Eksploracja Tekstu i Wyszukiwanie Informacji w Mediach Społecznościowych

LABORATORIUM 1

- Pakiet ggplot2
- Potoki
- Pakiet dplyr

Pakiet ggplot

Ggplot2 jest biblioteką, pozwalającą na tworzenie eleganckich grafik i wykresów. Jest szczególnie przydatna w przypadku analizy danych wielowymiarowych oraz jako pomoc do data mining. Bibliotekę uruchamiamy komendą **library(ggplot2)** (w przypadku braku, należy zainstalować ją uzywając komendy **install.packages("ggplot2")**).

```
# PRZYKŁAD 1.1
library(ggplot2)
```

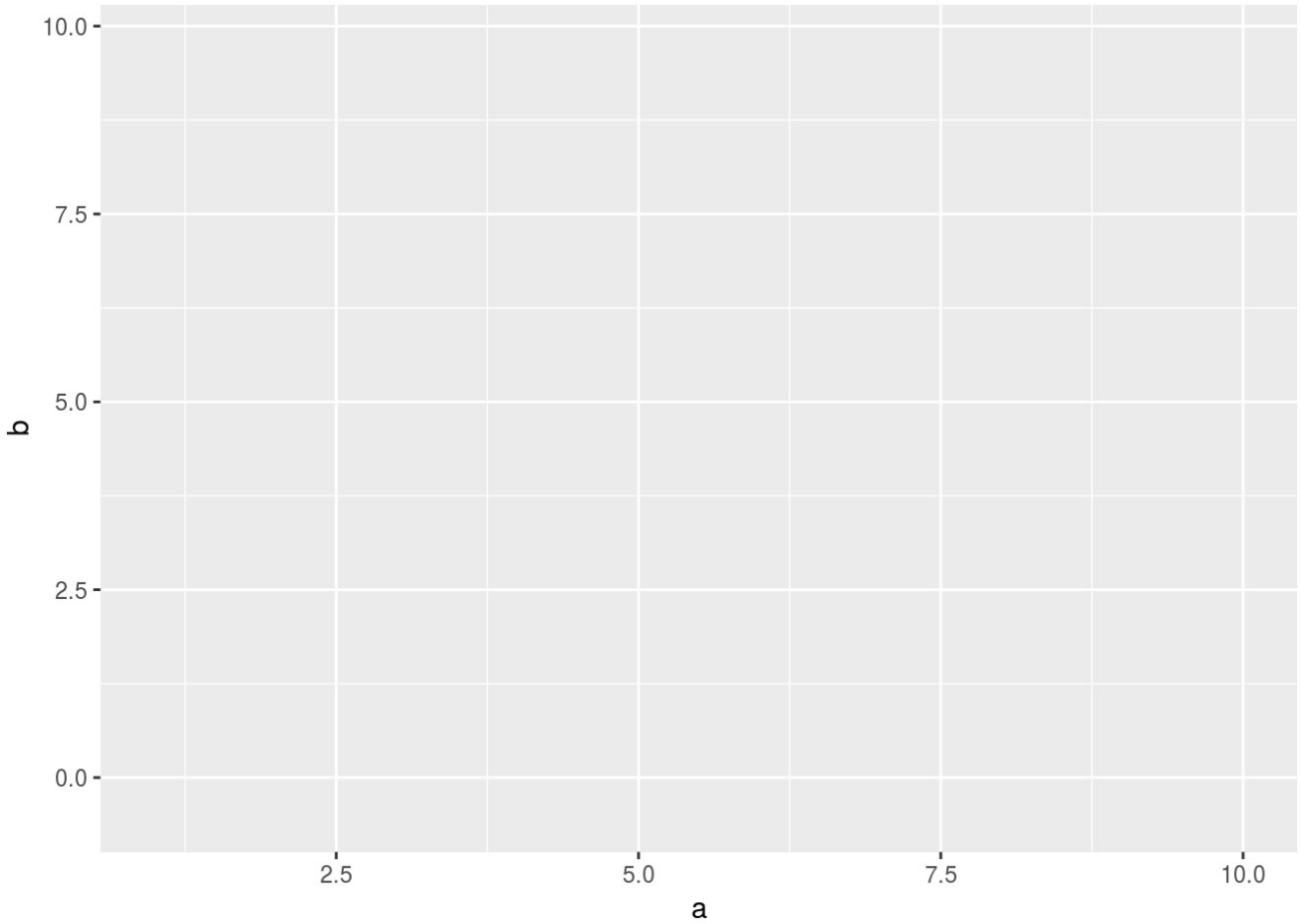
Podstawowe dwie cechy pakietu: *dane, które chcemy wyświetlić, muszą być typu dataframe,* wykresy składają się z warst, które dodajemy do bazowego obiektu.

Bazową funkcją, od której musimy zacząć tworzenie każdego wykresu jest **ggplot()**. Jako jej argument można podać od razu zbiór danych, który będziemy wykorzystywać, mapując poprzez funkcję **aes()** (aesthethics - właściwości) cechy. Samo wywołanie funkcji **ggplot()**, nawet z określeniem mapowania, nie da żadnego sensownego efektu.

```
# PRZYKŁAD 1.2
df1 <- data.frame(a = 1:10, b = (1:10) + runif(10,-2,2), c = 1:10)
df1
```

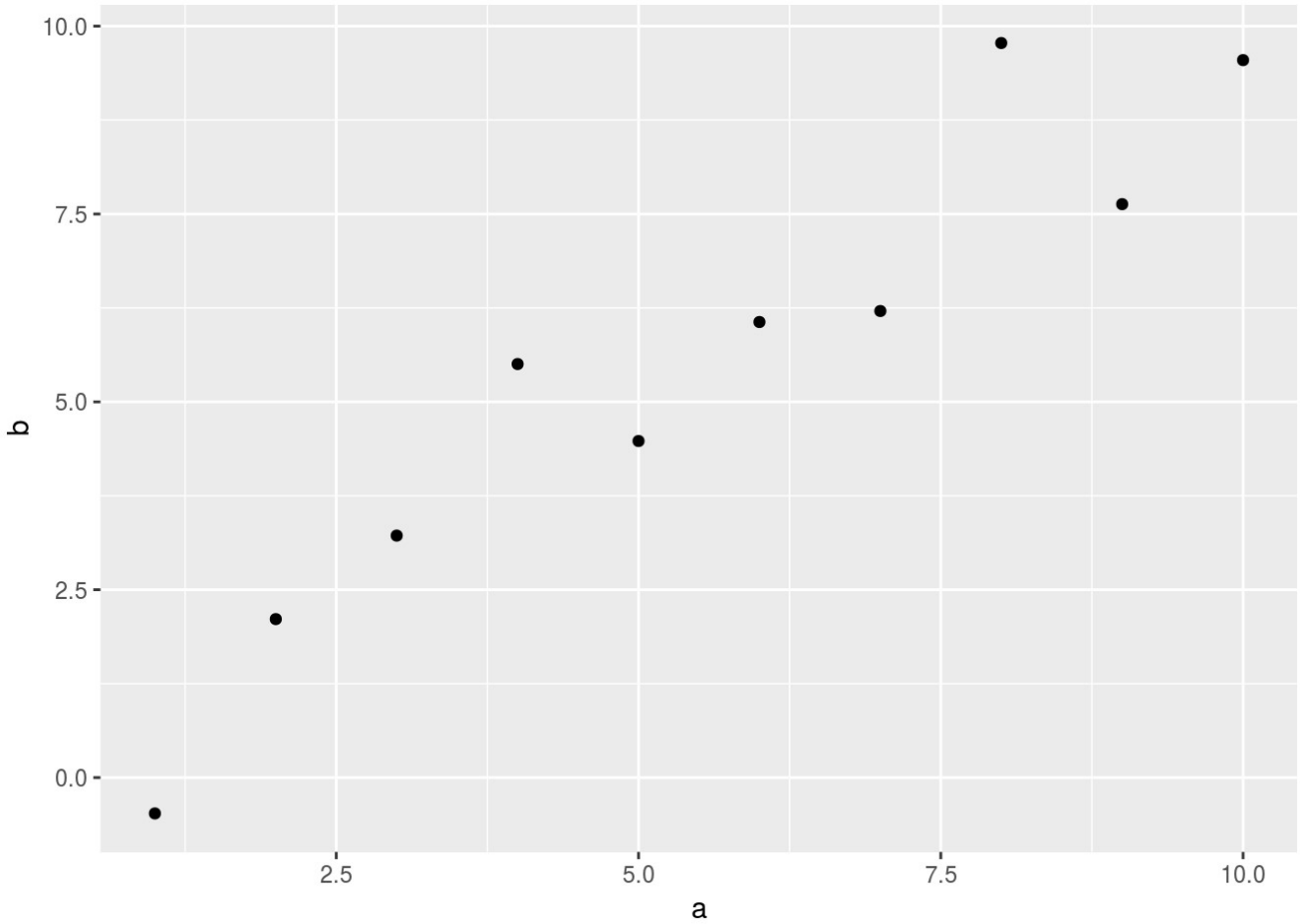
```
##     a          b  c
## 1   1 -0.4769591  1
## 2   2  2.1090795  2
## 3   3  3.2203672  3
## 4   4  5.5035895  4
## 5   5  4.4789019  5
## 6   6  6.0625283  6
## 7   7  6.2089541  7
## 8   8  9.7750896  8
## 9   9  7.6310188  9
## 10 10  9.5474727 10
```

```
ggplot(df1, aes(x = a, y = b))
```



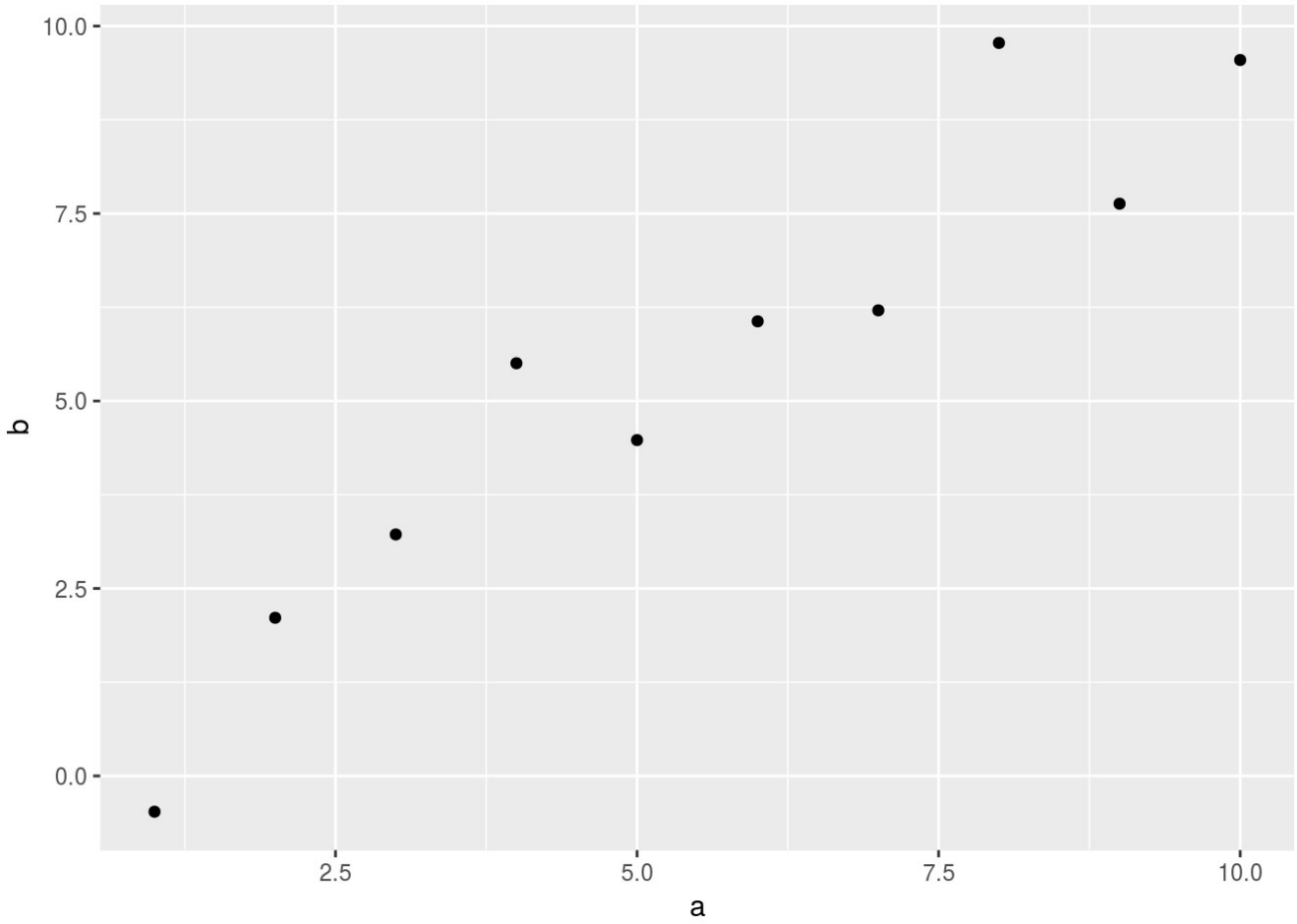
Dopiero ustalenie odpowiedniej warstwy (w tym przypadku chcemy narysować punkty za pomocą geometrii **geom_point()**) umożliwia otrzymanie wykresu.

```
# PRZYKŁAD 1.3
ggplot(df1, aes(x = a, y = b)) + geom_point()
```

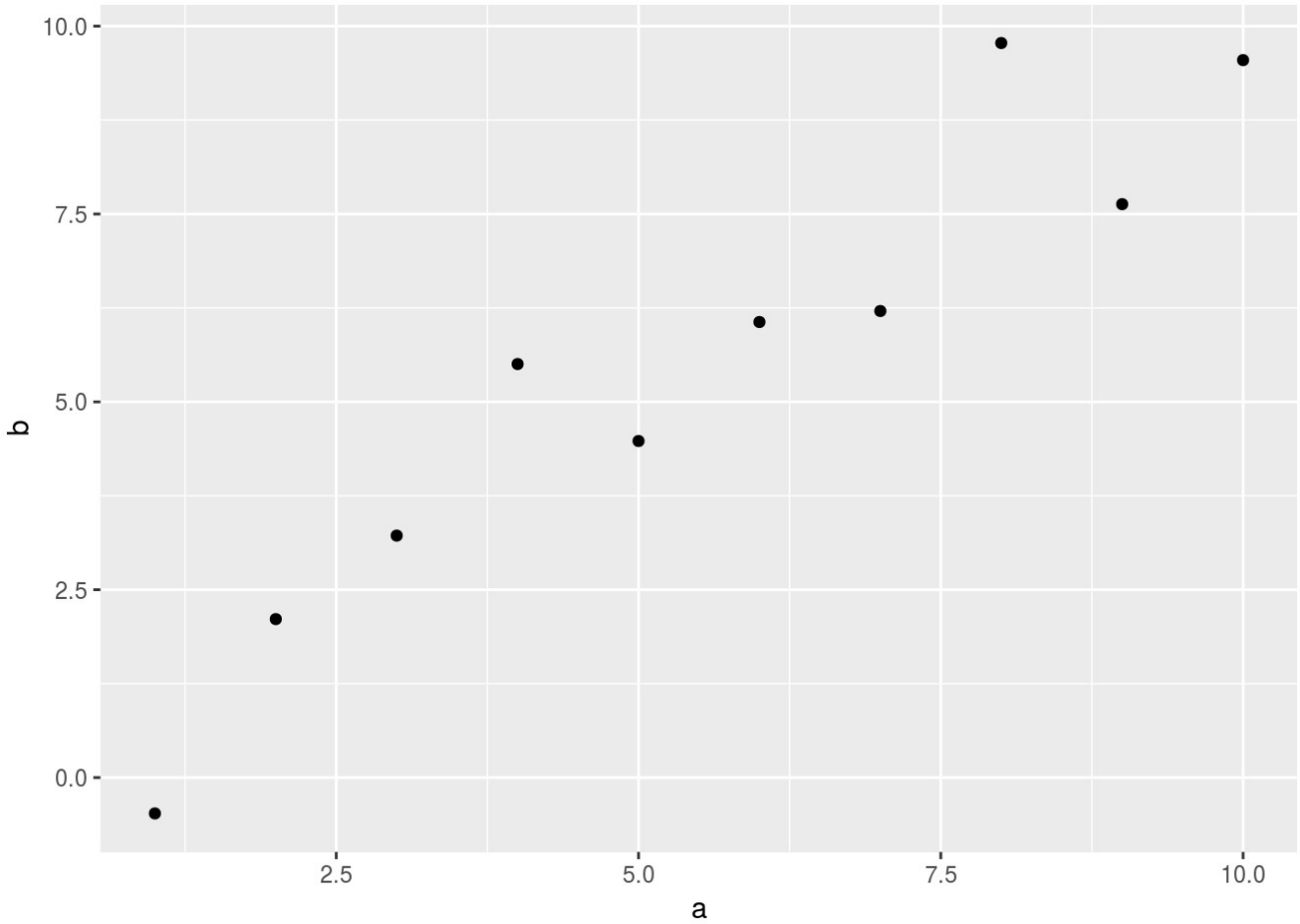


Należy przy tym zwrócić uwagę, że ten sam wykres może zostać otrzymany za pomocą różnych operacji

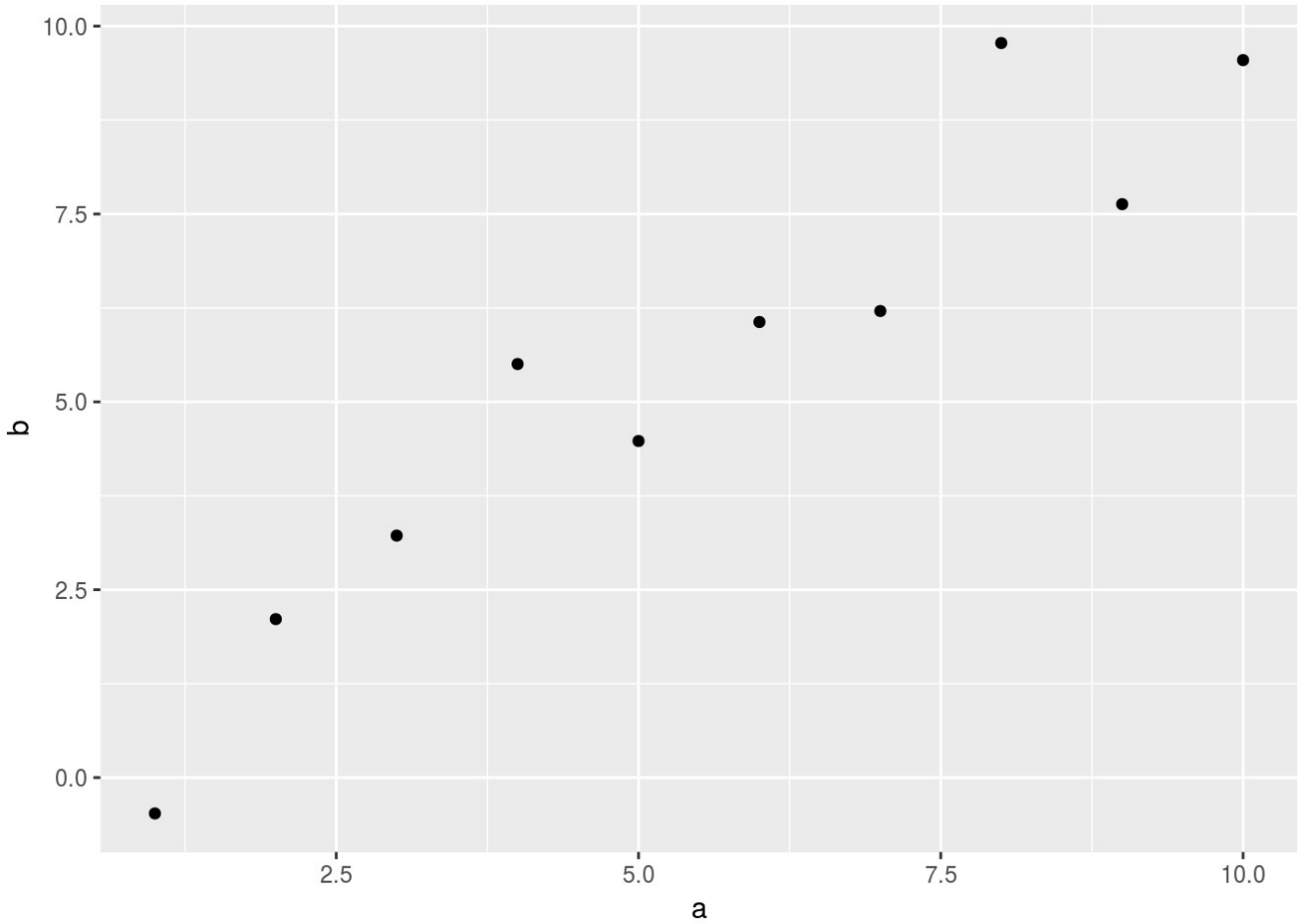
```
# PRZYKŁAD 1.4
ggplot(df1, aes(x = a, y = b)) + geom_point()
```



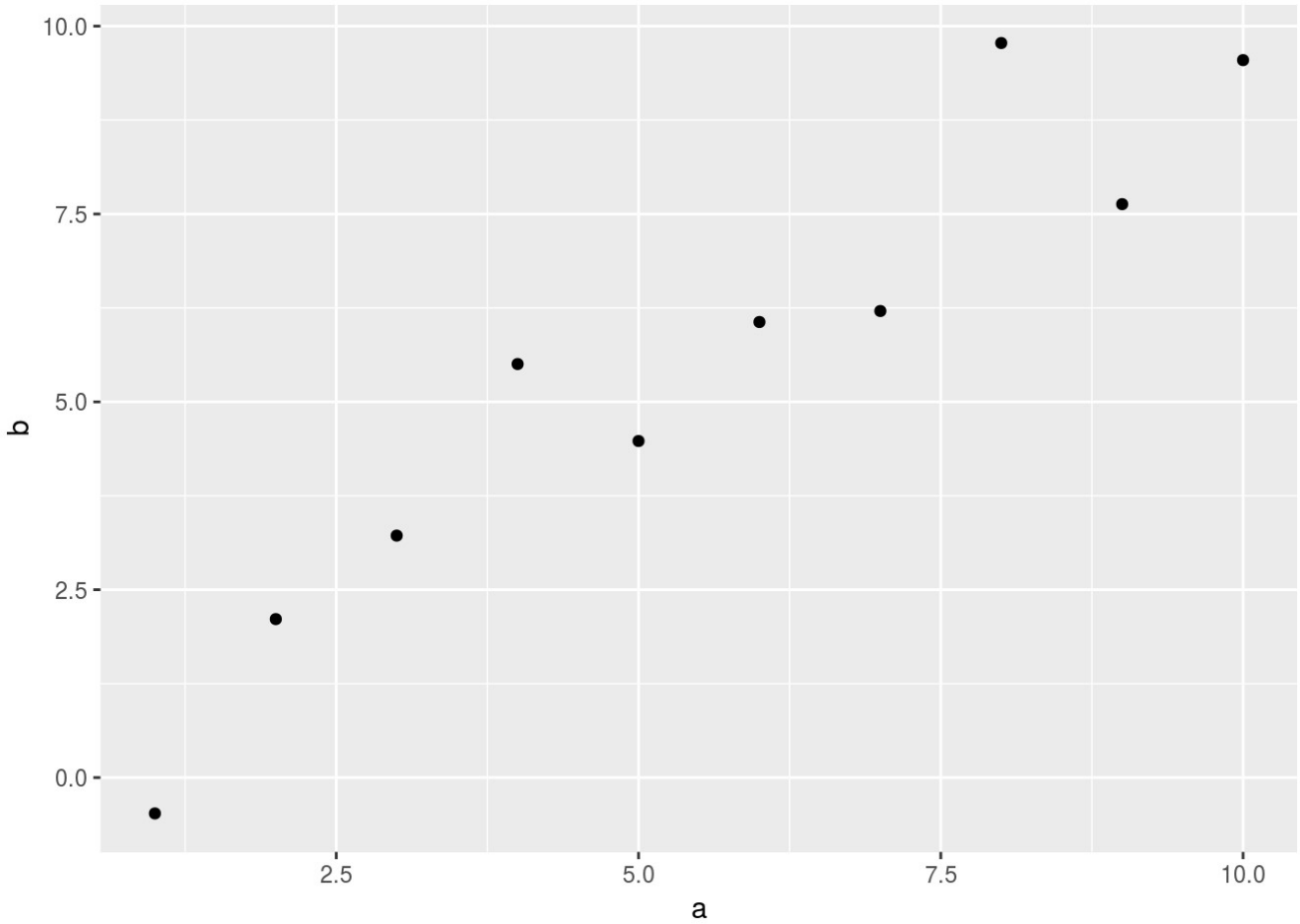

```
ggplot(df1) + geom_point(aes(x = a, y = b))
```




```
ggplot() + geom_point(data = df1, aes(x = a, y = b))
```

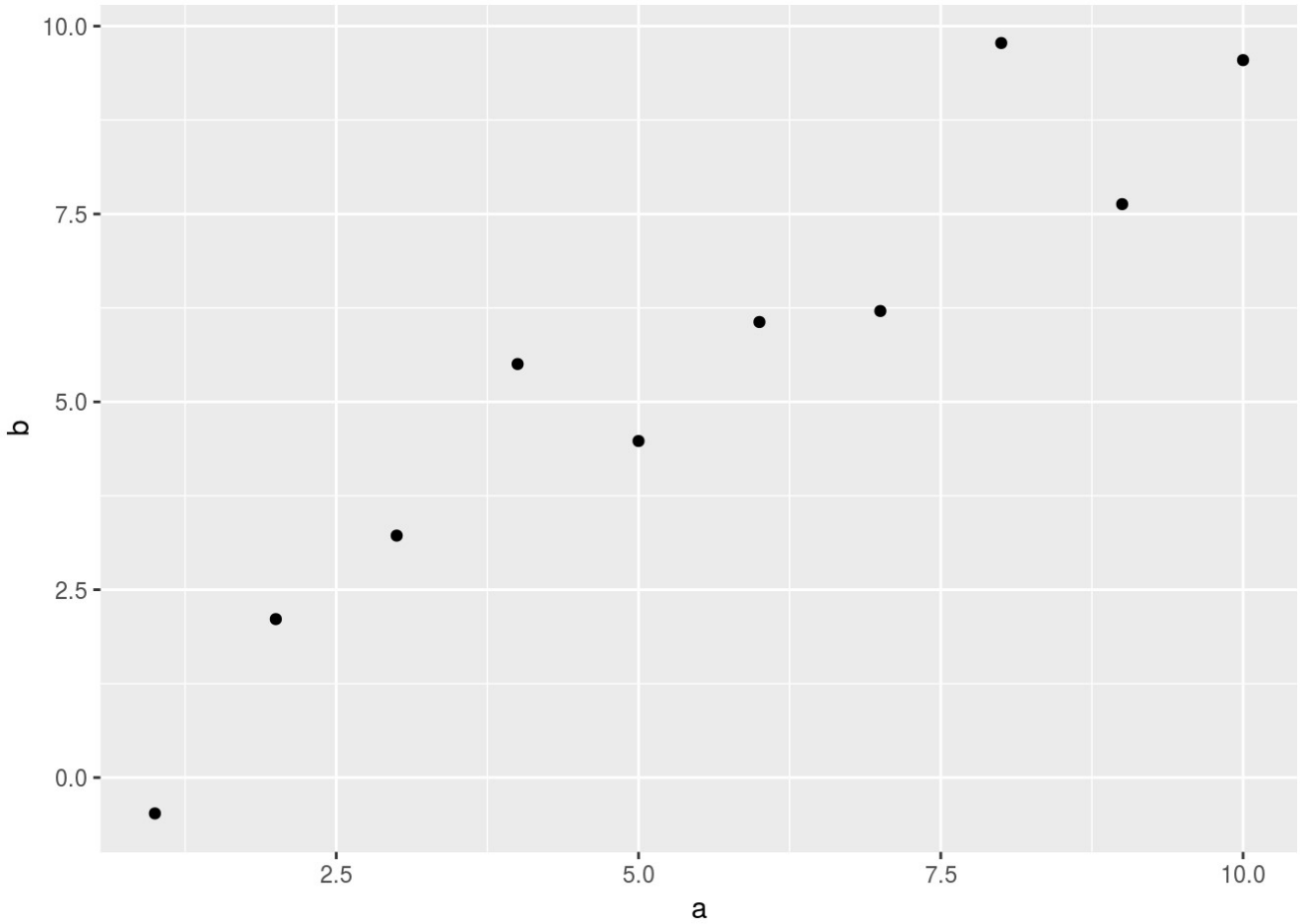



```
ggplot(df1, aes(x = a)) + geom_point(aes(y = b))
```



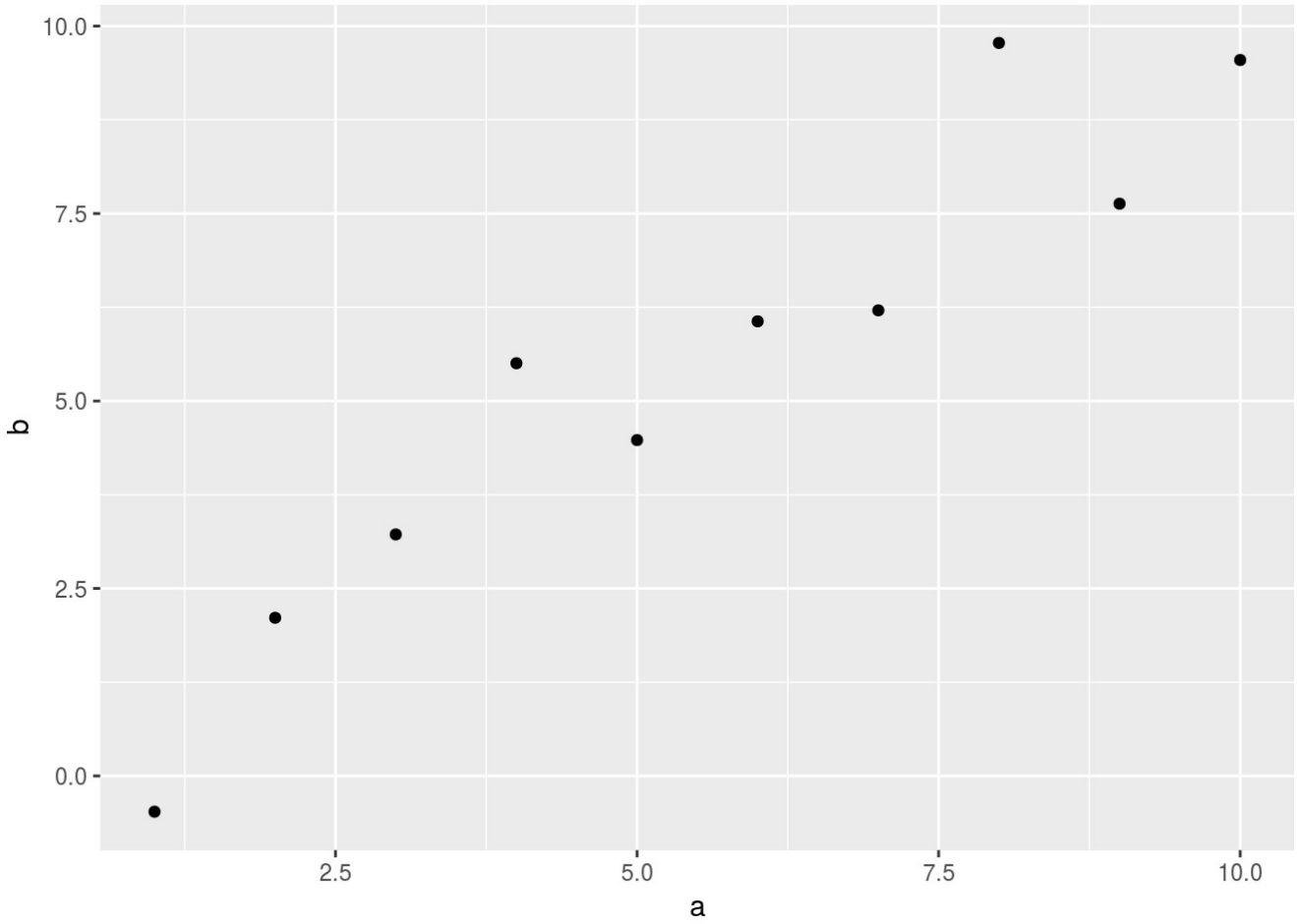
Poza tym można (a nawet jest dość polecane dla czystości kodu) umieścić bazową warstwę w zmiennej i nastepnie dodawać do niej kolejne warstwy.

```
# PRZYKŁAD 1.5
g <- ggplot(df1, aes(x = a, y = b))
g + geom_point()
```

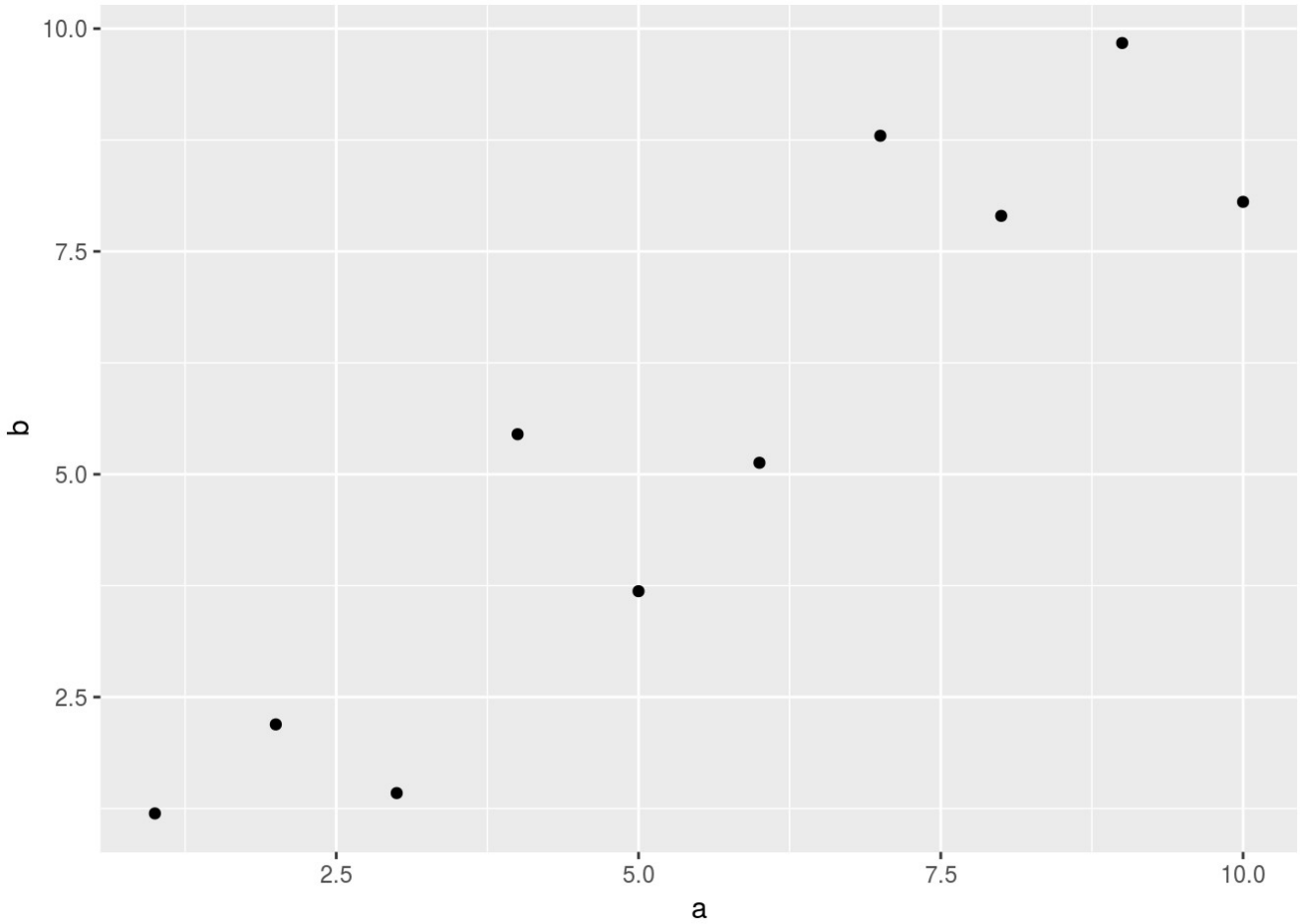


Istotną opcją jest możliwość "podmiany" źródłowego zbioru danych "w locie" za pomocą operatora **%+%**

```
# PRZYKŁAD 1.6
df2 <- data.frame(a = 1:10, b = (1:10)+runif(10,-2,2), c = 1:10)
g <- ggplot(df1)
g + geom_point(aes(x = a, y = b))
```

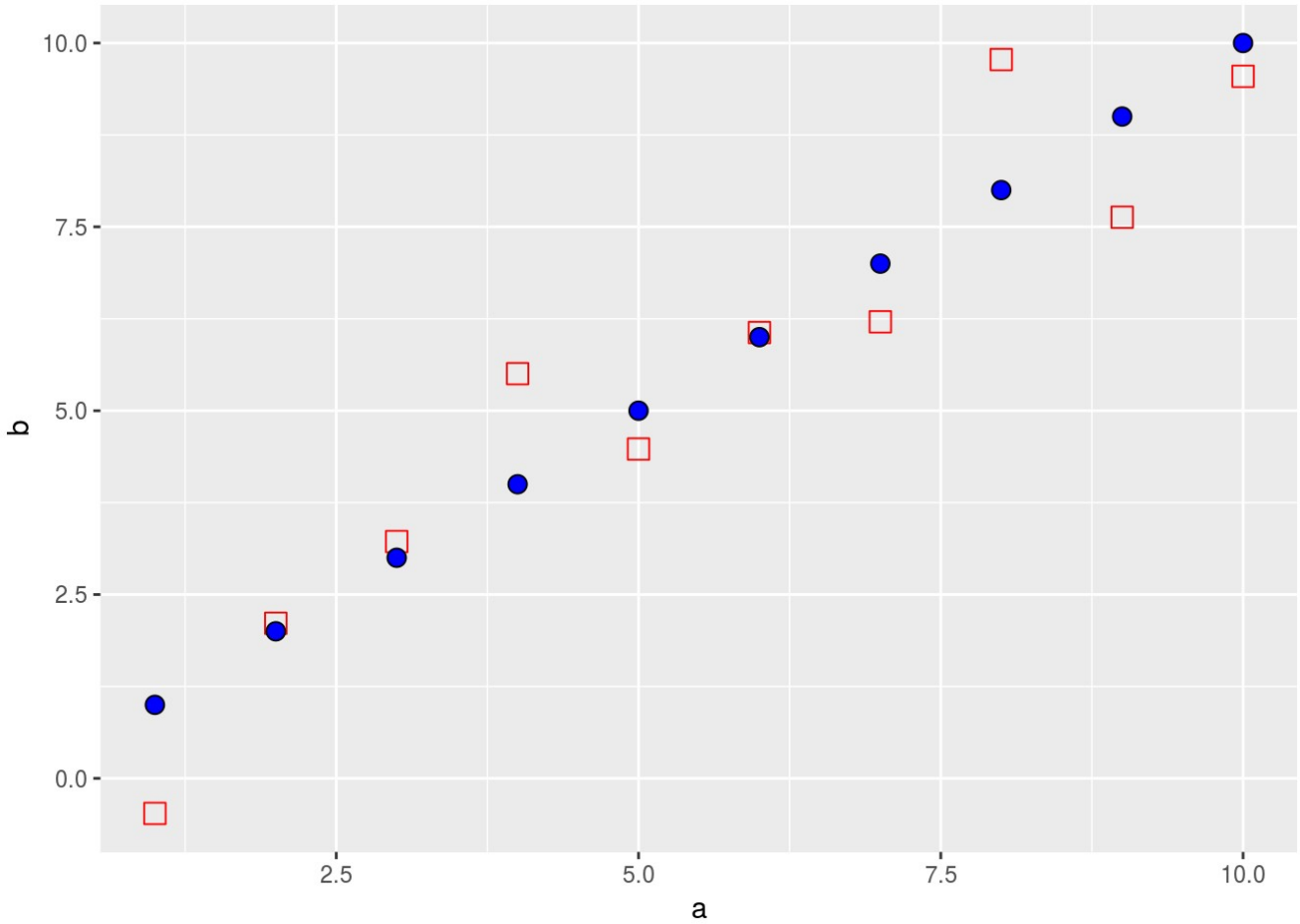



```
g %+% df2 + geom_point(aes(x = a, y = b))
```



Operując cechami **size=""**, **shape=""**, **colour=""** oraz **fill=""**, a także dodając kolejne warstwy, otrzymujemy pożądany efekt.

```
# PRZYKŁAD 1.7
g <- ggplot(df1, aes(x = a))
g + geom_point(aes(y=b), size=4, shape=22, colour="red") + geom_point(aes(y = c), size
=3, shape=21, fill="blue")
```

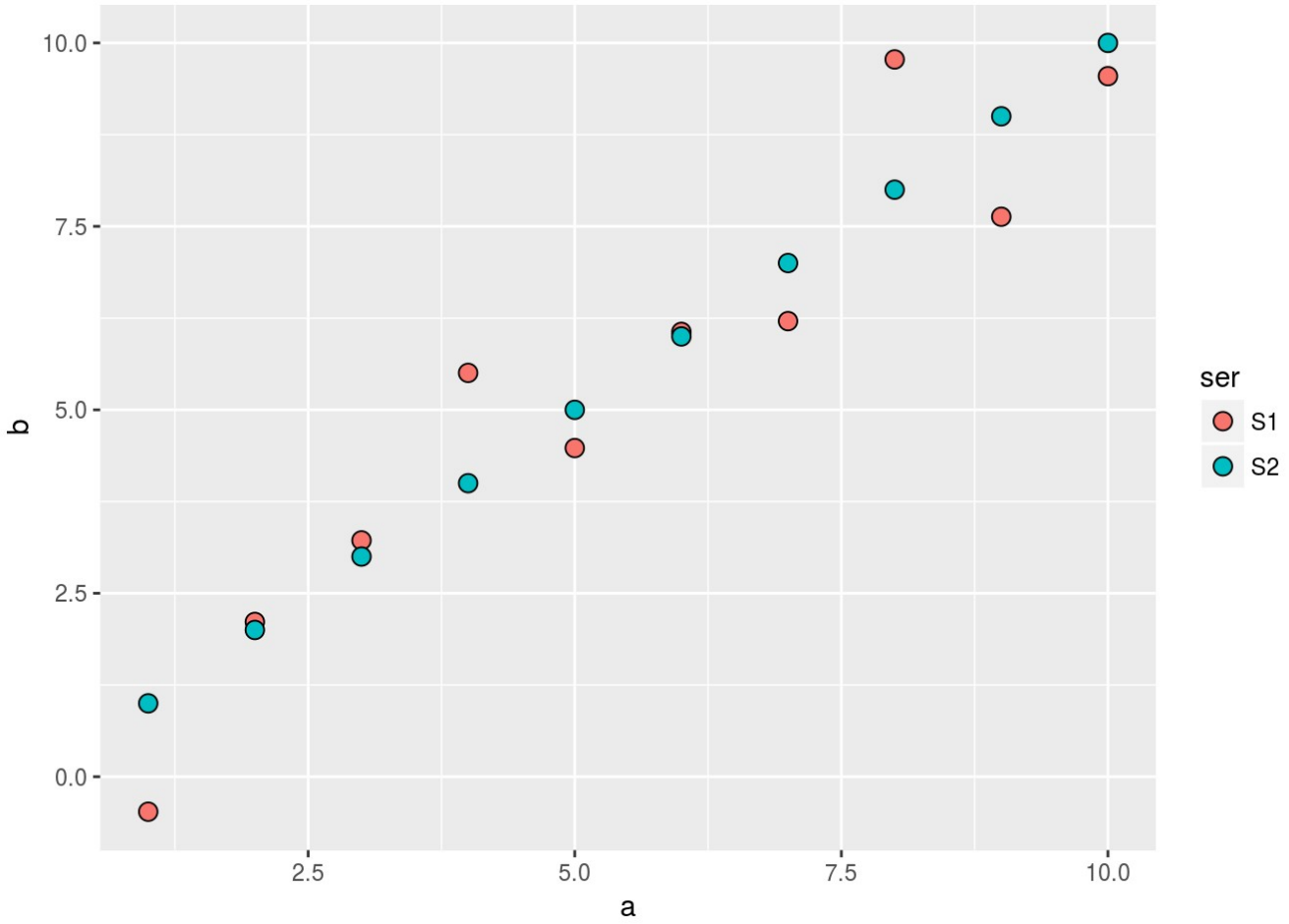


Umieszczenie wszystkich danych w jednej kolumnie oraz stworzenie dodatkowej (dodatkowych) kolumn, określających serie daje możliwość skorzystania z automatycznego przpisywania cech.

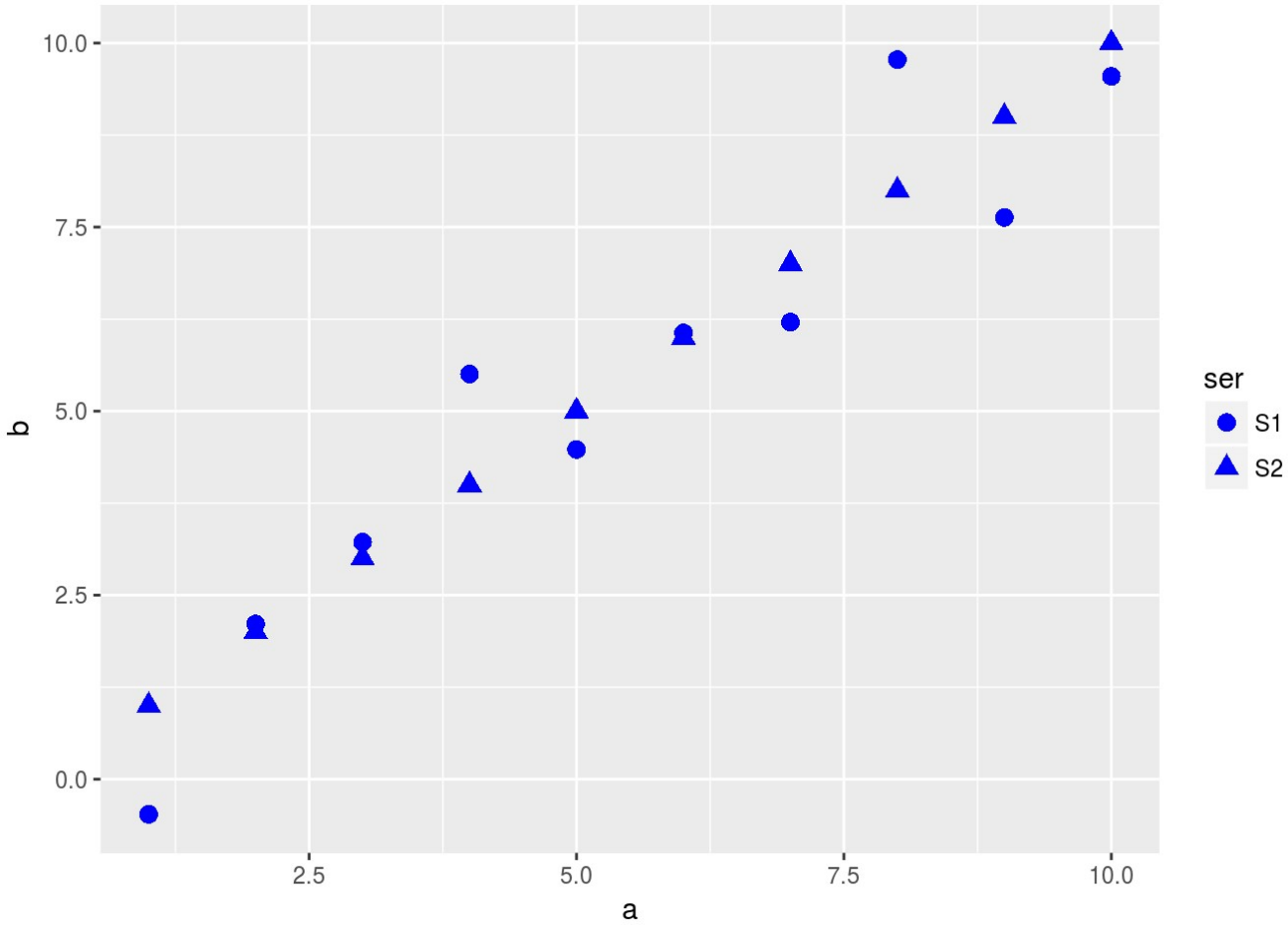
```
# PRZYKŁAD 1.8
df2 <- data.frame(a = c(df1$a, df1$a), b = c(df1$b, df1$c), n = sample(20), ser = c(re
p("S1",10), rep("S2", 10)))
df2
```

```
##     a          b  n ser
## 1   1 -0.4769591 10  S1
## 2   2  2.1090795  1  S1
## 3   3  3.2203672 15  S1
## 4   4  5.5035895  3  S1
## 5   5  4.4789019  6  S1
## 6   6  6.0625283  9  S1
## 7   7  6.2089541 12  S1
## 8   8  9.7750896 14  S1
## 9   9  7.6310188 18  S1
## 10 10  9.5474727 20  S1
## 11  1  1.0000000 11  S2
## 12  2  2.0000000  8  S2
## 13  3  3.0000000  5  S2
## 14  4  4.0000000 17  S2
## 15  5  5.0000000 13  S2
## 16  6  6.0000000  4  S2
## 17  7  7.0000000 16  S2
## 18  8  8.0000000  2  S2
## 19  9  9.0000000  7  S2
## 20 10 10.0000000 19  S2
```

```
g <- ggplot(df2)
g + geom_point(aes(x = a, y = b, fill=ser), size=3, shape=21)
```

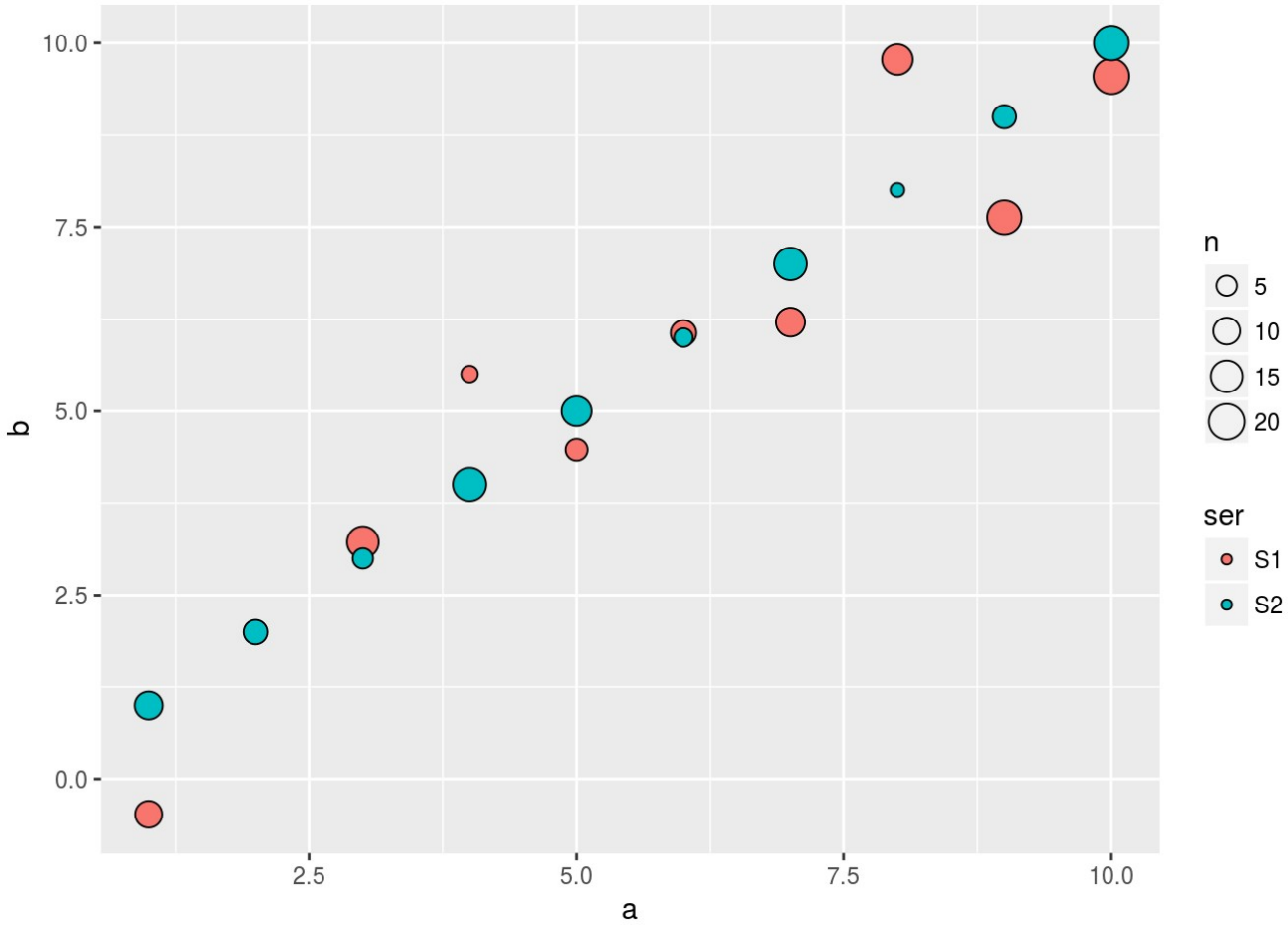



```
g + geom_point(aes(x = a, y = b, shape=ser), size=3, colour="blue")
```



Bardzo użytecznym zabiegiem jest także mapowanie rozmiaru, szczególnie, gdy ma on określać np liczbę obserwacji wchodzących do danego punktu.

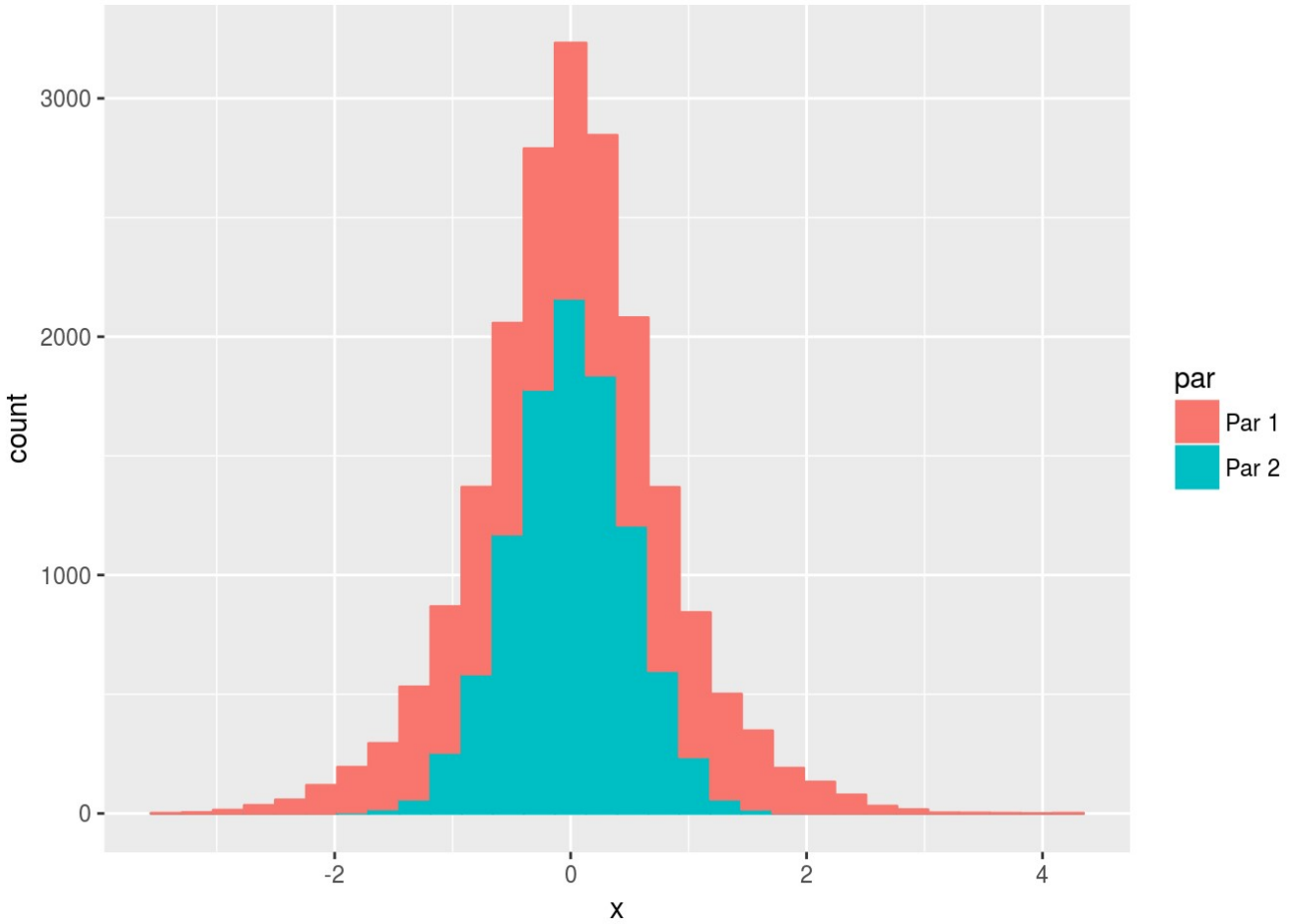
```
# PRZYKŁAD 1.9
g + geom_point(aes(x = a, y = b, fill=ser, size=n), shape=21)
```



Za pomocą geometrii **geom_histogram()** można utworzyć histogram wartości. W poniższym przykładzie najpierw w zmiennej typu dataframe utworzymy kilka serii danych, a nastepnie zacznimy od wizualizacji dwóch z nich.

```
# PRZYKŁAD 1.10
N1 <- 10000
N2 <- 5000
N3 <- 20000

sigma1 <- 1
sigma2 <- 0.5

mu <- 0

x1 <- rnorm(N1, mu, sigma1)
x2 <- rnorm(N2, mu, sigma1)
x3 <- rnorm(N3, mu, sigma1)
x4 <- rnorm(N1, mu, sigma2)
x5 <- rnorm(N2, mu, sigma2)
x6 <- rnorm(N3, mu, sigma2)

label.size <- c(rep("Size 1", N1), rep("Size 2", N2), rep("Size 3", N3))
label.par <- c(rep("Par 1", N1 + N2 + N3), rep("Par 2", N1 + N2 + N3))

df <- data.frame(x = c(x1, x2, x3, x4, x5, x6), size = c(label.size, label.size), par
= label.par)

g1 <- ggplot(data = df[df$size == "Size 1",])
g1 + geom_histogram(aes(x = x, fill = par, colour = par))
```

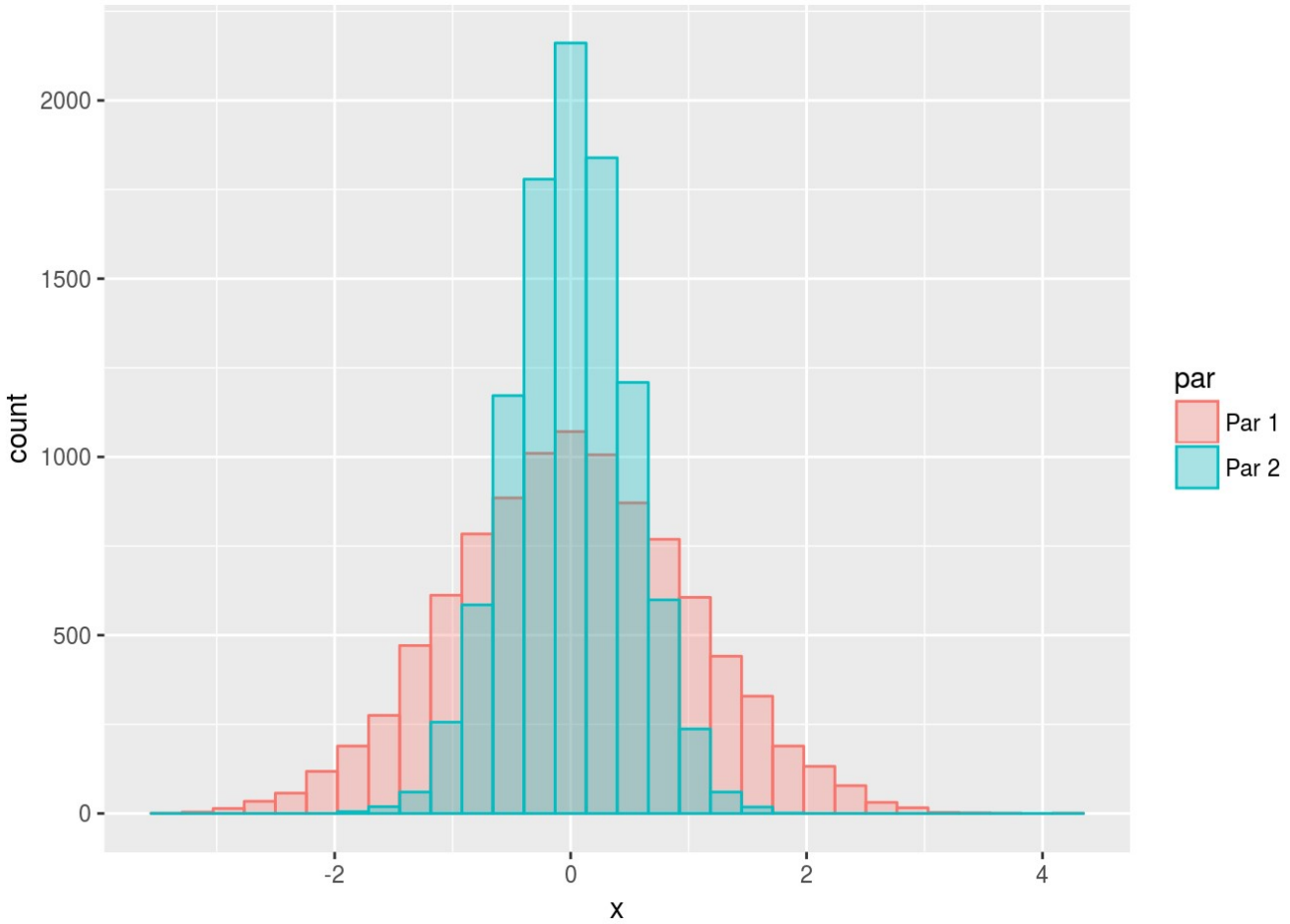
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Za pomocą opcji **position=""** można sterować wyglądem histogramów - domyślne **position="stack"** ustawia jeden słupek na drugim, **position="fill"** normalizuje sumę do jedynki, **position="dodge"** ustawia jeden obok drugiego. Opcja **position="identity"** przesłania jedną serię drugą, więc przydatne jest użycie cechy przezroczystości (za pomocą opcji **alpha=""**).

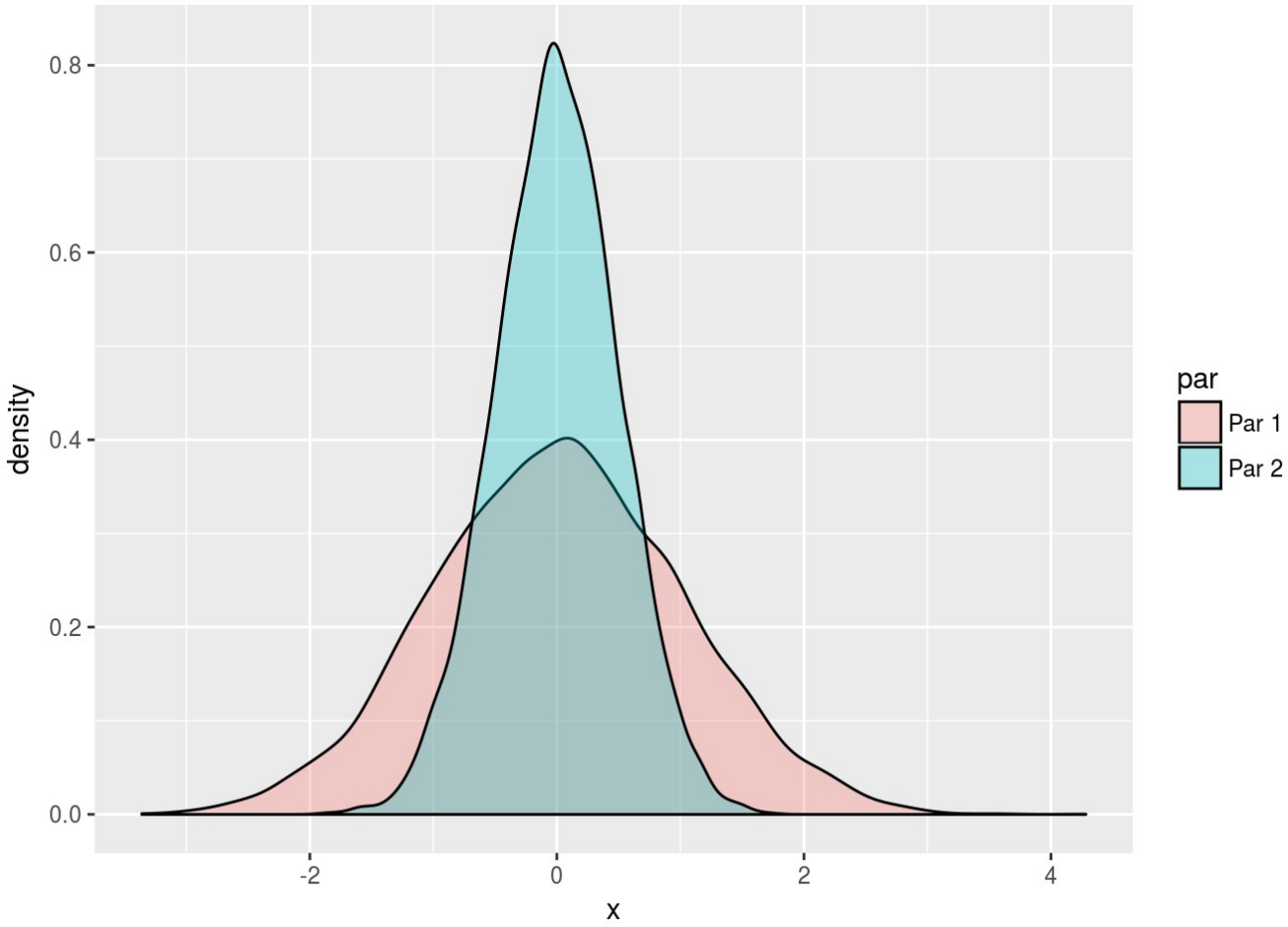
```
# PRZYKŁAD 1.11
g1 + geom_histogram(aes(x = x, fill = par, colour = par), position="identity", alpha=0
.3)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



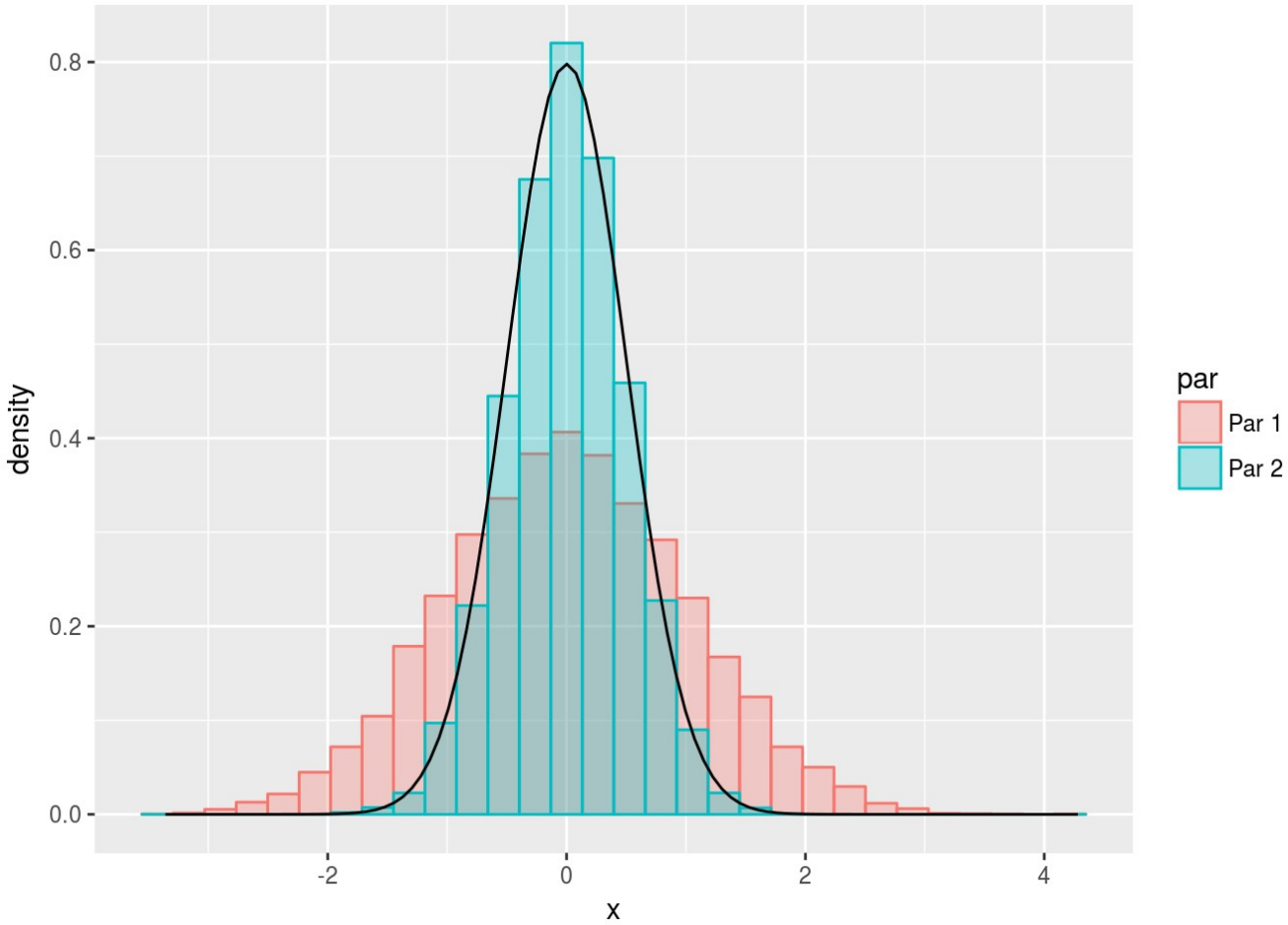
Aby zamiast liczby zliczeń uzyskać gęstość prawdopodobieństwa można skorzystać albo z geometrii **geom_density()** albo też z opcji **..density..**, podanej po zmiennej. Jeśli dodatkowo potrzebujemy wykreślić pewną funkcję (np. gęstość prawdopodobieństwa rozkładu normalnego), korzystamy wtedy z funkcji **stat_function()**.

```
# PRZYKŁAD 1.12
g1 + geom_density(aes(x = x, fill = par), colour="black", alpha=0.3)
```




```
g1 + geom_histogram(aes(x = x,..density.., fill = par, colour = par), position="identi
ty", alpha=0.3) +
stat_function(fun = dnorm, args = list(mean = 0, sd = 0.5))
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Potoki

R jest językiem funkcyjnym, co oznacza, że nasz kod jest często wręcz upstrzony kolejnymi nawiasami, które mocno zaciemniają obraz tego, co faktycznie jest wykonywane. Szczególnie, gdy kod jest skomplikowany, a funkcje zagnieżdżone w sobie prowadzi to estetycznych i programistycznych problemów.

```
# PRZYKŁAD 1.13
x <- c(0.109, 0.359, 0.63, 0.996, 0.515, 0.142, 0.017, 0.829, 0.907)

round(exp(diff(log(x))), 1)
```

```
## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1
```

Zamiast takiego zapisu, wygodnie jest skorzystać z pakietu **magrittr**, który oferuje operator potoku (pipe) **%>%**:

```
# PRZYKŁAD 1.14
library(magrittr)

x %>% log() %>%
    diff() %>%
    exp() %>%
    round(1)
```

```
## [1]  3.3  1.8  1.6  0.5  0.3  0.1 48.8  1.1
```

Ogólna idea jest taka, że operator potoku korzysta ze zmiennej po lewej stronie operacji i podmienia ją w funkcji (czy operacji) po prawej stronie. Najczęstyszym przypakiem jest zapisanie operacji **f(x)** jako **x %>% f** np.

```
# PRZYKŁAD 1.15
log(2)
```

```
## [1] 0.6931472
```

```
2 %>% log
```

```
## [1] 0.6931472
```

Inną opcją jest zastąpienie operacji **f(x,y)** za pomocą **x %>% f(y)**

```
# PRZYKŁAD 1.15
round(pi, 6)
```

```
## [1] 3.141593
```

```
pi %>% round(6)
```

```
## [1] 3.141593
```

W przypadku, gdy domyślny argument nie jest akurat pierwszy, korzystamy z ``wypełniacza'' realizowanego za pomocą kropki **f(x, .)**

```
# PRZYKŁAD 1.16
round(pi, 6)
```

```
## [1] 3.141593
```

```
6 %>% round(pi, digits=.)
```

```
## [1] 3.141593
```

Operator **%>%** nie jest jedynym operatorem potoku z pakietu **magrittr**. W przypadku, gdybyśmy chcieli dokonać zmiany zmiennej, tzn wykonać operację po prawej stronie operatora, a następnie przypisać wynik do lewej strony, z pomocą przychodzi operator **%<>%**

```
# PRZYKŁAD 1.16
x <- 2
x
```

```
## [1] 2
```

```
x %<>% log()
x
```

```
## [1] 0.6931472
```

Oczywiście po prawej stronie dalej można wykonywać inne operacje. Istotne, aby **%<>%** był pierwszym operatorem.

```
# PRZYKŁAD 1.17
x <- 2
x %<>% log() %>% round(2)
x
```

```
## [1] 0.69
```

Innym opcją jest wyłuskanie konkretnej zmiennej z większego obiektu po lewej stronie i przekazanie do wykorzystania po stronie prawej. Wtedy korzystamy z operatora **%$%**. Chodzi tu kwestię widoczności zmiennych - niektóre popularne funkcje nie mają takiej opcji.

```
# PRZYKŁAD 1.18
df1 <- data.frame(a = 1:10, b = (1:10) + runif(10,-2,2), c = 1:10)

# Pełna macierz korelacji
df1 %>% cor
```

```
##          a        b        c
## a 1.000000 0.955308 1.000000
## b 0.955308 1.000000 0.955308
## c 1.000000 0.955308 1.000000
```

```
# Teraz chcemy korelowac jedynie kolumny a i b
df1 %>% cor(a,b)
```

```
## Error in pmatch(use, c("all.obs", "complete.obs", "pairwise.complete.obs", : object
'b' not found
```

```
# A do tego potrzebny jest operator %$%
df1 %$% cor(a,b)
```

```
## [1] 0.955308
```

Pakiet dplyr

Często bardzo dużym problemem jest warunkowanie zmiennych (tzn. wypisywanie części obiektu, które spełniają pewne okreslone założenia), a w szczególności kolumn w obiekcie typu ramka danych. Podobnie jest z pozornie prostymi operacjami jak sortowanie, czy wręcz wybieranie konkretnych kolumn. Do obsługi takich przypadków bardzo wygodny jest pakiet **dplyr**. Wexmy prosty przypadek ramki danych **df1**, którą chcielibyśmy posortować wg. drugiej kolumny. Normalnie, musielibyśmy użyć następującej konstrukcji

```
# PRZYKŁAD 1.19
df1 <- data.frame(a = 1:10, b = runif(10,-2,2), c = 1:10)
df1[order(df1$b),]
```

```
##     a          b  c
## 10 10 -1.9421530 10
## 9   9 -1.8714934  9
## 5   5 -1.5333940  5
## 6   6 -1.4016227  6
## 3   3  0.8672414  3
## 7   7  0.8805913  7
## 2   2  0.9253768  2
## 1   1  1.2807686  1
## 4   4  1.3450420  4
## 8   8  1.4384216  8
```

Korzystając z **dyplr** używamy funkcji **arrange()** podając zmienną, z której chcemy skorzystać

```
# PRZYKŁAD 1.19
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
```

```
arrange(df1, b)
```

```
##     a          b  c
## 1  10 -1.9421530 10
## 2   9 -1.8714934  9
## 3   5 -1.5333940  5
## 4   6 -1.4016227  6
## 5   3  0.8672414  3
## 6   7  0.8805913  7
## 7   2  0.9253768  2
## 8   1  1.2807686  1
## 9   4  1.3450420  4
## 10  8  1.4384216  8
```

Podobnie funkcja **select()** wybiera kolumny z ramki

```
# PRZYKŁAD 1.20
df1[,c("b","a")]
```

```
##             b  a
## 1   1.2807686  1
## 2   0.9253768  2
## 3   0.8672414  3
## 4   1.3450420  4
## 5  -1.5333940  5
## 6  -1.4016227  6
## 7   0.8805913  7
## 8   1.4384216  8
## 9  -1.8714934  9
## 10 -1.9421530 10
```

```
select(df1, b, a)
```

```
##             b  a
## 1   1.2807686  1
## 2   0.9253768  2
## 3   0.8672414  3
## 4   1.3450420  4
## 5  -1.5333940  5
## 6  -1.4016227  6
## 7   0.8805913  7
## 8   1.4384216  8
## 9  -1.8714934  9
## 10 -1.9421530 10
```

Natomiast **filter()**, umożliwia wybranie konkretnych wartości rekordów

```
# PRZYKŁAD 1.20
df1[df1$a < 5 & df1$b < 0,]
```

```
## [1] a b c
## <0 rows> (or 0-length row.names)
```

```
filter(df1, a < 5, b < 0)
```

```
## [1] a b c
## <0 rows> (or 0-length row.names)
```

Oczywiście, nic nie stoi na przeszkodzie, aby zastosować poznany mechanizm potoku i połączyć to w jedną całość:

```
# PRZYKŁAD 1.20
df1 %>%
  filter(a < 5, b < 0) %>%
  select(c, b) %>%
  arrange(desc(b), c)
```

```
## [1] c b
## <0 rows> (or 0-length row.names)
```

Funkcja **summarise** wykonuje podsumowanie danych

```
# PRZYKŁAD 1.21
df1 %>%
  summarise(mean(a))
```

```
##   mean(a)
## 1     5.5
```

przy czym sama nie ma większego sensu, natomiast jest bardzop przydatna, gdy połączy się ją z funkcją **group_by**

```
# PRZYKŁAD 1.21
df1 <- data.frame(a = 1:10, b = runif(10,-2,2), c = rep(c(1,0), each = 5))
df1 %>%
  group_by(c) %>%
  summarise(ma = mean(a), mb = mean(b))
```

```
## # A tibble: 2 x 3
##       c    ma      mb
##   <dbl> <dbl>   <dbl>
## 1     0     8 -0.0185
## 2     1     3 -0.490
```