# 12 Gestion des cadres de données avec le `dplyr` package

Regarder une vidéo de ce chapitre

## 12.1 Cadres de données

La *trame de données* est une structure de données clé en statistiques et en R. La structure de base d'une trame de données consiste en une observation par ligne et chaque colonne représente une variable, une mesure, une caractéristique ou une caractéristique de cette observation. R a une implémentation interne de trames de données qui est probablement celle que vous utiliserez le plus souvent. Cependant, il existe des packages sur CRAN qui implémentent des trames de données via des éléments tels que des bases de données relationnelles qui vous permettent d'opérer sur des trames de données très volumineuses (mais nous n'en discuterons pas ici).

Étant donné l'importance de la gestion des blocs de données, il est important de disposer de bons outils pour les traiter. Dans les chapitres précédents, nous avons déjà abordé certains outils tels que la `subset()` fonction et l'utilisation des opérateurs `[` et `$` pour extraire des sous-ensembles de trames de données. Cependant, d'autres opérations, telles que le filtrage, la réorganisation et la réduction, peuvent souvent être fastidieuses en R dont la syntaxe n'est pas très intuitive. Le `dplyr` progiciel est conçu pour atténuer bon nombre de ces problèmes et pour fournir un ensemble hautement optimisé de routines spécialement conçues pour traiter les trames de données.

## 12.2 Le `dplyr` paquet

Le `dplyr` paquet a été développé par Hadley Wickham de RStudio et est une version optimisée et distillée de son `plyr` paquet. Le `dplyr` paquet ne fournit aucune «nouvelle» fonctionnalité à R en soi, dans la mesure où tout `dplyr` pourrait déjà être fait avec la base R, mais il simplifie *grandement* les fonctionnalités existantes dans R.

L'un des apports importants du `dplyr` progiciel est qu'il fournit une «grammaire» (en particulier des verbes) pour la manipulation de données et l'utilisation de trames de données. Avec cette grammaire, vous pouvez communiquer de manière sensée ce que vous faites à un bloc de données que d'autres personnes peuvent comprendre (en supposant qu'ils connaissent également la grammaire). Ceci est utile car il fournit une abstraction pour la manipulation de données qui n'existait pas auparavant. Une autre contribution utile est que les `dplyr` fonctions sont **très** rapides, car de nombreuses opérations clés sont codées en C ++.

## 12.3 `dplyr` Grammaire

Certains des «verbes» clés fournis par le `dplyr` paquet sont

- `select` : retourne un sous-ensemble des colonnes d'un cadre de données, en utilisant une notation flexible
- `filter` : extraire un sous-ensemble de lignes d'une trame de données en fonction de conditions logiques
- `arrange` : réorganiser les lignes d'un cadre de données
- `rename` : renommer des variables dans un cadre de données
- `mutate` : ajouter de nouvelles variables / colonnes ou transformer des variables existantes
- `summarise` / `summarize` : générer des statistiques récapitulatives sur les différentes variables du bloc de données, éventuellement dans les strates
- `%>%` : l'opérateur "pipe" est utilisé pour connecter plusieurs actions de verbe ensemble dans un pipeline

Le `dplyr` package est un nombre de ses propres types de données dont il tire parti. Par exemple, il existe une `print` méthode pratique qui vous empêche d'imprimer beaucoup de données sur la console. La plupart du temps, ces types de données supplémentaires sont transparents pour l'utilisateur et ne doivent pas être inquiétés.

### 12.3.1 `dplyr` Propriétés des fonctions communes

Toutes les fonctions que nous aborderons dans ce chapitre auront quelques caractéristiques communes. En particulier,

1. Le premier argument est un cadre de données.
2. Les arguments suivants décrivent quoi faire avec le bloc de données spécifié dans le premier argument, et vous pouvez faire référence aux colonnes du bloc de données directement sans utiliser l'opérateur $ (utilisez uniquement les noms de colonne).
3. Le résultat renvoyé par une fonction est un nouveau bloc de données.
4. Les trames de données doivent être correctement formatées et annotées pour que tout soit utile. En particulier, les données doivent être bien rangées . En bref, il devrait y avoir une observation par rangée et chaque colonne devrait représenter une caractéristique ou des caractéristiques de cette observation.

## 12.4 Installer le `dplyr` paquet

Le `dplyr` paquet peut être installé à partir de CRAN ou de GitHub en utilisant le `devtools` paquet et la `install_github()` fonction. Le référentiel GitHub contiendra généralement les dernières mises à jour du paquet et de la version de développement.

Pour installer à partir de CRAN, lancez simplement

```
> install.packages("dplyr")
```

Pour installer depuis GitHub, vous pouvez exécuter

```
> install_github("hadley/dplyr")
```

Après avoir installé le paquet, il est important de le charger dans votre session R avec la `library()` fonction.

```
> library(dplyr)

Attaching package: 'dplyr'
The following objects are masked from 'package:stats':

    filter, lag
The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

Vous pouvez recevoir des avertissements lors du chargement du package, car certaines fonctions du `dplyr` package portent le même nom que les fonctions des autres packages. Pour l'instant, vous pouvez ignorer les avertissements.

## 12.5 `select()`

Pour les exemples de ce chapitre, nous utiliserons un jeu de données contenant des données sur la pollution de l'air et la température de la ville de Chicago aux États-Unis. Ce jeu de données est disponible sur mon site Web.

Après avoir décompressé l'archive, vous pouvez charger les données dans R à l'aide de la `readRDS()` fonction.

```
> chicago <- readRDS("chicago.rds")
```

Vous pouvez voir certaines caractéristiques de base du jeu de données avec les fonctions `dim()` et `str()` .

```
> dim(chicago)
[1] 6940    8
> str(chicago)
'data.frame':   6940 obs. of  8 variables:
 $ city      : chr  "chic" "chic" "chic" "chic" ...
 $ tmpd      : num  31.5 33 33 29 32 40 34.5 29 26.5 32.5 ...
 $ dptp      : num  31.5 29.9 27.4 28.6 28.9 ...
 $ date      : Date, format: "1987-01-01" "1987-01-02" ...
 $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
 $ pm10tmean2: num  34 NA 34.2 47 NA ...
 $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
 $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

La `select()` fonction peut être utilisée pour sélectionner les colonnes d'un bloc de données sur lequel vous souhaitez vous concentrer. Souvent, vous avez un grand bloc de données contenant «toutes» les données, mais toute analyse *donnée* peut n'utiliser qu'un sous-ensemble de variables ou d'observations. La `select()` fonction vous permet d'obtenir les quelques colonnes dont vous pourriez avoir besoin.

Supposons que nous voulions prendre uniquement les 3 premières colonnes. Il y a quelques façons de le faire. On pourrait par exemple utiliser des indices numériques. Mais nous pouvons aussi utiliser les noms directement.

```
> names(chicago)[1:3]
[1] "city" "tmpd" "dptp"
> subset <- select(chicago, city:dptp)
> head(subset)
  city tmpd   dptp
1 chic 31.5 31.500
2 chic 33.0 29.875
3 chic 33.0 27.375
4 chic 29.0 28.625
5 chic 32.0 28.875
6 chic 40.0 35.125
```

Notez que le `:` paramètre ne peut normalement pas être utilisé avec des noms ou des chaînes, mais `select()` vous pouvez l'utiliser dans la fonction pour spécifier une plage de noms de variables.

Vous pouvez également *omettre des* variables à l'aide de la `select()` fonction en utilisant le signe négatif. Avec `select()` vous pouvez faire

```
> select(chicago, -(city:dptp))
```

ce qui indique que nous devrions inclure toutes les variables *sauf* les variables à `city` travers `dptp` . Le code équivalent en base R serait

```
> i <- match("city", names(chicago))
> j <- match("dptp", names(chicago))
> head(chicago[, -(i:j)])
```

Pas super intuitif, non?

La `select()` fonction permet également une syntaxe spéciale vous permettant de spécifier des noms de variables basés sur des modèles. Ainsi, par exemple, si vous voulez conserver chaque variable qui se termine par un «2», nous pourrions faire

```
> subset <- select(chicago, ends_with("2"))
> str(subset)
'data.frame':   6940 obs. of  4 variables:
 $ pm25tmean2: num  NA NA NA NA NA NA NA NA NA NA ...
 $ pm10tmean2: num  34 NA 34.2 47 NA ...
 $ o3tmean2  : num  4.25 3.3 3.33 4.38 4.75 ...
 $ no2tmean2 : num  20 23.2 23.8 30.4 30.3 ...
```

Ou si nous voulions garder chaque variable qui commence par un «d», nous pourrions faire

```
> subset <- select(chicago, starts_with("d"))
> str(subset)
'data.frame':   6940 obs. of  2 variables:
 $ dptp: num  31.5 29.9 27.4 28.6 28.9 ...
 $ date: Date, format: "1987-01-01" "1987-01-02" ...
```

Vous pouvez également utiliser des expressions régulières plus générales si nécessaire. Voir la page d'aide ( `?select` ) pour plus de détails.

# 12.6 `filter()`

La `filter()` fonction est utilisée pour extraire des sous-ensembles de lignes d'un cadre de données. Cette fonction est similaire à la `subset()` fonction existante dans R mais est un peu plus rapide dans mon expérience.

Supposons que nous voulions extraire les lignes du `chicago` cadre de données où les niveaux de PM2.5 sont supérieurs à 30 (ce qui est un niveau raisonnablement élevé), nous pourrions le faire.

```
> chic.f <- filter(chicago, pm25tmean2 > 30)
> str(chic.f)
'data.frame':   194 obs. of  8 variables:
 $ city      : chr  "chic" "chic" "chic" "chic" ...
 $ tmpd      : num  23 28 55 59 57 57 75 61 73 78 ...
 $ dptp      : num  21.9 25.8 51.3 53.7 52 56 65.8 59 60.3 67.1 ...
 $ date      : Date, format: "1998-01-17" "1998-01-23" ...
 $ pm25tmean2: num  38.1 34 39.4 35.4 33.3 ...
 $ pm10tmean2: num  32.5 38.7 34 28.5 35 ...
 $ o3tmean2  : num  3.18 1.75 10.79 14.3 20.66 ...
 $ no2tmean2 : num  25.3 29.4 25.3 31.4 26.8 ...
```

Vous pouvez voir qu'il n'y a plus que 194 lignes dans le cadre de données et que la distribution des `pm25tmean2` valeurs est.

```
> summary(chic.f$pm25tmean2)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  30.05   32.12   35.04   36.63   39.53   61.50
```

Nous pouvons placer une séquence logique arbitrairement complexe à l'intérieur de `filter()` , de manière à extraire par exemple les lignes où PM2.5 est supérieur à 30 *et la* température supérieure à 80 degrés Fahrenheit.

```
> chic.f <- filter(chicago, pm25tmean2 > 30 & tmpd > 80)
> select(chic.f, date, tmpd, pm25tmean2)
         date tmpd pm25tmean2
1  1998-08-23   81   39.60000
2  1998-09-06   81   31.50000
3  2001-07-20   82   32.30000
4  2001-08-01   84   43.70000
5  2001-08-08   85   38.83750
6  2001-08-09   84   38.20000
7  2002-06-20   82   33.00000
8  2002-06-23   82   42.50000
9  2002-07-08   81   33.10000
10 2002-07-18   82   38.85000
11 2003-06-25   82   33.90000
12 2003-07-04   84   32.90000
13 2005-06-24   86   31.85714
14 2005-06-27   82   51.53750
15 2005-06-28   85   31.20000
16 2005-07-17   84   32.70000
17 2005-08-03   84   37.90000
```

Il n'ya plus que 17 observations où ces deux conditions sont remplies.

## 12.7 `arrange()`

La `arrange()` fonction permet de réorganiser les lignes d'un bloc de données en fonction de l'une des variables / colonnes. Réorganiser les lignes d'un cadre de données (tout en préservant l'ordre correspondant des autres colonnes) est normalement une tâche ardue dans R. La `arrange()` fonction simplifie un peu le processus.

Ici, nous pouvons classer les lignes du bloc de données par date, de sorte que la première ligne représente l'observation la plus ancienne (la plus ancienne) et que la dernière ligne représente la dernière observation (la plus récente).

```
> chicago <- arrange(chicago, date)
```

Nous pouvons maintenant vérifier les premières lignes

```
> head(select(chicago, date, pm25tmean2), 3)
        date pm25tmean2
1 1987-01-01         NA
2 1987-01-02         NA
3 1987-01-03         NA
```

et les derniers rangs.

```
> tail(select(chicago, date, pm25tmean2), 3)
           date pm25tmean2
6938 2005-12-29    7.45000
6939 2005-12-30   15.05714
6940 2005-12-31   15.00000
```

Les colonnes peuvent également être classées par ordre décroissant en utilisant l' `desc()` opérateur spécial .

```
> chicago <- arrange(chicago, desc(date))
```

En regardant les trois premières et les trois dernières lignes, les dates sont classées par ordre décroissant.

```
> head(select(chicago, date, pm25tmean2), 3)
        date pm25tmean2
1 2005-12-31   15.00000
2 2005-12-30   15.05714
3 2005-12-29    7.45000
> tail(select(chicago, date, pm25tmean2), 3)
           date pm25tmean2
6938 1987-01-03         NA
6939 1987-01-02         NA
6940 1987-01-01         NA
```

## 12.8 `rename()`

Renommer une variable dans un bloc de données dans R est étonnamment difficile à faire! La `rename()` fonction est conçue pour faciliter ce processus.

Ici, vous pouvez voir les noms des cinq premières variables du `chicago` bloc de données.

```
> head(chicago[, 1:5], 3)
  city tmpd dptp       date pm25tmean2
1 chic   35 30.1 2005-12-31   15.00000
2 chic   36 31.0 2005-12-30   15.05714
3 chic   35 29.4 2005-12-29    7.45000
```

La `dptp` colonne est supposée représenter la température du point de rosée et `pm25tmean2` fournit les données PM2.5. Cependant, ces noms sont assez obscurs ou maladroits et devraient probablement être renommés en quelque chose de plus sensé.

```
> chicago <- rename(chicago, dewpoint = dptp, pm25 = pm25tmean2)
> head(chicago[, 1:5], 3)
  city tmpd dewpoint       date     pm25
1 chic   35     30.1 2005-12-31 15.00000
2 chic   36     31.0 2005-12-30 15.05714
3 chic   35     29.4 2005-12-29  7.45000
```

La syntaxe à l'intérieur de la `rename()` fonction est d'avoir le nouveau nom à gauche du `=` signe et l'ancien nom à droite.

Je le laisse comme exercice au lecteur pour comprendre comment vous faites cela en base R sans `dplyr` .

## 12.9 `mutate()`

La `mutate()` fonction existe pour calculer les transformations de variables dans un cadre de données. Souvent, vous souhaitez créer de nouvelles variables dérivées de variables existantes et `mutate()` fournir une interface propre pour le faire.

Par exemple, avec les données sur la pollution atmosphérique, nous souhaitons souvent *compromettre* les données en soustrayant la moyenne de celles-ci. De cette façon, nous pouvons déterminer si le niveau de pollution atmosphérique d'un jour donné est supérieur ou inférieur à la moyenne (au lieu de regarder son niveau absolu).

Ici, nous créons une `pm25detrend` variable qui soustrait la moyenne de la `pm25` variable.

```
> chicago <- mutate(chicago, pm25detrend = pm25 - mean(pm25, na.rm = TRUE))
> head(chicago)
  city tmpd dewpoint       date     pm25 pm10tmean2  o3tmean2 no2tmean2
1 chic   35     30.1 2005-12-31 15.00000       23.5  2.531250  13.25000
2 chic   36     31.0 2005-12-30 15.05714       19.2  3.034420  22.80556
3 chic   35     29.4 2005-12-29  7.45000       23.5  6.794837  19.97222
4 chic   37     34.5 2005-12-28 17.75000       27.5  3.260417  19.28563
5 chic   40     33.6 2005-12-27 23.56000       27.0  4.468750  23.50000
6 chic   35     29.6 2005-12-26  8.40000        8.5 14.041667  16.81944
  pm25detrend
1   -1.230958
2   -1.173815
3   -8.780958
4    1.519042
5    7.329042
6   -7.830958
```

Il y a aussi la `transmute()` fonction associée , qui fait la même chose que, `mutate()` mais *supprime* ensuite *toutes les variables non transformées* .

Nous détruisons ici les variables PM10 et ozone (O3).

```
> head(transmute(chicago,
+                pm10detrend = pm10tmean2 - mean(pm10tmean2, na.rm = TRUE),
+                o3detrend = o3tmean2 - mean(o3tmean2, na.rm = TRUE)))
  pm10detrend  o3detrend
1  -10.395206 -16.904263
2  -14.695206 -16.401093
3  -10.395206 -12.640676
4   -6.395206 -16.175096
5   -6.895206 -14.966763
6  -25.395206  -5.393846
```

Notez qu'il n'y a que deux colonnes dans la trame de données transmutée.

# 12.10 group_by()

La `group_by()` fonction est utilisée pour générer des statistiques récapitulatives à partir du cadre de données au sein de strates définies par une variable. Par exemple, dans cet ensemble de données sur la pollution atmosphérique, vous voudrez peut-être connaître le niveau moyen annuel de PM2,5. La strate est donc l'année et c'est quelque chose que nous pouvons déduire de la `date` variable. En conjonction avec la `group_by()` fonction, nous utilisons souvent la `summarize()` fonction (ou `summarise()` pour certaines parties du monde).

L'opération générale consiste ici à scinder un bloc de données en éléments distincts définis par une variable ou un groupe de variables ( `group_by()` ), puis à appliquer une fonction de synthèse à ces sous-ensembles ( `summarize()` ).

Premièrement, nous pouvons créer une `year` variable avec `as.POSIXlt()` .

```
> chicago <- mutate(chicago, year = as.POSIXlt(date)$year + 1900)
```

Nous pouvons maintenant créer un bloc de données séparé qui scinde le bloc de données d'origine par année.

```
> years <- group_by(chicago, year)
```

Enfin, nous calculons des statistiques récapitulatives pour chaque année dans le bloc de données avec la `summarize()` fonction.

```
> summarize(years, pm25 = mean(pm25, na.rm = TRUE),
+           o3 = max(o3tmean2, na.rm = TRUE),
+           no2 = median(no2tmean2, na.rm = TRUE))
# A tibble: 19 x 4
    year  pm25    o3   no2
   <dbl> <dbl> <dbl> <dbl>
 1  1987 NaN    63.0  23.5
 2  1988 NaN    61.7  24.5
 3  1989 NaN    59.7  26.1
 4  1990 NaN    52.2  22.6
 5  1991 NaN    63.1  21.4
 6  1992 NaN    50.8  24.8
 7  1993 NaN    44.3  25.8
 8  1994 NaN    52.2  28.5
 9  1995 NaN    66.6  27.3
10  1996 NaN    58.4  26.4
11  1997 NaN    56.5  25.5
12  1998  18.3  50.7  24.6
13  1999  18.5  57.5  24.7
14  2000  16.9  55.8  23.5
15  2001  16.9  51.8  25.1
16  2002  15.3  54.9  22.7
17  2003  15.2  56.2  24.6
18  2004  14.6  44.5  23.4
19  2005  16.2  58.8  22.6
```

`summarize()` renvoie une trame de données avec `year` comme première colonne, puis les moyennes annuelles de `pm25` , `o3` et `no2` .

Dans un exemple un peu plus compliqué, nous voudrions peut-être savoir quels sont les niveaux moyens d'ozone ( `o3` ) et de dioxyde d'azote ( `no2` ) dans les quintiles de `pm25` . Une méthode plus simple consiste à utiliser un modèle de régression, mais nous pouvons le faire rapidement avec `group_by()` et `summarize()` .

Premièrement, nous pouvons créer une variable catégorique de `pm25` divisée en quintiles.

```
> qq <- quantile(chicago$pm25, seq(0, 1, 0.2), na.rm = TRUE)
> chicago <- mutate(chicago, pm25.quint = cut(pm25, qq))
```

Nous pouvons maintenant regrouper le bloc de données par la `pm25.quint` variable.

```
> quint <- group_by(chicago, pm25.quint)
Warning: Factor `pm25.quint` contains implicit NA, consider using
`forcats::fct_explicit_na`
```

Enfin, nous pouvons calculer la moyenne de `o3` et `no2` dans les quintiles de `pm25` .

```
> summarize(quint, o3 = mean(o3tmean2, na.rm = TRUE),
+           no2 = mean(no2tmean2, na.rm = TRUE))
# A tibble: 6 x 3
  pm25.quint     o3   no2
  <fct>       <dbl> <dbl>
1 (1.7,8.7]    21.7  18.0
2 (8.7,12.4]   20.4  22.1
3 (12.4,16.7]  20.7  24.4
4 (16.7,22.6]  19.9  27.3
5 (22.6,61.5]  20.3  29.6
6 <NA>         18.8  25.8
```

De la table, il semble qu'il n'y ait pas de relation forte entre `pm25` et `o3` , mais il semble y avoir une corrélation positive entre `pm25` et `no2` . Une modélisation statistique plus sophistiquée peut aider à fournir des réponses précises à ces questions, mais une simple application de `dplyr` fonctions peut souvent vous aider à atteindre vos objectifs .

## 12.11 %>%

L'opérateur de pipeline `%>%` est très pratique pour rassembler plusieurs `dplyr` fonctions dans une séquence d'opérations. Notez ci-dessus que chaque fois que nous voulions appliquer plusieurs fonctions, la séquence est enterrée dans une séquence d'appels de fonctions imbriquées difficiles à lire, c'est-à-dire

```
> third(second(first(x)))
```

Cette imbrication n'est pas une façon naturelle de penser à une séquence d'opérations. L' `%>%` opérateur vous permet d'enchaîner les opérations de gauche à droite, c.-à-d.

```
> first(x) %>% second %>% third
```

Prenons l'exemple que nous venons de faire dans la dernière section, où nous avons calculé la moyenne de `o3` et `no2` dans les quintiles de `pm25` . Là nous avons dû

1. créer une nouvelle variable `pm25.quint`
2. diviser le bloc de données par cette nouvelle variable
3. calculer la moyenne de `o3` et `no2` dans les sous-groupes définis par `pm25.quint`

Cela peut être fait avec la séquence suivante dans une seule expression de R.

```
> mutate(chicago, pm25.quint = cut(pm25, qq)) %>%
+         group_by(pm25.quint) %>%
+         summarize(o3 = mean(o3tmean2, na.rm = TRUE),
+                   no2 = mean(no2tmean2, na.rm = TRUE))
Warning: Factor `pm25.quint` contains implicit NA, consider using
`forcats::fct_explicit_na`
# A tibble: 6 x 3
  pm25.quint     o3   no2
  <fct>       <dbl> <dbl>
1 (1.7,8.7]    21.7  18.0
2 (8.7,12.4]   20.4  22.1
3 (12.4,16.7]  20.7  24.4
4 (16.7,22.6]  19.9  27.3
5 (22.6,61.5]  20.3  29.6
6 <NA>         18.8  25.8
```

De cette façon, nous n'avons pas besoin de créer un ensemble de variables temporaires en cours de route ou de créer une séquence imbriquée d'appels de fonction.

Notez dans le code ci-dessus que je passe la `chicago` trame de données au premier appel à `mutate()` , mais ensuite je ne suis pas obligé de passer le premier argument à `group_by()` ou `summarize()` . Une fois que vous avez parcouru le pipeline `%>%` , le premier argument est considéré comme la sortie de l'élément précédent du pipeline.

Un autre exemple pourrait être le calcul du niveau moyen de polluants par mois. Cela pourrait être utile pour voir s'il existe des tendances saisonnières dans les données.

```
> mutate(chicago, month = as.POSIXlt(date)$mon + 1) %>%
+         group_by(month) %>%
+         summarize(pm25 = mean(pm25, na.rm = TRUE),
+                   o3 = max(o3tmean2, na.rm = TRUE),
+                   no2 = median(no2tmean2, na.rm = TRUE))
# A tibble: 12 x 4
   month  pm25    o3   no2
   <dbl> <dbl> <dbl> <dbl>
 1     1  17.8  28.2  25.4
 2     2  20.4  37.4  26.8
 3     3  17.4  39.0  26.8
 4     4  13.9  47.9  25.0
 5     5  14.1  52.8  24.2
 6     6  15.9  66.6  25.0
 7     7  16.6  59.5  22.4
 8     8  16.9  54.0  23.0
 9     9  15.9  57.5  24.5
10    10  14.2  47.1  24.2
11    11  15.2  29.5  23.6
12    12  17.5  27.7  24.5
```

Nous pouvons voir ici qu'il a `o3` tendance à être faible en hiver et élevé en été alors qu'il `no2` est plus élevé en hiver et plus bas en été.

# 12.12 Résumé

Le `dplyr` paquet fournit un ensemble d'opérations concis pour la gestion des trames de données. Avec ces fonctions, nous pouvons effectuer plusieurs opérations complexes en seulement quelques lignes de code. En particulier, nous pouvons souvent entamer une analyse exploratoire avec la puissante combinaison de `group_by()` et `summarize()` .

Une fois que vous avez appris la `dplyr` grammaire, il y a quelques avantages supplémentaires

- `dplyr` peut fonctionner avec d'autres «backends» de trames de données, telles que des bases de données SQL. Il existe une interface SQL pour les bases de données relationnelles via le package DBI

- `dplyr` peut être intégré au `data.table` paquet pour les grandes tables rapides

Le `dplyr` paquet est un moyen pratique de simplifier et d'accélérer votre code de gestion de trames de données. Il est rare que vous obteniez une telle combinaison en même temps!