# Package: DImodelsVis (via r-universe)

September 9, 2024

**Title** Visualising and Interpreting Statistical Models Fit to Compositional Data

**Version** 1.0.1

**Description** Statistical models fit to compositional data are often difficult to interpret due to the sum to 1 constraint on data variables. 'DImodelsVis' provides novel visualisations tools to aid with the interpretation of models fit to compositional data. All visualisations in the package are created using the 'ggplot2' plotting framework and can be extended like every other 'ggplot' object.

**License** GPL (>= 3)

**Encoding** UTF-8

**LazyData** true

**Imports** cli, DImodels (>= 1.3.1), dplyr (>= 1.0.0), forcats, ggfortify, ggplot2, ggtext (>= 0.1.2), glue, grDevices, insight, methods, metR, PieGlyph, plotwidgets, rlang, scales, stats, tidyr, utils

**RoxygenNote** 7.3.1

**Suggests** DImodelsMulti (>= 1.0.0), knitr, rmarkdown, spelling, testthat (>= 3.0.0)

**URL** https://rishvish.github.io/DImodelsVis/

**VignetteBuilder** knitr

**Language** en-US

**Config/testthat/edition** 3

**Repository** https://rishvish.r-universe.dev

**RemoteUrl** https://github.com/rishvish/dimodelsvis

**RemoteRef** HEAD

**RemoteSha** 21e8991e85d56b3f08447bd75df5b4f9aca29419

# Contents

`add_add_var` *Add additional variables to the data*

## Description

Utility function for incorporating any additional variables into the data. Each row in the data will be replicated and new columns will be added for each variable specified in 'add_var' with values corresponding to their cartesian product.

## Usage

```
add_add_var(data, add_var = NULL)
```

## Arguments

| | |
|---|---|
| `data` | A data frame containing the data in which to add the additional variables. |
| `add_var` | A named list or data-frame specifying the names and corresponding values of each new variable to add to the data. If a list is specified, each row in the data would be replicated for each unique combination of values of the specified variables (i.e., their cartesian product) in 'add_var', while specifying a data-frame would replicate each row in the data for each row in add_var (i.e., merge the two data-frames). |

## Value

A data-frame with all additional columns specified in 'add_var' and the following additional column.

**.add_str_ID** A unique identifier describing each element from the cartesian product of all variables specified in 'add_var'.

## Examples

```
test_data <- data.frame(diag(1, 3))
print(test_data)

## Adding a single variable
add_add_var(data = test_data,
            add_var = list("Var1" = c(10, 20)))

## Specifying multiple variables as a list will add values for
##  each unique combination
add_add_var(data = test_data,
            add_var = list("Var1" = c(10, 20),
                           "Var2" = c(30, 40)))

## Specifying add_var as a data.frame would simply merge the two data-frames
add_add_var(data = test_data,
            add_var = data.frame("Var1" = c(10, 20),
                                    "Var2" = c(30, 40)))

## If the list specified in `add_var` is not named, then the additional
## variables will be automatically named Var1, Var2, Var3, etc.
add_add_var(data = test_data,
            add_var = list(c(1, 2), c(3, 4)))
```

## add_ID_terms — *Add identity effect groups used in a Diversity-Interactions (DI) model to new data*

### Description

Utility function that accepts a fitted Diversity-Interactions (DI) model object along with a data frame and adds the appropriate species identity effect groupings to the data for making predictions.

### Usage

```
add_ID_terms(data, model)
```

### Arguments

| | |
|---|---|
| `data` | A data-frame with species proportions that sum to 1 to create the identity effect groupings. |
| `model` | A Diversity Interactions model object fit using the `DI()` or `autoDI()` functions from the `DImodels` or `DImulti()` from the `DImodelsMulti` R packages. |

### Value

A data-frame with additional columns appended to the end that contain the grouped species proportions.

### Examples

```
library(DImodels)
data(sim1)

# Fit DI models with different ID effect groupings
mod1 <- DI(y = "response", prop = 3:6,
           data = sim1, DImodel = "AV") # No ID grouping
mod2 <- DI(y = "response", prop = 3:6,
           data = sim1, DImodel = "AV",
           ID = c("ID1", "ID1", "ID2", "ID2"))
mod3 <- DI(y = "response", prop = 3:6,
           data = sim1, DImodel = "AV",
           ID = c("ID1", "ID1", "ID1", "ID1"))

# Create new data for adding interaction terms
newdata <- sim1[sim1$block == 1, 3:6]
```

```
print(head(newdata))

add_ID_terms(data = newdata, model = mod1)
add_ID_terms(data = newdata, model = mod2)
add_ID_terms(data = newdata, model = mod3)
```

---

## `add_interaction_terms` *Add interaction terms used in a Diversity-Interactions (DI) model to new data*

---

### Description

Utility function that accepts a fitted Diversity-Interactions (DI) model object along with a data frame and adds the necessary interaction structures to the data for making predictions using the model object specified in ‘model‘.

### Usage

```
add_interaction_terms(data, model)
```

### Arguments

| | |
|---|---|
| `data` | A data-frame with species proportions that sum to 1 to create the appropriate interaction structures. |
| `model` | A Diversity Interactions model object fit using the `DI()` or `autoDI()` functions from the `DImodels` or `DImulti()` from the `DImodelsMulti` R packages. |

### Value

The original data-frame with additional columns appended at the end describing the interactions terms present in the model object.

### Examples

```
library(DImodels)
data(sim1)

# Fit different DI models
mod1 <- DI(y = "response", prop = 3:6, data = sim1, DImodel = "AV")
mod2 <- DI(y = "response", prop = 3:6, data = sim1, DImodel = "FULL")
mod3 <- DI(y = "response", prop = 3:6, data = sim1, DImodel = "ADD")
mod4 <- DI(y = "response", prop = 3:6, data = sim1,
           FG = c("G", "G", "H", "H"), DImodel = "FG")

# Create new data for adding interaction terms
newdata <- sim1[sim1$block == 1, 3:6]
print(head(newdata))

add_interaction_terms(data = newdata, model = mod1)
add_interaction_terms(data = newdata, model = mod2)
add_interaction_terms(data = newdata, model = mod3)
add_interaction_terms(data = newdata, model = mod4)
```

---

## `add_prediction` *Add predictions and confidence interval to data using either a model object or model coefficients*

---

### Description

Add predictions and confidence interval to data using either a model object or model coefficients

### Usage

```
add_prediction(
  data,
  model = NULL,
  coefficients = NULL,
  coeff_cols = NULL,
  vcov = NULL,
  interval = c("none", "confidence", "prediction"),
  conf.level = 0.95
)
```

### Arguments

| | |
|---|---|
| `data` | A data-frame containing appropriate values for all the terms in the model. |
| `model` | A regression model object which will be used to make predictions for the observations in ‘data‘. Will override ‘coefficients‘ if specified. |
| `coefficients` | If a regression model is not available (or can’t be fit in R), the regression coefficients from a model fit in some other language can be used to calculate predictions. However, the user would have to ensure there’s an appropriate one-to-one positional mapping between the data columns and the coefficient values. Further, they would also have to provide a variance-covariance matrix of the coefficients in the ‘vcov‘ parameter if they want the associated CI for the prediction or it would not be possible to calculate confidence/prediction intervals using this method. |
| `coeff_cols` | If ‘coefficients‘ are specified and a one-to-one positional mapping between the data-columns and coefficient vector is not present. A character string or numeric index can be specified here to reorder the data columns and match the corresponding coefficient value to the respective data column. See the "Use model coefficients for prediction" section in examples. |
| `vcov` | If regression coefficients are specified, then the variance-covariance matrix of the coefficients can be specified here to calculate the associated confidence interval around each prediction. Failure to do so would result in no confidence intervals being returned. Ensure ‘coefficients‘ and ‘vcov‘ have the same positional mapping with the data. |

| | |
|---|---|
| `interval` | Type of interval to calculate:<br>**"none" (default)** No interval to be calculated.<br>**"confidence"** Calculate a confidence interval.<br>**"prediction"** Calculate a prediction interval. |
| `conf.level` | The confidence level for calculating confidence/prediction intervals. Default is 0.95. |

## Value

A data-frame with the following additional columns

**.Pred** The predicted response for each observation.

**.Lower** The lower limit of the confidence/prediction interval for each observation (will be same as ".Pred" if using ‘coefficients‘ and ‘vcov‘ is not specified).

**.Upper** The lower limit of the confidence/prediction interval for each observation (will be same as ".Pred" if using ‘coefficients‘ and ‘vcov‘ is not specified).

## Examples

```
library(DImodels)
data(sim1)

# Fit a model
mod <- lm(response ~ 0 + p1 + p2 + p3 + p4 + p1:p2 + p3:p4, data = sim1)

# Create new data for adding predictions
newdata <- head(sim1[sim1$block == 1,])
print(newdata)

# Add predictions to data
add_prediction(data = newdata, model = mod)

# Adding predictions to data with confidence interval
add_prediction(data = newdata, model = mod, interval = "confidence")

# Calculate prediction intervals instead
add_prediction(data = newdata, model = mod, interval = "prediction")

# Default is a 95% interval, change to 99%
add_prediction(data = newdata, model = mod, interval = "prediction",
               conf.level = 0.99)

####################################################################
##### Use model coefficients for prediction
coeffs <- mod$coefficients

# Would now have to add columns corresponding to each coefficient in the
# data and ensure there is an appropriate mapping between data columns and
# the coefficients.
newdata$`p1:p2` = newdata$p1 * newdata$p2
```

```
newdata$`p3:p4` = newdata$p3 * newdata$p4

# If the coefficients are named then the function will try to
# perform matching between data columns and the coefficients
# Notice that confidence intervals are not produced if we don't
# specify a variance covariance matrix
add_prediction(data = newdata, coefficients = coeffs)

# However, if the coefficients are not named
# The user would have to manually specify the subset
# of data columns arranged according to the coefficients
coeffs <- unname(coeffs)

subset_data <- newdata[, c(3:6, 8,9)]
subset_data # Notice now we have the exact columns in data as in coefficients
add_prediction(data = subset_data, coefficients = coeffs)

# Or specify a selection (either by name or index) in coeff_cols
add_prediction(data = newdata, coefficients = coeffs,
               coeff_cols = c("p1", "p2", "p3", "p4", "p1:p2", "p3:p4"))

add_prediction(data = newdata, coefficients = coeffs,
               coeff_cols = c(3, 4, 5, 6, 8, 9))

# Adding confidence intervals when using model coefficients
coeffs <- mod$coefficients
# We need to provide a variance-covariance matrix to calculate the CI
# when using `coefficients` argument. The following warning will be given
add_prediction(data = newdata, coefficients = coeffs,
               interval = "confidence")

vcov_mat <- vcov(mod)
add_prediction(data = newdata, coefficients = coeffs,
               interval = "confidence", vcov = vcov_mat)

# Currently both confidence and prediction intervals will be the same when
# using this method
add_prediction(data = newdata, coefficients = coeffs,
               interval = "prediction", vcov = vcov_mat)
```

---

## `conditional_ternary` *Conditional ternary diagrams*

---

### Description

We fix $n-3$ variables to have a constant value $p_1, p_2, p_3, ...p_{n-3}$ such that $P = p_1+p_2+p_3+...p_{n-3}$ and $0 \leq P \leq 1$ and vary the proportion of the remaining three variables between 0 and $1 - P$ to visualise the change in the predicted response as a contour map within a ternary diagram. This is equivalent to taking multiple 2-d slices of the high dimensional simplex space. Taking multiple 2-d slices across multiple variables should allow to create an approximation of how the response varies across the n-dimensional simplex. This is a wrapper function specifically for statistical models fit using the `DI()` function from the `DImodels` R package and would implicitly call `conditional_ternary_data` followed by `conditional_ternary_plot`. If your model object isn't fit using DImodels, consider calling these functions manually.

## Usage

```
conditional_ternary(
  model,
  FG = NULL,
  values = NULL,
  tern_vars = NULL,
  conditional = NULL,
  add_var = list(),
  resolution = 3,
  plot = TRUE,
  nlevels = 7,
  colours = NULL,
  lower_lim = NULL,
  upper_lim = NULL,
  contour_text = TRUE,
  show_axis_labels = TRUE,
  show_axis_guides = FALSE,
  axis_label_size = 4,
  vertex_label_size = 5,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| `model` | A Diversity Interactions model object fit by using the `DI()` function from the `DImodels` package. |
| `FG` | A character vector specifying the grouping of the variables specified in 'prop'. Specifying this parameter would call the grouped_ternary_data function internally. See `grouped_ternary` or `grouped_ternary_data` for more information. |
| `values` | A numeric vector specifying the proportional split of the variables within a group. The default is to split the group proportion equally between each variable in the group. |
| `tern_vars` | A character vector giving the names of the three variables to be shown in the ternary diagram. |
| `conditional` | A data-frame describing the names of the compositional variables and their respective values at which to slice the simplex space. The format should be, for example, as follows: `data.frame("p1" = c(0, 0.5), "p2" = c(0.2, 0.1))` One figure would be created for each row in 'conditional' with the respective |

| | |
|---|---|
| | values of all specified variables. Any compositional variables not specified in ‘conditional‘ will be assumed to be 0. |
| `add_var` | A list or data-frame specifying values for additional variables in the model other than the proportions (i.e. not part of the simplex design). This could be useful for comparing the predictions across different values for a non-compositional variable. If specified as a list, it will be expanded to show a plot for each unique combination of values specified, while if specified as a data-frame, one plot would be generated for each row in the data. |
| `resolution` | A number between 1 and 10 describing the resolution of the resultant graph. A high value would result in a higher definition figure but at the cost of being computationally expensive. |
| `plot` | A boolean variable indicating whether to create the plot or return the prepared data instead. The default `TRUE` creates the plot while `FALSE` would return the prepared data for plotting. Could be useful if user wants to modify the data first and then create the plot. |
| `nlevels` | The number of levels to show on the contour map. |
| `colours` | A character vector or function specifying the colours for the contour map or points. The number of colours should be same as ‘nlevels‘ if (‘show = "contours"‘).<br>The default colours scheme is the `terrain.colors()` for continuous variables and an extended version of the Okabe-Ito colour scale for categorical variables. |
| `lower_lim` | A number to set a custom lower limit for the contour (if ‘show = "contours"‘). The default is minimum of the prediction. |
| `upper_lim` | A number to set a custom upper limit for the contour (if ‘show = "contours"‘). The default is maximum of the prediction. |
| `contour_text` | A boolean value indicating whether to include labels on the contour lines showing their values (if ‘show = "contours"‘). The default is `TRUE`. |
| `show_axis_labels` | A boolean value indicating whether to show axis labels along the edges of the ternary. The default is `TRUE`. |
| `show_axis_guides` | A boolean value indicating whether to show axis guides within the interior of the ternary. The default is `FALSE`. |
| `axis_label_size` | A numeric value to adjust the size of the axis labels in the ternary plot. The default size is 4. |
| `vertex_label_size` | A numeric value to adjust the size of the vertex labels in the ternary plot. The default size is 5. |
| `nrow` | Number of rows in which to arrange the final plot (when ‘add_var‘ is specified). |
| `ncol` | Number of columns in which to arrange the final plot (when ‘add_var‘ is specified). |

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if ‘plot = FALSE‘)

## Examples

```
library(DImodels)
library(dplyr)
data(sim2)
m1 <- DI(y = "response", data = sim2, prop = 3:6, DImodel = "FULL")

#' ## Create data for slicing
## We only condition on the variable "p3"
conditional_ternary(model = m1, tern_vars = c("p1", "p2", "p4"),
                    conditional = data.frame("p3" = c(0, 0.2, 0.5)),
                    resolution = 1)

## Slices for experiments for over 4 variables
data(sim4)
m2 <- DI(y = "response", prop = paste0("p", 1:6),
         DImodel = "AV", data = sim4) %>%
         suppressWarnings()

## Conditioning on multiple variables
cond <- data.frame(p4 = c(0, 0.2), p3 = c(0.5, 0.1), p6 = c(0, 0.3))
conditional_ternary(model = m2, conditional = cond,
                    tern_vars = c("p1", "p2", "p5"), resolution = 1)

## Create separate plots for additional variables not a part of the simplex
m3 <- DI(y = "response", prop = paste0("p", 1:6),
         DImodel = "AV", data = sim4, treat = "treatment") %>%
         suppressWarnings()

## Create plot and arrange it using nrow and ncol

conditional_ternary(model = m3, conditional = cond[1, ],
                    tern_vars = c("p1", "p2", "p5"),
                    resolution = 1,
                    add_var = list("treatment" = c(50, 150)),
                    nrow = 2, ncol = 1)


## Specify `plot = FALSE` to not create the plot but return the prepared data
head(conditional_ternary(model = m3, conditional = cond[1, ],
                         resolution = 1, plot = FALSE,
                         tern_vars = c("p1", "p2", "p5"),
                         add_var = list("treatment" = c(50, 150))))
```

---

## conditional_ternary_data *Conditional ternary diagrams*

---

## Description

The helper function for preparing the underlying data for creating conditional ternary diagrams, where we fix $n - 3$ variables to have a constant value $p_1, p_2, p_3, ..., p_{n-3}$ such that $P = p_1 + p_2 +$

$p_3 + ...p_{n-3}$ and $0 \leq P \leq 1$ and vary the proportion of the remaining three variables between $0$ and $1 - P$ to visualise the change in the predicted response as a contour map within a ternary diagram. The output of this function can be passed to the `conditional_ternary_plot` function to plot the results. Viewing multiple 2-d slices across multiple variables should allow to create an approximation of how the response varies across the n-dimensional simplex.

## Usage

```
conditional_ternary_data(
  prop,
  FG = NULL,
  values = NULL,
  tern_vars = NULL,
  conditional = NULL,
  add_var = list(),
  resolution = 3,
  prediction = TRUE,
  ...
)
```

## Arguments

| | |
|---|---|
| `prop` | A character vector indicating the model coefficients corresponding to variable proportions. These variables should be compositional in nature (i.e., proportions should sum to 1). |
| `FG` | A character vector specifying the grouping of the variables specified in ‘prop‘. Specifying this parameter would call the grouped_ternary_data function internally. See `grouped_ternary` or `grouped_ternary_data` for more information. |
| `values` | A numeric vector specifying the proportional split of the variables within a group. The default is to split the group proportion equally between each variable in the group. |
| `tern_vars` | A character vector giving the names of the three variables to be shown in the ternary diagram. |
| `conditional` | A data-frame describing the names of the compositional variables and their respective values at which to slice the simplex space. The format should be, for example, as follows:<br>`data.frame("p1" = c(0, 0.5), "p2" = c(0.2, 0.1))`<br>One figure would be created for each row in ‘conditional‘ with the respective values of all specified variables. Any compositional variables not specified in ‘conditional‘ will be assumed to be 0. |
| `add_var` | A list or data-frame specifying values for additional variables in the model other than the proportions (i.e. not part of the simplex design). This could be useful for comparing the predictions across different values for a non-compositional variable. If specified as a list, it will be expanded to show a plot for each unique combination of values specified, while if specified as a data-frame, one plot would be generated for each row in the data. |

| | |
|---|---|
| `resolution` | A number between 1 and 10 describing the resolution of the resultant graph. A high value would result in a higher definition figure but at the cost of being computationally expensive. |
| `prediction` | A logical value indicating whether to pass the final data to the ‘add_prediction‘ function and append the predictions to the data. Default value is `TRUE`, but often it would be desirable to make additional changes to the data before making any predictions, so the user can set this to `FALSE` and manually call the ‘add_prediction‘ function. |
| `...` | Arguments passed on to `add_prediction` |

`model` A regression model object which will be used to make predictions for the observations in ‘data‘. Will override ‘coefficients‘ if specified.

`coefficients` If a regression model is not available (or can’t be fit in R), the regression coefficients from a model fit in some other language can be used to calculate predictions. However, the user would have to ensure there’s an appropriate one-to-one positional mapping between the data columns and the coefficient values. Further, they would also have to provide a variance-covariance matrix of the coefficients in the ‘vcov‘ parameter if they want the associated CI for the prediction or it would not be possible to calculate confidence/prediction intervals using this method.

`vcov` If regression coefficients are specified, then the variance-covariance matrix of the coefficients can be specified here to calculate the associated confidence interval around each prediction. Failure to do so would result in no confidence intervals being returned. Ensure ‘coefficients‘ and ‘vcov‘ have the same positional mapping with the data.

`coeff_cols` If ‘coefficients‘ are specified and a one-to-one positional mapping between the data-columns and coefficient vector is not present. A character string or numeric index can be specified here to reorder the data columns and match the corresponding coefficient value to the respective data column. See the "Use model coefficients for prediction" section in examples.

`conf.level` The confidence level for calculating confidence/prediction intervals. Default is 0.95.

`interval` Type of interval to calculate:

- **"none" (default)** No interval to be calculated.
- **"confidence"** Calculate a confidence interval.
- **"prediction"** Calculate a prediction interval.

## Value

A data-frame containing compositional columns with names specified in ‘prop‘ parameter along with any additional columns specified in ‘add_var‘ parameter. The first five columns of the data contain the three variables (specified in ‘tern_vars‘) shown in the ternary along with their 2-d projection and should not be modified. The following additional columns could also be present in the data.

**.x** The x-projection of the points within the ternary.

**.y** The y-projection of the points within the ternary.

**.add_str_ID** An identifier column for grouping the cartesian product of all additional columns specified in ‘add_var‘ parameter (if ‘add_var‘ is specified).

**.Sp** An identifier column specifying the variable(s) along which the high dimensional simplex is sliced.

**.Value** The value(s) (between 0 and 1) along the direction of variable(s) in ‘.Sp‘ at which the high dimensional simplex is sliced.

**.Facet** An identifier column formed by combining ‘.Sp‘ and ‘.value‘ to group observations within a specific slice of the high dimensional simplex.

**.Pred** The predicted response for each observation (if ‘prediction‘ is `TRUE`).

**.Lower** The lower limit of the prediction/confidence interval for each observation.

**.Upper** The upper limit of the prediction/confidence interval for each observation.

## Examples

```
library(DImodels)

## Load data
data(sim4)

## Fit model
mod <- glm(response ~ 0 + (p1 + p2 + p3 + p4 + p5 + p6)^2, data = sim4)

## Create data
## Any species not specified in `tern_vars` or conditional will be assumed
## to be 0, for example p5 and p6 here.
head(conditional_ternary_data(prop = c("p1", "p2", "p3", "p4", "p5", "p6"),
                              tern_vars = c("p1", "p2", "p3"),
                              conditional = data.frame("p4" = c(0, 0.2, 0.5)),
                              model = mod,
                              resolution = 1))

## Can also condition on multiple species
cond <- data.frame(p4 = c(0, 0.2), p5 = c(0.5, 0.1), p6 = c(0, 0.3))
cond
head(conditional_ternary_data(prop = c("p1", "p2", "p3", "p4", "p5", "p6"),
                              tern_vars = c("p1", "p2", "p3"),
                              conditional = cond,
                              model = mod,
                              resolution = 1))

## Fit model
mod <- glm(response ~ 0 + (p1 + p2 + p3 + p4 + p5 + p6)^2 + treatment,
           data = sim4)

## Can also add any additional variables independent of the simplex
## Notice the additional `.add_str_ID` column
head(conditional_ternary_data(prop = c("p1", "p2", "p3", "p4", "p5", "p6"),
                              tern_vars = c("p1", "p2", "p3"),
                              conditional = data.frame("p4" = c(0, 0.2, 0.5)),
                              add_var = list("treatment" = c(50, 150)),
                              model = mod,
                              resolution = 1))

## It could be desirable to take the output of this function and add
## additional variables to the data before making predictions
## Use `prediction = FALSE` to get data without any predictions
cond_data <- conditional_ternary_data(prop = c("p1", "p2", "p3", "p4", "p5", "p6"),
                                      tern_vars = c("p1", "p2", "p3"),
                                      conditional = data.frame("p4" = c(0, 0.2, 0.5)),
                                      prediction = FALSE,
                                      resolution = 1)
## The data can then be modified and the `add_prediction` function can be
## called manually using either the model object or model coefficients
cond_data$treatment <- 50
head(add_prediction(data = cond_data, model = mod))
```

## conditional_ternary_plot
*Conditional ternary diagrams*

### Description

The helper function for plotting conditional ternary diagrams. The output of the ‘`conditional_ternary_data`‘ should be passed here to visualise the n-dimensional simplex space as 2-d slices showing the change in the response across any three variables, when the other variables are conditioned to have fixed values.

### Usage

```
conditional_ternary_plot(
  data,
  col_var = ".Pred",
  nlevels = 7,
  colours = NULL,
  lower_lim = NULL,
  upper_lim = NULL,
  tern_labels = colnames(data)[1:3],
  contour_text = TRUE,
  show_axis_labels = TRUE,
  show_axis_guides = FALSE,
  points_size = 2,
  axis_label_size = 4,
  vertex_label_size = 5,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| `data` | A data-frame which is the output of the ‘conditional_ternary_data‘ function. |
| `col_var` | The column name containing the variable to be used for colouring the contours or points. The default is ".Pred". |
| `nlevels` | The number of levels to show on the contour map. |
| `colours` | A character vector or function specifying the colours for the contour map or points. The number of colours should be same as ‘nlevels‘ if (‘show = "contours"‘).<br>The default colours scheme is the `terrain.colors()` for continuous variables and an extended version of the Okabe-Ito colour scale for categorical variables. |
| `lower_lim` | A number to set a custom lower limit for the contour (if ‘show = "contours"‘). The default is minimum of the prediction. |
| `upper_lim` | A number to set a custom upper limit for the contour (if ‘show = "contours"‘). The default is maximum of the prediction. |
| `tern_labels` | A character vector containing the labels of the vertices of the ternary. The default is the column names of the first three columns of the data, with the first column corresponding to the top vertex, second column corresponding to the left vertex and the third column corresponding to the right vertex of the ternary. |
| `contour_text` | A boolean value indicating whether to include labels on the contour lines showing their values (if ‘show = "contours"‘). The default is `TRUE`. |
| `show_axis_labels` | A boolean value indicating whether to show axis labels along the edges of the ternary. The default is `TRUE`. |
| `show_axis_guides` | A boolean value indicating whether to show axis guides within the interior of the ternary. The default is `FALSE`. |
| `points_size` | If showing points, then a numeric value specifying the size of the points. |
| `axis_label_size` | A numeric value to adjust the size of the axis labels in the ternary plot. The default size is 4. |
| `vertex_label_size` | A numeric value to adjust the size of the vertex labels in the ternary plot. The default size is 5. |
| `nrow` | Number of rows in which to arrange the final plot (when ‘add_var‘ is specified). |
| `ncol` | Number of columns in which to arrange the final plot (when ‘add_var‘ is specified). |

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if ‘plot = FALSE‘)

## Examples

```
library(DImodels)

## Load data
data(sim4)

## Fit model
mod <- glm(response ~ 0 + (p1 + p2 + p3 + p4 + p5 + p6)^2, data = sim4)

## Create data for slicing
## We only condition on the variable "p3"
plot_data <- conditional_ternary_data(prop = c("p1", "p2", "p3", "p4", "p5", "p6"),
                                      tern_vars = c("p1", "p2", "p4"),
                                      conditional = data.frame("p3" = c(0, 0.2, 0.5)),
                                      model = mod,
                                      resolution = 1)

## Create plot
conditional_ternary_plot(data = plot_data)

## Condition on multiple variables
cond <- data.frame(p4 = c(0, 0.2), p5 = c(0.5, 0.1), p6 = c(0, 0.3))
cond
plot_data <- conditional_ternary_data(prop = c("p1", "p2", "p3", "p4", "p5", "p6"),
                                      tern_vars = c("p1", "p2", "p3"),
                                      conditional = cond,
                                      model = mod,
                                      resolution = 1)
## Create plot
conditional_ternary_plot(data = plot_data)

## Create multiple plots for additional variables using `add_var`
## Fit model

mod <- glm(response ~ 0 + (p1 + p2 + p3 + p4 + p5 + p6)^2 + treatment,
           data = sim4)

## Notice the additional `.add_str_ID` column
plot_data <- conditional_ternary_data(prop = c("p1", "p2", "p3", "p4", "p5", "p6"),
                                      tern_vars = c("p1", "p2", "p3"),
                                      conditional = data.frame("p4" = c(0, 0.2, 0.5)),
                                      add_var = list("treatment" = c(50, 150)),
                                      model = mod,
                                      resolution = 1)
## Create plot
## Use nrow to align plots
conditional_ternary_plot(data = plot_data, nrow = 2)
```

## `copy_attributes` *Copy attributes from one object to another*

## Description

This function copies over any additional attributes from ‘source‘ into ‘target‘. Any attributes already present in ‘target‘ would be left untouched. This function is useful after manipulating the data from the `*_data` preparation functions to ensure any attributes necessary for creating the plot aren’t lost.

## Usage

```
copy_attributes(target, source)
```

## Arguments

| | |
|---|---|
| `target` | The object to which attributes should be added. |
| `source` | The object whose attributes to copy. |

## Value

The object specified in ‘target‘ with all additional attributes in ‘source‘ object.

## Examples

```
## Simple example
a <- data.frame(Var1 = runif(1:10), Var2 = runif(1:10))
b <- data.frame(Var3 = runif(1:10), Var4 = runif(1:10))
attr(b, "attr1") <- "Lorem"
attr(b, "attr2") <- "ipsum"

print(attributes(a))
print(attributes(b))

## Copy over attributes of `b` into `a`
print(copy_attributes(target = a, source = b))
## Note the attributes already present in `a` are left untouched

## Can also be used in the dplyr pipeline
library(dplyr)

iris_sub <- iris[1:10, ]
attr(iris_sub, "attr1") <- "Lorem"
attr(iris_sub, "attr2") <- "ipsum"
attributes(iris_sub)

## Grouping can drop attributes we set
iris_sub %>%
   group_by(Species) %>%
   summarise(mean(Sepal.Length)) %>%
   attributes()

## Use copy_attributes with `iris_sub` object as source
##  to add the attributes again
iris_sub %>%
   group_by(Species) %>%
   summarise(mean(Sepal.Length)) %>%
   copy_attributes(source = iris_sub) %>%
   attributes()
```

---

`custom_filter` *Special custom filtering for compositional data*

---

## Description

A handy wrapper around the dplyr `filter()` function enabling the user to filter rows which satisfy specific conditions for compositional data like all equi-proportional communities, or communities with a given value of richness without having to make any changes to the data or adding any additional columns. All other functionalities are same as the dplyr `filter()` function.

## Usage

```
custom_filter(data, prop = NULL, special = NULL, ...)
```

## Arguments

| | |
|---|---|
| `data` | A data frame containing the compositional variables which should be used to perform the filtering. |
| `prop` | A character/numeric vector indicating the columns containing the compositional variables in ‘data‘. |
| `special` | A character string specifying the filtering condition. Four special keywords can be specified here for filtering 1. richness: A positive integer value to filter communities with a specific number of compositional variables (variables with non-zero values). 2. evenness: A numeric value between 0 and 1, to filter rows based on the relative abundances of the compositional variables where a higher value signifies a more even community with equal proportions of all variables. 3. equi: A boolean variable indicating whether to filter rows containing equi-proportional communities, i.e., communities where all variables have the same non-zero proportion. 4. monos: A boolean value indicating whether to filter communities containing a single compositional variable, i.e., richness == 1. These keywords can be combined using any logical operators and can even be combined with any other variables in the data. Please use the exact keywords (case-sensitive) in the query to get appropriate results. See examples for more details. |
| `...` | Any additional arguments specified to the dplyr `filter()` function. Filtering conditions for any additional variables can also be specified here. |

## Value

A subset of the original data which matches the specified filtering conditions.

## Examples

```
library(DImodels)
library(dplyr)

## Load data
data(sim3)

# The special filter keywords should be specified as a string
# Filter communities containing 3 species
head(custom_filter(data = sim3, prop = 4:12,
                   special = "richness == 3"))

# Filter communities at richness 6 OR evenness 0
head(custom_filter(data = sim3, prop = 4:12,
                   special = "richness == 6 | evenness == 0"), 12)

# Filter all monoculture AND treatment "A" (treatment is column present in data)
head(custom_filter(data = sim3, prop = 4:12,
                   special = "monos == TRUE & treatment == 'A'"), 10)

# Filter all equi proportional communities but NOT monocultures
head(custom_filter(data = sim3, prop = 4:12,
                   special = "equi == TRUE & monos == FALSE"))

# Can also use normal filter
sim3 %>% custom_filter(p1 == 1, special = NULL, prop = NULL)

# Both special filtering and normal filtering can be combined as well
sim3 %>% custom_filter(prop = paste0("p", 1:9),
                       special = "richness == 1",
                       community %in% c(7, 9))
```

---

`get_colours` *Return colour-blind friendly colours*

---

## Description

Utility function to return either a distinct colour-blind friendly colour for each variable or if a functional grouping is specified, then shades of the same colour for variables within a functional group

## Usage

```
get_colours(vars, FG = NULL)
```

## Arguments

| | |
|---|---|
| `vars` | Either a numeric value ‘n‘ to get n colours, or a character vector of values where each value will be mapped to a colour. |

| | |
|---|---|
| `FG` | A character vector describing the functional grouping to which each variable belongs. Variables within the same group will have different shades of the same colour. |

## Value

A named vector containing the hex codes of colours

## Examples

```
## Get n colours
get_colours(vars = 4)

# Get a color-map for each value specified in vars
get_colours(vars = c("p1", "p2", "p3", "p4"))

# Group values of vars using FG. Variables in the same group
# will have same shades of a colour
get_colours(vars = 4, FG = c("G1", "G1", "G2", "G2"))
```

---

`get_equi_comms` *Get all equi-proportional communities at specific levels of richness*

---

## Description

Get all equi-proportional communities at specific levels of richness

## Usage

```
get_equi_comms(
  nvars,
  richness_lvl = 1:nvars,
  variables = paste0("Var", 1:nvars),
  threshold = 1e+06
)
```

## Arguments

| | |
|---|---|
| `nvars` | Number of variables in the design |
| `richness_lvl` | The richness levels (number of non-zero compositional variables in a community) at which to return the equi-proportional communities. Defaults to each richness level from 1 up to ‘nvars‘ (both inclusive). |
| `variables` | Names for the variables. Will be used as column names for the final result. Default is "Var" followed by column number. |

| | |
|---|---|
| `threshold` | The maximum number of communities to select for each level of richness for situations when there are too many equi-proportional communities. Default value is a million.<br>Note: if threshold < ‘number of possible equi-proportional communities‘ at a given level of richness, a random selection of communities equal to the number specified in threshold would be returned. |

## Value

A dataframe consisting all or a random selection of equi-proportional communities at each level of richness

## Examples

```
## Get all equi-proportional communities for each level of richness upto 10
data10 <- get_equi_comms(10)
head(data10, 12)

## Change variable names
data4 <- get_equi_comms(4, variables = c("Lollium perenne", "Chichorum intybus",
                                         "Trifolium repens", "Trifolium pratense"))
head(data4)

## Get equi-proportional communities at specific levels of richness
## Get all equi-proportional communities of four variables at richness
## levels 1 and 3
data4_13 <- get_equi_comms(nvars = 4, richness = c(1, 3))
data4_13

## If threshold is specified and it is less than the number of possible
## equi-proportional communites at a given level of richness, then a
## random selection of communities from the total possible would be returned
## Return only 2 random equi-proportional communities at the chosen richness
## levels
data4_13_2 <- get_equi_comms(nvars = 4, richness = c(1, 3), threshold = 2)
data4_13_2

## Set threshold to a very high positive number to ensure
## random selection is never performed
data_no_random <- get_equi_comms(nvars = 15,
                                 threshold = .Machine$integer.max)
head(data_no_random)
```

## get_shades *Returns shades of colours*

## Description

Returns shades of colours

## Usage

```
get_shades(colours = c("#808080"), shades = 3)
```

## Arguments

| | |
|---|---|
| `colours` | A character vector of colours recognizable by R, to produces shades of |
| `shades` | A numeric vector giving the number of shades for each colour |

## Value

A list consisting of hex codes describing the shades of each colour

## Examples

```
## Shades for a single colour
get_shades(c("red"))

## Shades for multiple colours
get_shades(c("red", "blue" ,"#A5F8E3", "#808080"), shades = c(2, 3, 4, 5))

## A single value for shade would imply all colours get the same number of shades
get_shades(c("red", "blue" ,"#A5F8E3", "#808080"), shades = 2)
```

---

`gradient_change` *Visualise change in (predicted) response over diversity gradient*

---

## Description

A scatter-plot of the predicted response (or raw response) over a diversity gradient for specific observations is shown. The points can be overlaid with ‘`pie-glyphs`‘ to show the relative proportions of the compositional variables. The average change in any user-chosen variable over the chosen diversity gradient can also be shown using the ‘y_var‘ parameter.
This is a wrapper function specifically for statistical models fit using the `DI()` function from the `DImodels` R package and it implicitly calls `gradient_change_data` followed by `gradient_change_plot`. If your model object isn’t fit using DImodels, the associated data and plot functions can instead be called manually.

## Usage

```
gradient_change(
  model,
  data = NULL,
  gradient = c("richness", "evenness"),
  add_var = list(),
  plot = TRUE,
  average = TRUE,
  y_var = ".Pred",
  pie_data = NULL,
  pie_colours = NULL,
  pie_radius = 0.25,
  points_size = 3,
  facet_var = NULL,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| `model` | A Diversity Interactions model object fit by using the `DI()` function from the `DImodels` package. |
| `data` | A dataframe specifying communities of interest for which user wants to visualise the gradient. If left blank, the data used to fit the model will be used. |
| `gradient` | Diversity gradient to show on the X-axis, one of "richness" or "evenness". Defaults to "richness". See 'Details' for more information. |
| `add_var` | A list specifying values for additional predictor variables in the model independent of the compositional predictor variables. This could be useful for comparing the predictions across different values for a non-compositional variable. If specified as a list, it will be expanded to show a plot for each unique combination of values specified, while if specified as a data-frame, one plot would be generated for each row in the data and they will be arranged in a grid according to the value specified in 'nrow' and 'ncol'. |
| `plot` | A boolean variable indicating whether to create the plot or return the prepared data instead. The default 'TRUE' creates the plot while 'FALSE' would return the prepared data for plotting. Could be useful for if user wants to modify the data first and then call the plotting function manually. |
| `average` | A boolean value indicating whether to plot a line indicating the average change in the predicted response with respect to the variable shown on the X-axis. The average is calculated at the median value of any variables not specified. |
| `y_var` | A character string indicating the column name of the variable to be shown on the Y-axis. This could be useful for plotting raw data on the Y-axis. By default has a value of ".Pred" referring to the column containing model predictions. |
| `pie_data` | Showing all points on the graph as pie-glyphs could be resource intensive. Hence a subset of data-frame specified in 'data', can be specified here to visualise only specific points as pie-glyphs. |
| `pie_colours` | A character vector specifying the colours for the slices within the pie-glyphs. |
| `pie_radius` | A numeric value specifying the radius (in cm) for the pie-glyphs. |
| `points_size` | A numeric value specifying the size of points (when pie-glyphs not shown) shown in the plots. |
| `facet_var` | A character string or numeric index identifying the column in the data to be used for faceting the plot into multiple panels. |
| `nrow` | Number of rows in which to arrange the final plot (when 'add_var' is specified). |
| `ncol` | Number of columns in which to arrange the final plot (when 'add_var' is specified). |

## Details

Currently two diversity gradients are supported

- **Richness**: A metric describing the number of non-zero compositional variables in an observation.
- **Evenness**: A metric quantifying the relative abundances of all compositional variables in an observation. Defined as

  $$(2s/(s-1)) \sum_{i,j=1; i<j}^{s} p_i * p_j$$

  where $s$ is the total number of compositional variables and $p_i$ and $p_j$ are the proportions of the variables $i$ and $j$. See Kirwan et al., 2007 <doi:10.1890/081684.1> and Kirwan et al., 2009 <doi:10.1890/081684.1> for more information.

Here's a small example of how these metrics are calculated for a few observations. Suppose we have four compositional variables (i.e. $s = 4$) and have the following three observations

- A = (0.5, 0.5, 0, 0)
- B = (0.25, 0.25, 0.25, 0.25)
- C = (1, 0, 0, 0)

The richness values for these three observations would be as follows

- A = 2 (Since two of the four compositional variables were non-zero)
- B = 4 (Since all four compositional variables were non-zero)
- C = 1 (Since one of the four compositional variables were non-zero)

The evenness values would be calculated as follows

- A = $2 * 4/(4 - 1) * (0.5 * 0.5 + 0.5 * 0 + 0.5 * 0 + 0.5 * 0 + 0.5 * 0 + 0 * 0) = 0.67$
- B = $2*4/(4-1)*(0.25*0.25+0.25*0.25+0..25*0.25+0.25*0.25+0.25*0.25+0.25*0) = 1$
- C = $2 * 4/(4 - 1) * (1 * 0 + 1 * 0 + 1 * 0 + 0 * 0 + 0 * 0 + 0 * 0) = 0$

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if 'plot = FALSE')

## Examples

```
## Load DImodels package to fit the model
library(DImodels)
library(dplyr)

## Load data
data(sim4)
sim4 <- sim4 %>% filter(treatment == 50)

## Fit DI model
mod <- DI(prop = 3:8, DImodel = "AV", data = sim4, y = "response") %>%
         suppressWarnings()
```

```
## Create visualisation

## By default, 'richness' is the gradient and communities from the
## raw data are used to calculate average response
gradient_change(model = mod)

## Specify custom data
gradient_change(model = mod, data = sim4 %>% filter(richness <= 4))

## Create plot for all equi-proportional communities at a
## given level of richness
plot_data <- get_equi_comms(6, variables = paste0("p", 1:6))
gradient_change(model = mod, data = plot_data)

## Can also plot average response across evenness and
## change colours of the pie-slices using `pie_colours`
gradient_change(model = mod, gradient = "evenness",
                pie_colours = c("darkolivegreen1", "darkolivegreen4",
                                "orange1", "orange4",
                                "steelblue1", "steelblue4"))

## Manually specify only specific communities to be shown as pie-chart
## glyphs using `pie_data` and `facet_var` to facet the plot on
## an additional variable.
gradient_change(model = mod,
                pie_data = sim4 %>% filter(richness %in% c(1, 6)),
                facet_var = "treatment")

## Use `add_var` to add additional variables independent of the compositions
## Multiple plots will be produced and can be arranged using nrow and ncol
## Create plot arranged in 2 columns

gradient_change(model = mod,
                data = sim4[, -2],
                add_var = list("treatment" = c(50, 250)),
                pie_data = sim4[, -2] %>% filter(richness %in% c(1, 6)),
                ncol = 2)


## Specify `plot = FALSE` to not create the plot but return the prepared data
head(gradient_change(model = mod, plot = FALSE,
                     pie_data = sim4 %>% filter(richness %in% c(1, 6)),
                     facet_var = "treatment"))
```

## `gradient_change_data` *Calculate change in predicted response over diversity gradient*

## Description

Helper function for creating the data to visualise a scatter-plot of the response over a diversity gradient. The "richness" and "evenness" diversity gradients are currently supported. The average (predicted) response is calculated from all communities present at a given level of the chosen diversity gradient in ‘data‘. The output of this function can be passed to the `gradient_change_plot` function to visualise results.

## Usage

```
gradient_change_data(
  data,
  prop,
  add_var = list(),
  gradient = c("richness", "evenness"),
  prediction = TRUE,
  ...
)
```

## Arguments

| | |
|---|---|
| `data` | A data-frame consisting of variable proportions and any other necessary variables to make predictions from ‘model‘ or ‘coefficients‘. |
| `prop` | A vector identifying the column-names or indices of the columns containing the variable proportions in ‘data‘. |
| `add_var` | A list specifying values for additional predictor variables in the model independent of the compositional predictor variables. This could be useful for comparing the predictions across different values for a non-compositional variable. If specified as a list, it will be expanded to show a plot for each unique combination of values specified, while if specified as a data-frame, one plot would be generated for each row in the data and they will be arranged in a grid according to the value specified in ‘nrow‘ and ‘ncol‘. |
| `gradient` | Diversity gradient to show on the X-axis, one of "richness" or "evenness". Defaults to "richness". See ‘Details‘ for more information. |
| `prediction` | A logical value indicating whether to pass the final data to the ‘add_prediction‘ function and append the predictions to the data. Default value is TRUE, but often it would be desirable to make additional changes to the data before making any predictions, so the user can set this to FALSE and manually call the ‘add_prediction‘ function. |
| `...` | Arguments passed on to `add_prediction`<br>`model` A regression model object which will be used to make predictions for the observations in ‘data‘. Will override ‘coefficients‘ if specified.<br>`coefficients` If a regression model is not available (or can’t be fit in R), the regression coefficients from a model fit in some other language can be used to calculate predictions. However, the user would have to ensure there’s an appropriate one-to-one positional mapping between the data columns and the coefficient values. Further, they would also have to provide a variance-covariance matrix of the coefficients in the ‘vcov‘ parameter if they want |

the associated CI for the prediction or it would not be possible to calculate confidence/prediction intervals using this method.

`vcov` If regression coefficients are specified, then the variance-covariance matrix of the coefficients can be specified here to calculate the associated confidence interval around each prediction. Failure to do so would result in no confidence intervals being returned. Ensure ‘coefficients‘ and ‘vcov‘ have the same positional mapping with the data.

`coeff_cols` If ‘coefficients‘ are specified and a one-to-one positional mapping between the data-columns and coefficient vector is not present. A character string or numeric index can be specified here to reorder the data columns and match the corresponding coefficient value to the respective data column. See the "Use model coefficients for prediction" section in examples.

`conf.level` The confidence level for calculating confidence/prediction intervals. Default is 0.95.

`interval` Type of interval to calculate:

**"none" (default)** No interval to be calculated.

**"confidence"** Calculate a confidence interval.

**"prediction"** Calculate a prediction interval.

## Details

Currently two diversity gradients are supported

- **Richness**: A metric describing the number of non-zero compositional variables in an observation.
- **Evenness**: A metric quantifying the relative abundances of all compositional variables in an observation. Defined as

$$(2s/(s-1)) \sum_{i,j=1;i<j}^{s} p_i * p_j$$

where $s$ is the total number of compositional variables and $p_i$ and $p_j$ are the proportions of the variables $i$ and $j$. See Kirwan et al., 2007 <doi:10.1890/081684.1> and Kirwan et al., 2009 <doi:10.1890/081684.1> for more information.

Here’s a small example of how these metrics are calculated for a few observations. Suppose we have four compositional variables (i.e. $s = 4$) and have the following three observations

- A = (0.5, 0.5, 0, 0)
- B = (0.25, 0.25, 0.25, 0.25)
- C = (1, 0, 0, 0)

The richness values for these three observations would be as follows

- A = 2 (Since two of the four compositional variables were non-zero)
- B = 4 (Since all four compositional variables were non-zero)
- C = 1 (Since one of the four compositional variables were non-zero)

The evenness values would be calculated as follows

- A = $2 * 4/(4 - 1) * (0.5 * 0.5 + 0.5 * 0 + 0.5 * 0 + 0.5 * 0 + 0.5 * 0 + 0 * 0) = 0.67$
- B = $2*4/(4-1)*(0.25*0.25+0.25*0.25+0..25*0.25+0.25*0.25+0.25*0.25+0.25*0) = 1$
- C = $2 * 4/(4 - 1) * (1 * 0 + 1 * 0 + 1 * 0 + 0 * 0 + 0 * 0 + 0 * 0) = 0$

## Value

The data-frame with the following columns appended at the end

**.Richness** The richness (number of non-zero compositional variables) within each observation.

**.Evenness** The evenness (metric quantifying the relative abundance of each compositional variable) within each observation.

**.Gradient** An character string defining the diversity gradient used for averaging the response.

**.add_str_ID** An identifier column for grouping the cartesian product of all additional columns specified in ‘add_var‘ parameter (if ‘add_var‘ is specified).

**.Pred** The predicted response for each obsvervation.

**.Lower** The lower limit of the prediction/confidence interval for each observation.

**.Upper** The upper limit of the prediction/confidence interval for each observation.

**.Avg** The averaged value of the predicted response for each unique value of the selected diversity gradient.

## Examples

```
library(DImodels)
library(dplyr)

## Load data
data(sim2)

## Fit model
mod <- glm(response ~ 0 + (p1 + p2 + p3 + p4)^2, data = sim2)

## Create data
## By default response would be averaged on the basis of richness
head(gradient_change_data(data = sim2,
                          prop = c("p1", "p2", "p3", "p4"),
                          model = mod))

## Average response with respect to evenness
head(gradient_change_data(data = sim2,
                          prop = c("p1", "p2", "p3", "p4"),
                          model = mod,
                          gradient = "evenness"))

## Additional variables can also be added to the data by either specifying
## them directly in the `data` or by using the `add_var` argument
## Refit model
sim2$block <- as.numeric(sim2$block)
new_mod <- update(mod, ~. + block, data = sim2)
## This model has block so we can either specify block in the data
```

```
subset_data <- sim2[c(1,5,9,11), 2:6]
subset_data
head(gradient_change_data(data = subset_data,
                          prop = c("p1", "p2", "p3", "p4"),
                          model = mod,
                          gradient = "evenness"))
## Or we could add the variable using `add_var`
subset_data <- sim2[c(1,5,9,11), 3:6]
subset_data
head(gradient_change_data(data = subset_data,
                          prop = c("p1", "p2", "p3", "p4"),
                          model = new_mod,
                          gradient = "evenness",
                          add_var = list(block = c(1, 2))))
## The benefit of specifying the variable this way is we have an ID
## columns now called `.add_str_ID` which could be used to create a
## separate plot for each value of the additional variable


## Model coefficients can also be used, but then user would have
## to specify the data with all columns corresponding to each coefficient
coef_data <- sim2 %>%
               mutate(`p1:p2` = p1*p2, `p1:p3` = p1*p2, `p1:p4` = p1*p4,
                      `p2:p3` = p2*p3, `p2:p4` = p2*p4, `p3:p4` = p3*p4) %>%
               select(p1, p2, p3, p4,
                      `p1:p2`, `p1:p3`, `p1:p4`,
                      `p2:p3`, `p2:p4`, `p3:p4`) %>%
               slice(1,5,9,11)
print(coef_data)
print(mod$coefficients)
gradient_change_data(data = coef_data,
                     prop = c("p1", "p2", "p3", "p4"),
                     gradient = "evenness",
                     coefficients = mod$coefficients,
                     interval = "none")
```

## gradient_change_plot *Visualise change in (predicted) response over diversity gradient*

### Description

Helper function for plotting the average (predicted) response at each level of a diversity gradient. The output of the gradient_change_data function should be passed here to visualise a scatter-plot of the predicted response (or raw response) over a diversity gradient. The points can be overlaid with ‘pie-glyphs‘ to show the relative proportions of the compositional variables. The average change in any user-chosen variable over the chosen diversity gradient can also be shown using the ‘y_var‘ parameter.

### Usage

```
gradient_change_plot(
  data,
  prop = NULL,
  pie_data = NULL,
  pie_colours = NULL,
  pie_radius = 0.25,
  points_size = 3,
  average = TRUE,
  y_var = ".Pred",
  facet_var = NULL,
  nrow = 0,
  ncol = 0
)
```

### Arguments

| | |
|---|---|
| `data` | A data-frame which is the output of the ‘gradient_change_data‘ function, consisting of the predicted response averaged over a particular diversity gradient. |
| `prop` | A vector of column names or indices identifying the columns containing the species proportions in the data. Will be inferred from the data if it is created using the ‘`gradient_change_data`‘ function, but the user also has the flexibility of manually specifying the values. |
| `pie_data` | A subset of data-frame specified in ‘data‘, to visualise the individual data-points as pie-glyphs showing the relative proportions of the variables in the data-point. |
| `pie_colours` | A character vector specifying the colours for the slices within the pie-glyphs. |
| `pie_radius` | A numeric value specifying the radius (in cm) for the pie-glyphs. |
| `points_size` | A numeric value specifying the size of points (when pie-glyphs not shown) shown in the plots. |
| `average` | A boolean value indicating whether to plot a line indicating the average change in the predicted response with respect to the variable shown on the X-axis. The average is calculated at the median value of any variables not specified. |
| `y_var` | A character string indicating the column name of the variable to be shown on the Y-axis. This could be useful for plotting raw data on the Y-axis. By default has a value of ".Pred" referring to the column containing model predictions. |
| `facet_var` | A character string or numeric index identifying the column in the data to be used for faceting the plot into multiple panels. |
| `nrow` | Number of rows in which to arrange the final plot (when ‘add_var‘ is specified). |
| `ncol` | Number of columns in which to arrange the final plot (when ‘add_var‘ is specified). |

### Value

A ggplot object

## Examples

```
library(DImodels)
library(dplyr)

## Load data
data(sim4)
sim4 <- sim4 %>% filter(treatment %in% c(50, 150))

## Fit model
mod <- glm(response ~ 0 + (p1 + p2 + p3 + p4 + p5 + p6)^2, data = sim4)

## Create data
## By default response would be averaged on the basis of richness
plot_data <- gradient_change_data(data = sim4,
                                  prop = c("p1", "p2", "p3",
                                           "p4", "p5", "p6"),
                                  model = mod)

## Create plot
gradient_change_plot(data = plot_data)

## Average response with respect to evenness
plot_data <- gradient_change_data(data = sim4,
                                  prop = c("p1", "p2", "p3",
                                           "p4", "p5", "p6"),
                                  model = mod,
                                  gradient = "evenness")
gradient_change_plot(data = plot_data)

## Can also manually specify prop variables
## Add grouped proportions to data
plot_data <- group_prop(plot_data,
                       prop = c("p1", "p2", "p3", "p4", "p5", "p6"),
                       FG = c("Gr", "Gr", "Le", "Le", "He", "He"))
## Manually specify prop to show in pie-glyphs
gradient_change_plot(data = plot_data,
                     prop = c("Gr", "Le", "He"))

## Don't show line indicating the average change by using `average = FALSE` and
## Change colours of the pie-slices using `pie_colours`
gradient_change_plot(data = plot_data,
                     average = FALSE,
                     pie_colours = c("darkolivegreen1", "darkolivegreen4",
                                     "orange1", "orange4",
                                     "steelblue1", "steelblue4"))

## Manually specify only specific communities to be shown as pie-chart
## glyphs using `pie_data`.
## Note: It is important for the data specified in
## `pie_data` to have the .Pred and .Gradient columns.
## So the best use case for this parameter is to accept
## a subset of the data specified in `data`.#'
## Also use `facet_var` to facet the plot on an additional variable
gradient_change_plot(data = plot_data,
                     pie_data = plot_data %>% filter(.Richness %in% c(1, 6)),
                     facet_var = "treatment")

## If `add_var` was used during the data preparation step then
## multiple plots will be produced and can be arranged using nrow and ncol

new_mod <- update(mod, ~. + treatment, data = sim4)
plot_data <- gradient_change_data(data = sim4[c(seq(1, 18, 3), 19:47), -2],
                                  prop = c("p1", "p2", "p3",
                                           "p4", "p5", "p6"),
                                  model = new_mod,
                                  add_var = list("treatment" = c(50, 250)))
## Create plot arranged in 2 columns
gradient_change_plot(data = plot_data,
                     pie_data = plot_data %>% filter(.Richness %in% c(1, 6)),
                     ncol = 2)

## Create plot for raw data instead of predictions
## Create the data for plotting by specifying `prediction = FALSE`
plot_data <- gradient_change_data(data = sim4[sim4$treatment == 50, ],
                                  prop = c("p1", "p2", "p3",
                                           "p4", "p5", "p6"),
                                  prediction = FALSE)
## This data will not have any predictions
head(plot_data)
## Call the plotting function by specifying the variable you we wish to
## plot on the Y-axis by using the argument `y_var`
## Since this data wasn't created using `gradient_change_data`
## `prop` should be manually specified
gradient_change_plot(data = plot_data, y_var = "response",
                     prop = c("p1", "p2", "p3",
                              "p4", "p5", "p6"))
```

## `grouped_ternary` *Conditional ternary diagrams at functional group level*

### Description

Grouped ternary diagrams are created by combining the proportions of the compositional variables into groups and visualising these groups on a 2-d ternary diagram. These are very useful when we have multiple compositional variables that can be grouped together by some hierarchical grouping structure. For example, grouping species in a ecosystem based on the functions they perform, or grouping political parties based on their national alliances. Grouping variables this way allows us to reduce the dimensionality of the compositional data and visualise it. This is akin to looking at a 2-d slice of the high dimensional simplex. The relative proportions of each variable within a group can be adjusted to look at different slices of the simplex. Looking at multiple such slices would enable us to create an approximation of how the response varies across the original n-dimensional simplex. This is a wrapper function specifically for statistical models fit using the DI() function from the DImodels R package and would implicitly call grouped_ternary_data followed by grouped_ternary_plot. If your model object isn't fit using DImodels, consider calling these functions manually.

## Usage

```
grouped_ternary(
  model,
  FG,
  values = NULL,
  tern_vars = NULL,
  conditional = NULL,
  add_var = list(),
  resolution = 3,
  plot = TRUE,
  nlevels = 7,
  colours = NULL,
  lower_lim = NULL,
  upper_lim = NULL,
  contour_text = TRUE,
  show_axis_labels = TRUE,
  show_axis_guides = FALSE,
  axis_label_size = 4,
  vertex_label_size = 5,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| model | A Diversity Interactions model object fit by using the DI() function from the DImodels package. |
| FG | A character vector specifying the groupings of the variables specified in 'prop'. |
| values | A numeric vector specifying the proportional split of the variables within a group. The default is to split the group proportion equally between each variable in the group. |
| tern_vars | A character vector giving the names of the three variables to be shown in the ternary diagram. |
| conditional | A data-frame describing the names of the compositional variables and their respective values at which to slice the simplex space. The format should be, for example, as follows:<br>`data.frame("p1" = c(0, 0.5), "p2" = c(0.2, 0.1))`<br>One figure would be created for each row in 'conditional' with the respective values of all specified variables. Any compositional variables not specified in 'conditional' will be assumed to be 0. |
| add_var | A list or data-frame specifying values for additional variables in the model other than the proportions (i.e. not part of the simplex design). This could be useful |

| | |
|---|---|
| | for comparing the predictions across different values for a non-compositional variable. If specified as a list, it will be expanded to show a plot for each unique combination of values specified, while if specified as a data-frame, one plot would be generated for each row in the data. |
| `resolution` | A number between 1 and 10 describing the resolution of the resultant graph. A high value would result in a higher definition figure but at the cost of being computationally expensive. |
| `plot` | A boolean variable indicating whether to create the plot or return the prepared data instead. The default `TRUE` creates the plot while `FALSE` would return the prepared data for plotting. Could be useful if user wants to modify the data first and then create the plot. |
| `nlevels` | The number of levels to show on the contour map. |
| `colours` | A character vector or function specifying the colours for the contour map or points. The number of colours should be same as ‘nlevels‘ if (‘show = "contours"‘). The default colours scheme is the `terrain.colors()` for continuous variables and an extended version of the Okabe-Ito colour scale for categorical variables. |
| `lower_lim` | A number to set a custom lower limit for the contour (if ‘show = "contours"‘). The default is minimum of the prediction. |
| `upper_lim` | A number to set a custom upper limit for the contour (if ‘show = "contours"‘). The default is maximum of the prediction. |
| `contour_text` | A boolean value indicating whether to include labels on the contour lines showing their values (if ‘show = "contours"‘). The default is `TRUE`. |
| `show_axis_labels` | A boolean value indicating whether to show axis labels along the edges of the ternary. The default is `TRUE`. |
| `show_axis_guides` | A boolean value indicating whether to show axis guides within the interior of the ternary. The default is `FALSE`. |
| `axis_label_size` | A numeric value to adjust the size of the axis labels in the ternary plot. The default size is 4. |
| `vertex_label_size` | A numeric value to adjust the size of the vertex labels in the ternary plot. The default size is 5. |
| `nrow` | Number of rows in which to arrange the final plot (when ‘add_var‘ is specified). |
| `ncol` | Number of columns in which to arrange the final plot (when ‘add_var‘ is specified). |

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if ‘plot = FALSE‘)

## Examples

```
library(DImodels)
library(dplyr)
data(sim3)
m1 <- DI(y = "response", prop = paste0("p", 1:9),
         DImodel = "AV", data = sim3) %>%
         suppressWarnings()

## We have nine (p1 to p9) variables here and using `conditional_ternary`
## to visualise the simplex space won't be very helpful as there are too
## variables to condition on
## Instead we group the nine-variables into three groups called "G", "L" and "H"
grouped_ternary(model = m1, FG = c("G","G","G","G","G","L","L","H","H"),
                resolution = 1)
## By default the variables within a group take up an equal share of the
## group proportion. So for example, each point along the above ternary
## would have a 50:50 split of the variables in the group "L" or "H", thus
## the vertex where "L" is 1, would mean that p6 and p7 are 0.5 each,
## similarly, the vertex "H" is made up of 0.5 of p8 and p9 while the "G"
## vertex is comprised of 0.2 of each of p1, p2, p3, p4, and p5. The concepts
## also extend to points along the edges and interior of the ternary.

## We can also manually specify the split of the species within a group
## This would mean we are looking at a different slice of the simplex
## For example this would mean the groups "L" group is made up of 100% of
## p7 and doesn't contain any p6, while "H" group contains 30% of p8 and
## 70% of p9, while "G" group still contains 20% of each p1 to p5.
grouped_ternary(m1, FG = c("G","G","G","G","G","L","L","H","H"),
                resolution = 1,
                values = c(0.2, 0.2, 0.2, 0.2, 0.2,
                           0, 1,
                           0.3, 0.7))

## If here are more than three groups then, we could condition some groups
## to have a fixed value while three groups are manipulated within a ternary
## The group "G" is now split into two groups "G1" and "G2"
## We can create conditional ternary diagram at the grouped level as well
## Notice the values going in `tern_vars` and `conditional` are names
## of the groups and not the original compositional variables
grouped_ternary(m1, FG = c("G1","G1","G2","G2","G2","L","L","H","H"),
                resolution = 1,
                tern_vars = c("G1", "L", "H"),
                conditional = data.frame("G2" = c(0, 0.25, 0.5)))

## Specify `plot = FALSE` to not create the plot but return the prepared data
head(grouped_ternary(m1, FG = c("G1","G1","G2","G2","G2","L","L","H","H"),
                     resolution = 1, plot = FALSE,
                     tern_vars = c("G1", "L", "H"),
                     conditional = data.frame("G2" = c(0, 0.25, 0.5))))

## All other functionality from \code{\link{condtional_ternary_plot}} is
## available in this function too.
```

## `grouped_ternary_data` *Grouped ternary diagrams*

### Description

The helper function for preparing the underlying data for creating grouped ternary diagrams where the proportions of the compositional variables are combined into groups and visualised on a ternary diagram. These are very useful when we have multiple compositional variables that can be grouped together by some hierarchical grouping structure. For example, grouping species in a ecosystem based on the functions they perform, or grouping political parties based on their national alliances. Grouping variables this way allows us to reduce the dimensionality of the compositional data and visualise it. This is akin to looking at a 2-d slice of the high dimensional simplex. The relative proportions of each variable within a group can be adjust to look at different slices of the simplex. Looking at multiple such slices would enable us to create an approximation of how the response varies across the original n-dimensional simplex. The output of this function can be passed to the `grouped_ternary_plot` function to plot the results.

### Usage

```
grouped_ternary_data(
  prop,
  FG,
  values = NULL,
  tern_vars = NULL,
  conditional = NULL,
  add_var = list(),
  resolution = 3,
  prediction = TRUE,
  ...
)
```

### Arguments

| | |
|---|---|
| `prop` | A character vector indicating the model coefficients corresponding to variable proportions. These variables should be compositional in nature (i.e., proportions should sum to 1). |
| `FG` | A character vector specifying the groupings of the variables specified in 'prop'. |
| `values` | A numeric vector specifying the proportional split of the variables within a group. The default is to split the group proportion equally between each variable in the group. |
| `tern_vars` | A character vector giving the names of the three variables to be shown in the ternary diagram. |
| `conditional` | A data-frame describing the names of the compositional variables and their respective values at which to slice the simplex space. The format should be, for example, as follows:<br>`data.frame("p1" = c(0, 0.5), "p2" = c(0.2, 0.1))` |

One figure would be created for each row in ‘conditional‘ with the respective values of all specified variables. Any compositional variables not specified in ‘conditional‘ will be assumed to be 0.

`add_var` A list or data-frame specifying values for additional variables in the model other than the proportions (i.e. not part of the simplex design). This could be useful for comparing the predictions across different values for a non-compositional variable. If specified as a list, it will be expanded to show a plot for each unique combination of values specified, while if specified as a data-frame, one plot would be generated for each row in the data.

`resolution` A number between 1 and 10 describing the resolution of the resultant graph. A high value would result in a higher definition figure but at the cost of being computationally expensive.

`prediction` A logical value indicating whether to pass the final data to the ‘add_prediction‘ function and append the predictions to the data. Default value is `TRUE`, but often it would be desirable to make additional changes to the data before making any predictions, so the user can set this to `FALSE` and manually call the ‘add_prediction‘ function.

`...` Arguments passed on to `add_prediction`

- `model` A regression model object which will be used to make predictions for the observations in ‘data‘. Will override ‘coefficients‘ if specified.
- `coefficients` If a regression model is not available (or can’t be fit in R), the regression coefficients from a model fit in some other language can be used to calculate predictions. However, the user would have to ensure there’s an appropriate one-to-one positional mapping between the data columns and the coefficient values. Further, they would also have to provide a variance-covariance matrix of the coefficients in the ‘vcov‘ parameter if they want the associated CI for the prediction or it would not be possible to calculate confidence/prediction intervals using this method.
- `vcov` If regression coefficients are specified, then the variance-covariance matrix of the coefficients can be specified here to calculate the associated confidence interval around each prediction. Failure to do so would result in no confidence intervals being returned. Ensure ‘coefficients‘ and ‘vcov‘ have the same positional mapping with the data.
- `coeff_cols` If ‘coefficients‘ are specified and a one-to-one positional mapping between the data-columns and coefficient vector is not present. A character string or numeric index can be specified here to reorder the data columns and match the corresponding coefficient value to the respective data column. See the "Use model coefficients for prediction" section in examples.
- `conf.level` The confidence level for calculating confidence/prediction intervals. Default is 0.95.
- `interval` Type of interval to calculate:
  - **"none" (default)** No interval to be calculated.
  - **"confidence"** Calculate a confidence interval.
  - **"prediction"** Calculate a prediction interval.

## Value

A data-frame containing compositional columns with names specified in ‘FG‘ and ‘prop‘ parameters along with any additional columns specified in ‘add_var‘ parameter and the following columns appended at the end.

**.x** The x-projection of the points within the ternary.

**.y** The y-projection of the points within the ternary.

**.add_str_ID** An identifier column for grouping the cartesian product of all additional columns specified in ‘add_var‘ parameter (if ‘add_var‘ is specified).

**.Sp** An identifier column specifying the functional group along which the high dimensional simplex is sliced (if there are more than 3 groups).

**.Value** The value (between 0 and 1) along the direction of functional group in ‘.Sp‘ at which the high dimensional simplex is sliced.

**.Facet** An identifier column formed by combining ‘.Sp‘ and ‘.value‘ to group observations within a specific slice of the high dimensional simplex.

**.Pred** The predicted response for each observation. (if ‘prediction‘ is `TRUE`)

**.Lower** The lower limit of the prediction/confidence interval for each observation.

**.Upper** The upper limit of the prediction/confidence interval for each observation.

## Examples

```
library(DImodels)

## Load data
data(sim3)

## Fit model
mod <- glm(response ~ 0 + (p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9)^2,
           data = sim3)

## Create data
## We have nine (p1 to p9) variables here and using \code{\link{conditional_ternary}}
## to visualise the simplex space won't be very helpful as there are too
## variables to condition on
## Instead we group the nine-variables into three groups called "G", "L" and "H"
head(grouped_ternary_data(model = mod,
                          prop = paste0("p", 1:9),
                          FG = c("G","G","G","G","G","L","L","H","H"),
                          resolution = 1))

## By default the variables within a group take up an equal share of the
## group proportion. So for example, each point along the above ternary
## would have a 50:50 split of the variables in the group "L" or "H", thus
## the vertex where "L" is 1, would mean that p6 and p7 are 0.5 each,
## similarly, the vertex "H" is made up of 0.5 of p8 and p9 while the "G"
## vertex is comprised of 0.2 of each of p1, p2, p3, p4, and p5. The concepts
## also extend to points along the edges and interior of the ternary.
## Change the proportional split of species within an FG by using `values`
## `values` takes a numeric vector where the position of each element
## describes the proportion of the corresponding species within the
## corresponding FG
## For examples this vector describes, 2-% each of p1, p2, p3, p4 and p5,
## in G, 0% and 100% of p6 and p7, respectively in G2 and 30% and 70% of
## p8 and p9, respectively in G3.
vals <- c(0.2, 0.2, 0.2, 0.2, 0.2,
          0, 1,
          0.3, 0.7)
head(grouped_ternary_data(prop = paste0("p", 1:9),
                          FG = c("G","G","G","G","G","L","L","H","H"),
                          values = vals,
                          resolution = 1,
                          model = mod))

## Can also add any additional experimental structures
## Notice .add_str_ID in the data
head(grouped_ternary_data(prop = paste0("p", 1:9),
                          FG = c("G","G","G","G","G","L","L","H","H"),
                          add_var = list("treatment" = c("50", "150")),
                          values = vals,
                          model = mod,
                          resolution = 1))

## It could be desirable to take the output of this function and add
## additional variables to the data before making predictions
## Use `prediction = FALSE` to get data without any predictions
grouped_data <- grouped_ternary_data(prop = paste0("p", 1:9),
                                     FG = c("G","G","G","G","G","L","L","H","H"),
                                     values = vals,
                                     resolution = 1,
                                     prediction = FALSE)
grouped_data$treatment <- 250
# Add predictions
head(add_prediction(data = grouped_data, model = mod))
```

## grouped_ternary_plot *Conditional ternary diagrams at functional group level*

### Description

The helper function for plotting grouped ternary diagrams. The output of the ‘grouped_ternary_data‘ with the compositional variables combined into groups should be passed here to be visualised on a 2-d ternary diagram. These are very useful when we have multiple compositional variables that can be grouped together by some hierarchical grouping structure. For example, grouping species in a ecosystem based on the functions they perform, or grouping political parties based on their national alliances.

## Usage

```
grouped_ternary_plot(
  data,
  col_var = ".Pred",
  nlevels = 7,
  colours = NULL,
  lower_lim = NULL,
  upper_lim = NULL,
  tern_labels = colnames(data)[1:3],
  contour_text = TRUE,
  show_axis_labels = TRUE,
  show_axis_guides = FALSE,
  axis_label_size = 4,
  vertex_label_size = 5,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| `data` | A data-frame which is the output of the ‘conditional_ternary_data‘ function. |
| `col_var` | The column name containing the variable to be used for colouring the contours or points. The default is ".Pred". |
| `nlevels` | The number of levels to show on the contour map. |
| `colours` | A character vector or function specifying the colours for the contour map or points. The number of colours should be same as ‘nlevels‘ if (‘show = "contours"‘).<br>The default colours scheme is the `terrain.colors()` for continuous variables and an extended version of the Okabe-Ito colour scale for categorical variables. |
| `lower_lim` | A number to set a custom lower limit for the contour (if ‘show = "contours"‘). The default is minimum of the prediction. |
| `upper_lim` | A number to set a custom upper limit for the contour (if ‘show = "contours"‘). The default is maximum of the prediction. |
| `tern_labels` | A character vector containing the labels of the vertices of the ternary. The default is the column names of the first three columns of the data, with the first column corresponding to the top vertex, second column corresponding to the left vertex and the third column corresponding to the right vertex of the ternary. |
| `contour_text` | A boolean value indicating whether to include labels on the contour lines showing their values (if ‘show = "contours"‘). The default is `TRUE`. |
| `show_axis_labels` | A boolean value indicating whether to show axis labels along the edges of the ternary. The default is `TRUE`. |
| `show_axis_guides` | A boolean value indicating whether to show axis guides within the interior of the ternary. The default is `FALSE`. |

`axis_label_size`
: A numeric value to adjust the size of the axis labels in the ternary plot. The default size is 4.

`vertex_label_size`
: A numeric value to adjust the size of the vertex labels in the ternary plot. The default size is 5.

`nrow`
: Number of rows in which to arrange the final plot (when ‘add_var‘ is specified).

`ncol`
: Number of columns in which to arrange the final plot (when ‘add_var‘ is specified).

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if ‘plot = FALSE‘)

## Examples

```
library(DImodels)

## Load data
data(sim3)

## Fit model
mod <- glm(response ~ 0 + (p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9)^2,
           data = sim3)

## Create data
## We have nine (p1 to p9) variables here and using conditional_ternary
## to visualise the simplex space won't be very helpful as there are too
## variables to condition on
## Instead we group the nine-variables into three groups called "G", "L" and "H"
plot_data <- grouped_ternary_data(model = mod,
                                  prop = paste0("p", 1:9),
                                  FG = c("G","G","G","G","G","L","L","H","H"),
                                  resolution = 1)
grouped_ternary_plot(plot_data)

## By default the variables within a group take up an equal share of the
## group proportion. So for example, each point along the above ternary
## would have a 50:50 split of the variables in the group "L" or "H", thus
## the vertex where "L" is 1, would mean that p6 and p7 are 0.5 each,
## similarly, the vertex "H" is made up of 0.5 of p8 and p9 while the "G"
## vertex is comprised of 0.2 of each of p1, p2, p3, p4, and p5. The concepts
## also extend to points along the edges and interior of the ternary.

## Change the proportional split of species within an FG by using `values`
## `values` takes a numeric vector where the position of each element
## describes the proportion of the corresponding species within the
## corresponding FG
## For examples this vector describes, 2-% each of p1, p2, p3, p4 and p5,
## in G, 0% and 100% of p6 and p7, respectively in G2 and 30% and 70% of
## p8 and p9, respectively in G3.
vals <- c(0.2, 0.2, 0.2, 0.2, 0.2,
          0, 1,
          0.3, 0.7)
plot_data <- grouped_ternary_data(prop = paste0("p", 1:9),
                                  FG = c("G","G","G","G","G","L","L","H","H"),
                                  values = vals,
                                  resolution = 1,
                                  model = mod)
## Change number of contours and colour scheme
grouped_ternary_plot(plot_data,
                     nlevels = 8,
                     colours = hcl.colors(8))

## Can also add any additional experimental structures
## Notice .add_str_ID in the data
plot_data <- grouped_ternary_data(prop = paste0("p", 1:9),
                                  FG = c("G","G","G","G","G","L","L","H","H"),
                                  add_var = list("treatment" = c("50", "150")),
                                  values = vals,
                                  model = mod,
                                  resolution = 1)
grouped_ternary_plot(data = plot_data)
```

## group_prop *Combine variable proportions into groups*

### Description

Combine variable proportions into groups

### Usage

```
group_prop(data, prop, FG = NULL)
```

### Arguments

| | |
|---|---|
| data | A data frame containing the compositional variables which need to be grouped. |
| prop | A character/numeric vector indicating the columns containing the compositional variables in ‘data‘. |
| FG | A character vector of same length as ‘prop‘ specifying the group each variable belongs to. |

### Value

A data-frame with additional columns appended to the end that contain the grouped variable proportions.

### Examples

```
library(DImodels)

data(sim1)

head(group_prop(data = sim1, prop = 3:6,
                FG = c("Gr1", "Gr1", "Gr1", "Gr2")))

head(group_prop(data = sim1, prop = 3:6,
                FG = c("Group1", "Group2", "Group1", "Group3")))

## Data is returned as is, if no groups are specified in FG
head(group_prop(data = sim1, prop = 3:6))
```

---

## `model_diagnostics` *Regression diagnostics plots with pie-glyphs*

---

### Description

This function returns regression diagnostics plots for a model with points replaced by pie-glyphs making it easier to track various data points in the plots. This could be useful in models with compositional predictors to quickly identify any observations with unusual residuals, hat values, etc.

### Usage

```
model_diagnostics(
  model,
  which = c(1, 2, 3, 5),
  prop = NULL,
  FG = NULL,
  npoints = 3,
  cook_levels = c(0.5, 1),
  pie_radius = 0.2,
  pie_colours = NULL,
  only_extremes = FALSE,
  label_size = 4,
  points_size = 3,
  plot = TRUE,
  nrow = 0,
  ncol = 0
)
```

### Arguments

| | |
|---|---|
| `model` | A statistical regression model object fit using `lm`, `glm`, `nlme` functions, etc. |

| | |
|---|---|
| `which` | A subset of the numbers 1 to 6, by default 1, 2, 3, and 5, referring to 1 - "Residuals vs Fitted", aka "Tukey-Anscombe" plot 2 - "Normal Q-Q" plot, an enhanced qqnorm(resid(.)) 3 - "Scale-Location" 4 - "Cook's distance" 5 - "Residuals vs Leverage" 6 - "Cook's dist vs Lev./(1-Lev.)" *Note: If the specified model object does not inherit the* `lm` *class, it might not be possible to create all diagnostics plots. In these cases, the user will be notified about any plots which can't be created.* |
| `prop` | A character vector giving names of columns containing proportions to show in the pie-glyphs. If not specified, black points (geom_point) will be shown for each observation in the model. Note: `\code{prop}` can be left blank and will be interpreted if model is a `Diversity-Interactions` `(DI)` model object fit using the `DI()` function from the `DImodels` package. |
| `FG` | A character vector of same length as `prop` specifying the group each variable belongs to. |
| `npoints` | Number of points to be labelled in each plot, starting with the most extreme (those points with the highest absolute residuals or hat values). |
| `cook_levels` | A numeric vector specifying levels of Cook's distance at which to draw contours. |
| `pie_radius` | A numeric value specifying the radius (in cm) for the pie-glyphs. |
| `pie_colours` | A character vector specifying the colours for the slices within the pie-glyphs. |
| `only_extremes` | A logical value indicating whether to show pie-glyphs only for extreme observations (points with the highest absolute residuals or hat values). |
| `label_size` | A numeric value specifying the size of the labels identifying extreme observations. |
| `points_size` | A numeric value specifying the size of points (when pie-glyphs not shown) shown in the plots. |
| `plot` | A boolean variable indicating whether to create the plot or return the prepared data instead. The default `TRUE` creates the plot while `FALSE` would return the prepared data for plotting. Could be useful if user wants to modify the data first and then create the plot. |
| `nrow` | Number of rows in which to arrange the final plot. |
| `ncol` | Number of columns in which to arrange the final plot. |

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if `plot = FALSE`).

## Examples

```
library(DImodels)

## Load data
data(sim1)
```

```
## Fit model
mod1 <- lm(response ~ 0 + (p1 + p2 + p3 + p4)^2, data = sim1)

## Get diagnostics plot
## Recommend to store plot in a variable, to access individual plots later
diagnostics <- model_diagnostics(mod1, prop = c("p1", "p2", "p3", "p4"))
print(diagnostics)

## Access individual plots
print(diagnostics[[1]])
print(diagnostics[[4]])

## Change plot arrangement

model_diagnostics(mod1, prop = c("p1", "p2", "p3", "p4"),
                  which = c(1, 3), nrow = 2, ncol = 1)


## Show only extreme points as pie-glyphs to avoid overplotting
model_diagnostics(mod1, prop = c("p1", "p2", "p3", "p4"),
                  which = 2, npoints = 5, only_extremes = TRUE)

## If model is a DImodels object, the don't need to specify prop
DI_mod <- DI(y = "response", prop = c("p1", "p2", "p3", "p4"),
             DImodel = "FULL", data = sim1)
model_diagnostics(DI_mod, which = 1)

## Specify `plot = FALSE` to not create the plot but return the prepared data
head(model_diagnostics(DI_mod, which = 1, plot  = FALSE))
```

## model_diagnostics_data

*Data preparation for regression diagnostics plots with pie-glyphs*

### Description

This function prepares the data-frame with necessary attributes for creating regression diagnostics plots for a model with compositional predictors where points are replaced by pie-glyphs making it easier to track various data points in the plots. The output data-frame can be passed to `model_diagnostics_plot` to create the visualisation.

### Usage

```
model_diagnostics_data(model, prop = NULL)
```

### Arguments

| | |
|---|---|
| `model` | A statistical regression model object fit using `lm`, `glm`, `nlme` functions, etc. |
| `prop` | A character vector giving names of the compositional predictors in the model. If this is not specified then plots prepared using the data would not contain pie-glyphs. |

### Value

The original data used for fitting the model with the response and all model predictors along with the following additional columns

**.hat** Diagonal of the hat matrix.

**.sigma** Estimate of residual standard deviation when corresponding observation is dropped from model.

**.cooksd** The cook's distance (`cooks.distance()`) for each observation.

**.fitted** Fitted values of model.

**.resid** The residuals for the observations.

**.stdresid** The standardised (Pearson) residuals for the observations.

**Obs** A unique identifier for each observation.

**Label** The labels to be displayed besides the observations in the plot.

**.qq** The quantile values for the standardised residuals generated using `qqnorm()`.

**weights** The weights for each observation in the model (useful in the context of weighted regression).

### Examples

```
library(DImodels)

## Load data
data(sim1)

## Fit model
mod1 <- lm(response ~ 0 + (p1 + p2 + p3 + p4)^2, data = sim1)

## Get data for diagnostics plot
diagnostics_data <- model_diagnostics_data(mod1,
                                           prop = c("p1", "p2", "p3", "p4"))
print(head(diagnostics_data))

## The compositional predictors in the data are added as attributes to the data
attr(diagnostics_data, "prop")
```

---

## model_diagnostics_plot

*Regression diagnostics plots with pie-glyphs*

---

### Description

This function accepts the output of the `model_diagnostics_data` function and returns regression diagnostics plots for a model with points replaced by pie-glyphs making it easier to track various data points in the plots. This could be useful in models with compositional predictors to quickly identify any observations with unusual residuals, hat values, etc.

## Usage

```
model_diagnostics_plot(
  data,
  which = c(1, 2, 3, 5),
  prop = NULL,
  FG = NULL,
  npoints = 3,
  cook_levels = c(0.5, 1),
  pie_radius = 0.2,
  pie_colours = NULL,
  only_extremes = FALSE,
  label_size = 4,
  points_size = 3,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| `data` | A data-frame containing the model-fit statistics for a regression model. This data could be prepared using the ‘model_diagnostics_data‘ function, or be created manually by the user with the necessary information stored into the respective columns. |
| `which` | A subset of the numbers 1 to 6, by default 1, 2, 3, and 5, referring to<br>1 - "Residuals vs Fitted", aka "Tukey-Anscombe" plot<br>2 - "Normal Q-Q" plot, an enhanced qqnorm(resid(.))<br>3 - "Scale-Location"<br>4 - "Cook’s distance"<br>5 - "Residuals vs Leverage"<br>6 - "Cook’s dist vs Lev./(1-Lev.)"<br>*Note: If the specified model object does not inherit the* `lm` *class, it might not be possible to create all diagnostics plots. In these cases, the user will be notified about any plots which can’t be created.* |
| `prop` | A character vector giving names of columns containing proportions to show in the pie-glyphs. If not specified, black points (geom_point) will be shown for each observation in the model. Note: `\code{prop}` can be left blank and will be interpreted if model is a `Diversity-Interactions` (`DI`) model object fit using the `DI()` function from the `DImodels` package. |
| `FG` | A character vector of same length as `prop` specifying the group each variable belongs to. |
| `npoints` | Number of points to be labelled in each plot, starting with the most extreme (those points with the highest absolute residuals or hat values). |
| `cook_levels` | A numeric vector specifying levels of Cook’s distance at which to draw contours. |
| `pie_radius` | A numeric value specifying the radius (in cm) for the pie-glyphs. |
| `pie_colours` | A character vector specifying the colours for the slices within the pie-glyphs. |

| | |
|---|---|
| `only_extremes` | A logical value indicating whether to show pie-glyphs only for extreme observations (points with the highest absolute residuals or hat values). |
| `label_size` | A numeric value specifying the size of the labels identifying extreme observations. |
| `points_size` | A numeric value specifying the size of points (when pie-glyphs not shown) shown in the plots. |
| `nrow` | Number of rows in which to arrange the final plot. |
| `ncol` | Number of columns in which to arrange the final plot. |

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if ‘plot = FALSE‘).

## Examples

```
library(DImodels)

## Load data
data(sim1)

## Fit model
mod1 <- lm(response ~ 0 + (p1 + p2 + p3 + p4)^2, data = sim1)

## Get data for diagnostics plot
diagnostics_data <- model_diagnostics_data(mod1,
                                           prop = c("p1", "p2", "p3", "p4"))

## Create diagnostics plots
diagnostics <- model_diagnostics_plot(diagnostics_data)
print(diagnostics)

## Access individual plots
print(diagnostics[[1]])
print(diagnostics[[4]])

## Change plot arrangement
model_diagnostics_plot(diagnostics_data, which = c(1, 3),
                       nrow = 2, ncol = 1)

## Show only extreme points as pie-glyphs to avoid overplotting
model_diagnostics_plot(diagnostics_data, which = 2,
                       npoints = 3, only_extremes = TRUE)

## Change size and colours of pie_glyphs
model_diagnostics_plot(diagnostics_data, which = 2,
                       npoints = 3, only_extremes = TRUE,
                       pie_radius = 0.3,
                       pie_colours = c("gold", "steelblue", "tomato", "darkgrey"))
```

`model_selection` *Visualising model selection*

## Description

This function helps to visualise model selection by showing a visual comparison between the information criteria for different models. It is also possible to visualise a breakup of the information criteria into deviance (goodness-of-fit) and penalty terms for each model. This could aid in understanding why a parsimonious model could be preferable over a more complex model.

## Usage

```
model_selection(
  models,
  metric = c("AIC", "BIC", "AICc", "BICc", "deviance"),
  sort = FALSE,
  breakup = FALSE,
  plot = TRUE,
  model_names = names(models)
)
```

## Arguments

| | |
|---|---|
| `models` | List of statistical regression model objects. |
| `metric` | Metric used for comparisons between models. Takes values from c("AIC", "BIC", "AICc", "BICc", "logLik"). Can choose a single or multiple metrics for comparing the different models. |
| `sort` | A boolean value indicating whether to sort the model from highest to lowest value of chosen metric. |
| `breakup` | A boolean value indicating whether to breakup the metric value into deviance (defined as -2*loglikelihood) and penalty components. Will work only if a single metric out of "AIC", "AICc", "BIC", or "BICc" is chosen to plot. |
| `plot` | A boolean variable indicating whether to create the plot or return the prepared data instead. The default 'TRUE' creates the plot while 'FALSE' would return the prepared data for plotting. Could be useful if user wants to modify the data first and then call the plotting |
| `model_names` | A character string describing the names to display on X-axis for each model in order they appear in the models parameter. |

## Value

A ggplot object or data-frame (if 'plot == FALSE')

## Examples

```
library(DImodels)

## Load data
data(sim2)

## Fit different DI models
mod_AV <- DI(prop = 3:6, DImodel = "AV", data = sim2, y = "response")
mod_FULL <- DI(prop = 3:6, DImodel = "FULL", data = sim2, y = "response")
mod_FG <- DI(prop = 3:6, DImodel = "FG", FG = c("G","G","L","L"),
             data = sim2, y = "response")
mod_AV_theta <- DI(prop = 3:6, DImodel = "AV", data = sim2,
                   y = "response", estimate_theta = TRUE)
mod_FULL_theta <- DI(prop = 3:6, DImodel = "FULL", data = sim2,
                     y = "response", estimate_theta = TRUE)
mod_FG_theta <- DI(prop = 3:6, DImodel = "FG", FG = c("G","G","L","L"),
                   data = sim2, y = "response", estimate_theta = TRUE)

models_list <- list("AV model" = mod_AV, "Full model" = mod_FULL,
                    "FG model" = mod_FG, "AV model_t" = mod_AV_theta,
                    "Full model_t" = mod_FULL_theta,
                    "FG model_t" = mod_FG_theta)

## Specific metric
model_selection(models = models_list,
                metric = c("AIC"))

## Multiple metrics can be plotted together as well
model_selection(models = models_list,
                metric = c("AIC", "BIC"))

## If single metric is specified then breakup of metric
## between goodness of fit and penalty can also be visualised
model_selection(models = models_list,
                metric = c("AICc"),
                breakup = TRUE)

## Sort models
model_selection(models = models_list,
                metric = c("AICc"),
                breakup = TRUE, sort = TRUE)

## If multiple metrics are specified then sorting
## will be done on first metric specified in list (AIC in this case)
model_selection(models = models_list,
                metric = c("AIC", "BIC", "AICc", "BICc"), sort = TRUE)

## If the list specified in models is not named then
## By default the labels on the X-axis for the models will be
## created by assigning a unique ID to each model sequentially
## in the order they appear in the models object
names(models_list) <- NULL
```

```
model_selection(models = models_list,
                metric = c("AIC", "BIC", "AICc"), sort = TRUE)

## When possible the variables names of objects containing the
## individual models would be used as axis labels
model_selection(models = list(mod_AV, mod_FULL, mod_FG,
                              mod_AV_theta, mod_FULL_theta, mod_FG_theta),
                metric = c("AIC", "BIC"), sort = TRUE)

## If neither of these two situations are desirable custom labels
## for each model can be specified using the model_names parameter
model_selection(models = list(mod_AV, mod_FULL, mod_FG,
                              mod_AV_theta, mod_FULL_theta, mod_FG_theta),
                metric = c("AIC", "BIC"), sort = TRUE,
                model_names = c("AV model", "Full model", "FG model",
                                "AV theta", "Full theta", "FG theta"))

## Specify `plot = FALSE` to not create the plot but return the prepared data
head(model_selection(models = list(mod_AV, mod_FULL, mod_FG,
                                   mod_AV_theta, mod_FULL_theta, mod_FG_theta),
                     metric = c("AIC", "BIC"), sort = TRUE, plot = FALSE,
                     model_names = c("AV model", "Full model", "FG model",
                                     "AV theta", "Full theta", "FG theta")))
```

---

## model_selection_data *Prepare data for visualising model selection*

---

### Description

The data preparation function for visualising model selection. The output of this function can be passed to the model_selection_plot function for showing a visual comparison between the information criteria for different models. It is also possible to visualise a breakup of the information criteria into deviance (goodness-of-fit) and penalty terms for each model.

### Usage

```
model_selection_data(
  models,
  metric = c("AIC", "BIC", "AICc", "BICc", "deviance"),
  sort = FALSE,
  breakup = FALSE,
  model_names = names(models)
)
```

### Arguments

| | |
|---|---|
| models | List of statistical regression model objects. |

| | |
|---|---|
| `metric` | Metric used for comparisons between models. Takes values from c("AIC", "BIC", "AICc", "BICc", "logLik"). Can choose a single or multiple metrics for comparing the different models. |
| `sort` | A boolean value indicating whether to sort the model from highest to lowest value of chosen metric. |
| `breakup` | A boolean value indicating whether to breakup the metric value into deviance (defined as -2*loglikelihood) and penalty components. Will work only if a single metric out of "AIC", "AICc", "BIC", or "BICc" is chosen to plot. |
| `model_names` | A character string describing the names to display on X-axis for each model in order they appear in the models parameter. |

## Value

A data-frame with multiple columns containing values of several information criteria for each model specified in ‘models‘.

**model_name** An identifier name for each model object to be shown on X-axis.

**deviance** The deviance values for each model object.

**logLik** The -2*Log-Likelihood values for each model object.

**AIC** The Akaike information criteria (AIC) values for each model object.

**BIC** The Bayesian information criteria (BIC) values for each model object.

**AICc** The corrected AIC (AICc) values for each model object.

**BICc** The corrected BIC (BICc) values for each model object.

**Component** The names of the components to be shown in the plot.

**Value** The values for the components to be shown in the plot.

## Examples

```
## Fit different candidate models
mod1 <- lm(mpg ~ disp, data = mtcars)
mod2 <- lm(mpg ~ disp + hp, data = mtcars)
mod3 <- lm(mpg ~ disp + hp + wt, data = mtcars)
mod4 <- lm(mpg ~ disp + hp + wt + carb, data = mtcars)

## Group models into list
models_list <- list("Model 1" = mod1, "Model 2" = mod2,
                    "Model 3" = mod3, "Model 4" = mod4)

## Prepare data for visualisation
## Specific metric
model_selection_data(models = models_list,
                     metric = c("AIC"))

## Multiple metrics can be plotted together as well
model_selection_data(models = models_list,
                     metric = c("AIC", "BIC"))
```

```
## If single metric is specified then breakup of metric
## between goodness of fit and penalty can also be visualised
model_selection_data(models = models_list,
                     metric = c("AICc"),
                     breakup = TRUE)

## Sort models
model_selection_data(models = models_list,
                     metric = c("AICc"),
                     breakup = TRUE, sort = TRUE)

## If multiple metrics are specified then sorting
## will be done on first metric specified in list (AIC in this case)
model_selection_data(models = models_list,
                     metric = c("AIC", "BIC", "AICc", "BICc"), sort = TRUE)
```

---

## `model_selection_plot` *Visualise model selection*

---

### Description

This function accepts the output of the `model_selection_data` function and helps visualise model selection by showing a visual comparison between the information criteria for different models. It is also possible to visualise a breakup of the information criteria into deviance (goodness-of-fit) and penalty terms for each model. This could aid in understanding why a parsimonious model could be preferable over a more complex model.

### Usage

```
model_selection_plot(data)
```

### Arguments

| | |
|---|---|
| `data` | A data-frame consisting of the information criteria for different regression models. This data could be prepared using the ‘model_selection_data‘ function, or be created manually by the user with the necessary information stored into the respective columns. |

### Value

A ggplot2 object

### Examples

```
## Fit different candidate models
mod1 <- lm(mpg ~ disp, data = mtcars)
mod2 <- lm(mpg ~ disp + hp, data = mtcars)
mod3 <- lm(mpg ~ disp + hp + wt, data = mtcars)
mod4 <- lm(mpg ~ disp + hp + wt + carb, data = mtcars)
```

```
## Group models into list
models_list <- list("Model 1" = mod1, "Model 2" = mod2,
                    "Model 3" = mod3, "Model 4" = mod4)

## Prepare data for visualisation
## Specific metric
plot_data1 <- model_selection_data(models = models_list,
                                   metric = c("AIC"))
## Visualise
model_selection_plot(plot_data1)

## Multiple metrics can be plotted together as well
plot_data2 <- model_selection_data(models = models_list,
                                   metric = c("AIC", "BIC"))
## Visualise
model_selection_plot(plot_data2)

## If single metric is specified then breakup of metric
## between goodness of fit and penalty can also be visualised
plot_data3 <- model_selection_data(models = models_list,
                                   metric = c("AICc"),
                                   breakup = TRUE)
## Visualise
model_selection_plot(plot_data3)
```

---

## prediction_contributions *Model term contributions to predicted response*

---

### Description

A stacked bar_chart is shown where the individual contributions (parameter estimate * predictor value) for each term in a statistical model are stacked on top of another. The total height of the stacked bar gives the value of the predicted response. The uncertainty around the predicted response can also be shown on the plot. This is a wrapper function specifically designed for statistical models fit using the `DI()` function from the `DImodels` R package and it implicitly calls `prediction_contributions_data` followed by `prediction_contributions_plot`. If your model object isn't fit using DImodels, the associated data and plot functions can instead be called manually.

### Usage

```
prediction_contributions(
  model,
  data = NULL,
  add_var = list(),
  groups = list(),
  conf.level = 0.95,
  bar_labs = rownames(data),
  colours = NULL,
  se = FALSE,
  FG = NULL,
  interval = c("confidence", "prediction", "none"),
  bar_orientation = c("vertical", "horizontal"),
  facet_var = NULL,
  plot = TRUE,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| `model` | A Diversity Interactions model object fit by using the `DI()` function from the `DImodels` package. |
| `data` | A user-defined data-frame containing values for compositional variables along with any additional variables that the user wishes to predict for. If left blank, a selection of observations (2 from each level of richness) from the original data used to fit the model would be selected. |
| `add_var` | A list specifying values for additional predictor variables in the model independent of the compositional predictor variables. This could be useful for comparing the predictions across different values for a non-compositional variable. If specified as a list, it will be expanded to show a plot for each unique combination of values specified, while if specified as a data-frame, one plot would be generated for each row in the data and they will be arranged in a grid according to the value specified in ‘nrow‘ and ‘ncol‘. |
| `groups` | A list specifying groupings to arrange coefficients into. The coefficients within a group will be added together and shown as a single component on the respective bars in the plot. This could be useful for grouping multiple similar terms into a single term for better visibility. |
| `conf.level` | The confidence level for calculating confidence or prediction intervals. |
| `bar_labs` | The labels to be shown for each bar in the plot. The user has three options: - By default, the row-names in the data would be used as labels for the bars. - A character string or numeric index indicating an ID column in data. - A character vector of same length as the number of rows in the data, which manually specifies the names for each bar. If none of the three options are available, the function would assign a unique ID for each bar. |
| `colours` | A character vector specifying the colours for the contributions of the different coefficients. If not specified, a default colour-scheme would be chosen, however it might be uninformative in some situations (for examples when manual groupings are specified using ‘groups‘ parameter). |
| `se` | A logical value indicating whether to show prediction intervals for predictions in the plot. |
| `FG` | A higher level grouping for the compositional variables in the data. Variables belonging to the same group will be assigned with different shades of the same |

colour. The user can manually specify a character vector giving the group each variable belongs to. If left empty the function will try to get a grouping from the original DI model object.

`interval`
: Type of interval to calculate:

  **"none"** No interval to be calculated.

  **"confidence" (default)** Calculate a confidence interval.

  **"prediction"** Calculate a prediction interval.

`bar_orientation`
: One of "vertical" or "horizontal" indicating the orientation of the bars. Defaults to a vertical orientation.

`facet_var`
: A character string or numeric index identifying the column in the data to be used for faceting the plot into multiple panels.

`plot`
: A boolean variable indicating whether to create the plot or return the prepared data instead. The default 'TRUE' creates the plot while 'FALSE' would return the prepared data for plotting. Could be useful for if user wants to modify the data first and then call the plotting function manually.

`nrow`
: Number of rows in which to arrange the final plot (when 'add_var' is specified).

`ncol`
: Number of columns in which to arrange the final plot (when 'add_var' is specified).

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if 'plot = FALSE')

## Examples

```
#' ## Load DImodels package to fit the model
library(DImodels)

## Load data
data(sim2)

## Fit DI model
model1 <- DI(prop = 3:6, DImodel = 'FULL', data = sim2, y = 'response')

## Create visualisation
## If no communities are specified 2 communities at
## each level of richness from the original data are used
prediction_contributions(model1)

## Can also manually specify communities of interest
my_comms <- data.frame(p1 = c(1, 0, 0,   0.5, 1/3, 0.25),
                       p2 = c(0, 0, 0.5, 0,   1/3, 0.25),
                       p3 = c(0, 1, 0.5, 0,   1/3, 0.25),
                       p4 = c(0, 0, 0,   0.5, 0,   0.25))

prediction_contributions(model1, data = my_comms)

## Group contributions to show as a single component on the plot
```

```
prediction_contributions(model1, data = my_comms,
                         groups = list("Interactions" = c("`p1:p2`", "`p1:p3`",
                                                          "`p1:p4`", "`p2:p3`",
                                                          "`p2:p4`", "`p3:p4`")))

## Add a prediction interval using `se = TRUE` and show bars horizontally
prediction_contributions(model1, data = my_comms, se = TRUE,
                         bar_orientation = "horizontal",
                         groups = list("Interactions" = c("`p1:p2`", "`p1:p3`",
                                                          "`p1:p4`", "`p2:p3`",
                                                          "`p2:p4`", "`p3:p4`")))

## Facet the plot on any variable
my_comms$richness <- c(1, 1, 2, 2, 3, 4)
## Use `facet_var`
prediction_contributions(model1, data = my_comms, facet_var = "richness",
                         bar_orientation = "horizontal",
                         groups = list("Interactions" = c("`p1:p2`", "`p1:p3`",
                                                          "`p1:p4`", "`p2:p3`",
                                                          "`p2:p4`", "`p3:p4`")))

## Can also add additional variables independent of the simplex design
## to get a separate plot for unique combination of the variables
prediction_contributions(model1, data = my_comms,
                         add_var = list("block" = factor(c(1, 2),
                                                         levels = c(1, 2, 3, 4))))

## Manually specify colours and bar labels
## Model has 10 terms but we grouped 6 of them into 1 term,
## so we need to specify 5 colours (4 ungrouped terms + 1 grouped term)
## Bar labels can be specified using `bar_labs`
## Also, using nrow to arrange plots in rows
prediction_contributions(model1, data = my_comms,
                         colours = c("steelblue1", "steelblue4",
                                     "orange", "orange4",
                                     "grey"),
                         bar_labs = c("p1 Mono", "p3 Mono", "1/2 p2 p3",
                                      "1/2 p1 p4", "1/3 p1 p2 p3", "Centroid"),
                         add_var = list("block" = factor(c(1, 2),
                                                         levels = c(1, 2, 3, 4))),
                         nrow = 2,
                         groups = list("Interactions" = c("`p1:p2`", "`p1:p3`",
                                                          "`p1:p4`", "`p2:p3`",
                                                          "`p2:p4`", "`p3:p4`")))

## Specify `plot = FALSE` to not create the plot but return the prepared data
head(prediction_contributions(model1, data = my_comms, plot = FALSE,
                              facet_var = "richness",
                              bar_orientation = "horizontal"))
```

## prediction_contributions_data *Model term contributions to predicted response*

### Description

The helper function for preparing the data to split the predicted response from a regression model into contributions (predictor coefficient * predictor value) by the terms in the model. The output of this function can be passed to the ‘prediction_contributions_plot‘ function to visualise the results.

### Usage

```
prediction_contributions_data(
  data,
  model = NULL,
  coefficients = NULL,
  coeff_cols = NULL,
  vcov = NULL,
  add_var = list(),
  groups = list(),
  conf.level = 0.95,
  interval = c("confidence", "prediction", "none"),
  bar_labs = rownames(data)
)
```

### Arguments

| | |
|---|---|
| `data` | A user-defined data-frame containing values for compositional variables along with any additional variables that the user wishes to predict for. If left blank, a selection of observations (2 from each level of richness) from the original data used to fit the model would be selected. |
| `model` | A Diversity Interactions model object fit by using the `DI()` function from the `DImodels` package. |
| `coefficients` | If a regression model is not available (or can’t be fit in R), the regression coefficients from a model fit in some other language can be used to calculate predictions. However, the user would have to ensure there’s an appropriate one-to-one positional mapping between the data columns and the coefficient values. Further, they would also have to provide a variance-covariance matrix of the coefficients in the ‘vcov‘ parameter if they want the associated CI for the prediction or it would not be possible to calculate confidence/prediction intervals using this method. |
| `coeff_cols` | If ‘coefficients‘ are specified and a one-to-one positional mapping between the data-columns and coefficient vector is not present. A character string or numeric index can be specified here to reorder the data columns and match the corresponding coefficient value to the respective data column. See the "Use model coefficients for prediction" section in examples. |

| | |
|---|---|
| `vcov` | If regression coefficients are specified, then the variance-covariance matrix of the coefficients can be specified here to calculate the associated confidence interval around each prediction. Failure to do so would result in no confidence intervals being returned. Ensure ‘coefficients‘ and ‘vcov‘ have the same positional mapping with the data. |
| `add_var` | A list specifying values for additional predictor variables in the model independent of the compositional predictor variables. This could be useful for comparing the predictions across different values for a non-compositional variable. If specified as a list, it will be expanded to show a plot for each unique combination of values specified, while if specified as a data-frame, one plot would be generated for each row in the data and they will be arranged in a grid according to the value specified in ‘nrow‘ and ‘ncol‘. |
| `groups` | A list specifying groupings to arrange coefficients into. The coefficients within a group will be added together and shown as a single component on the respective bars in the plot. This could be useful for grouping multiple similar terms into a single term for better visibility. |
| `conf.level` | The confidence level for calculating confidence or prediction intervals. |
| `interval` | Type of interval to calculate:<br>**"none"** No interval to be calculated.<br>**"confidence" (default)** Calculate a confidence interval.<br>**"prediction"** Calculate a prediction interval. |
| `bar_labs` | The labels to be shown for each bar in the plot. The user has three options: - By default, the row-names in the data would be used as labels for the bars. - A character string or numeric index indicating an ID column in data. - A character vector of same length as the number of rows in the data, which manually specifies the names for each bar. If none of the three options are available, the function would assign a unique ID for each bar. |

## Value

A data-frame with the following columns. Any additional columns which weren’t used when fitting the model would also be present.

**.Community** An identifier column to discern each observation in the data. These are the labels which will be displayed for the bars in the plot.

**.add_str_ID** An identifier column for grouping the cartesian product of all additional columns specified in ‘add_var‘ parameter (if ‘add_var‘ is specified).

**.Pred** The predicted repsonse for each observation.

**.Lower** The lower limit of the prediction interval for each observation.

**.Upper** The lower limit of the prediction interval for each observation.

**.Contributions** An identifier describing the name of the coefficient contributing to the response.

**.Value** The contributed value of the respective coefficient/group to the total prediction.

**Examples**

```
library(DImodels)
library(dplyr)

## Load data
data(sim2)

## Fit model
mod <- glm(response ~ 0 + (p1 + p2 + p3 + p4)^2, data = sim2)

prediction_contributions_data(data = sim2[c(1,5,9,11), ],
                              model = mod)

## Specific coefficients can also be grouped together
## Either by their indices in the model coefficient vector
prediction_contributions_data(data = sim2[c(1,5,9,11), ],
                              model = mod,
                              groups = list("Interactions" = 5:10))
## Or by specifying the coefficient names as character strings
prediction_contributions_data(data = sim2[c(1,5,9,11), ],
                              model = mod,
                              groups = list("p1_Ints" = c("p1:p2",
                                                          "p1:p3",
                                                          "p1:p4")))

## Additional variables can also be added to the data by either specifying
## them directly in the `data` or by using the `add_var` argument
## Refit model
sim2$block <- as.numeric(sim2$block)
new_mod <- update(mod, ~. + block, data = sim2)
## This model has block so we can either specify block in the data
subset_data <- sim2[c(1,5,9,11), 2:6]
subset_data
head(prediction_contributions_data(data = subset_data,
                                   model = new_mod))
## Or we could add the variable using `add_var`
subset_data <- sim2[c(1,5,9,11), 3:6]
subset_data
head(prediction_contributions_data(data = subset_data,
                                   model = new_mod,
                                   add_var = list(block = c(1, 2))))
## The benefit of specifying the variable this way is we have an ID
## columns now called `.add_str_ID` which would be used to create a
## separate plot for each value of the additional variable


## Model coefficients can also be used, but then user would have
## to specify the data with all columns corresponding to each coefficient
coef_data <- sim2 %>%
               mutate(`p1:p2` = p1*p2, `p1:p3` = p1*p2, `p1:p4` = p1*p4,
                      `p2:p3` = p2*p3, `p2:p4` = p2*p4, `p3:p4` = p3*p4) %>%
               select(p1, p2, p3, p4,
                       `p1:p2`, `p1:p3`, `p1:p4`,
                       `p2:p3`, `p2:p4`, `p3:p4`) %>%
                slice(1,5,9,11)
print(coef_data)
print(mod$coefficients)
prediction_contributions_data(data = coef_data,
                              coefficients = mod$coefficients,
                              interval = "none")
## To get uncertainity using coefficients vcov matrix would have to specified
prediction_contributions_data(data = coef_data,
                              coefficients = mod$coefficients,
                              vcov = vcov(mod))

## Specifying `bar_labs`
## Our data has four rows so we'd need four labels in bar_labs
prediction_contributions_data(data = coef_data,
                              coefficients = mod$coefficients,
                              vcov = vcov(mod),
                              bar_labs = c("p1 Domm", "p2 Domm",
                                           "p3 Domm", "p4 Domm"))
```

---

## prediction_contributions_plot

*Visualise model term contributions to predicted response*

---

### Description

The plotting function to visualise the predicted response from a regression model as a stacked bar-chart showing the contributions (predictor coefficient * predictor value) of each model term to the total predicted value (represented by the total height of the bar). Requires the output of the ‘prediction_contributions_data‘ as an input in the ‘data‘ parameter.

### Usage

```
prediction_contributions_plot(
  data,
  colours = NULL,
  se = FALSE,
  bar_orientation = c("vertical", "horizontal"),
  facet_var = NULL,
  nrow = 0,
  ncol = 0
)
```

### Arguments

| | |
|---|---|
| `data` | A data-frame which is the output of the ‘prediction_contributions_data‘ function, consisting of the predicted response split into the contributions by the different coefficients. |

| | |
|---|---|
| `colours` | A character vector specifying the colours for the contributions of the different coefficients. If not specified, a default colour-scheme would be chosen, however it could be uninformative and it is recommended to manually choose contrasting colours for each coefficient group to render a more informative plot. |
| `se` | A logical value indicating whether to show prediction intervals for predictions in the plot. |
| `bar_orientation` | One of "vertical" or "horizontal" indicating the orientation of the bars. Defaults to a vertical orientation. |
| `facet_var` | A character string or numeric index identifying the column in the data to be used for faceting the plot into multiple panels. |
| `nrow` | Number of rows in which to arrange the final plot (when ‘add_var‘ is specified). |
| `ncol` | Number of columns in which to arrange the final plot (when ‘add_var‘ is specified). |

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if ‘plot = FALSE‘)

## Examples

```
library(DImodels)
library(dplyr)

## Load data
data(sim2)
sim2$AV <- DI_data_E_AV(data = sim2, prop = 3:6)$AV

## Fit model
mod <- glm(response ~ 0 + (p1 + p2 + p3 + p4 + AV), data = sim2)

## Create data for plotting
plot_data <- prediction_contributions_data(data = sim2[c(1,5,9,11,15,19,23), ],
                                           model = mod)
## Create plot
prediction_contributions_plot(data = plot_data)

## Choose distinct colours for groups of coefficients for better visibility
ID_cols <- get_colours(4, FG = c("G", "G", "H", "H"))
int_cols <- "#808080"
cols <- c(ID_cols, int_cols)
## Specify colours using `cols`
prediction_contributions_plot(data = plot_data, colours = cols)

## Show prediction intervals
prediction_contributions_plot(data = plot_data, colours = cols, se = TRUE)

## Change orientation of bars using `bar_orientation`
prediction_contributions_plot(data = plot_data, colours = cols,
                              se = TRUE, bar_orientation = "horizontal")
```

```
## Facet plot based on a variable in the data
prediction_contributions_plot(data = plot_data, colours = cols,
                              se = TRUE, bar_orientation = "horizontal",
                              facet_var = "block")

## If multiple plots are desired `add_var` can be specified during
## data preparation and the plots can be arranged using nrow and ncol
sim2$block <- as.character(sim2$block)
new_mod <- update(mod, ~. + block, data = sim2)
plot_data <- prediction_contributions_data(data = sim2[c(1,5,9,11,15,19,23), c(3:6, 8)],
                                           model = new_mod,
                                           add_var = list("block" = c("1", "2")))
## Arrange in two columns
prediction_contributions_plot(data = plot_data, ncol = 2)
```

---

## `prop_to_tern_proj` *Project 3-d compositional data onto x-y plane and vice versa*

---

### Description

Points in the 3-d simplex space with coordinates (x, y ,z) such that x + y + z = 1 are projected into the 2-d plane they reside in. This function can be used to convert the 3-d compositional data into 2-d and then be overlayed on the plots output by `ternary_plot`, `conditional_ternary_plot` and `grouped_ternary_plot`.

### Usage

```
prop_to_tern_proj(data, prop, x = ".x", y = ".y")

tern_to_prop_proj(data, x, y, prop = c("p1", "p2", "p3"))
```

### Arguments

| | |
|---|---|
| `data` | A data-frame containing the x-y coordinates of the points. |
| `prop` | A character vector specifying the columns names of variable containing the projected compositions. Default is "p1", "p2", and "p3". |
| `x` | A character string specifying the name for the column containing the x component of the x-y projection of the simplex. |
| `y` | A character string specifying the name for the column containing the y component of the x-y projection of the simplex. |

### Value

A data-frame with the following two columns appended (when transforming to x-y projection)

**.x (or value specified in "x")** The x component of the x-y projection of the simplex point.

**.y (or value specified in "y")** The y component of the x-y projection of the simplex point.

A data-frame with the following three columns appended (when transforming to compositional projection)

**p1 (or first value specified in "prop")** The first component of the 3-d simplex point.

**p2 (or second value specified in "prop")** The second component of the 3-d simplex point.

**p3 (or third value specified in "prop")** The third component of the 3-d simplex point.

## Examples

```
## Convert proportions to x-y co-ordinates
library(DImodels)
data(sim0)
sim0 <- sim0[1:16, ]

prop_to_tern_proj(data = sim0, prop = c("p1", "p2", "p3"))

# Change names of the x and y projections
prop_to_tern_proj(data = sim0, prop = c("p1", "p2", "p3"),
                  x = "x-proj", y = "y-proj")
## Convert x-y co-ordinates to proportions
library(DImodels)
data(sim0)
sim0 <- sim0[1:16, ]

proj_data <- prop_to_tern_proj(data = sim0, prop = c("p1", "p2", "p3"))

tern_to_prop_proj(data = proj_data, x = ".x", y = ".y")

# Change prop names
tern_to_prop_proj(data = proj_data, x = ".x", y = ".y",
                  prop = c("prop1", "prop2", "prop3"))
```

---

`simplex_path` *Visualising the change in a response variable between two points in the simplex space*

---

## Description

This function will prepare the underlying data and plot the results for visualising the change in a response variable as we move across a straight line between two points in the simplex space in a single function call. The two sets of points specified by the ‘starts‘ and ‘ends‘ parameters are joined by a straight line across the simplex space and the response is predicted for the starting, ending and intermediate communities along this line. The associated uncertainty along this prediction is also shown. The generated plot will show individual curves indicating the variation in the response between the points. ‘`Pie-glyphs`‘ are used to highlight the compositions of the starting, ending and midpoint of the straight line between the two points. This is a wrapper function specifically for statistical models fit using the `DI()` function from the `DImodels` R package and would implicitly call `simplex_path_data` followed by `simplex_path_plot`. If your model object isn't fit using DImodels, consider calling these functions manually.

## Usage

```
simplex_path(
  model,
  starts,
  ends,
  add_var = list(),
  interval = c("confidence", "prediction", "none"),
  conf.level = 0.95,
  se = FALSE,
  pie_positions = c(0, 0.5, 1),
  pie_colours = NULL,
  pie_radius = 0.3,
  FG = NULL,
  facet_var = NULL,
  plot = TRUE,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| `model` | A Diversity Interactions model object fit by using the '`DI()`' function from the '`DImodels`' package. |
| `starts` | A data-frame specifying the starting proportions of the compositional variables. If a model object is specified then this data should contain all the variables present in the model object including any additional non-compositional variables. If a coefficient vector is specified then data should contain same number of columns as the number of elements in the coefficient vector and a one-to-one positional mapping would be assumed between the data columns and the elements of the coefficient vector. |
| `ends` | A data-frame specifying the ending proportions of the compositional variables. If a model object is specified then this data should contain all the variables present in the model object including any additional non-compositional variables. If a coefficient vector is specified then data should contain same number of columns as the number of elements in the coefficient vector and a one-to-one positional mapping would be assumed between the data columns and the elements of the coefficient vector. |
| `add_var` | A list specifying values for additional variables in the model other than the proportions (i.e. not part of the simplex design). This would be useful to compare the predictions across different values for a categorical variable. One plot will be generated for each unique combination of values specified here. |
| `interval` | Type of interval to calculate:<br>**"none"** No interval to be calculated.<br>**"confidence" (default)** Calculate a confidence interval.<br>**"prediction"** Calculate a prediction interval. |
| `conf.level` | The confidence level for calculating confidence/prediction intervals. Default is 0.95. |

| | |
|---|---|
| `se` | A boolean variable indicating whether to plot confidence intervals associated with the effect of species increase or decrease |
| `pie_positions` | A numeric vector with values between 0 and 1 (both inclusive) indicating the positions along the X-axis at which to show pie-glyphs for each curve. Default is c(0, 0.5, 1) meaning that pie-glyphs with be shown at the start, midpoint and end of each curve. |
| `pie_colours` | A character vector indicating the colours for the slices in the pie-glyphs. If left NULL, the colour blind friendly colours will be for the pie-glyph slices. |
| `pie_radius` | A numeric value specifying the radius (in cm) for the pie-glyphs. Default is 0.3 cm. |
| `FG` | A higher level grouping for the compositional variables in the data. Variables belonging to the same group will be assigned with different shades of the same colour. The user can manually specify a character vector giving the group each variable belongs to. If left empty the function will try to get a grouping from the original `DI` model object. |
| `facet_var` | A character string or numeric index identifying the column in the data to be used for faceting the plot into multiple panels. |
| `plot` | A boolean variable indicating whether to create the plot or return the prepared data instead. The default 'TRUE' creates the plot while 'FALSE' would return the prepared data for plotting. Could be useful for if user wants to modify the data first and then call the plotting function manually. |
| `nrow` | Number of rows in which to arrange the final plot (when 'add_var' is specified). |
| `ncol` | Number of columns in which to arrange the final plot (when 'add_var' is specified). |

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if 'plot = FALSE')

## Examples

```
library(DImodels)
data(sim2)

# Fit model
mod <- DI(y = "response", prop = 3:6, DImodel = "AV", data = sim2)

# Create plot
# Move from p3 monoculture to p4 monoculture
simplex_path(model = mod,
             starts = data.frame(p1 = 0, p2 = 0, p3 = 1, p4 = 0),
             ends = data.frame(p1 = 0, p2 = 0, p3 = 0, p4 = 1))

# Move from each 70% dominant mixtures to p1 monoculture
simplex_path(model = mod,
             starts = sim2[c(1, 5, 9, 13), 3:6],
             ends = data.frame(p1 = 1, p2 = 0, p3 = 0, p4 = 0))
```

```
# Move from centroid community to each monoculture
simplex_path(model = mod,
             starts = sim2[c(18),],
             ends = sim2[c(48, 52, 56, 60), ])

# Show change across multiple points simultaneously and show confidence bands
# using `se = TRUE`
simplex_path(model = mod,
             starts = sim2[c(1, 17, 22), ],
             ends = sim2[c(5, 14, 17), ], se = TRUE)

# Change pie_colours using `pie_colours` and show pie-glyph at different
# points along the curve using `pie_positions`
simplex_path(model = mod,
             starts = sim2[c(1, 17, 22), ],
             ends = sim2[c(5, 14, 17), ], se = TRUE,
             pie_positions = c(0, 0.25, 0.5, 0.75, 1),
             pie_colours = c("steelblue1", "steelblue4", "orange1", "orange4"))

# Facet based on existing variables

simplex_path(model = mod,
             starts = sim2[c(1, 17, 22), ],
             ends = sim2[c(5, 14, 17), ], se = TRUE, facet_var = "block",
             pie_colours = c("steelblue1", "steelblue4", "orange1", "orange4"))

# Add additional variables and create a separate plot for each
simplex_path(model = mod,
             starts = sim2[c(1, 17, 22), 3:6],
             ends = sim2[c(5, 14, 17), 3:6], se = TRUE,
             pie_colours = c("steelblue1", "steelblue4", "orange1", "orange4"),
             add_var = list("block" = factor(c(1, 3),
                                             levels = c(1, 2, 3, 4))))


## Specify `plot = FALSE` to not create the plot but return the prepared data
head(simplex_path(model = mod, plot = FALSE,
                  starts = sim2[c(1, 17, 22), 3:6],
                  ends = sim2[c(5, 14, 17), 3:6], se = TRUE,
                  pie_colours = c("steelblue1", "steelblue4",
                                  "orange1", "orange4"),
                  add_var = list("block" = factor(c(1, 3),
                                                  levels = c(1, 2, 3, 4)))))
```

## simplex_path_data *Creating data for visualising the change in a response variable between two points in the simplex space*

## Description

This is the helper function to prepare the underlying data for visualising the change in a response variable between two points in a simplex space. The two points specified by the ‘starts‘ and ‘ends‘ parameters are joined by a straight line across the simplex space and the response is predicted for the starting, ending and intermediate communities along this line. The associated uncertainty along this prediction is also returned. The output of this function can be passed to the `simplex_path_plot` function to visualise the change in response.

## Usage

```
simplex_path_data(starts, ends, prop, add_var = list(), prediction = TRUE, ...)
```

## Arguments

| | |
|---|---|
| `starts` | A data-frame specifying the starting proportions of the compositional variables. If a model object is specified then this data should contain all the variables present in the model object including any additional non-compositional variables. If a coefficient vector is specified then data should contain same number of columns as the number of elements in the coefficient vector and a one-to-one positional mapping would be assumed between the data columns and the elements of the coefficient vector. |
| `ends` | A data-frame specifying the ending proportions of the compositional variables. If a model object is specified then this data should contain all the variables present in the model object including any additional non-compositional variables. If a coefficient vector is specified then data should contain same number of columns as the number of elements in the coefficient vector and a one-to-one positional mapping would be assumed between the data columns and the elements of the coefficient vector. |
| `prop` | A vector of column names identifying the columns containing the variable proportions (i.e., compositional columns) in the data. |
| `add_var` | A list or data-frame specifying values for additional variables in the model other than the proportions (i.e. not part of the simplex design). This could be useful for comparing the predictions across different values for a non-compositional variable. If specified as a list, it will be expanded to show a plot for each unique combination of values specified, while if specified as a data-frame, one plot would be generated for each row in the data. |
| `prediction` | A logical value indicating whether to pass the final data to the ‘add_prediction‘ function and append the predictions to the data. Default value is `TRUE`, but often it would be desirable to make additional changes to the data before making any predictions, so the user can set this to `FALSE` and manually call the ‘add_prediction‘ function. |
| `...` | Arguments passed on to `add_prediction`<br>`model` A regression model object which will be used to make predictions for the observations in ‘data‘. Will override ‘coefficients‘ if specified.<br>`coefficients` If a regression model is not available (or can’t be fit in R), the regression coefficients from a model fit in some other language can be used |

to calculate predictions. However, the user would have to ensure there's an appropriate one-to-one positional mapping between the data columns and the coefficient values. Further, they would also have to provide a variance-covariance matrix of the coefficients in the ‘vcov‘ parameter if they want the associated CI for the prediction or it would not be possible to calculate confidence/prediction intervals using this method.

`vcov` If regression coefficients are specified, then the variance-covariance matrix of the coefficients can be specified here to calculate the associated confidence interval around each prediction. Failure to do so would result in no confidence intervals being returned. Ensure ‘coefficients‘ and ‘vcov‘ have the same positional mapping with the data.

`coeff_cols` If ‘coefficients‘ are specified and a one-to-one positional mapping between the data-columns and coefficient vector is not present. A character string or numeric index can be specified here to reorder the data columns and match the corresponding coefficient value to the respective data column. See the "Use model coefficients for prediction" section in examples.

`conf.level` The confidence level for calculating confidence/prediction intervals. Default is 0.95.

`interval` Type of interval to calculate:

**"none" (default)** No interval to be calculated.

**"confidence"** Calculate a confidence interval.

**"prediction"** Calculate a prediction interval.

## Value

A data frame with the following columns appended at the end

**.InterpConst** The value of the interpolation constant for creating the intermediate compositions between the start and end compositions.

**.Group** An identifier column to discern between the different curves.

**.add_str_ID** An identifier column for grouping the cartesian product of all additional columns specified in ‘add_var‘ parameter (if ‘add_var‘ is specified).

**.Pred** The predicted response for each observation.

**.Lower** The lower limit of the prediction/confidence interval for each observation.

**.Upper** The upper limit of the prediction/confidence interval for each observation.

## Examples

```
library(DImodels)

## Load data
data(sim2)

## Fit model
mod <- glm(response ~ (p1 + p2 + p3 + p4)^2 + 0, data = sim2)

## Create data for visualising change in response as we move from
## a species dominated by 70% of one species to a monoculture of
## same species
head(simplex_path_data(starts = sim2[c(1, 5, 9, 13), 3:6],
                       ends = sim2[c(48, 52, 56, 60), 3:6],
                       prop = c("p1", "p2", "p3", "p4"),
                       model = mod))

## Create data for visualising change in response as we move from
## the centroid mixture to each monoculture
## If either of starts or ends have only row, then they'll be recycled
## to match the number of rows in the other
## Notice starts has only one row here, but will be recycled to have 4
## since ends has 4 four rows
head(simplex_path_data(starts = sim2[c(18),3:6],
                       ends = sim2[c(48, 52, 56, 60),3:6],
                       prop = c("p1", "p2", "p3", "p4"),
                       model = mod))

## Changing the confidence level for the prediction interval
## Use `conf.level` parameter
head(simplex_path_data(starts = sim2[c(18), 3:6],
                       ends = sim2[c(48, 52, 56, 60),3:6],
                       prop = c("p1", "p2", "p3", "p4"),
                       model = mod, conf.level = 0.99))

## Adding additional variables to the data using `add_var`
## Notice the new .add_str_ID column in the output
sim2$block <- as.numeric(sim2$block)
new_mod <- update(mod, ~ . + block, data = sim2)
head(simplex_path_data(starts = sim2[c(18), 3:6],
                       ends = sim2[c(48, 52, 56, 60), 3:6],
                       prop = c("p1", "p2", "p3", "p4"),
                       model = new_mod, conf.level = 0.99,
                       add_var = list("block" = c(1, 2))))

## Use predict = FALSE to get raw data structure
out_data <- simplex_path_data(starts = sim2[c(18), 3:6],
                              ends = sim2[c(48, 52, 56, 60), 3:6],
                              prop = c("p1", "p2", "p3", "p4"),
                              model = new_mod,
                              prediction = FALSE)
head(out_data)
## Manually add block
out_data$block = 3
## Call `add_prediction` to get prediction
head(add_prediction(data = out_data, model = new_mod, interval = "conf"))
```

## simplex_path_plot — *Visualising the change in a response variable between two points in the simplex space*

## Description

The helper function for plotting the change in a response variable over a straight line between two points across the simplex space. The output of the `simplex_path_data` function (with any desired modifications) should be passed here. The generated plot will show individual curves indicating the variation in the response between the points. ‘`Pie-glyphs`‘ are used to highlight the compositions of the starting, ending and midpoint of the straight line between the two points.

## Usage

```
simplex_path_plot(
  data,
  prop = NULL,
  pie_positions = c(0, 0.5, 1),
  pie_radius = 0.3,
  pie_colours = NULL,
  se = FALSE,
  facet_var = NULL,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame created using the `simplex_path_data` function. |
| `prop` | A vector of column names or indices identifying the columns containing the species proportions in the data. Will be inferred from the data if it is created using the ‘`simplex_path_data`‘ function, but the user also has the flexibility of manually specifying the values. |
| `pie_positions` | A numeric vector with values between 0 and 1 (both inclusive) indicating the positions along the X-axis at which to show pie-glyphs for each curve. Default is c(0, 0.5, 1) meaning that pie-glyphs with be shown at the start, midpoint and end of each curve. |
| `pie_radius` | A numeric value specifying the radius (in cm) for the pie-glyphs. Default is 0.3 cm. |
| `pie_colours` | A character vector indicating the colours for the slices in the pie-glyphs. If left NULL, the colour blind friendly colours will be for the pie-glyph slices. |
| `se` | A boolean variable indicating whether to plot confidence intervals associated with the effect of species increase or decrease |
| `facet_var` | A character string or numeric index identifying the column in the data to be used for faceting the plot into multiple panels. |
| `nrow` | Number of rows in which to arrange the final plot (when ‘add_var‘ is specified). |
| `ncol` | Number of columns in which to arrange the final plot (when ‘add_var‘ is specified). |

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if ‘plot = FALSE‘)

## Examples

```
library(DImodels)

## Load data
data(sim2)

## Fit model
mod <- glm(response ~ (p1 + p2 + p3 + p4)^2 + 0, data = sim2)

## Visualise change as we move from the centroid community to each monoculture
plot_data <- simplex_path_data(starts = sim2[c(19, 20, 19, 20), ],
                               ends = sim2[c(47, 52, 55, 60), ],
                               prop = c("p1", "p2", "p3", "p4"),
                               model = mod)
## prop will be inferred from data
simplex_path_plot(data = plot_data)

## Show specific curves
simplex_path_plot(data = plot_data[plot_data$.Group %in% c(1, 4), ])

## Show uncertainty using `se = TRUE`
simplex_path_plot(data = plot_data[plot_data$.Group %in% c(1, 4), ],
                  se = TRUE)

## Change colours of pie-glyphs using `pie_colours`
simplex_path_plot(data = plot_data[plot_data$.Group %in% c(1, 4), ],
                  se = TRUE,
                  pie_colours = c("steelblue1", "steelblue4", "orange1", "orange4"))

## Show pie-glyphs at different points along the curve using `pie_positions`
simplex_path_plot(data = plot_data[plot_data$.Group %in% c(1, 4), ],
                  se = TRUE,
                  pie_positions = c(0, 0.25, 0.5, 0.75, 1),
                  pie_colours = c("steelblue1", "steelblue4", "orange1", "orange4"))

## Facet plot based on specific variables
simplex_path_plot(data = plot_data,
                  se = TRUE,
                  facet_var = "block",
                  pie_colours = c("steelblue1", "steelblue4", "orange1", "orange4"))

## Simulataneously create multiple plots for additional variables
sim2$block <- as.numeric(sim2$block)
new_mod <- update(mod, ~ . + block, data = sim2)
plot_data <- simplex_path_data(starts = sim2[c(18), 3:6],
                       ends = sim2[c(48, 60), 3:6],
                       prop = c("p1", "p2", "p3", "p4"),
                       model = new_mod, conf.level = 0.95,
                       add_var = list("block" = c(1, 2)))

simplex_path_plot(data = plot_data,
                  pie_colours = c("steelblue1", "steelblue4",
                              "orange1", "orange4"),
                   nrow = 1, ncol = 2)
```

## ternary_data *Prepare data for showing contours in ternary diagrams.*

### Description

The data preparation function for creating an equally spaced grid of three compositional variables (i.e., the three variables sum to 1 at each point along the grid). The projection of each point in the grid on the x-y plane is also calculated. This data can be used with a relevant statistical model to predict the response across the ternary surface. The output of this function can then be passed to the `ternary_plot` function to visualise the change in the response as a contour plot.
*Note:* This function works only for models with three compositional predictors. For models with more than three compositional predictors see `conditional_ternary`.

### Usage

```
ternary_data(
  prop = c(".P1", ".P2", ".P3"),
  add_var = list(),
  resolution = 3,
  prediction = TRUE,
  ...
)
```

### Arguments

| | |
|---|---|
| `prop` | A character vector specifying the columns names of compositional variables whose proportions to manipulate. Default is ".P1", ".P2", and ".P3". |
| `add_var` | A list or data-frame specifying values for additional variables in the model other than the proportions (i.e. not part of the simplex design). This could be useful for comparing the predictions across different values for a non-compositional variable. If specified as a list, it will be expanded to show a plot for each unique combination of values specified, while if specified as a data-frame, one plot would be generated for each row in the data. |
| `resolution` | A number between 1 and 10 describing the resolution of the resultant graph. A high value would result in a higher definition figure but at the cost of being computationally expensive. |
| `prediction` | A logical value indicating whether to pass the final data to the ‘add_prediction‘ function and append the predictions to the data. Default value is `TRUE`, but often it would be desirable to make additional changes to the data before making any predictions, so the user can set this to `FALSE` and manually call the ‘add_prediction‘ function. |
| `...` | Arguments passed on to `add_prediction` |

`model` A regression model object which will be used to make predictions for the observations in ‘data‘. Will override ‘coefficients‘ if specified.

`coefficients` If a regression model is not available (or can’t be fit in R), the regression coefficients from a model fit in some other language can be used to calculate predictions. However, the user would have to ensure there’s an appropriate one-to-one positional mapping between the data columns and the coefficient values. Further, they would also have to provide a variance-covariance matrix of the coefficients in the ‘vcov‘ parameter if they want the associated CI for the prediction or it would not be possible to calculate confidence/prediction intervals using this method.

`vcov` If regression coefficients are specified, then the variance-covariance matrix of the coefficients can be specified here to calculate the associated confidence interval around each prediction. Failure to do so would result in no confidence intervals being returned. Ensure ‘coefficients‘ and ‘vcov‘ have the same positional mapping with the data.

`coeff_cols` If ‘coefficients‘ are specified and a one-to-one positional mapping between the data-columns and coefficient vector is not present. A character string or numeric index can be specified here to reorder the data columns and match the corresponding coefficient value to the respective data column. See the "Use model coefficients for prediction" section in examples.

`conf.level` The confidence level for calculating confidence/prediction intervals. Default is 0.95.

`interval` Type of interval to calculate:

- **"none" (default)** No interval to be calculated.
- **"confidence"** Calculate a confidence interval.
- **"prediction"** Calculate a prediction interval.

## Value

A data-frame with the following columns and any additional columns specified in ‘add_var‘ parameter

**.x** The x component of the x-y projection of the simplex point.

**.y** The y component of the x-y projection of the simplex point.

**.P1** The first variable whose proportion is varied across the simplex.

**.P2** The second variable whose proportion is varied across the simplex.

**.P3** The third variable whose proportion is varied across the simplex.

**.add_str_ID** An identifier column for grouping the cartesian product of all additional columns specified in ‘add_var‘ parameter (if ‘add_var‘ is specified).

**.Pred** The predicted response for each observation (if ‘prediction‘ is `TRUE`).

**.Lower** The lower limit of the prediction/confidence interval for each observation.

**.Upper** The upper limit of the prediction/confidence interval for each observation.

## Examples

```
library(DImodels)
library(dplyr)

## Load data
data(sim0)

## Fit model
mod <- lm(response ~ 0 + (p1 + p2 + p3)^2, data = sim0)

## Prepare data for creating a contour map of predicted response over
## the ternary surface
## Remember to specify prop with the same character values as the names
## of the variables in the model containing the prop.
plot_data <- ternary_data(resolution = 1, model = mod,
                          prop = c("p1", "p2", "p3"))
## Show plot
ternary_plot(data = plot_data)

## Can also add any additional variables independent of the simplex using
## the `add_var` argument
sim0$treatment <-  rep(c("A", "B", "C", "D"), each = 16)
new_mod <- update(mod, ~. + treatment, data = sim0)
plot_data <- ternary_data(prop = c("p1", "p2", "p3"),
                          add_var = list("treatment" = c("A", "B")),
                          resolution = 1, model = new_mod)
## Plot to compare between additional variables

ternary_plot(plot_data)


## It could be desirable to take the output of this function and add
## additional variables to the data before making predictions
## Use `prediction = FALSE` to get data without any predictions
contour_data <- ternary_data(prop = c("p1", "p2", "p3"),
                             model = mod,
                             prediction = FALSE,
                             resolution = 1)
head(contour_data)

## Manually add the treatment variable
contour_data$treatment <- "A"
## Make predictions
head(add_prediction(data = contour_data, model = new_mod))

## Manually add the interaction terms
contour_data <- contour_data %>%
                  mutate(`p1:p2` = p1*p2,
                         `p2:p3` = p2*p3,
                         `p1:p3` = p1*p3)

## Add predictions using model coefficients
```

```
contour_data <- add_prediction(data = contour_data,
                               coefficient = mod$coefficient)
head(contour_data)

## Note: Add predictions via coefficients would not give confidence intervals
## to get CIs using coefficients we need to specify the variance-covariance
## matrix using `vcov`
contour_data <- add_prediction(data = contour_data,
                               coefficient = mod$coefficient,
                               vcov = vcov(mod),
                               interval = "confidence")
head(contour_data)
## Show plot

ternary_plot(contour_data)

## See `?ternary_plot` for options to customise the ternary_plot
```

---

## ternary_plot *Ternary diagrams*

---

### Description

Create a ternary diagram showing the a scatter-plot of points across the surface or a contour map showing the change in a continuous variable across the ternary surface. The ternary surface can be created using the `ternary_data` function.

### Usage

```
ternary_plot(
  data,
  prop = NULL,
  col_var = ".Pred",
  show = c("contours", "points"),
  tern_labels = c("P1", "P2", "P3"),
  show_axis_labels = TRUE,
  show_axis_guides = FALSE,
  axis_label_size = 4,
  vertex_label_size = 5,
  points_size = 2,
  nlevels = 7,
  colours = NULL,
  lower_lim = NULL,
  upper_lim = NULL,
  contour_text = TRUE,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| `data` | A data-frame consisting of the x-y plane projection of the 2-d simplex. This data could be the output of the ‘ternary_data‘ function, and contain the predicted response at each point along the simplex to show the variation in response as a contour map. |
| `prop` | A character vector specifying the columns names of compositional variables. By default, the function will try to automatically interpret these values from the data. |
| `col_var` | The column name containing the variable to be used for colouring the contours or points. The default is ".Pred". |
| `show` | A character string indicating whether to show data-points or contours on the ternary. The default is to show "contours". |
| `tern_labels` | A character vector containing the labels of the vertices of the ternary. The default is the column names of the first three columns of the data, with the first column corresponding to the top vertex, second column corresponding to the left vertex and the third column corresponding to the right vertex of the ternary. |
| `show_axis_labels` | A boolean value indicating whether to show axis labels along the edges of the ternary. The default is `TRUE`. |
| `show_axis_guides` | A boolean value indicating whether to show axis guides within the interior of the ternary. The default is `FALSE`. |
| `axis_label_size` | A numeric value to adjust the size of the axis labels in the ternary plot. The default size is 4. |
| `vertex_label_size` | A numeric value to adjust the size of the vertex labels in the ternary plot. The default size is 5. |
| `points_size` | If showing points, then a numeric value specifying the size of the points. |
| `nlevels` | The number of levels to show on the contour map. |
| `colours` | A character vector or function specifying the colours for the contour map or points. The number of colours should be same as ‘nlevels‘ if (‘show = "contours"‘).<br>The default colours scheme is the `terrain.colors()` for continuous variables and an extended version of the Okabe-Ito colour scale for categorical variables. |
| `lower_lim` | A number to set a custom lower limit for the contour (if ‘show = "contours"‘). The default is minimum of the prediction. |
| `upper_lim` | A number to set a custom upper limit for the contour (if ‘show = "contours"‘). The default is maximum of the prediction. |
| `contour_text` | A boolean value indicating whether to include labels on the contour lines showing their values (if ‘show = "contours"‘). The default is `TRUE`. |
| `nrow` | Number of rows in which to arrange the final plot (when ‘add_var‘ is specified). |
| `ncol` | Number of columns in which to arrange the final plot (when ‘add_var‘ is specified). |

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if ‘plot = FALSE‘)

## Examples

```
library(DImodels)
library(dplyr)
library(ggplot2)

## Load data
data(sim0)

### Show raw data as points in ternary
## `ternary_plot` shows contours by default, use `show = "points"` to show
## points across the ternary
ternary_plot(data = sim0, prop = c("p1", "p2", "p3"), show = "points")

## The points can also be coloured using an additional variable by
## specifying it in `col_var`
ternary_plot(data = sim0, prop = c("p1", "p2", "p3"),
             col_var = "response", show = "points")

## Categorical variables can also be shown
## Also show axis guides using `show_axis_guides`
sim0$richness <- as.factor(sim0$richness)
ternary_plot(data = sim0, prop = c("p1", "p2", "p3"),
             col_var = "richness", show = "points",
             show_axis_guides = TRUE)

## Change colours by using `colours` argument
## and increase points size using `points_size`
ternary_plot(data = sim0, prop = c("p1", "p2", "p3"),
             col_var = "richness", show = "points",
             colours = c("tomato", "steelblue", "orange"),
             points_size = 4)

### Show contours of response
## Fit model
mod <- lm(response ~ 0 + (p1 + p2 + p3)^2, data = sim0)

## Create a contour map of predicted response over the ternary surface
## Remember to specify prop with the same character values as the names
## of the variables in the model containing the prop.
plot_data <- ternary_data(resolution = 1, model = mod,
                          prop = c("p1", "p2", "p3"))

## Create a contour plot of response across the ternary space
ternary_plot(plot_data)

## Change colour scheme
cols <- hcl.colors(7) # because there are 7 contour levels by default
ternary_plot(plot_data, colours = cols)
```

```
## Change number of contours using `nlevels`
## and set custom upper and lower limits for the scale
ternary_plot(plot_data, nlevels = 10, colours = hcl.colors(10),
             lower_lim = 10, upper_lim = 35)

## Change ternary labels along with their font-size
ternary_plot(plot_data, tern_labels = c("Sp1", "Sp2", "Sp3"),
             vertex_label_size = 6, axis_label_size = 5)

## Add additional variables and create a separate plot for each
sim0$treatment <-  rep(c("A", "B", "C", "D"), each = 16)
new_mod <- update(mod, ~. + treatment, data = sim0)
tern_data <- ternary_data(resolution = 1, model = new_mod,
                          prop = c("p1", "p2", "p3"),
                          add_var = list("treatment" = c("A", "C")))
## Arrange plot in 2 columns
ternary_plot(data = tern_data, ncol = 2)
```

## theme_DI *Default theme for DImodelsVis*

### Description

Default theme for DImodelsVis

### Usage

```
theme_DI(
  font_size = 14,
  font_family = "",
  legend = c("top", "bottom", "left", "right", "none")
)
```

### Arguments

| | |
|---|---|
| `font_size` | Base font size for text across the plot |
| `font_family` | Font family for text across the plot |
| `legend` | One of c("top", "bottom", "left", "right", "none") specifying the position of the legend. The legend position can also be specified as a numeric vector of form c(x, y) with x and y having values between 0 and 1. If specified as a numeric vector the legend within the plotting region where c(0,0) corresponds to the "bottom left" and c(1,1) corresponds to the "top right" position. The default position is "top". |

## Value

A ggplot theme object

## Examples

```
library(ggplot2)

plot_data <- mtcars
plot_data$gear <- as.factor(plot_data$gear)
ggplot(data = plot_data,
       aes(x = mpg, y = disp, colour = gear))+
   geom_point(size = 3)+
   facet_wrap(~cyl) +
   theme_DI()
```

---

## `visualise_effects` *Effects plot for compositional data*

---

## Description

This function will prepare the underlying data and plot the results for visualising the effect of increasing or decreasing the proportion of a predictor variable (from a set of compositional variables). The generated plot will show a curve for each observation (whenever possible) in the data. `Pie-glyphs` are used to highlight the compositions of the specified communities and the ending community after the variable interest either completes dominates the community (when looking at the effect of increase) or completely vanishes from the community (when looking at the effect of decrease) or both. This is a wrapper function specifically for statistical models fit using the `DI()` function from the `DImodels` R package and would implicitly call `visualise_effects_data` followed by `visualise_effects_plot`. If your model object isn't fit using DImodels, users can call the data and plot functions manually, one by one.

## Usage

```
visualise_effects(
  model,
  data = NULL,
  var_interest = NULL,
  effect = c("increase", "decrease", "both"),
  add_var = list(),
  interval = c("confidence", "prediction", "none"),
  conf.level = 0.95,
  se = FALSE,
  average = TRUE,
  pie_colours = NULL,
  pie_radius = 0.3,
  FG = NULL,
  plot = TRUE,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| model | A Diversity Interactions model object fit by using the ‘DI()‘ function from the ‘DImodels‘ package. |
| data | A dataframe specifying communities of interest for which user wants visualise the effect of species decrease or increase. If left blank, the communities from the original data used to fit the model would be selected. |
| var_interest | A character vector specifying the variable for which to visualise the effect of change on the response. If left blank, all variables would be assumed to be of interest. |
| effect | One of "increase", "decrease" or "both" to indicate whether to look at the effect of increasing the proportion, decreasing the proportion or doing both simultaneously, respectively on the response. The default in "increasing". |
| add_var | A list specifying values for additional variables in the model other than the proportions (i.e. not part of the simplex design). This would be useful to compare the predictions across different values for a categorical variable. One plot will be generated for each unique combination of values specified here. |
| interval | Type of interval to calculate: **"none"** No interval to be calculated. **"confidence" (default)** Calculate a confidence interval. **"prediction"** Calculate a prediction interval. |
| conf.level | The confidence level for calculating confidence/prediction intervals. Default is 0.95. |
| se | A boolean variable indicating whether to plot confidence intervals associated with the effect of species increase or decrease |
| average | A boolean value indicating whether to add a line describing the "average" effect of variable increase or decrease. The average is calculated at the median value of any variables not specified. |
| pie_colours | A character vector indicating the colours for the slices in the pie-glyphs. If left NULL, the colour blind friendly colours will be for the pie-glyph slices. |
| pie_radius | A numeric value specifying the radius (in cm) for the pie-glyphs. Default is 0.3 cm. |
| FG | A higher level grouping for the compositional variables in the data. Variables belonging to the same group will be assigned with different shades of the same colour. The user can manually specify a character vector giving the group each variable belongs to. If left empty the function will try to get a grouping from the original DI model object. |
| plot | A boolean variable indicating whether to create the plot or return the prepared data instead. The default ‘TRUE‘ creates the plot while ‘FALSE‘ would return the prepared data for plotting. Could be useful for if user wants to modify the data first and then call the plotting function manually. |

| | |
|---|---|
| `nrow` | Number of rows in which to arrange the final plot (when ‘add_var‘ is specified). |
| `ncol` | Number of columns in which to arrange the final plot (when ‘add_var‘ is specified). |

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if ‘plot = FALSE‘)

## Examples

```
library(DImodels)

## Load data
data(sim1)

## Fit model
mod <- DI(prop = 3:6, DImodel = "AV", data = sim1, y = "response")

## Get effects plot for all species in design
visualise_effects(model = mod)

## Choose a variable of interest using `var_interest`
visualise_effects(model = mod, var_interest = c("p1", "p3"))

## Add custom communities to plot instead of design communities
## Any variable not specified will be assumed to be 0
## Not showing the average curve using `average = FALSE`
visualise_effects(model = mod, average = FALSE,
                  data = data.frame("p1" = c(0.7, 0.1),
                                    "p2" = c(0.3, 0.5),
                                    "p3" = c(0,   0.4)),
                  var_interest = c("p2", "p3"))

## Add uncertainty on plot
visualise_effects(model = mod, average = TRUE,
                  data = data.frame("p1" = c(0.7, 0.1),
                                    "p2" = c(0.3, 0.5),
                                    "p3" = c(0,   0.4)),
                  var_interest = c("p2", "p3"), se = TRUE)

## Visualise effect of species decrease for particular species
## Show a 99% confidence interval using `conf.level`
visualise_effects(model = mod, effect = "decrease",
                  average = TRUE, se = TRUE, conf.level = 0.99,
                  data = data.frame("p1" = c(0.7, 0.1),
                                    "p2" = c(0.3, 0.5),
                                    "p3" = c(0,   0.4),
                                    "p4" = 0),
                  var_interest = c("p1", "p3"))

## Show effects of both increase and decrease using `effect = "both"`
## and change colours of pie-glyphs using `pie_colours`
```

```
visualise_effects(model = mod, effect = "both",
                  average = FALSE,
                  pie_colours = c("steelblue1", "steelblue4", "orange1", "orange4"),
                  data = data.frame("p1" = c(0.7, 0.1),
                                    "p2" = c(0.3, 0.5),
                                    "p3" = c(0,   0.4),
                                    "p4" = 0),
                  var_interest = c("p1", "p3"))

# Add additional variables and create a separate plot for each

visualise_effects(model = mod, effect = "both",
                  average = FALSE,
                  pie_colours = c("steelblue1", "steelblue4", "orange1", "orange4"),
                  data = data.frame("p1" = c(0.7, 0.1),
                                    "p2" = c(0.3, 0.5),
                                    "p3" = c(0,   0.4),
                                    "p4" = 0),
                  var_interest = c("p1", "p3"),
                  add_var = list("block" = factor(c(1, 2),
                                                  levels = c(1, 2, 3, 4))))


## Specify `plot = FALSE` to not create the plot but return the prepared data
head(visualise_effects(model = mod, effect = "both",
                       average = FALSE, plot = FALSE,
                       pie_colours = c("steelblue1", "steelblue4",
                                       "orange1", "orange4"),
                       data = data.frame("p1" = c(0.7, 0.1),
                                         "p2" = c(0.3, 0.5),
                                         "p3" = c(0,   0.4),
                                         "p4" = 0),
                       var_interest = c("p1", "p3")))
```

## visualise_effects_data

*Prepare data for effects plots for compositional data*

### Description

The helper function to create the underlying data for visualising the effect of increasing or decreasing (or both) the proportion of a variable from a set of compositional variables. This is a special case of the `simplex_path` function where the end points are either the monoculture (i.e. variable of interest = 1, while all others equal 0) of the variable of interest (when increasing the proportion) or a community without the variable of interest (when decreasing the proportion). The observations specified in ‘data‘ are connected to the respective communities (monoculture of the variable of interest or the community without the variable of interest) by a straight line across the simplex; This has the effect of changing the proportion of the variable of interest whilst adjusting the proportion of the other variables but keeping the ratio of their relative proportions unchanged, thereby preserving the compositional nature of the data. See examples for more information. The output of this function can be passed to the `visualise_effects_plot` function to visualise the results.

### Usage

```
visualise_effects_data(
  data,
  prop,
  var_interest = NULL,
  effect = c("increase", "decrease", "both"),
  add_var = list(),
  prediction = TRUE,
  ...
)
```

### Arguments

| | |
|---|---|
| `data` | A dataframe specifying the initial communities of interest for which to visualise the effect of increasing/decreasing a variable. If a model object is specified then this data should contain all the variables present in the model object including any additional variables not part of the simplex design. If a coefficient vector is specified then data should contain same number of columns as the number of elements in the coefficient vector and a one-to-one positional mapping would be assumed between the data columns and the elements of the coefficient vector. |
| `prop` | A vector of column names or indices identifying the columns containing the variable proportions (i.e., compositional columns) in the data. |
| `var_interest` | A character vector specifying the variable for which to visualise the effect of change on the response. If left blank, all variables would be assumed to be of interest. |
| `effect` | One of "increase", "decrease" or "both" to indicate whether to look at the effect of increasing the proportion, decreasing the proportion or doing both simultaneously, respectively on the response. The default in "increasing". |
| `add_var` | A list specifying values for additional variables in the model other than the proportions (i.e. not part of the simplex design). This would be useful to compare the predictions across different values for a categorical variable. One plot will be generated for each unique combination of values specified here. |
| `prediction` | A logical value indicating whether to pass the final data to ‘add_prediction‘ and add predictions to the data. Default value is `TRUE`, but often it would be desirable to make additional changes to the data before making any predictions, so the user can set this to `FALSE` and manually call the ‘add_prediction‘ function. |
| `...` | Arguments passed on to `add_prediction`<br>`model` A regression model object which will be used to make predictions for the observations in ‘data‘. Will override ‘coefficients‘ if specified.<br>`coefficients` If a regression model is not available (or can’t be fit in R), the regression coefficients from a model fit in some other language can be used to calculate predictions. However, the user would have to ensure there’s an |

appropriate one-to-one positional mapping between the data columns and the coefficient values. Further, they would also have to provide a variance-covariance matrix of the coefficients in the ‘vcov‘ parameter if they want the associated CI for the prediction or it would not be possible to calculate confidence/prediction intervals using this method.

`vcov` If regression coefficients are specified, then the variance-covariance matrix of the coefficients can be specified here to calculate the associated confidence interval around each prediction. Failure to do so would result in no confidence intervals being returned. Ensure ‘coefficients‘ and ‘vcov‘ have the same positional mapping with the data.

`coeff_cols` If ‘coefficients‘ are specified and a one-to-one positional mapping between the data-columns and coefficient vector is not present. A character string or numeric index can be specified here to reorder the data columns and match the corresponding coefficient value to the respective data column. See the "Use model coefficients for prediction" section in examples.

`conf.level` The confidence level for calculating confidence/prediction intervals. Default is 0.95.

`interval` Type of interval to calculate:

**"none" (default)** No interval to be calculated.

**"confidence"** Calculate a confidence interval.

**"prediction"** Calculate a prediction interval.

## Value

A data frame with the following columns appended at the end

**.Sp** An identifier column to discern the variable of interest being modified in each curve.

**.Proportion** The value of the variable of interest within the community.

**.Group** An identifier column to discern between the different curves.

**.add_str_ID** An identifier column for grouping the cartesian product of all additional columns specified in ‘add_var‘ parameter (if ‘add_var‘ is specified).

**.Pred** The predicted response for each observation.

**.Lower** The lower limit of the prediction/confidence interval for each observation.

**.Upper** The upper limit of the prediction/confidence interval for each observation.

**.Marginal** The marginal change in the response (first derivative) with respect to the gradual change in the proportion of the species of interest.

**.Threshold** A numeric value indicating the maximum proportion of the species of interest within a particular community which has a positive marginal effect on the response.

**.MarEffect** A character string entailing whether the increase/decrease of the species of interest from the particular community would result in a positive or negative marginal effect on the response.

**.Effect** An identifier column signifying whether considering the effect of species addition or species decrease.

## Examples

```
library(DImodels)

## Load data
data(sim1)

## Fit model
mod <- glm(response ~ p1 + p2 + p3 + p4 + 0, data = sim1)

## Create data for visualising effect of increasing the proportion of
## variable p1 in data
## Notice how the proportion of `p1` increases while the proportion of
## the other variables decreases whilst maintaining their relative proportions
head(visualise_effects_data(data = sim1, prop = c("p1", "p2", "p3", "p4"),
                            var_interest = "p1", effect = "increase",
                            model = mod))

## Create data for visualising the effect of decreasing the proportion
## variable p1 in data using `effect = "decrease"`
head(visualise_effects_data(data = sim1, prop = c("p1", "p2", "p3", "p4"),
                            var_interest = "p1", effect = "decrease",
                            model = mod))

## Create data for visualising the effect of increasing and decreasing the
## proportion variable p3 in data using `effect = "both"`
head(visualise_effects_data(data = sim1, prop = c("p1", "p2", "p3", "p4"),
                            var_interest = "p3", effect = "decrease",
                            model = mod))

## Getting prediction intervals at a 99% confidence level
head(visualise_effects_data(data = sim1, prop = c("p1", "p2", "p3", "p4"),
                            var_interest = "p1", effect = "decrease",
                            model = mod, conf.level = 0.99,
                            interval = "prediction"))

## Adding additional variables to the data using `add_var`
## Notice the new .add_str_ID column in the output
sim1$block <- as.numeric(sim1$block)
new_mod <- update(mod, ~ . + block, data = sim1)
head(visualise_effects_data(data = sim1[, 3:6], prop = c("p1", "p2", "p3", "p4"),
                            var_interest = "p1", effect = "both",
                            model = new_mod,
                            add_var = list("block" = c(1, 2))))

## Create data for visualising effect of decreasing variable p2 from
## the original communities in the data but using model coefficients
## When specifying coefficients the data should have a one-to-one
## positional mapping with specified coefficients.
init_comms <- sim1[, c("p1", "p2", "p3", "p4")]
head(visualise_effects_data(data = init_comms, prop = 1:4,
                            var_interest = "p2",
                            effect = "decrease",
                            interval = "none",
                            coefficients = mod$coefficients))

## Note that to get confidence interval when specifying
## model coefficients we'd also need to provide a variance covariance
## matrix using the `vcov` argument
head(visualise_effects_data(data = init_comms, prop = 1:4,
                            var_interest = "p2",
                            effect = "decrease",
                            interval = "confidence",
                            coefficients = mod$coefficients,
                            vcov = vcov(mod)))

## Can also create only the intermediary communities without predictions
## by specifying prediction = FALSE.
## Any additional columns can then be added and the `add_prediction` function
## can be manually called.
## Note: If calling the `add_prediction` function manually, the data would
## not contain information about the marginal effect of changing the species
## interest
effects_data <- visualise_effects_data(data = init_comms, prop = 1:4,
                                       var_interest = "p2",
                                       effect = "decrease",
                                       prediction = FALSE)
head(effects_data)
## Prediction using model object
head(add_prediction(data = effects_data, model = mod, interval = "prediction"))
## Prediction using regression coefficients
head(add_prediction(data = effects_data, coefficients = mod$coefficients))
```

---

## visualise_effects_plot *Effects plot for compositional data*

---

### Description

The plotting function to create plots showing the effect of increasing or decreasing the proportion of a variable from a set of compositional variables. The output of the ‘visualise_effects_data‘ function (with any desired modifications) should be passed here. The generated plot will show a curve for each observation (whenever possible) in the data. ‘Pie-glyphs‘ are used to highlight the compositions of the specified communities and the ending community after the variable of interest either completes dominates the community (when looking at the effect of increase) or completely vanishes from the community (when looking at the effect of decrease) or both.

### Usage

```
visualise_effects_plot(
  data,
  prop,
  pie_colours = NULL,
  pie_radius = 0.3,
  se = FALSE,
  average = TRUE,
  nrow = 0,
  ncol = 0
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame created using the `visualise_effects_data` function. |
| `prop` | A vector of column names or indices identifying the columns containing the compositional variables in the data. Will be inferred from the data if it is created using the ‘`visualise_effects_data`‘ function, but the user also has the flexibility of manually specifying the values. |
| `pie_colours` | A character vector indicating the colours for the slices in the pie-glyphs. If left NULL, the colour blind friendly colours will be for the pie-glyph slices. |
| `pie_radius` | A numeric value specifying the radius (in cm) for the pie-glyphs. Default is 0.3 cm. |
| `se` | A boolean variable indicating whether to plot confidence intervals associated with the effect of species increase or decrease |
| `average` | A boolean value indicating whether to add a line describing the "average" effect of variable increase or decrease. The average is calculated at the median value of any variables not specified. |
| `nrow` | Number of rows in which to arrange the final plot (when ‘add_var‘ is specified). |
| `ncol` | Number of columns in which to arrange the final plot (when ‘add_var‘ is specified). |

## Value

A ggmultiplot (ggplot if single plot is returned) class object or data-frame (if ‘plot = FALSE‘)

## Examples

```
library(DImodels)

## Load data
data(sim1)

## Fit model
mod <- glm(response ~ p1 + p2 + p3 + p4 + 0, data = sim1)

## Create data for visualising effect of adding species 1 to
## the original communities in the data
plot_data <- visualise_effects_data(data = sim1[sim1$block == 1, ],
                                    prop = c("p1", "p2", "p3", "p4"),
                                    var_interest = "p1",
                                    effect = "increase", model = mod)
```

```
## Create plot
visualise_effects_plot(data = plot_data)

## Show specific curves with prediction intervals
subset <- custom_filter(plot_data, .Group %in% c(7, 15))
visualise_effects_plot(data = subset, prop = 1:4, se = TRUE)

## Do not show average effect line
visualise_effects_plot(data = subset,
                       se = TRUE, average = FALSE)

## Change colours of the pie-glyph slices
visualise_effects_plot(data = subset,
                       pie_colours = c("darkolivegreen", "darkolivegreen1",
                                   "steelblue4", "steelblue1"))

#' ## Simultaneously create multiple plots for additional variables
sim1$block <- as.numeric(sim1$block)
new_mod <- update(mod, ~ . + block, data = sim1)
plot_data <- visualise_effects_data(data = sim1[c(1, 5, 9, 13), 3:6],
                                    prop = c("p1", "p2", "p3", "p4"),
                                    var_interest = "p3",
                                    model = new_mod, conf.level = 0.95,
                                    add_var = list("block" = c(1, 2)))

visualise_effects_plot(data = plot_data,
                       average = FALSE,
                       pie_colours = c("darkolivegreen", "darkolivegreen1",
                                       "steelblue4", "steelblue1"))
```

# Index