Porównanie dokładności klasycznych i zaawansowanych metod klasyfikacji

Analiza skuteczności modeli na zbiorze danych Glass

Dominika Szulc, Wiktoria Jarząb

2025-04-03

Spis treści

1 Cel analizy

Celem naszej analizy jest kompleksowe porównanie dokładności algorytmów klasyfikacji na zbiorze danych `Glass`. Badanie ma na celu wyłonienie metody o najmniejszym błędzie klasyfikacji oraz sprawdzenie, czy zastosowanie zaawansowanych metod przynosi poprawę wyników względem klasycznych rozwiązań.

Porównamy następujące grupy algorytmów:

1. Klasyczne metody klasyfikacji:

```
* metoda k-najbliższych sąsiadów (kNN),
* drzewa klasyfikacyjne,
* naiwny klasyfikator bayesowski.
```

Do modeli z tej grupy zastostujemy dodatkowo zaawansowane schematy oceny dokładności klasyfikacji:

- metoda *cross-validation*,
- schemat *bootstrap*:
  - schemat *bootstrap 632plus*.

2. Zaawansowane metody klasyfikacji:

```
* metoda wektorów nośnych (SVM) z różnymi funkcjami jądrowymi,
* rodziny klasyfikatorów:

    * metoda *bagging*,
    * metoda *boosting*,
    * metoda *random forest*.
```

Aby wybrać najlepszy klasyfikator podzielimy dane na zbiór uczący i testowy w proporcji $7:3$. Dla każdego modelu wyznaczymy macierze pomyłek oraz błędy klasyfikacji ($1-$dokładność klasyfikacji).

Przy formułowaniu wniosków i wyborze najlepszej metody będziemy się kierować błędami na zbiorach testowych. Błędy na zbiorach uczących możemy potraktujemy pomocniczo, ponieważ inforumją one głównie o tym jak dobrze model nauczył się na danych (np. czy nie nastąpiło przeuczenie).

2 Dane `Glass`

2.1 Przygotowanie danych

```
## 'data.frame':    214 obs. of  10 variables:
##  $ RI  : num  1.52 1.52 1.52 1.52 1.52 ...
##  $ Na  : num  13.6 13.9 13.5 13.2 13.3 ...
##  $ Mg  : num  4.49 3.6 3.55 3.69 3.62 3.61 3.6 3.61 3.58 3.6 ...
##  $ Al  : num  1.1 1.36 1.54 1.29 1.24 1.62 1.14 1.05 1.37 1.36 ...
##  $ Si  : num  71.8 72.7 73 72.6 73.1 ...
##  $ K   : num  0.06 0.48 0.39 0.57 0.55 0.64 0.58 0.57 0.56 0.57 ...
##  $ Ca  : num  8.75 7.83 7.78 8.22 8.07 8.07 8.17 8.24 8.3 8.4 ...
##  $ Ba  : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ Fe  : num  0 0 0 0 0 0.26 0 0 0 0.11 ...
##  $ Type: Factor w/ 6 levels "1","2","3","5",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Dane `Glass` z pakietu `mlbench` po wczytaniu nie wymagają żadnych modyfikacji. Wszystkie zmienne zostały poprawnie wczytane.

2.2 Informacje o danych

Dane mają 214 przypadków i 10 cech.

Tablica 1: Opis danych Glass

Nazwa Kolumny	Typ Danych	Opis Danych
RI	ciągłe	współczynnik załamania światła
Na	ciągłe	zawartość sodu
Mg	ciągłe	zawartość magnezu
Al	ciągłe	zawartość glinu
Si	ciągłe	zawartość krzemu
K	ciągłe	zawartość potasu
Ca	ciągłe	zawartość wapnia
Ba	ciągłe	zawartość baru
Fe	ciągłe	zawartość żelaza
Type	jakościowe	typ szkła

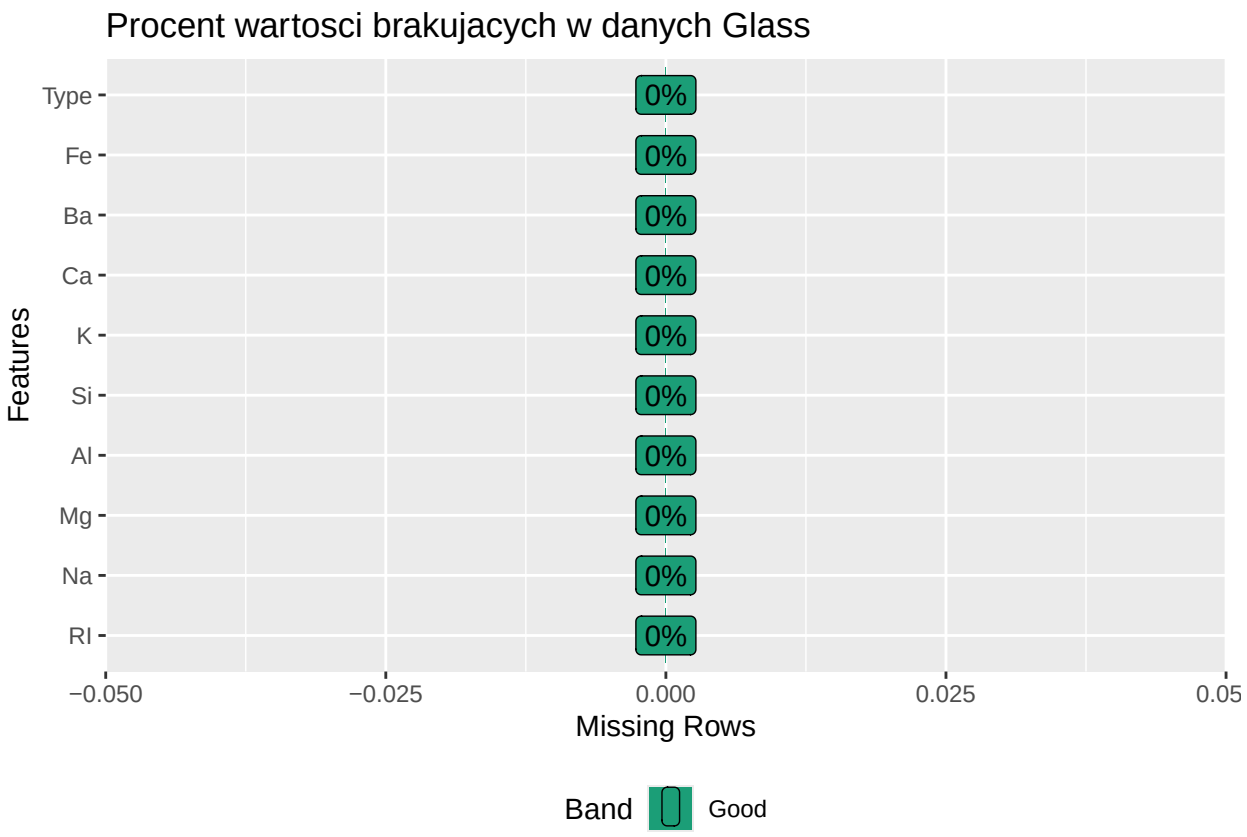
Zmienna `Type` zawiera etykietki klas. Zgodnie z dokumentacją na stronie

https://archive.ics.uci.edu/dataset/42/glass+identification są to odpowiednio:

- 1 - okno budunku (typ float),
- 2 - okno budynku (typ non-float),
- 3 - okno pojazdu (typ float),
- 4 - okno pojazdu (typ non-float) - brak w tym zbiorze danych,
- 5 - pojemniki,
- 6 - szkło stołowe,
- 7 - reflektory.

Mamy zatem 6 klas w tym zbiorze danych.

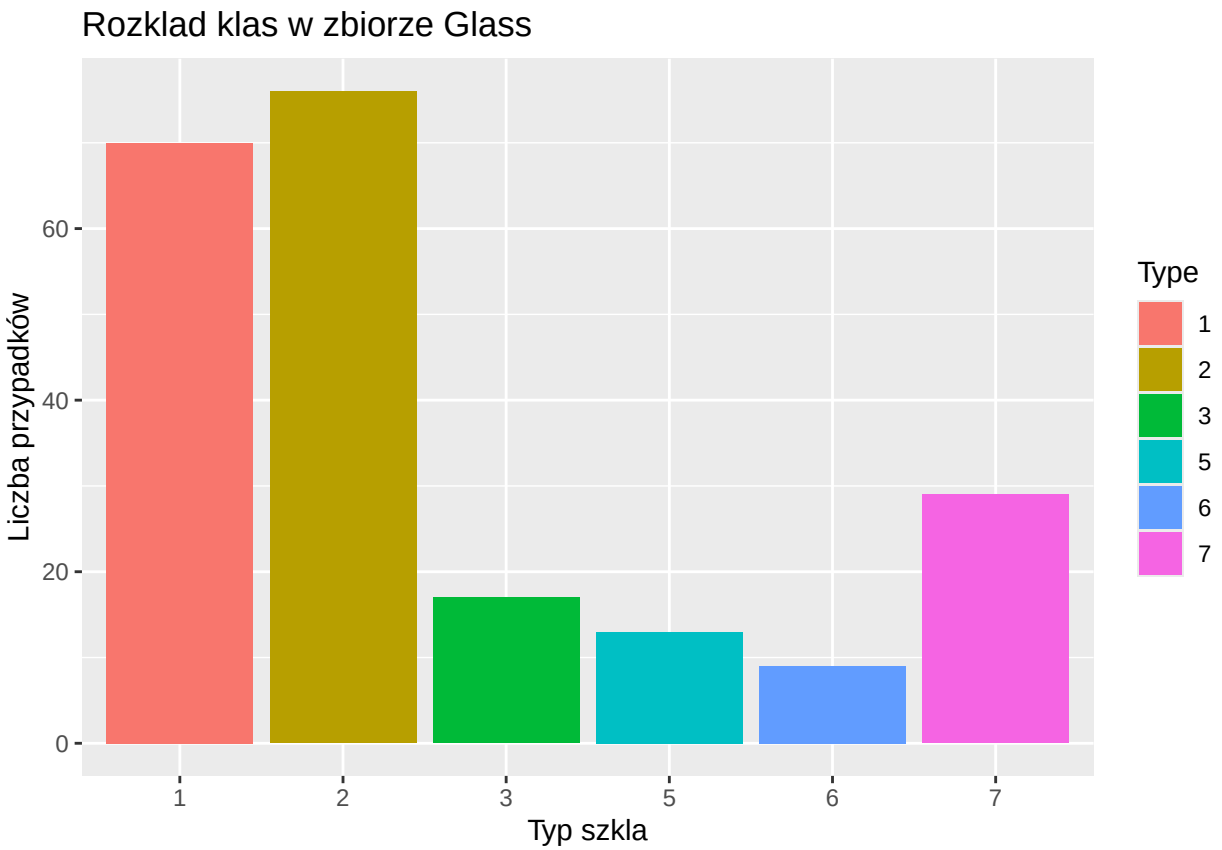
2.2.1 Wartości brakujące



Rysunek 1: Wykres przedstawiający procentowy udział wartości brakujących w danych Glass

Brak wartości brakujących.

2.3 Rozkład klas



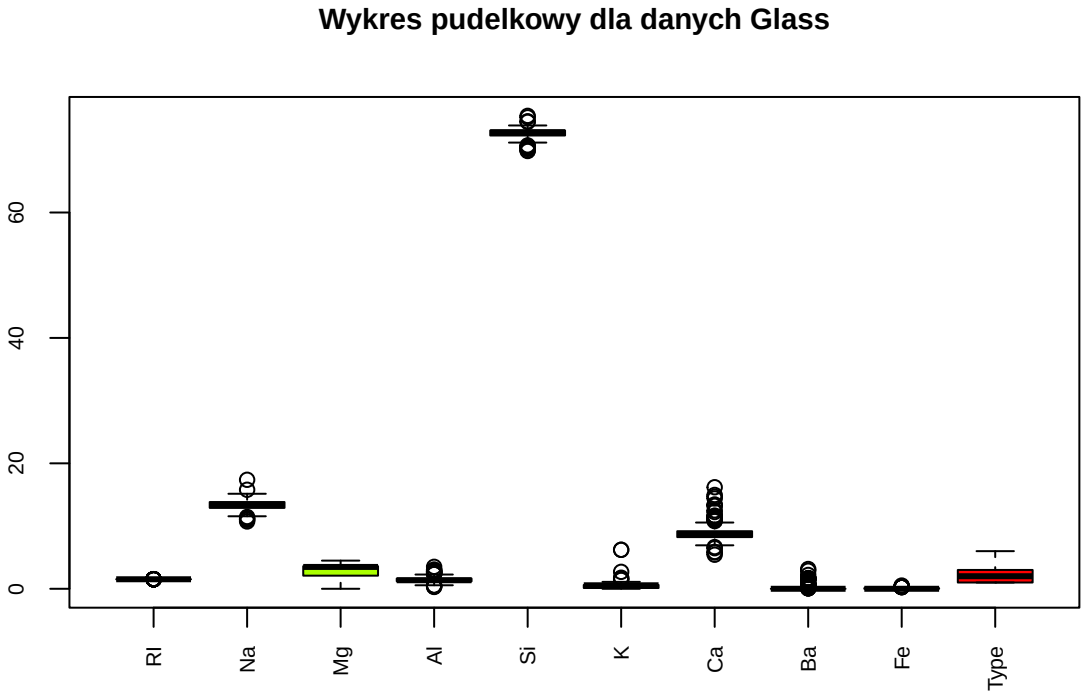
Rysunek 2: Wykres przedstawiający rozkład klas w zbiorze Glass

Na wykresie 2 widać istotną dysproporcję pomiędzy klasami. W klasie 1. i 2. jest zdecydowanie więcej przypadków niż w pozostałych czterech.

Błąd klasyfikacji, jaki otrzymalibyśmy przypisując wszystkie obiekty do najczęściej występującej klasy (czyli klasy 2.), wynosi 0.645. Wynik ten otrzymujemy, odejmując od 1 iloraz ilości elementów klasy 2. i liczby wszystkich obserwacji.

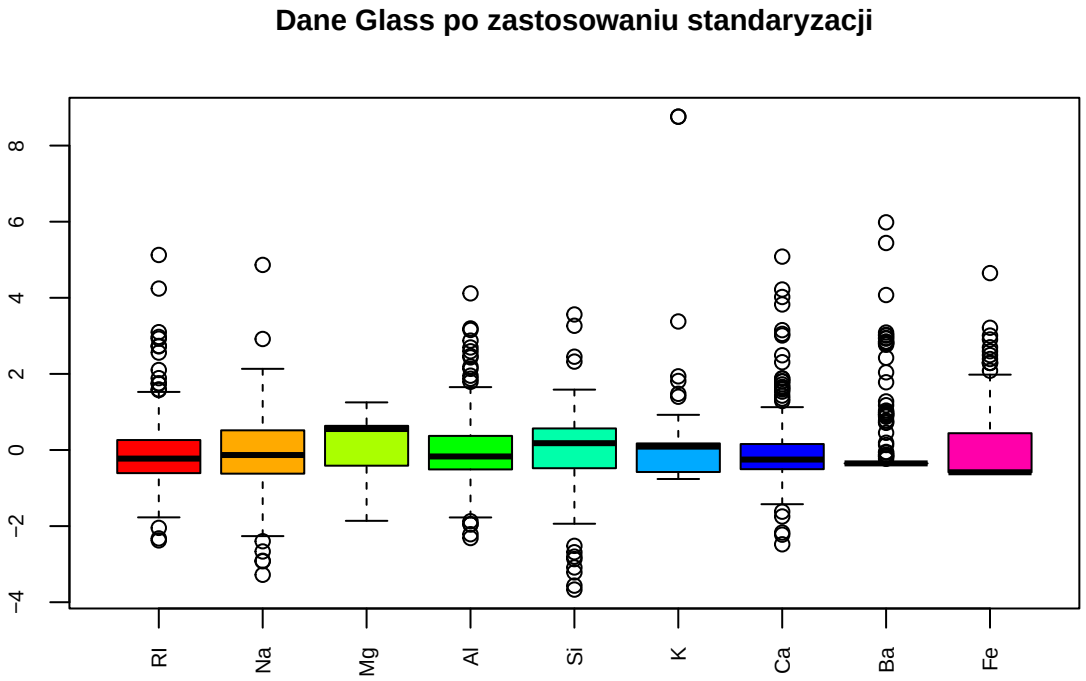
2.3.1 Standaryzacja

Przyjrzymy się teraz uważniej zmienności (wariancji) poszczególnych cech, co pomoże stwierdzić, czy należy zastosować standaryzację.



Rysunek 3: Wykres pudełkowy pozwalający rozstrzygnąć, czy należy zastosować standaryzację dla danych Glass.

Cechy charakteryzują się wyraźną zmiennością (wykres 3), dlatego zdecydowałyśmy się zastosować standaryzację dla niektórych algorytmów, żeby uniknąć sytuacji, w której zmienna `Si` zdominowałaby pozostałe cechy.

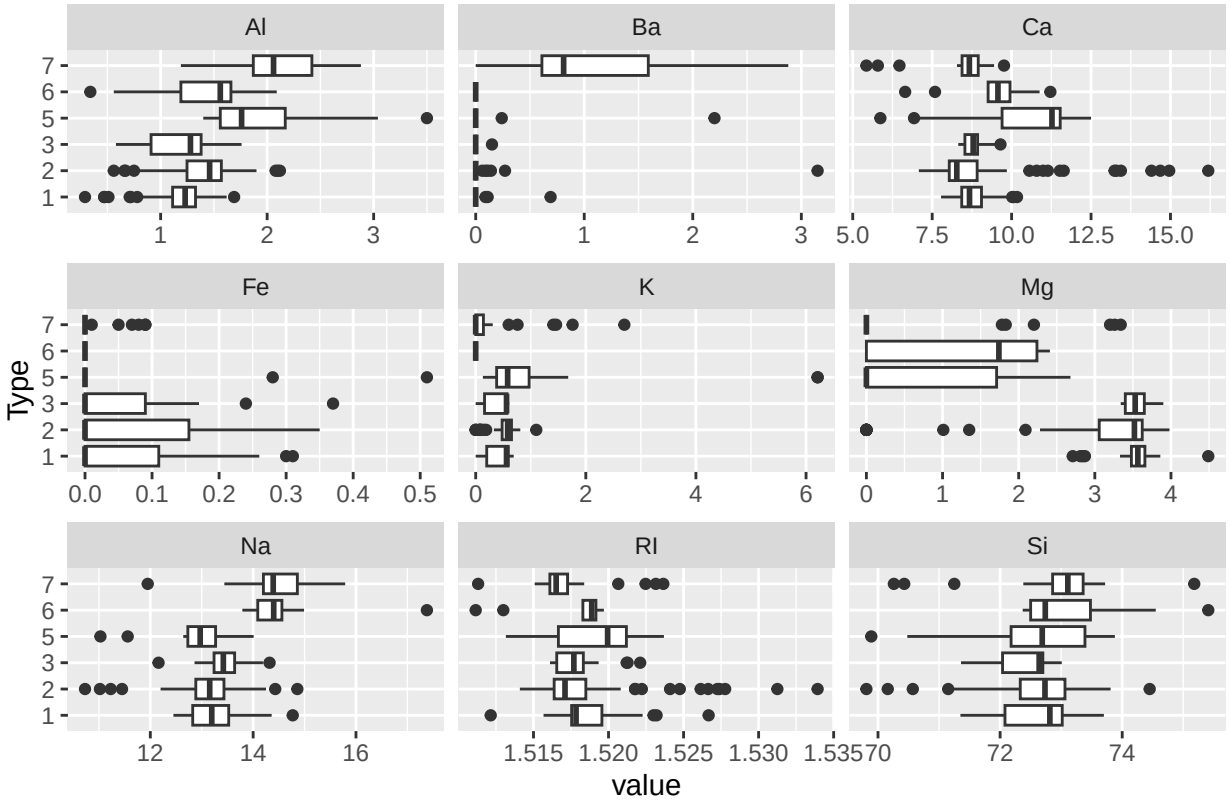


Rysunek 4: Wykres pudełkowy dla danych Glass po zastosowaniu standaryzacji.

Teraz sytuacja wygląda znacznie lepiej (wykres 4), jednak dla niektórych algorytów, których działanie będziemy porównywać w dalszej analizie, może być problematyczne, dlatego narazie będziemy nadal operować na oryginalnym zbiorze danych `Glass`.

2.3.2 Zmienność

Przyjrzymy się teraz zmienności naszych danych.



Rysunek 5: Wykresy pudełkowe przedstawiające rozkład wartości poszczególnych cech ze względu na zmienną Type.

Niektóre cechy powalają na istotne rozróżnienie przynależności do poszczególnych klas:

- `Ba` pozwala odróżnić klasę 7 od pozostałych,
- `Mg` i `Fe` wyróżniają klasy 1, 2 i 3,
- `Mg` pozwala również na rozróżnienie klas 5 i 6 od pozostałych,
- `Al`, `Ca`, `K`, `Na`, `RI` oraz `Si` cechują się ustandaryzowaną zmiennością klas.

2.4 Podział zbioru

Zaczynamy od podziału zbioru `Glass` na zbiór uczący (`glass.uczący`) oraz testowy (`glass.testowy`) w proporcji $7:3$. Dane wybieramy losowo.

3 Klasyczne metody klasyfikacji

3.1 Metoda k-najbliższych sąsiadów

W przypadku metody k-najbliższych sąsiadów zastosowanie standaryzacji jest zalecane, ponieważ algorytm bazuje na odległości między przypadkami, dlatego zmienne o większej skali będą wyraźnie dominować nad tymi o mniejszym zakresie.

Do analizy użyjemy zbioru `Glass.s` zawierającego zestandaryzowane zmienne ciągłe z danych `Glass` wraz z kolumną `Type` oraz zestandaryzowanego, wcześniej wylosowanego, zbioru uczącego i testowego.

3.1.1 Analiza

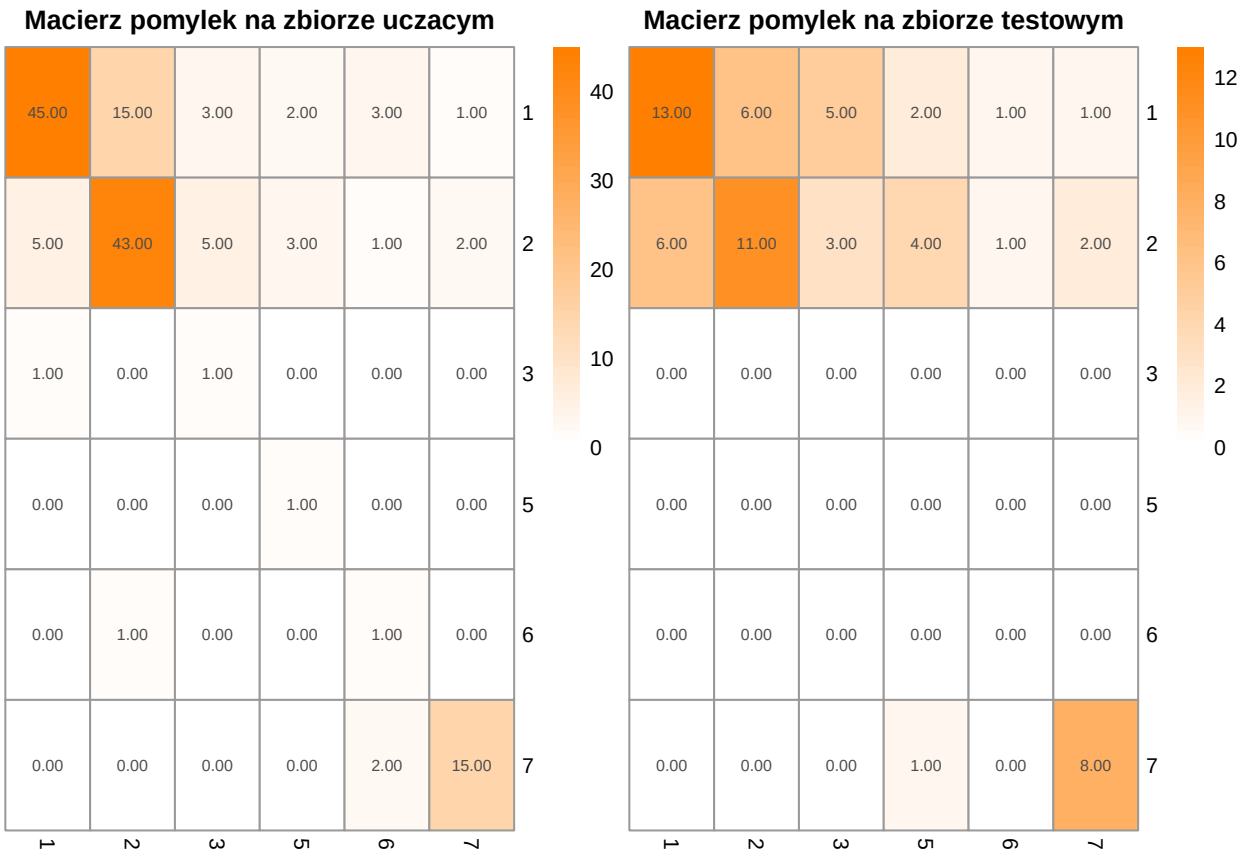
Zaczynamy od stworzenia własnej funkcji `knn.glass` dla argumentów:

- zbiór uczący,
- zbiór testowy,
- k - liczba najbliższych sąsiadów,
- `formuła` - wybrane cechy (domyślnie wszystkie zmienne z danych `Glass`).

Dla nich funkcja zwraca macierze pomyłek oraz błędy klasyfikacji wyznaczone na podstawie modelu zbudowanego za pomocą funkcji `ipredknn` (z pakietu `ipred`).

```
knn.glass <- function(uczący, testowy, k,
                      formula = ipredknn(Type ~ ., data = uczący, k = k)) {
  model.knn.u <- formula # model podstawowy

  # etykietki rzeczywiste
  etykietki.rzecz.t <- testowy$Type
  etykietki.rzecz.u <- uczący$Type

  # etykietki prognozowane - prawdopodobieństwo dla dominującej klasy
  etykietki.prog.t <- predict(model.knn.u, testowy, type="class")
  etykietki.prog.u <- predict(model.knn.u, uczący, type="class")

  # macierz pomyłek - przewidział wiersz, ale rzeczywistości była to kolumna
  pomyłki.t <- table(etykietki.prog.t, etykietki.rzecz.t)
  pomyłki.u <- table(etykietki.prog.u, etykietki.rzecz.u)

  # błąd klasyfikacji
  n.test.t <- dim(testowy)[1]
  error.t <- (n.test.t - sum(diag(pomyłki.t))) / n.test.t

  n.test.u <- dim(uczący)[1]
  error.u <- (n.test.u - sum(diag(pomyłki.u))) / n.test.u

  return(list(pomyłki.u, error.u, pomyłki.t, error.t))
}
```

Teraz, korzystając z funkcji `knn.glass`, wyznaczamy błędy klasyfikacji oraz macierze pomyłek dla zbioru uczącego i testowego dla 14-najbliższych sąsiadów.



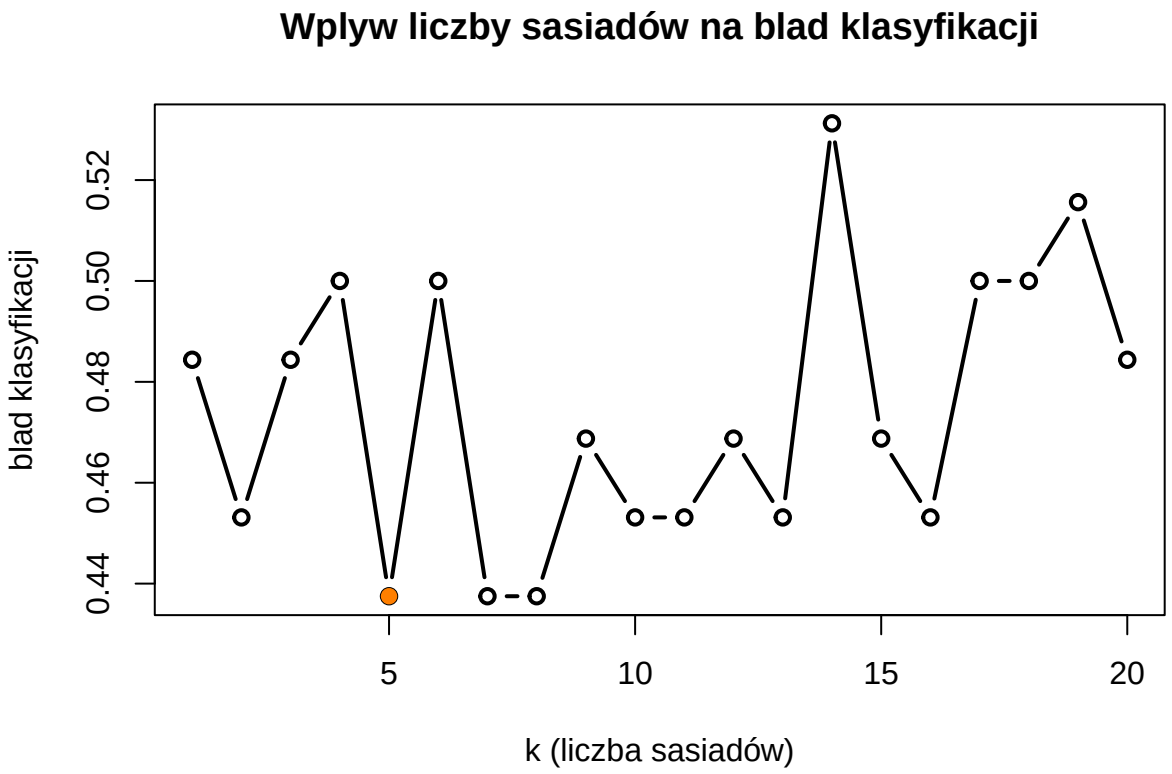
Rysunek 6: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla 14-u najbliższych sąsiadów.

Błąd klasyfikacji na zbiorze uczącym: 0.293.

Błąd klasyfikacji na zbiorze testowym: 0.5.

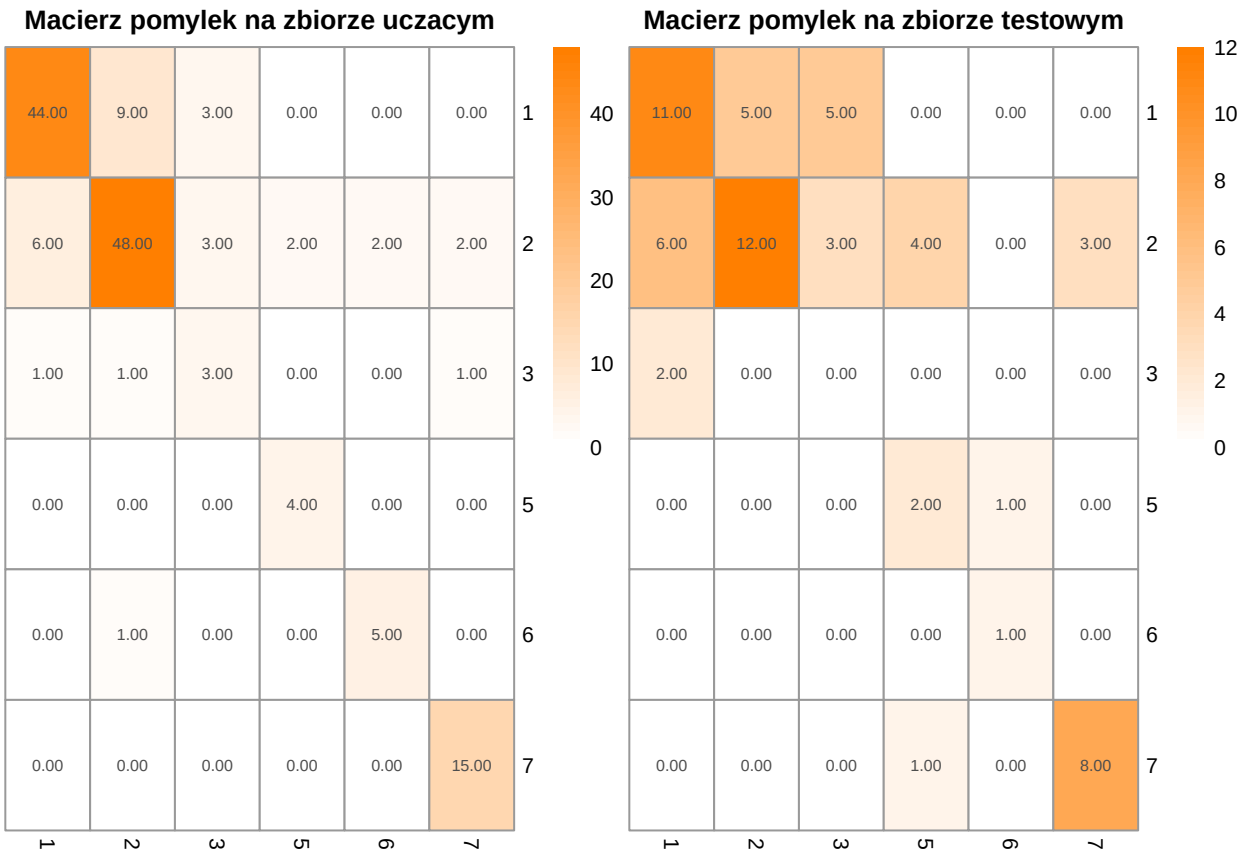
3.1.2 Wpływ liczby sąsiadów na dokładność modelu

Badamy teraz wpływ liczby sąsiadów na dokładność modelu dla naszego zbioru uczącego i testowego.



Rysunek 7: Wykres pozwalający wybrać najbardziej optymalną liczbą k najbliższych sąsiadów dla wygenerowanych wcześniej zbioru uczącego i testowego. Najoptymalniejsze k zostało zaznaczone kolorem.

Jak widać na wykresie 7 pierwsza najoptymalniejsza liczba sąsiadów dla zestandaryzowanych zbiorów uczącego i testowego wynosi 5. Stosujemy teraz dla niej naszą funkcję.



Rysunek 8: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla najoptymalniejszej liczby najbliższych sąsiadów.

Błąd klasyfikacji na zbiorze uczącym: 0.207.

Błąd klasyfikacji na zbiorze testowym: 0.469.

Porównując wyniki z tymi, otrzymanymi dla 14-najbliższych sąsiadów widzimy, że model dla najoptymalnejszej liczby sąsiadów jest lepszy - jego błędy na zbiorze uczącym i testowym są mniejsze niż dla 14 sąsiadów.

3.1.3 Podzbiór zmiennych o najlepszej i najgorszej zdolności dyskryminacyjnej

Tworzymy teraz dwa nowe modele dla najoptymalniejszej liczby najbliższych sąsiadów. Pierwszy będziemy tworzyć w oparciu o cechy o najlepszej zdolności dyskryminacyjnej (zatem `Ba`, `Fe` i `Mg`, jak widać na wykresie 5), a drugi w oparciu o cechy o najgorszej zdolności dyskryminacyjnej (`K`, `RI` i `Si`).

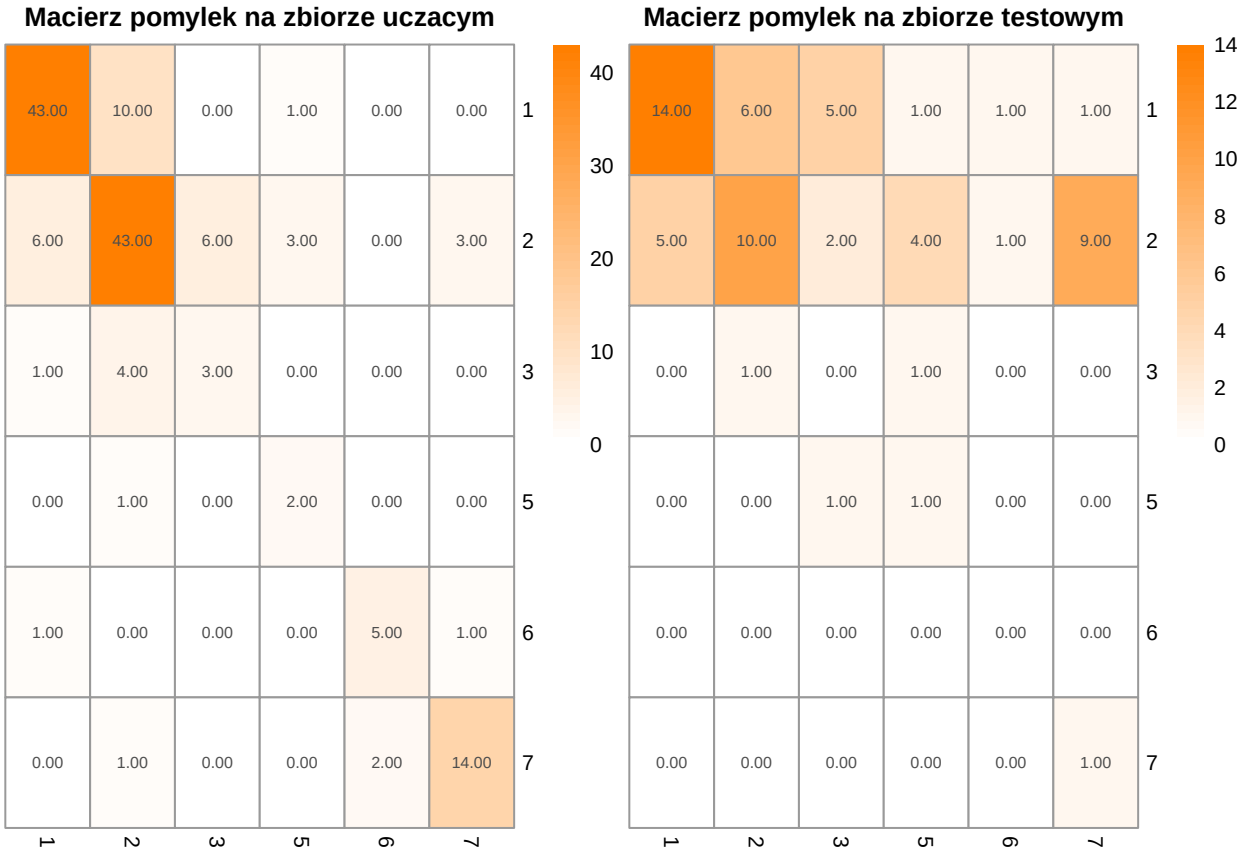
3.1.4 Zbiór zmiennych o najlepszej zdolności dyskryminacyjnej

Rysunek 9: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla zmiennych o najlepszej zdolności dyskryminacyjnej dla najoptymalniejszej liczby najbliższych sąsiadów.

Błąd klasyfikacji na zbiorze uczącym: 0.353.

Błąd klasyfikacji na zbiorze testowym: 0.422.

3.1.5 Zbiór zmiennych o najgorszej zdolności dyskryminacyjnej



Rysunek 10: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla zmiennych o najgorszej zdolności dyskryminacyjnej dla najoptymalniejszej liczby najbliższych sąsiadów.

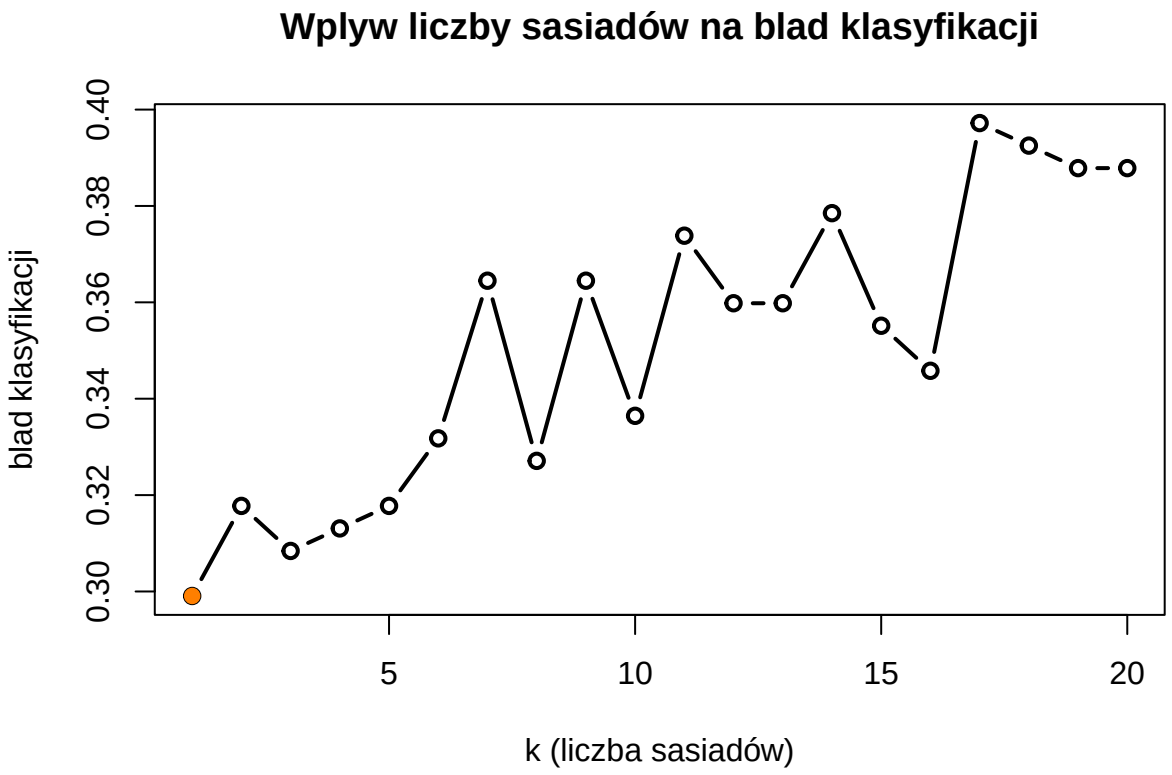
Błąd klasyfikacji na zbiorze uczącym: 0.267.

Błąd klasyfikacji na zbiorze testowym: 0.594.

Pomimo tego, że błąd na zbiorze uczącym dla modelu dla zmiennych o najlepszej dyskryminacji jest większy, to błąd na zbiorze testowym jest mniejszy. Z tego powodu uznajemy, że model dla zmiennych o najlepszej dyskryminacji jest lepszy.

3.1.6 Zaawansowane schematy oceny dokładności klasyfikacji

Poprzednia najoptymalniejsza wartość 5-najbliższych sąsiadów została ustalona dla konkretnego zbioru uczącego i testowego. W przypadku poniższych metod, bazujących na całym zbiorze danych `Glass.s`, musimy wyznaczyć nowe najoptymalniejsze k.



Rysunek 11: Wykres pozwalający wybrać najbardziej optymalną liczbą k najbliższych sąsiadów, używając metody cross-validation. Najoptymalniejsze k zostało zaznaczone kolorem pomarańczowym.

Nowa optymalna liczba sąsiadów wynosi 1.

3.1.7 Metoda *cross-validation* (10-krotna)

Metoda ta polega na wielokrotnym (tutaj 10-krotnym) podziale danych na zbiór uczący i testowy. Dla każdego podziału wyznaczany jest błąd klasyfikacji, a końcowy wynik to średnia wszystkich błędów.

Błąd klasyfikacji metodą *cross-validation* dla danych `Glass.s` wynosi 0.327.

3.1.8 Schemat *bootstrap* (50-krotny)

Metoda ta polega na wielokrotnym (tutaj 50-krotnym) losowaniu prób (ze zwracaniem) ze zbioru danych `Glass.s` i tworzeniu zbioru uczącego i testowego.

Błąd klasyfikacji metodą *bootstrap* dla danych `Glass.s` wynosi 0.369.

3.1.8.1 Schemat *bootstrap 632plus* (50-krotny) To odmiana metody *bootstrap*, która pozwala na uniknięcie przeuczenia modelu.

Błąd klasyfikacji metodą *632plus* dla danych `Glass.s` wynosi 0.313.

3.1.9 Wnioski cząstkowe

Tablica 2: Tabelka pozwalająca porównać wszystkie, wcześniej wyliczone, błędy klasyfikacji dla metody k-najbliższych sąsiadów.

Zestaw	Błąd_zbiór_uczący	Błąd_zbiór_testowy
14-najbliższych sąsiadów	0.293	0.500
Optymalna liczba k sąsiadów - 5	0.207	0.469
Zbiór cech o najlepszej zdolności dyskryminacyjnej	0.353	0.422
Zbiór cech o najgorszej zdolności dyksryminacyjnej	0.267	0.594
Metoda cross-validation	-	0.327
Schemat bootstrap	-	0.369
Schemat bootstrap 632plus	-	0.313

Porównując wszystkie wyznaczone w analizie błędy klasyfikacji (tabela **??**), widzimy, że dla metody k-najbliższych sąsiadów najmniejszy błąd klasyfikacji otrzymujemy, wykorzystując Schemat bootstrap 632plus i wynosi on 0.313. Natomiast największy, wynoszący Zbiór cech o najgorszej zdolności dyksryminacyjnej, wykorzystując 0.594.

Zastosowanie zaawansowanych schematów oceny dokładności miało zatem znaczący wpływ na nasz wniosek.

3.2 Drzewa klasyfikacyjne

W przypadku metody drzew klasyfikacyjnych stosowanie standaryzacji nie jest potrzebne, ponieważ działa ona na podstawie podziałów w oparciu o dane cechy (zwykle te, które najlepiej dyskryminują dane w danym węźle) i nie zależy od ich zakresu.

3.2.1 Analiza

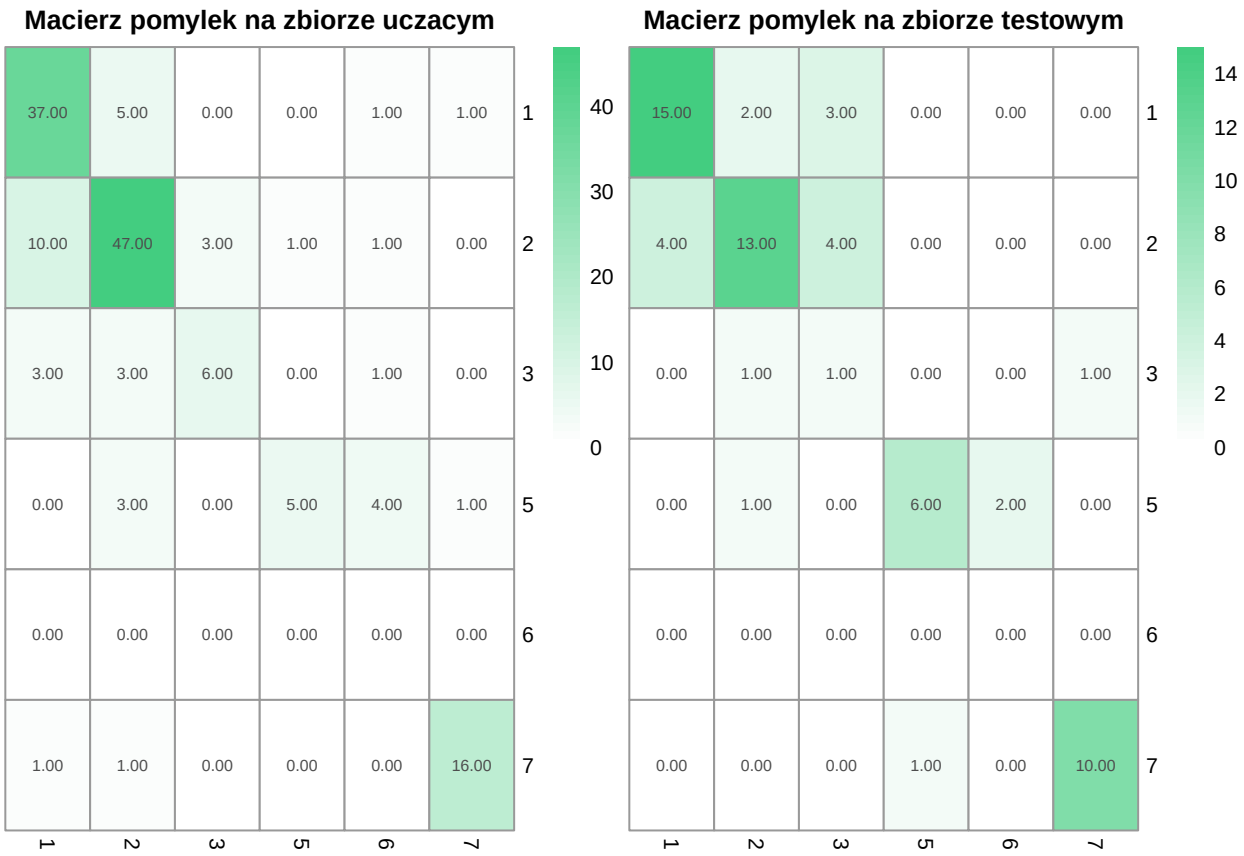
Tworzymy funkcję `tree.glass` dla argumentów:

- zbiór uczący,
- zbiór testowy,
- formuła (domyślnie wszystkie cechy),
- parametry tworzące drzewo (`cp`, `minsplit` i `maxdepth` ustawione na wartości domyślne),
- `model` (domyślnie `FALSE`) - określa, czy funkcja zwraca stworzony model drzewa,
- `wykres` (domyślnie `FALSE`) - określa, czy funkcja zwraca graficzną reprezentację drzewa.

Na podstawie tych argumentów buduje drzewo za pomocą funkcji `rpart` (z biblioteki `rpart`), a następnie zwraca macierze pomyłek oraz błędy klasyfikacji dla zadanych zbiorów lub (w zależności od argumentu `model`) stworzony model drzewa lub (w zależności od argumentu `wykres`) graficzną reprezentację drzewa.

```r
tree.glass <- function(uczący, testowy, formula = Type ~ .,
                       cp = 0.01, minsplit = 20, maxdepth = 30,
                       wykres = FALSE, model = FALSE) {
  # model <- Type ~ . formuła modelu

  # budujemy podstawowe drzewo
  Glass.tree <- rpart(formula, data = uczący,
                      control = rpart.control(cp = cp, minsplit = minsplit,
                                              maxdepth = maxdepth, model = TRUE))
  # dzięki model = TRUE model zostanie zapamiętany, co pozwoli zrobić wykres

  # etykietki rzeczywiste
  etykietki.rzecz.t <- testowy$Type
  etykietki.rzecz.u <- uczący$Type

  # prognozy dla zbioru uczącego
  etykietki.prog.u <- predict(Glass.tree, newdata = uczący, type = "class")

  # prognozy dla zbioru testowego
  etykietki.prog.t <- predict(Glass.tree, newdata = testowy, type = "class")

  # macierz pomyłek (confusion matrix)
  pomyłki.u <- table(etykietki.prog.u, etykietki.rzecz.u)
  pomyłki.t <- table(etykietki.prog.t, etykietki.rzecz.t)

  # błąd klasyfikacji (na zbiorze uczącym i testowy)
  n.u <- nrow(uczący)
  error.u <- (n.u - sum(diag(pomyłki.u))) / n.u

  n.t <- nrow(testowy)
  error.t <- (n.t - sum(diag(pomyłki.t))) / n.t

  if (wykres == TRUE) { # jeśli chcemy rysować wykres
    return(rpart.plot(Glass.tree))
  } else if (model == TRUE) {
    return(Glass.tree)
  } else {
    return((list(pomyłki.u, error.u, pomyłki.t, error.t)))
  }
}
```

Testujemy teraz naszą funkcję na wygenerowanym wcześniej zbiorze uczącym i testowym.

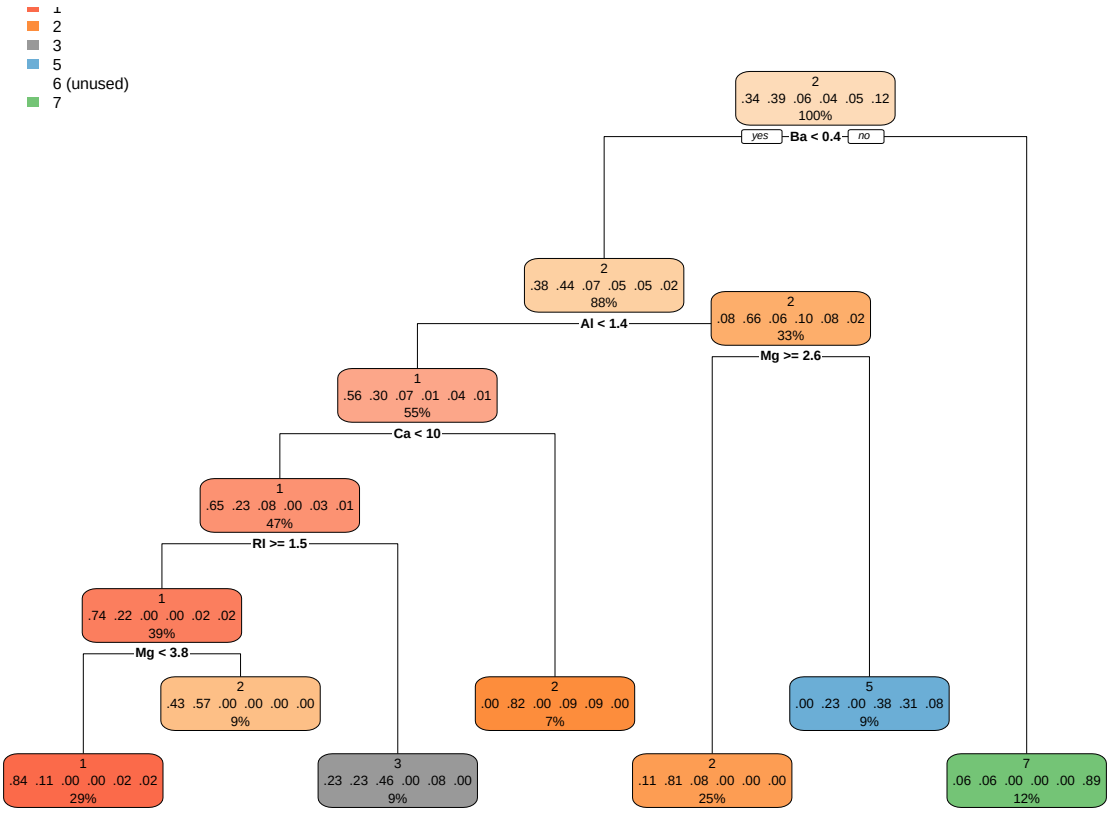


Rysunek 12: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla domyślnych parametrów modelu drzewa klasyfikacji.

Błąd klasyfikacji na zbiorze uczącym: 0.26.

Błąd klasyfikacji na zbiorze testowym: 0.297.

Zwizualizujemy teraz nasz model.



Rysunek 13: Drzewo klasyfikacji na podstawie stworzonego modelu dla domyślnych parametrów cp, minsplit i maxdepth.

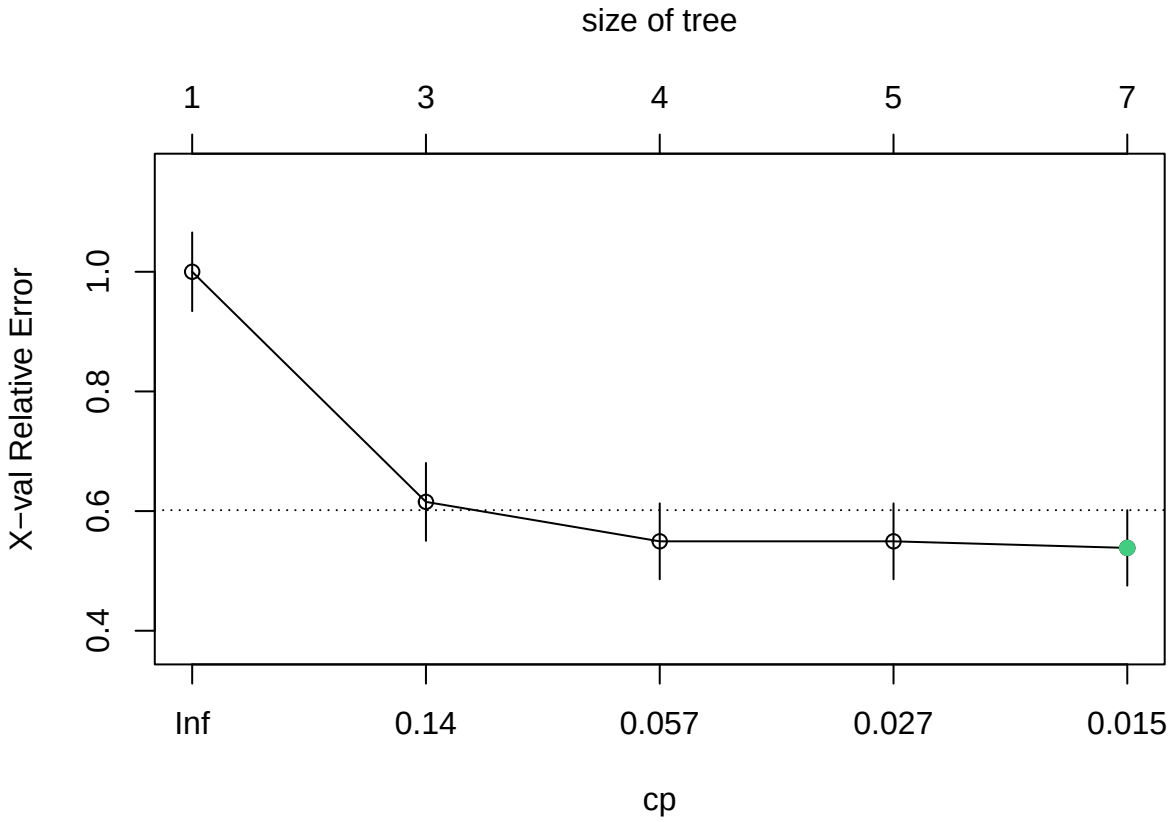
Nie wszystkie klasy są widoczne w liściach drzewa. Może być to spowodowane zbyt małą ilością elementów w tej klasie lub brakiem wyraźnie dyskryminującej cechy.

3.2.2 Różne parametry a dokładność klasyfikacji

W praktyce wszystkie trzy parametry, dla których dokładność klasyfikacji będziemy za moment testować, są od siebie zależne. Jednak stworzenie trójwymiarowego wykresu, uwzględniającego je wszystkie, wymagałoby ogromnej mocy obliczeniowej oraz czasu, dlatego zdecydowałyśmy się na ich podział. Najpierw wyznaczymy najoptymalniejsze `cp`, dla domyślnych wartości parameterów `minsplit` oraz `maxdepth`, a następnie wyznaczymy na wspólnym wykresie ich optymalne wartości dla wyliczonego `cp`.

3.2.3 `cp`

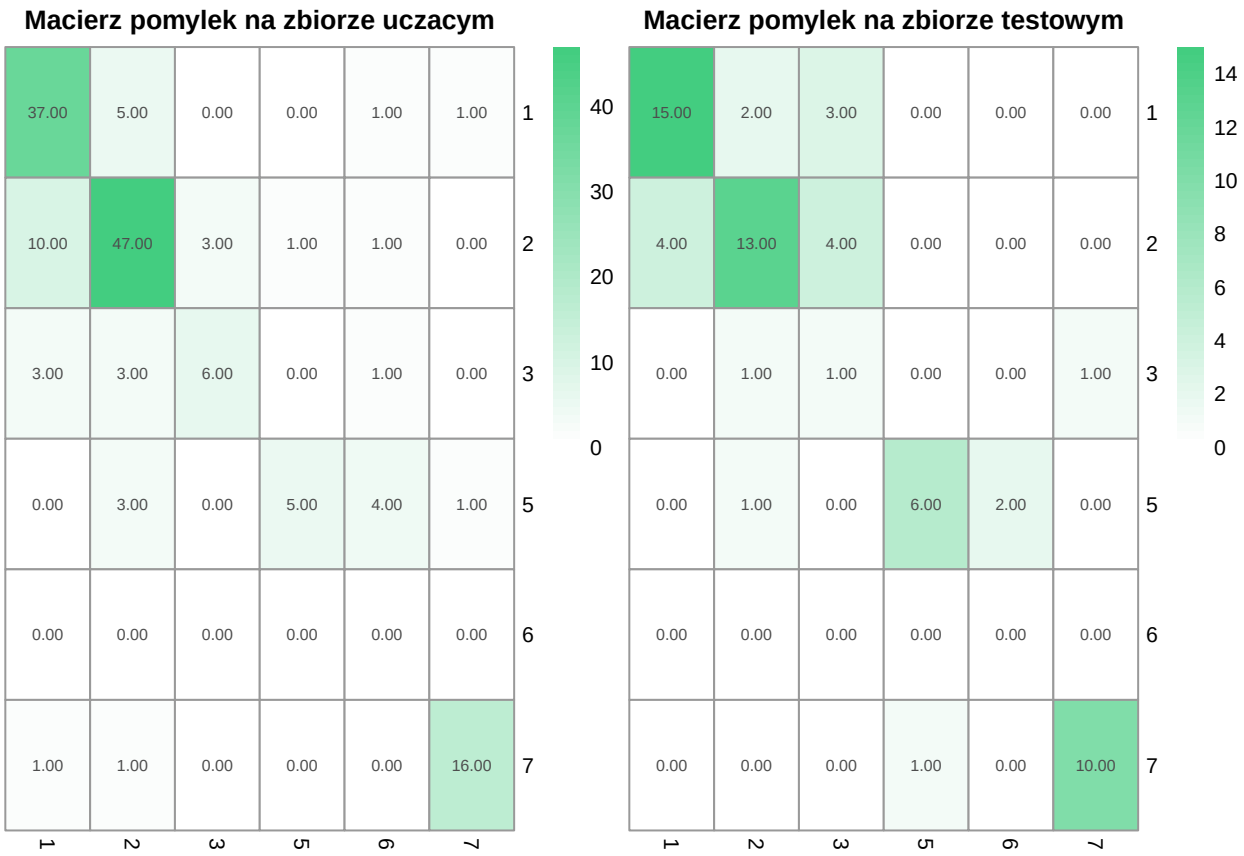
Badamy teraz wpływ wielkości czynnika `cp` (odpowiedzialnego za liczbę gałęzi (podziałów) naszego drzewa) na dokładność klasyfikacji dla domyślnych wartości parameterów `minsplit` oraz `maxdepth`.



Rysunek 14: Wykres pozwalający wybrać najoptymalniejszą wartość parametru cp, zaznaczonym kolorem morskozielonym.

Najoptymalniejsze `cp`, to te, które są na i pod linią przerywaną (minimalny błąd plus odpowiadające mu odchylenie standardowe) na wykresie 14. Preferowana wartość to ta, dla której dostaniemy najmniejsze drzewo. W naszym przypadku nie jest to jednak najoptymalniejszy błąd (wykonałyśmy testy), dlatego jako najoptymalniejsze `cp` wybieramy to, dla którego błąd jest najmniejszy. Ta wartość to 0.01.

Teraz, w oparciu o wyliczone `cp`, wyznaczymy macierze pomyłek i błędy klasyfikacji dla zbioru uczącego i testowego.



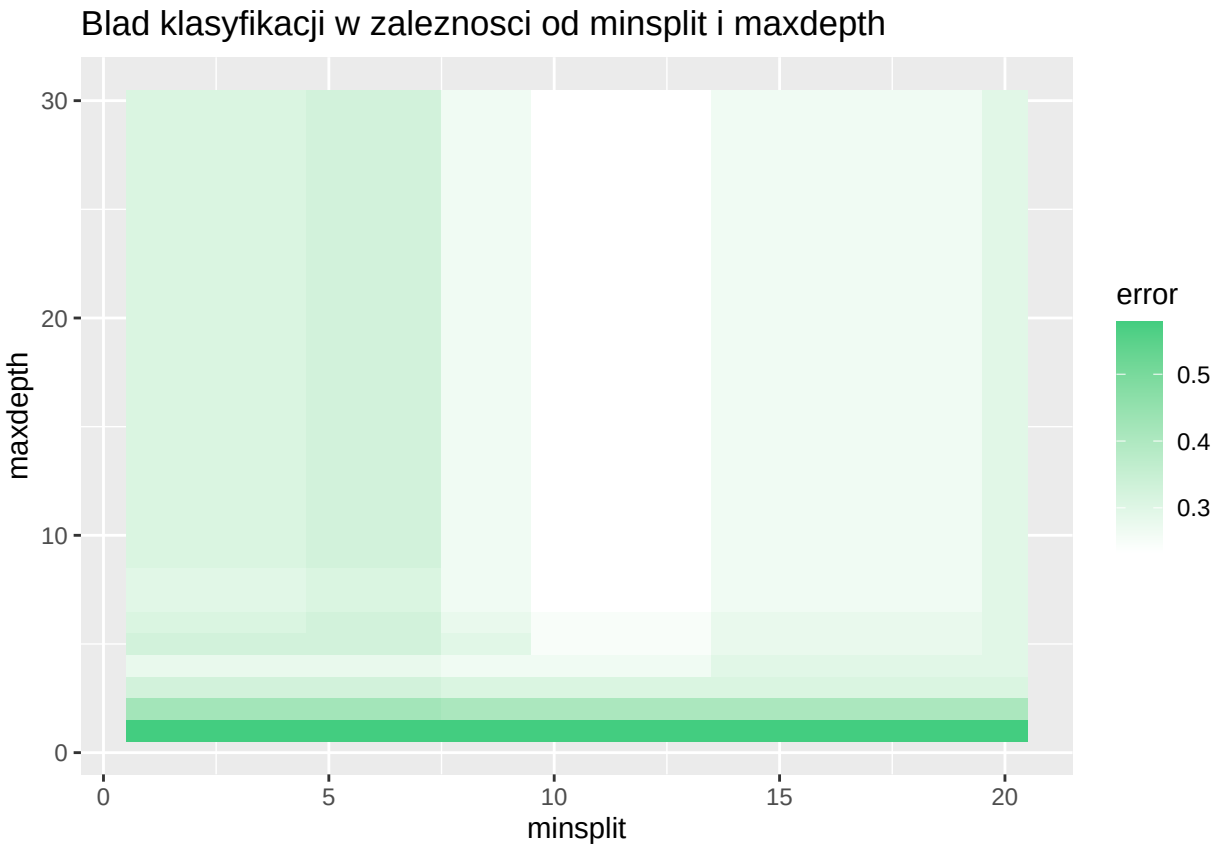
Rysunek 15: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla najoptymalniejszej wartości parametru cp.

Błąd klasyfikacji na zbiorze uczącym: 0.26.

Błąd klasyfikacji na zbiorze testowym: 0.297.

3.2.4 `minsplit` oraz `maxdepth`

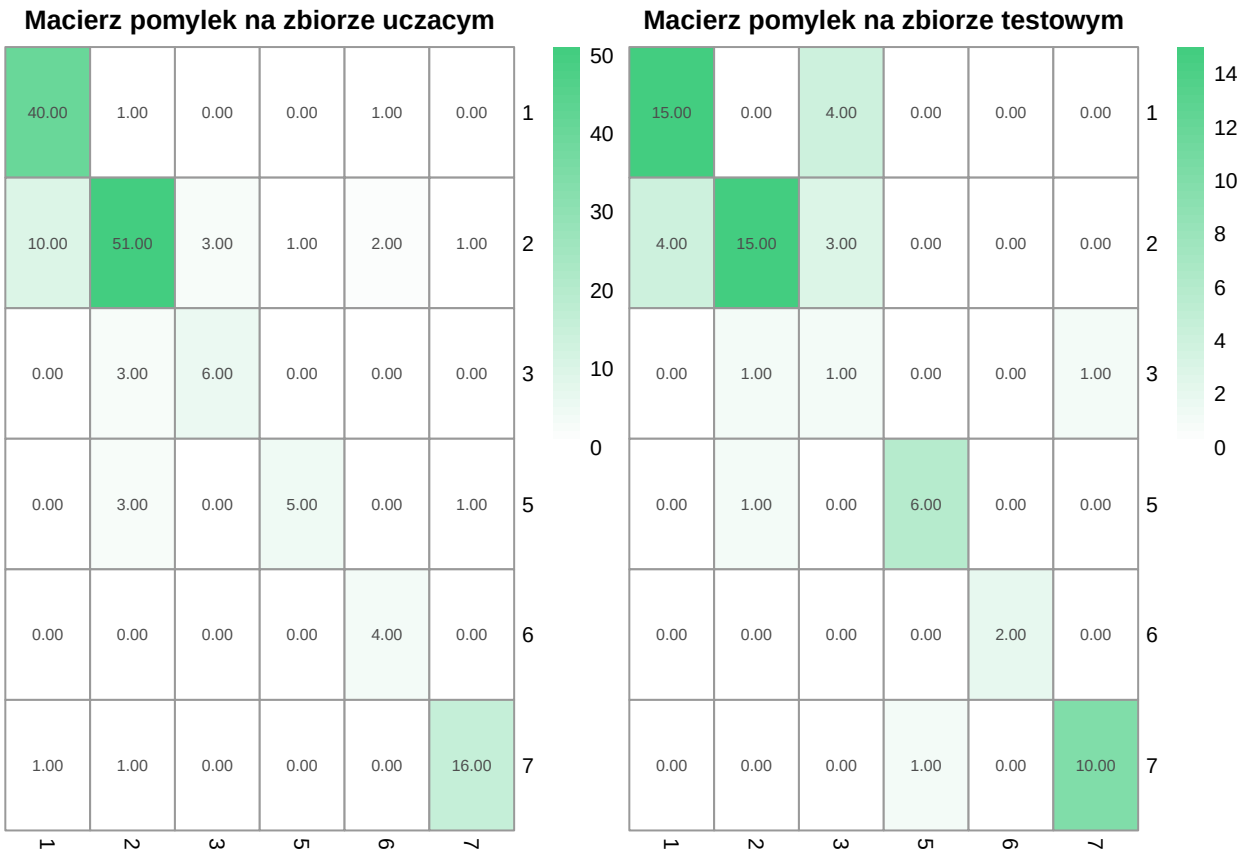
Dla parametrów `minsplit` (minimalna liczba obserwacji w węźle do rozważenia podziału) oraz `maxdepth` (maksymalna liczba poziomów drzewa) zadajemy przedziały [1, 20] oraz [1, 30] odpowiednio i sprawdzamy błąd dla optymalnego `cp`. Wybieramy najmniejsze możliwe wartości dla minimalnego błędu.



Rysunek 16: Wykres przedstawiający zmianę wartości błędu klasyfikacji, przy zmianie parametrów minsplit oraz maxdepth.

Jak widać na wykresie 16, dla optymalnej wartości parametru `cp` najmniejszy błąd otrzymujemy dla `minsplit` równego 10 oraz `maxdepth` równego 7.

Ponownie wyliczamy macierze pomyłek i błędy klasyfikacji dla wyznaczonych parametrów.

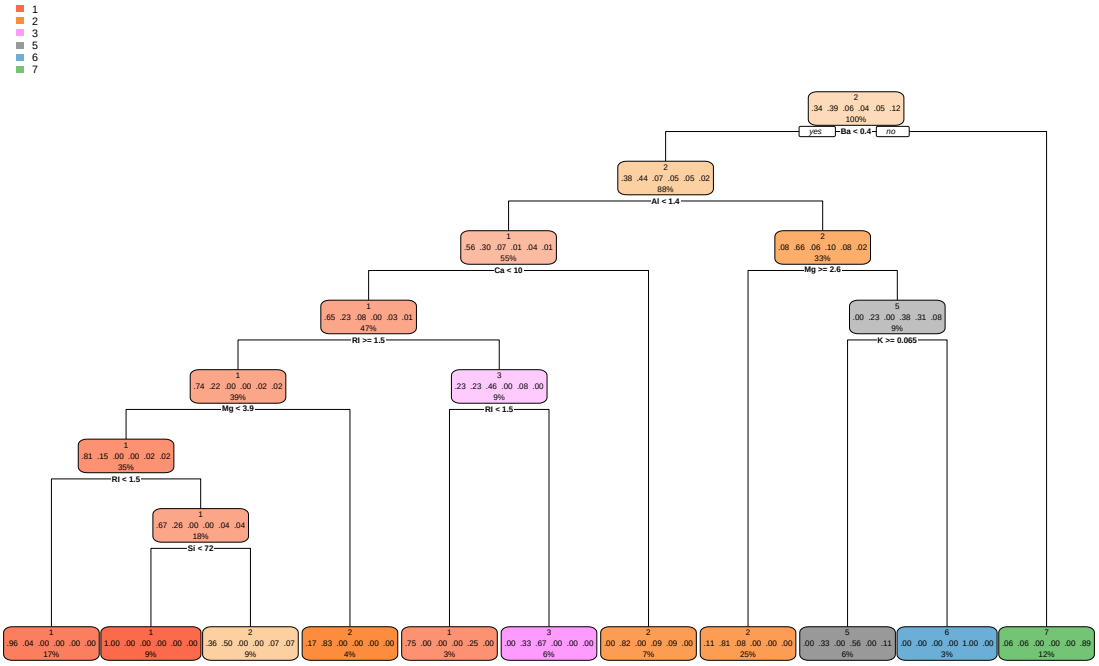


Rysunek 17: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla najoptymalniejszych wartości parametrów cp, minsplit i maxdepth.

Błąd klasyfikacji na zbiorze uczącym: 0.187.

Błąd klasyfikacji na zbiorze testowym: 0.234.

Zwizualizujemy jeszcze nasz model dla najoptymalniejszych parametrów.



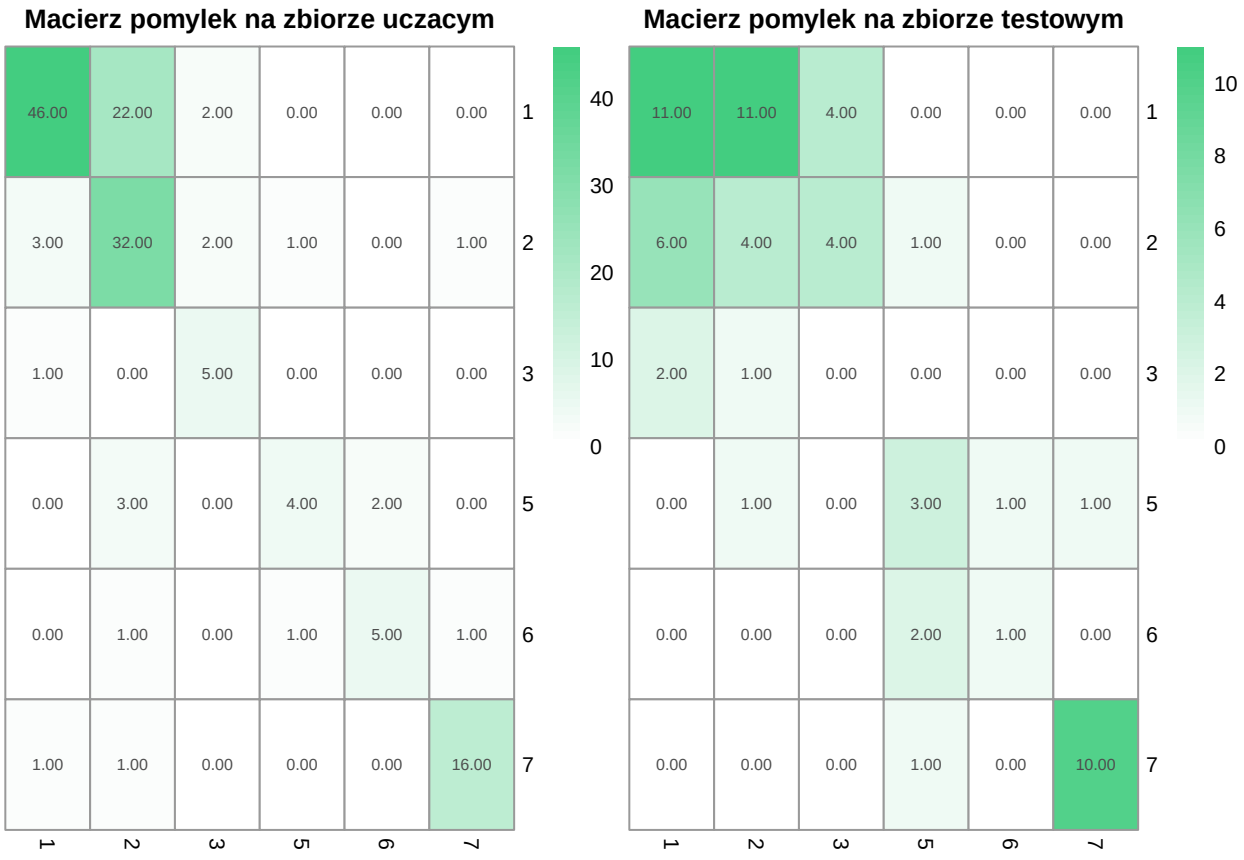
Rysunek 18: Drzewo klasyfikacji na podstawie stowrzonego modelu dla najoptymalniejszych parametrów cp, minsplit i maxdepth.

Nie wszystkie klasy są widoczne w liściach drzewa. Może być to spowodowane zbyt małą ilością elementów w tej klasie lub brakiem wyraźnie dyskryminującej cechy.

3.2.5 Dyskryminacja

Testujemy teraz nasz algorytm na modelach opartych o podzbiory zmiennych o najlepszej i najgorszej zdolności dyskryminacyjnej dla najoptymalniejszych parametrów.

3.2.6 Zbiór zmiennych o najlepszej zdolności dyskryminacyjnej

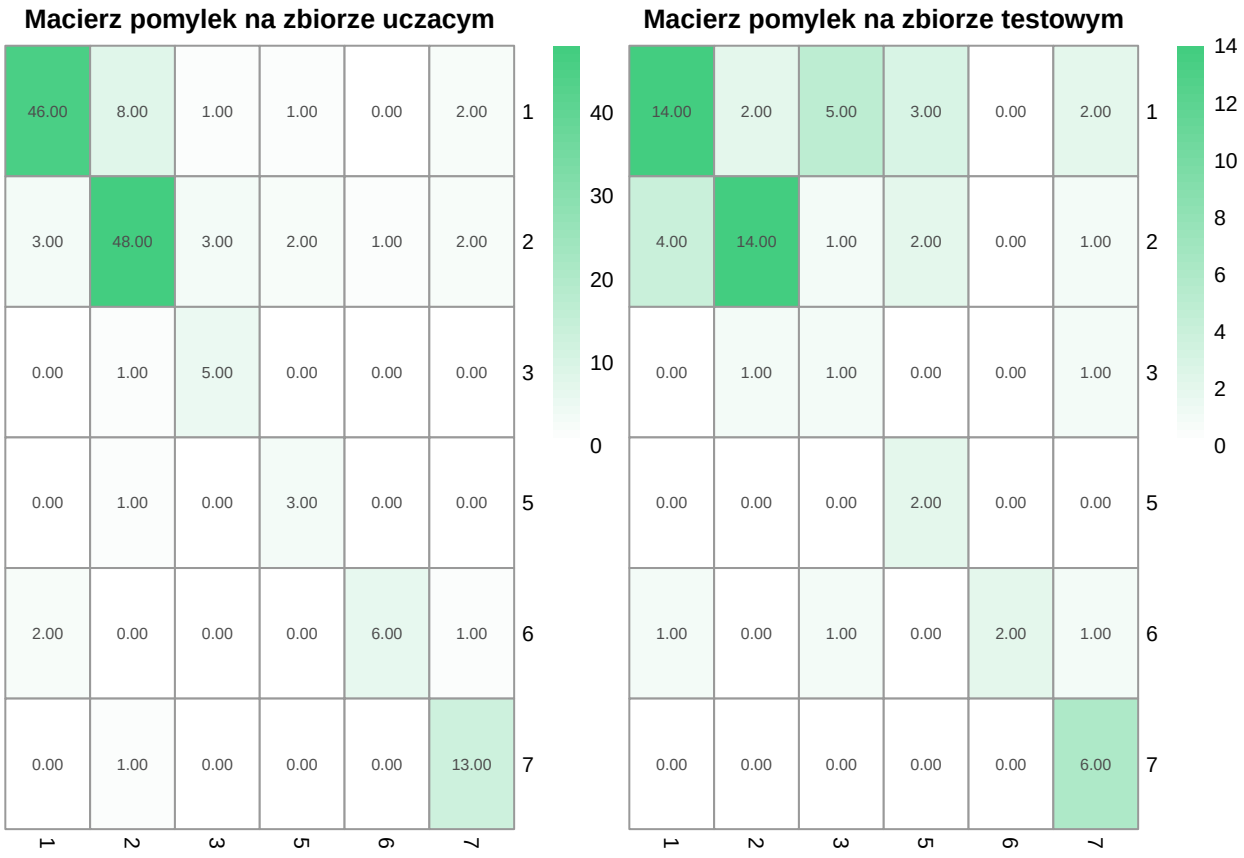


Rysunek 19: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla zmiennych o najlepszej zdolności dyskryminacyjnej dla najoptymalniejszych parametrów cp, minsplit oraz maxdepth.

Błąd klasyfikacji na zbiorze uczącym: 0.28.

Błąd klasyfikacji na zbiorze testowym: 0.547.

3.2.7 Zbiór zmiennych o najgorszej zdolności dyskryminacyjnej



Rysunek 20: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla zmiennych o najgorszej zdolności dyskryminacyjnej dla najoptymalniejszych parametrów cp, minsplit oraz maxdepth.

Błąd klasyfikacji na zbiorze uczącym: 0.193.

Błąd klasyfikacji na zbiorze testowym: 0.391.

Co zaskakujące model nauczył się rozróżniać przypadki lepiej, ucząc się na danych o najgorszej zdolności dyskryminacyjnej - wynik jest lepszy, niż w przypadku zmiennych o najlepszej zdolności dyskryminacyjnej.

3.2.8 Zaawansowane schematy oceny dokładności

3.2.9 Metoda *cross-validation* (10-krotna)

Błąd klasyfikacji metodą *cross-validation* dla danych `Glass` wynosi 0.304.

3.2.10 Schemat *bootstrap* (50-krotny)

Błąd klasyfikacji metodą *bootstrap* dla danych `Glass` wynosi 0.341.

3.2.10.1 Schemat *bootstrap 632plus* (50-krotny) Błąd klasyfikacji metodą *632plus* dla danych `Glass` wynosi 0.295.

3.2.11 Wnioski cząstkowe

Tu stwierdzamy, która metoda była najlepsza - jej błąd klasyfikacji był najmniejszy.

Tablica 3: Tabelka pozwalająca porównać wszystkie, wcześniej wyliczone, błędy klasyfikacji dla drzew klasyfikacji.

Zestaw	Błąd_zbiór_uczący	Błąd_zbiór_testowy
Model podstawowy - bez modyfikacji	0.26	0.297
Optymalne parametry cp, minsplit i maxdepth	0.187	0.234
Zbiór cech o najlepszej zdolności dyskryminacyjnej	0.28	0.547
Zbiór cech o najgorszej zdolności dyskryminacyjnej	0.193	0.391
Metoda cross-validation	-	0.304
Schemat bootstrap	-	0.341
Schemat bootstrap 632plus	-	0.295

Porównując wszystkie wyznaczone w analizie błędy klasyfikacji (tabela **??**), widzimy, że dla drzew klasyfikacji najmniejszy błąd klasyfikacji otrzymujemy, wykorzystując Optymalne parametry cp, minsplit i maxdepth i wynosi on 0.234. Natomiast największy, wynoszący Zbiór cech o najlepszej zdolności dyskryminacyjnej, wykorzystując 0.547.

Tutaj zastosowanie zaawansowanych schematów oceny dokładności nie miało znaczącego wpływu na nasz wniosek.

3.3 Naiwny klasyfikator bayesowski

Dla metody wykorzystującej naiwny klasyfikator bayesowski standaryzacja również nie jest potrzebna. Algorytm oparty jest na maksymalizacji prawdopodobieństwa warunkowego lub gęstości prawdopodobieństwa (jak w naszym przypadku), a nie na odległości między elementami.

3.3.1 Analiza

Tworzymy funkcję `bayes.glass` dla argumentów:

- zbiór uczący,
- zbiór testowy,
- formuła (domyślnie wszystkie zmienne),
- parametr `prior` (prawdopodobieństwo prior klas ze zbioru),
- `parametryczny` (domyślnie `TRUE`) - określa, czy model ma być parametryczny,
- `wykres` (domyślnie `FALSE`) - określa, czy funkcja zwraca wykresy gęstości prawdopodobieństwa poszczególnych cech z danych `Glass`.

Na ich postawie wyznaczamy model, wykorzystując funkcję `NaiveBayes` z pakietu `klaR` (dla modelu nieparametrycznego) lub funkcję `naiveBayes` z pakietu `e1071` (dla modelu parametrycznego), i zwracamy macierze pomyłek oraz błędy klasyfikacji lub, w przypadku modelu nieparametrycznego, (w zależności od argumentu `wykres`) wykres.

Tworzenie wykresu dla modelu parametrycznego dla danych `Glass` jest niemożliwe, ponieważ dla jednej z klas jedna ze zmiennych przyjmuje stałą wartość równą 0, z czym funkcja `NaiveBayes` nie jest w stanie sobie poradzić. Dlatego w przypadku modelu parametrycznego korzystamy z funkcji `naiveBayes`, która z kolei zwraca obiekt niemożliwy do przedstawienia na wykresie.

```r
bayes.glass <- function(uczący, testowy, formula = Type ~ .,
                        p = NULL, parametryczny = TRUE,
                        wykres = FALSE) {

  if (parametryczny == TRUE & is.null(p) == FALSE) {
    stop("Błąd. Nie można zadać prawdopodobieństwa dla modelu parametrycznego.")
  } else {

    # model na zbiorze uczącym
    if (parametryczny == TRUE) { # czyli chcemy model parametryczny
      model.bayes <- naiveBayes(formula, data = uczący)
      # prognozowane etykietki na zbiorze uczącym
      etykietki.prog.u <- predict(model.bayes, newdata = uczący)
      # prognozowane etykietki na zbiorze testowym
      etykietki.prog.t <- predict(model.bayes, newdata = testowy)

    } else {
      model.bayes <- NaiveBayes(formula, data = uczący, usekernel = TRUE, prior = p)
      # prognozowane etykietki na zbiorze uczącym
      etykietki.prog.u <- predict(model.bayes, newdata = uczący)$class
      # prognozowane etykietki na zbiorze testowym
      etykietki.prog.t <- predict(model.bayes, newdata = testowy)$class
    }

    # etykietki rzeczywiste
    etykietki.rzecz.t <- testowy$Type
    etykietki.rzecz.u <- uczący$Type

    # macierz pomyłek
    pomyłki.u <- table(etykietki.prog.u, etykietki.rzecz.u)
    pomyłki.t <- table(etykietki.prog.t, etykietki.rzecz.t)

    # błąd klasyfikacji
    n.u <- nrow(uczący)
    error.u <- (n.u - sum(diag(pomyłki.u))) / n.u

    n.t <- nrow(testowy)
    error.t <- (n.t - sum(diag(pomyłki.t))) / n.t

    if (wykres == TRUE & parametryczny == FALSE) {
      return(plot(model.bayes))
    } else {
      return(list(pomyłki.u, error.u, pomyłki.t, error.t))
    }
  }
}
```

Testujemy funkcję dla wygenerowanego wcześniej zbioru uczącego i testowego. Dzięki temu będziemy w stanie stwierdzić, który model, parametryczny czy nieparametryczny, jest lepszy dla naszego zbioru danych i to na nim będziemy się opierać w dalszej analizie.

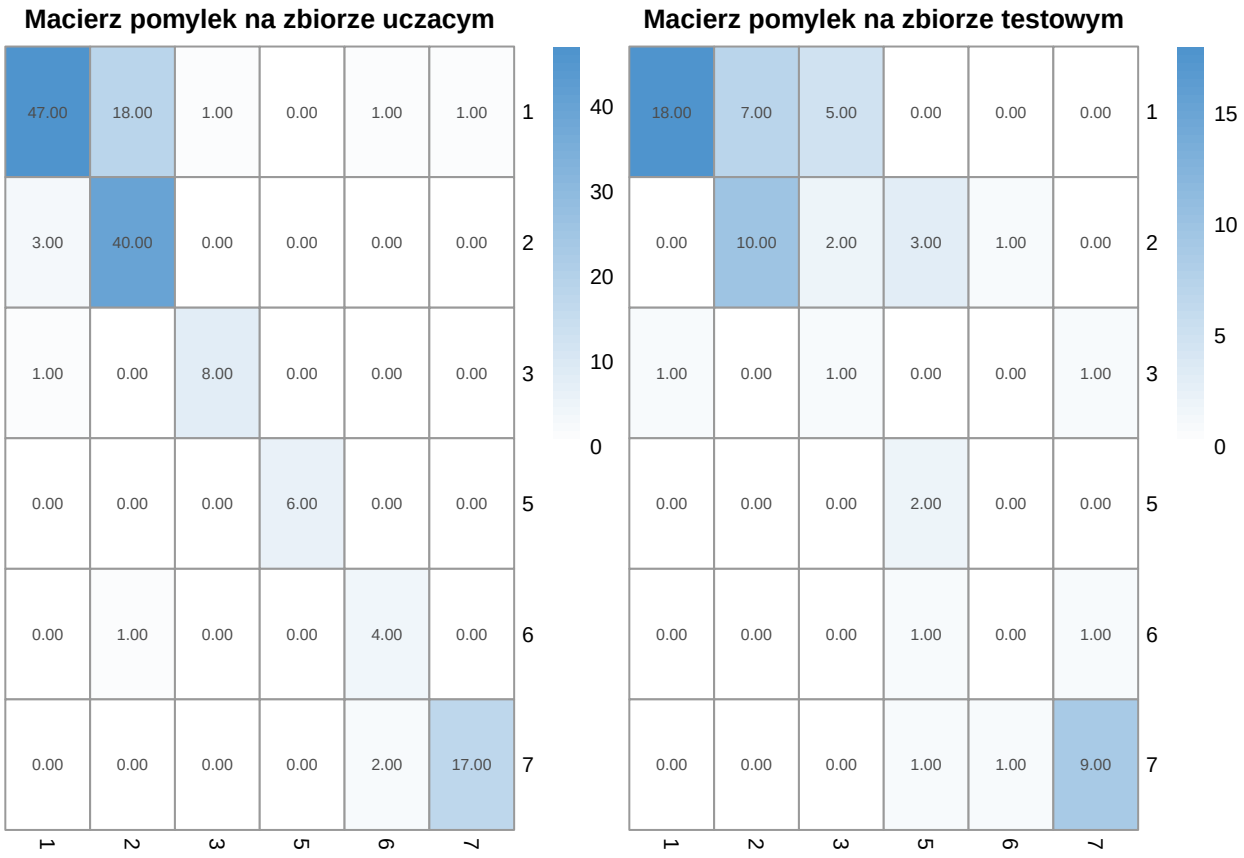
3.3.2 Model parametryczny

Rysunek 21: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla modelu parametrycznego dla naiwnego klasyfikatora bayesowskiego.

Błąd klasyfikacji na zbiorze uczącym: 0.533.

Błąd klasyfikacji na zbiorze testowym: 0.656.

3.3.3 Model nieparametryczny



Rysunek 22: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla modelu nieparametrycznego dla naiwnego klasyfikatora bayesowskiego.

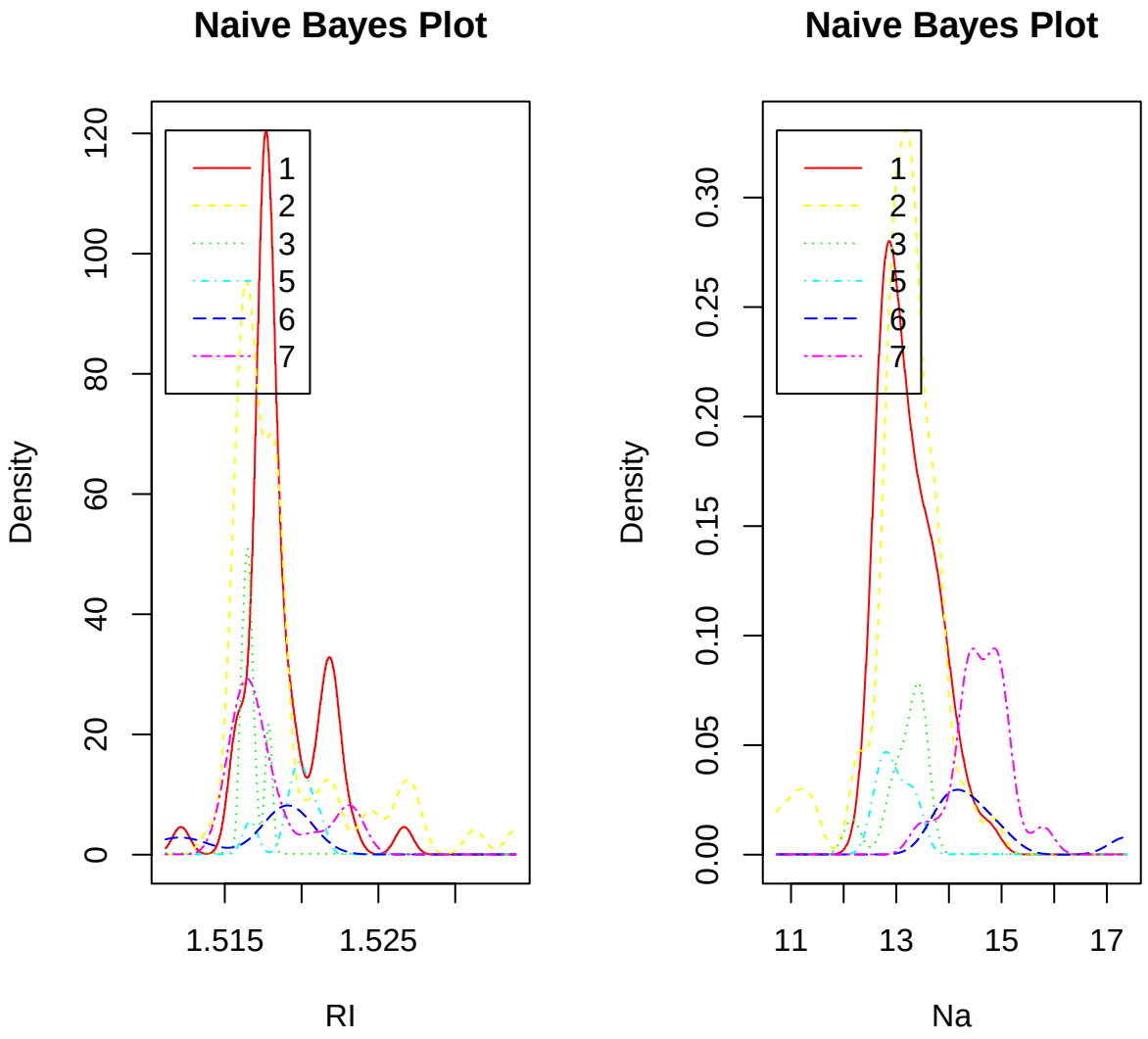
Błąd klasyfikacji na zbiorze uczącym: 0.187.

Błąd klasyfikacji na zbiorze testowym: 0.375.

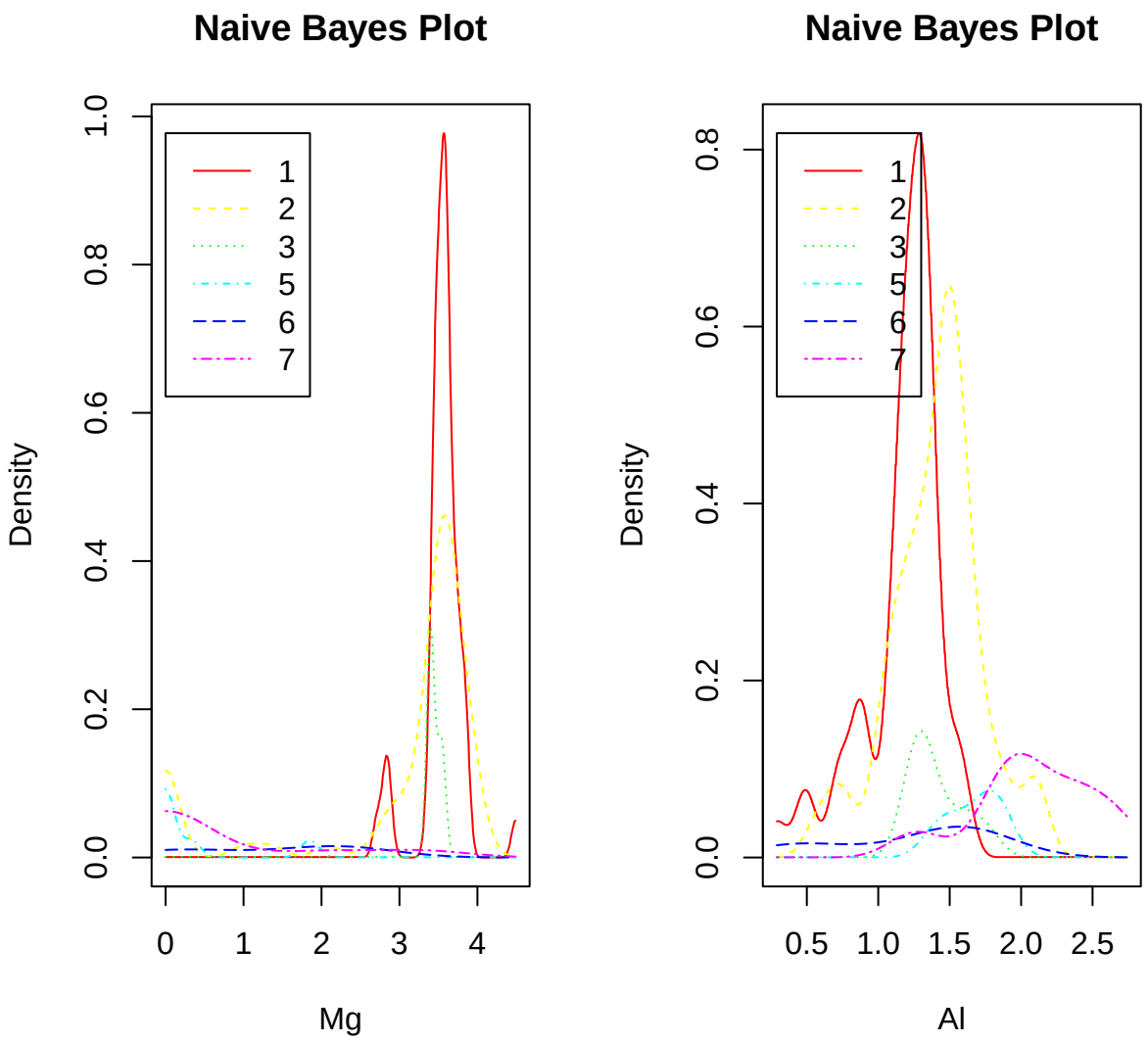
Model nieparametryczny jest lepszy - jego błędy klasyfikacji na zbiorze uczącym i testowym są mniejsze, niż w przypadku modelu parametrycznego.

3.3.4 Dyskryminacja

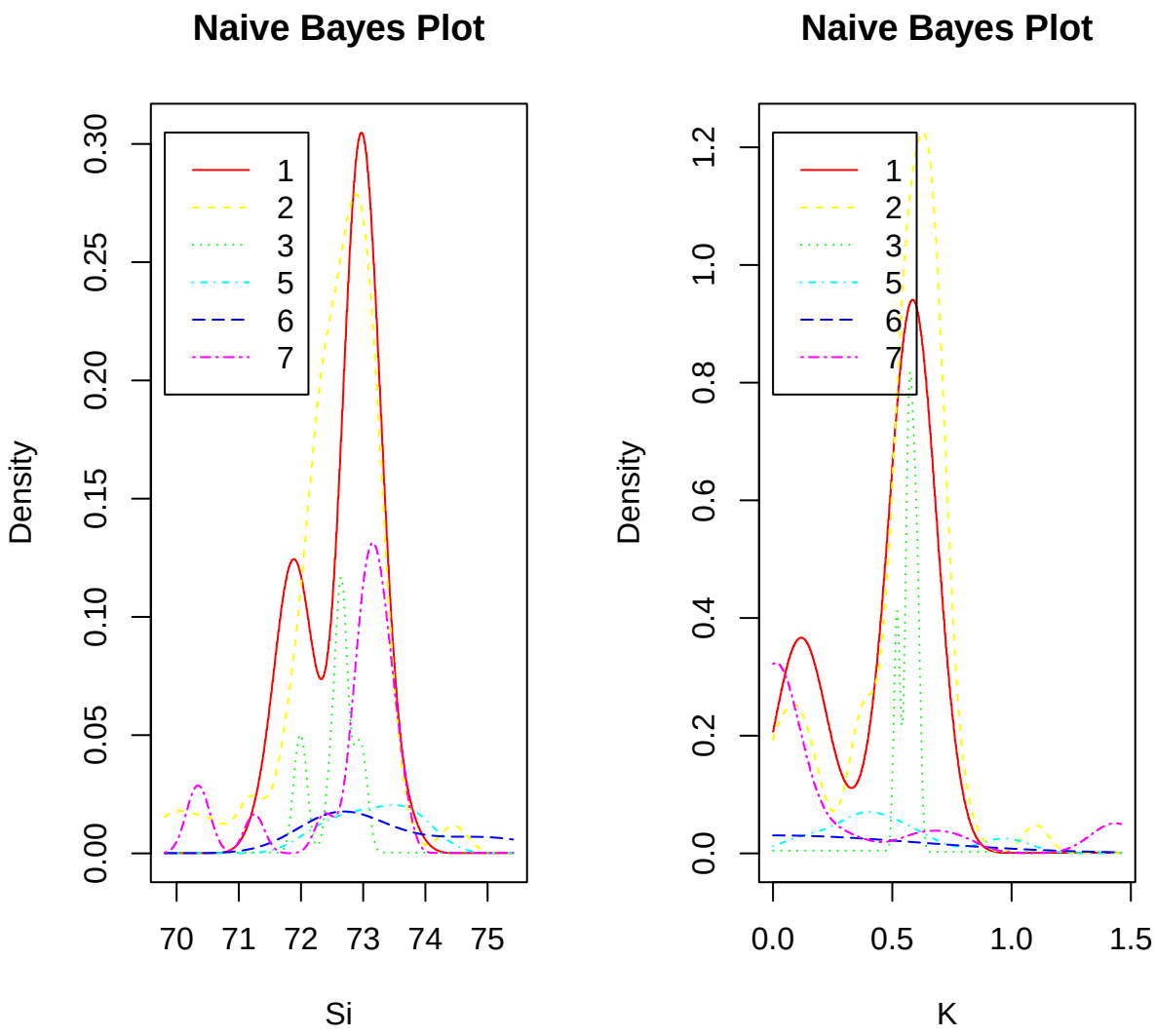
Upewniając się, że wcześniejsze zbiory zmiennych o najlepszej i najgorszej dyskryminacji zostały poprawnie wybrane, przedstawimy teraz wykresy gęstości prawdopodobieństwa, dla poszczególnych cech.



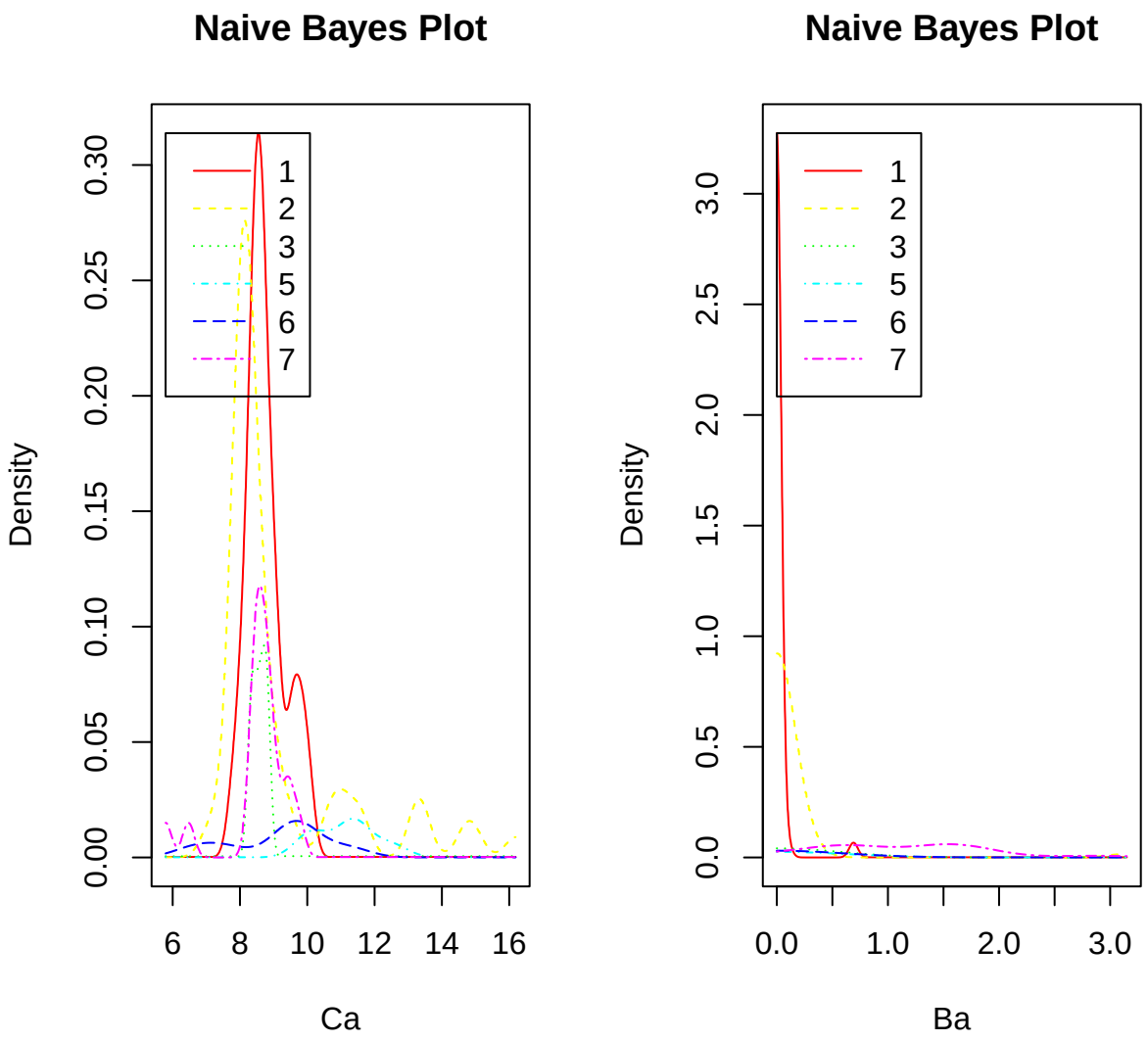
Rysunek 23: Wykresy gęstości prawdopodobieństwa dla modelu nieparametrycznego na zbiorze uczącym.



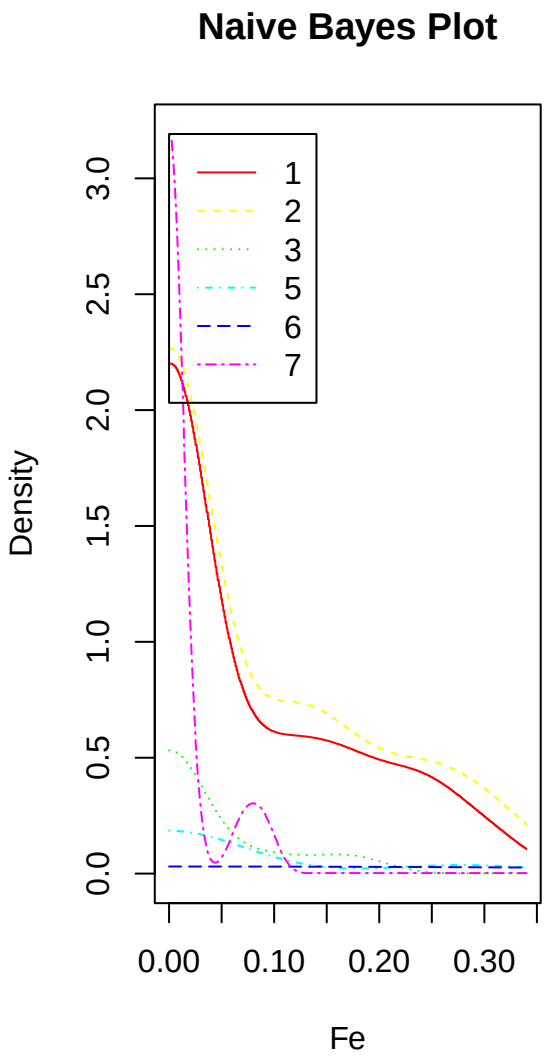
Rysunek 24: Wykresy gęstości prawdopodobieństwa dla modelu nieparametrycznego na zbiorze uczącym.



Rysunek 25: Wykresy gęstości prawdopodobieństwa dla modelu nieparametrycznego na zbiorze uczącym.



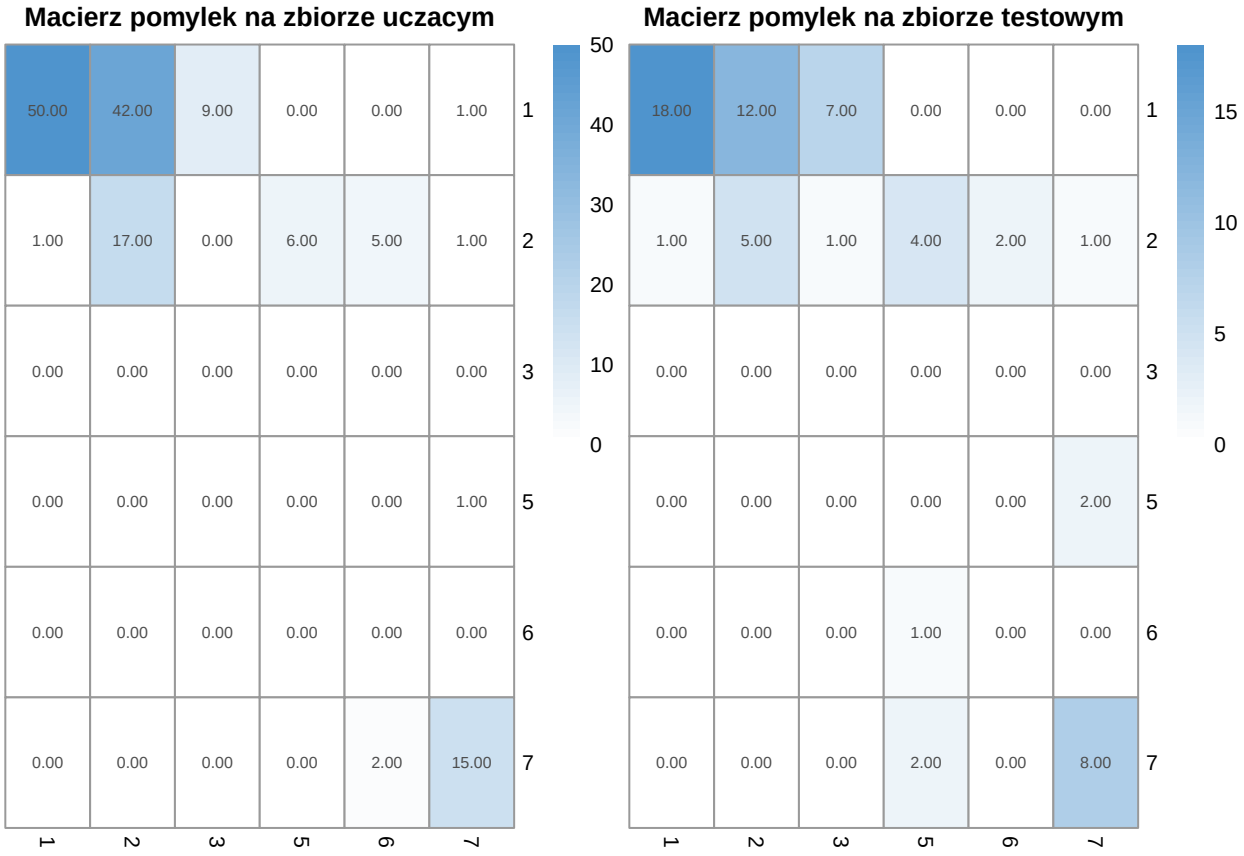
Rysunek 26: Wykresy gęstości prawdopodobieństwa dla modelu nieparametrycznego na zbiorze uczącym.



Rysunek 27: Wykresy gęstości prawdopodobieństwa dla modelu nieparametrycznego na zbiorze uczącym.

Jak widać na wykresach 27 zmienne zostały wcześniej poprawnie wybrane, dlatego przestestujemy teraz nasz algorym dla tych podzbiorów w oparciu o model nieparametryczny.

3.3.5 Zbiór zmiennych o najlepszej zdolności dyskryminacyjnej

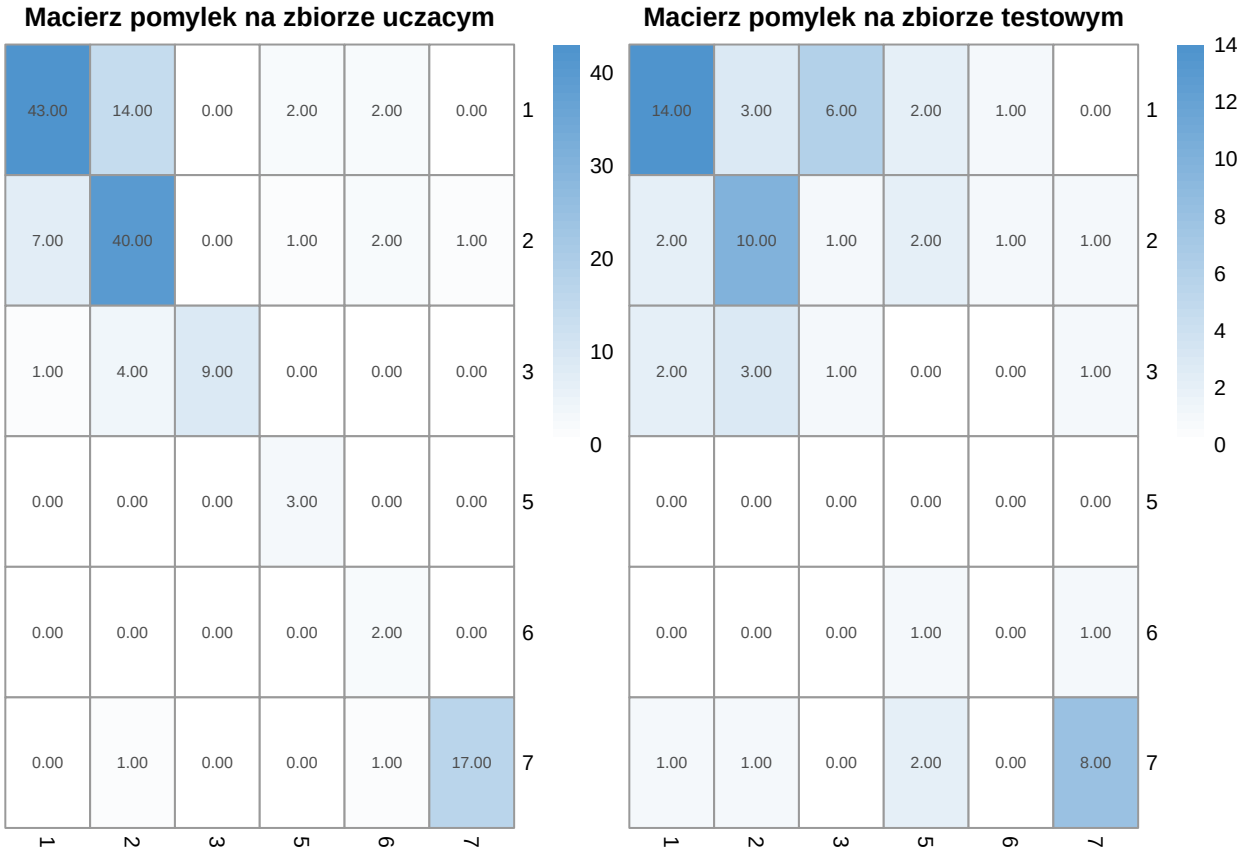


Rysunek 28: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla zmiennych o najlepszej zdolności dyskryminacyjnej dla modelu nieparametrycznego.

Błąd klasyfikacji na zbiorze uczącym: 0.453.

Błąd klasyfikacji na zbiorze testowym: 0.516.

3.3.6 Zbiór zmiennych o najgorszej zdolności dyskryminacyjnej



Rysunek 29: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla zmiennych o najgorszej zdolności dyskryminacyjnej dla modelu nieparametrycznego.

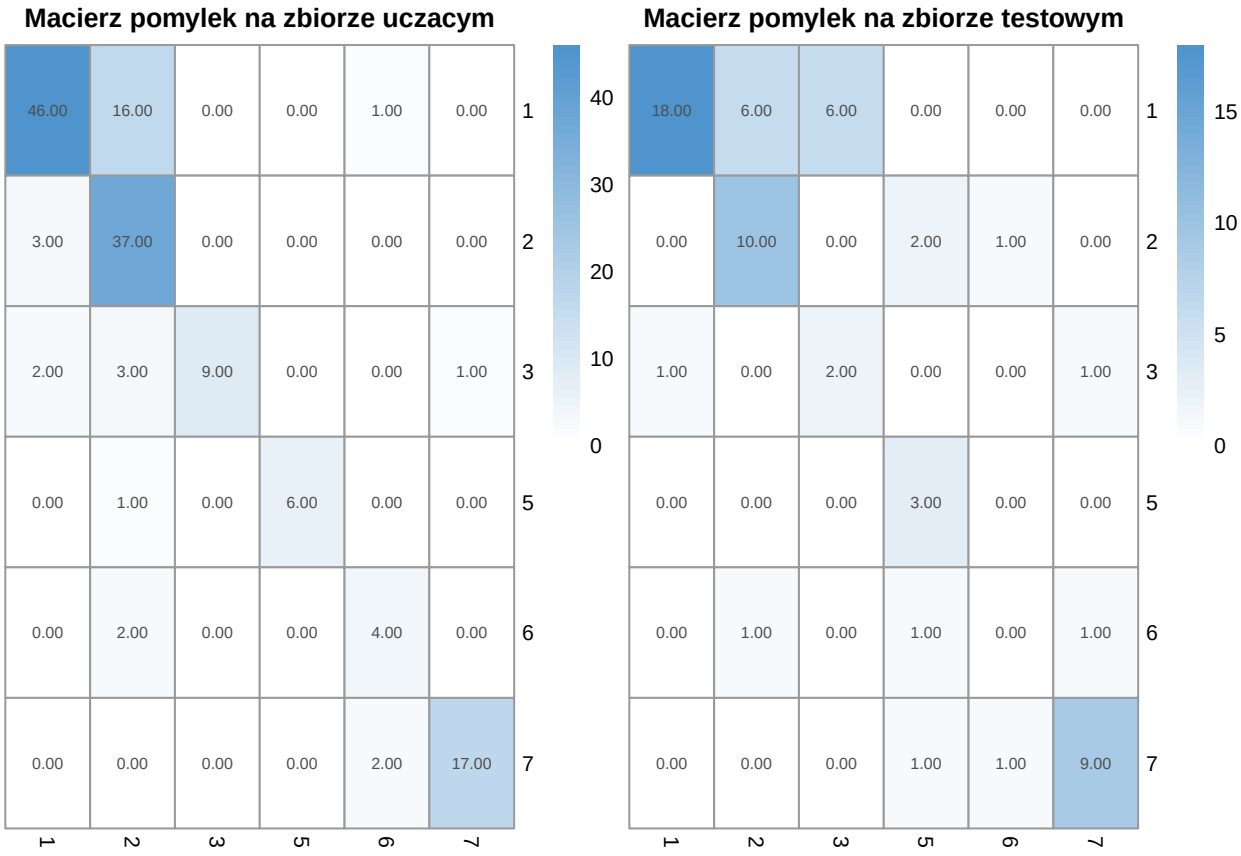
Błąd klasyfikacji na zbiorze uczącym: 0.24.

Błąd klasyfikacji na zbiorze testowym: 0.484.

Co zaskakujące model nauczył się rozróżniać przypadki lepiej, ucząc się na danych o najgorszej zdolności dyskryminacyjnej - wynik jest lepszy, niż w przypadku zmiennych o najlepszej zdolności dyskryminacyjnej.

3.3.7 Prawdopodobieństwo występowania klas a dokładność klasyfikacji

Ze względu na nierównomierne rozłożenie przypadków (wykres 2) postanowiłyśmy sprawdzić, czy ustawienie takiego samego prawdopodobieństwa dla wszystkich klas sprawi, że przypadki będą lepiej klasyfikowane.



Rysunek 30: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla modelu nieparametrycznego i występowaniem klas z jednakowym prawdopodobieństem.

Błąd klasyfikacji na zbiorze uczącym: 0.207.

Błąd klasyfikacji na zbiorze testowym: 0.344.

Pomimo tego, że błąd na zbiorze uczącym dla modelu z jednakowym prawdopodobieństwem wszystkich klas jest większy, to błąd na zbiorze testowym jest mniejszy. Dlatego model bez zmiany prawdopodobieństwa jest gorszy.

3.3.8 Zaawansowane schematy oceny dokładności

Dla tych schematów ingerencja w prawdopodobieństwo klas mogłaby nie przynieść oczekiwanych rezultatów (mniejszy błąd klasyfikacji), dlatego postanawiamy je pominąć. Wykorzystujemy model nieparametryczny.

3.3.9 Metoda *cross-validation* (10-krotna)

Błąd klasyfikacji metodą *cross-validation* dla danych `Glass` wynosi 0.411.

3.3.10 Schemat *bootstrap* (50-krotny)

Błąd klasyfikacji metodą *bootstrap* dla danych `Glass` wynosi 0.418.

3.3.10.1 Schemat *bootstrap 632plus* (50-krotny)

Błąd klasyfikacji metodą *632plus* dla danych `Glass` wynosi 0.369.

3.3.11 Wnioski cząstkowe

Tablica 4: Tabelka pozwalająca porównać wszystkie, wcześniej wyliczone, błędy klasyfikacji dla metody opartej o naiwny klasyfikator bayesowski.

Zestaw	Błąd_zbiór_uczący	Błąd_zbiór_testowy
Model parametryczny	0.533	0.656
Model nieparametryczny	0.187	0.375
Zbiór cech o najlepszej zdolności dyskryminacyjnej	0.453	0.516
Zbiór cech o najgorszej zdolności dyskryminacyjnej	0.24	0.484
Jednakowe prawdopodobieństwo klas	0.207	0.344
Metoda cross-validation	-	0.411
Schemat bootstrap	-	0.418
Schemat bootstrap 632plus	-	0.369

Porównując wszystkie wyznaczone w analizie błędy klasyfikacji (tabela **??**), widzimy, że dla metody opartej o naiwny klasyfikator bayesowski najmniejszy błąd klasyfikacji otrzymujemy, wykorzystując Jednakowe prawdopodobieństwo klas i wynosi on 0.344. Natomiast największy, wynoszący Model parametryczny, wykorzystując 0.656.

Tutaj również zastosowanie zaawansowanych schematów oceny dokładności miało znaczący wpływ na nasz wniosek.

3.4 Wnioski końcowe

Tablica 5: Tabelka pozwalająca porównać wszystkie, wcześniej wybrane, najmniejsze błędy klasyfikacji i sposoby ich wyznaczenia dla każdego z testowanych algorytów.

Zestaw	Błąd_zbiór_testowy
Metoda k-najbliższych sąsiadów - Schemat bootstrap 632plus	0.313
Drzewo klasyfikacji - Optymalne parametry cp, minsplit i maxdepth	0.234
Naiwny klasyfikator bayesowski - Jednakowe prawdopodobieństwo klas	0.344

Porównując błędy z najlepszej opcji w każdej metodzie najmniejszy błąd klasyfikacji otrzymujemy, wykorzystując Drzewo klasyfikacji - Optymalne parametry cp, minsplit i maxdepth. Ten błąd wynosi 0.234. Jest to zatem najlepsza metoda dla danych `Glass`.

Tablica 6: Tabelka pozwalająca porównać wszystkie, wcześniej wybrane, największe błędy klasyfikacji i sposoby ich wyznaczenia dla każdego z testowanych algorytów.

Zestaw	Błąd_zbiór_testowy
Metoda k-najbliższych sąsiadów - Zbiór cech o najgorszej zdolności dyksryminacyjnej	0.594
Drzewo klasyfikacji - Zbiór cech o najlepszej zdolności dyskryminacyjnej	0.547
Naiwny klasyfikator bayesowski - Model parametryczny	0.656

Porównując błędy z najlepszej opcji w każdej metodzie największy błąd klasyfikacji otrzymujemy, wykorzystując Naiwny klasyfikator bayesowski - Model parametryczny. Ten błąd wynosi 0.656. Jest to zatem najgorsza metoda dla danych `Glass`. Porównując z nim błąd klasyfikacji obliczony ręcznie - wyznaczony przypisując wszystkie zmienne do najczęściej występującej klasy, wynoszący (0.645) - widzimy, że błąd jest odrobinę lepszy, od najgorszego wyznaczonego błędu.

4 Zaawansowane metody klasyfikacji

4.1 Metoda wektorów nośnych (SVM)

Metoda SVM szuka granicy (hiperpłaszczyzny dyskryminującej), która najlepiej oddziela klasy, czyli maksymalizuje odległość (margines) między punktami z różnych klas. Algorytm opiera się na odległości, dlatego zastosujemy tu zestandaryzowany, wygenerowany wcześniej, zbiór uczący i testowy.

4.1.1 Analiza

Tworzymy funkcję `svm.glass` dla argumentów:

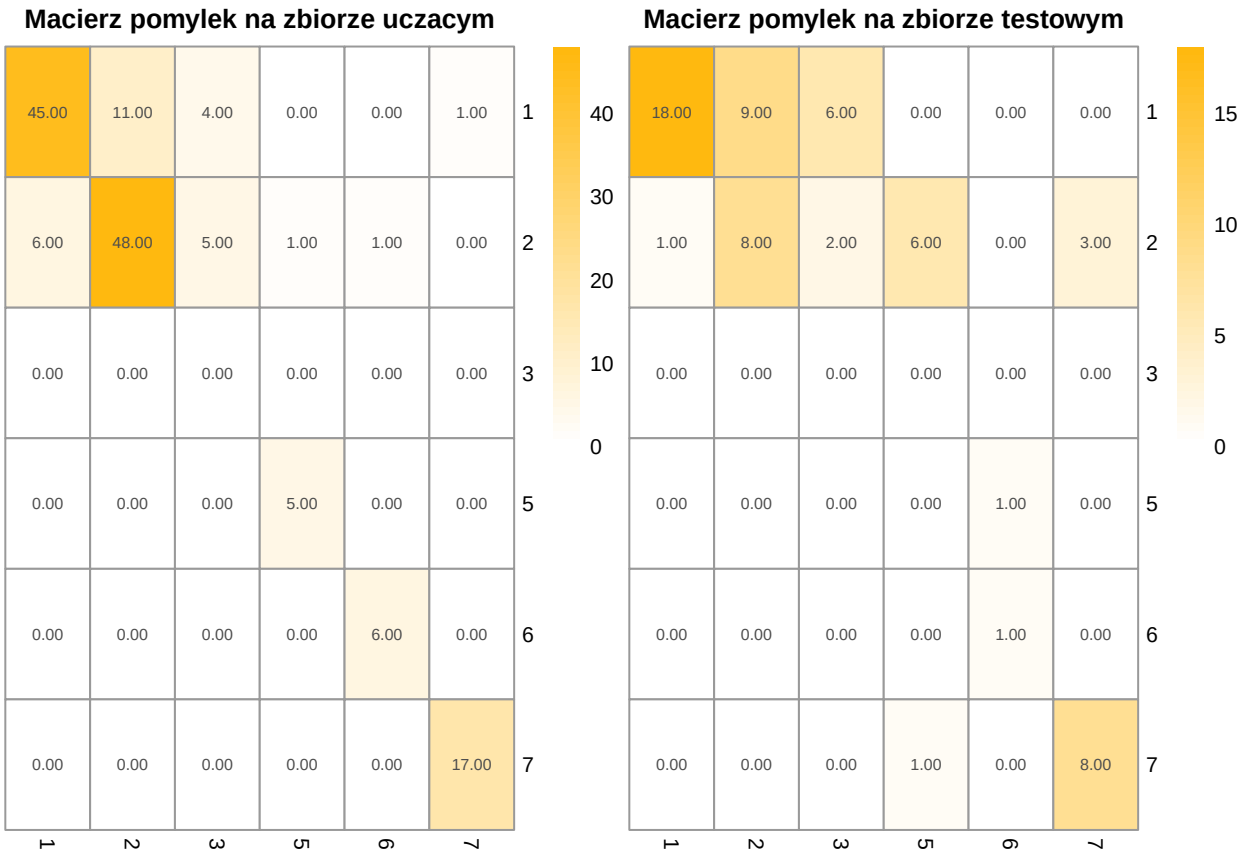
- zbiór uczący,
- zbiór testowy,
- formuła (domyślnie wszystkie cechy),
- parametr `kernel` określający, jakie jądro stosujemy (domyślnie `radial` - jądro gaussowskie),
- parametr `C` (domyślnie 1) - `cost` - tolerancja na błędy,
- parametr `g` - `gamma` - parametr zależny od wyboru jądra, wpływa na złożoność modelu,
- `model` (domyślnie `FALSE`) - określa, czy funkcja zwraca stworzony model SVM.

Na podstawie tych argumentów buduje model za pomocą funkcji `svm` (z biblioteki `e1071`), a następnie zwraca macierze pomyłek oraz błędy klasyfikacji dla zadanych zbiorów lub (w zależności od argumentu `model`) stworzony model SVM.

```r
svm.glass <- function(uczący, testowy, formula = Type ~ .,
                      kernel = "radial", C = 1,
                      g = if (is.vector(uczący)) 1 else 1 / ncol(uczący),
                      model = FALSE) {

  model.svm <- svm(formula, uczący, kernel = kernel, cost = C, gamma = g)

  # etykietki rzeczywiste
  etykietki.rzecz.t <- testowy$Type
  etykietki.rzecz.u <- uczący$Type

  # prognozy dla zbioru uczącego
  etykietki.prog.u <- predict(model.svm, newdata = uczący)

  # prognozy dla zbioru testowego
  etykietki.prog.t <- predict(model.svm, newdata = testowy)

  # macierz pomyłek (confusion matrix)
  pomyłki.u <- table(etykietki.prog.u, etykietki.rzecz.u)
  pomyłki.t <- table(etykietki.prog.t, etykietki.rzecz.t)


  # błąd klasyfikacji (na zbiorze uczącym i testowy)
  n.u <- nrow(uczący)
  error.u <- (n.u - sum(diag(pomyłki.u))) / n.u

  n.t <- nrow(testowy)
  error.t <- (n.t - sum(diag(pomyłki.t))) / n.t

  if (model == TRUE) {
    return(model.svm)
  } else {
    return((list(pomyłki.u, error.u, pomyłki.t, error.t)))
  }
}
```

Teraz, korzystając z funkcji `svm.glass`, wyznaczamy błędy klasyfikacji oraz macierze pomyłek dla zestandaryzowanego zbioru uczącego i testowego dla domyślnych parametrów.

4.1.2 Jądro gaussowskie



Rysunek 31: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla domyślnych parametrów funkcji svm.glass.

Błąd klasyfikacji na zbiorze uczącym: 0.193.

Błąd klasyfikacji na zbiorze testowym: 0.453.

4.1.3 Dokładność klasyfikacji a wybór jądra

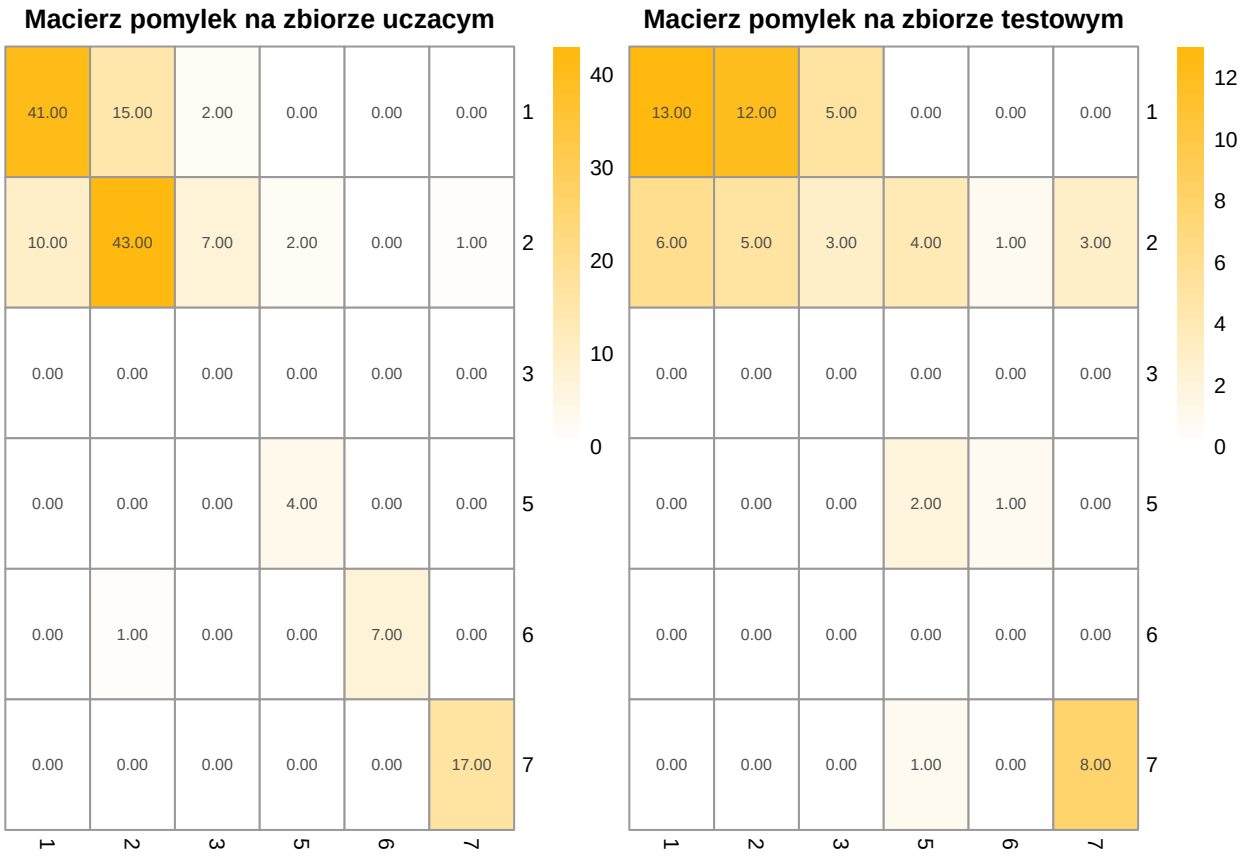
Metoda SVM daje nam możliwość wyznaczania błędu klasyfikacji w oparciu o różne jądra, które trenują nasz model. Mamy do dyspozycji cztery:

- jądro gaussowskie (domyślne),
- jądro liniowe (`linear`),
- jądro wielomianowe (`polynomial`),
- jądro sigmoidalne.

Korzystając z `svm.glass` wyznaczymy macierze pomyłek i błędy klasyfikacji dla każdego z nich, a następnie wybierzemy najlepsze jądra do optymalizacji.

Wyniki dla jądra gaussowskiego dla domyślnych parametrów były już sprawdzane wyżej, więc je tutaj pomijamy.

4.1.4 Jądro liniowe

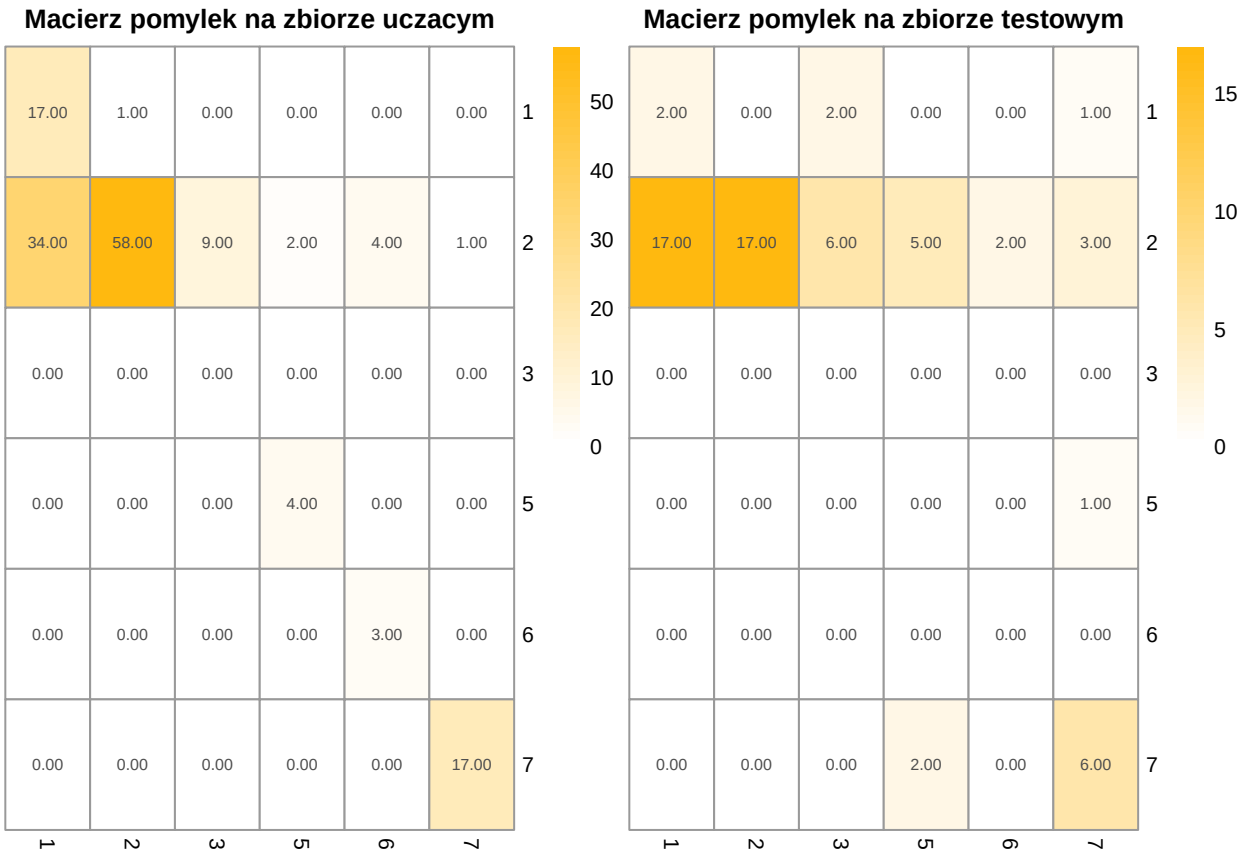


Rysunek 32: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla domyślnych parametrów funkcji svm.glass dla jądra liniowego.

Błąd klasyfikacji na zbiorze uczącym: 0.253.

Błąd klasyfikacji na zbiorze testowym: 0.562.

4.1.5 Jądro wielomianowe

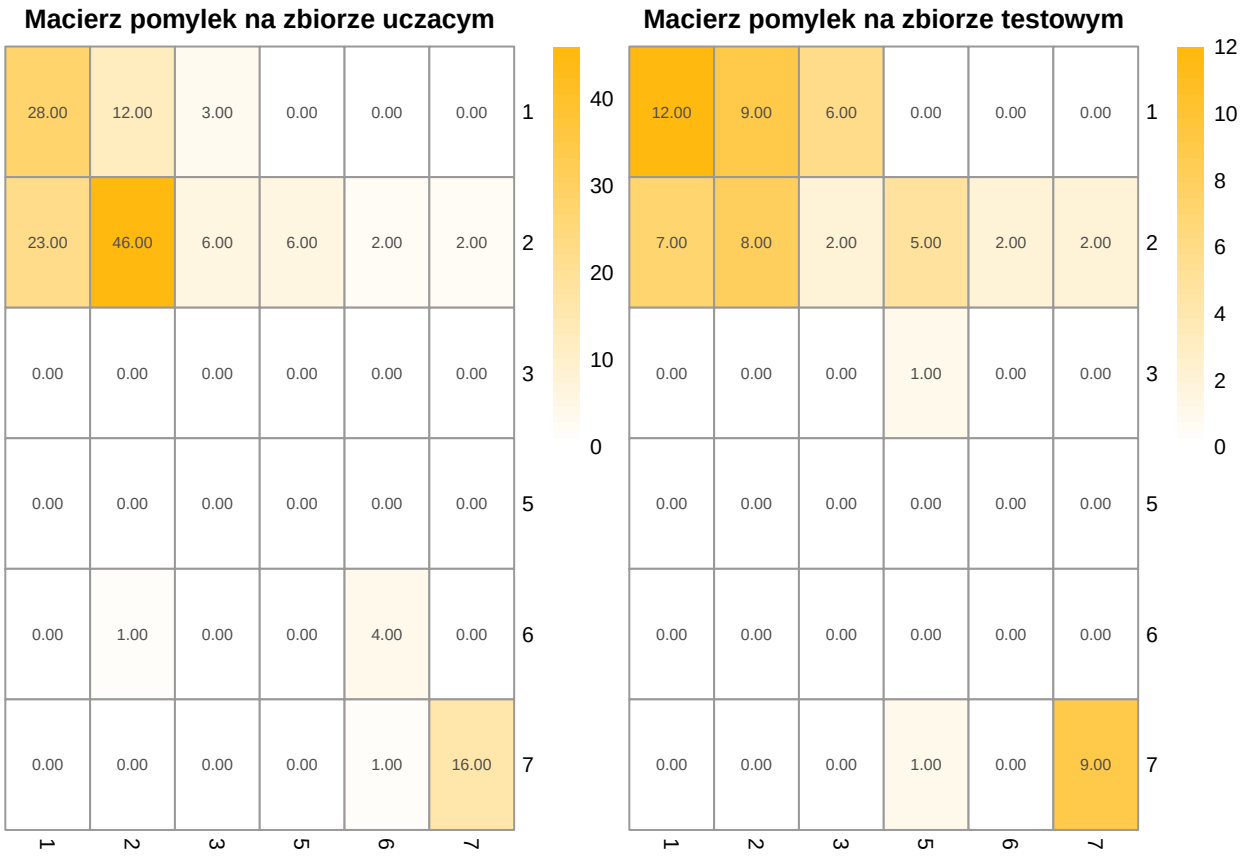


Rysunek 33: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla domyślnych parametrów funkcji svm.glass dla jądra wielomianowego.

Błąd klasyfikacji na zbiorze uczącym: 0.34.

Błąd klasyfikacji na zbiorze testowym: 0.609.

4.1.6 Jądro sigmoidalne



Rysunek 34: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla domyślnych parametrów funkcji svm.glass dla jądra sigmoidalnego.

Błąd klasyfikacji na zbiorze uczącym: 0.373.

Błąd klasyfikacji na zbiorze testowym: 0.547.

4.1.7 Które jądra optymalizować?

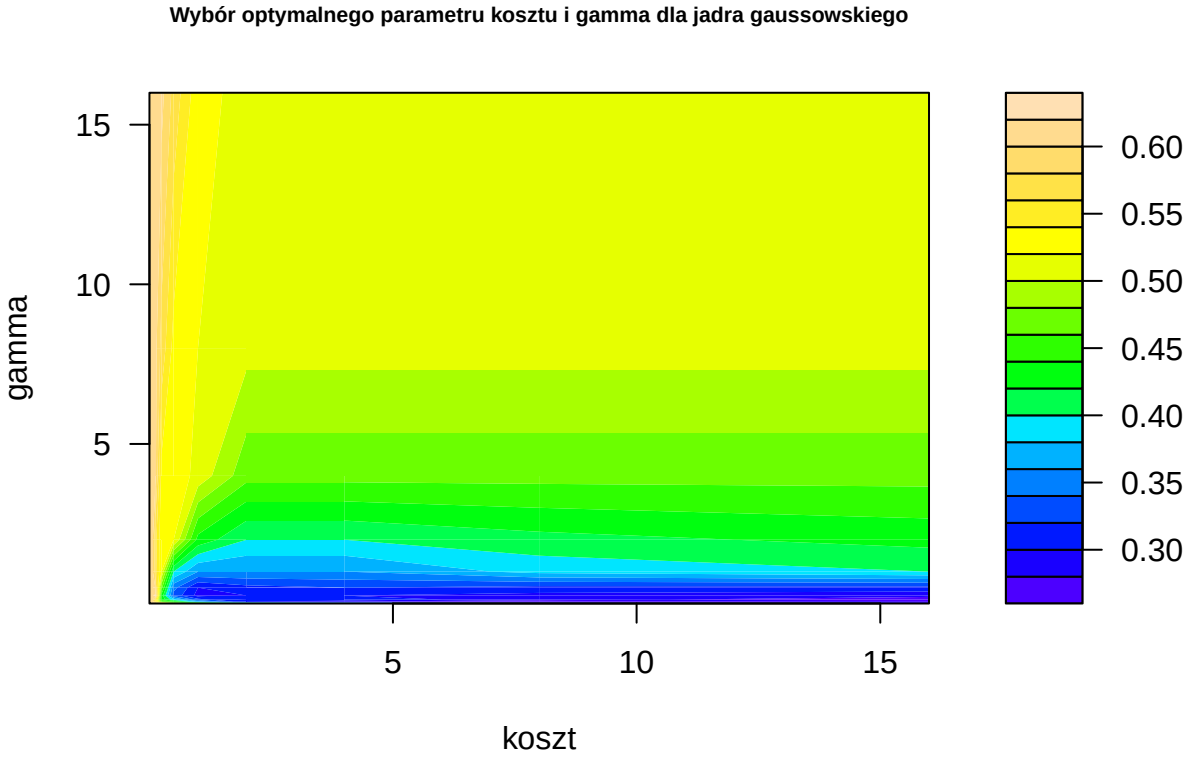
Tablica 7: Tabelka pozwalająca porównać wszystkie błędy klasyfikacji dla metody SVM dla różnych jąder z domyślnymi parametrami.

Jądro	Błąd_zbiór_uczący	Błąd_zbiór_testowy
Gaussowskie	0.193	0.453
Liniowe	0.253	0.562
Wielomianowe	0.340	0.609
Sigmoidalne	0.373	0.547

Na podstawie tabelki 7 w dalejszej części zajmiemy się optymalizacją parametrów jedynie dla jądra gaussowskiego i liniowego. Zdecydowałyśmy się na te jądra, ponieważ pierwsze z nich miało najmniejsze błędy na zbiorze uczącym i testowym (0.193 i 0.453). Natomiast błąd jądra liniowego na zbiorze testowym był trochę większy (0.562), niż w przypadku sigmoidalnego (0.547), ale sigmoidalne miało większy błąd na zbiorze uczącym ($0.373 > 0.253$), co może sugerować, że model nie jest w pełni stabilny.

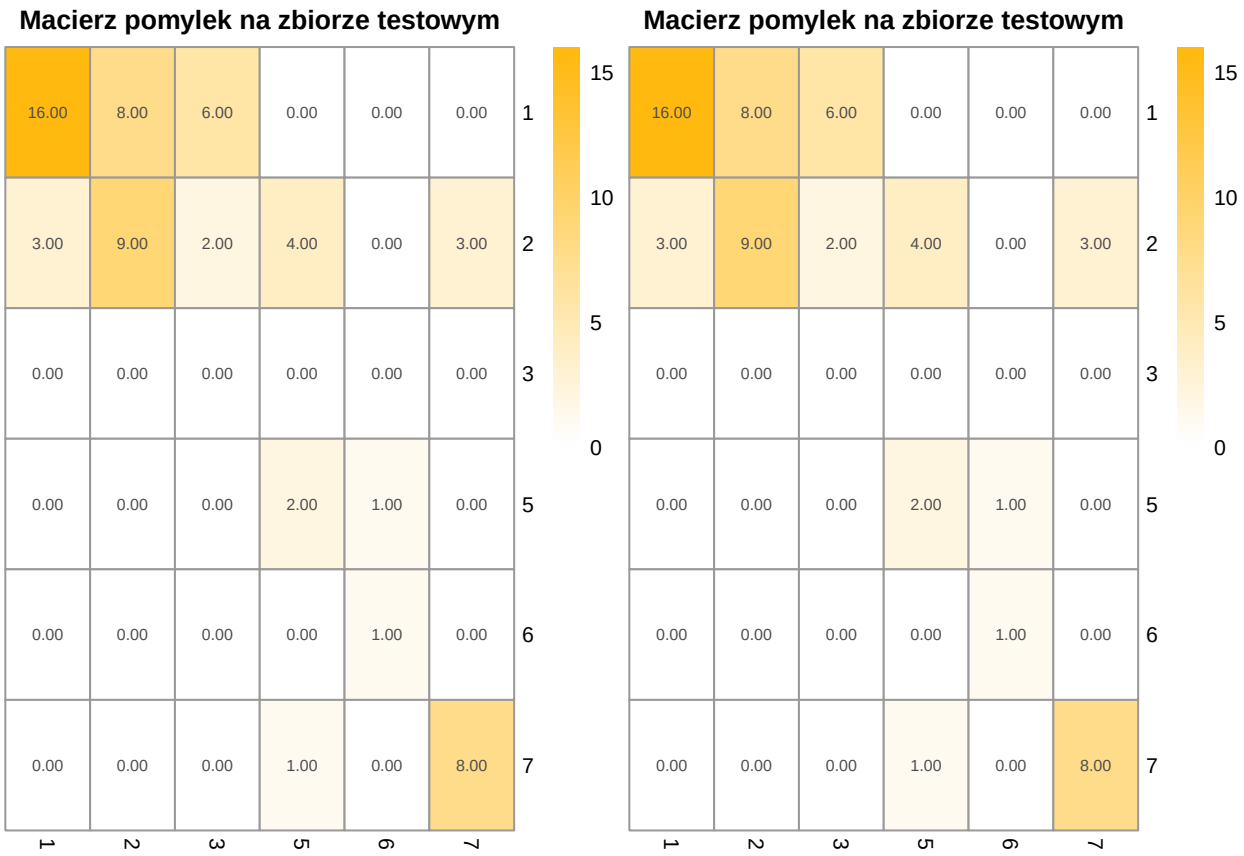
4.1.8 Optymalizacja parametrów dla jądra gaussowskiego

W przypadku jądra gaussowskiego możemy optymalizować parametr kosztu `C` i parametr `gamma`. Wybieramy najmniejsze możliwe wartości dla minimalnego błędu. Skorzystamy tu z funckji `tune` (z pakietu `e1071`) pozwalającej na znalezienie najoptymalniejszych parametrów dla zadanej metody.



Rysunek 35: Wykres pozwalający wybrać najoptymalniejszą wartość parametru kosztu oraz gamma dla jądra gaussowskiego dla wygenerowanego wcześniej zestandaryzowanego zbioru uczącego i testowego.

Jak widać na wykresie 35 najmniejszy błąd otrzymujemy dla `C` równego 8 oraz `gamma` równego 0.0625. Wyznaczamy teraz macierze pomyłek i błędy klasyfikacji dla tych wartości parametrów.



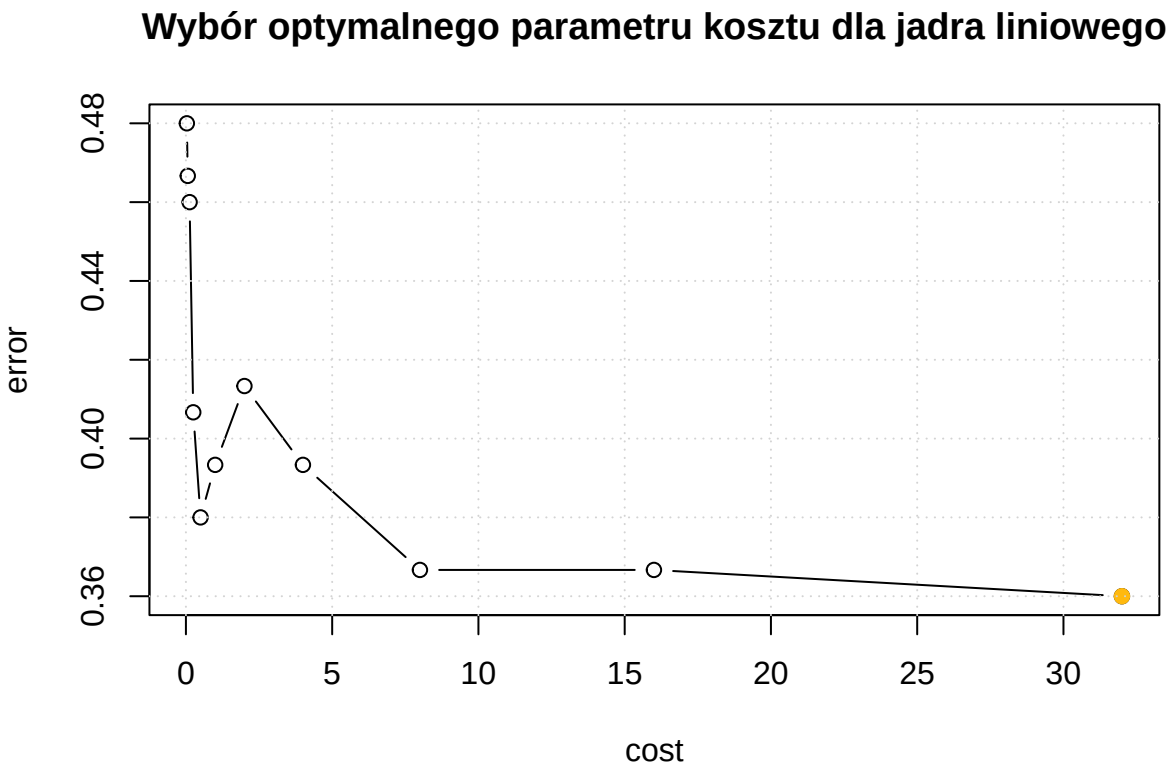
Rysunek 36: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla najoptymalniejszego C dla jądra gaussowskiego.

Błąd klasyfikacji na zbiorze uczącym: 0.173.

Błąd klasyfikacji na zbiorze testowym: 0.438.

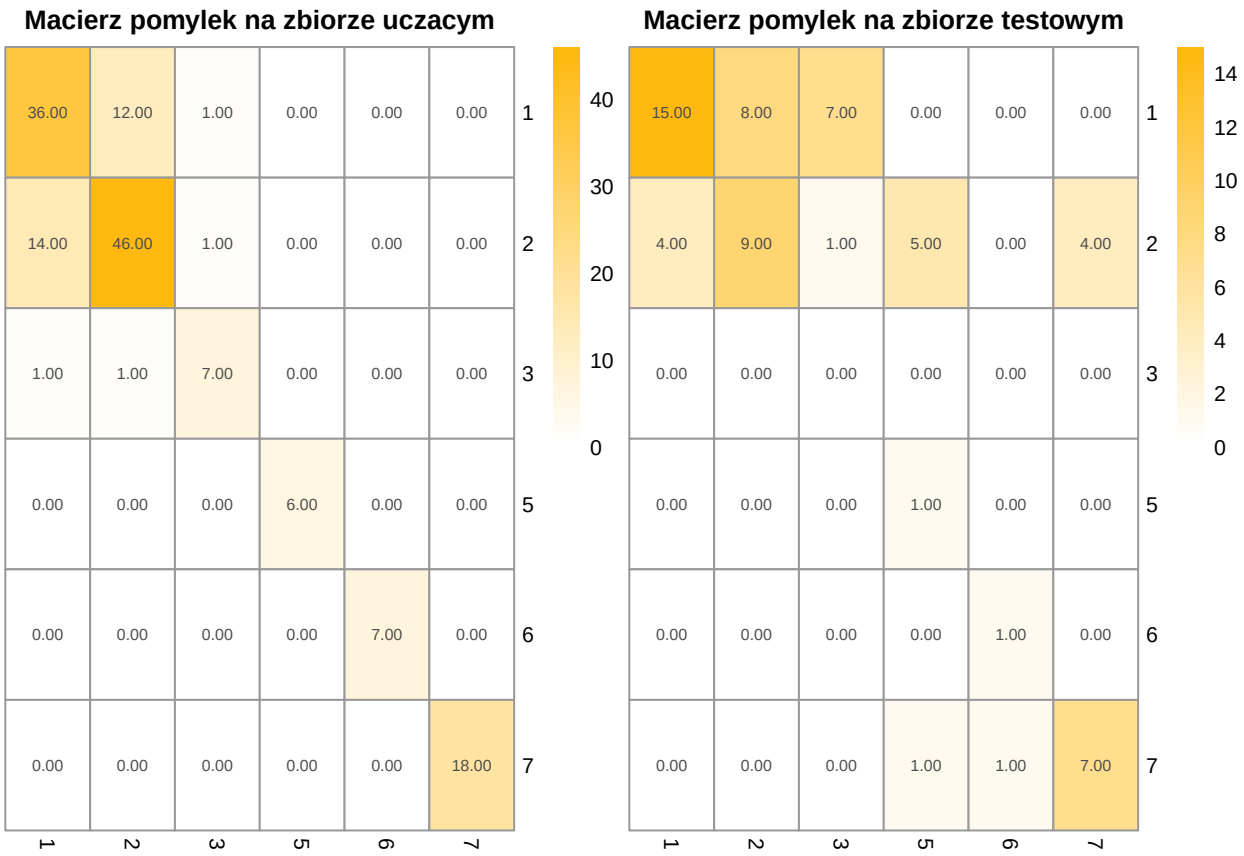
4.1.9 Optymalizacja parametrów dla jądra liniowego

W przypadku jądra liniowego jedynym parametrem, który będziemy optymalizować, jest parametr kosztu `C`.



Rysunek 37: Wykres pozwalający wybrać najoptymalniejszą wartość parametru kosztu dla jądra liniowego dla wygenerowanerowanego wcześniej zestandaryzowanego zbioru uczącego i testowego. Najoptymalniejsze C zostało zaznaczone kolorem złotożółtopomarańczowym.

Jak widać na wykresie 37 dla tego jądra najoptymalniejsza wartość parametru `C` wynosi 32. Stosujemy dla niej naszą funkcję.



Rysunek 38: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla najoptymalniejszego C dla jądra liniowego.

Błąd klasyfikacji na zbiorze uczącym: 0.2.

Błąd klasyfikacji na zbiorze testowym: 0.484.

4.1.10 Schemat *bootstrap 632plus*

To odmiana metody *bootstrap*, czyli metody opratej na wielokrotnym losowaniu prób (ze zwracaniem) ze zbioru danych `Glass.s` i tworzeniu zbioru uczącego i testowego. Ta odmiana pozwala pozwala na uniknięcie przeuczenia modelu. Wykonamy to dla dwóch optymalizowanych jąder.

Błąd klasyfikacji metodą *bootstrap 632plus* dla danych `Glass.s` dla jądra gaussowskiego wynosi 0.286.

Błąd klasyfikacji metodą *bootstrap 632plus* dla danych `Glass.s` dla jądra liniowego wynosi 0.335.

4.2 Wnioski cząstkowe SVM

Tablica 8: Tabelka pozwalająca porównać błędy klasyfikacji dla metody SVM.

Metoda	Błąd_zbiór_uczący	Błąd_zbiór_testowy
Model podstawowy - domyślne parametry	0.193	0.453
Najoptymalniejsze jądro gaussowskie	0.173	0.438
Najoptymalniejsze jądro liniowe	0.2	0.484
Jądro gaussowskie - schemat bootstrap 632plus	-	0.286
Jądro liniowe - schemat bootstrap 632plus	-	0.335

Porównując wszystkie wyznaczone w analizie błędy klasyfikacji (tabela 8), widzimy, że dla metody wektorów nośnych (SVM) najlepszy wynik otrzymujemy, wykorzystując Jądro gaussowskie - schemat bootstrap 632plus i wynosi on 0.286. Natomiast największy, wynoszący 0.484, wykorzystując Najoptymalniejsze jądro liniowe.

4.3 Rodziny klasyfikatorów

Tak jak wspominałyśmy, w naszej analizie skupimy na trzech algorytmach rodziny klasyfikatorów. Dla każdego z nich stworzymy funkcję zwracającą macierze pomyłek i błędy klasyfikacji oraz poszukamy najoptymalniejszych parametrów. W każdym przypadku klasyfikatorem bazowym będzie drzewo klasyfikacji, dlatego nie musimy korzystać z zestandarowyzowanej wersji zbioru uczącego i testowego.

4.4 Metoda *bagging*

Ten algorytm opiera się na generowaniu B-boostarpowych replikacji zbioru uczącego poprzez losowanie ze zwracaniem z oryginalnego zbioru uczącego. Dla każdej takiej replikacji konstruujemy klasyfikator na podstawie klasyfikatora bazowego i na koniec wyznaczamy klasyfikator zagregowany. Ostateczna decyzja podejmowana w oparciu o regułę głosowania większości (klasa równa tej, którą ma najwięcej elementów z danej grupy).

4.4.1 Analiza

Tworzymy funkcję `bag.glass` dla argumentów:

- zbiór uczący,
- zbiór testowy,
- formuła (domyślnie wszystkie cechy),
- parametry tworzące drzewo (`cp`, `minsplit` i `maxdepth` ustawione na wartości domyślne)
- `nbagg` - liczba podziałów (drzew).

Na podstawie tych argumentów buduje model oparty o algorytm *bagging* za pomocą funkcji `bagging` (z biblioteki `ipred`), a następnie zwraca macierze pomyłek oraz błędy klasyfikacji dla zadanych zbiorów.

```r
bag.glass <- function(uczący, testowy, formula = Type ~ .,
                      nbagg = 25, cp = 0.01, minsplit = 20, maxdepth = 30) {

  Bag.model <- bagging(formula = formula, data = uczący, nbagg = nbagg,
                       control = rpart.control(cp = cp, minsplit = minsplit,
                                               maxdepth = maxdepth))

  # etykietki rzeczywiste
  etykietki.rzecz.t <- testowy$Type
  etykietki.rzecz.u <- uczący$Type

  # prognozy dla zbioru uczącego
  etykietki.prog.u <- predict(Bag.model, newdata = uczący, type = "class")$class

  # prognozy dla zbioru testowego
  etykietki.prog.t <- predict(Bag.model, newdata = testowy, type = "class")$class

  # macierz pomyłek (confusion matrix)
  pomyłki.u <- table(etykietki.prog.u, etykietki.rzecz.u)
  pomyłki.t <- table(etykietki.prog.t, etykietki.rzecz.t)

  # błąd klasyfikacji (na zbiorze uczącym i testowy)
  n.u <- nrow(uczący)
  error.u <- (n.u - sum(diag(pomyłki.u))) / n.u

  n.t <- nrow(testowy)
  error.t <- (n.t - sum(diag(pomyłki.t))) / n.t

  return((list(pomyłki.u, error.u, pomyłki.t, error.t)))
}
```

Testujemy teraz naszą funkcję na wygenerowanym wcześniej zbiorze uczącym i testowym.

Rysunek 39: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla domyślnych parametrów funkcji bag.glass.

Błąd klasyfikacji na zbiorze uczącym: 0.187.

Błąd klasyfikacji na zbiorze testowym: 0.297.

4.4.2 Optymalizacja parametrów

Podobnie, jak w przypadku drzewa klasyfikacji chcemy wyznaczyć najoptymalniejsze parametry do konstrukcji drzewa. Jednak *bagging* opiera się na budowaniu wielu drzew, dlatego nie da się tego przeprowadzić w ten sam sposób co ostatnio. Postanowiłyśmy zatem skorzystać z wyliczonych wcześniej najoptymalniejszych parametrów dla pojedynczego drzewa - parametry najoptymalniejszego drzewa klasyfikacji.

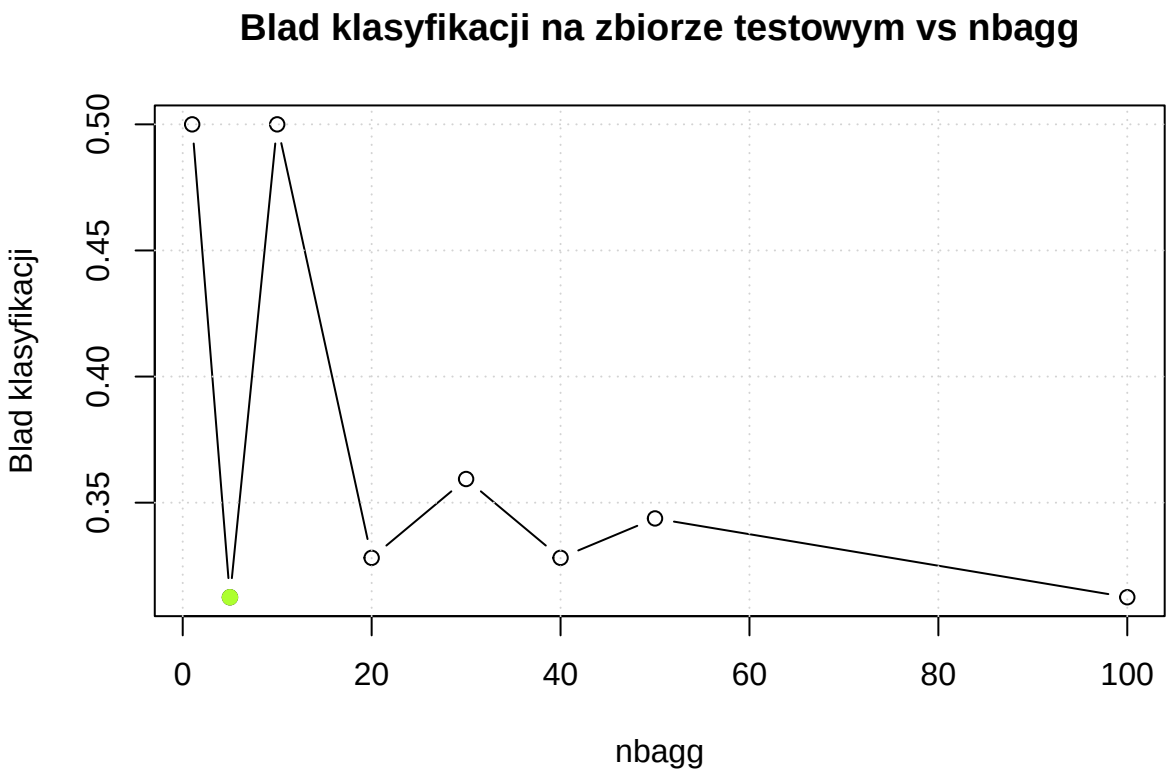
Sprawdzamy teraz wyniki dla tych wartości parametrów.

Rysunek 40: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla optymalnych parametrów cp, minsplit i maxdepth dla funkcji bag.glass.

Błąd klasyfikacji na zbiorze uczącym: 0.107.

Błąd klasyfikacji na zbiorze testowym: 0.297.

Teraz dla naszego optymalnego modelu szukamy najoptymalniejszej liczby podziałów `nbagg`.



Rysunek 41: Wykres przedstawiający, jak zmienia się błąd klasyfikacji dla algorytmu bagging z użyciem schematu bootstrap 632plus w zależności od liczby podziałów nbagg. Najoptymalniejszą wartość zaznaczono kolorem limonkowym.

Nagły wzrost błędu na wykresie 41 może wynikać z przybliżeń algorytmu. Najoptymalniejsza liczba podziałów dla wyznaczonowych wcześniej parametrów wynosi 5. Użyjemy teraz naszej funkcji do znalezienia błędów klasyfikacji i macierzy pomyłek.

Rysunek 42: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla optymalnego modelu i optymalnego nbagg bag.glass.

Błąd klasyfikacji na zbiorze uczącym: 0.1.

Błąd klasyfikacji na zbiorze testowym: 0.281.

4.4.3 Schemat *boostrap 632plus*

Dla tej metody wybierzemy jej indywidualną najoptymalniejszą liczbę podziałów `nbagg`.

Błąd klasyfikacji metodą *bootstrap 632plus* wynosi 0.238.

4.5 Wnioski cząstkowe *bagging*

Tablica 9: Tabelka pozwalająca porównać wcześniej wyliczone, błędy klasyfikacji dla algorytmu bagging.

Metoda	Błąd_zbiór_uczący	Błąd_zbiór_testowy
Model podstawowy - domyślne parametry	0.187	0.297
Optymalne parametry cp, minsplit i maxdepth	0.107	0.297
Optymalne parametry cp, minsplit, maxdepth i nbagg	0.1	0.281
Schemat bootstrap 632plus	-	0.238

Porównując wszystkie wyznaczone w analizie błędy klasyfikacji (tabela 9), widzimy, że dla metody *bagging* najlepszy wynik otrzymujemy, wykorzystując Schemat bootstrap 632plus i wynosi on 0.238. Natomiast największy, wynoszący 0.297, wykorzystując Model podstawowy - domyślne parametry.

4.6 Metoda *boosting*

Ten algorytm działa podobnie do metody *bagging*, różni ją jednak to, że kolejne klasyfikatory są wyznaczane sekwencyjnie - pierwszy klasyfikator uczy się na oryginalnym zbiorze danych, a każdy kolejny koncentruje się bardziej na tych przypadkach, które zostały źle sklasyfikowane przez poprzedni. Jest to możliwe dzięki wagom, które są przypisane do każdego przypadku ze zbioru uczącego. Po każdej iteracji wagi są zwiększane dla błędnie sklasyfikowanych przypadków i zmniejszane dla tych dobrych.

W naszej analizie wykorzystamy algorytm *AdaBoost*.

4.6.1 Analiza

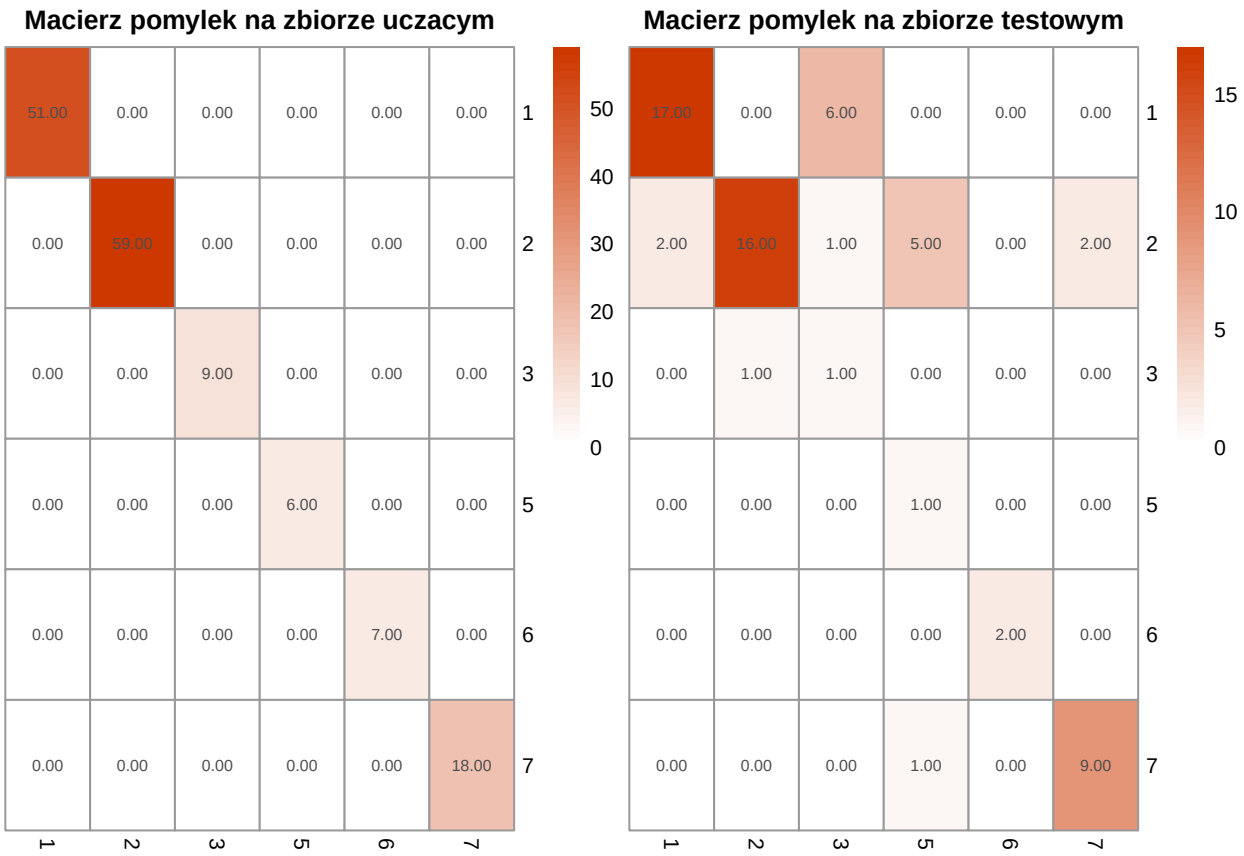
Tworzymy funkcję `boost.glass` dla argumentów:

- zbiór uczący,
- zbiór testowy,
- formuła (domyślnie wszystkie cechy),
- parametry tworzące drzewo (`cp`, `minsplit` i `maxdepth` ustawione na wartości domyślne)
- `mfinal` - liczba iteracji/stworzonych drzew.

Na podstawie tych argumentów buduje model oparty o algorytm *AdaBoost* za pomocą funkcji `boosting` (z biblioteki `adabag`), a następnie zwraca macierze pomyłek oraz błędy klasyfikacji dla zadanych zbiorów, prognozując w oparciu o funkcję `predict.boosting`.

```
boost.glass <- function(uczący, testowy, formula = Type ~ .,
                        cp = 0.01, minsplit = 20, maxdepth = 30, mfinal = 100) {

  Boost.model <- boosting(formula = formula, data = uczący, mfinal = mfinal,
                          control = rpart.control(cp = cp, minsplit = minsplit, maxdepth = maxdepth))

  # etykietki rzeczywiste
  etykietki.rzecz.t <- testowy$Type
  etykietki.rzecz.u <- uczący$Type

  # prognozy dla zbioru uczącego
  etykietki.prog.u <- predict.boosting(Boost.model, newdata = uczący)$class

  # prognozy dla zbioru testowego
  etykietki.prog.t <- predict(Boost.model, newdata = testowy)$class

  # macierz pomyłek (confusion matrix)
  pomyłki.u <- table(etykietki.prog.u, etykietki.rzecz.u)
  pomyłki.t <- table(etykietki.prog.t, etykietki.rzecz.t)

  # błąd klasyfikacji (na zbiorze uczącym i testowy)
  n.u <- nrow(uczący)
  error.u <- (n.u - sum(diag(pomyłki.u))) / n.u

  n.t <- nrow(testowy)
  error.t <- (n.t - sum(diag(pomyłki.t))) / n.t

  return((list(pomyłki.u, error.u, pomyłki.t, error.t)))
}
```

Testujemy teraz naszą funkcję na wygenerowanym wcześniej zbiorze uczącym i testowym.



Rysunek 43: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla domyślnych parametrów funkcji boost.glass.

Błąd klasyfikacji na zbiorze uczącym: 0.

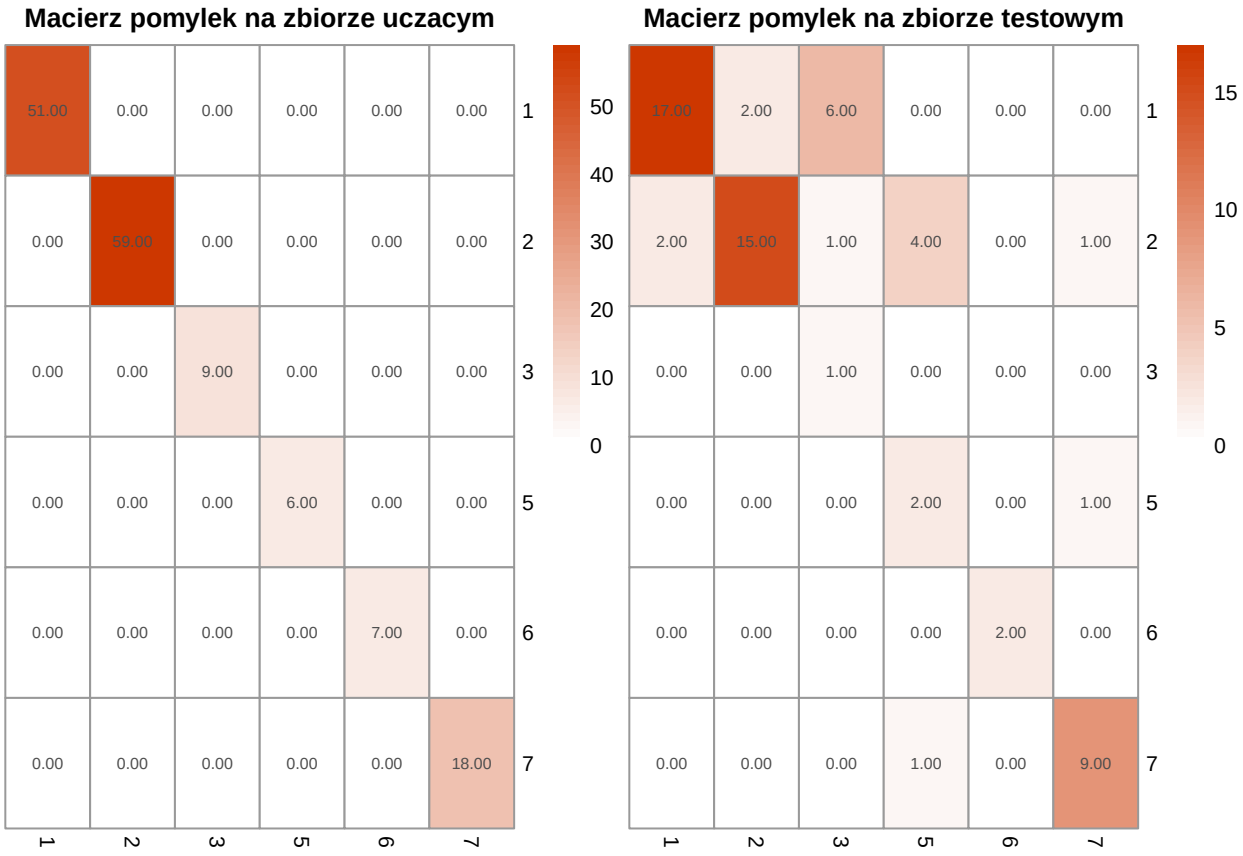
Musimy być ostrożne, ponieważ błąd klasyfikacji na zbiorze uczącym jest równy 0 - istnieje ryzyko przeuczenia modelu, co może prowadzić do zbyt optymistycznej oceny jego skuteczności.

Błąd klasyfikacji na zbiorze testowym: 0.281.

4.6.2 Optymalizacja parametrów

Podobnie, jak w przypadku metody *bagging* skorzystamy z najoptymalniejszych parametrów dla pojedynczego drzewa z tabelki **??**.

Sprawdzamy teraz wyniki dla tych wartości parametrów.



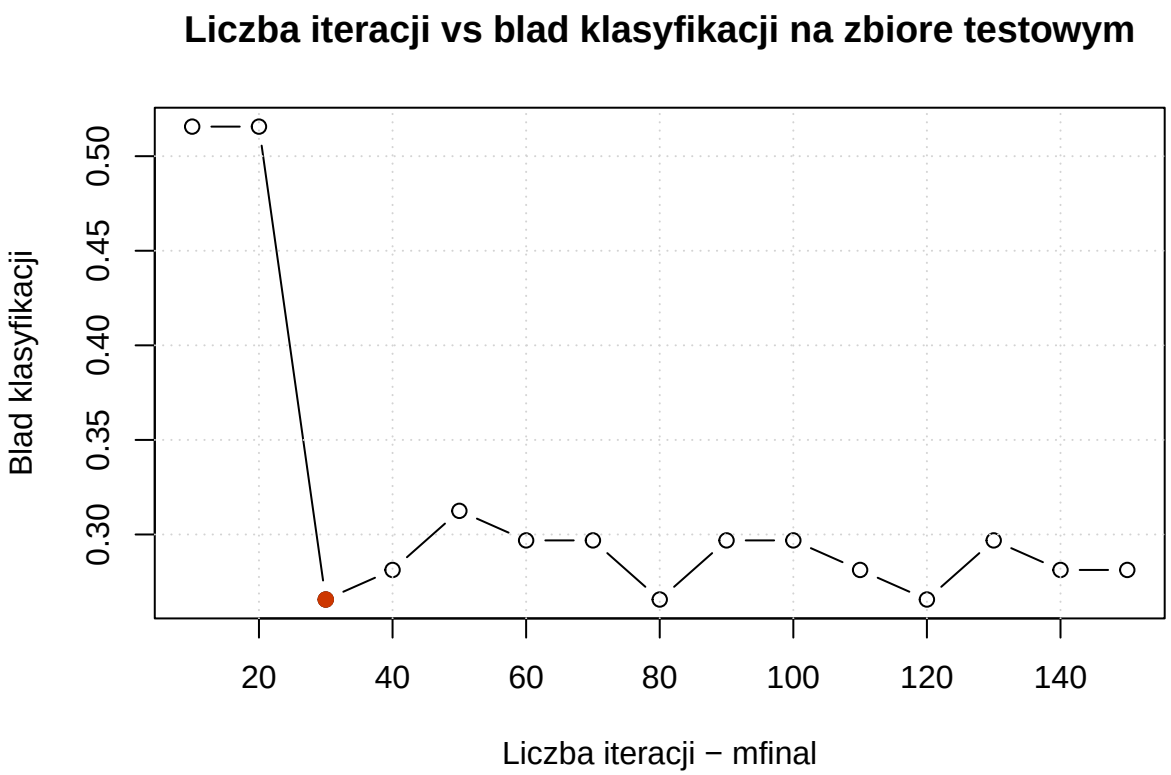
Rysunek 44: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla optymalnych parametrów cp, minsplit i maxdepth dla funkcji boost.glass.

Błąd klasyfikacji na zbiorze uczącym: 0.

Uwaga! Ryzyko przeuczenia.

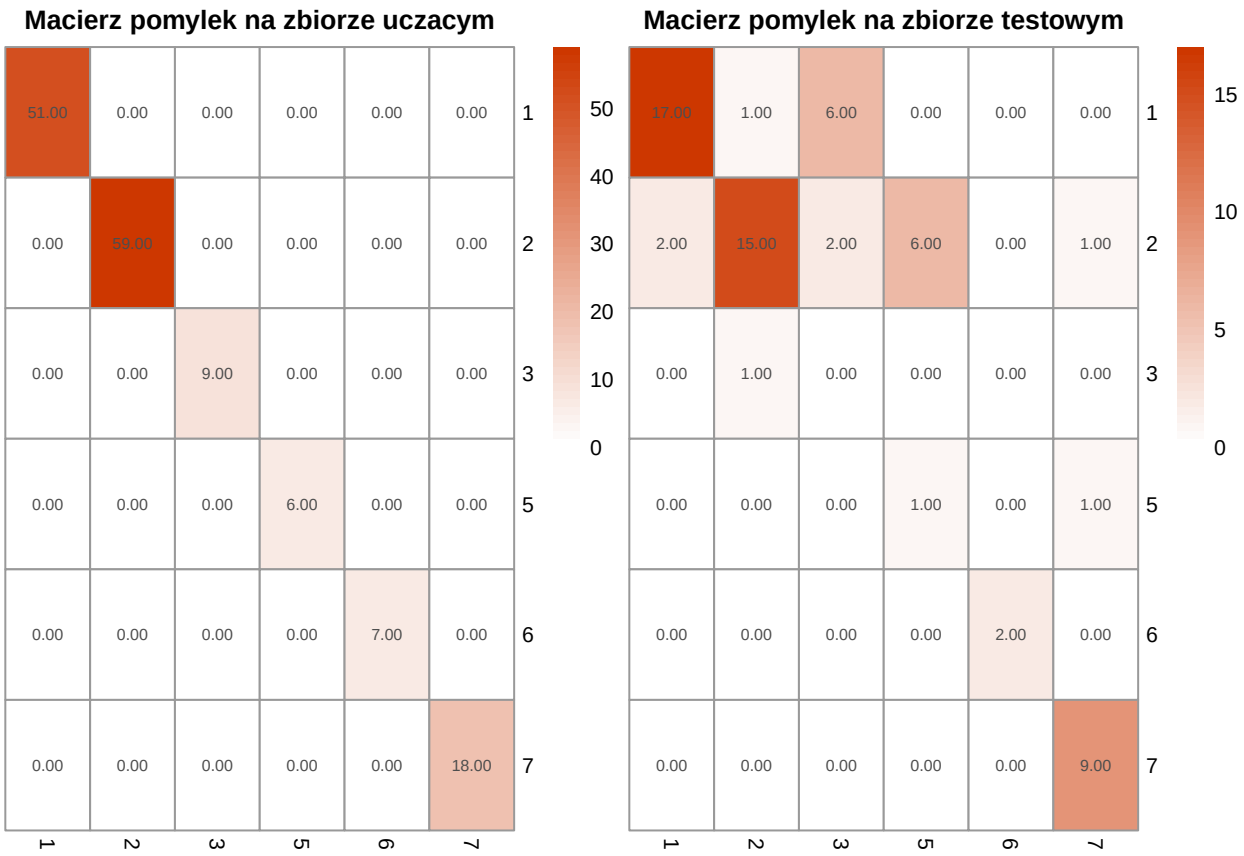
Błąd klasyfikacji na zbiorze testowym: 0.281.

Teraz dla naszego optymalnego modelu szukamy najoptymalniejszej liczby itrecji `mfinal`.



Rysunek 45: Wykres przedstawiający, jak zmienia się błąd klasyfikacji na zbiorze testowym dla naszego optymalnego modelu dla algorytmu boosting w zależności od liczby iteracji mfinal. Najoptymalniejszą wartość zaznaczono kolorem ceglastoczerwonym.

Najoptymalniejsza liczba iteracji dla wyznaczonowych wcześniej parametrów na naszym zbiorze testowym wynosi 30. Użyjemy teraz naszej funkcji do znalezienia błędów klasyfikacji i macierzy pomyłek.



Rysunek 46: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla optymalnych parametrów cp, minsplit i maxdepth oraz mfinal dla funkcji boost.glass.

Błąd klasyfikacji na zbiorze uczącym: 0.

Uwaga! Ryzyko przeuczenia.

Błąd klasyfikacji na zbiorze testowym: 0.312.

4.6.3 Schemat *boostrap 632plus*

Błąd klasyfikacji metodą *bootstrap 632plus* wynosi 0.179.

4.7 Wnioski cząstkowe *boosting*

Tablica 10: Tabelka pozwalająca porównać błędy klasyfikacji dla algorytmu boosting.

Metoda	Błąd_zbiór_uczący	Błąd_zbiór_testowy
Model podstawowy - domyślne parametry	0	0.281
Optymalne parametry cp, minsplit i maxdepth	0	0.281
Optymalne parametry cp, minsplit, maxdepth i mfinal	0	0.312
Schemat bootstrap 632plus	-	0.179

Porównując wszystkie wyznaczone w analizie błędy klasyfikacji (tabela 10), widzimy, że dla algorytmu *boosting* (*Adaboost*) najlepszy wynik otrzymujemy, wykorzystując Schemat bootstrap 632plus i wynosi on 0.179. Natomiast największy, wynoszący 0.312, wykorzystując Optymalne parametry cp, minsplit, maxdepth i mfinal.

4.8 Metoda *random forest*

Algorytm polega na tworzeniu zbioru wielu drzew, a następnie trenowaniu każdego drzewa na innej bootstrapowej próbce danych (wybieranej losowo). Ostateczna predykcja powstaje w opraciu o głosowanie większości.

4.8.1 Analiza

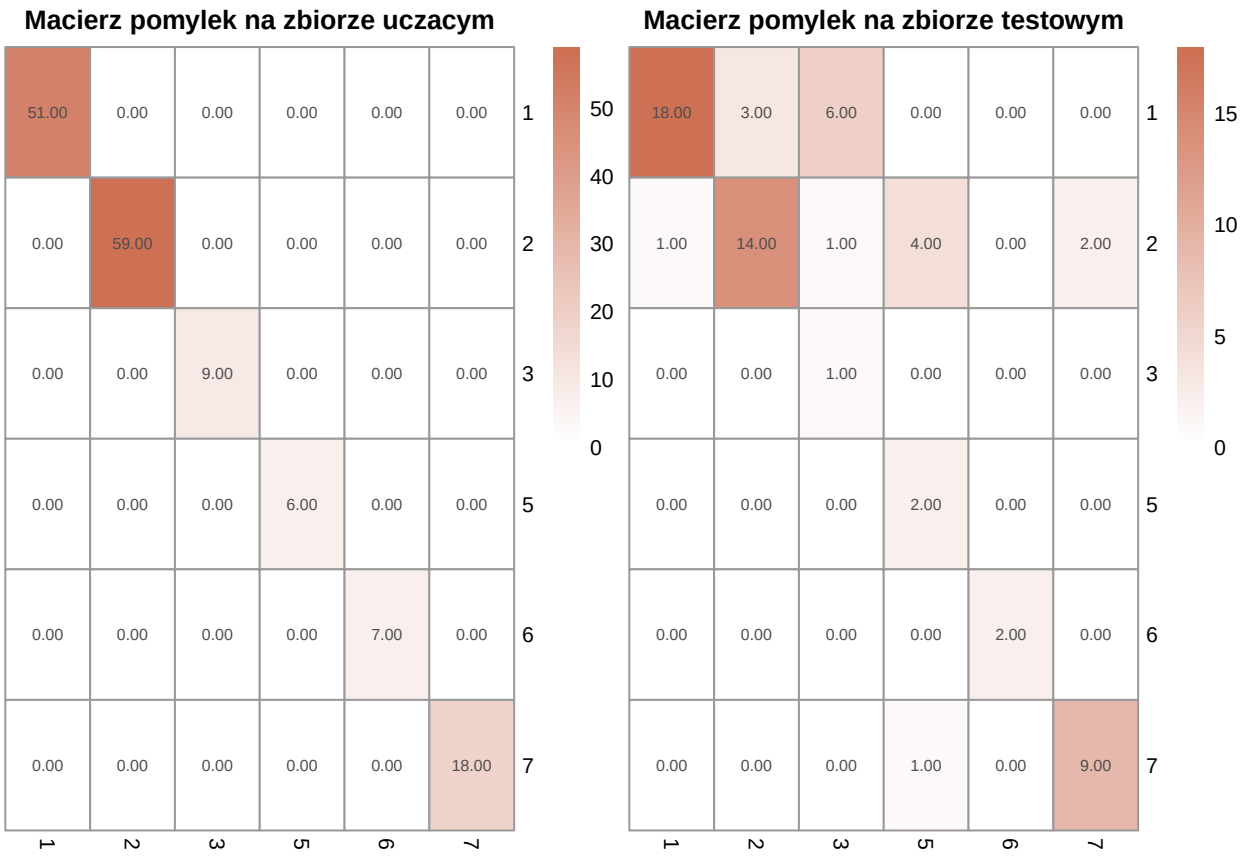
Tworzymy funkcję `forest.glass` dla argumentów:

- zbiór uczący,
- zbiór testowy (domyślnie `NULL`),
- formuła (domyślnie wszystkie cechy),
- `ntree` - liczba drzew,
- `mtry` - liczba wybieranych losowo cech,
- `ranking` (domyślnie `FALSE`) - określa, czy funkcja zwraca wykres przedstawiający ranking ważności cech,
- `OOB` (domyślnie `FALSE`) - określa, czy funkcja zwraca macierz pomyłek i błąd klasyfikacji na bazie OOB (Out-Of-Bag) oraz wykres błędu klasyfikacji OBB (ogólny oraz dla każdej klasy osobno) w zależności od liczby drzew.

Na podstawie tych argumentów buduje model oparty o algorytm *random forest* za pomocą funkcji `randomForest` (z biblioteki `randomForest`), a następnie zwraca macierze pomyłek oraz błędy klasyfikacji dla zadanych zbiorów lub (w zależności od argumentu `ranking`) wykres przedstawiający ranking ważności cech lub (w zależności od argumentu `OOB`) macierz pomyłek oraz wykres błędu klasyfikacji na bazie OOB.

```r
forest.glass <- function(uczący, testowy, formula = Type ~ .,
                         n = 500, m = floor(sqrt(ncol(uczący) - 1)),
                         model = FALSE, ranking = FALSE, OOB = FALSE){

  Glass.forest <- randomForest(formula, data = uczący, ntree = n, mtry = m, importance = TRUE)

  # etykietki rzeczywiste
  etykietki.rzecz.u <- uczący$Type

  # prognozy dla zbioru uczącego
  etykietki.prog.u <- predict(Glass.forest, newdata = uczący, type = "class")

  # macierz pomyłek (confusion matrix)
  pomyłki.u <- table(etykietki.prog.u, etykietki.rzecz.u)

  # błąd klasyfikacji (na zbiorze uczącym)
  n.u <- nrow(uczący)
  error.u <- (n.u - sum(diag(pomyłki.u))) / n.u

  etykietki.rzecz.t <- testowy$Type

  # prognozy dla zbioru testowego
  etykietki.prog.t <- predict(Glass.forest, newdata = testowy, type = "class")
  pomyłki.t <- table(etykietki.prog.t, etykietki.rzecz.t)
  n.t <- nrow(testowy)
  error.t <- (n.t - sum(diag(pomyłki.t))) / n.t

  if (ranking == TRUE) {
    varImpPlot(Glass.forest, main = "Ranking ważności cech")
  } else if (OOB == TRUE) {
    plot(Glass.forest, main = "Błąd klasyfikacji (OOB)")
    oob <- Glass.forest$confusion
    oob.error <- Glass.forest$err.rate[n, "OOB"]
    return(list(oob, oob.error))
  } else if (model == TRUE) {
    return(Glass.forest)
  } else {
    return((list(pomyłki.u, error.u, pomyłki.t, error.t)))
  }
}
```

Testujemy teraz naszą funkcję na wygenerowanym wcześniej zbiorze uczącym i testowym.



Rysunek 47: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla domyślnych parametrów funkcji forest.glass.

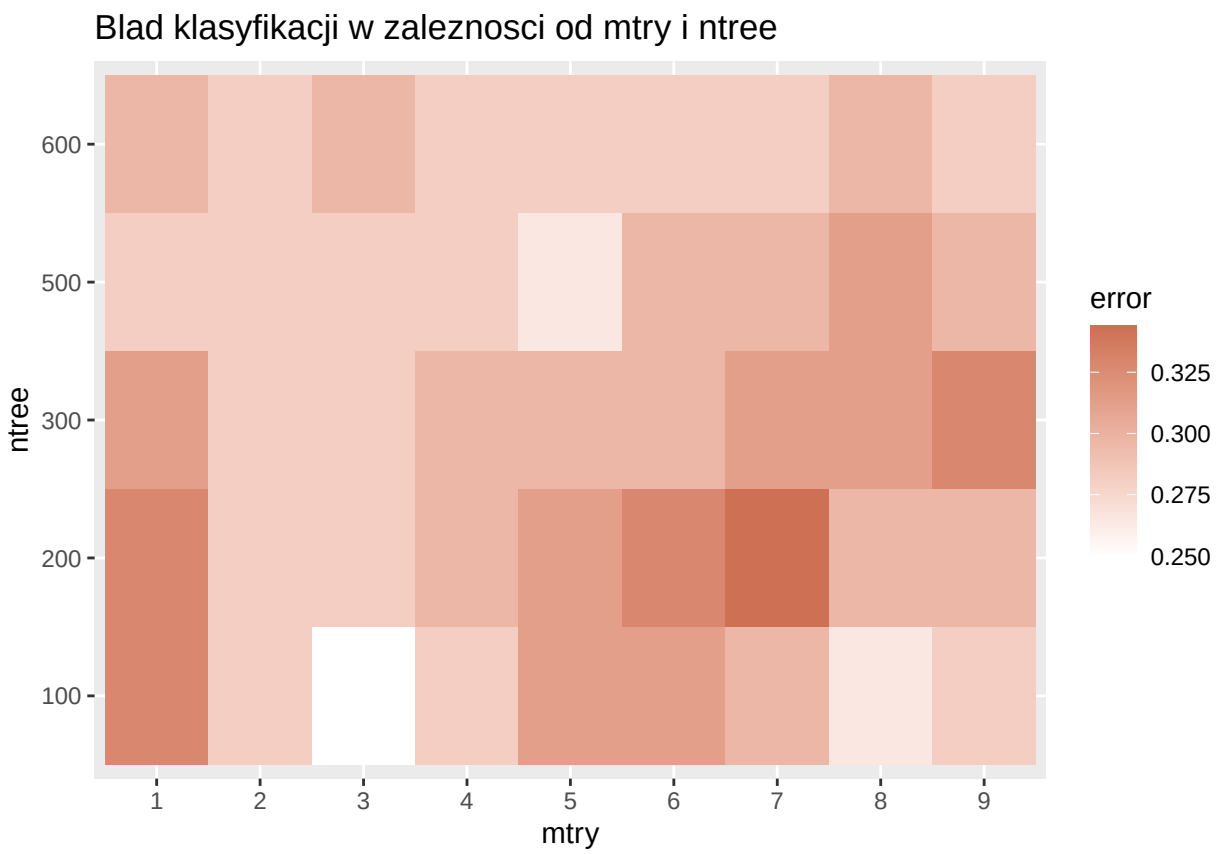
Błąd klasyfikacji na zbiorze uczącym: 0.

Uwaga! Ryzyko przeuczenia.

Błąd klasyfikacji na zbiorze testowym: 0.281.

4.8.2 Optymalizacja parametrów

W przypadku metody random forest możemy optymalizować parametry `mtry` oraz `ntree`.



Rysunek 48: Wykres przedstawiający zmianę wartości błędu klasyfikacji, przy zmianie parametrów mtry oraz ntree.

Jak widać na wykresie 48, dla naszego zbioru uczącego i testowego najmniejszy błąd otrzymujemy dla `mtry` równego 3 oraz `ntree` równego 100.

Sprawdzamy teraz wyniki dla tych wartości parametrów.

Rysunek 49: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla optymalnych parametrów mtry oraz ntree dla funkcji forest.glass.

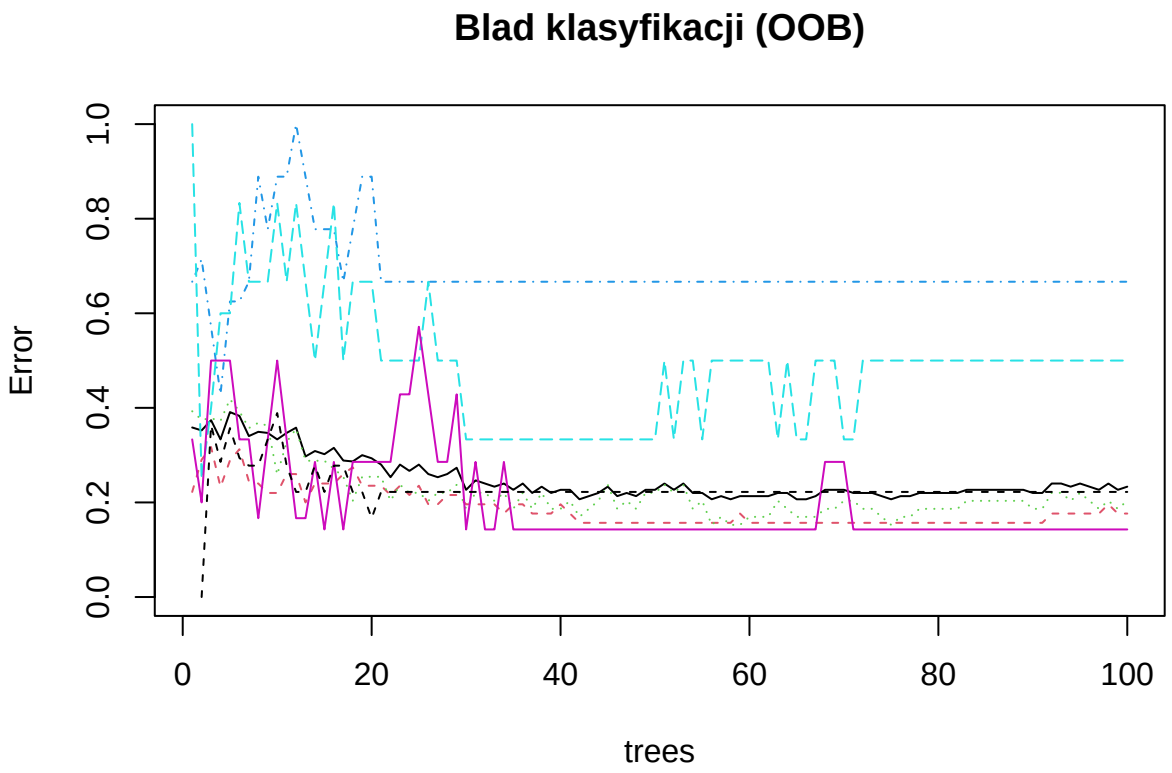
Błąd klasyfikacji na zbiorze uczącym: 0.

Błąd klasyfikacji na zbiorze testowym: 0.25.

4.8.3 Obserwacje Out-of-Bag (OOB)

Obserwacje OOB to obserwacje, które nie były wybrane do zbioru uczącego w danej replikacji podczas tworzenia drzewa w algorytmie. Pełnią rolę wewnętrzego zbioru testowego, co pozwala na wyznaczenie błędu klasyfikacji bez tworzenia zewnętrzego zbioru.

Zatem dla naszego modelu z optymalnymi parametrami wyznaczymy macierz pomyłek oraz wykres błądu klasyfikacji w zależności od liczby drzew.



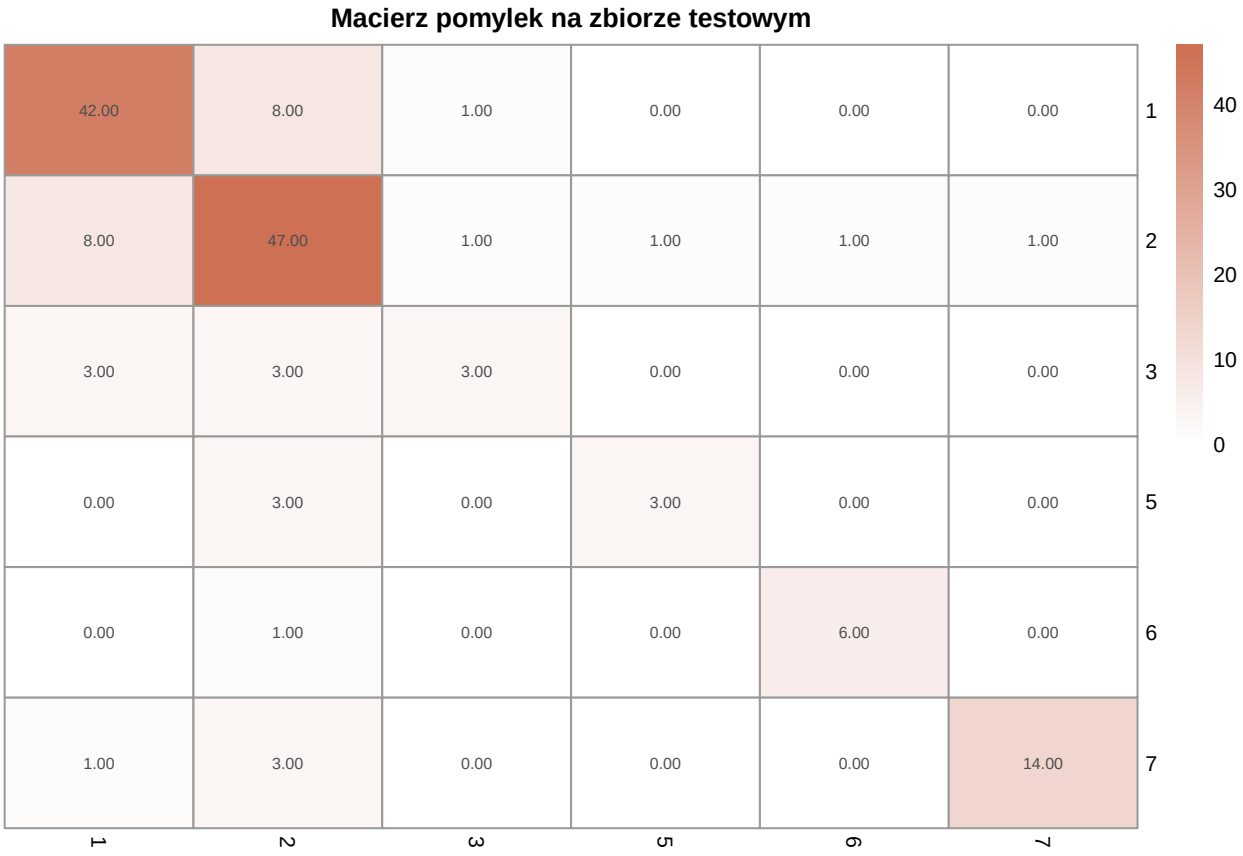
Rysunek 50: Wykres przedstawiający błąd klasyfikacji na zbiorze testowym OOB w zależności od liczby drzew. Czarną linią zaznaczono ogólny błąd dla naszego optymalnego modelu random forest, natomiast kolorowymi dla błędy poszczególnych klas.

Na podstawie wykresu 50 widać, że im więcej drzew ma nasz model, tym ogólny błąd na zbiorze testowym OOB (ciągła czarna linia) jest stabilniejszy i ma tendencję malejącą. W przypadku błędów dla poszczególnych klas (kolorowe linie i przerywana czarna) dla większości z nich mamy zależność, że im więcej drzew, tym stabilniejszy błąd.

Tablica 11: Tabelka przedstawiająca błędy klasyfikacji dla poszczególnych klas dla zbioru testowego OOB.

	Ostateczny_błąd
1	0.1764706
2	0.2033898
3	0.6666667
5	0.5000000
6	0.1428571
7	0.2222222

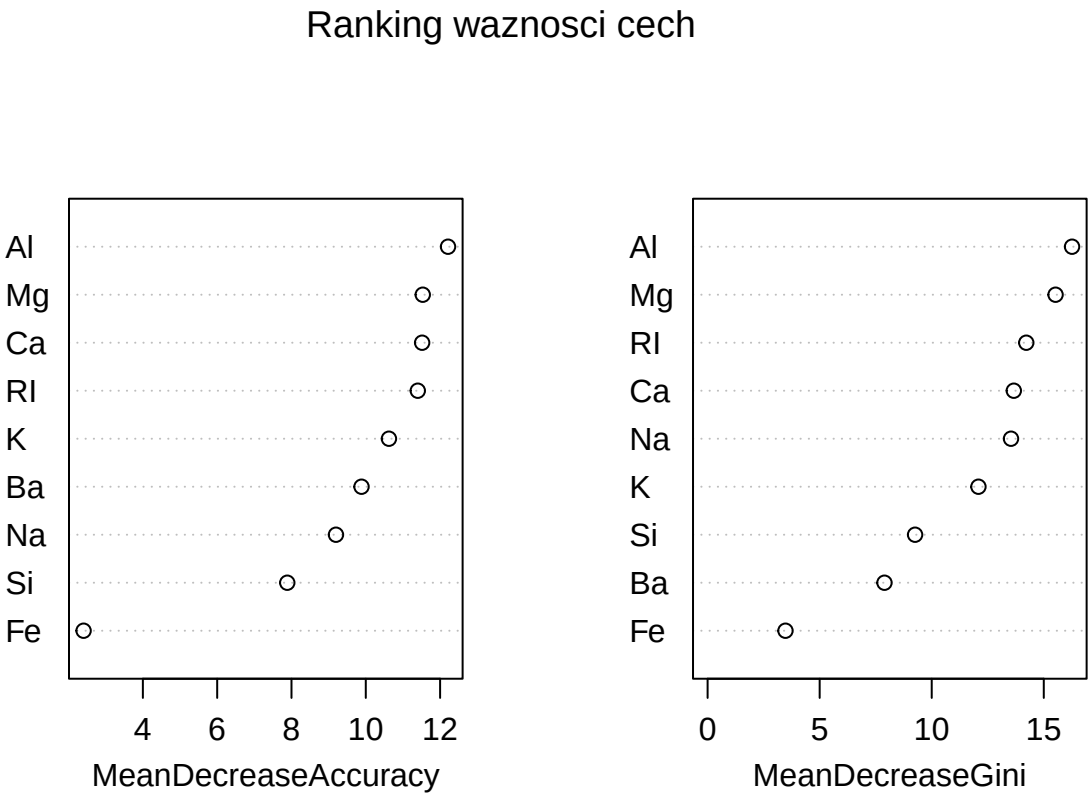
Ostateczny ogólny błąd klasyfikacji na zbiorze testowym OOB: 0.233.



Rysunek 51: Macierz pomyłek na zbiorze testowym OOB dla naszego optymalnego modelu.

4.8.4 Ranking ważności cech

Przyjrzymy się teraz rankingowi ważności cech na wygenerowanym zbiorze uczącym dla optymalnego modelu.



Rysunek 52: Wykres przedstawiający ranking ważności cech w wygenerowanym zbiorze uczącym dla optymalnego modelu.

Wykres 52 bazuje na dwóch miarach ważności cech:

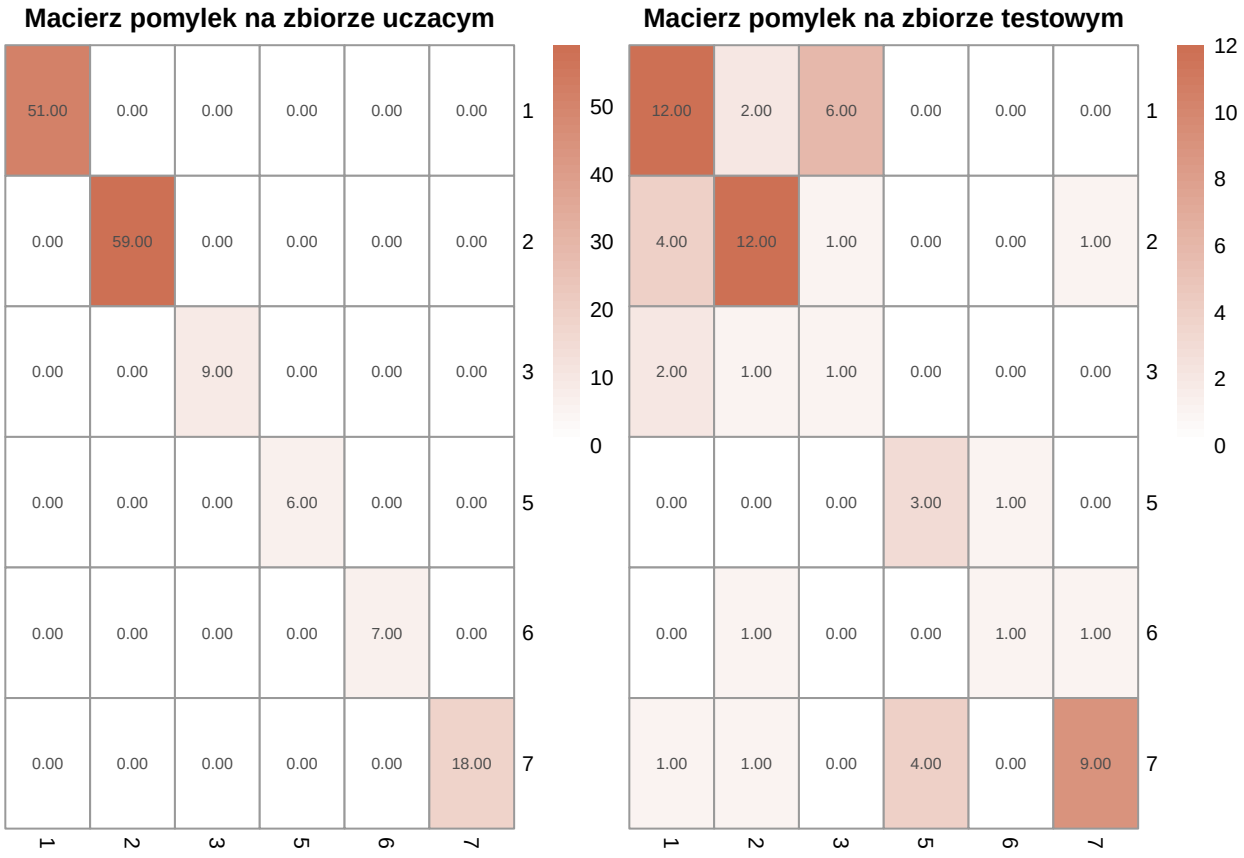
- `MeanDecreaseAccuracy` - miara globalnego wpływu cech na poprawność predykcji mierzona na bazie obserwacji OOB - jak bardzo dokładność spada po losowej permutacji wartości danej cechy.
- `MeanDecreaseGini` - miara czystości podziału - jak bardzo dana cecha pomaga budować drzewo.

Na tej podstawie możmy wnioskować, które cechy najlepiej rozróżniają klasy. Są to:

- `Al`,
- `Mg`,
- `Ca`.

Patrząc na wykres 5 zmienności cech w całym zbiorze danych, może to nie być oczywisty wniosek.

W oparciu o wybrane wyżej cechy i najoptymalniejsze wartości parametrów `mtry` oraz `ntree` spróbujemy wyznaczyć błędy klasyfikacji oraz macierze pomyłek.



Rysunek 53: Macierz pomyłek na zbiorze uczącym (po lewej) i testowym (po prawej) dla wybranych najlepiej rozróżniających klasy cech.

Błąd klasyfikacji na zbiorze uczącym: 0.

Błąd klasyfikacji na zbiorze testowym: 0.406.

4.8.5 Schemat *boostrap 632plus*

Błąd klasyfikacji metodą *bootstrap 632plus* wynosi 0.177.

4.9 Wnioski cząstkowe *random forest*

Tablica 12: Tabelka pozwalająca porównać błędy klasyfikacji dla algorytmu random forest.

Metoda	Błąd_zbiór_uczący	Błąd_zbiór_testowy
Model podstawowy - domyślne parametry	0	0.281
Optymalne parametry ntree i mtry	0	0.250
Optymalne parametry ntree i mtry - ważność	0	0.406
OOB	-	0.233
Schemat bootstrap 632plus	-	0.177

Porównując wszystkie wyznaczone w analizie błędy klasyfikacji (tabela **??**), widzimy, że dla algorytmu *random forest* najlepszy wynik otrzymujemy, wykorzystując Schemat bootstrap 632plus i wynosi on 0.177. Natomiast największy, wynoszący 0.406, wykorzystując Optymalne parametry ntree i mtry - ważność.

Co ciekawe, błąd na zbiorze testowym OOB okazał się gorszy jedynie od błędu wyznaczonego przez Schemat bootstrap 632plus. Ponadto wykorzystanie cech, które według wykresu 52 wydawały się najważniejsze, jako propozycja do zmniejszenia błędu klasyfikacji, okazało się złym pomysłem. Błąd jest znacznie większy od wszystkich pozostałych

5 Wnioski końcowe

Tablica 13: Tabelka pozwalająca porównać wszystkie najmniejsze błędy klasyfikacji i sposoby ich wyznaczenia dla każdego z testowanych algorytów.

Zestaw	Błąd_zbiór_testowy
Pojedyncze drzewo klasyfikacji	0.234
Metoda SVM - Jądro gaussowskie - schemat bootstrap 632plus	0.286
Metoda bagging - Schemat bootstrap 632plus	0.238
Metoda boosting - Schemat bootstrap 632plus	0.179
Random forest - Schemat bootstrap 632plus	0.177

Na podstawie tabelki 14 najmniejszy błąd klasyfikacji otrzymujemy, wykorzystując Random forest - Schemat bootstrap 632plus, 0.177. Ten błąd wynosi 0.177. Jest to zatem najlepsza metoda dla danych `Glass`. Można było się tego spodziewać, ponieważ oba podejścia opierają się na wielokrotnym tworzeniu drzew. Dodatkowo zastosowanie metody *bootstrap 632plus* skutecznie zapobiega przeuczeniu modelu, dzięki czemu jest to algorytm wyjątkowo stabilny i bezpieczny.

Najgorszy z najlepszych okazał się Metoda SVM - Jądro gaussowskie - schemat bootstrap 632plus, wynoszący 0.286.

Tablica 14: Tabelka pozwalająca porównać wszystkie, wcześniej wybrane, największe błędy klasyfikacji i sposoby ich wyznaczenia dla każdego z testowanych algorytów.

Zestaw	Błąd_zbiór_testowy
Pojedyncze drzewo klasyfikacji	0.234
Metoda SVM - Najoptymalniejsze jądro liniowe	0.484
Metoda bagging - Model podstawowy - domyślne parametry	0.297
Metoda boosting - Optymalne parametry cp, minsplit, maxdepth i mfinal	0.312
Random forest - Optymalne parametry ntree i mtry - ważność	0.406

Porównując błędy z najlepszej opcji w każdej metodzie największy błąd klasyfikacji otrzymujemy, wykorzystując Metoda SVM - Najoptymalniejsze jądro liniowe (tabela **??**). Ten błąd wynosi 0.484. Jest to zatem najgorsza metoda dla danych `Glass`.