Tidymodels

Xavier Gendre, Nicolas Klutchnikoff, Martial Krawier

Introduction

Chargement des packages

Dans cette section, nous allons apprendre à utiliser un ensemble de packages destinés à la modélisation statistique.

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.1.0     v dplyr   1.0.2
## v tidyr   1.1.2     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.0
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(tidymodels)
```

```
## -- Attaching packages -------------------------------------- tidymodels 0.1.2 --
```

```
## v broom     0.7.3      v recipes   0.1.15
## v dials     0.0.9      v rsample   0.0.9
## v infer     0.5.4      v tune      0.1.3
```

Prérecquis

- Tidyverse
- Connaissances statistiques
- Outils usuels (lm, anova, glmnet)

Travail typique :

- Importations des données
- Exploration/nettoyage/mise en forme
- modélisation
- vérification des performances
- comparaison de modèles

{Recipes}

Présentation

On utilisera des données sur l'habitat à Ames, Iowa. L'un des objectifs que l'on peut poursuivre (et qui a été proposé dans un projet Kaggle) est de prédire le prix de vente des logements.

```
data(ames)
Ames <- ames
```

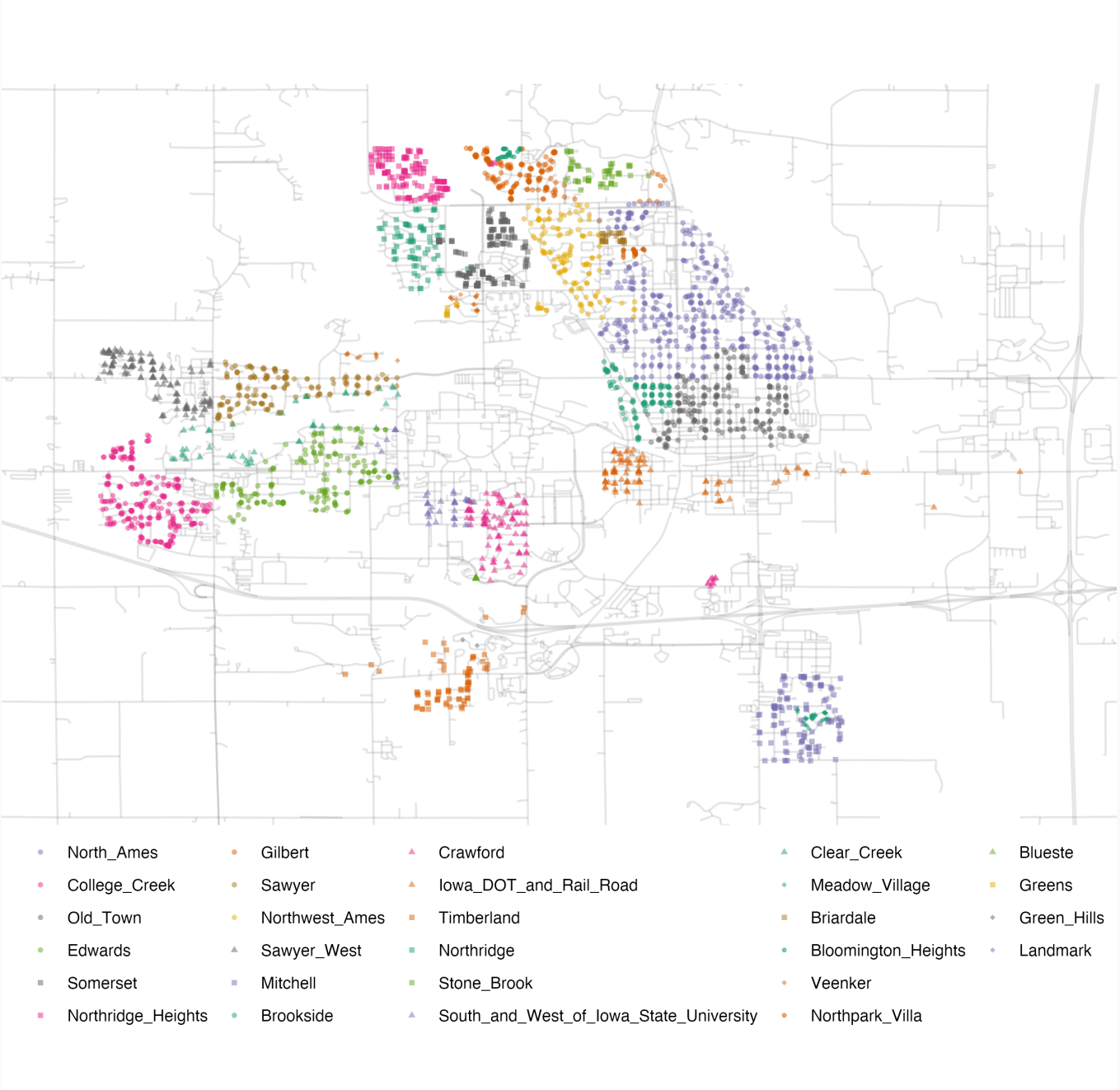
On peut explorer ces données rapidement pour s'en faire une idée. C'est une étape importante, qui ne doit pas être négligée.

Aperçu (1)

```
glimpse(ames)
```

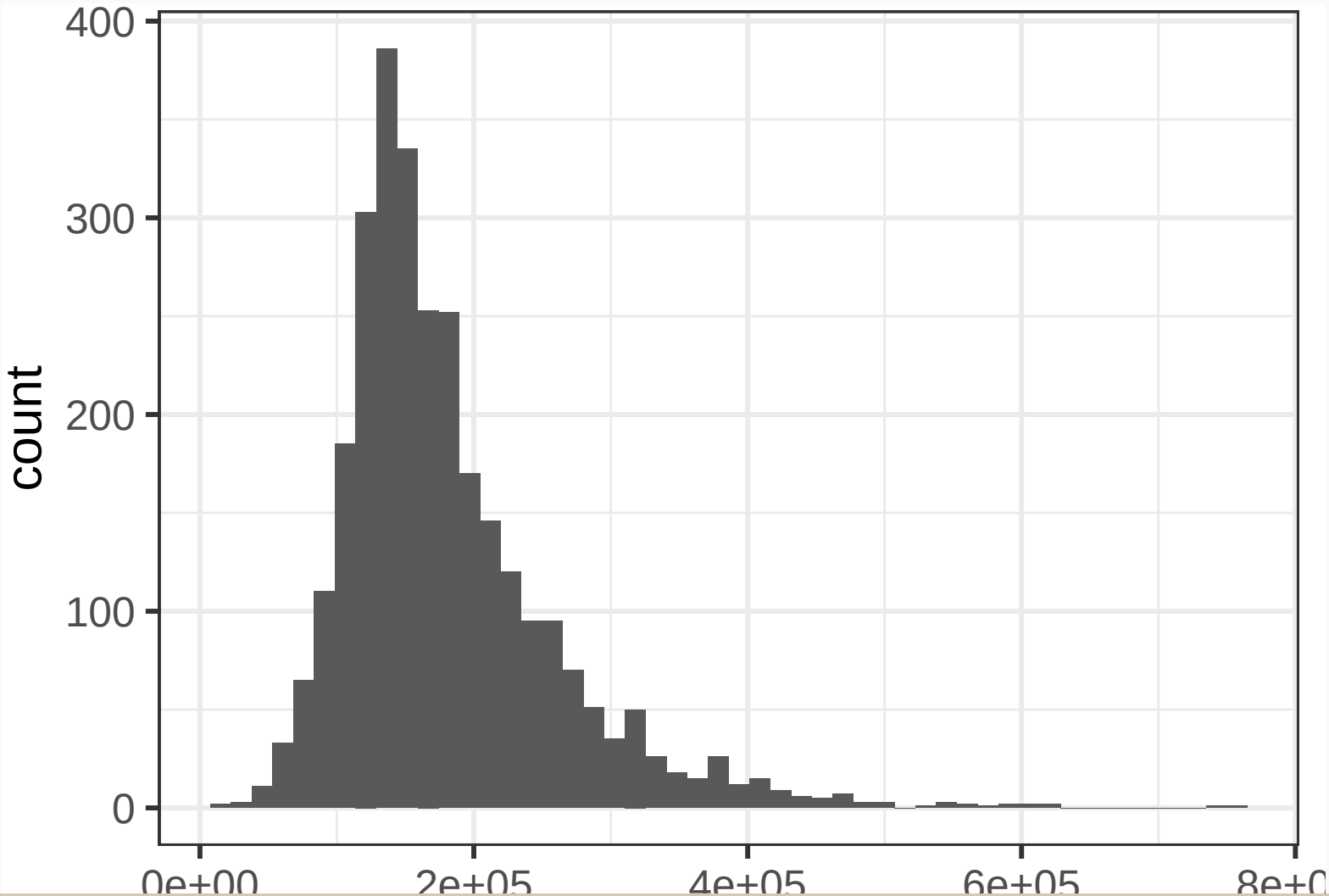
```
## Rows: 2,930
## Columns: 74
## $ MS_SubClass        <fct> One_Story_1946_and_Newer_All_Styles, One_Story_1946~
## $ MS_Zoning          <fct> Residential_Low_Density, Residential_High_Density, ~
## $ Lot_Frontage       <dbl> 141, 80, 81, 93, 74, 78, 41, 43, 39, 60, 75, 0, 63,~
## $ Lot_Area           <int> 31770, 11622, 14267, 11160, 13830, 9978, 4920, 5005~
## $ Street             <fct> Pave, Pave, Pave, Pave, Pave, Pave, Pave, Pave, Pav~
## $ Alley              <fct> No_Alley_Access, No_Alley_Access, No_Alley_Access, ~
## $ Lot_Shape          <fct> Slightly_Irregular, Regular, Slightly_Irregular, Re~
## $ Land_Contour       <fct> Lvl, Lvl, Lvl, Lvl, Lvl, Lvl, Lvl, HLS, Lvl, Lvl, L~
## $ Utilities          <fct> AllPub, AllPub, AllPub, AllPub, AllPub, AllPub, All~
## $ Lot_Config         <fct> Corner, Inside, Corner, Corner, Inside, Inside, Ins~
## $ Land_Slope         <fct> Gtl, Gtl, Gtl, Gtl, Gtl, Gtl, Gtl, Gtl, Gtl, Gtl, G~
## $ Neighborhood       <fct> North_Ames, North_Ames, North_Ames, North_Ames, Gil~
## $ Condition_1        <fct> Norm, Feedr, Norm, Norm, Norm, Norm, Norm, Norm, No~
## $ Condition_2        <fct> Norm, Norm, Norm, Norm, Norm, Norm, Norm, Norm, Nor~
## $ Bldg_Type          <fct> OneFam, OneFam, OneFam, OneFam, OneFam, OneFam, Twn~
## $ House_Style        <fct> One_Story, One_Story, One_Story, One_Story, Two_Sto~
## $ Overall_Cond       <fct> Average, Above_Average, Above_Average, Average, Ave~
## $ Year_Built         <int> 1960, 1961, 1958, 1968, 1997, 1998, 2001, 1992, 199~
```

Aperçu (2)


North_Ames
College_Creek
Old_Town
Edwards
Somerset
Northridge_Heights
Gilbert
Sawyer
Northwest_Ames
Sawyer_West
Mitchell
Brookside
Crawford
Iowa_DOT_and_Rail_Road
Timberland
Northridge
Stone_Brook
South_and_West_of_Iowa_State_University
Clear_Creek
Meadow_Village
Briardale
Bloomington_Heights
Veenker
Northpark_Villa
Blueste
Greens
Green_Hills
Landmark

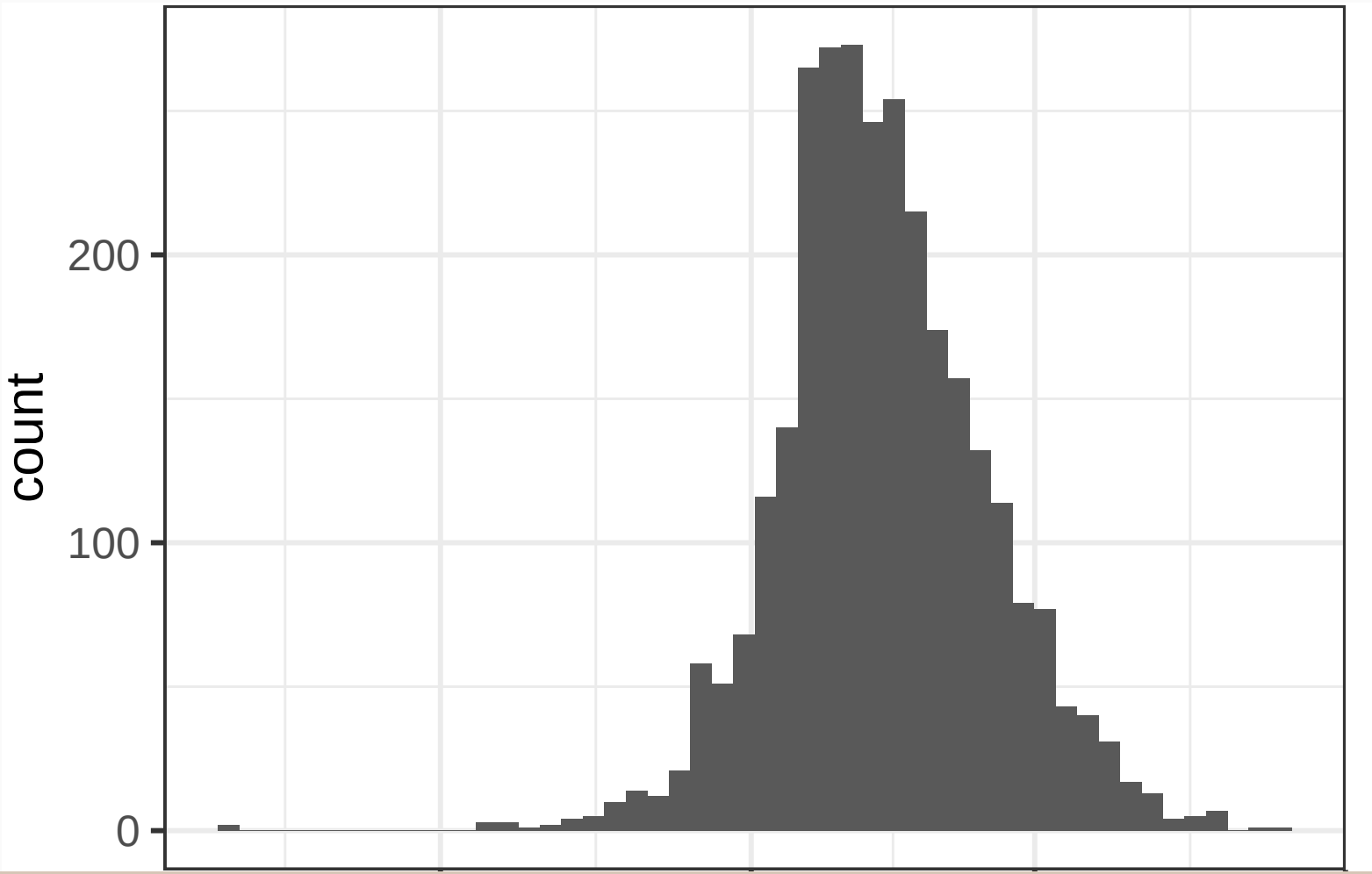
Visualisation du prix

```
ames %>%
  ggplot() +
  aes(x = Sale_Price) +
  geom_histogram(bins = 50)
```



Transformation logarithmique

```
ames %>%
  ggplot() +
  aes(x = Sale_Price) +
  geom_histogram(bins = 50) +
  scale_x_log10()
```



Pour aller plus loin

Il est également possible d'obtenir rapidement un rapport automatique en s'aidant de packages spécialisés.

```
DataExplorer::create_report(ames,
                            output_dir = "reports"
                            output_file = "ames")
```

Visualiser le rapport. Bien utile ?

Dans la suite

Pour la suite, nous travaillerons sur les prix transformés et sur une partie seulement des variables comme prédicteurs :

- `Neighborhood` : variable qualitative, avec 29 voisins ;
- `Gr_Liv_Area` : variable continue qui représente la surface habitable en pied au carré ;
- `Year_Built` : variable discrète qui représente l'année de construction.
- `Bldg_Type` : variable qualitative qui représente le type de bâtiment : OneFam (n=1814), TwoFmCon (n=45), Duplex (n=76), Twnhs (n=76) et TwnhsE (n=188).

```
ames <- Ames %>%
  mutate(Sale_Price = log10(Sale_Price)) %>%
  select(Sale_Price, Neighborhood, Gr_Liv_Area, Year_Built, Bldg_Type)
```

Version simple

```
set.seed(123)
# Save the split information for an 80/20 split of the data
ames_split <- initial_split(ames, prob = 0.80)
ames_split
```

```
## <Analysis/Assess/Total>
## <2198/732/2930>
```

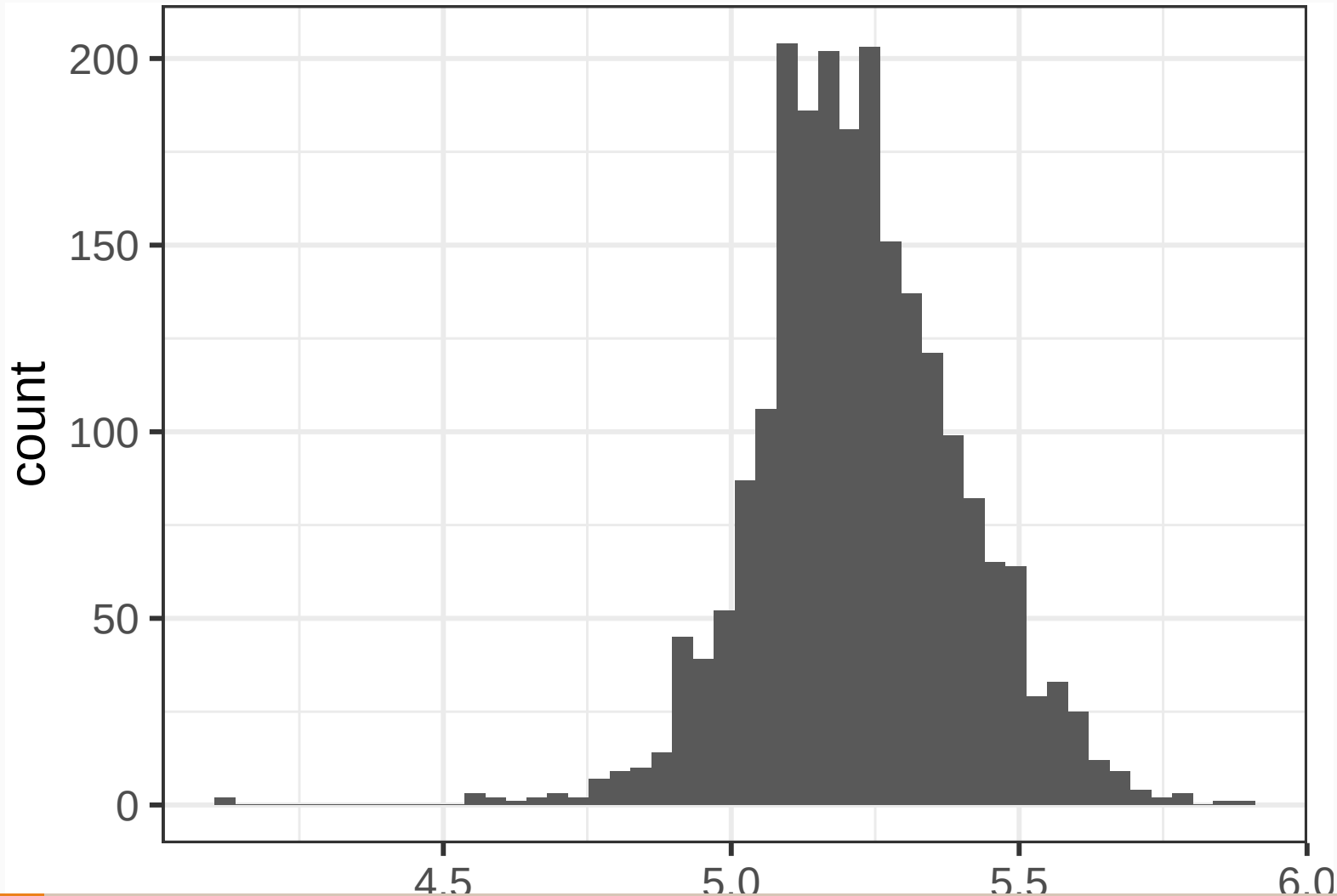
```
ames_train <- training(ames_split)
ames_test  <-  testing(ames_split)

dim(ames_train)
```

```
## [1] 2198    5
```

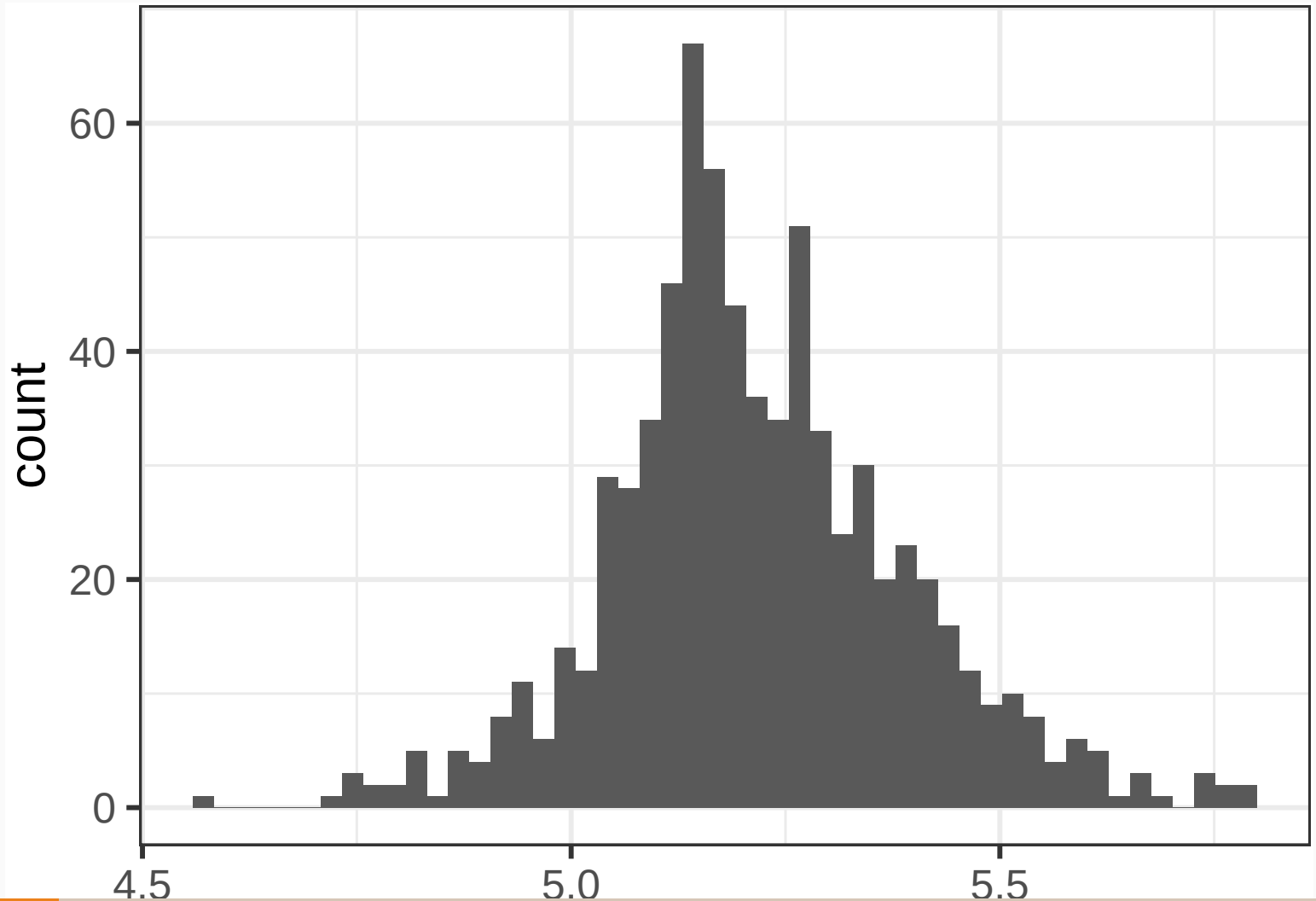
Visualisation (train)

```
ames_train %>%
  ggplot() +
  aes(x = Sale_Price) +
  geom_histogram(bins = 50)
```



Visualisation (test)

```
ames_test %>%
  ggplot() +
  aes(x = Sale_Price) +
  geom_histogram(bins = 50)
```



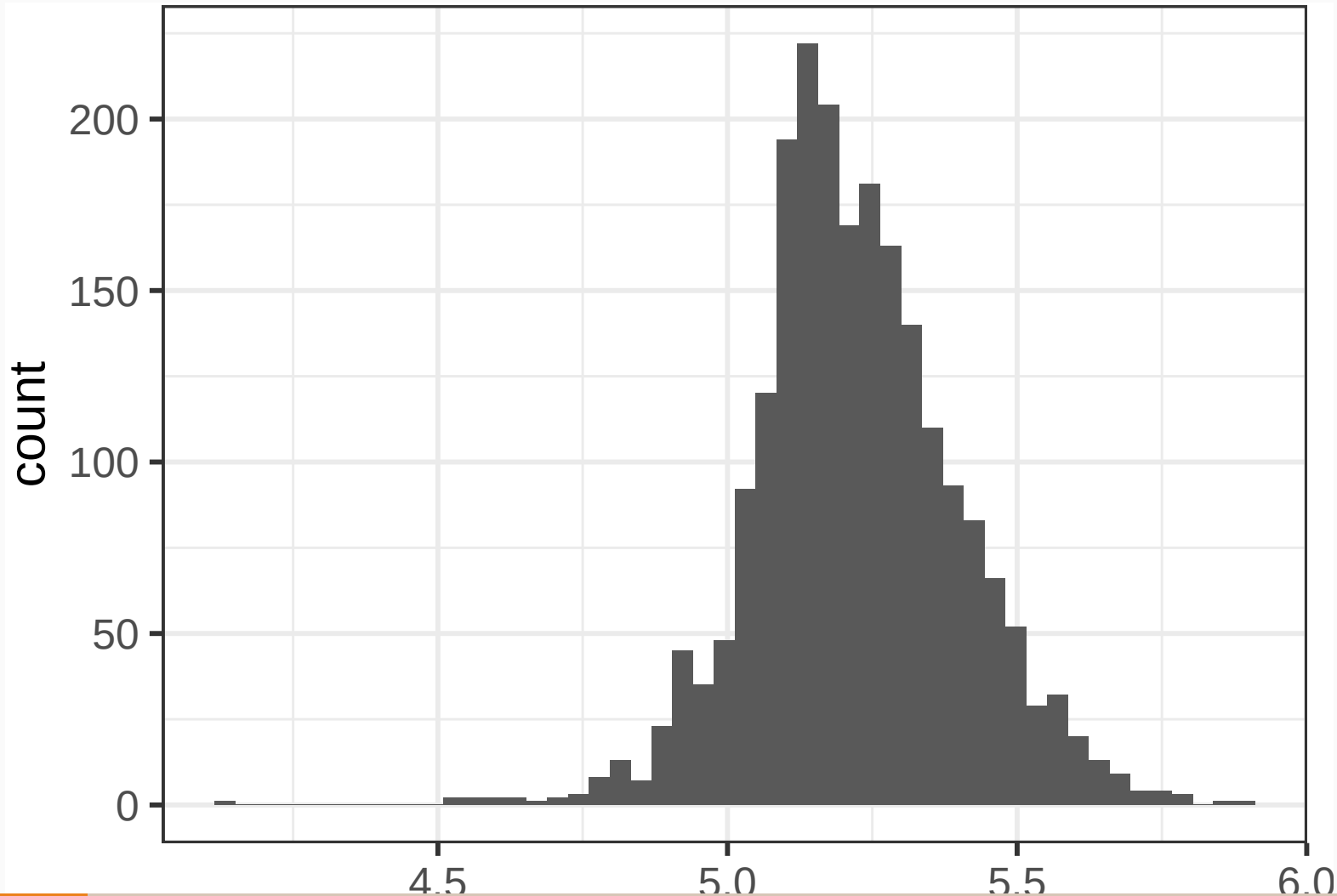
Échantillonnage stratifié

Pour corriger ce défaut, il est possible de stratifier l'échantillonnage par rapport à l'une des variables, ici la variable d'intérêt. Puisque cette variable est quantitative, les données sont séparées en quartiles et l'échantillonnage est effectué dans chacun des quartiles. Pour une variable qualitative l'échantillonnage est effectué dans chacune des modalités pour s'assurer d'une bonne distribution des modalités sous-représentées.

```
set.seed(123)
ames_split <- initial_split(ames, prob = 0.80, strata = Sale_Price)
ames_train <- training(ames_split)
ames_test  <-  testing(ames_split)
```

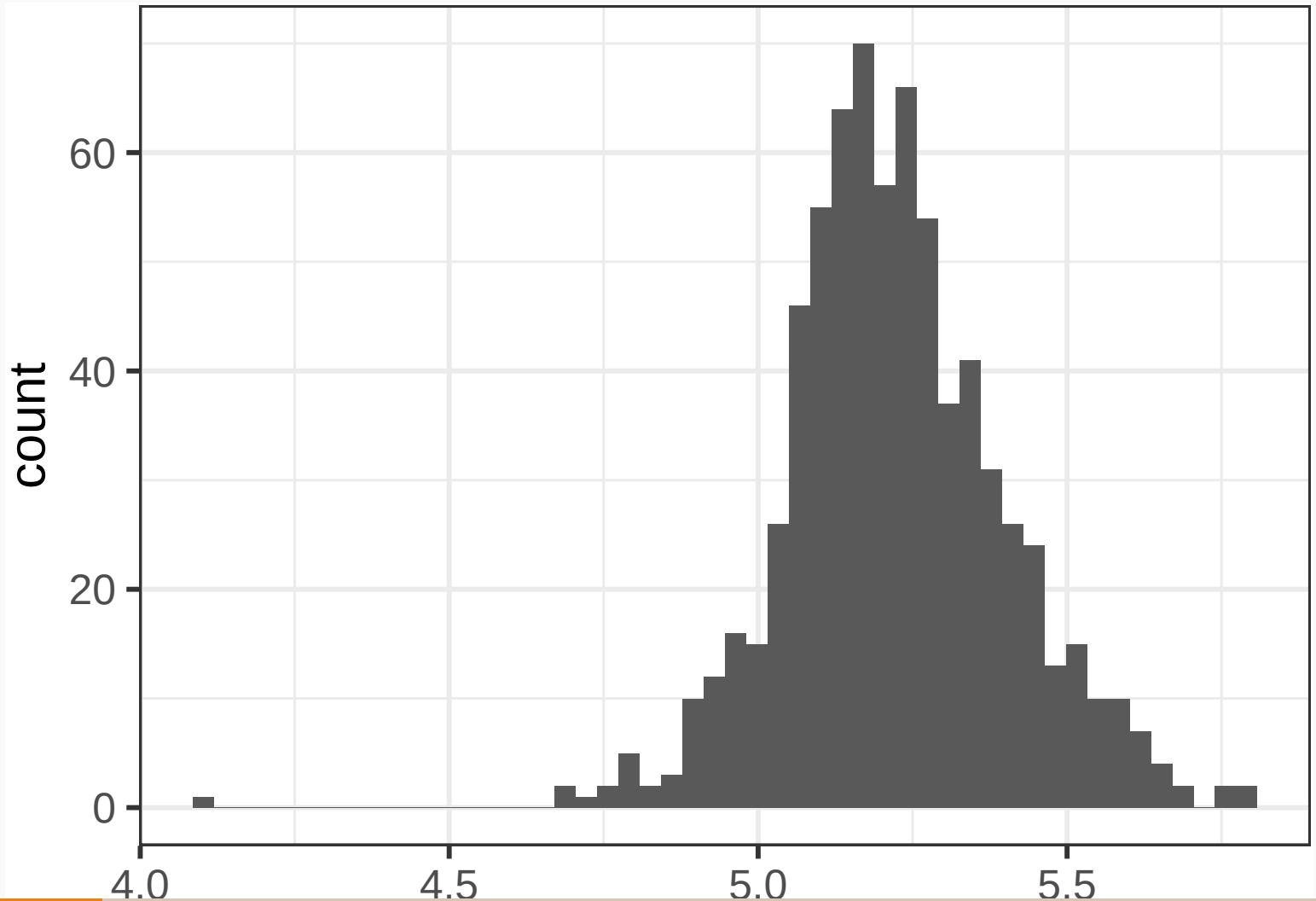
Visualisation (train)

```
ames_train %>%
  ggplot() +
  aes(x = Sale_Price) +
  geom_histogram(bins = 50)
```



Visualisation (test)

```
ames_test %>%
  ggplot() +
  aes(x = Sale_Price) +
  geom_histogram(bins = 50)
```



La méthode classique

```
lm(Sale_Price ~ Neighborhood +
     log10(Gr_Liv_Area) +
     Year_Built +
     Bldg_Type,
   data = ames_train)
```

```
##
## Call:
## lm(formula = Sale_Price ~ Neighborhood + log10(Gr_Liv_Area) +
##     Year_Built + Bldg_Type, data = ames_train)
##
## Coefficients:
##                                       (Intercept)
##                                         -0.827298
##                           NeighborhoodCollege_Creek
##                                          0.013164
##                               NeighborhoodOld_Town
##                                         -0.028325
##                                NeighborhoodEdwards
##                                         -0.048051
##                               NeighborhoodSomerset
##                                          0.050520
```

Que se passe-t-il ?

Pour construire la matrice de design, plusieurs manipulations des données sont effectuées. Nous pouvons les décomposer en plusieurs étapes :

1. La variable `Sale_Price` est définie comme variable d'intérêt tandis que les autres servent de variables prédictives ;
2. Une transformation continue (logarithmique) est appliquée à la variable continue `Gr_Liv_Area` ;
3. Les variables qualitatives `Bldg_Type` et `Neighborhood` qui ne peuvent pas être utilisées directement pour effectuer une régression linéaire sont transformées en plusieurs variables numériques (presque une par modalité).

Un mélange des genres

On comprend ici qu'on fait deux choses différentes simultanément dans le code ci-dessus :

1. On décrit le rôle des différentes variables et la façon dont on souhaite les utiliser à l'aide de la **formule** ;
2. On utilise une fonction de R qui va faire tourner une méthode statistique (ici la fonction `lm()` et la méthode des moindres carrés ordianaires).

Écrire une recette

```
simple_ames <- recipe(
    Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type,
    data = ames_train) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_dummy(all_nominal())
simple_ames
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          4
##
## Operations:
##
## Log transformation on Gr_Liv_Area
## Dummy variables from all_nominal()
```

Qu’est-ce ?

```
class(simple_ames)
```

```
## [1] "recipe"
```

Il s’agit d’un objet R qui définit une série d’étapes de préparation des données en vue du traitement statistique par différents modèles.

ATTENTION ! ces étapes ne sont pas effectuées. Il s’agit seulement de décrire ce qui devrait être fait.

En détail

1. La fonction `recipe()` sert à préciser les rôles des différentes variables (variable d'intérêt ou variable prédictive). De plus le fait de passer les données permet à la fonction de connaître la nature de chacune des variables (continue, catégorielle, etc.)

2. La première étape est d'appliquer une transformation logarithmique à la variable `Gr_Liv_Area` à l'aide de la fonction `step_log()`.

3. La dernière étape permet de sélectionner toutes les variables catégorielles (`all_nominal()`) et de les dummyfier.

Quel est l'avantage ?

1. Les recettes pourront servir à plusieurs modèles (compatibles mais on verra qu'il y en a de plus en plus). Elles ne sont pas couplées aux fonctions statistiques comme `lm()` ou `glmnet()`.

2. Les recettes sont plus souples que les formules et permettent plus de liberté dans la préparation des données.

3. Le syntaxe est compacte. `all_nominal()`, `all_numeric()`, `all_predictors()` et `all_outcomes()` qui peuvent être comparées à `start_with()` ou `contains()`de `{dplyr}`.

Utiliser la recette

Il faut commencer par **estimer** toutes les quantités requises pour les différentes étapes. Ceci se fait à l'aide de la fonction `prep()`.

```
simple_ames <- prep(simple_ames, training = ames_train)
simple_ames
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome           1
##  predictor           4
##
## Training data contained 2199 data points and no missing data.
##
## Operations:
##
## Log transformation on Gr_Liv_Area [trained]
## Dummy variables from Neighborhood, Bldg_Type [trained]
```

On peut mettre à cuire (1)

Enfin, on peut utiliser notre recette sur un jeu de données (l'échantillon de test par exemple) à l'aide de `bake()`.

```
test_ex <- bake(simple_ames, new_data = ames_test)
names(test_ex) %>% head()
```

```
## [1] "Gr_Liv_Area"                "Year_Built"
## [3] "Sale_Price"                 "Neighborhood_College_Creek"
## [5] "Neighborhood_Old_Town"      "Neighborhood_Edwards"
```

On peut mettre à cuire (2)

Ou sur l'échantillon d'apprentissage lui-même (remarquer le `NULL` pour éviter de refaire des calculs effectués lors de la préparation)

```
bake(simple_ames, new_data = NULL) %>% dim()
```

```
## [1] 2199   35
```

```
ames_train %>% dim()
```

```
## [1] 2199    5
```

```
simple_ames %>%
  bake(new_data = NULL, starts_with("Nei")) %>%
  dim()
```

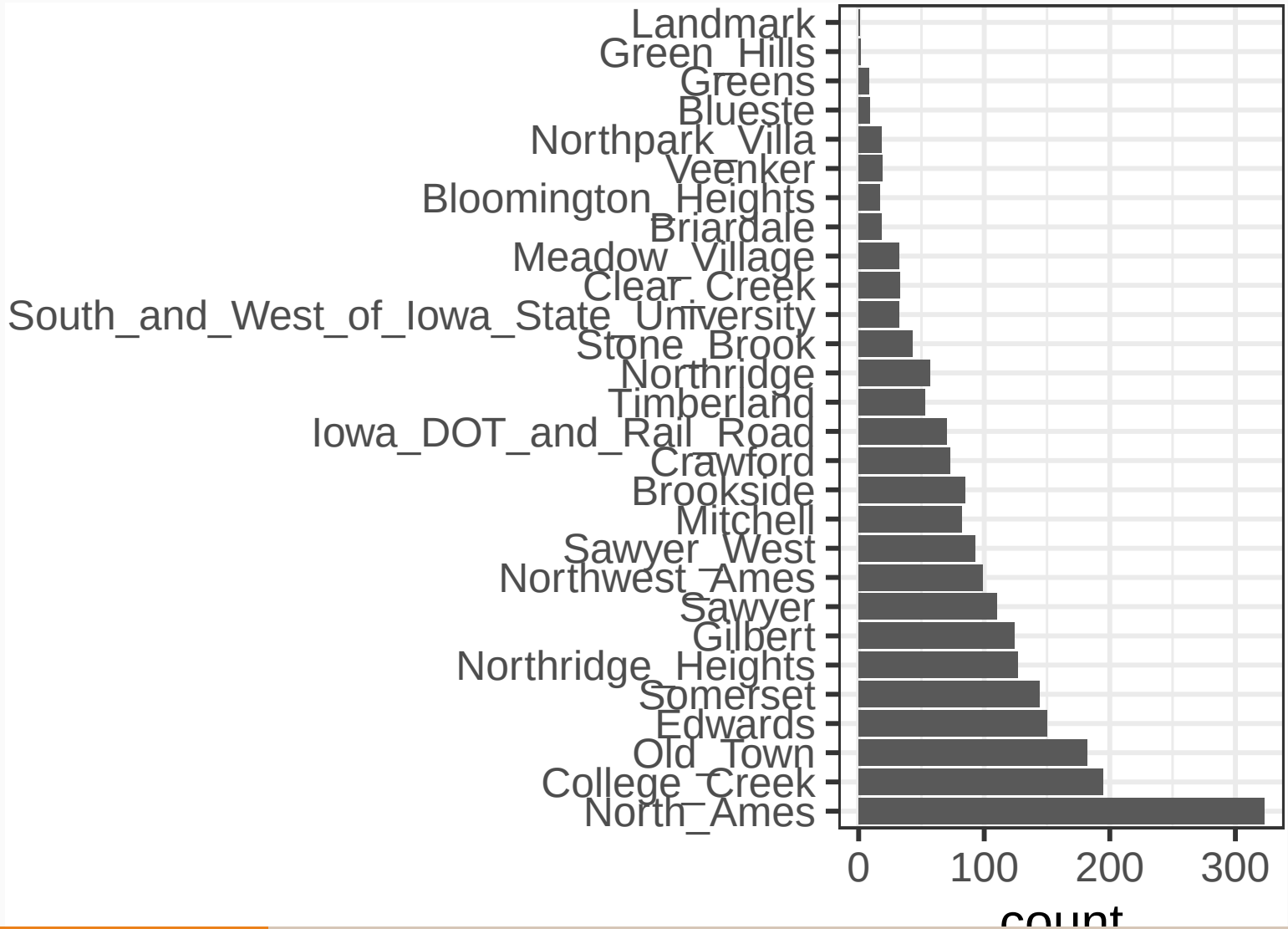
```
## [1] 2199   28
```

Étape d'encodage des données qualitatives

- `step_unknown()` permet de convertir les données manquantes vers un niveau spécifié d'une variable catégorielle.
- `step_novel()`autorise la présence d'un nouveau niveau dans les données au moment de la cuisson.
- `step_other()` permet d'analyser la fréquence des facteurs et de regrouper les facteurs les moins représentés dans un niveau fourre-tout (un seuil peut être précisé).

Exemple (1)

```
ggplot(ames_train, aes(y = Neighborhood)) +
  geom_bar() +
  labs(y = NULL)
```



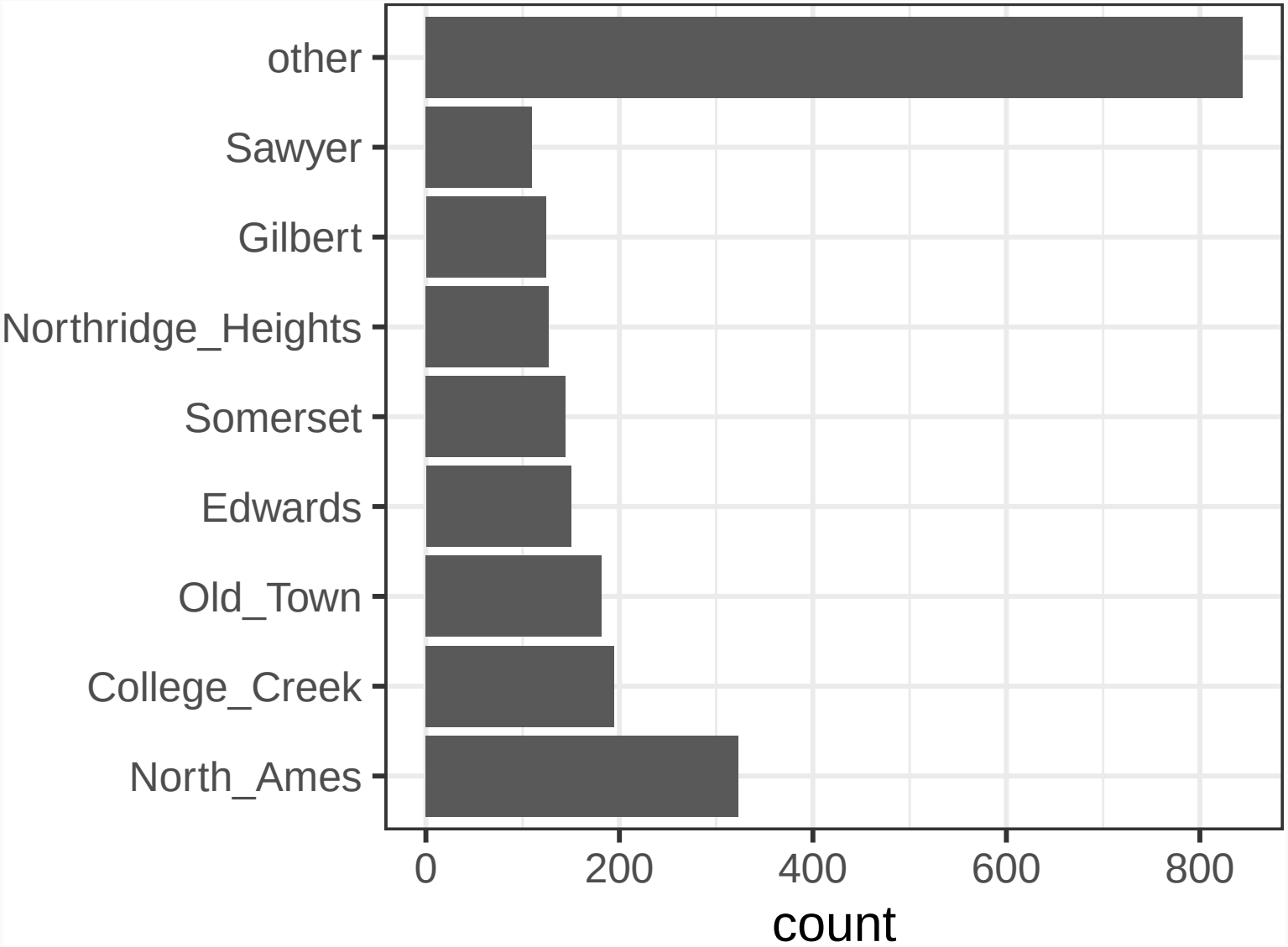
Exemple (2)

```
simple_ames <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type,
         data = ames_train) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_other(Neighborhood, threshold = 0.05)
```

```
gg <- simple_ames %>%
  prep() %>%
  bake(new_data = NULL) %>%
  ggplot(aes(y = Neighborhood)) +
    geom_bar() +
    labs(y = NULL)
```

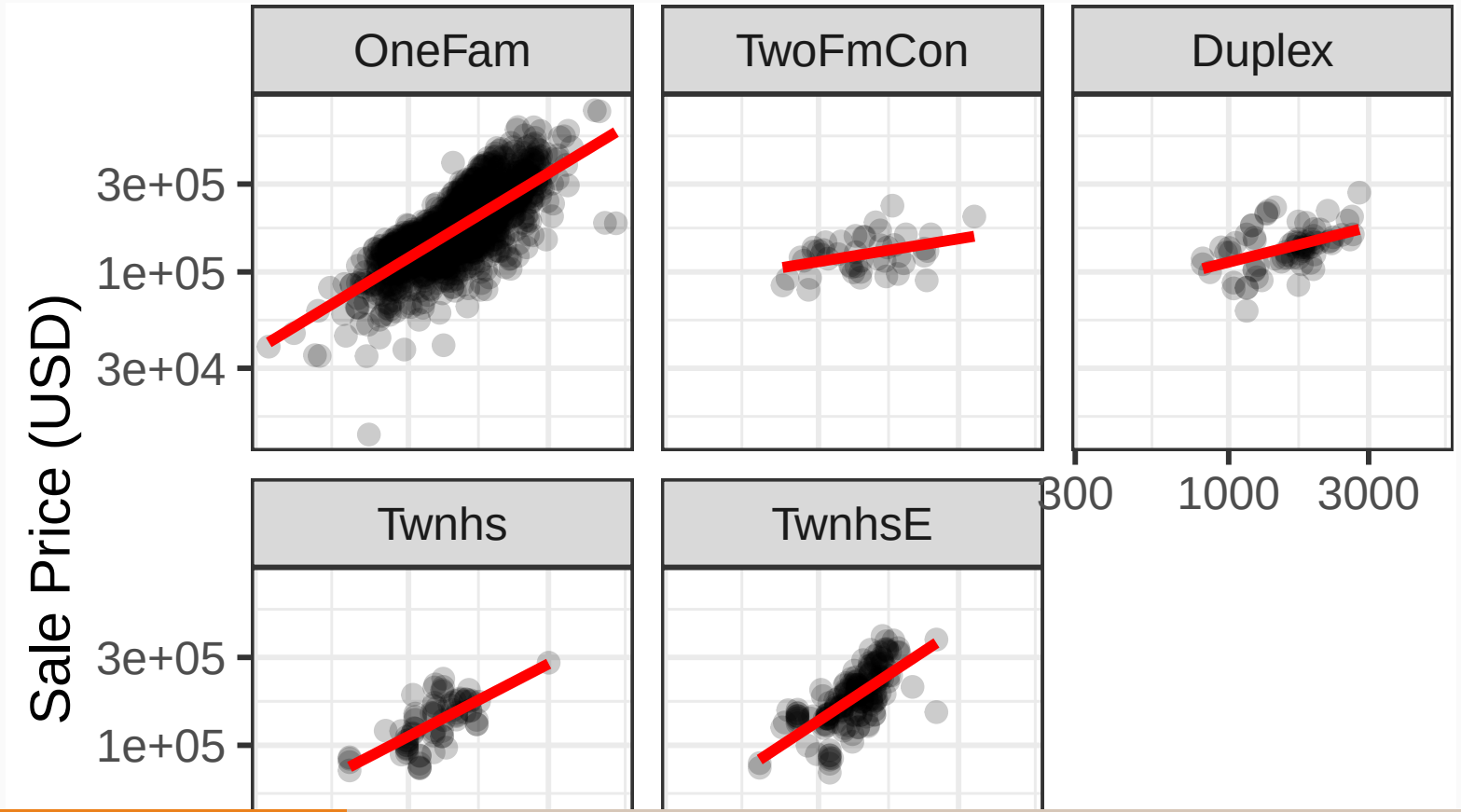
Exemple (3)

gg


other
Sawyer
Gilbert
Northridge_Heights
Somerset
Edwards
Old_Town
College_Creek
North_Ames
0
200
400
600
800
count

Prendre en compte les intéractions (1)

```
ggplot(ames_train, aes(x = Gr_Liv_Area, y = 10^Sale_Price)) +
  geom_point(alpha = .2) +
  facet_wrap(~ Bldg_Type) +
  geom_smooth(method = lm, formula = y ~ x, se = FALSE, col = "red") +
  scale_x_log10() +
  scale_y_log10() +
  labs(x = "General Living Area", y = "Sale Price (USD)")
```



Prendre en compte les intéractions (2)

```
simple_ames <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type,
         data = ames_train) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_other(Neighborhood, threshold = 0.01) %>%
  step_dummy(all_nominal()) %>%
  # Gr_Liv_Area is on the log scale from a previous step
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") )
```

Une remarque sur la remarque : contrairement à `Gr_Liv_Area`, la variable `Sale_Price` a été transformée à l'aide d'un `mutate()` en dehors de la recette. C'est la méthode recommandée.

Prendre en compte les intéractions (3)

Attention ! le comportement est différent de celui des formules usuelles de R.

```
step_interact(~ Gr_Liv_Area:Bldg_Type)
```

ne fonctionnerait pas car on ne peut créer une interaction qu'entre variables quantitatives. On doit donc :

- avoir créé les variables factices binaires ;
- compris la convention de nommage des variables ainsi crées.

D'ailleurs :

```
simple_ames %>%
  prep() %>%
  bake(new_data = NULL) %>%
  select(contains("_x_")) %>%
  names()
```

```
## [1] "Gr_Liv_Area_x_Bldg_Type_TwoFmCon" "Gr_Liv_Area_x_Bldg_Type_Duplex"
## [3] "Gr_Liv_Area_x_Bldg_Type_Twnhs"    "Gr_Liv_Area_x_Bldg_Type_TwnhsE"
```

Gérer les données non équilibrées

Attention, il y a un nouveau package **{themis}** qui contient des étapes de recettes dédiées au données non équilibrées. À utiliser sur l'échantillon d'entraînement.

```
recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type,
         data = ames_train) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_other(Neighborhood, threshold = 0.01) %>%
  themis::step_downsample(Neighborhood) %>%
  prep() %>%
  bake(new_data = NULL) %>%
  ggplot(aes(y = Neighborhood)) +
    geom_bar() +
    labs(y = NULL)
```

```
## Registered S3 methods overwritten by 'themis':
##   method                  from
##   bake.step_downsample    recipes
##   bake.step_upsample      recipes
##   prep.step_downsample    recipes
##   prep.step_upsample      recipes
```

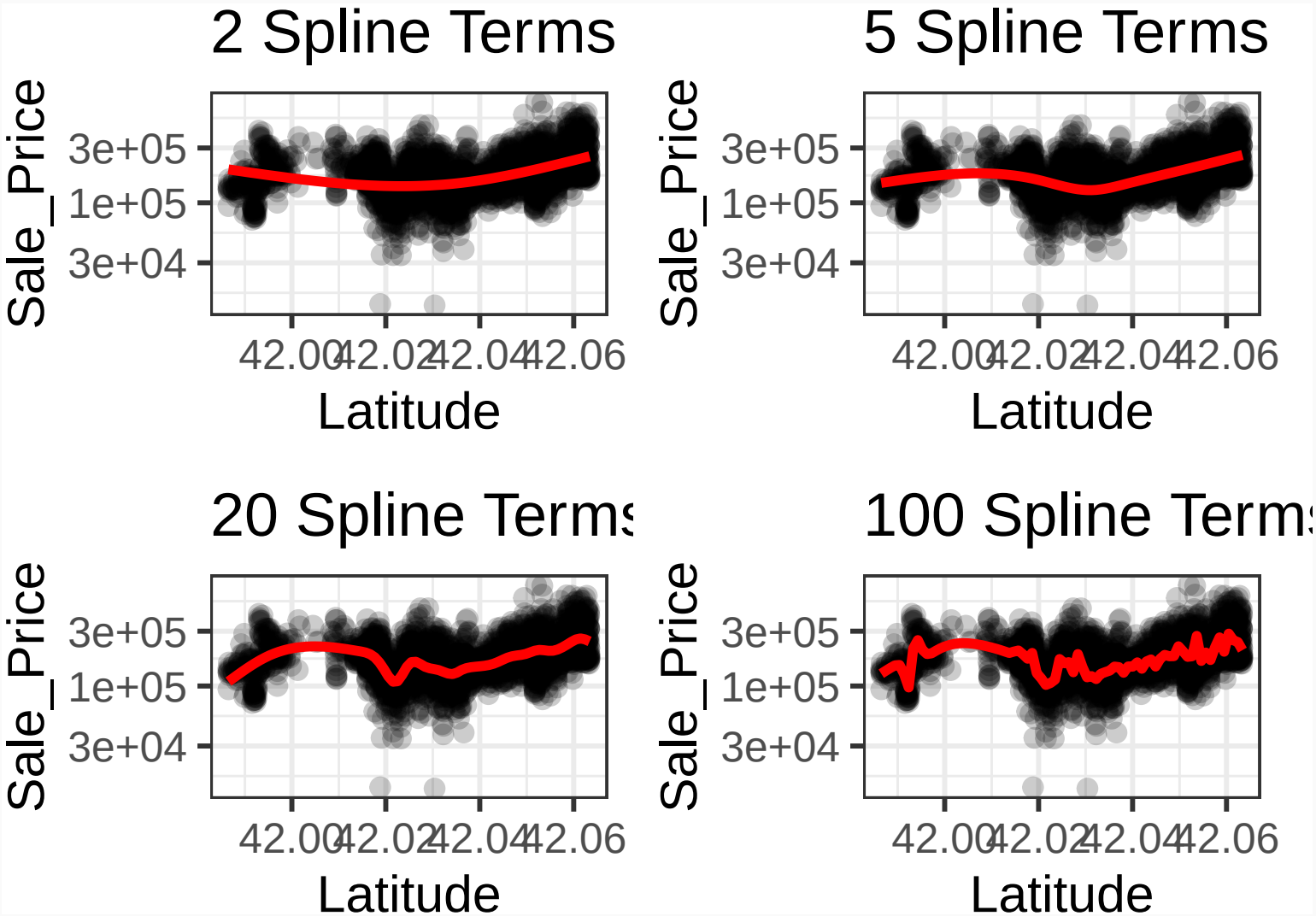
Utilisation des splines naturelles (1)

```
library(patchwork)
library(splines)

plot_smoother <- function(deg_free) {
  ggplot(Ames, aes(x = Latitude, y = Sale_Price)) +
    geom_point(alpha = .2) +
    scale_y_log10() +
    geom_smooth(
      method = lm,
      formula = y ~ ns(x, df = deg_free),
      col = "red",
      se = FALSE
    ) +
    ggtitle(paste(deg_free, "Spline Terms"))
}
```

Utilisation des splines naturelles (2)

```
( plot_smoother(2) + plot_smoother(5) ) / ( plot_smoother(20) + plot_smoother(100)
```



Utilisation des splines naturelles (3)

```
recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type + Latitude,
         data = Ames) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_other(Neighborhood, threshold = 0.01) %>%
  step_dummy(all_nominal()) %>%
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") ) %>%
  step_ns(Latitude, deg_free = 20) %>%
  ## On devrait arrêter ICI. C'est juste pour visualiser ;-)
  prep() %>%
  bake(new_data = NULL) %>%
  select(contains("_ns")) %>%
  names()
```

```
##  [1] "Latitude_ns_01" "Latitude_ns_02" "Latitude_ns_03" "Latitude_ns_04"
##  [5] "Latitude_ns_05" "Latitude_ns_06" "Latitude_ns_07" "Latitude_ns_08"
##  [9] "Latitude_ns_09" "Latitude_ns_10" "Latitude_ns_11" "Latitude_ns_12"
## [13] "Latitude_ns_13" "Latitude_ns_14" "Latitude_ns_15" "Latitude_ns_16"
## [17] "Latitude_ns_17" "Latitude_ns_18" "Latitude_ns_19" "Latitude_ns_20"
```

Bilan des recettes (1)

```
ames <- mutate(Ames, Sale_Price = log10(Sale_Price))

set.seed(123)
ames_split <- initial_split(ames, prob = 0.80, strata = Sale_Price)
ames_train <- training(ames_split)
ames_test  <-  testing(ames_split)

ames_rec <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type +
           Latitude + Longitude, data = ames_train) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_other(Neighborhood, threshold = 0.01) %>%
  step_dummy(all_nominal()) %>%
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") ) %>%
  step_ns(Latitude, Longitude, deg_free = 20)
```

Bilan des recettes (modèle usuel de R)

```
ames_rec_prepped <- prep(ames_rec)
ames_train_prepped <- bake(ames_rec_prepped, new_data = ames_train)
ames_test_prepped <- bake(ames_rec_prepped, ames_test)
# Fit the model; Note that the column Sale_Price has already been
# log transformed.
lm_fit <- lm(Sale_Price ~ ., data = ames_train_prepped)
```

Si la fonction `prep()` est utilise sans argument elle fait la préparation en utilisant les données qui ont été mentionnées dans l'agument `data = .` de la recette.

Bilan des recettes (coup d'oeil)

La fonction `glance()` est disponible dans le package `{broom}`dont on reparlera plus loin.

```
glance(lm_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     0.822         0.816 0.0757      140.       0    70  2592. -5040. -
4630.
## # ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

Bilan des recettes (estimation des paramètres du modèle)

Les coefficients du modèle peuvent être extraits à l'aide de la fonction `tidy()` du même package :

```
tidy(lm_fit)
```

```
## # A tibble: 71 x 5
##    term                          estimate std.error statistic   p.value
##    <chr>                            <dbl>     <dbl>     <dbl>     <dbl>
##  1 (Intercept)                    -0.531     0.296      -1.79   7.32e-  2
##  2 Gr_Liv_Area                     0.648     0.0161     40.3    1.46e-
264
##  3 Year_Built                      0.00194   0.000139   13.9    2.69e- 42
##  4 Neighborhood_College_Creek     -0.0898    0.0332     -2.71   6.83e-  3
##  5 Neighborhood_Old_Town          -0.0516    0.0129     -4.01   6.20e-  5
##  6 Neighborhood_Edwards           -0.153     0.0274     -5.57   2.91e-  8
##  7 Neighborhood_Somerset           0.0364    0.0189      1.92   5.50e-  2
##  8 Neighborhood_Northridge_Heights  0.0984    0.0272      3.61   3.10e-  4
##  9 Neighborhood_Gilbert            0.000728  0.0219      0.0333 9.73e-  1
## 10 Neighborhood_Sawyer            -0.156     0.0263     -5.93   3.59e-  9
## # ... with 61 more rows
```

Bilan des recettes (prédiction)

```
predict(lm_fit, ames_test_prepped %>% head(10))
```

```
##        1        2        3        4        5        6        7        8
## 5.307819 5.300849 5.167501 5.518595 5.087885 5.488895 5.510087 5.430288
##        9       10
## 5.553061 5.239981
```

Sur une recette directement

On peut **tifyfier** une recette pour voir à quoi elle ressemble... et beacoup d'autres objets R !

```
tidy(ames_rec_prepped)
```

```
## # A tibble: 5 x 6
##   number operation type     trained skip  id
##    <int> <chr>     <chr>    <lgl>   <lgl> <chr>
## 1      1 step      log      TRUE    FALSE log_u4pZz
## 2      2 step      other    TRUE    FALSE other_XqiIK
## 3      3 step      dummy    TRUE    FALSE dummy_ht3jv
## 4      4 step      interact TRUE    FALSE interact_K2XxD
## 5      5 step      ns       TRUE    FALSE ns_vqtIn
```

{parsnip}

```
library(rstanarm)
```

```
## Loading required package: Rcpp
```

```
##
## Attaching package: 'Rcpp'
```

```
## The following object is masked from 'package:rsample':
##
##     populate
```

```
## This is rstanarm version 2.21.1
```

```
## - See https://mc-stan.org/rstanarm/articles/priors for changes to default priors
```

```
## - Default priors may change, so it's safest to specify priors, even if equivalen
```

```
## - For execution on a local, multicore CPU with excess RAM we recommend calling
```

```
##   options(mc.cores = parallel::detectCores())
```

Création d'un modèle (les bases)

Pour créer un modèle avec `{tidymodels}` il faut :

- le modèle mathématique qu'on souhaite utiliser (régression linéaire, forêt aléatoire, plus proches voisins, etc.);
- indiquer quel **moteur** on souhaite utiliser (le plus souvent le package);
- préciser si nécessaire le mode de prédiction : regression, classification ;

Principe

Le package `{parsnip}` est en quelque sorte une couche d'abstraction qui permet de ne pas dépendre de la façon dont les différents package ont été conçus. La fonction `translate()` permet de visualiser comment le code saisi par l'utilisateur est converti dans la syntaxe requise par le package de traitement statistique.

Exemple (1)

```
linear_reg() %>% set_engine("lm") %>% translate()
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
##
## Model fit template:
## stats::lm(formula = missing_arg(), data = missing_arg(), weights = missing_arg()
```

Exemple (2)

```
linear_reg() %>% set_engine("glmnet") %>% translate()
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: glmnet
##
## Model fit template:
## glmnet::glmnet(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
##     family = "gaussian")
```

Exemple (3)

```
linear_reg() %>% set_engine("stan") %>% translate()
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: stan
##
## Model fit template:
## rstanarm::stan_glm(formula = missing_arg(), data = missing_arg(),
##     weights = missing_arg(), family = stats::gaussian, refresh = 0)
```

Utilisation sur des données (1)

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

lm_form_fit <-
  lm_model %>%
  fit(
    Sale_Price ~ Longitude + Latitude,
    data = ames_train
    )

lm_xy_fit <-
  lm_model %>%
  fit_xy(
    x = ames_train %>% select(Longitude, Latitude),
    y = ames_train %>% pull(Sale_Price)
    )
```

Utilisation sur des données (2)

```
lm_form_fit
```

```
## parsnip model object
##
## Fit time:  2ms
##
## Call:
## stats::lm(formula = Sale_Price ~ Longitude + Latitude, data = data)
##
## Coefficients:
## (Intercept)    Longitude     Latitude
##    -316.368       -2.083        3.010
```

Utilisation sur des données (3)

```
lm_xy_fit
```

```
## parsnip model object
##
## Fit time:  1ms
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
## (Intercept)    Longitude     Latitude
##    -316.368       -2.083        3.010
```

Utilisation sur des données (4)

Différence principale : avec `fit()` des variables factices sont potentiellement crées. Pas avec `fit_xy()` qui utilise les données telles qu'elles sont.

Un exemple pourles forêts aléatoires (1)

```
rand_forest(trees = 1000, min_n = 5) %>%
  set_engine("ranger") %>%
  set_mode("regression") %>%
  translate()
```

```
## Random Forest Model Specification (regression)
##
## Main Arguments:
##   trees = 1000
##   min_n = 5
##
## Computational engine: ranger
##
## Model fit template:
## ranger::ranger(x = missing_arg(), y = missing_arg(), case.weights = missing_arg(
##     num.trees = 1000, min.node.size = min_rows(~5, x), num.threads = 1,
##     verbose = FALSE, seed = sample.int(10^5, 1))
```

Un exemple pourles forêts aléatoires (2)

On remarque que le mode de est spécifié car les forêts aléatoires peuvent être utiliées aussi bien pour prédire des variables catégorielles que continues.

Les fonctions de modélisation (comme `rand_forest()`, possèdent deux catégories d'arguments :

- les arguments principaux sont les plus utilisés et existent dans les différents packages (engines) éventuellement sous des noms différents ;
- Les arguments spécifiques aux différents moteurs, ou plus rarement utilisés.

Cueillette (1)

De nombreuses quantités sont stockées dans les modèles de {parsnip}.

```
lm_xy_fit %>% pluck("fit") # lm_form_fit$fit
```

```
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
## (Intercept)    Longitude     Latitude
##    -316.368       -2.083        3.010
```

Cueillette (2)

```
tidy(lm_form_fit)
```

```
## # A tibble: 3 x 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)  -316.      14.9       -21.2 3.04e-91
## 2 Longitude      -2.08     0.133     -15.6 3.21e-52
## 3 Latitude        3.01     0.185      16.3 2.58e-56
```

Principes

Pour faire des prédictions, `{parsnip}` suit tout le temps ces quelques règles :

- Les résultats sont des tiblles, toujours ;
- Les noms des variables sont **prévisibles** ;
- Il y a toujours le même nombre de lignes que les données d'entrée.

Exemple (1)

```
ames_test_small <- ames_test %>% slice(1:5)
predict(lm_form_fit, new_data = ames_test_small)
```

```
## # A tibble: 5 x 1
##   .pred
##   <dbl>
## 1  5.22
## 2  5.29
## 3  5.28
## 4  5.26
## 5  5.24
```

Exemple (2)

```
ames_test_small %>%
  select(Sale_Price) %>%
  bind_cols(predict(lm_form_fit, ames_test_small)) %>%
  # Add 95% prediction intervals to the results:
  bind_cols(predict(lm_form_fit, ames_test_small, type = "pred_int"))
```

```
## # A tibble: 5 x 4
##   Sale_Price .pred .pred_lower .pred_upper
##        <dbl> <dbl>       <dbl>       <dbl>
## 1       5.39  5.22        4.90        5.53
## 2       5.28  5.29        4.97        5.60
## 3       5.27  5.28        4.96        5.59
## 4       5.60  5.26        4.95        5.58
## 5       5.02  5.24        4.93        5.55
```

Exemple (3)

On a toujours le même nom de variable pour la prévision ! Même en changeant de méthode statistique et de package.

```
tree_model <-
  decision_tree(min_n = 2) %>%
  set_engine("rpart") %>%
  set_mode("regression")

tree_fit <-
  tree_model %>%
  fit(Sale_Price ~ Longitude + Latitude, data = ames_train)

ames_test_small %>%
  select(Sale_Price) %>%
  bind_cols(predict(tree_fit, ames_test_small))
```

```
## # A tibble: 5 x 2
##   Sale_Price .pred
##        <dbl> <dbl>
## 1       5.39  5.16
## 2       5.28  5.31
## 3       5.27  5.31
```

Convention de nommage des variables

type value	column name(s)
numeric	.pred
class	.pred_class
prob	.pred_{class levels}
conf_int	.pred_lower, .pred_upper
pred_int	.pred_lower, .pred_upper

{Workflows}

Worflows = pipeline

Attention ! La terminologie pour ce qui est appelé Workflows dans `{tidymodels}` est appelé `pipeline` en Python ou Spark. Mais ona déjà `%>%` qui génère un pipeline.

Un workflow simpliste (1)

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

```
lm_wflow <-
  workflow() %>%
  add_model(lm_model)
```

Un workflow simpliste (2)

```
lm_wflow
```

```
## == Workflow ============================================================
## Preprocessor: None
## Model: linear_reg()
##
## -- Model -----------------------------------------------------------------
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

On ajoute une formule

```
lm_wflow <-
  lm_wflow %>%
  add_formula(Sale_Price ~ Longitude + Latitude)

lm_wflow
```

```
## == Workflow ============================================================
## Preprocessor: Formula
## Model: linear_reg()
##
```

Fit

```
lm_fit <- fit(lm_wflow, ames_train)
lm_fit
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Formula
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## Sale_Price ~ Longitude + Latitude
##
## -- Model -----------------------------------------------------------------------
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
## (Intercept)    Longitude     Latitude
##    -316.368       -2.083        3.010
```

Predict

```
predict(lm_fit, ames_test %>% head(5))
```

```
## # A tibble: 5 x 1
##   .pred
##   <dbl>
## 1  5.22
## 2  5.29
## 3  5.28
## 4  5.26
## 5  5.24
```

Un peu de changement (1)

```
lm_wflow %>% update_formula(Sale_Price ~ Longitude)
```

```
## == Workflow ============================================================
## Preprocessor: Formula
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------------------------
## Sale_Price ~ Longitude
##
## -- Model -----------------------------------------------------------------
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

Un peu de changement (2)

```
lm_wflow %>% remove_formula()
```

```
## == Workflow ============================================================
## Preprocessor: None
## Model: linear_reg()
##
## -- Model -----------------------------------------------------------------
```

Mais... et ma recette ? (1)

```
ames_rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          6
##
## Operations:
##
## Log transformation on Gr_Liv_Area
## Collapsing factor levels for Neighborhood
## Dummy variables from all_nominal()
## Interactions with Gr_Liv_Area:starts_with("Bldg_Type_")
## Natural Splines on Latitude, Longitude
```

Mais... et ma recette ? (2)

```
lm_wflow <-
  lm_wflow %>%
  remove_formula() %>%
  add_recipe(ames_rec)
```

Mais... et ma recette ? (3)

```
lm_wflow
```

```
## == Workflow ============================================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------------------------
## 5 Recipe Steps
##
## * step_log()
## * step_other()
## * step_dummy()
## * step_interact()
## * step_ns()
##
## -- Model -----------------------------------------------------------------
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

La puissance du workflow (1)

La puissance du workflow (2)

De même `predict()` automatise la cuisson ! Pas de `bake()`...

```
predict(lm_fit, ames_test %>% head(5))
```

```
## # A tibble: 5 x 1
##   .pred
##   <dbl>
## 1  5.31
## 2  5.30
## 3  5.17
## 4  5.52
## 5  5.09
```

BILAN (1)

```
ames <- mutate(Ames, Sale_Price = log10(Sale_Price))

set.seed(1234)
ames_split <- initial_split(ames, prob = 0.80, strata = Sale_Price)
ames_train <- training(ames_split)
ames_test  <-  testing(ames_split)
ames_test_small <- ames_test %>% head(5)

ames_rec <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type +
           Latitude + Longitude, data = ames_train) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_other(Neighborhood, threshold = 0.01) %>%
  step_dummy(all_nominal()) %>%
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") ) %>%
  step_ns(Latitude, Longitude, deg_free = 20)
```

BILAN (2)

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

lm_wflow <-
  workflow() %>%
  add_model(lm_model) %>%
  add_recipe(ames_rec)

lm_fit <- fit(lm_wflow, ames_train)

lm_pred <- ames_test_small %>%
  select(Sale_Price) %>%
  bind_cols(predict(lm_fit, ames_test_small)) %>%
  bind_cols(predict(lm_form_fit, ames_test_small, type = "pred_int"))
```

BILAN (3)

```
lm_pred
```

```
## # A tibble: 5 x 4
##   Sale_Price .pred .pred_lower .pred_upper
##        <dbl> <dbl>       <dbl>       <dbl>
## 1       5.24  5.16        4.91        5.54
## 2       5.33  5.38        4.95        5.57
## 3       4.98  5.06        4.93        5.55
## 4       5.31  5.35        4.99        5.62
## 5       5.30  5.24        4.99        5.62
```

{yardstick}

Critères d'évaluation empirique des modèles

L'idée est de valider la qualité d'un modèle sur la partie de l'échantillon qui n'a pas servie à estimer les paramètres du modèle. On dispose de plusiers **métriques** pour décider la qualité d'un modèle :

- RMSE ;
- R-carré ;
- etc.

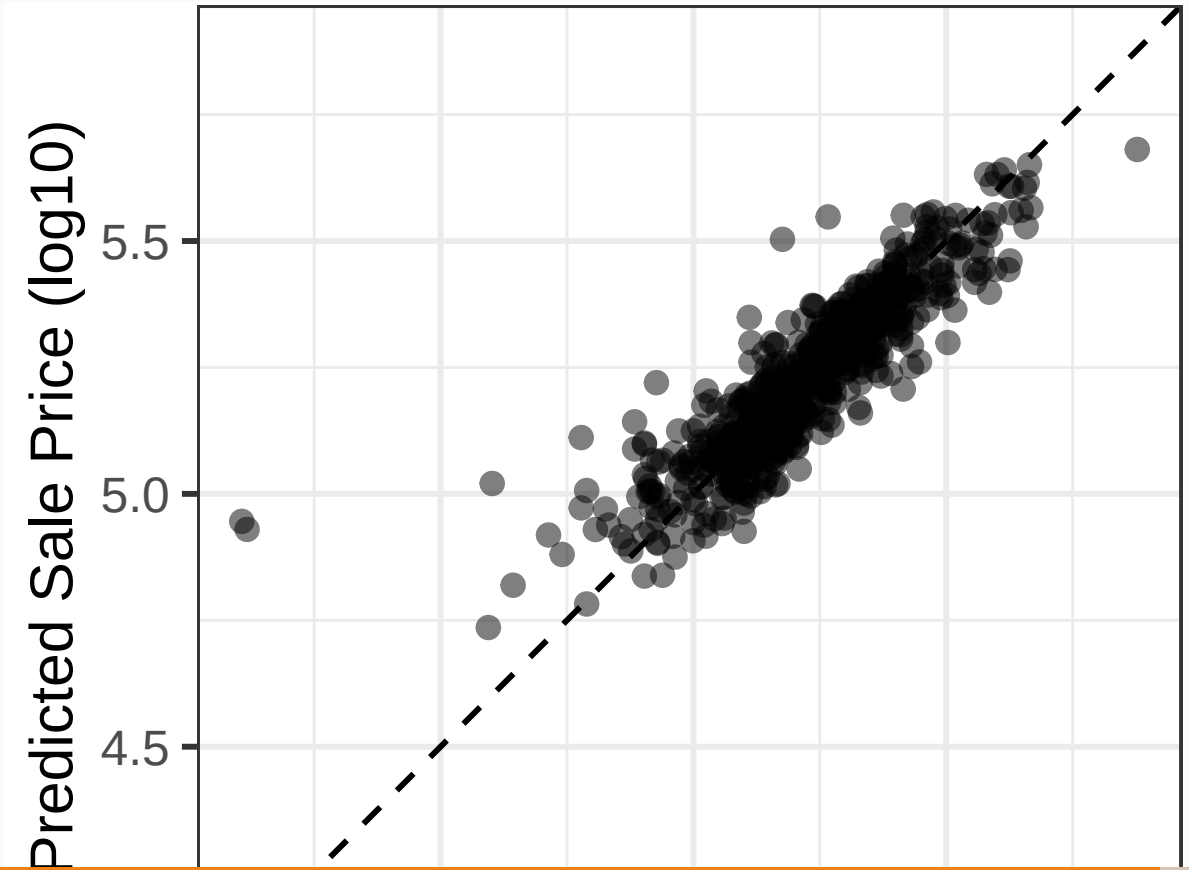
Métriques pour régression (1)

```
ames_test_res <-
  predict(lm_fit, new_data = ames_test %>% select(-Sale_Price)) %>%
  bind_cols(ames_test %>% select(Sale_Price))
ames_test_res
```

```
## # A tibble: 731 x 2
##    .pred Sale_Price
##    <dbl>      <dbl>
##  1  5.16       5.24
##  2  5.38       5.33
##  3  5.06       4.98
##  4  5.35       5.31
##  5  5.24       5.30
##  6  5.26       5.26
##  7  5.42       5.34
##  8  5.28       5.26
##  9  5.01       5.10
## 10  5.17       5.05
## # ... with 721 more rows
```

Métriques pour régression (2)

```r
ggplot(ames_test_res, aes(x = Sale_Price, y = .pred)) +
  # Create a diagonal line:
  geom_abline(lty = 2) +
  geom_point(alpha = 0.5) +
  labs(y = "Predicted Sale Price (log10)", x = "Sale Price (log10)") +
  # Scale and size the x- and y-axis uniformly:
  coord_obs_pred()
```



Métriques pour régression (3)

```
rmse(ames_test_res, truth = Sale_Price, estimate = .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      0.0854
```

Métriques pour régression (3)

On notera qu'il peut y avoir plusieurs variables dans `ames_test_res` et que certaines métriques ne sont pas symétriques. Il faut donc faire attention.

```
ames_metrics <- metric_set(rmse, rsq, mae)
ames_metrics(ames_test_res, truth = Sale_Price, estimate = .pred)
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      0.0854
## 2 rsq     standard      0.776
## 3 mae     standard      0.0564
```

Métrique pour classification binaire (1)

Nous considérons un exemple factice (les données sont disponibles dans le package {yardstick}).

```
data(two_class_example)
str(two_class_example)
```

```
## 'data.frame':    500 obs. of  4 variables:
##  $ truth    : Factor w/ 2 levels "Class1","Class2": 2 1 2 1 2 1 1 1 2 2 ...
##  $ Class1   : num  0.00359 0.67862 0.11089 0.73516 0.01624 ...
##  $ Class2   : num  0.996 0.321 0.889 0.265 0.984 ...
##  $ predicted: Factor w/ 2 levels "Class1","Class2": 2 1 2 1 2 1 1 1 2 2 ...
```

Métrique pour classification binaire (2)

Les probabilités estimées d'appartenance aux différentes classes (la somme vaut 1) permet de prédire la classe en affectant chaque ligne à la classe la plus probable.

```
conf_mat(two_class_example, truth = truth, estimate = predicted)
```

```
##           Truth
## Prediction Class1 Class2
##     Class1    227     50
##     Class2     31    192
```

Métriques pour classification binaire (3)

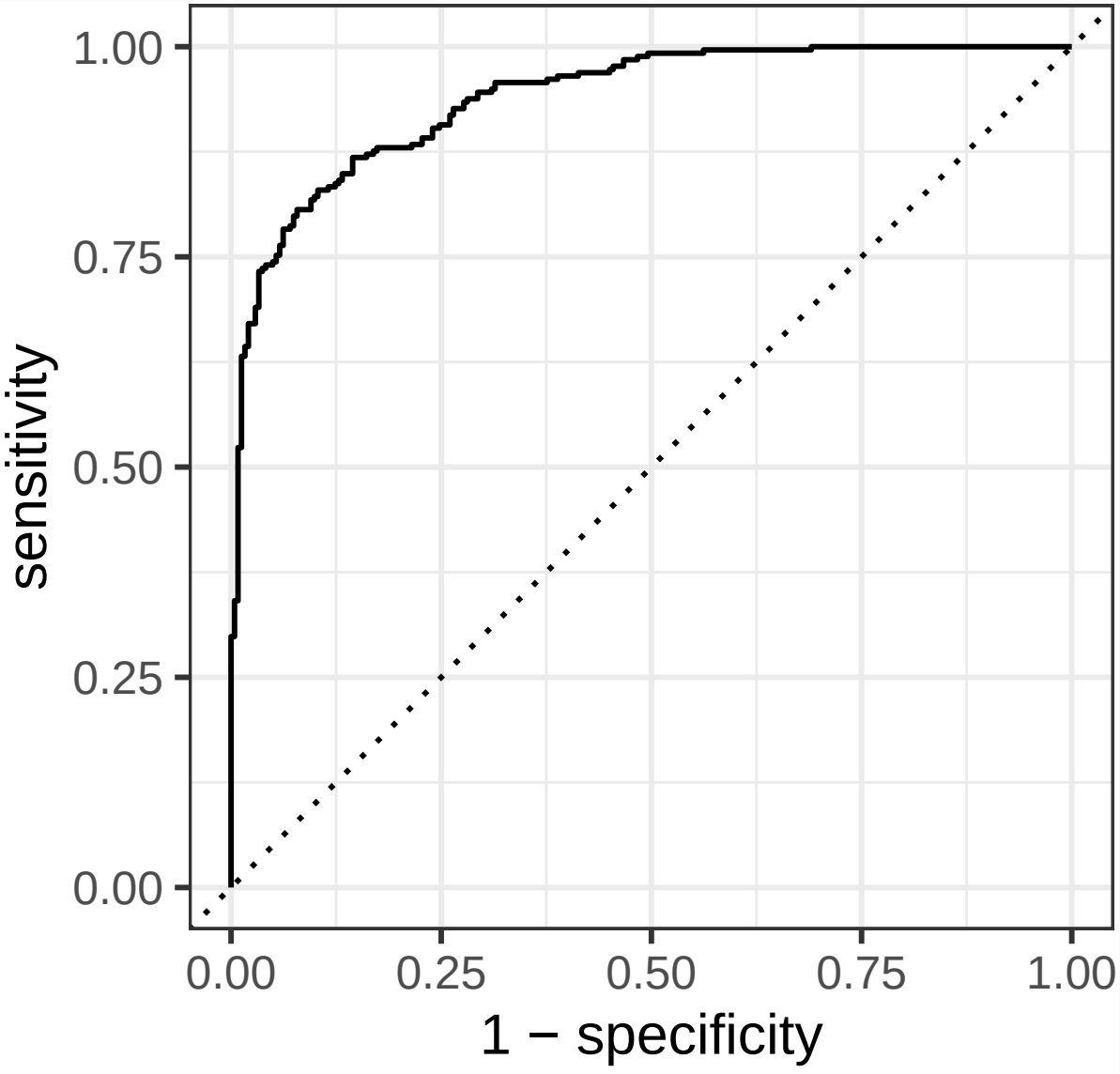
```
two_metrics <- metric_set(accuracy, mcc, f_meas)
```

```
two_metrics(two_class_example, truth = truth, estimate = predicted)
```

```
## # A tibble: 3 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.838
## 2 mcc      binary         0.677
## 3 f_meas   binary         0.849
```

Métriques pour classification binaire (4)

```
two_class_curve <- roc_curve(two_class_example, truth, Class1)
autoplot(two_class_curve)
```



Métriques pour classification binaire (5)

```
roc_auc(two_class_example, truth, Class1)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.939
```

Évaluation des performances

Après le modèle linéaire, les forêts aléatoires (1)

De la même manière qu'on a créé un worflow pour le modèle linéaire, on en crée un pour les forêts aléatoires.

```
rf_model <-
  rand_forest(trees = 1000) %>%
  set_engine("ranger") %>%
  set_mode("regression")

rf_wflow <-
  workflow() %>%
  add_formula(
    Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type +
      Latitude + Longitude) %>%
  add_model(rf_model)

rf_fit <- rf_wflow %>% fit(data = ames_train)
```

Après le modèle linéaire, les forêts aléatoires (2)

```
lm_fit
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## 5 Recipe Steps
##
## * step_log()
## * step_other()
## * step_dummy()
## * step_interact()
## * step_ns()
##
## -- Model -----------------------------------------------------------------------
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
##                                       (Intercept)
```

Après le modèle linéaire, les forêts aléatoires (2)

```
rf_fit
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Formula
## Model: rand_forest()
##
## -- Preprocessor ----------------------------------------------------------------
## Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type +
##     Latitude + Longitude
##
## -- Model -----------------------------------------------------------------------
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, num.trees = ~1000,      num.thre
##
## Type:                             Regression
## Number of trees:                  1000
## Sample size:                      2199
## Number of independent variables:  6
## Mtry:                             2
## Target node size:                 5
```

Comparaison naïve, erreur apparente (1)

```
estimate_perf <- function(model, dat) {
  # Capture the names of the objects used
  cl <- match.call()
  obj_name <- as.character(cl$model)
  data_name <- as.character(cl$dat)
  data_name <- gsub("ames_", "", data_name)

  # Estimate these metrics:
  reg_metrics <- metric_set(rmse, rsq)

  model %>%
    predict(dat) %>%
    bind_cols(dat %>% select(Sale_Price)) %>%
    reg_metrics(Sale_Price, .pred) %>%
    select(-.estimator) %>%
    mutate(object = obj_name, data = data_name)
}
```

Comparaison naïve, erreur apparente (2)

```
estimate_perf(lm_fit, ames_train)
```

```
## # A tibble: 2 x 4
##   .metric .estimate object data
##   <chr>       <dbl> <chr>  <chr>
## 1 rmse       0.0727 lm_fit train
## 2 rsq        0.829  lm_fit train
```

Comparaison naïve, erreur apparente (3)

```
estimate_perf(rf_fit, ames_train)
```

```
## # A tibble: 2 x 4
##   .metric .estimate object data
##   <chr>       <dbl> <chr>  <chr>
## 1 rmse       0.0344 rf_fit train
## 2 rsq        0.965  rf_fit train
```

On préfère conserver les forêts aléatoires.

Comparaison naïve, erreur apparente (4)

```
estimate_perf(rf_fit, ames_test)
```

```
## # A tibble: 2 x 4
##   .metric .estimate object data
##   <chr>       <dbl> <chr>  <chr>
## 1 rmse       0.0799 rf_fit test
## 2 rsq        0.806  rf_fit test
```

Déception !

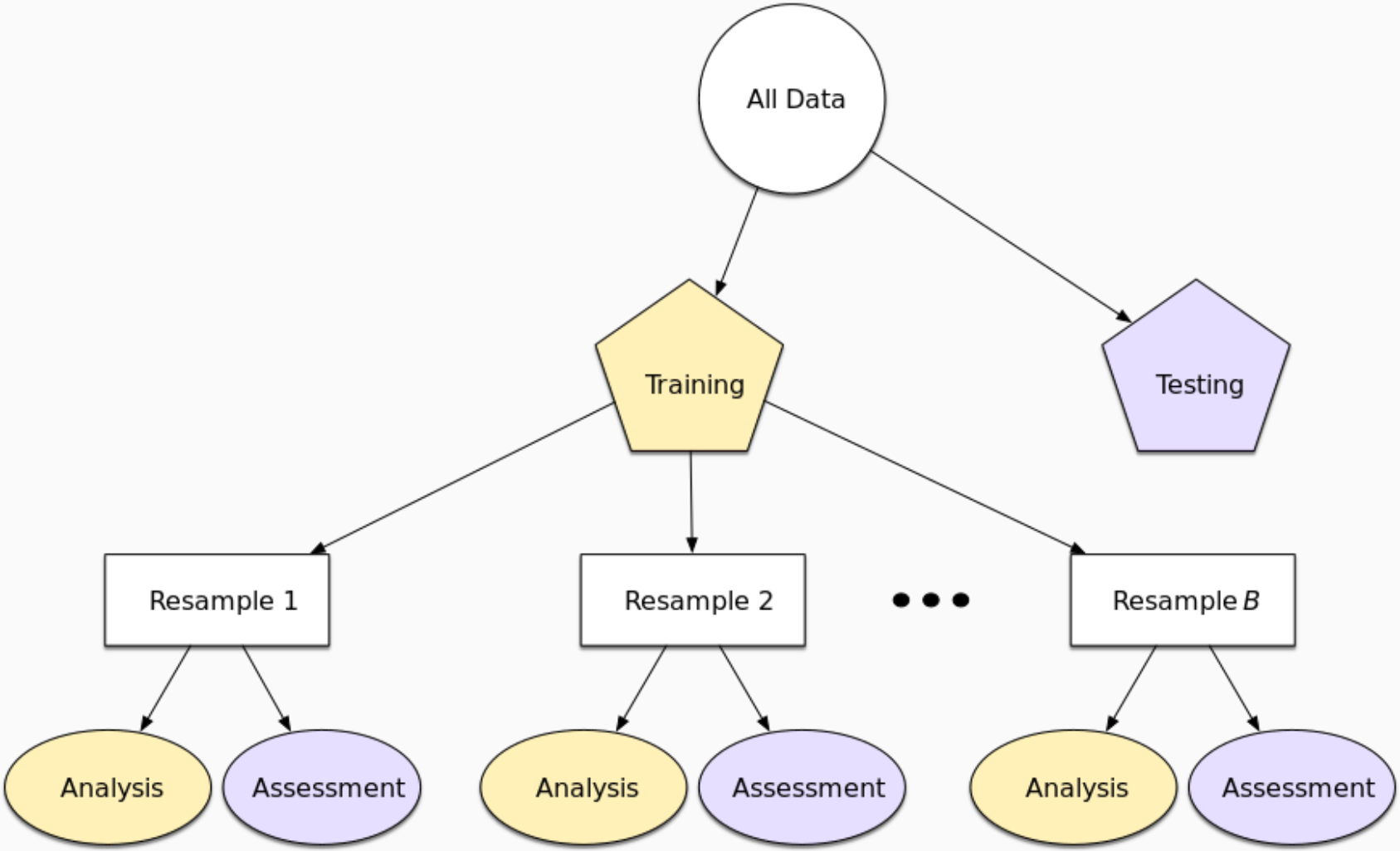


FIGURE 2 : Principe du rééchantillonage

Cross-validation (1)

Exemple avec 3 blocs.

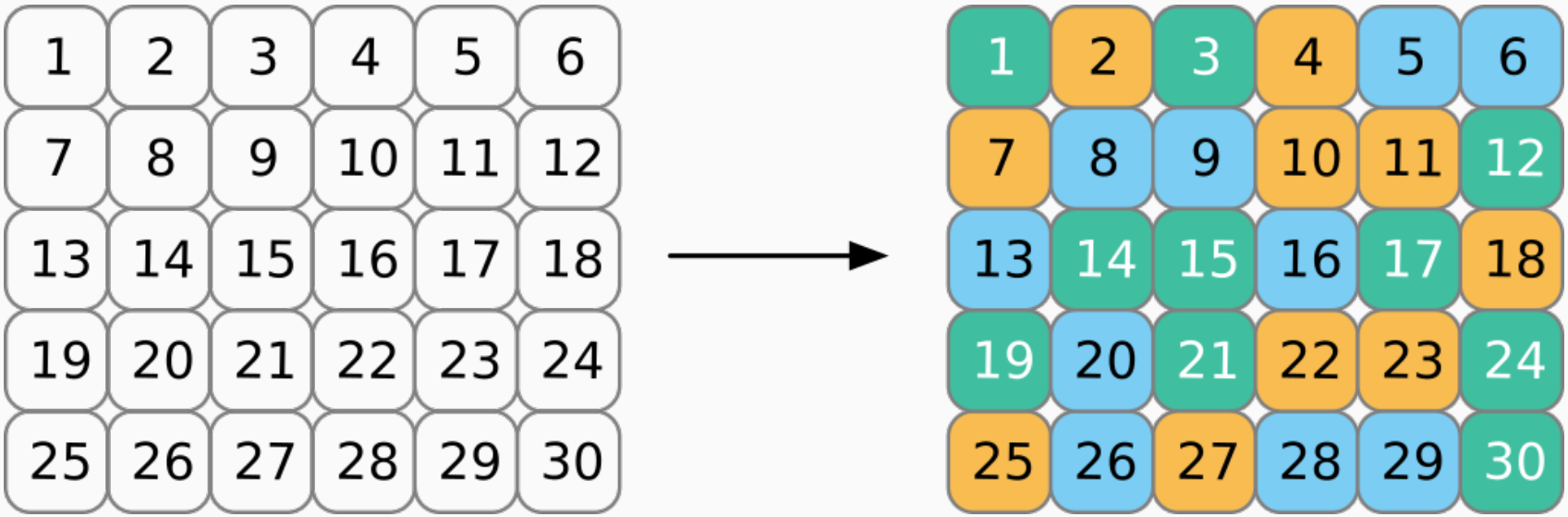


FIGURE 3 : Validation croisée 3_blocs

Cross-validation (2)

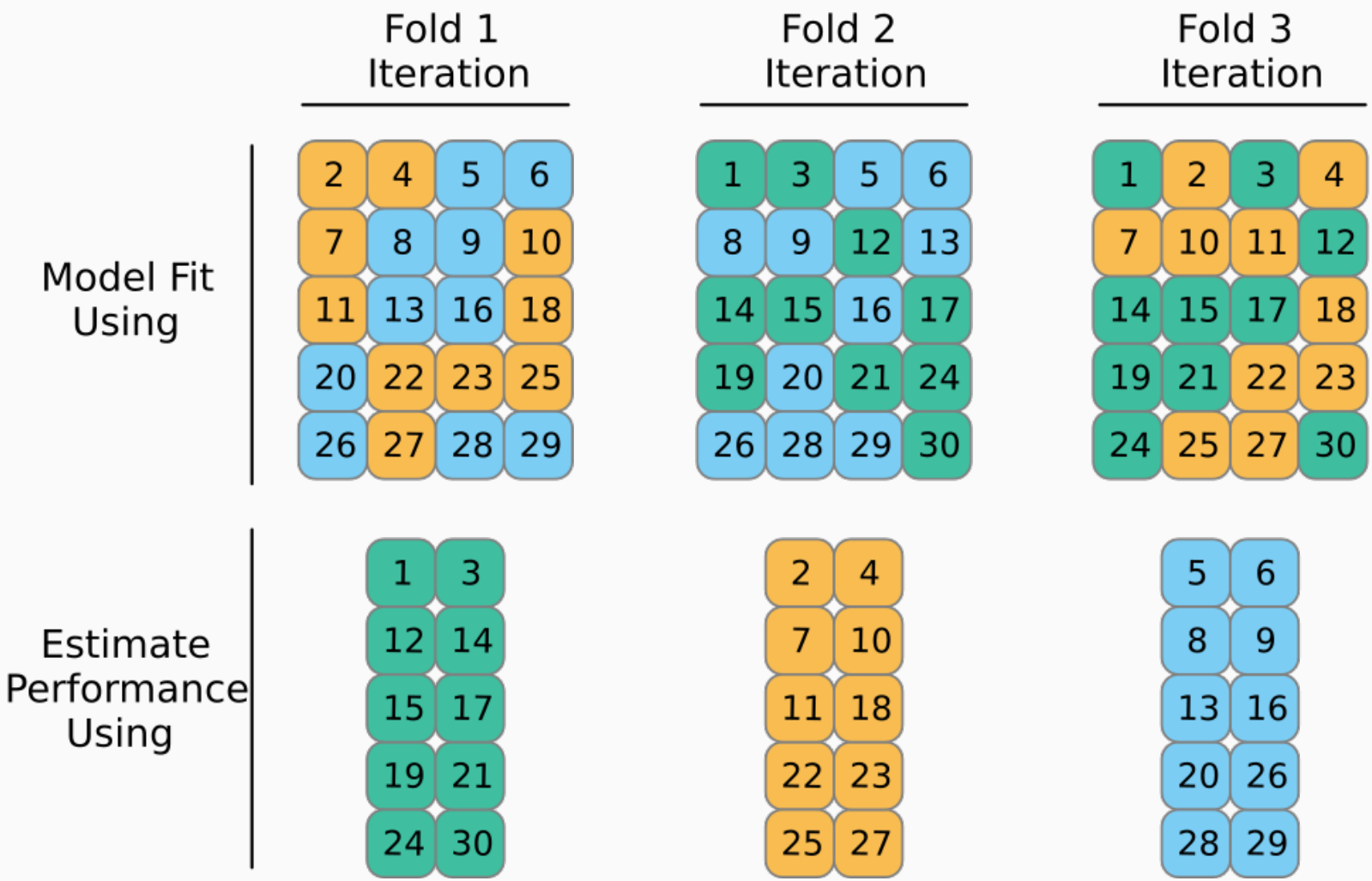


FIGURE 4 : Validation croisée en action

Bootstrap, tirage avec remise.

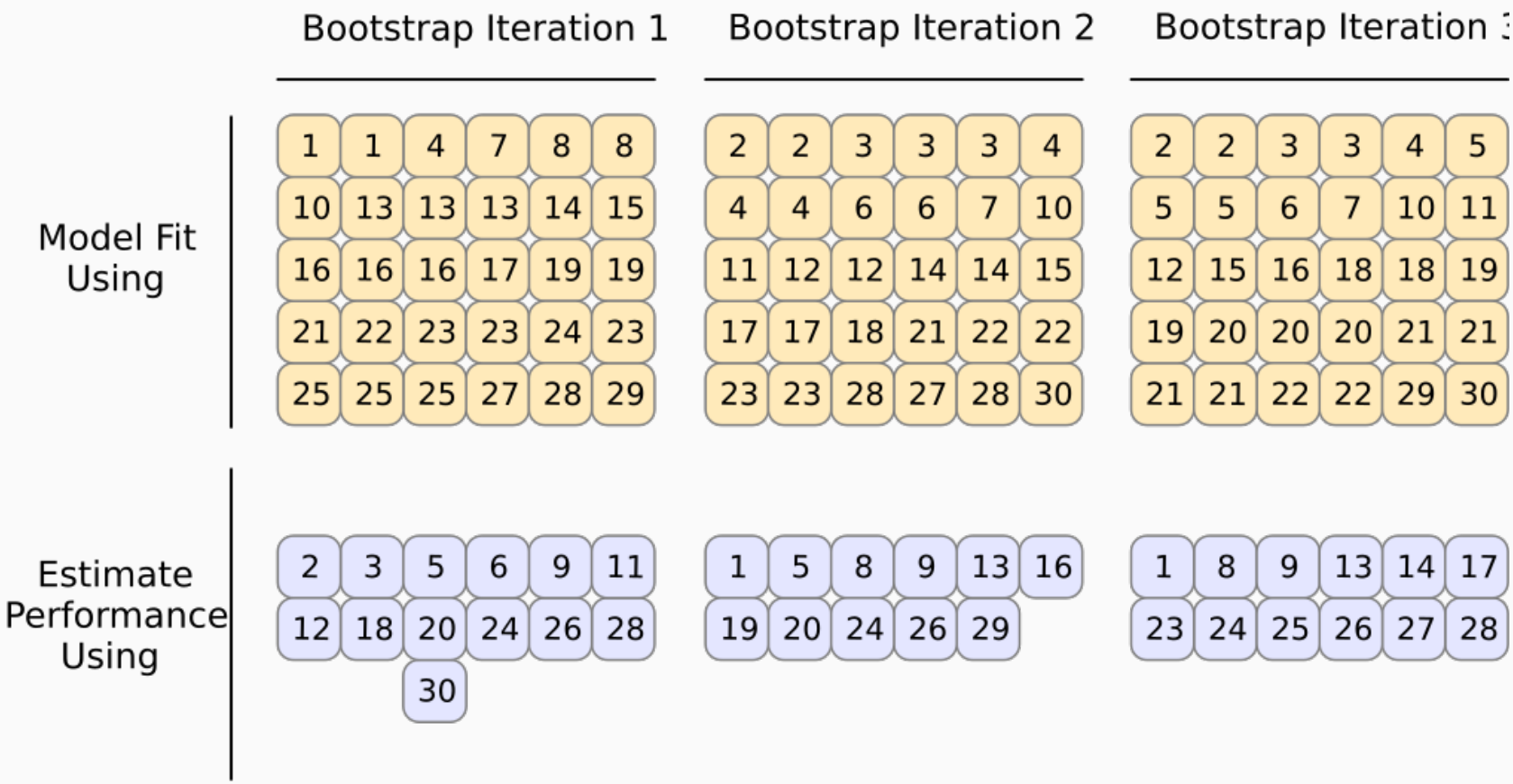


FIGURE 5 : Bootstrapt

Validation croisée en pratique (1)

```
set.seed(55)
ames_folds <- vfold_cv(ames_train, v = 10)
ames_folds %>% head(3)
```

```
## # A tibble: 3 x 2
##   splits             id
##   <list>             <chr>
## 1 <split [1979/220]> Fold01
## 2 <split [1979/220]> Fold02
## 3 <split [1979/220]> Fold03
```

Il faut comprendre que chaque ligne représente un découpage de l'échantillon **train** en deux sous-échantillon **analysis** et **assesment** un peu de la même façon qu'on avait découpé les données en **train** et **test** au début.

Validation croisée en pratique (2)

On peut retrouver ces données

```
# For the first fold:
ames_folds$splits[[1]] %>% analysis() %>% dim()
```

```
## [1] 1979   74
```

```
ames_folds$splits[[3]] %>% assessment() %>% dim()
```

```
## [1] 220  74
```

Bootstrap en pratique

```
ames_bootstrap <- bootstraps(ames_train, times = 5)
ames_bootstrap %>%
  pluck("splits", 1) %>%
  assessment() %>%
  dim()

## [1] 806  74
```

Principe

L'idée est d'évaluer les performances :

- à chacune des itérations ;
- en apprenant (`fit()`) sur l'échantillon d'analyse ;
- en calculant la métrique sur les prédictions sur l'échaintillon d'évaluation

PUIS de moyenner les métriques sur l'enble des itérations

Remarque

Comme le même procédé est fait de façon systématique, il existe une fonction pour automatiser cette tâche : `fit_resamples()` qui va remplacer `fit()` dans un workflow par exemple.

Exemple (1)

```
keep_pred <- control_resamples(save_pred = TRUE)
set.seed(130)
rf_res <-
  rf_wflow %>%
  fit_resamples(resamples = ames_folds, control = keep_pred)
rf_res
```

```
## # Resampling results
## # 10-fold cross-validation
## # A tibble: 10 x 5
##    splits             id     .metrics          .notes           .predictions
##    <list>             <chr>  <list>            <list>           <list>
##  1 <split [1979/220]> Fold01 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [220 x 4~
##  2 <split [1979/220]> Fold02 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [220 x 4~
##  3 <split [1979/220]> Fold03 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [220 x 4~
##  4 <split [1979/220]> Fold04 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [220 x 4~
##  5 <split [1979/220]> Fold05 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [220 x 4~
##  6 <split [1979/220]> Fold06 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [220 x 4~
##  7 <split [1979/220]> Fold07 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [220 x 4~
##  8 <split [1979/220]> Fold08 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [220 x 4~
##  9 <split [1979/220]> Fold09 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [220 x 4~
## 10 <split [1980/219]> Fold10 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [219 x 4~
```

Exemple (2)

```
collect_metrics(rf_res)
```

```
## # A tibble: 2 x 6
##   .metric .estimator   mean     n std_err .config
##   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1 rmse    standard   0.0697    10 0.00159 Preprocessor1_Model1
## 2 rsq     standard   0.843     10 0.00793 Preprocessor1_Model1
```

Exemple (3)

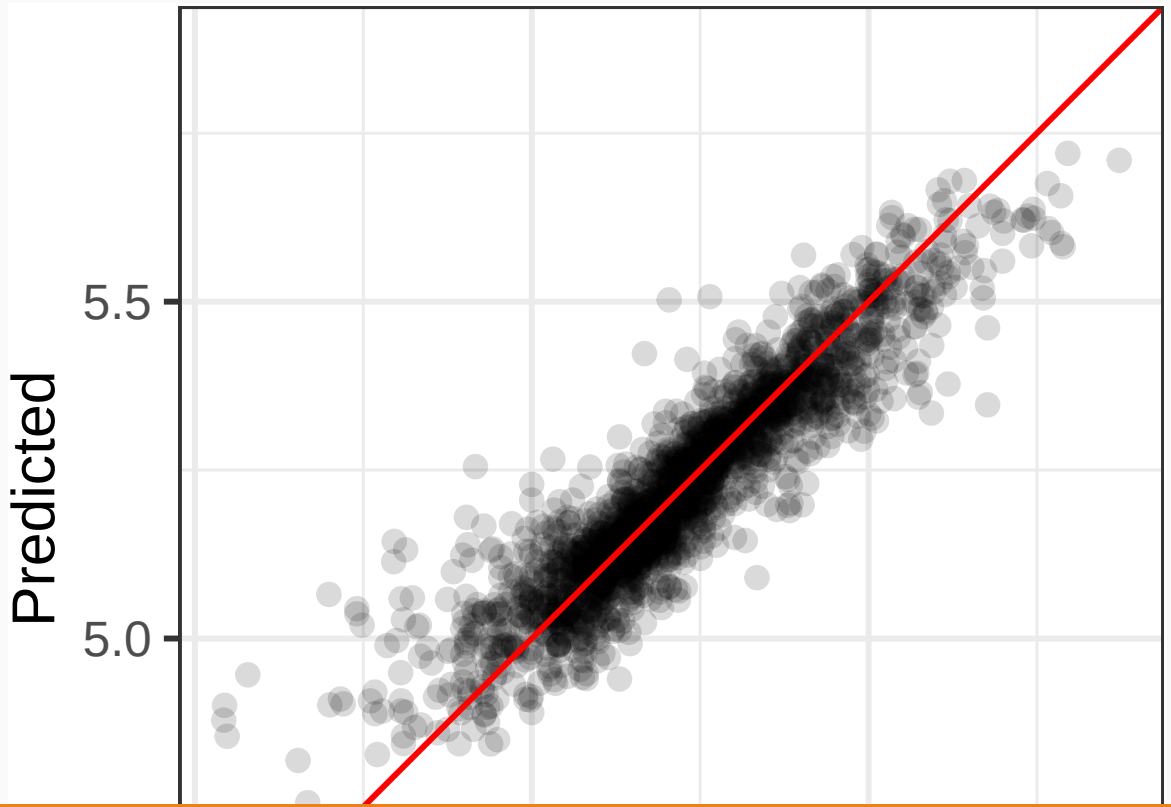
```
assess_res <- collect_predictions(rf_res)
assess_res
```

```
## # A tibble: 2,199 x 5
##    id     .pred  .row Sale_Price .config
##    <chr>  <dbl> <int>      <dbl> <chr>
##  1 Fold01  5.23    11       5.27 Preprocessor1_Model1
##  2 Fold01  5.29    47       5.24 Preprocessor1_Model1
##  3 Fold01  5.32    52       5.27 Preprocessor1_Model1
##  4 Fold01  5.44    94       5.39 Preprocessor1_Model1
##  5 Fold01  5.03   111       5.07 Preprocessor1_Model1
##  6 Fold01  5.17   115       5.25 Preprocessor1_Model1
##  7 Fold01  5.36   117       5.26 Preprocessor1_Model1
##  8 Fold01  5.26   118       5.28 Preprocessor1_Model1
##  9 Fold01  5.15   134       4.99 Preprocessor1_Model1
## 10 Fold01  5.12   140       5.20 Preprocessor1_Model1
## # ... with 2,189 more rows
```

Exemple (4)

C'est pas trop mal...

```
assess_res %>%
  ggplot(aes(x = Sale_Price, y = .pred)) +
  geom_point(alpha = .15) +
  geom_abline(col = "red") +
  coord_obs_pred() +
  ylab("Predicted")
```



Comparaison de modèles

Modèle linéaire avec splines

On redéfinit le workflow basé sur la recette avec les splines

```
lm_with_splines_res <-
  lm_wflow %>%
  fit_resamples(resamples = ames_folds, control = keep_pred)
```

Modèle linéaire sans spline (1)

On définit une nouvelle recette

```
no_spline_rec <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type +
           Latitude + Longitude, data = ames_train) %>%
  # Recall that Sale_Price is pre-logged
  step_log(Gr_Liv_Area, base = 10) %>%
  step_other(Neighborhood, threshold = 0.01) %>%
  step_dummy(all_nominal()) %>%
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") )
```

Modèle linéaire sans spline (2)

On définit un nouveau worflow basé sur cette recette

```
lm_no_splines_res <-
  lm_wflow %>%
  remove_recipe() %>%
  add_recipe(no_spline_rec) %>%
  fit_resamples(resamples = ames_folds, control = keep_pred)
```

Comparaison des modèles (1)

```
collect_metrics(lm_no_splines_res)
```

```
## # A tibble: 2 x 6
##   .metric .estimator   mean     n std_err .config
##   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1 rmse    standard   0.0762    10 0.00195 Preprocessor1_Model1
## 2 rsq     standard   0.811     10 0.0112  Preprocessor1_Model1
```

```
collect_metrics(lm_with_splines_res)
```

```
## # A tibble: 2 x 6
##   .metric .estimator   mean     n std_err .config
##   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1 rmse    standard   0.0753    10 0.00179 Preprocessor1_Model1
## 2 rsq     standard   0.816     10 0.00973 Preprocessor1_Model1
```

Comparaison des modèles (2)

Les deux modèles sont très proches => on conserve le plus simple.

Optimisation des paramètres d'un modèle

Paramètre à optimiser dans une recette

La fonction `tune()` permet de délarer certains paramètres comme étant à optimiser.

```
ames_rec <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type +
           Latitude + Longitude, data = ames_train)  %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_other(Neighborhood, threshold = tune()) %>%
  step_dummy(all_nominal()) %>%
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") ) %>%
  step_ns(Longitude, deg_free = tune("longitude df")) %>%
  step_ns(Latitude,  deg_free = tune("latitude df"))

recipes_param <- parameters(ames_rec)
recipes_param

## Collection of 3 parameters for tuning
##
##    identifier      type    object
##     threshold threshold nparam[+]
##  longitude df  deg_free nparam[+]
##   latitude df  deg_free nparam[+]
```

Workflow (1)

```
wflow_param <-
  workflow() %>%
  add_recipe(ames_rec) %>%
  add_model(rf_spec) %>%
  parameters()
wflow_param
```

```
## Collection of 6 parameters for tuning
##
##    identifier      type    object
##          mtry      mtry nparam[?]
##         trees     trees nparam[+]
##         min_n     min_n nparam[+]
##     threshold threshold nparam[+]
##  longitude df  deg_free nparam[+]
##   latitude df  deg_free nparam[+]
##
## Model parameters needing finalization:
##    # Randomly Selected Predictors ('mtry')
##
## See `?dials::finalize` or `?dials::update.parameters` for more information.
```

Workflow (2)

Des fonctions spécifiques ont été créées.

```
mtry()
```

```
## # Randomly Selected Predictors (quantitative)
## Range: [1, ?]
```

```
threshold()
```

```
## Threshold (quantitative)
## Range: [0, 1]
```

Workflow (3)

Mais attention aux conventions de nommage.

```
spline_degree()
```

```
## Piecewise Polynomial Degree (quantitative)
## Range: [1, 10]
```

Workflow (4)

```
flow_param <- wflow_param %>%
  update(mtry = mtry(c(1, 10)))
flow_param
```

```
## Collection of 6 parameters for tuning
##
##    identifier      type    object
##          mtry      mtry nparam[+]
##         trees     trees nparam[+]
##         min_n     min_n nparam[+]
##     threshold threshold nparam[+]
##  longitude df  deg_free nparam[+]
##   latitude df  deg_free nparam[+]
```

Workflow (5)

```
flow_param %>% pull_dials_object("trees")
```

```
## # Trees (quantitative)
## Range: [1, 2000]
```

Workflow (5)

```
grid_regular(flow_param, levels = 2)
```

```
## # A tibble: 64 x 6
##     mtry trees min_n threshold `longitude df` `latitude df`
##    <int> <int> <int>     <dbl>          <int>         <int>
##  1     1     1     2       0                1             1
##  2    10     1     2       0                1             1
##  3     1  2000     2       0                1             1
##  4    10  2000     2       0                1             1
##  5     1     1    40       0                1             1
##  6    10     1    40       0                1             1
##  7     1  2000    40       0                1             1
##  8    10  2000    40       0                1             1
##  9     1     1     2       0.1              1             1
## 10    10     1     2       0.1              1             1
## # ... with 54 more rows
```

Remarque 64 lignes = 2^6 (6 paramètres)

Workflow(6)

```
rf_tune <- workflow() %>%
  add_recipe(ames_rec) %>%
  add_model(rf_spec) %>%
  tune_grid(
    vfold_cv(ames_train),
    grid = flow_param %>% grid_regular(levels = 2),
    metrics = metric_set(rmse)
  )
```

Workflow (7)

```
show_best(rf_tune, metric = "rmse") %>%
  glimpse()
```

```
## Rows: 5
## Columns: 12
## $ mtry           <int> 10, 10, 10, 10, 10
## $ trees          <int> 2000, 2000, 2000, 2000, 2000
## $ min_n          <int> 2, 2, 2, 2, 2
## $ threshold      <dbl> 0.0, 0.1, 0.1, 0.1, 0.0
## $ `longitude df` <int> 1, 15, 15, 1, 15
## $ `latitude df`  <int> 1, 15, 1, 15, 1
## $ .metric        <chr> "rmse", "rmse", "rmse", "rmse", "rmse"
## $ .estimator     <chr> "standard", "standard", "standard", "standard", "standa~
## $ mean           <dbl> 0.07007959, 0.07052850, 0.07067338, 0.07096647, 0.07134~
## $ n              <int> 10, 10, 10, 10, 10
## $ std_err        <dbl> 0.001505927, 0.001789790, 0.001611371, 0.001664433, 0.0~
## $ .config        <chr> "Preprocessor1_Model4", "Preprocessor8_Model4", "Prepro~
```