Shiny App: Latent Class Analysis

Simon Kucharsky (11391111)

Contents

The full code and documentation can be found at https://github.com/Kucharssim/LCAapp

What is LCA?

Latent Class Analysis (LCA) is a statistical technique, a special case of mixture modelling. The focus of this approach is to find underlying (latent) categorical structure in the data which explains dependencies among the observed categorical variables. This method can be used if we want to find a distinct *classes* (groups) of individual cases which have similar response patterns across all observed variables.

To ilustrate the idea, let's imagine a simple problem. Suppose you asked random people whether they watched a 2017 Europa League final, if they put chocolate sprinkles on their toast and whether they use bike to travel around. You observed the following frequencies:

Final	Sprinkles	Bike	Frequency
No	No	No	187
No	No	Yes	285
No	Yes	No	94
No	Yes	Yes	170
Yes	No	No	76
Yes	No	Yes	94
Yes	Yes	No	23
Yes	Yes	Yes	71

If you applied some test (for example, log-linear analysis) you would find that people who watched the final also tend to eat sprinkles and use the bike. Now, here's the question: Should we conclude a genuine relationship between the variables? Probably every reasonable person would doubt that eating sprinkles makes you watch football, or that there is any other direct relationship between any pair of the variables. In this case, it is likely that the relationship between the variables is spurious, caused by other factor(s) we didn't observe. For example, one explanation could be that this relationship might be caused by whether a person is dutch or not (dutch people are more likely to watch a final game featuring dutch club, they are also more likely to put sprinkles on their toast and drive a bike).

And this is where LCA comes in handy, becase using this technique, we *assume* that the relationship between the variables is spurious and caused by mixture of different populations with different response probabilities. By conditioning on the group membership, the variables should become independent. (But we can also apply LCA when we don't observe any relationship between the variables in the first place.)

To apply the LCA model, we need to specify how many different latent classes (groups) are in the data and then we need to estimate two sets of parameters:

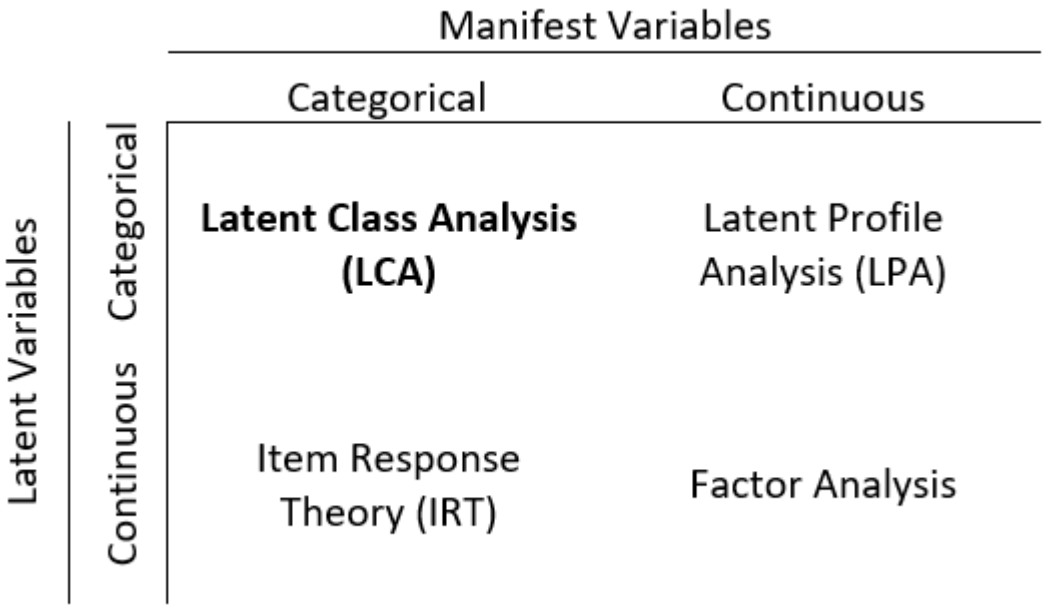
1) The relative sizes of the latent classes
2) The conditional probabilities of responses on all variables given class

If we applied the LCA to the data above estimating two classes, we could find, for example, that about 1/2 of the people in the sample have about 0.3 probability that they watched the final, 0.5 probability of eating sprinkles and 0.7 probability of using a bike. The other half has a 0.2 probability of watching the final, 0.2 of eating sprinkles and 0.5 of using a bike. Although it is a matter of intepretation what we think about this result (especially if we conclude that the first group is formed mainly by dutch people), it is clear that the LCA approach offers us completely different view on the data - rather than making claims about the variables on the populational level, we can find some groups with different probabilities of certain behavior.

Before we go ahead to show how the LCA works in detail, let us summarize what we had and what we did.

1) We had data of three categorical items.
2) Assumed a latent categorical structure which could explain the relationship between the variables.
3) By applying LCA, we were able to estimate the relative size of the two classes and the response patterns conditioned on the class membership.

To put into a context, LCA is a special case of a latent variable model. It is characterized by inferring a categorical latent structure from a categorical data, which distinguishes it from Latent Profile Analysis, Factor Analysis and Item Response Theory:



But now, let's briefly explain how the method works.

How is LCA done?

Parameter Estimation

In the previous section we showed that we can estimate relative sizes of the latent classes and their conditional probabilities of responses on a set of manifest variables. But how to come to the best estimates? There are many possible solutions to solve the problem, we will focus on a basic one, called Expectation-Maximisation algorithm (EMA).

Assume for a moment that when you were asking about the football match, sprinkles, bike, you also asked whether the respondent is dutch or not. In this case, nothing would be easier than:

1) Compute the relative size of the dutch and non-dutch group in the sample (just by counting how many people are dutch from the whole sample).
2) The conditional probabilities of the responses (by splitting the data by the nationality and compute the relative frequencies of answers to the questions for both groups separately).

But how to compute it without this information? At first sight, it seems the problem is tautological, because without knowing the conditional probabilities of the answers, we cannot decide who belongs to which group, and without this information, we cannot compute 1) and 2). Or can we?

For these problems is EMA very useful, because it does not require neither of the information to give the estimates. The magic comes with two steps, each using (imprecise) estimate for one of the set of information to get a better estimate of the other. The procedure goes like this:

1) *Expectation.* We start by a random guess of the conditional probabilities of answers given a class membership. Using those, we can compute for each case the likelihood of belonging to all classes given its responses.
2) *Maximisation.* Using the probabilities of class membership as weights, we can update the conditional probabilities of response patterns.

It is ensured that these two steps will get closer towards some local optima. We can then iterate over those two steps until the estimates do not change (much) anymore. Usually, instead of checking whether the parameters themselves do not change, we just compute the (log)likelihood of the data under the current parameters and stop whenever the change of the (log)likelihood is smaller than some arbitrary threshold.

Because this approach ensures only that the model converges to some local optima, the estimates are dependent on the starting guess of the parameters. Because of this, it is advised to run the estimation multiple times to find the global optimum.

Degrees of freedom, number of parameters and identifiability

When estimating the models, we should also pay attention which models we can actually compute with the information we have. Similarly to CFA (or more generally SEM), we cannot estimate more parameters than we have variables. In the LCA context, the number of variables is equal to the number of cells in the multidimensional contingency table minus one. In the example above, we have a $2 \times 2 \times 2$ table, which means we have 7 variables. In case of 4 binary items, we would have 2^4 - 1 = 15 variables, and so on.

Remember that when we estimate a certain model, we need to estimate two sets of parameters:

1) Proportions of the classes
2) Conditional probabilities of the responses given the class membership

For k classes, we have to estimate k-1 proportions (as we know they have to sum to 1). The conditional probabilities have to be computed for each class and each item, and because the probabities of the responses for each item also sum to one, we have to estimate $K \sum_{n=1}^{N}(L_n - 1)$ probabilities. The standard formula for computing the degrees of freedom (for unconstrained LCA models) is therefore:

$$df = \prod_{n=1}^{N} L_n - 1 - (K-1) - K\sum_{n=1}^{N}(L_n - 1)$$

Where L_n is the number of levels on the n-th item, N the number of items and K is number of classes. For the example above, the df would be:

$df = (2^3 - 1) - (2 - 1) - 2 \times (1 + 1 + 1) = 0$

So in this case, we would have saturated model.

If we get a negative degrees of freedom, we have an unidentified model, which means that we can get the same fit with different parameters (in other words, the likelihood function would have an optimum within a multidimensional space, not a single point).

Usually, we aim to have a positive df.

Model evaluation

After we estimated the model, we need to evaluate how it fits the data:

- χ^2 statistic tells us how well the model fits the data overall, meaning how big differences there are between the observed and implied multidimensional contingency table. The formula is:

$$\chi^2 = \sum_{i=1}^{C} \frac{(F_i - f_i)^2}{f_i}$$

Where F_i is the observed count for the cell C_i in the multidimensional contingency table and f_i is the implied count in the cell which is given by the model. This statistic has approximally χ^2 distribution with the degrees of freedom equal to the degrees of freedom of the model. The fit can be assessed by computing the p-value associated with the statistic and degrees of freedom, and when it does not go below some arbitrary (usually 0.05) threshold, we cannot reject the model. The problem with this approach is that this test cannot really tell us that the model is good, it can only indicate whether we can or cannot reject it. It is also well known that with small samples, the χ^2 approximation is imprecise leading to a low power to reject bad models, and on the other side, it tends to have a high power to reject even relatively good models with large sample sizes.

- G^2 (also called likelihood ratio or deviance statistic) is similar to the χ^2 with comparing the implied and observed frequencies. The difference is that instead of taking the differences beetween them, we calculate their ratios and log-transform them:

$$G^2 = 2\sum_{i=1}^{C} F_i log(\frac{F_i}{f_i})$$

This statistic can also be used to compare different (nested) models, because taking the difference between G^2 of both models have a χ^2 distribution with degrees of freedom equal to the difference of the degrees of freedom of the both models.

- **Information criteria.** For the purpose of model selection, we can also use so called information criteria, which is a family of statistical indices that compare the likelihood of the data under the model and penalize it by its complexity (which on a conceptual level prevents overfitting as more complex model are always capable of better fit, even if they fit noise). The two commonly used criteria are Akaike Information Criterion (AIC) and Bayesan Information Criterion (BIC). The formulas for those two indices are:

$$AIC = -2log(L) - 2k$$

$$BIC = -2log(L) - log(n)k$$

Where L is the likelihood, k the number of estimated parameters and n is the sample size. Usually we want to select the model with the lowest AIC and BIC.

- **Entropy of class separation.** This index does not tell anything about the model fit; however, it is useful for getting an insight into the confidence we have in the class assignment. In other words, if the conditional probabilities of certain responses are very similar for some classes, we cannot be sure under which class a (possibly) large number of cases belongs. The Entropy of class separation ranges from 0 to 1. High Entropy indicates the classess are well separated, low Entropy could actually mean some classes are so similar that it is questionable whether it makes sense to distinguish between them (but the decision must be made on other characteristics than the Entropy alone). If we want to use the modal posterior assignment, it is usually recommended to have a high Entropy (above 0.7 or 0.8), otherwise the model has a large amount of classification error. The formula for the Entropy is:

$$E = 1 + \frac{1}{Nlog(K)}[\sum_{i=1}^{N}\sum_{k=1}^{K} P(C = k|U_i)log(P(C = k|U_i))]$$

Where C is the latent class, N the sample size and U_i is the vector of all the answers on the items for case i.

How to work with the App?

The final version of the App is stored on GitHub repository `kucharssim/LCAapp`. The App is dependent on these packages: `shiny`, `parallel`, `gtools`, `DT`, `reshape2`, `dplyr`, `ggplot2`, `plotly`.

To initiate the app, you can open your R session and execute the following code:

```
if(!require('shiny')) {install.packages('shiny'}
if(!require('parallel')) {install.packages('parallel')}
if(!require('gtools')) {install.packages('gtools')}
if(!require('DT')) {install.packages('DT')}
if(!require('reshape2')) {install.packages('reshape2')}
if(!require('dplyr')) {install.packages('dplyr')}
if(!require('ggplot2')) {install.packages('ggplot2')}
if(!require('plotly')) {install.packages('plotly')}

shiny::runGitHub("kucharssim/LCAapp")
```

The collection of `if` statements will check whether you have the packages installed and install them if you miss some. You can also install the packages manually, if you wish to install them into a different folder than the default. The last command initiates the App, so you can run just that one if you know the packages are already installed.

Loading the data

When you initiate the App, you should see that there is already a preloaded example dataset in case you just want to try out how the LCA works. This dataset comes from a simulation of 400 cases as a mixture of three classes with 6 binary items:

Class 1) `n = 40`; probability of answering 1 is ~ `90 %` for all items

Class 2) `n = 80`; probability of answering 1 is ~ `15 %` for all items

Class 3) `n = 280`; probability of answering 1 is ~ `75 %` for items 1-3 and ~ `25 %` for items 4-6.

Try out the app if you can retrieve those three groups!

If you want to use your own data, beware that the current version can handle only column separated files (`.csv`) with the following specifications:

1) The first row should contain the names of the variables
2) The values can be numeric or string; if the strings contain multiple words separated by columns or spaces, they should be wrapped into a double quotes
3) The current app cannot handle '`NA`'s. If you have NA values in variables you want to include into the model, the app would think the `NA` is just another level of the variable (so for the time being, use only data with complete cases).

Otherwise, there is no warranty that the app works properly.

When you load the data, you should see it under the `data` panel; under the table with the raw data, you should see a summary per each column in the data.

Model definition

After you load the data, you can estimate the model(s). This is done in 4 steps:

1) *Select the variables.* By clicking on the `data table`, you can highlight columns (indicating variables) you want to include into the model. If you don't highlight anything, the app will use all variables. You should also select more than one column. You can check which variables will be used by scrolling down to see the variable `summary`; the variables you did not select should dissapear.

2) *Model selection.* By clicking on the dropdown menu `Number of classes`, you can specify which model(s) you want to estimate. The app will automatically recognize how many degrees of freedom you have with the selected data and won't allow you to fit unidentified model. If you have a lot of variables, the maximum number of classes is set to 30. Otherwise, you may estimate as many models you want.

3) *Number of replications.* By changing the number of in the numeric field, you can fit multiple replications of the same models to be able to check for convergence to the global optimum (see the next section). The default is 10, minimum 2, with undefined maximum constraints (but be aware that selecting a high number of replications could lead to a long time for the model estimation).

4) *Selecting the tolerance.* You can also change the tolerance level for the maximum likelihood estimation. The default $1 \times e^{-5}$ should do fine for most of the problems, but you may want to change it under some circumstances described in the next section. The minimum is $1 \times e^{-10}$, maximum $1 \times e^{-3}$.

After you specified the model(s), click on the button below (`Estimate Models`) to fit the models. On the bottom right, you should see a progress bar indicating how much time the app still needs.

Checking the convergence

After you run the estimation, you should check whether the individual model(s) converged to the global optimum. On a second tab (`Model diagnostics`), you can see how many times of the total number of replications you were able to replicate the highest maximum log-likelihood of the data under the model. In case your best fit(s) did not replicate, you may want to increase the number of replications or the tolerance level by inspecting the values of the maximum log-likelihood for individual replications:

1) If the values of log-likelihood per model differ significantly (by amount of units or decimals), try to increase the number of replications as that could mean the log-likelihood function has a lot of local maxima.

2) If the values of log-likelihood per model differ slightly, you can try to decrease the tolerance level, as the log-likelihood function may be too flat for the current level and stops the algorithm before it reaches the minimum.

Because the app fits the individual replications in parallel, increasing the number of replications may not lead to a significant increase of the computational time. However, fitting a lot of replications can still lead to somewhat long waiting time. Decreasing the tolerance level always leads to increased computational time and thus it is advised to increase the number of replications instead when you run into convergence problems. If you still cannot replicate the best fit even you changed the number of replications and tolerance, that could mean the model has a lot of optima, which could lead to questioning whether the model is not overly complex for the data at hand.

Comparing the models

On the next tab `Model comparison`, you can compare the different models you estimated. For each individual model, you will see the χ^2 and G^2 statistic, degrees of freedom, number of estimated parameters, AIC, BIC and Entropy of class separation. The app will automatically highlight the model with the lowest BIC (but note you should pay attention also to the absolute fit indices to check if you are not actually selecting among very bad models).

By clicking on the table, you can select the model you want to export and interpret. By clicking on the last tab `Details`, you can see some information about the selected model. Note that most of it is used in the `Output` section, so you may want to skip this tab.

Interpreting the results

In the `Output` section, you can see three tabs showing you different aspects of the selected model:

1) `Plots` In this tab, you will see graphs of the resulting model parameters. The first graph shows you the relative sizes of the classes. The second tab shows you the conditional probabilities of certain responses per item, given class membership. By clicking on the box `group by items`, you can rearange the plot so you can compare the probabilities per items. By clicking on the legend in the plot, you can suppress some levels, which may help you to understand the results.

2) `Parameter Estimates` Here you can see a table with the estimated parameters. This is the same information as it is visualised in the graphs, but here you can download it as a `.csv` file in case you want to work with the results further.

3) `Class membership` You will see a table which shows you the probability of being in each class for every case in the data. The last column indicates the class membership selected by the modal probability assignment. You can download the table which will export a `.csv` file, adding k+1 columns to your original data, where k is the number of classes (for each class one column) and the class membership assignment. Note that for working with the class assignment, the model should have high Entropy, otherwise it is advised to work with the probabilistic class membership.

If you run the app from your RStudio session in a window or in a pane, the download button may not work. For downloading the results, run the app externally in your browser.

How the app works

The app relies on the main function, `emLCA`. This function takes in the data[1], the number of classes to estimate, and can get some more options (starting values, tolerance level, and so on). This function uses the EM algorithm to estimate the parameters. For the Expectation step, it calls function `assignProb`, which computes the probability of the class membership for each case in the data, given the responses and the current parameter values. For the maximisation step, it just computes the average class membership probability to update the estimate of the class proportions and uses `updateTheta` function to update the conditional probabilities of responses given the class membership. It iterates over those two steps in a `while` loop and ends whenever the log-likelihood of the data under the current parameter values do not change (within the tolerance level). All the functions used in the `emLCA` function are stored in the `LCA.R` file.

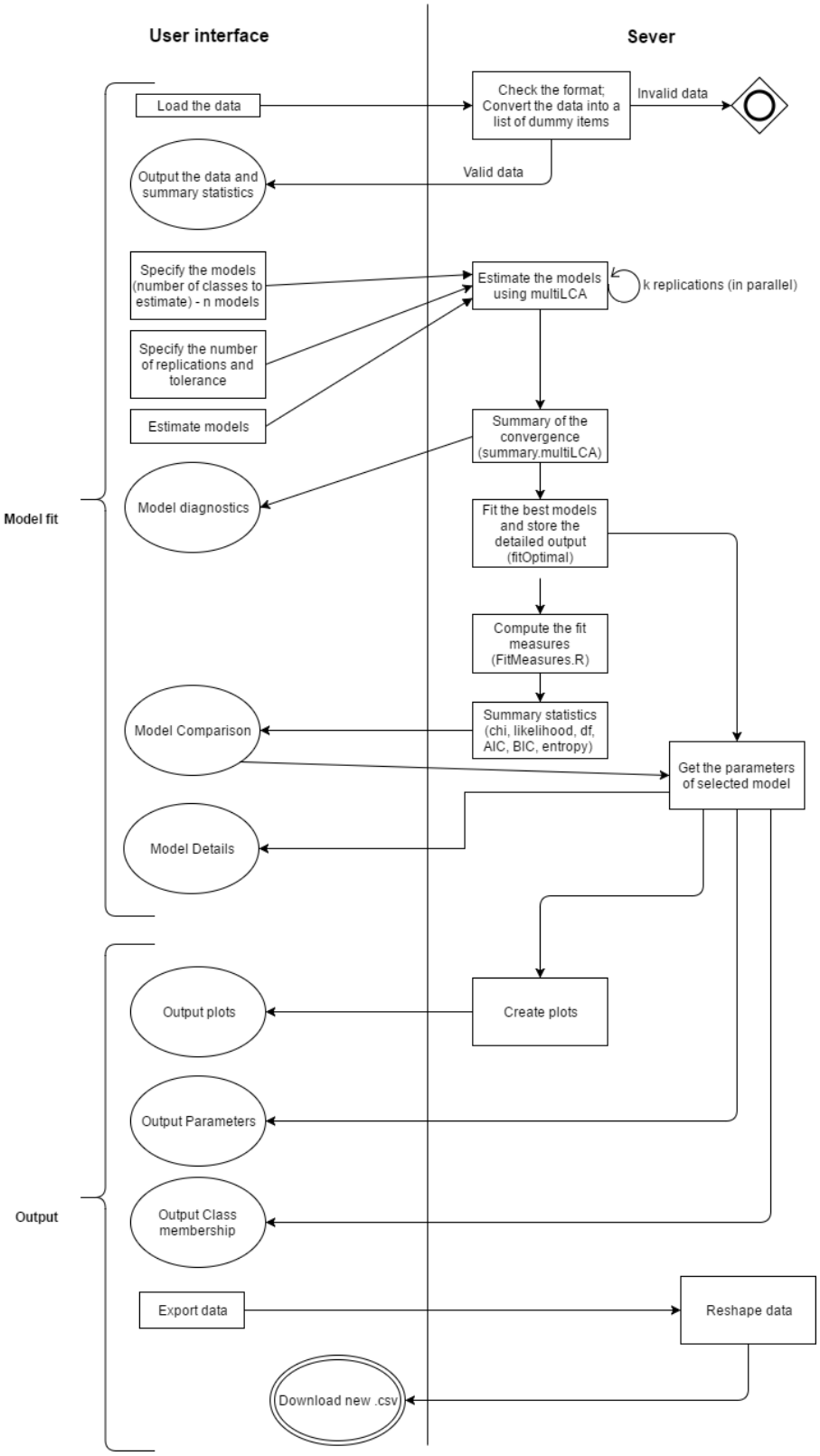
This function is used several ways in the app: First, it is wrapped into a `multiLCA` function, which takes in a vector of the desired models (numbers of classes to estimate) and the number of replication per each model (it computes the replication in parallel using the `parLapply` function from package `parallel`). This function then outputs the log-likelihood, number of iterations of the estimation and starting values for all estimated models. This information is then again passed into the `summary.multiLCA` function which is used for checking the convergence of the models; and then the best starting values (in terms of log-likelihoods and number of iterations) are used to estimate the final models (using `fitOptimal` function, which again calls `emLCA`), from which the detailed output is stored for results. This approach means the best models are actually estimated twice; but this is because for the check of convergence, we do not need all the output from the estimation, are storing them for each replication could lead to a lot of requirements for the memory capacity. All these functions are in the `multiLCA.R` file.

The functions in the file `FitIndices.R` are used for computing the fit indices of the optimal models. In the `Plots.R` are the functions for creating the plots.

The functions do not have much error control included in them (for example, the `emLCA` needs the data in a specific format, not a data frame, but it would crash in case you did not use the right format). This is because the functions are written for the purpose of the shiny App: to make sure the the functions get the right input, I used `validate` function with combination of `need` function in the `server.R` to ensure the input is right.

[1]Note that the data must be a list of dummarized items: for reshaping the data into that format, use `reshapeData()` in a `DataHangling.R` file on a matrix or data frame.

The workflow if the whole app is this:



Further reading

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum Likelihood from Incomplete Data via the EM Algorithm. Journal of the Royal Statistical Society. Series B (Methodological), 39(1), 1–38.

Do, C. B., & Batzoglou, S. (2008). What is the expectation maximization algorithm? Nature Biotechnology, 26(8), 897–899. https://doi.org/10.1038/nbt1406

Hagenaars, J. A., & McCutcheon, A. L. (2002). Applied latent class analysis. Cambridge; New York: Cambridge University Press.

Oberski, D. (2016). Mixture Models: Latent Profile and Latent Class Analysis. In J. Robertson & M. Kaptein (Eds.), Modern Statistical Methods for HCI (pp. 275–287). Springer International Publishing. Retrieved from http://link.springer.com/chapter/10.1007/978-3-319-26633-6_12