Intro to tidymodels

Ari Weil

5/21/2021

Intro

There are a lot of packages available for machine learning in R. Today I'll introduce one library I've used: `tidymodels`. `tidymodels` has a few benefits:

- Loads several modeling packages in just one library.
- Uses tidy syntax and easily fits into your broader tidyverse operations (ie using the pipe).
- Simple to fit different models using the same workflow and syntax, and then easily compare metrics across them.

However, one challenge you'll see is that it can be overly complex and involve too many steps if you're just running a simple model. So overall, I recommend it if you need to split data, are running multiple models, need to tune parameters, and/or want to compare metrics across multiple models.

Tidymodeling focuses on three steps:

- Pre-process: transform the data (scale, normalize, etc.) and split the data into testing and training sets. For more on train/test, see Pete's tutorial.
- Train: fit the model to the training set. Tune parameters as necessary, and then fit the best model.
- Validate: fit the model to the testing set, and then evaluate performance metrics.

My focus today will be to introduce the library's main functions and give some sample code, rather than focusing on specific models or approaches to machine learning (supervised/unsupervised). We'll train and tune three models: linear regression, k nearest neighbors regression, and a decision tree.

Code

Load Library

```
# Load tidymodels (make sure to install first if you don't have it)
library(tidymodels)
```

```
# View packages currently loaded
(.packages())
```

```
##  [1] "yardstick"  "workflows"  "tune"       "tidyr"      "tibble"
##  [6] "rsample"    "recipes"    "purrr"      "parsnip"    "modeldata"
## [11] "infer"      "ggplot2"    "dplyr"      "dials"      "scales"
## [16] "broom"      "tidymodels" "stats"      "graphics"   "grDevices"
## [21] "utils"      "datasets"   "methods"    "base"
```

```
# Also add two more for timing and parallel processing
library(tictoc)
library(doParallel)
```

Another benefit of tidymodels is that it contains many other packages. By loading in only tidymodels, we also get the commonly used tidy packages `dplyr` and `ggplot2`.

For modeling, `tidymodels` includes three core packages:

- `rsample`: for splitting the data into train and test.
- `parsnip`: used to actually build and fit the model.
- `yardstick`: for evaluating the accuracy and fit of your models.

For additional modeling needs, there is also:

- `workflow`: combines pre-processing (recipe), modeling, and post-processing into one object.
- `tune`: great for tuning model parameters and hyperparameters (such as neighbors in kNN or minimum number of trees in a decision tree).

Pre-process

We'll use the diamonds dataset from `ggplot2`. For the example today, we will be trying to predict the price of a diamond using the other variables (carat, cut, color, etc).

```
diamonds <- ggplot2::diamonds
```

Splitting Data

I'll demo three ways to split the data in `rsample`: proportion split, stratified sample, or k-fold cross validation.

```
# Set seed for reproducibility
set.seed(1234)

## 80-20 train-test split
split <- initial_split(diamonds, prop = 0.8)

# initial_split then stores the two samples in training() and testing()
train <- split %>% training()
test <- split %>% testing()

# Check to see that our split is correct
dim(train)
```

```
## [1] 43153    10
```

```r
dim(test)
```

```
## [1] 10787    10
```

```r
## Stratified sample split
# This is useful in classification problems when one class is overly represented
split_strat <- initial_split(diamonds, prop = 0.8, strata = price)

## Cross-validation
# If you wanted to do k-fold cross-validation:
split_cv <- vfold_cv(train, v = 10)
```

Recipe

`tidymodels` uses what are called recipes for other pre-processing steps, and these functions come from the `recipes` package. The three key functions are:

- `recipe()`: build a series of data cleaning steps.
- `prep()`: apply the steps to the training data.
- `bake()`: apply the recipe to the testing data.

Note: I don't use prep/bake, but use workflows instead (more on that in a minute).

The key element I use recipes for is setting up your dependent and independent variables, with the dv ~ iv + iv2 + iv3 formula from `lm()`. But recipes are also great for any final data preparation steps you need for a model.

This can be done with the `step_` functions. For example, you can scale a variable with `step_center()` to scale the mean to 0 and `step_scale()` to scale the data to a standard deviation to 1.

```r
# Set recipe
# DV is price, and . means all other variables are IVs
recipe <- recipe(price ~ ., data = diamonds) %>%
  step_log(price) %>% # take log of outcome
  # Normalize all IVs except for the categorical variables
  step_normalize(all_predictors(), - all_nominal()) %>%
  # Make dummy variables out of the categorical variables
  step_dummy(all_nominal())
```

Train

The actual modeling process involves a model, engine, and workflow.

We'll use three models today:

1. Linear Regression: simple linear and parametric model predicting price based on all other variables.
2. k Nearest Neighbors: non-parametric and non-linear model. Takes in parameter "k", and then finds the k number of observations closest to each point, and averages to generate predictions.
3. Decision Tree: model that splits data repeatedly based on one independent variable at a time.

Create Model

```
## Linear Regression
# Note: this is a bit tedious for just a linear model, compared to lm().
lm_mod <- linear_reg() %>%
  set_mode("regression") %>%
  set_engine("lm")

## kNN
knn_mod <- nearest_neighbor(
  neighbors = 5) %>%
  set_mode("regression") %>%
  # can alternatively set mode to "classification"
set_engine("kknn")

## Decision Tree
tree_mod <- decision_tree() %>%
  set_mode("regression") %>%
  set_engine("rpart")
```

Define a Workflow

```
# Linear model workflow
lm_wf <- workflow() %>%
  add_recipe(recipe) %>%
  add_model(lm_mod)

# Let's see what's in the workflow
lm_wf
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## 3 Recipe Steps
##
## * step_log()
## * step_normalize()
## * step_dummy()
##
## -- Model -----------------------------------------------------------------------
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

```
# kNN workflow
knn_wf <- workflow() %>%
  add_recipe(recipe) %>%
  add_model(knn_mod)
```

```r
# Decision tree workflow
tree_wf <- workflow() %>%
  add_recipe(recipe) %>%
  add_model(tree_mod)
```

Run the Models

```r
# Run the linear model
lm_res <- lm_wf %>%
  fit_resamples(resamples = split_cv,
                control = control_resamples(save_pred = T))

# Use collect_metrics() to see the error rate (how well our model did)
lm_res %>% collect_metrics()
```

```
## # A tibble: 2 x 6
##   .metric .estimator  mean     n std_err .config
##   <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1 rmse    standard   0.173    10 0.0131  Preprocessor1_Model1
## 2 rsq     standard   0.970    10 0.00485 Preprocessor1_Model1
```

```r
# Run the kNN
knn_res <- knn_wf %>%
  fit_resamples(resamples = split_cv,
                control = control_resamples(save_pred = T))

knn_res %>% collect_metrics()
```

```
## # A tibble: 2 x 6
##   .metric .estimator  mean     n  std_err .config
##   <chr>   <chr>      <dbl> <int>    <dbl> <chr>
## 1 rmse    standard   0.159    10 0.00137  Preprocessor1_Model1
## 2 rsq     standard   0.976    10 0.000484 Preprocessor1_Model1
```

```r
# Run the decision tree
tree_res <- tree_wf %>%
  fit_resamples(resamples = split_cv,
                control = control_resamples(save_pred = T))

tree_res %>% collect_metrics()
```

```
## # A tibble: 2 x 6
##   .metric .estimator  mean     n std_err .config
##   <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1 rmse    standard   0.295    10 0.00297 Preprocessor1_Model1
## 2 rsq     standard   0.915    10 0.00189 Preprocessor1_Model1
```

```r
# Let's compare the three models
collect_metrics(lm_res) %>%
  bind_rows(collect_metrics(knn_res)) %>%
  bind_rows(collect_metrics(tree_res)) %>%
  filter(.metric == "rmse") %>%
  mutate(model = c("lm", "kNN", "tree")) %>%
  relocate(model) %>%
  arrange(mean)
```

```
## # A tibble: 3 x 7
##   model .metric .estimator  mean     n std_err .config
##   <chr> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1 kNN   rmse    standard   0.159    10 0.00137 Preprocessor1_Model1
## 2 lm    rmse    standard   0.173    10 0.0131  Preprocessor1_Model1
## 3 tree  rmse    standard   0.295    10 0.00297 Preprocessor1_Model1
```

Tuning

Now let's try tuning our kNN and decicision tree models. `tidymodels` makes it easy to vary hyperparameters and the evaluate each model.

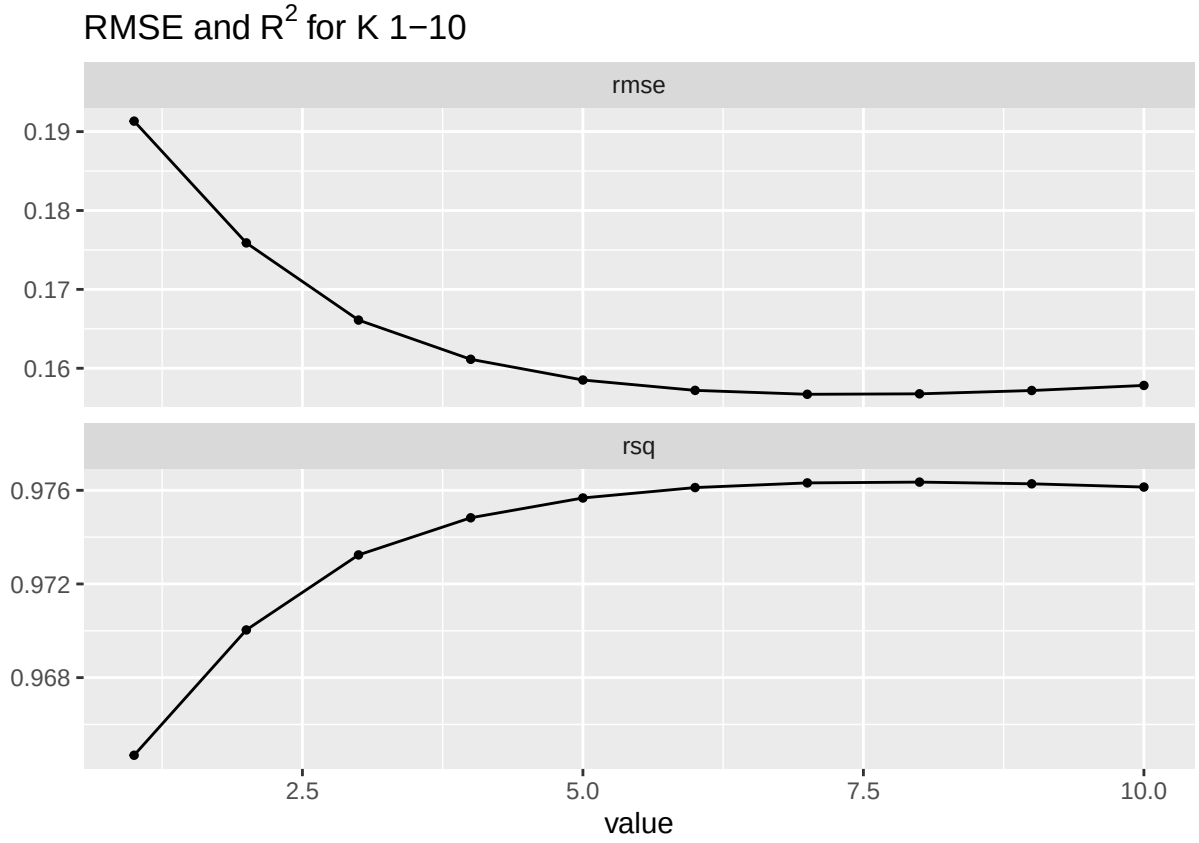
```r
# New knn model, with neighbors set to tune k
knn_mod2 <- nearest_neighbor(
  neighbors = tune("k")) %>%
  set_mode("regression") %>%
set_engine("kknn")

knn_wf2 <- workflow() %>%
  add_recipe(recipe) %>%
  add_model(knn_mod2)

# Set up grid for k to vary from 1 to 10
k_grid <- tibble(k = c(1:10))

# I just use this to save the outcomes--mostly useful for classification problems when you want to comp
ctrl <- control_grid(save_pred = TRUE)

# Run the models for k 1-10
set.seed(1234)
doParallel::registerDoParallel() # parallel processing for speed
tic() # time tuning

knn_tuning <- knn_wf2 %>%
  tune_grid(resamples = split_cv,
            grid = k_grid,
            control = ctrl)

toc()
```

```
## 288.23 sec elapsed
```

```r
# Took about 8 minutes normal, 4.8 min parallel

knn_metrics <- knn_tuning %>%
  collect_metrics()

# Let's look at the performance of the models
knn_metrics %>% dplyr::filter(.metric == 'rmse') %>%
  arrange(mean)
```

```
## # A tibble: 10 x 7
##        k .metric .estimator  mean     n std_err .config
##    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1     7 rmse    standard   0.157    10 0.00132 Preprocessor1_Model07
##  2     8 rmse    standard   0.157    10 0.00129 Preprocessor1_Model08
##  3     9 rmse    standard   0.157    10 0.00127 Preprocessor1_Model09
##  4     6 rmse    standard   0.157    10 0.00134 Preprocessor1_Model06
##  5    10 rmse    standard   0.158    10 0.00125 Preprocessor1_Model10
##  6     5 rmse    standard   0.159    10 0.00137 Preprocessor1_Model05
##  7     4 rmse    standard   0.161    10 0.00139 Preprocessor1_Model04
##  8     3 rmse    standard   0.166    10 0.00138 Preprocessor1_Model03
##  9     2 rmse    standard   0.176    10 0.00126 Preprocessor1_Model02
## 10     1 rmse    standard   0.191    10 0.00118 Preprocessor1_Model01
```

```r
# Looks like k = 7 is best

# Use autoplot to visualize accuracy metrics
autoplot(knn_tuning) + ggtitle(bquote("RMSE and"~R^2~ "for K 1-10"))
```




```
# Can now use select_best to pick the best model to use in future iterations
knn_best <- select_best(knn_tuning, metric = "rmse")
```

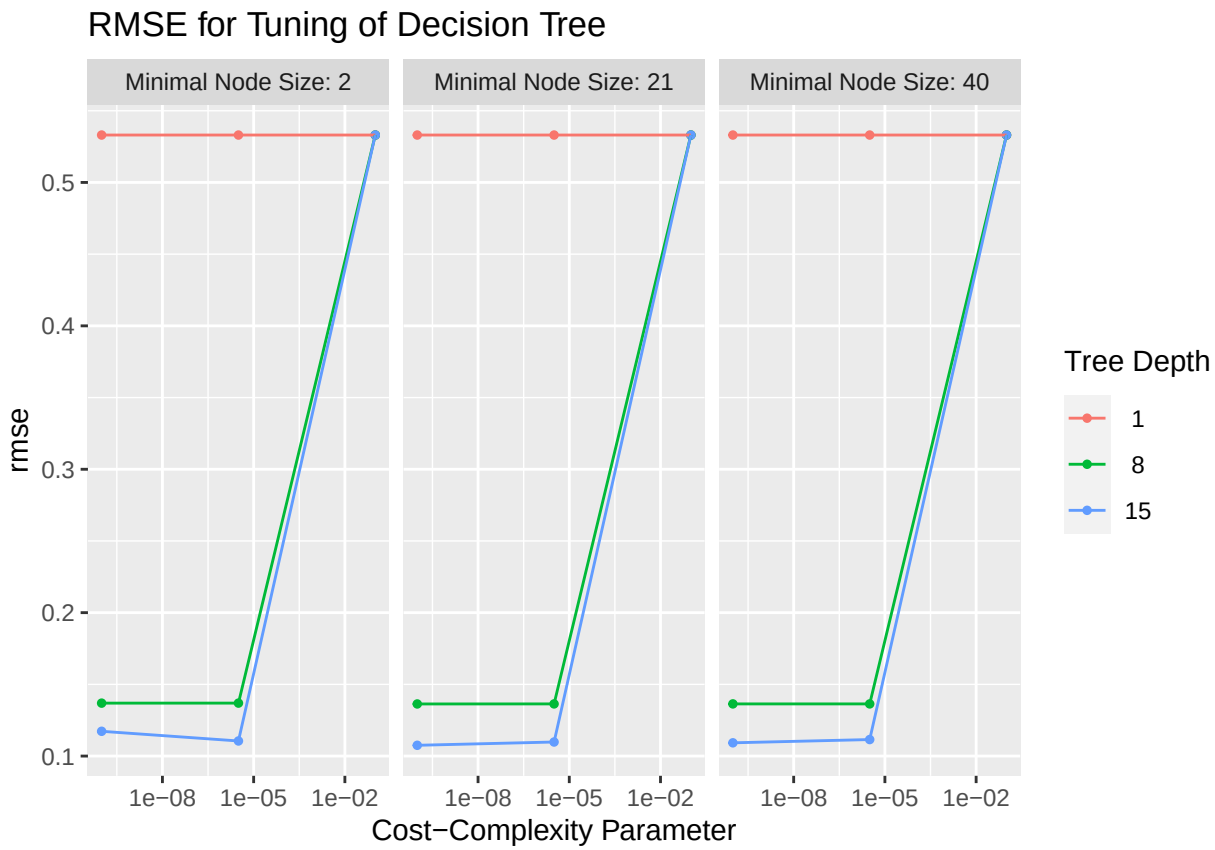
Let's tune our decision tree to show how we can tune multiple parameters at once.

```
# New decision tree model. Tuning:
  # tree_depth: how many nodes in the tree
  # min_n: how many data points before splitting
  # and cost complexity:
tree_mod2 <- decision_tree(
  cost_complexity = tune(),
  tree_depth = tune(),
  min_n = tune()) %>%
  set_mode("regression") %>%
set_engine("rpart")

tree_grid <- grid_regular(cost_complexity(),
                          tree_depth(),
                          min_n(),
                          levels = 3)

tree_wf2 <- workflow() %>%
  add_recipe(recipe) %>%
  add_model(tree_mod2)
```

```
set.seed(1234)
doParallel::registerDoParallel()
tic()

tree_tuning <- tree_wf2 %>%
  tune_grid(resamples = split_cv,
            grid = tree_grid)

toc() # 5.8 minutes
```

```
## 285.52 sec elapsed
```

```
# Visualize and compare performance across the parameter combinations
autoplot(tree_tuning, metric = "rmse") +
  ggtitle("RMSE for Tuning of Decision Tree")
```




```
# Tree depth of 1 doesn't improve, always has high error
# Tree depth of 15 has the lowest error
# And it looks like a low cost is best, so we want to keep model complex
# Lastly, 21 nodes performs best

# Let's see metrics in df form
tree_tuning %>% collect_metrics() %>%
  filter(.metric == "rmse") %>%
  arrange(mean)
```

```
## # A tibble: 27 x 9
##    cost_complexity tree_depth min_n .metric .estimator  mean     n std_err
##              <dbl>      <int> <int> <chr>   <chr>      <dbl> <int>   <dbl>
##  1    0.0000000001         15    21 rmse    standard   0.108    10 1.12e-3
##  2    0.0000000001         15    40 rmse    standard   0.109    10 8.68e-4
##  3    0.00000316           15    21 rmse    standard   0.110    10 9.83e-4
##  4    0.00000316           15     2 rmse    standard   0.111    10 1.08e-3
##  5    0.00000316           15    40 rmse    standard   0.112    10 7.76e-4
##  6    0.0000000001         15     2 rmse    standard   0.117    10 1.08e-3
##  7    0.0000000001          8    21 rmse    standard   0.136    10 5.54e-4
##  8    0.00000316            8    21 rmse    standard   0.136    10 5.56e-4
##  9    0.00000316            8    40 rmse    standard   0.136    10 5.44e-4
## 10    0.0000000001          8    40 rmse    standard   0.136    10 5.47e-4
## # ... with 17 more rows, and 1 more variable: .config <chr>
```

```
# Choose best
tree_best <- tree_tuning %>% select_best(metric = "rmse")
```

Validate

To finish, let's fit our best model to the test set adn see how it does. To do this, we'll select our best model, finalize our workflow, and then use `last_fit` to run the model on the test set.

```
# Finalize workflow (update it to use the best tuned model)
final_wf <- tree_wf2 %>%
  finalize_workflow(tree_best)

# Fit to the test set
set.seed(1234)
final_fit <- final_wf %>%
  last_fit(split)

# Evaluate test set performance
final_fit %>% collect_metrics()
```

```
## # A tibble: 2 x 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>          <dbl> <chr>
## 1 rmse    standard       0.108 Preprocessor1_Model1
## 2 rsq     standard       0.989 Preprocessor1_Model1
```

```
doParallel::stopImplicitCluster()
```

Final Notes

An additional `tidymodels` package, `usemodels`, writes model code for you! For example, here we specify our formula and model type (kNN), and it outputs a recipe, model, and workflow. You can then edit for any corrections/other needs.

```
library(usemodels)

use_kknn(price ~., data = diamonds)
```

```
## kknn_recipe <-
##   recipe(formula = price ~ ., data = diamonds) %>%
##   step_novel(all_nominal(), -all_outcomes()) %>%
##   step_dummy(all_nominal(), -all_outcomes()) %>%
##   step_zv(all_predictors()) %>%
##   step_normalize(all_predictors(), -all_nominal())
##
## kknn_spec <-
##   nearest_neighbor(neighbors = tune(), weight_func = tune()) %>%
##   set_mode("regression") %>%
##   set_engine("kknn")
##
## kknn_workflow <-
##   workflow() %>%
##   add_recipe(kknn_recipe) %>%
##   add_model(kknn_spec)
##
## set.seed(41964)
## kknn_tune <-
##   tune_grid(kknn_workflow, resamples = stop("add your rsample object"), grid = stop("add number of ca
```

Resources

- The Tidy Modeling with R book
- Julia Silge's blog is a great resource for `tidymodels` tips. Every other Tuesday she uploads a demo.