OPTYMALIZACJA NIELINIOWA

PROJEKT NR 1

SEBASTIAN ZŁOTEK

Opis danych

Dane użyte w projekcie zostały pobrane ze strony kaggle.com, oraz przedstawiają ceny rynkowe cebuli w 2020 roku, jej pochodzenie wraz z stanem, regionem, dystryktem, datą przybycia produktu, typu cebuli oraz ceny maksymalnej, minimalnej i modalnej.

Obróbka danych

Do obróbki danych użyłem języka R oraz środowiska R studio. W pierwszej kolejności wczytałem katalog z danymi, oraz usunąłem notacje wykładniczą, dzięki czemu małe liczby będą łatwiejsze do zapisania w dalszej części kodu.

Dane posortowałem według regionu Gujarat oraz rodzaju cebuli „white” a także zablokowałem duplikaty aby wykres w późniejszym czasie był bardziej przejrzysty.

Do utworzenia wykresu stworzyłem funkcję onion_modal przechowującą kolumnę z ceną modalną cebuli wcześniej już posortowaną według regionu i rodzaju, a także daty bez duplikatów

```r
#Wybranie katalogu glownego
getwd()
setwd("C:/Users/Sebastian/Desktop/studia/III rok 1 semestr/Optymalizacja_nieliniowa_PROJEKT")

#Usuniecie notacji wykladniczej
options(scipen = 999)

#Wczytanie danych
onion <- read.csv("onion2020.csv", sep = ',')

#Sortowanie danych wedlug stanu oraz rodzaju cebuli
onionsorted <- onion[which(onion$state == 'Gujarat' & onion$variety == 'White'),]

#Usuniecie duplikatow dat, dla uzyskania lepszego wykresu
onionsorted <- onionsorted[!duplicated(onionsorted[6]),]

#Utworzenie listy z cenami
onion_modal <- onionsorted$modal_price
```

Obrobiona ramka danych

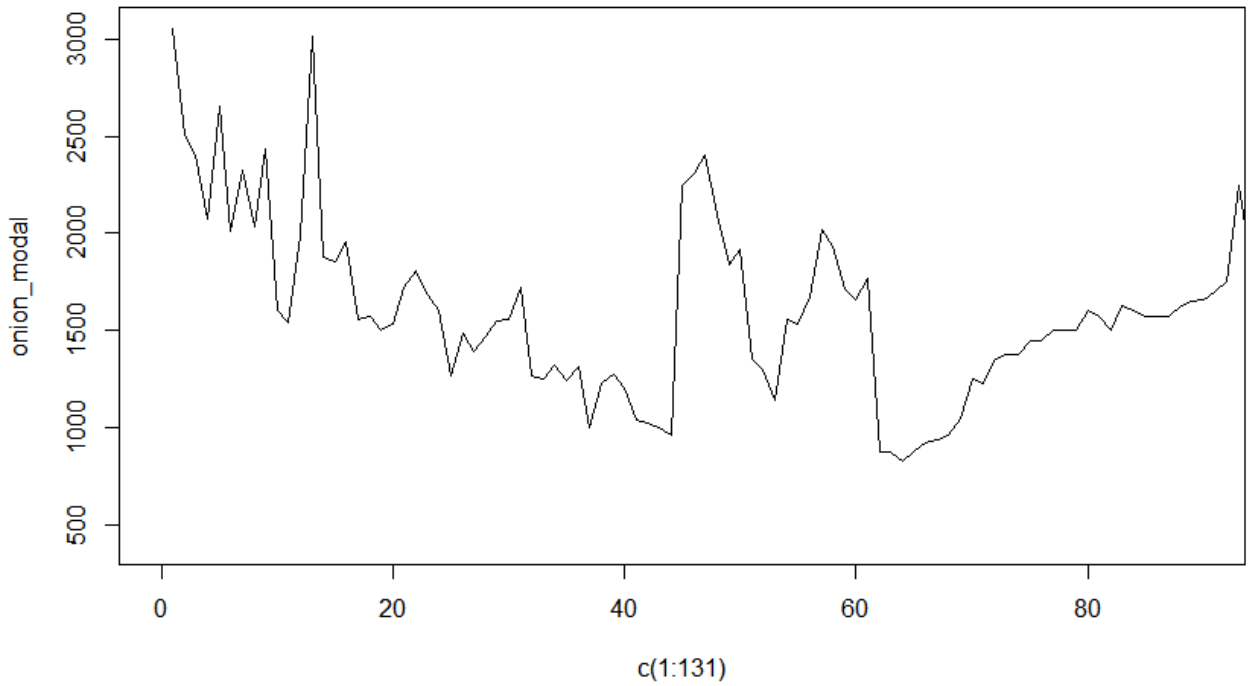
optymalizacja-p1.R* | onion_modal | onionsorted

Filter

	state	district	market	commodity	variety	arrival_date	min_price	max_price	modal_price
1862	Gujarat	Bhavnagar	Bhavnagar	Onion	White	01/01/2020	2755	3355	3055
1863	Gujarat	Bhavnagar	Bhavnagar	Onion	White	02/01/2020	2255	2755	2505
1864	Gujarat	Bhavnagar	Bhavnagar	Onion	White	03/01/2020	2205	2605	2405
1865	Gujarat	Bhavnagar	Bhavnagar	Onion	White	04/01/2020	1900	2255	2075
1866	Gujarat	Bhavnagar	Bhavnagar	Onion	White	06/01/2020	2300	3010	2655
1867	Gujarat	Bhavnagar	Bhavnagar	Onion	White	07/01/2020	1000	3015	2010
1868	Gujarat	Bhavnagar	Bhavnagar	Onion	White	08/01/2020	1500	3150	2325
1869	Gujarat	Bhavnagar	Bhavnagar	Onion	White	09/01/2020	1125	2950	2035
1870	Gujarat	Bhavnagar	Bhavnagar	Onion	White	10/01/2020	2355	2515	2435
1871	Gujarat	Bhavnagar	Bhavnagar	Onion	White	11/01/2020	1250	1950	1600
1872	Gujarat	Bhavnagar	Bhavnagar	Onion	White	13/01/2020	1305	1775	1540
1873	Gujarat	Bhavnagar	Bhavnagar	Onion	White	18/01/2020	1855	2125	1990
1874	Gujarat	Bhavnagar	Bhavnagar	Onion	White	20/01/2020	2825	3205	3015
1875	Gujarat	Bhavnagar	Bhavnagar	Onion	White	21/01/2020	1000	2760	1880
1876	Gujarat	Bhavnagar	Bhavnagar	Onion	White	22/01/2020	1175	2535	1855
1877	Gujarat	Bhavnagar	Bhavnagar	Onion	White	24/01/2020	1450	2475	1960
1878	Gujarat	Bhavnagar	Bhavnagar	Onion	White	25/01/2020	1060	2050	1555
1879	Gujarat	Bhavnagar	Bhavnagar	Onion	White	27/01/2020	1305	1845	1575
1880	Gujarat	Bhavnagar	Bhavnagar	Onion	White	28/01/2020	1005	2005	1505
1881	Gujarat	Bhavnagar	Bhavnagar	Onion	White	31/01/2020	1025	2035	1530
1882	Gujarat	Bhavnagar	Bhavnagar	Onion	White	01/02/2020	1250	2205	1725
1883	Gujarat	Bhavnagar	Bhavnagar	Onion	White	03/02/2020	1500	2110	1805

Showing 1 to 22 of 131 entries, 9 total columns

Dane opisuje wykres



Gdzie na osi y opisana jest cena modalna cebuli, natomiast na x wektor danych od 1 do 131 (liczba wierszy w kolumnie)

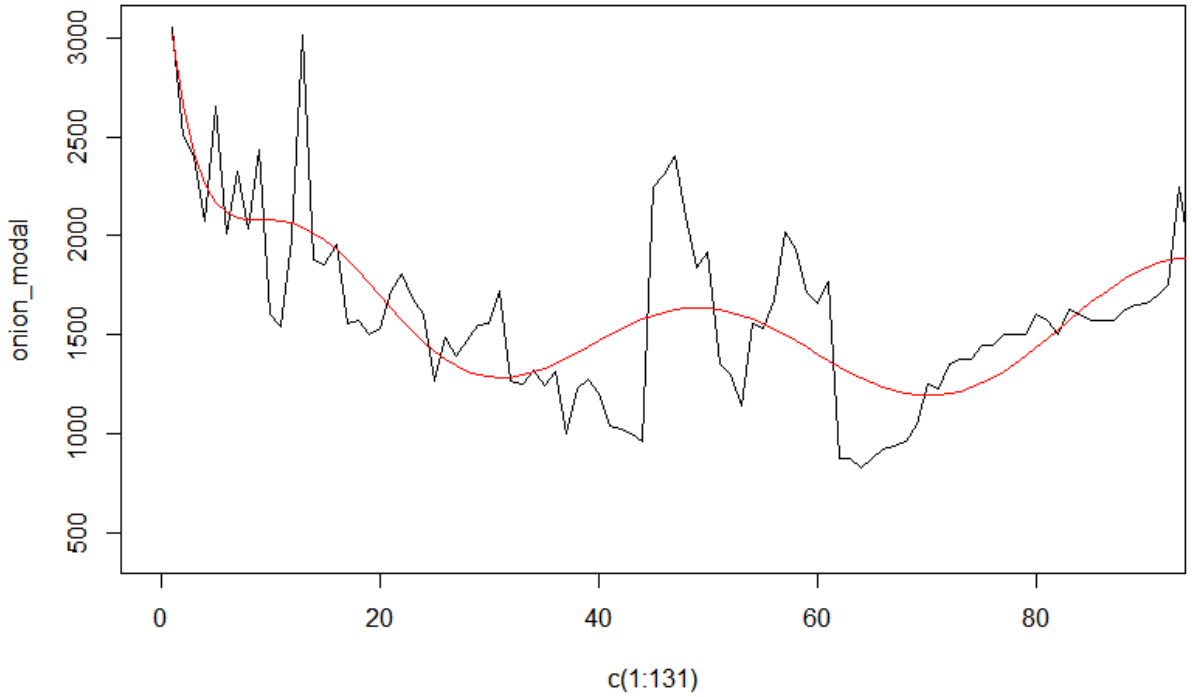
Tworzenie regresji

Tworze ramkę danych o długości równej ilości wierszy w onion_modal, dzięki czemu w następnej linijce kodu mam możliwość stworzenia linii regresji, której potęga przy wartości x wynosi 10, do tego używam funkcji lm oraz poly, które jako gotowe funkcje matematyczne wbudowane w R dają możliwość użycia ramki danych i potęgi do utworzenia wcześniej wspomnianej regresji.

Tworzę także sekwencje danych od 1 do 131 pomagającą funkcji „line” w umieszczeniu linii regresji na wykresie (kolor czerwony).

```
#Wykres cen
plot( x = c(1:131), y=onion_modal, col = "black", type = "l", xlim = c(0,90))

#Utworzenie ramki danych
onion_data <- data.frame( x = c(1:131), y = onion_modal )

#Utworzenie linii regresji 10 - potega przy x
onion_reg10 <- lm( y~poly( x, 10, raw = TRUE ), data = onion_data )

#Sekwencja danych
onionseq <- seq( 1, 131, length = 131 )

#Wyświetlenie linii regresji na wykresie (kolor czerwony)
lines( onionseq, predict( onion_reg10, data.frame(x = onionseq) ), col = 'red' )
```

Linia regresji na wykresie



Opis regresji oraz podobieństwo

Funkcje summary() i summary()$adj.r.squared opisują kolejno:

- regresję jako całość, dzięki czemu możemy odczytać wartości do naszej funkcji z kolumny „Estimate”,
- obliczyć procent podobieństwa regresji do danych, który w moim przypadku wynosi 0.5850153 co oznacza, że regresja do danych jest dopasowana jeszcze w odpowiednim stopniu.

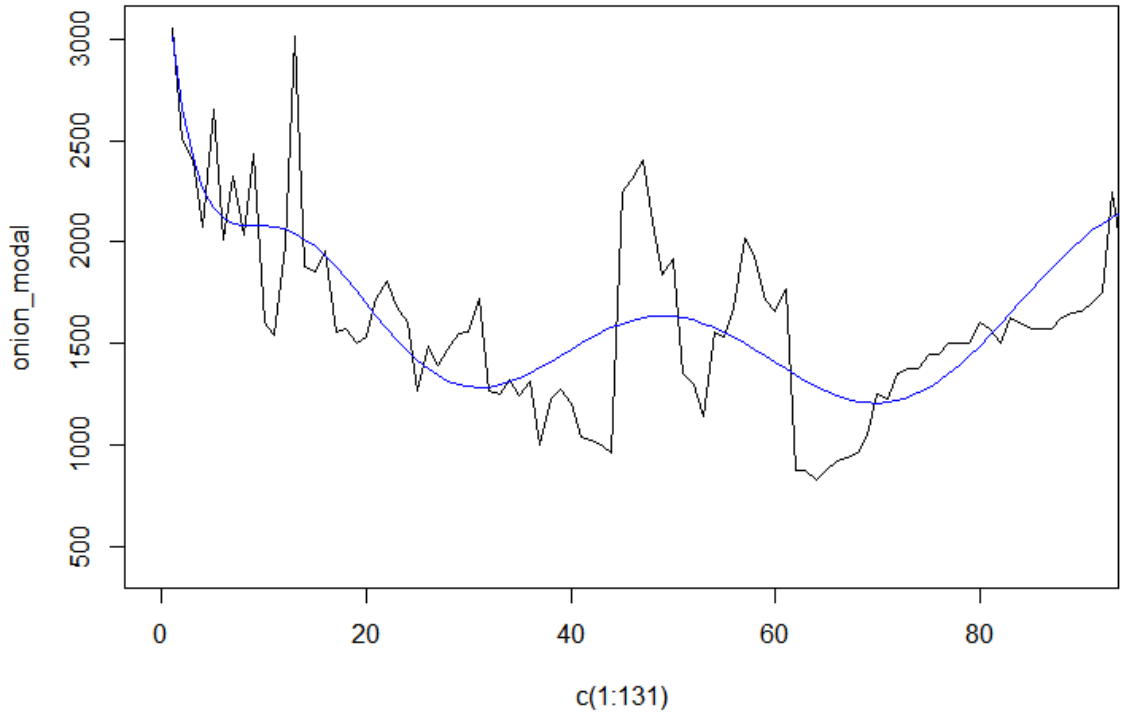
```
> #Obliczenie procentu podobienstwa regresji do danych
> summary(onion_reg10)$adj.r.squared
[1] 0.5850153
> #Opis regresji
> summary(onion_reg10)

Call:
lm(formula = y ~ poly(x, 10, raw = TRUE), data = onion_data)

Residuals:
    Min      1Q  Median      3Q     Max
-644.63 -171.20  -21.31  161.61  971.49

Coefficients:
                              Estimate             Std. Error t value             Pr(>|t|)
(Intercept)              3520.66787771067311041  368.58767145204359394   9.552 < 0.0000000000000002 ***
poly(x, 10, raw = TRUE)1  -590.76471308716702424  182.80568005324195724  -3.232              0.00159 **
poly(x, 10, raw = TRUE)2    94.61588510468364177   30.09630183764609157   3.144              0.00210 **
poly(x, 10, raw = TRUE)3    -7.52946661137093365    2.34022794018264602  -3.217              0.00166 **
poly(x, 10, raw = TRUE)4     0.32364959235614527    0.10109114407669079   3.202              0.00175 **
poly(x, 10, raw = TRUE)5    -0.00814851318300418    0.00263530324209718  -3.092              0.00247 **
poly(x, 10, raw = TRUE)6     0.00012566485574807    0.00004310602855516   2.915              0.00424 **
poly(x, 10, raw = TRUE)7    -0.00000120000276833    0.00000044510885619  -2.696              0.00803 **
poly(x, 10, raw = TRUE)8     0.00000000690730167    0.00000000281542171   2.453              0.01559 *
poly(x, 10, raw = TRUE)9    -0.00000000002191333    0.00000000000995473  -2.201              0.02963 *
poly(x, 10, raw = TRUE)10    0.00000000000002937    0.00000000000001506   1.949              0.05359 .
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 299.7 on 120 degrees of freedom
Multiple R-squared:  0.6169,    Adjusted R-squared:  0.585
F-statistic: 19.33 on 10 and 120 DF,  p-value: < 0.00000000000000022

>
```

Funkcja główna

Z wspomnianej kolumny Estimate z funkcji summary zostały otrzymane współczynniki do utworzenia funkcji, której będziemy używać w algorytmach.

```
#Funkcja utworzona z summary regresji
defonion <- function(x){ (0.000000000000002937) * (x^10) + (-0.00000000002191333) * (x^9) + (0.000000006907301 67)*(x^8)+
    (-0.00000120000276833) * (x^7) + (0.00012566485574807) * (x^6) + (-0.00814851318300418) * (x^5) + (0.32364959235614527) *(x^4) +
(-7.52946661137093365) * (x^3) + (94.61588510468364177) * (x^2) + ((-590.76471308716702424) * x) + (3520.66787771067311041)
}
```

Wykres funkcji



Algorytm złotego podziału

Idea metody opiera się na założeniu, że w każdym kroku obliczeń długość przedziału zmniejszana jest w stałym stosunku równym α, gdzie

$$\alpha = \frac{\sqrt{5}-1}{2} \approx 0.6180339 \quad \text{(złota liczba)}$$

Złota liczba jest to dodatnia liczba niewymierna będąca rozwiązaniem równania

$$x^2 - x - 1 = 0$$ Równa:

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180339$$

Czasami złotą liczbą określa się też liczbę odwrotną

Algorytm złotego podziału

Ciągła i unimodalna funkcja f : R → R dla której szukamy minimum w przedziale [a, b].
W startowej (zerowej) iteracji przyjmujemy:

$$a^{(0)} = a \text{ i } b^{(0)} = b$$

a następnie wybieramy dwa różne punkty $c^{(0)}$, $d^{(0)} \in [a^{(0)}, b^{(0)}]$ takie, że $c^{(0)} < d^{(0)}$

Przedział poszukiwań ma zmniejszać się, z iteracji na iterację, w stałym stosunku α.
Oznacza to, że w każdej iteracji, wybrane punkty pośrednie muszą dzielić i-ty przedział tak, aby spełniony był warunek.

$$\frac{b^{(i)} - c^{(i)}}{b^{(i)} - a^{(i)}} = \frac{d^{(i)} - a^{(i)}}{b^{(i)} - a^{(i)}} = \alpha.$$

Algorytm złotego podziału

Z ostatniego warunku możemy wyznaczyć wartości punktów $c^{(i)}$ i $d^{(i)}$ w każdej iteracji używając wzoru

$$c^{(i)} = b^{(i)} - \alpha\left(b^{(i)} - a^{(i)}\right) = \alpha a^{(i)} + (1-\alpha)b^{(i)},$$
$$d^{(i)} = a^{(i)} + \alpha\left(b^{(i)} - a^{(i)}\right) = (1-\alpha)a^{(i)} + \alpha b^{(i)}.$$

Algorytm złotego podziału – min/max

Jeżeli w i-tej iteracji zachodzi:

$$f(c^{(i)}) < f(d^{(i)})$$

Wówczas w następnej iteracji poszukiwania będą prowadzone w przedziale [$a^{(i)}$ i $d^{(i)}$].
Będziemy wtedy mieli:

$$\begin{aligned}
a^{(i+1)} &= a^{(i)}, \\
b^{(i+1)} &= d^{(i)}, \\
c^{(i+1)} &= \alpha a^{(i+1)} + (1-\alpha) b^{(i+1)}, \\
d^{(i+1)} &= (1-\alpha) a^{(i+1)} + \alpha b^{(i+1)} = (1-\alpha) a^{(i)} + \alpha d^{(i)} = \cdots = c^{(i)}.
\end{aligned}$$

Jak widać w (i + 1)-szej iteracji jest potrzeba wyznaczenia tylko jednego nowego punktu, tzn. c (i+1). Pozostałe punkty znamy z poprzedniej iteracji.

Algorytm złotego podziału – min/max

Analogiczna sytuacja ma miejsce, jeżeli w i-tej iteracji zachodzi:

$$f(c^{(i)}) > f(d^{(i)})$$

Wtedy w następnej iteracji poszukiwania będą prowadzone w przedziale $[c^{(i)}, b^{(i)}]$. Będziemy wtedy mieli

$$\begin{aligned}
a^{(i+1)} &= c^{(i)}, \\
b^{(i+1)} &= b^{(i)}, \\
c^{(i+1)} &= \alpha a^{(i+1)} + (1-\alpha) b^{(i+1)} = \alpha c^{(i)} + (1-\alpha) b^{(i)} = \cdots = d^{(i)}, \\
d^{(i+1)} &= (1-\alpha) a^{(i+1)} + \alpha b^{(i+1)}.
\end{aligned}$$

W tym przypadku w (i + 1)-szej iteracji jest również potrzeba wyznaczenia tylko jednego nowego punktu, tzn. d (i+1) . Pozostałe punkty są już znane z poprzedniej iteracji.

Algorytm złotego podziału – minimum

```r
#algorytm golden podstawowy - minimum
golden <- function(f,lower, upper, tol){
  czas1 <- Sys.time()
  alpha <- (sqrt(5)-1)/2
  x1 <- alpha * lower + (1-alpha) * upper
  f.x1 <- f(x1)
  x2 <- (1-alpha) * lower + alpha * upper
  f.x2 <- f(x2)

  i<-0
  cat("iteracja:",i,"a=",lower,"b=",upper,"b-a=",upper-lower,"\n")

  while(abs(upper - lower) > 2 * tol){
    i<-i+1
    if (f.x1 < f.x2){
      upper <- x2
      x2 <- x1
      f.x2 <- f.x1
      x1 <- alpha * lower + (1-alpha) * upper
      f.x1 <- f(x1)
    } else {
      lower <- x1
      x1 <- x2
      f.x1 <- f.x2
      x2 <- (1-alpha) * lower + alpha * upper
      f.x2 <- f(x2)
    }
    cat("iteracja:",i,"a=",lower,"b=",upper,"b-a=",upper-lower,"\n")
  }
  czas2 <- Sys.time()
  cat("Czas działania: ",czas2-czas1,"\n")
  return((lower + upper) / 2)
}
```

Kod w RStudio

```
> #wartość minimalna golden
> golden(defonion,22,38,0.000001)
iteracja: 0 a= 22 b= 38 b-a= 16
iteracja: 1 a= 28.11146 b= 38 b-a= 9.888544
iteracja: 2 a= 28.11146 b= 34.22291 b-a= 6.111456
iteracja: 3 a= 28.11146 b= 31.88854 b-a= 3.777088
iteracja: 4 a= 29.55418 b= 31.88854 b-a= 2.334369
iteracja: 5 a= 30.44582 b= 31.88854 b-a= 1.442719
iteracja: 6 a= 30.44582 b= 31.33747 b-a= 0.8916494
iteracja: 7 a= 30.7864 b= 31.33747 b-a= 0.5510697
iteracja: 8 a= 30.7864 b= 31.12698 b-a= 0.3405798
iteracja: 9 a= 30.91649 b= 31.12698 b-a= 0.2104899
iteracja: 10 a= 30.99689 b= 31.12698 b-a= 0.1300899
iteracja: 11 a= 30.99689 b= 31.07729 b-a= 0.08039998
iteracja: 12 a= 31.0276 b= 31.07729 b-a= 0.04968992
iteracja: 13 a= 31.04658 b= 31.07729 b-a= 0.03071006
iteracja: 14 a= 31.04658 b= 31.06556 b-a= 0.01897986
iteracja: 15 a= 31.04658 b= 31.05831 b-a= 0.0117302
iteracja: 16 a= 31.05106 b= 31.05831 b-a= 0.007249662
iteracja: 17 a= 31.05383 b= 31.05831 b-a= 0.004480537
iteracja: 18 a= 31.05383 b= 31.0566 b-a= 0.002769124
iteracja: 19 a= 31.05489 b= 31.0566 b-a= 0.001711413
iteracja: 20 a= 31.05555 b= 31.0566 b-a= 0.001057711
iteracja: 21 a= 31.05555 b= 31.0562 b-a= 0.0006537016
iteracja: 22 a= 31.0558 b= 31.0562 b-a= 0.0004040098
iteracja: 23 a= 31.05595 b= 31.0562 b-a= 0.0002496918
iteracja: 24 a= 31.05595 b= 31.0561 b-a= 0.000154318
iteracja: 25 a= 31.05601 b= 31.0561 b-a= 0.00009537378
iteracja: 26 a= 31.05601 b= 31.05607 b-a= 0.00005894424
iteracja: 27 a= 31.05603 b= 31.05607 b-a= 0.00003642954
iteracja: 28 a= 31.05604 b= 31.05607 b-a= 0.00002251469
iteracja: 29 a= 31.05604 b= 31.05606 b-a= 0.00001391485
iteracja: 30 a= 31.05604 b= 31.05605 b-a= 0.000008599848
iteracja: 31 a= 31.05605 b= 31.05605 b-a= 0.000005314998
iteracja: 32 a= 31.05605 b= 31.05605 b-a= 0.00000328485
iteracja: 33 a= 31.05605 b= 31.05605 b-a= 0.000002030149
iteracja: 34 a= 31.05605 b= 31.05605 b-a= 0.000001254701
Czas działania:  0.01204109
[1] 31.05605
>
```

Otrzymane minimum na przedziale [22;38]
Wynosi 31.05605 i wymagało 34 iteracji

Algorytm złotego podziału – maximum

```r
#Algorytm golden podstawowy - maximum
golden_max <- function(f,lower, upper, tol){
  czas1 <- Sys.time()
  alpha <- (sqrt(5)-1)/2
  x1 <- alpha * lower + (1-alpha) * upper
  f.x1 <- f(x1)
  x2 <- (1-alpha) * lower + alpha * upper
  f.x2 <- f(x2)

  i<-0
  cat("iteracja:",i,"a=",lower,"b=",upper,"b-a=",upper-lower,"\n")

  while(abs(upper - lower) > 2 * tol){
    i<-i+1
    if (f.x1 > f.x2){
      upper <- x2
      x2 <- x1
      f.x2 <- f.x1
      x1 <- alpha * lower + (1-alpha) * upper
      f.x1 <- f(x1)
    } else {
      lower <- x1
      x1 <- x2
      f.x1 <- f.x2
      x2 <- (1-alpha) * lower + alpha * upper
      f.x2 <- f(x2)
    }
    cat("iteracja:",i,"a=",lower,"b=",upper,"b-a=",upper-lower,"\n")
  }
  czas2 <- Sys.time()
  cat("Czas działania: ",czas2-czas1,"\n")
  return((lower + upper) / 2)
}
```

Kod w RStudio

```
> #wartosc maksymalna golden
> golden_max(defonion,44,60,0.000001)
iteracja: 0 a= 44 b= 60 b-a= 16
iteracja: 1 a= 44 b= 53.88854 b-a= 9.888544
iteracja: 2 a= 47.77709 b= 53.88854 b-a= 6.111456
iteracja: 3 a= 47.77709 b= 51.55418 b-a= 3.777088
iteracja: 4 a= 47.77709 b= 50.11146 b-a= 2.334369
iteracja: 5 a= 48.66874 b= 50.11146 b-a= 1.442719
iteracja: 6 a= 48.66874 b= 49.56039 b-a= 0.8916494
iteracja: 7 a= 48.66874 b= 49.21981 b-a= 0.5510697
iteracja: 8 a= 48.87923 b= 49.21981 b-a= 0.3405798
iteracja: 9 a= 48.87923 b= 49.08972 b-a= 0.2104899
iteracja: 10 a= 48.95963 b= 49.08972 b-a= 0.1300899
iteracja: 11 a= 48.95963 b= 49.04003 b-a= 0.08039998
iteracja: 12 a= 48.99034 b= 49.04003 b-a= 0.04968992
iteracja: 13 a= 48.99034 b= 49.02105 b-a= 0.03071006
iteracja: 14 a= 49.00207 b= 49.02105 b-a= 0.01897986
iteracja: 15 a= 49.00207 b= 49.0138 b-a= 0.0117302
iteracja: 16 a= 49.00655 b= 49.0138 b-a= 0.007249662
iteracja: 17 a= 49.00932 b= 49.0138 b-a= 0.004480537
iteracja: 18 a= 49.00932 b= 49.01209 b-a= 0.002769124
iteracja: 19 a= 49.00932 b= 49.01103 b-a= 0.001711413
iteracja: 20 a= 49.00997 b= 49.01103 b-a= 0.001057711
iteracja: 21 a= 49.00997 b= 49.01062 b-a= 0.0006537016
iteracja: 22 a= 49.00997 b= 49.01037 b-a= 0.0004040098
iteracja: 23 a= 49.01012 b= 49.01037 b-a= 0.0002496918
iteracja: 24 a= 49.01012 b= 49.01028 b-a= 0.000154318
iteracja: 25 a= 49.01012 b= 49.01022 b-a= 0.00009537378
iteracja: 26 a= 49.01016 b= 49.01022 b-a= 0.00005894424
iteracja: 27 a= 49.01016 b= 49.0102 b-a= 0.00003642954
iteracja: 28 a= 49.01016 b= 49.01018 b-a= 0.00002251469
iteracja: 29 a= 49.01017 b= 49.01018 b-a= 0.00001391485
iteracja: 30 a= 49.01017 b= 49.01018 b-a= 0.000008599848
iteracja: 31 a= 49.01017 b= 49.01018 b-a= 0.000005314998
iteracja: 32 a= 49.01017 b= 49.01018 b-a= 0.00000328485
iteracja: 33 a= 49.01017 b= 49.01018 b-a= 0.000002030149
iteracja: 34 a= 49.01017 b= 49.01018 b-a= 0.000001254701
Czas działania:  0.006947041
[1] 49.01018
```

Otrzymane maximum na przedziale [44;60]
Wynosi 49.01018 i wymagało 34 iteracji

Metoda siecznych

Metoda siecznych stanowi modyfikację metody Newtona pozwalająca na uniknięcie konieczności wyznaczania drugiej pochodnej.

Niech dana będzie różniczkowalna i unimodalna funkcja f : R → R dla której szukamy minimum w przedziale [a, b]. Jeżeli założymy, że została wykonana przynajmniej jedna iteracja algorytmu, to dysponujemy dwoma punktami $x^{(i)}$ oraz $x^{(i-1)}$, a także wartościami pochodnej w tych punktach. Możemy zatem przybliżyć f''($x^{(i)}$) za pomocą współczynnika kierunkowego siecznej przechodzącej przez punkty ($x^{(i)}$, f'($x^{(i)}$)) oraz ($x^{(i-1)}$, f'($x^{(i-1)}$)).

Metoda siecznych

Mamy

$$f''(x^{(i)}) \approx \frac{f'(x^{(i)}) - f'(x^{(i-1)})}{x^{(i)} - x^{(i-1)}},$$

zatem w algorytmie Newtona, w i-tej iteracji, wzór na przybliżenie szukanego minimum przyjmuje postać

$$x^{(i+1)} = x^{(i)} - f'(x^{(i)}) \frac{x^{(i)} - x^{(i-1)}}{f'(x^{(i)}) - f'(x^{(i-1)})}.$$

Metoda siecznych w procesie optymalizacji wykorzystuje tylko pierwszą pochodną, ale do rozpoczęcia optymalizacji potrzebuje dwóch punktów startowych.

Metoda siecznych

Zatem liczmy pochodną naszej funkcji i używamy algorytmu:

```
dfdefonion <- function(x){ (0.0000000000002937) * (x^9) + (-0.00000000019722) * (x^8) + (0.0000000552584)*(x^7)+
    (-0.00000840001937831) * (x^6) + (0.00075398913448842) * (x^5) + (-0.0407425659150209) * (x^4) + (1.2945983694245811) *(x^3) +
(-22.5883998341128010) * (x^2) + (189.2317702093672835) * (x) - 590.76471308716702424
}
```

Metoda siecznych

```r
#Algorytm siecznych - minimum
secant <- function(df, x1, x2, tol) {
  i <- 0
  cat("Iteracja: ",i,"x=",x1,"\n")
  df.x2 <- df(x2)
  repeat {
    i<-i+1
    df.x1 <- df(x1)
    new.x <- x1 - df.x1 * (x1 - x2) / (df.x1 - df.x2)

    cat("Iteracja: ",i,"x=",x1,"krok=",- df.x1 * (x1 - x2) / (df.x1 - df.x2),"\n")

    if (abs(new.x - x1) < tol) {
      return(new.x)
    }
    x2 <- x1
    df.x2 <- df.x1
    x1 <- new.x
  }
}
```

Kod w RStudio

```
> #Wartosc minimalna z uzyciem algorytmu siecznych
> secant(dfdefonion,28,38,0.000001)
Iteracja:  0 x= 28
Iteracja:  1 x= 28 krok= 4.29939
Iteracja:  2 x= 32.29939 krok= -1.13826
Iteracja:  3 x= 31.16113 krok= -0.1106383
Iteracja:  4 x= 31.05049 krok= 0.005639795
Iteracja:  5 x= 31.05613 krok= -0.00002158958
Iteracja:  6 x= 31.05611 krok= -0.000000004379006
[1] 31.05611
```

Otrzymane maximum na przedziale [28;38]
Wynosi 31.05611 i wymagało 6 iteracji

Wnioski

Ilość wymaganych iteracji różni się w zależności od wybranego algorytmu, w przypadku algorytmów golden (złoty podział) i secant (siecznych) lepiej działa drugi, podając wynik po 6 iteracjach czyli ponad 5 razy mniej niż 34 iteracje algorytmu golden. W obu metodach otrzymaliśmy skutecznie obliczone, podobne wyniki minimum, ponadto algorytm złotego podziału obliczył także maksiumum, również wykonując 34 iteracje.

Dane

https://www.kaggle.com/datasets/ravisane1/market-price-of-onion-2020