# Package 'conjurer'

May 8, 2026

**Type** Package

**Title** A Parametric Method for Generating Synthetic Data

**Version** 1.7.1

**Date** 2023-01-15

**Description** Generates synthetic data distributions to enable testing various modelling techniques in ways that real data does not allow. Noise can be added in a controlled manner such that the data seems real. This methodology is generic and therefore benefits both the academic and industrial research.

**Depends** R (>= 2.10)

**Imports** jsonlite(>= 1.8.0), httr (>= 1.4.2), methods

**License** MIT + file LICENSE

**URL** https://www.foyi.co.nz/posts/documentation/documentationconjurer/

**BugReports** https://github.com/SidharthMacherla/conjurer/issues

**Encoding** UTF-8

**RoxygenNote** 7.2.3

**Suggests** knitr, rmarkdown

**VignetteBuilder** knitr

**NeedsCompilation** no

**Author** Sidharth Macherla [aut, cre] (ORCID:
<https://orcid.org/0000-0002-4825-2026>)

**Maintainer** Sidharth Macherla <msidharthrasik@gmail.com>

**Repository** CRAN

**Date/Publication** 2023-01-18 08:30:06 UTC

## Contents

---

`buildCust` *Build a Unique Customer Identifier*

---

## Description

Builds a customer identifier. This is often used as a primary key of the customer dim table in databases.

## Usage

```
buildCust(numOfCust)
```

## Arguments

| | |
|---|---|
| `numOfCust` | A natural number. This specifies the number of unique customer identifiers to be built. |

## Details

A customer is identified by a unique customer identifier(ID). A customer ID is alphanumeric with prefix "cust" followed by a numeric. This numeric ranges from 1 and extend to the number of customers provided as the argument within the function. For example, if there are 100 customers, then the customer ID will range from cust001 to cust100. This ensures that the customer ID is always of the same length.

## Value

A character with unique customer identifiers

## Examples

```
df <- buildCust(numOfCust = 1000)
df <- buildCust(numOfCust = 223)
```

---

`buildDistr` *Build Data Distribution*

---

## Description

Builds data distribution. For example, the function `genTrans` uses this function to build the data distributions necessary. This function uses trigonometry based functions to generate data. This is an internal function and is currently not exported in the package.

## Usage

```
buildDistr(st, en, cycles, trend, n)
```

## Arguments

| | |
|---|---|
| `st` | A number. This defines the starting value of the number of data points. |
| `en` | A number. This defines the ending value of the number of data points. |
| `cycles` | A string. This defines the cyclicality of data distribution. |
| `trend` | A number. This defines the trend of data distribution i.e if the data has a positive slope or a negative slope. |
| `n` | A numeric. This specifies the number of values to be generated. It should be non-zero natural number. This parameter is currently used by the function `buildNum`. |

## Details

A parametric method is used to build data distribution. The data distribution function uses the formulation of

$$sin(a * x) + cos(b * x) + c$$

Where,

1. a and b are the parameters
2. x is a variable
3. c is a constant

Firstly, parameter 'a' defines the number of outer level crests (peaks in the data distribution). Generally speaking, the number of crests is approximately twice the value of a. This means that if a is set to a value 0.5, there will be one crest and if it is set to 2, there will be 4 crests. On account of this behavior, this parameter is set based on the argument cycles of the function. For example, if the argument cycles is set to "y" i.e yearly cycle, it means that there must be one crest i.e peak in the distribution. To have one crest, the parameter must be around 0.5. A random number is then generated between 0.2 and 0.6 to get to that one crest.

Secondly, the variable 'x' is the x-axis of the data distribution. Since the function `buildDistr` is used internally to generate data at different levels, this variable could have a range of 1 to 12 or 1 to 31 depending on the arguments 'st' and 'en'. For example, if the data is generated at the month level, then arguments 'st' is set to 1 and 'en' is set to 12. Similarly, if the data is set to day level, the 'st' is set to 1 and 'en' is set to the number of days in that month i.e 28 for month 2 and 31 for month 12 etc.

Thirdly, the parameter 'b' defines the inner level crests(peaks in data distribution). This parameter helps in making the data distribution seem more realistic by adding more "ruggedness" of the distribution.

Finally, the constant 'c' is the intercept part of the formulation and primarily serves as a way to ensure that the data distribution has a positive 'y' axis component. This value is randomly generated between 2 and 5.

## Value

A data frame with data distribution is returned.

---

`buildHierarchy` *Generate hierarchical data*

---

## Description

Generates hierarchical data by using an internal function `genTree`. For a working example, please see the vignette.

## Usage

```
buildHierarchy(type, splits, numOfLevels)
```

## Arguments

| | |
|---|---|
| `type` | A string. In its current state, this is only a placeholder function and is not mandatory. Currently, only one type of hierarchy is permitted namely $equalSplit$. |
| `splits` | A positive number. This specifies the number of splits at each branch. For instance, if $split$ is 2 then, each branch will have 2 sub-branches. |
| `numOfLevels` | A positive number. This specifies the number of layers in the hierarchy. |

### Details

This function helps in generating hierarchical data. If there are multiple categorical variables i.e. classes that are mapped to other classes in a hierarchical manner, this function helps in building the same. Some common use cases for this type of data are Linnaean system of classification in life sciences and product hierarchy in retail industry. The number of terminal nodes are dependent on the arguments $splits$ and $numOfLevels$. More precisely, the number of terminal nodes has the formulation of $splits^numOfLevels$. For instance, if $splits$ is 2 and $numOfLevels$ is 3, then the number of terminal nodes are $2^3$ i.e. 8. Furthermore, the number of columns of the output dataframe is equal to the $numOfLevels$. Although a hierarchical data sctructure is often represented as a tree structure, this function outputs the data in a denormalized form i.e a dataframe.

### Value

A dataframe.

### Examples

```
productHierarchy <- buildHierarchy(type = "equalSplit", splits = 2, numOfLevels = 3)
productHierarchy <- buildHierarchy(splits = 2, numOfLevels = 3)
```

---

## buildId *Build identifier*

---

### Description

Builds strings that could be used as identifiers.

### Usage

```
buildId(numOfItems, prefix)
```

### Arguments

| | |
|---|---|
| numOfItems | A number. This defines the number of elements to be output. |
| prefix | A string. This defines the prefix for the strings. |

### Details

This function can be used to build an alphanumeric sequence that can be used as a primary key in a data table or a unique identifier of an element.

### Value

A character with the alphanumeric strings is returned. These strings use the prefix that is mentioned in the argument "prefix"

### Examples

```
userId <- buildId(numOfItems = 3, prefix = "uid")
```

`buildModelData` *Generate Synthetic Data using uncovr API*

## Description

Please refer to the official documentation of uncovr at *https://www.foyi.co.nz/posts/documentation/documentationuncovr/* for a detailed explanation. This function generates data i.e. independent variables and dependent variable. Besides these variables, this function sources the linear function i.e. model formula.This function needs to be used along with other function such as `extractDf` so as to extract relevant portions of the response.

## Usage

```
buildModelData(numOfObs, numOfVars, key, modelObj)
```

## Arguments

| | |
|---|---|
| `numOfObs` | A number. This represents the number of observations in the data. In other words, the number of rows of data that are requested to be generated. The *numOfObs* argument must be a non-negative integer and in the current version, this function accepts a range of 100 to 10,000. |
| `numOfVars` | A number. This represents the number of independent variables in the data. In other words, the number of columns besides the dependent variable of data that are requested to be generated. The *numOfVars* argument must be a non-negative integer and in the current version, this function accepts a range of 1 to 100. |
| `key` | An alpha numeric. This is the subscription key that can be sourced from the developer portal of uncovr API available at *https://foyi.developer.azure-api.net/*. |
| `modelObj` | Optional argument. A glm or lm model object where both the dependent and independent variables are continuous. |

## Details

This is a function that helps in sending the details of the requested data to uncovr API end point and source its response. The purpose of this function can be best understood when explained within the context that is given below. There is a closed source SaaS(Software as a Service) software named *uncovr* that provides an API(Application Programming Interface). In its current state, the SaaS software is free to use with some constraints around the volume of data and the frequency of API calls. One of the functions of *uncovr* API takes an input of number of observations i.e. rows and number of independent variables namely columns and gives an output. The input of the *uncovr* function is required to be sent as part of the body of the html POST functionality. This function *buildModelData* creates the json in the form required by *uncovr* API and sources the response. This function uses an internal function `uncovrApi` to connect to the API endpoint and uses another internal function namely `genIndepDepJson` to build the necessary body of the POST function.

## Value

A json with details such as the requested data, model performance metrics and the model formula.

## buildName *Build Dynamic Strings*

### Description

Builds strings that could be further used as identifiers. This is an internal function and is currently not exported in the package.

### Usage

```
buildName(numOfItems, prefix)
```

### Arguments

| | |
|---|---|
| numOfItems | A number. This defines the number of elements to be output. |
| prefix | A string. This defines the prefix for the strings. For example, the function buildCust uses this function and passes the prefix "cust" while the function buildProd passes the prefix "sku" |

### Details

This function is used by other internal functions namely, buildCust and buildProd to produce the alphanumeric identifiers for customers and products respectively.

### Value

A character with the alphanumeric strings is returned. These strings use the prefix that is mentioned in the argument "prefix"

## buildNames *Generate Names*

### Description

Generates names based on a given training data or using the default data

### Usage

```
buildNames(dframe, numOfNames, minLength, maxLength)
```

### Arguments

| | |
|---|---|
| dframe | A dataframe. This argument is passed on to another function genMatrix for generating an alphabet frequency table. This dataframe is single column dataframe with rows that contain names. These names must only contain english alphabets(upper or lower case) from A to Z. |
| numOfNames | A numeric. This specifies the number of names to be generated. It should be non-zero natural number. |
| minLength | A numeric. This specifies the minimum number of alphabets in the name. It must be a non-zero natural number. |
| maxLength | A numeric. This specifies the maximum number of alphabets in the name. It must be a non-zero natural number. |

### Details

This function generates names. There are two options to generate names. The first option is to use an existing sample of names and generate names. The second option is to use the default table of prior probabilities.

### Value

A list of names.

### Examples

```
buildNames(numOfNames = 3, minLength = 5, maxLength = 7)
```

---

## buildNum *Build Numeric Data*

---

### Description

Build Numeric Data

### Usage

```
buildNum(n, st, en, disp, outliers)
```

### Arguments

| | |
|---|---|
| n | A number. This specifies the number of values to be generated. |
| st | A number. This defines the starting value of the number of data points. |
| en | A number. This defines the ending value of the number of data points. |
| disp | A number between $-(pi/2)$ and $(pi/2)$. This defines the dispersion of the distribution. |

`outliers` A number. This signifies the presence of outliers. If set to value 1, then outliers are generated randomly. If set to value 0, then no outliers are generated. The presence of outliers is a very common occurrence and hence setting the outliers to 1 is recommended. However, there are instances where outliers are not needed. For example, if the objective of data generation is solely for visualization purposes then outliers may not be needed. The default value is 1.

## Details

This function helps in generating numeric data such as age, height, weight etc. This function could be used along with other functions such as `buildCust` to make it more meaningful. The data distribution function uses the formulation of

$$sin((r * a) * x) + c$$

Where,

1. r is the random value such that $0.8 <= r <= 1.2$. This adds $+/-$ 20% randomness to the parameter $a$.
2. a is the parameter such that, $-(pi/2) <= a <= (pi/2)$.
3. x is a variable such that, $(pi/2) <= x <= (pi/2)$.
4. c is a constant such that $2 <= c <= 5$.

The key component of this function is $disp$. This helps in controlling the dispersion of the distribution. Let us assume that one would like to generate age of people in years. Furthermore, let us assume that the range of the age is between 23 and 80. If $disp = 1$, then the function will generate more data with a negative slope i.e more people with age closer to 23 than 80. If $disp = 1$ is used, then the opposite will be true. However, if one would like to generate data that is visually similar to normal distribution i.e more people in the middle age group and less towards 23 or 80, then $disp = 0.5$ could be used.

It is recommended to firstly plot the code and inspect visually to check which distribution is needed.

## Value

A dataframe

## Examples

```
age <- buildNum(n = 10, st = 23, en = 80, disp = 0.5, outliers = 1)
plot(age) #visualize the resulting distribution
```

`buildOutliers` *Build Outliers in Data Distribution*

## Description

Builds outlier values and replaces random data points with outliers. This is an internal function and is currently not exported in the package.

## Usage

```
buildOutliers(distr)
```

## Arguments

| | |
|---|---|
| `distr` | numeric vector. This is the target vector which is processed for outlier generation. |

## Details

It is a common occurrence to have outliers in production data. For instance, in the retail industry, there are days such as black Friday where the sales for that day are far more than the daily average for the year. For the synthetic data generated to seem similar to production data, package conjurer uses this function to build such outlier data.

This function takes a numeric vector and then randomly selects at least 1 data point and a maximum of 3 percent data points to be replaced with an outlier. The process for generating outliers is as follows. This methodology of outlier generation is based on a popular method of identifying outliers. For more details refer to the function 'outlier' in R package 'GmAMisc'.

1. First, the interquartile range(IQR) of the numeric vector is computed.
2. Second, a random number between 1.5 and 3 is generated.
3. Finally, the random number above is multiplied with the IQR to compute the outlier.

   These steps mentioned above are repeated for at least once and a maximum of 3

## Value

A numeric vector with random values replaced with outlier values.

`buildPareto` *Map Factors Based on Pareto Arguments*

## Description

Maps a factor to another factor in a one to many relationship following Pareto principle. For example, 80 percent of transactions can be mapped to 20 percent of customers.

## Usage

```
buildPareto(factor1, factor2, pareto)
```

## Arguments

| | |
|---|---|
| `factor1` | A factor. This factor is mapped to factor2 as given in the details section. |
| `factor2` | A factor. This factor is mapped to factor1 as given in the details section. |
| `pareto` | This defines the percentage allocation and is a numeric data type. This argument takes the form of c(x,y) where x and y are numeric and their sum is 100. If we set Pareto to c(80,20), it then allocates 80 percent of factor1 to 20 percent of factor 2. This is based on a well-known concept of the Pareto principle. |

## Details

This function is used to map one factor to another based on the Pareto argument supplied. If factor1 is a factor of customer identifiers, factor2 is a factor of transactions and Pareto is set to c(80,20), then 80 percent of customer identifiers will be mapped to 20 percent of transactions and vice versa.

## Value

A data frame with factor 1 and factor 2 as columns. Based on the Pareto arguments passed, column factor 1 is mapped to factor 2.

`buildPattern` *Build a pattern*

## Description

Builds data based on a pattern. This function uses another internal function `genPattern`.

## Usage

```
buildPattern(n, parts, probs)
```

### Arguments

| | |
|---|---|
| n | A natural number. This specifies the number of data points to build. |
| parts | A natural number. This specifies the parts that make up the pattern. |
| probs | A number between 0 and 1. |

### Details

This function helps in generating data based on a pattern. To explain in simple terms, this function aims to perform the exact opposite of a regular expression i.e regex function. In other words, this function generates data given a generic pattern. The steps in the process of building data from a pattern is as follows.

1. Identify the parts that make up the data. Ideally, these parts have a pattern and a probabilistic distribution of their own. For example, a phone number has three parts namely, country code, area code and a number.
2. Assign probabilities to each of the above parts. If a part contains only one member, then the corresponding probability must be 1. However, if there are multiple members in the part, then each member must have a probability provided in the respective order.

### Value

A vector.

### See Also

genPattern.

### Examples

```
parts <- list(c("+91","+44","+64"), c(491,324,211), c(7821:8324))
probs <- list(c(0.25,0.25,0.50), c(0.30,0.60,0.10), c())
phoneNumbers <- buildPattern(n=20,parts = parts, probs = probs)
head(phoneNumbers)
parts <- list(c("+91","+44","+64"), c("("), c(491,324,211), c(")"), c(7821:8324))
probs <- list(c(0.25,0.25,0.50), c(1), c(0.30,0.60,0.10), c(1), c())
phoneNumbers <- buildPattern(n=20,parts = parts, probs = probs)
head(phoneNumbers)
```

---

## buildProd *Build Product Data*

---

### Description

Builds a unique product identifier and price. The price of the product is generated randomly within the minimum and the maximum range provided as input.

### Usage

```
buildProd(numOfProd, minPrice, maxPrice)
```

### Arguments

| | |
|---|---|
| numOfProd | A number. This defines the number of unique products. |
| minPrice | A number. This is the minimum value of the product's price range. |
| maxPrice | A number. This is the maximum value of the product's price range. |

### Details

A product ID is alphanumeric with prefix "sku" which signifies a stock keeping unit. This prefix is followed by a numeric ranging from 1 and extending to the number of products provided as the argument within the function. For example, if there are 10 products, then the product ID will range from sku01 to sku10. This ensures that the product ID is always of the same length. For these product IDs, the product price will be within the range of minPrice and maxPrice arguments.

### Value

A character with product identifier and price.

### Examples

```
df <- buildProd(numOfProd = 1000, minPrice = 5, maxPrice = 100)
df <- buildProd(numOfProd = 29, minPrice = 3, maxPrice = 50)
```

---

## buildSpike *Build Spikes in the Data Distribution*

---

### Description

Builds spikes in the data distribution. For example, in retail industry transactions are generally higher during the holiday season such as December. This function is used to set the same.

### Usage

```
buildSpike(distr, spike)
```

### Arguments

| | |
|---|---|
| distr | numeric vector. This is the input vector for which the spike value needs to be set. |
| spike | A number. This represents the seasonality of data. It can take any value from 1 to 12. These numbers represent months in a year, from January to December respectively. For example, if the spike is set to 12, it means that December has the highest number of transactions. This is an internal function and is currently not exported in the package. |

## Value

A numeric vector reordered

---

`extractDf` *Extract Dataframe from uncovr API Response*

---

## Description

This function extracts the dataframe from the output of the `buildModelData` function. Please refer to the official documentation of uncovr at *https://www.foyi.co.nz/posts/documentation/documentationuncovr/*.

## Usage

```
extractDf(uncovrJson)
```

## Arguments

| | |
|---|---|
| `uncovrJson` | A json. This is the output of the `buildModelData` function. |

## Details

The purpose of this function can be best understood when explained within the context that is given below. There is a closed source SaaS(Software as a Service) software named *uncovr* that provides an API(Application Programming Interface). In its current state, the SaaS software is free to use with some constraints around the volume of data and the frequency of API calls. One of the functions of *uncovr* API takes an input of number of observations i.e. rows and number of independent variables namely columns and gives an output. This output is in the form of a json file and has many other elements besides the dependent and independent variables. This function *extractDf* helps in extracting the dataframe from the json.

## Value

A dataframe with dependent and independent variables. The independent variables are prefixed with *iv* and the dependent variable is named *dv*.

---

`genFirstPairs` *Extracts the First Two Alphabets of the String*

---

## Description

For a given string, this function extracts the first two alphabets. This function is further used by `genMatrix` function.

## Usage

```
genFirstPairs(s)
```

### Arguments

| | |
|---|---|
| s | A string. This is the string from which the first two alphabets are to be extracted. |

### Value

First two alphabets of the string input.

---

## genIndepDepJson *Generate Body for the POST Function of Uncovr*

---

### Description

This is an internal function used by `buildModelData` function.

### Usage

```
genIndepDepJson(numOfObs, numOfVars, modelObj)
```

### Arguments

| | |
|---|---|
| numOfObs | A number. This represents the number of observations in the data. In other words, the number of rows of data that are requested to be generated. The *numOfObs* argument must be a non-negative integer. |
| numOfVars | A number. This represents the number of variables in the data. In other words, the number of columns of data that are requested to be generated. The *numOfVars* argument must be a non-negative integer. |
| modelObj | An optional argument. An lm or glm model object. The current limitation is that the independent and dependent variables must be continuous. |

### Details

This function is one of the core functions for the generation of data that comprises of independent and dependent variables. The purpose of this function can be best understood when explained within the context that is given below. There is a proprietary SaaS(Software as a Service) software named *uncovr* that provides an API(Application Programming Interface). In its current state, the SaaS software is free to use with some constraints around the volume of data and the frequency of API calls. One of the functions of *uncovr* API takes is to source inputs such as number of observations i.e. rows and number of independent variables namely columns and gives an output. The input of the *uncovr* function is required to be sent as part of the body of the html POST functionality. This function *genIndepDepJson* creates the json in the form required by *uncovr* API. As an optional argument, an lm or glm model object can be passed using the *modelObj* argument. This will ensure that the coefficients of the independent variables are sourced from the model object instead of generating randomly by the *uncovr* API. The current limitation is that the independent and dependent variables must be continuous.

## Value

A json with the details of independent variable and the dependent variable. The format of this json is as required by the *uncovr* api end point.

---

`genMatrix` *Generate Frequency Distribution Matrix*

---

## Description

For a given names dataframe and placement, a frequency distribution table is returned.

## Usage

```
genMatrix(dframe, placement)
```

## Arguments

| | |
|---|---|
| `dframe` | A dataframe with one column that has one name per row. These names must be english alphabets from A to Z and must not include any non-alphabet characters such as as hyphen or apostrophe. |
| `placement` | A string argument that takes three values namely "first", "last" and "all". Currently, only "first" and "all" are used while the option "last" is a placeholder for future versions of the package **conjurer** |

## Details

The purpose of this function is to generate a frequency distribution table of alphabets. There are currently 2 tables that could be generated using this function. The first table is generated using the internal function `genFirstPairs`. For this, the argument *placement* is assigned the value "first". The rows of the table returned by the function represent the first alphabet of the string and the columns represent the second alphabet. The values in the table represent the number of times the combination is observed i.e the combination of the row and column alphabets.

The second table is generated using the internal function `genTriples`. For this, the argument *placement* is assigned the value "all". The rows of the table returned by the function represent two consecutive alphabets of the string and the columns represent the third consecutive alphabet. The values in the table represent the number of times the combination is observed i.e the combination of the row and column alphabets.

## Value

A table. The rows and columns of the table depend on the argument *placement*. A detailed explanation is as given below in the detail section.

genPattern *Generate a pattern*

## Description

Generates data based on a pattern. This function is used by another internal function buildPattern.

## Usage

```
genPattern(orderedList)
```

## Arguments

| | |
|---|---|
| orderedList | A list of lists. The element $values$ of the sublist is a vector of characters(string or numeric or special character) and the element $probs$ is a vector of probabilities. The range of the probs is 0 to 1 and length of the $probs$ vector is either equal to length of $values$ or NULL. |

## Details

This function helps in generating data based on a pattern. To explain in simple terms, this function aims to perform the exact opposite of a regular expression i.e regex function. In other words, this function generates data given a generic pattern. The input is a list of components that make up the pattern. Each component i.e element of the list is a also list with two vectors namely $values$ and $probs$. The vector $values$ has the set of values out of which one of them is selected randomly. If this random selection is supposed to be completely random, then the next vector $probs$ can be left empty i.e. NULL. However, if the random selection of values is expected to follow a a pre-determined probabilistic distribution, then the probabilities must be provided explicitly. To explain further, if there are three values $a$, $b$, $c$ and their probabilistic distribution must be 25 percent, 50 percent and 25 percent respectively, then the vector $values$ will take the form $c(a, b, c)$ and the vector $probs$ will take the form $c(0.25, 0.5, 0.25)$.

## Value

A character vector.

## See Also

[buildPattern()]

genTrans *Build Transaction Data*

## Description

Build Transaction Data

## Usage

```
genTrans(cycles, trend, transactions, spike, outliers)
```

## Arguments

| | |
|---|---|
| cycles | This represents the cyclicality of data. It can take the following values<br>1. "y". If cycles is set to the value "y", it means that there is only one instance of a high number of transactions during the entire year. This is a very common situation for some retail clients where the highest number of sales are during the holiday period in December.<br>2. "q". If cycles is set to the value "q", it means that there are 4 instances of a high number of transactions. This is generally noticed in the financial services industry where the financial statements are revised every quarter and have an impact on the equity transactions in the secondary market.<br>3. "m". If cycles is set to the value "m", it means that there are 12 instances of a high number of transactions for a year. This means that the number of transactions increases once every month and then subside for the rest of the month. |
| trend | A number. This represents the slope of data distribution. It can take a value of 1 or -1. If the trend is set to value 1, then the aggregated monthly transactions will exhibit an upward trend from January to December and vice versa if it is set to -1. |
| transactions | A number. This represents the number of transactions to be generated. |
| spike | A number. This represents the seasonality of data. It can take any value from 1 to 12. These numbers represent months in a year, from January to December respectively. For example, if the spike is set to 12, it means that December has the highest number of transactions. |
| outliers | A number. This signifies the presence of outliers. If set to value 1, then outliers are generated randomly. If set to value 0, then no outliers are generated. The presence of outliers is a very common occurrence and hence setting the outliers to 1 is recommended. However, there are instances where outliers are not needed. For example, if the objective of data generation is solely for visualization purposes then outliers may not be needed. |

## Value

A dataframe with day number and count of transactions on that day

## Examples

```
df <- genTrans(cycles = "y", trend = 1, transactions = 10000, spike = 10, outliers = 0)
df <- genTrans(cycles = "q", trend = -1, transactions = 32000, spike = 12, outliers = 1)
```

---

genTree *Generate complete m-ary connected graph*

---

## Description

Generates an m-ary connected graph that is complete. This function is used by another internal function buildHierarchy.

## Usage

```
genTree(m, depth)
```

## Arguments

| | |
|---|---|
| m | A positive number. This specifies the number of splits at each branch. |
| depth | A positive number. This specifies the number of levels of the tree. |

## Details

This function helps in generating data that is of a tree structure. To explain further, this function generates a data where there are less number of classes i.e. branches at the top i.e. the root and increase in number and increase towards the end i.e. the leaf nodes. The number of terminal nodes are dependent on the arguments $m$ and $depth$. More precisely, the number of terminal nodes has the formulation of

$$m^depth$$

. For instance, if $m$ is 2 and $depth$ is 3, then the number of terminal nodes are $2^3$ i.e. 8.

## Value

A dataframe.

## See Also

[buildHierarchy()] to build hierarchical data.

## genTriples *Extracts Three Consecutive Alphabets of the String*

### Description

For a given string, this function extracts three consecutive alphabets. This function is further used by genMatrix function.

### Usage

```
genTriples(s)
```

### Arguments

| | |
|---|---|
| s | A string. This is the string from which three consecutive alphabets are to be extracted. |

### Value

List of three alphabet combinations of the string input.

## missingArgHandler *Handle Missing Arguments in Function*

### Description

Replaces the missing argument with the default value. This is an internal function and is currently not exported in the package.

### Usage

```
missingArgHandler(argMissed, argDefault)
```

### Arguments

| | |
|---|---|
| argMissed | This is the argument that needs to be handled. |
| argDefault | This is the default value of the argument that is missing in the function called. |

### Details

This function plays the role of error handler by setting the default values of the arguments when a function is called without specifying any arguments.

### Value

The default value of the missing argument.

`nextAlphaProb` *Generate Next Alphabet*

## Description

Generates next alphabet based on prior probabilities.

## Usage

```
nextAlphaProb(alphaMatrix, currentAlpha, placement)
```

## Arguments

| | |
|---|---|
| `alphaMatrix` | A table. This table is generated using the `genMatrix` function . |
| `currentAlpha` | A string. This is the alphabet(s) for which the next alphabet is generated. |
| `placement` | A string. This takes one of the two values namely "first" or "all". |

## Details

The purpose of this function is to generate the next alphabet for a given alphabet(s). This function uses prior probabilities to generate the next alphabet. Although there are two types of input tables passed into the function by using the parameter *alphaMatrix*, the process to generate the next alphabet remains the same as given below.

Firstly, the input table contains frequencies of the combination of current alphabet *currentAlpha* (represented by rows) and next alphabet(represented by columns). These frequencies are converted into a percentage at a row level. This means that for each row, the sum of all the column values will add to 1.

Secondly, for the given *currentAlpha*, the table is looked up for the corresponding column where the probability is the highest. The alphabet for the column with maximum prior probability is selected as the next alphabet and is returned by the function.

## Value

The next alphabet following the input alphabet(s) passed by the argument *currentAlpha*.

---

`treeDf` *A supporting function.*

## Description

This is used by another internal function `genTree`.

## Usage

```
treeDf(...)
```

### Arguments

| | |
|---|---|
| ... | This is a placeholder argument. |

### Value

A dataframe.

---

`uncovrApi` *POST Function for Calling uncovr API*

---

### Description

This function makes the POST call to the *uncovr* API.

### Usage

```
uncovrApi(body, key)
```

### Arguments

| | |
|---|---|
| `body` | A json with the details of the independent and dependent variable. |
| `key` | An alpha numeric. This is the subscription key that can be sourced from the developer portal of uncovr API available at *https://foyi.developer.azure-api.net/*. |

### Details

The purpose of this function can be best understood when explained within the context that is given below. There is a closed source SaaS(Software as a Service) software named *uncovr* that provides an API(Application Programming Interface). In its current state, the SaaS software is free to use with some constraints around the volume of data and the frequency of API calls. One of the functions of *uncovr* API takes an input of number of observations i.e. rows and number of independent variables namely columns and gives an output. This function *uncovrApi* makes the connection to *uncovr* API and sources the response. #function to call uncovr api

### Value

A json.

# Index