Machine learning workflows

MODELING WITH TIDYMODELS IN R

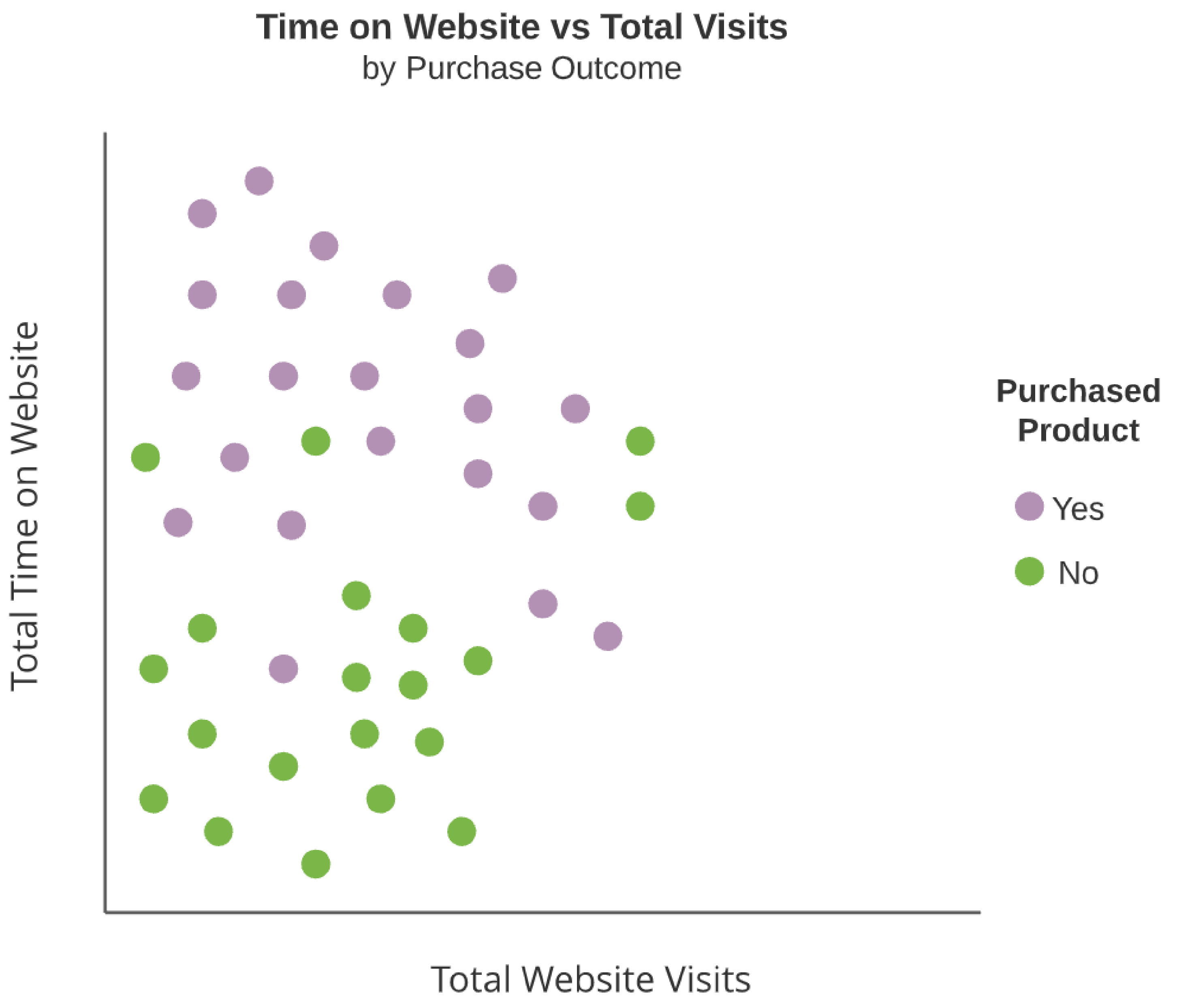
David Svancer
Data Scientist

Classification with decision trees

Decision trees segment the predictor space into **rectangular** regions

Recursive binary splitting

- Algorithm that segments predictor space into non-overlapping rectangular regions

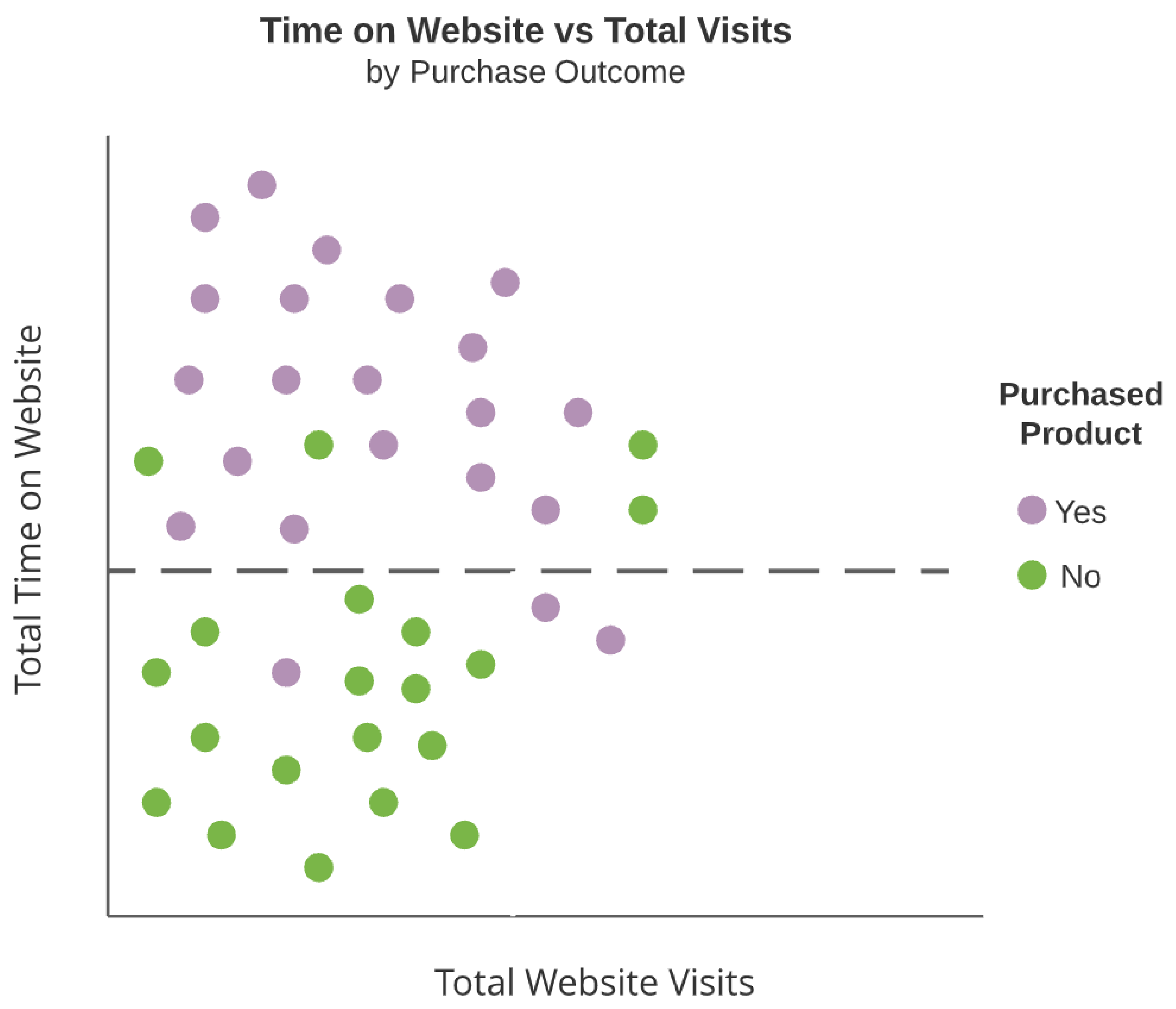


Classification with decision trees

Decision trees segment the predictor space into **rectangular** regions

Recursive binary splitting

- Algorithm that segments predictor space into non-overlapping rectangular regions
- Decision splits are added iteratively
  - Either horizontal or vertical cut points

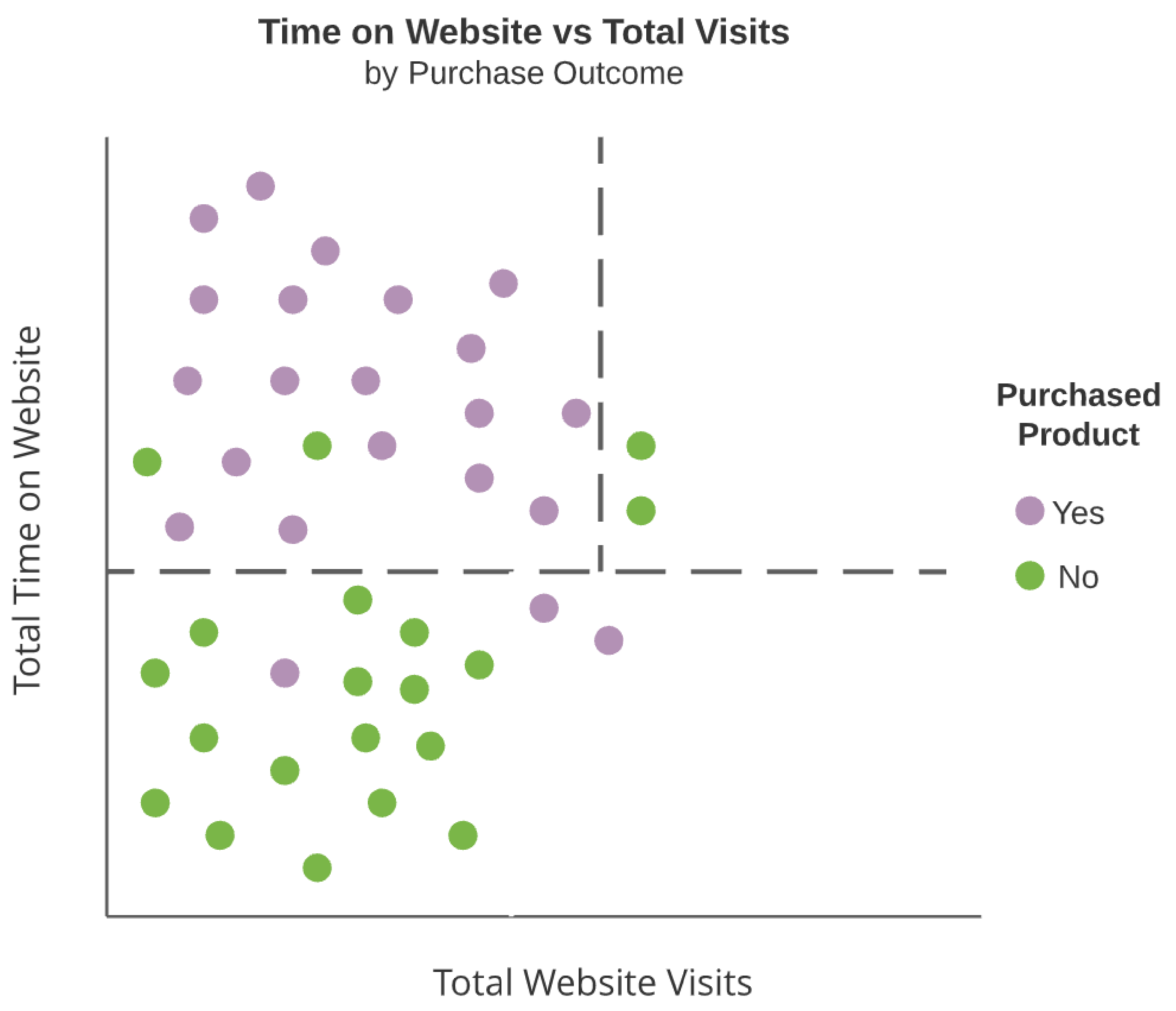


Classification with decision trees

Decision trees segment the predictor space into **rectangular** regions

Recursive binary splitting

- Algorithm that segments predictor space into non-overlapping rectangular regions
- Decision splits are added iteratively
  - Either horizontal or vertical cut points

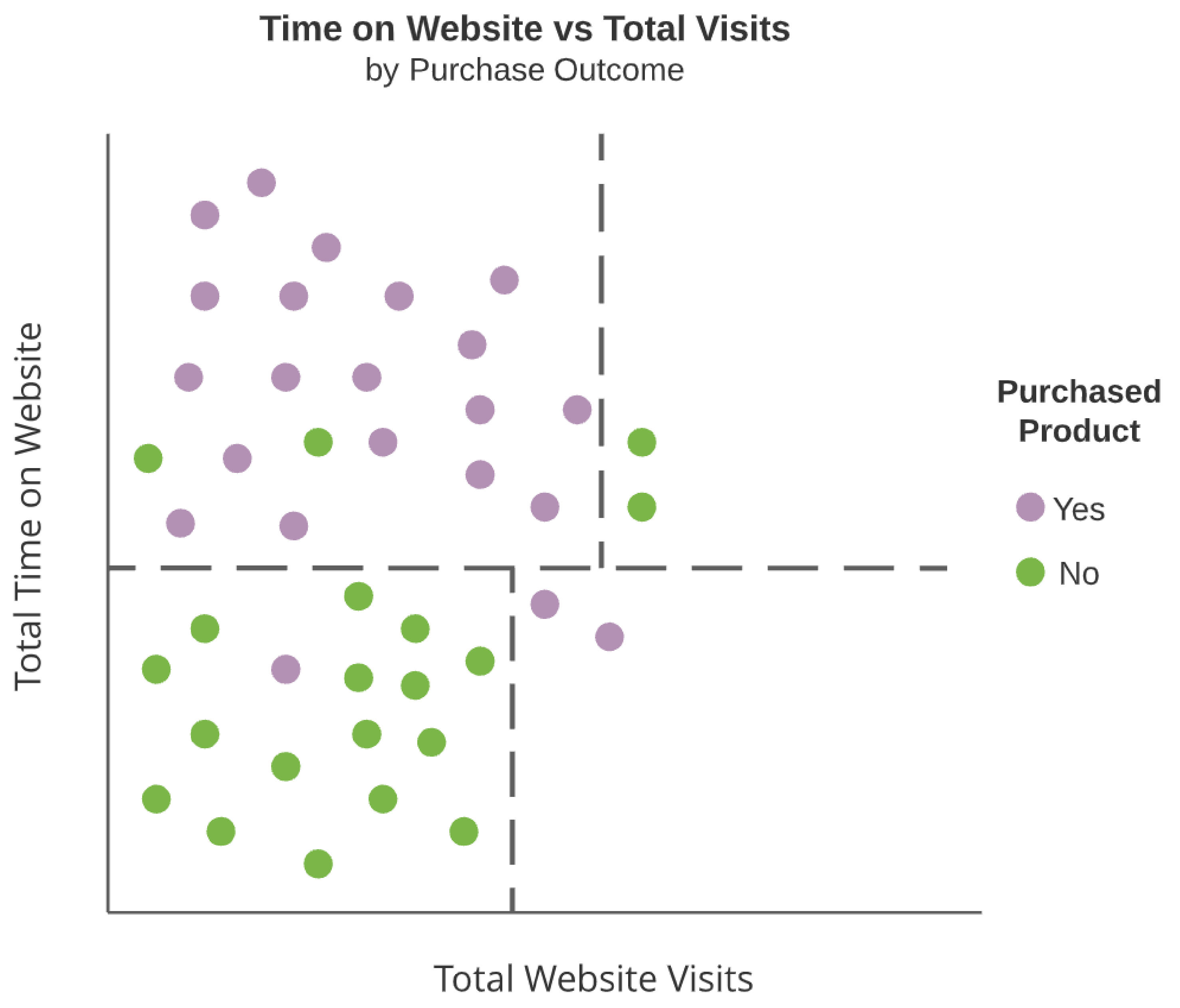


Classification with decision trees

Decision trees segment the predictor space into **rectangular** regions

Recursive binary splitting

- Algorithm that segments predictor space into non-overlapping rectangular regions
- Decision splits are added iteratively
  - Either horizontal or vertical cut points

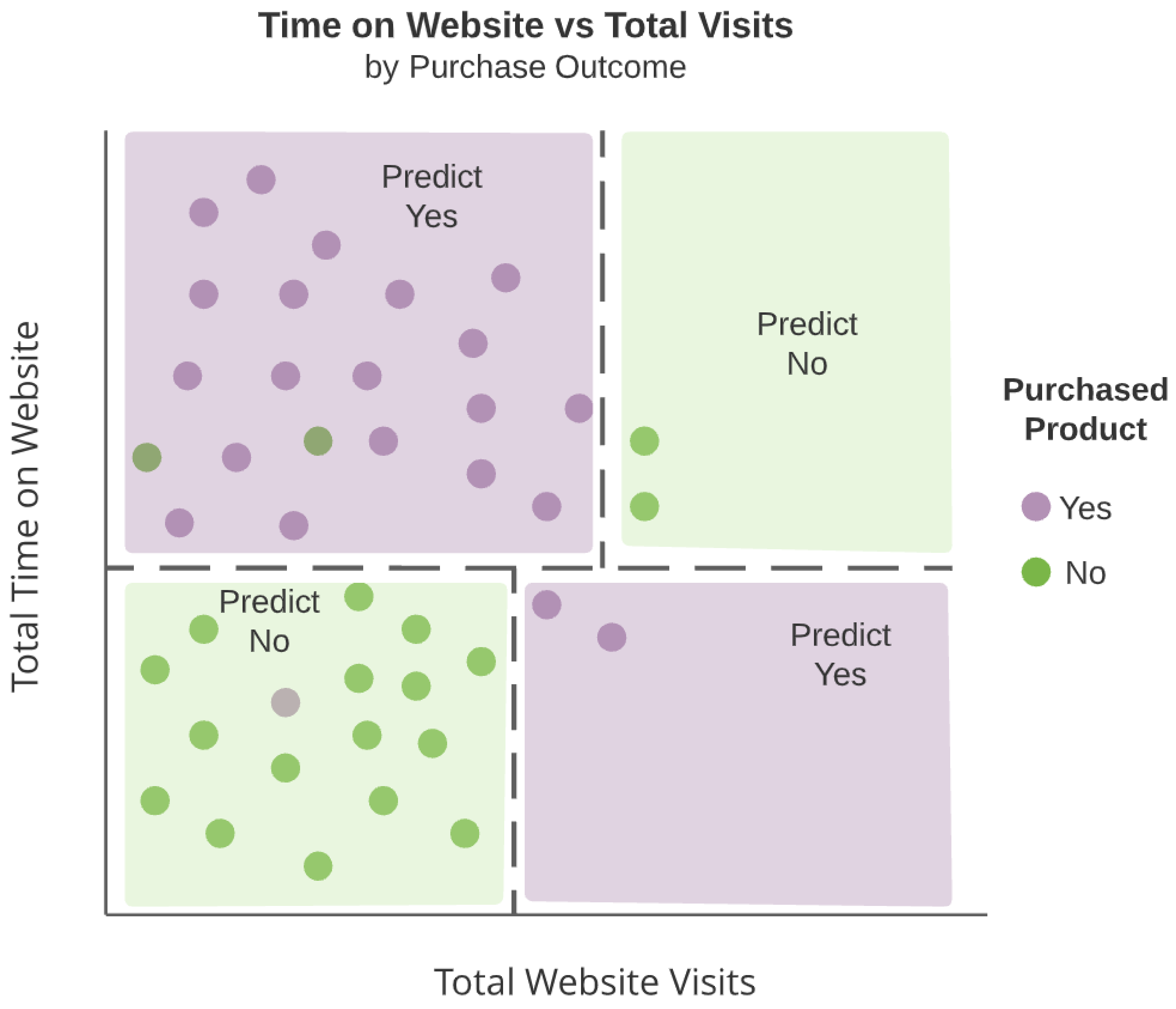


Classification with decision trees

Decision trees segment the predictor space into **rectangular** regions

Recursive binary splitting

- Algorithm that segments predictor space into non-overlapping rectangular regions
- Decision splits are added iteratively
  - Either horizontal or vertical cut points

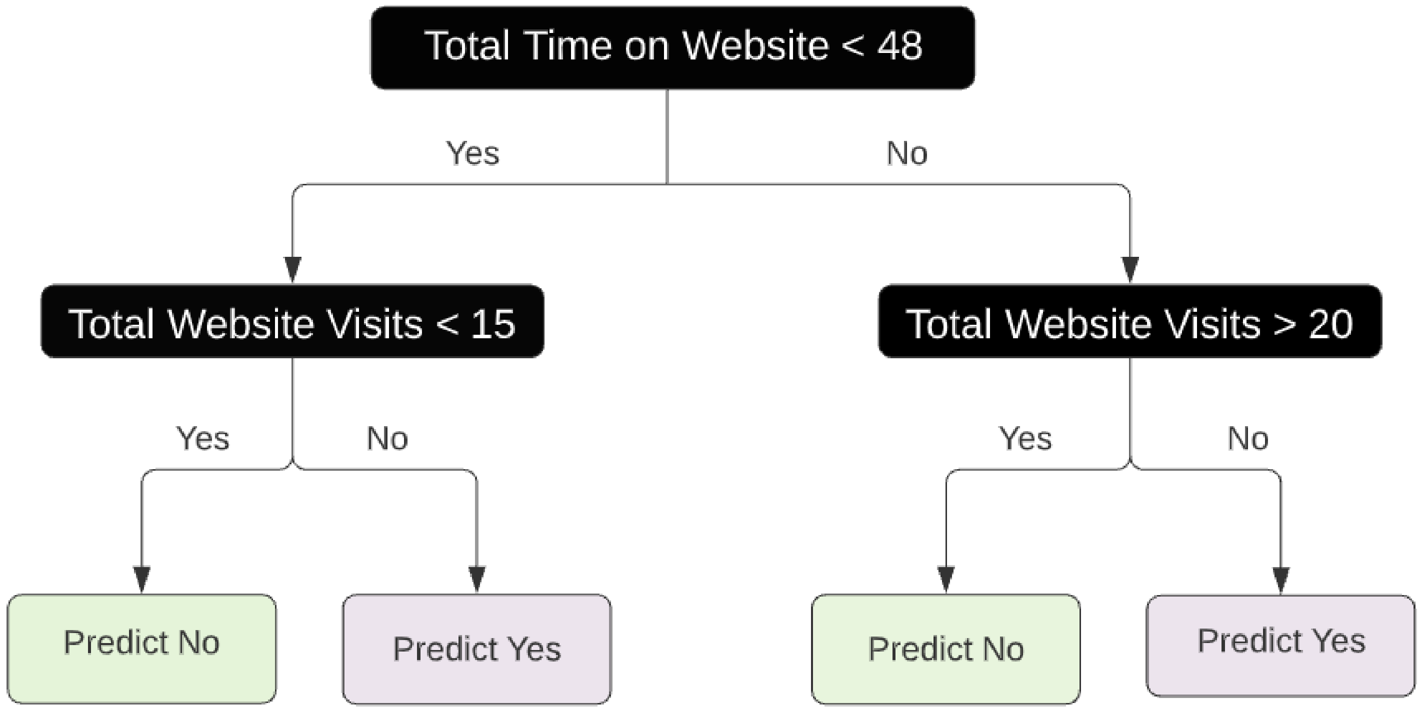


Produces distinct rectangular regions

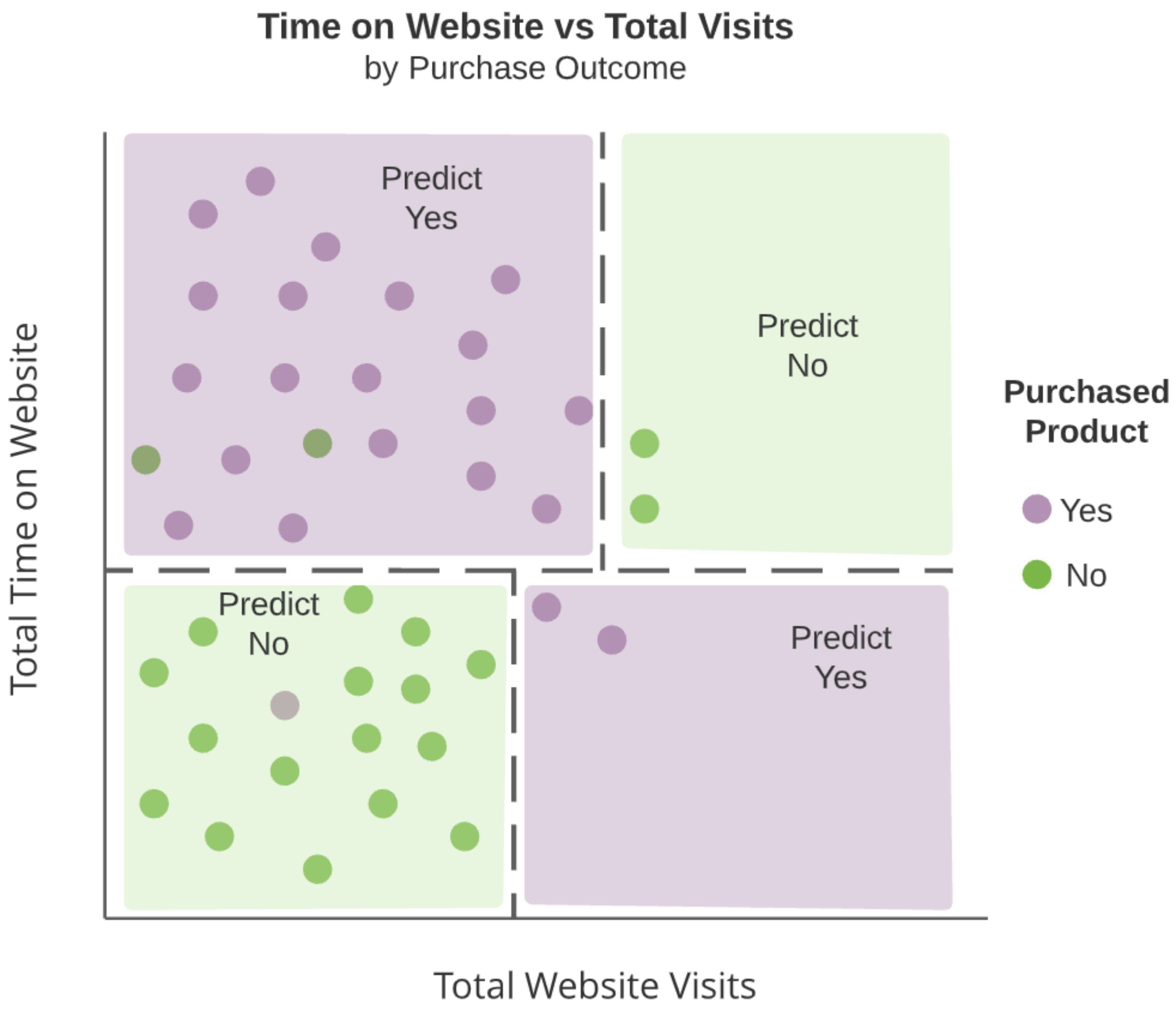
- For classification, majority class is

Tree diagrams

- **Interior nodes**
  - Decision tree splits (dark boxes)
- **Terminal nodes**
  - Regions which are not split further
  - Green and purple boxes



Interior nodes are dashed lines and terminal nodes are highlighted rectangular regions



Model specification

Model specification in `parsnip`

- `decision_tree()`
  - General interface to decision tree models in `parsnip`
  - Common engine is `'rpart'`
  - Mode can be either `'classification'` or `'regression'`
    - For lead scoring data, we need `'classification'`

```
dt_model <- decision_tree() %>%
  set_engine('rpart') %>%
  set_mode('classification')
```

Feature engineering recipe

Data transformations for lead scoring data

- Encoded in a `recipe` object
  - Remove multicollinearity
  - Normalize numeric predictors
  - Create dummy variables for nominal predictors

Two R objects to manage

- `parsnip` model and `recipe` specification
- Combining into one object would make life easier

```
leads_recipe <- recipe(purchased ~ .,
                       data = leads_training) %>%
  step_corr(all_numeric(), threshold = 0.9) %>%
  step_normalize(all_numeric()) %>%
  step_dummy(all_nominal(), -all_outcomes())
```

```
leads_recipe
```

```
Data Recipe
Inputs:
      role #variables
   outcome          1
 predictor          6

Operations:
Correlation filter on all_numeric()
Centering and scaling for all_numeric()
Dummy variables from all_nominal(), -all_outcomes()
```

Combining models and recipes

The `workflows` package is designed for streamlining the model process

- Combines a `parsnip` model and `recipe` object into a single `workflow` object

Initialized with the `workflow()` function

- Add model object with `add_model()`
- Add `recipe` object with `add_recipe()`
  - Must be specification, not a trained `recipe`

```
leads_wkfl <- workflow() %>%
  add_model(dt_model) %>%
  add_recipe(leads_recipe)

leads_wkfl
```

```
== Workflow =====================
Preprocessor: Recipe
Model: decision_tree()
-- Preprocessor ----------------
3 Recipe Steps
* step_corr()
* step_normalize()
* step_dummy()
-- Model -------------------------
Decision Tree Model Specification (classification)
Computational engine: rpart
```

Model fitting with workflows

Training a `workflow` object

- Pass `workflow` to `last_fit()` and provide data split object
- View model evaluation results with `collect_metrics()`

Behind the scenes

- Training and test datasets created
- `recipe` trained and applied
- Decision tree trained with training data
- Predictions and metrics on test data

```
leads_wkfl_fit <- leads_wkfl %>%
  last_fit(split = leads_split)

leads_wkfl_fit %>%
  collect_metrics()
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.771
2 roc_auc  binary         0.775
```

Collecting predictions

A `workflow` trained with `last_fit()` can be passed to `collect_predictions()`

- Produces detailed results on the test data
- Like before, can be used with `yardstick` functions to explore performance custom metrics

```
leads_wkfl_preds <- leads_wkfl_fit %>%
  collect_predictions()

leads_wkfl_preds
```

```
# A tibble: 332 x 6
   id               .pred_yes .pred_no  .row .pred_class purchased
  <chr>                 <dbl>    <dbl> <int>  <fct>          <fct>
train/test split    0.120    0.880     2    no             no
train/test split    0.755    0.245    17    yes            yes
train/test split    0.120    0.880    21    no             no
train/test split    0.120    0.880    22    no             no
train/test split    0.755    0.245    24    yes            yes
# ... with 327 more rows
```

Exploring custom metrics

Create a custom metric set with `metric_set()`

- Area under the ROC curve, sensitivity, and specificity

```
leads_metrics <- metric_set(roc_auc, sens, spec)

leads_wkfl_preds %>%
  leads_metrics(truth = purchased,
                estimate = .pred_class,
                .pred_yes)
```

Pass predictions datasets to `leads_metrics()` to calculate metrics

```
# A tibble: 3 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 sens    binary         0.75
2 spec    binary         0.783
3 roc_auc binary         0.775
```

Loan default dataset

Financial data for consumer loans at a bank

- Outcome variable is `loan_default`

```
loans_df
```

```
# A tibble: 872 x 8
loan_default  loan_purpose   missed_payment_2_yr loan_amount interest_rate installment annual_income debt_to_income
 <fct>            <fct>            <fct>                 <int>         <dbl>        <dbl>         <dbl>        <dbl>
 no          debt_consolidation      no                  25000          5.47         855.         62823        39.4
 yes         medical                 no                  10000         10.2          364.         40000        24.1
 no          small_business          no                  13000          6.22         442.         65000        14.0
 no          small_business          no                  36000          5.97        1152.        125000         8.09
 yes         small_business          yes                 12000         11.8          308.         65000        20.1
# ... with 867 more rows
```

Let's practice building workflows!

MODELING WITH TIDYMODELS IN R

Estimating performance with cross validation

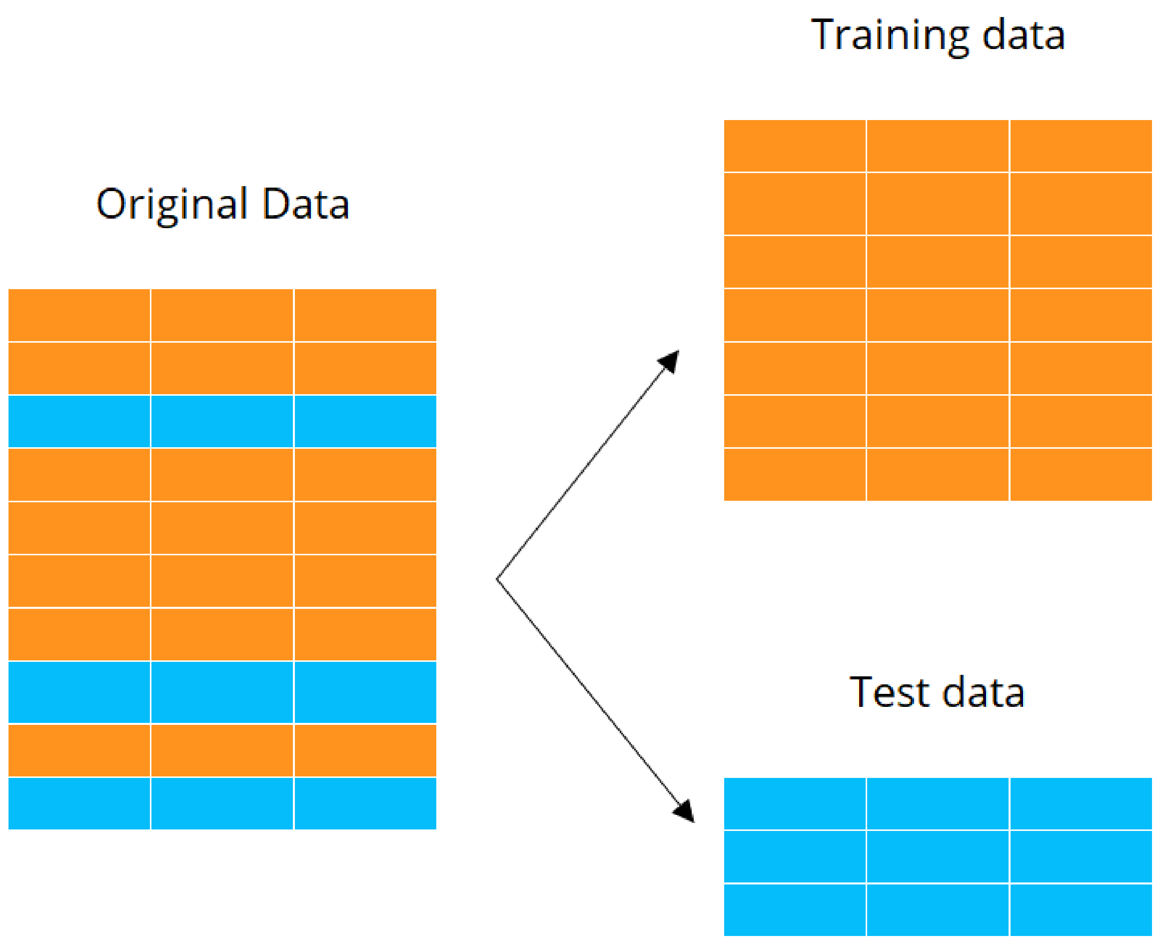
MODELING WITH TIDYMODELS IN R

David Svancer
Data Scientist

Training and test datasets

Creating training and test datasets is the first step in the modeling process

- Guards against **overfitting**
  - Training data is used for model fitting
  - Test data is used for model evaluation



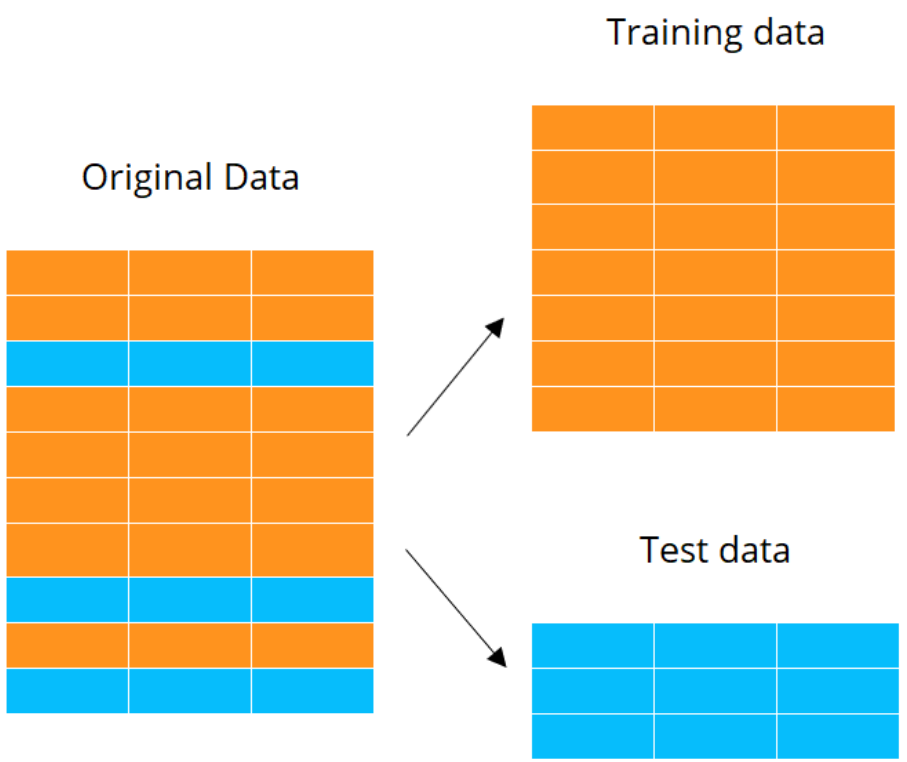
Downside

- Only **one** estimate of model performance

K-fold cross validation

Resampling technique for exploring model performance

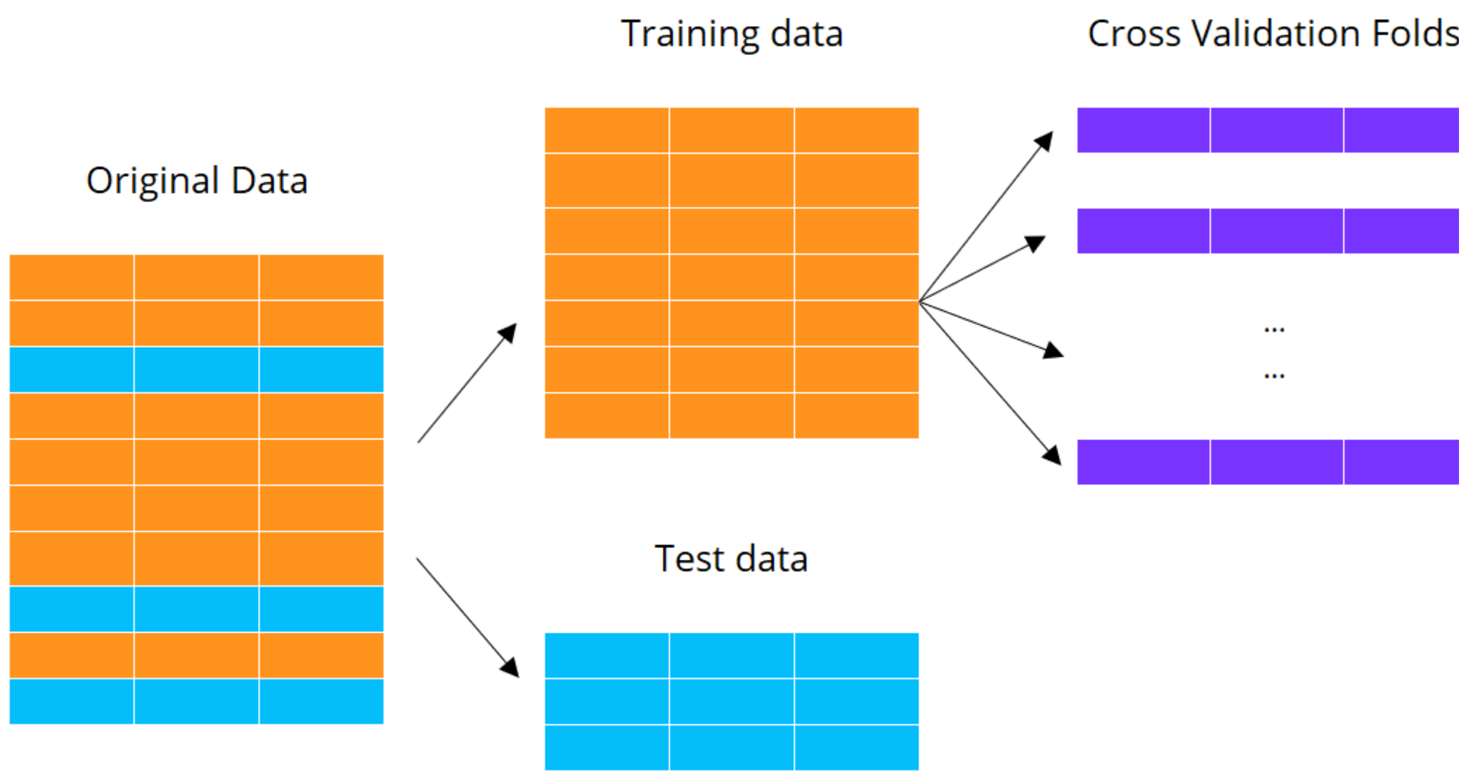
- Provides *K* estimates of model performance during the model fitting process



K-fold cross validation

Resampling technique for exploring model performance

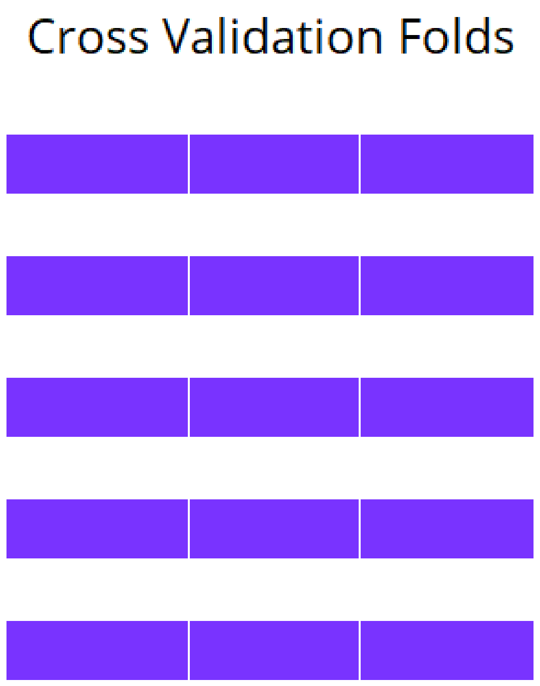
- Provides *K* estimates of model performance during the model fitting process
- Training data is randomly partitioned into *K* sets of roughly equal size
- Folds are used to perform *K* iterations of model fitting and evaluation



Machine learning with cross validation

Performing 5-fold cross validation

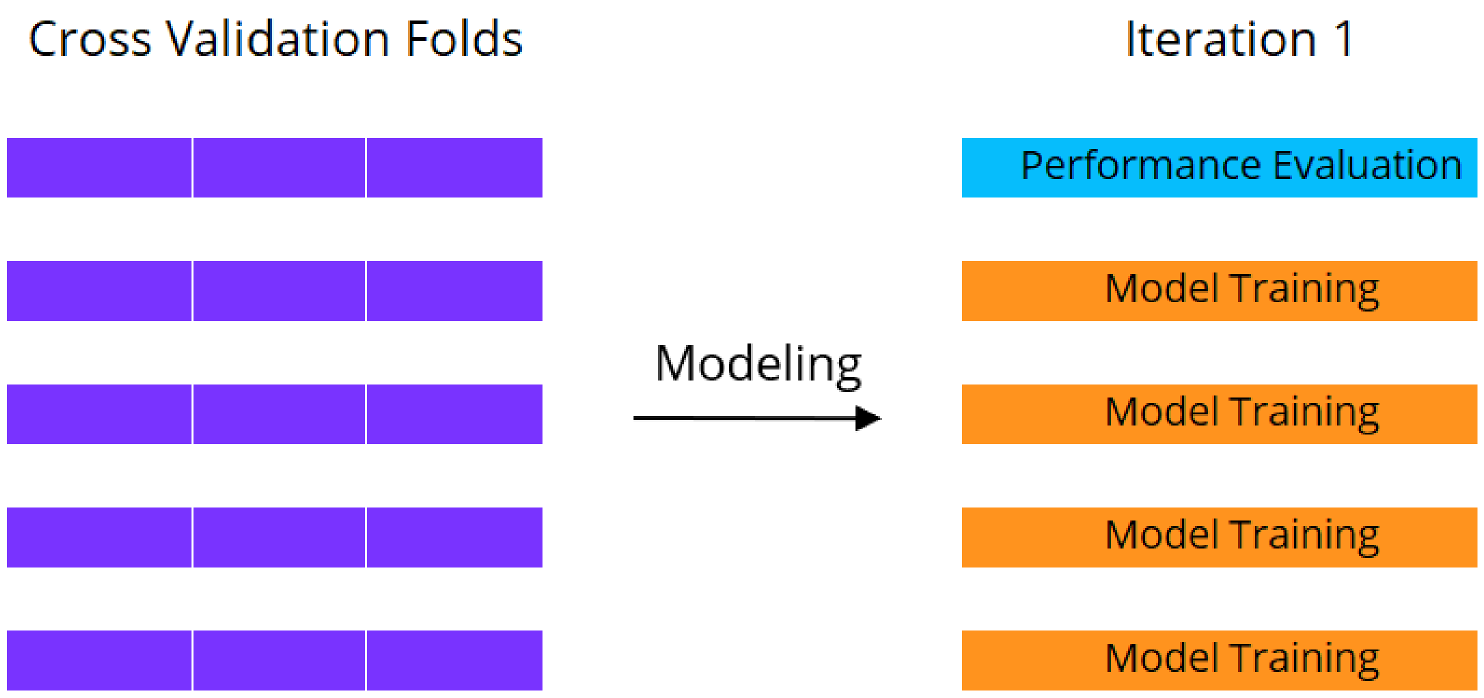
- Five iterations of model training and evaluation



Machine learning with cross validation

Performing 5-fold cross validation

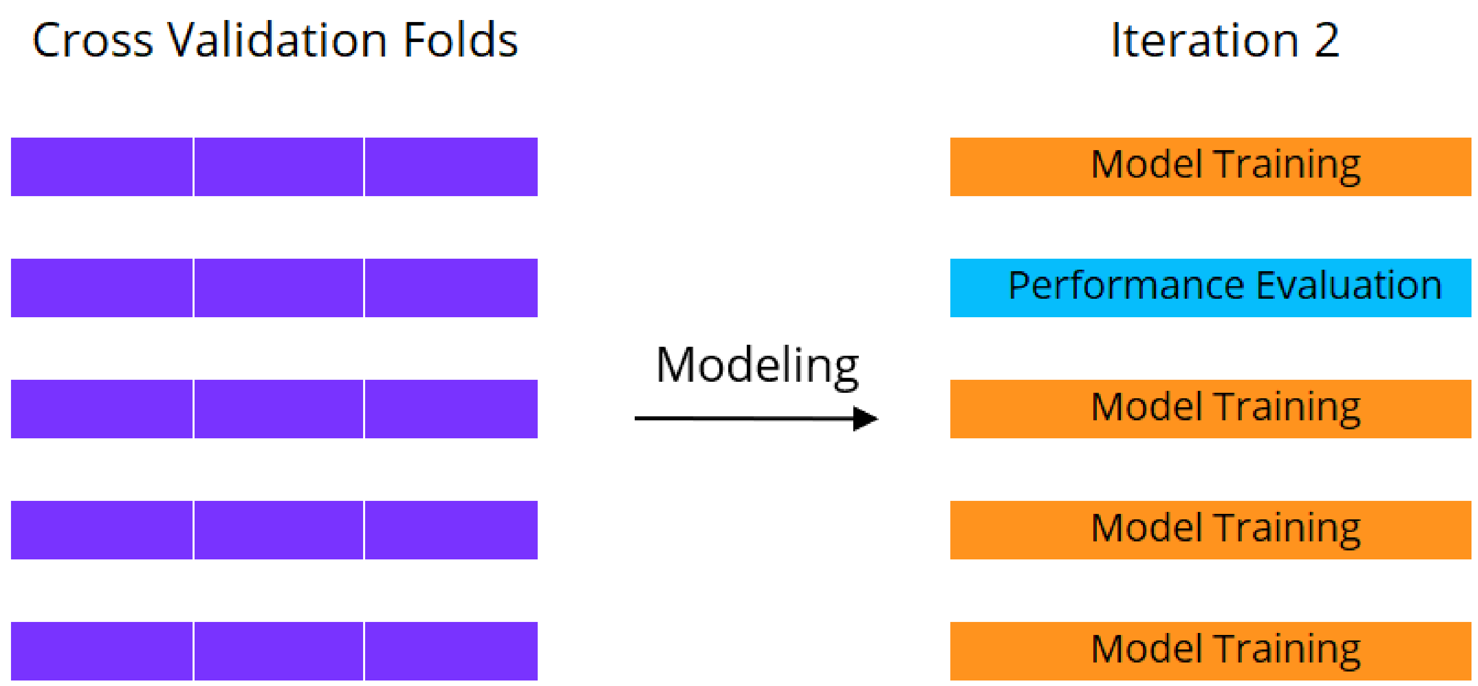
- Five iterations of model training and evaluation
- Iteration 1
  - Fold 1 reserved for model evaluation and folds 2 through 5 for model training



Machine learning with cross validation

Performing 5-fold cross validation

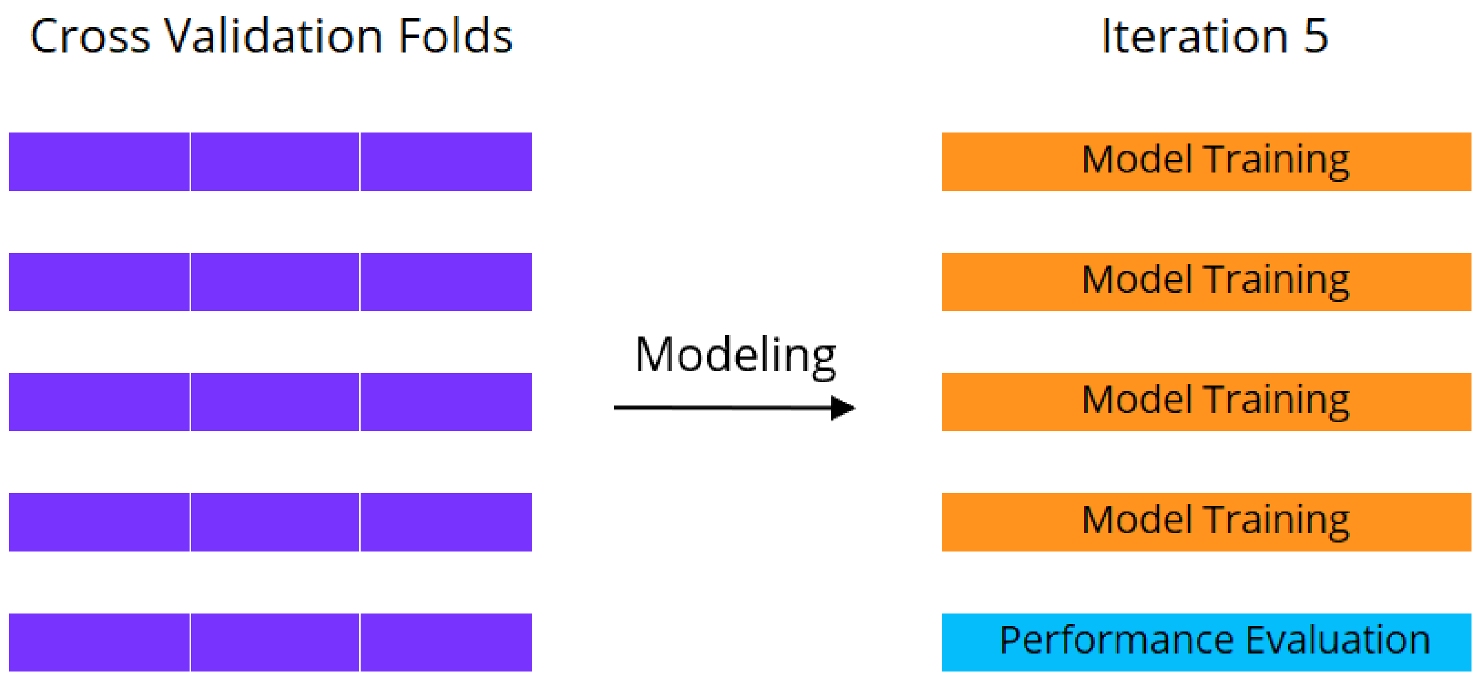
- Five iterations of model training and evaluation
- Iteration 1
  - Fold 1 reserved for model evaluation and folds 2 through 5 for model training
- Iteration 2
  - Fold 2 reserved for model evaluation



Machine learning with cross validation

Performing 5-fold cross validation

- Five iterations of model training and evaluation
- Iteration 1
  - Fold 1 reserved for model evaluation and folds 2 through 5 for model training
- Iteration 2
  - Fold 2 reserved for model evaluation



Five estimates of model performance in total

Creating cross validation folds

The `vfold_cv()` function

- Training data
- Number of folds, `v`
- Stratification variable, `strata`
- Execute `set.seed()` before `vfold_cv()` for reproducibility
- `splits`
  - **List column** with data split objects for creating fold

```
set.seed(214)
leads_folds <- vfold_cv(leads_training,
                        v = 10,
                        strata = purchased)
leads_folds
```

```
#  10-fold cross-validation using stratification
# A tibble: 10 x 2
   splits             id
   <list>             <chr>
 1 <split [896/100]> Fold01
 2 <split [896/100]> Fold02
 3 <split [896/100]> Fold03
 . .................  ......
 9 <split [897/99]>  Fold09
10 <split [897/99]>  Fold10
```

Model training with cross validation

The `fit_resamples()` function

- Train a `parsnip` model or `workflow` object
- Provide cross validation folds, `resamples`
- Optional custom metric function, `metrics`
  - Default is accuracy and ROC AUC

Each metric is estimated 10 times

- One estimate per fold
- Average value in `mean` column

```
leads_rs_fit <- leads_wkfl %>%
  fit_resamples(resamples = leads_folds,
                metrics = leads_metrics)

leads_rs_fit %>%
  collect_metrics()
```

```
# A tibble: 3 x 5
  .metric .estimator  mean     n std_err
  <chr>   <chr>      <dbl> <int>   <dbl>
1 roc_auc binary     0.823    10  0.0147
2 sens    binary     0.786    10  0.0203
3 spec    binary     0.855    10  0.0159
```

Detailed cross validation results

The `collect_metrics()` function

- Passing `summarize = FALSE` will provide all metric estimates for every cross validation fold
- 30 total combinations (3 metrics x 10 folds)
  - `.metric` column identifies metric
  - `.estimate` column gives estimated value for each fold

```
rs_metrics <- leads_rs_fit %>%
  collect_metrics(summarize = FALSE)

rs_metrics
```

```
# A tibble: 30 x 4
   id     .metric .estimator .estimate
   <chr>  <chr>   <chr>          <dbl>
 1 Fold01 sens    binary         0.861
 2 Fold01 spec    binary         0.891
 3 Fold01 roc_auc binary         0.885
 4 Fold02 sens    binary         0.778
 5 Fold02 spec    binary         0.969
 6 Fold02 roc_auc binary         0.885
# ... with 24 more rows
```

Summarizing cross validation results

The `collect_metrics()` function returns a tibble

- Results can be summarized with `dplyr`
  - Start with `rs_metrics`
  - Form groups by `.metric` values
  - Calculate summary statistics with `summarize()`

```
rs_metrics %>%
  group_by(.metric) %>%
  summarize(min = min(.estimate),
            median = median(.estimate),
            max = max(.estimate),
            mean = mean(.estimate),
            sd = sd(.estimate))
```

```
# A tibble: 3 x 6
 .metric   min  median   max   mean     sd
  <chr>   <dbl>  <dbl>  <dbl>  <dbl>   <dbl>
1 roc_auc 0.758  0.806  0.885  0.823   0.0466
2 sens    0.667  0.792  0.861  0.786   0.0642
3 spec    0.810  0.843  0.969  0.855   0.0502
```

Cross validation methodology

Models trained with `fit_resamples()` **are not** able to provide predictions on new data sources

- `predict()` function does not accept resample objects

Purpose of `fit_resample()`

- Explore and compare the performance profile of different model types
- Select best performing model type and focus on model fitting efforts

```
predict(leads_rs_fit,
        new_data = leads_test)
```

```
Error in UseMethod("predict") :
  no applicable method for 'predict' applied to
  an object of class
  "c('resample_results',
      'tune_results',
      'tbl_df',
      'tbl', 'data.frame')"
```

Let's cross validate!

MODELING WITH TIDYMODELS IN R

Hyperparameter tuning

MODELING WITH TIDYMODELS IN R

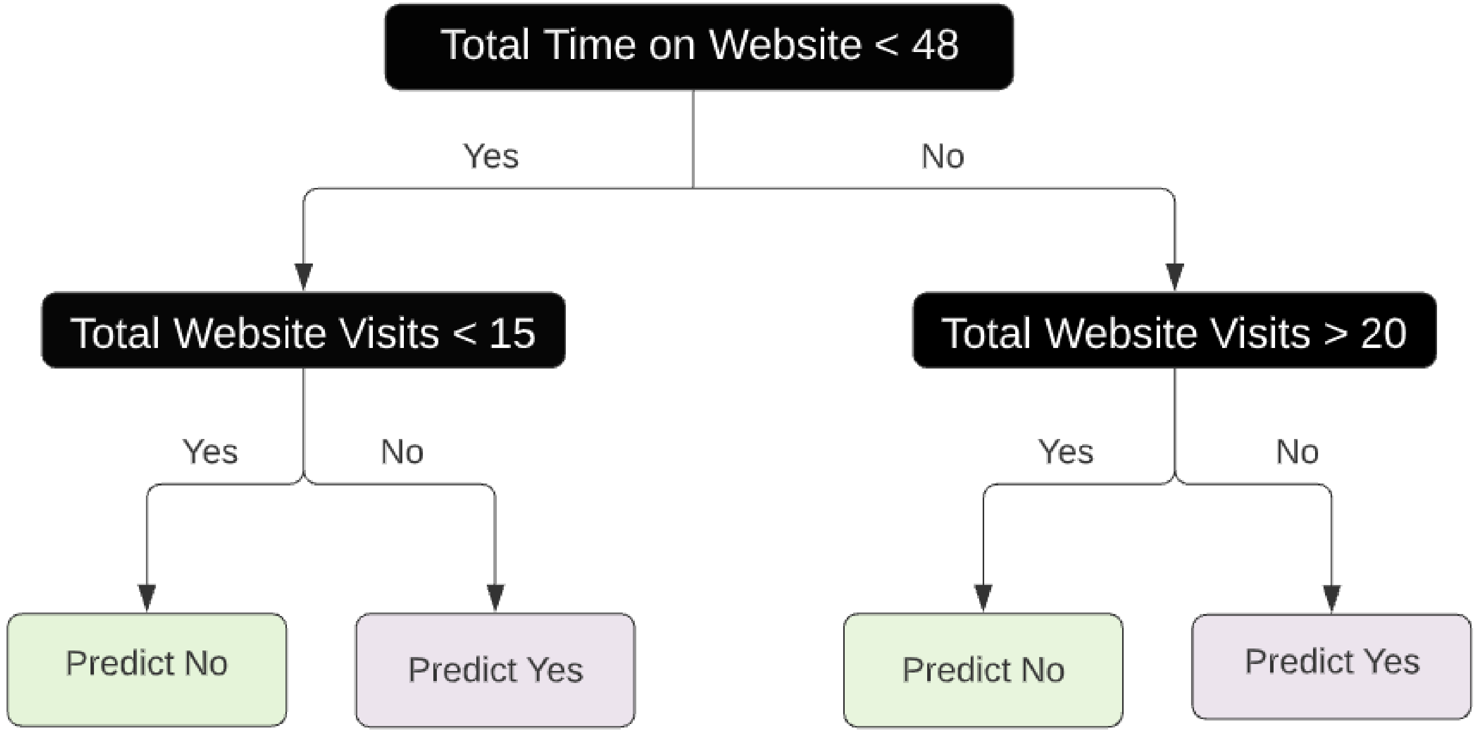
David Svancer
Data Scientist

Hyperparameters

Model parameters whose values are set prior to model training and control model complexity

`parsnip` **decision tree**

- `cost_complexity`
  - Penalizes large number of terminal nodes
- `tree_depth`
  - Longest path from root to terminal node
- `min_n`
  - Minimum data points required in a node for further splitting



Default hyperparameter values

`decision_tree()` function sets default hyperparameter values

- `cost_complexity` is set to 0.01
- `tree_depth` is set to 30
- `min_n` is set to 20

These may not be the best values for all datasets

- **Hyperparameter tuning**
  - Process of using cross validation to find the optimal set of hyperparameter values

```
dt_model <- decision_tree() %>%
  set_engine('rpart') %>%
  set_mode('classification')
```

Labeling hyparameters for tuning

The `tune()` function from the `tune` package

- To label hyperparameters for tuning, set them equal to `tune()` in `parsnip` model specification
- Creates model object with tuning parameters
  - Will let other functions know that they need to be optimized

```
dt_tune_model <- decision_tree(cost_complexity = tune(),
                               tree_depth = tune(),
                               min_n = tune()) %>%
  set_engine('rpart') %>%
  set_mode('classification')

dt_tune_model
```

```
Decision Tree Model Specification (classification)

Main Arguments:
  cost_complexity = tune()
  tree_depth = tune()
  min_n = tune()

Computational engine: rpart
```

Creating a tuning workflow

`workflow` objects can be easily updated

- Prior `leads_wkfl`
  - Feature engineering steps for lead scoring data and decision tree model with default hyperparameters
- Pass `leads_wkfl` to `update_model()` and provide new decision tree model with tuning parameters

```
leads_tune_wkfl <- leads_wkfl %>%
  update_model(dt_tune_model)

leads_tune_wkfl
```

```
== Workflow ===============
Preprocessor: Recipe
Model: decision_tree()
-- Preprocessor -----------
3 Recipe Steps
* step_corr()
* step_normalize()
* step_dummy()
-- Model ------------------
Decision Tree Model Specification (classification)
Main Arguments: cost_complexity = tune()
                tree_depth = tune()
                min_n = tune()
Computational engine: rpart
```

Grid search

Most common method for tuning hyperparameters

- Generate a grid of unique combinations of hyperparameter values
  - For each combination, use cross validation to estimate model performance
- Choose best performing combination

cost_complexity	tree_depth	min_n
0.001	20	35
0.001	20	15
0.001	35	35
0.001	35	15
0.2	20	35
...	...	...

Identifying hyperparameters

The `parameters()` function from the `dials` package

- Takes a `parsnip` model object
- Returns a tibble with the hyperparameters labeled by the `tune()` function, if any
  - Used for generating tuning grids with the `dials` package

```
parameters(dt_tune_model)
```

```
Collection of 3 parameters for tuning

  identifier             type     object
cost_complexity  cost_complexity nparam[+]
tree_depth       tree_depth      nparam[+]
min_n            min_n           nparam[+]
```

Random grid

Generating random combinations

- This method tends to provide greater chances of finding optimal hyperparameter values

The `grid_random()` function

- First argument is the results of the `parameters()` function
- `size` sets the number of random combinations to generate
  - Execute `set.seed()` function before `grid_random()` for reproducibility

```
set.seed(214)
grid_random(parameters(dt_tune_model),
            size = 5)
```

```
# A tibble: 5 x 3
  cost_complexity  tree_depth min_n
            <dbl>       <int> <int>
1    0.0000000758          14    39
2    0.0243                 5    34
3    0.00000443            11     8
4    0.000000600            3     5
5    0.00380                5    36
```

Saving a tuning grid

First step in hyperparameter tuning

- Create and save a tuning grid
- `dt_grid` contains 5 random combinations of hyperparameter values

```
set.seed(214)
dt_grid <- grid_random(parameters(dt_tune_model),
                       size = 5)

dt_grid
```

```
# A tibble: 5 x 3
  cost_complexity  tree_depth min_n
            <dbl>       <int> <int>
1    0.0000000758          14    39
2    0.0243                 5    34
3    0.00000443            11     8
4    0.000000600            3     5
5    0.00380                5    36
```

Hyperparameter tuning with cross validation

The `tune_grid()` function performs hyperparameter tuning

Takes the following arguments:

- `workflow` or `parsnip` model
- Cross validation object, `resamples`
- Tuning grid, `grid`
- Optional `metrics` function

Returns tibble of results

- `.metrics`
  - List column with results for each fold

```
dt_tuning <- leads_tune_wkfl %>%
             tune_grid(resamples = leads_folds,
                       grid = dt_grid,
                       metrics = leads_metrics)
```

```
dt_tuning
```

```
# Tuning results
# 10-fold cross-validation using stratification
# A tibble: 10 x 4
   splits             id        .metrics          ..
   <list>            <chr>       <list>           ..
<split [896/100]>   Fold01   <tibble [15 x 7]>  ..
................    ......   ...............    ..
<split [897/99]>    Fold09   <tibble [15 x 7]>  ..
<split [897/99]>    Fold10   <tibble [15 x 7]>  ..
```

Exploring tuning results

The `collect_metrics()` function provides summarized results by default

- Average estimated metric values across all folds per combination

```
dt_tuning %>%
  collect_metrics()
```

```
# A tibble: 15 x 9
   cost_complexity tree_depth min_n .metric .estimator  mean     n std_err .config
             <dbl>      <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
 1    0.0000000758         14    39 roc_auc binary     0.827    10 0.0147  Model1
 2    0.0000000758         14    39 sens    binary     0.728    10 0.0277  Model1
 3    0.0000000758         14    39 spec    binary     0.865    10 0.0156  Model1
 4    0.0243                5    34 roc_auc binary     0.823    10 0.0147  Model2
 .    ......               ..    .. ....    ......     .....    .. .....   ......
14    0.00380               5    36 sens    binary     0.747    10 0.0209  Model5
15    0.00380               5    36 spec    binary     0.858    10 0.0161  Model5
```

Let's get tuning!

MODELING WITH TIDYMODELS IN R

Selecting the best model

MODELING WITH TIDYMODELS IN R

David Svancer
Data Scientist

Detailed tuning results

The `collect_metrics()` function provides summarized results by default

- Passing `summarize = FALSE` will provide all hyperparameter tuning results

```
dt_tuning %>%
  collect_metrics(summarize = FALSE)
```

```
# A tibble: 150 x 8
 id      cost_complexity tree_depth min_n .metric  ...  .estimate  .config
<chr>          <dbl>          <int>   <int>  <chr>    ...    <dbl>       <chr>
Fold01    0.0000000758      14        39    sens     ...     0.75      Model1
Fold01    0.0000000758      14        39    spec     ...     0.906     Model1
Fold01    0.0000000758      14        39    roc_auc ...     0.888     Model1
.....     ............      ..        ..    ......   ...     .....     ......
Fold10    0.00380           5         36    roc_auc ...     0.789     Model5
```

Exploring tuning results

Selecting `summarise = FALSE` within `collect_metrics()` returns a tibble

- Easy to explore results with `dplyr`
- Exploring ROC AUC
  - Select `roc_auc` metric
  - Form groups by `id` column
  - Calculate `.estimate` summary statistics

```
dt_tuning %>%
  collect_metrics(summarize = FALSE) %>%
  filter(.metric == 'roc_auc') %>%
  group_by(id) %>%
  summarize(min_roc_auc = min(.estimate),
            median_roc_auc = median(.estimate),
            max_roc_auc = max(.estimate))
```

```
# A tibble: 10 x 4
 id      min_roc_auc  median_roc_auc  max_roc_auc
<chr>        <dbl>          <dbl>        <dbl>
Fold01       0.830          0.885        0.888
Fold02       0.857          0.882        0.885
Fold03       0.818          0.836        0.836
......       ....           ....         ....
Fold10       0.762          0.790        0.813
```

Viewing the best performing models

The `show_best()` function

- Displays the top `n` performing models based on average value of `metric`
- `Model1` is the winner

```
dt_tuning %>%
  show_best(metric = 'roc_auc', n = 5)
```

```
# A tibble: 5 x 9
cost_complexity  tree_depth  min_n  .metric .estimator   mean    n    std_err   .config
    <dbl>            <int>    <int>    <chr>    <chr>       <dbl>  <int>  <dbl>     <chr>
0.0000000758          14       39     roc_auc   binary     0.827   10   0.0147    Model1
0.00380                5       36     roc_auc   binary     0.825   10   0.0146    Model5
0.0243                 5       34     roc_auc   binary     0.823   10   0.0147    Model2
0.00000443            11       8      roc_auc   binary     0.816   10   0.00786   Model3
0.000000600            3       5      roc_auc   binary     0.814   10   0.0131    Model4
```

Selecting a model

The `select_best()` function

- Pass `dt_tuning` results to `select_best()`
- Select the `metric` on which to evaluate performance

```
best_dt_model <- dt_tuning %>%
  select_best(metric = 'roc_auc')

best_dt_model
```

Returns a tibble with the best performing model and hyperparameter values

```
# A tibble: 1 x 4
cost_complexity tree_depth  min_n  .config
     <dbl>           <int>   <int>  <chr>
0.0000000758          14      39    Model1
```

Finalizing the workflow

The `finalize_workflow()` function will finalize a `workflow` that contains a model object with tuning parameters

- Pass `workflow` object
- A tibble with one row of final model hyperparameter values
  - Column names must match hyperparameters in model object

Returns a `workflow` object with set hyperparameter values

```
final_leads_wkfl <- leads_tune_wkfl %>%
  finalize_workflow(best_dt_model)
final_leads_wkfl
```

```
== Workflow ==========================================
Preprocessor: Recipe
Model: decision_tree()
-- Preprocessor ------------------------------------
3 Recipe Steps
* step_corr()
* step_normalize()
* step_dummy()
-- Model ---------------------------------------------
Decision Tree Model Specification (classification)
Main Arguments:
  cost_complexity = 0.0000000758
  tree_depth = 14
  min_n = 39
Computational engine: rpart
```

Model fitting

Finalized `workflow` object can be trained with `last_fit()` and original data split object, `leads_split`

```
leads_final_fit <- final_leads_wkfl %>%
  last_fit(split = leads_split)

leads_final_fit %>%
  collect_metrics()
```

```
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.771
2 roc_auc  binary         0.793
```

Behind the scenes

- Training and test datasets created
- `recipe` trained and applied
- **Tuned decision tree** trained with entire training dataset
- Predictions and metrics on test data

Let's practice!

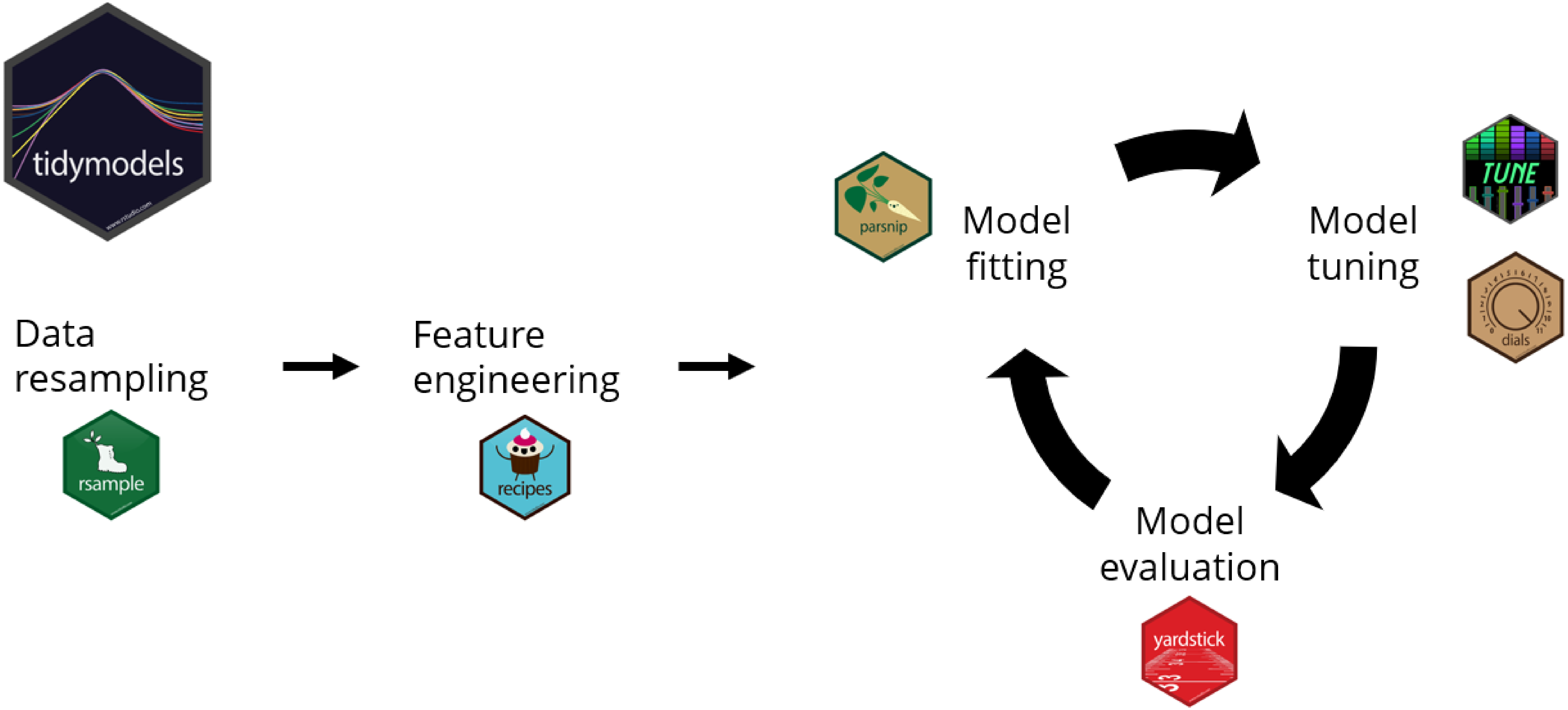
MODELING WITH TIDYMODELS IN R

Congratulations!

MODELING WITH TIDYMODELS IN R

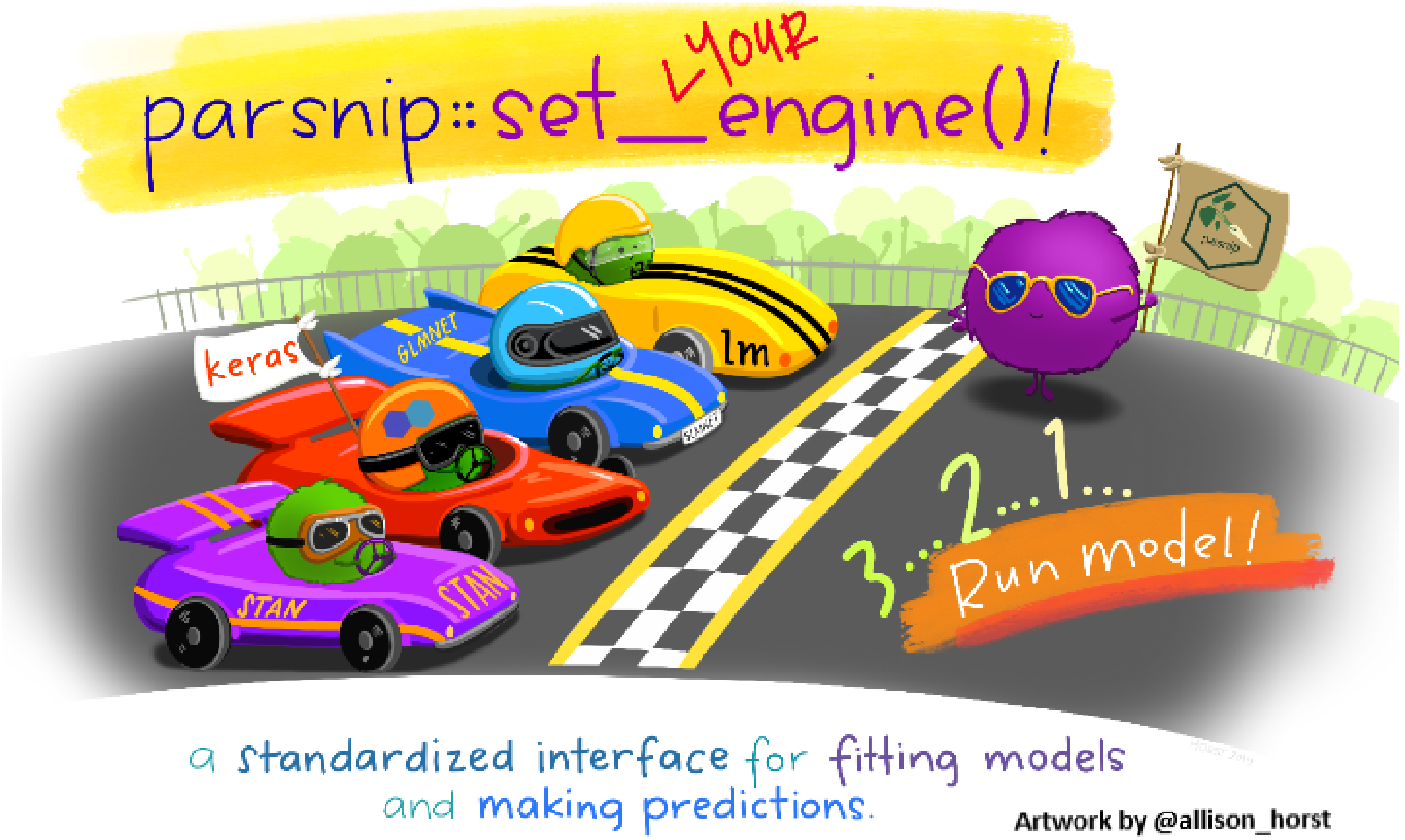
David Svancer
Data Scientist

The tidymodels ecosystem



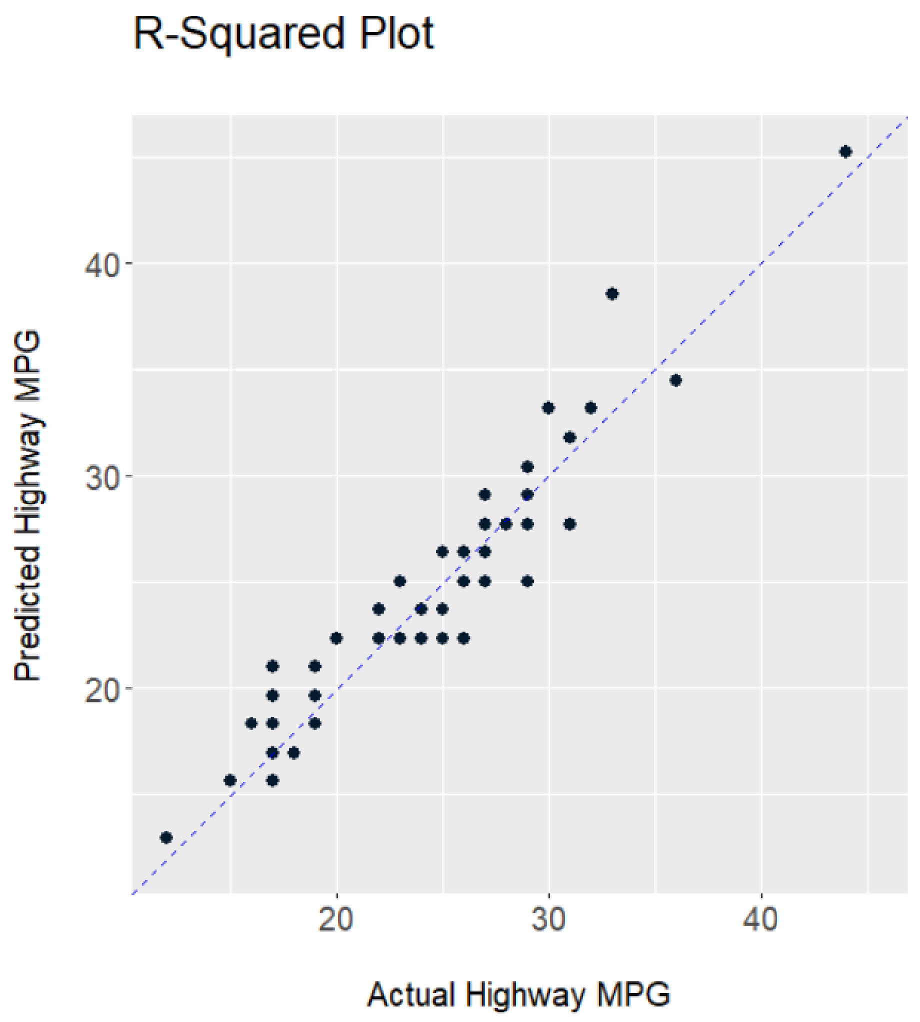
Regression modeling

Specifying models with `parsnip`



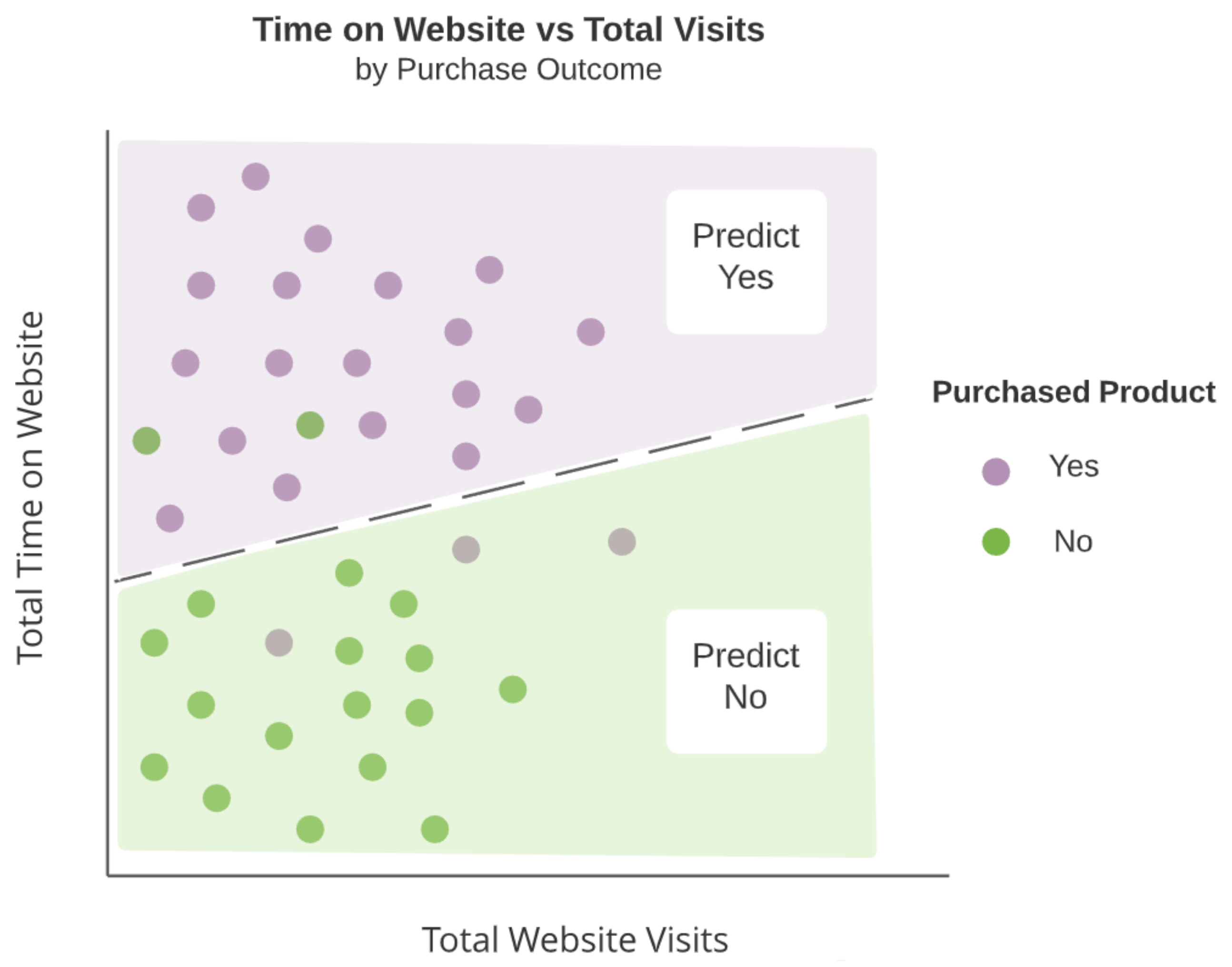
Artwork by @allison_horst

Training and evaluating linear regression models

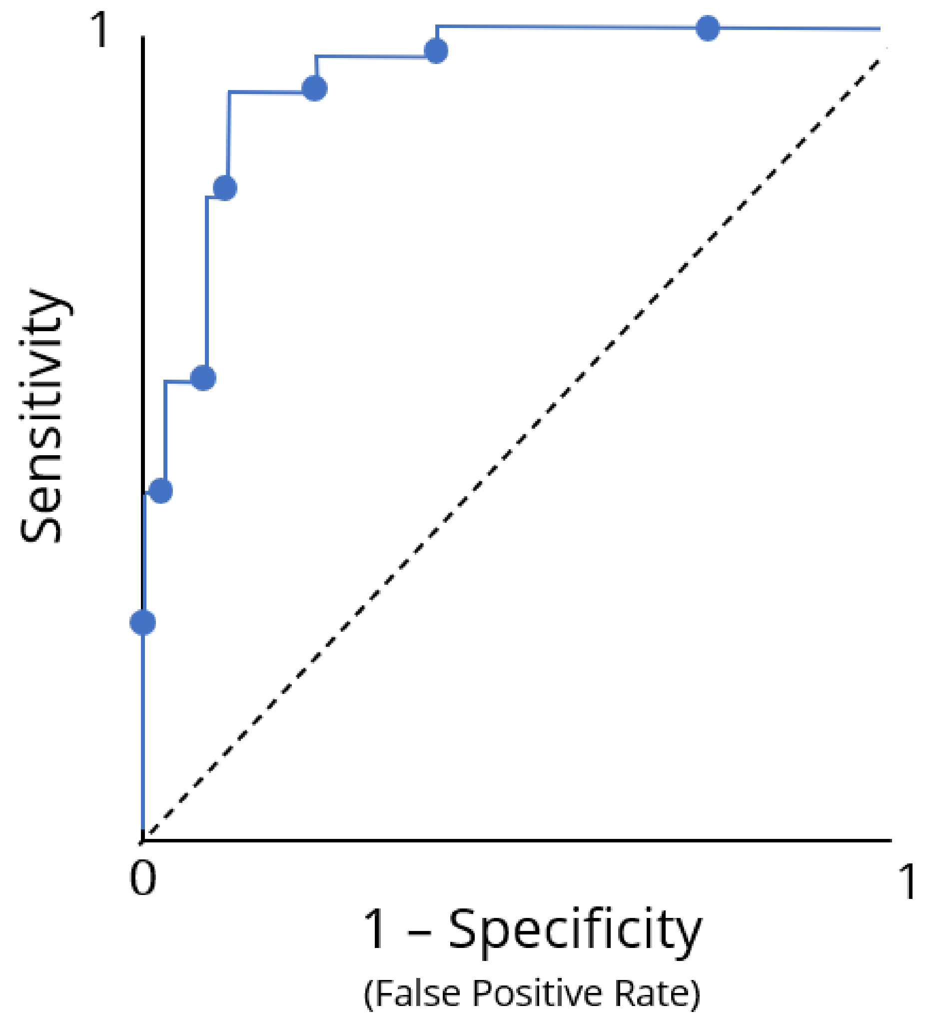


Classification modeling

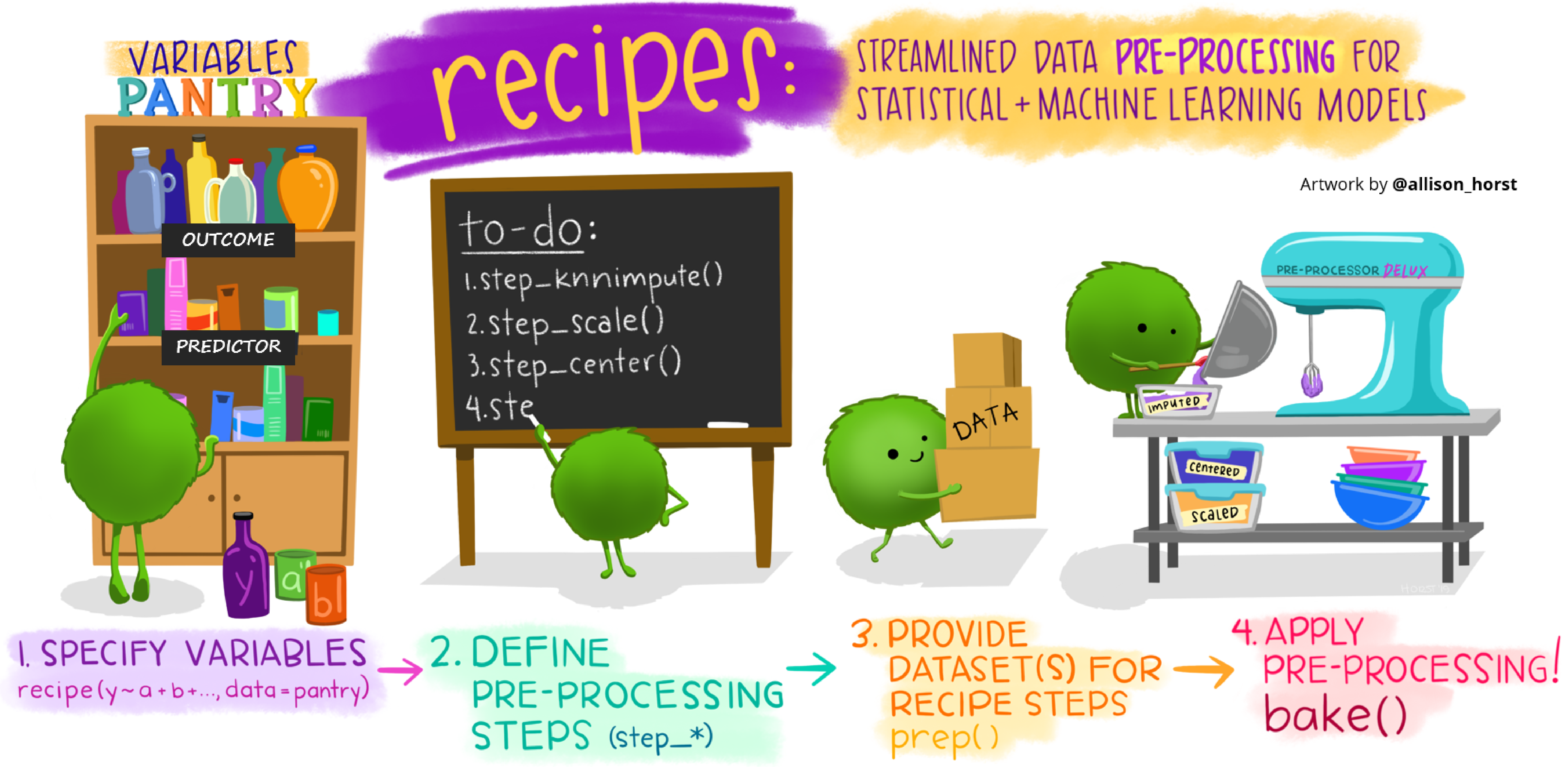
Logistic regression with `logistic_reg()`



Evaluating classification performance with confusion matrices and ROC curves



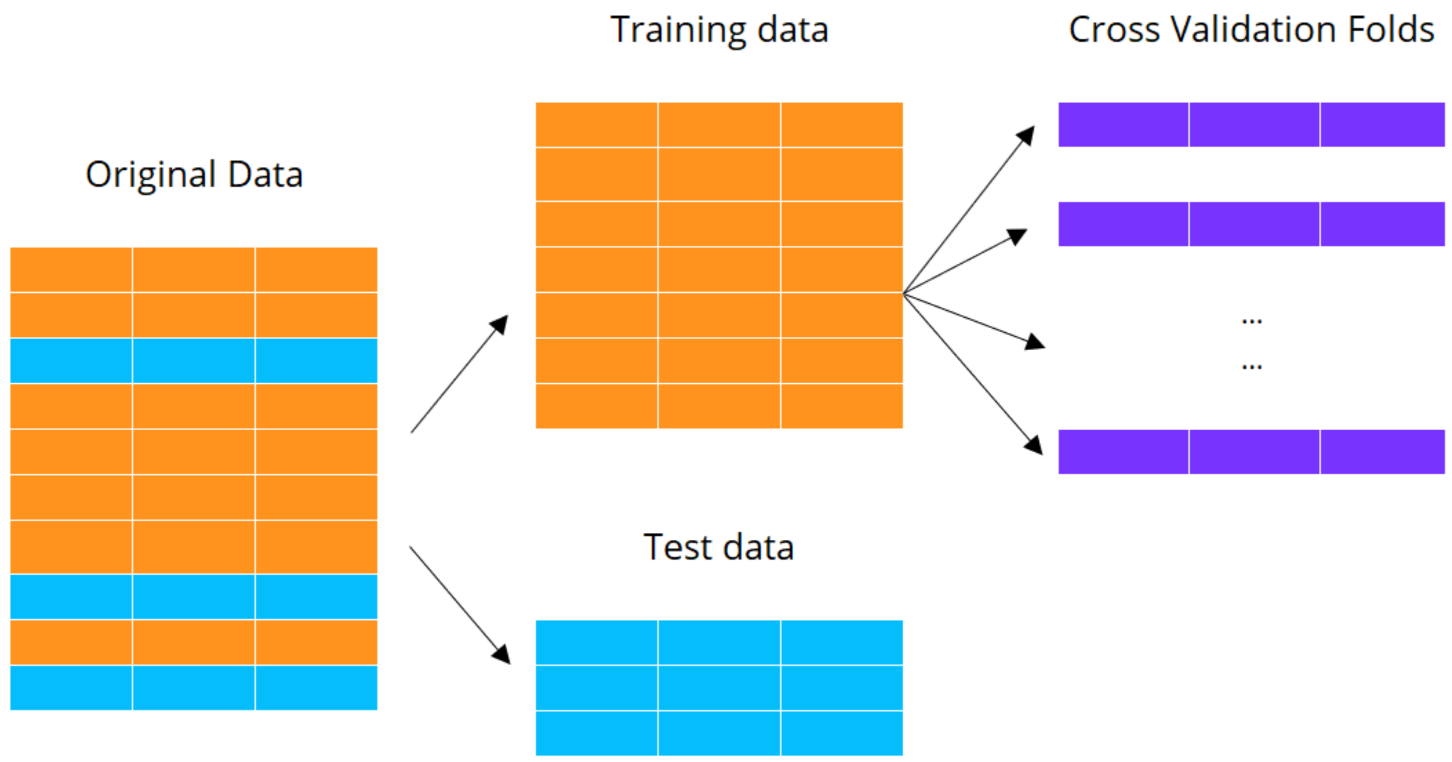
Feature engineering



Fine tuning models with cross validation

Model performance profiles with cross validation and `fit_resamples()`

- Hyperparameter tuning with grid search
- Finalizing model workflows



cost_complexity	tree_depth	min_n
0.001	20	35
0.001	20	15
0.001	35	35
0.001	35	15
0.2	20	35
...	...	...

Thank you!

MODELING WITH TIDYMODELS IN R