So what are we talking about? tidymodels are an integrated, modular, extensible set of packages that implement a framework that facilitates creating predicative stochastic models. tidymodels are first class members of the tidyverse. They adhere to tidyverse syntax and design principles that promote consistency and well-designed human interfaces over speed of code execution. Nevertheless, they automatically build in parallel execution for tasks such as resampling, cross validation and parameter tuning. Moreover, they don’t just work through the steps of the basic modeling workflow, they implement conceptual structures that make complex iterative workflows possible *and* reproducible.

If you are an R user and you have building predictive models then there is a good chance that you are familiar with the [caret](https://cran.r-project.org/package=caret) package. One straightforward path to investigate tidymodels is to follow the thread that leads form caret to [parsnip](https://cran.r-project.org/package=parsnip). caret, the result of a monumental fifteen year plus effort, incorporates two hundred thirty-eight [predictive models](https://topepo.github.io/caret/available-models.html) into a common framework. For example, any one of the included models can be substituted for lm in the following expression.

lmFit <- train(Y ~ X1 + X2, data = training,

method = "lm",

trControl = fitControl)

By itself this is a pretty big deal. parsnip refines this idea by creating a specification structure that identifies a class of models that allows users to easily change algorithms and also permits the models to run on different “engines”.

spec\_lin\_reg <- linear\_reg() %>% # a linear model specification

set\_engine( "lm") # set the model to use lm

# fit the model

lm\_fit <- fit(spec\_lin\_reg, Y ~ X1 + X2, data = my\_data)

This same specification can be modified to run a Bayesian model using [Stan](https://mc-stan.org/), or any number of other linear model backends such as glmnet, keras or spark.

spec\_stan <-

spec\_lin\_reg %>%

set\_engine("stan", chains = 4, iter = 1000) # set engine specific arguments

fit\_stan <- fit(spec\_stan, Y ~ X1 + X2, data = my\_data)

On its own, parnsnip provides a time saving framework for exploring multiple models. It is really nice not to have to worry about the idiosyncratic syntax developed for different model algorithms. But, the real power of tidymodels is baked into the [recipes](https://cran.r-project.org/package=parsnip) package. Recipes are structures that bind a sequence of preprocessing steps to a training data set. They define the roles that the variables are to play in the design matrix, specify what data cleaning needs to take place, and what feature engineering needs to happen.

To see how all of this comes together, lets look at recipe used in the tidymodels [recipes](https://www.tidymodels.org/start/recipes/) tutorial that uses the New York City flights data set, [nycflights13](https://cran.r-project.org/package=nycflights13). We assume that all of the data wrangling code in the tutorial has been executed, and we pick up with the code to define the recipe:

flights\_rec <-

recipe(arr\_delay ~ ., data = train\_data) %>%

update\_role(flight, time\_hour, new\_role = "ID") %>%

step\_date(date, features = c("dow", "month")) %>%

step\_rm(date) %>%

step\_dummy(all\_nominal(), -all\_outcomes())

The first line identifies the variable arr\_delay as the variable to be predicted and the other variables in the data set train\_data to be predictors. The second line amends that by updating the roles of the variables flight and time\_hour to be identifiers and not predictors. The third and fourth lines continue with the feature engineering by creating a new date variable and removing the old one. The last line explicitly converts all categorical or factor variables into binary dummy variables.

The recipe is ready to be evaluated, but if a modeler thought that she might want to keep track of this workflow for the future, she might bind the recipe and model together in a workflow() that saves everything as a reproducible unit with a command something like this.

lr\_mod <- logistic\_reg() %>% set\_engine("glm")

flights\_wflow <-

workflow() %>%

add\_model(lr\_mod) %>%

add\_recipe(flights\_rec)

Then, fitting the model is just a matter calling fit with the workflow as a parameter.

flights\_fit <- fit(flights\_wflow, data = train\_data)

At this point, everything is in place to complete a statistical analysis. A modeler can extract coefficients, p-values etc., calculate performance statistics, make statistical inferences and easily save the workflow in a reproducible markdown document. However the real gains from tidymodels become apparent when the modeler goes on to build predictive models.

The [Kuhn and Johnson (2019)](https://bookdown.org/max/FES/resampling.html) illustrates a typical predictive modeling workflow.

It indicates that before going on to predict model performance on new data (the test set), a modeler will want to make use of cross validation or some other resampling technique to first evaluate the performance of multiple candidate models, and then tune the selected model. This is where the great power of the recipe() and workflow() constructs becomes apparent. In addition, to encouraging experiments with multiple models by rationalizing algorithm syntax, providing interchangeable model constructs, and enabling modelers to grow chains of recipe steps with the pipe operator; recipies *helps to enforce good statistical practice*.

For example, although it is common practice to split the available data between training and test sets before preprocessing the training data set, it is also very common to see pipelines where data preparation is applied to the entire training set at one go. It is not common to see data cleaning and preparation processes individually applied to each fold of a ten-fold cross validation effort. But, that is exactly the right thing to do to mitigate the deleterious effects of data imputation, centering and scaling and numerous other preparation steps that contribute to bias and limit the predictive value of a model. This is the whole point of resampling, but it is not easy to do in a way that saves necessary intermediate artifacts, and provides a reproducible set of instructions for others on the modeling team.

Because, *recipes are not evaluated until the model is fit* tidymodel workflows make an otherwise laborious and error prone process very straightforward. This is a game changer!

The next two lines of code set up and execute ten-fold cross-validation for our example.

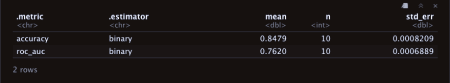
set.seed(123)

folds <- vfold\_cv(train\_data, v = 10)

flights\_fit\_rs <- fit\_resamples(flights\_wflow, folds)

And then, another line of code collects the metrics over the folds and prints out the statistics for accuracy and area under the ROC curve.

collect\_metrics(flights\_fit\_rs)



So, here we are with a mediocre model, and I’ll stop now having shown you only a small portion of what tidymodels can do