Modeling in the Tidyverse

Max Kuhn (RStudio)

Modeling in R

- R has always had a rich set of modeling tools that it inherited from S. For example, the formula interface has made it simple to specify potentially complex model structures.

- *R has cutting edge models*. Many researchers in various domains use R as their primary computing environment and their work often results in R packages.

- *It is easy to port or link to other applications*. R doesn't try to be everything to everyone. If you prefer models implemented in C, C++, `tensorflow`, `keras`, `python`, `stan`, or `Weka`, you can access these applications without leaving R.

However, there is a huge *consistency pr*
For example:

- There are two primary methods for specify
terms are in a model. Not all models have
- 99% of model functions automatically gen
dummy variables.
- Sparse matrices can be used (unless the c
- Many package developers don't know muc
language and omit OOP and other core R c

Two examples follow...

Between-Package Inconsistency

Syntax for computing predicted class probabilities:

Function	Package	Code
`lda`	`MASS`	`predict(obj)`
`glm`	`stats`	`predict(obj, type = "response")`
`gbm`	`gbm`	`predict(obj, type = "response", n.trees)`
`mda`	`mda`	`predict(obj, type = "posterior")`
`rpart`	`rpart`	`predict(obj, type = "prob")`
`Weka`	`RWeka`	`predict(obj, type = "probability")`
`logitboost`	`LogitBoost`	`predict(obj, type = "raw", nIter)`
`pamr.train`	`pamr`	`pamr.predict(obj, type = "posterior")`

Within-Package Inconsistency: `glmnet` Predictions

The `glmnet` model can be used to fit regularized generalized linear models with a mixture of L_1

penalties.

We'll look at what happens when we get predictions for a regression model (i.e. numeric *Y*) as w

classification models where *Y* has two or three categorical values.

The models shown below contain solutions for three regularization values (λ).

The predict method gives the results for all three at once (👍).

Numeric `glmnet` Predictions

Predicting a numeric outcome for two new data points:

```
new_x
```

```
##              x1     x2     x3     x4
## sample_1 1.649 -0.483 -0.294 -0.815
## sample_2 0.656 -0.420  0.880  0.109
```

```
predict(reg_mod, newx = new_x)
```

```
##            s0   s1 s2
## sample_1 9.95 9.95 10
## sample_2 9.95 9.95 10
```

A matrix result and we will assume that the λ values are in the same order as what we gave to model fit function.

`glmnet` Class Predictions

Predicting an outcome with two classes:

```
predict(two_class_mod, newx = new_x, type = "class")
```

```
##          s0  s1  s2
## sample_1 "a" "b" "b"
## sample_2 "a" "b" "b"
```

Not factors! That's different from what is required for the `y` argument. From `?glmnet`:

> For `family="binomial"` [`y`] should be either a factor with two levels, or a two-column ma
> counts or proportions

I'm guessing that this is because they want to keep the result a matrix (to be consistent).

`glmnet` Class Probabilities (Two Classes)

```
predict(two_class_mod, newx = new_x, type = "response")
```

```
##              s0  s1    s2
## sample_1 0.5 0.5 0.506
## sample_2 0.5 0.5 0.526
```

Okay, we get a matrix of the probability for the *second* level of the outcome factor.

To make this fit into most code, we can manually calculate the other probability. No biggie!

glmnet Class Probabilities (Three Classes)

```r
predict(three_class_mod, newx = new_x,
        type = "response")
```

```
## , , s0
##
##                a     b     c
## sample_1 0.333 0.333 0.333
## sample_2 0.333 0.333 0.333
##
## , , s1
##
##                a     b     c
## sample_1 0.333 0.333 0.333
## sample_2 0.333 0.333 0.333
##
## , , s2
##
##                a     b     c
## sample_1 0.373 0.244 0.383
## sample_2 0.327 0.339 0.334
```

No more matrix results. 3D array and w
the probabilities back this time.

Maybe a structure like this would work

```
## # A tibble: 6 x 4
##       a     b     c lambda
##   <dbl> <dbl> <dbl>  <dbl>
## 1 0.333 0.333 0.333   1
## 2 0.333 0.333 0.333   1
## 3 0.333 0.333 0.333   0.1
## 4 0.333 0.333 0.333   0.1
## 5 0.373 0.244 0.383   0.01
## 6 0.327 0.339 0.334   0.01
```

What We Need

Unless you are doing a simple one-off data analysis, the lack of consistency between, and som
within, R packages can be very frustrating.

If we could agree on a set of common conventions for interfaces, return values, and other comp
everyone's life would be easier.

Once we agree on conventions, **two challenges** are:

- As of 8/2018, there are over 12K R packages on CRAN. How do we "harmonize" these without
  everything?
- How can we guide new R users (or people unfamiliar with R) in making good choices in thei
  modeling packages?

These prospective and retrospective problems will be addressed in a minute.

The Tidyverse

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

The principles of the tidyverse:

1. Reuse existing data structures.
2. Compose simple functions with the pipe.
3. Embrace functional programming.
4. Design for humans.

This results in more specific conventions around interfaces, function naming, etc. For example:

```
## [1] "glue_col"      "glue_collapse"
## [3] "glue_data"     "glue_data_col"
## [5] "glue_data_sql" "glue_sql"
```

There is also the notion of tidy data:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a ta

Based on these ideas, we can create m
packages that have predictable results
pleasure to use.

Tidymodels

`tidymodels` is a collection of modeling packages that live in the tidyverse and are designed in t
way.

My goals for tidymodels are:

1. Encourage empirical validation and good methodology.
2. Smooth out diverse interfaces.
3. Build highly reusable infrastructure.
4. Enable a wider variety of methodologies.

The `tidymodels` packages address the *retrospective* and *prospective* issues. We are also develop
of principles and templates to make *prospective* (new R packages) easy to create.

Current Modeling Packages

- `broom` takes the messy output of built-in functions in R, such as `lm`, `nls`, or `t.test`, and turns them into tidy data
- `dials` has tools for creating and validating tuning parameter values.
- `infer` is a modern approach to statistical inference.
- `recipes` is a general data preprocessor with a modern interface. It can create model matrices that incorporate fea engineering, imputation, and other tools.
- `rsample` has infrastructure for *resampling* data so that models can be assessed and empirically validated.
- `tidyposterior` can be used to compare models using resampling and Bayesian analysis.
- `tidytext` contains tidy tools for quantitative text analysis, including basic text summarization, sentiment analysis, modeling.
- `yardstick` contains tools for evaluating models (e.g. accuracy, RMSE, etc.)

More on the way... blog post.

Loading the Meta-Package

```r
library(tidymodels)
```

```
## ── Attaching packages ──────────────────────────────────── tidymodels 0.0.1 ──
```

```
## ✔ ggplot2   3.0.0     ✔ recipes   0.1.3
## ✔ tibble    1.4.2     ✔ broom     0.5.0
## ✔ purrr     0.2.5     ✔ yardstick 0.0.1
## ✔ dplyr     0.7.6     ✔ infer     0.3.1
## ✔ rsample   0.0.2
```

```
## ── Conflicts ───────────────────────────────────── tidymodels_conflicts() ──
## ✖ purrr::accumulate()      masks foreach::accumulate()
## ✖ dplyr::collapse()        masks glue::collapse()
## ✖ Biobase::combine()       masks BiocGenerics::combine(), dplyr::combine()
## ✖ rsample::fill()          masks tidyr::fill()
## ✖ dplyr::filter()          masks stats::filter()
## ✖ dplyr::lag()             masks stats::lag()
## ✖ BiocGenerics::Position() masks ggplot2::Position(), base::Position()
## ✖ recipes::step()          masks stats::step()
## ✖ purrr::when()            masks foreach::when()
```

`broom` Example

Model fit from `?lm`

```
summary(lm.D9)$coefficients
```

```
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    5.032      0.220   22.85 9.55e-15
## groupTrt      -0.371      0.311   -1.19 2.49e-01
```

```
broom::tidy(lm.D9)
```

```
## # A tibble: 2 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)    5.03      0.220     22.9  9.55e-15
## 2 groupTrt      -0.371     0.311     -1.19 2.49e- 1
```

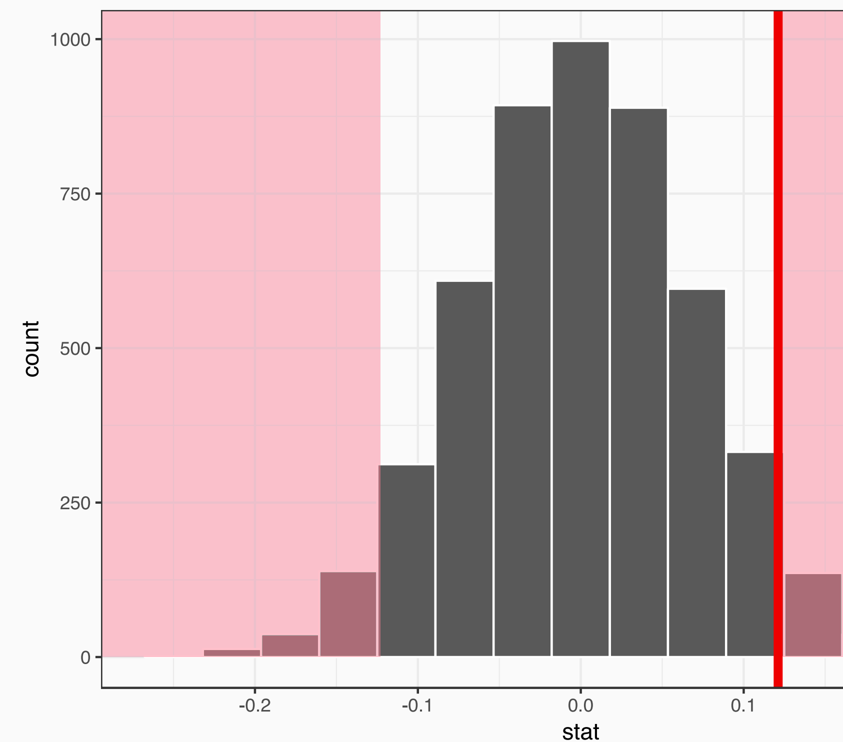
Find the differences…

Evaluating Hypotheses via `infer`

```r
library(caret)
data(BloodBrain)

dat <-
  data.frame(
    mol_weight = bbbDescr$mw,
    log_ratio = logBBB
  )

set.seed(3555)
perms <-
  dat %>%
  specify(log_ratio ~ mol_weight) %>%
  hypothesize(null = "independence") %>%
  generate(reps = 5000, type = "permute") %>%
  calculate(stat = "correlation", method = "spearman")

observed <-
  dat %>%
  specify(log_ratio ~ mol_weight) %>%
  calculate(stat = "correlation", method = "spearman")

perms %>% get_pvalue(obs_stat = observed, direction = "two_sided")
```

```
## # A tibble: 1 x 1
##   p_value
##     <dbl>
## 1  0.0854
```

```r
perms %>%
  visualize(obs_stat = observed, direction = "two_sided")
```



Recipes for Preprocessing Data

`recipes` provides a `dplyr`-like utility for

- Defining roles of variables in a model (e.g. outcome, predictor, etc).
- One or more *steps* are specified that do various types of operations, such as centering, imputation, feature extraction, term specification, re-encodings, etc.

Using a recipe is a stage-wise process:

```
recipe()          {define}
   ⬇️                 ⬇️
prep()            {estimate}
   ⬇️                 ⬇️
bake()/juice()    {apply}
```

```
bbb_data <- bbbDescr %>% mutate(log_ratio =

rec <- recipe(log_ratio ~ ., data = bbb_dat
  step_nzv(all_predictors()) %>%
  step_corr(all_predictors(), threshold = 0
  step_YeoJohnson(all_predictors()) %>%
  step_interact(~ nbasic:rotatablebonds) %>
  step_center(all_predictors()) %>%
  step_scale(all_predictors()) %>%
  step_pca(all_predictors(), num = 3)

rec <- prep(rec, training = bbb_data)

bake(rec, bbb_data %>% slice(1:3))
```

```
## # A tibble: 3 x 4
##   log_ratio   PC1   PC2   PC3
##       <dbl> <dbl> <dbl> <dbl>
## 1      1.08  2.67 -7.88 -1.58
## 2     -0.4  -4.14 -2.15 -2.09
## 3      0.22  1.08  1.86  1.91
```

Comparing Models Using Resampling and Bayes

`tidyposter` and `rsample` can be used to make comparisons between and within types of models.

A model is resampled and it's performance metrics (e.g. R^2, ROC AUC, etc.) can be used as the *outcome* in a Bayesian meta-model. From this, posteriors for the differences can be computed.

Let's say that I have these two models:

```
coxph(Surv(time, status) ~ ph.ecog + age + sex, data = lung)
coxph(Surv(time, status) ~ ph.ecog + age,       data = lung)
```

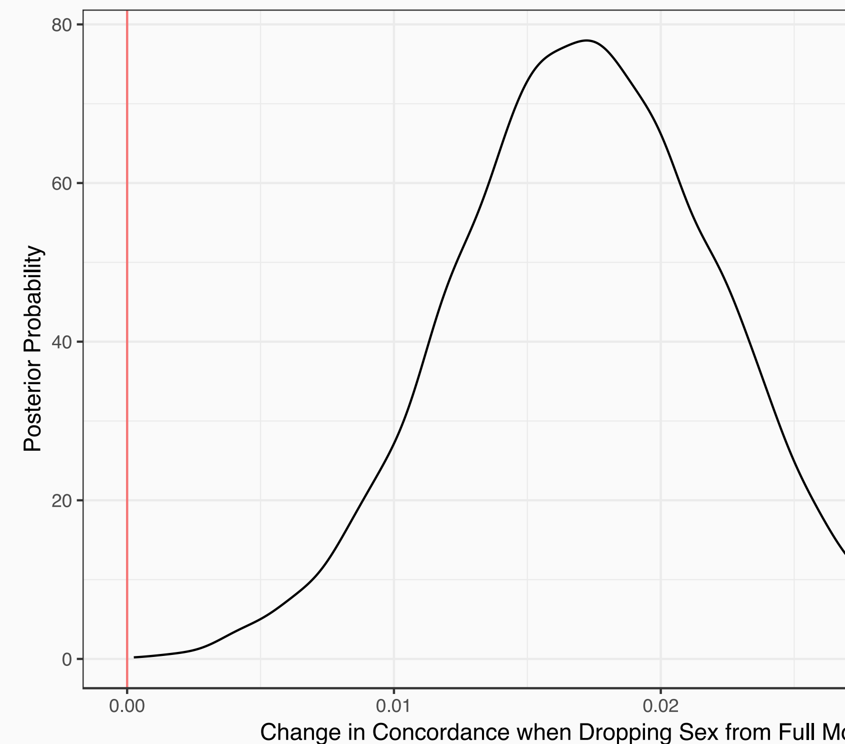
I can compare them using a standard h
model comparison or a simple Wald-ty
On this case, the Wald p-value is 0.000
doesn't tell me much about the *effect s*

Suppose I look at the change in the sta
concordance statistic between the two
What is the distribution of the *change*
concordance when I remove `sex`?

Cross-validation can be used to compu
difference on out-of-sample data and a
model can be used to compute the pos
the difference.

Results

```
## #  10-fold cross-validation repeated 10 times using stratif
## # A tibble: 100 x 5
##    id       id2    full_model without_sex diff
##  * <chr>    <chr>       <dbl>       <dbl> <chr>
##  1 Repeat01 Fold01      0.701       0.687 worse  by 0.01
##  2 Repeat01 Fold02      0.570       0.554 worse  by 0.02
##  3 Repeat01 Fold03      0.731       0.702 worse  by 0.03
##  4 Repeat01 Fold04      0.599       0.550 worse  by 0.05
##  5 Repeat01 Fold05      0.594       0.528 worse  by 0.07
##  6 Repeat01 Fold06      0.742       0.663 worse  by 0.08
##  7 Repeat01 Fold07      0.525       0.514 worse  by 0.01
##  8 Repeat01 Fold08      0.714       0.674 worse  by 0.04
##  9 Repeat01 Fold09      0.491       0.549 better by 0.06
## 10 Repeat01 Fold10      0.764       0.692 worse  by 0.07
## # ... with 90 more rows
```



A positive difference would imply that
important in explaining the outcome.

Principles of Modeling Packages

We are in the process of developing a set of *guidelines* for making good modeling packages. For example:

- Separate the interface that the **modeler** uses from the code to do the computations. They serve two v different purposes.
- Have multiple interfaces (e.g. formula, x/y, etc).
- The *user-facing interface* should use the most appropriate data structures for the data (as opposed t computations). For example, factor outcomes versus 0/1 indicators and data frames versus matrices.
- `type = "prob"` for class probabilities 😄.
- Use S3 methods.
- The `predict` method should give standardized, predictable results.

Rather than try to make methodologists into software developers, we will provide GitHub repositories wit packages that can be used to meet these guidelines (along with documentation and examples on *why*).

Next Steps

- Hash out the principles of modeling functions. Let me know if you'd like to contribute.
- Packages on the horizon:
  - `parsnip`: a unified interface to models. This should significantly reduce the amount of syntactical
    that you'll need to memorize by having one standardized model function across different package
    harmonizing the parameter names across models.
  - `embed`: an add-on package for `recipes`. This can be used to efficiently encode high-cardinality ca
    predictors using supervised methods such as likelihood encodings and entity embeddings.
  - A pipeline(ish) structure that can contain specifications for a model, recipe, feature filter, and pos
    processing. This will easily enable a *data analysis process*.
  - A model tuning package with grid search, Bayesian optimization, and other search algorithms.
  - A calibration package for post-processing regression and classification predictions as well as impl
    equivocal zones.

Thanks!