})^2}$, where

- Cov - covariance with normalization by N (the total number of observations given)
- H - the human score
- M - the system score
- $\bar{H}$ - mean of H
- $\bar{M}$ - mean of M
- $Var(X)$ - variance of X

Parameters

- **y_true_observed** (*array-like*) – The observed scores.
- **y_pred** (*array-like*) – The predicted scores.
- **ddof** (*int, optional*) – Means Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements. When ddof is set to zero, the results for discrete case match those from the standard implementations. Defaults to 0.

Returns **kappa** – The quadratic weighted kappa

Return type float

Raises `AssertionError` – If the number of elements in `y_true_observed` is not equal to the number of elements in `y_pred`.

rsmtool.utils.metrics.**standardized_mean_difference**(*y_true_observed*, *y_pred*, *population_y_true_observed_sd=None*, *population_y_pred_sd=None*, *method='unpooled'*, *ddof=1*)

Compute the standardized mean difference between system and human scores.

The numerator is calculated as mean(y_pred) - mean(y_true_observed) for all of the available methods.

Parameters

- **y_true_observed** (*array-like*) – The observed scores for the group or subgroup.
- **y_pred** (*array-like*) – The predicted score for the group or subgroup.

- **population_y_true_observed_sd** (*float, optional*) – The population true score standard deviation. When the SMD is being calculated for a subgroup, this should be the standard deviation for the whole population. Defaults to `None`.
- **population_y_pred_sd** (*float, optional*) – The predicted score standard deviation. When the SMD is being calculated for a subgroup, this should be the standard deviation for the whole population. Defaults to `None`.
- **method** (*str, optional*) – The SMD method to use. Possible options are:
  - "williamson": Denominator is the pooled population standard deviation of `y_true_observed` and `y_pred` computed using `population_y_true_observed_sd` and `population_y_pred_sd`.
  - "johnson": Denominator is `population_y_true_observed_sd`.
  - "pooled": Denominator is the pooled standard deviation of `y_true_observed` and `y_pred` for this group.
  - "unpooled": Denominator is the standard deviation of `y_true_observed` for this group.

  Defaults to "unpooled".
- **ddof** (*int, optional*) – The delta degrees of freedom. The divisor used in calculations is N - ddof, where N represents the number of elements. Defaults to 1.

Returns **smd** – The SMD for the given group or subgroup.

Return type float

Raises

- `ValueError` – If method is "williamson" and either `population_y_true_observed_sd` or `population_y_pred_sd` is `None`.
- `ValueError` – If method is "johnson" and `population_y_true_observed_sd` is `None`.
- `ValueError` – If method is not one of {"unpooled", "pooled", "williamson", "johnson"}.

Note:

- The "williamson" implementation was recommended by Williamson, et al. (2012).
- The metric is only applicable when both sets of scores are on the same scale.

---

rsmtool.utils.metrics.**compute_expected_scores_from_model**(*model*, *featureset*, *min_score*, *max_score*)

Compute expected scores using probability distributions over labels.

This function only works with SKLL models.

Parameters

- **model** (*skll.learner.Learner*) – The SKLL learner object to use for computing the expected scores.
- **featureset** (*skll.data.FeatureSet*) – The SKLL featureset object for which predictions are to be made.
- **min_score** (*int*) – Minimum score level to be used for computing expected scores.
- **max_score** (*int*) – Maximum score level to be used for computing expected scores.

Returns **expected_scores** – A numpy array containing the expected scores.

Return type np.array

Raises

- `ValueError` – If the given model cannot predict probability distributions.
- `ValueError` – If the score range specified by `min_score` and `max_score` does not match what the model predicts in its probability distribution.

`rsmtool.utils.notebook.`**`get_thumbnail_as_html`**(*path_to_image*, *image_id*, *path_to_thumbnail=None*)

Generate HTML for a clickable thumbnail of given image.

Given the path to an image file, generate the HTML for a clickable thumbnail version of the image. When clicked, this HTML will open the full-sized version of the image in a new window.

Parameters

- **`path_to_image`** (*`str`*) – The absolute or relative path to the image. If an absolute path is provided, it will be converted to a relative path.
- **`image_id`** (*`int`*) – The id of the <img> tag in the HTML. This must be unique for each <img> tag.
- **`path_to_thumbnail`** (*`str, optional`*) – If you would like to use a different thumbnail image, specify the path to this thumbnail. Defaults to `None`.

Returns **image** – The HTML string generated for the image.

Return type str

Raises `FileNotFoundError` – If the image file cannot be located.

`rsmtool.utils.notebook.`**`show_thumbnail`**(*path_to_image*, *image_id*, *path_to_thumbnail=None*)

Display the HTML for an image thumbnail in a Jupyter notebook.

Given the path to an image file, generate the HTML for its thumbnail and display it in the notebook.

Parameters

- **`path_to_image`** (*`str`*) – The absolute or relative path to the image. If an absolute path is provided, it will be converted to a relative path.
- **`image_id`** (*`int`*) – The id of the <img> tag in the HTML. This must be unique for each <img> tag.
- **`path_to_thumbnail`** (*`str, optional`*) – If you would like to use a different thumbnail image, specify the path to the thumbnail. Defaults to `None`.
- **`Displays`** –
- **`--------`** –
- **`display`** (*`IPython.core.display.HTML`*) – The HTML for the thumbnail image.

`rsmtool.utils.files.`**`parse_json_with_comments`**(*pathlike*)

Parse a JSON file after removing any comments.

Comments can use either `//` for single-line comments or or `/* ... */` for multi-line comments. The input filepath can be a string or `pathlib.Path`.

Parameters **`filename`** (*`str or os.PathLike`*) – Path to the input JSON file either as a string or as a `pathlib.Path` object.

Returns **obj** – JSON object representing the input file.

Return type dict

Note: This code was adapted from: https://web.archive.org/web/20150520154859/http://www.lifl.fr/~riquetd/parse-a-json-file-with-comments.html

---

From `writer` Module

Class for writing DataContainer frames to disk.

author Jeremy Biggs (jbiggs@ets.org)

author Anastassia Loukina (aloukina@ets.org)

author Nitin Madnani (nmadnani@ets.org)

organization ETS

class rsmtool.writer.**DataWriter**(*experiment_id=None*)

Bases: object

Class to write out DataContainer objects.

write_experiment_output(*csvdir*, *container_or_dict*, *dataframe_names=None*, *new_names_dict=None*, *include_experiment_id=True*, *reset_index=False*, *file_format='csv'*, *index=False*, ***kwargs*)

Write out each of the named frames to disk.

This function writes out each of the given list of data frames as a ".csv", ".tsv", or `.xlsx` file in the given directory. Each data frame was generated as part of running an RSMTool experiment. All files are prefixed with the given experiment ID and suffixed with either the name of the data frame in the DataContainer (or dict) object, or a new name if `new_names_dict` is specified. Additionally, the indexes in the data frames are reset if so specified.

Parameters

- **csvdir** (*str*) – Path to the output experiment sub-directory that will contain the CSV files corresponding to each of the data frames.
- **container_or_dict** (*container.DataContainer or dict*) – A DataContainer object or dict, where keys are data frame names and values are `pd.DataFrame` objects.
- **dataframe_names** (*list of str, optional*) – List of data frame names, one for each of the data frames. Defaults to `None`.
- **new_names_dict** (*dict, optional*) – New dictionary with new names for the data frames, if desired. Defaults to `None`.
- **include_experiment_id** (*str, optional*) – Whether to include the experiment ID in the file name. Defaults to `True`.
- **reset_index** (*bool, optional*) – Whether to reset the index of each data frame before writing to disk. Defaults to `False`.
- **file_format** (*str, optional*) – The file format in which to output the data. One of {"csv", "xlsx", "tsv"}. Defaults to "csv".
- **index** (*bool, optional*) – Whether to include the index in the output file. Defaults to `False`.

Raises `KeyError` – If `file_format` is not valid, or a named data frame is not present in `container_or_dict`.

write_feature_csv(*featuredir*, *data_container*, *selected_features*, *include_experiment_id=True*, *file_format='csv'*)
Write out the selected features to disk.

Parameters

- **featuredir** (*str*) – Path to the experiment output directory where the feature JSON file will be saved.
- **data_container** (container.DataContainer) – A data container object.
- **selected_features** (*list of str*) – List of features that were selected for model building.
- **include_experiment_id** (*bool, optional*) – Whether to include the experiment ID in the file name. Defaults to `True`.
- **file_format** (*str, optional*) – The file format in which to output the data. One of {"csv", "tsv", "xlsx"}. Defaults to "csv".

static write_frame_to_file(*df*, *name_prefix*, *file_format='csv'*, *index=False*, ***kwargs*)
Write given data frame to disk with given name and file format.

Parameters

- **df** (*pandas DataFrame*) – Data frame to write to disk
- **name_prefix** (*str*) – The complete prefix for the file to be written to disk. This includes everything except the extension.
- **file_format** (*str*) – The file format (extension) for the file to be written to disk. One of {"csv", "xlsx", "tsv"}. Defaults to "csv".
- **index** (*bool, optional*) – Whether to include the index in the output file. Defaults to `False`.

Raises `KeyError` – If `file_format` is not valid.

1.11 Utility Scripts

In addition to the *rsmtool*, *rsmeval*, *rsmpredict*, *rsmcompare*, and *rsmsummarize* scripts, RSMTool also comes with a number of helpful utility scripts.

1.11.1 render_notebook

Convert a given Jupyter notebook file (`.ipynb`) to HTML (`.html`) format.

ipynb_file
Path to input Jupyter notebook file.

html_file
Path to output HTML file.

-h, --help
Show help message and exit.

1.11.2 convert_feature_json

Convert an older feature JSON file to a new file in tabular format.

--json
Path to input feature JSON file that is to be converted.

--output
Path to output CSV/TSV/XLSX file containing the features in tabular format.

--delete
Delete original feature JSON file after conversion.

-h, --help
Show help message and exit.

1.12 Contributing to RSMTool

Contributions to RSMTool are very welcome. You can use the instructions below to get started on developing new features or functionality for RSMTool. When contributing to RSMTool, all of your contributions must come with updates to documentations as well as tests.

1.12.1 Setting up

To set up a local development environment, follow the steps below:

1. Clone the Github repository for RSMTool. If you have already have a local version of the repository, pull the latest version from GitHub and switch to the `main` branch.
2. If you already have the `conda` package manager installed, skip to the next step. If you do not, follow the instructions on this page to install conda.
3. Create a new conda environment (say, `rsmdev`) and install the packages specified in the `requirements.txt` file by running:

```
conda create -n rsmdev -c conda-forge -c ets --file requirements.txt
```

4. Activate the environment using `conda activate rsmdev`.[1]
5. Run `pip install -e .` to install rsmtool into the environment in editable mode which is what we need for development.
6. Create a new git branch with a useful and descriptive name.
7. Make your changes and add tests. See the next section for more on writing new tests.
8. Run `nosetests -v --nologcapture tests` to run the tests. We use the `--nologcapture` switch, since otherwise test failures for some tests tend to produce very long Jupyter notebook traces.

1.12.2 Documentation

Note that the file `doc/requirements.txt` is meant specifically for the ReadTheDocs documentation build process and should not be used locally. To build the documentation locally, you *must* use the same conda environment created above.

If you are on macOS and use the Dash app, follow steps 1 and 2 *here* to build the RSMTool Dash docset locally.

[1] For older versions of conda, you may have to do `source activate rsmtool` on Linux/macOS and `activate rsmtool` on Windows.

1.12.3 Code style

The RSMTool codebase follows certain best practices when it comes to the code style and we expect any contributed code to do the same. These best practices are:

1. The imports at the top of any Python files should be grouped and sorted as follows: STDLIB, THIRDPARTY, FIRSTPARTY, LOCALFOLDER. As an example, consider the imports at the top of `reporter.py` which look like this:

```
import argparse
import asyncio
import json
import logging
import os
import sys
from os.path import abspath, basename, dirname, join, splitext

from nbconvert.exporters import HTMLExporter
from traitlets.config import Config

from . import HAS_RSMEXTRA
from .reader import DataReader
```

   Rather than doing this grouping and sorting manually, we recommend to use the isort Python library to do this. The best way to use `isort` is via plugins for your favorite editor, e.g., Sublime Text, VS Code, and PyCharm.

2. All classes, functions, and methods in the main code files should have numpy-formatted docstrings that comply with PEP 257. For Sublime Text, this can be done using the AutoDocstring and SublimeLinter-pydocstyle plugins. For VS Code, these two links may be relevant. PyCharm does not seem to support automatic numpy-format docstrings out of the box.

3. When writing docstrings, make sure to use the appropriate quotes when referring to argument names vs. argument values. As an example, consider the docstring for the train_skll_model method of the `rsmtool.modeler.Modeler` class. Note that string argument values are enclosed in double quotes (e.g., "csv", "neg_mean_squared_error") whereas values of other built-in types are written as literals (e.g., `True`, `False`, `None`). Note also that if one had to refer to an argument name in the docstring, this referent should be written as a literal. In general, we strongly encourage looking at the docstrings in the existing code to make sure that new docstrings follow the same practices.

1.12.4 RSMTool tests

Existing tests for RSMTool are spread across the various `test_*.py` files under the `tests` directory after you check out the RSMTool source code from GitHub.

There are two kinds of existing tests in RSMTool:

1. The first type of tests are **unit tests**, i.e., very specific tests for which you have a single example (usually embedded in the test itself) and you compare the generated output with known or expected output. These tests should have a very narrow and well defined scope. To see examples of such unit tests, see the test functions in the file *tests/test_utils.py*.

2. The second type of tests are **functional tests** which are generally written from the users' perspective to test that RSMTool is doing things that users would expect it to. In RSMTool, most (if not all) functional tests are written in the form of "experiment tests", i.e., we first define an experimental configuration using an `rsmtool` (or `rsmeval`/`rsmpredict`/`rsmcompare`/`rsmsummarize`) configuration file, then we run the experiment, and then compare the generated output files to expected output files to make sure that RSMTool

components are operating as expected. To see examples of such tests, you can look at any of the `tests/test_experiment_*.py` files.

Note: RSMTool functional tests are *parameterized*, i.e., since most are identical other than the configuration file that needs to be run, the basic functionality of the test has been factored out into utility functions. Each line starting with *param* in any of the `test_experiment_*.py` files represents a specific functional test.

Any new contributions to RSMTool, no matter how small or trivial, *must* be accompanied by updates to documentations as well as new unit and/or functional tests. Adding new unit tests is fairly straightforward. However, adding new functional tests is a little more involved.

1.12.5 Writing new functional tests

To write a new experiment test for RSMTool (or any of the other tools):

(a) Create a new directory under `tests/data/experiments` using a descriptive name.

(b) Create a JSON configuration file under that directory with the various fields appropriately set for what you want to test. Feel free to use multiple words separated by hyphens to come up with a name that describes the testing condition. The name of the configuration file should be the same as the value of the `experiment_id` field in your JSON file. By convention, that's usually the same as the name of the directory you created but with underscores instead of hyphens. If you are creating a new test for `rsmcompare` or `rsmsummarize`, copy over one or more of the existing `rsmtool` or `rsmeval` test experiments as input(s) and keep the same name. This will ensure that these inputs will be regularly updated and remain consistent with the current outputs generated by these tools. If you must create a test for a scenario not covered by a current tool, create a new `rsmtool`/`rsmeval` test first following the instructions on this page.

(c) Next, you need to add the test to the list of parameterized tests in the appropriate test file based on the tool for which you are adding the test, e.g., `rsmeval` tests should be added to `tests/test_experiment_rsmeval.py`, `rsmpredict` tests to `tests/test_experiment_rsmpredict.py`, and so on. Tests for `rsmtool` can be added to any of the four files. The arguments for the *param()* call can be found in the *Table 1* below.

(d) In some rare cases, you might want to use a non-parameterized experiment test if you are doing something very different. These should be few and far between. Examples of these can also be seen in various `tests/test_experiment_*.py` files.

(e) Another rare scenario is the need to create an entirely new `tests/test_experiment_X.py` file instead of using one of the existing ones. This should *not* be necessary unless you are trying to test a newly added tool or component.

Table 1: Table 1: Arguments for `param()` when adding new parameterized functional tests

Writing test(s) for `rsmtool` • First positional argument is the name of the test directory you created. • Second positional argument is the experiment ID from the JSON configuration file. • Use `consistency=True` if you have set *second_human_score_column* in the configuration file. • Use `skll=True` if you are writing a test for a SKLL model. • Set `subgroups` keyword argument to the same list that you specified in the configuration file. • Set `file_format="tsv"` (or `"xlsx"`) if you specified the same field in the configuration file.
Writing test(s) for `rsmeval` • Same arguments as RSMTool except the `skll` keyword argument is not applicable.
Writing test(s) for `rsmpredict` • The only positional argument is the name of the test directory you created. • Use `excluded=True` if you want to check the excluded responses file as part of the test. • Set `file_format="tsv"` (or `"xlsx"`) if you specified the same field in the configuration file.
Writing test(s) for `rsmcompare` • First positional argument is the name of the test directory you created. • Second positional argument is the comparison ID from the JSON configuration file.
Writing test(s) for `rsmsummarize` • The only positional argument is the name of the test directory you created. • Set `file_format="tsv"` (or `"xlsx"`) if you specified the same field in the configuration file.

Once you have added all new functional tests, commit all of your changes. Next, you should run `nosetests --nologcapture` to run all the tests. Obviously, the newly added tests will fail since you have not yet generated the expected output for that test.

To do this, you should now run the following:

```
python tests/update_files.py --tests tests --outputs test_outputs
```

This will copy over the generated outputs for the newly added tests and show you a report of the files that it added. It will also update the input files form tests for `rsmcompare` and `rsmsummarize`. If run correctly, the report should *only* refer the files affected by the functionality you implemented. If you run `nosetests` again, your newly added tests should now pass.

At this point, you should inspect all of the new test files added by the above command to make sure that the outputs are as expected. You can find these files under `tests/data/experiments/<test>/output` where `<test>` refers to the test(s) that you added.

However, if your changes resulted in updates to the inputs to `rsmsummarize` or `rsmcompare` tests, you will first need to re-run the tests for these two tools and then re-run the `update_files.py` to update the outputs.

Once you are satisified that the outputs are as expected, you can commit them.

The two examples below might help make this process easier to understand:

Example 1: You made a code change to better handle an edge case that only affects one test.

1. Run `nosetests --nologcapture tests/*.py`. The affected test failed.
2. Run `python tests/update_files.py --tests tests --outputs test_outputs` to update test outputs. You will see the total number of deleted, updated and missing files. There should be no deleted files and no missing files. Only the files for your new test should be updated. There are no warnings in the output.
3. If this is the case, you are now ready to commit your change and the updated test outputs.

Example 2: You made a code change that changes the output of many tests. For example, you renamed one of the evaluation metrics.

1. Run `nosetests --nologcapture tests/*.py`. Many tests will now fail since the output produced by the tool(s) has changed.
2. Run `python tests/update_files.py --tests tests --outputs test_outputs` to update test outputs. The files affected by your change are shown as added/deleted. You also see the following warning:

```
WARNING: X input files for rsmcompare/rsmsummarize tests have been updated. You␣
↪need to re-run these tests and update test outputs
```

3. This means that the changes you made to the code changed the outputs for one or more `rsmtool`/`rsmeval` tests that served as inputs to one or more `rsmcompare`/`rsmsummarize` tests. Therefore, it is likely that the current test outputs no longer match the expected output and the tests for those two tools must be be re-run.
4. Run `nosetests --nologcapture tests/*rsmsummarize*.py` and `nosetests --nologcapture tests/*rsmcompare*.py`. If you see any failures, make sure they are related to the changes you made since those are expected.
5. Next, re-run `python tests/update_files.py --tests tests --outputs test_outputs` which should only update the outputs for the `rsmcompare`/`rsmsummarize` tests.
6. If this is the case, you are now ready to commit your changes.

1.12.6 Advanced tips and tricks

Here are some advanced tips and tricks when working with RSMTool tests.

1. To run a specific test function in a specific test file, simply use `nosetests --nologcapture tests/test_X.py:Y` where `test_X.py` is the name of the test file, and `Y` is the test functions. Note that this will not work for parameterized tests. If you want to run a specific parameterized test, you can comment out all of the other `param()` calls and run the `test_run_experiment_parameterized()` function as above.
2. If you make any changes to the code that can change the output that the tests are expected to produce, you *must* re-run all of the tests and then update the *expected* test outputs using the `update_files.py` command as shown *above*.
3. In the rare case that you *do* need to create an entirely new `tests/test_experiment_X.py` file instead of using one of the existing ones, you can choose whether to exclude the tests contained in this file from updating their expected outputs when `update_files.py` is run by setting `_AUTO_UPDATE=False` at the top of the file. This should *only* be necessary if you are absolutely sure that your tests will never need updating.

4. The `--pdb-errors` and `--pdb-failures` options for `nosetests` are your friends. If you encounter test errors or test failures where the cause may not be immediately clear, re-run the `nosetests` command with the appropriate option. Doing so will drop you into an interactive PDB session as soon as a error (or failure) is encountered and then you inspect the variables at that point (or use "u" and "d" to go up and down the call stack). This may be particularly useful for tests in `tests/test_cli.py` that use `subprocess.run()`. If these tests are erroring out, use `--pdb-errors` and inspect the "stderr" variable in the resulting PDB session to see what the error is.
5. In RSMTool 8.0.1 and later, the tests will pass even if any of the reports contain warnings. To catch any warnings that may appear in the reports, run the tests in strict mode (`STRICT=1 nosetests --nologcapture tests`).

1.13 Internal Documentation

This section of the documentation is meant only for the project administrators, not users and developers.

1.13.1 RSMTool Release Process

This process is only meant for the project administrators, not users and developers.

1. Recreate the development environment so all unpinned packages are updated to their latest versions. See instructions for this here.
2. Make sure any and all tests are passing in `main`. Make sure you have also run tests locally in strict mode (`STRICT=1 nosetests --nologcapture tests`) to catch any warnings in the HTML report that can be fixed before the release.
3. Run the `tests/update_files.py` script with the appropriate arguments to make sure that all test data in the new release have correct experiment ids and filenames. If any (non-model) files need to be changed this should be investigated before the branch is released. Please see more details about running this here.

   Note: Several files have been excluded from the repository due to their non-deterministic nature so please do not add them back to the repository. The following files are currently excluded:

   - Fairness test files for *lr-eval-system-score-constant* test
   - Predictions and all evaluation files for *linearsvr* test.

   Note that the full set of outputs from these test files are also used as input for *rsmcompare* and *rsmsummarize* tests. These *input* files need to be updated following the process under **Example 2** in Writing new functional tests. You can also see this pull request for more information.

   ---

4. Create a release branch `release/XX` on GitHub.
5. In the release branch:
   1. update the version numbers in `version.py`.
   2. update the conda recipe.
   3. update the documentation with any new features or details about changes.
   4. run `make linkcheck` on the documentation (i.e. from the `doc/` folder) and fix any redirected/broken links.
   5. update the README and this release documentation, if necessary.

6. Build the PyPI source and wheel distributions using `python setup.py sdist build` and `python setup.py bdist_wheel build` respectively.
7. Upload the source and wheel distributions to TestPyPI using `twine upload --repository testpypi dist/*`. You will need to have the `twine` package installed and set up your `$HOME/.pypirc` correctly. See details here. You will need to have the appropriate permissions for the `ets` organization on TestPyPI.
8. Install the TestPyPI package as follows:

```
pip install --index-url https://test.pypi.org/simple/ --extra-index-url https://
↪pypi.org/simple rsmtool
```

9. Then run some tests from a RSMTool working copy. If the TestPyPI package works, then move on to the next step. If it doesn't, figure out why and rebuild and re-upload the package.
10. Build the new conda package by running the following command in the `conda-recipe` directory (note that this assumes that you have cloned RSMTool in a directory named `rsmtool`). Note that you may need to comment out lines in your *$HOME/.condarc* file if you are using ETS Artifactory and you get conflicts:

```
conda build -c conda-forge -c ets .
```

11. This will create a noarch package with the path to the package printed out to the screen.
12. Upload the package file to anaconda.org using `anaconda upload --user ets <path_to_file>`. You will need to have the appropriate permissions for the `ets` organization.
13. Create pull requests on the rsmtool-conda-tester and rsmtool-pip-tester repositories to test the conda and TestPyPI packages on Linux and Windows.
14. Draft a release on GitHub while the Linux and Windows package tester builds are running.
15. Once both builds have passed, make a pull request with the release branch to be merged into `main` and request code review.
16. Once the build for the PR passes and the reviewers approve, merge the release branch into `main`.
17. Upload the already-built source and wheel packages to PyPI using `twine upload dist/*`. You will need to have the `twine` package installed and set up your `$HOME/.pypirc` correctly. You will need to have the appropriate permissions for the `ets` organization on PyPI.
18. Make sure that the ReadTheDocs build for `main` passes by examining the badge at this URL - this should say "passing" in green.
19. Tag the latest commit in `main` with the appropriate release tag and publish the release on GitHub.
20. Make another PR to merge `main` branch into `stable` so that the the default ReadTheDocs build (which is `stable`) always points to the latest release. There are two versions of the documentation, one for the `stable` branch and the other for the `main` branch. The `stable` version of the documentation needs to be updated with each new release.
21. Update the CI plan for RSMExtra (only needed for ETS users) to use this newly built RSMTool conda package. Do any other requisite changes for RSMExtra. Once everything is done, do a release of RSMExtra.
22. Update the RSMTool conda environment on the ETS linux servers with the latest packages for both RSMTool and RSMExtra.
23. Send an email around at ETS announcing the release and the changes.
24. Create a Dash docset from the documentation by following the instructions *here*.

1.13.2 Generating Dash Docset

These instructions are for creating a docset based on the RSMTool documentation to be used for the macOS Dash app.

1. If you have created/deleted any documentation files, update the variable `FILES_TO_MODIFY` at the top of the file `doc/add_dash_anchors.py` to reflect these changes.
2. Run `make dash` in the `doc` directory. This will compile the HTML documentation using the Alabaster theme (which is better suited for this purpose), run `add_dash_anchors.py` for Dash TOC support, and create the docset file (`_build/RSMTool.docset`).
3. Clone the repository at https://github.com/Kapeli/Dash-User-Contributions.
4. Follow the instructions in that repository's README. Note that we do not need add explicit icons for the docset since our icon is already included in the docset file created above. Make sure to submit a pull request to that repo in the end!

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index

Symbols

A

C

D

F

G

H

I

K

L

M

O

P

Q

R

S

T

U

V

W