linemodels R package (v.0.5.0)

Matti Pirinen, University of Helsinki, Finland

June 18, 2025

Summary Estimation of effects of multiple explanatory variables on multiple outcome variables has become routine across life sciences with high-throughput molecular technologies. The `linemodels` R-package allows a probabilistic clustering of explanatory variables based on their observed effect sizes on multidimensional outcomes. User can specify each candidate model by three parameters:

- scale, i.e., the magnitude of the effects,
- slopes, i.e., the multiplicative relationship between the expected values of the effects on multiple outcomes, and
- correlation, i.e., the allowed deviation from the expectation.

`linemodels` provides functions to estimate membership probabilities of each explanatory variable in different linemodels, together with the mixture proportions of the linemodels, by taking into account uncertainty in the effect estimates and a possible correlation of the estimators of effects on multiple outcome variables. The package further allows for optimization of the model parameters, use of features to model the prior probabilities of each model via multinomial regression and use of Gaussian mixture models as the prior distribution of effect sizes.

Available at: https://github.com/mjpirinen/linemodels

1. Motivating example

Consider effect estimates of $n = 100$ genetic variants on two subtypes Y_1 and Y_2 of a disease (Fig. 1A). Suppose that we are interested whether each variant has

(1) similar effects on both subtypes ($\beta_1 \approx \beta_2$), or
(2) a smaller effect on Y_2 than on Y_1 ($\beta_2 < \beta_1$), or
(3) no effect at all on Y_2 ($\beta_2 \approx 0$).

(While additional models might also be of interest, we keep this motivating example simple by only considering the three models above.)

We apply `linemodels` by specifying the three models that are visualized in Fig. 1B as colored lines. Output includes variant-specific posterior probabilities of the membership in each linemodel (Fig. 1C) as well as proportion estimates for each linemodel (Fig. 1D). From panel C we learn, for example, that while variant 3 has a very high probability of belonging to M_1, the data for variant 2 are indecisive between M_1 and $M_{0.5}$. From panel D we see, for example, that about 18% (95% credible interval from 0.12 to 0.27) of the variants belong to M_0, and hence have $\beta_2 \approx 0$.

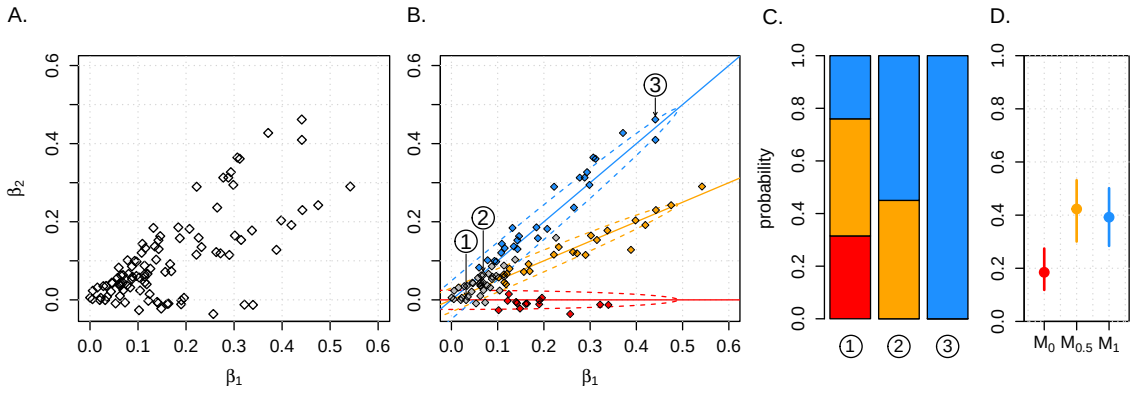


Figure 1. A. Effect estimates of $n = 100$ genetic variants on two outcome variables. **B.** Three line models shown as colored lines with their 95% highest probability regions (dashed lines). The slopes of the models are 1 (model M_1, blue), 0.5 ($M_{0.5}$, orange) and 0 (M_0, red) corresponding to the above mentioned three relationships $\beta_1 \approx \beta_2$, $\beta_2 < \beta_1$ and $\beta_2 \approx 0$, respectively. Three exemplar variants are labelled as "1", "2" and "3". **C.** Variant-specific posterior probabilities of the membership in each model for the three variants labelled in panel **B**. **D.** Posterior distribution of proportion estimates of the linemodels across all $n = 100$ variants.

Note that the analysis using `linemodels` is not a trivial distance comparison between the points and the lines since

- we account for varying uncertainty of the effects on the outcomes,
- we account for possible correlation between the effects on the outcomes, e.g., due to shared control individuals in disease studies,
- we have defined Gaussian probability models surrounding the lines.

With the `linemodels` package, it is also possible to optimize the parameters of each linemodel rather than fix them before the analysis as we have done in Fig. 1.

Next, we introduce the functions available in the `linemodels` package. Mathematical descriptions of the model and algorithms can be found at the end of this document.

2. Install `linemodels`

To install `linemodels` from GitHub, you need to install 'devtools' package (in case you don't already have it).

```
install.packages("devtools")
```

After that, install `linemodels` with the following commands

```
library(devtools)
install_github("mjpirinen/linemodels")
library(linemodels)
```

If you can't get the above working, all functions are in a single R file which you can also read in directly to R using command below

```
source("https://raw.githubusercontent.com/mjpirinen/linemodels/main/R/linemodels.R")
```

3. Specifying models

Each line model k is specified by three parameters called **scale** (s_k), **slope** ($\boldsymbol{b}_k = (b_{k2}, \ldots, b_{kM})$) and **correlation** ($r_k$), where $M \geq 2$ is the dimension of the data. When we run K linemodels jointly, these parameters are given as separate K-vectors `scales` and `cors`, and $K \times (M-1)$-matrix `slopes`. (If $M = 2$, also `slopes` can be given as K-vector.) Thus, to specify the following $K = 3$ 2-dimensional linemodels from Fig.1,

- M_1 with $s = 0.2, b = 1, r = 0.995$,
- $M_{0.5}$ with $s = 0.2, b = 0.5, r = 0.995$,
- M_0 with $s = 0.2, b = 0, r = 0.995$,

we will define vectors:

```
scales = c(0.2, 0.2, 0.2)
slopes = c(1, 0.5, 0)
cors = c(0.995, 0.995, 0.995)
```

visualize.line.models( ) We can visualize the linemodels using `visualize.line.models( )` to check that they seem reasonable. (Note: This function allows adjusting also other plotting parameters; see the help page of the function.)

```
visualize.line.models(scales, slopes, cors,
                      model.names = c("M1","M.5","M0"),
                      model.cols = c("dodgerblue", "orange", "red"),
                      legend.position = "topleft",
                      xlim = c(0, 0.6), ylim = c(0, 0.6),
                      xlab = expression(beta[1]),
                      ylab = expression(beta[2]),
                      emphasize.axes = FALSE)
```

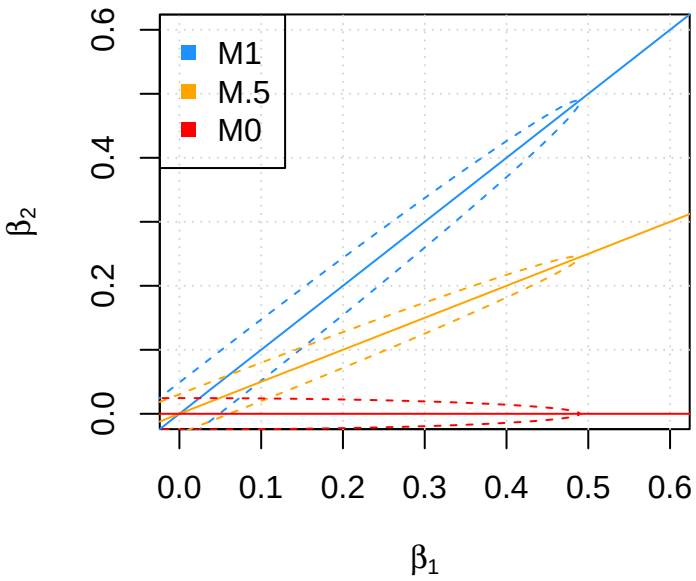


Figure 2. The scale parameter s defines the standard deviation of the zero-centered effect size distribution of the larger effect size. The correlation r defines how much deviation we allow around the exact line. Here the 95% regions determined by r are shown by the dashed curves.

More than 2 dimensions

If the data have M dimensions, a linemodel is defined by $M-1$ slopes. The slopes are always defined with respect to the outcome variable 1. Thus, to specify a line in 3-dim space that satisfies $x_2 = 2x_1$ and $x_3 = -0.5x_1$, the two slopes are $(2, -0.5)$. When there are K linemodels, the slopes are given in a $K \times (M-1)$ matrix with one row per model and slopes in the columns, starting from the second outcome variable. Let's extend the earlier 2-dimensional linemodels to a third dimension so that the slopes are $(1, 1)$, $(0.5, 0.2)$ and $(0, 0)$. We keep `scales` and `cors` at their previous values.

```
scales = c(0.2, 0.2, 0.2)
slopes = rbind(c(1,1), #matrix with K = 3 rows and M-1 = 2 columns
               c(0.5,0.2),
               c(0,0))
cors = c(0.995, 0.995, 0.995)
```

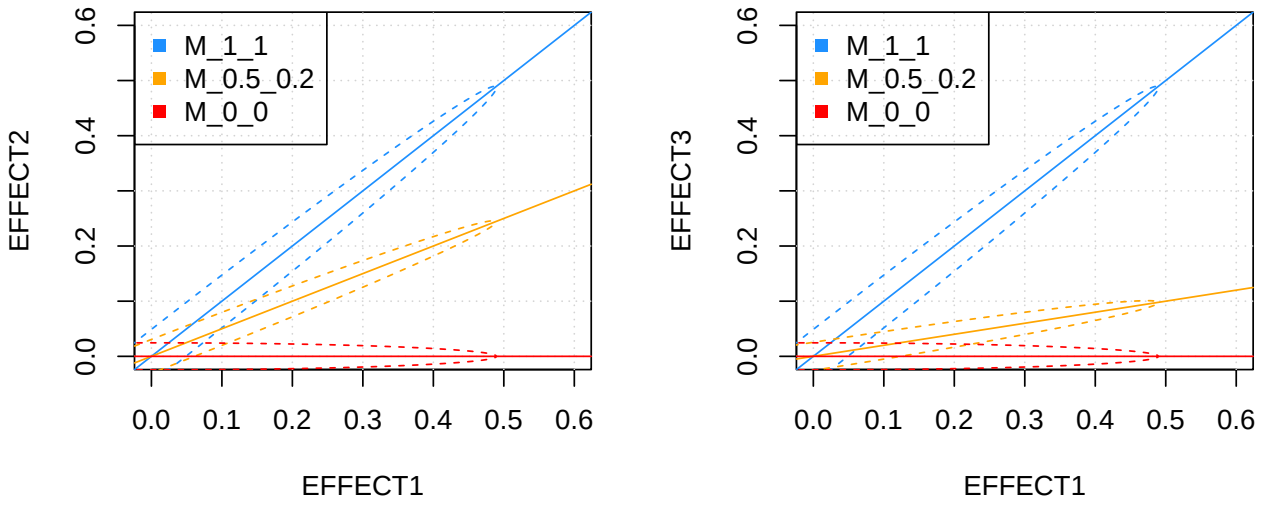
Given the `slopes` matrix, we can ask for the slope between variables i and j by `slope.for.pair(i,j,slopes)`.

```
slope.for.pair(3, 2, slopes)
```

```
## [1] 1.0 0.4 0.0
```

We can visualize multi-dimensional linemodels using `visualize.multidimensional.line.models()`. This plots 2-dimensional projections, sequentially, for pairs specified in matrix `plot.pairs`. For example, to plot the effect pairs 1 and 2, and 1 and 3 we call

```
par(mfrow = c(1,2))
visualize.multidimensional.line.models(
  scales, slopes, cors,
  plot.pairs = rbind(c(1,2), c(1,3)),
  model.names = c("M_1_1","M_0.5_0.2","M_0_0"),
  model.cols = c("dodgerblue", "orange", "red"),
  legend.position = "topleft",
  xlim = c(0, 0.6), ylim = c(0, 0.6),
  emphasize.axes = FALSE)
```



See Appendix 1 for more examples.

4. Estimating line models

We will use the example data from Fig. 1 to demonstrate the analyses. The data set is included in the package under the name `linemodels.ex1`, and can also be found from the GitHub page under /data.

```
dat = linemodels.ex1
head(dat, 2)
```

```
##       beta1       beta2        beta3        se1        se2        se3       maf
## 1 0.3628608 0.003556203  0.007179944 0.01658392 0.01658392 0.01658392 0.1766436
## 2 0.3285394 0.027833753 -0.008693408 0.01270613 0.01270613 0.01270613 0.4526828
```

The data has 3 dimensions but here we will use only the first two. For each variable, we need its effect size estimates on the first two outcomes (here `beta1` and `beta2`) and the corresponding standard errors of these estimates (`se1` and `se2`). Additionally, we will need the correlation `r.lkhood` between the estimators of the two effect sizes. Here we assume that the two estimates come from independent data sets, and hence `r.lkhood = 0`. If overlapping samples and/or correlated outcome measures were used, then a non-zero `r.lkhood` value should be specified. (See section 8 for examples.)

line.models( ) To estimate the model probabilities separately for each observation, we need to fix the prior probabilities of the linemodels. Let's use equal priors.

```
model.priors = c(1/3, 1/3, 1/3)
model.names = c("M1","M.5","M0")
```

Now we can apply `line.models()` that gives us the membership probabilities of each observation in each model.

```
scales = c(0.2, 0.2, 0.2)
slopes = c(1, 0.5, 0)
cors = c(0.995, 0.995, 0.995)
res = line.models(X = dat[,c("beta1","beta2")], SE = dat[,c("se1","se2")],
                  scales = scales, slopes = slopes, cors = cors,
                  model.names = model.names,
                  model.priors = model.priors,
                  r.lkhood = 0)
head(res, 2)
```

```
##                M1          M.5 M0
## [1,] 6.718430e-30 2.908731e-14  1
## [2,] 4.700843e-27 2.966454e-11  1
```

The result has one row per each observation and one column per each linemodel. The example output above tells that for the first two observations the posterior probability is close to 1 that they follow the model M_0 when the other two candidates were M_1 or $M_{0.5}$.

line.models.with.proportions( ) If we do not want to fix the prior probabilities of the linemodels before the analysis, we can assign a prior distribution on those probabilities and estimate them. The prior distribution is Dirichlet$(a_1, \ldots, a_K)$, where the user can give the values a_k in a K-dimensional vector called `diri.prior`. By default, we have $a_k = \frac{1}{K}$ for all $k \leq K$. Since this approach uses an iterative Gibbs sampler, we can also specify the number of iterations (default 200) and the length of the burn-in period (default 20). Larger numbers of iterations may improve accuracy.

```
res.2 = line.models.with.proportions(
  X = dat[,c("beta1","beta2")],
  SE = dat[,c("se1","se2")],
  scales = scales, slopes = slopes, cors = cors,
  model.names = model.names,
  r.lkhood = 0,
  n.iter = 1000, n.burnin = 20)
```

The result is a list with two members:

- `params` contains the posterior distribution of the proportion of observations belonging to each of the linemodels (mean, 95% highest posterior region and standard deviation)
- `groups` has the posterior probabilities of each observation belonging to each of the linemodels (similar to output of `line.models( )` above)

```
res.2$params
```

```
##          mean    95%low     95%up         sd
## M1  0.4413605 0.3314115 0.5554540 0.05749024
## M.5 0.3652402 0.2505183 0.4846871 0.05798846
## M0  0.1933993 0.1142522 0.2857652 0.04431844
```

The proportion estimates above and their 95% intervals correspond to Fig.1D.

```
head(res.2$groups, 2)
```

```
##      M1 M.5 M0
## [1,]  0   0  1
## [2,]  0   0  1
```

In these data, the membership probabilities are highly similar between `line.models.with.proportions( )` and `line.models( )` since the estimated proportions of the three groups do not differ much from the uniform prior that we used in `line.models( )`. Note that in `line.models.with.proportions( )`, the unit of accuracy of the probabilities is determined by the number of iterations and, for example, with 1000 iterations the unit is $1/1000 = 0.1\%$. This explains why we see here probabilities that are exactly 1 whereas earlier with `line.models( )` we saw some tiny deviations from 1.

visualize.classification.regions( ) For any two 2-dimensional linemodels, we can visualize the regions on the plane according to the posterior probabilities of the putative observations if they were observed in the region. The probabilities will depend strongly on the standard errors (parameter `SE`) and the prior probabilities of the linemodels (parameter `model.priors`).

Let's visualize the comparison of the first two models with both effects estimated with an SE = 0.05 and both models being equally probable *a priori*.

```
visualize.classification.regions(
  scales[1:2], slopes[1:2], cors[1:2],
  SE = c(0.05, 0.05), r.lkhood = 0,
  model.priors = c(0.5, 0.5),
  add.legend = TRUE,
  breakpoints = 200,
  xlab = expression(beta[1]),
  ylab = expression(beta[2]),
  emphasize.axes = TRUE)
```

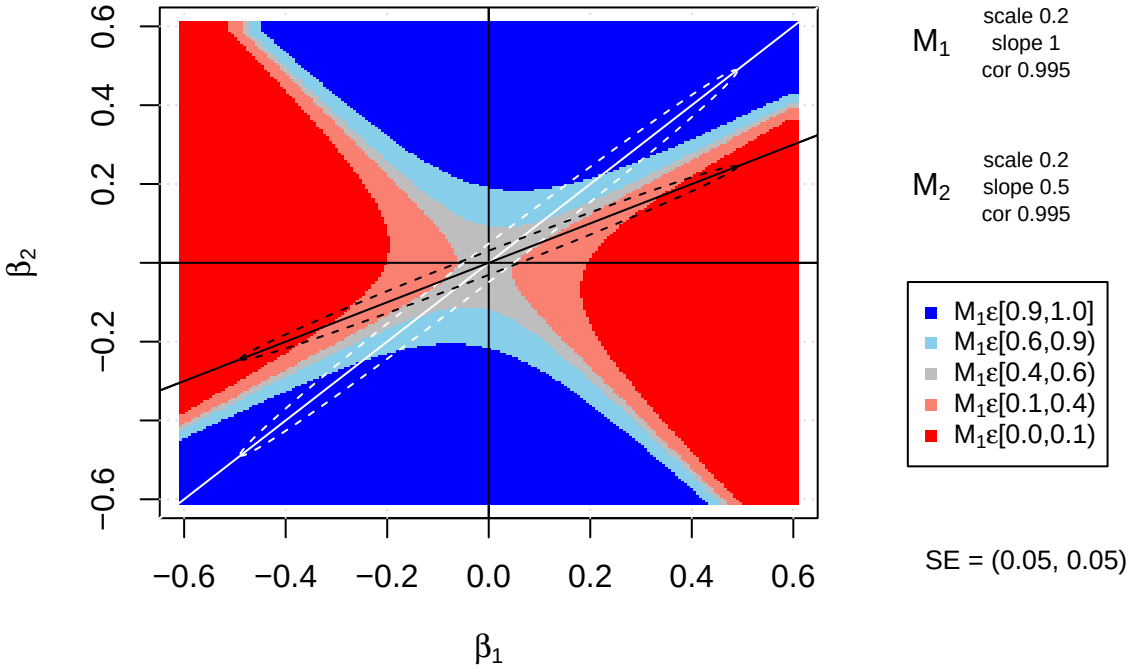


Figure 3. Each point on the plane within 3 scale units from the origin is colored according to the posterior probability of the first model (here model with slope = 1) where blue colors mark higher probabilities and red colors mark lower probabilities (see legend). This plot depends on the values of standard errors (`SE`), prior probabilities of the models (`model.priors`) and the correlation of the estimators (`r.lkhood`).

5. Optimizing parameters of line models

Sometimes we may not have a good reason to fix the parameters of a linemodel to any particular values of scale, slope or correlation, but rather we may want to estimate from the data a set of suitable values for the parameters.

line.models.optimize( ) The optimization function takes in initial values of all parameters and a list named `par.include` containing K-vectors `scales` and `cors` and a $K \times (M-1)$ matrix `slopes`. They all have `TRUE`/`FALSE` entries that describe which parameters will be included in the optimization process.

For example, let's optimize the slope of the second model that represents variants where $\beta_1 > \beta_2 > 0$. We may not want to assume that $\beta_2 = 0.5 \cdot \beta_1$ in case that some other slope would clearly better describe the relationship in the data. Let's optimize over b_2 and use an initial value of $b_2 = 0.3$.

```
#par.include here says that only the slope of 2nd model will be optimized
par.include = list(scales = c(FALSE, FALSE, FALSE),
```

```
                  slopes = cbind(c(FALSE, TRUE, FALSE)),
                  cors = c(FALSE, FALSE, FALSE))
line.models.optimize(
  X = dat[,c("beta1","beta2")],
  SE = dat[,c("se1","se2")],
  par.include = par.include,
  init.scales = scales,
  init.slopes = c(0, 0.3, 1), #initial value for 2nd slope is 0.3
  init.cors = cors,
  model.priors = model.priors,
  model.names = model.names,
  r.lkhood = 0)
```

```
##
##
## Initial values
## scales: 0.2, 0.2, 0.2
## slopes:
##  0
##  0.3
##  1
## cors: 0.995, 0.995, 0.995
## proportions: 0.333, 0.333, 0.333
## Initial log-lkhood: 67.16402
## Optimizing w.r.t: slope2_1, over the range: (-Inf,Inf)
## Convergence criteria:
##  Relative diff in parameters <= 0  or
##  Difference in log-likelihood < 0.001
##
## iter: 6 ; log-lkhood: 171.47986
## Relative diffs in optimized parameters: 9.75e-05
## proportions: 0.189, 0.372, 0.439
## scales: 0.2, 0.2, 0.2
## slopes:
##  0
##  0.5098
##  1
## cors: 0.995, 0.995, 0.995
## Log-likelihood value converged.
```

```
## $scales
## [1] 0.2 0.2 0.2
##
## $slopes
##           [,1]
## [1,] 0.0000000
## [2,] 0.5098276
## [3,] 1.0000000
##
## $cors
## [1] 0.995 0.995 0.995
##
## $weights
```

```
##        M1       M.5        M0
## 0.1893305 0.3718141 0.4388554
```

As the optimization proceeds, the function prints on the console some status of the process (see `print.steps` parameter to control output). The final result shows that the EM-algorithm converged to value $b_2 = 0.5098$. This is very close to the value 0.5 that we had been using, and hence updating this slope value in the line model analysis would not lead to a noticeable difference from the results of our earlier analyses.

Additional arguments of line.models.optimize()

- `assume.constant.SE`, logical, if TRUE then, separately for each effect variable, SE of every data point is set to the median of the observed SEs for that effect variable across all the data points. This considerably speeds up the optimization. In genetics applications, SEs may be nearly constant for one phenotype, when the effect size estimates and SEs of variant v is multiplied by $\sqrt{f_v(1-f_v)}$ where f_v is the minor allele frequency of variant v.
- `force.same.scales`, logical, if TRUE then all those scale parameters that are being optimized will be forced to be equal. This may stabilize the optimization in case of convergence problems. Defaults to FALSE.
- `op.method`, determines the method used in the optimization by the R's `optim()` function. Default is "BFGS"; another option is "Nelder-Mead". If one method crashes, try the other. Does not affect the univariate optimization, which is always done with "Brent".
- `tol.loglk`, defines the largest increase in log-likelihood between consecutive iterations that is considered small enough that the algorithm has converged. Defaults to 0.001.
- `tol.par`, defines the largest absolute value of the relative change in parameter values that is considered small enough that the algorithm has converged. Defaults to 0, that is, convergence is determined purely by the log-likelihood value.
- `print.steps` takes value of 0 (no output to console), 1 (only initial and final results shown, default) or 2 (prints state after every EM-step).

When to optimize? We shouldn't just blindly optimize the model parameters for model comparison since we may overfit to the data and the optimized model may not anymore describe exactly that property which we wanted it to describe in the first place. In cases where we can come up with a particular model that exactly describes our hypothesis of interest, such as no effect on disease subtype 2 ($\beta_2 = 0$) or same effect in both subtypes ($\beta_1 = \beta_2$), we should simply use such a model rather than use the optimization to improve the fit. On the other hand, optimization is useful when we can see from the data that there is a particular relationship that we want to model, but we do not know exactly what would be an appropriate parameter value that captured well that relationship.

Suitable scales. The suitable values for the scale parameters are crucial in order to get sensible results from the linemodels. If, for example, scales are too small for the observed data, then none of the line models can describe the data well and the resulting assignment probabilities may not be sensible. To let the data decide suitable scales, we can optimize the scale parameters first and then estimate the proportion parameters using an empirical Bayes method with scales set to their maximum likelihood estimates. If it is reasonable to assume that all linemodels have the same scale, then we can set `force.same.scales = TRUE` to force the scale parameter to be shared by all the linemodels. This may add robustness to the parameter estimation escpecially when there are not much data. If the scales of the linemodels are expected to be different, then forcing them to be the same may lead to misleading results.

Outlier distribution. If there are outlier observations that do not fit well to any of the linemodels suitable for the bulk of the data, then we can fit an outlier distribution with a large scale value and a correlation of 0 (and in this case the value of slope does not matter). The idea is that the outliers will be assigned to this vague outlier distribution, which allows the linemodels to be appropriately tuned by the points that indeed fit well with those linemodels.

See Appendix 1 for more examples of the optimization.

6. Likelihood ratios for model comparison

A way to compare two mixture models built from sets of linemodels is via likelihood ratio. The optimization procedure reports both the initial and the optimized log-likelihoods of the whole mixture model that is being considered in the optimization. The difference in log-likelihoods between different mixtures of linemodels can be used as a statistic for model comparison. For example, we can see whether forcing same scales causes a big decrease in log-likelihood or whether a mixture of two linemodels improves a lot over a model with only a single linemodel. Unfortunately, there is no theoretically known distribution for the logarithm of the likelihood ratio under different null hypotheses so one needs to estimate the null distribution by simulation. Such a procedure is implemented in function `simulate.loglr()`.

This function takes in the null model (NULL) and the alternative model (ALT) specified similarly as in the optimization function. That is, for the null model, we need to specify `par.include.null`, `force.same.scales.null`, `init.scales.null`, `init.slopes.null`, `init.cors.null`. Additionally, `model.priors.null` is optional as it is used as the initial value; a good guess can speed up the computation. To specify the alternative model, the same parameters are used except `.null` is replaced by `.alt`. Additional control parameters used in optimization can be specified as well.

This function first optimizes both ALT and NULL on the observed data and records the logarithm of likelihood ratio (logLR) between ALT and NULL.

Then the function generates such data according to the optimized null model that mimic the observed data in terms of sample size and scales. For each data set, ALT and NULL are optimized and logLR is recorded. At the end, the observed logLR can be compared to the values of logLR generated under the NULL, for example, to estimate an empirical P-value to assess
the need for the ALT model compared to the NULL model.

Let's assess whether an addition of a third linemodel helps in the example data. We fully specify the null model to have two lines: (scale = 0.2, slope = 0, cor = 0.995) and (0.2, 1, 0.995). Then we let the alternative model to have an additional line with unknown slope but fixed scale (0.2) and correlation (0.995). For illustration, here we estimate the null distribution of logLR using only 10 simulated data sets. (See Appendix 1, for a larger analysis.)

```
set.seed(1655)
sim.loglr = simulate.loglr(
  X = dat[,c("beta1","beta2")],
  SE = dat[,c("se1","se2")],
  n.sims = 10,
  par.include.null = list(scales = c(FALSE,FALSE),
                          slopes = cbind(FALSE,FALSE),
                          cors = c(FALSE,FALSE)),
  init.scales.null = c(0.2, 0.2),
  init.slopes.null = c(0, 1),
  init.cors.null = c(0.995, 0.995),
  par.include.alt = list(scales = c(FALSE,FALSE,FALSE),
                          slopes = cbind(FALSE,FALSE,TRUE),
                          cors = c(FALSE,FALSE,FALSE)),
  init.scales.alt = c(0.2, 0.2, 0.2),
  init.slopes.alt = c(0, 1, 0.5),
  init.cors.alt = c(0.995, 0.995, 0.995),
  r.lkhood = 0, tol.loglk = 1e-2,
  assume.constant.SE = FALSE)
```

The result shows the logLR for the observed data and the distribution of the logLR values for the null simulations.

```
sim.loglr
```

```
## $obs.loglr
## [1] 232.8534
##
## $null.loglr
##  [1] 0.3551428 0.3756598 1.6128194 0.2150691 0.4810218 0.3452886 0.6344106
##  [8] 0.3616687 1.9053013 0.4893359
```

Here the observed logLR is much larger than the values computed from simulated null data, which suggests that the third line model is very useful in explaining the data.

NOTE: More simulations than done here should be made in order to have a reliable idea about statistical significance of the alternative model compared to the null. (See Appendix 1.)

NOTE: By setting `assume.constant.SE = TRUE` we could considerably speed up this function. Thus, if SE can be assumed constant per each effect variable across the observed data points, then one should make use of this option. (See Appendix 1.)

7. A mixture of Gaussians as the prior distribution of effects

By default, the prior distribution of the largest effect variable is $\mathcal{N}(0, s^2)$ where s is the scale parameter. (For 2-dimensional case, the largest effect is β_1 when $|b| \leq 1$ and β_2 when $|b| > 1$ where b is the slope parameter. For multi-dimensional case, analogous considerations hold.) For the `line.models( )` function, it is also possible to specify the effect size prior as a mixture of Gaussians. This can be done by defining `scale.weights` matrix, that has one row per each linemodel and the number of columns is the number of mixture components (L). Each row is normalized to sum to 1. Denote by $w_{k\ell}$ the element (k, ℓ) of `scale.weights` matrix. Then the prior for the largest effect in linemodel k is

$$\sum_{\ell=1}^{L} w_{k\ell} \mathcal{N}\left(0, \left(\frac{s_k}{L-\ell+1}\right)^2\right),$$

where s_k is the scale parameter of the model k. Note that s_k is given in standard deviation units and hence s_k^2 is the corresponding variance. (In case the weights of each linemodel are the same, we can also give the weights as a single vector instead of a matrix that had K identical rows.)

For example, if we want to model the effect size of linemodels

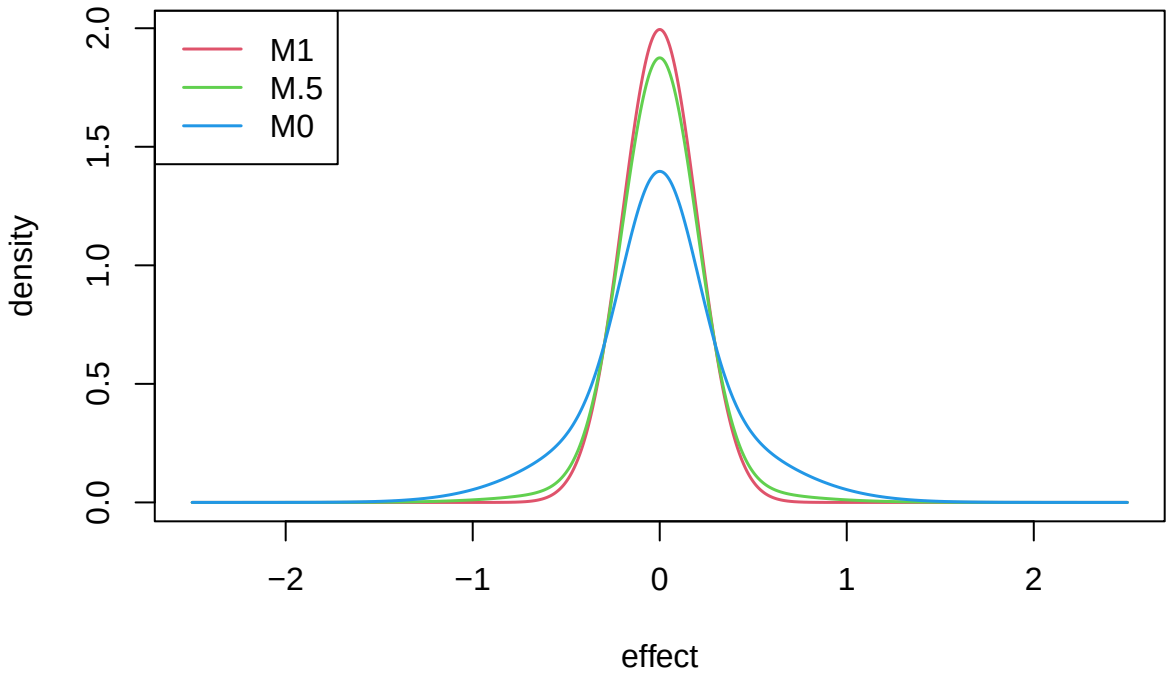
- M_1 as $\mathcal{N}(0, 0.2^2)$,
- $M_{0.5}$ as a 10%:90% mixture of $\mathcal{N}(0, 0.5^2)$ and $\mathcal{N}(0, 0.2^2)$,
- M_0 as a 50%:50% mixture of $\mathcal{N}(0, 0.5^2)$ and $\mathcal{N}(0, 0.2^2)$,

then we can set all scales to $s_k = 1.0$ and specify the `scale.weights` as

```
scales = c(1, 1, 1)
# elements of scale.weights will now corresp. to scales of
# 0.2 = 1.0/5, 0.25 = 1.0/4, 0.33 = 1.0/3, 0.5 = 1.0/2 and 1.0 = 1.0/1:
scale.weights = rbind(
  c(1.0, 0, 0, 0.0, 0.0),  #N(0, 0.2^2)
  c(0.9, 0, 0, 0.1, 0.0),  #0.9*N(0, 0.2^2) + 0.1*N(0, 0.5^2)
  c(0.5, 0, 0, 0.5, 0.0))  #0.5*N(0, 0.2^2) + 0.5*N(0, 0.5^2)
```

visualize.scales( ) We can visualize the mixture prior distribution of effect sizes:

```
visualize.scales(scales, scale.weights, model.names)
```




```
## Plotted the following effect distributions:
## M1: Scale = 1 weights = 1,0,0,0,0.
## M.5: Scale = 1 weights = 0.9,0,0,0.1,0.
## M0: Scale = 1 weights = 0.5,0,0,0.5,0.
```

Applying these priors to the effect sizes in `line.models( )` is now straighforward: we just add `scale.weights` as an argument in the function call.

```
res.sc = line.models(
  X = dat[,c("beta1","beta2")],
  SE = dat[,c("se1","se2")],
  scales = scales, slopes = slopes, cors = cors,
  model.names = model.names,
  model.priors = model.priors,
  r.lkhood = 0,
  scale.weights = scale.weights)
head(res.sc, 2)
```

```
##                M1          M.5        M0
## [1,] 6.618542e-30 5.896538e-07 0.9999994
## [2,] 3.222500e-27 7.049895e-05 0.9999295
```

8. How to set the value of `r.lkhood`?

When the effect estimators on the two outcomes are correlated, we must account that in our observation model through the parameter `r.lkhood`. The input value of `r.lkhood` can either be given as a $M \times M$

correlation matrix, or as a vector of the upper-triangle of the correlation matrix in row-major order. Thus, for 2-dimensional data, input `r.lkhood = 0.9` specifies the same model as `r.lkhood = matrix(c(1,0.9, 0.9, 1), ncol = 2)`.

1. Suppose Y_1 (outcome 1) and Y_2 (outcome 2) are continuous outcomes and we have measured the effect of variable X on both of them separately on the same samples using linear regression models $Y_1 = \mu_1 + X\beta_1 + \varepsilon_1$ and $Y_2 = \mu_2 + X\beta_2 + \varepsilon_2$. Then,

$$\text{r.lkhood} = \text{cor}\left(\widehat{\beta}_1, \widehat{\beta}_2\right) = \text{cor}(\varepsilon_1, \varepsilon_2) \approx \text{cor}(Y_1, Y_2),$$

   where the last approximation assumes that X explains only a little of the variance of the outcome variables. Thus, to a first approximation, we may set the pairwise correlation values in `r.lkhood` equal to the pairwise correlations of the outcome variables. If the samples are only partially overlapping, then we will need to shrink the correlation of the residuals towards zero by a factor of $n_{12}/\sqrt{n_1 n_2}$, where n_{12} is the number of samples that are shared between the analyses and n_1 and n_2 are the total sample sizes of each analysis.

2. If our data come from two case-control analyses A and B, where samples are partially overlapping, we can derive a value for `r.lkhood` using the formula by Bhattacharjee et al. (2012). Denote by $n_A^{(1)}$ the number of cases in A and by $n_A^{(0)}$ the number of controls in A, and similarly for study B. Mark by $n_{AB}^{(11)}$ and $n_{AB}^{(00)}$, respectively, the number of shared cases and shared controls between the studies and by $n_{AB}^{(10)}$ and $n_{AB}^{(01)}$, respectively, the number of cases of A that are controls of B and the number of controls of A that are cases of B. Then

$$\text{r.lkhood} = \sqrt{\frac{n_A^{(1)} n_A^{(0)}}{n_A^{(1)} + n_A^{(0)}}} \sqrt{\frac{n_B^{(1)} n_B^{(0)}}{n_B^{(1)} + n_B^{(0)}}} \left(\frac{n_{AB}^{(11)}}{n_A^{(1)} n_B^{(1)}} - \frac{n_{AB}^{(10)}}{n_A^{(1)} n_B^{(0)}} - \frac{n_{AB}^{(01)}}{n_A^{(0)} n_B^{(1)}} + \frac{n_{AB}^{(00)}}{n_A^{(0)} n_B^{(0)}} \right).$$

   This expression can be evaluated using `beta.cor.case.control( )` function of the `linemodels` package.

9. Features informing the prior probabilities of the linemodels.

If each data point has one or more features available those can be used to inform the prior probabilities of the linemodels in a data point specific way. Features can be any numerical vectors. In genetics context, examples of features could be a binary indicator whether the variant is close to an immune-related gene or a continuous measure of variant's deleteriousness (e.g. CADD score).

The features are give in a matrix (one row per data point, one column per feature) via the parameter `features` of the function `line.models.with.features()`. As an example, consider data set `linemodels.ex2` where there are 101 observations, three sets of effect sizes and standard errors and one feature called `annotation`.

```
dat2 = linemodels.ex2
head(dat2, 2)
```

```
##        beta1       beta2       beta3   se1   se2   se3 annotation
## 1 0.04293033 0.004829285 -0.09449804 0.005 0.005 0.005          0
## 2 0.04457683 0.008704558 -0.06985898 0.005 0.005 0.005          0
```

To estimate the memberships of the variants on the following two linemodels

```
slopes = matrix(c(0, -2,
                  3,  6), byrow = TRUE, ncol = 2)
scales = rep(0.1, nrow(slopes))
cors = rep(0.999, nrow(slopes))
```

using the annotations, we call

```
res.f = line.models.with.features(X = dat2[,1:3], SE = dat2[,4:6],
                                  scales = scales, slopes = slopes, cors = cors,
                                  n.iter = 500, n.burnin = 20,
                                  features = dat2$annotation,
                                  verbose = FALSE)
```

Posterior probability of each point in each model is in `res.f$groups` Thus, for the point 101, we have

```
res.f$groups[101,]
```

```
##   Mo1   Mo2
## 0.184 0.816
```

Thus, it is more probably assigned to the second model. You can check (in `linemodels_examples.R`) that without annotations (i.e. using `line.models.with.proportions()`), this observations was about equally probable to come from either of the two models.

The regression coefficients of the annotation model can be found in

```
res.f$params
```

```
##                 mean    95%low     95%up        sd
## Mo1_intcp.  0.000000  0.000000  0.000000 0.0000000
## Mo1_X1      0.000000  0.000000  0.000000 0.0000000
## Mo2_intcp. -1.656067 -2.351010 -1.056387 0.3322939
## Mo2_X1      3.288569  2.340255  4.246895 0.5024613
```

and are explained in the section about "Mathematical and technical details".

10. Mathematical and technical details

Let $\hat{\beta}_{ij}$ be the effect estimate of observation $i = 1, \ldots, n$ on the outcome variable $j = 1, \ldots, M$, and $\hat{\sigma}_{ij}$ its estimated standard error. Define, for $k = 1, \ldots K$, a line model $\mathcal{M}_k$ via three parameters $\mathcal{M}_k = (s_k, \boldsymbol{b}_k, r_k)$ called scale s_k (scalar), slope vector $\boldsymbol{b}_k = (b_{k2}, \ldots, b_{kM})$ (dimension $M - 1$) and correlation r_k (scalar), respectively. Intuitively, $\mathcal{M}_k$ models the true effects as centered around line ℓ on which $\beta_{ij} = b_{kj} \cdot \beta_{i1}$ for $j = 2, \ldots, M$. Let m be the effect variable that has the largest magnitude on that line among all M dimensions. The prior standard deviation ("scale") of the effect for dimension m is s_k, and the prior scales for the remaining $M - 1$ dimensions are determined by their slopes with respect to the variable m. The deviation of the effects from the exact line is determined by the correlation coefficient r_k. We first define the linemodel for a diagonal case (all slopes = 1), and then use an orthogonal transformation to rotate the linemodel to match the target line ℓ.

For the given $\mathcal{M}_k = (s_k, \boldsymbol{b}_k, r_k)$, define the corresponding diagonal distribution of effect sizes as an M-dimensional Gaussian $\mathcal{N}_M(\boldsymbol{0}, \boldsymbol{D}_k)$, where the covariance matrix is

$$\boldsymbol{D}_k = \begin{pmatrix} 1 & r_k & \dots & r_k \\ r_k & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & r_k \\ r_k & \dots & r_k & 1 \end{pmatrix}.$$

Let $\boldsymbol{Q}_k$ be the rotation matrix that transforms the diagonal line to the line ℓ (with slope $\boldsymbol{b}_k$).

$$\boldsymbol{Q}_k = \boldsymbol{I}_M - \boldsymbol{d}\boldsymbol{d}^T - \boldsymbol{v}\boldsymbol{v}^T + (\boldsymbol{d}, \boldsymbol{v})\,\boldsymbol{T}\,(\boldsymbol{d}, \boldsymbol{v})^T,$$

where $\boldsymbol{d}$ is the unit vector of the diagonal, $\boldsymbol{v}$ is the unit vector orthogonal to $\boldsymbol{d}$ and on the plane spanned by $\boldsymbol{d}$ and line ℓ, and the 2-dimensional orthogonal transformation $\boldsymbol{T}$ rotates the diagonal to the target line in the plane they define:

$$\boldsymbol{T} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix},$$

where α is the angle between the two lines.

The prior distribution of the effect sizes according to model $\mathcal{M}_k$ is then defined as $\mathcal{N}_M(\boldsymbol{0}, \boldsymbol{\Theta}_k)$, where covariance matrix $\boldsymbol{\Theta}_k = \frac{s_k^2}{C_k}\boldsymbol{Q}_k\boldsymbol{D}_k\boldsymbol{Q}_k^T$ and normalization by $C_k = \max\{\boldsymbol{Q}_k\boldsymbol{D}_k\boldsymbol{Q}_k^T\}$ confirms that the largest of the standard deviations of the effects is s_k. It is also possible to specify the prior distribution of the effect size as a mixture of Gaussians, e.g., to model heavier tails (see section 7 above).

In terms of the parameters defined above, we can use line models to define, for example,

- null model ($s_k = 0$),
- independent effects model ($s_k > 0, r_k = 0$),
- fixed effects model ($s_k > 0, b_k = 1, r_k = 1$),
- outcome 1 specific effect ($s_k > 0, b_k = 0, r_k = 1$),
- and any generalizations of these where deviation from exact line is allowed ($0 \le r_k < 1$).

The observation model for the effect size estimates is a Gaussian distribution around the true effect sizes with a covariance matrix

$$\hat{\boldsymbol{\Sigma}}_i = \mathrm{diag}(\mathrm{SE}_i)\,\boldsymbol{R}_{\mathrm{lkhood}}\,\mathrm{diag}(\mathrm{SE}_i),$$

where $\boldsymbol{R}_{\mathrm{lkhood}}$ describes how the effect size estimators are correlated, for example, because of the sample overlap in the data sets from where the two effects have been estimated (see section 8 above), and SE_i is the M-vector of standard errors for observation i.

It follows that by combining the Gaussian prior with the Gaussian observation model, the marginal distribution for the observed effect size estimates for observation i, $\widehat{\boldsymbol{\beta}}_i = \left(\widehat{\beta}_{i1}, \dots, \widehat{\beta}_{iM}\right)^T$, under model $\mathcal{M}_k$ is

$$\widehat{\boldsymbol{\beta}}_i \,|\, \mathcal{M}_k \sim \mathcal{N}_M\left(\boldsymbol{0}, \boldsymbol{\Theta}_k + \hat{\boldsymbol{\Sigma}}_i\right).$$

10.1. Probabilistic assignment of observations into models Given K line models $(\mathcal{M}_k)_{k=1}^K$ and their prior probabilities $\boldsymbol{\pi} = (\pi_k)_{k=1}^K$, we can estimate the posterior probability that observation i follows the model k as

$$\Pr(i \sim \mathcal{M}_k \,|\, \mathrm{Data}_i) = \frac{\pi_k\,\mathcal{N}_M\left(\widehat{\boldsymbol{\beta}}_i \,|\, \boldsymbol{0}, \boldsymbol{\Theta}_k + \hat{\boldsymbol{\Sigma}}_i\right)}{\sum_{\ell=1}^K \pi_\ell\,\mathcal{N}_M\left(\widehat{\boldsymbol{\beta}}_i \,|\, \boldsymbol{0}, \boldsymbol{\Theta}_\ell + \hat{\boldsymbol{\Sigma}}_i\right)}.$$

This calculation can be done separately for each variable and is implemented in the `line.models( )` function.

If we do not want to pre-specify the exact prior probabilities of each model, we can set a prior distribution on $\boldsymbol{\pi}$ and estimate its posterior distribution together with assignment probabilities of observations into models. For this task, we use a prior distribution $\boldsymbol{\pi} \sim \text{Dirichlet}(\delta_1, \ldots, \delta_K)$, where the default values of the hyper-parameter are set to $\delta_k = \frac{1}{K}$ for each $k = 1, \ldots, K$. A Gibbs sampler to estimate the posterior distribution of this model is implemented in the `line.models.with.proportions( )` function. Note that this model estimates the probabilities jointly across all observations rather than separately for each observations as was done by `line.models( )`.

10.2. Likelihood function of the mixture model We can combine the K linemodels into a single mixture model that can be thought as generating the data in two steps. First, we choose one of the K possible linemodels according to their probabilities $\boldsymbol{\pi}$ and conditional on the chosen linemodel we sample an observation from it. This mixture model gives the observed data a probability density function

$$\Pr\left(\widehat{\boldsymbol{\beta}} \,|\, \boldsymbol{\Gamma}, \boldsymbol{\pi}\right) = \prod_{i=1}^{n} \sum_{k=1}^{K} \pi_k \, \mathcal{N}_M\left(\widehat{\boldsymbol{\beta}}_i \,; 0, \widehat{\boldsymbol{\Sigma}}_i + \boldsymbol{\Theta}_k\right),$$

where $\boldsymbol{\Gamma} = \{s_k, \boldsymbol{b}_k, r_k \,|\, k \leq K\}$ contains all $(M+1)K$ line parameters of the K models. Logarithm of this, considered as a function of the parameters, is the mixture model log-likelihood function of the parameters:

$$\log L\left(\boldsymbol{\Gamma}, \boldsymbol{\pi} \,;\, \text{Data}\right) = \sum_{i=1}^{n} \log\left(\sum_{k=1}^{K} \pi_k \, \mathcal{N}_M\left(\widehat{\boldsymbol{\beta}}_i \,; 0, \widehat{\boldsymbol{\Sigma}}_i + \boldsymbol{\Theta}_k\right)\right).$$

10.3. EM-algorithm to optimize the linemodel parameters Using the `line.models.optimize()` function, one can optimize the likelihood function with respect to any set of the model parameters (see section 5 above for an example).

With K linemodels, our model parameters are scales s_k, slopes $\boldsymbol{b}_k$, correlations r_k and proportions π_k for each model $k \leq K$, but because of the constraint $\sum_k \pi_k = 1$, we can write $\pi_K = 1 - \pi_1 - \ldots - \pi_{k-1}$ and we have only $(M+2)K - 1$ free parameters. As before, denote the $(M+1)K$ line parameters by $\boldsymbol{\Gamma} = \{s_k, \boldsymbol{b}_k, r_k \,|\, k \leq K\}$ and the K proportion parameters by $\boldsymbol{\pi} = \{\pi_k \,|\, k \leq K\}$, and their values at iteration t of the algorithm as $\boldsymbol{\Gamma}^{(t)}$ and $\boldsymbol{\pi}^{(t)}$, respectively, with the initial values corresponding to $t = 0$.

As is typical for EM-algorithms for mixture models, we introduce latent variable $z_i \in \{1, ..., K\}$ for each observation $i = 1, \ldots, n$ that tells to which linemodel the observation belongs. The augmented likelihood of the data consisting of the effect estimates $\left(\widehat{\beta}_i\right)_{i=1}^{n}$ and the corresponding standard errors $(\widehat{\sigma}_i)_{i=1}^{n}$ is

$$L(\boldsymbol{\Gamma}, \boldsymbol{\pi}, \boldsymbol{z}; \text{Data}) = p(\boldsymbol{\pi}) \prod_{i=1}^{n} p(z_i \,|\, \boldsymbol{\pi}) \, p(\text{Data}_i \,|\, z_i) = c \cdot \prod_{i=1}^{n} \pi_{z_i} \, \mathcal{N}_M\left(\widehat{\boldsymbol{\beta}}_i \,|\, \boldsymbol{0}, \boldsymbol{\Theta}_{z_i} + \hat{\boldsymbol{\Sigma}}_i\right),$$

where c is the normalizing constant of the K-dimensional uniform distribution of the proportion parameter $\boldsymbol{\pi}$. The log-likelihood is

$$\mathcal{L}(\boldsymbol{\Gamma}, \boldsymbol{\pi}, \boldsymbol{z}) = \log L(\boldsymbol{\Gamma}, \boldsymbol{\pi}, \boldsymbol{z}; \text{Data}) = \log(c) + \sum_{i=1}^{n} \left[\log(\pi_{z_i}) + \log \mathcal{N}_M\left(\widehat{\boldsymbol{\beta}}_i \,|\, \boldsymbol{0}, \boldsymbol{\Theta}_{z_i} + \hat{\boldsymbol{\Sigma}}_i\right)\right].$$

The EM-algorithm at iteration $t > 0$ consists of two steps.

1. Compute the posterior distribution $\left(p_{ik}^{(t)}\right)_{i,k}$ of latent variables $(z_i)_{i=1}^{n}$ conditional on the current values of parameters, $\boldsymbol{\Gamma}^{(t-1)}$ and $\boldsymbol{\pi}^{(t-1)}$. This computation can be done separately for each variable i using the assignment probability formula

$$p_{ik}^{(t)} = \Pr(i \sim \mathcal{M}_k \,|\, \boldsymbol{\Gamma}^{(t-1)}, \boldsymbol{\pi}^{(t-1)}, \text{Data}_i) = \frac{\pi_k^{(t-1)} \, \mathcal{N}_M\left(\widehat{\boldsymbol{\beta}}_i \,|\, \boldsymbol{0}, \boldsymbol{\Theta}_k^{(t-1)} + \hat{\boldsymbol{\Sigma}}_i\right)}{\sum_{\ell=1}^{K} \pi_\ell^{(t-1)} \, \mathcal{N}_M\left(\widehat{\boldsymbol{\beta}}_i \,|\, \boldsymbol{0}, \boldsymbol{\Theta}_\ell^{(t-1)} + \hat{\boldsymbol{\Sigma}}_i\right)}.$$

2. Define new values $\boldsymbol{\pi}^{(t)}$ and $\boldsymbol{\Gamma}^{(t)}$ as maximizers of the expected log-likelihood function where the expectation is over the distribution of latent variables $\boldsymbol{z}$. Thus, the target function to be optimized is

$$\mathrm{E}_{\boldsymbol{z}^{(t)}}\left(\mathcal{L}(\boldsymbol{\Gamma}, \boldsymbol{\pi}, \boldsymbol{z})\right) = \log(c) + \sum_{i=1}^{n}\sum_{k=1}^{K}\left[p_{ik}^{(t)}\ \log(\pi_k) + p_{ik}^{(t)}\ \log \mathcal{N}_M\left(\widehat{\boldsymbol{\beta}}_i \,|\, \mathbf{0}, \boldsymbol{\Theta}_k + \hat{\boldsymbol{\Sigma}}_i\right)\right].$$

   We can analytically maximize with respect to $\boldsymbol{\pi}$ by setting $\pi_k^{(t)} = \frac{1}{n}\sum_i p_{ik}^{(t)}$. For the remaining parameters in $\boldsymbol{\Gamma}$, we use numerical optimization of the above function.

The EM-algorithm is run until the difference in the maxmized log-likelihood between consecutive iterations drops below a given tolerance (`tol.loglk`), such as the default value of 0.001. The convergence can also be defined by a threshold (`tol.par`) on the relative change in parameter values between consecutive iterations. Note that if `tol.par` is negative, then the log-likelihood value alone defines the convergence.

10.4. Feature model Let $\boldsymbol{X}$ be an $n \times p$ matrix of feature values per each observation where the number of features $p > 0$. The prior probability of observation i belonging to model $\mathcal{M}_k$ is determined by logistic regression (if the number of models $K = 2$) or multinomial regression (if $K > 2$). In both cases, the prior probability is (proportional to)

$$\pi_{ik} = \Pr(i \sim \mathcal{M}_k) \propto \exp\left(\sum_{j=0}^{p} x_{ij}\eta_{jk}\right),$$

where x_{ij} is the feature value of observation i on feature j and η_{jk} is the regression coefficient of feature j for model k. The feature 0 is the intercept term ($x_{i0} = 1$ for all i) and is added to the model by default; It should not be added by the user to the feature matrix. To make the coefficients identifiable, one of the models is chosen as the baseline and its coefficients are set to 0. Thus, the coefficients can be interpreted as the log-odds values per unit change in the feature with respect to the baseline model. The feature model is implemented in the function `line.models.with.features()` that outputs also the posterior distributions of the regression coefficients.

References

Bhattacharjee S., Rajaraman P., Jacobs K.B., et al. 2012. A Subset-Based Approach Improves Power and Interpretation for the Combined Analysis of Genetic Association Studies of Heterogeneous Traits, *The American Journal of Human Genetics*, 90(5): 821-835.

Appendix 1 : Analysis of `linemodels.ex1`

The `linemodels` package includes a data set called `linemodels.ex1` which can also be found from the GitHub page under /data.

```
str(linemodels.ex1)
```

```
## 'data.frame':    100 obs. of  7 variables:
##  $ beta1: num  0.3629 0.3285 0.1275 0.2333 0.0541 ...
##  $ beta2: num  0.00356 0.02783 0.01202 -0.01882 -0.00561 ...
##  $ beta3: num  0.00718 -0.00869 -0.00316 0.01252 -0.01426 ...
##  $ se1  : num  0.0166 0.0127 0.015 0.015 0.0163 ...
##  $ se2  : num  0.0166 0.0127 0.015 0.015 0.0163 ...
##  $ se3  : num  0.0166 0.0127 0.015 0.015 0.0163 ...
##  $ maf  : num  0.177 0.453 0.231 0.233 0.185 ...
```

The data have three effect estimates (`beta1`, `beta2`, `beta3`), their standard errors (`se1`, `se2`, `se3`) and minor allele frequency (`maf`) for 100 genetic variants ("observations"). Here the 3 betas could be, for example, regression coefficients of variants for three related diseases from three genome-wide association studies.

Let's first see whether we could treat SEs across variants for each outcome variable as constant to speed up the optimization algorithm.

```
summary(linemodels.ex1[,c("se1","se2","se3")])
```

```
##       se1               se2               se3
##  Min.   :0.01265   Min.   :0.01265   Min.   :0.01265
##  1st Qu.:0.01281   1st Qu.:0.01281   1st Qu.:0.01281
##  Median :0.01342   Median :0.01342   Median :0.01342
##  Mean   :0.01481   Mean   :0.01481   Mean   :0.01481
##  3rd Qu.:0.01532   3rd Qu.:0.01532   3rd Qu.:0.01532
##  Max.   :0.02883   Max.   :0.02883   Max.   :0.02883
```

There are some variation among them for each variable. Let's try scaling the effects by allele frequencies and check the resulting distributions. (For info about "scaled effects" in this context, see

https://www.mv.helsinki.fi/home/mjxpirin/GWAS_course/material/GWAS7.html.)

```
sc = sqrt(2 * linemodels.ex1$maf * (1 - linemodels.ex1$maf) ) #scaling constants
summary(linemodels.ex1[,c("se1","se2","se3")]*sc)
```

```
##       se1                se2                se3
##  Min.   :0.008944   Min.   :0.008944   Min.   :0.008944
##  1st Qu.:0.008944   1st Qu.:0.008944   1st Qu.:0.008944
##  Median :0.008944   Median :0.008944   Median :0.008944
##  Mean   :0.008944   Mean   :0.008944   Mean   :0.008944
##  3rd Qu.:0.008944   3rd Qu.:0.008944   3rd Qu.:0.008944
##  Max.   :0.008944   Max.   :0.008944   Max.   :0.008944
```

Now they became perfectly constant, so we will work with the scaled effects from now on, and then we can set `assume.constant.SE = TRUE` to speed up the computation.

```
X = sc * linemodels.ex1[,c("beta1","beta2","beta3")]
SE = sc * linemodels.ex1[,c("se1","se2","se3")]
```

Case of one linemodel ($K = 1$) We fit a single linemodel to the whole data. We consider all $M = 3$ dimensions, and estimate all parameters. We use generic initial values (scale = 1, slopes = 1, cors = 0.99), and we assume that `r.lkhood` is a diagonal identity matrix (default) to denote that there are no correlations between the estimators of different effects. In the genetics context, this is true if all 3 estimates come from different studies.

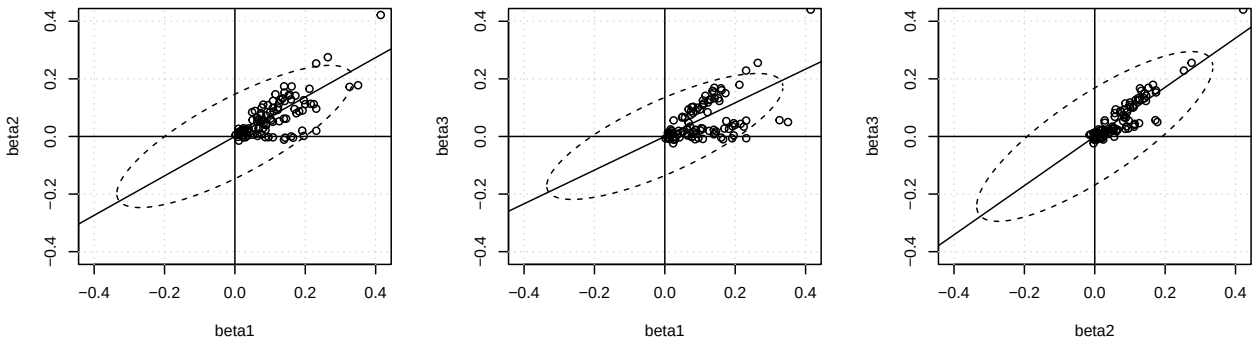
```
par.include.1 = list(scales = c(TRUE),
                     slopes = matrix(c(TRUE, TRUE), ncol = 2),
                     cors = c(TRUE))
linm.1 = line.models.optimize(
  X, SE,
  par.include = par.include.1,
  force.same.scales = FALSE,
```

```
  init.scales = c(1),
  init.slopes = matrix(c(1,1), nrow = 1),
  init.cors = c(0.99),
  op.method = "BFGS",
  assume.constant.SE = TRUE)
```

```
##
##
## Initial values
## scales: 1
## slopes:
##  1, 1
## cors: 0.99
## proportions: 1
## Initial log-lkhood: 99.480794
## Optimizing w.r.t: scale1 slope1_1 slope1_2 cor1
## Convergence criteria:
##  Relative diff in parameters <= 0  or
##  Difference in log-likelihood < 0.001
##
## iter: 2 ; log-lkhood: 356.2546
## Relative diffs in optimized parameters: 0, 0, 0, 0
## proportions: 1
## scales: 0.1371
## slopes:
##  0.6847, 0.5844
## cors: 0.8235
## Parameter values converged.
## Log-likelihood value converged.
```

The fitted line has scale 0.1371, slopes (0.6847, 0.5844) and cors 0.8235. Let's visualize the 3 pairwise plots of the effect estimates with the fitted model.

```
par(mfrow = c(1,3))
visualize.multidimensional.line.models(
  scales = linm.1$scales, linm.1$slopes, linm.1$cors,
  plot.pairs = rbind(c(1,2), c(1,3), c(2,3)),
  var.names = paste0("beta",1:3),
  legend.position = NULL,
  X = X, SE = NULL,
  undetermined.col = "black")
```



While the single linemodel captures the main trend of positive correlation in the data, clearly there are additional patterns that encourage us to fit more lines. Let's do it with two lines. To make the lines more separable from each other, we will force correlation parameters to be high (here set to 0.99). We will also use "Nelder-Mead" method to optimize since "BFGS" fails to converge (you can try it).

```
par.include.2 = list(scales = c(TRUE, TRUE),
                     slopes = matrix(rep(TRUE,4), ncol = 2),
                     cors = c(FALSE,FALSE))
linm.2 = line.models.optimize(
  X, SE,
  par.include = par.include.2,
  force.same.scales = FALSE,
  init.scales = c(0.15, 0.15),
  init.slopes = matrix(c(1,1, 0.5,0.5), nrow = 2, byrow = TRUE),
  init.cors = c(0.99, 0.99),
  model.names = NULL,
  model.priors = c(0.5, 0.5),
  assume.constant.SE = TRUE,
  op.method = "Nelder-Mead")
```

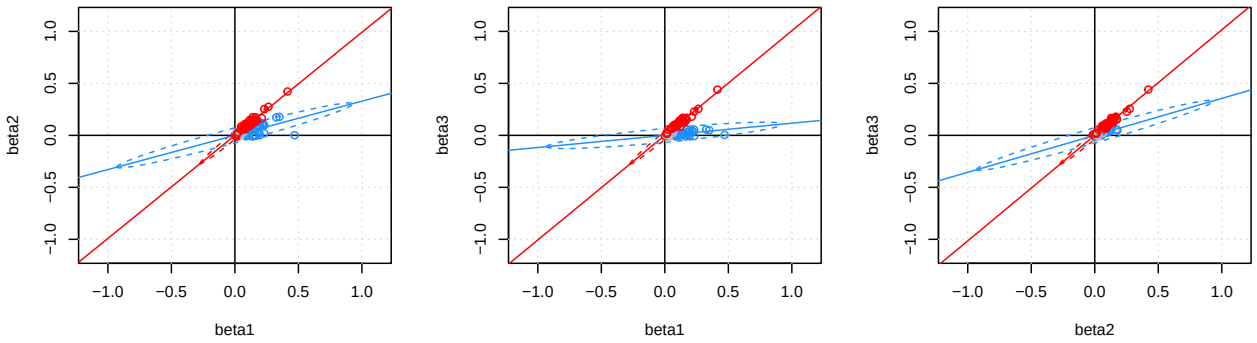
```
##
##
## Initial values
## scales: 0.15, 0.15
## slopes:
##  1, 1
##  0.5, 0.5
## cors: 0.99, 0.99
## proportions: 0.5, 0.5
## Initial log-lkhood: 8.7059771
## Optimizing w.r.t: scale1 scale2 slope1_1 slope2_1 slope1_2 slope2_2
## Convergence criteria:
##  Relative diff in parameters <= 0  or
##  Difference in log-likelihood < 0.001
##
## iter: 6 ; log-lkhood: 452.15086
## Relative diffs in optimized parameters: 0.000222, 0.00111, 1.84e-06, 6.21e-05, 0.000111, 6.58e-05
## proportions: 0.559, 0.441
## scales: 0.1128, 0.3799
## slopes:
##  0.9914, 1.006
##  0.3289, 0.1169
## cors: 0.99, 0.99
## Log-likelihood value converged.
```

Since we now have more than one model, let's also estimate the membership probabilities of each observation in each model.

```
res.2 = line.models.with.proportions(X, SE,
                                     scales = linm.2$scales,
                                     slopes = linm.2$slopes,
                                     cors = linm.2$cors,
                                     verbose = FALSE)
```

Let's visualize the results by coloring the points that have membership probability > 80% in a particular model.

```
par(mfrow = c(1,3))
visualize.multidimensional.line.models(
  scales = linm.2$scales, linm.2$slopes, linm.2$cors,
  plot.pairs = rbind(c(1,2), c(1,3), c(2,3)),
  var.names = paste0("beta",1:3),
  legend.position = NULL,
  model.cols = c("red","dodgerblue"),
  model.thresh = 0.8,
  model.prob = res.2$groups,
  X = X, SE = NULL,
  undetermined.col = "gray")
```



This starts to look good. The proportion of points in each model is given in the results:

```
res.2$params
```

```
##          mean    95%low     95%up         sd
## Mo1 0.5481447 0.4534030 0.6531943 0.05454432
## Mo2 0.4518553 0.3468057 0.5465970 0.05454432
```

Thus, roughly half of the points are estimated to originate from each of the models.

Let's fit a third linemodel, and visualize the results.

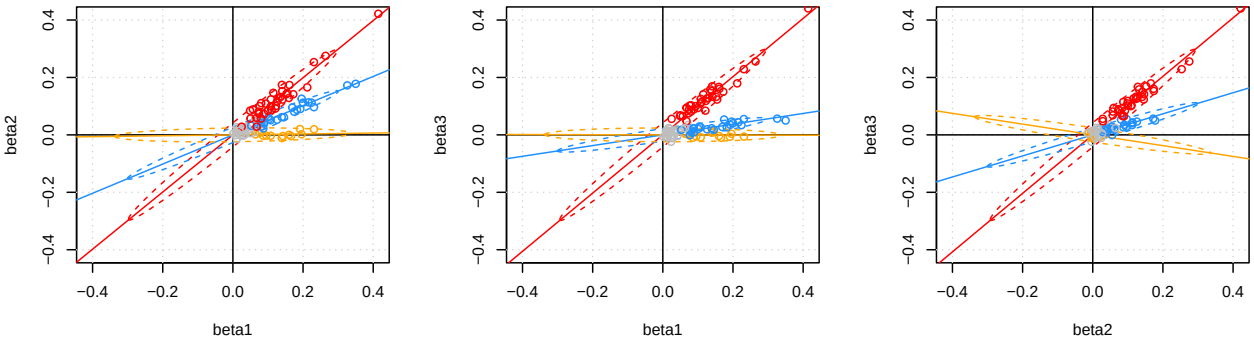
```
par.include.3 = list(scales = rep(TRUE, 3),
                     slopes = matrix(rep(TRUE,6), ncol = 2),
                     cors = rep(FALSE, 3))
linm.3 = line.models.optimize(
  X, SE,
  par.include = par.include.3,
  force.same.scales = FALSE,
  init.scales = c(0.15, 0.15, 0.15),
  init.slopes = matrix(c(1,1, 0.5,0.5, 0,0), nrow = 3, byrow = TRUE),
  init.cors = c(0.99, 0.99, 0.99),
  assume.constant.SE = TRUE,
  op.method = "BFGS")
```

```
##
##
```

```
## Initial values
## scales: 0.15, 0.15, 0.15
## slopes:
##  1, 1
##  0.5, 0.5
##  0, 0
## cors: 0.99, 0.99, 0.99
## proportions: 0.333, 0.333, 0.333
## Initial log-lkhood: 389.48131
## Optimizing w.r.t: scale1 scale2 scale3 slope1_1 slope2_1 slope3_1 slope1_2 slope2_2 slope3_2
## Convergence criteria:
##  Relative diff in parameters <= 0  or
##  Difference in log-likelihood < 0.001
##
## iter: 6 ; log-lkhood: 554.98014
## Relative diffs in optimized parameters: 0.00021, 0.00121, 0.00304, 6.73e-05, 0.000294, 0.000668, 6.6
## proportions: 0.424, 0.391, 0.185
## scales: 0.1217, 0.1228, 0.1375
## slopes:
##  0.9953, 1.014
##  0.5086, 0.1865
##  0.01571, -0.002945
## cors: 0.99, 0.99, 0.99
## Log-likelihood value converged.
```

Now "BFGS" works again. We have similar scales in each model, and the slopes are close to the true values that were used to generate these data, namely, (1,1), (0.5,0.2) and (0,0). The proportion parameters also match well to their true values in data generation (40%, 40%, 20%).

```
res.3 = line.models.with.proportions(X, SE,
                                     scales = linm.3$scales,
                                     slopes = linm.3$slopes,
                                     cors = linm.3$cors,
                                     verbose = FALSE)
par(mfrow = c(1,3))
visualize.multidimensional.line.models(
  scales = linm.3$scales, linm.3$slopes, linm.3$cors,
  plot.pairs = rbind(c(1,2), c(1,3), c(2,3)),
  var.names = paste0("beta",1:3),
  legend.position = NULL,
  model.cols = c("red","dodgerblue","orange"),
  model.prob = res.3$groups,
  model.thresh = 0.8,
  X = X, SE = NULL,
  undetermined.col = "gray")
```



The gray points are not assigned to any of the linemodels with a probability $> 80\%$. They tend to be close to the origin, where the models are less separable from each other.

Assessment of a need for additional linemodels

How could we assess whether the 3-line model improves over the 2-line model? We can use `simulate.loglr()` to generate data sets under the null hypothesis (here the 2-line model) and compute log of likelihood ratio between the 3-line model and the 2-line model for each data set. For each data set, we always optimize the model parameters for both models just like we did with our original data set. If the observed logLR value is larger than those under the null, then we may conclude that the 3-line model seems to improve over the 2-line model. If, instead, the observed logLR is not larger than a typical logLR value generated under the null, then we can conclude that there is little statistical evidence for a need for a more complex model over the null model.

```
loglrs = simulate.loglr(
  X, SE, n.sims = 100,
  par.include.null = par.include.2,
  force.same.scales.null = FALSE,
  init.scales.null = c(0.15, 0.15),
  init.slopes.null = matrix(c(1,1, 0.33,0.12), nrow = 2, byrow = TRUE),
  init.cors.null = c(0.99, 0.99),
  par.include.alt = par.include.3,
  force.same.scales.alt = FALSE,
  init.scales.alt = c(0.15, 0.15, 0.15),
  init.slopes.alt = matrix(c(1,1, 0.51,0.19, 0.02, 0.00), nrow = 3, byrow = TRUE),
  init.cors.alt = c(0.99, 0.99, 0.99),
  r.lkhood = rep(0, ncol(X)*(ncol(X)-1)/2),
  tol.loglk = 1e-3,
  tol.par = 0,
  op.method = "Nelder-Mead",
  assume.constant.SE = TRUE,
  print.steps = c(0,0))
```

The observed logLR was

```
loglrs$obs.loglr
```

```
## [1] 102.8293
```

while the 100 simulations under the null model gave the following distribution of logLR values:

```r
summary(loglrs$null.loglr)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.731   6.449   8.142   8.838  10.808  18.759
```

Thus, we can conlcude that the observed logLR is much larger than we expect under the null. We could say that an empirical P-value under the null is below 0.01 since none of 100 data set under the null gave logLR value larger than the observed one. By doing more simulations, we could estimate even a smaller upper bound estimate for a P-value.

NOTE: There is no theoretical distribution for logLR available and this simulation approach is based on several assumptions that may not always hold. Therefore, these empirical P-values should be used only as a guideline, not as an exact method.