09 - Machine Learning

PRO1036 - Analyse de données scientifiques en R

Tim Bollé

November 25, 2024

Modélisation

Buts de l’analyse de données

- Comprendre les données
- Extraire des informations
- Prédire des valeurs
- Prendre des décisions

Modélisation

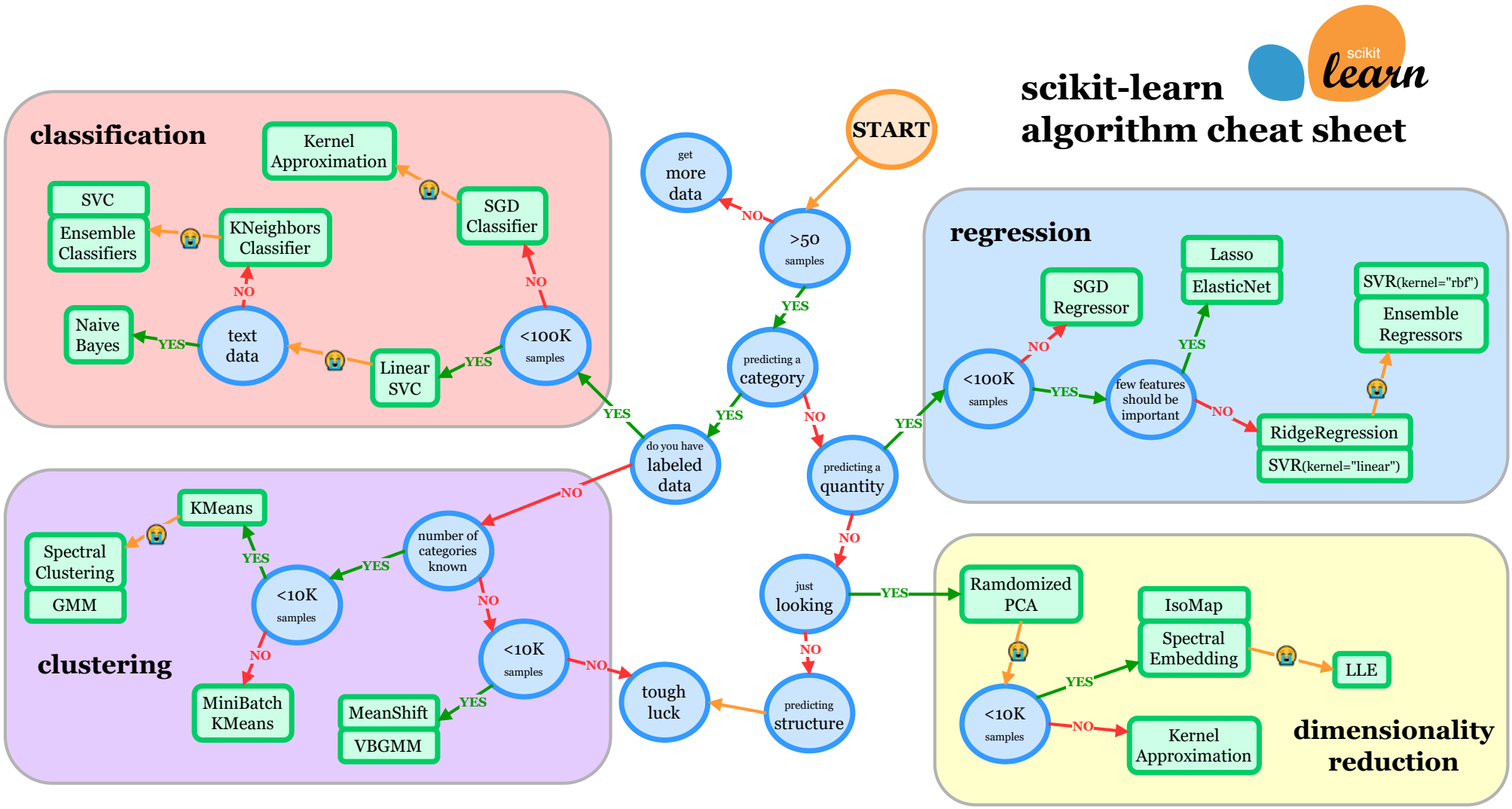
Intervient dans le processus d'analyse de données

Le machine learning permet notamment de prédire des valeurs et prendre des décisions.

Types de modèles

- Modèles descriptifs
- Modèles inférentiels
- Modèles prédictifs

Types de modèles



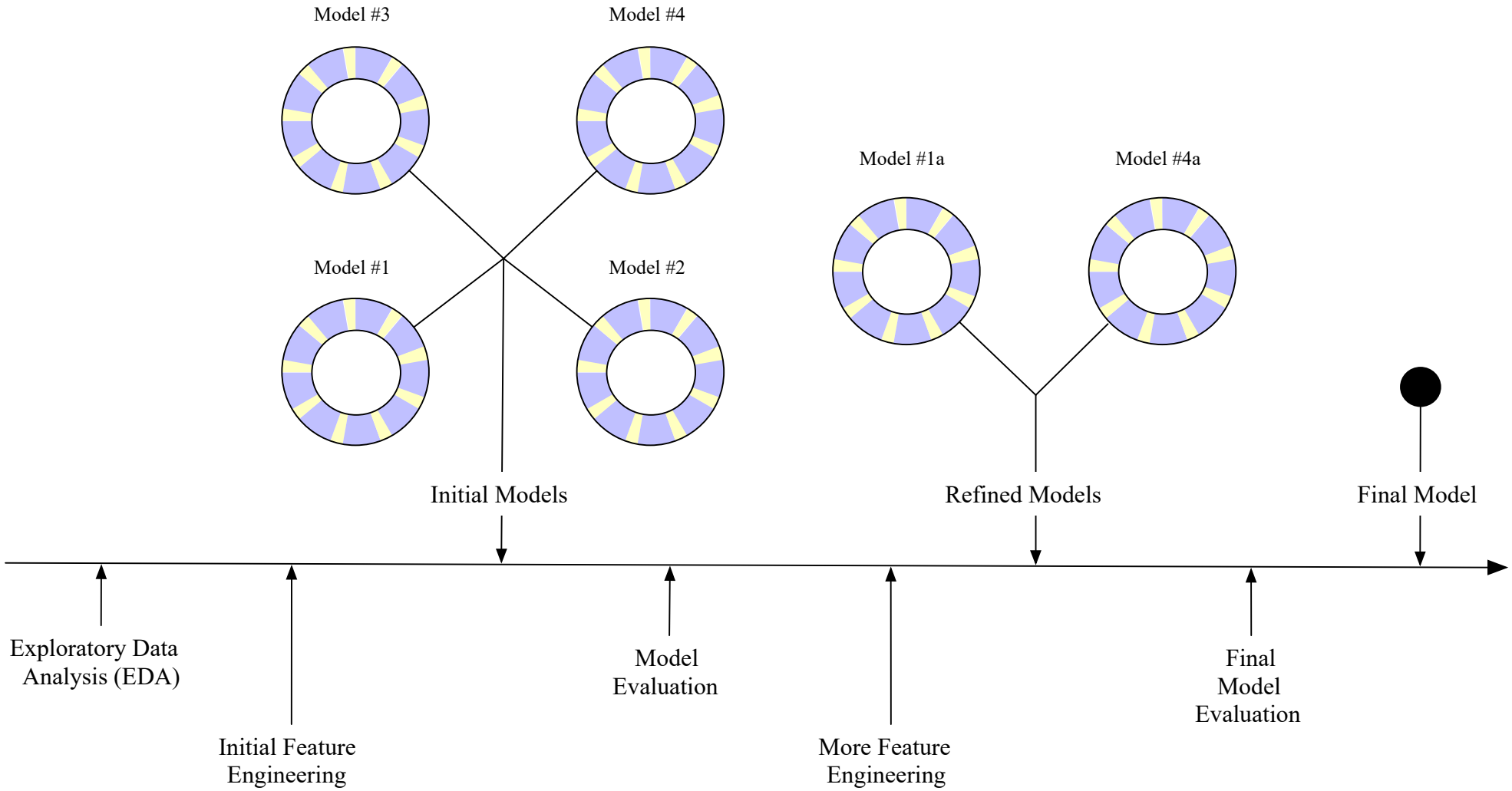
Types de machine learning

- **Supervised learning**: On a des données étiquetées
  - Classification
  - Régression
- **Unsupervised learning**: On a des données non étiquetées
  - Clustering
  - Réduction de dimension
- …

Processus de modélisation

- **Exploratory data analysis (EDA)**: Comprendre les données, se poser des questions
- **Feature engineering**: Créer de nouvelles variables à partir des données ou sélectionner les variables les plus pertinentes
- **Sélection de modèle et tuning**: Choisir le modèle le plus adapté et ajuster les hyperparamètres
- **Évaluation du modèle**: Mesurer la performance du modèle

Processus de modélisation



Modeling et tidyverse

Tidymodel est un ensemble de packages qui permettent de faire du machine learning avec le tidyverse.

On retrouve les packages:

- `parsnip`: Interface pour les modèles
- `dials`: Sélection d'hyperparamètres
- `tune`: Tuning des hyperparamètres
- `workflow`: Workflows pour les modèles
- `yardstick`: Métriques de performance
- `recipes`: Préparation des données

Modélisation - Étapes générales

Étapes générales

0. **Préparation des données**: Nettoyage, transformation, normalisation
1. **Séparation des données**: Entraînement et test
2. **Feature engineering**: Création de nouvelles variables ou sélection des variables les plus pertinentes
3. **Choix du modèle**: Sélection du modèle le plus adapté
4. **Entraînement du modèle**: Apprentissage du modèle sur les données d'entraînement
5. **Évaluation du modèle**: Mesure de la performance du modèle sur les données de test

Les données

Pour les slides suivantes, nous allons utiliser le jeu de données `cells` du package `modeldata`, publiées par Hill, LaPan, Li, and Haney (2007).

Le jeu de données concerne des images de cellules cancéreuses et indique si elles sont bien segmentées (`WS`) ou non (`PS`).

```
1 glimpse(cells)
```

```
Rows: 2,019
Columns: 58
$ case                          <fct> Test, Train, Train, Train, Test, Test, Te…
$ class                         <fct> PS, PS, WS, PS, PS, WS, WS, PS, WS, WS, W…
$ angle_ch_1                    <dbl> 143.247705, 133.752037, 106.646387, 69.15…
$ area_ch_1                     <int> 185, 819, 431, 298, 285, 172, 177, 251, 4…
$ avg_inten_ch_1                <dbl> 15.71186, 31.92327, 28.03883, 19.45614, 2…
$ avg_inten_ch_2                <dbl> 4.954802, 206.878517, 116.315534, 102.294…
$ avg_inten_ch_3                <dbl> 9.548023, 69.916880, 63.941748, 28.217544…
$ avg_inten_ch_4                <dbl> 2.214689, 164.153453, 106.696602, 31.0280…
$ convex_hull_area_ratio_ch_1   <dbl> 1.124509, 1.263158, 1.053310, 1.202625, 1…
$ convex_hull_perim_ratio_ch_1  <dbl> 0.9196827, 0.7970801, 0.9354750, 0.865829…
$ diff_inten_density_ch_1       <dbl> 29.51923, 31.87500, 32.48771, 26.73228, 3…
$ diff_inten_density_ch_3       <dbl> 13.77564, 43.12228, 35.98577, 22.91732, 2…
$ diff_inten_density_ch_4       <dbl> 6.826923, 79.308424, 51.357050, 26.393701…
$ entropy_inten_ch_1            <dbl> 4.969781, 6.087592, 5.883557, 5.420065, 5…
$ entropy_inten_ch_3            <dbl> 4.371017, 6.642761, 6.683000, 5.436732, 5…
$ entropy_inten_ch_4            <dbl> 2.718884, 7.880155, 7.144601, 5.778329, 5…
$ eq_circ_diam_ch_1             <dbl> 15.369541, 32.514368, 23.44892, 19.50279, 1…
```

```
$ eq_ellipse_lwr_ch_1          <dbl> 3.060676, 1.558394, 1.375386, 3.391220, 2…
$ eq_ellipse_oblate_vol_ch_1   <dbl> 336.9691, 2232.9055, 802.1945, 724.7143, …
$ eq_ellipse_prolate_vol_ch_1  <dbl> 110.0963, 1432.8246, 583.2504, 213.7031, …
$ eq_sphere_area_ch_1          <dbl> 742.1156, 3278.7256, 1727.4104, 1104.9220…
```

Séparation des données

Séparation des données

Il est commun de séparer les données en deux partitions:

- **Données d'entraînement**: Pour estimer les paramètres, comparer les modèles, ajuster les hyperparamètres, etc.
- **Données de test**: Pour évaluer la performance finale du modèle. Cette partie est gardée en réserve jusqu'à la fin du projet.

Il existe différentes façons de créer ces partitions des données. L'approche la plus courante est d'utiliser un échantillon aléatoire. Supposons qu'un quart des données soit réservé pour les données de test. L'échantillonnage aléatoire sélectionnerait aléatoirement 25% pour les données de test et utiliserait le reste pour les données d'entraînement. On peut utiliser le package `rsample` à cet effet.

Séparation des données

Une première approche assez simple serait de séparer les données en deux groupes selon une proportion fixée:

```
cell_preprossed <- cells %>%
  select(-case)

cell_split <- initial_split(cell_preprossed)
```

Séparation des données

Par défaut, la proportion de données dans le set d'entrainement est de 75%. Ici, nous avons un petit soucis: nos classes ne sont pas équilibrées.

```
1 cells %>%
2   count(class) %>%
3   mutate(prop = n/sum(n))
```

```
# A tibble: 2 × 3
  class     n  prop
  <fct> <int> <dbl>
1 PS     1300 0.644
2 WS      719 0.356
```

La solution est de préciser que nous voulons une stratification des données.

```
1 cell_split <- initial_split(cell_preprossed,
2                             strata = class)
```

Séparation des données

Nous pouvons alors séparer nos deux jeux de données.

```
1 cell_train <- training(cell_split)
2 cell_test  <- testing(cell_split)
```

Les proportions sont maintenant équilibrées.

```
1 cell_train %>%
2   count(class) %>%
3   mutate(prop = n/sum(n))
```

```
# A tibble: 2 × 3
  class     n  prop
  <fct> <int> <dbl>
1 PS      975 0.644
2 WS      539 0.356
```

```
1 cell_test %>%
2   count(class) %>%
3   mutate(prop = n/sum(n))
```

```
# A tibble: 2 × 3
  class     n  prop
  <fct> <int> <dbl>
1 PS      325 0.644
2 WS      180 0.356
```

Feature engineering

Dans cet exemple, il n’est pas nécessaire de faire une feature engineering. Nous allons utiliser toutes les variables disponibles.

Modélisation

Choix du modèle

Nous allons utiliser un modèle de régression logistique pour prédire si une cellule est bien segmentée ou non.

```
logistic_reg <- logistic_reg() %>%
  set_engine("glm") %>%
  set_mode("classification")
```

Entraînement du modèle

```
1 logistic_fit <- logistic_reg %>%
2   fit(class ~ ., data = cell_train)
```

Évaluation du modèle

Commençons par voir les performances du modèle sur les données d'entraînement.

```
gml_training_pred <-
  predict(logistic_fit, cell_train) %>%
  bind_cols(predict(logistic_fit, cell_train, type = "prob")) %>%
  # Add the true outcome data back in
  bind_cols(cell_train %>%
              select(class))
```

```
gml_training_pred %>%                # training set predictions
  roc_auc(truth = class, .pred_PS)   # truth and predicted probabilities
```

```
# A tibble: 1 × 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 roc_auc binary         0.896
```

```
gml_training_pred %>%                # training set predictions
  accuracy(truth = class, .pred_class)
```

```
# A tibble: 1 × 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.822
```

Prédiction avec le modèle

```
gml_test_pred <-
  predict(logistic_fit, cell_test) %>%
  bind_cols(predict(logistic_fit, cell_test, type = "prob")) %>%
  # Add the true outcome data back in
  bind_cols(cell_test %>%
              select(class))
```

```
gml_test_pred %>%                  # training set predictions
  roc_auc(truth = class, .pred_PS)  # truth and predicted probabilities
```

```
# A tibble: 1 × 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 roc_auc binary         0.894
```

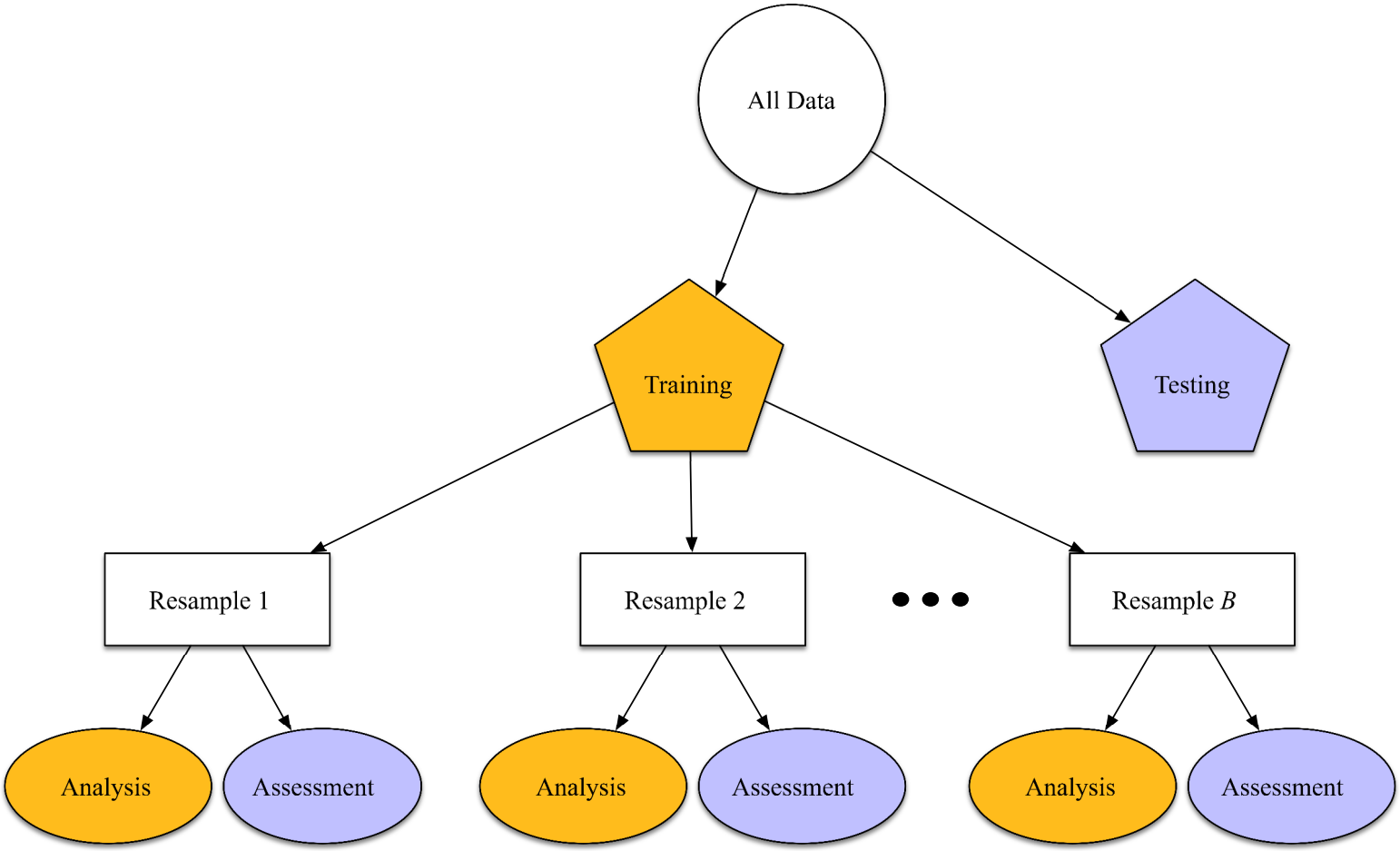
```
gml_test_pred %>%                  # training set predictions
  accuracy(truth = class, .pred_class)
```

```
# A tibble: 1 × 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.814
```

Rééchantillonnage

Rééchantillonnage

Pour obtenir une meilleure estimation de la performance du modèle, on peut utiliser le rééchantillonnage.



Rééchantillonnage

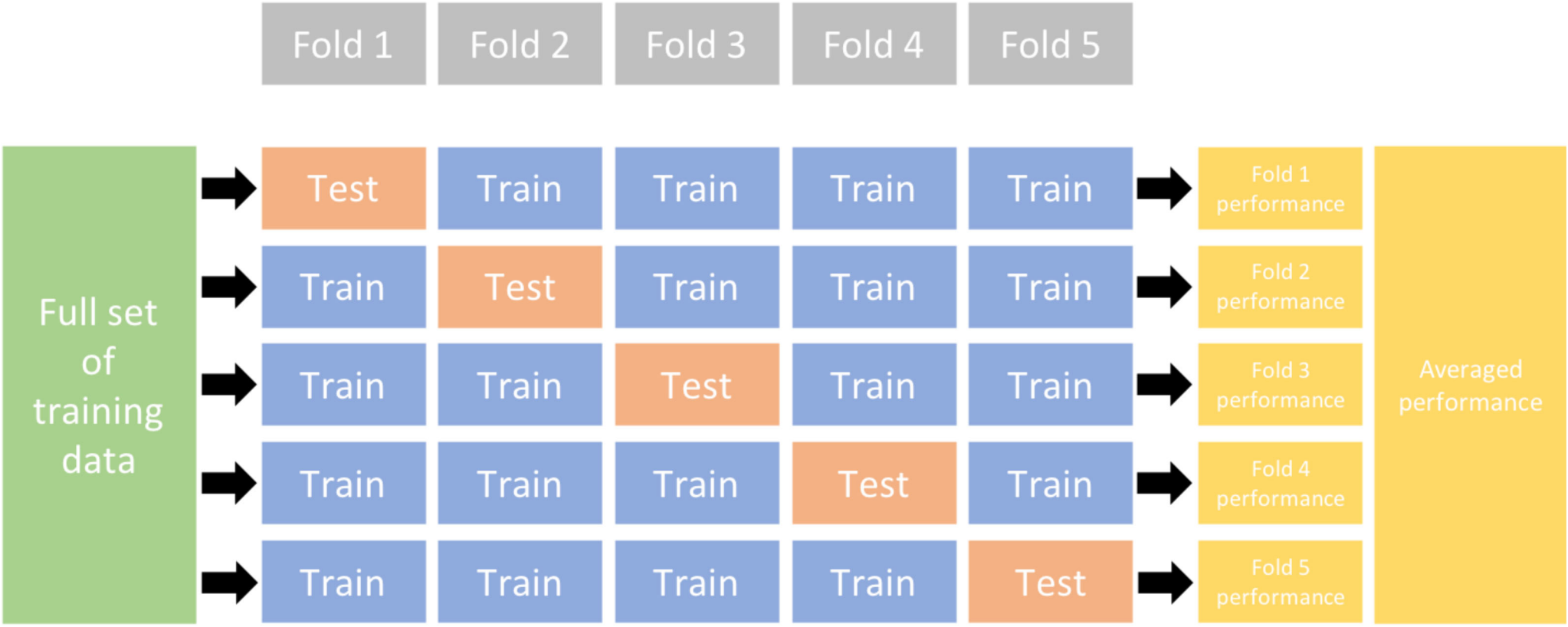
Il existe plusieurs méthodes de rééchantillonnage:

- Validation croisée
- Bootstrap
- Leave-one-out
- …

Nous allons utiliser la validation croisée.

```
1  folds <- vfold_cv(cell_train, v = 10)
```

Validation croisée



Rééchantillonnage

Nous pouvons alors entraîner notre modèle sur les différents *folds*.

```
gml_res <- logistic_reg %>%
  fit_resamples(class ~ .,
                resamples = folds)

collect_metrics(gml_res)
```

```
# A tibble: 3 × 6
  .metric     .estimator  mean     n std_err .config
  <chr>       <chr>      <dbl> <int>   <dbl> <chr>
1 accuracy    binary     0.797    10 0.00904 Preprocessor1_Model1
2 brier_class binary     0.139    10 0.00511 Preprocessor1_Model1
3 roc_auc     binary     0.871    10 0.00875 Preprocessor1_Model1
```

Nous voyons que les performances sont assez similaires à celles obtenues avec les données de test !

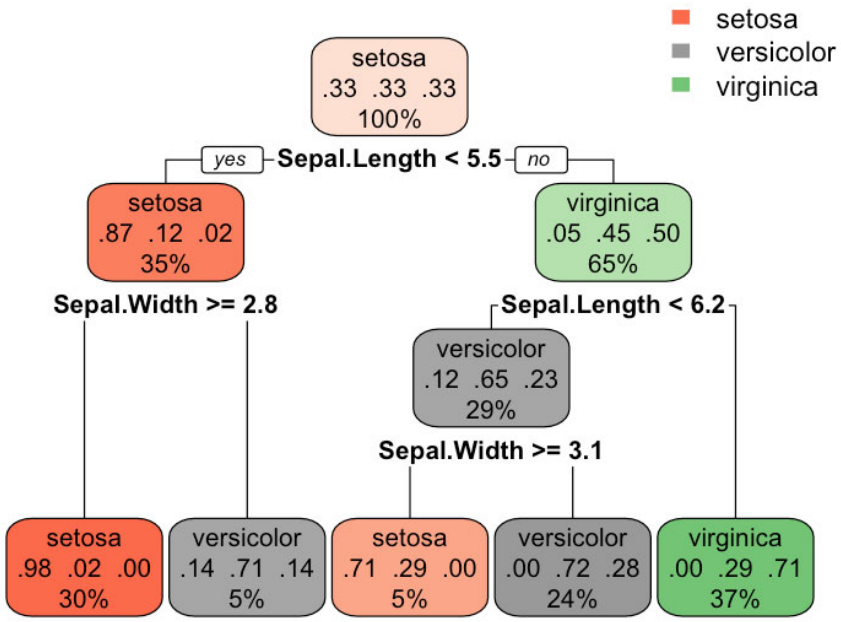
Optimisation des hyperparamètres

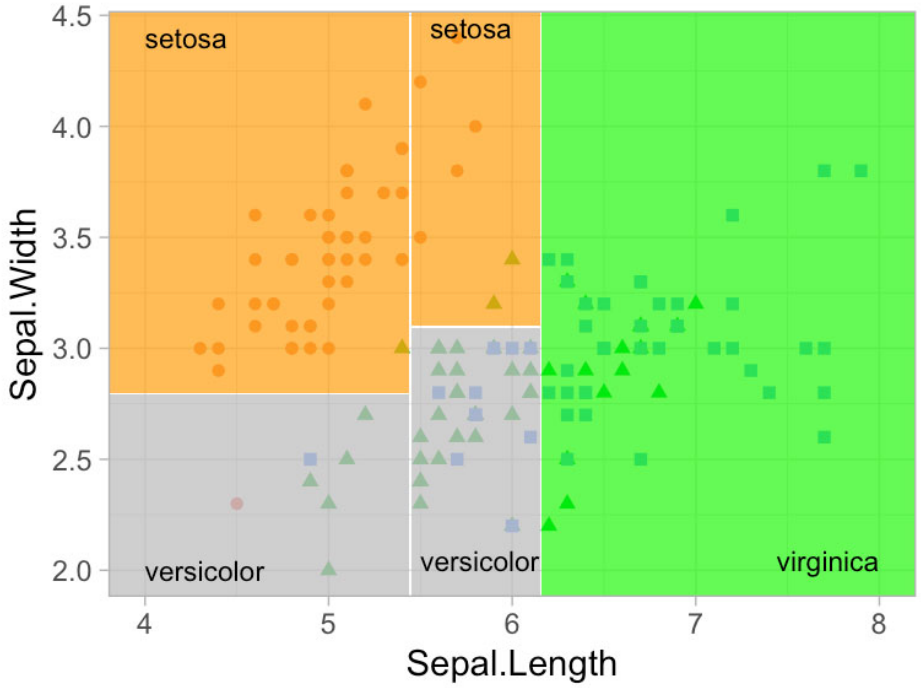
`tune()`

Certains modèles possèdent des hyperparamètres qui peuvent être ajustés pour améliorer la performance du modèle.

Nous allons prendre comme exemple le même jeu de données mais cette fois nous allons utiliser un modèle d’arbre de décision.

Arbre de décisions





Arbre de décisions

Il y a deux hyperparamètres principaux pour les arbres de décisions:

- `cost_complexity`: Ajoute une pénalité pour la complexité de l'arbre et permet d'éviter l'overfitting
- `tree_depth`: Profondeur maximale de l'arbre

```
tune_spec <-
  decision_tree(
    cost_complexity = tune(), # Paramètre à tuner
    tree_depth = tune()       # Paramètre à tuner
  ) %>%
  set_engine("rpart") %>%
  set_mode("classification")

tune_spec
```

```
Decision Tree Model Specification (classification)

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

Computational engine: rpart
```

Recherche des hyperparamètres

Nous allons ensuite lui donner une grille de recherche. Il va tester toutes les combinaisons possibles des hyperparamètres pour trouver la meilleure combinaison.

```
tree_grid <- grid_regular(cost_complexity(),
                          tree_depth(),
                          levels = 5)
```

Recherche des hyperparamètres

On commence par séparer nos données en données d'entraînement et de test.

```
set.seed(42)
cell_split <- initial_split(cells %>% select(-case),
                            strata = class)
cell_train <- training(cell_split)
cell_test  <- testing(cell_split)

cell_folds <- vfold_cv(cell_train)
```

Recherche des hyperparamètres

On peut maintenant lancer la recherche des hyperparamètres.

```
tune_res <- tune_grid(tune_spec,
                      class ~ .,          # On indique la formule
                      resamples = folds,
                      grid = tree_grid)
```

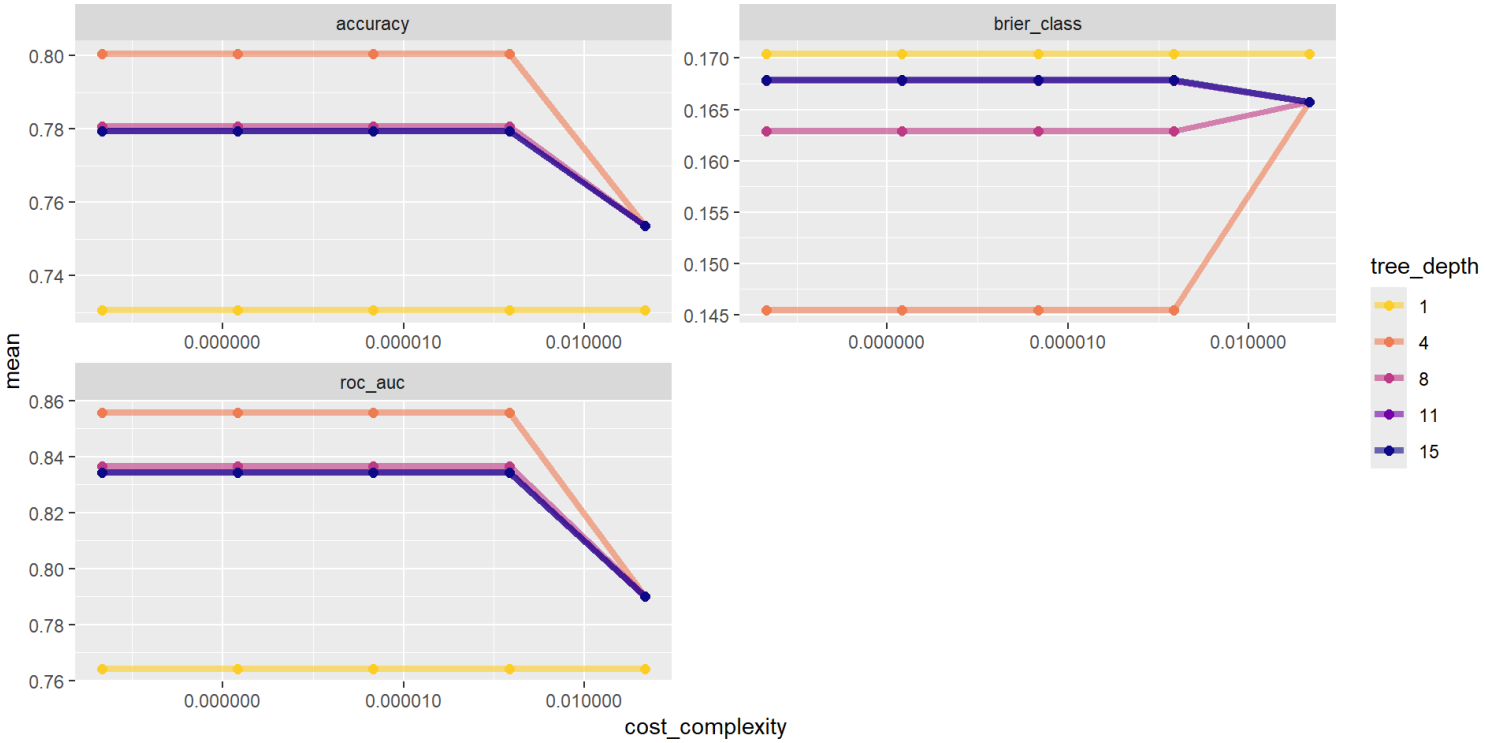
Analyse des résultats

```
1 tune_res %>%
2   collect_metrics()
```

```
# A tibble: 75 × 8
   cost_complexity tree_depth .metric     .estimator  mean     n std_err .config
             <dbl>      <int> <chr>       <chr>      <dbl> <int>   <dbl> <chr>
 1    0.0000000001          1 accuracy    binary     0.731    10 0.0121  Prepro…
 2    0.0000000001          1 brier_class binary     0.170    10 0.00294 Prepro…
 3    0.0000000001          1 roc_auc     binary     0.764    10 0.00790 Prepro…
 4    0.0000000178          1 accuracy    binary     0.731    10 0.0121  Prepro…
 5    0.0000000178          1 brier_class binary     0.170    10 0.00294 Prepro…
 6    0.0000000178          1 roc_auc     binary     0.764    10 0.00790 Prepro…
 7    0.00000316            1 accuracy    binary     0.731    10 0.0121  Prepro…
 8    0.00000316            1 brier_class binary     0.170    10 0.00294 Prepro…
 9    0.00000316            1 roc_auc     binary     0.764    10 0.00790 Prepro…
10    0.000562              1 accuracy    binary     0.731    10 0.0121  Prepro…
# ℹ 65 more rows
```

Analyse des résultats

```
tune_res %>%
  collect_metrics() %>%
  mutate(tree_depth = factor(tree_depth)) %>%
  ggplot(aes(cost_complexity, mean, color = tree_depth)) +
  geom_line(size = 1.5, alpha = 0.6) +
  geom_point(size = 2) +
  facet_wrap(~ .metric, scales = "free", nrow = 2) +
  scale_x_log10(labels = scales::label_number()) +
  scale_color_viridis_d(option = "plasma", begin = .9, end = 0)
```



Choix des hyperparamètres

```
tune_res %>%
  show_best(metric = "accuracy")
```

```
# A tibble: 5 × 8
  cost_complexity tree_depth .metric  .estimator  mean     n std_err .config
            <dbl>      <int> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
1    0.0000000001          4 accuracy binary     0.801    10 0.00792 Preprocess…
2    0.0000000178          4 accuracy binary     0.801    10 0.00792 Preprocess…
3    0.00000316            4 accuracy binary     0.801    10 0.00792 Preprocess…
4    0.000562              4 accuracy binary     0.801    10 0.00792 Preprocess…
5    0.0000000001          8 accuracy binary     0.781    10 0.00853 Preprocess…
```

On peut maintenant entraîner notre modèle avec les hyperparamètres optimisés. Nous allons

```
best_tree <- tune_res %>%
  select_best(metric = "accuracy")

final_tree <- finalize_model(tune_spec, best_tree)
```

Entraînement du modèle

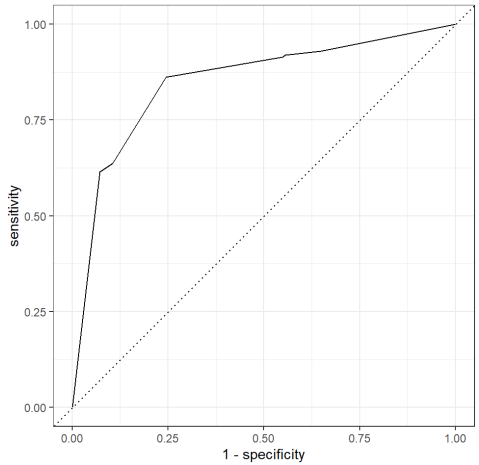
```
final_fit <- last_fit(final_tree,
                      class ~ .,
                      cell_split)
```

Performances :

```
final_fit %>%
  collect_metrics()
```

```
# A tibble: 3 × 4
  .metric     .estimator .estimate .config
  <chr>       <chr>          <dbl> <chr>
1 accuracy    binary         0.824
Preprocessor1_Model1
2 roc_auc     binary         0.848
Preprocessor1_Model1
3 brier_class binary         0.141
Preprocessor1_Model1
```

```
final_fit %>%
  collect_predictions() %>%
  roc_curve(class, .pred_PS) %>%
  autoplot()
```



Recettes et workflows

Feitures engineering avec `recipes`

`tidymodels` et son package `recipes` permettent de créer des recettes pour préparer les données. Cela permet de transformer certaines variables, de créer de nouvelles variables, etc.

Nous allons prendre ici comme exemple le jeu de données `nyflights13`.

Data prerocessing

Commencons par préparer et nettroyer nos données:

```r
set.seed(123)
library(nycflights13)

flight_data <-
  flights %>%
  mutate(
    # On change le retard en facteur
    arr_delay = ifelse(arr_delay >= 30, "late", "on_time"),
    arr_delay = factor(arr_delay),
    # Nous aurons besoin de la date
    date = lubridate::as_date(time_hour)
  ) %>%
  # On ajoute les données de météo
  inner_join(weather, by = c("origin", "time_hour")) %>%
  # On ne garde que les colonnes utiles
  select(dep_time, flight, origin, dest, air_time, distance,
         carrier, date, arr_delay, time_hour) %>%
  # On enlève les lignes avec des données manquantes
```

Nous allons chercher à prédire le retard en fonction de différentes variables

Data preprocessing

```
1 glimpse(flight_data)
```

```
Rows: 325,819
Columns: 10
$ dep_time  <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, 558, …
$ flight    <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 49, 71…
$ origin    <fct> EWR, LGA, JFK, JFK, LGA, EWR, EWR, LGA, JFK, LGA, JFK, JFK, …
$ dest      <fct> IAH, IAH, MIA, BQN, ATL, ORD, FLL, IAD, MCO, ORD, PBI, TPA, …
$ air_time  <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 158, 3…
$ distance  <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, 1028,…
$ carrier   <fct> UA, UA, AA, B6, DL, UA, B6, EV, B6, AA, B6, B6, UA, UA, AA, …
$ date      <date> 2013-01-01, 2013-01-01, 2013-01-01, 2013-01-01, 2013-01-01,…
$ arr_delay <fct> on_time, on_time, late, on_time, on_time, on_time, on_time, …
$ time_hour <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 05:00:…
```

Séparation des données

```
1 data_split <- initial_split(flight_data, prop = 3/4)
2
3 train_data <- training(data_split)
4 test_data  <- testing(data_split)
```

Recettes

Une recette de base prend une **formule** et des **données**

```
1 flights_rec <-
2   recipe(arr_delay ~ ., data = train_data)
```

Par défaut, toutes les variables sont des `predictors`.

```
1 summary(flights_rec)
```

```
# A tibble: 10 × 4
   variable  type      role      source
   <chr>     <list>    <chr>     <chr>
 1 dep_time  <chr [2]> predictor original
 2 flight    <chr [2]> predictor original
 3 origin    <chr [3]> predictor original
 4 dest      <chr [3]> predictor original
 5 air_time  <chr [2]> predictor original
 6 distance  <chr [2]> predictor original
 7 carrier   <chr [3]> predictor original
 8 date      <chr [1]> predictor original
 9 time_hour <chr [1]> predictor original
10 arr_delay <chr [3]> outcome   original
```

Recettes

Nous pouvons maintenant appliquer des transformations sur les colonnes pour créer des variables adaptées à notre modèle.

Il existe de nombreuses recettes dans le package `recipes`. Elles ont toute la forme `step_X()` où `X` sera le nom d'une recette.

Par exemple, la recette `step_date` permet de transformer une date en une autre variable (jour de la semaine, mois, années, …)

```
flights_rec <-
  recipe(arr_delay ~ ., data = train_data) %>%
  step_date(date, features = c("dow", "month"))

prep(flights_rec) %>% juice() %>% select(starts_with("date")) # Pour voir le résultat
```

```
# A tibble: 244,364 × 3
   date       date_dow date_month
   <date>     <fct>    <fct>
 1 2013-05-03 Fri      May
 2 2013-03-04 Mon      Mar
 3 2013-02-20 Wed      Feb
 4 2013-04-02 Tue      Apr
 5 2013-06-13 Thu      Jun
 6 2013-02-21 Thu      Feb
```

```
 7 2013-05-08 Wed      May
 8 2013-10-21 Mon      Oct
 9 2013-11-12 Tue      Nov
10 2013-09-11 Wed      Sep
# ℹ 244,354 more rows
```

Recettes

Nous allons transformer les variables de différentes manières:

- `update_role`: Pour indiquer les variables que l'on veut garder comme identifiant (`ID`)
- `step_date`: Pour transformer la date en variables plus utiles
- `step_holiday`: Pour indiquer si une date tombe un jour férié
- `step_dummy`: Pour transformer les variables catégorielles en variables binaires

```
flights_rec <-
  recipe(arr_delay ~ ., data = train_data) %>%
  update_role(flight, time_hour, new_role = "ID") %>%
  step_date(date, features = c("dow", "month")) %>%
  step_holiday(date,
               holidays = timeDate::listHolidays("US"),
               keep_original_cols = FALSE) %>%
  step_dummy(all_nominal_predictors())
```

Faire un `workflow`

Un `workflow` est une combinaison d'une recette et d'un modèle. Il permet d'enchainer différentes opérations.

Commençons par définir un modèle:

```
lr_mod <-
  logistic_reg() %>%
  set_engine("glm")
```

workflow

Nous pouvons maintenant combiner notre recette et notre modèle pour créer un `workflow`.

```
flights_wf <-
  workflow() %>%
  add_recipe(flights_rec) %>%
  add_model(lr_mod)
```

Entrainer un `workflow`

Nous pouvons ensuite entraîner notre modèle. Pour cela, nous pouvons directement utiliser la fonction `fit` du `workflow`.

```
flights_fit <-
  flights_wf %>%
  fit(data = train_data)
```

Rééchantillonnage avec un `workflow`

Nous pouvons également fiter notre modèle avec une validation croisée.

```
flights_folds <- vfold_cv(train_data, v=10)

flights_res <-
  flights_wf %>%
  fit_resamples(resamples = flights_folds)
```

Évaluation du modèle

Comme précédemment, nous pouvons directement accéder aux résultats du modèle

```
1 flights_res %>%
2   collect_metrics()
```

```
# A tibble: 3 × 6
  .metric     .estimator  mean     n  std_err .config
  <chr>       <chr>      <dbl> <int>    <dbl> <chr>
1 accuracy    binary     0.849    10 0.000521 Preprocessor1_Model1
2 brier_class binary     0.114    10 0.000359 Preprocessor1_Model1
3 roc_auc     binary     0.766    10 0.00111  Preprocessor1_Model1
```

Hyperparamètres et `workflow`

Nous pouvons également utiliser les fonctions pour optimiser les hyperparamètres directement avec un workflow.

Précédemment, nous avions:

```
tune_res <- tune_grid(tune_spec,
                      class ~ .,          # On indique la formule
                      resamples = folds,
                      grid = tree_grid)

best_tree <- tune_res %>%
  select_best(metric = "accuracy")

final_tree <- finalize_model(tune_spec, best_tree)

final_fit <- last_fit(final_tree,
                      class ~ .,
                      cell_split)
```

Hyperparamètres et `workflow`

Avec un `workflow`, cela devient:

```
tree_wf <- workflow() %>%
  add_model(tune_spec) %>%
  add_formula(class ~ .) %>%

best_tree <-
  tree_wf %>%
  tune_grid(
    resamples = cell_folds,
    grid = tree_grid
    ) %>%
  select_best(metric = "accuracy")

final_wf <-
  tree_wf %>%
  finalize_workflow(best_tree)

final_fit <-
  final_wf %>%
  last_fit(cell_split)
```

Références