rrrsa

Ben Peloquin

[created sept16/2016, compiled by tim, oct06/2017]

Rational speech act model (RSA) of pragmatic inference

`rrrsa` is an R package for running RSA models – Bayesian models of pragmatic inference. `rrrsa` was created by Ben Peloquin in collaboration with Michael C. Frank and has been optimized for analysis of experimental data such as those presented in Frank, et al. (Under Review) and Peloquin & Frank (2016). For other, more flexible variants of RSA models, please see http://forestdb.org/models/scalar-implicature.html.

Installation

You can install the latest version of `rrrsa` by installing `devtools` and running:

```
# install.packages("devtools")
# devtools::install_github("benpeloquin7/rrrsa")
```

What is RSA?

Rational speech act (RSA) models frame language understanding as a special case of social cognition in which `speakers` and `listeners` reason about one another recursively. A `pragmatic listener` $P_{L_n}(m|u)$, reasons about intended meaning m of an utterance u by a `rational speaker` $P_{S_n}(u|m)$ who chooses an utterance according to the expected utility of an utterance $U(m; u)$. α is a decision noise parameter.

$$P_{L_n}(m|u) \propto P_{S_n}(u|m)P(m)$$

$$P_{S_n} \propto e^{U(m;u)}$$

$$U(m; u) = -\alpha(-\log(P_{L_{n-1}}(m|u)) - C(u))$$

See Frank & Goodman (2012) and Goodman & Stuhmuller (2013) for the original descriptions of the RSA framework. Frank, et al. (Under Review) also provides a comprehensive presentation and evaluation of RSA.

rrrsa includes empirical data

Data from “Rational speech act models of pragmatic reasoning in reference games” (Frank, et al., Under Review) and “Determining the alternatives in scalar implicature” (Peloquin & Frank, 2016) are also included in this package. Examples using data from these studies are included below.

rrrsa includes access to all model components

`rrrsa` provides users with access to all model components. The following sections demonstrate how this functionality can be used.

```
library(knitr)
library(ggplot2)
library(tidyr)
library(dplyr)
library(purrr)
library(rrrsa)
```

Calculating the informativity of an utterance with `rsa.informativity()`

`rsa.informativity()` takes three arguments, literal semantics P_{L_0}, `alpha` level (default 1), and `cost` (default 0). This function returns the surprisal of an utterance minus cost, multiplied by alpha.

```
rsa.informativity(0.4)
```

```
## [1] 0.4
```

```
rsa.informativity(rsa.informativity(0.4), alpha = 2, cost = 0.5)
```

```
## [1] 0.4349251
```

Calculating the utility of an utterance with `rsa.utility()`

`rsa.utility` takes an input vector of literal listener semantics and outputs a normalized vector of speaker likelihoods. If `costs` are not specified the default 0's vector is used. If `alpha` is not specified a default value of 1 is used.

```
literalSemantics <- c(0.0, 0.0, 0.3, 0.3, 0.4)
costs <- c(0.0, 0.0, 0.2, 0.3, 0.4)
rsa.utility(items = literalSemantics, costs = costs, alpha = 3)
```

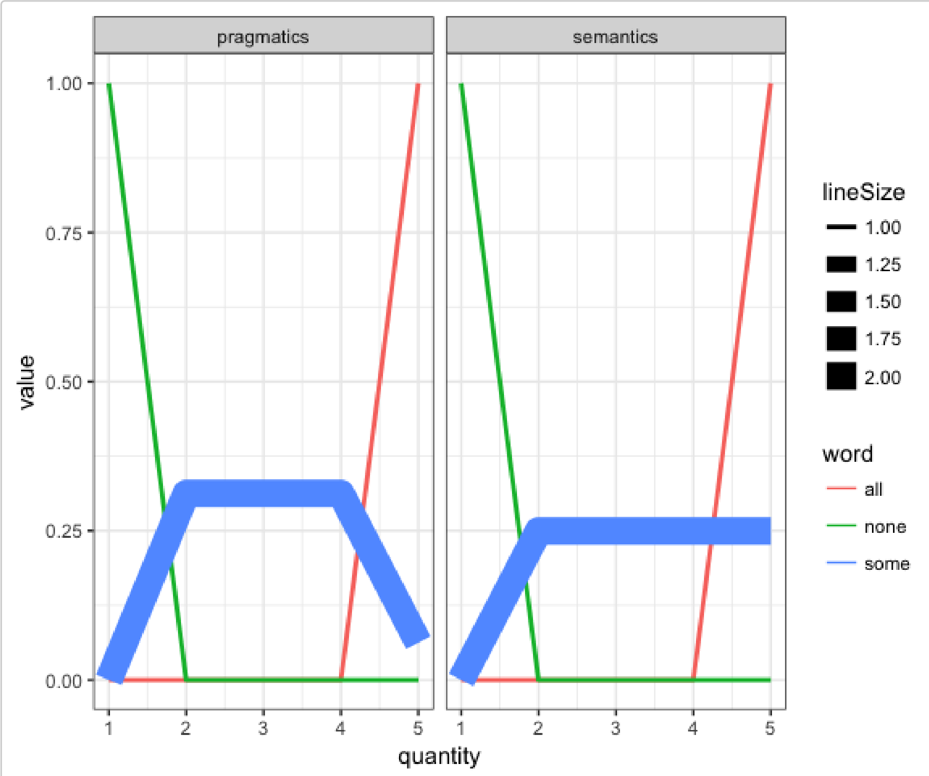
```
## [1] 0.0000000 0.0000000 0.1499485 0.2024093 0.6476421
```

Computing one full recursion with `rsa.fullRecursion()`

In the RSA framework one full recursion consists of a `pragmatic listner` P_{L_1} who reasons about a `rational speaker` P_{s_1} who reason about a `literal listener` P_{L_0}. Expected input is an m matrix of P_{L_0} `literal listener` values in which columns correspond to items (words) and rows correspond to semantic quantity (i.e. "stars" in Peloquin & Frank, 2016). Optional arguments include a `costs` vector which must be the same length as `ncol` and an optional `priors` vector which must be the same length as `nrows`. `rsa.fullRecursion()` provides safety checking for these cases. Output corresponds with pragmatic listener posterior predictions.

```
m <- matrix(data = c(1.0, 0.0, 0.0, 0.0, 0.0,
                     0.0, 0.25, 0.25, 0.25, 0.25,
                     0.0, 0.0, 0.0, 0.0, 1.0), nrow = 5)
colnames(m) <- c("none", "some", "all")
rownames(m) <- 1:5
# costs <- c("none" = 0, "some" = 0, "all" = 0)
# priors <- rnorm(n = nrow(m), mean = 0.5, sd = 0.1)
res <- rsa.fullRecursion(m = m)
res <- as.data.frame(res) %>%
  mutate(quantity = rownames(.))

## Prep data
pragmaticsTidied <- res %>%
  gather(word, pragmatics, -quantity)
semanticsTidied <- as.data.frame(m) %>%
  mutate(quantity = rownames(.)) %>%
  gather(word, semantics, c(none, some, all))
fullData <- merge(pragmaticsTidied, semanticsTidied) %>%
  gather(type, value, c(pragmatics, semantics)) %>%
  mutate(quantity = as.numeric(quantity),
         lineSize = ifelse(word == "some", 2, 1))

## Visualize implicature
ggplot(fullData, aes(x = quantity, y = value, col = word)) +
  geom_line(aes(size=lineSize)) +
```

```
  facet_wrap(~type) +
  theme_bw()
```



Running multiple recursions with `rsa.reason()`

`rsa.reason()` is a wrapper function for `rsa.fullRecursion` which provides an additional `depth` parameter specifying the recursive depth during reasoning. If depth is not provided, default value is 1.

```
all(rsa.reason(m = m, depth = 2) == rsa.fullRecursion(rsa.fullRecursion(m = m)))
```

[1] TRUE

```
rsa.reason(m = m, depth = 2) %>%
  kable()
```

none	some	all
1	0.0000000	0
0	0.3269231	0
0	0.3269231	0
0	0.3269231	0
0	0.0192308	1

Running data frames with `rsa.runDf()`

Run RSA on a tidied data frame and avoid running individual model components individually with `rsa.runDf()`. This is the primary workhorse function of `rrrsa`. An rrrsa-ready, tidied data frame must contain columns for semantic `quantity`, `item` and `semantics`, where each row corresponds with unique `item/quantity` combination. A user should specify their naming convention for these items in the `quantityVarName`, `itemVarName` and `semanticsVarName` arguments. The `costVarName` and `priorsVarName` arguments correspond with `costs` and/or `priors` data. Users can specify values for `alpha` and `depth` hyperparamenters. `rsa.runDf()` will return a data frame with model predictions '`preds`' appended as a new column.

```r
## Hypothetical literal listener data we might want to use RSA to simulate implicature.
df <- data.frame(scales = rep("some_all", 15),
                 stars = as.factor(rep(1:5, 3)),
                 starsChar = as.factor(rep(c("one", "two", "three", "four", "five"), 3)),
                 words = c(rep("all", 5), rep("some", 5), rep("none", 5)),
                 listenerSemantics = c(rep(0.0, 4), 1.0,
                              0.0, rep(0.25, 4),
                              1.0, rep(0.0, 4)))
rsa.runDf(df,
          quantityVarName="stars",
          semanticsVarName="listenerSemantics",
          itemVarName="words") %>%
  kable()
```

scales	stars	starsChar	words	listenerSemantics	preds
some_all	1	one	all	0.00	0.0000
some_all	2	two	all	0.00	0.0000
some_all	3	three	all	0.00	0.0000
some_all	4	four	all	0.00	0.0000
some_all	5	five	all	1.00	1.0000
some_all	1	one	some	0.00	0.0000
some_all	2	two	some	0.25	0.3125
some_all	3	three	some	0.25	0.3125
some_all	4	four	some	0.25	0.3125
some_all	5	five	some	0.25	0.0625
some_all	1	one	none	1.00	1.0000
some_all	2	two	none	0.00	0.0000
some_all	3	three	none	0.00	0.0000
some_all	4	four	none	0.00	0.0000
some_all	5	five	none	0.00	0.0000

Importantly, `rsa.runDf()` maintains all column naming and can handle multiple data types. For example, we can run `rsa.runDf()` with a character vector for `quantity` (contrast with the factor vector used above):

```r
all(rsa.runDf(df, quantityVarName = "starsChar",
              semanticsVarName = "listenerSemantics",
              itemVarName = "words") ==
      rsa.runDf(df, quantityVarName = "stars",
                semanticsVarName = "listenerSemantics",
                itemVarName = "words"))
```

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

A frequent use case for RSA will require running RSA over multiple groups of data. Rather than subsetting data frames and running RSA iteratively, we recommend using `map_df()` from the `purrr` package. Here we split by the `scales` variable.

```r
## RSA of literal semantics we'd like to run RSA on
df <- data.frame(scales = c(rep("some_all", 10), rep("good_excellent", 10)),
                 stars = as.factor(rep(1:5, 4)),
                 words = c(rep("all", 5), rep("some", 5), c(rep("excellent", 5), rep("good",
5))),
                 listenerSemantics = c(rep(0.0, 4), 1.0,
                              0.0, rep(0.25, 4),
                              rep(0.0, 4), 1.0,
                              0.0, rep(0.25, 4))) %>% mutate(priors = 0.20)
```

```r
df$costs <- c(rep(3, 5), rep(4, 5), rep(9, 5), rep(4, 5))

## Using purrr::map_df() run rsa.runDf() over df subsets
df %>%
  split(.$scales) %>%
  map_df(~rsa.runDf(
    data=.x,
    quantityVarName="stars",
    semanticsVarName="listenerSemantics",
    itemVarName="words",
    costsVarName="costs",
    depth=2)) %>%
  head(n=10) %>%
  kable()
```

scales	stars	words	listenerSemantics	priors	costs	preds
good_excellent	1	excellent	0.00	0.2	9	0.0000000
good_excellent	2	excellent	0.00	0.2	9	0.0000000
good_excellent	3	excellent	0.00	0.2	9	0.0000000
good_excellent	4	excellent	0.00	0.2	9	0.0000000
good_excellent	5	excellent	1.00	0.2	9	1.0000000
good_excellent	1	good	0.00	0.2	4	0.0000000
good_excellent	2	good	0.25	0.2	4	0.3333329
good_excellent	3	good	0.25	0.2	4	0.3333329
good_excellent	4	good	0.25	0.2	4	0.3333329
good_excellent	5	good	0.25	0.2	4	0.0000013

An example tuning hyperparameters using `rsa.runDf()` and `purrr::map_df()` with empirical data from Peloquin & Frank (2016)

Data description for Peloquin & Frank (2016) "Determining the alternatives for scalar implicature"

`rrrsa` includes empirical literal listener P_{L_0} which can be used as input to `rrrsa` as well as P_{L_1} pragmatic judgments for model tuning and comparison. Four data sets are included:

`peloquinFrank_2Alts`: data set with entailment alternatives

`peloquinFrank_3Alts`: data set with entailment alternatives + universal none

`peloquinFrank_4Alts`: data set with entailment alternatives + top two empirically derived alts

`peloquinFrank_5Alts`: data set with entailment alternatives + top two empirically derived alts + neutral valence alternative

```r
str(peloquinFrank_2Alts) %>%
  kable()
```

```
## 'data.frame':    50 obs. of  7 variables:
##  $ exp      : Factor w/ 3 levels "e10","e12","e8": 3 3 3 3 3 3 3 3 3 3 ...
##  $ scale    : Factor w/ 5 levels "good_excellent",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ stars    : int  1 2 3 4 5 1 2 3 4 5 ...
##  $ speaker.p: num  0 0.0345 0.069 0.4138 0.4828 ...
##  $ words    : chr  "excellent" "excellent" "excellent" "excellent" ...
##  $ e11      : num  0 0 0 0.302 0.698 ...
##  $ e6       : num  0 0 0 0.0732 0.9268 ...
```

See `?peloquinFrank_2alts` for data descriptions and [link to CogSci paper here…]

RSA hyperparameters `alpha` and `depth` can be tuned using custom functions. Here we give a simple example of hyperparamter tuning with grid search using nested for loops. We examine empirical data from Peloquin & Frank (2016).

In this case our grouping variable is `scales` – we run `rsa.runDf()` on each scale subset for each alpha and depth we're interested in saving the results to `res`.

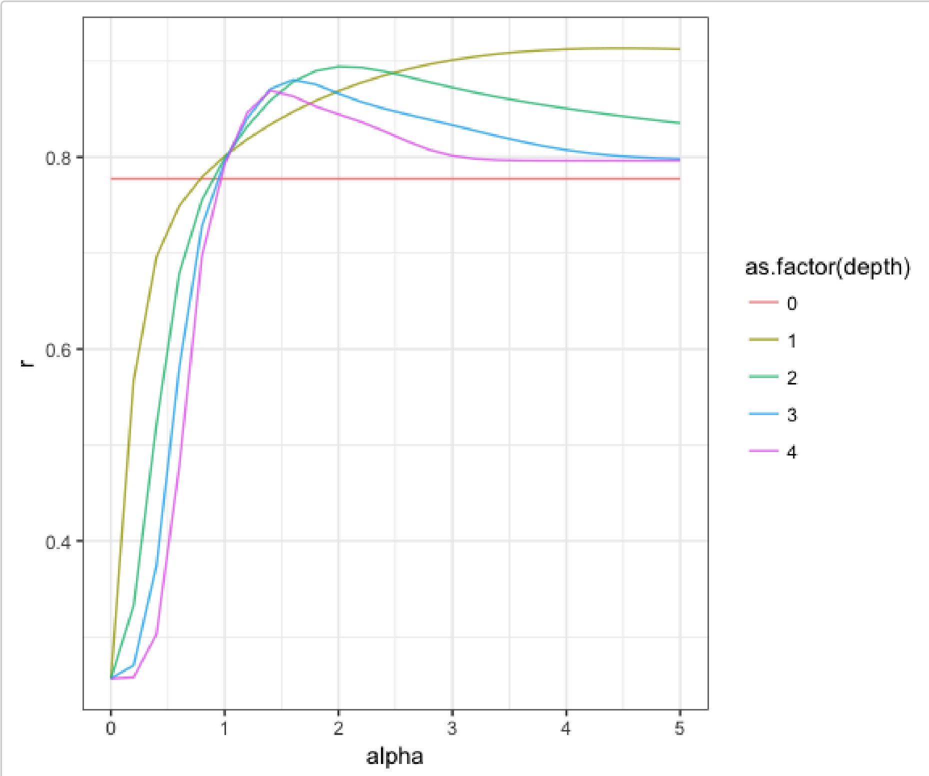
```
## hyperparams
depths <- seq(0, 4)
alphas <- seq(0, 5, by=0.2)

## Grid search using nested for-loops
res <- c()
for (alpha in alphas) {
  for (depth in depths) {
    curr_out <- peloquinFrank_5Alts %>%
      split(.$scale) %>%
      map_df(~rsa.runDf(
        data=.x,
          quantityVarName="stars",
          semanticsVarName="speaker.p",
          itemVarName="words",
          depth=depth,
          alpha=alpha)) %>%
      mutate(alpha=alpha,
             depth=depth)
    res <- rbind(res, curr_out)
  }
}
```

In this case, we're only interested in how the model predicts some of the items (entailment items; see Peloquin & Frank, 2016).

```
target_items <- c("good", "excellent",
                  "liked", "loved",
                  "memorable", "unforgettable",
                  "palatable", "delicious",
                  "some", "all")

d_cors <- res %>%
  filter(words %in% target_items) %>%
  group_by(alpha, depth) %>%
  summarise(r=cor(e11, preds)) %>%
  ungroup
```

How does the model respond to the hyperparameters?

```
ggplot(d_cors, aes(x=alpha, y=r, col=as.factor(depth))) +
  geom_line(alpha=0.8) +
  theme_bw()
```




```
d_cors %>%
  summarise(max_r=max(r),
            max_alpha=alpha[which.max(r)],
            max_depth=depth[which.max(r)]) %>%
  kable()
```

max_r	max_alpha	max_depth
0.9133501	4.4	1

Simulating pragmatic inference with data from “Rational speech act models of pragmatic reasoning in reference games” - Frank, et al. (Under Review)

Data description for Frank, et al. (Under Review)

`rrrsa` includes empirical data used in Frank et al. (Under Review) model simulations in the `rrrsa::d_pragmods` data frame. We provide users with access to the data and include a short example of running simulations.

```
head(d_pragmods) %>%
  kable()
```

X	cond	expt	matrix	prior	query	object	count	p	n	cil	cih	priorType	priorValu
1	0.33	baserate	simple	0.33-baserate	glasses	foil	1	0.0158730	63	0.0000285	0.0603907	foil	0.052631
2	0.33	baserate	simple	0.33-baserate	glasses	logical	11	0.1746032	63	0.0908007	0.2742637	logical	0.543859
3	0.33	baserate	simple	0.33-baserate	glasses	target	51	0.8095238	63	0.7070716	0.8969162	target	0.403508
4	0.33	baserate	simple	0.33-baserate	hat	foil	NA	NA	63	NA	NA	foil	0.052631

X	cond	expt	matrix	prior	query	object	count	p	n	cil	cih	priorType	priorValu
5	0.33	baserate	simple	0.33-baserate	hat	logical	NA	NA	63	NA	NA	logical	0.543859
6	0.33	baserate	simple	0.33-baserate	hat	target	NA	NA	63	NA	NA	target	0.403508

You'll notice some `NA` values. This is because empirical data were not measured for all items, however we can supply the model with literal semantics for those items. These are present in the `speaker.p` column.

For more detailed information on the data included in `d_pragmods` run.

```
?d_pragmods
```

RSA simulations

Unlike in the previous example, we'd like to run rsa via `rsa.runDf()` over individual experiments rather than scales. We've provided a grouping variable in the data frame (`grouper`) which allows us to do this.

We split by the grouping variable `grouper` and use `purrr::map_df()` which allows us to run `rsa.runDf()` over subsets of the data frame. This equivalent to subsetting `d_pragmods` by each factor in `grouper`, running `rsa.runDf` and `rbind()`ing the results.

```
d_preds_priors <- d_pragmods %>%
  split(list(.$grouper)) %>%
  map_df(~rsa.runDf(
    data=.x,
    quantityVarName="object",
    semanticsVarName="speaker.p",
    itemVarName="query",
    priorsVarName = "priorValue",
    depth=1,
    alpha=1)) %>%
  mutate(keep_indices = ifelse(!is.na(count), "keep", "throwout")) %>%
  filter(keep_indices=="keep")
```

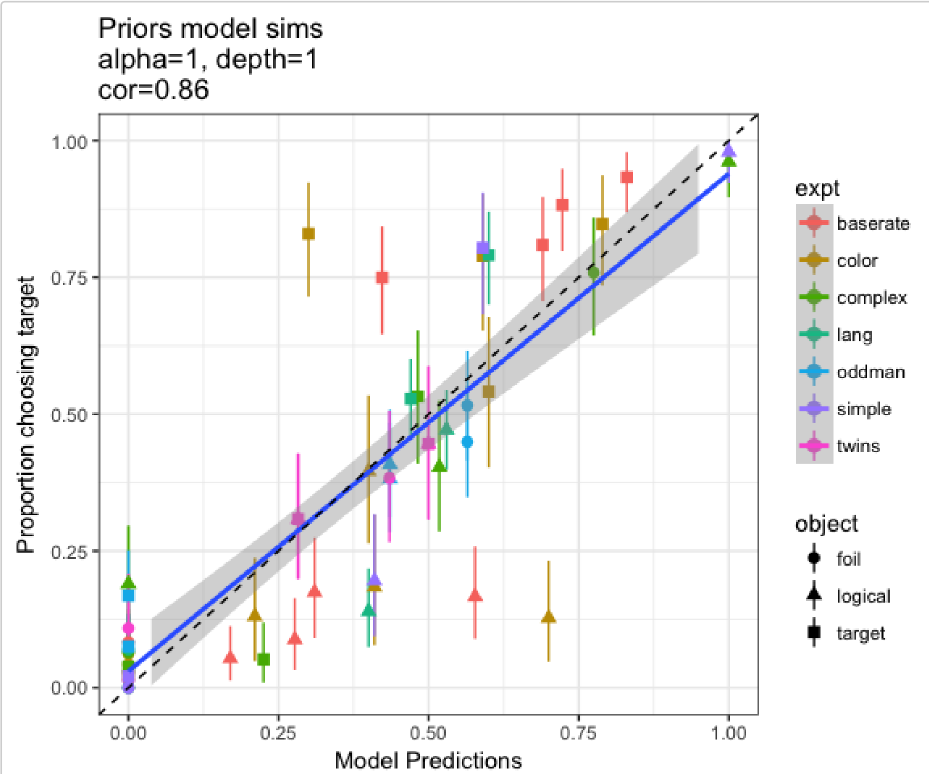
```
## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector
```

Notice that we're not interested in all the data here. We only collected empirical judgments for some of the items, but `rrrsa` requires that we include all the alternatives. We handle this by identifiying all the rows that contain `NA`s after running the simulations and removing them.

How does our model do?

```
ggplot(d_preds_priors, aes(x=preds, y=p, col=expt, pch=object)) +
  geom_pointrange(aes(ymin = cil, ymax = cih)) +
  xlim(c(0,1)) + ylim(c(0,1)) +
```

```
  ylab("Proportion choosing target") +
  xlab("Model Predictions") +
  geom_smooth(method="lm", aes(group=1)) +
  geom_abline(slope = 1, intercept = 0, lty=2) +
  ggtitle(paste0("Priors model sims\nalpha=1, depth=1\n",
                 "cor=",
                 round(cor(d_preds_priors$p, d_preds_priors$preds), 2))) +
  theme_bw()
```



This data corresponds with the second row of Table 2 in Frank, et al. (Under Review).

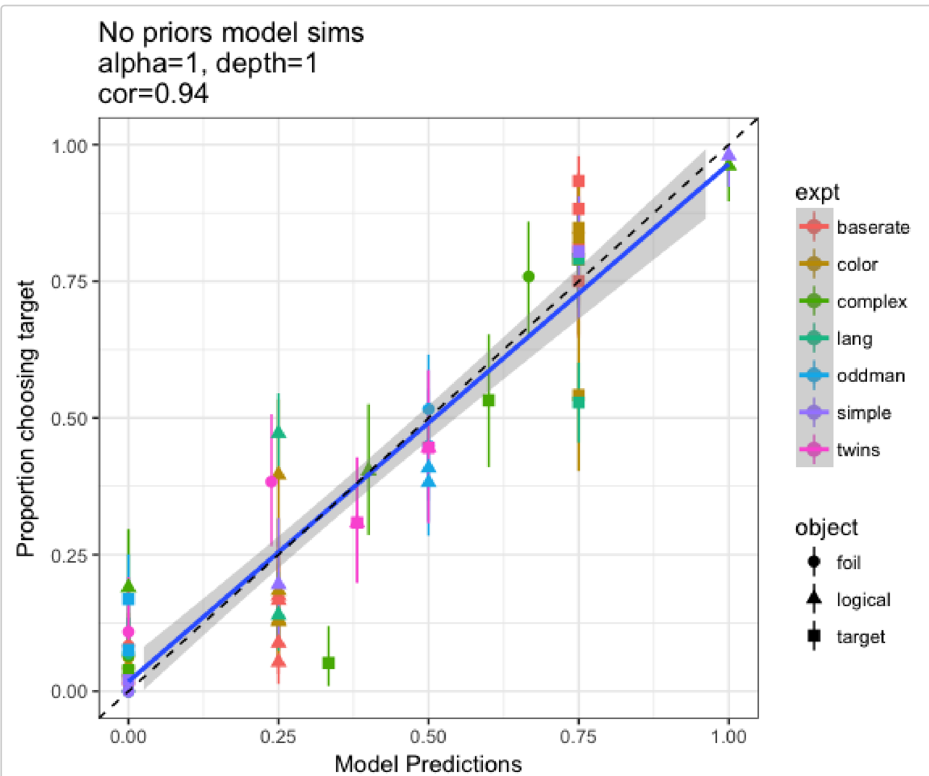
We can re-run without including priors by simply omitting the `priors` column name, in this case `priorValue`.

```
d_preds_no_priors <- d_pragmods %>%
  split(list(.$grouper)) %>%
  map_df(~rsa.runDf(
    data=.x,
    quantityVarName="object",
    semanticsVarName="speaker.p",
    itemVarName="query",
    # priorsVarName = "priorValue",
    depth=1,
    alpha=1)) %>%
  mutate(keep_indices = ifelse(!is.na(count), "keep", "throwout")) %>%
  filter(keep_indices=="keep")
```

```
## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector
```

```
## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector
```

```r
ggplot(d_preds_no_priors, aes(x=preds, y=p, col=expt, pch=object)) +
  geom_pointrange(aes(ymin = cil, ymax = cih)) +
  xlim(c(0,1)) + ylim(c(0,1)) +
  ylab("Proportion choosing target") +
  xlab("Model Predictions") +
  geom_smooth(method="lm", aes(group=1)) +
  geom_abline(slope = 1, intercept = 0, lty=2) +
  ggtitle(paste0("No priors model sims\nalpha=1, depth=1\n",
                 "cor=",
                 round(cor(d_preds_no_priors$p, d_preds_no_priors$preds), 2))) +
  theme_bw()
```



This data corresponds with the second row of Table 4 in Frank, et al. (Under Review).