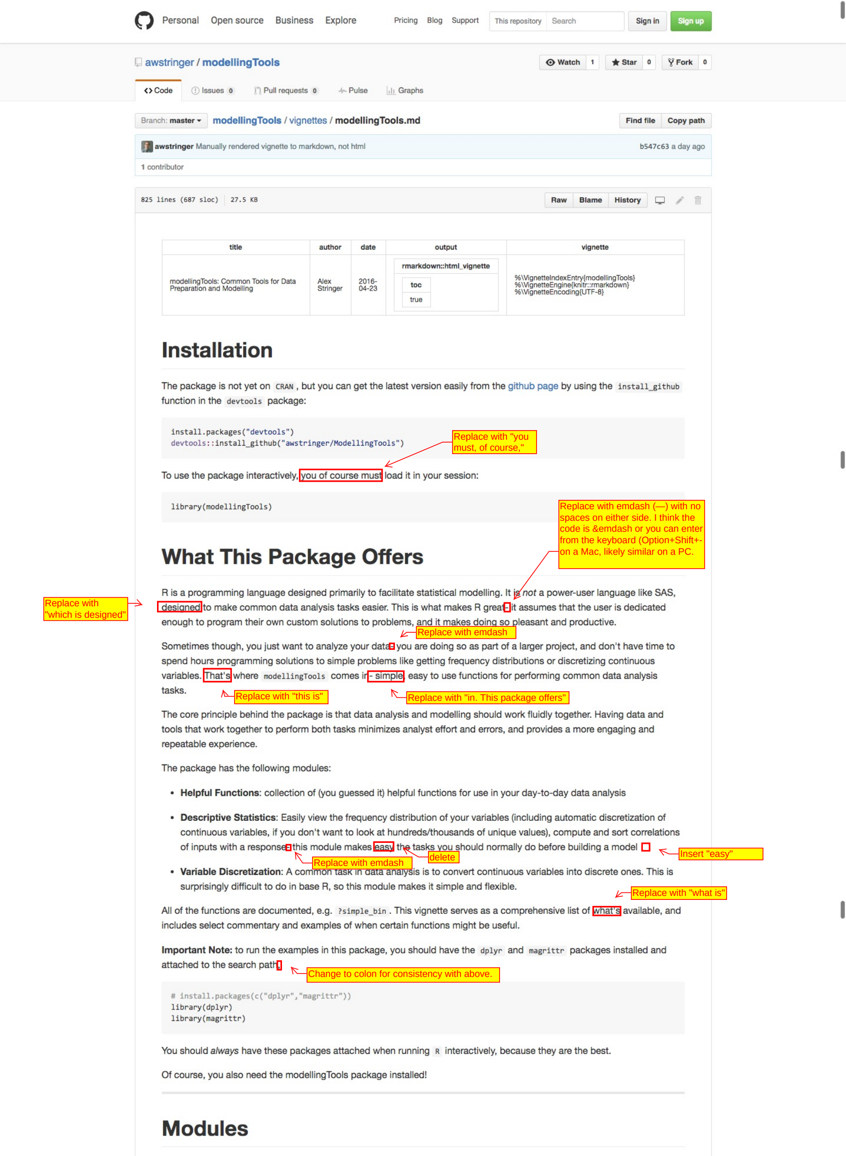

title	author	date	output	vignette
modellingTools: Common Tools for Data Preparation and Modelling	Alex Stringer	2016-04-23	rmarkdown::html_vignette toc true	%\VignetteIndexEntry{modellingTools} %\VignetteEngine{knitr::rmarkdown} %\VignetteEncoding{UTF-8}

Installation

The package is not yet on `CRAN`, but you can get the latest version easily from the github page by using the `install_github` function in the `devtools` package:

```
install.packages("devtools")
devtools::install_github("awstringer/ModellingTools")
```

Replace with "you must, of course,"

To use the package interactively, you of course must load it in your session:

```
library(modellingTools)
```

Replace with emdash (—) with no spaces on either side. I think the code is &emdash or you can enter from the keyboard (Option+Shift+- on a Mac, likely similar on a PC.

What This Package Offers

Replace with "which is designed"

R is a programming language designed primarily to facilitate statistical modelling. It is *not* a power-user language like SAS, designed to make common data analysis tasks easier. This is what makes R great - it assumes that the user is dedicated enough to program their own custom solutions to problems, and it makes doing so pleasant and productive.

Replace with emdash

Sometimes though, you just want to analyze your data - you are doing so as part of a larger project, and don't have time to spend hours programming solutions to simple problems like getting frequency distributions or discretizing continuous variables. That's where `modellingTools` comes in - simple, easy to use functions for performing common data analysis tasks.

Replace with "this is"

Replace with "in. This package offers"

The core principle behind the package is that data analysis and modelling should work fluidly together. Having data and tools that work together to perform both tasks minimizes analyst effort and errors, and provides a more engaging and repeatable experience.

The package has the following modules:

- **Helpful Functions**: collection of (you guessed it) helpful functions for use in your day-to-day data analysis
- **Descriptive Statistics**: Easily view the frequency distribution of your variables (including automatic discretization of continuous variables, if you don't want to look at hundreds/thousands of unique values), compute and sort correlations of inputs with a response - this module makes easy the tasks you should normally do before building a model
- **Variable Discretization**: A common task in data analysis is to convert continuous variables into discrete ones. This is surprisingly difficult to do in base R, so this module makes it simple and flexible.

Insert "easy"

delete

Replace with emdash

Replace with "what is"

All of the functions are documented, e.g. `?simple_bin`. This vignette serves as a comprehensive list of what's available, and includes select commentary and examples of when certain functions might be useful.

Important Note: to run the examples in this package, you should have the `dplyr` and `magrittr` packages installed and attached to the search path.

Change to colon for consistency with above.

```
# install.packages(c("dplyr","magrittr"))
library(dplyr)
library(magrittr)
```

You should *always* have these packages attached when running `R` interactively, because they are the best.

Of course, you also need the modellingTools package installed!

Modules

Helpful Functions

Replace with "that, frankly,"

This module contains some common functions that I use when programming to simplify tasks that frankly, should be simple. Most of them are simple one or two-liners, but they have been unit tested and documented for your convenience.

column_vector

This is a simple convenience function that grabs a single column of a dataframe and returns it as a vector, either `numeric` , `factor` or `character` depending on its type in the dataframe. Designed to work with the `tbl_df` class from the `dplyr` package but there is no reason it won't work on arbitrary dataframes.

I find that using this function makes my code cleaner and more predictable. For example,

```
d <- iris
d[,1]
```

should return a numeric vector with a name. However

Insert comma

```
d[,c(1,2)]
```

Replace with emdash

returns a dataframe. Maybe this makes sense but how many of us would look at the above code snippets and *know* what would be returned, without thinking? The `dplyr` package provides the `tbl_df` class (which you should always use!):

Delete comma

```
d <- tbl_df(iris)
d[,1]
```

```
## Source: local data frame [150 x 1]
##
##    Sepal.Length
##           (dbl)
## 1           5.1
## 2           4.9
## 3           4.7
## 4           4.6
## 5           5.0
## 6           5.4
## 7           4.6
## 8           5.0
## 9           4.4
## 10          4.9
## ..          ...
```

```
d[,c(1,2)]
```

```
## Source: local data frame [150 x 2]
##
##    Sepal.Length Sepal.Width
##           (dbl)       (dbl)
## 1           5.1         3.5
## 2           4.9         3.0
## 3           4.7         3.2
## 4           4.6         3.1
## 5           5.0         3.6
## 6           5.4         3.9
## 7           4.6         3.4
## 8           5.0         3.4
## 9           4.4         2.9
## 10          4.9         3.1
## ..          ...         ...
```

which provides consistent behaviour. But then, how do you get a numeric vector containing just one column?

```
unlist(d[,1])
```

Save yourself some typing and use `column_vector` :

```
column_vector(d,1)
```

This will work with both types of dataframe, as well as objects of class `matrix` and `array` , so your code will be clean and predictable.

Since I am claiming that behaviour of functions should be simple, I should point out the following. What happens if you run

You may want to specify what this returns

```
column_vector(d,c(1,2))
```

Replace with emdash

Probably not what you would expect, or want I recommend against using it this way. Indeed, `column_vector` will give a warning if the user supplies multiple requested columns.

Descriptive Statistics

Insert "easily"

Delete

This module provides functions for performing these simple, yet sometimes tricky to implement data analysis tasks.

For example, everyone should be able to easily compute the frequency distribution of their variables. R provides many useful functions for doing this, however they are often confusing and have inconsistent output formats.

proc_freq

How do you get the frequency distribution of a variable in `R`? You can use `table`:

```
data(CO2)
CO2 %$% table(Plant)
```

```
## Plant
## Qn1 Qn2 Qn3 Qc1 Qc3 Qc2 Mn3 Mn2 Mn1 Mc2 Mc3 Mc1
##   7   7   7   7   7   7   7   7   7   7   7   7
```

This works for cross-tabulation as well:

```
CO2 %$% table(Plant,Type)
```

```
##      Type
## Plant Quebec Mississippi
##   Qn1      7           0
##   Qn2      7           0
##   Qn3      7           0
##   Qc1      7           0
##   Qc3      7           0
##   Qc2      7           0
##   Mn3      0           7
##   Mn2      0           7
##   Mn1      0           7
##   Mc2      0           7
##   Mc3      0           7
##   Mc1      0           7
```

Replace with "three-way"

What about three way tables? You can try the following yourself:

```
CO2 %$% table(Plant,Type,Treatment)
```

(by the way, the `%$%` operator is from the `magrittr` package. Highly recommend checking this out, it will change the way you program in `R`).

This works well, but

- the output format is a named vector, which can be hard to read
- the output format changes as you provide more arguments (for two variables it is a matrix, for 3+ it is a multidimensional array)

  Insert comma
- the documentation is confusing (if you figure it out let me know!)

Why should there be 10 arguments to a function that only does one thing? `proc_freq` has three: the `dat` aset that the variable is in, the `var` iable that you want the distribution of, and an optional **awesome** argument that specifies whether you want to discretize a continuous variable into `bins` before summarizing.

The usage is simple:

```
proc_freq(iris,"Species")
```

```
## Source: local data frame [3 x 3]
##
##        level count percent
##       (fctr) (int)   (chr)
## 1     setosa    50   33.3%
## 2 versicolor    50   33.3%
## 3  virginica    50   33.3%
```

The output is a `tbl_df`, a great data structure from the `dplyr` package, and contains counts and percentages.

What about the mysterious third argument to `proc_freq`: `bins`. This is used to discretize a continuous variable prior to summarizing. Why might one want to do this? When dealing with smaller, evenly distributed datasets, the need for this is not apparent:

```
proc_freq(iris,"Sepal.Length")
```

```
## Source: local data frame [35 x 3]
##
##    level count percent
##    (dbl) (int)   (chr)
## 1    4.3     1   0.67%
## 2    4.4     3   2.00%
## 3    4.5     1   0.67%
## 4    4.6     4   2.67%
## 5    4.7     2   1.33%
## 6    4.8     5   3.33%
## 7    4.9     6   4.00%
## 8    5.0    10   6.67%
## 9    5.1     9   6.00%
## 10   5.2     4   2.67%
## ..   ...   ...     ...
```

This provides manageable, interpretible output. In my day to day work though, I deal with massive datasets (10,000,000+ rows), which often have

Replace with emdash

- large proportions of *special* values datapoints that are missing for some known reason
- large proportions of *actual* missing values

I don't want to know how many times each unique value occurs in a dataset with 10,000,000 rows, and I *definitely* don't want to exclude missing values (have you noticed that that is the default behaviour of `table`?). What happens when you use `table`?

```
library(foreach) # Check this out if you haven't already
# Try for 1,000 observations, for illustration purposes
d <- data_frame(v1 = times(1e04) %do% if (runif(1) < .1) NA else rnorm(1))
```

Try running the following yourself:

```
d %$% table(v1)
proc_freq(d,"v1")
```

The output of `table` is a giant vector that is essentally unreadable without some further intervention on the part of the programmer not ideal. The output of `proc_freq` is readable but useless, since there are so many unique values. There is a better way:

Replace with emdash

```
proc_freq(d,"v1",bins = 3)
```

```
## Source: local data frame [4 x 3]
## 
##           level count percent
##          (fctr) (int)   (chr)
## 1 [-3.86,-1.13]  1113   11.1%
## 2   (-1.13,1.6]  7368   73.7%
## 3    (1.6,4.34]   511    5.1%
## 4            NA  1008   10.1%
```

Using `bins = 3` with `proc_freq` gives the user insight into the distribution of the data and the proportion of missing/special values using one line of code and a single, consistent output format.

get_top_corrs

This function helps to answer the simple questions "are any input variables correlated with the response of interest and if so, which ones and how much?". You can get this information using base `R` by, for example:

```
x <- mtcars[,-1]
y <- mtcars[,1]
apply(x,2,function(a,b) cor(a,b),b = y)
```

```
##        cyl       disp         hp       drat         wt       qsec 
## -0.8521620 -0.8475514 -0.7761684  0.6811719 -0.8676594  0.4186840 
##         vs         am       gear       carb 
##  0.6640389  0.5998324  0.4802848 -0.5509251
```

This works okay, but the output is a bit verbose and difficult to work with. Oh and hard to read if you have more than a few variables. Oh and what if some of them are non-numeric? Okay I'll build in an `if` statement to check and darn, what about `NA` s...

Insert comma

Stop. Use `get_top_corrs` instead! This function

- Returns a `tbl_df` with two columns: the `var_name` and the `correlation` with `response_var`
- Sorts variables in descending order of `correlation`, for easy viewing
- Drops all non-numeric variables
- Filters out `NA` values from correlation computations
- Is paralellized, if you have a parallel backend registered (see package `doParalell`)

So we can do

```
get_top_corrs(mtcars,"mpg")
```

```
## Source: local data frame [10 x 2]
## 
##    var_name correlation
##       (chr)       (dbl)
## 1        wt  -0.8676594
## 2       cyl  -0.8521620
## 3      disp  -0.8475514
## 4        hp  -0.7761684
## 5      drat   0.6811719
## 6        vs   0.6640389
## 7        am   0.5998324
## 8      carb  -0.5509251
## 9      gear   0.4802848
## 10     qsec   0.4186840
```

and gain immediate insights, instead of trying to do it the hard way for 5 minutes and just giving up.

Variable Discretization

Discretizing continuous variables is often useful practice for removing undesirable patterns or improving model predictions. R provides useful functions (`cut`) for binning vectors, but it takes some work on the part of the programmer to apply these consistently to a whole dataset.

What is meant by *discretization*, also known as *binning*, is to convert a numeric variable into a categorical one, by assigning each observation to a "bin" depending on its numeric value. For example, if you aren't interested in whether a value is `1.9` or `2.1` , but only whether it is `< 2` or `>= 2` , you might want to try binning your data.

There are three types of binning available in `modellingTools` :

- **Equal Width Binning**: this will bin a continuous variable into a specified number of bins, each of which has the same length in units of the original variable. For example, binning into ranges (0,1], (1,2], (2,3] ... and so on. This is useful for analyzing the frequency distribution of a continuous variable with many unique levels in fact, this is exactly what you are doing when you make a histogram

  Replace with emdash

- **Equal Height Binning**: this will bin a continuous variable into a specified number of bins that each have the same *number of observations* in them. That is, the actual cutpoints of the bins are chosen by the data. This helps when attempting to estimate how a rate, differs according to a continuous variable, but the distribution of the continuous variable is not flat. For example, electoral ridings in Canada are (roughly) chosen to have the same number of people in them. Imagine if they were chosen instead based on fixed geographic area- Toronto/Vancouver/Montreal would only get one MP each!

  Insert comma

- **Arbitrary Binning**: this simply lets the user specify their own cutpoints. This is provided to ensure complete generality, but it is particularly useful when the user wishes to create bins based on one dataset, and then analyze the same variable on a different dataset using the same bins for example when calculating the same descriptive statistics on datasets containing the same variables, but collected at different points in time.

  Replace with emdash

simple_bin

This is currently the main function in the discretization module. The purpose is to make it simple to convert each continuous variable in a dataset to a categorical variable by applying *equal-height* or *equal-width* bins, and to make it simple to bin the same variables in a new dataset into those same bins. The primary use would be in building a model on a training set and applying it to a test set, which requires the categorical variables in both to have the same levels.

For details on the arguments to `simple_bin` , see the documentation `?modellingTools::simple_bin` . We can observe the basic usage of `simple_bin` as follows:

```
data(Seatbelts)
d <- tbl_df(as.data.frame(Seatbelts))
d
```

```
## Source: local data frame [192 x 8]
##
##    DriversKilled drivers front  rear   kms PetrolPrice VanKilled   law
##            (dbl)   (dbl) (dbl) (dbl) (dbl)       (dbl)     (dbl) (dbl)
## 1            107    1687   867   269  9059   0.1029718        12     0
## 2             97    1508   825   265  7685   0.1023630         6     0
## 3            102    1507   806   319  9963   0.1020625        12     0
## 4             87    1385   814   407 10955   0.1008733         8     0
## 5            119    1632   991   454 11823   0.1010197        10     0
## 6            106    1511   945   427 12391   0.1005812        13     0
## 7            110    1559  1004   522 13460   0.1037740        11     0
## 8            106    1630  1091   536 14055   0.1040764         6     0
## 9            107    1579   958   405 12106   0.1037740        10     0
## 10           134    1653   850   437 11372   0.1030264        16     0
## ..           ...     ...   ...   ...   ...         ...       ...   ...
```

```
d_bin <- simple_bin(d,bins = 3)
d_bin
```

```
## Source: local data frame [192 x 8]
##
##    DriversKilled             drivers         front      rear
##           (fctr)              (fctr)        (fctr)    (fctr)
## 1       [60,110] [1.52e+03,1.76e+03]     [764,905] [224,365]
## 2       [60,110] [1.06e+03,1.52e+03]     [764,905] [224,365]
## 3       [60,110] [1.06e+03,1.52e+03]     [764,905] [224,365]
## 4       [60,110] [1.06e+03,1.52e+03]     [764,905] [365,434]
## 5      [110,131] [1.52e+03,1.76e+03] [905,1.3e+03] [434,646]
## 6       [60,110] [1.06e+03,1.52e+03] [905,1.3e+03] [365,434]
## 7      [110,131] [1.52e+03,1.76e+03] [905,1.3e+03] [434,646]
## 8       [60,110] [1.52e+03,1.76e+03] [905,1.3e+03] [434,646]
## 9       [60,110] [1.52e+03,1.76e+03] [905,1.3e+03] [365,434]
## 10     [131,198] [1.52e+03,1.76e+03]     [764,905] [434,646]
## ..           ...                 ...           ...       ...
## Variables not shown: kms (fctr), PetrolPrice (fctr), VanKilled (fctr), law
##   (dbl)
```

Notice that the variable `law` retains its `dbl` class- `simple_bin` automatically detects if there are not enough unique values of a variable to propoerly bin, and leaves them untouched. Variables that are already categorical are also automatically ignored. If you would like to know the cutpoints for the bins, check out `modellingTools:binned_data_cutpoints` below.

When building a model, you very well may wish to fit the model on one subset of the data (a training set) and evaluate predictions on a different subset (a testing set). If dealing with variables that have been discretized (by you), this requires the variables in the testing set to be binned into the same ranges.

`simple_bin` automatically bins an optional test dataset into the same bins used on the training set:

```
d %<>% mutate(on_train = times(n()) %do% if (runif(1) < .7) 1 else 0)
d_train <- d %>% filter(on_train == 1) %>% select(-on_train)
d_test <- d %>% filter(on_train == 0) %>% select(-on_train)

d_split_binned <- simple_bin(d_train,test = d_test,bins = 3)
d_split_binned
```

```
## $train
## Source: local data frame [129 x 8]
##
##    DriversKilled             drivers         front      rear
##           (fctr)              (fctr)        (fctr)    (fctr)
## 1       [60,111] [1.52e+03,1.76e+03]     [780,901] [224,365]
## 2       [60,111] [1.06e+03,1.52e+03]     [780,901] [224,365]
## 3       [60,111] [1.06e+03,1.52e+03]     [780,901] [224,365]
## 4       [60,111] [1.06e+03,1.52e+03]     [780,901] [365,431]
## 5       [60,111] [1.06e+03,1.52e+03] [901,1.3e+03] [365,431]
## 6       [60,111] [1.52e+03,1.76e+03] [901,1.3e+03] [431,646]
## 7       [60,111] [1.52e+03,1.76e+03] [901,1.3e+03] [431,646]
## 8      [132,198] [1.52e+03,1.76e+03]     [780,901] [431,646]
## 9      [111,132] [1.52e+03,1.76e+03] [901,1.3e+03] [224,365]
## 10     [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [224,365]
## ..           ...                 ...           ...       ...
## Variables not shown: kms (fctr), PetrolPrice (fctr), VanKilled (fctr), law
##   (dbl)
##
## $test
## Source: local data frame [63 x 8]
##
##    DriversKilled             drivers         front      rear
##           (fctr)              (fctr)        (fctr)    (fctr)
## 1      [111,132] [1.52e+03,1.76e+03] [901,1.3e+03] [431,646]
## 2       [60,111] [1.52e+03,1.76e+03] [901,1.3e+03] [365,431]
## 3      [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [431,646]
## 4      [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [431,646]
## 5       [60,111] [1.52e+03,1.76e+03]     [780,901] [224,365]
## 6      [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [431,646]
## 7      [132,198] [1.52e+03,1.76e+03] [901,1.3e+03] [431,646]
## 8      [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [431,646]
## 9      [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [365,431]
## 10     [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [431,646]
## ..           ...                 ...           ...       ...
## Variables not shown: kms (fctr), PetrolPrice (fctr), VanKilled (fctr), law
##   (dbl)
```

Replace with emdash

But wait why can't we just bin the data *before* we split it up into training/test sets? This implicitly assumes that the test data will *always* be distributed *exactly* the same as the training data. This may be true when you are building the model (e.g. if you randomly split a complete dataset into the two parts) but if you go to deploy the model, it won't be true on the next data that you need predictions on.

Even if you don't have the test set at the time you build the model, or if you wish to use the same model on future datasets, `simple_bin` can take custom bins. These must be passed as a named list the exact format output by `modellingTools::binned_data_cutpoints` (see below):

Replace with emdash

```
d_train_bin <- simple_bin(d_train,bins = 3)
train_cutpoints <- binned_data_cutpoints(d_train_bin)
train_cutpoints
```

```
## $DriversKilled
## [1]  60 111 132 198
##
## $drivers
## [1] 1060 1520 1760 2650
##
## $front
## [1]  426  780  901 1300
##
## $rear
## [1] 224 365 431 646
##
## $kms
## [1]  7680 13500 16700 21500
##
## $PetrolPrice
## [1] 0.0812 0.0974 0.1120 0.1330
##
## $VanKilled
## [1]  2  7 11 17
```

```
d_test_bin <- simple_bin(d_test,bins = train_cutpoints)
d_test_bin
```

```
## Source: local data frame [63 x 8]
##
##    DriversKilled             drivers         front      rear
```

```
##          (fctr)              (fctr)        (fctr)    (fctr)
## 1      [111,132] [1.52e+03,1.76e+03] [901,1.3e+03] [431,646]
## 2       [60,111] [1.52e+03,1.76e+03] [901,1.3e+03] [365,431]
## 3      [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [431,646]
## 4      [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [431,646]
## 5       [60,111] [1.52e+03,1.76e+03]     [780,901] [224,365]
## 6      [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [431,646]
## 7      [132,198] [1.52e+03,1.76e+03] [901,1.3e+03] [431,646]
## 8      [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [431,646]
## 9      [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [365,431]
## 10     [132,198] [1.76e+03,2.65e+03] [901,1.3e+03] [431,646]
## ..           ...                 ...           ...       ...
## Variables not shown: kms (fctr), PetrolPrice (fctr), VanKilled (fctr), law
##   (dbl)
```

```
identical(d_test_bin,d_split_binned$test)
```

```
## [1] TRUE
```

vector_bin

This function makes it easy to discretize a vector of variables into equal height/width or custom bins. This is a convenience function which is essentially just a wrapper around `cut` / `cut_number` / `cut_interval` , with one major difference: by default, missing values are given a bin. This can be useful when modelling data that is missing not-at-random.

Usage is straightforward:

```
x <- rnorm(10)
vector_bin(x,bins = 3)
```

```
##  [1] [-0.719,0.589] [0.589,1.15]   [0.589,1.15]   [-2.31,-0.719]
##  [5] [-2.31,-0.719] [-0.719,0.589] [-0.719,0.589] [0.589,1.15]
##  [9] [-2.31,-0.719] [0.589,1.15]
## Levels: [-0.719,0.589] [-2.31,-0.719] [0.589,1.15]
```

```
vector_bin(x,bins = 3,type = "width")
```

```
##  [1] [-1.15,-0.0022] [-0.0022,1.15]  [-0.0022,1.15]  [-1.15,-0.0022]
##  [5] [-2.31,-1.15]   [-1.15,-0.0022] [-1.15,-0.0022] [-0.0022,1.15]
##  [9] [-1.15,-0.0022] [-0.0022,1.15]
## Levels: [-0.0022,1.15] [-1.15,-0.0022] [-2.31,-1.15]
```

```
vector_bin(x,bins = c(-1,0,1))
```

```
##  [1] [-1,0]  Missing [0,1]   [-1,0]  Missing [-1,0]  [-1,0]  [0,1]
##  [9] [-1,0]  Missing
## Levels: [-1,0] [0,1] Missing
```

get_vector_cutpoints

I found it was surprisingly difficult in `R` to perform the simple task of figuring out the actual cutpoints of a vector that has been discretized using one of the `cut` family. What if you try

```
x <- vector_bin(rnorm(100),bins = 3)
levels(x)
```

```
## [1] "[-0.419,0.307]" "[-3.01,-0.419]" "[0.307,2.67]"
```

You get a vector of character strings, containing brackets and commas. Okay, well I can use a string replace function from `stringr` to get rid of the brackets (don't forget to escape them twice!), then split the string on the commas, oh darn that returns a list, okay unlist it...

Stop. Use `get_vector_cutpoints` . This uses a regular expression to parse the unique numeric values from the above character vector and returns a numeric vector of the (sorted!) unique cutpoints.

```
get_vector_cutpoints(x)
```

```
## [1] -3.010 -0.419  0.307  2.670
```

binned_data_cutpoints

Typically data is stored in dataframes, not individual vectors. Rather than calling `column_vector` on a vector before using `get_vector_cutpoints` , and then doing this for all variables in your dataset (more programming!), use `binned_data_cutpoints` to quickly get a named list, each element being a numeric vector of cutpoints for the corresponding factor variable.

For example, remember the `Seatbelts` data that we binned? If you'd like to know what the actual binned values are for each variable, try:

```
binned_data_cutpoints(d_bin)
```

```
## $DriversKilled
## [1]  60 110 131 198
## 
## $drivers
## [1] 1060 1520 1760 2650
## 
## $front
## [1]  426  764  905 1300
## 
## $rear
## [1] 224 365 434 646
## 
## $kms
## [1]  7680 13600 16600 21600
## 
## $PetrolPrice
## [1] 0.0812 0.0974 0.1110 0.1330
## 
## $VanKilled
## [1]  2  7 11 17
```

Replace with emdash

The primary use case for this is when binning training set variables for binning a model in order to predict on a test set, you are going to need to be able to bin the test set variables into the *same* bins that you used for the training set variables. Otherwise, you will be comparing factor variables with different levels, which doesn't make sense.

create_model_matrix

When using the base `R` functions for modelling (`lm` , `glm` , etc) and many established advanced functions in other packages (`nlme::nlme` , `glmm:glmm` , etc), the user is able to specify a model using a formula interface:

```
mod <- lm(y ~ x,data = mydata)
```

This is very convenient because

- It corresponds to how we write models down on paper
- You can keep your input and target variables in the same dataframe

The base `R` package provides the `model.matrix` function, which takes a formula and a matrix/dataframe and creates the corresponding dummy variables. `create_model_matrix` is a wrapper around this function that takes only a dataframe, and automatically creates dummies for only the factor variables, and outputs either a matrix or a `tbl` .

The usage is straightforward:

```
# Print this in your own R console:
x <- create_model_matrix(d_train_bin)

# Prettier output:
create_model_matrix(d_train_bin,matrix_out = FALSE)
```

```
## Source: local data frame [129 x 22]
## 
##    DriversKilled[111,132] DriversKilled[132,198] DriversKilled[60,111]
##                      (dbl)                  (dbl)                 (dbl)
## 1                        0                      0                     1
## 2                        0                      0                     1
## 3                        0                      0                     1
## 4                        0                      0                     1
## 5                        0                      0                     1
## 6                        0                      0                     1
## 7                        0                      0                     1
## 8                        0                      1                     0
## 9                        1                      0                     0
## 10                       0                      1                     0
## ..                     ...                    ...                   ...
## Variables not shown: drivers[1.06e+03,1.52e+03] (dbl),
##   drivers[1.52e+03,1.76e+03] (dbl), drivers[1.76e+03,2.65e+03] (dbl),
##   front[426,780] (dbl), front[780,901] (dbl), front[901,1.3e+03] (dbl),
##   rear[224,365] (dbl), rear[365,431] (dbl), rear[431,646] (dbl),
##   kms[1.35e+04,1.67e+04] (dbl), kms[1.67e+04,2.15e+04] (dbl),
##   kms[7.68e+03,1.35e+04] (dbl), PetrolPrice[0.0812,0.0974] (dbl),
##   PetrolPrice[0.0974,0.112] (dbl), PetrolPrice[0.112,0.133] (dbl),
##   VanKilled[11,17] (dbl), VanKilled[2,7] (dbl), VanKilled[7,11] (dbl), id
##   (int)
```

The default, matrix output is suitable for input into other `R` packages, such as `glmnet` .

© 2016 GitHub, Inc.