Building Better Predictive Models with R: Tidymodels approach

Dr. Kam Tin Seong
Assoc. Professor of Information Systems

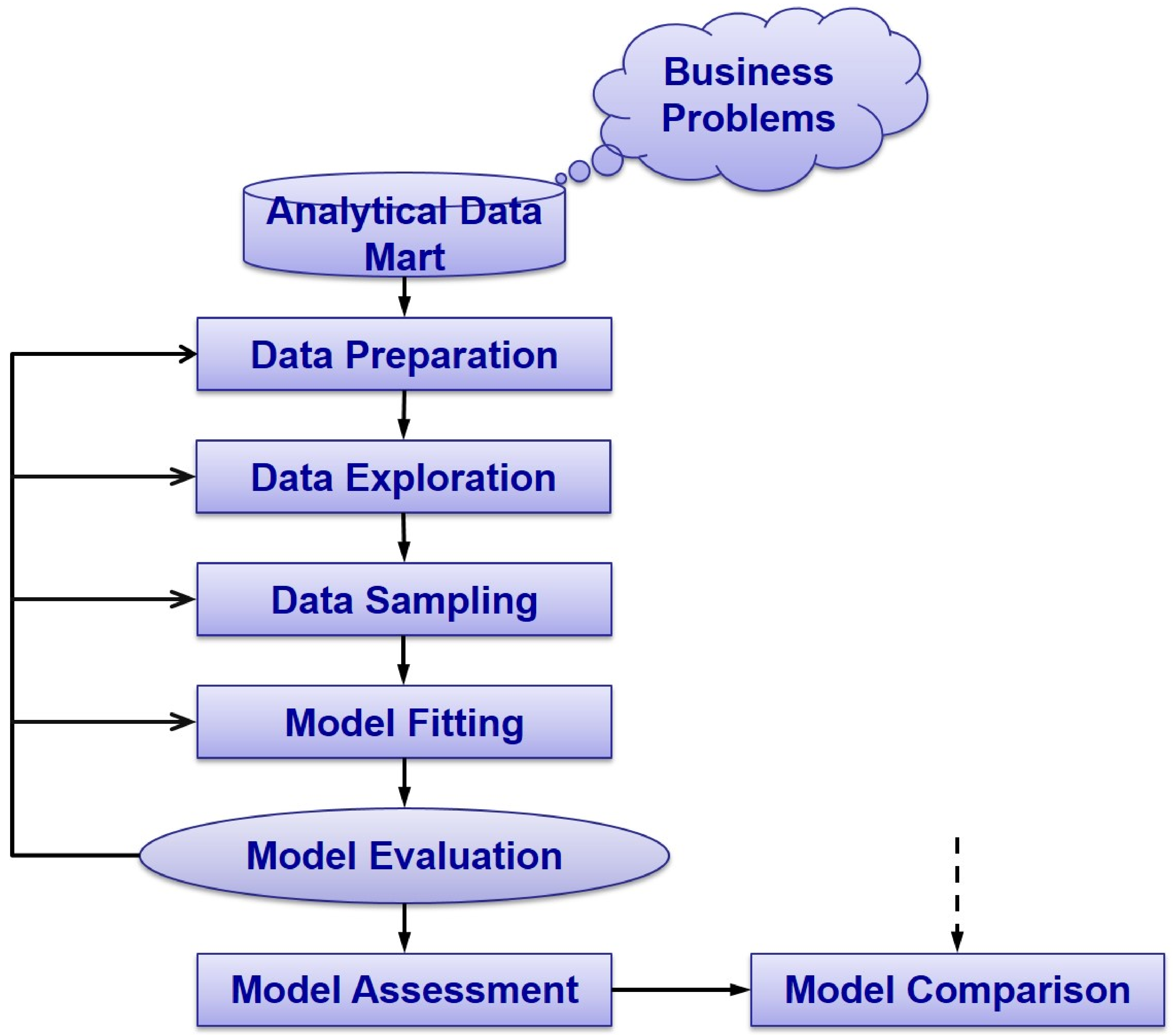
School of Computing and Information Systems,
Singapore Management University

2022-08-20 (updated: 2022-12-09)

Content

- An Overview of Predictive Modelling
- Predictive Modeling with tidymodels
    - rsample
    - recipes
    - parsnip
    - broom
    - yardstick

Predictive Modelling Process



Why R for Modeling?

- R and R packages are built by people who **do** data analysis.
- The machine learning environment in R is extremely rich.
- R has cutting edge models.
  - Machine learning developers in some 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, Weka python, or stan, you can access these applications without leaving R.

Conventional R packages for building predictive models

- caret
- mlr3
- rpart
- many others

Downsides to modeling in R

- R is a data analysis language and is not C or Java. If a high performance deployment is required, R can be treated like a prototyping language.
- R is mostly memory-bound. There are plenty of exceptions to this though.
- The main issue is one of **consistency of interface**. For example:
  - There are two methods for specifying what terms are in a model[1]. Not all models have both.
  - 99% of model functions automatically generate dummy variables.
  - Sparse matrices can be used (unless they can't).

[1] There are now three but the last one is brand new and will be discussed later.

Syntax for computing predicted class probabilities

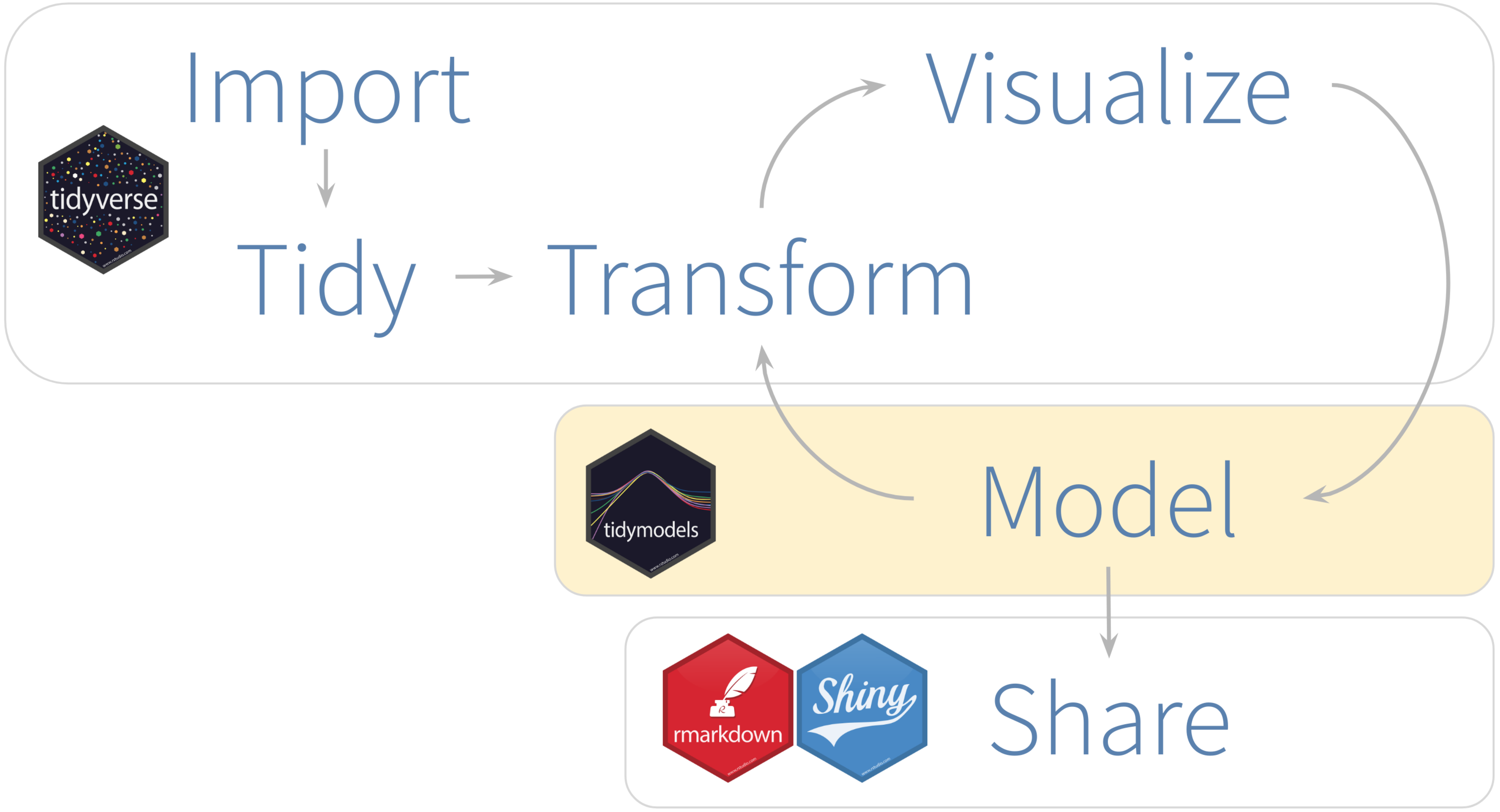
Function	Package	Code
`lda`	`MASS`	`predict(obj)`
`glm`	`stats`	`predict(obj, type = "response")`
`gbm`	`gbm`	`predict(obj, type = "response", n.trees)`
`mda`	`mda`	`predict(obj, type = "posterior")`
`rpart`	`rpart`	`predict(obj, type = "prob")`
`Weka`	`RWeka`	`predict(obj, type = "probability")`
`logitboost`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

What is tidymodels

- An integrated, modular, extensible set of packages that implement a framework that facilitates creating predicative stochastic models.
- They adhere to tidyverse syntax and design principles that promote consistency and well-designed human interfaces over speed of code execution.
- They automatically build in parallel execution for tasks such as resampling, cross validation and parameter tuning.
- They implement conceptual structures that make complex iterative workflows possible and reproducible.

- Plus `tidypredict`, `tidyposterior`, `tidytext`, and more in development.

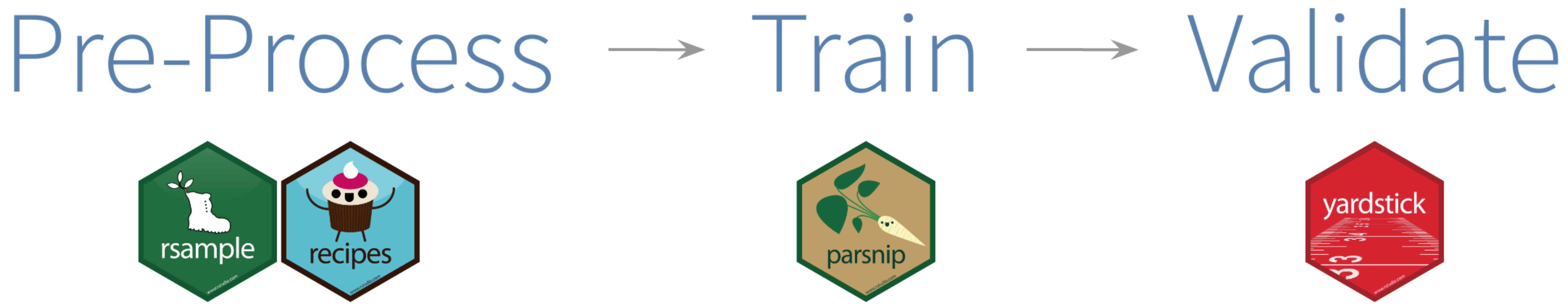
tidymodels in the context of Data Science and Analytics framewok



tidymodels

In a way, the Model step itself has sub-steps. For these sub-steps, tidymodels provides one or several packages. This article will showcase functions from four tidymodels packages:

- rsample - Different types of re-samples
- recipes - Transformations for model data pre-processing
- parsnip - A common interface for model creation
- yardstick - Measure model performance



Modelling Diamond Prices Case Study

- To build a model for predicting the prices of diamonds by using the `Diamond` data set includes in **ggplot2** package.
- The data set consists of 53940 records and 10 variables. the variables are:
  - price: price in US dollars ($326–$18,823)
  - carat: weight of the diamond (0.2–5.01)
  - cut: quality of the cut (Fair, Good, Very Good, Premium, Ideal)
  - color: diamond colour, from D (best) to J (worst)
  - clarity: a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, IF (best))
  - x: length in mm (0–10.74)
  - y: width in mm (0–58.9)
  - z: depth in mm (0–31.8)

- depth: total depth percentage = z / mean(x, y) = 2 * z / (x + y) (43–79)
- table: width of top of diamond relative to widest point (43–95)

Be warned: This data set is relative clean. Real world data are very untidy and dirty and significant amount of efforts are required to prepare the final modelling data set.

Getting Started

Before we getting started, it is important for us to install the necessary R packages into R and launch these R packages into R environment.

The R packages needed for this exercise are as follows:

- tidyverse for importing, wrangling, visualising data, and
- tidymodels for sampling, calibrating, modeling and assessing models.

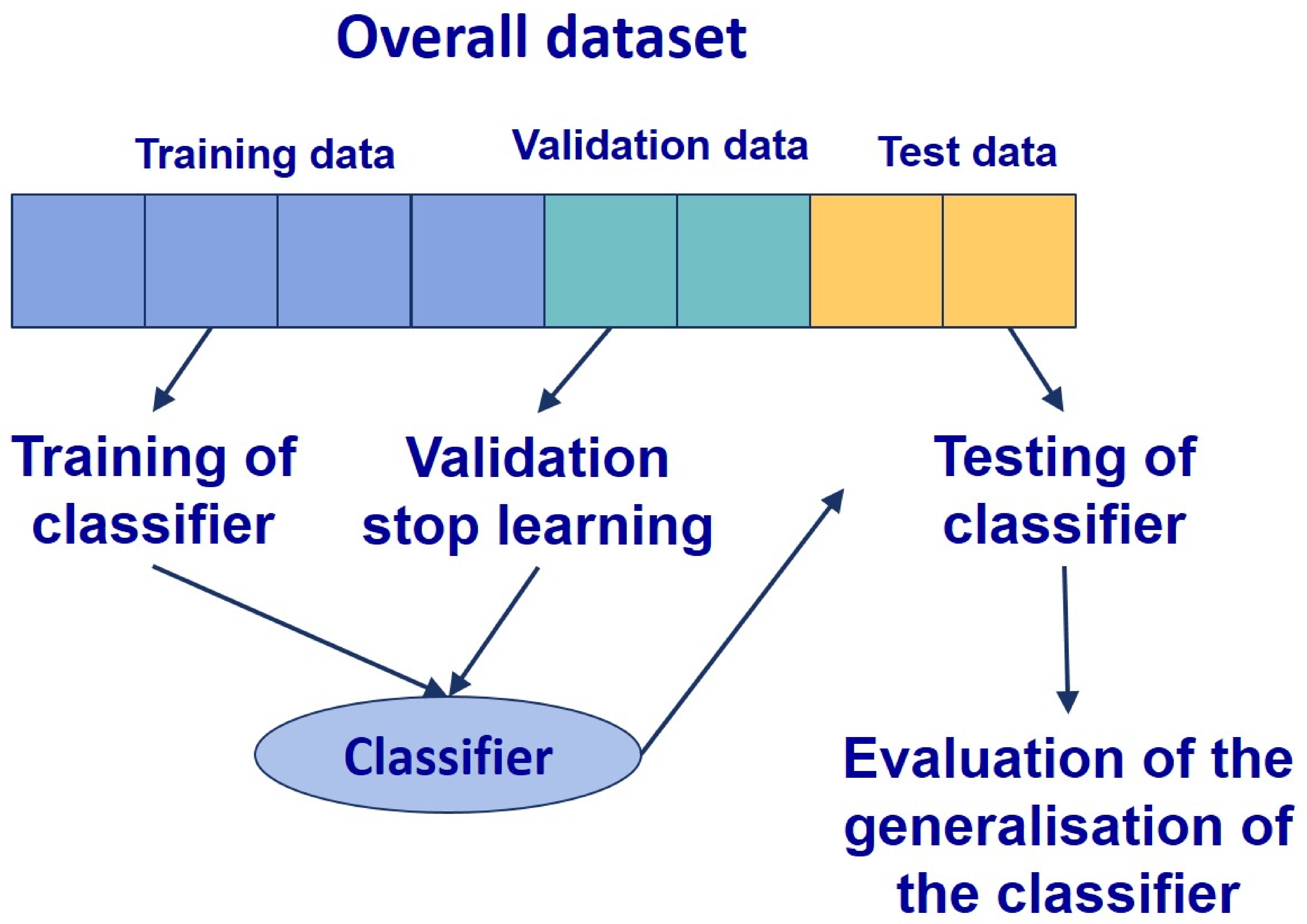
The code chunks below installs and launches these R packages into R environment.

```r
pacman::p_load(tidymodels, tidyverse)
```

Next, `data()` is used to load diamonds data set from **ggplot2** package.

```r
data(diamonds)
```

Data Sampling in Predictive Analytics



rsample

- rsample contains a set of functions to create different types of resamples and corresponding classes for their analysis.
- The goal is to have a modular set of methods that can be used across different R packages for:
  - traditional resampling techniques for estimating the sampling distribution of a statistic and
  - estimating model performance using a holdout set.
- The scope of rsample is to provide the basic building blocks for creating and analyzing resamples of a data set but does not include code for modeling or calculating statistics.

Data sampling with *initial_split()*

- *initial_split()* creates a single binary split of the data into a training set and testing set.
- *training()* and *testing()* are used to extract the resulting data.

- data: A data frame.
- prop: The proportion of data to be retained for modeling/analysis.
- strata: A variable that is used to conduct stratified sampling to create the resamples. This could be a single character value or a variable name that corresponds to a variable that exists in the data frame.
- breaks: A single number giving the number of bins desired to stratify a numeric stratification variable.
- lag: A value to include an lag between the assessment and analysis set. This is useful if lagged predictors will be used during training and testing.
- x: An rsplit object produced by `initial_split()`

Data Sampling: rsample method

The code chunk on the right is used to perform the following tasks:

- *set.seed()* is used to ensure that the data sampling process is reproducible.
- select columns 1-7, and
- split the data into training and test data sets whereby 60% of the data for training and 40% of the data for testing.

```r
set.seed(1243)
diamonds_split <- diamonds %>%
  select(c(1:7)) %>%
  initial_split(prop = .6,
                strata = price)

training_data <- training(diamonds_split)
testing_data  <- testing(diamonds_split)
```

Creating cross-validation data

- V-fold cross-validation randomly splits the data into V groups of roughly equal size (called “folds”).
- A resample of the analysis data consisted of V-1 of the folds while the assessment set contains the final fold.
- In basic V-fold cross-validation (i.e. no repeats), the number of resamples is equal to V.

- data: A data frame.
- v: The number of partitions of the data set.
- repeats: The number of times to repeat the V-fold partitioning.
- strata: A variable that is used to conduct stratified sampling to create the folds. This could be a single character value or a variable name that corresponds to a variable that exists in the data frame.
- breaks: A single number giving the number of bins desired to stratify a numeric stratification variable.

Creating cross-validation data sets: rsample method

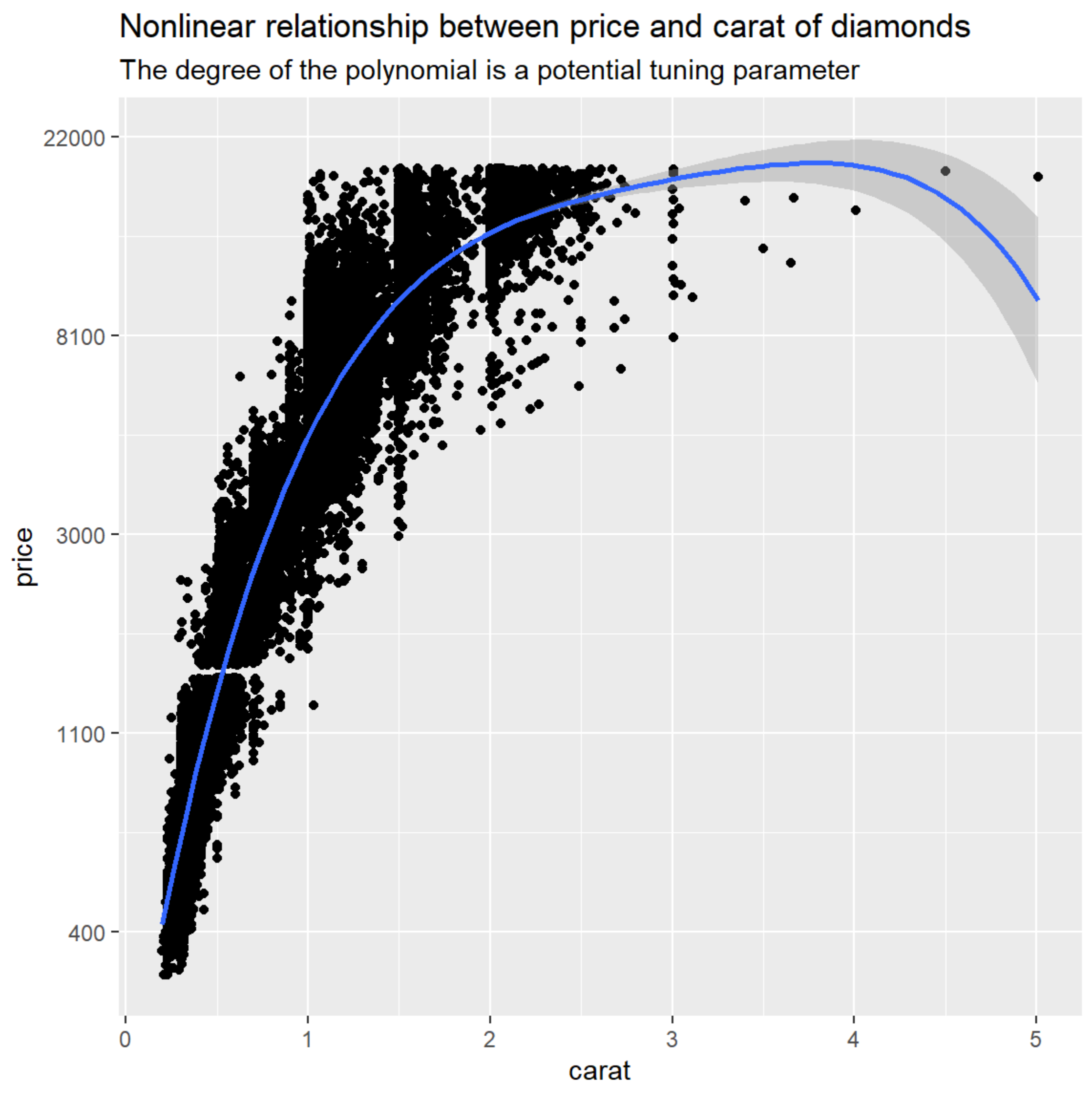
In the code chunk below, the training data set is prepared for 3-fold cross-validation (using three here to speed things up).

```r
vfold_data <- vfold_cv(training_data,
                       v = 3,
                       repeats = 1,
                       strata = price)
vfold_data %>%
  mutate(df_ana = map(splits, analysis),
         df_ass = map(splits, assessment))
```

```
#  3-fold cross-validation using stratification
# A tibble: 3 × 4
  splits                id    df_ana                 df_ass
  <list>                <chr> <list>                 <list>
1 <split [21576/10788]> Fold1 <tibble [21,576 × 7]> <tibble [10,788 × 7]>
2 <split [21576/10788]> Fold2 <tibble [21,576 × 7]> <tibble [10,788 × 7]>
3 <split [21576/10788]> Fold3 <tibble [21,576 × 7]> <tibble [10,788 × 7]>
```

Data Pre-Processing and Feature Engineering: recipes

- recipes package is an alternative method for creating and preprocessing design matrices that can be used for modeling or visualization.
- It provides a collection of useful functions for performing data pre-processing feature engineering tasks. For example, the plot on the right indicates that there may be a nonlinear relationship between price and carat.



recipe’s functions for data pre-processing

- `step_log` creates a specification of a recipe step that will log transform data.
- `step_normalize` creates a specification of a recipe step that will normalize numeric data to have a standard deviation of one and a mean of zero.
- `step_dummy` creates a a specification of a recipe step that will convert nominal data (e.g. character or factors) into one or more numeric binary model terms for the levels of the original data.
- `step_poly()` creates a specification of a recipe step that will create new columns that are basis expansions of variables using orthogonal polynomials.
- `prep()` estimates the required parameters from a training set that can be later applied to other data sets.
- `juice()` extracts transformed training set.

Working with recipe

Herein, I want to log transform price (step_log()), I want to center and scale all numeric predictors (step_normalize()), and the categorical predictors should be dummy coded (step_dummy()). Furthermore, a quadratic effect of carat is added using step_poly().

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

Working with recipe

Lastly, `prep()` calls on a recipe applies all the steps and `juice()` is used to extract the transformed data set on a new data set.

```r
prep(processed_data)
```

```
Recipe

Inputs:

      role #variables
   outcome          1
 predictor          6

Training data contained 32364 data points and no missing data.

Operations:

Log transformation on price [trained]
Centering and scaling for carat, depth, table [trained]
Dummy variables from cut, color, clarity [trained]
Orthogonal polynomials on carat [trained]
```

```r
juiced_data <- juice(prep(processed_data))
names(juiced_data)
```

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

parsnip package

parsnip is a new tidymodels package that generalizes model interfaces across packages.

- The idea is to have a single function interface for types of specific models (e.g. logistic regression) that lets the user choose the computational engine for training.
- **parsnip** also standardizes the return objects and sets up some new features for some upcoming packages.

The goals of parsnip are to:

- separate the definition of a model from its evaluation.
- decouple the model specification from the implementation (whether the implementation is in R, spark, or something else). For example, the user would call rand_forest instead of ranger::ranger or other specific packages.
- harmonize argument names (e.g. n.trees, ntrees, trees) so that users only need to remember a single name. This will help across model types too so that trees will be the same argument across random forest as well as boosting or bagging.

- Refer to this article to learn more about parsnip.

parsnip in the nutshell

A way to generate a specification of a model before fitting and allows the model to be created using different packages in R, Stan, keras, or via Spark.

- mode: A single character string for the type of model. The only possible value for this model is "regression".
- penalty: A non-negative number representing the total amount of regularization (glmnet, keras, and spark only). For keras models, this corresponds to purely L2 regularization (aka weight decay) while the other models can be a combination of L1 and L2 (depending on the value of mixture; see below).
- mixture: A number between zero and one (inclusive) that is the proportion of L1 regularization (i.e. lasso) in the model. When mixture = 1, it is a pure lasso model while mixture = 0 indicates that ridge regression is being used. (glmnet and spark only).
- object: A linear regression model specification.
- parameters: A 1-row tibble or named list with main parameters to update. If the individual arguments are used, these will supersede the values in parameters. Also, using engine arguments in this object will result in an error.
- fresh: A logical for whether the arguments should be modified in-place of or replaced wholesale.

Calibrating a multivariate linear regression model: parsnip method

In the code chunk below, `linear_reg()` of parsnip is used to calibrate a multivariate linear regression model.

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

Working with fit()

In the code chunk below, `fit()` of parsnip package is used to fit the `lm_model` prepared earlier..

```r
lm_fit <- fit(lm_model,
              price ~ .,
              juiced_data)
lm_fit
```

```
parsnip model object


Call:
stats::lm(formula = price ~ ., data = data)

Coefficients:
 (Intercept)          depth          table          cut_1
cut_2
   7.721e+00     -3.345e-03     -2.002e-03      8.743e-02
-7.044e-03
       cut_3          cut_4        color_1        color_2
color_3
   1.031e-02      8.402e-04     -4.575e-01     -8.609e-02
-1.453e-03
```

Calibrating a random forest model: parsnip method

In the code chunk below, **ranger** engine of **Ranger** package is used to calibrate a random forest model.

```r
rf_fit <- rand_forest(
  mode = "regression",
  engine = "ranger",
  mtry = .preds(),
  trees = 100) %>%
  fit(price ~ .,
      data = juiced_data)
rf_fit
```

```
parsnip model object

Ranger result

Call:
 ranger::ranger(x = maybe_data_frame(x), y = y, mtry =
min_cols(~.preds(),      x), num.trees = ~100, num.threads
= 1, verbose = FALSE, seed = sample.int(10^5,      1))

Type:                             Regression
Number of trees:                  100
Sample size:                      32364
Number of independent variables:  21
Mtry:                             21
Target node size:                 5
Variable importance mode:         none
```

Calibrating a random forest model: parsnip method

We can change to alternative package (i.e. randomForest)by using one of the supported engine as shown in the code chunk below.

```r
rf2_fit <- rand_forest(
  mode = "regression",
  engine = "randomForest", #<<
  mtry = .preds(),
  trees = 100) %>%
  fit(price ~ .,
      data = juiced_data)
rf2_fit
```

broom package

broom package takes the messy output of built-in functions in R, such as lm, nls, or t.test, and turns them into tidy tibbles.

It provides three S3 methods that do three distinct kinds of tidying.

- *tidy()*: constructs a tibble that summarizes the model's statistical findings. This includes coefficients and p-values for each term in a regression, per-cluster information in clustering applications, or per-test information for multtest functions.
- *augment()*: add columns to the original data that was modeled. This includes predictions, residuals, and cluster assignments.
- *glance()*: construct a concise one-row summary of the model. This typically contains values such as R^2, adjusted R^2, and residual standard error that are computed once for the entire model.
- For an introduction to broom package, please read this article.

Summarizing Fitted Models: glance()

In the code chunk below, `glance()` gives us information about the whole model. Here, R squared is pretty high and the RMSE equals 0.154.

```r
glance(lm_fit$fit)
```

```
# A tibble: 1 × 12
  r.squared adj.r.s…¹ sigma stati…² p.value    df logLik     AIC     BIC devia…³
      <dbl>     <dbl> <dbl>   <dbl>   <dbl> <dbl>  <dbl>   <dbl>   <dbl>   <dbl>
1     0.973     0.973 0.165  56472.       0    21 12307. -24567. -24375.    886.
# … with 2 more variables: df.residual <int>, nobs <int>, and abbreviated
#   variable names ¹adj.r.squared, ²statistic, ³deviance
```

Summarizing model parameters: tidy()

In the code chunk below, `tidy()` is used to reveal the model parameters, and we see that we have a significant quadratic effect of carat.

```r
tidy(lm_fit) %>%
  arrange(desc(abs(statistic)))
```

```
# A tibble: 22 × 5
   term          estimate std.error statistic   p.value
   <chr>            <dbl>     <dbl>     <dbl>     <dbl>
 1 (Intercept)     7.72     0.00185    4176.  0
 2 carat_poly_1 189.        0.191       990.  0
 3 carat_poly_2 -54.8       0.169      -324.  0
 4 clarity_1      0.828     0.00574     144.  0
 5 color_1       -0.458     0.00328    -140.  0
 6 clarity_2     -0.221     0.00538     -41.0 0
 7 color_2       -0.0861    0.00299     -28.8 3.92e-180
 8 clarity_3      0.118     0.00460      25.7 3.59e-144
 9 cut_1          0.0874    0.00419      20.8 6.89e- 96
10 clarity_4     -0.0466    0.00367     -12.7 7.16e- 37
# … with 12 more rows
```

Getting predictions: augment()

Finally, `augment()` can be used to get model predictions, residuals, etc.

```r
lm_predicted <- augment(lm_fit$fit,
                        data = juiced_data) %>%
  rowid_to_column()
select(lm_predicted,
       rowid,
       price,
       .fitted:.std.resid)
```

```
# A tibble: 32,364 × 8
   rowid price .fitted .resid      .hat .sigma   .cooksd .std.resid
   <int> <dbl>   <dbl>  <dbl>     <dbl>  <dbl>     <dbl>      <dbl>
 1     1  5.79    6.04 -0.254 0.000527  0.165 0.0000564     -1.53
 2     2  5.79    6.10 -0.311 0.000556  0.165 0.0000894     -1.88
 3     3  5.79    6.34 -0.545 0.00136   0.165 0.000674      -3.30
 4     4  5.81    6.19 -0.376 0.000642  0.165 0.000151      -2.27
 5     5  5.82    6.04 -0.220 0.00115   0.165 0.0000928     -1.33
 6     6  5.82    6.23 -0.413 0.000924  0.165 0.000262      -2.50
 7     7  5.82    6.05 -0.229 0.000610  0.165 0.0000532     -1.38
 8     8  5.82    6.14 -0.315 0.000761  0.165 0.000125      -1.90
 9     9  5.83    5.94 -0.113 0.000972  0.165 0.0000208     -0.686
10    10  5.83    6.09 -0.260 0.000579  0.165 0.0000651     -1.57
# … with 32,354 more rows
```

yardstick package

yardstick is a package specially designed for `model assessment` and `comparison` using tidy data principles.

- `metrics()` estimates one or more common performance estimates depending on the class of truth (see Value below) and returns them in a three column tibble.
- Arguments
  - data: A data.frame containing the truth and estimate columns and any columns specified by
  - truth: The column identifier for the true results (that is numeric or factor). This should be an unquoted column name although this argument is passed by expression and support quasiquotation (you can unquote column names).
  - estimate: The column identifier for the predicted results (that is also numeric or factor). As with truth this can be specified different ways but the primary method is to use an unquoted variable name.
  - na_rm: A logical value indicating whether NA values should be stripped before the computation proceeds.

Deriving processed version of the test set predictors

The code chunk below pre-processes the test set predictors.

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

The code chunk below derives the processed version of the test data by using *bake()* of **recipe** package.

```r
test_data <- bake(price_recipe,
                  new_data = testing_data,
                  all_predictors())
```

Note that all computations during the baking process are done in a non-sparse format. Also, note that this argument should be called after any selectors and the selectors should only resolve to numeric columns (otherwise an error is thrown).

Computing the prediction values

In the code chunks below, `predict()` of parsnip package is used to compute the predicted values for both the lm and rf models

lm Model

```r
lm_pred <- predict(
  lm_fit,
  new_data = test_data) %>%
  rename(lm = .pred)
```

Random Forest Model

```r
rf_pred <- predict(
  rf_fit,
  new_data = test_data) %>%
  rename(rf = .pred)
```

Model comparison

First, the code chunk below joins *testing_data* and newly created data tables of *rf_pred* and *lm-pred*.

```r
test_results <- testing_data %>%
  select(price) %>%
  mutate(log_price = log(price)) %>%
  bind_cols(rf_pred) %>%
  bind_cols(lm_pred)
```

Model comparison

Then, `metrics()` used to calculate the accuracy of the lm model and rf model by using the code chunk below.

lm Model

```r
test_results %>% metrics(
  truth = log_price,
  estimate = lm)
```

```
# A tibble: 3 × 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard       0.160
2 rsq     standard       0.975
3 mae     standard       0.124
```

RF Model

```r
test_results %>% metrics(
  truth = log_price,
  estimate = rf)
```

```
# A tibble: 3 × 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard      0.108
2 rsq     standard      0.989
3 mae     standard      0.0803
```

Note that `metrics()` has default measures for numeric and categorical outcomes, and here RMSE, R squared, and the mean absolute difference (MAE) are returned. You could also use one metric directly like `rmse()` or define a custom set of metrics via `metric_set()`.

```r
model_metrics <- metric_set(rmse, rsq, mase)
```

Model comparison

In the code chunk below, a group comparison report is generated.

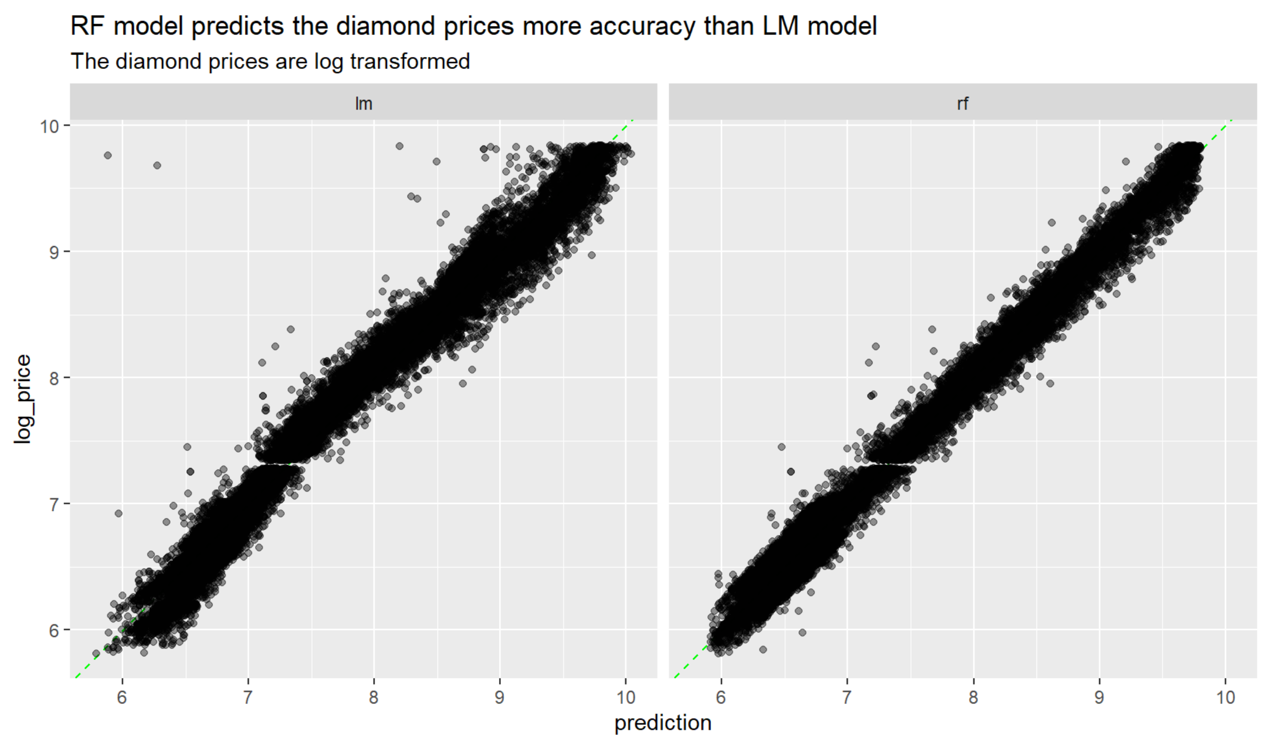
```r
test_results %>%
  select(c(2:4)) %>%
    pivot_longer(
      !log_price,
      names_to = "model",
      values_to = "prediction") %>%
  group_by(model) %>%
  metrics(truth = log_price,
          estimate = prediction)
```

```
# A tibble: 6 × 4
  model .metric .estimator .estimate
  <chr> <chr>   <chr>          <dbl>
1 lm    rmse    standard      0.160
2 rf    rmse    standard      0.108
3 lm    rsq     standard      0.975
4 rf    rsq     standard      0.989
5 lm    mae     standard      0.124
6 rf    mae     standard      0.0803
```

Visualising model performance

In the code chunk below, **ggplot2** is used to plot the predicted values of both lm model and rf model again the *log_price* for visualising the prediction results.

```r
test_results %>%
  select(c(2:4)) %>%
    pivot_longer(
      !log_price,
      names_to = "model",
      values_to = "prediction") %>%
  ggplot(aes(x = prediction,
             y = log_price)) +
  geom_abline(col = "green",
              lty = 2) +
  geom_point(alpha = .4) +
  facet_wrap(~model) +
  coord_fixed() +
  labs(title = "RF model predicts the diamond
       prices more accuracy than LM model",
         subtitle = "The diamond prices are
       log transformed")
```



Reference

- Tidy Modeling with R by Max Kuhn and Julia Silge.
- tidymodels homepage.
- resample
- recipe
- parsnip
- broom
- yardstick