Modeling and Machine Learning in R: tidymodels

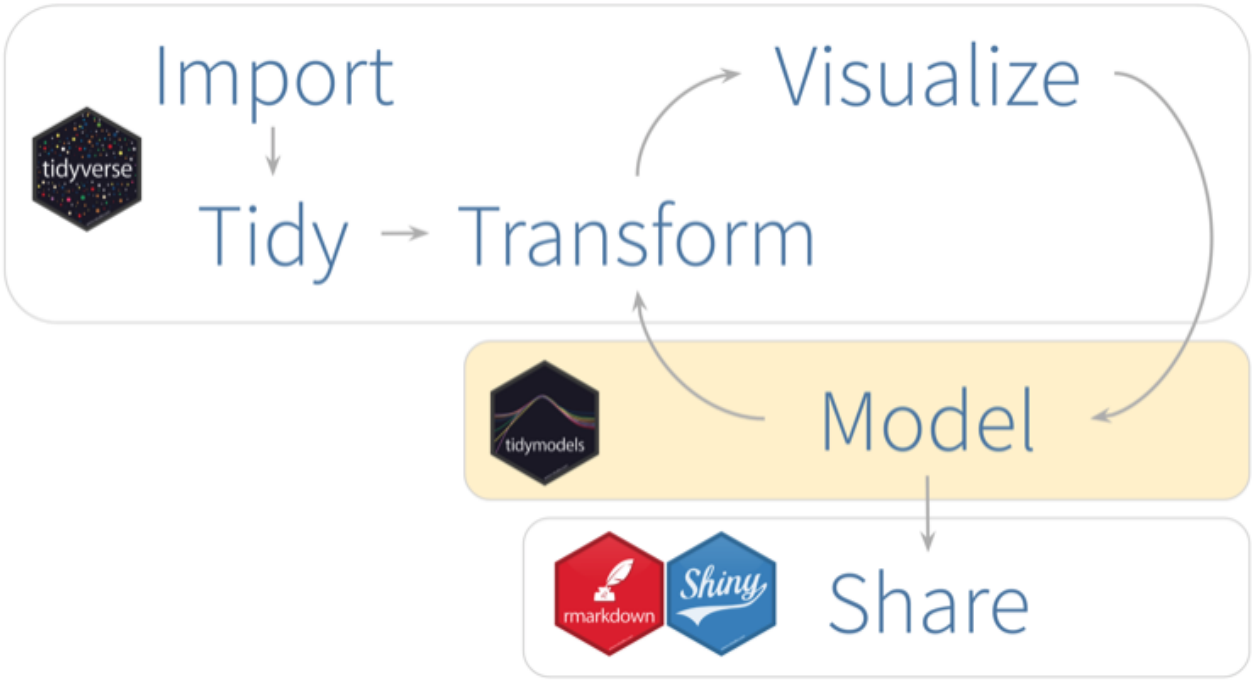
TIAGO SOUZA

Acknowledgements

The practical example in the slides is based on a blog post **Tutorial on tidymodels for Machine Learning** by Hansjörg Plieninger.

https://hansjoerg.me/2020/02/09/tidymodels-for-machine-learning/

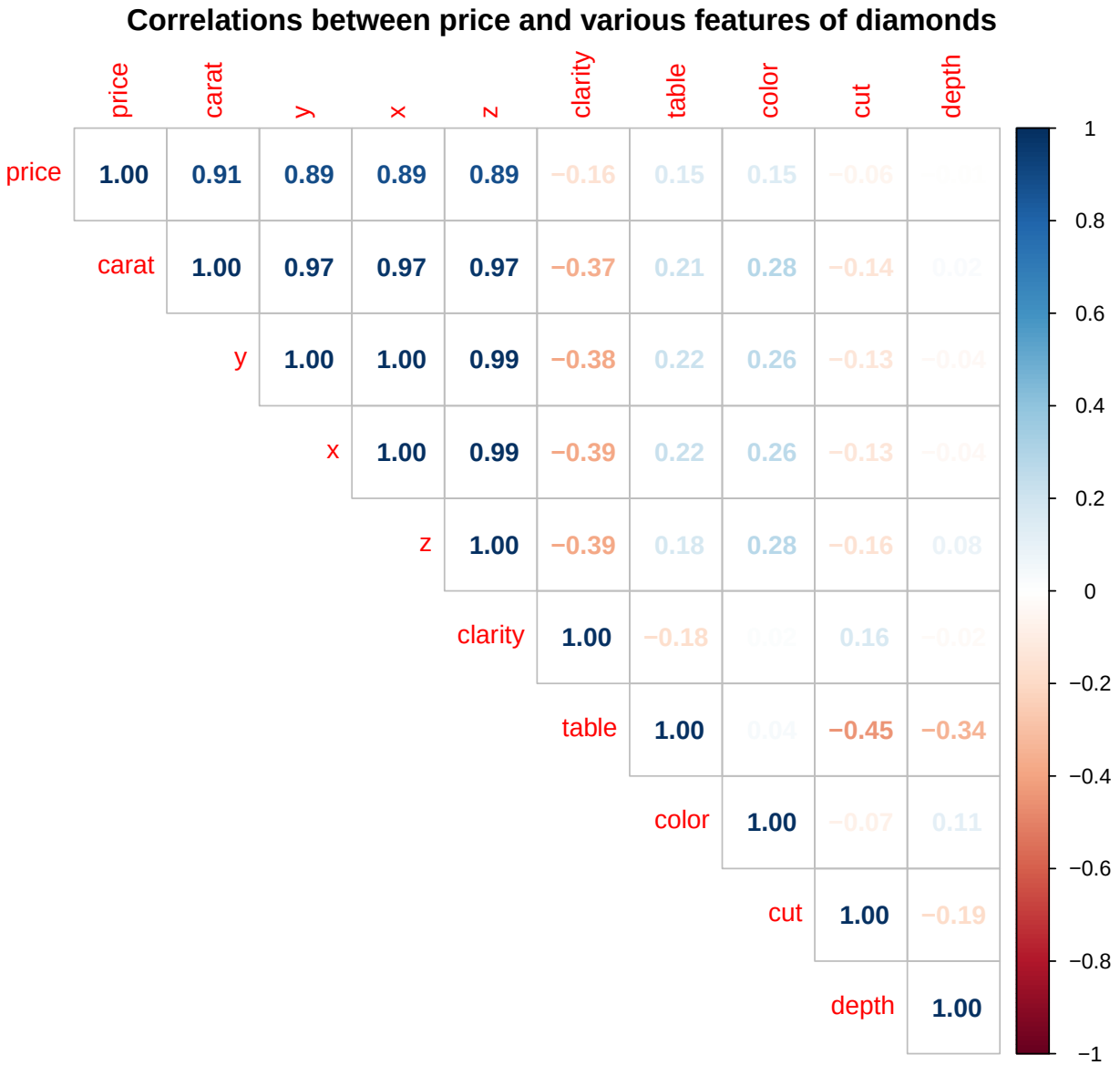
tidymodels within the R Universe



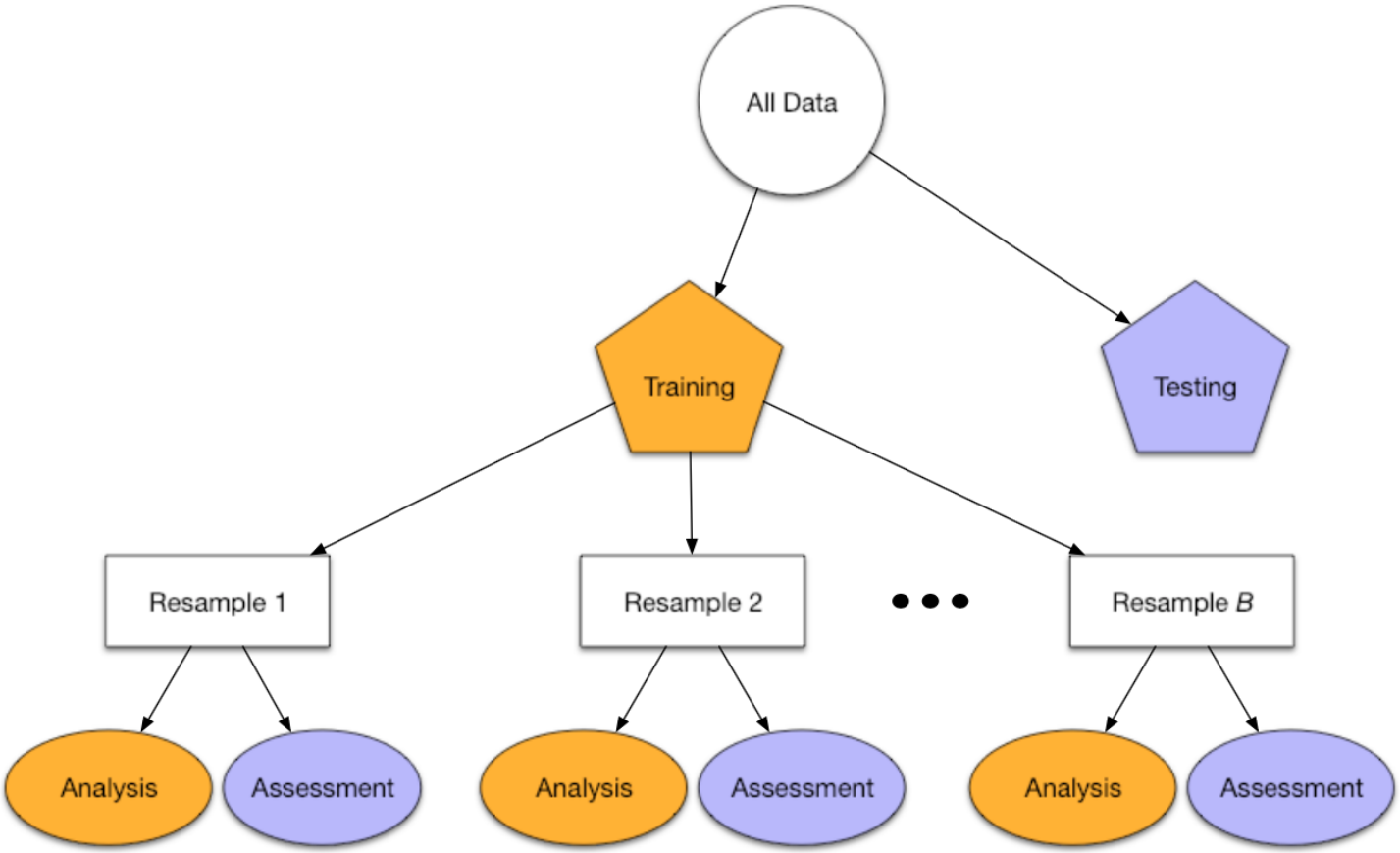
- **tidymodels** is to MODELING what the **tidyverse** is to DATA WRANGLING;
- **tidymodels** has a modular approach: specific, smaller packages are designed to work hand in hand.

tidymodels' main packages

Goal: predict diamond prices



How are we going to do it?



What tools do we have?

Pre-Process → Train → Validate

Separating Testing and Training Data

- *rsample* contains a set of functions to create different types of resamples and corresponding classes for their analysis:
  - Traditional resampling techniques for estimating the sampling distribution of a statistic and;
  - Estimating model performance using a holdout set.

Separating Testing and Training Data

```
dia_split <- initial_split(diamonds, pro = .1, strata = price)

dia_train <- training(dia_split)
dia_test <- testing(dia_split)

dia_vfold <- vfold_cv(dia_train, v = 3, repeats = 1, strata = price)
print(dia_vfold)
```

```
## #  3-fold cross-validation using stratification
## # A tibble: 3 x 2
##   splits               id
##   <list>               <chr>
## 1 <split [3594/1799]> Fold1
## 2 <split [3595/1798]> Fold2
## 3 <split [3597/1796]> Fold3
```

Data Pre-Processing and Feature Engineering

- *recipes* is a method for creating and pre-processing design matrices used for modeling or visualization;
- Idea: define a blueprint that can be used to sequentially define the encodings and pre-processing of the data;
- It is used to prepare a data set (for modeling) using different 'step_*()' functions;
- The 'recipe()' takes a formula and a data set, and then the different steps are added.

Data Pre-Processing and Feature Engineering

```
dia_rec <-
    recipe(price ~ ., data = dia_train) %>%
    step_log(all_outcomes()) %>%
    step_normalize(all_predictors(), -all_nominal()) %>%
    step_dummy(all_nominal()) %>%
    step_poly(carat, degree = 2)

prep(dia_rec)
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          9
##
## Training data contained 5393 data points and no missing data.
##
## Operations:
##
## Log transformation on price [trained]
## Centering and scaling for carat, depth, table, x, y, z [trained]
## Dummy variables from cut, color, clarity [trained]
## Orthogonal polynomials on carat [trained]
```

Data Pre-Processing and Feature Engineering

- Calling 'prep()' on a recipe applies all steps;
- Call 'juice()' to extract the transformed data set;
- Call 'bake()' on a new data set.

```
dia_juiced <- juice(prep(dia_rec))
names(dia_juiced)
```

```
##  [1] "depth"       "table"       "x"           "y"           "z"
##  [6] "price"       "cut_1"       "cut_2"       "cut_3"       "cut_4"
## [11] "color_1"     "color_2"     "color_3"     "color_4"     "color_5"
## [16] "color_6"     "clarity_1"   "clarity_2"   "clarity_3"   "clarity_4"
## [21] "clarity_5"   "clarity_6"   "clarity_7"   "carat_poly_1" "carat_poly_2"
```

Defining and Fitting Models

- The goal is to provide a tidy, unified interface to models that can be used to try a range of models without getting bogged down in the syntactical minutiae of the underlying packages;
- Has wrappers around many popular machine learning algorithms, and you can fit then using a unified interface.

Defining and Fitting Models

1. Function specific to each algorithm;
2. 'set_mode()' (regression or classification);
3. 'set_engine()' back-end/engine/implementation

```
lm_model <-
    linear_reg() %>%
    set_mode("regression") %>%
    set_engine("lm")

print(lm_model)
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

Defining and Fitting Models

- Random Forest: 'ranger' or 'randomForest'?
- How to handle their different interfaces?

```
rand_forest(mtry = 3, trees = 500, min_n = 5) %>%
    set_mode("regression") %>%
    set_engine("ranger", importance = "impurity_corrected")
```

```
## Random Forest Model Specification (regression)
##
## Main Arguments:
##   mtry = 3
##   trees = 500
##   min_n = 5
##
## Engine-Specific Arguments:
##   importance = impurity_corrected
##
## Computational engine: ranger
```

Defining and Fitting Models

- This example, with a formula. You can also set 'x' and 'y'.

```
lm_fit1 <- fit(lm_model, price ~ ., dia_juiced)
lm_fit1
```

```
## parsnip model object
##
## Fit time:  13ms
##
## Call:
## stats::lm(formula = price ~ ., data = data)
##
## Coefficients:
##  (Intercept)         depth         table             x             y
##     7.729257      0.009178      0.003533      0.042671      0.295996
##            z         cut_1         cut_2         cut_3         cut_4
##     0.038552      0.081400     -0.001890      0.004473     -0.004180
##      color_1       color_2       color_3       color_4       color_5
##    -0.442545     -0.086210     -0.010589      0.022553     -0.009546
##      color_6     clarity_1     clarity_2     clarity_3     clarity_4
##     0.001549      0.836278     -0.215930      0.101644     -0.050261
##    clarity_5     clarity_6     clarity_7  carat_poly_1  carat_poly_2
##     0.005939     -0.007926      0.028804     49.458215    -16.835516
```

Summarizing Fitted Models

- Takes the messy output of built-in function in R, such as 'lm', 'nls', and turns them into tidy tibbles;
- From **tidyverse**.

Summarizing Fitted Models

- 'glance()' reports information about the entire model;
- 'tidy()' summarizes information about model components.

```
glance(lm_fit1$fit)
```

```
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic p.value    df logLik    AIC    BIC
##       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <dbl>  <dbl>  <dbl>  <dbl>
## 1     0.978         0.978 0.149    10064.       0    24  2629. -5206. -5035.
## # ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

```
tidy(lm_fit1) %>%
    arrange(desc(abs(statistic))) %>%
    print()
```

```
## # A tibble: 25 x 5
##    term         estimate std.error statistic   p.value
##    <chr>           <dbl>     <dbl>     <dbl>     <dbl>
##  1 (Intercept)    7.73     0.00410   1885.    0
##  2 clarity_1      0.836    0.0125      66.7   0
##  3 color_1       -0.443    0.00706    -62.7   0
##  4 carat_poly_2 -16.8      0.284      -59.3   0
##  5 carat_poly_1  49.5      1.26        39.3   4.70e-297
##  6 clarity_2     -0.216    0.0117     -18.5   9.45e- 74
##  7 color_2       -0.0862   0.00646    -13.3   5.65e- 40
##  8 y              0.296    0.0269      11.0   8.93e- 28
##  9 clarity_3      0.102    0.0100      10.1   5.79e- 24
## 10 cut_1          0.0814   0.00948      8.58  1.19e- 17
## # ... with 15 more rows
```

Summarizing Fitted Models

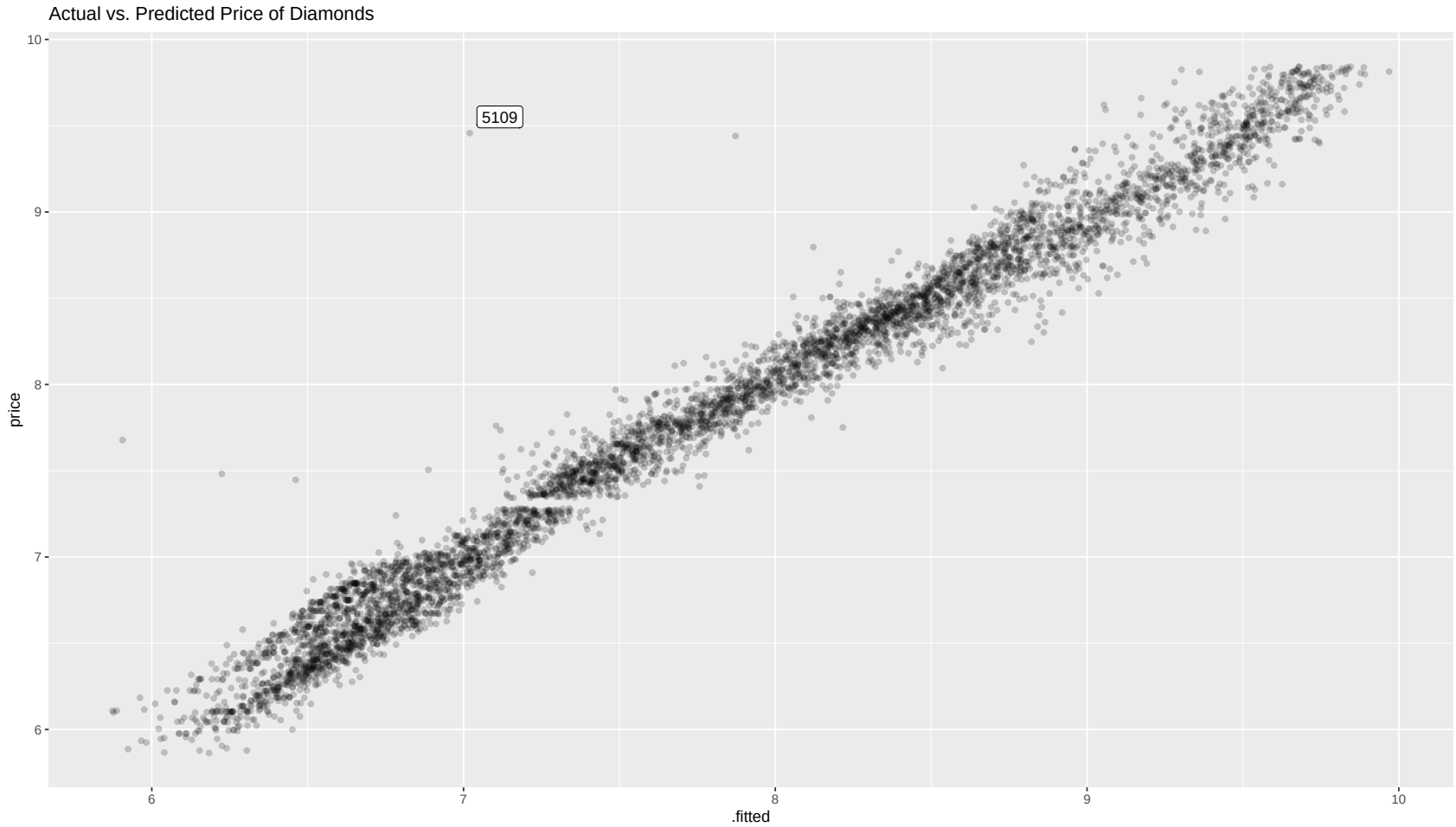
- 'augment()' is used to get model predictions, residuals, etc.

```
lm_predicted <- augment(lm_fit1$fit, data = dia_juiced) %>%
    rowid_to_column()
print(select(lm_predicted, rowid, price, .fitted:.std.resid))
```

```
## # A tibble: 5,393 x 8
##    rowid price .fitted .resid    .hat .sigma   .cooksd .std.resid
##    <int> <dbl>   <dbl>  <dbl>   <dbl>  <dbl>     <dbl>      <dbl>
##  1     1  5.86    6.18 -0.320 0.00378  0.149 0.000704      -2.15
##  2     2  5.87    6.04 -0.174 0.00444  0.149 0.000244      -1.17
##  3     3  6.00    6.26 -0.264 0.00439  0.149 0.000557      -1.78
##  4     4  6.00    6.45 -0.452 0.00750  0.149 0.00280       -3.05
##  5     5  6.31    6.48 -0.162 0.00288  0.149 0.000138      -1.09
##  6     6  6.32    6.56 -0.239 0.00493  0.149 0.000511      -1.61
##  7     7  6.32    6.13  0.191 0.00426  0.149 0.000281       1.28
##  8     8  6.32    6.44 -0.123 0.00233  0.149 0.0000642     -0.828
##  9     9  6.32    6.45 -0.124 0.00253  0.149 0.0000711     -0.837
## 10    10  6.32    6.50 -0.176 0.00398  0.149 0.000224      -1.18
## # ... with 5,383 more rows
```

Visualizing Results

```
ggplot(lm_predicted, aes(.fitted, price)) +
    geom_point(alpha = .2) +
    ggrepel::geom_label_repel(aes(label = rowid),
        data = filter(lm_predicted, abs(.resid) > 2)) +
    labs(title = "Actual vs. Predicted Price of Diamonds")
```



Evaluating Model Performance

Evaluating Model Performance

- Use 'rsample', 'parsinp' and 'yardstick' for cross-validation (3).

```
print(dia_vfold)
```

```
## #  3-fold cross-validation using stratification
## # A tibble: 3 x 2
##   splits               id
##   <list>               <chr>
## 1 <split [3594/1799]> Fold1
## 2 <split [3595/1798]> Fold2
## 3 <split [3597/1796]> Fold3
```

- Extract analysis/training and assessment/testing data.

```
lm_fit2 <- mutate(dia_vfold,
                  df_ana = map(splits, analysis),
                  df_ass = map(splits, assessment))
print(lm_fit2)
```

```
## #  3-fold cross-validation using stratification
## # A tibble: 3 x 4
##   splits               id    df_ana                df_ass
##   <list>               <chr> <list>                <list>
## 1 <split [3594/1799]> Fold1 <tibble [3,594 x 10]> <tibble [1,799 x 10]>
## 2 <split [3595/1798]> Fold2 <tibble [3,595 x 10]> <tibble [1,798 x 10]>
## 3 <split [3597/1796]> Fold3 <tibble [3,597 x 10]> <tibble [1,796 x 10]>
```

Evaluating Model Performance

- Prepare data / fit model / predict.

```
lm_fit2 <-
    lm_fit2 %>%
    # prep, juice, bake
    mutate(
        recipe = map(df_ana, ~prep(dia_rec, training = .x)),
        df_ana = map(recipe, juice),
        df_ass = map2(recipe, df_ass, ~bake(.x, new_data = .y))
    ) %>%
    #fit
    mutate(
        model_fit = map(df_ana, ~fit(lm_model, price ~ ., data = .x))
    ) %>%
    # predict
    mutate(
        model_pred = map2(model_fit, df_ass, ~predict(.x, new_data = .y))
    )

print(select(lm_fit2, id, recipe:model_pred))
```

```
## # A tibble: 3 x 4
##   id    recipe   model_fit model_pred
##   <chr> <list>   <list>    <list>
## 1 Fold1 <recipe> <fit[+]>  <tibble [1,799 x 1]>
## 2 Fold2 <recipe> <fit[+]>  <tibble [1,798 x 1]>
## 3 Fold3 <recipe> <fit[+]>  <tibble [1,796 x 1]>
```

Evaluating Model Performance

- Select original and predicted values.

```
lm_preds <-
    lm_fit2 %>%
    mutate(res = map2(df_ass, model_pred, ~data.frame(price = .x$price,
                                                      .pred = .y$.pred))
    ) %>%
    select(id, res) %>%
    tidyr::unnest(res) %>%
    group_by(id)

print(lm_preds)
```

```
## # A tibble: 5,393 x 3
## # Groups:   id [3]
##    id    price .pred
##    <chr> <dbl> <dbl>
##  1 Fold1  6.00  6.28
##  2 Fold1  6.31  6.48
##  3 Fold1  6.32  6.57
##  4 Fold1  6.32  6.45
##  5 Fold1  6.32  6.56
##  6 Fold1  6.32  6.45
##  7 Fold1  6.32  6.23
##  8 Fold1  6.33  6.48
##  9 Fold1  6.33  6.41
## 10 Fold1  6.33  6.46
## # ... with 5,383 more rows
```

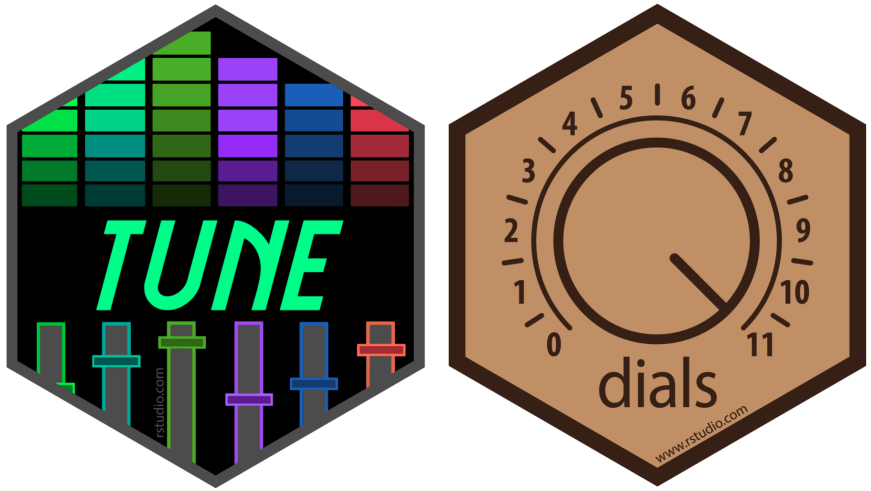
Evaluating Model Performance

- 'metrics()' has default measures for numeric and categorical outcomes (numeric - 'rmse', 'rsq', 'mae');
- You can choose other if you'd like with 'metric_set()'.

```
print(metrics(lm_preds, truth = price, estimate = .pred))
```

```
## # A tibble: 9 x 4
##   id    .metric .estimator .estimate
##   <chr> <chr>   <chr>          <dbl>
## 1 Fold1 rmse    standard       0.141
## 2 Fold2 rmse    standard       0.237
## 3 Fold3 rmse    standard       0.147
## 4 Fold1 rsq     standard       0.981
## 5 Fold2 rsq     standard       0.945
## 6 Fold3 rsq     standard       0.978
## 7 Fold1 mae     standard       0.114
## 8 Fold2 mae     standard       0.110
## 9 Fold3 mae     standard       0.114
```

Tuning Model Parameters



- 'tune' wants to facilitate hyper-parameter tuning for the **tidymodels** packages;
- 'dials' contains tools to create and manage values of tuning parameters;
- Let's tune the 'mtry' and 'degree' parameters.

Tuning Model Parameters

- Preparing a 'parsnip' Model for tuning.

```
rf_model <-
    rand_forest(mtry = tune()) %>%
    set_mode("regression") %>%
    set_engine("ranger")

print(parameters(rf_model))
```

```
## Collection of 1 parameters for tuning
##
##  identifier type    object
##        mtry mtry nparam[?]
##
## Model parameters needing finalization:
##    # Randomly Selected Predictors ('mtry')
##
## See 'dials::finalize' or 'dials::update.parameters' for more information.
```

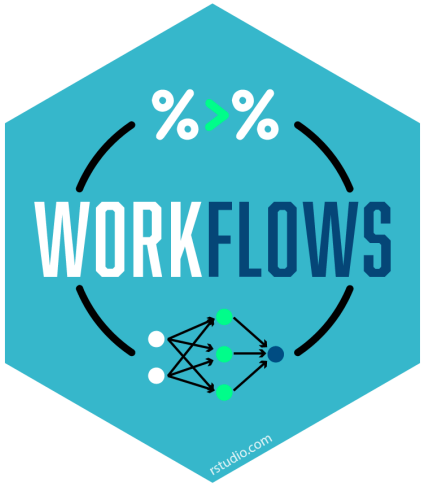
Tuning Model Parameters

- Preparing Data for Tuning: 'recipes';
- Tune the degree of the polynomial for the variable 'carat'.

```
dia_rec2 <-
    recipe(price ~ ., data = dia_train) %>%
    step_log(all_outcomes()) %>%
    step_normalize(all_predictors(), -all_nominal()) %>%
    step_dummy(all_nominal()) %>%
    step_poly(carat, degree = tune())

dia_rec2 %>%
    parameters() %>%
    pull("object") %>%
    print()
```

```
## [[1]]
## Polynomial Degree (quantitative)
## Range: [1, 3]
```

Combine Everything



- Object that can bundle together pre-processing, modeling and post-processing requests;
- The recipe prepping and model fitting can be executed using a single call to 'fit()'.

Combine Everything

```
rf_wflow <-
    workflow() %>%
    add_model(rf_model) %>%
    add_recipe(dia_rec2)

print(rf_wflow)
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: rand_forest()
##
## -- Preprocessor ----------------------------------------------------------------
## 4 Recipe Steps
##
## * step_log()
## * step_normalize()
## * step_dummy()
## * step_poly()
##
## -- Model -----------------------------------------------------------------------
## Random Forest Model Specification (regression)
##
## Main Arguments:
##   mtry = tune()
##
## Computational engine: ranger
```

Tuning Parameters

- Update the parameters in the workflow;
- Cross-validation for tuning: select the best combination of hyper-parameters.

```
rf_param <-
    rf_wflow %>%
    parameters() %>%
    update(mtry = mtry(range = c(3L, 5L)),
           degree = degree_int(range = c(2L, 4L)))

print(rf_param$object)
```

```
## [[1]]
## # Randomly Selected Predictors (quantitative)
## Range: [3, 5]
##
## [[2]]
## Polynomial Degree (quantitative)
## Range: [2, 4]
```

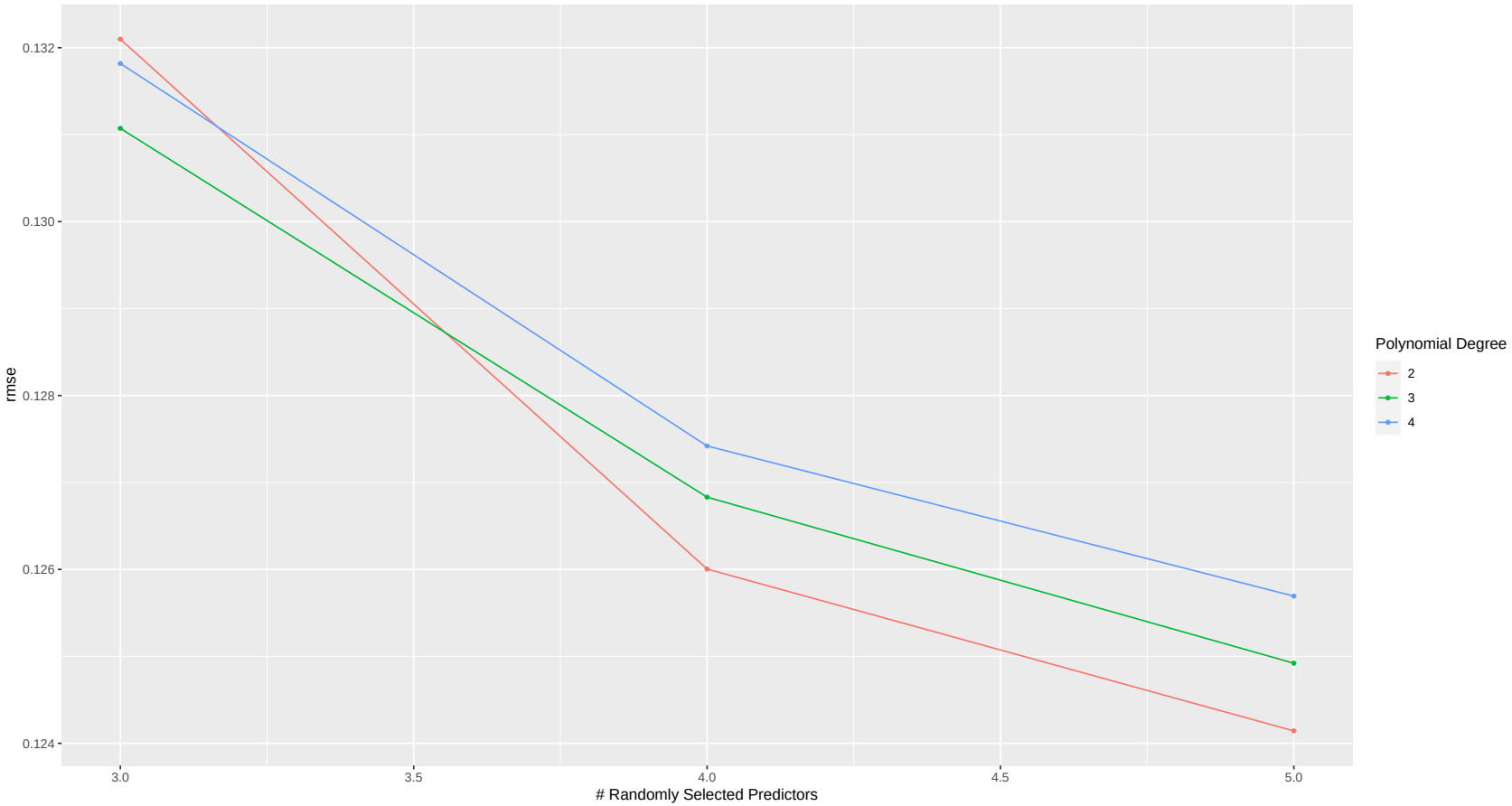
Tuning Parameters

```
rf_grid <- grid_regular(rf_param)

print(rf_grid)
```

```
## # A tibble: 9 x 2
##    mtry degree
##   <int>  <int>
## 1     3      2
## 2     4      2
## 3     5      2
## 4     3      3
## 5     4      3
## 6     5      3
## 7     3      4
## 8     4      4
## 9     5      4
```

Tuning Parameters

```
rf_search <- tune_grid(rf_wflow,
                       grid = rf_grid,
                       resamples = dia_vfold,
                       param_info = rf_param)

autoplot(rf_search, metric = "rmse") +
    labs("Results of Grid Search for Two Tuning Parameters of a Random Forest")
```



Model Selection

```
print(show_best(rf_search, "rmse", 9))
```

```
## # A tibble: 9 x 8
##    mtry degree .metric .estimator  mean     n std_err .config
##   <int>  <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     5      2 rmse    standard   0.124     3 0.00328 Preprocessor1_Model3
## 2     5      3 rmse    standard   0.125     3 0.00344 Preprocessor2_Model3
## 3     5      4 rmse    standard   0.126     3 0.00338 Preprocessor3_Model3
## 4     4      2 rmse    standard   0.126     3 0.00322 Preprocessor1_Model2
## 5     4      3 rmse    standard   0.127     3 0.00347 Preprocessor2_Model2
## 6     4      4 rmse    standard   0.127     3 0.00331 Preprocessor3_Model2
## 7     3      3 rmse    standard   0.131     3 0.00278 Preprocessor2_Model1
## 8     3      4 rmse    standard   0.132     3 0.00296 Preprocessor3_Model1
## 9     3      2 rmse    standard   0.132     3 0.00319 Preprocessor1_Model1
```

```
print(select_best(rf_search, metric = "rmse"))
```

```
## # A tibble: 1 x 3
##    mtry degree .config
##   <int>  <int> <chr>
## 1     5      2 Preprocessor1_Model3
```

```
print(select_by_one_std_err(rf_search, mtry, degree, metric = "rmse"))
```

```
## # A tibble: 1 x 10
##    mtry degree .metric .estimator  mean     n std_err .config       .best .bound
##   <int>  <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>         <dbl>  <dbl>
## 1     4      2 rmse    standard   0.126     3 0.00322 Preprocessor~ 0.124  0.127
```

Best Model and Final Predictions

```
rf_param_final <- select_by_one_std_err(rf_search, mtry, degree, metric = "rmse")

rf_wflow_final <- finalize_workflow(rf_wflow, rf_param_final)

rf_wflow_final_fit <- fit(rf_wflow_final, data = dia_train)
```

- Want to use 'predict()' on data never seem before ('dia_test');
- However, it does not work, because the outcome is modified in the recipe via 'step_log()'.

Best Model and Final Predictions

- Workaround:
  1. Prepped recipe is extracted from the workflow;
  2. This is used to 'bake()' the testing data;
  3. Use this baked data set together with extracted model for final predictions.

```
dia_rec3 <- extract_recipe(rf_wflow_final_fit)
rf_final_fit <- extract_fit_parsnip(rf_wflow_final_fit)

dia_test$.pred <- predict(rf_final_fit,
                          new_data = bake(dia_rec3, dia_test))$.pred
dia_test$logprice <- log(dia_test$price)

metrics(dia_test, truth = logprice, estimate = .pred)
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       0.113
## 2 rsq     standard       0.988
## 3 mae     standard       0.0846
```