Technical Reference to the `oda` Software Package for Stata

Ariel Linden, DrPH
Linden Consulting Group, LLC
alinden@lindenconsulting.org

Purpose and scope

This document is a companion to the `oda` help file and describes the exact computation performed for every option `oda` allows. Two conventions recur throughout and are stated once here rather than in every section. First, `oda` always evaluates a candidate model using an *effect strength* (ESS) score, computed from a mean accuracy $\overline{\text{acc}}$ (defined per option below, since it changes between the priors/weights combinations) on a 0–100 scale with 50 as the chance baseline for a binary class variable:

$$\text{ESS} = \frac{\overline{\text{acc}} - 50}{50} \times 100. \tag{1}$$

For a K-category class variable the chance baseline generalizes to $100/K$:

$$\text{ESS}_K = \frac{\overline{\text{pac}} - 100/K}{100 - 100/K} \times 100. \tag{2}$$

Second, every performance measure `oda` reports for a fitted model – Overall classification accuracy, sensitivity (SENS), predictive value (PV), and their Effect Strengths, at training, leave-one-out, and (if requested) cross-validated stages – is computed *twice* whenever `wt()` is given: once using a constant weight of 1 for every observation (the "plain" track, always computed, even without `wt()`), and once using the real weight variable (the "_wtd"-suffixed track, computed only when `wt()` is given). Both use the exact same fitted cutpoint or category grouping – `wt()` never causes two different models to be fit. This dual reporting is described once here and referenced, not repeated, in every option section below.

To briefly describe how an ODA model is obtained: consider a continuous `attrvar` (attribute) and a binary `classvar` (class variable). The `attrvar` values are ordered from low to high, and every point along that ordering where the next observation belongs to the other class from the previous one becomes a candidate cutpoint, taken as the mean of the two adjacent values: cutpoint = (previous value + current value)/2. Directionality determines which class is predicted on which side of a candidate cutpoint – "less than" (one class tends to have lower `attrvar` values) or "greater than" (that class tends to have higher values); an exploratory, two-sided run evaluates both directions at every candidate cutpoint, while a confirmatory, one-sided run (`direction()`) evaluates only the specified one. For each candidate cutpoint and direction, the resulting classification is scored using the mean accuracy $\overline{\text{acc}}$ and its corresponding Effect Strength, ESS (Eq. 1) or, for $K > 2$ classes, ESS_K (Eq. 2) – the maximally accurate model is the cutpoint/direction combination (or, for a categorical `attrvar`, the category grouping; for a K-category `classvar` against a continuous `attrvar`, the ordered segmentation) with the single highest associated ESS. Based on simulation research, ESS values under 25% conventionally indicate a relatively weak effect, 25–50% a moderate

effect, 50–75% a relatively strong effect, and above 75% a strong effect (Yarnold & Soltysik 2005, 2016).

Statistical reliability of the maximally accurate model is assessed via Monte Carlo permutation experiments: class labels are repeatedly shuffled while `attrvar` is held fixed, the entire search is rerun on each shuffled dataset, and the permutation p-value is the proportion of shuffles whose ESS meets or exceeds the ESS of the real, unshuffled fit.

Generalizability of a fitted model is assessed via leave-one-out (jackknife) cross-validation (`loo`, §10): a model estimated on a training sample is applied to one or more held-out observations, and if its accuracy measures on the held-out data remain consistent with the training-sample measures, the model is considered generalizable. This matters most when the goal is to identify new candidates for an intervention, or to extend it to other settings; it matters less when the goal is only to estimate a treatment effect within the existing sample. `gen()` (§9) and `cross()` (§5) extend this idea to, respectively, reporting one fitted model's performance broken down by subgroup, and validating a model fit on one designated sample against one or more separate holdout samples.

Users are strongly encouraged to read Yarnold & Soltysik (2016) for a comprehensive guide to the underlying ODA methodology.

1 `attrvar` (attribute)

Stata syntax: the second positional argument, `oda classvar attrvar`.

An *attribute* can be thought of as a *dependent* or *outcome* variable. By default `attrvar` is treated as continuous (ordered), and `oda` searches for a cutpoint (binary `classvar`) or a full ordered segmentation (K-category `classvar`, §**??**). `cat` switches it to categorical (§`cat`) instead. When `cat` is given, `tab `attr'` must report at least 2 categories or will produce an error.

`attrvar` cannot be the same variable as `classvar`, `wt()`, `gen()`, or `cross()`'s sample identifier – each of those options checks explicitly against both `classvar` and `attrvar` at parse time and exits with an error.

2 `classvar`

Stata syntax: the first positional argument, `oda classvar attrvar`.

`classvar` must be categorical. ODA's entire measure set (sensitivity, specificity, PAC, PV, and their Effect Strengths) is defined from a classification confusion table, which has no meaning against a continuous outcome. A user must categorize a continuous outcome (a median split, quantiles, a clinically meaningful cutoff) before passing it to `oda`.

`classvar` cannot be the same variable as `attrvar`, `wt()`, `gen()`, or `cross()`'s sample identifier, per the same parse-time checks described under `attrvar` above. For a binary `classvar` combined with `gen()`, there is an additional constraint: every `gen()` group must contain at least one observation of each class or an error is produced – there is no such constraint for a K-category `classvar`.

3 `cat`

Stata syntax: `cat`

When `cat` is specified, `attrvar`'s distinct values are treated as an unordered set of categories, and `oda` searches over assignments of categories to predicted classes instead of a numeric threshold.

The categorical search computes, for each distinct category level of `attrvar`, the weighted count of class-1 and class-0 observations falling in that level. It then sorts categories by their own within-category class-1 proportion and considers every possible split of the sorted category list into a "predict class 1" side and a "predict class 0" side (an ordered-by-proportion search over the categories, not an exhaustive 2^{levels} enumeration, which would be needed only if categories had no natural accuracy ordering to sort by – sorting first make the search linear instead of exponential). The same mean-accuracy formula used for the continuous case is evaluated at every such split, and the tie-break machinery (§15) is applied identically to the continuous case.

For a categorical attribute with more than two classes, each category level is handled independently: it computes that category's own weighted class tally and assigns the category to whichever class has the highest (priors-adjusted or raw, per `nopriors`) representation within it, an argmax/majority vote. No joint combinatorial optimization across categories is needed here (unlike the continuous $K > 2$ case below), because categories are unordered: each category's best class assignment does not depend on any other category's assignment.

4 `ci, cireps(#), cilevel(#)`

Stata syntax: `ci` [`cireps(#)`] [`cilevel(#)`]

`ci` computes confidence intervals using a standard nonparametric percentile-method bootstrap `cireps()` times (default 200). The training rows are resampled with replacement (uniformly, per §20), the *entire* search is refit on that resample (respecting `direction()`, `nopriors`, `degen`, `wt()`'s search-weighting behavior), and every reported measure (both plain and _wtd tracks, per §20) is recomputed at that replicate's own refit cutpoint and stored in a column of a replicate-by-measure matrix. After all replicates are completed, each measure's confidence bound are computed by linear interpolation between order statistics of its own column at $\alpha = (1 - \texttt{cilevel}/100)/2$ (default `cilevel`=95, giving $\alpha = 0.025$) for the lower bound and $1 - \alpha$ for the upper bound.

When `loo` is also specified, LOO-side confidence intervals use a more efficient bootstrap out-of-bag (OOB) scheme rather than literally rerunning the full n-fold exact leave-one-out procedure `cireps()` times: on each bootstrap replicate, whichever original observations were *not* drawn into that replicate's resample serve as that replicate's own held-out test set, evaluated against that replicate's refit model. This reuses the same `cireps()` refits already being done for the training-sample CIs rather than requiring a second, separate set of `cireps` $\times\, n$ refits.

When `gen()` is also given, the identical set of bootstrap replicates is additionally partitioned by `gen()` group and each group's own measures are bootstrapped from the same replicate set – no extra refits are performed for the `gen()` breakdown's confidence intervals.

When `cross()` is also given, each bootstrap replicate resamples only from the training sample's own rows; a holdout observation is therefore always "out-of-bag" on every replicate and is evaluated, on every replicate, against that replicate's training-refit model – giving each holdout sample its own bootstrap confidence interval built from the same replicate set as the training sample's own CIs.

5 cross()

Stata syntax: `cross(varname [, train# [, hold#]])`

`cross()` reproduces original ODA's HOLDOUT mechanism (normally driven by separate holdout data files) using a single sample-identifier variable within one dataset. Every observation whose `varname` value is not in the requested training/holdout set is disregarded. The model is fit exactly once, using only the rows where `varname=train#` (or, if `train#` is omitted, the numerically smallest observed value) – with `trainreps()`, `loo`, and `looreps()` all running against that training subset only, matching original ODA's own structure where these diagnostics are computed from the training sample and then applied to a fixed model. Each holdout sample (`hold#`, or, if omitted, every other observed value of `varname`) is then classified by that same fixed, already-fit model – no refitting per holdout sample – and its own performance is tallied the same way `gen()` tallies a subgroup (§9), giving a Train/LOO/Hold three-column report per holdout sample when `loo` is active.

When `trainreps()` is also given, `cross()` additionally runs a cross-sample significance test on the training sample and on every holdout sample: this is specifically a *fixed, non-searched, identity-mapping* permutation test – the actual class labels are shuffled while the model's fixed predicted labels are held constant, and the count of agreement between shuffled-actual and fixed-predicted at or above the observed level is tallied over `trainreps()` iterations. This is deliberately not a free categorical search of the kind `cat` performs elsewhere: a free search would find an equally good relabeling in permuted data just as easily as in the real data, which would make it structurally incapable of ever rejecting the null regardless of how strong the real relationship is. If `sidak(#)` (§17) is specified, this test's own auto-computed Sidak factor of 1+(number of holdout samples) is overridden.

`gen()` cannot be combined with `cross()` in the same run. `ci` can be specified in combination with `cross()`, per §20's description of its out-of-bag treatment of holdout rows.

6 degen

Stata syntax: `degen`

`degen` applies to a binary classvar against a continuous attribute only. By default, the candidate cutpoint list is built strictly from midpoints between consecutive distinct attribute values, so every candidate cutpoint necessarily places at least one observation on each side. When `degen` is specified, two additional "outer" cutpoints are appended – one below the smallest observed attribute value and one above the largest – so that the all-one-class solution (everyone predicted into a single class) becomes a legal candidate and can win the search if its score is actually the best available.

7 direction()

Stata syntax: `direction(< | > | lt | gt)` (binary classvar); `direction(< | > value list)` (K-category classvar, continuous attribute only).

By default, the search (§**??**) evaluates both possible directions at every candidate cutpoint (class 1 predicted above the cutpoint, or class 1 predicted below it) and lets the tied-best score decide which one is reported. `direction(<)` or `direction(lt)` restricts the search to only the direction where

the higher class value is predicted on the high-attribute-value side; `direction(>)`/`direction(gt)` restricts it to the other direction.

For a K-category *classvar*, `direction()` takes a full ordering of every class value, e.g. `direction(< 2 3 1 4 5)`. This is converted into a fixed left-to-right index ordering resulting in a one-dimensional dynamic program over $K-1$ boundary cutpoints in $O(n_u \times K)$ time, where n_u is the number of distinct attribute values, that finds the optimal boundary placement subject to the class order already being fixed. This is fundamentally cheaper than the free-order search described in §**??**, because the combinatorial question of *which* class goes in which segment has already been answered by the user.

8 dots

Stata syntax: `dots`

`dots` prints a progress dot for each iteration of whichever Monte Carlo permutation test(s) (`trainreps()`, `looreps()`, `cross()`'s significance test) or leave-one-out fold (`loo`) is currently running.

9 gen(varname)

Stata syntax: `gen(varname)`

When `gen()` is specified, the model is fit *once*, on the whole sample (exactly as it would be without `gen()`) then, for each distinct value of `varname`, the already-computed predicted-vs-actual pairs are subset to that group's rows and a fresh confusion matrix (and hence PAC, PV, Overall Accuracy, and Effect Strength) is tallied from that subset – both a plain and a _wtd version, per §20. No refitting of any kind happens per group; `gen()` answers "how well does the one model that was fit on everyone perform within this particular subgroup," not "what is the best model for this subgroup." The same subset-and-tally step is applied to the leave-one-out predictions too, when `loo` is also given, again with no additional refitting beyond the n LOO folds already being computed regardless of `gen()`.

For a binary classvar specifically, when `loo` and `looreps(#)` are both given, the per-group breakdown's own significance test switches from a per-group Fisher's exact test to a per-group LOO permutation test: class labels are shuffled globally (the same shuffle used for the top-level `looreps()` test), the full n-fold leave-one-out refit is rerun once per permutation, and each group's own permutation p-value is tallied from that same set of `#` permutations, restricted to that group's rows, against that group's own observed LOO accuracy – one shared permutation loop drives every group's p-value, not a separate loop per group. This does not extend to a K-category classvar, where the per-group breakdown carries no significance test regardless of `looreps()`.

10 loo

Stata syntax: `loo`

`loo` performs a leave-one-out procedure, in which the program loops over every one of the n training observations, refitting the *entire* search on the other $n-1$ observations each time (respecting

`direction()`, `nopriors`, `degen`, and the search-weighting behavior of `wt()` exactly as in the training fit), classify the single held-out observation with that fold's own refit model, and accumulate the result into one pooled confusion matrix across all n folds – both a plain-weight and a real-weight version of that pooled matrix are accumulated in the same pass, per §20. The reported LOO sensitivity/specificity/PAC/PV/ESS figures are computed from this single pooled matrix, *not* as an average of n per-fold accuracy numbers.

11 looreps(#)

Stata syntax: `looreps(#)` (requires `loo`)

For a binary classvar, a permutation procedure is nested inside the leave-one-out refit: class labels are shuffled once per iteration and the entire n-fold LOO procedure (§10) is rerun on the shuffled labels, producing one pooled LOO accuracy figure per permutation; the top-level p-value is the proportion of the `#` permutations whose pooled LOO accuracy meets or exceeds the observed one, and (per §9) the same permutation loop also drives each `gen()` group's own LOO p-value when `gen()` is active. This replaces the default Fisher's exact test.

For a K-category classvar, a nested permutation procedure is implemented: for each of `#` permutation iterations, the class labels are shuffled once, and then a *complete* n-fold leave-one-out procedure (as in §10, itself n refits) is run on the shuffled labels, producing one pooled LOO accuracy figure for that permutation. The p-value is the proportion of the `#` permutations whose pooled LOO accuracy meets or exceeds the real, observed pooled LOO accuracy. Total cost is $O(\texttt{looreps} \times n)$ refits, which can be substantially more expensive than `trainreps()`'s $O(\texttt{trainreps})$ refits. No per-group significance test is computed here regardless of `gen()` (§9).

12 missing(#)

Stata syntax: `missing(#)`

A single missing-value code `#` (e.g. -99) is checked against `classvar`, `attrvar`, the `wt()` variable (if specified), and the `gen()` variable (if specified). Any observation where any one of those equals `#` is excluded from the analysis.

13 name(string)

Stata syntax: `name(string)`

`name()` prints `string` as a title line ahead of the results, to help tell multiple `oda` runs apart in the same log or do-file. This affects only how results are displayed, never what is computed – it has no bearing on which model is fit or how any reported measure is computed.

14 nopriors

Stata syntax: `nopriors`

`nopriors` selects which of two mean-accuracy formulas the search maximizes at every candidate cutpoint. Let TP, TN, FP, FN be the (possibly weighted, see §20) confusion-matrix counts at a

candidate cutpoint, $W_1 = TP + FN$ and $W_0 = TN + FP$ the total weighted mass of each class, sens $= TP/W_1$, spec $= TN/W_0$. By default (priors on):

$$\overline{\text{acc}} = \frac{\text{sens} + \text{spec}}{2} \times 100, \tag{3}$$

which weights each class's own accuracy equally regardless of how many observations that class actually has. With `nopriors`:

$$\overline{\text{acc}} = \frac{TP + TN}{W_1 + W_0} \times 100, \tag{4}$$

plain overall accuracy, which lets a numerically larger class dominate the objective if the classes are imbalanced. Both formulas are evaluated in `oda` as vectorized operations across every candidate cutpoint simultaneously (§**??**), not one cutpoint at a time. See §20 for the third variant of this formula that applies specifically when `wt()` and priors are both active together.

15 `primary()` and `secondary()`

Stata syntax: `primary(maxsens | meansens | genmean | balanced | samplerep | distance | random | sens # | gensens #)`, `secondary(... same list ...)`.

The core search (§**??**) always finds the maximum ESS score across every candidate cutpoint/direction (or category split) combination, then collects every candidate whose score is within 10^{-8}–10^{-9} of that maximum into a tied set. If exactly one candidate is tied-best, `primary()`/`secondary()` never run at all – there is nothing to break a tie between. `secondary()` is only called if, after `primary()` is applied, more than one candidate is *still* tied (the code checks `rows(tied)>1` a second time before calling the secondary tie-break).

`maxsens`, `meansens`, `genmean`, `balanced`, `samplerep`, and `distance` are implemented as one identical rule, despite their different names: for every tied candidate, the precision of whichever class is predicted on the low-attribute-value side of that candidate's cutpoint is comnputed:

$$\text{metric} = \begin{cases} \dfrac{TP}{TP + FP} & \text{direction} = -1 \text{ rows} \\ \dfrac{TN}{TN + FN} & \text{direction} = +1 \text{ rows} \end{cases} \tag{5}$$

and whichever candidate(s) maximize the metric (again within a small tolerance) is retained. Naming a different one of these six keywords therefore never changes the result. `random` is the only keyword in this group with genuinely different behavior: it draws one candidate uniformly at random from the tied set (`uniform(1,1)` indexing into the tied matrix), so it can matter in `secondary()` even after `primary()`.

`sens #`. For a binary classvar, `sens #` is resolved once, at the start of the fit, into a specific side of the cutpoint to maximize (whichever side corresponds to class value `#`), and then maximizes that side's own sensitivity among tied candidates – this is a genuinely distinct rule from the six above, not folded into them. For a K-category classvar, `sens #` is validated (must name an observed class value) but not given distinct behavior; it currently falls back to the same aggregate rule as the other keywords.

`gensens #` selects among maximin-tied candidates, whichever one maximizes group `#`'s own accuracy. For every maximin-tied candidate, each `gen()` group's own accuracy is retained (not just

the min and mean across groups), and `gensens #` ranks the tied set by that specific group's own accuracy rather than falling back to the `genmean`-style aggregate rule used by every other keyword.

Default. `maxsens` is the default when `priors` is on and `meansens` otherwise; because both those keywords reduce to the identical rule in `oda`, `oda`'s default (`maxsens` unconditionally) produces the same numeric result either way.

16 seed(string)

Stata syntax: `seed(string)`

`seed()` sets a user-defined seed before any of `oda`'s random draws run – the Monte Carlo permutation tests (`trainreps()`, `looreps()`, `cross()`'s significance test), the bootstrap resampling in `ci`, and `primary(random)`/`secondary(random)`'s tied-candidate draw.

17 sidak(#)

Stata syntax: `sidak(#)`

`sidak(#)` applies Sidak's post-hoc adjustment for multiple testing, to an already-computed Monte Carlo p-value:

$$p_{\text{adj}} = 1 - (1 - p)^{\#}. \tag{6}$$

This is applied to the training-sample `trainreps()` p-value, to the `looreps()` p-value if given, and, when `cross()` is also active, to `cross()`'s own cross-sample significance test p-values. In the case of `cross()`, specifying `sidak()` explicitly overrides an auto-computed default of 1+ the number of holdout samples. `sidak()` requires `trainreps()` to be given at all – there is no p-value to adjust otherwise.

18 store(string)

Stata syntax: `store(string)`

`store(string)` writes a plain-text copy of everything the command displays – the ODA model table, performance tables, and any `gen()`/`ci` sections – to the file named by `string`. This affects only how results are saved, never what is computed – it has no bearing on which model is fit or how any reported measure is computed.

19 trainreps(#)

Stata syntax: `trainreps(#)`

When `trainreps(#)` is specified, a classic label-permutation Monte Carlo procedure is implemented in which the the observed class labels are shuffled `#` times (the attribute values are left in place). The *entire* search (including `direction()`, `nopriors`, `degen`, and, if `wt()` is specified, the same weighted search objective) is rerun from scratch on the shuffled labels, and the count of permutations whose best ESS score is $\geq$ the real, observed ESS score is tallied. The reported p-value is that count divided by `#`. Every permutation is a full independent refit – there is no shortcut that reuses partial results across permutations.

20 wt(varname)

Stata syntax: `wt(varname)`

This is `oda`'s most involved option, so this section covers, in order: what kind of weight it is, exactly how it enters the search objective (including its interaction with priors), how it propagates to every reported measure, and – because this has been a specific point of interest – exactly how it does *not* interact with resampling in `loo` and `ci`.

What kind of weight `wt()` is

`oda`'s `wt()` behaves exactly like Stata's `iweight` (see help weight): it is a per-observation multiplicative factor that enters every accuracy computation and the search objective directly, and *never* changes how observations are drawn during resampling (bootstrap or leave-one-out).

How `wt()` enters the search itself

Everywhere that `oda` computes a weighted count (TP, TN, FP, FN), each observation's class indicator is multiplied by its own weight before cumulative-summing, so `wt()` genuinely changes which cutpoint is found to be optimal, not merely how the winning cutpoint's performance is reported afterward. An observation with a larger weight counts more toward whichever class-accuracy sum it belongs to at every candidate cutpoint evaluated.

The priors + weights interaction

When both `wt()` and priors are active (the default; not `nopriors`), the search uses the exact same 50/50 formula as the plain-priors, no-weights case in §14 – $\overline{\text{acc}} = (\text{sens} + \text{spec})/2 \times 100$ – with sens and spec computed from whichever weight vector was actually passed into the search: `wt()`'s real weights when specified, or an implicit weight of 1 per observation otherwise. `wt()` therefore still fully participates in the search whenever its weights genuinely vary from one observation to the next – it changes the weighted TP, TN, FP, FN sums that sens and spec are computed from, and so can change which cutpoint scores best – but priors' own sensitivity/specificity balancing is always the same flat 50/50 average, never blended by each class's raw sample share.

How `wt()` propagates to every reported measure

Once the (possibly weight-influenced) cutpoint or category grouping is determined, `oda` computes the *entire* set of performance measures twice at the identical fitted cutpoint: once with a constant weight vector of one (every observation counted once) and once with the real weight vector. This happens regardless of whether the search itself was weighted, and regardless of `nopriors` – the plain track is always present, and the _wtd track is added whenever `wt()` is specified, whether or not that same weight was used to influence the fit. The same dual-track computation is applied identically to the training-sample confusion matrix, the leave-one-out pooled confusion matrix (§10), and every `gen()` group's breakdown (§9).

`wt()` and resampling

The bootstrap resampling loop inside `oda` draws each replicate's rows with a plain uniform draw with replacement over row *indices*, with no reference anywhere in that expression to the weight variable. Every observation in the training sample has exactly the same probability, $1/n_{\text{train}}$, of

being drawn into any given bootstrap replicate, irrespective of its `wt()` value. The same is true of the leave-one-out loop, which holds out each observation exactly once regardless of weight, and of every permutation test's label-shuffling step which shuffles the class labels uniformly at random and is likewise blind to `wt()`. Thus, `wt()` has only two effects; (1) altering the search objective, and (2) altering the computed value of every _wtd-suffixed reported measure, as a multiplicative factor applied after the resample/fold is already chosen.

References

Yarnold PR, Soltysik RC. Theoretical distributions of optima for univariate discrimination of random data. *Decision Sciences* 1991;22:739–752.

Yarnold PR, Soltysik RC. *Optimal Data Analysis: A Guidebook with Software for Windows.* Washington, DC: APA Books, 2005.

Yarnold PR, Soltysik RC. *Maximizing Predictive Accuracy.* Chicago, IL: ODA Books. DOI: 10.13140/RG.2.1.1368.3286, 2016.

Linden A, Yarnold PR. Using machine learning to assess covariate balance in matching studies. *Journal of Evaluation in Clinical Practice* 2016;22:848–854.

Linden A, Yarnold PR. Using machine learning to identify structural breaks in single-group interrupted time series designs. *Journal of Evaluation in Clinical Practice* 2016;22:855–859.

Linden A, Yarnold PR, Nallomothu BK. Using machine learning to model dose-response relationships. *Journal of Evaluation in Clinical Practice* 2016;22:860–867.

Linden A, Yarnold PR. Combining machine learning and matching techniques to improve causal inference in program evaluation. *Journal of Evaluation in Clinical Practice* 2016;22:868–874.

Linden A, Yarnold PR. Combining machine learning and propensity score weighting to estimate causal effects in multivalued treatments. *Journal of Evaluation in Clinical Practice* 2016;22:875–885.

Linden A, Yarnold PR. Using machine learning to evaluate treatment effects in multiple-group interrupted time series analysis. *Journal of Evaluation in Clinical Practice* 2018;24:740–744.